Scala Defined Hardware Generators for Chisel

Martin Schoeberl*, Hans Jakob Damsgaard†, Luca Pezzarossa*, Oliver Keszocze*,
Erling Rennemo Jellum‡, Scott Beamer§
*Technical University of Denmark, Denmark †Tampere University, Finland ‡Norwegian University of Science and Technology, Norway §University of California, Santa Cruz, USA Emails: {masca, lpez, olike}@dtu.dk,
hans.damsgaard@tuni.fi, erling.r.jellum@ntnu.no, sbeamer@ucsc.edu

***Abstract*—We describe digital hardware designs in hardware description languages such as VHDL and SystemVerilog. Both languages were developed in the 1980s and, although regularly updated, are still in the style of their time. They lack the constructs to write more configurable generators than just the number of bits for an operation. Based on Scala, Chisel is a hardware construction language that helps to write hardware generators.**

Hardware generators are not a new idea. Scripting languages, such as Perl and TCL, are often used to generate VHDL or Verilog code from other sources of system description. However, mixing two languages and embedding VHDL or Verilog strings in generator code is not scalable.

As Chisel is embedded in Scala, we can write the generators using the same language/environment as we use to describe the digital logic. This paper explores different examples and patterns to describe parameterizable hardware generators. We are confident that practices from software development can improve the productivity of hardware designers to build and test the next billion transistor chips.

***Index Terms*—hardware generators, open-source hardware, and open-source digital design tools.**

I. Introduction

The state-of-the-art digital hardware design uses hardware description languages (HDLs). The two most used languages are VHDL and SystemVerilog. Both languages are well-established and have been around for about 40 years, with VHDL being standardized by the IEEE in 1987 [1], and SystemVerilog resulting from the merger of the Verilog language invented in the 1980s and transferred to the IEEE in 1995 [2] and its corresponding SystemVerilog extensions in 2005 [3]. However, the two languages have not picked up on developments in software languages, such as object orientation, functional programming, or dynamic typing for describing hardware. SystemVerilog supports object-oriented design patterns only for simulation. Therefore, they are not expressive enough to write hardware generators.

The current practice is to use other languages, mostly scripting languages such as Perl or Python, to write hardware generators [4], [5]. The approach is to read some specification (generator parameters) and output HDL code in the target language. That code is usually embedded as strings in the generator scripting language. This is a brittle, error-prone approach. The main issue is that the HDL code is represented as strings, where no syntax checks are performed [6], [7]. This complicates the maintenance of the generators. Furthermore, upgrading to a newer version of the target language is cumbersome. With a hardware construction language, the expressiveness is improved, and the compiler can validate the syntax [8], [9]. The Chisel runtime spills out the Verilog code needed for synthesis and simulation.

Let us look at a real-world problem of one of the authors as a concrete example of a table generation issue. Before switching to Chisel to teach undergraduates digital design, we used VHDL. In that course, students must display a binary number in decimal on a 7-segment display. We can use a table to convert a binary number into a binary-coded decimal (BCD). For a two-digit display, that table has 100 entries. There is a case for generating that table with a script. We provided the students with a small Java program to generate this table. That program has about 100 lines of code, mixing the table generation logic with strings containing VHDL code. The core of that generation was just six lines of code. With Chisel, we generated the same table in four lines of code instead of 100 lines. Furthermore, this generation code is now part of the hardware construction; no additional script run to generate VHDL source code is needed.

This paper proposes to use Chisel together with Scala to write hardware generators. Writing generators instead of concrete register-transfer level (RTL) code makes designs reusable and reduces debugging time. This paper explores hardware generators by providing several examples. The examples illustrate how simple generators can be. For example, the code for a parameterizable fair arbitration circuit is around 30 lines and fits into one column of this paper.

Writing hardware generators is more like software development than classic hardware design. We assume that future hardware engineers will also be well-educated in classic software development and will explore the domain of hardware generators in ways that are not yet imaginable.

While previous papers and usage in projects have demonstrated the use of Chisel and Scala for hardware generators, the knowledge is still fragmented across project source codes, case studies, tool documentation, and teaching material. This paper aims to consolidate these ideas, enrich them with new examples, and present them in a way that is accessible to both researchers and educators. Our primary contribution is to explore and demonstrate the potential of hardware generators using Chisel and Scala and to inspire further research in this domain.

This paper is an extension of [10]. Besides reworking all sections and extending some, we have added the following contributions:

- To bridge research and practice, we teach hardware generators as part of a course on agile hardware design. We included a dedicated section discussing our experiences in teaching this topic.
- One common problem of SoC design and firmware writing is a common framework to generate all needed artifacts. We added a subsection on a framework in Chisel that solves this problem for hardware-software co-design.
- We added a section on how testing in the Scala/Java world can speed up developing and debugging digital designs written in Chisel.

This paper is organized into eleven sections. The following section gives background on Chisel. Section III provides simple examples of hardware generators. Section IV explores how functions in Chisel can simplify and reduce code to describe circuits. Section V adds object-oriented programming to the hardware description. Section VI explores how functional programming from Scala can be used to write flexible hardware generators. Section VII provides an example of how the functional and object-oriented programming in Chisel/Scala can help build applications. Section VIII explore some possibilities when the test and verification code can be written in Scala and having open-source Java libraries available. Section IX describes how we can transfer the research knowledge of hardware generators into education. Section X presents related work, and Section XI concludes the paper.

II. Chisel

Chisel is a domain-specific language embedded within Scala, designed for the description of digital circuits [8]. Unlike traditional HDLs, such as VHDL and SystemVerilog, which primarily describe digital circuits at the register-transfer level, Chisel leverages the capabilities of a general-purpose programming language. The embedding in Scala allows Chisel to utilize powerful language features, including type parameterization and elements of both object-oriented and functional programming paradigms. Consequently, Chisel enables the creation of hardware generators, which help design complex hardware systems more efficiently than using traditional HDLs.

One of the key advantages of Chisel's integration with Scala is its execution mode. Chisel is a Scala library, and when a Scala program that uses this library is executed, it generates a digital circuit. This process introduces another step in the compilation pipeline, transitioning from a hardware description to the bit-stream for an FPGA or the masks for ASIC production. This step, known as "elaboration," executes the necessary operations to implement parameterizable hardware generators by constructing a graph of hardware elements and their interconnections.

Embedding generator code within the hardware description simplifies the design process. A straightforward build script is sufficient to execute the Chisel code, eliminating the need for external scripts. This integration ensures that the hardware description and the generator code are always in sync, reducing the risk of incomplete or missing files. For instance, one of the authors encountered a project where three VHDL files needed to be generated by a Java program. The project faced several complaints due to missing VHDL files, which prevented synthesis. Such issues are mitigated when the generator code is integral to the hardware description, ensuring completeness and consistency.

Moreover, the high-level abstractions of the Chisel/Scala combination enable designers to focus on the architectural aspects of the hardware design rather than the low-level implementation details. This abstraction enhances productivity and improves the maintainability and scalability of the hardware designs. By leveraging Scala's rich ecosystem, Chisel provides a robust and flexible environment for digital circuit design, making it a powerful tool for modern hardware development.

To most users, Chisel appears to be simply a language, but it is powered by a set of cooperating tools similar to a compiler flow. The Chisel frontend is a Scala library that is directly utilized by a user design, which is in fact a valid Scala program. The frontend produces FIRRTL [11], an intermediate representation to enable the rest of the processing by the tool flow. That FIRRTL is progressively lowered and processed according to both the intended output (e.g. Verilog) and user-requested transformations. CIRCT is the current hardware intermediate representation [12], [13], and it includes FIRRTL as a dialect, and its `firtool` consumes FIRRTL to produce Verilog or SystemVerilog.

Chisel provides a productivity advantage by allowing its users to use the power of Scala, a general-purpose programming language, to programmatically construct hardware. Hardware generators written in Chisel simply instantiate hardware components and connect them together, and the programming glue is to control what is instantiated and how they are connected. Thus, the use of Chisel introduces no power, performance, or area overhead compared to writing the design directly in low-level RTL. There is no high-level synthesis, and the overall efficiency is determined by the architecture of the design produced by the generator. The price of Chisel's productivity advantage is the complexity of writing a generator, not efficiency. Generators can be more complex to write than direct designs because they consider multiple design variants and the logic to choose between them. Many designers can sidestep this challenge by using available generators without needing to create their own generators.

III. Simple Examples

To introduce Chisel, we begin by presenting some simple examples. While they are not very advanced concerning the hardware or the Chisel features used, they provide practical building blocks that are practical and can save development and debugging time.

A. Table Generation

Tables containing constant values generated at compile time are helpful in many situations. They allow the realization of small combinational circuits as lookup tables (LUTs) or the description of read-only memory (ROM). Depending on the language used, such tables can be created as initialization functions of vectors or arrays or using switch/case statements.

Traditional HDLs support the declaration and initialization of array-type constants that serve as tables. Unfortunately, the associated syntax in VHDL is rather verbose [14, Chap. 4] but offers many configuration options. Conversely, SystemVerilog provides fewer options for advanced initialization [15, Sec. 9.9.1]. Small experiments we carried out suggest that some compilers cannot synthesize tables for iterative patterns referencing previously set array elements.

Scala, and consequently Chisel, offers a very flexible table generation. The following code shows how to generate the binary to BCD conversion table, that was mentioned in the introduction.

```
val table = Wire(Vec(100, UInt(8.W)))
for (i <- 0 until 100) {
  table(i) := (((i/10)<<4) + i%10).U
}
```

B. Assembler

Embedded systems (embedded processors) usually come with static code in assembly language, such as a boot program. One approach for designing such a system is to have some HDL code describing the hardware and initializing memory with the static code. Additionally, a separate assembler software generates the static code, i.e., the binary file that is read to create the ROM.

Using Scala, integrating the whole process into a single language is easy by writing the assembler in Scala itself. We successfully used this approach when implementing the two small utility processors Lipsi [16] and Leros [17]. The assemblers that generate the ROMs for the processors in terms of Chisel `Vec`s are implemented in very few lines of code.[1]

C. Register Address Mapping

In large-scale hardware design, a common practice involves automatically generating memory-mapped configuration registers from a specification file, such as XML or JSON. This specification file outlines the registers' memory-mapped addresses, sizes, sub-fields, and access modes (read-only, read-write, write-only). Typically, this file is created and modified using a graphical user interface that visually represents the register structure or is derived from a standardized IP specification scheme like IP-XACT [18].

Scripts written in languages like Perl or Python convert this specification into practical hardware and software components. These scripts parse the specification file and generate the corresponding HDL code (VHDL or Verilog) and software header files necessary for register definition and documentation.

This automated process offers several significant benefits. It ensures consistency across the HDL code, software headers, and documentation, minimizing the risk of manual coding errors and discrepancies. Additionally, it enhances maintainability, as any changes to the specification file can be quickly propagated to all generated outputs, keeping the hardware and software components synchronized.

[1]See as an example: https://github.com/schoeberl/lipsi/blob/master/src/main/scala/lipsi/util/Assembler.scala

However, this approach often relies on custom, ad-hoc scripts, which can introduce additional complexity in script maintenance and scalability. The lack of a unified generation environment can make managing and updating these scripts challenging as the design evolves.

Chisel addresses these challenges by providing a high-level design environment that integrates the generation process within the design itself. The processing of specification files can be written in Scala and executed as part of the hardware generation. For example, we use XML parsing in Scala to read the T-CREST [19] system's configuration and generate the hardware accordingly. We can also imagine using Scala's XML parsing to read in IP-XACT [18] specifications.

IV. Functions

Functions can be used in Chisel to generate hardware components during the elaboration phase. Specifically, they can return flexible and modular hardware by encapsulating reusable logic. Unlike Chisel `Module`s, which are Scala classes, functions are lightweight and avoid the overhead of a class definition. An additional advantage is the possibility of using functions as parameters of higher-order ones, leveraging functional programming for powerful dynamic hardware generation.

While the idea of using functions to describe reusable hardware is not exclusive to Chisel, its implementation is different from traditional HDLs. Both VHDL and SystemVerilog also allow the use of functions for logic encapsulation. However, their use is primarily restricted to purely combinational logic [14, Chap. 4] [15, Chap. 10]. VHDL functions and procedures resemble processes, while SystemVerilog functions and tasks resemble `always` (or `always_comb`) blocks.

In contrast, Chisel functions allow defining both combinational logic and registers (sequential logic). A simple example is the following circuit, which detects when a signal transitions from low to high by comparing its current value with the stored value from the previous clock cycles.

```
def rising(v: Bool) = v & !RegNext(v)
```

Listing 1 introduces a more advanced example with the function `regPipeline`, which generates a pipeline of registers to introduce a delay to a signal. This delay, in clock cycles, is defined by the (`pipeDepth`) parameter, making this function especially useful in pipelined architectures where such delays need to be configured dynamically during elaboration. If the specified depth is zero or negative, the function bypasses the pipeline and directly forwards the input signal. An interesting aspect showcasing the flexibility and adaptability of this approach is that the type `T` of the delayed signal is also configurable, enabling the function to pipeline complex structures like `Vec` and `Bundle`).

The function in Listing 1 is implemented following a traditional imperative approach, using a loop to connect each register in the pipeline. Alternatively, Scala's support for functional programming offers a cleaner and more expressive solution by replacing the loop with the higher-order function `foldLeft`, as shown in the following:

```
def regPipeline[T <: Data]
  (input: T, pipeDepth: Int, init: T): T = {
    if (pipeDepth <= 0) {
      input
    } else {
      val pipeReg = RegInit(
          VecInit(Seq.fill(pipeDepth)(init)))
      pipeReg(pipeDepth - 1) := input
      for (i <- 0 until pipeDepth - 1)
        pipeReg(i) := pipeReg(i + 1)
      pipeReg(0)
    }
}
```

Listing 1: A function that generates a pipeline of registers for a generic type with specified depth and initial values.

```
def regPipeline[T <: Data]
  (input: T, depth: Int, init: T): T = {
    (0 until depth).foldLeft(input) {
      case (acc, _) => RegNext(acc, init) }
}
```

Chisel functions are not limited to producing single outputs. Scala tuples can be used to return multiple values. For example, the following function describes a circuit that identifies both the rising and falling edges of an input signal, combining the outputs into a single tuple.

```
def edges(v: Bool): (Bool, Bool) = {
  val delayedV = RegNext(v)
  val rising = signal & !delayedV
  val falling = !signal & delayedV
  (rising, falling)
}
```

In summary, Chisel functions offer an efficient and streamlined way to design parameterized hardware and can be used as an alternative to implementing small modules.

V. Object-Oriented Programming

Scala is built on Java and fully supports its object-oriented programming features, including polymorphism, inheritance, and encapsulation. These features are available when writing flexible hardware generators in Chisel.

With abstraction, designers can focus on high-level features rather than low-level implementation details. Inheritance from abstract classes and traits lets us define generic hardware interfaces later specialized by concrete implementations. For example, a generic FIFO may be defined with an abstract class in which different implementations (e.g., a bubble FIFO, a memory-based FIFO, or others) may extend, as exemplified in [20]. VHDL and SystemVerilog support reusable interfaces, with a combination of records and views in the former [21, Section 6.5] and the interface construct in the latter [3, Chap. 20], but not inheritance for hardware description.

Generics is another concept that allows for writing code that can operate on different data types, with the type being the parameter. Scala and Chisel support generics through type erasure in the underlying JVM.[2] VHDL and SystemVerilog have provided similar concepts since 2008 [14] and 2005 [3], respectively. Chisel permits using generic types for general hardware structures, including the `Decoupled` ready/valid interface with a generic parameter for the data type it wraps. The register pipeline and FIFO examples above rely on generics for any data type.

Inheritance allows for the direct reuse of (elements of) existing hardware modules when creating new ones. It is key to Chisel as any user-defined module extends the `Module` base class and inherits all its members and methods. Similarly, it is useful when constructing variations of a hardware component with a common interface, for example, the FIFOs above, which may have some common logic too.[3] Notably, despite their support for shared interfaces, neither traditional HDL permits inheritance from one module to another

Chisel modules encapsulate their internal hardware state and logic similarly to any other Scala class, exposing only the necessary interfaces (e.g., through its IO bundle). By default, all `val` and `var` fields declared in a Scala class' body are `public`, meaning they are visible outside the class. This also applies to Chisel registers and wires. However, their values are not readable or writable from outside the module, aligning with encapsulation principles.

VI. Functional Programming

Functional programming lifts functions to be first-class citizens of the programming language. This allows functions to be passed as parameters to so-called higher-order functions. A common use case for this is when operating on a collection (`Vec` in Chisel) of data. As the motivating example, we consider summing up all values in a vector. We first define the hardware to carry out the addition via the function definition

```
def add(a: UInt, b: UIn): UInt = a + b
```

We assume that the values we want to sum are referenced by the variable `values` of Chisel type `Vec`. We then pass the `add` function/hardware to the `reduceLeft` higher-order function operating on `values`

```
val sum = values.reduceLeft(add)
```

The function is called `reduceLeft` as it reduces the values of the vector to a single value by iterating over the vector, applying the provided binary operation (`add` in our case) on the values. It begins at the left-hand side of the vector. The generated hardware is shown in Fig. 1a.

However, the generated hardware is not ideal. Generating an adder tree would result in hardware with a shorter combinational delay. For this, Chisel offers the `reduceTree` function that results in the adder tree shown in Fig. 1b.

Explicitly defining and naming a function for simple tasks such as addition is cumbersome. Scala allows to define anonymous function on-the-fly using *function literals*:

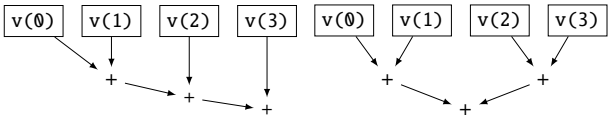
```
(param1, param2, ...) => function body
```

So, the previous example could be written as

```
val sum = values.reduceLeft((a, b) => a + b)
```

[2] https://en.wikipedia.org/wiki/Generics_in_Java

[3] See: https://github.com/freechipsproject/ip-contributions/tree/master/src/main/scala/chisel/lib/fifo



(a) Reducing from the left. (b) Reducing as a tree.

Fig. 1: Different ways to reduce a vector of values to a single value.

instead. To reduce the boilerplate, Scala offers the wildcard "_" to indicate an operand. This wildcard expands `f(_, _)` to

```
(param1, param2) => f(param1, param2)
```

allowing to further simplify the code of the example to

```
val sum = values.reduceLeft(_ + _)
```

We will use function literals throughout the rest of the paper.

A. Finding the Minima Example

Building upon the reduction functions and function literals, we present more elaborate examples of using Chisel to implement common operations: finding the minimal value in a `Vec`tor of values.

The function literal used for doing so takes two inputs `x` and `y` and forwards the smaller of both. We realize this by instantiating a multiplexer via the `Mux` object. The following single line of code then gives the full solution to the task.

```
val min = vec.reduceTree((x, y) =>
  Mux(x < y, x, y))
```

To further increase the function of the circuit, we extend it so that it not only selects the smallest value but also returns the position (i.e., index) of the smallest value within the vector. For this, we define a custom `Bundle` type `ValWithIndex` that stores a value and its index within the vector. We then create a new vector of `ValWithIndex`'es that we fill using a for loop with the original values and their corresponding indices. The function that we then pass to `reduceTree` compares the value component `v` of the supplied bundle but returns a multiplexer that passes down the whole bundle. The corresponding code is shown in Listing 2.

The manual process of creating the `IndexedVec` using a custom `Bundle` is cumbersome. We can use Scala's functional programming features to simplify the code. Listing 3 produces hardware that is functionally equivalent to that from Listing 2. Here, multiple functions are chained, resulting in a larger combinational circuit.

The `zipWithIndex` function creates from a sequence of values a sequence of tuples containing the original values and their corresponding index within the original vector. The index is a Scala integer that must be cast into a Chisel `UInt`. This is done via the `map` function that transforms a sequence of values by replacing each entry with the result of a function call that received the current entry as a parameter. These two function calls replace the explicit `indexedVec` and for loop in Listing 2, simplifying the code.

Finally, the `reduce` function generates the multiplexer chain to find the minimum in the generated sequence by comparing

```
// Parameters n and w defined by the encapsulating
↪  class
class ValWithIndex extends Bundle {
  val v = UInt(w.W)
  val idx = UInt(8.W)
}

val indexedVec = Wire(Vec(n, new ValWithIndex()))
for (i <- 0 until n) {
  indexedVec(i).v := vec(i)
  indexedVec(i).idx := i.U
}

val res = indexedVec.reduceTree((x, y) =>
  Mux(x.v < y.v, x, y))

val minVal = res.v
val minIdx = res.idx
```

Listing 2: The code for the minimum search, including the index (essential part only).

```
val resFun = vec.zipWithIndex
  .map(x => (x._1, x._2.U))
  .reduce((x, y) =>
    (Mux(x._1 < y._1, x._1, y._1),
     Mux(x._1 < y._1, x._2, y._2)))

val minVal = resFun._1
val minIdx = resFun._2
```

Listing 3: Using `zip` and tuples for the minimum search.

elements two at a time and returning a tuple of two Chisel `UInt`s.

The former example returns hardware with Chisel types but uses a Scala `Vector` to hold intermediate results. However, a Scala vector does not contain the `reduceTree` method; it is part of Chisel's `Vec`. We can substitute the Scala tuple with a Chisel `MixedVec`. A `MixedVec` is an indexable collection that can contain different Chisel types, a construct similar to a Scala tuple.

Listing 4 shows how to switch between Chisel and Scala types. With the `zipWithIndex` we use Chisel types `MixedVec` and create a Scala collection (a `Vector`). We then convert the Scala `Vector` into a Chisel `Vec` by invoking `VecInit` with the Scala `Vector` as a parameter. Then, we can use the tree reduction to find the minimum and the index, returned as a Chisel `MixedVec`.

A more elegant solution would be to add `reduceTree`

```
val scalaVector = vec.zipWithIndex
  .map(x => MixedVecInit(x._1, x._2.U))
val resFun2 = VecInit(scalaVector)
  .reduceTree((x, y) =>
    Mux(x(0) < y(0), x, y))

val minVal = resFun2(0)
val minIdx = resFun2(1)
```

Listing 4: Using a `MixedVec` to hold the value and index as Chisel types.

```
class Arbiter[T <: Data]
  (n: Int, private val gen: T) extends Module {
    val io = IO(new Bundle {
      val in = Flipped(
        Vec(n, new DecoupledIO(gen)))
      val out = new DecoupledIO(gen)
    })
    // Definition of `arbitrate` goes here
    io.out <> io.in.reduceTree(
      (a, b) => arbitrate(a, b))
}
```

Listing 5: Arbiter built from a tree of 2-to-1 arbiters, i.e., the `arbitrate` function.

to the Scala standard library. Another solution would be to add functions to the standard Scala collection using implicit classes. This is similar to how Scala itself adds methods to the original `String` class from Java.

B. Arbitration Example

We can now combine the features used in the previous section to construct a more advanced circuit easily. For example, we consider a circuit that arbitrates between n components. The arbiter should use a ready/valid interface. We can construct the corresponding hardware as a tree of 2-to-1 arbiters. The output of the arbitration circuit will be a single ready/valid interface. We define an `arbitrate` function that will be used by the `reduceTree` function to arbitrate between two requests. The code is shown in Listing 5.

A simple approach to the `arbitrate` function would be to build a simple priority-based arbitration, i.e., always select the first input. The resulting circuit is a combinational circuit. We must add registers for those signals (and the data) to avoid a combinational path between the ready and valid signals. The listing of this initial circuit can be found in [20, Section 10.6.2].

This arbitration circuit prioritizes the first input. To create a fair arbiter, we must remember which of the two inputs won the arbitration the last time. We do so by introducing a state register storing the previous choice. This mechanism can be viewed as a basic form of round-robin arbitration, where the arbiter alternates between the inputs to ensure fairness over time. Assuming that the 2-to-1 arbiter is fair, we assume this also results in a fair arbitration scheme on a balanced tree.

The resulting fair 2-to-1 arbitration circuit is shown in Listing 6. We use two registers: (1) `regData` storing the data to be arbitrated and (2) `regState` storing the current state of the arbiter. When no data is stored in the arbiter to be forwarded, it constantly switches between the two idle states `idleA` and `idleB`, indicating that it will accept data from the corresponding input port. When data has been accepted, the arbiter switches to the corresponding full state (`hasA` or `hasB`). When the data has successfully been delivered to the consumer, the arbiter continues with the idle state of the other input.

As we have only a single data register, the arbiter can only accept and store data from one input at a time. A second data register is required to accept two inputs in the same clock cycle, i.e., when both inputs are ready.

```
object State extends ChiselEnum {
  val idleA, idleB, hasA, hasB = Value
}

def arbitrate(a: DecoupledIO[T],
              b: DecoupledIO[T]) = {
  import State._
  val regData  = Reg(gen)
  val regState = RegInit(idleA)
  a.ready := regState === idleA
  b.ready := regState === idleB

  switch(regState) {
    is (idleA) {
      when (a.valid) {
        regData  := a.bits
        regState := hasA
      }.otherwise {
        regState := idleB
      }
    }
    is (idleB) {
      when (b.valid) {
        regData  := b.bits
        regState := hasB
      }.otherwise {
        regState := idleA
      }
    }
    is (hasA) {
      when (out.ready) {
        regState := idleB
      }
    }
    is (hasB) {
      when (out.ready) {
        regState := idleA
      }
    }
  }

  val out = Wire(new DecoupledIO(gen))
  out.valid :=
    (regState === hasA || regState === hasB)
  out.bits := regData
  out
}
```

Listing 6: A fair 2-to-1 arbitration function for the `Arbiter` class.

VII. Applications

In this section, we provide examples of how to use object-oriented and functional programming from Scala to help build applications.

A. Mealy Machine Generator

The model often used to describe sequential hardware is that of a *Mealy machine*. A mealy machine holds the system's current state and further consists of two functions, `outFun` and `stateFun`, that, given input to the hardware system, compute

```
class Mealy[S <: Data, I <: Data, O <: Data](
  initState: S,
  genIn: I,
  genOut: O,
  stateFun: (S, I) => S,
  outFun: (S, I) => O
) extends Module {
  val io = IO(new Bundle {
    val in = Input(genIn)
    val out = Output(genOut)
  })

  val state = RegInit(initState)
  state := stateFun(state, io.in)
  io.out := outFun(state, io.in)
}
```

Listing 7: A Mealy machine generator.

the system's output and new state, respectively. For this, they also consider the system's current state. Figure 2 depicts this general and abstract view of Mealy machines.

Chisel allows the definition of a generic generator for Mealy machines that is parameterized by the internal state (type `S`), the inputs (type `I`), and its output (type `O`). It needs to be provided with the two functions `outFun` and `stateFun`. The corresponding code (see Listing 7) is concise, fitting in less than 20 lines, of which only three lines describe the actual operation; the other lines are boilerplate code. As the implementation is trivial, it is easy to understand. It should be noted, though, that the parameters `genIn` and `genOut` do not contain an actual value used by the machine. They are used to specify what types should be used by the Mealy machine. The implementation uses functions as first-class citizens and injects them into the polymorphic `Mealy` module.

For example, we present a Mealy machine that detects n consecutive 1's in an input stream. The visualization using n nodes is shown in Figure 3. The corresponding Chisel code is straightforward: basically, only the two functions `outFun` and `stateFun` have to be specified. The corresponding functions in Listing 8 are `cntOutput` and `cntState`, respectively. The state variable `initState` is filled with the value 0 whose required bit-width is computed from the number of 1's to count. Another Scala feature is used: multiple parameter lists (also known as *currying*). Supplying values for the first parameter list only yields a function expecting the rest of the parameters, e.g., `cntOutput(4)` is a function that expects a `UInt` (the internal state) and a `Bool` (the input) and then decides whether to emit a 1 (at least 4 consecutive ones have been detected) or not. This new function is then used by the polymorphic `Mealy` module. Another technical detail is that the `cntMealy` function defines the `initState` variable within its scope that is then used by the `cntOutput` and `cntState` functions and, hence, by the Mealy machine. The last lines of the `cntMealy` function returns an instantiated `Mealy` module.

B. Software-defined IO

In previous work, we used Chisel to transpile high-level descriptions of generic coarse-grained reconfigurable array

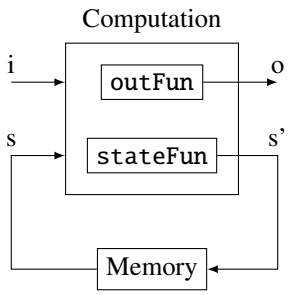


Fig. 2: Abstract view of a Mealy machine entirely determined by the functions `outFun` and `stateFun`.

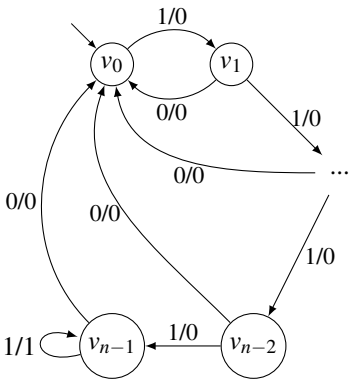


Fig. 3: Abstract visualization of a Mealy machine detecting n consecutive 1's in the input.

(CGRA) architectures into Verilog code [22]. We later automated using these architectures with approximate computing features [23]. The source language is based on XML to limit the implementation effort needed to describe the highly regular architectures.

Modules described in the source language do not necessarily have hierarchical depths or a priori known ports, as certain basic elements, *primitives*, needed for mapping application kernels to the architectures are converted into ports in hardware. The hardware generation flow must, therefore, be able to build arbitrarily deep module hierarchies but also support interfaces specified at elaboration time, as higher-level

```
def cntMealy(n: Int) = {
  val initState = 0.U(log2Ceil(n + 1).W)

  def cntOutput(n: Int)(s: UInt, i: Bool): Bool = {
    Mux(i, s === n.U, false.B)
  }

  def cntState(n: Int)(s: UInt, i: Bool): UInt = {
    Mux(i, Mux(s < n.U, s + 1.U, s), 0.U)
  }

  new Mealy(initState, Bool(), Bool(), cntState(n),
   ↪ cntOutput(n))
}
```

Listing 8: Code generating a Mealy machine for detecting n consecutive 1's.

```
class PortRecord[T <: Data](
  elts: Seq[(String, T)]
) extends Record {
  elts.map(_._2).foreach(requireIsChiselType(_))
  val elements =
   ↪  immutable.ListMap(elts.map(_.cloneType):_*)
  def apply(elt: String): T = elements(elt)
}

// The below is encapsulated in a Module
val _ips = mutable.HashMap.empty[String, UInt]
val _ops = mutable.HashMap.empty[String, UInt]

lazy val io = IO(new Bundle {
  val ins  = Input(new PortRecord[UInt](
    _ips.toSeq.sortBy(_._1)))
  val outs = Output(new PortRecord[UInt](
    _ops.toSeq.sortBy(_._1)))
})
```

Listing 9: An implementation of software-defined IO.

modules may inherit and pass through ports from sub-modules. The flow combines functional, object-oriented, and imperative programming features to achieve this. Composite modules are generated as follows:

1) Name the module using Chisel's `desiredName` API.
2) Recursively create all sub-modules and add them to the current module.
3) Create all ports and add them to the current module.
4) Connect current module and sub-module ports.

The flow maintains the elaboration-time IO in two `mutable.HashMap[String, UInt]` fields that are converted into Chisel `Record`s, as shown in Listing 9. For proper elaboration, this conversion must not be eager. Therefore, the fields marked `lazy`, meaning their elaboration is delayed until just before they are needed, and the maps are fully specified before their reference. However, the possible inheritance of ports from sub-modules conflicts with the execution order of inherited constructors in Scala. This conflict is circumvented by wrapping steps 2 through 4 above into a `build` method called at the beginning of the parent class' constructor.

To guarantee elaboration, the `io` field is referenced right after the call to `build`. Subsequently, all ports may be accessed by their names using the `PortRecord` class' `apply` method. To the best of the authors' knowledge, this strategy was not applied outside of RocketChip's *diplomacy* package before [24]. The presented generation of IO types has been used in a framework for the connection of dynamic generated CGRAs [25].

C. Hardware-Software Codesign

Heterogeneous platforms, such as System-on-chip (SoC) FPGAs that combine multiple CPU cores with an FPGA fabric on the same die, are becoming increasingly popular. However, designing the composition of custom accelerators and software components remains challenging for several reasons. The SW components typically interact with the accelerators via the shared memory hierarchy or by instantiating control and status registers (CSRs) in the FPGA, which can be accessed through AXI ports. Implementing communication between accelerators and hardware is often tedious and vendor-specific. In Chisel, polymorphism can abstract multiple FPGA platforms behind the same interface. Moreover, software drivers needed to interact with the different ports of the accelerator can be code-generated at compile time using convenient string formatting in Scala.

This is used in Chisel projects such as FPGA shells[4] and fpga-tidbits [26]. FPGA shells are part of the Rocket Chip project and are used to easily deploy Rocket cores onto different FPGA boards from multiple vendors. They provide a uniform interface to the resources of the different boards, both on and off-chip.

While FPGA shells are mostly used for CPU emulations, fpga-tidbits target building and deploying embedded systems using SoC FPGAs. fpga-tidbits is a vendor-agnostic Chisel library for rapid prototyping of HW/SW codesigns. It provides a generic abstraction of an FPGA platform on which portable accelerators can be built. It also includes a component library of memories, queues, and data movers. Finally, it includes a co-simulation environment based on Verilator, where FPGA accelerators can be simulated with the software interacting with them.

Listing 10 shows a simple accelerator `GrayScale` designed using fpga-tidbits. The top-level IO between the accelerator and the host CPU is defined in `GrayScaleIO`. The CPU writes the input signals and includes a signal to start the accelerator, the address and size of an RGB image, and the destination for the grayscale image. It also includes a finished signal driven by the accelerator to notify the CPU that the image has been written to shared memory.

All IO signals defined in `GrayScaleIO` are compiled by fpga-tidbits into CSRs, which can be written or read from software. Depending on the target FPGA, these registers might be read and written through an AXI port between the CPU and the FPGA. With fpga-tidbits, such low-level details are not part of the accelerator logic.

The pixels of the image are read from shared memory, and fpga-tidbits provides generic shared memory ports, which will be compiled to concrete implementations depending on the FPGA platform. A component library is included, which includes a `StreamReader` and `StreamWriter`, which are parameterizable DMA modules. They produce memory requests and expose a `Decoupled` interface to the accelerator.

The gray filter is implemented in the `GrayScaleFilter` module, which is not shown for brevity.

When compiling `GrayScale` from Listing 10, fpga-tidbits will generate platform-independent C++ drivers for interacting with the accelerator. This enables hardware-software to codesign without committing to a vendor or a platform. The C++ drivers expose an interface to read and write the CSRs, defined in the top-level IO, and declare, read, and write to shared memory between the CPU and the FPGA.

Eventually, the platform-independent C++ drivers must be paired with a platform-specific driver that reads and writes to CSRs and shared memory.

[4]https://github.com/chipsalliance/rocket-chip-fpga-shells

```
class GrayScaleIO(p: PlatformWrapperParams) extends
 ↪  GenericAcceleratorIF(1, p) {
  val start = Input(Bool())
  val finished = Output(Bool())
  val baseAddr = Input(UInt(64.W))
  val byteCount = Input(UInt(32.W))
  val resBaseAddr = Input(UInt(64.W))
  val resByteCount = Input(UInt(32.W))
}

class GrayScale(p: PlatformWrapperParams) extends
 ↪  GenericAccelerator(p) {
  val numMemPorts = 1
  val io = IO(new ExampleGrayScaleIO(p))
  io.signature := makeDefaultSignature()

  val rdP = new StreamReaderParams(
    streamWidth = 24, fifoElems = 8, mem =
     ↪  p.toMemReqParams(),
    maxBeats = 1, chanID = 0, disableThrottle = true
  )

  val wrP = new StreamWriterParams(
    streamWidth = 8, mem=p.toMemReqParams(), chanID =
     ↪  0, maxBeats = 1
  )

  val reader = Module(new StreamReader(rdP)).io
  val writer = Module(new StreamWriter(wrP)).io

  reader.start := io.start
  reader.baseAddr := io.baseAddr
  reader.byteCount := io.byteCount
  reader.doInit := false.B
  reader.initCount := 8.U

  writer.start := io.start
  writer.baseAddr := io.resBaseAddr
  writer.byteCount := io.resByteCount

  io.finished := writer.finished

  reader.req <> io.memPort(0).memRdReq
  io.memPort(0).memRdRsp <> reader.rsp
  writer.req <> io.memPort(0).memWrReq
  writer.wdat <> io.memPort(0).memWrDat
  writer.rsp <> io.memPort(0).memWrRsp

  val grayFilter = Module(new GrayScaleFilter)
  grayFilter.rgbIn.valid := reader.out.valid
  grayFilter.rgbIn.bits :=
   ↪  reader.out.bits.asTypeOf(new Colour)
  reader.out.ready := grayFilter.rgbIn.ready

  grayFilter.grayOut <> writer.in
}
```

Listing 10: A simple gray scale filter accelerator implemented with fpga-tidbits

VIII. Testing and Verification

Although this paper focuses on writing hardware generators, testing and verifying such generators is equally important. Therefore, we briefly describe how Scala supports testing. Writing test benches in Scala can also be more efficient than writing them in SystemVerilog or cocotb [27].

Chisel designs can be tested with ChiselTest [28].[5] ChiselTest is a Scala library driving the Chisel circuit, either with an internal simulation engine or the generated Verilog via Verilator (or other backends). Switching between simulation backends is transparent to the testing code.

ChiselTest provides basic support for setting input ports, advancing the clock, and reading output ports. Compared to solutions like cocotb, ChiselTest is type-safe with Chisel designs under test. The main feature of ChiselTest is that it is also embedded in Scala as a Scala library. Similar to generators, we have all the features of Scala and many Java libraries available to write tests.

As a simple example, assume you want to test a processor and need to preload the memory simulation with an .elf file containing a program to execute. Without Scala, one would probably use an `objdump` tool as part of the compiler and generate binary files that can be read in Verilog. As all Java libraries are available, we can use an ELF library in Chisel and read the .elf file directly in the testbench into the memory simulation.

One of the authors is interested in processor designs. A processor can easily be tested when co-simulating a golden model and the hardware description. For example, in the Wildcat project [29], [30], we initially wrote a simulator of the RISC-V instruction set in Scala. Then, we implemented different pipelined versions in Chisel. The simulator is a golden model, and we perform co-simulation in Scala of standard RISC-V test cases. In those tests, we call out from Scala tests to the compiler to generate the .elf file and then use a Java library to read those .elf files into the simulation. We can even leverage the Scala testing framework `ScalaTest`[6] to run all hardware tests.

ChiselVerify [31] is a library for verification that extends the capabilities of ChiselTest. It adds, besides other functions, functional coverage to ChiselTest. Functional coverage is a metric used in hardware verification to ensure that all design aspects have been exercised during testing. We can use or extend ChiselVerify to test designs in Chisel and other designs in Verilog by wrapping them into a Chisel black box. This feature enables advanced Scala features to be used to write tests and verification code for hardware designs in SystemVerilog.

As Scala executes on the Java virtual machine and is compatible with Java, we can use many open-source libraries for hardware generators and testing code. With the `sbt` build tool, a library can be referenced with a single line entry in `build.sbt` and automatically downloaded from the Maven

[5] We are aware that the newer versions of Chisel, starting with version 7, will probably not be supported by ChiselTest and a new simulation engine is in development. However, until that development has stabilized, we will continue to use ChiselTest. We also expect that ChiselSim will provide a minimal compatibility layer for ChiselTest for the transition phase.

[6] https://www.scalatest.org/

server. We can use Java libraries for tasks like logging, data processing, reference implementations, and more, which complement the Chisel test environment.

Another example of one of the authors is testing, verification, and performance analysis of a network-on-chip, written in Chisel [32]. Besides writing functional tests we also implemented traffic generators in Scala. We explored different buffer schemes of the network interface with traffic generators written in Scala. We used Scala queues that can grow to simulate an "ideal" buffer scheme. This experiment sets the baseline for the following experiments. Then, we substituted those queues with concrete hardware queues written in Chisel. We can use the same language and execution environment for testing, simulation, and hardware design, making the swapping in and out of software/simulation implementations with hardware implementations seamless and efficient.

One questions might be: If we verify the Chisel code at Chisel level do we trust that the generated Verilog code is correct? However, with ChiselTest we can run the tests on the generated Verilog code with Verilator.[7] Furthermore, we can even reuse the tests with Verilog code generated by synthesis with a Chisel black box.

Debugging hardware generators can become a non-trivial task. One key challenge is that bugs may occur either in the hardware description itself (Chisel part) or in the generator logic that constructs the hardware (Scala part), making it necessary to debug across both the Scala and Chisel domains. Fortunately, in most cases, one can start by debugging a single instance, which allows them to focus on the hardware description and temporarily ignore the generator logic. Once the bug in the instance is identified, the implications and fixes needed for the generator can be determined. However, as a bonus we have the full power of Scala to write these testing procedures.

Writing a design in Chisel but debugging it from its generated Verilog is a minor hindrance rather than a major issue. One can generally focus their time rereading the Chisel rather than the Verilog because the Chisel to Verilog flow is quite reliable. The only time the existence of the Verilog is apparent to most designers is when examining the waveform from the simulation because some of the signal names are tweaked and there are additional signals they did not declare. Those additional signals are intermediate values used by the generated Verilog, and they can be typically ignored because debugging their endpoints (user-defined signals) is sufficient. The naming of signals makes it readily apparent which originated from the user Chisel versus those that are intermediate values. Newer versions of Chisel and especially firtool (within CIRCT) have placed an emphasis on making the generated Verilog and its signal names more readable. Like any language, writing the language idiomatically can help reduce errors and improve readability, and there are Chisel style guides [33], [34].

Here, we provided just small examples of how a modern programming language can improve test writing and explore design alternatives. We look forward to further improving testing in Scala/Java with a project inspired by UVM.

[7]The testing of Chisel code with the Chisel level simulation is deprecated by the newer Chisel versions.

IX. Teaching Hardware Generators

One of the authors has been teaching agile hardware design since 2021 [35], and one author has planned a course on agile hardware design that will begin in the fall of 2025. Agile hardware design is, to a large extent, software programs implementing hardware generators. Teaching agile hardware design is challenging as the prerequisites are quite divergent: hardware design, software engineering, object-oriented programming, and functional programming. At DTU, where we plan to start this type of course in fall 2025 for a Bachelor's in Computer Engineering, which combines electrical engineering with classic computer science. However, the students do not know functional programming.

We have created a course to teach agile hardware design methodologies using hardware generators written in Chisel [35], and we have released all of the materials open-source online.[8] Developing hardware generators calls for a combination of conceptual understanding and technical/practical expertise. On the design side, designers must consider the module's functionality, interfaces, and parameterization. Technically, they must be proficient in both Chisel and Scala to implement it. The most novel aspect is they must understand the hardware construction process during the elaboration stage. We briefly cover the methodologies we aim to teach and the course itself.

Our course is designed around the agile methodology of directed incremental improvement. An emphasis is placed on doing whatever simplifications are necessary to "close the loop" as soon as possible, i.e., get a design running through the tool flow (the loop). Once the loop is established, it does not take too much effort to tweak the design and rerun the flow. The loop through the tool flow needs to be largely automated because it will be run many times. This agile approach contrasts with a conventional waterfall approach, which fully completes each step (e.g., design specification, design implementation, verification, physical design) before moving on to the next step. By using incremental improvement with an automated flow, there is always a design and awareness of its current functionality, performance, power, and area. The design's functionality can be extended, or its physical design can be optimized until the designer is satisfied. With this incremental approach, their efforts can be targeted exactly where needed. With a waterfall approach, there is a decent amount of uncertainty, as the need for certain optimizations may not be obvious at the initial design time since the design has not yet gone through the tool flow.

The course's resulting practical methodology is built around closing the loop early. We want students to have designs that can run through the tool flow early on. We encourage them first to implement a functional model of the intended module in Scala. With that, they can use it in co-simulation to make unit tests to test the Chisel generator as it is being developed.

[8]https://classes.soe.ucsc.edu/cse228a/Winter24/

To teach this incremental design approach, the assignments frequently task students with revising prior designs. For example, one problem from the first homework assignment tasks students with building a generator to make a module that evaluates a polynomial with exactly three terms with fixed bit widths. The following assignment has them revise the generator to take an arbitrarily sized list of coefficients, and the corresponding module will be generated to accommodate the corresponding number of terms. The theme of incremental improvement is covered throughout the course. Later on, the course focuses on design techniques to identify the simplest starting point and a roadmap for additional functionality and optimizations. For example, one lecture is a case study on the design of a FIFO. The first version is a single entry and, after five revisions, reaches a parameterized generator whose output uses a circular buffer.

The emphasis on incremental improvement and revision has benefits beyond the course and hardware design and is thus an important meta-learning objective. Students may have encountered polished code from mature open-source projects, but they have typically only written what amounts to a first draft for typical assignments. By seeing the process of incremental improvement and revising a code base, they can better appreciate how to make these larger, more sophisticated projects.

Teaching students to develop generators also entails understanding the hardware construction and elaboration process. Chisel can be used very straightforwardly, and its functionality and expressiveness are equivalent to structural Verilog. Still, it brings stronger types' benefits, as exemplified in DinoCPU [36], [37]. A hardware generator is more flexible and uses Scala to stitch together pieces of hardware (in Chisel), so the run time behavior of constructing hardware before elaboration must be appreciated. This is a tricky concept, so we teach it in various ways. In the lecture, we have examples where we go through the code line-by-line, and simultaneously, with diagrams, we show how various Scala variables exist in memory and how each line mutates that in-memory state. In the labs and homework assignments, we intermix problems that are pure Scala with those that are Chisel. Regarding course content, we continuously switch between covering Scala, Chisel, and agile methodologies while describing increasingly powerful generators. Intermixing the instruction of these various topics is yet another example of the incremental approach.

X. Related Work

A. Generator Languages and Tools

Hardware generators are a rapidly developing field in which much research and engineering is being carried out. New languages and tools are frequently proposed in the literature and made publicly available. Like Chisel, some languages aim to be all-purpose HDLs with a level of abstraction. Examples of this are myHDL [38],[9] using Python as the host language, and SpinalHDL,[10] which is very similar to Chisel; not only in using the same host language Scala. Some languages target more specific domains. For example, the DFiant HDL [39][11] also uses Scala as the host language and aims to join dataflow, register-transfer level, and event-driven development domains. For this, it entirely abstracts away clocks and registers.

Another example, using Rust as the host language, is the Spade language [40].[12] Spade uses functions to represent modules. Chisel, in contrast, supports both styles of describing a module: as a `Module` with input and output ports or as a function with inputs and outputs, as we use it in this paper. Similar to Chisel, Spade supports generics. Spade targets pipelined designs and forces the designer to instantiate registers explicitly.

A language raising the level of abstraction for describing hardware even further is Clash [9].[13] Clash is based on the functional language Haskell. Within Clash, a combinational circuit is considered a function, transforming an input x from a given domain X to a value y in the codomain Y, i.e., circuit: $X \rightarrow Y$. As Haskell is a general-purpose programming language, it supports constructs that are unavailable in hardware. Consequently, Clash forbids the use of infinite or recursive data structures. The designer, nevertheless, can still make use of almost all the abstractions offered by Haskell, such as union types (e.g., `Either (Signed 8) (Unsigned 16)`) or the monadic programming style. The Clash compiler can automatically derive bit representations for all valid data types.

Sequential circuits do not immediately fit the function model and must be modeled differently. For this, Clash introduces a streaming type `Signal dom a` that is parameterized over the clock domain `dom` and the domain `a`. Each clock cycle produces a value of type `a`. Clash is restricted to synchronous circuits only. As Mealy machines are an often used abstraction to describe sequential circuits, Clash offers the `mealy` function that transforms a function of type `f :: s -> i -> (s,o)` that maps a state `s` and an input `i` into a tuple containing the new system state and the corresponding output into the corresponding hardware. This is similar to the approach discussed in Section VII-A but comes out of the box.

The Bluespec language pursues a different approach.[14] It differs from "traditional" HDLs because it uses a fundamentally different execution model. Instead of threads and processes, atomic actions serve as the main building blocks [41]. There are two language implementations, one integrated into SystemVerilog and one into Haskell. Both implementations feature construct `rule (predicate); action statements; endrule` (with minor syntactic differences between the implementations) that is the major way to describe arbitrarily complicated actions that occur when the `predicate` holds. These actions can very well be synchronous and sequential. Bluespec uses object-oriented features, such as customizable, shared interfaces and functions for hardware generation.

[9] https://github.com/myhdl/myhdl
[10] https://github.com/SpinalHDL/SpinalHDL
[11] https://dfianthdl.github.io/
[12] https://spade-lang.org/
[13] http://www.clash-lang.org
[14] https://github.com/B-Lang-org/bsc

The Kactus2 tool[15] aims to automate the integration of IP-XACT-defined hardware blocks. The idea is to promote and simplify the reuse of IP cores [42], [43]. IP-XACT is an XML-based standard for describing IP cores [18]. Kactus has a graphical user interface that allows you to easily define bus connections, mapping, and parameter configurations. It generates Verilog code and all the necessary interconnections between the IP blocks. The end-user only has to provide the concrete implementation of the used IP blocks. On top of that, Kactus2 comes with a verification step for the correctness of the address mapping. This ensures that the address space of all IP blocks is consistent.

The tool Genesis 2 [5] introduces the concept of *recipes* that allows hardware designers to write programs to construct building blocks of hardware. For this, one has to define a set of parameters and constraints via an XML file. Genesis 2 is written in Perl and emits Verilog based on generated and interconnected hardware blocks. It further generates verification components, directives for the physical implementation, and files that target the software stack (e.g., header files).

B. Applications

The Rocket processor project [44], known as "The Rocket Chip Generator," focuses on generating and testing complete SoCs. It includes Diplomacy and TileLink [24]. Diplomacy addresses parameter negotiation and hardware unit integration into a shared address space, ensuring endpoint compatibility and optimization based on mutual knowledge. TileLink, a standard for chip-scale shared-memory interconnects, leverages Diplomacy to adapt to various protocol requirements.

Similarly, Chipyard [45] enables developers to design complete SoCs with parameterized CPU cores, buses, caches, accelerators, and peripherals using Chisel. It employs lazy evaluation to defer hardware graph construction until all components are instantiated, akin to the approach described in Section VII-B.

The `reactor-chisel` library[16] targets reactor-oriented programming of SoC FPGAs [46]. The reactor design can be expressed by instantiating and connecting modules from the `reactor-chisel` library. At Chisel compile-time a hardware runtime is also synthesized which takes care of the communication between modules and enables them in a deterministic order. `reactor-chisel` leverages `fpga-tidbits` to also support reactor-oriented hardware-software codesign.

Chisel (or related tools) have been used in many other places to make designing and generating parameterized hardware easy. This includes RISC-V cores [47], accelerators for quantized neural networks [48], [49], for spiking neural networks [50], or fast Fourier transforms [51], and even very generic networks-on-chip designs for SoCs [52].

XI. Conclusion

Scala provides a tasteful mix of object-oriented and functional programming in a single language. Chisel uses those features to implement an embedded domain-specific language for hardware construction. As users of Chisel, we can extend this combination of Scala and Chisel to write hardware generators. When Chisel is executed, it is a Scala program that runs to spill out a hardware description that tools can use further downstream to test and generate Verilog code. This execution of the Scala program is where we add the generator functionality. In this paper, we showed that with a few lines of functional code, one can write powerful abstractions of digital circuits. This paper showed a handful of examples of how hardware generators can be written. We expect that future development and research will extend these ideas to provide powerful abstractions for different domains.

[15]https://github.com/kactus2/kactus2dev

[16]https://github.com/erlingrj/reactor-chisel

Acknowledgments

Work partially supported with funding from European Union's Digital Europe Programme (DIGITAL) under the European Health and Digital Executive Agency (HaDEA) grant agreement No. 101123086 (Edu4Chip).

References

[1] *IEEE Standard VHDL Language Reference Manual*, IEEE Computer Society, 1987.

[2] *IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language*, IEEE Computer Society, 1995.

[3] *IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language*, IEEE Computer Society, 2005.

[4] 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.

[5] O. Shacham, *Chip Multiprocessor Generator: Automatic Generation of Custom and Heterogeneous Compute Platforms*. Stanford University, 2011.

[6] S. A. Murtza, O. Hasan, and K. Saghar, "Vertgen: An Automatic Verilog Testbench Generator for Generic Circuits," in *2016 International Conference on Emerging Technologies (ICET)*. IEEE, 2016, pp. 1–5.

[7] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A Unified Framework for CGRA Modelling and Exploration," in *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2017, pp. 184–189.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing Hardware in a Scala Embedded Language," in *The 49th Annual Design Automation Conference (DAC 2012)*. San Francisco, CA, USA: ACM, June 2012, pp. 1216–1225.

[9] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Cλash: Structural Descriptions of Synchronous Hardware using Haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 2010, pp. 714–721.

[10] M. Schoeberl, H. J. Damsgaard, L. Pezzarossa, O. Keszocze, and E. R. Jellum, "Hardware Generators with Chisel," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, 2024, pp. 168–175.

[11] 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.

[12] S. Eldridge, P. Barua, A. Chapyzhenka, A. Izraelevitz, J. Koenig, C. Lattner, A. Lenharth, G. Leontiev, F. Schuiki, R. Sunder *et al.*, "MLIR as hardware compiler infrastructure," in *Workshop on Open-Source EDA Technology (WOSET)*, vol. 3, 2021.

[13] "CIRCT: circuit ir compilers and tools," https://circt.llvm.org, 2021.

[14] *IEEE Standard VHDL Language Reference Manual*, IEEE Computer Society, 2009.

[15] *IEEE Standard for Verilog Hardware Description Language*, IEEE Computer Society, 2006.

[16] M. Schoeberl, "Lipsi: Probably the Smallest Processor in the World," in *Architecture of Computing Systems – ARCS 2018*. Springer International Publishing, 2018, pp. 18–30.

[17] M. Schoeberl and M. Petersen, "Leros: The Return of the Accumulator Machine," in *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, M. Schoeberl, T. Pionteck, S. Uhrig, J. Brehm, and C. Hochberger, Eds. Springer, May 2019, pp. 115–127.

[18] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEEE Std 1685-2022 (Revision of IEEE Std 1685-2014)*, pp. 1–750, 2023.

[19] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable Multi-Core Architecture for Embedded Systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[20] M. Schoeberl, *Digital Design with Chisel*. Kindle Direct Publishing, 2019, available at https://github.com/schoeberl/chisel-book.

[21] *IEEE Standard VHDL Language Reference Manual*, IEEE Computer Society, 2019.

[22] H. J. Damsgaard, A. Ometov, and J. Nurmi, "Generating CGRA Processing Element Hardware with CGRAgen," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 1–7.

[23] H. J. Damsgaard, "Reconfigurable Approximating Accelerators for Edge Computing," Ph.D. dissertation, Tampere University, 2024.

[24] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic Design Patterns: A TileLink Case Study," in *1st Workshop on Computer Architecture Research with RISC-V*, vol. 23, 2017.

[25] H. J. Damsgaard, A. Ometov, and J. Nurmi, "Adaptive approximate computing with cgragen," *IEEE Design & Test*, 2025.

[26] E. R. Jellum, Y. Umuruglu, M. Orlandic, and M. Schoeberl, "fpga-tidbits: Rapid Prototyping of FPGA Accelerators in Chisel," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 153–160.

[27] B. J. Rosser, "Cocotb: a python-based digital logic verification framework," in *Micro-electronics Section seminar. CERN, Geneva, Switzerland*, 2018.

[28] R. Lin. ChiselTest. https://github.com/ucb-bar/chisel-testers2.

[29] M. Schoeberl, "The Educational RISC-V Microprocessor Wildcat," in *Proceedings of the Sixth Workshop on Open-Source EDA Technology (WOSET)*, 2024.

[30] ——, "Wildcat: Educational RISC-V microprocessors," in *Architecture of Computing Systems – ARCS 2025*, 2025.

[31] A. Dobis, K. Laeufer, H. J. Damsgaard, T. Petersen, K. J. H. Rasmussen, E. Tolotto, S. T. Andersen, R. Lin, and M. Schoeberl, "Verification of Chisel Hardware Designs with ChiselVerify," *Microprocessors and Microsystems*, vol. 96, p. 104737, 2023.

[32] M. Schoeberl, "Exploration of Network Interface Architectures for a Real-Time Network-on-Chip," in *Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*. United States: IEEE, 2024, 2024 IEEE 27th International Symposium on Real-Time Distributed Computing, ISORC ; Conference date: 22-05-2024 Through 25-05-2024.

[33] "Chisel developers style guide," https://www.chisel-lang.org/docs/developers/style, 2025.

[34] C. Celio, "Chisel style guide," https://github.com/ccelio/chisel-style-guide, 2016.

[35] S. Beamer, "Teaching Agile Hardware Design with Chisel," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, 2024, pp. 161–167.

[36] "Davis In-Order (DINO) CPU," https://github.com/jlpteaching/dinocpu, 2019.

[37] J. Lowe-Power and C. Nitta, "The Davis In-Order (DINO) CPU: A Teaching-Focused RISC-V CPU Design," in *Proceedings of the Workshop on Computer Architecture Education*, 2019, pp. 1–8.

[38] K. Jaic and M. C. Smith, "Enhancing Hardware Design Flows with myHDL," in *2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 28–31.

[39] O. Port and Y. Etsion, "Registerless Hardware Description," 2021.

[40] F. Skarman and O. Gustafsson, "Spade: An HDL Inspired by Modern Software Languages," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 454–455.

[41] R. S. Nikhil and Arvind, "What is Bluespec?" *ACM SIGDA Newsletter*, vol. 39, no. 1, pp. 1–1, 2009.

[42] A. Kamppi, E. Pekkarinen, J. Virtanen, J.-M. Määttä, J. Järvinen, L. Matilainen, M. Teuho, and T. D. Hämäläinen, "Kactus2: A Graphical EDA Tool Built on the IP-XACT Standard," *Journal of Open Source Software*, vol. 2, no. 13, p. 151, 2017.

[43] E. Pekkarinen and T. D. Hämäläinen, "Modeling RISC-V Processor in IP-XACT," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 140–147.

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

[45] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[46] E. R. Jellum, M. Schoeberl, E. A. Lee, and M. Orlandic, "Codesign of Reactor-Oriented Hardware and Software for Cyber-Physical Systems," *ACM Trans. Reconfigurable Technol. Syst.*, jun 2024, just Accepted.

[47] 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.

[48] J. Vreča and A. Biasizzo, "Towards Deploying Highly Quantized Neural Networks on FPGA Using Chisel," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 161–167.

[49] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 769–774.

[50] P. Plagwitz, F. Hannig, J. Teich, and O. Keszocze, "DSL-based SNN Accelerator Design using Chisel," in *Euromicro Conference on Digital System Design*, Paris, France, Aug. 2024.

[51] V. M. Milovanović and M. L. Petrović, "A Highly Parametrizable Chisel HCL Generator of Single-Path Delay Feedback FFT Processors," in *2019 IEEE 31st International Conference on Microelectronics (MIEL)*, 2019, pp. 247–250.

[52] J. Zhao, A. Agrawal, B. Nikolic, and K. Asanović, "Constellation: An Open-Source SoC-Capable NoC Generator," in *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, 2022, pp. 1–7.

Martin Schoeberl received his PhD from the Vienna University of Technology in 2005. From 2005 to 2010, he was an assistant professor at the Institute of Computer Engineering. He is now a professor of real-time computer architecture at the Technical University of Denmark. His research interest is in hard real-time systems, time-predictable computer architecture, and real-time Java. Martin Schoeberl has been involved in several national and international research projects: JEOPARD, CJ4ES, T-CREST, RTEMP, the TACLe and CERCIRAS COST actions, and PREDICT. He has been the technical lead of the EC-funded project T-CREST. He has more than 200 publications in peer-reviewed journals, conferences, and books.

Hans Jakob Damsgaard is a Doctoral Researcher MSCA at Tampere University, Finland. He received his M.Sc. (Eng.) degree in computer science and engineering from the Technical University of Denmark, Lyngby, Denmark. His research interests include approximate computing and reconfigurable hardware platforms.

Luca Pezzarossa is an Associate Professor in Computer Engineering at the Technical University of Denmark. He received his Ph.D. degree from the same university in 2018. His research topics include reconfigurable systems, real-time cyber-physical/embedded systems, and embedded AI for audio applications.

Oliver Keszocze received his PhD ein 2017 from the University of Bremen, Germany. From 2018 to 2024 he was an assistant professor at the Friedrich-Alexander-University (FAU), Erlangen, Germany. Since 2024 he is an associate professor at the Embedded Systems Engineering section of Denmark Technical University (DTU), Lyngby, Denmark. His research interests include logic synthesis, computer-aided design, approximate computing and modern hardware description languages.

Erling Rennemo Jellum is a Postdoctoral researcher at the industrial Cyber-Physical Systems group at the University of California, Berkeley. He received his PhD from the Norwegian University of Science and Technology in 2024. His research is focused on embedded systems design.

Scott Beamer as an assistant professor of Computer Science and Engineering at the University of California, Santa Cruz. He received his Ph.D. degree from the University of California, Berkeley. His resesarch interests include computer architecture, agile and open-source hardware design, and high-performance graph processing.