# A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity

**Lenny Truong**
Stanford University, USA
lenny@cs.stanford.edu

**Pat Hanrahan**
Stanford University, USA
hanrahan@cs.stanford.edu

## Abstract

Leading experts have declared that there is an impending golden age of computer architecture. During this age, the rate at which architects will be able to innovate will be directly tied to the design and implementation of the hardware description languages they use. Thus, the programming languages community stands on the critical path to this new golden age. This implies that we are also on the cusp of a golden age of hardware description languages. In this paper, we discuss the intellectual challenges facing researchers interested in hardware description language design, compilers, and formal methods. The major theme will be identifying opportunities to apply programming language techniques to address issues in hardware design productivity. Then, we present a vision for a multi-language system that provides a framework for developing solutions to these intellectual problems. This vision is based on a meta-programmed host language combined with a core embedded hardware description language that is used as the basis for the research and development of a sea of domain-specific languages. Central to the design of this system is the core language which is based on an abstraction that provides a general mechanism for the composition of hardware components described in any language.

**2012 ACM Subject Classification** Hardware → Hardware description languages and compilation

**Keywords and phrases** Hardware Description Languages

**Digital Object Identifier** 10.4230/LIPIcs.SNAPL.2019.7

**Funding** The authors would like to thank the DARPA DSSoC program (contract # FA8650-18-2-7861), the AHA and SystemX affiliate programs, and Intel's Agile ISTC for supporting this work.

**Acknowledgements** The authors would like to thank the anonymous reviewers and Will Crichton for providing feedback on paper drafts. They would also like to thank their colleagues at Stanford, particularly members of the HardwarePL Reading Group and the AHA Agile Hardware Center, as many of the ideas in this paper are the direct or indirect result of the many intellectual discussions that they have collectively participated in. Finally, the authors would like to thank the organizers of SNAPL for creating a venue for visionary papers on programming languages.

## 1 Introduction

Turing award winners John Hennessy and David Patterson recently declared that we are on the cusp of a new golden age of computer architecture [29, 30]. Current trends in silicon manufacturing are signaling the end of Moore's law and Dennard scaling. This, combined with the inherent inefficiencies in general-purpose processor design, indicates that new innovations in computer architecture will come from the design of domain-specific architectures. The recent proliferation of application accelerators, such as Apple's neural engine for the A12 and Google's Tensor Processing Unit, support the idea that the hardware community is transitioning to specialized chip design. This shift signals a new golden age because researchers have an opportunity to develop radically different architectures rather than incremental improvements to existing processor designs.

Domain-specific chips necessarily target smaller markets which implies that design teams will become smaller and demand more of their tools in order to be productive. This indicates that we are also on the cusp of a new golden age of hardware description languages (HDLs) because hardware designers are actively seeking radically new technologies that will dramatically reduce design time and cost. A major impediment to design productivity is the fact that the hardware ecosystem is a monoculture comprised of a few chip designs from a small number of manufacturers. It is essential that the hardware community shifts towards a diverse ecosystem of easily accessible intellectual property blocks that can be composed to construct new chip designs. Underlying this shift will be advances in HDLs that promote the proliferation of hardware libraries.

The development of HDL abstractions that promote reuse, correctness, and performance represents the main challenge for this new golden age of HDLs. Fortunately, the programming languages (PL) community has enjoyed a rich history of success in developing techniques to address these issues. HDL researchers have already started to tackle the reuse problem by applying standard software programming language techniques such as meta-programming, polymorphism, and abstract data types [4, 11]. Evidently HDL researchers stand to benefit greatly from the lessons learned by their software language counterparts.

This paper identifies three problem domains that lie at the intersection of programming languages and hardware: language design, compiler infrastructure, and formal methods. For those unfamiliar with the hardware design process or HDLs, Section 2 covers the essential concepts required to understand the intellectual challenges discussed in Section 3. Section 4 presents a vision of a multi-language system for constructing hardware that is designed to address these intellectual challenges. With this impending golden age of HDLs, it is an exciting time to be interested in programming languages and hardware.

## 2 Background

A *hardware description language* (HDL) is an instance of a programming language that has been designed to provide abstractions for describing circuits. This paper will focus on the discussion of digital hardware, where circuits are described as logic operating on discrete, binary signals. Digital hardware can be further divided into two categories: synchronous and asynchronous. In a synchronous circuit, state changes are synchronized by a clock signal. In contrast, asynchronous circuits can contain state elements that change at any time. A majority of modern digital designs are synchronous, but the increasing demand for efficiency has renewed interest in asynchronous designs or hybrid models such as globally asynchronous, locally synchronous [58]. A digital HDL is defined to be *expressively complete* if it can be used to express both synchronous and asynchronous designs.

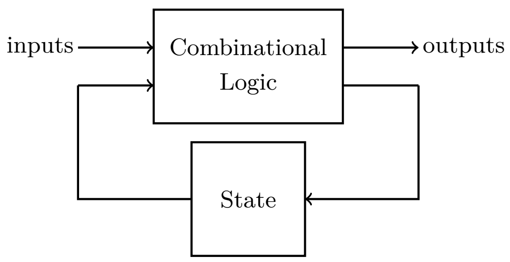

**Figure 1** An abstract depiction of a sequential logic circuit constructed as the composition of combinational logic with state. Notice that the outputs *could* depend on the inputs, implying that the circuit *could* describe a Mealey machine [43]. Depending on the mechanism chosen for storing state, the circuit could be synchronous or asynchronous. See Section 2.1 for more details.

## 2.1 Digital Design

In digital circuit theory, *combinational logic* refers to circuits where the output is a pure and total function of the inputs. In contrast, *sequential logic* refers circuits where the outputs are dependent on the sequence of past inputs. Figure 1 depicts the canonical design pattern for using combinational logic circuits composed with state to construct sequential logic circuits. Sequential logic circuits are used to implement *finite-state machines* (FSMs), a fundamental component of building digital systems.

When discussing FSMs in the context of hardware design, it is important to recognize the distinction between Mealey [43] and Moore [45] machines. For a Mealey machine, the output of the circuit is a function of the state and the inputs, while for a Moore machine, the output of the circuit is purely dependent on the state. The differences between these classes of FSMs are more pronounced in hardware than in software because they exhibit different timing characteristics. When using registers to store state, a Moore machine can be viewed as a purely synchronous entity where changes to the state and output values are triggered by a clock, while a Mealey machine can exhibit asynchronous behavior where output values immediately respond to changes to input values. An expressively complete HDL will be able to describe and compose both Moore and Mealey machines.

## 2.2 Verilog

The *Verilog* language is the dominant HDL used in practice today [23]. The language was originally developed as a commercial verification and simulation product [24] and was later adopted as a basis for logic synthesis. As a result, the semantics of the language are defined in terms of a hardware simulation being executed as a software program [57]. The design of the language is directly inspired by C, exhibiting many of the same features including a preprocessor, control flow, and operators. Like C, Verilog has become the *lingua franca* of the HDL ecosystem and is used as the common interchange format for design tools.

The core of Verilog's semantics is based on a module abstraction which shares many similarities to function abstraction from software languages. A *module* has an interface and a definition. An *interface* is a set of typed ports. A *port* is similar to a function argument or return value and represents a named entity that is used to consume or produce data. A *definition* is a list of statements that describe the module behavior using various language features such as the wiring and module instancing operators. Verilog designs are comprised of hierarchically composed modules that are simulated using a dataflow execution model. Figure 2 shows an edge detector FSM written in Verilog.

```
 1  module edge_detector(input in, output out, input clk);
 2      localparam A=0, B=1, C=2;
 3
 4      reg [1:0] state,      // Current state
 5                nextState; // Next state
 6
 7      always @(posedge clk) begin
 8          if (reset) begin
 9              state <= A; // Initial state
10          end else begin
11              state <= nextState;
12          end
13      end
14
15      always @(*) begin
16          nextState = state;
17          out = 0;
18          case (state)
19              A : if (in) nextState = C;
20                  else nextState = B;
21              B : if (in) begin
22                      out = 1;
23                      nextState = C;
24                  end
25              C : if (~in) begin
26                      out = 1;
27                      nextState = B;
28                  end
29              default : begin
30                  out = 1'bX;
31                  nextState = 3'bX;
32              end
33          endcase
34      end
35  endmodule
```

**Figure 2** Verilog implementation of an edge detector FSM adapted from the University of Washington CSE370 course materials [20]. The circuit has two inputs and one output and is designed as a Mealey machine where the output is 1 if the current value of `in` is the inverse of the previous value of `in` (i.e., the input is changing from 1 to 0 or from 0 to 1). Line 1 declares the module name and interface. The ports have an implicit width of 1 bit and are qualified with a direction `input` or `output`. Line 2 declares a set of constants that are used to abstract the encoding of the FSM states. Lines 4 and 5 declare variables to hold the current and next state. Lines 7-13 describe the state update logic inside a Verilog `always` block. This block of code defines a procedure to run when a `posedge clk` event occurs. That is, on a positive clock edge, update or reset the `state` variable. Lines 15-34 define another `always` block that is sensitive to changes to any input signal, denoted by the `@(*)`. This means that if any input value changes, this block of code will fire. The block encodes the combinational logic for computing the output and next state values as a function of the input and current state values. Because the second `always` block is sensitive to any input change, the semantics are defined asynchronously. Contrast this with the first `always` block which enforces the state updates to be synchronous by only executing on the positive edge of the clock. On lines 30 and 31, the values of `out` and `nextState` are assigned the value `X` to explicitly indicate they are undefined and can be any value. See Section 2.2 for more details.

```haskell
acc1 :: Stream Word -> Stream Word
acc1 in = out
     where
         out = (delay out 0) + in

-- input -> current state -> (new state, output)
acc2 :: Word -> Word -> (Word, Word)
acc2 in s = (s', out)
     where
         out = s + in
         s' = out
```

**Figure 3** An example of two mechanisms for encoding state in a Haskell embedded HDL adapted from the Clash documentation [39]. Both functions describe an accumulator architecture that stores a running count of the input values over time. The function `acc1` shows the first approach which is based on a `Stream` data structure with a `delay` operator. The `delay` operator returns the input stream with the values shifted by one cycle. The second argument to `delay` is used to specify the first value of the stream. The function `acc2` uses a different approach where the current state is passed as an argument and the next state is returned as an output. See Section 2.3 for more details.

## 2.3 Functional HDLs

HDL development has a long tradition in the functional languages community [46]. Functional HDLs leverage the idea that a pure function can be used to model combinational logic. The fundamental problem these languages face is integrating the concepts of time and state in order to enable the description of sequential logic.

$\mu$FP [55] and Daisy [35] both introduced a technique based on reactive programming where a stream data structure is used to describe circuits where the output can depend on the history of the inputs. $\mu$FP extends the FP language with a recursively defined $\mu$ operator which takes a function and produces a new function with internal state. The essence of the $\mu$ operator is that it supplies the current value of the state as an input to the function, and it uses an output of the function to set the next value of the state. Daisy uses a different approach by modeling sequential logic using recursive equations. Both these approaches required the development of a new language in order to implement their ideas. The designers of Clash [39] recognized that Haskell's lazy evaluation can be used to construct infinite streams, indicating that it could serve as a host for an embedded HDL. Figure 3 shows how Haskell's semantics enable the description of sequential logic circuits.

One interesting technique applied to functional HDLs is the use of combinators to describe circuit structure. Hydra [50] showed that a recursive, stream-based abstraction enabled the use of higher-order functions to capture structural patterns. Lava [11] extended the use of recursive data types with the ability to describe general circuit networks rather than just tree-like structures. This technique provides powerful facilities for code reuse by enabling the description of circuits as a regular pattern of components. In practice, this approach has proved particularly useful when applied to the problem of circuit layout [56].

## 2.4 Term Rewriting Systems

Another lineage of work [32] has explored the application of term rewriting systems (TRS) [52] to the description of hardware. In these systems, circuits are described as a set of rewrite rules which are applied to the inputs and state values to produce the outputs and next

```
1  module mkCounter(Counter);
2      Reg#(Bit#(8)) value <- mkReg(0);
3
4      method Bit#(8) read();
5          return value;
6      endmethod
7
8      method Action load(Bit#(8) newval);
9          value <= newval;
10     endmethod
11
12     method Action increment();
13         value <= value + 1;
14     endmethod
15 endmodule
```

**Figure 4** Implementation of a synchronous counter adapted from the Bluespec tutorial [12] with the module interface specification omitted. Line 2 declares an 8-bit register named `value`. Lines 4-14 define the implementation of the `read`, `load`, and `increment` methods which define the behavior of the module. Notice that the compiler must handle the data race between `load` and `increment` on `value`. See Section 2.4 for more details.

state values. An important quality of TRS is that they model the non-determinism and concurrency that are intrinsically present in hardware. For example, a conflict could occur when two rules match the same input data and try to update the same state element. The development of schemes for detecting and arbitrating conflicts represents the main intellectual challenge for these systems. Figure 4 shows a synchronous counter written in Bluespec [47], an established HDL based on TRS.

## 2.5 High-level Synthesis

High-level synthesis (HLS) is a technique that is broadly defined as compiling general software programs to hardware [65]. This paper will eschew the use of the term HLS due to the ambiguity of what is considered high-level. For example, a recent survey evaluated HLS tools using benchmarks written in C [46]. However, the PL community would consider C to be a low-level language. Instead, this paper will use the concept of the virtual machine abstraction to encompass the languages used as input to HLS systems. Languages based on a virtual machine abstraction provide some notion of unbounded resources such as an infinite register space. Section 3.1.3 discusses this in more detail.

A hardware compiler for a general purpose programming language relies on a strategy for mapping a program that may be unbounded in time and space into a finite set of resources. Typically this involves exploring the trade-offs between scheduling computation in space or time. If the compiler can determine parallelism in some computation, this logic can be mapped into concurrently executing hardware modules. However, data dependencies, finite resources, and suboptimal cost models complicate the task for larger applications. The compiler must use heuristics to schedule computation into the time dimension and insert the requisite logic to orchestrate the sequencing of the computation. Figure 5 shows a synchronous counter implemented in SystemC [22], a subset of the C language used for HLS.

```
 1 SC_MODULE (counter) {
 2     sc_in_clk clock;
 3     sc_in<bool> reset;
 4     sc_out<sc_uint<4> > counter_out;
 5
 6     sc_uint<4> count;
 7
 8     void incr_count () {
 9         if (reset.read() == 1) {
10             count = 0;
11             counter_out.write(count);
12         } else {
13             count = count + 1;
14             counter_out.write(count);
15         }
16     }
17
18     SC_CTOR(counter) {
19         SC_METHOD(incr_count);
20         sensitive << reset;
21         sensitive << clock.pos();
22     }
23 };
```

**Figure 5** Example of a 4-bit counter defined in SystemC adapted from EDAplayground [1]. Lines 2-4 declare the interface of the module. Line 6 declares an internal state variable. Lines 8-16 define a method `incr_count` which implements the behavior of the counter using the SystemC data types. Notice that input ports are read using the `read` method and outputs are written using the `write` method. The rest of the body of the definition is interpreted as normal C code. Lines 18-22 define a constructor for the counter object and is mainly responsible for defining the sensitivity of the module to the `reset` input as well as the positive edge of the `clock` input. See Section 2.5 for more details.

## 3 Intellectual Challenges

This section divides the concerns of HDL research into three intellectual domains: language design, compiler infrastructure, and formal methods. Each of these domains represents a subset of a more general research area that is of interest to the broader PL community.

### 3.1 Language Design

The general discipline of programming language design revolves around the development of abstractions. A language designer will employ abstractions to enable the user to ignore certain details about a program. Well-designed abstractions make the development and maintenance of programs easier. In some cases, such as in domain-specific languages (DSLs), abstractions also serve as a basis for the development of compiler optimizations. In this impending golden age, the main challenge for HDL designers will be devising and composing abstractions that enable code reuse, improve correctness of programs, and that can be used to construct designs that produce high quality results from a compiler.

There are three major levels of abstractions employed in modern HDL design. The lowest level is the *circuit* abstraction where hardware is modeled as a graph of connected components. The next level is the *register-transfer* abstraction where hardware is described as the computation on data flowing between registers. The highest level is the *virtual machine* abstraction where hardware is modeled as a set of instructions for an abstract machine. Many HDLs incorporate abstractions from multiple levels. For example, the Verilog language is based on the circuit abstraction but also provides various facilities for describing hardware using the register-transfer abstraction.

### 3.1.1 Circuit Abstraction

In the *circuit* abstraction, hardware is described as a graph of connected components. The abstraction consists of three primitive concepts: circuits, ports, and wires. A *circuit* has an interface and a definition. An *interface* consists of a set of ports. A *port* is a named entity that is used to consume or produce data. A *definition* contains a set of circuit instances and wires. A *wire* connects two ports. In the hardware community, language features based on the circuit abstraction are described as *structural*. For example, a design written in structural Verilog will only use the language features for defining, instancing and wiring up Verilog modules. Purely structural designs are called *netlists*.

The dominant structural HDL in use today is Verilog. Chisel [4] and Magma [27] are examples of an emerging subclass of structural languages called *Hardware Construction Languages*. These languages embed the circuit abstraction into a general purpose programming language which provides a mechanism for meta-programming circuit definitions. This approach exhibits a distinct advantage over Verilog; moving features related to parametrization and code generation to the host language simplifies the precise specification of the HDL.

In theory, a purely structural HDL is expressive enough to capture any real world digital hardware design. This is argued by the fact that the physical result from manufacturing hardware is always a component that is composed of connected sub-components, recursing all the way down to the transistors. Based on this fact, we posit that the circuit abstraction is the fundamental primitive upon which all other HDL abstractions can be constructed. Remark that the abstraction is agnostic as to whether the behavior of the circuit is synchronous or asynchronous which indicates that it is expressively complete.

The main challenge for the circuit abstraction is determining whether the connection between two ports is semantically correct with respect to the intended behavior of the design. Most HDLs attach a notion of direction to ports which enables the use of a type system to check that only an output can be connected to an input. An interesting research direction moving forward will be increasing the expressiveness of the types used for circuit ports. Ideally these types are able to capture the semantics of the protocols used to communicate between two components. Section 3.3.2 provides a more detailed discussion on how session types might be used to address this issue.

An interesting quality of the circuit abstraction is that, while it is based on the low-level details of hardware design, it can be used to compose black box modules at any level in the design hierarchy. This makes it a compelling basis for the development of hardware libraries. As discussed in Section 2.3, functional HDLs with a circuit abstraction can leverage combinator patterns to construct reusable circuit structures. Further research on language facilities that enable the construction of hardware libraries based on the circuit abstraction will be an essential component of this new golden age of HDLs.

### 3.1.2 Register-Transfer Abstraction

The *register-transfer* abstraction models hardware as computation on data flowing between registers. Registers are defined as primitive data storage elements that update their values based on a clock signal. Because register semantics are intrinsically tied to a clock, this abstraction is concerned with the description synchronous digital circuits. However, it important to note that this abstraction could be composed with other abstractions for describing asynchronous logic. In the hardware community, languages using the register-transfer abstraction are described as *register-transfer level* (RTL) languages.

A structural HDL that includes a notion of a register provides a register-transfer abstraction; the computation on data flowing between registers is described using circuit instances and connections. However, the register-transfer level of abstraction encompasses a broader set of concepts such as functions and operators. In practice, most HDLs combine the register-transfer abstraction with the circuit abstraction by extending the concept of a circuit definition to include constructs such as expressions, statements, and procedures. This technique raises the level of abstraction by removing the need to explicitly define, instance, and wire up register circuits. Instead registers are treated as language primitives that behave similarly to variables in a standard software programming language.

The fundamental issue in RTL language design is the precise choice of semantics for abstracting the concept of a register. For example, the Verilog `always` block provides a procedural abstraction for describing the simulation behavior of a component. To model a register, the designer uses variables to store data across clock events. The Verilog specification is explicit in stating that a variable does not imply a hardware register [57], instead it is left to the synthesis tool to determine how the simulation behavior of an `always` block can be mapped into an implementation using concrete hardware registers. This design choice raises the level of abstraction by enabling the user to ignore details about how the program is concretely implemented in hardware. However, it also removes the ability for the user to explicitly specify the registers used in the synthesized design. In practice, this choice can result in a mismatch between the results of register synthesis and the designer's intent. To remedy this, Verilog design teams enforce style guidelines that restrict the usage of `always` blocks such that the synthesis results are transparent. An alternative design could use qualifiers to explicitly declare variables that should be hardware registers.

Choosing how to map the concept of a variable to a register is a fundamental design issue for all imperative RTL HDLs. A related issue is reconciling the synchronous update semantics of registers with the asynchronous update semantics of standard variables. For example, given standard imperative evaluation semantics, writing to two different variables in a procedure would happen at different steps in the evaluation. However, if both variables are mapped to hardware registers, they would be updated at the same time in the synthesized hardware. One design choice would be to explicitly model the synchronized temporal update semantics of a register variable in the evaluation semantics of the language, ensuring that the user's model of the computation exactly matches the behavior of the synthesized hardware.

Verilog's non-blocking assignments provide the capability to explicitly model synchronous storage. In Verilog, a blocking assignment is executed before the subsequent statements in a block of sequential code. In contrast, a non-blocking assignment does not block procedural flow and is performed near the end of a time-step. When combined with clock events, the non-blocking assignment can be used to model the synchronous update semantics of hardware registers by delaying variable updates until the end of a clock period.

The interplay between these two forms of assignments comprise a major component of the complexity of the Verilog specification. One might think that the semantics of non-blocking assignment could be simply to delay the evaluation until all blocking assignments have been completed. However, the evaluation of a non-blocking assignment will trigger an event on the variable being assigned, which in turn may trigger an event involving a blocking assignment. To handle this, Verilog's semantics include a loop for each time step that moves between blocking and non-blocking assignments until there are no events left to process. The complexity of these evaluation semantics can make it difficult to reason about a Verilog design involving both forms of assignment. In practice, Verilog design teams will follow style guides that enforce the usage of assignments in a reasonable to understand manner.

The functional HDLs described in Section 2.3 demonstrate two more techniques for abstracting the concept of a register. One approach is to encode the state in the interface of a function. This is done by describing a circuit as a function that consumes the current state as one of its inputs and produces the next state as one of its outputs. A function of this form describes the transition function of a finite-state transducer (FST), providing a basis for a simple hardware synthesis algorithm where the current state is stored in registers. This technique necessarily implies a synchronized state update because the next values of the state are produced at the end of the evaluation of a function. There is no means to specify a state update at any other time.

The second approach uses a stream-based abstraction to encode state. The inputs and outputs of a function are a `Stream` data structure with a special `delay` operator that allows the user to look into the past values of a `Stream`. Given this operator, the user may describe a circuit where the values of an output stream depend on the values of an input stream at a prior clock cycle. The simplest compilation algorithm for this approach will insert registers to implement the behavior specified by the `delay` operator. This approach provides a convenient abstraction for working with the past values of the input, but prevents the user from explicitly managing state. For example, the output of a circuit could depend on some computation on a window values of the input. In this case, it may be most efficient to store a partial computation on the input value as the state, but the stream-based abstraction forces the user to describe the computation as a function of the delayed input stream values. The compiler is then responsible for discovering the fact that the intermediate computation can be stored as opposed to storing just the stream values and redoing computation.

Term rewriting systems for hardware [32] abstract registers by mapping terms to synchronous storage elements and rewrite rules to combinational logic. This approach shares many similarities to the functional HDLs that encode state in the interface of a function. However, TRS faces a unique challenge because the technique introduces the possibility of conflicts during state updates. Two rules may fire and try to update the same register which means the compiler must be sophisticated enough to detect conflicts and insert arbitration logic when possible. This issue is compounded when considering the modular composition of rules. The compiler could schedule the rules for each module separately, or it could lift the rules into a single top-level module which is then scheduled as a single unit. Prior work has shown that both approaches could be viable depending on the input design [36]. Minimizing the overhead of the compiler generated logic for scheduling rewrite rules is the key challenge for applying term rewriting systems as a register-transfer abstraction.

Devising abstractions for FSMs is another essential design problem for RTL languages. For example, Verilog provides abstractions that enable logic synthesis to generate optimized implementations of FSMs. A key issue for FSM synthesis is choosing the best representation for state. Consider a design where the state of an FSM is being consumed by a circuit performing an arithmetic operation. In this case, using a non-binary state encoding would require the insertion of decode logic. On the other hand, if the target platform is an FPGA, a one-hot state encoding often maps more efficiently to a lookup table architecture. As shown in Figure 2, Verilog designers can use *parameters* to abstract the encoding of the state. Control logic dispatches on the abstract parameters, and the concrete parameter values can be easily changed or selected by an automated tool.

Related to the abstraction of state encoding is the synthesis of control logic for the FSM. The canonical pattern for describing a Verilog FSM, shown in Figure 2, uses a `case` statement that dispatches on a state variable. The semantics of the Verilog `case` statement introduces complexity for the synthesis tool because cases are not necessarily mutually exclusive. Furthermore, the tool must also synthesize logic to handle behavior for cases that have not been listed. For example, consider a binary encoded FSM with 12 states. The state will be stored in a 4-bit quantity that is used to dispatch a `case` statement. Without guidance, the synthesis tool must be sophisticated enough to prove that there are only 12 possible values of the 4-bit quantity, otherwise it must insert extra logic to handle the illegal values. The SystemVerilog language avoids this issue by introducing the `unique` qualifier for indicating that all legal cases have been listed and are mutually exclusive.

One major issue in the canonical design pattern for Verilog FSMs is that for large FSMs with complex transitions, the description exhibits a serious lack of structure. For example, a simple SDRAM controller in Verilog contains 25 states with the entire FSM transition behavior defined in a single `case` statement [21]. Understanding the code requires the reader to follow large jumps between arbitrary cases. This design pattern incurs a significant cognitive load on the designer who must manage a large amount of complex temporal behavior to read the code. There is a direct relation between the use of large case statements to the use of `goto` statements, and we consider this design pattern to be similarly harmful [19].

It is important to remark that this program structuring issue is not restricted to the description of hardware FSMs, but is in fact an instance of a more general problem for imperative languages. Fortunately, the software community has found a promising solution based on coroutines [10]. Recent work is investigating the application of this technique to hardware FSMs by restricting the semantics of a coroutine so it can be precisely compiled to a circuit [60]. The main challenge is restricting the coroutine semantics to be only able to describe FSMs while still enabling the use of coroutine composition.

The essence of this technique is to augment the semantics of the Verilog `always` block to describe a coroutine. The designer may suspend the procedure in arbitrary locations to incorporate more structure in the code. For example, the sequencing of two states can be achieved by separating the logic with a `yield` statement, and the looping of a state can be described using a `while` loop containing a `yield`. Compare this to the `case` statement pattern where the structure of sequences and loops are not explicit in the flat list of cases.

A related issue is the description of the sequential composition of FSMs. Using just the circuit abstraction composed with a basic RTL abstraction, the sequential composition of distinct FSM circuits is achieved by using wires between the two circuits. These wires are used to relay signals indicating that an FSM should start or that an FSM has ended. In order to abstract away these wires and the accompanying control logic, the creators of Lava developed the Pace [13] language. Blarney, a modern variant Lava, provides a similar concept in the form of *Recipes* [44]. The developers of Bluespec also created similar language called STMTFSM [48]. Underlying all these languages is a notion of modular, sequential composition of program fragments. Early work on the Silica language [60] is exploring the use of coroutine composition as another abstraction for modular, sequential composition.

### 3.1.3 Virtual Machine Abstraction

The *virtual machine* abstraction models hardware as a set of instructions for an abstract machine. This technique hides certain details found in lower levels of abstraction such as the finiteness of resources. For example, the C language provides an abstraction over a virtual machine that can store an infinite number of variables. In order to synthesize a hardware implementation of a C program, the compiler must perform a variant of register allocation that maps variables to registers depending on a set of constraints provided by the user. Historically, this approach has been mostly applied to the synthesis of hardware from traditional software languages such as C, but an emerging body of work is exploring the application of this technique to software DSLs.

The major advantage of this approach is that it greatly improves the productivity of the user by enabling them to design hardware as if they were developing software. However, in practice, the user is required to have deep knowledge of the hardware they are trying to generate in order to achieve the desired performance [46]. This negates much of the advantage of using a traditional software language because instead of simply reasoning about a software program, the user must break the virtual machine abstraction and reason about how the program will be mapped to hardware. The central challenge for designing languages based on the virtual machine abstraction is to find what aspects of hardware can be abstracted in order to improve productivity without weakening the performance of the compiler.

The problem of compiling a general software program into hardware shares many of qualities found in the problem of automatically parallelizing a general software program. Fortunately, recent work has mirrored the domain of parallel computing by leveraging DSLs to facilitate better compiler mappings to hardware. DSLs are able to provide productivity and performance by leveraging domain-specific abstractions.

For example, recent work on Halide, a DSL for high performance array and image processing code, extended the compiler to support hardware synthesis by introducing directives for hardware-specific optimizations [53]. This approach maintains the virtual machine abstraction for describing image processing algorithms while giving the user a level control over how the compiler synthesizes hardware. While this technique still requires the user to think about the hardware they are designing, it avoids changing the original source input in order to effect change in the compiler output. The main advantage of this approach is that the user can achieve the desired synthesis results by guiding the compiler rather than relying on optimizations based on heuristics.

The Spatial [38] DSL takes another approach to simplifying the problem by introducing a virtual machine abstraction with an alternative design for memory. Instead of using the uniformly accessible address space abstraction presented by standard CPUs, Spatial programs explicitly interact with the memory hierarchy. This design choice is motivated by the fact that a major challenge for compiling a general software program to hardware is determining an optimal memory architecture. Spatial allows the user to explicitly set a memory architecture using a template while still leveraging the compiler to schedule other aspects of the computation. Spatial is an example of discarding the traditional virtual machine abstraction used by most modern software languages and replacing it with a new abstraction that is tailored to the problem of hardware compilation.

Underlying both the Halide and Spatial approaches is a technique that involves identifying a key problem for the compiler and developing a language abstraction to simplify this problem. Moving forward, researchers interested in developing HDLs based on a virtual machine abstraction should explore techniques that balance the productivity of the user with the quality of the hardware synthesized by the compiler.

## 3.2 Compiler Infrastructure

The proliferation of software languages based on LLVM [40] demonstrates the value of shared compiler infrastructure for both industrial and academic purposes. For researchers, LLVM provides a means for rapidly prototyping languages without having to implement standard compiler passes or create backends for standard architectures. Efforts to develop common infrastructure for hardware compilers are underway and have elucidated key issues when compared to their software compiler counterparts [34, 15]. There is a clear need for the development of hardware-specific compiler passes. Optimization passes are of critical importance because new ideas in HDLs will see no practical use unless they can be compiled into high performance implementations.

While some standard compiler passes, such as constant folding, can be directly applied to HDLs, there exists an entire class of passes that are specific to hardware. As an example, hardware computations are always predicated in the sense that if a computation is mapped to physical components, those components will be continuously executing. Conditional logic is implemented using multiplexers on the flow of data. In order to simplify the lowering of conditional logic, programs can be rewritten into single-static assignment form. In software, leaving it in this form would incur a cost because instructions in a non-traversed branch would always be executed. However, this is already the case for hardware, so leaving the program description in this form incurs no cost. In fact, this simplifies the synthesis stage of the compiler by enabling a one-to-one mapping from phi nodes to multiplexers. The key challenge facing researchers interested in developing HDL compiler infrastructure will be devising reusable, hardware-specific analysis and transformation passes.

Another impediment to design productivity is the development of software compilers that target a novel architecture. For example, developing an extension to the RISC-V ISA [3, 64] would require extending an existing compiler backend to target the new instructions. This means that hardware design teams must include compiler experts, which in turn indicates a need for the ability to automatically synthesize compiler backends for new hardware architectures. The Tensilica processor generator [25] demonstrated the feasibility of automatically generating a compiler that targets new instructions. However, this capability required that the user conform to a fixed processor architecture. Future work should explore extending this technique to support the extension of a broader class of architectures.

Given that many software compilers have converged on the ISA abstraction for backend targets, it is essential that the HDL community converge on an ISA specification language that is machine readable. Convergence would allow researchers to experiment with automatically synthesizing compiler backends for a new ISA described using a standard input format. ISP [7] is an older example of a processor specification language that could describe ISAs. Early stage research on the Peak language [28] is exploring the use of `smt-lib` [9] to develop a modern variant of ISP with formal semantics.

Finally, a critical issue facing hardware compiler developers is performance of the compiler itself. For example, a recent paper [37] touting a new methodology for high productivity hardware design reported that compiling their RTL design to an integrated circuit layout took *only* 12 hours. This is an obvious bottleneck in the design space exploration process. Their technique for reducing the runtime of the compiler was based on reducing the complexity of place and route, a stage in the compiler where logical components of the design are placed into physical space. Optimizing the place and route phase of hardware compilers is just one opportunity for researchers interested in improving the runtime performance of HDL compilers.

## 3.3 Formal Methods

Programming languages have long enjoyed an abundance of elegant theories that form the basis of useful formal methods. In the style of Grothendieck [42], researchers working on foundational theories have created a sea of techniques for developing practical solutions to difficult problems. The recent development and proliferation of WebAssembly [26] demonstrates the utility of designing a new language with formal specification in mind. Rather than face the challenge of retrofitting formal methods to old language designs, researchers in this new golden age of HDLs should leverage the opportunity to develop new languages specifically designed for the application of advanced formal methods.

### 3.3.1 Execution Semantics

Formalizing the execution semantics of an HDL is essential requirement for the integration with formal tools such as model checkers [14]. The major challenge is capturing the intrinsic concurrency and parallelism in hardware. Process calculi [5], specifically with a notion of time [6] present one approach. Real and discrete time could be used to describe the semantics of analog and digital circuits respectively. One issue is the integration of real and discrete time for the modeling of mixed-signal circuits. A similar issue is the modeling of synchronous and asynchronous digital logic. Communicating sequential processes [31] are one technique that have been applied to modeling of asynchronous circuits [62].

The existence of the circuit abstraction as an expressively complete primitive is reminiscent of function abstraction from the domain of software languages. This raises the question as to whether a core calculus can be constructed that captures the execution semantics of the circuit abstraction in the same way that the lambda calculus [8] captures the semantics of function abstraction. While function abstraction presents a compelling basis for the development of this calculus, there are two key issues that must be addressed: circuits can hold state and must have finite size. Contrast this with the basic definition of function abstraction which can be used to describe infinite computation through recursion. A type system can be used to enforce the finiteness of computation [49], which leaves the issue of managing state. Section 2.3 discusses two techniques for encoding state in a functional HDL. One of the key challenges with using these techniques is that it restricts the language to describing synchronous circuits. An essential contribution to the community will be the development of a state encoding mechanism that can capture both synchronous and asynchronous logic.

### 3.3.2 Type Systems

The fact that Verilog is the dominant HDL indicates that the hardware community has not enjoyed the benefits of the latest advances in type systems. The consequences of this is demonstrated by the fact that the ARM Advanced Peripheral Bus (APB) interface [41] uses special prefixes in port names to indicate that they are part of the protocol. This requires users to manage interface connections using name matching, which is considerably less safe than what embedding this in the type system could offer.

One major issue is that the Verilog type system does not provide a concept of algebraic data types. Introducing the concept of a finite size product type would enable the APB interface specification to be defined as a tuple or record type rather than using a naming convention. The use of a product type offers the same benefits to HDL designers as it does to software developers. They are also an example of an *abstraction without overhead* [61] because they can be compiled out of a design by flattening the types into their leaf elements. Compare this to sum types which would require inserting extra logic into the generated design to distinguish between variants. Despite this cost, sum types still provide the same useful static guarantees as they do for software. They also provide a mechanism for abstracting away the details of the control logic, which creates an opportunity for the compiler to synthesize an optimized implementation.

Another interesting avenue of research is the application of behavioral types [2], specifically session types [33], to hardware interfaces. Hardware communication protocols exhibit many of the same characteristics as the software protocols that session types have already been applied to. The core problem will be weaving the session type semantics into HDL execution semantics. Researchers interested in this problem should consider how the domain of hardware protocols differ from more general software protocols, with the intention of finding opportunities to make the problem simpler. One crucial aspect of hardware protocols is the use of *magic* bit patterns to encode portions of the protocol. More generally, hardware protocols can involve data-dependent communication. This reveals an opportunity for the application of dependent type techniques to specify properties on the values of data moving through an interface. While this is a generally difficult problem, restricting the domain to hardware protocols might provide opportunities for practical applications of these ideas.

## 4 Vision: A Multi-Language System for Hardware Construction

A golden age of HDLs presents an opportunity to experiment with alternative HDL designs. This section presents a vision for a multi-language system where a meta-programmed host language is used to implement embedded DSLs for hardware construction. The multi-language approach is directly inspired by Lua/Terra [17, 16, 18], a two-language system that integrates a statically typed, low-level programming language with a dynamically typed, high-level language. Much like software development, hardware design involves components written in multiple languages. For example, a software model of a specific component could be implemented in C and used by a test written in Verilog. Furthermore, the hardware implementation of the module may be defined using Verilog generated by a Perl meta-program.

Rather than treat an HDL as a standalone language like Verilog, this vision adopts the approach of embedding an HDL in a general purpose programming language. This vision is based on a core structural embedded DSL that is used as a common compilation target for a sea of DSLs each targeting various aspects of the hardware design process. Unifying all aspects of the hardware design process into subsets of the same language reduces the cognitive load on the hardware designer. They are only required to learn a single syntax and integration via embedding enables DSLs to be composed without requiring glue code.

### 4.1 Meta-programmed Host Language

The vision of this multi-language system is based on a meta-programmed host language. The extent to which it may be meta-programmed must enable the implementation of rich DSLs. For example, Magma [27] uses Python's metaclass features to embed a circuit abstraction, and Silica [60] inspects the Python AST to compile coroutines to hardware finite-state machines. The implementation of these DSLs require language support for meta-programming that is much richer than simple preprocessing. A side-effect of this requirement is that the meta-programming features of the host language will become meta-programming features for the embedded DSLs.

A standard multi-stage programming approach is sufficient for supporting the flexible code generation required for implementing hardware generators [51]. A hardware *generator* is a program that consumes a set of parameters and produces an instance of a hardware design [54]. Embedding an HDL in a meta-programmed host language enables hardware generators to be implemented using standard meta-programming techniques. Applying the technique of multi-stage programming to hardware generators is somewhat easier than in the software domain because, in practice, the interaction between the meta-language and the HDL is one directional. Compare this to Lua/Terra where control can be transferred between both languages. In hardware generators, meta-programs construct fragments of HDL programs, but the generated HDL code does not typically invoke code in the host language. In theory this might be possible given a reconfigurable hardware system, but in practice this is limited by the slow performance of hardware compilers. Improving compiler performance could enable the construction of JIT compiler systems for hardware, which could then leverage two way interaction between generator code and generated hardware.

One key issue is whether the host language is statically or dynamically typed. The major trade-off is between productivity and correctness. A statically typed host language would provide increased safety, which could be viable for large, complex systems. However, a dynamically typed language would promote rapid design space exploration and provide the flexibility required for more complex hardware generators. For example, generators in dynamically typed languages can employ dynamically constructed types. Another important issue is the cognitive load placed on the user by the type system. Hardware designers are not experts in software engineering and therefore stand to benefit greatly from a type system that is simple and easy to understand. Furthermore, the correctness of the generated hardware design is of greater importance than the correctness of the generators used to construct the design fragments. These requirements suggest that a system based on a dynamically typed host language such as Python composed with a statically typed embedded DSL presents a compelling solution that balances productivity and correctness. A ubiquitous language like Python has the added benefit that many hardware engineers are likely to have already encountered it for scripting purposes. Compare this to a language like Scala; while it provides many compelling language features for building DSLs, it is highly unlikely that a hardware engineer has encountered the language in their schooling or professional work.

## 4.2 The Core Structural DSL

After the host language, the second essential ingredient of the vision is an embedded DSL that provides a structural circuit abstraction. The definition of this core DSL should be simple and precise because many of the complexities of a traditional HDL will be offloaded to other DSLs or the host language. As discussed in Section 3.1.1, a circuit abstraction is expressively complete, which means that this structural DSL can serve as a common compiler target for all other languages in the system. This design enables the structural abstraction to be used to compose modules defined in different DSLs. This is achieved by performing staged execution where in the final stage, all program fragments have been compiled to the core DSL. During this final phase of execution, the user defines a program to structurally compose the various components. Magma [27] and Chisel [4] are concrete examples of embedded structural languages that could serve this purpose. An important requirement is that this core language be formally specified, which then provides a consistent basis for the formal specification of other DSLs in the system.

## 4.3 A Sea of Hardware DSLs

The combination of a host language with an embedded core structural HDL serves as the basis for the research and development of other DSLs to address the intellectual challenges discussed in Section 3. For example, recent work has used Magma [27] and Python as the basis for developing the Peak language [28] for specifying processing elements. Peak supports compilation to Magma, which can then be composed with other components written in other DSLs such as a Silica [60], a DSL for describing hardware finite-state machines using coroutines. A major theme in this design is *separation of concerns through separation of languages*. That is, the description of different hardware components may benefit from being described using a different set of abstractions. If this is the case, these components can be implemented using different DSLs and composed through a well-defined structural interface.

The aforementioned DSLs address issues specific to the design of concrete hardware. This vision includes another class of DSLs that target accelerating specific applications. For example, a DSL based on Numpy [63] could be used to compile numerical computation algorithms into hardware circuits. While these languages should be designed primarily to provide application specific abstractions to the user, they should also be designed to interoperate with the core structural DSL. This would enable code written in application oriented DSLs to be integrated with libraries that generate harnesses for application accelerators. In this case, a library routine could instance the compiled version of the algorithm and wire it up to other components using the core structural DSL.

### 4.4 Verification

Because the host language is a general purpose programming language, it provides the necessary facilities for performing verification tasks. Underlying this will be a connection between general purpose code in the host language and circuits defined in the core structural DSL. Recent work on *fault* [59] has explored solutions to this by developing an embedded DSL that allows users to interact with circuits through a set of actions. A key advantage of this approach is that it enables verification components, such as random number generation, to be implemented as libraries in the host language. Also, the tests can be meta-programmed in the same fashion as the hardware, which reduces verification cost through more flexible testing infrastructure. Compare this approach to SystemVerilog, where the core language for describing hardware was extended with abstractions specifically for verification such as a class system and string data type. Overtime, this has resulted in feature creep and complexity in the SystemVerilog specification. This is another example of the vision's fundamental design pattern based on decoupling features that are not hardware specific from the HDL.

## 5 Conclusion

The PL community stands on the critical path to a new golden age of computer architecture. Fortunately, there is an abundance of intellectual challenges that indicate that we are on the cusp of a new golden age of HDLs. This paper develops a vision for a multi-language system for hardware construction that will provide the productivity gains required to induce this new golden age of computer architecture. This is an exciting time to be a researcher interested in PL and hardware.

## References

1 *A 4 bit up-counter with synchronous active high reset*, 2009 (accessed April 5, 2019). URL: `https://www.edaplayground.com/x/3cf`.

2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.

3 Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

4 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

5 Jos CM Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

6 Josephus Cornelis Maria Baeten and Cornelis Adam Middelburg. Process algebra with timing: real time and discrete time. In *Handbook of process algebra*, pages 627–684. Elsevier, 2001.

7 M Barbacci, C Gordon Bell, and Allen Newell. *ISP: A language to describe instruction sets and other register transfer systems*. Citeseer, 1972.

8 H.P. BARENDREGT. Chapter 1 - Introduction. In H.P. BARENDREGT, editor, *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*, pages 3–21. Elsevier, 1984. doi:10.1016/B978-0-444-87508-2.50009-5.

9 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

10 Eli Bendersky. Co-routines as an alternative to state machines. https://eli.thegreenplace.net/2009/08/29/co-routines-as-an-alternative-to-state-machines, 2009.

11 Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, volume 34(1), pages 174–184. ACM, 1998.

12 Inc. Bluespec. *BSV 101: DESIGNING A COUNTER Using the Bluespec Development Workstation*, 2009 (accessed April 5, 2019). URL: http://wiki.bluespec.com/Home/Getting-Started/Tutorials.

13 K Claessen and M Sheeran. A slightly revised tutorial on lava: A hardware description and verification system, 2007.

14 Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

15 Ross Daly, Lenny Truong, and Pat Hanrahn. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Proceedings of the 1st Workshop on Open-Source EDA Technology*, 2018.

16 Zachary DeVito. *Terra: Simplifying High-performance Programming Using Multi-stage Programming*. PhD thesis, Stanford University, 2014.

17 Zachary DeVito and Pat Hanrahan. The Design of Terra: Harnessing the best features of high-level and low-level languages. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

18 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48(6), pages 105–116. ACM, 2013.

19 Edgar Dijkstra. *Edgar Dijkstra: Go To Statement Considered Harmful*, 1968 (accessed February 6, 2019). URL: https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf.

20 Carl Ebeling. *CSE370 - XV - Verilog for Finite State Machines (Spring 2010)*, 2010 (accessed April 5, 2019). URL: https://courses.cs.washington.edu/courses/cse370/10sp/pdfs/lectures/15-VerilogIIPrint.pdf.

21 Mike Field. *Simple SDRAM Controller (Verilog Memory controller v0.1)*, 2014 (accessed April 5, 2019). URL: http://hamsterworks.co.nz/mediawiki/index.php/File:Verilog_Memory_controller_v0.1.zip.

22 Frank Ghenassia et al. *Transaction-level modeling with SystemC*, volume 2. Springer, 2005.

23 Steve Golson and Leah Clark. Language Wars in the 21st Century: Verilog versus VHDL–Revisited. In *Synopsys Users Group (SNUG)*, 2016.

24 Steve Golson, Gardner Hendrie, and Philip Moorby. *Moorby, Phil (Philip Raymond) oral history*, 2013 (accessed April 5, 2019). URL: `https://www.computerhistory.org/collections/catalog/102746653`.

25 Ricardo E Gonzalez. Xtensa: A configurable and extensible processor. *IEEE micro*, 20(2):60–70, 2000.

26 Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, volume 52(6), pages 185–200. ACM, 2017.

27 Pat Hanrahan. *magma*, 2019 (accessed February 6, 2019). URL: `https://github.com/phanrahan/magma`.

28 Pat Hanrahan. *peak*, 2019 (accessed February 6, 2019). URL: `https://github.com/phanrahan/peak`.

29 John Hennessy and David Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–29, June 2018. `doi:10.1109/ISCA.2018.00011`.

30 John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

31 Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

32 James C Hoe et al. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, pages 595–619. Springer, 2000.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

34 Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017.

35 Steven Dexter Johnson. *Synthesis of digital designs from recursion equations*. PhD thesis, Indiana University, 1983.

36 Michal Karczmarek et al. *Synthesis of multi-cycle circuits from guarded atomic actions*. PhD thesis, Massachusetts Institute of Technology, 2011.

37 Brucek Khailany, Evgeni Khmer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer. A Modular Digital VLSI Flow for High-productivity SoC Design. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 72:1–72:6, New York, NY, USA, 2018. ACM. `doi:10.1145/3195970.3199846`.

38 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311. ACM, 2018.

39 M. Kooijman. Haskell as a higher order structural hardware description language, December 2009. URL: `http://essay.utwente.nl/59381/`.

40 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

41 Arm Limited. *Arm AMBA (Advanced Microcontroller Bus Architecture) Protocols*, 2019 (accessed February 6, 2019). URL: `https://developer.arm.com/products/architecture/system-architectures/amba`.

42 Colin McLarty. The Rising Sea: Grothendieck on simplicity and generality, 2007.

43 G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955. `doi:10.1002/j.1538-7305.1955.tb03788.x`.

44 mn416. *Blarney – Example 7: Recipes*, 2019 (accessed April 6, 2019). URL: `https://github.com/mn416/blarney#example-7-recipes`.

45 Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

46 Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

47 Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

48 Rishiyur S Nikhil and Kathy R Czeck. *BSV by Example.* Createspace Independent Publishing Platform, 2010.

49 Russell O'Connor. Simplicity: a new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120. ACM, 2017.

50 John O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. *The Fusion of Hardware Design and Verification*, pages 309–328, 1988.

51 Gordon J Pace and Christian Tabone. Multi-Stage Languages in Hardware Design, 2008.

52 Gordon D Plotkin. A structural approach to operational semantics, 1981.

53 Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):26, 2017.

54 Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, et al. Rethinking digital design: Why design must change. *IEEE micro*, 30(6):9–24, 2010.

55 Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM, 1984.

56 Satnam Singh and Philip James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 91–100. IEEE, 2001.

57 IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, February 2018. `doi:10.1109/IEEESTD.2018.8299595`.

58 Paul Teehan, Mark Greenstreet, and Guy Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Computers*, 24(5), 2007.

59 Lenny Truong. *fautl*, 2019 (accessed February 6, 2019). URL: `https://github.com/leonardt/fault`.

60 Lenny Truong. *silica*, 2019 (accessed February 6, 2019). URL: `https://github.com/leonardt/silica`.

61 Aaron Turon. *Abstraction without overhead: traits in Rust*, 2015 (accessed April 5, 2019). URL: `https://blog.rust-lang.org/2015/05/11/traits.html`.

62 Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the conference on European design automation*, pages 384–389. IEEE Computer Society Press, 1991.

63 S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

64 Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

65 Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.