Because of its expressive power (comparable to the most advanced programming languages), full synthesizability, and quality of synthesis, Bluespec is fundamentally changing long-held assumptions about design flow:

- **What must be done in SW vs. in HW**
- **How to achieve early, fast simulation for spec development, early SW development, architectural exploration and verification**
- **How to more closely and formally link specs to implementations**
- **How to make testbenches, models and designs more maintainable and reusable**
- **... and more**
BSV’s unique advantages, at a glance

1. Familiar module hierarchy from Verilog
   - precise, transparent control over architecture, unlike C, C++, SystemC.

2. Atomic Transactions (rules)
   - instead of Verilog ‘always blocks’ and SystemC ‘sc_threads’
   - automatically generate control logic for complex, concurrent access to shared resources

3. Atomic Methods instead of Verilog ‘port lists’ and SystemC methods
   - reduces/eliminates interface-related bugs/race conditions and integration issues, promoting reuse

4. More powerful Types and Strong Type-checking than Verilog or SystemC
   - statically eliminates representational errors, enforces clock domain discipline.

5. Fully synthesizable at all levels of abstraction
   - early architectural feedback
   - early and very fast simulation on off-the-shelf FPGA boards for architectural exploration, verification and SW development

6. Rich, parameterized libraries of interconnect IP (TLM 2.0, OCP, AXI, AHB, ...)
   - Very quick development of models, designs, and testbenches

7. Transactional Verification IP
   - Architectural (approx)
   - Very high level (>>100X speed)
   - Strong formal theoretical basis
     - Composition modules → IPs → Systems
     - Reuse (IP and Testbenches)
     - Path to formal verification

Some implications of BSV’s unique advantages

Software

- Early fast SW platform
- Executable spec
- BlueSpec Tools
  - Synthesis & Execution at every step
  - BlueSim (1x speed)
  - BlueSpec Synthesis
  - Verify/Analyze/Debug
  - RTL simulation (1x speed)
  - FPGA Emulation (>100X speed)
  - RTL synthesis
  - Power Estimation

Verification IP

- Initial Testbench (approx)
- Final Testbench (approx)
- Add Functionally
- Refine/Refactor
- Final IP
- Final Testbench

Design IP/System

- Initial IP (approx)
- Add Functionally
- Refine/Refactor
- Final IP (approx)
- Final IP

Strong formal theoretical basis
- Composition modules → IPs → Systems
- Reuse (IP and Testbenches)
- Path to formal verification

Much higher level than RTL, with same QoR (quality of results)

Synthesizable testbenches for fast verification
Atomic transactions are the best known tool to tame complex concurrency

For decades: in Operating Systems, Databases, Distributed Systems

Recently: for software for multi-core/multi-threaded architectures
"I think we ultimately will see atomic transactions in most, if not all, languages. That's a bit of a guess, but I think it's a good bet."

Burton Smith,
Technical Fellow,
Parallel Computing

Very recently: HW support for “Transactional Memory” in processors

BSV: well positioned for formal verification

- Many formal specification languages use the same computation model as BSV, because it is parallel, and because atomicity of rules enables reasoning about correctness with invariants
  - UNITY (Chandy&Mishra), TLA+ (Lamport), Event B (Abrial), ...
  - Historical note: BSV’s technology roots are in fact in formal verification—verification of complex processor pipelines and complex cache coherence protocols (ask for references)

- The Rules computation model can be used from high levels of abstraction (executable specs) to lower levels (implementations)
  - There is a vast literature on provably correct refinement
  - This is the most promising approach for formal verification
How customers/partners are using BSV:
- Modeling for SW development
- Modeling for Architecture Exploration
- Verification
- and, of course, IP creation

Initially, BSV was only used for IP creation

Example: *modeling for SW devel.*
(@ a major IP company)

- Cycle-accurate model of a production LPDDR memory controller
  - Will be shipped to each customer of the IP company
  - Developed in < 3 man months

Why BSV?
- Very quick model creation
- Parameterization
- Fast sim (Bluesim)
- Potential for much faster sim (FPGA)
- Potential to replace IP creation as well
How customers/partners are using BSV:
- Modeling for SW development
- **Modeling for Architecture Exploration**
- Verification
- and, of course, IP creation

Example: **CPU architecture exploration**
(processor microarchitectures @ a major microprocessor company)

- Rewriting 1000s of lines of legacy C++ code for CPU pipeline architecture modeling in BSV, so that it can be synthesized and run on FPGAs
  - Expected performance advantage > 1000x

- Background: existing microarchitect’s workbench for cycle-accurate exploration of alternative microarchitectures for future CPUs
  - Written in C++, developed over a decade
  - Highly parameterized and configurable, to facilitate experimentation on alternatives
  - Heavily used, but running out of gas for simulation speed (multicore, multithreading)

- Status: Demonstrated for pipelined out-of-order model; development continues

Why BSV?
- Only tool with expressive power to replace C++, for speed
- and synthesizable to FPGA, for speed
Example: **CPU and cache architecture exploration**
(@ a major microprocessor/systems company)

- Creating a flexible platform for fast exploration of cache microarchitectures for future multithreaded + multicore CPUs

- Status: executing significant prefix of Linux boot sequence, on an FPGA platform, within 6 months of start of project

---

Example: **CPU architecture exploration**
(processor microarchitecture
Prof. Derek Chiou @ UT Austin)

- Goal: flexible platform for fast exploration of pipeline microarchitectures for multithreaded + multicore CPUs

- “FAST” system (functional model + predictive timing model)
  - Predictive timing model written in BSV, synthesized and running on single FPGA (DRC/Xilinx in Opteron socket)
  - Boots unmodified Windows XP and x86 Linux with unmodified x86 apps
  - **100% cycle accurate x86 model, at 1.2 MIPS**
    - Expecting at least 10x more, with simple optimizations
    - Compare 1-200 KIPS for software-based cycle-accurate sim
  - Ref [Chiou et. al. ICCAD 2007 and MICRO 2007]

---

**Why BSV?**
- Very quick creation of high level, architecturally (cycle) accurate models and synthesizable to FPGA, for speed
How customers/partners are using BSV:
- Modeling for SW development
- Modeling for Architecture Exploration
- Verification
- and, of course, IP creation

Example: **IP verification** @ Qualcomm (from publicly available information)
- BSV for complex transactors on EVE platform
  - TLM interfaces
  - Transactor functions: data transformation, clock management, timestamp management, statistics management
  - and, ... moving testbench functionality to EVE side
- Productivity improvements in transactor development:
  - 6 months → 3 months (EVE) → 3 weeks (BSV)
Example: **IP verification**
(© major semi company)

- Multi channel DMA controller
  - ~350K gates (11% smaller than reference design)
  - 227 MHz (35% faster than 166 MHz spec)
- Verification results:
  - Single channel verification detected ~12 bugs
    - 50% typos, rest logic errors
  - Multi-channel verification detected ~9 bugs
    - Several of which were in test environment
  - Most bugs found and fixed in 1-2 minutes
    - None longer than 15 minutes

Why BSV?
- Fewer bugs with atomic transactions, both within a module and surrounding it (correct-by-construction control logic)
- Remaining bugs (few) are easier to find and fix

---

Example: **IP verification**
(AXI demo on FPGA © Bluespec)

![AXI demo diagram](image-url)
Example: **IP verification**
(AXI demo on FPGA @ Bluespec)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bluespec lines of code:</td>
<td>2,000 (including comments)</td>
</tr>
<tr>
<td>ASIC gates:</td>
<td>125K</td>
</tr>
<tr>
<td>Virtex-4 FX100 slices:</td>
<td>4,723 (10% utilization)</td>
</tr>
</tbody>
</table>

**Verilog**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C/SystemC/Bluesim</td>
<td>Verilog simulation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bluesim**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C/SystemC/Bluesim</td>
<td>Bluesim Simulation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Emulation**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C/SystemC/Bluesim</td>
<td>FPGA Emulation</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

22 day Verilog sim → 1 day Bluesim → 53 sec on FPGA

---

Example: **IP verification**
(AXI demo on FPGA @ Bluespec)

- Why BSV?
  - Inexpensive emulation platform (low cost FPGA board)
  - Quick setup (BSV transactors)
  - Easy availability (synthesizability of early approximation)

---

Why BSV? •Inexpensive emulation platform (low cost FPGA board) •Easy setup (BSV transactors) •Quick availability (synthesizability of early approximation)
How customers/partners are using BSV:
- Modeling for SW development
- Modeling for Architecture Exploration
- Verification
- and, of course, IP creation

Initially, BSV was only used for IP creation

Example: **FPGA IP creation**  
(MEMOCODE 2008 codesign contest)

Goal: Speed up a software reference application running on the PowerPC on Xilinx XUP reference board using SW/HW codesign

Application: Decrypt/sort/re-encrypt list of records. Optimal solutions must solve both sorting and crypto problems with large databases

27 teams started the four week contest. 8 teams delivered 9 solutions.

[Xilinx XUP](http://rijndael.ece.vt.edu/memocontest08/)
Example: **FPGA IP creation**
(MEMOCODE 2008 codesign contest)

**Normalized Speedup**

<table>
<thead>
<tr>
<th>Contest Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference: <a href="http://rijndael.ece.vt.edu/memocontest08/everybodywins/">http://rijndael.ece.vt.edu/memocontest08/everybodywins/</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Team</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunita (2)</td>
<td>11X</td>
</tr>
<tr>
<td>Eric</td>
<td>5X in 2007</td>
</tr>
<tr>
<td>Rob</td>
<td></td>
</tr>
<tr>
<td>Vijay</td>
<td></td>
</tr>
<tr>
<td>Suniita (1)</td>
<td></td>
</tr>
<tr>
<td>Marco</td>
<td></td>
</tr>
<tr>
<td>Brian</td>
<td></td>
</tr>
<tr>
<td>Kermin</td>
<td></td>
</tr>
</tbody>
</table>

**Why BSV?**
- Expressive power enabled all-BSV solution
- and fully synthesizable to FPGA, for speed

---

Example: **ASIC IP creation**
(@ three major semi companies)

- **Company A:**
  - High performance video data mover for video subsystem
    - In silicon; derivatives in progress

- **Company B:**
  - System DMA for wireless handset platform
    - In silicon; derivatives in progress
  - Image DMA
    - Verified, synthesized, meeting PPA. Designer reports 50% fewer bugs compared to Verilog flow
  - LCD controller
    - In progress

- **Company C:**
  - Wireless Turbo Decoder
    - 15% area (ASIC) than original VHDL
    - 5X faster verification in Bluesim
    - Now running on FPGA, expecting > 1000x faster verification runs

**Why BSV?**
- Complex concurrency
- Parameterization
Example: **ASIC IP creation**
(OFDM @ MIT and Tampere U. Technology)

- Highly parameterized source code, from which one can instantiate three different wireless protocols:
  - 802.11a (WiFi) (MIT)
  - 802.16 (WiMax) (MIT)
  - 802.15.3 (WUSB) (Tampere)
    - Starting with MIT code for WiFi and WiMax, two Tampere students were able to extend it to WUSB in 6 weeks, learning BSV and WUSB at the same time!

- Powerful block-level parameterization
  - E.g., IFFT (85% of 802.11a transmitter) is parameterized to allow a wide range of micro-architectures
    - Variable number of “butterfly-4s”—from 1 (full reuse) to 16 (no reuse) per stage
    - Variable number of stages—from 1 (full reuse) with, of course, different area, performance

This BSV code is open sourced at [http://csg.csail.mit.edu/oshd](http://csg.csail.mit.edu/oshd)

Note: parts of this IP are considered the “sweet spot” for C-based synthesis

---

Example: **ASIC IP creation**
(H.264 decoder @ MIT)

- Complete decoder, not just kernel blocks
- Range of implementations from a single, parameterized source code:
  - from QCIF: 176x144 @ 15 frames/sec
  - to 1080p: (1280x1080)p @ 60 frames/sec

- Synthesized at 180 nm

- < 10K lines of BSV source code. Compare:
  - Original reference code: > 80K lines of C
  - H.264 slice of FFmpeg: 20K lines of C

This BSV code is open sourced at [http://csg.csail.mit.edu/oshd](http://csg.csail.mit.edu/oshd)

Note: parts of this IP are considered the “sweet spot” for C-based synthesis
A few small examples, for flavor

- What follows are a few small examples to illustrate what is possible in BSV

- Please hold off on detailed technical questions until later; don’t worry if these are not fully understandable now
  - This is just the “overture music”, to bathe you in the leitmotifs, the idioms, the themes, as you settle comfortably into your seats leaving behind the stresses of the street
  - The plot of the opera will be revealed to you as we progress through the tutorial!
    - Patience: the soprano will sing; Baron Scarpia will die!
Multiplier Example

Simple binary multiplication:

\[
\begin{array}{c}
\text{x} & \text{1001} & \text{// d = 4'd9} \\
\text{0101} & \text{// r = 4'd5} \\
\text{1001} & \text{// d << 0 (since r[0] == 1)} \\
\text{0000} & \text{// 0 << 1 (since r[1] == 0)} \\
\text{1001} & \text{// d << 2 (since r[2] == 1)} \\
\text{0000} & \text{// 0 << 3 (since r[3] == 0)} \\
\text{01011011} & \text{// product (sum of above) = 45}
\end{array}
\]

What does it look like in Bluespec?

module mkTest ();
endrule
rule finish (state == 1);
endrule
rule go (state == 0);
Mult_ifc m <- mkMult1();
Reg#(int) state <- mkReg(0);
m.start (9, 5);
method result () if (r == 0);
method Action start (x,y) if (r == 0);
endrule
rule cycle (r != 0);
Reg#(Tin)  r             <- mkReg(0);
Reg#(Tout) d           <- mkReg(0);
method Action start (Tin x, Tin y);
return product;
d <= x; r <= y; product <= 0;
endmethod
endmodule: mkTest

interface Mult_ifc;
method Action start (Tin x, Tin y);
method Tout result ();
endinterface: Mult_ifc

module mkMult1 (Mult_ifc);
Reg#(Tout) product <- mkReg(0);
Reg#(Tout) d        <- mkReg(0);
Reg#(Tin)    r            <- mkReg(0);
rule cycle (r != 0);
if (r[0] == 1) product <= product + d;
d <= d << 1;
r <= r >> 1;
endrule
method Action start (x,y) if (r == 0);
d <= x; r <= y; product <= 0;
endmethod
method result () if (r == 0);
return product;
endmethod
endmodule: mkMult1
Concurrency and Shared Resources

- Process 0 increments register x under some condition cond0
- Process 1 transfers a unit from register x to register y under cond1
- Process 2 decrements register y under some condition cond2

- Each register can only be updated by one process on each clock.
  - Priority: 2 > 1 > 0

- This is an abstraction of some real applications:
  - Bank account: 0=deposit, 1=transfer to savings, 2=withdraw from savings
  - Packet processor: 0=packet arrives, 1=packet is processed, 2=packet departs
  - ...

*There are other ways to write this RTL, but all suffer from same analysis*

**Fundamentally, we are scheduling three potentially concurrent atomic transactions that share resources.**

What if the priorities changed: cond1 > cond2 > cond0? What if the processes are in different modules?
Concurrency and Shared Resources (VHDL)

```
process(CLK) begin  // process 0
  if (CLK = '1' and CLK'event) then
    y <= y + 1;
  end if;
end process;

process(CLK) begin  // process 1
  if (CLK = '1' and CLK'event) then
    x <= x - 1;
    y <= y + 1;
  end if;
end process;

process(CLK) begin  // process 2
  if (CLK = '1' and CLK'event) then
    x <= x + 1;
    y <= y - 1;
  end if;
end process;
```

With Bluespec, the design is direct

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

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

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

Hand-written RTL:
- **Explicit scheduling**
- **Complex clutter, unmaintainable**

**BSV:**
- Functional correctness follows directly from rule semantics (atomicity)
- **Executable spec (operation-centric)**
- Automatic handling of shared resource control logic
- Same hardware as the RTL
Now, let’s make a small change: add a new process and insert its priority

Process priority: 2 > 3 > 1 > 0

Changing the Bluespec design

Pre-Change

(* descending_urgency = "proc2, proc1, proc0" *)
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, proc3, proc1, proc0" *)
rule proc0 (cond3);
  x <= x + 1;
endrule
rule proc1 (cond1);
  y <= y + 1;
  x <= x - 1;
endrule
rule proc2 (cond2);
  y <= y - 1;
endrule
rule proc3 (cond3);
  y <= y - 2;
  x <= x + 2;
endrule
Changing the Verilog design

Pre-Change
always @(posedge clk) begin
    if ((cond2 && cond1) || (cond0 && !cond1 && !cond3))
        x <= x + 1;
    else if (cond3)
        y <= y - 1;
    else if (cond1)
        y <= y + 1;
    end

endcase

Alternate RTL style (more common)
always @(posedge clk) begin
    case ((cond0, cond1, cond2))
        3'b000: begin // nothing happens
            x <= x; y <= y;
        end
        3'b001: begin // proc2 fires
            y <= y-1;
        end
        3'b010: begin // proc1
            x <= x-1; y <= y+1;
        end
        3'b101: begin // proc2 fires
            (2:1)
            y <= y-1;
        end
        3'b110: begin // proc0
            x <= x+1;
        end
        3'b111: begin // proc2 + proc0
            x <= x+1; y <= y+1;
        end
        3'b111: begin // proc2 + proc0
            x <= x+1; // NOTE: subtle!
        end
    endcase

Process priority: 2 > 3 > 1 > 0
The concurrency complexities illustrated in the simple bank account example are greatly magnified in real designs.

A more complex example, from CPU design

- Speculative, out-of-order
- Many, many concurrent activities
Many concurrent actions on common state: nightmare to manage explicitly

But in BSV...

- ..you can code each operation in isolation, as a rule
- ..the tool guarantees that operations are INTERLOCKED (i.e. each runs to completion without external interference)
Example: a butterfly switch (crossbar)

Basic building blocks:

Recursive construction: 1x1 ➔ 2x2 ➔ 4x4 … ➔ NxN

Butterfly switch: interface

```vhdl
interface XBar #(numeric type n, type t);
    interface Vector#(n, Put#(t))    input_ports;
    interface Vector#(n, Get#(t))    output_ports;
endinterface
```

- Parameterized by #ports (n), packet type (t)
- Intuitive high-level structures for the interface:
  - Hierarchical sub-interfaces
  - Aggregation (vectors of interfaces, lists, ...)

Copyright © Bluespec Inc. 2005-2008
Butterfly switch: module
(< 60 lines, fully synthesizable)

module mkXBar #(function UInt #(32) destinationOf (t x),
    module #(Merge2x1 #(t)) mkMerge2x1)
    ( XBar #(n, t) ) :
    if (logn == 0) .. // BASE CASE
        FIFO#(t) f <- mkFIFO;
    else .. // RECURSIVE CASE
        ..
        XBar#(nhalf, t) upper <- mkXBar ( ..);
        XBar#(nhalf, t) lower <- mkXBar ( ..);
        ..
        for (Integer j = 0; j < n; j = j + 1) ..
            rule route; ..
            if (! flip) merges [j].iport1.put (x);
            else
                merges [jFlipped].iport1.put (x);
        endrule
    endmodule: mkXBar

Key Implementation Notes:
- The switch module: < 60 lines of BSV code
- First working (tested) prototype: < 1 day
  (including simple testbench)
- Fully synthesizable:
  Synthesized to layout (Magma, TSMC 0.18u, 550 MHz)

(see also whitepaper for full source code)
Elevating Design above RTL

Bluespec SystemVerilog

Behavioral

- Correct: Concurrency and Communications
  - Atomic Transactions using Rules and interface methods for complex control and concurrency:
    - Across multiple shared resources
    - Across module boundaries

Structural

- Correct: Construction and Configurability
  - High-level types closer to spec
  - Much more powerful "generate"
  - Powerful parameterization
  - Powerful static checking & formal semantics
  - Advanced clock management

VHDL/Verilog/SystemVerilog/SystemC

Setting your expectations about what you will learn from a few days of training

“Earn your Ph.D. from home in just 6 months! Call toll-free now!”

(spam)

- Mapping an idea or specification into BSV hardware, and using abstraction well, takes training and practice: one can’t become an expert overnight!

- However, even after these few days of training, you will be able to design using BSV, and still expect at least a 2x improvement in your productivity
  - The basics are easy!
  - Even easier than Verilog/VHDL, we believe, because of the direct, natural way to describe HW that is HW-centric and not simulation-centric. Many people have learned HW design for the first time using BSV.
  - Experience in previous training sessions has borne this out time and again!
  - “Aha! I get it now! Wow! Why would I ever go back to RTL?”
Prerequisites

- Basic digital design
- Exposure to basic microarchitectures: pipelining, FSMs, data and control paths,
- Done some coding in RTL (Verilog or VHDL)
- Done some verification of RTL
- Aware of typical flow: architecture exploration, design, verification, physical design
- Not assuming any prior exposure to SystemVerilog