Real-time Hand Tracking with Neural Nets on an FPGA

Brian Kubisiak <bkubisia@caltech.edu>
Quinn Osha <qosha@caltech.edu>

1 Functional Specification

For our project, we will be designing and implementing a system for tracking a hand in real time from a camera input. The system will be implemented using a Digilent Nexys 4 FPGA development board. It will take an input from a camera connected to the USB port, analyze the data on the Artix 7 FPGA, and output each from over the VGA port with a cursor overlay. The cursor overlay will indicate where in the frame the hand is located. While the neural net is computing the hand position, the frame will temporarily be stored in the board's memory—the FPGA is too small to process an entire frame at once.

1.1 Operation

The FPGA design will consist of three distinct layers:

Input Layer For communicating with the USB camera, writing the frame to memory, and converting the frame into a format that the neural net can use.

Neural Network For analyzing the camera data to determine where the hand is in the frame.

Output Layer For reading the frame back from memory, overlaying a cursor on the hand position, and generating control signals to output the frame through a VGA port.

A top-level block diagram showing the layers and their connections is shown in figure (1). The Rx and Tx inputs come from the USB-UART bridge. The Data, Address, Wr, and Rd signals go to the memory on the Digilent board. Note that the Data and Address buses must be muxed before connecting to the pins. The HSYNC, VSYNC, and Data outputs will go to the VGA port on the board.

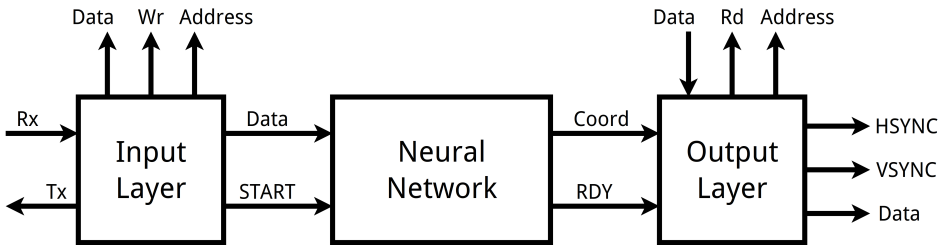

Figure 1: Top-level block diagram for the system.

1.1.1 Input Layer

The input layer is responsible for controlling the USB port, subsampling the frame data, shifting the data into the neural net, and writing each frame into memory. This process is shown in figure (2).

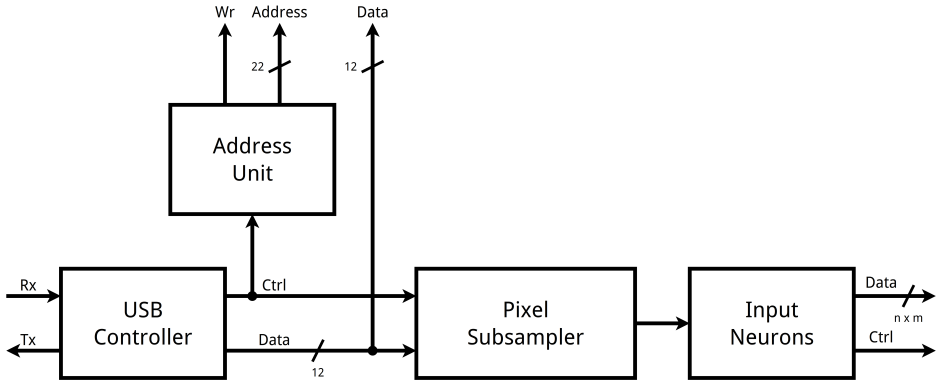

Figure 2: Block diagram for the input layer.

The USB controller interfaces with the FT2232 USB-UART bridge on the Nexys 4 board through the Tx and Rx signals. The Tx signal is needed in order to give commands to the camera over the USB port. The Rx line is responsible for reading in the data.

Once the data is read in through the UART, it is parsed by a finite state machine. This FSM will read the USB packets from the camera to extract the pixel and frame data. It will condense each 24-bit pixel into 12 bits (the size needed for the VGA) and output this data along with control signals for determining the frame timing.

The address unit will take in the control signals from the USB controller and use them to generate the proper addresses for the data. In order to improve performance, the data will be written to memory in a format suitable for VGA controller in the output layer. This address unit will take in the frame and line data to ensure that the pixels are written to the proper addresses.

Figure out the memory layout that we need.

Due to size limitations in the FPGA, the neural net cannot process the entire frame. Pixels will be averaged togther in order to compress the data without losing too much important information.

The input neurons will take the subsampled data on a parallel bus and shift the data through the input layer until the entire subsampled frame is loaded into the neurons. These neurons will then output all $n \times m$ bits of frame data to the neural net in parallel for processing. Control signals will tell the neural net to start processing the frame.

Figure out how we want to subsample.

1.1.2 Neural Network

The neural network will take in the frame data and process it to determine the coordinates of the hand in the frame. We will be using a multi-layer perceptron (MLP) network to implement our design. The weights for each neuron will be learned off chip, freeing up a lot of space for more neurons.

There will be an input neuron for each pixel in the subsampled frame. The input neurons will be unique in that they will shift the data through the entire layer in order to process the entire frame in parallel.

We will then have a single layer in the middle for the data processing. Each neuron in this layer will have a single MAC unit and a sigmoid function. Since the network will be trained off-chip, the weights will be pre-programmed into the design. This will significantly reduce the size of the neurons.

The output layer will consist of only two neurons: one for computing the x-coordinate of the hand, and another for computing the y-coordinate. The output of the neural net will then just be a coordinate identifying where the hand is in the frame.

1.1.3 Output Layer

The output layer is responsible for converting the output of the neural net into a group of pixel addresses, fetching the frame from memory, clearing the pixels corresponding to the hand location, and outputting the frame using the VGA protocol. A block diagram for this process is shown in figure (3).

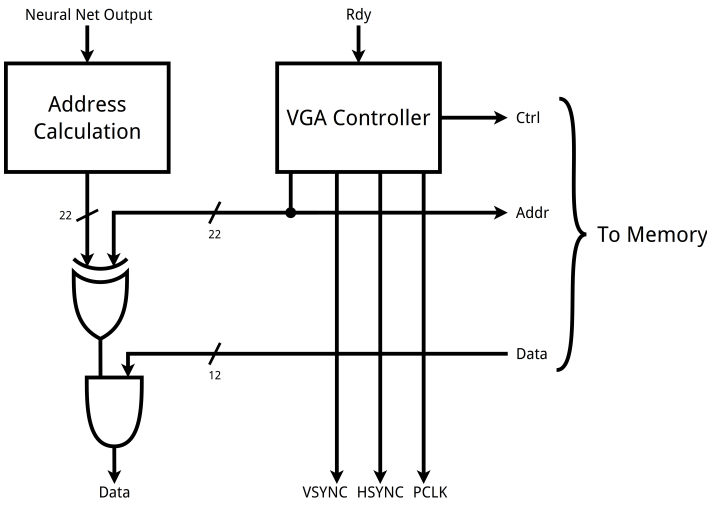

Figure 3: Block diagram for the output layer.

The address calculation takes in the coordinates for the hand in the frame (from the neural net) and converts them into a group of addresses. These addresses correspond to the pixels in memory that will be blanked out for the cursor overlay. These addresses are compared against the addresses output to the memory, clearing the data output when they match.

The VGA controller is a finite state machine that begins running once it receives the Rdy signal from the neural net. The controller will read the frame from memory, one pixel at a time. It will also generate the proper line sync, frame sync, and pixel clock signals to the VGA output. The data for the VGA output will come directly from the data bus.

1.2 Algorithms

We will be using a neural network to identify where the hand is located in each frame. The neural net will be organized as a multi-layer perceptron network. The parameters for the neural network will be learned in advance on the computer, then hard-coded into the FPGA neural net.

1.3 Inputs

The input to this system will be video input through a USB controller. Each frame that is input to the USB controller will be stored in an external memory before being processed by the neural net.

1.4 Outputs

The output from this system will be through a VGA controller. The VGA output will be taken from an output frame buffer. The output frame will be generated from the input frame with a cursor overlaid on the position of the hand (identified by the neural net).

2 Schedule

Date	Tasks
4/7–4/13	Research current implementations of neural nets on FPGAs. Decide on video input and output devices (size and pixel depth). Write code for USB controller.
4/14–4/20	Write code for input data buffer and input data handling. Design—in a high-level-language—the desired neural net.
4/21–4/27	Test and configure the neural net on the computer. Write code for output data buffer and handling.
4/28–5/4	Begin coding the neural net on the FPGA. Write code for output display handling using VGA.
5/5–5/11	Finish coding the neural net.
5/12–5/18	Write code for the cursor video output. That is, given x and y coordinates, print the cursor at that point.
5/19–5/25	Write the top-level entity for the system, combining all the blocks. Train (unsupervised) the neural net.
5/26–6/1	Finish training the neural net. Fix any other bugs/issues.
6/2–6/5	Finish documentation and final touches.

3 Demonstration

This system will be demonstrated by attaching a video camera to the USB input and a monitor to the VGA output. Then, the system should detect a hand in the input frame and output the

frame to the monitor with a cursor overlaid on the position of the hand.

4 Source Code Listing

Below is the VHDL source for our neural net and input/output layers. For a high-level description of the design, see the functional specification.

4.1 UART Receiver

```
1  --
2  -- uartrx.vhd
3  --
4  -- Universal asynchronous receiver/transmitter---receiver design.
5  --
6  -- This file contains the design for the receiver on a UART. The UART will take
7  -- in data serially on the RX line, starting with a single low start bit. This
8  -- implementation uses N-bit characters, 1 stop bit, and no parity bit. The baud
9  -- rate is the same as the input clock frequency. The design is parameterized on
10 -- the number of bits in the character.
11 --
12 -- Once a full character has been received over the serial line, the 'rdy'
13 -- output will be pulsed low for one clock to indicate that data is ready. This
14 -- data will remain latched until the next pulse on the 'rdy' signal.
15 --
16 -- If the stop bit is '0' instead of '1', then the 'err' signal will be pulsed
17 -- low instead of 'rdy'. Even if an error occurred, the received data will still
18 -- be output.
19 --
20 -- The data is loaded as if we received the MSB first. That is, each new bit is
21 -- shifted into the LSB of the shift register.
22 --
23 -- Revision History:
24 --      17 Apr 2015     Brian Kubisiak      Initial revision.
25 --      21 Apr 2015     Brian Kubisiak      Changed to MSB-first.
26 --      22 Apr 2015     Brian Kubisiak      Updated documentation.
27 --
28
29 library ieee;
30 use     ieee.std_logic_1164.all;
31
32 --
33 -- uartrx
34 --
35 --  Parameters:
36 --      N (integer)             Number of bits per character.
37 --
38 --  Inputs:
39 --      reset (std_logic)       Active-low line to reset the UART receiver.
40 --      clk (std_logic)         Baud rate clock. The UART will sample the input
41 --                              signal on every rising edge of the clock.
42 --      rx (std_logic)          Receive line carrying the serial data.
43 --
44 --  Outputs:
45 --      data (std_logic_vector) Last character received over the UART.
46 --      rdy (std_logic)         Active-low line indicating when the data is
47 --                              ready. Once a character is received, this will
48 --                              pulse low for one clock.
49 --      err (std_logic)         Active-low signal indicating an error occurred.
```

```
50 --                            Currently, this means that the stop bit was not
51 --                            high.
52 --
53 entity uartrx is
54     generic ( N : integer := 8 );
55     port
56     (
57         reset   : in  std_logic;                        -- Reset the UART.
58         clk     : in  std_logic;                        -- Baud clock.
59         rx      : in  std_logic;                        -- Serial data input.
60         data    : out std_logic_vector(N-1 downto 0);   -- Parallel data output.
61         rdy     : out std_logic;                        -- New data ready.
62         err     : out std_logic                         -- Error occurred.
63     );
64 end entity;
65
66
67 architecture procedural of uartrx is
68
69     -- Shift register for loading new data from the serial line.
70     signal shftrx   : std_logic_vector(N-1 downto 0);
71
72     -- Buffer for outputting the received data.
73     signal outbuf   : std_logic_vector(N-1 downto 0);
74
75     -- Counter for indicating the number of bits received. There is 1 start bit,
76     -- N data bits, and 1 stop bit for a total of N+2 bits.
77     signal state    : integer range 0 to N+1;
78
79 begin
80
81     -- Always output the buffered data.
82     data <= outbuf;
83
84     --
85     -- LoadNextBit
86     --
87     -- On each clock edge, this process will shift a new bit into the shift
88     -- register. Once the character is full, it will pulse the 'rdy' signal low
89     -- and output the data. It will then wait for the next start bit to begin
90     -- receiving again.
91     --
92     -- If the stop bit is found to be '0' instead of '1', then the 'err' signal
93     -- will be pulsed low instead of 'rdy'.
94     --
95     LoadNextBit: process (clk)
96     begin
97         if rising_edge(clk) then
98
99             -- By default, data is not ready and no error has occurred.
100            rdy <= '1';
101            err <= '1';
102
103            -- By default, latch the output buffer and shift register.
104            outbuf <= outbuf;
105            shftrx <= shftrx;
106
107            -- First, check for reset conditions.
108            if (reset = '0') then
109
110                -- When resetting, we set the counter back to zero to wait for
111                -- the next start bit.
112                state <= 0;
113
114            -- If no bits loaded, wait for start bit.
```

```
115             elsif (state = 0) then
116
117                 if (rx = '0') then
118                     -- Start bit received, begin shifting in data.
119                     state <= 1;
120                 else
121                     -- Otherwise, continue waiting.
122                     state <= 0;
123                 end if;
124
125             -- All data has been received.
126             elsif (state = N+1) then
127
128                 -- Reset counter regardless of data validity.
129                 state   <= 0;
130                 -- Always output the shifted data, even if it is found to be
131                 -- invalid.
132                 outbuf  <= shftrx;
133
134                 -- Check for a valid stop bit.
135                 if (rx = '1') then
136                     -- Stop bit is valid; indicate data is ready.
137                     rdy <= '0';
138                 else
139                     -- Stop bit invalid; indicate an error occurred.
140                     err <= '0';
141                 end if;
142
143             -- Still just shifting in the data.
144             else
145
146                 -- Shift the new data bit into the LSB of the shift register.
147                 shftrx <= shftrx(N-2 downto 0) & rx;
148                 -- Go to next state.
149                 state  <= state + 1;
150
151             end if;
152
153         end if;
154     end process;
155
156 end architecture procedural;
```

4.2 UART Transmitter

```
1  --
2  -- uarttx.vhd
3  --
4  -- Universal asynchronous receiver/transmitter---transmitter design.
5  --
6  -- This file contains the design for the transmitter on a UART. The UART will
7  -- take in data in parallel on the data bus. It will then output the character
8  -- in serial on the TX line. This implementation uses N-bit characters, 1 stop
9  -- bit, and no parity bit. The baud rate is the same as the input clock
10 -- frequency. The design is parameterized on the number of bits in each
11 -- character.
12 --
13 -- To transmit data, output the desired character to the 'data' input and send
14 -- the 'rdy' line low. These signals must be latched until the UART sends the
15 -- 'ack' line low to acknowledge the data. Note that the 'ack' signal will be
16 -- pulsed low for only a single clock.
17 --
```

```
18 -- The data is transmitted MSB first; that is, the data is transmitted from the
19 -- most-significant bit first, down to the least-significant bit last.
20 --
21 -- Revision History:
22 --      22 Apr 2015     Brian Kubisiak      Initial revision.
23 --
24
25 library ieee;
26 use     ieee.std_logic_1164.all;
27
28 --
29 -- uarttx
30 --
31 --  Parameters:
32 --      N (integer)             Number of bits per character.
33 --
34 --  Inputs:
35 --      reset (std_logic)       Active-low line to reset the UART transmitter.
36 --      clk (std_logic)         Baud rate clock. The UART will transmit one bit
37 --                              on every rising edge of the clock.
38 --      data (std_logic_vector) Next character to send over the UART.
39 --      rdy (std_logic)         Active-low line indicating that data is ready to
40 --                              transmit. It should be sent low for at least one
41 --                              baud.
42 --
43 --  Outputs:
44 --      tx (std_logic)          Transmit line for transmitting the serial data.
45 --      ack (std_logic)         Active-low signal acknowledging the ready data,
46 --                              indicating that the new data will now be
47 --                              transmitted.
48 --
49 entity uarttx is
50     generic ( N : integer := 8 );
51     port
52     (
53         reset   : in  std_logic;                        -- Reset the UART.
54         clk     : in  std_logic;                        -- Baud clock.
55         data    : in  std_logic_vector(N-1 downto 0);   -- Parallel data output.
56         rdy     : in  std_logic;                        -- New data ready.
57         ack     : out std_logic;                        -- Ack data ready.
58         tx      : out std_logic                         -- Serial data input.
59     );
60 end entity;
61
62
63 architecture procedural of uarttx is
64
65     -- Register for shifting data out on the transmit line
66     signal shfttx   : std_logic_vector(N-1 downto 0);
67
68     -- Counter for indicating the number of bits transmitted. Tere is 1 start
69     -- bit, N data bits, and 1 stop bit for a total of N+2 bits.
70     signal state    : integer range 0 to N+1;
71
72     -- Latches the rdy signal, resetting whenever the resetrdy signal is active.
73     signal rdylatch : std_logic;
74
75 begin
76
77     --
78     --  DataRdyLatch
79     --
80     --  This process latches the data ready signal at the end of each character
81     --  transmission. The latch will hold for an entire cycle.
82     --
```

```
83      DataRdyLatch: process(clk)
84      begin
85          if rising_edge(clk) then
86
87              -- When reset is active, the latch signal goes inactive.
88              if (reset = '0') then
89                  rdylatch <= '1';
90
91              -- When transmitting the stop bit, check to see if new data is
92              -- ready. If it is ready, latch the ready signal.
93              elsif (state = N+1) then
94                  rdylatch <= rdy;
95
96              -- Else, just keep holding the latch.
97              else
98                  rdylatch <= rdylatch;
99
100             end if;
101
102         end if;
103     end process;
104
105
106     --
107     --  TransmitData
108     --
109     --  Transmit the data one bit at a time over the 'tx' line whenever data is
110     --  valid. After transmission, if new data is available, transfer it to the
111     --  shift register and start the transmission cycle again. When data is
112     --  shifted into the shift register, send an acknowledgement pulse.
113     --
114     TransmitData: process(clk)
115     begin
116         if rising_edge(clk) then
117
118             -- By default, latch the shift register.
119             shfttx <= shfttx;
120             -- Do not acknowledge any transmissions.
121             ack    <= '1';
122
123             -- When resetting or no data available, output stop bit.
124             if (reset = '0' or rdylatch = '1') then
125
126                 -- Output stop bit
127                 tx      <= '1';
128                 -- Hold in this state to receive new data to transmit.
129                 state   <=  N+1;
130
131             -- If we are beginning a transmission, output the start bit
132             elsif (state = 0) then
133
134                 -- Beginning of transmission; output start bit and advance to
135                 -- next state.
136                 tx      <= '0';
137                 state   <=  1;
138
139             -- At the end of a transmission, start new transmission. Note that
140             -- we can only get here if new data is available (rdylatch = '0').
141             elsif (state = N+1) then
142
143                 -- All data has been sent; output a stop bit.
144                 tx      <= '1';
145                 -- Start a new transmission.
146                 state   <=  0;
147                 -- Load the new data into the shift register.
```

```
148                 shfttx  <= data;
149                 -- Acknowledge the new transmission
150                 ack     <= '0';
151
152             -- Otherwise, we are in the process of shifting out the data.
153             else
154
155                 -- Output the next MSB from the shift register.
156                 tx      <= shfttx(N-1);
157                 -- Shift the data.
158                 shfttx  <= shfttx(N-2 downto 0) & 'X';
159                 -- Advance to next state.
160                 state   <= state + 1;
161
162             end if;
163
164         end if;
165     end process;
166
167 end architecture procedural;
```

4.3 UART Receiver Testbench

```
1  --
2  --  uartrx-tb.vhd
3  --
4  --  Test Bench for UART receiver.
5  --
6  --  This is not an exhaustive testbench; it simply clocks data into the UART
7  --  reciever (including one character with an invalid stop bit), and ensures
8  --  that the correct data comes out the other side. It also tests to make sure
9  --  that invalid data will not stop the receiver from running.
10 --
11 --  The testbench assumes that the receiver interprets data as MSB first.
12 --
13 --  Revision History:
14 --      21 Apr 2015     Brian Kubisiak      Initial revision.
15 --      22 Apr 2015     Brian Kubisiak      Updated documentation.
16 --
17
18 library ieee;
19 use     ieee.std_logic_1164.all;
20
21 -- Our testbench entity has no ports; it is completely self-contained.
22 entity uartrx_tb is
23 end uartrx_tb;
24
25
26 architecture TB_ARCHITECTURE of uartrx_tb is
27
28 --
29 -- uartrx
30 --
31 --  This component is the unit-under-test. It is an asynchronous receiver for
32 --  the UART design. It takes in serial data at the baud rate determined by the
33 --  clock, and outputs the data in parallel.
34 --
35 --  Parameters:
36 --      N (integer)             Number of bits per character.
37 --
38 --  Inputs:
39 --      reset (std_logic)       Active-low line to reset the UART receiver.
```

```
40 --      clk (std_logic)         Baud rate clock. The UART will sample the input
41 --                              signal on every rising edge of the clock.
42 --      rx (std_logic)          Receive line carrying the serial data.
43 --
44 --  Outputs:
45 --      data (std_logic_vector) Last character received over the UART.
46 --      rdy (std_logic)         Active-low line indicating when the data is
47 --                              ready. Once a character is received, this will
48 --                              pulse low for one clock.
49 --      err (std_logic)         Active-low signal indicating an error occurred.
50 --                              Currently, this means that the stop bit was not
51 --                              high.
52      component uartrx is
53          generic ( N : integer := 8 );
54          port (
55              reset   : in  std_logic;                        -- Reset the UART
56              clk     : in  std_logic;                        -- Baud clock
57              rx      : in  std_logic;                        -- Serial data in
58              data    : out std_logic_vector(N-1 downto 0);   -- Parallel data out
59              rdy     : out std_logic;                        -- New data ready
60              err     : out std_logic                         -- Error occurred
61          );
62      end component;
63
64      -- Stimulus signals - signals mapped to the input ports of tested entity
65      signal clk      : std_logic;    -- System clock
66      signal reset    : std_logic;    -- Reset receiver, preparing it for new data
67      signal rx       : std_logic;    -- Serial data input to receiver
68
69      -- Outputs - signals that are checked against the expected outputs of the
70      -- test bench.
71      signal data     : std_logic_vector(7 downto 0); -- Data received by UART
72      signal rdy      : std_logic;    -- Data from UART ready for reading
73      signal err      : std_logic;    -- UART read invalid data
74
75      -- Indicates that the simulation has finished
76      signal END_SIM  : boolean := FALSE;
77
78      -- This signal holds the test pattern to shift into the receiver. It has a
79      -- couple of bits at the beginning to allow the UART to reset, followed by
80      -- 2 valid bytes, then an invalid, then another valid.
81      constant data_in: std_logic_vector(0 to 10*4 + 1) :=
82          "11" & "0010101011" & "0101010101" & "0110011010" & "0001100101";
83
84      -- Type for holding the output vectors. We need this because VHDL won't
85      -- declare an array of std_logic_vector without a new type.
86      type OutData is array (0 to 3) of std_logic_vector(7 downto 0);
87
88      -- We should see the following data output from the UART:
89      constant data_out: OutData :=
90          ("01010101", "10101010", "11001101", "00110010");
91
92      -- The 'rdy' signal should exhibit this pattern. For the third pattern, the
93      -- 'err' signal will be pulsed instead of the 'rdy' signal.
94      constant rdy_sig: std_logic_vector(0 to 3) := "0010";
95
96  begin
97
98      -- Declare the unit-under-test and wire up all its inputs to stimulus
99      -- signals and outputs to tested signals.
100     UUT: uartrx
101         generic map ( N => 8 )
102         port map  (
103             -- Map inputs to stimulus signals:
104             clk   => clk,
```

```
105             reset => reset,
106             rx    => rx,
107 
108             -- Map outputs to tested signals:
109             data  => data,
110             rdy   => rdy,
111             err   => err
112         );
113 
114     -- This process will reset the UART for a couple of clocks, then stimulate
115     -- the serial data input with the test signal. Once all the test bits have
116     -- been 'transmitted', then simulation will end.
117     StimulateInputs: process
118     begin
119 
120         -- Reset the UART
121         reset <= '0';
122 
123         -- Set stop bit so UART doesn't begin reading immediately.
124         rx <= '1';
125 
126         -- Wait a couple of clock cycles to the UART to reset.
127         wait until clk = '1';
128         wait until clk = '0';
129         wait until clk = '1';
130         wait until clk = '0';
131 
132         -- Start running the receiver
133         reset <= '1';
134 
135         for i in 0 to 41 loop
136 
137             -- Wait for the next rising edge
138             wait until clk = '1';
139 
140             -- Output the next bit in the stimulus sequence with a short delay
141             -- after the clock.
142             rx <= data_in(i) after 1 ns;
143 
144             -- Hold here for the clock to go low again.
145             wait until clk = '0';
146 
147         end loop;
148 
149         -- Wait a couple more clocks to make sure the data propagates through
150         wait until clk = '1';
151         wait until clk = '0';
152         wait until clk = '1';
153         wait until clk = '0';
154 
155         -- Simulation is over
156         END_SIM <= TRUE;
157         wait;
158 
159     end process;
160 
161     -- This process will wait until the receiver indicates that it has data,
162     -- then check to make sure that the data is correct.
163     TestOutputs: process
164     begin
165 
166         for i in 0 to 3 loop
167 
168             -- Wait until a new byte is received (whether in error or not)
169             wait until (rdy = '0' or err = '0');
```

```
170             -- Check data on the next rising edge of the clock
171             wait until (clk'event and clk = '1');
172 
173             -- Check to make sure that the byte is correct and errors are found
174             -- only at the proper times
175             assert (data = data_out(i) and rdy = rdy_sig(i))
176                 report "Wrong data on output."
177                 severity ERROR;
178 
179             -- Wait for the signal to be reset
180             wait until (rdy = '1' and err = '1');
181 
182         end loop;
183 
184         -- Notify the simulation once all tests have passed.
185         assert (FALSE) report "All outputs tested." severity NOTE;
186         wait;
187 
188     end process;
189 
190     -- This process will generate a clock with a 20 ns period and a 50% duty
191     -- cycle. Once the end of the simulation has been reached (END_SIM = TRUE),
192     -- then the clock will stop oscillating.
193     GenClock: process
194     begin
195 
196         -- this process generates a 20 ns 50% duty cycle clock
197         -- stop the clock when the end of the simulation is reached
198         if END_SIM = FALSE then
199             clk <= '0';
200             wait for 10 ns;
201         else
202             wait;
203         end if;
204 
205         if END_SIM = FALSE then
206             clk <= '1';
207             wait for 10 ns;
208         else
209             wait;
210         end if;
211 
212     end process;
213 
214 end TB_ARCHITECTURE;
```

4.4 UART Testbench

```
1  --
2  --  uart-tb.vhd
3  --
4  --  Test Bench for UART transmitter and receiver.
5  --
6  --  This is not an exhaustive testbench; it simply puts data into the UART
7  --  transmitter, and checks that the same data comes out the other side from the
8  --  receiver.
9  --
10 --  Revision History:
11 --      24 Apr 2015     Brian Kubisiak      Initial revision.
12 --
13 
14 library ieee;
```

```
15 use     ieee.std_logic_1164.all;
16
17 -- Our testbench entity has no ports; it is completely self-contained.
18 entity uart_tb is
19 end uart_tb;
20
21
22 architecture TB_ARCHITECTURE of uart_tb is
23
24 --
25 -- uarttx
26 --
27 --  This component is one of the units-under-test. It is an asynchronous
28 --  transmitter for the UART design. It takes in data on the parallel input
29 --  (along with a signal telling it to start transmitting), and outputs the data
30 --  serially.
31 --
32 --  Parameters:
33 --      N (integer)             Number of bits per character.
34 --
35 --  Inputs:
36 --      reset (std_logic)       Active-low line to reset the UART transmitter.
37 --      clk (std_logic)         Baud rate clock. The UART will transmit one bit
38 --                              on every rising edge of the clock.
39 --      data (std_logic_vector) Next character to send over the UART.
40 --      rdy (std_logic)         Active-low line indicating that data is ready to
41 --                              transmit. It should be sent low for at least one
42 --                              baud.
43 --
44 --  Outputs:
45 --      tx (std_logic)          Transmit line for transmitting the serial data.
46 --      ack (std_logic)         Active-low signal acknowledging the ready data,
47 --                              indicating that the new data will now be
48 --                              transmitted.
49 --
50     component uarttx is
51         generic ( N : integer := 8 );
52         port
53         (
54             reset   : in  std_logic;                        -- Reset the UART
55             clk     : in  std_logic;                        -- Baud clock
56             data    : in  std_logic_vector(N-1 downto 0);   -- Parallel data out
57             rdy     : in  std_logic;                        -- New data ready
58             ack     : out std_logic;                        -- Ack data ready
59             tx      : out std_logic                         -- Serial data input
60         );
61     end component;
62
63 --
64 -- uartrx
65 --
66 --  This component is the unit-under-test. It is an asynchronous receiver for
67 --  the UART design. It takes in serial data at the baud rate determined by the
68 --  clock, and outputs the data in parallel.
69 --
70 --  Parameters:
71 --      N (integer)             Number of bits per character.
72 --
73 --  Inputs:
74 --      reset (std_logic)       Active-low line to reset the UART receiver.
75 --      clk (std_logic)         Baud rate clock. The UART will sample the input
76 --                              signal on every rising edge of the clock.
77 --      rx (std_logic)          Receive line carrying the serial data.
78 --
79 --  Outputs:
```

```
80  --      data (std_logic_vector) Last character received over the UART.
81  --      rdy (std_logic)         Active-low line indicating when the data is
82  --                              ready. Once a character is received, this will
83  --                              pulse low for one clock.
84  --      err (std_logic)         Active-low signal indicating an error occurred.
85  --                              Currently, this means that the stop bit was not
86  --                              high.
87      component uartrx is
88          generic ( N : integer := 8 );
89          port (
90              reset   : in  std_logic;                        -- Reset the UART
91              clk     : in  std_logic;                        -- Baud clock
92              rx      : in  std_logic;                        -- Serial data in
93              data    : out std_logic_vector(N-1 downto 0);   -- Parallel data out
94              rdy     : out std_logic;                        -- New data ready
95              err     : out std_logic                         -- Error occurred
96          );
97      end component;
98
99
100     -- Stimulus signals - signals mapped to the input ports of tested entity
101     signal clk      : std_logic;    -- System clock
102     signal reset    : std_logic;    -- Reset UART
103     signal data_tx  : std_logic_vector(7 downto 0); -- Data to send
104     signal rdy_tx   : std_logic;    -- Data input should be sent
105
106     -- Outputs - signals that are checked against the expected outputs of the
107     -- test bench.
108     signal ack      : std_logic;    -- New data is being transmitted
109     signal data_rx  : std_logic_vector(7 downto 0); -- Received data
110     signal rdy_rx   : std_logic;    -- New data has been received
111     signal err      : std_logic;    -- Error ocurred during transmission
112
113     -- Line for transferring data between the transmitter and receiver.
114     signal transmit : std_logic;
115
116     -- Indicates that the simulation has finished
117     signal END_SIM  : boolean := FALSE;
118
119     -- Type for holding the test vectors. We need this because VHDL doesn't like
120     -- declaring an array of std_logic_vector without a new type.
121     type DataVec is array (0 to 5) of std_logic_vector(7 downto 0);
122
123     -- We should see the following data output from the UART:
124     constant test_vec: DataVec := ("11010111", "11111111", "00000000",
125                                    "10101010", "00001111", "11110000");
126
127 begin
128
129     -- Declare the units-under-test and wire up all its inputs to stimulus
130     -- signals and outputs to tested signals.
131     UUT_rx: uartrx
132         generic map ( N => 8 )
133         port map (
134             -- Map inputs to stimulus signals:
135             clk   => clk,
136             reset => reset,
137
138             -- Line for transferring the data:
139             rx    => transmit,
140
141             -- Map outputs to tested signals:
142             data  => data_rx,
143             rdy   => rdy_rx,
144             err   => err
```

```
145         );
146     UUT_tx: uarttx
147         generic map ( N => 8 )
148         port map (
149             -- Map inputs to stimulus signals:
150             reset => reset,
151             clk   => clk,
152             data  => data_tx,
153             rdy   => rdy_tx,
154 
155             -- Line for transferring the data:
156             tx    => transmit,
157 
158             -- Map outputs to tested signals:
159             ack   => ack
160         );
161 
162     -- Send data into the transmitter. This will test transmitting a burst as
163     -- well as transmitting isolated characters.
164     TxData: process
165     begin
166 
167         -- Reset the UART
168         reset <= '0';
169 
170         -- Start without transmitting data
171         rdy_tx <= '1';
172 
173         -- Wait a clock for it to completely reset
174         wait until clk = '1';
175         wait until clk = '0';
176 
177         -- Transmitter should be transmitting stop bits now, we can stop
178         -- resetting
179         reset <= '1';
180 
181         -- Transmit the first 4 characters in quick succession
182         for i in 0 to 3 loop
183 
184             -- Start transmitting data.
185             data_tx <= test_vec(i);
186             rdy_tx  <= '0';
187 
188             -- Once the transfer is acknowledged, go on to the next vector
189             wait until ack = '0';
190 
191         end loop;
192 
193         -- Done transmitting for a little bit.
194         rdy_tx <= '1';
195 
196         for i in 4 to 5 loop
197 
198             -- Wait for a long time for a break between transmissions
199             wait for 500 ns;
200 
201             -- Start the next transmission.
202             data_tx <= test_vec(i);
203             rdy_tx  <= '0';
204 
205             -- Wait for acknowledgement before continuing
206             wait until ack = '0';
207             rdy_tx <= '1';
208 
209         end loop;
```

```
210
211         -- Finished transmitting all data; just wait here.
212         wait;
213
214     end process;
215
216     -- Check the data on the receiver, making sure that it is the same as the
217     -- transmitted data.
218     RxData: process
219     begin
220
221         for i in 0 to 5 loop
222
223             -- Wait until a new character is received.
224             wait until rdy_rx = '0';
225             -- Check data on the next rising edge of the clock.
226             wait until (clk'event and clk = '1');
227
228             assert (data_rx = test_vec(i))
229                 report "Incorrect data received."
230                 severity ERROR;
231
232             -- Now wait for the ready signal to clear.
233             wait until rdy_rx = '1';
234
235         end loop;
236
237         -- Notify once all tests have passed.
238         assert (FALSE) report "All tests completed." severity NOTE;
239
240         -- Simulation is over once all the outputs have been tested.
241         END_SIM <= TRUE;
242         wait;
243
244     end process;
245
246     -- Check to make sure that the error signal is never asserted.
247     CheckErr: process(err)
248     begin
249
250         -- Make sure that the error signal isn't asserted.
251         assert (err = '1')
252             report "Error found on receiver."
253             severity ERROR;
254
255     end process;
256
257     -- This process will generate a clock with a 20 ns period and a 50% duty
258     -- cycle. Once the end of the simulation has been reached (END_SIM = TRUE),
259     -- then the clock will stop oscillating.
260     GenClock: process
261     begin
262
263         -- this process generates a 20 ns 50% duty cycle clock
264         -- stop the clock when the end of the simulation is reached
265         if END_SIM = FALSE then
266             clk <= '0';
267             wait for 10 ns;
268         else
269             wait;
270         end if;
271
272         if END_SIM = FALSE then
273             clk <= '1';
274             wait for 10 ns;
```

```
275         else
276             wait;
277         end if;
278 
279     end process;
280 
281 end TB_ARCHITECTURE;
```

4.5 USB Video Decoder

```
 1 ----------------------------------------------------------------------------------
 2 -- Company:         EE119c
 3 -- Engineer:        Quinn Osha
 4 --
 5 -- Create Date:     17:04:43 04/17/2015
 6 -- Design Name:     neaural-net-fpga
 7 -- Module Name:     usb_video_decoder - Behavioral
 8 -- Project Name:
 9 -- Target Devices:
10 -- Tool versions:
11 -- Description:     This entity implements a decoder of the USB video class
12 --                  data stream.
13 --
14 -- Dependencies:
15 --
16 -- Revision:
17 -- Revision 0.01 - File Created
18 -- Additional Comments:
19 --
20 ----------------------------------------------------------------------------------
21 library IEEE;
22 use IEEE.STD_LOGIC_1164.ALL;
23 
24 -- Uncomment the following library declaration if using
25 -- arithmetic functions with Signed or Unsigned values
26 --use IEEE.NUMERIC_STD.ALL;
27 
28 -- Uncomment the following library declaration if instantiating
29 -- any Xilinx primitives in this code.
30 --library UNISIM;
31 --use UNISIM.VComponents.all;
32 
33 entity usb_video_decoder is
34      Generic (
35         num_input_bits        : integer; -- number of bits from UART tx/rx
36         num_red_output_bits   : integer; -- number of red bits to keep
37         num_green_output_bits : integer; -- number of green bits to keep
38         num_blue_output_bits  : integer  -- number of blue bits to keep
39     );
40     Port (
41         Clock   : in  STD_LOGIC;    -- system clock
42         Reset   : in  STD_LOGIC;    -- system reset
43         -- data from UART input stream
44         DataIn  : in  STD_LOGIC_VECTOR (num_input_bits - 1 downto 0);
45 
46         Hsync   : out STD_LOGIC;    -- horizontal sync out
47         Vsync   : out STD_LOGIC;    -- vertical sync out
48         Fsync   : out STD_LOGIC;    -- frame sync out
49         DataOut : out  STD_LOGIC_VECTOR (num_red_output_bits -- Output data bus
50                 + num_green_output_bits + num_blue_output_bits - 1 downto 0)
51     );
52 end usb_video_decoder;
```

```
53
54
55  architecture Behavioral of usb_video_decoder is
56
57  constant HEADER_LENGTH    : integer := 32;  -- num of bytes in video header
58  constant INACTIVE_SAMPLES : integer := 160; -- num of inactive videoSamples/line
59  constant ACTIVE_SAMPLES   : integer := 640; -- num of active videoSamples/line
60  constant TOTAL_SAMPLES    : integer := 800; -- num of total videoSamples/line
61  constant INACTIVE_LINES   : integer := 45;  -- num of inactive lines/frame
62  constant ACTIVE_LINES     : integer := 480; -- num of active lines/frame
63  constant TOTAL_LINES      : integer := 525; -- num of total lines/frame
64
65  -- FSM variables
66  type states is (
67      idle,
68      activeSubSlot1,
69      activeSubSlot2,
70      inactiveSubSlot1,
71      inactiveSubSlot2,
72      inactiveLineSlot1,
73      inactiveLineSlot2
74  );
75
76  signal currentState : states; -- The current state of the FSM
77  signal nextState    : states; -- The next state of the FSM
78
79  -- line and column counters
80
81  -- counter to track the current line in the frame
82  signal rowCounter : integer range 0 to TOTAL_SAMPLES;
83  -- counter to track the current column in the line
84  signal colCounter : integer range 0 to TOTAL_LINES;
85
86
87  -- pre-DFF output data bus
88  signal internalDataOut :  STD_LOGIC_VECTOR (num_red_output_bits
89                      + num_green_output_bits + num_blue_output_bits - 1 downto 0);
90  begin
91
92      -- nextState logic
93      FSM: process(currentState)
94      begin
95          -- Current byte is first(low) byte in active videoSubSlot
96          if(currentState = activeSubSlot1) then
97
98              -- always going to get second part of subSlot
99              nextState <= activeSubSlot2;
100             -- output first byte
101             internalDataOut(num_input_bits - 1 downto 0) <= DataIn;
102
103         -- Current byte is second (high) byte in active videoSubSlot
104         elsif(currentState = activeSubSlot2) then
105
106             if(colCounter < ACTIVE_SAMPLES - 1) then    -- Active samples of line
107                 nextState <= activeSubSlot1;
108             else                                        -- Inactive samples of line
109                 nextState <= inactiveSubSlot1;
110             end if;
111
112             -- output second byte
113             internalDataOut(2 * num_input_bits - 1 downto num_input_bits) <= DataIn;
114
115         -- Current byte is first (low) byte in inactive videoSubSlot
116         elsif(currentState = inactiveSubSlot1) then
117
```

```
118             nextState <= inactiveSubSlot2; -- always going to get second part of subSlot
119
120         -- Current byte is second (high) byte in inactive videoSubSlot
121         elsif(currentState = inactiveSubSlot2) then
122
123             if(colCounter < TOTAL_SAMPLES - 1) then   -- still in inactive part of line
124                 nextState <= inactiveSubSlot1;
125             elsif(rowCounter < ACTIVE_LINES - 1) then -- at the end of inactive part of line
126                 nextState <= activeSubSlot1;
127             else
128                 nextState <= inactiveLineSlot1;
129             end if;
130
131         elsif(currentState = inactiveLineSlot1) then
132
133             nextState <= inactiveLineSlot2; -- always going to get second part of lineSlot
134
135         elsif(currentState = inactiveLineSlot2) then
136
137             if(rowCounter < TOTAL_LINES) then   -- still in inactive lines of frame
138                 nextState <= inactiveLineSlot1;
139             else                                -- finished the inactive lines of frame
140                 nextState <= activeSubSlot1;    -- reset to start of next frame
141             end if;
142
143         end if;
144     end process;
145
146
147     process(Clock)
148     begin
149         if(Clock'event and Clock = '1') then
150             if(reset = '0') then
151                 Hsync    <= '0';        -- default to no newline
152                 Vsync       <= '0';     -- default to no newSample
153                 Fsync    <= '0';        -- default to no newFrame
154
155                 -- Update row and column counters
156                 if(colCounter = TOTAL_SAMPLES - 1 and rowCounter = TOTAL_LINES - 1) then
157                     -- End of a frame
158                     rowCounter <= 0;
159                     colCounter <= 0;
160                     Fsync         <= '1';   -- signal a new frame
161                 elsif(colCounter = TOTAL_SAMPLES - 1) then
162                     -- End of a line
163                     rowCounter <= rowCounter + 1;
164                     colCounter <= 0;
165                     Hsync   <= '1'; -- a new Line has been started
166                 else
167                     -- Middle of a line
168                     rowCounter <= rowCounter;
169
170                     if(nextState = activeSubSlot2 or nextState = inactiveSubSlot2
171                             or nextState = inactiveLineSlot2)
172                     then
173                         -- Only have one of the bytes from SubSlot
174                         colCounter <= colCounter;
175                     else
176                         -- finished current SubSlot, move to next in line
177                         colCounter <= colCounter + 1;
178                         Vsync <= '1'; -- a new SubSlot sample is ready
179                     end if;
180                 end if;
181
182                 -- Update the current state of FSM
```

```
183                 currentState <= nextState;
184             else -- reset FSM
185                 rowCounter  <= 0;
186                 colCounter  <= 0;
187                 currentState <= idle;
188             end if;
189         end if;
190 
191     end process;
192 
193     dataDFF: process(Clock)
194     begin
195         if(Clock'event and Clock = '1') then
196             DataOut <= internalDataOut; -- Latch internal data bus and send out
197         end if;
198     end process;
199 
200 end Behavioral;
```