```
// Parallel-serial signed-signed multiplier
// Copyright (C) Markus Nentwig 2014
// Distributed under the MIT licence.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
// THE SOFTWARE.
module sk61_parMulSignedSigned(clk, inSigned_A, inSigned_B, in_start, out_ready, outSigned_AB,
 out_valid);
   input clk;
   input in_start;
   parameter width = 0; // must be >= 3
   localparam width_AB = 2*width;

   input [width-1:0] inSigned_A;
   input [width-1:0] inSigned_B;
   output reg [width_AB-1:0] outSigned_AB;
   output reg                out_valid;
   reg                       ready = 1;
   output                    out_ready;

   // input registers
   reg [width-1:0]           A;
   reg [width-1:0]           B;

   // state counter
   //   Xilinx ISE 14.7 does not recognize that state <= width.
   //   A possible optimization for fixed width:
   //   - write out states 2, 3, ..., width-1 explicitly in place of "default" case
   //   - use "s ynthesis full_case" attribute
   (* FSM_ENCODING="ONE-HOT", SAFE_IMPLEMENTATION="NO" *) reg [$clog2(width+1)-1:0]
       state = 0;

   // working register
   reg [width-1:0]           work;

   // output register
   reg [width-2:0]           resultLsb;

   // "ready" output (note: zero delay on in_start)
   assign out_ready = ready & !in_start;

   // partial product
   wire [width-1:0] pp = {width{A[0]}} & B;

   // modified partial product in all but final row
   wire [width-1:0] pMod1 = {~pp[width-1], pp[width-2:0]};

   // modified partial product in final row
   wire [width-1:0] pMod2 = ~pMod1;

   // iteration
`ifdef SK61LIB_FASTERMUL
   // variant 1
   //   - straightforward implementation of algorithm
   //   - easier to read
   //   - uses separate adder for final row (high resource cost)
   //   - slightly faster
   wire [width:0]   sum1 = work + pMod1;
   wire [width:0]   sum2 = work + pMod2;
`else
   // variant 2
   //   - share adder with final row
   //   - use this by default
```

```
   wire [width:0]   sum12 = work + ((state == width) ? pMod2 : pMod1);
   wire [width:0]   sum1 = sum12;
   wire [width:0]   sum2 = sum12;
`endif

   always @(posedge clk) begin
      out_valid <= 0; // default

      // shift argument
      A <= {1'bx, A[width-1:1]};
      case (state)
        0: begin
           // ************************************************************
           // idle
           // ************************************************************
           // wait here
        end
        1: begin
           // ************************************************************
           // startup
           // ************************************************************
           work <= {1'b1, pMod1[width-1:1]};
           resultLsb <= {pMod1[0], {width-2{1'bx}}};
           state <= state + 1;
        end
        default: begin
           // ************************************************************
           // non-final row
           // ************************************************************
           // accumulate and shift
           work <= sum1[width:1];

           // LSB drops out and shifts into result
           resultLsb <= {sum1[0], resultLsb[width-2:1]};

           state <= state + 1;
        end
        width: begin
           // ************************************************************
           // final row
           // ************************************************************

           // combine work register and LSB part into result
           // add 1 at highest position and discard carry => invert
           outSigned_AB <= {~sum2[width], sum2[width-1:0], resultLsb};
           out_valid <= 1;
           work <= 'bx;
           state <= 0;
           resultLsb <= 'bx;
           B <= 'bx;
           ready <= 1;
        end
      endcase

      if (in_start) begin
         // ************************************************************
         // startup
         // ************************************************************
         if (ready == 0) begin
            $display("multiplier started while not ready");
            $finish();
         end
         A <= inSigned_A;
         B <= inSigned_B;
         state <= 1;
         ready <= 0;
      end
   end
endmodule
```