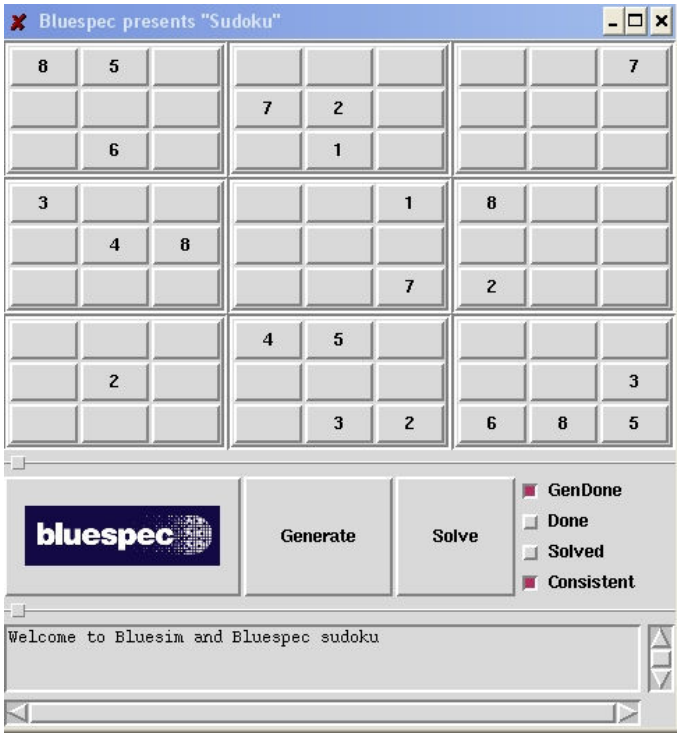
# BluDACu

## A Bluespec Hardware Implementation of Sudoku

**June 2007**
**www.bluespec.com**
**Document version 1.1**

## *Introduction*




So, you might be asking, "Why would anyone create a hardware implementation of Sudoku?" Or even, "What does this have to do with me and what I do?" The simple answer is that we wanted a fun, thought-provoking demo for DAC – and, it fits well with our electronic Sudoku giveaway. The more serious answer starts with "we could, and easily, with Bluespec SystemVerilog", but we'll get into that a bit later. Most importantly, while this Sudoku implementation is a novelty of sorts, though a very complex one, it nicely illustrates how Bluespec's ESL Synthesis technology effectively tackles key, contemporary System-on-Chip (SoC) development issues:

- How to test software early with accurate hardware representations
- How to create intellectual property (IP) blocks that enable extreme reuse because they are highly parameterized and flexible to change
- How to quickly implement designs that contain both extremely complex algorithms and complex concurrency, especially those with convoluted control logic and numerous shared resource hazards (it is often assumed that dealing with high complexity is only feasible in software)

While we hope you enjoy this design, we also hope that it intrigues you to pursue Bluespec toolsets further. Of course, Bluespec is typically used for more serious fare. Our other DAC demos included an AXI® bus based design and an H.264 video decoder – and we've been used for DMA controllers, cache controllers, processors, network interfaces, and many more… Upon learning more about Bluespec, we think you'll be surprised at just how high-level it is possible to express implementations—and how Bluespec offers a unified environment for what are currently disparate activities: virtual prototyping, architectural modeling, verification and implementation.

If you are not yet familiar with Bluespec, then prepare yourself for something very different from what you have seen or experienced before. Though some vendors in the algorithm space have been successful at

efficiently synthesizing higher-level math/DSP designs, Bluespec is the only solution that succeeds both for control *and* complex datapaths. We invite you to take a deeper look – the typical response we get upon learning about Bluespec is "Bluespec wasn't what I expected at all". What you'll find is truly unique, both in approach and results:

- Elevated hardware design and modeling that keeps designers 100% in control of the architecture and micro-architecture of their implementations.
- A unified environment for virtual prototyping, architectural exploration and IC implementation.
- No opportunity cost in adoption. As it layers incrementally on current flows – and generates readable, predictable Verilog RTL – Bluespec can be used one block at a time, without upending your current toolsets and methodologies. With less than a week of training, designers have consistently completed their first project, including design and verification, in less than half the time compared to designing with RTL or SystemC.

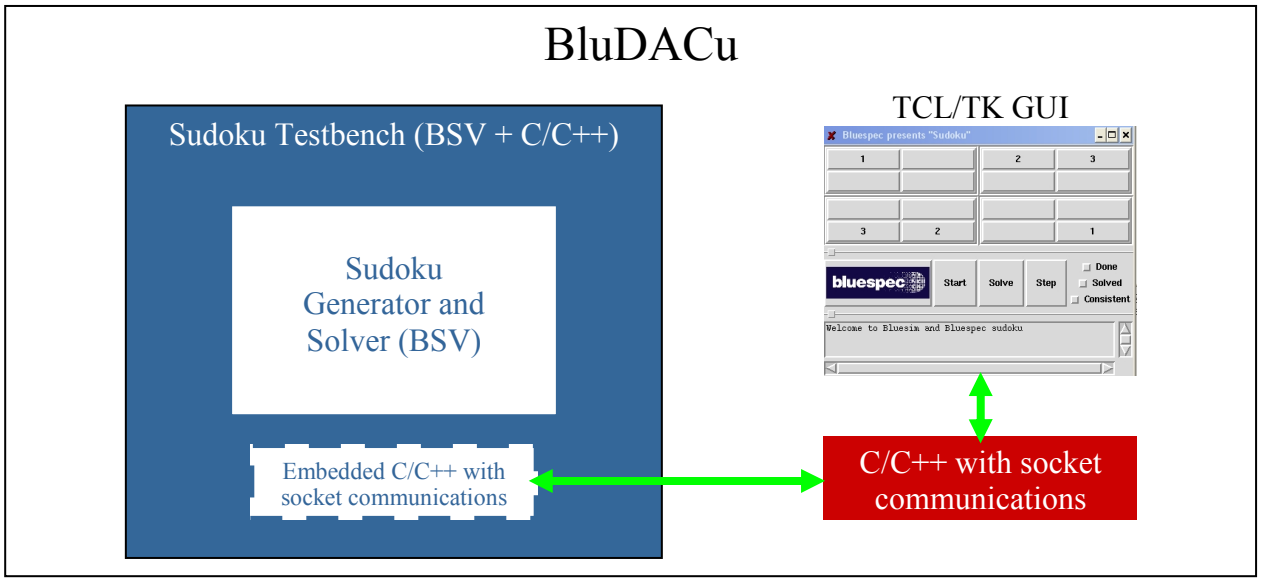
This document describes BluDACu, a Bluespec parameterized hardware implementation of Sudoku, including both puzzle generation and puzzle solving. It should be a fun read – and illustrates powerful capabilities not before seen in hardware system design. All this is made possible because of BSV's high level of abstraction, powerful types and strong static verifications, clean semantics based on atomic transactions (Rules) and atomic-transactional (Rule-based) Interfaces, and automatic regeneration of correct control logic as we substitute one component with another.

## *BluDACu Overview*

You are probably familiar with the game of Sudoku. If not, one resource is the Wikipedia entry: http://en.wikipedia.org/wiki/Sudoku. Nowadays Sudoku puzzles seem to be offered in many daily newspapers, magazines and books, and there are also several Sudoku sites on the web. The initial setup and the challenge are simply described. You are given a 9x9 grid of cells, which can be further regarded as a 3x3 grid of boxes, each of which is a 3x3 grid of cells (see opening illustration in this document). Some of the cells are already filled in, the rest are blank. The challenge is to fill in all the cells, where each cell contains a symbol (1-9), such that each 9-cell row, each 9-cell column, and each 3x3-cell box contains exactly one occurrence of each symbol 1 through 9. A legal puzzle must have exactly one solution so that, in principle, it can be solved by pure reasoning, i.e., without any "guessing".

The usual 9x9 puzzle can be regarded as a special case of *N* x *N* puzzles, where *N* is a square. We refer to the 9x9 puzzle as "order 3", and you could also have "order 2" (4x4 cells with 2x2 boxes and using symbols 1-4), "order 4" (16x16 cells with 4x4 boxes and using symbols 1-9,A-G), and so on.

BluDACu is a parameterized hardware implementation of the game of Sudoku. The "order" of the puzzle is a parameter to the design, so that the same source code can be used for size 2x2, 4x4, 9x9, 16x16, and so on. The implementation includes a puzzle generator, a puzzle solver, and a software-based GUI front-end to control the hardware and display the game for human play. The following is a block diagram outlining the architecture of these components:




The implementation was done using Bluespec SystemVerilog (BSV) – alternatively, it could have been done using Bluespec's ESE SystemC, which has the same notion of atomic transactions (Rules) and atomic-transactional interfaces. The core of the design, the Sudoku generator and solver, and part of the surrounding testbench are in BSV. The testbench makes calls to C functions using BSV's DirectC interface – the BSV design and embedded C can be run at BSV source level using Bluesim, or at the RTL level, in Verilog, using standard Verilog simulators. The BSV-generated Verilog can be further synthesized into FPGA or ASIC netlists, and run on an FPGA or in ASIC silicon.

## *The general approach used in BluDACu*

The Sudoku solver was purposely developed to mimic the way a human solves the puzzle, so that we can provide the human player with meaningful assistance if he requests a hint—e.g., instead of just magically filling in the solution symbol for a blank cell, we can describe the reasoning process that solves that cell. Further, this structure also allows us to tune the level of difficulty of the generated puzzles.

Our solver does *not* use "backtracking", which is one of the "easy" ways to write a Sudoku solver in software, i.e., at each step, one could "guess" the solution for a blank cell and proceed, possibly realizing after many steps that the guess was wrong because it leads to an inconsistency. Such a solver would require saving the state of the grid at this "guess point", so that if the guess turned out be wrong, it could backtrack, or restore the grid state to this guess point, and make a different guess. Of course, after making one guess it may be necessary to guess again for some other cell, and so on, so that at some point one may be "nested" in several levels of guesses. Such a solver needs no further intelligence—it is a kind of "brute force" search of the solution space. This is easy to do in software, but humans typically do not employ this approach, since backtracking is hard to do with pencil on a folded newspaper while hanging on to a strap on the morning bus or subway! Further, in such a solver, it is not easy to provide a meaningful hint to the player at any intermediate point, i.e., a focused hint about *how* to solve a particular cell.

Finally, a backtracking solver would be very bad for hardware implementation. It would need much memory to save all the grid states at all the pending guess points, and it might consume a lot of power.

Instead, BluDACu works like a human solver, repeatedly employing a *tactic* chosen from a repertoire of tactics (a "bag of tricks"). Each successful application of a tactic makes concrete progress by solving one cell. An example of a simple tactic applied to a particular cell is:

> Tactic *elim_other_singletons*: Eliminate as possibilities all symbols from this cell that already appear as the solution to any other cell in the same "constraint group" (same row, same column, or same box).

Said in a reverse way, if you've already solved a cell, eliminate it as a possibility from all other cells in the same row, column or box. A more complex tactic:

> Tactic *repeated_2_set*: If, in a constraint group (row, column, box) of this cell A, two other cells B and C contain the same "2-set" { *j, k* }, i.e., symbols *j* and *k* are the only remaining possibilities in cells B and C, then *j* and *k* can be eliminated from this cell A (because *j* and *k* must be in the cells B and C, even though we may not yet know which one goes where).

By repeatedly applying each tactic to each cell in the grid, and to each of the cell's containing constraint groups (row, cell, box), steady progress is made until the puzzle is sovled. Note that there is no guarantee that the repertoire of tactics is "complete", i.e., that they are capable of solving *any* Sudoku puzzle. The stronger the tactics, the more difficult the puzzle that can be solved. BluDACu's tactics succeed on many puzzles found in newspapers, books and web sites, including those that are labeled by their authors as "hard", "difficult", "evil", etc. (You will also see in the discussion below how easy it is to add a new tactic to the solver, when you discover one.)

## *Expressive and parameterized data types*

The file `Sudoku.bsv` contains declarations for a number of the parameterized data types used in the solver. For example:

```
typedef Bit#(TSquare#(order)) Cell#(numeric type order);
```

defines the type *Cell#(O)*, the representation of a cell in an order *O* puzzle, to be a bit-vector of $O^2$ bits. For example, in an order 3 puzzle, each cell has 9 bits. This encodes which values are possible candidates for that cell. When a value is ruled out at a particular cell location, its corresponding bit is cleared. When only one bit remains set in the mask for a cell, the value of that cell has been determined (solved). The declaration:

```
typedef Vector#(rows, Vector#(cols, t))
        Grid#(numeric type rows, numeric type cols, type t);
```

defines the type *Grid(nr,nc,t)* as a vector or *nr* rows containing a vector of *nc* columns containing items of some type *t.* By parameterizing in this way, the type can be reused in various ways. For example, the declaration:

```
typedef Grid#(TSquare#(order), TSquare#(order), Reg#(Cell#(order)))
        SudokuRegGrid#(numeric type order);
```

defines the type of the "state" for the Sudoku solver of order *O*— it is a *Grid* with $O^2$ rows and $O^2$ columns of registers (type *Reg#()*), where each register contains a *Cell* value (bit vector of $O^2$ bits). But the *Grid* type can also be used to represent the abstract *values* in the Suduko grid:

```
Grid#(TSquare#(order), TSquare#(order), Cell#(order))
```

i.e., here the values in the grid are just *Cell* values, not registers. Or,

```
Grid#(order, order, Cell#(order))
```

can be used to represent the values in a box (e.g., a 3x3 box in a 9x9 puzzle). Similarly:

```
typedef Vector#(TSquare#(order), Cell#(order)) Group#(numeric type order);
```

defines *Group(O)* as a parameterized type that is a vector of $O^2$ cells. It can be used to refer to the values in a row, column, or box of cells, each of the three kinds of "constraint groups". By parameterizing our tactics in terms of *Groups*, the same tactic can be used on rows, columns or boxes.

This level of parameterization is made possible by Bluespec SystemVerilog and is not practical in legacy RTL languages. The "order" parameter controls the size of the puzzle. In general, an order *O* puzzle contains $O^4$ cells arranged in an $O^2 x O^2$ grid and each of the $O^2$ rows, columns and boxes contains the symbols 1 through $O^2$. Thus, an order *O* puzzle requires $O^6$ bits of state! You see some hint of these "constraint" relationships (which you can think of as formal assertions) in the use of phrases like *TSquare()* in the above examples.

The files `SolveTest2.bsv` and `SolveTest3.bsv` contain small unit-level testbenches for solvers of order 2 (4x4) and 3 (9x9), respectively. In those files, you'll see lines like

```
SudokuSolver#(2) solver <- mkSudokuSolver();
```

and

```
SudokuSolver#(3) solver <- mkSudokuSolver();
```

respectively. This simple, single-parameter change, from 2 to 3, automatically adjusts a *lot* of generated hardware circuitry, including:

- The bit-width of each cell value and cell register
- The number of cell registers in each row and column
- The width of arguments and results in the various interfaces
- The scope of all of the functions for accessing and modifying Sudoku grids
- The logic for each tactic
- The number of tactic applications
- The sequence of states in the solver FSM
- The sequence of states in the generator FSM
- The logic to place a given solved value somewhere in the grid
- and so on.

In other words, formal constraints like *TSquare()* are exploited by the BSV synthesizer to produce all the logic correctly sized for each order *O* of puzzle. This *correct-by-construction* generation of datapath and control logic eliminates a large source of bugs encountered in legacy RTL.

## *Helper functions, higher-order functions, and library functions*

Good BSV style typically involves heavy use of functions to encapsulate the abstract concepts of each application domain. In legacy RTLs, function arguments and results are usually just bit-vectors and numeric types. In BSV, argument and result types can be of *any* type at all, including Actions (e.g., encapsulating a set of register, memory or fifo ops), Rules (e.g., encapsulating a common set of behaviors), Modules, Interfaces, and so on. The BSV libraries come with many small but useful predefined functions.

For example, several tactics require the concept of a cell with only 2 possible values, and this is easily defined using the supplied library function *countOnes()*:

```
// Determine if a cell has only 2 possibilities
function Bool is2set(Cell#(order) c);
   return (countOnes(c) == 2);
endfunction: is2set
```

Similarly, to check if a cell value appears more than once in a constraint group, the library function *countElem()* is available, which counts how many times a particular value appears in a vector:

```
// Determine if a value is repeated in a group
function Bool repeated(Group#(order) g, Index#(order) n);
   return (countElem(g[n],g) > 1);
endfunction: repeated
```

Beyond such simple utilities, BSV functions, both in the library and user-defined, can be extremely powerful because they are *higher-order functions,* i.e., an argument or result is itself a function. For instance, the *is2set()* function defined above can be used with the higher-order function *map()* to succintly define a mask which shows which cells in a constraint group have only 2 possible values:

```
let two_set_mask = map(is2set, g);
```

The *map()* function here applies the *is2set()* function to each cell in a group *g* (a vector of cells). Each application or course returns a boolean for its particular cell, true if the cell is a 2-set and false otherwise. The *map()* function collects these boolean results and returns a vector of booleans, where the boolean is true only when the corresponding original cell was a 2-set. Such a mask can, in turn, be applied to a constraint group to mask out cell values using another higher-order function, *zipWith()*:

```
// Apply a mask bit to a cell value
function Cell#(order) applyMask(Cell#(order) c, Bool m);
   return (m ? c : impossible());
endfunction: applyMask

// Mask several cells in a group
function Group#(order) maskN(Group#(order) g, Mask#(order) m);
   return zipWith(applyMask, g, m);
endfunction: maskN
```

The *zipWith()* library function takes a 2-argument function *f()* and two vectors *A* and *B*, applies *f()* to each corresponding pair of elements $A_j$ and $B_j$ , and returns a vector of all the results $f(A_j,B_j)$. Here, the *maskN()* function uses it to return a vector created by applying the *applyMask()* function to corresponding elements of a constraint group *g* and a mask *m*.

Given a group with some cells masked out, we can determine which values are still possible in the remaining cells by applying another higher-order library function, *fold()*:

```
// Determine all values which are possible in a group
function Cell#(order) possibles(Group#(order) g)
   provisos(Add#(1,_,SizeOf#(Cell#(order))));
   return fold(\| , g);
endfunction: possibles
```

Here we are folding the bitwise-OR operator ("|") over the cell values in the group. This is different from a standard reduction-OR, because the result is more than 1-bit wide -- it is equivalent to several reduction-OR operations applied in parallel across the bits of each cell value in the group. The *proviso* phrase is a

constraint that cells in the group and in the result are at least one bit wide; such constraints are statically checked by the BSV compiler.

Taking advantage of BSV's higher-order function capabilities leads to the development of a toolbox of (sometimes very small) functions that can be combined in different and powerful ways to define new hardware at a very high level of abstraction. For example, *the forced_in_intersection* tactic combines the two functions shown above to provide a simple definition (in 1 line of code) of a quite complicated idea:

```
// Tactic: If in a constraint group which intersects a constraint group
//         containing this cell (but which does not itself contain the cell),
//         a value does not appear in the portion outside of the
//         intersection, then it must appear within the intersection and
//         can therefore be eliminated from this cell.
// Arguments: g - constraint group intersecting a group containing the cell
//                but not containing the cell itself
//            m - mask of the portion of g outside of the intersection
method Cell#(order) forced_in_intersection(Group#(order) g, Mask#(order) m);
   return possibles(maskN(g,m));
endmethod: forced_in_intersection
```

This idea of using powerful function combination, which is sometimes found in very advanced software programming languages, is available in BSV and is fully synthesizable to hardware. In fact, there is zero "function-calling overhead" in the hardware, because the BSV compiler inlines and optimizes these functions extensively, so that the hardware you obtain is exactly what you would have gotten, and often better than if you had painfully and laboriously written out all this functionality by hand in legacy RTL.

## *Atomic-transactional, strongly-typed module interfaces*

In BSV, all module interfaces are *transactional,* i.e, instead of descending to signals, wires and timing diagrams of legacy RTL, we express interaction as a collection of *interface methods.* Each method encapsulates a *transaction*, i.e., a higher-level of interaction with the module. Moreover, these transactions support *atomicity,* i.e., they formally incorporate constraints on *simultaneity*—whether or not two transactions on a module can occur simultaneously (in the same clock), and *order*—if they can be invoked simultaneously, are there any ordering constraints due to the logical data flow from one transaction to the other inside the module? Atomicity ensures that the system state is always kept consistent—this eliminates, by construction, many of the protocol and "race condition" bugs that are only found (painfully) in post-design verification of legacy RTL. Maintaining consistent state requires control logic, which is automatically and correctly synthesized by the BSV compiler, whereas it must be laboriously (and often erroneously and unmaintainably) described explicitly by a designer using legacy RTL.

The methods in the interface of a BSV module, like functions, take arguments and return results, and these are strongly type-checked by the BSV compiler.

For example, a key module in BluDACu is the *mkTactics* module (in file `Tactics.bsv`). It encapsulates the solver's "bag of tricks", i.e., its repertoire of tactics. It contains a method for each tactic. The arguments to the tactic methods are typically the values of a constraint group (one row, column or box) along with the index of a distinguished cell within the group, and in some cases additional values from the grid. Each tactic method returns a bitmask of the values allowed for the distinguished cell in the constraint group according to the tactic. Its interface is shown below (the actual code also contains detailed comments explaining the tactics).

```
interface Tactics#(numeric type order);
   method Cell#(order) elim_other_singletons(Group#(order) g,
                                             Index#(order) n);
```

```
    method Cell#(order) process_of_elimination(Group#(order) g,
                                               Index#(order) n);
    method Cell#(order) forced_in_intersection(Group#(order) g,
                                               Mask#(order) m);
    method Cell#(order) repeated_2_set(Group#(order) g, Index#(order) n);
    method Cell#(order) hidden_pair(Group#(order) g, Index#(order) n);
endinterface: Tactics
```

Note that everything—the interface type itself, and method argument and result types, are parameterized by the order *O* of the puzzle. This ensures that *Groups* have the correct number of cells, *Masks* have the correct number of bits, and *Index* values are sized correctly to address all of the cells in a *Group*, etc. If either the module implementing a *Tactics* interface or the module using it has a mismatched type, the compiler will detect the error during type-checking.

Any *mkTactics* module implementing this interface is a purely abstract tactics module—it neither contains the actual Sudoku grid, nor does it refer to the grid in the method arguments or results. Every tactic is expressed as an abstract function from constraint groups and cells and other arguments to new cell values. It is left to an outer level module to extract constraint groups from the actual Sudoku grid, apply these tactics, and then to incorporate the new cell values back into the grid. This also makes it easy to incorporate a newly discovered tactic, because it is not entangled with the specifics of the grid.

## *Atomicity (correct-by-construction muxing and control logic)*

Each tactic is typically applied multiple times to different constraint groups. For instance, *elim_other_singletons* is applied 3 times for each cell: once to look for singletons within the row, once within the column, and once within the box.

Other tactics are applied in complicated ways that depend on the order of the puzzle. For instance, *forced_in_intersection* is applied at each cell once for each constraint group which intersects a constraint group containing the cell but which does not contain the cell itself.

In addition, many tactics operate on the same constraint groups. This leads to a situation where there is potential sharing of the logic to extract constraint groups as well as the logic to implement tactics. Achieving this sharing requires separating the logic to extract constraint groups and the logic to implement each tactic on a generalized constraint group.

Then, the correct constraint group and index number must be routed to the correct tactic logic each cycle. In legacy RTL this muxing of related control logic would be very complicated, but it has to be expressed by the designer. Further, imagine the nightmare of having to add a new tactic (e.g., to strengthen the solver). In BSV the correct muxing logic is automatically generated by the compiler. All that appears in the BSV source is the definition of the constraint groups:

```
Group#(order) current_row = getRow(grid, row);
Group#(order) current_col = getColumn(grid, col);
...
```

and calls to the methods of the tactics module (within the *apply()* function in file `Solver.bsv`):

```
tactic_result = tactics.elim_other_singletons(g,i);
```

The compiler analyzes which arguments are supplied to each method call in each cycle and inserts the correct muxing logic to connect the data to the tactic arguments. The muxing doesn't clutter the code or the designer's mind. Adding a new tactic is a trivial matter of just writing the new tactic itself—the compiler will take care of regenerating all the muxing and control logic implied by its interactions with other tactics.

## *Composable State Machines*

The solver is implemented as an FSM controlling the application of tactics to cells. Each time a tactic is applied to a cell, the result of the tactic method is AND'ed into the cell's bit mask. The solver simply sweeps across the grid of cells applying each tactic in turn to every cell.

BSV includes a rich sub-language for describing FSMs and composing small fragments of FSMs into larger FSMs. This has the dual advantage of allowing the design intent to be clearly expressed in the code, and making modification of the FSMs simple and error-free. For example, the top-level of the solver FSM is defined as:

```
Stmt tactic_sequence =
  seq
    while (True)
    seq
       action
          found_inconsistent <= False;
          all_cells_complete <= True;
          made_some_progress <= False;
       endaction
       for (r <= 0; r < fromInteger(valueOf(size)); r <= r + 1)
       seq
          for (c <= 0; c < fromInteger(valueOf(size)); c <= c + 1)
          seq
            apply(… singleton tactic to row r …);
            apply(… singleton tactic to column c …);
            apply(… singleton tactic to box containing row r and column c …);
               … similarly, apply tactics Elimination, Pairs, HiddenPairs …

            for (t <= 0; t < fromInteger(valueOf(order)-1); t <= t + 1)
            seq
               apply(… intersect tactic to other col …);
               apply(… intersect tactic to other row …);
               apply(… intersect tactic to other box in box rank …);
               apply(… intersect tactic to other box in box file …);
            endseq
          endseq
       endseq
       await(!results.notEmpty());
       if (found_inconsistent || all_cells_complete || !made_some_progress)
          break;
    endseq
  endseq;

FSM controller <- mkFSM(tactic_sequence);
```

The code is fully parameterized on the size of the puzzle. As written, it applies tactics sequentially, one at a time. It is a trivial matter to change the order in which tactics are applied. By changing the *seq* constructs to *par,* it is possible to apply tactics in parallel—of course, this could imply a lot more control logic because tactics may need simultaneous access to the Sudoku grid registers, depending on whether or not they are touching the same cells. Further, if a new tactic is added, it is trivial to incorporate its invocation into the FSM. These changes can imply *major* changes to the underlying hardware control logic, but it is all automatically regenerated by the BSV compiler.

As the solver operates, it monitors its own progress to detect if any cell's mask becomes completely empty (indicating an inconsistent puzzle), to detect if every cell is complete (indicating the puzzle has been solved), and to detect if every tactic has been tried at every cell without eliminating any candidates (indicating the puzzle is underconstrained relative to the tactics applied).

The generator is implemented as an FSM layer above the solver. It operates as follows. Starting with an empty grid, it runs the solver until it stops making progress, i.e., it gets "stuck" because no tactics make any progress; then, it "promotes" a randomly chosen cell in the partial solution to a known cell in the puzzle, randombly fixing one of its possible values as the solution. If the solver detects an inconsistency, the puzzle is discarded and the generator restarts. When the solver reaches a complete solution, the cells which were promoted to known values form the initial "given" cells in the generated puzzle. The key part of the code (in the file `Generator.bsv`) is:

```
Stmt try_to_generate =
        seq
            reset_puzzle;
            load_solver;
            while (True)
                   seq
                          run_solver;
                          if (!solver.isConsistent() || solver.isSolved())
                                                 break;
                   add_one_given;
                   endseq
        endseq;

Stmt generate_puzzle =
        seq
           while (True)
                   seq
                          try_to_generate;
                          if (solver.isConsistent()) break;
                          $display("discarding puzzle.");
            endseq
           $display("successfully generated puzzle:");
           displaySudoku(getGrid(puzzle));
        endseq;
```

Here *try_to_generate* is an FSM fragment used within the top-level *generate_puzzle* FSM. In turn, *try_to_generate* composes its own smaller FSM fragments *reset_puzzle, load_solver, run_solver* and *add_one_given*. Compare the clarity and ease-of-modification of FSMs specified this way with FSMs as traditionally expressed in RTL. This is in fact partially a "brute force" mechanism for generating puzzles; we could have a whole separate discussion on the limitations of this approach and how to improve it.

## *Flexibility for Refinement and Architecture exploration*

Compared to RTL, BSV's higher level of abstraction dramatically improves capabilities for step-wise refinement and architecture exploration. We have already mentioned several of these, but we repeat them here for emphasis:

- Write the solver first, and test it, and then write the generator to use the solver.
- Because of the parameterization, test it quickly with low-size puzzles before trying larger-size puzzles.
- Write a simple solver first with simple tactics, test it, and gradually add new tactics to strengthen the solver.
- Explore which tactics should be done in parallel vs. sequentially, using trivial changes to the FSM specifications, and see the effect on area, clock speed, and power consumption
- Easily

Many of these changes, if designing in RTL, would require such dramatic changes that they would be not be considered at all.

Finally, this example also illustrates how easy it is to incorporate C/C++ and other languages into the system using BSV. The GUI is in Tcl/Tk, and socket communications between the GUI and the solver/generator is in C.

## *Tour guide for the source code*

The complete source code distribution for BluDACu is available from Bluespec, Inc. We do not expect someone who has not yet studied or been trained in BSV to follow every nuance and detail in the source code, but with the explanations in this document plus the comments in the code, we hope you will be able to follow the general ideas and still appreciate the power of the language and see how far it goes beyond RTL. We also hope that it will inspire you to learn and use BSV!

You may wish to peruse the source files in the following ("bottom up") order:

| | |
|---|---|
| `TypeUtil.bsv` | A shorthand for the square of a parameter |
| `Sudoku.bsv` | Typedefs for Cells, Grids, Sudoku register grids, indexes, contraint groups, along with numerous "help" functions on these types |
| `Tactics.bsv` | All the tactic functions, plus their encapsulation into a *mkTactics* module that has a *Tactics* interface |
| `Solver.bsv` | The solver, encapsulated into the *mkSolver* module that has a *SudokuSolver* interface |
| `SolveTest2.bsv` | Small stand-alone testbench for solving one fixed puzzle of order 2 |
| `SolveTest3.bsv` | Small stand-alone testbench for solving one fixed puzzle of order 3 |
| `Generator.bsv` | The puzzle generator, encapsulated into the *mkGenerator* module that has a *SudokuGenerator* interface, and using the solver. |
| `GenerateTest2.bsv` | Small stand-alone testbench for generating one puzzle of order 2 |
| `GenerateTest3.bsv` | Small stand-alone testbench for generating one puzzle of order 3 |

The above are the key Bluespec SystemVerilog (BSV) files. The remaining files in the distribution, in BSV, C, and Tcl/Tk, are for the GUI and its connection to the BSV code. They are useful if you wish to see how to incorporate C code into a BSV system.

# Summary

So, what does this say about SoC design for now and the future? Although this example is a somewhat whimsical application of Bluespec SystemVerilog, it has some powerful meta-level messages.

*Expressive power: simplifying the Implementation of Complex Designs*

Hopefully you've got a taste of why Bluespec is a simpler and more scalable way of managing complex algorithms involving complex concurrency and interface protocols vs. RTL. The Bluespec model of atomic transactions, atomic-transactional interfaces, strong static constraints and static elaboration is also the only one significantly improving the design of control logic and complex datapaths. Such troublesome designs are:

- The hardest to get right. Managing complex concurrency with lots of shared resources (both local and across chip) is very hard to implement correctly. Think about the challenges properly managing back pressure, interface protocols, race conditions, deadlock/livelock conditions...
- The areas where most of the bugs (especially the subtle ones that later bite you) reside.

- The most prominent by far. One of our customers surveyed their IP development roadmap and found that designs with control logic and complex datapaths represented 90% of their planned projects.

Since it keeps you 100% in control of architecture and micro-architecture, Bluespec enables implementations that retain the quality of results of RTL, while at a much higher level of abstraction.

Somewhat surprisingly, our BSV solver, parameterized and synthesizable into hardware, had fewer lines of source code than an unparameterized and of course unsynthesizable comparable solver (same strategy, same tactics) we wrote in C (both about 800 lines)! We have seen similar comparisons for more serious applications like video and wireless codecs which are often originally written in C—the BSV code is often shorter, clearer, and of course synthesizable.

*Creating IP blocks that enable Extreme Reuse*

Bluespec enables extreme reuse through the creation of IP that is highly parameterized to support many optional capabilities, easy to customize correctly when changes are required, and interfaces that ensure proper connectivity and communication protocol automatically.

*Testing Software Early with Accurate Hardware*

The BluDACu generator and solver were developed in less than one man-week – and, the surrounding testbench/GUI took one week in addition. With Bluespec, you can develop highly complex models and implementations significantly faster and with fewer errors than with SystemC or RTL. And, because Bluespec keeps models and implementations in a single environment, you don't have two environments to maintain. As illustrated with BluDACu, the hardware runs directly with C/C++ software, testbenches and models. This means you get a single environment to develop and maintain, you are up and running in a fraction of the time of other environments, and software gets to run with accurate designs.

Not only do you get accuracy, but you also get speed. Bluespec simulations in SystemC or Bluesim run natively at high-speeds. And, if that's not fast enough, Bluespec's high-level constructs are fully synthesizable, which means you can run models and implementations on emulators or FPGA prototypes. While this may be a challenge with an error-prone RTL design, the correctness of a Bluespec design means you can safely do this significantly earlier.

*Changing the hardware-software partitioning tradeoff*

Complex IP blocks are often "relegated" to software because creating a hardware implementation is seen as too complex and risky, even though the software implementation is likely to be significantly slower and/or consume significantly higher power. The expressive power, robustness and synthesizability of Bluespec changes this calculation.

**This is where Bluespec shines. Everything else is just RTL.**

**Can you afford not to take a closer look?**