Developing algorithmic designs using Bluespec

Document version 2007-Oct-22

Introduction

Modern SoCs often contain IP blocks or subsystems for audio, image, video, security, phone, wireless, networking and other applications. They require hardware implementation or acceleration because of low-power considerations. For many of these applications, the starting point is a software implementation, often in Matlab, C or C++, either because that is how they were originally prototyped, or because that code is the "reference" code for a public standard. In this document we outline the typical development process for implementing such algorithmic applications in Bluespec SystemVerilog (BSV).

This document is aimed at two audiences:
- Designers who just want to do such an implementation, reading this as a methodology cookbook.
- Technologists who want better to understand why BSV is advantageous for this activity. In particular, to gain insight into the apparent paradox that, while the designer specifies and controls microarchitecture precisely in BSV, it is nevertheless a very powerful "High Level Synthesis" tool.

Abbreviations:

HLS	High Level Synthesis
IP	"Intellectual Property" block—the design under consideration
DUT	Device Under Test—the design under consideration
Tb	Testbench
BSV	Bluespec SystemVerilog
DC	Synopsys' Design Compiler (used here as a representative example of any commercial RTL-to-netlist synthesis tool)

Step 1: Initial functional implementation in BSV

Our first step is to create an initial BSV design that is functionally correct. This happens in one of two ways:

1. Whenever we have a functional or mathematical spec that is sufficiently clear and detailed, it is easier to directly implement it in BSV rather than starting with C code. This is because the style in which C code is written often obscures the 'natural' parallelism of an algorithm. This, in turn, is a natural consequence of (a) C's definition as a sequential language, (b) the computation model assumed by the C programmer, including stacks, heaps, global variables, pointers and so on, and (c) the "cost model" assumed by the C programmer for data access, namely uniform RAM memory access. These properties of C are not naturally suitable for parallel hardware description.

2. If we don't have a separate functional spec and only have C code available, then we do an initial transformation of the C code into BSV. This is a manual step—there is no automatic tool to translate Matlab or C or C++ to BSV. The ease or difficulty of this task depends on the cleanliness of the C code (see previous para about complicating assumptions often made by the C programmer).

   As a concrete "good" data point, we recently converted a certain wireless signal processing application from about 1200 lines of C to about 1100 lines of BSV. The (manual) conversion took about two hours. The final BSV looked very similar to the source C, modulo small BSV vs. C syntax differences. After final synthesis, this represented a design of about 200 Kum^2 in 65nm technology.

   On the other hand, some C codes may be written with such heavy use of global variables, pointers, (algorithmically) unnecessary sequencing, and obscure data flow that this task can be quite tedious (and

this is the reason that there is no automatic tool to perform the translation). Sometimes it is beneficial to first clean up the structure of the C code within C itself, before attempting to transform it into BSV (this is usually necessary anyway with other C-based synthesis tools.) From this point of view we expect Matlab sources to be easier to translate, because they are often written in a cleaner, more mathematical style, where the data flow is much more evident and well structured.

As a concrete example, the reference code for an H.264 decoder (video) is 80K lines of C. The ffmpeg code for Linux which incorporates many codecs is 200K lines of C, of which it is estimated that 20K lines are for H.264 decoding. In contrast, the BSV source is 10K lines, and much cleaner and simpler than the reference C code.

BSV also has a capability for 'importing' C code into a BSV design. This allows us to reuse original testbench C code, possibly in its entirety. The capability also allows us to incrementally implement the design itself, by reusing some parts of the C code and postponing their conversion into BSV.

The initial functional model need not contain the full functionality of the IP. For example, many algorithmic codes compute a family of related functions, perhaps based on some mode parameters. The initial BSV code may contain just a subset (perhaps one) of the modes. As discussed later, BSV allows easy incremental addition of functionality.

The initial functional model is immediately tested and verified (for the functionality implemented so far) with respect to the C code, on the same data sets used by the C code.

The Bluespec tools can also encapsulate a BSV subsystem as a SystemC object. This can be directly plugged into an existing standard SystemC environment, and thus this is another option for verification, that is, the testbench may be an existing or a new SystemC program driving the DUT.

As long as it does not use the 'import C' mechanism, this initial DUT (and even the testbench) is immediately synthesizable by our tool into RTL, which is itself synthesizable by DC into a netlist. Of course, at this stage it will mostly be a giant combinational circuit, and therefore not realistic hardware. We mention this here to emphasize that BSV designs provide early availability of netlists—this enables well-informed architectural exploration, early fast simulation/verification on FPGA platforms and emulators, and early access to software developers.

Step 2: Repeated refinement of microarchitecture to final implementation

Then, we refine the initial BSV design using dozens, perhaps hundreds of iterations, where we incrementally and repeatedly

- add more functionality, or
- add more microarchitectural detail

After every refinement, we test it. So, verification is fully intertwined with development, and not a separate post-design activity.

Also, as the design matures, we start doing periodic DC synthesis runs, for feedback on microarchitecture feasibility. We do further iterations to adjust microarchitecture accordingly.

This kind of frequent iterative refinement is feasible with BSV (unlike with RTL), because of the very powerful abstraction and parameterization mechanisms available in BSV, which we now describe.

First, BSV allows one to abstract out the concept of a 'functional component' as a reusable building block. Then, separately, one can express how to compose these functional components into microarchitectures, such as combinational, pipelined, iterative, or concurrent structures. For example, a function of 'ActionValue' type in BSV expresses piece of sequential behavior.. A function of type 'Rule' expresses a complete piece of reactive

behavior, in fact a complete reactive atomic transaction. All these components are “first class” data types, so one can build “collections” such as lists and vectors of ActionValues, Rules, and so on.

Second, BSV has some powerful 'generate' mechanisms that allow us to do composition microarchitectures flexibly and succinctly. For example, the microarchitectural structure can be expressed using conditionals, loops, and even recursion. These can manipulate lists of rules, interfaces, modules, ActionValues, and so on.

Third, BSV has very powerful parameterization. One can write a single piece of parameterized code that, based on the choice of parameters, results in different microarchictures (such as pipelined vs. concurrent vs. iterative, or varying a pipeline pitch, or using an alternative modules, and so on.).

Finally, and most importantly, the feature of BSV that makes all this flexibility feasible is that, at its core, BSV is based on synthesis from *atomic transactions*. Each change in microarchitecture from the above capabilities of course needs a corresponding change in the control logic. For example, if two functional components are composed in a pipelined or concurrent fashion, then they may conflict on access to some shared resource, whereas when composed iteratively, they may not—these require different control logics. When designing with RTL, it is simply too tedious and error-prone to even contemplate such changes and to redesign all this control logic from scratch. Because of BSV's synthesis from atomic semantics, this control logic is resynthesized automatically—the designer does not have to think about it.

For example, in a mathematical algorithm, many sections of the code represent N-way 'data parallel' computations, or 'slices'. We first abstract out this slice function, and then we can write a parameterized piece of code that chooses whether to instantiate N concurrent copies of this slice, or N/2 copies to be used twice, or N/4 copies to be used 4 times, and so on. Similarly, each of these slices could be pipelined, or not. BSV automatically generates all the intermediate buffering, muxing and control logic needed for this.

So, the designer can rapidly adjust the microarchitecture in response to timing and area results from actual DC synthesis, and converge quickly on an optimized design. The baseline atomicity semantics of BSV is key to preserving correctness and eliminating the effort that would be needed to redesign the control logic.

Implementing mathematical data and operations

Orthogonal to the above microarchitectural considerations, BSV has other features that helps the implementor of mathematical algorithms.

BSV's very strong type system permits definition of abstract mathematical data types, such as fixed point data. Widths of data fields can be specified precisely, with detailed static checking of constraints between widths of related data (for example, the width of the output of a multiplier based on the widths of the operands). BSV provides libraries for fixed-point arithmetic types and operations. Being in source code, these are modifiable by the user.

In addition,

- We have meta-level tools to generate RAM lookup tables for fixed matrix data (tables of constants).
- We have meta-level tools to generate RAM lookup tables with intrapolation for doing more complicated functions like square roots, exponentiation and trigonometric functions.

Finally, BSV has mechanisms to import Verilog components, for example if one wishes to use a very specific implementation of a high-performance divider.

Some comparisons with classical C-based High Level Synthesis

Having described the BSV approach, we can now make some brief comparisons with classical C-based HLS.

In classical C-based HLS, the design-capture language is C (or C++). To this are added proprietary "constraints" that specify, or at least guide the synthesis tool in microarchitecture selection, such as loop unrolling, loop fusion, number of resources available, technology library bindings, and so on. The synthesis tool uses this, and knowledge about a particular target technology and technology libraries, to produce the synthesized output.

It is generally not possible to take an off-the-shelf C code directly into a C-based High Level Synthesis tool. Global variables, pointers and pointer arithmetic, dynamically allocated data (malloc), recursion, dynamic loop indexes, complex control structures, stack-based arrays etc. are typical obstacles. An existing C code must often be "massaged" into a form suitable for the synthesis tool. In addition, the user must provide several constraints that specify or guide the synthesis effort (such as degrees of loop unrolling, fusion, resource sharing, etc.). The following paper illustrates the kinds of transformation necessary on off-the-shelf C code to make it suitable for C-based High Level Synthesis:

> *A Code Refinement Methodology for Performance-Improved Synthesis from C,*
> Greg Stitt, Frank Vahid and Walid Najjar, in *Proc. ICCAD'06, November 5-9, 2006, San Jose, CA*
> http://www.cs.ucr.edu/~vahid/pubs/iccad06_guidelines.pdf
> http://www.cs.ucr.edu/~vahid/pubs/iccad06_guidelines.ppt

In our experience, writing a BSV version of the application can take less effort than this kind of massaging and constraint creation for C-based tools.

In BSV, the microarchitecture is specified precisely in the source, but with such powerful generative and parameterization mechanisms that a single source can flexibly represent a rich family of microarchitectures, different choices within which may be appropriate for different performance targets (area, clock speed, power). Further, the structure can be changed quickly and easily without compromising correctness or hardware quality, in order quickly to converge to a satisfactory implementation.

BSV synthesis is currently technology neutral—it does not try to perform technology-specific optimizations or retimings (BSV users rely on downstream tools such as DC Ultra to perform such technology-specific local retiming optimizations).

Conclusion

The techniques described here have been applied successfully to several real designs, including: an H.264 decoder, a parameterized OFDM transmitter and receiver (parameterized to cover 802.11a, 802.16 and WUSB), a wireless front end signal processing application, an image color manipulation application, a video encoder prediction function, and encryption and decryption functions. In each case the quality of synthesized results were competitive with hand-coded RTL, but it was produced with significantly less effort (including verification), and with significantly more reusable components.