Achieving Timing Closure with Bluespec SystemVerilog

White Paper
August 18, 2004
© 2004, Bluespec, Inc. All rights reserved

Introduction

Achieving timing closure becomes increasingly difficult with more aggressive technologies at higher clock speeds. Today's designer, typically coding in RTL using Verilog or VHDL, is accustomed to making fine-grained, controlled changes to the RTL, with reasonably predictable consequences as it is taken through synthesis and physical design tools, so that the design's timing improves continuously until the target is met.

Of course, changes to achieve timing must not compromise the correctness of the design, which is usually established through extensive verification. Thus, the designer typically attempts to make changes that are as localized and small as possible, employs formal tools such as equivalence checkers, and runs regressions on the verification suite.

A common concern about high-level synthesis is a perceived loss of control in this process of timing closure. This is a legitimate concern because, in most traditional approaches to high-level synthesis, e.g., compiling from a "behavioral" C-like description into hardware, the semantic model of the source (sequential code) is so different from the semantic model of hardware that the designer loses predictability. It is hard for the designer to imagine what should be changed in the source to effect a particular desired timing improvement in the hardware. Further, small and apparently similar changes to the source can result in radically different hardware.

Bluespec SystemVerilog (BSV) uses a very different approach to high-level synthesis, where the designer retains control. The semantic model of the source (guarded atomic state transitions) maps very naturally into the semantic model of clocked synchronous hardware. The designer can make controlled changes to the source, with predictable effects on timing. Further, because of the extensive static checking in BSV, these changes can be more dramatic than localized tweaking, thereby allowing the designer to achieve timing goals sooner without compromising correctness.

In this document we describe some of the techniques used by the BSV designer to achieve timing closure. Most of them will be quite familiar to the RTL designer, but BSV makes some of them much easier to accomplish, and much more likely to be correct:

- Adding a pipeline stage to an existing pipeline
- Adding a pipeline stage where pipelining was not anticipated
- Spreading a calculation over more clocks (longer iteration)
- Moving logic across a register stage (rebalancing)

- Restructuring combinational clouds for shallower logic
- Incorporating hand-optimized logic
- Controlling downstream tools (synthesis, PD, etc.)

After a general section on the transparent correspondance between BSV source and the generated Verilog, these techniques are explored in more detail in the sections that follow.

Transparent correspondance between BSV source and generated RTL

The BSV compiler translates BSV source into Verilog 95 RTL. Downstream tools (simulators, synthesis, physical design, etc.) are standard tools that operate on Verilog RTL. Unlike other approaches to high-level synthesis, there is a transparent correspondance between the BSV source and Verilog 95 RTL. This makes it straightforward to relate control and observations in the downstream tools to the BSV source, so that issues can be easily identfied and fixed directly in the BSV source.

The mechanism of module instantiation, resulting in a module instance hierarchy, are identical in BSV and in Verilog. The BSV designer decides which modules should be inlined and which modules should be retained. The names of modules and module instances are carried through from the BSV source to Verilog. Thus, designer will have no problem in telling downstream tools about modules, nor in interpreting reports about modules from downstream tools.

State elements (like registers) are carried through untouched. The BSV compiler neither adds nor subtracts state elements. State elements are not moved across module boundaries. The names of state elements are carried through to the generated code.

Module interface wires in the generated Verilog have a direct correspondance with module interface method names, argument names and types in the BSV source.

BSV has a “probe” mechanism by which the compiler can be forced to retain an intermediate value, together with its name, into the generated code.

Much of the logic in the generated RTL will therefore be easy to relate to the BSV source. The scheduling logic introduced by the BSV compiler consists of ANDs for enable signals and MUXs for data paths, in predictable forms.

For all these reasons, the output of the BSV compiler is not at all mystical. In a very short time, the BSV designer quickly develops a simple and reliable mental picture of the output RTL for the BSV he is writing. This transparency makes it straightforward to understand what needs to be fixed in the source in order to improve timing.

As an example, here is an excerpt from a timing report from an FPGA synthesis tool applied to the generated Verilog for a BSV implementation of the Pong video game:

```
Timing Detail:
 - - - - - - - -
 All values displayed in nanoseconds (ns)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 Timing constraint: Default period analysis for Clock 'CLK'
 Delay:               22.994ns (Levels of Logic = 25)
   Source:            toplevel_the_ball_the_change_y_Q_OUT_8
   Destination:       toplevel_the_ball_the_ball_y_b_Q_OUT_10
   Source Clock:      CLK rising
   Destination Clock: CLK rising
   Data Path: toplevel_the_ball_the_change_y_Q_OUT_8 to
toplevel_the_ball_the_ball_y_b_Q_OUT_10
                                Gate     Net
     Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
     - - - - - - - - - - - - - - - - - - -   - - - - - -
      FDCE:C->Q             2   1.085   1.206
toplevel_the_ball_the_change_y_Q_OUT_8 (toplevel_the_ball_the_change_y_Q_OUT_8)
      LUT4:I0->O            2   0.549   1.206  toplevel_X_dm1229 (CHOICE877)
      …
      … <path details omitted> …
      …
      FDCE:D                       0.709
toplevel_the_ball_the_ball_y_b_Q_OUT_10
     - - - - - - - - - - - - - - - - - - -
     Total                     22.994ns (10.880ns logic, 12.114ns route)
                                        (47.3% logic, 52.7% route)
```

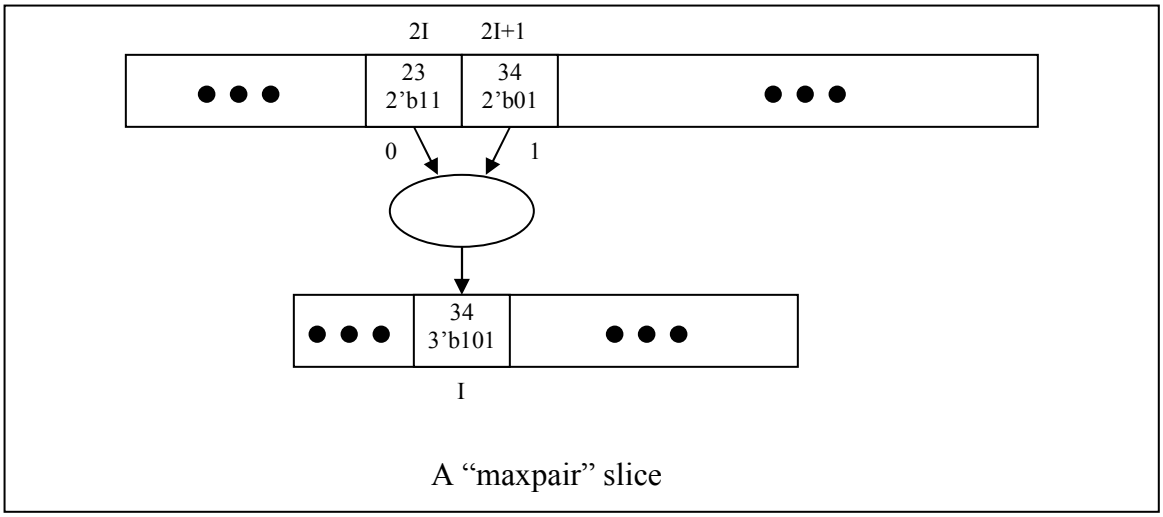
The designer familiar with BSV will immediately understand that this path is in a module `the_ball` inside the `toplevel` module and, further, that it is between the `change_y` and `ball_y_b` registers, and can proceed to solve the problem (the solution used in this case is described in the Section “Spreading a calculation over more clocks”).

Adding a pipeline stage to an existing pipeline

When a design is already pipelined, it may be possible just to split a stage into two stages, for faster speed (but with an increased latency). BSV makes it particularly easily to implement this, and to do it safely (correctly).

Consider the computation of the maximum of an array of N numbers, together with its index J in the original array (0 <= J < N), for example to locate the priority and port number of the highest-priority port in a list of ports. This can be done with a binary tree-reduction. Such a tree can be built with a series of reusable “slices”.

A slice’s input is an array of M items, and its output is an array of M/2 items, where each item is a number together with an index. For the I’th output, it compares the 2I’th and 2I+1’th inputs. The output item contains the larger input number and its index extended with a higher-order bit that is 0 or 1 depending on which input item was selected. The following figure illustrates:



A "maxpair" slice

If we stack together log(N) of these slices, we obtain a fabric containing a binary tree of comparisons. The initial slice will have N items, each with a 0-bit index, and the last slice will have 1 item with a log(N)-bit index, i.e., the unique item in the last slice is our desired result.

Note: one could have no registers in the entire tree, or registers in every slice, or only on occasional slices; this corresponds to choosing different "pitches" for the pipeline, and should be easily changeable.

BSV makes it extremely easy to express this kind of structure. Here is an excerpt from the code for a 128-item maxtree (i.e., log(128)=7 comparison slices):

```
pipe( passed(wrapStartData),
pipe( passed(maxpair),
pipe( passed(maxpair),
pipe( buffered(maxpair),
pipe( passed(maxpair),
pipe( passed(maxpair),
pipe( buffered(maxpair),
pipe( passed(maxpair),
      passed(head, output_fifo)  ))))))));
```

The pipe() function stacks two slices to form a composite slice. The passed(wrapStartData) function creates the initial slice, which attaches a 0-bit index to each of its inputs. Each passed(maxpair) function performs the comparison illustrated in the figure, with no buffering. Each buffered(maxpair) is similar, but it also has a register buffer on input. The final passed(head, output_fifo) extracts the unique element of the 1-element array, and pushed it into an output fifo. Thus, this implements a 3-stage pipeline: two combinational maxpair slices, a buffer, 3 combinational maxpair slices, a buffer, and 2 combinational maxpair slices.

It is now trivial to add or remove a level of pipelining. Changing a passed() to a buffered() adds a pipeline stage. Changing a buffered() to a passed() does the reverse. Note that when buffering is moved to a different level in the reduction tree, the number of elements in the buffer changes (by 2x or ½ x) and, further, the size of each element changes (because the index grows or shrinks by 1 bit).

All these resizings are automatically checked and managed by the BSV compiler, so that these changes can be accomplished with full confidence that correctness is not affected.

What if the output_fifo is full? The BSV compiler automatically produces logic that stalls the pipeline if the output cannot consume a value. Again, when we change the pipeline structure by moving the buffering up or down, this stall logic is automatically regenerated, correctly.

If we had used qbuffer(maxpair) instead of buffer(maxpair), we would get a pipeline with interlocks, i.e., the pipeline registers would have full/empty bits and a stage could only advance its data if its output buffers were empty. Again, for such an elastic pipeline, the BSV compiler completely generates all the pipeline control logic for this purpose, even if these buffer stages are repositioned up or down the pipe.

It is important to emphasize that none of these are special cases in BSV, i.e., they naturally fall out of the high-level semantics of guarded atomic actions. All the functions mentioned above-- pipe(), passed(), buffered(), qbuffered()-- are just library functions written in BSV and can be written by any designer.

In summary: if pipelining has been part of the design from the very beginning, it is trivial to rapidly and safely adjust the pipeline structure with fewer or more stages to meet timing optimally.

Adding a pipeline stage where pipelining was not anticipated

If splitting a stage into two is necessary at a later moment in the design process, this can be done very straightforwardly in BSV, taking advantage of the compiler's power of type inference to minimize the bureaucratic overhead of declaring intermediate variables.

Consider the following rule that dequeues four values from four input FIFOs A, B, C and D, performs some calculations on those values, and enqueues the two results on two output FIFOs X and Y.

```
rule calc;
   let a = fifoA.first; fifoA.deq;
   let b = fifoB.first; fifoB.deq;
   let c = fifoC.first; fifoC.deq;
   let d = fifoD.first; fifoD.deq;

   let x = (a * b) * (c * d);
```

```
   let y = (a + b) * (a - b);

   fifoX.enq(x);
   fifoY.enq(y);
endrule
```

Suppose that this calculation must be split into two stages (in practice, of course, this would not be necessary for such a simple calculation, but this is to illustrate the approach). We proceed step by step, very cautiously (in practice some of these steps would no doubt be run together).

First we introduce new local variables, for the values we decide to buffer. We introduce them in pairs, defining the first ones from the first half of the calculation, and using the second ones in the second half. Note how it is often unnecessary (as in this case) to worry about the types of the variables introduced.

```
rule calc;
   let a = fifoA.first; fifoA.deq;
   let b = fifoB.first; fifoB.deq;
   let c = fifoC.first; fifoC.deq;
   let d = fifoD.first; fifoD.deq;

   let aPLUSb = a + b;
   let aMINUSb = a - b;
   let aTIMESb = a * b;
   let cTIMESd = c * d;

   let aPLUSb1 = aPLUSb;
   let aMINUSb1 = aMINUSb;
   let aTIMESb1 = aTIMESb;
   let cTIMESd1 = cTIMESd;

   let x = aTIMESb1 * cTIMESd1;
   let y = aPLUSb1 * aMINUSb1;

   fifoX.enq(x);
   fifoY.enq(y);
endrule
```

Next we bundle up these variables into a structure, and unbundle them again. We have chosen to use a tuple (a pre-defined struct type), as that is simplest, and the positional matching is sufficiently reliable in this very localized usage; if preferred, however, a structure with named fields could be declared instead.

```
rule calc;
   let a = fifoA.first; fifoA.deq;
   let b = fifoB.first; fifoB.deq;
   let c = fifoC.first; fifoC.deq;
   let d = fifoD.first; fifoD.deq;

   let aPLUSb = a + b;
   let aMINUSb = a - b;
   let aTIMESb = a * b;
   let cTIMESd = c * d;
```

```
   let t0 = tuple4(aPLUSb, aMINUSb, aTIMESb, cTIMESd);
   match {.aPLUSb1, .aMINUSb1, .aTIMESb1, .cTIMESd1} = t0;

   let x = aTIMESb1 * cTIMESd1;
   let y = aPLUSb1 * aMINUSb1;

   fifoX.enq(x);
   fifoY.enq(y);
endrule
```

Note that the RTL produced from all the versions so far is identical (and may easily be checked by a program such as "diff"). We are now ready to introduce a buffer register.

```
let r();
mkReg#(unpack(0)) the_r(r);

rule calc;
   let a = fifoA.first; fifoA.deq;
   let b = fifoB.first; fifoB.deq;
   let c = fifoC.first; fifoC.deq;
   let d = fifoD.first; fifoD.deq;

   let aPLUSb = a + b;
   let aMINUSb = a - b;
   let aTIMESb = a * b;
   let cTIMESd = c * d;

   r <= tuple4(aPLUSb, aMINUSb, aTIMESb, cTIMESd);
   match {.aPLUSb1, .aMINUSb1, .aTIMESb1, .cTIMESd1} = r;

   let x = aTIMESb1 * cTIMESd1;
   let y = aPLUSb1 * aMINUSb1;

   fifoX.enq(x);
   fifoY.enq(y);
endrule
```

And we are done. Note that although the assignment to the register appears above the read-out, in fact they happen simultaneously; so the values used in the second part of the calculation will be those stored in the previous cycle.

This example demonstrates (a) the ease of preparing to add the new register, and the availability of a check that it has not changed the output RTL; and (b) that the final addition of the register is very straightforward and therefore reliable.

Spreading a calculation over more clocks (longer iteration)

Sometimes it is acceptable, in order to meet timing, to spread a calculation over several cycles without pipelining, i.e, a longer latency is acceptable. In this case it is easy to define a finite-state-machine to handle the calculation.

Consider the following action taken from an implementation of the Pong video game (the details are not important, just the fact that there is a lot of computation within one action):

```
// steps in updating the ball
Action updateBall =
  action
    ball_vx<= (posX(ball_vx)== ball_dx ? ball_vx : negate(ball_vx));
    let ball_vy2 = (posY(ball_vy)== ball_dy ? ball_vy
                  : negate(ball_vy));

    // random velocity change
    Bit#(2) r2;
    YCoord randV;
    r2 = truncate(random);
    case (r2)
      2'b00    :  randV = 1;
      2'b01    :  randV = -1;
      default  :  randV = 0;
    endcase
    let ball_vy3 = ball_vy2 + shiftY(change_y, 3)
                  + ((change_y != 0) ? randV : 0);

    ball_vy <= limitY(ball_vy3);

    ball_x <= ball_x + ball_vx; ball_y <= ball_y + ball_vy;

    ball_x_r <= ball_x + ballWidthC;
    ball_y_b <= ball_y + ballHeightC;
  endaction
```

This has some fairly long combinational paths, and is unlikely to meet timing. It turns out, however, that the results of the calculation are not needed for several clock cycles after it is initiated; so an iterative (FSM) implementation is acceptable. The fragment above is transformed as follows:

```
// steps in updating the ball
Stmt updateBallStmt =
seq
   action
      ball_vx<= (posX(ball_vx)== ball_dx ? ball_vx : negate(ball_vx));
      ball_vy<= (posY(ball_vy)== ball_dy ? ball_vy : negate(ball_vy));
   endaction
   action
      // random velocity change
      Bit#(2) r2;
      YCoord randV;
      r2 = truncate(random);
      case (r2)
         2'b00    :  randV = 1;
         2'b01    :  randV = -1;
         default  :  randV = 0;
      endcase
      ball_vy <= ball_vy + shiftY(change_y, 3)
                 + ((change_y != 0) ? randV : 0);
```

```
   endaction
   action
      ball_vy<= limitY(ball_vy);
   endaction
   action
      ball_x <= ball_x + ball_vx; ball_y <= ball_y + ball_vy;
   endaction
   action
      ball_x_r <= ball_x + ballWidthC;
      ball_y_b <= ball_y + ballHeightC;
   endaction
endseq;

FSM updateBallFSM();
mkFSM#(updateBallStmt) the_FSM(updateBallFSM);

Action updateBall = updateBallFSM.start;
```

The single action has become a sequence of actions (which is of the Stmt type). This sequence becomes a parameter of the module constructor mkFSM, which constructs an FSM to run the calculation. Finally the action to perform the calculation is now simply an invocation of the "start" method of the FSM's interface. (We have also realized that there is no need to define new registers for ball_vy2 and ball_vy3, as these values can be held temporarily in ball_vy.) The "seq" construct, the Stmt and FSM types, and the mkFSM constructor are all standard parts of BSV, and are described in the Reference Guide.

In the Pong video game implementation, this transformation improved timing from 22.994 ns (43.49 MHz) to 18.093 ns (55.270 MHz). The goal was 50 MHz.

This example shows how it is straightforward to take a single-cycle complex action and break it up into a multi-cycle action sequenced with an FSM. The split action has a direct correspondence to the original composite action, and the FSM construction is completely automated. Both these features, again, make it safe to perform this transformation.

Moving logic across a register stage (rebalancing)

When there is too much logic in a stage of the computation, it may be possible to move a part of that computation into the previous or following stage. In BSV, each stage is expressed using a rule, and so this transformation corresponds to moving computation from the action of one rule into the action of another rule. This is very similar to what one would do in Verilog, but more reliable because of the higher level of notation and stronger type-checking.

Consider the following fragment:

```
rule foo;
  … compute y …
  x <= f(y);
```

```
endrule

rule bar;
  … <= x;
endrule
```

The first rule computes y, applies a function f() to it, and stores the result in register x, which is read in the second rule. If the first rule is not meeting timing, it may be possible to transform this fragment into:

```
rule foo;
  … compute y …
  x <= y;
endrule

rule bar;
  … <= f(x);
endrule
```

The application of function f(), which represents some logic, has been moved into the second rule. Similarly, logic can also be moved backward into a temporally previous rule. As in Verilog, such "retiming" transformations must of course be done with care. For example, if the register x was read elsewhere (e.g., in a rule or method condition, or another rule's action) that may preclude direct application of this transformation. Sometime, many computations have to be moved simultaneously in order to achieve this retiming. But, modulo those considerations, the BSV compiler will perform its usual static checks on the modified code. For example, in the above excerpt, the type of the data held by register x has changed from the output type of f() to the input type of f(), and BSV will ensure that this is so. These automated checks are particularly valuable when retiming involves multiple simultaneous code movements.

Restructuring combinational clouds for shallower logic

When there is too much combinational logic in an action, certain transformations can remove layers of logic. A classic example of this is the three input adder.

```
rule foo;
  sum <= a + b + c;
endrule
```

A standard trick is to run all three inputs through a carry save adder stage, which is two xor levels deep, and then run through a final adder stage. This removes almost a full adder stage from the timing of this path.

```
rule foo;
  cr = (a ^ b ^ c) << 1;
  sm = a&b | b&c | a&c;
  sum <= cr + sv;
endrule
```

This kind of change is similar to what one would do in Verilog; BSV's advantage here is mainly in stronger type-checking, ensuring more safety in making these changes. However, note that synthesis tools have gotten much smarter about these trade-offs in recent years.

It is also easier to carry out algebraic manipulation, which can sometimes help achieve timing. For example, in our trivial example to calculate the value (a+b)*(a-b), the addition and subtraction is performed in the first stage, and the multiplication in the second. Suppose it proved desirable to move the multiplication to the first stage. It is easy to transform:

```
let aPLUSb = a + b;
let aMINUSb = a - b;

r <= tuple4(aPLUSb, aMINUSb, aTIMESb, cTIMESd);
match {.aPLUSb1, .aMINUSb1, .aTIMESb1, .cTIMESd1} = r;

let y = aPLUSb1 * aMINUSb1;
```

to

```
let aSQUARED = a * a;
let bSQUARED = b * b;

r <= tuple4(aSQUARED, bSQUARED, aTIMESb, cTIMESd);
match {.aSQUARED1, .bSQUARED1, .aTIMESb1, .cTIMESd1} = r;

let y = aSQUARED1 - bSQUARED1;
```

In some cases, the designer can use knowledge about the particular data to express further optimization. For example, knowing that certain address values will be doubleword-aligned (i.e., bottom 3 bits will be zero), can reduce the width of an adder and therefore shrink the carry chain. Again, this kind of change is similar to what one would do in Verilog; BSV's advantage here is mainly in stronger type-checking, including bit-width checking, ensuring more safety in making these changes.

In general, in BSV, producing shallower logic for a rule involves reducing the data path (the explicit computations mentioned in the action of a rule), and/or reducing the control (the computations mentioned in the rule condition and the logic introduced by the compiler to schedule rules).

To reduce the data path, there are several approaches which may be tried:

- Reduce the width of the path, to reduce logic that chains bitwise over the path.
- Change the logic as described in BSV functions and expressions, as in the previous examples.

To reduce the control paths, the first possibility is to simplify the logic for computing rule predicates, i.e., the expressions for rule conditions and method implicit conditions. Any

logic which can be moved from the predicates to the actions (rule or method bodies) can help move logic out of critical control paths. If predicates are still too large, one can even pipeline the computation of the predicate, by adding more rules.

When multiple rules conflict, the BSV compiler introduces scheduling logic and muxing in front of shared resources. To reduce muxing, one can try to make rules conflict less. If two rules do not conflict, but a third rule which conflicts with them both causes a relationship, one has the option of breaking the third rule into parts, or merging its parts with the other rules. Thus, breaking rules apart is a way of controlling the scheduling logic.

Incorporating hand-optimized logic

A design can contain a mixture of modules, some written in BSV and some written directly in Verilog. BSV modules can instantiate Verilog modules, and vice versa. It is fairly straightforward to "import" a Verilog module into BSV. In fact, even if the entire design is written in BSV, this mechanism is used by BSV libraries to implement all primtive elements in BSV, such as registers and memories. It is also typically used to incorporate existing Verilog IP.

This import mechanism can, in the worst case, be used by the designer to import a hand-optimized Verilog module into the design.

(There is another verification benefit from importing Verilog like this. Behavioral constraints on Verilog interfaces are often expressed only informally in accompanying text or in timing diagrams. Bluespec wrappers force these constraints to be expressed formally in the interface types and scheduling information; these are then machine-checked by the compiler on every use of the module.)

Controlling downstream tools (synthesis, PD, etc.)

There are many standard tricks in controlling downstream tools in order to achieve timing closure.

- Giving more realistic, fine-grained timing constraints to the synthesis tool.
- Adjust constraints to increase effort made by synthesis tool's heuristics in reaching a solution.
- Adjust fine-grain constraints concerning load for certain outputs, inducing stronger (faster) buffers.
- Relax timing constraints on non-critical paths, thereby improving effort on critical paths.
- Use more conservative wire load constraints.
- Feed real wire loads, from layout, back into synthesis constraints.

All these tricks continue to be possible in the BSV flow, because of the transparent correspondance between the BSV source and the generated Verilog, which was discussed

at the start of this document. It is easy both to interpret reports from these downstream tools, and to apply the necessary controls in the above list.

Summary

Bluespec SystemVerilog is unique amongst high-level synthesis approaches in that, while significantly raising the level of abstraction and correctness-by-construction, it retains the traditional hardware model of cooperating FSMs, and retains a high degree of transparency and predictability between the source and the generated RTL. For these reasons, the designer can continue to use tried-and-true techniques for improving timing, with the additional significant advantages that:

- The high level of the language makes change much easier.
- The extensive static checking ensures that changes do not destroy correctness.
- The high level and correctness-preservation properties enable more dramatic attacks on the timing problem than "local hill-climbing".