Tim Menninger

HW9

EE119a

For whatever reason, no software I've tried to download has worked for me. The only way I've been able to test was with timing simulations in Quartus, but haven't been able to run any test bench files. That said, my solutions seem to be correct, but if they don't run with the test benches, let me know and when I come back on Saturday, I'll fix them when I can use the machines in Moore. The issue is likely something with nomenclature.

Notes:
RLLEncoder: Without running the tests, I don't know if this is correct. But from reading the test bench, it looks like it assumes the first output bit lags the first input bit by 5 clocks. My solution only lagged by four clocks. The hard copy is this 4-clock-delayed solution, but I submitted both that file as well as an identical file with a dummy wait flag when START goes active. This causes a fifth bit lag. This is in the RLLEnc5.vhd file.

SerialMultiplier: I wrote a test bench the way I would have if I were testing, but I haven't run it for aforementioned reasons.

```
 1  -- bring in the necessary packages
 2  library  ieee;
 3  use  ieee.std_logic_1164.all;
 4  use  ieee.numeric_std.all;
 5  use  ieee.std_logic_unsigned.all;
 6
 7  ------------------------------------------------------------------------------------------
 8  --
 9  --  n-bit Bit-Serial Multiplier
10  --
11  --  This is an implementation of an n-bit bit serial multiplier.  The
12  --  calculation will take 2n^2 clocks after the START signal is activated.
13  --  The multiplier is implemented with a single adder.  This file contains
14  --  only the entity declaration for the multiplier.
15  --
16  --  Parameters:
17  --      numbits - number of bits in the multiplicand and multiplier (n)
18  --
19  --  Inputs:
20  --      A       - n-bit unsigned multiplicand
21  --      B       - n-bit unsigned multiplier
22  --      START   - active high signal indicating a multiplication is to start
23  --      CLK     - clock input (active high)
24  --
25  --  Outputs:
26  --      Q       - (2n-1)-bit product (multiplication result)
27  --      DONE    - active high signal indicating the multiplication is complete
28  --                and the Q output is valid
29  --
30  --
31  --  Revision History:
32  --      7 Apr 00  Glen George       Initial revision.
33  --     12 Apr 00  Glen George       Changed Q to be type buffer instead of
34  --                                  type out.
35  --     21 Nov 05  Glen George       Changed nobits to numbits for clarity
36  --                                  and updated comments.
37  --      7 Dec 16  Tim Menninger     Implemented serial multiplier state machine
38  --
39  ------------------------------------------------------------------------------------------
40
41  entity  BitSerialMultiplier  is
42
```

```
43      generic (
44          numbits  :  integer := 8    -- number of bits in the inputs
45      );
46
47      port (
48          A      :  in      std_logic_vector((numbits - 1) downto 0);     -- multiplicand
49          B      :  in      std_logic_vector((numbits - 1) downto 0);     -- multiplier
50          START  :  in      std_logic;                                    -- start calculation
51          CLK    :  in      std_logic;                                    -- clock
52          Q      :  buffer  std_logic_vector((2 * numbits - 1) downto 0); -- product
53          DONE   :  out     std_logic                                     -- calc completed
54      );
55
56  end  BitSerialMultiplier;
57
58  --
59  -- DataFlow architecture
60  --
61  -- Defines the state machine that performs the n-bit serial multiplier.
62  --
63  -- Inputs:  A - Multiplicand
64  --          B - Multiplier
65  --          START - Goes active when multiplication should start
66  --
67  -- Outputs: Q - Product of A and B
68  --          DONE - Set when the Q output is valid.
69  --
70  architecture DataFlow of BitSerialMultiplier is
71
72      --
73      -- Types
74      --
75      -- state: State machine for serial multiplier.
76      --      states:
77      --          idle     - Holds the output Q value, and asserts DONE.
78      --          setup    - Sets up local signals in preparation for serial multiplication
79      --          adder    - Full adder, used to compute carry bits and sum bits
80      --          multiply - Puts the low bit of the sum into the output and multiplies the
81      --                     multiplicand by next bit in multiplier
82      --
83      type multiplyState is (idle, setup, adder, multiply);
84
```

```
85      --
86      -- Signals
87      --
88      -- currentState (state): State machine controlling multiplication
89      -- Y (std_logic_vector): A copy of the original input, B, that we use to generate each
90      --      partial product.
91      -- carryBits (std_logic_vector): Contains the carry bits generated from the full adder
92      --      on each bit.  The low bit will always be 0, which helps facilitate adding the
93      --      low bit using the same full adder.
94      -- sumBits (std_logic_vector): Contains the sum bits generated from the full adder.  For
95      --      the first iteration, this will contain A.  After each set of full-adding, the
96      --      low bit of this is put into the output and then this is shifted right, leaving
97      --      the high bit 0 for the next addition.
98      -- partial (std_logic_vector): Contains the partial product, which is A if the current
99      --      bit in B (a.k.a Y) is 1, and 0 otherwise.
100     -- bitsAdded (integer): Counts how many of the bits we have full-added on a given
101     --      iteration.  On each addition, we need numbits full adder iterations.
102     -- bitsMultiplied (integer): Counts how many bits in the product have been set.  This
103     --      will count to 2*numbits.
104     --
105     signal currentState                 : multiplyState;
106     signal Y                            : std_logic_vector((numbits - 1) downto 0);
107     signal carryBits, sumBits, partial  : std_logic_vector(numbits downto 0);
108     signal bitsAdded                    : integer range 0 to numbits;
109     signal bitsMultiplied               : integer range 0 to 2*numbits;
110
111 begin
112     --
113     -- multiplyBits process
114     --
115     -- This process takes care of the state machine state and the DONE signal.  Whenever the
116     -- state machien is in its idle state, DONE is active.  The way to get out of this state
117     -- is if START is active.  It then goes to setup to initialize values, then for each
118     -- of 2*numBits multiply state occurrences (one for each product bit), it goes through
119     -- numBits+1 adder states with DONE inactive.  After the last multiply state, it goes
120     -- back to the idle state.  There is no reset.  Once the state machine leaves idle,
121     -- it will go through all states.  States only change on the rising edge of the global
122     -- CLK input.
123     --
124     -- This is the state machine that does multiplication.  It defines each state.  It is
125     -- left to another process to define when to be in which state.  The definition of each
126     -- state can be found in the state type header.
```

```
127     --
128     -- Inputs:  CLK - State machine only changes values on rising edge of CLK
129     --          A - Multiplicand for product
130     --          B - Multiplier for product
131     --
132     -- Outputs: Q - The product of A and B is generated and output in Q.
133     --
134     multiplyBits: process (CLK, A, B) is
135     begin
136         -- Only active on rising edge of CLK
137         if (rising_edge(CLK)) then
138             -- State machine
139             case currentState is
140                 ---------------------------
141                 -- idle state
142                 ---------------------------
143                 when idle =>
144                     DONE <= '1';
145                     -- Maintain Q
146                     Q <= Q;
147                     -- Using low 2 bits of B in partial products, so shift right twice before
148                     -- loading into Y register as initial value
149                     Y((numbits-3) downto 0) <= B((numbits-1) downto 2);
150                     Y((numbits-1) downto (numbits-2)) <= "00";
151                     -- Initialize everything else to 0, although values don't really matter
152                     carryBits <= std_logic_vector(to_unsigned(0, numbits+1));
153                     sumBits <= std_logic_vector(to_unsigned(0, numbits+1));
154                     partial <= std_logic_vector(to_unsigned(0, numbits+1));
155                     -- Set these to initial values
156                     bitsAdded <= 0;
157                     bitsMultiplied <= 0;
158                     -- Stay in idle state until START goes active
159                     if (START = '1') then
160                         currentState <= setup;
161                     else
162                         currentState <= idle;
163                     end if;
164                 ---------------------------
165                 -- setup state
166                 ---------------------------
167                 when setup =>
168                     DONE <= '0';
```

```
169                     -- Initialize Q output to all 0s
170                     Q <= std_logic_vector(to_unsigned(0, 2*numbits));
171                     -- Same initial value as in idle
172                     Y((numbits-3) downto 0) <= B((numbits-1) downto 2);
173                     Y((numbits-1) downto (numbits-2)) <= "00";
174                     -- Nothing carried yet
175                     carryBits <= (numbits downto 0 => '0');
176                     -- Sum bits will be partial product from low bit of B
177                     sumBits(numbits) <= '0';
178                     if (B(0) = '0') then
179                         sumBits((numbits - 1) downto 0) <=
180                             A and std_logic_vector(to_unsigned(0, numbits));
181                     else
182                         sumBits((numbits - 1) downto 0) <=
183                             A and std_logic_vector(to_unsigned(-1, numbits));
184                     end if;
185                     -- Partial bits will be partial product from second-lowest bit of B
186                     if (B(1) = '0') then
187                         partial(numbits downto 1) <=
188                             A and std_logic_vector(to_unsigned(0, numbits));
189                     else
190                         partial(numbits downto 1) <=
191                             A and std_logic_vector(to_unsigned(-1, numbits));
192                     end if;
193                     partial(0) <= '0';
194                     -- Only stay in setup state for one clock
195                     currentState <= adder;
196                     -- Initial values
197                     bitsAdded <= 0;
198                     bitsMultiplied <= 0;
199                 ---------------------------
200                 -- adder state
201                 ---------------------------
202                 when adder =>
203                     DONE <= '0';
204                     -- Maintaining Q and Y
205                     Q <= Q;
206                     Y <= Y;
207                     -- Rotate carry and sum for next full adder iteration
208                     carryBits((numbits-1) downto 0) <= carryBits(numbits downto 1);
209                     sumBits((numbits-1) downto 0) <= sumBits(numbits downto 1);
210                           partial((numbits-1) downto 0) <= partial(numbits downto 1);
```

```
211                     -- Cout of full adder of partial products (one of which is sum).  Cin for
212                     -- the full adder comes from the carryBits vector.  This will be loaded
213                     -- into the carryBits vector
214                     carryBits(numbits) <= (carryBits(0) and (sumBits(0) or partial(0))) or
215                                           (sumBits(0) and partial(0));
216                     -- Sum of full adder of partial products (one of which is sum).  Cin for
217                     -- the full adder comes from the carryBits vector.  This will be loaded
218                     -- into the sumBits vector
219                     sumBits(numbits) <= carryBits(0) xor sumBits(0) xor partial(0);
220                           partial(numbits) <= '0';
221                     -- Maintain value for next round of multiply state
222                     bitsMultiplied <= bitsMultiplied;
223                     -- Stay in adder state for numbits iterations before moving to multiply
224                     if (bitsAdded = numbits) then
225                         currentState <= multiply;
226                         bitsAdded <= 0;
227                     else
228                         currentState <= adder;
229                         bitsAdded <= bitsAdded + 1;
230                     end if;
231                 ---------------------------
232                 -- multiply state
233                 ---------------------------
234                 when multiply =>
235                     DONE <= '0';
236                     -- Rotate product for next multiply state
237                     Q((2*numbits-2) downto 0) <= Q((2*numbits-1) downto 1);
238                     -- Load low bit of sum into product
239                     Q(2*numbits-1) <= sumBits(0);
240                     -- Shift sum bits for next adder
241                     sumBits((numbits-1) downto 0) <= sumBits(numbits downto 1);
242                     sumBits(numbits) <= '0';
243                     -- Load next partial prodcut
244                     if (Y(0) = '0') then
245                         partial(numbits downto 1) <=
246                             A and std_logic_vector(to_unsigned(0, numbits));
247                     else
248                         partial(numbits downto 1) <=
249                             A and std_logic_vector(to_unsigned(-1, numbits));
250                     end if;
251                     partial(0) <= '0';
252                     -- Rotate B input for next partial product
```

```
253                     Y((numbits-2) downto 0) <= Y((numbits-1) downto 1);
254                     Y(numbits-1) <= '0';
255                     -- Initial state of bitsAdded before moving back to adder state
256                     bitsAdded <= 0;
257                     -- Stay in multiply for one clock.  If done multiplication, move back
258                     -- to idle.  Otherwise, go for next round of adder
259                     if (bitsMultiplied = 2*numbits - 1) then
260                         currentState <= idle;
261                         bitsMultiplied <= 0;
262                     else
263                         currentState <= adder;
264                         bitsMultiplied <= bitsMultiplied + 1;
265                     end if;
266             end case;
267         end if;
268     end process;
269
270 end DataFlow;
271
```

```
 1  ----------------------------------------------------------------------------------------------
 2  --
 3  --  Test Bench for BitSerialMultiplier
 4  --
 5  --  This is a test bench for the BitSerialMultiplier entity.  The test bench tests
 6  --  the multiplication of 0 in both the multiplicand and the multiplier.  It also multipleis
 7  --  255 by 255 (this assumes the number of bits is set to 8), which shows that there is no
 8  --  overflow.  Finally, it multiples 15 by 15 to show that leftmore zeroes still appear in
 9  --  the output.
10  --
11  --
12  --  Revision History:
13  --      4/11/00  Automated/Active-VHDL    Initial revision.
14  --      4/11/00  Glen George              Modified to add documentation and
15  --                                           more extensive testing.
16  --      4/12/00  Glen George              Added more complete testing.
17  --      4/16/02  Glen George              Updated comments.
18  --      4/18/04  Glen George              Updated comments and formatting.
19  --     11/21/05  Glen George              Updated comments and formatting.
20  --     12/08/16  Tim Menninger            Copied from RLLEncTB and adapted to test bitserme
21  --
22  ----------------------------------------------------------------------------------------------
23
24
25  library ieee;
26  use ieee.std_logic_1164.all;
27  use ieee.numeric_std.all;
28
29
30  entity bitserialmultiplier_tb is
31      generic (
32          numTests : integer := 5,
33          numBits  : integer := 8
34      );
35  end bitserialmultiplier_tb;
36
37  architecture TB_ARCHITECTURE of bitserialmultiplier_tb is
38      -- Component declaration of the tested unit
39      component BitSerialMultiplier
40          port(
41              A      :  in      std_logic_vector((numbits - 1) downto 0);     -- multiplicand
42              B      :  in      std_logic_vector((numbits - 1) downto 0);     -- multiplier
```

```
43              START  :  in      std_logic;                                    -- start calculation
44              CLK    :  in      std_logic;                                    -- clock
45              Q      :  buffer  std_logic_vector((2 * numbits - 1) downto 0); -- product
46              DONE   :  out     std_logic
47          );
48      end component;
49
50      -- Stimulus signals - signals mapped to the input and inout ports of tested entity
51      signal  A      :  std_logic_vector((numbits - 1) downto 0);     -- multiplicand
52      signal  B      :  std_logic_vector((numbits - 1) downto 0);     -- multiplier
53      signal  START  :  std_logic;                                    -- start calculation
54      signal  CLK    :  std_logic;                                    -- clock
55
56      -- Observed signals - signals mapped to the output ports of tested entity
57      signal  Q      :  std_logic_vector((2 * numbits - 1) downto 0); -- product
58      signal  DONE   :  std_logic;
59
60      --Signal used to stop clock signal generators
61      signal  END_SIM     :  BOOLEAN := FALSE;
62
63
64  begin
65
66      -- Unit Under Test port map
67      UUT : BitSerialMultiplier
68          port map(
69              A      => A,
70              B      => B,
71              START  => START,
72              CLK    => CLK,
73              Q      => Q,
74              DONE   => DONE
75          );
76
77
78      -- now generate the stimulus and test the design
79      process
80
81          -- some useful variables
82          variable  i  :  integer;        -- general loop index
83
84          -- inputs for A (numBits bits each)
```

```
 85          signal TestVectorA : std_logic_vector(numTests*numBits - 1 downto 0)
 86              :=  "00000000" &
 87                  "00000000" &
 88                  "10101010" &
 89                  "11111111" &
 90                  "00001111";
 91          -- inputs for B (numBits bits each)
 92          signal TestVectorB : std_logic_vector(numTests*numBits - 1 downto 0)
 93              :=  "00000000" &
 94                  "10101010" &
 95                  "00000000" &
 96                  "11111111" &
 97                  "00001111";
 98          -- outputs for each Q = A * B (2*numBits bits each)
 99          signal TestVectorQ : std_logic_vector(numTests*2*numBits - 1 downto 0)
100              :=  "0000000000000000" &
101                  "0000000000000000" &
102                  "0000000000000000" &
103                  "1111111000000001" &
104                  "0000000011100001";
105
106      begin  -- of stimulus process
107
108          -- initially input is X and mutliplier shouldn't start
109          START  <= '0';
110          A      <= 'X';
111          B      <= 'X';
112
113          -- run for a few clocks
114          wait for 100 ns;
115
116          for  i  in  0  to  (numTests-1)  loop
117              -- Set A and B inputs (multiplicand and multiplier)
118              A <= TestVectorA((numTests-i)*numBits-1 downto (numTests-i-1)*numBits);
119              B <= TestVectorB((numTests-i)*numBits-1 downto (numTests-i-1)*numBits);
120
121              -- run for a few clocks before starting
122              wait for 100 ns;
123              START <= '1';
124
125              -- After a few clocks, clear START
126              wait for 100 ns;
```

```
127             START <= '0';
128 
129             -- let the multiplication occur (occurs in ~2n^2.  Give twice that many clocks)
130             wait for 4*numBits*numBits*20 ns;
131 
132             -- check the output (from old input value)
133             assert (std_match(DONE, '1');
134                     report  "DONE Output Test Failure"
135                     severity  ERROR;
136             assert (std_match(Q,
137                 TestVectorQ((numTests-i)*2*numBits-1 downto (numTests-i-1)*2*numBits)));
138                     report  "Product Q Output Test Failure"
139                     severity  ERROR;
140 
141         end loop;
142 
143         END_SIM <= TRUE;        -- end of stimulus events
144         wait;                   -- wait for simulation to end
145 
146     end process; -- end of stimulus process
147 
148 
149     CLOCK_CLK : process
150 
151     begin
152 
153         -- this process generates a 20 ns period, 50% duty cycle clock
154 
155         -- only generate clock if still simulating
156 
157         if END_SIM = FALSE then
158             CLK <= '0';
159             wait for 10 ns;
160         else
161             wait;
162         end if;
163 
164         if END_SIM = FALSE then
165             CLK <= '1';
166             wait for 10 ns;
167         else
168             wait;
```

```
169             end if;
170
171         end process;
172
173
174     end TB_ARCHITECTURE;
175
```

```
 1  -- bring in the necessary packages
 2  library  ieee;
 3  use  ieee.std_logic_1164.all;
 4
 5  ----------------------------------------------------------------------------------------
 6  --
 7  --  RLLEncoder
 8  --
 9  --  This is an implementation of an RLL(2, 7) encoder.  It takes a constant input stream of
10  --  bits and outputs a constant stream of bits at twice the frequence of the RLL encoding
11  --  of the input.  Because the RLL output requires a two-bit lookahead, but we can't see
12  --  into the future (as of December 2017), the output lags the input by two bits.  The output
13  --  encoding follows this structure:
14  --              Input Stream            Output Stream
15  --                    10                    0100
16  --                    11                    1000
17  --                   000                   000100
18  --                   010                   100100
19  --                   011                   001000
20  --                  0010                  00100100
21  --                  0011                  00001000
22  --
23  --  Inputs:
24  --      DataIn  - Input stream of bits.  These change on CLK and are valid for two CLKs
25  --      Reset   - Active low reset
26  --      CLK     - Global clock
27  --
28  --  Outputs:
29  --      RLLOut  - Output stream of RLL encoded DataIn, lagging by two bits.  This is output
30  --                at twice the frequency of DataIn (frequency of CLK)
31  --
32  --  Revision History:
33  --      7 Dec 16  Tim Menninger     Initial revision
34  --
35  ----------------------------------------------------------------------------------------
36
37  --
38  -- entity RLLEncoder
39  --
40  -- Defines the inputs and outputs for the RLL bit stream encoder
41  --
42  entity RLLEncoder is
```

```
43      port (
44          DataIn  :  in  std_logic;   -- Input stream of data
45          Reset   :  in  std_logic;   -- Active low reset
46          CLK     :  in  std_logic;   -- Global CLK
47          RLLOut  :  out std_logic    -- Output stream of RLL encoded input
48      );
49  end RLLEncoder;
50
51  --
52  -- encoder architecture
53  --
54  -- Defines the dataflow of the RLL encoding of the data input.  To achieve this, it uses a
55  -- state machine that transitions on certain input bit sequences, and outputs based on its
56  -- known relevant history.
57  --
58  --  Inputs:
59  --      DataIn  - Input stream of bits.  These change on CLK and are valid for two CLKs
60  --      Reset   - Active low reset
61  --      CLK     - Global clock
62  --
63  --  Outputs:
64  --      RLLOut  - Output stream of RLL encoded DataIn, lagging by two bits.  This is output
65  --                at twice the frequency of DataIn (frequency of CLk)
66  --
67  architecture encoder of RLLEncoder is
68
69      --
70      -- Types:
71      --
72      -- RLLState: State machien for RLL encoding of input data bit stream
73      --      states:
74      --          RLL* - Each state represents a particular known history of inputs, where
75      --                 the star here would be replaced by the relevant recent history.  In
76      --                 the case of RLL, we know nothing.  This should only be the active
77      --                 state for one clock immediately after reset.
78      --
79      type RLLState is (
80          RLL,
81          RLL0,
82          RLL1,
83          RLL00,
84          RLL01,
```

```
85          RLL10,
86          RLL11,
87          RLL000,
88          RLL001,
89          RLL010,
90          RLL011,
91          RLL0010,
92          RLL0011
93      );
94
95      --
96      -- Signals:
97      --
98      -- RLLMachine (RLLState) - Keeps track of the current state in the state machine.
99      -- secondClk (std_logic) - Each input bit corresponds to two output bits.  This flag
100     --      alternates every clock, keeping us in each state for two clocks.  States only
101     --      change on the second clock, at which point this would be high.
102     --
103     signal RLLMachine : RLLState;
104     signal secondClk  : std_logic;
105
106 begin
107     --
108     -- process
109     --
110     -- Handles the state machine and RLLOut encoded output.  The state machine only changes
111     -- states every other input CLK, which it refers to secondClk to know about.  Each state
112     -- is aware of as many previous bits as it needs to know in order to output the next
113     -- two RLL-encoded bits.  Input and corresponding output sequences can be viewed in
114     -- file header.
115     --
116     -- Inputs:  CLK - Global CLK governing RLLOut frequency
117     --          Reset - Active low reset
118     --          DataIn - Input data stream, which changes every other CLK
119     --
120     -- Outputs: RLLOut - Sets the RLLOut output based on state and input.
121     --
122     process (CLK, Reset, DataIn) is
123     begin
124         if (Reset = '0') then
125             RLLMachine <= RLL;
126             RLLOut <= '0';
```

```
127             secondClk <= '0';
128         elsif (rising_edge(CLK)) then
129             secondClk <= not secondClk;
130             -- State machine
131             case RLLMachine is
132                 ---------------------------
133                 -- RLL state
134                 ---------------------------
135                 when RLL =>
136                     -- Output here doesn't matter
137                     RLLOut <= '0';
138                     -- Idle state, should only be here when next bit is first in sequence
139                     if (secondClk = '1') then
140                         if (DataIn = '0') then
141                             RLLMachine <= RLL0;
142                         else -- (DataIn = '1')
143                             RLLMachine <= RLL1;
144                         end if;
145                     end if;
146                 ---------------------------
147                 -- RLL0 state
148                 ---------------------------
149                 when RLL0 =>
150                     -- Last two bits of any encoding is always 00
151                     RLLOut <= '0';
152                     -- Move states on second clock
153                     if (secondClk = '1') then
154                         -- Move states
155                         if (DataIn = '0') then
156                             RLLMachine <= RLL00;
157                         else -- (DataIn = '1')
158                             RLLMachine <= RLL01;
159                         end if;
160                     end if;
161                 ---------------------------
162                 -- RLL1 state
163                 ---------------------------
164                 when RLL1 =>
165                     -- Last two bits of any encoding is always 00
166                     RLLOut <= '0';
167                     -- Move states on second clock
168                     if (secondClk = '1') then
```

```
169                         -- Move states
170                         if (DataIn = '0') then
171                             RLLMachine <= RLL10;
172                         else -- (DataIn = '1')
173                             RLLMachine <= RLL11;
174                         end if;
175                     end if;
176                 ---------------------------
177                 -- RLL00 state
178                 ---------------------------
179                 when RLL00 =>
180                     -- First and second bits of all 00* states is 00
181                     RLLOut <= '0';
182                     -- Change state on second clock
183                     if (secondClk = '1') then
184                         if (DataIn = '0') then
185                             RLLMachine <= RLL000;
186                         else -- (DataIn = '1')
187                             RLLMachine <= RLL001;
188                         end if;
189                     end if;
190 
191                 ---------------------------
192                 -- RLL01 state
193                 ---------------------------
194                 when RLL01 =>
195                     if (secondClk = '0') then
196                         -- Output of first clock in 01* sequence always inverts input
197                         RLLOut <= not DataIn;
198                     else -- (secondClk = '1')
199                         -- Second bit in 01* encoding is always a 0
200                         RLLOut <= '0';
201                         -- Move states
202                         if (DataIn = '0') then
203                             RLLMachine <= RLL010;
204                         else -- (DataIn = '1')
205                             RLLMachine <= RLL011;
206                         end if;
207                     end if;
208                 ---------------------------
209                 -- RLL10 state
210                 ---------------------------
```

```
211                 when RLL10 =>
212                     -- First and second bits of 10 always 01
213                     RLLOut <= secondClk;
214                     -- Move states on second clock
215                     if (secondClk = '1') then
216                         -- Move states
217                         if (DataIn = '0') then
218                             RLLMachine <= RLL0;
219                         else -- (DataIn = '1')
220                             RLLMachine <= RLL1;
221                         end if;
222                     end if;
223                 ---------------------------
224                 -- RLL11 state
225                 ---------------------------
226                 when RLL11 =>
227                     -- First and second bits of 11 always 10
228                     RLLOut <= not secondClk;
229                     -- Move states on second clock
230                     if (secondClk = '1') then
231                         -- Move states
232                         if (DataIn = '0') then
233                             RLLMachine <= RLL0;
234                         else -- (DataIn = '1')
235                             RLLMachine <= RLL1;
236                         end if;
237                     end if;
238                 ---------------------------
239                 -- RLL000 state
240                 ---------------------------
241                 when RLL000 =>
242                     -- Third and fourth bits of 000 always 01
243                     RLLOut <= secondClk;
244                     -- Move states on second clock
245                     if (secondClk = '1') then
246                         -- Move states
247                         if (DataIn = '0') then
248                             RLLMachine <= RLL0;
249                         else -- (DataIn = '1')
250                             RLLMachine <= RLL1;
251                         end if;
252                     end if;
```

```
253                 --------------------------
254                 -- RLL001 state
255                 --------------------------
256                 when RLL001 =>
257                     -- On first clock, output inverts input (third bit in encoding)
258                     if (secondClk = '0') then
259                         RLLOut <= not DataIn;
260                     else -- (secondClk = '1')
261                         -- Second clock of 001* is always 0 (fourth bit in encoding)
262                         RLLOut <= '0';
263                         -- Move states
264                         if (DataIn = '0') then
265                             RLLMachine <= RLL0010;
266                         else -- (DataIn = '1')
267                             RLLMachine <= RLL0011;
268                         end if;
269                     end if;
270                 --------------------------
271                 -- RLL010 state
272                 --------------------------
273                 when RLL010 =>
274                     -- Third and fourth bits of 010 is always 01
275                     RLLOut <= secondClk;
276                     -- Move states on second clock
277                     if (secondClk = '1') then
278                         -- Move states
279                         if (DataIn = '0') then
280                             RLLMachine <= RLL0;
281                         else -- (DataIn = '1')
282                             RLLMachine <= RLL1;
283                         end if;
284                     end if;
285                 --------------------------
286                 -- RLL011 state
287                 --------------------------
288                 when RLL011 =>
289                     -- Third and fourth bits of 010 is always 10
290                     RLLOut <= not secondClk;
291                     -- Move states on second clock
292                     if (secondClk = '1') then
293                         -- Move states
294                         if (DataIn = '0') then
```

```
295                             RLLMachine <= RLL0;
296                         else -- (DataIn = '1')
297                             RLLMachine <= RLL1;
298                         end if;
299                     end if;
300                 ---------------------------
301                 -- RLL0010 state
302                 ---------------------------
303                 when RLL0010 =>
304                     -- Fifth and sixth bits of 0010 always 01
305                     RLLOut <= secondClk;
306                     -- Move states on second clock
307                     if (secondClk = '1') then
308                         -- Move states
309                         if (DataIn = '0') then
310                             RLLMachine <= RLL0;
311                         else -- (DataIn = '1')
312                             RLLMachine <= RLL1;
313                         end if;
314                     end if;
315                 ---------------------------
316                 -- RLL0011 state
317                 ---------------------------
318                 when RLL0011 =>
319                     -- Fifth and sixth bits of 0011 always 10
320                     RLLOut <= not secondClk;
321                     -- Move states on second clock
322                     if (secondClk = '1') then
323                         -- Move states
324                         if (DataIn = '0') then
325                             RLLMachine <= RLL0;
326                         else -- (DataIn = '1')
327                             RLLMachine <= RLL1;
328                         end if;
329                     end if;
330             end case;
331         end if;
332     end process;
333
334 end encoder;
335
```