A Case Study in Design by Refinement using Bluespec SystemVerilog$^{(TM)}$

February 2006, www.bluespec.com

Abstract

In this document, we show how to design an SoC interconnect (a crossbar switch) between initiators and targets using Bluespec SystemVerilog. We begin 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. 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 (along with testbenches) for all four versions. A simple "diff" between corresponding 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 Rules and Rule-based Interfaces, and automatic regeneration of correct control logic as we substitute one component with another.

Problem specification

Suppose we need to design an bus/interconnect (a crossbar switch) for an SoC (System on a Chip), connecting multiple *initiators* to multiple *targets.* Examples of SoC initiators include processors, DSPs and DMA engines. Examples of SoC targets include memories, I/O blocks, and the DMA configuration port. For high performance, typically

- Requests (initiators to targets) are completely decoupled from responses (targets to initiators).
- Both requests and responses are pipelined through the switch.
- The switch should preserve request order from a particular initiator to a particular target, and response order from a particular target to a particular initiator.
- Not all initiators may send requests to all targets, i.e., the internal connectivity may be heterogeneous.
- For simultaneous requests from multiple initiators towards the same target, and for simultaneous responses from multiple targets towards the same initiator, there should be some arbitration scheme. Typically one specifies a *fair* arbitration scheme, such as round-robin, LRU (Least Recently Used) or Random, so that a continuous stream from one source does not indefinitely delay an item from another source. These parts of the switch are called *merge* points.
- There is usually some kind of *flow control* or *back pressure* which, in turn, implies some regime for buffering requests and responses. For example, a target may temporarily not be ready to accept a request or an initiator may temporarily not be ready to accept a response. Or, an arbitration decision favoring one source may imply that an item cannot temporarily be accepted from a competing source at that merge point. Requests and responses must not be lost ("dropped on the floor") as a result of such flow control. Ultimately, flow control will extend back into initiators (to stop them from sending requests) and into targets (to stop them from sending responses).
- There is typically a stringent latency requirement, i.e., a specification of how many cycles a request or a response takes to transit the switch, assuming that it is not delayed for arbitration or flow-control reasons.
- Each connection between an initiator or target and the switch must follow some specific *signalling* protocol.

In this paper we use a specific example whose specifications are:

- Two initiators and two targets (see left side of Figure 1 for overall structure).
- All merge points have round-robin arbitration.
- In the best case (i.e., if allowed by arbitration and absence of flow control), 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 (see right side of Figure 1):

- 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.

Figure 1 clarifies the architecture, and the terminology of *master* and *slave* interfaces. Note that master and slave interfaces occur on both initiator side and target side of the interconnect. Requests flow from a master to a slave, and responses flow from a slave to a master, on all four sockets.

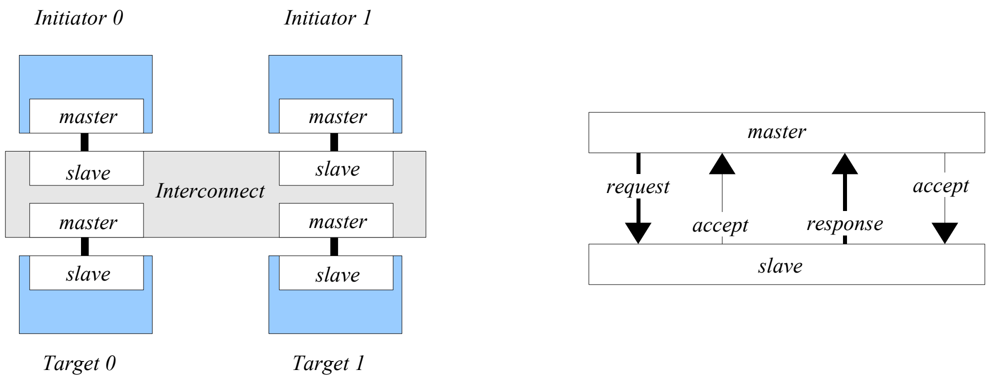

Figure 1 The SoC with a 2x2 crossbar interconnect (left) and master-slave signalling (right)

Introduction

In the following sections we 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 meets the round-robin scheduling requirement.
- Version 3 meets the 1-cycle latency requirement.
- Version 4 meets the socket signaling protocol requirement.

This document is distributed with the complete BSV source codes for the four versions (including testbenches and Makefiles), and the description herein is intended to be read while at least viewing the source codes. If you have a Bluespec installation, it is also instructive to execute the code and observe outputs. The four versions are in separate directories named v1, v2, v3 and v4, respectively. In case you only have this document and not the source codes in front of you, then reading this document will give you a flavor of the refinement methodology without the full experience; we urge you to obtain and read the source codes!

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. We strongly recommend using a program like "diff" on corresponding files in different versions to highlight what has changed. Because the changes are very local, these diffs make it easy to see the differences.

This document is not intended as a detailed tutorial on BSV. It assumes you already have some familiarity with BSV, and the emphasis is on the methodology of refinement. However, even if you are unfamiliar with BSV, we believe you should be able to follow the commentary and get the gist of the code.

Version 1

The file **Socket_IFC.bsv** defines the bit-true representations of requests and responses on each socket.

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 "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 simulated using any of the standard ways to simulate BSV programs. For example, it can be compiled using the Bluespec compiler *bsc* to Verilog, 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, i.e., each item spends at least one cycle in an incoming FIFO and one cycle in an outgoing FIFO.

Version 2

In this version, we add round-robin arbitration in file **Switch.bsv.** We recommend doing a "diff" between the **v1/Switch.bsv** and **v2/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 enables, 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.

Version 3

In this version 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 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.

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 READYs and ENABLEs. 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** 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. In *Socket_master_req_ifc,* note that we have separate methods *getReqOp, getReqInfo, getReqAddr* and *getReqData*, instead of a single method to get an entire *Socket_Req* struct. Either way is functionally fine and will result in the same number of wires and behavior in the generated Verilog. The advantage of separate methods is that wires in the generated Verilog are organized as separate ports with the method names, instead of a flat vector of wires, i.e., the Verilog is more readable. For exactly the same reason, in *Socket_master_resp_ifc,* the *putResp* method takes four separate arguments *respOp, respInfo, respAddr* and *respData,* instead of a single *Socket_resp* struct argument.

The *Socket_master_req_ifc* and *Socket_master_resp_ifc* both have an *always_ready* attribute. 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. Once again, the Bluespec compiler will verify this at any use of this interface, and will remove the thereby redundant 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 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 **Socket_IFC.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 550 lines of source code (Version 3, without the impedance-matching changes, is 362 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 form 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.

Summary

This white paper 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.

[Document revision: 15 February 2006 22:00 EST]