A Scala Based Framework for Developing Acceleration Systems Using FPGA

Yanqiang Liu[†], Yao Li[†], Weilun Xiong[†], Meng Lai[†], Cheng Chen[‡], Zhengwei Qi[†], Haibing Guan[†]
Shanghai Jiao Tong University[†], Morgan Stanley[‡]

Abstract

Field Programmable Gate Arrays (FPGAs) have gained momentum in the past decade, when the improvements of high end CPUs started to level-off in terms of clock frequencies. Since FPGAs can provide more flexible and more efficient solutions in most occasions, manufacturers gradually adopt FPGA chips in their products such as CPUs and mobile devices. In the foreseeing future, FPGAs will be accessible computing resources and software developers will build applications interacting with FPGAs. However, the efficiency of development for applications 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 descriptions. This paper proposes a new open-source Domain-Specific Language (DSL) based 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 debug and test hardware design interactively in real FPGA environment. Through the evaluation on real world applications, we show that VeriScala provides a practical approach for rapid prototyping of hardware acceleration systems.

1 Introduction

Due to the rapidly expanding data scale, emerging data analysis applications start to meet performance bottleneck and require hardware acceleration in recent years, which follows the long-standing hardware/software co-design [9] tradition. Known for their high customizability and parallel nature, FPGAs have gained momentum in a broad range of applications, *e.g.*, parallelization and distribution [25, 16], cryptography [18], and deep learning [26]. FPGAs have been gradually becoming mainstream accelerators in the industry [17, 19]. Besides adopting FPGAs on the server side, more and more FPGAs are adopted in mobile devices to provide high flexibility and customizability. It is foreseeable that software developers will build applications with FPGAs in the near future [6].

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 [13]. Compared with modern software programming languages, HDLs have been long criticized for their limited functionalities such as code reuse, high-level abstractions, and maintainability [13].

Furthermore, the gap between HDLs and software programming languages is still enlarging as more features are constantly being developed in the field of modern programming languages. A number of programming languages have appeared (*e.g.* Clojure, Scala, etc.) with new features such as 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 [22]), 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 working context of software engineers who are demanding the aid of hardware acceleration. Semantic gap between hardware descriptions and software applications prevents

[1] https://github.com/VeriScala/VeriScala.

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 software application developers have no direct way to participate 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 to traditional hardware design including object-oriented programming, parameterized types, type safety, test automation, etc. By providing an interactive software-hardware cooperating framework as a part of VeriScala, we provide convenience for rapid prototyping and propose an approach 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 describes the programmatic test supports; Section 5 shows hardware runtime details and cooperating system design in VeriScala; Section 6 provides evaluations on functionality; Section 7 and Section 8 shows the related work and concludes the paper, respectively.

2 Overview

2.1 Design Goal

Building 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 at Register Transfer Level (RTL), 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 that are used for designing, testing, and hardware runtime controlling.

Designing: 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.

Testing: 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.

Hardware Runtime: The Controller code and configuration file are generated for building and accessing hardware runtime. The controller is a wrapper of the developer defined hardware modules. Interface signals of those modules are buffered by the controller. The configuration file is used by driver libraries to specify the information of monitored signals. By using the driver libraries we build an interactive debugger as an example of software-hardware cooperating applications. Detailed discussion of hardware runtime will be presented in Section 5.

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 functional test. Verilog code and affiliated configuration file generated by VeriScala are used by specific driver libraries to build software-hardware cooperating applications such as an interactive debugger or a prototype of acceleration

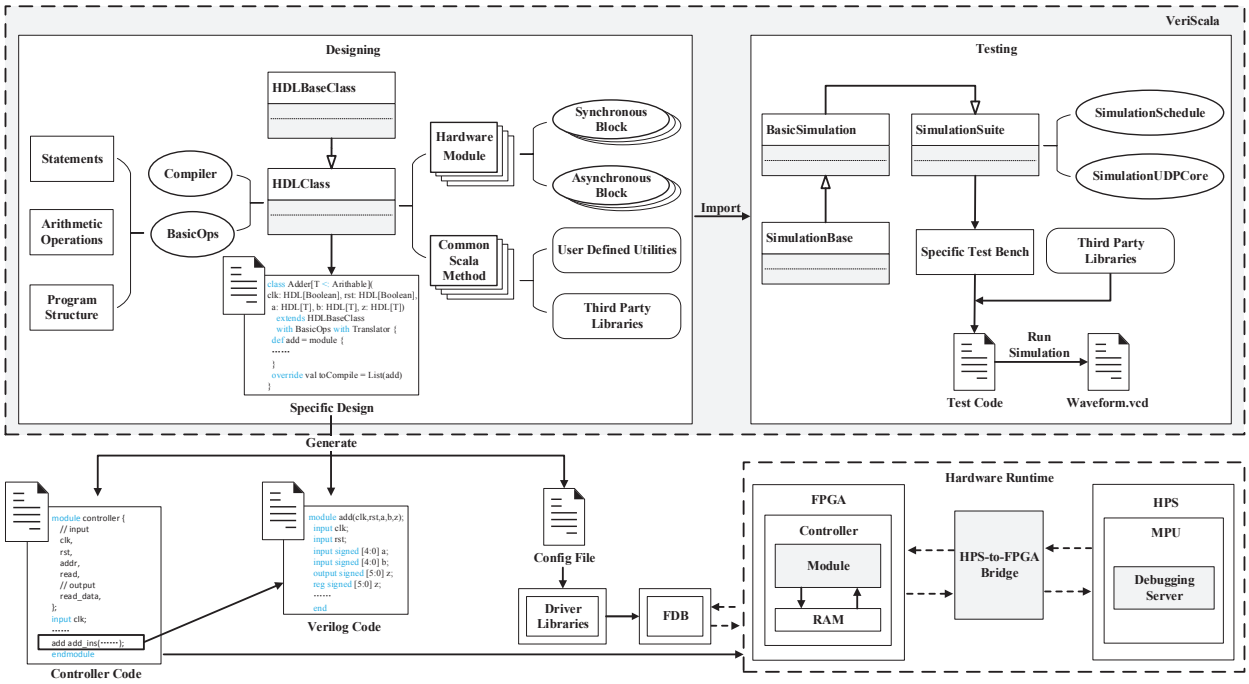

Figure 1: The architecture of VeriScala framework.

system.

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 accessing hardware runtime. 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. After validation of hardware design, by implementing and utilizing corresponding driver libraries, developers can easily build hardware acceleration systems in Scala context.

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) [21]. LMS is a novel implementation of multi-stage programming [23] on Scala. By using the `traits` 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 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.

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

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 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) {
```

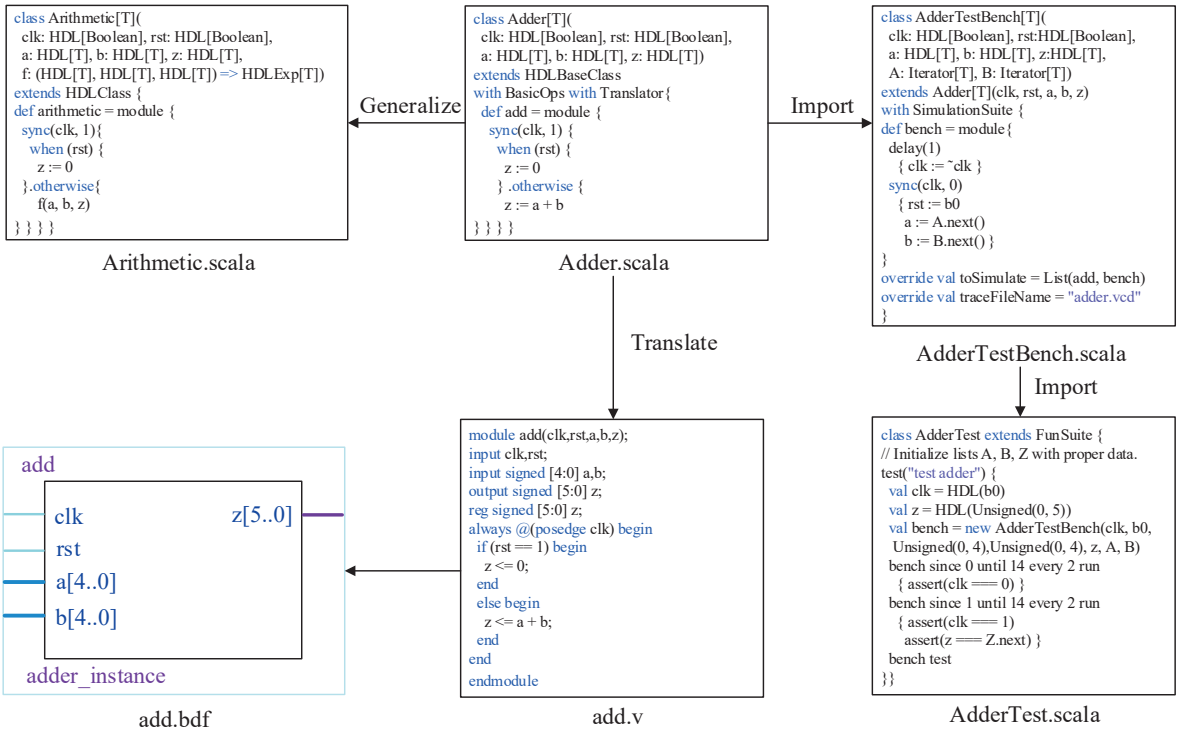

Figure 2: Derivation of variants of `Adder` code.

```
    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 im-

proved. 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 Extending the Language

Designers are encouraged to implement additional operations, syntactical sugar, or even HDL translators and simulators in VeriScala by themselves. For example, syntactical sugar for simulation can be provided by the `SimulationSchedule` trait shown in Figure 1, which can be implemented without modifying the VeriScala library. Additional operations can also be added to the three native types in VeriScala without modifying the library. However, without the support for infix

operators provided by Scala-Virtualized, this would be more difficult than it was shown in LMS [21]. This limitation can be overcome by using implicit conversions provided by Scala to first convert the original VeriScala types to some types defined by users, and then implement these additional operations on the customized types.

3.8 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

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.

Table 2: Two variants of APIs for different configurations of the controller module.

Format (Debug)	Explanation
`s arg1 arg2`	Buffer[arg1] ¡= arg2
`p arg1`	Read buffer[arg1]
`n`	Flip clock signal
Format (Release)	**Explanation**
`s arg1 arg2`	Buffer[arg1] ¡= arg2
`r arg1 arg2`	Run until buffer[arg1] is ready, and buffer[arg2] is the result

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.

5 Hardware Runtime

Utilities to access and control hardware runtime compose the fundamental infrastructure to bring software applications and hardware modules into the same semantic context. Figure 1 gives the overview of the hardware runtime. The system is divided into two parts: a host in hardcore processor system (HPS) and a controller that encloses the developer defined modules. To access the hardware runtime, specific driver library needs to be defined according to interface information specified in generated configuration file. This section will discuss the host server, controller, driver library and an example application.

5.1 Host Server and Controller

We build our demo framework in Altera DE1-SoC development board, which has a hardcore processor in its architecture. We use this HPS as a proxy between PC and FPGA. This is reasonable since, for the sake of system security, applications usually will not be allowed to access hardware resources directly. The proxy server is responsible to deal with the communication with PC and FPGA modules. The server program in the HPS is implemented in C programming language, and it is running within a Linux operating system booted from SD card on the board. To communicate with hardware

modules, Avalon-MM protocol is used to build mapping from Linux virtual address space to register files in FPGA. After being mounted in Linux system, the controller module can be manipulated by modifying specific registers, which is the same way how we access normal peripheral devices. On the other side, we adopt serial port interface to communicate with software applications on PC. The Server program provides a set of APIs, as shown in Table 2, for building driver libraries. Though we use HPS in our demo implementation, the whole design can be easily ported to pure FPGA with soft processor such as NIOS II.

The controller module is generated automatically according to defined hardware module. It is configured as an Avalon-MM slave device with a register file. By buffering interface signals of the module, we are able to access the hardware runtime information, for example clock counts and signal values. Input data stream from software applications will firstly be transferred to server in HPS, then the buffer in controller, and finally the working hardware module. The controller module can be configured into two modes, debugging mode and releasing mode. The only difference is the source of clock signal. In debugging mode, we need to run the module step by step, so we use the clock signal in buffer of controller that is provided by software applications such as debugger. While in releasing mode, we need the module to run at full performance, so we provide clock signal from real clock generator on the development board and listen when the work is not ready.

5.2 Driver library and Debugger

The driver library implemented in Scala gives the abstraction of hardware module by providing a set of programming interfaces. It hides communication details between software applications and FPGA from developers. Though it is needed to define different drivers for different hardware modules, we have define a template driver using serial port communication protocol, and thus slight modification is required according to configuration file to implement new drivers.

With the help of driver libraries, building software-hardware cooperating applications is convenient. Utilizing hardware assistant will be like accessing Remote Procedure Call (RPC) services. A GDB-sytle interactive debugger, shown in Figure 1 as FDB, is built as an example application in our framework to provide full features for developing acceleration systems. By importing right driver and providing corresponding configuration file, developers can debug the acceleration system in real FPGA environment.

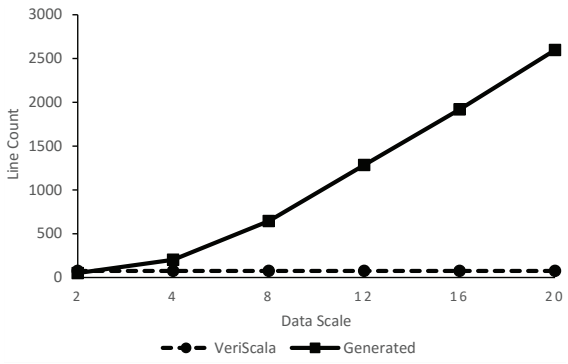

Figure 3: Line counts of Bitonic Sort implementation with increasing amount of data to be sorted.

6 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.

6.1 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 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.

6.2 Functional Test

6.2.1 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 passed tests both in software simulator and hardware. As presented in Figure 3, the speed ratio of generated and manual

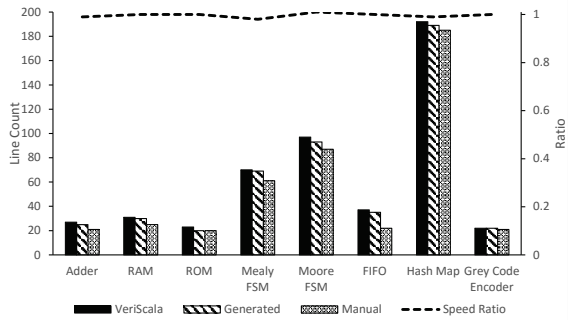

Figure 4: Line counts comparison among VeriScala, generated and manual implementation. Upper line shows the speed ratio of generated and manual implementation running the same test.

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 3 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.

6.2.2 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.*

Table 3: 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

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 $O(n*(\log n)^2)$.

6.2.3 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 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

```
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.

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 3 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.

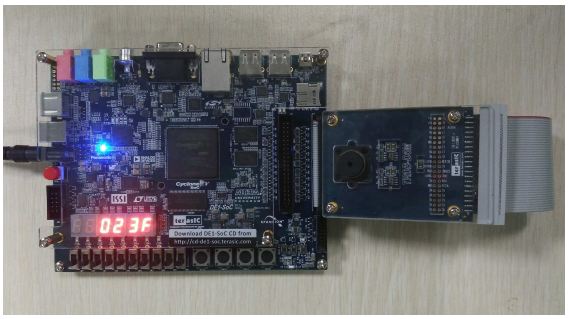

Figure 6: Video transcoder demo. The left side is the SoC-DE1 board, and the right side is the TRDB-D5M camera. The displays show the frame counts at the moment.

6.2.4 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 3, there is little difference between them.

7 Related Work

HDL: Verilog and VHDL are traditional and dominant HDLs to provide description for hardware design and testbench [12]. 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 wills [8].

HLS: High-Level Synthesis (HLS) [10] 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 [20]), 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 [7]. 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 [11]. 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) [14] 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 [15], Pyverilog [24] 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.

8 Conclusion

In this paper, we propose an open-source DSL based framework for developing hardware-acceleration applications. In the framework, designers can describe their hardware designs using all the tools provided by the rich eco-system of Scala, 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 evaluation on a set of real-world applications, 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. The project is open-source and all of the implementation and testbench mentioned in this paper is available at our project page.

References

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

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

[3] BACHRACH, J., VO, H., RICHARDS, B., LEE, Y., WATERMAN, A., AVIŽIENIS, R., WAWRZYNEK, J., AND ASANOVIĆ, K. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (2012), ACM, pp. 1216–1225.

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

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

[6] COUGHLIN, M., ISMAIL, A., AND KELLER, E. Apps with hardware: enabling run-time architectural customization in smart phones. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), USENIX Association, pp. 621–634.

[7] COUSSY, P., GAJSKI, D. D., MEREDITH, M., AND TAKACH, A. An introduction to high-level synthesis. *IEEE Design; Test of Computers 26*, 4 (2009), 8–17.

[8] FLYNN, M. Yesterday and tomorrow: a view on progress in computer design. In *IEEE Transactions on Very Large Scale Integration Systems* (Oct 2005), p. 239.

[9] FRANKE, D. W., AND PURVIS, M. K. Hardware/software codesign: A perspective. In *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991*. (1991), pp. 344–352.

[10] GAJSKI, D. D., DUTT, N. D., WU, A. C., AND LIN, S. Y. *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[11] GEORGE, N., LEE, H., NOVO, D., ROMPF, T., BROWN, K., SUJEETH, A., ODERSKY, M., OLUKOTUN, K., AND IENNE, P. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on* (Sept 2014), pp. 1–8.

[12] HARRIS, D., AND HARRIS, S. *Digital Design and Computer Architecture*. Elsevier Science, Amsterdam, 2007.

[13] JAIC, K., AND SMITH, M. C. 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* (2015), pp. 28–31.

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

[15] LOGARAS, E., AND MANOLAKOS, E. S. 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* (2010), pp. 762–765.

[16] MASUNO, S., MARUYAMA, T., YAMAGUCHI, Y., AND KONAGAYA, A. Multidimensional dynamic programming for homology search on distributed systems. In *Euro-Par 2006 Parallel Processing*. Springer, 2006, pp. 1127–1137.

[17] OUYANG, J., LIN, S., QI, W., WANG, Y., YU, B., AND JIANG, S. Sda: Software-defined accelerator for large-scale dnn systems. In *Hot Chips 26 Symposium (HCS), 2014 IEEE* (2014), IEEE, pp. 1–23.

[18] PAGLIARI, D. J., CASU, M. R., AND CARTONI, L. P. Acceleration of microwave imaging algorithms for breast cancer detection via high-level synthesis. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on* (2015), IEEE, pp. 475–478.

[19] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., ET AL. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on* (2014), IEEE, pp. 13–24.

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

[21] ROMPF, T., AND ODERSKY, M. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices* (2010), vol. 46, ACM, pp. 127–136.

[22] SUTHERLAND, S. What's new in verilog-2001. In *Verilog-2001*. Springer, 2002, pp. 7–7.

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

[24] TAKAMAEDA-YAMAZAKI, S. 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* (2015), pp. 451–460.

[25] WAWRZYNEK, J., PATTERSON, D., OSKIN, M., LU, S.-L., KOZYRAKIS, C., HOE, J. C., CHIOU, D., AND ASANOVIĆ, K. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 2 (2007), 46–57.

[26] ZHANG, C., LI, P., SUN, G., GUAN, Y., XIAO, B., AND CONG, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2015), ACM, pp. 161–170.