Rapid ASIC Design for Digital Signal Processors

Steven Bailey

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-32
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-32.html

May 1, 2020

Rapid ASIC Design for Digital Signal Processors

by

Steven Bailey

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 Elad Alon
Professor Aaron Parsons

Fall 2018

Rapid ASIC Design for Digital Signal Processors

Abstract

Rapid ASIC Design for Digital Signal Processors

by

Steven Bailey

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Application-specific integrated circuit (ASIC) signal processors are necessary to achieve the high performance and low power requirements of modern applications, but long development times are one hurdle contributing to their declining adoption. A significant percentage of their development time goes into the design and verification of the architecture, with the remainder consumed by back-end ASIC flow work and chip testing. Agile hardware principles, borrowed from a similar successful software approach and previously applied to general-purpose processors, offer a promising solution to continuing the development of signal processing systems on a chip (SoCs).

This work presents a digital signal processing SoC design framework that, when coupled with agile design principles, supports rapid prototyping and designing of ASICs for signal processing applications. First, applications and existing ASIC solutions are explored and analyzed in Chapter 2 to glean useful properties and trends. From this, Chapter 3 proposes a model for a generic signal processing SoC is developed. Next, a new Chisel generator design framework is presented in Chapter 4. Chisel is a hardware construction language written as a DSL in Scala, allowing for high-level and functional programming use when designing hardware. This framework couples a general-purpose processor with a signal processing accelerator, and much of the library code for connectivity, memory mapping, and programming is made available. This framework, when coupled with an agile design process, supports rapid development of ASICs. The accelerator performs streaming signal processing to offload high-throughput computational kernels from the CPU. As processing elements for the desired application are produced, processing moves from the CPU to the accelerator. Low-rate processing tasks are computed on the CPU, meaning tape-out occurs on time and produces a working chip able to perform the entire application.

The methodology and proposed agile design process were validated on two separate chips in Chapters 5 and 6, spanning two applications and two process nodes. The ASIC spectrometer (Splash2), for which the RTL was designed in eight weeks by one person, demonstrates the power of Chisel to rapidly construct processing element generators. These generators were then improved and the parameters adjusted as physical design and timeline constraints

imposed new restrictions. The radar receive processor design fleshed out the generator framework details. A significantly larger design, this chip required about 300 engineering-weeks of work over 9 months, equivalent to a team of 8 engineers working full time. About 30% of that time was spent designing the framework and reusable processing elements. This represents a 56% reduction in development time compared to the estimated 14.4 months from standard practices (excluding time for framework design, fabrication, and testing). Both efforts produced working chips competitive against state-of-the-art custom ASICs in terms of performance, power, and capabilities.

Contents

List of Figures

List of Tables

Acknowledgments

The PhD journey is not a solitary sojourn; rather, it requires one person to work with countless others, both past and present, to push the bubble of knowledge outward one iota in one direction. I would like to acknowledge the shoulders on which I stood and the arms that held mine during this adventure. Thanks to those mentioned here, and thanks to those not mentioned who also participated in my efforts. I could not have it done it without y'all.

Thanks to my undergraduate advisor, Professor Mircea Stan, and UVA Professor Ben Calhoun, for pressuring me to apply for PhD programs. And thanks to fellow UVA student Kevin Linger; our competitive views on exams and thorough exploration of SRAMs in our class project forced me to expand my knowledge. Kyle Powers, I hope when you took over my resilient adder project that you learned from my mistakes and improved upon the simple ideas of that project.

My first few years of graduate school were replete with classmates who all formed a tight-knit group of dedicated learners. Garen Der-Khachadourian, thanks for being my project partner and friend, helping me to feel comfortable in a new state and new school. Ben Keller, you still astound me with your efficiency; I learned a lot from you while we shared classes and projects. Pi-Feng Chiu, thanks for pushing me to attend core blast, and for sharing the learning of CAD tools during the Raven projects with me. Rachel Hochman, you helped me keep my interest in space alive, and I enjoyed our pre-radio lab Chipotle runs. Angie Wang, in you I finally found a more perfect perfectionist than myself. Nathan Narevsky, you're doing great. And thanks to all the other students in my year, including Jackie Leverett, Yongjun Li, Amanda Pratt, Bonjern Yang, Ozzy LaCaille, and any others I've forgotten.

Most of my learning came from mentoring by students and postdocs above my year. Brian Zimmer, you embody the ideal that all students hope to achieve. Thanks for your patience, teachings, and exemplary attitude. Yunsup Lee, hopefully one day I can keep up with your typing speed. Jaehwa Kwak, thanks for being the lighthearted one who taught me that work is not without play. Ruzica Jevtic and Miki Blagojevic, how you both put up with my inexperience I'll never know. Martin Cochet, you were the nicest project partner ever. And to my other instructors, Michael Zimmer, Yue Lu, Lingkai Kong, and those I have forgotten, it is because of you I've learned so much.

Other project and tape-out partners where invaluable in completing my program in a timely manner. These include John Wright, Jaeduk Han, Amy Whitcombe, Eric Chang, Zhonkgai Wang, Nandish Mehta, Colin Schmidt, Adam Izraelevitz, Edward Wang, Chick Markley, Keertana Settaluri, Woorham Bae, Albert Magyar, Alon Amid, Richard Lin, Howie Mao, and Paul Rigge.

The Berkeley Wireless Research Center provided the tools and support I needed to get through this program. So thanks to the students, staff, and faculty of the BWRC, especially James Dunn and Brian Richards. And thanks to various BWRC collaborators, both inside and outside Berkeley, including Calvin Cheng, Aaron Parsons, Hong Chen, Rick Raffanti, Robert Jarnot, Paul Stek, JPL, STMicroelectronics, Cadence Design Systems, and

Northrop Grumman Corporation. My projects were funded in part by the DARPA PERFECT project (HR00111320007) and Berkeley's ASPIRE program, NASA's Earth Science Technology Office's Instrument Incubator Program as part of the *Compact, Adaptable Microwave Limb Sounder* project, grant number NNX12AK39G; the DARPA CRAFT program (HR001116C0052); and the Intel iSTC on Agile Design (ADEPT).

Finally, thanks to my parents, Sandy and Tim, who continue to support me in all I choose to do.

Chapter 1

Introduction

1.1 Motivation

Signal processing system designers trade off algorithmic hardware design metrics like performance, area, and power for development time. The first choice is between off-the-shelf (CPUs or DSP-based processors), reconfigurable (FPGAs), or custom (ASICs) hardware [17]. Writing software for existing CPUs saves time and money, but such applications will never approach the low power, high performance, and small volume of specialized hardware often needed for embedded signal processing applications. And though FPGAs provide a convenient middle ground, high-end signal processing applications require high-end, application-specific integrated circuits (ASICs) to meet cutting edge specifications. However, designing custom chips imposes a burden on both the budget and the timeline of a project. Typically quoted non-recurring engineering (NRE) costs are in the tens of millions of dollars stemming from the complexity of design, verification, validation, and programming that takes many engineer-years and prototypes before calling the design finished. Figure 1.1 shows the tradeoff between development time and power consumed by a particular design from DARPA [1].

Figure 1.1 also shows a rough breakdown of where the development time goes and why it takes so long to produce custom hardware. First, the design itself consumes development time. There is no standard design framework above Verilog RTL that is universally accepted, so academic researchers and commercial design teams often develop their own in Python [18], Perl [19], SystemVerilog [20], C++ [21], and Simulink [22]. at different abstraction levels (RTL, HLS). And though attempts have been made, few frameworks outside of hardened IP couple RTL design with physical design, so even RTL reuse requires new physical design development. This is somewhat evident by Intel's tick-tock methodology [23], whereby moving to a new technology is equated in time with redesigning and optimizing the architecture. Second, verification suffers a similar lack of consistency, and it arguably consumes more work hours, sometimes as high as 70% [24]. In signal processing, a typical flow connects Verilog designs to MATLAB (or sometimes Python) golden models using SystemVerilog direct programming interface (DPI) function calls to a C program, which runs

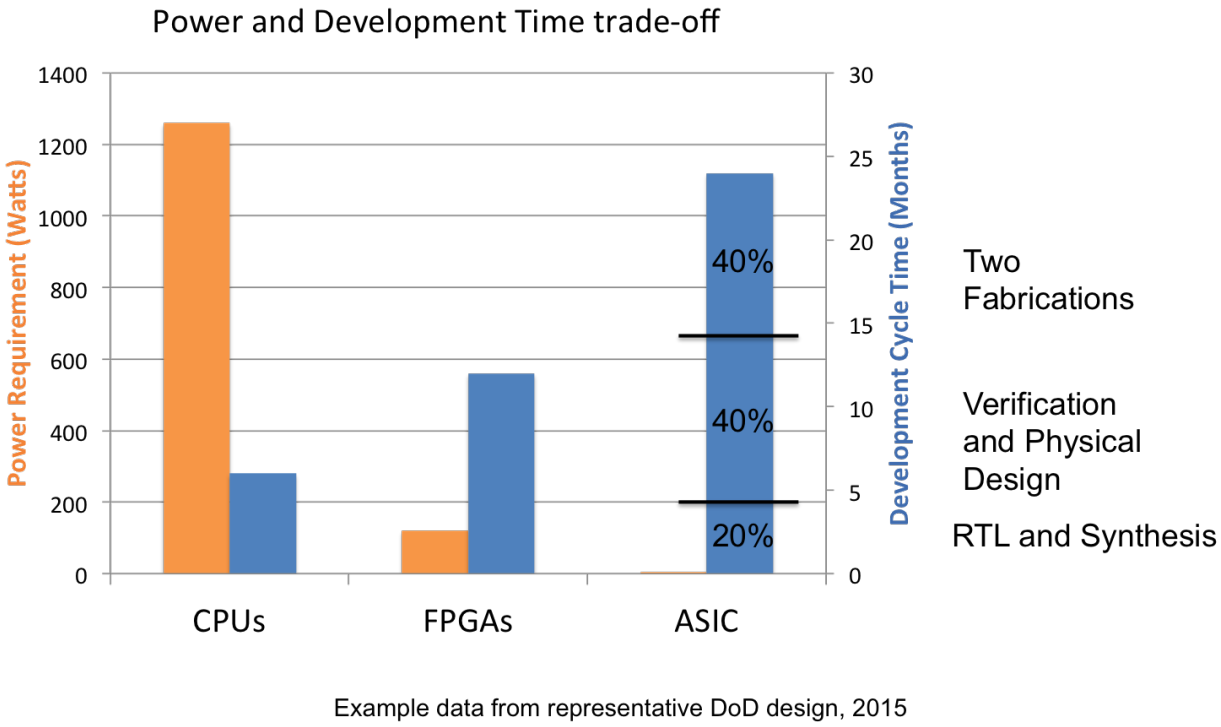


Figure 1.1: ASICs are much lower power than CPUs but require significantly more development time for a particular application [1].

MATLAB library routines [25]. However, this translation complicates the process and can lead to errors. Third, physical design flows take significant time to both set up and run. Even minimal design changes necessitate rerunning a full synthesis and place-and-route run, so much of CAD tool runtime is spent redoing the same optimizations [26]. Finally, fabrication and testing present time costs that, as bugs are found, require repeated fabrication and testing. Including sufficient design-for-test (DfT) structures and backup measures can reduce the time to find bugs and yield issues introduced by physical design and fabrication [27]. Entire tools exist solely to estimate the effort, cost, time-to-market, etc. of custom hardware projects [28].

One effort to tackle the latency of hardware design focuses on agile development, a phrase borrowed from software. The original agile manifesto came out in 2001 [29]. It promoted four values, which are copied below:

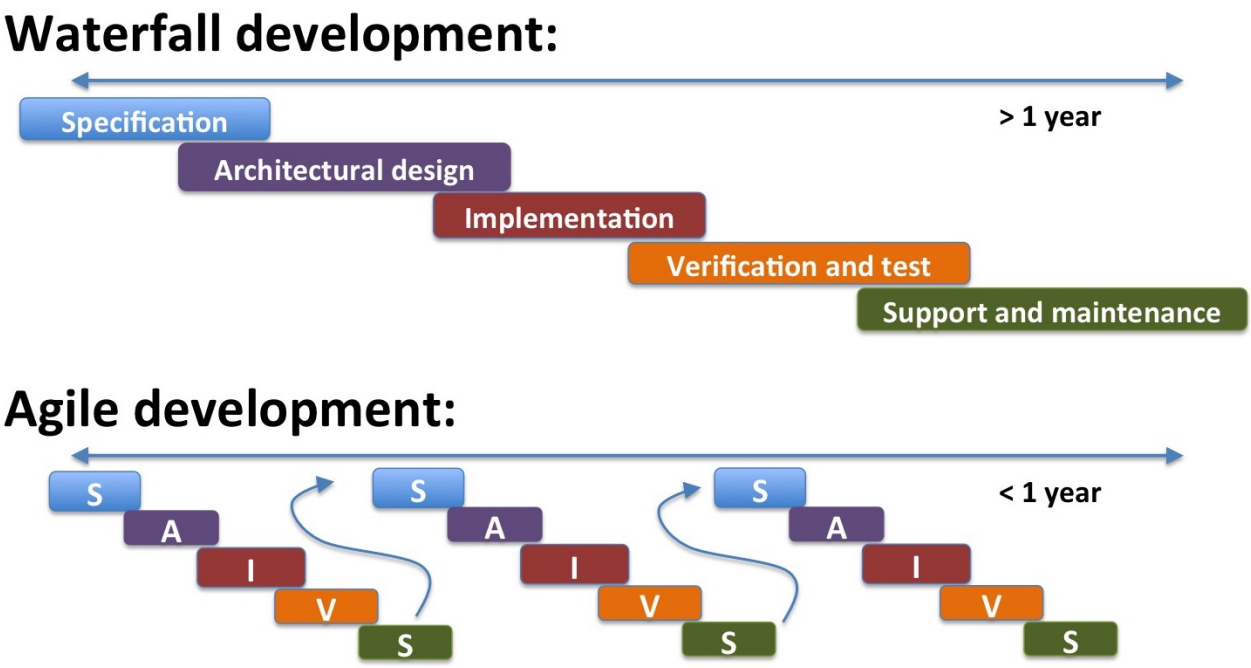


Figure 1.2: A comparison of waterfall and agile software development paradigms shows the differences in development cycles and project completion timeframes. Waterfall development uses a top-down approach and frequently takes over a year. Agile development uses a smaller, iterative approach with sufficient results completed in less than a year.

Table 1.1: Agile software development led to more projects finishing on time, where "finishing" means delivering a product that satisfies the customer [16].

Approach	On Time	Late	Canceled
Waterfall	10%	52%	38%
Agile	76%	20%	4%

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

These principles were in contrast to the traditional software development model, termed waterfall development. Waterfall development took a top-down approach, nearly completing each step in the flow before moving on to the next. This caused long project cycles, delays in product release, and sometimes even project cancellations. Agile development recognized the need for communication and adaptation, and projects following agile principles tended to finish on time much more frequently than those following the traditional waterfall style [16]. Figure 1.2 compares waterfall and agile software development styles, and Table 1.1 compares project success rates for each. It's important to emphasize that not all project completions are equal. Just because a project is declared finished and on time does not mean it meets all project goals, perform all the initially desired features, or contain complete documentation. Rather, features were incrementally added over time, adjusting to changing customer specifications. When the project deadline hit, a sufficient version was declared complete. But the agile argument declares this superior to a significantly delayed or canceled product, as, indeed, delay is expensive [30].

A form of agile development has been deployed in processor system design efforts. Companies have more recently used agile hardware development to varying degrees of success [31, 32]. This success hinges on adoption and agreement by the design team members on the details of agile development and how it applies to their work. Software and hardware are inherently different, as hardware product timeframes are heavily quantized on the order of about a year, and quick updates are all but impossible. So agile hardware design must still avoid show-stopping bugs and ensure sufficient verification coverage the first time around. Academics have attempted to solidify a modified form of the software manifesto into a set of principles more relevant to hardware development. These agile hardware development principles include (from [33]):

1. **Functional (albeit incomplete) prototypes** over fully featured models.
2. **Collaborative, flexible teams** over rigid silos.
3. **Improving tools and generators** over improving the instance.
4. **Responding to change** over following a plan.

Successful use of these principles has yielded complex SoC systems built within a limited timeframe by small teams of designers [33]. These 11 chips built over 5 years by a total of about 20 students and staff tested a new dynamic voltage and frequency scaling paradigm and CMOS silicon photonics designs in multiple technology nodes using and reusing a RISC-V

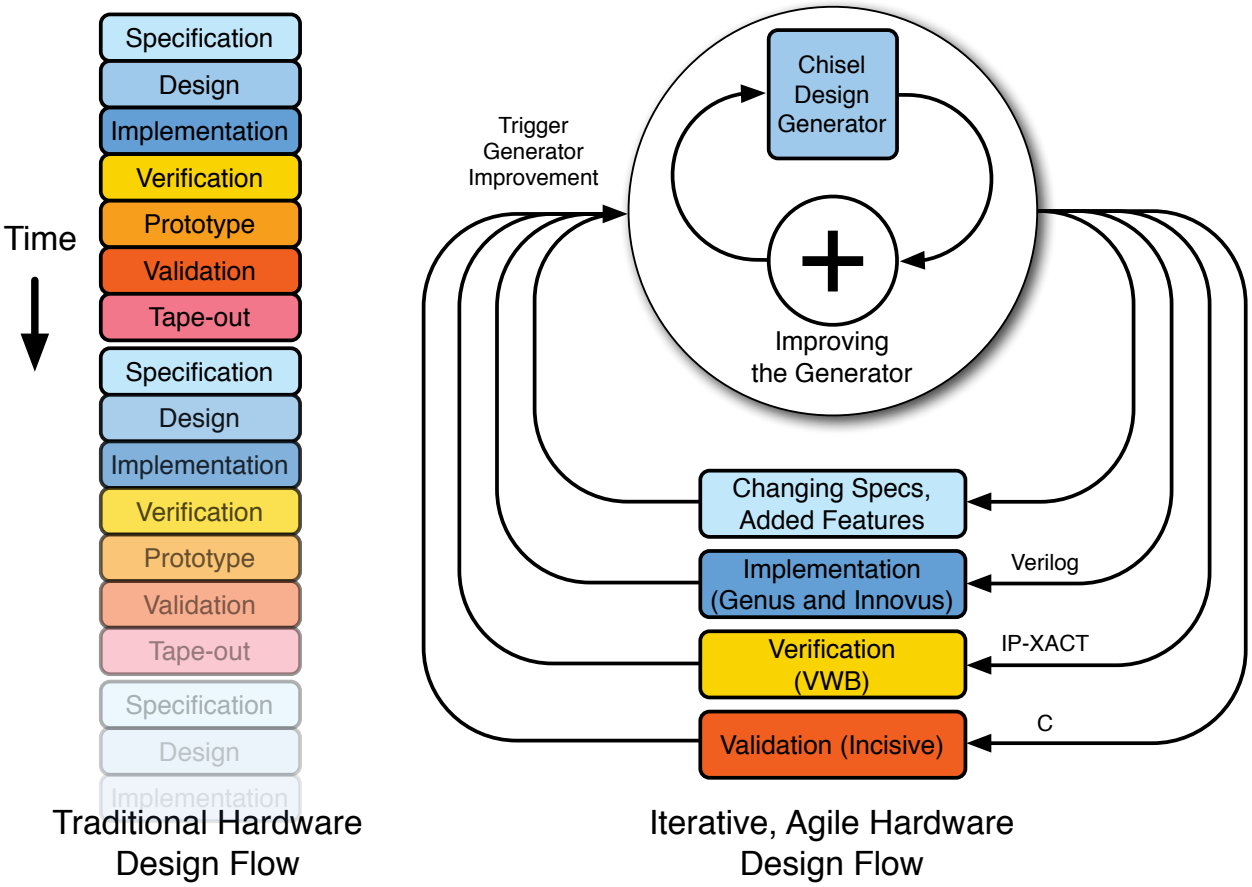


Figure 1.3: A comparison of traditional, open-loop design flow with our agile, iterative design flow. Changing specifications and issues during the process trigger generator improvement work. Design generators are also portable across designs, so these iterative enhancements persist.

central processor. Some of these team members have since gone on to apply their rapid design skills to the corporate world [34]. Figure 1.3 shows the traditional waterfall development style compared to an iterative, agile hardware design process used by the students. It's clear that such a flow responds to change more easily, and that solved issues become embedded in the design generator and propagate to future designs. Moving forward, agile hardware design, when applied correctly, is an effective tool for rapidly producing RISC-V processor chips.

Agile hardware development was previously applied to general purpose ASICs, but to date no efforts are known that successfully apply these principles to signal processing ASIC development. Though tools exist that support rapid development of signal processing ASICs. For example, Xilinx's System Generator for DSP [35] offers a graphical design interface, but it targets FPGA platforms, so results are often optimized for FPGA resources and not immediately suitable for ASIC implementation. MathWorks offers HDL Coder, which

generates RTL from MATLAB and/or Simulink models [36]. But without easy access to gate-level manipulations of algorithms, a user can easily write suboptimal designs that explode in size or suffer in hardware performance. Previous work has attempted to map designs from FPGA design environments to IC-suitable Verilog, which speeds up the design and verification process at the algorithmic level, but not at the ASIC level [37, 38]. It's common practice instead to design for ASICs and map to FPGAs for verification, as this ensures the hardware is optimized for custom circuits and not targeted to fit in the LUTs and BRAMs efficient for an FPGA. The DSP designs are typically accelerators tightly coupled with general-purpose processors [39] or dataflow models mapped directly into hardware [40]. Accelerators are more flexible but less performant and require more programming overhead.

In summary, ASIC signal processors are necessary to achieve the high performance and low power requirements of modern applications, but their long development time hinders their progress and ubiquity. A significant percentage of their development time goes into the design and verification of the architecture, with the remainder consumed by back-end ASIC flow work. Agile hardware principles, borrowed from a similar successful software approach and previously applied to general-purpose processors, offer a promising solution to signal processing SoCs. But such a solution has yet to be viably demonstrated on real systems running real applications.

1.2 Background

Traditional signal processing ASIC design, shown in Figure 1.4, follows a waterfall development model: the algorithm is designed, then this algorithm is mapped to an RTL implementation, and finally the RTL is synthesized and placed-and-routed to create a chip [2]. MATLAB or Python numerical libraries are used to explore algorithmic considerations, typically in floating-point precision. The translation to fixed-point introduces quantization errors, so the algorithm is adjusted to accommodate the added noise. Once a suitable fixed-point RTL implementation is ready, physical design begins. Issues with constraints like timing, area, and congestion further necessitate algorithm and RTL modifications. And eventually, once the chip is ready, the final product looks quite different from the initial concept. Also, the finished design works well for this application and technology, but porting the design to a new process or application generally requires starting the process over.

The rest of this section explores these pieces and previous attempts at improving productivity in the design of digital signal processing hardware.

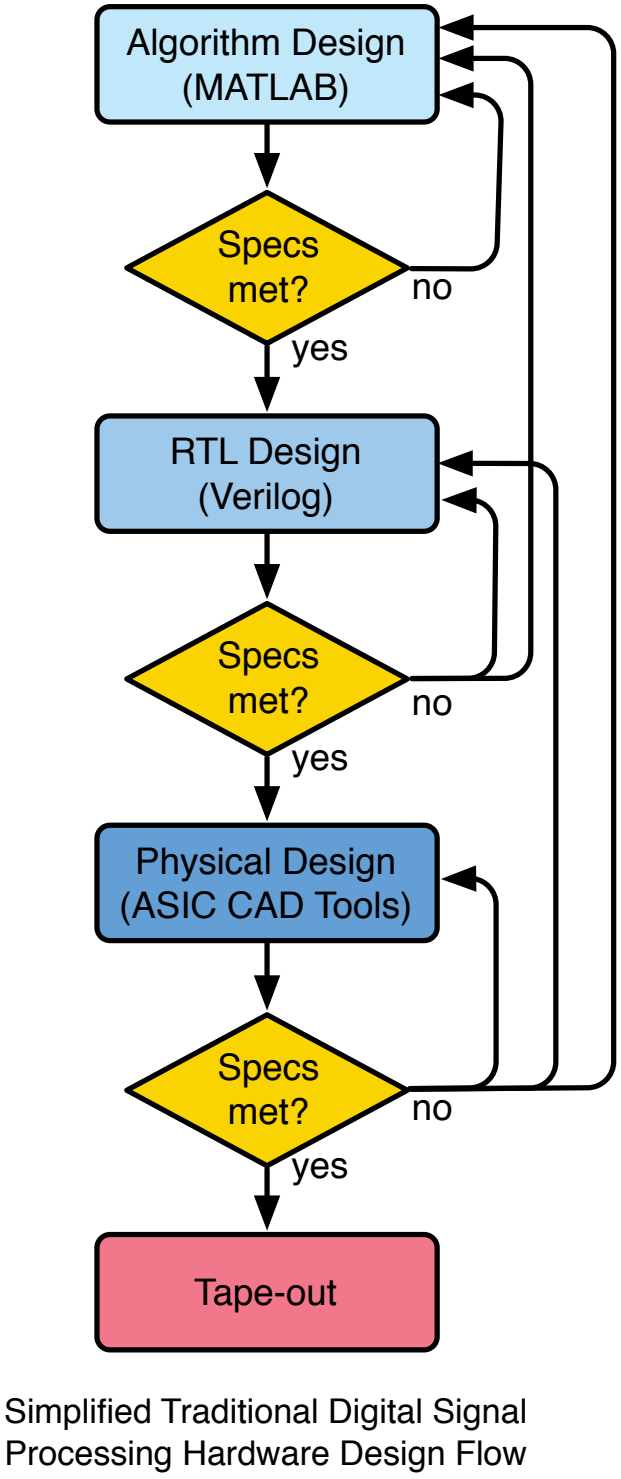


Figure 1.4: Traditional DSP hardware design produces an instance tailored to the specific application and technology. It is not easily reusable across applications and technologies [2].

1.2.1 Signal Processing Models

Specific applications are discussed in Chapter 2, but this section provides an overview of generic signal processing models. Algorithms are often designed in MATLAB or Python (especially with the NumPy and SciPy libraries) since they provide both a simple programming interface and pre-designed library components for signal processing. An algorithm defines which computational kernels are performed on which data, e.g. performing digital down conversion (DDC) requires mixing a signal with an LO to down convert it, then applying a low-pass filter to remove aliases. One algorithmic modeling technique represents these kernels as nodes in a flow graph, with edges representing data moving between kernels. Software models consider how the computation moves through the data, or how the data move through the computations. For example, DDC could be executed sequentially, with mixing and filtering performed one after another on each datum, or in parallel, with mixing and filtering performed simultaneously on a parallel set of data. The software model should be influenced by the hardware model. A general-purpose processor can handle either sequential or stream processing, though it will not take advantage of the data locality and consistent kernel sequencing of stream processing. Other hardware models specialize in stream processing, and will perform a DDC much more efficiently. Algorithms are often used as golden models for correctness verification, software models should efficiently "glue" algorithms and hardware, and hardware models should take advantage of commonly occurring algorithmic paradigms to optimally process signals. This section explores computational, software, and hardware models.

Computational Models

Signal processing computes on large, or sometimes infinite, data sets. However, the control flow for signal processing is often simple. These realizations permit a modeling approach that is focused on how data moves through computational kernels, rather than one that determines which computations to perform on the data. Kahn process networks do this by representing streams of data as edges in a directed graph, and computation as nodes. The most common subsets of these data flow graphs are synchronous data flow (SDF) and asynchronous data flow (ADF) [41].

Figure 1.5 shows an example of a simple SDF system. In synchronous data flow, nodes, sometimes called actors, transform input data tokens in some way to produce output tokens. Tokens are typically sets of data. To account for rate mismatches between consumption and production, FIFO buffers are often employed between actors. The data flow graph is synchronous because each actor consumes a fixed number of tokens and produces a fixed number of tokens each time it fires. This feature allows for static scheduling, so compilers can ensure FIFO depths are minimal and throughput is guaranteed without concerns for excessive backpressure. In asynchronous data flow, rates of production and consumption are not fixed, meaning they vary with data or time. Static scheduling is no longer possible, so runtime scheduling is required. This leads to much more complex control flows.

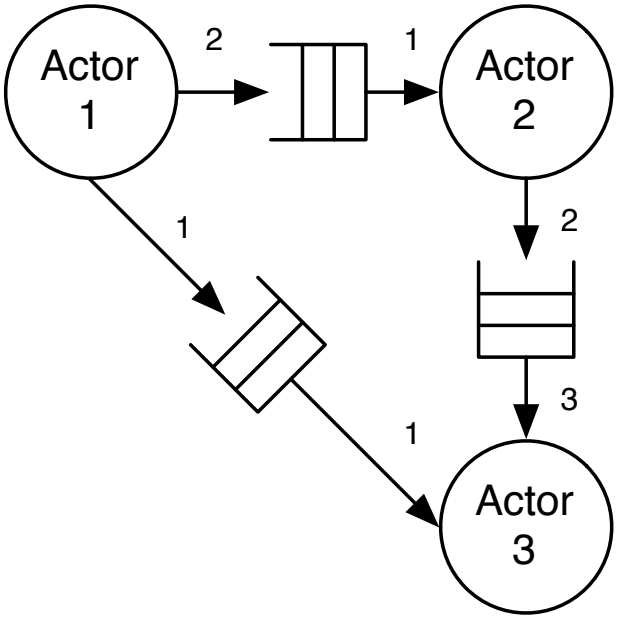


Figure 1.5: Simple synchronous data flow model showing actors as nodes connected through FIFOs as edges. Production and consumption rates are fixed and shown on the figure. Fixed rates support static scheduling.

More formally, a Kahn process network (KPN) is an example of a distributed or concurrent computational model. Operations happen in parallel as data moves through them. But a KPN cannot express complex control flow succinctly. Sequential models are one alternative, though they are less suited for signal processing. In sequential models, operations typically move through the data, where data-dependent control flow is easier to handle. One operation happens at a time, and data may or may not be accessed randomly. Examples of sequential models include finite state machines and Turing machines. Computational models are useful for describing algorithms in meaningful ways, but translating these models into something capable of being computed by a modern computer is nontrivial.

Hardware Models

The choice of hardware influences which software model is needed, thus we begin by exploring how to map from computational models to gates. A custom hardware mapping requires an extensive, low-level design effort but limits the amount of software needed. Reusing an existing, flexible hardware solution is also possible but requires a more complex software layer. Flynn's taxonomy condenses hardware models into dimensions of instruction concurrency and data concurrency. Extra dimensions, such as the memory architecture (centralized or distributed) and thread support add richness to the original taxonomy, allowing it to usefully model a wider array of architectures. Figure 1.6 shows the set of architectures in Flynn's taxonomy. Single instruction single data (SISD) performs a single instruction on a single piece

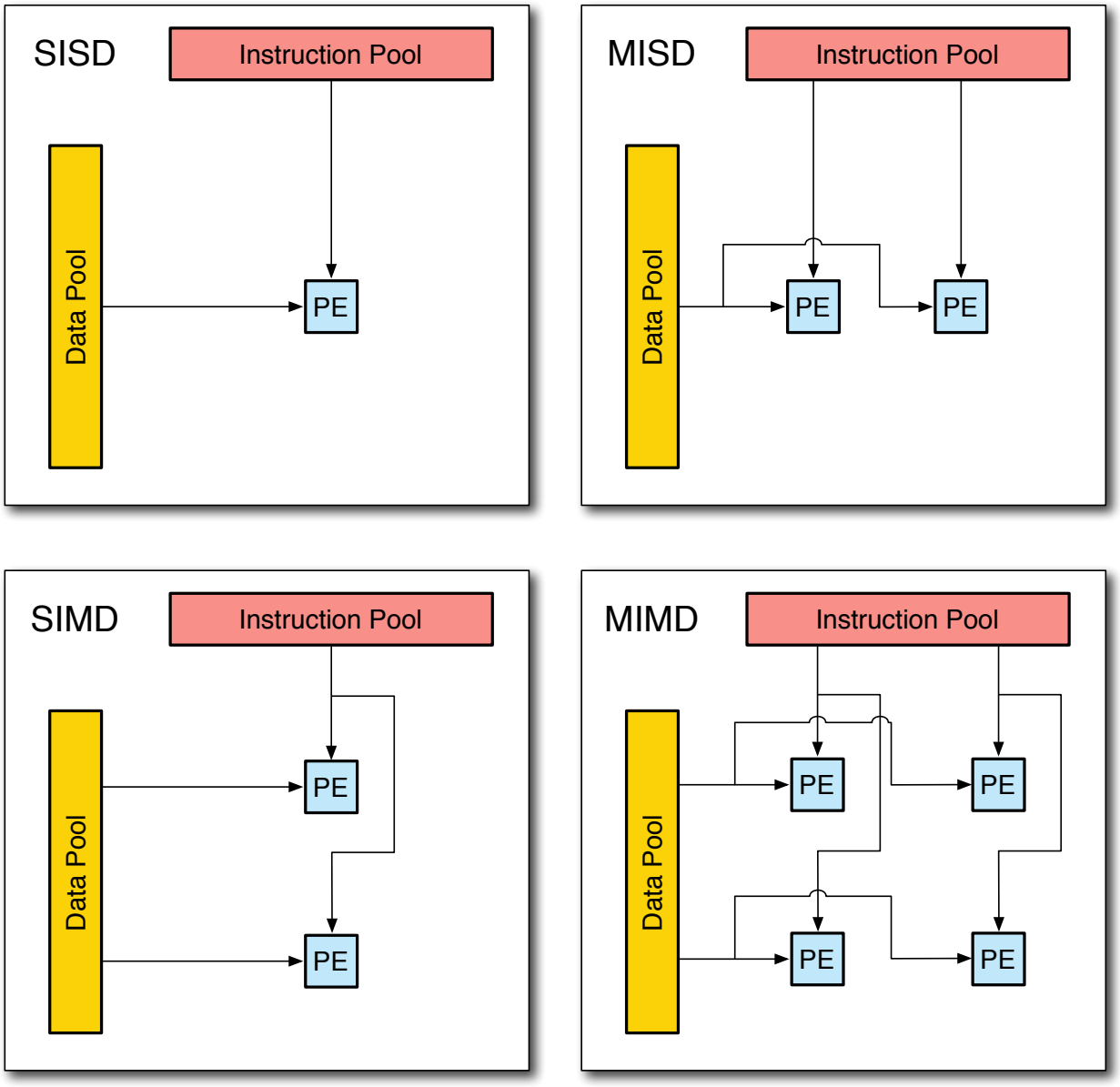


Figure 1.6: Flynn's taxonomy of computer architectures.

of data in every time step. Typical minimal processor cores are SISD architectures. Multiple instruction single data (MISD) performs a multiple instructions on a single piece of data in every time step. While rarely used, MISD supports applications requiring fault tolerance. Single instruction multiple data (SIMD) performs a single instructions on multiple data pieces in every time step. Examples of SIMD processors are vector machines and GPUs. Multiple instruction multiple data (MIMD) performs a different instruction on each data piece for each processing element (PE) in every time step. MIMD processors offer the most performance for targeted applications (modern supercomputers use MIMD architectures).

Centralized memory-based hardware operates on data stored in random access memory. Sequential computational models and sequential or vector software models map closely to memory-based hardware. Data are often stored in various locations in memory, so random access is needed to gather data before performing an operation on it. Thus data movement and indexing become significant computational overheads in memory-based hardware designs. However, complex control flow is feasible, so algorithms with heavy data dependence require memory-based hardware to execute. Signal processing in memory-based hardware is done either through ALUs performing multiply accumulate (MAC) operations, LUT-based arithmetic, or any other processing element [42].

Some general-purpose processors offload specific computational kernels to accelerators designed to quickly compute those kernels. A SIMD or vector co-processor is an example of an accelerator. Previous work showed that accelerating thirteen *motifs* would sufficiently cover the space of computational needs [39]. This approach is much more flexible, allowing a processor to compute a wider array of applications efficiently. In reality, the only accelerators modern processors contain are some kind of parallel computational accelerator (such as a GPGPU) and/or very specialized accelerators (such as an LDPC decoder or MPEG decoder). This is done because programming applications to target generalized accelerators requires expert knowledge, and tuning that application for a specific accelerator instance is difficult to automate [43]. Also, many applications are memory or IO bound instead of compute bound, so building memory access accelerators instead of compute accelerators may be more productive [39].

Data-heavy workloads and high-throughput requirements push architectures further toward specialization. Stream processors provide flexible signal processing kernel capabilities while optimizing memory for streaming data [3], as seen in Figure 1.7. At a high level, stream processors place ALU clusters or vector operation accelerations in series (a type of MIMD architecture), allowing data to flow from one to the next without the overhead of memory reads and writes in between. Such an architecture performs signal processing algorithms efficiently, but maximizing throughput and minimizing power requires a custom ASIC. Many application-specific processors exist to support common applications, but they lack flexibility.

Signal flow graphs (SFGs) are useful representations for how data moves through computational kernels. While these graphs map more directly to custom hardware, they can also be translated to SIMD or accelerator-based architectures. In this case, the same kernel (node on the graph) is computed in parallel on different data sets. Languages like Cuda and OpenCL

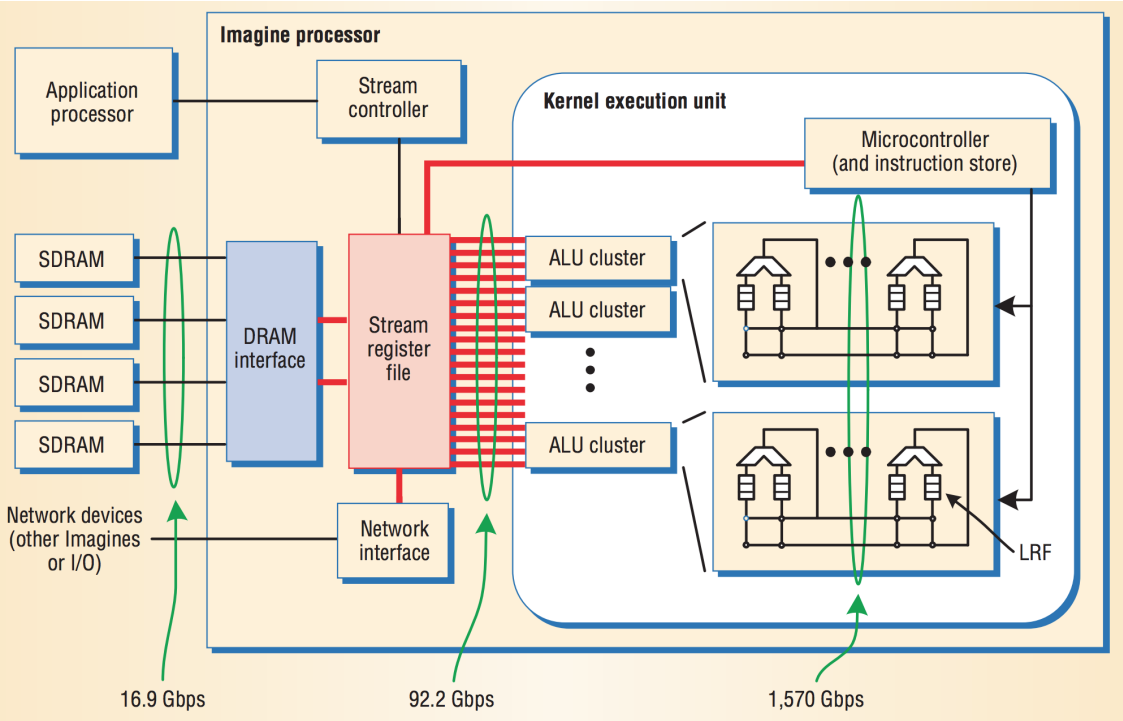


Figure 1.7: Stanford's Imagine stream processor. Bandwidths assume a core clock of 180 MHz [3].

support mapping of C code to SIMD architectures. But performance becomes limited when control flows diverge, such as when half the parallel graphs do not need to compute the current kernel. Whole Function Vectorization (WFV) is one automated approach to solving this through an automated compiler [44].

Mapping from a signal flow graph computational model to custom hardware is a multi-step process with iterative development and feedback, as illustrated in Figure 1.8. General purpose processors already contain reasonable hardware architectures, so this mapping assumes custom hardware is desired. Also, SFGs typically assume distributed memory and embed concurrency into the graph structure, so optimizing the hardware may require adjusting the SFG. First, the precision requirements of an SFG must be explored to determine appropriate data types and word lengths. Next, the SFG is mapped to a hardware architecture, ideally with reusable blocks. This step involves converting graph nodes into HDL models and graph edges into HDL interconnect. It requires verification that the hardware matches the algorithm. It also requires the exploration of timing, with concurrency within a node affecting algorithmic latency and throughput. Finally, the hardware model is ready for further mapping to an FPGA or ASIC. The rest of this section explores alternate hardware implementations which require a software layer to convert between the computational model SFG to the hardware model.

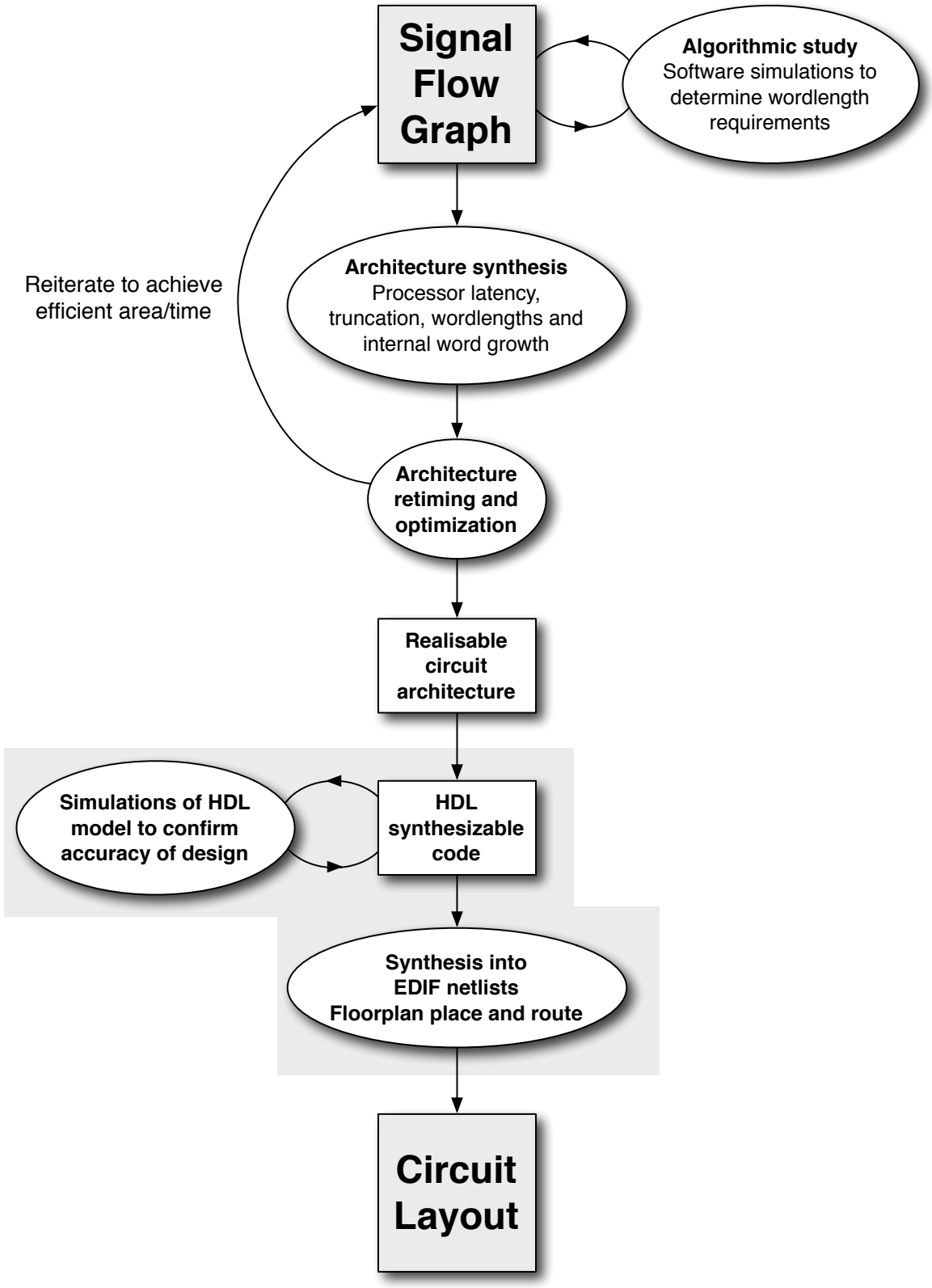


Figure 1.8: VLSI circuit design flow [4].

Software Models

Software models for signal processing algorithms typically fall into a few modes of concurrency, namely sequential execution, vector processing, and stream processing. Sequential execution performs one computation at a time, stepping through data and kernels one after another. Listing 1.1 shows pseudocode for sequential execution of an operation that adds two 100-element arrays, then squares the result. Ordering the operations as data-first or kernel-first takes the same amount of time because each operation is performed sequentially. However, the order may affect the maximum memory requirement.

Listing 1.1: Sequential software model

```
for (int i = 0; i < 100; i++)
    sum[i] = a[i] + b[i]
    sqr[i] = sum[i] * sum[i]
```

Parallel processing is usually a single-instruction multiple-data (SIMD) paradigm which performs the same operation in parallel on multiple data. A SIMD version of the previous code might look like Listing 1.2, where a 25-element vector operation is assumed available. Typical SIMD architectures fix the vector length and require vector-scalar operations to be expanded into vector-vector operations. Typical vector architectures provide variable vector lengths per iteration and support both vector-vector and vector-scalar operations [45].

Listing 1.2: Vectorized software model

```
for (int i = 0; i < 4; i++)
    start = i*25
    end = (i+1)*25
    vector_add(sum[start:end], a[start:end], b[start:end])
    vector_mul(sqr[start:end], sum[start:end], sum[start:end])
```

Finally, stream processing streams data through kernels, where allocation, synchronization, and communication between processing units is not handled by a CPU explicitly. This is equivalent to having multiple accelerators that support chaining. Processing units are initialized and data are gathered in the first step. Execution involves invoking kernels onto data sets. And at the end, results are returned, typically by "scattering" back to memory locations. Listing 1.3 shows pseudocode for a stream processing algorithm.

Listing 1.3: Streaming software model

```
kernel0 = kernel.init(add)
kernel1 = kernel.init(mul)
stream0 = gather(a)
stream1 = gather(b)
sum = kernel0.invoke(stream0, stream1)
sqr = kernel1.invoke(sum, sum)
```

1.2.2 RTL Design Frameworks

Translating algorithms and hardware models into synthesizable designs requires choosing an implementation language and paradigm. This section discusses the large number of programming and modeling languages available, and it explores the pros and cons of each. The discussed projects and languages accelerate the writing of synthesizable hardware by mapping higher-level constructs into lower-level RTL. Some of the ideas compiled here were first presented in [46].

High-level synthesis tools abstract away low-level implementation details from the algorithm to hardware translation process. HLS code is written in extensions to C, MATLAB, and SystemVerilog (see SystemC, ANSI-C/C++, Bluespec SystemVerilog [20]). For example, Xilinx Vivado allows designers to use C, C++, or SystemC to describe algorithms that are then compiled into RTL [47]. Bluespec uses guarded atomic actions, where actors fire atomically once some guard condition is met, which may or may not be the same as the SDF guard (input tokens are ready and output space is available). HLS is especially useful for designing blocks that require complex memory access scheduling, deep pipelining, and more generally, sequential logic. However, many DSP datapaths exhibit high degrees of parallelism that are best described structurally, necessitating that the SystemC code be carefully structured. Functionally correct algorithms often produce inefficient or even unsynthesizable code. Finally, hardware generation from recursive functions is generally not allowed, except through template metaprogramming.

MathWorks Simulink can save about 30% in design time [48] when systems are built from pre-existing IP by allowing users to pass intent to early stages of design and verification in a model-based design framework. It supports systems validation/verification with floating-point simulations and uses application-representative test vectors to optimize fixed-point bitwidths. The Simulink environment supports the "Chip-in-a-Day" methodology [49] but only accelerates a single instance's design for a single hardware platform, rather than accelerating *generator* design targeting *multiple* platforms. It lacks transparent, programmatic and easily extensible abstractions, requiring significant manual work to support multiple targets (FPGA/ASIC) and multiple data types. Unlike Simulink, HDL coder is platform-agnostic, meaning pre-existing IP and library designs map to a variety of FPGAs and custom ASIC designs. However, without difficult low-level manipulations, signal processing RTL is often insufficiently optimized [46].

High-level models and abstractions of data processing can simplify ASIC design efforts, but they present a large disconnect between the design and implementation of an algorithm. Generators support reuse of ideas and designer methodologies rather than IP instances, which often fail to meet the specific requirements of a new system. They also bridge the gap between algorithm design and RTL implementation better. Generator-based methodologies include simple constructs like `genvar` and parameters in Verilog, though this is insufficient to capture high-level designer intent. Perl and python scripts wrapped around Verilog (or even inside SystemVerilog, such as Stanford's Genesis2 [19]) are a cumbersome albeit common approach to extending parameterization and generation of RTL. Because Genesis2 relies on

two decoupled languages, there is a higher likelihood of generating difficult-to-debug syntax errors. Spiral is a generator specifically targeting DSP hardware [50]. Although powerful, Spiral is designed for static function generation and is not well suited for dynamic or reconfigurable architectures. New domain-specific languages (DSLs), such as Berkeley's Chisel [51] and Stanford's Magma [52], enable powerful processor generators but come with a steep learning curve [15]. But even IP generators are only as (re)useful as their interoperability. Efforts to standardize interfaces and IP metadata have produced AMBA protocols like AXI, APB, and AHB, and design XML files like IP-XACT [53, 54]. Existing commercial tools support automated verification of designs using these standards through the universal verification methodology (UVM).

Hardware construction/generation languages like Bluespec (closed-source) and Chisel (open-source) enable systems modeling, generator construction, and test environment creation all within a single, but powerful, underlying language with both functional programming and object-oriented constructs [20, 51]. They support (1) custom-defined types for high-level number abstractions, (2) stronger type-checking to catch errors at compile-time, (3) type polymorphism and operator overloading for hardware template parameterization, (4) recursive functions for hardware generation, and (5) functional constructs like map and reduce that help to concisely express highly structured data paths (more so than SystemC). Bluespec and Chisel are both architecturally transparent, exposing fine-grain implementation details for optimization that are typically hidden by HLS tools and allowing for more optimization opportunities.

1.2.3 System Design

Any digital design must interface with the real world eventually. These interfaces are typically analog designs, with common examples being PLLs for clock generation; high-speed serial links for communication; analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) for data domain conversion; low-dropout (LDO), switched capacitor, and boost/buck regulators for power supply generation; and antennas, amplifiers, and mixers for radio signal processing. Most analog designs are hand created and customized heavily for each application, but recent work in producing analog generators shows that alternative, automated design approaches are possible [55]. The digital generators presented in this thesis should be considered in a system context which includes analog generators, as exemplified in some later chapters.

Building and verifying digital integrated circuits is a difficult task, but no design is complete without test structures to prove functionality post fabrication. Good DfT requires

1. large coverage as quickly as possible
2. limited intrusion into the design
3. limited area, power, and performance overhead

4. useful feedback when tests fail

Typical test methods are *structural*, meaning they only care about the logic gates implemented and not the function of the circuit. The most common approach is through scan chains, which combine all or a subset of registers in the design into long chains when scan is enabled. Data may be shifted in and out, or propagated through combinational logic through toggling of the scan clock, resulting in extensive controllability and observability with limited overhead. Automatic test pattern generation (ATPG) seeks to create good input and output test vectors, where *good* may mean high coverage, fast runtime, localized error isolation, or a combination of these. This largely solves verification of the instance, where the fabricated design is tested for equivalence to the placed-and-routed model (which, hopefully, was verified to match the RTL logic).

Validation is the process of proving that the design meets the needs of the customer. Functional tests should be solved through validation routines, which depend on the design's intent. Providing functional test points in the system is one solution. These may be pattern generators to drive known data into the logic, or logic analyzers to monitor and snapshot data at key design boundaries. Having direct outside access to these test points is ideal, as any intervening logic could also be a source of error. Thus many designs implement debug modes and ports. On-chip validation support is necessary because involved validation routines often take too long to simulate.

1.3 Thesis Outline

This thesis solves the issue of slow signal processing ASIC design by creating a set of tools and methodologies that support agile development principles. It then applies those tools in the creation of several signal processing ASICs.

Chapter 2 explores signal processing applications and models. Before launching into a discussion on ASIC generators and creating a solution to designing custom hardware for digital signal processing, it's important to enumerate common applications to ensure a suitable landscape is covered. This chapter researches spectrometers, radio baseband processors, and radar transceivers.

Chapter 3 introduces a hardware model for signal processing accelerators. The model combines ideas of synchronous dataflow and accelerator-based processors to produce a hybrid SoC with specialized subcomponents. Standardized interfaces simplify composition and verification, and design-for-test structures support rapid development and debugging.

Chapter 4 shows a new generator that couples the algorithm and hardware models into a usable framework. Using Chisel, Scala, and commercial CAD tools, the generator boasts an array of automated checks and features such as address mapping, datatype checking, and DfT insertion. Included in this chapter are some open-source signal processing actor generators, which are used in subsequent chapters to implement real designs.

Chapters 5 and 6 demonstrate the generator in action with examples of real ASIC designs created from it. The first example is a spectrometer as an exploration of several of the

processing element generators. The second example is a radar receiver. Though, given the system's flexibility, it can act as a general-purpose signal analysis SoC.

Chapter 7 concludes this thesis by summarizing the key contributions and mapping out future areas of research to explore.

Chapter 2

Signal Processing Applications and Models

One of key contributions of this thesis is a set of generators that produce SoCs targeting signal processing. These types of generators can support applications in a variety of domains. In particular, we target applications with strict performance goals that require specialized signal processing hardware. Many of these signal processing frameworks share a common set of computational kernels, but require a different parameterization of these kernels for each application. In this chapter, we present three different domains with signal processing applications that can benefit from parameterized custom hardware generators.

2.1 Astronomical and Atmospheric Spectroscopy

Space-based measurements are necessary for observations outside of atmospheric windows, since the Earth's atmosphere blocks certain frequency bands of light, as seen in figure 2.1. SoC spectrometer implementations provide a low-power, tightly-integrated, rad-hard solution to space-based, balloon-borne applications.

2.1.1 Algorithms and Instruments

Astronomical spectroscopy measures existing emissions, so no transmitters or actuators are needed. The simplest spectroscopy just requires taking the fast Fourier transform (FFT) of measured data, but often more processing is needed. For example, antennas operating in the THz bands look for molecular emissions, then mix the results down to baseband. Because of the high-frequency nature of these signals, this mixing is done with discrete analog components. But baseband processing can be done digitally by a CPU, FPGA, or ASIC. Measured signals contain a very low signal-to-noise ratio (SNR), so, to compensate, spectrometers accumulate spectral measurements for many seconds. To avoid gaps in measurements and the latency of sweeping frequency bands, a wide-bandwidth system is preferred. And appropriate

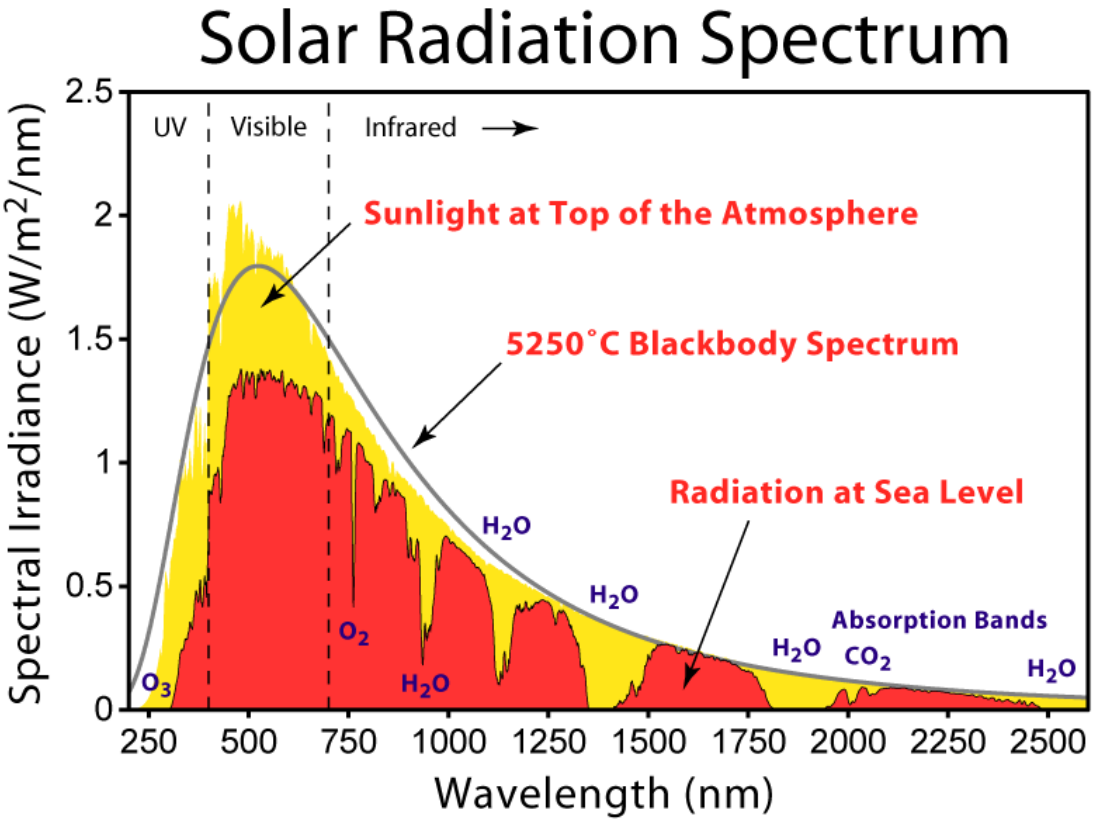


Figure 2.1: The Sun emits blackbody radiation, which is intermittently absorbed by particles in the corona and in Earth's atmosphere [5].

spectral resolution of wide bandwidth signals requires a large FFT. Also, spectral leakage is a sampling phenomenon where power in one spectral bin leaks into neighboring bins. A rectangular spectral window can significantly reduce this leakage, but the windowing is typically done with a sinc function in the time domain using polyphase filters (parallel finite impulse response, or FIR, filters) [56]. Finally, mixing high-frequency signals down to baseband folds the upper and lower sidebands on either side of the center frequency on top of each other. To prevent this, sideband separating systems measure both I and Q quadrature signals, which may be treated mathematically as one complex-valued signal. These signals are processed either together as one complex signal or separately as two real-valued signals with a merge occurring in post-processing.

Instruments targeting terahertz emission spectra measurements must be low power and low weight since they typically fly on weather balloons or satellites. Thus, for digital processing, FPGA and ASIC implementations are preferred, with cost and development time preventing all designs from using ASICs. Figure 2.2 shows an example full system digram of a new satellite terahertz receiver, and Figure 2.3 gives more detail on the digital spectrometer design. The polyphase filter bank (PFB) combines a polyphase filter with an FFT.

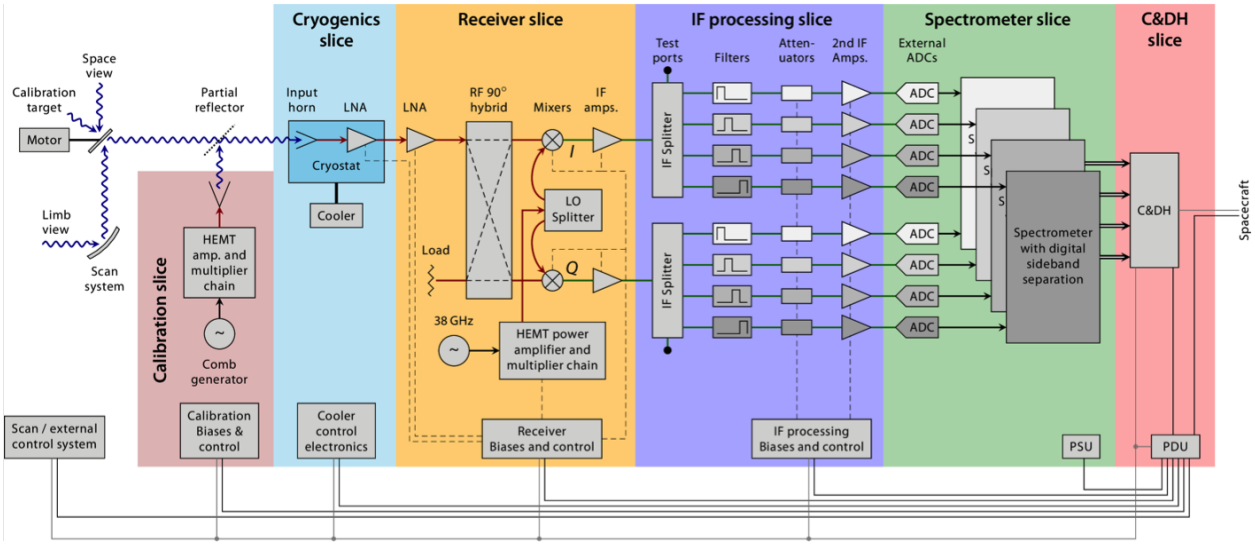


Figure 2.2: A full satellite terahertz receiver design for the compact adaptable microwave limb sounder (CAMLS) project [6].

The nodes labeled C1-C4 are complex multiplies with calibrated coefficients, typically in a lookup table (LUT), used to fix mismatch in the analog front-end components. Also needed are power and integration elements. Except for the calibration coefficients, the rest of the digital processing components have essentially no runtime programming requirements, so no live control is needed. This datapath is expected to run continuously, and latency is not constrained because the instrument is only measuring. Thus long pipeline depths can be used to improve throughput as needed.

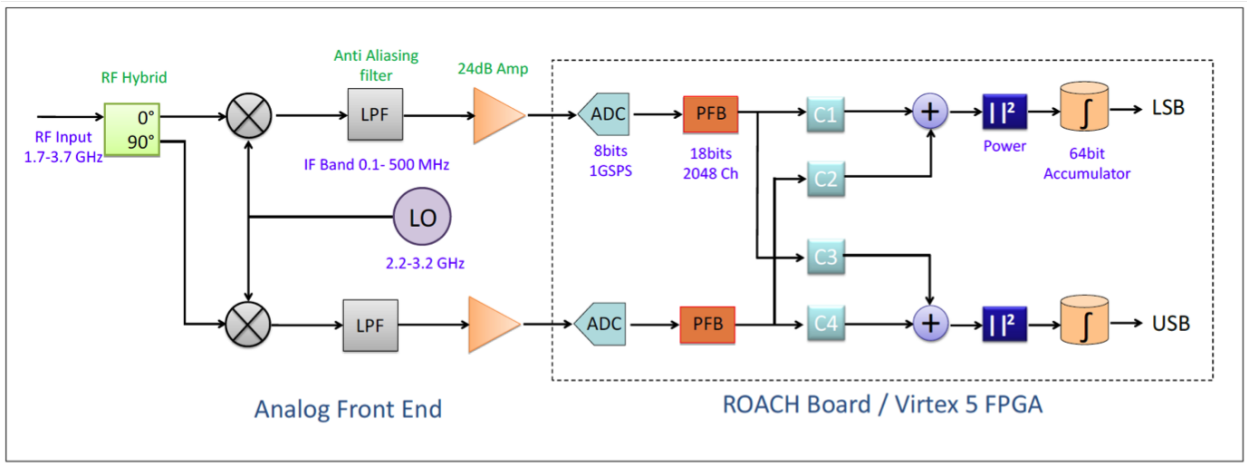


Figure 2.3: The receiver block diagram for the CAMLS instrument, a modification of a previous design [7]. Analog and digital blocks amenable to ASIC implementation are in the dotted box.

2.2 Radio Baseband Processing

Unlike astronomical spectroscopy, radio baseband processing relies on known signals generated relatively locally. Latency is a much larger concern, as communication is typically both ways and the user's experience is on the line. Digital and analog encoding is used to improve signal-to-noise ratio and increase data rates amidst noise from the environment and interfering signals. Since both the transmitted and received signal are controlled by the standard, forward error correction (FEC) and symbol framing/packetization help improve communication performance. Heterodyne signal processing with multiple carrier frequencies is done by modulator demodulators (MODEMs), as described here. Given the variety of standards, some systems use software-defined radio (SDR) to provide a software-reconfigurable interface to processing radio signals.

To simplify discussion, this thesis investigates two of the most common radio communication standards in use today, WiFi and 4G LTE. WiFi here is discussed as IEEE standard 802.11ac and is commonly used in local area wireless networks. 4G LTE (long-term evolution) is a standard by the 3GPP (3rd Generation Partnership Project), specifically Releases 8 and 9, defining wireless cellular communication networks. We ignore LTE Advanced, a more performant extension to traditional LTE that supports higher bandwidths using multi-carrier access. Table 2.1 compares the properties of WiFi and LTE standards. Both use versions of orthogonal frequency-division multiplexing (OFDM) (such as multiple access, OFDMA) modems. OFDM divides the space of frequencies into sub-bands or channels, with each user getting one channel. OFDMA assigns multiple subsets of channels to users, providing frequency diversity to improve SNR.

Table 2.1: A comparison of properties in the 802.11ac WiFi and LTE standards.

Parameter	**802.11ac**	**LTE**
Bandwidths	20, 40, 80, 80+80 MHz	1.4, 3, 5, 10, 15, 20 MHz
Sample Rates	20, 40, 80 MHz	1.92, 3.84, ..., 30.72 MHz
Useful Symbol Length	3.2 us	67 us
FFT Sizes	64, 128, 256	$2^n 3^m 5^k$
Cyclic Prefix	0.8 or 0.4 us	5.21 or 16.67 us
Modulation	BPSK, QPSK, 16-QAM, 64-QAM, 256-QAM	QPSK, 16-QAM, 64-QAM
Error Correction	Convolutional/LDPC	Convolutional/Turbo/Hybrid ARQ
Synchronization, Channel Estimation	Preamble, Pilots	Embedded Pilots, Control Channels

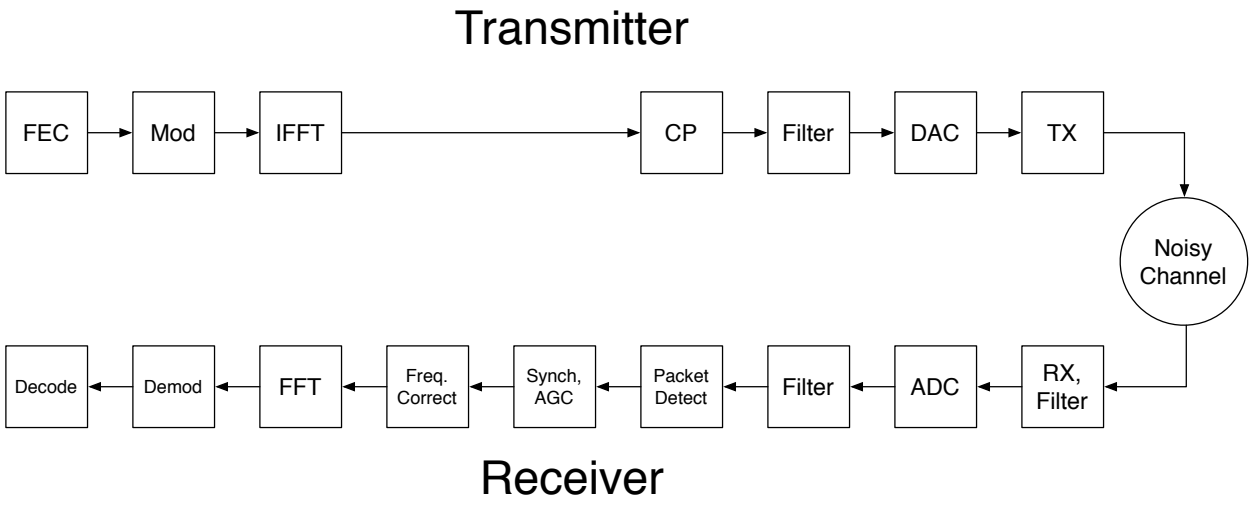


Figure 2.4: A simplified block diagram of a typical OFDM modem.

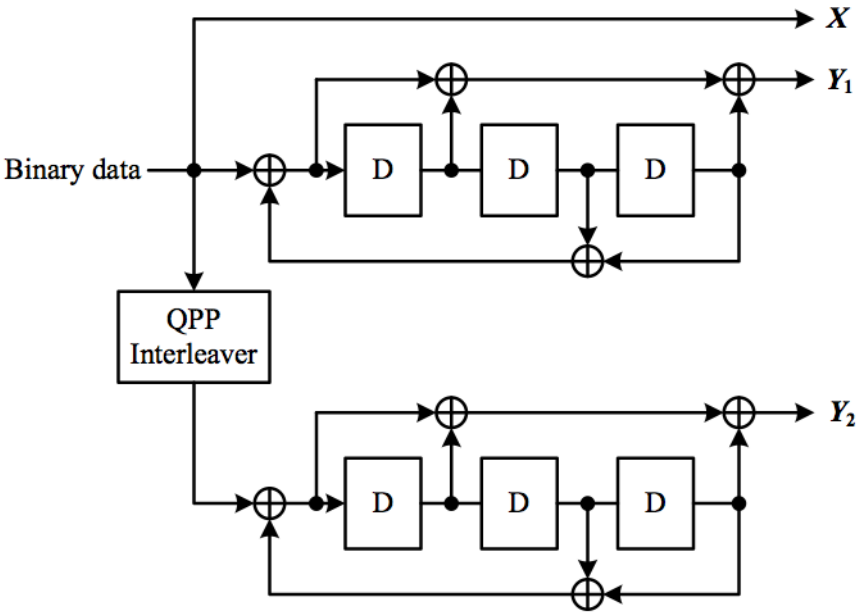
2.2.1 An OFDM Modem

Figure 2.4 shows a generic datapath for an OFDM modem, including both transmitter and receiver architectures. A modem is typically divided into layers, with the media access control (MAC) layer defining how the data are packetized, and the physical (PHY) layer defining how the bits are FEC encoded and transmitted/received. The blocks in Figure 2.4 are PHY layer operations. Older systems separated the two layers, making different versions modularly composable as needed. Modern systems combine the two layers into a single ASIC or SoC. As seen, the transmitter path includes significant digital signal processing before conversion to analog and transmission. The receiver filters the received data, then performs the opposite of the transmit processing. The rest of this section goes over each processing step mentioned in Figure 2.4 except the IFFT and FFT, which simply convert between the time and frequency domains.

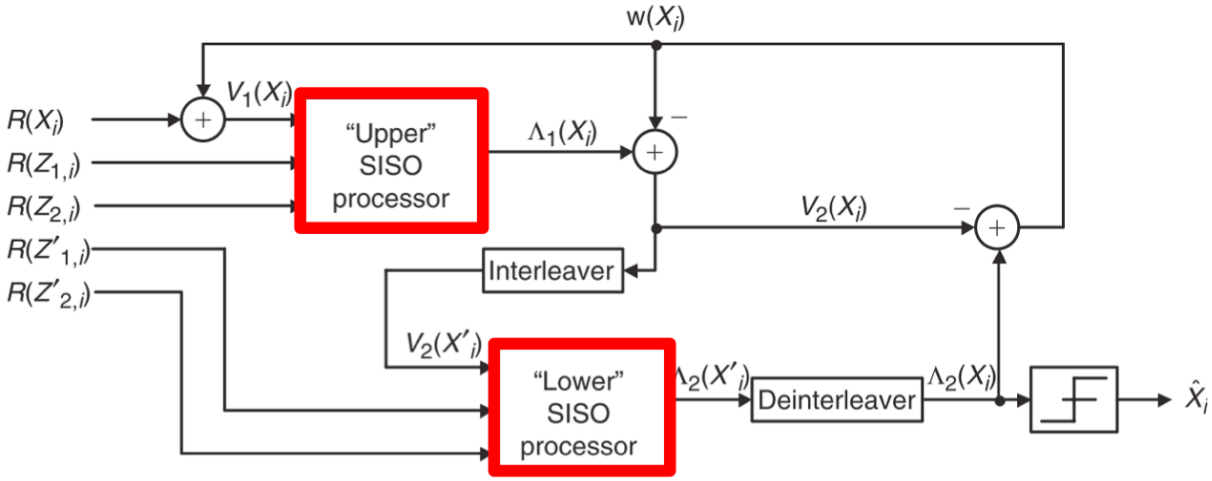
2.2.2 Forward Error Correction

Electronic and environmental noise introduce errors in analog signals, limiting the theoretical capacity of a noisy channel to pass information. Forward error correction adds redundancy to a data stream so that receivers can detect and correct errors introduced by a noisy channel. Typically, encoding is much simpler to implement in hardware than decoding. Encoding is done on raw bits, while decoding uses soft estimates, which require more bits. This section discusses Turbo and LDPC codes, which are two of the most common capacity-approaching codes and are used in WiFi and LTE.

Turbo codes are a variation on convolutional codes which perform a finite convolution of the input data sequence with itself in a sliding fashion [57]. In the 4G standard, parallel convolutions of the input and scrambled input produce an output sequence with code rate



(a) An example encoder for an LTE turbo code [57]

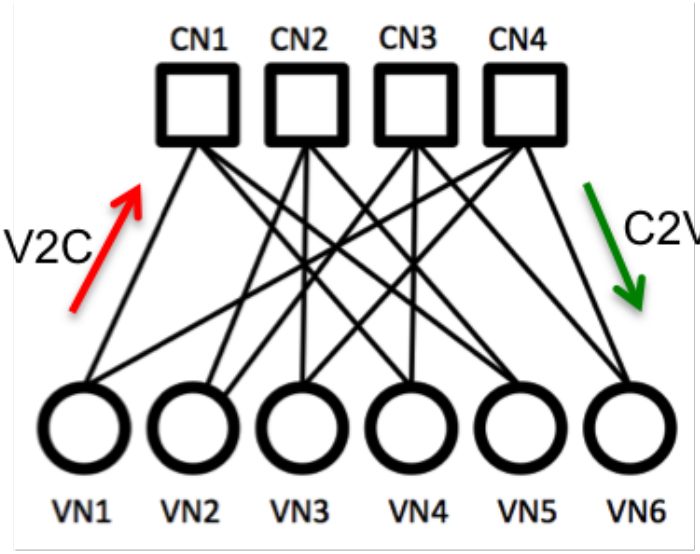


(b) Turbo decoder with SISO units. X_i and X_i' are the systematic bits in de-interleaved and interleaved order, respectively. Z_1 and Z_2 are the encoded bits, and their primed counterparts are in interleaved order. Each SISO block is considered one half-iteration of the decoding algorithm.

Figure 2.5: LTE turbo encoding and decoding block diagrams.

0	-	-	-	0	0	-	-	0	-	-	0	1	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	0	0	-	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	0	0	-	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	0	-	-	-	-	-	0	0	-	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	0	0	-	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	0	0	-
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	0	0	
3	-	-	-	16	-	-	2	25	5	-	-	1	-	-	-	-	-	-	-	-	-	-	0

(a) One parity check matrix for 802.11ac code. The block length is 648 bits, rate is 1/2, and sub-block size is 27 bits. A non-blank entry represents a cyclically shifted identity matrix.



(b) Factor graph representation of a simple linear block code, with circles representing variable nodes and squares representing check nodes [58].

Figure 2.6: LDPC parity check matrix and message-passing decoder diagram.

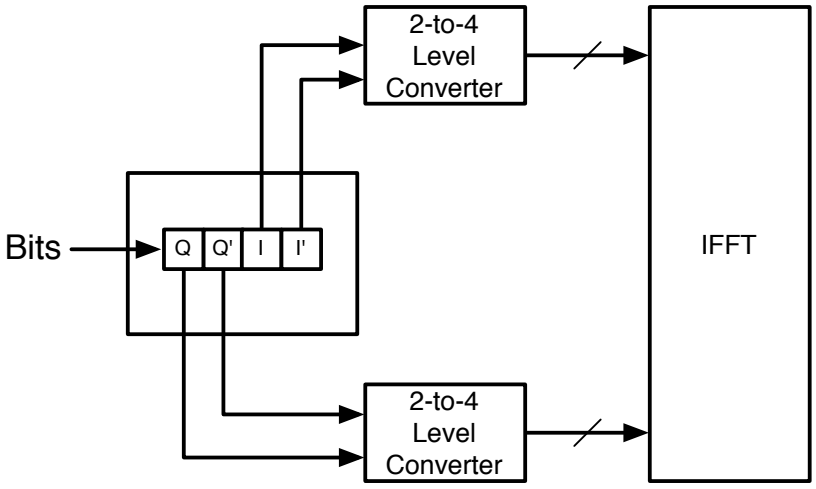
1/3. Figure 2.5 shows an example encoder with 8-state convolutions. The idea is that each state (set of three outputs) can only map to a limited set of next states, and sequencing these state transitions (e.g. in a trellis diagram) helps predict correct states amidst errors. Decoding takes soft input estimates and produces soft output estimates using an algorithm that assigns probabilities to state sequences. The decoder looks similar to the encoder, with the same interleaver, though the soft-input soft-output (SISO) processors perform the BCJR algorithm instead of convolution. Details of the BCJR algorithm are omitted here, but suffice it to say it can be implemented in hardware with reasonable overhead. A decoder diagram is seen in Figure 2.5.

Low-density parity check (LDPC) codes map long blocks of data to redundant code words by multiplying with a low-density parity check matrix, similar to a Hamming code [58]. Most of the entries in the parity check matrix are zero, and non-zero entries represent cyclically shifted identity matrices. These two optimizations simplify hardware implementations. Encoding requires multiplying input words with the parity matrix, an example of which is in Figure 2.6. Decoding involves iteratively passing soft information between variable and check nodes (encoded and decoded words) to guess the decoded word until a solution is reached. Connections between check and variable nodes exist if the parity matrix is nonzero between them, as seen in Figure 2.6. An estimate of the decoding confidence allow the algorithm to terminate early if few to no errors are present. Thus decoding takes an indeterminate number of cycles. Similar to turbo decoding, the operations performed at the check and variable nodes (an offset min-sum algorithm) can be implemented in hardware with reasonable overhead.

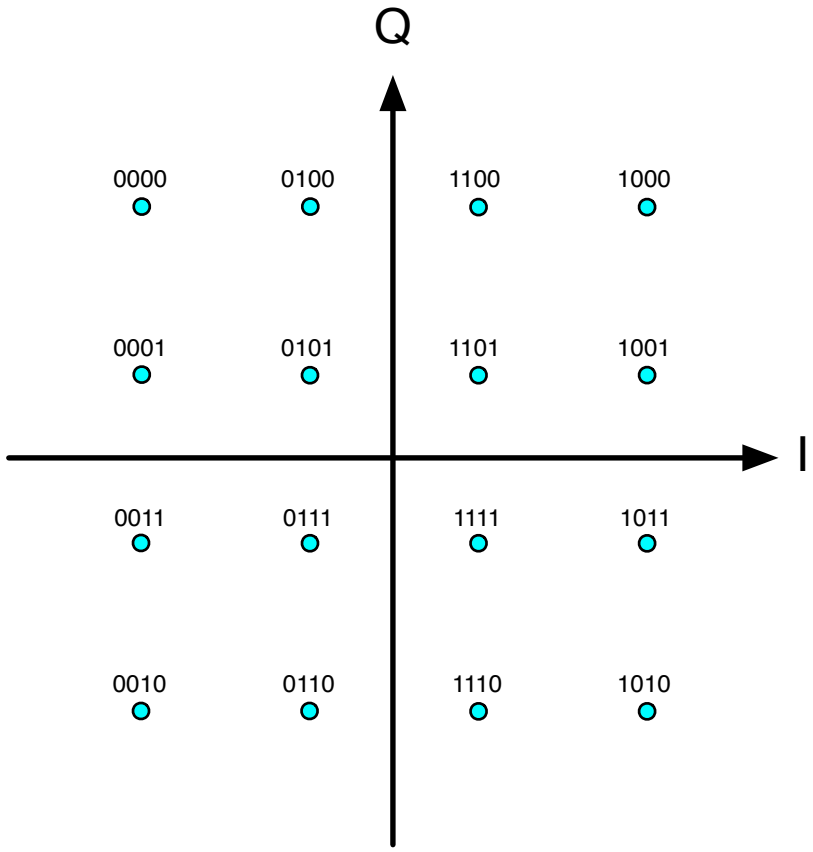
2.2.3 Digital Modulation

Digital modulation techniques encode bits into the phase and/or amplitude of the transmitted signal. Different modulation schemes trade off data rate for signal robustness. Phase-shift and amplitude-shift keying (PSK and ASK) are common in WiFi and LTE, while frequency-shift keying (FSK) is only used in low-frequency communication channels. Quadrature amplitude modulation (QAM), despite the name, usually incorporates both PSK and ASK.

BPSK encoding simply modulates a sine wave with plus and minus one. Encoding of an N-QAM signal above $N = 2$ requires alternately splitting digital bits into I and Q components, then performing amplitude modulation on each one to achieve a constellation of values. Later, these values are multiplied by in phase and quadrature phase signals, then added together. Sometimes pulse shaping is included to smooth out the results. Figure 2.7a shows a block diagram of an encoder, which outputs a constellation like that shown in Figure 2.7b. Decoding works much in the opposite way as encoding. Incoming data are digitized, split into I and Q components, multiplied by 90^o-offset sinusoids, then matched filtered. Matched filtering involves convolving the input with an ideal symbol pulse (i.e. multiplying by an unscaled, time-reversed signal). The results are thresholded and interleaved to form the received data. OFDM



(a) Example of a digital 16-QAM modulation block diagram scheme.



(b) A 16-QAM modulation scheme produces a constellation diagram like the one shown here.

Figure 2.7: LTE turbo encoding and decoding block diagrams.

2.2.4 Other Processing Blocks

The remaining transmitter processing blocks perform filtering and provide a cyclic prefix, which is the end of a symbol inserted at the beginning. This allows the receiver to perform circular convolution on the received signal, which is computationally easier, but requires it to discard the cyclic prefix. The remaining receiver processing blocks perform filtering and correction for any non-idealities in the signal such as carrier frequency offset (CFO) and frequency, phase, and gain errors. These errors arise from channel imperfections or, in the case of CFO, local oscillator mismatch.

Not shown is a sensing receiver datapath, in which input data are spectrally analyzed without any demodulating or decoding. This allows cognitive radio and spectrum sensing applications to monitor the spectral state of the environment and act when channels become available. A sensing datapath contains just filters, a Fourier transform, and often a received signal strength indicator (RSSI).

2.3 Radar Transceivers

The third application explored here is radar transceivers. Radar is the process of emitting and detecting electromagnetic waves, measuring how the wave is transformed by an object to learn properties of that object. Typical applications include the detection and locating of objects and measuring the motion of moving objects. Ground-based radar finds airplanes, missiles, rain, and other airborne objects of interest. Radar is also added to manned and unmanned aerial vehicles to detect other airborne or grounded objects.

Early radar instruments used simple pulsed signals, measuring time-of-flight to measure distance. Modern radar uses modulated pulses or continuous wave signals between the ultra-high frequency (UHF) and millimeter wave bands, or 0.3 to 100 GHz [59]. Linear frequency modulated (LFM) pulses, also known as chirps, and continuous-wave frequency-modulated (CWFM) signals are the most common in use today. Chirps reduce the ambiguity of measured distances, and CWFM allows finer time resolutions as the transmitter and receiver can both operate at the same time. Signals vary in duration, magnitude, and frequency depending on the desired measurement, and beamforming is used to improve angular resolution.

A typical radar digital receiver is seen in Figure 2.9. Digital down conversion requires mixing the received signal with a complex sinusoid to reduce the frequency and data rate of later processing stages. It also includes a low-pass filter for removing the extra higher-frequency tone generated during mixing. Pulse compression convolves the received data with a time-reversed version of the transmitted pulse. This calculation is performed either in the time domain using a finite impulse response (FIR) filter or, more commonly, in the frequency domain by simple multiplication. Frequency-domain pulse compression requires a forward FFT first, and an inverse FFT after the pulse compression. Doppler compensation, sometimes called range cell migration correction (RCMC), shifts frequencies bin-by-bin to account for Doppler shift. Coherent integration is an accumulator that sums the complex

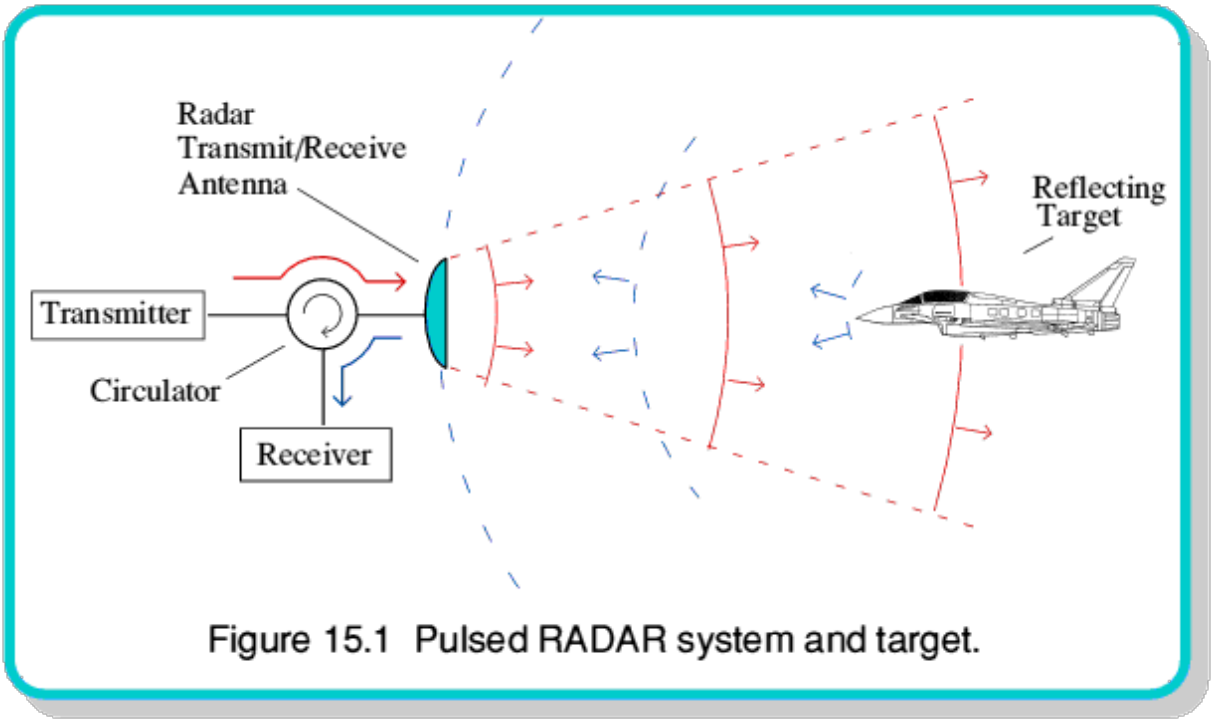


Figure 2.8: A radar system transmits electromagnetic pulses and measures return time to detect objects and determine their distances [8].

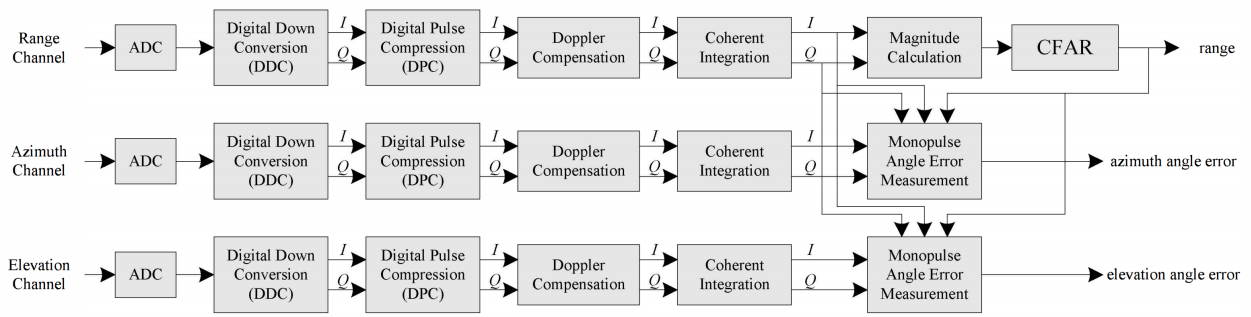


Figure 2.9: Typical radar receiver block diagram, here showing multiple receive channels with each one processed similarly [9]. Computation after coherent integration is slow enough to be done in software.

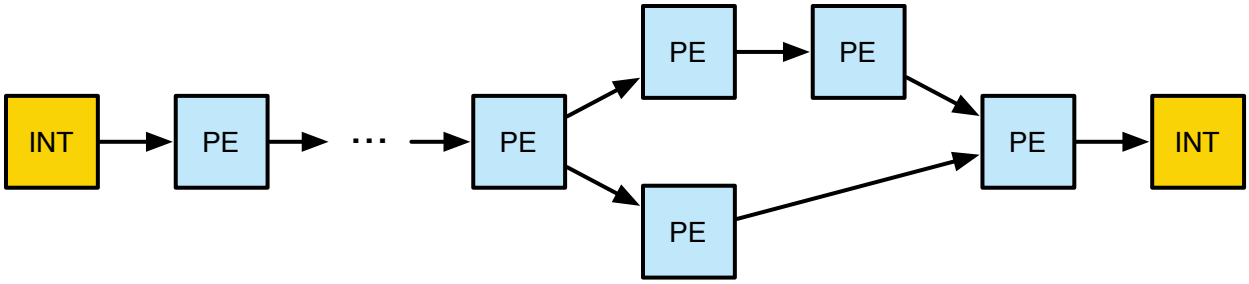
frequency-domain results from multiple received signals. Since this reduces the data rate, further computation is typically done with software.

2.4 A Common Signal Processing Algorithm Model

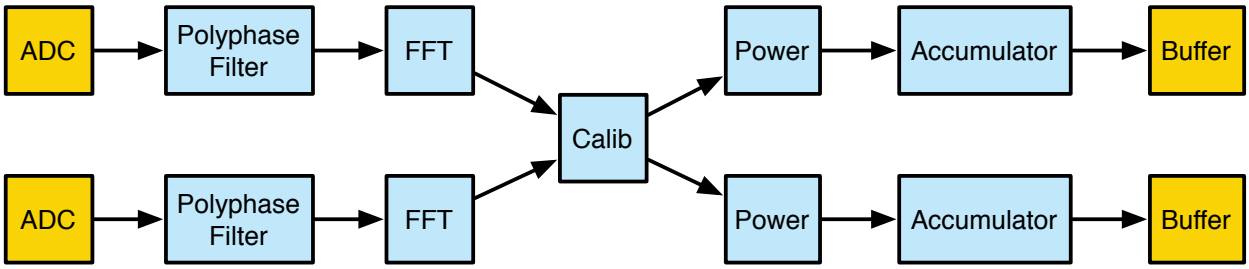
The applications mentioned in this chapter represent a range use cases and requirements. Some require transmitters while others do not. Some prioritize latency, while other prioritize throughput. Each process data in different ways, but similarities in computational methods exist. For example, all three applications convert the time-domain data into frequency-domain data, most often done with an FFT. Also, data tend to flow in one direction. Elements that need sequences or packets of data can buffer them up internally, but rarely does information from some downstream processing affect upstream processing in real time. These realizations suggest we can design a cohesive model that encompasses many common signal processing algorithms.

We propose a model that is reminiscent of synchronous dataflow [41]. While data rates may be data dependent, such as FEC decoders terminating early when SNR is high but taking longer when SNR is low, data flow is static. Flow proceeds along well defined paths, and real time feedback is not allowed across processing elements. This simplifies control, and data rate analysis does not have to contend with loops. Figure 2.10 shows an example datapath for this model, where PE stands for processing element. Each processing element performs some computation, such as filtering of the data, converting the data to the frequency domain, or monitoring. Feedback within a PE is possible, like with FEC encoding and decoding algorithms. Communication is gated by handshaking, and queues can perform data rate conversions between blocks talking at different rates. Data path splitting and merging is allowed. Interface blocks (INT) bridge communication between the signal processing path and other circuits. These might be converters, such as analog-to-digital (ADC) or digital-to-analog (DAC), or they could be a memory.

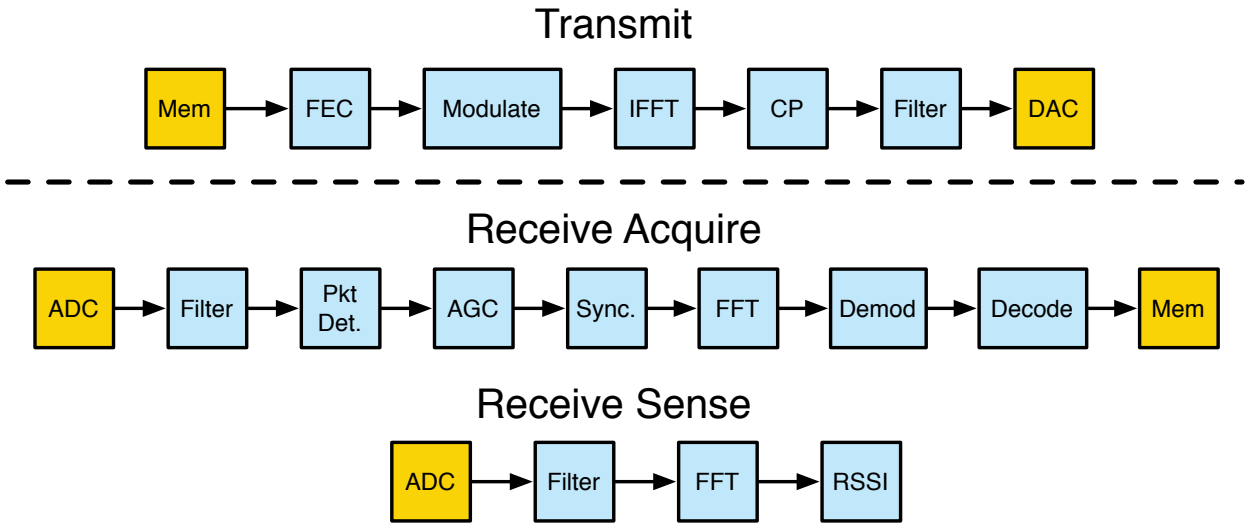
Applying the model to two of the previously mentioned applications produces the results also shown in Figure 2.10. Many of the details are abstracted, and will be discussed in later chapters. The point here is to imply that these algorithms have a commonality that can be generalized into one model. But not all signal processing fits this model. Neural networks process signals, like audio in 1D or pictures and videos in 2D. However, their memory and computational footprints are large enough to require a different style of processing. They fall into a called memory-based processors, which, as opposed to stream-based processors, operate on values read from and written back to a central memory [60].



(a) A common signal processing algorithm example model.



(b) A sideband separating spectrometer targeting atmospheric measurements in the style of the common model.



(c) The radio transmit processor on top, and multiple receivers on bottom. One monitors and detects the presence of signals, and the other actual decodes received data into messages.

Figure 2.10: Some examples of a common signal processing algorithm model, with a generic one on top and two specific applications below.

Chapter 3

Digital Signal Processing SoC Model

Designing ASICs that perform the applications mentioned in the last chapter can take substantial effort and time. But the common model presented in the previous chapter lends itself to hardware generators. Before diving into generators, translating a generic signal processing algorithm model into something amenable to hardware is necessary. The first section in this chapter explores commercial and published digital signal processors to illuminate key design choices and motivate the new model. The rest of the chapter presents an SoC model for a dedicated signal processor, motivated by balancing support for many applications and having a simple, usable model.

3.1 Existing Models

3.1.1 Centralized Digital Signal Processors

Texas Instruments produces a line of digital signal processors called TMS320. Looking specifically at the C5545 and C6671, it's clear they are trying to cater to a wide set of applications. The C5545 contains a fixed-point CPU with custom instructions and a deep pipeline to support DSP [61]. A computation stage performs multiply accumulate (MAC) operations, which are common in signal processing applications. An independent FFT hardware accelerator supports 8-1024 point real- and complex-valued FFTs, but only in powers of 2 [10]. On-chip ROM boots the device, and the rest of the chip consists of periphery communication, a 10-bit SAR ADC, USB 2.0, clocking, and power management. Four DMAs move data into, out of, and around the chip memory space. Figure 3.1 shows a block diagram of the C5545 chip architecture. The C6671 CPU contains floating point support, and multicore versions are available. Internal and external memory controllers move data within and outside the core. This SoC targets networking applications with its packet and security accelerators [62]. Finally, it contains numerous periphery communication devices, including ethernet, and support for external DDR3 memory. Programming the TMS320 processors is done through the TI library-based toolchain in C, C++, and/or assembly.

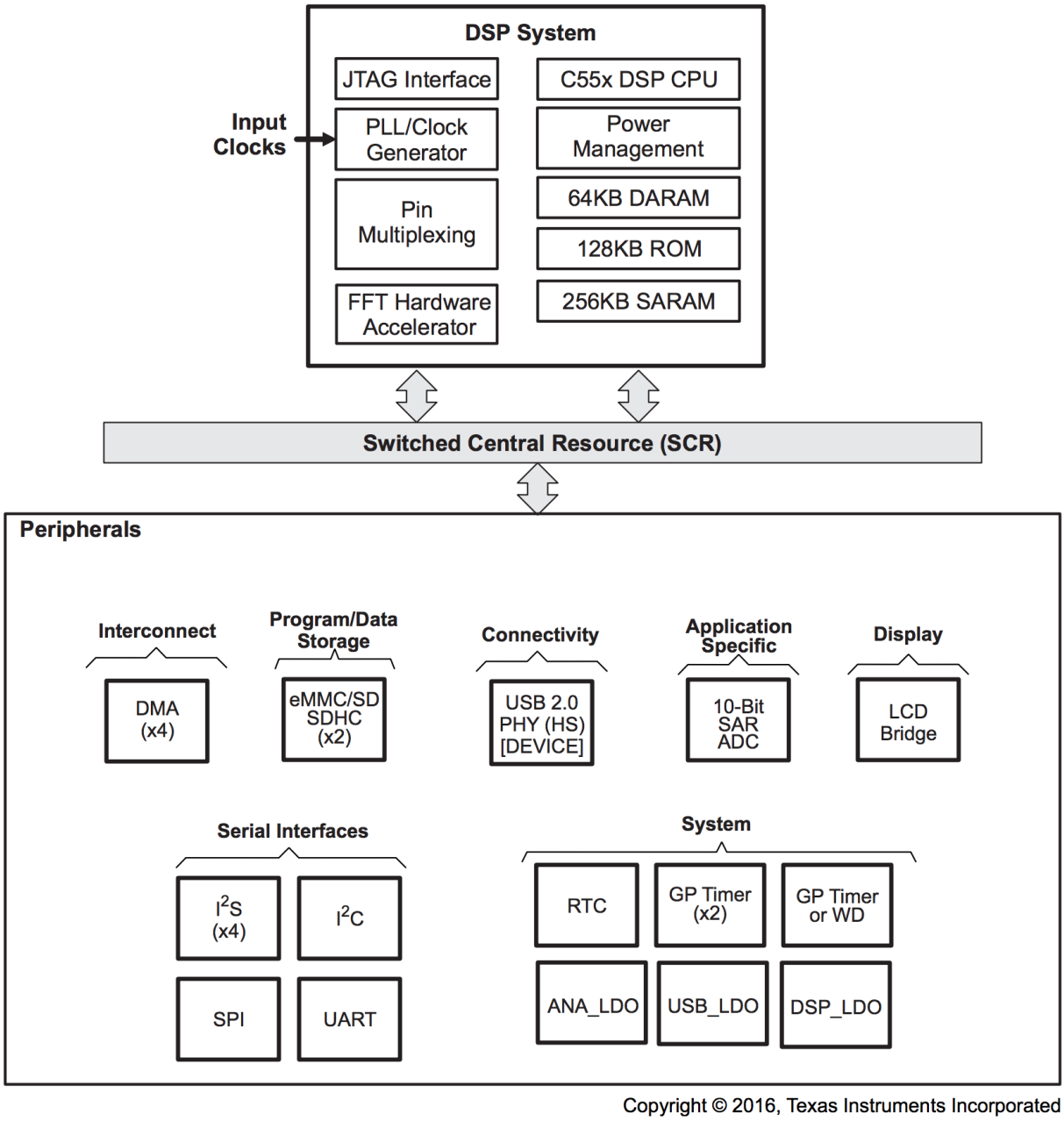


Figure 3.1: The Texas Instruments TMS320C5545 DSP SoC block diagram [10].

Ceva offers DSPs as well. Their XC5 core targeting software-defined radio (SDR) enhances a traditional CPU with common features like VLIW, vector processing units, a custom ISA, multiple DMAs, and a power scaling unit [63]. Communication is handled through AXI and APB interfaces, both AMBA specifications. There are no special function accelerators, which would be nice but are not required for SDR. Another processor, the Teaklite-4 SoC,

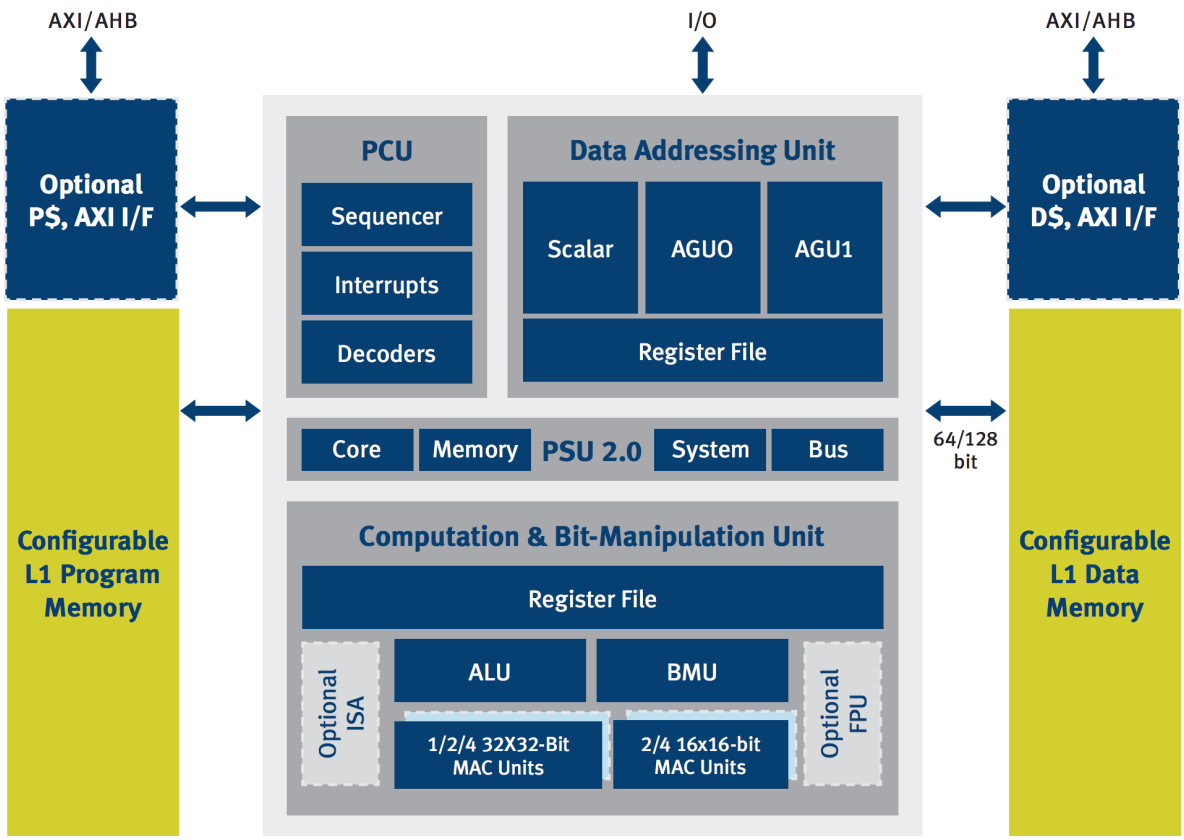


Figure 3.2: The Ceva Teaklite-4 architecture, showing the physically separate program and data memories (Harvard architecture) [11].

contains an always-on audio processor for applications like Google Home and Amazon Echo [11]. It uses a Harvard SIMD architecture. The Harvard architecture physically separates instruction and data storage and communication, as seen in Figure 3.2, allowing for a number of benefits. These benefits include simultaneous access of instructions and data, as is often done in microcontrollers, and separate word lengths between them. But again, there are no special function accelerators, just software IP targeting applications like voice processing and bluetooth support.

3.1.2 Distributed Digital Signal Processors

Manycore DSP processors typically feature small, custom RISC-like CPUs in a mesh network. With distributed processing, mapping and scheduling a program onto the SoC becomes difficult, as poor scheduling can lead to underutilized resources or contention and deadlock in the data flow. Despite the programming difficulty, attempts have been made. The RC64 architecture features 64 VLIW SIMD cores with a shared memory space and separate radio

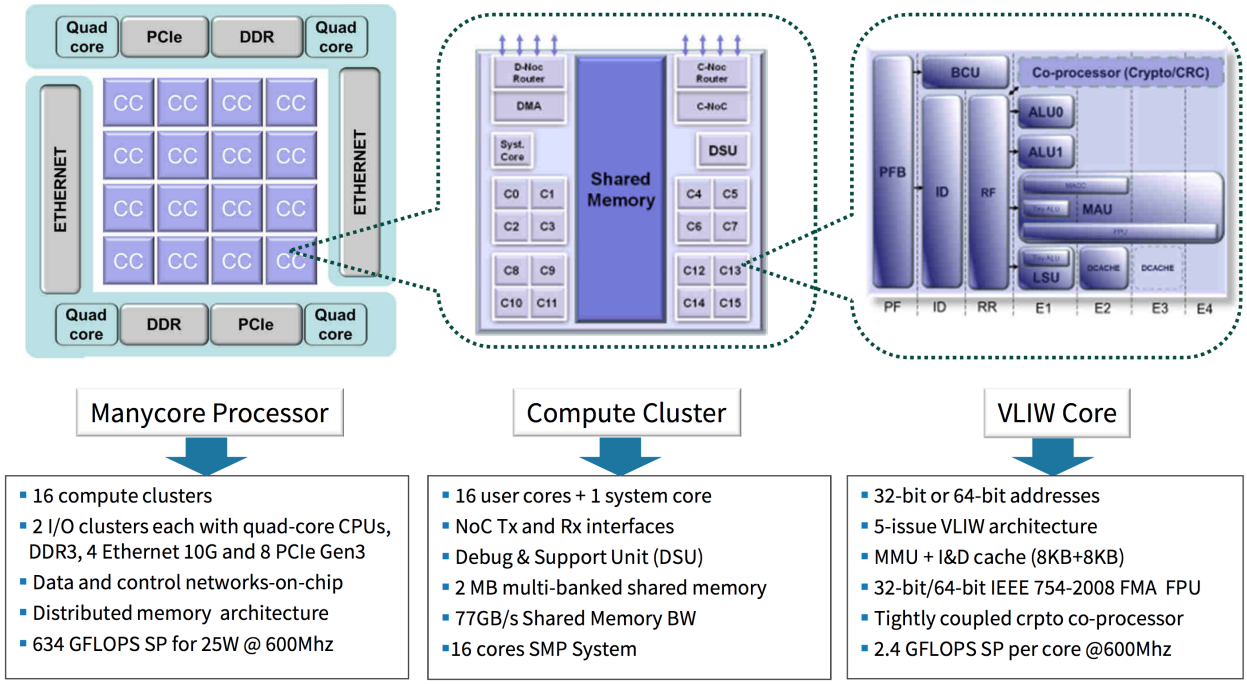


Figure 3.3: The Kalray MPPA-256 Bostan Processor Architecture, showing hierarchical manycore clustering [12].

baseband processing accelerators, such as FEC [64]. A separate scheduler and DMAs provide control and data movement, and high-speed links provide fast off-chip communication. The Kalray Massively Parallel Processing Array (MPPA) hierarchically groups VLIW cores into 16 clusters of 16 cores each [12]. Each core has a FPU and crypto co-processor, and each cluster shares a 2 MB memory. A system core provides control and scheduling to each cluster, and a DMA and routers help move instructions and data around the network-on-chip (NoC). Figure 3.3 shows the architectural diagram and description of this processor. A third manycore DSP processor comes from XMOS, which targets audio and video processing [65]. Its 16 cores have custom DSP instructions like MAC and cyclic redundancy check (CRC). Custom C-extensions support its programming model.

3.1.3 Hardened Digital Signal Processors

The previous sections demonstrated flexible architectures targeting multiple applications. However, peak performance and efficiency comes from hardware performing only the necessary calculations for the desired application. This requires restricting the set of supported applications to one. Image and video compression, encoding, decompression, and decoding are commonly implemented as hard-coded accelerators because of extreme low power or high throughput requirements. These may be either streaming or memory-based architectures. For example, Gu presents an image compression scheme and ASIC for wireless capsule

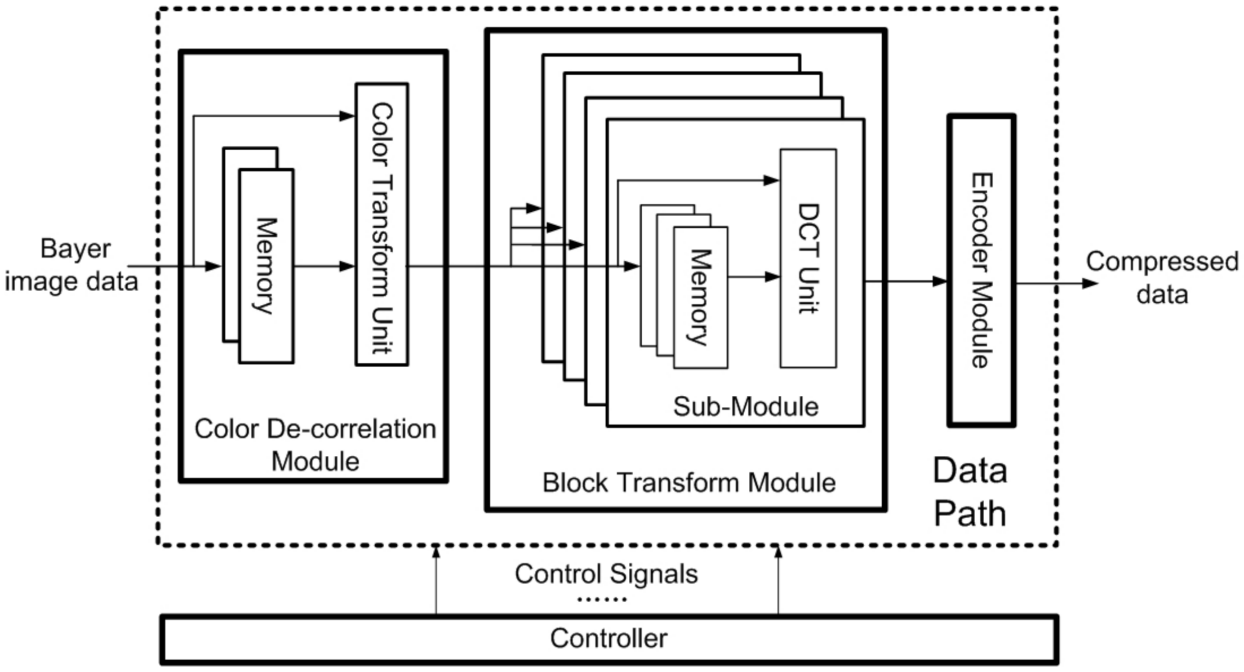


Figure 3.4: Image compression architectural datapath block diagram for a capsule endocscopy ASIC [13].

endoscopy, which requires low power due to limited battery size [13]. Figure 3.4 gives the streaming datapath block diagram of this chip, which in a 0.18 μm process consumes just over 1 mW. Conversely, Tikekar demonstrates a high-efficiency video coding (HEVC) decoder using a memory-based architecture with significant optimizations to meet the high throughput and data size of 4K Ultra HD 30-fps streaming video [66]. Almost half the chip is consumed by SRAMs used to buffer video frames and computation outputs. The logic is custom-designed for HEVC decoding, but also included are DMAs for data access and movement

Signal processors for space applications require both low power and high throughput, implying that ASICs are ideal. Recent work in producing high-bandwidth, low-power chips from NASA's Jet Propulsion Laboratory (JPL) and UCLA use dedicated, hardened digital signal processing elements [14, 67]. Accumulation improves the SNR and reduces the output data rate, allowing for simplified off-chip communication protocols like SPI. Figure 3.5 shows a block diagram of one spectrometer, receiving complex I and Q data before filtering, Fourier transforming, and accumulating.

3.1.4 Summary and Trends

Digital signal processing chip architectures tend to trade off flexibility for performance. Distributed and centralized processors without accelerators can support the widest range

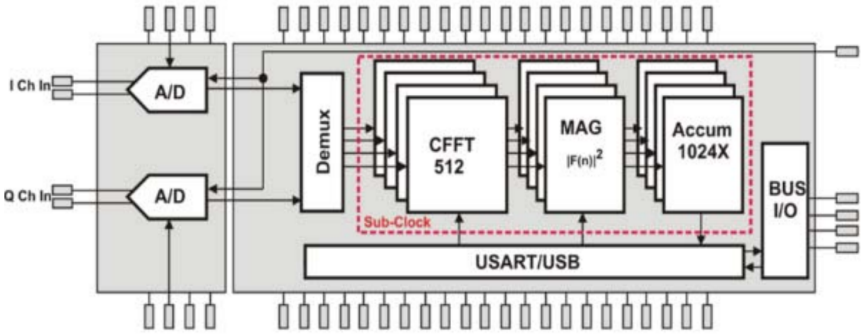


Figure 3.5: Spectrometer architectural block diagram, showing the digital processing path after the ADC inputs [14].

of DSP applications, but they excel at none. Accelerator-based architectures find a middle ground with decent performance for applications requiring the accelerators they contain and flexibility in application coverage. Hardened ASICs only do one application, but they do it best.

To support agile development of SoCs, an accelerator-based architecture is recommended. Pieces of the algorithm can be hardened over time, but anything left unoptimized or not mapped to RTL can be done in software on a CPU. Most DSPs contain FFT hardware, as frequency-domain analysis and processing is used in many applications. Also, DMAs are common for moving data around, as seen in all memory-based architectures. Finally, nearly all DSPs contain efficient, parallel MAC units. So the recommended accelerator-based architecture should contain at least a DMA, MAC accelerator, and FFT accelerator.

3.2 System Overview

This signal processing SoC model couples a general-purpose processor with a signal processing accelerator. The general-purpose processor provides a convenient programming interface, and for the sake of agility, it allows the designer to push certain tasks to software. Though the architecture for this processor is flexible, the model requires that it have at least one core and access to memory-mapped peripherals. The signal processing accelerator includes any number of DSP chains communicating with the general-purpose processor through two crossbars, as shown in Figure 3.6. Specifically, each DSP co-processor attaches to the processor through two crossbars as memory-mapped peripherals. The first connection is the control interface, and the second connection is the data interface. On each crossbar, the main processor is the master and the components of the DSP chain are the slaves. Asynchronous FIFOs separate the main processor from the DSP co-processor. A JTAG debugger provides direct access to the signal processing accelerator, supporting test and backup control modes of operation.

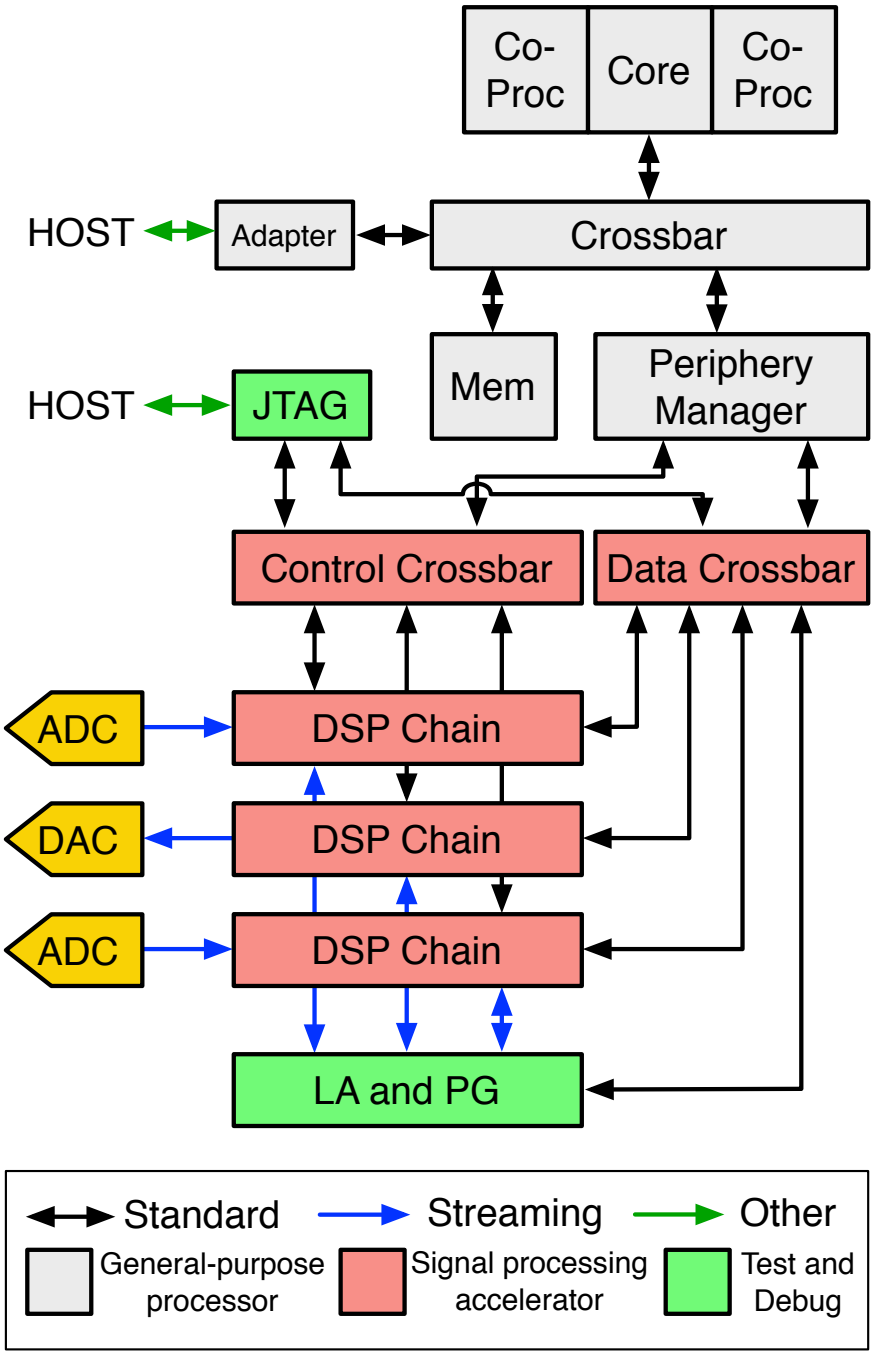


Figure 3.6: A digital signal SoC model. The general-purpose processor and local co-processors talk to the digital signal processing accelerator through a periphery manager and memory-mapped IO.

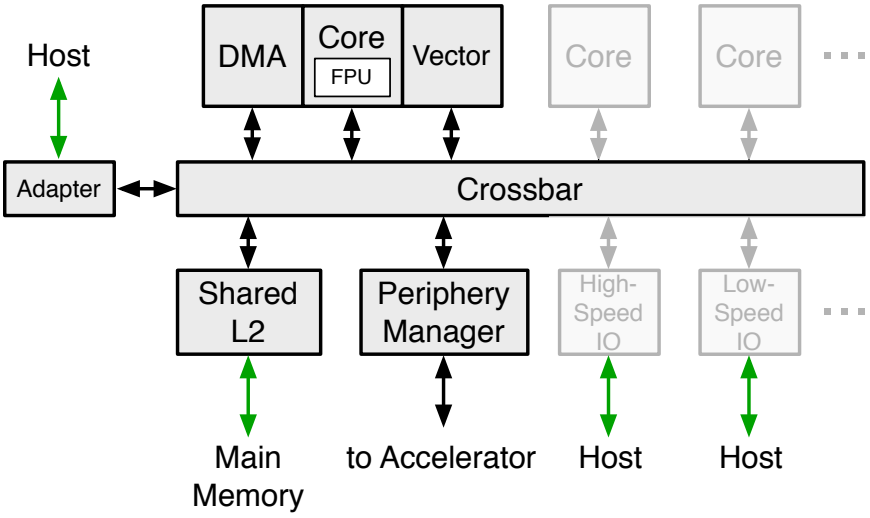


Figure 3.7: General-purpose processor architecture. Grayed out boxes show possible extensions.

3.3 General-Purpose Processor

The requirements for the general-purpose processor are flexible. In general, the processor includes one or more cores communicating through a crossbar to a shared, coherent memory. The architecture of the cores should be chosen for the desired application, such as lightweight for low-power spectrometers or complex and diverse for a high-performance radio baseband processor. A floating-point unit is recommended to support larger dynamic range computations. Each core may contain local co-processors designed to accelerate or off-load certain tasks from the cores. One suggested co-processor is a direct memory access (DMA) controller. Signal processing produces large sets of data, and moving that data efficiently with a DMA relieves the cores from burdening themselves with data movement. A second suggestion is a vector or SIMD co-processor to speed up signal processing not handled by the dedicated signal processing accelerator, such as any remaining MAC operations.

The cores and their co-processors communicate with the signal processing accelerator through memory transfers. Thus the memory model, whether cache-based or scratchpad-based, requires static physical addressing and memory-mapped peripherals. The cores move data from the accelerator to their local memory (cache or scratchpad) for further processing, or to move data off chip. Moving data requires a communication protocol, such as ARM Advanced Microcontroller Bus Architecture (AMBA) protocols. Communication off-chip is possible through a master connection to the general-purpose processor crossbar. An adapter serializes the data and reduces the data rate to off-chip controllers. Programming is done by the off-chip host, and slow delivery of data and results is possible through this host interface.

Many extensions to this model are possible though not explored in depth. Instead of slow communication with a host, high-speed serial links or memory interfaces may be connected to

the crossbar. Numerous peripherals are seen on processors at the beginning of this chapter, such as I2C, SPI, UART, USB, PCIe, and Ethernet. These are lumped together in the model as possible low-speed or high-speed IO ports. Untethered architectures would remove the need for a host FPGA or processor. Figure 3.7 shows more details of the general-purpose processor architecture, with unexplored model extensions grayed out.

3.4 Signal Processing Accelerator

The digital signal processor is a set of processing elements (PEs), or actors, which are combined to form a processing chain, as seen in Figure 3.8. Each PE contains a custom DSP function of arbitrary complexity, for example a power element that squares input complex data or an FFT element. One streaming input and one streaming output comprise the data interfaces for each PE, and one more interface supports control. A PE wrapper converts these interfaces into standardized instances, for example AXI4-Stream for data and TileLink for control. Control and monitoring is handled through a status and control register (SCR) file. This SCR file contains registers that are assigned memory addresses and accessible through the memory-mapped control interface. The motivations behind modeling a DSP processor this way are covered in Chapter 4.

A collection of PEs connected together comprise a DSP chain. For reasons discussed in the previous chapter, data flows in one direction only. On both ends of the DSP chain are special interface (INT) elements. Typical interface elements are ADCs and DACs for analog conversion and . At one end of the chain is typically an analog-to-digital converter (ADC) for receive chains or a digital-to-analog converter (DAC) for transmit chains. At the other end of the chain is often a buffer for data storage and access from the CPU. For receive chains, the buffer stores streaming data and provides a memory-mapped interface for the processor or a DMA controller to read. For transmit chains, the buffer stores transmit data written by the processor or DMA controller, then streams it into the DSP chain. Chains terminated on both ends with buffers or on both ends by analog interfaces are permitted, though atypical for applications explored in this thesis. Each chain contains an additional SCR file unaffiliated with any PE for controlling data flow, such as swapping in data from test structures.

Multiple chains are supported by growing the number of crossbar slave ports and increasing the address space allotted to the periphery. Though bandwidth between the processor and accelerator is limited by the memory-mapped IO manager, so more chains may oversaturate data movement rates. Separate crossbars for data and control were chosen to avoid exploding crossbar sizes. Clocking is left flexible in this model. Any standard or streaming interface may be cut with an asynchronous first-in first-out (FIFO) queue, which isolates the clock domains on either side. However, as numerous PLLs or clock receivers add pins and area, this option is only minimally explored in later chapters of this thesis. The intended use model for PEs and chains is to configure them through the control interface, then let them constantly stream data through as the processor moves data through interface elements. Pre-

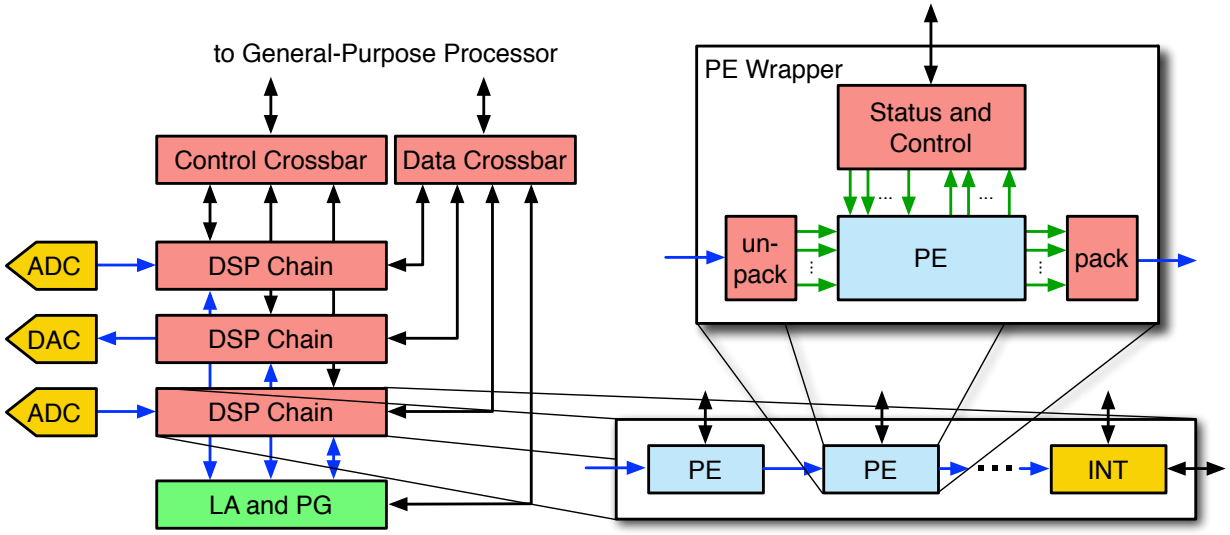


Figure 3.8: Signal processing accelerator architecture.

and post-processing of datasets is done on the CPU, and results are communicated to the host.

3.5 Debug and Test

Debugging, test, and control interfaces are also included in the model, as seen in Figure 3.6. Each DSP chain control and data port acts as a slave device, while the CPU and JTAG debugger act as master devices on the crossbars (so each crossbar is 2xN). Here, JTAG is an optional debugging input, meant to bypass the main processor for direct access to the DSP co-processor. It contains registers for each interface signal, which then pass through an asynchronous FIFO to enter the DSP accelerator clock domain. Also connected to the crossbars, but not contained within a DSP chain, are a logic analyzer (LA) and pattern generator (PG). These blocks incorporate design-for-test (DfT) memories and control that optionally connect to each PE for unit-level post-silicon testing. Additionally, each PE can be individually run on a dataset by writing the dataset to the PG, playing it through the PE, then reading the result from the LA.

Chapter 4

A DSP SoC Generator

By modeling a DSP processor in the way presented in Sections 3.2 through 3.5, we are able to design a suite of tools that support easy design, verification, test, and programming of *any* processor prescribing to the model. The design of this framework comes from the agile methodology's emphasis on tools and generators. Chosen framework features underscore the rest of the agile principles, such that small, collaborative design teams can focus on quickly constructing functional prototypes and easily respond to changing specifications. This DSP co-processor generator is written in Chisel, a domain-specific language (DSL) embedded in Scala for constructing hardware.

4.1 Generator Overview

The source code for the SoC generator is open-source and available online at `https://github.com/ucb-art/craft2-chip`. The generator was written in Chisel, Berkeley's hardware construction language. Chisel is a DSL written inside Scala, and the SoC generator utilizes the Scala Build Tool (SBT) to manage project dependencies and compile the design. The top-level git repository has submodules containing the required dependencies. The dsp-framework submodule is a general submodule useful for Chisel-based DSP projects, even those outside the purview of this SoC generator. Other submodules mostly contain processing elements, such that an SoC only needs to include code for relevant processing elements rather than having a single library containing all existing processing elements.

Before creating the SoC generator, we created a collection of useful tools usable by many different projects, called dsp-framework. Inside this repository is the rocket-chip repository as a submodule. This repo includes its compatible version of Chisel and FIRRTL, as well as the RISCV toolchain needed to build and test code for the processor. The rocket-dsp-utils repo contains Chisel designs which depend on the rocket-chip repo, which includes much of this SoC generator's glue code. A generalized pattern generator and logic analyzer exist in the builtin-debuggers repository. The dsptools repo includes DSP-specific low-level utilities for Chisel, such as support for floating-point and complex data types. The remaining repositories

provide useful infrastructure code for building and testing the SoC generator.

Listing 4.1: Directory structure of the generator

```
dsp-framework/ - collection of useful dependencies
  rocket-chip/ - RISCV processor generator with FIRRTL, Chisel, and Hwacha repos
  chisel-testers/, firrtl-interpreter/ - testing repos
  testchipip/ - design for test repo, also contains the main memory
  dsptools/ - useful DSP code, like DspComplex type
  rocket-dsp-utils/ - craft-specific rocket-chip-dependent resources
  builtin-debugger/ - logic analyzer and pattern generator
  barstools/ - convenient compilation functions
vsim/ - VCS simulation directory
ncsim/ - irun simulation directory
bootrom/ - sources for the first-stage bootloader
src/main/scala/ - scala source files
riscv-dma2/ - the memcopy DMA RoCC accelerator
tests/ - custom C code tests for peripherals
fft/, pfb/, etc. - code for DSP blocks used in the design
```

This SoC generator reuses the rocket-chip generator, a Chisel-based, parameterized RISCV processor generator. Reusing this existing code prevented the need for constructing a custom CPU generator. The bulk of the work done for this thesis was on the signal processing accelerator, a custom generator complete with design verification checking. That includes the contents of the `rocket-dsp-utils` directory, DSP block directories, and simulation and test directories.

4.2 General-Purpose Processor Generator

The RISCV ISA is an open-source, reduced instruction set architecture developed at Berkeley for the purposes of creating a flexible, simple, usable, free standardized ISA [68, 69]. The rocket-chip repository is also a free, parameterized, open-source generator producing designs that implement the RISCV ISA [15, 70]. The chosen parameterization supports a single tile (core), communicating through a crossbar to memory, periphery devices, and the host. An ISA extension called the Rocket Custom Coprocessor interface supports tightly-integrated accelerators. Two RoCC accelerators previously designed were selected for this generator. These are the direct memory access (DMA) and Vector (Hwacha) accelerators. The DMA accelerator offloads structured memory movement from the CPU, and the Vector accelerator performs SIMD-like computation, useful for many signal processing kernels.

The rocket CPU includes L1 instruction and data caches of programmable size. The vector accelerator also includes an integrated cache for quick data access. Both communicate through a crossbar to a shared, coherent L2, which can be backed on-chip by a large main memory or off-chip by a host processor. In addition to an L2, the memory model supports

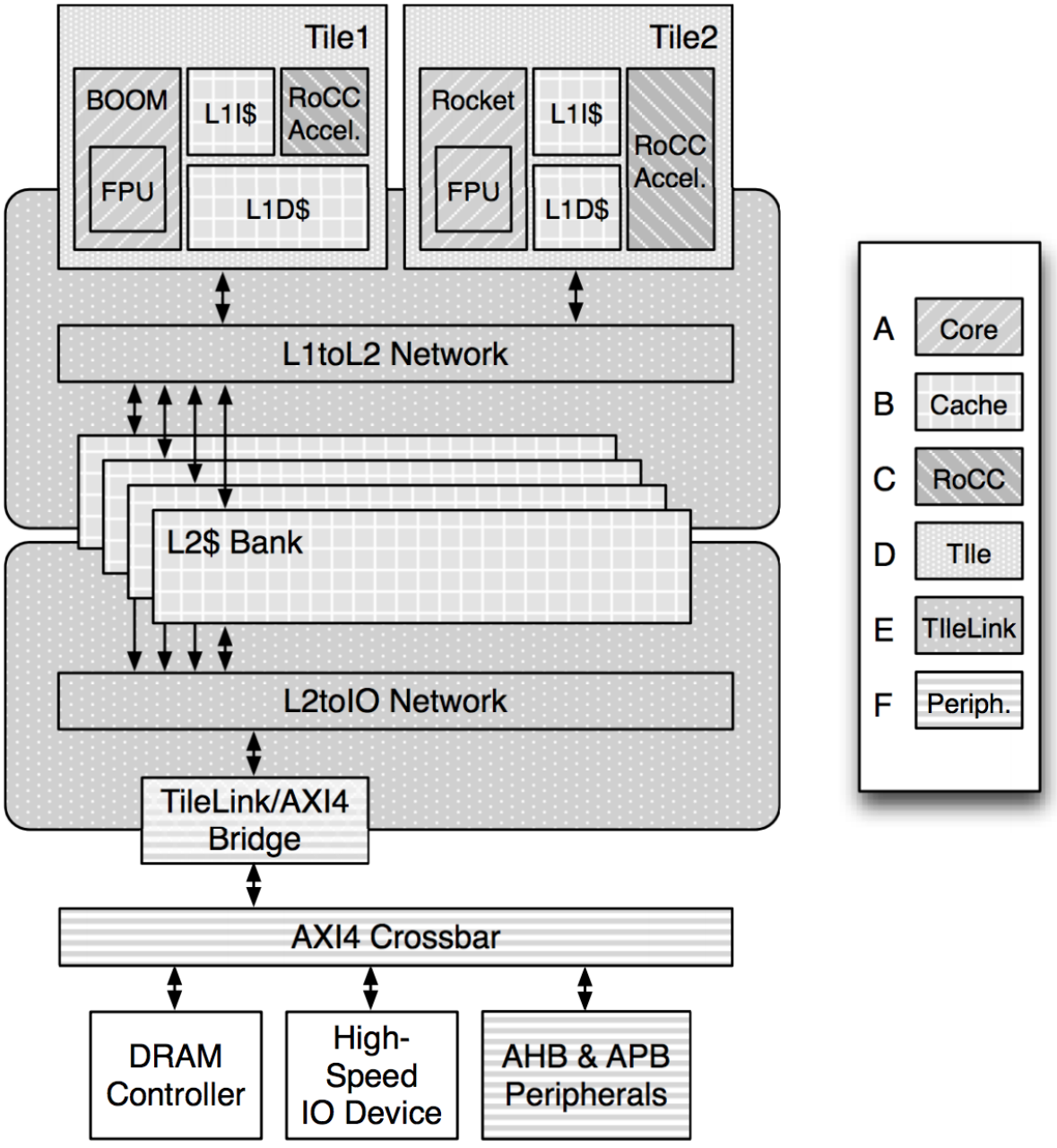


Figure 4.1: The rocket-chip generator can produce various topologies using hierarchical generators [15].

memory-mapped peripheral devices, connected as a slave to the crossbar through a periphery manager. The crossbar and device interfaces use a TileLink protocol. TileLink is an extensible SoC communication standard with both cached and uncached channels. The rocket-chip repository contains TileLink crossbars, routers, and arbiters, as well as converters for AXI, AHB, and APB AMBA protocols. Figure 4.1 shows a block diagram of rocket-chip generator components.

The version of rocket-chip used requires an external tether to load programs into the instruction memory and read outputs. A serial interface adapter translates low-bandwidth serial data to on-chip TileLink transactions. For simulation, a C shim dumps a program into the processor through the serial adapter, then the bootrom jumps the processor to the starting address. For testing, a tethered FPGA design communicates with the chip through a front-end server, allowing a user to run the same programs as used for simulation.

4.3 Signal Processing Accelerator Generator

The presented generator was constructed to simplify design and interconnectivity of custom DSP co-processors. To support existing verification methodologies, the interfaces in the signal processing accelerator were chosen to be variations on AXI4, such as regular AXI4 and AXI4-Stream. Thus the interface between the two processors starts with a TileLink to AXI4 converter, then an AXI4 asynchronous FIFO converts between the core and DSP clock domains. This allows for independent clocking between the processors. Two AXI4 crossbars arbitrate data and control plane communication. While the GPP is the main master for each crossbar, a JTAG to AXI4 interface acts as a backup master for direct access to the signal processing accelerator during testing. Multiple processing chains are all connected to the same two crossbars.

The control crossbar connects to each processing element (PE), and the number of output ports in the crossbar automatically scales with the number of processing elements added. For each chain, an independent SCR file provides status and control registers for the ADC, input valid signal, and LA/PG mux control signals. Chains are constructed from an ordered sequence of processing elements. For each processing element, a connection to the logic analyzer, a connection to the pattern generator, and a SAM are independent parameter choices. Parameters for the SAMs, LA, and PG allow the user to specify how deep the memories are. Addresses for all AXI4 connections are automatically generated. Figure 4.2 shows these higher level features, and Figure 4.3 shows how these features are specified in Chisel.

Processing elements have programmable input and output data types. These ports are packed and unpacked using a simple function into AXI4 Stream interfaces. When two processing elements are connected, their stream interface bitwidths are checked to ensure they match. Each processing element comes with an SCR file, and adding registers to it can be done with the `addStatus` and `addControl` functions. Status registers are writable by the PE, while control registers are read-only by the PE. Both are 64 bits. Processing elements

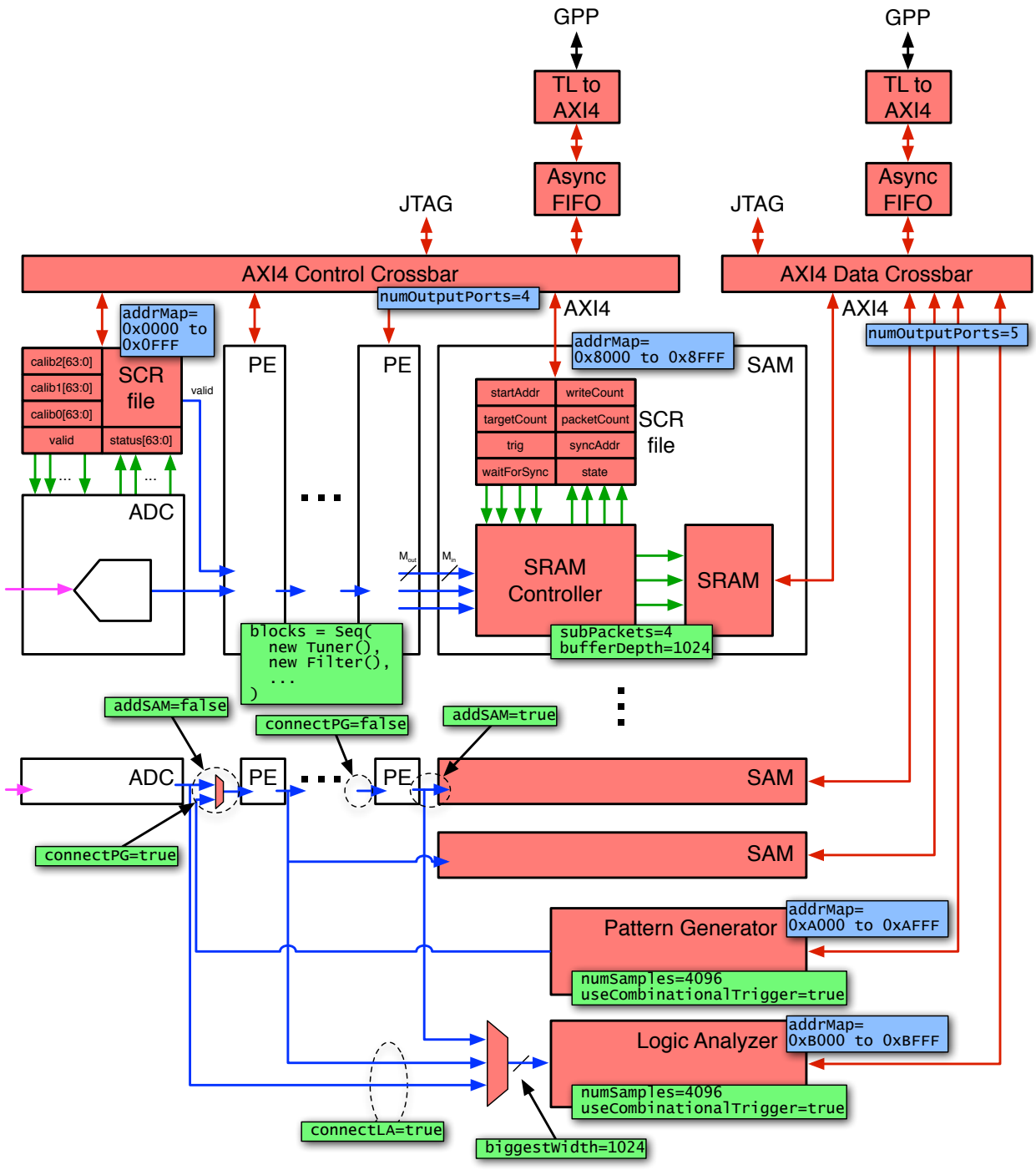


Figure 4.2: Signal processing accelerator generator features, with user input parameters in green, generated parameters and automated checks in blue, and auto-generated designs in red.

```
170   def radar(id: String = "craft-radar"): Config = {
171     new Config(
172       (pname, site, here) => pname match {
173         case DefaultSAMKey => SAMConfig(1, 4096)
174         case DspChainId => id
175         case DspChainKey(_id) if _id == id => DspChainParameters(
176           blocks = Seq(
177             // (parameter to block function, unique id, BlockConnectionParameters, SAM Config (optional))
178             (implicit p => new BitManipulationBlock[T], id + ":bm1", bm1Connect(), bm1SAMConfig()),
179             (implicit p => new BitManipulationBlock[T], id + ":bm2", bm2Connect(), bm2SAMConfig()),
180             (implicit p => new TunerBlock[T, T], id + ":tuner", tunerConnect(), tunerSAMConfig()),
181             (implicit p => new FIRBlock[DspComplex[T]], id + ":fir", firConnect(), firSAMConfig()),
182             (implicit p => new PFBBlock[DspComplex[T]], id + ":pfb", pfbConnect(), pfbSAMConfig()),
183             (implicit p => new FFTBlock[T], id + ":fft", fftConnect(), fftSAMConfig())
184           ),
185           logicAnalyzerSamples = 8192,
186           logicAnalyzerUseCombinationalTrigger = true,
187           patternGeneratorSamples = 8192,
188           patternGeneratorUseCombinationalTrigger = true,
189           biggestWidth = 512
190         )
191         case _ => throw new CDEMatchError
192       }
193     ) ++
194     ConfigBuilder.writeHeaders("./tests") ++
195     ConfigBuilder.nastiTLParams(id) ++
196     BitManipulationConfigBuilder(id + ":bm1", bm1Config(), bm1Input, bm1Output) ++
197     BitManipulationConfigBuilder(id + ":bm2", bm2Config(), bm2Input, bm2Output) ++
198     TunerConfigBuilder(id + ":tuner", tunerConfig(), tunerInput, tunerOutput, Some(() => tunerMixer)) ++
199     FIRConfigBuilder(id + ":fir", firConfig(), firInput, Some(() => firOutput), Some(() => firTaps)) ++
200     PFBConfigBuilder(id + ":pfb", pfbConfig(), pfbInput, pfbConvert, Some(() => pfbOutput), Some(pfbTap)) ++
201     FFTConfigBuilder(id + ":fft", fftConfig(), fftInput, Some(() => fftOutput))
202   }
```

Figure 4.3: Creating a signal processing chain amounts to specifying a sequence of processing elements (blocks), configurations for each processing element using ConfigBuilders, and a handful of other parameters.

are typically parameterized themselves. Specific examples of processing elements and their parameterizations are given in later chapters.

Signal processing chains may be terminated by an ADC or DAC on one end. Parameterized analog IP generators are possible using the Berkeley BAG framework [55].

4.4 Verifying the Generators

The generator framework includes verification and continuous integration checks for both the general-purpose processor and signal processing accelerator. Figure 4.4 shows a high-level map of testing capabilities and checks performed in the generator framework. A RISCV toolchain and cross compiler provide code for compiling, testing, and interfacing with rocket processors. The compiler converts C code into RISCV-compatible assembly binaries. A library of ISA tests exercise individual instructions for full processor pipeline coverage. Interactions between instructions are tested through various benchmarks and pre-written programs, such as matrix multiply. Benchmarking programs, like dhrystone, characterize the performance of the chosen rocket parameterization. RoCC co-processors provide their own test suites.

Verification of the signal processing accelerator was designed to be independent of the processor choice. Individual processing elements are verified both through custom Chisel tests and external UVM tests. The Scala Breeze library provides MATLAB-like signal processing functions, which may be used to characterize the validity and tolerance of signal processing hardware functions. Plot.ly provides free, web-based plotting integrated directly into Scala. The combination of these features allows quick verification of signal processing element generators in Chisel.

After selecting parameters and generating Verilog, an IP-XACT file facilitates integration of processing elements with UVM testbenches. IP-XACT is an XML file format providing metadata about HDL IP files and modules, including interface protocols, address mapping, and parameterization [54]. Tools such as Cadence Verification Workbench can automatically generate UVM testbenches to verify proper functionality of interface protocols. Custom python testbench generators create stimuli and expected results for individual processing elements to run in a UVM verification framework. Travis continuous integration ensures Chisel tests pass as changes to processing element and framework repositories are committed.

Testing the chip after tape-out is done the same way as verifying the chip before tape-out. The general-purpose processor runs the same suite of ISA tests and benchmarks. The signal processing accelerator plays the same processing element test vectors through the pattern generator and checks the results, which get stored in the logic analyzer. This methodology allows for quick bring-up and testing of chip instances.

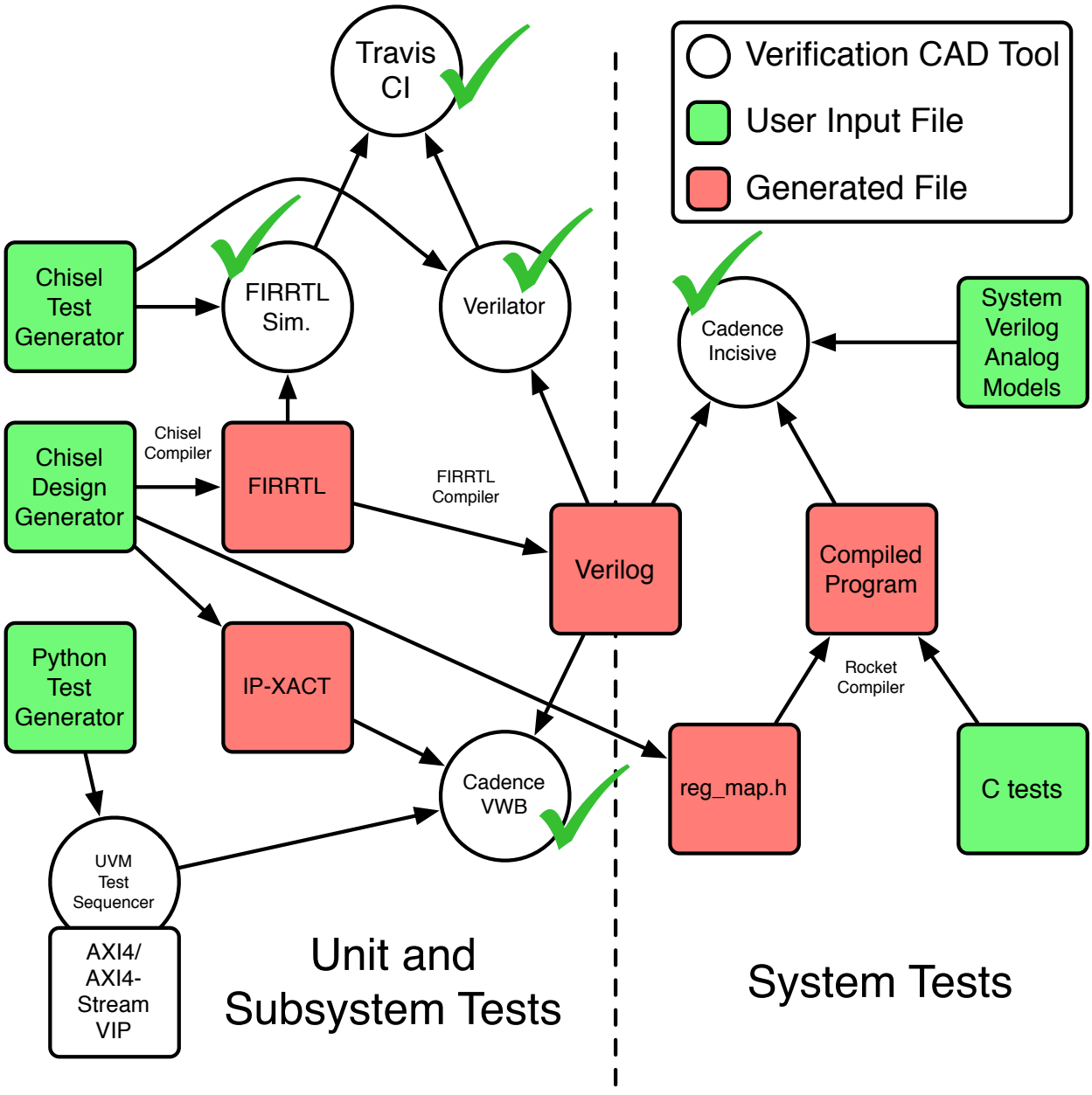


Figure 4.4: Verification framework.

Chapter 5

A Digital Spectrometer Design

5.1 Introduction

Many satellites, including Aqua, Aura, and OCO-2, use digital spectrometer back-ends to measure Earth's atmospheric composition. Other spacecraft bound for the moon and beyond also require spectrum analysis. These digital baseband spectrometers are typically implemented using FPGAs, which provide simplicity, low cost, and known radiation tolerance but add unnecessary weight and power consumption [71]. Application-specific integrated circuit (ASIC) implementations bring down the weight and power, but their high cost, complexity, and long development and production time make it difficult to justify their use in low volume applications. Previous work has attempted to simplify ASIC design by mapping designs in FPGA environments to ASIC implementations (see [72, 38]). Other work created single ASIC implementations at high cost and design effort [14].

Table 5.1 compares a few parameters of published spectrometers. Some perform sideband separation while others do not. Since signal-to-noise ratios of input data are low, designs focus on wide bandwidth FFTs and long accumulation depths instead of high ADC resolution. Thus ADC resolution and sampling rates are not necessarily increasing over time. This variety of design choices shows the need for parameterized generators.

This chapter applies parts of the previous SoC design methodology to a spectrometer targeting space-borne systems, leveraging open source code reuse through generator-based

Table 5.1: Properties of recently published digital spectrometer backends suggest a single parameterized generator covering the gamut of possible design choices would avoid repeated design efforts.

Source	Year	Platform	Inputs	ADC Sampling Rate	ADC Resolution	FFT Channels
[38]	2009	ASIC	1	1560 Msps	8 bits	4096
[71]	2010	FPGA	2	205 Msps	12 bits	1024
[7]	2013	FPGA	2	1000 Msps	8 bits	2048
[14]	2015	ASIC	2	2200 Msps	7 bits	512

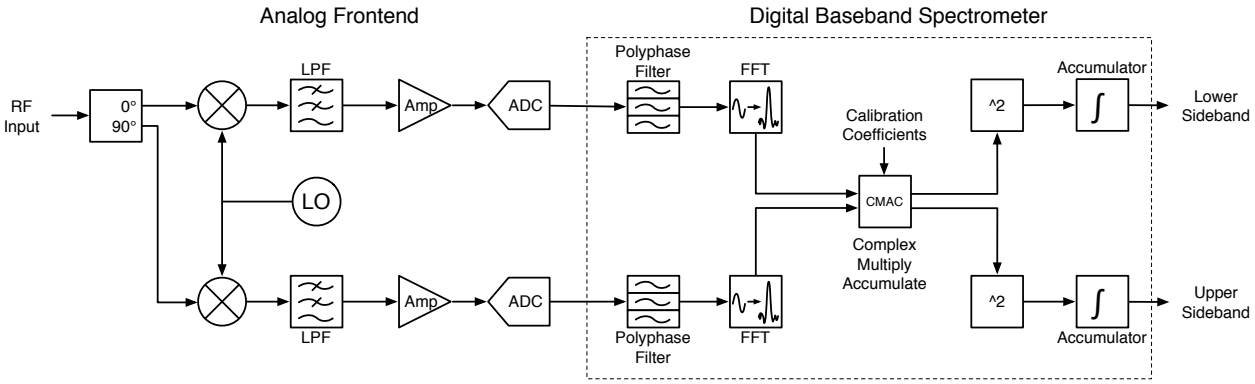


Figure 5.1: Generic spectrometer receiver. The analog front-end performs RF to baseband conversion, optional sideband (IQ) separation, filtering, and amplification. The digital baseband backend performs additionally filtering, transformation to the frequency domain, calibration, magnitude calculation, and accumulation.

testing and design, written in Chisel [51]. The methodology supports an array of spectrometer designs and is applicable to a variety of problems. Parameterized hardware generators and testers let users customize their implementation, quickly explore design spaces, and cut both cost and time when designing an ASIC. To demonstrate the methodology, an instance of a spectrometer is designed and evaluated in a 28nm FDSOI process. The signal processing element generators developed here were a precursor to the SoC generator framework presented in previous chapters. Some architectural and verification work is consistent between this chip and later chips, and lessons learned from the process of creating this chip were applied to the framework of later chips. Also, agile principles were followed during the process of designing and taping out this chip.

5.2 Spectrometer Generator

A generic spectrometer receiver system is shown in Figure 5.1. The digital portion occurs after the ADC and can be performed on an FPGA or ASIC, while the analog front-end is typically done with discrete components. This generator implements the entire digital baseband signal processing portion. The digital system design comes from CASPER, Collaboration for Astronomy Signal Processing and Electronics Research, as published in Asilomar by Parsons et al. [73]. The full system design and recent advances are summarized by Hickish et al. [**?**].

The spectrometer generator is designed in Chisel (version 2). Chisel was chosen because it supports flexible, parameterizable generator development, and the agile principles mentioned earlier suggest that designers improve tools and generators over instances. Also, its open source and open licensing choices enable these generators to be made publicly available for design reuse. Version 2 of Chisel includes an internal C++-based tester for fast, cycle-accurate verification. Chisel also produces low-level Verilog for FPGA and ASIC synthesis

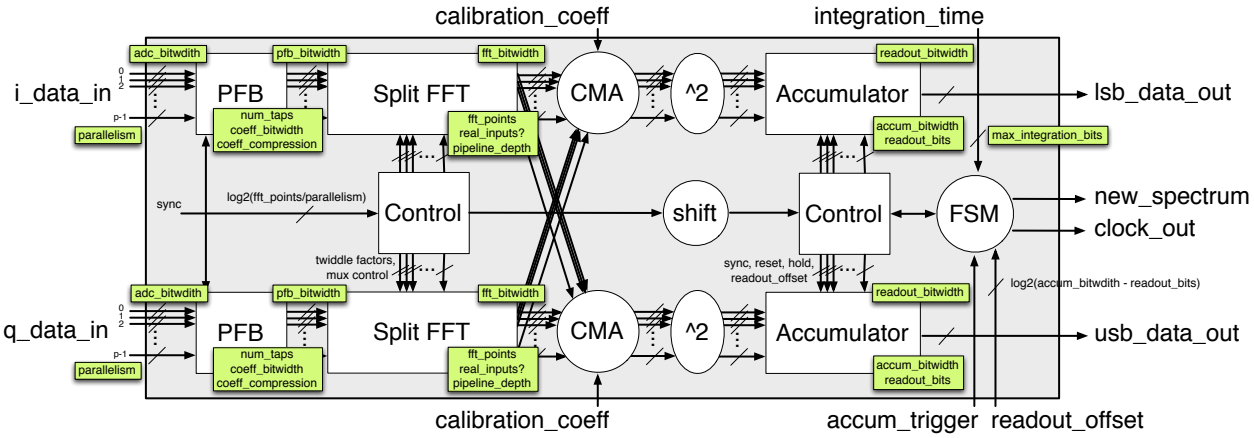


Figure 5.2: Spectrometer generator. Parameters are shown in green. Separate I and Q inputs arrive from two ADCs.

tools.

The hierarchical design uses unpacked versions of the AXI4-Stream interface presented earlier for easy, flexible datapath construction. Each major block comprises a datapath unit and a control unit. Separating the datapath from the control allows for datapaths to share control logic, saving area in multi-datapath systems. A global synchronization signal is pipelined through the control blocks, acting like the TLAST signal in AXI4-Stream. Optimal pipeline register locations are automatically calculated for the specified pipeline depth. Figure 5.2 presents a block diagram of the entire digital system generator. The following subsections describe individual block generator details.

5.2.1 Polyphase Filter

A polyphase finite impulse response (FIR) filter bank (PFB) shapes the signal spectrum by flattening the response within a spectral bin and suppressing overlap between bins [74]. Hard-coded coefficients come from a product of a Hamming window and a sinc function. The PFB generator has an option to compress the coefficients, as seen in Figure 5.3. For a large number of channels and a small coefficient bitwidth, the coefficient storage can be dramatically reduced by storing just the initial coefficients for each filter in the bank, and also storing when these coefficients need to change (delta compression). The change is always either plus one, zero, or minus one. For a 4-tap PFB with 8-bit coefficients and a 8192-point FFT, a coefficient ROM is reduced from 32 kB to 8 kB using delta compression. If the ROM is mapped to logic, synthesis tools may further optimize the coefficient calculation. A transposed polyphase FIR reduces the critical path to one multiplier and one adder at a cost of larger delay memories. The inputs, coefficients, and outputs are all signed and real-valued. The generator is parameterized in the properties shown in Table 5.2. Symmetric coefficient compression assumes the window function is symmetric, so it only stores half the values in

a ROM. Delta compression reduces the ROM further using the scheme shown in Figure 5.3.

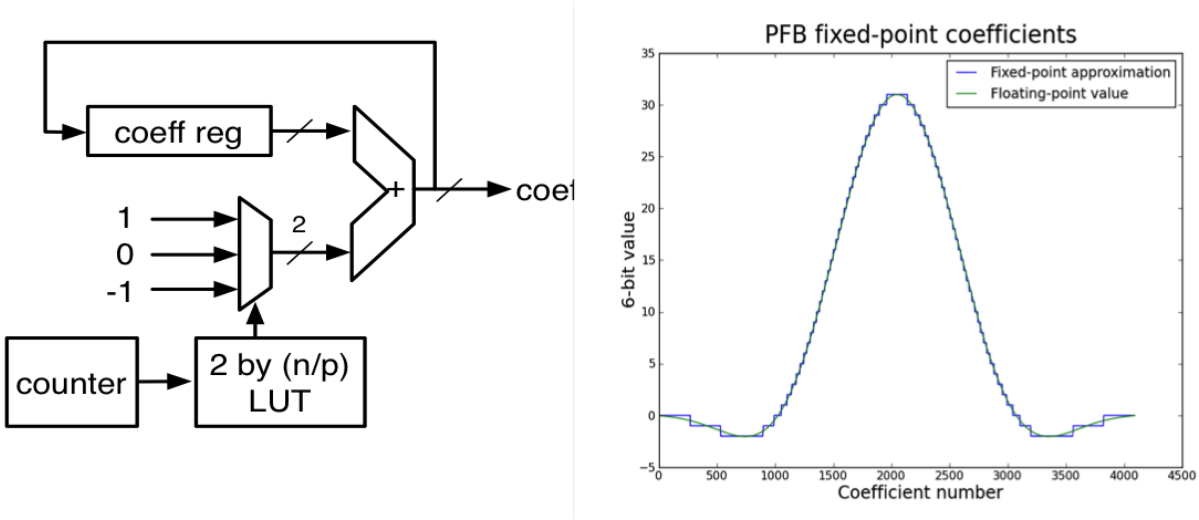


Figure 5.3: PFB coefficient calculation hardware and waveform. Both adjust for different parameter values, such as the bitwidths, number of taps, and number of channels.

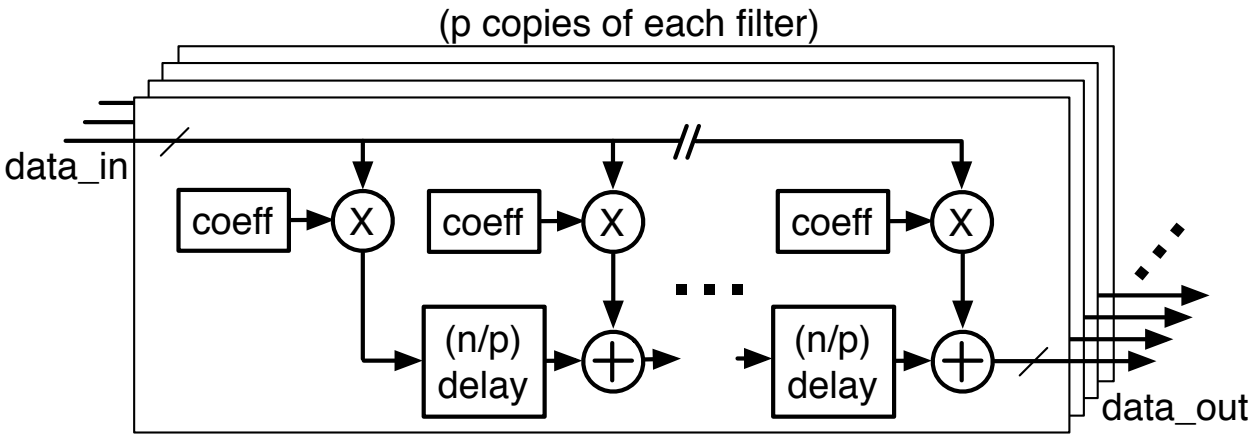


Figure 5.4: Parameterized polyphase filter. Coefficient values are addressed by the global synchronization signal. Delays are mapped to SRAMs when long. These delays dual as pipeline registers, and one more pipeline register is added to the output.

Table 5.2: PFB Parameterization

Parameter	Supported Range
number of taps	≥ 1
input bitwidth	≥ 1
coefficient bitwidth	≥ 1
coefficient compression	"none", "symmetric", "delta"
window type	"sinc Hamming"
window size (channels, n)	2^N where $N \geq 2$
parallelization (p)	any integer divisor of (channels $\times$ taps)
pipeline depth (output)	≥ 0
output bitwidth	≥ 1

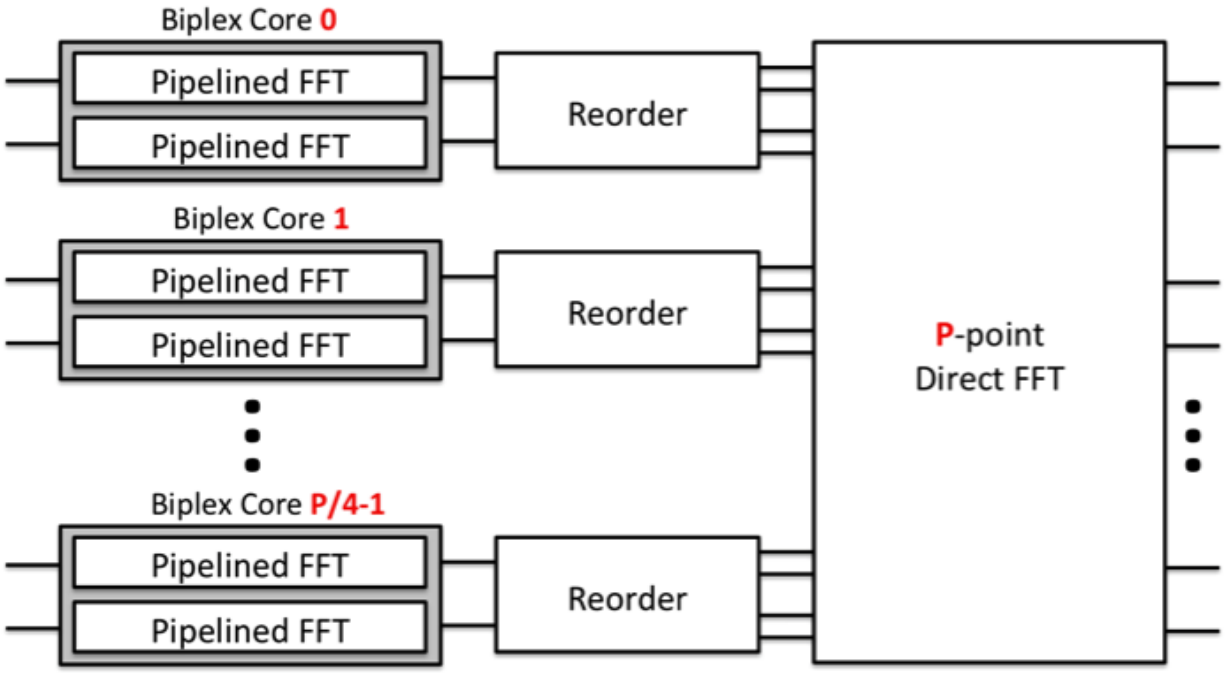
5.2.2 Fast Fourier Transform

To reduce hardware overhead, the FFT is decomposed into smaller FFTs. A number of pipelined FFTs are placed in parallel before a small direct FFT [75]. Such an architecture has high latency and memory requirements, but permits a large number of channels with fewer butterflies. The FFT generator supports any power-of-two FFT size and number of parallel inputs, with twiddle factors and pipeline register locations automatically calculated. Inputs may be real or complex. More channels improve spectral resolution, and 16384 channels were chosen to meet timing with the 10 GHz bandwidth and appropriate parallelization while still improving upon the previous ASIC spectrometer (8192 channels).

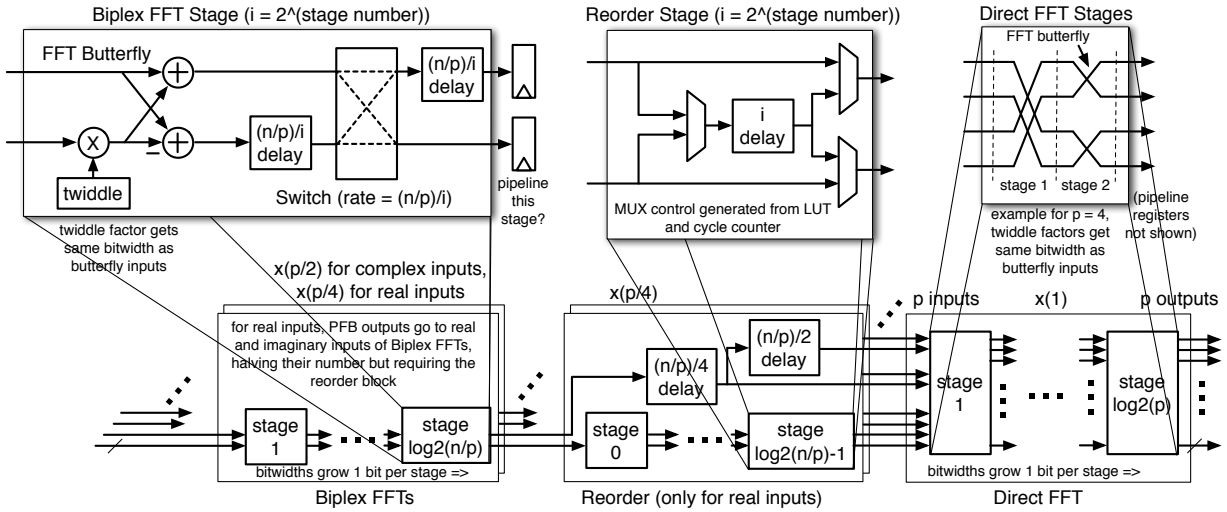
Biplex pipelined FFTs have 100% memory utilization, much better than the 50% of a traditional pipelined FFT [76]. This reduces the number of front-end FFTs required by 2. When using purely real data, the number of biplex FFTs can be halved again by feeding these data into both the real and imaginary inputs of the biplex FFTs. The outputs must then be reordered before or after the direct FFT, and one more complex multiply-add element is required. So an FFT with 8-way parallelism only needs $8/4 = 2$ biplex FFTs. Figure 5.5a shows an example 8-way parallelized version of the split-radix FFT using real-valued inputs and reordering before the direct FFT. Figure 5.5b gives a more detailed block diagram of the FFT generator.

A custom reorder block was designed to reorder the data before entering the direct FFT. Its design borrows heavily from the biplex FFT design. Delay elements growing by a factor of 2 each stage are selected or bypassed by muxes. The mux select signals are precomputed by simulation and stored in a ROM. Figure 5.5b shows block diagrams of the entire reorder block and the individual delay blocks with muxes.

Other FFT features include bitwidth growth, an option to use single-ported memories, and an option to use a complex input FFT. The PFB currently has 10-bit outputs, but the FFT has 16-bit outputs. To reduce the design size, the biplex and direct FFT sub-blocks feature bitwidth growth. Each butterfly stage grows the data by 1 bit, which allows for smaller



(a) High-level block diagram of the parameterized FFT generator. The reorder blocks shuffle data ordering when inputs are purely real.



(b) Detailed diagram of the parameterized streaming Fourier transform generator components.

Figure 5.5: The FFT generator.

memory and complex multiply-add elements. Truncation occurs after every stage once the maximum bitwidth is reached. The delay blocks are implemented as either dual-port SRAM memories, single-port SRAM memories, or basic shift registers. A separate FFT IP block removes the reordering, doubles the number of biplex FFTs, and accepts complex-valued inputs. Table 5.3 shows parameter options for the FFT generator.

Table 5.3: FFT Parameterization

Parameter	**Supported Range**
number of channels (n)	2^N where $N \geq 2$
input bitwidth	≥ 1
parallelization (p)	any integer divisor of n
pipeline depth	≥ 0
output bitwidth	≥ 1
input type	"real", "complex"

5.2.3 Sideband Separation, Power, and Accumulation

To remove front-end non-idealities and mismatch *across* the ADCs, frequency-domain calibration is included in the generator. The calibration coefficients must be calculated externally and stored in an on-chip RAM. A complex multiply-accumulate (CMA) performs the calibration operation. Finger [77] describes how to determine the coefficients. After calibration, the power of each frequency bin is calculated and accumulated. Results are scrambled, but not simply bit reversed, so unscrambling is not performed on-chip to reduce hardware complexity. A final bank of SRAMs buffers the accumulated data for readout while a new set of data is being accumulated. Parameterized control bits decide how many samples to accumulate, how many output bits to read, and the speed of the outputs readout. The outputs are the upper and lower sideband signals (USB and LSB). Finally, a snapshot mode feeds input data from the ADC directly to the accumulators for testing purposes.

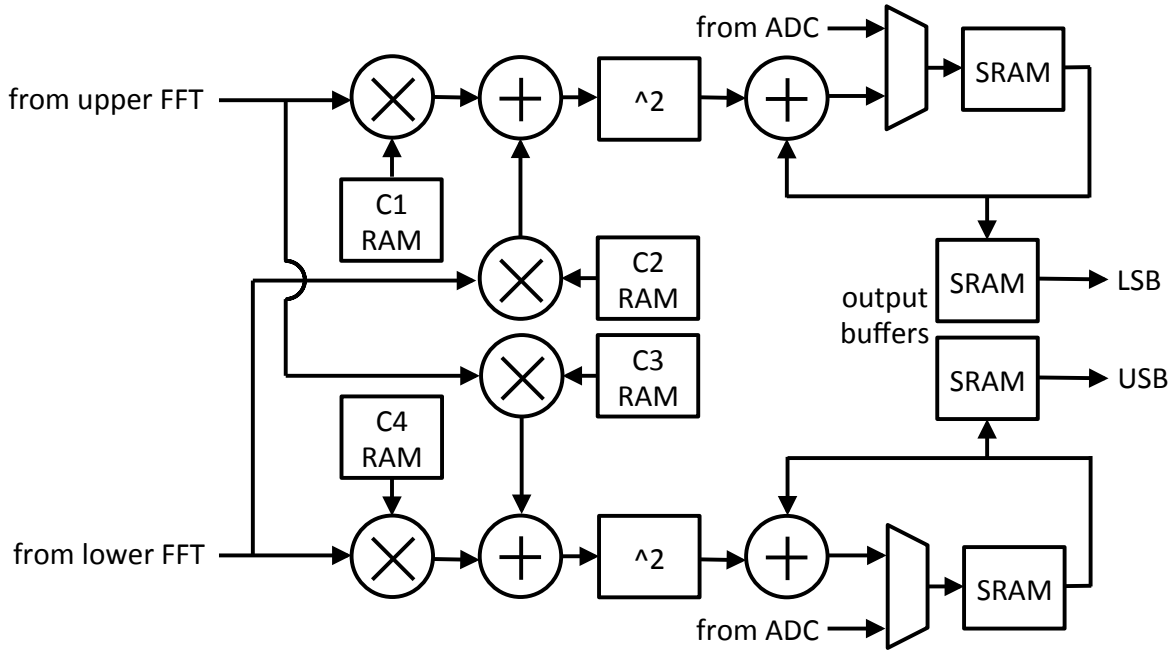


Figure 5.6: Calibration, power, and accumulation signal path block diagram. Calibration fixes mismatch between front-end receivers and ADCs.

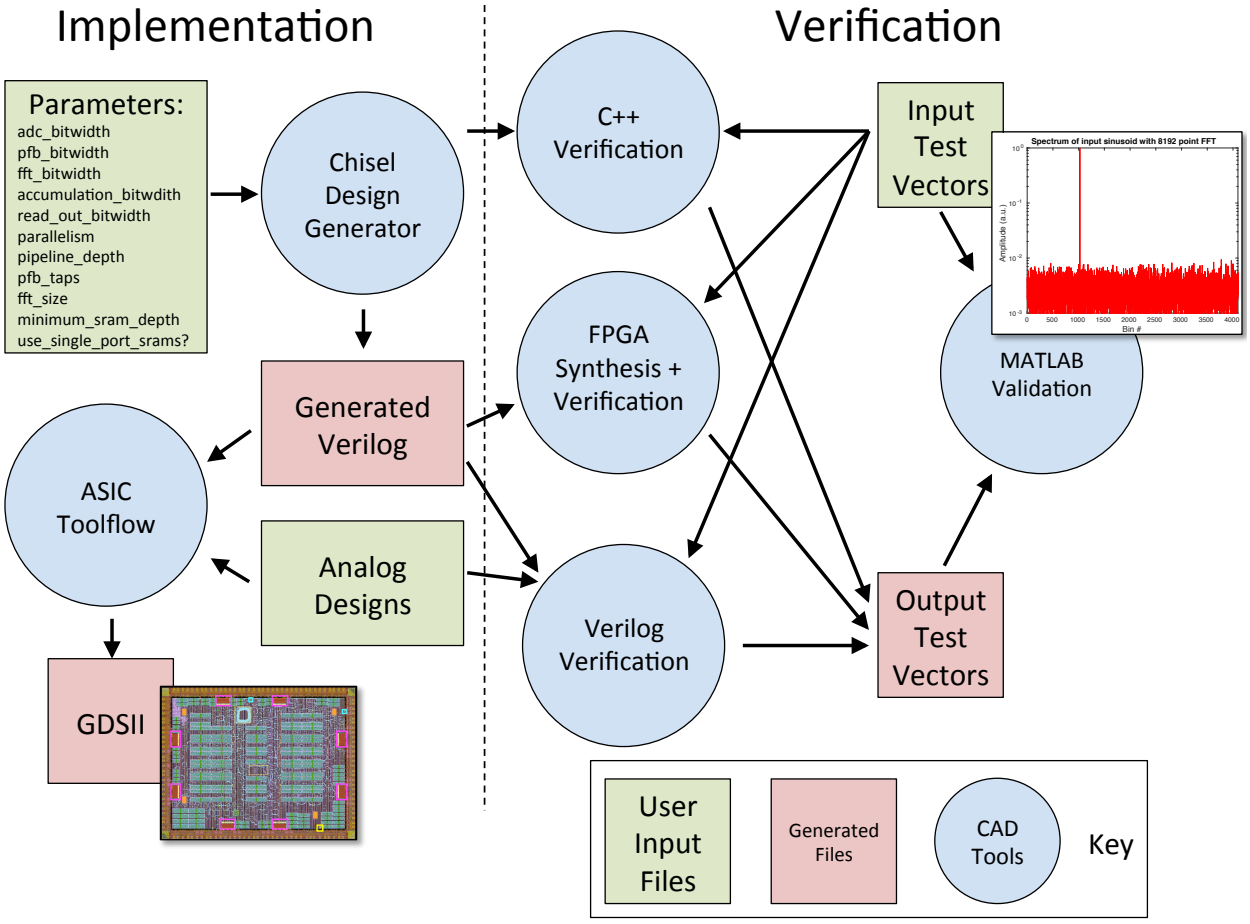


Figure 5.7: Generator flow diagram, showing verification paths on the right and the implementation path on the left.

5.2.4 ASIC Design and Verification

The parameterized generators presented are verified at multiple stages of the ASIC flow, as seen in Figure 5.7. Each generator has unit tests run in C++ to immediately verify that the chosen parameters function as intended. These unit tests are written in Scala in the Chisel testbench framework. Input test vectors are generated in the Scala testbench and written to a file for reference. Resultant test vectors are also written to a file. These input and output files are read in to a MATLAB tester, which plots the data and compares the output against a MATLAB golden reference version of the intended computation. System-level tests are implemented the same way.

Once the system design is verified, Verilog instances of the units and system are generated. These Verilog files are mapped to an FPGA for more comprehensive testing. Longer testbenches are run to verify the system operates correctly over a wide range of input data. Results are again evaluated in a MATLAB environment.

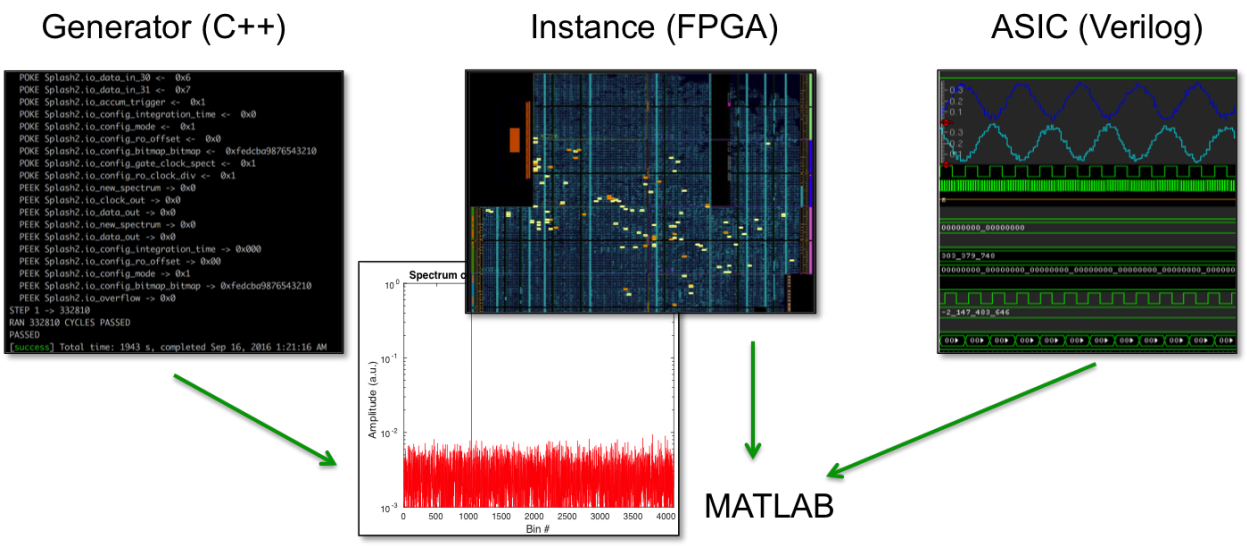
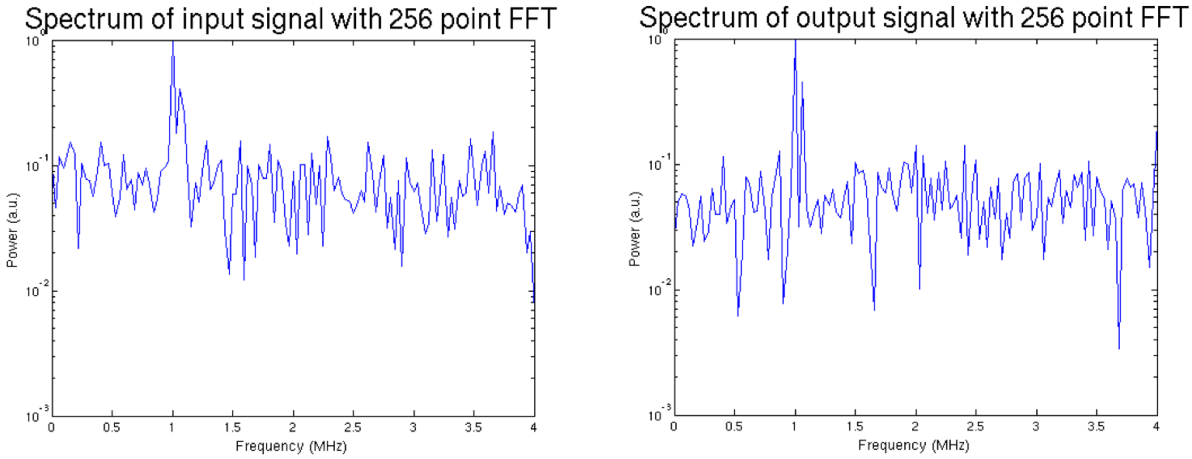


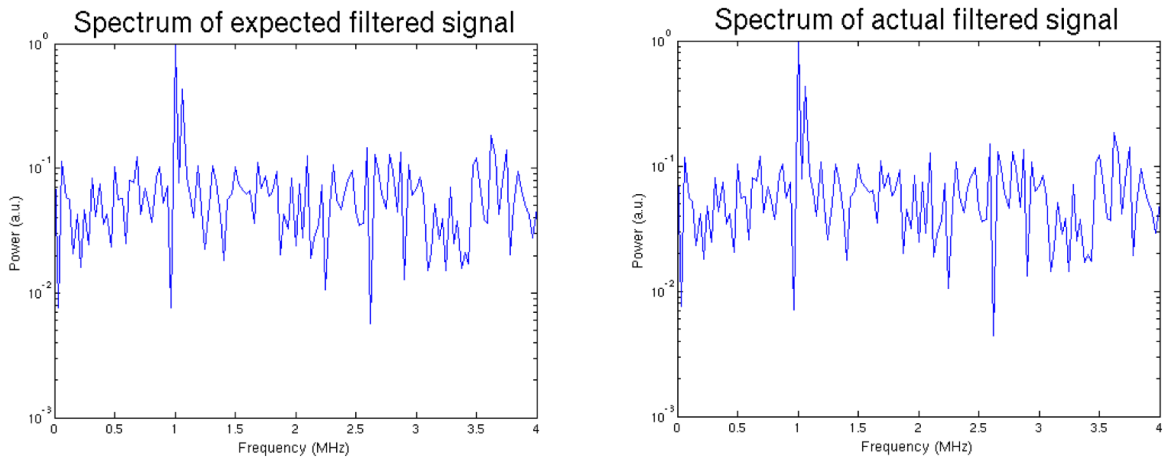
Figure 5.8: Verification of the spectrometer at different stages of the design process.

A selection of input stimuli and expected output data are selected for ASIC verification. A Verilog testbench applies the inputs and ensures the outputs are cycle-accurate to the outputs produced by the C++ and FPGA testbenches. The Verilog files are then combined with custom analog components and pushed through a standard ASIC flow. Post-synthesis and post-place-and-route designs are simulated with the same Verilog testbench as before. Figure 5.8 shows how verification at different stages in the design process reuse the same MATLAB testing framework. Figures 5.9 to 5.11 show example MATLAB and FPGA verification waveforms. Early simulation used a smaller design to reduce runtime.



(a) MATLAB FFT of a two-tone input signal. (b) MATLAB FFT of the signal after the PFB.

Figure 5.9: PFB verification. Note the better isolation of the two tones after the filter.



(a) Reference output of an input signal passed through a MATLAB PFB and FFT.

(b) Generator output of the same signal passed through a Chisel PFB and FFT.

Figure 5.10: PFB and FFT verification. Quantized coefficients were applied to the MATLAB PFB, and the input signal was quantized, but all other operations were floating point.

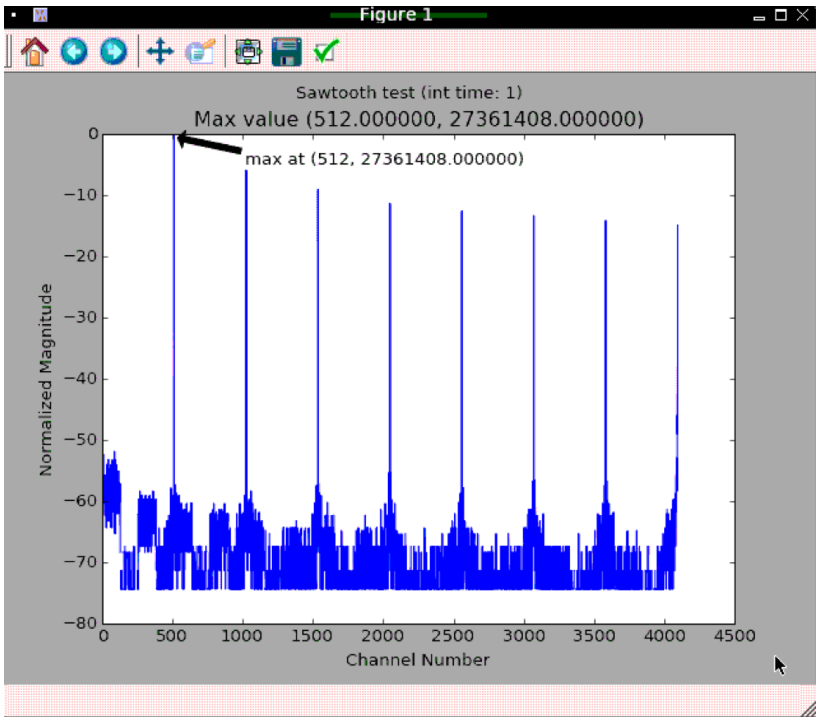


Figure 5.11: FPGA verification allowed for full design simulation. This figure shows the spectrometer response from a sawtooth input simulated on the FPGA.

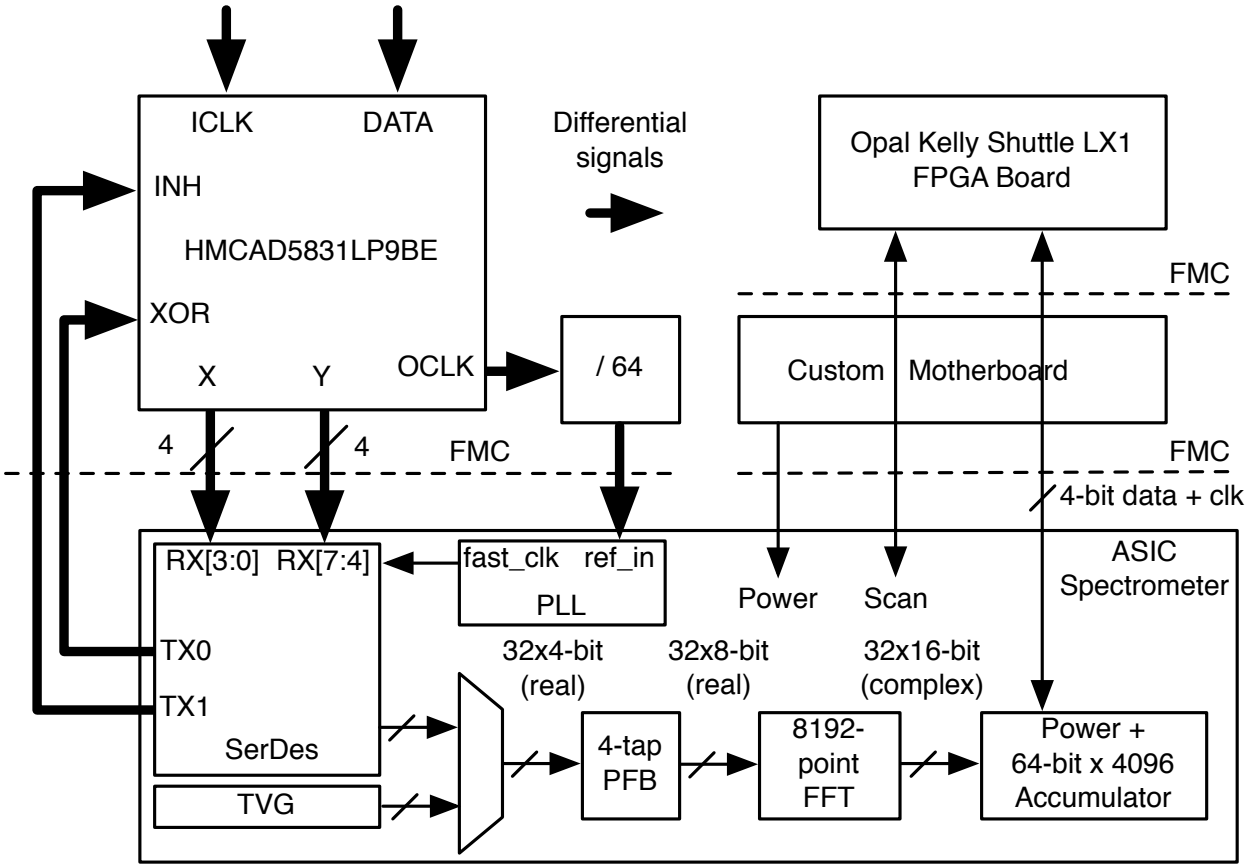


Figure 5.12: Implementation system block diagram with selected parameters.

5.3 ASIC Implementation

An instance of the generator was fabricated to demonstrate its functionality and effectiveness. The full system diagram is shown in Figure 5.12. The external ADC is a 3-bit (plus over/under range bit) 26 GS/s Analog Devices part. Its output data is XORed by a modulating signal, which we generate from a pseudo-random binary sequence (PRBS) calculated by a linear feedback shift register (LFSR) on chip. The system includes a set of eight high-speed serial links, designed to capture data from the ADC and deserialize it down to a rate that the digital back-end can handle. Also included is an all-digital bang-bang PLL for up-converting the input clock from the ADC, and several high-speed clock receivers for testing. Digital input controls and some output data are handled through a custom scan chain, while spectrometer outputs go to discrete pins and are aligned to a programmable, divided output clock for slow readout and synchronization.

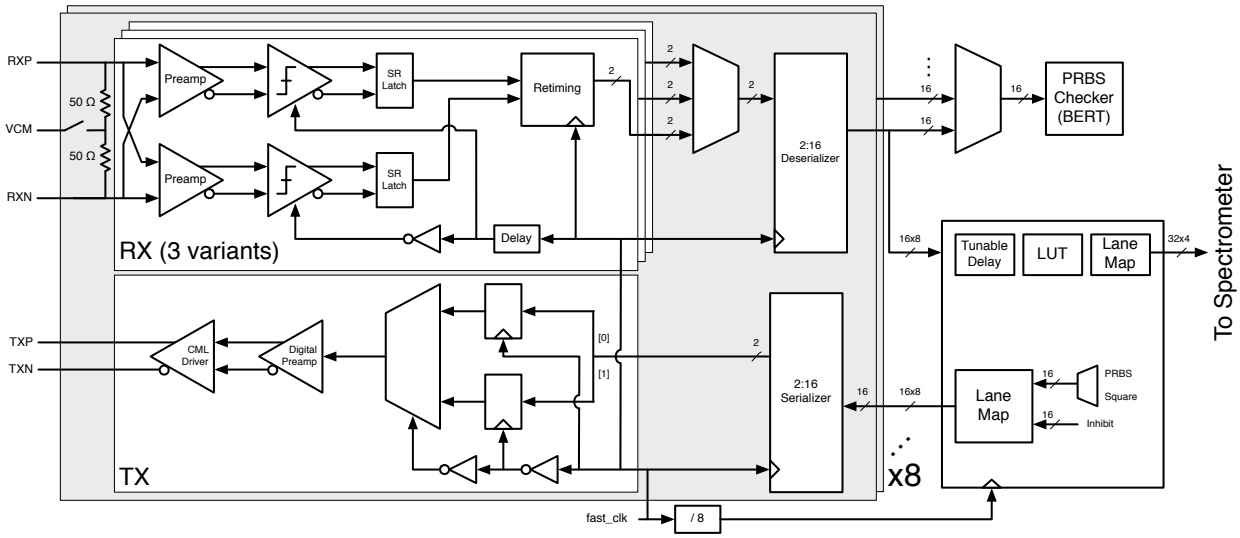


Figure 5.13: SerDes block diagram, including digital back-ends for monitoring and spectrometer pre-processing.

5.3.1 Serial Links

Figure 5.13 shows a block diagram of the on-chip serial link.[1] The serial links are eight 10 Gb/s DDR 50-Ω-terminated Current Mode Logic (CML) transceivers. Each transceiver contains one differential CML driver (TX) and three differential receivers (RX), each a different architecture. Only one receiver is active at a time. The TX comprises a digital single-ended-to-differential pre-driver and a differential CML driver pair. The RX comprises two analog pre-amplifiers, two samplers, two dynamic-to-static latches, retiming latches and clock tuning circuits. The input to the RX is terminated with two 50 Ω resistors tied to an external common-mode voltage. The common-mode can be bypassed to high impedance for DC-coupled applications. The TX and RX each receive reference currents (nominally 500 μA and 60 μA, respectively) provided by on-chip current mirrors. The RX also receives an additional offset reference current of nominally 5 μA. The TX reference current is supplied via 4 on-chip PMOS current mirrors which are themselves supplied by an on-chip NMOS current mirror. The RX currents are supplied directly from on-chip NMOS current mirrors. A 9-output version of the NMOS current mirror provides an additional output for monitoring the current off-chip. Each transceiver also has four clock pins: one for each receiver and one for the transmitter. The receiver has a number of clock, current, and control settings. The transmitter has far fewer–the only ones of interest being XOR (to invert the polarity of the output signals), enable (to turn the transmitters on), and swing (to adjust the output current and voltage).

The data to and from the transceivers are controlled by a digital bit error rate tracker

[1]The analog receiver was designed by Nandish Mehta. The analog transmitter, BERT, and digital pre-processor were designed by John Wright.

(BERT) control block and an 8:1 digital serializer/deserializer (SerDes). Receiver data first enters the SerDes, which deserializes it by a factor of 8. Next it enters the BERT block for monitoring and the spectrometer back-end for signal processing. Demodulation logic cancels a transmitted PRBS XOR signal. Finally, bit mapping logic converts the datatype of incoming words, for example from 4-bit unsigned integers to 4-bit signed integers. Any 4-bit bit mapping may be selected.

5.3.2 Digital Instance

Sideband separation calibration was removed to simplify the instance, so the chip implements a single sideband spectrometer, as seen in Figure 5.12. A test mode bypasses the input data and instead sends pseudo-random data from a test vector generator (TVG) into the spectrometer. These real data then enter the PFB, which has four taps, 8-bit coefficients, and 8-bit outputs. An 8192-point FFT receives the 8-bit real data, 32 words at a time, and produces 32 16-bit complex data (16-bit real, 16-bit imaginary) spectral bins each cycle. Delays in both the PFB and FFT above 64 total bits are implemented as single-port SRAMs. Bitwidths grow in the FFT one bit per stage until 16-bits are reached, followed by five stages of truncation. About two pipeline registers per FFT stage reduce the critical path and get retimed during synthesis. A 64-bit accumulator receives the 16-bit real bin powers, and accumulates from 1 to 65,520 spectra, programmable in increments of 16. Alternatively, the accumulator can take a snapshot of the input for testing. The output is fed to an SRAM buffer, which stores the data for slow readout while the accumulator accumulates the next set of spectra. Any contiguous 32 bits of the final 64-bit results may be read out, split into 4-bit chunks. The output data rate is a programmable division of the digital on-chip data rate.

5.3.3 Clocking

An all-digital bang-bang phase-locked loop (BBPLL) provides a synchronized on-chip clock for the serial links.[2] The system level implementation of the BPPLL is illustrated in Figure 5.14. In the randomization block there has to exist a pseudo-random number generator that will generate a pseudo-random number between 0 and the frequency divider/multiplication ratio N, which is the integer in the case of an integer-N PLL. This pseudo-random number defines the position of a DCO frequency control pulse rising edge within the reference clock period. The pseudo-random generator uses a 16-bit modular linear-feedback shift register (LFSR), out of which a 9-bit pseudo-random number Z is obtained. Simply calculating Z mod N yields the pseudo-random number in a necessary range. Modulo operation actually produces a remainder in Euclidean division, and for its implementation a standard digital divider is used. Having the output of the digital loop filter (corresponding to the pulse width in the DCO dithering frequency control signal) and the pseudo-random number

[2]The BBPLL was designed by Vladimir Milovanović.

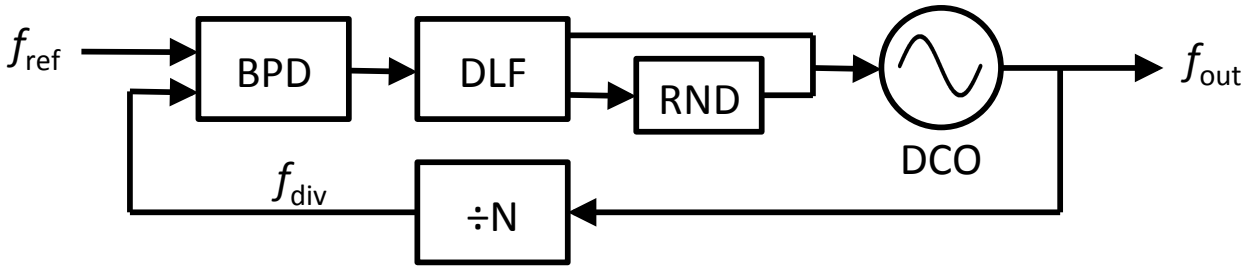


Figure 5.14: Block diagram of the BBPLL with pseudo-random fractional dithering.

in a desired range (corresponding to the pulse phase within the reference cycle), with a help of two down-counters (operating at the output frequency) and a bit of combinational logic (operating at the reference frequency) it is possible to create the DCO dithering control signal. It is also important to highlight that since all of the described PLL subcomponents are digital in their nature they can be implemented with static CMOS logic gates and thus completely synthesized, grasping all the benefits of the digital design flow. This includes the DCO, as it is fully standard-cell based and its structure is tiled, so it can be assembled in within the digital flow. However, for this design, the DCO was laid out by hand.

To ensure testability and usability, we used a complex clocking scheme with lots of controllability. Figure 5.15 shows a block diagram of the primary clocking system, excluding the scan chain clocking. The inputs to the clocking system are two pairs of LVDS signals and a reference clock for the BBPLL. The BBPLL reference clock enters the BBPLL, from which the BBPLL generated clock is created. The two LVDS clocks and the BBPLL generated clock feed into the SerDes clock mux. There is no difference between the LVDS receivers, so either

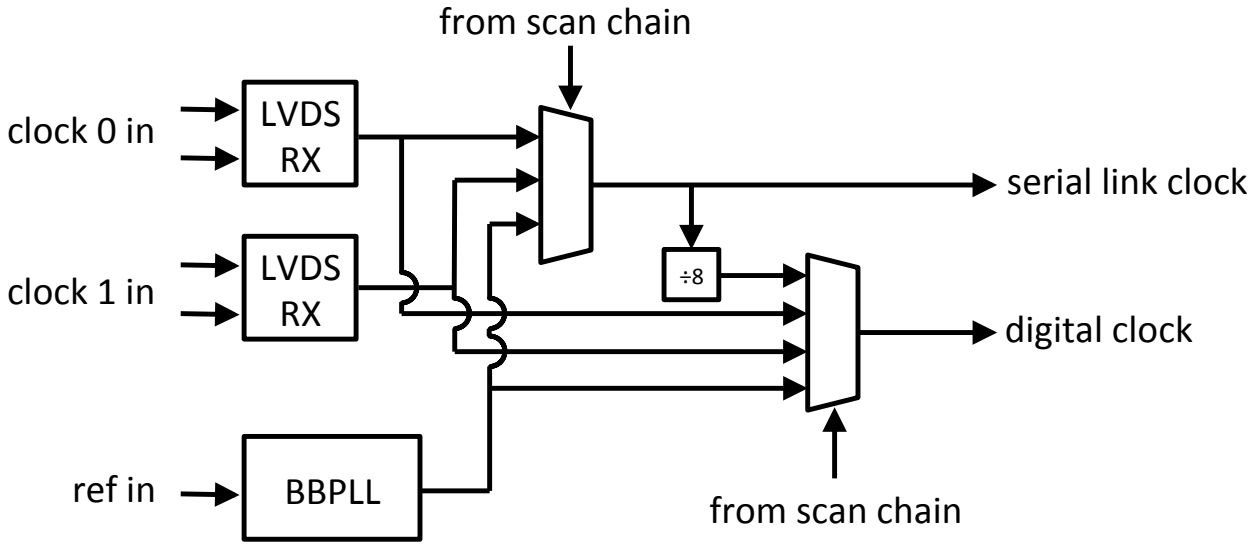


Figure 5.15: System clocking diagram.

source could be used as the 5 GHz source. The digital clock mux has the same three input clocks, but also has an input for the divided SerDes clock. This allows us to synchronize the digital data with the SerDes data. Asynchronous clock dividers were used. A manual buffer was inserted on the SerDes clock for constraining the tools. Clock select signals come from the scan chain. A dedicated pad-level signal allows the user to force the digital clock to the 1 GHz source. This is done because the scan chain runs off the digital clock (see the chapter on the scan chain). This creates a clock loop, since the digital clock clocks the flip-flop which holds the digital clock select signal. It is assumed this is okay, as a change in this signal will take time to propagate to the mux.

5.4 Chip Details and Testing Results

The design was fabricated in a 28nm UTBB-FDSOI process; Figure 5.16 shows a die photo of the 1.8 mm by 2.3 mm chip, annotated with the floorplan. The chip was designed to work with several ADCs, and was tested with the 3-bit 26 GS/s ADC. To prevent DC drift, this ADC features external modulation, the signal for which we provide and cancel from a PRBS on our chip. A custom daughterboard and motherboard have been developed to test the complete system.

The test vector generator was used to verify correct operation of the digital instance independently of the serial links. Figure 5.17 shows measured data from the TVG. The red line shows captured TVG outputs in bypass mode, meaning the PRBS data are directly

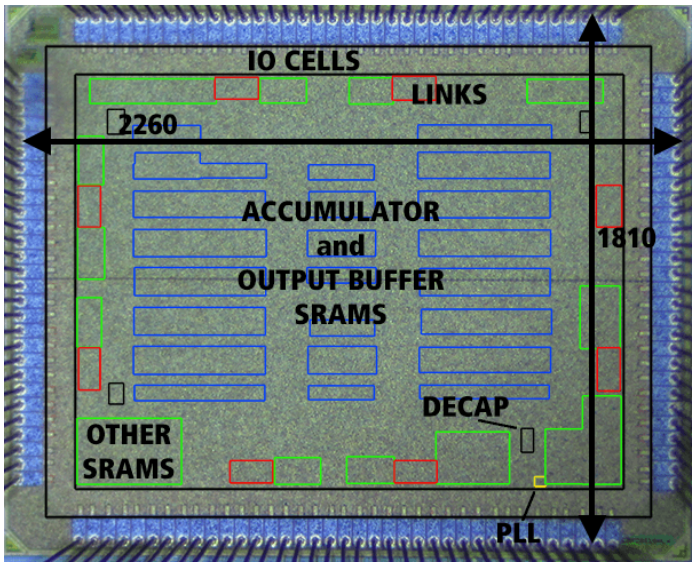


Figure 5.16: Chip die photo, with large components annotated. Note the distributed serial links and memory-dominated floorplan. The size was 4.2 mm^2. Annotated dimensions are in μm.

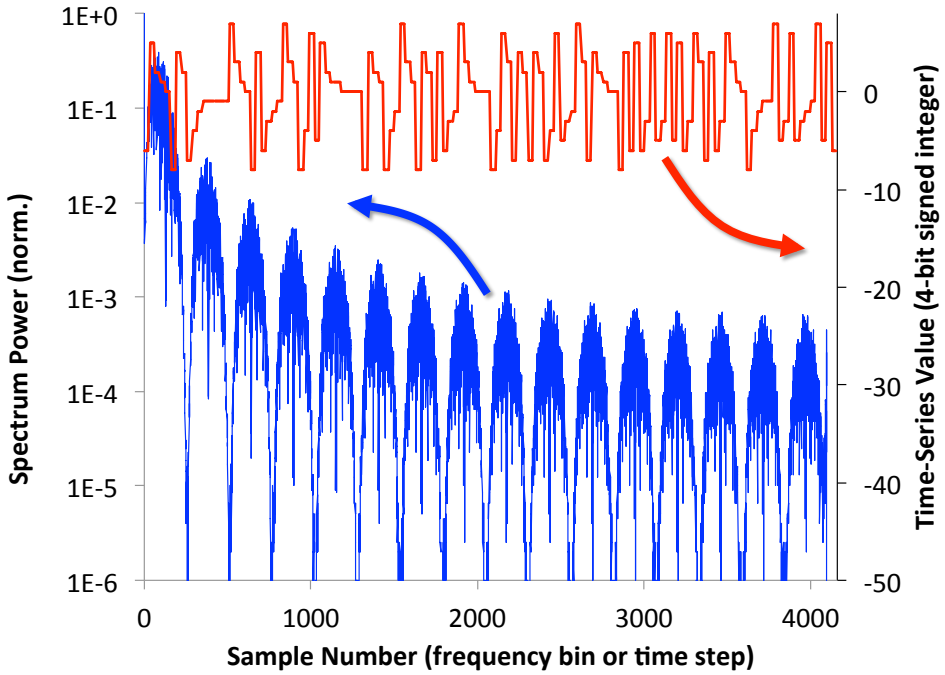


Figure 5.17: Time-series snapshot and spectrum power from the PRBS TVG. Both match cycle-accurate, bit-level simulation results, proving functionality.

buffered in the accumulator and read out to the host. Since the PRBS is replicated 32 times, one for each parallel input lane, the time-series plot contains groups of 32-sample values. The frequency response is shown in blue. Both time and frequency TVG outputs were compared to the expected values from the Verilog testbench for bit-level matching. Figure 5.18 shows the tradeoff between digital system power and operating frequency. Note that this power excludes power consumed by the ADC and serial link front-ends. The digital spectrometer in TVG mode works with zero errors up to 530 MHz, which translates to a 17 GS/s ADC (8.5 GHz bandwidth).

A built-in data and error monitor supports verification and testing of the serial links. The receivers function with BER below 10^{-7} at 5 GHz double data rate (DDR), which translates to a 20 GS/s ADC (10 GHz bandwidth). The transmitters operate as intended, and they support automatic data modulation, variable output swings, and independent lane mapping.

Figure 5.19 show an example of the full system operating at 1 Gs/s. Accumulating samples reduces the noise floor of the system, allowing signals buried in noise to be exposed. Together, the system functions up to 1.5 GS/s, limited by noise injected by the digital clock through the digital supply into programmable clock delay elements for the receivers. This periodic supply noise translates into phase noise in the serial link clock. When this phase noise causes the serial links to sample at the wrong time, large spectral spurs are evident in the output data. Careful tuning of the phase between the ADC and ASIC clocks helps, but at higher

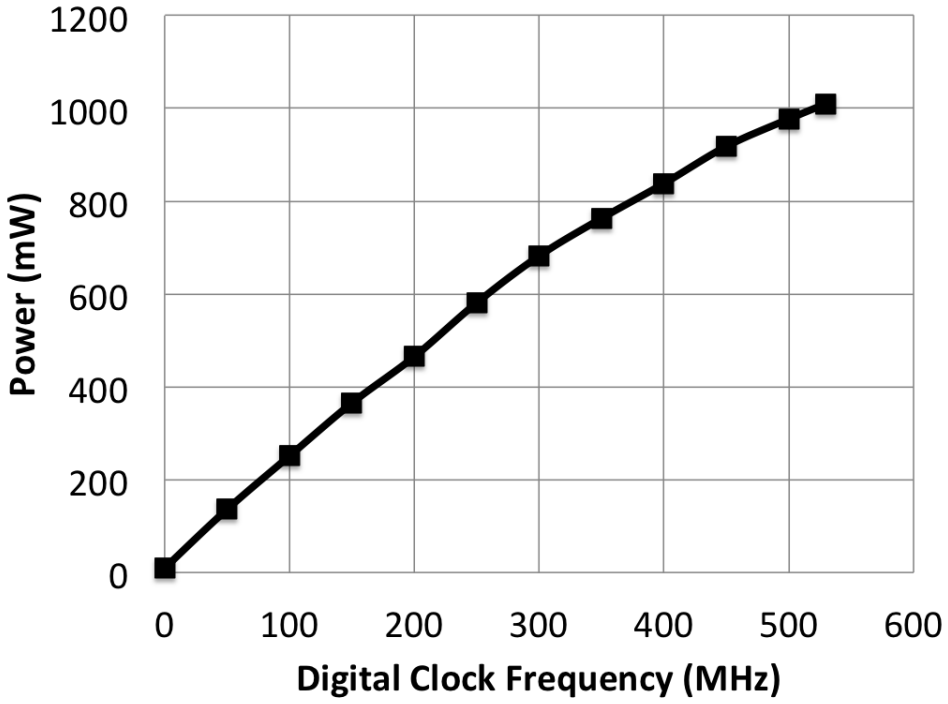


Figure 5.18: Power of the digital supply at 1 V at various digital clock frequencies.

frequencies, the errors are unavoidable. Figure 5.20 shows example measured spectra of the system at 1 Gs/s displaying both correct functionality and the sampling phase error. With the digital clock at 530 MHz, the chip consumes 1.0 W. The ADC consumes 4.2 W regardless of frequency.

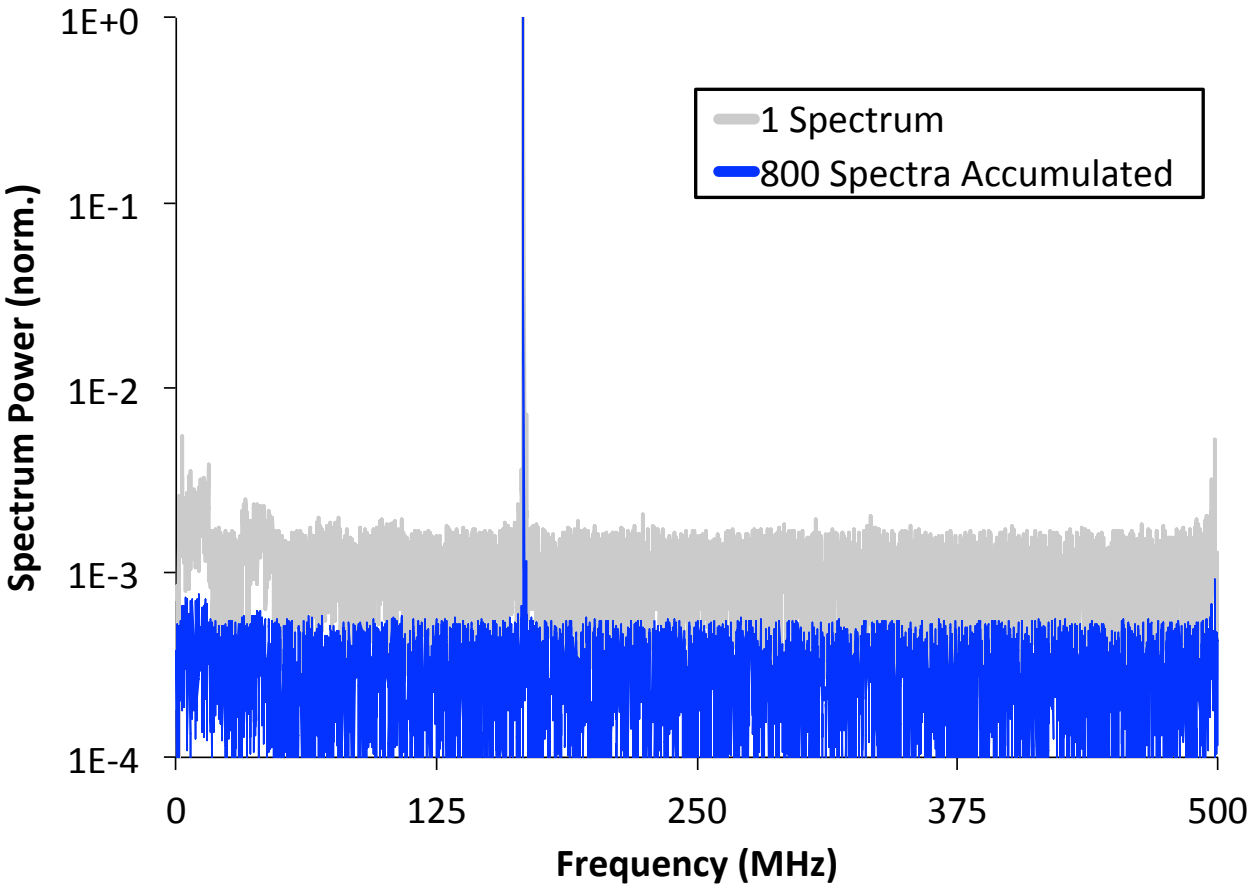
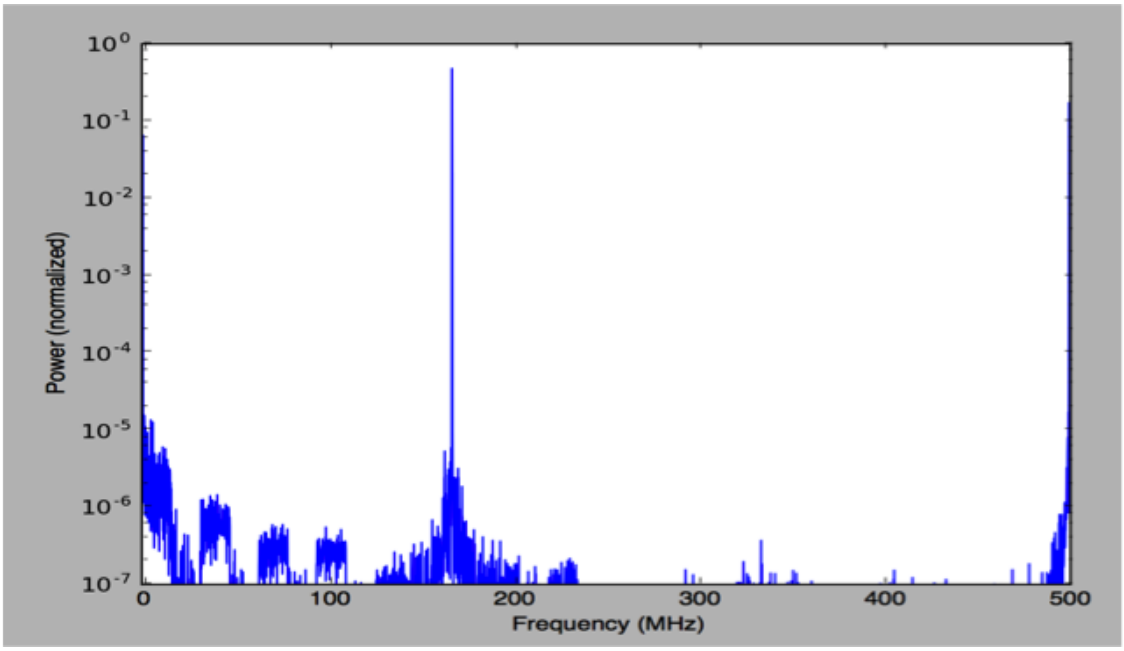
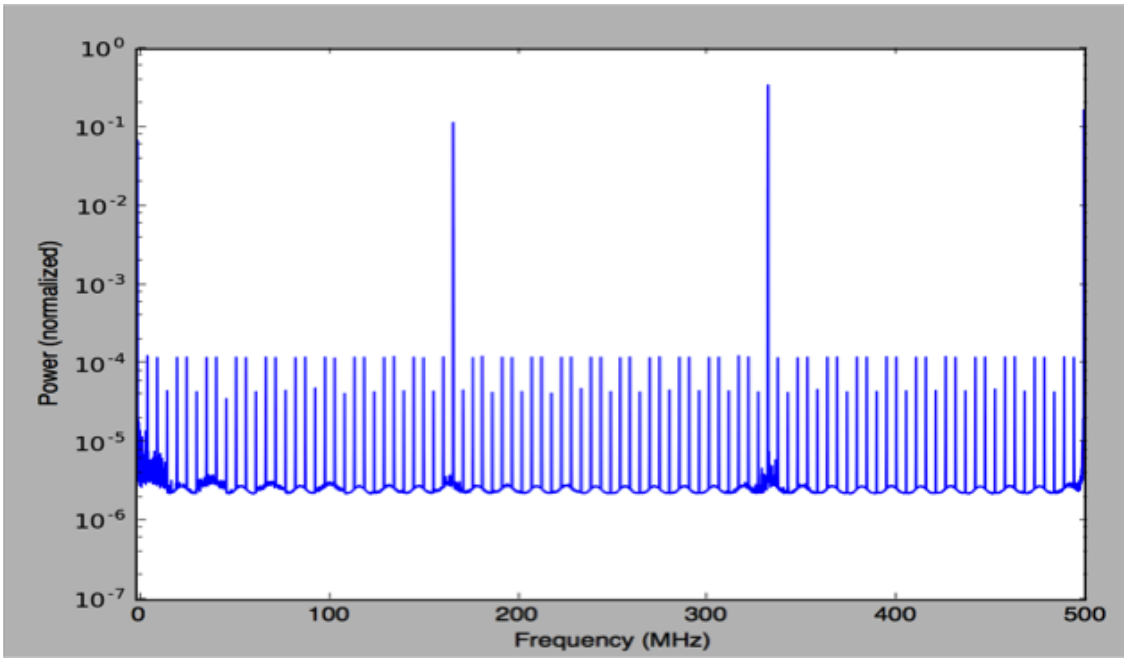


Figure 5.19: Example full system measured spectra at 1 GHz sampling frequency. Signal and system noise is reduced by accumulating many spectra.



(a) Low sampling phase noise.



(b) High sampling phase noise.

Figure 5.20: Measured spectra at 1 Gs/s with a 166 MHz input signal, accumulated over 16384 spectra (134 ms).

The full system without the sampling phase error is compared with other published digital ASIC spectrometers in Table 5.4. The presented design benefits from its large bandwidth and large FFT, despite the power overhead of using an external ADC.

Table 5.4: Comparison of state-of-the-art ASIC spectrometers.

	This Work	**CICC '15 [14]**	**CICC '09 [38]**
Technology	28nm FDSOI	65nm CMOS	90nm CMOS
Bandwidth	**8.5 GHz**	1.1 GHz	0.75 GHz
FFT Size	**8192 pts**	512 pts	**8192 pts**
Max Accumulation Length	65520 Samples	1024 Samples	**16M Samples**
Integrated ADC	No	**Yes**	No
Digital Power	1000 mW		1500 mW
System Power	5200 mW	**188 mW**	
FOM (pts×GHz)/mW	**13.4**	3.0	4.1

5.5 Agile Principles Applied

During the tape-out of this chip, the principles of agile hardware design were developed to ensure a functional design produced on time. A small design team (about five designers and three managers) limited the complexity of the chip, but close collaboration and fast communication enabled us to rapidly converge on a system that integrates each of our individual components. The initial spectrometer generator was designed in Chisel over the course of 8 weeks. The set of digital generators mentioned supported last-minute changes when physical design limited chip capabilities. For example, the original specification asked for sideband separation and calibration. These features were included in the digital generator, but they were not verified sufficiently to include in the final design. Also, our reliance on an external ADC and serial links required extra pins. The chip could not fit the pins of another set of 8 serial links to support another ADC for separate I and Q inputs. Finally, the 8192-point PFB, FFT, and accumulator consumed significant resources on the chip. Doubling this for sideband separation would have required a much larger chip, something we weren't sure we could tape out in time. Despite these changes in the specification, the digital generators required just a few parameter changes to create an updated, validated design omitting the sideband separation and calibration. In conclusion, having a flexible set of generators allowed us to respond to changing specifications and design revelations, and still tape-out on time.

5.6 Conclusion

This chapter presented a method for generating sideband-separating ASIC spectrometers using the Chisel hardware construction language, and it discussed an implementation that was

fabricated and tested in 28nm UTBB-FDSOI. The results suggest that writing generators to support an array of design options rather than single instances targeted for specific applications may lead to less work overall, as they can adjust for specification changes on-the-fly. Also, chips produced from these generators can still compete with the high performance and low power of state-of-the-art, hand-tuned designs. Future work will include analog component generators, so that high-speed links or ADCs may be automatically generated to meet desired specifications for inclusion on chip. Finally, radiation hardening is paramount for hardware targeting deployment in space, so future work is needed to precisely and automatically characterize the hardness of a design and improve the soft error tolerance as needed.

Chapter 6

Example: A Signal Analysis SoC

6.1 Introduction

The second example design presented in this thesis is a complete signal analysis SoC. The motivations behind producing this chip were twofold. First, the ideas and framework given in previous chapters require an implementation to prove the effectiveness with which they can be used to design complex SoCs. Second, there is a growing need for lightly integrated, low-power radar signal analysis. The automotive industry uses radar for advanced driver assistance systems (ADAS) and autonomous driving to detect nearby objects. Also, radar in drones and airplanes is used for topography and collision detection and avoidance. The specifications and designs arose in collaboration with Northrop Grumman Corporation (NGC) and Cadence Design Systems (CDN), and the design framework and paradigm come from the ideas given in earlier chapters. Although the design initially targeted radar receive processing, the signal processing chain combined with the general-purpose processor are general enough to support a variety of signal analysis applications.

This work demonstrates a complex SoC containing a signal processing accelerator and a general-purpose processor from a set of open-source digital and analog generators [78] written in Chisel [51] and BAG [55], as presented in previous chapters. The general-purpose processor is a generated RISC-V core with new ISA accelerators, and the signal processing accelerator comes from a new generator of streaming signal processing functions. Individual processing elements borrow their architecture and sometimes their code from the ASIC spectrometer presented in the previous chapter. Section 6.2 describes the architecture produced by the generator. Testing and measurement results are given in Section 6.5. The presented instance is general enough to apply in a variety of signal processing contexts, and Section 6.6 demonstrates several of these applications. Finally, Section 6.7 walks through the evolution of this chip, noting how an agile design flow influenced the process and ensured the tape-out deadline was met.

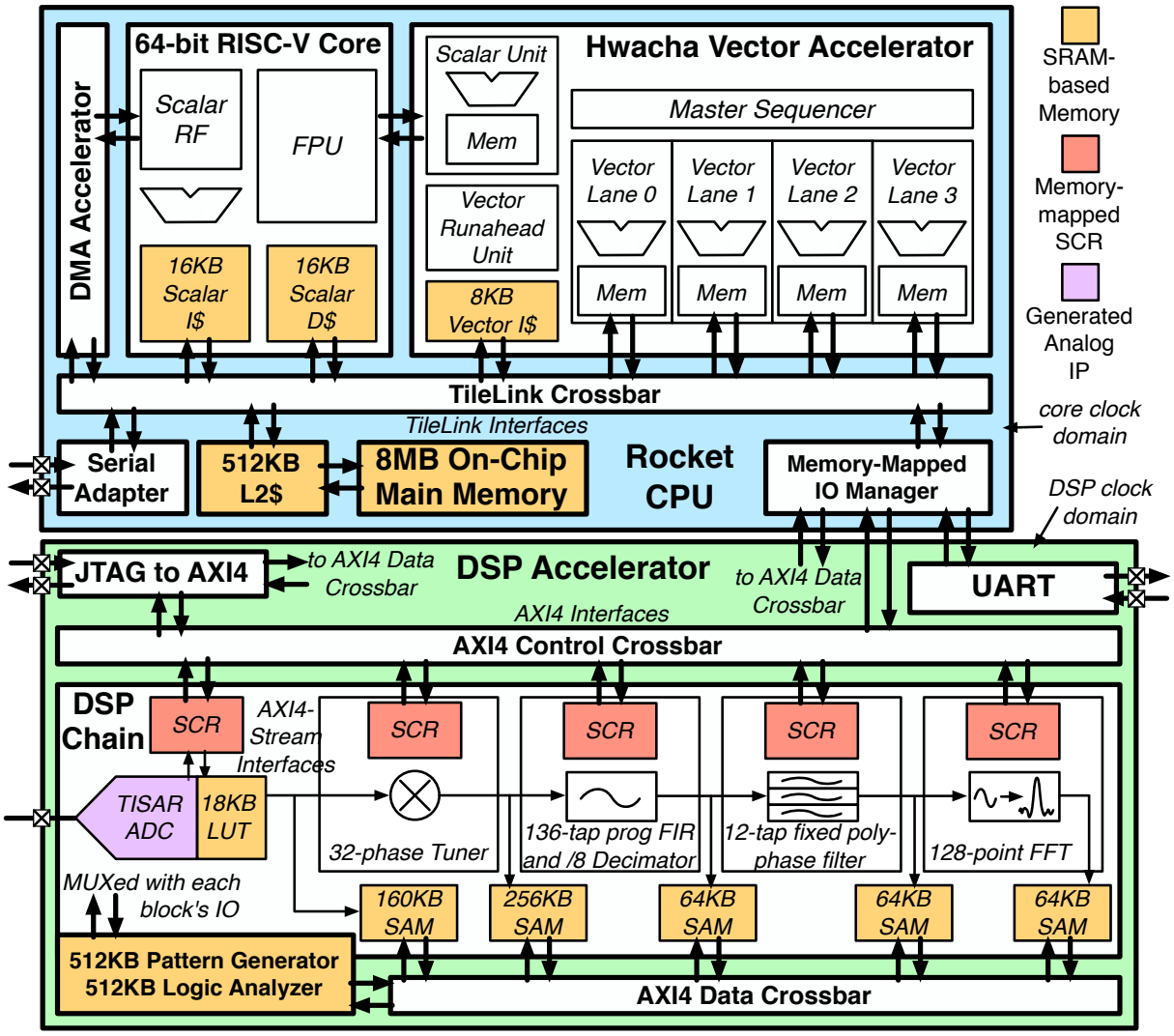


Figure 6.1: Block diagram of the SoC architecture.

6.2 SoC Architecture

Figure 6.1 shows the system architecture. The SoC is divided into a general-purpose processor and a dedicated signal processing accelerator. Communication between the processor and the accelerator is handled through a memory-mapped IO manager.

6.2.1 General-Purpose Processor

The chip includes a general-purpose processor that connects a host with the chip, programs the signal processing accelerator, moves data, and computes what other on-chip accelerators cannot. The 64-bit single-issue in-order RISC-V Rocket CPU includes a single-/double-

precision (SP/DP) floating-point unit (FPU). Provisions in the RISC-V ISA support extensions to the instruction set, and this chip includes two accelerators using ISA extensions. A direct memory access (DMA) accelerator extends the ISA through the Rocket Custom Coprocessor (RoCC) interface and offloads memory movement between the processors from the CPU, allowing both the general-purpose processor and the signal processing accelerator to continue computing while data moves around. The 4-lane high-performance Hwacha vector accelerator implements a decoupled vector-fetch architecture and can perform compute-intensive parallel workloads not handled by the signal processing accelerator. A new addition to the Rocket Chip memory hierarchy was the inclusion of on-chip main memory. This limits the memory size to something that can fit on-chip, but speeds up memory access. The main memory is split into channels and banks, with a programmable number of channels and banks per channel. This SRAM array backs the L2 through a TileLink interface. A serial adapter tethers the CPU to the FPGA host, which writes programs to the on-chip 8 MB main memory before booting the core. An asynchronous FIFO permits low-speed, off-chip data from the host to communicate with the high-speed Rocket CPU. A custom clock receiver supplies the core clock, which clocks the CPU, accelerators, entire memory hierarchy, and TileLink interconnects.

6.2.2 Digital Signal Processing Accelerator

The digital signal processing accelerator architecture is a streaming processor with processing elements communicating through AXI4-Stream interfaces. The Chisel generator contains memory-mapped IO registers that take commands from either the CPU or a JTAG debug port. Asynchronous FIFOs buffer data between the core clock, DSP clock, and JTAG clock domains. Separate AXI4 crossbars access status and control registers (SCRs) and data buffers (SAMs). For testing, a 512 KB pattern generator and 512 KB logic analyzer allow direct access to inputs and outputs of individual processing elements (PEs). A chain of PEs receives data from a BAG-generated 8-bit time-interleaved successive approximation (TISAR) ADC with lookup table (LUT)-based static calibration, and it also provides a clock to the PEs and AXI4 crossbars. The UART provides a backup interface into the RISC-V core, and the JTAG provides a backup interface into the accelerator.

6.2.3 Processing Elements

The selected PEs implement a signal analysis accelerator targeting spectral analysis or radar receive chain processing. Figure 6.2 shows the parameterization of PEs in green atop the final implementation diagram. Later sections describe individual elements. The ADC LUT outputs 9 bits for testing, so a custom bit manipulator (BM) PE truncates this to 8 bits. The next two PEs comprise a digital down-converter (DDC). A 32-entry LUT-based digital tuner mixes the input signal with a complex sinusoid, and a fully-programmable 136-tap complex FIR filter shapes the signal and decimates it by 8. A 12-tap fixed-function polyphase filter multiplies the time-series data by a sinc function to window each FFT bin and reduce frequency-

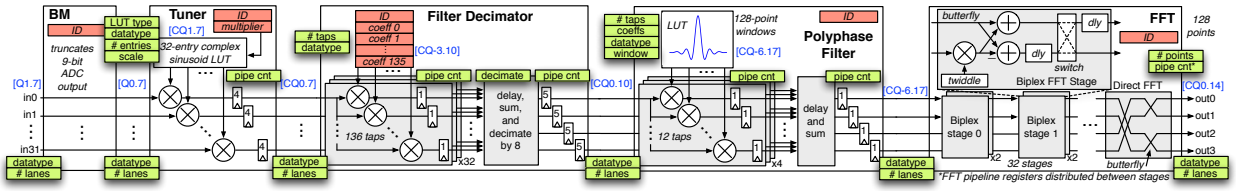


Figure 6.2: Detailed diagram of the processing elements in the DSP accelerator. Red boxes indicate memory-mapped IO SCRs. Green overlays show generator parameters. Blue text gives fixed-point data type parameters chosen. CQ = complex fixed-point number.

domain spectral leakage. A 128-point radix-2 FFT, comprised of 32-point biplex pipelined FFTs and a 4-point direct-form FFT, produces the complex spectrum output. The chain generator supports arbitrary ordering and duplication of PEs, so the chosen arrangement of PEs represents just one possible DSP accelerator configuration.

6.2.4 Bit Manipulator

The simplest processing element, the bit manipulator directly passes the input data through to the output. However, given that input and output datatypes are parameterized, automatic datatype conversion provides the utility of this block. For example, this chip used the bit manipulator to truncate the 9-bit ADC calibration output into 8 bits, since the MSB was only useful for testing. Also, as all processing elements support connections to the logic analyzer, pattern generator, and a SAM, including a bit manipulator without any data-level modifications allows the designer to place these DfT connections anywhere. The radar chip includes two bit manipulators right at the beginning of the chain. The first one connects the 9-bit ADC calibration RAM outputs to a SAM and the logic analyzer for direct monitoring of calibrated ADC data. The second block truncates these 9 bits into 8, as the ADC produces only 8 real bits. Listing 6.1 shows an example of a bit manipulator definition that truncates the MSB and provides connections to a SAM and the logic analyzer. Figure 6.3 gives a block diagram of what the configured bit manipulator looks like, without the external DfT connections.

Listing 6.1: Bit Manipulator configuration example

```
def bmConfig() = BitManipulationConfig(lanes = 32)
def bmInput() = FixedPoint(9.W, 0.BP)
def bmOutput() = FixedPoint(8.W, 0.BP)
def bmConnect() = BlockConnectionParameters(connectPG = false, connectLA = true,
    addSAM = true)
def bmSAMConfig() = Some(SAMConfig(subpackets = 32, bufferDepth = 128))
```

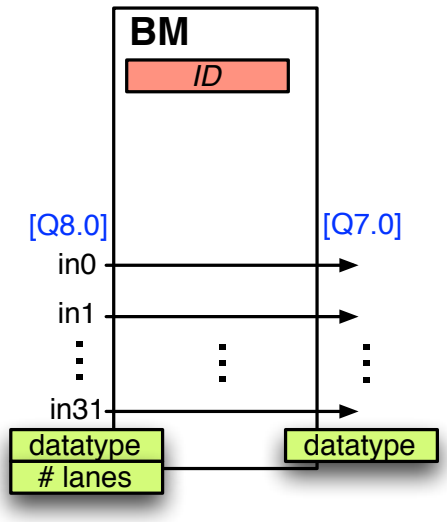


Figure 6.3: Bit manipulator, showing how the output is directly connected to the input, but datatype conversion is implicit.

6.2.5 Tuner

The tuner processing element implements a digital downconversion. The inputs can be either complex or real. If the inputs are real, they are mapped to the real part of complex values before being multiplied by the mixer coefficients. The coefficients and outputs are always complex.

There are loosely two main configuration options for the mixer. The options are called *SimpleFixed* or *Fixed*. In the SimpleFixed scenario, input lanes are multiplied by programmable phase values defined in the SCRFile. The SCRFile provides one phase value for each lane. Phase values are always signed and complex.

In the Fixed scenario, the phase values are restricted to being one of $\frac{2\pi}{N}$ phases where N is defined by the parameter mixerTableSize. The mixerTableSize must be an integer multiple of the number of lanes. The SCR file allows control of input k, also called FixedTunerMultiplier, which will specify to use phase 0, $\frac{2\pi k}{N}$, $2 \times \frac{2\pi k}{N}$, $3 \times \frac{2\pi k}{N}$. This sequence will inherently roll over and repeat after at most N phases and thus each input lane will always use the same value once k is chosen. The values of sin and cos may be multiplied by a small constant slightly less than 1 (e.g. .999) to prevent rounding or truncation from causing bit growth. See Figure 6.4. Note that some lanes may be able to use any phase while other lanes will be restricted to a subset of the phases. For example, lane 0 will always use a constant of approximately $1 + 0i$.

Listing 6.2 shows sample configuration code, and Figure 6.5 gives the block diagram for this configuration. Note this example uses mixed data types, so the input is real valued but the output and coefficients are complex.

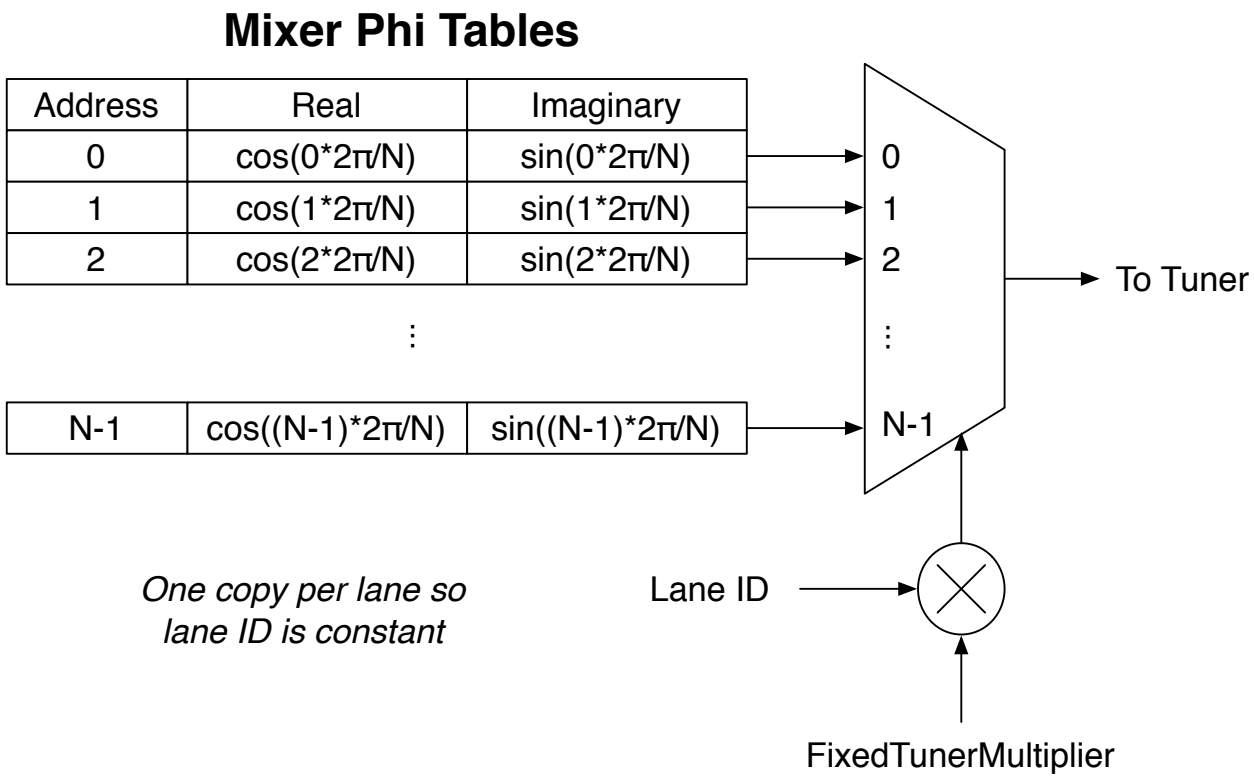


Figure 6.4: Tuner coefficient LUT diagram for the *Fixed* configuration. This architecture simplifies the hardware by hard-coding coefficients and requiring a single multiplier input.

Listing 6.2: Tuner configuration example

```
def tunerConfig() = TunerConfig(pipelineDepth = 4, lanes = 32, phaseGenerator =
    "Fixed", mixerTableSize = 32, shrink = 1.0)
def tunerInput() = FixedPoint(8.W, 7.BP)
def tunerMixer() = DspComplex(FixedPoint(9.W, 7.BP), FixedPoint(9.W, 7.BP))
def tunerOutput() = DspComplex(FixedPoint(8.W, 7.BP), FixedPoint(8.W, 7.BP))
def tunerConnect() = BlockConnectEverything
def tunerSAMConfig() = Some(SAMConfig(subpackets = 1, bufferDepth = 4096))
```

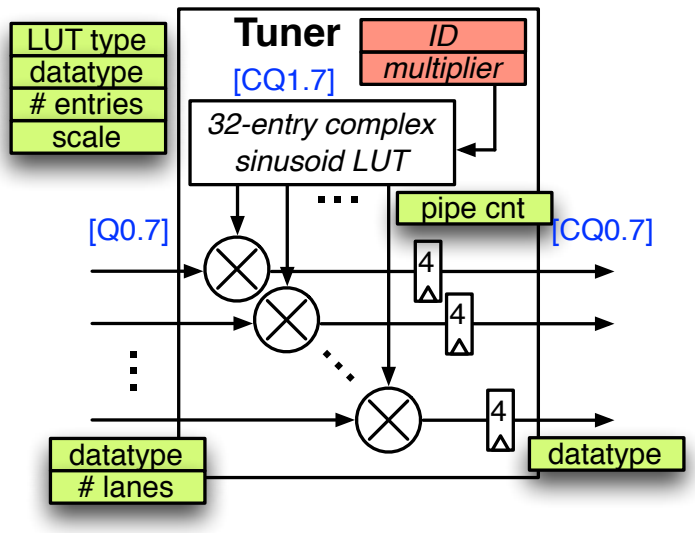


Figure 6.5: Example tuner block diagram. Note the implicit conversion from real to complex. Pipeline registers are all placed at the output. Direct synthesis tools to retime as needed for improved performance.

6.2.6 Decimating Filter

The filter implements an FIR filter, with decimation done by dividing down the number of parallel outputs. The user can choose any number of taps and parallel inputs. Multiplication is performed first, followed by summation. The number of outputs must be an integer divisor of the number of inputs. All outputs are calculated, and it is assumed that synthesis tools will remove unnecessary logic if there are fewer outputs than inputs. Tap coefficients are programmable through the SCR file, and can be a distinct data type from the input and output. Pipeline registers and decimation are handled at the output. A highly parallel implementation will benefit from pipeline registers and retiming within synthesis tools. Listing 6.3 shows an example configuration of a decimating filter. The processing delay parameter lets the user specify an explicit group delay. This value determines the number of pipeline registers applied to the TLAST and TVALID signals between the input and output in the AXI4-Stream protocol. Figure 6.6 shows a simplified block diagram of this filter decimator.

Listing 6.3: Decimating Filter configuration example

```
def firConfig() = FIRConfig(numberOfTaps = 136, lanesIn = 32, lanesOut = 4,
    processingDelay = 2, multiplyPipelineDepth = 1, outputPipelineDepth = 5)
def firInput() = DspComplex(FixedPoint(8.W, 7.BP), FixedPoint(8.W, 7.BP))
def firTaps() = DspComplex(FixedPoint(8.W, 10.BP), FixedPoint(8.W, 10.BP))
def firOutput() = DspComplex(FixedPoint(11.W, 10.BP), FixedPoint(11.W, 10.BP))
def firConnect() = BlockConnectEverything
def firSAMConfig() = Some(SAMConfig(subpackets = 1, bufferDepth = 4096))
```

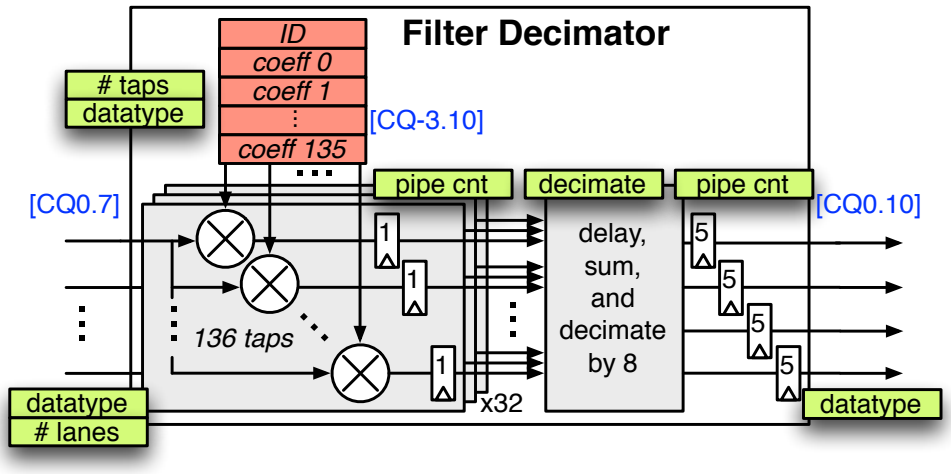


Figure 6.6: Simplified decimating filter block diagram, with delay and summation logic obfuscated.

6.2.7 Polyphase Filter

The polyphase filterbank (PFB) is implemented as described in the CASPER website [56], and this architecture is the same as the spectrometer ASIC architecture. In Chisel, the design is split into parallel lanes. Each parallel lane implements a FIR filter in transposed form, using Chisel Mems for delays, which may be mapped to SRAMs. Currently, only constant coefficients are allowed. However, a selection of windowing functions are provided in `Windows.scala`. Pipelining is split into registers placed at the output of each multiplier, and registers placed at the output. The number of taps is an independent parameter, but each lane has the same number of taps. When specifying a configuration, the tap datatype can be distinct from the the input and output datatypes, but a conversion function must be supplied, as seen in Listing 6.4. Similar to the decimating filter, the processing delay parameter specifies a custom group delay for pipelining the TLAST and TVALID signals. Figure 6.7 has the block diagram.

Listing 6.4: Polyphase Filter configuration example

```
def pfbConfig() = PFBConfig(windowFunc = BlackmanHarris.apply, processingDelay =
    192, numTaps = 12, outputWindowSize = 128, lanes = 4, multiplyPipelineDepth =
    1, outputPipelineDepth = 1)
def pfbInput() = DspComplex(FixedPoint(11.W, 10.BP), FixedPoint(11.W, 10.BP))
def pfbTap = DspComplex(FixedPoint(12.W, 17.BP), FixedPoint(12.W, 17.BP))
def pfbConvert(x: Double) = DspComplex(FixedPoint.fromDouble(x, 12.W, 17.BP),
    FixedPoint.fromDouble(0.0, 12.W, 17.BP))
def pfbOutput() = DspComplex(FixedPoint(12.W, 17.BP), FixedPoint(12.W, 17.BP))
def pfbConnect() = BlockConnectEverything
def pfbSAMConfig() = Some(SAMConfig(subpackets = 1, bufferDepth = 4096))
```

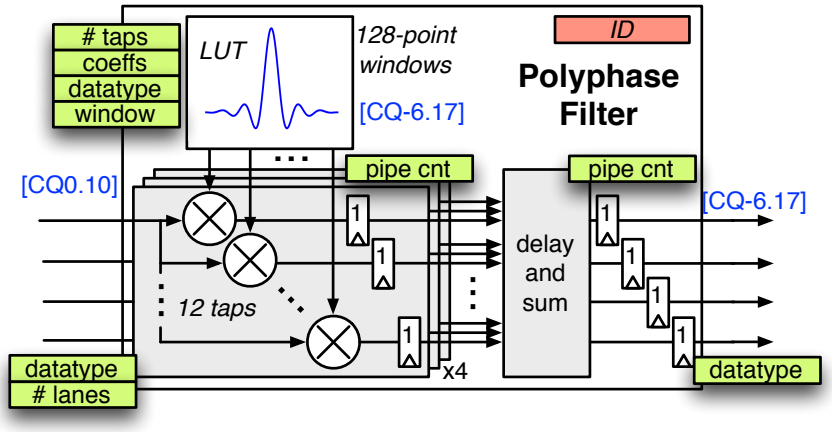


Figure 6.7: The polyphase filter looks almost identical to the decimating filter, but the polyphase filter lanes are kept distinct, while the decimating filter treats all lanes as one continuous input.

6.2.8 Fourier Transform

The fast Fourier transform (FFT) is implemented a similar way to the FFT in the spectrometer ASIC. The FFT supports any power of two size of 4 or greater ($n \geq 4$). The input rate may be divided down, resulting in a number of parallel input lanes different from the FFT size. But the input lanes (p) must be a power of 2, greater than or equal to 2, but less than or equal to the FFT size. Twiddle factors are hard-coded for you. The transform is split into pipelined biplex FFTs and a direct form FFT to multiplex the logic for large FFT sizes [75].

When the number of parallel inputs equals the FFT size, a simple, direct form, streaming FFT is used, as seen in Figure 6.8 with the number of biplex stages being zero. The dotted line marks an example *stage* boundary, or place where pipeline registers may be inserted. The input will never be pipelined, but the output might be pipelined in Chisel based on the desired pipeline depth. Pipeline registers are automatically inserted in reasonable locations.

When the input is serialized, the FFT may have fewer input lanes than the size of the FFT. In this case, the inputs are assumed to arrive in time order, time-multiplexed on the same wires. To accommodate this time multiplexing, the FFT architecture changes. Pipelined biplex FFTs are inserted before the direct form FFT. These FFTs efficiently reuse hardware and memories to calculate the FFT at a slower rate but higher latency. Figure 6.8 shows their architecture, as taken from the JPL technical report. The dly boxes are shift registers of decreasing delay values, and the boxes with dotted lines in them are 2-input barrel shifters, periodically crossing or passing the inputs at decreasing rates. An extra shift register on the stage 0 bottom input aligns the adjacent channels, and extra shift registers at the output unscramble the data before they arrive at the direct form FFT. The reorder block seen in the previous chapter was removed for simplicity, so real-valued FFTs must use complex inputs with zeros as imaginary components. Pipeline registers may be inserted after each butterfly,

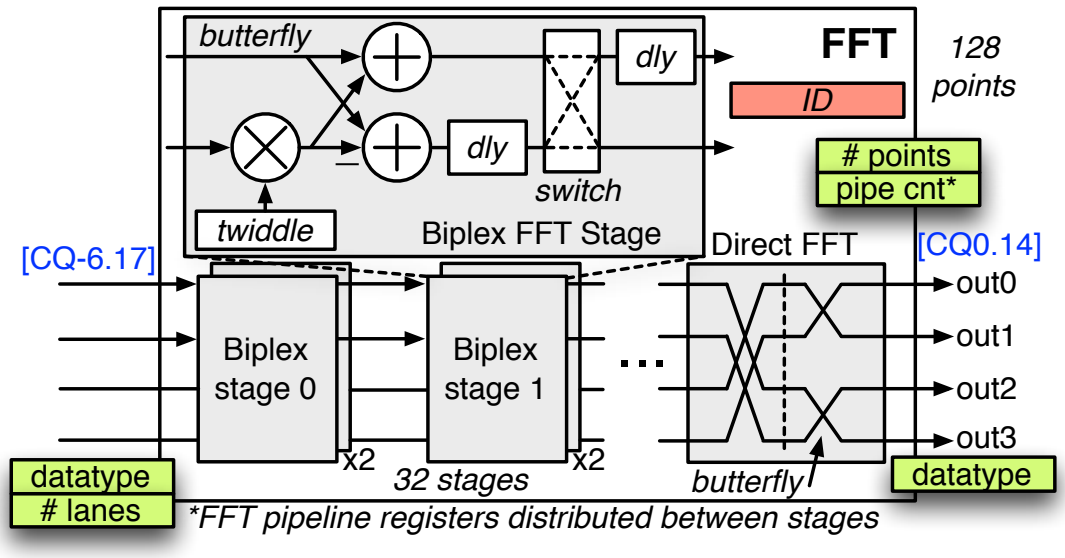


Figure 6.8: Fast Fourier transform block diagram. The ratio of biplex to direct FFT butterflies scales automatically with the specified number of points and lanes. Twiddle factors and intermediate bitwidths are calculated automatically for any design size.

but never at the input or output of the biplex portion. The input is assumed pipelined by previous blocks, and the output of the biplex has shift registers already. A final direct form FFT sits at the output of the biplex FFTs, finishing the Fourier transform. Pipeline registers favor the direct form FFT slightly, though the critical path through this circuit is still through $log_2(n)$ butterflies, so one pipeline register per stage (a pipeline depth of $log_2(n)$) is recommended. Bitwidths grow immediately to the estimated output bitwidth assuming one bit growth per stage. This growth is split between the biplex and direct FFT stages, so bitwidth growth happens twice. If the output has more or fewer bits than normal growth would imply, the result is automatically truncated or sign-extended as needed. Listing 6.5 and Figure 6.8 give parameterization code and block diagram examples, respectively.

Listing 6.5: FFT configuration example

```
def fftConfig() = FFTConfig(n = 128, lanes = 4, pipelineDepth = 7)
def fftInput() = FixedPoint(12.W, 17.BP) // gets complexed automatically
def fftOutput() = FixedPoint(15.W, 14.BP) // gets complexed automatically
def fftConnect() = BlockConnectEverything
def fftSAMConfig() = Some(SAMConfig(subpackets = 1, bufferDepth = 4096))
```

6.3 IP Integration

Components outside the purview of the generator framework required special integration to be fully supported. This section explores these blocks, including architectural and programmatic considerations required to incorporate outside designs. Though analog IP play a key role in any signal processing system, this work considers them an external IP that must be integrated, as the generation of analog designs are not part of the framework. Thus the ADC and clock receiver are the first IP considered. The UART design comes from Cadence, and its inclusion was simplified by nature of its standardized interface. Finally, any digital system is going to include SRAMs from a memory compiler and IO cells, both likely from the technology vendor. However, connecting these components into the system required external scripts.

6.3.1 ADC and Calibration

The ADC is integrated as an analog black box.A Chisel black box contains only the interface, with the expectation that the module itself will be replaced with a model for simulation or IP for synthesis and place-and-route. The top-level IO includes differential clock and data inputs, supplies, and references. These are punched through the hierarchy by mixing in a special trait and manually declaring the connections. Digital configuration pins for the ADC are manually added to the floating SCR file, as seen in Listing 6.6.

The output of the ADC is are 8 9-bit words, though the data rate requires a 1.2 GHz clock. To avoid such a high-speed signal processing path, the outputs are deserialized by a factor of 4 before being calibrated. The deserializer on the output of the ADC was written in Verilog to account for the intricate clocking. It is integrated as a black box in Chisel, just like the ADC, and it produces 32 9-bit words at 300 MHz.

Calibration is handled by a runtime-programmable SRAM memory bank, written in Chisel. It connects the deserialized ADC outputs to the address ports of 32 SRAMs, and outputs the read port into the signal processing chain. Initial TVALID and TLAST signals for the AXI4-Stream interface just come from registers in the floating SCR file. Special modes of the calibration SRAMs support loading of calibration data, storing ADC samples directly into the memory instead of using them as the address, and reading out of these SRAMs through the SCR file. To store calibrated ADC data samples directly for testing, a SAM must be connected after the calibration RAMs. This is done through a bit manipulator processing element that passes its input to its output without any modifications.

Listing 6.6: Some of the ADC and calibration integration code required

```
trait ADCTopLevelIO {
  val ADCBIAS     = Analog(1.W)
  val ADCINP      = Analog(1.W)
  val ADCINM      = Analog(1.W)
  val ADCCLKP     = Input(Bool())
  val ADCCLKM     = Input(Bool())
  val adcclkreset = Input(Bool())
}

trait LazyADC {
  def scrbuilder: SCRBuilder

  scrbuilder.addControl("OSP", 0.U)
  scrbuilder.addControl("OSM", 0.U)
  scrbuilder.addControl("ASCLKD", 0.U)
  scrbuilder.addControl("EXTSEL_CLK", 0.U)
  scrbuilder.addControl("VREF0", 0.U)
  scrbuilder.addControl("VREF1", 0.U)
  scrbuilder.addControl("VREF2", 0.U)
  scrbuilder.addControl("CLKGCAL", 0.U)
  scrbuilder.addControl("CLKGBIAS", 0.U)
  scrbuilder.addControl("ADC_VALID", 0.U)
  scrbuilder.addControl("ADC_SYNC", 0.U)
}

trait LazyCAL {
  def scrbuilder: SCRBuilder

  scrbuilder.addControl("MODE", 0.U)
  scrbuilder.addControl("ADDR", 0.U)
  scrbuilder.addControl("WEN", 0.U)
  (0 until 32).foreach { i =>
    scrbuilder.addControl(s"CALCOEFF$i", 0.U)
    scrbuilder.addStatus(s"CALOUT$i")
  }
}

trait ADCModule {
  def io: DspChainIO with DspChainADCIO
  def scrfile: SCRFile
  def clock: Clock // module's implicit clock
  def reset: Bool
```

```
  val adc = Module(new TISARADC)

  attach(io.ADCBIAS, adc.io.ADCBIAS)
  attach(io.ADCINP, adc.io.ADCINP)
  attach(io.ADCINM, adc.io.ADCINM)
  adc.io.ADCCLKM := io.ADCCLKM
  adc.io.ADCCLKP := io.ADCCLKP

  def wordToByteVec(u: UInt): Vec[UInt] = u.asTypeOf(Vec(8, UInt(8.W)))
  def wordToNibbleVec(u: UInt): Vec[UInt] = u.asTypeOf(Vec(16, UInt(4.W)))
  def wordToBoolVec(u: UInt): Vec[Bool] = u.asTypeOf(Vec(64, Bool()))

  val osp = wordToByteVec(scrfile.control("OSP"))
  val osm = wordToByteVec(scrfile.control("OSM"))
  val asclkd = wordToNibbleVec(scrfile.control("ASCLKD"))
  val extsel_clk = wordToBoolVec(scrfile.control("EXTSEL_CLK"))
  val vref0 = scrfile.control("VREF0")
  val vref1 = scrfile.control("VREF1")
  val vref2 = scrfile.control("VREF2")
  val clkgcal = wordToByteVec(scrfile.control("CLKGCAL"))
  val clkgbias = scrfile.control("CLKGBIAS")
...
```

6.3.2 Clock Receiver

A custom analog clock receiver takes a differential LVDS input and produces a single, sharp clock on chip. Like other IP, it is a black box in Chisel, and ports are punched through the hierarchy using traits and analog connection commands. Since Chisel has implicit clock ports everywhere, instantiating the clock receiver inside a module then connecting the clock produced to that module's registers and submodules is nontrivial. To simplify this, an extra hierarchical module is added above the usual top level, and the clock receiver output is looped around and connected to the implicit clock port, as seen in Figure 6.9. The new hierarchical top module's clock pin becomes the clock for the serial interface asynchronous FIFOs.

6.3.3 UART

As with other IP, the UART is included as a black box in Chisel, backed by a Verilog model for verification and synthesis. The UART is a third-party IP developed by Cadence and integrated as a slave coming off the MMIO manager. Internally the UART contains an AXI4 to APB converter, and the UART speaks APB directly. A TileLink to AXI4 and asynchronous FIFO connect the UART to the processor, and the UART IO get punched to the top level through mixins.

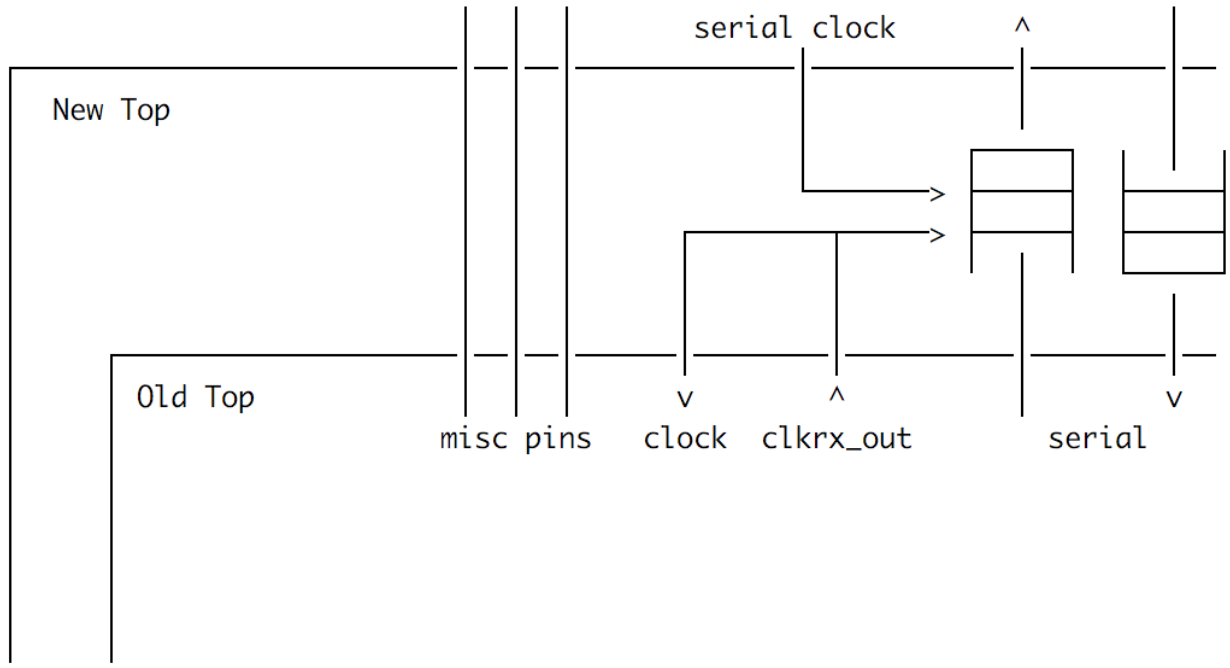


Figure 6.9: Clock looping to override default clock connections. The asynchronous FIFOs convert between the slow external clock domain and the fast on-chip clock domain.

6.3.4 Memories and IO

Mapping SRAMs from FIRRTL abstractions to physical memories is handled through a python script, which calls the memory compiler under the hood. The `vlsi_mem_gen` script contains a dictionary of all the possible memories. When called with desired memories (as a .conf file emitted by FIRRTL), the script maps these to real memories using simple heuristics. For example, it first tries to match the size perfectly. Large width and depth memories are split into banks of smaller memories, with the requisite logic to split the transactions. Memories that were generated previously are not re-generated, and new memories are generated in parallel, to save time. The memory Verilog models support initialization of the memory bits to known values to save time. The python script assumes you want random values, and so the generated `.cde` files which initialize the memories reflect this.

A python script, `create_pads.py` inserts a pad frame around the top-level Verilog module, creating a new top-level. Port names must be manually specified, along with their pad cell types, in the dictionary at the top. Pad-frame-specific cells are included and handled by the script. The JTAG TDO output signal is unique in requiring a tristate output, driven by two binary signals from the JTAG unit where one signal disables the output and the other determines the data value. Since the digital IO cells support switching between input mode and output mode using tristate inverters, this is how the JTAG TDO output is handled. As outputs, the script produces a pad frame Verilog file, a new top-level file that connects the

pad frame to the previous top-level module, a `.io` file for consumption by Innovus, and a single Verilog file containing the behavioral models for only those pad frame cells used.

6.4 Verification and Design-For-Test

Verification of the generators and instances was split into unit level tests and system level tests. The Rocket CPU comes with its own verification infrastructure, and this is considered at the system level. Design for test structures in the hardware support simple translation of design-time verification tests into hardware instance validation tests.

6.4.1 Unit Tests

At the unit level, two verification paradigms are adopted. The first involved writing Chisel tests for the processing element generators. Since the processing elements all contain the same interfaces (AXI4-Stream data inputs and outputs, and an AXI4 control interface), boilerplate functions simplify transaction definitions for verification. For example, SCR file registers are all 64-bit unsigned integers accessed through the AXI4 interface. But some processing elements, such as the FIR filter, have programmable coefficients residing in the SCR file but converted to an alternate datatype internally. Manually converting a number like -3.72 from a double to the arbitrarily defined datatype of the filter coefficients, then to a 64-bit unsigned integer, would be a hassle. Thus custom functions `pokeAs` and `peekAs` provide the capability of writing and reading values represented by some arbitrary datatype (such as a DspComplex with underlying FixedPoint) to and from interfaces using an unsigned integer datatype (such as the AXI4 data signals). Another function added is the ability to read and write memory-mapped SCR file registers using their name reference rather than their physical address. This allows testbenches to scale correctly when new registers are added to processing element SCR files. This was coupled with generalized AXI interface driver functions to create `axiWrite` and `axiRead`, and further combined with `pokeAs` and `peekAs` to create `axiWriteAs` and `axiReadAs`, as seen in Listing 6.7 for the FIR filter.

Listing 6.7: Example AXI interface function definition and usage

```
// write a value formatted as type T to an address through the AXI4 interface
def axiWriteAs[T<:Data](addr: BigInt, value: Double, typ: T): Unit = {

    // AXI write address and data channels
    poke(axi.aw.valid, 1)
    poke(axi.aw.bits.addr, addr)
    poke(axi.aw.bits.len, 0)
    poke(axi.aw.bits.size, log2Ceil(axiDataBytes))
    poke(axi.w.valid, 1)
    dspPokeAs(axi.w.bits.data, value, typ)
    poke(axi.w.bits.strb, 0xFF)
    poke(axi.w.bits.last, 1)

    var waited = 0
    var a_written = false
    var d_written = false

    while (!a_written || !d_written) {
      if (!a_written) { a_written = aw_ready }
      if (!d_written) { d_written = w_ready }
      require(waited < maxWait, "Timeout waiting for AXI AW or W to be ready")
      step(1)
      if (a_written) { poke(axi.aw.valid, 0) }
      if (d_written) { poke(axi.w.valid, 0); poke(axi.w.bits.last, 0) }
      waited += 1
    }

    // AXI write response channel
    waited = 0
    do {
      require(waited < maxWait, "Timeout waiting for AXI B to be valid")
      step(1)
      waited += 1
    } while (!b_ready);

    poke(axi.b.ready, 1)
    step(1)
    poke(axi.b.ready, 0)
}

// write all the FIR filter coefficients to its SCR file registers
coeffs.zipWithIndex.foreach { case(x, i) =>
    axiWriteAs(addrmap(s"Coefficient_$i"), x, fgk.genCoeff) }
```

To simplify verification of the streaming interface, AXI4-Stream inputs and outputs interface with software buffers. An input sequence is defined and subsequently *played* through the streaming input interface using the `playStream` function. To support AXI4 status and control interactions with the processing element without considering their latency overhead, the input stream can be *paused* with the `pauseStream` function, which invalidates input data and does not increment the read pointer of the input buffer. When output data are valid, they are added to the output buffer. Conversion between processing element IO datatypes and AXI4-Stream unsigned bits is automatically handled through software packing and unpacking, similar to `pokeAs` and `peekAs` but on an entire stream of data. These are the `packInputStream` and `unpackOutputStream` functions. Listing 6.8 gives examples of these functions for the tuner testbench.

Listing 6.8: Example definition and usage of some convenient processing element testing functions for the AXI4-Stream interface

```
// define input datasets here
val in = Seq.tabulate(test_length)(_=>
  Seq.tabulate(sync_period)(_=>
  Seq.tabulate(gk.lanesIn)(_=>
  Random.nextDouble*2-1 )))
def streamIn = in.map(packInputStream(_, gk.genIn))

// configure tuner
reset(5)
pauseStream
if (config.phaseGenerator == "Fixed") {
  axiWrite(addrmap("FixedTunerMultiplier"), 2)
} else {
  mult.zipWithIndex.foreach { case(x, i) =>
    axiWriteAs(addrmap(s"mult$i"), x, genMult.getOrElse(gk.genOut[T])) }
}

// play stream and get output
playStream
step(test_length*sync_period)
val output = unpackOutputStream(gk.genOut, gk.lanesOut)
```

Easy programming through the AXI4 control interface and communication with the AXI4-Stream data interfaces still requires useful tests and golden models to properly verify a processing element. Some processing elements, such as the filters and FFT, leverage a Scala numerical processing library called Breeze [79]. This obviates interfacing with MATLAB or other common signal processing algorithm design frameworks. Listing 6.9 shows how the FFT is verified by generating tones in each bin and comparing the results with a Breeze Fourier transform reference. The `compareOutputComplex` function takes a tolerance

value (third argument) since hardware is fixed point and the reference is floating point. And, though comparing against a reference is convenient, visualizing the data is sometimes required. Plot.ly is a free, web-based data plotting service with Scala integration libraries. This was used to visualize the PFB filter response, for example, as seen in Figure 6.10.

Listing 6.9: Example of using the Breeze fourierTr() function as a golden reference against which the hardware is compared

```
import breeze.math.{Complex}
import breeze.signal.{fourierTr}
import breeze.linalg._

...

// bin-by-bin testing
val m = 16 // at most 16 bins
(0 until min(fftSize, m)).foreach{ bin =>
  val b = if (fftSize > m) round(fftSize/m*bin) else bin // frequency
  val tester = setupTester(c, verbose)
  // generate and test tone in this frequency bin
  val tone = getTone(fftSize, b.toDouble/fftSize)
  val testResult = testSignal(tester, tone)
  // run Breeze to get reference output, then compare
  val expectedResult = fourierTr(DenseVector(tone.toArray)).toArray
  compareOutputComplex(testResult, expectedResult, 1e-15)
  teardownTester(tester)
}
```

Verification inside Chisel is convenient, but typically the testbenches are written by the PE designers. Separating the two helps prevent designers from simply creating designs that match their testbenches or testbenches that match their designs. Additionally, given their standardized interfaces, the processing elements are useful outside this chip or even this framework. Metadata languages, such as IP-Xact, allow outside designers and verifiers to understand what the PE is doing and easily write their own unit tests. For this chip, Chisel generates IP-Xact files for each PE, then NGC used the IP-Xact to help verify the design. NGC used the IP-Xact and Cadence Verification Workbench (VWB) to generate a UVM testbench, then used python scripts to read the IP-Xact parameters section and automatically generate appropriate test vectors. Details on this approach are omitted as they are outside the scope of my work, but suffice it to say their independent testing revealed design bugs not exposed by the Chisel unit tests. For example, the Chisel unit tests vary parameters to ensure functionality is maintained across designs. However, to decouple functionality issues from numerical error, bitwidths and datasets are chosen so as not to saturate operations or test noise limits. The NGC tests took the chosen parameterization and exercised these test cases, discovering bugs hidden in the design that would limit performance.

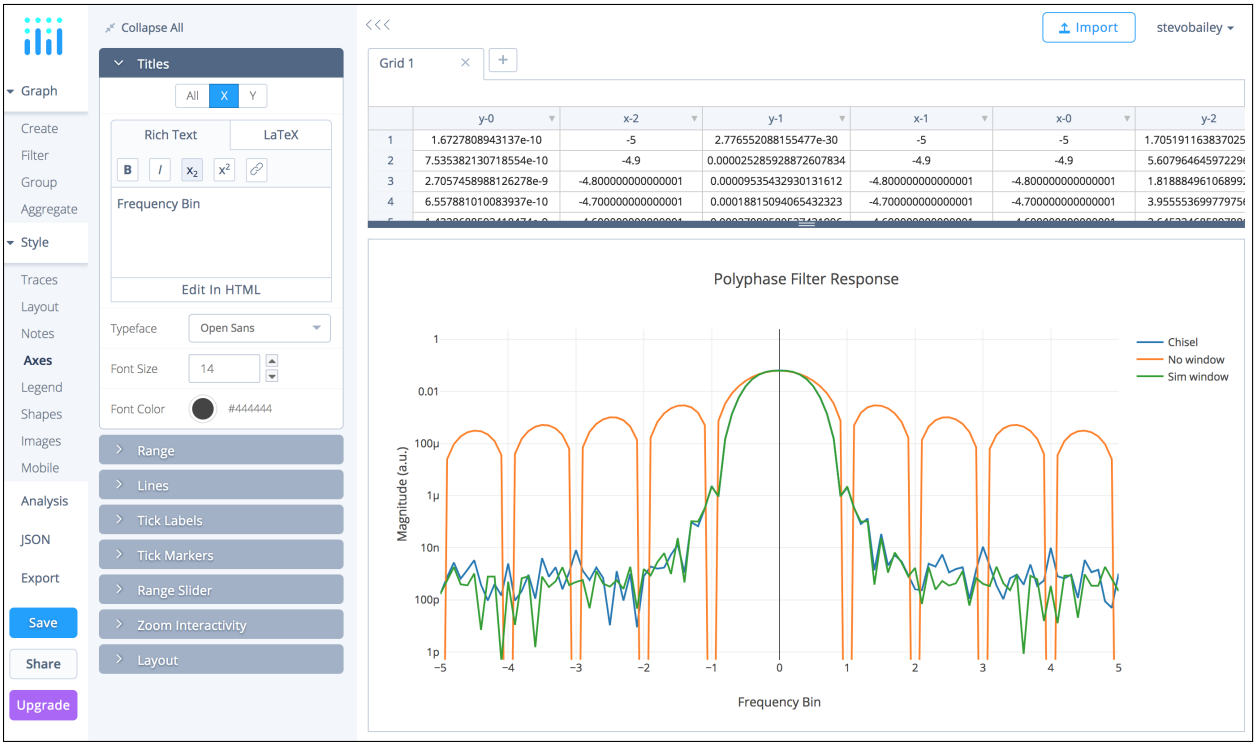


Figure 6.10: Visualizing the PFB filter response in Plot.ly.

6.4.2 System Tests

System-level testing of the general-purpose processor and signal analysis accelerator relied on running compiled C programs. The RISC-V compiler for Rocket systems lives in the `rocket-chip` repository as a submodule. Building the toolchain produces cross compilers capable of compiling custom C code for both Verilog simulation and hardware testing. To simplify testing, Chisel produces a C header file which maps the names of all the SCR file registers to their addresses in `define` statements. This header file includes other helper functions, like functions for reading and writing these memory-mapped registers and SAM functions. Some of these features are shown in Listing 6.10, which contains a snippet of the generated chain API header file.

Listing 6.10: Generated C header snippet showing the mapping between registers and their addresses and functions to access these registers

```
#define craft_radar_bm2_Wrapback 0x6800L
#define craft_radar_bm2_uuid 0x6808L

#define craft_radar_tuner_Wrapback 0x7000L
#define craft_radar_tuner_Data_Set_End_Clear 0x7008L
#define craft_radar_tuner_FixedTunerMultiplier 0x7010L
#define craft_radar_tuner_Data_Set_End_Status 0x7020L
#define craft_radar_tuner_uuid 0x7018L

#define craft_radar_fir_Coefficient_46 0x7980L
#define craft_radar_fir_Coefficient_52 0x79b0L
#define craft_radar_fir_Wrapback 0x7800L
#define craft_radar_fir_Coefficient_67 0x7a28L

...

static inline void write_reg(unsigned long addr, unsigned long data) {
    volatile unsigned long *ptr = (volatile unsigned long *) addr;
    *ptr = data;
}

static inline unsigned long read_reg(unsigned long addr) {
    volatile unsigned long *ptr = (volatile unsigned long *) addr;
    return *ptr;
}
```

Running programs requires first compiling the Verilog design and testbench in a simulator. The simulation side of the serial interface, written in C, connects the hardware to testbench software that writes programs into the chip. This enables any compiled program to be run in simulation. Design and testbench modifications allow integrated IP to be simulated in conjunction with the processor. First, all IP requires a Verilog or SystemVerilog model to simulate properly. The ADC has a custom SystemVerilog model, the clock receiver has a simple differential-to-single-ended Verilog model, the UART design is Verilog already, the memory compiler generates Verilog models for simulation, and the IO cells have their own Verilog models as well. Second, certain IP, like the ADC, use real-valued analog pins. To support SystemVerilog simulation with real-valued signals, a custom Chisel annotation was added to add the `real` keyword to these analog pins. A noisy sine wave was driven into the top-level ADC inputs to simulate a real signal, as seen in Figure 6.11.

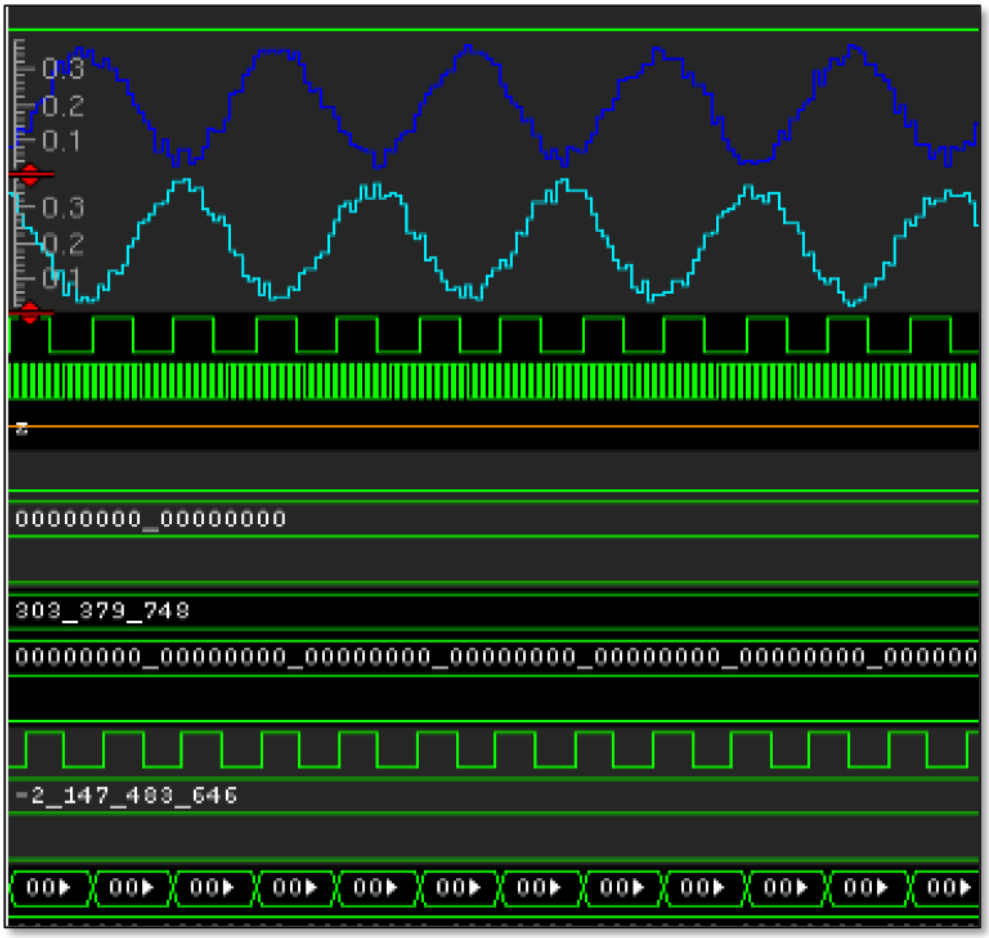
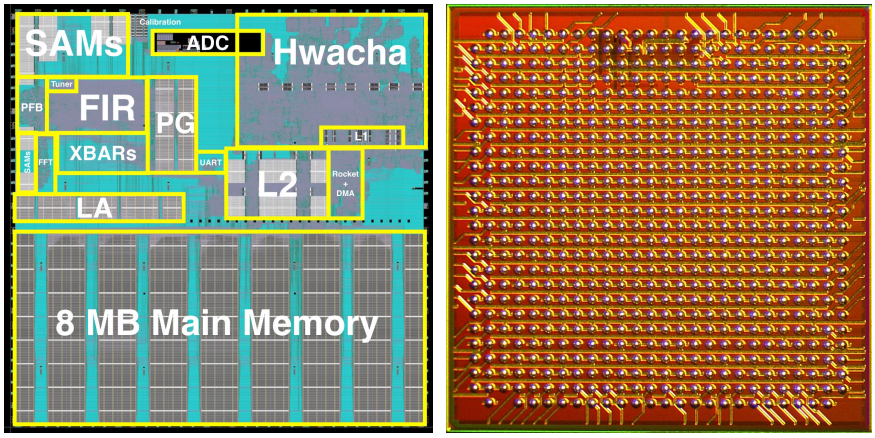


Figure 6.11: System simulations included a noisy sine wave input to the ADC to check for end-to-end functionality.

6.4.3 DfT Considerations

Testing the hardware is aided by coupling the design with design-for-test (DfT) structures. A JTAG debug module provides access to the signal processing accelerator as a backup to the CPU. A pattern generator and logic analyzer connect to each PE's IO and perform the same unit-level verification on the chip after fabrication. The python testvector generators also generate C code, playing inputs through the pattern generator and recording outputs in the logic analyzer. These programs are then compiled and run on the chip after fabrication. System-level tests are already compiled C programs that can run on the hardware as easily as the simulator. Thus no additionally system-level DfT structures are required.



Annotated Layout Die Photo

Technology	16nm FinFET	
Die Area	5mm x 5mm (25mm^2)	
	General-Purpose Processor	**Signal-Analysis Processor**
Area	1.1 mm^2 (gates) 9.2 mm^2 (SRAM)	1.5 mm^2 (gates) 0.8mm^2 (SRAM)
Total SRAM Size	71 Mbits	14 Mbits
Voltage	0.56 V - 0.98 V	0.56 V - 0.98 V
Max Frequency	410 MHz	417 MHz
Power	349 mW @ 0.75V, 410 MHz	210 mW @ 0.75 V, 417 MHz
Max Throughput (Mspectra/s)	0.46 (vector) 0.004 (scalar) @ 410 MHz	13 @ 417 MHz
Efficiency	23.4 GFLOPS/W (0.56V, 191MHz) (DGEMM on vector accelerator)	19.2 TOPS/W (0.56V, 192MHz) (1 op = 8-bit add ~ 17-bit mul)

Figure 6.12: Chip layout, die photo, and summary.

6.5 Testing Results and Measurements

The chip is implemented in TSMC's 16nm FinFET technology and signed off at 300 MHz for both the core and DSP clock domains at 0.72 V and 125°C. Figure 6.12 shows the 5 mm by 5 mm annotated layout, die photo, and chip summary. The 8 MB main memory, Hwacha vector accelerator, and various other memories comprise most of the area. Also visible is the 136-tap fully programmable FIR filter, composed of many complex multipliers and adders. By using the Hwacha vector accelerator, at these conditions the general-purpose processor achieves 23.4 GFLOPS/W running double-precision matrix multiply on 256×256 matrices. For the general-purpose processor, throughput is measured by moving FFT output data from the SAM to the CPU memory and accumulating spectra using either the vector co-processor or the scalar ALU and DMA. Throughput is measured for the signal-analysis processor at the maximum operating frequency (see Figure 6.16 for more on spectral rates). Efficiency for the signal-analysis processor accounts for all PEs, and one operation is anything from a real 8-bit add to a 17-bit multiply, with complex adds and multiplies broken into their real operations. The signal-analysis processor achieves 19.2 TOPS/W, with a breakdown of

Table 6.1: Breakdown of operations in the signal-analysis processor. Each operation is performed once per cycle. C represents a complex number, R represents a real number, and square braces show the number of bits for that operand. I assume C×R is 2 multiplies, C×C is 3 multiplies and 5 adds, and C+C is 2 adds.

Block	Operation	Bitwidth	Count	Description
Tuner	C×R	8	32	Input times value in ROM
Filter	C×C	8	32×136 / 8	Input times 136 coefficients
Filter	C+C	11	32×135 / 8	Adder trees
PFB	C×C	12	4×12	Input times 12 coefficients
PFB	C+C	12	4×11	Adder trees
FFT (Biplex)	C×C	17	10	Butterfly multiplier
FFT (Biplex)	C+C	17	20	Butterfly adders
FFT (Direct)	C×C	16	4	Butterfly multiplier
FFT (Direct)	C+C	16	8	Butterfly adders

operations shown in Table 6.1. It is assumed that decimation by eight in the filter reduces the number of operations by eight, since decimation is performed by disconnecting outputs and letting synthesis optimize away unnecessary operations. Also, the direct FFT operations are 19 bits but the output is truncated to 16 bits, so this is counted as a 16 bit operation.

Figure 6.13 shows the shmoo and power plots for two processors. A shmoo plot indicates under which operating conditions, typically frequency and voltage, the processor functions correctly. Both function down to 0.56 V (the nominal supply is 0.8 V) and up to 410 MHz. A success on the shmoo plot requires the general-purpose processor to pass all ISA unit tests and the signal-analysis accelerator to pass all PE unit tests. Annotated on the shmoo plots are the corner values, i.e. the minimum voltage at the maximum frequency and the maximum frequency at the minimum supply voltage. The chip consumes less than 1 W total for all modes.

Figure 6.14 shows the power and a typical calibration result for the 8 slice time-interleaved SAR ADC, which features a fractional radix to produce 8 real output bits from 9 total bits. The ADC reaches a max of 6.6 ENOB per slice and 6.4 ENOB total at 6 GS/s under a 0.9V supply. Analog supplies are independent from digital supplies.

6.6 Signal Analysis Applications

In this section we present two applications running on the SoC. Using the pattern generator to produce arbitrary waveforms and replace the ADC as the input source hastened the development and debugging of writing these programs by decoupling program errors from ADC miscalibration, input noise, or instrument errors. The applications utilize many SoC features, including the complete signal processing chain, DMA, vector accelerator, and general-purpose processor. We compare the results with similar, fixed-function processors.

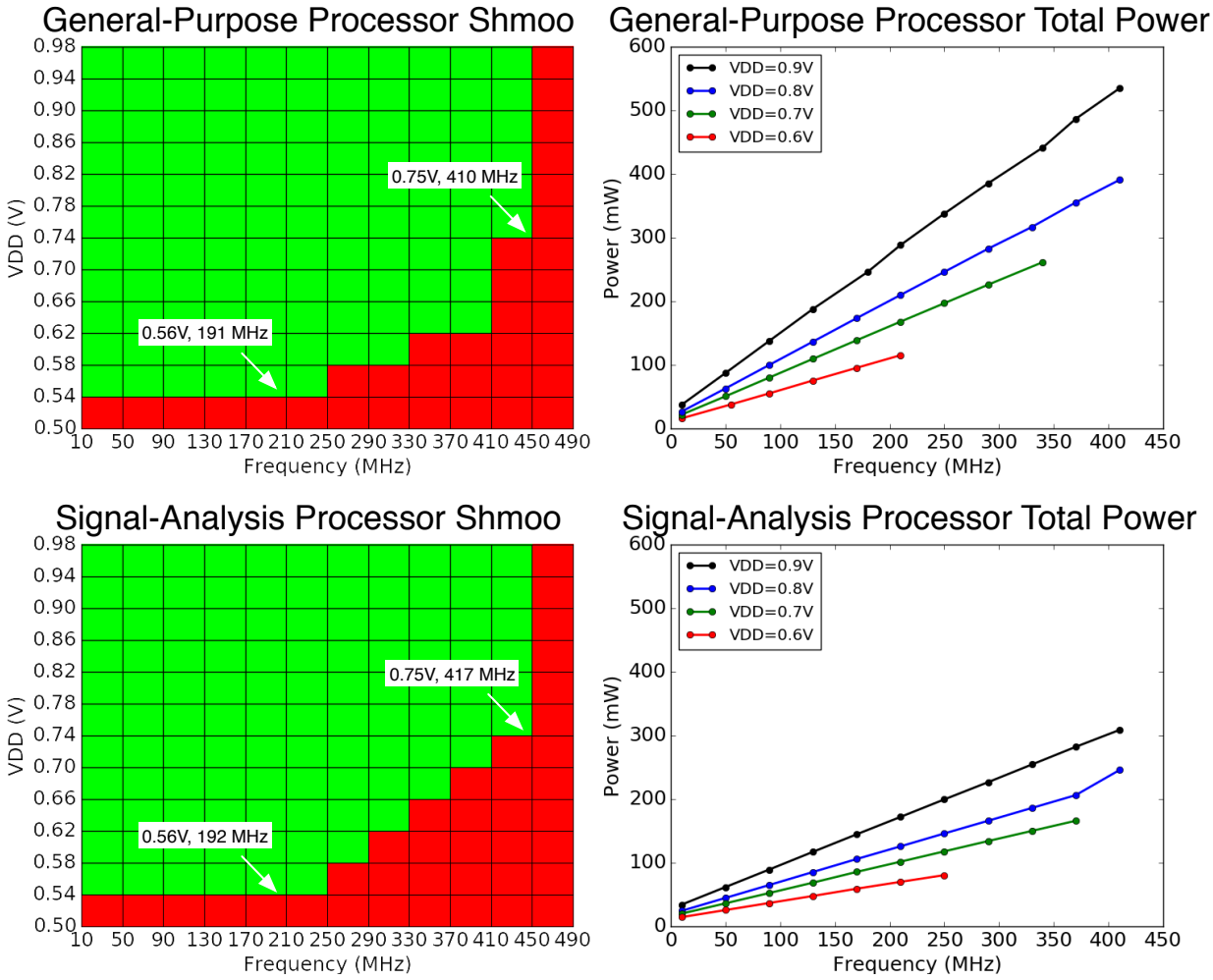


Figure 6.13: Both processors function under similar operating condition ranges. The general-purpose processor consumes more power because of the 8 MB main memory.

6.6.1 Spectrometry

Atmospheric spectrometers monitor molecule emissions to determine the composition of gases. Given the low SNR of these emissions, spectrometers with a wide bandwidth and long accumulation time are desired. A 512-pt FFT is formed by sweeping the tuner frequency to allow the signal processor to analyze up to four frequency bands with a 128-pt FFT engine (the filter decimates the data rate by eight, but half the bands are symmetric because the input is real-valued). Figure 6.15 shows the signal processing path of a spectrometer input dataset. The low-pass filter, with constrained equiripple coefficients designed in MATLAB, passes the lower eighth of the spectrum to avoid aliasing when down converting. Its stopband is at 40 dB below the passband, resulting in visible aliasing above the noise floor in the combined spectrum. For each band, the tuner is set and the FFT outputs are stored in

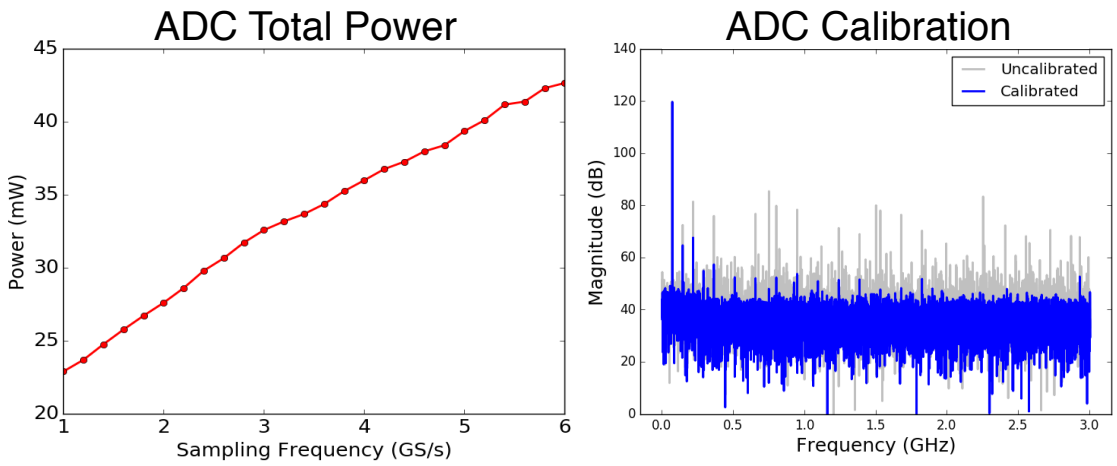


Figure 6.14: Typical ADC power consumption is less than 50 mW at 0.9V. Calibration of the ADC reduces noise and spurs.

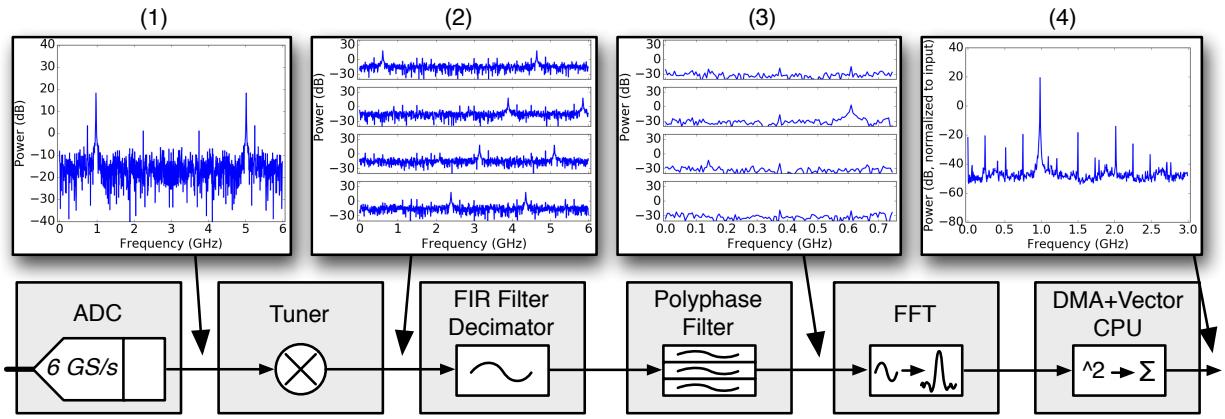


Figure 6.15: Spectrometer signal processing example. Snapshots captured in the SAMs. (1) A real-valued signal is sampled through the calibrated ADC, producing a symmetric spectrum. (2) Four tuner LO frequencies are mixed with the input, producing four frequency-shifted spectra. (3) These spectra are low-pass filtered and down-converted by 8, resulting in four separate frequency bands. (4) The bands are Fourier transformed, accumulated, and combined on the CPU. This figure shows 100 accumulated spectra.

the SAM before being moved into the general-purpose processor's memory by the DMA. The power is calculated and accumulated using the vector accelerator. The use of these accelerators boosts the data processing rate for this application by over 10x, as seen in Figure 6.16. While not designed specifically for spectrometry, this work is competitive with other published ASIC spectrometers, as seen in Table 6.2.

Table 6.2: Comparison of state-of-the-art ASIC spectrometers

	CICC'09 [38]	**CICC'15** [14]	**CICC'18** [80]	**This Work**
Technology	90nm CMOS	65nm CMOS	28nm FDSOI	16nm FinFET
Bandwidth	0.75 GHz	1.1 GHz	**8.5 GHz**	3.0 GHz
FFT Size	**8192 pts**	512 pts	**8192 pts**	128 - 512 pts
Integrated ADC	No	**Yes**	No	**Yes**
Power	1500 mW*	**188 mW**	5200 mW	586 mW
ADC Output	**8 bits**	7 bits	3 bits	**8 bits**
Can post-process	No	No	No	**Yes**
On-chip Accum. Depth	16M Spectra	1024 Spectra	65520 Spectra	**Infinite**

*excludes ADC power

6.6.2 Radar

Unlike atmospheric spectrometry, radar operates on fixed or variable short pulses or frequency-modulated continuous wave (FMCW) signals. These higher SNR signals require less accumulation, but processing speed limits the detectable range resolution. Figure 6.17 shows an example measured spectrogram of 4 µs fixed-frequency pulses, repeating every 8 µs. The

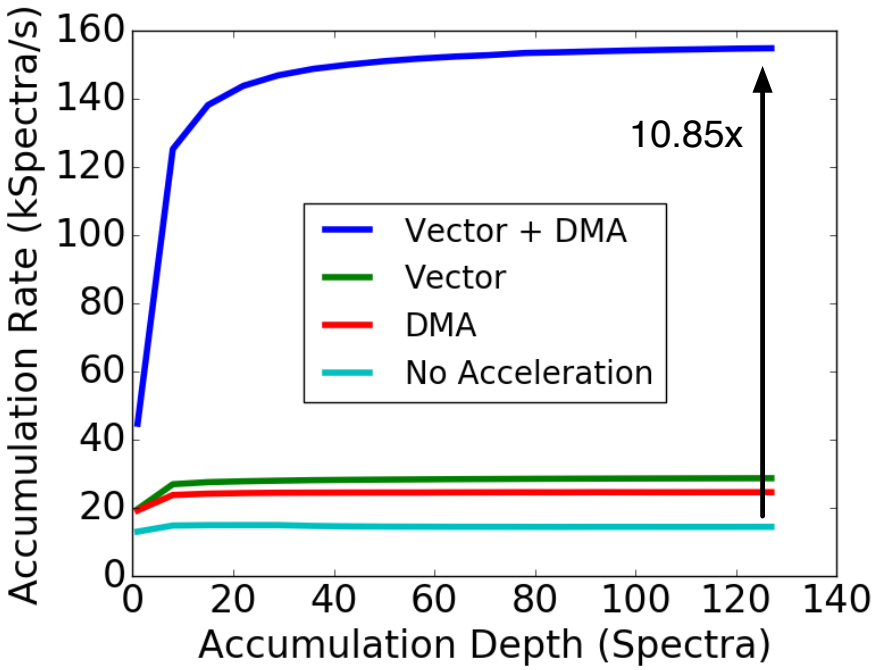


Figure 6.16: Using the vector and DMA accelerators speeds up spectral accumulation by over 10x. This plot includes the overhead of sweeping the tuner frequency to monitor four frequency bands, so each spectrum is 512 channels.

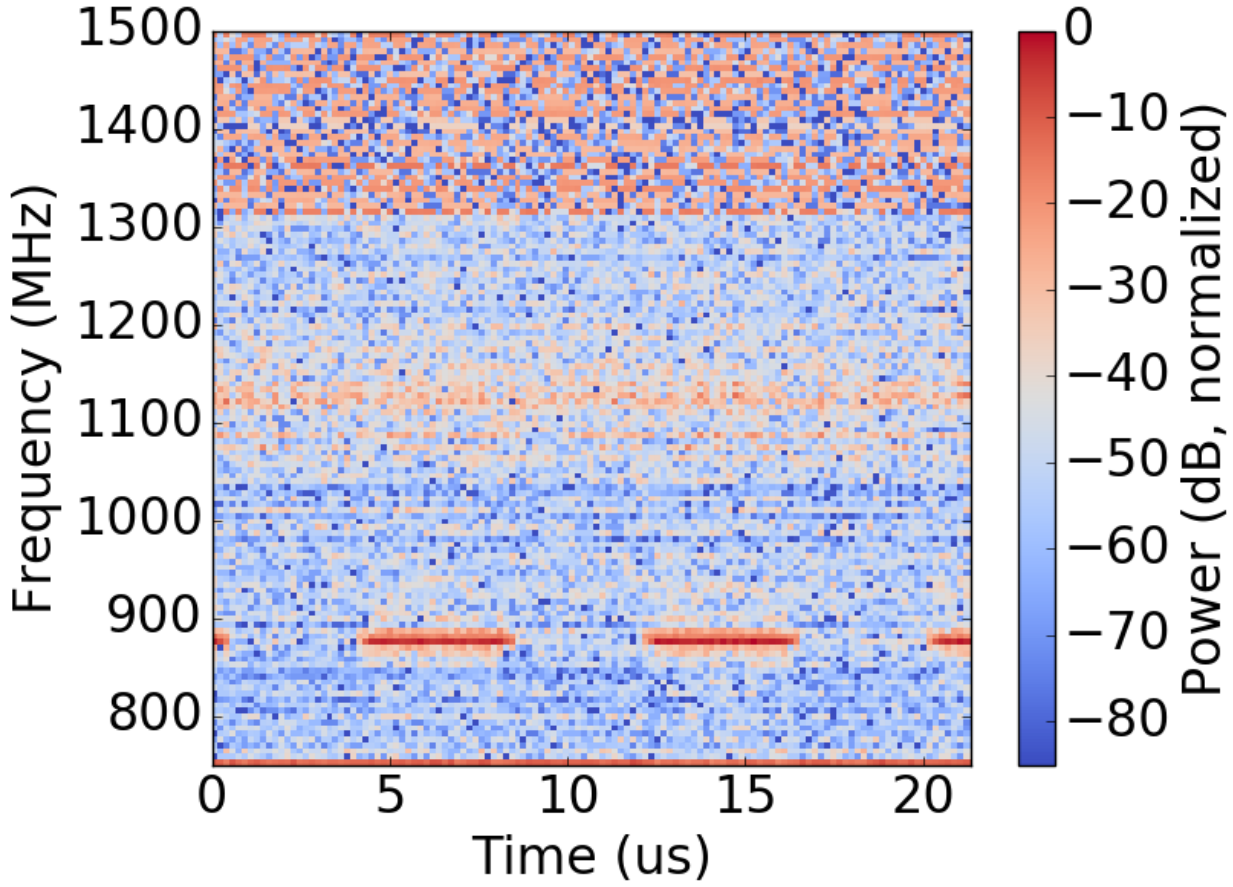


Figure 6.17: Measured spectrogram of a 4 us pulse at 876 MHz.

tuner is set to view frequency bands containing the expected signal, and the FFT outputs a spectrum every 171 ns when the ADC operates at 6 GS/s. For unmodulated pulsed signals as in Figure 6.17, the minimum pulse repetition frequency (PRF) this SoC can resolve is 2.9 MHz (pulses per second), leading to a minimum range resolution of

$$\frac{c}{2 \times PRF} = 51.7\,\text{m}. \tag{6.1}$$

It is possible to increase resolution by implementing pulse compression. The vector accelerator has sufficient throughput to convolve the received signal with the expected signal, and the FFT may be reused to perform an IFFT to recover the compressed signal. At 6 GS/s and by using a single frequency band with a 750 MHz wide linear frequency-modulated (LFM) chirp, this system has a minimum range resolution of 0.2 m.

6.7 Agile Design Process

This section explores agile prototyping of the radar receive chain by building it up piece-by-piece and how to respond to changing specifications.

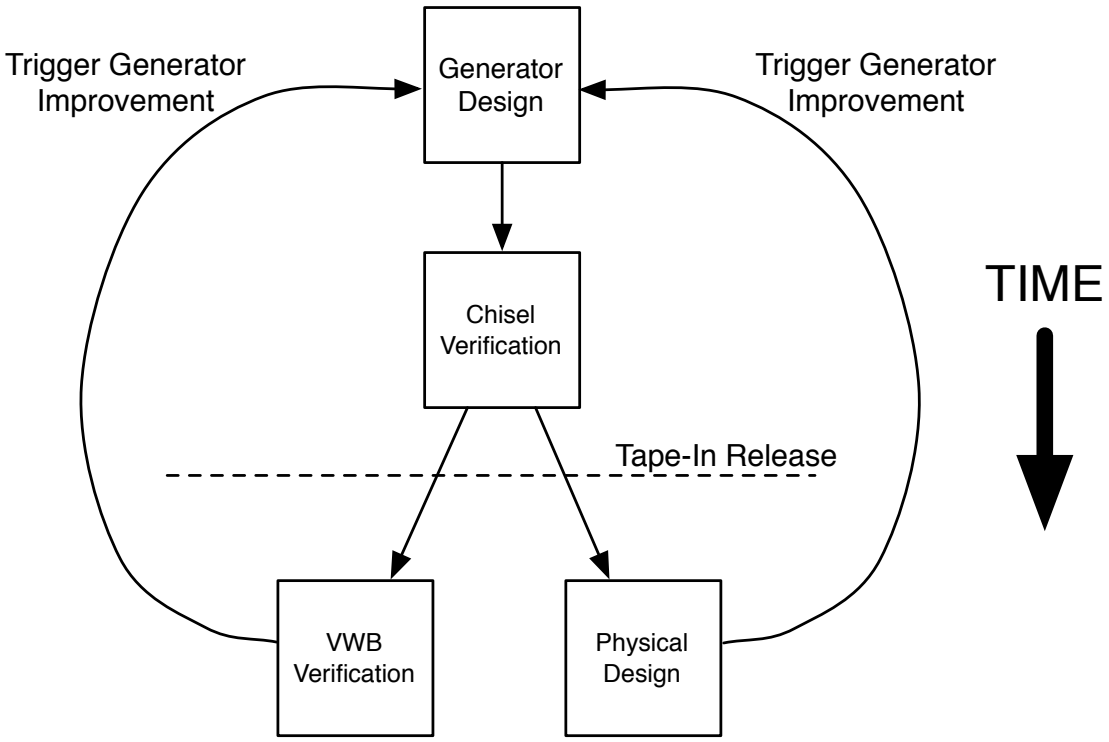
Previous agile chip design flows considered a tape-in product to be a GDS (a Graphic Database System, which is the final submission that a foundry uses to create the chip).

> "We first push a trivial prototype with a minimal working feature set all the way through the toolflow to a point where it could be taped out for fabrication.... The agile hardware methodology will always have an available tape-out candidate with some subset of features for any tape-out deadline, whereas the conventional waterfall approach is in danger of grossly overshooting any tape-out deadline because of issues at any stage of the process."
>
> (Lee 2016 [33])

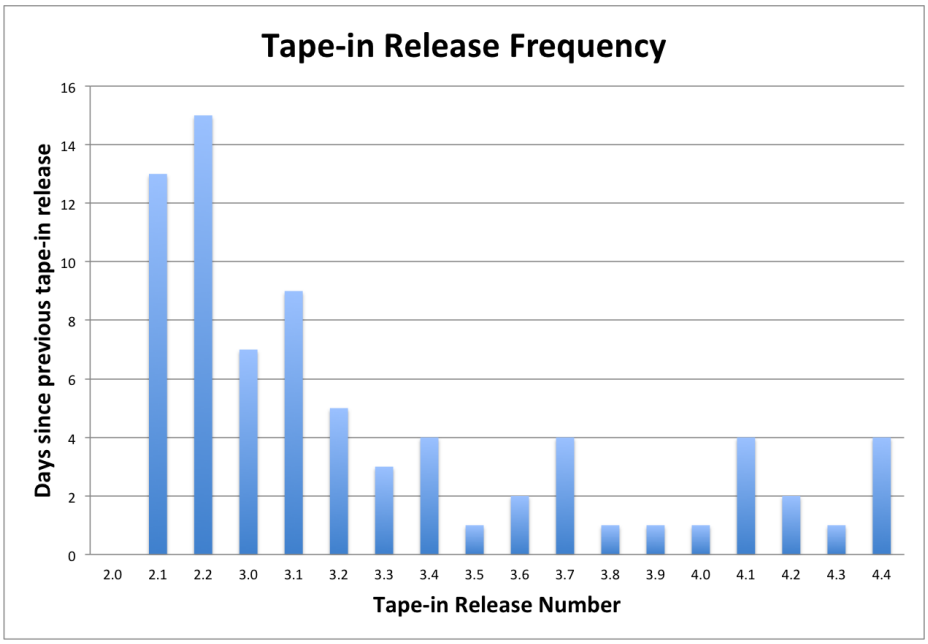
For this chip, the RTL design and physical design teams were separated, so we redefined what a tape-in meant for the RTL design team. At Berkeley, the Chisel design passed all Chisel unit tests before NGC began their verification and physical design. When new features were added, such as more processing elements, IP, or GPP accelerators, the design was again verified then pushed through the physical design flow. Thus for us, a tape-in represented an RTL release that was ready for physical design and tape-out. Figure 6.18a visually shows this tape-in boundary. Any feedback from NGC while they verified on their side or ran physical design triggered an improvement in the generator and a new tape-in release. And as the handoff and feedback procedure matured, so too did the frequency of tapein releases, as seen in Figure 6.18b. This convergence suggests a coupling of process flows is possible, allowing for a near continuous evolution of design improvements, verification improvements, and physical design improvements, all living in the same generator framework, as seen in Figure 1.2.

Initial specifications included a 10 GHz ADC with deserialized output data feeding into the DSP chain. The chain consists of multiple programmable tuners, filters, and decimators followed by a polyphase filter and FFT. Spectral bins are tagged when surpassing a threshold in the receive signal strength indicator (RSSI). Results are buffered in a SAM for readout. Figure 6.19 shows more complete initial specifications, omitting control interfaces and the rest of the system for simplicity.

Already obvious from the specification is the need for tuner and filter generators, since each copy will have different data types, parallelization, and coefficients. The input tuner accepts real data but outputs complex data, and later tuners all operate on complex data. Bitwidth growth is desired during each filtering PE and the FFT. So instead of designing unique tuners, we designed one parameterized tuner generator which works in any context. Also note uncertainty in the specifications. Some details did not solidify until well into the design process. This required us to be flexible in our implementation. The next paragraphs highlight the process of designing and verifying an example PE and present a discussion into the flexible system construction methodology.



(a) Tape-ins represent verified RTL releases ready for physical design.



(b) The frequency of tape-in releases increased as the handoff procedure matured.

Figure 6.18: Tape-in metrics.

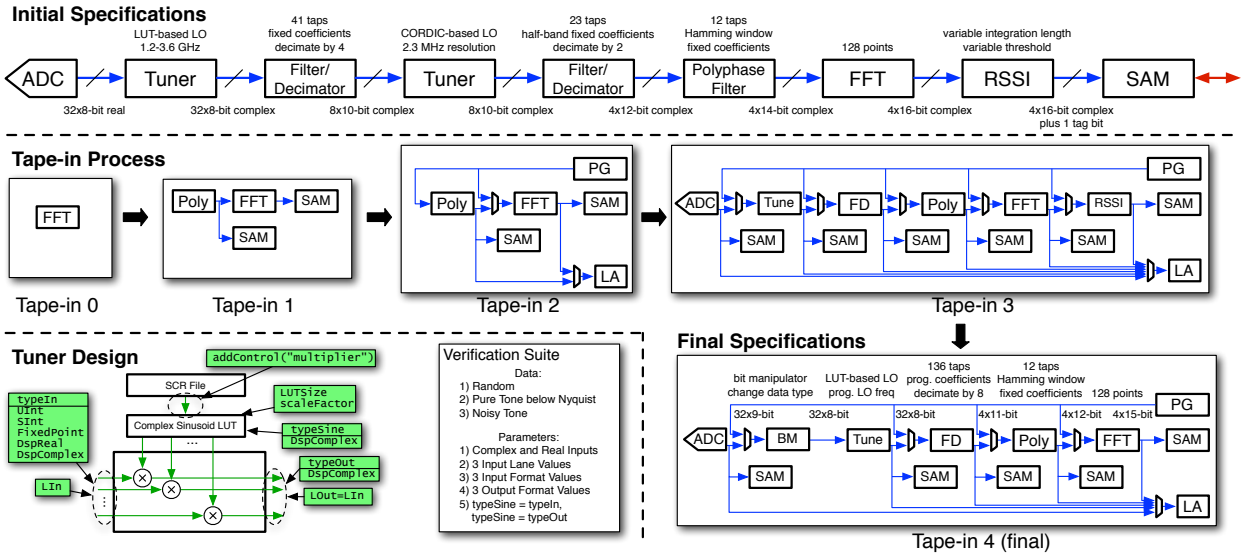


Figure 6.19: Radar processor design flow. Top: Initial customer specifications. Middle: Agile design evolution, Bottom-Left: Example tuner design showing parameterization in green and the verification suite.

Designing a tuner generator, along with its verification generator, required collaboration between team members. First the tuner parameter set was determined. It needed to accept real or complex inputs of any bitwidth, produce complex outputs of any bitwidth, and contain either a programmable LUT or CORDIC from which to get the local oscillator signal. Next a verification suite was outlined and implemented as part of a test-driven development design style. Randomly generated test vectors and parameter values ensure the tuner operates across a wide usage range. Finally, the tuner was built to meet these specifications and pass these tests. However, to demonstrate functionality in a limited time, only the LUT-based tuner was implemented. Continuous integration on Github meant any design change had to pass the test suite before being accepted.

The tape-in process, seen in Figure 6.19, shows how the design evolved over time and drifted away from the initial specifications. To simplify verification and the design itself, the tuner and filter stages were combined into one set. Changing the filter design to 136 taps with programmable coefficients resulted in a massive physical design, yet implementation was feasible. The originally-planned RSSI block required too many SRAM instances to fit in the prescribed floorplan, so it was omitted in the final design. Deleting only three lines of code resulted in a new DSP chain without the RSSI block, which was implemented in software instead. Memory-mapped addresses were recalculated, crossbars adjusted their size, and connections to the PG and LA were deleted, all automatically. Extra SAM blocks were added as a back-up measure, and a new PE, a bit manipulator, was added to convert the ADC output data type into the tuner's input data type. The result was a verified and validated radar system that, despite being substantially different from the initial design, met

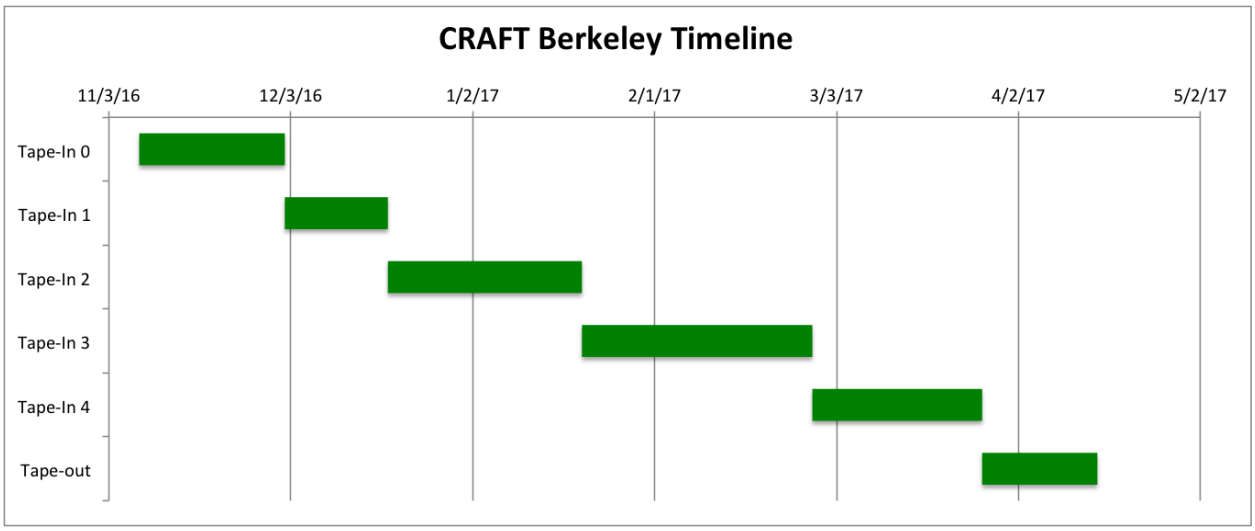


Figure 6.20: Early proposed tape-in schedule.

the requirements.

In all, the timeline for this project spanned roughly 9 months from inception to tape-out. Initial attempts at a Gantt chart (Figure 6.20) to schedule the sequences and contents of tape-ins (Figure 6.21) quickly broke down. Unforeseen programming and design hurdles required modifications to the schedule to still tape out in time. Given a hard deadline, the easiest way to deal with schedule hiccups is to squeeze less time into later steps, leaving more time for current steps. But this is unsustainable. The rigid plan approach was quickly abandoned for a flexible approach, where highest priority features were implemented first. In all, the tape-out occurred on time, taking a total of about 12000 engineering hours spread across three companies. That's equivalent to 300 weeks, or 9 months of 8 engineers working full time. This represents a 38% improvement over the DARPA estimated timeline for a chip to be taped out (14.4 months for a 10 person team) [1].

6.8 Conclusion

This chapter demonstrates an ASIC, designed by using parameterized digital and analog generators, that achieves a peak efficiency of over 19 TOPS/W and 23 GFLOPS/W in 16 nm CMOS. The implemented RISC-V signal analysis SoC, generated from Chisel and BAG frameworks, performs spectrometry and radar signal processing with performance comparable to the state of the art. On-chip DfT facilitates quick bring-up and validation of the design instance. Generators used in this work are open source and may be easily adjusted and reused for a variety of applications [78]. The process of designing the chip highlights how proper application of agile hardware development principles contributes to the success of a project, in this case a streaming DSP processor.

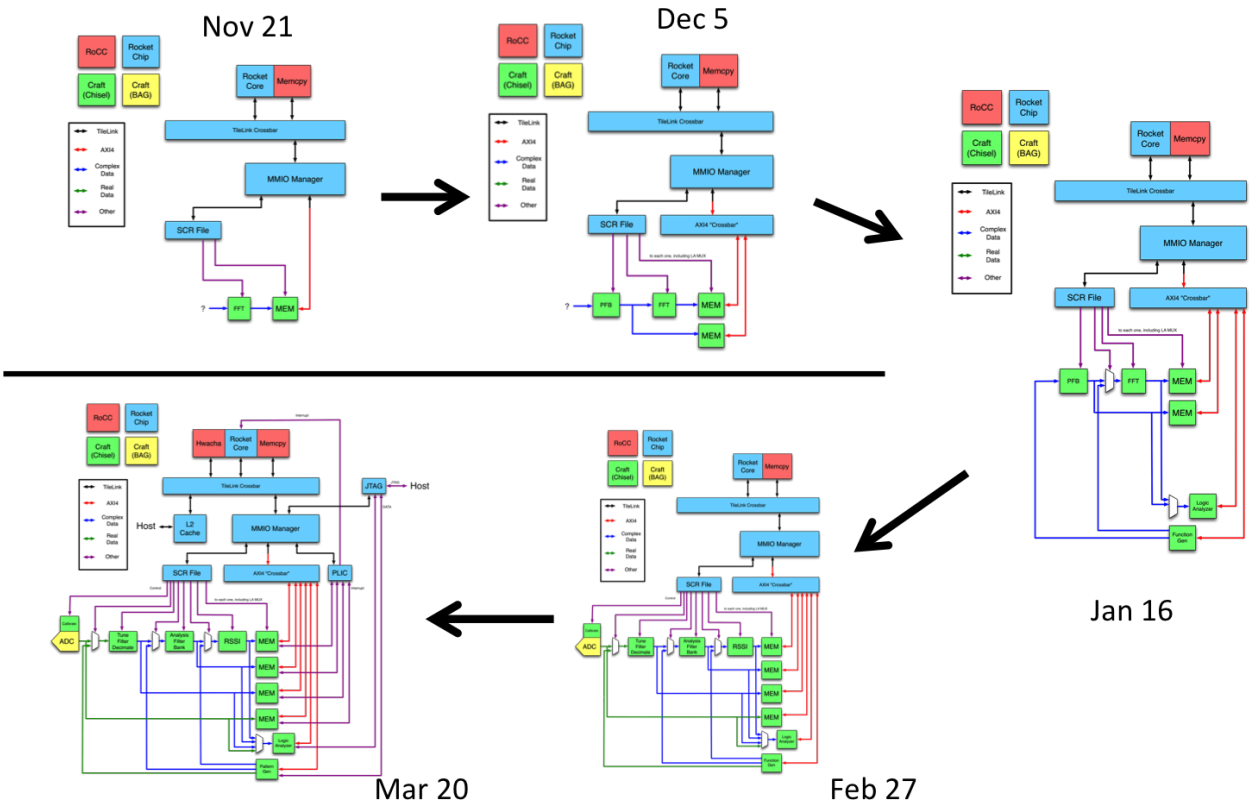


Figure 6.21: The incremental addition of features in the framework and design defined a tape-in schedule, which quickly broke down.

Chapter 7

Conclusion

This work presents a digital signal processing SoC framework that, when coupled with agile design principles, supports rapid prototyping and designing of ASICs for signal processing applications. The methodology and proposed agile design process were validated on two separate chips, spanning two applications and two process nodes. A 56% reduction in development time is achieved when using the generator framework over similar implementations designed from scratch.

7.1 Summary of Contributions

Strict performance, power, and form factor constraints require the use of ASICs, but their complexity makes them slow to design. The development time is split between logical design, verification, validation, and physical design. Agile hardware design, stemming from a similar, successful software design technique, has seen growing adoption in the development of CPUs. The biggest contribution of this thesis is an exploration and evaluation of these agile hardware design principles to signal processing ASIC development.

Specifically, this work makes the following contributions. Chapter 4 presents an extensible generator framework that supports custom parameterization and design of processing elements, customizable sequencing of PEs for target applications, and a verification framework. A set of processing element generators, including a polyphase filter generator, a scalable fast Fourier transform generator, a vector accumulator generator, a finite impulse response filter generator, and a programmable digital mixer generator are made open source and presented in Chapters 5 and 6. Validation of the generator framework is done through the design, fabrication, and testing of two signal processing ASICs in Chapters 5 and 6. Chapter 5 describes the digital ASIC spectrometer design and chip with high bandwidth and low power for NASA, targeting atmospheric measurements from satellites or weather balloons. Chapter 6 describes the signal analysis SoC design and chip with an integrated ADC for Northrop Grumman Corporation, targeting radar receive analysis. Finally, these last two chapters highlight the application of agile hardware methodology principles which, when coupled with the generator

framework, reduces the effort and design time of signal processing SoCs.

7.2 Future Work

During the course of this research many first-of-a-kind attempts were made. This was the first design where agile generator-based design was metricized. Still, many tradeoffs were made, and there are opportunities for further research.

- The complexity of the framework restricts its widespread adoption and usage. Chisel and Scala obfuscate certain levels of parameter passing and port punching that make tracing the source of errors difficult. Simplifying the code may reduce functionality and efficacy, but if it permits use, then the effort is worthwhile.
- While the verification framework provides sufficient coverage for many applications, tighter integration between standardized platforms (like UVM) and the standardized interfaces (like AXI) could be done in Chisel itself, rather than relying on third-party tools. Also, an analysis of coverage would clarify how much remains.
- A second significant reduction in design time could be realized through coupling RTL design generators with backend synthesis and place-and-route flows. For example, embedding timing and placement constraints in the generator would allow for generator-based ASIC tool flows that scale automatically with the design. This effort is already underway at Berkeley with the Hammer project.
- Methodologies are inherently hard to metricize, given that they embody a diverse set of constraints including abstract and subjective ones. Nonetheless, a more rigid approach to characterizing the benefits and pitfalls of applying agile design principles to hardware is desirable.

Signal processing spans nearly every domain of science and engineering, and the need for low-power, high-throughput processing systems will never disappear. Continued work in reducing the effort it takes for new algorithms to be realized in ASIC platforms will help streamline certain aspects of scientific progress.

Bibliography

[1] L. Salmon, "A perspective on facilitated access to custom ic design in leading-edge cmos technology," in *DAC workshop 6: design automation for HPC, clouds, and server-class SoCs*, 2015.

[2] W. R. Davis, "Digital signal processing architecture: Esl design methodologies." `http://www4.ncsu.edu/~wdavis/doc/ece747spr06_2_1up.pdf`, 2006.

[3] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, pp. 54–62, Aug 2003.

[4] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based Implementation of Signal Processing Systems*. Wiley Publishing, 2nd ed., 2017.

[5] R. Rohde, "Solar radiation spectrum." `https://commons.wikimedia.org/wiki/File:Solar_Spectrum.png`, 2007.

[6] N. Livesey, P. Stek, G. Chattopadhyay, R. Jarnot, J. Kocz, R. Stachnik, W. Deal, and D. Werthimer, "The compact adaptable microwave limb sounder (camls), an esto instrument incubator program (iip) 2013 project iip-13-0046," October 2014. Presentation.

[7] R. Finger, P. Mena, N. Reyes, R. Rodriguez, and L. Bronfman, "A calibrated digital sideband separating spectrometer for radio astronomy applications," *PASP 2013*, vol. 125, no. 925, pp. 263–269, 2013.

[8] J. Lesurf, "Ranging radar: Pulsed radar." `https://www.st-andrews.ac.uk/~www_pa/Scots_Guide/RadCom/part15/page1.html`.

[9] L. Pang, X. Li, and Y. d. Luo, "A structured asic based implementation platform for digital radar receiver," in *2014 12th International Conference on Signal Processing (ICSP)*, pp. 2003–2006, Oct 2014.

[10] Texas Instruments, "Tms320c5545 fixed-point digital signal processor." `http://www.ti.com/lit/ds/symlink/tms320c5545.pdf`.

[11] Ceva, "Ceva-teaklite-4 a multifunctional dsp architecture for high-performance, low-power audio/voice/sensing and wireless communication applications." `https://www.ceva-dsp.com/wp-content/uploads/2017/02/CEVA-TeakLite-4-DSP-Family.pdf`.

[12] B. D. de Dinechin, "Kalray mppa: Massively parallel processor array: Revisiting dsp acceleration with the kalray mppa manycore processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, pp. 1–27, Aug 2015.

[13] Y. Gu, H. Jiang, X. Xie, G. Li, and Z. Wang, "An image compression algorithm for wireless endoscopy and its asic implementation," in *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 103–106, Oct 2016.

[14] F. Hsiao, A. Tang, Y. Kim, B. Drouin, G. Chattopadhyay, and M. C. F. Chang, "A 2.2GS/s 188mW spectrometer processor in 65nm CMOS for supporting low-power THz planetary instruments," in *CICC 2015*, pp. 1–3, Sept 2015.

[15] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016.

[16] A. Fox and D. Patterson, *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC, 2013.

[17] J. Dorsch, "Cpu, gpu, or fpga?." `https://semiengineering.com/cpu-gpu-or-fpga/`, June 2016.

[18] E. Logaras and A. Weitzer, "Using python tools to assist mixed-signal asic design and verification methodologies," in *2017 Austrochip Workshop on Microelectronics (Austrochip)*, pp. 41–46, Oct 2017.

[19] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson, "Avoiding game over: Bringing design to the next level," in *DAC Design Automation Conference 2012*, pp. 623–629, June 2012.

[20] D. L. Rosenband and Arvind, "Hardware synthesis from guarded atomic actions with performance specifications," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 784–791, November 2005.

[21] Mentor Graphics, "Catapult HLS."

[22] "Matlab simulink." `https://www.mathworks.com/products/simulink.html`.

[23] "Intel tick-tock model." `https://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html`.

[24] N. K. Doshi, S. Suryawanshi, and G. N. Kumar, "Development of generic verification environment based on uvm with case study on hmc controller," in *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pp. 550–553, May 2016.

[25] S. Jain, P. Govani, K. B. Poddar, A. K. Lal, and R. M. Parmar, "Functional verification of dsp based on-board vlsi designs," in *2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA)*, pp. 1–4, Jan 2016.

[26] J. Lipman, "Current chip design flow is flawed." `https://www.eetimes.com/document.asp?doc_id=1275808`, April 2003.

[27] D. Lowenstein and M. Woerner, "Test techniques to combat tighter design margins," in *2017 IEEE AUTOTESTCON*, pp. 1–5, Sept 2017.

[28] Galorath, "Seer." `http://galorath.com/products/hardware/cost-estimating-software-hardware-projects`.

[29] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development." `http://agilemanifesto.org/`, 2001.

[30] E. Graves, "Cost of delay." `https://www.playbookhq.co/blog/cost-delay-critical-project-metric-calculate/`, Jan 2017.

[31] N. Johnson and B. Morris, "Agile soc." `http://agilesoc.com/`, 2011.

[32] B. Fitzgerald and G. Hartnett, "A study of the use of agile methods within intel," in *Business Agility and Information Technology Diffusion* (R. L. Baskerville, L. Mathiassen, J. Pries-Heje, and J. I. DeGross, eds.), (Boston, MA), pp. 187–202, Springer US, 2005.

[33] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P. F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, pp. 8–20, Mar 2016.

[34] SiFive. `https://www.sifive.com/`.

[35] Xilinx. `https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html`.

[36] MathWorks, "Hdl coder." `https://www.mathworks.com/products/hdl-coder.html`.

[37] D. Markovic, C. Chang, B. Richards, H. So, B. Nikolic, and R. W. Brodersen, "Asic design and verification in an fpga environment," in *2007 IEEE Custom Integrated Circuits Conference*, pp. 737–740, Sept 2007.

[38] B. Richards, N. Nicolici, H. Chen, K. Chao, R. Abiad, D. Werthimer, and B. Nikolic, "A 1.5GS/s 4096-point digital spectrum analyzer for space-borne applications," in *CICC 2009*, pp. 499–502, Sept 2009.

[39] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[40] L. Li, T. Fanni, T. Viitanen, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, and S. Bhattacharyya, "Low power design methodology for signal processing systems using lightweight dataflow techniques," in *DASIP*, pp. 82–89, Oct 2016.

[41] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sept 1987.

[42] P. K. Meher, "Memory-based hardware for resource-constraint digital signal processing systems," in *2007 6th International Conference on Information, Communications Signal Processing*, pp. 1–4, Dec 2007.

[43] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "Sejits: Getting poductivity and performance with selective embedded jit specialization," in *PMEA*, 2009.

[44] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 141–150, IEEE Computer Society, 2011.

[45] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovi, "The hwacha vector-fetch architecture manual, version 3.8.1," Tech. Rep. UCB/EECS-2015-262, EECS Department, University of California, Berkeley, Dec 2015.

[46] A. Wang, P. Rigge, A. Izraelevitz, C. Markley, J. Bachrach, and B. Nikolić, "Aced: A hardware library for generating dsp systems," in *DAC*, June 2018.

[47] "UltraFast High-Level Productivity Design Methodology Guide," Oct. 2017.

[48] A. Gai, "Model-Based Design with MATLAB, Simulink, and Altera DSP Builder," 2006.

[49] W. R. Davis, N. Zhang, K. Camera, F. Chen, D. Markovic, N. Chan, B. Nikolic, and R. W. Brodersen, "A design environment for high throughput, low power dedicated signal processing systems," in *Proceedings of the IEEE 2001 Custom Integrated Circuits Conference (Cat. No.01CH37169)*, pp. 545–548, 2001.

[50] P. Milder, F. Franchetti, J. C. Hoe, and M. Pschel, "Computer Generation of Hardware for Linear Digital Signal Processing Transforms," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, pp. 15:1–15:33, Apr. 2012.

[51] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, June 2012.

[52] L. Truong and P. Hanrahan, "Magma." `https://github.com/phanrahan/magma`.

[53] ARM, "Amba axi and ace protocol specification," tech. rep., ARM, 2018.

[54] S. Consortium, "Ip-xact working group," tech. rep., Accellera Systems Initiative, 2013.

[55] E. Chang, J. Han, W. Bae, Z. Wang, N. Narevsky, B. Nikolić, and E. Alon, "Bag2: A process-portable framework for generator-based ams circuit design," in *CICC*, Apr. 2018.

[56] J. Chennamangalam, "The polyphase filter bank technique." `https://casper.berkeley.edu/wiki/The_Polyphase_Filter_Bank_Technique`, August 2011.

[57] Y. Sun, J. R. Cavallaro, Y. Zhu, and M. Goel, "Configurable and Scalable Turbo Decoder for 4G Wireless Receivers," in *Fourth Edition Wireless Networks: Applications and Innovations* (S. Adibi, A. Mobasher, and T. Tofigh, eds.), ch. 27, pp. 622–643, IGI Global, 2010.

[58] M. Weiner and B. Nikolic, "A high-throughput, flexible ldpc decoder for multi-gb/s wireless personal area networks," Master's thesis, EECS Department, University of California, Berkeley, Dec 2010.

[59] M. Parker, "Radar basics," *EE Times*, May 2011.

[60] X. B. Mao, Z. G. Ma, F. Yu, and Q. J. Xing, "A continuous-flow memory-based architecture for real-valued fft," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, pp. 1352–1356, Nov 2017.

[61] Texas Instruments, "Tms320c55x dsp v3.x cpu reference guide (rev. e)." `http://www.ti.com/lit/ug/swpu073e/swpu073e.pdf`.

[62] Texas Instruments, "Tms320c6671 fixed and floating-point digital signal processor datasheet (rev. e)." `http://www.ti.com/lit/ds/symlink/tms320c6671.pdf`.

[63] Ceva, "Ceva-xc5 / ceva-xc323: Enabling low power communications in machine to machine multi-mode endpoints." `https://www.ceva-dsp.com/wp-content/uploads/2017/02/CEVA-XC5-Presentation.pdf`.

[64] R. Ginosar, P. Aviely, T. Israeli, and H. Meirov, "Rc64: High performance rad-hard manycore," in *2016 IEEE Aerospace Conference*, pp. 1–9, March 2016.

[65] XMOS, "Xmos xcore architecture." `http://www.xmos.com/published/xcore-architecture-flyer`.

[66] M. Tikekar, C. T. Huang, C. Juvekar, V. Sze, and A. P. Chandrakasan, "A 249-mpixel/s hevc video-decoder chip for 4k ultra-hd applications," *IEEE Journal of Solid-State Circuits*, vol. 49, pp. 61–72, Jan 2014.

[67] Y. Zhang, Y. Kim, A. Tang, J. Kawamura, T. Reck, and M. C. F. Chang, "A 2.6gs/s spectrometer system in 65nm cmos for spaceborne telescopic sensing," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2018.

[68] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[69] `https://riscv.org/`.

[70] `https://github.com/freechipsproject/rocket-chip`.

[71] L. Dong, M. Wang, and S. Shi, "A new digital spectrometer for low frequency solar radio observation based on FPGA," in *ICSPS 2010*, vol. 1, pp. V1–119–V1–126, July 2010.

[72] D. Markovic, C. Chang, B. Richards, H. So, B. Nikolic, and R. W. Brodersen, "ASIC design and verification in an FPGA environment," in *CICC 2007*, pp. 737–740, Sept 2007.

[73] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Crescini, C. de Jesus, C. Dick, P. Droz, D. MacMahon, K. Meder, J. Mock, V. Nagpal, B. Nikolic, A. Parsa, B. Richards, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright, "Petaop/second fpga signal processing for seti and radio astronomy," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, pp. 2031–2035, Oct 2006.

[74] J. Chennamangalam, "The polyphase filter bank technique." `https://casper.berkeley.edu/wiki/images/2/24/Casper_memo_pfb.pdf`, August 2011.

[75] A. Parsons, "The symmetric group in data permutation, with applications to high-bandwidth pipelined fft architectures," *IEEE Signal Processing Letters*, vol. 16, pp. 477–480, June 2009.

[76] R. Emerson, "Biplex pipelined FFT," *The Deep Space Network Progress Report 42-34*, pp. 54–59, August 1976.

[77] R. Finger, P. Mena, N. Reyes, R. Rodriguez, and L. Bronfman, "A calibrated digital sideband separating spectrometer for radio astronomy applications," *PASP 2013*, vol. 125, no. 925, pp. 263–269, 2013.

[78] `https://github.com/ucb-art/craft2-chip`.

[79] D. H. et al., "Breeze." `https://github.com/scalanlp/breeze`, 2009.

[80] S. Bailey, J. Wright, N. Mehta, R. Hochman, R. Jarnot, V. Milovanović, D. Werthimer, and B. Nikolić, "A 28nm fdsoi 8192-point digital asic spectrometer from a chisel generator," in *CICC*, Apr. 2018.