Scala Based FPGA Design Flow

ABSTRACT

With the rapid growth of data scale, data analysis applications start to meet the performance bottleneck, and thus requiring the aid of hardware acceleration. At the same time, Field Programmable Gate Arrays (FPGAs) have gained momentum in the past decade. However, the efficiency of development for acceleration system based on FPGAs is severely constrained by the traditional languages and tools, due to their deficiency in expressibility, extendability, limited libraries and semantic gap between software and hardware design. This paper proposes a new open-source DSL based hardware design framework called VeriScala [1] that supports highly abstracted object-oriented hardware defining, programmatical testing, and interactive on-chip debugging. By adopting DSL embedded in Scala, we introduce modern software developing concepts into hardware designing including object-oriented programming, parameterized types, type safety, test automation, etc. VeriScala enables designers to describe their hardware designs in Scala, generate Verilog code automatically and interactively debug and test hardware design in real FPGA environment. Through the evaluation on real world applications and usability test, we show that VeriScala provides a practical approach for rapid prototyping of hardware acceleration systems.

Keywords

FPGA, DSL, VeriScala

Categories and Subject Descriptors

B.1.4 [**Microprogram Design Aids**]: Languages and compilers

1. INTRODUCTION

Due to the rapidly expanding data scale, emerging data analysis applications start to face performance bottleneck and require hardware acceleration in recent years, which follows the long-standing hardware/software co-design [8] tradition. Known for their high customizability and parallel nature, Field Programmable Gate Arrays (FPGAs) have gained momentum in a broad range of applications, *e.g.*, parallelization [21], distribution [15], deep learning [22]. However, developments on FPGAs are severely constrained by the languages and tools that are currently used [4]. For instance, Hardware Description Languages (HDLs) such as Verilog and VHDL, developed in the 1980's, are still dominant languages in describing hardware design [12]. Compared with modern software programming languages, HDLs have been long criticized for their limited functionalities such as code reuse, high-level abstractions, and maintainability [12].

Furthermore, the gap between HDLs and software programming languages is still enlarging as more features are constantly being developed in the area of modern programming languages. A number of programming languages have appeared (*e.g.* Clojure, Scala, etc.) with new features such as types/macros, traits, automatic test execution, and interoperability with other languages. Even though some of these features have been adopted by HDLs (*e.g.* recursive functions have been introduced to Verilog in 2001 [18]), it would still require significant work to catch up with developments in software programming languages.

Besides the outdated language features and difficulty in learning, what is more critical is that conventional FPGA design flow is independent from the developing flow of software engineers who are demanding the aid of hardware acceleration. Semantic gap between hardware descriptions and software applications prevents software engineers from rapid prototyping and easy maintenance. They heavily rely on C/C++ libraries that manipulate hardware modules in register level to build acceleration systems. In this approach, sophisticated knowledge on hardware is required, and application developers have no direct way to take part in hardware design in their working context.

In this paper, we propose VeriScala, a DSL based hardware design framework that supports highly abstracted object-oriented hardware defining, programmatical testing, and interactive on-chip debugging. By adopting DSL embedded in Scala, we introduce modern software developing concepts for traditional hardware design including object-oriented programming, parameterized types, type safety, test automation, etc. By providing an interactive on-chip debugging framework as a part of VeriScala, we provide convenience for rapid prototyping and propose an approach

[1]The URL is omitted due to double-blind policy.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

to bring software application and hardware modules into the same context.

The rest of this paper is organized as follows: Section 2 provides an overview of VeriScala, including the design goal, architecture, and workflow; Section 3 introduces the basic syntax and language features of VeriScala DSL; Section 4 shows the programmatic test and interactive debugging abilities provided by VeriScala; Section 5 provides evaluations on functionality and usability; Section 6 and Section 7 shows the related work and concludes the paper, respectively.

2. OVERVIEW

2.1 Design Goal

Designing hardware logic in an HDL, such as Verilog, is totally different from implementing some algorithms in software programming languages. This is mainly because traditional HDLs have very low-level abstraction about detailed logic elements, even in RTL design, and thus they require sophisticated hardware knowledge to write correct code. Our goal in this paper is to build a hardware design framework to make it easier to establish a hardware design by adopting achievements in software programming languages. Specifically, we would like to support the following four types of features:

- **Code reuse mechanisms**: Modern programming languages usually contain code reuse mechanisms such as *inheritance* (for object-oriented languages), *first-class functions* (for functional languages), and *parametric polymorphism*, etc. This can be helpful in both implementation and maintenance.
- **Abstractions**: Object-oriented languages consider objects as servers exporting procedural interfaces [5]. Under uniform interfaces, the implementations can be polymorphism and usually hidden from users.
- **Libraries**: A modern software programming language is typically equipped with a standard library and a lot of free third-party libraries for all sorts of purposes. Usually, there is also a package manager or a repository center for these languages to help developers easily access these libraries (*e.g.* Maven Repository Center).
- **Programmatical testing framework**: Testing framework makes it easy to organize test suites and test cases. Furthermore, the programs and tests share the same syntax, which means it is also possible to apply code reuse mechanisms, define or use abstract objects, and use other libraries in tests.

2.2 Architecture

To achieve the goal described above, we build the hardware design framework based on a Scala DSL. As shown in Figure 1, the framework consists of three modules used for designing, testing, and debugging in hardware design flow.

Design: A hardware description in VeriScala is defined as a `HDLClass`, a class which mixes in necessary traits, `BasicOps` and `Compiler` that give specific definition of the DSL, where Scala methods and hardware description code cooperate in one context.

Test: A testbench can be established by inheriting VeriScala hardware definition and mixing in traits that provide simulation utilities. Specific test code instantiates the testbench and runs simulation under the routines of self-defined or third party test libraries.

Debug: Controller code and configuration file are generated for debugging purpose. The controller is a wrapper of code to be debugged and will be in fact running on FPGA chip when debugging. And the configuration file is used by debugger application to specify the information of signals to be monitored. Detailed discussion of debugging framework will be presented in Section 4.2.

The core part of the framework is the VeriScala DSL consisting of *Design* and *Test* modules, which provides language features to define hardware logic and run functional tests. Verilog code and affiliated configuration file generated by VeriScala are used by hardware debugging module. By running the application *interactive debugger*, designers can interactively debug and test hardware design in real FPGA environment.

2.3 Workflow

In a basic VeriScala workflow, designers are firstly responsible for writing their hardware descriptions in high-level abstraction using VeriScala. After the design of the hardware description classes, compiling method can be called to automatically generate synthesizable Verilog code as well as affiliated files for on-chip debugging. Since a cycle-accurate simulator implemented in Scala is available, designers are strongly encouraged to first run the simulation in VeriScala, or even write some functional tests, to validate their design before continuing with the FPGA tool chain. For experienced hardware designers, after downloading generated controller modules into FPGA chip, running interactive debugger, and establishing the connection between the debugger and the debugging sever on Hardcore Processor System (HPS) system, they can debug their implementation in real hardware environment.

3. DESIGN

VeriScala is implemented as libraries of Scala without modifications on Scala compiler. We choose Scala because it is a modern multi-paradigm programming language on Java Virtual Machine (JVM), and it is statically typed and supports both object-oriented programming and functional programming. To use VeriScala, designers are only required to install the environment necessary for running Scala, and import the VeriScala libraries properly. Our design of VeriScala is greatly inspired by lightweight modular staging (LMS) [17]. LMS is a novel implementation of multi-stage programming [19] on Scala. By using the `trait`s provided by Scala, it enables users to extend and compose components with different features in flexible ways. As shown in Figure 1, the VeriScala library consists of two parts: class `HDLClass` for hardware description and class `SimulationSuite` for functional testing. In both of these two classes and their ancestors, only necessary data fields are defined, while specific methods and data structures to define concrete semantics and functions are separated in independent traits. For example:

BasicOps: This trait gives the semantics definition of VeriScala language, which includes basic types and operations, statements such as assignment statement and conditional statement, and the structure of VeriScala program that will be described in Section 3.1.

Compiler: This trait provides methods to translate VeriScala expressions into Verilog code.

In this way, VeriScala is designed and implemented highly

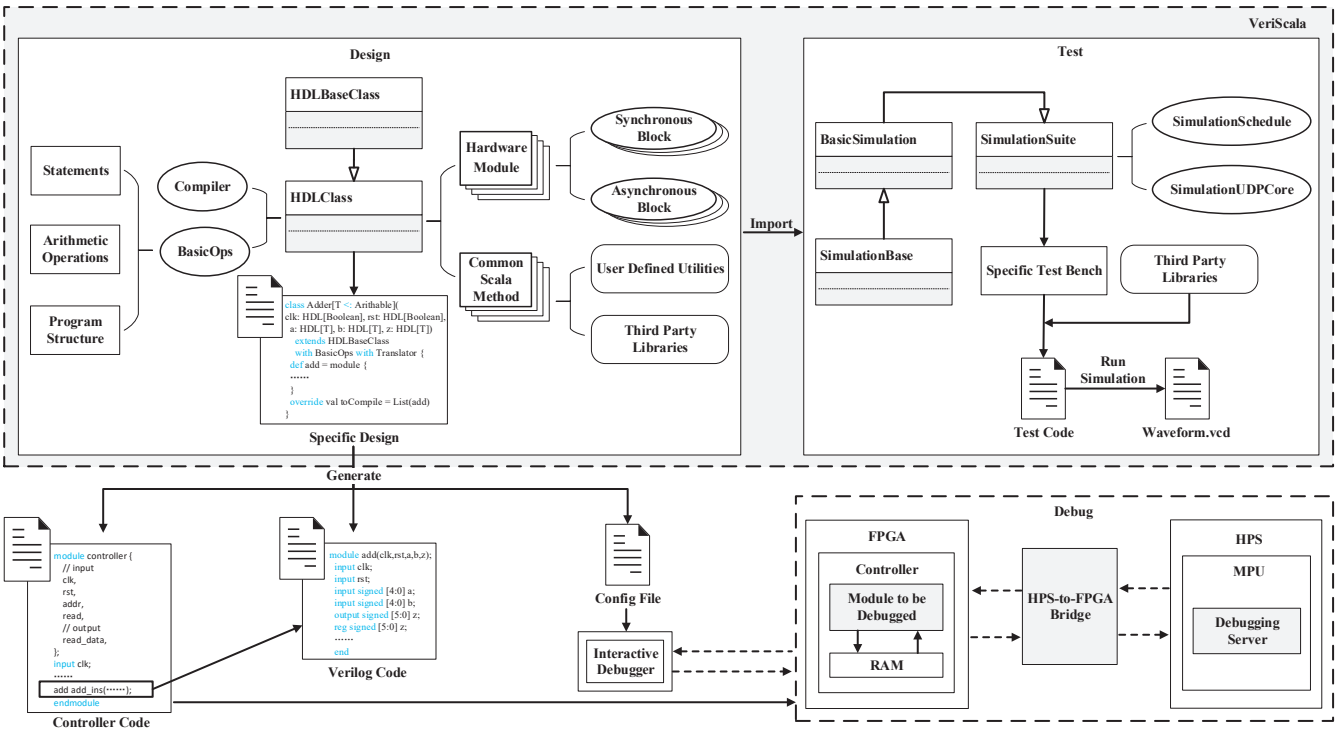

Figure 1: The architecture of VeriScala framework.

extendible. By mixing the basic classes with different traits, we can easily give the language different features. In fact, designers are encouraged to implement additional operations, syntactical sugar, or even HDL translators and simulators in VeriScala by themselves.

By embedding VeriScala in Scala programming language, it is possible to call methods from the extensive amount of Scala/Java libraries to help describe the designs. For example, as shown in the *Test* part in Figure 1, designers can use their favourite test libraries to run automatic tests in VeriScala. In some occasions, designers are required to perform some extra computations before writing the hardware specifications. For instance, a 0-1 integral linear program should be solved to make an efficient finite impulse response filter. To make this type of designs, hardware designers are usually required to use two distinct languages. In VeriScala, this can be done by using the Java linear programming libraries and then the results can be passed directly to functions responsible for generating the hardware designs. Designers can further package all the codes so others can generate the same efficient designs simply by importing the package.

3.1 Structure

The structure of a design in VeriScala is shown in the *Design* part in Figure 1. The HDL class can be considered as a container of modules. All the constructor parameters of an HDL class which are hardware registers will be scanned by the module macro and be converted to interface signals of the corresponding module. Inside each module, there can be an arbitrary number of synchronous or/and asynchronous blocks. For example, the code snippet *Adder.scala* in Figure 2 defines a module to add two inputs together.

The `add` method defines a module called `add` using the `module` macro. Macros are experimental features in Scala since 2.10, which allows programmers to define processes that will be expanded at compile time. Using this feature, VeriScala can traverse the parameter list of the enclosing HDL class and fetch all hardware registers from them which will later be inferred as input or output ports.

In this case, inside the `module`, a synchronous block is used whose inner logic will be triggered at every positive edge of `clk` as defined. In VeriScala, a `sync` block is used to describe sequential circuits and, correspondingly, an `async` block is used for combinational circuits.

It is worth noticing that designers can still define methods in an HDL module body to provide abstractions and code reusability to achieve parameterized design. A good example is to generate a sorting network using recursive functions, which can be found in Section 3.6.

3.2 Data Types and Operations

In VeriScala, hardware data types are distinguished by the `HDL` symbol in their type declarations, *i.e.*, `Boolean` representing a boolean type in Scala, while `HDL[Boolean]` representing a 1-bit hardware register.

Three built-in types in VeriScala can be filled in the `HDL` container, *i.e.*, `Boolean`, `Unsigned`, and `Signed`. `Boolean` is a Scala native type, while `Unsigned` and `Signed` are implemented by VeriScala for their absence in Scala. Although there are only three types that are natively supported by VeriScala, they can be easily extended by users.

The hardware registers are just special types in Scala, which can be used like all other values/variables in Scala. To use the operations and features provided by VeriScala, traits `BasicOps` and `compiler` must be mixed in, as shown

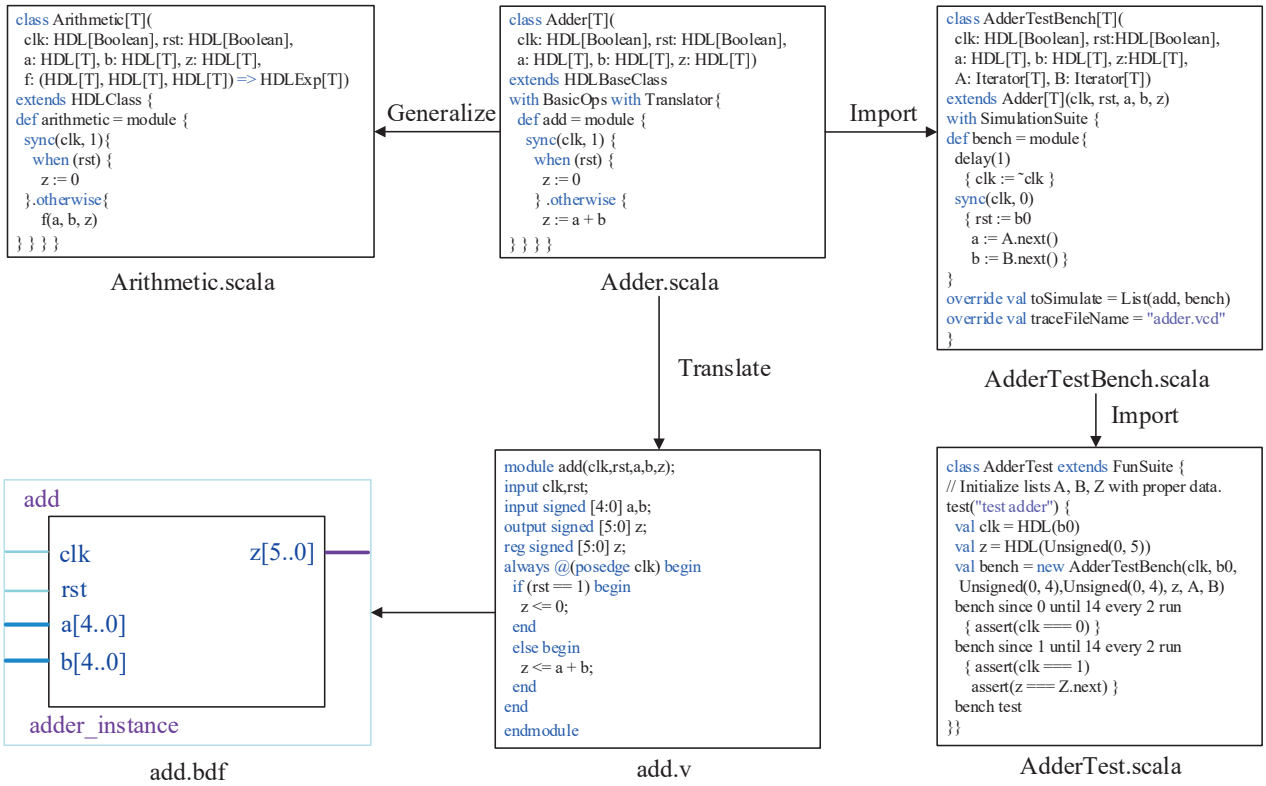

Figure 2: Derivation of variants of `Adder` code.

Table 1: Part of VeriScala operators on builtin data types.

Examples	Explanation
`a + b, a - b, a * b, a / b, a % b`	Arithmetic operations
`a > b, a < b, a >= b, a <= b`	Arithmetic comparisons
`a is b, a isnot b`	Equality comparison
`a > > Num, a < < Num`	Shifts
`a & b, a | b, a^b`	Bitwise operations
`a ∼∼ b`	Concatenation

in the code snippet in Figure 2. The trait `BasicOps` is built with support for all the basic arithmetic, bitwise, and logic operations on the three native types. Operations supported by VeriScala are listed in Table 1.

3.3 Statements

3.3.1 *Conditional Statement*

A conditional statement in hardware description generation could serve two distinct purposes: 1, generate different hardware descriptions based on the condition; 2, generate a hardware description with a conditional statement. In VeriScala, the first type of conditional statement can be achieved by using the original Scala `if` keyword, while the second type by using a special VeriScala `when` keyword. The `when` keyword takes type `HDL[Boolean]` as the condition, for the condition is to be determined at the hardware level. We can use both kinds of conditional statements as:

```
if (weNeedAMux2 == 1) {
    when (sel) {
        out := a
    }.otherwise {
        out := b
    }
} else {
    out := a
}
```

When the condition variable `weNeedAMux2` is assigned to `1`, the above code will specify a two-way multiplexer.

3.3.2 *Assignment Statement*

Assignment statements in hardware description do not always behave the same as that in software programming language. To differentiate these two kinds of assignments, we introduce `:=` to denote the assignments in hardware description. When hardware assignments appear in an `async` block, the statements will behave as expected. However, when they appear in a `sync` block, all the assignment statements will function at the same time, that is to say, the following code

```
sync(clk, 0) {
    a := b
    b := a
}
```

will swap the value of `a` and `b` at every negative edge of signal `clk`.

3.4 Functions

Using functions is a universal approach for code reuse. In VeriScala, we can define Scala functions with returning type as `HDLBlock` or `HDLExp` to reuse certain hardware logic:

```
def func(a: HDL[T], b: HDL[T]): HDLBlock
    = async { b := a }
def func(a: HDL[T], b: HDL[T]): HDLExp[T]
    = b := a
```

The keyword `def` is preserved by Scala and introduces a function definition, with each parameter and its type (either a native Scala type or a VeriScala hardware register type), and the logic descriptions as the function body.

3.5 Parametric Polymorphism

Parametrization is another important technique for code reuse. However, it is generally difficult for hardware designers to parameterize their design. Even though HDLs such as Verilog have already provided the keyword `parameter` to define frequently used constants and mechanisms to override them, it is still not possible to, for example, parameterize the data types.

By taking advantages of software programming features, the scope of parametrization is significantly improved. For example, by using generic type, we can design an additional module for an arbitrary type, as shown in Section 3.1. The actual type is determined only when the module is generated. We call this feature *parametric polymorphism*, since the generated code is also dependent on the type specified. Thus, by enabling the designers to write one single piece of generic code this can be derived into many similar designs. This brings the well-known principle "Don't Repeat Yourself" in software development to the world of hardware design.

This piece of code for adder can be further abstracted by making the circuit as a parameter, benefiting from the first class citizenship property provided by Scala. An example *Arithmetic.v* is shown in Figure 2. This arithmetic module takes a method `f` with type `(HDL[T], HDL[T], HDL[T]) => HDLExp[T]` as an argument, and thus the function of the module will be determined only when the method `f` is specified. This is another type of parametric polymorphism which allows designers to plug in the circuit described as they require in different scenarios.

3.6 Recursion

Besides flexible parametrization, recursive creation of hardware systems is also supported by VeriScala for code reuse. A sorting network would be a good example for illustrating this feature. Firstly, a comparator circuit is defined that outputs a pair of arguments `a` and `b` as `x` and `y` in the order specified by argument `dir`:

```
def compare(a: HDL[T], b: HDL[T],
x: HDL[T], y: HDL[T], dir: Int) {
   async {
     if (dir == ASC) {
       when (a > b) {
         x := b
         y := a
       } .otherwise {
         x := a
         y := b
       }
     } else {
       //Mirror code for DSC direction.
     }
   }
 }
```

It is worth mentioning that both types of conditional statements are used here, which enables designers to write more generic codes. To achieve the same effect in Verilog, we have to use compiler instructions or copy-and-paste approach to make two similar piece of code serving respective purposes. Neither of these methods meets the requirements of maintainability and readability. Then a helper method `bitonicMerge` is defined to sort a bitonic sequence:

```
def bitonicMerge(a: List[HDL[T]],
b: List[HDL[T]], dir: Int): HDLBlock = {
    val n = a.size
    val k = n / 2
    if (n > 1) {
      //Initialize list t.
      for (i <- 0 until k) {
        compare(a(i), a(i + k), t(i), t(i + k),
            dir)
      }
      bitonicMerge(t.take(k), b.take(k), dir)
      bitonicMerge(t.drop(k), b.drop(k), dir)
    } else {
      async {
        b.head := a.head
      }
    }
  }
```

It is noticed that this helper method is already recursive, and native Scala statements, for example `for` statement, are used freely in method definition. Finally we form the definition of method `bitonicSort`:

```
def bitonicSort(a: List[HDL[T]],
b: List[HDL[T]], dir: Int): HDLBlock = {
    val n = a.size
    val k = n / 2
    if (n > 1) {
      //Initialize list t.
      bitonicSort(a.take(k), t.take(k), ASC)
      bitonicSort(a.drop(k), t.drop(k), DES)
      bitonicMerge(t, b, dir)
    } else {
      async {
        b.head := a.head
      }
    }
  }
```

By recursively invoking method `bitonicMerge` from bottom to up, in each level, unordered sequence is first formed bitonically and then sorted. The way we define the sorting network circuits is just like the way we implement a typical divide-and-conquer algorithm in software programming language. Though `automatic` function is introduced in 2001 standard, Verilog is not able to describe this type of recursion, and extra dedicated scripts will be needed to generate code from such a recursive routine. By using VeriScala, the effort for implementing recursive design is significantly reduced.

3.7 Generating Synthesizable Code

VeriScala can automatically generate the code for synthesis (for now, only Verilog is supported). To do this

translation, the designer simply needs to override the `toCompile` value of the HDL class to specify the modules to be translated, and call the `compile()` method. Calling this method will generate the necessary Verilog code for synthesis. Type information such as whether a register is an input register or an output register, or whether it is a `wire` or a `register` in Verilog, will be automatically inferred by VeriScala without designers paying extra concerns.

The generated Verilog code for the `adder` described in Figure 1 is shown as the *add.v*. In the depicted code snippet, `Input/Output` wires are inferred correctly, the `sync` block is translated to the `always` block with the same sensitive list, and the `when` statement is translated to the `if` statement.

4. TEST AND DEBUG

Testing and debugging utilities in VeriScala framework compose the fundamental infrastructure to bring software applications and hardware modules into the same semantic context. They are both implemented as Scala libraries, which enable designers to complete the whole work flow without switching workspaces.

4.1 Programmatical Testing

In conventional approach to build a hardware test to find bugs, designers usually need to create a new project, *e.g.* a Modelsim project, write testbench and watch the generated waves. This is tedious and not effective. However, it is a common practice to write programs with automatic tests to validate its functionality and keep track of its quality in software development. VeriScala allows designers to utilize the test libraries of Scala and Java to automatically test their designs by providing a cycle-accurate software simulator. This simulator is implemented in Scala as a library as well and is integrated in the VeriScala environment seamlessly.

For example, to test the `Adder` module shown in Section 3.1, designers are firstly required to write a test bench before defining the tests. The code snippet *AdderTestBench.scala* listed in Figure 2 shows a test bench. In this case, the test bench is implemented as a class `AdderTestBench` that extends class `Adder` and includes specific test tasks. In the class, we define a `clk` that has a cycle of 2 absolute time units, for it flips every 1 unit, and a `sync` block that feeds the test data to `Adder` at each negative edge of the `clk`.

Because the simulator interface provided by `SimulationBase` is rather primitive, we have implemented another trait called `SimulationSchedule` which enables designers to arrange tasks using a scheduler and have the simulator run according to the tasks arranged.

After implementing the test bench, designers can import their favourite Java/Scala test libraries and write their test programs. In the listed code snippet *AdderTest.scala* depicted below *AdderTestBench.scala*, we use `FunSuite`, a test utility from `org.scalatest`, to set up the test cases.

4.2 Interactive Debugging

In addition to programmatical testing, interactive debugger is also widely used in developing software programs. In VeriScala, a primitive GDB-style interactive debugger is embedded in the software simulator. Furthermore, since software simulation sometimes violates the hardware behaviors, people would hope to debug the hardware designs locally on real FPGAs. By utilizing the HPS in the Altera DE1-SoC development board, we build a framework to support interactive debugging between client on PC and on-chip modules. The debugging interface is the same as that in software debugger, and thus the debugger on chip will support automatic testing as well. Figure 1 gives the overview of the debugging framework. The system is divided into two parts: a host on hardcore processor and a controller that encloses the modules to be tested.

The host program is implemented in C programming language, and it is running within a Linux operating system booted from SD card on the board. On the one hand, we build a `Putty` tool set based application, the interactive debugger, to build connection between the host and PC via serial port interface; on the other hand, an Avalon-MM device, the controller, is mapped to the host's address space which enables access to controller's register files. The Verilog code of controller is auto-generated by the VeriScala compiler according to specific annotations that specify the signals designers want to monitor, and is implemented as an Avalon-MM slave device.

The additionally generated configuration file records the name and data length of each inspected signal which helps the debugger application provide user-friendly interfaces. In the debugger, users can directly use the variables by their names without knowing the actual offsets in the hardware register file. Since the debugger is also implemented in Scala as a library, acceleration applications that read the computation results from hardware can be easily prototyped and tested in Scala context.

5. EVALUATION

To validate the implementation of VeriScala, we implement a number of basic circuits as well as some sophisticated systems such as MIPS CPU and video transcoder. By comparing the automatically generated Verilog code with the manual one in each case, we evaluate the VeriScala in three aspects:

- We check the correctness and compare the line counts of series of circuits to evaluate whether VeriScala has the same capability as Verilog on describing hardware design.
- We compare the resource consumption of both variants to evaluate whether the generated code has similar quality with manual one.
- We focus on readability and maintainability of Scala code to evaluate how will VeriScala improve productivity of hardware designing.

We also conduct a usability test on ten experienced high-level language programmers. By comparing, from different perspectives, the time costs and line counts for describing the same hardware logic both using VeriScala and Verilog, we evaluate the usability of VeriScala.

5.1 Environment Setup and Verifiability

In this section all the evaluation on hardware are based on Altera SoC-DE1 FPGA board with Cyclone V 5CSEMA5F31C6N chip, which is a widely used educational development board. To be compatible with our hardware platform, for synthesis, compilation and simulation, we choose Quartus II 14.0 and Modelsim 14.0, which are also provided by Altera.

[t]

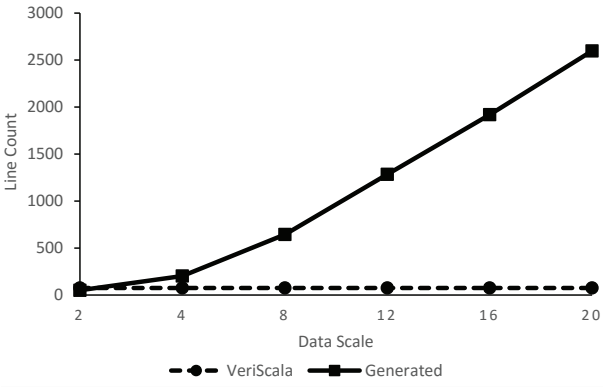

Figure 3: Line counts of Bitonic Sort implementation with increasing amount of data to be sorted.

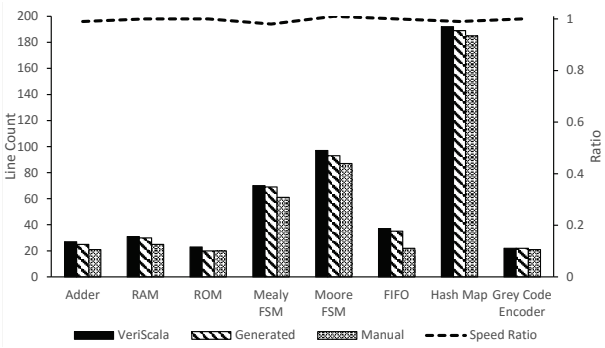

Figure 4: Line counts comparison among VeriScala, generated and manual implementation. Upper line shows the speed ratio of generated and manual implementation running the same test.

5.2 Functional Test

5.2.1 Basic Circuits

Firstly a set of basic circuits including FIFO, arithmetic units, gray code encoder, etc. are implemented as part of VeriScala's standard library. These modules are tested and work well both in software simulator and hardware. As presented in Figure 3, the speed ratio of generated and manual implementation when running the same test is very close to 1. Besides functional testing, the generated codes are also manually reviewed carefully, thus, it is reasonable to conclude that these two variants of implementations have the same logic. They are so simple that we do not need to include third party IP cores and basic syntax of VeriScala is sufficient. Figure 3 also shows a comparison of line counts of source code among VeriScala, generated Verilog and manual Verilog, which indicates that there is little difference in these cases when basic syntax is used. It is noticed that code of VeriScala is generally a little more than hand-written Verilog. Reasonably, this is because high level language suffered more from language overhead (*e.g.* import statements) that consists of nearly constant number of lines.

Further, we measure the resource consumption where both variants of code (generated and hand-written) are compiled with configuration to Cyclone V 5CSEMA5F31C6N. In this case, consumptions of Adaptive logic modules (ALMs), registers and pins are considered. Table 2 gives the results showing two variants need the same resources, since we draft our VeriScala implementation by referring to hand-written Verilog, which indicates that our system is able to map basic statements from VeriScala to Verilog without making significant affect to the nature of the expressed Verilog design.

Table 2: Resource consumptions of both variants (generated code/manual code). We measure the consumptions in ALMs, Pins and Registers.

Module		#ALMs	#Pins	#Regs
Basic circuits	RAM	9 / 9	10 / 10	15 / 15
	ROM	1 / 1	2 / 2	2 / 2
	FIFO	19 / 19	14 / 14	36 / 36
	Add	4 / 4	18 / 18	6 / 6
	Subtractor	4 / 4	18 / 18	6 / 6
	And	1 / 1	5 / 5	1 / 1
	Or	1 / 1	5 / 5	1 / 1
Common circuits	Encoder	2 / 2	8 / 8	0 / 0
	Mealy FSM	3 / 3	10 / 10	1 / 1
	Moore FSM	7 / 7	8 / 8	7 / 7
	Hash Map	102 / 107	58 / 58	119 / 116
	Bitonic Sort	3 / 3	6 / 6	6 / 6
Application systems	Mips CPU	1356 / 1163	108 / 108	1094 / 1094
	Transcoder	786 / 785	207 / 207	1349 / 1359

5.2.2 Bitonic Sorting

Bitonic sorting network is a typical application applying advanced language features, as shown in Section 3.6, recursion and parameterization, which are not synthesizable expressions in Verilog but commonly used in software programming languages. In terms of code structure, this implementation hardly differs from writing a normal software program. However, since the compiler of VeriScala generates synthesizable Verilog directly, *i.e.* generates a new copy of circuit each recursion, the number of Verilog code lines will explode when the scale of data to be sorted grows. Correspondingly, the hand-written code will have the same scale as the generated one, somehow, if there is a person who is willing to write this code manually. Figure 4 shows that line counts of VeriScala source code grows conforming to the theoretical complexity $O(n * (\log n)^2)$.

5.2.3 MIPS CPU

For a real world application, we implement core part of a full feature system, a five-stage single cycle CPU supporting MIPS instruction set. To validate the generated code, we run the same test bench for both variants on Modelsim, and additionally we build support to I/O with FPGA board along with a simple subtractor based on it to see if the

```
wire i_and = r_type & func[5] & ~func[4] &
  ~func[3] & func[2] & ~func[1] & ~func[0];
wire i_or = r_type & func[5] & ~func[4] &
  ~func[3] & func[2] & ~func[1] & func[0];
wire i_xor = r_type & func[5] & ~func[4] &
  ~func[3] & func[2] & func[1] & ~func[0];
... // 10+ more wires like this
```

(a) A Code Snippet from a Verilog MIPS CPU Design

```
def logic(a: HDL[Unsigned], s: String) = {
  val l = a.length
  (0 until l).zip(s).map(p =>
    if (p._2 == '1') a(p._1) else (~a(p._1)))
}
def logic2(r: HDL[Boolean],
  a: HDL[Unsigned], s: String) = {
  logic(a, s).foldLeft(HDLBitwiseAnd(r, u(1)))
    ((a, b) => a & b)
}
val map = Map(
  "i_and" -> "100100",
  "i_or" -> "100101",
  "i_xor" -> "100110",
  ... // 10+ more entries like this
}
...
i_and := logic2(r_type, func, map("i_and"))
```

(b) A Code Snippet from a VeriScala MIPS CPU Design

Figure 5: Control Unit Implementation Comparison.

system runs well on real hardware. This substractor is implemented by MIPS assembly code and interacts via the on-board interface, where switches are used to control each bit of input data and the 7-segment displays show the operators and result. It turns out that the automatically generated CPU works well both on simulator and hardware, for it has the same behavior as the manually written one.

The MIPS CPU is a relatively complex system that consists of multiple files and has an architecture of multiple layers. Besides basic statements, multi-inherit technique is introduced to express the CPU design. As mentioned in Section 3.1, VeriScala takes a Scala class as container of modules, that is to say, the CPU top design will be a Scala class inherited from some other classes of detail design (Control Unit, Arithmetic&Logical Unit, etc.). In describing this CPU, techniques used for developing software programs are applied freely, and we can define helper functions and data structures to make the code more easily readable and maintainable. Taking the control unit as an example, a piece of typical logic is shown in Figure 5a. With numbers of code lines in the same pattern, this error-prone code snippet is hard to maintain or debug. The hardware designers struggle with this sort of inability in abstraction of traditional HDLs, while the modern software developers are enjoying the benefits brought by high-order functions, first class functions, advanced data structures, etc. We are able to rewrite the logic in VeriScala to join the party.

Inspecting the code of lower part in Figure 5b, we first define a function named `logic` to generate a list that operations performed on each bit according to the provided string pattern, where `1` means nothing to do while `0` means flipping the bit. Then, another function `logic2` is defined that takes the result of function `logic` and folds it from left generating a one bit output. By applying these two functions, we give an abstraction of operation in Figure 5a, which makes it convenient to debug and for someone else to read. At last, we construct a map that maintains the corresponding operation code of each instruction, which makes, when operation code changed, no code need altering except the mapping values. Though we can reach the same goal by using bitwise operations and some other techniques in Verilog, the VeriScala style is a more intuitive way for high-level language programmers.

Table 2 gives the compiled results of both variants. We measure the consumption of ALMs and registers. It is shown that the generated code cuts 14.2% of consumption of ALMs and uses the same amount of registers. Since no manual optimization is applied in the VeriScala code, this reduction is most probably because that well organized code is more likely to be optimized by synthesizer. Thus, although we make more efforts to reconstruct the code in VeriScala, we enjoy the benefit of software engineering and gain quality improvements of the generated Verilog code as well.

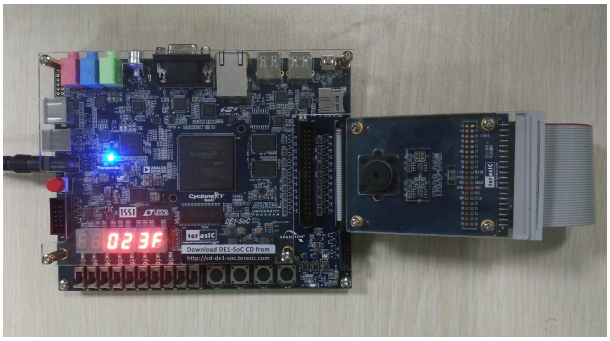

Figure 6: Video transcoder demo. The left side is the SoC-DE1 board, and the right side is the TRDB-D5M camera. The displays show the frame counts at the moment.

5.2.4 *Video Transcoder*

The Verilog design of the video transcoder is an example from the CD that comes along with the SoC-DE1 board. We implement the VeriScala variant based on this code. However, many encrypted IP cores are introduced and there is no corresponding implementation in current VeriScala libraries, which results in that we have to manually write some module instantiation statement in generated code. Figure 6 demonstrates how this system looks like. It uses a TRDB-D5M camera to capture raw data, then after transcoding and buffering, it displays the pictures via a VGA bus on the screen. We also measure the resource consumption of both variants, and, as shown in Table 2, there is little difference between them.

5.3 Usability Test

The test is conducted on ten experienced high-level language programmers with close age and years of programming career. The test includes the following steps:

1. Divide ten people into two groups. One is called VeriScala group, and the other Verilog group.

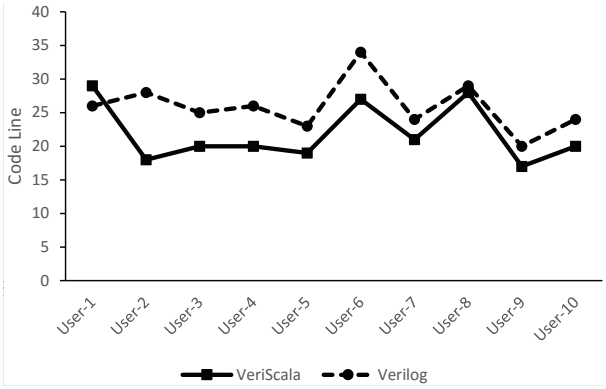

Figure 7: Line counts comparison between VeriScala and Verilog in implementing FlashLED circuit of ten volunteers.

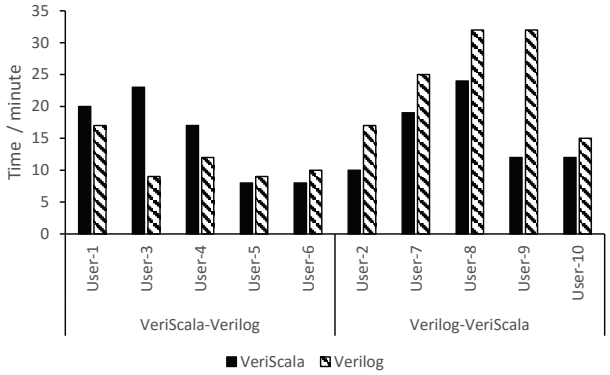

Figure 8: Time costs comparison between groups applying different implementing order. The left group firstly implement the VeriScala version, while the right group do the opposite.

2. Read the prepared specification booklet of VeriScala or Verilog corresponding to their groups, implement a FlashLED module after reading, and record the total costed time.
3. Swap the group, and do the jobs in last step again.
4. Count the code lines of each implementation and check the correctness.
5. Analyze the collected data.

Without or with little modification on syntax, every piece of code passes our test, which indicates that people find no trouble in using VeriScala to describe correct hardware design, and VeriScala is effective to learn. From the data, we firstly find that nearly every one use less code to do the job when using VeriScala, which can be seen in Figure 7. This result is different from the result shown in 5.2.1 where code lines of VeriScala will be a little more than that of Verilog. By reading the code, we find that people do not always use the same logic to implement the FlashLED, they will change their designs according to language features, such as high-order functions in Scala, which will obviously decrease the code lines.

Another discovery is that, shown in Figure 8, the total implementation time tends to be shorter, if they first learn VeriScala other than Verilog. Interviews indicate the reason, which is that Verilog is hard for beginners and VeriScala serves as a good warm-up.

6. RELATED WORK

HDL: Verilog and VHDL are traditional and dominant HDLs to provide description for hardware design and test-bench. [11]. However, on the one hand, they lack high-order features developed in the area of modern software programming languages, for example, they can not provide object-oriented abstractions and highly automatic code-reuse mechanism; on the other hand, they have very limited APIs, for example, it is not possible to open a network connection without complicated configuration. Moreover, these languages themselves are difficult to extend, that is to say, users can not implement syntax sugar at their will [7].

HLS: High-Level Synthesis (HLS) [9] provides a new way to help hardware designs with software programming paradigms. With HLS, it is easy for hardware designers to start with the high-level description of an application (mostly written in ANSI C/C++, SystemC [2], or SystemVerilog [16]), a RTL component library, and specific design constraints, then a HLS tool will generate the RTL architecture with automatic resource allocation, operation scheduling, and variable and transfer bindings [6]. However, there are still drawbacks of this approach. Firstly, HLS tools only provide limited customization, and thus it still requires huge amounts of manual optimizations to develop high-quality and practical designs [10]. Secondly, software developers benefit little from this approach for it requires sophisticated hardware knowledge. Finally, HLS provides no shared semantics for software application development, which introduces extra effort for development and maintenance.

DSL: Recently, some DSLs have been developed to generate the RTL architecture. HML (Hardware ML) [13] is a high-order hardware description language which supports polymorphic functions. Its HML-to-VHDL translator automatically infers types and interfaces, and generates a synthesizable subset of VHDL. Syspy [14], Pyverilog [20] and MyHDL [1] are the Python-based DSLs. The defined hardware components in SysPy can be used to build top-level structural descriptions of SoCs for the targeted Xilinx FPGA device. Pyverilog is a toolkit for RTL design analysis and code generation of Verilog HDL. It can help develop the framework for rapid prototyping and efficient functionality to implement a CAD tool. MyHDL uses functions with decorators to define modules, which contains a cycle-accurate simulator implemented in Python. Thus MyHDL enables designers to programmatically test their designs and call methods from Python libraries. However, Python is a dynamically typed language, though provides convenience in quick start-up, it lacks guarantee for type safety in hardware design.

Chisel [3] is an open-source hardware construction language based on Scala that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages. Chisel needs special semantics to specify input/output interface ports of modules since it treats Scala types and Chisel types distinctively, which result in no support for generalized parametrization (*e.g.* type parametrization). Chisel employs a C++ simulator, though it is fast, it denies access to the Scala testing framework and the extensive amount of Java/Scala

libraries.

Inspired by these high level DSLs, VeriScala not only supports the modeling and simulation of abstract design, but also supports low level hardware test and debugging by sharing with data structure between hardware and software, which provides an integrated framework for rapid prototyping of hardware acceleration applications. To the best of our knowledge, VeriScala is the first Scala DSL based framework that supports abstracted object-oriented hardware defining, programmatic testing, and interactive on-chip debugging both in software simulator and hardware.

7. CONCLUSION

In this paper, we propose an open-source DSL based framework for development of hardware-acceleration applications. In the framework, designers can describe their hardware designs using all the tools provided by the rich Scala eco-system, generate general and parameterized designs using the software programming features, and debug and test their designs programmatically both in software simulation and real FPGA environment. Through a set of real-world applications and usability test, we show that the framework is practical and reliable.

To the best of our knowledge, VeriScala is the first framework that integrates software and hardware design in the same design flow. We believe it is a novel practice in the field of software/hardware co-design. And thanks to the extendibility of Scala language, VeriScala is easy to have user-defined features by mixing in specific traits. We are actively pursuing future work to (1) provide optimization on generated Verilog code, (2) complete mechanism for invoking hardware modules in software semantics.

8. REFERENCES

[1] Myhdl. http://www.myhdl.org/.

[2] Systemc. http://www.systemc.org/home/.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[4] D. Chen, J. Cong, and P. Pan. Fpga design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[5] W. R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

[6] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design; Test of Computers*, 26(4):8–17, 2009.

[7] M. Flynn. Yesterday and tomorrow: a view on progress in computer design. In *IEEE Transactions on Very Large Scale Integration Systems*, page 239, Oct 2005.

[8] D. W. Franke and M. K. Purvis. Hardware/software codesign: A perspective. In *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991.*, pages 344–352, 1991.

[9] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[10] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[11] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Elsevier Science, Amsterdam, 2007.

[12] K. Jaic and M. C. Smith. Enhancing hardware design flows with myhdl. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 28–31, 2015.

[13] Y. Li and M. Leeser. Hml, a novel hardware description language and its translation to vhdl. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(1):1–8, 2000.

[14] E. Logaras and E. S. Manolakos. Syspy: using python for processor-centric soc design. In *17th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010, Athens, Greece, 12-15 December, 2010*, pages 762–765, 2010.

[15] S. Masuno, T. Maruyama, Y. Yamaguchi, and A. Konagaya. Multidimensional dynamic programming for homology search on distributed systems. In *Euro-Par 2006 Parallel Processing*, pages 1127–1137. Springer, 2006.

[16] D. I. Rich. The evolution of systemverilog. *IEEE Design & Test of Computers*, 20(4):82–84, 2003.

[17] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[18] S. Sutherland. What's new in verilog-2001. In *Verilog-2001*, pages 7–7. Springer, 2002.

[19] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[20] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog HDL. In *Applied Reconfigurable Computing - 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings*, pages 451–460, 2015.

[21] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanović. Ramp: Research accelerator for multiple processors. *IEEE Micro*, (2):46–57, 2007.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.