Designing Digital Signal Processors with RocketChip

Paul Rigge
Dept. of Electrical Engineering and Computer Science
University of California, Berkeley
rigge@eecs.berkeley.edu

Borivoje Nikolić
Dept. of Electrical Engineering and Computer Science
University of California, Berkeley
bora@eecs.berkeley.edu

ABSTRACT

Systems-on-a-chip (SoCs) integrate general-purpose processor cores with specialized fixed function blocks to maximize efficiency while maintaining programmability. Integrated radios, audio and image processors are some common examples that perform signal processing through fixed-function blocks. Defining and implementing the hardware-software interface between such units is error prone and labor intensive, often making design space exploration difficult.

We present a set of extensions to the RISC-V RocketChip generator that aid in designing signal processors. An implementation of the AXI4-Stream protocol using RocketChip's Diplomacy framework performs parameter negotiation across levels of hierarchy. A standard software template, called `DspBlock`, is defined with streaming IO and optional memory mapped IO. The type of memory interface is left as a parameter that need not be specified by the designer of the DSP. Several useful building blocks that conform to this template are implemented, such as splitters, vector registers, queues, and fuzzers. An example design implementing synchronization for an OFDM baseband is presented.

CCS CONCEPTS

• **Hardware → Digital signal processing**; **Hardware description languages and compilation**; **Software tools for EDA**;

KEYWORDS

RISC-V, DSP, Chisel, Hardware Generators, Scala

ACM Reference Format:
Paul Rigge and Borivoje Nikolić. 2018. Designing Digital Signal Processors with RocketChip. In *Proceedings of Proceedings of Second Workshop on Computer Architecture Research with RISC-V (CARRV'18).* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

SoCs typically integrate several general-purpose compute cores as well as specialized hardware accelerators onto one chip. The fixed-function hardware blocks perform specialized signal-processing tasks with much higher energy and area efficiency than general-purpose processors . General-purpose processors are interfaced with specialized blocks with drivers, libraries, and APIs that allow application writers to partition computation across specialized hardware blocks and the CPUs.

CARRV'18, June 2018, Los Angeles, CA USA

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Designing and verifying SoCs is challenging, time consuming, and expensive. A major challenge is often encountered in translating application requirements into hardware specifications. For example, a requirement that a mobile chipset be able to hold a video call with < 500ms latency over the cellular network for over an hour on battery is easy to express superficially, but challenging to translate into hardware specification. Applications require not only individual specialized hardware blocks to meet performance and efficiency requirements, but also require the integrated system with its complex interdependencies and its software stack to meet them. Modeling systems at sufficient detail to make good decisions is difficult. [16] presents an automated way of identifying hot loops in an application and exploring the design space of hardware accelerators by leveraging HLS, but these approaches are only as good as the limitations of HLS allow them to be.

Signal-processing SoCs are typically designed in a top-down manner [14], with an early specification that is often frozen before architecture definition, and is therefore difficult to change late in the design process. Of course, developing a large SoC is complex and difficult, so in practice the specifications will have to be changed despite the difficulty it poses. The difficulty in changing specs late in a design and still meeting application requirements often makes initial designs too risk averse, and also results in many projects missing deadlines and overrunning costs.

Agile development practices were developed to manage similar uncertainty and delays in large software projects [5]. Agile design practices have also been applied to several hardware projects involving the RocketChip generator, an open source processor generator that implements the RISC-V ISA. RocketChip is implemented in Scala using the Chisel hardware construction language [3] and makes extensive use of advanced hardware generator features.

RocketChip, along with related projects like Hwacha [9] and BOOM [6], is a valuable tool for exploring the processor design space with its software stack. RISC-V's strong software ecosystem enables easy development and porting of real applications to evaluate points in the design space. RocketChip has been used to generate SoCs with digital signal processors [4, 15].

This paper discusses a method for developing SoCs with specialized digital signal processors using RocketChip. Dedicated signal-processing accelerator blocks are added as peripheral devices to core. We present resources for writing DSP generators with Chisel, including a Diplomacy-based implementation of the AXI-4 Stream interface. We also discuss interfacing DSP components with RocketChip using the `DspBlock` construct as well building blocks like `DspRegisters` and `DspQueues`. We present a discussion of codeveloping software and hardware and present an example of an OFDM synchronization block (suitable for a 802.11 radio baseband) interfaced with a processor.

2 DSP GENERATORS IN CHISEL

Chisel has many facilities for developing hardware generators for digital signal processing. One of the base types in Chisel is a fixed point type, which is essential for efficient DSP implementation. Generator authors can specify literals for fixed point types with floating point numbers and the appropriate integer literal will be computed, even when the width and binary point are inferred. The appropriate shifts are automatically generated for adds and subtracts, and several rounding modes are supported.

The dsptools library extends Chisel's basic functionality with more features for developing DSP hardware [10]. It adds a simulation-only floating point type to Chisel as well as a complex number type. It adds typeclasses (based on those from the Scala library Spire [11]) for developing type-generic hardware generators. By using these typeclasses a single circuit generator can generate instances that use fixed or floating point numbers. When testing generated circuits that may have different precisions or number types, it is important that the tester is also as flexible as the design under test (DUT), so dsptools provides extensions to Chisel's standard testers to support type-generic generators and different precisions (e.g. checking that a number is within a parameterized relative error of the exact expected value). Dsptools also supports automatic insertion of pipeline registers after typical operations, such as multiply and add. Global defaults can be overridden hierarchically.

3 DIPLOMATIC AXI4-STREAM INTERFACE

The AXI4-Stream protocol is a standard interface defined by AMBA for generic streaming interconnects [2]. Among other things, it is suitable for a wide range of streaming DSP applications. It defines an interface with the following signals, also illustrated in Fig. 1:

- Clock and reset
- TVALID: Indicates valid transfer
- TREADY: Indicates slave is ready to receive
- TDATA: Data
- TSTRB: Indicates if bytes in TDATA are data or position
- TKEEP: Indicates if bytes in TDATA are null or not
- TLAST: Marks end of packet
- TID: Identifies the source of stream
- TDEST: Identifies the destination of stream
- TUSER: No specific purpose, implementors can assign their own meaning

Most of the signals are optional; only the clock, reset, and TVALID are mandatory - even TDATA is optional. There are some required relationships between the widths of some fields; for example, TDATA must be in multiples of a byte, say $8n$ bits wide, and TSTRB and TKEEP must be n bits wide if they exist. As you would expect, the standard also defines the timing and semantics of transfers. However, because so many signals are optional and because the standard allows for some implementation freedom, it also discusses default values for ports, how to connect interfaces that support different subsets of the ports, and how to connect ports that have different size interfaces.

Diplomacy is a framework for negotiating parameters for hardware interfaces. [7] presents an example with TileLink, the main memory interconnect standard used by RocketChip, although implementations for other memory interfaces such as AXI, APB, and AHB also exist.

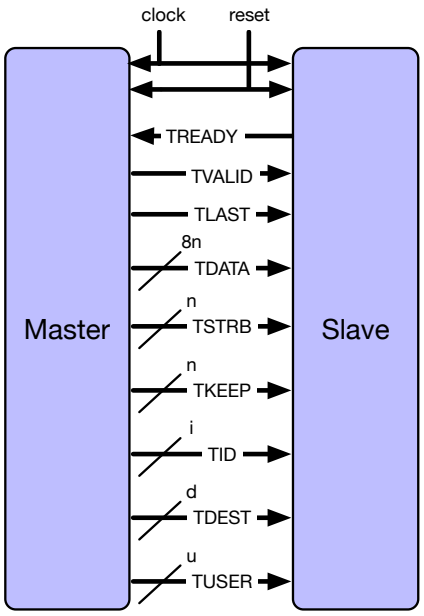

Figure 1: Illustration of the AXI4-Stream interface. There are 4 parameters that dictate the widths of the ports: n, i, d, and u. n determines the widths of TDATA, TSTRB, and TKEEP. i determines the width of TID and d determines the width of TDEST. u determines the width of TUSER.

An implementation of AXI4-Stream using Diplomacy was developed [13]. Fig. 2 shows how parameters flow in this implementation. The slave parameters specify which subset of the optional ports are supported[1], how many distinct addressable endpoints are downstream of the node, and if the slave is always ready for new transactions. The master parameters specify the width of the TDATA/TSTRB/TKEEP (whichever are present) and TUSER. TUSER can be entirely omitted by setting its width to zero. Master parameters also specify the number of distinct addressable masters that are upstream of the node. Edge parameters join the master and slave parameters and computes the actual widths of the AXI4-Stream ports. TID and TDEST widths are computed by finding the number of bits required to represent all the upstream masters and downstream slaves, respectively.

As in the case of TileLink in [7], diplomacy can be used to make "thin" adapters for AXI4-Stream. For example, adapters to add new fields and assign the default values from the standard are trivial. Adapters that resize fields according to the guidelines in the standard are also straightforward. These thin adapters can be composed to build more complex adapters.

In addition to parameters that determine the absence or presence of ports or their widths, the `alwaysReady` parameter is used to

[1] TLAST and TRESET are optional in [2] but not optional in this implementation. However, the `alwaysReady` option can be set to true to indicate that a slave is always ready for new transactions.

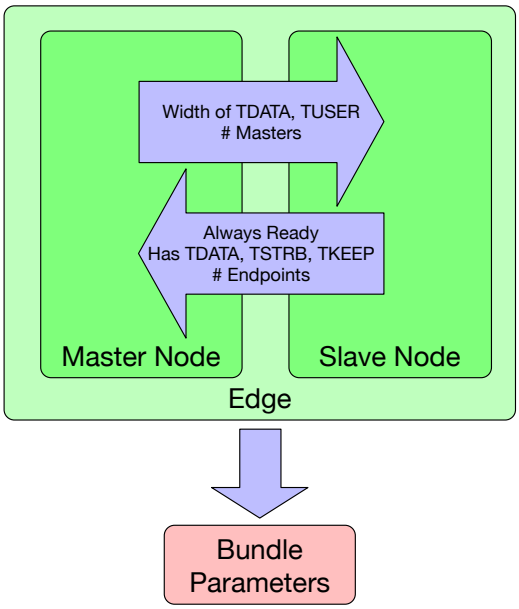

Figure 2: Diagram showing how parameters flow for the AXI4-Stream implemented with diplomacy.

conditionally generate hardware. Section 5.3 discusses an example where a stream splitter is able to avoid generating a queue and simply use wires because it knows that slaves will be always ready.

4 MEMORY-MAPPED PERIPHERALS WITH ROCKETCHIP

RocketChip provides a wide array of interconnect generators, ranging from converters (e.g. TileLink ↔ AXI4), crossbars, buses, width adapters, fragmenters, and interrupt buses. Before using any of these powerful generators to connect a streaming DSP design to a core (typically via the periphery bus), the DSP blocks needs to have a memory interface.

4.1 `DspBlock`

The basic building block for streaming DSP hardware that can be interfaced to the core is called `DspBlock`, defined in [13]. It is a Scala **`trait`** that defines two members:

- `streamNode`, the AXI4-Stream diplomacy node
- `mem`, an optionally present generic memory interface diplomacy node

`DspBlock` does not mandate what type of interface `mem` is defined as; rather, it is generic. This allows authors of blocks to design the DSP without worrying about the specifics of the memory interface being used. The DSP expert can defer making decisions about interconnect to an expert at integrating peripherals and interfacing them with processors. The memory interface used can easily be a parameter to sweep in a design space exploration.

The concrete (i.e., the type of the memory interface has been specified) instance of the generator must specify the logic that manages memory interface transactions. When the details the memory interface's logic aren't important (e.g. for status and control registers that do not require burst mode accesses), RocketChip's `RegisterRouters` are very useful. Register routers provide a declarative API for defining status and control registers as well as interrupts and will automatically add appropriate entries to the device tree generated by RocketChip. They are implemented for TileLink, AXI4, APB, and AHB. A diagram showing the inheritance relationships for this design pattern is shown in Fig. 3.

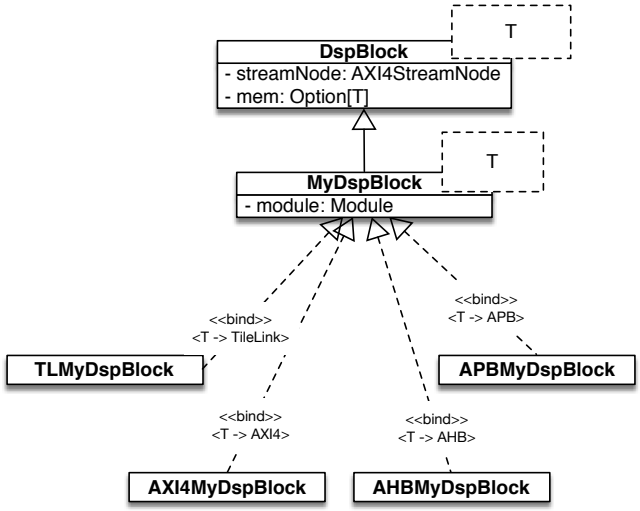

Figure 3: Diagram showing inheritance relationships for `DspBlocks` with generic memory interfaces. `DspBlock` has a generic type `T` that stands for the memory interface. `MyDspBlock` defines the DSP functionality in `Module`, but does not define the memory interface. Classes like `TLMyDspBlock` bind `T` to a concrete type such as TileLink or AXI-4.

4.2 `DspChain`

A `DspChain` is a collection of `DspBlocks` that is itself a `DspBlock`. They generate a crossbar that connects to each block and connects the blocks in sequence, and support hierarchical design. A pre-diplomacy version was used in [4] to generate design-for-test (DFT) structures that were muxed into each block in the chain, which was very valuable for both verification and bring-up. These DFT structures allowed unit tests to be translated into C programs that ran on the processor. The same binaries were used in simulation before taping out and on hardware during bringup. Section 6 describes this methodology in greater detail.

`DspBlocks` and `DspChains` can be integrated into a RocketChip SoC via the periphery bus. The periphery bus uses the uncached variant of TileLink and has a convenient API for adding any number of peripherals. Fig. 4 illustrates how the memory interconnect is organized.

5 USEFUL BUILDING BLOCKS

Synchronous data flow (SDF) is a well-studied abstraction for designing DSP systems [8].A collection of actors exchanges tokens via queues at known rates. There are other models that place more or less strict constraints on the state of the processors and the

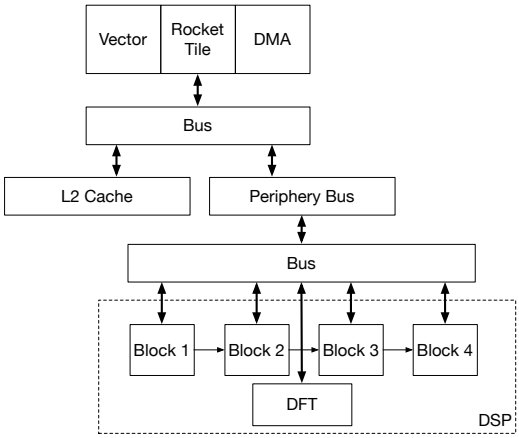

Figure 4: Integrating DSP blocks with a Rocket core via the periphery bus.

rates of token consumption and production, but models generally partitions the computation into a graph of processors communicating via channels. The goal is to be able to make some sort of useful guarantee, such as when an SDF is deadlock-free or what the minimum/average throughput is.

To support this style of design, some useful building blocks have been developed within this framework.

5.1 DspRegister

A `DspRegister` is a vector register loosely based on the vector registers in Hwacha [9]. They have a programmable (via the memory interface) vector length called `veclen` and can store (read from the the input AXI4-Stream interface into the memory) and load (read out of the memory to the output AXI4-Stream interface). A load and store can run concurrently (i.e. passthrough) as long as the store stays ahead of the load. When a load or a store has transferred `veclen` samples, TLAST is asserted and the load or store is complete. They can function as queues in the context of SDF.

They are implemented as `DspBlocks` and thus have versions for all of RocketChip's supported memory interfaces. The contents of the vector register can optionally be memory mapped. This is used when a portion of the SDF is implemented on a Rocket core or on a vector co-processor, or if the `DspRegister` is located at the output of the computation.

5.2 DspQueue

`DspQueues` buffer input samples and generate an interrupt if they get to full. They are always ready on the AXI4-Stream input with the assumption being that if it gets too full the interrupt will be asserted and the processor will step in to guarantee that the queue drains. They generate two interrupts: one triggers when the queue is totally full and the other triggers when the queue is filled beyond some programmable threshold.

5.3 Splitters

Splitters take one input AXI4-Stream and send it to multiple endpoints. In the simple case that all the downstream endpoints are always ready, the logic is trivial: the input is mapped directly to all of the outputs. If the downstream endpoints are not always ready, the designer needs to decide whether to wait for every endpoint to be ready or to place queues in front of each endpoint and allow the faster outputs to move ahead.

6 VERIFICATION

When designing hardware generators, it is important that the verification software be at least as parameterizable as the generator being tested. If not, there is a risk that a subset of the design space to be explored is unverified.

Fixed precision representation of DSP generators often presents a verification challenge. Dsptools allows for testers that take precision into account via assertions that check that outputs fall within an expected range. It also provides hooks to convert outputs and inputs of the circuit to and from floating point numbers. This enables use cases such as a single test with floating point test vectors and assertions that can test both floating point and fixed point instances of the generator under test.

[4] uses DFT structures for verification. These DFT structures not only allow for debugging in-situ, they assist in pipecleaning hardware/software interfaces in simulation early on in the design process. Performing unit-tests using DFT structures also tests the interconnect, the core, and the software stack.

Still, testing hardware units with DFT structures and software running on the CPU has a relatively slow simulation time[2]. In [4], loading the program and printing the results dominated runtime, which is obviously undesirable. Performing the tests by simulating the block in isolation is one obvious solution, but ideally these tests would be as similar as possible to the DFT unit tests.

To facilitate this, [13] implements several bus protocols for Chisel's built in `PeekPokeTesters`. Models exist for TileLink, AXI4, APB, AHB, and AXI4-Stream. These models can concurrently write to or read from memory ports while streaming data in and out of a `DspBlock`. This allows for writing the same types of tests that would run on the CPU, but with substantially faster simulation time. An illustrative example is shown in Fig. 5.

Another important inclusion in [13] are fuzzers for AXI4-Stream. A fuzzer is a `DspBlock` with an AXI4-Stream master node that outputs potentially random data. They are parameterized to optionally include any legal subset of the ports. By default, every port is randomly assigned to, but they can, for example, be used to generate fixed length packets, increasing sequences of data, etc.

7 SOFTWARE INTERACTION

After a DSP task has been mapped into a collection of processing elements (actors), `DspRegisters`, `DspQueues`, splitters, etc., the computation is driven via interactions over the memory interfaces. Typically, a processor is responsible for scheduling the computation and performing the necessary reads and writes to parameterize the actors and move data. Some portion of the computation may be performed on the CPU or on a specialized coprocessor like a vector accelerator.

[2] Or long FPGA compile times

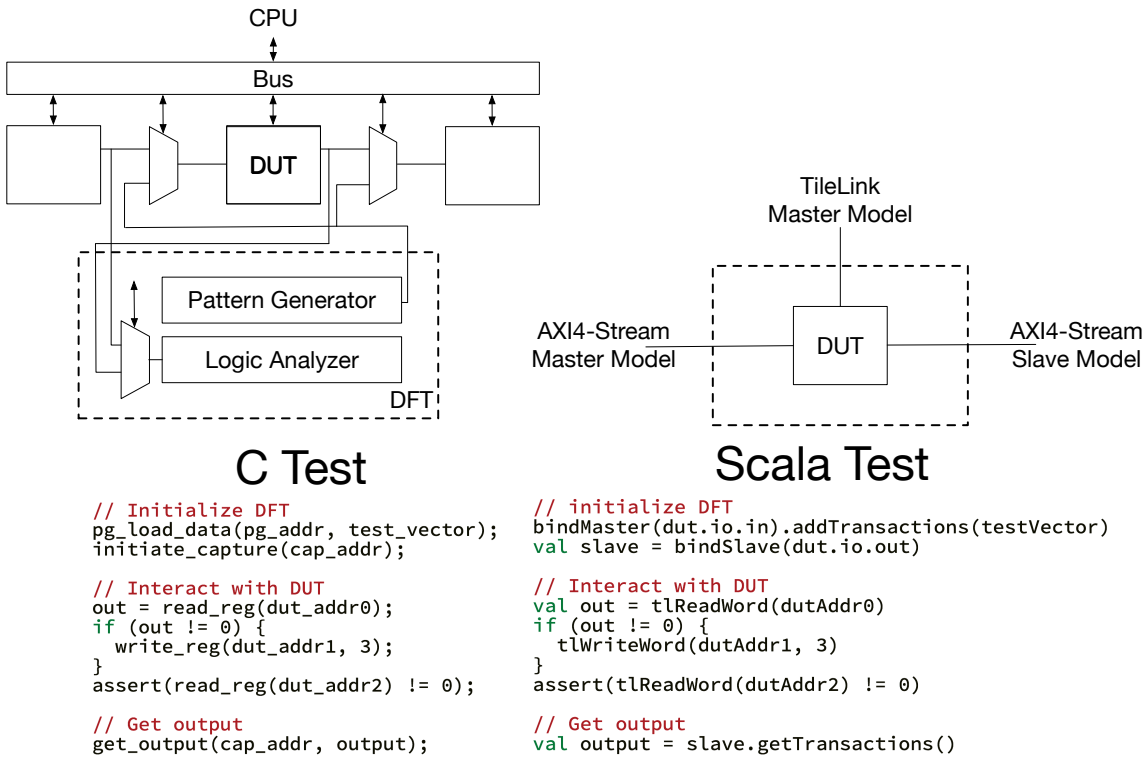

Figure 5: Example showing two styles of unit testing. On the left is a unit test being run as software running on the RISC-V CPU that uses DFT structures to test the device. API calls load input data into the pattern generator and set the mux select to the DUT to accept input from the pattern generator. The logic analyzer is also primed to record output from the DUT. Then the CPU reads from and writes to registers in the DUT. On the right is a unit test being run in simulation with the bus models. The TileLink master model provides `tlReadWord()` and `tlWriteWord()` for interacting with registers in the DUT.

The task of scheduling computation for these kinds of problems has been a subject of extensive studies [8]. It may be computationally expensive and require the CPU to be responsive to a large number of interrupts, so we have explored dedicating a small 32-bit RISC-V core running a real-time operating system.

One common way to simultaneously develop hardware and software is through the use of an executable model [14]. As hardware blocks are completed, pieces of the executable model should be updated to use the new hardware. To support agile hardware/software codesign, as much of this software as possible should be generated along with the hardware. For example, addresses of configuration registers and driver functionality for running transactions can often be generated. This was done in [4] for the purpose of agile verification.

8 IP-XACT

IP-Xact is an XML-based standard that documents modeling information about IP such as port names, bus types and configurations, and pointers to different views of the IP [1]. There are a variety of use cases for IP-Xact ranging from IP integration to documentation generation, but here we will focus on its use for verification. The project in [4] used IP-Xact along with Cadence's Verification Workbench to automatically generate test harnesses and scoreboards based on IP-Xact generated by the Chisel generators. This allowed verification engineers to generate test vectors in a productivity language (Python) and automate the tedious, repetitive, and error-prone task of wiring up test harnesses and running the memory and streaming transactions.

The concept of diplomacy nodes maps nicely to many constructs in IP-Xact. The process of porting some of the functionality from [4] (written before the addition of diplomacy to RocketChip)to work with diplomacy is on-going. The goal is to both produce IP-Xact from diplomatic graphs and consume IP-Xact to create diplomatic nodes. This would be used for automating unit testing as well as integrating 3rd-party IP with RocketChip.

9 OFDM EXAMPLE

An example using the constructs presented here to build an OFDM baseband synchronization block has been developed [12]. OFDM is the modulation scheme used by modern wireless standrads, including the WiFi and LTE. This architecture is intended to be used to implement a hardware-accelerated, highly-efficient software-defined radio (SDR) capable of running OFDM physical layers. The block diagram for a transceiver using elements like `DspRegister` and `DspBlock` is shown in Fig. 6.

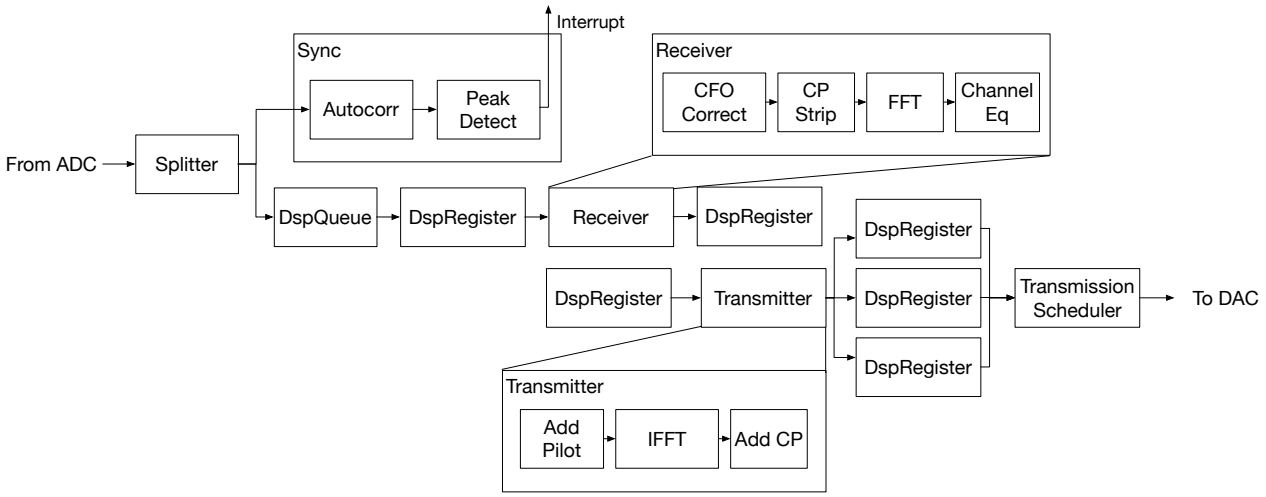

Figure 6: Block diagram for an OFDM transceiver using `DspBlock`, `DspRegister`, `DspQueue`, etc. On the receiver side, a splitter takes the input from the ADC and sends it to a synchronization block that generates interrupts upon packet detection and into a receiver chain. On the transmitter side, a register feeds into a sequence of transmitter blocks that then feeds into a number of parallel registers. A transmission scheduler plays samples out of those registers at the appropriate time. Having multiple registers allows multiple transmissions to be scheduled.

The output of the analog-to-digital converter (ADC) is provided as an AXI4-Stream input and immediately goes into a splitter. Both outputs of the splitter are always ready. The sync block performs a programmable width and depth autocorrelation followed by a peak detection. If a peak is detected indicating a packet has arrived, an interrupt is thrown and the result of the autocorrelation is provided. Several of these blocks could be independently programmed and run in parallel.

The other output of the splitter is the receiver. After the synchronization block detects a packet, the CPU throws away the unwanted samples at the beginning to align the input. It can do this by setting the first register's vector length to the number of samples to drop, performing a store, and then resetting the vector length to the desired vector length for decoding and performing another store. After this second store, a load can be performed immediately and the data can pass through to the decoder and ultimately to the register that collects the output.

10 CONCLUSION

This paper presents a collection of extensions to Chisel and the RocketChip generator for designing digital signal processors and integrating them into an SoC. A diplomacy-based implementation of AXI4-Stream, type-generic memory mapped building blocks, verification models for streaming and memory interfaces, and design and verification methodologies aid designers in building complex designs and exploring design spaces. An example application with an OFDM synchronization block is shown, demonstrating some of the utility of this approach. Future work includes developing an SDR capable of running customized versions of 802.11 or 4G/5G.

ACKNOWLEDGMENTS

The authors acknowledge the NSF GRFP for support, as well as Adept and the Berkeley Wireless Research Center, especially Stevo Bailey, Angie Wang, Adam Izraelevitz, Chick Markley, Timo Joas, and Jim Lawson.

REFERENCES

[1] Accellera Ip-xact Working Group. 2018. IP-XACT User Guide. (2018).
[2] ARM. 2010. AMBA 4 AXI4-Stream Protocol. (2010).
[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel. *Proceedings of the 49th Annual Design Automation Conference on - DAC '12* (2012), 1216. https://doi.org/10.1145/2228360.2228584
[4] Stevo Bailey et al. 2016. CRAFT2 Chip. (2016). https://github.com/ucb-art/craft2-chip
[5] Kent Beck et al. 2001. Manifesto for Agile Software Development. (2001), 2–3. https://www.researchgate.net/file.PostFileLoader.html?id=57d055b593553b11467ddd59{&}assetKey=AS{%}3A403742915612673{%}401473271220194
[6] Christopher Celio, Pi-feng Chiu, Borivoje Nikolić, David Patterson, and Krste Asanović. 2017. BOOM v2. In *CARRV*.
[7] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic Design Patterns: A TileLink Case Study. In *CARRV*, Vol. 7. https://doi.org/
[8] E. A. Lee and D. G. Messerschmitt. 1987. Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing. *IEEE Trans. Comput.* C, 1 (1987), 24–35.
[9] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. 2015. The Hwacha Vector-Fetch Architecture Manual, Version 3.8.1. (2015). https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-262.pdf
[10] Chick Markley, Paul Rigge, Stevo Bailey, and Angie Wang. 2016. dsptools: A Library of Chisel3 Tools for Digital Signal Processing. (2016). https://github.com/ucb-bar/dsptools
[11] T. Osheim, E. and Switzer. 2011. Spire: Powerful new number types and numeric abstractions for Scala. (2011). https://github.com/non/spire
[12] Paul Rigge. 2017. Chisel Things for OFDM. (2017). https://github.com/grebe/ofdm
[13] Paul Rigge and Stevo Bailey. 2017. Rocket DSP Utilities. (2017). https://github.com/ucb-art/rocket-dsp-utils/tree/diplomacyPort
[14] Jürgen Teich. 2012. Hardware/software codesign: The past, the present, and predicting the future. *Proc. IEEE* 100, SPL CONTENT (2012), 1411–1430. https://doi.org/10.1109/JPROC.2011.2182009
[15] Angie Wang et al. 2016. FFT2 Chip. (2016). https://github.com/ucb-art/fft2-chip
[16] Wei Zuo et al. 2017. Accurate High-level Modeling and Automated Hardware/-Software Co-design for Effective SoC Design Space Exploration. *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17* (2017), 1–6. https://doi.org/10.1145/3061639.3062195