A Parameterized Model of a Crossbar Switch In Bluespec SystemVerilog(TM)

June 30, 2005

Introduction

This document describes the design of a highly parameterized, recursively defined crossbar switch, written in Bluespec SystemVerilog(TM) (BSV). The code shown here has been simulated and verified using Bluesim, synthesized to RTL using the Bluespec synthesis tools, and further to a netlist using a standard RTL-to-netlist synthesis tool. Our objective is to demonstrate that even synthesizable designs can be expressed at a very high level in Bluespec SystemVerilog.

Switch architecture

There is a vast literature on the architecture of crossbar switches. Typically there is trade off between the size of hardware resources (gates, flip flops, wires) and measures of congestion, such as bisection bandwidth. For example, at one extreme, a bus has minimal hardware, but high congestion, while at the other extreme, a full NxN interconnect has maximal hardware and least congestion. Popular points in between include "logarithmic" switches, such as butterflies and fat trees (N log(N) hardware).

In this demonstration we show a butterfly switch which, because of its recursive description, is arguably one of the more difficult structures to express in a hardware description language. Figure 1 shows the basic building blocks: a 1-in, 1-out element (a FIFO) and a 2-in, 1-out merge block.

1x1
2x1 merge

Figure 1 Basic building blocks

Figure 2 illustrates a 4x4 butterfly, built recursively from 2x2 butterflies which, in turn, are built from 1x1 butterflies. A 1x1 butterfly (each block in the leftmost column) is just a 1-in, 1-out FIFO. Given two 1x1 butterflies, we build a 2x2 butterfly by adding a column of two 2x1 merges to their right. We connect each output of the 1x1 butterfly to two places: one input of a merge in the same row, and the other input of a corresponding merge in the "other half" of the column. Given two 2x2 butterflies, we build a 4x4 butterfly by adding a column of four 2x1 merges on the right. We connect each output of the 2x2 butterflies to two places: one input of a merge in the same row, and the other input of a corresponding merge in the "other half" of the column. The generalization to the NxN case is follows this general pattern.

Note that there is a route from every input to every output. Routing is very simple: if a packet currently in row a_1a_0 is destined for output b_1b_0 (where these are binary representations of row numbers), then after passing through column j, if $a_j = b_j$ the packet goes straight through, else it goes to the other destination.

Butterfly switch sizes are NxN, where N is a power of 2 (please see "Other Parameterizations" section for discussion of switches of other sizes).

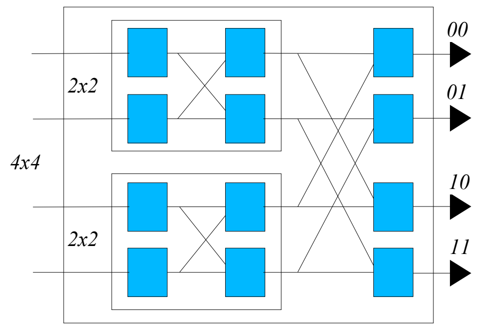

Figure 2 A 4x4 butterfly, built recursively from 2x2 butterflies

Parameterization

The code shown below is parameterized in the following ways:

- "N", the size of the NxN switch. Essentially, the module constructor "generates" a switch of the desired size during static elaboration.
- The type "t" of items (packets) passing through the switch.
- Although we parameterize by the packet type "t", we need some way to determine the desired destination of each packet in order to route it through the switch. We achieve this with another parameter: a function from "t" to integers which, when applied to a packet, returns its destination output port number. In many applications, this would just extract a field from the packet header, but by parameterizing it this way, more general routing functions are possible. This function represents a combinational circuit.
- The butterfly construction module is parameterized by another module, the 2-to-1 merge module. This is because different 2-to-1 merge modules can implement different arbitration policies (when two packets collide at the merge), different queuing disciplines, etc. We demonstrate two particular 2x1 merge modules, one with a simple static priority across the 2 inputs, and one with fair arbitration, using an LRU (Least Recently Used) policy.

Note: everything remains synthesizable, despite all this parameterization. Further, there is no hardware overhead for expressing it in this high-level manner.

The Bluespec SystemVerilog code for the butterfly

The Butterfly module is implemented in one file, **XBar.bsv,** less than 200 lines of BSV code (including two versions of the 2-to-1 merge module).

There is also a simple test bench, T**b.bsv**, about 150 lines of code, which sends and receives various patterns of packets, demonstrating that all routes work, that packets flow through in parallel when there is no congestion, and demonstrating the LRU arbitration when there is congestion.

The entire code is included in the Appendix of this document. We will now do a high-level walk-through of the code. This is not intended to be a detailed tutorial on BSV. Rather, it is meant to be just enough commentary so that the reader gets a plausible idea of what is going on in the code. The goal is to persuade the reader that these facilities of BSV provide a way to express synthesizable designs at a very *high level*, even though this way is radically different from traditional methods associated with the terms "behavioral synthesis" or "high-level synthesis".

XBar.bsv

All the code in the file is enclosed within "package-endpackage" brackets, standard SV notation for grouping together related definitions and controlling visibility of names. After a few import statements to make certain library packages visible, the file is in two major sections, separated by "=====" lines. The first section describes the basic building block, i.e., the 2-to-1 merge interface and two possible module implementations of it. The second section describes the butterfly itself.

A basic building block: a 2-to-1 merge

The *interface Merge2x1 ... endinterface* section specifies the interface of the basic 2-to-1 merge building block. It contains three sub-interfaces: two *Put* interfaces *iport0* and *iport1* for the two inputs, and one *Get* interface *oport* for the single output. *Get* and *Put* are standard interface types from the BSV library (they are written in BSV). *Get* interfaces have a single *get* method using which one can get/receive a value from a module, and *Put* interfaces have a single *put* method using which one can put/send a value to a module. Note that the interface is parameterized by type *t*, i.e., it is generic, or polymorphic.

The *module mkMerge2x1_static ... endmodule* section describes one particular module implementing the above interface. As in Verilog, it is a module constructor, i.e., it can be instantiated more than once. It instantiates a single FIFO *f* containing items of type *t,* and then defines the required three interface methods. The *iport0. put* and *iport1.put* methods simply enqueue into the FIFO, and the *oport.get* method simply dequeues from the FIFO. Note that the methods do not test for whether the FIFO is full or empty. These tests are *implicit* in the FIFO methods, i.e., the BSV compiler will take care of it, generating the implied control circuits. The BSV compiler will recognize that only one enqueue operation can be performed at a time, and will automatically introduce a static priority arbitration. (It is possible to tell the compiler which of the two static priorities it should use.) The *provisos (Bits#(t,wt))* phrase at the top of the module is a constraint that the type *t* must be representable in bits. Not all types in BSV need to be representable in bits. In particular, several types are used just during static elaboration, and have no run-time existence; these typically do not need to be representable in bits. The proviso is given here because the FIFO constructor *mkFIFO*, in turn, requires *t* to be representable in bits.

The *module mkMerge2x1_lru ... endmodule* section describes another particular module constructor implementing the same *Merge2x1* interface, this time with fair arbitration (LRU) between the two inputs. It instantiates three FIFOFs containing items of type *t.* A FIFOF is just like a FIFO, except that it also exposes the *notEmpty* and *notFull* conditions as boolean methods. This is necessary in order to do fair arbitration. The *iport0.put* and *iport1.put* methods simply enqueue into FIFOFs *fi0* and *fi1*, respectively, and the *oport.get* method simply dequeues from FIFO *fo.* The three rules take care of the situations where only *fi1* has data, only *fi0* has data, or both have data, respectively. Arbitration is only necessary in the latter case. This is done by examining the Boolean register *fi0HasPrio* and picking either *fi0* or *fi1* as the source accordingly, from which a packet is moved to *fo*. The rules also set the *fi0HasPrio* register appropriately. Note again that the *enq* operation does not check whether *fo* is full; again, this implicit condition is taken care of by the BSV compiler.

Similarly, other *mkMerge2x1_<policy>* modules can be written implementing other arbitration policies (see "Other Parameterizations" section below for more discussion).

The Butterfly Crossbar

The *interface XBar ... endinterface* section defines the interface for the crossbar. It contains a *List* of input ports, each with a *Put* interface, and a *List* of output ports, each with a *Get* interface. The whole interface is parameterized by the type *t* of items (packets) flowing through the crossbar.

The *function Bool flipCheck ... endfunction* pulls out, for convenience, the routing decision at each node. It takes the destination and source rows, and the column number, as arguments, and compares the appropriate bit to determine whether the packet should go straight through or be "flipped" to the "opposite half". The function represents a combinational circuit.

The *module mkXBar ... endmodule* section constitutes the rest of the package, and describes the construction of the butterfly. It has three parameters:

- *logn*, specifying the size of the NxN switch
- a function *destinationOf* which, when applied to a packet, returns its destination row
- the 2-to-1 merge module constructor

The module constructor *mkXBar* implements and exports/provides the *XBar#(t)* interface. The module constructor is recursive, following the recursive description given earlier.

After defining the two lists *iports* and *oports* for the input and output ports, respectively, there is a large *if-then-else* conditional that takes care of the base case (1x1 "switch") and the recursive case (NxN where N > 1). The base case simply instantiates a FIFO, and defines *iports* and *oports* to be singleton lists of ports that simply enqueue and dequeue into the FIFO.

In the recursive case, we first define some useful temporary variables, *n* and *nHalf*. Then, we recursively invoke the module being currently defined, *mkXBar,* to instantiate an *upper* half and a *lower* half, each of size N/2 x N/2. We define *iports* to be simply the concatenation of the input ports of the upper and lower halves. We define *oports_mid* to be the concatenation of the output ports of the upper and lower halves, as a convenient way to name and access all these ports. Next, we define *merges* to be a column of 2x1 merges of height N, using the given merge building block parameter and replicating it N times using the *replicateM* library function (which is written in BSV). Now, *oports* is just the output ports from these merges. Finally, the for-loop implements N BSV *rules*, one for each port in *oports_mid,* i.e., one for each "fork" point on the right-hand side of Figure 2. For each packet emerging from an *oport_mid* port, we use the *destinationOf* function to extract its destination, then we use the *flipCheck* function to determine its routing, and then we *put* it into the appropriate merge port.

Finally, we return *iports* and *oports* as this module's interface ports.

Note that the recursion in *mkXBar,* and the if-then-else that controls the recursion, are resolved during static elaboration. That is, when *mkXBar* is invoked with a particular N, it "generates" the full, nested hardware structure shown in Figure 2. There is no run-time implication or cost associated with this recursion and conditional.

Tb.bsv (a test bench)

We will not go through the test bench in detail, just a few interesting parts. The line

```
XBar#(int) xbar <- mkXBar (2, destOf, mkMerge2x1_lru);
```

instantiates a crossbar that carries *int* "packets" through. The parameter 2 specifies that it is a 4x4 crossbar (2^2x2^2). The *destOf* function is defined earlier in the file and just looks at the bottom 4 bits of the *int* as the destination. The *mkMerge2x1_lru* parameter is the module constructor for the merger with lru arbitration.

The section beginning *Stmt test_seq = (seq ...endseq);* specifies an FSM with sequential and parallel parts, to inject packets into the *xbar.* The parallel parts are intended to show that under no congestion, packets traverse the crossbar with the same latency, whereas with congestion, they get arbitrated using LRU and take different latencies.

The first *rule* starts off the FSM, injecting packets.

The final for-loop defines N rules, one per output port. Each rule, whenever it receives a packet, prints it out. For correctness, a packet whose bottom 4 bits are *x* must emerge from port *x,* no matter into which input port it was injected.

Simulation, Verification and Synthesis

The code, as shown (test bench and crossbar switch with LRU arbitration), has been simulated successfully in the Bluesim simulator, and produces the expected results. The tests succeeded on first run. There is a common aphorism amongst people who use languages with this level of types and abstraction: "Once you get the types right, it just works." This was certainly true in this design: all the hard work was in formulating the recursion and parameterization correctly to satisfy the type-checker and other static checks.

The code (test bench and crossbar switch) was then synthesized using the Bluespec synthesis tool into Verilog RTL, and this was tested successfully in a Verilog simulation. The outputs were verified to be the same as the Bluesim outputs. The RTL was visually examined to ensure that it had the expected structure.

The generated RTL, in turn, was further synthesized to a netlist using Magma tools. Both a 4x4 and an 8x8 instance of the butterfly switch easily met timing at 500 MHz using a TSMC 0.18 micron technology library.

Other Parameterizations

The following additional parameterizations would be quite straightforward.

Switches of different size

In general, one may want a switch of size MxN where M may not be equal to N and, and neither M nor N may be powers of 2. In this case, we simply build an RxR switch where R is the smallest power of two greater than or equal to M and N, and rely on dead-code elimination to remove unused hardware. The Bluespec compiler itself can do this, if the switch is used in a BSV context; otherwise, downstream RTL-to-netlist synthesis tools will do so.

If N is not a power of 2, what to do about "illegal" destinations in packets? There are several possibilities:

- Use BSV *enum* declarations for destinations fields in packet headers, which will guarantee statically, through type-checking, that illegal routes are impossible.
- Use assertions in the BSV code to catch illegal routes (illegal results of the *destinationOf* function).
- Add code in mkXBar to drop packets with illegal routes, or to convert them into default correct routes

More complex arbitration

In the sample code, there is a fixed arbitration scheme at every 2x1 merge (static, or lru). By further parameterizing the *mkMerge2x1* module constructor by the row and column in the butterfly where it is being instantiated, the arbitration can be customized separately for every instance. Again, there is no hardware penalty here—each instance will only have its chosen arbitration circuit.

The *mkMerge2x1* module can also be coded to examine the actual packets being arbitrated. For example, the packet headers may contain a priority field that affects arbitration. This can be done very elegantly using BSV's overloading mechanism (typeclasses). First, we add a proviso on the packet type *t* that says it should be in the *Ord* class, which ensures that the "<" and "<=" operations are defined on that type. Then, in *mkMerge2x1,* we simply use these operators to compare the priorities of two contending packets. Separately, for each instance of the butterfly crossbar for a given type *t*, we can define the meaning of the comparison operators.

Summary

This white paper has shown how designs can be written in Bluespec SystemVerilog (BSV) succinctly and correctly. The attributes of BSV that enable this include:

- A high-level description of behavior*: Rules*
- A high-level way to modularize behavior with consistent, transactional semantics: *Interface Methods*
- The power to parameterize one module with another module
- *Polymorphism*, or the ability to parameterize with types
- A mechanism to parameterize designs with circuit fragments, such as the *destinationOf* function
- A seamless way to mix in complex, parameterized static elaboration with behavioral descriptions

Many of these capabilities exist in high-level programming languages, and even in some so-called hardware *modeling* languages. But in BSV, these descriptions also have clear hardware semantics, and the Bluespec tool provides a demonstrated route to synthesizing such descriptions into quality hardware.

[Document revision: 1 July 2005 23:45 EDT]

Appendix: The file XBar.bsv

```
// Copyright (c) 2005 Bluespec, Inc.  All rights reserved.

// Crossbar switch using butterfly topology

package XBar;

import Monad :: *;
import FIFO :: *;
import FIFOF :: *;
import List :: *;
import GetPut :: *;

// ================================================================
// Basic building block: a 2-to-1 merge

interface Merge2x1 #(type t);
   interface Put#(t) iport0;
   interface Put#(t) iport1;
   interface Get#(t) oport;
endinterface

// ----------------
// An implementation of Merge2x1
// Arbitration on two inputs: static unspecified priority

module mkMerge2x1_static (Merge2x1#(t))
   provisos (Bits#(t, wt));

   FIFO#(t) f <- mkFIFO;

   interface iport0 = interface Put
                         method Action put (x);
                            f.enq (x);
                         endmethod
                      endinterface;
   interface iport1 = interface Put
                         method Action put (x);
                            f.enq (x);
                         endmethod
                      endinterface;
   interface oport =  interface Get
                         method ActionValue#(t) get;
                            f.deq;
                            return f.first;
                         endmethod
                      endinterface;
endmodule: mkMerge2x1_static

// ----------------
// An implementation of Merge2x1
// Arbitration on two inputs: LRU (fair)

module mkMerge2x1_lru (Merge2x1#(t))
   provisos (Bits#(t, wt));

   FIFOF#(t) fi0 <- mkFIFOF;
   FIFOF#(t) fi1 <- mkFIFOF;
   FIFOF#(t) fo  <- mkFIFOF;
   Reg#(Bool) fi0HasPrio <- mkReg (True);

   rule fi0_is_empty (! fi0.notEmpty);
      let x = fi1.first;
      fi1.deq;
      fo.enq (x);
      fi0HasPrio <= True;
   endrule

   rule fi1_is_empty (! fi1.notEmpty);
      let x = fi0.first;
      fi0.deq;
      fo.enq (x);
      fi0HasPrio <= False;
   endrule

   rule both_have_data (fi0.notEmpty && fi1.notEmpty);
      FIFOF#(t) fi = ((fi0HasPrio) ? fi0 : fi1);
      let x = fi.first;
      fi.deq;
      fo.enq (x);
      fi0HasPrio <= ! fi0HasPrio;
   endrule

   interface iport0 = interface Put
                         method Action put (x);
                            fi0.enq (x);
                         endmethod
                      endinterface;
   interface iport1 = interface Put
                         method Action put (x);
                            fi1.enq (x);
                         endmethod
                      endinterface;
   interface oport =  interface Get
                         method ActionValue#(t) get;
                            fo.deq;
                            return fo.first;
                         endmethod
                      endinterface;
endmodule: mkMerge2x1_lru

// ================================================================
// The XBar module, using the basic Merge2x1 building block

// ----------------
// The XBar module interface

interface XBar #(type t);
   interface List#(Put#(t))  input_ports;
   interface List#(Get#(t))  output_ports;
endinterface

// ----------------
// The routing function: decides whether the packet goes straight
// through, or gets "flipped" to the opposite side

function Bool flipCheck (Bit #(32) dst, Bit #(32) src, Integer logn);
    return (dst[fromInteger(logn-1)] != src [fromInteger(logn-1)]);
endfunction: flipCheck

// ----------------
// The XBar module constructor
```

```
module mkXBar #(Integer logn,
                function Bit #(32) destinationOf (t x),
                module #(Merge2x1 #(t)) mkMerge2x1)
              (XBar #(t))
   provisos (Bits#(t, wt));

   List#(Put#(t)) iports;
   List#(Get#(t)) oports;

   // ---- BASE CASE (n = 1 = 2^0)
   if (logn == 0) begin
      FIFO#(t) f <- mkFIFO;
      iports = cons (fifoToPut (f), nil);
      oports = cons (fifoToGet (f), nil);
   end

   // ---- RECURSIVE CASE (n = 2^logn, logn > 0)
   else begin
      Integer n     = exp(2, logn);
      Integer nHalf = div (n, 2);

      // Recursively create two switches of half size
      XBar#(t) upper <- mkXBar (logn-1, destinationOf, mkMerge2x1);
      XBar#(t) lower <- mkXBar (logn-1, destinationOf, mkMerge2x1);

      // input ports are just the input ports of upper and lower halves
      iports = append (upper.input_ports, lower.input_ports);

      // intermediate ports are output ports of upper and lower halves
      List#(Get#(t)) oports_mid =
               append (upper.output_ports, lower.output_ports);

      // Create new column of n 2x1 merges
      List#(Merge2x1#(t)) merges <- replicateM (n, mkMerge2x1);

      // output ports are just the output ports of the new merges
      oports = nil;
      for (Integer j = n-1; j >= 0; j = j - 1)
         oports = cons (merges [j].oport, oports);

      // Routing from each intermediate oport to new column
      for (Integer j = 0; j < n; j = j + 1) begin
         rule route;
            let x <- oports_mid [j].get;
            Bool flip = flipCheck (destinationOf (x), fromInteger (j), logn);
            let jFlipped = ((j < nHalf) ? j + nHalf : j - nHalf);
            if (! flip)
               merges [j]       .iport0.put (x);
            else
               merges [jFlipped].iport1.put (x);
         endrule
      end
   end

   interface input_ports  = iports;
   interface output_ports = oports;
endmodule: mkXBar

// ================================================================

endpackage: XBar
```

Appendix: The file Tb.bsv (testbench)

```
// Copyright (c) 2005 Bluespec, Inc.  All rights reserved.

// Testbench for cross-bar switch XBar.bsv

package Tb;

import Monad :: *;
import List :: *;
import GetPut :: *;
import StmtFSM :: *;

import XBar :: *;

function Bit#(32) destOf (int x);
   return pack (x) & 'hF;
endfunction

(* synthesize *)
module mkTb (Empty);

   // A register to count clock cycles, and a rule to increment it,
   // used in displaying time with inputs and outputs
   Reg#(int) ctr <- mkReg(0);
   rule inc_ctr;
      ctr <= ctr+1;
   endrule

   // Instantiate the DUT (4x4 switch, int packets, lru arbitration)
   // XBar#(int) xbar <- mkXBar (2, destOf, mkMerge2x1_static);
   XBar#(int) xbar <- mkXBar (2, destOf, mkMerge2x1_lru);

   // This function encapsulates the action of sending a datum x into input port i
   function Action enqueue (Integer i, int x);
      action
         xbar.input_ports[i].put (x);
         int ii = fromInteger (i);
         $display("%d: In:  (port %1d, val %h)", ctr, i, x);
      endaction
   endfunction

   // We define a sequence of actions to exercise the DUT.  (This is a
   // particularly simple example: the feature allows considerably more
   // complicated "programs" than this.)
   Stmt test_seq =
     (seq
         enqueue(0, 'h11);
         enqueue(0, 'h10);
         enqueue(0, 'h12);
         enqueue(0, 'h13);

         enqueue(1, 'h20);
         enqueue(1, 'h23);
         enqueue(1, 'h22);
         enqueue(1, 'h21);

         enqueue(2, 'h30);
         enqueue(2, 'h32);
         enqueue(2, 'h33);
         enqueue(2, 'h31);
         enqueue(3, 'h43);
         enqueue(3, 'h40);
         enqueue(3, 'h42);
         enqueue(3, 'h41);

         action    // no collisions
            enqueue(0, 'h50);
            enqueue(1, 'h51);
            enqueue(2, 'h52);
            enqueue(3, 'h53);
         endaction

         action    // no collisions
            enqueue(0, 'h62);
            enqueue(1, 'h63);
            enqueue(2, 'h60);
            enqueue(3, 'h61);
         endaction

         action    // no collisions
            enqueue(0, 'h71);
            enqueue(1, 'h70);
            enqueue(2, 'h73);
            enqueue(3, 'h72);
         endaction

         action    // collisions
            enqueue(0, 'h81);
            enqueue(1, 'h83);
            enqueue(2, 'h80);
            enqueue(3, 'h82);
         endaction

         // Test arbitration
         action
            enqueue(0, 'h900);
            enqueue(1, 'h910);
         endaction
         action
            enqueue(0, 'ha00);
            enqueue(1, 'ha10);
         endaction
         action
            enqueue(0, 'hb00);
            enqueue(1, 'hb10);
         endaction
         action
            enqueue(0, 'hc00);
            enqueue(1, 'hc10);
         endaction

         // ---- sentinel, to finish simulation
         enqueue (0, 'hFFF0);
      endseq);

   // Next we use this sequence as argument to a module which instantiates a
   // FSM to implement it.
   FSM test_fsm <- mkFSM(test_seq);

   // A register to control the start rule
   Reg#(Bool) going <- mkReg(False);

   // This rule kicks off the test FSM, which then runs to completion.
```

```
   rule start (!going);
      going <= True;
      test_fsm.start;
   endrule

   List#(Reg#(int)) xs <- replicateM (4, mkRegU);

   // Rules to dequeue items from each output port
   for (Integer oj = 0; oj < 4; oj = oj + 1) begin
      rule recv;
         let x <- xbar.output_ports[oj].get;
         (xs [oj]) <= x;
         int intoj = fromInteger (oj);
         $display("%d: Out:      (port %1d, val %h)", ctr, intoj, x);

         // ---- Finish when we see sentinel value
         if (x == 'hFFF0) $finish (0);
      endrule
   end

endmodule: mkTb

// ================================================================

endpackage: Tb
```