Bluespec SystemVerilog for IP Delivery and Effective RTL Debug

Comprehending someone else's RTL is a significant challenge. Successfully implementing a change in someone else's RTL, without error, is even harder. To succeed, designers must work methodically through the RTL code, familiarize themselves with the design style, catch coding subtleties, and fully understand the architecture and micro-architecture – this is no small task with RTL.

Bluespec is a fundamentally better IP delivery vehicle than RTL:

- Bluespec source code represents a specification of the design and is significantly more readable
- Bluespec source code is far easier to change and enhance safely
- Bluespec designs retain powerful capabilities to parameterize, far greater than available in RTL technologies. This allows IP vendors and designers to incorporate many anticipated customizations as a core part of the design itself
- Designers have control into the RTL over the naming, inlining of various constructs, changes in module hierarchy, inclusion of debug signals, initialization conditions as well as directives for simulation output
- Although Bluespec designs are significantly higher quality than RTL, customers still anticipate having to fix bugs in vendor supplied IP. In addition to being higher quality, Bluespec designs are significantly easier to correct
- Within a Bluespec environment, Bluespec interfaces can guarantee proper IP connectivity, as they are self-documenting with implicit assertions, formally checked at compile time

Bluespec generates well-structured, readable, interoperable and predictable Verilog RTL:

- Bluespec RTL directly correlates to Bluespec source, following the identical architecture and micro-architecture, and neither adding nor deleting state elements
- Bluespec's generated RTL code is organized with a consistent and clear structure. Design elements and comments are organized into clear groupings such as modules, interfaces, state elements, scheduling logic and combinational logic – this simplifies both the understanding of the generated code as well as the ease of making changes
- Naming can be controlled from the source and is passed through into the RTL
- Comments, in addition to those automatically generated by the compiler, can be entered into the source for inclusion in the RTL addressing module headers, state elements, and rules
- Particularly for those selling IP, an active focus is required to ensure that the Verilog constructs and techniques will be accepted by the EDA tools with which it will be used. Bluespec automates the generation of Verilog RTL with proven, broad interoperability.

Bluespec enables multiple IP delivery options:

Bluespec SystemVerilog for use by end-customer
For this delivery model, you supply your design in Bluespec SystemVerilog. This model provides IP customers with the greatest flexibility: for parameterization, customizations, and bug fixes, if warranted. Optionally, you can also provide a Bluespec toolset, with a limited license for use only with the IP as delivered plus custom parameter settings. The limited use licenses will be provided by Bluespec at no charge.

Bluespec source code as specification and Verilog RTL for use by end-customer
For this delivery model, you supply a high-level specification of the implementation in the form of Bluespec SystemVerilog source and documented, clearly organized Verilog RTL code to your customer. Note that since the Verilog is automatically generated, the specification is guaranteed to be in sync with the RTL.

What does the generated Verilog RTL output look like?
To get a feel for the quality and source correlation of Bluespec's generated Verilog RTL, please review the following Bluespec SystemVerilog design example and the associated, tool-generated Verilog RTL code. For illustrative purposes only, this mkWidget module implements a simplistic, shift-and-add algorithm to multiply of two numbers.

Bluespec SystemVerilog design example: mkWidget

```
package Widget;

//
// Interface to the multiplier module
//
typedef Bit#(16) Tin;
typedef Bit#(32) Tout;

interface Widget_IFC;
   (*ready = "StartIsReady", enable = "StartShouldGo", prefix = ""*)
    method Action  start (Tin m1, Tin m2);
// Leave the rest of the methods for standard naming by BSV
    method Tout    result();
    method Action acknowledge();
endinterface


//
// Simple (naive) binary multiplier
//
 (* synthesize *)

(* doc = "This module performs a simple (naive) multiplication of two input values" *)
module mkWidget( Widget_IFC );


//
// State elements
//
(* doc = "The 'product' register holds the result of the multiply" *)
   Reg#(Tout)    product   <- mkReg(0);

(* doc = "The 'mcand' reg holds the value of the multiplicand, which is the intermediate result" *)
   Reg#(Tout)    mcand     <- mkReg(0);

(* doc = "The 'mplr' reg holds the value of the multiplier" *)
   Reg#(Tin)     mplr      <- mkReg(0);

(* doc = "The 'available' reg indicates whether the unit is currently available for further calculations" *)
   Reg#(Bool)    available <- mkReg(True);


(* doc = "The 'cycle' rule defines the core functionality of the multiply" *)
(* doc = "The rule does shift and adds to perform each stage of the multiply. \n The rule ‘fires’ (executes) on
any cycle that mplr!=0.\n If the LSB of the mplr is 1, then mcand is added to the interim calculation.  \nOn every
cycle, mcand is shifted left and mplr is shifted right." *)

// This rule will ‘fire’ or run every cycle that (mplr != 0)
   rule cycle ( mplr != 0 );
      let localsum = product+mcand;
      if (mplr[0] == 1) product <= localsum;
      mcand   <= mcand << 1;
      mplr    <= mplr  >> 1;
      $display("rule cycle just fired!");
   endrule


//
// Interface Methods
//
   method Action start(Tin m1, Tin m2) if (mplr == 0 && available);
       product <= 0;
       mcand   <= {0, m1};
       mplr    <= m2;
       available <= False;
   endmethod

   method Tout result() if (mplr == 0);
      return product;
   endmethod

   method Action acknowledge() if (mplr == 0 && !available);
      available <= True;
   endmethod

endmodule : mkWidget

endpackage : Widget
```

Verilog for mkWidget produced by Bluespec's compiler

```
//
// Generated by Bluespec Compiler, version 3.8.67 (build 8255, 2006-04-11)
//
// On Wed May 10 13:51:13 EDT 2006
//
// Method conflict free info:
// [result CF [acknowledge, result], start CF acknowledge, result SB start]
//
// Ports:
// Name                         I/O  size props
// StartIsReady                   O     1
// result                         O    32 reg
// RDY_result                     O     1
// RDY_acknowledge                O     1
// CLK                            I     1
// RST_N                          I     1
// m1                             I    16
// m2                             I    16
// StartShouldGo                  I     1
// EN_acknowledge                 I     1
//
// No combinational paths from inputs to outputs
//
//
// This module performs a simple (naive) multiplication of two input values
//
//

`ifdef BSV_ASSIGNMENT_DELAY
`else
`define BSV_ASSIGNMENT_DELAY
`endif

module mkWidget(CLK,
                RST_N,

                m1,
                m2,
                StartShouldGo,
                StartIsReady,

                result,
                RDY_result,

                EN_acknowledge,
                RDY_acknowledge);
  input  CLK;
  input  RST_N;

  // action method start
  input  [15 : 0] m1;
  input  [15 : 0] m2;
  input  StartShouldGo;
  output StartIsReady;

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

  // action method acknowledge
  input  EN_acknowledge;
  output RDY_acknowledge;

  // signals for module outputs
  wire [31 : 0] result;
  wire RDY_acknowledge, RDY_result, StartIsReady;

  // register available
  // The 'available' reg indicates whether the unit is currently available for further calculations
  reg available;
  wire available$D_IN, available$EN;

  // register mcand
  // The 'mcand' reg holds the value of the multiplicand, which is the intermediate result
  reg [31 : 0] mcand;
  wire [31 : 0] mcand$D_IN;
  wire mcand$EN;

  // register mplr
  // The 'mplr' reg holds the value of the multiplier
  reg [15 : 0] mplr;
  wire [15 : 0] mplr$D_IN;
```

```
wire mplr$EN;

// register product
// The 'product' register holds the result of the multiply
reg [31 : 0] product;
wire [31 : 0] product$D_IN;
wire product$EN;

// rule scheduling signals
wire CAN_FIRE_RL_cycle,
     CAN_FIRE_acknowledge,
     CAN_FIRE_start,
     WILL_FIRE_RL_cycle,
     WILL_FIRE_acknowledge,
     WILL_FIRE_start;

// inputs to muxes for submodule ports
wire [31 : 0] MUX_mcand$write_1__VAL_1,
              MUX_mcand$write_1__VAL_2,
              MUX_product$write_1__VAL_1;
wire [15 : 0] MUX_mplr$write_1__VAL_2;
wire MUX_product$write_1__SEL_1;

// action method start
assign StartIsReady = mplr == 16'd0 && available ;
assign CAN_FIRE_start = StartShouldGo ;
assign WILL_FIRE_start = StartShouldGo ;

// value method result
assign result = product ;
assign RDY_result = mplr == 16'd0 ;

// action method acknowledge
assign RDY_acknowledge = mplr == 16'd0 && !available ;
assign CAN_FIRE_acknowledge = EN_acknowledge ;
assign WILL_FIRE_acknowledge = EN_acknowledge ;

// rule RL_cycle
//   The 'cycle' rule defines the core functionality of the multiply
//   The rule does shift and adds to perform each stage of the multiply.
//    The rule fires (executes) on any cycle that mplr!=0.
//    If the LSB of the mplr is 1, then mcand is added to the interim calculation.
//   On every cycle, mcand is shifted left and mplr is shifted right.
assign CAN_FIRE_RL_cycle = mplr != 16'd0 ;
assign WILL_FIRE_RL_cycle = CAN_FIRE_RL_cycle ;

// inputs to muxes for submodule ports
assign MUX_product$write_1__SEL_1 = WILL_FIRE_RL_cycle && mplr[0] ;
assign MUX_mcand$write_1__VAL_1 = { 16'd0, m1 } ;
assign MUX_mcand$write_1__VAL_2 = { mcand[30:0], 1'd0 } ;
assign MUX_mplr$write_1__VAL_2 = { 1'd0, mplr[15:1] } ;
assign MUX_product$write_1__VAL_1 = product + mcand ;

// register available
assign available$D_IN = !StartShouldGo ;
assign available$EN = StartShouldGo || EN_acknowledge ;

// register mcand
assign mcand$D_IN =
           StartShouldGo ?
             MUX_mcand$write_1__VAL_1 :
             MUX_mcand$write_1__VAL_2 ;
assign mcand$EN = StartShouldGo || WILL_FIRE_RL_cycle ;

// register mplr
assign mplr$D_IN = StartShouldGo ? m2 : MUX_mplr$write_1__VAL_2 ;
assign mplr$EN = StartShouldGo || WILL_FIRE_RL_cycle ;

// register product
assign product$D_IN =
           MUX_product$write_1__SEL_1 ? MUX_product$write_1__VAL_1 : 32'd0 ;
assign product$EN = WILL_FIRE_RL_cycle && mplr[0] || StartShouldGo ;

// handling of inlined registers

always@(posedge CLK)
begin
  if (!RST_N)
    begin
      available <= `BSV_ASSIGNMENT_DELAY 1'd1;
      mcand <= `BSV_ASSIGNMENT_DELAY 32'd0;
      mplr <= `BSV_ASSIGNMENT_DELAY 16'd0;
      product <= `BSV_ASSIGNMENT_DELAY 32'd0;
    end
  else
    begin
```

```
        if (available$EN) available <= `BSV_ASSIGNMENT_DELAY available$D_IN;
        if (mcand$EN) mcand <= `BSV_ASSIGNMENT_DELAY mcand$D_IN;
        if (mplr$EN) mplr <= `BSV_ASSIGNMENT_DELAY mplr$D_IN;
        if (product$EN) product <= `BSV_ASSIGNMENT_DELAY product$D_IN;
      end
  end

  // synopsys translate_off
  `ifdef BSV_NO_INITIAL_BLOCKS
  `else // not BSV_NO_INITIAL_BLOCKS
  initial
  begin
    available = 1'b0 /* unspecified value */ ;
    mcand = 32'hAAAAAAAA /* unspecified value */ ;
    mplr = 16'b1010101010101010 /* unspecified value */ ;
    product = 32'hAAAAAAAA /* unspecified value */ ;
  end
  `endif // BSV_NO_INITIAL_BLOCKS
  // synopsys translate_on

  // handling of system tasks

  // synopsys translate_off
  always@(negedge CLK)
  begin
    #0;
    if (RST_N) if (WILL_FIRE_RL_cycle) $display("rule cycle just fired!");
  end
  // synopsys translate_on
endmodule  // mkWidget
```