Generators for Wireless Systems Prototyping

Paul Rigge

Electrical Engineering and Computer Sciences
University of California, Berkeley

December 1, 2022

Generators for Wireless Systems Prototyping

by

Paul Jeffrey Rigge

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 Anant Sahai
Professor Bin Yu

Summer 2020

Generators for Wireless Systems Prototyping

Abstract

Generators for Wireless Systems Prototyping

by

Paul Jeffrey Rigge

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Low-latency and high-reliability wireless systems are of increasing interest. There are a number of important emerging use-cases including autonomous vehicles, augmented and virtual reality, and robotics that demand low latency and high reliability. Current wireless systems do not deliver the performance that these applications require. There are many reasons, some of which are being addressed by standardization efforts such as 5G Ultra-Reliable Low-Latency Communication (URLLC), but reliability is only addressed to a limited extent.

The fundamental problem that must be overcome by ultra-high reliability wireless systems is fading. Relaying is a good way to overcome fading and has been shown to be a promising technique for low-latency high-reliability wireless communication.

Relaying techniques have promising results with analytical models and in simulations, but it is not clear how well these models translate to the real world. The low-latency, high-reliability regime is different from the space occupied by Wi-Fi and LTE, which are more throughput-oriented. To demonstrate the efficacy of these relaying techniques, it is important to build a prototype that demonstrates the system concept and allows performance measurements to be made.

Building prototypes of a low-latency, high-reliability wireless system is difficult. For one, it makes using common software-based prototyping techniques difficult because of latency and performance requirements. Custom hardware is necessary for evaluating many of the promising ideas for achieving URLLC. Furthermore, in the process of implementing the system, designers will likely discover new problems and better solutions. Unfortunately, custom hardware is difficult and time-consuming to design and verify. Small conceptual changes to the wireless scheme can result in significant architectural changes to a hardware implementation. Current mainstream hardware design tools and methodologies are ill-suited for tracking these sorts of changes as high-level abstractions, code reuse, and open-source libraries are very limited, especially compared to conceptually similar software tools.

This work envisions a future where designers use high-level abstractions for custom

hardware. We appeal to the way current machine learning frameworks allow productive exploration and fast iteration on network architectures while still affording efficient implementation on a wide range of target platforms. Similarly, future hardware design tools and methodologies should allow domain experts to be productive with high level abstractions that generate efficient and correct hardware. In the context of wireless systems, protocol designers should be able to generate hardware with the same ease they use tools like MATLAB or Python for simulation.

Working towards this vision, this thesis presents a generator-based agile design methodology for wireless systems. Dsptools, a Chisel library for writing reusable signal processing hardware, is an important tool that enables this methodology. Agile design methodologies, generator-based design, and powerful hardware libraries are all important building blocks for realizing this vision.

<div style="text-align:right;">

_____

Professor Borivoje Nikolić
Dissertation Committee Chair

</div>

To Priya.

Contents

List of Figures

List of Tables

Acknowledgements

It is a joy to write these acknowledgements and recall all the wonderful people who have impacted my life while at Berkeley. My advisor Borivoje Nikolić has had a great impact from beginning to end, and I am thankful for so much. My interests are somewhat eclectic and varied, and I'm grateful for Bora's support and his broad knowledge and interests. Beyond research, teaching a special topics course with Bora was a lot of fun and very rewarding, as was getting to visit Serbia with him. Thank you, Bora, for your guidance and support.

Other faculty have also been important in my time at Berkeley. Anant Sahai has been a wonderful influence as a researcher, a teacher, and a citizen. Being a TA for his EE16A class (along with Ali Niknejad, who is also wonderful) was a very memorable experience. Anant's genuine belief in the ideals of Berkeley is invigorating. Fiat Lux! Elad Alon has been a wonderful force in my time at Berkeley; the CRAFT project was a rewarding experience and his leadership and support were instrumental. Jonathan Bachrach's support of using Chisel to do new things was much appreciated. It's exciting to me that Gireeja Ranade is in the faculty paragraph because I had the privilege of knowing her as a fellow grad student! I respect her so much and am thankful for her early mentorship. Venkat Anantharam was my temporary advisor when I first came to Berkeley wondering what I wanted to do. He was a wonderful teacher and his thoughtfulness and patience are a model I aspire to.

I must also thank the people I worked most closely with for the huge impact they had on me. Vasuki Narasimha Swamy was wonderful to work with and be friends with. From seminar talks to TAing for EE16A, Vasuki pulled me into a lot of things that I'm glad to have been exposed to. I admire Angie Wang's drive to have ownership of meaningful and interesting work, and to see it all the way through. I also admire her effectiveness in pushing for catered lunches. Thanks to Stevo Bailey for being a great tape-out leader. Chick Markley was a great partner to work with and is a big reason anything works at all. I admire his curiosity and willingness to jump into new things, as well as his perseverance biking through the hills of Berkeley. Adam Izraelevitz's leadership and stewardship of Berkeley projects has been wonderful, and I'm grateful to have been in the same room while these things were happening.

Matt Weiner and Milos Jorgovanovic were early mentors when I joined Bora's group, and I'm grateful for how welcoming they were. There are so many members of and visitors to Comic, Aspire/Adept, and BLISS who have been a big part of my life, including Amy Whitcombe, Keertana Settaluri, Colin Schmidt, Harrison Liew, Daniel Grubb, Pi-Feng Chiu, Ben Keller, Yue Dai, Sameet Ramakrishnan, Vignesh Iyer, Sijun Du, Antonio Puglielli, Vidya Muthukumar, Rachel Hochman, Zhaokai Liu, Katerina Papadopoulou, Brian Zimmer, Albert Ou, Albert Magyar, Jaeduk Han, Nathan Narevsky, Jack Koenig, Jim Lawson, Brian Richards, Richard Lin, Sahaana Suri , Meryem Simsek, Marko Kosunen, Timo Joas, Vladimir Milovanovic, and Alon Amid. Chris Yarp, John Wright, and I had a fun early project making a Viterbi/BCJR decoder together, and I've felt fortunate to know both of you ever since. You've been great friends and I've learned a lot from you. Mirjana Videnovic-Misic (Mira): you were my desk-neighbor, my host

Chapter 1

Introduction

Prototyping is where promising new ideas meet reality for the first time. Being able to rapidly prototype new ideas allows for them to be iterated upon and improved more quickly or abandoned if unforeseen shortcomings are found. This work is centered around prototyping techniques for ultra-reliable low-latency communication (URLLC), and how those techniques evolve through the process of prototyping.

This work discusses not only building wireless URLLC, but also prototyping generally. The organization of the writing represents this. Early sections focus on wireless techniques, which are made progressively more concrete as the discussion shifts towards building prototypes. Of course, the evolution of these ideas was not a linear progression; there was a back-and-forth and ideas evolved mutually.

1.1 Ultra-Reliable Low-Latency Communication

Low-latency wireless communication is increasingly important for emerging applications. Cyber-physical systems involve many computer-driven systems interacting with the physical world. These systems benefit from being able to communicate and may interact with physical phenomenon that require decisions to be made at very short timescales, often at least as quickly as human reaction time. These systems often can be thought of as control systems, either centralized or distributed. Autonomous vehicles communicating with each other for platooning or traffic management are one prominent family of use-cases. Augmented reality (AR) and virtual reality (VR) are another category of applications that demand low latencies at timescales dictated by human reaction speeds. Low latency is critical for providing immersive interaction, for example overlaying virtual objects on the physical world [54] and catching balls in VR[34]. In [32], Fettweis calls this family of applications the "tactile internet".

Existing applications using wired networks for latency-sensitive communication may also benefit from low-latency wireless communication. Critical control applications in factory automation and robotics applications see frequent downtime from stressed wires fracturing. Many existing systems that use wires could significantly reduce cost, complexity, and weight by replacing them with wireless communication systems [122].

Reliability is critically important for many low-latency applications. Many cyber-physical applications by their nature have important and challenging safety requirements. Control algorithms can be designed to take unreliable communication into account [91], but this often relies on a good model of the reliability of the channel and has implications on stability and cost of control. The design of control algorithms is simplified by being able to assume reliable communication, and existing wired systems can be retrofitted to be wireless much more easily if the wireless link is as reliable as the wired.

Ultra-reliable low-latency communication (URLLC) is desirable if it can be achieved wirelessly, but there are many questions:

- What techniques are needed for URLLC?

- How different is a URLLC system from existing wireless systems that optimize for throughput, power, range, etc.?

- How much does it cost in power, computing, and spectrum to achieve URLLC? Is it practical?
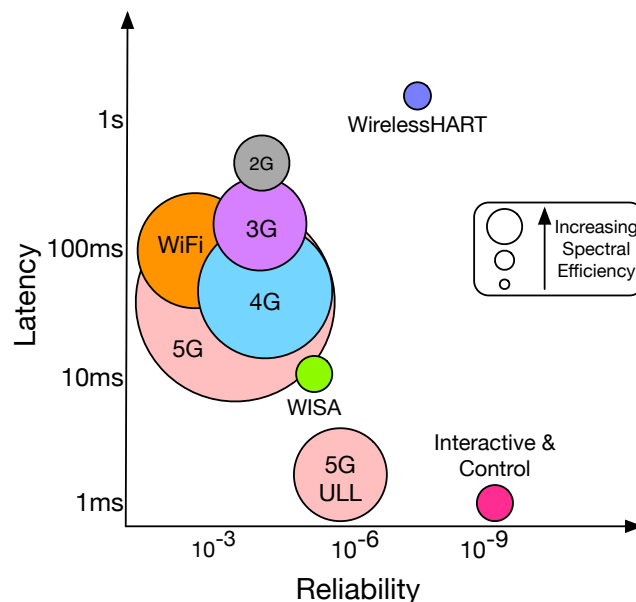
Figure 1.1. Illustration of relationship between typical reliability, latency, and spectral efficiency for various wireless standards. This is meant to be illustrative but not exact, the values of these parameters will be situational.

Existing wireless communication standards are not optimized for URLLC. Fig. 1.1 shows where interactive and control applications sit in the reliability/latency/spectral efficiency space relative to some wireless standards. The 802.11 and cellular standards occupy an area in the figure where latency and reliability are "good enough" for applications such as web browsing, VOIP calls, and gaming. Latency has steadily improved over the generations of cellular standards, but the primary improvement has been spectral efficiency. WISA is a standard for wireless audio, and as such requires modest spectral efficiency with low latency, but the latency and reliability are not good enough for interactive and control applications. WirelessHART has the ability to achieve very high reliability but is not designed to have very low latency. Interactive and control applications occupy their own corner of the figure. 5G ultra-reliable low-latency communication (URLLC) proposals also have low latency, but the reliability is not to the same level demanded by interactive and control applications.

The requirements demanded by interactive and control applications are different than those provided by existing wireless standards. As a result, the techniques used to achieve URLLC may need to be very different from techniques used in throughput- or power-oriented standards.

1.1.1 Problem Formulation

Latency and Reliability

Latency is often specified as a single number, for example "the propagation time of the signal is $1\,\mu s$". However, in many cases a latency constraint without a reliability constraint is not very useful. Formally, latency can be thought of as a stopping time associated with an event E of interest, and E can be probabilistic in such a way that the stopping time associated with E has a distribution.

It is important to discuss latencies corresponding to events that are useful. A degenerate example is designing a system with "zero latency" by choosing an event where a unit of information is communicated, called E, where E takes 0 time but $\Pr[\text{Error}|E] = 1$. In less degenerate circumstances, error correction codes and retransmissions can be used to increase the reliability beyond the raw error rate of the channel, but these can add (potentially random) latency. Reasoning about the latency of more complicated events can be difficult; rather than a single quantity, there may be an unwieldy distribution. When latency specifications allow for randomness at all, it is usually via a percentile, for example "the 99.99th percentile latency is no more than $50\,\mu s$." This style of specification is most useful when building a system out of parts that are much more reliable than the overall system needs to be, allowing the designer to ignore the rare events and treat the latency as a number instead of a distribution. However, when building ultra-reliable systems, it may be prohibitively expensive to get individual components that are more reliable than the overall system. In that case, specifying latency with a percentile is less useful as

the 99.99th percentile latency does not necessarily say very much about the 99.9999th percentile latency.

Control Application

Industrial control applications are widely deployed and require low-latency and high-reliability communication. They serve as a good exemplar for determining requirements for wireless networks that will enable useful new applications. Industrial control applications are implemented with wired fieldbus standards, commonly SERCOS III. Consider the following problem formulation based on such applications [123].

There is a controller, called C, and n nodes $\{s_i\}_{0 \le i < n}$. We consider n chosen such that $10 \le n \le 100$. The controller and the nodes are in an indoor environment and relatively close to each other such that all nodes and the controller are "within range" of each other, that is on average each node expects to have similar SNR to each other node.

For control applications, nodes send packets containing sensor measurements to the controller, and receive packets containing actuation instructions. These packets are generally a position and/or velocity vector, typically between 10 and 50 bytes in a packet. Throughout this work, unless otherwise specified there are 20 bytes per packet. Call this entire information exchange a cycle.
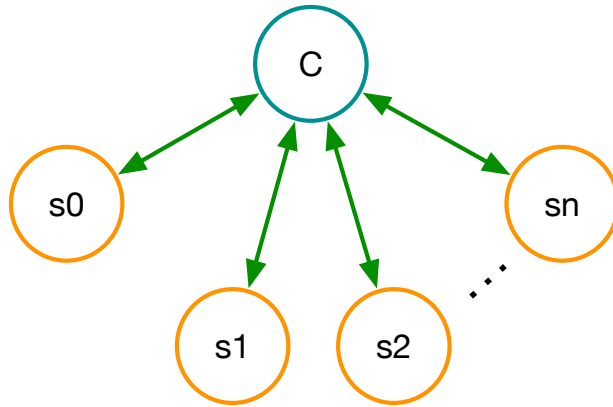
Figure 1.2. Star topology. The controller C is the center of the star, with individual connections to each node $\{s_i\}_{0 \le i < 0}$.

In this arrangement, information moves in a star topology, depicted in Fig. 1.2. Note that this information flow does not necessarily reflect the communication that takes place, it only depicts the way entities produce and consume information.

A cycle must take a short amount of time in order to ensure that the system is responsive when interacting with the physical world. This work assumes a cycle time of 2 ms, which in [32] is said to be the minimum latency to be able to interact physically with humans. This cycle time is not a strict constraint on individual transmissions; nei-

ther the order nor the time an individual transmission takes matters as long as the entire cycle completes successfully in time.

Control systems are typically implemented with wired systems, and it is desirable to have a drop-in wireless replacement for these systems[1]. One way to achieve this is to try to build a wireless system where point-to-point links perform similarly to a wired system, but this is difficult. The gigabit ethernet over copper commonly used in control applications requires a link-level bit error rate (BER) better than 10^{-10} [92]. Building a wireless link with this kind of performance is difficult and prohibitively expensive. However, the reliability of an individual link is not what truly matters; rather, the reliability of a cycle is what ultimately matters.

A wireless implementation of URLLC does not need to achieve 10^{-10} BER for each link in order to achieve an acceptable cycle error rate. A cycle time of 2 ms and probability of cycle error of 10^{-9} to 10^{-8} are representative targets for many control applications [123], which corresponds to an error occurring roughly once a year on average.

The high-level, application-centric system parameters discussed here are summarized in Table 1.1. A wireless system that meets these requirements is a drop-in replacement for wired fieldbus systems in industrial control applications. Furthermore, meeting these requirements enables other emerging applications such as AR/VR and autonomous vehicles.

Parameter	Value	Description
n	$10 - 100$	Number of nodes in the system
T_{cyc}	2 ms	Duration of a complete cycle
P_{err}	10^{-9}	Probability a cycle fails

Table 1.1. Requirements for communication systems supporting low latency applications.

1.1.2 Achieving Reliability

In the indoor, close-range communication scenario posed here, multipath fading is a fundamental barrier to achieving URLLC [102]. Multipath fading is the result of reflected copies of a transmitted signal interfering with each other. With some probability, the interference results in a receiver getting very low or no measurable signal power, resulting in dropped packets, repeated retransmissions, or similar depending on

---

[1]Development and certification of control systems is expensive, time-consuming, and often requires domain-specific expertise, especially when safety is a concern. Retrofitting existing networks without having to update the rest of the system minimizes switching time and costs.

the communication standard. Having a low-latency requirement compounds this problem. For carrier frequencies used in cellular and Wi-Fi standards[2], the coherence time of the channel is longer than the latency requirement. This means that a node in a deep fade does not have enough time to wait for its channel to improve.

One potential solution to fading is to increase transmit power until the receiver has an acceptable SNR. However, achieving high reliability this way may require overpowering very deep fades, and a transmitter that can achieve such high transmit power is prohibitively expensive and may not be allowed for safety, regulatory, or cost reasons. Instead, some form of diversity scheme is needed to overcome fading. Studies [102, 101, 103, 99] have shown that spatial diversity is a useful technique for achieving URLLC. Spatial diversity exploits the fact that the wireless channel is a broadcast medium and uses extra users to relay information around bad channels.

Relaying techniques are promising because they provide a mechanism for achieving extremely high reliability. However, relaying adds some latency, and if done poorly can add a large amount of latency. If every user takes turns relaying for every other user, it increases the total number of transmissions by a factor of n, and latency for the network scales (poorly) by n^2. Therefore, a key feature of the studies mentioned above is ways of limiting the overhead in having relays. In particular, space-time codes and network coding can dramatically reduce the overhead of relaying. However, it is not entirely clear that all these techniques are practical to implement and will achieve their promised reliability in real-world settings. Furthermore, these techniques rely on modeling assumptions that need to be verified because of the extreme reliability requirements. This work focuses on building prototypes that enable studies of the practicality of these low-overhead relaying schemes.

1.2   Demonstration of System Concepts

The relay-based URLLC systems discussed in this work involves using new wireless techniques. Demonstrating the value of these ideas involves validating predictability of channels and relay selection algorithms with real-world measurements. Building prototypes further demonstrates the value of these ideas and is especially critical for high-reliability systems because these systems are built around modeling assumptions that a prototype helps validate.

---

[2]Cellular and Wi-Fi standards are designed with different latency-reliability targets, and as a result can exploit time diversity. In simple terms, this means they choose to keep retransmitting until the channel improves.

1.3 Prototyping

A prototype should be as simple as possible while still capturing the important characteristics of the solution. Simplicity allows for faster iteration and makes it easier to interpret how well the prototype works. However, prototypes still need to be realistic enough to actually demonstrate the system concept and make meaningful performance measurements.

One important aspect when building a prototype is choosing the system boundary. Often, prototypes can be made by isolating one piece of the system and testing it, later using standard parts, models, or offline simulation to extract relevant performance metrics and fill in the missing parts of the system. In other cases, especially where models are difficult to make (e.g. there is complex interaction with a human), isolating one piece of the system may lead to less meaningful results.

Where possible, it is best to prototype on general purpose processors rather than implementing custom hardware. Software prototyping is faster, cheaper, and requires less specialized knowledge. Software-defined radio (SDR) platforms such as GNURadio [39] are convenient for quickly implementing a working system, but they are ill-suited for the extremely low-latency relaying that is required. In GNURadio, high level functionality can be implemented in Python, a productivity language, and performance-critical functionality can be implemented in C++, a performance language. This allows designers to iterate quickly and still achieve reasonable performance, especially as general purpose processors continue to get faster and have more cores. However, it is not always possible to use SDRs to implement prototypes.

Relay-based URLLC systems present several challenges for software-based implementation, and as a result must be implemented with custom hardware [48]. One challenge is that SDRs typically batch up inputs to achieve good performance. There is a substantial amount of overhead in synchronizing across cores, performing vector operations efficiently, scheduling workers, and moving data. To achieve good performance, it is important that systems such as GNURadio batch data into large enough chunks. Batchings and schedules that achieve good throughput can have large latencies. This issue is further compounded by the fact that the latency may be unpredictable on general purpose hardware. Caching effects, operating systems, and the overall complexity of general purpose cores can make workload runtimes variable, which is undesirable when evaluating an ultra-reliable system.

Where general purpose hardware struggles, purpose-built custom hardware can be matched to the latency and reliability requirements of an application. FPGAs are commonly used platform for prototyping custom hardware, especially for wireless systems. FPGAs use a similar design flow as application-specific ICs (ASICs), but can be reprogrammed quickly and easily. This work uses FPGAs with commercial RF daughter cards for the frontend and custom digital hardware for the baseband.

While FPGAs enable prototyping of custom hardware, designing hardware is time-consuming and difficult compared to writing software. This is a huge problem for making

a prototype where it is essential that a small team of domain experts be able to quickly iterate on ideas.

For software prototypes, there are a rich set of open source libraries and frameworks that allow developers to be instantly productive. Furthermore, software prototypes can combine the use of performance languages for the portions of the design that matter and productivity languages for everything else, as in GNURadio or TensorFlow. Software prototyping favors starting with small examples and working incrementally towards a goal with an agile methodology.

In contrast, open source hardware has traditionally been of limited use and difficult to use. Hardware tools struggle to make a good trade between performance or productivity, tending towards one extreme or the other. Agile development is difficult for hardware designs.

This work focuses on making the design of custom signal processing hardware easier, faster, and less error prone. These efforts are important steps in enabling a future where hardware prototyping is just as productive and efficient as software prototyping.

1.3.1   Agile Design

Typical hardware designs are built using a waterfall model [57]. The waterfall model, depicted in Fig. 1.3, involves breaking down the design task into a series of tasks that happen one after another. For hardware design, the sequence of tasks is specification writing, architecture design, implementation, testing, and finally shipping. There is a relatively rigid dependency between these tasks; the later stages cannot finish before previous stages are complete, and late design changes can be very difficult.

In contrast, agile design models have been very successful for software development. The main emphasis of the agile model is iterative improvement. Instead of delivering a complete, finalized output, the first iteration under the agile model should deliver a simplified version of the system. These simplifications should make it much easier and faster to deliver. Then the design process is repeated and improvements are incrementally made.

There are several advantages to the agile model. One of them is that performing the design loop multiple times makes it easier to estimate the time and difficulty of the steps in the design process. Under the waterfall model, it is possible to run into difficulties partway through a project that an agile project could expose at the beginning. Furthermore, specifications evolve due to external factors like market changes or uncertain customers, and agile projects are better equipped to incorporate changes to specifications throughout the design cycle.

The agile design model relies on being able to perform design and verification iterations quickly. Design automation is a powerful tool for increasing the speed of iteration. Reusable components, fast verification, and high level design are important enablers for agile design.
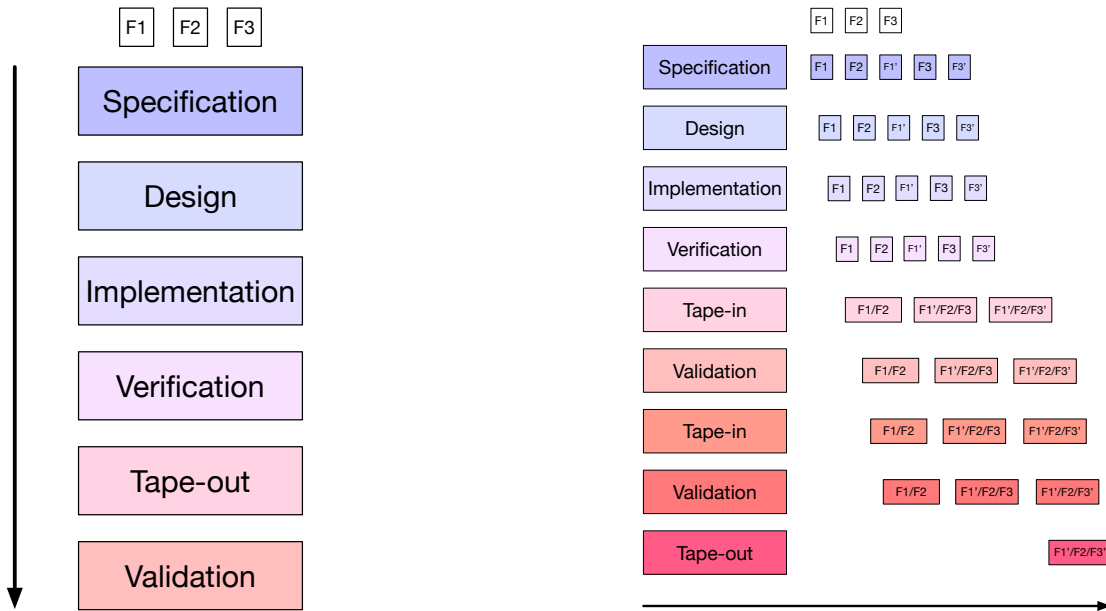
Figure 1.3. A representation of the waterfall and agile design models. The F labels represent features. Under the waterfall model, depicted on the left, all features go through the design sequence together, producing a tape-out result when all features have completed the entire design process. Under the agile model, depicted on the right, features are added incrementally and individual features can be iterated upon or even abandoned.

Reuse in hardware design via IP blocks is typically done at a coarse level. IP blocks generally implement some high level standardized functionality and has standard interfaces and may be configurable to some degree. This style of reuse is useful, but the degree to which IP blocks are configurable is usually limited. Furthermore, IP blocks are not typically useful for more fine-grained functionality or design patterns that are scattered throughout a design.

1.3.2 High Level Synthesis

High level synthesis (HLS) is a popular technique that enables reuse and higher level abstractions for hardware design. HLS maps a description of an algorithm from a general purpose programming language (generally C/C++) to a hardware implementation. This is analogous to synthesis, which maps algorithms expressed in a hardware description language to a hardware implementation, but general purpose programming languages generally do not have constructs for expressing low-level circuit elements, for example registers or clocks. The idea is that a general purpose programming language will express the algorithm at a higher level of abstraction and allow the synthesis tool to make choices about the hardware implementation.

Like machine learning frameworks such as Tensorflow [8] or PyTorch [73], the ultimate goal of HLS is to allow designers to be productive in a high level language and

still get good quality-of-result (QoR). Unfortunately, current HLS tools are not entirely successful at providing both of these features. Getting good QoR with HLS tools is commonly cited as a challenge and often pushes HLS designers into abandoning some of the high-level abstractions that make HLS appealing.

Designers typically must guide HLS tools via `#pragmas` that direct the compiler to perform some action, for example specifying how to unroll loops or how to map IO. These `#pragmas` are important knobs for performance tuning a design. However, they can be difficult to reason about and can interact with each other, so small changes to the design or pragmas can result in large changes in the performance of the output. The proper tuning can differ based on the context the code is being used in, which makes getting reusable code that has good QoR difficult under HLS.

Another issue with HLS is the fact that the general purpose programming language generally does not make a distinction between the code that gets mapped to hardware and the metaprograms that determine what code should be generated. For example,

```
if (add) {

  out = in0 + in1;

} else {

  out = in0 - in1;

}
```

In the previous example listing, it is not clear if `add` is a variable used as part of a meta-program that determines whether or not the circuit implements an addition or subtraction operation, or if instead it is an input to a circuit that performs both addition and subtraction and selects one of the operations to output. This distinction can be important for QoR.

HLS tools that produce high-quality output are generally closed source commercial tools. They are typically not very extensible. Pragmas or TCL directives are the preferred way of interacting with the tool, but these tools generally do not allow for creating custom pragmas to interact with a compiler plugin. Even though many HLS tools are LLVM-based, most tools map LLVM to a hardware-oriented internal representation and do not expose this to users. As such, compiler plugins are not typically supported.

1.3.3 Generators

HLS can be thought of as adopting a "top-down" approach, starting with a high-level abstraction and mapping it down to a low level hardware implementation. Hardware generators represent an alternative, "bottom-up" approach. Like HLS, hardware generators offer the ability to design hardware with high-level abstractions. However,

10

generators also offer a much higher degree of control over the output hardware. This combination of high-level abstraction with fine-grained control is perhaps spiritually similar to Halide [78] or TVM [25] compilers that allow for fine-grained optimizations of deep learning workloads.

A hardware generator is a program that takes input parameters and generates a circuit according to those parameters. Chisel [18], a hardware construction language for writing generators, uses underlying constructs similar to those in hardware description languages (HDLs) like Verilog. However, Chisel is embedded in Scala and thus can use the powerful constructs available in a modern, general-purpose programming language. Furthermore, Chisel enforces a strict separation from the hardware constructs and meta-programming constructs.

One simple example is the distinction between `3.U`, which is a hardware integer literal with value 3, and `3`, which is an integer that can be used during circuit generation, but does not represent hardware. For example, `3` can be used as a limit in a for loop, but cannot be assigned to an output of a module. `3.U` should not be used[3] as an integer, but it can be used to assign to an output of a module. This separation makes it very clear what parts of the program will generate hardware and what parts do not, a distinction which can be less clear in an HLS flow. By separating general purpose programming language constructs from constructs that map to hardware, Chisel allows for designers to use low-level circuit primitives available in a typical HDL with high-level constructs available in a modern productivity language.

The Chisel ecosystem is extensible. Chisel generated designs are divided into a frontend and backend inspired by the design of LLVM. The compiler frontend consists of a program accepting parameters and elaborating the design, producing an intermediate representation called the Flexible Intermediate Representation for RTL (FIRRTL). The frontend can label elements in the output circuit with annotations that may trigger different functionality in the backend. The backend takes the FIRRTL from the frontend, which contains some higher level abstractions, and runs a series of transformations on the intermediate representation (IR) that ultimately lower the circuit to a concrete representation, which can be serialized as Verilog. Annotations from the frontend trigger user-defined transformations to be run. The user-defined transformations can modify the circuit in very powerful ways, for example instrumenting the circuit with performance counters automatically.

The Chisel ecosystem is extensible in another important way: libraries in Chisel can provide new functionality at multiple levels of abstraction. In contrast to HDLs like Verilog where the notion of a library usually means something like an IP block, Chisel libraries can provide reusable functionality at high granularity and at different levels of abstraction. For example, a Chisel library can provide a high level API for functionality that adds a backend transform to perform very low level modifications to the circuit.

---

[3]With some introspection, the value of the literal can sometimes be recovered and could hence be used as a limit in a for loop. However, expressions like `2 < 3.U` will not type check and it is generally a type error to use a `UInt` like it is a normal integer.

11

Dsptools is an example of a Chisel library. It provides reusable components that help with writing custom signal processing hardware. It was developed for and used extensively in this work. Reusable components in Dsptools are not just the commonly used IP blocks. Typeclasses in Dsptools allow a designer to abstract over the numeric type used in a module, for example allowing a designer to write one description of an FIR filter that can target both real- and complex-valued filters. This allows more fine-grained reuse and flexibility of reconfiguration than is typical of IP blocks.

1.3.4 Verification

Verification is the process of testing and debugging a design, generally evaluated against requirements or a golden model. It is often cited as the long pole in the chip-design tent. Verification has a strong impact on time-to-market as well as the number of re-spins needed for a design, both of which are critical to the success of a project.

The Universal Verification Methodology (UVM) is an industry-standard methodology for verifying designs. In UVM, verification tasks are split into separate components: drivers that generate stimulus, monitors that collect outputs, assertions that enforce circuit properties, and scoreboards that check if the actual outputs matched the expected outputs based on the input. One of the benefits of this partitioning of functionality is that these components can be reusable, saving verification time.

Generating designs can make verification more difficult and can add complexity when using UVM-style methodologies. Generated names are a commonly cited issue for designs generated by Chisel. Extra nodes generated by the compiler in the process of lowering more abstract operations to more concrete operations often have meaningless names like _T_30 or _GEN_4096, which can obfuscate the circuit and make tracing the root cause of a bug more difficult. This problem is somewhat fundamental to the nature of generated hardware; generators should be generating code. This kind of problem also exists to some extent in HLS designs. With time to mature and be integrated alongside standard industry tools, these sorts of problems can be mitigated.

Another way in which generated designs can increase the verification burden is by enabling designers to make sweeping changes late in the design process. To the design team, this is a feature: tweaking the specification at the last second and getting a new, substantially different design out quickly is powerful. To the verification team, this sort of late change has the potential to throw a lot of their previous work out the window.

A good solution to this problem is to make the verification environment parameterized in the same way as the design environment. Chisel testers can introspect the design to extract parameters and programmatically generate stimulus and assertions. This is very powerful, but it is important that the verification environment be at least as powerful as the design environment. To that end, Dsptools provides testing infrastructure for testing type-generic designs.

1.4 Research Contributions

This work presents a system architecture for achieving wireless URLLC. It also presents a methodology for prototyping systems with custom digital signal processing hardware and a transceiver developed with this methodology. The needs of the system architecture inform the design of the prototyping methodology.

The architecture consists of a custom OFDM-based baseband written in Chisel using Dsptools. This custom baseband enables experimentation with relay-based communication schemes for URLLC. The baseband is interfaced with an RF frontend and an FPGA with ARM CPU running Linux with a driver for the baseband, allowing for conventional Linux-based software to interact easily with the radio.

Reusable components that use Dsptools are key enablers of the methodology. The methodology includes designing generators for custom signal processing hardware. There are constructs that aid in translating the mathematical abstractions to efficient hardware. Furthermore, the methodology automates packaging this custom signal processing hardware into an SoC using standard interfaces. Automation for verification is also discussed, especially in the context of using generator parameters to create flexible verification suites.

These tools and methodologies are important building blocks for enabling productive high-level design exploration. Ultimately, the goal of these building blocks is to build compilers that enable domain experts, especially in wireless systems, to quickly and efficiently build hardware to evaluate new ideas.

1.5 Outline

Chapter 2 presents an overview of important background such as existing wired standards, 5G standardization of URLLC, and channel modeling. Background for agile hardware development and automation in design and verification are also presented.

Chapter 3 discusses some techniques for achieving URLLC. This includes work from [102, 101, 103, 104]. Different relay-based schemes and their relative merits are discussed.

Chapter 4 discusses how to implement these techniques at the PHY and MAC levels, as well the requirements this imposes at the PHY and MAC levels. OFDM-based transceivers are discussed, and design decisions in the context of relay-based URLLC are explored.

Chapter 5 discusses the viability of dynamic relay selection proposed in Chapter 3 with analysis, simulation, and real measurement data. This includes work from [95, 100, 99].

Chapter 6 discusses a generator-based methodology for designing and verifying cus-

tom signal processing hardware. This methodology is employed to develop a prototype system. This includes work from [120, 82, 19, 119].

Chapter 7 presents work on designing and implementing pieces of a prototype, along with a discussion of the methodology for designing and verifying prototypes.

Chapter 8 concludes this work and proposes some areas of future research.

Chapter 2

Background

2.1 Low-Latency, High-Reliability Communication

This work combines concepts from both wireless systems and hardware design. This chapter reviews literature in both areas, starting with wireless concepts including cooperative communication, network coding, and channel modeling, as well as discussion of recent standardization efforts with 5G. Then we review literature in hardware design and verification, including high-level synthesis, UVM, and generator-based hardware design.

2.1.1 Wired Protocols

Typically, applications in industrial control use wires to communicate between controllers, sensors, and actuators. Point-to-point communication via simple analog current loops was an early method used for control systems and is still used today. The rise of digital signaling allows for more complex network topologies than simple point-to-point systems. Fieldbus systems are more scalable than point-to-point systems and dramatically reduce the number of wires for large systems. Some examples of industrially oriented fieldbus standards are SERCOS III, Profibus, and CAN [86, 76, 27]. SERCOS III is one of several standards that use ethernet. Time-sensitive networking (TSN) is a set of standards that augment ethernet with mechanisms for bounding latency, controlling variation of latency, and other mechanisms for ensuring reliable communication [109]. Originally developed for audio-video bridging (AVB), TSN is being considered for use in industrial and automotive applications [68].

For automotive and aerospace applications, wires add significant weight and

cost [123]. Furthermore, wires add points of failure in systems where wires are under strain, for example a robotic arm with wires that run through moving joints. Wires eventually break after flexing too many times, which are a significant source of downtime in robotic systems. Wired systems also present challenges for scalability and deployment that make wireless systems look attractive [127].

2.1.2   Wireless Systems for Industrial Automation and Control

The many shortcomings of wired networks for industrial automation and control motivate the investigation of wireless solutions. Wireless Sensor Networks (WSNs) employ wireless communication for monitoring applications [11]. Typically, WSNs consist of small, battery-powered, low-cost motes that collect data and report relatively infrequently to extend battery life. Often, WSNs maximize the amount of time a mote stays in the sleep state to conserve power, which is a very different design goal than communication for real-time control, and leads to very different solutions [40].

Wireless communication has been proposed for other industrial control applications. WirelessHART and ISA100 support reliable communication via techniques such as frequency hopping and mesh networking with path diversity [51]. However, they do so with high latency as each packet takes 10 ms and multi-hop communication and retransmissions are common. These standards are designed and more suited for monitoring applications than for critical control applications. ZigBee PRO [12] is another standard targeting industrial control, but does not employ frequency hopping and is thus ill suited for providing high reliability [51]. Wireless extensions to existing fieldbuses have also been proposed, but the use of techniques like CSMA result in unbounded delays that make them unsuitable [24].

Another approach for bringing wireless communication to low-latency, high-reliability applications is to take popular standards like 802.11 or LTE and modify them to support this new class of applications. As is, these standards are not suitable for URLLC. Both standards have many sources of deterministic delay; fixed delays, long preambles, and coarse scheduling granularity make very low latency impossible. Further, there are unpredictable sources of error. In 802.11, CSMA allows for potentially unbounded delays when many users are trying to access the channel. Long ACK/NACK chains in LTE routinely make some transmissions in LTE take several frames to complete, each of which is 10 ms.

802.11 for Control Applications

There are some standardization efforts to address these shortcomings and allow 802.11 and 5G systems to be used for URLLC scenarios. 802.11ax introduces several new features of interest for latency- and reliability-sensitive applications. Trigger-based multi-user uplink allows many users to transmit smaller packets together with consid-

erably less overhead, and new techniques for scheduling allow for efficient scheduling of transmissions to meet latency requirements [30]. Together, these techniques address many of the inefficiencies 802.11 incurs when used for control-oriented networks.

5G Standardization of URLLC

5G standardization efforts recognize supporting URLLC as an important goal [58]. Industrial use-cases are envisioned, planned, and being evaluated.

5G standardization work is gathered in discrete releases, and at any given time there is generally one release that implementations are being designed for, one release that is being evaluated, and a new release that is being discussed. 3GPP Release 15 ("Rel-15") introduced new features for 5G systems to begin adding support for URLLC communication [6]. This release addresses the latency component of URLLC more than the reliability component [37]. It does so by addressing several of the limitations in the conventional way 4G and 5G operate, including coarse frame granularity, long orthogonal frequency-division multiplexing (OFDM) symbols, and scheduling issues, illustrated in Fig. 2.1. 5G is expected to be more flexible and support a wider range of applications than previous 3GPP standards, and one such way it achieves this is through flexible numerology. Sizing for important parameters such as cyclic prefix length, subcarrier spacing, symbol duration, and scheduling intervals are very flexible, and some parameters have been chosen to efficiently support the short packet lengths and low latency required by URLLC applications. Front-loaded demodulation reference signal (DMRS) places pilots before data and allows channel estimation to be performed earlier. Mini-slots in the downlink as well as grant-free uplink scheduling allow for efficient scheduling of short packets with low latency. URLLC traffic is expected to coexist with other traffic, and there are multiple strategies proposed for allowing this [45].

3GPP Release 16 ("Rel-16") added several new features for URLLC [7]. Time Sensitive Computing (TSC) is based on ethernet's TSN [37] and has features, largely centered around providing good synchronization, low latency, and low variability in latency. Rel-16 also enhances 5G-LAN services, which allows for more efficient user-to-user communication when users are collocated. Transmission and reception points (TRPs) can transmit or receive jointly (generally non-coherently), allowing the possibility of exploiting spatial diversity.

Rel-15 supports data duplication at the packet data convergence protocol (PDCP) layer. Path diversity is discussed in the context of directional mmWave radios. Joint multi-TRP may provide a mechanism for exploiting diversity, but as of now it is a building block for a solution rather than a solution. Some studies have been performed for multi-user diversity in the 5G context [58], but standardization efforts seem not to have moved in any direction towards providing diversity techniques for URLLC.
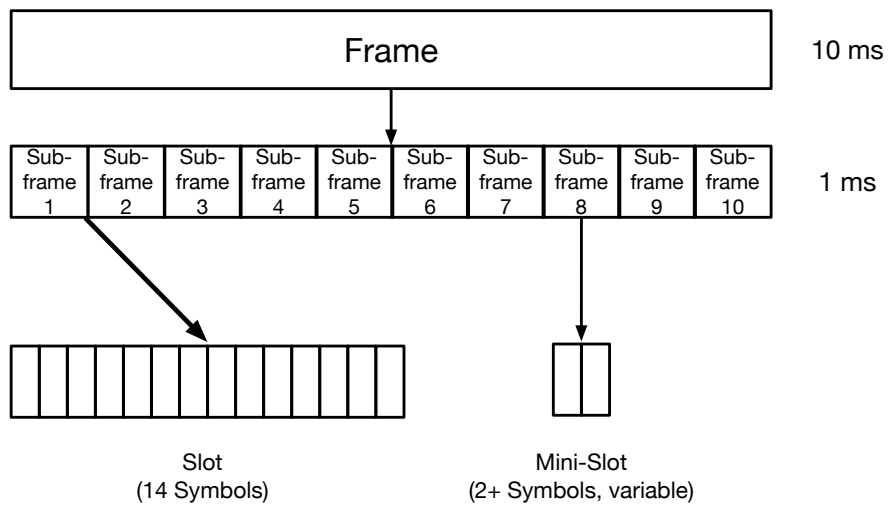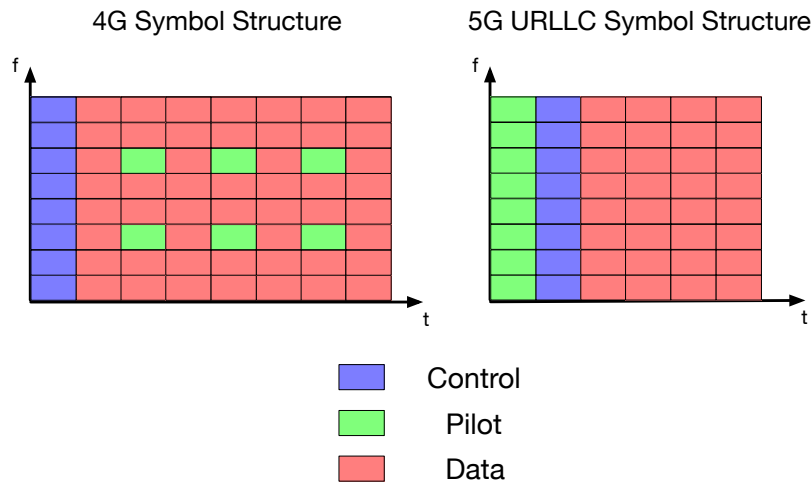
Figure 2.1. Illustration of 5G URLLC features. A new symbol structure specifically tailored for URLLC is described. In contrast to the conventional symbol structure that exists in 4G (top left), the new symbol structure puts pilots first and separate from data, followed by control and data (top right). A new type of slot is also allowed, depicted on the bottom. A frame is composed of sub-frames which have a fixed number of OFDM symbols. Mini-slots are defined for the small packets encountered in URLLC. They are variable-length, with as few as 2 symbols. They are also scheduled differently to support low latency.

2.1.3 Diversity Techniques for URLLC

Diversity is a key technique for achieving reliable wireless communication in the presence of fading[1]. The idea behind diversity techniques is that the same information can be sent via different channels to improve reliability[111]. The channels can be spread over some combination of time, frequency, or space, but as long as they are not failing in correlated ways, adding more paths will increase the probability that the transmission succeeds.

Diversity techniques are critical for reliable wireless communication. WSNs commonly employ time (via retransmissions), frequency (via frequency hopping), and spatial (via path diversity) diversity to achieve high reliability. WirelessHART employs all these techniques. Time diversity is unfortunately not suitable for URLLC applications as the latency requirement is often on the same time scale as or shorter than the coherence time. As a result, the channel will change too slowly to get enough realizations to achieve high reliability.

Frequency diversity is more feasible to implement for URLLC, but has some scalability limitations [102]. Furthermore, frequency diversity depends on the availability of independent channels at different carrier frequencies, which may impose restrictive environmental constraints and implementation burden.

Spatial diversity via user cooperation [84] is an attractive technique for achieving URLLC. Target applications have a large number of nodes, and therefore have a large number of antennas to exploit. Spatial diversity can be exploited in two ways. Transmit diversity uses multiple transmit antennas to reliably send a message to a specific user, and receive diversity uses multiple receive antennas to ensure a specific message is received. Distributed antennas can achieve the full transmit diversity, so physical arrays are not intrinsically more reliable [55].

2.1.4 Coding for URLLC

Coding techniques are not suited for overcoming fades, but they are still important for efficient and reliable communication when the channel is not faded. URLLC imposes some challenges upon conventional coding techniques because of the short packet sizes. Polyanskiy [75] gives us tools to understand ideal code performance of short block-length codes. Several 5G candidate codes including polar codes, LDPC, turbo, and tail-biting convolutional codes (TBCC) have been studied at short block lengths [107, 89]. There is not a clear winner among these, although TB-CC and polar codes may be more attractive than the alternatives. BCH codes are also promising [90]. Other work in coding-free wireless control has also been performed [60], although without addressing deep fades for ultra-reliable systems.

Recent work in [46] proposed RNN-based codes with locality inspired by convolu-

---

[1]Notably, coding techniques are not suited to overcoming fades.

tional codes that allow for decoding before receiving the entire message. These RNN-based codes outperform convolutional codes and are also robust and adaptive to channel conditions.

2.1.5 Channel Modeling

Work in [99, 100] presents a modification of [102] that reduces implementation complexity by removing all simultaneous transmissions, instead selecting a small number of high-quality relays. If a small number of relays can be used with confidence that they will have good channel realizations, this will dramatically reduce requirements for synchronization and channel coding. Time-division multiplexing of relays is attractive because of its simplicity, and if high reliability can be achieved with a small number of relays, the overhead may be acceptable. Of course, this approach relies critically on reliably choosing relays with good channels, which in turn depends on the characteristics of the wireless channel.

Relay selection algorithms in the literature typically use some long-term average behavior of a link as optimization criteria [47]. However, these algorithms are less suitable for ultra-reliable communication because of multipath fading. Links with the same average SNR may have very different instantaneous SNRs because of multi-path fading; or, even worse, a link with lower average SNR may have higher instantaneous SNR because the normally better link may be fading. Studies in [98, 99] investigate channel dynamics in this regime and propose using algorithms to predict whether a relay's channel is going into or out of a fade.

The performance of relay selection algorithms depends critically on channel dynamics. Therefore, it is fair to question if a model captures all the behavior salient to selecting a good relay. In an ultra-reliable setting, rare events neglected in a model may result in failures a model would not predict. It is not clear if standard models such Jakes' model [44] and Rayleigh fading are useful in evaluating relay selection algorithms for URLLC.

2.2 System Prototyping

Prototyping systems with custom hardware is difficult. In contrast to the software world where prototyping is often done by combining open source software which allows the designer to focus on novel features, the hardware world has a much smaller ecosystem and much less reuse. The stark difference between the two worlds seems to be outsized relative to the fundamental differences between the two communities; hardware design often consists of similar tasks to software design. This section goes through elements of the hardware ecosystem and identifies pain points as well as potential solutions.

2.2.1 Agile Methodologies

Moore's law has been a driving force in the hardware industry for some time. With exponential improvements in both transistor cost and performance, general-purpose hardware was an attractive solution to many problems. Even if specialized hardware performed better and cost less at large volume, it might have only an 18-month window before a general-purpose part matched or improved upon it. As Moore's law is slowing/ending, general-purpose hardware enjoys a less advantageous position, and numerous specialized hardware accelerators have become widespread. The modern system on chip (SoC) integrates multiple specialized accelerators with general-purpose processing to provide a good combination of performance, energy efficiency, and programmability. As SoCs grow in functionality and see widespread use in an array of applications, the number of accelerators has grown.

Increasing transistor counts allow for increased design complexity, which may also increase time to market and non-recurring engineering (NRE) expenses. One major revolution was the advent of digital synthesis, which reflected the reality that manual design could not scale to designs of a certain complexity [85, 94, 41]. Some amount of lost performance was accepted, with the understanding that the critical portions of the design could still be hand-crafted. Since the advent of synthesis, the trend of specialization has only increased the complexity of designs, but there has not been a corresponding increase in designer productivity. The design flow has been improved over time, but still fundamentally works upon similar principles and levels of abstraction.

The waterfall model, introduced in Sec. 1.3.1, is the industry standard, and hardware tools and methodologies reflect this. The waterfall model is more challenging to apply as systems increase in complexity. This legacy also presents challenges for prototyping, which by their nature have some unknown challenges.

Agile design methodologies are a response to the waterfall model. Originally advocated for large scale software development [35], agile methodologies emphasize iteration, late changes, early and continuous delivery, and a collaborative working environment. There have been attempts to apply agile methodologies to hardware design [57], but challenges still remain. One important component of agile methodologies for hardware design is the use of hardware generators [88], which will be discussed in more detail in Sec. 2.2.4.

Agile methodologies are a natural fit for prototyping. By its very nature, a prototype has uncertainty that can be addressed well by iterating on working subsets of the desired functionality.

2.2.2 Hardware Description Languages

The popular hardware description languages (HDLs), Verilog and VHDL, were originally designed in the 1980s. Programming languages have undergone significant

changes as new approaches address shortcomings or find new ways of working. In parallel, the combined effect of Moore's law, increasing design complexity, and improving synthesis flows mean that the cost of abstractions in hardware design decreased. However, HDLs have been slow to adopt new constructs that allow for higher-level abstractions.

SystemVerilog added some useful constructs, especially those for object-oriented design, but in practice they are used more for verification rather than design. In particular, language features in SystemVerilog enable the universal verification methodology (UVM), an industry standard methodology for verifying designs in a way that promotes reuse via modular design. Support for SystemVerilog constructs in synthesis tools was slowly adopted and is still incomplete in most tools.

SystemVerilog has very limited support for metaprogramming, and is also poorly suited for the mathematics required in custom signal processing blocks. SystemVerilog is not a "productivity language" in the spirit of Python or Ruby. Packaging, modules, dependency management, etc. are poorly supported or entirely unsupported by HDLs.

There are many ways to express some constructs in HDLs, some of which may not be synthesizable or may have unexpected results (e.g. accidentally creating latches). Linters for HDLs are essential for avoiding these problems. They are also proprietary and expensive, which is problematic for open-source projects.

A design is more than the HDL description of a circuit. There is a lot of other design collateral that goes into implementing a design. This collateral can take the form of constraint files, TCL scripts, custom macros, or manual actions taken by an engineer driving a CAD tool[2]. Whatever the form it takes, tools can often produce better results when given guidance about the specific design. High-level abstractions can obscure these efforts, and this extra collateral can make the savings from using high-level abstractions smaller.

2.2.3   High-Level Synthesis

High-level synthesis (HLS) tools differs from HDLs in that a high level software language is translated to RTL. The input language can be a general-purpose programming language, often C or C+ [126, 66, 22]. Generally, only a subset of the language can be synthesized. Memory allocation, operating system interaction, and unbounded recursion are some examples of constructs that generally cannot be synthesized. They can also be domain specific languages [42, 67].

Because HLS tools translate software abstractions to hardware implementation, there are places where mismatch between those models can lead to inefficiencies [124]. HLS tools often require external guidance via directives or `#pragmas` to achieve good

---

[2]Software, firmware, package designs, and datasheets are further from the design but are also important collateral

quality of result (QoR), which limits reusability and couples the algorithm description with the implementation.

Commercial HLS tools are generally closed source and do not support plugin architectures. This lack of extensibility makes it difficult to customize compiler behavior.

2.2.4 Generator-Based Hardware Design

Like HLS, hardware generators attempt to raise the level of abstraction in hardware design. However, rather than starting with a software description that gets translated into hardware, generators directly produce RTL.

The simplest version of a hardware generator is a small standalone program that substitutes strings into an RTL template. These are easily written as one-offs, but do not compose well or lead to much flexibility or reusability across a design.

The next step up from this is inline code generation, conceptually similar to how PHP is embedded in HTML. Genesis2 [87] is an example that uses Perl as the embedded language. Similar approaches are used internally at many companies. This style of generator can be difficult to impose a strong type system or other mechanisms that enforce safety constraints.

Other generators are implemented as their own language distinct from the RTL that is ultimately emitted. Spiral [77] is focused on signal processing applications and uses its own mathematically-oriented domain specific language (DSL). It often makes sense to embed these languages as DSLs within an extant popular general-purpose language. Magma [110], PyMTL [62], MyHDL [29], Nmigen/Migen [63], are some examples in Python. Clash [17] is an example in Haskell. Simulink and MATLAB with HDL-Coder [65] can generate RTL that includes IP from many MATLAB library components. Chisel [18] and SpinalHDL [72] (a fork of Chisel) are generator languages embedded in Scala.

Chisel is used for this work. Scala is a general-purpose programming language that combines object-oriented and functional language features [70]. Chisel was initially designed with RISC-V processor generators in mind, in particular the Rocketchip generator [16].

Chisel has been used successfully to create complex hardware libraries. Diplomacy is a library from Rocketchip, a RISC-V SoC generator, that allows for general parameter negotiation, and is especially useful for generating SoC interconnect [26]. BOOM, an out-of-order RISC-V processor generator, uses Rocketchip extensively as a library [23]. Gemmini is a systolic array generator that can be easily integrated into Rocketchip SoCs [36]. Chipyard is a project that aggregates many Chisel projects and allows cores and accelerators written in Chisel (as well as other IP) to be integrated into complex SoCs [13].

Chisel's design is inspired by LLVM [56] in that there is a frontend language that

generates an intermediate representation (IR), a backend that transforms the IR in a sequence of transformations, and then a final emission stage where the IR is converted to an output form. Chisel's IR is called the Flexible Intermediate Representation for RTL (FIRRTL), and the backend is referred to as the FIRRTL compiler [43]. After a series of transformations that perform tasks such as width inference, dead code elimination, constant propagation, deduplication, etc., the FIRRTL compiler emits Verilog that can be used with any synthesis flow.

FIRRTL has a notion called annotations that attach metadata to entities in the FIRRTL graph. User-defined custom transformations can consume, modify, and add these annotations, as well as modify the IR directly. This functionality is very powerful; libraries can generate annotations that trigger custom functionality later in the implementation flow. This enables libraries to implement functionality across many levels of hierarchy. Some examples of powerful FIRRTL transformations include Chiffre [31], which generates fault injection logic; Strober [53], which generates performance counters for power modeling; and MIDAS [52] or Golden Gate [64], which transform RTL for efficient FPGA simulation.

FIRRTL also includes powerful functionality for scheduling transformations. A dependency API allows transformations to specify what transformations they expect to run before them, and which transformations they invalidate. A scheduler ensures that transformations are run in the right order and the right number of times to satisfy the constraints.

2.2.5 Verification

Verification is the bottleneck for design. Headcount, cost, and time-to-market are all dominated by verification. Improving designer productivity at the expense of verification is counterproductive to the overall design effort, and is a real danger of generator-based methodologies [61].

The bulk of verification tasks in industry are accomplished via simulation-based methods. UVM is the industry-standard design pattern by which simulation-based verification is decomposed into reusable tasks [113]. Assertions, checking for properties on outputs, and evaluation against golden models are all used to evaluate the correctness of the design under test (DUT). Coverage metrics are used to determine what fraction of the design has been evaluated by the verification suite and guide future verification efforts. Hitting complete coverage is an important milestone in the design process.

Stimulus is typically generated manually by verification engineers, but automatic stimulus generation is sometimes applicable and remains an active area of research. Stimulus can be generated randomly, but for most designs it is critical to bias the sampling such that interesting parts of the design are exercised quickly. Coverage information can be useful feedback for quickly generating "interesting" inputs [114, 108, 33].

Formal verification is a family of techniques that, in contrast to simulation-based

verification, evaluates the correctness of the DUT based on some sort of proof system, formal specification, or formal method. Model checking is a popular technique within the formal verification umbrella, which involves checking properties that are generally expressed using temporal logic. Commercially-used tools such as JasperGold must also apply a number of heuristics to designs in order to get good performance.

Formal tools are valued for the confidence they can give in the correctness of a design, but suffer from scalability issues and limited user expertise. As a result, in industry they are generally targeted at small portions of the design that are well suited to formal methods or are particularly high value.

These techniques are all applicable to generated designs, although there are some unique challenges. As discussed in Chap. 1, generated names are a commonly cited issue for Chisel-generated designs, as well as other code generation schemes[3]. Generated RTL may have different hierarchy than the code generating the RTL, or some tools flatten layers of hierarchy[4]. Generators that optimize the output RTL may change size, name, or appear and disappear without necessarily being closely related to the portion of the design being tweaked. Answering questions such as "why is the output valid signal stuck at 1" can be made more difficult to answer if the generation flow removes all the logic that resulted in a constant output. All of these issues add friction to the debug loop.

Solutions to these problems must rely on increased automation for verification and physical design. This may in turn require more expert engineers to support the automation flows as opposed to the current approaches which scale by increasing the number of verification engineers. One example of this kind of automation is generating circuit metadata along with the circuit, and having testbenches generate test harnesses and input sequences based on the metadata. IPXact [115] is a format for expressing this kind of metadata. It is used by commercial products like Cadence's Verification Workbench and JasperGold to automate test environment generation. Is has also been used with Chisel designs to generate stimulus for unit tests of generated blocks [19].

2.2.6   Physical Design

Physical design is also of critical importance to the outcome of a hardware design. QoR directly impacts performance, power, and cost, all of which are critical for a product's ability to compete in the marketplace. Even for FPGAs, getting good QoR can be important; getting a design to fit on even a large FPGA can be difficult.

Similar to verification, generated designs can increase friction for physical design flows. There have been some attempts to use machine learning for floorplanning and place-and-route [28] which may eventually lower the need for the sort of manual in-

---

[3]Beyond the ergonomic issues, this may present difficulty in iterative design processes, for example correlating with a coverage database or floorplanning.

[4]The hierarchy of the generated design may also have important physical design implications. Hierarchical flows are critical for large designs and requires careful partitioning.

tervention that generated designs make difficult. Until then, solutions that increase the ability to automate physical design flows are attractive.

Hammer [121] is a modular VLSI flow that has been integrated into the Chipyard project. The idea behind hammer is to abstract over process, technology, and EDA tools by providing APIs that ultimately generate the TCL scripts, constraints, etc. that are needed to run a flow. The ultimate goal of Hammer is that the APIs will essentially be usable like any other FIRRTL annotation and any layer of the Chisel/FIRRTL/Hammer stack can feed metadata down the implementation path.

2.2.7   Numerics in Hardware Design

HDLs are generally unfriendly to implementing math-heavy hardware. In languages that are friendly for numerics, features such as generics, numeric towers, and higher-order functions are often useful for writing reusable and clear code. HDLs generally provide +, -, $_\star$, and maybe \% operations on 2's complement integers and not much more.

In many workloads, the cost of mathematical operations may be incidental to other costs such as memory size/power, routing overhead, etc. Custom signal processing workloads are generally very math-heavy, and often rely on using as few bits as possible throughout the design to get good power, performance, and area. As a result, it is critical that designers have fine-grained control over their designs.

MathWorks's HDLCoder, mentioned earlier, is much friendlier for describing math-heavy algorithms. There are automatic float-to-fixed conversion, filter generators, and lots of useful IP. However, as a closed source flow with limited metaprogramming facilities, combining separate designs can be difficult.

HLS systems are also generally more friendly for expressing numerics. One concern for HLS systems is having fine-grained control of the output. These systems often have special types for arbitrary-width integers or fixed-point numbers that can help get better efficiency, but it can be difficult to control other important aspect of the output design.

This work makes extensive use of libraries designed to aid in designing custom signal processing hardware [120]. Details are discussed more in Chap. 6. At a high level, the libraries provide high-level abstractions for specifying numeric algorithms based on Spire, a new interval type that has tighter width-inference semantics than standard Chisel types, and a bitwidth optimization platform. Portions of this methodology have been used to successfully tape out multiple projects [19, 119] as well as to perform this work.

Chapter 3

Techniques for Achieving URLLC

This chapter discusses various techniques for achieving URLLC, especially techniques that exploit spatial diversity with relaying. Occupy CoW is the basic technique for relaying with low latency. Other techniques that add network coding and relay selection on top of Occupy CoW are also discussed.

3.1   Problem Formulation

Many of the standardization efforts for URLLC remove protocol-level sources of latency and overhead that make small packets inefficient. However, fading is generally not addressed by these measures and occurs with high enough probability to be relevant to URLLC. As a result, even if the average and median latency are improved, the tail behavior will still be dominated by fading as illustrated in Fig. 3.1.

As discussed in Sec. 2.1.3, exploiting spatial diversity via relaying is an attractive technique for overcoming deep fades. The remainder of this chapter outlines several approaches for relaying in the URLLC context that we have investigated.

3.2   Occupy CoW

Occupy CoW (Control over Wireless) employs simultaneous relaying to achieve reliability. "Simultaneous relaying" refers to the fact that many relays may share the same information (e.g. the uplink packet for user 6) and can therefore use a distributed space-time code to allow multiple transmitters to utilize the same transmission resource (e.g. timeslot). A full treatment is given in [105], but a general overview is provided here.

Latency PDF

Time
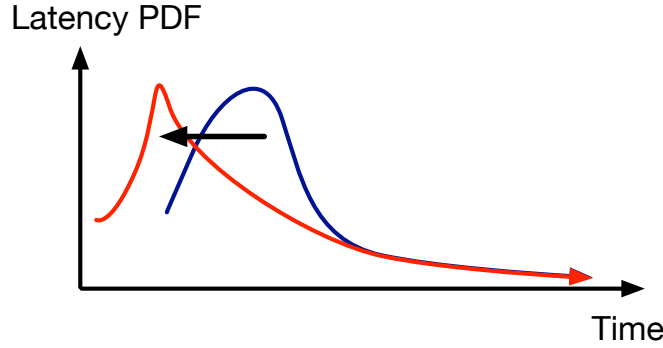
Figure 3.1. Many of the standardization efforts for URLLC address sources of latency from overhead in the protocol, but do not address the sources of latency that come about from poor channel conditions.

3.2.1 Protocol Description

We assume each user generates one fixed-size packet for the controller per cycle, and receives one packet from the controller of the same size every cycle, i.e. the information flows in a star topology. There is a variant in [105] for generic information topologies, but for the sake of clarity we will describe the simpler star-topology variant. The protocol is scheduled and known to all users in the network in advance and each packet has a dedicated timeslot.

Occupy CoW consists of three main stages, each of which have their own dedicated timeslots:

1. **Downlink:** The controller broadcasts every controller-to-actuator packet. Every user attempts to receive the entire packet (instead of going to sleep, for example). Each user needs to receive the controller-to-actuator packet intended for them, but also tries to receive every other packet in the event that they are needed to act as a relay.

2. **Uplink:** The uplink portion is divided into timeslots for each user. During their timeslot, a user broadcasts its sensor-to-controller packet. The controller attempts to receive the message, but all other users in the network try to receive the packet in the event that they are needed to act as a relay.

3. **Relay:** The relay portion is divided into timeslots for each user, and each user's timeslot is further divided into a timeslot for the controller-to-actuator packet and another for the sensor-to-controller packet. During the relay timeslot for user i, every user who successfully received a controller-to-actuator or sensor-to-controller packet for user i will transmit it in the appropriate timeslot. There may be multiple relay phases to allow multi-hop relaying Having one relaying phase will be referred to as "two-hop" because the transmission took one hop for the relay to receive it and another hop for the relay to transmit it. Having two relaying phases will be referred to as "three-hop".

28

There are two scheduling variants: a fixed relaying schedule variant and an adaptive relaying schedule variant. The fixed variant schedules a relay timeslot unconditionally for every user, even if their transmission succeeded in the first two phases. The adaptive variant only assigns timeslots to users who had a failed transmission in the uplink phase. This requires the controller to assess which users had failed transmissions and an extra timeslot for the controller to broadcast a schedule for the relay phase. This scheduling phase occurs between the uplink and relaying phases. The full time in the relaying phase is always utilized; the fewer the number of users that need to be relayed, the lower-rate the transmission is.

The description above is illustrated in Fig. 3.2 for a 10-user example with the adaptive variant.
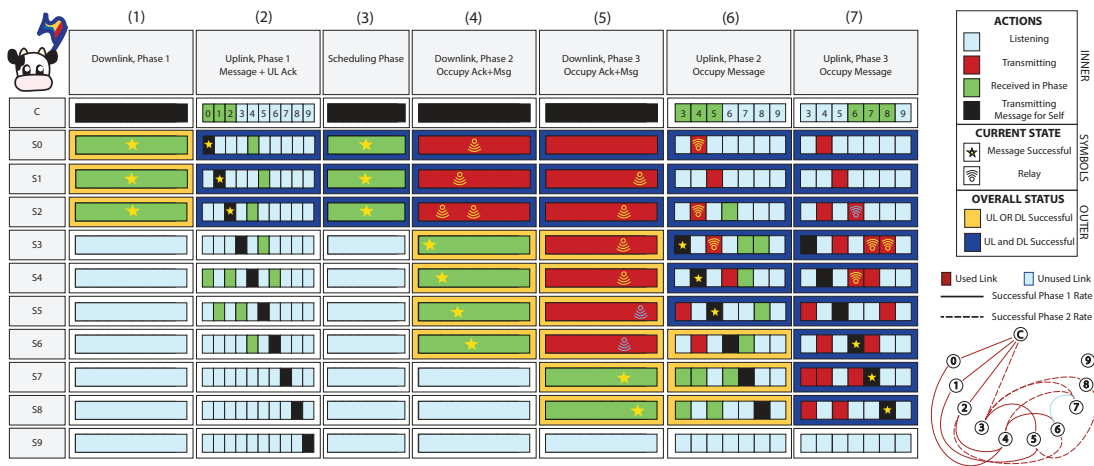
Figure 3.2. Illustration of Occupy CoW, adaptive variant. The bottom right corner depicts the connectivity of the users; some have sufficient SNR to succeed in phase 1 and others only have sufficient SNR to succeed in phase 2. Each column moving from right to left shows the actions taken by users in the network at each phase. Each row is a different user, starting with the controller followed by the 10 users S0 through S9. Each rectangle has a color depicting the actions being taken, symbols indicating if the user succeeded receiving the message or is relaying, and an outline indicating the status of the user's uplink and downlink messages. This figure originally appeared in [102].

Another way in which Occupy CoW schemes can be varied is by phase length. The canonical version of Occupy CoW assigns the same amount of time for the Downlink, Uplink, and Relay phases. However, it is reasonable to optimize the length of each of these phases for reliability and such schemes are also considered.

3.2.2 Performance

The work in [105] produced analytical expressions for the probability of cycle error for various Occupy CoW schemes under certain conditions. These conditions are:

- All channels are assumed to have the same fixed nominal SNR and are assumed to have independent Rayleigh fading. Each user is assumed to have perfect knowledge of the channel state information (CSI) of its own good channels, but are not assumed to have any knowledge of its own faded channels or any other user's channels.

- Channels are assumed to have a single tap.

- Each link is "good" (i.e. information is ultimately received with no errors or erasures) if the rate of transmission is less than or equal to the channel capacity.

- Channels are assumed to be reciprocal.

- Sender diversity is harvested perfectly; that is, if there are k simultaneous transmitters, then this is the same as there being k independent attempts for communicating the mession.

- Implementation effects for coding are not considered and no transmission or decoding errors are undetected.

Under these assumptions, analytical expressions are derived and some interesting observations can be made. Fig. 3.3 shows a comparison of some Occupy CoW schemes with some alternative schemes.

The one-hop scheme is a scheme with no relaying; only the downlink and uplink phases are used with relaying phase. No spatial diversity is exploited, so overcoming fades requires potentially huge SNRs in the 80 dB to 100 dB range. Even with a genie that optimally allocates channel resources to the users in deep fades (called '1 hop with genie-aided HARQ' in the plot), 1 hop schemes require very high SNRs. Coding is not very effective at overcoming deep fades.

The 'frequency hopping repetition code' line in Fig. 3.3 shows the effectiveness of exploiting frequency diversity instead of spatial diversity. The repetition code refers to the idea that failed transmissions are retried at a different, independently-faded carrier frequency potentially many times. The annotated numbers reference the minimal number of independently-faded carrier frequencies required to achieve the reliability spec at the given SNR. One thing to note is that the number of frequencies is fairly large; it may be unrealistic to expect the environment to natively have 20 carrier frequencies that fade independently. Furthermore, the frequency hopping scheme scales poorly in comparison to the Occupy CoW schemes because the SNR requirement increases with network size.

Once the Occupy CoW schemes reach approximately 7 dB, there is enough spatial diversity for the Occupy CoW scheme to perform better than the frequency hopping
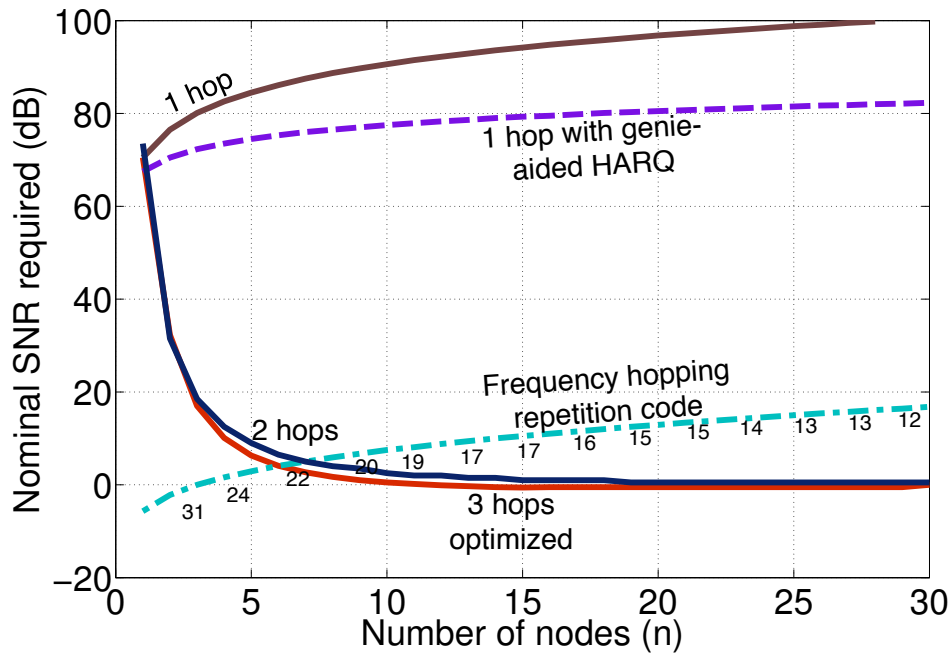
Figure 3.3. Performance of Occupy CoW. The x-axis represents the network size and the y-axis represents the minimum SNR required to meet the latency and reliability requirements. Lower SNR is better. The 1 hop lines represent schemes with no relaying and show the SNR required to overcome fades directly. The frequency hopping line represents a scheme where users can retry failed transmissions at a new, independently faded carrier frequency. The 2 hops and 3 hops optimized lines represent Occupy CoW schemes. This figure originally appeared in [102].

scheme. Furthermore, the Occupy CoW schemes scale much better with network size. The two lines represent two different variants of the Occupy CoW scheme: an unoptimized 2-hop scheme and a phase-length optimized 3-hop scheme. Optimizing phase lengths and using 3 hops does provide some benefit, but the SNR savings are relatively small. Simplicity of implementation may favor the 2-hop scheme because of the small difference.

3.2.3   Summary

Occupy CoW is a relatively simple scheme that performs well for URLLC scenarios under some assumptions. It exploits the rich supply of spatial diversity available in a network with a large number of nodes without a large amount of overhead. The practicality of Occupy CoW depends largely on how realistic the modeling assumptions are, and how sensitive it is to modeling error.

3.3   XOR-CoW

Network coding encompasses a large category of coding techniques wherein performing coding across multiple nodes in a network can achieve better performance than coding along a single link as if it were a point-to-point connection. The famous examples in [10] demonstrate that network coding can be applied to great effect when the information structure supports it. XORs in the Air [50] is a simple but powerful application of network coding to relaying systems. We apply this technique to Occupy CoW in a set of schemes we call XOR-CoW [101], pictured in Fig. 3.4.
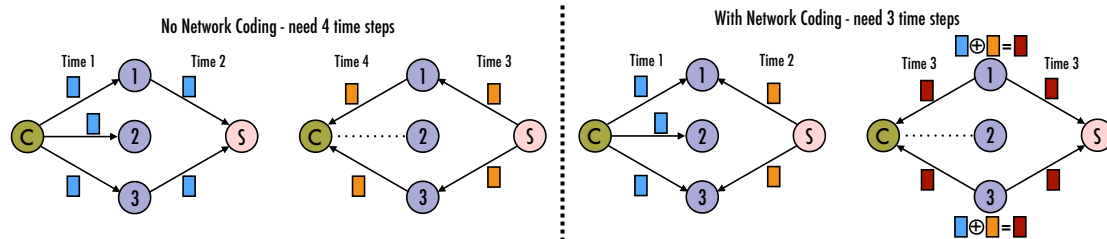
Figure 3.4. Illustration network coding, in particular the XORs in the Air scheme. The left depicts using relays with no network coding. Because the $C \rightarrow S$ messages and $S \rightarrow C$ messages are performed separately, it takes 4 time steps total. The right depicts using relays with network coding, namely by having the relays XOR the $C \rightarrow S$ and $S \rightarrow C$ messages. This allows the two relay time steps to be merged, and the entire scheme takes 3 time steps instead of 4. This figure originally appeared in [101].

The idea is that the downlink-relay and uplink-relay phases can be fused. Simply XORing the downlink and uplink packets is sufficient; the controller can remember the downlink packet and XOR it with the relay packet to recover the uplink packet, and the sensor node can similarly recover the downlink packet. Fusing the two relay phases into one phase provides more channel resources per bit of information and thus reduces the SNR required to meet URLLC constraints.
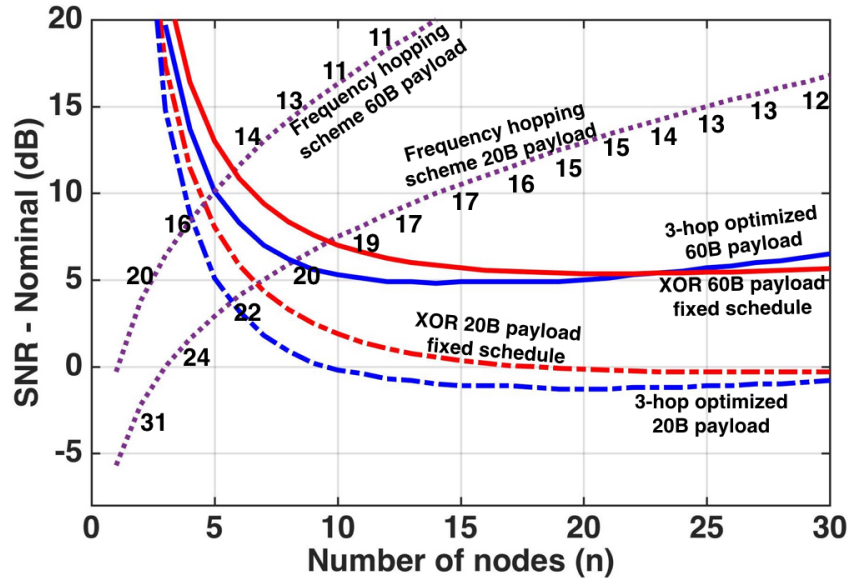
Figure 3.5. Performance of XOR-CoW. Performance with 20-byte and 60-byte packets is shown. Frequency hopping, 3-hop optimized Occupy CoW, and XOR-CoW are compared. 3-hop optimized Occupy CoW and XOR-CoW perform similarly. This figure originally appeared in [101].

This works best when there is one relay phase as the XORing can only be applied to the last relay phase. Therefore, it is useful to compare the 2-hop XOR-CoW scheme to the 3-hop Occupy CoW scheme, shown in Fig. 3.5 for 20 and 60 byte payloads. Interestingly, for smaller network sizes optimized 3-hop schemes are slightly better, but only slightly. Furthermore, after the network size is large enough the XOR-CoW scheme performs better.

The relative simplicity of the XOR-CoW scheme, as well as the potentially better performance for larger networks, makes it an attractive option for a practical URLLC system. Of course, this performance is somewhat specific to scenarios where the uplink and downlink packet sizes are equal. In more general scenarios where information flows are non-symmetric, XOR-CoW may be less favorable.

33

3.3.1 Robustness

Work in [97] analyzes the sensitivity of Occupy CoW and XOR-CoW schemes to modeling error. Non-reciprocal channels, non-quasi-static channels, imperfect diversity harvesting, and non-independence are all studied. Non-reciprocal and non-quasi-static channels have a relatively small impact. The impact of imperfectly harvesting diversity depends on just how much diversity can be harvested, but the most important modeling assumption is independence. When channels are too likely to fade dependently, the SNR requirements skyrocket. This is unsurprising as the entire basis of these techniques is to exploit the independence provided by spatial diversity.

It is critical to understand the sensitivity of these techniques to modeling error when building prototype systems.

3.4 Relay Selection

After independence, the next most important modeling assumption behind the schemes presented here is the assumption of being able to harvest full sender diversity. There are several reasons to think this may not be true in implementations. After a certain number of relays, it seems reasonable that synchronization error, phase noise, channel effects, or other phenomena may cause an extra relay to add more noise than signal.

One way to achieve reliable communication without requiring simultaneous transmissions to achieve full sender diversity is to use a smaller number of "good" relays. Studies in [98, 100, 96] show how users can used observed channel information to reliably choose relays that will have good channels to both the sensor node and the controller. These techniques will be addressed in Chap. 5.

3.5 Summary

This chapter outlined techniques for achieving URLLC. They utilized spatial diversity via relaying to overcome fading channels and achieve high reliability. Network coding and relay selection techniques are useful refinements to these ideas. The next chapter will discuss implementing these ideas, in particular the PHY and MAC requirements.

Chapter 4

PHY and MAC Considerations for Relay-Based URLLC

The techniques discussed in Chapter 3 abstract away many PHY and MAC implementation details. These abstractions are valuable when exploring new ideas and getting a sense of what circumstances are needed to achieve the requirements, but ultimately it is important that an implementation have an PHY and MAC that is practical to realize.

There are some ideas from Chapter 3 that guides the design of a PHY and MAC. One idea is that transmissions should be scheduled in a time-divided fashion. Another idea is that relays will be transmitting the same message, which allows for the possibility that relays can transmit simultaneously. It is important that these relay transmissions do not interfere with each other.

Besides these things, there is a lot of freedom in how to implement a transceiver for relay-based URLLC. In the cases where there is freedom to make arbitrary choices about the design of the PHY or MAC, we choose to make similar choices to those in the 802.11 standard. This is not necessarily because 802.11 has made the most optimal choices. Rather, it is because 802.11 has seen widespread deployment and usage in consumer and industry spaces. It is useful to frame these design decisions like so: "What changes must be made to 802.11 in order to support URLLC?"[1]

---

[1]It might seem more natural to use 5G systems, especially the URLLC proposals, than 802.11. To the extent that 5G allows for new numerologies, it could be argued that for this work there is not a substantial difference between 802.11 and 5G. However, URLLC proposals in 5G add significant complexity to allow for different numerologies and low latency scheduling. It is this author's judgement that 802.11 is a simpler baseline system with a fewer changes needed to support relay-based URLLC.

4.1 OFDM
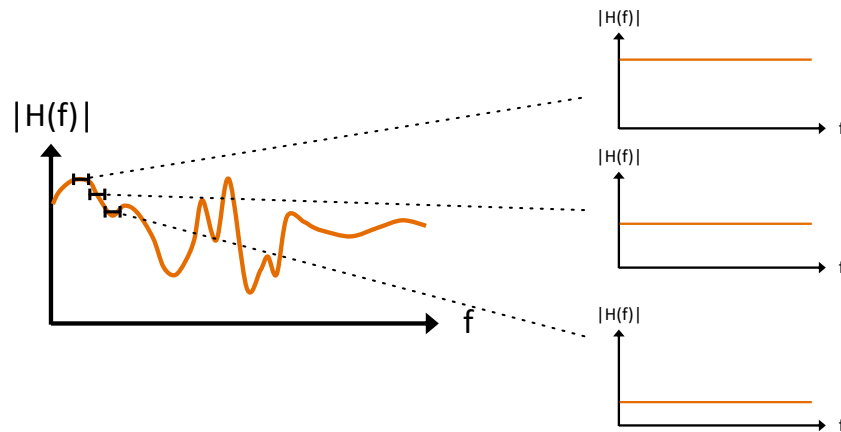
Figure 4.1. OFDM allows for simple channel estimation and equalization for wideband channels. A frequency selective wideband channel gets convolved with the input, and after the FFT in OFDM each subcarrier will correspond to a small portion of the channel. If N_{FFT} is chosen large enough, these small portions of the channel can be assumed to be approximately constant.

Orthogonal frequency-division multiplexing (OFDM) is a common scheme for encoding data for transmission over a channel. In contrast to single-carrier modulation schemes, which directly modulate a signal to the full available bandwidth, OFDM maps data to multiple subcarriers. OFDM does this by utilizing the fast-Fourier transform (FFT), which can be thought of as dividing a single wideband channel into a number of parallel narrowband channels as depicted in Fig. 4.1. Channel estimation and equalization can be easier for these narrowband channels.

Channel estimation can be performed in many ways with OFDM. The time-domain estimation techniques used for single-carrier systems can be used with OFDM, and equalization can be performed in either the time or frequency domain. However, it is more typical to see channel estimation and equalization performed in the frequency domain. Generally, the FFT size for an OFDM system is chosen to be large enough that each subcarrier can be thought of as a narrowband channel (that is, a single-tap channel). The subcarrier spacing, given by BW/N_{FFT}, is an important parameter. N_{FFT} can be chosen such that the subcarrier spacing is significantly smaller than the coherence bandwidth, which is a measure of how variable the channel is. If this condition is true, each subcarrier can be thought of as a single-tap channel, which greatly simplifies the complexity of channel estimation. N_{FFT} can be increased to accommodate a wider channel, and the FFT, channel estimation, and channel equalization can scale well to accommodate a more complex channel. This feature is one of the main reasons OFDM is popular.

Preamble-based estimation schemes, depicted in Fig. 4.2, involve transmitting at
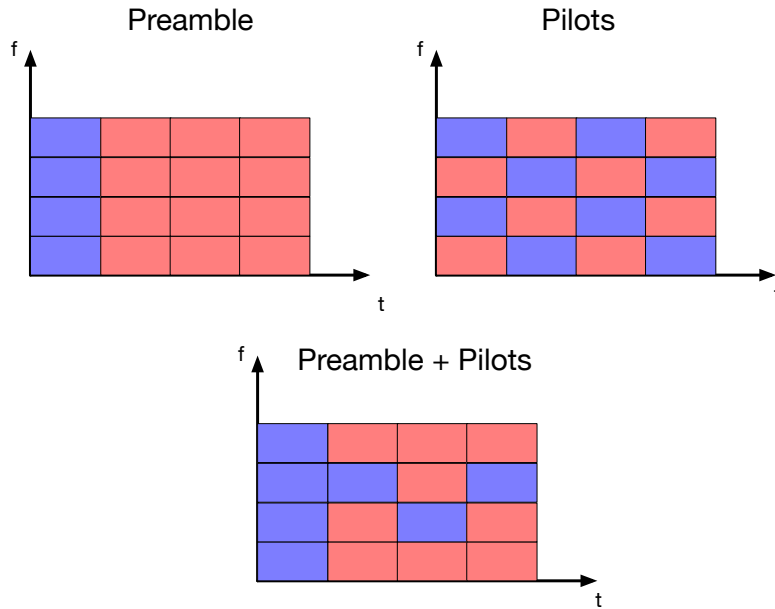
Figure 4.2. Representation of how subcarriers can be organized into pilots and data, shown on the time-frequency plane. Preamble-based schemes, depicted on the top left, have pilots at the beginning of a transmission and then all data after. Pilot-based schemes always have data and pilots intermixed, as shown on the top right. Preambles and pilots can be used together, as shown on the bottom.

least one known OFDM symbol before transmitting data. The receiver will see a value on each subcarrier i given by $y_i = h_i x_i + n_i$. In the high-SNR case, a simple division is all that is needed to estimate h_i for each subcarrier. Knowledge about $\vec{h}$ and $\vec{n}$, for example that the noise is colored or that the channel has at most 8 taps, may allow for more ways to refine estimates of h_i. Generally, though, estimating h_i is parallelizable and requires at most a division, which comes with some hardware expense but is often cheaper than an equivalent time-domain based equalizer.

Pilot-based estimation schemes intermix known data, called pilots, with the data to transmit that is unknown to the receiver. The estimation problem is similar to the preamble case, but because the pilots are interleaved with data, some subcarriers' channels may need to be inferred from the channel estimates of nearby carriers. There are many possible interpolation schemes that interpolate channel estimates across time and frequency. These interpolation schemes, as well as choosing the number of pilots to use in a symbol, require knowledge about the channel complexity and dynamics as well as application requirements.

Preamble- and pilot-based estimation schemes can be used together. The preamble can give a good initial estimate and then pilots can track changes to the channel over the course of a long message. OFDM provides very simple and flexible tools for performing channel estimation and equalization that can be tailored to specific channel and application conditions.

Channel equalization in OFDM is generally performed by a single complex multiply by the inverse of the channel estimate.

OFDM is a popular choice for many widely deployed standards including 4G, 5G, 802.11, DSL, and cable modems. The cost of an FFT is often a good tradeoff for the ease of channel equalization and high-order modulation it enables. OFDM was selected for this project because of these properties. Where possible, the numerology (FFT size, size of cyclic prefix, etc.) was kept consistent with 802.11.

The choice of preamble- or prefix-based channel estimation is an important decision. For the uplink and relaying phase, the overhead of having a preamble for each user is hard to justify. For 30 users with $N_{FFT} = 64$ (given by 802.11G), almost 10% of a 2ms cycle time would be consumed by preambles alone.

4.2 Synchronization

Synchronization is of critical importance for any scheduled MAC. In the schemes discussed here, there will be downlink, uplink, and relay phases. The downlink phase consists of a one-to-many transmission, and therefore does not need to be synchronized with other users particularly well. The uplink and relay phases, however, consist of many small packets being transmitted by different users. Each transmission will have a timeslot scheduled, and poor synchronization will result in missing the timeslot, failing to send the desired data, and potentially interfering with some other transmission.

Furthermore, some of these schemes depend on scheduling simultaneous relay transmission. This occurs when multiple relays all share the same information and transmit it at the same time. The problem with poor synchronization is the same as for the uplink case with the added danger of poor frequency synchronization, as illustrated in Fig. 4.3. If there is a large carrier-frequency offset (CFO) between two transmitters, they may interfere with each other and ruin the transmission. As a result, it is important that users be well synchronized in both time and frequency[2].

Estimation of the CFO can be done in multiple ways. In OFDM, a blind CFO estimation can be performed by taking advantage of the fact that the cyclic prefix of a symbol should be the same as the end of the symbol; finding the phase shift between the CP and end of the symbol can be used to find the CFO. This can be relatively slow to converge because of short CPs and short OFDM symbols resulting in noisy estimates. Preambles can also be used to estimate CFO, at the expense of adding the overhead of a preamble that blind estimation does not require.

Adding a preamble to the controller packet and performing CFO correction at the transmit side in every user allows CFO estimation to be performed accurately based on a

---

[2]There are techniques for synchronizing networks in a scalable way, for example [38]. However, using these techniques in the URLLC context is challenging as they add overhead and achieving highly-reliable synchronization is still a challenge.
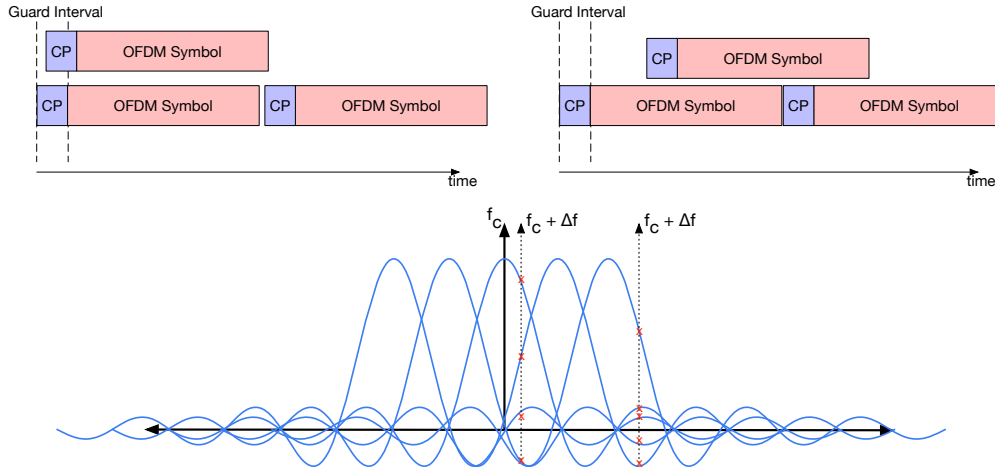
Figure 4.3. Time and frequency synchronization are critical to URLLC systems. The top figures depict good time synchronization on the left, where two simultaneous transmitters are within a guard interval (determined by the CP), and bad time synchronization on the right, where one of the simultaneous transmitters is out of the guard interval and clobbers the subsequent transmission. The bottom figure depicts how CFO results in inter-carrier interference (ICI). A single transmitter's CFO can be corrected at the receiver, but multiple transmitters with CFOs that are too large will degrade SNR significantly.

preamble. This gives a better CFO estimate than blind estimation at reasonable overhead (i.e., the overhead does not increase as the number of users increases).

4.2.1   Synchronization in OFDM

OFDM has a standard technique for proscribing the required time synchronization accuracy. Each OFDM symbol can have a cyclic prefix (CP) of a parameterized length. It is called a cyclic prefix because the samples of the prefix come from the end of the OFDM symbol, making the combined CP + symbol look periodic. The beginning of the OFDM symbol can be taken as anywhere within the cyclic prefix and because of the cyclic shift property of the FFT, it will only look like a linear phase shift after the FFT. The CP can be chosen to be any size, but ideally should be chosen to be small relative to the size of the OFDM symbol length N_{FFT} to avoid large overhead.

In an OFDM system, frequency synchronization is usually performed by correcting the CFO at the receiver before taking the FFT. This is because the correction, a multiplication by a complex exponential with the appropriate frequency, is fairly cheap in the time domain. In the frequency domain, this correction becomes a convolution.

However, in the case of simultaneous transmissions, each transmitter may have different CFOs relative to the controller and each other. Different relative motions between

users will mean that the Doppler shifts will be different. This makes correcting CFO very difficult at the receiver.

A simpler approach is to try to correct CFO at the transmitter. Each user will estimate their CFO relative to the controller, and all future transmissions will be multiplied by a complex exponential to match the carrier frequency of the controller. If this estimation is performed well, the remaining offsets between users in the network will come from Doppler shift. Even if perfect knowledge of all other users in the network is available to every user (which is generally impractical), it may not be possible to find an offset that results in zero perceived CFO between receivers.

Therefore, it is generally preferable to choose system parameters that make Doppler shifts small enough to be negligible. In the case of OFDM, the rule of thumb is generally that the remaining CFO after correction should be less than 1% of the subcarrier spacing. If we take channel bandwidth and carrier as given, this creates a relation between the Doppler frequency (determined by the maximum velocity of scatterers and users) and the subcarrier spacing (determined by N_{FFT}). Thus, we see there is some tension here between time and frequency synchronization. Decreasing N_{FFT} makes the frequency synchronization easier by increasing subcarrier spacing, but makes the overhead of a cyclic prefix more expensive.

For the parameters generally used in this work, say $f_C = 2.4\,\mathrm{GHz}$, BW = 20 MHz, and $N_{FFT} = 64$, subcarrier spacing is 312 kHz and the Doppler shift is 3.12 kHz when the maximum velocity is 390 m/s. This is rather large, so for these parameters Doppler shift is not an important contributor to CFO. However, for LTE-inspired parameters BW = 30.72 MHz, and $N_{FFT} = 2048$ (keeping $f_C = 700\,\mathrm{MHz}$), subcarrier spacing is 15 kHz and the corresponding maximum velocity is 18.75 m/s, which is quite a bit slower and much more likely to be problematic[3].

In contrast to the simultaneous-relay transmission case, time-dividing relay transmissions does not require correcting CFO at the transmit side. However, if the parameters are chosen such that the Doppler shift is small relative to the subcarrier spacing, it may be beneficial to do so. This is because users will be getting more information from the controller than any one individual user.

---

[3]The tradeoff between time and frequency synchronization is not entirely real in an important sense that this example is ignoring. For URLLC, there may not be enough data to meaningfully use large FFT sizes. If there are only 20 bytes being transmitted, having $N_{FFT} = 2048$ is probably not making good use of the channel because there are too many subcarriers in one symbol. In this case, increasing FFT size to amortize a large cyclic prefix is not necessarily going to be helpful.

4.3 Spatial Coding with Simultaneous Relay Transmissions

Simultaneous relay transmissions are motivated by the idea of many users transmitting the same information simultaneously to increase the reliability of the transmission. It is analogous to a crowd of people shouting a sentence to make sure a nearby person gets the message. Like the analogy of the crowd, it is possible that the message will get lost if the transmitters interfere with each other. Previous sections have discussed the consequences of poor synchronization, but another possible failure mode is unlucky channel realizations that result in transmitters destructively interfering. This phenomenon is essentially the same as the fading this technique is trying to overcome.

Spatial codes can be used to ensure that this kind of interference does not occur with too high a probability. Spatial codes describe how a group of transmitters should transmit together to achieve some goal. MIMO with independent spatial streams is one form of spatial coding which has the goal of increasing throughput. Spatial codes can also be used to increase reliability; in fact, throughput and reliability exist in a well-studied tradeoff[129].
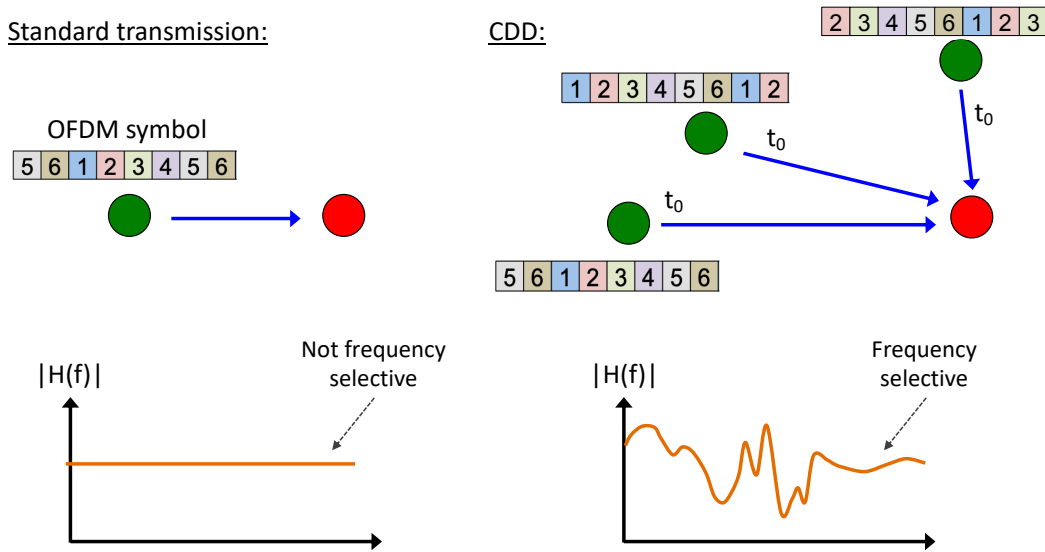
Figure 4.4. Cyclic Delay Diversity (CDD) applies random cyclic shifts to different transmitters OFDM symbols. This prevents relays from destructively interfering and allows simultaneous relays to exploit spatial diversity.

Cyclic delay diversity (CDD) is a simple space-time code that is easily applied to help OFDM systems to exploit diversity [21]. It has been proposed as a way for relay-based OFDM systems to avoid interfering with each other and exploit spatial diversity in [125]. The idea is to perform a random (per-transmitter) cyclic shift on an OFDM symbol before performing the IFFT. The same cyclic properties of the DFT that the CP

relies on mean that the randomly adding cyclic shift at the transmitter looks like a channel with a random delay. This decorrelates the channel responses of different transmitters without requiring significant changes to the architecture of OFDM transceivers. This prevents relay transmitters from destructively interfering, effectively manifesting spatial diversity as a more frequency selective channel. This is depicted in Fig. 4.4.

4.4 Channel Estimation

Channel estimation is difficult for the relay-based URLLC proposed here. Using numerous relays to increase reliability necessitates estimating numerous channels, but meeting low latency requirements means that this estimation must come with relatively low overhead.

The typical way channel estimation is done for an OFDM based system was discussed earlier in Sec. 4.1. This involves using either preambles or pilots to send pre-agreed symbols on some subcarriers. Using preambles results in very large overhead for the relays which have very small packets to transmit. Therefore, it makes sense to use pilot tones for the relay transmissions. However, the controller is sending data for every user, and has enough data to amortize a preamble. The added benefits of a preamble for CFO estimation and synchronization, as well as being consistent with how 802.11 is defined, makes using a preamble for the controller a good choice. This design uses a preamble for the downlink transmissions, selected to be the L-STF preamble signal from 802.11 [74].

4.4.1 Simultaneous Relay Transmissions

Under the simultaneous relay transmission scenario, it is possible that most users in the network will transmit simultaneously. This has the consequence that that channel may be significantly more complex than a point-to-point channel; that is, there may be more degrees of freedom in the channel. In fact, CDD by design increases the complexity of the channel by trying to decorrelate the channel responses.

Increased channel complexity means that more resources, be they preambles or pilots, must be dedicated to channel estimation. This increases the overhead of communication, especially for short packets that cannot amortize the time spent estimating the channel. Determining precisely how many pilots are needed is challenging because it depends on the environment in a fairly complex way. The number of relays transmitting simultaneously, the correlation of the channels, and the complexity of a point-to-point channel factor into how many pilots are needed. It is expected that most potential relays will have good channels, so the worst case (a very complex channel) is also the common case. However, if the number of users is large relative to N_{FFT}, or if the underlying channel is of relativeley high complexity, the number of pilots needed to estimate the channel may

be prohibitively expensive, perhaps even more expensive enough that using a preamble is preferable.

Further compounding these issues, each relay transmission can potentially have different relays participating. This makes it difficult to reuse channel estimates over time, which would be one way of potentially reducing the number of pilots needed.

For the parameterization chosen for this design, the number of degrees of freedom in virtual channel that results from the simultaneous transmissions may be greater than or equal to the number of subcarriers N_{FFT}. With $N_{FFT} = 64$ and the number of users set to 30, each user having a 2-tap channel will make using a preamble preferable to pilots, even for very short transmissions. In different systems with a smaller number of users or a larger amount of data per user may change this relationship. For this system, every relay transmission consists of 3 OFDM symbols, so an extra symbol for a preamble adds considerable overhead. This high overhead makes the extra cost of time-multiplexing a small number of relays less drastic, and the simplicity of time-multiplexing relays is very appealing.

4.5   Channel Coding

Channel coding is also of critical importance for relay-based URLLC. In throughput-oriented scenarios like those contemplated by 802.11 or LTE, the goal is to close the gap to capacity to be as small as possible. Techniques like coding over large blocks (e.g. 1024 or 2048 bits) and precise rate-matching via hybrid ARQ (HARQ) help achieve that goal. However, in the URLLC scenarios discussed here, packets are too small to allow for such large block sizes and latency requirements make feedback techniques like HARQ difficult. Therefore, short-block length performance is an important consideration for URLLC.

The Polyanskiy bound characterizes the relationship between coding rate and error probability for finite block-length codes [75]. The study in [123] uses this bound that coding schemes are able to achieve the URLLC requirements considered here, but does not make specific recommendations as to a specific code that will achieve the requirements. The architecture in [123] uses long, potentially very low rate retransmissions[4]. Repeat-accumulate (RA) and accumulate-repeat-accumulate (ARA) codes are convenient codes when low rate is desired, which for ultra-high reliability could be desirable. Similarly, fountain codes provide a straightforward way of generating precisely as many bits as are needed and no more. However, in the case of a URLCC system, real-time

---

[4]Having a large pool of relays, which was not considered in [123], makes low rate transmissions unnecessary. As long as all users have similar nominal SNRs, having enough relays ensures that at least one of them will have a good channel.

feedback about the reception of a transmission is expensive[5]. Instead, it is important to pick codes that perform well for a short packet length.

Studies in [89] present codes that perform well for short packet lengths. Tail-biting convolutional codes are competitive in this regime. Convolutional codes are already defined in the 802.11 standard and decoders are simple to implement, so are a good choice for this work. For development purposes, a simple rate $1/2$ code with constraint length $K = 2$ and generator polynomial $[3, 1]$, but a higher performance implementation should use one of the codes with longer constraint length evaluated in [89].

4.6   Summary
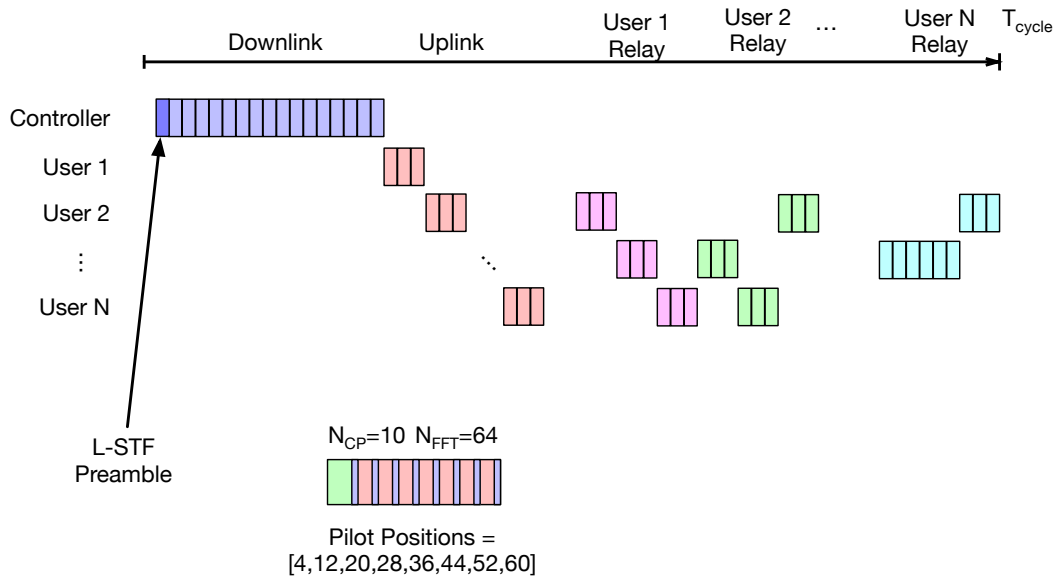
Figure 4.5. Description of communication scheme for URLLC. There are three phases within a cycle: downlink, uplink, and relay. In the downlink, the controller sends a preamble followed by data symbols containing actuation commands for each user. In the uplink, each user takes turns transmitting. The relay phase consists of several sub-phases, each one where relays retransmit data for a different user.

This chapter outlines the rationale behind design decisions for an OFDM-based URLLC communication scheme. These decisions are summarized in Fig. 4.5.

---

[5]Here perhaps is an opportunity for a full-duplex system to make a good impact. Full duplex systems allow for very fast feedback paths, allowing a receiver to tell a transmitter that the transmission has been successfully received with very little delay. This might favor usage of fountain codes or low-rate codes with HARQ.

Chapter 5

Predicting Relay Quality

Simultaneous relaying can achieve low latency, but has implementation challenges as discussed earlier. Instead, using a subset of possible relays was proposed because it is more implementation friendly. Fewer relays simplifies synchronization and channel equalization. If a small enough subset is used, the overhead of time-dividing the relays can be an acceptable trade for its simplicity of implementation.

Of critical importance is the number of relays used. If the subset of relays used is large, then there is little improvement over using them all, and possibly increased overhead from coordinating them. If the subset of relays used is too small, the system may not be able to achieve the reliability requirement.

If a user could know a priori which relays would have faded channels and which would not, then only one relay would be needed to achieve the reliability requirement. In reality, users must contend with the future channel state of the network being uncertain. A relay that has a high-quality channel during a user's uplink phase may go into a fade during the user's relay phase, as depicted in Fig. 5.1. Channel dynamics dictate the ability of users to predict future channel state, and therefore determine the minimum number of relays needed to achieve the reliability requirement.
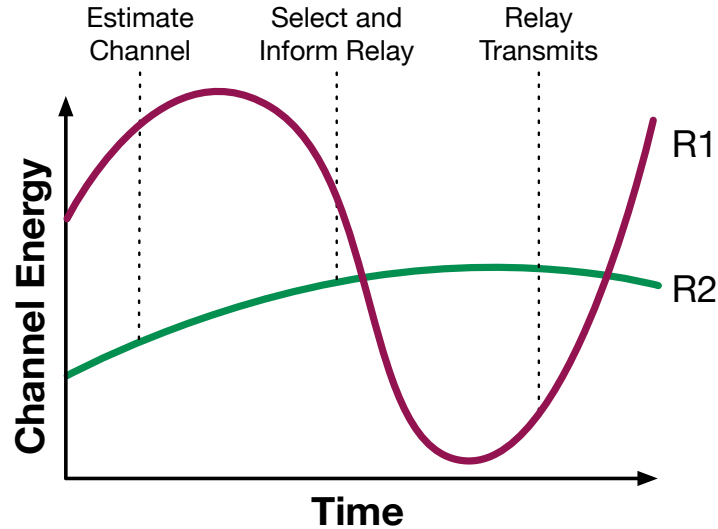
Figure 5.1. Given two potential relays, *R*1 and *R*2, determining which will be the best future relay depends on channel dynamics. Between the time that the channel is estimated, the relay is selected and informed, and the actual relay transmission occurs, the channel state *R*1 may have changed considerably. The best relay at the time of channel estimation (*R*1) may not be the best when the transmission is scheduled (*R*2). Channel dynamics must be accounted for to do a good job with relay selection. This figure originally appeared in [83].

5.1 Channel Models

5.1.1 Coherence Time

Coherence time is a useful abstraction for understanding wireless systems. The basic idea of coherence time is that when a channel does not change instantaneously[1], it is reasonable to approximate the channel as being piecewise-constant; that is, for a short interval of time determined by the coherence bandwidth the channel does not change[2]. This approximation is useful; for example, the idea of a preamble followed by data relies on the premise that the channel does not change significantly in the interim. This kind of approximation is not uncommon in URLLC work, for example [49, 59] which both assume channels are static throughout a cycle.

Making a piecewise-constant approximation can hide important details in some circumstances. One example is in finding zeros, in which case you have a (potentially poor) approximation of the location of the zero, or could even entirely miss a zero.

---

[1]In this case, the fading process is assumed to be bandlimited. Moving objects in the environment are assumed to have bounded speed.

[2]This is analogous to the sampling theorem. If the channel variation is band-limited, it can be represented and reconstructed with its values at discrete times. Of course, the channel is not actually constant between these intervals, but it is a useful approximation in some contexts.

Finding zeros is closely related to finding which channels cross the decodability threshold. For the purposes of relay selection, it is important to know that even though a channel is above the decodability threshold at the start of a cycle, it may not stay that way throughout the entire cycle. By the sampling theorem, as long as the channel is being measured frequently enough, it can be reconstructed. It is theoretically possible that a relay could be in fade for less than a cycle and not to be able to see it for an arbitrary amount of time.

Relay selection techniques for URLLC require a more nuanced notion of coherence time. Rather than making a piecewise-constant approximation, we are instead interested in time horizons over which we can make probabilistic statements about future events given the past channel state. Given an event $E(t + \tau)$ at some future time τ, we say that τ is the time horizon over which the estimator $\hat{E}$ can predict E with reliability $(1 - p)$ if:

$$\Pr\left[\mathbb{1}\{\hat{E}(t + \tau)\} \neq \mathbb{1}\{E(t + \tau)\}|h(t)\right] \leq p.$$

If, for example, $E(t) = \{\|h(t)\|^2 > x\}$ for some decoding threshold x, then we interpret τ as the time horizon over which we can predict if $h(t)$ will be faded. In the case of relaying, we are actually interested in two channels: the channel between the user to the relay and the channel between the relay and the controller. In that case, $E(t) = \{\min(\|h_{UR}\|, \|h_{CR}\|) > x\}$ for some decoding threshold x.

This notion of time horizon is tailored to an application in a way that coherence time is not. Of course, this makes the time horizon less generally applicable, but it is much more useful in making precise statements for system design.

5.1.2 Channel Dynamics

We wish to analyze channel models with this more application-driven notion of coherence time in mind. Rayleigh fading is a useful model to start with as a lower bound; having no line-of-sight (LoS) path in an indoor environment increases the probability of being in a fade relative to fade distributions with a LoS component (for example, Rician fading). Authors in [9] show that channel dynamics are similar in urban and suburban environments for Rayleigh, Rician, and Nakagami channel models at 3GPP frequencies. They present a model for channel variation where the key parameter is maximum Doppler shift, not a model-specific channel parameter, implying that more nuanced channel models give rise to similar channel dynamics. In most of the cases analyzed, this model showed good fit with measured data. From this, we conclude that analysis using Rayleigh fades is a good model for design exploration in this context.

Jakes's model treats Rayleigh fades as a sum of sinusoids [44]. We intend to isolate the contribution of multipath to channel variations at small timescales.

Consider a two-dimensional room with n static scatterers distributed uniformly at random with a static single-antenna transmitter in the middle of the room and a single-
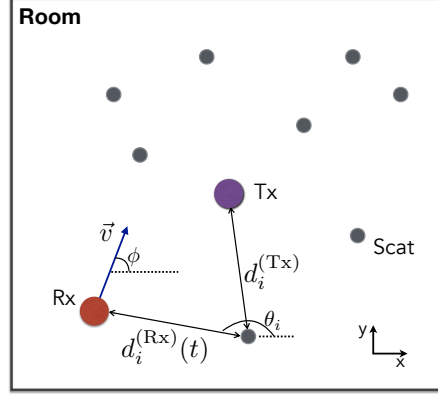
Figure 5.2. Room setup with n static scatterers, a static transmitter, and a mobile receiver. This figure originally appeared in [99].

antenna mobile receiver moving at constant speed v in some uniformly random direction. This scenario is depicted in Fig. 5.2, and has been explored in [99]. For a transmitter transmitting a tone at frequency f_c (wavelength λ_c), the channel coefficient between the transmitter and the receiver at time t is given by

$$h(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \exp\left(j \frac{2\pi \left(d_i^{(\text{Rx})}(t) + d_i^{(\text{Tx})}(t) \right)}{\lambda_c} \right),$$

where $d_i^{(\text{Rx})}(t)$ is the distance of scatterer i from the receiver at time t and $d_i^{(\text{Tx})}(t)$ is the distance of scatterer i from the transmitter. Normalizing by $\frac{1}{\sqrt{n}}$ makes the marginal variance equal for different numbers of scatterers.

The position of the receiver $\vec{s}(t)$ is given by

$$\vec{s}(t) = \vec{s}_0 + \vec{v}t = (x_0 + vt \cos \phi, y_0 + vt \sin \phi),$$

where $\vec{s}_0 = (x_0, y_0)$ is the initial position of the receiver (uniformly distributed in the room) and ϕ is the angle of motion with respect to the x-axis (unfiromly distributed on $[0, 2\pi)$. Given scatterer i with position $\vec{s}_i = (x_i, y_i)$, the distance of the receiver from scatterer i at time t is

$$
\begin{aligned}
d_i^{(\text{Rx})}(t) &= \|\vec{s}(t) - \vec{s}_i\| \\
&= \sqrt{(x_0 + vt \cos \phi - x_i)^2 + (y_0 + vt \sin \phi - y_i)^2} \\
&= \sqrt{d_i^{(\text{Rx})}(0)^2 + (vt)^2 + 2vt d_i^{(\text{Rx})}(0) \cos(\theta_i - \phi)},
\end{aligned}
\tag{5.1}
$$

48

where θ_i is the angle made by the line joining the scatterer and receiver at time $t = 0$, which is independent of ϕ.

The cross-covariance of the channel coefficient $h(t)$ is given by

$$
\begin{aligned}
k(v, t) &= \mathbb{E}[h(t)h^*(0)] \\
&= \frac{1}{n} \mathbb{E}\left[\sum_{i=1}^{n} \exp\left(j\frac{2\pi}{\lambda_c}\left(d_i^{(\mathrm{Rx})}(t) - d_i^{(\mathrm{Rx})}(0)\right)\right)\right. \\
&\quad \left. + \sum_{i \neq j} \exp\left(j\frac{2\pi}{\lambda_c}\left(d_i^{(\mathrm{Rx})}(t) - d_j^{(\mathrm{Rx})}(0) + d_i^{(\mathrm{Tx})} - d_j^{(\mathrm{Tx})}\right)\right)\right].
\end{aligned}
$$

Because $d_i^{(\mathrm{Rx})}(t)$ and $d_j^{(\mathrm{Rx})}(0)$ are independent for $i \neq j$ and the scatterers are distributed uniformly throughout the room, the expectation of the cross term is 0.

$$
k(v, t) = \mathbb{E}\left[\exp\left(j\frac{2\pi}{\lambda_c}\left(d_i^{(\mathrm{Rx})}(t) - d_i^{(\mathrm{Rx})}(0)\right)\right)\right]. \tag{5.2}
$$

Substituting (5.1) into (5.2) with a small movement approximation of $\frac{vt}{d_i} \approx 0$ gives a covariance function of

$$
k(v, t) = J_0\left(\frac{2\pi}{\lambda_c}vt\right), \tag{5.3}
$$

where $J_0(\cdot)$ is the Bessel function of the first kind.

Fig. 5.3 shows the autocorrelation and power spectral density given by (5.3), as well as some simulation results. The simulation is performed by repeatedly placing scatterers and a receiver uniformly at randomly, choosing a random direction for the receiver to move, and moving the receiver a small interval in each timestep. The simulations add a LoS component and show no significant divergence from the analytical model.

Each time a user performs channel estimation, it can be thought of as sampling the channel. In a low-latency system, small transmissions occur frequently and channel estimation is performed frequently, so the channel is being sampled with high frequency. By the sampling theorem for bandlimited random processes, we know that with high enough frequency sampling captures all the information content of the process. In this sense, low-latency communication is favorable for relay prediction; frequent transmissions give users the information needed to track the channel state without significant overhead.

A natural question is how frequently the channel must be sampled. From the PSD in Fig. 5.3, we see a peak beyond which the PSD rapidly decays which corresponds to the
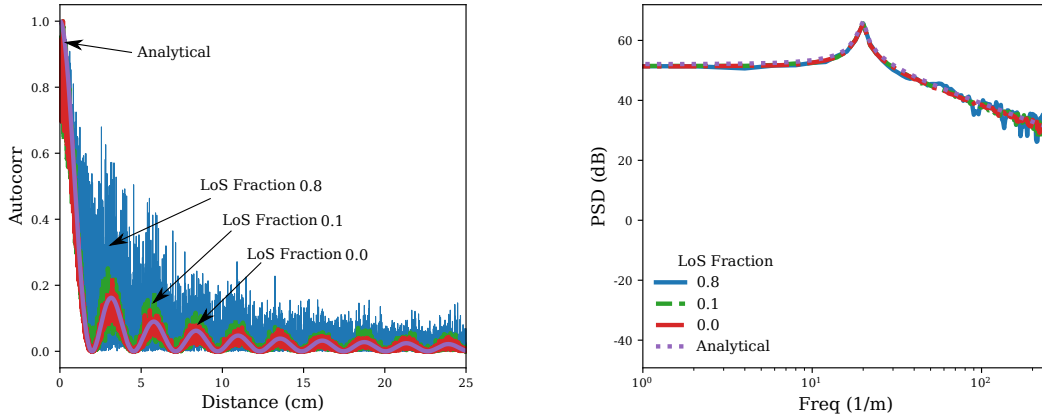
Figure 5.3. Simulated autocorrelation and PSD for channels with $f_c = 5.8\,\text{GHz}$. "LoS Fraction" is the fraction of the signal energy in the line-of-sight path. These figures originally appeared in [83].

maximum Doppler shift, i.e. the maximum relative velocity between the receiver and the scatterers or transmitter. As a result, the sampling rate needed to track the channel well is directly related to the maximum velocity of objects in the system. For $f_c = 5.8\,\text{GHz}$ and maximum velocity of $10\,\text{m\,s}^{-1}$, sampling at $1\,\text{kHz}$ is sufficient.

We apply the model discussed above to understand the relationship between the channel, communication latency, and number of relays required for reliable communication. Assuming there are k independent relays, we are interested in the probability that the best relay (as determined by an estimator based on sampling the channel) is not above the decoding threshold. The authors in [100] call this p_{lower}. The main parameters relevant to determining p_{lower} are

1. The frequency at which the channel is sampled (estimated)

2. The future time horizon over which predictions about the channel are being made

3. The number of potential relays to choose between

4. The nominal SNR

Fig. 5.4 has a result from [100], which shows how sampling frequency and the future time horizon are related. For $k = 9$, we see that prediction error is very low for short future time horizons and fast sampling frequencies. As the future time horizon increases, or the sampling frequency decreases, the prediction error quickly degrades to the error probability if a random relay is picked without any prior knowledge.
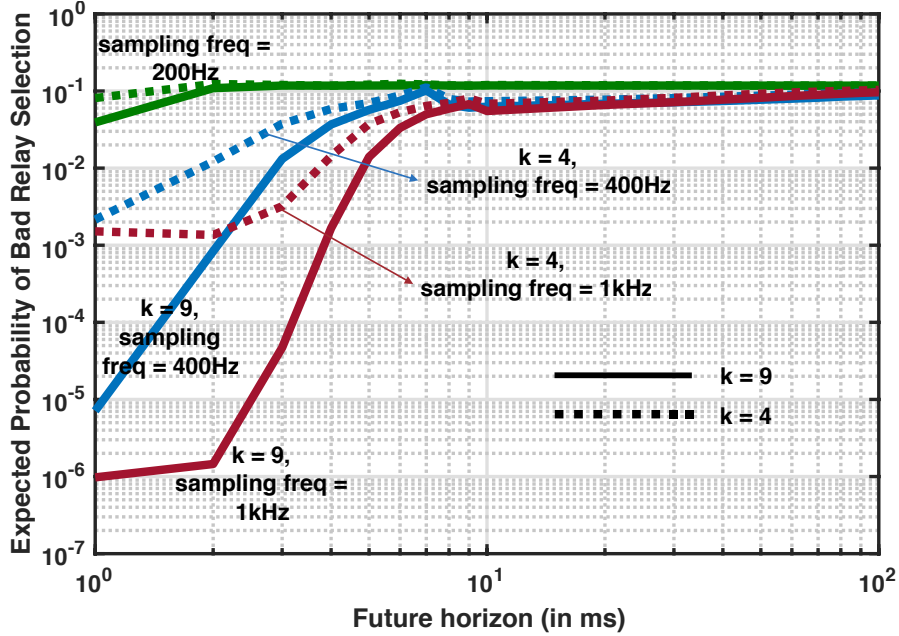
Figure 5.4. Probability of the 'best' relay (as determined based on past measurements with a predictive model) being in a deep fade for varying sampling frequency and future time horizons. The model order is 3, the number of relays to choose from is $K = 9$ (solid lines) and $k = 4$ (dotted lines), nodes are moving in a random direction at speed $10\,\text{m}/\text{s}$ and nominal SNR is 5 dB. This figure originally appeared in [100].

5.2   Channel Measurements

5.2.1   Experimental Setup

We perform measurements in a way conceptually similar to the simulations performed in Fig. 5.3. The experiment consists of a receive antenna and a transmit antenna in an indoor environment. The receive antenna is moved along a straight line while the receiver performs channel estimation. The transmit antenna is held in a fixed position for a sequence of measurements and is moved to a new location after the receiver completes its path. Line-of-sight between the transmitter and receiver is suppressed with a sheet of aluminum foil at the transmitter.

Two different measurement setups were employed. The first used a commercial transceiver with RF frontend, ADC, and DAC to estimate channel coefficients. The second used a vector network analyzer (VNA) to achieve the same task.

Transceiver-Based Setup

The measurement setup was located in an office building in downtown Berkeley. The hardware platform consisted of an Analog Devices (ADI) FMCOMM-3 RF frontend and Xilinx ZC706 FPGA for the baseband. The transmitter and receiver utilized the same board in order to share a local oscillator (LO), removing the need to correct for a frequency offset[3].

The ADI reference designs for the FMCOMMS boards [14] were modified to add a custom baseband implementation as shown in Fig. 5.7 and Fig. 5.8 [80], including stream ↔ memory DMAs and logic to control the timing of a capture. These blocks represent a subset of the baseband described in Chap. 7. This baseband was implemented using Dsptools and the methodology described in Chap. 6[4].

A Gold sequence with period 4095 is truncated to length 1029 and transmitted with period 1029[5]. The sequence was transmitted repeatedly, and raw samples were captured at the receiver, saved to a file, and postprocessed to find a series of channel estimates.

The position of the receiver was controlled by an XY-table with Parker Automation 404XE linear actuators, Ares servos, and 6K6 motion controller, shown in Fig. 5.5. A serial interface was used to send commands to the motion controller to position the antenna. This apparatus positions the receive antenna with accuracy $< 0.1\,\text{mm}$. A computer coordinated the collection of channel estimates and antenna movement, and a web application running on the FPGA provided an API for initiating and downloading channel estimates [81]. This is depicted in Fig. 5.2.1.

The antenna is moved on a linear path approximately $50\,\text{cm}$ long at a velocity of $150\,\text{mm/s}$. The carrier frequency was $5.8\,\text{GHz}$, approximately the highest frequency for common Wi-Fi deployments. Each measurement is 3 seconds long with samples taken at $10\,\text{MS/s}$. The AGC is fixed to a constant value for every measurement. The transmitted signal is 1029-periodic and the channel is estimated for each period of the sequence, so the effective rate at which the channel is estimated is approximately $9.7\,\text{kHz}$. Channel estimation recovers a complex value for the dominant channel coefficient.

The stream of samples is streamed to main memory via a stream ↔ memory DMA. A Linux kernel module implements a driver for using the synchronizer and DMA. A Django web application running on the ZC706 FPGA provides an API for interacting with the driver to initiate and download data captures.

---

[3]A frequency offset adds difficulty because it presents the same way as a Doppler shift. When measuring the channel, we would like to remove the LO contribution to frequency offset while leaving in Doppler shift.

[4]Chap. 7 goes into more detail, but in summary `DspBlocks` and diplomacy are used to implement custom streaming memory-mapped peripherals, parameterized Chisel testers unit test these blocks, and a top-level block integrates these blocks and presents IOs that can be connected to the Xilinx and ADI IP that are present in the reference design.

[5]Truncating the sequence was not done for any benefit, it was a result of an implementation detail on the FPGA. The number 1029 was not chosen for any benefit.
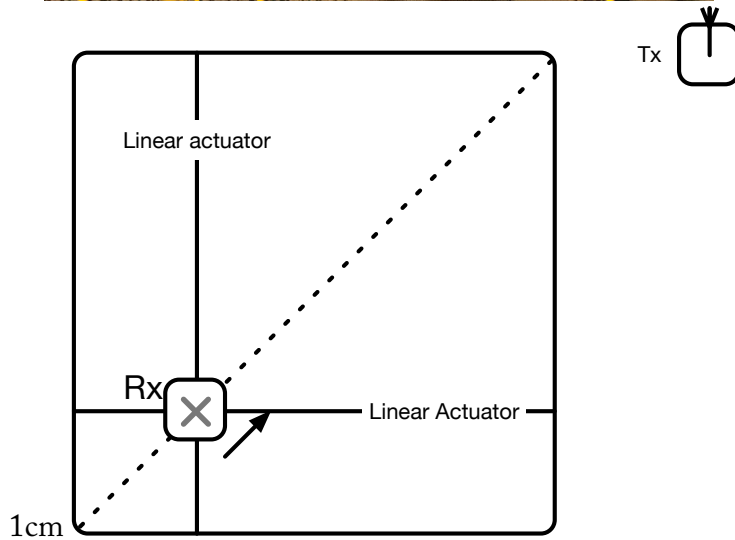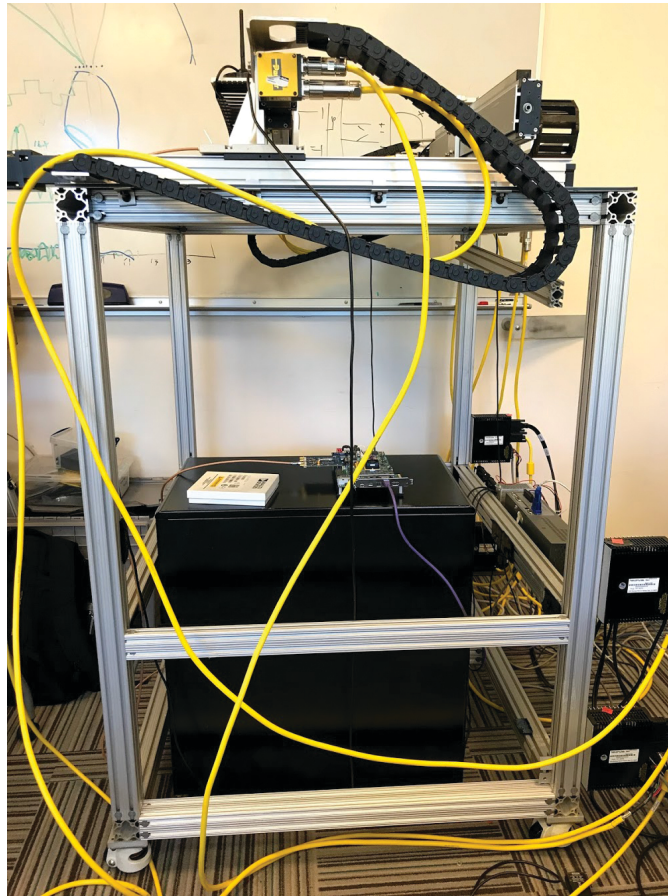
Figure 5.5. The XY-table. On the top is the XY-table, consisting of a Parker 6K6 motion controller is connected to two servo controllers for X and Y axis control. On the bottom is a diagram of the XY-table shown from above looking down.
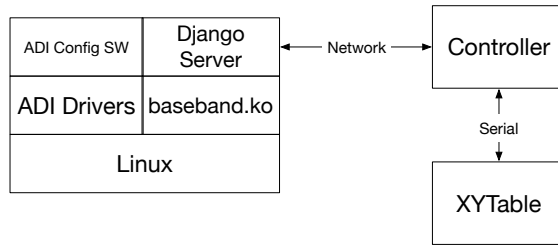
Figure 5.6. Block diagram showing how the XY-Table and FPGA are used together to record captures.
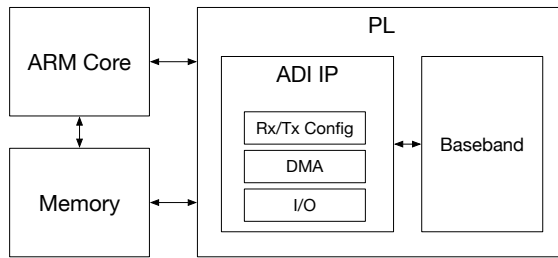
Figure 5.7. Block diagram showing custom baseband integrated on PL fabric with ADI IP.

VNA-Based Setup

The second measurement setup consists of a Rohde & Schwarz ZNB8 2-port vector network analyzer (VNA), SkyCross 2-2931-A wide-band antennas, and a positioner that moved 1 mm between channel measurements. The channels are captured in a rich scattering environment with no line-of-sight component. To achieve a high SNR, the IF bandwidth was set to 1 kHz which resulted in a SNR in the range of 30 dB to 40 dB. The VNA captured the channel at frequencies from 2.3 GHz to 6.0 GHz with a linear frequency spacing of 500 kHz. Several campaigns were conducted to prove that the measurements are repeatable. The VNA-based measurements were performed in Lund, Sweden by Christian Nelson [83], for which the author is grateful.
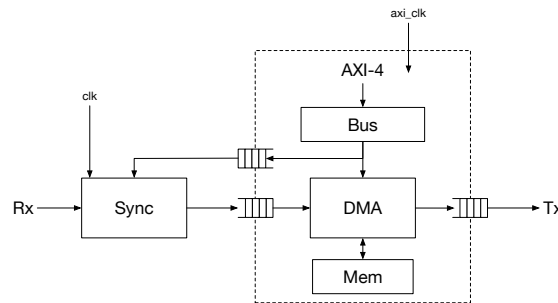
Figure 5.8. Block diagram of custom capture blocks.

5.2.2 Measurement Results
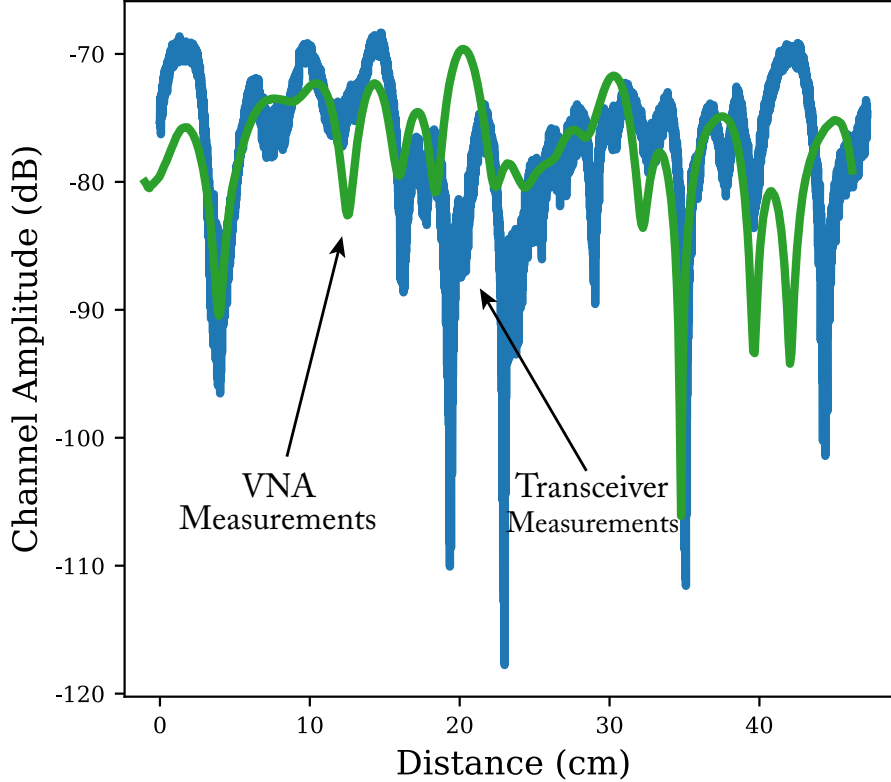
Figure 5.9. Representative examples of time-varying channel amplitude from the transceiver-based setup and VNA-based setup. This figure originally appeared in [83].

Fig. 5.9 is the amplitude of the estimated channel for a typical measurement from the transceiver-based setup. It is evident that the experimental setup is able to observe meaningful channel dynamics. Channel autocorrelations (specifically, circular autocorrelation) and power spectral densities are shown for the transceiver-based measurements in Fig. 5.10.

The spatial autocorrelation plot does not show perfect agreement, but they have similar drop-offs from the main lobe with periodic ripples after the initial drop. The power spectral density plots show good agreement. The PSD is relatively flat at low frequencies until it peaks at the maximum Doppler shift. All measurements showed peaking at approximately the same frequency. The peaks at 3 Hz are consistent with the velocity of the antenna, which is given by $v\frac{f_c}{c}$ =2.9 Hz for the parameters in this experiment. After the peak, all measurements show the expected roll-off of 20 dB/decade.

Captured power spectral densities are shown for the VNA-based measurements in Fig. 5.11. As in Fig. 5.10, the measurements have the expected shape with a peak at the
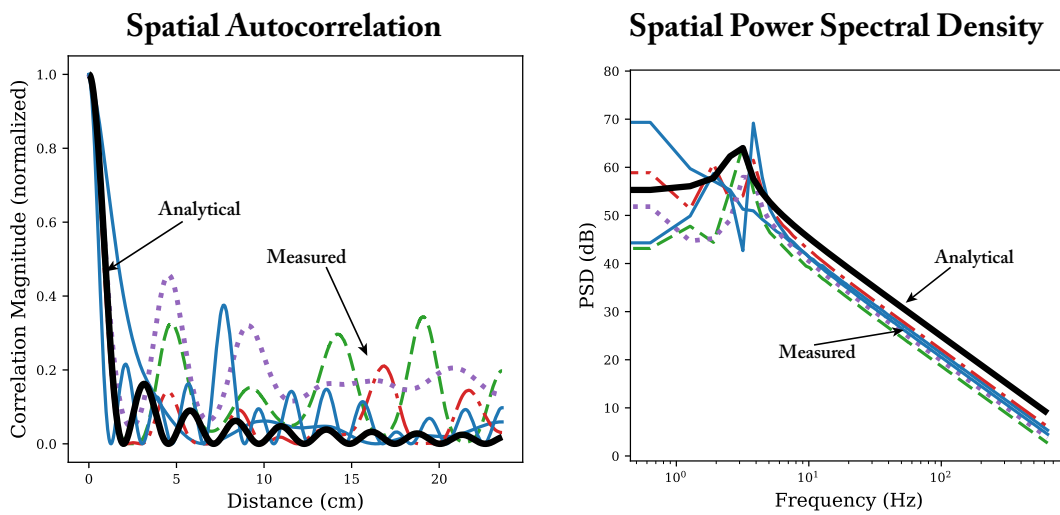
Figure 5.10. Normalized correlation and power spectral density of channel measurements at 5.8 GHz. Note that the spatial autocorrelation plot is in distance units whereas the power spectral density plot is in temporal frequency units. The scaling of the PSD plot emphasizes that the peak is at the expected frequency corresponding to the maximum Doppler shift. Both figures are normalized so that lag 0 of the correlation is 1. Bold black lines are the expected analytical result. All other lines are different measured results with the transmit antenna moved in-between measurements. These figures originally appeared in [83].
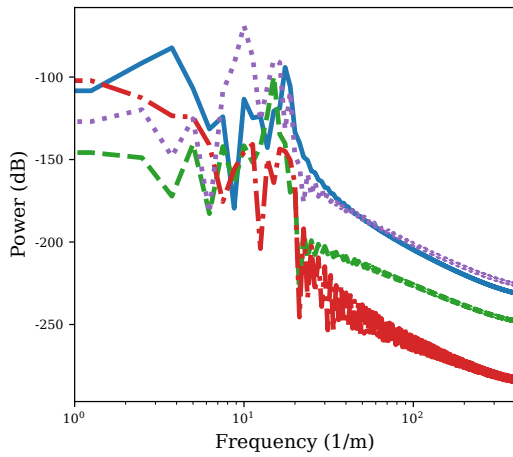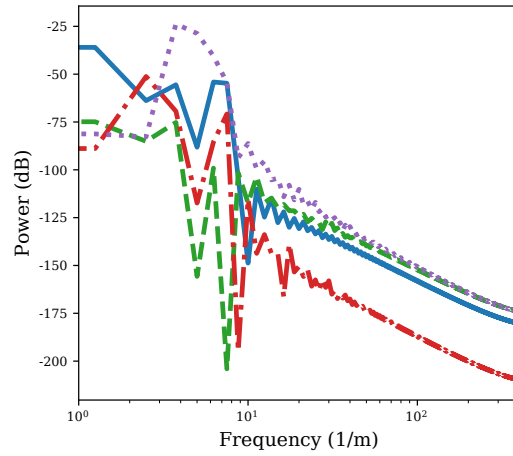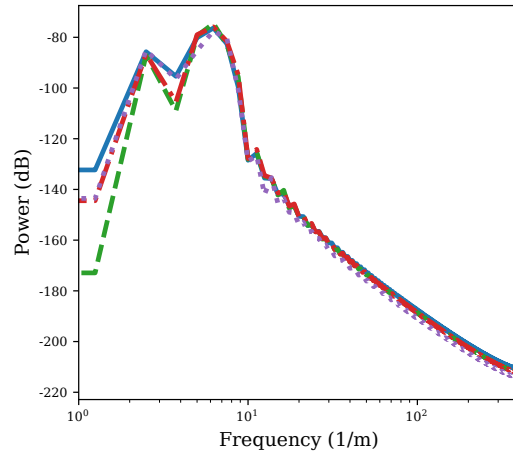
Figure 5.11. The figures above show measured power spectral densities from the VNA-based setup. Each line corresponds to a different measurement. The top figure has $f_c = 915\,\text{MHz}$, the center has $f_c = 2.45\,\text{GHz}$, and the bottom has $f_c = 5.8\,\text{GHz}$. These figures originally appeared in [83].
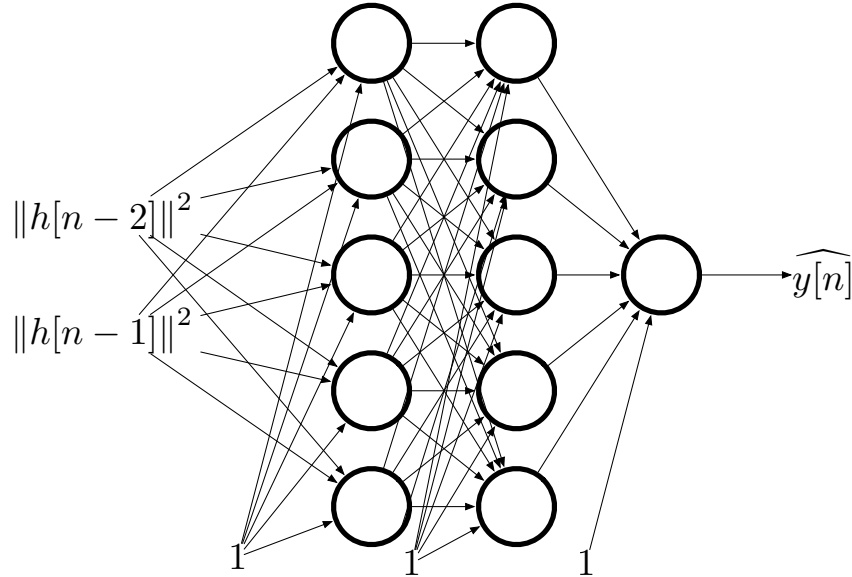
Figure 5.12. Architecture of neural network used to predict channel quality. The two inputs are the power of the past two channel measurements. All layers are fully connected with bias offsets. The sigmoid function is used as the activation function for all nodes. There is an input layer, two hidden layers, and an output layer. $\widehat{y[n]}$ gives a score that can be interpreted as the estimated probability that $h[n]$ will be above the decoding threshold.

maximum Doppler shift. Unlike Fig. 5.10, the figures are presented with spatial (rather than temporal) frequency. In spatial frequency, the peak location is given by $\frac{f_c}{c}$, so we expect the peak to be 8 Hz for the 2.45 GHz measurement and approximately 19 Hz for the 5.8 GHz measurement. The measurements are consistent with this. Overall, these measurements show good consistency with Jakes's model and the simulations in Fig. 5.3.

5.3 Relay Selection

Armed with measurements, we now apply relay selection algorithms to the data and evaluate their performance. This is similar to the study in [95] which applied relay selection algorithms to simulated data.

5.3.1 Problem Setup

We evaluate some simple relay selection algorithms. A time horizon of $\Delta = 6.67$ ms is chosen, which corresponds to 1 mm of movement at the chosen maximum velocity of 15 m/s. Each prediction algorithm uses m past channel estimates spaced Δ apart, i.e. $\{h(t - \Delta), h(t - 2\Delta) \ldots h(t - m\Delta)\}$, to predict if $h(t)$ is a good channel. Channel

powers are used in all cases for prediction. Channel measurements are normalized and a decoding threshold is chosen such that each channel measurement has outage probability of 27%, which corresponds to a nominal SNR of 4 dB and decoding threshold of 0 dB.

The measured data is preprocessed as described in Sec. 5.2.1. The input to the network is a m-tuple of channel qualities $(|h(t - i\Delta)|)_{0 < i \leq m}$. The expected output is a 0/1 variable that indicates if the channel to be predicted was above or below the decoding threshold. For each position the transmitter is placed at, which we consider to be a distinct emulated relay, there are 941594 input/output pairs.

5.3.2   Prediction Algorithms

Polynomial (Lagrangian) interpolation and a simple neural net were evaluated as channel prediction algorithms. The neural-net in [95] used $m = 2$ past channel estimates, but did not yield good results on measured data. Adding another hidden layer and widening the hidden layers from 2 to 5 resulted in improved performance[6]. The neural net is fully connected and has two hidden layers with five nodes each, as depicted in Fig. 5.12. The sigmoid function is used as the activation function at each node. The output can be considered an estimate of the probability that the channel will be satisfactory, given the past channel qualities.

The data is divided into training, validation, and testing sets, which consist of 9, 8, and 30 relays respectively. Training is performed via standard stochastic gradient descent on the training set. Hyperparameters are optimized with a tree-structured Parzen estimator on the validation set. Each channel prediction algorithm is evaluated in two ways: single-link estimation and paired-link estimation. In the single-link case, the prediction algorithm looks at a pool of n links corresponding to n potential relays. The estimator gives a score to each channel based on past measurements and the link with the highest score is selected. The fraction of time the selected link is good is an estimate of the prediction algorithm's ability to select good links on an individual basis.

In a relay selection scenario, there are two channels that need to be good for the relay to succeed: the controller-to-relay channel and the sensor-to-relay channel. The paired-link estimation evaluation emulates this scenario. Channel measurements are assigned to each other such that two channels form a pair. The channels are individually scored by the chosen prediction algorithm and a combined score is computed as the minimum of the two scores. The pair of channels with the best combined score is evaluated, and if it is below the decoding threshold we call it a failure. The fraction of time that a prediction algorithm fails is therefore an estimate for the probability that the relay selection algorithm would fail to choose a good relay.

---

[6]It is not clear why the network should be a particular size, nor is using a fully connected network particularly well-suited to this scenario. The intention is to demonstrate that a relatively small, simple network can achieve good performance. As the measurements here are noisier than the simulated data in [95], it is perhaps not surprising that the network needed to be expanded to perform well.

Single-Link Estimation   **Paired-Link Estimation**
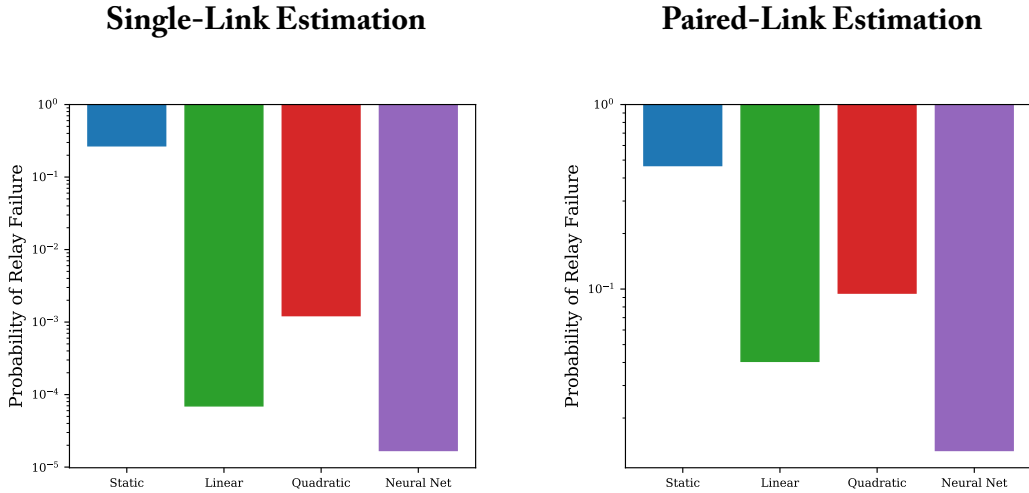
Figure 5.13. Results of various channel prediction algorithms. The single-link estimation results have $n = 30$ relays and the paired-link estimation results have $n = 15$ relays for a total of 30 distinct channels. These figures originally appeared in [83].

5.3.3   Channel Prediction Results

Both single- and paired-link estimation results are shown in Fig. 5.13. Static relay selection performs poorly because the outage probability is relatively high. Linear interpolation works better than quadratic interpolation, which may be explained by Lagrange interpolation's sensitivity to noise. The neural net performs best, but linear interpolation performs well too.

The paired-link estimation in Fig. 5.13 has worse failure probabilities than the single-link estimation. There are only 15 relays to choose between as opposed to 30 relays for single-link). Furthermore, there are two different ways to fail. However, the same trend is present for the paired-link estimation and the linear interpolation and neural net estimators are dramatically better than static relay selection.

5.4   Summary

This chapter discussed the efficacy of relay selection techniques for relay-based URLLC. Measured data was compared to analytical models and simulation results and showed good agreement, validating earlier results that showed the promise of relay selection techniques. Furthermore, predictive models were applied to the measured data to show that users can reliably select good relays. The next chapter will change topics to discuss techniques for prototyping custom hardware.

Chapter 6

Methodology for
Signal Processing Designs

Early in the development of a system, requirements are often fluid. As more of a system gets fleshed out, designers will understand what they are building better and identify shortcomings and improvements for what they initially set out to build. Good design methodologies and tools enable designers to change a component with minimal disruption to the rest of the system.

Some principles that are important for a good hardware design and verification methodology are:

- Re-use

- Safety

- Agility

The following sections will discuss these principles and how to design systems and tools that allow for flexible design.

6.1 Re-use

Abstraction is recognizing a pattern and giving it a name. Once an abstraction is made, it can be reapplied in different contexts without reiterating all the details. Abstraction, and the shortcuts through re-use it enables, are critical to designing complex systems.

Hardware design relies critically on abstractions at every level of the design hierarchy, but re-use is often hampered by tools and methodologies. This lack of ability to make good abstractions is especially stark in comparison to software design, where there is a rich history of tools and methodologies that support very powerful abstractions. This section discusses ways of promoting re-use in hardware design.

6.1.1 Generators

Metaprogramming is a set of features that allow programs to introspect and manipulate programs. Macros, `eval()`, reflection, and templates are all examples of language features that can be considered metaprogramming. Metaprogramming is very powerful because it can allow for very concise expression and reusable code.

The key differentiation between conventional HDLs and generators is that generators provide rich metaprogramming features that HDLs lack. How is metaprogramming useful for hardware design? Like software, many aspects of hardware designs can be generalized.

Today, a "hardware library" is typically an IP block, i.e. a coarse-grained reusable block with mostly standard interfaces. Sometimes, they can be configured to some extent. For example, Xilinx provides many configurable IP blocks such as an FFT with configurable size or a crossbar with configurable number of ports[1]. These IP blocks are reusable components that generally use standard interfaces to ease integration into a larger system.

IP blocks are very useful, but unfortunately only within limited scope. One issue is that the ability to reconfigure IP is usually fairly limited; things like input/output bitwidths, number of processing elements, and number of pipeline registers are fairly common. Less common is configuration that supports dependency injection that would allow for powerful customization. As a consequence, a common problem is that IP cannot implement the exact desired functionality. Changing IP or adding workarounds outside the IP can be difficult and add risk.

Chisel provides many more opportunities for reuse. Modules in Chisel can have much richer parameterization. Scala has a powerful type system and supports object oriented and functional programming paradigms. These are all useful for parameterizing hardware blocks.

Powerful programming language features provide opportunities for reuse. Encoding enums in the type system allows the compiler to help catch improperly configured hardware and provide warnings for when a module's author missed a case. Object oriented design allows parameters to be put in objects that collect and organize information. Having functions as first class objects allows for powerful generators where RTL can be injected by a user of the library. Polymorphic generators make it possible to design reusable components.

6.1.2 Modular Design

Modular design involves separating functionality into discrete, independent units. When a design pattern emerges, modular design allows the common functionality to be factored out and reused.

Using common interfaces, for example AMBA interfaces, is an important part of modular design. Common interfaces aide integration and verification, as well enabling as LEGO-style design where individual blocks are interchangeable.

Diplomacy Background

Diplomacy is a Chisel library that is shipped as a part of the Rocketchip generator [16]. It is a high-level library for different components to negotiate parameters in a principled way. It is particularly useful for generating interconnect where both master and slave sides of the interface may have requirements on the parameterization of the bus interface[1]. Diplomacy provides a mechanism by which both sides of the interface can specify parameters, after which a diplomatic `Node` implementation will determine if the connection can be formed and what parameterization the hardware bundle will have. To facilitate the "parameter negotiation," diplomacy employs a two-stage elaboration:

1. In the first stage, parameters are specified but no hardware is generated. Every diplomatic component is elaborated so parameters can be negotiated throughout the entire system.

2. In the second stage, every diplomatic component has been stage-1 elaborated. All parameters are known and the diplomatic `Node` can resolve all the bundle parameterizations. Concrete IOs and hardware are generated in this stage.

Rocketchip contains diplomatic implementations of AHB, APB, AXI-4, and TileLink, as well as commonly used functions such as crossbars, memories, and converters for the different interfaces.

Diplomatic AXI-4 Stream Interface

AXI4-Stream is an AMBA standard for streaming interfaces. The standard has many optional fields, but is at its core fairly simple. It defines the semantics of a ready/valid handshake, and then there are a number of fields that contain, describe, or modify data.

The `AXI4Bundle` type contains the fields depicted in Fig. 6.1:

In the standard, all signals except `valid` are optional. In the implementation here, `ready` and `last` are also mandatory for Chisel-implementation reasons. Optional fields

---

[1]An overview of how diplomacy works with a TileLink use-case can be found in [26].
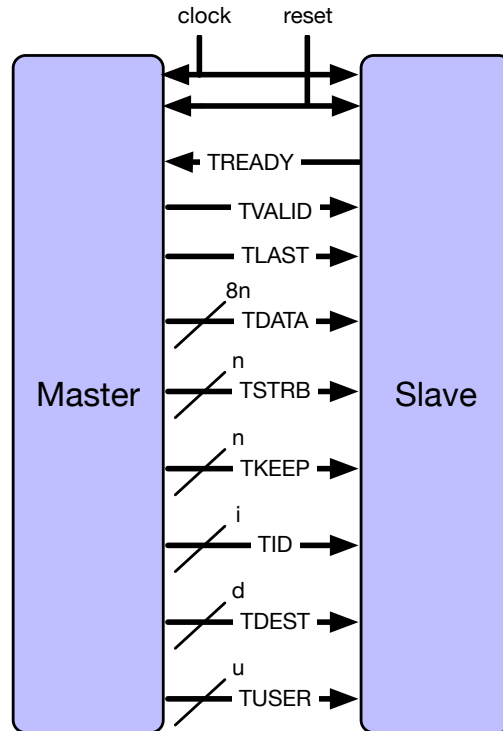
Figure 6.1. Illustration of signals in the AXI-4 Stream standard.

are implemented using `UInts` with width equal to zero, which get treated as constants by FIRRTL and removed from IOs. In many cases, an absent signal being replaced with 0 is fine, but special care should be taken for `strb` and `keep`. However, `Bool` in Chisel is a `UInt` with fixed width equal to one, so it cannot be easily treated as optional. `ready` and `last` are mandatory because they are chosen to be `Bool`s. `ready` is a `Bool` because it is desirable that the bundle be a `DecoupledIO` to make it easy to use with library components such as `Queue`, `Arbiter`, etc., and `DecoupledIO` has `ready` as a `Bool`.

The AXI4-Stream bundles can be used directly, but the recommended way to use AXI4-Stream is as a diplomatic interface. This means there is a two-stage elaboration: in the first, parameters are elaborated by a `Node`, and in the second hardware is generated, giving a `Bundle` as depicted in Fig. 6.2.

There are node types for kinds of blocks:

- `AXI4StreamIdentityNode`: a node that passes parameters through without any modification

- `AXI4StreamMasterNode`: a node that can only be on the right-hand-side of assignments (a source)

- `AXI4StreamSlaveNode`: a node that can only be on the left-hand-side of assignments (a sink)

64

```
class LM extends LazyModule {
  // used in stage 1 of elaboration
  // identity node have in and out with the same parameters
  val streamNode = AXI4StreamIdentityNode()


  lazy val module = new LazyModuleImp(this) {
    // used in stage 2 of elaboration
    // in and out are bundles corresponding to the input and
       output streaming bundles
    // (the underscores are parameters objects- sometimes it's
       nice to see the
    // parameters that diplomacy came up with
    val (ins, _) = streamNode.in.unzip
    val (outs, _) = streamNode.out.unzip
    ins.zip(outs).foreach { case (in, out) =>
      out <> in // connect in to out
    }
  }
}
```

Figure 6.2. Example illustrating diplomatic AXI-4 Stream interface.

65

- `AXI4StreamNexusNode`: a node that accepts many inputs and many outputs, and may mutate parameters accordingly

- `AXI4StreamAdapterNode`: a node that changes parameters

Some useful blocks for AXI-4 stream designs are implemented in Dsptools:

- `AXI4StreamFuzzer`: a fuzzer useful in synthesizable tests

- `AXI4StreamWidthAdapter`: adapters for adjusting the width by integer ratios

- `Mux`: programmable number of inputs and outputs, with control status registers (CSRs) to set which input goes to which output

- `DMA`: a stream `<->` memory map (AXI-4) DMA.

There are also async node types that can be used for automated clock domain crossings. This allows one to use crossing wrappers to cross in or out of a clock domain. However, the built-in rocket `CrossingWrapper` doesn't know about this implementation of AXI4-Stream, so the trait `HasAXI4StreamCrossing` needs to be mixed in.

Fig. 6.3 shows a sketch of what clock crossings with AXI4-Stream looks like.

```scala
val island = LazyModule(new

    CrossingWrapper(AsynchronousCrossing()) with

    HasAXI4StreamCrossing)
island.clock := someOtherClock
val streamBlock = island { LazyModule(new

    WhateverStreamModule) }
val out = AXI4StreamSlaveNode()
out := island.crossAXI4StreamOut(streamBlock.streamNode)
```

Figure 6.3. This is an example of how to use diplomacy to perform clock crossings with AXI-4 stream interfaces. The `island` variable defines a clock domain, which is set to use `someOtherClock` as the clock. The parameter used to construct the `island` variable is called `AsynchronousChrossing`, which indicates that the default async FIFO should be instantiated for the crossing. This behavior can be customized. `streamBlock` is constructed inside the island and therefore uses `someOtherClock` as its clock. The `crossAXI4StreamOut` function crosses the streaming interface to the outer clock domain, instantiating async FIFOs automatically.

DspBlock

A DspBlock is a unit of signal processing hardware that can be integrated into an SoC. It has a streaming interface and a memory interface. More concretely, `DspBlock` is a trait that has two members:

- an AXI4-Stream diplomatic node

- an optional, abstract memory interface diplomatic node.

The memory node is optional, that is it has type `Option[T]` where `T` is the type of the node. If the memory node is `None`, the block will not have a memory interface and will only have streaming interfaces.

The type of the memory node is a generic type `T`. In practice this means that a `DspBlock` can utilize any of the memory interfaces supported by rocket (AHB, APB, AXI-4, TileLink). It is a common design pattern to make abstract versions of a `DspBlock` and then bind implementations of the memory interface later, for example:

- An abstract `FIRBlock` that describes an FIR filter with programmable taps

- `AXI4FIRBlock` `extends` `FIRBlock` that instantiates an AXI-4 interface

- `TLFIRBlock` `extends` `FIRBlock` that instantiates a TileLink interface

In fact, a library author could make their own diplomatic implementation of a memory interface not supported by rocket (along with an implementation of regmapper) and create their own flavor of `FIRBlock` without changing the original `FIRBlock` class at all.

The definition of `DspBlock` is

```
trait DspBlock[D, U, EO, EI, B <: Data] extends LazyModule {

  ... }
```

The type parameters `D`, `U`, `EO`, and `EI` are parameter types of the memory interface and `B` is the bundle type of the memory interface. There are flavors of `DspBlock` for each memory interface, for example

```
trait TLDspBlock extends DspBlock[

  TLClientPortParameters,

  TLManagerPortParameters,

  TLEdgeOut,

  TLEdgeIn,

  TLBundle] { ... }
```

Control Status Registers

Rocketchip's `regmap` API is a useful way of generating logic for control and status registers (CSRs). The `regmap` API is generic with respect to the type of the memory interface. The trait `HasCSR` can be mixed into `DspBlock` to make the block's memory node use the regmap API. An example `DspBlock` called `ByteRotate` is shown below.

```scala
// D, U, EO, EI are memory interface parameter types

// B is memory interface bundle type

abstract class ByteRotate[D, U, EO, EI, B <: Data]()(implicit

  p: Parameters) extends DspBlock[D, U, EO, EI, B] with

  HasCSR {

 // Identity node- input and output have the same parameters

 val streamNode = AXI4StreamIdentityNode()


 // module is the hardware that gets generated after

    parameters are resolved

 lazy val module = new LazyModuleImp(this) {

   // get bundles for streaming inputs and outputs

   val (in, _) = streamNode.in.unzip

   val (out, _) = streamNode.out.unzip

   val n = in.head.bits.params.n

   val nWidth = log2Ceil(n) + 1


   // register to store rotation amount

   val byteRotate = RegInit(0.U(nWidth.W))


   def rotateBytes(u: UInt, n: Int, rot: Int): UInt = {

     Cat(u(8*rot-1, 0), u(8*n-1, 8*rot))

   }


   out.head.valid := in.head.valid
```

```scala
    in.head.ready := out.head.ready

    out.head.bits := in.head.bits


    for (idx <- 1 until n) {
      when (byteRotate === idx.U) {
        out.head.bits.data := rotateBytes(in.head.bits.data, n,
          idx)
      }
    }


    // generate logic for memory interface
    regmap(
      // address 0 -> byteRotate register
      // uses default read and write behavior- can override with
        RegReadFn and RegWriteFn
      0x0 -> Seq(RegField(1 << log2Ceil(nWidth), byteRotate))
    )
  }
}
```

regmap() takes a list of pairs of type (address -> Seq[RegField]. RegField
contains a reference to the hardware being accessed by the memory field, as well as meta-
data about the register such as name and read/write/execute permissions. The defaults
for reading and writing a register perform as expected, but also allow for adding custom
behavior on reads and writes.

Chains

Composability is important for libraries to be useful. A group of DspBlocks can be
connected to form a large DspBlock.

HierarchicalBlock is the general version of this concept. It has a list of
blocks called blocks: Seq[Block] and a list of connections called connections:
Seq[(Block, Block)]. It also defines a connect function that describes how the
edges in connections should be connected. The default is to simply do the diplomatic
connection on the streamNode (lhs.streamNode :=rhs.streamNode), but it may

be desirable to add queues, instrumentation, etc. depending on the implementation context.

One version of a `HierarchicalBlock` is a `Chain`. A `Chain` connects blocks sequentially, makes a crossbar, and connects every block with a memory interface to the crossbar. Because `Chains` are themselves `DspBlocks`, `Chains` can be nested.

Standalone Blocks
The primary function of diplomacy is to enable parameters to be negotiated by different blocks across a design. Diplomacy can make unit testing somewhat difficult:

- Diplomatic nodes are not meant to be top level IOs, but unit tests need the DUT to be top level

- Sink and/or source nodes are needed to parameterize diplomatic nodes

Preparing a `DspBlock` to be unit tested (especially by chisel-testers `PeekPokeTester`) can be tedious and error-prone, so Dsptools includes functionality to automate making `DspBlock` a top-level standalone DUT. This is achieved with mixin traits: - `StandaloneBlock` is the base trait and creates top level IOs that are connected to AXI4StreamMasterNode and AXI4StreamSlaveNode for the input and output of the `DspBlock` - Flavors like `TLStandaloneBlock` and `AXI4StandaloneBlock` specialize to specific memory interfaces

`StandaloneBlock` et. al. should not generally be mixed in with a `DspBlock`'s class. This mixin is typically reserved for top level blocks.

```
// DON'T DO THIS!!! (UNLESS YOU'RE POSITIVE IT'S WHAT YOU WANT)

class MyBlock() extends AXI4DspBlock with AXI4StandaloneBlock

  { ... }
```

Instead, it should be mixed into the tester, like these truncated examples below:

```
abstract class PassthroughTester[D, U, EO, EI, B <: Data](dut:

  Passthrough[D, U, EO, EI, B] with StandaloneBlock[D, U, EO,

  EI, B])

extends PeekPokeTester(dut.module)

class AXI4PassthroughTester(c: AXI4Passthrough with

  AXI4StandaloneBlock)

 extends PassthroughTester(c)
```

The tester then needs to be invoked with the mixin, like so:

```
// do the mixin here
val lazymod = LazyModule(new AXI4Passthrough(params) with
    AXI4StandaloneBlock)
val dut = () => lazymod.module


chisel3.iotesters.Driver.execute(Array("-tbn", "firrtl",
    "-fiwv"), dut) {
  c => new AXI4PassthroughTester(lazymod)
} should be (true)
```

`StandaloneBlock` et. al. work by using Rocketchip's `BundleBridges`. `BundleBridges` are diplomatic nodes that can contain any kind of bundle and can also be used to punch out IOs. Diplomatic converters converter `BundleBridges` to/from AXI4-Stream, AXI-4, TileLink, etc., and the `BundleBridge` is punched out to IOs.

6.2 Safety

One risk of generated RTL is that bugs can also be generated faster. It is critical to not only leverage automation for speed, but also for preventing and detecting bugs. A rich type system is a powerful tool for preventing bugs.

6.2.1 Types

One goal of Dsptools is to map mathematical constructs to hardware. In typical RTL descriptions of hardware, the underlying mathematical structure is not well preserved. Verilog and VHDL provide addition and multiplication for signed and unsigned integers, but beyond that the language doesn't provide much support for mathematical operations. Designers must implement their own reusable mathematical functionality. This functionality typically is reused at the module level, which can be unwieldy and encourages relatively coarse reuse.

Chisel can make use of powerful language features available in Scala that are not present in Verilog or VHDL. In Scala, operators can be overloaded, which allows one to instantiate complex behaviors for a wide range of operations.

Dsptools implements several useful types with implementations of mathematical operations. This allows designers to directly express a mathematical idea.

Floating Point

Dsptools provides a non-synthesizable floating point type. This allows a designer to start implementing an algorithm without needing to specify precision everywhere so IOs, control logic, and the algorithm can be validated. A floating point version of a circuit can be validated against a golden model with similar precision, or can itself serve as the golden model.

The floating point implementation uses Verilog `real` types, which are generally non-synthesizable. The FIRRTL interpreter and treadle backends do not simulate Verilog, so an alternative mechanism uses Scala models for floating point operations.
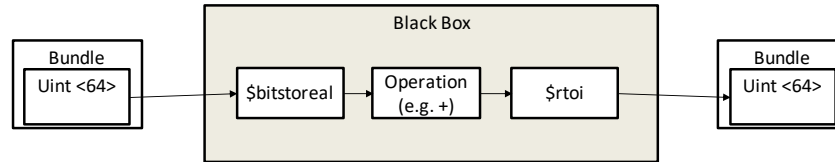
Figure 6.4. Non-synthesizable floating point implementation. A black box is used that calls operations like `$bitstoreal` that are a part of FIRRTL.

Fixed Point

Fixed point types are implemented in Chisel and FIRRTL[2]. Like `UInt` and `SInt`, types, `FixedPoint` numbers can be declared without a width and have FIRRTL's width inference algorithm infer the widths. `FixedPoint` numbers also have a binary point associated with them, which is also inferred by the width inference algorithm, as show in Fig. 6.5, which is a good example of a module using the `Ring` typeclass.

Intervals

Chisel's width inference algorithm is relatively simple and conservative. The analysis is performed individually for each operation, ignoring the context of previous operations. This means the width inference algorithm can be overly pessimistic, for example in this somewhat degenerate case:

---

[2]Bits is sealed, so fixed point types need to be defined in chisel. A library cannot define them unless it is wrapped in an aggregate type. Literals do not work very well with aggregate types. This is an ongoing issue to be resolved. Ideally, something like Fixed Point would be able to be added as a Chisel library.

```
val sel = Wire(Bool())

val a = Wire(FixedPoint(width = 10.W, binaryPoint = 9.BP))

val b = Wire(FixedPoint(width = 12.W, binaryPoint = 10.BP))

val reg = Reg(FixedPoint())

when (sel) {

    reg := a

} .otherwise {

    reg := b

}
```

Figure 6.5. An example showing width inference of FixedPoint numbers. The register `reg` is not given a width or binary point; instead, it is inferred.

```
 UInt(2.W) + UInt(1.W) + UInt(1.W) + UInt(1.W) + UInt(1.W) //

    => UInt(6.W), but max is 7 (3 bits!)
```

For `Intervals`, inference is performed on ranges rather than on widths. Each node in the circuit has a range associated with it that is inferred, just as in the case of width and binary points.

Complex Numbers

Dsptools defines a complex type. It is generic as to the underlying type, e.g. `DspComplex[SInt]` or `DspComplex[FixedPoint]` are valid. The only requirement is that the type of the fields have a `Ring` typeclass that provides addition, multiplication, and additive and multiplicative identities.

DspContext

`DspContext` is a mechanism for automatically managing metadata about signal processing operations. This metadata consists of pipelining, rounding, and precision settings.

Operations in Dsptools come in two flavors: a 'normal' flavor and a `context_` flavor. Normal operations like addition (+) work as expected, not adding any pipelining or rounding. `context_` operations insert pipeline registers and perform rounding as

described by the scope of the operation, so (a `context_+` b) will behave differently than (a + b).

The context's settings are set by a hierarchical scoping mechanism like so:

```
DspContext.alter(DspContext.current.copy(trimType = NoTrim,

    binaryPointGrowth = 3, numMulPipes = 2)) {

  val prod = a context_* b

  io.out := prod

}
```

6.2.2 Numeric Polymorphism

Polymorphism is a language feature that allows programmers to write interfaces that accept objects of different types. Conceptually, one might think of the type of an object as being a parameter to a function, class, etc. In the context of circuit generators, polymorphism can be used in many ways. Polymorphism is used internally throughout the Chisel codebase; `Reg`, `Wire`, etc. are all polymorphic and can be used to make registers or wires of `UInts`, `SInts`, `Bundles`, etc. In the context of hardware generators, we call this *data polymorphism*, that is polymorphism where the contents of the data are opaque to the generator and the generated circuit performs some function related to storage, movement, serialization, deserialization, etc.

A more sophisticated and familiar example of data polymorphism is a FIFO, as shown in Fig. 6.6.

```
object Queue {

def apply[T <: Data](enq: ReadyValidIO[T], entries: Int = 2,

  pipe: Boolean = false, flow: Boolean = false):

    DecoupledIO[T] = {

    ... }}
```

Figure 6.6. Function signature for Chisel's standard library implementation of Queue. Notice the type constraint `T <:Data` that enforces `T` is a subtype of `Data`, meaning that it is a hardware type that can exist in a circuit. There are no other constraints, so no operations like addition or multiplication can be performed on objects of type `T`.

Data polymorphism is useful, especially for writing the general purpose processors that Chisel was initially designed to implement. However, in the context of writing cus-

tom signal processing hardware, we often run into situations where we want to perform an operation on the data[3].

We say that polymorphic generators that perform numeric operations on the polymorphic types have numeric polymorphism. Some use-cases for generators with numeric polymorphism are outlined here:

- **Float to Fixed "Conversion":** Writing a floating point model and then automatically converting it to use fixed point numbers is a common use-case. Polymorphic generators can be used to achieve similar functionality, although conversion is not the best way to describe how it works. One polymorphic generator is written that can generate circuits for either floating point or fixed point numbers. A generator author can begin writing and testing with floating point in mind, verify that it works, and then specialize it for fixed point. One advantage of this is that tweaking the fixed point version automatically updates the floating point version, preventing golden model-implementation mismatch. See Fig. 6.7 for an illustration.

- **Real and Complex Generators:** Many signal processing blocks are applicable to real or complex inputs with relatively minor modification. FIR filters, summing, and interpolators are some examples. Operations like addition, subtraction, and multiplication are conceptually the same for these types but are implemented differently. Polymorphic generators allow a designer to make a test one block that can be applied to both real- and complex-valued inputs.

- **Special Number Representations:** In some situations, alternative number representations can be advantageous for a specific application, for example sign-magnitude representations for situations in which the inputs are frequently very close to zero (transitions between -1 and 0 flip more bits in two's complement than sign-magnitude) or n-ary representations when modulus operations are frequently performed. With many design methodologies, deciding to use an alternative number representation means throwing away existing blocks that use two's complement or performing conversion. Polymorphic generators open the possibility for reusing existing designs with new number representations in a typesafe way.

Implementing generators with numeric polymorphism comes with some challenges which are outlined here.

One approach is to enumerate all possible operations in a type and use it as a type constraint, for example replacing `T <:Data` with `T <:Data with Num[T]`[4]. The type

---

[3]Even data polymorphism is outside of the ability for conventional HDLs to support well because of their lack of first class functions. It can be emulated to some degree with parameters for widths and string representing different categories for the module to generate different logic conditionally, but numeric polymorphism relies even more heavily on support for first class functions and is thus awkward to emulate in HDLs.

[4]Chisel has a `Num` trait that defines many operations, including `+`, `-`, `*`, `/`, `%`, and comparison operators.
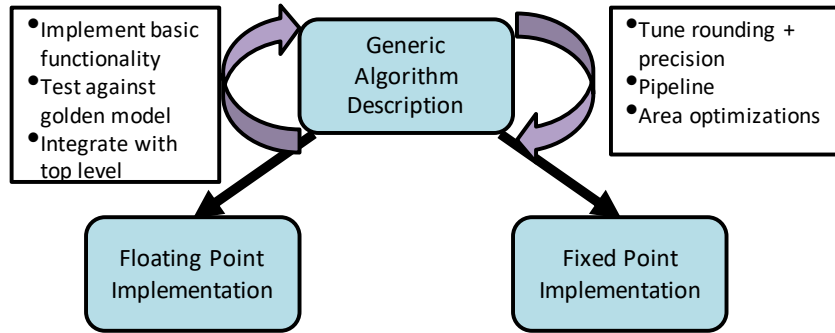
Figure 6.7. Illustration of how generators with numeric polymorphism can be used to provide functionality similar to float-to-fixed conversion.

parameter `T` is necessary to determine the return type of operations, which immediately presents one problem. If there is a type `A extends Data with Num[A]` and a type `B extends A`, that is, `A` is a type with numeric operations and `B` inherits from `A` (perhaps to customize the behavior of some operation), the return type of those operations in `B` will be `A` instead of the desired `B`. Scala has some patterns to work around this, for example the standard library has abstract types with Like as a suffix that concrete instances inherit (`Seq` vs `SeqLike`).

Another problem with `Num` is that one might want to have multiple notions of a number. Complex numbers should not support all the operations that a real number does (for example, comparison). If inheritance is being used as a tool to describe numeric types, should real inherit from complex because more operations are possible for real numbers? Should complex inherit from real because conceptually complex numbers are an extension of the reals? Furthermore, inheritance can make it difficult to incorporate types from outside of the inheritance structure, for example types from a different library. Often, wrappers and converters need to be made between two different libraries, and the complexity can grow very quickly as more libraries are added.

Duck typing is an alternative to using inheritance structures within a type system, but comes with strong downsides. Using the type system to catch errors is very valuable, especially in the context of hardware where verification generally takes more effort than design.

Ad-hoc polymorphism represents a good compromise between the structure of inheritance-based subtyping and the flexibility of duck typing.

6.2.3 Typeclasses

Typeclasses are a language feature for implementing ad-hoc polymorphism. In Scala, they are implemented with syntactic sugar for implicit parameters, and typeclasses are similar to implicit parameters in some ways. A polymorphic function that is generic

with respect to a type `T` can be declared with a type constraint requiring a typeclass `TypeClass` like so:

```scala
trait TypeClass[A] { def doSomething(a: A): A }

def polyFunc[T : TypeClass](in: T): T = { ... }
```

`polyFunc` will typecheck when called on variables of any type, as long as a typeclass with type `TypeClass[T]` is defined, in which case we say `T` has an implementation of `TypeClass` available. If an implementation of `TypeClass` is not available for `T`, it will not typecheck.

The type constraint `T : TypeClass` allows `polyFunc` to call `TypeClass[T].doSomething(in)`. The use of Scala implicits allows this somewhat awkward construction to be simplified to `in.doSomething()`, but this more convenient syntax is not actually calling a function on `in`, but is in fact redirecting it to the typeclass implicit object. This is an important distinction.

It is important to note that this relationship is different than inheritance, written in Scala `T <:BaseClass`. In this case, `polyFunc` can only be called on types `T` that are or inherit from `BaseClass`. The typeclass constraint places no requirements on an inheritance structure and performing operations based on the typeclass constraint does not directly call anything on `in`, instead proxying those calls through a typeclass object. This indirection is what allows typeclasses to be somewhat like duck typing.

Dsptools provides numeric typeclasses based on those from the Spire library [112]. Some important ones are:

- `Ring`, which refers to the algebraic structure. It supports `+`, `-`, `*`, `**`, and has members for the additive identity `zero` and the multiplicative identity `one`. An example usage looks like

```scala
def doSomething[T <: Data : Ring](a: T, b: T, c: T): T = {
  (a - b) * c
}
```

- `Eq`, which defines `===` and `=/=`, the equality operators. Notably, these operations return Chisel `Bool`s as opposed to Scala `Boolean`s. This is the biggest point of divergence between the Dsptools and Spire typeclasses. In Spire, conditionals are all evaluated at runtime and hence deal with `true` and `false`. In Dsptools, the typeclasses are being used to generate a circuit, so the value of a conditional is not known at circuit elaboration time.

- `PartialOrder` `extends` `Eq`, which adds comparison operators that return a `valid` signal alongside the value to indicate if the values are comparable.

- `Order` `extends` `PartialOrder`, which defines comparison operations.

- Sign, which defines abs and related sign-testing operations

- Real `extends` Ring `with` Order `with` Sign, which aggregates all the operations above while also adding ceil, round, floor, and isWhole.

- Integer `extends` Real, which adds a mod operation to Real.

```scala
class TransposedStreamingFIR[T <: Data:Ring](
  inputGenerator: => T, outputGenerator: => T,
  tapGenerator: => T, numberOfTaps: Int) extends Module {
  val io = IO(new Bundle {
    val input = Input(inputGenerator)
    val output = Output(outputGenerator)
    val taps = Input(Vec(numberOfTaps, tapGenerator))
  })


  val products: Seq[T] = io.taps.reverse.map { tap: T =>
    io.input * tap // Ring multiplication
  }


  val last = Reg(products.head.cloneType)
  last := products.reduceLeft { (left: T, right: T) =>
    val reg = Reg(left.cloneType)
    reg := left
    reg + right // Ring addition
  }


  io.output := last
}
```

Figure 6.8. Example FIR filter using Dsptools typeclasses.

Most operations defined in these typeclasses have two flavors, a 'normal' flavor and a 'contextual' flavor. The 'normal' flavor should perform the operation in a single-cycle fashion consistent with how Chisel width-inference rules normally work. For example, `a + b` using the `Ring` typeclass on a `UInt` should operate the same way as if there were no typeclass[5]. The 'contextual' flavor performs the operation using the `DspContext` and its settings in the current scope, as discussed in Sec. 6.2.1. Pipeline registers and rounding may be performed differently based on the current context.

The following use-cases refer to the FIR filter example in Fig. 6.8.

Use Case: Float-to-Fixed Conversion

The following code shows an example of using a polymorphic generator to generate both floating point and fixed point versions of an FIR filter from Fig. 6.8. This is a good alternative workflow to popular "float-to-fixed conversion" workflows.

```
val protoFloat = DspReal()

val protoFixed = FixedPoint(10.W, 5.BP)

val protoFixedOut = FixedPoint(20.W, 8.BP)

Module(new TransposedStreamingFIR(protoFloat, protoFloat,

    protoFloat, 10)

Module(new TransposedStreamingFIR(protoFixed, protoFixed,

    protoFixedOut, 10)
```

Use Case: Real and Complex Filters

The following code shows an example of two module instantiations using the same generator from Fig. 6.8. They both instantiate FIR filters using `FixedPoint` representations, but the first is a real-valued filter and the second is a complex-valued filter. They both use the same polymorphic generator.

```
val protoFixed = FixedPoint(10.W, 5.BP)

val protoFixedOut = FixedPoint(20.W, 8.BP)

val protoComplex = DspComplex(protoFixed)

val protoComplexOut = DspComplex(protoFixedOut)
```

---

[5]This prevents bugs from confusion around whether there is a typeclass present.

```
Module(new TransposedStreamingFIR(protoFixed, protoFixed,
    protoFixedOut, 10)
Module(new TransposedStreamingFIR(protoComplex, protoComplex,
    protoComplexOut, 10)
```

Use Case: CORDIC

CORDIC is an algorithm for computing a number of hyperbolic and transcendental functions that maps efficiently to hardware, using only shifts and add/subtract operations with a look-up table (LUT). CORDIC is an interesting example for polymorphic generators because the division operation relies on the representation of the number in order to efficiently implement the division with a shift. A CORDIC using a built-in Chisel type does not need to take this into account because built-in types are all two's complement and define shifting. However, a polymorphic CORDIC must use an additional typeclass, called `BinaryRepresentation`, which adds operations that are possible for numbers using binary representation. Division via shifting is one of those operations[6].

Fig. 6.9 shows a stage of the CORDIC algorithm implemented for `FixedPoint`, and Fig. 6.10 shows the same stage implemented in a polymorphic way with the `BinaryRepresentation` typeclass. It is important to note that they look almost identical. Even blocks that rely on the underlying representation of the number are not so difficult to generalize to more types.

Use Case: User-Defined Types with Library Generators

Composability is important for code to be reusable. A polymorphic generator written with Dsptools should be usable with a type not defined by Dsptools or any of its dependencies.

The example in Fig. 6.11 shows what it looks like to take a pre-existing generator (in this case, an integrator) and generate a version of it that uses a newly defined, custom type that does not exist in Dsptools (in this case, an integer represented in sign-magnitude form).

---

[6]These operations could be emulated for non-binary representations, but this circumvents the intent of having an efficient division operation. In some cases, like non-synthesizable floating point, it is useful to do so anyways.

```scala
class CordicStage(params: CordicParams[FixedPoint]) extends
    Module {
  val io = IO(new Bundle {
    val in = Input(CordicBundle(params))
    val vectoring = Input(Bool())
    val shift = Input(UInt())
    val romIn = Input(FixedPoint(params.nStages.W,
        (params.nStages-1).BP))
    val out = Output(CordicBundle(params))
  })
  val xshift = io.in.x >> io.shift
  val yshift = io.in.y >> io.shift


  val d = Mux(io.vectoring,
    io.in.y.signBit(),
    !io.in.z.signBit()
  )


  io.out.x := AddSub(!d, io.in.x, yshift)
  io.out.y := AddSub( d, io.in.y, xshift)
  io.out.z := AddSub(!d, io.in.z, io.romIn)


}
```

Figure 6.9. Fixed point implementation of CORDIC stage.

```
class CordicStage[T <: Data : Ring : BinaryRepresentation](val
    params: CordicParams[T]) extends Module {
  val io = IO(CordicStageIO(params))


  val xshift = io.in.x >> io.shift
  val yshift = io.in.y >> io.shift


  val d = Wire(Bool())


  when (io.vectoring) {
    d := io.in.y.signBit()
  } .otherwise {
    d := !io.in.z.signBit()
  }


  io.out.x := AddSub(!d, io.in.x, yshift)
  io.out.y := AddSub( d, io.in.y, xshift)
  io.out.z := AddSub(!d, io.in.z, io.romin)
}
```

Figure 6.10. Polymorphic implementation of CORDIC.

```scala
class Integrator[T<:Data:Ring](genIn: T, genReg: T) extends
    Module {
  val io = IO(new Bundle {
    val in = Input(genIn.cloneType)
    val out = Output(genReg.cloneType)
  })


  val reg = RegInit(genReg, Ring[T].zero) // init to zero
  reg := reg + io.in
  io.out := reg
}
// Declare new type
class SignMag(val magWidth: Option[Int]=None) extends Bundle {
  val sign = Bool()
  val magnitude = magWidth.map(w => UInt(w.W)).getOrElse(UInt())
}
// Implement typeclass with operations for new type
trait SignMagRing extends Ring[SignMag] {
  def plus(f: SignMag, g: SignMag): SignMag = ???
  def times(f: SignMag, g: SignMag): SignMag = ???
  // etc.
}
// Run the generator!
Module(new Integrator(new SignMag(Some(4)), new
    SignMag(Some(10))))
```

Figure 6.11. An implementation of a custom sign-magnitude type that can be used with a polymorphic integrator generator.

6.2.4 Integrating Signal Processing Blocks into an SoC

Custom signal processing hardware is generally one component of a large, complicated system. The usefulness of this custom hardware is often dependent on how it is integrated into an SoC. Memory interface standards and parameterizations, DMA designs, and programming interfaces are important dimensions to the design-space exploration.

Dsptools provides functionality that makes it easier to interface custom signal processing hardware with a Rocketchip-based SoC. A `DspBlock` can be made generic with respect to its memory interface, so it is possible to explore the trade-off between performance, area, and power for different memory interfaces for different blocks in an SoC. This is also helpful when prototyping a system in different contexts. It might be desirable to target AXI-4 for emulation with Xilinx FPGAs, but to target TileLink for taping out a Rocketchip based SoC. Polymorphic generators allow one description to target both environments.

Dsptools is integrated with Chipyard [13], a project for building RISC-V based SoCs. Dsptools-based designs can be integrated into SoCs that use Chipyard's wide range of cores and accelerators, including RISC-V cores such as Rocket, BOOM, and Ariane [128] and accelerators like Hwacha, Gemmini, and NVDLA [69].

6.2.5 Collateral Generation

In addition to RTL, generators can output collateral that aides in verifying, integrating, and writing software for a design. For verifying generated designs, it can be very useful to have metadata describing the DUT. Chisel testers can introspect the design, but other verification flows cannot. IP-XACT [115] is a serialization format for expressing metadata about circuits. It was used in [19] to allow Chisel-generated RTL to be tested by Python scripts that generated stimulus based on the parameterization of the design, as well as test harnesses that were generated based on the IP-XACT description of the circuit. This sort of metadata serialization is a critical step in making verification flows for generated designs.

This IP-XACT flow was enhanced to work generically with any block written in Chisel. The main mechanism by which metadata is described in Chisel is with annotations. Annotations can target any entity with a FIRRTL circuit, can trigger transformations to run in the FIRRTL compiler, and can be generated at any point in the design, be it from the user, a library author, or even a FIRRTL transformation. Many Rocketchip components will automatically annotate entities with useful information, for example memory interfaces may be annotated with information about their parameterization and the memory map, if one exists.

An experimental Dsptools feature[7] implements a custom FIRRTL transform that

---

[7]This can be found at `https://github.com/ucb-bar/dsptools/tree/ipxact-emitter-2`

searches for FIRRTL annotations that describe a design, gathers them, and translates them into IP-XACT. This feature works incrementally; if no relevant annotations are found, IP-XACT that describes the RTL's physical port structure and file location is emitted. If some annotations describing the type and parameterization of an IO is found, then a physical-to-logical mapping is also emitted. If an annotation describing a memory map is also found, then that is also translated and emitted as IP-XACT. Other annotations describing the parameterization of the generated module can also be captured by this transform and properly emitted in the output IP-XACT.

Generated designs can also emit software for interacting with the design. In [19], C headers with address maps, design information (e.g. presence of blocks, how many, etc.), and helper functions were emitted alongside the design RTL. This helps prevent software from being out of sync with the RTL, or cause a compilation error if it there is a problem.

Another kind of collateral that can be useful for generators to emit is useful debug information. Chisel has an annotation for commenting output Verilog. This functionality can be used by FIRRTL passes to add comments describing what was done, which may help debugging generated Verilog. One example is an experimental FIRRTL pass that adds comments for every `FixedPoint` value that labels what the binary point is for that value[8]. This can be very helpful when debugging values with long chains of inferred widths.

Verilog attributes are another kind of metadata that can be added with annotations via a FIRRTL pass. One use case for this is to add Verilog attributes adding a signal to Xilinx's ChipScope debugger for on-chip debugging. Without this sort of automation, this kind of task is often done via TCL scripts that must be kept in sync with the design as names and hierarchy change. This is especially frustrating when out-of-sync TCL scripts cause failures late in the implementation flow, so automation is especially attractive.

6.3   Agility

Re-use and safety are of critical importance for agility. Reusable code saves wasted time reinventing the wheel and lets one invest time in the interesting parts of the design. Safety allows one to make changes with some amount of confidence that the changes will work. Without safety, agility is impossible because most time will be spent verifying changes instead of making changes. The three important principles discussed at the beginning of this chapter are all related.

Agility merits its own discussion. The features discussed here may not provide much benefit in terms of re-use and safety, but they serve the ultimate goal of agility.

---

[8]This can be found at `https://github.com/freechipsproject/firrtl/pull/1036`.

6.3.1 Backend Inference

Complex generators can make it difficult to figure out the widths and precisions for every section of a circuit. However, part of the reason for designing custom signal processing hardware is to benefit from the use of lower precision where possible. Width inference in Chisel supports a philosophy of specifying widths where they are important, e.g. high impact on QoR, and infer widths elsewhere. Places where widths are likely to have high impact on QoR tend to be IOs, memories, and multiplies.

`FixedPoint` types bring this philosophy to a richer kind of number than a `UInt` or `SInt`. In addition to widths, binary points are also inferred. `Intervals` are similar to `FixedPoint` numbers but even more aggressive at removing unneeded bits, as described in Sec. 6.2.1.

Beyond these kinds of static width inference[9], it is possible to optimize bitwidths based on observations from simulating the design with real data. An instrumented version of the FIRRTL interpreter [3], which is one of the execution engines for Chisel testers, tracked maximal values each node in the FIRRTL graph experienced for a number of simulations. Using this information, criteria such as "fit the maximal value" or "size each node to fit values $\leq n\sigma$ away from mean" could be applied to trim bits for some nodes[10]. This process is applied iteratively until no bitwidths are reduced.

These width optimization techniques are compared in Fig. 6.12. `Interval` can provide a significant improvement over `FixedPoint`, and the dynamic methods generally improve even further. The gains are generally largest for designs that use a relatively small number of bits.
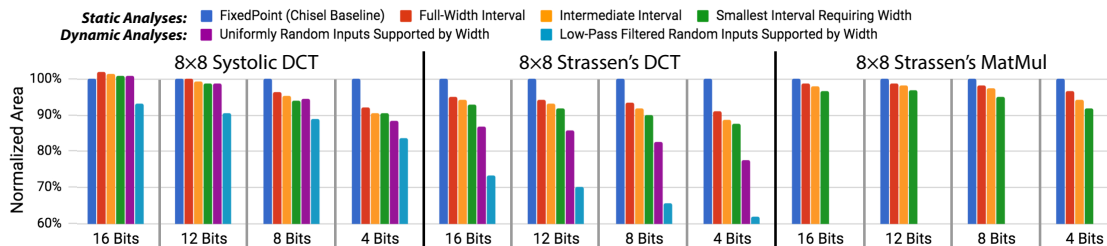
Figure 6.12. Area results for various methods of width optimization for representative designs. This figure originally appeared in [120].

6.3.2 Lightweight Unit Testing

Unit testing is a critical piece of the overall verification effort. Unit testing aids a designer in understanding what hardware was generated from the code they have written.

---

[9]In some cases, such as for linear time-invariant (LTI) systems as in [118], direct formulas for minimal bitwidths can be evaluated.

[10]For dynamic bitwidth optimization it is important that the input be representative of the actual inputs.

Writing generators is metaprogramming, so it is often non-obvious to a designer what hardware they have generated, especially in degenerate parameterizations. Unit tests help the integration testing effort by performing simple tests in a fast, lightweight environment before getting to the slower, generally more cumbersome integration testing environment.

Making it easy and quick to write good unit tests is very important. Making tests easy to write makes it more likely that tests will be written early in the design cycle and makes it so that more tests are written. These early tests are very valuable because catching bugs early is more valuable than catching bugs late.

Chisel testers provides a harness for writing lightweight tests for Chisel designs. These testers elaborate the design under test (DUT), provide a low-level API for interacting with the DUT, and provide a mechanism for signaling success or failure and marking where failures occurred.

The low level API consists of:

- **peek:** read the value currently held by a top-level IO

- **poke:** set a top-level IO to a given value

- **expect:** `peek` a top-level IO and signal failure if the value does not match the expected value

- **step:** advance the simulation one clock cycle

Dsptools extends these testers to provide convenience APIs for interacting with numeric types. One example is being able to call `peek`, `poke`, and `expect` on fixed and floating point numbers with double precision numbers. Without these convenience functions, a test for a fixed point design would need to find the binary point of all fixed point IOs and scale inputs and outputs accordingly. With the convenience functions, it is much more direct. `expect` calls can be configured to have an error tolerance so that calls such as `expect(io.out, 1.0 / 3.0)` work.

6.3.3 Generic Testers

Verification generally takes more engineering effort and time than design. Hardware generators make design faster, but can add to the verification effort. In order for generators to be useful, it is important that the verification infrastructure be just as flexible as the designs being testing.

One area where this is difficult is when testing polymorphic generators. One of the design goals of such generators is to enable library users to use a generator with their own custom types. How can such designs possibly be tested?

It is infeasible to test every possible design point for polymorphic generators. A more practical solution is to write polymorphic testers. Like the generators they are testing, polymorphic testers treat types as arguments.
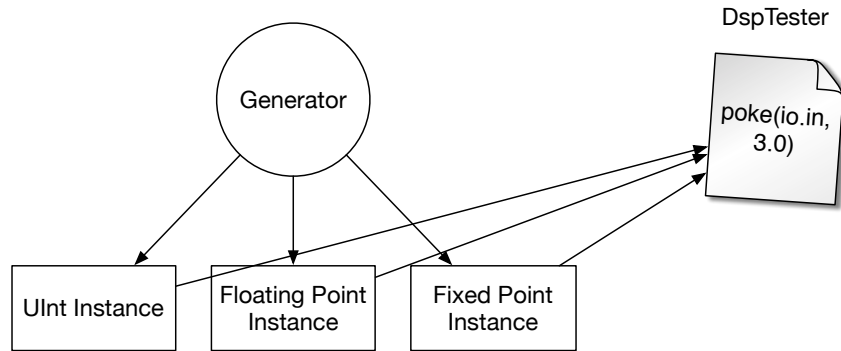
Figure 6.13. Generic tester

Dsptools provides testers with generic flavors of `peek` and `poke` which figure out if the underlying type is fixed point, floating point, real, complex, etc. and perform the correct conversion. Chisel-testers also has a typeclass mechanism for specializing `peek` and `poke` for custom types, which would be suitable for user-defined types. In this way, someone writing a test should make a test generator that is generic with respect to `T` and can be specialized to any type, even user-defined types.

6.3.4   VIPs for Chisel Testers

Verification IPs (VIPs) raise the level of abstraction in verifying a design. VIPs generally allow a test author to generate and gather stimulus and outputs at the transaction level, decoupling what the input and output transactions contain from how they are sent.

Chisel testers provide a low-level API for interacting with DUTs, and because they are written in a powerful programming language it is possible to build powerful abstractions on top of these low level operations. Dsptools implements VIP-style drivers and monitors in Chisel testers for some common interfaces, including AXI-4, AXI-4 Stream, and TileLink[11].

`AXI4StreamModel` is a mixin trait that can be added to a Chisel `PeekPokeTester`. It adds the ability to add VIP-style drivers and slave/monitors to AXI4-Stream interfaces. This lets one specify behavior at a transaction level and not worry about the

---

[11] These VIPs were written before Chisel testers 2, which added primitives for concurrency to Chisel testers. These concurrency primitives make it much easier to write VIPs and provide a safe mechanism that should be preferred to the way Dsptools emulates concurrency. Additionally, since the VIPs were written Chisel has added support for bundle literals, which removes the need for some of the scaffolding in Dsptools's VIPs.

cycle-by-cycle behavior. `AXI4StreamModel` is composed of two more granular traits: `AXI4StreamMasterModel` and `AXI4StreamSlaveModel` which only provide drivers and slaves respectively.

A driver is made by calling `bindMaster(port)` on a top-level AXI4-Stream port, and a slave is made by calling `bindSlave(port)`. After making the master and slave drivers, the test code should enqueue transactions and expected transactions. `step()` will still step the clock, and the drivers will manage the streaming interfaces automatically. `stepUntilCompletion()` will step until every driver is done, or until a programmable timeout.

The `PassthroughTester` in Dsptools is a good example:

```
val master = bindMaster(in)

val slave = bindSlave(out)


// fill queue

master.addTransactions((0 until expectedDepth).map(x =>

    AXI4StreamTransaction(data = x)))

stepToCompletion()


// queue should be full

expect(in.ready, 0)

expect(out.valid, 1)


// empty queue

slave.addExpects((0 until expectedDepth).map(x =>

    AXI4StreamTransactionExpect(data = Some(x))))

stepToCompletion()
```

Because it is such a common pattern to make `DspBlocks` generic with respect to the memory interface, it is important to make it easy to write testers that are also generic with respect to the memory interface. `MemMasterModel` is a mix-in that provides a simple API for reading and writing to memory interfaces via a generic VIP. Mix-ins that specialize `MemMasterModel` to AXI-4, TileLink, etc. flavors are also provided. This allows a test author to write one test that works against AXI-4, TileLink, etc. flavors of the same design. An example is shown below.

```scala
abstract class StreamMuxTester[D, U, EO, EI, B <: Data](c:
    StreamMux[D, U, EO, EI, B]) extends PeekPokeTester(c.module)
  with AXI4StreamModel with MemMasterModel {

  val inMasters = c.ins.map(i=>bindMaster(i.getWrappedValue))
  val outSlaves = c.outIOs.map(o=>bindSlave(o.getWrappedValue))

  // queue up input transactions
  for ((in, inIdx) <- inMasters.zipWithIndex) {
    in.addTransactions(Seq.fill(outSlaves.length)(
    AXI4StreamTransaction(data = inIdx)))
  }

  for (offset <- 0 until c.module.ins.length) {
    // set the input->output mapping
    for (outIdx <- 0 until c.module.outs.length) {
      memWriteWord(c.beatBytes * outIdx, (offset + outIdx) %
          c.module.ins.length)
    }
    // add output assertions
    for ((out, outIdx) <- outSlaves.zipWithIndex) {
      out.addExpects(Seq(AXI4StreamTransactionExpect(data =
          Some((offset + outIdx) % inMasters.length))))
    }
    step(20)
  }
  stepToCompletion(silentFail = true)
}
```

```scala
// Tester for AXI-4 flavor of StreamMux
class AXI4StreamMuxTester(c: AXI4StreamMux) extends
    StreamMuxTester(c) with AXI4MasterModel {
 override def memAXI: AXI4Bundle = c.ioMem.get.getWrappedValue
}


// Tester for TileLink flavor of StreamMux
class TLStreamMuxTester(c: TLStreamMux) extends
    StreamMuxTester(c) with TLMasterModel {
 override def memTL: TLBundle = c.registerNode.in.head._1
}
```

6.3.5 Tape-ins
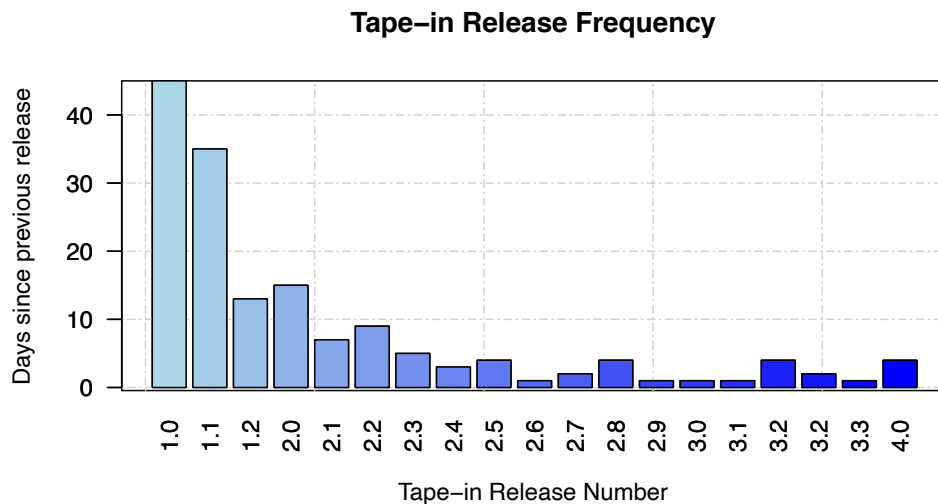
Figure 6.14. Release frequency of tape-ins for CRAFT II project. A version of this figure originally appeared in [19].

One coarse way to measure the agility of a methodology is to see how long it takes to add new features. In [20, 19], RTL dumps were handed off from a design team to a verification and implementation team. Each iteration on an RTL dump was considered a tape-in. Fig. 6.14 depicts the frequency of releases during this project. After initial releases which involved setting up the flow, the time between releases dropped before reaching a steady state of approximately 1-3 days between releases. Of course, not every release represents the same amount of work, but each release did contain new features. The key takeaway is that this methodology was delivering on frequent releases of new features.

6.4 Summary

This chapter presented tools and methodologies for agile development of custom signal processing hardware. Much of this methodology is supported by Dsptools, a Chisel library for this kind of application. These tools and methodologies are applied to the URLLC problem in the next chapter.

Chapter 7

System Concept Demonstration

Demonstrating the feasibility of new ideas via prototyping is an important step in proving the value of new ideas, especially when there are questions as to the accuracy of models and to the feasibility of implementation. Chap. 6 presented tools and methodologies for designing custom signal processing hardware. This chapter shows how to apply these techniques to build prototypes of OFDM-based systems based on those described in Chap. 4.

7.1   Transceiver Modeling

The experiments in Chap. 5 give strong evidence that relay selection algorithms work well in a real-world environment. Still, there is a key question about the validity of the modeling assumptions that needs to be answered: does it actually harness enough diversity to achieve high-reliability communication? There are several difficult-to-model ways that it might fail to achieve high-reliability communication:

- Channels could be correlated in a way that interferes with relay selection, e.g. the relay selection algorithm could fail for different users at correlated times.

- Synchronization error could cause relays to miss their timeslot.

- The control channel for relaying could be too error prone. Relays could incorrectly receive their request to relay.

- The analog frontend may have trouble receiving from multiple relays with guard intervals that are too small, perhaps because of automatic gain control struggling to converge before data is received.

Some of these factors depend on the implementation of the transceivers and the protocols they use to communicate. Building working wireless transceivers can be difficult, and so it might seem desirable to model a transceiver. However, for the high-reliability regime it is important not to sweep sources of error "under the rug" (in a model).

Attempting to create a model that captures everything that could go wrong in something you might build is not a fruitful endeavor. Such a model could be overly pessimistic, or it could be that you miss a critical problem. Building prototypes is one way to cut through these modeling problems.

Replacing models with prototypes creates new challenges. Working prototypes are an existence proof, but non-working prototypes are not a non-existence proof. It is important to have some confidence that the prototype is feasible to build. Another problem with prototypes is that they must exist in the real world, occupy space, and exist in real time. To show that something is reliable may take too much time with a prototype; models are often needed to show reliability.

For the target specifications of this work (30-user network with approximately one communication error per year), it is simply not practical for us to deploy such a large network for the amount of time it would take to demonstrate the targeted reliability. We *must* use a model to allow us to evaluate our system more quickly. The question then arises as to what simplifications can be made. Importance sampling is a popular and powerful technique for investigating rare events. The basic idea is to evaluate a function on a subspace of the sample space, carefully selecting the subspace such that

- The rare events of interest occur more frequently in the subspace than in the entire sample space

- The function's behavior on the subspace can be generalized to the entire sample space (often analytically)

In the case of our low-latency, high-reliability communication system, the function of interest is system error rate of a given system implementation, the sample space is the set of channel realizations. Generally, what is done is to evaluate the system in harsh conditions, for example low SNR, and use a model to extrapolate from the low SNR behavior how the system would behave at nominal SNR.

This type of extrapolation is more troublesome when the model is complex. If the model is too simple, however, it may neglect an important error event that is irrelevant at low reliabilities but dominates at high reliabilities. This is of particular concern when a model is not designed with high reliability in mind. Models designed to be useful in high-throughput scenarios may not capture all of the relevant behavior for URLLC scenarios.

Importance sampling is more dangerous for complicated models because of the danger of neglecting small but critical corner cases. For our relaying-based wireless systems, the model is very complicated, and because there are relays, the entirety of the receive and transmit chains needs to be well understood. Furthermore, there are multiple, simultaneous relays, so the behavior of the network as a whole must be understood (as

opposed to a single node). Therefore, it is best to restrict the modeling effort to a portion of the problem that is well understood, for example the channel, and use a prototype in place of a model where the problem is less well understood.

7.2   Transceiver Prototype

The following sections outline the components of the prototype transceiver [79]. The transceiver is implemented on a Xilinx ZC706, which is a board that includes a Zynq-7000 SoC, memory, and a large amount of I/O connectivity. The Zynq-7000 SoC has both a hard ARM core and programmable logic (PL) for implementing custom hardware. An ADI FMCOMMS-3 daughter card is connected via FMC. ADI has reference designs [14] that have been modified to add a custom baseband implementation as shown in Fig. 7.1.
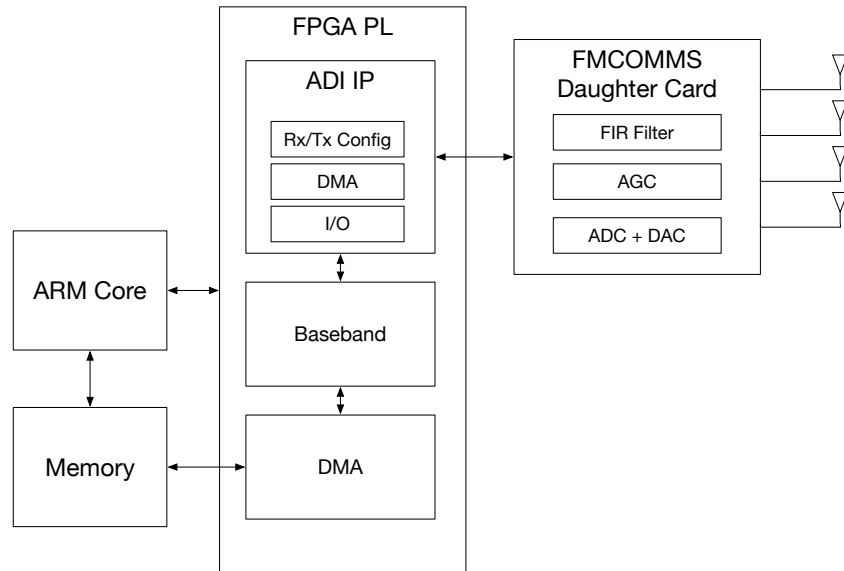
Figure 7.1. The ZC706 FPGA features a hard ARM core with integrated programmable logic (PL) on FPGA.

The ADI reference designs also include a software environment for interacting with the FMCOMMS board. A full desktop Linux operating system is augmented with drivers and application software for configuring the board and IP. Example applications allow for capturing samples, looking at the PSD, changing carrier frequency or sampling bandwidth, etc.

A custom Linux kernel module is written for interacting with the baseband, as depicted in Fig. 7.2. The driver exposes a file /dev/baseband that applications can use to interact with the custom baseband hardware. A set of ioctl calls can be used to con-

figure and interrogate the state of the baseband and the DMA. Once configured, `read` and `write` will trigger the DMA to capture or send samples, respectively.

Similarly, kernel drivers provided by ADI expose files for configuring the FM-COMMS daughter card and ADI IP, for example allowing applications to configure the channels enabled, bandwidth, center frequency, transmit/receive mode, or filtering of the received signal. ADI also provides graphical applications that interact with the drivers to provide an easy to use interfaces for configuring the radio. The typical way we worked with the boards was to use the graphical configuration tool to set up the FM-COMMS board and IP first, and then run a separate application that only interacts with the custom baseband[1].
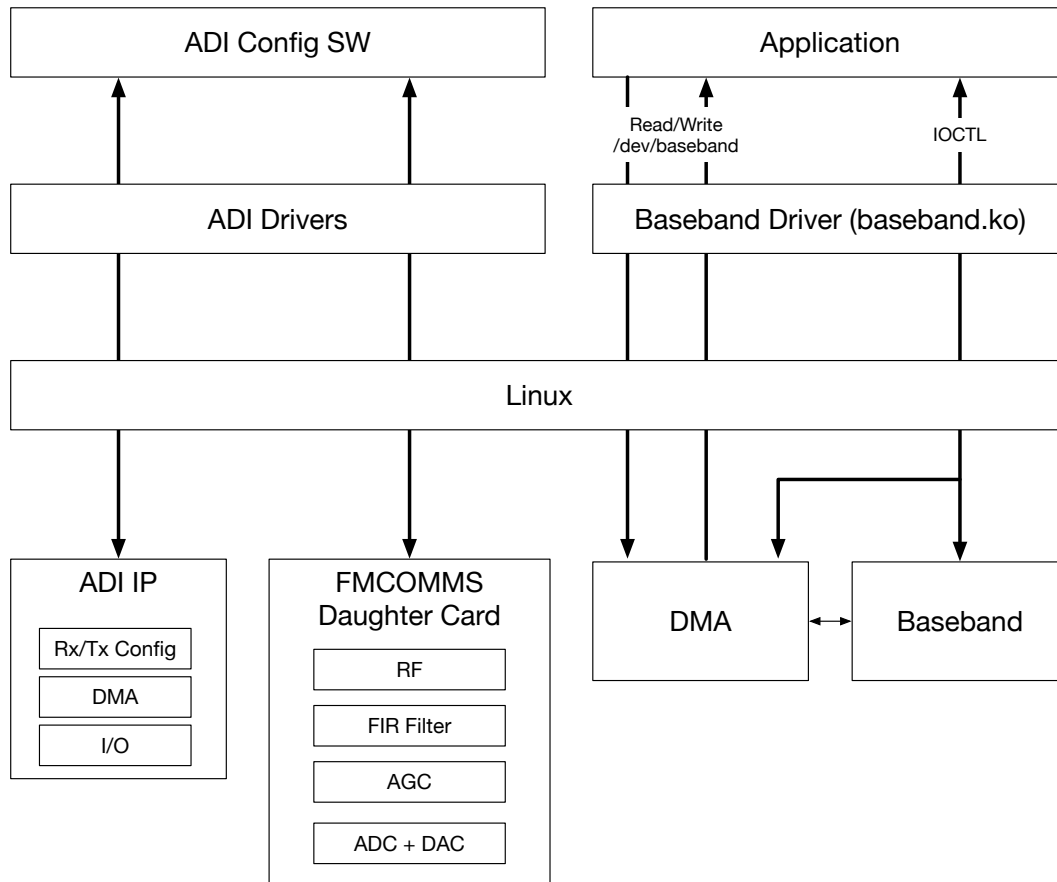
Figure 7.2. A Linux kernel module implements a driver that exposes a file called `/dev/baseband` that allows applications to interact with the baseband.

Fig. 7.3 shows a more detailed view of the custom baseband and DMA. There are two clock domains: the AXI clock domain and the Rx clock domain (which is shared with the Tx). The AXI clock domain runs at 100 MHz and the Rx clock is configurable

---

[1]If dynamic reconfiguration of the ADI daughter card or ADI IP is necessary, for example if frequency hopping is employed, then a single application would need to interact with both the ADI drivers and the custom baseband driver.

at runtime, but for this work is usually 20 MHz. The Rx input stream comes from the ADI IP via an AXI-4 stream interface. It goes through an aligner block first, which is generally unused[2]. The baseband can be configured to bypass most of the receive chain via a programmable splitter and stream mux. If the bypass is enabled, raw samples go to a skid buffer before arriving at the DMA to be dumped into memory. If the bypass is not enabled, samples go through the TimeRx block and FreqRx blocks, and then ultimately a width adapter that packs decoded bits into the word size the DMA expects. The TimeRx block performs all processing (e.g. packet detection, CFO correction, etc.) in the time domain, and the FreqRx block performs an FFT and then performs all processing (e.g. channel estimation, demodulation, etc.) in the frequency domain. The DMA has access to a local scratchpad as well as an AXI-4 master port that masters the SoC's DRAM. The DMA can also be used to stream from memory to an AXI-4 stream master, which feeds into to the Tx block before a clock crossing that ultimately goes to the ADI transmitter.
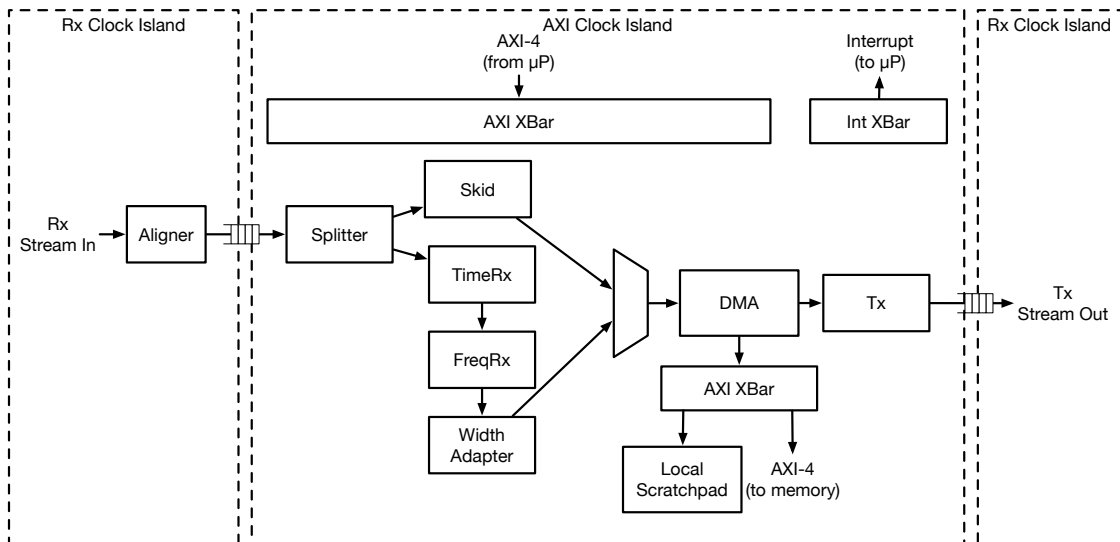
Figure 7.3. Top level of custom baseband.

7.2.1 Time-Domain Portion of Receiver

The TimeRx block is depicted in Fig. 7.4. It performs time synchronization, frequency synchronization (i.e. CFO correction), and removes cyclic prefixes. Synchronization is based on autocorrelation, similar to Matlab's wlanPacketDetect [5]. Because the preamble is periodic, its autocorrelation will have peaks when the overlap is

---

[2]This block is generally only used if the baseband is configured to bypass the TimeRx and FreqRx block and only capture raw samples. The aligner block allows separate captures to begin at the same time within a programmable-length period.

97

close to the period. An autocorrelation block, depicted in Fig. 7.5, outputs both the autocorrelation of the input signal and a running sum of the energy of the signal. These two quantities go to a peak detection block that determines if there is a preamble and where it starts. The output of the autocorrelation block is also fed to a phase estimator, which is used to estimate the CFO.

The autocorrelation block is configurable, both in the sense that the generator is parameterized and in the sense that at runtime it can be reconfigured to detect different preambles. The autocorrelation generator is polymorphic; it will generate autocorrelators for any type that has a `Ring` typeclass. The other parameters include

- `maxApart`: The maximum delay to use for the correlation.

- `maxDepth`: The maximum length of the running sum.

- `addPipeDelay`: The number of pipeline registers to insert after additions.

- `mulPipeDelay`: The number of pipeline registers to insert after multiplications.
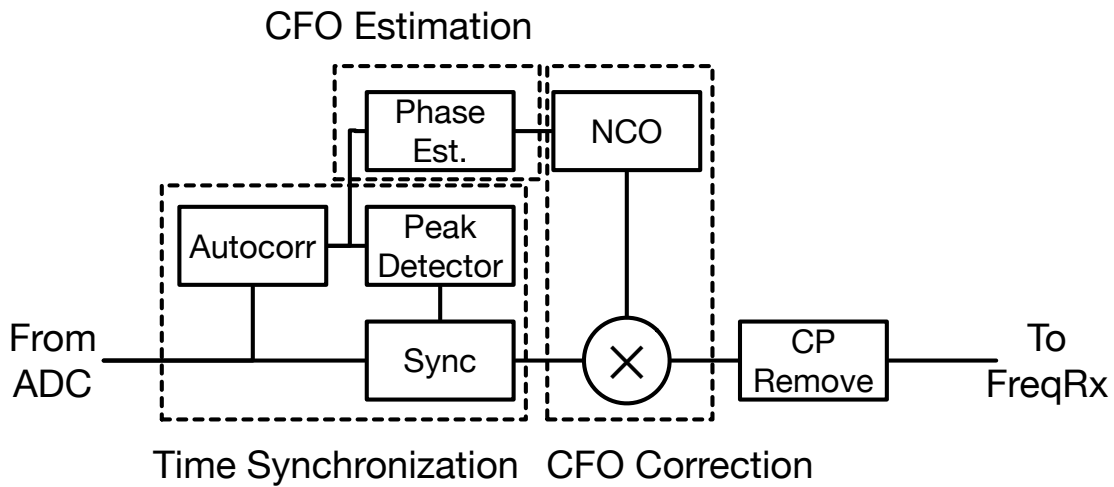
Figure 7.4. TimeRx block.

The output of the autocorrelator goes to a peak detector. The criterion for detecting a peak is runtime-configurable, implementing

$$\|A[n]\|^2 > \alpha \|E[n]\|^2 + \beta$$

where $A[n]$ is the output of the correlation, $\|E[n]\|^2$ is the running sum of the energy[3], and α and β are programmable coefficients determining the threshold. A programmable-length running sum is used to check that a configurable number of peaks are seen within a window of time. This avoids spurious detections.

---

[3]Rescaling $\alpha' = \frac{1}{n^2}\alpha$ allows for using the running average of the energy in a simple way.
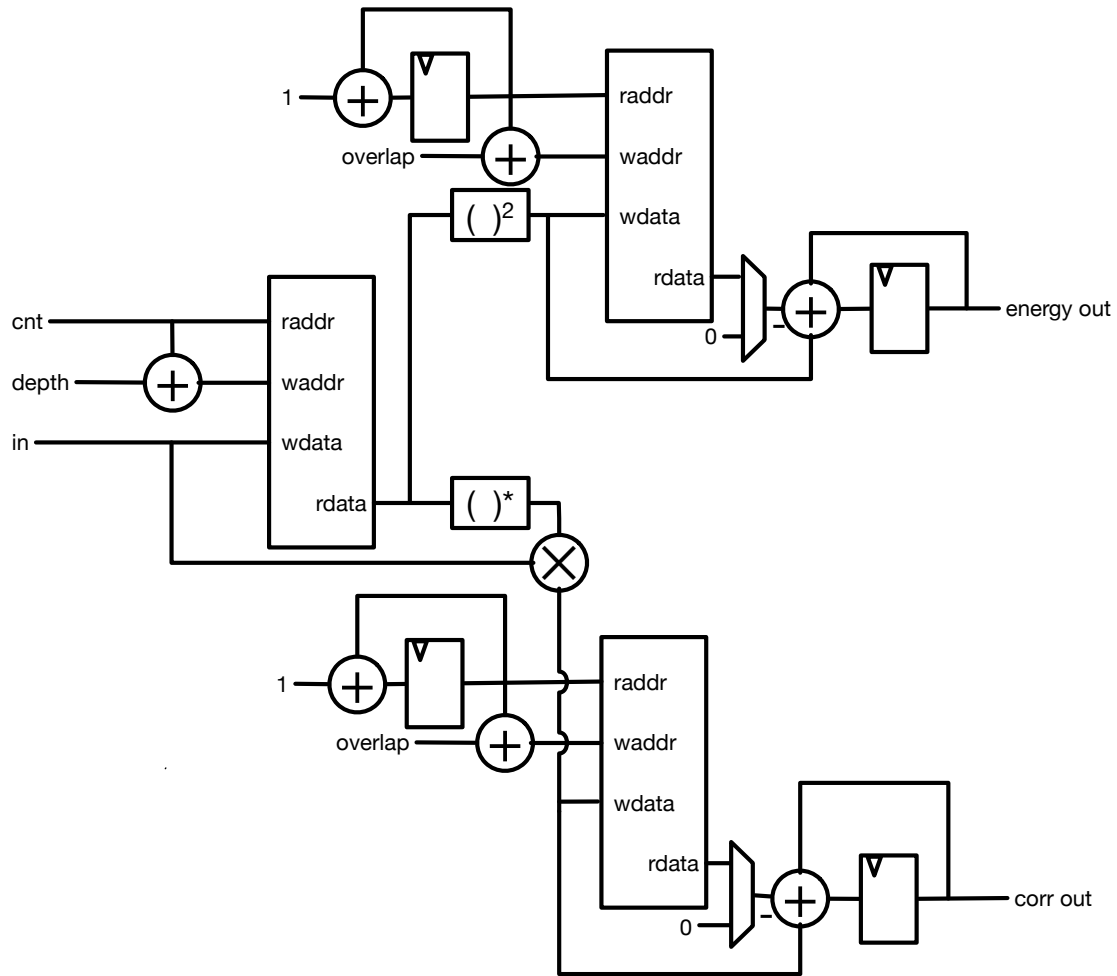
Figure 7.5. Block diagram showing the autocorrelation block. A memory based shift register delays the input data, and the output is split to two different paths. The top path computes the magnitude of the delayed signal and goes through a configurable-length moving average, implemented with a shift register and accumulator that subtracts the output of the shift register. The bottom path computes the conjugate and then multiplies that by the un-delayed input signal before also going through a configurable-length moving average.

Once the peak detector determines that there is indeed a packet, the Sync block will pass the packet to the downstream blocks. When there is no packet detected, it simply accepts all samples and drops them until a packet is found. When there is a packet detected, it drops a runtime-programmable number of samples, passes a runtime-programmable number of samples to the next blocks, and then returns to waiting for another packet. This is implemented as a state machine with a counter.

The output of the autocorrelation block is used in a second path for estimating and correcting CFO. Phase estimation is implemented with a CORDIC block in vectoring mode [117]. The CORDIC algorithm is an efficient way of converting vectors between Cartesian and polar coordinate systems using only shifts, adds, and a small lookup table [15]. In this case, the so-called vectoring mode converts the Cartesian coordinates to polar coordinates, the angular part of which will be used as a phase estimate. The phase of the autocorrelation is directly related to the CFO via

$$\frac{\widehat{\Delta f_n}}{f_c} = \frac{1}{2\pi L} \angle \left(\sum_{i=0}^{N-1} x_{n-i} x_{n-r-L}^* \right),$$

where N is the length of the running sum and L is the autocorrelation lag.

Rather than directly taking the output of the autocorrelation block and feeding it into a CORDIC, the autocorrelation output is averaged first. The averaging is implemented with a simple forgetting factor, i.e. $x_{n+1} = \alpha x_n + y_n$ where x_n are the estimates and y_n is the new autocorrelation output. Choosing $\alpha = 0$ removes all averaging. It is important to do this averaging before performing the phase estimation due to the nonlinear nature of the phase estimation.

There are many variants of CORDIC implementations. CORDICs can perform one of or both vectoring and rotation modes, and via different lookup tables compute hyperbolic functions or square roots. CORDICs can also vary based on how the function is implemented; one broad category is if it is iterative or pipelined, and if it rescales the output.

A CORDIC generator written in Chisel attempts to support this wide range of variants of CORDIC. Both iterative and pipelined CORDICs are supported. A goal of the generator design is that different flavors of CORDIC should re-use as much common code as possible. The core reusable unit of the CORDIC block is called `CORDICStage`, depicted in Fig. 7.6.

`CORDICStage` is generic with respect to the type of the data it operates on and uses typeclasses to provide the necessary operations. The constraints are `T <:Data :`
`Ring : Signed : BinaryRepresentation`. Each typeclass is necessary for:

- `Data`: `T` must be a Chisel type, `Data` is the supertype of all types that can become hardware elements in Chisel.

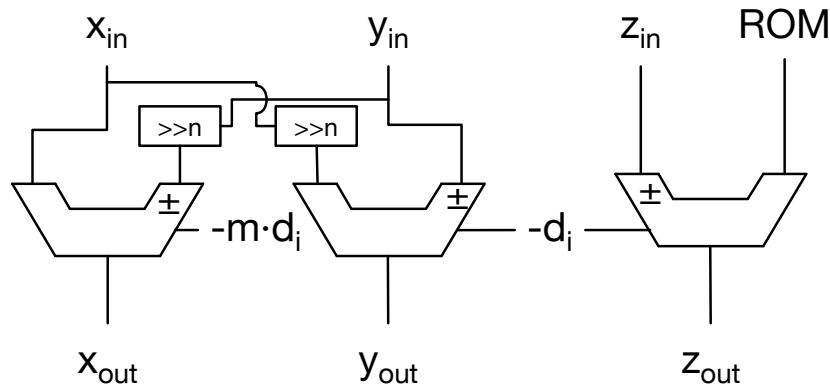- `Ring`: CORDIC requires being able to add and subtract.

Figure 7.6. Block diagram for a CORDIC stage. n is the shift amount, which is an input to the block. For some parameterizations, n may be fixed for a stage, but it is not fixed generally. m and d_i and functions that may vary with the selected mode of the CORDIC. The design of the generator allows for these to be implemented as generic functions of the inputs, and also allows more inputs to be added (e.g. a select between vectoring and rotation).

- `Signed`: CORDIC requires being able to check the sign of inputs and the residue.

- `BinaryRepresentation`: CORDIC divides by powers of 2 via shifts. This typeclass provides an interface for shifting numbers.

A stage has x_{in}, y_{in}, z_{in} as inputs and $x_{out}, y_{out}, z_{out}$ as outputs. There is another input for the value coming out of the ROM. If the stage is being used in a setting where it can switch between different functions (for example trigonometric and hyperbolic), the value of the ROM may be changed externally based on which function is being computed. If the stage is being used in a pipelined setting, the ROM input may be a fixed value. The is also an input for the shift amount, which in a pipelined setting may also be fixed.

The final input is a config object. The idea of the config object is that in some cases, the CORDIC will need to be configured differently to do different functions. For example, if the CORDIC can switch between vectoring and rotation modes, there should be an input setting which mode is being used, and there should be logic changing the behavior based on what the input says. The config object provides both of these. There is a portion that is a (potentially empty) bundle that, when populated, provides fields for determining the mode. Then there is another portion that provides functions that generate the logic that consumes these inputs.

Another important portion of the CORDIC generator is the generation of the ROM values. The code that generates the ROM's values requires another typeclass: `CovertableTo`. This typeclass describes how to generate hardware for literal values, for example how to represent 1.45 as a generic type `T`. For the trigonometric and hyperbolic functions, these ROMs are generated using the `fromDouble(value, proto)` function of the `ConvertableTo` typeclass. This constructs an instance of `T` with literal

value `value` and which matches width with a prototype `proto`. This ensures that all values in the table have the same type/width. A `SyncROM` is used to generate a ROM given the values generated by `fromDouble`.

`SyncROM`[4] is a way of generating ROMs that map well to FPGAs. It makes use of Chisel's `BlackBox` facility because the way Chisel natively generates Verilog is not very efficient for ROMs. Chisel's normal way of generating a ROM using a `Vec` generates a lot of temporary values and if statements. This does not scale well for large ROMs, which ideally would get mapped to a BRAM on Xilinx IP. `SyncROM` generates a `BlackBox` which follows the Xilinx template for BRAM-mapped ROMs (using a case statement and a registered output). This is generated with the `HasBlackBoxInline` facility in Chisel which supports annotating `BlackBoxes` with a string that will get written to a separate Verilog file as part of the FIRRTL compilation flow. This string is programmatically generated from the values generated by `ConvertableTo`. A dot-f file is also generated by this flow, which simplifies the integration of these generated blackboxes into a build system[5].

The CORDIC stage is then wrapped by a top level generator. For a pipelined CORDIC, many stages are laid out, some with pipeline registers between them. For the iterative CORDIC used in this design, one stage and one ROM are generated along with a register to store x, y, z and a state machine to configure the CORDIC stage and update the registers. In this case, only the phase is used so no gain correction factor is applied. A top-level API is provided so that a CORDIC can be constructed as follows:

```
val cv = IterativeCORDIC.circularVectoring(protoXY, protoZ)

val cr = IterativeCORDIC.circularRotation(protoXY, protoZ)
```

Once the phase is estimated, it is multiplied by a programmable coefficient to convert it into an estimate for frequency correction that must be applied to remove the CFO. This frequency estimate is fed into a numerically controlled oscillator (NCO) that modulates the input stream. The NCO uses a quarter-wave sine table implemented once again using the `SyncROM` construct. The NCO has two outputs, one 90° out of phase from the other so the output can be interpreted as a complex exponential. A Taylor approximation (number of terms is a parameter) is performed to interpolate between entries in the table.

The output of the NCO modulates (via a complex multiply) the input samples once a packet is detected. Subsequently, a simple state machine with programmable size cyclic prefix and FFT strips the prefix and passes the data symbol through to the FreqRx block.

---

[4]`SyncROM` was based on earlier work by Angie Wang for generating efficient LUTs for ASICs. This implementation maps to BRAMs on Xilinx FPGAs instead, and uses the `HasBlackBoxInline` facility to simplify the build system.

[5]This is especially useful for when the number or name of the black boxes changes with different parameterizations. It's a good practice to include serialized parameters in the name of black boxes to avoid namespace collisions, so this is very common to have happen in practice. Having to rewrite build scripts every time a parameter changes is a waste of time and energy, so it is important to have the backend implementation and verification flows be flexible to parameterization changes.

7.2.2 Frequency-Domain Portion of the Receiver

The first step of the frequency-domain portion of the receiver is the FFT. The FFT is a radix 2^2 single-path delay feedback FFT [93].
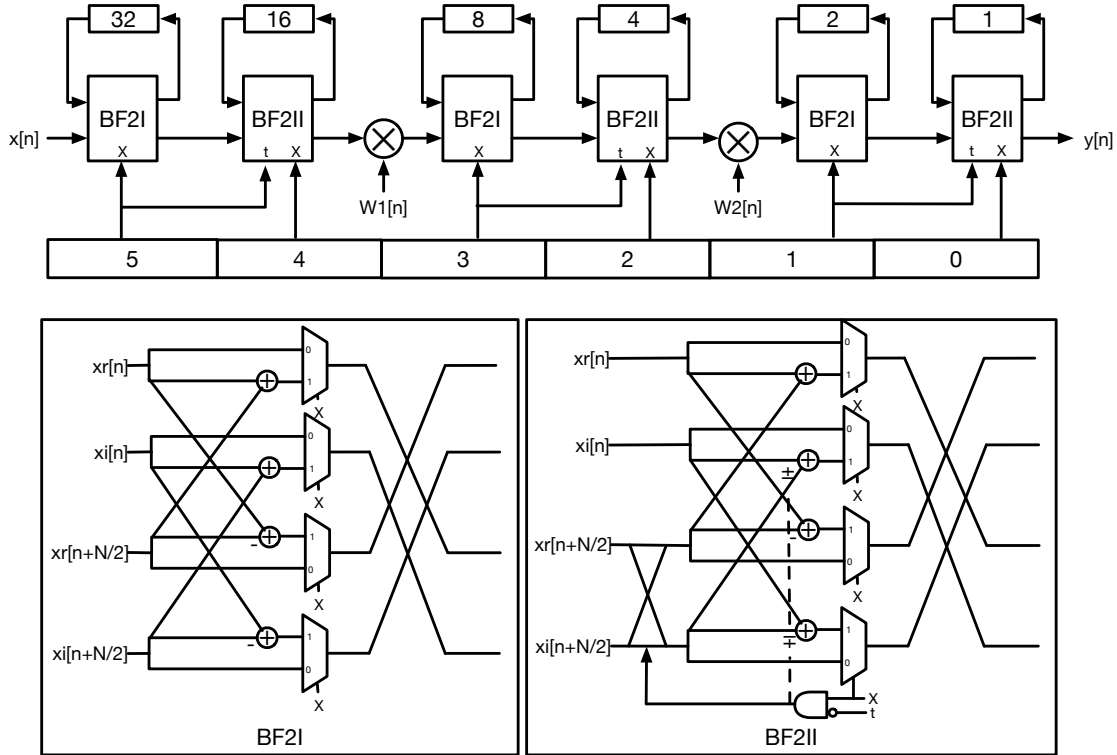
Figure 7.7. Block diagram of radix-2^2 single-path delay feedback FFT [93]. A size 64 FFT is pictured here.

One interesting aspect of implementing this FFT in Chisel is how to manage bit-growth. Depending on the input statistics, different bit-growth (or normalization) techniques are used, but one common rule of thumb is to add a bit every other stage of a radix-2 FFT. Implementing these kinds of rules takes some care when writing a type-generic generator because the underlying representation is abstracted.

In this FFT generator, the `proto` (i.e. a variable that contains the prototype for constructing a new instance of type `T`, including width information) is tracked from stage to stage, right alongside the control signals, inputs, and outputs. A new `proto` is computed every other stage by doing `(x context_+ x).cloneType`, where `x` is some concrete hardware instance constructed from `proto`. Using `context_+` ensures that the `DspContext` setting for how addition should be performed (e.g. with or without bit-growth) will be used to determine how bit-growth should be managed. For example, if addition is saturating, then no bit-growth will occur. Furthermore, for fixed-width types such as floating point where it does not make sense to add extra bits, no extra bits

103

will be added. However, in cases where addition does add extra bits, performing the addition and calling `cloneType` will give a new `proto` that has the right number of bits. Computing new protos by performing operations on them is generally preferable to special-casing each type that the generator supports.

After the FFT, channel estimation is performed. Pilot subcarrier locations are given as a parameter to the channel estimation generator, and a state machine keeps track of which samples are pilots and which are data. Each subcarrier corresponding to a pilot is put in a queue that goes through an estimator block. The estimator block implements

$$\hat{y} = \frac{x^*}{\|x\|^2},$$

where x is the received pilot and $\hat{y}$ is an estimate of the correction factor to invert the channel.

The magnitude function is implemented with a complex multiply, and the divider is implemented using non-restoring division. One important aspect of the divider is that it operates directly on `UInt`s. This is because the non-restoring division algorithm relies on an alternative number representation where bits represent one of $\{-1, 1\}$. Non-restoring division allows for efficient implementation, but does so by exploiting the number representation in a way that Dsptools's typeclasses abstracts away. The `BinaryRepresentation` typeclass provides some ability to expose a limited API for doing things like division via powers of two by shifting, but this API is intentionally limited to prevent baking in too many assumptions about the underlying representation. Non-restoring division performs shifting, sign bits, and bitwise negation, so it was written using `UInt`s rather than generic type `T` with typeclasses. The channel estimator block is responsible for converting `T` to `UInt` and back. In the case of the `FixedPoint` implementation that ultimately gets implemented, this involves manually doing some bookkeeping for the binary point.

On the output side, each subcarrier corresponding to a pilot will output the coefficient for correcting the channel. Each subcarrier corresponding to data will simply pass through the data.

The output of the channel estimation block goes to the equalizer block. Subcarriers that contain pilots are treated separately from subcarriers that contain data. Each data subcarrier is multiplied by a coefficient corresponding to the neighboring pilot subcarriers (which now contain the estimated correction factor). The coefficients are sinc interpolations of the neighboring coefficients.

The output of the channel equalization block is fed into a block that removes the now unneeded pilot subcarriers, and then is fed into a demodulation block. The demodulation block implements simple soft QPSK demodulation where the values of the log-likelihood ratios (LLRs) corresponds to the real and imaginary part of the symbol.

The final block in the `FreqDomainRx` is a deserializer which bundles all the LLRs into words that match the DMA's width. The output of the receive chain feeds into the

DMA, which can efficiently move the stream into main memory for further processing by the ARM CPU. In particular, FEC and header parsing can occur, after which enough information is known as to whether a relay transmission needs to be scheduled.

7.2.3 Transmitter

For prototyping purposes, the transmitted signal can be precomputed. The baseband has a scratchpad memory that can store the precomputed transmit signal to ensure there is no contention for main memory between the receive and transmit chains.

It is critical that transmissions be able to be scheduled very precisely. To enable this, a time gate block is placed between the DMA streaming output and the transmit DAC output. The time gate applies backpressure until a counter in the DAC clock domain reaches a programmable value, which is interpreted as the desired transmit time. After the counter reaches the desired value, the backpressure is removed and the output flows from DMA to DAC.

7.3 Verification

Unit tests were implemented using Chisel-testers and the Dsptools extensions that allow for testing type-generic generators with typeclasses. Unit tests for some `DspBlocks` that use the VIP-style drivers and monitors are also implemented.

It is important to test that all the pieces function correctly once they are integrated. Clock crossings are particularly important because Chisel-testers model is largely single clock[6].

Furthermore, some of the advantages Chisel has for testing related to introspection and the type system were less useful for top-level integration testing. Xilinx IP used a different naming convention and different subset of AXI signals than Chisel's Rocketchip-based AXI implementations. To ease integration with the Xilinx IP, the top-level Chisel design did not use Rocketchip's AXI `Bundle` types, but instead used different types that used Xilinx's naming convention. Using different bundle types makes it difficult to use the VIPs that rely on being able to introspect the design for parameters and expect the bundle to have the expected types and names.

For some top-level integration tests, SystemVerilog was used. The Xilinx AXI-4 and AXI-4 Stream VIPs were used to generate stimulus and check outputs. A simple test that checked the DMA worked to send and receive samples was implemented.

This test was not enough to ensure the DMA worked with the memory subsystem of

---

[6]Testers-2 [2] has good support for multiple clocks and concurrency, but was not mature at the time this verification work started.

the FPGA, however. The AXI-4 memory model was more permissive than the AXI-4 port to the memory controller on the FPGA. The ChipScope Integrated Logic Analyzer (ILA) was instrumental in debugging the DMA for this.

More top-level integration tests of the baseband were implemented using cocotb [4]. Cocotb is a Python-based library for writing testbenches. It uses Python coroutines to model concurrent events in a simulation, and uses Python's powerful and flexible language constructs to make writing simulation components easy. Furthermore, because of Python's powerful standard library and wider ecosystem, a lot of models can be created by stitching together library components. For projects like this involving wireless systems, the NumPy [71] and SciPy [116] projects can be particularly useful.

One problem to overcome when using cocotb with this project had to do with naming. Similarly to chisel-testers, cocotb drivers and monitors expect certain naming conventions for the buses they model. A small modification to cocotb enabled monitors and drivers to perform logical to physical name mapping. For example, an AXI-4 driver that internally calls the write address channel `AWADDR` could be connected to a bus that calls the same port `mem_awaddr` with

```
AXI4Driver(dut, "mem", clock, AWADDR="awaddr")
```

This enabled cocotb to use existing drivers and monitors with signals that were named with Xilinx naming conventions[7].

The testing setup is depicted in Fig. 7.8. The testbench has functions for configuring the many blocks within the baseband. In particular, functions that perform DMA actions are used to initiate transmission and reception. A C model of the transmitter is used via SWIG [106] to populate the memory with a transmission, and ultimately transmit a signal via a DMA memory-to-stream action. The transmission ultimately leaves via the `dac` interface, goes through a channel model, and then arrives at the `adc` interface to be processed by the receive portion of the baseband. The DMA writes this reception to the memory via a stream-to-memory action, whereupon the testbench checks the reception.

7.4   Implementation

The custom baseband described above is included as an IP block [80] in the ADI reference design [14]. A TCL script removes connections between the FMCOMMS ADC and DAC with DMAs, inserting the custom baseband in-between. The reference design includes drivers and applications for interacting with the design that function as

---

[7]Name mapping functionality could be added to chisel-testers, but the testers perform type checking and introspect on those types to extract parameters. It would require a substantial reimagining of the testers API to change this, and removing the ability to introspect those types safely removes much of the utility of chisel-testers.
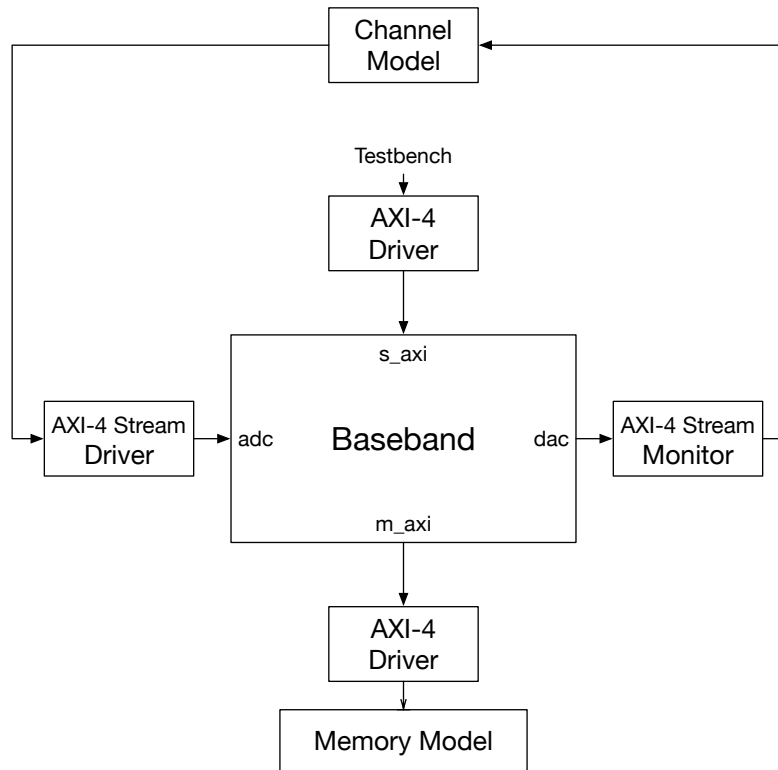
Figure 7.8. Block diagram of the cocotb simulation environment. The baseband DUT has four interfaces that have VIPs connected to them. The AXI-4 Stream interfaces have a driver at the ADC input and a monitor at the DAC output, and are connected by a loopback that applies a software model of a channel. The AXI master interface (`m_axi`), which in the complete system would master a bus connected to main memory, is connected to a VIP with a simple model of a memory. This memory has a mechanism for backdoor access by the testbench. The AXI slave interface (`s_axi`) has a VIP which serves as the interface by which the testbench stimulates the design.

before after adding the baseband design[8]. The TCL script also adds the custom baseband to the memory map, and the Linux devicetree is modified to include entries for the custom baseband[9].

The time-domain and frequency-domain portions of the receiver are assigned to their own pblocks and manually placed to help ensure good QoR. The resulting utilization is given in Table 7.1.

Name	Slice LUT	Slice LUT % of Baseband
TimeRx	1714	6.9%
FreqRx	15228	61.4%
Scratchpad	4376	17.6%
DMA	1037	4.2%
Busses	690	2.8%
Clock crossings	816	3.3 %
Misc.	912	3.7%
Total	24773	100%

Table 7.1. FPGA utilization (post-implementation).

7.5   Summary

This chapter presented a number of useful blocks for building custom basebands for OFDM-based systems. Techniques discussed in Chap. 6 were applied in to design and verify blocks useful for OFDM-based systems.

---

[8]The reference design's ADC-to-DMA connections are not changed, the baseband is simply a second, passive consumer of the ADC stream. The transmit side of the baseband passes everything through by default; a register needs to be set to enable the baseband to do its own transmissions.

[9]Rocketchip has functionality for generating devicetrees, but unfortunately it is not used here. One reason is that AXI peripherals do not generate devicetree information in Rocketchip.

Chapter 8

Conclusion

Many interesting ideas require custom hardware prototyping to evaluate their usefulness. Communications schemes that use relaying to achieve ultra-reliable low-latency communication are one such example, and there are a wide array of other applications that demand custom signal processing hardware.

This work presents the journey of building tools useful for prototyping an ultra-reliable low-latency wireless system. One part of that journey is designing the communication scheme and making sure that it is both realizable and able to meet the target requirements. The other part of that journey is building tools and methodologies to build a prototype system as the communication scheme evolved. This thesis contributes the following:

1. An implementable system design for relay-based URLLC

2. Methodologies and tools for developing and verifying custom signal processing hardware generators. Dsptools is a Chisel library that enables much of this methodology.

3. Implementation of baseband components for realizing OFDM-based communication.

Chapter 1 introduces the URLLC problem and discusses the prototyping with custom hardware. Chapter 2 presents background information about URLLC, 5G standardization, channel modeling, as well as a review of hardware design and verification practices, especially design automotation that promotes agile design. Chapter 3 presents relay-based URLLC techniques in detail, along with commentary on their relative merits.

Chapter 4 discusses PHY and MAC level implementation details for the relay-based URLLC techniques discussed earlier. In particular, OFDM is discussed in the context

of URLLC, as well as requirements for synchronization, channel estimation and equalization, and coding.

Chapter 5 discusses the viability of relay selection techniques that are motivated by the desire for simple implementation. Analysis, simulation, and measurement data establish the efficacy of dynamic relay selection.

Chapter 6 presents a generator-based methodology for the design and verification of custom signal processing hardware. Dsptools, Chisel language features, and methodologies are discussed in a more general context, and then are applied to URLLC in Chapter 7. There is a discussion of design decisions for the prototype, as well as a discussion of the methodology that was applied for the design.

Exploring system ideas and developing tools and methodologies for enabling that exploration creates a virtuous cycle. The exploration is aided by new tools and methodologies, and the tools and methodologies are made more useful by having a real problem to be applied to.

Ultimately, these tools and methodologies are useful steps towards a future where domain experts can efficiently explore design spaces with hardware prototypes. For wireless systems, this would allow protocol designers to specify (or change) a system at a high level and quickly generate (or regenerate) high quality hardware prototypes to evaluate the system.

8.1   Future Work

These communication schemes are by no means ready to be deployed in a production environment. One interesting area for future work would involve trying to map these ideas about relaying into existing standards. For simplicity, many network functions such as joining and leaving, authentication, and coexistence have been neglected from being fully addressed by this work.

There are numerous opportunities for future work to improve on these tools and methodologies. Developing a larger portfolio of reusable signal processing blocks for the Chisel ecosystem is one important component. Efforts to improve the overall verification environment in Chisel are being undertaken by others in the community, and should continue to provide functionality to support signal processing applications. In particular, native support for concurrency in the Chisel verification environment enables UVM-style verification, which the VIPs developed for `DspBlocks` would benefit from.

FIRRTL has recently evolved to have more support for fine-grained scheduling of transforms and to have transforms with complex dependencies on each other. This functionality enables a new style of Chisel generator which is especially useful for signal processing where specific microarchitectures for blocks are chosen after other parts of the system are elaborated, for example an FIR filter implementation could be chosen to rate-match the blocks before and after the filter. This style of development enables

optimizations that have been available to HLS compilers but have been more difficult to apply for generators.

Another direction for exciting future work is in using diplomacy to manage data movement. Diplomacy provides many powerful tools; `regmapper` in particular is very useful for `DspBlock`s, as well as generating bus structures and managing clock crossings and reset. However, there are fewer tools for generating bus masters, especially things that look like specialized DMAs. A stream ↔ AXI-4 DMA was written for this design, but it could be made to be more flexible and general. Making an API like `regmapper` for bus masters would be an interesting and useful thing to build.

More abstractly, this work is part of a larger movement for using specialized compilers for increased productivity. This kind of work seeks to empower domain experts with tools that better reflect the mental model employed within a domain, and by so doing allow simple conceptual changes to actually be simple to implement. By allowing designers to be more productive, teams can be smaller, move faster, and by trying more ideas find the best ones. Furthermore, more productive hardware design makes designing hardware more valuable and more accessible.

Bibliography

[1] "Xilinx Intellectual Property." [Online]. Available: https://www.xilinx.com/products/intellectual-property.html

[2] "Chiseltest," 2018. [Online]. Available: https://github.com/ucb-bar/chisel-testers2

[3] "Firrtl interpreter," 2018. [Online]. Available: https://github.com/freechipsproject/firrtl-interpreter

[4] "cocotb," 2020. [Online]. Available: https://github.com/cocotb/cocotb

[5] "OFDM packet detection using L-STF," 2020. [Online]. Available: https://www.mathworks.com/help/wlan/ref/wlanpacketdetect.html

[6] G. T. 21.915, "Release 15 Description," 2017.

[7] G. T. 21.916, "Release 16 Description," 2018.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[9] A. Abdi, K. Wills, H. Barger, M.-S. Alouini, and M. Kaveh, "Comparison of the level crossing rate and average fade duration of Rayleigh, Rice and Nakagami fading models with mobile channel data," *Vehicular Technology Conference Fall 2000. IEEE VTS Fall VTC2000. 52nd Vehicular Technology Conference (Cat. No.00CH37152)*, vol. 4, pp. 1850–1857, 2000. [Online]. Available: http://ieeexplore.ieee.org/document/886139/

[10] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung, "Network information flow," *IEEE Transactions on information theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[11] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[12] Z. Alliance, "ZigBee PRO specification," *Standard, Oct*, 2007.

[13] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.

[14] Analog Devices Inc., "HDL Reference Designs," 2014. [Online]. Available: https://github.com/analogdevicesinc/hdl

[15] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, 1998, pp. 191–200.

[16] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[17] C. Baaij, "CλasH: From haskell to hardware," Master's thesis, University of Twente, 2009.

[18] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[19] S. Bailey, P. Rigge, J. Han, R. Lin, E. Y. Chang, H. Mao, Z. Wang, C. Markley, A. M. Izraelevitz, A. Wang, *et al.*, "A Mixed-Signal RISC-V Signal Analysis SoC Generator With a 16-nm FinFET Instance," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 10, pp. 2786–2801, 2019.

[20] S. Bailey, J. Han, P. Rigge, R. Lin, E. Chang, H. Mao, Z. Wang, C. Markley, A. Izraelevitz, A. Wang, *et al.*, "A Generated Multirate Signal Analysis RISC-V SoC in 16nm FinFET," in *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2018, pp. 285–288.

[21] M. Bossert, A. Huebner, F. Schuehlein, H. Haas, and E. Costa, "On cyclic delay diversity in ofdm based transmission schemes," in *OFDM workshop*, vol. 2, 2002.

[22] Cadence, "Stratus high-level synthesis." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[23] C. Celio, P.-F. Chiu, B. Nikolić, D. A. Patterson, and K. Asanovic, "Boomv2: an open-source out-of-order RISC-V core," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[24] G. Cena, A. Valenzano, and S. Vitturi, "Hybrid wired/wireless networks for real-time communications," *IEEE industrial electronics magazine*, vol. 2, no. 1, pp. 8–20, 2008.

[25] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[26] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[27] O. I. de Normalización, *ISO 11898: Road Vehicles: Interchange of Digital Information: Controller Area Network (CAN) for High-speed Communication.* ISO, 1993.

[28] J. Dean, "1.1 The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC).* IEEE, 2020, pp. 8–14.

[29] J. Decaluwe, "MyHDL: a Python-Based Hardware Description Language." *Linux journal*, no. 127, pp. 84–87, 2004.

[30] M. Eisen, M. M. Rashid, K. Gatsis, D. Cavalcanti, N. Himayat, and A. Ribeiro, "Control Aware Communication Design for Time Sensitive Wireless Systems," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE, 2019, pp. 4584–4588.

[31] S. Eldridge, A. Buyuktosunoglu, and P. Bose, "Chiffre: A Configurable Hardware Fault Injection Framework for RISC-V Systems."

[32] G. P. Fettweis, "The Tactile Internet: Applications and Challenges," *IEEE Vehicular Technology Magazine*, vol. 9, no. 1, pp. 64–70, mar 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6755599

[33] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 286–291.

[34] P. W. Fink, P. S. Foo, and W. H. Warren, "Catching fly balls in virtual reality: A critical test of the outfielder problem," *Journal of vision*, vol. 9, no. 13, pp. 14–14, 2009.

[35] M. Fowler, J. Highsmith, *et al.*, "The agile manifesto," *Software Development*, vol. 9, no. 8, pp. 28–35, 2001.

[36] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *arXiv preprint arXiv:1911.09925*, 2019.

[37] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5G evolution: A view on 5G cellular technology beyond 3GPP release 15," *IEEE Access*, vol. 7, pp. 127 639–127 651, 2019.

[38] A. Giridhar and P. R. Kumar, "Distributed clock synchronization over wireless networks: Algorithms and analysis," in *Proceedings of the 45th IEEE Conference on Decision and Control.*   IEEE, 2006, pp. 4915–4920.

[39] GNU Radio, July 2020. [Online]. Available: https://www.gnuradio.org

[40] V. C. Gungor and G. P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *IEEE Transactions on industrial electronics*, vol. 56, no. 10, pp. 4258–4265, 2009.

[41] A. Gupta, L. A. Rigge, D. A. Brooks, B. J. Dutt, R. L. Freyman, G. Komoriya, C. R. Miller, and R. N. Kershaw, "A Design Rule Updatable Layout for A Single Chip 32 Bit CMOS Floating Point DSP," in *VLSI Signal Processing, III*, R. W. Brodersen and H. S. Moscovitz, Eds.   IEEE, 1988, ch. 29, pp. 307–318.

[42] "IEEE Standard for Standard SystemC Language Reference Manual - Redline," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline*, 2012.

[43] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, *et al.*, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.   IEEE, 2017, pp. 209–216.

[44] W. C. Jakes and D. C. Cox, *Microwave mobile communications*.   Wiley-IEEE Press, 1994.

[45] H. Ji, S. Park, J. Yeo, Y. Kim, J. Lee, and B. Shim, "Ultra-reliable and low-latency communications in 5G downlink: Physical layer aspects," *IEEE Wireless Communications*, vol. 25, no. 3, pp. 124–130, 2018.

[46] Y. Jiang, H. Kim, H. Asnani, S. Kannan, S. Oh, and P. Viswanath, "Learn codes: Inventing low-latency codes via recurrent neural networks," *IEEE Journal on Selected Areas in Information Theory*, 2020.

[47] Y. Jing and H. Jafarkhani, "Single and multiple relay selection schemes and their achievable diversity orders," *IEEE Transactions on Wireless Communications*, 2009.

[48] T. Joas, P. Rigge, and B. Nikolić, "Towards a RISC-V platform with integrated radio IP, Report," 2018. [Online]. Available: https://github.com/timoML/zephyr-riscv/blob/cow/apps/irqperipheral_test/doc/report/Report_Towards_a%20RISC-V_platform_with_integrated_radio_IP.pdf

[49] R. Jurdi, S. R. Khosravirad, and H. Viswanathan, "Variable-rate ultra-reliable and low-latency communication for industrial automation," in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*.   IEEE, 2018, pp. 1–6.

[50] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "Xors in the air: Practical wireless network coding," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006, pp. 243–254.

[51] A. N. Kim, F. Hekland, S. Petersen, and P. Doyle, "When HART goes wireless: Understanding and implementing the WirelessHART standard," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. IEEE, 2008, pp. 899–907.

[52] D. Kim, C. Celio, D. Biancolin, J. Bachrach, and K. Asanovic, "Evaluation of RISC-V RTL with FPGA-accelerated simulation," in *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[53] D. Kim, A. Izraelevitz, C. Celio, H. Kim, B. Zimmer, Y. Lee, J. Bachrach, and K. Asanovicc, "Strober: fast and accurate sample-based energy simulation for arbitrary RTL," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 128–139.

[54] J. Knibbe, H. Benko, and A. D. Wilson, "Juggling the effects of latency: motion prediction approaches to reducing latency in dynamic projector-camera systems," Microsoft Research Technical Report MSR-TR-2015–35, Tech. Rep., 2015.

[55] J. N. Laneman, D. N. Tse, and G. W. Wornell, "Cooperative diversity in wireless networks: Efficient protocols and outage behavior," *IEEE Transactions on Information theory*, vol. 50, no. 12, pp. 3062–3080, 2004.

[56] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 2004, pp. 75–86.

[57] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, *et al.*, "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[58] Z. Li, M. A. Uusitalo, H. Shariatmadari, and B. Singh, "5G URLLC: Design challenges and system concepts," in *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*. IEEE, 2018, pp. 1–6.

[59] L. Liu and W. Yu, "A D2D-based protocol for ultra-reliable wireless communications for industrial automation," *IEEE Transactions on Wireless Communications*, vol. 17, no. 8, pp. 5045–5058, 2018.

[60] W. Liu, P. Popovski, Y. Li, and B. Vucetic, "Wireless networked control systems with coding-free data transmission for industrial iot," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 1788–1801, 2019.

[61] D. Lockhart, S. Twigg, D. Hogberg, G. Huang, R. Narayanaswami, J. Coriell, U. Dasari, R. Ho, D. Hogberg, G. Huang, A. Kane, C. Kaur, T. Liu, A. Maggiore, K. Townsend, and E. Tuncer, "Experiences Building Edge TPU with Chisel," https://chisel-community-conference.org/ccc18, 2018, Chisel Community Conference. [Online]. Available: https://www.youtube.com/watch?v=x85342Cny8c

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

[63] M-Labs, "nmigen." [Online]. Available: https://github.com/m-labs/nmigen

[64] A. Magyar, D. Biancolin, J. Koenig, S. Seshia, J. Bachrach, and K. Asanovic, "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes." in *ICCAD*, 2019, pp. 1–8.

[65] MathWorks, "HDL Coder." [Online]. Available: https://www.mathworks.com/products/hdl-coder.html

[66] Mentor, "Catapult high-level synthesis." [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[67] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 2004, pp. 69–70.

[68] S. Nsaibi, L. Leurs, and H. D. Schotten, "Formal and simulation-based timing analysis of Industrial-Ethernet sercos III over TSN," in *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2017, pp. 1–8.

[69] Nvidia, "Nvidia Deep Learning Accelerator." [Online]. Available: http://nvdla.org/

[70] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," Tech. Rep., 2004.

[71] T. Oliphant, "NumPy: A guide to NumPy," USA: Trelgol Publishing, 2006–, [Online; accessed <today>]. [Online]. Available: http://www.numpy.org/

[72] C. Papon, "Spinalhdl," 2014.

[73] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[74] E. Perahia and R. Stacey, *Next generation wireless LANs: 802.11 n and 802.11 ac*. Cambridge university press, 2013.

[75] Y. Polyanskiy, H. V. Poor, and S. Verdú, "Channel coding rate in the finite block-length regime," *IEEE Transactions on Information Theory*, vol. 56, no. 5, pp. 2307–2359, 2010.

[76] Profibus, July 2020. [Online]. Available: https://www.profibus.com

[77] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[78] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[79] P. Rigge, "OFDM," 2020. [Online]. Available: https://github.com/grebe/ofdm

[80] P. Rigge and R. Avižienis, "adi-bbp," 2017. [Online]. Available: https://github.com/grebe/adi-bbp

[81] P. Rigge and C. Nelson, "XYTable," 2018. [Online]. Available: https://github.com/grebe/xytable

[82] P. Rigge and B. Nikolić, "Designing digital signal processors with RocketChip," in *Second workshop on computer architecture research with RISC-V (CARRV 2018)*, 2018.

[83] P. Rigge, V. N. Swamy, C. Nelson, F. Tufvesson, A. Sahai, and B. Nikolić, "Wireless Channel Dynamics for Relay Selection under Ultra-Reliable Low-Latency Communication," to appear in IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'2020), 2020.

[84] A. Sendonaris, E. Erkip, and B. Aazhang, "User cooperation diversity. part i. system description," *IEEE transactions on communications*, vol. 51, no. 11, pp. 1927–1938, 2003.

[85] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sis: A system for sequential circuit synthesis," 1992.

[86] Sercos, July 2020. [Online]. Available: https://www.sercos.org

[87] O. Shacham, "Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms," Ph.D. dissertation, Stanford University, 2011.

[88] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, B. Lee, *et al.*, "Rethinking digital design: Why design must change," *IEEE micro*, vol. 30, no. 6, pp. 9–24, 2010.

[89] M. Shirvanimoghaddam, M. S. Mohammadi, R. Abbas, A. Minja, C. Yue, B. Matuz, G. Han, Z. Lin, W. Liu, Y. Li, *et al.*, "Short block-length codes for ultra-reliable low latency communications," *IEEE Communications Magazine*, vol. 57, no. 2, pp. 130–137, 2018.

[90] ——, "Short block-length codes for ultra-reliable low latency communications," *IEEE Communications Magazine*, vol. 57, no. 2, pp. 130–137, 2018.

[91] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. I. Jordan, and S. S. Sastry, "Kalman filtering with intermittent observations," *IEEE Transactions on Automatic Control*, 2004.

[92] I. Std, L. A. N. Man, S. Committee, I. C. Society, and I.-s. S. Board, *Supplement to Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications — Physical Layer Parameters and Specifications for 1000 Mb/s Operation Over 4-Pair of Category 5 Balanced Copper Cabling ,*, 1999, vol. 1999.

[93] V. Stojanović, "Fast Fourier Transform: VLSI Architectures," *Lecture series, Massachusetts Institute of Technology*, 2006.

[94] C. E. Stroud, R. R. Munoz, and D. A. Pierce, "Behavioral model synthesis with cones," *IEEE Design & Test of Computers*, vol. 5, no. 3, pp. 22–30, 1988.

[95] V. N. Swamy, "Real-time Communication Systems For Automation Over Wireless: Enabling Future Interactive Tech," Ph.D. dissertation, University of California Berkeley, 2018.

[96] V. N. Swamy, "Real-time Communication Systems For Automation Over Wireless: Enabling Future Interactive Tech," Ph.D. dissertation, UC Berkeley, 2018.

[97] V. N. Swamy, G. Ranade, and A. Sahai, "Robustness of cooperative communication schemes to channel models," in *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2016, pp. 2194–2198.

[98] V. N. Swamy, P. Rigge, G. Ranade, B. Nikolić, and A. Sahai, "Predicting Wireless Channels for Ultra-Reliable Low-Latency Communications," in *IEEE International Symposium on Information Theory - Proceedings*, 2018.

[99] ——, "Wireless Channel Dynamics and Robustness for Ultra-Reliable Low-Latency Communications," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 4, pp. 705–720, 2018. [Online]. Available: http://arxiv.org/abs/1806.08777

[100] V. N. Swamy, P. Rigge, G. Ranade, B. Nikolić, and A. Sahai, "Wireless channel dynamics and robustness for ultra-reliable low-latency communications," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 4, pp. 705–720, 2019.

[101] V. N. Swamy, P. Rigge, G. Ranade, A. Sahai, and B. Nikolić, "Network coding for high-reliability low-latency wireless control," in *2016 IEEE Wireless Communications and Networking Conference Workshops, WCNCW 2016*, 2016.

[102] V. N. Swamy, S. Suri, P. Rigge, M. Weiner, G. Ranade, A. Sahai, and B. Nikolić, "Cooperative communication for high-reliability low-latency wireless control," *IEEE International Conference on Communications*, vol. 2015-Septe, pp. 4380–4386, 2015.

[103] ——, "Real-time cooperative communication for automation over wireless," *IEEE Transactions on Wireless Communications*, 2017.

[104] ——, "Real-time cooperative communication for automation over wireless," *IEEE Transactions on Wireless Communications*, 2017.

[105] V. N. Swamy, S. Suri, P. Rigge, M. Weiner, G. Ranade, A. Sahai, and B. Nikolić, "Real-time cooperative communication for automation over wireless," *IEEE Transactions on Wireless Communications*, vol. 16, no. 11, pp. 7168–7183, 2017.

[106] SWIG, August 2020. [Online]. Available: http://swig.org/

[107] M. Sybis, K. Wesolowski, K. Jayasinghe, V. Venkatasubramanian, and V. Vukadinovic, "Channel coding for ultra-reliable low-latency communication in 5g systems," in *2016 IEEE 84th vehicular technology conference (VTC-Fall)*. IEEE, 2016, pp. 1–5.

[108] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A functional validation technique: biased-random simulation guided by observability-based coverage," in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*. IEEE, 2001, pp. 82–88.

[109] Time-Sensitive Networking Task Group, July 2020. [Online]. Available: http://www.ieee802.org/1/pages/tsn.html

[110] L. Truang *et al.*, "Magma." [Online]. Available: https://github.com/phanrahan/magma

[111] D. Tse and P. Viswanath, *Fundamentals of wireless communication*. Cambridge university press, 2005.

[112] Typelevel, "Spire: Powerful new number types and numeric abstractions for Scala." [Online]. Available: https://github.com/typelevel/spire

[113] "Universal Verification Methodology (UVM) 1.2 User's Guide," Accelera, Tech. Rep., 2015.

[114] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*. IEEE, 1999, pp. 175–180.

[115] Accelera, Tech. Rep., 2018.

[116] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in Python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[117] J. Volder, "The CORDIC computing technique," in *Papers presented at the the March 3-5, 1959, western joint computer conference*, 1959, pp. 257–261.

[118] A. Volkova, T. Hilaire, and C. Lauter, "Reliable evaluation of the worst-case peak gain matrix in multiple precision," in *2015 IEEE 22nd Symposium on Computer Arithmetic*. IEEE, 2015, pp. 96–103.

[119] A. Wang, W. Bae, J. Han, S. Bailey, O. Ocal, P. Rigge, Z. Wang, K. Ramchandran, E. Alon, and B. Nikolić, "A real-time, 1.89-GHz bandwidth, 175-kHz resolution sparse spectral analysis RISC-V SoC in 16-nm FinFET," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 7, pp. 1993–2008, 2019.

[120] A. Wang, P. Rigge, A. Izraelevitz, C. Markley, J. Bachrach, and B. Nikolić, "ACED: A hardware library for generating DSP systems," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[121] E. Wang, C. Schmidt, A. Izraelevitz, J. Wright, B. Nikolić, E. Alon, and J. Bachrach, "A methodology for reusable physical design," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2020, pp. 243–249.

[122] J. Wang, J. Liu, and N. Kato, "Networking and communications in autonomous driving: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1243–1274, 2018.

[123] M. Weiner, M. Jorgovanovic, A. Sahai, and B. Nikolie, "Design of a low-latency, high-reliability wireless communication system for control applications," in *2014 IEEE International Conference on Communications, ICC 2014*, 2014.

[124] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *2013 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2013, pp. 362–365.

[125] G. Wu, Z. Li, H. Wang, and W. Zou, "Selective random cyclic delay diversity for HARQ in cooperative relay," in *2010 IEEE Wireless Communication and Networking Conference*. IEEE, 2010, pp. 1–6.

[126] Xilinx, "Vivado high-level synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[127] P. Zand, S. Chatterjea, K. Das, and P. Havinga, "Wireless industrial monitoring and control networks: The journey so far and the road ahead," *Journal of sensor and actuator networks*, vol. 1, no. 2, pp. 123–152, 2012.

[128] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fd-soi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[129] L. Zheng and D. N. C. Tse, "Diversity and multiplexing: A fundamental tradeoff in multiple-antenna channels," *IEEE Transactions on information theory*, vol. 49, no. 5, pp. 1073–1096, 2003.