

# Design Challenge #1: a Basic 2x2 Interconnect

February 2007  
www.bluespec.com

© Copyright Bluespec, Inc., 2007 All Rights Reserved.

## Overview

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 in the algorithm space have been successful at efficiently synthesizing higher-level math/DSP designs, Bluespec is the only solution that succeeds 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.

This document outlines Bluespec’s design solution to Design Challenge #1, the basic 2x2 interconnect, as summarized in DeepChip’s ESNUG #459, Item #5. The motivation behind the design challenge was to shift from marketing claims about high-level design into actual examples using real designs. Hopefully, this will enable us to debate what it means to implement control logic intensive designs at a high level versus RTL and explore the quality of the resulting implementations.

We invite you to review this 2x2 switch design (as well as the separately outlined DMA controller design). These designs are intended to give you a taste of what it means to design control logic and complex datapaths at a level above RTL. And, please keep in mind that this design was written to conform to the problem specification – Bluespec provides complete control over the micro-architecture and can be used to design any type of switch interconnect.

Highlights of the 2x2 interconnect design include:

- The **refinement methodology** with which it was designed. The design began as a functionally correct, bit-true version that was pulled together in about an hour – and was subsequently refined to meet the specification in three easy steps.
- The **high-level transactions** that are used to describe the core functionality of the switch – and the interfaces. These make the design faster to design correctly, easier to read, and much more extensible.
- The **quality of results** of the final design. The design meets the tight latency requirement – which is a critical attribute in a switch interconnect as it directly impacts system performance. In 65nm TSMC (using Artisan libraries for CLN65GP), using Design Compiler the area is 15.7Kum<sup>2</sup> and the design frequency is 1.11 GHz under worst case conditions and extended operating range (SS, 0.90V, 125 Degrees C). In 0.18u TSMC (using Artisan libraries for CL018G), using Design Compiler the area is 64.87 Kum<sup>2</sup> and the design frequency is 380 MHz under worst case conditions and extended operating range (SS, 1.62V, 125 Degrees C).

While the solution is implemented five ways, four versions using Bluespec SystemVerilog (BSV) and one using SystemC, this document will primarily describe the BSV designs – an appendix will provide a short introduction to the SystemC version.

The BSV design starts with a functionally correct, executable, synthesizable design (Version 1) that can be put together in about an hour, using existing BSV library facilities for buffering, interfaces and connections. We then refine it in three steps. Version 2 implements the desired round-robin arbitration inside the interconnect. Version 3 fixes up the buffering at the ingresses and egresses to meet the desired 1-cycle latency across the switch. The final design, Version 4 fixes up the “sockets” of the interconnect to meet the desired socket signaling protocol exactly. The steps genuinely constitute a *refinement* because each change is a local change, keeping the rest of the code intact. All four versions are fully synthesizable.

This document is distributed along with full BSV source codes, generated Verilog, testbenches and test logs for all four versions. A simple “diff” between corresponding source files in successive versions is adequate to observe the refinement changes.

All this is made possible because of BSV's high level of abstraction, powerful types and strong type-checking, clean semantics based on transactions (Rules) and transactional (Rule-based) Interfaces, and automatic regeneration of correct control logic as we substitute one component with another.

## Problem specification

For your reference, the problem specification from DeepChip is repeated in this section.

### Design Challenge #1: a Basic 2x2 Interconnect

Suppose we design a crossbar switch for an SoC connecting initiators to targets like processors and DMA engines. Example targets are memories, I/O blocks, and the DMA configuration port. The specs are:

- For uniformity, all busses are 32 bits wide.
- Two initiators and two targets.
- Requests (initiators to targets) are completely decoupled from responses (targets to initiators).
- Both requests and responses can be pipelined.
- The switch should preserve request order from a particular initiator to a particular target, and the response order from a particular target to a particular initiator.
- For simultaneous requests from the 2 initiators towards the same target, and for simultaneous responses from the 2 targets to the same initiator, there should be round-robin arbitration.
- In the best case (i.e., if allowed by arbitration and absence of back-pressure), requests and responses should make it across the switch in one clock cycle, i.e., they should be buffered for just one cycle in the switch.

Each connection between an initiator and the switch and between a target and the switch (also called a socket) has the following structure and protocol, similar to the OCP-IP protocol:

- Each initiator has a master interface, connecting to a slave interface on the switch.
- Each target has a slave interface, connecting to a master interface on the switch.

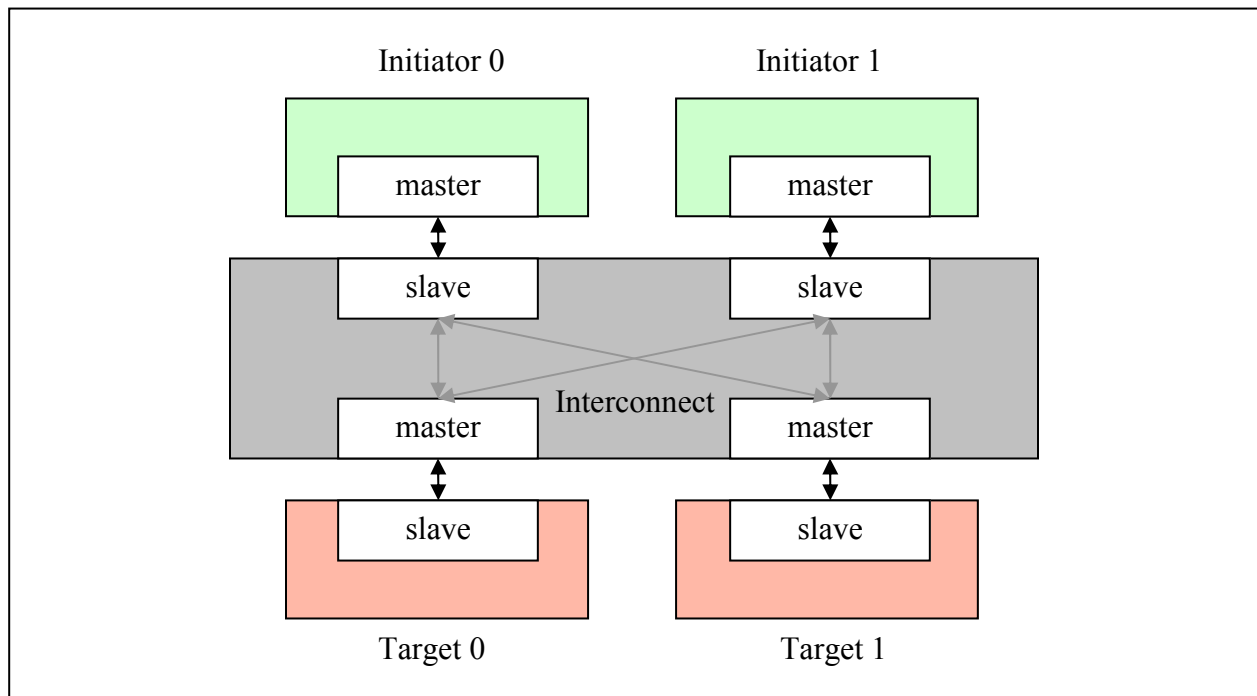
- A master must send a request on every clock cycle (it sends a NOP request if it does not have a real request to send). It can advance to the next request whenever it sees an accept signal from the slave. An accept refers to the request on the current cycle, and so the master can send the next request on the very next cycle, i.e., requests can be pipelined at full bandwidth of one request per clock.

- Symmetrically, a slave must send a response to a master on every clock cycle (it sends a NOP response if it does not have a real response to send). It can advance to the next response whenever it sees an accept signal from the master.

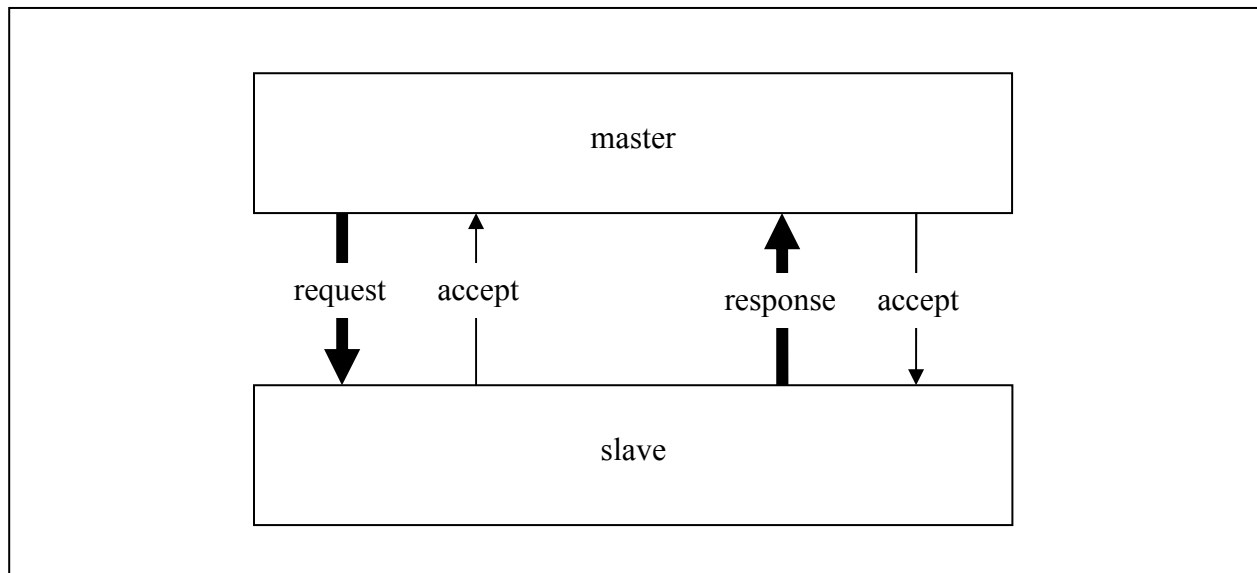
#### Testbench Requirements:

- Must ensure that simultaneously sent traffic from 2 different initiators to 2 different targets occurs at full bandwidth.
- Must ensure that simultaneously sent traffic from 2 different initiators to the same target occurs in properly arbitrated form (round-robin).

In the following figure 1, there is a block diagram of the architecture of the design:



In the following figure 2, there is a diagram of the protocol between master and slave interfaces. Master and slave interfaces occur on both the initiator and target sides of the interconnect. On all four sockets in the design, requests flow from a master interface to a slave interface – responses flow from a slave to a master:



## Bluespec 101

Bluespec enables design at a much higher level than RTL, while leaving the specification of the architecture to the designer, giving him/her 100% control over the quality of the implementation including latency, area, power and timing. With Bluespec, designs and models are expressed using transactions to define both internal module behavior within modules and interfaces between modules.

This section is not intended to make you an expert, but it is intended to give you enough background to understand the design of the 2x2 switch interconnect.

There are four key ideas to keep in mind while looking through the code:

- **Explicit state** – all state, e.g. registers, RAMs, FIFOs, ROMs, any sub-module,..., is explicitly specified by the designer. No state is added or subtracted by the Bluespec compiler. Every module will have its state explicitly coded – though you get to leverage high-level, abstract types for more succinct and expressive specification of that state. You'll see this state reflected, as is, in the Verilog generated by the Bluespec compiler.
- **Transactions for behavior within modules** – instead of ALWAYS blocks, you'll see behavior coded in structures called RULEs, which describe behavior using isolated transactions. A rule is a description of what to do for a particular situation. A rule describes what state to update when certain designer-specified conditions are true. For example, if your design required some things to happen when an interrupt happened, you would code a rule to perform these things whenever the conditions for an interrupt were true.

The key advantage of this approach is that designers can code each "RULE" in isolation without regard to what else is happening in the system – this allows designers to design incrementally (which really simplifies the complexity of hardware design, where you typically have to pay extremely close attention to resource contention, such as multiplexing, communication, synchronization). For example, when you code the interrupt rule, you can do so without worrying about its interaction with the rest of the design, such as the logic responsible for "normal" operation – you let the compiler worry about properly flagging and handling unanticipated interactions. The tool is responsible for scheduling as many rules to run in parallel as possible. During this process, it will identify any shared resource situations, flag them for you, and automatically insert the proper multiplexing for you as well

as the logic to schedule access to these common resources.

Here's the syntax for a rule (using Bluespec SystemVerilog (BSV)):

```
rule <ruleName> (<boolean cond>);  
  <state update(s)>  
endrule
```

On first blush, a rule sort of operates like an **if...then...** When the <boolean cond> is TRUE, then the <state update(s)> occur(s). If there are multiple updates, then all have to happen together, or none can happen at all.

Well, that's the simplified, introductory view. The power is in the additional sophistication. In addition to the <Boolean cond>, there are two additional considerations to whether a rule "executes" its state updates:

1. The first is that, although most rules will execute in parallel in hardware, Bluespec's compiler ensures that each rule executes in the hardware *as though* it were the ONLY thing operating at that moment. Another way to describe these rules is ATOMIC transactions. This is what allows designers to think about and design each rule by itself and get it right – the compiler ensures that this atomic behavior is maintained in the hardware.

When does this really matter? It comes into play particularly when there are rules that might want to update common (or shared) state. In RTL hardware design, designers need to keep track of all the potential updates to shared state AND properly schedule access to this shared state by many different operations in the hardware. With Bluespec, these shared resource conditions are automatically recognized and scheduled (albeit with 100% designer visibility and control). And, the compiler will even recognize when rules have mutually exclusive access to the same state and ensure that they will always have access to that state when needed.

The end result: you can never have unintended race conditions accessing shared resources. This eliminates a major source of obscure, subtle errors in hardware design.

As such, a rule's Boolean condition may be TRUE, but its operation may conflict with another rule which takes precedence which may prevent its execution in a given cycle (due to an attempt to access shared state). The compiler generates the optimal parallel execution of rules in hardware while properly scheduling access to shared resources.

2. The second consideration has to do with how interfaces work. With Bluespec, all state elements such as registers and memories have interfaces just like traditional design modules. But with Bluespec, these interfaces are transactional – they not only describe the wire interfaces, but they also understand how to pass data and the proper conditions for doing so.

These proper conditions also get considered in determining when a rule can execute. The easiest way to understand this is to review the following rule example:

```
rule pass_through (state == FULL_THROTTLE);  
  y <- fifoIn.deq();  
  z = f(y);  
  fifoOut.enq(z);  
endrule
```

This rule has an explicit condition that says it should execute when in the "FULL\_THROTTLE" state. But, when it executes, it is supposed to dequeue a value from fifoIn and, at the same time, enqueue the result of f(y) into fifoOut. What if there's nothing to dequeue from fifoIn? What if

fifoOut, the place you want to put the data, is full? Normally, designers have to design the control logic to check all of these things. With Bluespec, these conditions (for all interfaces of modules and state, such as registers, FIFOs, ...) become implicit conditions of the rules that use the interfaces. If ANY of the interfaces is not ready, then the rule cannot execute.

Even in this basic rule example, a lot of complex behavior is simplified. The “pass\_through” rule will move data from one FIFO to another after manipulating it with f() only when:

- In the proper state: state == FULL\_THROTTLE
- All the interfaces allow it (in this case: that there is data to be moved from fifoIn and there is also a place to put it into fifoOut)
- There are no other rules that want to dequeue from fifoIn or queue into fifoOut in this cycle. If there were, only one would be allowed.

When the rule executes, all three actions will happen simultaneously: data would be dequeued, passed through the combinational logic in the function f(), and the result would be queued into another FIFO. A lot of explicit control logic in RTL is handled succinctly and implicitly with rule and interface transactions with Bluespec.

- **Transactions for interfaces between modules** – as we just introduced in the last section, instead of port lists, interfaces are transactions defined using rule-based interface methods, as shown below, using some extensions to the notation of SystemVerilog or SystemC (the former is shown below):

```
interface FIFOBuf#(type x_t);
    method Action enq (x_t x);           // Note: an “Action” means that it changes state
    method ActionValue#(x_t) deq ();    // Note: an “ActionValue” means that it returns a value
                                         // and changes state
    method Action clear ();
endinterface
```

[Note: the (type x\_t) construct is just SystemVerilog's notation for type parameterization (also known as polymorphism, or genericity), i.e., x\_t is a type variable representing the type of items stored in the FIFO.]

The enq interface method encapsulates all the ports for enqueueing: the input data bus (whose width depends on the particular data type to which the generic type x\_t is instantiated), the output READY signal (which tells the outside world when the FIFO is able to accept an enqueue), and the input ENABLE signal (which the outside world uses to tell the FIFO that it is enqueueing). Similarly, the deq method encapsulates all the ports required for dequeueing: the output data bus, the output READY signal (which tells the outside world when valid data is ready to be dequeued) and the input ENABLE signal. Based on the interface method's guard conditions in the design (e.g. “not full” for an enqueue), the compiler generates the ready and enable signals automatically in the output Verilog RTL.

What does it mean to encapsulate the ports? Let's look at the code for an enqueue interface method in a FIFO:

```
method Action enq(x) if (notFull);
    rw_enq.wset(x); // Pass the input through (when 'notFull')
endmethod
```

The *notFull* is a Boolean signal that is part of the internal FIFO implementation which simply indicates whether the FIFO has space or not. While not exposed to an external user of the FIFO, this value becomes an implicit condition on whether the method can be called and it forms the value behind the READY signal associated with this method in the generated RTL. This *notFull* condition becomes an additional condition that is ANDed into the composite conditions on any rules that use this method.

In general, method arguments become module input data bus ports. Method results (such as that returned by `deq`), become output data bus ports. A method can have multiple output data bus ports because return types can be structs (records) with multiple fields, and vectors. All methods have an output `READY` signal (which is the interface method's condition). All Action and ActionValue methods (like those shown) have input `ENABLE` signals. Action and ActionValue methods can cause a state change inside the module. A third kind of method, which we call Value methods (not shown in this example), are purely combinational—their results are combinational functions of their arguments and internal module state. The compiler optimizes away `READY` and `ENABLE` signals of a method if it proves that they are always asserted.

A client module that uses the FIFO contains Rules that invoke the `enq` and `deq` methods, as in the example below:

```
module mkClient (...);
  ... instantiate fifo ..
.
  rule upstream (... cond1 ...);
    ... other actions ...
    fifo.enq (expr1);
  endrule

  rule downstream (... cond2 ...);
    x <- fifo.deq ();
    ... other actions ...
  endrule
endmodule
```

As discussed before, each rule has an explicit condition, depicted above as the expressions *cond1* and *cond2*. These are pure combinational Boolean expressions. Each rule also contains one or more actions that can be executed atomically only if the rule condition is true. For example, the *upstream* rule contains an action that enqueues the value of expression *expr1* into the FIFO, and the *downstream* rule contains an action that dequeues an item *x* from the FIFO.

The conditions of all methods invoked by a rule are incorporated into the overall condition of the rule. For example, the `ENQ_READY` signal is “AND”ed with *cond1* to determine the overall condition of the upstream rule. The `DEQ_READY` signal is “AND”ed with *cond2* to determine the overall condition of the downstream rule.

So to reiterate and summarize, a rule can only execute if all its conditions permit. When it executes (we call it: fires), all its actions, including all the actions in all the methods that it invokes, are executed simultaneously as one composite atomic action. Thus, the upstream rule can only fire if `ENQ_READY` is true, and then the enqueueing action becomes part of the overall atomic action of the rule. When the rule fires, the enqueued data is driven and `ENQ_ENABLE` is asserted.

The condition of a method or a rule is necessary, but not sufficient, for a rule to fire. In particular, since rules can share resources (such as the FIFO above), simultaneous firing might not be possible while maintaining atomicity, i.e., if simultaneous firing would lead to inconsistent states. The compiler emits scheduling logic to ensure that simultaneous firing is only possible if it maintains atomicity.

When compiling a FIFO implementation, the compiler performs a systematic analysis that infers whether the `enq` and `deq` methods can be operated simultaneously safely, and under what conditions. Note, different FIFO designs may or may not permit such simultaneous operation.

This interface information is recorded by the compiler with the FIFO implementation. Then, when compiling `mkClient`, the compiler uses this information to introduce suitable control logic in `mkClient` to guarantee that the upstream and the downstream rules can fire simultaneously only when

conditions permit them to do so safely.

- **High-level, abstract types, parameters and polymorphism** – along with the power to express the behavior of the design as a set of transactions and transactional interfaces, you'll also see more expressive design using high-level abstract types, such as structures and enumerated types. Also, though less emphasized in this particular design, there is the power to parameterize designs along almost any dimension. Examples include the ability to parameterize with: interfaces (which type of interface to use), functions (e.g. what kind of sort algorithm), types (polymorphic design), sizes, and even modules. At compile time, the tool will generate the hardware specified by the chosen parameters – and elaborate the hardware using the parameters as well as other procedural controls: conditionals such as *if*, *for* loops, and even recursion.

## Problem Solution: Introduction

The following sections show how we approach this design through systematic refinement in Bluespec SystemVerilog (BSV). We develop the design in four stages of refinement (all synthesizable):

- Version 1 is a quick first cut, including a testbench, in which we can run bit-true traffic through the system. We make extensive use of BSV library facilities for buffering, interconnects and connections, allowing us to rapidly implement a working system (in about an hour). It will not contain round-robin arbitration; it will not meet the 1-cycle latency requirement; it will not follow the socket signaling protocol, but
  - it is bit-true, i.e., transports all request and response bits,
  - and is functionally correct, i.e., correctly transports requests and responses to the correct targets and initiators, respectively and so can be used in simulations.
- Version 2 adds support for the round-robin scheduling requirement.
- Version 3 adds support for the 1-cycle latency requirement.
- Version 4 meets all the requirements, adding support for the socket signaling protocol requirement.

The steps genuinely constitute a *refinement* – each version reuses most of the source code from the previous version, i.e., each change is very local.

This document is not intended as a detailed tutorial on BSV. It is only intended to give a feel for doing high-level design in a Bluespec environment – while you may not understand all the details, hopefully you will be able to follow the commentary and get the gist of the code.

## Problem Solution: Version 1

The file **Socket\_IFC.bsv**, (the sources for this version are located in the directory: InitialVersion.v1/Sources) defines the bit-true representations of requests and responses on each socket of the switch interface.

A request (struct *Socket\_Req*) contains an opcode field (RD, WR or NOP), an info field, an address field and a data field. If the opcode is NOP, none of the other fields are relevant. If RD, the address is relevant. If WR, both address and data are relevant. The info field is an “application-specified” field; here we just use the LSB to tell the target which initiator sent this request (0 or 1), so that it knows where to send the response.

The switch uses the LSB of the address bit to route requests either to target 0 or target 1, i.e., even addresses go to target 0, odd requests to target 1.

A response has similar fields. A response is only expected for RD requests. The info field is used by a target to inform the initiator which target sent this response (0 or 1). The LSB of the address field is used to route responses back to initiator 0 or 1.



After this, we use typedefs to define a master interface to be just the BSV library Client interface, from which one can use the *get* method to obtain requests, and the *put* method to give it responses. Similarly, we define a slave interface to be just the BSV library Server interface, from which one can use the *put* method to give it requests, and the *get* method to obtain responses.

Finally, we define two functions *fifos\_to\_master\_ifc* and *fifos\_to\_slave\_ifc* that convert pairs of FIFO interfaces to master and slave interfaces, respectively.

The file **Switch.bsv** defines the crossbar switch. First, we define *Switch\_IFC*, the interface of the switch. Then, we define some address-decode functions for requests and responses. Finally, we define *mkSwitch*, the switch module itself. It has 8 FIFOs (from the BSV library) for buffering requests and responses, a pair for each socket. Each pair holds incoming and outgoing items. These are followed by 8 rules, one for each combination of request/response, initiator0/initiator1, and target0/target1. Finally, we use the functions *fifos\_to\_master\_ifc* and *fifos\_to\_slave\_ifc* to create the interfaces of the switch.

The above two files, representing the complete first cut at the design, have less than 300 lines of BSV source code. By using FIFOs, GetPut, ClientServer etc. from the BSV library, the whole design can be put together rapidly.

The file **Tb.bsv** is a small testbench. It starts with the definition of the top-level module *mkTb*, which corresponds directly to Figure 1, i.e., it instantiates two initiators, two targets, the switch, and makes all the connections. This is more than just a “wiring” of the top-level modules! The interface methods have full BSV Rule semantics, with all the benefits therein (protocol correctness). Each method in each interface incorporates a complete signalling protocol, complete with flow control, and the *mkConnection* constructs implement logic that ensure that this signalling protocol is followed precisely. In BSV, using such connection constructs, one *never* encounters the kind of timing errors that one often encounters in RTL interfaces, such as asserting a READY signal when it was not supposed to, or reading/writing a data bus on the wrong cycle, etc.

In the file, *mkTb* is followed by *mkInitiatorModel*, a definition of a model of an initiator. The *id* parameter is assumed to be unique for each instance of an initiator (we use 0 and 1). We instantiate a random number generator using *mkFeedLFSR* from the BSV library, giving it a different seed for each instantiation (i.e., for each *id*) so that the different instances generate different pseudo-random sequences. We instantiate two FIFOs *reqs* and *resps* to buffer outgoing requests and incoming responses. We also instantiate two FIFOs *expected\_resps\_0* and *expected\_resps\_1* to hold expected responses from targets 0 and 1, respectively. For these last two FIFOs, we have sized them sufficiently large (10 deep) to account for pipeline latency, i.e., we will be able to send out multiple requests before we receive the first response.

In rule *gen\_reqs*, the expression  $((randx \& 7) > 5) ? WR : RD$  randomly generates WR (25% probability) and RD (75% probability) requests. We also generate random addresses and data. For RD requests, it creates expected responses and holds them in the FIFOs *expected\_resps\_0* and *expected\_resps\_1*, depending on whether the request is going to target 0 or 1. We cannot hold all expected responses in a single FIFO because the two targets may return responses out of order. The rules *accept\_resps\_from\_0* and *accept\_resps\_from\_1* accept incoming requests from targets 0 and 1, respectively, and check them against expected responses.

The module *mkTargetModel* has one rule, *respond*, which simply consumes all requests and generates responses for RD requests.

This entire program can be simulated using any of the standard ways to simulate BSV programs, for example, using Bluesim, the native source code Bluespec simulator, or Verilog simulation. It can be compiled using the Bluespec compiler *bsc* to Verilog (including the testbench!), and simulated using any Verilog simulator. The Verilog is further synthesizable to a netlist using a standard RTL-to-netlist synthesis tool.

When compiling **Switch.bsv** using *bsc*, the compiler will issue four warnings. For example, it warns that, since rules *initiator\_0\_to\_target\_0* and *initiator\_1\_to\_target\_0* may both want to enqueue a request simultaneously for target 0, it must arbitrate, and therefore it picks a static priority of one over the other. The other three warnings are similar, for arbitration of requests towards target 1, and responses towards initiators 0 and 1, respectively. We will ignore these warnings, since we will anyway fix this with round-robin arbitration in Version 2.

Because of the FIFOs for incoming and outgoing items in *mkSwitch*, the minimum latency across the switch will be 2 cycles for this version, i.e., each item spends at least one cycle in an incoming FIFO and one cycle in an outgoing FIFO.

## Problem Solution: Version 2

In this version, we add round-robin arbitration in file **Switch.bsv**, (the sources for this version are located in the directory: v2/Sources). We recommend doing a “diff” between the **InitialVersion.v1/Sources/Switch.bsv** and **v2/Sources/Switch.bsv** to see the change. In the *mkSwitch* module, we have added four Boolean registers to hold the round-robin state for the four outlets of the switch (two for requests towards targets, and two for responses towards initiators). We have added four rules, one for each egress, such as rule *both\_initiators\_to\_target\_0*, which fires when both initiators have requests destined for target 0. These rules implement the round-robin policy. Note that the condition of the rule *both\_initiators\_to\_target\_0* includes the condition of the rule *initiator\_0\_to\_target\_0*, so that whenever the former is enabled, so will the latter. The *descending\_urgency* attribute gives priority to the former rule in this situation. Thus, if both initiator FIFOs have requests for target 0, then the “both” rule will fire; if only one of the FIFOs has a request for target 0, then the original rules will fire. The original rules did not have to be touched. Note: even though we have replicated some text from the original rules into the new rules, the BSV compiler will automatically share all common logic.

The files **Socket\_IFC.bsv** and **Tb.bsv** remain completely unchanged.

## Problem Solution: Version 3

In this version, (the sources for this version are located in the directory: v3/sources), we fix the latency across the switch to achieve the desired 1-cycle latency.

Again, the files **Socket\_IFC.bsv** and **Tb.bsv** remain completely unchanged.

We introduce a new package in the file **EdgeFIFOs.bsv** that is properly considered a library element because it is in no way specific to the current SoC switch design. It contains highly generic (parameterized) facilities that have broad applicability in a number of designs. However, we include the package here for completeness and for discussion. The package defines two FIFOs that are useful to achieve certain performance requirements. Please see the detailed comments at the head of the file.

The module *mkPipelineFIFO* is a 1-element FIFO into which it is possible to simultaneously enqueue and dequeue an item when it already contains an item (i.e., this is the same as an interlocked pipeline register).

The module *mkBypassFIFO* is a 1-element FIFO into which it is possible to simultaneously enqueue and dequeue an item when it is empty—the newly enqueued item is immediately “bypassed” through to the dequeue operation. Thus, the Bypass FIFO can have zero latency, i.e., if an enqueued item can be buffered in the FIFO for one or more cycles before it is dequeued, but in the best case the item can spend zero cycles in the FIFO if it is bypassed through.

If we look at the new version of file **Switch.bsv**, note that the sole change is to replace the previous

*mkFIFO* module instantiations by *mkPipelineFIFO* and *mkBypassFIFO* instantiations. This is possible because they have exactly the same interface as ordinary FIFOs. By making sure that in each path through the switch we have one *PipelineFIFO* and one *BypassFIFO*, we automatically achieve the desired 1-cycle latency, i.e., in the best case, an item spends 1 cycle in the *PipelineFIFO* and zero cycles in the *BypassFIFO*. Note: the Switch still has exactly the same interface as before, and is still fully round-robin-arbitrated, and is still fully flow-controlled as before. Further, none of the Rules had to be changed.

There is much subtlety under the covers, managed by the Bluespec compiler. In *mkPipelineFIFO*, there is a control dependency (and a combinational path in the ensuing circuits) from the *enq* operation to the *deq* operation, because *enq* is allowed if *deq* is simultaneously attempted when the FIFO already contains an item. Similarly, in *mkBypassFIFO*, there is a control dependency (and a combinational path in the ensuing circuits) from the *deq* operation to the *enq* operation, because *deq* is allowed if *enq* is simultaneously attempted when the FIFO is empty. Because these properties are formally captured in interface method scheduling semantics, the Bluespec compiler checks that everything remains consistent (e.g, no combinational paths), and it constructs the correct control logic to take these properties into account. However, in the source code, it is as simple as substituting *mkFIFO* by one of these FIFOs.

## Problem Solution: Version 4

In this final version, we fix up the switch ports so that they follow the socket protocol exactly.

We often refer to this step as “impedance matching” for the following reason. Had we been free to choose the switch socket protocol, we would have stopped at Version 3—it is a perfectly fine socket protocol and is fully synthesizable, and efficient. Indeed, when we are working entirely within BSV, we take advantage of these high-level, robust BSV protocols and don't worry any more about nitty-gritty signaling details. Thus, typically this “impedance-matching” activity is only needed at the edge of a BSV subsystem where it has to interact with some existing IP for which a protocol has already been specified.

In implementing normal BSV interfaces, every method has a *READY* output signal indicating when the method can be used. For value methods, this *READY* signal indicates when the output value is valid. For Action methods, there is also an input *ENABLE* signal by which the external circuit indicates to the module when it is using the method. In the current socket protocol spec, there are no such *READY*s and *ENABLE*s. A request/response must be supplied *on every clock* (using NOP opcodes if necessary), and can be advanced when an *accept* signal is asserted from the other side. We implement this in BSV by *exposing* all these signals as actual method arguments and results, and asserting that the methods *are used on every clock*. We say that the methods are *always ready* and *always enabled*.

For a method to be always ready it cannot, in turn, invoke any other methods that may not be ready. Thus, if a method must enqueue into or dequeue out of a FIFO, then the usual *implicit* conditions on those methods must be sacrificed. We say that such FIFOs have *unguarded* enqueue or dequeue operations. These unguarded operations must be used with care, by always using them inside explicit conditional statements that check whether it is ok to enqueue or dequeue, respectively.

The new version of the file **EdgeFIFOs.bsv** (in FinalVersion.v4/Sources) now contains variants of the Pipeline and Bypass FIFOs with *unguarded* enqueue and dequeue operations. In each variant, note that only one end (e.g., enqueue or dequeue) is unguarded, whereas the other end is guarded as usual. We use the unguarded ends at the interfaces to the external world, keeping the more robust, guarded semantics for the inward-facing ends. Also, note that **EdgeFIFOs.bsv** is still just a library package, not at all specific to the current SoC switch design. It has wide applicability in a number of designs.

In the file **Socket\_IFC.bsv**, we now redefine *Socket\_master\_ifc* and *Socket\_slave\_ifc* to expose all the requests, responses and *accept* signals explicitly. *Socket\_master\_ifc* consists of two sub-interfaces, *Socket\_master\_req\_ifc* and *Socket\_master\_resp\_ifc*, for the request side and response side, respectively

The *Socket\_master\_req\_ifc* and *Socket\_master\_resp\_ifc* both have an *always\_ready* attribute (this is

applied in **Switch.bsv**). This Bluespec compiler will verify that this is true, i.e., that these methods do not depend on any conditions. The compiler will also remove the thereby redundant READY wires. The *Socket\_master\_resp\_ifc* also has an *always\_enabled* attribute (also applied in **Switch.bsv**). Once again, the Bluespec compiler will verify this at any use of this interface, and will remove the ENABLE wire.

The function *fifos\_to\_master\_ifc* is redefined so that it uses unguarded FIFO interfaces and fully encapsulates the socket signaling protocol, i.e.,

- The generation of a request on every cycle, inserting a NOP request if a real request is not available
- Acceptance of the *accept* signal for requests and advancing to the next request
- Acceptance of a response on every cycle, discarding any NOP responses
- Generation of the response *accept* signal on every cycle

Similarly, we also define the new *Socket\_slave\_ifc* with its sub-interfaces *Socket\_slave\_req\_ifc* and *Socket\_slave\_resp\_ifc*, and we define the function *fifos\_to\_slave\_ifc* that fully encapsulates the socket signalling protocol. The result is that the interfaces will contain *exactly* the desired wires—no more, no less—specified by the socket protocol interface.

All these facilities for the socket signaling protocol are fully reusable in any block that has a similar socket. In fact, we shall reuse them immediately, below, to fix up the testbench in **Tb.bsv**.

Finally, in **Tb.bsv**, we define *mkConnection* to work on the new socket interface *Socket\_master\_ifc* and *Socket\_slave\_ifc*, and vice versa. This illustrates the use of BSV's powerful, user-extensible *overloading* mechanism. I.e., if you think of *mkConnection* as a module that encapsulates all the state and behavior necessary to connect an interface of type  $T_1$  to an interface of  $T_2$ , then here we have just extended the overloading of *mkConnection* to also work on the types *Socket\_master\_ifc* and *Socket\_slave\_ifc*. Thus, in the top-level module *mkTb* at the top of this file, we did not have to touch the source code—lines such as *mkConnection (initiator\_0, switch.initiator\_0)* automatically (through overloading resolution) instantiate the right module to connect our new master and slave interface types.

In the file **Switch.bsv**, the sole difference is to replace the previous *mkPipelineFIFO* and *mkBypassFIFO* instances with versions that have unguarded enqueue and dequeue operations facing the exterior.

Even with all these refinements, the final versions of **Socket\_IFC.bsv** and **Switch.bsv** together remain about 500 lines of source code (Version 3, without the impedance-matching changes, is 368 lines).

In the file **Tb.bsv**, the *mkInitiatorModel* and *mkTargetModel* modules similarly just replace *mkFIFO* by *mkPipelineFIFO* with suitable unguarded ends pointing towards the switch. Because we redefined *fifos\_to\_master\_ifc* and *fifos\_to\_slave\_ifc* in **Socket\_IFC.bsv**, we get, for free, the socket signaling protocol at the interfaces for the initiators and targets, without any change in the interface part of the modules' source code.

In Version 4, we also provide two more testbenches, **Tb1.bsv** and **Tb2.bsv**. In **Tb1.bsv**, instead of generating random traffic, initiator 0 pumps RD requests to target 0 and initiator 1 pumps RD requests to target 1. By running this test and observing the outputs we verify the following:

- We achieve full bandwidth in both directions, from initiators to targets and back. After an initial start up transient, both initiators send requests on every cycle, both targets receive requests and send responses on every cycle, and both initiators receive responses on every cycle. Since requests from the two initiators go to separate targets, there is no arbitration necessary.
- We meet the latency specification in both directions, i.e., requests and responses spend just one cycle buffered in the switch.
- The socket signaling protocol (request/accept and response/accept) is working.

In **Tb2.bsv**, both initiators pump RD requests to target 0. By running this test and observing the outputs we verify the following:

- Round-robin arbitration is working correctly, i.e., target 0 alternately gets requests from initiator 0 and initiator 1, respectively.
- We still achieve full bandwidth, despite the arbitration, i.e., target 0 receives requests and sends responses on every cycle. Of course, because of the flow control inherent in the socket protocols and in the arbitration, each initiator is only able to send a request and receive a response every other cycle.

In all these tests, item order (request or response) between any particular pair of endpoints is preserved.

## DC Synthesis Results

For details on the synthesis results, including a Verilog netlist for the design and information about the runs, please refer to the directory: Basic2x2Interconnect/FinalVersion.v4/DCSynthesis. The design was synthesized for both 65 nm and 180 nm.

Here were the main assumptions used for 65 nm:

- Design Compiler Version: X-2005.09-SP3
- Technology: TSMC CLN65GP process
- Library: Artisan 10-Track Advantage (SS Process Spice Models, 0.90V, 125 Degrees C)
- Added Input/Output delay = 25% of clock period.

And the results for 65 nm were as follows:

- Frequency: 1.11 GHz
- Area: 9.81 K gates, 15.70  $\mu\text{m}^2$  (1 gate = 1.60  $\mu\text{m}^2$ )

Here were the main assumptions used for 180 nm:

- Design Compiler Version: X-2005.09-SP3
- Technology: TSMC CL018G process
- Library: Artisan SAGE-X (SS Process Spice Models, 1.62V, 125 Degrees C)
- Added Input/Output delay = 25% of clock period.

And the results for 180 nm were as follows:

- Frequency: 380 MHz
- Area: 6.50 K gates, 64.87  $\mu\text{m}^2$  (1 gate = 9.98  $\mu\text{m}^2$ )

## Summary

The design of the 2x2 switch interconnect has shown a case study in design via refinement using Bluespec SystemVerilog (BSV). We were able to quickly produce a working, synthesizable model using BSV's powerful library facilities for buffering, interfaces and connections. Then, we systematically refined it in a series of steps, each involving a few highly *local* changes, to meet requirements on arbitration, latency, and exact socket signaling protocol. Even though the changes are textually local, the implications in the generated circuits can be more far-reaching, because a lot of control logic must accommodate the new protocols; however, these are automatically regenerated by the Bluespec compiler. Further, many of the components are highly reusable in other designs, i.e., facilities such as PipelineFIFOs and BypassFIFOs, and facilities encapsulating the details of the socket signalling protocol. For example, if we were now to design an actual initiator or target block, the question of interfacing to the

switch with the actual socket protocol is already solved, and in a robust, encapsulated way where the internals of the block are completely insulated from details of the socket signalling protocol.

All of this was only possible because of the high-level abstraction mechanisms, formal semantics and composability of the constructs in Bluespec SystemVerilog -- and, of course, we support these capabilities in SystemC as well.

For the first time, there is:

1. A solution that unifies virtual prototyping, architectural exploration and IC implementation.
2. A high-level solution for control and complex datapaths -- but with 100% of the control and quality of results of hand-coded RTL.

Hopefully you've been able to get a taste of why we say that Bluespec is a simpler and more scalable way of managing complex concurrency & interface protocols vs. RTL. We are 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.

This is where Bluespec shines. Everything else is just RTL. Can you afford not to take a closer look?

## Appendix A: SystemC Version

In the directory, AltFinalVersion\_ESE\_SystemC.v4, there is an alternative final version of the design written in SystemC with ESL extensions for rules and interface methods. The internal behavior of the two designs was structured slightly different – but you’ll see that it uses the same high-level semantic constructs for atomic transactions and transactional interfaces. The syntax is different – but the high-level semantics are the same.

With Bluespec, you work in the familiar syntax of your choice, SystemVerilog or SystemC. But, with either environment, you get the same high-level semantics – whether you are doing a design or model, you have a common way to express your design or model using atomic transactions and transactional interfaces.

Here are a few examples showing direct correspondence between the two implementations:

Example 1a: Interface for the Switch (in Bluespec SystemVerilog design)

```
interface Switch_IFC;
    // we have 2 slave interfaces towards the initiators
    interface Socket_slave_ifc initiator_0;
    interface Socket_slave_ifc initiator_1;

    // we have 2 master interfaces towards the targets
    interface Socket_master_ifc target_0;
    interface Socket_master_ifc target_1;
endinterface: Switch_IFC
```

Example 1b: Interface for the Switch (in SystemC design)

```
ESL_INTERFACE (Switch_IFC)
{
    ESL_SUBINTERFACE ( initiator_0, Socket_slave_ifc );
    ESL_SUBINTERFACE ( initiator_1, Socket_slave_ifc );

    ESL_SUBINTERFACE ( target_0, Socket_master_ifc );
    ESL_SUBINTERFACE ( target_1, Socket_master_ifc );
};
```

Example 2a: Address decode function (in Bluespec SystemVerilog design)

```
function Bool addr_is_for_target_0 (ReqAddr addr);
    return (addr[0] == 0);
endfunction
```

Example 2b: Address decode function (in SystemC design)

```
bool
addr_is_for_target_0 (ReqAddr addr)
{
    return ((addr & 0x1) == 0);
}
```

Example 3a: Instantiating state (in Bluespec SystemVerilog design)

```
// ---- for incoming requests (from initiators)
FIFO#(Socket_Req)    from_initiator_0    <- mkPipelineFIFO_ug_enq;
FIFO#(Socket_Req)    from_initiator_1    <- mkPipelineFIFO_ug_enq;
```

Example 3b: Instantiating state (in SystemC design)

```
// ---- for incoming requests (from initiators)
FIFO<Socket_Req>      *from_initiator_0;
FIFO<Socket_Req>      *from_initiator_1;
```