UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA
DE SISTEMAS ELECTRÓNICOS

TRABAJO FIN DE MÁSTER

Performance Optimisation of Variable Precision Digital Signal Processing Algorithms

IGNACIO AMAT HERNÁNDEZ

2023

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS ELECTRÓNICOS

TRABAJO FIN DE MÁSTER

Título: **Performance Optimisation of Variable Precision Digital Signal Processing Algorithms**

Autor: **Ignacio Amat Hernández**

Tutor: **Dr. Juan Antonio López Martín**

Departamento: **Ingeniería Electrónica**

MIEMBROS DEL TRIBUNAL

Presidente: D. ……

Vocal: D. ……..

Secretario: D. ……..

Suplente: D. ……..

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: ……

Madrid, a de de 20…

A thesis submitted in fulfillment of the requirements for the degree of

Master in Electronic Systems Engineering

Performance Optimisation

of

Variable Precision

Digital Signal Processing

Algorithms

Tutor: Dr. Juan Antonio López Martín

Author: Ignacio Amat Hernández

Escuela Técnica Superior de Ingeniería de Telecomunicación

Universidad Politécnica de Madrid

2023

ABSTRACT

In this Thesis we pursue a quantitative and exhaustive design effort for the logic architecture of various common Digital Signal Processing (DSP) Algorithms. First, we investigate digital architecture implementations for digital Proportional Integral Derivative (PID) controllers with precision ranging from low to high (up to 96 bit width) implemented on Xilinx Field Programmable Gate Arrays (FPGAs) with varying capabilities (Artix, Kintex, Virtex; Ultrascale, Ultrascale+), while attempting to achieve the maximum possible clock rates. We seek to develop our ability to efficiently implement systems with pipeline and feedback loops on FPGA devices that make use DSP slices (Digital Signal Processing Arithmetic Logic Units) and distributed logic simultaneously.

We take this analysis further by evaluating generic matrix multiplications of arbitrary size. We optimise our designs by minimising resources using only DSP slices and achieving maximum clock rates. Afterwards, we adapt our designs to complex numbers allowing for the parallel computation of complex matrix multiplications of generic size with variable precisions. We proceed analysing different strategies for efficient complex number multiplication.

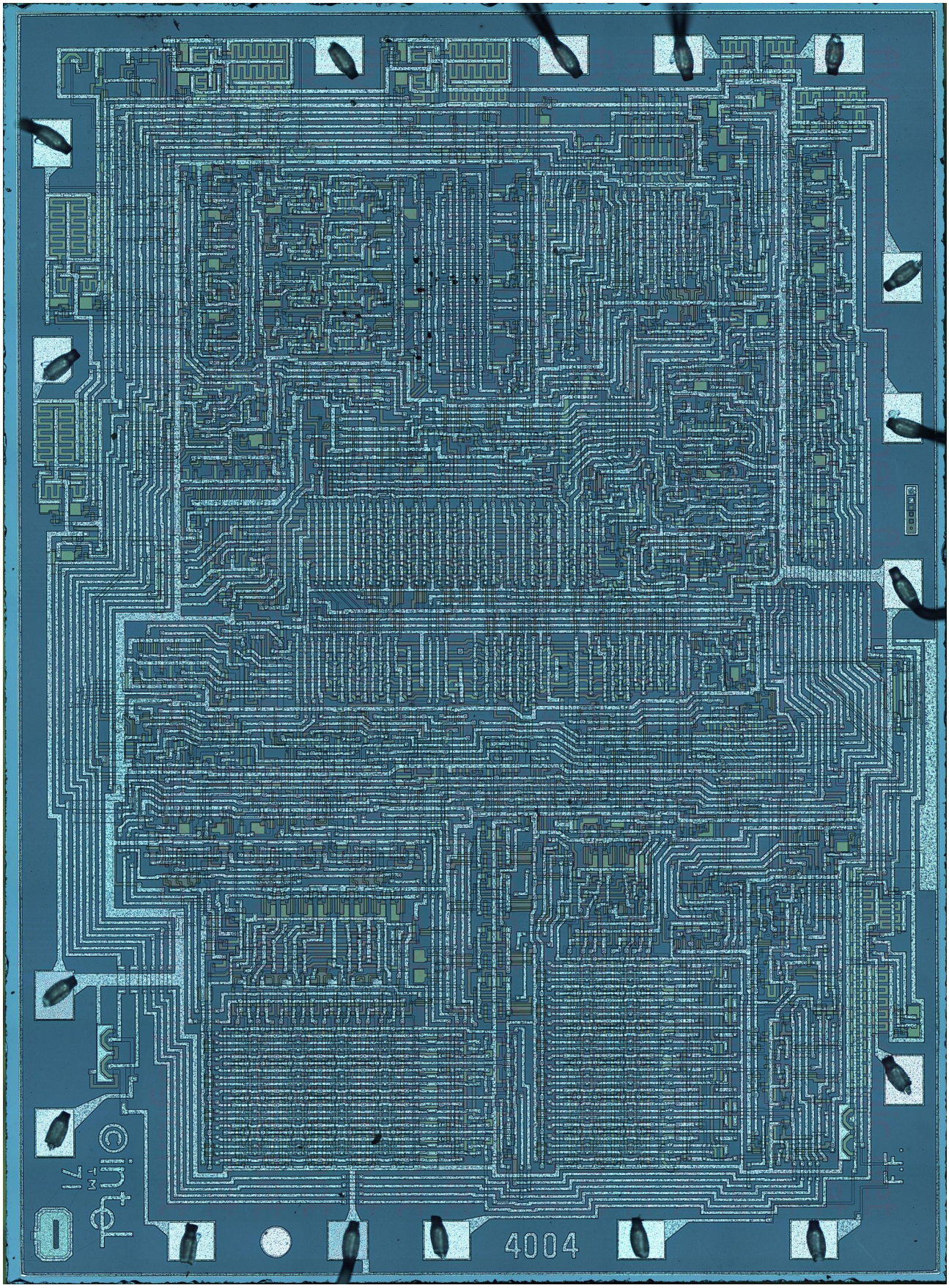
Finally, we apply the complex-matrix multiplication algorithm to generate a recursive implementation of the Fast Fourier Transform (FFT). The previous results allow us to develop efficient FPGA implementations of parallel FFTs for a generic (power-of-r) number of points. Thanks to this example we also develop our understanding and expertise of optimising the Look Up Tables (LUTs) as distributed-logic-only implementations, as well as deepening our understanding of the synthesis and implementation tools.

RESUMEN

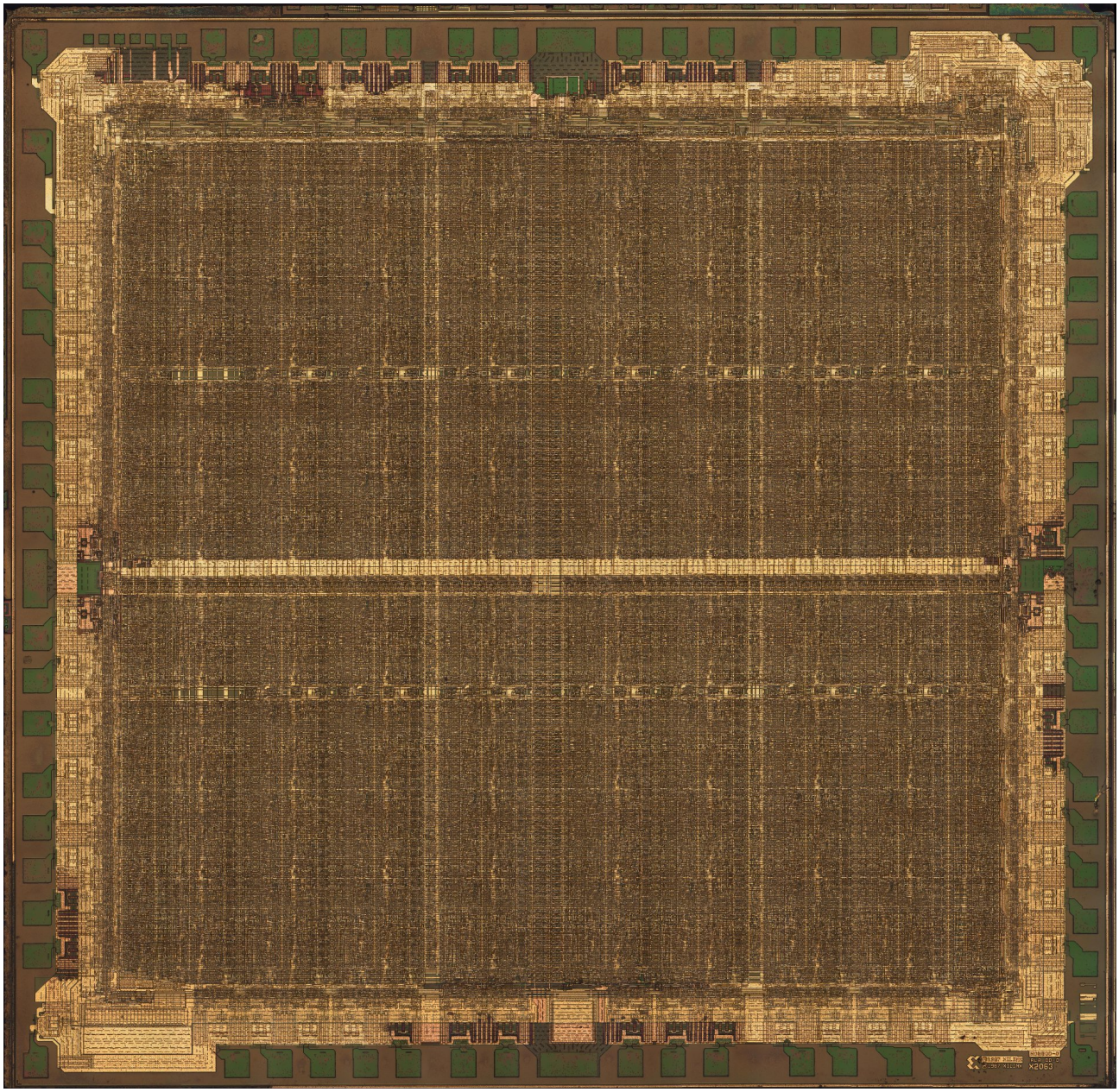
En este trabajo realizamos un esfuerzo de diseño cuantitativo y exhaustivo para la arquitectura lógica de varios algoritmos comunes de Procesamiento de Señales Digitales (DSP). Primero, investigamos implementaciones de arquitectura digital para controladores Proporcionales Integrales Derivativos (PID) con precisión que varía desde bajas a altas (hasta 96 bits de ancho) implementados en Field Programmable Gate Arrays (FPGAs) con distintas capacidades (Artix, Kintex, Virtex; Ultrascale, Ultrascale+), intentando alcanzar la máxima frecuencia de reloj posible. Con ello pretendemos desarrollar nuestra capacidad para implementar correctamente y hacer *pipeline* a sistemas con bucles de realimentación e implementaciones FPGA mixtas, diseños implementados con *slices* DSP (Unidades aritmético-lógicas para el procesado digital de señales) y lógica distribuida lógica distribuida simultáneamente.

Profundizamos en nuestro estudio proponiendo multiplicadores matriciales genéricos de tamaño arbitrario. Nos proponemos optimizar nuestros diseños minimizando los recursos y alcanzando la máxima velocidad de reloj utilizando únicamente multiplicadores DSP. Posteriormente, adaptamos nuestros diseños a números complejos, lo que permite el cálculo paralelo de multiplicaciones de matrices complejas de tamaño genérico con precisiones variables. A continuación analizamos tres posibles estrategias para la multiplicación de números complejos (en un solo DSP, o con tres o cuatro multiplicadores).

Por último, aplicamos el algoritmo de multiplicación de matrices complejas para generar una implementación recursiva de la Fast Fourier Transform (FFT). Los resultados anteriores nos permiten desarrollar implementaciones FPGA eficientes de FFT paralelas para un número genérico (potencia-de-r) de puntos. Gracias a este ejemplo también desarrollamos nuestra comprensión y experiencia en la optimización el uso de Look Up Tables (LUTs) como implementaciones de lógica distribuida, así como profundizar en nuestro conocimiento de las herramientas de síntesis e implementación.



Intel 4004 [1] microphotography of the first microprocessor taken from [2].



XC2064 [3] microphotography of the first FPGA taken from [4].

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER

1

INTRODUCTION

With the breakdown of Dennard scaling around 2005 and with Gordon Moore's correctly prophesied half century of exponential transistor count per chip growth slowing down, a new era in computing is dawning on us. Up until the turn of the 21^{st} century, large strides were made in terms of performance, power consumption and complexity of silicon semiconductor integrated microprocessors, this innovation was mainly driven by process technology improvements. Nowadays we are living through the age of the architect, which began with the multiprocessor revolution, where computer architecture evolved integrating multiple processing cores into a single multicore processor; this has harnessed the potential of parallel computing, inasmuch as compiler and other software limitations have allowed. To continue driving innovation, State-of-the-Art systems have to develop more and more complex instructions such as vector and matrix operations or alternatively offload computationally heavy workloads such as video encoding, deep neural networks, image signal processing, and others to coprocessors integrated on or off chip. These coprocessors almost invariably owe their efficiency and throughput to minimising memory accesses and to massively parallel domain specific architectures, the main focus of this Thesis.

This Thesis will pursue a quantitative and exhaustive design effort for the domain specific logic architecture of various common digital signal processing algorithms. In doing so, we seek to make design considerations and optimisations exploring different aspects of the design space. We understand the design space as the abstract multidimensional region on which certain defining variables characterize the point occupied by the particular configuration of a system.

In general, if we seek to explore different implementations of a particular typical DSP (Digital Signal Processing) system, such as digital filtering, correlation, convolution, adaptive filter, motion estimation, etc., we may for example define the design space with the variables of precision, area and power consumption. According to the

context under which these systems are developed, the defining variables may take on different meanings. If we were pursuing the design a digital Application Specific Integrated Circuit (ASIC), the variable of precision of the system, could be defined by signal-to-noise ratio and modified by altering the number of bits of signals. Area, would refer the amount of physical surface on a silicon wafer necessary to implement the Very Large Scale Integrated (VLSI) circuit, which among others factors, can be altered by deciding on parallel or serial implementations. Power consumption, in ASICs is characterized by the product of the current, voltage, capacitance, switching frequency, etc., necessary to perform a computation, affected by the process node of implementation, logic family style, frequency of operation and others.

The same defining variables could take more nuanced meanings on different contexts, if instead of focusing on ASIC design we were concerned with programmable logic implementations, the area variable could instead be defined as the total number of available primitives (Look Up Tables, Arithmetic Logic Units, Phase Lock Loops, registers, etc.)utilised by our design. Likewise, when considering Microprocessor implementations of DSP algorithms other more appropriate footprint metrics could be used.

In this work we will adopt the principles of quantitative design as applied to digital logic architecture. Our research will follow a systematic approach when analysing empirical observations measured from in-device implementations. Whenever possible, exhaustive comparisons will be made between different configurations to establish the characteristics of the system under test. Conclusions drawn from these experiences will serve to appropriately inform our decisions when evaluating the fitness of a system for a particular purpose.

During our architecture design process investigation we will cover the maximum possible design space, resource and time permitting. Additionally, effort will be put into the clarity, robustness and maintainability of the hardware descriptions developed and accompanying customization scripts. To the furthest extent possible Free and Open Source tools will be employed and developed throughout the production of this work.

The principles above described are demonstrated on three common digital signal processing algorithms, PID (Proportional Integral Derivative) controllers, matrix multipliers and Fast Fourier Transforms. They represent the three main parts of this work, the general objectives set for all three systems are a detailed and comprehensive analysis of the boundaries and limitations of real-world system implementations on current state of the art Field Programmable Gate Array (FPGA) platforms. The conclusions drawn, analysis procedure and developed hardware descriptions could be adapted in future work for the design of ASICs.

First, we will investigate digital architecture implementations for digital PID controllers. As defining variables of our design space we will take precision, ranging from low to high precisions (up to 96-bit widths) implemented on Xilinx FPGAs with varying capabilities (Artix, Kintex, Virtex; Ultrascale, Ultrascale+); performance, attempting to achieve the maximum possible clock rate on each platform; and area, using the least possible amount of FPGA resources. We seek to develop our ability to efficiently implement systems with pipeline and feedback loops on FPGA devices that make use DSP slices and distributed logic simultaneously.

Then we will take our analysis of digital signal processing algorithms further by evaluating generic parallel matrix multipliers of arbitrary size. We plan to optimise our designs by minimising resources using only

DSP slices and achieving maximum clock rates. Afterwards, we will adapt our designs to complex numbers allowing for the parallel computation of complex matrix multiplications of generic size with variable precisions. Afterwards we will analyse different strategies for efficient complex number multiplication by developing a novel approach to low precision 6 bit unsigned complex number multiplication packed into a single DSP multiplier block.

Finally, we will apply the knowledge complex-matrix multiplication algorithm to generate recursive mixed radix implementations of the Fast Fourier Transform. The previous results allow us to develop efficient FPGA implementations of parallel Fast Fourier Transforms (FFTs) for a generic number of points. Our versatile approach when defining the digital architecture enables efficient pipelining to be applied, with multiple configurations available. Thanks to this example we will develop our understanding and expertise of optimising distributed-logic-only designs, as well as deepening our understanding of the synthesis and implementation tools.

With the completion of this work, we aim at the publication of three different papers. A journal paper about the performance of the variable precision generic controller implementations on FPGA devices, a conference/journal article about the implementation of a complex multiplication using a single DSP48 multiplier and a journal article presenting the results of the recursive matrix parallel FFT implementation.

CHAPTER

2

FIELD PROGRAMMABLE GATE ARRAYS

In this section we introduce the principal technologies and working principles of FPGAs. These devices are silicon semiconductor integrated circuits that posses the ability to implement any other logic arbitrary circuit simply by being programmed. Alternative circuit implementation technologies, such as ASICs, require new chips to be fabricated for each new design iteration. In many cases this is prohibitive in terms of cost, slow to reach the market and additionally extensive verification is needed to avoid design errors, which may be difficult or impossible to correct once fabricated.

Programmable technology is the key that reduces non-recurring engineering costs and time-to-market, often times FPGAs are the most cost effective solutions for medium to small production volumes. For large production runs, very simple, or highly performant integrated circuits, the economies of scale play in favour of ASICs, full-custom solutions, dedicated architecture signal processors and others.

The downside of the benefits FPGAs provide comes at the cost of speed and area. Circuits implemented in this medium tend to be an order of magnitude slower and occupy more silicon, this has detrimental effects on power efficiency. To overcome the slower clock rates, in digital signal processing, data throughput can be increased by parallelizing the implemented algorithm to process several samples at the same time. In general, in VLSI systems, efficiency is inversely proportional to flexibility.

FPGAs are used in a variety of areas, including embedded systems (image processing [7], signal processing [8], and motor control [9]), communications (wireless base stations [10], network switches [11, 12, 13], and routers [14, 15]), aerospace (avionics [16], guidance and navigation [17], and flight control systems [18, 19]), automotive (engine control [20], power train control [21], and safety systems [22]), and industrial automation (machine vision [23], motion control [24], and process control [25]) amongst others.

I FPGA Configurable Elements

Configurable Logic Arrays (CLAs) where invented by American Electrical Engineer Ross H. Freeman [26]. As stated in the patent, *configurable logic array*s are comprised of *configurable logic elements* which are *variably interconnected* according to *control signals* in order to *perform a selected logic function*. Functions of logic elements can be varied by changing the control information stored in rewritable internal memory cells – thus granting this device the ability of being fully programmed in the field.

The first commercially available implementation, XC2064 [3], serves to illustrate the working principles common to all devices. Logic Arrays include three main categories of configurable elements as depicted in **Fig.** 2.1, chiefly:

- **I/O Blocks**: which interface with the package pins.
- **Configurable Logic Blocks**: which realise the user programmed logic function.
- **Programmable Interconnect**: that networks logic signals between blocks.

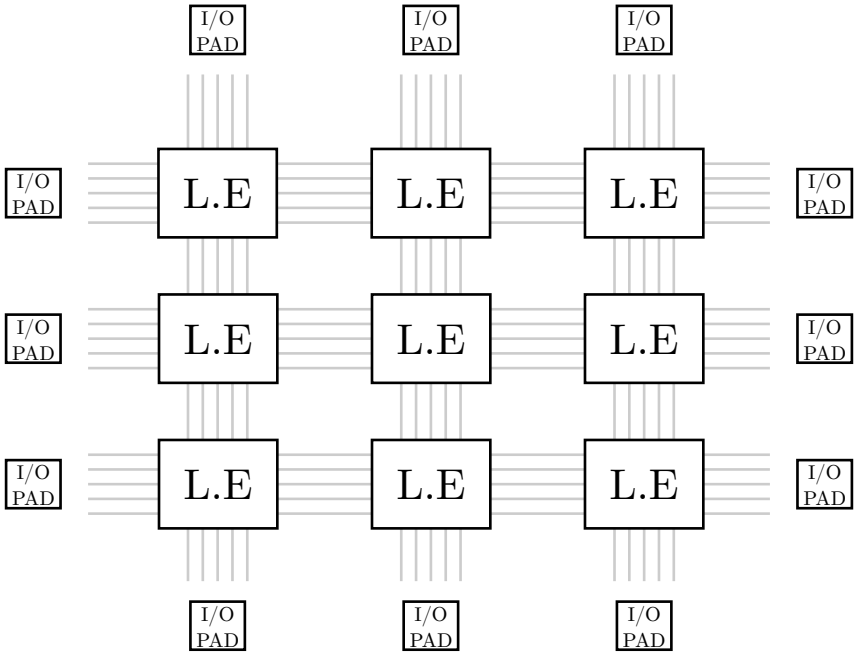


Fig. 2.1: *Configurable Logic Array adapted from XC2064 [3].*

As technology has evolved, functionality, capabilities, and performance has increased exponentially. Much as we've seen a shift microprocessors in the past decades – from a stand alone monolithic entity into a mesh of dedicated hardware for specialised tasks we are living a similar trend in FPGAs. Nowadays, like microprocessors, FPGAs are complex heterogeneous multi processing systems on a chip (sometimes multiple dies) combining a wide array of specialised embedded resources. At a lower level they integrate cores tailored for fast fixed and floating point arithmetic, memory blocks, PCIe controllers, multicore processing units and even graphical processing units.

i Configurable Logic Blocks

CLBs are arranged in a matrix with incoming and outgoing signals from and to the interconnect fabric, XC2064 contains 64 CLBs arranged in an 8×8 matrix. In their most basic implementation, they consist of a block of combinatorial logic, a storage element and multiplexers for logic control flow. Typically vendors extend the functionality with additional elements within these blocks. The combinatorial logic contains a Look Up Table (LUT), it implements a user programmed Boolean function. Currently, the latest designs pack 6 to 8 input LUTs. The multiplexing logic allows bypassing the combinatorial logic, thus giving CLBs the ability to operate as registers, likewise combinatorial logic output can be routed directly to the interconnect bypassing the output register. This behaviour is expected in some form by all implementations. Regarding the register, it can be operated as an edge sensitive D flip-flop or as a level sensitive D latch. The memory element can be clocked from a global clock, from the logic function or from an external input, asynchronous SET and RESET are also provided from a CLB input or from a LUT output as can be seen in **Fig.** 2.2.

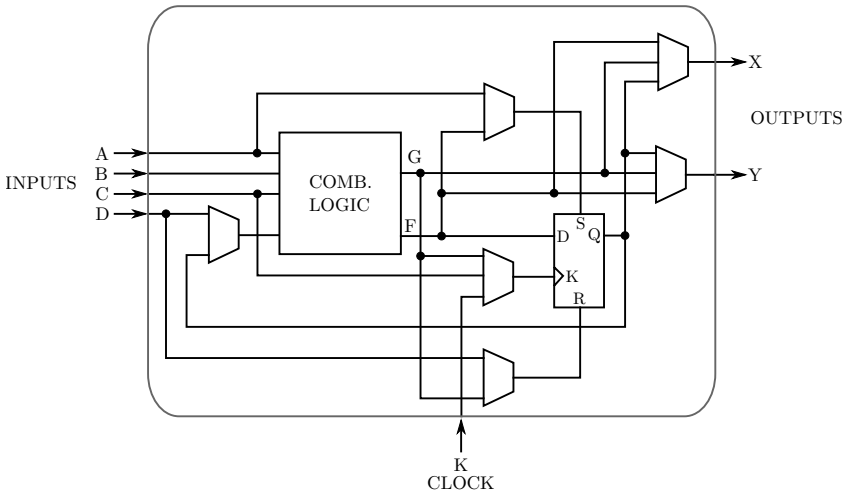


Fig. 2.2: *Configurable Logic Block adapted from XC2064 [3].*

As expected, an n input lookup table implements any n input Boolean function, on top of this, it is usually possible to pack more functions of fewer inputs. XC2064 has three modes of operation, firstly, its 4 input LUT may implement any function of 4 variables, secondly, it may implement two of any 3 variable functions or thirdly, it may use one of the inputs to dynamically select one of two 3 variable functions. **Fig.** 2.3 is a diagram of the three modes of operation. Other vendors adopt and extend this idea.

This idea is still present in present day FPGA architectures. At the time of writing, state of the art FPGAs offered by Intel with their Stratix 10 products. In their terminology Configurable Logic Blocks are called Adaptive Logic Modules (ALMs) **Fig.** 2.4, their are analogous in their architecture. ALMs are the basic building blocks of LABs (Logic Array Blocks), which also contains embedded memory blocks, variable precision DSPs, fractional PLLs (32), memory controllers, General Purpose Input/Output cells. Each ALM contains two combinational adaptive LUTs (up to 8 inputs), two full-adders, four registers. As well as a carry chain between the full adders. Output multiplexer allows for either LUT, adder or register to directly drive the ALM output to general routing. A subset of inputs may be fed directly to the registers (register packing).

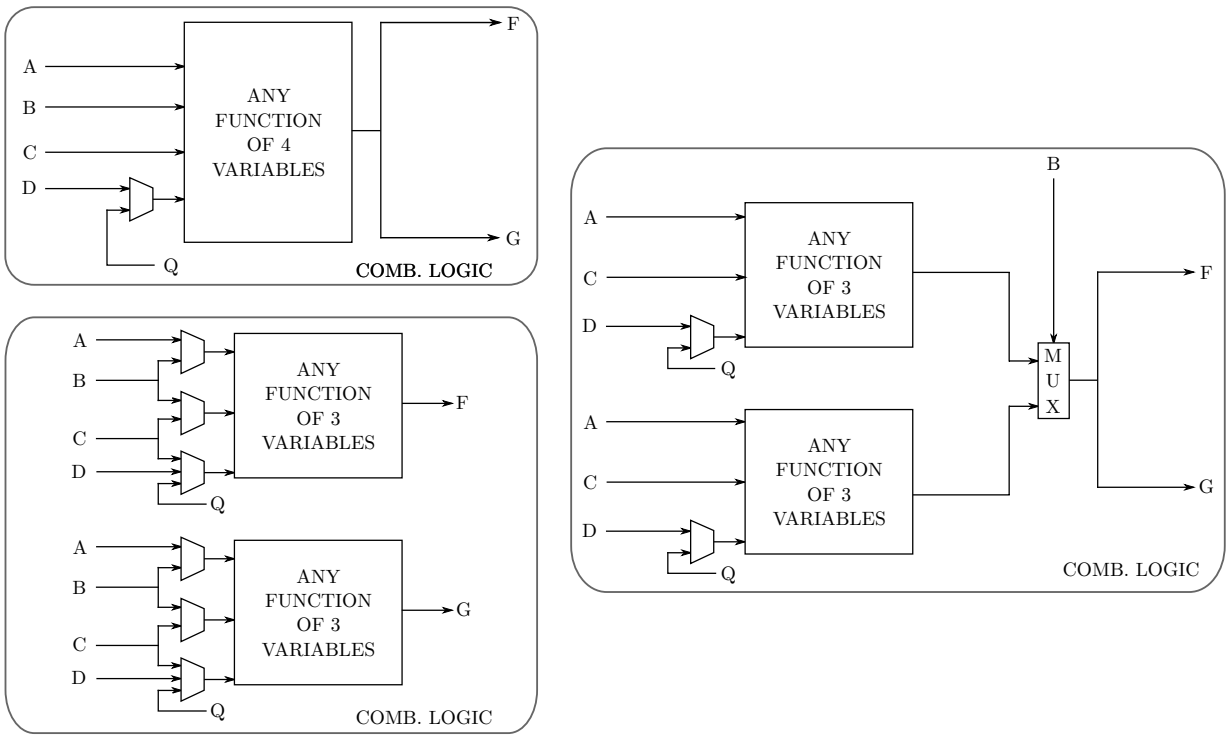


Fig. 2.3: *Combinatorial Logic modes adapted from XC2064 [3].*

Intel's ALMs also have three distinct operating modes:

- **Normal**: two logic functions per ALM with up to six combined inputs, input sharing is possible (packing) or a single function with up to six inputs.
- **Extended LUT**: supports 8 input functions in a single ALM.
- **Arithmetic**: two sets of two 4 input luts (one for each full adder input), fast carry propagation.

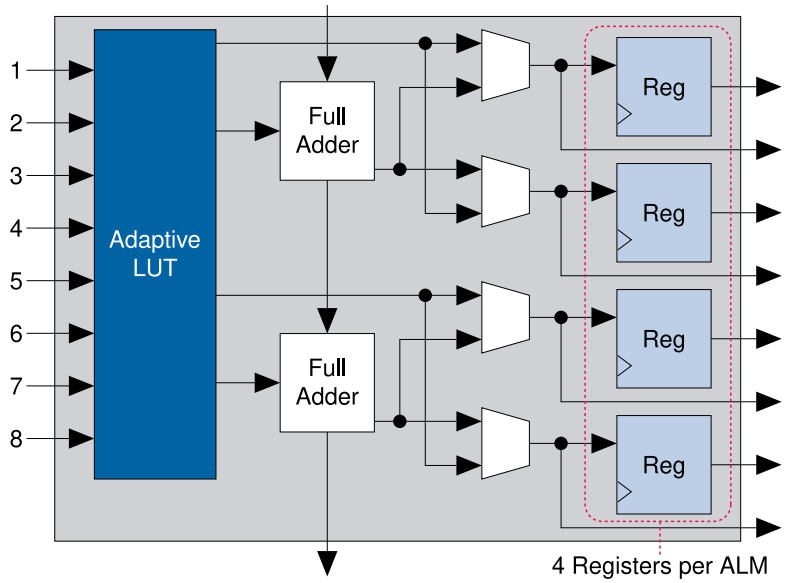


Fig. 2.4: *Simplified schematic of an ALM taken from Intel [5].*

ii Programmable Interconnect

Interconnect consists of wires that route signals between blocks, the programmability comes from a switch matrix that directs the logic flow. It represents a critical aspect of FPGA design and implementation, it is a rich area for research and optimization. In many designs, interconnect latency and power are the limiting factor. As is typical in any VLSI system, wire length and capacitance play a large role in delay, vendors adopt strategies to minimize it since interconnect wires tend to be long. Many interconnect network topologies exist (Manhattan grid, Mesh-of-Trees, hypercube, pyramidal multi-grid, multi-butterfly, multi-Benes, and others as described by Leighton [27] and Arora et al. [28]), they leverage compromises in area, delay and power and represent one aspect of interconnect innovation. In modern FPGA designs programmable interconnect may represent upwards of 90% of total area, it also accounts for the majority of a circuits delay, as opposed to CLBs [29]. Interconnect also scales poorly as process technologies improves this means that every new generation the percentage of delay due to routing increases rather delay coming from combinatorial logic as might be expected.

The previously discussed limitations play a determining role in performance of high-end FPGA chips. Innovations in this area of FPGA design are illustrated with Intel's Hyperflex Core Architecture depicted in **Fig.** 2.5. It consists of bypassable registers within the interconnect fabric and block inputs. It claims higher throughput (2× frequency), improved efficiency, greater design functionality, and increases in design productivity. It also enables enables new design and optimisation techniques such as hyper-retiming, hyper-pipelining, hyper-optimization. Additionally Hyper registers prevent having to use ALM registers, which imply overhead. They enable optimal retiming of the critical path with finer grain control and little added overhead.

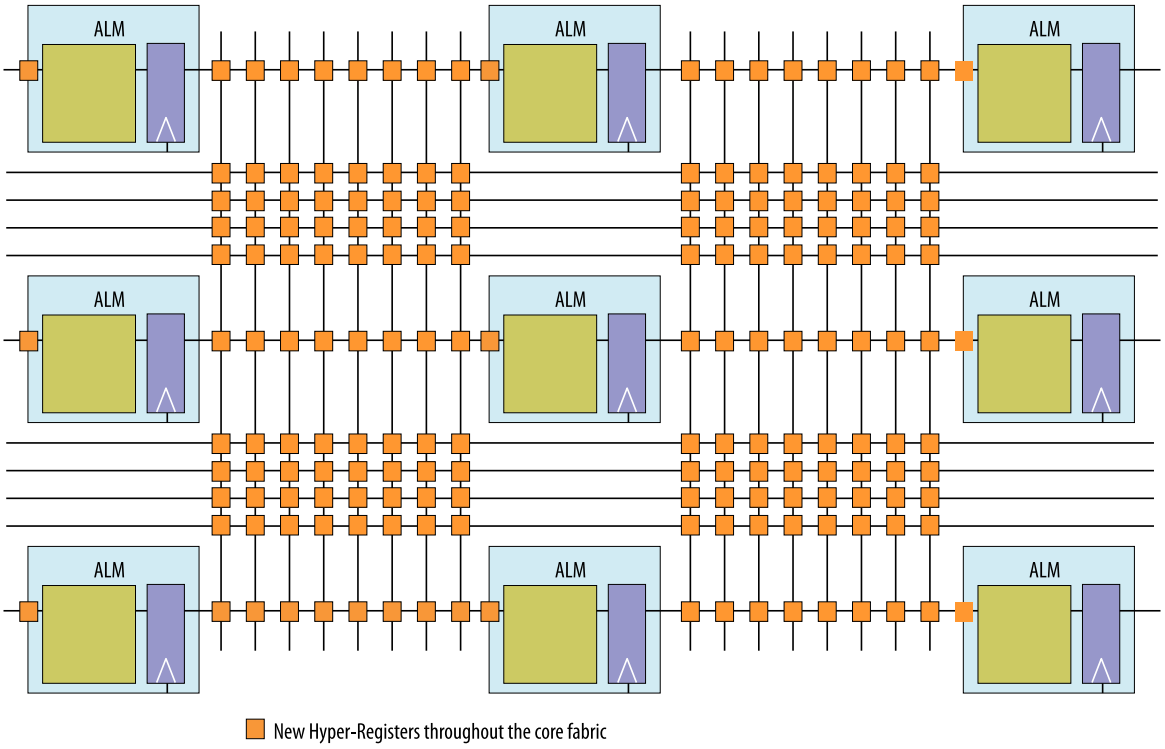


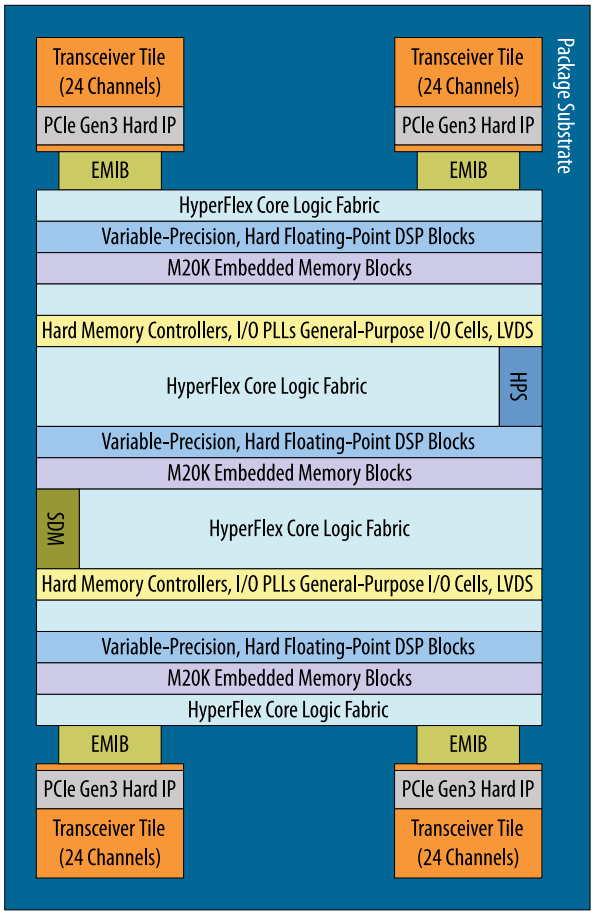
Fig. 2.5: *Hyperflex Core fabric taken from [5].*

iii Clocking

Synchronous logic requires a clock signal to update the state of memory elements. FPGAs contain a clock to orchestrate the affair, oftentimes more than one, some of which are user programmable. Clock signals are routed by means of a clocking network, it must ensure a balanced distribution of the signal. For the system to function predictably, efficiently, and correctly, clocks will preferably have high speed, low jitter, and low skew. Much like with interconnect, this is ensured by careful consideration of network topology (spine and rib, H tree, ring and others as mentioned by Lamoureux and Wilton [30]). The clock network distributes global clocks into local clocks that feed logic elements. On modern FPGAs, depending on the requirements of a particular design implementation, different clocking regions may be switched off in order to conserve energy. The clocking network can source the clock signal from external oscillators through I/O pads, from PLLs (Phase Locked Loops) which are special purpose blocks universally included in die, or they could be generated in the FPGA fabric by means of a Ring Oscillator.

iv Hardened IP

As programmable logic design has matured and use cases for FPGAs have broaden, vendors extend the capabilities of their programable platforms with a multitude of hardened logic blocks such as adders, multipliers, memories, transceivers, receivers, fully fledged micro controllers, etc. These circuits are etched into the same silicon die as opposed to the logic circuits programmed into the FPGA fabric – termed soft IP. Hard IP saves a large amounts of programmable logic fabric for commonly reused blocks such as Ethernet and PCI-Express. It also increases the efficiency by reducing power usage of arithmetical operations, this helps bridge the gap with ASICs. The heterogeneity of logic blocks contained in modern FPGAs turns the homogeneity of early FPGA architectures (**Fig.** 2.1) into a mesh of different hardened IP blocks that carry out different and specialised duties as seen in **Fig.** 2.6 and described in the following section.



HPS: Quad ARM Cortex-A53 Hard Processor System
SDM: Secure Device Manager
EMIB: Embedded Multi-Die Interconnect Bridge

Fig. 2.6: *Stratix 10 Architecture Block Diagram taken from Intel [5].*

II Modern FPGA families

This section discusses the two highest end FPGA platforms in the market as of the time of writing, Intel's Stratix 10 and Xilinx's Virtex Ultrascale+. We analyse their features and briefly discuss their usage in real world applications with examples extracted from scientific literature.

i Stratix 10

The Stratix 10 architecture represents the highest end offerings in programmable logic SoCs by Intel. These chips are built on Intel's 14 nm tri-gate transistor (FinFET) process node, they feature key technologies that enable the development of sophisticated signal processing systems. Alongside the expected configurable logic, the value proposition by Intel includes common hardened IPs such as a 64-bit Quad Core ARM Cortex-A53 multiprocessing system, High Bandwidth Memory (HBM2), PCIe, tensor multiplication, and more on top of other common DPS. Embedded Multi-die Interconnect Bridge (EMIB) allows for different process node dies to interface with the main FPGA die. Through their heterogeneous in package integration (3D System in Package) technology transceivers (up to 6, type L or H) up to 28.3 Gbps can support additional PCIe Gen4, Ethernet, and optical networking tiles among others. A transceiver is composed by 4 banks, each of which contain 6 full duplex (or independent simplex) communication channels. Clocking and other resources are shared between channels in the same transceiver.

Applications for this family of devices is found in accelerating data center workloads, where it delivers up to 8.6 TFLOPs. For machine learning, where low precision multiply and accumulate operations are common, Stratix 10 boasts up to 143 INT8 TOPs or up to 286 INT4 TOPs. Que et al. [31] realizes the potential of Stratix 10 when implementing Recurrent Neural Networks, they achieve 6.5× higher performance and 15.6× higher power efficiency in comparison to a equivalent Tesla V100 GPU implementation. Similarly, Ibrahim [32] observes a 7× speedup when implementing convolutional neural networks, furthermore he manages to leverage the 100 Gbps Ethernet connectivity between multiple Stratix 10 devices to achieve 2× performance improvement when adding an additional FPGA device. Similarly, Shipton et al. [33] proves the feasibility of a real time implementation of WaveNet, a text-to-speech synthesizer by Oord et al. [34]. In this implementation, 256 concurrent streams of the model achieve almost human like speech. Stratix 10 manages to outperform previous reference state of the art implementations on NVIDIA* V100 GPUs by a 8×.

Regarding larger players in the field, Microsoft™ deploys Stratix 10 in the data center through Project Brainwave Chung et al. [35]. It serves as their primary infrastructure for serving Artificial Intelligence in real time. With this data center implementation Microsoft™ augments their services (Bing and Azure) with deep learning neural network inference in real time.

The networking prowess of Stratix 10 is demonstrated by Plakalovic et al. [36], where a high throughput Ethernet package generator is implemented. Such a device is employed during stress testing and validation of networking equipment. The solution manages to fulfill the required constant 40 Gbps bit rate. This level of

performance is unachievable with CPU implementations and more competitive in terms of price with commercial solutions. This already impressive implementation at 40 Gbps pales in comparison to the claimed maximum networking capability of Stratix 10, Intel [37, 38] offers IP for up to 400 Gb Ethernet using eight 58 Gbps PAM4 lanes.

To illustrate DSP applications of Stratix 10, Kamp et al. [39] implements Complex Multiply Accumulate Cells (CMACs) using the FPGAs variable precision DSP blocks. CMACs are used for the correlation of different incoming signals in radio telescopes, amongst other systems. The implementation achieves performance upwards of 700 MHz letting the vendor tool infer and instantiate the appropriate number and type of DSP. Stratix 10 provides fixed point blocks configurable as a 27×27 or double 18×18 bit multipliers and hard pre-adder support. For floating point arithmetic, Intel [40] provides multiplication, addition, subtraction, multiply-add, and multiply-subtract.

One of the defining features of the Stratix 10 architecture is its Hyperflex core fabric. It enables precise, near-optimum retiming and pipelining optimizations. Tan et al. [41] evaluates real world performance of Hyperflex through selected benchmarks designed to be representative of a variety of scenarios (pipelining with and without loops, designs with feedback loops). The systems demonstrated (DES, MD5, AES, SHA1 encryption, Context-Adaptive Variable-Length Coding, dot products and DFT), along with implementation effort estimates illustrate the effect Hyperflex has on the final product. Most of the benchmark systems demonstrate a frequency speed up between 1.5× and 2× with about an average investment of about 20 engineering hours. The fastest systems where optimized in excess of 800 MHz, hinting at an upper bound on implemented system performance. The cost of such optimizations comes in the form of power, which increased between 5× and 10×.

ii Virtex UltraScale+

The UltraScale platform epitomizes Xilinx's top performing configurable logic MPSoCs, the Virtex UltraScale+ variant promises the "*highest transceiver bandwidth, highest DSP count, and highest on-chip and in-package memory available in the UltraScale architecture*" according to Xilinx [42]. Fabricated in a 14nm/16nm FinFET node and utilizing 3D ICs (stacked silicon dies) Virtex UltraScale+ delivers high bandwidth I/O and cutting edge DSP capabilities. It implements routing between silicon dies operating at more than 600 MHz. Key features include the aforementioned 3D stacking, integrated PCIe blocks, 32.75 Gbps Transceivers, DSP cores up to 38 TOPs, high bandwidth memory (HBM). Applications for this architecture are found in accelerating workloads in the data center, high-speed switching for 5G infrastructure, wired connectivity thanks to its integrated 100G Ethernet and 150G Interlaken networking IP, RADAR signal processing, and testing and measurement amongst others. For Artificial Intelligence applications, Virtex UltraScale+ allows low precision INT8 multiplication in its DPS 18×27 DPS for increased efficiency with little precision loss.

High speed connectivity with this chip has been explored by Wanotayaroj et al. [43], they demonstrate PAM4 (4-Level Pulse-Amplitude Modulation) transceiver to achieve 53.125 Gbps communication lanes through electro-optical lanes. Adding to high speed networking, Martinasek et al. [44] augments dual full duplex 100

Gbps Ethernet connections with a practical implementation of real time AES encryption at 200 Gbps total throughput. The system is based on the IP security protocol, they use a NFB-200G2QL network card as described by Kekely et al. [45]. The maximum frequency achieved for the AES encryption module is of 378 MHz, representing a maximum throughput of 48 Gbps, to achieve the required 200 Gbps eight AES components where deployed parallelized at a frequency of 200 MHz.

Data center High Performance Computing workload acceleration benefits from Virtex Ultrascale+ in the fields of genomics, astrophysics, big data analysis, cyber security, finance, machine learning, molecular dynamics, oil & gas, and weather & climate.

With this platform, in genetic sequence base-pair alignment, Haghi et al. [46] achieves 13.5× speedup with 14.6× less energy when compared with reference CPU implementations. This team achieves implementations running at clock rates of 200 MHz pairing two XCVU37P chips. Other efforts in this field leverage technologies such as in-memory computation, GPUs and ASICs. FPGA projects from Fei et al. [47] present similar implementations on Virtex 7 of the Smith-Waterman algorithm using fine grained parallelized systolic arrays obtaining a speedup between 3.6× and 25.2× operating at 200 MHz when compared to similar CPU and FPGA implementations.

Big data for Cloud Service Providers computation time is reduced by 10^4 times using Smekal et al. [48] approach for data splitting techniques at 80 MHz. This is a computationally intensive workload that enables privacy-preserving computation in an external cloud in the same vain as homomorphic encryption. Barrow et al. [49] manages a 12× speedup when computing ZFP numerical CODEC compression for eliminating bottlenecks in scientific data when communicating different nodes of supercomputing clusters. Their approach is coded using a SystemC synthesis tool chain.

CHAPTER

3

PERFORMANCE OPTIMISATION OF VARIABLE PRECISION GENERIC CONTROLLER FPGA IMPLEMENTATIONS.

In this chapter we present the results of a quantitative and exhaustive design effort of the logic architecture of a generic digital Proportional Integral Derivative controller. Precision ranges from 8 to 96 bits are evaluated, and the maximum frecuency of each FPGA is obtained. Proof of concept real world implementations are demonstrated using Xilinx FPGAs, spanning from 464 MHz in Series 7 FPGA to 890 MHz Kintex and Virtex Ultrascale+.

I Introduction

"PID control is the most common form of feedback."–Åström [50], it is characterized by computing the error between a process variable and a set point, and scaling corrective action according to the past, present and future errors by the use of proportional, integral and derivative terms, origin of the initialism PID. Feedback systems are the cornerstone of control theory, they are exemplified by Huygens' pendulum clocks and centrifugal governors for windmills, as described by Fuller [51], governors where later notably adapted to steam engines by Watt [52] and mathematically modelled in '*On governors*' by Maxwell [53]. In most of control theory, feedback is most commonly understood to be negative, that is it counteracts output fluctuations due to external disturbances. Negative feedback was first used in electronics by Black [54] in analogue amplifiers, allowing for precise systems to arise from imprecise components. Analogue amplifiers with negative feedback can be used to implement PID controllers as is elegantly demonstrated by [55] and their single operational amplifier PI controller. Mainly analogue implementations have given way to digital approximations, be it as software running in general purpose microprocessors or as configurable logic in FPGAs.

General purpose microcontroller systems have been traditionally used to realise PID controllers ([56, 57, 58]) but in recent years FPGAs have become an attractive alternative. This is mainly due to FPGAs being able to offer high performance while also being highly flexible, they can also be tailored for low power consumption. PID control serves to regulate systems according to any measured variable, most commonly temperature, position, speed, concentration, pressure, force or any other magnitude susceptible of being measured. Thanks to its versatility, the fields of application of PID controllers go, from robotics, automotive, industrial, to space, defense and many others.

FPGAs are better suited at implementing digital controllers than ASICs because they avoid the high upfront costs, offer higher flexibility and are cheaper. Digital systems are mostly immune to noise or are less affected. Only highly specialised applications necessitate the advantages posed by ASICs, namely, lower power consumption and lower area.

PID controller FPGA implementations reported in scientific literature, tend to be generated using high level synthesis. [59] explains how this approach has inherently more overhead, leading to less than optimal hardware. In addition to the loss of performance by the use of HLS, most of the systems discussed are developed to work at slower clock rates or with low precisions. This presents the need for the development of hardware implementations of PID controllers that appropriately and effectively use the FPGAs available. In this work we optimise the digital logic architecture in terms of area, performance and precision. Our proposed approach yields circuits that operate at faster clock rates, with higher precision signals while utilising fewer resources.

The main features and objectives of this work are:

- PID controller architecture design and validation.
- Optimised implementation designs for a wide range of precisions from 8 to 96-bit signals.

- All proposed systems operate at the maximum possible FPGA clock rate.
- Design considerations for implementation in a wide range of available FPGAs.

The structure of section I introduces the common notions discussed in this work and the work itself. Section II derives the discretized standard form of a PID controller. It also breaks down the resulting equation into simple operations for FPGA implementation. Section III arrives at the architecture for the proposed implementations. Section IV characterizes the implementations of our proposed controllers (PID/PI/pipelined PI) on different FPGAs (Series 7, Artix, Kintex and Virtex Ultrascale+). Section V reviews and compares the features and utilization of controllers in literature at different widths. Section VI details the concluding remarks summarising the proposed architecture, performance and resource utilization of our design approach.

II Theoretical Background

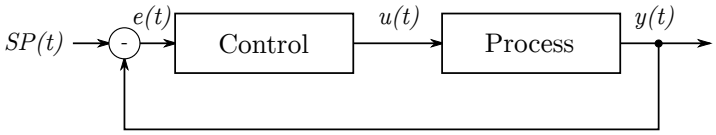


Fig. 3.1: *Control feedback loop.*

Fig. 3.1 shows a typical control system scheme. The process to be controlled produces a signal $y(t)$ which is measured and used as input alongside the desired set-point $SP(t)$. Both of this signals are used to calculate the error $e(t)$. The behaviour of the PID controllers is described with a transfer function, the standard form of which is as follows:

$$u(t) = K_\mathrm{p} \cdot \left(e(t) + \frac{1}{T_\mathrm{i}} \cdot \int_0^t e(\tau)\,\mathrm{d}\tau + T_\mathrm{d} \cdot \frac{\mathrm{d}e(t)}{\mathrm{d}t} \right) \tag{3.1}$$

In this form, the parameters have direct physical counterparts, K_p refers to the gain of the correction; T_i and T_d the integral and derivative times; $u(t)$ represents the control action applied to the system to control; $e(t)$ is the error at instant t. The first term corresponds to the proportional action, the second and third to the integral and derivative action.

The difference equation of a standard form PID over a sample time T would be:

$$u[k] = K_\mathrm{p} \cdot \left(e[k] + \frac{T}{T_\mathrm{i}} \cdot \sum_{\tau=0}^{k-1} e[\tau] + \frac{T_\mathrm{d}}{T} \cdot (e[k] - e[k-1]) \right) \tag{3.2}$$

This equation needs all values of the error between $t = 0$ and $t = k - 1$ to calculate the corrective action, which makes realization unfeasible. To solve this problem we observe the control output at sample time $k - 1$:

$$u[k-1] = K_\mathrm{p} \cdot \left(e[k-1] + \frac{T}{T_\mathrm{i}} \cdot \sum_{\tau=0}^{k-2} e[\tau] + \frac{T_\mathrm{d}}{T} \cdot (e[k-1] - e[k-2]) \right) \tag{3.3}$$

and calculate the difference as (3.2) - (3.3):

$$u[k] - u[k-1] = K_{\mathrm{p}} \cdot \left(e[k] - e[k-1] + \frac{T}{T_{\mathrm{i}}} \cdot e[k-1] + \frac{T_{\mathrm{d}}}{T} \cdot \left(e[k] - 2 \cdot e[k-1] - e[k-2] \right) \right) \tag{3.4}$$

which yields the control output u[k] recursively simplified as:

$$u[k] = u[k-1] + a_0 \cdot e[k] + a_1 \cdot e[k-1] + a_2 \cdot e[k-2] \tag{3.5}$$

where each constant corresponds:

$$a_0 = K_{\mathrm{p}} \cdot \left(1 + \frac{T_{\mathrm{d}}}{T} \right) \tag{3.6}$$

$$a_1 = -K_{\mathrm{p}} \cdot \left(1 - \frac{T}{T_{\mathrm{i}}} + 2 \cdot \frac{T_{\mathrm{d}}}{T} \right) \tag{3.7}$$

$$a_2 = K_{\mathrm{p}} \cdot \frac{T_{\mathrm{d}}}{T} \tag{3.8}$$

Equation (3.5) calculates the corrective action for instant k, $u[k]$, using the previously calculated value $u[k-1]$ and the values of the error at instant $k, k-1$ and $k-2$. This form for the system is realisable as hardware, unlike (3.2).

Often, in real-world applications, only PI control is implemented. This is because the derivative term tends to be too sensitive, thus making the system less stable. If $T_{\mathrm{d}} = 0$ our system is further reduced to:

$$u[k] = u[k-1] + K_{\mathrm{p}} \cdot e[k] - K_{\mathrm{p}} \left(1 - \frac{T}{T_{\mathrm{i}}} \right) \cdot e[k-1] \tag{3.9}$$

The recursive PID algorithm derived can be broken down into simple arithmetic operations as follows:

$$e[k] = \mathrm{SP}[k] - y[k] \tag{3.10}$$

$$P[k] = a_0 \cdot e[k] \tag{3.11}$$

$$I[k] = a_1 \cdot e[k-1] \tag{3.12}$$

$$D[k] = a_2 \cdot e[k-2] \tag{3.13}$$

$$S_1[k] = P[k] - I[k] \tag{3.14}$$

$$S_2[k] = S_1[k] + D[k] \tag{3.15}$$

$$u[k] = u[k-1] + S_2[k] \tag{3.16}$$

These simple operations and connections are translated into VHDL to appropriately generate the hardware circuit to be implemented in FPGA.

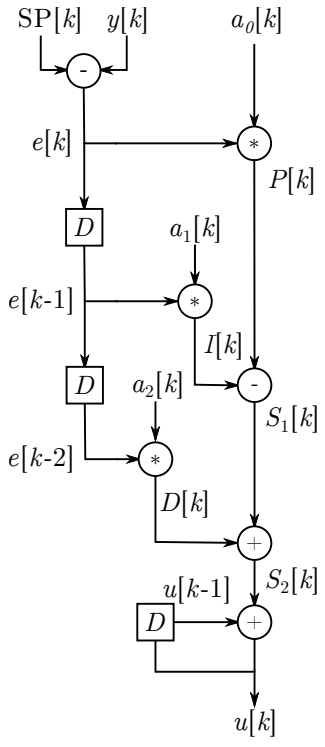


Fig. 3.2: *PID data flow graph.*

III Proposed Approach

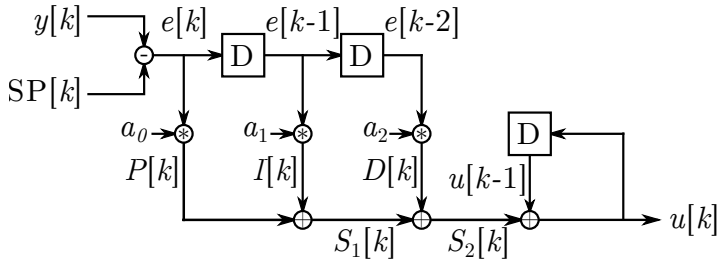


Fig. 3.3: *Digital PID controller design.*

The alterante representation in **Fig.** 3.3 better illustrates the topology of our system. At the core we find a 3−tap FIR filter whose input is the error signal (the difference between SP$[k]$ and $y[k]$) and whose output signal is simply accumulated. When only a PI controller is needed, one of the delays, one multiplier and one adder are omitted, leaving an even simpler 2−tap FIR filter. The longest combinatorial path crosses through one multiplier and four adder/subtracters. The large latency in the critical path limits the circuit to operating at a slower clock rate.

Our approach seeks to overcome this frequency limitation and achieve the maximum possible performance. To do so we breakup the critical paths using registers. **Fig.** 3.4 design shows an approach where at any given point we find the minimum adders/multipliers between registers. The additional registers add a latency of 6 clock cycles and increase the maximum frequency to 891 MHz as calculated by Vivado's `report_pipeline_analysis` tool. The limiting factor the of the maximum frequency are the DPS slices.

The FPGAs considered to realize our proposed controller architectures has a wide range of maximum frequencies as summarised in **Table** 3.1. The theoretical upper bound for the clock rate of the FPGA is limited by the DSP and PLL cells. It ranges from 464 MHz at the lower end, and tops out at 891 MHz with the Speed Grade 3 Ultrascale+ variant. As we will see in the following sections, with the proposed approach we manage to produce working implementations for a wide range of precisions on a wide range of FPGAs fully utilizing available resources and satisfying the maximum achievable clock rate of each platform respectively.

Registers have been added taking into to consideration the fact that, for maximum performance, DPS slices can be optimized with three pipeline stages. Thus for Vivado to infer this optimization we add three registers spread between the inputs and outputs of the multipliers. Proper design techniques impose having to add input and output registers for each module in our design, this is why we add registers to $y[k], u[k]$ and SP$[k]$.

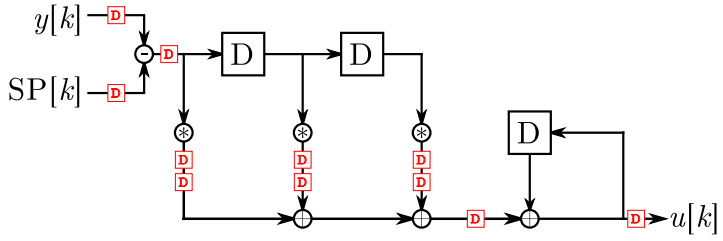


Fig. 3.4: *PID after pipelining and retiming.*

The FPGAs considered to realize our proposed controller architectures has a wide range of maximum frequencies as summarised in **Table** 3.1. The theoretical upper bound for the clock rate of the FPGA is limited by the DSP and PLL cells. It ranges from 464 MHz at the lower end, and tops out at 891 MHz with the Speed Grade 3 Ultrascale+ variant. As we will see in the following sections, with the proposed approach we manage to produce working implementations for a wide range of precisions on a wide range of FPGAs fully utilizing available resources and satisfying the maximum achievable clock rate of each platform respectively.

Table 3.1: *Xilinx maximum operating frequency.*

		f_{max} (MHz)		
	Grade	-1	-2	-3
Artix	Series 7	464	550	628
	Ultrascale	-	-	-
	Ultrascale+	645	775	-
Kintex	Series 7	464	550	741
	Ultrascale	594	661	741
	Ultrascale+	645	775	891
Virtex	Series 7	547	650	741
	Ultrascale	594	661	741
	Ultrascale+	645	775	891

The developed systems are tuned with a process to be controlled modelled by a transfer function on the test-bench. The value of the tuning parameters are dependant on the system to be controlled and the desired behaviour of the controller thus have to be specified and programmed when deploying a new implementation. Because the value of the constants can change we decide to use DSP multipliers, because they can achieve full performance regardless of the value of the constant. LUT based constant multipliers speed and area are dependant on the value of the constant, its analysis would requiere optimisation for the worst case scenario. The approach taken covers any possible value the tuning parameters may adopt, it comes at the cost of using DSP multipliers.

In **Fig.** 3.5 we depict the controllers response with a test-bench of a example system. When derivative action is applied, the system tends to oscillate and takes longer to settle around the set point. The PI configuration tends to settle quicker though this is also affected to the tuning parameters which have been manually approximated. We can also observe a 6−sample lag corresponding to the pipeline registers. The demonstration of the systems is made by initially setting the set-point at a given value and letting the controller correct the system to reach it, afterwards the set-point was lowered and left constant. We then analyse the evolution of the systems. In **Fig.** 3.5(a) we see after the step down the system is not able to reach the set-point. This may be intrinsic to the controller and the set-point chosen it may also be affected by quantization errors and truncation errors, in **Fig.** 3.5(b) wee see this problem is fixed.

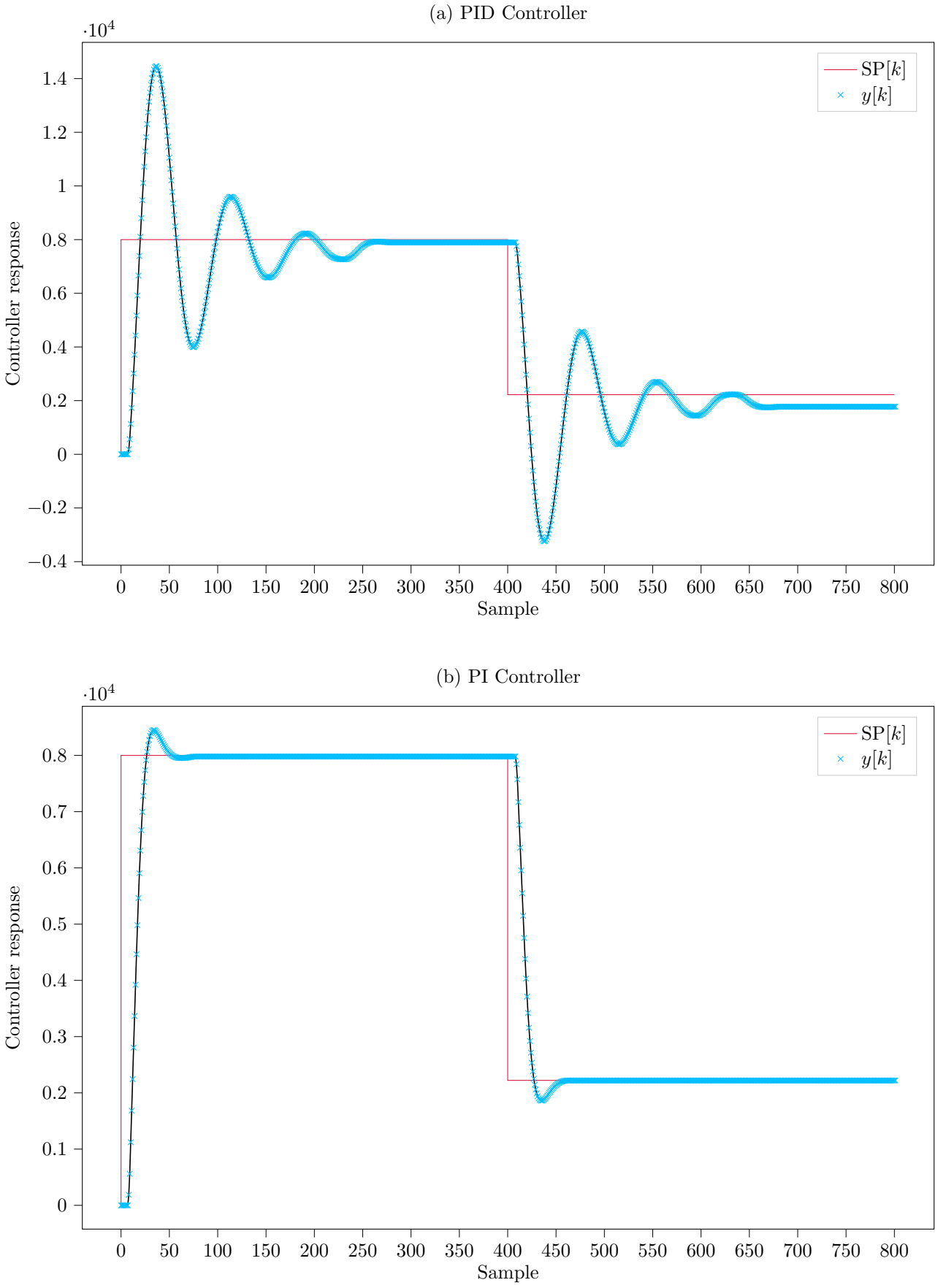


Fig. 3.5: *(a) PID and (b) PI functional simulation.*

IV Results

In this section we proceed analysing implementations of our proposed design. We pay particular attention to timing and resource utilization. To properly define the characteristics of the systems we perform a sweep analysis increasing the width of our design and the necessary additional pipeline stages required to meet the highest supported frequency. First we present our results for the largest and highest frequency Ultrascale+ Speed Grade-3 devices, results have been measured to be applicable to both Kintex and Virtex family devices (xcku5p-ffvd900-3-e and xcvu35p-fsvh2892-3-e), as previously detailed we achieve 891 MHz.

Following this we proceed investigating lower end Artix-7 implementations. This approach allows us to demonstrate the relative ease with which we can adapt our design to obtain the maximum possible performance under different conditions. The robustness of our design process increases the flexibility of our architecture and allows us to port our digital signal processor to whatever platform may be available.

The Artix-7 (xc7a100tcsg324-1) FPGA has dedicated DSP slice multipliers (DSP48E1) that operate at a reported maximum clock rate of 464 MHz, when fully pipelined. We take this maximum frequency as the target for our designs.

i PID Implementation

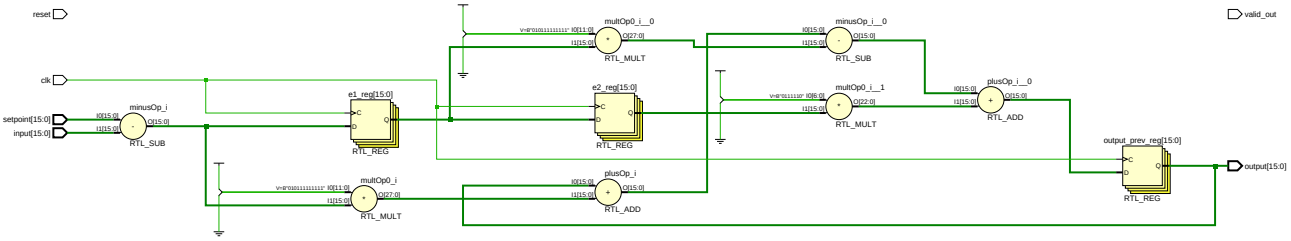


Fig. 3.6: *PID inferred RTL.*

The naive PID controller has three registers and a long combinational path as depicted in **Fig.** 3.3. The resource utilization is summarised in **Table** 3.2 of a example controller with 16-bit signals. The maximum clock rate is 450 MHz which is good though far from optimum. When we include the differential action, the controller tends to oscillate around the set point, it also consumes more resources, namely one extra multiplier. To avoid this and simplify our design we set the constant $a_0 = 0$. This allows the Vivado synthesizer to eliminate the unused logic leading to the following PI implementation:

Table 3.2: *Components of the pipelined PID controller.*

	Inputs	Bits	Quantity
Adders	3	16	2
	2	16	2
Registers		16	10
DSP Blocks			3

This simplified design boasts two registers, two multipliers and two adders. Performance is only slightly higher than the previously discussed PID implementation at 500 MHz. Functionally we observe less oscillation as expected.

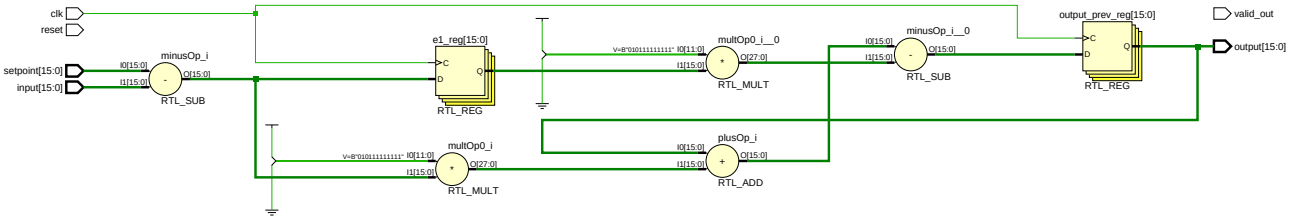


Fig. 3.7: *PI inferred RTL.*

ii PI Pipelined Implementation

We proceed pipelining and retiming the system. First we analyse a fully pipelined PID controller, it manages to achieve full performance at the FPGA's theoretical maximum. The pipelined 16-bit PI variant only needs two DSPs while achieving full performance at 891 MHz and good functional response as seen in **Fig.** 3.5. Resource wise, **Table** 3.3 summarises the utilization of a reference 16-bit wide design.

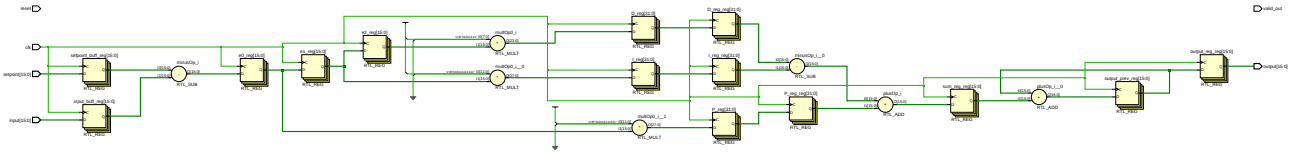


Fig. 3.8: *Pipelined PID inferred RTL.*

Table 3.3: *Resource utilization PI controller.*

Resource	Utilization	Available	Utilization %
LUT	48	216960	0.022
FF	143	433920	0.033
DSP	2	1824	0.110

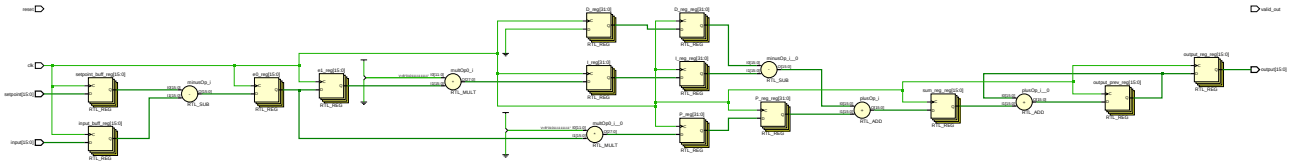


Fig. 3.9: *Pipelined PI inferred RTL.*

Lastly, the pipelined 16 bit PI variant only needs two DSPs while achieving full performance at 891 MHz and good functional response as seen in **Fig.** 3.5. We will continue out analysis using the pipelined PI controller. Resource wise this is a light implementation as can be seen in the **Table** 3.4 and **Table** 3.5. The 16 bit variant uses almost negligible routing resources in comparison to available resources. Detailed resource utilization and inferred RTL analysis gives us confidence that the developed hardware described is as expected and no unwanted or redundant logic structures are generated.

Table 3.4: *Slice Logic utilization.*

	Used	Fixed	Prohibited	Available	Util%
CLB LUTs	48	0	0	216960	0.02
LUT as Logic	48	0	0	216960	0.02
LUT as Memory	0	0	0	99840	0.00
CLB Registers	112	0	0	433920	0.03
Register as Flip Flop	112	0	0	433920	0.03
Register as Latch	0	0	0	433920	0.00
CARRY8	6	0	0	27120	0.02
F7 Muxes	0	0	0	108480	0.00
F8 Muxes	0	0	0	54240	0.00
F9 Muxes	0	0	0	27120	0.00

Table 3.5: *Primitives utilization.*

	Used	Functional Category
FDRE	112	Register
LUT2	47	CLB
INBUF	33	I/O
IBUFCTRL	33	Others
OBUF	16	I/O
LUT4	15	CLB
CARRY8	6	CLB
DSP48E2	2	Arithmetic
OBUFT	1	I/O
BUFGCE	1	Clock

With our parametrized design we can now perform a sweep increasing the number of pipeline registers (from +0 to +14 stages) after each of the multipliers with increasingly wider word lengths, from a precision of 8-bits through to 96-bits. **Fig.** 3.10 is a plot of the Worst Negative Slack (WNS) as calculated by the Vivado Implementation tool. A positive WNS indicates that the timing requirements are met, a negative value is representative of how much slower the delay of the slowest path is in comparison to the clock period.

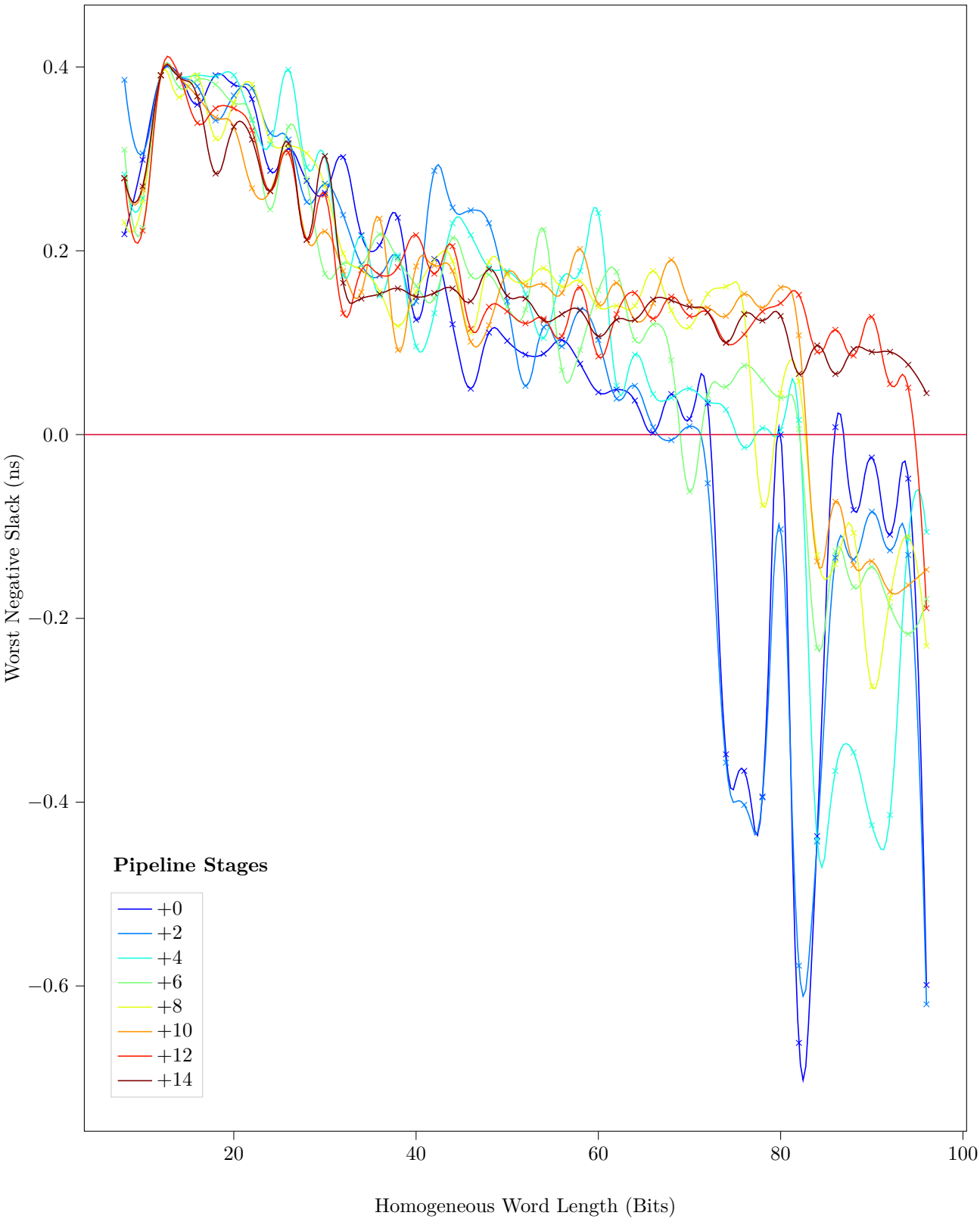


Fig. 3.10: *Static timing analysis of the proposed PI controller on UltraScale+ targeting 891MHz.*

As expected we see a linear drop off in positive slack between 8 and 64-bits. When working with signals wider than 64-bits additional pipeline stages are needed to operate at the fastest clock rates. With 4 additional pipeline stages we manage to make 76-bit wide designs work, with +12 pipeline stages timing is met up until 94-bits finally meeting all requirements at 96 bits with +14 pipeline stages.

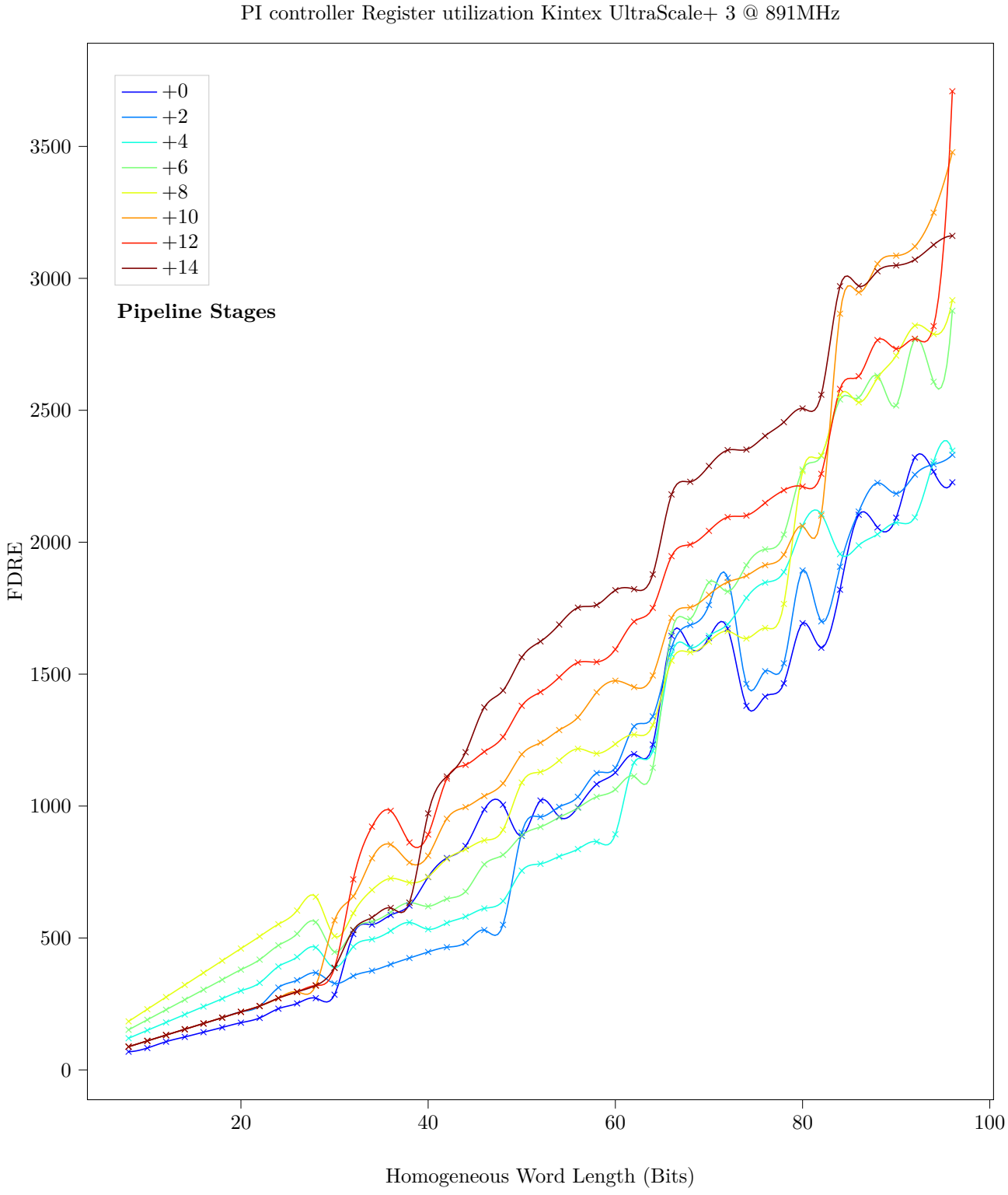


Fig. 3.11: *D Flip-Flop with Clock Enable and Synchronous Reset primitive utilization as signal bit width increases on UltraScale+ PI controller.*

In **Fig.** 3.11 we evaluate register utilization, they are placed breaking up the critical paths formed by carry chains during additions and subtractions, at narrow word lengths they are almost insignificant, while wide designs require a large amount of them. With designs larger than 22-bits (requiring more than one DSP per multiplication) more registers are employed by more highly pipelined designs, as expected.

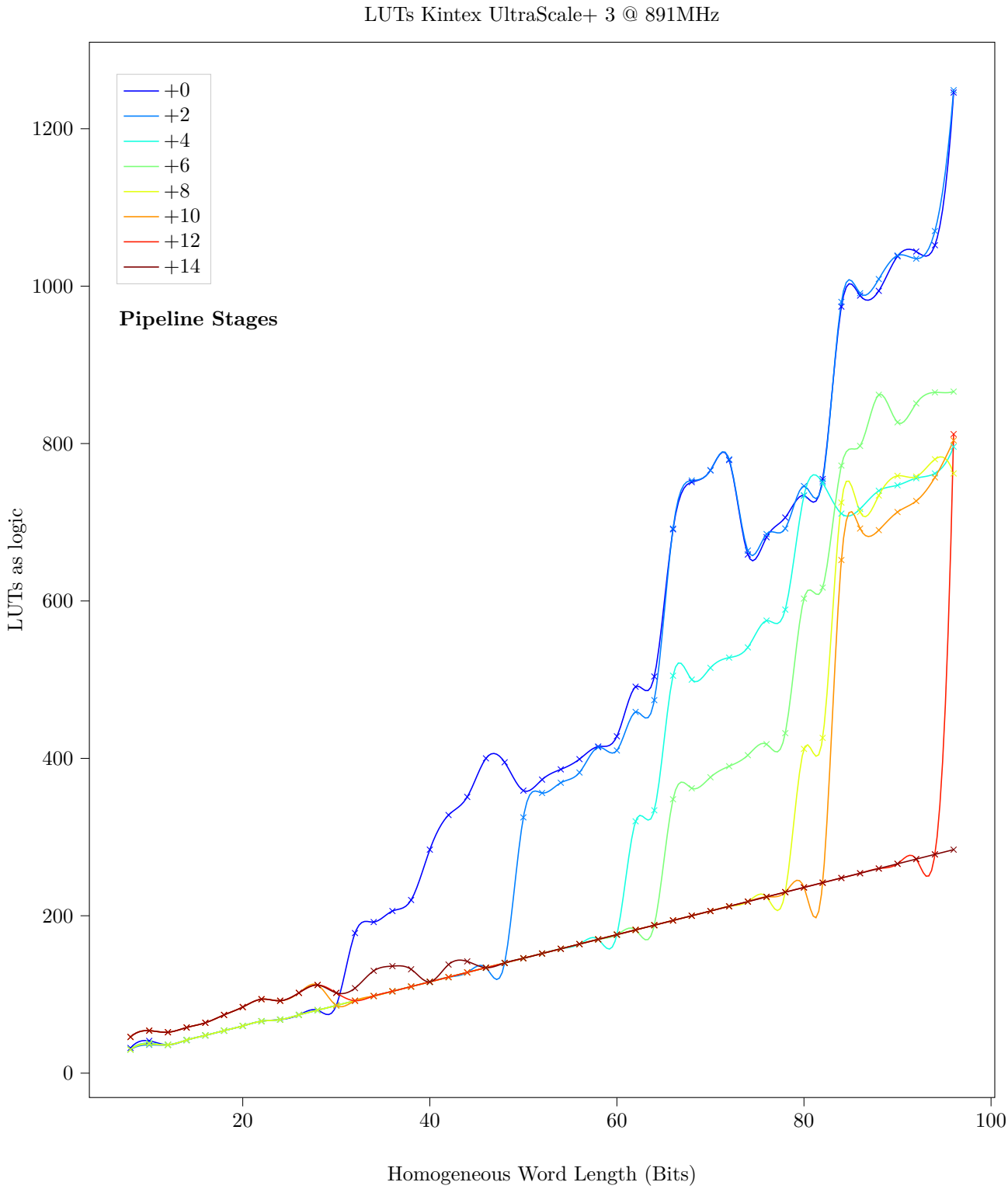


Fig. 3.12: *Look Up Tables as signal bit width increases on UltraScale+ PI controller.*

In **Fig.** 3.12, LUTs as logic show a less defined correlation, wide designs use slightly more LUTs and less pipelined designs tend to use twice or thrice as many LUTs as highly pipelined results. Once again we see this trend emerge past the 22-bit width, indicating they are due to multiple DSPs utilized per multiplication. It is notable that more LUTs are utilised whenever the system does not meet timing requirements, only the +14 pipeline stage system (that meets timing at all bit widths) has a almost constant linear LUT utilization progression.

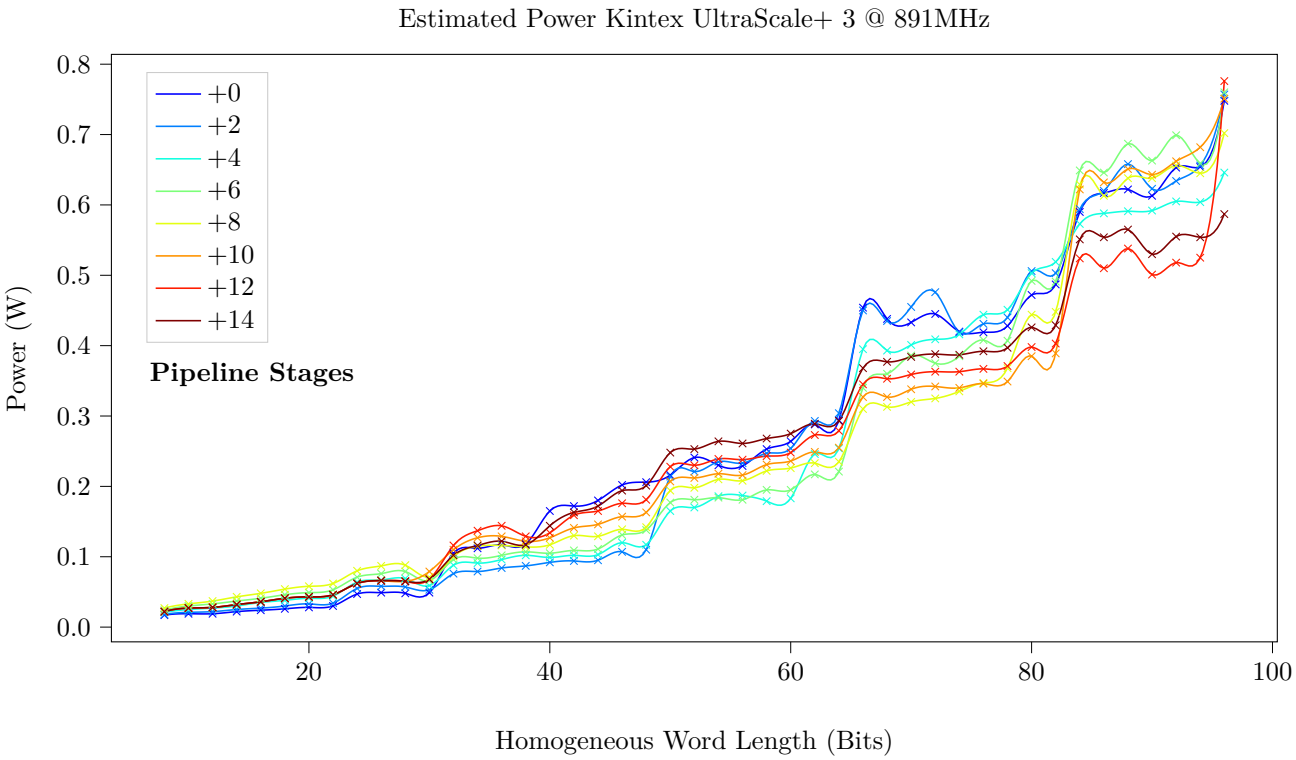


Fig. 3.13: *Power consumption increase as word length widens on Kintex UltraScale+ implementations.*

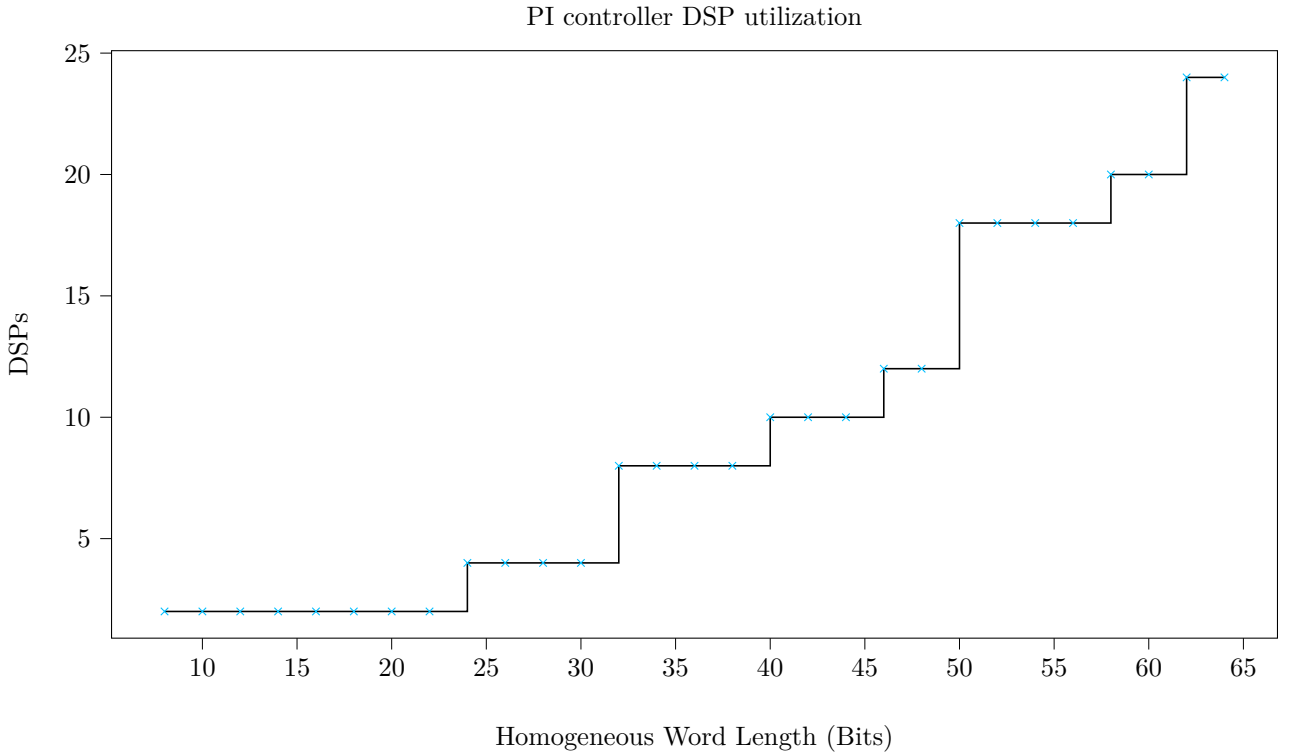


Fig. 3.14: *Number of DSP multipliers needed to implement a PI controller with different width signals.*

Power consumption is shown in **Fig.** 3.13. It is related to the width of the design and only slightly affected by the amount of pipeline stages. There is correlation with the number of DSPs implemented, we see step increments at at around 22, 44, 66 and 88-bits, these correspond to the DSP multipliers widest port width.

From the sweep analysis we see how additional pipeline stages are needed to meet timing requirements above 70-bits and that +14 pipeline stages are necessary to operate the PI controller at maximum frequency at 96 bits.

Worst case performance at 96-bits without additional pipeline stages would fall to about 580 MHz, maximum frequency caps out at 891 MHz as limited by DSP slices. DSP utilization is independent of pipeline stages, it varies only according to word length it can be analysed in **Fig.** 3.14. As is expected, below 22 bits (DSP width) only two DSPs are utilized, one for the proportional term, and another for the integral term.

iii PI Implementation on Artix 7

Preliminary considerations prove unfeasible implementing additions and subtractions on LUTs due to the latency incurred by the carry chains, particularly at wide word lengths. To overcome this limitation, we choose to instead use DSP48E1 slices to perform additions. This is possible because, on top of a multiplier, there is also an adder included in the DSP slice. The cost of course is that we occupy some of the limited DSP48E1 slices, a trade-off we have to make in order to achieve maximum performance.

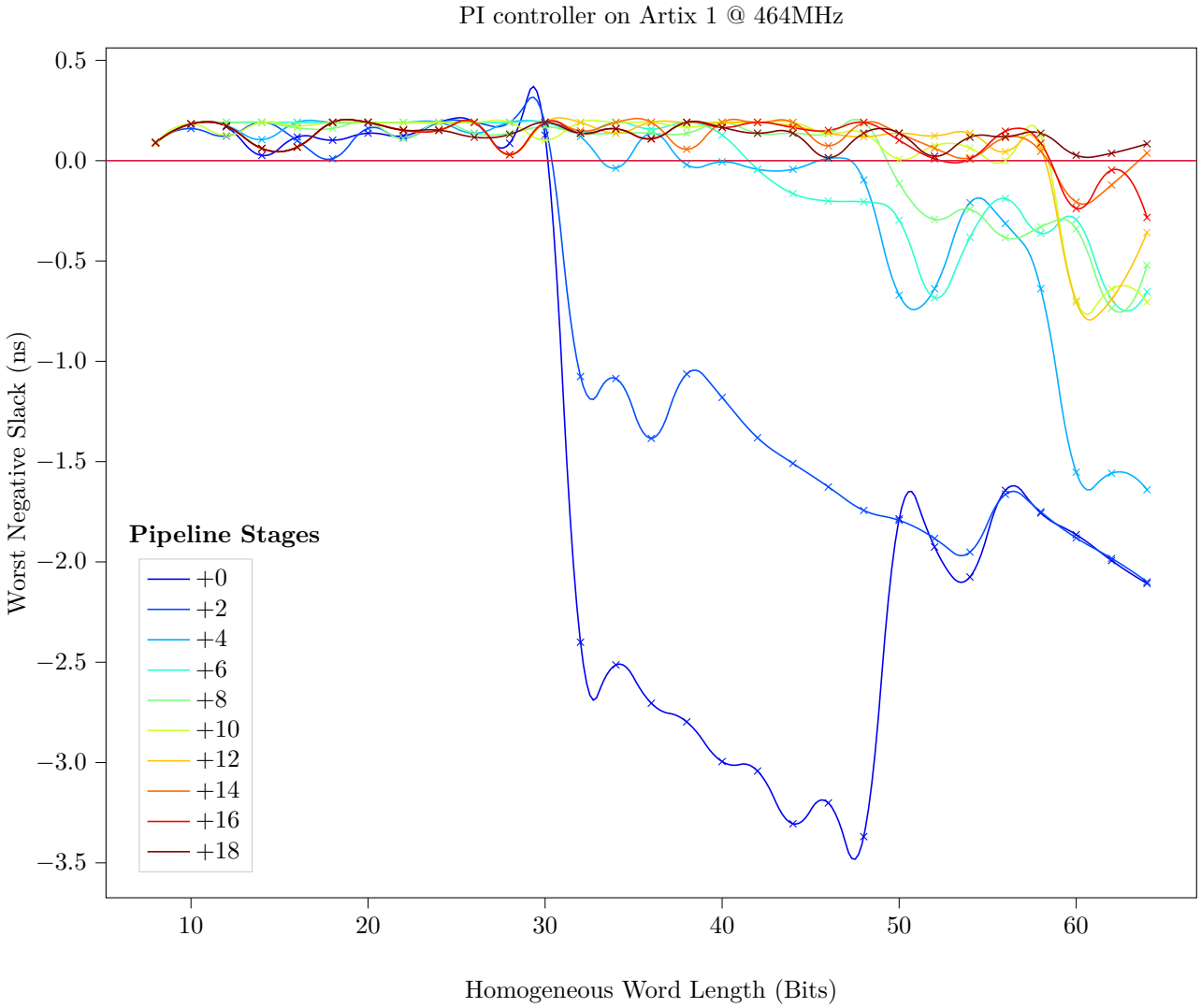


Fig. 3.15: *Timing Analysis PI controller on Artix.*

The performance of our system is depicted in **Fig.** 3.15. The performance trend is similar to **Fig.** 3.10. We have solid performance with designs narrower than 30 bits, not needing additional stages of pipeline. Beyond this point there is a steep drop off in performance, mainly due to multiple unpipelined DSP slices. As additional

stages of registers are added, wider and wider designs begin to meet timing requirements. Finally with 18 additional pipeline stages we can obtain maximum performance at a 64 bit width.

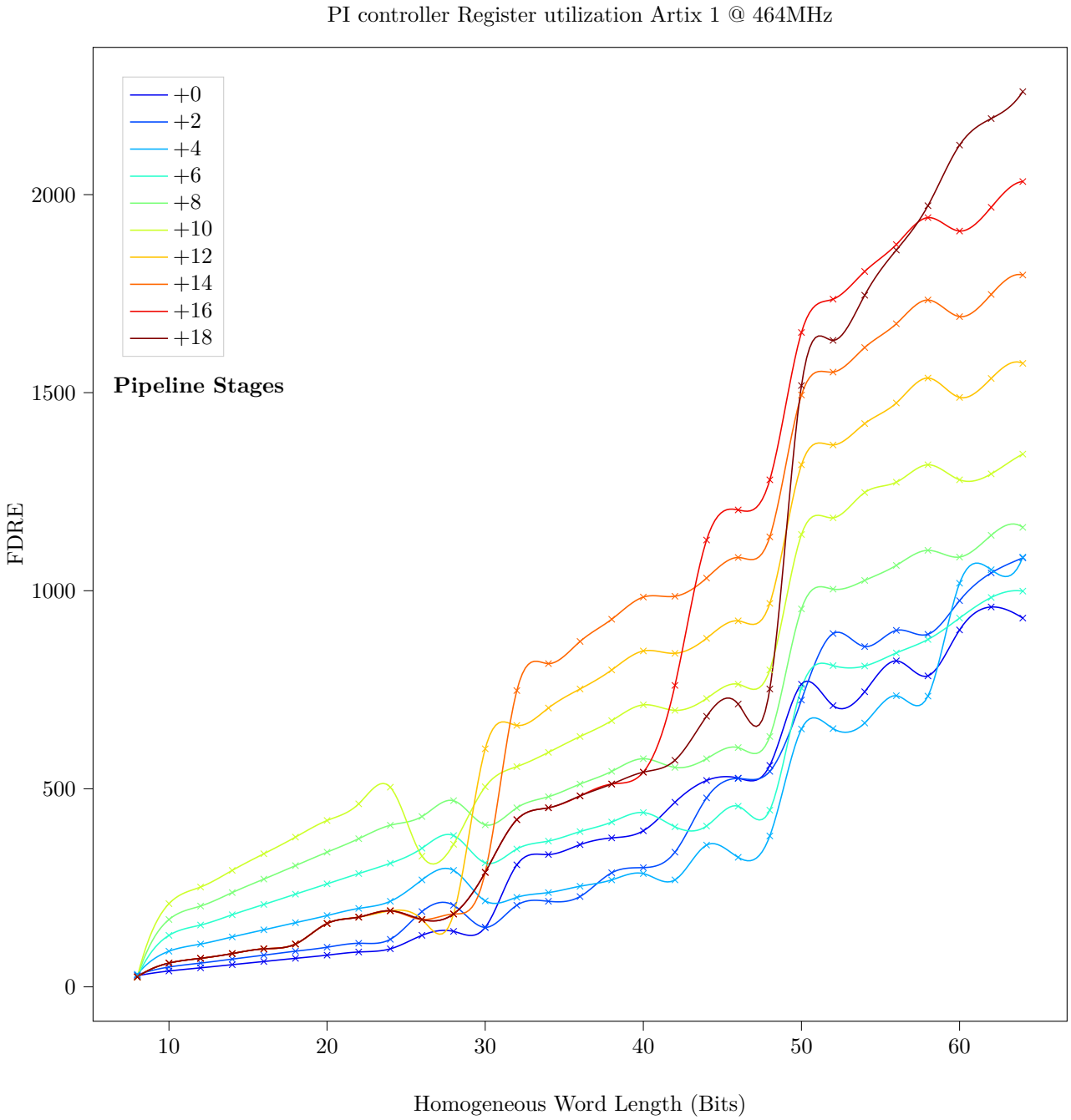


Fig. 3.16: *Register utilization increases with more pipeline stages Artix implementation.*

Looking at utilization in **Fig.** 3.16 the results are coherent with previously obtained measurements on Kintex Ultrascale+. Wider designs show higher utilization, and more pipelined designs use more registers. LUTs as logic usage is abnormally high in wide designs with few pipeline stages that don't meet timing.

Listing 3.1 contains the generic VHDL description of the system developed. **Listing** 3.2 is the generator script that allows us to perform the sequential sweep analysis over a variety of widths and pipeline depths and extract the desired metrics.

```
BEGIN -- Datapath
        PROCESS(clk)
        BEGIN
                IF rising_edge(clk) THEN
                        input_buff     <= input;
                        setpoint_buff  <= setpoint;

                        e0             <= e0_buff;
                        e1             <= e1_buff;
                        e2             <= e2_buff;

                        P_extra        <= P_extra(PIPE - 1 downto 0) & P_buff;
                        P_reg          <= P_extra(PIPE);
                        I_extra        <= I_extra(PIPE - 1 downto 0) & I_buff;
                        I_reg          <= I_extra(PIPE);
                        D_extra        <= D_extra(PIPE - 1 downto 0) & D_buff;
                        D_reg          <= D_extra(PIPE);

                        sum_reg        <= sum_buff;
                        sum_extra_buff <= sum_extra;

                        output_prev    <= output_buff;
                        output_reg     <= output_pipe;

                        error_buff     <= error_reg;
                        error_buff1    <= error_reg1;
                END IF;
        END PROCESS;

        -- Calculate error
                error_reg  <= setpoint_buff - input_buff;

        -- Error Delay
                error_reg1 <= error_buff;
                e0_buff <= error_buff1;
                e1_buff <= e0;
                e2_buff <= e1;

        -- Calculate Correction
                P_buff <= a0*e0;
                I_buff <= a1*e1;
                D_buff <= a2*e2;

        -- Accumulate
                sum_buff  <= P_reg(TRUNC) - I_reg(TRUNC);
                sum_extra <= sum_reg + D_reg(TRUNC);

        -- Apply Correction
                output_buff <= output_prev + sum_extra_buff;
                output_pipe <= output_prev;
                output      <= output_reg;
END Behavioral;
```

Listing 3.1: *PID datapath logic.*

```
script_head = '''
set_param general.maxThreads 16
open_project /home/nah/Documents/uni/muise/ada/PID/PID/PID.xpr
update_compile_order -fileset sources_1
set_property STEPS.SYNTH_DESIGN.ARGS.GATED_CLOCK_CONVERSION on [get_runs synth_1]
set_property strategy Flow_AlternateRoutability [get_runs synth_1]
set_property strategy Performance_Explore [get_runs impl_1]
'''
script_foot = '''
set_property generic {WIDTH=%s PIPE=%s} [current_fileset]
reset_run synth_1
reset_run impl_1
launch_runs impl_1 -jobs 16
wait_on_run impl_1
open_run impl_1
report_timing -file ./reports_width_pipe/report_time_%s_%s.txt
report_power -file ./reports_width_pipe/report_power_%s_%s.txt
report_utilization -file ./reports_width_pipe/report_util_%s_%s.txt
'''
#  report_pipeline_analysis -file ./reports_width_pipe/report_pipe_%s_%s.txt

#  gen tcl
f = open("report_widths_pipe.tcl", "w")
f.write(script_head)
count = 0
#  for pipe in range(12,13,2):
for pipe in range(2,11,2):
    for width in range(8,98,2):
        count = count + 1
        f.write("\nset_property STEPS.SYNTH_DESIGN.ARGS.RETIMING true [get_runs synth_1]")
        report = script_foot % (width,pipe,
                                width,pipe,
                                width,pipe,
                                width,pipe)
        f.write(report)
print(count)
f.write('quit')
f.close()
```

Listing 3.2: *Sweep generation script.*

V Discussion

Table 3.6: *Comparison between PID controllers found on scientific literature and our developed systems.*

Ref.	Width (bits)	Freq. (MHz)	LUTs	Registers	DSPs	FPGA
Zurita-Bustamante et al. [60]	8	60	8737	8722	-	Spartan 3E
Proposed Approach	8	464	0	28	4	Artix 7
Proposed Approach	8	891	32	69	2	Ultrascale+ -3
Tomov et al. [61]	12	160	-	-	-	Cyclone II
Ponce et al. [62]	12	400	-	-	-	Spartan 6
Proposed Approach	12	464	0	48	6	Artix 7
Proposed Approach	12	891	36	107	2	Ultrascale+ -3
Chan et al. [63]	16	50	1142	327	-	Spartan IIE
Martínez-Prado et al. [64]	16	82	3792	2998	-	Spartan 3
Proposed Approach	16	464	0	64	6	Artix 7
Proposed Approach	16	891	48	143	2	Ultrascale+ -3
Phan et al. [65]	32	100	2150	261	-	Zynq 7000
Proposed Approach	32	464	0	226	12	Artix 7
Proposed Approach	32	891	178	515	8	Ultrascale+ -3
Proposed Approach	64	464	64	2260	30	Artix 7
Proposed Approach	64	891	504	1233	24	Ultrascale+ -3
Proposed Approach	96	891	284	3161	60	Ultrascale+ -3

To benchmark our results we analyse several PID FPGA implementations found in scientific literature, although some resource utilization figures are not given in some cases. Designs range in their precision and operating frequencies. **Table** 3.6 summarises implementations proposed by others and compares them with our own approach. Lower precision systems are exemplified by Zurita-Bustamante et al. [60], they propose a system that occupies 8k LUTs and 8k registers operating at a maximum clock rate of 50 MHz on Spartan 3E while our approach uses 4 DSP slices, no LUTs and 28 registers, we achieve 464 MHz.

Tomov et al. [61] presents his high resolution PID implemented at 160 MHz and 12-bit while proposing the possibility of extending the functionality to 400 MHz at 14 bits. Ponce et al. [62] applied their expertise to computer numerical control machining managing 400 MHz with 12-bits of precision. At this width our proposed work manages to outperform their approach using Artix-7 FPGAs with a 464 MHz clock rate.

Reference designs with 16-bit widths realised by Martínez-Prado et al. [64] and Chan et al. [63] achieve 50 MHz using a modular approach, embedding the PID in a feedback temperature control system alongside an ADC, PWM and user interface. At this width we run our controllers at 891 MHz, exceeding other approaches.

The largest design in this comparison comes from Phan et al. [65], they develop a PID controller clocked at 100 MHz with 32-bits of precision tailored for robotic arm control, similar work by Martínez-Prado et al. [64] achieves 80 MHz at 16 bits. We can match their precision and exceed the clock rate by a factor of $\times 8.91$ while occupying fewer LUTs with 250 more registers. Higher precision systems are also proposed in this section, from 32 to 96-bits with clock rates of 891 MHz, to the best knowledge of the authors this is novel work.

A generic and parametrized logic architecture proposed for the implementation of digital PI controllers has been presented and analysed in depth. It proves to be robust enough to accommodate a wide range of precisions, from 8-bits at the lower end, appropriate for applications where power and resource utilization is of major concern, to 96-bit implementations with no performance penalty. Although resource utilization and power will be higher.

VI Chapter Conclusions

In this section we have proposed controllers operating at 891 MHz on Kintex and Virtex Ultrascale+ with widths from 8 to 96-bits, we also further prove the flexibility of our architecture by porting our design to the lower end of available FPGAs with Artix-7. Our Artix-7 implementations achieve 464 MHz (the upper bound limit of DSP slices) with a wide range of precision, from 8 to 64-bit. Further gains in precision could be found, at the cost of increasing the number of pipeline stages. Similarly as with Ultrascale+, we can also obtain optimal performance in regards to clock rate with the more constrained chip.

CHAPTER

4

SINGLE DSP48 COMPLEX MULTIPLICATION & CONSIDERATIONS FOR EFFICIENT FPGA IMPLEMENTATION

In this section we propose a novel 6-bit complex unsigned multiplier entirely contained within a single `DSP48` *Slice. As well as proposing and analysing other efficient complex multipliers for larger bit leghts achieving the maximum frequency supported by* Xilinx *Ultrascale+ FPGAs.*

I Introduction

Complex multiplication is one of the key operations in the field of digital signal processing, particularly for algorithms involving integral transforms. Field programmable gate arrays are an attractive option for implementing such algorithms, chiefly due to their massively parallel architecture. As such, optimal complex multiplication schemes are sought after, in particular methods that are simultaneously area efficient and can operate at high frequencies. Bleeding edge implementations requiere careful consideraron of FPGA resources, notably DSP slices, a limited and valuable resource included for both fast and power efficient fixed point Multiply-Accumulate (MAC) operations.

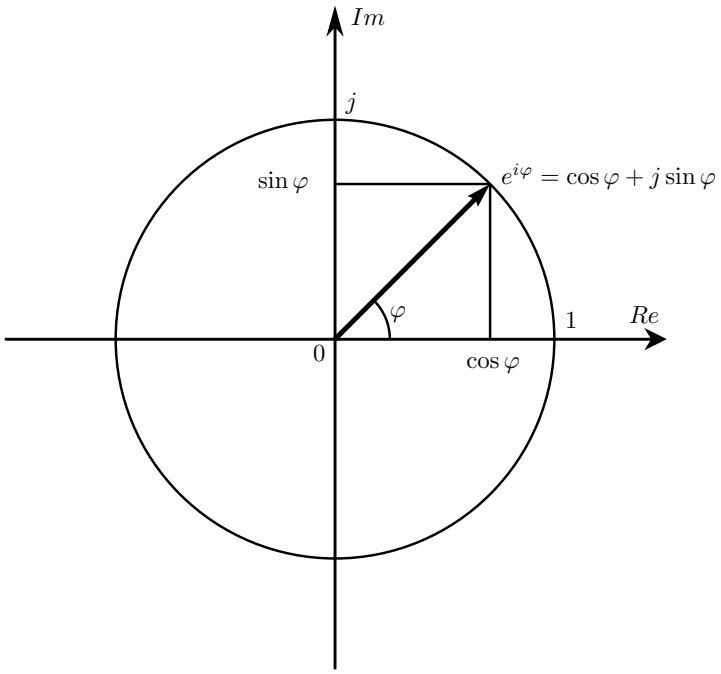


Fig. 4.1: *Argand diagram.*

Complex numbers ($\mathbb{C}$) extend real numbers ($\mathbb{R}$) this gives the real (Re) axis of the number line a new dimension, the imaginary (Im) axis, as is commonly depicted in Argand diagrams (**Fig.** 4.1). This newly formed complex plane can be indexed by a Cartesian coordinate system using an ordered pair of real numbers, a complex number z can thus be represented in Cartesian form by the number pair ($Re(z)$, $Im(z)$). The general form of the complex number corresponding to those Cartesian coordinates is represented as $z = Re(z) + j \cdot Im(z)$, where j the indeterminate that satisfies the defining condition $j^2 = -1$.

Addition in Cartesian form of two complex numbers, $a = x + y \cdot j$ and $b = u + v \cdot j$, can be computed by adding the Re and Im parts separately, such that $c = a + b = (Re(a) + Re(b)) + (Im(a) + Im(b)) \cdot j = (x + u) + (y + v) \cdot j$ leaving $Re(c) = x + u$ and $Im(c) = y + v$.

Multiplication of two complex numbers is similarly defined as $(x + y \cdot j) \cdot (u + v \cdot j) = (xu - yv) + (xv + yu) \cdot j$. It requieres four real multiplications an two real additions though it can be reduced to three real multiplications and five additions, this is sometimes termed the Gauss trick.

II State of the Art

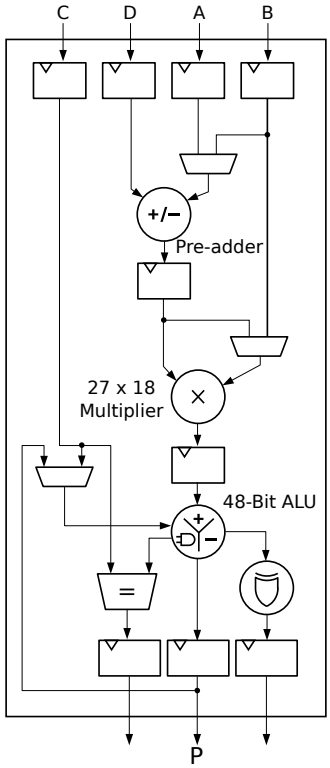


Fig. 4.2: *'Basic DSP48E2 Functionality' adapted from [6].*

DSP48E2 represents the highest performance digital signal processing slice included in XILINX FGPAs. At its core, it features a 27 × 18 bit signed multiplier, as well as a 27 bit pre-adder and a 48 bit post-accumulator logic unit. DSP48E2 also contains bypassable pipeline registers to achieve maximum performance. Not shown in **Fig.** 4.2 are carry signals internal to DSP columns, these are not accessible for general purpose use and are only used when implementing wide multiplications and additions with multiple DSP48E2 slices. The post multiplication logic unit features bitwise logic operations (AND, OR, NOT...).

DSP48E2 is a superset of DSP48E1, an earlier DSP architecture in XILINX products. The updated design is backwards compatible. The main differences are the width of the multiplier and pre-adder is increased from 25 to 27 bits.

This work builds on previous techniques and extends them to complex number operations, achieving new functionalities. XILINX has proposed optimizations for INT-8 operations mainly focused in Deep Learning. The white paper by Fu et al. [66] outlines a way of encoding three 8 bit inputs (a, b and c) into the ports of a DSP48 slice such that the multiplication result yields, in parallel, the products $a \times c$ and $b \times c$.

Further work has been presented by XILINX optimising INT-4 operations in the context of highly efficient Convolutional Neural Networks. Han et al. [67] introduces a tighter packing scheme for DSP48 slices where four 4 bit operands (a, b, c and d) are mapped to the input ports of the multiplier. The clever padding results in the output of the slice containing the product of the four multiplications, $(a \times c), (a \times d), (b \times c), (b \times d)$.

Table 4.1: *Integer multiplication packing.*

									a_8	a_7	a_6	0	0	0	a_2	a_1	a_0
									b_8	b_7	b_6	0	0	0	b_2	b_1	b_0
								b_0a_8	b_0a_7	b_0a_6	0	0	0	b_0a_2	b_0a_1	b_0a_0	
							b_1a_8	b_1a_7	b_1a_6	0	0	0	b_1a_2	b_1a_1	b_1a_0		
						b_2a_8	b_2a_7	b_2a_6	0	0	0	b_2a_2	b_2a_1	b_2a_0			
					0	0	0	0	0	0	0	0	0				
				0	0	0	0	0	0	0	0	0					
			0	0	0	0	0	0	0	0	0						
		b_6a_8	b_6a_7	b_6a_6	0	0	0	b_6a_2	b_6a_1	b_6a_0							
	b_7a_8	b_7a_7	b_7a_6	0	0	0	b_7a_2	b_7a_1	b_7a_0								
b_8a_8	b_8a_7	b_8a_6	0	0	0	b_8a_2	b_8a_1	b_8a_0									
c_{17}	c_{16}	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

The main multiplication packing scheme utilized in this work is thoroughly analysed by Huang et al. [68] and explained in detail in the following section. This technique packs four inputs (a, b, c and d) and outputs not only the result of two multiplications but also the sum of the other two. That is $(a \times c), (a \times d + b \times c), (b \times d)$.

The idea of using the techniques discussed above to implement a complex multiplier has been successfully implemented by Kamp et al. [39]. Their works is geared towards correlators for the Square Kilometre Array radio telescope. They manage to efficiently utilize DSP slices to compute complex multiplications, the only drawback of their implementation is that they occupy more than one DSP slice per complex multiplication. This work proposes a novel technique to pack all the operations required for a 6 bit complex multiplication into a one single DSP slice.

III Methods

Building on previous research, to pack a complete complex multiplication into a single DSP slice, we pack the four operands (Re_1, Re_2, Im_1,Im_2) into DSP the input ports padding with zeros as depicted in **Table** 4.1. Then we perform the subtraction needed for the real part of the product using the post-adder.

The lowest 8 bits will contain the product of the operands in the least significant bits of the inputs and the highest 8 bits similarly will contain the product of the multiplication of the most significant 8 bits of the input. The middle 8 bits of the multiplier output will contain the addition of the cross product between the least and most significant operands. To achieve the final optimization we feed back the lowest 8 bits padded with 16 zeros to the DSP input C so the highest 8 bits of DSP output P contain the difference between ($Re_1 \times Re_2$) and ($Im_1 \times Im_2$), this corresponds to the desired real part of the output. The imaginary part of the output is contained in the middle 8 bits.

In general, a multiplication of an n bit complex number (n bits for Re, n bits for Im) will require $3n$ bit wide inputs and $6n$. Given that the narrowest input for the DSP48 slice is 18 bits, the widest n we accommodate on

a single slice is $18/3 = 6$ bits wide.

To compare our proposed architecture we develop a conventional complex multiplier with four multipliers, as detailed earlier, and a slightly more efficient three multiplier approach:

$$X_1 = (a + b) \cdot c \tag{4.1}$$

$$X_2 = (d - c) \cdot a \tag{4.2}$$

$$X_3 = (c + d) \cdot b \tag{4.3}$$

$$Re(z) = X_1 - X_3 \tag{4.4}$$

$$Im(z) = X_1 + X_2 \tag{4.5}$$

IV Results

The proposed architecture is implemented with a generic width in a vendor neutral manner such that the single DSP complex multiplication is inferred by the tool and not hardwired by the designer. In order to achieve maximum frequency the DSP is pipelined with two stages of registers which get absorbed into the DSP slice, this results in our design having a two clock cycle latency. The advantage pipelining grants us is that we can now operate our circuit at the device's f_{max}, in the case of Xilinx Ultrascale+, this frequency is either 645, 775, or 891 MHz depending on speed grade according to Xilinx [69].

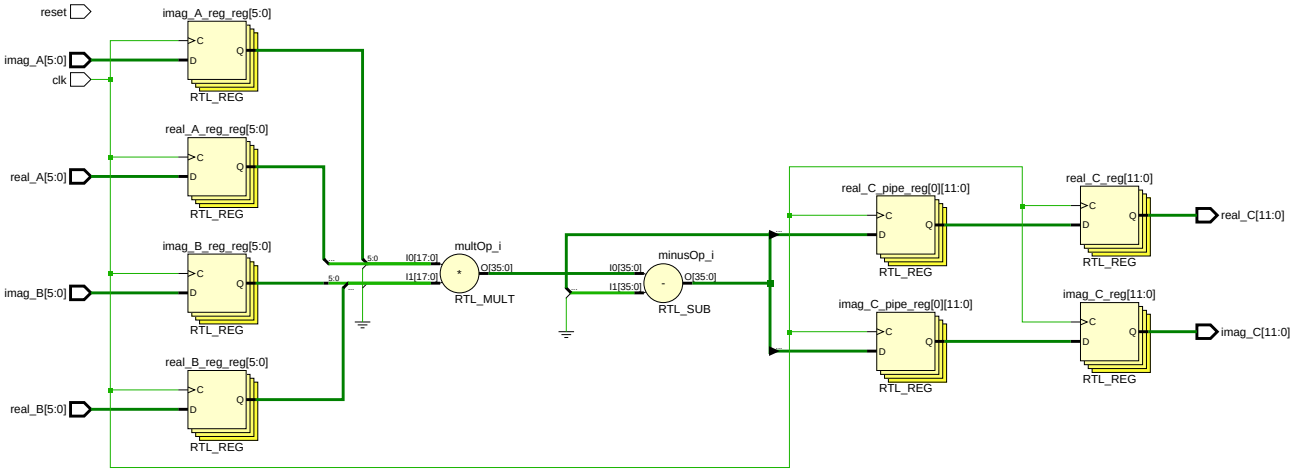


Fig. 4.3: *RTL circuit.*

On top of demonstrating the proof of concept single DSP implementation of our 6 bit complex multiplier, we proceed implementing a 4 bit version that similarly also only occupies one single DSP slice and a pure LUT variant. The latter manages a 66% reduction in LUT utilization. Both operate at the maximum possible frequency. Beyond 6 bits we demonstrate a 9 bit multiplier that utilizes 2 DSP48E2 and 18 LUTs, this represents a reduction in LUTs of 79%.

As seen in **Table** 4.2 Xilinx's own IP core complex multiplier only goes as narrow as 8 bits. At this width their implementation needs 3 DSPs, our proposed implementation at 8 bits uses one fewer DSP slice (33%

reduction), the same as our 9 bit design. Increasing the width to 16 bits, Xilinx's core occupies 12 DSPs and 258 LUTs, we manage area reductions of 25% for DSP slices and of 57% in LUTs.

Table 4.2: *Comparison implementations in Ultrascale+.*

Ref.	Bits	DSPs	LUTs	FF	$f_{max}(MHz)$
Kamp et al. [39]	4	0	260	147	470
Proposed Approach	4	0	88	32	645
Kamp et al. [39]	4	2	53	66	650
Proposed Approach	4	1	0	0	645
Kamp et al. [39]	6	2	64	92	650
Proposed Approach	6	1	0	0	645
Three Multiplier	6	0	181	0	645
Four Multiplier	6	0	201	0	645
Xilinx IP [70]	8	3	2	181	833
Proposed Approach	8	2	16	0	645
Kamp et al. [39]	9	2	86	54	640
Proposed Approach	9	2	18	0	645
Three Multiplier	9	0	380	0	645
Four Multiplier	9	0	377	0	645
Xilinx IP [70]	16	12	258	674	691
Proposed Approach	16	9	112	0	645
Three Multiplier	16	3	64	0	645
Four Multiplier	16	4	64	0	645
Proposed Approach	32	36	445	0	645
Three Multiplier	32	12	365	0	645
Four Multiplier	32	16	316	0	645

For wider multiplications the conventional four multiplier approach utilizes fewer resources than trying to extends the width of the single DSP approach and letting the tool infer a circuit. At 32 bits per real/complex part the three multiplier approach proves optimal, using 12 DSP slices.

V Discussion

Our proposed multiplier architecture can be used to implement low precision but large massively parallel systems. For this we demonstrate a parallel complex matrix multiplier of 9 bit unsigned numbers and analyse its design considerations. A parallel 9×9 complex matrix multiplier requieres $9^3 = 729$ simultaneous complex multiplications. Using a four multiplier approach 729 complex multiplications would translate into 2916 real multiplications. The three multiplier approach would need 2187 real multiplications. This amount of simultaneous multiplications is unfeasible, whether using LUTs or DSP slices it doesn't fit in the FPGA. At 9 bits wide, the approach demonstrated in this section is implemented as 1458 real multiplications, which can be implemented in FPGA. The proposed mapping achieves identical result while requiring fewer resources as can be seen in **Table** 4.3. At the chosen size, 9×9, placement of the three and four multiplication approaches fails and cannot be completed, only the single DSP proposed approach succeeds.

Table 4.3: *9 bit* 9×9 *complex matrix parallel multiplier.*

Approach	DSPs	LUTs	FF	$f_{max}(MHz)$
Proposed	1458	36k	0	600
	80%	15%	-	-
Three Mult.	0	312k	0	*
	-	144%	-	*
Four Mult.	0	328k	FF	*
	-	105%	-	*

* Design placement failed.

Listing 4.1 shows the VHDL code which is inferred as the proposed single DSP complex multiplier, the RTL of which is depicted in **Fig.** 4.3 and the implementation resource utilisation is given in **Table** 4.2 for a range of `WIDTH`s.

```
1  -- Inputs
2  in_A(  WIDTH-1 DOWNTO 0)       <= imag_A_reg;
3  in_A(3*WIDTH-1 DOWNTO 2*WIDTH) <= real_A_reg;
4  in_B(  WIDTH-1 DOWNTO 0)       <= imag_B_reg;
5  in_B(3*WIDTH-1 DOWNTO 2*WIDTH) <= real_B_reg;
6
7  -- Multiplication & Subtraction
8  out_reg <= (in_A * in_B) - (out_reg(2*WIDTH-1 DOWNTO 0) & (4*WIDTH-1 DOWNTO 0
       => '0'));
9
10 -- Outputs
11 real_C_reg <= out_reg(6*WIDTH-1 DOWNTO 4*WIDTH);
12 imag_C_reg <= out_reg(4*WIDTH-1 DOWNTO 2*WIDTH);
```

Listing 4.1: *Single DSP Complex Multiplier – Main Logic*

The generic sized complex matrix multiplier (**Lst.** 4.3) is instantiated as a set of parallel scalar products of the i^{th} row with the j^{th} column. Each dot product (**Lst.** 4.2) calls the proposed complex multiplier (**Lst.** 4.1) for each of the inputs, it calculates the Multiply Accumulate operations. The developed system takes an $(M \times N)$ and an $(N \times P)$ array of input signals and computes an $(M \times P)$ output array.

```
MULTS: FOR I IN 0 TO N GENERATE
        uut: complex
        GENERIC MAP(
                 WIDTH => WIDTH/2
        )
        PORT MAP(
                 real_A => input_1reg(I)(WIDTH-1     DOWNTO WIDTH/2),
                 imag_A => input_1reg(I)((WIDTH/2)-1 DOWNTO 0),
                 real_B => input_2reg(I)(WIDTH-1     DOWNTO WIDTH/2),
                 imag_B => input_2reg(I)((WIDTH/2)-1 DOWNTO 0),
                 real_C => real_C(I),
                 imag_C => imag_C(I),
                 clk    => clk,
                 reset  => reset);
        sum_buff_out (N/2)(I) <= sum_buff_out (N/2)(I+1) + (real_C(I));
        sum_buff_out1(N/2)(I) <= sum_buff_out1(N/2)(I+1) + (imag_C(I));
END GENERATE;
```

Listing 4.2: *Complex Dot Product Multiplier – Main Logic*

```
ROW:FOR I IN 0 TO M GENERATE
COL:FOR J IN 0 TO P GENERATE
        DOT_PRODUCT: dot_complex
        GENERIC MAP(
                WIDTH   => WIDTH,
                N       => N,
                PIPE    => PIPE
                )
        PORT MAP(
                input_1 => input_1reg(I)(N DOWNTO 0),
                input_2 => input_2reg(J)(N DOWNTO 0),
                clk     => clk,
                output  => output_in(I)(J),
                reset   => reset
                );
END GENERATE;
END GENERATE;
```

Listing 4.3: *Matrix Multiplier – Main Logic*

VI Chapter Conclusions

In this section we present a novel and optimal method of packing all operations needed for a 6 bit unsigned complex multiplication into one single DSP48. Our tweaked method manages to reduce DSP utilization by up to 50% in the case of 6 bit wide multiplication when compared to previous works. The method demonstrated also manages reductions of 79% of LUT utilization at 9 bits wide, though 2 DSP slices are still needed for this. Additionally we have developed and investigated other approaches for complex number multiplication. The need and benefits of our novel approach is demonstrated with parallel matrix multiplication, a resource intensive system.

CHAPTER

5

ANY-RADIX PARALLEL FOURIER TRANSFORM FPGA IMPLEMENTATIONS.

In this chapter we present the results of a thorough investigation of arbitrary and mixed radix parallel Fourier transforms implementations on FPGA. To do, so we develop a novel hardware description recursive approach for the logic architecture of the system using parametrized complex matrix multipliers. The approach presented in this chapter enables us to easily generate parallel Fourier transforms for an arbitrary number of points efficiently. Additionally, the architecture develops accepts aggressive pipelining strategies to meet strict timing requirements. The largest Fourier transforms demonstrated surpass the 100 GS/s throughput threshold.

I Introduction

Fourier transforms are a considered a staple of digital signal processing due to their wide range of applications. The Fourier transform (FT) and its inverse are mathematical tools which are used to analyse and synthesize signals in digital systems. Their fields of application range from, engineering, physics, chemistry, to computer science, medicine, biology and many others. Some examples of its real-world applications include:

- Digital Communication: the frequency content of digital signals produced by the FT is used for common tasks such as equalization, error correction, noise reduction, in wireless communication for Orthogonal Frequency Division Multiplexing (OFDM) in particular filter-based multicarrier (FBMC)[71].
- Image Processing: used to analyse the frequency components of images, which is used for image enhancement, noise reduction, compression by removing redundant information and to identify features or patterns in the image[72].
- Medical Exploration: in techniques such as Computed Tomography (CT)[73], Magnetic Resonance Imaging (MRI)[74], Positron emission tomography (PET)[75], Single photon emission computed tomography (SPECT) [76] and others to reconstruct images from measurements of the spatial frequency content of the body's tissues and organs. It proves to be a valuable tool in medical imaging that enables physicians to non-invasively visualize the internal structures and functions of the body, which can be useful for diagnosis and treatment planning.
- Financial Analysis: commonly used for financial time series data, commonly stock prices, in identifying trends and patterns that may not be immediately apparent in the raw data thus providing insights into the underlying dynamics of the market [77].
- Scientific Research: in astronomy[78], where data from telescopes and other instruments is processed to study celestial objects and phenomena, in seismology [79], analysing seismic data to study natural hazards such as earthquakes.

Software implementations of the Fourier transform for digital signal processing on general-purpose microprocessor offer the highest flexibility with the least amount of effort at the cost of throughput, resource and energy efficiency. Notable efforts include the Fastest Fourier Transform in the West (FFTW) software library by Frigo and Johnson [80], an open source parametrized FFT generator for a wide array of parameters and architectures. It consists of different algorithms for computing the discrete Fourier transform (DFT) in one or more dimensions. It is one of the most popular and widely used libraries for computing the DFT in CPUs. FFTW is highly optimized for modern processors, it is also highly scalable, allowing it to be used on a wide range of systems, from small embedded systems to large distributed many-core multi-node high performance computing platforms. FFTW is also highly portable, and can be used on a variety of platforms, including Linux, Mac OS X, and Windows.

The NVIDIA cuFFT[81] software toolkit is a GPU-accelerated library for computing FFTs. It is designed to provide high performance and scalability for a wide range of architectures. The library is optimized for use with

multiple NVIDIA GPUs, it provides several set of features, including support for single and double precision arithmetic, in-place (input array gets overwritten) and out-of-place transforms, and support for multidimensional transforms. It also provides a range of performance tuning options, including batching, tiling, and data layout optimisation. The library, like FFTW, is designed to be easy to use, with a simple API and comprehensive documentation. It is also highly portable, with support for a wide range of operating systems and programming languages. The library is available as part of the NVIDIA CUDA Toolkit[82], though it is not open-source like FFTW, it is free for academic and commercial use.

Several hardware topologies have been proposed to realise the Fourier transform efficiently in FPGAs and as ASICs. As is expected the increase in performance and efficiency comes at the cost of ease of development. Hardware architectures leverage two design techniques, spatial parallelism and temporal parallelism, sometimes termed time multiplexing or pipelining. Designs that skew heavily on time multiplexing yield systems that work in series, processing one sample every clock cycle, and taking as many cycles as samples are needed to complete the computation. On the other hand, designs that opt for spatial parallelism arrive at parallel implementations. They process multiple samples per cycle, when completely parallelised, all samples are processed in a single clock cycle. In-between this two extremes we find hybrid architectures, they find a balance between resource footprint and throughput by keeping some portions of the circuit parallel and others serialised. This trade-off allows for tailored implementations to satisfy the required specifications.

Serial hardware implementations boast a small footprint in terms of resources, at the expense of throughput. Pipelined implementations are built by concatenating butterfly processing elements alongside memory registers, each unit is reused for the computation of each sample[83, 84]. Parallel implementations offer the highest possible throughput, N samples processed per clock cycle for an N point FT, to do so, they require more resources.

i From the Fourier Series to the Fourier Transform

Fourier [85] established that any arbitrary continuous function $f(x)$ can be expressed as a trigonometric series. Lejeune Dirichlet [86] generalised this idea to piecewise-smooth functions, he stated that a Fourier Series will converge to $f(x)$ if in a finite interval it is "*periodic, single-valued and continuos except at a finite number of finite discontinuities, has a finite number of maxima/minima in one period and the integral over on period of* $|f(x)|$ *converges*" [87].

Any function can be expressed as a sum of even $[f(-x) = f(x)]$ and odd $[f(-x) = -f(x)]$ functions as:

$$\begin{aligned} f(x) &= \frac{f(x) + f(-x)}{2} + \frac{f(x) - f(-x)}{2} \\ f(x) &= f_{even}(x) + f_{odd}(x) \end{aligned} \tag{5.1}$$

An even function $f_{even}(x)$ can be expressed as sum of cosines:

$$f_{even}(x) = \sum_{n=0}^{\infty} a_n \cdot \cos n\omega_0 x \tag{5.2}$$

where ω_0 is some fundamental frequency. Similarly, odd functions $f_{odd}(x)$ as sum of sines as:

$$f_{odd}(x) = \sum_{n=0}^{\infty} b_n \cdot \sin n\omega_0 x \tag{5.3}$$

(5.2) and (5.3) are known as the synthesis equations. They can express a function $f(x)$ with period T and $\omega_0 = 2\pi/T$ as:

$$f(x) = \sum_{n=0}^{\infty} (a_n \cdot \cos n\omega_0 x + b_n \cdot \sin n\omega_0 x) \tag{5.4}$$

This is the trigonometric form of the Fourier synthesis equation. The terms a_n and b_n are the constant Fourier coefficients. (5.4) can be expressed more compactly thanks to Euler's formula [88] $e^{ix} = \cos x + i \sin x$ as:

$$f(x) = \sum_{n=-\infty}^{+\infty} c_n \cdot e^{jn\omega_0 x} \tag{5.5}$$

Which is the exponential form of the Fourier series expansion synthesis equation. The coefficients c_n can be expressed in terms of a_n and b_n and are given by the analysis equation:

$$c_n = \frac{1}{T} \int_T f(x) \cdot e^{-jn\omega_0 x} \mathrm{d}x \tag{5.6}$$

Now as the $T \to \infty$, the fundamental frequency $\omega = n\omega_0 = 2\pi/T$ becomes infinitesimally small thus $\mathrm{d}\omega = 2\pi/T$ and c_n becomes a continuous variable dependent on the frequency, thus (5.6) becomes:

$$\lim_{T\to\infty} c_n = c(\omega) \cdot \mathrm{d}\omega = \lim_{T\to\infty} \frac{1}{T} \int_T f(x) \cdot e^{-jn\omega_0 x} \mathrm{d}x \tag{5.7}$$

Taking into consideration $1/T = \mathrm{d}\omega/2\pi$:

$$\begin{aligned} c(\omega) \cdot \mathrm{d}\omega &= \frac{\mathrm{d}\omega}{2\pi} \int_T f(x) \cdot e^{-j\omega x} \mathrm{d}x \\ c(\omega) &= \frac{1}{2\pi} \int_T f(x) \cdot e^{-j\omega x} \mathrm{d}x \end{aligned} \tag{5.8}$$

We can substitute (5.8) into (5.5):

$$f(x) = \sum_{n=-\infty}^{+\infty} \frac{1}{2\pi} \int_T f(t) \cdot e^{-j\omega t} \mathrm{d}t \cdot e^{jn\omega_0 x} \tag{5.9}$$

and substitute $n\omega_0$ once again taking into consideration it becomes infinitesimal as $T \to \infty$ and rewriting the summation of infinitesimals as an integral (using the Riemann definition of an integral as the limit of a sum):

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} e^{j\omega x} \mathrm{d}\omega \int_{-\infty}^{+\infty} f(t) \cdot e^{-j\omega t} \mathrm{d}t \tag{5.10}$$

Which is the Fourier Integral Theorem expressed in its exponential form. It is the fundamental underlying integral transform pairs. We can proceed our analysis by defining a pair of equations (5.11) and (5.12) such

that the theorem (5.10) is satisfied. This transform pair is the Fourier Transform (FT) and Inverse Fourier Transform (IFT). Formally [89], the FT decomposes a complex-valued function into its constituent frequencies. Temporal functions will thus be expressed as a function of frequencies, the inverse also holds true. Though functions of any domain and any dimensions may be transformed, Fourier analysis is often carried out on one dimensional time domain functions. We continue using the notation $x(t)$ for a function in continuous time whose FT is defined as:

$$X(\omega) = FT\{x(t)\} = \int_{-\infty}^{+\infty} x(t) \cdot e^{-j\omega t} \cdot \mathrm{d}t \tag{5.11}$$

Where $X(\omega)$ is a function in the frequency domain. For the inversion theorem to hold true then the following expression is defined such that $x(t) = FT^{-1}\{FT\{x(t)\}\}$:

$$x(t) = FT^{-1}\{X(\omega)\} = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) \cdot e^{j\omega t} \cdot \mathrm{d}\omega \tag{5.12}$$

This is know as the inverse FT, used in Fourier synthesis. It can be checked by substitution that (5.10) holds. We see by inspection that the FT is the convolution of the input function and the complex exponential. Similarly, for an aperiodic discrete time signal $x[n] \to X(\omega)$, the Discrete Time Fourier transform is expressed as:

$$X(\omega) = DTFT\{x[n]\} = \sum_{n=-\infty}^{+\infty} x[n] \cdot e^{-j\omega n} \tag{5.13}$$

Which is periodic with period 2π as a consequence of the complex exponential function. It is defined in the continuous frequency domain and therefore contains an infinite number of harmonics. Given the periodicity of $X(\omega)$, all information carried by the signal is contained within the samples of the fundamental frequency $0 \leq \omega < 2\pi$. We can take N equidistant samples in this interval, with spacing $2\pi/N$. We can now evaluate $X(\omega)$ at $\omega = 2\pi k/N$:

$$X\left(\frac{2\pi k}{N}\right) = \sum_{n=-\infty}^{+\infty} x[n] \cdot e^{-j\frac{2\pi nk}{N}}, \qquad k = 0, 1, \cdots, N-1 \tag{5.14}$$

which can be divided into an infinite number of N length summations:

$$X\left(\frac{2\pi k}{N}\right) = \cdots + \sum_{n=-N}^{-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}} + \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}} + \sum_{n=N}^{2N-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}} + \cdots \tag{5.15}$$

whose summation interval can be factorised as iterating from $l \cdot N$ to $N \cdot (l+1) - 1$ with $l \in \mathbb{Z}$ and compactly written as:

$$X\left(\frac{2\pi k}{N}\right) = \sum_{l=-\infty}^{\infty} \sum_{n=l\cdot N}^{N\cdot(l+1)-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}} \tag{5.16}$$

within the summation we substitute the variable n with $n - l \cdot N$ taking into consideration the periodicity of the complex exponential function to simplify the indexes and switch the order of the summations:

$$X\left(\frac{2\pi k}{N}\right) = \sum_{l=-\infty}^{\infty} \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}} = \sum_{n=0}^{N-1} \left(\sum_{l=-\infty}^{\infty} x[n - l \cdot N] \right) \cdot e^{-j\frac{2\pi nk}{N}} \tag{5.17}$$

and we define $x_p[n] = \sum_{l=-\infty}^{\infty} x[n - l \cdot N]$ such that:

$$X\left(\frac{2\pi k}{N}\right) = \sum_{n=0}^{N-1} x_p[n] \cdot e^{-j\frac{2\pi nk}{N}} \tag{5.18}$$

$x_p[n]$ is an N-periodic sequence aliased version of $x[n]$. In the case where $x[n]$ is a finite signal of length $L \leq N$, $x_p[n]$ will simply contain a repetition of $x[n]$ in each periodic interval padded with $(N - L)$ zeros. Therefore, from $x[n] \equiv x_p[n]$ over a single period we deduce:

$$X[k] \equiv X\left(\frac{2\pi k}{N}\right) = DFT\{X[k]\} = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi nk}{N}}, \qquad k = 0, 1, \cdots, N-1 \tag{5.19}$$

This is the definition of the Discrete Fourier Transform. Whose inverse is given by:

$$x[n] = DFT^{-1}\{X[k]\} = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{j\cdot\frac{2\pi nk}{N}}, \qquad n = 0, 1, \cdots, N-1 \tag{5.20}$$

The DFT is a sampled DTFT at a finite number of equidistant frequency points on the interval $[0, 2\pi]$. Though N may be made arbitrarily large it will always be finite, therefore it lends itself to be calculated on digital devices using a non-infinite number of adders and multipliers.

The main features and objectives of this work are:

- Arbitrary size any-radix parallel Fourier transform architecture design, validation and implementation in FPGA.
- Pipeline architecture to obtain the highest clock rates achievable (891 MHz) on Virtex 7 Ultrascale+ Speed Grade -3.
- Comparison in resource utilization between differently sized Fourier transforms according the number of points and radix.

II Theoretical Background

The discrete Fourier transform (5.19) of N points can be compactly expressed in matrix form as the product of an N length input vector, x and an $N \times N$ transform matrix T_N:

$$DFT_N(x) = x \cdot T_N \tag{5.21}$$

Where each entry $(i, j) = (n, k)$ of the transform matrix T_N corresponds to the complex valued Fourier coefficient computed as:

$$T_{N\,ij} = e^{-j \cdot \frac{2\pi \cdot n \cdot k}{N}} = W_N^{nk} \tag{5.22}$$

$$T_N = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & \cdots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)(N-1)} \end{pmatrix} \tag{5.23}$$

In linear algebra [90], (5.23) is an example of a Vandermonde matrix, named after Alexandre-Théophile Vandermonde, who introduced it in 1771 while studying the interpolation of polynomials [91]. A Vandermonde matrix (V) is a matrix that has the terms of a geometric progression in each row. The elements $V_{i,j} = x_i^{j-1}$, where $x_1, x_2, ..., x_n$ are the elements of a given vector $\mathbf{x}$. For example, when $\mathbf{x} = [1, 2, 3]$, then the corresponding Vandermonde matrix is:

$$V_{\mathbf{x}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \tag{5.24}$$

In this case, (5.23) is the Vandermonde matrix of the roots of unity. A root of unity is a complex number z such that $z^n = 1$ for some positive integer n. For example, when $n = 2$, the roots of unity are 1 and -1. When $n = 3$, the roots of unity are $z = 1$, $\frac{-1+\sqrt{3}j}{2}$, and $\frac{-1-\sqrt{3}j}{2}$ and for $n = 4$, the solutions to $z^4 = 1$ are $1, j, -1, -j$. These roots of unity have important properties and applications in various fields of mathematics, including algebraic geometry and number theory [92] They also have applications in physics, such as in the study of quantum mechanics [93]. In addition to its applications in interpolation [94], the Vandermonde matrix also has applications in linear algebra and numerical analysis, such as the computation of the matrix inverse and the solution of linear systems [91], the matrix inverse of (5.23) corresponds to the inverse DFT (5.20).

Using the $n = 2$ roots of unity, the two-point discrete Fourier transform matrix T_2 is:

$$T_2 = \begin{pmatrix} 1 + 0j & 1 + 0j \\ 1 + 0j & -1 + 0j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{5.25}$$

Likewise, three and four-point transform matrices T_3 and T_4 correspond to:

$$T_3 = \begin{pmatrix} 1+0j & 1+0j & 1+0j \\ 1+0j & -0.5-0.866j & -0.5+0.866j \\ 1+0j & -0.5+0.866j & -0.5-0.866j \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -0.5-0.866j & -0.5+0.866j \\ 1 & -0.5+0.866j & -0.5-0.866j \end{pmatrix} \tag{5.26}$$

$$T_4 = \begin{pmatrix} 1+0j & 1+0j & 1+0j & 1+0j \\ 1+0j & 0-1j & -1+0j & 0+1j \\ 1+0j & -1+0j & 1+0j & -1+0j \\ 1+0j & 0+1j & -1+0j & 0-1j \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \tag{5.27}$$

As is evident from this matrix representations of the different Fourier transforms the computation of an N length DFT requieres one vector-matrix product of size $(1 \times N) \cdot (N \times N)$ which has a computational complexity of the order of $\mathcal{O}(N^2)$, far from ideal. A single point of an N point DFT requires N complex multiplications and $N-1$ complex additions. An N point DFT requires N^2 complex multiplications and $N \cdot (N-1)$ additions.

Additionally, by inspection, we see that in some cases, the Fourier transforms are composed primarily of trivial multiplications, mainly by ± 1 and $\pm j$, this is of great importance since implementation of such multiplications requiere negligible hardware.

Efficient Fourier implementations can be obtained by carefully factorising the transform matrix to our advantage. Thanks to the symmetry of this system, we can minimize the number of operations that are repeated. These factorisations are shown in works by Pease [95], Sloate [96], Cortés et al. [97], and produce the following recursive matrix form Fourier transform which expresses the Cooley and Tukey [98] algorithm.

$$T_N = P_N^{(r)} \cdot [I_r \otimes T_{N/r}] \cdot D_N^{(r)} \cdot [T_r \otimes I_{N/r}] \tag{5.28}$$

The symbol $\otimes$ denotes the Kroenecker product, the generalization of a scalar-matrix multiplication. $P_N^{(r)}$ is termed the permutation matrix. D_N^r is corresponds to the diagonal matrix containing the Fourier coefficients. $I_{N/r}$ is the N/r identity matrix, where N is the number of points, and r is the radix.

This generalises the Fourier transform for any N number of points which are calculated recursively as any arbitrary smaller transforms of size N/r. The total number of operations is reduced leading to complexities of the order of $\mathcal{O}(N \cdot \log_2(N))$ in some cases.

The complexity concisely expressed in (5.28) is the key that enables us to very naturally and very simply generate parallel hardware implementations of factorised Fourier transforms. Thanks to this expression we can define the digital logic architecture for an arbitrary number of points with an arbitrary number of factorisation stages without adding any computational overhead to the Cooley and Tukey [98] algorithm.

III Proposed Approach

The 4 point Fourier transform (5.27) can be factorised according to (5.28) to produce (5.29) using the 2 point Fourier transform (5.25). The 4 point Fourier decomposition illustrates the expected particularly efficient behaviour, as all entries of the matrices are ± 1 or $\pm j$ they can be implemented with very little hardware. Higher order Fourier transforms are then built using deeper levels of recursion. In expansion (5.29), the first matrix corresponds to the permutation matrix, which can be implemented in hardware by carefully ordering signal connections without needing a matrix multiplication. The second matrix corresponds to the $[I_r \otimes T_{N/r}]$ term of (5.28), for which $T_{N/r}$ is the transform matrix that has to be calculated recursively. The next matrix is $D_N^{(r)}$ which contains the Fourier coefficients (5.22). The last matrix is $[T_r \otimes I_{N/r}]$, it is dependent on the number of points N and the radix r of each stage. The entries of the last two matrices are precomputed and added to our designs as constants, this allows the implementation tool to simplify trivial multiplications and eliminate multiplications by zero entirely.

$$T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -j \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \tag{5.29}$$

$$T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} T_2 & & 0 & 0 \\ & & 0 & 0 \\ 0 & 0 & & \\ 0 & 0 & & T_2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -j \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \tag{5.30}$$

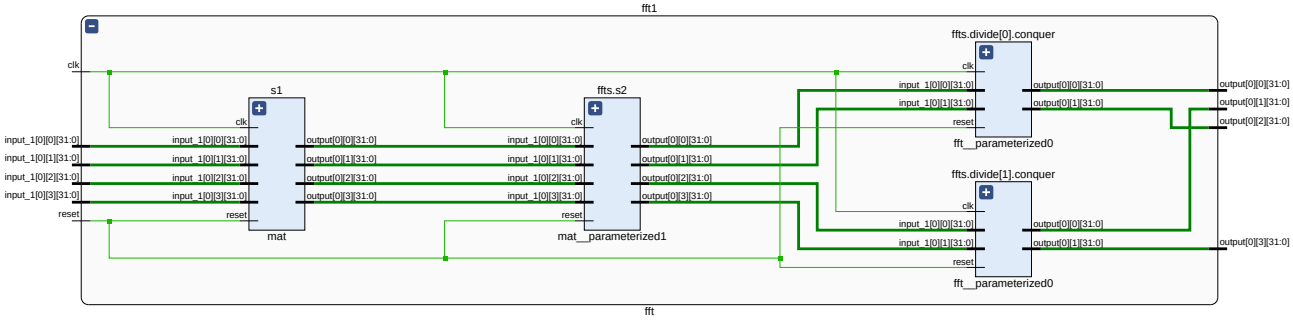


Fig. 5.1: *T_4 inferred RTL.*

Fig. 5.1 shows the T_4 (5.27) inferred RTL, it features two matrix multiplications which are then divided into two T_2 (5.25) matrices. The output of these signals is then rearranged as to satisfy the permutation requirements, this can be done programmed in VHDL without requiring an additional matrix multiplication. The call to the lower order FFT is done recursively, the same module instantiates itself with different values for the generic parameters. This same idea is carried forward with an 8-point FFT, T_8, **Fig.** 5.2 which is devided into two 4-point FFTs, T_4 as previously drawn in **Fig.** 5.1, each of which is in term divided into two 2-point FFTs, T_2.

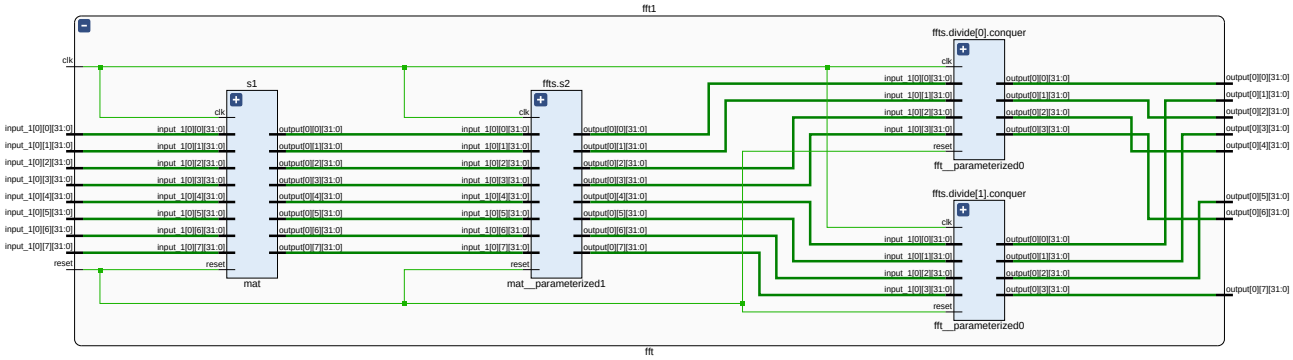


Fig. 5.2: *T_8 divided into two T_4 FFTs.*

Fig. 5.3 shows one of the two 4-point FFTs expanded. It contains the previously discussed system which contains two 2-point FFTs. The proposed IP core generates the permutation of the output signals at each stage.

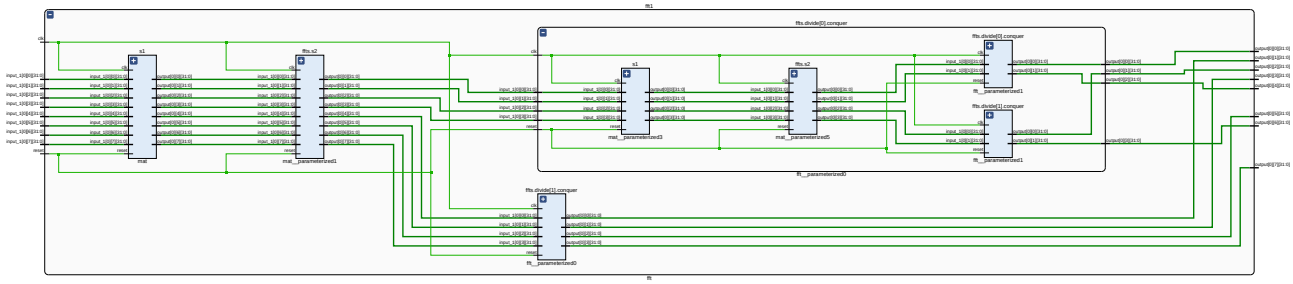


Fig. 5.3: *T_8 expanded to show recursive inclusion of **Fig.** 5.1.*

In the case of the 8-point FFT it can be factorised into two 4-point FFTs each of which is calculated as two 2-point FFTs as previously shown or it can be factorised as four 2-point FFTs directly as shown in **Fig.** 5.4, these factorisations are the radix-2 and radix-4 decompositions.

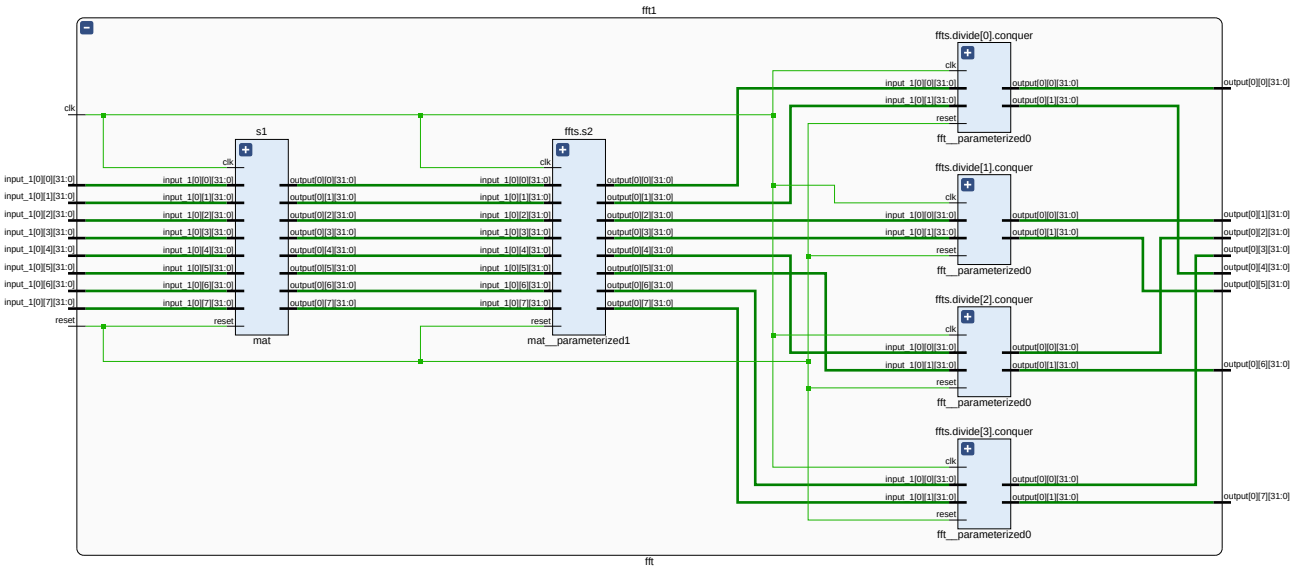


Fig. 5.4: *T_8 divided into four T_2 FFTs.*

The flexibility of our approach allows us to tackle any arbitrary number points decomposed in whatever possible radices. Traditional approaches have been restricted to powers-of-2. In **Fig.** 5.5 we demonstrate a 9-point FFT factorised into three 3-point FFTs.

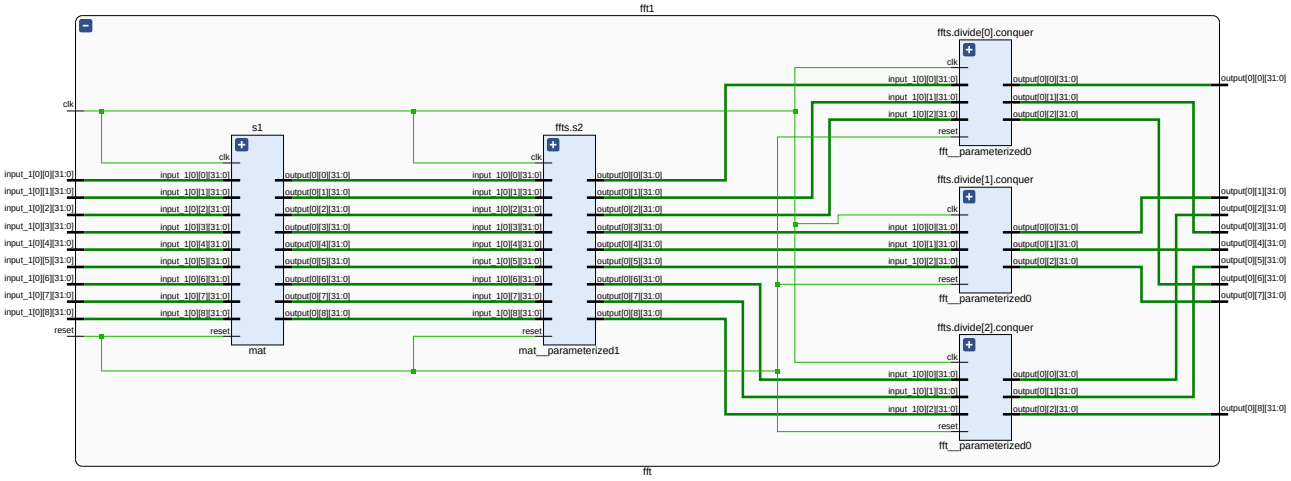


Fig. 5.5: *T_9 divided into three T_3 FFTs.*

Continuing with our demonstration, 10-point FFT can be factorised either as two 5-point FFTs (**Fig.** 5.6) or as five 10-point FFTs (**Fig.** 5.7).

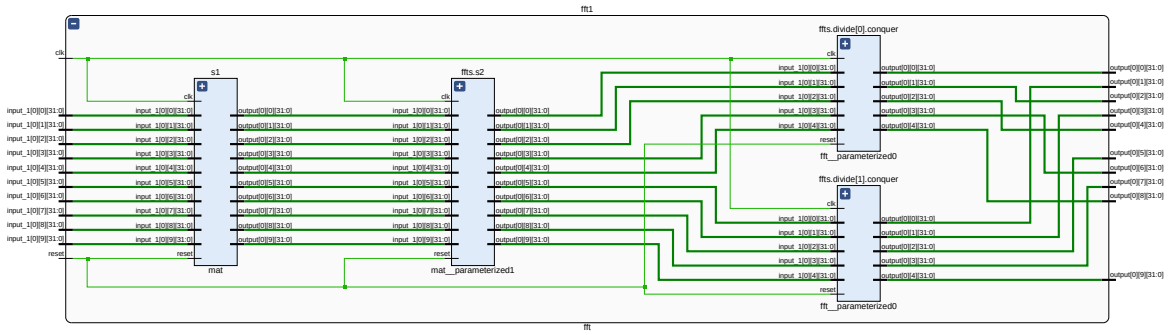


Fig. 5.6: *T_{10} divided into two T_5 FFTs.*

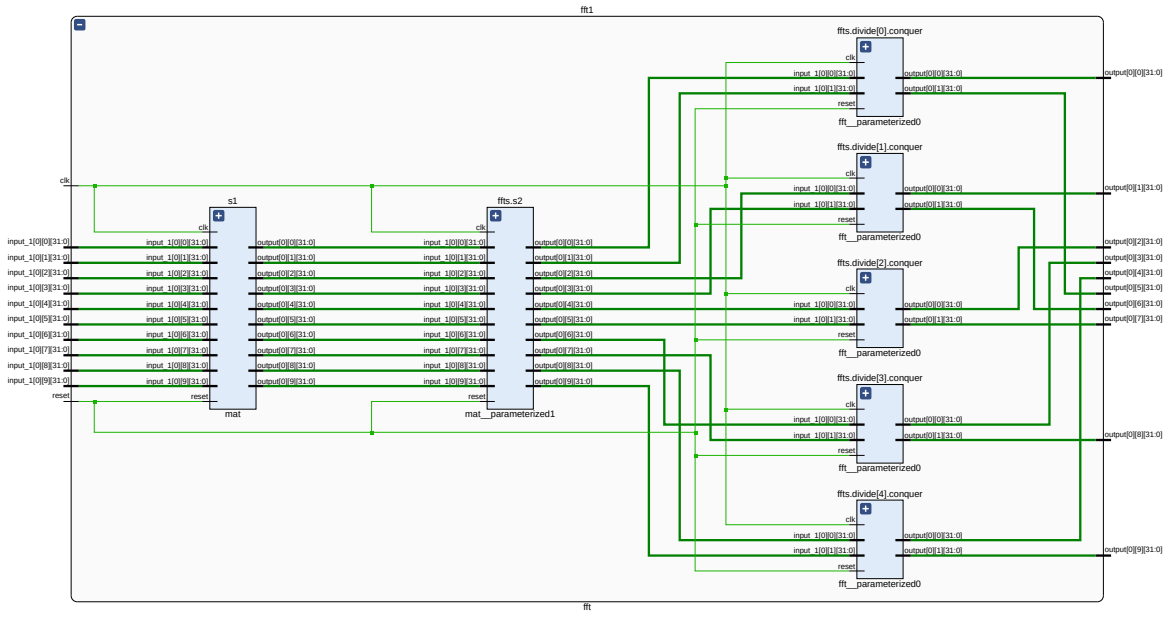


Fig. 5.7: *T_{10} divided into five T_2 FFTs.*

The 12-point FFT (**Fig.** 5.8) can be reduced to two 6-point FFTs (**Fig.** 5.9), each of which is computed as two 3-point FFTs. The 12-point FFT may also be computed as four 3-point FFTs **Fig.** 5.11 or as three 4-point FFTs **Fig.** 5.12.

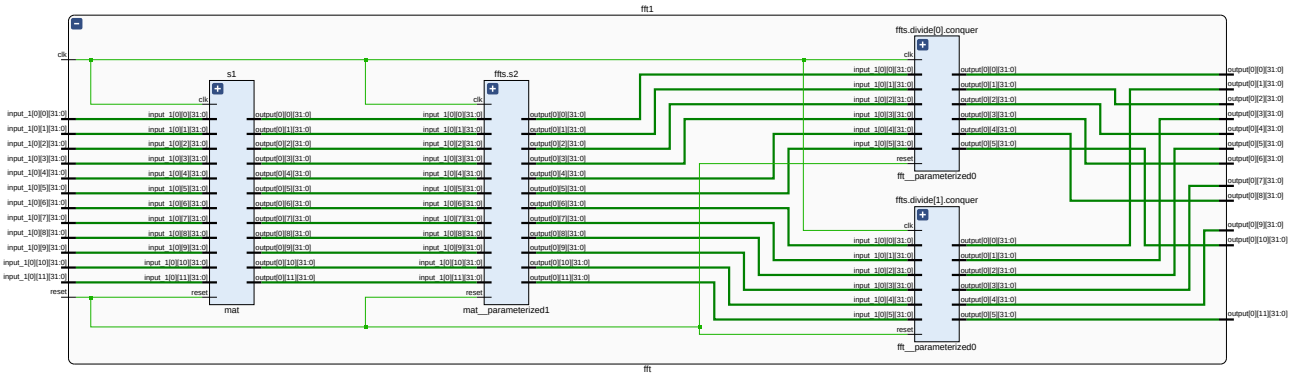


Fig. 5.8: *T_{12} divided into two T_6 FFTs.*

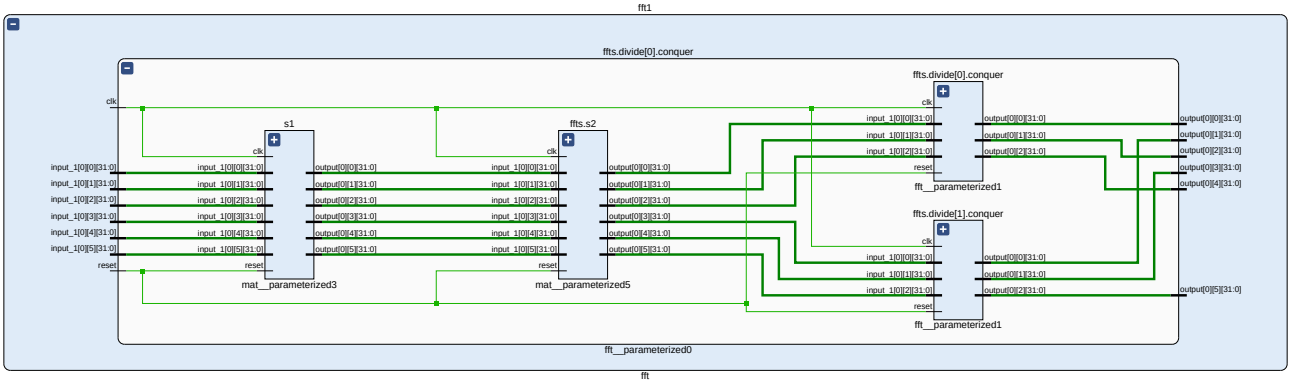


Fig. 5.9: *T_6 from **Fig.** 5.8 divided into two T_3 FFTs.*

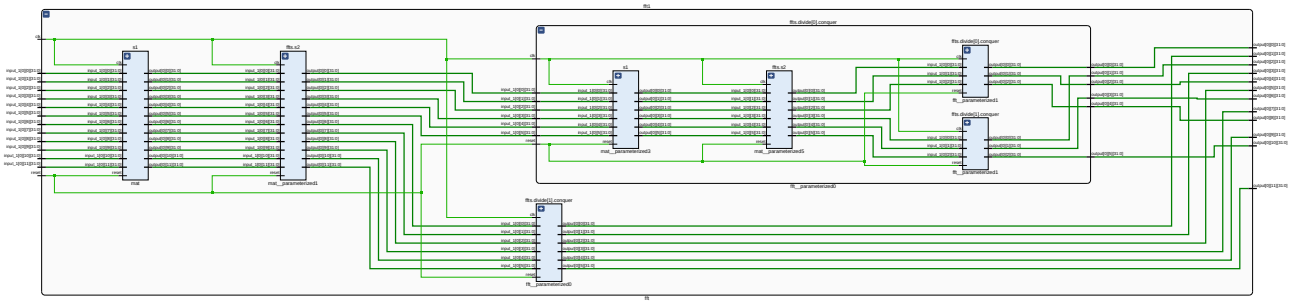


Fig. 5.10: *T_{12} (**Fig.** 5.8) expanded into two T_6 (**Fig.** 5.9) divided into two T_3 FFTs.*

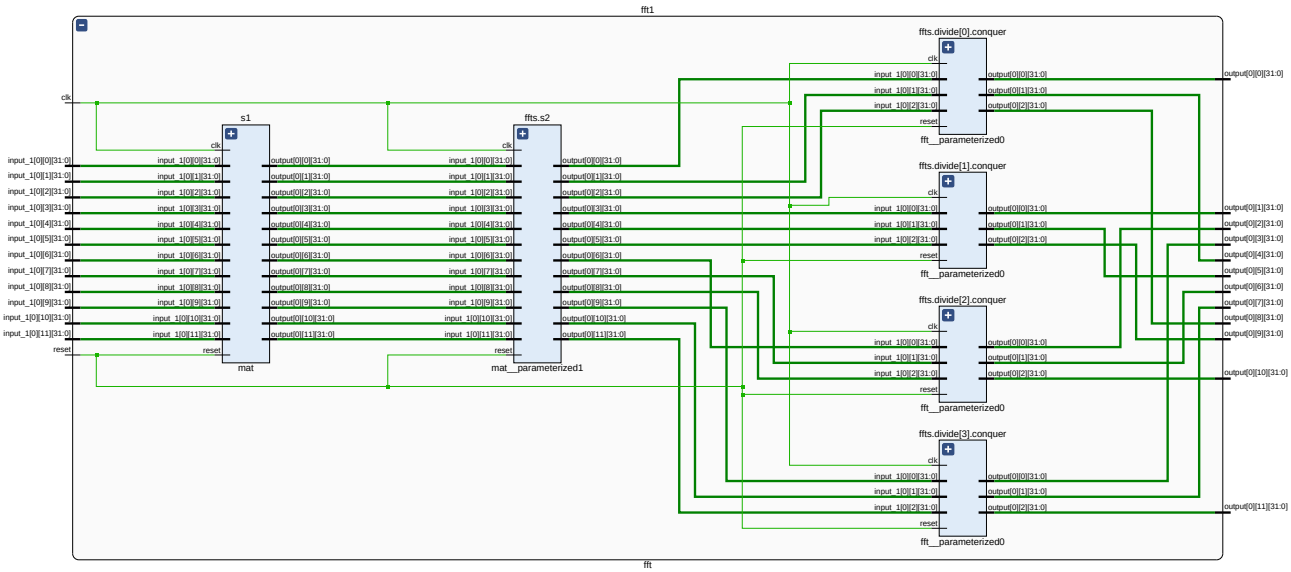


Fig. 5.11: *T_{12} divided into four T_3 FFTs.*

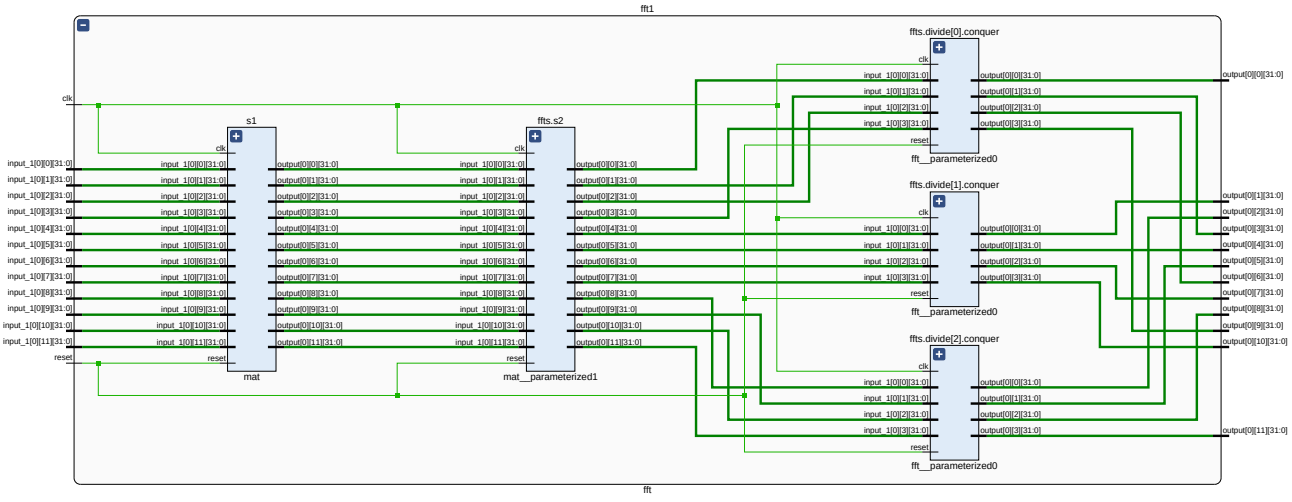


Fig. 5.12: *T_{12} divided into three T_4 FFTs.*

i 64 point radix 2 FFT RTL

This is the RTL circuit generated by the proposed architecture, the first stage features 64 inputs, each successive stage is divide in half. **Fig.** 5.13 illustrates the highest level, with each preceding figure showing the contents of the recursive FFTs instantiated.

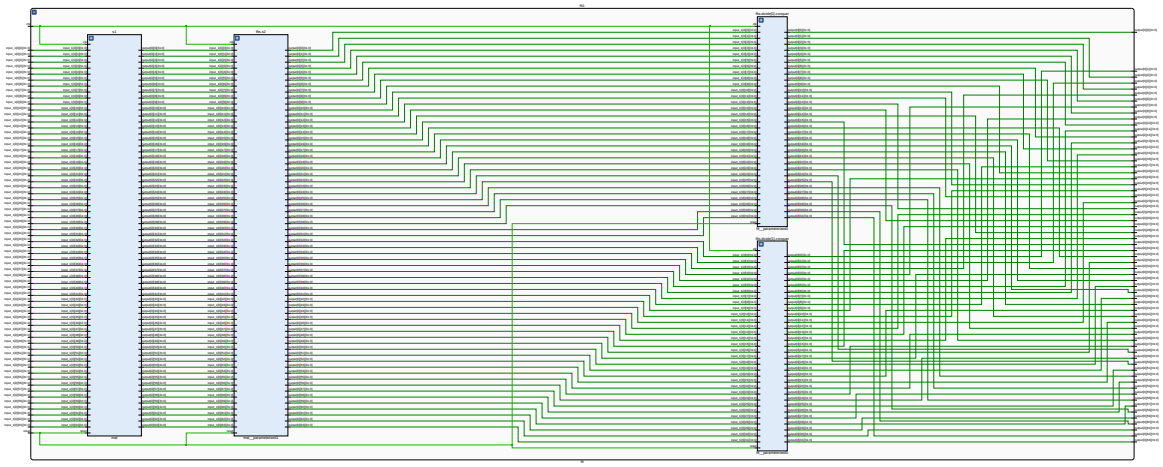


Fig. 5.13: *T_{64} divided into two T_{32} FFTs.*

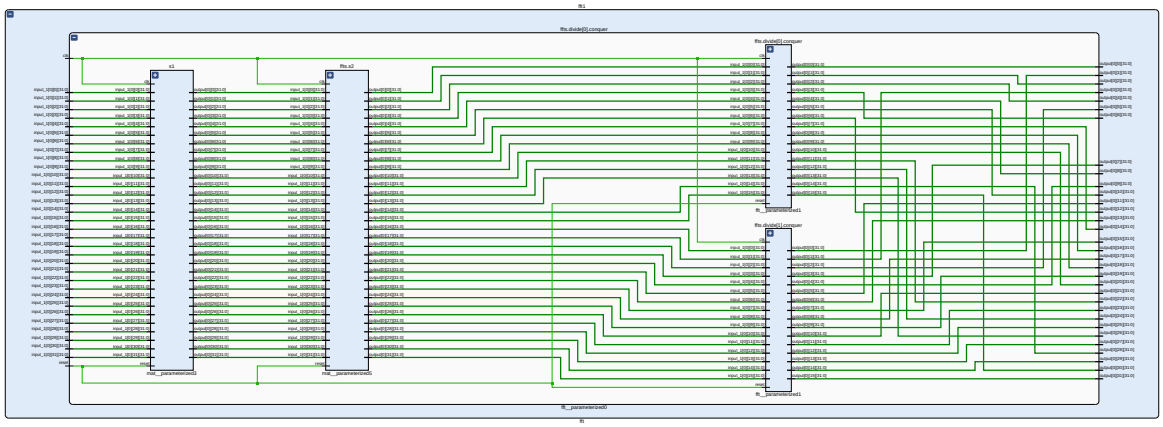


Fig. 5.14: *T_{32} divided into two T_{16} FFTs.*

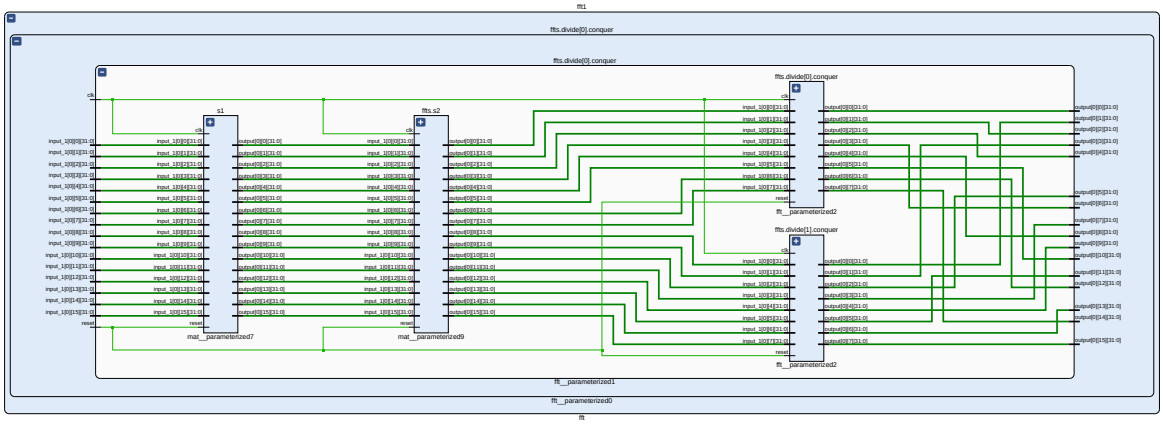


Fig. 5.15: *T_{16} divided into two T_{8} FFTs.*

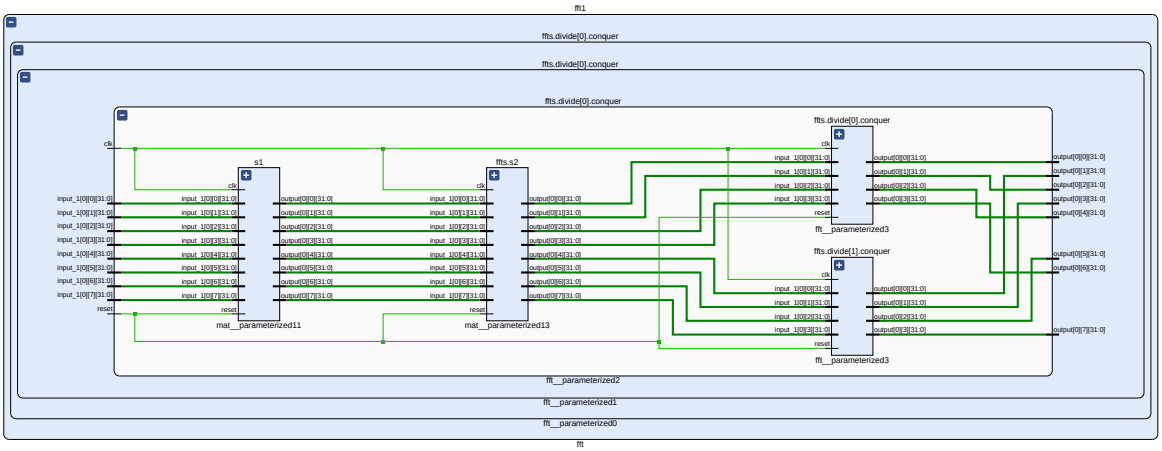


Fig. 5.16: *T_8 divided into two T_4 FFTs.*

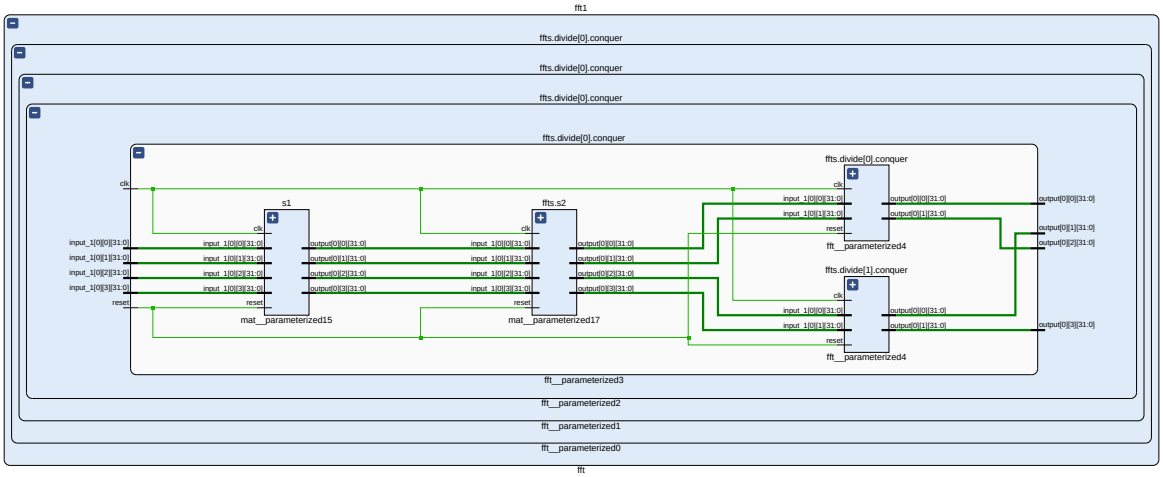


Fig. 5.17: *T_4 divided into two T_2 FFTs.*

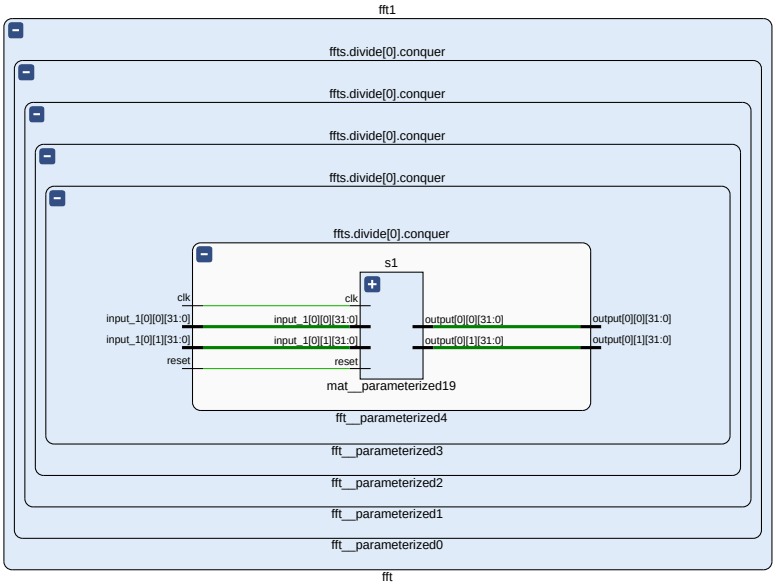


Fig. 5.18: *T_2 final stage matrix multiplier.*

ii 81 point radix 3 FFT RTL

This section illustrates the high level architecture signal routing generated for an 81-point radix 3 parallel FFT. The first 81-point contains three 27-point FFTs, each of which contain three 9-point FFTs which contain three 3-point FFTs.

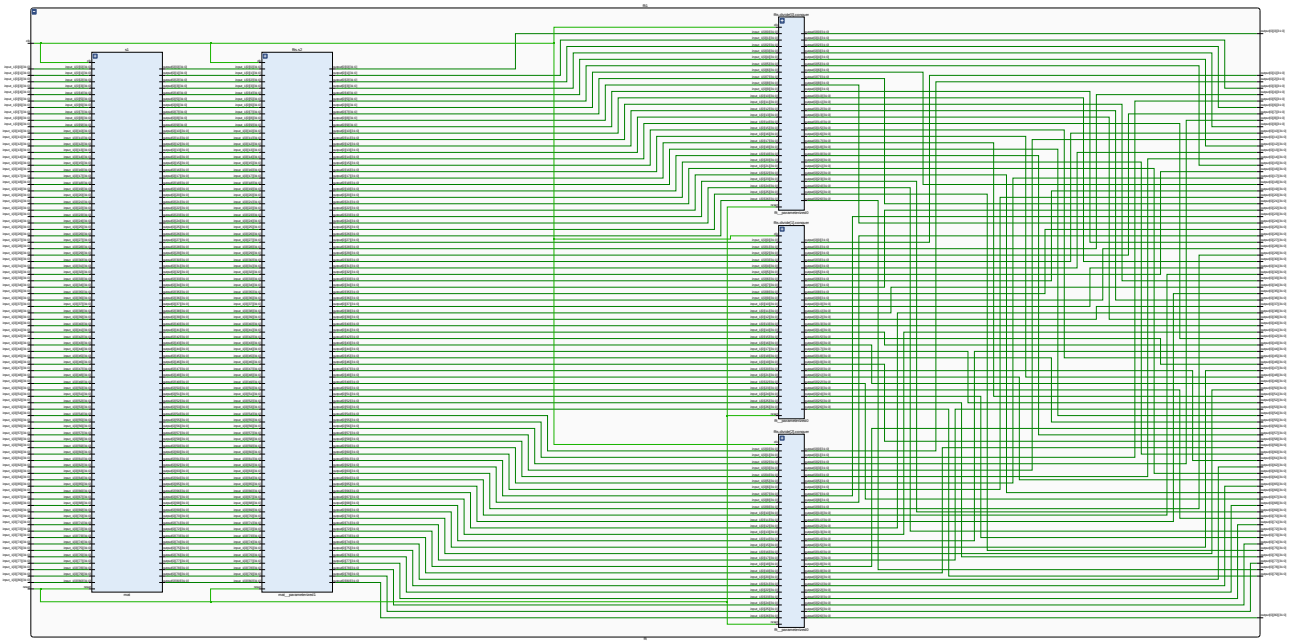


Fig. 5.19: *T_{81} divided into three T_{27} FFTs.*

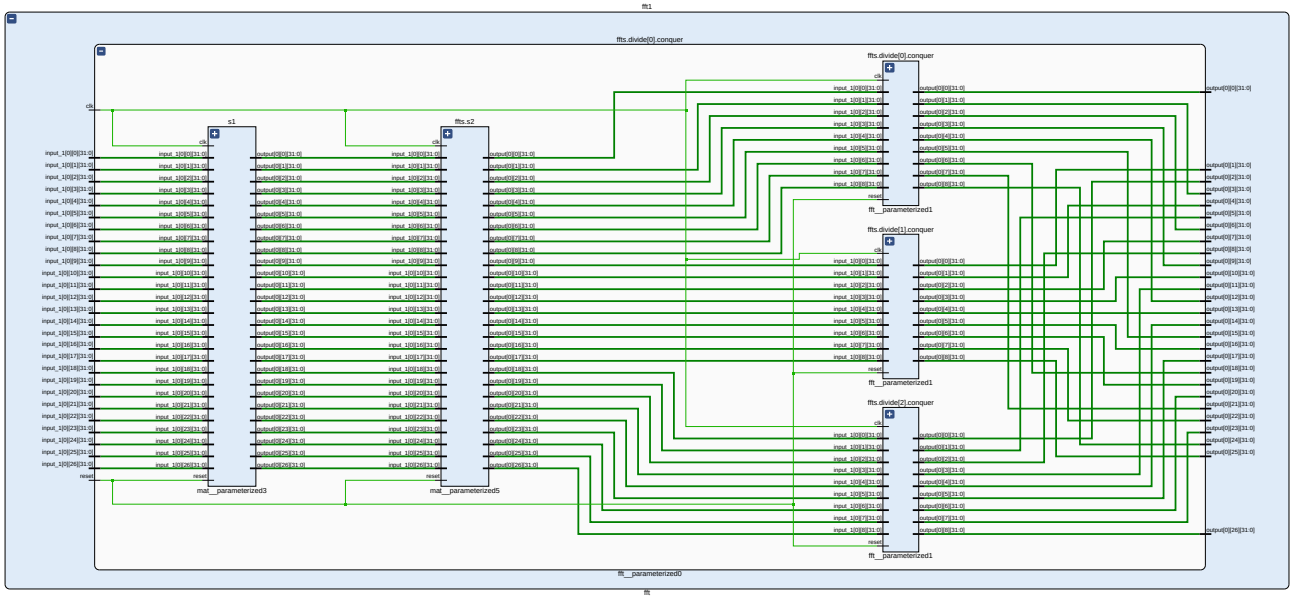


Fig. 5.20: *T_{27} divided into three T_9 FFTs.*

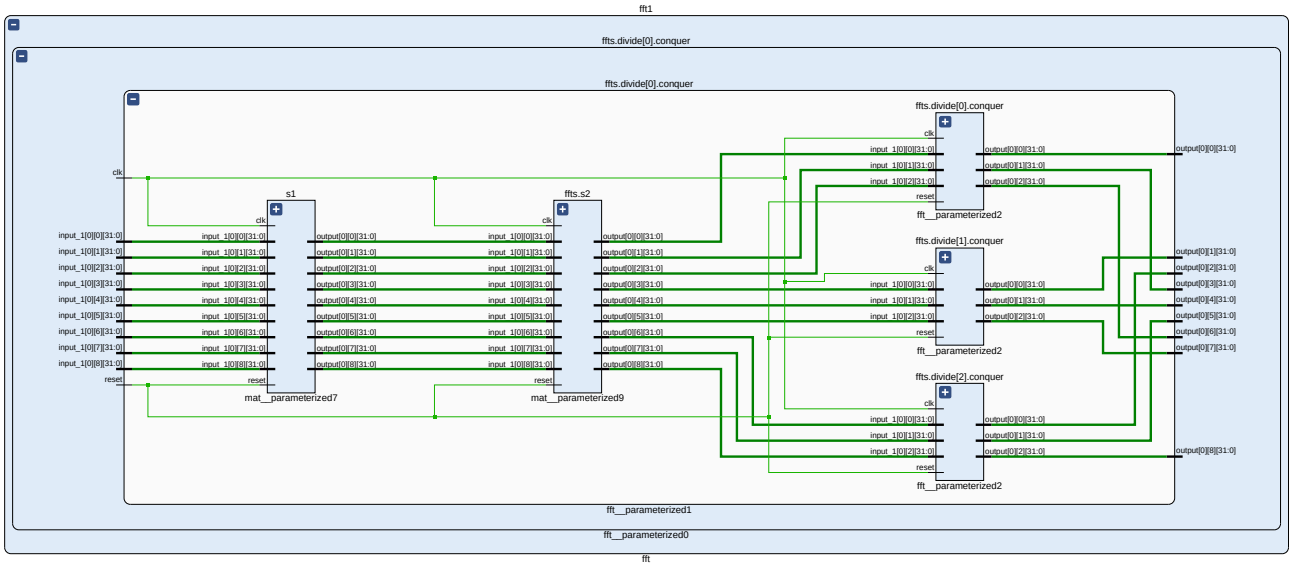


Fig. 5.21: *T_9 divided into three T_3 FFTs.*

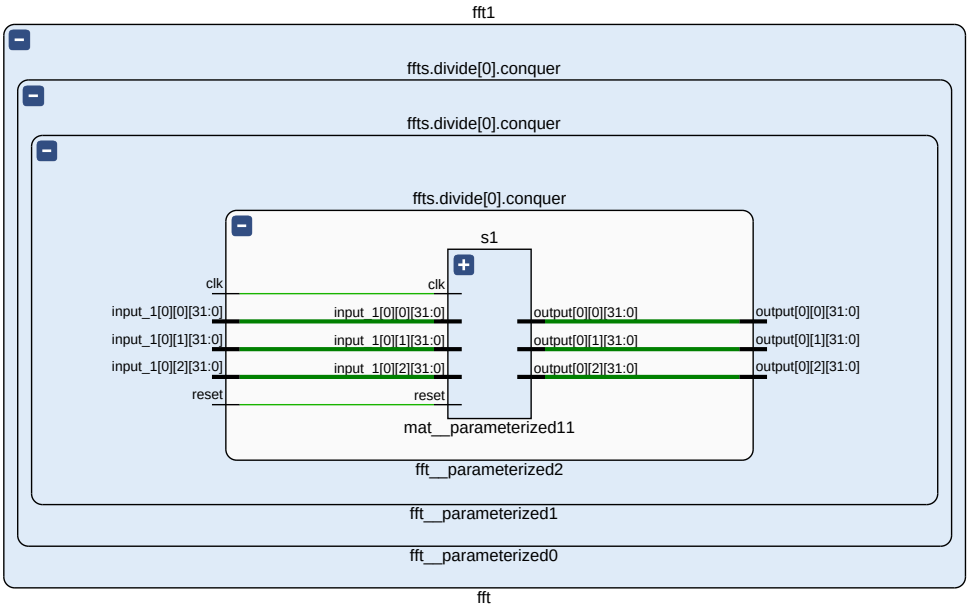


Fig. 5.22: *T_3 RTL.*

The proposed systems operate in parallel, an arbitrary N point FFT takes N input samples within the same clock cycle and generates N outputs simultaneously, after the pipeline number of cycles of latency. This aggressive approach represents the first step towards achieving the maximum processing throughput, higher clock rates are achieved adding pipeline registers to break up the critical combinational path.

To appropriately define the architecture of an arbitrarily sized Fourier transform with whatever radix is desired a novel approach is taken, to the best knowledge of the authors. Our system is instantiated recursively in VHDL (**Lst.** 5.1) using a complex matrix multiplier (**Lst.** 5.2). The constants for each of the matrices, containing the Fourier coefficients are generated using a Python script. This is the key development of our approach that enables unprecedented flexibility.

```
1  input <= input_1;
2  s1: mat GENERIC MAP(input_2 => INIT_DIAG(1))
3          PORT    MAP(clk, reset, input_1 => input, output => output_diag1);
4  ffts: IF (INDEX /= 0) GENERATE
5          s2: mat GENERIC MAP(input_2 => INIT_DIAG(0))
6          PORT MAP(clk, reset, input_1 => output_diag1, output => output_diag);
7          c0: FOR I IN M       DOWNTO 0 GENERATE
8          c1: FOR J IN N/RADIX DOWNTO 0 GENERATE
9          c2: FOR K IN RADIX-1 DOWNTO 0 GENERATE
10                 diag_to_R_arr(K)(I)(J)   <= output_diag (I)(J+(K*(N+1)/RADIX));
11                 output_reg(I)(J*RADIX+K) <= output_R_arr(K)(I)(J);
12         END GENERATE;
13         END GENERATE;
14         END GENERATE;
15                 divide: FOR I IN 0 TO RADIX-1 GENERATE
16                         conquer: ENTITY fft
17                         GENERIC MAP(
18                                 INDEX   => INDEX-1,
19                                 N       => N/RADIX)
20                         PORT MAP(
21                                 input_1 => diag_to_R_arr(I),
22                                 output  => output_R_arr (I),
23                                 clk     => clk,
24                                 reset   => reset);
25                 END GENERATE;
26 ELSE GENERATE
27         output_reg <= output_diag1;
28 END GENERATE;
29 output <= output_reg;
```

Listing 5.1: *Fast Fourier Transform – Main Logic*

Multiplications are truncated in half as to keep the word length of our datapath constant, that is, an $n \times n$ multiplication yields an n bit wide result. Both signals and coefficients are quantised to 18 bits for each of the real and complex parts, yielding 36 bit wide samples. The real and complex parts are formatted to signed fixed point representation with 12 bits of precision for the fraction portion, truncation is performed accordingly. 18 bits per part is chosen as this is the size of the smallest port of the DSP48E2, the architecture has been parametrized as to accept any arbitrary width for any of the signals, including a different width for the coefficients. Similarly, any arbitrary fixed point representations can be set as a parameter.

```
input_1reg <= input_1;
ROW:FOR I IN 0 TO M GENERATE
COL:FOR J IN 0 TO N GENERATE
        DOT: dot_complex
        GENERIC MAP(
                WIDTH   => WIDTH,
                N       => N,
                input_2 => input_2   (J)(N DOWNTO 0)
                )
        PORT MAP(
                input_1 => input_1reg(I)(N DOWNTO 0),
                clk     => clk,
                output  => output_in (I)(J),
                reset   => reset
                );
END GENERATE;
END GENERATE;
output     <= output_in;
```

Listing 5.2: *Constant Complex Matrix Multiplier – Main Logic*

The complex multiplier strategy uses 4 real multiplications and 2 additions. For two complex numbers, $a = x+y\cdot j$ and $b = u+v\cdot j$ conventional multiplication is defined as $a\cdot b = (x+y\cdot j)\cdot(u+v\cdot j) = (xu-yv)+(xv+yu)\cdot j$.

To achieve the maximum clock rate in order to maximise throughput we direct the implementation tool to implement trivial multiplications as distributed logic using LUTs and non-trivial real multiplications on DSP slices. This compromise solution shares the utilization load between the logic fabric and available DSP slices, though it might seem inefficient to use DSP multipliers for signal-constant multiplication, the total DSPs occupied represent a small percentage of the total available. Additionally, the proposed architecture can easily be adapted for LUT only implementation if an area-efficient optimisation is needed, though this might come with optimisation penalty or an increase of pipeline registers.

All transforms have been tested on Virtex Ultrascale+ speed grade -3 FPGA (xpcvu35p-fsvh2104-3-e) using the synthesis strategy Flow Alternate Routability and the implementation strategy Performance Explore Post Route Phys. Opt. The target frequency was set at 891 MHz with a clock period of 1.123 ns, this is the maximum clock rate supported by the FPGA. The speed grade -3 variant has the fastest clock of any FPGAs presently offered by Xilinx. This FPGA features 872k Look Up Tables, 1743k D type Flip Flop registers and 5952 DSP48E2 Arithmetic Logic Units. DSP48E2 slices feature an 18×27 bit signed multiplier and a pre- and post-adder that performs addition and subtraction, as well as other logic operations, when this slice is fully pipelined it achieves the aforementioned 891 MHz clock rate.

The constant complex dot product generator (**Lst.** 5.3) features a generic tree adder following the complex multiplications, this reduces the critical path in comparison to chaining adders linearly. During implementation skipping multiplications by zero with an `IF` statement results in more stream lined hardware. Throughout the hardware descriptions stages of registers have been added to breakup the critical signal paths. This can be seen at the outputs of the scalar product and withing the constant complex multipliers.

```
input_1reg <= input_1;
-- Mults
MULTS: FOR I IN 0 TO N GENERATE
skip: IF (input_2(I)(WIDTH-1 DOWNTO WIDTH/2)&input_2(I)((WIDTH/2)-1 DOWNTO 0)
   /= 0) GENERATE
        uut: complex
        GENERIC MAP(
                WIDTH  => WIDTH/2,
                real_B => input_2   (I)(WIDTH-1     DOWNTO WIDTH/2),
                imag_B => input_2   (I)((WIDTH/2)-1 DOWNTO 0)
        )
        PORT MAP(
                real_A => input_1reg(I)(WIDTH-1     DOWNTO WIDTH/2),
                imag_A => input_1reg(I)((WIDTH/2)-1 DOWNTO 0),
                real_C => real_C(I),
                imag_C => imag_C(I),
                clk    => clk,
                reset  => reset);
END GENERATE;
END GENERATE;
-- Tree Adder
ADDS:  FOR I IN 0 TO N/2   GENERATE
FIRST: IF  I =  0          GENERATE
TREE:  FOR J IN 0 TO N + 1 GENERATE
EVEN:  IF  J MOD 2 = 1     GENERATE
        sum_buff_out (I)(((J-1)+2)/2-1) <= real_C(J-1) + real_C(J);
        sum_buff_out1(I)(((J-1)+2)/2-1) <= imag_C(J-1) + imag_C(J);
END GENERATE;
END GENERATE;
END GENERATE;
REST: IF  I >  0           GENERATE
TREE: FOR J IN 0 TO N/I + 1 GENERATE
EVEN: IF  J MOD 2 = 1       GENERATE
        sum_buff_out (I)(((J-1)+2)/2-1) <= sum_buff_out (I-1)(J-1) +
           sum_buff_out (I-1)(J);
        sum_buff_out1(I)(((J-1)+2)/2-1) <= sum_buff_out1(I-1)(J-1) +
           sum_buff_out1(I-1)(J);
END GENERATE;
END GENERATE;
END GENERATE;
END GENERATE;
output_buff <= sum_buff_out(N/2)(0) & sum_buff_out1(N/2)(0);
out_pipe    <= out_pipe(PIPE1 DOWNTO 0) & output_buff WHEN rising_edge(clk);
output      <= out_pipe(PIPE1);
```

Listing 5.3: *Constant Complex Dot Product – Main Logic*

The most efficient complex multiplier observed with the particular setup proposed is the four multiplier approach as shown in **Lst.** 5.4. The result is truncated at the output stage, the fixed point is maintained at the same position as to not alter the magnitude of the output number.

```
-- Input
real_A_reg <= real_A;
imag_A_reg <= imag_A;

-- 4 Multiplier
mult1 <= real_A_reg * real_B;
mult2 <= imag_A_reg * imag_B;
mult3 <= real_A_reg * imag_B;
mult4 <= real_B * imag_A_reg;

real_C_reg <= mult1 - mult2 WHEN rising_edge(clk);
imag_C_reg <= mult3 + mult4 WHEN rising_edge(clk);

-- Output Truncation
real_C <= real_C_reg(2*WIDTH-(BIT_I+1)DOWNTO WIDTH-BIT_I)WHEN rising_edge(clk);
imag_C <= imag_C_reg(2*WIDTH-(BIT_I+1)DOWNTO WIDTH-BIT_I)WHEN rising_edge(clk);
```

Listing 5.4: *Constant Complex Multiplier – Main Logic*

LUT only multiplication is optimised with a generic Canonical Signed Digit multipliers. The non-adjacent form of the constant is precomputed with the Prodinger [99] method. This approach seeks to reduce the complexity of a multiplication by using both positive and negative partial products.

```
CONSTANT xh : STD_LOGIC_VECTOR(WIDTH - 1 downto 0) :=  input_constant SRL 1;
CONSTANT x3 : STD_LOGIC_VECTOR(WIDTH - 1 downto 0) :=  input_constant + xh;
CONSTANT c  : STD_LOGIC_VECTOR(WIDTH - 1 downto 0) :=  xh XOR x3;
CONSTANT np : STD_LOGIC_VECTOR(WIDTH - 1 downto 0) :=  x3 AND c;
CONSTANT nm : STD_LOGIC_VECTOR(WIDTH - 1 downto 0) :=  xh AND c;

BEGIN -- Datapath
        input_reg <= input WHEN rising_edge(clk);

        mult_buff1 <= mult_buff1(PIPE1 DOWNTO 0) & (input_reg*np) WHEN
           rising_edge(clk);
        m1         <= mult_buff1(PIPE1);
        mult_buff2 <= mult_buff2(PIPE1 DOWNTO 0) & (input_reg*nm) WHEN
           rising_edge(clk);
        m2         <= mult_buff2(PIPE1);

        out_buff <= m1 - m2 WHEN rising_edge(clk);
        output  <= out_buff WHEN rising_edge(clk);
END Behavioral;
```

Listing 5.5: *Canonical Signed Digit Product – Main Logic*

DSP systems have 3 cycles of latency per complex matrix multiplication, that is 6 clock cycle latency per stage of the decomposition. LUT only systems have 5 cycles/matrix multiply for a total of 10 clock cycles of added latency per stage.

The values of the constant matrices are calculated with a simple Python script using `numpy` [100] functions. The values are written in a file and read from the VHDL source code.

```
def diag(n, radix):
    Kp = [np.eye(n//radix)]
    d  = np.array([np.e**((-2j*np.pi*k)/n) for k in range(n)])
    [Kp.append((np.eye(n//radix)*d[0:n//radix])**i) for i in range(1,radix)]
    S1 = np.kron(dftmtx(radix),np.eye(n//radix))
    S2 = sparse.block_diag(Kp).A
    return S1, S2
```

Listing 5.6: *Python constant matrix generator*

The developed systems have been tailored for throughput, though in future work area optimisation could be investigated. We match our surpass the throughput of state of the art parallel FFT implementations with a reasonable increase in resource usage. On top of this, the proposed architecture also covers FFT sizes and radices not commonly found in scientific literature. They are simulated with a test signal and the results are compared to the expected values calculates using the numpy library FFT function. The noise is calculated as the difference and used to calculate the Signal-to-Noise ratio. An example is given in **Fig.** 5.23.

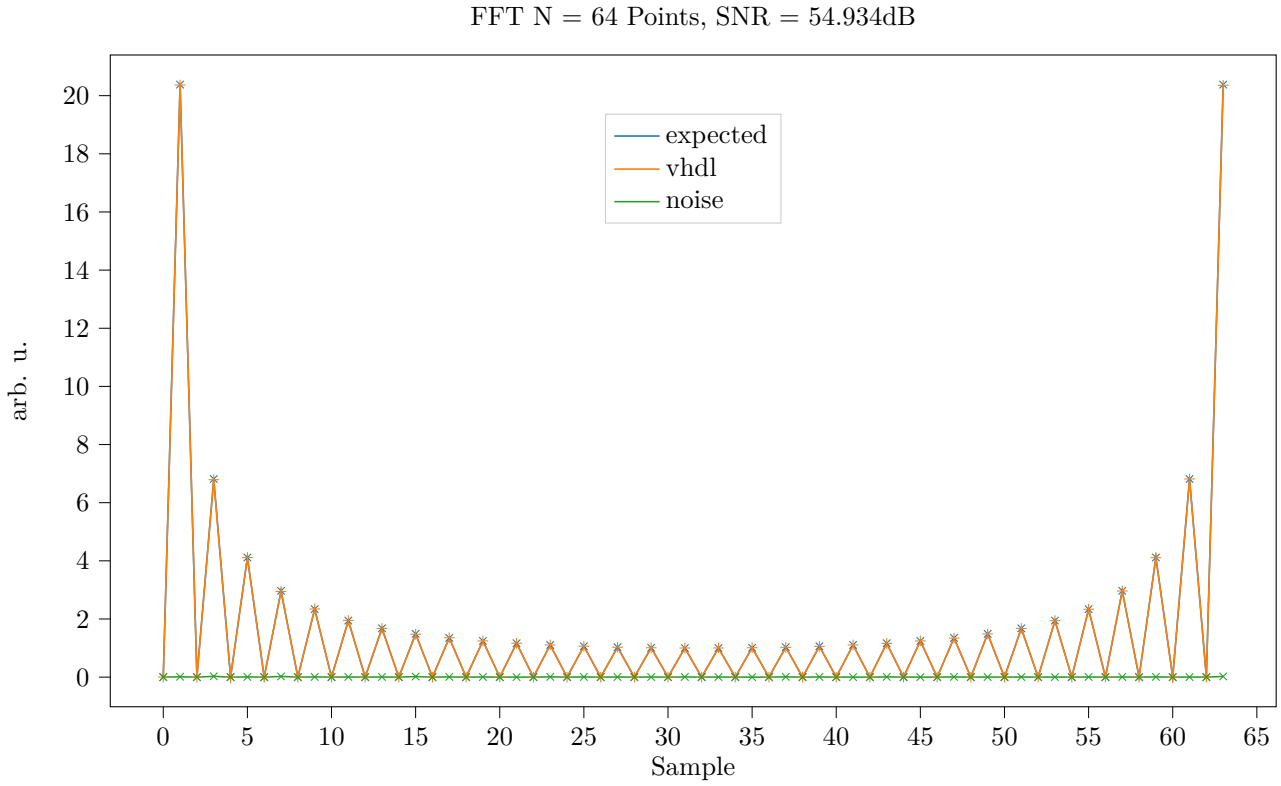


Fig. 5.23: *64 point FFT simulation*

Fourier synthesis is demonstrated in **Fig.** 5.24 by generating the symbols of 36 point Quadrature Amplitude Modulation. Each symbol is generated with 36 samples using the inverse 36 point variant of the FFT systems developed, each of the samples has a precision of 36 bits for the real part of the signal and 36 bits for the complex part.

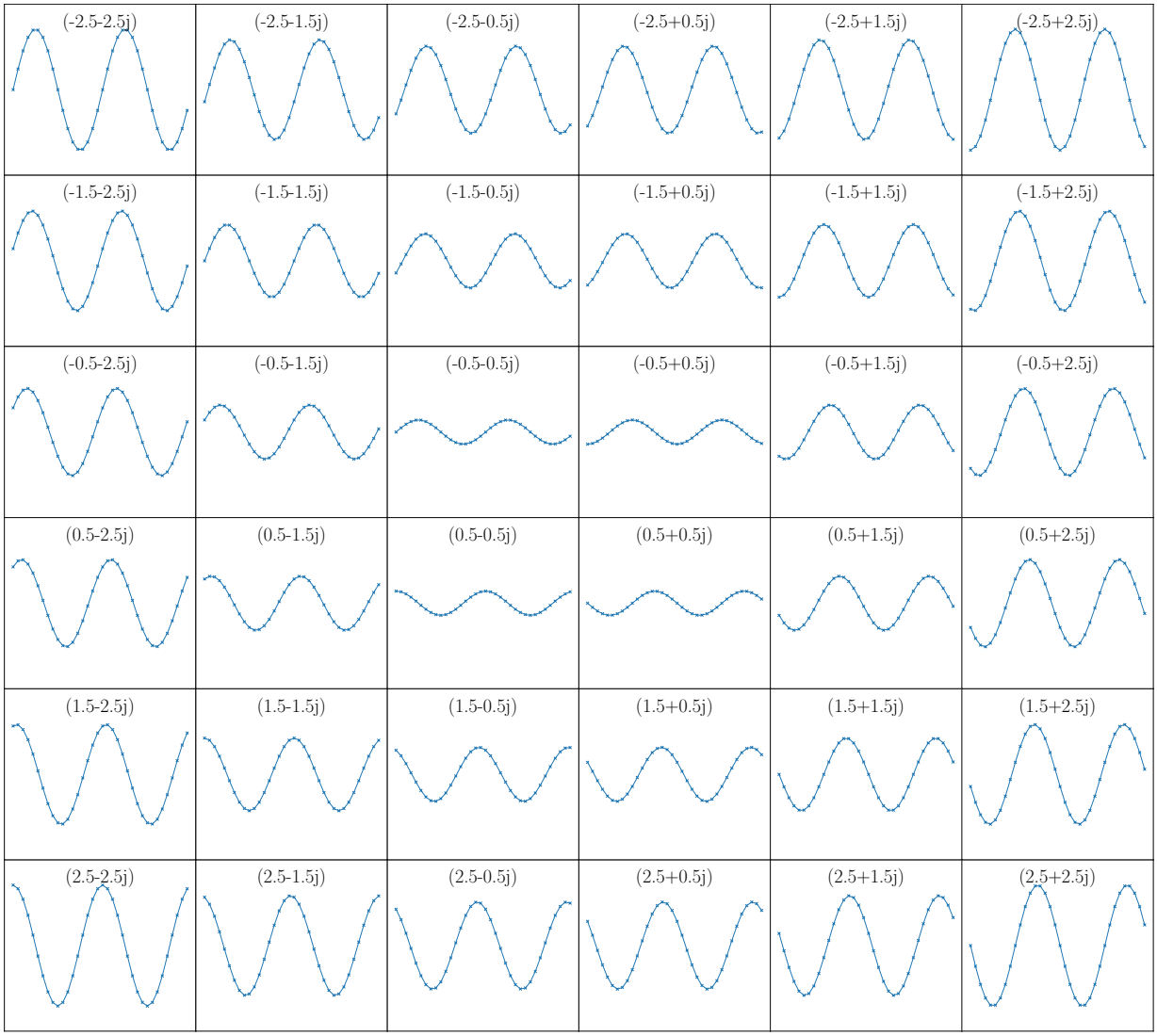


Fig. 5.24: *36-QAM with 36 points/symbol with 36 bits/point.*

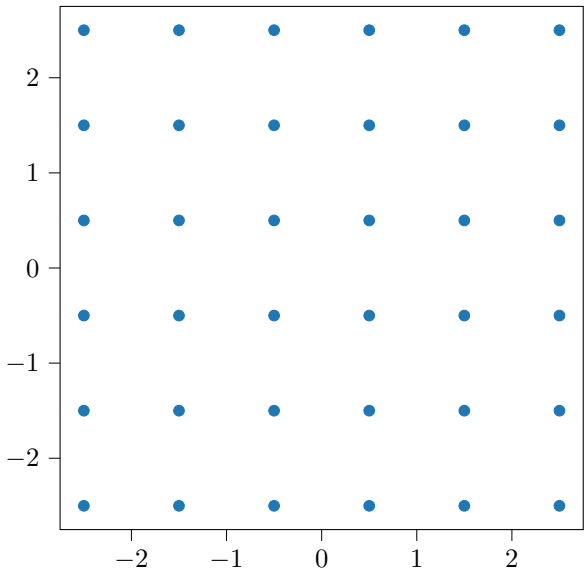


Fig. 5.25: *36-QAM constellation diagram.*

IV Results

i DSP-based implementations

Table 5.1: *Fourier transform DSP implementations on Virtex Ultrascale+.*

Points/Stage	F_{max} (MHz)	LUT	FF	DSP	SNR (dB)	GS/s
12|4	891	3.9k	12.1k	40	62	10.69
15|3	724	8.0k	19.4k	264	63	10.86
16|4	891	5.3k	14.8k	30	75	14.25
18|6|2	718	8.1k	24.7k	144	66	12.92
24|12|6|3	891	10.8k	38.3k	127	70	21.38
27|9|3	744	14.9k	42.1k	328	66	20.08
30|6	648	25.1k	49.5k	694	66	19.44
32|8|2	891	13.6k	42.2k	106	71	28.51
36|12|4	715	19.4k	55.5k	288	67	25.74
48|24|12|6|3	829	29.8k	88.5k	333	69	39.70
48|12|3	843	25.9k	68.5k	266	71	40.46
54|18|6|2	678	35.9k	107.5k	708	64	36.61
64|32|16|8|4|2	891	34.4k	119.6k	377	68	57.16
64|32|16|8|4	817	33.9k	109.0k	377	68	53.16
64|16|4	862	35.1k	97.0k	294	69	55.16
81|27|9|3	651	60.8k	161.6k	1408	56	52.73
96|48|24|6|3	824	63.2k	208.0k	841	59	79.10
128|32|8|2	788	82.8k	245.4k	790	62	100.23
128|64|32|16|8|4|2	796	81.5k	283.4k	1001	60	101.88

Table 5.1 summarises the characteristics of the developed systems. The Signal-to-Noise ratio is approximated by generating an input signal, simulating the circuits response and calculating the noise as the difference of the obtained output and the expected result, this has only been done non-exhaustively for one signal, so it can only be taken as a coarse estimate, future work is needed to more precisely measure the systems signal characteristics.

The trends measured are coherent with what is expected. Power-of-two Fourier transforms are implemented using fewer DSP multipliers, this is due to the fact they require fewer non-trivial multiplications. Radix-3 decompositions invariable utilise more DSP blocks and resources in general while achieving poorer throughput. Overall, regardless of radix we see an increase in resources as the number of point increase. Despite the complexity increase our architecture and the pipelining realises manages to main high clock rates. Our 128 point parallel Fourier transform achieves a 100 GS/s throughput both in radix-2 and radix-4 variants, as anticipated the radix-4 variant uses 21% DSP slices while demonstrating comparable performance. This is variant also largest occupies 10% of total LUTs, 14% FF and 13% of all DSPs.

ii LUT-based implementations

Table 5.2: *Fourier transform LUT implementations on Virtex Ultrascale+.*

Points/Stage	F_{max} (MHz)	LUT	FF	DSP	SNR (dB)	GS/s
16|8|4|2	891	7.9k	28.2k	0	75	14.25
16|4	891	8.3k	22.8k	0	75	14.25
24|12|6|3	891	20.5k	59.4k	0	72	21.38
27|9|3	738	38.5k	92.7k	0	60	19.92
32|16|4|2	891	22.6k	67.2k	0	65	28.51
32|8|2	891	22.9k	62.0k	0	65	28.51
48|12|3	708	46.8k	126.7k	0	70	33.98
64|32|16|8|4|2	794	59.2k	166.3k	0	54	50.08
128|64|32|16|8|4|2	688	133.7k	431.4k	0	50	88.06

These designs are implemented without using the dedicated DSP multipliers, this strategy may be more appropriate for ASIC designs as it doesn't rely on hardened IP for its realisation. These designs prove that the proposed architecture definition is efficient in terms of area without compromising on speed. The digital architecture developed allows for fine grain control of pipeline registers, leading to short critical paths while maintaining the synchronization of data.

The designs are implemented with (16+16) (real+complex) bit wide signals, though the proposed IP core is parametrised and can be configured with arbitrarily large or small signals as well as be configured with custom fixed point number codifications. The reference systems demonstrated have 5 bits for the integer part and 11 for the fractional, this codification has been measured to avoid overflow in the integer part, however it can me tailored to depending on the magnitude of the signals to be processed.

The trends in resource utilization follow the patters we expect, as the number of points processed grows, the number of LUTs and FFs employed increases. Similarly power-of-two FFTs require the fewest resources while power-of-three decompositions are less efficient. Smaller systems can operated at theoretical maximum of 891 MHz while there is a slight drop-off in performance with larger systems. Larger systems would require further optimisation to achieve higher clock rates.

Comparing **Table** 5.1 and **Table** 5.3 we see that the DSP implementations manage higher clock rates and slightly higher SNRs due to the fact they are (18+18) bit wide vs. (16+16) bit wide. The cost to pay is the usage of some DSP multipliers for multiplications by constants, which may seem wasteful for this resource. The LUT only approach saves all the DSP slices though it obviously occupies more LUTs and also more registers. For example, 64 point FFT LUT only implementation uses 42.3% more LUTs, 50.1% more registers buy 377 (100%) fewer DSP multipliers. From the 16 and 32 point circuits we see how different decompositions have little impact on the final result.

V Discussion

Reference parallel radix-2 and radix-4 are given by the aptly named '*World's fastest FFT architectures [...]*' by Garrido et al. [101]. In all cases we achieve faster architectures.

Table 5.3: *Power-of-two FFT comparison.*

Points	Ref.	F_{max} (MHz)	LUT	FF	DSP	SNR (dB)	GS/s
8	Garrido et al. [101]	655	1.3k	1.6k	0	79	6.77
8	Proposed	891	2.6k	7.9k	0	82	7.13
8	Proposed	891	1.2k	6.9k	13	82	7.13
16	Garrido et al. [101]	655	4.6k	5.5k	0	75	10.80
16	Proposed	891	7.9k	28.2k	0	75	14.25
16	Proposed	891	5.3k	14.8k	30	75	14.25
32	Garrido et al. [101]	655	13.3k	18.5k	0	72	20.96
32	Proposed	891	22.6k	67.2k	0	65	28.51
32	Proposed	891	13.6k	42.2k	106	71	28.51
64	Garrido et al. [101]	590	36.0k	50.6k	0	68	37.87
64	Proposed	794	59.2k	166.3k	0	54	50.08
64	Proposed	891	34.4k	119.6k	377	68	57.16
128	Garrido et al. [101]	531	90.3k	127.6k	0	65	57.97
128	Proposed	688	133.7k	431.4k	0	50	88.06
128	Proposed	788	82.8k	245.4k	790	62	100.23

At 32 points we increase throughput by 40% in both of our proposed architectures. The DSP version does so occupying the same number of LUTs and 106 DSP slices, the LUT only architecture occupies 69% more LUTs as compared to the reference. At 64 points the DSP variant achieves a throughput 54% higher, LUT usage remains the same and 377 DSP multipliers are used. At 128 points by occupying an additional 790 DSP blocks we mange a speedup of 75% in terms of throughput, netting us 100 GS/s.

VI Chapter Conclusions

In this paper we demonstrate a number of parallel Fourier transform implementations capable of operating at clock rates of 891 MHz on Virtex Ultrascale+ with signals of (18+18) bits (real+complex) for systems implemented with DSP multipliers and (16+16) bits when using LUT only multiplication. We manage to mesure resource utilization efficiency in accordance to the limits of the factorisation applied, as expected. Additionally we review the areas of application of Fourier analysis and synthesis, we summarise the mathematical theory behind the Fourier Integral Theorem, deduce the discrete Fourier transform, discuss the matrix form of the FT and the FFT decompositions. We also demonstrate the RTL of FFTs with various points and factorisations.

CHAPTER

6

CONCLUSIONS AND FUTURE WORK

I Conclusions

This Thesis has presented an automated methodology that performs exhaustive optimization of the clock rate of the digital logic architecture of a variety of systems in the context of Digital Signal Processing. We have successfully discretised the transfer function of PID and PI controllers and proposed an efficient hardware architecture to implement them in FPGA. Additionally we thoroughly investigate the behaviour of the system with a large range of precisions, from low resolution 8 bits signals up to high precision 96 bit signals. The proposed architecture is optimised to achieve the maximum possible clock rate for the implementation in all available Xilinx FPGAs, Artix, Kintex and Virtex and their Ultrascale and Ultrascale+ variants. With this we have demonstrated our ability to develop and optimise high resolution systems without compromising on performance. The approach developed exhaustively maps the design space, this allows to make decisions objectively so the system can be implemented with confidence according to a wide range of specification criteria. The developed systems surpass other implementations described in scientific literature in terms of speed and precision, also in terms of flexibility, customizability and architectural robustness.

After that, we have also proposed a novel approach for integer packing DSP48 multipliers in order to minimise the area of 6+6 bit (real+complex) unsigned complex multiplication. The developed architecture scales well for 8 and 9-bits. We also implement and study two other complex multiplication strategies, the common 4 multiplier method and a compact 3 multiplier method which prove to be more efficient at large word lengths. To demonstrate the application of the proposed complex multiplier developed we design a complex parallel matrix multiplier, a highly resource intensive operator. We show that a 9-bit 9×9 matrix multiplication can only be implemented in FPGA with our DSP packing approach, and implementation fails with other complex multiplication methods.

Lastly after developing digital processing systems highly optimised in terms of resolution, speed and area we tackle any-radix arbitrary number of points parallel fast Fourier transforms. Our novel design approach and architectural definition tackles the complexity of highly parallelized processing systems. It handles the methodical study of Fourier transform FPGA implementations in ways not usually seen, in particular in terms of non power-of-two FFTs and unusual radices FFTs. The versatility and robustness of the digital logic architecture design approach followed is further exemplified by the fact that both LUT only and LUT+DSP implementations have been demonstrated. In all cases the developed FFTs surpass other implementations in terms of clock rate and therefore data throughput with modest increases in resource consumption, in addition to demonstrating arbitrary number of points and arbitrary radix variants.

All of the compound operators developed in this Thesis have a great number of application domains in domain specific architectures. Particularly the scalar and matrix products, whether the real or complex or constant or general implemented as LUTs or using DSPs in FPGAs or ASICs. They are a vital for image processing, machine learning, artificial intelligence, deep neural network training in data centers or inference on the edge, digital signal processing for telecommunication, aerospace and defense, automotive, big data, high performance computing, and many others where CPU and GPU processing is proving to be insufficient in terms of flexibility, efficiency and throughput.

II Future work

Following this work, further research is needed to correctly characterise the systems in terms of round-off and quantisation noise, as well as investigating higher precision (larger word length) Fourier transforms and higher order (larger number of points) and different radices. Another natural path that follows from the RTL developed is to design and implement an ASIC microchips for digital signal processing FFTs and inverse FFTs.

The multiplication of matrices is a major operation in many scientific and engineering applications. As FPGA technology continues to develop, additional studies are needed to develop more efficient matrix multiplication architectures. These studies may include a study of new algorithms, optimizing existing algorithms and the development of new architectures that can benefit from the unique features of FPGAs. In addition, it is necessary to develop techniques for effective mapping of matrix multiplication to FPGA, as well as techniques for effective data movement between the FPGA and the host processor. The knowledge, know-how and insight drawn from the realisation of this Thesis could be used developing other complex architectures for computationally intensive digital processing systems. The expertise demonstrated with the multiple matrix product systems proposed are directly applicable accelerating neural networks, encoding and decoding video streams or modulating and demodulating telecommunications.

Lastly, the high performance computer architectures developed here, and the methodology used to obtain them, are important because they allow for the development of faster, smaller, and more powerful semiconductor microchips. This is needed for a variety of reasons, including the ability to create more powerful computers, faster communication networks, and more capable consumer electronics. In essence, more efficient microchips contribute to the reduction of energy consumption and improve the performance of signal processing systems.

APPENDIX

A

BUDGET

i Labour

N	Description	Units	Quantity	Unit Price	Total
1	Hardware Researcher	h	350	50 €	17500 €
2	Tutor	h	49	100 €	4900 €
				Total	22400 €

Table A.1: *Labour Costs.*

ii Hardware

N	Description	Quantity	Price	Amortization	Total
1	PC	1	3000 €	10 years	300€
2	Server	1	5000 €	10 years	500€
3	Virtex Ultrascale+	1	10000 €	10 years	1000€
				Total	1800 €

Table A.2: *Hardware Costs.*

iii Software

N	Description	Quantity	Price	Amortization	Total
1	Vivado	1	3000 €	10 years	300€
				Total	300 €

Table A.3: *Software Costs.*

iv Total

N	Description	Total
1	Labour	22400 €
2	Hardware	1800 €
3	Software	300 €
	Total sin IVA	24500 €

Table A.4: *Total Costs.*

APPENDIX

B

ECONOMICAL, ETHICAL, SOCIAL, ENVIRONMENTAL ASPECTS

The developed systems and design methodology have positive effects in economical, ethical, and social issues. The technology developed has enabled businesses to become more innovative and creative, leading to new products and services that benefit society. It has also enabled businesses to become more efficient and reduce costs, leading to lower prices for consumers. In general the developed technological systems have the following effects:

- Economic: it enables businesses to become more efficient and productive, leading to increased profits and economic growth. It has also enabled businesses to reach new markets and customers, creating new jobs and opportunities.
- Ethical: enables businesses to become more transparent and accountable, leading to greater trust and confidence in the marketplace. It has also enabled businesses to better protect their customers' data and privacy, leading to greater consumer protection.
- Social: enables people to connect with each other in ways that were not possible before, leading to greater understanding and collaboration. It has also enabled people to access information and resources that were previously unavailable, leading to greater access to education and knowledge.
- Environmental: technology has enabled businesses to become more efficient and reduce their environmental impact. It has also enabled businesses to develop renewable energy sources, leading to a reduction in greenhouse gas emissions.

BIBLIOGRAPHY

[1] Intel 4004, "4004 Microprocessor Datasheet pdf - P-Channel Microprocessor. Equivalent, Catalog." [Online]. Available: https://datasheetspdf.com/pdf/787753/Intel/4004/1

[2] Siliconpr0n, "Index of /map/intel/4004/." [Online]. Available: https://siliconpr0n.org/map/intel/4004/

[3] XC2064, "XC2064 Array Datasheet - Cell Array. Equivalent, Catalog," 1985. [Online]. Available: https://datasheetspdf.com/pdf/625540/Xilinx/XC2064/1

[4] Siliconpr0n, "Index of /map/xilinx/xc2064." [Online]. Available: https://siliconpr0n.org/map/xilinx/xc2064/

[5] Intel, "1.6. Intel® Hyperflex™ Core Architecture," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683729/current/core-architecture.html

[6] Xilinx, "Ultrascale architecture dsp slice user guide," 2021.

[7] D. G. Bailey, "Image processing using fpgas," p. 53, 2019.

[8] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based implementation of signal processing systems*. John Wiley & Sons, 2008.

[9] J. U. Cho, Q. N. Le, and J. W. Jeon, "An fpga-based multiple-axis motion control chip," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 3, pp. 856–870, 2008.

[10] N. Lashkarian, E. Hemphill, H. Tarn, H. Parekh, and C. Dick, "Reconfigurable digital front-end hardware for wireless base-station transmitters: Analysis, design and fpga implementation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 8, pp. 1666–1677, 2007.

[11] J. Meng, N. Gebara, H.-C. Ng, P. Costa, and W. Luk, "Investigating the feasibility of fpga-based network switches," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 218–226.

[12] P. Papaphilippou, J. Meng, N. Gebara, and W. Luk, "Hipernetch: High-performance fpga network switch," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 1, pp. 1–31, 2021.

[13] J. Meng, N. Gebara, H.-C. Ng, P. Costa, and W. Luk, "Investigating the feasibility of FPGA-based network switches," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, jul 2019. [Online]. Available: https://doi.org/10.1109%2Fasap.2019.00010

[14] L. Hey, P. Cheung, and M. Gellman, "FPGA based router for cognitive packet networks," in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.* IEEE. [Online]. Available: https://doi.org/10.1109%2Ffpt.2005.1568586

[15] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. IEEE, jun 2007. [Online]. Available: https://doi.org/10.1109%2Fmse.2007.69

[16] C. Insaurralde, "Reconfigurable computer architectures for dynamically adaptable avionics systems," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 30, no. 9, pp. 46–53, sep 2015. [Online]. Available: https://doi.org/10.1109%2Fmaes.2015.140077

[17] K. Krishnakumar, J. Kaneshige, R. Waterman, C. Pires, and C. Ippolito, "A plug and play GNC architecture using FPGA components," in *Infotech@Aerospace*. American Institute of Aeronautics and Astronautics, jun 2005. [Online]. Available: https://doi.org/10.2514%2F6.2005-7120

[18] P. Kemao, D. Miaobo, M. C. Ben, C. Guowei, L. K. Yew, and H. L. Tong, "Design and implementation of a fully autonomous flight control system for a UAV helicopter," in *2007 Chinese Control Conference*. IEEE, jul 2006. [Online]. Available: https://doi.org/10.1109%2Fchicc.2006.4347398

[19] B. Fuller, J. Kok, N. Kelson, and F. Gonzalez, "Hardware design and implementation of a mavlink interface for an fpga-based autonomous uav flight control system," in *Proceedings of the 16th Australasian Conference on Robotics and Automation 2014*. Australian Robotics and Automation Association (ARAA), 2014, pp. 1–6.

[20] B. P. Anand and C. Saravanan, "Development of research engine control unit using fpga-based embedded control system," *Journal of KONES*, vol. 19, pp. 9–18, 2012.

[21] B. Du and L. Sterpone, "An fpga-based testing platform for the validation of automotive powertrain ecu," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2016, pp. 1–7.

[22] S. Chappell, A. Macarthur, D. Preston, D. Olmstead, B. Flint, and C. Sullivan, "Exploiting real-time fpga based adaptive systems technology for real-time sensor fusion in next generation automotive safety systems," in *The IEE Seminar on Target Tracking: Algorithms and Applications 2006 (Ref. No. 2006/11359)*. IET, 2006, pp. 61–68.

[23] G. J. García, C. A. Jara, J. Pomares, A. Alabdo, L. M. Poggi, and F. Torres, "A survey on fpga-based sensor systems: towards intelligent and reconfigurable low-power sensors for computer vision, control and signal processing," *Sensors*, vol. 14, no. 4, pp. 6247–6278, 2014.

[24] X. Shao and D. Sun, "A fpga-based motion control ic design," in *2005 IEEE International Conference on Industrial Technology.* IEEE, 2005, pp. 131–136.

[25] N. Anish, B. Kowshick, and S. Moorthi, "Ethernet based industry automation using fpga," in *2013 Africon.* IEEE, 2013, pp. 1–4.

[26] Ross H. Freeman, "Configurable electrical circuit having configurable logic elements and configurable interconnects," Sep. 26 1989, uS Patent 4,870,302.

[27] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays . Trees . Hypercubes.* Amsterdam: Morgan Kaufmann, 2014, oCLC: 1100857979.

[28] S. Arora, F. T. Leighton, and B. M. Maggs, "On-line algorithms for path selection in a nonblocking network," *SIAM Journal on Computing*, vol. 25, no. 3, pp. 600–625, 1996. [Online]. Available: https://doi.org/10.1137/S0097539791221499

[29] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," *Tree-based heterogeneous FPGA architectures*, pp. 7–48, 2012.

[30] J. Lamoureux and S. J. E. Wilton, "Fpga clock network architecture: Flexibility vs. area and power," in *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 101–108. [Online]. Available: https://doi.org/10.1145/1117201.1117216

[31] Z. Que, H. Nakahara, H. Fan, H. Li, J. Meng, K. H. Tsoi, X. Niu, E. Nurvitadhi, and W. Luk, "Remarn: A reconfigurable multi-threaded multi-core accelerator for recurrent neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, may 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3534969

[32] M. A. A. Ibrahim, "Extending data flow architectures for convolutional neural networks to object detection and multiple fpgas," Ph.D. dissertation, 2022.

[33] J. Shipton, J. Fowler, C. Chalmers, S. Davis, S. Gooch, and G. Coccia, "Implementing wavenet using intel® stratix® 10 nx fpga for real-time speech synthesis," 2020.

[34] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016. [Online]. Available: https://arxiv.org/abs/1609.03499

[35] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *iEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[36] M. Plakalovic, E. Kaljic, and M. Mehic, "High-speed fpga-based ethernet traffic generator," in *2022 XXVIII International Conference on Information, Communication and Automation Technologies (ICAT).* IEEE, 2022, pp. 1–6.

[37] Intel, "Intel® Stratix® 10 TX E-Tile Achieves 400G Connectivity with QSFP-DD..." 2022. [Online]. Available: https://www.intel.com/content/www/in/en/wireline/products/programmable/applications/stratix-10-tx-e-tile-achieves-400g-video.html

[38] ——, "FPGA Transceiver - 58G Pam4 Transceiver - Intel® FPGA," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/programmable/transceiver/overview.html

[39] W. Kamp, N. Abel, and G. Comoretto, "Complex multiply accumulate cells for the square kilometre array correlators," in *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2018, pp. 1–6.

[40] Intel, "1.1. Features," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683832/21-2/features-87200.html

[41] T. Tan, E. Nurvitadhi, D. Shih, and D. Chiou, "Evaluating the highly-pipelined intel stratix 10 fpga architecture using open-source benchmarks," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 206–213.

[42] Xilinx, "Ds890 ultrascale architecture and product data sheet: Overview," 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp450-base-stn-connect

[43] C. Wanotayaroj, E. Mendes, and S. Baron, "Pam-4 implementation study for future high-speed links," *Journal of Instrumentation*, vol. 17, no. 05, p. C05011, 2022.

[44] Z. Martinasek, J. Hajny, D. Smekal, L. Malina, D. Matousek, M. Kekely, and N. Mentens, "200 gbps hardware accelerated encryption system for fpga network cards," in *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*, 2018, pp. 11–17.

[45] L. Kekely, M. Spinler, S. Friedl, J. Sikora, J. Korenek, and V. Pus, "Demonstration of full-duplex packet transfers over PCI express with sustained 200 gbps throughput," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, dec 2018. [Online]. Available: https://doi.org/10.1109%2Ffpt.2018.00079

[46] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "An fpga accelerator of the wavefront algorithm for genomics pairwise alignment," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 151–159.

[47] X. Fei, Z. Dan, L. Lina, M. Xin, and Z. Chunlei, "FPGASW: Accelerating large-scale smith–waterman sequence alignment application with backtracking on FPGA linear systolic array," *Interdiscip Sci Comput Life Sci*, vol. 10, no. 1, pp. 176–188, apr 2017. [Online]. Available: https://doi.org/10.1007%2Fs12539-017-0225-8

[48] D. Smekal, S. Ricci, P. Dzurenda, and Z. Martinasek, "Privacy-enhancing cloud computing solution for big data," in *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, oct 2019. [Online]. Available: https://doi.org/10.1109%2Ficumt48472.2019.8970982

[49] M. Barrow, Z. Wu, S. Lloyd, M. Gokhale, H. Patel, and P. Lindstrom, "ZHW: A numerical CODEC for big data scientific computation," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, dec 2022. [Online]. Available: https://doi.org/10.1109%2Ficfpt56656.2022.9974258

[50] K. J. Åström, "Control system design," *Department of Mechanical and Environmental Engineering University of California Santa Barbara*, vol. 333, 2002.

[51] A. T. Fuller, "The early development of control theory," 1976.

[52] J. Watt, *Steam engine*. Art, Architecture and Engineering Library, 1781.

[53] J. C. Maxwell, "I. on governors," *Proceedings of the Royal Society of London*, no. 16, 1868. [Online]. Available: https:ep:INTECHPublishing

[54] H. S. Black, "Stabilized feedback amplifiers," *Bell system technical journal*, vol. 13, no. 1, 1934.

[55] M. T. Kara and M. E. Rizkalla, "Single op-amp proportional-integral compensator with antiwindup," in *1993 IEEE International Symposium on Circuits and Systems*. IEEE, 1993.

[56] K. J. Åström and B. Wittenmark, *Computer-controlled systems: theory and design*. Courier Corporation, 2013.

[57] M. Sobaszek, "Self-tuned class-d audio amplifier with post-filter digital feedback implemented on digital signal controller," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 3, pp. 797–805, 2019.

[58] L. Hazeleger, R. Beerens, and N. van de Wouw, "Proportional–integral–derivative-based learning control for high-accuracy repetitive positioning of frictional motion systems," *IEEE Transactions on Control Systems Technology*, vol. 29, no. 4, pp. 1652–1663, 2020.

[59] D. G. Bailey, "The advantages and limitations of high level synthesis for fpga based image processing," in *Proceedings of the 9th International Conference on Distributed Smart Cameras*, 2015.

[60] E. W. Zurita-Bustamante, J. Linares-Flores, E. Guzmán-Ramírez, and H. Sira-Ramírez, *FPGA Implementation of PID Controller for the Stabilization of a DC-DC "Buck" Converter*. Chicago, 2012.

[61] V. Tomov, I. Iliev, and V. Krasteva, "High resolution fpga pulse width modulation control of full-bridge dc–dc converters," *IET Circuits, Devices & Systems*, vol. 14, no. 7, 2020.

[62] P. Ponce, A. Molina, G. Tello, L. Ibarra, B. MacCleery, and M. Ramirez, "Experimental study for fpga pid position controller in cnc micro-machines," *IFAC-PapersOnLine*, vol. 48, no. 3, 2015.

[63] Y. F. Chan, M. Moallem, and W. Wang, "Design and implementation of modular fpga-based pid controllers," *IEEE transactions on Industrial Electronics*, vol. 54, no. 4, 2007.

[64] M.-A. Martínez-Prado, J. Rodríguez-Reséndiz, R.-A. Gómez-Loenzo, G. Herrera-Ruiz, and L.-A. Franco-Gasca, "An fpga-based open architecture industrial robot controller," *IEEE Access*, vol. 6, 2018.

[65] T.-P. Phan, P. C.-P. Chao, and Z.-W. Huang, "Design and implementation of a new torque controller via fpga for 6-dof articulated robots," *Microsystem Technologies*, 2022.

[66] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on xilinx devices," *White Paper*, 2016.

[67] T. Han, T. Zhang, D. Li, G. Liu, L. Tian, D. Xie, and Y. Shan, "Convolutional neural network with int4 optimization on xilinx device," Tech. rep. Xilinx, Tech. Rep., 2010.

[68] Z. Huang, S. Zhang, and W. Wang, "An efficient method of parallel multiplication on a single dsp slice for embedded fpgas," *IEEE Access*, vol. 7, pp. 100 993–101 008, 2019.

[69] Xilinx, "Virtex ultrascale+ fpga data sheet: Dc and ac switching characteristics," *Retrieved November 29, 2020 from https://www. xilinxcom/support/documentation/data_sheets/ds923-virtex-ultrascale-plus. pdf*, 2020.

[70] Xilinx IP, "Performance and Resource Utilization for Complex Multiplier v6.0," 2022. [Online]. Available: https://www.xilinx.com/htmldocs/ip_docs/pru_files/cmpy.html

[71] G. Akkad, A. Mansour, B. ElHassan, F. Le Roy, and M. Najem, "Fft radix-2 and radix-4 fpga acceleration techniques using hls and hdl for digital communication systems," in *2018 IEEE international multidisciplinary conference on engineering technology (IMCET)*. IEEE, 2018, pp. 1–5.

[72] I. S. Uzun, A. Amira, and A. Bouridane, "Fpga implementations of fast fourier transforms for real-time signal and image processing," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 152, no. 3, pp. 283–296, 2005.

[73] I. Hatai, R. Biswas, and S. Banerjee, "Asic implementation of a 512-point fft/ifft processor for 2d ct image reconstruction algorithm," in *IEEE Technology Students' Symposium*. IEEE, 2011, pp. 220–225.

[74] J. A. Fessler, "Model-based image reconstruction for mri," *IEEE signal processing magazine*, vol. 27, no. 4, pp. 81–89, 2010.

[75] J. Hamill, C. Michel, and P. Kinahan, "Fast pet em reconstruction from linograms," *IEEE Transactions on Nuclear Science*, vol. 50, no. 5, pp. 1630–1635, 2003.

[76] S. J. Glick and W. Xia, "Iterative restoration of spect projection images," *IEEE Transactions on Nuclear Science*, vol. 44, no. 2, pp. 204–211, 1997.

[77] A. Černỳ, "Introduction to fast fourier transform in finance," *The Journal of Derivatives*, vol. 12, no. 1, pp. 73–88, 2004.

[78] R. Buccheri and B. Sacco, "Time analysis in astronomy: Tools for periodicity searches," in *Data Analysis in Astronomy*. Springer, 1985, pp. 15–27.

[79] Y. Lu, Y. Huang, W. Xue, and G. Zhang, "Seismic data processing method based on wavelet transform for de-noising," *Cluster Computing*, vol. 22, no. 3, pp. 6609–6620, 2019.

[80] M. Frigo and S. G. Johnson, "Fftw: Fastest fourier transform in the west," *Astrophysics Source Code Library*, pp. ascl–1201, 2012.

[81] NVIDIA, "Cufft library," May 2020. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

[82] N. Corporation, "Nvidia cuda toolkit," May 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[83] E. Wold and A. Despain, "Pipeline and parallel-pipeline fft processors for vlsi implementations," *IEEE Transactions on Computers*, vol. 33, no. 05, pp. 414–426, 1984.

[84] C.-W. J. W.-C. Yeh, "High-speed and low-power split-radix fft," *IEEE Transactions on Signal Processing*, vol. 51, no. 3, pp. 864–874, 2003.

[85] J.-B. J. Fourier, "Mémoire sur la propagation de la chaleur dans les corps solides, présenté le 21 décembre 1807 à l'institut national—nouveau bulletin des sciences par la société philomatique de paris. i," in *Paris: First European Conference on Signal Analysis and Prediction*, 1807, pp. 17–21.

[86] G. Lejeune Dirichlet, "Sur la convergence des séries trigonométriques qui servent à représenter une fonction arbitraire entre des limites données." 1829.

[87] K. F. Riley, M. P. Hobson, and S. J. Bence, "Mathematical methods for physics and engineering third edition paperback set," *Mathematical Methods for Physics and Engineering-3rd Edition*, p. 1362, 2006.

[88] M. A. Moskowitz, *A course in complex analysis in one variable*. World Scientific, 2002.

[89] G. Proakis John and G. Manolakis Dimitris, "Digital signal processing: principles, algorithms, and applications," *Pentice Hall*, 1996.

[90] D. Poole, *Linear algebra: A modern introduction*. Cengage Learning, 2014.

[91] M. Gasca and T. Sauer, "On the history of multivariate polynomial interpolation," in *Numerical Analysis: Historical Developments in the 20th Century*. Elsevier, 2001, pp. 135–147.

[92] W. Fulton, *Algebraic curves: an introduction to algebraic geometry*. Addison-Wesley, 1989.

[93] R. P. Feynman, R. B. Leighton, and M. Sands, "The feynman lectures on physics; vol. i," *American Journal of Physics*, vol. 33, no. 9, pp. 750–752, 1965.

[94] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis*. Cengage learning, 2015.

[95] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.

[96] H. Sloate, "Matrix representations for sorting and the fast fourier transform," *IEEE Transactions on Circuits and Systems*, vol. 21, no. 1, pp. 109–116, 1974.

[97] A. Cortés, I. Vélez, and J. F. Sevillano, "Radix r^k ffts: Matricial representation and sdc/sdf pipeline implementation," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2824–2839, 2009.

[98] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[99] H. Prodinger, "On binary representations of integers with digits -1, 0, 1," *Integers*, pp. A8–14, 2000.

[100] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[101] M. Garrido, K. Möller, and M. Kumm, "World's fastest fft architectures: Breaking the barrier of 100 gs/s," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 4, pp. 1507–1516, 2018.