Project Report

TFE-4171 Design of Digital Systems 2

Andreas Varntresk & Anders Nilsen

April 2018

Contents

1. Introduction

Module verification can be a time-consuming process, in some cases requiring more resources and time than module design. The introduction of verification-oriented tools such as SystemVerilog, UVM and formal verification have helped decrease the so called "Verification Gap", the efficiency gap between design and verification.

One of the advantages provided by SystemVerilog is assertions. Assertions are sequence-specific tests which check a specified functionality, such as variables and counters being reset to 0 after the reset has been enabled. The assertions are used to verify the functionality of the circuit provided in this exercise, based on the specifications provided which are to be verified.

This report was written for the project in Design of Digital Systems 2 (TFE4171), which is about testing and verifying the functionality of a HDLC communication protocol. The goal of the project is to give insight with working on larger a projects, adding assertions at module design time, adding assertions to legacy modules and functional coverage. To complete the project, concurrent assertions have been written in the assertion file, and modifications have been done in the testbench to accomodate for the use of different variables in the system.

2. HDLC description

High-level Data Link Control (HDLC) is a data communication protocol providing bit-oriented transfer, and can be used for synchronous or asynchronous transfer, and in the case of this project is used for synchronous. When transmission is idle, only ones are transmitted over the line.

Information is transmitted in frames, with a flag sequence indicating either the start or end of a frame. The flag sequence is defined as 01111110. Because the flag sequence contains 6 consecutive ones, the data transmitted cannot contain more than 5 consecutive ones. To stop this from happening, bit stuffing is used, and as a result, whenever 5 or more ones are being transmitted, a zero bit is inserted after the consecutive ones and whenever 5 or more ones are being received, there should be a zero following the sequence that needs to be removed. If the sixth bit is one and the seventh bit is zero, it is interpreted as a flag sequence and if the sixth bit is one and the seventh is one, it is interpreted as an error.

The structure of one frame is shown in Figure 2.1. In this project address and control

Flag	Address	Control	Information	FCS	Flag
8 bits	8 or more bits	8 bits	Variable length, n * 8 bits	16 bits	8 bits

Figure 2.1.: HDLC-Frame, where $n \in \{0, 126\}$

byte are not used, so they can be any 8 bit value. The information field is a multiple of 8 and can be as long as 126 bytes. The last field is the FCS(Frame Check Sequence) bits, it is 16 bits long and used for error detection.

The error detection bits are calculated with CRC (Cycle Redundancy Check) method. The method uses polynomial division on the frame to calculate the FCS bits on the transmitter, and for the receiver division with the same polynomial on the same frame should result in all zeros. The division on the sender side is performed on the entire length of the frame, including address and control bytes, excluding flag bytes, with zeros inserted in the FCS field. The division on the receiver side is performed on the entire frame plus the FCS field. The polynomial used in this system is $X^{16} + X^{15} + X^2 + 1$.

A figure of the HDLC module with the input and output signals are shown in 2.2. Clock is the input for the clock signal, and reset is, in this case, and active-low signal. To communicate with the module the address interface can be used. This uses the signals

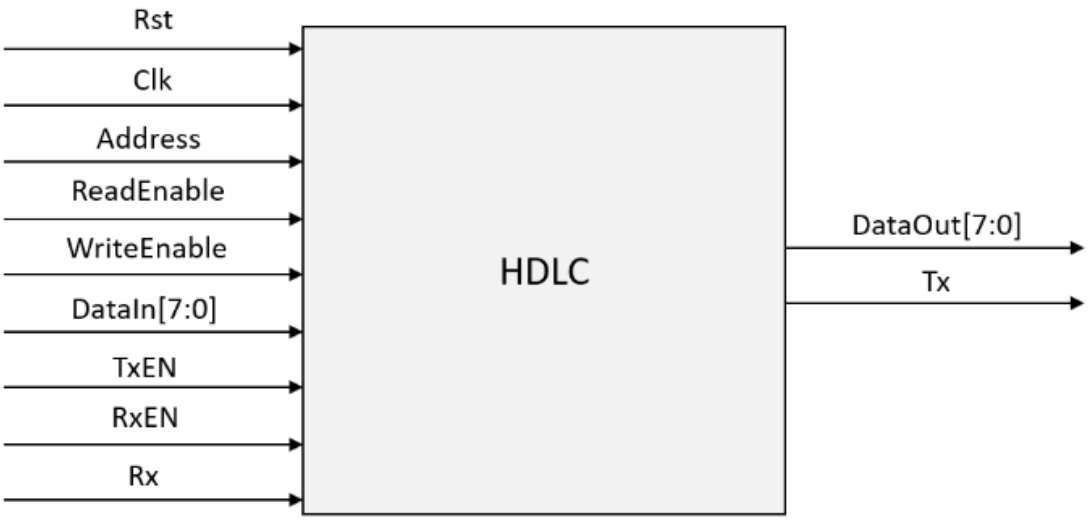

Figure 2.2.: HDLC-Module

Address, ReadEnable, WriteEnable, DataIn and DataOut. This interface uses a set of 5 registers to read and write to the module, ReadEnable and WriteEnable is set if a register is being read or written to, respectively. The value sent to Address input decides what register is being read or written. DataIn and DataOut either writes or reads the value to the register. When the module is being used as a transmitter, TxEN is set and the address interface is used to put data into the module and the data is then transmitted through Tx. When used as a reciever RxEn is set, data then streams through the Rx input and can be read through the address interface. The module also has several internal signals, such as data buffers control signals.

3. Assertions

To feed the transmitter and receiver with data random data is generated and used as input to the transmitter, the transmitter then feeds the receiver with the data that it outputs on each clock cycle. This is done in the test bench, and the code for this is shown below.

```
1  task random_loop();
2      for(int i = 0; i < num_loops; i=i+1) begin
3          random_input();
4      end
5  endtask;
6
7  task random_input();
8      automatic logic[7:0] Data = '0;
9      logic [7:0] size;
10     size = $urandom_range(3, 130);
11
12     //Generate random data and write it to TX_buffer
13     for (int i = 0; i < size; i++) begin
14         Data = $urandom();
15         WriteAddress(TX_BUFF, Data);
16     end
17     $display("%d bytes of data written to TX_BUFF", size);
18
19     //Initiate transfer
20     WriteAddress(TX_SC, TX_ENABLE);
21     for(int i = 0; i < 2200; i=i+1) begin
22         Data = $urandom();
23         if(i==1000 && Data==127) begin
24             WriteAddress(3'b000, 8'b00000100);
25         end
26         @(posedge uin_hdlc.Clk);
27         uin_hdlc.Rx = uin_hdlc.Tx;
28   end
29 endtask
```

The random_input task generates a frame for transmission. The variable called size get assigned a random value between 3 and 130 on line 10. On line 13 to 16 a random value of 1 byte gets generated and put into the Tx buffer, the number of bytes are decided with the size variable. Testing shows that the data has to be at least 3 bytes, so that all the fields in the frame gets filled up. As the information field can only be a maximum of

126 bytes, the upper bound of 130 is used to generate cases where Tx_Full is supposed to be asserted. The Tx transfer gets enabled on line 20 and on line 21 to 28 a for loop writes the value from Tx output to Rx input on each clock cycle. Line 22 to 25 writes an abort frame signal to the transmitter at a pseudo-random moment after 1000 clock cycles has passed since the Tx transfer began. The loop runs for 2200 clock cycles to ensure that the transmission gets enough time to finish before a new transmission gets initiated. The random_loop task initiates the random_input task for a specified number of times, generating different inputs to the system.

In addition to this random-test, a test to generate an overflow on Rx is used, as well as a test which generates sequential data or a constant stream of data to DataIn.

The assertions are written based on the list of specifications that needs to be verified in the project description and is organised in subsections in this chapter thereafter. Many of the assertions use internal signals from both the transmitter and receiver in it's verification, and a list of all the internal signals and what they are used for can be found in the project description.

3.1. Correct data in RX buffer according to RX input

To verify that the data in the RX buffer is correct according to the serially received data on Rx, a function was made. The function, called *in_out_equal*, checks bit for bit that Rx_Data is equal to Rx. The code for the function is as follows:

```
1  function bit in_out_equal(logic [127:0][7:0]Data, logic [127*10:0]
       Serial_Data, int size);
2    automatic int seq = 0;
3    automatic int cnt = 0;
4    automatic bit zero_detected = 0;
5    automatic bit pass = 1;
6      automatic int unroll_count = 0;
7    automatic logic [(128*8)-1:0]Data_Unrolled  = 0;
8
9      for(int i = 0; i < size; i++) begin  //Unroll Data to one long
           array
10          for(int j = 0; j < 8; j++) begin
11              Data_Unrolled[unroll_count] = Data[i][j];
12              unroll_count += 1;
13          end
14     end
15     size=size*8; //Byte to bit size
16   for(int i = 0; i < size; i++) begin //Check bit for bit if the data
         matches
17     if(Serial_Data[i] == 1) begin
18       seq += 1;
19     end else begin
20       seq = 0;
```

```
21      end
22      if(zero_detected == 1) begin
23        pass &= (Serial_Data[i] == 0);
24              zero_detected = 0;
25              size++;
26              //$display("Pass: %b, Serial_Data[i]: %b, i: %d",pass,
                    Serial_Data[i],i);
27      end else begin
28        pass &= (Serial_Data[i] == Data_Unrolled[cnt]);
29        cnt += 1;
30              //$display("Pass: %b, Data_Unrolled[cnt]: %b, Serial_Data
                    [i]: %b, i: %d",pass,Data_Unrolled[cnt],Serial_Data[i
                    ],i);
31      end
32      if(seq == 5) begin
33        zero_detected = 1;
34              seq = 0;
35      end
36    end
37  //      $display("1: %b", Serial_Data[255:0]);
38  //      $display("2: %b", Data_Unrolled[255:0]);
39    return pass;
40  endfunction;
```

The function "unrolls" the Data array into one long array to make it easier to compare bits, requiring only one for-loop to cycle through. The framesize is input into the function and is multiplied by 8 to change it from byte to bit. The bit-comparison runs until the number of bits reported by FrameSize has been reached, but this size is increased by one bit each time a zero is inserted. When the counter for the number of consecutive ones reaches 5, the next bit on Rx should be 0. The function compares Rx to Rx_Data, and zero when a zero is inserted, and writes the boolean result to the variable pass, which is initialised as 1.

The function is used in an assertion, which is included below. Rx_Store and Rx_DataArray are two local variables used to store the data on Rx and Rx_Data, respectively. The assertion uses RxD, which is the data on Rx delayed by 9 clock cycles. This assumes that RxD is correct and that is always is delayed 9 clock cycles, which might be incorrect as no assertion has been written to verify this. The assertion stores the data on Rx_Data on each Rx_WrBuff-signal. The storage to Rx_Store and Rx_DataArray is repeated for 1 to infinite clock cycles until Rx_ValidFrame falls, which happens three clock cycles after Rx_ValidFrame is high. The comparison function is run three clock cycles after ValidFrame falls, as Rx_FrameSize is updated to the current framesize at that point.

```
1  property Data_Rx_buff; //Correct data in RX buffer according to RX
       input
2      logic[127*10:0] Rx_Store;
3      logic[127:0][7:0] Rx_DataArray;
4      int i = 0;
```

```
    int j = 0;
    @(posedge Clk) disable iff (!Rst) $rose(Rx_ValidFrame) ##7 (##0 (
        Rx_ValidFrame, Rx_Store[i] = $past(RxD,7), i++) [*2:10] ##0 (
        Rx_WrBuff, Rx_DataArray[j] = Rx_Data,j++))[*1:$] ##3 ($fell(
        Rx_ValidFrame)) |-> ##3 in_out_equal_rx(Rx_DataArray, Rx_Store
        , Rx_FrameSize);
  endproperty
```

3.2. Attempting to read RX buffer after aborted frame, frame error or dropped frame should result in 0

Three assertions are used to check that the buffer is zero on the specified occasions, and the code for these are included below. All assertions check that DataOut is zero when either of the specified signals are raised, indicating either an aborted frame, a frame error or a dropped frame.

```
property correct_data_rxBuff_Drop;
    @(posedge Clk) $rose(Rx_Drop) |-> DataOut==8'b00000000;
  endproperty


  property correct_data_rxBuff_Abort;
    @(posedge Clk) $rose(Rx_AbortSignal) |-> DataOut==8'b00000000;
  endproperty


  property correct_data_rxBuff_Error;
    @(posedge Clk) $rose(Rx_FrameError) |-> DataOut==8'b00000000;
endproperty
```

3.3. Correct bits set in RX status/control register after receiving frame

To assert that the correct bits are set in the status/control register, the assertion below has been written:

```
  property Rx_StatusControl; //Correct bits set in Rx status/control-
      register after receiving frame
    @(posedge Clk) disable iff (!Rst) $rose(Rx_EoF) |->
    if(Rx_AbortSignal)
        (##0 !Rx_Overflow ##0 Rx_AbortSignal ##0 !Rx_FrameError ##1 !
            Rx_Ready, $display("Aborted frame at: ",$time))
    else if (Rx_Overflow)
```

```
6         (##0 Rx_Overflow ##0 !Rx_AbortSignal ##0 !Rx_FrameError ##0
              Rx_Ready, $display("Overflow at: ",$time))
7     else if (Rx_FrameError)
8         (##0 !Rx_Overflow ##0 !Rx_AbortSignal ##0 Rx_FrameError ##0
              !Rx_Ready, $display("Frame error at: ",$time))
9     else
10        ##0 !Rx_Overflow ##0 !Rx_AbortSignal ##0 !Rx_FrameError ##0
              Rx_Ready;
11  endproperty
```

The antecedent is activated when Rx_EoF rises and a if-else statement decides what consequent to when asserting. If either Rx_AbortSignal, Rx_Overflow or Rx_FrameError is high, then then the other two has to be low. If Rx_AbortSignal or Rx_FrameError is high then Rx_Ready must be low and if Rx_Overflow is high then Rx_Ready also needs to be high. If there is no abort, overflow or frame error then Rx_Ready is high.

3.4. Correct TX output according to written TX buffer

This assertion is similar to the assertion for verifying the data in Rx and Rx_Data in chapter 3.1. It uses the same function, *in_out_equal*, to compare Tx_DataArray to the data transmitted on Tx. The function stores Tx_DataArray when it initialises as the data on the buffer is changed when the transfer has finished. The assertion then logs the data on Tx during the period when Tx_ValidFrame is high. Afterwards, it compares the data on Tx to Tx_DataArray, inserting zeros where needed. The code for the assertion is as follows:

```
1   property Tx_Output; //Tx output is correct based on Tx_buffer
2     logic[127*10:0] Tx_Store;
3     logic[127:0][7:0] Tx_DataArray_Log;
4     int i = 0;
5     @(posedge Clk) disable iff (!Rst || Tx_AbortedTrans) ($rose(
          Tx_ValidFrame), Tx_DataArray_Log = Tx_DataArray) ##10 (##0 (
          Tx_ValidFrame, Tx_Store[i] = Tx, i++))[*1:$] ##1 ($fell(
          Tx_ValidFrame)) |-> in_out_equal(Tx_DataArray_Log, Tx_Store,
          Tx_FrameSize);
6   endproperty
```

The assertion fails if the returned result from in_out_equal is false. In addition, the assertion is disabled if the transfer is aborted, removing false negatives as Tx is not done transmitting the data when ValidFrame falls, as the transfer is aborted.

3.5. Start and end frame generation

Since the start and end frame have the same pattern, one assertion are written to check that the correct frame are inserted at the beginning and end of a transmission. Once valid frame goes high, transmission starts after two clock cycles, and after valid frame goes low, the end pattern starts after one clock cycle. Unless the frame is abortet. The assertion below uses this by saying if Tx_ValidFrame changes and Tx_AbortFrame has not been high two clock cycle ago, the start or end frame sequence must have started to be transmittet within two clock cycles on Tx.

```
1
2   sequence startendframe;
3    !Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 !Tx;
4   endsequence
5
6   property Flag; //Start and end frame generation, 01111110
7     @(posedge Clk) disable iff (!Rst) !$stable(Tx_ValidFrame) ##0
          $past(!Tx_AbortFrame,2)|-> ##[0:2] startendframe;
8   endproperty
```

3.6. Zero insertion and removal

To check that zeros are inserted when transmitting, Tx_NewByte must be high and there has to be either 6 consecutive ones in Tx_Data or 6 consecutive ones between two consecutive Tx_Data bytes. If this is true, 24-35 clock cycle later, there must be five ones followed by a zero on the data transmitted on Tx. The code for the assertion performing this check is as follows:

```
1 property Zero_Insert; //Zero insert
2     @(posedge Clk) disable iff (!Rst || !Tx_ValidFrame ) Tx_NewByte
          ##1 (Tx_Data==?8'bxx111111 or Tx_Data==?8'bx111111x or Tx_Data
          ==?8'b111111xx or (Tx_Data==?8'b1xxxxxxx ##1 Tx_Data==?8'
          bxxx11111) or
3     (Tx_Data==?8'b11xxxxxx ##1 Tx_Data==?8'bxxxx1111) or (Tx_Data
          ==?8'b111xxxxx ##1 Tx_Data==?8'bxxxxx111) or (Tx_Data==?8'
          b1111xxxx ##1 Tx_Data==?8'bxxxxxx11) or (Tx_Data==?8'b11111xxx
           ##1 Tx_Data==?8'bxxxxxxx1))
4     ##[24:35] Tx[*5] |=> !Tx;
5   endproperty
```

To check that zeros are removed when receiving, the following assertion was written:

```
1   property Zero_Remove; //Zero removal
2     logic a;
3     reg [7:0] data_RX;
```

```
4       @(posedge Clk) disable iff (!Rst || !Tx_ValidFrame) (($rose(Rx)
            ##1 Rx[*4] ##1 !Rx ##1 (Rx==1 or Rx==0)), a = Rx) |-> ##13 (
            Rx_Data==?{2'bxx, a, 5'b11111} or Rx_Data==?{1'bx, a, 6'
            b11111x} or Rx_Data==?{a, 7'b11111xx} or (Rx_Data==?8'
            b1xxxxxxx ##8 Rx_Data==?{3'bxxx, a, 4'b1111}) or
5         (Rx_Data==?8'b11xxxxxx ##8 Rx_Data==?{4'bxxxx, a, 3'b111}) or (
              Rx_Data==?8'b111xxxxx ##8 Rx_Data==?{5'bxxxxx, a, 2'b11}) or
              (Rx_Data==?8'b1111xxxx ##8 Rx_Data==?{6'bxxxxxx, a, 1'b1}) or
               (Rx_Data==?8'b11111xxx ##8 Rx_Data==?{7'bxxxxxxx, a}));
6   endproperty
```

The left side of the expression is true when 5 consecutive ones are recieved followed by a zero. The incoming value after this pattern is stored in a local varable called "a". Then on the right side of the expression, the value in Rx_Data register should have the 5 consecutive ones followed by the value in the local variable 13 clock cycles later. If this is the case, the expression is true, else, the assertion fails.

3.7. Idle pattern generation and checking

To see if there only ones are generated and transmitted during the idle state, the following assertion was written:

```
1   property IDLE; //Idle pattern generation, 11111111
2       @(posedge Clk) disable iff (!Rst || Tx_FCSDone) !Tx_FCSDone[*10]
            ##0 !Rx_FrameError ##0 !Tx_ValidFrame[*10] |=>
3         //##[0:8] Tx[*8];
4           if($fell(Tx_FCSDone))
5               ##8 Tx[*8]
6           else
7               $past(Tx) ##0 Tx[*7];
8     endproperty
```

In this assertion the antecedent checks if the transmitter is idle, if it is idle Tx_FCSDone must be zero for 10 consecutive cycles, there can't be any frame error and Tx_ValidFrame must be low for 10 consecutive cycles. When this is true, an if-else statement checks that Tx is high for eight cycles, and if Tx_FCSDone is high after one cycle. Ff it is not, Tx has to be high for eight cycles right after Tx_ValidFrame was high.

3.8. Abort pattern generation and checking

To see if a flag sequence is detected, the assertion provided with the project files is used. Tt detects when there is a flag sequence in Rx and if Rx_FlagDetect is high two clock cyles later, the assertion passes. The code for the assertion is as follows:

```
 sequence Rx_flag;
    !Rx ##1 Rx [*6] ##1 !Rx;
  endsequence

// Check if flag sequence is detected
  property Abort_pattern;
    @(posedge Clk) Rx_flag |-> ##2 Rx_FlagDetect;
endproperty
```

An assertion to check both the error signal generation and checking has been made. It checks that Tx_AbortFrame has risen, and if it has, the assertion gets evaluated, then a non overlapping implication and a time delay of 3 clock cycles followed by the avbort sequence has to be generated on Tx, then after 3 clock cycles the Rx_AbortSignal must be asserted. This code is as follows:

```
  sequence abort;
    !Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx ##1 Tx;
endsequence

 property Verify8; //Abort signal generated correctly and checked by
    Rx
    @(posedge Clk) disable iff (!Rst) $rose(Tx_AbortFrame) |=>  ##3
       abort ##3 $rose(Rx_AbortSignal);
  endproperty
```

3.9. When aborting frame during transmission, Tx_AbortedTrans should be asserted

To check that Tx_AbortedTrans is asserted during transmission the assertion below is used:

```
  property Tx_AbortedTrans; //Aborting frame during transmission
     asserts Tx_AbortedTrans
    @(posedge Clk) disable iff (!Rst) $rose(Tx_AbortFrame) ##0
       Tx_DataAvail |-> Tx_AbortedTrans;
  endproperty
```

Tx_AbortFrame has to rise and Tx_DataAvail has to be high for the assertion to be evaluated and then Tx_AbortedTrans must be high the same clock cycle for it to be true.

3.10. Abort pattern detected during valid frame should generate Rx_AbortSignal

To assert that the Rx_AbortSignal is high when abort pattern is detected during a valid frame, the Rx_ValidFrame siganl must remain stable and the Rx_AbortSiganl has to go from zero to one on the antecedent of the expression. If this is true, the abort pattern needs to have happened 4 to 10 clock cycles ago for the assertion to pass. The code for the assertion is as follows:

```
sequence prev_abort;
    $past(Rx,4) and $past(Rx,5) and $past(Rx,6) and $past(Rx,7) and
        $past(Rx,8) and $past(Rx,9) and $past(!Rx,10);
endsequence

property Rx_AbortSignal; //Abort pattern detected and Rx_AbortSignal
    generated properly
    @(posedge Clk) disable iff (!Rst) $stable(Rx_ValidFrame) ##0
        $rose(Rx_AbortSignal) |-> prev_abort;
endproperty
```

3.11. CRC generation and checking

To check that the CRC is generated properly, the function checkCRC is used. The function calculates the CRC for the data in Tx_DataArray and compares it with the CRC calculated by the DUT. The function is as follows:

```
function automatic logic checkCRC([127:0] [7:0] arrayA, int size,
    logic FCSen); //Function for performing CRC-check
    automatic logic noError = 1'b1;
    logic [15:0] tempCRC;
    logic [23:0] tempStore;
//    $display("InArray =%h", arrayA[10:0]);
//    $display("Framesize =%d", size);

  tempCRC = {arrayA[size-1],arrayA[size-2]};
  arrayA[size-1] = '0;
  arrayA[size-2] = '0;

    tempStore[7:0]  = arrayA[0];
    tempStore[15:8] = arrayA[1];

    for (int i = 2; i < size; i++) begin
      tempStore[23:16] = arrayA[i];
      for (int j = 0; j < 8; j++) begin
        tempStore[16] = tempStore[16] ^ tempStore[0];
        tempStore[14] = tempStore[14] ^ tempStore[0];
```

```
20          tempStore[1]  = tempStore[1]  ^ tempStore[0];
21          tempStore[0]  = tempStore[0]  ^ tempStore[0];
22          tempStore = tempStore >> 1;
23        end
24      end
25  //    $display("time = %t", $time);
26
27
28  //    $display("tempCRC =%h", tempCRC);
29  //    $display("calcCRC =%h", tempStore[15:0]);
30
31     if (FCSen) begin
32  //    $display("Return =%b", tempCRC == tempStore[15:0]);
33
34       return (tempCRC == tempStore[15:0]);
35
36       end else begin
37  //    $display("Return force = 1");
38
39       return 1'b1;
40     end
41  endfunction
```

To check if the CRC is generated and checked properly, two assertions were made. First, the generate assertion initially stores the data on Tx_DataArray, along with the framesize, when Tx_FCSDone is finished. The assertion then waits for the first match of Tx_WriteFCS, using the first_match-operator to avoid trigging on more than two subsequent Tx_WriteFCS cycles. After 1 to 10 clock cycles, Tx_WriteFCS is active again, and the data on Tx_Data is stored to the local buffer variable already storing Tx_DataArray. The data on this cycle, and the next, is the calculated CRC. The data is input into the checkCRC function and compared. The code for the generate assertion is as follows:

```
1  property CRC_generate; //Check if CRC generation is correct
2    logic [127:0] [7:0] Tx_Buff;
3      logic [7:0] framesize;
4      @(posedge Clk) disable iff (!Rst || Tx_AbortedTrans) ($rose(
           Tx_FCSDone),Tx_Buff = Tx_DataArray,framesize = Tx_FrameSize)
           ##0 first_match(##[0:$] Tx_WriteFCS)
5      ##[1:10] (Tx_WriteFCS,Tx_Buff[framesize]=Tx_Data,framesize++) ##1
            (1'b1,Tx_Buff[framesize]=Tx_Data)
6       |-> checkCRC(Tx_Buff,framesize+1,1'b1);
7    endproperty
```

To check the CRC value on Rx, the assertion performs the same actions as for the generation. The difference is that it stores the received Rx_Data in a register and uses that for the CRC-checking. The assertion checks that Rx_FrameError is the opposite of the value returned from the checkCRC-function. The code for the CRC check function

is as follows:

```
1  property CRC_check; //Check if CRC is checked correctly, and that
       Rx_FrameError is set if the check fails.
2  logic [127:0] [7:0] tempBuffer = '0;
3  int framesize = 0;
4   @(posedge Clk) disable iff (!Rst) Rx_flag ##[18:19] Rx_NewByte ##0
        (##[7:9] (Rx_NewByte,tempBuffer[framesize]=Rx_Data, framesize
       ++)) [*1:128] ##0 Rx_FlagDetect
5   |-> ##6 (1,tempBuffer[framesize]=Rx_Data, framesize++) ##0 (
       Rx_FrameError == !checkCRC(tempBuffer,framesize,Rx_FCSen));
6  endproperty
```

3.12. Check if Rx_EoF is generated after receiving a whole Rx_Frame

To see if the Rx_EoF is high when a frame is received, the code below was written. The assertion gets evaluated if Rx_ValidFrame goes from high to low. For the assertion to be true, Rx_EoF has to be low and then rise after one clock cycle and then low again after another clock cycle. In hindsight, the $rose and $fell functions in SystemVerilog could have been used to simplify the antecedent expression.

```
1  property Rx_EoF; //Check that Rx_EoF is generated when a whole frame
       has been received
2    @(posedge Clk) disable iff (!Rst) $fell(Rx_ValidFrame) |-> !
       Rx_EoF ##1 Rx_EoF ##1 !Rx_EoF;
3  endproperty
```

3.13. Check if Rx_Overflow is asserted after receiving more than 128 bytes

In this assertion, Rx_StartFCS has to go from high to low, then 6 cycles later Rx_NewByte has to be high 129 or more times. To check that Rx_NewByte has been high 129 times, the non-consecutive repetition operator is used. When the left side of the expression is true, a non overlapping implication is used so that Rx_Overflow must be high after one cycle. The code for the assertion is as follows:

```
1  property Rx_Overflow; //Rx_Overflow is asserted when receiving more
       than 128 bytes of data, [->x] = consecutive sequence
2    @(posedge Clk) disable iff (!Rst || Rx_EoF) $fell(Rx_StartFCS)
       ##6 (Rx_NewByte)[->129] |=> Rx_Overflow;
3  endproperty
```

3.14. Rx_FrameSize should be equal to the number of received bytes in a frame

To assert that Rx_FrameSize is equal to the number of received bytes, the following assertion was written:

```
property Rx\_FrameSize; //Check that Rx_FrameSize is equal to the
    number of bytes received during the frame
  int bytecount = 0;
    @(posedge Clk) disable iff (!Rst) $rose(Rx_ValidFrame) ##0
        (##[7:9]$rose(Rx_NewByte),bytecount++)[*1:127] ##5 Rx_EoF |->
        ##5 Rx_FrameSize == (bytecount-2);
endproperty
```

In the antecedent, Rx_ValidFrame must go high. Then Rx_NewByte has to rise after 7 to 9 cycles, it must do so at least once and not more than 127 times. Each time this happens, a local variable called bytecount is incremented. The antecedent is true when Rx_EoF is high 5 cycles after Rx_NewByte is high. On the consequent, Rx_FrameSize gets compared to the bytecount after 5 cycles. The total bytecount is subtracted by two to account for the two FCS-frames which are not included in Rx_FrameSzie.

3.15. Rx_Ready should indicate byte(s) ready to be read in Rx_buffer

To check if the buffer is ready to be read, Rx_EoF signal get checked if it goes from high to low after Rx_Ready is set to high. This assertion is as follows:

```
property Rx_Ready; //Rx_Ready should indicate bytes in RX buffer are
    ready to be read
    @(posedge Clk) disable iff (!Rst) $rose(Rx_Ready) ##1 Rx_Ready
        |-> $fell(Rx_EoF);
endproperty
```

3.16. Non-byte aligned data or error in FCS checking should result in frame error

To assert that Rx_FrameError is high when there is non-byte aligned transfer, Rx_FlagDetect gets checked if it is high 1-7 clock cycles after Rx_NewByte is high. When that happens Rx_FrameError must go high 2 clock cycles later. This assertion is as follows:

```
 property Byte_align; //Check if non-byte aligned transfer. During
    normal operation, Rx_Newbyte and Rx_FlagDetect are triggered
    simultaneously.
```

```
2     @(posedge Clk) disable iff (!Rst)  Rx_NewByte ##[1:7]
          Rx_FlagDetect
3     |-> ##2 $rose(Rx_FrameError);
4   endproperty
```

A frame error as a result of an error in the FCS checking is included in the assertion for the FCS checking in chapter 3.11

3.17. Tx_Done should be asserted when the entire TX buffer has been read for transmission

To see that Tx_Done is asserted when the buffer has been read, Tx_DataAvail gets checked if it has gone from high to low. If it has, Tx_Done must have been high one clock cycle ago. The code for this assertion is as follows:

```
1 property Tx_Done; //Tx_Done should be asserted when the entire TX
      buffer has been read for transmission
2     @(posedge Clk) disable iff (!Rst) $fell(Tx_DataAvail) |-> $past(
          Tx_Done, 1);
3   endproperty
```

3.18. Tx_Full should be asserted after writing 126 or more bytes to the Tx buffer

The assertion for checking that the transmitter signals when the buffer is full is as follows:

```
1 property Tx_Full; //Tx_Full should be asserted after writing 126 or
      more bytes to Tx buffer
2     @(posedge Clk) disable iff (!Rst) $rose(Tx_Enable) ##1
          Tx_DataAvail ##1 Tx_FrameSize == 8'd126 |-> $past(Tx_Full, 2);
3   endproperty
```

For the antecedent to be true, Tx_Enable must go high, then after one clock cycle Tx_DataAvail must be high and then one clock cycle later Tx_FrameSize has to be equal to 126. For the consequent to be true Tx_Full has to have been high 2 clock cycles prior for the antecedent to be true.

4. Coverage

To check that all different states of the system are reached, a covergroup has been made in the test bench. The code for this is shown below. The internal signals that get covered here are Tx_FrameSize, Tx_AbortedTrans, Tx_Full, Tx_Done, Rx_Overflow, Rx_AbortSignal and Rx_FrameError. DataIn also gets covered to see that all the different input values are covered. DataIn and FrameSize gets crossed to see that all the different values get covered for all the different frame sizes. This covergroup does not cover all the different sequences of DataIn for all the different frame sizes.

```
1  //Cover
2  covergroup hdlc_cg () @(posedge uin_hdlc.Clk);
3        dataIn: coverpoint uin_hdlc.DataIn {
4          bins DataIn[] = {[0:255]};
5        }
6          Framesize: coverpoint uin_hdlc.Tx_FrameSize {
7              bins FrameSizes[] = {[0:126]};
8          }
9          Tx_AbortedTrans: coverpoint uin_hdlc.Tx_AbortedTrans {
10             bins Tx_not_Aborted = {0};
11             bins Tx_Aborted = {1};
12         }
13         Tx_Full: coverpoint uin_hdlc.Tx_Full {
14             bins Tx_not_Full = {0};
15             bins Tx_Full = {1};
16         }
17         Tx_Done: coverpoint uin_hdlc.Tx_Done {
18             bins Tx_not_Done = {0};
19             bins Tx_Done = {1};
20         }
21         Rx_Overflow: coverpoint uin_hdlc.Rx_Overflow {
22             bins Rx_no_Overflow = {0};
23             bins Rx_Overflow = {1};
24         }
25         Rx_AbortSignal: coverpoint uin_hdlc.Rx_AbortSignal {
26             bins Rx_not_Aborted = {0};
27             bins Rx_Aborted = {1};
28         }
29         Rx_FrameError: coverpoint uin_hdlc.Rx_FrameError {
30             bins Rx_no_FrameError = {0};
31             bins Rx_FrameError = {1};
32         }
```

```
33      dataIn_and_FrameSize: cross dataIn, Framesize;
34
35        endgroup
```

The coverage report is included as Appendix A. There, the random input is used with 3000 random values, followed by a test directly on Rx to simulate an overflow on the receiver, followed by a random test to check that the system works correctly after an overflow. The test results in 100% on all coverpoints but one, the crossed value between DataIn and FrameSize got 99.8%, another test with 10 000 random values was also tried and it got 99.9% coverage. To get 100% coverage, a testbench which goes through all the values for all the different frame sizes could have been written, or the random input test could run for more cycles. This would probably have given the same result in regards to the assertions, as the data on DataIn is the same, albeit in a different order.

TThe coverage report shows that all assertions are triggered and finish without error, except for 4 and 11 which fail 42 times each. The cause of this is that the framesize is less than three. This problem will be discussed in the next chapter.

5. Discussion

During testing, several errors in the DUT were discovered. Framesizes of less than three bytes result in errors in transfer and causes frame errors, most likely as a result of improper CRC-generation. There is another bug with CRC. When the calculated CRC checksum has five consecutive ones, TxChannel inserts a zero from the zero-injection, but RxChannel doesn't remove it, resulting in a non-byte aligned data and a frame error. Other than these edge cases, the HDLC module works as expected, as far as the testing has showed. Another error is that Rx_Ready is high for one clock cycle simultaneously as Rx_AbortSignal is high, while it should remain low.

The accuracy of the results of the cross-coverage between Tx_FrameSize and DataIn is not certain as the data is sample on each positive clock edge of Clk, and as a result several invalid values are registered, such as new DataIN during old frame sizes. How detrimental to the coverage results this is uncertain, but it is minimised by the fact that random input data is used, as this should, given enough simulations, cover most cases of varying DataIn and Tx_FrameSize.

Whether the use of concurrent assertions was the most effective and accurate way is uncertain, as immediate assertions could have been better in the cases where a specific problem is tested, such as an overflow on Rx or an aborted frame. Still, concurrent assertions allow for more accurate coverage and have in some cases highlighted unexpected situations.

A. Coverage report

```
ASSERTION RESULTS:
---------------------------------------------------------------------------
Name                          File(Line)                  Failure  Pass
                                                          Count    Count
---------------------------------------------------------------------------
/test_hdlc/u_assertion_bind/Receive_FlagDetect_Assert
                              assertions_hdlc.sv(346)        0   5953
/test_hdlc/u_assertion_bind/correct_data_rxBuff_Drop_Assert
                              assertions_hdlc.sv(350)        0      1
/test_hdlc/u_assertion_bind/correct_data_rxBuff_Abort_Assert
                              assertions_hdlc.sv(354)        0      1
/test_hdlc/u_assertion_bind/correct_data_rxBuff_Error_Assert
                              assertions_hdlc.sv(358)        0     78
/test_hdlc/u_assertion_bind/Data_Rx_buff_assert
                              assertions_hdlc.sv(362)        0   2897
/test_hdlc/u_assertion_bind/Rx_SC_assert
                              assertions_hdlc.sv(366)        0   2977
/test_hdlc/u_assertion_bind/Tx_Output_assert
                              assertions_hdlc.sv(370)       42   2932
/test_hdlc/u_assertion_bind/Flag_assert
                              assertions_hdlc.sv(374)        0   5949
/test_hdlc/u_assertion_bind/Zero_Insert_Assert
                              assertions_hdlc.sv(378)        0   2328
/test_hdlc/u_assertion_bind/Zero_Remove_Assert
                              assertions_hdlc.sv(382)        0  24402
/test_hdlc/u_assertion_bind/IDLE_assert
                              assertions_hdlc.sv(386)        0 5089533
/test_hdlc/u_assertion_bind/Abort_pattern_assert
                              assertions_hdlc.sv(390)        0      1
/test_hdlc/u_assertion_bind/Tx_AbortedTrans_Assert
                              assertions_hdlc.sv(394)        0      2
/test_hdlc/u_assertion_bind/Rx_AbortSignal_Assert
                              assertions_hdlc.sv(398)        0      1
/test_hdlc/u_assertion_bind/CRC_generate_Assert
                              assertions_hdlc.sv(402)       42   2932
/test_hdlc/u_assertion_bind/CRC_check_Assert
                              assertions_hdlc.sv(406)        0   2888
/test_hdlc/u_assertion_bind/Rx_EoF_Assert
                              assertions_hdlc.sv(411)        0   2977
/test_hdlc/u_assertion_bind/Rx_Overflow_Assert
                              assertions_hdlc.sv(415)        0      1
```

```
/test_hdlc/u_assertion_bind/Rx_FraneSize_Assert
                        assertions_hdlc.sv(419)        0  2788
/test_hdlc/u_assertion_bind/correct_data_rxBuff_Abort_Error
                        assertions_hdlc.sv(423)        0    78
/test_hdlc/u_assertion_bind/Rx_Ready_Assert
                        assertions_hdlc.sv(427)        0  2898
/test_hdlc/u_assertion_bind/Byte_align_Assert
                        assertions_hdlc.sv(431)        0    78
/test_hdlc/u_assertion_bind/Tx_Done_Assert
                        assertions_hdlc.sv(435)        0  2976
/test_hdlc/u_assertion_bind/Tx_Full_Assert
                        assertions_hdlc.sv(439)        0   112


COVERGROUP COVERAGE:
----------------------------------------------------------------------------------------------------------------------------------

Covergroup                                                                                                              Metric
    Goal     Status

----------------------------------------------------------------------------------------------------------------------------------

 TYPE /test_hdlc/u_testPr/hdlc_cg                                                                                        99.8%
     100     Uncovered
     covered/total bins:                                                                                                32521
         32907
     missing/total bins:                                                                                                  386
         32907
     % Hit:                                                                                                            98.8%
         100
     Coverpoint hdlc_cg::dataIn                                                                                       100.0%
         100     Covered
          covered/total bins:                                                                                             256
              256
          missing/total bins:                                                                                               0
              256
          % Hit:                                                                                                      100.0%
              100
     Coverpoint hdlc_cg::Framesize                                                                                    100.0%
         100     Covered
          covered/total bins:                                                                                             127
              127
          missing/total bins:                                                                                               0
              127
          % Hit:                                                                                                      100.0%
              100
     Coverpoint hdlc_cg::Tx_AbortedTrans                                                                              100.0%
         100     Covered
```

```
75          covered/total bins:                                    2
                      2
76          missing/total bins:                                    0
                      2
77          % Hit:                                             100.0%
                100
78      Coverpoint hdlc_cg::Tx_Full                            100.0%
            100     Covered
79          covered/total bins:                                    2
                      2
80          missing/total bins:                                    0
                      2
81          % Hit:                                             100.0%
                100
82      Coverpoint hdlc_cg::Tx_Done                            100.0%
            100     Covered
83          covered/total bins:                                    2
                      2
84          missing/total bins:                                    0
                      2
85          % Hit:                                             100.0%
                100
86      Coverpoint hdlc_cg::Rx_Overflow                        100.0%
            100     Covered
87          covered/total bins:                                    2
                      2
88          missing/total bins:                                    0
                      2
89          % Hit:                                             100.0%
                100
90      Coverpoint hdlc_cg::Rx_AbortSignal                     100.0%
            100     Covered
91          covered/total bins:                                    2
                      2
92          missing/total bins:                                    0
                      2
93          % Hit:                                             100.0%
                100
94      Coverpoint hdlc_cg::Rx_FrameError                      100.0%
            100     Covered
95          covered/total bins:                                    2
                      2
96          missing/total bins:                                    0
                      2
97          % Hit:                                             100.0%
                100
98      Cross hdlc_cg::dataIn_and_FrameSize                     98.8%
            100     Uncovered
99          covered/total bins:                                32126
```

```
                32512
100           missing/total bins:                                  386
                32512
101           % Hit:                                             98.8%
                100
102   Covergroup instance \/test_hdlc/u_testPr/hdlc_cg_inst
103                                                              99.8%
                                                                   100
                                                                   Uncovered
104       covered/total bins:                                    32521
              32907
105       missing/total bins:                                      386
              32907
106       % Hit:                                                 98.8%
              100
```