Implementing Synthetic Aperture Radar Backprojection in Chisel – A Field Report

Niklas Rother[1][0000−0001−6374−3379], Christian Fahnemann[1][0000−0002−1092−582X], and Holger Blume[1][0000−0002−0640−6875]

Institute of Microelectronic Systems, Leibniz University Hannover, Hannover, Germany
{rother,fahnemann,blume}@ims.uni-hannover.de

Abstract. Chisel is an emerging hardware description language which is especially popular in the RISC-V community. In this report, we evaluate its application in the field of general digital hardware design. A dedicated hardware implementation of a Synthetic Aperture Radar (SAR) processing algorithm is used as an example case for a real-world application. It is targeting a modern high performance FPGA platform. We analyze the difference in code size compared to a VHDL implementation. In contrast to related publications, we classify the code lines into several categories, providing a more detailed view. Overall, the number of lines was reduced by 74 % while the amount of boilerplate code was reduced by 83 %. Additionally, we report on our experience using Chisel in this practical application. We found the generative concept and the flexibility introduced by modern software paradigms superior to traditional hardware description languages. This increased productivity, especially during timing closure. However, additional programming skills not associated with classic hardware design are required to fully leverage its advantages. We recommend Chisel as a language for all hardware design tasks and expect its popularity to increase in the future.

Keywords: Hardware description languages · Chisel · SAR · VHDL

1 Introduction

Following Moore's law, the complexity of digital designs has continuously increased in the last years. Today, most designs are not created from scratch, but rather are modifications or combinations of existing blocks (IP cores). In this context, the hardware description languages that are traditionally used in this field—Verilog and VHDL—have begun to show their limitations. One attempt to create a more modern hardware design language is *Chisel* [3], developed at the University of California, Berkeley.

While Chisel has found adoption in the RISC-V community, its use as a general language for digital design is limited up to now. In this paper we report on the use of Chisel for an FPGA implementation of the Backprojection algorithm used in Synthetic Aperture Radar (SAR). Our existing implementation [4] was

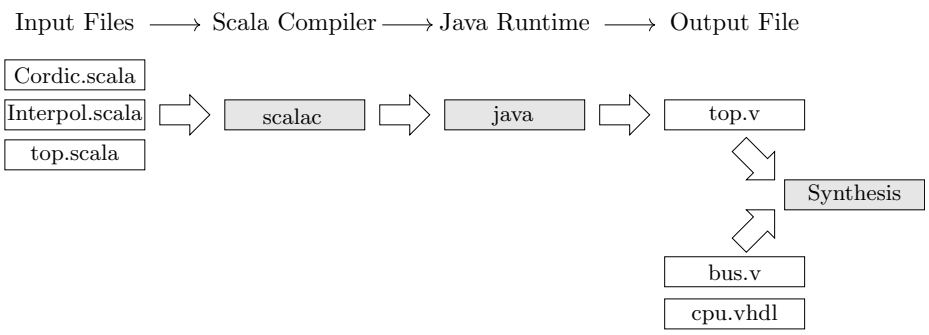

Fig. 1. Chisel synthesis process. The input files are compiled using the Scala compiler and the result is executed on the Java Runtime. This results in a single Verilog file, containing the generated code. This is used together with (optional) additional files in the conventional synthesis workflow.

implemented in VHDL. To explore the possibilities of a more modern language, the design was re-implemented from scratch in Chisel. This paper provides an extensive comparison of both implementations in terms of code size. Additionally, we describe our experience working with Chisel and list strengths and weaknesses we discovered.

Chisel is described by its authors as a *hardware construction language*, compared to Verilog and VHDL, which are *hardware description languages* [3]. The latter were originally developed to describe and simulate existing hardware designs. Only later, they have been repurposed for hardware synthesis. Current digital designs often are a collection of pre-existing modules (IP cores) that are parametrized and interconnected using a bus interface (e.g. AXI, Avalon, etc.) to form larger systems (System-on-Chip, SoC). Highly configurable structures were not envisioned during the design of Verilog and VHDL and are therefore notoriously hard to describe with them.

Chisel is built around the idea of *hardware generators*, i.e. executable design descriptions that can flexibly adapt themselves based on parameters. From a technical point of view, Chisel is an extension of the Scala programming language, which itself is build atop the Java Virtual Machine. The output of the Chisel build flow is a Verilog file which can be processed by conventional tools. A visual overview of this flow is shown in Fig. 1. The input files are compiled using the Scala compiler, and the resulting program is executed using the Java Runtime. This program then generates a single Verilog file as the output, containing a description of the generated hardware. This file can then be processed by conventional synthesis tools, optionally together with other source files written in VHDL, Verilog, etc. From a user perspective, the complete process from the input to the output files is usually orchestrated using the *sbt* tool and is executed using `sbt run`.

One important aspect of Chisel is, that the hardware generators are runnable programs that are *executed* during the elaboration phase to assemble the final

circuit. Since the generators are implemented in Scala, arbitrary code can be executed in this stage to e.g. generate lookup tables or adapt the generated design to the chosen parameterization. Chisel inherits most aspects of Scala, like object-oriented design, type inference and functional programming. This allows complex hardware generators to be expressed in a few lines of code. Besides this, Chisel can also be used as a mere replacement for Verilog or VHDL. In this case, some features of Chisel, like the bulk-connect operator (<>), can help to reduce the verbosity of the code.

One shall note that Chisel is not a *high level synthesis* (HLS) language. The hardware design must still be carried out at the register level, Chisel can not e.g. extract a datapath design from a algorithmic description. Area and timing results are therefore expected to be a close match to a Verilog or VHDL description of the same circuit (cf. [9]).

Part of the Chisel ecosystem is *ChiselTest* [11], a testing library integrated with *ScalaTest*. It allows to write unit-tests for hardware described in Chisel. The tests can be either executed in a simulator implemented in Scala, or delegated to *Verilator*, an open-source Verilog simulator. As the tests are implemented in Scala, arbitrary Scala code can be executed during the test to generate test stimuli or check the expected results against a software implementation, in comparison to classic workflows where stimuli and expected results are often generated outside the testbench.

Comparable studies on the relation of Verilog and Chisel have previously been reported [2, 3, 7, 9]. The original paper on Chisel [3] reports a 65 % decrease in lines of code (LoC) for a 3-stage 32-bit RISC processor converted from Verilog to Chisel. Im et al. [7] also found a 53 % decrease in LoC for a 64-bit RISC-V processor. Both performed physical layout for an ASIC process and found comparable area requirements for the Chisel and Verilog version.

Lennon et al. [9] performed an extensive evaluation of a Chisel implementation compared to a workflow where Verilog code is generated using a Tcl script. Typical building blocks of FPGA-designs, like arbiters and FIFOs were used for evaluation. They found the synthesis time, FPGA resource requirements, and maximum reachable frequency to be comparable, but noted that Chisel did not support asynchronous resets[1], which reduced the maximum frequency for FPGAs not supporting synchronous resets. The LoC were reduced by 0 % to 37 %, depending on the design. Compared to the Tcl+Verilog workflow, the authors experienced an increase in coding speed by a factor of 3, which they attribute to the more helpful error messages of Chisel and the fact that Tcl is not aware of Verilog syntax. In their conclusion, Lennon et al. mention the steep learning curve of Chisel and the need to understand the software aspects: "Although Chisel designers require a foundation in software design to allow them to utilize Chisel's power, constructing circuits with Chisel requires an identical appreciation of the hardware being described to that held by Verilog designers." [9]

Arcas et al. [2] compared hardware implementations of algorithms commonly found in database applications. Verilog, Bluespec SystemVerilog, Altera

[1] Support for asynchronous resets has since then been added to Chisel in version 3.2.0.

OpenCL, LegUp, and Chisel were used for the comparison. Resource requirements and maximum frequency are reported to be on par between the Verilog and Chisel implementation. The LoC of the Chisel implementation are reduced by 11 % to 16 %, aside the "hash probe" design, where the LoC are increased by 25 %. They likewise conclude that "some Scala knowledge" [2] is needed to efficiently work with Chisel.

The cited studies either focus on processors designs, or use very small examples to evaluate Chisel. While the reported results look promising, it remains open whether Chisel is also beneficial for general data processing applications, besides processor design. To the best of the authors knowledge, this is the first report evaluating the use of Chisel for a sophisticated real-world application involving a dedicated hardware architecture.

The rest of this work is structured as follows: Section 2 describes the system architecture and the underlying algorithm used for this evaluation. A quantitative comparison of the Chisel and VHDL implementation are given in Section 3, followed by a report on our experience while using Chisel in Section 4. A conclusion is drawn in Section 5.

2 System Description

In the following sections we briefly describe the principle and application of Synthetic Aperture Radar (SAR), the image formation algorithm Backprojection (BP), and give an architectural overview of our implementation. Both, the VHDL and the Chisel implementation, are fully working on an FPGA platform and produce identical results.

2.1 Synthetic Aperture Radar (SAR)

Acquiring near-live images of remote locations has many applications ranging from surveillance to disaster control. Often, electro-optic sensors (cameras) are used for this. While these provide a good solution for aerial imaging in decent environmental conditions, they cannot provide usable images without daylight or when clouds or smoke block the cameras' view.

An alternative imaging technique is synthetic aperture radar (SAR). Radar pulses are transmitted continuously and the received echoes are captured and stored while flying alongside the area of interest [5]. By merging the captured echo signals using a SAR processor, an aerial image of the scene is generated which shows the reflectiveness per ground area. Since radar waves have a much lower frequency than visible light and actively illuminate the scene with their signal, these systems can operate independent of daylight and penetrate most weather phenomena and air pollution.

2.2 Backprojection (BP)

To generate a usable image from the acquired raw data, an image formation step is necessary. The main part of this is called azimuth compression (AC)

and essentially synthesizes a huge virtual antenna which, in turn, has a high spatial resolution. Historically, algorithms used in digital SAR processors have been optimized using frequency domain operations. While these enable a computationally efficient image formation, they are sensitive to deviations in the flight path, which need to be compensated separately. In face of current technology, especially in the field of FPGAs, time-domain-based image processing was shown to be feasible. This enables ideal correction of known flight errors or even deliberately chosen nonlinear flight trajectories [6].

One straightforward AC algorithm for SAR is Backprojection (BP). This time domain based technique can be summarized as follows: The output image is defined as a pixel grid of size $X \times Y$ encompassing the secene. For each of these pixels $g_{x,y}$, BP takes the echo captures d_m of all radar pulses into account. Based on the real-world locations of the the image pixel and the antenna while capturing the echo, their geometric distance Δr is calculated. This information indicates, where echo data from this image position is stored. It is then used to interpolate a sample $d_m[\Delta r]$ from the echo data. These complex-valued samples from all radar pulses are then added per pixel. Also, a phase-correction factor k is used to enable coherent processing. After this process, areas with strong reflectors will have a high absolute pixel value while the coherent sum for non-reflective pixels will dissolve into noise. Equation (1) summarizes the BP computation per pixel.

$$g_{x,y} = \sum d_m[\Delta r] \cdot e^{i2k\Delta r} \tag{1}$$

Since there is no interdependence between the pixels, the image dimensions $x \in [0, (X-1)]$ and $y \in [0, (Y-1)]$ yield two straight-forward parallelization domains.

2.3 Architecture overview

This section gives an overview of our architecture for the SAR Backprojection algorithm, so that the extent and complexity of the implementation can be estimated. A more detailed description of the VHDL implementation has been published earlier [4].

For our design we chose the aforementioned parallelization strategy in the pixel domain: The basic processing element is called a "submodule"; a block diagram of it is shown in Fig. 2. A submodule is capable of storing a number of neighboring pixels in an output buffer, called the line accumulator. The required part of the echo data from one pulse is fed into an input buffer. The input buffer and accumulating output buffer both consist of two Block-RAM-based memories each. These are swapped back and forth using multiplexers in order to allow filling and flushing the buffers while the projection of the next/previous line is ongoing.

The beginning of the processing chain is marked by the "Pixel Position Generator", that successively generates the real-world coordinate of the pixels on the current image region. The generated position is subtracted from the antenna

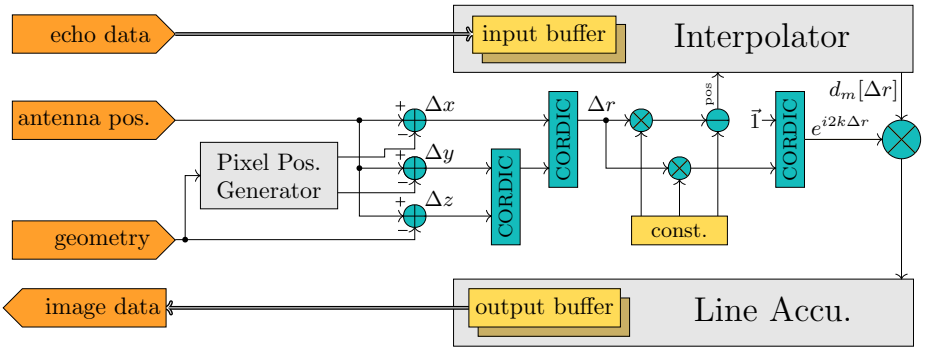

Fig. 2. GBP submodule core architecture. Two CORDICs are used to calculate the distance Δr for each pixel, which is sent to the interpolator. The result $d_m[\Delta r]$ is multiplied by a phase correction factor $e^{i2k\Delta r}$ and stored in the accumulator. Arrow types denote data types: $\rightarrow$ for normal busses and $\Rightarrow$ for AXI Stream connections.

position, and two CORDIC instances [12] are used to calculate the 3D vector length, which is the distance Δr from the antenna to the pixel. The result is given to the interpolator, which fetches the required samples from the input buffer. Additionally, a phase correction $e^{i2k\Delta r}$ is calculated using another CORDIC instance. The phase correction is applied before the sample is transferred to the accumulator and is stored in the output buffer. After all input data has been processed, the output buffer is flushed to main memory and a new set of pixels is computed.

An arbitrary number of these submodules can be used in parallel, each computing different subsets of the final image. As all submodules operate on the same echo data, this data can be broadcasted, keeping memory read accesses constant. The outer interface of these submodules consists of the echo data input stream, the image data output stream, control signals, and ports for the geometric parameters (cf. Fig. 2).

While the older VHDL implementation manages the coordination of the submodules using a central finite state machine (FSM), the newer Chisel implementation uses a miniature control processor called PacoBlaze [8]. This has the advantage of providing more flexibility and reprogrammability compared to a hardcoded FSM while still not taking up as much area as a full-featured microcontroller.

The interpolator implements a sinc interpolation to generate a sample at a fractional position, using a structure similar to a FIR filter. A block diagram of the interpolator is shown in Fig. 3. The incoming position is split into an integer and a fraction part. The integral part is used to select samples from the input buffer. The buffer memory is organized in a way that t samples are stored in a single memory word. It is implemented with dual port access so that two locations can be accessed at the same time. To perform the interpolation, two adjacent words—containing $2t$ samples in total—are fetched. From them,

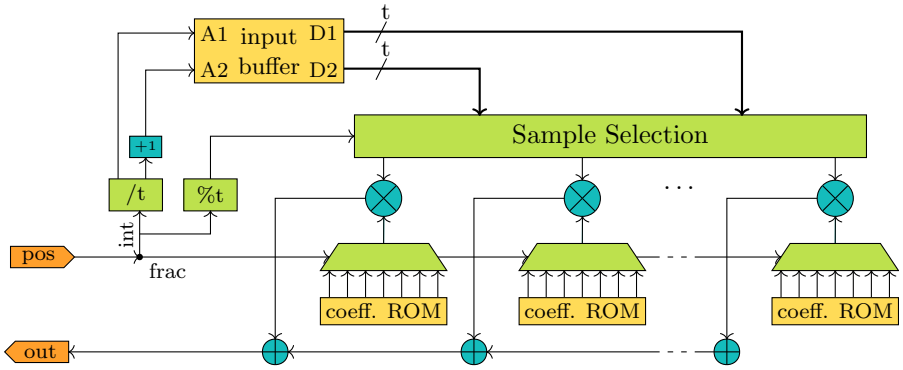

Fig. 3. Interpolator architecture. The incoming position is split into an interger and a fractional part. The integral part is used to fetch t samples from the dual-port memory, centered around the requested position. The samples are multiplied by coefficients selected by the fractional part, and summed up to form the interpolation result.

t samples, centered around the integer position, are extracted. The fractional part is used to select a set of coefficients which a pre-calculated and stored in a ROM. Every sample is multiplied with one of the coefficients; and the sum of these forms the interpolation result.

Outside of the submodules, apart from the control unit, there are only memory interfaces and other bus infrastructure. This is highly dependent on the platform and not in the scope of this paper. Although both implementations are proven to work on the system level, comparisons will focus on the internals of the submodule (including the interpolator), since they are closely matched among both implementations.

3 Measurements

In order to assess the expressivity of Chisel we performed measurements of the code size (number of lines of code) and the readability of the generated Verilog code. Both investigation considered only the backprojection submodule described above, which is implemented similar in the VHDL and Chisel version of the code. The remaining parts of the systems, namely the control logic and bus infrastructure was realized differently between both versions and would therefore be meaningless to compare. While the Chisel implementation describes the same hardware design, it was created independently of the existing VHDL code, as opposed to a mere transcription between languages. Both designs have been carried out by experienced hardware designers.

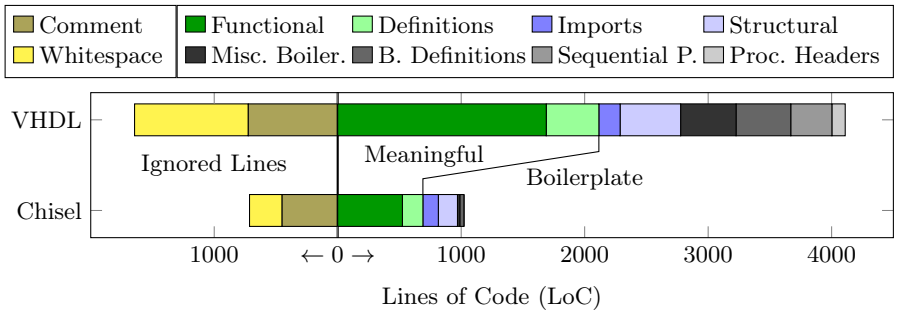

Fig. 4. Bar chart of the code size analysis result, divided into ignored lines, meaningful code and boilerplate code.

3.1 Code size

The Chisel implementation of the Backprojection submodule consists of 19 files with 1736 LoC in total, whereas the VHDL description consists of 25 files and 5755 LoC. To further investigate the differences of the two languages, we classified every line of code into one of the following categories:

Whitespace (WS) Empty lines
Comment (COM) Lines containing only source code comments
Structural (STR) Lines containing only structual syntax elements, such as opening or closing braces (`{`/`}`) or `begin`/`end`
Imports (IMP) Import statements for other namespaces/packages
Boilerplate Definitions (BD) Lines consisting only of a declaration of a `signal` (VHDL) or `Wire` (Chisel), without any value assignment
Sequential Process (SQ) *(VHDL only)* Lines attributed to the sequential process of the Two Process design methodology [10] which separates the register description from combinatorial logic
Process Headers (PH) *(VHDL only)* Lines containing process headers, including sensitivity lists
Misc. Boilerplate (MB) Miscellaneous statements required for syntax, but containing no actual information; i.a. `component` declarations in VHDL
Definitions (DEF) Definitions of types, entities, `Bundles` and functions
Functional code (FUN) Everything else, i.e. code that describes the system's actual function, including module instantiations and signal connections

Examples for every category, as well as the distributions of the lines over the categories can be found in Table 1. A graphical overview of the line counts per category is shown in Fig. 4. Additionally, Appendix A contains a code example with classifications in VHDL and Chisel. The code examples are stylistically similar to the real implementations.

For all further analyses, lines from the *Whitespace* and *Comment* categories will be excluded. Furthermore, the categories *Structural*, *Imports*, *Boilerplate*

Definitions, *Sequential Process*, *Process Headers*, and *Misc. Boilerplate* can be summed up as *Boilerplate code*; *Definitions* and *Functional code* as *Meaningful code*. With this high-level separation, it can be seen that the amount of *Meaningful* code is reduced by 67 % comparing the VHDL and Chisel implementation, while the the amount of *Boilerplate* code decreased by 83 %. This gives a first indication that Chisel generally allows for more compact and expressive code, while especially verbose and redundant code is reduced.

Concerning the meaningful code, most of the surplus lines in the VHDL implementation can be attributed to individual signal connections among entities. Chisel allows complete bi-directional `Bundles` to be connected using the bulk-

Table 1. Code categories with examples, line count and their prevalence in the two implementations. Examples annotated using the markers can be found in Appendix A.

Category	Marker	Examples	LoC	Frac. [%]
		Chisel		
Whitespace	WS		264	15.2
Comment	COM	`// single line comment`	450	25.9
		`/* multi line comment ...`		
Functional	FUN	`val reg_b = RegNext(c);`	524	30.2
		`inst_d.io.x <> inst_e.io.y;`		
Definitions	DEF	`class X (...) extends Module`	166	9.6
		`val in_a = Input(SInt(n.W));`		
Imports	IMP	`import chisel3._`	125	7.2
Structural	STR	`)};`	155	8.9
Misc. Boilerplate	MB	`val io = IO(new Bundle {`	19	1.1
Boilerplate Def.	BD	`val wire_a = Wire(Bool());`	33	1.9
		VHDL		
Whitespace	WS		921	16.0
Comment	COM	`-- single line comment`	724	12.6
Functional	FUN	`constant B : integer := C+1;`	1689	29.3
		`sig_b <= not sig_c;`		
Definitions	DEF	`in_a : in std_logic;`	426	7.4
		`entity X is`		
Imports	IMP	`use ieee.std_logic.all;`	172	3.0
Structural	STR	`begin`	490	8.5
		`end if;`		
Misc. Boilerplate	MB	`architecture Y of X is`	449	7.8
		`port map (`		
		`component Z ...`		
Boilerplate Def.	BD	`signal sig_b : std_logic;`	443	7.7
		`variable var_k : real;`		
Sequential Process	SQ	`process (clk)`	333	5.8
		`if rising_edge(clk) then`		
Process Headers	PH	`process (a, b, c, d)`	108	1.9

Table 2. Number of signal names in the Verilog code generated by Chisel, split by readability.

Category	Example	Count	Fraction [%]
Clear	`resetAccu`	6650	86.0
Derived	`dmaIndex_T_5`	478	6.2
Opaque	`GEN_47`	607	7.8

connect operator (<>), which significantly reduces the number of lines required. Another cause for the reduced code size in Chisel is the implicit connection of clock and reset signals, as well as the implementation of the lookup table generation for the CORDICs and the interpolator, which is very verbose in VHDL.

The additional boilerplate code in VHDL is twofold. First, in the VHDL code a `component` declaration exists for every entity referenced; in Chisel this is not required. Second, register definitions are much more verbose in VHDL than in Chisel. In VHDL, following the Two Process methodology, a typical register definition consists of the declaration of two signals, an entry in the sequential process and the sensitivity list of the combinatorial process, and a reset definition. In Chisel, a register, together with its reset value, can be declared in one line of code; the clock and reset connections are implicit. Generally, Chisel allows signals to be declared inline with their driving logic, while in VHDL declaration and assignment have to be split up.

3.2 Transparency of Generated Code

Generally, the Verilog code files generated by Chisel are not intended to be human-readable and should be considered an opaque artifact. On the other hand, the warnings reported in the synthesis process and messages from timing analysis always refer to this generated code. It is therefore important to be able to find the Chisel source code location which is related to a certain Verilog line. A classic example is timing closure, where a failing path may be reported, consisting of Verilog signal names. To measure how easy it will be to find the root cause of a timing violation, we counted the number of signal definitions in the generated Verilog code and divided them into three categories: *Clear* names, having exactly the name of a Chisel signal; *derived* names, starting with a Chisel name but having a suffix in the form of "`_T_1`"; and *opaque* names, which are purely artificial and have the form like "`GEN_57`".

Numerous wires from the latter category are used to describe the content of read-only memories (e.g. the instruction memory of the microcontroller). This memory is inferred as Block-RAM by the synthesis tool, the wires therefore were excluded from the counting. We performed this measurements using Chisel 3.5.0. The results are reported in Table 2. Only 7.8 % of the signal names were found to be opaque; most of them occurred in more abstract code like an AXI register file generator.

4 Experience using Chisel

In this section we summarize our own experience while working with Chisel. The author had no prior knowledge in Chisel and used the re-implementation of the described Backprojection module as an opportunity to learn Chisel.

We found working with Chisel to be highly productive. As shown in Section 3.1 VHDL is overly verbose when it comes to declaring registers or connecting large busses. Also the fact that only synthesizable hardware can be described in Chisel made the coding experience more pleasant; the feeling of "fighting the tool" to reach a certain implementation was removed.

As described above, our design includes the *PacoBlaze* softcore [8], which is only available as Verilog code. We used a Chisel `BlackBox` to include it in the design. This generally worked well, although simulation with *ChiselTest* and Verilator was only possible after combining all source files of the PacoBlaze into a single file. Generally, the workflow of writing small modules and thoroughly testing them using unit tests with ChiselTest has proven to be very successful. In many cases, the expected test results and stimuli could be generated using a software implementation of the algorithm in Scala, so that all code was implemented in the same language. Compared to the typical workflow, where test vectors are generated in MATLAB or Python and then executed using a Verilog/VHDL testbench, this approach turned out to have less overhead, encouraging the designer to actually include tests for every single module.

Timing closure for the design was easy to achieve. The mapping from reported timing violations in the synthesis tool (*Xilinx Vivado* in our case) to the corresponding Chisel source code location was generally easy. As shown above, only about 8 % of the wires in the generated Verilog code used generic names, therefore the reported paths always included enough information to find their context. To insert a register in a problematic path, it often was enough to wrap the expression with `RegNext()`. When the failing path was inside an AXI-Stream, a more complex register slice needed to be inserted. The object-oriented approach of Chisel allowed us to define a concise generator function to insert an arbitrary number of these register slices. Combined with the bulk-connect operator of Chisel, this could be applied with a single line of code, similar to a simple register. We generally found the iteration speed in timing closure very high.

While implementing the CORDIC algorithm we found the possibility to execute arbitrary Scala code during elaboration time very useful. Using this, we could calculate the CORDIC gain constant and the angular lookup tables for arbitrary iteration counts on the fly, without falling back to hard-coded constants in the source code.

One downside of Chisel is its steep learning curve. This is partly due to Chisel being a relatively young language, so not much learning material being available yet. Also the documentation of Chisel is sparse in some parts. In line with the findings of Lennon et al. and Alon et al. [9, 1], we also see a requirement to have some experience in modern programming languages—something not necessarily found among hardware designers—to fully harness the power of Chisel. Since

the full knowledge of digital design is still required, this raises the bar for new designers. On the other hand we would not consider that a shortcoming of Chisel, but as an effect coming from the situation the field of digital design is in: Chisel can be used on a very similar level as Verilog or VHDL, and then mainly helps to reduce the verbosity. For complex and parametrizable designs, as today's IP cores, a certain complexity in the hardware description will always be required. Chisel here allows for highly sophisticated hardware generators which enable a flexibility that might be unreachable using a classic Tcl+Verilog design flow.

5 Conclusion

In this field report, we analyzed the practical use of the Chisel hardware description language. As an example design, we used an implementation of the Backprojection algorithm for Synthetic Aperture Radar (SAR). To assess the advantages over traditional languages, we examined the differences to a VHDL implementation by classifying every line of code into one of several categories. The analysis showed that the number of code lines was reduced by about 74 % from the VHDL to the Chisel implementation; boilerplate code was reduced by 83 %. This indicates that Chisel is more expressive and allows to describe the same design in fewer lines of code. Especially, the amount of boilerplate code was reduces significantly.

Furthermore, we analyzed the Verilog code generated by Chisel and found only 7.8 % of the signals to have generic, meaningless names. Thus, few problems are to be expected when trying to find the Chisel code corresponding to a signal in the generated code.

Finally, we gave a personal impression of our experience with Chisel. We generally found Chisel to be very productive and powerful, but also saw the need of some programming knowledge to fully unleash the power of the language. It allows for higher productivity and flexible reuse of designs.

In our opinion, Chisel is an exciting new technology that has the power to free the field of digital design from the corset of traditional hardware description languages. It is by far not limited to processor design and thus should be considered for all fields of digital hardware design.

References

1. Alon, E., Asanović, K., Bachrach, J., Nikolić, B.: Open-source EDA tools and IP, a view from the trenches. In: Proceedings of the 56th Annual Design Automation Conference 2019. ACM (jun 2019). https://doi.org/10.1145/3316781.3323481
2. Arcas-Abella, O., Ndu, G., Sonmez, N., et al.: An empirical evaluation of high-level synthesis languages and tools for database acceleration. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–8. IEEE (2014). https://doi.org/10.1109/FPL.2014.6927484
3. Bachrach, J., Vo, H., Richards, B., et al.: Chisel: constructing hardware in a scala embedded language. In: DAC Design Automation Conference 2012. pp. 1212–1221. IEEE (2012). https://doi.org/10.1145/2228360.2228584

4. Cholewa, F., Pfitzner, M., Fahnemann, C., Pirsch, P., Blume, H.: Synthetic aperture radar with backprojection: A scalable, platform independent architecture for exhaustive fpga resource utilization. In: 2014 International Radar Conference. pp. 1–5. IEEE (2014). https://doi.org/10.1109/RADAR.2014.7060455
5. Curlander, J.C., McDonough, R.N.: Synthetic aperture radar, vol. 11. Wiley, New York (1991)
6. Duersch, M.I.: Backprojection for synthetic aperture radar. Ph.D. thesis, Brigham Young University (2013)
7. Im, J., Kang, S.: Comparative analysis between verilog and chisel in risc-v core design and verification. In: 2021 18th International SoC Design Conference (ISOCC). pp. 59–60. IEEE (2021). https://doi.org/10.1109/ISOCC53507.2021.9614007
8. Kocik, P.B.: Pacoblaze - a synthesizable behavioral verilog picoblaze clone. https://bleyer.org/pacoblaze/, accessed: 31/01/2022
9. Lennon, P., Gahan, R.: A comparative study of chisel for fpga design. In: 2018 29th Irish Signals and Systems Conference (ISSC). pp. 1–6. IEEE (2018). https://doi.org/10.1109/ISSC.2018.8585292
10. Pedroni, V.A.: Finite state machines in hardware: theory and design (with VHDL and SystemVerilog). MIT press (2013)
11. Richar Lin, Kevin Laeufer, C.M., et al.: Chiseltest: The official testing library for chisel circuits. https://github.com/ucb-bar/chiseltest, accessed: 31/01/2022
12. Volder, J.E.: The cordic trigonometric computing technique. IRE Transactions on Electronic Computers **EC-8**(3), 330–334 (1959). https://doi.org/10.1109/TEC.1959.5222693

A Code examples

This appendix provides the code of an example design in Chisel and VHDL to make our methodology more clear. Every line of the source code has been classified according to Table 1. The code presented here is not part of the original source used for this study, but is written in a similar style. In can be seen, that especially the amount of boilerplate code (categories MB, BD, and SQ) is reduced, but the Chisel code is also less verbose in general.

A.1 Chisel code

```
DEF   package example
WS
IMP   import chisel3._
      import chisel3.util._
WS
DEF   class Example(n : Int) extends Module
STR   {
BD      val io = IO(new Bundle {
DEF       val in_val = Input(UInt(n.W))
          val out_val = Output(UInt(n.W))
STR     })
WS
FUN     val bypass = RegNext(io.in_val, init = 0.U)
        val in_masked = io.in_val & "b111".U
WS
FUN     val exampleInst = Module(new Example2())
        exampleInst.io.data_in := in_masked
WS
FUN     io.out_val := Mux(io.in_val > 5.U, exampleInst.io.data_out, bypass)
STR   }
```

A.2 VHDL code

```
IMP   library IEEE;
      use IEEE.std_logic_1164.all;
      use IEEE.numeric_std.all;
WS
DEF   entity example is
STR     generic (
DEF       N : integer := 8
STR     );
        port (
DEF       clk     : in  std_logic;
          reset   : in  std_logic;
          in_val  : in  std_logic_vector(N-1 downto 0);
          out_val : out std_logic_vector(N-1 downto 0)
STR     );
      end entity;
WS
MB    architecture rtl of example is
BD      component example2
          generic ( N : integer );
STR       port (
BD          clk      : in  std_logic;
            reset    : in  std_logic;
            data_in  : in  std_logic_vector(N-1 downto 0);
            data_out : out std_logic_vector(N-1 downto 0)
STR       );
        end component;
DEF     signal c_bypass, n_bypass : std_logic_vector(N-1 downto 0);
        signal in_masked : std_logic_vector(N-1 downto 0);
        signal comp_out : std_logic_vector(N-1 downto 0);
STR   begin
FUN     in_masked <= in_val and std_logic_vector(to_unsigned(2#111#, N));
        n_bypass <= in_val;
WS
FUN     example_inst : example2
STR       generic map (
FUN         N  => N
STR       ) port map (
FUN         clk      => clk,
            reset    => reset,
            data_in  => in_masked,
            data_out => comp_out
STR       );
WS
SQ      SEQ : process(clk)
        begin
         if rising_edge(clk) then
          if reset = '1' then
            c_bypass <= (others => '0');
          else
            c_bypass <= n_bypass;
          end if;
         end if;
        end process;
WS
PH      MUX : process (c_bypass, in_val, comp_out)
STR     begin
FUN       out_val <= c_bypass;
          if unsigned(in_val) > 5 then
            out_val <= comp_out;
STR       end if;
        end process;
      end rtl;
```