February 1, 2005

Automatic Generation of Control Logic with Bluespec SystemVerilog

Introduction

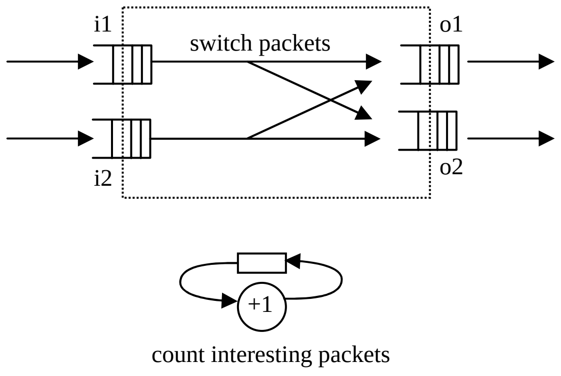
These notes accompany a small example illustrating the kind of control logic that is automatically generated for the designer by the Bluespec SystemVerilog (BSV) compiler. When coding in RTL (Verilog or VHDL), this logic would have to be designed and expressed explicitly by the designer. The benefits of BSV's automation are:

- Much less clutter in the source code, making it more readable
- Less likelihood of errors because of the reduced clutter and significantly smaller source code size
- Easier verification, because the design intent is not obscure as in RTL, and because the automatically generated code is correct-by-construction
- More changeable in the future, because of less clutter, clearer design intent, and automatic regeneration of correct control logic in the changed code

Of course, whenever evaluating such claims, it is important to remember that these benefits come into play particularly when scaling to large designs, especially when coding under severe time deadlines. Thus, when viewing this small example, the reader should think not just about how this particular example would be done explicitly in RTL, but should also constantly ask what would happen in real (much larger) designs.

The specification

The example problem is a small 2x2 switch that collects some statistics. The figure below shows the datapaths:



Data ("packets") arrive on two input FIFOs. Each packet is routed to one of two output FIFOs. Certain "interesting" packets (from both input FIFOs) are counted in a register. For this example, packets are 16-bit quantities. Routing is based on bit [0]. A packet is "interesting" if bits [3:1] are all zero.

A packet is available from an input FIFO if it is not empty. It can move into a destination output FIFO if it is not full. In addition, there are two kinds of potential "shared resource" collisions: when the packets at the head of both input FIFOs have the same destination, or when the packets at the head of both input FIFOs need to be counted (even if they have different destinations. Note: to illustrate an aspect of the compiler, the counter is specifically limited to only incrementing by one per clock cycle (for optimal performance, one would design this counter so that it could be incremented by one or two each cycle)). In case of a collision, the packet at the head of FIFO i1 should be given priority (i.e., the packet at the head of i2 is delayed).

The design should have maximum throughput, i.e., a packet should move if it can, modulo the above rules.

The BSV code

The file **SmallSwitch.bsv**, in Exhibit A, shows the Bluespec SystemVerilog (BSV) code for this example. Please view the code while reading the following:

Code walk-through

The code is bracketed by **package-endpackage**, which is a SystemVerilog mechanism for namespace control. The **import** section is a SystemVerilog mechanism for importing names from other packages. The **export** section describes which identifiers defined in this package will be visible when it is imported into other packages. The **typedef** just defines a shorthand for 16-bit values.

The **interface-endinterface** phrase defines a new interface type called IfcSmallSwitch. It contains four sub-interfaces and one **method**. The first sub-interface, called put_i1, has type Put#(B16). A Put#(*t*) interface is defined in the imported package GetPut and contains one method, called put(), into which a client can put a value of type *t*. Later in this code, we will connect this interface to the input FIFO i1. The remaining sub-interfaces are similar, connecting to the input FIFO i2 and output FIFOs, o1 and o2, respectively. The one method, get, is used to retrieve the value of the interesting-packet counter.

The rest of the code, bracketed by **module-endmodule,** defines the exported module mkSmallSwitch, which presents the IfcSmallSwitch interface, just defined, to its clients. The first section defines the internal state elements of the module: four FIFOs and a register. Each pair of lines uses the SystemVerilog idiom for instantiation: the first line instantiates an interface object, and the second line instantiates the module itself, and this also defines the interface for the module. The notation FIFO#(B16) represents the type of a FIFO interface, specialized to the type B16, i.e., such a FIFO can only hold values of type B16. Similarly, Reg#(int) represents the register interface type, specialized to int, i.e., such a register can only hold an int. This kind of parameterized strong typing is characteristic of SystemVerilog.

The two rules, each bracketed by **rule-endrule**, represent the desired behavior completely. In rule r1, i1.first invokes the method first() in the interface i1; it returns the value of the packet at the head of FIFO i1. The second line picks a destination FIFO, based on the routing condition (bit [0] of the packet). The third line dequeues the packet from the input FIFO, and the fourth line enqueues it into the chosen output FIFO. The final line counts the packet if it is "interesting" (bits [3:1] are zero). Rule r2 is similar.

The final section of the module defines its external interface, and is not important for this discussion.

Discussion

Implicit conditions: Note that there is no test for input FIFOs having available data (non-empty), nor for output FIFOs having available space (not-full). This is because in BSV, every interface method can have an implicit condition, defined by the module's implementer, which determines when it can be invoked. The FIFO implementer associates the non-empty condition with the first() and deq() methods, and the not-full condition with the enq() method. The BSV compiler takes these conditions into account when determining whether a rule can execute within the current clock. The user of these methods does not have to worry about them.

Thus, it is *impossible* in BSV to read a junk value at the head of an empty FIFO, to attempt to dequeue a value from an empty FIFO, or to attempt to enqueue a value into a full FIFO. *This kind of benefit is not unique to FIFOs – it is a uniform methodology applied to all methods, whether primitive or user-defined.* The meaning of these conditions is determined by the module implementer. For example, a memory module's read() method may be disabled during a refresh or table-maintenance operation. A calculation module's result-delivery method may be disabled while it is busy calculating.

When coding directly in RTL, such conditions are usually shown as extra ports (1-bit signals) in the module interface (e.g., NOT_FULL), and their relationships to other ports are documented only informally, and often incompletely (e.g., sample documentation: "ENQ_ENABLE *must not be asserted when* NOT_FULL *is false*"). The designer has the additional obligation to ensure that he is obeying all these constraints, and the verification engineer must confirm this with explicit tests.

Data multiplexing: The output FIFOs, o1 and o2, must have multiplexers in front of them to feed data from the input FIFOs, i1 and i2. The BSV code does not mention any such multiplexers – they are inserted automatically by the BSV compiler. Further, the logic to control these multiplexers is worked out automatically by the BSV compiler.

In general, all inputs to any shared resource must be multiplexed, and those multiplexers need control logic. When writing RTL, these must be expressed explicitly by the designer; when writing BSV, they are managed automatically by the compiler.

Control: The BSV code does not mention anything about the shared-resource collision conditions, whereas it would be very explicit (and possibly quite messy) in hand-written RTL. This is because the semantics of rules in BSV automatically takes care of consistent access to shared resources.

Rules in BSV have the property of *atomicity*, which is well-known in computer science (operating systems, databases, parallel programming, distributed systems, etc.) as a powerful technique to express consistent access to shared resources. Thus, even though rules can execute in parallel, the BSV compiler will insert the appropriate control logic to ensure that rules are suppressed *if they would violate atomicity.*

In other words, the BSV compiler guarantees that parallel execution is always consistent with one-rule-at-a-time execution, so that any correctness properties that are established by reasoning about one rule at a time in the source code are guaranteed to hold in the generated parallel hardware. Thus, it is *impossible to get race conditions* in the hardware due to a mismanagement of collisions at shared resources.

In the code, the single-attribute **descending_urgency** resolves the collision in favor of rule r1, per the problem specification. If this line was omitted, the compiler would pick a resolution and produce an information message during compilation (*the designer can always choose whether to accept it or to specify a different resolution*).

The generated RTL code

The file **mkSmallSwitch.v**, in Exhibit B, shows the generated Verilog 95 RTL code produced by the BSV compiler from the BSV source file, SmallSwitch.bsv. Please view the code while reading the following:

(Note: in practice, one rarely looks at the BSV-generated RTL, treating it instead like a compiler-generated "assembler". It is possible to debug directly at the source code. However, as this example illustrates, it is not difficult to read the generated RTL.)

The module header for mkSmallSwitch shows the ports. These are not important for the current discussion but, briefly, the CLK and RST_N signals were added implicitly (there is a whole separate topic of how clocks and resets can be managed in a disciplined manner using BSV, which is outside the scope of this discussion). The "put1" sub-interface has three associated ports: put_i1_put_1 for the data, RDY_put_i1_put for its implicit condition (corresponding to when the input FIFO i1 is not full), and EN_put_i1_put for the client to enable the enqueueing of data. The other sub-interfaces are similar.

Later in the file, the five state elements are instantiated (one register and four FIFOs). The BSV compiler can inline these, if desired.

The remaining lines in the file express the control and muxing logic. For example, the assignment to WILL_FIRE_RL_r1_F_F represents the sub-condition of rule r1 when:

- i1 is not empty (the_i1$EMPTY_N)
- The packet at the head of i1 is destined for o2 (_d26, i.e., the_i1$D_OUT[0])
- o2 is not full (the_o2$FULL_N)
- The packet is not interesting (! d_29, i.e., the_i1$D_OUT[3:1] == 3'd0) so that there can be no collision on the counter

Similarly, the code includes other WILL_FIRE_RL_... assignments for the other control conditions. Note that some WILL_FIRE_RL... assignments include a negation of other

WILL_FIRE_RL... conditions. These negations correspond to giving rule r1 priority when there is a collision.

The counter is enabled (assign the_c$EN =...) whenever either r1 fires or r2 fires and the packet is interesting.

Something is enabled on FIFO o1 (assign the_o1$ENQ =...) whenever either rule r1 fires or r2 fires and the destination is o1.

Data is multiplexed into FIFO o1 (assign the_o1$D_IN =...) from i2 whenever rule r2 fires and the destination is o1, and from i1 otherwise. Note that this includes situations where no data is transferred, but that is ok since the control on the FIFO's Enable signal will ensure that no junk is enqueued.

In summary: the generated RTL contains a lot of multiplexing and control logic generated automatically by the BSV compiler, which would normally have been written manually when coding in RTL. The control logic is a combination of implicit conditions of methods, together with rule scheduling logic that ensures atomicity.

Notice that the control conditions contribute to significant clutter in the RTL code, whereas the BSV source remains simple and readable. This is not just because the RTL is computer generated – it would be true even if the RTL were carefully hand-structured. The benefits of BSV’s automatic generation of control become even more significant as the complexity of the control conditions grows.

(Note: the control logic can contain some local redundancies. The BSV compiler does major common sub-expression optimizations and many logic optimizations, but leaves the final local logic optimization to the downstream RTL-to-GDS-II synthesis tool, which is better equipped to do this within the context of specific technology libraries.)

Control conditions may span module boundaries. In this example, the external conditions are the non-emptiness and not-fullness of the FIFO sub-modules. In general, in BSV, since method bodies can themselves contain method invocations, a rule's overall condition may transitively incorporate local conditions, a method's conditions, a method's method's conditions, and so on, across many modules. BSV semantics and methodology enables building such non-local control structures systematically, in a modular fashion.

Scalability

Now consider extensions to the specification:

- Some packets are multicast packets (e.g., bit [4] specifies multicast), i.e., they must be sent to both output FIFOs. Some packets must be dropped (sent to neither FIFO).
- Collision arbitration is more complex. E.g.,
    - If they collide on destination, FIFO i1 should get priority, else if they collide on the counter, FIFO i2 should get priority.
    - Arbitration should be fair.
- Instead of a single counter for interesting packets, there are several, for different kinds of

interesting packets. (See Exhibits D and E as an example of this change.)

- Counters are non-exclusive (e.g., a TCP packet is also an IP packet, so that certain increments imply certain other increments), so that some control conditions include others.
- MxN instead of 2x2, i.e., there are M input FIFOs and N input FIFOs. Now each possible subset of input FIFOs and counters can have a shared-resource collision. M and N are parameters, not constants.

Further, imagine that some of these extensions came 6 months after the original coding, when the designer has forgotten the details about the original coding.

All such extensions can be handled simply in BSV – the original code is straightforward, and the modifications are also straightforward. In RTL code, the original code would be cluttered, making it more difficult to understand and to change, and the resulting code would be even more cluttered with detail that is automatically generated for the designer by the BSV compiler.

Conclusion

In summary, the BSV methodology of expressing behavior using atomic *rules*, and expressing interfaces using *methods* with *implicit conditions*, results in source codes that are dramatically shorter, clearer and simpler. This reduction is not merely a textual trick; it encapsulates and formally expresses the deep property of *atomicity*, so that the compiler can automatically generate complex multiplexing and control logic that would otherwise have to be hand-coded. A whole class of race conditions is eliminated entirely because of the correct-by-construction automatic control logic generation.

In this way, while preserving the familiar RTL notations for expressing module hierarchy, the level of abstraction is raised substantially.

Appendix A – SmallSwitch.bsv

Appendix A contains the BSV source code for the small example described in these notes.

```
// Copyright 2000--2004 Bluespec, Inc.  All rights reserved.

package SmallSwitch;

 import FIFO::*;
 import GetPut::*;

 export IfcSmallSwitch(..);
 export mkSmallSwitch;
 export B16;

 // ----------------

 typedef bit [15:0] B16;

 // ----------------
 // The interface of of the module.

 interface IfcSmallSwitch;
     interface Put#(B16) put_i1;    // for input  FIFO i1
     interface Put#(B16) put_i2;    // for input  FIFO i2
     interface Get#(B16) get_o1;    // for output FIFO o1
     interface Get#(B16) get_o2;    // for output FIFO o2

     method int getCount;                // to retrieve the counter value
 endinterface: IfcSmallSwitch

 // ----------------

 (* synthesize *)    // Create a verilog module for this; i.e., don't inline
it.

 module mkSmallSwitch (IfcSmallSwitch);

     // ---- MODULE STATE ----

     FIFO#(B16) i1();         // i1      is interface to input FIFO
     mkFIFO the_i1(i1);       // the_i1  is the input FIFO instance

     FIFO#(B16) i2();         // ... similarly for i2 ...
     mkFIFO the_i2(i2);

     FIFO#(B16) o1();         // ... similarly for o1 ...
     mkFIFO the_o1(o1);

     FIFO#(B16) o2();         // ... similarly for o2 ...
     mkFIFO the_o2(o2);
     Reg#(int) c();            // c      is the counter register interface
     mkReg#(0) the_c(c);       // the_c  is the counter register instance

     // ---- RULES ----

     (* descending_urgency = "r1, r2" *)

     // Rule for moving packets from input FIFO i1
     rule r1;
         let x = i1.first;
         let out = ((x[0] == 0) ? o1 : o2);    // pick destination FIFO
         i1.deq;
         out.enq (x);
         if (x[3:1] == 0) c <= c + 1;
     endrule

     // Rule for moving packets from input FIFO i2
     rule r2;
         let x = i2.first;
         let out = ((x[0] == 0) ? o1 : o2);
         i2.deq;
         out.enq (x);
         if (x[3:1] == 0) c <= c + 1;
     endrule

     // ---- MODULE EXTERNAL INTERFACE ----

     interface put_i1 = fifoToPut (i1);
     interface put_i2 = fifoToPut (i2);
     interface get_o1 = fifoToGet (o1);
     interface get_o2 = fifoToGet (o2);

     method getCount;
       return c;
     endmethod: getCount

 endmodule: mkSmallSwitch

endpackage: SmallSwitch
```

Appendix B – mkSmallSwitch.v

This appendix contains the corresponding Verilog 95 RTL code generated by the BSV compiler for the small example described in these notes.

```
//
// Generated by Bluespec Compiler, version 3.8.32 (build 5329, 2004-09-21)
//
// On Wed Sep 29 13:51:39 EDT 2004
//
// Method conflict free info:
// [get_o1_get_avAction CF [getCount,
//                 get_o2_get_avValue,
//                 put_i1_put,
//                 put_i2_put],
//  get_o1_get_avValue CF [getCount, get_o2_get_avValue],
//  get_o2_get_avAction CF [getCount,
//                 get_o1_get_avAction,
//                 get_o1_get_avValue,
//                 put_i1_put,
//                 put_i2_put],
//  get_o2_get_avValue CF getCount,
//  put_i1_put CF [getCount,
//          get_o1_get_avValue,
//          get_o2_get_avValue],
//  put_i2_put CF [getCount,
//          get_o1_get_avValue,
//          get_o2_get_avValue,
//          put_i1_put],
//  get_o1_get_avValue SB get_o1_get_avAction,
//  get_o2_get_avValue SB get_o2_get_avAction]
//
// Ports:
// Name                         I/O  size props
// RDY_put_i1_put                 O     1 reg
// RDY_put_i2_put                 O     1 reg
// get_o1_get_avValue             O    16 reg
// RDY_get_o1_get_avValue         O     1 reg
// RDY_get_o1_get_avAction        O     1 reg
// get_o2_get_avValue             O    16 reg
// RDY_get_o2_get_avValue         O     1 reg
// RDY_get_o2_get_avAction        O     1 reg
// getCount                       O    32 reg
// RDY_getCount                   O     1 const
// CLK                            I     1 clock
// RST_N                          I     1 clock
// put_i1_put_1                   I    16
// put_i2_put_1                   I    16
// EN_put_i1_put                  I     1
// EN_put_i2_put                  I     1
// EN_get_o1_get_avAction         I     1
// EN_get_o2_get_avAction         I     1
//
//
module mkSmallSwitch(CLK,
                 RST_N,
                 put_i1_put_1,
                 put_i2_put_1,
                 EN_put_i1_put,
                 EN_put_i2_put,
                 EN_get_o1_get_avAction,
                 EN_get_o2_get_avAction,
                 RDY_put_i1_put,
                 RDY_put_i2_put,
                 get_o1_get_avValue,
                 RDY_get_o1_get_avValue,
                 RDY_get_o1_get_avAction,
                 get_o2_get_avValue,
                 RDY_get_o2_get_avValue,
                 RDY_get_o2_get_avAction,
                 getCount,
                 RDY_getCount);
  input  CLK;
  input  RST_N;
  input  [15 : 0] put_i1_put_1;
  input  [15 : 0] put_i2_put_1;
  input  EN_put_i1_put;
  input  EN_put_i2_put;
  input  EN_get_o1_get_avAction;
  input  EN_get_o2_get_avAction;
  output RDY_put_i1_put;
  output RDY_put_i2_put;
  output [15 : 0] get_o1_get_avValue;
  output RDY_get_o1_get_avValue;
  output RDY_get_o1_get_avAction;
  output [15 : 0] get_o2_get_avValue;
  output RDY_get_o2_get_avValue;
  output RDY_get_o2_get_avAction;
  output [31 : 0] getCount;
  output RDY_getCount;
  wire [31 : 0] getCount, the_c$D_IN, the_c$Q_OUT;
  wire [15 : 0] get_o1_get_avValue,
           get_o2_get_avValue,
           the_i1$D_IN,
           the_i1$D_OUT,
           the_i2$D_IN,
           the_i2$D_OUT,
           the_o1$D_IN,
           the_o1$D_OUT,
           the_o2$D_IN,
           the_o2$D_OUT;
  wire RDY_getCount,
       RDY_get_o1_get_avAction,
       RDY_get_o1_get_avValue,
       RDY_get_o2_get_avAction,
       RDY_get_o2_get_avValue,
       RDY_put_i1_put,
       RDY_put_i2_put,
       WILL_FIRE_RL_r1_F_F,
       WILL_FIRE_RL_r1_F_T,
       WILL_FIRE_RL_r1_T_F,
       WILL_FIRE_RL_r1_T_T,
       WILL_FIRE_RL_r2_F_F,
       WILL_FIRE_RL_r2_F_T,
       WILL_FIRE_RL_r2_T_F,
       WILL_FIRE_RL_r2_T_T,
       _d26,
       _d29,
       _d4,
       _d7,
       the_c$EN,
       the_i1$CLR,
       the_i1$DEQ,
       the_i1$EMPTY_N,
       the_i1$ENQ,
       the_i1$FULL_N,
       the_i2$CLR,
       the_i2$DEQ,
       the_i2$EMPTY_N,
       the_i2$ENQ,
       the_i2$FULL_N,
       the_o1$CLR,
       the_o1$DEQ,
       the_o1$EMPTY_N,
       the_o1$ENQ,
       the_o1$FULL_N,
       the_o2$CLR,
       the_o2$DEQ,
       the_o2$EMPTY_N,
       the_o2$ENQ,
       the_o2$FULL_N;
  RegN #(32, 0) the_c(.CLK(CLK),
                  .RST_N(RST_N),
                  .D_IN(the_c$D_IN),
                  .EN(the_c$EN),
                  .Q_OUT(the_c$Q_OUT));

  FIFO2 #(16) the_o2(.CLK(CLK),
                 .RST_N(RST_N),
                 .D_IN(the_o2$D_IN),
                 .ENQ(the_o2$ENQ),
                 .DEQ(the_o2$DEQ),
                 .D_OUT(the_o2$D_OUT),
                 .CLR(the_o2$CLR),
                 .FULL_N(the_o2$FULL_N),
                 .EMPTY_N(the_o2$EMPTY_N));

  FIFO2 #(16) the_i2(.CLK(CLK),
                 .RST_N(RST_N),
                 .D_IN(the_i2$D_IN),
                 .ENQ(the_i2$ENQ),
                 .DEQ(the_i2$DEQ),
                 .D_OUT(the_i2$D_OUT),
                 .CLR(the_i2$CLR),
                 .FULL_N(the_i2$FULL_N),
                 .EMPTY_N(the_i2$EMPTY_N));

  FIFO2 #(16) the_o1(.CLK(CLK),
                 .RST_N(RST_N),
                 .D_IN(the_o1$D_IN),
                 .ENQ(the_o1$ENQ),
                 .DEQ(the_o1$DEQ),
                 .D_OUT(the_o1$D_OUT),
                 .CLR(the_o1$CLR),
                 .FULL_N(the_o1$FULL_N),
                 .EMPTY_N(the_o1$EMPTY_N));

  FIFO2 #(16) the_i1(.CLK(CLK),
                 .RST_N(RST_N),
                 .D_IN(the_i1$D_IN),
                 .ENQ(the_i1$ENQ),
                 .DEQ(the_i1$DEQ),
                 .D_OUT(the_i1$D_OUT),
                 .CLR(the_i1$CLR),
                 .FULL_N(the_i1$FULL_N),
                 .EMPTY_N(the_i1$EMPTY_N));

  assign RDY_getCount = 1'd1 ;
  assign RDY_get_o1_get_avAction = the_o1$EMPTY_N ;
  assign RDY_get_o1_get_avValue = the_o1$EMPTY_N ;
  assign RDY_get_o2_get_avAction = the_o2$EMPTY_N ;
  assign RDY_get_o2_get_avValue = the_o2$EMPTY_N ;
  assign RDY_put_i1_put = the_i1$FULL_N ;
  assign RDY_put_i2_put = the_i2$FULL_N ;
  assign WILL_FIRE_RL_r1_F_F =
           the_i1$EMPTY_N && the_o2$FULL_N && _d26 && !_d29 ;
  assign WILL_FIRE_RL_r1_F_T =
           the_i1$EMPTY_N && the_o2$FULL_N && _d26 && _d29 ;
  assign WILL_FIRE_RL_r1_T_F =
           the_i1$EMPTY_N && the_o1$FULL_N && !_d26 && !_d29 ;
  assign WILL_FIRE_RL_r1_T_T =
           the_i1$EMPTY_N && the_o1$FULL_N && !_d26 && _d29 ;
  assign WILL_FIRE_RL_r2_F_F =
           the_i2$EMPTY_N && the_o2$FULL_N && _d4 && !_d7 &&
           !WILL_FIRE_RL_r1_F_F &&
           !WILL_FIRE_RL_r1_F_T ;
  assign WILL_FIRE_RL_r2_F_T =
           the_i2$EMPTY_N && the_o2$FULL_N && _d4 && _d7 &&
           !WILL_FIRE_RL_r1_F_F &&
           !WILL_FIRE_RL_r1_F_T &&
           !WILL_FIRE_RL_r1_T_T ;
  assign WILL_FIRE_RL_r2_T_F =
           the_i2$EMPTY_N && the_o1$FULL_N && !_d4 && !_d7 &&
           !WILL_FIRE_RL_r1_T_F &&
           !WILL_FIRE_RL_r1_T_T ;
  assign WILL_FIRE_RL_r2_T_T =
           the_i2$EMPTY_N && the_o1$FULL_N && !_d4 && _d7 &&
           !WILL_FIRE_RL_r1_F_T &&
           !WILL_FIRE_RL_r1_T_F &&
           !WILL_FIRE_RL_r1_T_T ;
  assign _d26 = the_i1$D_OUT[0] ;
  assign _d29 = the_i1$D_OUT[3:1] == 3'd0 ;
  assign _d4 = the_i2$D_OUT[0] ;
  assign _d7 = the_i2$D_OUT[3:1] == 3'd0 ;
  assign getCount = the_c$Q_OUT ;
  assign get_o1_get_avValue = the_o1$D_OUT ;
  assign get_o2_get_avValue = the_o2$D_OUT ;
  assign the_c$D_IN = the_c$Q_OUT + 32'd1 ;
  assign the_c$EN =
           WILL_FIRE_RL_r2_T_T || WILL_FIRE_RL_r2_F_T ||
           WILL_FIRE_RL_r1_T_T ||
           WILL_FIRE_RL_r1_F_T ;
  assign the_i1$CLR = 1'b0 ;
  assign the_i1$DEQ =
           WILL_FIRE_RL_r1_T_T || WILL_FIRE_RL_r1_T_F ||
           WILL_FIRE_RL_r1_F_T ||
           WILL_FIRE_RL_r1_F_F ;
  assign the_i1$D_IN = put_i1_put_1 ;
  assign the_i1$ENQ = EN_put_i1_put ;
  assign the_i2$CLR = 1'b0 ;
  assign the_i2$DEQ =
           WILL_FIRE_RL_r2_T_T || WILL_FIRE_RL_r2_T_F ||
           WILL_FIRE_RL_r2_F_T ||
           WILL_FIRE_RL_r2_F_F ;
  assign the_i2$D_IN = put_i2_put_1 ;
  assign the_i2$ENQ = EN_put_i2_put ;
  assign the_o1$CLR = 1'b0 ;
  assign the_o1$DEQ = EN_get_o1_get_avAction ;
  assign the_o1$D_IN =
           (WILL_FIRE_RL_r2_T_T || WILL_FIRE_RL_r2_T_F ?
            the_i2$D_OUT :
            the_i1$D_OUT) ;
  assign the_o1$ENQ =
           WILL_FIRE_RL_r2_T_T || WILL_FIRE_RL_r2_T_F ||
           WILL_FIRE_RL_r1_T_T ||
           WILL_FIRE_RL_r1_T_F ;
  assign the_o2$CLR = 1'b0 ;
  assign the_o2$DEQ = EN_get_o2_get_avAction ;
  assign the_o2$D_IN =
           (WILL_FIRE_RL_r2_F_T || WILL_FIRE_RL_r2_F_F ?
            the_i2$D_OUT :
            the_i1$D_OUT) ;
  assign the_o2$ENQ =
           WILL_FIRE_RL_r2_F_T || WILL_FIRE_RL_r2_F_F ||
           WILL_FIRE_RL_r1_F_T ||
           WILL_FIRE_RL_r1_F_F ;
```

```
endmodule  // mkSmallSwitch
```

Appendix C – HandSmallSwitch.v

This appendix includes an example hand-coded Verilog RTL design delivering the same functionality of the small switch described in the notes.

```
// inputs are
//               .     .     .     .     .
//                      ______
//  i1_put_rdy   ______/      \___________
//                                       _
//  i1_put       _______________________/
//  i1_put_data
//    (two cycle turn around from put ready to data drives
//    drop rdy, two cycles before full!

//
module HandSmallSwitch(CLK,
                       RST_N,

                       i1_put,
                       i1_put_data,
                       i1_put_rdy,

                       i2_put,
                       i2_put_data,
                       i2_put_rdy,

                       o1_get,
                       o1_get_data,
                       o1_get_rdy,

                       o2_get,
                       o2_get_data,
                       o2_get_rdy,

                   getCount);

   input          CLK, RST_N;
   input          i1_put;
   input [15:0]   i1_put_data;
   output         i1_put_rdy;
   input          i2_put;
   input [15:0]   i2_put_data;
   output         i2_put_rdy;
   input          o1_get;
   output [15:0]  o1_get_data;
   output         o1_get_rdy;
   input          o2_get;
   output [15:0]  o2_get_data;
   output         o2_get_rdy;
   output [31:0]  getCount;
```

```
////////////////////////////////////////
// output registers
reg            i1_put_rdy;
reg            i2_put_rdy;
reg [15:0]     o1_get_data;
reg            o1_get_rdy;
reg [15:0]     o2_get_data;
reg            o2_get_rdy;

////////////////////////////////////////
//
reg [15:0]     i1[0:7];
reg [2:0]      i1_wptr, i1_rptr, i1_cnt;

reg [15:0]     i2[0:7];
reg [2:0]      i2_wptr, i2_rptr, i2_cnt;

reg [15:0]     o1[0:7];
reg [2:0]      o1_wptr, o1_rptr, o1_cnt;

reg [15:0]     o2[0:7];
reg [2:0]      o2_wptr, o2_rptr, o2_cnt;

reg [31:0]     cntr;
reg            collide;

// lots of temp variables
// I like to define these as a default value and then change them later
reg            i1_cnt_dec,i1_cnt_inc,i1_full,i1_empty;
reg            i2_cnt_dec,i2_cnt_inc,i2_full,i2_empty;
reg            o1_cnt_dec,o1_cnt_inc,o1_full,o1_empty;
reg            o2_cnt_dec,o2_cnt_inc,o2_full,o2_empty;

reg [15:0]     i1_out, i2_out, write_o1_data, write_o2_data;
reg            interesting_i1, interesting_i2;
reg            write_o1, write_o2;


always @(posedge CLK or negedge RST_N) begin
   if (RST_N == 0) begin
      i1_wptr <= 0;
      i1_rptr <= 0;
      i1_cnt  <= 0;
      i2_wptr <= 0;
      i2_rptr <= 0;
      i2_cnt  <= 0;
      o1_wptr <= 0;
      o1_rptr <= 0;
      o1_cnt  <= 0;
      o2_wptr <= 0;
      o2_rptr <= 0;
      o2_cnt  <= 0;
      cntr    <= 0;
      i1_put_rdy  <= 0;
      i2_put_rdy  <= 0;

      o1_get_data <= 0;
      o1_get_rdy  <= 0;
      o2_get_data <= 0;
      o2_get_rdy  <= 0;
   end
   else begin
      // put data onto input fifos
      i1_cnt_dec = 0;
      i1_cnt_inc = 0;
      i1_empty   = (i1_cnt==0);
      i1_put_rdy <= (i1_cnt < 5);
      if (i1_put && (i1_cnt < 7)) begin
         i1[i1_wptr] <= i1_put_data;
         i1_cnt_inc = 1;
      end

      i2_cnt_dec = 0;
      i2_cnt_inc = 0;
      i2_full    = (i2_cnt==7);
      i2_empty   = (i2_cnt==0);
      i2_put_rdy <= (i2_cnt < 5);
      if (i2_put && (i2_cnt < 7)) begin
         i2[i2_wptr] <= i2_put_data;
         i2_cnt_inc = 1;
      end

      // move data from input fifos to output fifos
      // we need to calculate some state before deciding who gets written
      i1_out = i1[i1_rptr];
      i2_out = i2[i2_rptr];
      interesting_i1 = i1_out[3:1]==0;
      interesting_i2 = i2_out[3:1]==0;

      ////////////////////////////////////////////////////////////
      ////////////////////////////////////////////////////////////
      // moving data out of i1 has priority
      o1_cnt_inc = 0;
      o1_cnt_dec = 0;
      o1_full  = (o1_cnt == 7);
      o1_empty = (o1_cnt == 0);

      // collision - only let one or the other go if both
      // are going to cause the counter to increment
      // not a data problem, it's a "counter can only
      // increment by one" problem.  What fun, eh?
      collide = (!i1_empty && interesting_i1) &&
                (!i2_empty && interesting_i2);

      if (collide)
        i2_empty = 1;  // make it look like i2 is empty for now
```

```
// we have data to write from i1, write it somewhere
if (!o1_full) begin
   if (!i1_empty && !i1_out[0]) begin
      i1_cnt_dec = 1;
      o1[o1_wptr] <= i1_out;
      o1_cnt_inc = 1;
      if (interesting_i1) cntr <= cntr + 1;
   end
   else if (!i2_empty && !i2_out[0]) begin
      i2_cnt_dec = 1;
      o1[o1_wptr] <= i2_out;
      o1_cnt_inc = 1;
      if (interesting_i2) cntr <= cntr + 1;
   end
end

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
o2_cnt_inc = 0;
o2_cnt_dec = 0;
o2_full  = (o2_cnt == 7);
o2_empty = (o2_cnt == 0);

if (!o2_full) begin
   if (!i1_empty && i1_out[0]) begin
      i1_cnt_dec = 1;
      o2[o2_wptr] <= i1_out;
      o2_cnt_inc = 1;
   end
   else if (!i2_empty && i2_out[0]) begin
      i2_cnt_dec = 1;
      o2[o2_wptr] <= i2_out;
      o2_cnt_inc = 1;
   end
end

// pop data off input fifos and send them to where they need to go
i1_wptr <= i1_wptr + i1_cnt_inc;
i1_rptr <= i1_rptr + i1_cnt_dec;
i1_cnt  <= i1_cnt  + i1_cnt_inc - i1_cnt_dec;
i2_wptr <= i2_wptr + i2_cnt_inc;
i2_rptr <= i2_rptr + i2_cnt_dec;
i2_cnt  <= i2_cnt  + i2_cnt_inc - i2_cnt_dec;

// pop data off output fifos
o1_cnt_dec = o1_get && (o1_cnt!=0);
o1_rptr   <= o1_rptr + o1_cnt_dec;
o1_wptr   <= o1_wptr + o1_cnt_inc;
o1_cnt    <= o1_cnt  + o1_cnt_inc - o1_cnt_dec;

o2_cnt_dec = o2_get && (o2_cnt!=0);
o2_rptr   <= o2_rptr + o2_cnt_dec;
o2_wptr   <= o2_wptr + o2_cnt_inc;
```

```
        o2_cnt    <= o2_cnt  + o2_cnt_inc - o2_cnt_dec;

        o1_get_data <= (o1_cnt != 0) ? o1[o1_rptr] : 'hx;
        o1_get_rdy  <= (o1_cnt != 0);
        o2_get_data <= (o2_cnt != 0) ? o2[o2_rptr] : 'hx;
        o2_get_rdy  <= (o2_cnt != 0);
      end
   end

   assign getCount    = cntr;

endmodule
```

Appendix D – SmallSwitch3.bsv

This appendix includes a Bluespec SystemVerilog small switch design that has one change made to it:

- It has three special counters in the design tracking three different ‘special’ packets instead of one, as with the original small switch design

This design was based on SmallSwitch.bsv

```
// Copyright 2000--2004 Bluespec, Inc.  All rights reserved.

package SmallSwitch;

 import FIFO::*;
 import GetPut::*;

 export IfcSmallSwitch(..);
 export mkSmallSwitch3;
 export B16;

 // ----------------

 typedef bit [15:0] B16;

 // ----------------
 // The interface of of the module.

 interface IfcSmallSwitch;
     interface Put#(B16) put_i1;    // for input  FIFO i1
     interface Put#(B16) put_i2;    // for input  FIFO i2
     interface Get#(B16) get_o1;    // for output FIFO o1
     interface Get#(B16) get_o2;    // for output FIFO o2

     method int getCountc;                // to retrieve the counter value
     method int getCounte;                // to retrieve the counter value
     method int getCountf;                // to retrieve the counter value
 endinterface: IfcSmallSwitch

 // ----------------

 (* synthesize *)    // Create a verilog module for this; i.e., don't inline
it.

 module mkSmallSwitch3 (IfcSmallSwitch);

     // ---- MODULE STATE ----

     FIFO#(B16) i1();         // i1      is interface to input FIFO
     mkFIFO the_i1(i1);       // the_i1  is the input FIFO instance

     FIFO#(B16) i2();         // ... similarly for i2 ...
```

```
mkFIFO the_i2(i2);

FIFO#(B16) o1();         // ... similarly for o1 ...
mkFIFO the_o1(o1);

FIFO#(B16) o2();         // ... similarly for o2 ...
mkFIFO the_o2(o2);

Reg#(int) c();           // c      is the counter register interface
mkReg#(0) the_c(c);      // the_c  is the counter register instance
Reg#(int) d();           // d      is the counter register interface
mkReg#(0) the_d(d);      // the_d  is the counter register instance
Reg#(int) e();           // e      is the counter register interface
mkReg#(0) the_e(e);      // the_e  is the counter register instance

// ---- RULES ----

(* descending_urgency = "r1, r2" *)

// Rule for moving packets from input FIFO i1
rule r1;
    let x = i1.first;
    let out = ((x[0] == 0) ? o1 : o2);    // pick destination FIFO
    i1.deq;
    out.enq (x);
    if (x[3:1] == 0) c <= c + 1;
    if (x[3:1] == 0x3) d <= d + 1;
    if (x[3:1] == 0x5) e <= e + 1;
endrule

// Rule for moving packets from input FIFO i2
rule r2;
    let x = i2.first;
    let out = ((x[0] == 0) ? o1 : o2);
    i2.deq;
    out.enq (x);
    if (x[3:1] == 0) c <= c + 1;
    if (x[3:1] == 0x3) d <= d + 1;
    if (x[3:1] == 0x5) e <= e + 1;
endrule

// ---- MODULE EXTERNAL INTERFACE ----

interface put_i1 = fifoToPut (i1);
interface put_i2 = fifoToPut (i2);
interface get_o1 = fifoToGet (o1);
interface get_o2 = fifoToGet (o2);

method getCount;
  return c;
endmethod: getCount

method getCountd;
  return d;
```

```
     endmethod: getCountd

     method getCounte;
       return e;
     endmethod: getCounte

 endmodule: mkSmallSwitch3

endpackage: SmallSwitch
```

Appendix E – HandSmallSwitch3.v

This design was based on HandSmallSwitch.v, a hand-coded RTL design of the small switch example. In this derivative design, two additional counters were added for tracking ‘special’ packets.

Note how extensive the changes are made in the Verilog domain, from HandSmallSwitch.v to HandSmallSwitch3.v, versus those made with Bluespec SystemVerilog, from SmallSwitch.bsv to SmallSwitch3.bsv.

```
// inputs are
//               .     .     .     .     .
//                      ______
//  i1_put_rdy   ______/      \___________
//                                       _
//  i1_put       _______________________/
//  i1_put_data
//    (two cycle turn around from put ready to data drives
//    drop rdy, two cycles before full!

//
module HandSmallSwitch3(CLK,
                       RST_N,

                       i1_put,
                       i1_put_data,
                       i1_put_rdy,

                       i2_put,
                       i2_put_data,
                       i2_put_rdy,

                       o1_get,
                       o1_get_data,
                       o1_get_rdy,

                       o2_get,
                       o2_get_data,
                       o2_get_rdy,

              getCount0,
              getCount1,
              getCount2
                       );


   input          CLK, RST_N;
   input          i1_put;
   input [15:0]   i1_put_data;
   output         i1_put_rdy;
   input          i2_put;
   input [15:0]   i2_put_data;
   output         i2_put_rdy;
   input          o1_get;
   output [15:0]  o1_get_data;
   output         o1_get_rdy;
   input          o2_get;
   output [15:0]  o2_get_data;
   output         o2_get_rdy;
   output [31:0]  getCount0;
   output [31:0]  getCount1;
   output [31:0]  getCount2;

   ////////////////////////////////////////
   // output registers
   reg            i1_put_rdy;
   reg            i2_put_rdy;
   reg [15:0]     o1_get_data;
   reg            o1_get_rdy;
   reg [15:0]     o2_get_data;
   reg            o2_get_rdy;

   ////////////////////////////////////////
   //
   reg [15:0]     i1[0:7];
   reg [2:0]      i1_wptr, i1_rptr, i1_cnt;

   reg [15:0]     i2[0:7];
   reg [2:0]      i2_wptr, i2_rptr, i2_cnt;

   reg [15:0]     o1[0:7];
   reg [2:0]      o1_wptr, o1_rptr, o1_cnt;

   reg [15:0]     o2[0:7];
   reg [2:0]      o2_wptr, o2_rptr, o2_cnt;

   reg [31:0]     cntr0, cntr1, cntr2;
   reg            collidea, collideb, collidec;

   // lots of temp variables
   // I like to define these as a default value and then change them later
   reg            i1_cnt_dec,i1_cnt_inc,i1_full,i1_empty;
   reg            i2_cnt_dec,i2_cnt_inc,i2_full,i2_empty;
   reg            o1_cnt_dec,o1_cnt_inc,o1_full,o1_empty;
   reg            o2_cnt_dec,o2_cnt_inc,o2_full,o2_empty;


   reg [15:0]     i1_out, i2_out, write_o1_data, write_o2_data;
   reg            interesting_i1a, interesting_i2a;
   reg            interesting_i1b, interesting_i2b;
   reg            interesting_i1c, interesting_i2c;
   reg             write_o1, write_o2;


   always @(posedge CLK or negedge RST_N) begin
      if (RST_N == 0) begin
         i1_wptr <= 0;
         i1_rptr <= 0;
         i1_cnt  <= 0;
         i2_wptr <= 0;
         i2_rptr <= 0;
         i2_cnt  <= 0;
         o1_wptr <= 0;
         o1_rptr <= 0;
         o1_cnt  <= 0;
         o2_wptr <= 0;
         o2_rptr <= 0;
         o2_cnt  <= 0;
         cntr0   <= 0;
         cntr1   <= 0;
         cntr2   <= 0;

         i1_put_rdy  <= 0;
         i2_put_rdy  <= 0;

         o1_get_data <= 0;
         o1_get_rdy  <= 0;
         o2_get_data <= 0;
         o2_get_rdy  <= 0;
      end
      else begin
         // put data onto input fifos
         i1_cnt_dec = 0;
         i1_cnt_inc = 0;
         i1_empty   = (i1_cnt==0);
         i1_put_rdy <= (i1_cnt < 5);
         if (i1_put && (i1_cnt < 7)) begin
            i1[i1_wptr] <= i1_put_data;
            i1_cnt_inc = 1;
         end

         i2_cnt_dec = 0;
         i2_cnt_inc = 0;
         i2_full    = (i2_cnt==7);
         i2_empty   = (i2_cnt==0);
         i2_put_rdy <= (i2_cnt < 5);
         if (i2_put && (i2_cnt < 7)) begin
            i2[i2_wptr] <= i2_put_data;
            i2_cnt_inc = 1;
         end

         // move data from input fifos to output fifos
         // we need to calculate some state before deciding who gets written
         i1_out = i1[i1_rptr];
         i2_out = i2[i2_rptr];
```

```
interesting_i1a = i1_out[3:1]==0;
interesting_i1b = i1_out[3:1]==2;
interesting_i1c = i1_out[3:1]==3;

interesting_i2a = i2_out[3:1]==0;
interesting_i2b = i2_out[3:1]==2;
interesting_i2c = i2_out[3:1]==3;

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// moving data out of i1 has priority
o1_cnt_inc = 0;
o1_cnt_dec = 0;
o1_full  = (o1_cnt == 7);
o1_empty = (o1_cnt == 0);

// collision - only let one or the other go if both
// are going to cause the counter to increment
// not a data problem, it's a "counter can only
// increment by one" problem.  What fun, eh?
collidea = (!i1_empty && interesting_i1a) &&
           (!i2_empty && interesting_i2a);
collideb = (!i1_empty && interesting_i1b) &&
           (!i2_empty && interesting_i2b);
collidec = (!i1_empty && interesting_i1c) &&
           (!i2_empty && interesting_i2c);

if (collidea || collideb || collidec)
  i2_empty = 1;  // make it look like i2 is empty for now


// we have data to write from i1, write it somewhere
if (!o1_full) begin
   if (!i1_empty && !i1_out[0]) begin
      i1_cnt_dec = 1;
      o1[o1_wptr] <= i1_out;
      o1_cnt_inc = 1;
      if (interesting_i1a) cntr0 <= cntr0 + 1;
      if (interesting_i1b) cntr1 <= cntr1 + 1;
      if (interesting_i1c) cntr2 <= cntr2 + 1;
   end
   else if (!i2_empty && !i2_out[0]) begin
      i2_cnt_dec = 1;
      o1[o1_wptr] <= i2_out;
      o1_cnt_inc = 1;
      if (interesting_i2a) cntr0 <= cntr0 + 1;
      if (interesting_i2b) cntr1 <= cntr1 + 1;
      if (interesting_i2c) cntr2 <= cntr2 + 1;
   end
end

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
o2_cnt_inc = 0;
```

```
         o2_cnt_dec = 0;
         o2_full  = (o2_cnt == 7);
         o2_empty = (o2_cnt == 0);

         if (!o2_full) begin
            if (!i1_empty && i1_out[0]) begin
               i1_cnt_dec = 1;
               o2[o2_wptr] <= i1_out;
               o2_cnt_inc = 1;
            end
            else if (!i2_empty && i2_out[0]) begin
               i2_cnt_dec = 1;
               o2[o2_wptr] <= i2_out;
               o2_cnt_inc = 1;
            end
         end

         // pop data off input fifos and send them to where they need to go
         i1_wptr <= i1_wptr + i1_cnt_inc;
         i1_rptr <= i1_rptr + i1_cnt_dec;
         i1_cnt  <= i1_cnt  + i1_cnt_inc - i1_cnt_dec;
         i2_wptr <= i2_wptr + i2_cnt_inc;
         i2_rptr <= i2_rptr + i2_cnt_dec;
         i2_cnt  <= i2_cnt  + i2_cnt_inc - i2_cnt_dec;

         // pop data off output fifos
         o1_cnt_dec = o1_get && (o1_cnt!=0);
         o1_rptr   <= o1_rptr + o1_cnt_dec;
         o1_wptr   <= o1_wptr + o1_cnt_inc;
         o1_cnt    <= o1_cnt  + o1_cnt_inc - o1_cnt_dec;

         o2_cnt_dec = o2_get && (o2_cnt!=0);
         o2_rptr   <= o2_rptr + o2_cnt_dec;
         o2_wptr   <= o2_wptr + o2_cnt_inc;
         o2_cnt    <= o2_cnt  + o2_cnt_inc - o2_cnt_dec;

         o1_get_data <= (o1_cnt != 0) ? o1[o1_rptr] : 'hx;
         o1_get_rdy  <= (o1_cnt != 0);
         o2_get_data <= (o2_cnt != 0) ? o2[o2_rptr] : 'hx;
         o2_get_rdy  <= (o2_cnt != 0);
      end
   end

   assign getCount0   = cntr0;
   assign getCount1   = cntr1;
   assign getCount2   = cntr2;

endmodule
```

Appendix F – GCD.bsv

We provide, without comment, source code (**GCD.bsv**) and, in Appendix G, generated RTL code (**mkGCD.v**) for a much simpler example module. Its “start” message gives it two input numbers. It computes their Greatest Common Divisor, using Euclid's algorithm, and delivers the result using the “result” method. Euclid's algorithm to compute the GCD is:

```
GCD (x,y) =   while (y > 0)
                      if (x > y) swap x and y
                      else y := y – x
              return x
```

Note that the start and result method definitions have implicit condition (y == 0). This ensures that they cannot be invoked when the module is busy computing the GCD of the most recent inputs.

```
// Copyright 2000--2004 Bluespec, Inc.  All rights reserved.

package GCD;

export ArithIO_IFC(..);
export mkGCD;

// The following is a general interface for modules that do arithmetic
// on two inputs and produce one output.

interface ArithIO_IFC #(type aTyp); // ArithIO_IFC for any aTyp
    method Action start(aTyp num1, aTyp num2);
    method aTyp result();
endinterface: ArithIO_IFC

// The following is an attribute that tells the compiler to generate
// separate code for mkGCD

(* synthesize *)

module mkGCD(ArithIO_IFC#(int)); // here aTyp is defined to be type Int

    // ---- MODULE STATE ----

    Reg#(int) x();      // x is the interface to the register
    mkReg#(?) the_x(x); // the_x is the register instance
                        // ? means: unspecified initial value
    Reg#(int) y();
    mkReg#(0) the_y(y);

    // ---- RULES ----

    rule swap (x > y && y != 0);
        x <= y;
        y <= x;
    endrule

    rule subtract (x <= y && y != 0);
        y <= y - x;
    endrule

    // ---- MODULE EXTERNAL INTERFACE ----

    method Action start(int num1, int num2) if (y == 0);
        action
            x <= num1;
            y <= num2;
        endaction
    endmethod: start

    method int result() if (y == 0);
        result = x;
    endmethod: result

endmodule: mkGCD

endpackage: GCD
```

Appendix G – mkGCD.v

```
//
// Generated by Bluespec Compiler, version 3.8.32 (build 5329, 2004-09-21)
//
// On Wed Sep 29 13:52:04 EDT 2004
//
// Method conflict free info:
// [result SB start]
//
// Ports:
// Name                         I/O  size props
// RDY_start                      O     1
// result                         O    32 reg
// RDY_result                     O     1
// CLK                            I     1 clock
// RST_N                          I     1 clock
// start_num1                     I    32
// start_num2                     I    32
// EN_start                       I     1
//
//
module mkGCD(CLK,
           RST_N,
           start_num1,
           start_num2,
           EN_start,
           RDY_start,
           result,
           RDY_result);
  input  CLK;
  input  RST_N;
  input  [31 : 0] start_num1;
  input  [31 : 0] start_num2;
  input  EN_start;
  output RDY_start;
  output [31 : 0] result;
  output RDY_result;
  wire [31 : 0] result,
            the_x$D_IN,
            the_x$Q_OUT,
            the_y$D_IN,
            the_y$Q_OUT;
  wire RDY_result,
       RDY_start,
       WILL_FIRE_RL_subtract,
       WILL_FIRE_RL_swap,
       _d3,
       _d4,
       the_x$EN,
       the_y$EN;
  RegN #(32, 32'hAAAAAAAA) the_x(.CLK(CLK),
                        .RST_N(RST_N),
                        .D_IN(the_x$D_IN),
                        .EN(the_x$EN),
                        .Q_OUT(the_x$Q_OUT));

  RegN #(32, 0) the_y(.CLK(CLK),
                .RST_N(RST_N),
                .D_IN(the_y$D_IN),
                .EN(the_y$EN),
                .Q_OUT(the_y$Q_OUT));

  assign RDY_result = _d4 ;
  assign RDY_start = _d4 ;
  assign WILL_FIRE_RL_subtract = _d3 && !_d4 ;
  assign WILL_FIRE_RL_swap = !_d3 && !_d4 ;
  assign _d3 =
           (the_x$Q_OUT ^ 32'h80000000) <=
           (the_y$Q_OUT ^ 32'h80000000) ;
  assign _d4 = the_y$Q_OUT == 32'd0 ;
  assign result = the_x$Q_OUT ;
  assign the_x$D_IN = (EN_start ? start_num1 : the_y$Q_OUT) ;
  assign the_x$EN = EN_start || WILL_FIRE_RL_swap ;
  assign the_y$D_IN =
           {32{EN_start}} & start_num2 |
           {32{WILL_FIRE_RL_swap}} & the_x$Q_OUT |
           {32{WILL_FIRE_RL_subtract}} & the_y$Q_OUT - the_x$Q_OUT ;
  assign the_y$EN =
           EN_start || WILL_FIRE_RL_swap || WILL_FIRE_RL_subtract ;
endmodule  // mkGCD
```