Reliable Design with Multiple Clock Domains

Ed Czeck, Ravi Nanavati and Joe Stoy
Bluespec Inc.
Waltham MA 02451, USA

Abstract: **We present a set of guiding principles for the management of multiple clocks domains in the design of a high-level hardware description language. Our motivation of the requirements is based on typical design problems; the solutions are based on common engineering practices and appropriate language abstractions. We include examples, and conclude with some comments based on a design experience.**

1. Introduction

Hardware designs these days typically make use of several clocks. This is partly to save power (by gating the clock to a part of the circuit temporarily not in use, and by ensuring that parts of the design are not run unnecessarily fast, both of which reduce the design's "dynamic" power consumption), and also to allow the design to communicate with parts of the external environment running asynchronously. Moreover, designs are increasingly becoming "systems on a chip" ("SoC"s); this design methodology brings together various blocks (IPs), possibly from difference vendors, and often each block has its own clocking requirement. These different requirements may arise simply because each block was designed independently; but it might be because different blocks are constrained by different standards (for example for external buses, or audio or video I/O), each with its own clocking regime.

Where signals cross clock domain boundaries, the normal design conventions of digital logic break down. Special precautions must be taken to ensure that the signals get across correctly; and care must be taken that there are no accidental crossings which neglect such precautions. A good hardware description language, particularly one which claims to operate at a high level of abstraction, should have features which make it natural for the designer to construct circuits which correctly observe these constraints.

2 Principles

A hardware description language, notation or system which supports multiple-clock-domain designs should ideally have the following characteristics.

1. The aim should be to make the simplest situations trivial, other simple situations easy, and all situations expressible in a sensible way.

2. Thus in a design, or part of a design, with just one clock domain, clock handling should be completely implicit. Each instantiated module needs to be connected to "the" clock, and having to say so explicitly adds unnecessary clutter.

3. *Clock* should be a datatype of the language (each value of which will include an oscillator, as well as an optional gating signal); the type system should ensure that clock oscillators are never confused with level-sampled signals.

4. The system should keep track of which signals are clocked by which clocks, and ensure that no signal crosses a clock-domain boundary without the use of appropriate synchronizing logic.

5. For efficiency's sake, the system should be able to recognize when two clocks are driven by the same oscillator (that is, they differ only in gating); this should be exploited to simplify domain-crossing logic between them.

6. Many groups of designers have their own preferred designs for domain-crossing logic; the system should allow the automatic deployment of such designs when appropriate (though also providing default designs for use when these are not available).

In the remainder of this paper we address, by way of example, how these principles are addressed in Bluespec SystemVerilog (BSV). In particular we shall first illustrate how they apply to a particular design, and then describe further features available in BSV which were not exploited in that design.

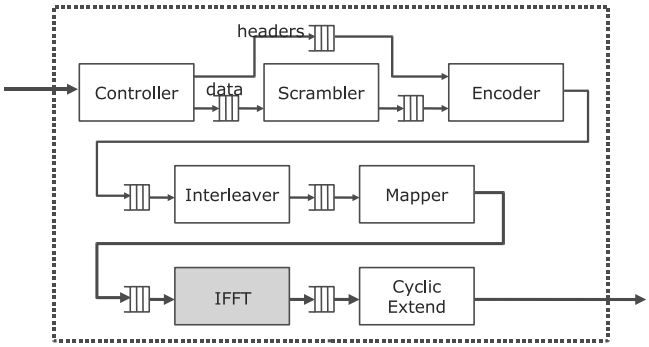

Figure 1. 802.11a Transmitter Design

3. A Brief Introduction to BSV

Bluespec SystemVerilog is a strongly typed, high-level behavioral hardware description language, intended to raise the level of behavioral abstraction while still allowing automatic synthesis into efficient hardware. As with Verilog, designs are structured into modules. The internal behavior of a module is specified by *rules* (instead of always-blocks), which have a guard and an action part. Modules communicate with their environment by the *methods* of their interfaces, which may be invoked by rules in other modules. All methods have "ready" conditions, which become "implicit" conditions" of rules which invoke them—only if all the conditions, explicit and implicit, of a rule are satisfied may the rule be executed. Taken together, the rules of a design have the semantics of a state-transition system (or term-rewriting system), and execute atomically. The Bluespec compiler generates logic which schedules as many rules as possible in each clock cycle; but the overall effect of each cycle is exactly the same as if the rules executed one at a time in some order—this greatly simplifies the analysis of the design's correctness. (It avoids many race conditions, replacing them with compile time warnings that two rules could not be scheduled simultaneously because of a resource conflict; it also means that proofs of correctness can analyze each rule separately, without worrying about their interaction.)

A slightly fuller description is provided in the Appendix; the reader is also referred to the Language Reference Guide [3] for a complete account.

4. The Example Design

For our example we shall use the transmitter design discussed in [7] for the 802.11a wireless protocol [10] (see Fig. 1)—we are in fact addressing the suggestion in the final sentence of that paper. One of the versions given in [7] for the IFFT brick, optimized for minimum hardware area, takes 48 clocks to process a single packet; the other bricks process a packet in just one clock, except that the interleaver brick might input as many as four packets for every one output. Thus, if the IFFT brick is clocked at frequency f, it makes sense to consider clocking the Mapper and Cyclic-Extend bricks at frequency $f/48$, and the remaining bricks at $f/12$. Let us consider how to achieve this in BSV.

4.1. Clocks—The Simplest Case

The simplest case is one where a design, or part of a design, uses just one clock. In line with the principles outlined above, in this case the clock is not mentioned in the BSV text at all. Each module in the RTL generated by the Bluespec compiler will have a clock port, called *CLK*; this is connected to any module instantiated within that module. At the lowest level of the generated module hierarchy, this clock is connected to the flip-flops, registers and other primitive state elements.

The advantage of an implicit clock is that designer is spared the tedium of mentioning the clock and its explicit use in the generation of the flops, registers, and other primitives. The use of standard primitives based on common engineering practice (in this case positive-edge-triggered flops) further reinforces that abstraction for the common case, while the complete Bluespec SystemVerilog language allows for a full range of flexibility.

In our example this means that none of the sub-modules will refer to any clock—the only place these are mentioned is at the top level of the transmitter.

4.2. The Type "Clock"

In designs with more complicated clocking arrangements, clocks are mentioned explicitly. A clock is a value of type *Clock*. It enjoys most of the general "first-class citizenship" properties of other BSV values: it may be passed as an argument, or returned as the result, of a function; it may be a field of an interface. Clocks may be selected by conditional expressions, and two clocks may be tested for equality. We shall see examples of these capabilities as we develop our 802.11a example.

Note that some of these features are available only at compile-time, not in the running synthesized hardware. For example, two clocks cannot be tested for equality in the hardware itself; and though dynamic selection of clocks is allowed, it must be handled by a special library multiplexer module (in case the downstream tools require specialized logic, rather than the normal combinational gates used for ordinary conditionals).

4.3. The Features Needed for the Example

The Default Clock Every module instantiation has a "default clock" which is used, unless otherwise specified, to clock any interior instantiation. If another clock is desired instead, it may be specified explicitly (using a "*clocked_by*" argument, as in the examples below). It is also usually necessary to specify a non-default reset signal too, as each reset must be associated with a particular clock (this is true even for asynchronous resets, as even they are synchronized with a clock when coming out of reset).

New clocks These arise in several ways. They are often passed in as arguments to the design, having been generated by external electronics. They may also be generated in IP cores (for which the BSV designer will provide a wrapper) to handle external interfaces (such as a SPI-4) which provide their own clocks. For our present example we shall assume that the extra clocks are supplied as arguments to the *mkTransmitter* module; we shall discuss other clock-generation possibilities later. Note that this choice implies that, even though the frequencies involved are related, the tool can make no assumptions about their synchronization—we shall look at another possibility in Section 5.4 below

Synchronizers When data values are to cross between different clock domains, we must assume that the oscillators of the clocks concerned are not synchronized in any way (unless we have any information to the contrary—we shall consider some such situations later). In general domain crossing requires special logic (a circuit known as a *synchronizer*) to prevent data being corrupted by metastability when the clocking edges of the two clocks become close but not coincident. The tool ensures that no domain crossing occurs without the use of such a circuit and that the logic used connects the domains appropriately. As with any design language, the tool can ensure only that a given module is used properly: it cannot verify that the module was correctly written or correctly selected in the first place.

The design of synchronizers has matured over the years, and there are now common design rules [6, 12] dealing with the various issues which arise (such as the problems of data-skewing and hand shaking and their solutions using Gray codes[2] and multi-phase protocols[5]). There are even commercial tools [1, 13] which automatically verify that clock domain crossing follows design rules. Bluespec provides a library containing several data synchronization primitives, all based on these common engineering practices. Groups of designers are, however, free to replace this library, or part of it, with their own synchronizer library (written in Verilog)—the tool will continue to ensure that synchronizers are not accidentally omitted. We shall discuss the Bluespec repertoire more later; for the present example the appropriate synchronizer is a FIFO, with the *enq* and *deq* methods clocked by different clocks.

```
(* synthesize *)
module mkTransmitter(Transmitter#(24,81));
  let controller   <- mkController;
  let scrambler    <- mkScrambler_48;
  let conv_encoder <- mkConvEncoder_24_48;
  let interleaver  <- mkInterleaver;
  let mapper       <- mkMapper_48_64;
  let ifft         <- mkIFFT_Pipe;
  let cyc_extender <- mkCyclicExtender;

  rule controller2scrambler;
     connect(controller.getData, scrambler.fromControl);
  endrule
  ...
endmodule
```

Figure 2. Part of the Transmitter's Original Code

```
(* synthesize *)
module mkTransmitter(Clock fastClk, Clock outClk,
                                Transmitter#(24,81) ifc);
   let currentReset <- exposeCurrentReset;

   let fastReset <- mkAsyncResetFromCC(2, fastClk);
   let outReset  <- mkAsyncResetFromCC(2, outClk);

   let controller   <- mkController;
   let scrambler    <- mkScrambler_48;
   let conv_encoder <- mkConvEncoder_24_48;
   let interleaver  <- mkInterleaver;
   let mapper       <- mkMapper_48_64;
   let ifft         <- mkIFFT_Pipe
               (clocked_by fastClk, reset_by fastReset);
   let cyc_extender <- mkCyclicExtender
                (clocked_by outClk, reset_by outReset);

   rule controller2scrambler;
      connect(controller.getData, scrambler.fromControl);
   endrule
   ...
endmodule
```

Figure 3. Part of the New Code

4.4. Putting the Example Together

An excerpt from the original version of the transmitter's top level is shown in Fig. 2. *connect* is a function (not shown) which takes two methods as argument, and produces an action which transfers data from one to the other. There are also other rules (not shown) to effect the other connections, and the definitions of the methods of the transmitter's overall interface. The same part of the new version is shown in Fig. 3. We assume that the module as a whole is clocked by the clock used for data input to the pipeline; the module's default clock will thus serve for the first few sub-modules, and their instantiations remain unchanged. Two extra clocks, *fastClk* and *outClk*, are sup-

```
let m2ifftFF <- mkSyncFIFOFromCC(4, fastClk);

rule mapper2FF;
   connect(mapper.toIFFT, m2ifftFF.enq);
endrule

rule ff2ifft;
   connect(pop(m2ifftFF), ifft.fromMapper);
endrule
```

Figure 4. Transferring Between Two Domains

```
let ifft2ceFF <- mkSyncFIFOFromCC(4, outClk,
           clocked_by fastClk, reset_by fastReset);
```

Figure 5. Instantiation with Explicit Clocks

plied as arguments for the *IFFT* and the final sub-modules respectively; other sub-modules are instantiated to provide appropriate resets (from the default reset of the surrounding module), and these clocks and resets are used for the instantiation of the sub-modules for which they are intended. No change is necessary to the *controller2scrambler* rule: it will automatically be clocked by the clocks of its methods.

The tool will, however, object if no change is made to the rule connecting the *Mapper* and *IFFT* sub-modules, as this now attempts to connect methods from different clock domains without using an appropriate synchronizer. The required replacement for this rule is shown in Fig. 4: the required synchronizer is instantiated (the integer parameter specifies the depth of the FIFO), and two rules now connect the two ends to the methods corresponding to their clocks. Similar modification is required for the rule connecting the *IFFT* with the *CyclicExtender*; but in this case neither the source nor the destination domain is the default for the surrounding module, so the synchronizer's own "current clock" must be specified explicitly, as shown in Fig. 5.

These are the only changes required to the design. Moreover, if there are any inconsistencies in instantiating the sub-modules with the appropriate clocks and connecting them to appropriate synchronizers, the tool will detect and report the error at compile-time: this is of course a much more immediate and reliable process than the detection of rarely occurring run-time errors during simulation.

4.5. Comparison With Similar Changes in Verilog HDL

This change of the clocking for the 802.11 design (described for the Bluespec SystemVerilog model above) could also be accomplished for a model written in Verilog. Although a full Verilog model is not presented here, we briefly discuss the changes that would be involved, in order to highlight the benefits of the principles outlined in this paper.

To add two clocks into the module, an addition to the Verilog module's port list would be required, just as for BSV. Likewise the modules to be clocked by these new clocks must have their instantiation changed. After these preliminary (and incomplete) changes, the Verilog model can be compiled, and tests simulated. During compilation of this Verilog model, no objections are raised to the incorrect crossing of signals across clock domains. These errors must be identified by other means such as simulation, or with other "lint-like" tools (as discussed in Section 6.)

As the Verilog model development continues, data synchronizers must be added as with the Bluespec model. Assuming that there are already developed IP blocks for this purpose within the designer's environment, the typical Verilog-RTL methodology will have a module instantiation, followed by the connection of its ports with the other existing modules. Although the connection of the ports of the synchronizers to the ports of other modules is straightforward, the possibility of errors grows as this process is tedious; moreover Verilog with its weak type system will detect few incorrect port connections at compile time. With Bluespec SystemVerlog, however, the use of interfaces (with methods instead of ports) reduces the tedium of the connections; but more significantly, the compiler checks that connections are valid with respect to both data and clocking.

5. Other Features

5.1. First-class Properties of Clocks

We have stated that values of the *Clock* type enjoy most of the privileges of first-class objects; and we have seen them used as arguments to modules. This section illustrates more such capabilities.

Selection of Clocks Let us suppose that for our *mkTransmitter* design there is a Boolean global variable *oneClock*, to specify whether the design should use multiple clocks: if it is *True* at compilation time, all the sub-modules are to be clocked by the surrounding module's default clock, and the extra clocks are to be ignored. The relevant code for this is shown in Fig. 6. The *IFFT* and final sub-modules are clocked either by the current clock or by the special clocks supplied, depending on the setting of the Boolean. The same setting also determines whether to instantiate the synchronizers (only if necessary) and which rules to use to connect things up. Note that the clock selection is necessary—the tool will not allow synchronizers to be omitted between two different clocks supplied from outside.

This way of choosing clocks, by ordinary conditional expressions, is available only at compile time. It is possible to

```
Bool oneClock = False; // or True

(* synthesize *)
module mkTransmitter(Clock fastClk, Clock outClk,
                                Transmitter#(24,81) ifc);
   let currentClock <- exposeCurrentClock;
   let currentReset <- exposeCurrentReset;
   let ifftClk = oneClock ? currentClock : fastClk;
   let optClk  = oneClock ? currentClock : outClk;

   let ifftReset = currentReset;
   if (!oneClock) ifftReset
                    <- mkAsyncResetFromCC(2, fastClk);
   let optReset = currentReset;
   if (!oneClock) optReset
                    <- mkAsyncResetFromCC(2, optClk);

   let controller   <- mkController;
   ...
   let ifft         <- mkIFFT_Pipe
              (clocked_by ifftClk, reset_by ifftReset);
   let cyc_extender <- mkCyclicExtender
               (clocked_by optClk, reset_by optReset);

   ...

   if (oneClock)
      begin
         rule mapper2ifft;
            connect(mapper.toIFFT, ifft.fromMapper);
         endrule

         rule ifft2cyclicExtender;
            connect(ifft.toCyclicExtender,
                            cyc_extender.fromIFFT);
         endrule
      end
   else
      begin
         if (ifftClk==currentClock)
            error("unnecessary synchronizers");

         let m2ifftFF <- mkSyncFIFOFromCC(4, ifftClk);

         rule mapper2FF;
            connect(mapper.toIFFT, m2ifftFF.enq);
         endrule

         rule ff2ifft;
            connect(pop(m2ifftFF), ifft.fromMapper);
         endrule

         let ifft2ceFF <- ...

         ...
      end
```

Figure 6. Selection of Clocks, and Equality Testing

choose between clocks in the generated hardware (for example, to allow a particular sub-module's speed, and hence power consumption, to be varied dynamically), but this requires the use of a special primitive, *mkClockMux*, in case the downstream flow requires special provisions for multiplexing within clock trees.

Testing for Equality The reader may have noticed the test "`ifftClk==currentClock`" in Fig. 6—the designer has chosen to insert a test to check for confusion in the clock-handling part of the design. It illustrates that two clocks may be tested for equality (though this feature, like the previous one, is available only at compile time).

The *clockOf* Function Clocks, like other first-class objects, may also be returned as the results of functions. A particularly useful example of this is the polymorphic *clockOf* function, which may be applied to an expression of any type, and returns the clock which should be used to sample its values. If the expression is an unclocked constant, the special value *noClock* is returned. The value of this function is always well-defined—expressions for which this would not be so involve mixing values from several clock domains, and would be illegal.

An example of the use of this function may be seen in Fig. 9, below.

5.2. Gated Clocks

A simple way to save power is to switch off the clock for parts of the design when they are temporarily not in use. Differently gated versions of the same clock allow a simplified treatment, since when both are running they are exactly in phase. BSV provides special facilities to handle this case efficiently.

A BSV clock value contains two signals: an oscillator and a gating signal. If the gating signal is high, the clock is assumed to be ungated, and clearly the oscillator should be running. The tool remains agnostic about whether the reverse is true. Stopping the oscillator when the clock is gated saves the power being dissipated by charging and discharging the capacitance in the clock-tree itself, but it may require complicated interaction with the clock-handling tools downstream in the synthesis flow. The tool will, however, ensure that when the gating signal is low, all state transitions in that clock's domain are inhibited.

If a clock B is a gated version of clock A, we say that A is an *ancestor* of B. We therefore know that if a clock is running, then so are all of its ancestors. Clocks which are driven by the same oscillator, and differ merely in gating, are said to be in the *same family*. Both these relationships may be tested by functions available to the design: thus a

```
module mkWrapper(Bool transmit,
                 Clock fastClk, Clock outClk,
                 Transmitter#(24,81) ifc);
 Clock gc <- mkGatedClock(transmit);
                     // gated version of current clock
 Clock gf <- mkGatedClock(transmit,
                          clocked_by fastClk);
 Clock go <- mkGatedClock(transmit,
                          clocked_by  outClk);

 Transmitter#(24,81) trifc <-
     mkTransmitter(gf, go, clocked_by gc);
 return trifc;
endmodule
```

Figure 7. Use of Gated Clocks

library package can automatically arrange to exploit the extra simplicity of this situation without the user's having to be aware of it.

Each method of a primitive module is explicitly associated with a particular clock, often (but not necessarily) the default clock of the primitive module's instantiation. These methods are invoked by other methods (of other modules, written in BSV) or by rules: the tool rigorously insists that all methods invoked by any one method or rule are in the same family, thereby avoiding the risk of paths accidentally crossing between different clock domains. The invoking method or rule will be clocked by a clock in that same family, which is running if and only if all the invoked methods' clocks are running—if necessary, a new clock in the same family will be produced which satisfies this condition. The guard of any method which effects a state transition (that is, in general, any method which has an *ENABLE* signal) will include the gating condition of the method's clock—so such a method will not be *READY* unless its clock is running, and neither will any rule which invokes that method. (A method which merely returns a value, without executing a state transition, remains *READY* when its clock is switched off, returning the value set by the latest transition, provided that it was already *READY* when the switch-off occurred.)

All this mechanism obviates the need for any special clock-domain-crossing logic between same-family domains: everything is handled by the normal implicit-condition mechanism of BSV methods.

As an example of clock gating, Fig. 7 shows a wrapper for our transmitter design, which allows its clocking to be suspended when it is not in use, depending on the value of a Boolean dynamic argument *transmit*.

5.3. Clock-domain Crossing Between Families

As we have seen, when the clocks concerned are from different families domain crossing requires special logic. Practical design frequently requires a balance between conservative practice and the desire to meet design performance objectives. Often with synchronizers there is opportunity to save a clock cycle or two by removing handshake acknowledgements, or by removing one or even all synchronizing flops. While design methodologies and tools should encourage conservative practices, they should allow alternatives for optimization. For this reason, several alternative synchronizers (including a null or wire synchronizer) are provided. Even if these allow the designer to do without some or all synchronizing logic, the tool methodology enforces that a "synchronizer" be used, and that all clock domain checking remains in place during design.

The facilities provided by Bluespec fall into two groups, following two different approaches. The "hardware approach" provides modules with source and destination ports, which the designer instantiates and connects up explicitly; the low-level primitives are provided only in this form. The "linguistic approach" provides modules which transform an interface into one of the same type, but differently clocked: this allows a smoother treatment in the BSV notation. These two approaches are described and illustrated below.

The Hardware Approach Several synchronizers are provided to move data from a source clock domain to a destination domain. The details and use are dependant on the data, the expected handshaking, and the expected use. The synchronization primitives, from the simplest bit synchronizer to a full-handshaking synchronizing FIFO, are as follows.

(1) bit synchronizer: a bit change to the source causes a bit change in the destination;

(2) pulse synchronizer: a pulse on the source causes a pulse in the destination.

If the crossing is from a fast clock domain to a slower one, there is a danger in these two cases that information may be lost: a bit change might not be noticed if it persists only for a short time, and a sequence of fast pulses might result in fewer pulses on the destination side. The next synchronizers guard against this by providing a four-phase handshake protocol [9]:

(3) handshake pulse synchronizer: the same as (2), but a second source pulse is not accepted until a pulse has been delivered to and acknowledged from the destination.

(4) word synchronizer: a word delivered to the source eventually appears at the destination; a subsequent send cannot occur until the first has been delivered and the delivery acknowledged.

In these last two cases, even though an event is guaranteed to have happened on the destination side, there is no

```
let ifftIn = ifft.fromMapper;
let ifftOut = ifft.toCyclicExtender;

if (!oneClock)
   begin
      if (ifftClk==currentClock)
         error("unnecessary synchronizer");
      ifftIn  <- mkConverter(4, ifftIn);
      ifftOut <- mkConverter(4, ifftOut);
   end

rule mapper2ifft;
   connect(mapper.toIFFT, ifftIn);
endrule

rule ifft2cyclicExtender;
   connect(ifftOut, cyc_extender.fromIFFT);
endrule
```

Figure 8. The Linguistic Approach

guarantee that it has been noticed. The final case avoids this problem.

(5) FIFO synchronizer: data items enqueued on the source side will arrive at the destination side, and remain there until they are dequeued.

The Linguistic Approach This approach obviates the need for the designer to manage the synchronizer "plumbing" explicitly. For interfaces of suitable type, the use of this approach results in another interface of the *same* type, but differently clocked. Thus if *ifc* is such an interface, clocked by any clock, the instantiation

```
new_ifc <- mkConverter(n, ifc);
```

will produce an interface of the same type, but clocked by the clock of the current environment. There is no need to specify the clock of the original *ifc*, as the *mkConverter* module can determine that for itself. The parameter n specifies the depth of the conversion FIFO to be used between the two domains. Since the types of the original and the new interfaces are the same, and many of the details are implicit, this approach lends itself to generalization: the *mkConverter* name is overloaded, and can be used to implement clock-domain conversion on a whole class of interfaces.

Our previous examples of the 802.11a transmitter used the hardware approach to the definition of its synchronizers. Fig. 8 is comparable with Fig. 6, but uses the linguistic approach instead. Note how much more of the detail is implicit in this style.

This linguistic approach is provided as a means to keep the design abstraction at a higher level and not have designers worry about synchronizer details. The underlying hardware in the implementation of *mkConverter* is usually a synchronizing FIFO; an example is shown in Fig. 9; it is the one used for the *ifftIn* interface of Fig. 8. Note that the

```
module mkConverter#(Integer d)(function Action af(a x),
                               function Action ifc(a x));
  SyncFIFOIfc#(a) ff<-mkSyncFIFOFromCC(d, clockOf(af));
  rule dequeue;
    af(ff.first);
    ff.deq;
  endrule
  return (ff.enq);
endmodule
```

Figure 9. The Linguistic Approach—Implementation

types of the two interface arguments (the supplied interface *af* and the provided interface *ifc*) are the same. The module instantiates a synchronizing FIFO (as it were, using the hardware approach), defines a rule to connect the dequeue end to the *af* interface, and exports the enqueue method as the interface it provides.

5.4. Other Clock Generators

The system supporting multiple clocks must provide a means to generate clocks for both simulation and synthesis. A basic generator is provided, which creates a new clock family. Moreover, a clock divider is provided; although this generates a new clock family, special synchronization primitives are provided to take advantage of the known clock relationship. Both of these are further described below. As with many other primitives, users have the ability to include their own models in place of the provided ones.

Simulator Clocks For simulation purposes, the module

```
module mkAbsoluteClock#(Integer start, Integer period)
                       ( Clock );
```

is provided, and returns a clock. The two parameters specify the period and initial offset relative to the tick of the underlying simulator. This is useful for writing test-benches; it is not used in designs to be synthesized to hardware.

Clock Dividers It is often desired to create a new clock which runs at a sub-multiple of the frequency of the given clock. In this case it is easy to arrange that the slower clock's clocking edges coincide with clocking edges of the faster clock; problems of metastability are therefore avoided, and the task of a synchronizer is limited to ensuring that data values cross from the faster domain to the slower one only when the latter is ready to receive it. To handle this situation we provide the library module *mkClockDivider*, which has the following interface:

```
interface ClockDividerIfc ;
    interface Clock  fastClock ;  // The original clock
    interface Clock  slowClock ;  // The derived clock
    method    Bool  clockReady ;  //
endinterface
```

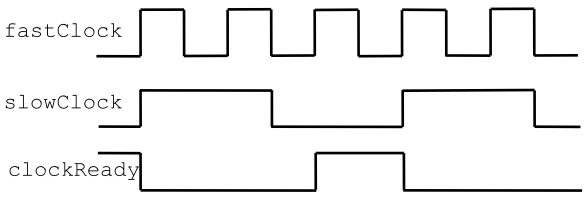

Figure 10. The Clock Divider Signals

```
let clockdiv12 <- mkClockDivider(12);
let clockdiv48 <- mkClockDivider(48);

let clk12th = clockdiv12.slowClock;
let clk48nd = clockdiv48.slowClock;
let reset13th <- ...

let m2ifftFF <- mkSyncFIFOToFast
                        (4, clockdiv12, reset12th);
let ifft2ceFF <- mkSyncFIFOToSlow
                        (4, clockdiv48, reset48nd);
```

Figure 11. The Use of Clock Division

As well as the two clocks, this interface provides an extra Boolean signal, *clockReady*. This signal, and its relationship to the two clocks, are shown in Fig. 10; it is clocked by the faster clock, and indicates that a clocking edge of the slow clock will occur at the next fast clock edge. This is used by the synchronizer as an extra implicit condition, allowing data to cross from the fast domain only on appropriate cycles.

A fragment of a transmitter version which generates its extra clocks internally, by clock division, is show in Fig. 11. In this version we assume that the transmitter is clocked by the fastest of the clocks required (that is, the one intended for the *IFFT* sub-module). The two dividers are instantiated for the slower clocks; the slower clocks themselves are extracted from the two interfaces, *clockdiv12* and *clockdiv48*; and the interfaces themselves are supplied to the two synchronizers. The synchronizers themselves contain much less logic than the more general ones. As usual, the tool reports an error if there is any inconsistency in the use of these synchronizers.

6 Conclusion and Relation to Other Work

The handling of multiple clock domains in a design can be a rich source of error. The original way of preventing such errors was visual inspection of the source code, coupled with testing; it was not very reliable, particularly since many of the runtime errors would only manifest themselves rarely, as they depended on particular phase-relationships of the participating clocks. Various companies have provided "lint-like" tools [1, 4, 13], which pro-actively analyze the source code for likely errors and other infelicities. A better solution is for the source code language to include a set of features which allows designers "to get it right first time", enforcing adherence to the relevant design rules at compile-time.

As far as we know, the only system apart from ours which attempts to do this is the recently-announced version of Esterel [8]. Within their different frameworks, the two systems provide a roughly similar range of facilities, though the emphasis is somewhat different. Our major concern is hardware synthesis; so, for example, we enforce the use of synchronizers between clock domains (though we allow designers to choose deliberately to make these simple, or even null). In Esterel, perhaps more concerned with verification in a simulation environment (where, because of the simulator's global tick, metastability issues do not arise), synchronizers are optional—the designer must remember to provide them when moving towards hardware synthesis. Both systems, however, raise the reliability of multi-clock designs substantially above the more old-fashioned approaches.

The features described here have been used in a substantial design (a UTMI block [11] for USB2.0). This handled USB transmission both at 480MHz and 12MHz; most of the logic was clocked at 120MHz (dealing with 4-bit nibbles in parallel). Thus many clocks were involved, and some careful design was required, some of it iterative, to perform the required domain crossing within the latency constraints of the specification (as tested by the Verisity Specman test suite [15]). It was found that the support provided by the language avoided all accidental mistakes, allowing iterative improvements to be implemented and tested quickly. It is estimated that the design was completed (and passed the test suite) in at most half the time it would have taken had it been written in standard Verilog.

7 Acknowledgments

The authors thank their colleagues at Bluespec Inc., and members of the Computation Structures Group at MIT's Computer Science and Artificial Intelligence Laboratory, for advice and assistance with this project.

References

[1] Atrenta, Inc. *SpyGlass Predictive Analyzer User Guide*, 2005. www.atrenta.com.

[2] P. E. Black. Gray code. In P. E. Black, editor, *Dictionary of Algorithms and Data Structures [online]*. U.S. National In-

stitute of Standards and Technology, 2005. Available from: http://www.nist.gov/dads/HTML/graycode.html.
[3] Bluespec, Inc. *Bluespec Language Reference Guide*, 2006. Please consult http://www.bluespec.com.
[4] Cadence Design Systems, Inc. *Incisive HDL Analysis (HAL)*, 2004.
[5] M. Crews and Y. Yuenyongsgool. Practical design for transferring signals between clock domains. *EDN*, pages 65–71, February 20 2003. http://www.edn.com/article/CA276202.html.
[6] C. E. Cummings. Synthesis and scripting techniques for designing multi-asynchronous clock designs. In *Synopsys Users Group Conference*, San Jose, March 2001. www.sunburst-design.com/papers.
[7] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a Transmitter: A Case Study in Microarchitectural Exploration. In *these proceedings*, 2006.
[8] Esterel Technologies, Inc. *Esterel Studio Version 5.3*, 2006.
[9] R. Ginosar. Fourteen ways to fool your synchronizer. In *Ninth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'03)*, page 89, 2003.
[10] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.
[11] Intel Corporation. *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*, 2001.
[12] M. Stein. Crossing the abyss: asynchronous signals in a synchronous world. *EDN*, pages 59–69, July 24 2003. http://www.edn.com/article/CA310388.html.
[13] Synopsys, Inc. *Leda Programmable RTL Checker*, 2006.
[14] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification , and Verification Language, November 2005. IEEE Std 1800-2005, http://standards.ieee.org.
[15] Verisity Design, Inc. *USB eVC*, 2004.

A. Bluespec SystemVerilog

Previous attempts at behavioral synthesis have tried to optimize along three axes simultaneously: choice of micro-architecture, allocation of resources, and scheduling of concurrent operations. Doing all this together, however, is computationally "hard". Besides, designers are good at evaluating micro-architectures, and like to be in control of resource allocation; handling concurrency, however, often becomes excessively complex. The Bluespec tool takes over this task, while leaving the designer in control of the other two. The result is a flexible tool, with which it is easy to do architectural experiments, while producing RTL of comparable quality to hand-crafted Verilog.

The HDL for the Bluespec tool is Bluespec SystemVerilog (BSV). This is a variant of SystemVerilog [14] in which behavior is specified, not by the usual "always-blocks", but by "design assertions" also known as "rules". A rule consists of a condition and an action part. Only if the condition is satisfied may the state transition specified by the action part be executed. Each action executes in a single clock cycle. The tool generates scheduling logic which executes as many rules as possible in each cycle, but with the restriction that the overall effect must be the same as if they had each executed one at a time in some order. This restriction avoids many race conditions, replacing them with a compile-time warning that two rules could not be simultaneously scheduled because of a resource conflict, which is much easier to deal with.

As in standard SystemVerilog, a BSV design is partitioned into modules. The designer can control which modules are synthesized by the Bluespec tool into separate RTL modules (output in low-level Verilog2001) and which are inlined. The action part of a rule effects a state transition by invoking a *method* of some other module's interface. Even individual registers are actually instantiations of primitive modules (written in standard Verilog)—references to them are "desugared" into invocations of their *_read* and *_write* methods.

The interfaces of these other modules might be available in the module's environment; or the modules might be instantiated within the module being defined; or they might be supplied as arguments to that module. These appear in the list corresponding to the port list of a standard Verilog module; the input/output distinction is inappropriate for these arguments (as each interface, and indeed each of its methods, contain both input and output signals), so instead we distinguish between the interfaces "used" by a module, and the interface (by convention the last one in the list) it "provides" by defining each of its methods.

For modules which are separately synthesized, the methods of a module's interface become collections of ports in the RTL version. As well as the data ports (input or output), each method in general has an output *READY* signal, asserting that the method may validly be invoked; methods which effect state transitions also have an input *ENABLE* signal, asserting that the transition is to be executed. The tool enforces the protocol that the *ENABLE* signal may not be asserted unless the corresponding *READY* signal is also asserted. At the language level, the method's validity conditions are implicitly added to the conditions of any rule (or other method) which invokes it; similarly, its action becomes part of the rule's (or other method's) one-cycle action. Thus we achieve modularity, and the "separation of concerns", by defining methods within the module which provides them, while preserving the atomicity of the rule which invokes them.

A.1. Example—Factorial

Figure 12 shows a simple complete BSV design, containing a module *mkFact* for computing the factorial function, and a testbench module *mkTestFact* for exercising it.

The interface provided by *mkFact* is of type *NumFn*; it consists of a method *start* to initiate a calculation, and a

```
package Factorial;

typedef UInt#(32) Nat;

interface NumFn;
   method Action start(Nat x);
   method Nat result;
endinterface

(* synthesize *)
module mkFact(NumFn ifc);
   Reg#(Nat) n <- mkReg(0);
   Reg#(Nat) a <-  mkRegU;

   rule calc (n!=0);
      a <= a * n;
      n <= n - 1;
   endrule

   method Action start(x) if (n==0);
      a <= 1;
      n <= x;
   endmethod

   method result if (n==0);
      return a;
   endmethod
endmodule

(* synthesize *)
module mkTestFact(Empty);
   NumFn fact <- mkFact;
   Reg#(UInt#(2)) state <- mkReg(0);

   rule start_test (state==0);
      state <= 1;
      fact.start(7);
   endrule

   rule show_result (state==1);
      state <= 2;
      $display("%d", fact.result);
   endrule

   rule end_test (state==2);
      $finish(0);
   endrule
endmodule

endpackage
```

Figure 12. A simple BSV package

```
module mkFact(CLK,
              RST_N,
              start_x,
              EN_start,
              RDY_start,
              result,
              RDY_result);
  input  CLK;
  input  RST_N;

  // action method start
  input  [31 : 0] start_x;
  input  EN_start;
  output RDY_start;

  // value method result
  output [31 : 0] result;
  output RDY_result;
  ...
```

Figure 13. Part of the RTL for the simple example

method *result* to retrieve the result. The actual computation is performed by the rule *calc*, which can run only when $n \neq 0$. The two methods, on the other hand, are ready only if $n = 0$: the result cannot be read while a computation is still in progress, nor can a new computation be started.

The first few lines of the RTL synthesized from *mkFact* are shown in Figure 13. As well as the methods' signals, already described, clock and reset ports will be noticed.

In the testbench, *mkTestFact*, the *mkFact* module is instantiated, giving an interface called *fact*. Its *start* method is invoked by the *start_test* rule of *mkTestFact*. Thus the complete condition of *start_test* is `state==0 & n==0`: the first test comes from the rule's condition and the second from the method's. Similarly, the action part is

```
state <= 1;
a <= 1;
n <= x;
```

amalgamating the actions of the rule and the method; all the assignments are executed simultaneously.

Notice that in this very simple design, all the rules are mutually exclusive—at most one is enabled at any one time. There is therefore no possibility of conflict, and the scheduling is trivial: each rule may fire whenever it is enabled. In general, however, many non-conflicting rules may fire during any one cycle.