Abstraction in Hardware System Design

Applying lessons from software languages to hardware languages using Bluespec SystemVerilog

Rishiyur S. Nikhil, Bluespec Inc., USA

The history of software engineering is one of continuing development of abstraction mechanisms designed to tackle ever-increasing complexity. Hardware design, however, is not as current. For example, the two most commonly used HDLs (hardware description languages)—Verilog and VHDL^{12,9}—date back to the 1980s. Updates to the standards lag behind modern programming languages in structural abstractions such as types, encapsulation, and parameterization. Their behavioral semantics lag even further. They are specified in terms of event-driven simulators running on uniprocessor von Neumann machines (and this is true even for their recent descendents, SystemVerilog and SystemC^{10,11}).

These HDLs all have “synthesizable subsets” that constrain them in an effort to narrow this behavioral gap, but the mismatch is never completely eliminated. The strain is beginning to show as hardware chip capacity has grown exponentially according to Moore’s law and we are called upon to design entire SoCs (systems-on-a-chip) of astonishing diversity and complexity.

Another important issue is that verification (testing) has so far been done using simulation, but this is decreasingly practical.³ In modern SoCs, the hardware is large and complex, and it runs heavy software loads such as full-featured operating systems and applications. Verification involves simulating all of these together, and it is not unusual for software simulations to run for days or weeks. Simulation speeds are typically cited in the 10s to 100s of KHz, whereas what are needed are MHz speeds.

The only feasible solution to this problem is what hardware people call *emulation*, which is simulation of the hardware design on FPGAs (field-programmable gate arrays). This FPGA-based emulation cannot be left to final drafts of a design, however; it must begin at the very start of the design process, from early models onward. Doing so places further requirements on the behavioral semantics of HDLs.

Simple emulation isn’t enough, however. Previously it might have been acceptable to use separate high-level languages for models and test benches that run only in software simulation, such as SystemC TLM (Transaction Level Modeling)¹⁹ and SystemVerilog VMM (Verification Methodology Manual).² Today, however, what is needed is an HDL that can be used for all these purposes—from early models and test benches to detailed implementation—where these different components can be mixed freely and where *all* these components can be synthesized for FPGA simulation. Thus, the HDL needs to be universal, both in the sense that it is suitable for all kinds of digital designs (for example, not just signal processing) and in the sense that it should be fully synthesizable, whether used for models, test benches, or implementations.

These requirements mean a radical reconsideration of HDLs. This article describes BSV (Bluespec SystemVerilog),¹⁸ the design of which was motivated by just such a reconsideration while reusing features from SystemVerilog wherever possible. BSV’s structural extensions (involving more

expressive types, overloading, encapsulation, and parameterization) are inspired by Haskell,²⁰ the functional programming language. Its behavioral semantics are inspired by Term Rewriting Systems¹³ (or Guarded Atomic Actions), which are best suited for describing complex, concurrent hardware. This is more than a theoretical exercise—BSV and its tools have been used in industry for more than five years (your smartphone or tablet today may well contain components designed with BSV).

WHY NOT USE AN EXISTING SOFTWARE LANGUAGE FOR HARDWARE DESIGN?

Before looking at BSV, it is useful to consider whether a new language is necessary at all. Although from a functional point of view “it’s all just computation,” hardware system design is characterized by features very different from software design.

CONCURRENCY/PARALLELISM

Hardware systems typically have parallelism that is massive, fine-grain, heterogeneous, and reactive. (Unlike some authors, I make no distinction between the terms *concurrency* and *parallelism*). This parallelism results in increased speed, which is often the main reason to implement something in hardware. *Massive* in this context means that the number of parallel activities may number in the thousands or even millions. *Fine-grain* means that parallel activities interact frequently (measured in time scales of clock cycles) over possibly very small shared resources (such as individual registers of a few bits). *Heterogeneous* means that parallel activities involve diverse tasks—in contrast, for example, to SIMD (single instruction, multiple data) or SPMD (single process, multiple data) computation. Finally, *reactive* means that parallel activities are often triggered by asynchronous events, such as unpredictable data arrival, interrupts, arbitration resolution, and I/O.

Unfortunately, most software languages are not parallel at all, but instead are entirely sequential. Some extensions for massive parallelism address only SIMD parallelism. Thread extensions can handle heterogeneity, but they are notoriously difficult for massive, fine-grain, or reactive parallelism.^{1,22,15}

ARCHITECTURE AND ALGORITHMS

In hardware system design, good architecture (with its attendant cost model) is a central design goal and an outcome of design activity, whereas software is mostly designed for a fixed, given input architecture (typically von Neumann, perhaps with extensions such as SIMD), as illustrated in figure 1. Thus, in hardware design, algorithm and architecture are inseparable, and it is meaningless to talk about “pure algorithm design” in the abstract, without some architectural model that gives it a concrete cost metric.

Since they are all Turing-complete, any architecture can be modeled or simulated with existing programming languages, but there are two fundamental problems with this. First, it takes extraordinary discipline (and therefore a lot of time and effort) to model a complex architecture accurately. Second, modeling an architecture often means losing orders of magnitude in execution speed, both because of the extra layer of interpretation and because the natural parallelism of the architecture being modeled is essentially completely discarded.³ DSLs (domain-specific languages) can address the first issue, but not the second. For most software programming languages, the “native” architectural model is the von Neumann model (one sequential process, with constant-time access to a large, flat memory), and it is only native von Neumann algorithms that execute at speed.

The term *architectural transparency* expresses the idea that the source program directly reflects the

FIGURE 1 Hardware vs. Software Design

desired architecture. Abstraction mechanisms can hide detail but should not distort the resulting architecture. This property is essential for the programmer/designer, for whom abstraction is good, provided that it does not compromise predictability and control. This property is also essential for compilers (synthesis) to produce efficient implementations.

THE INADEQUACY OF SOFTWARE SIMULATION

As mentioned, today's SoCs need FPGA-based emulation to achieve acceptable simulation speed, and this requires universal applicability of HDL (models, test benches, and implementations of the full gamut of SoC components) and universal synthesizability. Unfortunately, no software languages are suitable for this.²

Many in the industry advocate using a combination of C++ and SystemC—the former to describe the algorithmic content of individual IP (intellectual property) blocks, and the latter to describe system-level communication, hierarchy, and integration of these IP blocks into an SoC. The C++ inside IP blocks can then be subjected to so-called HLS (high-level synthesis⁵), using tools that automatically generate parallel hardware from sequential C++, given declarative objectives such as latency, throughput, area, power, and target silicon technology. A detailed critique would require another article, but this section and Stephen A. Edward's article on the topic⁷ provide some background.

BSV (BLUESPEC SYSTEMVERILOG)

Let's focus first on the computation model of BSV (i.e., its behavioral semantics), because that is the greatest limitation of existing software languages for hardware design. Then we present an example to demonstrate the use of modern structural abstraction mechanisms.

RULES: THE BASIC COMPUTATION MODEL

Verilog and VHDL, with their origins as simulation languages, are built on the uniprocessor von Neumann model: sequential processes, stack-based procedure calls, traditional stack-based updatable variables, and cooperative multitasking. BSV, on the other hand, is born out of direct hardware description—state and parallel state transitions. The computer science literature has a computation model that is well suited for this, variously called Term Rewriting Systems,¹³ Guarded Atomic Actions, or Rewrite Rules. It is used in many formal specification languages for complex concurrent systems, including Dijkstra’s Guarded Commands,⁶ Chandy and Misra’s UNITY,⁴ Lamport’s TLA+,¹⁴ Abrial’s Event-B,¹⁶ and many more. The following BSV excerpt illustrates a computation to sort four integers in ascending order.

```
Reg #(int) x1 <- mkRegU;
Reg #(int) x2 <- mkRegU;
Reg #(int) x3 <- mkRegU;
Reg #(int) x4 <- mkRegU;
```

```
rule swap12 (x1 > x2);
  x1 <= x2;
  x2 <= x1;
endrule
```

```
rule swap23 (x2 > x3);
  x2 <= x3;
  x3 <= x2;
endrule
```

```
rule swap34 (x3 > x4);
  x3 <= x4;
  x4 <= x3;
endrule
```

The first few lines instantiate four registers (storage elements) named `x1 ... x4`, containing values of type `int`. The `mkRegU` right-hand sides are the constructors, whose details are not important here.

The rule (or Guarded Atomic Action) `swap12` is like a reactive process (i.e., it can “fire” whenever its Boolean condition `x1>x2` is true). The body of a rule is an *Action* and is semantically an *instantaneous* event. Here it is composed of two smaller Actions: one that assigns the value in `x1` into `x2`, and vice versa. The net effect is to swap the values in the two registers. Rules `swap23` and `swap34` are similar.

There is no inherent sequencing between rules—a rule can fire whenever its condition is true. Thus, in the example, it may happen that rules `swap12` and `swap34` perform their swaps in parallel. But what about rules that act on shared state, such as `swap12` and `swap23`, both of which update `x2`? Rules have the semantics of *atomic transactions*, and thus, in this case, those two rules can fire only in some order (i.e., sequentially). For this example, it does not matter which of the two possible orderings is chosen; the registers will eventually be sorted anyway.

The previous fragment seems repetitive—four similar registers and three similar rules—but we wrote it that way first to explain the semantics. It would more likely be written this way:

```
Vector #(4, Reg #(int)) <- replicateM (mkRegU);
```

```
for (int j = 1; j < 4; j = j + 1)
  rule swap_neighbors (x[j-1] > x[j]);
    x[j-1] <= x[j];
    x[j]   <= x[j-1];
  endrule
```

This expands (“statically elaborates,” in hardware language jargon) to what is essentially the same fragment as before. Of course, the 4 here could be abstracted into a parameter. The fragment represents the classic bubble-sort algorithm of introductory programming texts—with the same $O(N^2)$ worst-case complexity—except that here the “bubbling” can happen in parallel, modulo atomicity (i.e., any pair of enabled rules that do not conflict on a shared register can fire in parallel).

When atomicity requires two enabled conflicting rules to be sequenced, the order chosen can be left unspecified. This is at first alarming to hardware designers, where nondeterminism is equated with unpredictable (i.e., bad) results. It is usually welcomed for formal specifications, however, where insisting on a schedule too early is regarded as an unnecessary over-specification. In BSV various techniques are available to nail down particular scheduling choices whenever necessary.

WHAT RULES MEAN FOR HARDWARE, AND WHY THEY MATTER

All hardware design involves specifying parallel activities that update state. The classic style in Verilog (all of these remarks apply to VHDL as well) is state-centric. For each state element x :

```
always @ (posedge CLK)
  if (...cond1...)
    x <= ... value1 ...
  else if (...cond2...)
    x <= ... value2 ...
  else if ...
    ...
  else
    x <= ... valueN ...
```

More generally, multiple state elements may be updated in each conditional arm, the conditional could be written as a case statement, etc. In synthesizable code, however, each state element may be updated in only one always block and may be updated at most once in each conditional arm.

This is how the problem of parallel updates to a shared resource is solved in (synthesizable) Verilog: it is updated in only one “process” (always block) and, for every clock, exactly *one* possible new value for x is specified. It is almost a direct, explicit specification of the hardware that is generated: the conditional represents a *multiplexer* on the input of the x register; the conditions

specify how one of the multiplexer inputs is selected to be clocked into the register; and the conditions may also specify whether the register is updated at all. Collectively, all this logic is referred to as *control logic*.

This organization of behavior in terms of states is, unfortunately, orthogonal to how behavior is typically conceptualized: as a collection of steps, or *transactions*. Each step of behavior may involve updates (perhaps conditionally) to multiple state elements. This “transpose” from the transaction-centric dimension to the state-centric dimension is at the heart of the problem with Verilog. The state-centric view does not scale well as the number of parallel activities increases and becomes more complex in regard to how the activities compete for shared state. Consider the problem in figure 2, which illustrates two registers *x* and *y* updated by three parallel processes under certain conditions. Assuming that process 0 has lower priority than process 1 for the update of *x*, and process 1 has lower priority than process 2 for the update of *y*, the Verilog solution may look like this:

```
always @ (posedge CLK) begin
  if (cond2)
    y <= y - 1;
  else if (cond1) begin
    y <= y + 1;
    x <= x - 1;
  end

  if (cond0 && !cond1)
    x <= x + 1;
end
```

This Verilog code could be written in many styles, but they are all state-centric. The “transpose” from problem statement into state-centric code has intertwined and obscured the original transactions. Also, the priorities are wired into the code structure—the priority of process 2 over

FIGURE 2 Two Registers Updated by Three Parallel Processes

process 1 is expressed in the sequentiality of `if ... else`. The priority of process 1 over process 0 is expressed in the `!cond1` term. In fact, a clever designer may recognize that when `cond2` is true, process 1 cannot update `x`, thus opening an opportunity for process 0 to do so. Therefore, the designer might “improve” the last two lines to:

```
if (cond0 && (!cond1 || cond2))
  x <= x + 1;
```

Note the transitivity of control: Process 0’s update of `x` is affected by the condition of the nonadjacent Process 2. Third, some of the process conditions are duplicated in multiple clauses, with the usual risks of incorrect duplication.

Imagine a change to the spec (this happens all too frequently in the real world) requesting a different priority among the three processes. Or imagine adding another process, or possibly another shared state variable, or both, with new control conditions. The whole code would need to be revised; it’s hard to make local incremental changes.

The fact that even such a small example carries such subtleties in control logic should sensitize the reader to the fact that reasoning about parallel updates in a state-centric manner on a per-clock basis can get very tricky while scaling to more parallel activities and shared state elements, and scaling the complexity of the update conditions.

The BSV code for this example looks like this:

```
rule proc0 (cond0);
  x <= x + 1;
endrule
```

```
rule proc1 (cond1);
  y <= y + 1;
  x <= x - 1;
endrule
```

```
rule proc2 (cond2);
  y <= y - 1;
endrule
```

```
(* descending_urgency = "proc2, proc1, proc0" *)
```

The rules are a direct expression of the process descriptions (i.e., they are transaction-centric, not state-centric), and the final line expresses the scheduling priority. Synthesis from this code produces essentially the same Verilog code shown earlier—namely, the complex control logic necessary to manage parallel update to shared state. It is easy to add more processes, or more shared state, or more complex update conditions.

Rules matter even more when scaling up to systems organized into modules; we will return to this discussion after describing modularity mechanisms.

ORGANIZING RULES INTO MODULES

In object-oriented programming, complex systems are organized into objects. The key behavioral construct is a *process* or *thread* (just one in a sequential language). A process is rooted in some module (perhaps “main”), but it may span object boundaries through method calls. Because methods may themselves call methods, a process may thus access an unbounded number of objects. Each method is semantically a fragment of a process.

Similarly, in BSV a syntactic rule in one module can invoke a method in another module. Methods can, in turn, invoke other methods. Each method is semantically a fragment of a Rule. Thus, a method is more than just a procedure—it is a guarded expression. A semantic Rule is composed of syntactic components: a rule construct and (transitively) all the methods it invokes; this semantic Rule is the unit of parallelism and atomicity.

A small module that implements a FIFO containing integers illustrates this. First its interface declaration:

```
interface FIFOint;
  method Action   enq    (int x);
  method int      first  ();
  method Action   deq    ();
endinterface
```

Like a C++ virtual class, it merely describes the externally visible methods of a module. The `enq` method enqueues an item `x`. Its `Action` type indicates that it returns no value and just has an effect. The `first` method takes no arguments and returns the value of the element at the head of the queue. The `deq` method is a pure `Action`—it has no arguments or results; it just has the effect of discarding the first element in the queue.

Here is the code for one possible module that implements this interface:

```
module mkFIFOint (FIFOint);
  Reg #(int) data <- mkRegU;
  Reg #(Bool) empty <- mkReg (True);

  method Action enq (int x) if (empty);
    data <= x; empty <= False;
  endmethod

  method int first () if (! empty);
    return data;
  endmethod

  method Action deq () if (! empty);
    empty <= True;
  endmethod
endmodule
```


FIGURE 3

Butterfly Crossbar Switch

This is a module *constructor* that can be *instantiated* multiple times to create actual modules (hence the stylistic choice of `mk` in the module name `mkFIFOint`, suggesting the word *make*). `FIFOint` in the header specifies the interface. The next two lines instantiate two registers: `data`, containing an integer, initially unspecified; and `empty` containing a Boolean, initially `True`. The `enq` method is guarded by the condition `if (empty)`—any rule from which it is invoked will be enabled only if the guard is true. When invoked, it stores its argument `x` in the `data` register and sets `empty` to `False`. The `first` and `deq` methods are guarded by the condition `if (!empty)`—any rule from which they are invoked will be enabled only if this guard is true. The `first` method returns `data`'s value and does not change the FIFO. The `deq` method has the effect of setting `empty` to `True`.

This example is simple, but it could be generalized to an `N`-element FIFO by extending `data` to a vector of registers and maintaining the usual head and tail pointers. The FIFO could also be generic (polymorphic) in its content type, instead of just containing `int` values.

Many modules have FIFO-like interfaces—that is, flow-controlled ports that accept or yield values. We can define polymorphic interfaces for this:

```
interface FIFOin #(type t);
  method Action enq (t x);
endinterface
```

```
interface FIFOout #(type t);
  method t first ();
  method Action deq ();
endinterface
```

Other than in the syntactic details, these capabilities are similar to what is seen in object-oriented languages. The key difference is that while OO methods are fragments of sequential processes, BSV methods are fragments of rules and have guards.

A LARGER EXAMPLE

Figure 3 illustrates a Butterfly crossbar switch, which is a hardware device with N input ports and N output ports (here, $N=4$). Packets enter at input ports, and each is routed to an output port. Note that the switch has a recursive structure: the 4x4 switch shown contains, as components, two 2x2 switches, and each of those in turn contains two 1x1 switches (buffers). Two of these 4x4 switches, in turn, could be used to make an 8x8 switch, and so on. Also note that there are three basic components: FIFOs (1 in, 1 out), merge boxes (2 in, 1 out), and routing logic (1 in, 2 out). Here is the interface type declaration:

```
interface XBar #(type packet_t);
  interface List #(FIFOin #(packet_t))  input_ports;
  interface List #(FIFOout #(packet_t)) output_ports;
endinterface
```

It is generic, or polymorphic, in the type `packet_t` of packets flowing through the switch. It is nested, or hierarchical, in that it is defined in terms of two sub-interfaces: `input_ports` and `output_ports`. These, in turn, use standard `List` data structures to refer to collections of `FIFOin` and `FIFOout` interfaces. Here is a module `mkXBar` implementing this interface:

```
module mkXBar #(Integer n,                                // (P1)
  Function UInt #(32) destinationOf (t x),                // (P2)
  Module #(Merge2x1 #(t)) mkMerge2x1                     // (P3)
  (XBar #(t) );
  List #(Put #(t)) iports; List #(Get #(t)) oports;

  if (n == 1) begin
    FIFO #(t) f <- mkFIFO;
    iports = cons(toFIFOin(f), Nil);
    oports = cons(toFIFOout(f), Nil);
  end
  else begin
    XBar #(t) upper <- mkXBar(n/2, destinationOf, mkMerge2x1);
    XBar #(t) lower <- mkXBar(n/2, destinationOf, mkMerge2x1);
```

```

iports = append (upper.input_ports, lower.input_ports);
let midports = append (upper.output_ports, lower.output_ports);

List#(Merge2x1) merges <- replicateM (n, mkMerge2x1);
oport = map (oport_of, merges);

for (Integer j = 0; j < n; j = j + 1)
  rule route;
    let x = midports [j].first();
    let jDest = computeRoute (destinationOf (x), j, n);
    if (jDest == j) merges [j] .iport0.enq (x);
    else merges [jDest].iport1.enq (x);
    midports [j].deq();
  endrule
end
interface input_ports = iports;
interface output_ports = oport;
endmodule: mkXBar

```

P1-P3 are parameters: n requests an $n \times n$ switch; `destinationOf` is a function from packets (of type `t`) to destinations (of type `UInt #(32)`, unsigned 32b integers) that can be used to take routing decisions; and `mkMerge2x1` is a constructor for the 2-input, 1-output merge modules. In the semantics, a module constructor is just a function from parameters to modules. Further, it is a higher-order function—that is, its parameters may themselves be functions and module constructors. These features are standard in advanced programming languages such as Haskell²⁰ and SML (Standard ML)¹⁷ but are very unusual in HDLs, particularly in *synthesizable* HDLs.

The bulk of the module is a recursive definition of the module. The `then` part of the big conditional is the base case ($n==1$). A 1×1 switch is implemented using `mkFIFO`. The next line uses functions `toFIFOin` and `toFIFOout` (easy; not shown) to separate the FIFO interface into `FIFOin` and `FIFOout` interfaces, and builds them into singleton lists `iports` and `oport`.

The `else` part is the recursive case ($n>1$). The code first recursively instantiates `mkXBar` twice, at size $n/2$, for the upper and lower 2×2 subswitches. It appends their `input_ports` and `output_ports` and holds them in the lists `iports` and `midports`, respectively. It then instantiates the vertical column of n 2×1 merge blocks at the right edge of the switch shown in figure 3 using the library higher-order function `replicateM` on parameter `mkMerge2x1`. It uses the standard library list-processing higher-order function `map` with the `oport_of` function (not shown; simply returns the output port of a `mkMerge2x1` module) to create the `oport` list.

The `for` loop generates n instances of the rule. In any one instance, the name `x` refers to the packet at the head of the j^{th} port in `midports`. The function `computeRoute` (not shown; a simple numeric calculation) is applied to get the index of the merge module into which `x` should be forwarded, which is done in the small conditional. The rule also dequeues the packet.

Each rule expressing the switch's behavior hides much control detail:

- The rule is enabled only when a packet is available on the rule's upstream FIFO (because of guards on the `first` and `deq` methods).
- The rule is enabled only when a packet can be sent into its downstream merge block (because of the guard on `enq`). For example, it may be blocked as a result of flow control.
- This last control condition is further nuanced by the fact that the downstream merge block is dynamically selected, depending on the packet, and, therefore, only the `enq` guard of the selected block matters.

An even subtler control condition arises out of conflict between two rules. In the `for` loop, pairs of rules enqueue into the same merge blocks. This contention may require one of the competing rules to be stalled (i.e., not enabled) to satisfy atomicity.

WHY RULES MATTER (CONTD.)

In Verilog, all of the above control considerations manifest themselves as explicit, visible control wires on the physical input and output ports of the merge modules, together with a protocol on how to use those wires. These are specified explicitly by the designer, and require effort on a per-module basis. They are *ad hoc* in that these control interfaces and protocols vary from module to module, from designer to designer, and even from day to day for the same designer. The protocol (semantics) is usually conveyed in accompanying text and timing diagrams and, as you might imagine, is often incomplete, ambiguous, and sometimes wrong. When working with rules, the compiler handles this control complexity naturally, automatically, and formally.

This automation is especially important in accommodating change (for fixing a bug, reacting to changing specs, etc.). The `mkMerge2x1` module is a formal parameter, a black box. One actual parameter may permit both `enq` ports to be activated simultaneously, whereas another may not, perhaps because one has separate internal buffering, while the other shares internal buffers. This affects whether competing rules in `mkXBar` can fire simultaneously or not (this is another example of the transitivity of control). With *ad hoc* control logic (as in register-transfer level, or RTL), it may not be able to accommodate such a change without redesigning `mkXBar`, whereas with rules, the compiler handles it.

Consider this analogy: in compiling software, imagine doing register allocation by hand, and designing and documenting an *ad hoc* calling convention for every function you write. It is rather difficult and painful doing it even once, but it's much worse if the source code changes because register allocation has a transitive effect across function boundaries, such as control logic across hardware module boundaries. Just adding an argument to a function may require redoing register allocation for both the callers and the callees, and this may have a ripple effect across multiple functions. Fortunately, compilers automate this, and adding an argument to a function is trivial for the programmer. Similarly, Rules simplify writing source code and modifying it; the compiler (synthesis) figures out the required changes in control logic.

Rules are part of a good DSL for hardware description because they are like state machine transitions, familiar and intuitive to hardware engineers. Rules, however, are more profoundly powerful because they are parallel and atomic.

SYNTHESIS INTO HARDWARE

MAPPING RULES INTO CLOCKED HARDWARE

Historically, rules in programming and specification languages were concerned just with functionality and not performance; there is only an abstract notion of time in the sense that one rule may fire before another. In hardware design, performance is usually a central requirement, not merely an implementation detail; so the computation model must make this visible to the designer.

Space does not permit a full description here, but BSV indeed defines such a concrete mapping of rules into clocked hardware. Clocks and Resets are abstract data types, and rules and methods are mapped into clock and reset domains. Strong static type checking verifies the isolation of these domains. In summary, BSV has easy-to-use and semantically rigorous facilities for the robust creation of designs with multiple clock domains.

Another very important consideration today is power consumption. At a fine granularity, BSV clocks can be gated, with gating conditions integrated organically into rule conditions. At the coarser granularity of IP blocks, BSV has power-management facilities to switch off or scale power and clocks to entire modules or subsystems in a disciplined manner.

One intriguing thought is that BSV could also be compiled into asynchronous logic (which has many potential advantages relating to circuit timing and power consumption), but how to do this, and how to reason about system performance in this regime, are still open research questions.

SYNTHESIS QUALITY FOR ASICS

Technology to compile Rules into efficient hardware was first developed more than a decade ago.⁸ It has been improved continuously since then and has been used on hundreds of designs. In some there was an existing RTL design for apples-to-apples comparisons, and in general, quality has been comparable (as measured in silicon area and performance), typically within 5 percent.

In a few cases, BSV-generated RTL has significantly better quality (15-25 percent) than hand-coded RTL. This is surprising at first because in software you typically pay a performance penalty for higher abstraction. As illustrated in figure 1, however, higher abstraction in hardware design can yield significant *algorithmic* advantages (i.e., more efficient architectures).

SYNTHESIS FOR FPGA-BASED SIMULATION

The universal applicability and synthesizability of BSV has opened the door for many to use FPGAs as their standard simulation engine, from the earliest stages of design, and with a “design-by-refinement” methodology.

Bluespec Inc., by “eating its own dog food,” has used BSV to create highly parameterized communication, debugging, and IP libraries for FPGA boards. These form a kind of operating system for FPGAs, so that the user has high-level facilities for communicating with a host and debugging a model or design instead of the customary painful wrestling with the raw FPGA board hardware.

Where users previously might have written test generators or analyzers or models in C++, they can now write them in BSV without loss of abstraction but with the ability now also to synthesize them and run them on the FPGA adjacent to their designs.

As a result, given that boards with significant FPGA capacity can now be purchased for less than \$2,000 (such as Xilinx ML605), with comfortable high-level abstractions for communications,

debugging, and libraries in BSV, designers can now treat FPGAs as another computation weapon in their arsenal, just like GPGPUs (general-purpose computing on graphics processing units).²¹

CONCLUSION

Historically there has been an almost total separation between software and hardware designers, much of it attributable to the wide cultural (semantic) gap between the languages they use to design their respective systems. Modern systems demand a reduction or elimination of this specialization. All interesting hardware systems today must run sophisticated software, and many complex software computations must move to hardware to meet performance and power consumption goals.²¹

BSV is an attempt to address this problem. Rather than trying to force fit solutions for von Neumann machines (such as C++ and threads) into hardware description, BSV has taken excellent ideas from software that are naturals for hardware description. It describes behavior using Rules (from Term Rewriting Systems), which are excellent for massive, fine-grain, heterogeneous, reactive concurrency (hardware!). It describes structure and structural abstraction using types, overloading, higher-order functions, parameterization, and even monads from Haskell. These ideas have been tested in the field for well over five years and used in finished products, one of which might be in your pocket right now.

REFERENCES

1. Adve, S. 2010. Data races are evil with no exceptions. *Communications of the ACM* 53(11): 84.
2. Bergeron, J., Cerny, E., Hunter, A., Nightingale, A. 2006. *Verification Methodology Manual for SystemVerilog*. Springer.
3. Burger, D., Emer, J., Hoe, J. C., Chiou, D., Sendag, R., Yi, J. J. 2010. The future of architectural simulation. *IEEE Micro* (May/June): 8-18.
4. Chandy, K., Misra, J. 1988. *Parallel Program Design: A Foundation*. Addison Wesley.
5. Coussy, P., and A. Morawiec, eds. 2008. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer.
6. Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8): 453-457.
7. Edwards, S. A. 2006. The challenge of synthesizing hardware from C-like languages. *IEEE Design and Test of Computers* 23(5).
8. Hoe, J. C. 2000. Operation-centric hardware description and synthesis. Ph.D. thesis, MIT.
9. IEEE. 2002. IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993.
10. IEEE. 2005. IEEE Standard for System Verilog—Unified Hardware Design, Specification and Verification Language. IEEE Std 1800-2005.
11. IEEE. 2005. IEEE Standard SystemC Language Reference Manual. IEEE Std 1666-2005.
12. IEEE. 2005. IEEE Standard Verilog Hardware Description Language. IEEE Std 1364-2005.
13. Klop, J. 1992. *Term Rewriting Systems*, vol. 2. Oxford University Press, 1-116.
14. Lamport, L. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional (Pearson Education).
15. Lee, E. A. 2006. The problem with threads. *IEEE Computer* 39(5): 33-42.
16. Metayer, C., Abrial, J.-R., Voisin, L. 2005. The Event-B Language <http://rodin.cs.ncl.ac.uk/deliverables.htm>.

17. Milner, R., Tofte, M., Harper, R., MacQueen, D. 1997. *The Definition of Standard ML* (revised). Cambridge, MA: MIT Press.
18. Nikhil, R. S., Czeck, K. R. 2010. *BSV by Example*. CreateSpace.
19. OSCI. 2009. OSCI TLM-2.0 Language Reference Manual; www.systemc.org.
20. Peyton Jones, S., ed. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge: Cambridge University Press; www.haskell.org.
21. Singh. S. 2011. Computing without processors. *Communications of the ACM* 54(8): 46-54.
22. Williams, R. 2010. Photoshop scalability: Keeping it simple. *Communications of the ACM* 53(10): 36.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

RISHIYUR NIKHIL (nikhil@bluespec.com) worked on Haskell and functional programming languages and compilers for parallelism, and dataflow and multithreaded architectures for about 20 years. Since 2000 he has been applying those ideas to hardware design.

© 2011 ACM 1542-7730/11/0800 \$10.00