Agile Design of Generator-Based Signal Processing Hardware

by

Angie Wang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Vladimir Stojanović
Professor Aaron Parsons

Spring 2018

Agile Design of Generator-Based Signal Processing Hardware

Abstract

Agile Design of Generator-Based Signal Processing Hardware

by

Angie Wang

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Custom, application-specific implementations of digital signal processing (DSP) systems offer high performance and high energy efficiency, but require significant design and verification effort. Fast Fourier transform (FFT) processors with a broad range of performance requirements are needed for many modern-day signal processing applications, ranging from medical imaging and machine learning to communication and radio astronomy. Certain applications, including modern-day wireless communications, require runtime reconfigurability across a multitude of mixed-radix FFT sizes, high throughput, low latency, and lower power. Convolutional neural networks use multi-dimensional FFTs with stringent requirements on quantization bounds. Despite sharing underlying algorithms and hardware constructs, FFT designs are often difficult to reuse on a per-application or even per-platform basis, leading to redeveloping and reverifying conceptually similar instances. Hardware *generators* are attractive solutions for effectively balancing fine-grained control of implementation details with simple, rapidly retargetable hardware descriptions, but the existing FFT generators do not support key features like runtime reconfigurability or more general mixed-radix FFTs, limiting their applicability.

This thesis presents ACED (A Chisel Environment for DSP), a library extension to the Chisel hardware construction language and the FIRRTL (Flexible Intermediate Representation for RTL) compiler specifically created to simplify the development of hardware DSP generators. ACED allows DSP designs to be specified at a higher level of abstraction, making it easier to add new features as they become necessary. Optimization and specialization are handled via platform-specific compiler passes that promote generator reusability. The ACED library has been used to create a parameterizable memory-based, runtime-reconfigurable $2^n3^m5^k7^l$ FFT generator to support next-generation wireless systems prototyping. The generator uses a conflict-free, in-place, multi-bank SRAM design, and exploits the duality of decimation-in-frequency and decimation-in-time FFTs to support continuous data flow with $\sim 2N$ memory. The hardware itself is templated so that Chisel/Scala code can be written to add additional functionality, and users pass parameters to the hardware template via a "firmware" block. This is the essence of the Chisel DSP generator methodology.

The FFT generator has been proven via a 0.37-mm^2 LTE/Wi-Fi compatible FFT RISC-V accelerator instance with measured performance and area comparable to state-of-the-art. To demonstrate the use case of the FFT generator in a larger systems context, a low-power signal acquisition front end capable of sensing frequency-sparse signals in a 1.89-GHz bandwidth with a resolution of 175 kHz in real time has also been prototyped in a 16-nm process. The spectral analysis chip relies on mixed-radix FFTs and reconstruction via the fast Fourier aliasing-based sparse transform (FFAST) algorithm to recover signals in compressed form from a subsampled input. This thesis presents new tools and design methodologies to rapidly design DSP hardware generators—with a particular focus on FFTs—for use in emerging applications such as spectrum sensing for cognitive radio, RADAR, and more.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my advisor and dissertation chair Professor Borivoje Nikolić for his guidance and support during my PhD. He believed in me at times when I had lost confidence in myself, and gave me the encouragement necessary to keep chugging along, in the face of adversity and, sometimes, extremely stressful deadlines. I would also like to thank Professor Vladimir Stojanović and Professor Aaron Parsons for being on my dissertation committee. Professor Parsons agreed with my last-minute request, and Professor Stojanović provided me with valuable guidance when I was proposing my dissertation topic. I'm also grateful to Professor Paul Wright for serving on my qualifying examination committee.

I pivoted midway through my PhD; when I first started, I had intended to become a domain expert in analog/RF, despite knowing full well that digital systems design came to me a lot more naturally. Luckily, Bora had domain expertise in both areas, so transitioning was extremely easy. My dissertation work was motivated by a special topics class on signal processing hardware taught by Bora and Dr. Sriram Sundararajan. Although DSP systems architecture is a fairly mature field, we realized that there is actually much to be done in this space to increase designer productivity and facilitate the development of next-generation wireless systems.

During my PhD, I was able to leverage and adapt the Chisel hardware construction language created by Professor Jonathan Bachrach—who I must also thank for being on my qualifying examination committee–and his students to develop some pretty exciting DSP systems. I'm grateful for the opportunity to collaborate with Adam Izraelevitz (who happens to be amazing at project management—crucial to the success of large projects), Chick Markley, and Paul Rigge on the ACED library for hardware DSP generation in Chisel. More generally, I'm grateful for the feedback I received from Paul, Chris Yarp, and Stephen Twigg (who are walking bodies of DSP/Chisel knowledge) that impacted various design choices and research directions. I had a lot of fun learning about and implementing the FFAST algorithm developed by Professor Kannan Ramchandran's group and am grateful for the various fruitful discussions I was able to have with him and his student, Orhan Ocal. The associated chip required a significant amount of analog (and analog generator) expertise, particularly from students in Elad Alon's group—Jaeduk Han and Zhongkai Wang. I'd like to give a special shout out to Stevo Bailey and Woorham Bae, without whom the FFAST chip would not have gone out in time. In much the same way, I'd like to thank Brian Richards, who worked tirelessly with me to get a 16nm FinFET flow setup at BWRC for the first time. Furthermore, I'd like to thank Howie Mao, James Dunn, Eric Chang, Amy Whitcombe, and David Biancolin for their help in various aspects of the testing process and Nandish Mehta for providing valuable advice on the digital design flow at crazy hours in the morning.

I would like to thank Professor Elad Alon for being the principal investigator on the DARPA CRAFT project and for providing guidance on system architecture. My PhD was funded by the National Science Foundation's Graduate Research Fellowship Program, DARPA through the CRAFT program, and Intel through the Intel Science and Technology Center for Agile Hardware. My research was made possible through the BWRC, ASPIRE,

and ADEPT labs at Berkeley. I would also like to thank Professor Ali Niknejad and his student Nai-Chung Kuo for their advice when I was working on RF frontends. I'm grateful to Bill Dally for his feedback on my work at the ASPIRE retreats. It really helped to keep things in perspective and motivate me to continue.

My decision to complete a PhD in electrical engineering was influenced by my mom and dad, who first introduced me to the wonders of science and technology through at-home science experiments and trips to NASA's Jet Propulsion Laboratory. I also extend my sincere gratitude to my high school calculus teacher, Mr. DeVore, who, more importantly, ran our school's FIRST robotics team. The 2007 robotics season was my first exposure to the amazing feeling of accomplishing some complex technical feat with a team of dedicated individuals who had diverse skill sets. I would like to thank Professor Azita Emami and Professor Ali Hajimiri at Caltech for inspiring me to pursue circuits, and I am incredibly grateful to Glen George for his constant mentorship, both at Caltech and during my PhD.

Finally, I'm grateful for all of the friendships I've made throughout the years. At Berkeley, I'm glad to have gotten to befriend and receive advice from Pi-Feng Chiu, Amy Whitcombe, Amanda Pratt, Sharon Xiao, Yunsup Lee, Alberto Pugelli, Charles Wu, Yue Lu, Simon Scott, Steven Callender, Matt Weiner, Jaehwa Kwak, Andrew Townley, Ping-Chen Huang, Katerina Papadopoulou, Rachel Hochman, Chris Sutardja, Nick Sutardja, Hao-Yen Tang, Bo Zhao, Yi-An Li, Luya Zhang, Jenny Huang, Pengpeng Lu, Yongjun Li, Shaoyi Cheng, Ranko Sredojevic, Krishna Settaluri, DJ Seo, Matt Spencer, Kristel Deems, Daniel Gerber, Filip Maksimovic, Nicolas Le Dortz, Jackie Leverett, Bonjern Yang, Sameet Ramakrishnan, Nathan Narevsky, John Wright, Ben Keller, and everyone else at BWRC and ASPIRE. I would also like to thank Candy Corpus and Amber Sanchez for being great support at BWRC—for myself and everyone else as well. I'm grateful towards Juhwan Yoo, Manuel Monge, Matt and Joy Loh, Mayank Raj, Saman Saeedi, Costis Sideris, Amir Safaripour, Alex Pai, and Kaushik Dasgupta for being amazing mentors and friends during my time at Caltech and beyond. I enjoyed feeling like I was part of one really close-knit family. Lastly, I want to thank Michael Zhang, Brian Hong, Aroutin Khachaturian, Lita Yang, Jerry Chang, Steven Okai, Kelvin Fang, Helena Zhang, Jennifer Hu, Viet Anh Nguyen Huu (a.k.a Tomu), Raymond Jimenez, Jomya Lei, Daisy Lin, Noddy Eruchalu, Samson Chen, Max Wang, Angad Rekhi, Mimi Yang, Alex Hu, Brian Lawrence, Danny Bankman, Joyce Wei, Elaina Chai, Jama Mohamed, Bonnie Zhang, and my brother Allen Wang, for brightening up the time I spent in the Bay Area, regardless of where they were actually situated. To everyone I've mentioned and those I might have missed, you have my eternal gratitude for helping me survive grad school.

Chapter 1

Introduction: Agile DSP Hardware Generation. Why the Need?

1.1 FFTs: A Motivating Example for Hardware DSP Generators

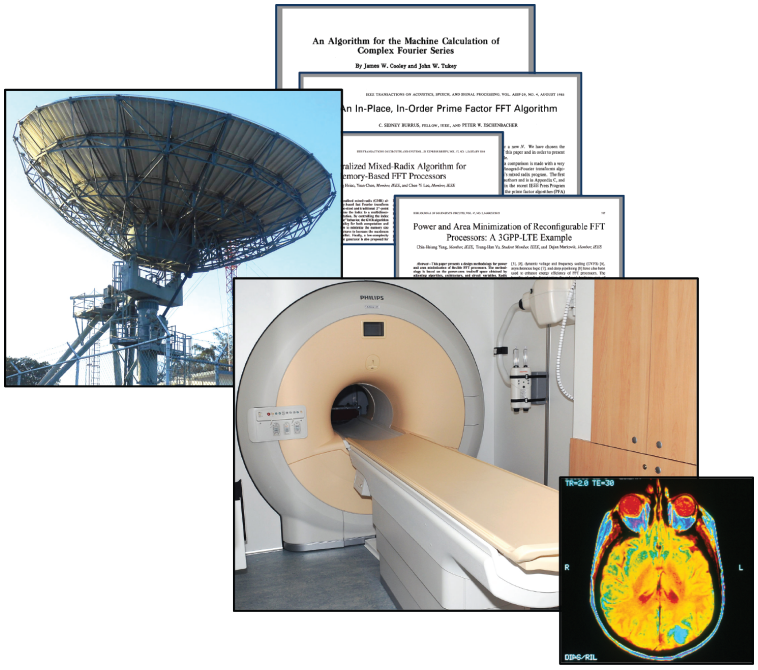

Figure 1.1: Seminal FFT papers and applications relying on FFTs. [1]

Fast Fourier transforms (FFTs) have played an integral part in various technological advancements since at least the 1960's, when J. W. Cooley and John Tukey devised the

widely-used Cooley-Tukey algorithm (CTA) to bring the computational complexity of computing a discrete Fourier transform (DFT) down from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$[2]. Dedicated hardware FFTs or FFTs implemented in software are used in a multitude of modern-day electronics systems. They serve as core digital signal processing (DSP) blocks in applications ranging from radio astronomy to image compression, magnetic resonance imaging (MRI), cognitive- and software-defined radio, and machine learning. Because each application has different requirements in terms of power, precision, throughput, latency, etc., FFTs must be custom-tailored to a given system. Although the same fundamental algorithm is used for converting time or spatial representations into the frequency domain, hardware FFTs have been optimized and *re*-implemented countless times.

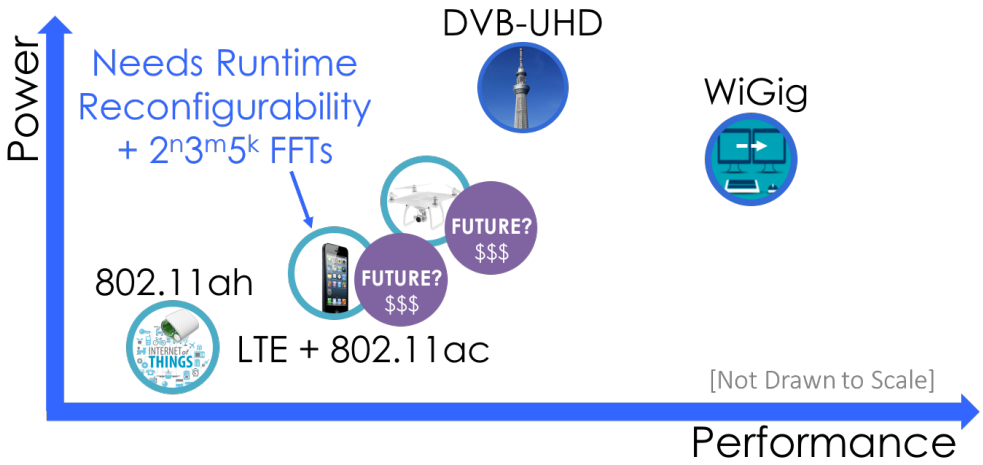

Figure 1.2: Cartoon illustration of power vs. performance requirements for different wireless standards. 802.11ah is used for IoT devices, which require low power, low data rate operation. Digital TV broadcasts require high power radios that reach large swaths of area. WiGig is intended for close-proximity, high data rate video and audio streams.

FFT Length	64	128	256	512
Bandwidth/IO Rate (MHz)	20	40	80	160

Table 1.1: Wi-Fi 802.11ac FFT requirements. The symbol duration is 3.2μs with an additional 800ns guard interval. FFT computation can be performed across a total of 4μs.

Even within the narrow application scope of orthogonal frequency-division multiplexing (OFDM) wireless communication, different standards necessitate highly customized FFT configurations (Fig. 1.2). Wi-Fi basebands must support different channel bandwidths and modulation schemes over a symbol duration of 3.2μs, requiring runtime reconfigurability across different 2^n FFT sizes and multiple data rates (Table 1.1). Basic OFDM for the

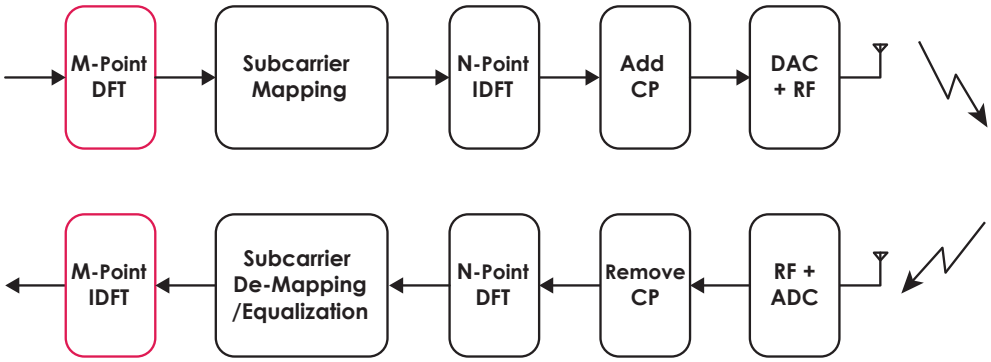

Figure 1.3: OFDM transmitter and receiver. The additional DFT/IDFT blocks in pink are required for single-carrier frequency-division multiple access (SC-FDMA), used by LTE [3].

FFT Length	128	256	512	1024	1536	2048
Bandwidth (MHz)	1.25	2.5	5	10	15	20
IO Rate (MHz)	1.92	3.84	7.68	15.36	23.04	30.72

Table 1.2: LTE FFT requirements. The symbol duration is 66.67μs. An additional 4.76μs is reserved for the cyclic prefix.

12	96	216	384	648	972
24	108	240	432	720	1080
36	120	288	480	768	1152
48	144	300	540	864	1200
60	180	324	576	900	1296
72	192	360	600	960	

Table 1.3: $2^n3^m5^k$ FFT sizes required for LTE SC-FDMA precoding.

Long-Term Evolution (LTE) standard requires a different set of data rates and FFT sizes (Table 1.2) [4]. Radios that support LTE's single-carrier frequency-division multiple access scheme (SC-FDMA), which relaxes the peak-to-average power ratio at the transmitter, must additionally perform mixed-radix $2^n3^m5^k$ FFTs (Table 1.3).

1.2 Background and Prior Art

A significant amount of effort has gone into the optimization of FFTs in software. The most well-known work in this space is the C subroutine library FFTW, also known as the "Fastest Fourier Transform in the West" [5]. It is able to compute one- or many-dimensioned DFTs of arbitrary input size and can be optimized for either real or complex data. The

backbone behind FFTW is a "codelet generator" that produces optimized C code for specific FFT sizes on a given hardware architecture. The resultant C code has performance that is comparable to, if not better than, most other software FFT implementations. The generator can target different architectures, exhibiting a high degree of portability.

Although suitable for many applications, when real-time computation, runtime configuration, and energy efficiency are required, as is the case with wireless communication, even the most optimized software FFT implementation cannot outperform its hardware counterpart. To satisfy general industry needs, FPGA and ASIC companies provide reconfigurable, but costly/closed-platform, FFT IP cores [6], [7]. These solutions are not extensible to new or more exotic standards such as Chinese terrestrial Digital TV (which requires a 3,780-point FFT and radix-7 butterfly support) [8]. Additionally, runtime-reconfigurable cores are not resource-optimized for particular applications, which is problematic when users only need a subset of FFT sizes and are otherwise severely hardware constrained. This is a pain point when prototyping next-generation wireless on resource-constrained devices.

Academia has attempted to address these problems with parameterized FFT generators [9], [10]. These generators ideally exploit our understanding of algorithms and architecture regularity to create hardware-optimized FFT engines for any application, enabling design space exploration and rapid systems prototyping, while supporting state-of-the-art performance. Highly parameterized generators should allow throughput/area trade-offs, memory access methods, fixed point optimizations [11], etc. to be studied in detail. However, existing generators are sub-optimal/have incomplete feature sets. They do not support runtime reconfigurability/non-2^n FFTs and are thus inadequate for applications like software-defined radio (SDR). In particular, Carnegie Mellon's Spiral is sub-optimal in its memory requirements, necessitating (best-case) $4N$ memory to compute an N-point FFT [9] with a radix-2 iterative architecture (Table 1.4). It relies on the fixed-interconnect Pease algorithm [12], which makes it ill-suited for runtime-reconfigurability. Additionally, Stanford's Genesis2 FFT generator does not support IO unscrambling.

	Radix-2 Streaming	**Radix-2 Iterative**
Data Memory	7.99N	4N
Twiddle ROM	0.99N	N
Clock Cycles	1,024	11,287
Multipliers	40	4

Table 1.4: Generated FFT configurations from Spiral for fixed $N = 2,048$.

Significantly more research effort has been spent on the optimization of individual FFT instances for particular applications. Several chip implementations are highlighted below.

- In [13], Yang *et al.* developed a reconfigurable FFT based off of the pipelined single-path delay feedback (SDF) architecture to support 128- to 2048-point FFT sizes for 3GPP-LTE. It uses radix factorization to achieve high energy efficiency while supporting runtime reconfigurability.

- In [14], Hickish *et al.* summarize a decade of development of open-source hardware DSP generators targeting Xilinx FPGAs for the radio astronomy community. These include a radix-2 FFT generator that supports multiple parallel data streams with $\leq 5N/2$ memory [15]. An FFT instance generated by this FFT generator was mapped using an automated design flow to an application-specific integrated circuit (ASIC) by Richards *et al.* [16] for use in a high-performance 4096-point digital spectrometer.

- In [17], G. Yahalom built runtime-reconfigurable, mixed-radix FFT and IFFT instances for an LTE uplink channel. Like [13], the FFT instances rely on the SDF architecture, but DFT sizes of 12-1200 are supported for single-carrier frequency-division multiple access (SC-FDMA).

- In [18], Xia *et al.* implemented memory-based FFT processors to support runtime reconfigurability among FFT sizes of 12-1296 and 128-2048 for LTE. The work relies on the prime factor algorithm (PFA) for FFT decomposition and a custom conflict-free memory addressing scheme. It is able to achieve continuous data flow, although $3N$ memory is required to support SC-FDMA FFT sizes.

- In [19], Chen *et al.* also implemented memory-based FFT processors for FFT lengths of 12-1296 and 128-2048. These processors achieve continuous data flow with only $2N$ memory through the use of the PFA + CTA and a high-radix butterfly unit.

Much of this research has tended to focus on one aspect of the FFT design at a time, without considering implications on overall system complexity or performance. Table 1.5 highlights particular blocks targeted for optimization in recent literature and potential costs of these optimizations.

	Enhancements	**Possible Costs**
Qureshi *et al.* [20]	Reconfigurable WFTA BF	
Richardson *et al.* [21]	Multi-BF Conflict-Free Calculation Schedules	I/O Addressing[1]
		Twiddle ROM Addressing
Chen *et al.* [19]	PFA + CTA to Reduce Non-Trivial Twiddles	Complex Calculation Addressing Scheme
Hsiao *et al.* [22]	In-Place IO Mem. Reduction	
	Single-BF Conflict-Free Addressing	

Table 1.5: Areas of focus for hardware FFT optimizations in recent literature. Genesis2 data is pre-stored in memory.[1]

One of the more "unique" hardware FFT instances was reported by Abari *et al.* in [23]. The instance is capable of calculating a 746,496-point FFT with the caveat that the frequency spectrum being computed must be sparse—only 0.1% of the spectrum should

be occupied. This hardware FFT relies on MIT's sparse Fourier transform algorithm [24], [25]—which happens to be quite similar to the Fast Fourier Aliasing-based Sparse Transform (FFAST) developed by Pawar *et al.* at Berkeley—and coprime subsampling. The same authors applied sparse FFT concepts to a GHz-wide spectrum analyzer called BigBand [26], capable of detecting signals at more realistic sparsity levels. More traditional spectrum analyzers like those used in the Microsoft spectrum observatory sequentially scan small bandwidth windows, making real-time operation difficult and leading to missed signals—particularly those that are short-lived. BigBand is instead able to capture a GHz of spectrum in real time using commodity subsampling analog-to-digital converters (ADCs) instead of power-hungry, costly, and low-resolution, but high-speed ADCs.

In designing new and complex systems like BigBand, it is imperative that focus be spent on systems-level optimizations that enhance the performance of individual blocks, which themselves can be improved over time. For example, BigBand is not reliant on high-speed ADCs, but can achieve real-time performance, because the digital backend—consisting of subsampled frequency bucketization, frequency estimation, and collision (due to aliasing) resolution components and reliant on several small FFTs instead of one large one—can resolve information "lost" in the initial subsampling stage.

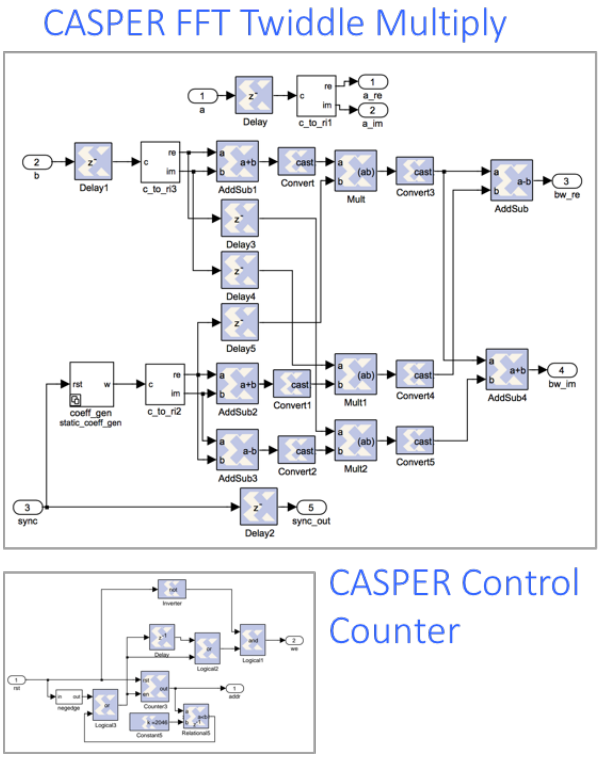

Figure 1.4: Simulink-generated blocks for a mars spectrometer designed at the Berkeley Wireless Research Center.

The design of complex systems for next-generation applications greatly benefits from tools that enhance designer productivity, particularly at the systems level. In [16], researchers at the Berkeley Wireless Research Center experimented with building DSP instances using Simulink data flow graphs. However, graphical representations do not always equate to the most intuitive representations, especially as designs balloon in complexity (Fig. 1.4). In particular, mathematical functions like complex twiddle multiplication are often much easier to grasp in text. Although Simulink flows are decent at mapping designs to generic hardware, specialization—like mapping to ASIC SRAMs—becomes challenging, because the final RTL is not transparent and additional tools are required, potentially resulting in translation errors. This suggests that there is a need to build tightly integrated tools that operate at various design abstraction levels. Additionally, much like FFTW in the software space, generators must exist in the hardware space to:

- Enhance designer productivity and encourage rapid design development/deployment by supporting a high degree of parameterization, and
- Enable automatic platform specialization and optimization.

1.3 Agile Hardware Design: Building Hardware Generators

It should be clear that one-off design instances are not resource- and time-efficient and do not provide the agility required for fast systems development in the modern age. For example, the design and verification of a single FFT instance for a particular application may take months, if not years, to complete. To spur innovation at the hardware systems level, hardware engineers need to be comfortable with an intersection of low-level circuit design, algorithms, and software tools and methodology. More specifically, they should be taking a "big-picture" approach to hardware design, building hardware generators (Fig. 1.5), rather than single instances that are micro-optimized for a particular application. As with designing single instances, generator design must focus on systems-level optimization.

Unfortunately, as exemplified in the previous section, neither commercial nor academic tools have catered to the needs of designers of hardware *DSP generators*, and thus support limited "agility." Verilog and VHDL support minimal generator capability with simple parameterization and "for-generate" statements that are prone to indexing errors. Numeric processing with powerful scientific libraries and tools like Matlab is not integrated. As a concept is funneled down various abstraction layers along its path to hardware, several manual translation and verification steps must occur. Miscommunication and mistranslation at abstraction and tool boundaries (e.g., porting constants in the wrong format or defining incorrect fixed-point representations) incur significant penalties in the technology development timeline, resulting in algorithms becoming "stale" before their hardware implementations are realized.

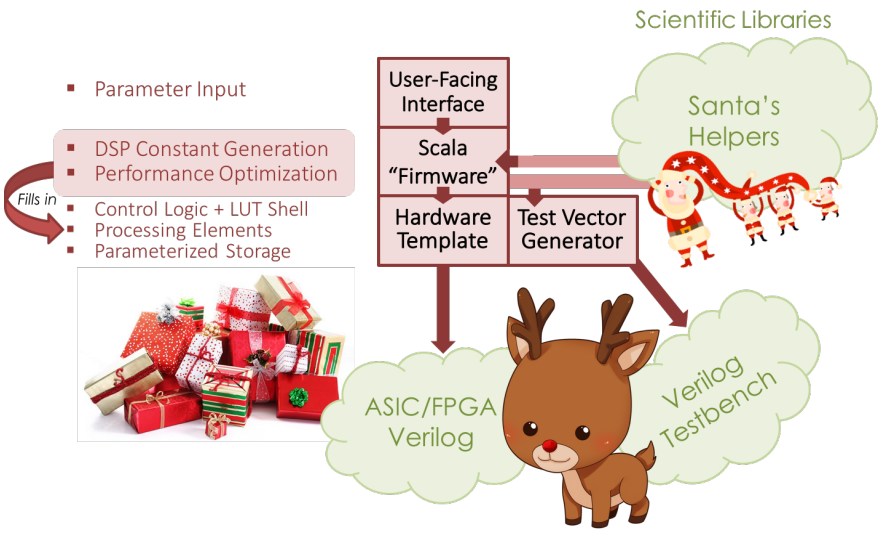

Figure 1.5: In the approach highlighted in this thesis, hardware generators consist of a user-facing interface that allows users to input design constraints (e.g., FFT sizes). These constraints are then used by "firmware" to determine optimal hardware allocation. Calculated parameters are fed into a hardware template to generate synthesizable Verilog. A similar mechanism is used to automatically generate tests associated with a parameterized hardware instance. Scientific libraries built for Scala can be directly used at each generator layer.

The Chisel hardware construction language [27] improves upon some of these limitations. It supports high levels of parameterization and, due to its functional programming nature, more intuitive and concise syntax for structured math. It promotes a unified development environment, with its own Tester class for validation and verification, but, up until recently, did not support DSP number types and verification. Chisel provided no mechanism for floating-point to fixed-point translation (which requires significant design effort) and math-to-hardware was extremely error prone, because Chisel, like Verilog, merely understood bits rather than numbers.

To address these pain points, we enhanced Chisel with the library ACED: A Chisel Environment for DSP. ACED has been used in the design of various hardware DSP generators that have been verified in silicon. It natively supports DSP-specific number representations, enabling features like automatic bitwidth reduction via static/dynamic range analysis, while still affording the designer tight control over implementation details. It supports easy design parameterization with DSP-specific typeclasses, facilitating code reuse across platforms and applications. This key feature also enables architectures to be validated in floating-point and a one-step translation to fixed-point hardware (and associated tests) for verification and evaluation of hardware metrics like quantization without rewriting code.

1.4 Thesis Overview

This thesis focuses on the design of DSP—and in particular, FFT—generators for use in next-generation wireless systems and additionally introduces new tools to simplify the development of such generators. This thesis presents a design methodology and library utilizing the Chisel hardware construction language [27] to support rapid prototyping of DSP hardware generators with a focus on high-level, minimal-cost abstraction, unobtrusive optimization that preserves design re-usability, and portable systems modeling and testing.

To support SDR and beyond, in [28], we designed a compile-time parameterizable generator of memory-based, runtime-reconfigurable $2^n3^m5^k7^l$ FFTs using Chisel and the ACED hardware DSP library [29]. The generator uses a conflict-free, in-place, multi-bank SRAM design and supports continuous data flow. Generated FFT instances use less data/twiddle memory than comparable (iterative, $N = 2,048$) instances from the Spiral FFT generator (50%/25% savings, respectively) [9].

A 0.37mm^2 FFT accelerator meeting LTE/Wi-Fi requirements has been generated and taped out in 16nm FinFET. The accelerator is optimized for radix-2/3/4/5 butterfly reuse [20] and continuous data flow with just $2.23N_{max}$ total SRAM (4,576×48-bit, 24-bit real/imaginary). It requires a twiddle LUT depth of only 1,718 ($0.84N_{max}$), despite supporting all LTE/Wi-Fi FFT configurations. The FFT accelerator is attached to a RISC-V Rocket core [30] to simplify testing with C code, and the FFT + Rocket system has been verified to function with a core clock up to 940MHz. The system operates at 22.6mW to meet the more stringent Wi-Fi specifications. Measurement results show that generator-based designs are *competitive with state-of-the-art*.

Furthermore, to demonstrate generator versatility, the same FFT generator has been exercised in a real-time ($< 20\mu$s runtime), 1.89-GHz bandwidth, 175-kHz resolution sparse spectral analysis system-on-chip (SoC), which uses the FFAST sparse FFT algorithm developed by S. Pawar in K. Ramchandran's group at U.C. Berkeley [31]. The SoC is able to recover realistic spectra with up to 3.2% sparsity (compared to 0.1% in [23]). This algorithm relies on "co-prime subsampling" and thus necessitates the use of mixed-radix $2^n3^m5^k$ FFTs. The SoC, taped out in 16nm, is the first fully-integrated demonstration of a sparse spectral analysis system, consisting of custom subsampling SAR ADCs designed with the Berkeley Analog Generator [32] at the frontend, FFTs generated via the aforementioned Chisel generator, a Chisel-generated spectrum reconstruction backend, and a Rocket RISC-V processor used for simplifying testing with C code and performing additional post-processing. The spectrum reconstruction backend consists of common DSP instances like CORDIC that were also designed as generators using ACED.

Analog and digital hardware generators enabled the rapid 16-nm implementation of both chips in less than two months each.

Chapter 2

A Runtime-Reconfigurable, Mixed-Radix Hardware FFT Generator

2.1 Overview

This chapter takes a bottom-up approach to the design of hardware FFT generators. In order to build hardware FFT instances and generators, it is first necessary to understand the underlying algorithms and architectures (e.g., memory-based vs. pipelined) behind them. Here, we outline widely used algorithms for enabling efficient FFT computations—the Cooley-Tukey algorithm, the prime factor algorithm (specifically used to support DFT sizes that can be decomposed into coprime numbers), and the Winograd's Fourier transform algorithm. Key properties of the algorithms, which are used to optimize memory/compute resources and develop the butterfly datapath and control blocks like the index vector generator, twiddle address generator, etc., are highlighted. Finally, we show how the individually described functional blocks are packaged into a compile-time-parameterized generator of runtime-reconfigurable, mixed-radix hardware FFTs.

2.2 Computing the Discrete Fourier Transform via FFTs

The discrete Fourier transform (DFT), which transforms the N-length complex sequence of $(x_n)_{n=0}^{N-1}$ into the complex sequence $(X_k)_{k=0}^{N-1}$, is defined by [33]

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}, \quad W_N^{nk} = e^{-\frac{j2\pi nk}{N}}. \tag{2.1}$$

The DFT is an important tool in applications such as wireless communication, computer vision, radio astronomy, medical imaging, etc., that require time $\leftrightarrow$ frequency domain transforms of finite-duration, discretely sampled data. Because the computational complexity of (2.1) is $\mathcal{O}(N^2)$, applications that deal with large input sequences benefit from fast algorithms for DFT computation known collectively as fast Fourier transforms (FFT). This work attempts to provide a thorough explanation of how the FFT can be mapped into an efficient hardware *generator* with wide application coverage.

2.2.1 Cooley-Tukey FFT

The Cooley-Tukey algorithm (CTA) is synonymous with the FFT [2]. It applies a recursive "divide and conquer" one-dimension ($N = N_1 N_2$) to two-dimension (N_1, N_2) decomposition strategy to reduce the computational complexity of calculating a DFT to $\mathcal{O}(N \log N)$.

2.2.1.1 Decimation-in-Frequency FFT

The CTA is often associated with the following index mappings (detailed in Section 2.2.2):

$$n_{DIF} = N_2 n_1 + n_2 \tag{2.2}$$
$$k_{DIF} = k_1 + N_1 k_2\,, \tag{2.3}$$

where $n_1, k_1 \in [0, N_1)$, $n_2, k_2 \in [0, N_2)$, and $N_1, N_2 \in \mathbb{Z}$. When recursively applied for $N = N_1 \times (N_2 \times N_3)$, the resulting index maps are

$$n_{DIF} = (N_2 N_3) n_1 + \tilde{n}_{2,DIF}\,, \qquad \tilde{n}_{2,DIF} = N_3 n_2 + n_3 \tag{2.4}$$
$$k_{DIF} = k_1 + N_1 \tilde{k}_{2,DIF}\,, \qquad \tilde{k}_{2,DIF} = k_2 + N_2 k_3\,, \tag{2.5}$$

where $\tilde{n}_2, \tilde{k}_2 \in [0, N_2 N_3)$. Substituting (2.2) and (2.3) into (2.1) so that $x_n \rightarrow x[n_1, n_2]$ and $X_k \rightarrow X[k_1, k_2]$ leads to the decimation-in-frequency (DIF) CTA equation [22], [19]:

$$\begin{aligned} X[k_1, k_2] &= \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_N^{(N_2 n_1 + n_2)(k_1 + N_1 k_2)} \\ &= \sum_{n_2=0}^{N_2-1} \left\{ \left\{ \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right\} W_N^{n_2 k_1} \right\} W_{N_2}^{n_2 k_2} \\ &= \sum_{n_2=0}^{N_2-1} \left\{ y[k_1, n_2] W_N^{n_2 k_1} \right\} W_{N_2}^{n_2 k_2}\,. \end{aligned} \tag{2.6}$$

Recall that W_N^a for $a \in [0, N)$ are the Nth roots of unity. $W_N^{aN} = e^{-j2\pi a} = 1$ for $a \in \mathbb{Z}$. Equation (2.6) indicates that the N-point DFT can be obtained by

1. Computing N_2 sub-DFTs of size N_1 in the first calculation stage (where, for fixed n_2, the inputs/outputs to/from each N_1-point DFT are $x[n_1, n_2]$ and $y[k_1, n_2]$ respectively),

2. Performing N twiddle rotations as indicated by $W_N^{n_2 k_1}$ (some of which are trivial multiplications by $W_N^0 = 1$), and

3. Computing N_1 sub-DFTs of size N_2 in the second calculation stage (where k_1 is fixed at the input of each N_2-point DFT, and the outputs are $X[k_1, k_2]$).

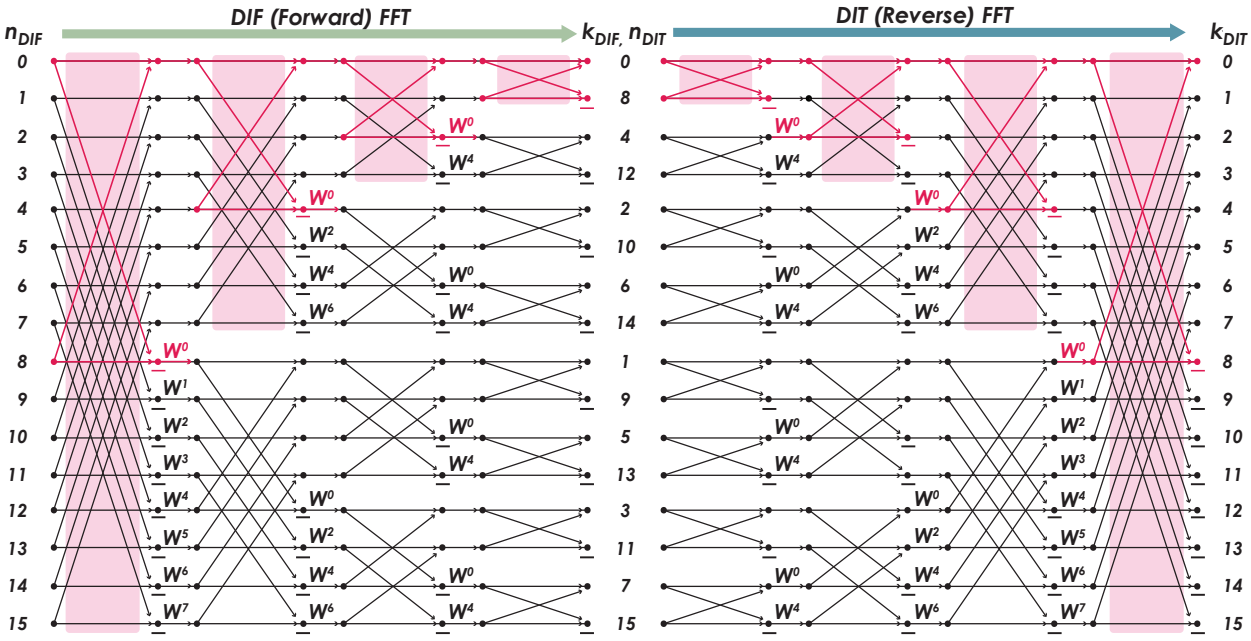

Figure 2.1: $N = 16$, radix-2 DIF (a) and DIT (b) CTA signal-flow graphs [34]. One butterfly "group" (containing $N/2^i$ butterflies for the DIF FFT) in each of the four $N/2^{i-1}$, $i \in [1, 4]$ decomposition stages is highlighted. The number of butterfly groups in the ith stage is given by 2^{i-1}. Different butterfly groups within the same stage share the same twiddle factors. Example radix-2 DFT butterflies are in pink.

For $n', k' \in \left[0, \frac{N}{N_1}\right)$, (2.6) can also be written as

$$X[k_1 + N_1 k'] = DFT_{\frac{N}{N_1}} \left\{ \left\{ \sum_{n_1=0}^{N_1-1} x \left[\frac{N}{N_1} n_1 + n' \right] W_{N_1}^{n_1 k_1} \right\} W_N^{n' k_1} \right\} . \tag{2.7}$$

In particular, when $N_1 = 2$, the recursive formulas for describing an N-point DFT in terms

of $N/2$-point DFTs are derived to be [34]

$$X[2k'] = DFT_{\frac{N}{2}} \left\{ x\left[n'\right] + x\left[n' + \frac{N}{2}\right] \right\} \tag{2.8}$$

$$X[2k'+1] = DFT_{\frac{N}{2}} \left\{ \left(x\left[n'\right] - x\left[n' + \frac{N}{2}\right] \right) W_N^{n'} \right\}, \tag{2.9}$$

where the sum and difference correspond to radix-2 butterfly operations. If $N = 2^n$, those $N/2$-point DFTs can be calculated with $N/4$-point DFTs, and so forth, until the last decomposition stage involving a 2-point DFT is reached. Using (2.8) and (2.9), the summations in (2.6) can be unrolled and represented in a fully parallel form by a *radix-2* DIF CTA signal-flow graph (SFG), as seen for $N = 16$ in Fig. 2.1a. The index mappings in (2.2) and (2.3) are recursively applied, resulting in

$$n_{DIF} = 8n_1 + (4n_2 + (2n_3 + n_4)) \tag{2.10}$$

$$\begin{aligned} k_{DIF} &= k_1 + 2(k_2 + 2(k_3 + 2k_4)) \\ &= k_1 + 2k_2 + 4k_3 + 8k_4\,. \end{aligned} \tag{2.11}$$

The number of decomposition stages needed for an $N = 2^n$ radix-2 FFT is $\log_2 N$ (4 for $N = 16$). This determines the computational complexity and is set by the recursion depth.

Due to hardware area and power constraints, typically, SFGs are translated into sequential (i.e., memory-based or pipelined) architectures with potentially some degree of parallelism, as addressed in Section 2.4.1. When in-place computation is performed for a DIF radix-2 FFT, the outputs of the individual radix-2 DFT butterflies are stored at the locations of corresponding inputs in the case of a memory-based architecture. In this case, as evidenced by (2.10), (2.11), and Fig. 2.1a, the time-domain inputs are in order and frequency-domain outputs appear in "bit-reversed" order, e.g., time sample $1 = 0_20_20_20_21_2 \rightarrow$ frequency sample $8 = 1_20_20_20_2$. Thus, input/output streaming cannot be performed in-place, and a separate output unscrambling block is needed. To support continuous data flow, 1) input streaming, 2) the N-point FFT calculation, and 3) output unscrambling must all be occurring simultaneously, necessitating a "ping-pong" memory scheme with nominally $3N$ memory (Fig. 2.2).

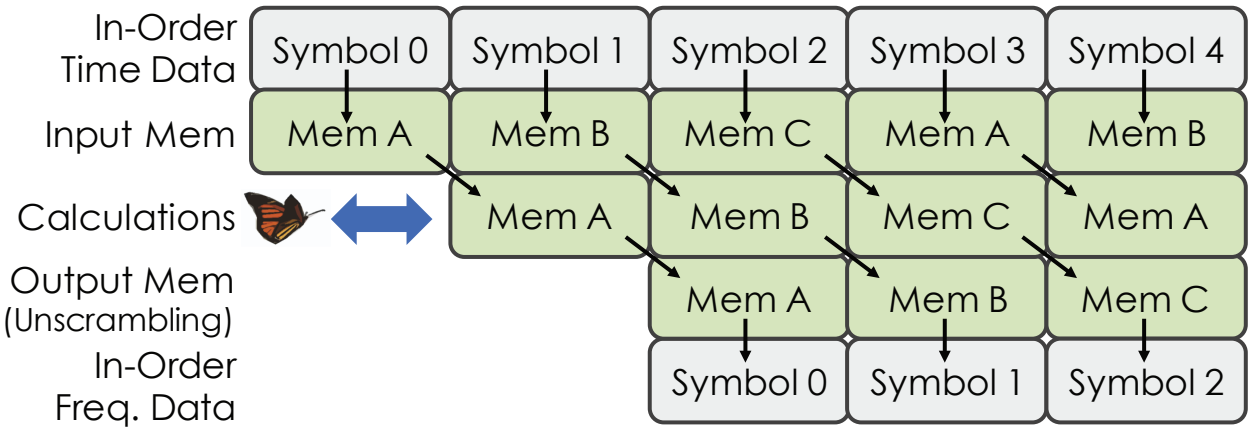

Figure 2.2: N-point DIF FFT memory access timing to support continuous data flow with $3N$ memory. Memories A-C are simultaneously accessed. One symbol contains N data samples.

2.2.1.2 Decimation-in-Time FFT and DIF $\leftrightarrow$ DIT Duality

The SFG for an $N = 16$ decimation-in-time (DIT) radix-2 CTA implementation is illustrated in Fig. 2.1b. In general, DIT index mappings, which can be obtained from "reversing" the DIF input/output mappings in (2.2) and (2.3), are given by [35]

$$n_{DIT} = n_1 + N_1 n_2 \tag{2.12}$$
$$k_{DIT} = N_2 k_1 + k_2 \,. \tag{2.13}$$

Note that reversing the DIF mappings, or equivalently, reversing the decomposition order for DIF $\rightarrow$ DIT, implies that $N_2 \rightarrow N_1$, $N_1 \rightarrow N_2$, $n_2, k_2 \rightarrow n_1, k_1$, and $n_1, k_1 \rightarrow n_2, k_2$. When recursively applied for $N = (N_3 \times N_2) \times N_1$, the following index maps are obtained:

$$n_{DIT} = n_1 + N_1 \tilde{n}_{2,DIT} \,, \qquad \tilde{n}_{2,DIT} = n_2 + N_2 n_3 \tag{2.14}$$
$$k_{DIT} = (N_2 N_3) k_1 + \tilde{k}_{2,DIT} \,, \qquad \tilde{k}_{2,DIT} = N_3 k_2 + k_3 \,. \tag{2.15}$$

Therefore, the DIT index mappings for $N = 16$ are

$$n_{DIT} = n_1 + 2n_2 + 4n_3 + 8n_4 \tag{2.16}$$
$$k_{DIT} = 8k_1 + 4k_2 + 2k_3 + k_4 \,. \tag{2.17}$$

Substituting (2.12) and (2.13) into (2.1) leads to the DIT equation [22], [19]:

$$X[k_1, k_2] = \sum_{n_1=0}^{N_1-1} \left\{ W_N^{n_1 k_2} \left\{ \sum_{n_2=0}^{N_2-1} x[n_1, n_2] W_{N_2}^{n_2 k_2} \right\} \right\} W_{N_1}^{n_1 k_1} \,. \tag{2.18}$$

For $n', k' \in \left[0, \frac{N}{N_1}\right)$, (2.18) can be written as

$$X\left[\frac{N}{N_1}k_1 + k'\right] = \sum_{n_1=0}^{N_1-1}\left\{W_N^{n_1k'}\left\{\sum_{n'=0}^{N/N_1-1} x\left[n_1 + N_1n'\right] W_{N/N_1}^{n'k'}\right\}\right\} W_{N_1}^{n_1k_1}. \tag{2.19}$$

When $N_1 = 2$, the DIT formulas can be expressed by [34]

$$X[k'] = DFT_{\frac{N}{2}}\{x[2n']\} + W_N^k DFT_{\frac{N}{2}}\{x[2n'+1]\} \tag{2.20}$$

$$X[k'+N/2] = DFT_{\frac{N}{2}}\{x[2n']\} - W_N^k DFT_{\frac{N}{2}}\{x[2n'+1]\}. \tag{2.21}$$

A radix-2 DIT FFT also requires nominally $3N$ memory. However, (2.12) and (2.13) indicate that the inputs are in bit-reversed order, i.e., the time-domain input is decimated and divided into even/odd samples, and the outputs are in order, as per (2.20) and (2.21). An important observation (Fig. 2.1) is that DIF and DIT FFT SFGs are mirror images of each other, both in terms of input/output ordering, as described earlier, and in terms of the location of twiddle multiplications. As indicated by (2.8) and (2.9), twiddle multiplication occurs after the DIF radix-2 butterfly; however, according to (2.20) and (2.21), it occurs before the sum and difference operations of a radix-2 butterfly in a DIT FFT. By exploiting the DIF/DIT input/output duality, assuming an N-point FFT calculation can be completed in N clock cycles, continuous data flow with in-place IO and output unscrambling can be achieved with only $2N$ memory, as shown in Fig. 2.3. In this case, the FFT hardware alternates between performing DIF and DIT FFTs every $2a$th symbol e.g., symbol 0's $x_0[n]$ is streamed into Memory A in natural order (corresponding to n_{DIF}), a DIF FFT calculation is performed, and the result is streamed out from Memory A using the bit-reversed k_{DIF} mapping. As $X_0[k]$ is streamed out, symbol 2's $x_2[n]$ is simultaneously read into the same location via the n_{DIT} index mapping for a subsequent DIT FFT calculation. More formally, reversing the decomposition order in such a way allows $x_{2a+2}[n], x_{2a+3}[n]$ to be written to memory as $X_{2a}[k], X_{2a+1}[k]$ is read out, in order, with $n = k$ [28], [22]. While calculations are being performed on data in Memory A, data are being input/output to/from Memory B.

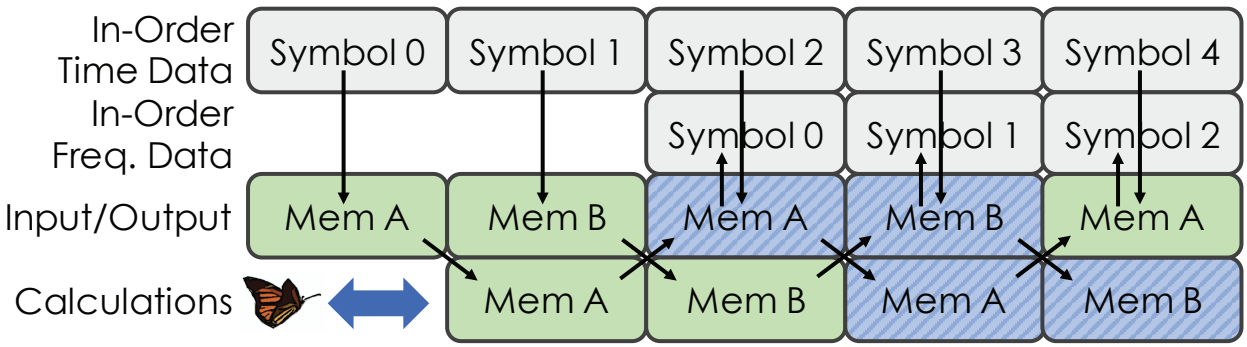

Figure 2.3: Memory access timing to support in-order, continuous IO with $2N$ memory. Memories A and B are simultaneously accessed, while DIF (green) $\leftrightarrow$ DIT (blue) alternates every $2a$th cycle.

2.2.1.3 Generalizing the CTA Index Mapping: Digit Reversal

Table 2.1: n, k index mappings for radix-2 and radix-4 $N = 16$ FFTs.

n	**Base-2** n	**Base-4** n	**Base-2** k	**Base-4** k
0	$0_20_20_20_2$	0_40_4	$0 = 0_20_20_20_2$	$0 = 0_40_4$
1	$0_20_20_21_2$	0_41_4	$8 = 1_20_20_20_2$	$4 = 1_40_4$
2	$0_20_21_20_2$	0_42_4	$4 = 0_21_20_20_2$	$8 = 2_40_4$
3	$0_20_21_21_2$	0_43_4	$12 = 1_21_20_20_2$	$12 = 3_40_4$
4	$0_21_20_20_2$	1_40_4	$2 = 0_20_21_20_2$	$1 = 0_41_4$
5	$0_21_20_21_2$	1_41_4	$10 = 1_20_21_20_2$	$5 = 1_41_4$
6	$0_21_21_20_2$	1_42_4	$6 = 0_21_21_20_2$	$9 = 2_41_4$
7	$0_21_21_21_2$	1_43_4	$14 = 1_21_21_20_2$	$13 = 3_41_4$
8	$1_20_20_20_2$	2_40_4	$1 = 0_20_20_21_2$	$2 = 0_42_4$
9	$1_20_20_21_2$	2_41_4	$9 = 1_20_20_21_2$	$6 = 1_42_4$
10	$1_20_21_20_2$	2_42_4	$5 = 0_21_20_21_2$	$10 = 2_42_4$
11	$1_20_21_21_2$	2_43_4	$13 = 1_21_20_21_2$	$14 = 3_42_4$
12	$1_21_20_20_2$	3_40_4	$3 = 0_20_21_21_2$	$3 = 0_43_4$
13	$1_21_20_21_2$	3_41_4	$11 = 1_20_21_21_2$	$7 = 1_43_4$
14	$1_21_21_20_2$	3_42_4	$7 = 0_21_21_21_2$	$11 = 2_43_4$
15	$1_21_21_21_2$	3_43_4	$15 = 1_21_21_21_2$	$15 = 3_43_4$

Although this section has focused on radix-2 FFT implementations for computing $N = 2^n$-point DFTs, the CTA also applies to higher order radix (e.g., radix-4, as in Fig. 2.4) and mixed-radix ($N = 2^n3^m5^k...$) DFTs. For non-radix-2 CTA FFTs, input and output index mappings are more generally "digit-reversed," and DIF/DIT duality still applies. General-

Table 2.2: n, k index mappings for radix-2 and radix-4/2 $N = 8$ FFTs.

n	**Base-2** n	**Base-4/2** n	**Base-2** k	**Base-2/4** k
0	$0_20_20_2$	0_40_2	$0 = 0_20_20_2$	$0 = 0_20_4$
1	$0_20_21_2$	0_41_2	$4 = 1_20_20_2$	$4 = 1_20_4$
2	$0_21_20_2$	1_40_2	$2 = 0_21_20_2$	$1 = 0_21_4$
3	$0_21_21_2$	1_41_2	$6 = 1_21_20_2$	$5 = 1_21_4$
4	$1_20_20_2$	2_40_2	$1 = 0_20_21_2$	$2 = 0_22_4$
5	$1_20_21_2$	2_41_2	$5 = 1_20_21_2$	$6 = 1_22_4$
6	$1_21_20_2$	3_40_2	$3 = 0_21_21_2$	$3 = 0_23_4$
7	$1_21_21_2$	3_41_2	$7 = 1_21_21_2$	$7 = 1_23_4$

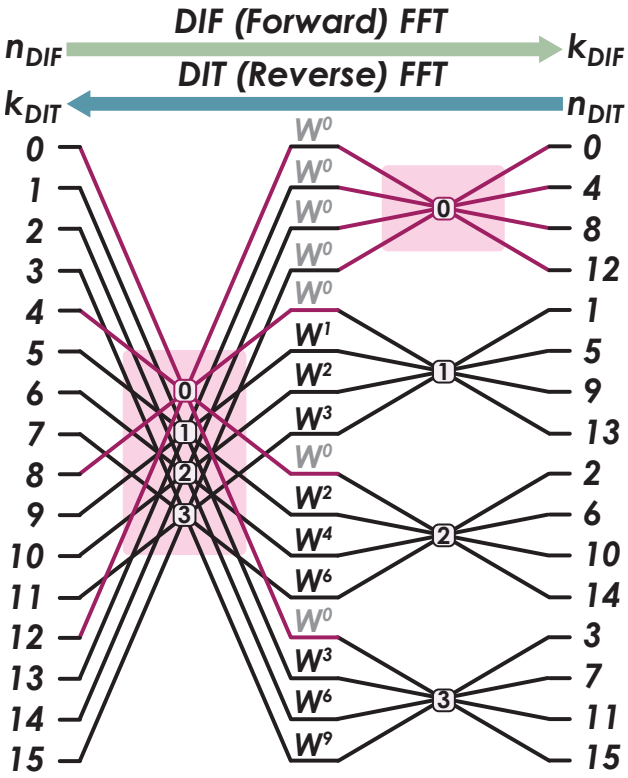

Figure 2.4: $N = 16$, radix-4 DIF (forward) and DIT (reverse) CTA signal-flow graph.

izing (2.4) for mixed-radix results in

$$n_{DIF} = \left[\sum_{i=1}^{S-1}\left(n_i \prod_{l=i+1}^{S} N_l\right)\right] + n_S \tag{2.22}$$

when a DFT of size $N = N_1N_2...N_S$ is decomposed into S stages. In mixed-radix form, n is represented as $n_1n_2...n_S$, where n_i is a digit with base N_i and n_S is the least significant

digit. Digit reversal leads to

$$k_{DIF} = \left[\sum_{i=1}^{S-1}\left(k_{S+1-i}\prod_{l=i+1}^{S}N_{S+1-l}\right)\right] + k_1\,, \tag{2.23}$$

where k_1 is the least significant digit and k is represented as $k_S k_{S-1}...k_1$. Here, k_{S+1-i} is associated with the base N_{S+1-i}. As an example, if $N = N_1N_2 = 4 \times 2$ is decomposed with two stages, $n = 6$ is represented as $3_4 0_2$, and $k = 6$ is represented as $1_2 2_4$, according to (2.22) and (2.23). Table 2.1 shows the complete n and k index mappings for radix-2 and radix-4 $N = 16$ FFTs, and Table 2.2 shows the complete index mappings for radix-2 and radix-4/2 $N = 8$ FFTs.

2.2.1.4 CTA Twiddle Multiplication

Equation (2.6) indicates that each of the N_1 DIF butterfly outputs is multiplied by a twiddle factor, given by $W_N^{n_2k_1}$ for $k_1 = 0, 1, ...N_1 - 1$ and fixed n_2. Since $W_N^{n_2k_1} = 1$ when $k_1 = 0$, each N_i butterfly, where $i \in [1, S)$, requires at most $N_i - 1$ non-trivial twiddle multiplications. Twiddle multiplications are expensive to perform in hardware. Additionally, the twiddle values, specifically tailored for the N-point FFT, must be stored in look-up tables. A naïve implementation of a *runtime-reconfigurable* DIF or DIT CTA FFT, required by many applications, would store unique twiddles for each supported N, incurring a potentially large read-only memory (ROM) area penalty.

When $N = 24$ is decomposed into $N = N_1 \times (N_2 \times N_3) = 4 \times (2 \times 3)$, the twiddles found after the first DIF stage are

$$W_N^{(0,1,...,N_1-1)\times(0,1,...,N_2N_3-1)} = W_{24}^{(0,1,...,3)\times(0,1,...,5)}\,. \tag{2.24}$$

The twiddles used in the second stage are

$$W_{N_2N_3}^{(0,1,...,N_2-1)\times(0,1,...,N_3-1)} = W_6^{(0,1)\times(0,1,2)}\,. \tag{2.25}$$

A key observation is that the set of second stage twiddles comprises a subset of the first stage twiddles due to the fact that the values can be renormalized:

$$W_6^{(0,1)\times(0,1,2)} = W_{24}^{2\times(0,1)\times2\times(0,1,2)} = W_N^{N_1\times(0,1,...,N_2-1)\times(0,1,...,N_3-1)}\,. \tag{2.26}$$

Thus, only first stage twiddles must be calculated for storage. As mentioned earlier, the first stage of a naïve 24-point FFT uses 24 twiddles. Because twiddle multiplication can be bypassed (and the butterfly output directly passed through) when $k_1 = 0$, a more intelligent design might store $\frac{3}{4} \times 24 = 18$ (not necessarily unique) twiddles while still remaining simple to programmatically control. If more complex control logic is tolerable, only 12 unique (or 11 unique, non-trivial) twiddles must be stored. Similarly, the 12 twiddles associated with $N = 4 \times 3$ are given by

$$W_N^{(0,1,...,N_1-1)\times(0,1,...,N_2-1)} = W_{12}^{(0,1,...,3)\times(0,1,2)} = W_{24}^{2\times(0,1,...,3)\times(0,1,2)}\,. \tag{2.27}$$

If reconfigurability between 12-point and 24-point FFTs is required, an intelligent design will only store the twiddles associated with $N = 24$, since the $N = 12$ twiddles are a subset of the $N = 24$ twiddles. The renormalization in (2.27) is possible because 24 is divisible by 12. In a final $N = 20 = 4 \times 5$ example, the 20 first stage twiddles are $W_{20}^{(0,1,\ldots,3)\times(0,1,\ldots,4)}$. Ignoring twiddles associated with $k_1 = 0$, only $\frac{4}{5} \times 20 = 16$ twiddles must be stored. Further storage optimization results in 9 unique (or 8 unique, non-trivial) twiddles. Since 24 is not divisible by 20, twiddle factors cannot be shared, and a reconfigurable FFT supporting $N = 12$, $N = 20$, and $N = 24$ must store, at best, 19 unique, non-trivial twiddles.

The previous examples illustrated the possibility of minimizing the number of explicit twiddle multiplications when constructing a more intelligent runtime-reconfigurable CTA FFT accelerator. Minimizing non-trivial twiddle multiplications reduces dynamic power and ROM area. However, the most efficiently implemented Cooley-Tukey algorithm is still suboptimal for handling large, mixed-radix N's individually. Additionally, it is suboptimal, both in terms of control logic complexity and in terms of twiddle storage, when one desires runtime reconfigurability among mixed-radix FFT sizes that are not fully divisible with each other.

2.2.2 Generalized Index Mapping

The CTA mappings in (2.2), (2.3), (2.12), and (2.13) are specific versions of the following more general 1D-to-2D index mapping equations [36]:

$$n = (A_1 n_1 + A_2 n_2) \bmod N \tag{2.28}$$

$$k = (B_1 k_1 + B_2 k_2) \bmod N\,. \tag{2.29}$$

Applying (2.28) results in the two-dimensional data map:

$$\begin{bmatrix} x_0 & x_1 & x_2 & \ldots & x_{N-1} \end{bmatrix} \mapsto \begin{bmatrix} x[0,0] & x[0,1] & \ldots & x[0,N_2-1] \\ x[1,0] & x[1,1] & \ldots & x[1,N_2-1] \\ \vdots & \vdots & \ddots & \vdots \\ x[N_1-1,0] & x[N_1-1,1] & \ldots & x[N_1-1,N_2-1] \end{bmatrix}. \tag{2.30}$$

Substituting (2.28) and (2.29) into (2.1) results in

$$\begin{aligned} X[k_1,k_2] &= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x[n_1,n_2] W_N^{(A_1n_1+A_2n_2) \bmod N \times (B_1k_1+B_2k_2) \bmod N} \\ &= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x[n_1,n_2] W_N^{(A_1n_1+A_2n_2)(B_1k_1+B_2k_2)} \\ &= \sum_{n_2=0}^{N_2-1}\sum_{n_1=0}^{N_1-1} x[n_1,n_2] W_N^{A_1n_1B_1k_1} W_N^{A_1n_1B_2k_2} W_N^{A_2n_2B_1k_1} W_N^{A_2n_2B_2k_2}\,. \end{aligned} \tag{2.31}$$

Because powers of the roots of unity are periodic with period N,

$$W_N^a = W_N^{a+bN}, \quad b \in \mathbb{Z}. \tag{2.32}$$

Therefore, the exponent a can be taken modulo N, i.e., $W_N^a = W_N^{a \bmod N}$. This, along with the property:

$$(ab) \bmod N = (a \bmod N \times b \bmod N) \bmod N \tag{2.33}$$

enables the simplification in (2.31).

2.2.2.1 Case A: $\gcd(N_1, N_2) \neq 1$

When $\gcd(N_1, N_2) \neq 1$, as is the case for $N = 2^n$, the (2.28) mapping is one-to-one if

$$A_1 = \alpha N_2 \text{ and } A_2 \neq \beta N_1 \text{ and } \gcd(\alpha, N_1) = \gcd(A_2, N_2) = 1 \tag{2.34}$$

or

$$A_1 \neq \alpha N_2 \text{ and } A_2 = \beta N_1 \text{ and } \gcd(A_1, N_1) = \gcd(\beta, N_2) = 1. \tag{2.35}$$

Note that $\alpha, \beta \in \mathbb{Z}$. Equation (2.29) is likewise bijective if B_1, B_2 are substituted for A_1, A_2 in (2.34) and (2.35). Setting $A_1 = N_2$, $A_2 = 1$, $B_1 = 1$, and $B_2 = N_1$ results in the DIF CTA index mappings in (2.2) and (2.3). Note that since $N_2 n_1 + n_2 < N$ and $k_1 + N_1 k_2 < N$, the modulo operation is not explicitly needed. Likewise, setting $A_1 = 1$, $A_2 = N_1$, $B_1 = N_2$, and $B_2 = 1$ results in the DIT CTA index mappings in (2.12) and (2.13).

2.2.2.2 Case B: $\gcd(N_1, N_2) = 1$

When $\gcd(N_1, N_2) = 1$, (2.28) is one-to-one if

$$A_1 = \alpha N_2 \text{ and/or } A_2 = \beta N_1 \text{ and } \gcd(A_1, N_1) = \gcd(A_2, N_2) = 1. \tag{2.36}$$

Equation (2.29) is likewise bijective if B_1, B_2 are substituted for A_1, A_2 in (2.36). As can be seen in Section 2.2.3, when $\gcd(N_1, N_2) = 1$, careful selection of A_1, A_2, B_1, and B_2 satisfying the conditions given in (2.36) eliminates the need for twiddle multiplication in (2.31).

2.2.3 Prime-Factor Algorithm

To eliminate twiddle multiplications from (2.31), one needs to set

$$W_N^{A_1 n_1 B_1 k_1} = W_{N_1}^{n_1 k_1} \text{ and } W_N^{A_2 n_2 B_2 k_2} = W_{N_2}^{n_2 k_2} \tag{2.37}$$

$$W_N^{A_1 n_1 B_2 k_2} = W_N^{A_2 n_2 B_1 k_1} = 1. \tag{2.38}$$

Condition (2.37) is true when

$$A_1 B_1 = \delta N + N_2 \text{ and } A_2 B_2 = \gamma N + N_1 \tag{2.39}$$

for $\delta, \gamma \in \mathbb{Z}$. This is equivalent to saying [37]

$$(A_1 B_1) \bmod N = N_2 \text{ and } (A_2 B_2) \bmod N = N_1 \,. \tag{2.40}$$

Additionally, condition (2.38) is true when

$$(A_1 B_2) \bmod N = 0 \text{ and } (A_2 B_1) \bmod N = 0 \,. \tag{2.41}$$

Let

$$A_1 = N_2 \tag{2.42}$$
$$A_2 = N_1 \tag{2.43}$$

and

$$B_1 = N_2 \mu = N_2 \times (N_2^{-1}) \bmod N_1 \tag{2.44}$$
$$B_2 = N_1 \rho = N_1 \times (N_1^{-1}) \bmod N_2 \,. \tag{2.45}$$

ρ is the (integer) multiplicative inverse of N_1 reduced modulo N_2. Stated another way, $1 + \kappa N_2 = N_1 \rho$, where κ is an integer. Likewise, $1 + \nu N_1 = N_2 \mu$, where μ, ν are integers. It can then be easily shown that (2.42)–(2.45) satisfy the conditions (2.40) and (2.41). Bézout's identity states that

$$\gcd(a, b) = au + bv \,, \tag{2.46}$$

where $a, b, u, v \in \mathbb{Z}$ and a, b are not both equal to 0. This implies that $\gcd(B_1, N_1) = 1$ and $\gcd(B_2, N_2) = 1$, so that the Case B condition (2.36) is met for coprime N_1, N_2. The resultant index mapping equations for n (known as the Good's mapping) and k (known as the Chinese remainder theorem (CRT) mapping) are given by [36], [38]

$$n = (N_2 n_1 + N_1 n_2) \bmod N \tag{2.47}$$
$$k = (N_2 \times \langle N_2^{-1} \rangle_{N_1} \times k_1 + N_1 \times \langle N_1^{-1} \rangle_{N_2} \times k_2) \bmod N \,. \tag{2.48}$$

Note that the shorthand notation $\langle A^{-1} \rangle_B = (A^{-1}) \bmod B$ is used to represent the multiplicative inverse of A reduced modulo B. The Chinese remainder theorem is discussed in more detail in Section 2.6.2.3.1. For $N = N_1 N_2$ DFT decompositions where N_1 and N_2 are *coprime*, substituting (2.47) and (2.48) into (2.1) thus leads to the recursive prime-factor algorithm (PFA) [19]:

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} \left\{ \sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1} \right\} W_{N_2}^{n_2 k_2} \,. \tag{2.49}$$

Equation (2.49) represents a true bi-dimensional transform without twiddle factors, such that reversing the decomposition order results in the transpose mapping, i.e., the summations in (2.49) can be performed in either order. $N = 4^{n_a} 2^{n_b} 3^m 5^k 7^l$ mixed-radix hardware

FFTs, which are supported by the generator presented in this work, can attain better hardware efficiency by recursively utilizing the PFA to eliminate twiddle multiplications. As an example, for $N = N_1N_2 = 3 \times 5$, the index mapping equations are

$$n = (5n_1 + 3n_2) \bmod N \tag{2.50}$$
$$k = (10k_1 + 6k_2) \bmod N\,, \tag{2.51}$$

resulting in the 1D-to-2D mapping illustrated in Fig. 2.5 and detailed in Table 2.3.

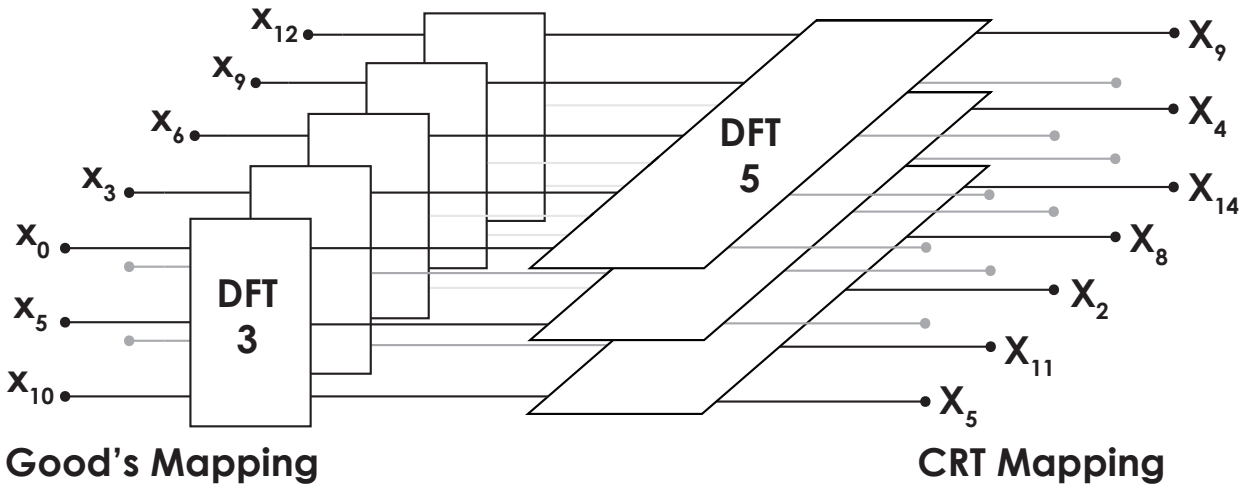

Figure 2.5: $N = 15$ 1D-to-2D PFA mapping [35].

Table 2.3: n, k index mappings for $N = 15$ FFTs.

Good's Input Mapping n					
$n_1 \backslash n_2$	0	1	2	3	4
0	x[0]	x[3]	x[6]	x[9]	x[12]
1	x[5]	x[8]	x[11]	x[14]	x[2]
2	x[10]	x[13]	x[1]	x[4]	x[7]

CRT Output Mapping k					
$k_1 \backslash k_2$	0	1	2	3	4
0	X[0]	X[6]	X[12]	X[3]	X[9]
1	X[10]	X[1]	X[7]	X[13]	X[4]
2	X[5]	X[11]	X[2]	X[8]	X[14]

A benefit of the CTA was that the same index counters could be used to map both n_{DIT} and k_{DIF} (and likewise, n_{DIF} and k_{DIT}) data, supporting in-place IO. From (2.47) and (2.48), it is evident that reversing the decomposition order does not allow the same index

counters to be used for $n_{reverse}$ and $k_{forward}$, as $n_{forward} = n_{reverse}$. Therefore, since in-place IO cannot be performed, the use of Good's mapping and the CRT mapping necessitates nominally $3N$ memory.

2.2.4 A Generalized Mixed-Radix Algorithm

The Good and CRT mappings given by (2.47) and (2.48) are the most commonly used index mappings when decomposing N into coprime N_1 and N_2 with the PFA. However, other mappings exist that satisfy the conditions in (2.36), (2.40), and (2.41), such as the index mappings proposed in [22]:

$$n = (N_2 n_1 + A_2 n_2) \bmod N \tag{2.52}$$
$$k = (B_1 k_1 + N_1 k_2) \bmod N\,, \tag{2.53}$$

where

$$A_2 = p_1 N_1 = q_1 N_2 + 1 \tag{2.54}$$
$$B_1 = p_2 N_2 = q_2 N_1 + 1\,. \tag{2.55}$$

Equations (2.54) and (2.55) indicate that $\gcd(A_1, N_1) = \gcd(N_2, N_1) = 1$, $\gcd(A_2, N_2) = 1$, $(A_1 B_1) \bmod N = (N_2 q_2 N_1 + N_2) \bmod N = N_2$, $(A_2 B_2) \bmod N = (q_1 N_2 N_1 + N_1) \bmod N = N_1$, $(A_1 B_2) \bmod N = (N_2 N_1) \bmod N = 0$, and $(A_2 B_1) \bmod N = (p_1 N_1 p_2 N_2) \bmod N = 0$. As in Section 2.2.3, (2.54) and (2.55) can be restated:

$$A_2 = N_1 \times (N_1^{-1}) \bmod N_2 \tag{2.56}$$
$$B_1 = N_2 \times (N_2^{-1}) \bmod N_1\,. \tag{2.57}$$

Thus, the (forward) generalized mixed-radix index mapping can be written as

$$n_{forward} = (N_2 n_1 + N_1 \times \langle N_1^{-1} \rangle_{N_2} \times n_2) \bmod N \tag{2.58}$$
$$k_{forward} = (N_2 \times \langle N_2^{-1} \rangle_{N_1} \times k_1 + N_1 k_2) \bmod N\,. \tag{2.59}$$

Reversing the decomposition order results in

$$n_{reverse} = (N_1 n_2 + N_2 \times \langle N_2^{-1} \rangle_{N_1} \times n_1) \bmod N \tag{2.60}$$
$$k_{reverse} = (N_1 \times \langle N_1^{-1} \rangle_{N_2} \times k_2 + N_2 k_1) \bmod N\,. \tag{2.61}$$

Because (2.58)–(2.61) indicate that the same index counters can be used to map both $n_{reverse}$ and $k_{forward}$ (and likewise, $n_{forward}$ and $k_{reverse}$), in-place IO with $2N$ memory is achievable using the index mappings from [22] and alternating decomposition orders every $2a$th cycle.

To illustrate that the recursive decomposition of N into relatively prime N_1, N_2, N_3 supports in-place IO, recall that numbers A and B are coprime when $\gcd(A, B) = 1$. Therefore,

$$A_{2,n} = p_1 N_1 \quad = Q_1(N_2N_3) + 1 = \langle N_1^{-1} \rangle_{N_2N_3} \times N_1 \tag{2.62}$$
$$A_{2,\tilde{n}_2} = p_2 N_2 \quad = Q_2 N_3 + 1 = \langle N_2^{-1} \rangle_{N_3} \times N_2 \tag{2.63}$$
$$B_{1,\tilde{k}_2} = p_3 N_3 \quad = Q_3 N_2 + 1 = \langle N_3^{-1} \rangle_{N_2} \times N_3 \tag{2.64}$$
$$B_{1,k} = p_4(N_2N_3) \quad = Q_4 N_1 + 1 = \langle (N_2N_3)^{-1} \rangle_{N_1} \times N_2N_3\,. \tag{2.65}$$

Note that n, k are associated with mapping $[0, N)$ to $[0, N_1) \times [0, \tilde{N}_2)$, where $\tilde{N}_2 = N_2N_3$, and $\tilde{n}_2, \tilde{k}_2$ are associated with mapping $\tilde{N}_2$ to $[0, N_2) \times [0, N_3)$. The forward decomposition of $N = N_1 \times (N_2 \times N_3)$ thus results in the following input/output maps:

$$n_{forward} = ((N_2N_3)n_1 + N_1 \times \langle N_1^{-1} \rangle_{N_2N_3} \times \tilde{n}_{2,forward}) \bmod N \tag{2.66}$$
$$\tilde{n}_{2,forward} = (N_3n_2 + N_2 \times \langle N_2^{-1} \rangle_{N_3} \times n_3) \bmod N_2N_3 \tag{2.67}$$
$$k_{forward} = ((N_2N_3) \times \langle (N_2N_3)^{-1} \rangle_{N_1} \times k_1 + N_1\tilde{k}_{2,forward}) \bmod N \tag{2.68}$$
$$\tilde{k}_{2,forward} = (N_3 \times \langle N_3^{-1} \rangle_{N_2} \times k_2 + N_2k_3) \bmod N_2N_3\,. \tag{2.69}$$

The reverse decomposition of $N = (N_3 \times N_2) \times N_1$ leads to

$$n_{reverse} = (N_1\tilde{n}_{2,reverse} + (N_2N_3) \times \langle (N_2N_3)^{-1} \rangle_{N_1} \times n_1) \bmod N \tag{2.70}$$
$$\tilde{n}_{2,reverse} = (N_2n_3 + N_3 \times \langle N_3^{-1} \rangle_{N_2} \times n_2) \bmod N_2N_3 \tag{2.71}$$
$$k_{reverse} = (N_1 \times \langle {N_1}^{-1} \rangle_{N_2N_3} \times \tilde{k}_{2,reverse} + (N_2N_3)k_1) \bmod N \tag{2.72}$$
$$\tilde{k}_{2,reverse} = (N_2 \times \langle N_2^{-1} \rangle_{N_3} \times k_3 + N_3k_2) \bmod N_2N_3\,. \tag{2.73}$$

2.2.5 Combining the CTA and PFA

It is possible to order the decomposition of $N = 2^n3^m5^k$ such that the PFA is first used to factorize N into coprime $N_1 = 2^n$, $N_2 = 3^m$, and $N_3 = 5^k$, and then the relatively prime N_1, N_2, N_3 can each be individually decomposed into $N_1 = N_{1,1}N_{1,2}... = 2^{n_a}2^{n_b}...$, etc. via the CTA. By performing PFA decomposition before CTA decomposition, the number of twiddle multiplications is minimized [19], and consequently, the twiddle ROM size is reduced, while still supporting in-place IO with $2N$ memory. This is because the PFA does not require twiddle multiplications, so the number of twiddles is a function of the twiddles required to decompose each N_i with the CTA. It is $\mathcal{O}(\max(N_i))$ instead of $\mathcal{O}(N)$.

When the CTA decomposition is ordered from largest to smallest radix, the twiddles required to decompose N_i are fully determined by the twiddles used in the first DIF CTA decomposition stage. As in Section 2.2.1.4, the twiddle factors of all subsequent stages can be renormalized to a subset of the first stage's twiddles. If the first DIF decomposition stage is associated with radix-$N_{i,1}$, $N_{i,1}$ should always be associated with the largest radix-r corresponding to each of 2^n, 3^m, and 5^k. As explained in Section 2.5, the number of butterfly

operations associated with unique twiddles in the first stage is $N_i/N_{i,1}$. Note once again that this is not a function of N. Additionally, as described in Section 2.2.1.4, every radix-r butterfly requires at most $r-1$ non-trivial twiddle rotations. Therefore, each N_i is associated with at most $N_i/N_{i,1} \times (N_{i,1}-1)$ twiddles whose values are $W_{N_i}^{(0,1,\ldots,N_i/N_{i,1}-1)\times(1,2,\ldots,N_{i,1}-1)}$. Without further optimization, when applying the PFA + CTA to decompose the DFT into smaller 2-point, 3-point, 4-point, and 5-point DFT butterfly operations, only

$$T = \sum_{i=1}^{3} \frac{N_i}{N_{i,1}} \times (N_{i,1}-1) = \frac{3}{4} \times 2^n + \frac{2}{3} \times 3^m + \frac{4}{5} \times 5^k \tag{2.74}$$

twiddle factors must be stored. Here, the largest butterfly radix associated with 2^n is 4. Note that (2.74) still accounts for trivial twiddle multiplications associated with $k_{i,1} \neq 0$ for $k_{i,1} \in [0, N_{i,1})$. Those multiplications can be bypassed to save power with some additional control complexity. Additionally, trivial twiddles can be easily removed for each $N_i = N_{i,1}$, since $W_{N_i}^0 = 1$.

When runtime reconfigurability is desired, due to renormalization, the total number of twiddles is governed by the maximum $N_{1,max} = 2^{n,max}$, $N_{2,max} = 3^{m,max}$, and $N_{3,max} = 5^{k,max}$ required by all supported N's:

$$T = \frac{3}{4} \times 2^{n,max} + \frac{2}{3} \times 3^{m,max} + \frac{4}{5} \times 5^{k,max}. \tag{2.75}$$

This implies that the ROM savings from PFA + CTA decompositions are even more pronounced for reconfigurable architectures. Furthermore, because the twiddle ROMs can be partitioned by the N_1, N_2, N_3 coprimes used in PFA decompositions, simpler control logic can be used to support runtime reconfigurability. As an example, if an accelerator is designed to support $N = 24 = (4 \times 2) \times 3$, $N = 12 = 4 \times 3$, and $N = 20 = 4 \times 5$, PFA + CTA decomposition requires that the 12 twiddles: $W_8^{(0,1)\times(1,2,3)}$, $W_3^{(0)\times(1,2)}$, and $W_5^{(0)\times(1,2,3,4)}$ be stored, corresponding to $N_{1,max} = 8$, $N_{2,max} = 3$, and $N_{3,max} = 5$, respectively. This is an improvement over a similarly reconfigurable FFT accelerator using only the CTA, which requires, at best, 19 unique, non-trivial twiddles, as detailed in Section 2.2.1.4. Eliminating non-trivial twiddles via control optimizations would bring the number of unique twiddles to 3 when using PFA + CTA. All three of these twiddles arise from $N_{1,max} = 8$ associated with $N = 24$. There are no twiddles associated with $N = 20$ and $N = 12$ (although 1's are stored, according to (2.75)), because the CTA is not required when radix-4 butterflies are used.

Fig. 2.6 illustrates the forward/reverse signal flow graph for a 24-point FFT using PFA + CTA that requires only $2N$ memory for in-place computation and IO.

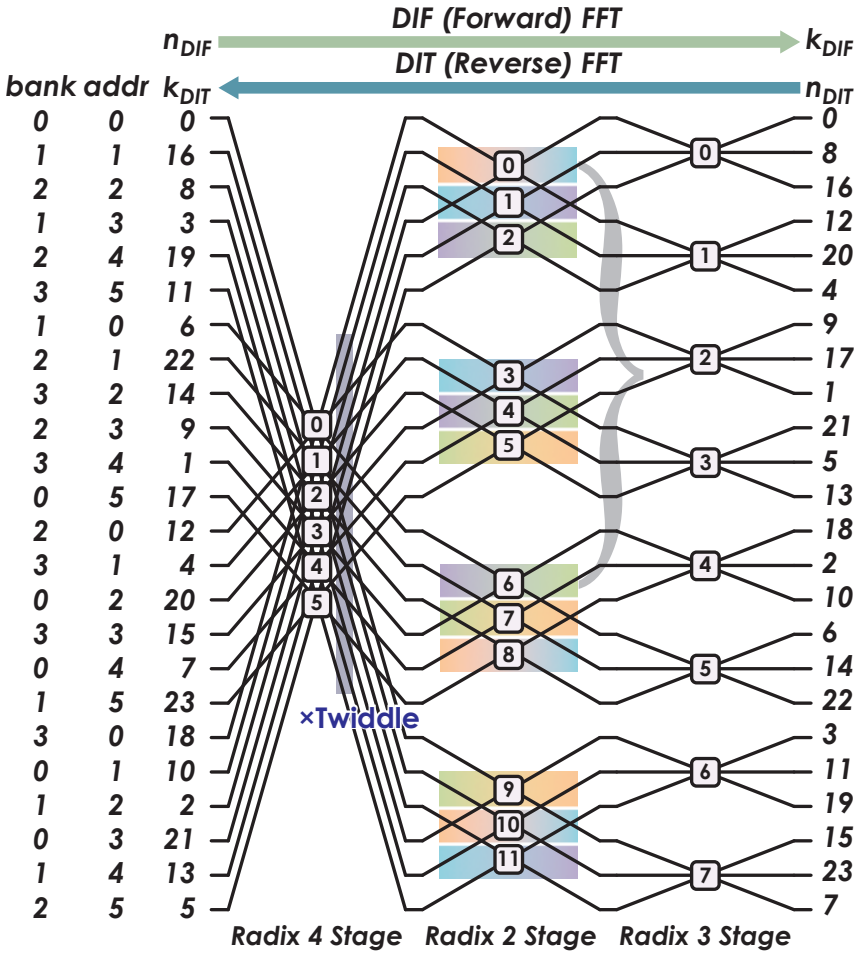

Figure 2.6: FFT $N = 24$ signal flow graph. Forward and reverse decompositions mirror each other. Calculations can be performed in-place, but n_{DIF}, k_{DIF} input/output orders are scrambled relative to each other. The ith calculation stage requires N/r_i butterfly operations, where r_i is the radix associated with stage i. Colors in the radix-2 stage represent different memory banks needed at each butterfly iteration. As an example, butterflies 0 & 6 use non-conflicting banks. This will be described further in Section 2.4.3.

2.3 Input/Output Index Vector Generator

Recall from Section 2.2.4 that an N-point forward FFT is associated with the input map

$$n = \langle N_2 n_1 + (Q_1 N_2 + 1) n_2 \rangle_N \,, \tag{2.76}$$

when N is decomposed into coprime N_1, N_2 with $\gcd(N_1, N_2) = 1$. Since N_1 and N_2 are known, the recursive extended Euclidean algorithm (Algorithm 1) can be used to solve for

Q_1 in

$$p_1 N_1 - Q_1 N_2 = 1\,, \tag{2.77}$$

which is a rearrangement of (2.54).

Algorithm 1: Extended Euclidean algorithm for $ax + by = \gcd(a, b)$.

```
function ExtendedEuclidean (a, b)  Input  : Two nonnegative integers a and b
Output: gcd(a, b), x, y
if a = 0 then
    return (b, 0, 1)
else
    (gcd, y, x) = ExtendedEuclidean(b mod a, a)
    return (gcd, x − (b/a)y, y)
end
```

To enable reconfigurability, rather than mapping data indices to memory addresses/banks via LUTs, a custom index vector generator based off of [22] is implemented. To build an index vector generator for use with an $N = N_1 N_2$-point FFT, assume there is a series of cascaded counters whose counts are $n_1' \in [0, N_1)$ and $n_2' \in [0, N_2)$. The counter sequence is incremented as follows:

$$n_2' := \begin{cases} n_2' + 1 & \text{if } n_2' \neq N_2 - 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.78}$$

$$n_1' := \begin{cases} n_1' + 1 & \text{if } n_1' \neq N_1 - 1 \\ 0 & \text{otherwise} \end{cases}, \quad \text{on } n_2' \to 0\,. \tag{2.79}$$

The $:=$ indicates that the update happens on the next clock cycle (i.e., sequential logic). n_1' is not updated until the n_2' counter wraps. The purpose of the index vector generator is to map these cascaded n_x' counter values to n_x so that sequential time-domain inputs are stored in memory locations corresponding to the correct multidimensional index mapping. In other words, it must satisfy

$$n = N_2 n_1' + n_2' = \langle N_2 n_1 + (Q_1 N_2 + 1) n_2 \rangle_N\,. \tag{2.80}$$

When attempting to satisfy this requirement, we set $n_2 = n_2'$. It can then be proven that setting

$$n_1 = \langle n_1' + Q_1' n_2 \rangle_{N_1} \tag{2.81}$$

satisfies this condition for $Q_1' = \langle N_1 - Q_1 \rangle_{N_1}$ as follows:

$$\begin{aligned} n &= \langle N_2 n_1 + (Q_1 N_2 + 1) n_2 \rangle_N && (2.82) \\ &= \langle N_2 (n_1 + Q_1 n_2) + n_2 \rangle_N \\ &= \langle \langle N_2 (n_1 + Q_1 n_2) \rangle_{N_1 N_2} + n_2 \rangle_N \\ &= \langle N_2 \langle n_1 + Q_1 n_2 \rangle_{N_1} + n_2 \rangle_N\,. && (2.83) \end{aligned}$$

In particular,

$$\begin{aligned}\langle n_1 + Q_1 n_2 \rangle_{N_1} &= \langle \langle n_1' + \langle N_1 - Q_1 \rangle_{N_1} n_2 \rangle_{N_1} + Q_1 n_2 \rangle_{N_1} && (2.84)\\ &= \langle n_1' + \langle N_1 - Q_1 \rangle_{N_1} n_2 + Q_1 n_2 \rangle_{N_1} \\ &= \langle n_1' + (Q_1 + \langle N_1 - Q_1 \rangle_{N_1}) n_2 \rangle_{N_1} \\ &= \langle n_1' + \langle \langle Q_1 + \langle N_1 - Q_1 \rangle_{N_1} \rangle_{N_1} \langle n_2 \rangle_{N_1} \rangle_{N_1} \rangle_{N_1} \\ &= \langle n_1' + \langle \langle Q_1 + N_1 - Q_1 \rangle_{N_1} \langle n_2 \rangle_{N_1} \rangle_{N_1} \rangle_{N_1} \\ &= \langle n_1' + \langle 0 \times \langle n_2 \rangle_{N_1} \rangle_{N_1} \rangle_{N_1} \\ &= n_1' \,. && (2.85)\end{aligned}$$

Therefore,

$$\begin{aligned} n &= \langle N_2 n_1' \ + n_2' \rangle_N && (2.86)\\ &= N_2 n_1' + n_2' \,. && (2.87)\end{aligned}$$

In addition to (2.33), the following properties were used to verify the correct operation of the index vector generator:

$$(a \bmod N) \bmod N = a \bmod N \tag{2.88}$$

$$(a + b) \bmod N = [(a \bmod N) + (b \bmod N)] \bmod N \tag{2.89}$$

$$(a \bmod N + b) \bmod N = (a + b) \bmod N \tag{2.90}$$

$$(Ab) \bmod (AN) = A(b \bmod N) \,. \tag{2.91}$$

When decomposing N into more coprime factors, n_x' is always incremented when n_{x+1}' wraps, and the counter associated with the largest index, denoted x_{max}, continually updates. More generally, $n_{x_{max}} = n_{x_{max}}'$, and $Q_x' = (N_x - Q_x) \bmod N_x$, where $x < x_{max}$, are calculated and stored in look-up tables for index vector generation. When N is decomposed into N_1, N_2, and N_3,

$$n_1 = \langle n_1' + Q_1' \tilde{n}_2 \rangle_{N_1} \tag{2.92}$$

$$n_2 = \langle n_2' + Q_2' n_3 \rangle_{N_2} \,. \tag{2.93}$$

The $\tilde{n}_2$ count increments and wraps when the wrap condition (W.C.) is met: $n_2' = N_2 - 1$ and $n_3' = N_3 - 1$, although it is not explicitly used. Instead, the index vector generator performs:

$$R_1 := \begin{cases} 0 & \text{if W.C.} \\ (R_1 + Q_1') \bmod N_1 & \text{otherwise} \end{cases} \tag{2.94}$$

$$R_2 := \begin{cases} 0 & \text{if } n_3' = N_3 - 1 \\ (R_2 + Q_2') \bmod N_2 & \text{otherwise} \end{cases} \tag{2.95}$$

$$n_1 = (n_1' + R_1) \bmod N_1 \tag{2.96}$$

$$n_2 = (n_2' + R_2) \bmod N_2 \,. \tag{2.97}$$

Because $R_x, Q'_x, n'_x \in [0, N_x)$, the sums are $< 2N_x - 1$, so a simple subtractor and two-input Mux can compute the modulus when base-10 representations are used. One such index vector generator can be used for both DIF and DIT configurations with N_1, N_2, N_3 reversal, but the Q'_xs must be calculated separately.

To support coprime factorization and then CTA, the vector $(n_{1,v}, n_{1,v-1}, ..., n_{1,0}, n_{2,u}, ..., n_{2,0}, n_{3,t}, ..., n_{3,0})$ can be obtained by using the (n_1, n_2, n_3) values to address corresponding decimal-to-base-r LUTs (when all digits are represented in radix-r). However, we have chosen instead to store the Q'_xs in base-r notation and implement base-r adders. The IO control logic consists of a series of cascaded base-r (or mixed-radix) counters/adders that map $[0, N_x)$, where $x \in [1, 3]$, with digits $n_{x,y}, ..., n_{x,0}$, to an index vector. The counters are built by replacing standard unsigned adders with base-r building blocks. When base-r numbers $a_2a_1a_0$ and $b_2b_1b_0$ are added, the digits of the sum and the intermediate carry outs can be computed with a simple subtraction and mux-based mod unit because $a_y + b_y + c_{in,y} < 2r_y$ (where r_y is the radix of the yth digit). The modulo operations with the decomposed N_x are obtained "for free" by simply masking out the appropriate base-r digits, and the final index vector is directly obtained without additional LUTs. As an example, if $N_{3,DIF}$ is 5^k, n_3 is decomposed into the base-5 digits $(n_{3,t}, ..., n_{3,0})$. For an $N_{1,DIT}, N_{2,DIT}, N_{3,DIT} = N_{3,DIF}, N_{2,DIF}, N_{1,DIF}$ decomposition, n_1 uses base-5, and the index vector is reversed (digit reversal is performed) so that the same hardware constants may be used for address/bank generation: $(n_{3,0}, ..., n_{3,t}, n_{2,0}, ..., n_{2,u}, n_{1,0}, ..., n_{1,v})$. A subtle but important point is that the radix order is swapped to support in-place IO, e.g., if the counters are mixed-radix 4/2 in the forward mode (DIF), they operate as mixed-radix 2/4 counters in the reverse mode (DIT), requiring programmability of the counters' radices. This index vector generation/IO unscrambling scheme is not unique to specific flavors of FFT and supports a recursive (PFA, then CTA) decomposition. Fig. 2.7 illustrates the index vector generator used.

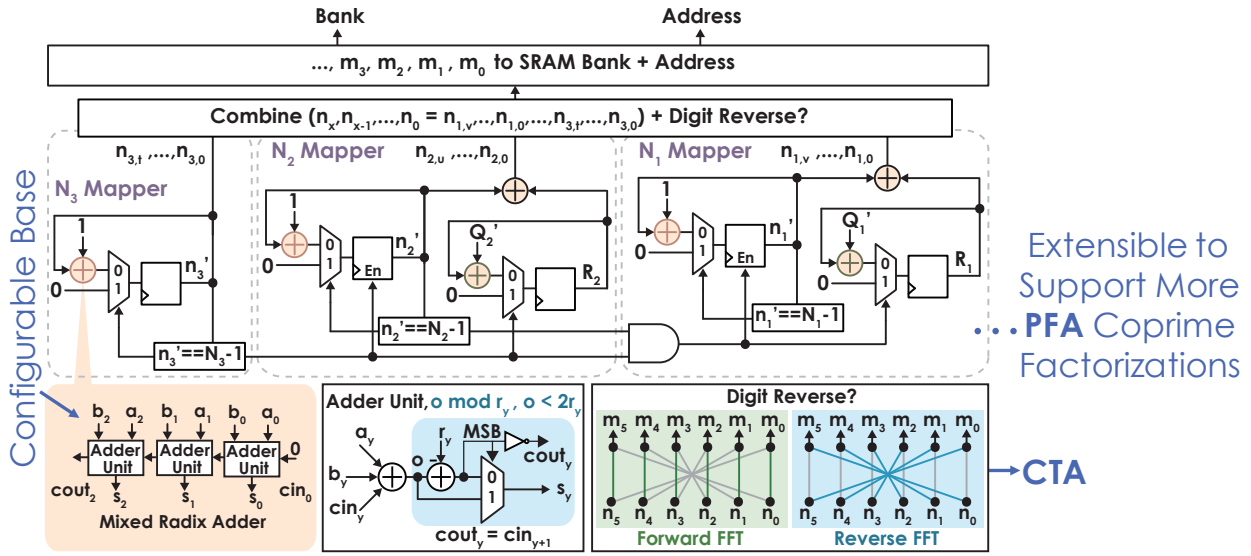

Figure 2.7: IO control logic: Index to memory bank/address mapper, consisting of n'_x and R_x mixed-radix counters and a digit reversal block for forward/reverse decompositions. Modulo operations, which only need basic digit masking, are built into adders, as required. n'_2 increments when n'_3 wraps. n'_1 increments when n'_3, n'_2 wrap. $R_{x<3}$ wraps when the corresponding n'_x increments. The mixed-radix digits of the N_x mapper outputs are combined into one index vector. Banks and addresses are obtained from the n_x values via (2.104) and (2.109).

2.4 Memory-Based Architecture with Conflict-Free Calculation Scheduling

Although details in the previous section, such as input/output unscrambling and PFA + CTA decomposition strategies, are generally architecture-agnostic, hardware to perform the actual FFT computation is not. The choice of hardware architecture and scheduling scheme is largely dependent on the performance and area requirements of a particular application, as explained below.

2.4.1 Pipelined vs. Memory-Based Architectures

Hardware FFT designs often use either in-place memory-based (Fig. 2.8) or pipelined architectures. Pipelined designs use a fixed number of processing elements for a given butterfly radix to guarantee throughput and more easily support continuous data flow [39], [17]. Often, butterflies are over-provisioned to achieve a desired throughput. Although various flavors of pipelined architectures (e.g., multi-path delay commutator) exist, one of the most frequently used is the single-path delay feedback (SDF) architecture shown in Fig. 2.9. The SDF architecture requires $\log_2(N)$ radix-2 butterflies to calculate $N = 2^n$ FFTs. Although

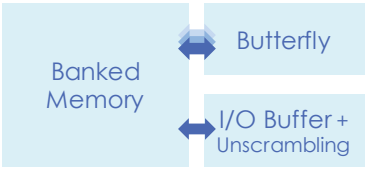

Figure 2.8: High-level diagram of a memory-based architecture.

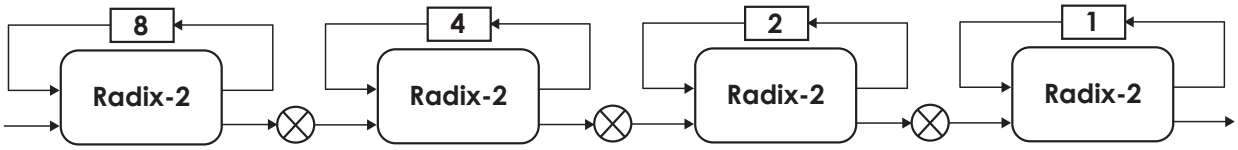

Figure 2.9: $N = 16$ radix-2 single-path delay feedback (SDF) FFT.

butterfly utilization is only at 50%, a significant benefit of SDF FFTs when compared to memory-based FFTs is the ability to sequentially access memory. Additionally, memory is localized to each PE. A drawback of pipelined FFTs is that they are less flexible, potentially leading to a wasteful design, especially when runtime-reconfigurability is desired. For example, excluding memory required for unscrambling, the reconfigurable 12–1200 point FFT engine in [17] required 1.84× the memory of a 1200-point FFT. Memory-based FFTs have fairly complex control logic when compared to their stream-based counterparts, but once the logic is designed, it is easily generalizable to support runtime reconfigurability across various mixed-radix FFT lengths with minimum additional overhead. For large FFT sizes, memory-based designs are smaller in area—usually relying on a single iterating butterfly—but support lower throughput [22]. Table 2.4 compares memory-based radix-2 FFTs with radix-2 SDF FFTs.

	Memory	**SDF**
Storage Requirements	N	$N-1$
Memory Banks	2	$\log_2(N)$
# of Complex Multipliers	1	$\log_2(N)-1$
# of Complex Adders	2	$2\log_2(N)$
# of Computation Cycles	$N\log_2(N)/2$	N

Table 2.4: Comparison of radix-2 memory-based vs. SDF FFTs.

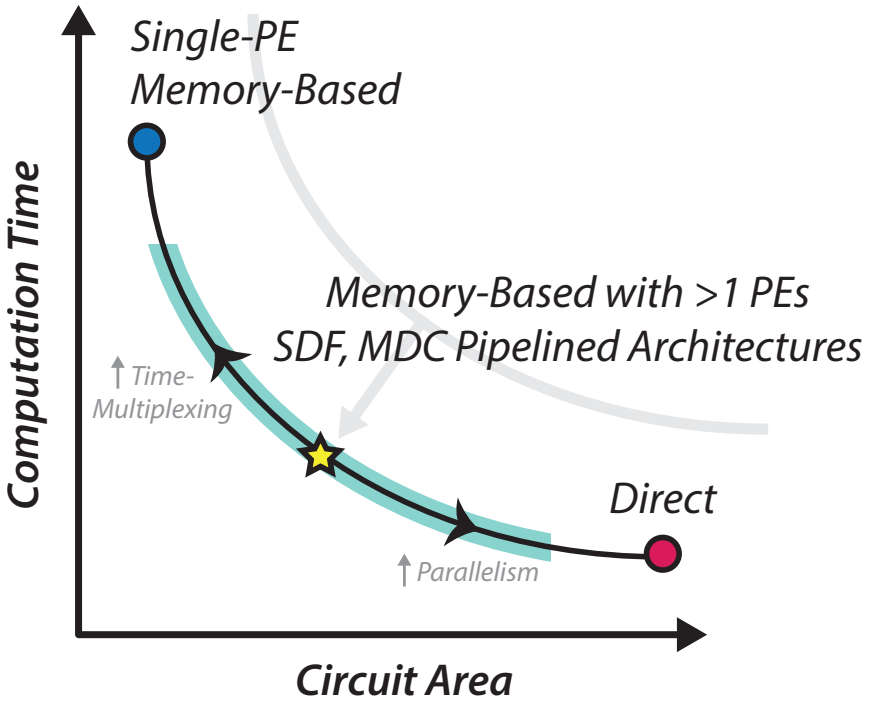

Figure 2.10: Area/throughput trade-offs for different FFT architectures. Given a throughput constraint (e.g., continuous data flow), the optimal number of parallel butterflies should be used. However, this complicates conflict-free scheduling.

2.4.2 Computation Clock Cycles

Assuming that conflict-free memory access can be scheduled, memory-based designs with simultaneously active PEs can be fine-tuned to just barely meet throughput requirements without over-provisioning PEs and wasting area. An N-point FFT can be calculated in

$$C = SP + \sum_{i=0}^{S-1} \left\lceil \frac{N}{r_i B} \right\rceil \tag{2.98}$$

computation clock cycles, where B is the number of parallel butterflies used, r_i is the radix at the $(i+1)$th calculation stage corresponding to some $N_{x,y}$, S is the number of calculation stages, and P represents the pipelining between memory accesses (butterfly pipeline and sequential memory read delay). The first term shows that all stage $(i-1)$ operations must finish before stage i calculations begin. As the pipeline is flushed out, calculations must be stalled (Fig. 2.11), so subsequent stages can compute on fresh data. Fig. 2.12 illustrates the number of clock cycles needed (normalized to the FFT length) to complete each FFT calculation with one iterating PE. As N increases, typically $> N$ clock cycles must be reserved for computation (Fig. 2.12). Unfortunately all operands ingested in parallel must

come from different memory banks, so scheduling to decrease the cycle count by supporting butterfly parallelization, especially for reconfigurable mixed-radix FFT engines, is still a topic of ongoing research.

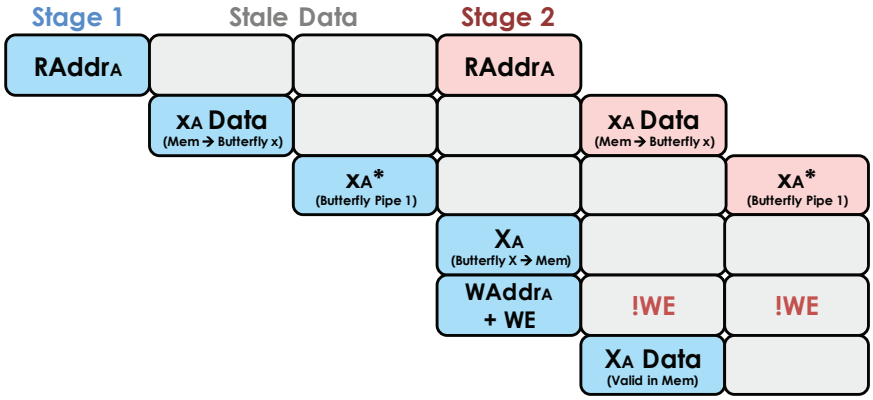

Figure 2.11: Calculation stalling assuming the butterfly and twiddle computation have a total pipeline delay of one. There is an additional one cycle latency from read address to valid data from the SRAM. Stale data should not be written to memory, necessitating that the write enable be disabled. The SRAM logic is implemented such that if a computation block requires data from an address that is currently being written to, the new data is automatically routed to the output, although it takes an extra cycle to actually write to memory.

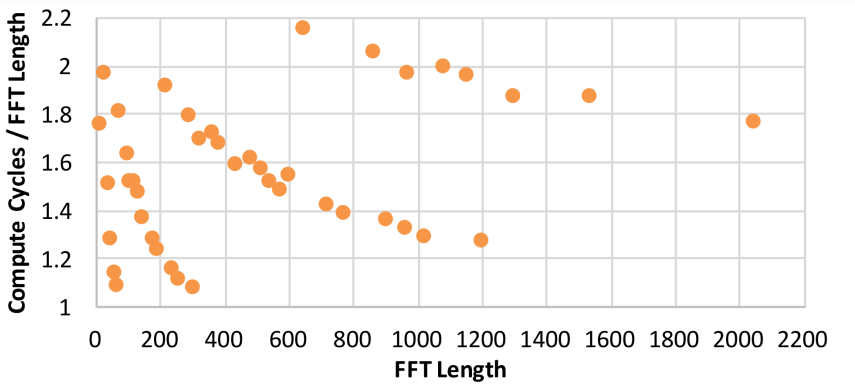

Figure 2.12: Normalized number of computation cycles needed for various LTE/Wi-Fi compatible FFT lengths using one butterfly. Most of the LTE/Wi-Fi FFTs require $> N$ computation cycles to complete. Stall cycles needed for pipelining increase the number of computation clock cycles.

Equation (2.98) shows that only two radix-2/4 butterflies are needed to complete a 2048-point FFT in < 2048 clock cycles. This is generally true for LTE/Wi-Fi compatible FFT lengths. Stated another way, most of these FFTs can be computed with a single iterating butterfly running at twice the IO clock rate to support continuous data flow, which is how the FFT instances described in this thesis are implemented. Unfortunately, Fig. 2.12 also indicates that some FFTs need $> 2N$ computation cycles when a single butterfly is used. $N = 648$ and $N = 864$ calculations, requiring > 1350 and > 1728 computation cycles respectively, cannot be completed in time to support continuous data flow. To ensure that those FFTs can meet streaming requirements with calculations occurring at twice the IO rate, 2 radix-2 butterflies are performed in parallel, reusing existing hardware in the reconfigurable butterfly described in Section 2.6.3. Only the radix-2 stage is targeted, because it requires the most butterfly computations (where the $(i+1)$th stage has N/r_i butterfly computations) and does not have twiddle multiplications, since DIF radix stages are ordered from highest to lowest radix in an N_x group (e.g., 32 is decomposed into $4 \times 4 \times 2$). The latter fact allows for scheduling modifications without adding complexity to twiddle address generation. Fig. 2.13 illustrates how doubling the number of radix-2 butterflies ensures that all LTE/Wi-Fi compatible FFTs are able to complete in time to support continuous data flow by halving the number of iterations needed in the radix-2 stage.

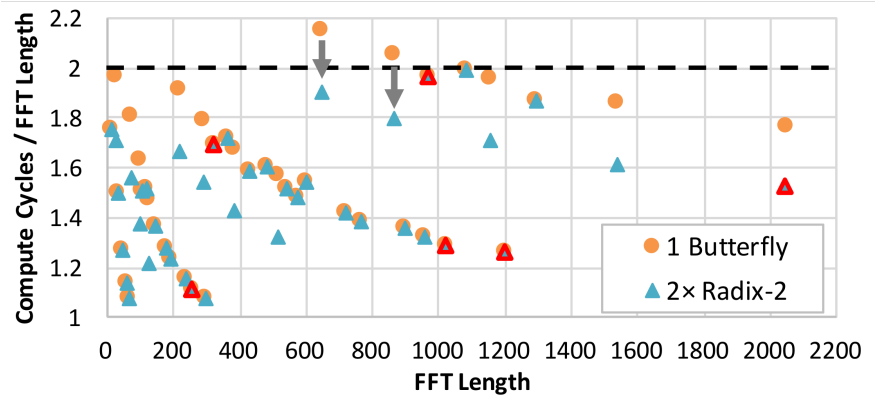

Figure 2.13: Compute cycles normalized to FFT length for 1 radix-2/3/4/5 butterfly (circles) vs. 1 radix-2x2/3/4/5 butterfly (triangles). Power measurements are performed for FFT lengths highlighted in red.

2.4.3 Butterfly Scheduling and Calculation Control Logic

2.4.3.1 Computation with a Single Iterating Butterfly

As previously mentioned, the FFT generator uses the simplest memory-based architecture: one with a single butterfly that iterates through stages of the signal flow graph at twice the IO rate. To achieve conflict-free memory access when only 1 PE is active, the calculation

SRAM is split into

$$r_{max} = \max(r_0, r_1, ...r_{S-1}) \tag{2.99}$$

banks.

In mixed-radix FFTs as in Fig. 2.6, butterflies occur in

$$\begin{cases} \prod_{x=0}^{i-1} r_x & \text{if } i > 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.100}$$

groups of

$$\begin{cases} \prod_{x=i+1}^{S-1} r_x & \text{if } i < S-1 \\ 1 & \text{otherwise} \end{cases} \tag{2.101}$$

ordered operations, where $(i+1)$ corresponds to a particular one-indexed stage. When a DIF FFT is performed, stages in Fig. 2.6 are traversed left to right. When a DIT FFT is performed, the stages are traversed from right to left. A set of cascaded m'_x counters are used to track the current butterfly iteration (in mixed-radix form) at a given stage:

$$m'_{x=S-1} := \begin{cases} m'_{S-1} + 1 & \text{if } m'_{S-1} \neq r_{S-1} - 1 \;\&\; i \neq S-1 \\ 0 & \text{otherwise} \end{cases} \tag{2.102}$$

$$m'_{x<S-1} := \begin{cases} m'_x + 1 & \text{if } m'_x \neq r_x - 1 \;\&\; x \neq i \\ 0 & \text{otherwise} \end{cases}, \quad \text{on } m'_{x+1} \to 0\,. \tag{2.103}$$

The counter i associated with the current stage $(i+1)$ is zeroed out so that associated counts (corresponding to j) can be used to refer to individual operands at each butterfly. Note that stages were identified by one-indexed counts at the beginning of this chapter, but here, they are associated with zero-indexed $i \in [0, S)$ to simplify equation indexing.

Conflict-free banking relies on the mixed-radix representations of operand locations. Because $r_i \leq r_{max}$, each butterfly operand indexed by $j < r_i$ is guaranteed to come from a different memory bank b_j [22], given by:

$$b_{i,j=0} = \sum_{x=0}^{S-1} m'_x \bmod r_{max} \tag{2.104}$$

$$b_{i,0<j<r_i} = (b_{i,0} + j) \bmod r_{max}\,. \tag{2.105}$$

Conflict-free memory access is guaranteed, because the operand indices associated with a single PE differ only in 1 mixed-radix digit j. Modulo operations are more complex than the XOR logic in [40], but they simply extend XOR (modulo-2 addition) to more general base-r's, which is useful in mixed-radix designs. Because the N data are split amongst r_{max} banks, data in each bank are mapped to addresses in $[0, N/r_{max})$. A possible addressing

scheme, where $a_{i,j}$ is the address of the jth butterfly operand of a butterfly in stage $(i+1)$, is:

$$r'_x = \begin{cases} r_x & \text{if } r_x \text{ is not the first (left-most) occurrence of } r_{max} \\ 1 & \text{otherwise} \end{cases} \tag{2.106}$$

$$A_{x=S-1} = 1 \tag{2.107}$$

$$A_{x<S-1} = A_{x+1} \times r'_{x+1} \tag{2.108}$$

$$a_{i,j=0} = \sum_{x=0}^{S-1} \begin{cases} A_x m'_x & \text{if } r_x = r'_x \\ 0 & \text{otherwise} \end{cases} \tag{2.109}$$

$$a_{i,0<j<r_i} = a_{i,j-1} + A_i \,. \tag{2.110}$$

To guarantee conflict-free memory access with parallel PEs, butterflies must be reordered. As an example, Fig. 2.6 shows that, for $N = 24$, radix-2 butterflies 0-5 can be performed simultaneously with radix-2 butterflies 6-11, respectively. Corresponding operand indices are mapped to the same SRAM addresses, but at unique banks with the banking scheme described. Adding support for two parallel radix-2 butterflies is simple in this case, because $r_i B = r_{max}$ $(2 \times 2 = 4)$. As previously described, a schedule that takes advantage of this pattern is used to reduce the cycle count when a combination of radix-2/3/4 is needed by reducing the number of computation cycles in a radix-2 stage.

2.4.3.2 Computation with Multiple Parallel Butterflies

Parallelizing PEs helps to meet throughput requirements with a lower calculation clock rate. Unfortunately, supporting several butterflies in parallel is especially difficult when one desires a general solution for runtime-reconfigurable, mixed-radix FFTs without adding considerable complexity to the I/O (as would be incurred in [21]). However, it is not necessary to change operand locations in order to prevent bank conflicts. Instead, using a modified banking strategy, butterfly operations within a radix-r stage may be reordered to minimize conflicts. To simplify scheduling logic with conflict-free memory access, N should be divisible by $r_{max}B$.

A new scheduling scheme has been devised for supporting multiple butterflies with the restriction that at least two radices amongst $r_0, r_1, ..., r_{S-1}$ must be divisible by the number of parallel butterflies. This means that only two butterflies can be computed in parallel for $N = 4 \times 2 \times 3$, and three butterflies should be parallelized for $N = 2 \times 3 \times 3$. In such cases, the number of SRAM banks required to support conflict-free memory access is $r_{max}B$, where B is the number of parallel butterflies used. The data in each bank are mapped to addresses in the space of $[0, N/(r_{max}B))$.

Assuming that B is equal to some r_i for $i \in [0, S)$, let r_o represent the first occurrence of B and r_p represent the first occurrence of r_{max} among $r_0, r_1, ..., r_{S-1}$, such that $o \neq p$. p has precedence as the index of the left-most occurrence of r_{max}. Therefore, if $r_p = r_o$, $p < o$.

The memory addressing logic is modified as follows:

$$r'_x = \begin{cases} r_x & \text{if } x \neq o \text{ and } x \neq p \\ 1 & \text{otherwise} \end{cases} \tag{2.111}$$

$$A_{x=S-1} = 1 \tag{2.112}$$

$$A_{x<S-1} = A_{x+1} \times r'_{x+1} \tag{2.113}$$

$$a_{i,j=0} = \sum_{x=0}^{S-1} \begin{cases} A_x m'_x & \text{if } r_x = r'_x \\ 0 & \text{otherwise} \end{cases} \tag{2.114}$$

$$a_{i,0<j<r_i} = a_{i,j-1} + A_i \,. \tag{2.115}$$

Here $(i+1)$ corresponds to the current stage, and $j \in [0, r_i)$ is associated with butterfly operand indices. The banking logic is modified so m'_x, where $x = p$, becomes Bm'_x:

$$b_{i,j=0} = \sum_{x=0}^{S-1} \left(\begin{cases} m'_x & \text{if } x \neq p \\ Bm'_x & \text{otherwise} \end{cases} \right) \bmod (r_{max} B) \tag{2.116}$$

$$b_{i,0<j<r_i} = \begin{cases} (b_{i,0} + j) \bmod (r_{max} B) & \text{if } i \neq p \\ (b_{i,0} + jB) \bmod (r_{max} B) & \text{otherwise} \end{cases} . \tag{2.117}$$

On stage $(i+1)$, j is substituted for $m_{x=i}$. Supporting 3 parallel butterflies with $N = 2 \times 3 \times 3$ requires that $b_{i,0} = (m'_0 + 3m'_1 + m'_2) \bmod 9$.

In order for the m'_x values to serve as mixed-radix representations of butterfly counts, they are best represented as $m_x = m'_{S-x-1}$, such that $m_{x=0}$ is the least significant digit and counts up first (from the right). These m_x values correspond to the ones in Fig. 2.7, indicating that the same address/bank generation logic can be used both for calculation and IO indexing.

So far, we have attempted to illustrate how memory addresses and banks can be calculated from m_x. However, to "re-order" the butterflies so that parallel PEs can have conflict-free memory access, the least significant mixed-radix digit is circularly shifted amongst the m_x. The exact shift amount is determined empirically for each FFT N. An evolution of this scheduling scheme is described below.

Using the banking scheme described in (2.104), calculations can be scheduled so that memory accessed per PE is conflict-free. This is shown in Fig. 2.14 for $N = 48$. It is also evident that (2.104) does not support parallel butterflies, since there are only 4 memory banks and at least 8 are needed to support two PEs without bank overlap. Changing the banking scheme to

$$b = (4m_2 + m_1 + m_0) \bmod 16 \,, \tag{2.118}$$

where $m'_{x=i}$ is replaced by the $(j < r_i)$th operand index at the $(i+1)$th stage, allows 4 processing elements to be parallelized.

However, butterflies must be grouped differently, and therefore, counting up starting with m_0 as the least significant digit does not work. Instead, the least significant digit in stage 1

Figure 2.14: $N = 48$ calculation scheduling. Banks are individually color-coded to illustrate that, on a per-PE basis, all operands come from different memory banks.

is m_1, and the least significant digit in stages 2 and 3 is m_2, owing to the fact that $m_0 = 0$ in stage 3. As shown in Fig. 2.16, by rearranging butterflies in this way, 4 butterflies in a sequence can be run simultaneously.

Although scheduling has thus far been described for butterfly parallelization, it also allows single-ported SRAMs (instead of dual-ported SRAMs) to be used in pipelined designs, reducing the amount of on-chip memory—and therefore, area—by $> 30\%$ [21]. Finally, higher radix butterflies can also be used to reduce the number of clock cycles needed.

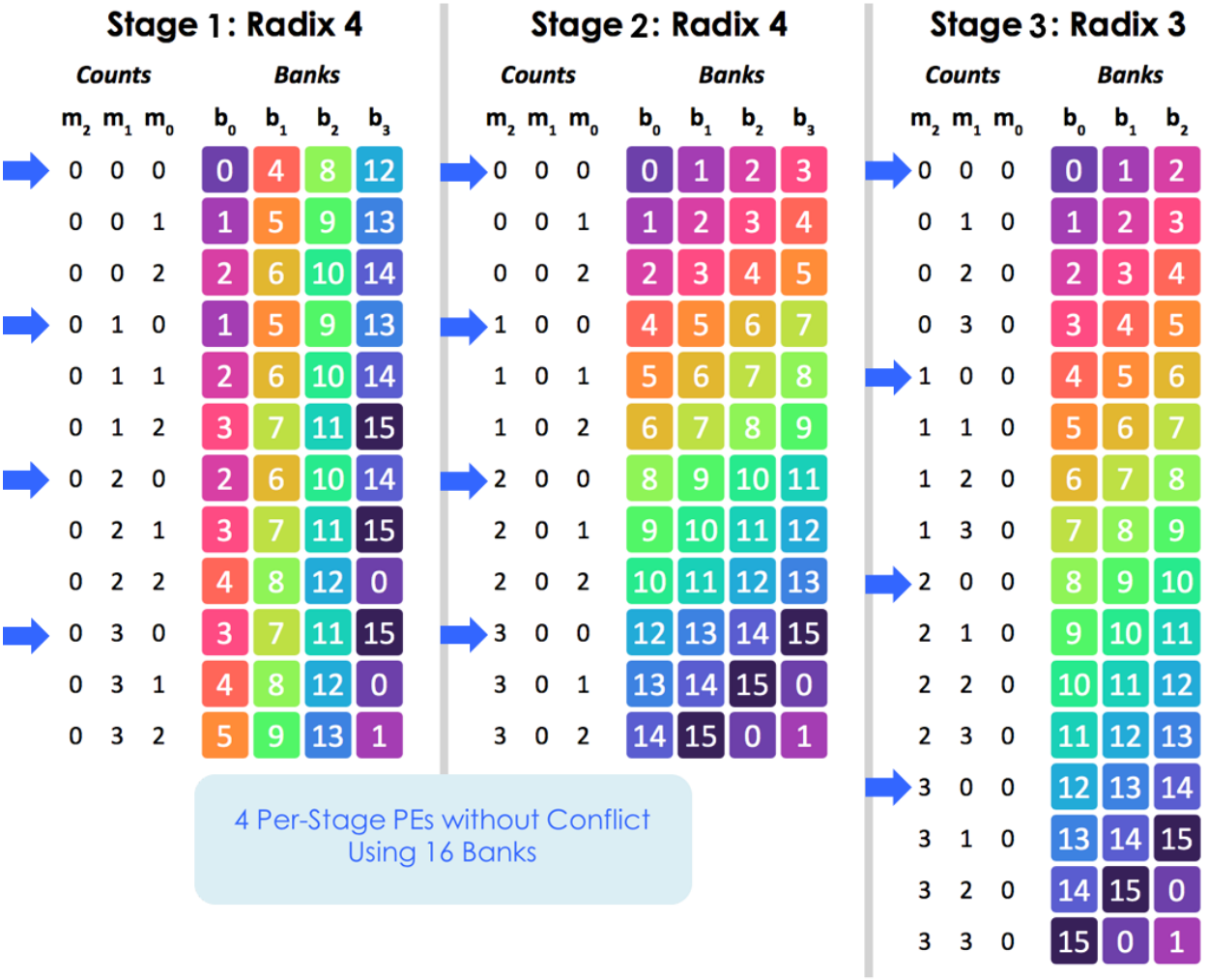

Figure 2.15: $N = 48$ calculation scheduling so that 4 butterflies can be parallelized. Arrows indicate butterflies that are associated with operands from non-conflicting banks. These butterflies are not consecutive and need to be re-ordered.

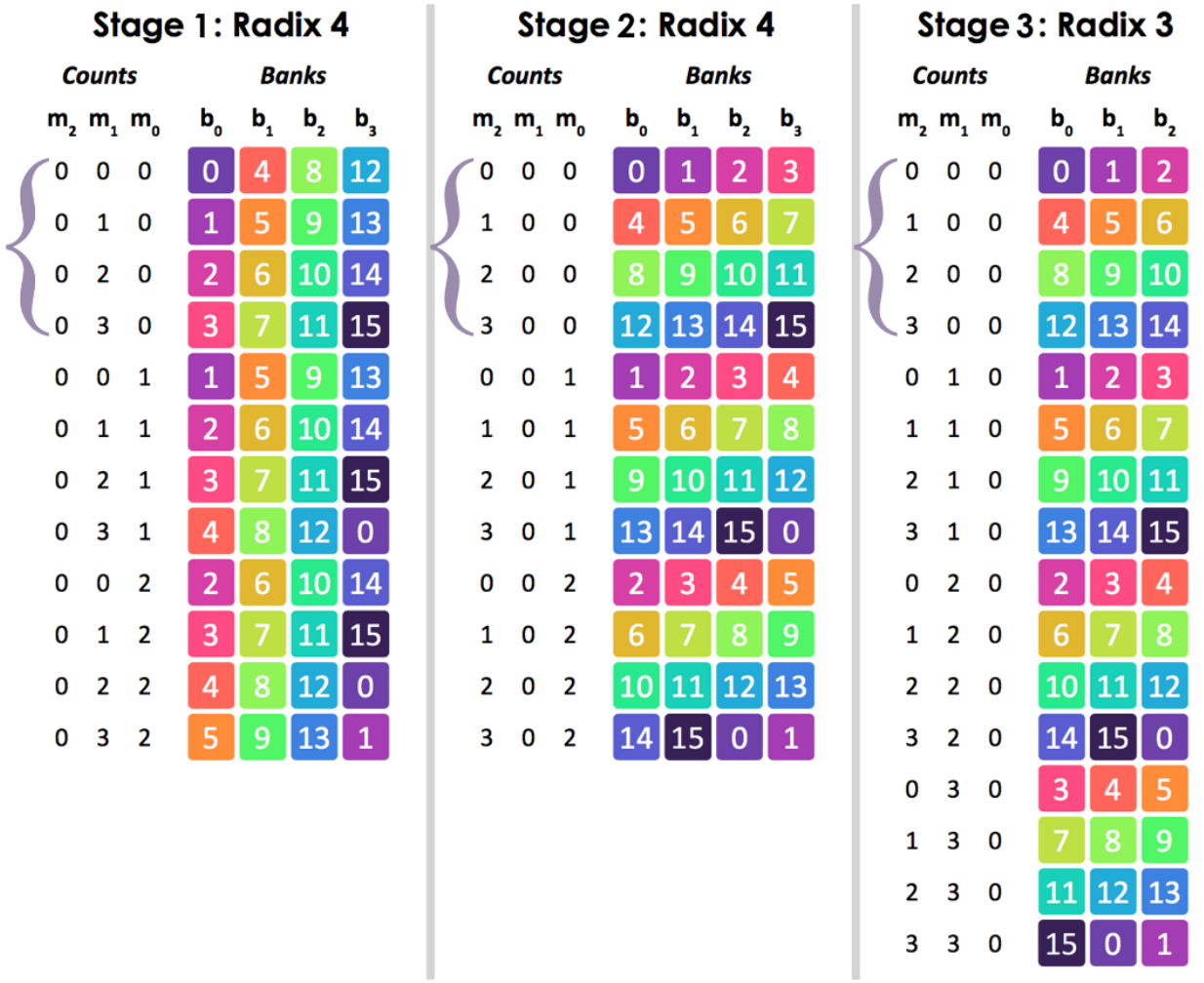

Figure 2.16: $N = 48$ calculation scheduling so that 4 butterflies can be parallelized, with butterfly re-ordering to group conflict-free sets.

2.5 Twiddle Address Generation

As elucidated in Section 2.2.1.4, the non-trivial first stage twiddles $W_N^{n_{tw,0}j}, j \in [1, r_0)$ of an N-point DIF CTA FFT can be calculated as

$$W_N^{n_{tw}j} = e^{-\frac{i2\pi n_{tw,0}j}{N}} \tag{2.119}$$

and stored into ROM look-up tables. Note that here, i represents the imaginary number and j represents the jth operand index of a butterfly. Since one ROM is associated with each $j \in [1, r_{a,max})$, where $r_{a,max}$ is described in the next paragraph, and subsequent stage twiddles can be derived from the twiddles in the first stage, the ROMs themselves can be addressed with n_{tw} as follows:

$$n_{tw,i=0} \in \left[0, \frac{N}{r_0}\right) \tag{2.120}$$

$$n_{tw,0<i<S-1} \in \left(\prod_{x=0}^{i-1} r_x\right) \times \left[0, \frac{N}{\prod_{x=0}^{i} r_x}\right) . \tag{2.121}$$

The first CTA DIF stage in Fig. 2.1 requires the most twiddle factors. Because the DFT is decomposed into smaller sizes, each subsequent stage uses fewer unique twiddles—in particular, butterfly groups in subsequent stages share the same twiddles— and the last stage requires none. The twiddle index is renormalized back to the full ROM range, e.g., $W_4^1 = W_{16}^4$.

Define $N_{a,max}$ to be the largest coprime associated with radix r_a, such that $r_{a,max}$ is the largest radix r_a that can be divided into any N_a divisible by r_a. Since the range of twiddle addresses for a given r_a is set by $N_{a,max}/r_{a,max}$, all twiddle values associated with $N_1 = 2^n \leq 2^{n,max}$, $N_2 = 3^m \leq 3^{m,max}$, and $N_3 = 5^k \leq 5^{k,max}$ are supported by only 9 ROMs with address renormalization (i.e., 3 for N_1's, 2 for N_2's, and 4 for N_3's). Using such ROMs, when iterating through stages associated with r_a's for an FFT that is decomposed with N_a, the n_{tw} previously calculated must be further renormalized by the amount $N_{a,max}/N_a$.

To extend this addressing scheme to combined PFA/CTA decompositions, the twiddle address $n_{tw,i}$ of a radix-r_i stage (where the subscript again is associated with the $(i+1)$th calculation stage rather than coprime designation, as was the case with r_a) is held for z_i butterflies prior to changing. If r_i's corresponding coprime is $N_a, a \in [1, D]$, where D represents the number of coprimes in the decomposition (e.g., $D = 3$ for $N = N_1N_2N_3$), then

$$z_i = \begin{cases} \prod_{x=a+1}^{D} N_x & \text{if } a < D \\ 1 & \text{otherwise} \end{cases} . \tag{2.122}$$

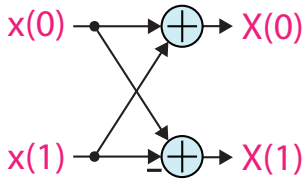

Figure 2.17: Radix-2 butterfly.

2.6 Butterfly Construction

2.6.1 Using the Cooley-Tukey Algorithm

In Section 2.2.1.1, it was shown that (2.7) could be used to derive the radix-2 butterfly equations:

$$X_0 = x_0 + x_1 \tag{2.123}$$
$$X_1 = x_0 - x_1\,. \tag{2.124}$$

These equations correspond to the signal-flow graph in Fig. 2.17. More generally, (2.7) can be rewritten as [41], [17]

$$X[k_1 + N_1 k'] = DFT_{\frac{N}{N_1}} \left\{ \left\{ \sum_{i=0}^{N_1-1} x_i W_{N_1}^{ik_1} \right\} W_N^{n'k_1} \right\}, \tag{2.125}$$

where $x_i = x\left[\frac{N}{N_1} i + n'\right]$. Equation (2.125) can be used to derive SFGs for any radix-r butterfly. When $N_1 = 3$, (2.125) becomes

$$X[k_1 + 3k'] = DFT_{\frac{N}{3}} \left\{ \left[x_0 + x_1 W_3^{k_1} + x_2 W_3^{2k_1} \right] W_N^{n'k_1} \right\}. \tag{2.126}$$

Table 2.5: Radix-3 SFG equations. 6 additions and 2 constant multiplications (one implemented as a shift operation) are required.

Stage 1	Stage 2	Stage 3	Stage 4
$a_1 = x_1 + x_2$	$m_1 = c_{21} a_1$	$X_0 = x_0 + a_1$	$X_1 = c_1 + m_2$
$a_2 = x_1 - x_2$	$m_2 = j c_{22} a_2$	$c_1 = x_0 - m_1$	$X_2 = c_1 - m_2$

This results in

Table 2.6: Radix-3 constants.

c_{21}	$1/2$
c_{22}	$-\sin(2\pi/3)$

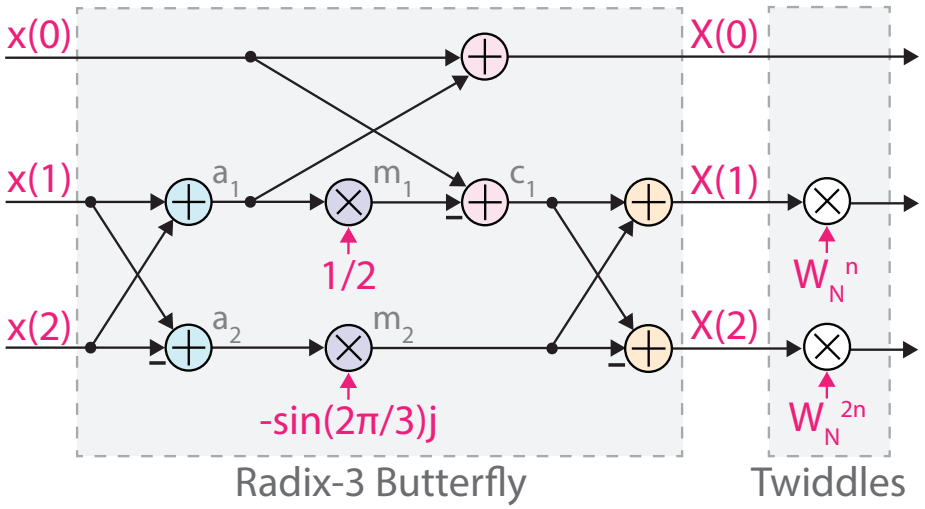

Figure 2.18: Radix-3 butterfly with two non-trivial, complex twiddle multiplications at the output (DIF).

$$X[3k'] = DFT_{\frac{N}{3}}\{[x_0 + x_1 + x_2]\} \tag{2.127}$$

$$\begin{aligned}
X[3k'+1] &= DFT_{\frac{N}{3}}\left\{\left[x_0 + x_1 W_3^1 + x_2 W_3^2\right] W_N^{n'}\right\} \\
&= DFT_{\frac{N}{3}}\left\{\left[x_0 + x_1 e^{-j2\pi/3} + x_2 e^{-j4\pi/3}\right] W_N^{n'}\right\} \\
&= DFT_{\frac{N}{3}}\left\{\left[x_0 + x_1\left(-\frac{1}{2} - j\sin\left(\frac{2\pi}{3}\right)\right) + x_2\left(-\frac{1}{2} + j\sin\left(\frac{2\pi}{3}\right)\right)\right] W_N^{n'}\right\} \\
&= DFT_{\frac{N}{3}}\left\{\left[x_0 - \frac{1}{2}(x_1 + x_2) - j\sin\left(\frac{2\pi}{3}\right)(x_1 - x_2)\right] W_N^{n'}\right\}
\end{aligned} \tag{2.128}$$

$$\begin{aligned}
X[3k'+2] &= DFT_{\frac{N}{3}}\left\{\left[x_0 + x_1 W_3^2 + x_2 W_3^1\right] W_N^{2n'}\right\} \\
&= DFT_{\frac{N}{3}}\left\{\left[x_0 + x_1\left(-\frac{1}{2} + j\sin\left(\frac{2\pi}{3}\right)\right) + x_2\left(-\frac{1}{2} - j\sin\left(\frac{2\pi}{3}\right)\right)\right] W_N^{2n'}\right\} \\
&= DFT_{\frac{N}{3}}\left\{\left[x_0 - \frac{1}{2}(x_1 + x_2) + j\sin\left(\frac{2\pi}{3}\right)(x_1 - x_2)\right] W_N^{2n'}\right\}.
\end{aligned} \tag{2.129}$$

Radix-3 butterfly operations are derived from (2.127)–(2.129) and found in the $[\cdot]$'s. Equation (2.32) implies that $W_3^4 = W_3^1$. The resulting radix-3 SFG, along with the two non-trivial

twiddle multiplications (at the butterfly output for a DIF FFT), is shown in Fig. 2.18. The equations at intermediate nodes are given by Table 2.5. Equations (2.186) and (2.187) illustrate that multiplying two complex numbers, as is the case with twiddle multiplication, requires three real multiplications. However, the Stage 2 multiplications in Table 2.5 involve a complex number and either a *constant real or imaginary* number. In such cases, only two real multiplications are necessary. When a complex number $a + jb$ is multiplied by the imaginary number jy,

$$(a + jb) \times jy = -by + jay\,. \tag{2.130}$$

It is also useful to note that, for fixed-point numbers, multiplication by 1/2 corresponds to a simple arithmetic right shift by one. The SFG in Fig. 2.18 requires the minimum amount of real multiplications for a radix-3 butterfly, but manual optimization is required to achieve this. Manual optimization becomes less tractable when large butterflies are desired.

2.6.2 Winograd's (Short) Fourier Transform Algorithm

When implementing mixed-radix DFT hardware, Winograd's short Fourier transform algorithm (WFTA) [42], which exploits fast cyclic convolution, is commonly used to reduce the number of expensive hardware multipliers in favor of a greater number of inexpensive adder units per butterfly. WFTA butterflies achieve the minimum theoretical number of multiplications for a given prime radix-r [43] and can be systematically derived by combining the use of Rader's DFT algorithm [44], which formulates the DFT as a circular convolution problem, and Winograd's short convolution algorithm.

2.6.2.1 Primitive Roots of N

When N is prime, there exists a primitive root modulo N called g such that $g^a \bmod N$ "generates" all elements of a in the field $\mathbb{Z}_N$ except 0, i.e., $g^a \bmod N \in [1, N)$ [45]. An interesting property of g is that the sequence $(g^a \bmod N)_{a=0}^{\infty}$ always repeats with a period of $N - 1$ after a certain value of a. Therefore,

$$g^0 \bmod N = g^{N-1} \bmod N\,. \tag{2.131}$$

Consider $N = 5$. A primitive root $g = 2$ exists because

$$\begin{aligned}
a = 0\,, &\quad 2^0 \bmod 5 = 1 && (2.132)\\
a = 1\,, &\quad 2^1 \bmod 5 = 2 && (2.133)\\
a = 2\,, &\quad 2^2 \bmod 5 = 4 && (2.134)\\
a = 3\,, &\quad 2^3 \bmod 5 = 3 && (2.135)\\
a = 4\,, &\quad 2^4 \bmod 5 = 1 && (2.136)\\
a = 5\,, &\quad 2^5 \bmod 5 = 2 && (2.137)\\
a = 6\,, &\quad 2^6 \bmod 5 = 4 && (2.138)\\
a = 7\,, &\quad 2^7 \bmod 5 = 3 && (2.139)\\
a = 8\,, &\quad 2^8 \bmod 5 = 1\,. && (2.140)\\
\vdots &\qquad\qquad \vdots
\end{aligned}$$

Here, the period of repetition is 4, and the sequence $a = (0, 1, 2, ...)$ is mapped to the repeated sequence $(1, 2, 4, 3, 1, 2, ...)$. Multiple g's can exist for a particular N. Additionally, numbers that are not prime but have the form 2, 4, p^b, or $2p^b$, where $b \geq 1$ and p is an odd prime, also have primitive roots [46], [45]. Table 2.7 lists the g's associated with the first few N for which primitive roots exist.

Table 2.7: Primitive roots of N [45].

N	$g(N)$
2	1
3	2
4	3
5	2, 3
6	5
7	3, 5
9	2, 5
10	3, 7
11	2, 6, 7, 8
13	2, 6, 7, 11

2.6.2.2 Rader's DFT Algorithm

To formulate the DFT as a circular convolution problem [47], (2.1) is rewritten as follows:

$$X[k] = x[0] + \sum_{n=1}^{N-1} x[n] W_N^{nk}\,. \tag{2.141}$$

Assuming N has primitive roots, performing the index maps $k \to g^k \bmod N$ and $n \to g^n \bmod N$ and utilizing (2.32) and (2.33) results in

$$\begin{aligned} X[g^k \bmod N] &= x[0] + \sum_{n=1}^{N-1} x[g^n \bmod N] W_N^{(g^n \bmod N)\times(g^k \bmod N)} \\ &= x[0] + \sum_{n=1}^{N-1} x[g^n \bmod N] W_N^{g^{n+k} \bmod N} . \end{aligned} \tag{2.142}$$

Because of the $(N-1)$ periodicity of the mapping, the sequence $(1, 2, ..., N-1)$ can be uniquely mapped to another sequence containing the same elements (but in different order). This means that (2.141) and (2.142) are equivalent for $X[1], X[2], ..., X[N-1]$. The DC term $X[0]$ is unaccounted for, but can be easily computed via

$$X[0] = \sum_{n=0}^{N-1} x[n] . \tag{2.143}$$

Rearranging the terms in (2.142) leads to

$$X[g^k \bmod N] - x[0] = \sum_{n=1}^{N-1} x[g^n \bmod N] W_N^{g^{n+k} \bmod N} . \tag{2.144}$$

Using (2.32), this can be expressed in matrix form $\vec{y} = \boldsymbol{W}\vec{x}$ as

$$\begin{bmatrix} X[\langle g^1 \rangle_N] \\ X[\langle g^2 \rangle_N] \\ X[\langle g^3 \rangle_N] \\ \vdots \\ X[\langle g^{N-1} \rangle_N] \end{bmatrix} - \begin{bmatrix} x[0] \\ x[0] \\ x[0] \\ \vdots \\ x[0] \end{bmatrix} = \begin{bmatrix} W_N^{g^2} & W_N^{g^3} & \dots & W_N^{g^{N-1}} & W_N^{g^N} \\ W_N^{g^3} & W_N^{g^4} & \dots & W_N^{g^N} & W_N^{g^{N+1}} \\ W_N^{g^4} & W_N^{g^5} & \dots & W_N^{g^{N+1}} & W_N^{g^{N+2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ W_N^{g^N} & W_N^{g^{N+1}} & \dots & W_N^{g^{2N-3}} & W_N^{g^{2N-2}} \end{bmatrix} \begin{bmatrix} x[\langle g^1 \rangle_N] \\ x[\langle g^2 \rangle_N] \\ x[\langle g^3 \rangle_N] \\ \vdots \\ x[\langle g^{N-1} \rangle_N] \end{bmatrix} . \tag{2.145}$$

For clarity, using (2.131), (2.145) is rewritten as

$$\begin{bmatrix} X[\langle g^1 \rangle_N] \\ X[\langle g^2 \rangle_N] \\ X[\langle g^3 \rangle_N] \\ \vdots \\ X[\langle g^{N-1} \rangle_N] \end{bmatrix} - \begin{bmatrix} x[0] \\ x[0] \\ x[0] \\ \vdots \\ x[0] \end{bmatrix} = \begin{bmatrix} W_N^{g^2} & W_N^{g^3} & \dots & W_N^{g^0} & W_N^{g^1} \\ W_N^{g^3} & W_N^{g^4} & \dots & W_N^{g^1} & W_N^{g^2} \\ W_N^{g^4} & W_N^{g^5} & \dots & W_N^{g^2} & W_N^{g^3} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ W_N^{g^1} & W_N^{g^2} & \dots & W_N^{g^{N-2}} & W_N^{g^0} \end{bmatrix} \begin{bmatrix} x[\langle g^1 \rangle_N] \\ x[\langle g^2 \rangle_N] \\ x[\langle g^3 \rangle_N] \\ \vdots \\ x[\langle g^{N-1} \rangle_N] \end{bmatrix} . \tag{2.146}$$

Because the sequence $(g^a \bmod N)_{a=0}^{\infty}$ is cyclic, $\boldsymbol{W}$ is an $(N-1) \times (N-1)$ *left* circulant matrix. Each row of $\boldsymbol{W}$ is the previous row circularly shifted left by one element, wrapping

at row edges. This implies that such a circulant matrix can be fully specified by the vector corresponding to the first row of $\boldsymbol{W}$. More importantly, if the sequence represented by $\vec{y}$ is reversed, the result is $\vec{y_{new}} = \boldsymbol{W_{new}}\vec{x}$, or, expanded out,

$$\begin{bmatrix} X[\langle g^{N-1}\rangle_N] \\ \vdots \\ X[\langle g^3\rangle_N] \\ X[\langle g^2\rangle_N] \\ X[\langle g^1\rangle_N] \end{bmatrix} - \begin{bmatrix} x[0] \\ x[0] \\ x[0] \\ \vdots \\ x[0] \end{bmatrix} = \begin{bmatrix} W_N^{g^1} & W_N^{g^2} & \dots & W_N^{g^{N-2}} & W_N^{g^0} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ W_N^{g^4} & W_N^{g^5} & \dots & W_N^{g^2} & W_N^{g^3} \\ W_N^{g^3} & W_N^{g^4} & \dots & W_N^{g^1} & W_N^{g^2} \\ W_N^{g^2} & W_N^{g^3} & \dots & W_N^{g^0} & W_N^{g^1} \end{bmatrix} \begin{bmatrix} x[\langle g^1\rangle_N] \\ x[\langle g^2\rangle_N] \\ x[\langle g^3\rangle_N] \\ \vdots \\ x[\langle g^{N-1}\rangle_N] \end{bmatrix} . \tag{2.147}$$

It is now evident that $\boldsymbol{W_{new}}$ is a *right* circulant matrix, and $\boldsymbol{W_{new}}\vec{x}$ computes the $(N-1)$-point discrete circular convolution of the sequence $(x[\langle g^n\rangle_N])_{n=1}^{N-1}$ with $(W_N^{g^1}, W_N^{g^2}, ..., W_N^{g^{N-2}}, W_N^{g^0})$. Thus, if primitive roots of N exist, the N-point DFT can be reformulated as a circular convolution problem.

2.6.2.3 Winograd's Short Convolution Algorithm

The matrix multiplication involving the left circulant matrix in (2.146) has the form

$$\begin{bmatrix} t_m \\ t_{m-1} \\ \vdots \\ t_0 \end{bmatrix} = \begin{bmatrix} a_m & a_{m-1} & \dots & a_1 & a_0 \\ a_{m-1} & a_{m-2} & \dots & a_0 & a_m \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_0 & a_m & \dots & a_2 & a_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} . \tag{2.148}$$

Let the associated polynomials be

$$A_m(x) = \sum_{i=0}^{m} a_i x^i \tag{2.149}$$

$$B_m(x) = \sum_{i=0}^{m} b_i x^i \tag{2.150}$$

$$T_m(x) = \sum_{i=0}^{m} t_i x^i \,, \tag{2.151}$$

where it can be proven that [48]

$$T_m(x) = \langle A_m(x) B_m(x) \rangle_{x^n-1} \,. \tag{2.152}$$

Here, $n = m+1$ is the order of the matrix. $x^n - 1$ can be factorized into k distinct irreducible polynomials $m_i(x)$ such that

$$x^n - 1 = m(x) = \prod_{i=1}^{k} m_i(x) \,. \tag{2.153}$$

Factorizations for $n = 2, 4, 6$ are given in Table 2.8. Winograd showed that only $2n - k$ multiplications are required to compute $T_m(x)$ (not counting multiplications by rational numbers) [42], [48]. This is achievable by leveraging properties derived from the Chinese remainder theorem.

Table 2.8: Factorizations of $x^n - 1$ into irreducible polynomials [48].

n	$x^n - 1$ **Factorization**	k	$2n - k$
2	$(x-1)(x+1)$	2	2
4	$(x-1)(x+1)(x^2+1)$	3	5
6	$(x-1)(x+1)(x^2+x+1)(x^2-x+1)$	4	8

2.6.2.3.1 Chinese Remainder Theorem

Define $m = m_1 m_2 ... m_k$, where $m_i \in \mathbb{Z}$ and $\gcd(m_i, m_j) = 1$ for $i \neq j$. That is, all m_i's are pairwise coprime with each other. Any $u_i \in \mathbb{Z}_{m_i}$ is uniquely mapped to one $x \in \mathbb{Z}_m$, determined by the relation [49]:

$$u_i = x \bmod m_i \,. \tag{2.154}$$

The map

$$x \bmod m \mapsto (x \bmod m_1, x \bmod m_2, ..., x \bmod m_k) \tag{2.155}$$

is injective. Using Bézout's identity (2.46), i.e. $q_1 m_1 + q_2 m_2 = 1$ for some $q_1, q_2 \in \mathbb{Z}$, a solution to the system of equations:

$$u_1 = x \bmod m_1 \tag{2.156}$$
$$u_2 = x \bmod m_2 \tag{2.157}$$

is

$$x = u_1 m_2 q_2 + u_2 m_1 q_1 \tag{2.158}$$
$$= \langle x \rangle_{m_1} m_2 q_2 + \langle x \rangle_{m_2} m_1 q_1 \,. \tag{2.159}$$

More generally, for $x \in [0, m)$, it can be proven that

$$x = x \bmod m = \sum_{i=1}^{k} \langle x \rangle_{m_i} \frac{m}{m_i} Q_i \,, \tag{2.160}$$

where $q_i m_i + Q_i \frac{m}{m_i} = 1$, satisfies (2.154) for $i = 1, 2, ..., k$.

The Chinese remainder theorem can also be extended to polynomials as well. Define $m(x) = m_1(x)m_2(x)...m_k(x)$, where $m_i(x)$ are pairwise coprime polynomials and $d_i = \deg m_i(x)$. Then the polynomial $u_i(x)$, where $\deg u_i(x) < d_i$, is uniquely mapped to a polynomial $p(x)$ with $\deg p(x) < d_1 + d_2 + ... + d_k$ satisfying

$$u_i(x) = p(x) \bmod m_i(x)\,. \tag{2.161}$$

The solution of the simultaneous congruence system (2.161) for $i = 1, 2, ..., k$ is given by

$$p(x) = p(x) \bmod m(x) = \sum_{i=1}^{k} \langle p(x) \rangle_{m_i(x)} \frac{m(x)}{m_i(x)} Q_i(x)\,. \tag{2.162}$$

This highlights the fact that $\langle p(x) \rangle_{m(x)}$ can be found when (2.161) are known, and therefore, $T_m(x)$ can be calculated after $u_i(x) = \langle A_m(x)B_m(x) \rangle_{m_i(x)}$ are determined. Garner devised a convenient algorithm for computing $p(x)$ given (2.161) [50]. It is restated here:

$$\langle c_{ij}(x)m_i(x) \rangle_{m_j(x)} = 1\,, \quad 1 \le i < j \le k \tag{2.163}$$

$$\begin{aligned}
v_1(x) &= u_1(x) && (2.164)\\
v_2(x) &= [(u_2(x) - v_1(x))c_{12}(x)] \bmod m_2(x) && (2.165)\\
v_3(x) &= [((u_3(x) - v_1(x))c_{13}(x) - v_2(x))c_{23}(x)] \bmod m_3(x) && (2.166)\\
&\vdots \\
v_k(x) &= [(...(((u_k(x) - v_1(x))c_{1k}(x) - v_2(x))c_{2k}(x) - v_3(x))c_{3k}(x) - \\
&\qquad ... - v_{k-1}(x))c_{(k-1)k}(x)] \bmod m_k(x) && (2.167)
\end{aligned}$$

$$p(x) \bmod m(x) = p(x) = v_1(x) + \sum_{i=1}^{k-1} \left(v_{i+1}(x) \left(\prod_{j=1}^{i} m_j(x) \right) \right)\,. \tag{2.168}$$

2.6.2.3.2 2-Point Convolution Example

A 2-point discrete convolution can be evaluated using a matrix of order $n = 2$:

$$\begin{bmatrix} t_1 \\ t_0 \end{bmatrix} = \begin{bmatrix} a_1 & a_0 \\ a_0 & a_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}\,. \tag{2.169}$$

Here, t_0, t_1 are the coefficients of the polynomial $T_1(x) = \langle A_1(x)B_1(x) \rangle_{x^2-1}$, and

$$p(x) = A_1(x)B_1(x) = (a_0 + a_1x)(b_0 + b_1x)\,. \tag{2.170}$$

Because $x^2 - 1$ can be factorized into $(x-1)(x+1)$, $T_1(x)$ can be obtained from

$$u_1(x) = \langle A_1(x)B_1(x)\rangle_{x-1} \tag{2.171}$$
$$u_2(x) = \langle A_1(x)B_1(x)\rangle_{x+1}\,. \tag{2.172}$$

When $m_i(x) = x - x_i$ where $x_i = \pm 1$, $u_i(x) = r_i$, which has degree zero. In this case, if $p(x)$ has degree n, $p(x) = (x - x_i)s_{n-1}(x) + r_i$. $s_{n-1}(x)$ is a polynomial with degree $n-1$. It is evident that $p(x_i) = r_i$. Therefore,

$$u_1 = p(x_1 = 1) \;\; = (a_0 + a_1)(b_0 + b_1) \tag{2.173}$$
$$u_2 = p(x_2 = -1) = (a_0 - a_1)(b_0 - b_1)\,. \tag{2.174}$$

Since $m_j(x)$ is of degree 1, $\langle c_{ij}(x)m_i(x)\rangle_{m_j(x)} = c_{ij}(x_j)m_i(x_j) = 1$. Therefore, $c_{ij}(x)$ is chosen so that

$$c_{ij}(x) = c_{ij} = \frac{1}{m_i(x_j)}\,. \tag{2.175}$$

This leads to

$$c_{12}(x) = \frac{1}{m_1(x_2)} = \frac{1}{m_1(-1)} = -\frac{1}{2}\,. \tag{2.176}$$

The remaining polynomials are calculated as follows:

$$v_1 = u_1 \tag{2.177}$$
$$\begin{aligned} v_2(x) &= [(u_2 - u_1)(-\frac{1}{2})] \bmod (x+1) \\ &= \frac{1}{2}(u_1 - u_2) \end{aligned} \tag{2.178}$$
$$\begin{aligned} T_1(x) &= v_1 + v_2 m_1(x) \\ &= u_1 + \frac{1}{2}(u_1 - u_2)(x-1) \\ &= \frac{1}{2}[(u_1 - u_2)x + (u_1 + u_2)]\,. \end{aligned} \tag{2.179}$$

The procedure demonstrated above can be used to derive algorithms requiring a minimal number of multiplications for calculating other short-length circular convolutions. The resultant algorithms can be found in [48] and [42].

2.6.2.4 3-Point WFTA Butterfly Derivation and N-Point WFTA Equations

To derive the equations for an $N = 3$ WFTA butterfly, the primitive root $g = 2$ is used. Therefore (2.146) is

$$\begin{bmatrix} X[2] \\ X[1] \end{bmatrix} - \begin{bmatrix} x[0] \\ x[0] \end{bmatrix} = \begin{bmatrix} W_3^1 & W_3^2 \\ W_3^2 & W_3^1 \end{bmatrix} \begin{bmatrix} x[2] \\ x[1] \end{bmatrix}\,. \tag{2.180}$$

The butterfly equations are derived from (2.143) and (2.179) as follows:

$$\begin{aligned} u_1 &= (W_3^2 + W_3^1)(x[2] + x[1]) \\ &= -(x[1] + x[2]) \end{aligned} \tag{2.181}$$

$$\begin{aligned} u_2 &= (W_3^2 - W_3^1)(x[2] - x[1]) \\ &= -j2\sin\left(\frac{2\pi}{3}\right)(x[1] - x[2]) \end{aligned} \tag{2.182}$$

$$X[0] = x[0] + (x[1] + x[2]) \tag{2.183}$$

$$\begin{aligned} X[1] &= x[0] + \frac{1}{2}(u_1 + u_2) \\ &= (x[0] - \frac{1}{2}(x[1] + x[2])) - j\sin\left(\frac{2\pi}{3}\right)(x[1] - x[2]) \end{aligned} \tag{2.184}$$

$$\begin{aligned} X[2] &= x[0] + \frac{1}{2}(u_1 - u_2) \\ &= (x[0] - \frac{1}{2}(x[1] + x[2])) + j\sin\left(\frac{2\pi}{3}\right)(x[1] - x[2])\,. \end{aligned} \tag{2.185}$$

These are consistent with the equations in Table 2.5 and indicate that a 3-point WFTA butterfly requires 2 multiplications (one of which is a simple shift right) and 6 additions. Equations for other short-length butterflies can be derived in a similar fashion [48], [42]. Radix-5 and radix-7 SFG equations are given in Table 2.9 and Table 2.12. Although $N = 4$ is not prime, it also has an associated WFTA butterfly consisting entirely of additions and a simple multiplication by the imaginary number j. Its SFG equations and the corresponding diagram are given in Table 2.11 and Fig. 2.19, respectively.

Table 2.9: Radix-5 SFG equations [51], [52]. 17 additions and 5 constant multiplications (one implemented as a shift operation) are required.

Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
$a_1 = x_1 + x_4$	$a_5 = a_2 + a_4$	$m_1 = jc_{51}a_5$	$c_1 = x_0 - m_5$	$X_1 = c_2 - c_4$
$a_2 = x_1 - x_4$	$a_6 = a_1 - a_3$	$m_2 = jc_{52}a_2$	$c_2 = c_1 + m_4$	$X_2 = c_3 - c_5$
$a_3 = x_2 + x_3$	$a_7 = a_1 + a_3$	$m_3 = jc_{53}a_4$	$c_3 = c_1 - m_4$	$X_3 = c_3 + c_5$
$a_4 = x_2 - x_3$		$m_4 = c_{54}a_6$	$c_4 = m_1 - m_3$	$X_4 = c_2 + c_4$
		$m_5 = c_{55}a_7$	$c_5 = m_2 - m_1$	
			$X_0 = x_0 + a_7$	

Table 2.10: Radix-5 constants, where $u = 2\pi/5$ [20].

c_{51}	$\sin(u)$
c_{52}	$\sin(u) + \sin(2u)$
c_{53}	$\sin(u) - \sin(2u)$
c_{54}	$\frac{\cos(u) - \cos(2u)}{2}$
c_{55}	$1/4$

Table 2.11: Radix-4 SFG equations [51]. 8 additions and 1 simple constant multiplication are required.

Stage 1	**Stage 2**	**Stage 3**
$a_1 = x_0 + x_2$	$m_1 = ja_4$	$X_0 = a_1 + a_3$
$a_2 = x_0 - x_2$		$X_1 = a_2 - m_1$
$a_3 = x_1 + x_3$		$X_2 = a_1 - a_3$
$a_4 = x_1 - x_3$		$X_3 = a_2 + m_1$

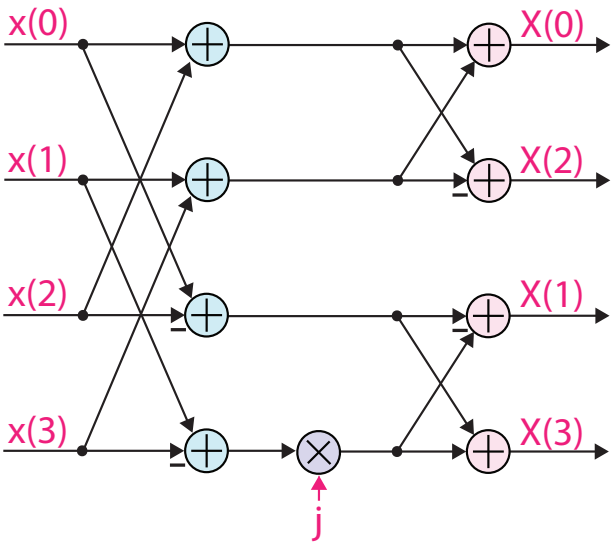

Figure 2.19: Radix-4 butterfly.

Table 2.12: Radix-7 SFG equations [51]. 36 additions and 8 constant multiplications are required.

Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
$a_1 = x_1 + x_6$	$a_7 = a_1 + a_3 + a_5$	$m_1 = c_{71}a_7$	$c_1 = x_0 - m_1$	$X_1 = c_2 - c_5$
$a_2 = x_1 - x_6$	$a_8 = a_1 - a_5$	$m_2 = c_{72}a_8$	$c_2 = c_1 + m_2 + m_3$	$X_2 = c_3 - c_6$
$a_3 = x_2 + x_5$	$a_9 = -a_3 + a_5$	$m_3 = c_{73}a_9$	$c_3 = c_1 - m_2 - m_4$	$X_3 = c_4 - c_7$
$a_4 = x_2 - x_5$	$a_{10} = -a_1 + a_3$	$m_4 = c_{74}a_{10}$	$c_4 = c_1 - m_3 + m_4$	$X_4 = c_4 + c_7$
$a_5 = x_3 + x_4$	$a_{11} = a_2 + a_4 - a_6$	$m_5 = jc_{75}a_{11}$	$c_5 = m_5 + m_6 - m_7$	$X_5 = c_3 + c_6$
$a_6 = x_3 - x_4$	$a_{12} = a_2 + a_6$	$m_6 = jc_{76}a_{12}$	$c_6 = m_5 - m_6 - m_8$	$X_6 = c_2 + c_5$
	$a_{13} = -a_4 - a_6$	$m_7 = jc_{77}a_{13}$	$c_7 = -m_5 - m_7 - m_8$	
	$a_{14} = -a_2 + a_4$	$m_8 = jc_{78}a_{14}$	$X_0 = x_0 + a_7$	

Table 2.13: Radix-7 constants, where $u = 2\pi/7$ [20].

c_{71}	$-\frac{\cos(u) + \cos(2u) + \cos(3u)}{3}$
c_{72}	$\frac{2\cos(u) - \cos(2u) - \cos(3u)}{3}$
c_{73}	$\frac{\cos(u) - 2\cos(2u) + \cos(3u)}{3}$
c_{74}	$\frac{\cos(u) + \cos(2u) - 2\cos(3u)}{3}$
c_{75}	$\frac{\sin(u) + \sin(2u) - \sin(3u)}{3}$
c_{76}	$\frac{2\sin(u) - \sin(2u) + \sin(3u)}{3}$
c_{77}	$\frac{-\sin(u) + 2\sin(2u) + \sin(3u)}{3}$
c_{78}	$\frac{\sin(u) + \sin(2u) + 2\sin(3u)}{3}$

The WFTA butterfly is constructed so that operation stages are either associated with addition/subtraction or multiplication. Additionally, the progression of operations always follows an 1) add/subtract, 2) constant multiply, 3) add/subtract pattern, and the multiplication stage always consists only of multiplications with either real or imaginary constants, which require less hardware than complex multiplications. In the next section, a runtime reconfigurable butterfly with the same general WFTA structure is detailed. Using [20] as a starting point, some butterfly operations are moved around to more easily create the reconfigurable butterfly. Fig. 2.20 indicates how this is done for a radix-3 butterfly. In particular, the output of the pink node is $c_1 = -(1 + c_{21})a_1 + X_0 = x_0 - m_1$, as in the original radix-3 equations.

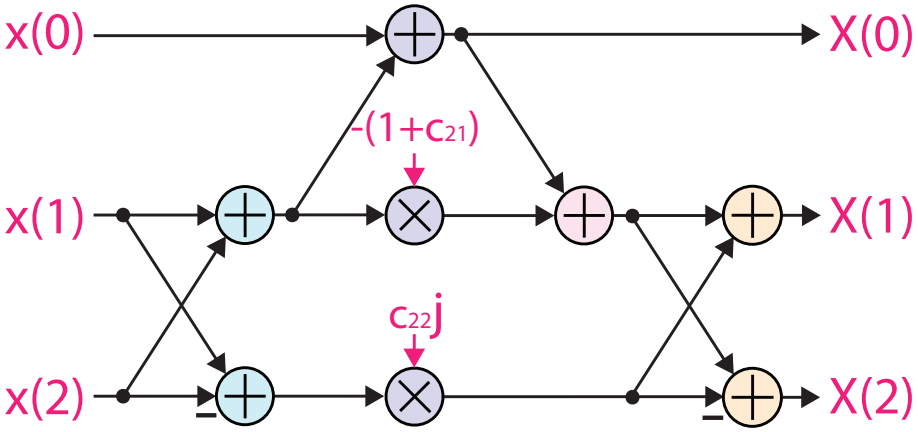

Figure 2.20: Radix-3 butterfly with rearranged operations for easier integration.

2.6.3 Reconfigurable WFTA Butterfly

Because butterflies up to radix-7 are needed to support various OFDM radios—in particular, Chinese-DTV— and because radix-7 butterflies use the most adders and multipliers amongst all radices ≤ 7, a runtime reconfigurable butterfly has been built that reuses radix-7 hardware in other radix modes. The butterfly, seen in Fig. 2.21, supports up to 36 complex additions and 8 multiplications. The multiplications involve a complex number and either a real or imaginary number that is programmable depending on the runtime radix. Tables 2.14, 2.15, and 2.16 outline parameters that are used to configure the butterfly at runtime.

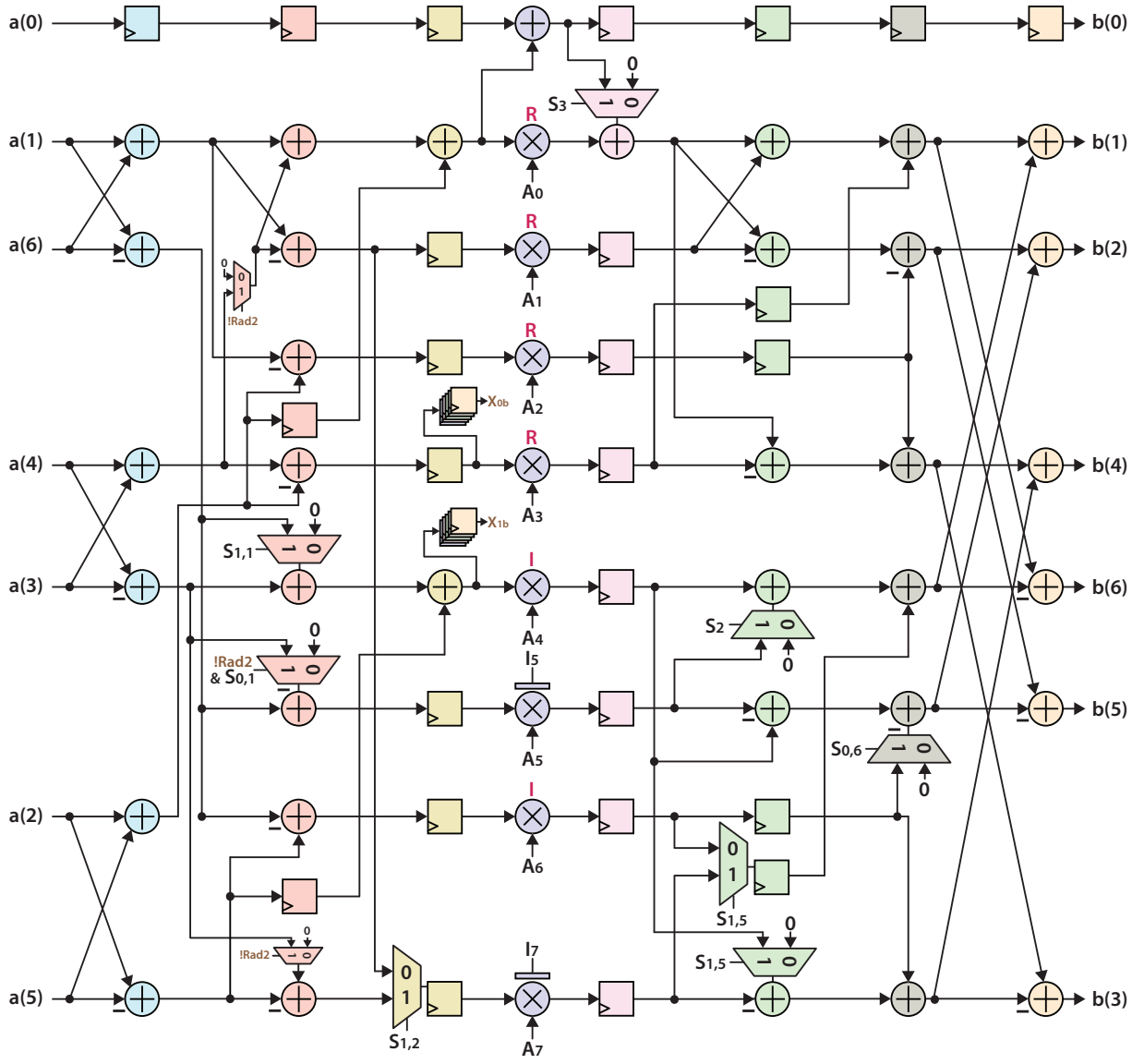

Figure 2.21: Reconfigurable radix-2×2/3/4/5/7 butterfly with operator reuse [20], [28]. Control signals and outputs in brown are for supporting 2 radix-2 butterflies simultaneously. Each colored stage can be programmatically pipelined. Two internal multipliers are reconfigured for multiplication by real or imaginary constants depending on the current radix. The other multipliers interpret associated constants as real (R) or imaginary (I), without reconfigurability.

Table 2.14: Butterfly input/output mappings, i.e., $x_i \rightarrow a_j$ and $b_k \rightarrow X_l$. * are required for simultaneous operation of two radix-2 butterflies.

Butterfly Input Mappings							
Radix	a_0	a_1	a_2	a_3	a_4	a_5	a_6
2	0	x_0	0	x_3*	x_2*	0	x_1
3	x_0	x_1	0	0	0	0	x_2
4	0	x_0	0	x_3	x_1	0	x_2
5	x_0	x_1	0	x_3	x_2	0	x_4
7	x_0	x_1	x_2	x_3	x_4	x_5	x_6
Butterfly Output Mappings							
Radix	X_0	X_1	X_2	X_3	X_4	X_5	X_6
2	b_0	b_5	X_{0b}*	X_{1b}*	0	0	0
3	b_0	b_1	b_6	0	0	0	0
4	b_0	b_1	b_3	b_5	0	0	0
5	b_0	b_1	b_5	b_2	b_6	0	0
7	b_0	b_1	b_2	b_3	b_4	b_5	b_6

Table 2.15: Mux selects used by the reconfigurable butterfly. I_i control signals determine whether multiplication is by an imaginary (1) or real (0) constant. * are required for two radix-2 butterflies.

Radix	$S_{1,1}$	$S_{0,1}$	$S_{1,2}$	S_3	S_2	$S_{1,5}$	$S_{0,6}$	I_5	I_7
2	0*	0	1	0	0	1	0	0	1
3	1	0	1	1	0	0	0	1	1
4	0	0	0	0	1	0	0	0	0
5	1	0	1	1	0	1	0	1	1
7	1	1	1	1	1	1	1	1	1

Table 2.16: Reconfigurable WFTA multiplication constants. Multiplication by 0, 1, or -1 can be bypassed or replaced with simpler operations.

Radix	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
2	0	0	0	0	0	1	0	0
3	$-(1+c_{21})$	0	0	0	c_{22}	0	0	0
4	0	0	0	0	-1	1	0	1
5	$-(1+c_{55})$	c_{54}	0	0	$-c_{51}$	$-c_{52}$	0	c_{53}
7	$-(1+c_{71})$	c_{72}	c_{74}	c_{73}	$-c_{75}$	$-c_{76}$	$-c_{78}$	c_{77}

Fig. 2.22 illustrates how the butterfly is reconfigured at runtime for radix-3 operation. External control logic passes a one-hot vector to the butterfly unit, indicating that radix-3 operation (and not radix-2/4/5/7) is to be performed. This is used to generate mux select signals that enable relevant data to pass through muxes while others are zeroed out. Likewise, multiplication constants associated with radix-3 equations are selected and fed into active multipliers. In this case, 6 adders and 2 multipliers, highlighted in purple, are active.

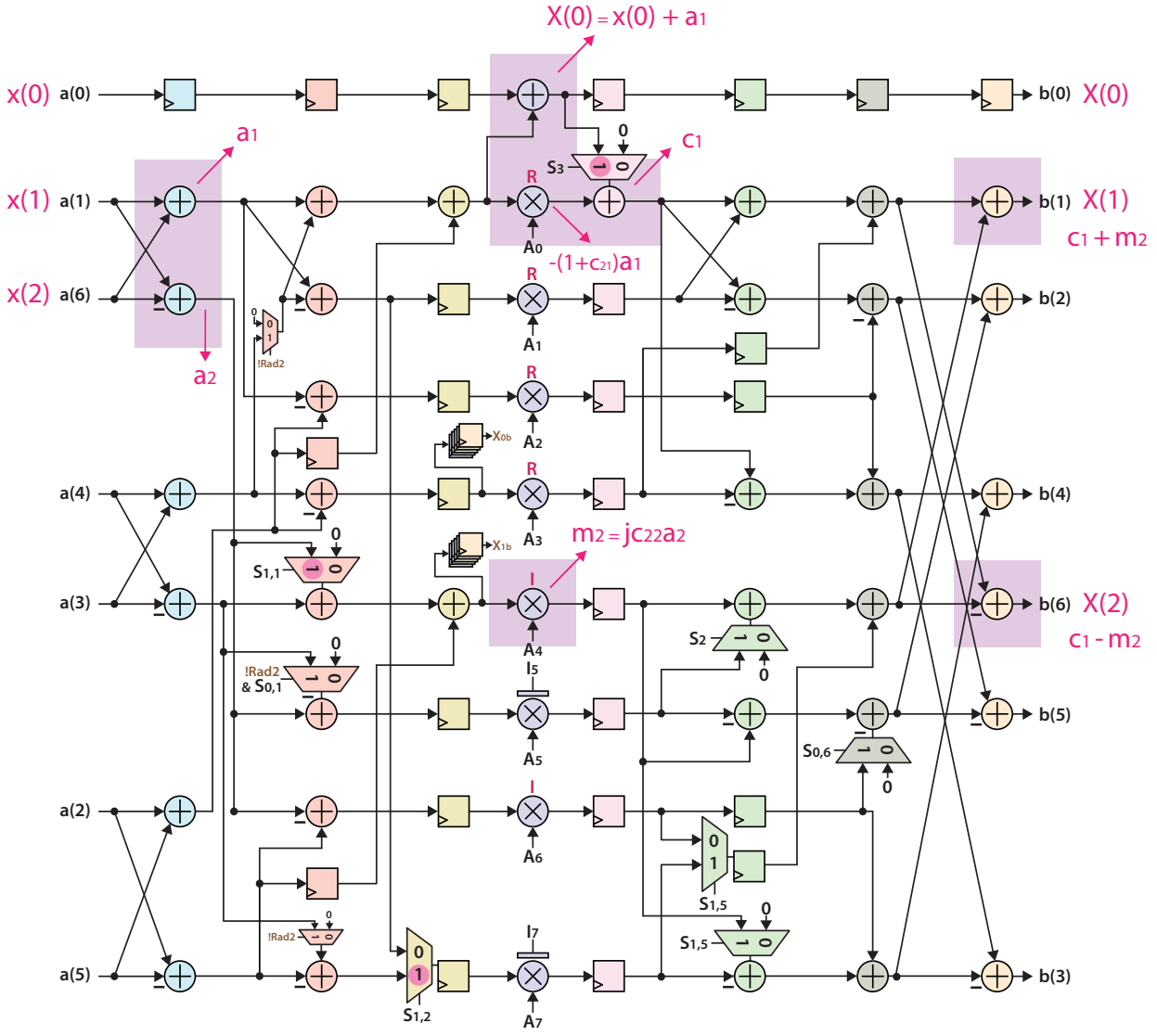

Figure 2.22: Butterfly configured (at runtime) for radix-3 operation.

However, the butterfly is not only runtime reconfigurable but also compile-time configurable. It has been designed to support:

- *Custom stage pipelining*, so that the correct number of registers can be inserted when targeting the design for FPGA or ASIC implementation,
- *Tunable internal bit growth* for SQNR optimizations, and
- *Flexible parameterization*, to enable optimal stage bypassing (bypassing adders/multipliers) when radices are not needed by the generated instance.

If a designer desires only to support radix-3 FFTs, a butterfly containing only radix-3 logic will be generated, eliminating unnecessary multiplications and additions used by higher order radices and saving area. The relative areas of different butterfly configurations are shown in Fig. 2.23. An interesting observation is that the area of a runtime-reconfigurable radix-2/3/4/5/7 butterfly is comparable to the sum of the areas of the individual butterflies. This is due to the fact that constant multiplication is significantly cheaper than multiplication where both inputs are unknown and suggests that additional butterfly optimizations are needed to make runtime reconfigurability worthwhile (e.g., [53]).

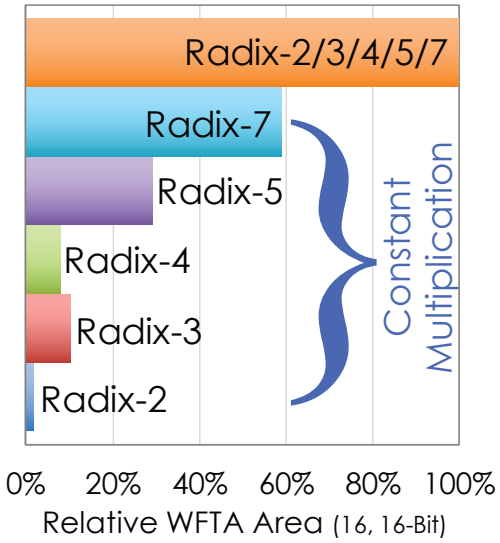

Figure 2.23: Relative area of generated butterflies using a 28nm process. Synthesis results were obtained from Design Compiler. Constant multiplication requires fewer resources. Supporting reconfigurability across radix-2/3/4/5/7 requires 70% more area than that of a static radix-7 butterfly.

2.6.4 The Complete Processing Element

The data path consists of both the runtime-reconfigurable butterfly and twiddle multiplication, although the ordering of the two depends on whether a DIF or DIT FFT is being performed.

Each twiddle multiplication performs a complex multiplication. Complex multiplication is usually performed with 4 real multipliers. However, in actuality, it can be performed with only 3 multiplications. Consider $x = a + ib$ and $y = c + id$. The product $z = xy$ can be written as

$$\Re(z) = ac - bd \tag{2.186}$$
$$\Im(z) = (a + b)(c + d) - ac - bd\,. \tag{2.187}$$

The twiddle multiplications can be optionally performed in this way.

As Fig. 2.24 shows, the processing element (butterfly + twiddle multipliers) supports DIF/DIT reconfiguration using one set of twiddle multipliers with at least one stage of pipelining between memory accesses.

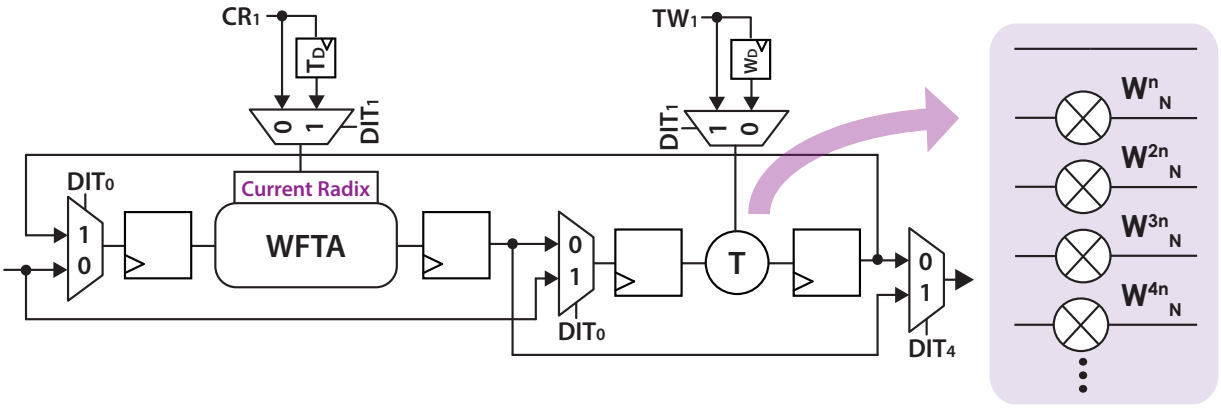

Figure 2.24: Reconfigurable processing element for DIF/DIT consisting of a runtime-reconfigurable butterfly and twiddle multipliers. Pipeline delays across the data paths need to be matched in both the DIF and DIT cases.

2.7 Generating FFT Instances from a Hardware Template

A hardware generator that generalizes the control logic and datapath described in previous sections has been built using the Chisel hardware construction language. The generator has two components. Given a set of user constraints (i.e. desired FFT N's), the firmware component (Fig. 2.25) uses Scala and the Breeze numerical processing library [54] to calculate parameters and create lists of constants for look-up table (LUT) generation. The FFT sizes are factorized to support PFA + CTA, determine the number of calculation stages needed, optimize a reconfigurable WFTA butterfly, and generate twiddle factors and index vector generator constants. As indicated in Fig. 2.25, the firmware also determines an appropriate banking scheme, which is a function of the maximum radix supported in a given

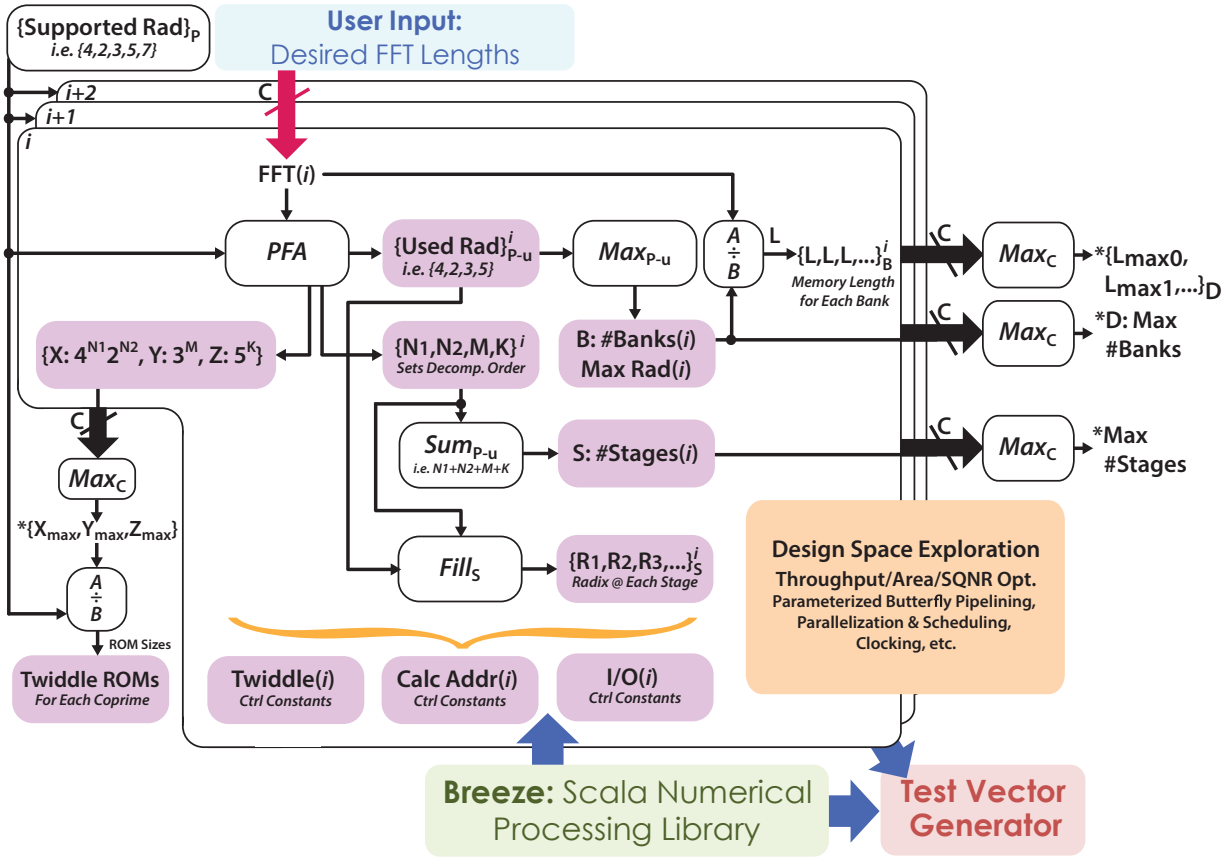

Figure 2.25: Firmware component of the FFT generator. It calculates constants for LUT generation as well as optimized hardware parameters. It is also used for test vector generation.

instance (or in the multi-butterfly scheduling case, the product of the maximum radix and the number of parallel butterflies), and allocates memory per bank to best distribute the FFT data. Because Chisel is a domain-specific language (DSL) for hardware written in Scala, LUT construction can be performed in a few short lines of Scala code, without requiring the user to switch to some intermediate representation.

The hardware template contains configurable controllers that handle data flow between IO, memories, and the butterfly unit, and an optional output normalization block, as in Fig. 2.26. For a given configuration, the hardware template produces just enough optimally-sized instances of each block and connects them together. The FFT template uses the calculated parameters to specify memory sizes and distribution, calculation + I/O rates, the amount of butterfly pipelining, signal bitwidths, and a butterfly to support all needed radices. Parameters can be updated without rewriting code, and synthesizable Verilog is generated directly by Chisel. The only fixed component of the generator is the runtime-

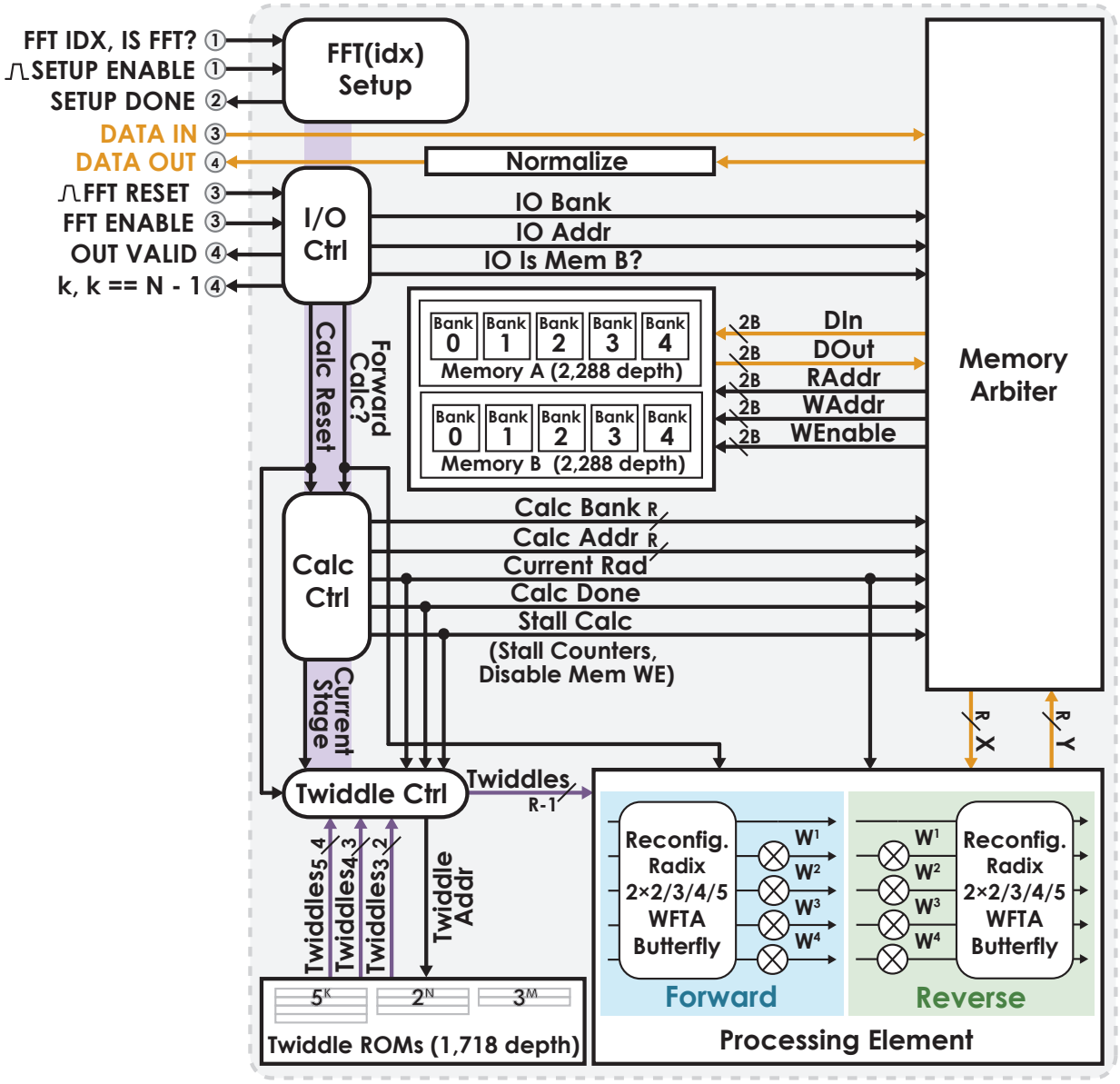

Figure 2.26: Hardware template consisting of LUTs for reconfiguration + twiddles and configurable blocks controlling data flow between IO, SRAMs, and PE(s). The complexity of memory-based designs is primarily in the control logic.

reconfigurable butterfly. This means that it needs to be redesigned to support radices > 7. All other control blocks are completely parameterizable, given PFA + CTA constraints.

For LTE/Wi-Fi, the supported Ns are factorized, and information about the corresponding coprimes ($2^{n,max} = 2048$, $3^{m,max} = 243$, $5^{k,max} = 25$) results in memory that is split into 2×5 SRAM banks (with depths of 4×512 and 240), a reconfigurable radix-$2 \times 2/3/4/5$

butterfly, and a PE with 4 complex twiddle multipliers (supporting up to radix-5) that can be configured for forward or reverse decomposition. A total of 22 real multipliers are used by the Winograd's Fourier transform butterfly + twiddle unit for configurability across all LTE/Wi-Fi sizes. 10 real multipliers are used in the WFTA butterfly, and 12 are used for twiddle multiplication.

2.7.1 Generator Comparison

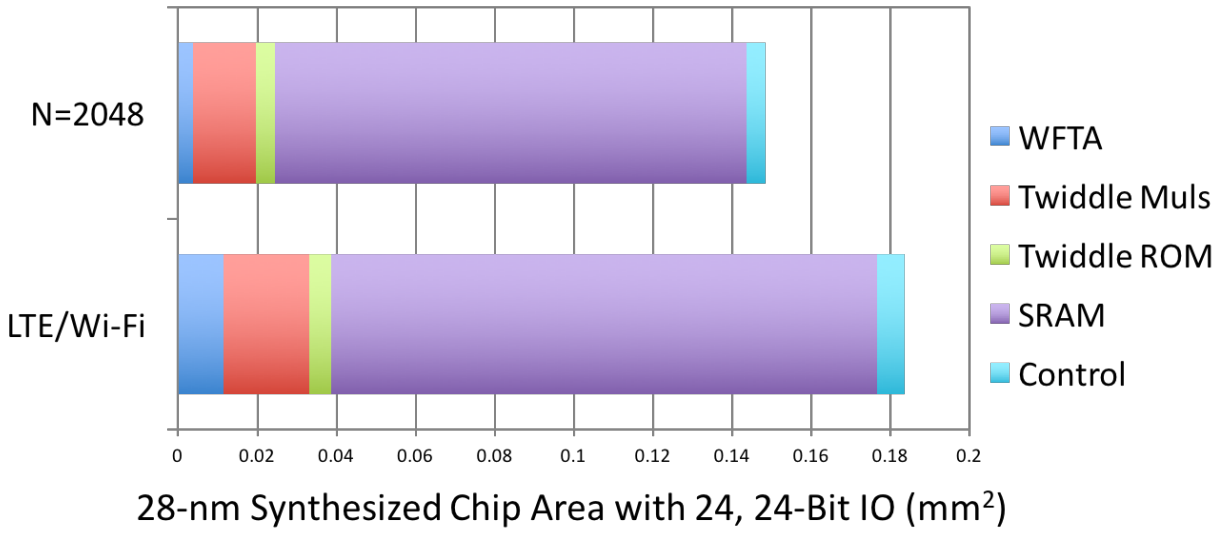

Figure 2.27: Post-synthesis (cell) area breakdown.

The generator capabilities were initially verified in a cycle-accurate fashion via Chisel's built-in tester. As shown in Fig. 2.27, two generated FFT engines with 24, 24-bit complex IO were synthesized with 28nm standard cells and a clock target of 3.9ns to meet stringent Wi-Fi requirements while utilizing the Wi-Fi guard interval for computation. There is a 24% area penalty (compared to a fixed $N = 2048$ engine) to support mixed-radix reconfigurability. The radix-4/2 WFTA butterfly is smaller than the reconfigurable 2/3/4/5 butterfly, and memories occupy a vast majority of the area. Therefore a $3N \rightarrow 2N$ memory reduction is extremely beneficial to both power and area.

As shown in Table 2.19, the FFT generator can create FFT engines with performance comparable to state-of-the-art reconfigurable FFTs [22], [19]. Wang *et al.* [55] propose a pipelined architecture that can be reconfigured/scaled, but it is not implemented as a generator. While achieving higher throughput, the generated FFT engine uses 50% less data memory and 25% smaller twiddle storage than the radix-2 $N = 2048$ iterative FFT from Spiral's online generator [9]. Memory savings are even greater relative to Spiral's streaming FFTs, but with clear area/throughput trade-offs, as seen in Table 2.17. Designs in Table

	This Work	Spiral [9], Rad-4 Streaming	Spiral [9], Rad-2 Streaming	Spiral [9], Rad-2 Iterative	Löfgren *et al.* [41] Streaming
Data Memory	2N	7.3N	7.99N	4N	N-1*
Twiddle ROM	0.75N	0.99N	0.99N	N	-
Clock Cycles	$\sim 3,072$	512	1,024	11,287	2,048
Multipliers	12	56	40	4	30

Table 2.17: Resource comparison for fixed $N = 2,048$. For fair comparison, 4 real multipliers are assumed per complex multiply, and trivial multiplications are not counted. The number of clock cycles of our $N = 2,048$ implementation is calculated assuming a radix-$2\times2/3/4/5/7$ butterfly. * No memory is allocated for I/O unscrambling in [41].

2.18 with fewer multipliers require higher calculation clock rates. Genesis2 [10] and the fixed pipelined architecture from [41] do not address I/O and its impact on memory.

	This Work	Chen *et al.* [19]	Hsiao *et al.* [22]	Xilinx
Data Memory	$2.23N_{max}$	$2N$	$2N$	$2N$
Twiddle ROM	1,718	-	-	-
Calc. Clk./IO Clk.*	$2\times$	$1\times$	$2\times$	$4\times$
Multipliers	22	44	26	16

Table 2.18: Resource comparison for reconfigurable LTE/Wi-Fi FFTs. Comparison numbers are taken from [19]. * represents the ratio of calculation to IO clock rates to support continuous data flow. Note that $2N$ is achievable with our generator for specific FFT lengths, but for LTE/Wi-Fi, we require $2.23N_{max}$ memory.

2.8 Generator Verification via Silicon Implementation

A generated LTE/Wi-Fi compatible FFT engine has been implemented as an accelerator attached to a tethered 64-bit RISC-V Rocket core (Fig. 2.28) [30], all in a 16nm FinFET process. To test the digital system at sufficiently high clock rates without a dedicated clock receiver, the Rocket core and FFT accelerator are clocked via the divided output of a voltage tunable ring oscillator.

The Rocket core allows reading and writing to a series of dedicated memory-mapped I/O registers and SRAMs supporting runtime FFT configuration and input/output data loading/unloading. To simplify testing, a snapshot memory is included at the Rocket/FFT interface, and Rocket can pause FFT streaming to load new test vectors. As seen in Fig. 2.28, the Calculate and Setup registers are strobe/status registers—they are pulsed high for one cycle to align the first time-domain data in a frame to the start of an FFT computation

(in the case of Calculate) and indicate that new FFT parameters such as current FFT N are valid (in the case of Setup). When a valid frame is output by the FFT block and can be read by the Rocket core, the Calculate register is set to 1 by an interface state machine. The Rocket core polls the Calculate and Setup registers to determine when it can proceed with testing.

A Rocket tile consists of an in-order pipeline implementing the RISC-V 64-bit instruction set architecture, a floating-point unit, and L1 instruction and data caches. It is able to communicate with the FFT MMIO registers and SRAM via the chip-scale interconnect standard TileLink. Likewise, TileLink is used to interface with the L2 cache, and parallel/serial adapters are implemented on chip to communicate with the testing infrastructure. Main memory is realized by off-chip DRAM that is accessed via a Xilinx Zynq FPGA. A RISC-V frontend server on the FPGA's ARM processing system loads compiled C code onto the chip.

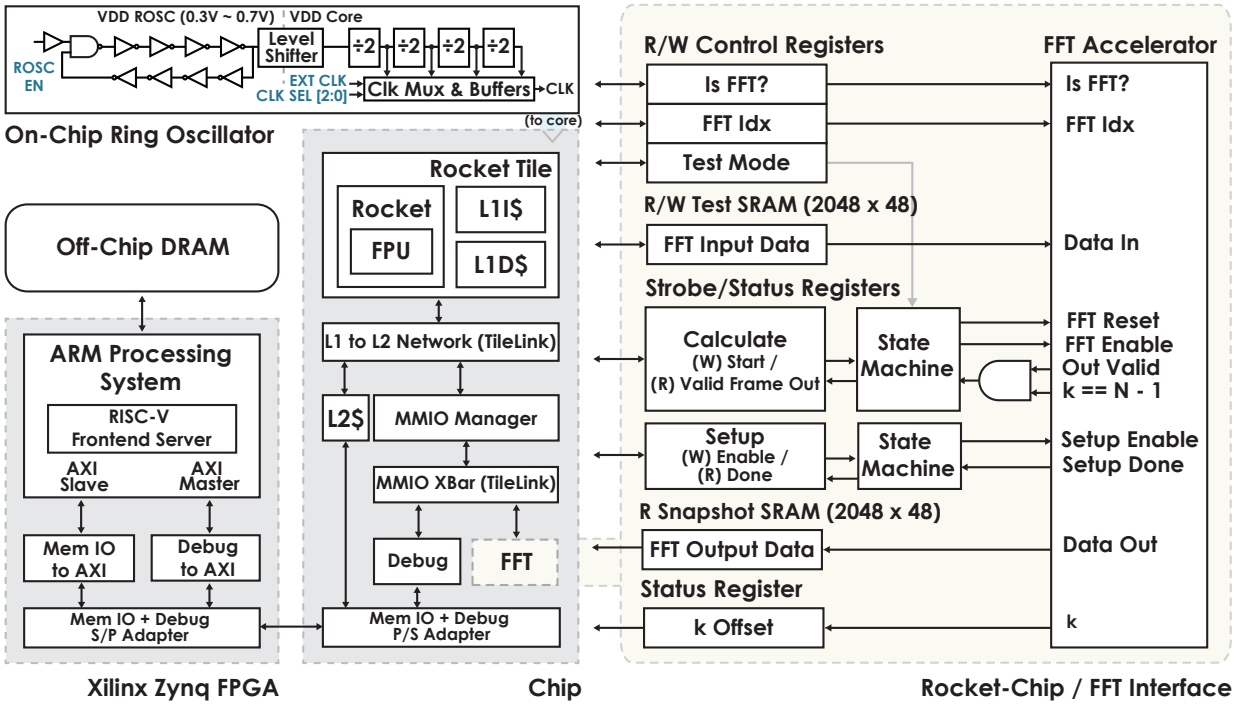

Figure 2.28: Rocket-Chip + FFT system, with snapshot memory.

2.8.1 Generator Verification Methodology with Rocket-Chip

The fixed-point FFT is characterized with complex random inputs. Its quantized results are compared with outputs from a floating-point software FFT from the Scala Breeze numerical processing library [54]. SQNRs for 16-bit and 24-bit implementations across different

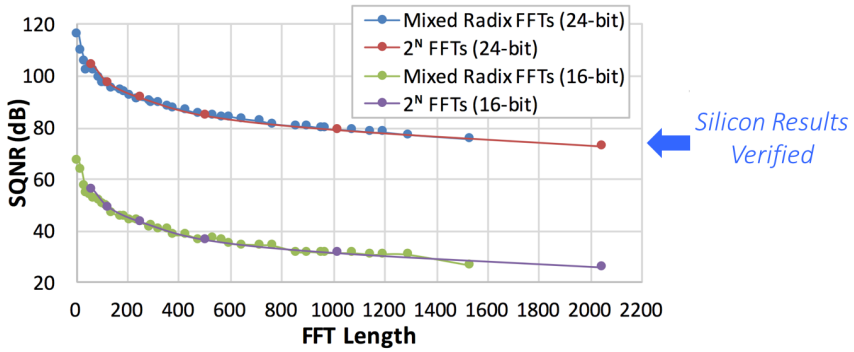

Figure 2.29: Fixed-point SQNR (vs. floating-point) for different FFT lengths and bitwidths.

FFT lengths are shown in Fig 2.29. As expected, there is a ∼6dB/bit SQNR improvement. Additionally, due to the accumulation of rounding errors, the SQNR (when compared to a floating-point implementation) degrades by ∼6dB with each 2× increase in N. Although a 24-bit FFT was taped out to evaluate the high-SQNR regime, a 16-bit implementation meets LTE/Wi-Fi requirements with ∼30% lower power/area than the 24-bit design.

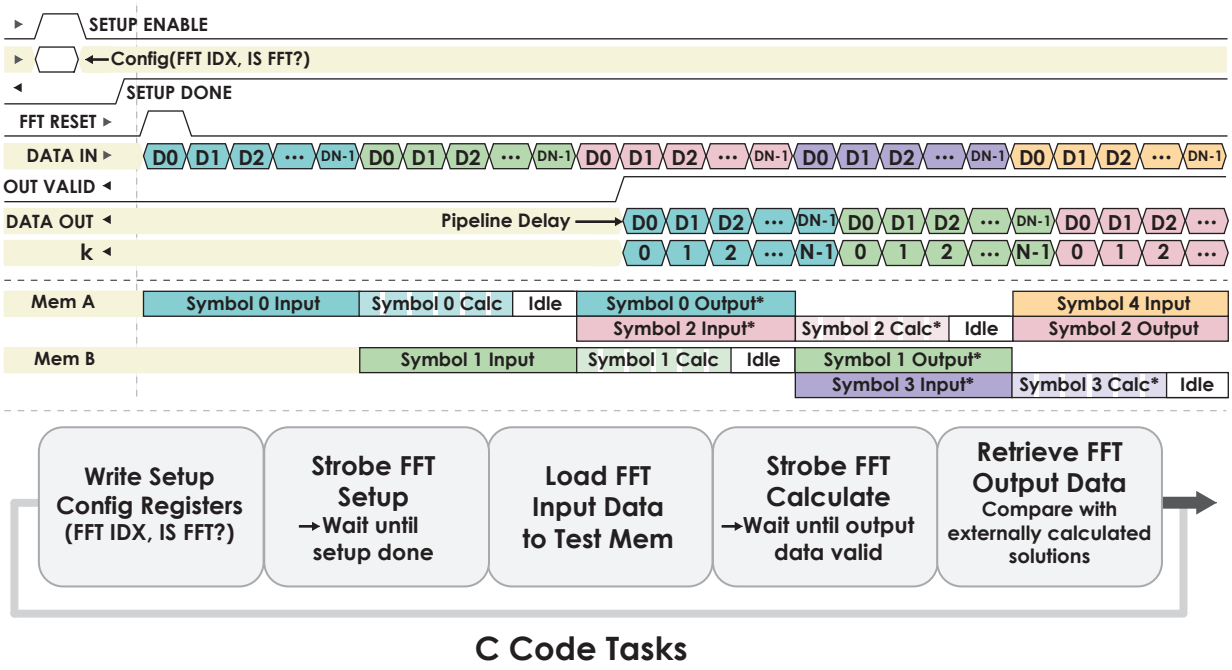

Figure 2.30: FFT timing for runtime configuration and continuous input/output, along with the C task sequence for chip verification. Ping-pong memory timing is shown with alternating forward and reverse* decompositions. Calculation idle/stall periods are marked in white.

Bit-accurate outputs from Chisel simulations are used to automatically generate Verilog test benches for post-place-and-route verification. Likewise, C tests are automatically generated to simplify chip verification. The sequence of C tasks used to verify FFT functionality on the chip is illustrated in Fig. 2.30. The C code exercises the Rocket/FFT interface to reconfigure the FFT length being computed. It then loads random test vectors originally generated in Scala into the FFT test memory, waits until the FFT is done churning on the data (approximately two frames later), and finally verifies that the FFT output matches simulated outputs bit-for-bit. The Scala-generated test vectors are printed to a templated C file as array elements, and the C code is compiled for loading onto the chip.

2.8.2 Measurement Results

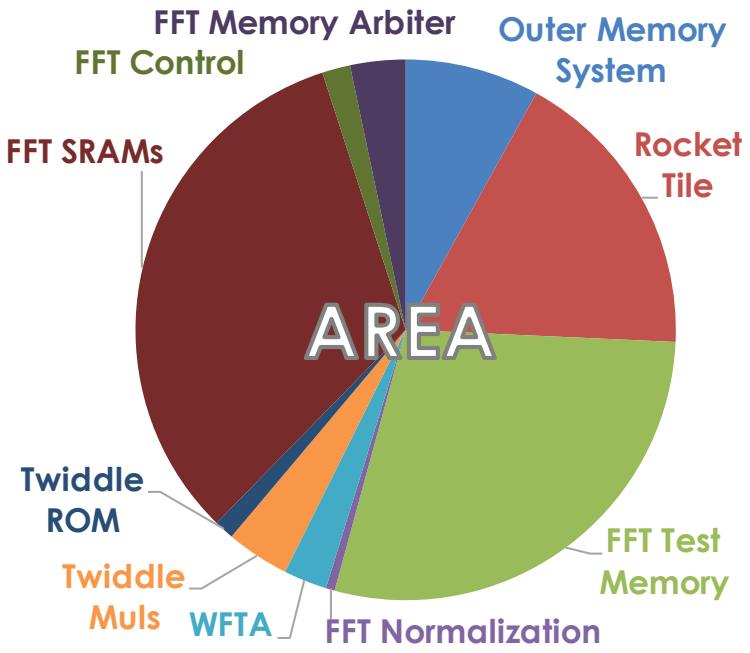

Figure 2.31: Gate area breakdown (0.24mm^2 total).

The FFT occupies an area of 0.37mm^2, while Rocket-Chip, the outer memory system, and FFT test memory occupy 0.39mm^2. The actual gate area, dominated by SRAMs, is approximately 0.24mm^2, as shown in Fig. 2.31. The FFT gate area is 0.11mm^2. Operating at a 570mV supply voltage and running C tests associated with LTE FFT requirements, the total chip power (dominated by the FFT) ranges from 0.46mW to 4.8mW, with clock frequencies up to 61.4MHz. At the same supply, the chip consumes between 2.7mW and 22.6mW when running Wi-Fi tests, which require clock frequencies up to 320MHz (Fig. 2.32).

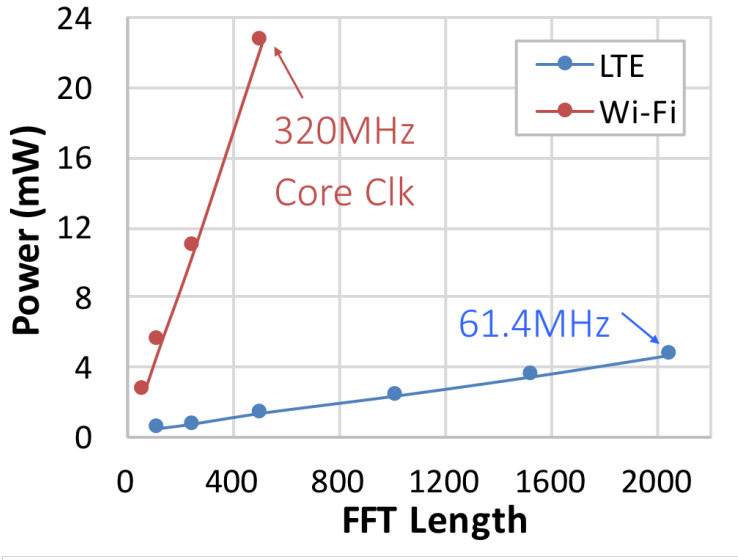

Figure 2.32: Total (FFT + Rocket) measured power at 570mV for LTE/Wi-Fi.

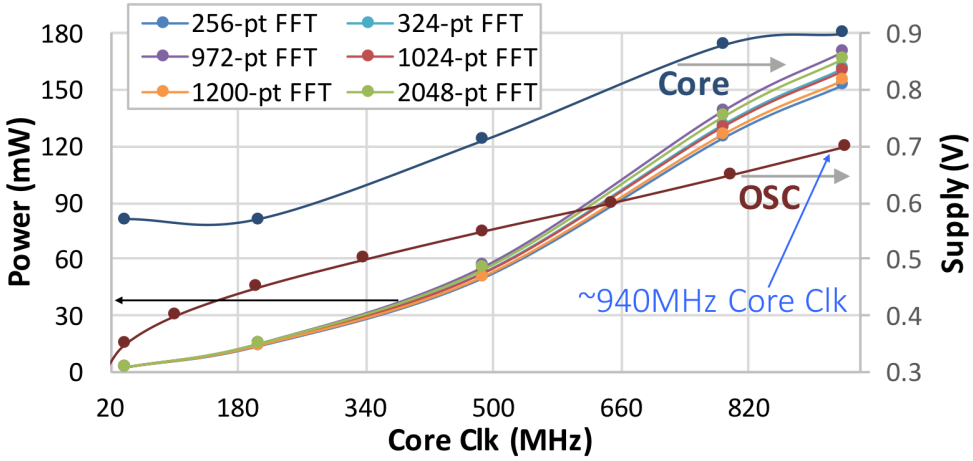

Figure 2.33: Total power required for various FFTs and corresponding supply voltages used at different core frequencies (ring oscillator frequency / 8).

Functionality has been verified up to 940MHz with a 0.9V supply. Measurements are taken by scaling the supply voltage (0.57V to 0.9V) along with frequency (40MHz to 940MHz from the ring oscillator). The total chip power ranges from 2.8mW to 170mW (Fig. 2.33). Because Rocket-Chip is mostly idle while the FFT is running, the FFT accelerator and corresponding test memories draw most of the power. A power breakdown is obtained with a Primetime simulation deploying the same C tests as in measurement; as expected and shown in Figs. 2.31 and 2.34, the SRAMs are the largest single contributor to both FFT

power and area.

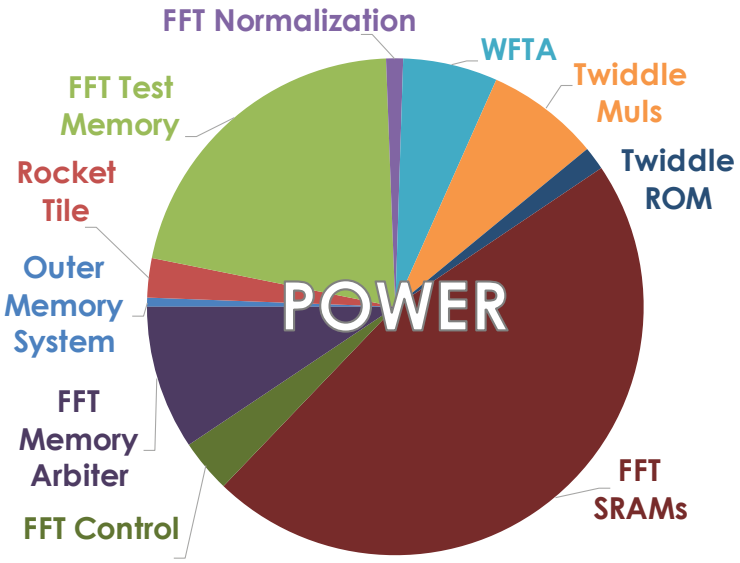

Figure 2.34: Primetime post-synthesis power breakdown for a 2048-point FFT at 520MHz and 0.72V core supply.

In Fig. 2.13, the configuration with the largest ratio of compute cycles to FFT N determines the ratio of calculation to IO clock rates. With a ratio of 2, all other sizes complete in a smaller fraction of the symbol period and hence exhibit lower power (Fig. 2.33). For example, a 972-point FFT that uses 1,905 compute cycles (98% of the $2N$ clock cycles allocated, as illustrated in Fig. 2.13) requires more power than all other measured FFTs. On the other hand, only 284 cycles are needed to complete a 256-point FFT, allowing the FFT engine to sit idle (and in a lower power state) nearly half the time.

2.8.3 Summary

Hardware generators enable rapid and reusable design of hardware instances in advanced technology nodes. A 0.37mm^2 LTE/Wi-Fi compatible $2^n3^m5^k$ FFT instance with performance and area comparable to state-of-the-art (Table 2.19) and integrated as an accelerator within a complete RISC-V processing system was designed and taped out within *1 month* of PDK delivery. The accelerator is optimized for radix-2 $\times$ 2/3/4/5 butterfly reuse and continuous data flow with just $2.23N_{max}$ total SRAM. It requires a twiddle LUT depth of only 1,718 ($0.84N_{max}$), despite supporting *all* LTE and Wi-Fi FFT configurations. The 0.37mm^2 encompasses all blocks that are needed to stream data in/out of the FFT accelerator, in order. The chip's performance and functional correctness have been verified up to 940MHz via C tests loaded onto the Rocket core.

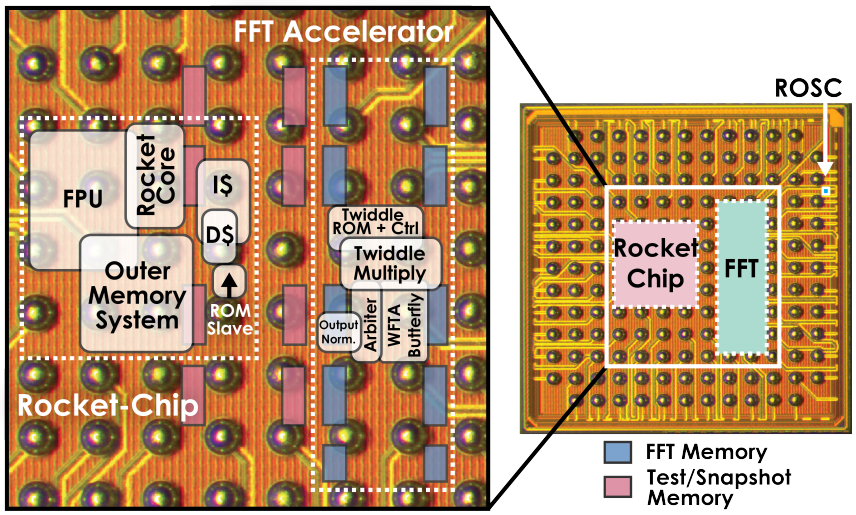

Figure 2.35: Die photo. The total active area is 1.28mm^2. The FFT occupies 0.37mm^2. The active area can be significantly smaller with improved floor planning.

Although not implemented on this chip, the scheduling scheme described in Section 2.4.3.2 to support additional parallel PEs can be used to lower the system clock rate and power consumption.

	This Work	**Yahalom [17]**	**Chen *et al.* [19]**	**Xia *et al.* [18]**
Architecture	Mem.	SDF	DEM	Mem.
FFT Size	64-2048, 1536, 12-1296	128-2048, 1536 + 12-1200	128-2048 + 12-1296	128-2048 + 12-1296
Technology	16nm	28nm	0.18μm	55nm
Word Width	2×24	2×16	2×16	2×16
Mem. Depth	4576 ($2.23N_{max}$)	2047 + 2213[d]	$2N_{FFT}$+ $2N_{DFT}$	$2N_{FFT}$+ $3N_{DFT}$
Gate Count	700K	170K + 511K	316K + 482K	136K + 340K
Area (mm^2)	0.37	0.31	25	1.678
Voltage (V)	0.57[a,b]	0.61-1[a]	-	1.08
Power (mW)	0.46-4.8[a,c] 2.7-22.6[b,c]	0.08-2.93[a]	320	45.5
Clock (MHz)	3.84-61.44[a] 40-320[b]	1.92-30.72[a]	122.88	122.88
Throughput (MS/s)	1.92-30.72[a] 20-160[b]	1.92-30.72[a]	122.88	122.88
Incl. Processor	Yes	No	No	No

Table 2.19: Comparison with other LTE-compatible FFTs.
[a] LTE, [b] Wi-Fi, [c] including Rocket + snapshot memory, [d] unscrambling memory is not reported.

Chapter 3

ACED: A Hardware Library for Generating DSP Systems

3.1 Introduction

Algorithm development in the application domains of computer vision, big data, sensor fusion, and wireless communication has greatly outpaced our ability to deliver dedicated hardware accelerators that promise orders-of-magnitude energy-efficiency and performance improvements over CPU/GPU realizations. Many systems targeting these applications, such as reconfigurable/cognitive radios, RADAR receivers, and wideband spectum analyzers, rely on new algorithmic and architectural solutions, but also on common functions like FFT, matrix multiplication, and digital filtering. Although underlying algorithms and hardware primitives are often reused, architectural realizations must be heavily—and manually—tuned to meet specific performance and deployment platform requirements. In practice, reusing blocks with structural variations increases development and verification time, elucidating the need for reusable and highly parameterized *generators* of hardware instances that support application retargetability, in addition to the generation of optimized and specialized RTL, normally only achievable with one-off instance design. The preceding chapter focused specifically on the motivation behind and design of a runtime-reconfigurable hardware FFT *generator*, satisfying the needs of various stream-based radio applications and useful for prototyping next-generation wireless.

While designing the FFT generator, it became clear that existing tools (both for instance *and* generator design) were lacking in their ability to support DSP contexts and math-to-hardware translation. For example, the default behavior of the + operator in Chisel is to wrap upon overflow. This is the norm in computer arithmetic, where data is represented in bits, but results in incorrect mathematical outcomes. The original version of Chisel presented in [27] did not include a fixed-point type. Therefore, a DSP designer needed to manually convert decimal variables in high-level models to signed integers (i.e., bits) for hardware implementation and propagate binary points across operations by hand. This

same effort was required at module interfaces to evaluate system performance using Chisel testers. As a result, significant debug effort was often spent to align hardware models with their intended fixed-point behavior. In [56], it is estimated that 25% to 50% of the total implementation time is spent on the tedious and error-prone task of floating-point to fixed-point conversion. Matlab HDL Coder provides a more automatic way to convert floating-point Matlab algorithm models into fixed-point hardware. This is accomplished by profiling the ranges of internal variables during a simulation, using representative input data. However, the model and test bench must be "templated" in an HDL Coder-friendly manner, and manual iteration (with user-proposed bitwidths) might be required to achieve desired results.

To address these barriers for developing DSP generators, we have created the hardware library ACED: A Chisel Environment for DSP. Built upon the Chisel hardware construction language [27] and its compiler FIRRTL[57], it operates on three key principles:

1. *Powerful meta-programming with zero-cost abstractions.* Designers can concisely express algorithm intent at higher levels of abstraction by utilizing Scala programming constructs that support DSP-specific and user-defined number representations, while maintaining tight control over implementation details. Hardware optimizations like informed bitwidth reduction via static/dynamic range analysis are performed as library-specific FIRRTL compiler passes [57] that do not change the original user intent.

2. *Unobtrusive optimization and specialization.* Chisel's support for easy design parameterization is enhanced with DSP-specific typeclasses. This enables generator code reuse among all platforms and applications and automatic optimization of each generated hardware instance per context. This simplifies design exploration.

3. *Unified, yet portable, modeling, validation, verification, and testing.* Architecture validation is decoupled from hardware evaluation of quantization effects. By relying on type generics and folding the SystemVerilog Real type into the library, simply switching data types enables easy verification of functional correctness via floating-point simulation *and* analysis of hardware metrics like SQNR. Additionally, the same verification criteria can be used across all design tools and abstractions—tests against a golden model are automatically ported to various points in the design cycle, propagating user intent to reduce errors arising from mistranslation.

To support these principles, ACED introduces the following library and compiler features, which preserve code reusability and enable a single unified validation and verification framework, while still producing highly-optimized hardware instances:

- *Operator and data type parameterization*,
- *Unified systems modeling*, and
- *Powerful static and trace-based bitwidth optimizations.*

While successfully used in software-defined/cognitive radio DSP chips, this thesis separately evaluates ACED's ability to foster design space exploration at the algorithmic, architectural, and implementation levels.

3.2 Background

Traditionally, algorithm-to-hardware is done by iteratively applying designer intent across many tools/abstraction boundaries: algorithm and systems models, RTL implementations, and physical design. Expert architects tediously hand-optimize hardware realizations of algorithms, decreasing portability and malleability. For example, many algorithms use floating-point arithmetic, which cannot be synthesized with basic Verilog/VHDL; designers must undergo an error-prone, unintuitive, and manual process to convert to fixed point, track binary point locations, and create bit-level tests.

3.2.1 MathWorks Simulink

MathWorks's closed-source, model-based design flow, Simulink, simplifies passing user intent to early stages of design/verification, saving 30% in design time [58] when using pre-existing IP. It supports systems validation and verification with double-precision floating-point simulations and leverages Matlab's powerful DSP utilities and "automatic" fixed-point bitwidth optimization for power and area reduction when provided a representative set of application-specific test vectors. The Simulink environment supports the "Chip in a Day" methodology [59], [60], but only accelerates the design of one instance for one hardware platform, rather than accelerating *generator* design targeting *multiple* platforms. It lacks transparent, programmatic, and extensible abstractions, requiring non-trivial manual effort to support multiple targets (FPGA/ASIC) and data types. While high level models are automatically translated into target-independent HDL via HDL Coder, platform-specific specialization requires that IP blocks be manually targeted to tools like Xilinx System Generator or Altera DSP Builder. In addition, ASIC specializations necessary to use foundry-specific SRAMs are not supported. Finally, to support a higher degree of parallelization, blocks must be manually added and hooked up; the model must be redrawn to support different numeric types; and graphical entry limits block-level complexity when custom-defined IPs need to be integrated.

3.2.2 High-Level Synthesis

High-level synthesis (HLS) tools abstract away low-level implementation details from the algorithm-to-hardware translation process. CAD vendors have developed various C (or C++/SystemC/etc.)-to-RTL compilers for this purpose: Mentor Graphics Catapult C, Synopsys Synphony C Compiler, Cadence Stratus HLS, Intel HLS Compiler, and Xilinx Vivado [61], to name a few [62]. LegUp is likewise an academic-turned-commercial HLS tool for tar-

geting C to FPGAs or hardware accelerators [63]. The academic endeavor FCUDA presents a CUDA-to-FPGA flow [64]. Intel's FPGA SDK for OpenCL allows designers to write code for FPGAs using OpenCL. Darkroom is a compiler that maps high-level image processing code (written in the custom Darkroom programming language) to FPGAs, ASICs, and CPUs [65]. HLS is useful for designing blocks that require complex memory access scheduling, deep pipelining, and sequential logic. However, many DSP datapaths have high degrees of parallelism, requiring carefully structured SystemC code. Finally, hardware generation from recursive functions is not allowed, except with template metaprogramming.

3.2.3 Tools for Hardware Generation

Other tools also enable designing generators. Genesis2 augments synthesizable SystemVerilog with Perl extensions for generator parameterization [10]. The generator code is fairly verbose; however, the end user is expected only to tweak high-level parameters for specialization. Genesis2 relies on two decoupled languages, increasing the likelihood of generating difficult-to-debug syntax errors. Spiral is a generator specifically targeting DSP hardware [9]. Designers input a linear transform of a fixed size, and through a series of algorithm-informed optimization phases, the tool chooses the most suitable architecture, e.g., the optimal amount of sequential reuse required of a datapath, to meet application needs. Although powerful, Spiral is designed for static function generation and is not well suited for runtime-reconfigurable architectures, which require that interconnects be non-static. We believe a better approach is building generators in an embedded domain-specific language catered to hardware.

3.2.4 Hardware Construction (Generation) Languages

Domain-specific languages for hardware like PyMTL, which is built on top of Python, enable a unified design environment for hardware modeling and generation with more "software-friendly" syntax [66]. Likewise, the hardware construction/generation languages Bluespec (closed-source) and Chisel (open-source) enable systems modeling, generator construction, and test environment creation with one underlying functional and object-oriented programming language [67], [27]. Each supports:

- *Custom-defined number abstractions*,
- *Compile-time type-checking*,
- *Type polymorphism and operator overloading for hardware template parameterization*,
- *Recursive functions to generate hardware*, and
- *Functional constructs (e.g., map, reduce, etc.) that enable concisely expressing structured datapaths.*

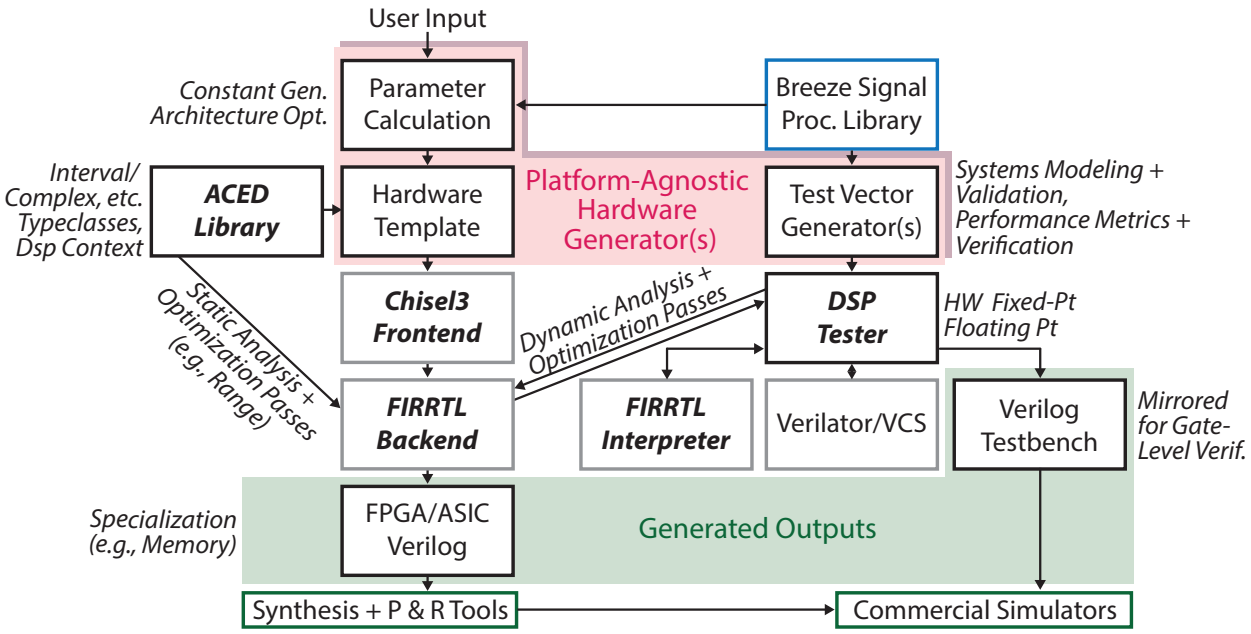

Figure 3.1: ACED hardware generator design environment.

Bluespec and Chisel are both architecturally transparent, supporting fine-grained control of implementation details and leading to faster convergence to *optimal* designs when compared to HLS tools that sacrifice implementation flexibility for automatic (but potentially suboptimal) optimization via tool heuristics. Bluespec provides an additional abstraction layer for multi-rate interfaces between heterogeneous functional units. However, end users can create custom interface classes like those in Bluespec for their Chisel-derived hardware library. The Chisel design methodology in particular encourages unit testing with minimal boiler plate code, making it easy to observe the behavior of submodules outside the systems context. Most importantly, design specialization—HLS-like automatic pipelining, memory macro to SRAM mapping, etc.—is achieved at the lowest level of abstraction because of the open-compiler FIRRTL framework that supports user-created optimization passes.

3.3 ACED: A Chisel Environment for DSP

ACED (Fig. 3.1) provides constructs to capture and propagate designer intent down the hardware abstraction hierarchy without redundant specification. To allow designers to focus on algorithms, it separates algorithmic and implementation abstractions, yet supports platform/application-specific specialization and optimization.

ACED is implemented on top of the open-source Chisel hardware construction language and its open-source compiler, FIRRTL [57]. The FIRRTL (Flexible Intermediate Representation for RTL) compiler enables the creation of new hardware libraries that are agnostic to

specific deployment platforms like FPGAs and ASICs by providing mechanisms to convert high-level, target-independent constructs into technology-specific RTL. The open-compiler framework supports custom analysis passes that provide design introspection. Additionally, compiler passes for hardware optimization/specialization are supported. As an example, implementations of "memory" are platform-specific. Memory can be mapped to block RAMs or distributed RAMs on FPGAs or registers and SRAMs on ASICs, all of which might have different input and output timing requirements, etc. In a more traditional approach, a custom memory wrapper must be written for each target platform and manually swapped into the RTL as needed. Instead, Chisel supports a generic memory construct that gets mapped to technology-specific blocks automatically and correctly through a FIRRTL compiler pass, without any intervention from the designer. Such a pass requires both design introspection and transformation capabilities, which we leverage in ACED for interval analysis and bitwidth reduction.

Because Chisel is written in Scala, existing Scala and Java libraries like the Breeze numerical processing library can be used to generate "golden model" references and constant coefficients for DSP systems [54]. The same underlying language is used by generators, optimizing/specializing compiler passes, and the testing environment, allowing a single set of tests to verify the system. This reduces the likelihood of translation errors and simplifies development and design-space exploration. High-level data structures support fast, cycle-accurate simulation and validation with non-synthesizable floating-point through the FIRRTL Interpreter in addition to automatic instance generation for domain-dependent hardware using fixed-point, complex, mixed-radix, and other number abstractions without redesign.

3.3.1 Operator and Data-Type Parameterization

Type-generic generators increase code reuse; for example, one type-generic FIR filter generator can replace separate complex- and real-type FIR filter generators. Type-generic generators require:

- *Operator parameterization*,
- *Native operator and data types*,
- *Flexibility to specify new number types*, and
- *The ability to parameterize the type used by a Chisel circuit via type generics.*

Many DSP operations have implementation details, such as rounding modes (round half-up, floor, ceil, & truncate), overflow modes (grow, wrap, & saturate), and depth of pipelining, that affect performance/area trade-offs. Consistently specifying these per-operation properties is tedious and error-prone; ACED provides a *DspContext* to set and override these properties. The hardware designer does not need to manually add shift registers after operators like multiply and add; this is done automatically when the pipeline depth is specified.

Additionally, after specifying rounding modes to achieve a desired number of fractional bits, the rounding circuitry (e.g., mux) is automatically added by ACED. *DspContext* sets defaults at the top-level module scope and may be overridden hierarchically. If necessary, a single operator's parameters can be overridden.

Powerful native types help achieve functionally correct designs. To design fixed-point DSP hardware in HDLs such as Verilog or VHDL, one must manually track binary points across operations. Chisel's *FixedPoint* type automatically propagates binary points. To enable range-based analyses & optimizations, the native type *Interval* was added. To support specifying new number types and parameterizing Chisel circuits via type generics, ACED abstracts data types and representations using typeclasses, a functional programming pattern that enables type-safe polymorphic code. ACED extends the Scala numeric library Spire [68] with typeclasses for Chisel/ACED types including *UInt*, *SInt*, *FixedPoint*, *Interval*, *DspComplex*, and *DspReal*.

3.3.1.1 Typeclass Construction

Designers can create new number types and use them with other ACED library components. For example, non-2^n FFTs benefit from mixed-radix number types, where full adders are replaced with custom adders built with simple subtractor + mux modulo operations. The following illustrates a custom complex number type for use with type-generic generators. First, a *DspComplex* Chisel type is created from a Chisel *Bundle* (like a struct):

Listing 3.1: *T <: Data:Ring* says generic type *T* is a Chisel hardware type and implements Ring typeclass.

```
class DspComplex[T<:Data:Ring](val real:T, val imag:T)
    extends Bundle { ... }
```

ACED uses typeclasses for algebraic structures (ring, order, signed, real, etc.). A typeclass *Ring[T]* defines the behavior of addition, multiplication, subtraction, and negation for objects of type *T*; *DspComplex* supports ring operations via a *Ring[DspComplex[T]]* instance. The complex × operator uses three real multipliers:

Listing 3.2: Typeclass for *DspComplex*.

```
class DspComplexRing[T<:Data:Ring] extends Ring[DspComplex[T]] {
  def times(f: DspComplex[T], g: DspComplex[T]) = {
    val p1 = f.real * (g.real + g.imag)
    val p2 = (f.real + f.imag) * g.imag
    val p3 = (f.imag - f.real) * g.real
    return DspComplex.wire(p1 - p2, p1 + p3)
  } ...
}
```

Implementing *DspComplexRing* allows *DspComplex* to be used for any generator with type constraint *T <: Data:Ring*. If such a generator is invoked with an instance of *DspComplex*, all instances of $f * g$ will call this implementation of *times(f, g)*.

3.3.1.2 An FIR Filter Generator Example:

The following direct-form, real-coefficient FIR filter template illustrates how ACED is used to make hardware generators. It directly maps the equation $y[n] = \sum_{k=0}^{N} h[k]x[n-k]$ into hardware.

Listing 3.3: FIR Filter Generator

```
class FIR[T <: Data:Ring:ConvertableTo](genI: => T,
    genO: => T, cfs: Seq[Double]) extends Module {
  // Set module input/output ports
  val io = IO(new FilterIO(genI, genO))
  // Specify pipeline stages and dataflow precision
  val newContext = DspContext.current.copy(numMulPipes=3,
      binaryPoint= Some(14))
  // New scope has newContext parameters
  DspContext.alter(newContext) {
    // Generate register sequence to delay input (taps)
    val tps = cfs.tail.scanLeft(io.in)(
      (in, _) => RegNext(in, init = Ring[T].zero))
    // Make constants from floating-point coefficients
    val cs = cfs.map(c => ConvertableTo[T].fromDouble(c))
    // Create one multiplier per tap/coefficient pair
    val ms = tps.zip(cs).map{case (t,c) => t context_* c}
    // Set output to sum of all multiplier outputs
    io.out := ms reduce (_ context_+ _)
  }
}
```

The template takes inputs/outputs of type *Data* that have the typeclasses *Ring* (for add/multiply operations) and *ConvertableTo* (for translating Scala Doubles into the correct Chisel literal type via the *fromDouble* function). The Scala compiler checks that any invocation of the generator satisfies these type constraints. Functional programming constructs represent DSP operations concisely. Scan is used to create the delay taps, map is used for coefficient/tap multiplication, and reduce is used for the summation. The filter order is indirectly specified via the number of coefficients in the *cfs* parameter. *DspContext* sets the number of pipeline registers for multiplication and the number of fractional bits used to represent the filter coefficients.

This code creates an FIR module specialized for Interval types:

Listing 3.4: FIR instantiation with *Interval* types.

```
new FIR(Interval(range"[-16, 16).2"),
        Interval(range"[?, ?].4"), ...)
```

Likewise, a filter with complex inputs is instantiated with:

Listing 3.5: FIR instantiation with *DspComplex* types.

```
new FIR(DspComplex(Interval(range"[-16, 16).2")),
        DspComplex(Interval(range"[?, ?].4")), ...)
```

3.3.2 Unified Systems Modeling and Verification

When verifying and validating systems, the same tests should be propagated across all layers of the design hierarchy, from the systems model down to behavioral and gate-level RTL code. Reusable tests prevent design intent from being mistranslated across abstraction and tool boundaries. The ACED library contains a Chisel tester extension that natively supports DSP-specific number representations and error tolerance parameters. ACED's *DspTester* tracks the sequence of user inputs, prompted design outputs, and time steps, mirroring them one-for-one in an automatically generated Verilog testbench that can be run on specialized designs. This generated testbench verifies that gate-level netlists are properly inferred and that macro replacement, such as swapping out Chisel memories for FPGA BRAMs or ASIC SRAMs, does not result in functionality or timing changes.

To decouple algorithm/architecture validation (whether the *fundamental* algorithm or architecture is able to achieve a performance target) from secondary effects like insufficient bitwidth allocation that degrade system metrics, ACED again relies on typeclasses. It can often be hard to determine if poor DSP system performance is due to quantization error from finite word length or an algorithmic error. Using floating-point data representations virtually eliminates quantization errors and makes it easy to evaluate algorithmic performance and correctness. ACED provides a *DspReal* type that implements non-synthesizable double-precision floating point, simulatable with either the FIRRTL Interpreter or SystemVerilog's Real type. Typeclasses have been implemented for *DspReal*, so floating-point numbers can be used with type-generic generators without any changes to the generator. As in Section 3.3.1, a block intended to be used with *FixedPoint* or *Interval* inputs can be verified with floating point data simply by swapping out those synthesizable types for *DspReal* at instantiation. Once mathematical correctness has been verified, the template can be re-parameterized to use a synthesizable type for implementation.

It is frequently useful to be able to integrate analog or non-synthesizable floating-point models together with synthesizable hardware for system evaluation. For example, a designer building a radio baseband may want to study the effects of ADC or FFT quantization error on either a subsequent demodulator's reliability or the entire system performance. One way to do this is to model a radio transmitter and over-the-air channel using library functions in

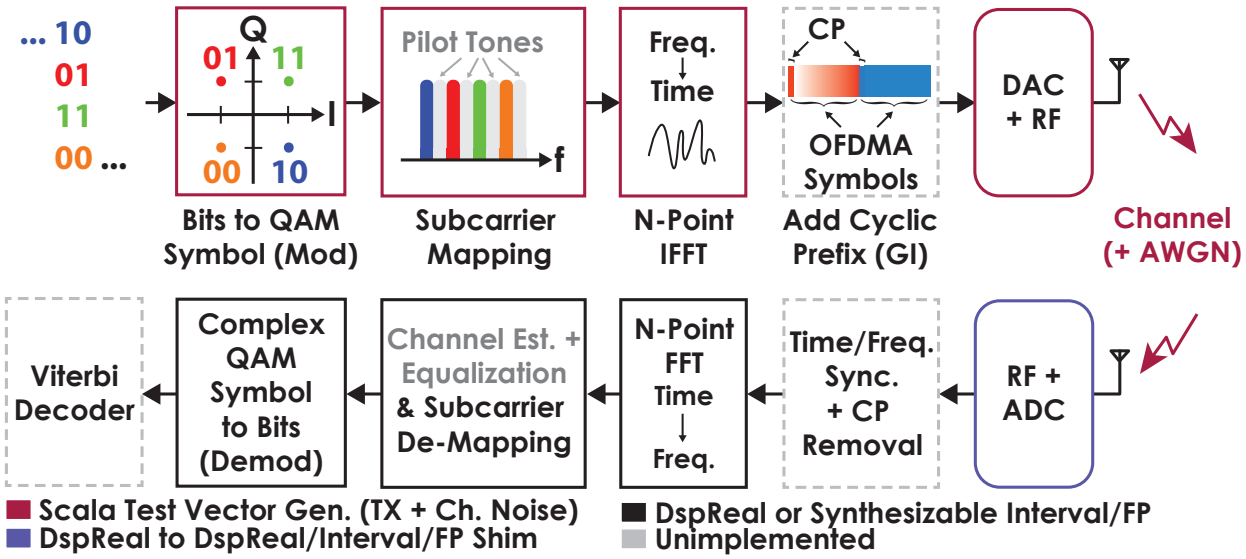

Figure 3.2: A system model of an OFDM transmitter/receiver pair with an additive white Gaussian noise channel. At the transmitter, each set of b data bits is represented as a point on a complex (I/Q) constellation diagram and mapped onto a subcarrier in the frequency domain. The time-domain waveform is cyclically extended to improve robustness against multipath interference and turn the channel response into a cyclic convolution, allowing for single-tap equalization at the receiver. Then it is up-converted in the analog domain and transmitted over the antenna. The receiver undoes channel effects and the steps used by the transmitter, corrects for any timing and frequency mismatches between itself and the transmitter, and tries to recover the data from its noise-corrupted input.

Scala and then feed the model output via the *DspTester* to a simple receive chain consisting of an analog/mixed-signal ADC model followed by parameterized FFT and demodulator blocks that can use either synthesizable or non-synthesizable constructs (Fig. 3.2). Using FIRRTL transformation passes, the ADC model can be swapped out for a custom-designed ADC macro achieving the same effective number of bits at implementation. To support this, a *ChiselConvertableFrom* typeclass that implements "shims" between *DspReals* and synthesizable types has been created. Interfaces at module boundaries can be mated together by pattern matching input/output data types and using this typeclass.

One important system metric for radio receivers is the symbol error rate (SER) vs. the input signal-to-noise ratio (SNR). Alternatively, the bit error rate (BER) can also be studied. When an M-QAM modulation scheme is used, e.g., Fig. 3.3, where $M = 2^b$ and b is even, the best theoretically achievable SER (using ideal RX blocks) is given by [69]

$$\mathrm{SER} = 1 - \left(1 - 2\left(1 - \frac{1}{\sqrt{M}}\right) Q\left(\sqrt{\frac{3E_s/N_0}{M-1}}\right)\right)^2 . \tag{3.1}$$

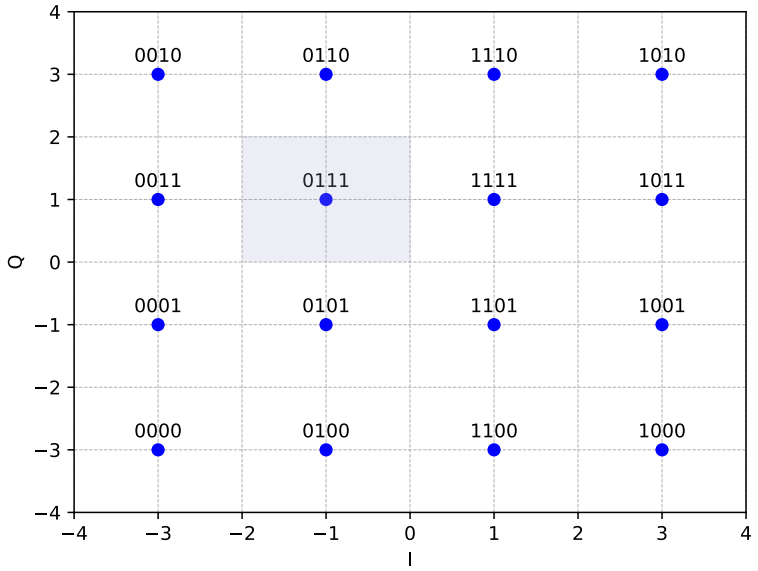

Figure 3.3: 16-QAM constellation diagram. A sequence of 4 bits is converted to a complex symbol whose real and imaginary components taken on a value of -3, -1, 1, or 3. Gray codes are used so that adjacent symbols only differ by one bit, reducing bit errors. Upon demodulation, noisy inputs located at regions adjacent to a symbol are mapped to that symbol, e.g., all inputs in the blue box are mapped to 0b0111.

Here, $Q(x) = \frac{1}{2}\text{erfc}(x/\sqrt{2})$, and the baseband SNR is defined as E_s/N_0 i.e., the ratio of the signal power to noise power. The SER vs. SNR for Fig. 3.2's systems model with various combinations of synthesizable and non-synthesizable blocks is compared to the theoretical limit in Fig. 3.4. This can be used to determine system sensitivity to the performance of individual blocks. Additionally, the FFT generator described previously was evaluated for use in the context of this baseband receiver chain and a separate spectral analysis system. The SQNR achievable for different FFT bitwidths is given in Fig. 3.5 and used to inform bitwidths of generated FFT instances in practical designs, which also integrated custom SAR ADCs that were modeled using ACED constructs.

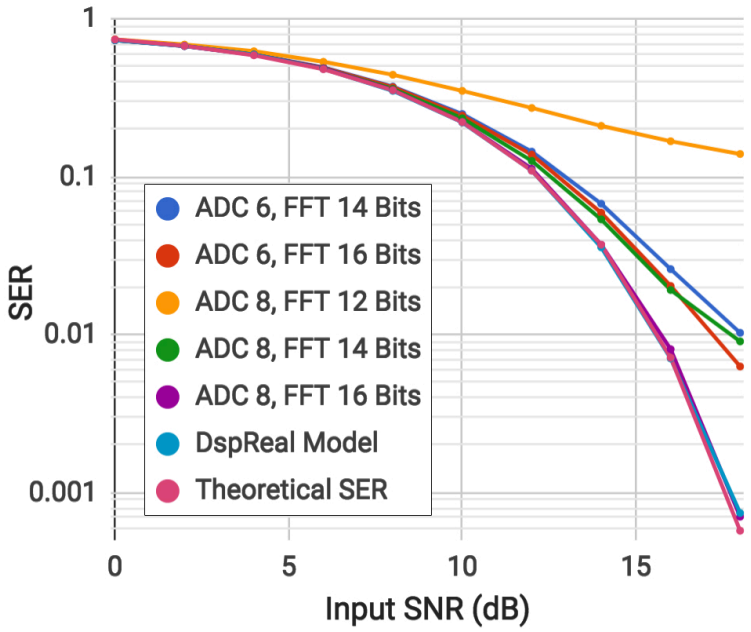

Figure 3.4: SER vs. input SNR @ baseband using 16-QAM modulation and an $N = 128$ FFT. ADC quantization and FFT output bits are indicated for each I and Q channel. The SER is very sensitive to FFT output bitwidths below 16 bits.

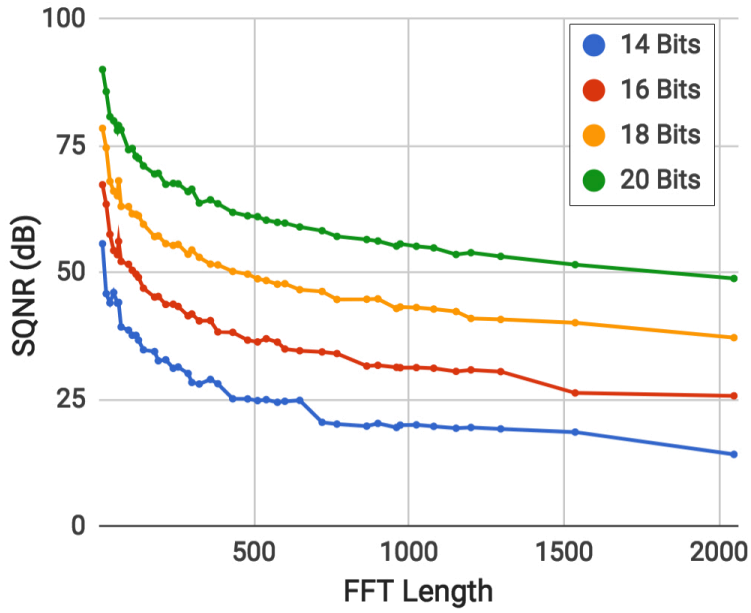

Figure 3.5: SQNR vs. FFT length for different output I/Q bitwidths.

3.3.3 Interval-Based Arithmetic and Bitwidth Reduction

DSP designers may know module input/output ranges, but often rely on synthesis tools to infer internal node widths. Unfortunately, the tools cannot infer a signal's range from its bitwidth, resulting in suboptimal power and area results. Designer intent can be better captured by directly encoding input ranges into the design, allowing automatic range propagation and bitwidth reductions.

Simple forward/backward range propagation [70], [71], while still conservative, offers improvements without complex symbolic analysis of ranges. Potential ranges are tracked for each operation, and the solver finds the worst case bitwidth. Affine analysis of ranges [72] can cancel correlated terms (e.g., $A - A$ has a range $[0, 0]$), leading to a more compact design at the expense of more complex analysis. These techniques are used in high-level synthesis flows [73], which unfortunately require users to express their designs using non-zero-cost abstractions. The benefits of these optimizations can be offset by the difficulties encountered when porting existing RTL to a new flow and/or application. Dynamic bitwidth analyses (e.g., used by Matlab's floating-point to fixed-point conversion tool [58]) can significantly improve upon these methods. They require a thorough set of application-specific test vectors, potentially leading to long runtimes. Without complete coverage, the design's correctness is only statistically guaranteed. Unrepresentative test vectors can incorrectly add or remove hardware and result in wasted bits or reduced dynamic range. At some expense, inserting guard bits and saturation logic makes the output "more correct" for inputs that exceed the test vector bounds.

Manual bitwidth optimization, especially for *generators*, is inconvenient and error-prone. To address this, ACED supports automatic (static) range propagation that aids bitwidth optimization via FIRRTL passes. It also provides dynamic analysis—FIRRTL Interpreter simulation results can automatically direct bitwidth reduction.

3.3.3.1 Static Interval Optimization

An *Interval* type containing precision, a lower bound, and an upper bound was added to Chisel and FIRRTL. Bounds are closed [], open (), or unknown. Additionally, they can be half open e.g., the bounds [0, 4) and (0, 4] contain 0/4, but do not contain 4/0, respectively. Precisions are unknown or an integer. Hardware components like wires, ports, and registers can be declared with these *Interval* types. Additionally, two new primitive operators enable limiting an expression to an interval: *clip* and *wrap* will clip (or wrap) the first input argument's value to the interval of the second argument. For example, if $x = 5$ and $y : Interval[0, 3]$, then $clip(x, y)$ is 3, while $wrap(x, y)$ is 1. For common but simple use cases as in the previous example, *wrap* acts like a modulo operation, while *clip* saturates a value. Chisel *Reg* passes interval bounds between its input and output. Intervals can also be manually reassigned with *reassignInterval*.

To generated Verilog with signed data from *Interval* types, FIRRTL passes perform the following:

Op	Chisel Baseline	Interval Lower Bound	Interval Upper Bound	Precision
$\text{add}(x, y)$	$\max(x_w, y_w) + 1$	$x_l + y_l$	$x_h + y_h$	$\max(x_p, y_p)$
$\text{sub}(x, y)$	$\max(x_w, y_w) + 1$	$x_l - y_h$	$x_h - y_l$	$\max(x_p, y_p)$
$\text{mul}(x, y)$	$x_w + y_w$	$\min(x_l y_l, x_l y_h, x_h y_l, x_h y_h)$	$\max(x_l y_l, x_l y_h, x_h y_l, x_h y_h)$	$x_p + y_p$
$\text{mux}(p, x, y)$	$\max(x_w, y_w)$	$\min(x_l, y_l)$	$\max(x_h, y_h)$	$\max(x_p, y_p)$
$\text{wrap}(x, y)$	x_w	y_l	y_h	x_p
$\text{clip}(x, y)$	$\min(x_w, y_w)$	$\max(x_l, y_l)$	$\min(x_h, y_h)$	x_p
$\text{shl}(x, C)$	$x_w + C$	$2^C x_l$	$2^C x_h$	x_p
$\text{shr}(x, C)$	$x_w - C$	$\lfloor 2^{x_p - C} x_l \rfloor / 2^{x_p}$	$\lfloor 2^{x_p - C} x_h \rfloor / 2^{x_p}$	x_p
$\text{dshl}(x, u)$	$x_w + 2^{u_w} - 1$	$\min(x_l, 2^{2^{u_w} - 1} x_l)$	$\max(x_u, 2^{2^{u_w} - 1} x_u)$	x_p
$\text{dshr}(x, u)$	x_w	x_l	x_u	x_p
$\text{bpshl}(x, C)$	$x_w + C$	x_l	x_u	$x_p + C$
$\text{bpshr}(x, C)$	$x_w - C$	$\lfloor 2^{x_p - C} x_l \rfloor 2^{C - x_p}$	$\lfloor 2^{x_p - C} x_h \rfloor 2^{C - x_p}$	$x_p - C$
$\text{bpset}(x, C)$	$x_w - x_p + C$	$\lfloor 2^C x_l \rfloor / 2^C$	$\lfloor 2^C x_h \rfloor / 2^C$	C

Table 3.1: A subset of corresponding constraint expressions for each supported FIRRTL primitive operation. Listed operation arguments are FIRRTL interval-typed expressions (x, y), a constant integer C, or an unsigned-typed (*UInt*) expression u. The argument subscripts refer to the lower bound, upper bound, precision, or width of the expression (e.g., x_l, x_u, x_p, or x_w, respectively). Constraint expressions are: $+$, $-$, $*$, max, min, and floor.

1. *Resolve unknown precisions at intermediate and output nodes.*
2. *Trim interval bounds to known precision.* Open bounds are converted to closed bounds. For example, [0, 3).0 is converted to [0, 2].0. If a fractional bound (e.g., 0.9) cannot be represented with the specified precision (e.g., 1 binary point), it is trimmed to a representable bound (e.g., 0.5) within the original interval.
3. *Resolve all upper and lower bounds at intermediate and output nodes.*
4. *Align precisions by shifting binary points.*
5. *Convert Interval types and associated operations to SInt types and operations.* For example, *bpshl* is ultimately implemented as a *shl* (arithmetic shift left) operation.

The relationships between bound/precision constraints and associated FIRRTL expressions are complex. For output precision, many primitive operations simply take the maximum precision of their input arguments. Others explicitly set the precision or shift the binary point position. Finally, some have unique rules e.g., multiplication takes the sum of the input arguments' precisions. A subset of these relationships is summarized in Table 3.1.

Unknown bounds and precisions are uniquely named with variables. After collecting monotonic constraints from analyzing FIRRTL connections and expressions and merging constraints on the same variable, a custom constraint solver uses a forward-backward algorithm to solve the constraints. Finally, all values specified by variable names are replaced with their solved bounds or precisions.

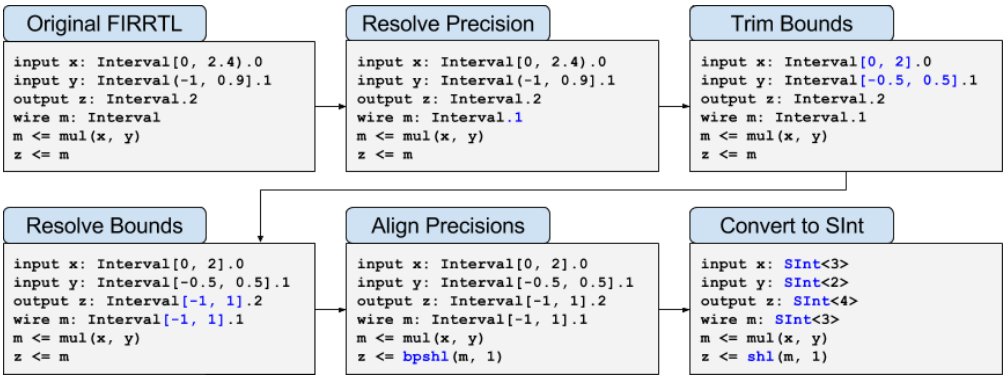

Figure 3.6: A demonstration of transformations which infer precisions and bounds of *Interval* circuit elements that are then converted to signed integer types.

A simple example can demonstrate the savings of ACED range analysis over Chisel's standard bitwidth propagation. The product of three 2-bit unsigned numbers requires 6 bits according to Chisel (Table 3.1), but since the input ranges can be at worst [0, 3], the worst-case output range is [0, 27], requiring only 5 unsigned bits; the savings are amplified as operations are chained together.

3.3.3.2 Dynamic Interval Optimization via Hardware Profiling

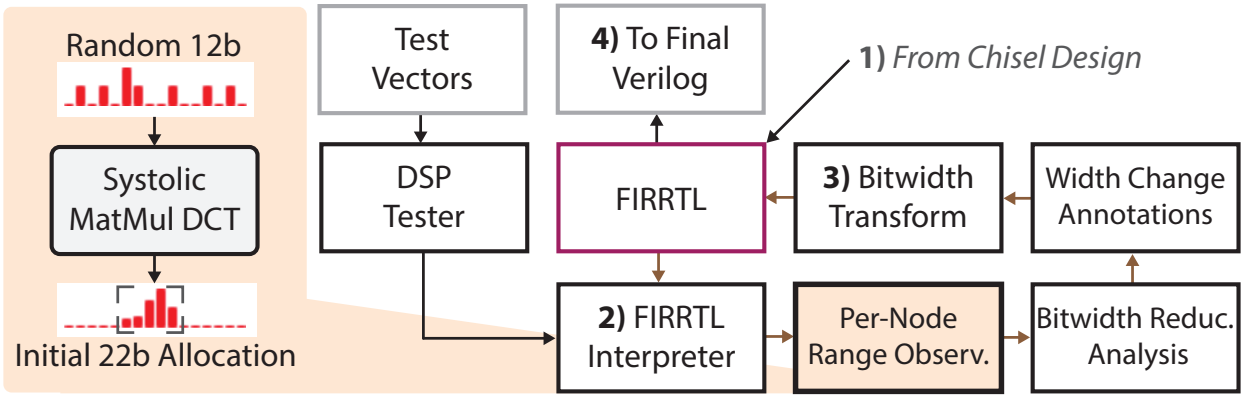

Figure 3.7: Bitwidth optimization procedure using the FIRRTL Interpreter for dynamic range analysis. Utilization is reported visually to help build design intuition.

As shown in Fig. 3.7, to further optimize the bitwidths of a design with a thorough set of test vectors, dynamic interval analysis using the FIRRTL Interpreter can be turned on to automatically trim bitwidths of internal nodes.

Starting with an unoptimized circuit, the FIRRTL Interpreter captures the smallest and largest values seen at significant (named) internal nodes. It tracks the min./max., mean,

and standard deviation at these nodes and displays a histogram of values to give designers a better understanding of data flow through the circuit. As an example, when randomly generated test vectors occupying a 12-bit input range are fed into the circuit in Fig. 3.7, it is useful to see that the original allocation of 22 bits at the output is far too conservative, and only ∼31% of the range is actually utilized.

FIRRTL annotations are generated for downstream optimization passes to: (1) adjust node widths downward based off of their simulated extrema (additional guard bits can be added) or (2) adjust widths to support $x\sigma$ ranges. At the expense of additional logic, saturation operations can be added to protect from undesirable overflow behavior. Finally, bitwidth-optimized Verilog RTL is generated.

3.4 Bitwidth Optimization Results

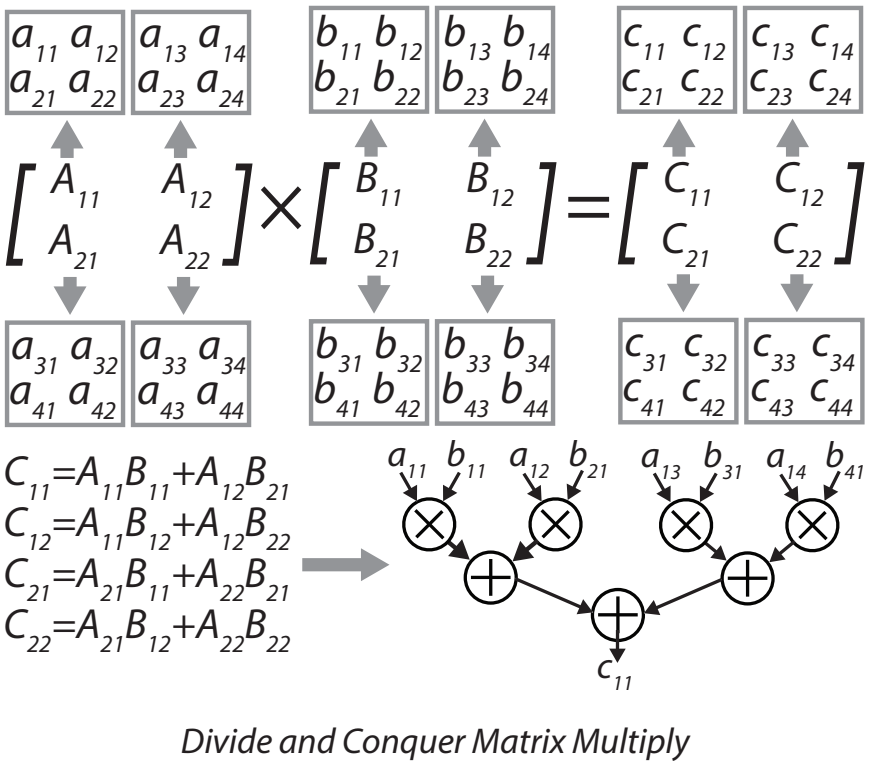

Figure 3.8: a) Divide + conquer MatMul. *Intervals* are automatically propagated.

To evaluate ACED, generators supporting different $\boldsymbol{AB} = \boldsymbol{C}$ matrix multiplication algorithms and architectures (Figs. 3.8, 3.9) have been built. Recursive divide-and-conquer MatMul algorithms showcase bitwidth reduction from static and dynamic interval analyses. The associated generators are constructed via a *Matrix* data type supporting *Ring* opera-

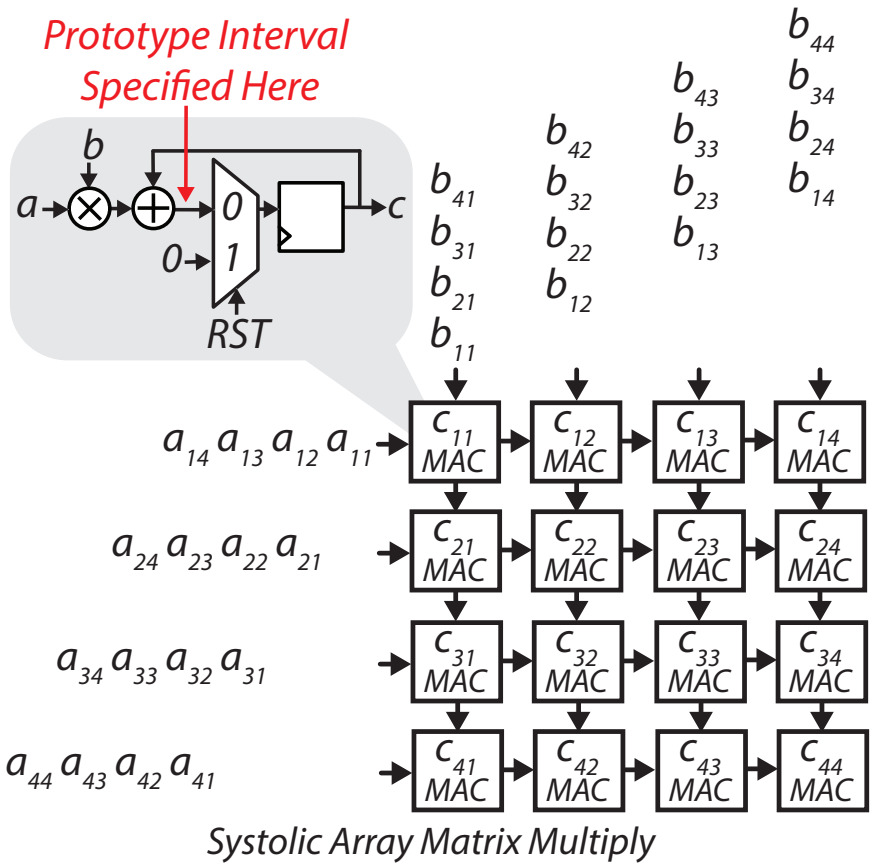

Figure 3.9: Systolic array MatMul. Prototype *Intervals* from loop unrolling must be supplied.

tions like $+$, $-$, and $\times$. $n \times n$ matrices are recursively partitioned into 4 new block matrices, and either a standard $\mathcal{O}(n^3)$ algorithm or Strassen's $\mathcal{O}(n^{2.81})$ algorithm [74] is used to create the fully parallel data-flow graph. A hardware template for more typical systolic array-based MatMuls that rely on iterative multiply-accumulate (MAC) operations has also been created. Systolic implementations are much more compact (e.g., the 8-bit 8×8 systolic DCT using Chisel *FixedPoint* has 16.4% of the area of an equivalent DCT implemented using Strassen's algorithm), but require more compute cycles. A prototype *Interval* used for range analysis is specified at the MAC's adder output with the equivalent (unsynthesized) sum of products in unrolled form.

Hardware instances are generated for 8×8 MatMuls, and *Interval* ranges at fixed input bitwidths are swept. As an example, the smallest *Interval* needing 8 bits is $[-65, 65)$—where the MSB is relatively underutilized, and savings accumulate more quickly—while the largest (full-width *Interval*) is $[-128, 128)$. For DCT hardware, the MatMuls are configured so matrix $\boldsymbol{A}$ contains constant DCT coefficients, and 1-D DCTs are calculated for the columns of $\boldsymbol{B}$. The generated instances are synthesized with a commercial 16nm FinFET technology

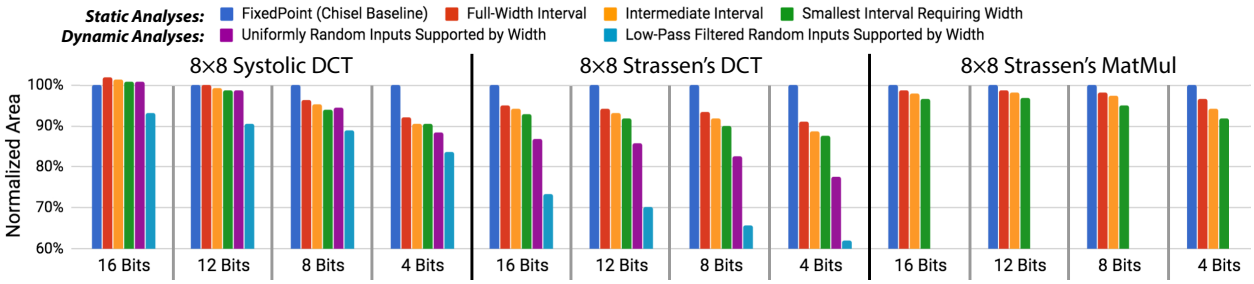

Figure 3.10: Area comparison using *FixedPoint* bitwidth propagation (Chisel baseline) vs. ACED *Interval* range optimization (with equivalent input bitwidths). Static and dynamic range analyses can automatically reduce area down to 62%, with zero (static) or minimal (dynamic) performance penalties using representative test vectors and input conditions. Constant coefficients are built into DCT-specific MatMul circuits. Area is calculated post-synthesis and includes the cell + estimated net area.

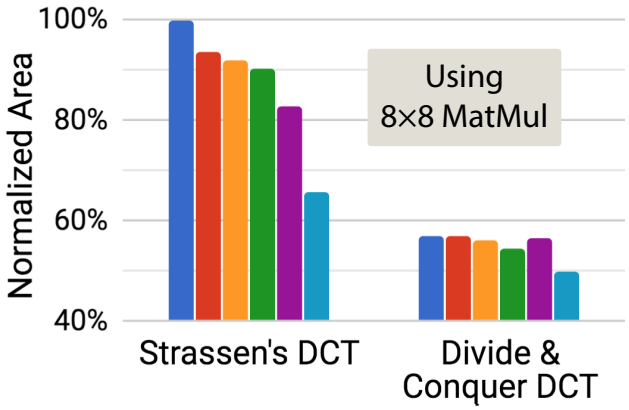

Figure 3.11: Relative areas for 8-bit divide + conquer & Strassen's MatMul DCTs.

at 500MHz (SS process corner, 0.72V supply, 125°C, and cworst_CCworst RC corner with Cadence Genus) to observe how FIRRTL range optimizations affect the QoR of the final design.

Static range analyses promise the most benefit over baseline *FixedPoint* designs relying on width inference for small input bitwidths (9% smaller area for full-width 4-bit input *Intervals* with DCT using Strassen's algorithm); the number of bits trimmed makes up a larger fraction of bits in the Chisel baseline design. Additionally, the DCT examples show that static range analysis performs better with specific input information, e.g., that elements of $\boldsymbol{A}$ are fixed values (Fig. 3.10). However, even when only full-width input *Intervals* are used, range analysis provides some area savings. The algorithms are easily compared: Fig. 3.11 shows that range analysis benefits the standard divide-and-conquer MatMul less, but the standard algorithm is more area efficient than Strassen's algorithm for 8×8 matrices. Unfortunately,

the synthesized area is larger for a systolic DCT implementation with 16-bit inputs when the full-width *Interval* type is used (vs. *FixedPoint*); bitwidth trimming using range analysis removes some design regularity that synthesis tools use for deep optimizations across PEs. Finally, ACED static range optimization times are compared with Chisel width inference times in Table 3.2. Recursive implementations using range analyses require significantly longer optimization and constant propagation times.

8×8 Design	16b Fix.	16b Intvl.
Systolic DCT	1.361	2.597
Strass. DCT	4.004	19.020
Strass. MatMul	4.065	17.104

Table 3.2: Static range optimization times (s) for designs using 16-bit *FixedPoint* and full-width *Interval* inputs.

User intent is embedded into the design via dynamic range analysis. Test vectors spanning the input range are generated from a uniform random distribution, and bitwidth optimization is performed, leading to significant area savings. In particular, properly capturing the system context in simulation (e.g., when the inputs are first low-pass filtered) leads to greater dynamic analysis savings—up to 38% in Fig. 3.10.

MatMul Generator Design	Slice LUTs	DSPs
Matlab HDL Coder (2017B)	41528	0
C++ Vivado HLS (2017.4)	18364	196
ACED (Smallest Full-Width *Interval*)	22924	328

Table 3.3: Synthesis results for ACED, HLS, and Matlab HDL Coder (2017B) outputs using Vivado 2017.4. 8-bit 8x8 Strassen's matrix multiplies were synthesized with 10.0ns 50% duty cycle clocks for the ZC706 FPGA. Matlab used dynamic optimization with uniform random matrices as inputs. ACED used static range optimization.

To evaluate ACED, a similar Strassen MatMul has been implemented using Matlab HDL Coder and C++ with Vivado HLS. Synthesis results are summarized in Table 3.3. It should be noted that the Vivado HLS results required extensive refactoring and heavy use of pragmas—without which the designs had poor throughput and high resource utilization—to achieve reasonable QoR. The Chisel designs were generated for ASICs and reused here without any tuning.

3.5 ACED Summary

Extensibility is important to a useful library. ACED operates at several different levels of abstraction, from high level type-generic generators to low-level bitwidth optimizations, but

these abstractions are kept separate to facilitate extensibility. For example, static bitwidth optimization and *Intervals* can be used on any design (e.g., a processor) and do not make assumptions about being run with the DSP-centric types. Bitwidth optimization techniques are directly usable without needing to modify the compiler, enabling the generator designer to focus more on rapid algorithm to hardware translation and less on tedious bitwidth optimization/decimal point alignment, etc., while still achieving a design with good QoR.

The ACED DSP library is a straightforward enhancement to Chisel. We have shown how the ACED hardware library can be used for generating DSP systems. It supports zero-cost abstraction and high-level data types, "free" optimization and specialization via open-source compiler passes, and a unified systems modeling framework. *Interval* bitwidth reduction is analyzed closely, and the ACED library has been successfully used to create DSP hardware blocks including FFTs, CORDICs, and FIR filters for various tapeouts.

ACED brings tried-and-true techniques immediately into the hands of the designer, making them accessible for everyday use. It is important to emphasize that our contribution is not just the particular bitwidth optimization techniques, but rather their implementation in an open compiler framework and a discussion of the value of these techniques in the context of writing generators. Other optimization techniques can be added to the compiler chain in a way that is transparent to the designer of a top level generator.

Chapter 4

A Real-Time, Analog/Digital Co-Designed 1.89-GHz Bandwidth, 175-kHz Resolution Sparse Spectral Analysis RISC-V SoC in 16-nm FinFET

4.1 Introduction

Today, different kinds of wireless systems are typically assigned to fixed, exclusively licensed bands. This nearly century old scheme prevents, for example, TV from interfering with radio. However, as wireless demands increase, preventing interference simply by way of rationing spectrum becomes intractable. First, this methodology cannot adapt to real-time supply and demand. At any given time, many bands are unused; typically, a wide swath of bandwidth might only be 5% occupied. However, other bands are completely saturated. Additionally, many modern-day wireless systems do not have fixed system operation—meaning they hop from frequency to frequency periodically, and bands might only be occasionally occupied.

Real-time GHz spectrum sensing ideally enables dynamic spectrum access. Secondary radios can, in theory, search for spectral holes (as in Fig. 4.1) to occupy, increasing spectrum efficiency. In practice, it is difficult to discern a weak user from noise without prior knowledge of the transmission scheme, à la matched filtering. The percentage of false negatives is an important metric for spectrum sensing success, since it is extremely undesirable to transmit over other users. A simpler task for spectrum sensing with FFT-based energy detection is interference avoidance, where large interfering signals are detected and "notched out." It requires no a-priori knowledge of signal types, but provides the feedback mechanism necessary so that tunable filters may adapt for blocker suppression.

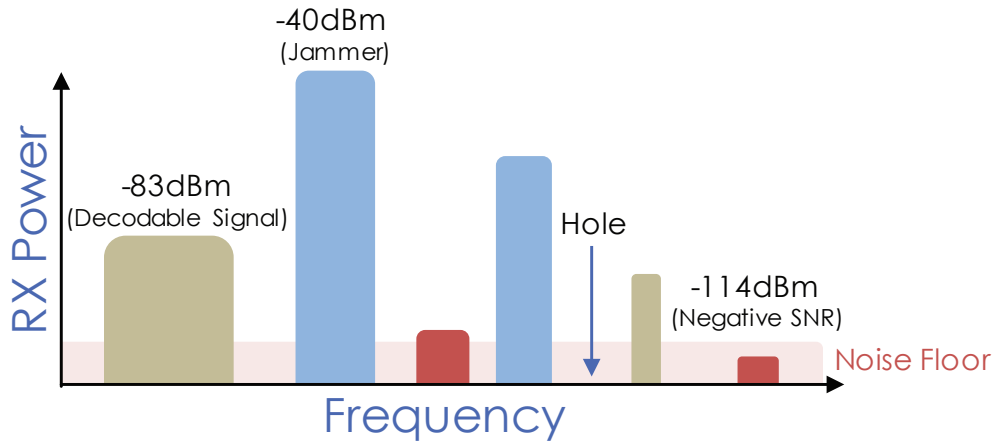

Figure 4.1: Cartoon representation of the RF spectrum.

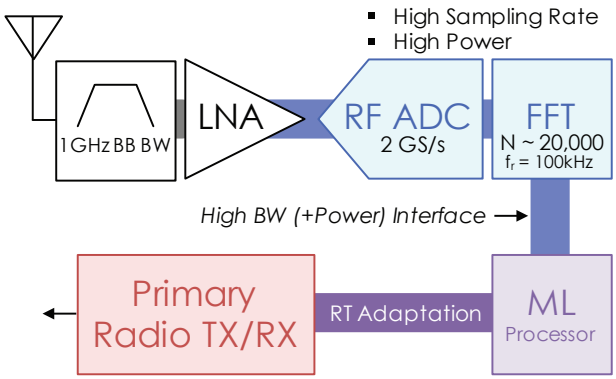

Figure 4.2: Nyquist-rate wideband sensing requires a high-sample-rate and, therefore, high-power ADC. Because the $n = 21,600$ FFT (with computational complexity $\mathcal{O}(n \log n)$) outputs a significant amount of raw, real-time data, the interface between it and any subsequent processor must be high bandwidth and high power.

Like RADAR and radio spectrometry, wideband spectrum sensing and analysis for unknown signal detection and adaptation typically requires either a high sample rate, high-power ADC (Fig. 4.2) or scanning and tuning a narrowband filter across a wide frequency range. When narrow-band frontends are used, typically only tens of MHz are acquired at a time, and sequentially scanning a GHz-wide bandwidth means that bands are monitored over a small fraction of time [26]. It is easy to miss short-lived signals, and real-time operation—so that adaptation can occur when a signal is actually present—becomes challenging. In practice, many applications

- *Deal with signals that are sparse in the frequency domain,*

- *Greatly benefit from data compression early in the processing chain*, and
- *Do not require perfect spectrum reconstruction, tolerating probabilistic errors on the order of a percent.*

Recently, compressed sensing techniques [75], [76] have been used to reduce the ADC sampling rate and analog power, but the complexity and overhead of hardware reconstruction has not been adequately addressed [77]. Additionally, compressed sensing typically requires analog mixing at GHz speeds [26], which, given stringent requirements on jitter, etc., results in comparable power requirements to high-speed ADCs. An on-chip, all-digital sparse FFT accelerator [23] reconstructs signals sparse in frequency, but targets a sparsity of only 0.1% and does not address algorithm degradation at low SNRs, making it impractical for use in real applications. We demonstrate an analog/digital co-designed sparse spectral analysis SoC supporting real-time signal detection for frequency spectra with sparsities up to 3.2% and input SNRs down to 9.7dB (Fig. 4.37) with a 17.5μs signal acquisition + analysis time. This is a significant improvement over commercial spectrum monitors like RFeye or off-the-shelf USRPs, which have scan times in the range of tens of milliseconds for significantly smaller observation bandwidths [26]. The output data is compressed to 4.4%.

The premise of this chip resembles that of MIT's BigBand [26]. Low-speed ADCs are used for uniform subsampling, saving power. Mixed-radix FFTs, generated by the FFT generator described earlier, are required for spectrum reconstruction. Additionally, the chip is capable both of detecting occupied frequency bands and also decoding the signals within them, assuming a sufficiently sparse spectrum. Unlike BigBand, which uses off-the-shelf components, the ADCs and digital reconstruction backend are fully integrated on chip. Finally, this chip covers more than twice the bandwidth of BigBand (1.89 GHz vs. 0.9GHz).

4.2 FFAST Algorithm Overview

4.2.1 Noiseless FFAST

The FFAST algorithm relies on controlled aliasing via Chinese-Remainder-Theorem-guided (CRT) subsampling [78] to reduce the computational complexity of large FFTs and minimize data storage requirements, while still supporting a high probability of spectrum recovery for sufficiently sparse spectra. For a k sparse spectrum, meaning that only k frequency bins contain non-zero data, where $k << n$, the computational complexity is reduced from $\mathcal{O}(n \log n)$ to $\mathcal{O}(k \log k)$, corresponding to FFT computation on $\mathcal{O}(k)$ subsampled time-domain data.

The CRT can be used to show that a frequency bin $j \in [0, n)$ is uniquely mapped to subsampled frequency bins a_1, a_2 such that $j \equiv a_1 \bmod n_1$ and $j \equiv a_2 \bmod n_2$ for $n = n_1 n_2$, where n_1 and n_2 are pairwise coprime. Subsampling typically poses a problem, because signals at various frequency locations alias on top of each other and cannot be distinguished.

This is explained by the equation:

$$X_{n_i}[b] = \sum_{j \equiv b \bmod n_i} X[j]\,. \tag{4.1}$$

However, as shown in Fig. 4.4, when a spectrum is sufficiently sparse, most subsampled frequency bins will contain either noise (= zeroton) or only a signal from a single frequency (= singleton), and only a few bins will contain multiple aliased signals (= multiton).

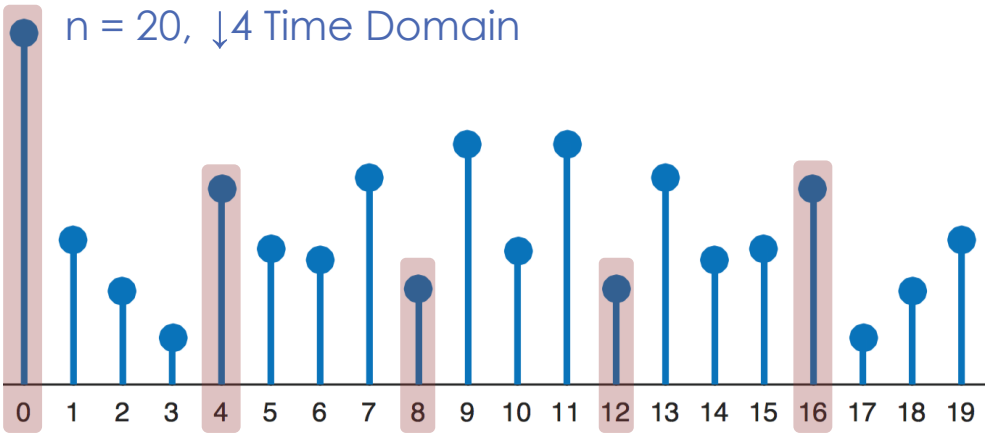

Figure 4.3: $n = 20$ time-domain sequence, subsampled by 4 in pink.

Consider the $n = 20$ time-domain sequence $\vec{x}$ shown in Fig. 4.3. Its frequency-domain representation $\vec{X}$ has sparsity $k = 5$, as indicated by Fig. 4.4, where the non-zero signals are $X[3]$, $X[7]$, $X[9]$, $X[12]$, and $X[14]$. As shown in Fig. 4.5, a frontend consisting of d subsampling stages collects time-domain samples, as guided by the CRT. $d = 3$ is typically chosen for reasonable degrees of sparsity, although more stages are required as k increases. The aliased frequency spectra of the subsampled data are calculated by n_1- and n_2-point FFTs. The toy example in Fig. 4.4 (with $d = 2$) illustrates the FFT outputs after subsampling by 5 (left) and 4 (right) respectively.

Assume that singleton bins can be distinguished from multiton bins and that the locations and values of signals in singleton bins can be easily identified. As these signal properties are iteratively discovered, they can be "peeled off" of the bipartite graph resulting from FFAST subsampling to recover additional signals (Fig. 4.4) [78]. The bipartite graph consists of "variable nodes," which are associated with the unknown non-zero frequency bins in the n-length $\vec{X}$. It also consists of "check nodes" with unresolved singleton and multiton bins containing aliased DFT coefficients. If, after subsampling, a particular bin in $\vec{X}$ (variable node) contributes to a subsampled singleton or multiton bin (check node), the corresponding variable node and check node are connected together. Singleton bins only have a single associated variable node, whereas multiton bins are connected to multiple variable nodes. The number of peeling iterations needed to resolve $\vec{X}$ is dependent on the input sparsity;

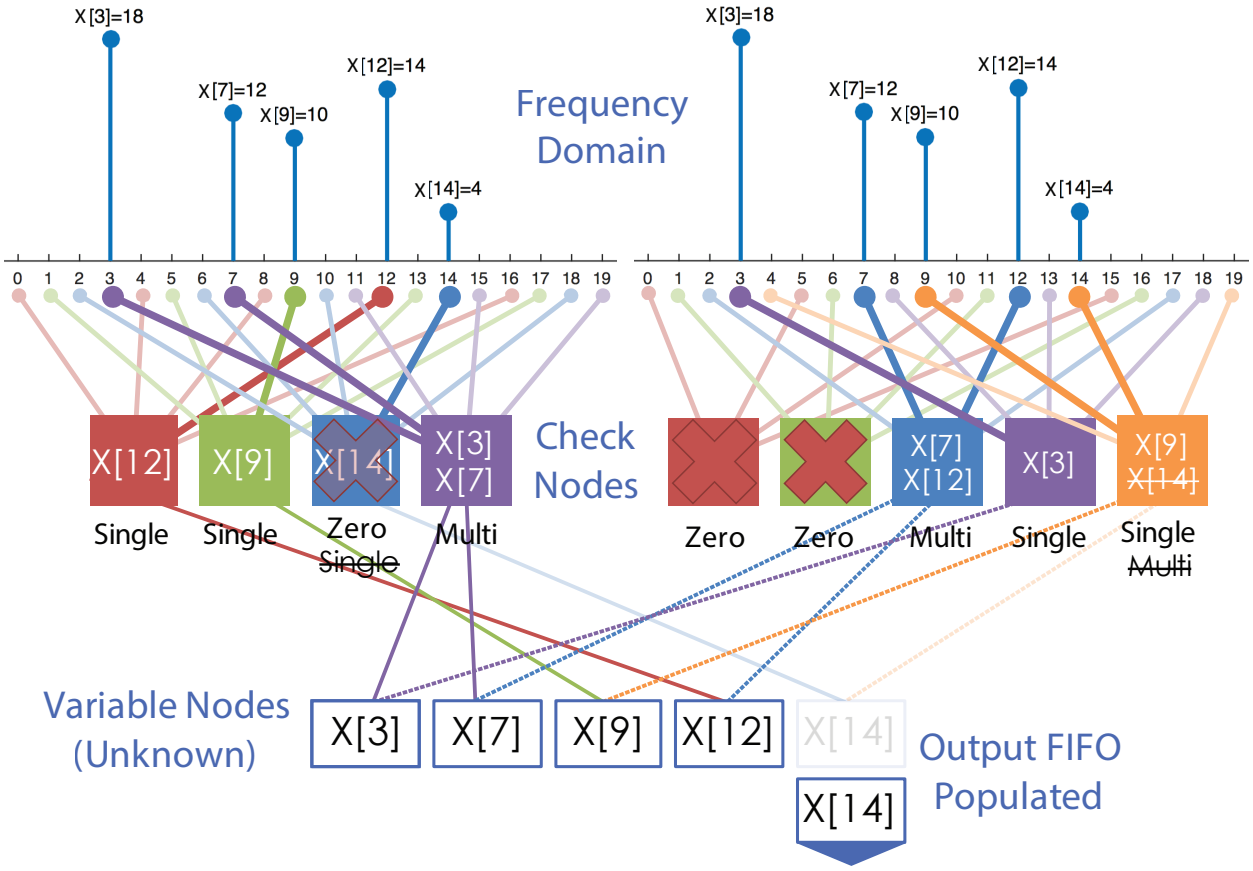

Figure 4.4: Signal recovery from subsampled inputs, where $n_1 = 4, n_2 = 5$ are coprime. The $j = 14$ signal recovered from the blue singleton bin in stage 1 is peeled off (i.e., subtracted) from the associated multiton bin in stage 2 (orange, determined by $j \equiv a_2 \bmod n_2$), such that the multiton bin becomes a recoverable singleton bin on the next peeling iteration. Zeroton bins, which can be determined via simple thresholding, are not included in the graph, because they are already known.

more iterations are needed as k increases, up until k is higher than supported by the chosen FFAST parameters. The process of "peeling" is known as sparse-graph decoding [79].

Table 4.1: Fourier relationships.

Time	Frequency
$x[a]$	$X[j]$
$x[a-\tau]$	$X[j]e^{-i\omega\tau}$

To differentiate between singleton and multiton bins, recall the relationship in Table 4.1. Here, $\omega = 2\pi j/N$, and the phase rotation is $\theta = \omega\tau$. Ignoring noise, if only a single

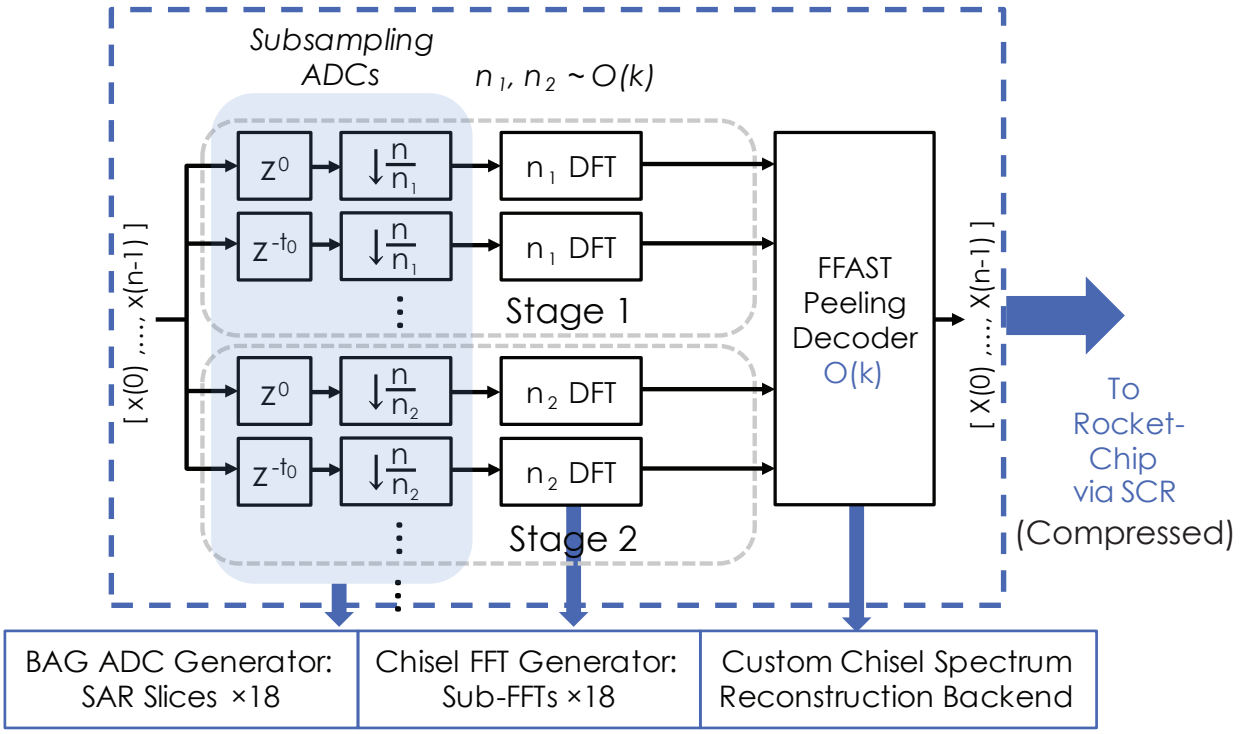

Figure 4.5: High-level FFAST architecture, consisting of frontend subsampling, sub-FFTs, and a peeling reconstruction backend.

tone is present in a subsampled frequency bin b, the magnitude at that bin should remain unchanged, regardless of whether the time-domain input was delayed or not prior to subsampling. This is not true in the case of multiton bins, since aliased frequency components rotate by different amounts before folding. Therefore, to resolve singleton bins from multiton bins, each subsampling stage must contain D delay chains. In the noiseless case, $D = 2$ can be used. The delay chains of a given subsampling stage use the same sampling period, but the input signal is circularly time-shifted by a fixed amount prior to subsampling, as shown in Figs. 4.5 and 4.6.

For the subsampled frequency bin b, $Y_{n_i,b,t}$ corresponds to the original input, and $Y_{n_i,b,t+1}$ corresponds to the delayed input. Therefore, an estimate of the phase rotation of a singleton bin can be determined via

$$\theta_{est} = \angle \left[Y_{n_i,b,t+1} \overline{Y_{n_i,b,t}} \right] . \tag{4.2}$$

From this, a location estimate can be calculated as

$$\jmath_{est} = \frac{\theta_{est} n}{2\pi\tau} . \tag{4.3}$$

To support the analysis of less sparse spectra, coprime subsampling factors 25, 27, and 32 (each associated with a computation stage) are chosen, decreasing the likelihood that

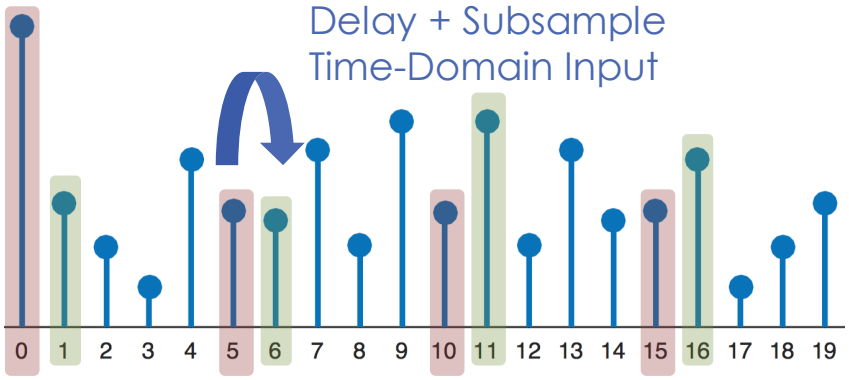

Figure 4.6: $n = 20$ time-domain sequence, subsampled by 5. $\tau = 1$ is used to disambiguate singletons from multitons.

a specific frequency collision in one stage occurs in other stages. This leads to "peeling-friendly" aliasing patterns that aid in multiton recovery.

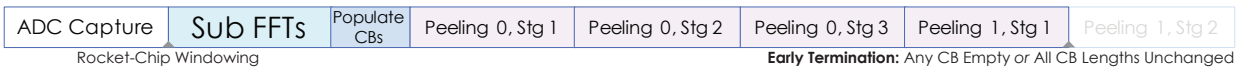

Figure 4.7: Signal acquisition and analysis stages.

A high-level sequence of FFAST analysis steps is shown in Fig. 4.7. For $n = 21,600$, $n_1' = n/25, n_2' = n/27, n_3' = n/32$ are not coprime (requiring a more general version of the CRT), but the number of recoverable signals is $\mathcal{O}(n_1', n_2', n_3')$.

4.2.2 Noise-Robust FFAST

4.2.2.1 Problem Setup

To improve the robustness of the FFAST algorithm when time-domain samples are corrupted by white Gaussian noise, i.e., $\vec{y} = \vec{x} + \vec{z}$ and $\vec{x}$ is the desired signal, instead of using *consecutive* shifts in the subsampling stages' delay chains, a greater number of pseudo-*random* delay shifts are used. This implies that the input is shifted by a random (but fixed, for ease of hardware implementation) amount prior to subsampling. Because of hardware limitations, redundant time samples are collected, making the sample complexity worse than the theoretical best-case in practice. Random delay shifts are used, so that, as in compressed sensing [80], the sensing matrix satisfies the restricted isometry property (RIP) [81] and has reasonable mutual incoherence [78]. Matrices characterized by RIP are nearly orthonormal when operating on sparse vectors.

Assume that the time-domain input in Fig. 4.3 is now corrupted by noise and $D = 4$ delay chains are used, corresponding to delays of 0, 1, 6, and 9. The set of observations

associated with the **zeroton** bin $Y_{n_i,b} = Y_{5,0}$ is given by

$$\vec{Y}_{5,0} = \begin{bmatrix} z_{5,0,0} \\ z_{5,0,1} \\ z_{5,0,6} \\ z_{5,0,9} \end{bmatrix} . \tag{4.4}$$

Here, the $z_{n_i,b,\tau}$ terms are uncorrelated and result from noise at higher frequencies folding down due to subsampling. The set of observations associated with the **singleton** bin $Y_{5,3}$ is given by

$$\vec{Y}_{5,3} = \begin{bmatrix} 1 \\ e^{-i2\pi(1\times3)/20} \\ e^{-i2\pi(6\times3)/20} \\ e^{-i2\pi(9\times3)/20} \end{bmatrix} X[3] + \begin{bmatrix} z_{5,3,0} \\ z_{5,3,1} \\ z_{5,3,6} \\ z_{5,3,9} \end{bmatrix} . \tag{4.5}$$

The set of observations associated with the **multiton** bin $Y_{5,4}$ containing two aliased tones is given by

$$\vec{Y}_{5,4} = \begin{bmatrix} 1 \\ e^{-i2\pi(1\times9)/20} \\ e^{-i2\pi(6\times9)/20} \\ e^{-i2\pi(9\times9)/20} \end{bmatrix} X[9] + \begin{bmatrix} 1 \\ e^{-i2\pi(1\times14)/20} \\ e^{-i2\pi(6\times14)/20} \\ e^{-i2\pi(9\times14)/20} \end{bmatrix} X[14] + \begin{bmatrix} z_{5,4,0} \\ z_{5,4,1} \\ z_{5,4,6} \\ z_{5,4,9} \end{bmatrix} \tag{4.6}$$

$$= \begin{bmatrix} \vec{a}(4) & \vec{a}(9) & \vec{a}(14) & \vec{a}(19) \end{bmatrix} \begin{bmatrix} X[4] \\ X[9] \\ X[14] \\ X[19] \end{bmatrix} + \begin{bmatrix} z_{5,4,0} \\ z_{5,4,1} \\ z_{5,4,6} \\ z_{5,4,9} \end{bmatrix} \tag{4.7}$$

$$= \boldsymbol{A}_{5,4}\vec{X} + \vec{z}_{5,4} . \tag{4.8}$$

$\boldsymbol{A}_{n_i,b} = \boldsymbol{A}_{5,4}$ is the bin measurement matrix associated with bin $b = 4$ of the $n_2 = 5$ subsampling stage. Its jth column is

$$\vec{a}(j) = \begin{cases} \begin{bmatrix} 1 \\ e^{-i2\pi(1\times j)/20} \\ e^{-i2\pi(6\times j)/20} \\ e^{-i2\pi(9\times j)/20} \end{bmatrix} & \text{if } j \equiv 4 \bmod 5 \\ \vec{0} & \text{otherwise} . \end{cases} \tag{4.9}$$

4.2.2.2 FFAST Peeling Decoding

The pseudo-code of the noise-robust FFAST algorithm to recover the non-zero $X[j]$'s from these observations is given in Algorithm 2. In an actual hardware implementation

Algorithm 2: Noise-robust FFAST algorithm [78] with early termination. Derived from [82].

Input : Noise-corrupted singleton and multiton bin observations $\vec{Y}_{n_i,b}$ stored in *check_nodes*$_i$ per subsampling stage i. n_i corresponds to the sub-FFT computed at stage i, and b is the subsampled bin.

Input : Noise threshold T_{noise}, below which a bin does not contain a signal, only noise.

Output: An estimate $\vec{X}$ of the n-point FFT compressed to $\mathcal{O}(k)$.

▷ Keep iterating if signal info was recovered on the previous peeling cycle.
while *length of any check_nodes*$_i$ *was updated* **do**
  for *each subsampling stage i* **do**
    for *each bin b in check_nodes*$_i$ **do**
      if $||\vec{Y}_{n_i,b}||^2 < T_{noise}$ **then**
        bin b is a *zeroton*
        remove b from *check_nodes*$_i$
      else
        ▷ v_j is the zero-delay signal estimate at j.
        (isSingleton, v_j, j) = **SingletonEstimator**($\vec{Y}_{n_i,b}$, T_{noise})
        if *isSingleton* **then**
          ▷ Peel off for all stages. Assumes all stages use the same delays.
          $\vec{Y}_{n_i,b} = \vec{0}$
          ▷ $j \equiv q \bmod n_l$
          if $||\vec{Y}_{n_l \neq n_i,q}||^2 < T_{noise}$ **then**
            $\vec{Y}_{n_l \neq n_i,q} = \vec{0}$
          else
            $\vec{Y}_{n_l \neq n_i,q} = \vec{Y}_{n_l \neq n_i,q} - v_j \vec{a}(j)$
          end
          add $X[j] = v_j$ to output
          remove b from *check_nodes*$_i$
        else
          ▷ No new information uncovered from bin b.
          bin b is a *multiton*
        end
      end
    end
    if *check_nodes*$_i$ *is empty* **then**
      ▷ Done!
      break out of *while loop*
    end
  end
end
return $\vec{X}$

$check\_nodes_i$ are implemented as $\mathcal{O}(k)$ circular buffers that keep track of non-zeroton bin locations. Observed bins b are peeled one-at-a-time, and early termination is implemented to minimize energy. Fig. 4.8 illustrates how this is accomplished with read and write pointers that remove singleton or zeroton bins from the circular buffers as they are discovered. Unresolvable multiton bins are replaced back into the circular buffers for a decoding attempt during the next peeling iteration.

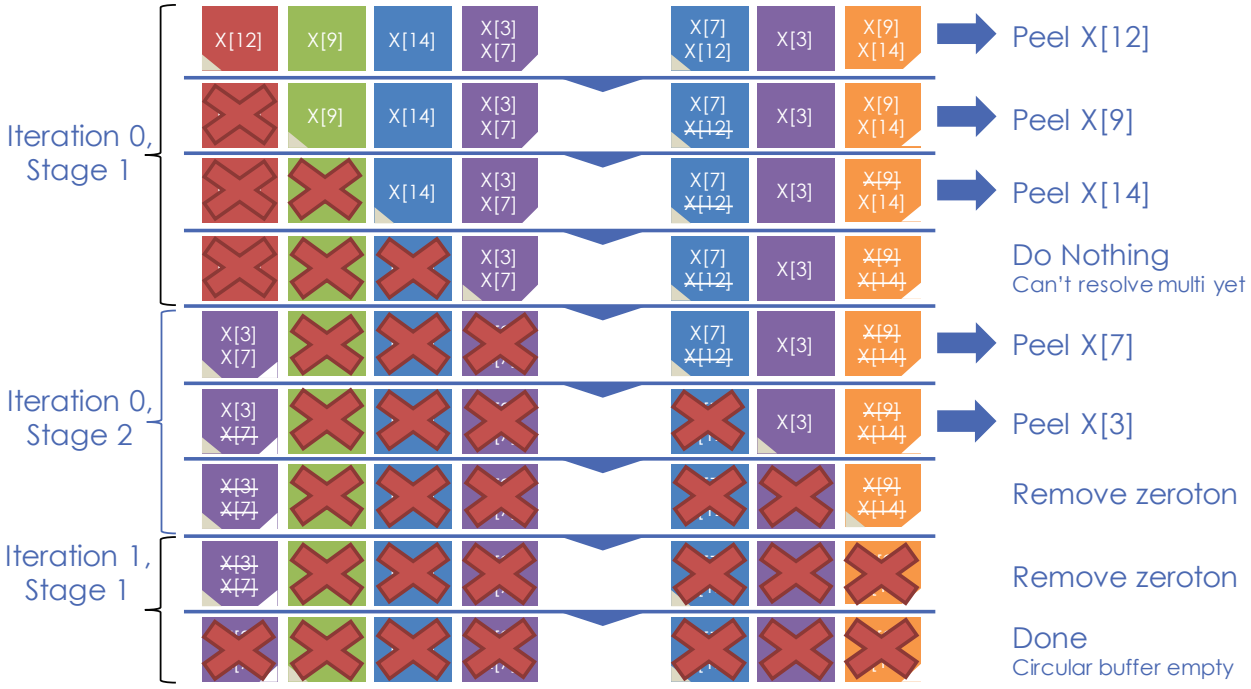

Figure 4.8: Peeling using circular buffers. Read pointers (brown) are updated for each b read. When a zeroton or singleton bin is discovered, b is removed from the corresponding circular buffer. Write pointers (white) are updated when subsampled bins are determined to be unresolvable multitons. In such a case, bin b is replaced back into the circular buffer.

The noise-robust FFAST algorithm is able to reconstruct a sparse frequency spectrum with probability of at least $1 - \mathcal{O}(1/k)$ using $\mathcal{O}(k \log^3 n)$ time-domain samples [78]. The noise-robust algorithm has a computational complexity of $\mathcal{O}(k \log^4 n)$.

4.2.2.3 Singleton Estimator

To understand the function of the singleton estimator, first consider N evenly spaced observations of a single complex sinusoid corrupted by white Gaussian noise:

$$y(t) = Ae^{i(\omega t+\phi)} + z(t)\,, \quad t = 0, 1, 2, ..., N-1\,, \tag{4.10}$$

where the amplitude A, frequency ω, and phase ϕ are fixed but unknown. When the input SNR is sufficiently high (e.g., 5-7dB) [83], the observations can be approximated as

$$y(t) \approx Ae^{i(\omega t+\phi+u(t))}\,, \tag{4.11}$$

where $u(t)$ is zero-mean white Gaussian noise. The phase of $y(t)$ is thus

$$\angle y(t) = \omega t + \phi + u(t)\,. \tag{4.12}$$

Assuming that only ω needs to be estimated, phase unwrapping can be avoided by considering [83]

$$\Delta(t) = \angle y(t+1) - \angle y(t)\,, \quad t = 0, 1, ..., N-2 \tag{4.13}$$

$$= \omega + u(t+1) - u(t)\,. \tag{4.14}$$

The minimum mean square error (MMSE) estimate, also known as Kay's estimate, of the unknown frequency ω is then given by [83]

$$\omega_{est} = \sum_{t=0}^{N-2} \beta(t) \angle \left[y(t+1)\overline{y(t)} \right]\,, \quad \beta(t) = \frac{3N/2}{N^2-1} \left\{ 1 - \left[\frac{t-(N/2-1)}{N/2} \right]^2 \right\}\,. \tag{4.15}$$

Note that when two samples delayed by unit time (whatever that is chosen to be) are used,

$$w_{est} = \angle \left[y(t+1)\overline{y(t)} \right]\,. \tag{4.16}$$

Because matching delays across subsampling frontends is extremely difficult in practice, and Kay's estimator requires evenly spaced time samples, Kay's estimator is applied only to two samples at a time, with (parameterizable) unit delays of $\tau_0 = 1$, $\tau_1 = 3$, and $\tau_2 = 7$.

As mentioned earlier, the total number of delay chains per stage is given by D. For noise robustness, D is split into C clusters, with each cluster consisting of $K = 2$ delay chains. Therefore, $D = 2C$. Singleton estimation occurs in two steps. First, Kay's estimator is used to obtain C estimates of ω_s, for $s = 0, ..., C-1$, from sets of two bin observations. The estimates are combined using successive refinement to obtain a final ω_{est} corresponding to the location of the (potential) singleton bin. The 0^{th} delay chain in each cluster delays the input by an amount d_s, a number pseudo-randomly chosen between 0 and $n-1$ to satisfy RIP. To use successive refinement, the rth delay chain in the sth cluster delays the input signal by $d_s + r2^s$ (where we define the unit delay used by Kay's estimator to be $\tau_s = 2^s$ and $r \in [0, 2)$ for $K = 2$) in typical software implementations. As s increases, larger powers of two are used. In our implementation, $d_0 = 0$, $d_1 = 6$, and $d_2 = 12$. However, to ease ADC layout, as described later, rather than using powers of two, $\tau_0 = 1$, $\tau_1 = 3$, and $\tau_2 = 7$ are used for Kay's estimator. Therefore, a total of $D = 6$ delay chains are used per stage. The

jth column of the bin measurement matrix is thus

$$\vec{a}(j) = \begin{cases} \begin{bmatrix} e^{-i2\pi((0+0)\times j)/n} \\ e^{-i2\pi((0+1)\times j)/n} \\ e^{-i2\pi((6+0)\times j)/n} \\ e^{-i2\pi((6+3)\times j)/n} \\ e^{-i2\pi((12+0)\times j)/n} \\ e^{-i2\pi((12+7)\times j)/n} \end{bmatrix} & \text{if } j \equiv b \bmod n_i \\ \vec{0} & \text{otherwise}\,. \end{cases} \tag{4.17}$$

Kay's estimator is used for processing observations in cluster s:

$$\theta_s = \tau_s\omega + \delta\,, \tag{4.18}$$

where δ represents the noise term. When $\tau_s = 1$, an estimate of ω can be directly obtained. However, in the presence of noise, the range around θ_s containing the true $\tau_s\omega$ with high probability is given by

$$\Omega_s = \left(\theta_s - \frac{\pi}{c_1}, \theta_s + \frac{\pi}{c_1}\right) \tag{4.19}$$

for some constant c_1. Here, the length of the interval is denoted as $|\Omega_s| = 2\pi/c_1$. To illustrate how successive refinement works, take $\tau_s = 2^s$ (Fig. 4.9). Because $\langle 2\omega\rangle_{2\pi} = \langle 2(\omega+\pi)\rangle_{2\pi}$, ω *and* $\omega+\pi$ map to the same frequency after multiplication by two. Therefore, for sufficiently large c_1, $|\Omega_0 \cap \Omega_1/2| \leq 2\pi/(2c_1)$. As more delay clusters are added to C, the refined estimate of ω is isolated to [78]

$$\left|\cap_{s=0}^{C-1}\Omega_s/2^s\right| \leq \frac{2\pi}{2^{C-1}c_1}\,. \tag{4.20}$$

This corresponds to the region of red/blue overlap in Fig. 4.9. Similar reasoning applies when τ_s's are not powers of two, as long as c_1 is sufficiently small. Hardware to perform successive refinement and obtain an estimate of the signal location j_{est} is described in Section 4.3.3 and illustrated in Fig. 4.20.

Once j_{est} is found for a given stage i and bin b, a signal estimate is obtained by rotating the delayed signals back and averaging i.e.,

$$v_j = \vec{Y}_{n_i,b}\overline{\vec{a}(j_{est})}/D\,. \tag{4.21}$$

Finally, the bin contains a singleton if

$$||\vec{Y}_{n_i,b} - v_j\vec{a}(j_{est})||^2 < T_{noise}\,. \tag{4.22}$$

Otherwise, it is a multiton, because the location estimate does not justify the actual bin observations.

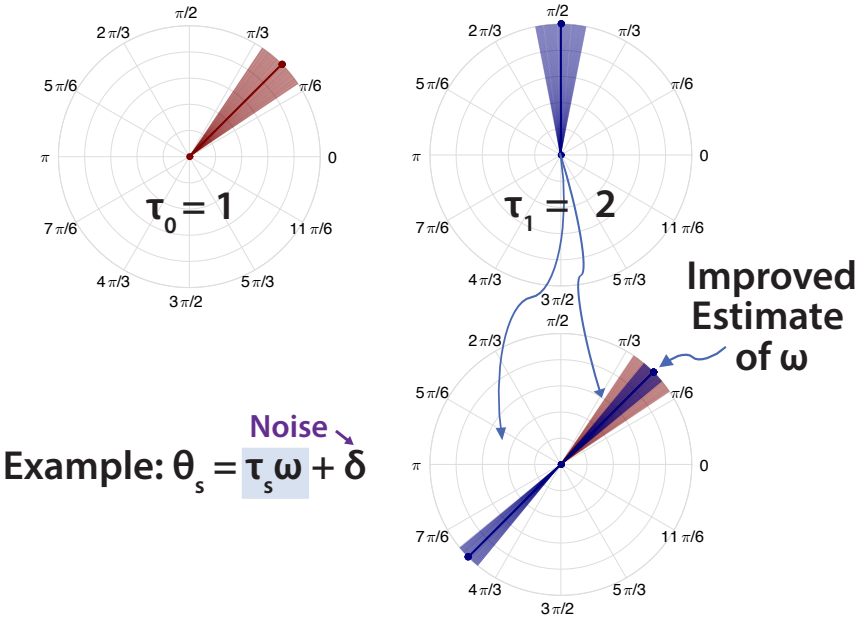

Figure 4.9: Successive refinement of the ω estimate for $\tau_s = 2^s$ [84].

4.2.2.4 Theoretical Limits via Simulation Results

When the FFAST architecture is fixed, the designer must provision for worst-case sparsity. To support a more realistic sparsity of ∼3.6% for $n = 21,600$, $n_1' = 2^5 \times 3^3 = 864$, $n_2' = 2^5 \times 5^2 = 800$, and $n_3' = 3^3 \times 5^2 = 675$ are used. A Matlab implementation of FFAST using floating point numbers was used to determine upper bounds on algorithm performance. The results are shown in Figs. 4.10 and 4.11. The percentage of false negatives remains at a reasonable level for input sparsities below 8.5%. Above 8.5% sparsity, the percentage of false negatives dramatically increases, due to the FFAST architecture's inability to further resolve multiton bins. The subsampling factor can be reduced to improve this metric, at greater hardware cost. Additionally, as additional tones are introduced into the spectrum, the number of peeling iterations required to decode the frequency spectrum increases. However, above the ∼ 8.5% sparsity "cliff," the number of peeling iterations rapidly drops off, because the algorithm has been written to terminate early when no further multiton bins can be resolved.

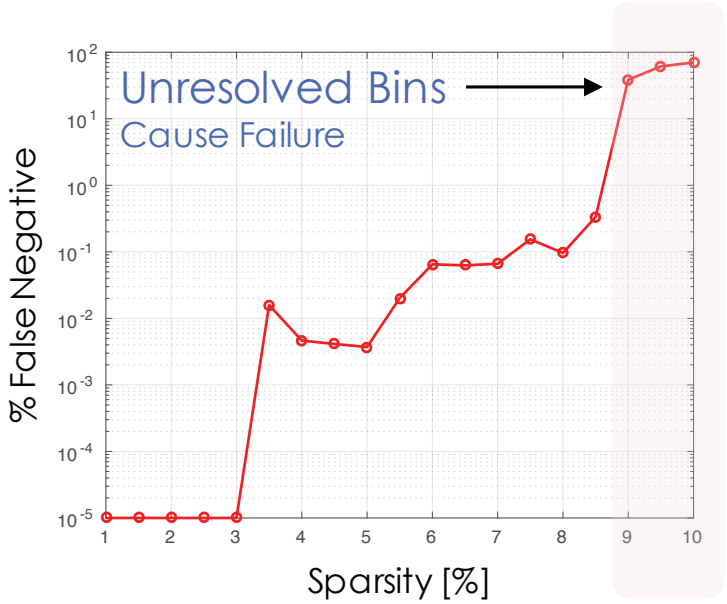

Figure 4.10: Percentage of false negatives vs. input sparsity.

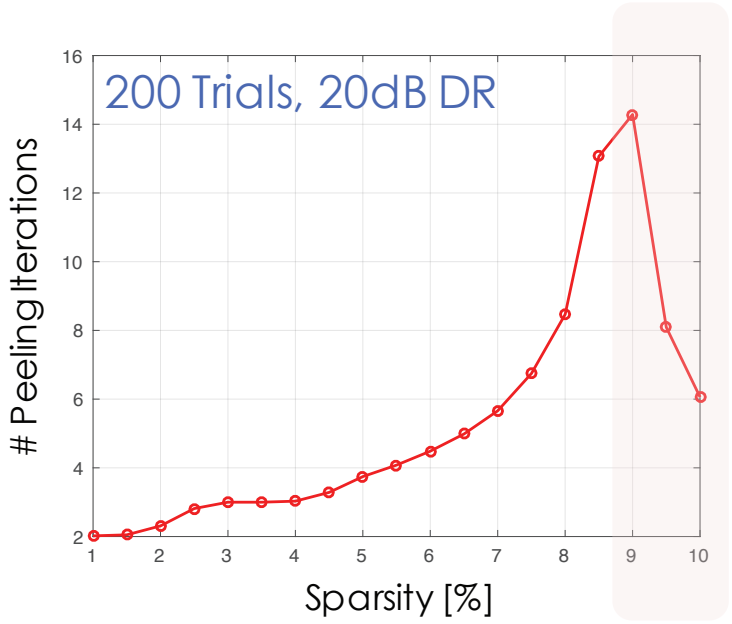

Figure 4.11: Peeling iterations required to decode the frequency spectrum as a function of input sparsity.

4.3 Implementation Details

4.3.1 Analog Frontend

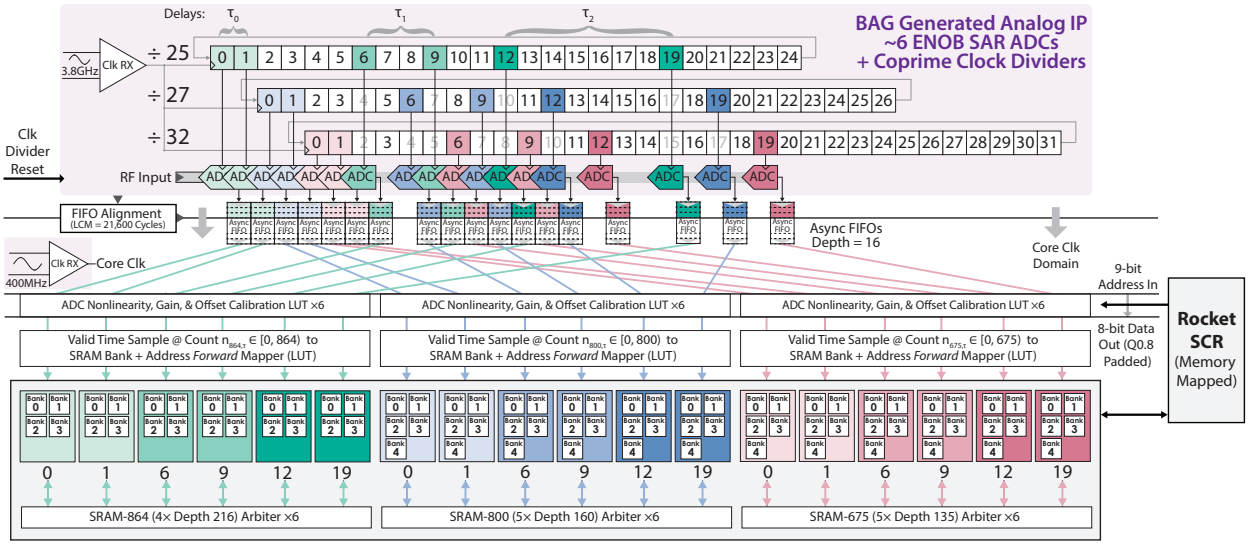

Figure 4.12: Analog frontend with shift-register-based clock dividers that generate (non-50% duty cycle) 151.2MHz, 140MHz, and 118.125MHz subsampling clocks with unit delays of 0, 1, 6, 9, 12, and 19 from a 3.78GHz source. Data from associated 9-wire, < radix-2, SAR ADC slices are time-synchronized and stored in memory using 8 bits, following capacitor mismatch, offset, and gain correction.

In the analog frontend illustrated in Fig. 4.12, a flip-flop-based clock divider generates 3 different clock frequencies for CRT-guided subsampling with 6 different phase shifts each for singleton/multiton disambiguation and noise resilience from a 3.78GHz input. The clock frequencies have coprime ÷25, ÷27, and ÷32 relationships with the input frequency. The τ_0, τ_1, and τ_2 delay deltas are used to create the three delay clusters (themselves containing two delays) used for the successive approximation procedure in singleton estimation. These clocks are used to run 18 lanes of asynchronous, constant V_{CM}-switched SAR ADC slices generated using [32] (Fig. 4.13). The ADC has a 9-wire output, where < radix-2 is used for the 3 MSBs to provide redundancy against missing decision levels, which can only be corrected in the analog domain with capacitor tuning. The resultant missing output codes (corresponding to the discontinuities in the mean input vs. raw ADC code plot in Fig. 4.15 and where the number of code occurrences is zero for a particular code in Fig. 4.14) are corrected digitally by re-weighing the output bits [85]. The coefficients that maximize the SNDR, determined via a least-squares fit to a known sinusoidal input, are used to generate calibration LUT parameters so that mismatch corrections (in addition to gain/offset corrections) can be done on-the-fly before 8-bit data is stored into memory. The 9-bit raw ADC outputs are used to

address the LUTs, and 8-bit remapped outputs with the improved mean input vs. calibrated ADC code characteristics shown in Fig. 4.16 are output.

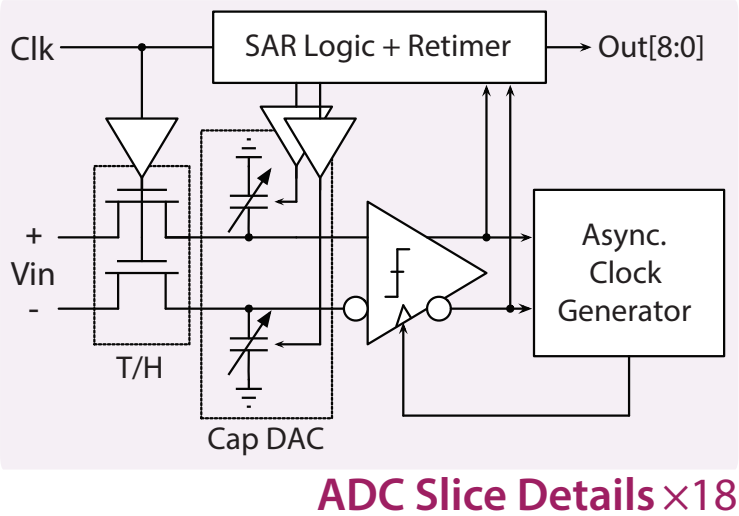

Figure 4.13: Asynchronous SAR ADC slice [86].

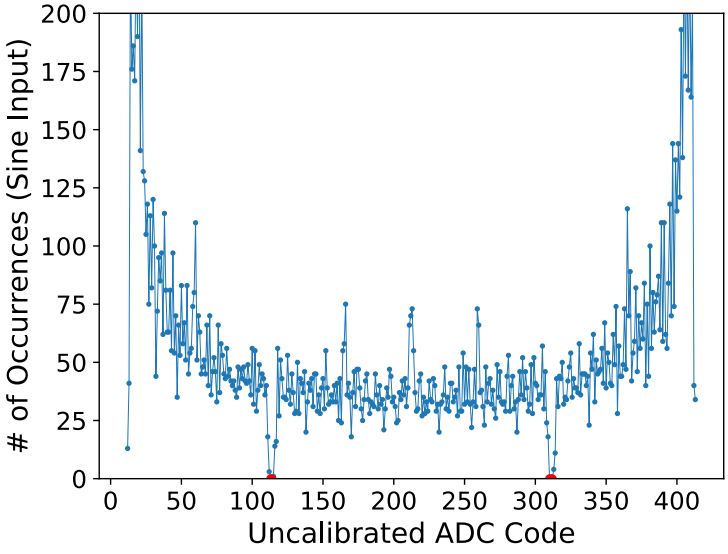

Figure 4.14: Missing raw ADC output codes due to using < radix-2 for the 3 MSBs.

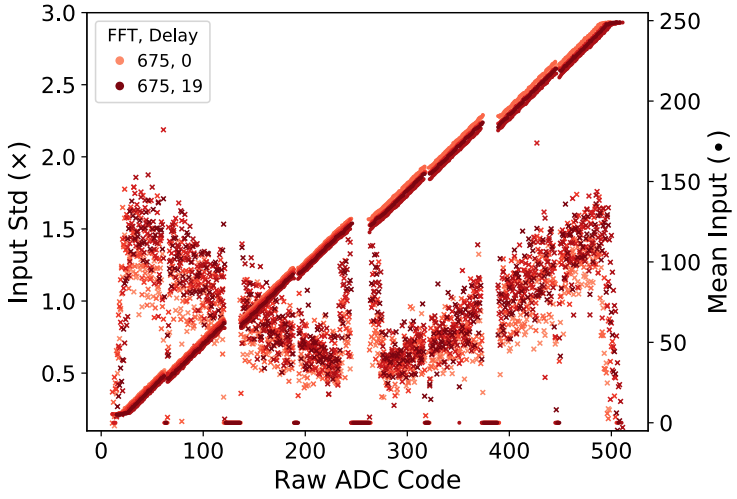

Figure 4.15: Fitted input (and its standard deviation) vs. *raw* ADC code. Missing codes are due to < radix-2 capacitor weights. Gain and offset mismatches across different ADC slices are evident. A 0.875MHz sine input and a 3.78GHz input clock were used for calibration.

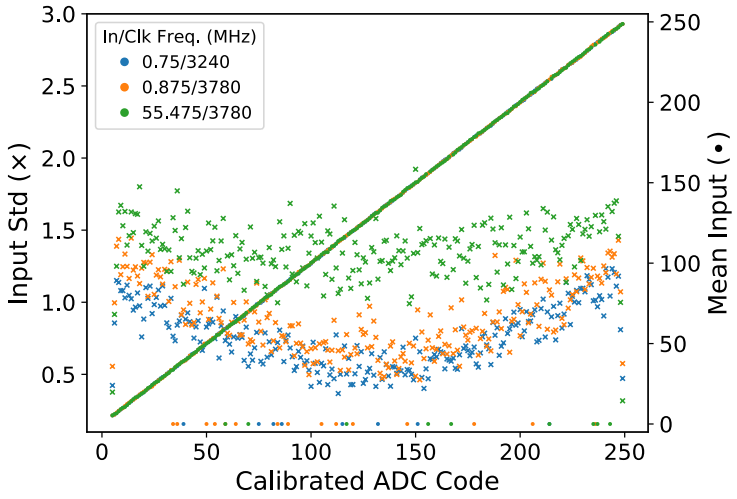

Figure 4.16: Fitted input (and its standard deviation) vs. *calibrated* ADC code after the LUT. The post-processed data is 8 bits wide, and missing codes are mostly eliminated.

Because the divided clocks are aligned once every 21,600 cycles, an alignment circuit (Fig. 4.17) generates appropriate valid signals used by asynchronous FIFOs at the eighteen subsampling-clock-to-core-clock boundaries. The digital core clock runs at 400MHz, regardless of whether the ADC clock input is at 3.24GHz or 3.78GHz, necessitating the asynchronous FIFOs. The state of the alignment circuit stabilizes some time after the initial

ADC clock reset without regard for the DSP state, so handshake bits are used to ensure that downstream processing is able to get properly aligned frames of data, which are mapped to SRAM banks and addresses via a LUT version of the index vector generator described in Chapter 2.

Overall, due to delay redundancy added for noise robustness (i.e., the 6 phase shifts), the ADCs sample 35% fewer time-domain samples compared to a 3.78Gs/s full-rate ADC. Note that there is a trade-off between number of added delays for noise robustness and performance degradation due to mismatches in bandwidth, etc. from PVT (and especially spatial locality) variations across the ADC lanes. PVT denotes process, voltage, and temperature.

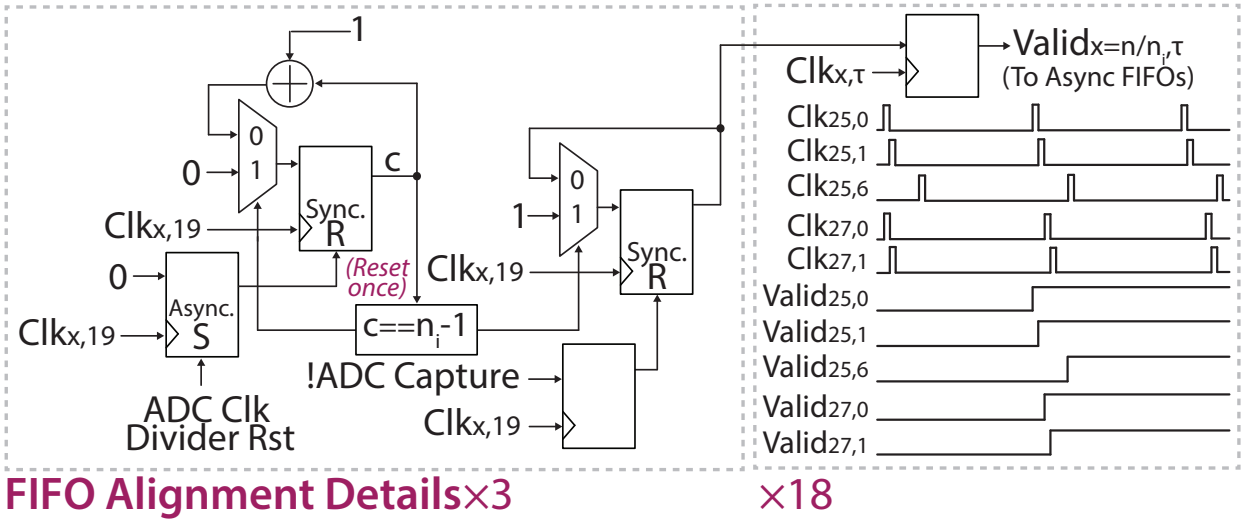

Figure 4.17: ADC-to-core FIFO alignment circuit (aligned every 21,600 ADC clock cycles).

4.3.2 Sub-FFTs

The subsampled ADC outputs are fed into complex (20, 20)-bit, *mixed-radix*, 864-, 800-, and 675-point FFTs, corresponding to n_1', n_2', and n_3', respectively. This requires 70.2kB of memory. The decimation-in-frequency, non-2^n sub-FFTs are generated via the Chisel [27] FFT generator [28] described earlier. As shown in Fig. 4.18, each sub-FFT utilizes a single-PE, conflict-free memory access scheme with runtime-configurable butterfly units that reallocate hardware blocks to support multiple radices (e.g., 4, 2, and 3 for the 864-point FFT). Control logic is shared between the six lanes associated with a subsampling stage, and memory banks are calculated via mixed-radix counters and simple subtract/mux-based modulo units. Finally, depending on the current FFAST analysis stage, FFT input/output unscrambling (i.e., the index-to-memory-bank/address mapping) is done by muxing between LUTs whose values are associated with in-order inputs and outputs indexed in quasi-digit-reversed order, as described by the FFT generator. The fixed-point representations of the complex outputs are normalized by the sub-FFT size and reinterpreted (so that the location

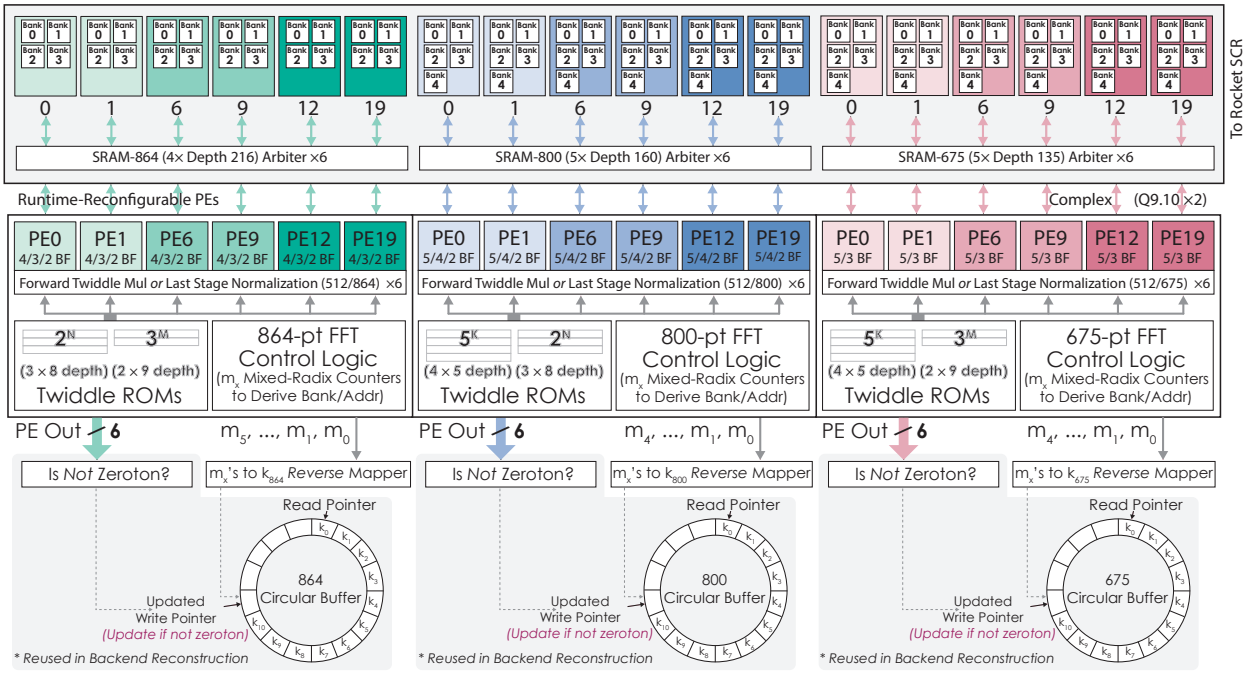

Figure 4.18: Sub-FFT DSP blocks with shared control logic and runtime-reconfigurable butterflies. Abusing notation, k_{n_i} corresponds to the current sub-FFT bin b, and the m in k_m represents the position that the bin is stored in the circular buffer.

of the binary point is changed, but the bitwidth stays the same). Therefore, rather than dividing by n_x for renormalization, the location of the binary point is shifted to the left by 9 places (corresponding to a divide by 512), and the output is instead multiplied by $512/n_x$. Normalization is necessary for digital reconstruction.

4.3.3 Singleton Estimation and Peeling Reconstruction

Bins that only contain noise are pre-determined via thresholding and discarded from future analysis. The locations of the remaining bins, which may either be singleton or multiton, are stored in per-sub-FFT circular buffers (Fig. 4.18). As described earlier, because signals at different frequencies do not rotate by the same amount when delayed ($\langle\theta\rangle_{2\pi} = 2\pi\langle j\tau\rangle_N/N$), delayed versions of the signal are collected to distinguish between singleton bins and multiton bins. As shown in Fig. 4.20, the phase differences θ_i between τ_i delayed versions of the subsampled signals are calculated via vectoring CORDICs. Note that the delays are chosen pseudo-randomly, but in a way to make ADC layout (specifically routing) more regular.

Successively larger τ_i are used to obtain better estimates of the θ associated with unit T, and, thus, the signal's frequency location, in the presence of uncorrelated noise between

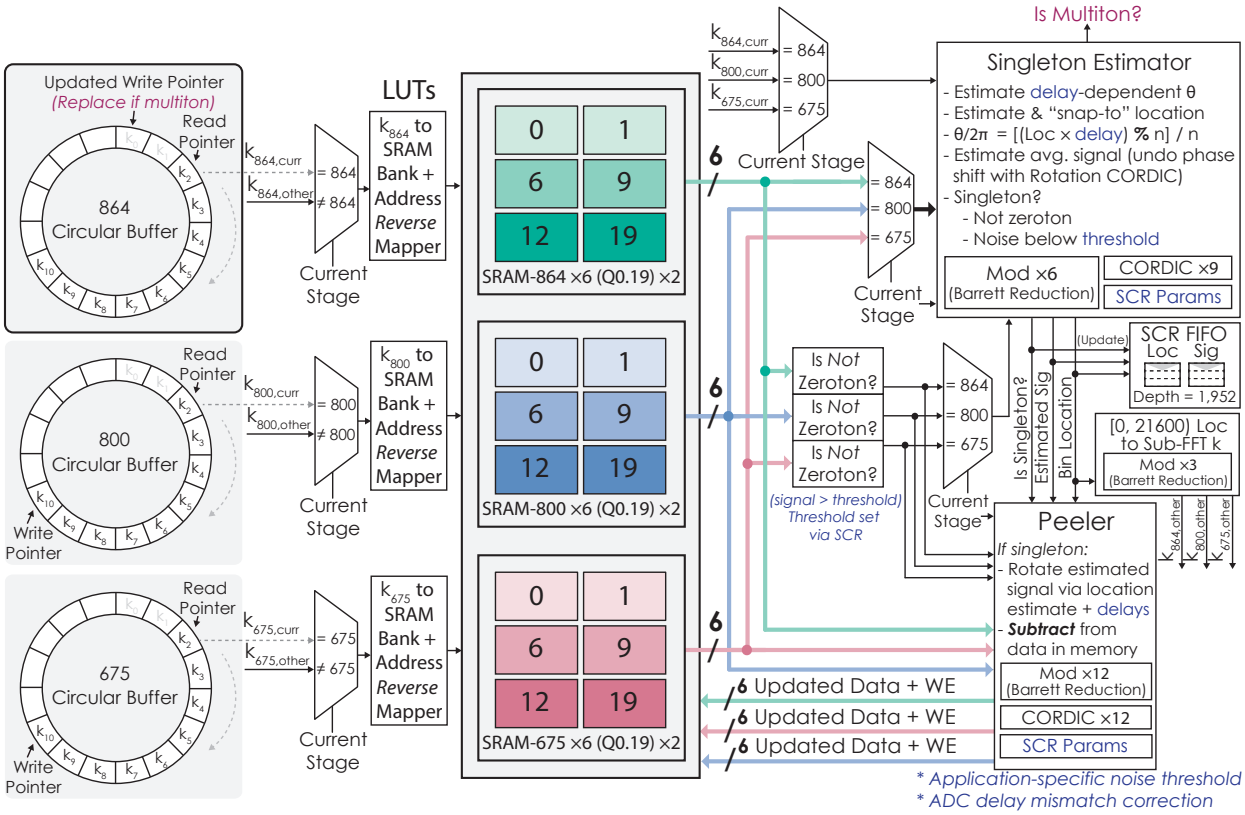

Figure 4.19: Peeling reconstruction backend with a singleton estimator. Abusing notation, k_{n_i} corresponds to the current sub-FFT bin b, and the m in k_m represents the position that the bin is stored in the circular buffer.

ADC channels. In software implementations of FFAST, the τ_i are usually chosen to be incrementally larger powers-of-two. However, to maintain layout regularity, the τ_i's increase at a faster rate. This is still a valid configuration if the angle uncertainty due to noise is sufficiently small. The frequency location of the signal is estimated and refined with knowledge of the subsampled bin location (i.e., snapped to a location, as shown in Fig. 4.20), and a signal estimate is obtained by undoing the delay-based phase rotations via another set of rotation-mode CORDICs and averaging (Fig. 4.19). Modulo operations are performed via the Barrett reduction algorithm [88]. Singletons are determined by a runtime-reconfigurable decision threshold.

The algorithm does not require timing calibration at the ADC outputs, because the (DC) lane-to-lane skew can be determined and corrected via adjustments to the delays and τ_i's stored in the Rocket processor's status & control registers (SCRs). A known sine input is fed into the system, and a sine fit is used to calculate the real delay offset given limited bandwidth mismatch between slices. The amount of bandwidth mismatch is estimated by observing the gain differences between the eighteen FFT outputs across the input bandwidth,

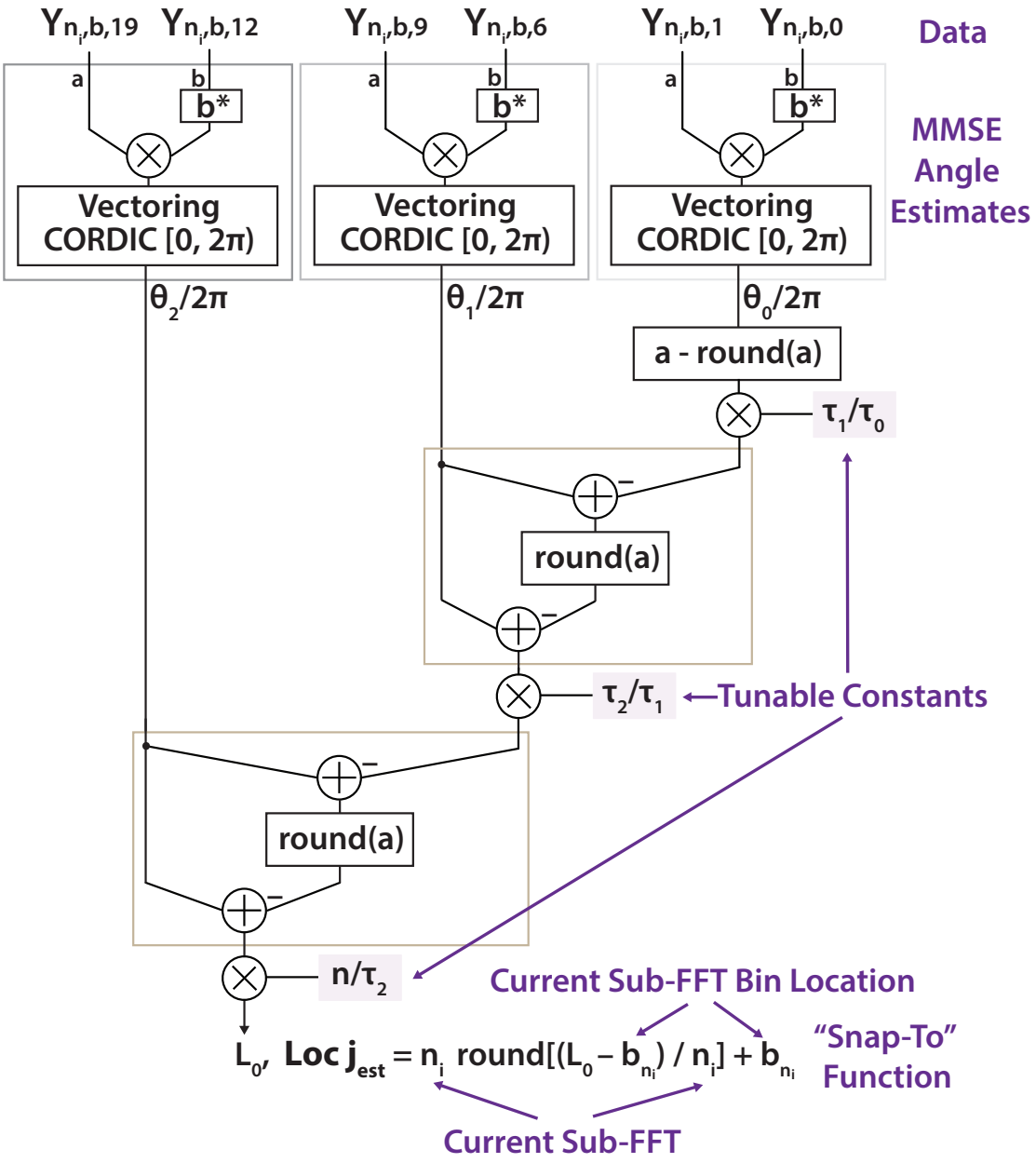

Figure 4.20: Successive approximation used to improve the signal location estimate from the angle deltas (obtained via CORDIC [87]) of delayed input samples.

as shown in Fig. 4.21. The relevant control registers are then updated with the estimated delay offsets shown in Fig. 4.22.

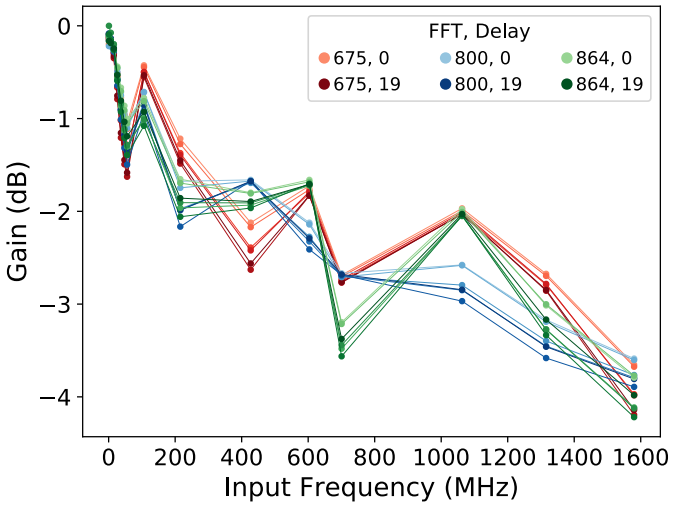

Figure 4.21: Gains across different ADC slices and input frequencies @ 3.78GHz ADC input clock. This plot gives a designer perspective on bandwidth mismatch between lanes. Gains are not monotonically decreasing with higher input frequency, since they are not completely determined by the input samplers' bandwidths, and signals above individual subsampling frequencies are folded down.

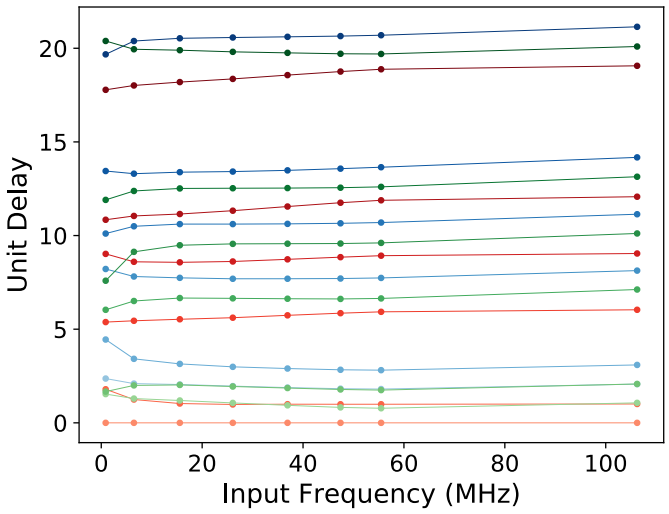

Figure 4.22: Actual delay offsets across different ADC slices and input frequencies @ 3.78GHz ADC input clock. Delay calibration is required, as DC lane-to-lane skew shifts the delays away from ideal. The analog layout does not need to be optimized to minimize skew, as the amount of skew can be determined after chip fabrication and delay-related control registers used by the singleton estimator can be updated from their nominal values.

As illustrated in Fig. 4.19, discovered singletons are iteratively peeled off by subtracting their contributions from bins associated with other sub-FFTs and removing their locations from the list of unresolved bins. If a bin is still unresolvable after singleton estimation, its location is placed back into the circular buffer list. This peeling process is terminated early when any circular buffer is empty or no new signal information is uncovered (i.e., when the lengths of all circular buffers have not changed after a peeling iteration). As singletons are found, their bin locations and signal values are output into a FIFO readable by the processor via SCR. Note that this data is not sorted. The frequency/signal data output is compressed to the same order of magnitude as the input sparsity (4.4% for an input spectrum with 3.2% sparsity, accounting for bin indices not required in the normal FFT case, which are 15 bits wide).

4.3.3.1 Calculating $x \bmod n$

Modulo operations are used extensively in the singleton estimator and peeling decoder. When $x \leq 2n - 1$,

$$x \bmod n = \text{Mux}(x - n \leq 0, x - n, x)\,. \tag{4.23}$$

However, if x is arbitrarily large, $x \bmod n$ can be calculated using Barrett reduction [88], as described in Algorithm 3. Here, the smallest k is chosen, given that the maximum allowable

Algorithm 3: Barrett reduction for computing $x \bmod n$.

Input : x, unsigned.
Input : n, unsigned and constant.
Output: $x \bmod n$

$m = \lfloor 2^k/n \rfloor$
$q = \lfloor xm/2^k \rfloor$
$r = x - qn$
if $n \leq r$ **then**
  return r - n
else
  return r
end

value of x is

$$x < \frac{1}{\frac{1}{n} - \frac{\lfloor 2^k/n \rfloor}{2^k}}\,. \tag{4.24}$$

4.3.3.2 CORDIC

CORDIC is an iterative algorithm—requiring only additions, subtractions, and shifts—used by the singleton estimator and peeling decoder to determine the phase difference be-

tween two delayed samples (in vectoring mode) and rotate signals for signal estimates (in rotation mode). Its computation converges to the desired value at a rate of 1 bit per iteration. n-bit signed angles used by CORDIC span the full $[-\pi, \pi)$ interval, although they can likewise be interpreted as unsigned $[0, 2\pi)$. This means the 4-bit value 0b1100 can be interpreted either as $-\pi/2$ or $3\pi/2$. Likewise, 0b1000 can be interpreted as either $-\pi$ or π and 0b0100 is $\pi/2$. The pseudo-code for CORDIC is found in Algorithm 4.

Algorithm 4: CORDIC (Coordinate Rotation Digital Computer) algorithm [87].

Input : x, n bits, signed.
Input : y, n bits, signed.
Input : θ, n bits. The signed interpretation is $\theta_s \in [-\pi, \pi)$. It is equivalently represented as unsigned $\theta_u \in [0, 2\pi)$.
Input : isRotation. If not, then CORDIC is in vectoring mode.
Output: x, y, θ

▷ CORDIC can rotate an input by a maximum of $\pm\pi/2$.
if *(isRotation* **and** $\pi/2 < \theta_u < 3\pi/2$*)* **or** *(***!***isRotation* **and** $x < 0$*)* **then**
  $x = -x$
  $y = -y$
  $\theta_s = \theta_s - \pi$
end
for i *in* $0 < i < numStages$ **do**
  if *(isRotation* **and** $\theta_s \geq 0$*)* **or** *(***!***isRotation* **and** $y < 0$*)* **then**
    $d = +1$
  else
    $d = -1$
  end
  $x' = x - 2^{-i} dy$
  $y = y + 2^{-i} dx$
  $x = x'$
  ▷ Angle constant needs to be properly normalized to bitwidth.
  $\theta_s = \theta_s - \text{round}(\frac{2^n \arctan(2^{-i})}{2\pi}) d$
end
return x, y, θ

4.3.4 Rocket Processor

The SoC includes a 64-bit RISC-V Rocket core with 1MB of main memory and a 128-kB L2 cache (Fig. 4.23). In addition to supplying calibration and control information to the spectral analysis engine, the Rocket processor post-processes data. C code can be run to

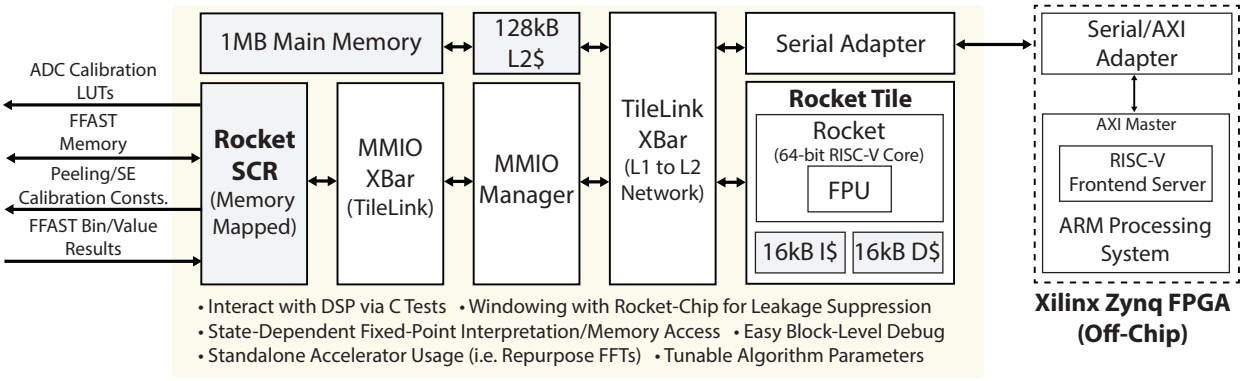

Figure 4.23: The Rocket RISC-V processor & Xilinx Zynq FPGA testing setup.

output signal locations and bandwidths after sorting the compressed frequency information. The processor can also aid the spectral analysis engine, e.g., if latency requirements are relaxed, the processor can window the ADC data to suppress spectral leakage. Finally, because Rocket has visibility into the sparse FFT memories, individual blocks in the signal processing chain like the SAR ADC slices and FFTs can be re-purposed for general applications.

As with the chip described in Chapter 2, the Rocket core interfaces with the FFAST DSP via memory-mapped SCR registers. Interfaces heavily rely on the TileLink on-chip interconnect fabric. The Rocket processor asynchronously communicates with the Xilinx Zynq FPGA used for testing via a serial adapter. Although the main memory is on-chip for this application, the general protocol for using the ARM frontend server on the FPGA to communicate with the chip and load tests remains the same.

4.4 Measurement Results

The effective number of bits (ENOBs) for individual ADC slices (corresponding to specific FFT lengths and delays) were measured both at 3.24GHz and 3.78GHz input clocks. In the ideal case, calibration LUT parameters were updated to maximize SNDR at each input frequency. Additionally, the positive and negative ADC inputs were tuned to remove phase imbalance and maintain a 180° phase offset. Fig. 4.24 indicates that there is an ENOB degradation of approximately 0.2 bits at lower unit delays when using the higher frequency input clock. This ENOB degradation is worse at higher unit delays, which are associated with ADC slices further away from the reference supply. In general, as indicated by Fig. 4.25, ENOBs are better at lower delay offsets due to spatial proximity resulting in reduced V_{ref} noise.

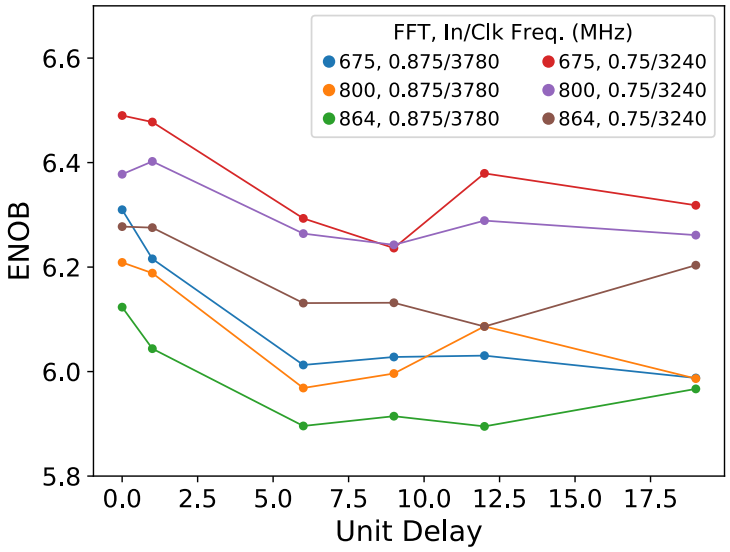

Figure 4.24: Ideal ENOB comparison between 3.24GHz and 3.78GHz clocks. Unit delays are ideal.

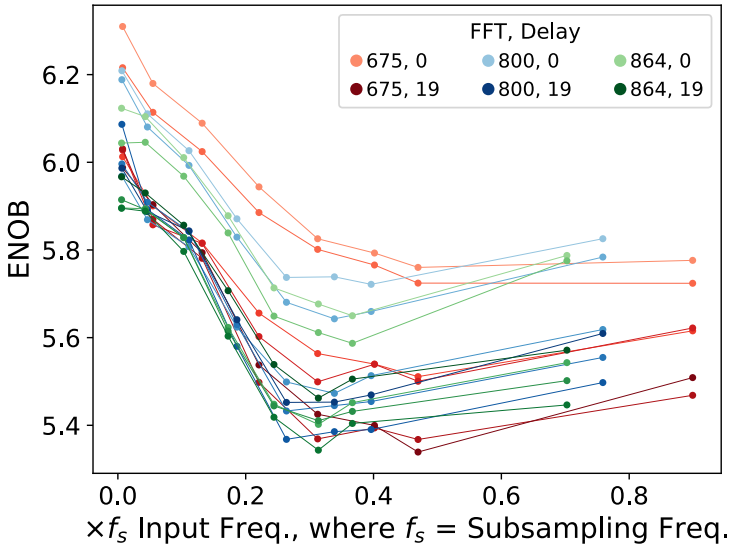

Figure 4.25: ENOBs calibrated at different input frequencies (with a 3.78GHz clock) for different ADC slices. ENOBs are higher at lower delay offsets due to the corresponding ADCs' closer proximity to the voltage reference, resulting in less noise. Phase imbalance is tuned during testing.

An important observation is that optimal calibration parameters are frequency-dependent,

so SNDR can only be maximized at one frequency point in practice. In Fig. 4.26, the SNDR is optimized around a 0.875MHz input, whereas in Fig. 4.27, the SNDR is optimized around a 15.575MHz input.

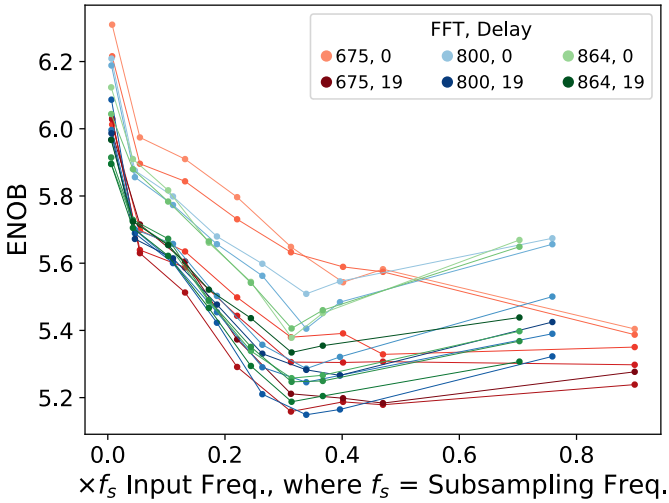

Figure 4.26: ENOBs @ 3.78GHz clock using calibration parameters taken with a 0.875MHz input. Phase imbalance was tuned during testing. SNDR is input-frequency dependent and optimized at 0.875MHz.

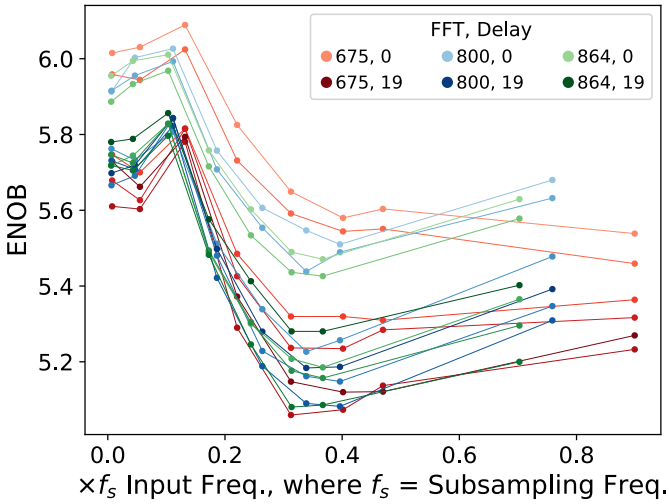

Figure 4.27: ENOBs @ 3.78GHz clock using calibration parameters taken with a 15.575MHz input. Phase imbalance was tuned during testing. SNDR is input-frequency dependent and optimized at 15.575MHz.

To understand system performance in a real application setting, the ADCs were calibrated at one input frequency (0.875MHz), and those calibration values were stored and used to determine the ENOBs across the desired input bandwidth. Additionally, a balun with approximately 5° of phase imbalance was used, resulting in SNDR degradation from HD2. The ENOBs achieved using a discrete balun are shown in Fig. 4.28. Fig. 4.29 shows the frequency spectrum obtained when a balun is not used, whereas Fig. 4.30 shows the frequency spectrum with a balun. In the first FFT plot, the second harmonic of the input frequency is not readily observable, but in the latter plot, it is one of the dominant contributors to the approximately 2.3dB SNDR degradation.

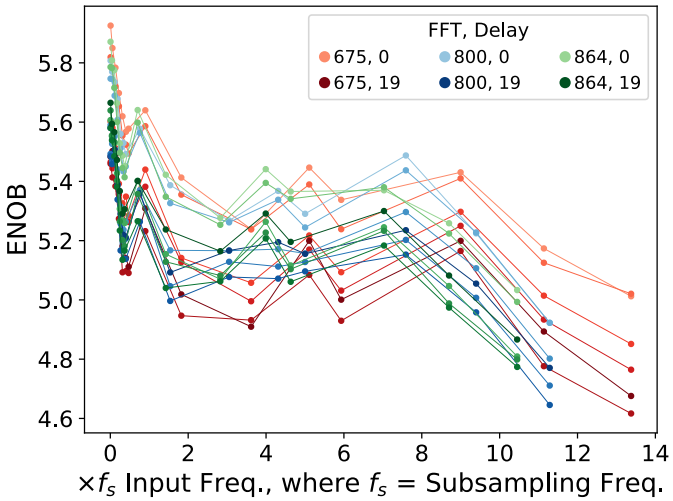

Figure 4.28: ENOBs @ 3.78GHz clock using calibration parameters taken with a 0.875MHz input. A balun with $\sim 5°$ of phase imbalance is used to measure SNDR at higher input frequencies.

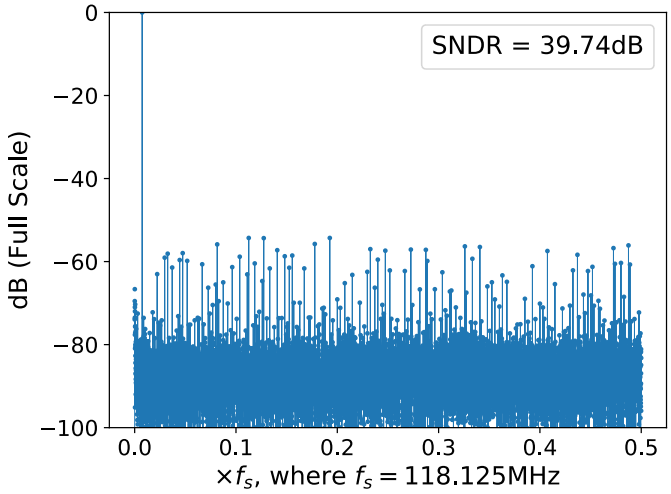

Figure 4.29: Frequency spectrum at a sub-ADC output (0.875MHz input). Tuning the differential P and N phase imbalance reduces HD2.

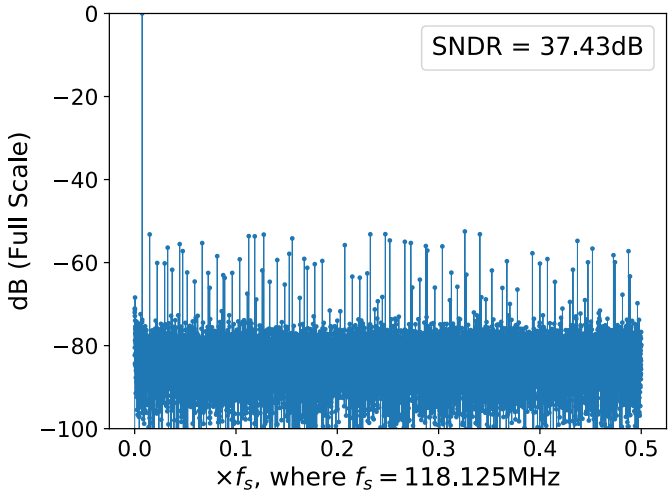

Figure 4.30: Frequency spectrum at a sub-ADC output (0.875MHz input). HD2 caused by the external balun's phase imbalance degrades SNDR.

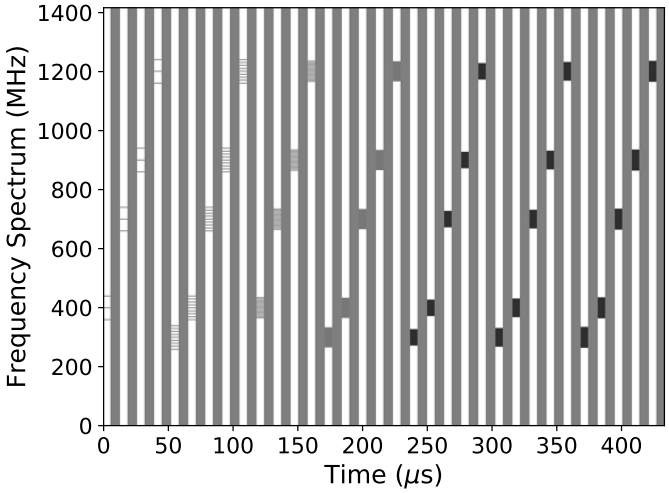

Figure 4.31: Recovered spectra over time for 3- to 37-tone (dense) vector signal generator inputs. The input tones are spread out across an approximately 80MHz bandwidth and have center frequencies ranging from 300MHz to 1.2GHz. Gray vertical regions indicate time spent on analysis, when the ADC input is not observed. A full signal acquisition and analysis cycle takes 13.3μs.

As shown in Fig. 4.31, functionality across the entire ADC/DSP chain has been verified by successfully recovering the spectrum of ∼80-MHz bandwidth, 3- to 37-tone inputs generated by a vector signal generator at 300-MHz to 1.2-GHz center frequencies. Note that the ADC input is not observed when analysis on a previous frame is being performed, corresponding to the gray periods in Fig. 4.31. The multi-tone signal at various stages of analog/digital processing is illustrated in Figs. 4.32 (time-domain output of one of the eighteen ADC slices), 4.33 (frequency-domain representation of the previous result, after the FFT), and 4.34 (the fully reconstructed sparse spectrum). Approximately 7.6μs elapse between the end of signal acquisition and the start of processor spectrum availability.

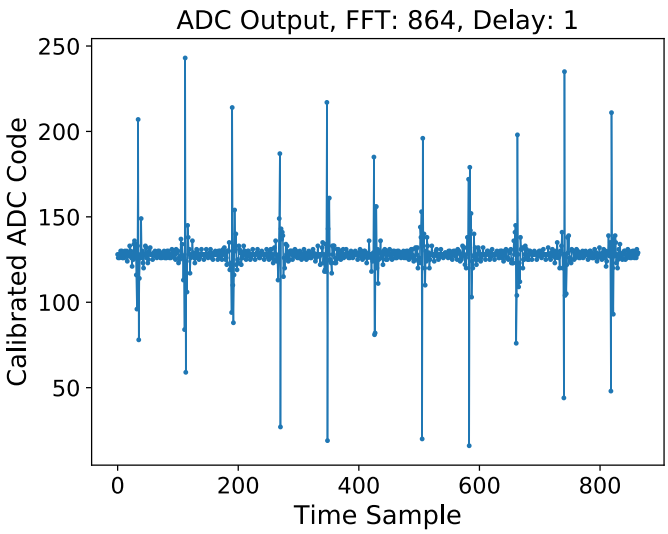

Figure 4.32: 37 tones (corresponding to 0.35% sparsity) are generated from a vector signal generator, centered at 700MHz, with a 1.925MHz spacing. The time-domain waveform is subsampled 25× in the stage corresponding to an 864-point FFT.

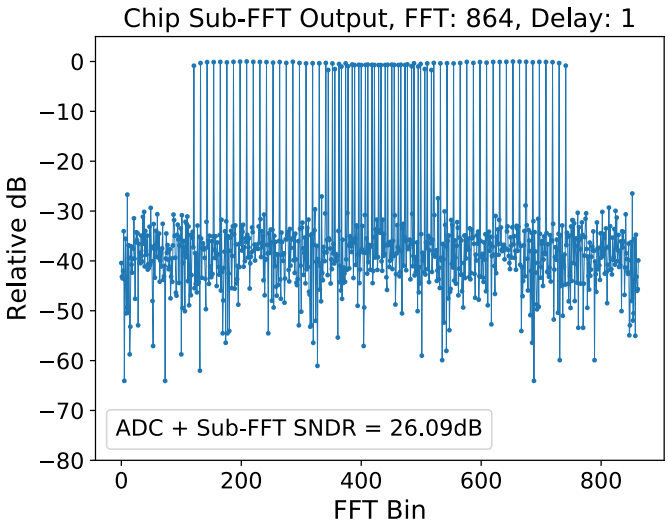

Figure 4.33: Subsampled frequency domain result at the output of the 864-point FFT. Tones have been folded down to different locations due to subsampling.

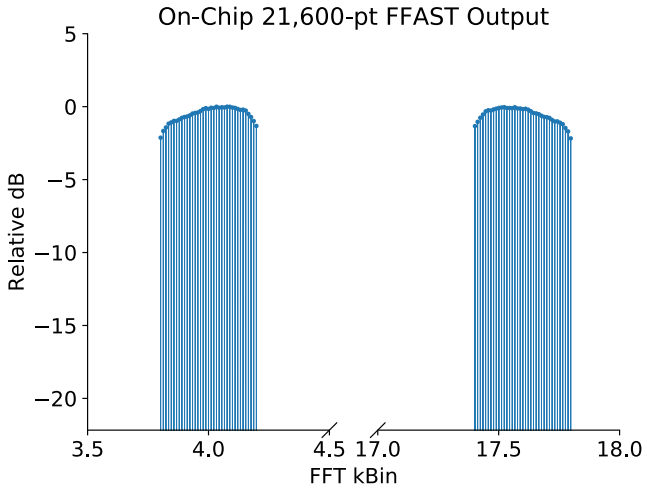

Figure 4.34: FFAST is able to reconstruct 100% of the spectrum in 13.3μs (including signal capture and analysis). The data is compressed to 0.48%, including 15-bit bin locations.

The analysis of a synthesized 3.2% sparse spectrum (achieving 4.5% false –'s and 0.8% false +'s) has been benchmarked. ∼4,700 Rocket core clock cycles elapse before the processor sees valid reconstruction data. ∼1,500 cycles are allocated for the sub-FFTs, and the remainder are used during peeling. The worst-case number of clock cycles required by the sub-FFTs corresponds to the calculation of 864-point FFTs with single iterating butterflies (although 2 radix-2 butterflies operate simultaneously). The peeling time (and number of iterations required) is a function of sparsity and increases when there are more frequency collisions. The total reconstruction time is 17.5μs; signal acquisition accounts for 5.71μs. This determines the minimum signal duration to guarantee SoC observability. An example of a spectrum generated from C test vectors is shown in Fig. 4.35. In this case, although the FFAST hardware is able to recover all real signals, some noise bins are also interpreted to be low-magnitude signal bins, resulting in false positives. These false positives can be cleaned up with an additional thresholding step during post-processing.

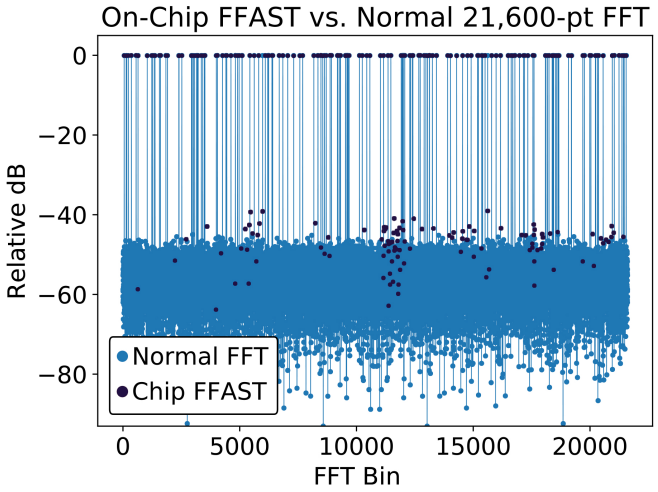

Figure 4.35: FFAST vs. normal FFT for C test vectors with 33.5dB SNR and 0.79% sparsity. Reconstruction with 0% false negatives and 0.5% false positives.

The percentages of false positives and false negatives for inputs with different sparsities and SNRs are illustrated in Figs. 4.36 and 4.37 respectively. In general, the system is more susceptible to false negatives. Unfortunately, false negatives tend to be the more application-critical metric. Therefore, further tuning of noise/signal thresholds can be performed to trade off a decrease in false negatives for an acceptable increase in false positives.

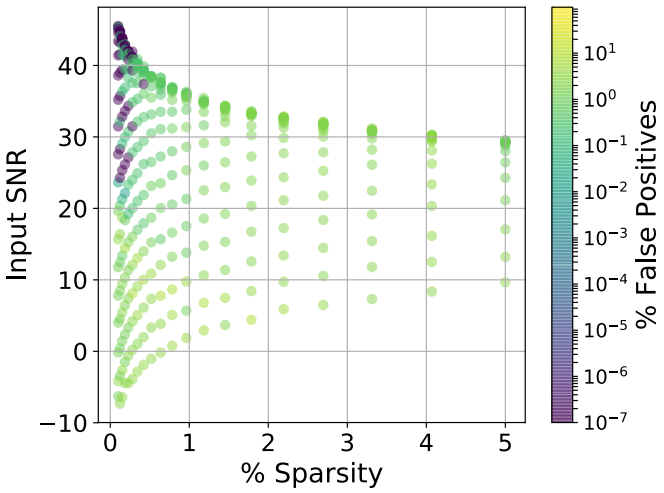

Figure 4.36: False positive rates. False positives remain below 5% for sparsities $< 5.0\%$. With respect to the number of false positives, FFAST supports input SNRs > 8.4dB for less populated spectra.

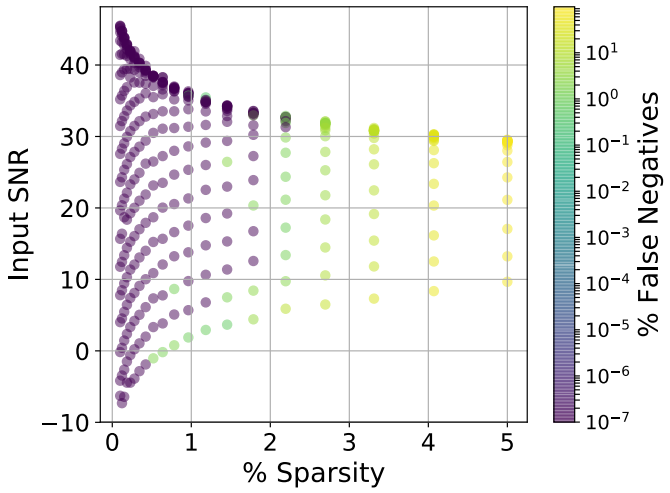

Figure 4.37: False negative rates. False negatives (worst case) remain below 5% for sparsities $< 2.7\%$. With respect to the number of false negatives, FFAST supports input SNRs > 9.7dB for less populated spectra.

4.5 Chip Summary

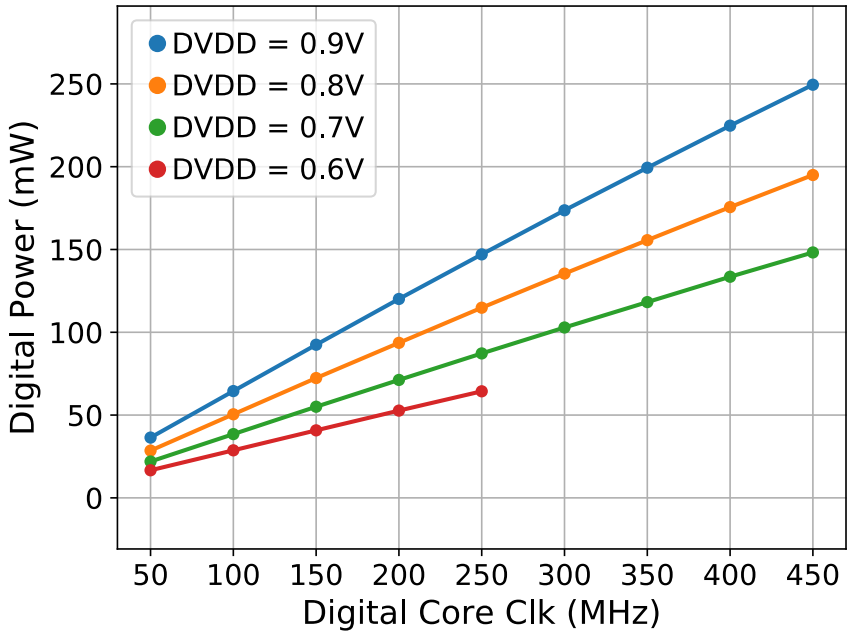

Figure 4.38: Power vs. frequency/supply with an FFAST workload.

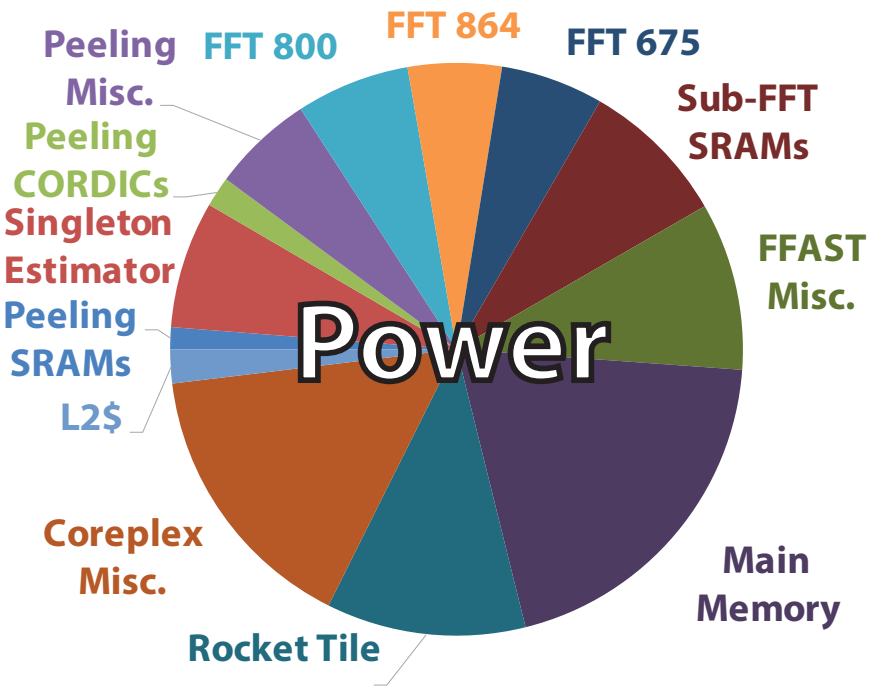

Figure 4.39: Post-place-and-route power breakdown simulated with representative test vectors.

The sparse signal analysis system has been designed as a set of highly parameterized analog/digital hardware generators for the ease of design exploration, and the described instance has been fabricated in a 16-nm FinFET process. The chip (Fig. 4.40), summarized in Table 4.2, operates at 400MHz at 0.7V VDD. The RISC-V Rocket core consumes 65.2mW, and the FFAST DSP consumes 68.3mW (Figs. 4.38, 4.39). The set of ∼6 ENOB ADCs (at 0.85V) and associated clock dividers (at 0.95V) consume 49.8mW/48.1mW with a 3.78GHz/3.24GHz input clock. At 3.24GHz, the SAR ADC cores consume 33.7mW, while the clock dividers consume 14.4mW. At 3.78GHz, the SAR ADC cores consume 33.8mW, while the clock dividers consume 16mW.

The set of analog and digital hardware generators enabled the rapid 16-nm implementation of the FFAST algorithm in < 2 months. This is the first fully-integrated sparse spectral analysis SoC, with an on-chip ADC (see Table 4.3). The $< 20\mu$s runtime enables real-time frequency adaptation in closed-loop systems.

Finally, one might wonder why the supported sparsity in hardware is only $\sim 3.2\%$ when a floating-point Matlab implementation achieves $\sim 8.5\%$. The ADCs and fixed-point arithmetic incur a quantization noise penalty (SNR = 6.02B + 1.76dB). Furthermore, because of the computational complexity required by the singleton estimator, individual operations are heavily pipelined. Since pipeline stalls were not implemented, occasionally, data from a bin b at a given stage i is requested before its value has been updated. The probability that this occurs increases as sparsity degrades, limiting the achievable sparsity of the sys-

Table 4.2: Chip summary.
3.2% sparsity, signal dependent.[1] @ 3.78Gs/s effective, 0.85V/0.95V ADC/Clk supplies.*

Technology	16nm FinFET	
Sensing Bandwidth	1.89GHz	
Sensing Resolution	175kHz	
Signal Acquisition Time	5.71μs	
Analysis Time	11.8μs (3.2% Sparsity, 4.5% False –, 0.8% False +, Signal Dependent)	
ADC ENOBs	Up to 6.3 ENOB/Slice	
Compression	**ADC Output**	65% of Samples
	FFAST Output	4.4% of Samples[1]
Sub-FFT SQNRs	**675-pt FFT**	38-52dB (Signal Dependent)
	800-pt FFT	42-56dB (Signal Dependent)
	864-pt FFT	41-56dB (Signal Dependent)
Area	**Analog (ADC + Clocks)**	0.27mm^2
	FFAST Digital	0.85mm^2
	Rocket Core	0.37mm^2 (Including L2$)
	Main Memory	1.07mm^2
Memory	**Main Memory**	1MB
	L2$	128kB
	L1$	2 × 16kB (Data + Instruction)
	FFAST	97kB
Power	**SAR ADC**	49.8mW*
	FFAST DSP	68.3mW (400MHz, 0.7V)
	64-Bit Rocket-Chip	65.2mW (400MHz, 0.7V)

tem. Additional tones can be supported if pipeline stalls are properly inserted, although the analysis time will be slightly degraded. Eliminating the data hazard in the deeply pipelined data path of the sparse FFT's reconstruction back end should roughly double the number of signals detectable in real time. Furthermore, to support more realistic, off-grid signals—which effectively degrade sparsity—in real time, windowing can be performed at the front end (either digitally or in the analog domain, using an architecture similar to that reported in [90]). Again, this requires that the lane-to-lane skew be accurately measured.

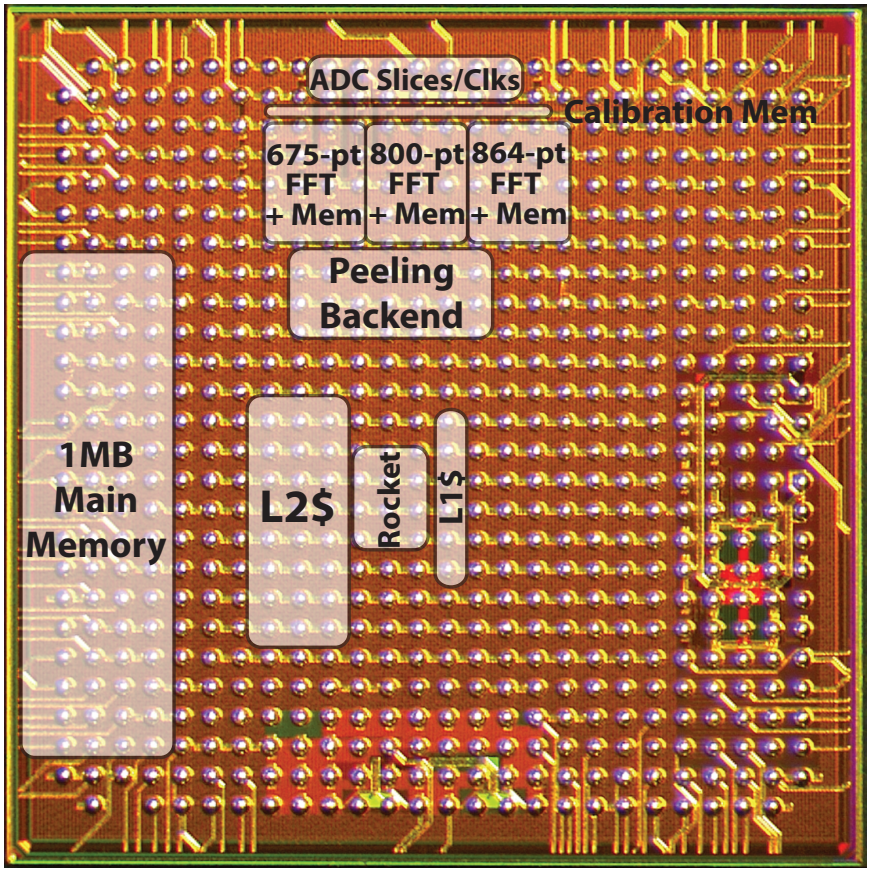

Figure 4.40: Die photo.

Table 4.3: Comparison with state-of-the-art.

	Abari *et al.* [23]	**Bailey *et al.* [89]**	**This Work**
Technology	45nm	28nm	16nm
Bandwidth (GHz)	-	8.5	1.89
FFT Points	746,496	8,192	21,600
Integrated ADC	N	N	Y
Power (mW), *incl. ADC	174.8	5200*	183.3*
Supported Sparsity	0.1%	100%	3.2%
Can Post-Process	N	N	Y
Data Compressed	Y	N	Y
Noise Robust	N	-	Y
Runtime, *incl. ADC	6.8μs	-	17.5μs*
FOM (Pts. $\times$ BW) / Pwr.	-	13.4	222.7

Chapter 5

Conclusions

This work presents a methodology, associated tools, and general examples of agile development of DSP hardware systems. The ACED library has been developed to augment Chisel and specifically address the needs of agile DSP development. Namely, it offers end-to-end support for the algorithm-to-hardware translation process, providing systems designers with the ability to rapidly evaluate algorithms at high levels of abstraction via extensive parameterization and cycle-accurate floating-point support, automatic translation into synthesizable hardware via DSP typeclasses, verification support that spans abstraction and tool boundaries, and unobtrusive optimization/platform specialization via FIRRTL compiler passes. It has been used extensively to build DSP generators. Examples include FFT generators, CORDIC generators, FIR filter generators, etc. In particular, this thesis has focused on the methodology used to build a runtime-reconfigurable, mixed-radix, memory-based hardware FFT generator. It outlined the algorithm and architecture understanding necessary to generalize (and parameterize) a hardware FFT template, and illustrated how the hardware template can be populated via a separate software firmware block that analyzes user requirements. The FFT generator has been used in two rapidly designed DSP systems that have been tightly coupled to the Rocket-Chip ecosystem. The first fabricated 16-nm chip included a Wi-Fi/LTE compatible FFT engine. The second fabricated 16-nm chip, a real-time, high-bandwidth, high-resolution sparse signal analysis SoC, used mixed-radix FFTs as part of the back end signal processing chain.

5.1 Summary of Contributions

This work adapts the agile hardware methodology—previously used to develop and tape-out RISC-V processors [91] and built around the Chisel [27] and FIRRTL [57] ecosystem—to the design of DSP generators. In particular, it presents:

- The first *generator of runtime-reconfigurable, mixed-radix FFTs* recorded in open literature, capable of producing hardware instances that are comparable to custom-tailored state-of-the-art,

- A mixed-radix control logic representation that simplifies the design of the FFT generator and simplifies modulo operations in hardware while supporting the prime factor algorithm,
- A new scheduling approach to support multiple parallel butterflies—where the FFT N is divisible by the number of butterflies B—in runtime-reconfigurable, mixed-radix FFTs,
- A demonstration of a DSP (FFT) system designed in Chisel as an accelerator attached to a RISC-V core and measured in a 16nm process to have performance comparable to state-of-the-art,
- A DSP library for Chisel called ACED that simplifies the mapping of algorithms into hardware by supporting:
  - Operator and data type parameterization that enables DSP generators to target various data types like complex or real numbers without any redesign,
  - A unified design and systems modeling environment enabling float-to-fixed verification and the automatic propagation of tests across abstraction boundaries,
  - Design profiling to automatically reduce bitwidths, and
  - FIRRTL compiler passes that trim bitwidths via range propagation, and
- The first automatically generated, fully on-chip sparse spectral analysis SoC, with a BAG-designed custom SAR ADC frontend, generated mixed-radix FFTs, digital reconstruction backend based off of the FFAST algorithm [31], and RISC-V processor. The system targets realistic spectral sparsities/SNR and is able to operate in real time.

5.2 Future Work

This work presents a complete, self-contained DSP development cycle—from methodologies, algorithms, and libraries to implementations and applications. Many extensions and applications are possible. Future work may continue to iterate on this development cycle, focusing on individual classes of improvements, as highlighted below:

- *ACED Library Improvements and Additional Use Cases*
  - ACED can be used to perform a more comprehensive system-level SQNR study of various DSP blocks like the FFT for more aggressive bitwidth optimization.
  - Block floating-point can be generalized and natively supported with ACED. Its use along the FFT data path would enable better trade-offs between SQNR and power/area.

  - More advanced FIRRTL compiler passes (e.g., using affine analysis) can be developed for range analysis and bitwidth reduction.

- *Higher-Level DSP Generators and Building a Generator Library*
  - A streaming FFT generator can be designed, so that a top-level generator can choose to use either the steaming- or memory-based flavor depending on application constraints (e.g., throughput vs. area, degree of runtime reconfigurability needed, etc.).
  - Ultimately, it is desirable to create a library of common high- or low-level DSP generators so that their outputs can be easily stitched together through a common interface to quickly prototype complex DSP systems. Example blocks include correlators, Viterbi decoders, frequency equalizers, etc.

- *Improved Systems and Further Application Exploration*
  - Further analysis can be performed to better understand the impact of analog nonidealities on FFAST performance (in terms of the amount of sparsity supported, false positives and negatives, etc.).
  - Future sparse spectral analysis chips can support reconfigurable hardware that would allow runtime adaptation for different spectral sparsities and enable sparsity vs. power trade-offs.
  - Although not considered in this work, windowing, either in the analog domain via capacitor multiplication or in the digital domain, can be performed to improve the robustness of the sparse spectrum analyzer against unknown, off-grid signals that degrade effective sparsity.
  - It would be interesting to evaluate the performance of the spectrum analyzer in the feedback path of a closed-loop system for interference mitigation.
  - It would likewise be interesting to see the FFT generator applied towards systems not in the wireless space (e.g., in convolutional neural networks).

Bibliography

[1] *Wikimedia commons*, `https://commons.wikimedia.org`, Wikimedia Foundation, 2018 (cit. on p. 1).

[2] J. Cooley and J. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, Apr. 1965 (cit. on pp. 2, 11).

[3] H. G. Myung, "Introduction to single carrier FDMA", in *2007 15th European Signal Processing Conference*, Sep. 2007, pp. 2144–2148 (cit. on p. 3).

[4] *LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); physical channels and modulation (3GPP TS 36.211 version 12.3.0 release 12)*, 3rd Generation Partnership Project, Oct. 2014 (cit. on p. 3).

[5] M. Frigo, "A fast Fourier transform compiler", in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99, Atlanta, Georgia, USA: ACM, 1999, pp. 169–180 (cit. on p. 3).

[6] *Discrete Fourier transform v4.0*, `https://www.xilinx.com/support/documentation/ip_documentation/dft/v4_0/pg106-dft.pdf`, Xilinx, 2015 (cit. on p. 4).

[7] *DFT/IDFT reference design*, `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an464.pdf`, Altera, 2007 (cit. on p. 4).

[8] Z.-X. Yang, Y.-P. Hu, C.-Y. Pan, *et al.*, "Design of a 3780-point IFFT processor for TDS-OFDM", *IEEE Transactions on Broadcasting*, vol. 48, no. 1, pp. 57–61, Mar. 2002 (cit. on p. 4).

[9] M. Püschel, J. M. F. Moura, J. R. Johnson, *et al.*, "SPIRAL: Code generation for DSP transforms", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005 (cit. on pp. 4, 9, 62, 63, 74).

[10] O. Shacham, *Creating chip generators using Genesis2*, `http://genesis2.stanford.edu`, Stanford University (cit. on pp. 4, 63, 74).

[11] R. Koutsoyannis, P. A. Milder, C. R. Berger, *et al.*, "Improving fixed-point accuracy of FFT cores in O-OFDM systems", in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2012, pp. 1585–1588 (cit. on p. 4).

[12] P. Milder, F. Franchetti, J. C. Hoe, *et al.*, "Computer generation of hardware for linear digital signal processing transforms", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, Apr. 2012 (cit. on p. 4).

[13] C. H. Yang, T. H. Yu, and D. Marković, "Power and area minimization of reconfigurable FFT processors: A 3GPP-LTE example", *IEEE Journal of Solid-State Circuits*, vol. 47, no. 3, pp. 757–768, Mar. 2012 (cit. on pp. 4, 5).

[14] J. Hickish, Z. Abdurashidova, Z. Ali, *et al.*, "A Decade of Developing Radio-Astronomy Instrumentation using CASPER Open-Source Technology", *Journal of Astronomical Instrumentation*, vol. 5, 1641001-12, pp. 1 641 001–12, Mar. 2016. DOI: 10.1142/S2251171716410014. arXiv: 1611.01826 [astro-ph.IM] (cit. on p. 5).

[15] A. Parsons, "The symmetric group in data permutation, with applications to high-bandwidth pipelined FFT architectures", *IEEE Signal Processing Letters*, vol. 16, no. 6, pp. 477–480, Jun. 2009 (cit. on p. 5).

[16] B. Richards, N. Nicolici, H. Chen, *et al.*, "A 1.5GS/s 4096-point digital spectrum analyzer for space-borne applications", in *2009 IEEE Custom Integrated Circuits Conference*, Sep. 2009, pp. 499–502 (cit. on pp. 5, 7).

[17] G. Yahalom, "Analog-digital co-existence in 3D-IC", PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2016 (cit. on pp. 5, 30, 31, 42, 70).

[18] K. F. Xia, B. Wu, T. Xiong, *et al.*, "A memory-based FFT processor design with generalized efficient conflict-free address schemes", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 6, pp. 1919–1929, Jun. 2017 (cit. on pp. 5, 70).

[19] J. Chen, J. Hu, S. Lee, *et al.*, "Hardware efficient mixed radix-25/16/9 FFT for LTE systems", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 2, pp. 221–229, Feb. 2015 (cit. on pp. 5, 11, 14, 21, 24, 62, 63, 70).

[20] F. Qureshi, M. Garrido, and O. Gustafsson, "Unified architecture for 2, 3, 4, 5, and 7-point DFTs based on Winograd Fourier transform algorithm", *Electronics Letters*, vol. 49, no. 5, pp. 348–349, Feb. 2013 (cit. on pp. 5, 9, 52, 53, 55).

[21] S. Richardson, D. Marković, A. Danowitz, *et al.*, "Building conflict-free FFT schedules", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 4, pp. 1146–1155, Apr. 2015 (cit. on pp. 5, 36, 38).

[22] C. F. Hsiao, Y. Chen, and C. Y. Lee, "A generalized mixed-radix algorithm for memory-based FFT processors", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 57, no. 1, pp. 26–30, Jan. 2010 (cit. on pp. 5, 11, 14, 15, 23, 27, 31, 35, 62, 63).

[23] O. Abari, E. Hamed, H. Hassanieh, *et al.*, "A 0.75-million-point Fourier-transform chip for frequency-sparse signals", in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 458–459 (cit. on pp. 5, 9, 93, 127).

[24] H. Hassanieh, P. Indyk, D. Katabi, *et al.*, "Simple and practical algorithm for sparse Fourier transform", in *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '12, Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 1183–1194 (cit. on p. 6).

[25] P. Indyk, M. Kapralov, and E. Price, "(Nearly) sample-optimal sparse Fourier transform", in *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '14, Portland, Oregon: Society for Industrial and Applied Mathematics, 2014, pp. 480–499 (cit. on p. 6).

[26] H. Hassanieh, L. Shi, O. Abari, *et al.*, "GHz-wide sensing and decoding using the sparse Fourier transform", in *IEEE INFOCOM 2014—IEEE Conference on Computer Communications*, Apr. 2014, pp. 2256–2264 (cit. on pp. 6, 92, 93).

[27] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a Scala embedded language", in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 1212–1221 (cit. on pp. 8, 9, 71, 72, 74, 108, 128).

[28] A. Wang, J. Bachrach, and B. Nikolić, "A generator of memory-based, runtime-reconfigurable $2^n3^m5^k$ FFT engines", in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2016, pp. 1016–1020 (cit. on pp. 9, 15, 55, 108).

[29] A. Wang, P. Rigge, A. Izraelevitz, *et al.*, "ACED: A hardware library for generating DSP systems", in *DAC Design Automation Conference 2018*, Jun. 2018, to be published (cit. on p. 9).

[30] K. Asanović, R. Avižienis, J. Bachrach, *et al.*, "The Rocket Chip generator", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html` (cit. on pp. 9, 63).

[31] S. Pawar and K. Ramchandran, "FFAST: An algorithm for computing an exactly k-sparse DFT in $\mathcal{O}(k\log k)$ time", *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 429–450, Jan. 2018 (cit. on pp. 9, 129).

[32] E. Chang, J. Han, W. Bae, *et al.*, "BAG2: A process-portable framework for generator-based AMS circuit design", in *2018 IEEE Custom Integrated Circuits Conference (CICC)*, Apr. 2018, to be published (cit. on pp. 9, 105).

[33] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009 (cit. on p. 10).

[34] C. S. Burrus, *Fast Fourier transforms*, `http://cnx.org/contents/82e6ba6f-b828-42ef-9db1-8de4b448b869@22.1`, OpenStax CNX, Nov. 2012 (cit. on pp. 12, 13, 15).

[35] V. Stojanović, "Fast Fourier transform: Theory and algorithms", in *Course Materials for 6.973 Communication System Design*, MIT OpenCourseWare (`http://ocw.mit.edu`), Massachusetts Institute of Technology, Spring 2006 (cit. on pp. 14, 22).

[36] C. Burrus, "Index mappings for multidimensional formulation of the DFT and convolution", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 3, pp. 239–242, Jun. 1977 (cit. on pp. 19, 21).

[37] U. Meyer-Baese, *Digital signal processing with field programmable gate arrays*, 3rd. Springer Publishing Company, Incorporated, 2007 (cit. on p. 21).

[38] I. J. Good, "The relationship between two fast Fourier transforms", *IEEE Transactions on Computers*, vol. C-20, no. 3, pp. 310–317, Mar. 1971 (cit. on p. 21).

[39] T. Chiueh and P. Tsai, *OFDM baseband receiver design for wireless communications*. Wiley, 2008 (cit. on p. 30).

[40] H. Sorokin and J. Takala, "Conflict-free parallel access scheme for mixed-radix FFT supporting I/O permutations", in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011, pp. 1709–1712 (cit. on p. 35).

[41] J. Löfgren and P. Nilsson, "On hardware implementation of radix 3 and radix 5 FFT kernels for LTE systems", in *2011 NORCHIP*, Nov. 2011, pp. 1–4 (cit. on pp. 42, 63).

[42] S. Winograd, "On computing the discrete Fourier transform", *Mathematics of Computation*, vol. 32, no. 141, pp. 175–199, 1978 (cit. on pp. 44, 48, 50, 51).

[43] P. Duhamel and M. Vetterli, "Fast Fourier transforms: A tutorial review and a state of the art", *Signal Processing*, vol. 19, no. 4, pp. 259–299, Apr. 1990 (cit. on p. 44).

[44] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime", *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, Jun. 1968 (cit. on p. 44).

[45] E. W. Weisstein, *Primitive root*, `http://mathworld.wolfram.com/PrimitiveRoot.html`, MathWorld–A Wolfram Web Resource, 2018 (cit. on pp. 44, 45).

[46] D. M. Burton, *Elementary number theory*, ser. International Series in Pure and Applied Mathematics. McGraw-Hill Higher Education, 2007 (cit. on p. 45).

[47] H. J. Nussbaumer, *Fast Fourier transform and convolution algorithms*, ser. Springer Series in Information Sciences. Springer-Verlag Berlin Heidelberg, 1982 (cit. on p. 45).

[48] T. S. Huang, *Two-dimensional digital signal processing II: Transforms and median filters*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1981 (cit. on pp. 47, 48, 50, 51).

[49] E. W. Weisstein, *Chinese remainder theorem*, `http://mathworld.wolfram.com/ChineseRemainderTheorem.html`, MathWorld–A Wolfram Web Resource, 2018 (cit. on p. 48).

[50] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): Seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997 (cit. on p. 49).

[51] S. S. Narayan, M. J. Narasimha, and A. M. Peterson, "DFT algorithms—analysis and implementation", Center for Radar Astronomy, Stanford Electronics Laboratories, Department of Electrical Engineering, Stanford University, Tech. Rep. 3606-12, May 1978. [Online]. Available: `http://www.dtic.mil/dtic/tr/fulltext/u2/a058049.pdf` (cit. on pp. 51–53).

[52] D. Kolba and T. Parks, "A prime factor FFT algorithm using high-speed convolution", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 4, pp. 281–294, Aug. 1977 (cit. on p. 51).

[53] F. Qureshi, M. Ali, and J. Takala, "Multiplierless reconfigurable processing element for mixed radix-2/3/4/5 FFTs", in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct. 2017, pp. 1–6 (cit. on p. 58).

[54] *Breeze: Breeze is a numerical processing library for Scala*, `https://github.com/scalanlp/breeze`, ScalaNLP, 2018 (cit. on pp. 59, 64, 76).

[55] G. Wang, B. Yin, I. Cho, *et al.*, "Efficient architecture mapping of FFT/IFFT for cognitive radio networks", in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2014, pp. 3933–3937 (cit. on p. 62).

[56] *Floating-point to fixed-point conversion*, `https://www.mathworks.com/examples/matlab-hdl-coder`, The MathWorks Inc., 2018 (cit. on p. 72).

[57] A. Izraelevitz, J. Koenig, P. Li, *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations", in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 209–216 (cit. on pp. 72, 75, 128).

[58] A. Gai, *Model-based design with MATLAB, Simulink, and Altera DSP builder*, `ftp://ftp.altera.com/outgoing/download/education/events/code-dsp06/MathWorks_RF.pdf`, Altera, 2006 (cit. on pp. 73, 83).

[59] W. R. Davis, N. Zhang, K. Camera, *et al.*, "A design environment for high throughput, low power dedicated signal processing systems", in *Proceedings of the IEEE 2001 Custom Integrated Circuits Conference*, May 2001, pp. 545–548 (cit. on p. 73).

[60] D. Marković, C. Chang, B. Richards, *et al.*, "ASIC design and verification in an FPGA environment", in *2007 IEEE Custom Integrated Circuits Conference*, Sep. 2007, pp. 737–740 (cit. on p. 73).

[61] *Ultrafast high-level productivity design methodology guide*, `https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf`, Xilinx, Dec. 2017 (cit. on p. 73).

[62] Y. Shao, "Design and modeling of specialized architectures", Harvard University, Tech. Rep., 2016 (cit. on p. 73).

[63] A. Canis, J. Choi, M. Aldham, *et al.*, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems", *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, 24:1–24:27, Sep. 2013 (cit. on p. 74).

[64] A. Papakonstantinou, K. Gururaj, J. A. Stratton, *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs", in *2009 IEEE 7th Symposium on Application Specific Processors*, Jul. 2009, pp. 35–42 (cit. on p. 74).

[65] J. Hegarty, J. Brunhaver, Z. DeVito, *et al.*, "Darkroom: Compiling high-level image processing code into hardware pipelines", *ACM Transactions on Graphics*, vol. 33, no. 4, 144:1–144:11, Jul. 2014 (cit. on p. 74).

[66] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research", in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 280–292 (cit. on p. 74).

[67] R. Nikhil, "Bluespec System Verilog: Efficient, correct RTL from high level specifications", in *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, ser. MEMOCODE '04, Jun. 2004, pp. 69–70 (cit. on p. 74).

[68] E. Osheim, T. Switzer, R. Klaehn, *et al.*, *Spire: Powerful new number types and numeric abstractions for Scala*, `https://github.com/non/spire`, 2018 (cit. on p. 77).

[69] J. R. Barry, D. G. Messerschmitt, and E. A. Lee, *Digital communication: Third edition*. Norwell, MA, USA: Kluwer Academic Publishers, 2003 (cit. on p. 80).

[70] H. Keding, M. Willems, M. Coors, *et al.*, "FRIDGE: A fixed-point design and simulation environment", in *Proceedings Design, Automation and Test in Europe*, Feb. 1998, pp. 429–435 (cit. on p. 83).

[71] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation", in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00, Vancouver, British Columbia, Canada: ACM, 2000, pp. 108–120 (cit. on p. 83).

[72] D.-U. Lee, A. A. Gaffar, O. Mencer, *et al.*, "MiniBit: Bit-width optimization via affine arithmetic", in *Proceedings of the 42nd Annual Design Automation Conference*, Jun. 2005, pp. 837–840 (cit. on p. 83).

[73] C. C. Wang, C. Shi, R. W. Broderson, *et al.*, "An automated fixed-point optimization tool in MATLAB XSG/SynDSP environment", *ISRN Signal Processing*, 2011 (cit. on p. 83).

[74] V. Strassen, "Gaussian elimination is not optimal", *Numer. Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969 (cit. on p. 87).

[75] D. L. Donoho, "Compressed sensing", *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006 (cit. on p. 93).

[76] M. Mishali and Y. C. Eldar, "From theory to practice: Sub-Nyquist sampling of sparse wideband analog signals", *IEEE Journal of Selected Topics in Signal Processing*, vol. 4, no. 2, pp. 375–391, Apr. 2010 (cit. on p. 93).

[77] J. Yoo, S. Becker, M. Loh, *et al.*, "A 100MHz-2GHz 12.5x sub-Nyquist rate receiver in 90nm CMOS", in *2012 IEEE Radio Frequency Integrated Circuits Symposium*, Jun. 2012, pp. 31–34 (cit. on p. 93).

[78] S. Pawar and K. Ramchandran, "R-FFAST: A robust sub-linear time algorithm for computing a sparse DFT", *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 451–466, Jan. 2018 (cit. on pp. 93, 94, 97, 99, 100, 102).

[79] R. Gallager, "Low-density parity-check codes", *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962 (cit. on p. 95).

[80] E. J. Candès, J. K. Romberg, and T. Tao, "Stable signal recovery from incomplete and inaccurate measurements", *Communications on Pure and Applied Mathematics*, vol. 59, no. 8, pp. 1207–1223, 2006 (cit. on p. 97).

[81] E. J. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information", *IEEE Transactions on Information Theory*, vol. 52, no. 2, pp. 489–509, Feb. 2006 (cit. on p. 97).

[82] F. Ong, S. Pawar, and K. Ramchandran, "Fast sparse 2-D DFT computation using sparse-graph alias codes", in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Mar. 2016, pp. 4059–4063 (cit. on p. 99).

[83] S. Kay, "A fast and accurate single frequency estimator", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 1987–1990, Dec. 1989 (cit. on p. 101).

[84] X. Li, S. Pawar, and K. Ramchandran, "Sub-linear time compressed sensing using sparse-graph codes", in *2015 IEEE International Symposium on Information Theory (ISIT)*, Jun. 2015, pp. 1645–1649 (cit. on p. 103).

[85] D. Stepanović and B. Nikolić, "A 2.8 GS/s 44.6 mW time-interleaved ADC achieving 50.9 dB SNDR and 3 dB effective resolution bandwidth of 1.5 GHz in 65 nm CMOS", *IEEE Journal of Solid-State Circuits*, vol. 48, no. 4, pp. 971–982, Apr. 2013 (cit. on p. 105).

[86] Y. Duan and E. Alon, "A 12.8 GS/s time-interleaved ADC with 25 GHz effective resolution bandwidth and 4.6 ENOB", *IEEE Journal of Solid-State Circuits*, vol. 49, no. 8, pp. 1725–1738, Aug. 2014 (cit. on p. 106).

[87] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers", in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98, Monterey, California, USA: ACM, 1998, pp. 191–200 (cit. on pp. 111, 114).

[88] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor", in *Advances in Cryptology–CRYPTO' 86*, A. M. Odlyzko, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323 (cit. on pp. 110, 113).

[89] S. Bailey, J. Wright, N. Mehta, *et al.*, "A 28nm FDSOI 8192-point digital ASIC spectrometer from a Chisel generator", in *Proc. IEEE CICC*, Apr. 2018 (cit. on p. 127).

[90] D. Bankman and B. Murmann, "An 8-bit, 16 input, 3.2 pJ/op switched-capacitor dot product circuit in 28-nm FDSOI CMOS", in *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, Nov. 2016, pp. 21–24 (cit. on p. 126).

[91] Y. Lee, A. Waterman, H. Cook, *et al.*, "An agile approach to building RISC-V microprocessors", *IEEE Micro*, vol. 36, no. 2, pp. 8–20, Mar. 2016 (cit. on p. 128).

[92] O. Shacham, O. Azizi, M. Wachs, *et al.*, "Rethinking digital design: Why design must change", *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov. 2010.

[93] A. Wang, B. Richards, P. Dabbelt, *et al.*, "A 0.37mm^2 LTE/Wi-Fi compatible, memory based, runtime-reconfigurable $2^n3^m5^k$ FFT accelerator integrated with a RISC-V core in 16nm FinFET", in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, Nov. 2017, pp. 305–308.

[94] A. Wang, W. Bae, J. Han, *et al.*, "A real-time, analog/digital co-designed 1.89-GHz bandwidth, 175-kHz resolution sparse spectral analysis RISC-V SoC in 16-nm FinFET", submitted for publication.

[95] C. Burrus and P. Eschenbacher, "An in-place, in-order prime factor FFT algorithm", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 4, pp. 806–817, Aug. 1981.

[96] P. Y. Tsai and C. Y. Lin, "A generalized conflict-free memory addressing scheme for continuous-flow parallel-processing FFT processors with rescheduling", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 12, pp. 2290–2302, Dec. 2011.

[97] R. Patterson and J. McClellan, "Fixed-point error analysis of Winograd Fourier transform algorithms", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 5, pp. 447–455, Oct. 1978.

[98] P. Welch, "A fixed-point fast Fourier transform error analysis", *IEEE Transactions on Audio and Electroacoustics*, vol. 17, no. 2, pp. 151–157, Jun. 1969.

[99] R. B. Perlow and T. C. Denk, "Finite wordlength design for VLSI FFT processors", in *Conference Record of Thirty-Fifth Asilomar Conference on Signals, Systems and Computers*, vol. 2, Nov. 2001, 1227–1231 vol.2.

[100] A. Agarwal, H. Hassanieh, O. Abari, *et al.*, "High-throughput implementation of a million-point sparse Fourier transform", in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–6.

[101] A. López-Parrado and J. Velasco-Medina, "SoC-FPGA implementation of the sparse fast Fourier transform algorithm", in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2017, pp. 120–123.

[102] O. Shacham, S. Galal, S. Sankaranarayanan, *et al.*, "Avoiding game over: Bringing design to the next level", in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 623–629.

[103] J. Ou and V. K. Prasanna, "MATLAB/Simulink based hardware/software co- simulation for designing using FPGA configured soft processors", in *19th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2005, 148b–148b. DOI: 10.1109/IPDPS.2005.275.

[104] D. Zaretsky, M. Mittal, X. Tang, *et al.*, "Overview of the FREEDOM compiler for mapping DSP software to FPGAs", in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2004, pp. 37–46.