TFE4171 - Design of Digital Systems 2

Term project

Bjørg Solem
Stine Olsen
Emanuela Tran
May 2022

Contents

List of Figures

List of Tables

1 Introduction

This report covers the verification of an untested HDLC(High-Level Data Link Control) module. First the report will explain what the HDLC is composed of, introduce the different approaches to implement assertions, explain how assertions has been used to verify the module and last cover the coverage report.

1.1 Description of the HDLC module

The high-level data link control(HDLC) is a data communication protocol, which provides bit oriented code-transparent data transmission[1]. Table 1 illustrates the structure of a HDLC frame where the FCS(Frame Check Sequence) is generated in the module.

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

Table 1: HDLC Frame

As seen in the table every frame should start and end with a flag sequence where the flag sequence should be "0111_1110". An abort pattern, "1111 1110", can be generated such that a frame can be terminated. If this pattern is detected the current frame will be discarded.

The FCS bytes are generated inside the HDLC module and calculated with the Cycle Redundancy Check (CRC) method[p.2][1]. CRC is used to ensure that and check that a frame has arrived as expected. Figure1 illustrates a block diagram om the HDLC module and its interface. The module consists of 9 inputs and 2 outputs.

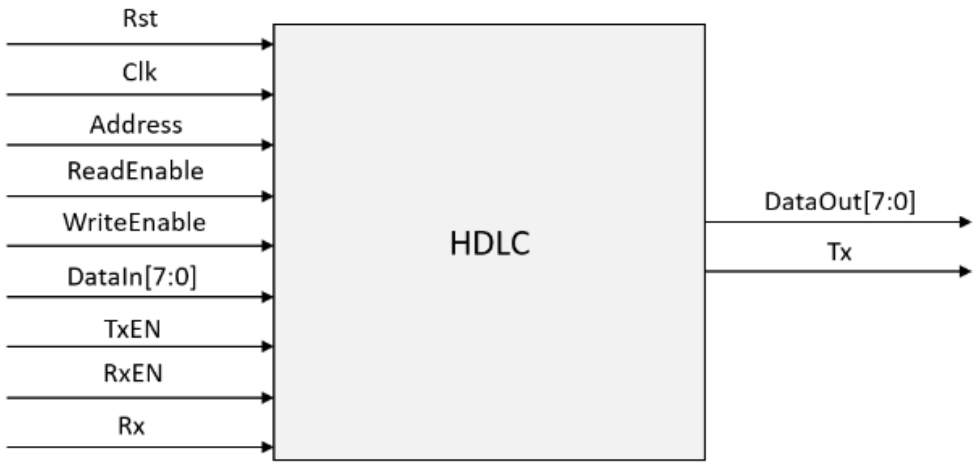

Figure 1: Block diagram of HDLC module[1]

Figure 2 and figure 3 illustrates how receive and transmit are implemented in the HDLC module. This report will use the different signals shown in both block diagrams to verify and test the module.

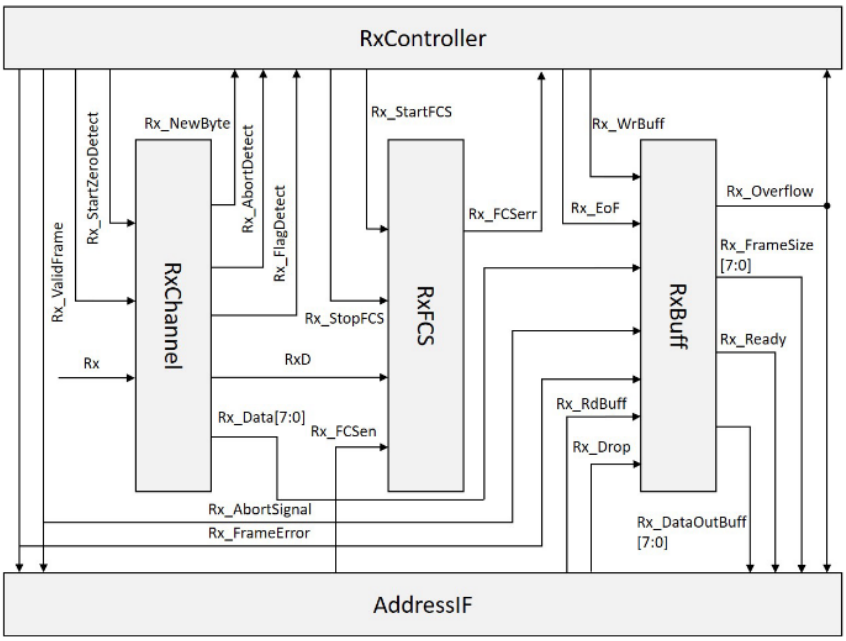

Figure 2: Block diagram of RX logic [1]

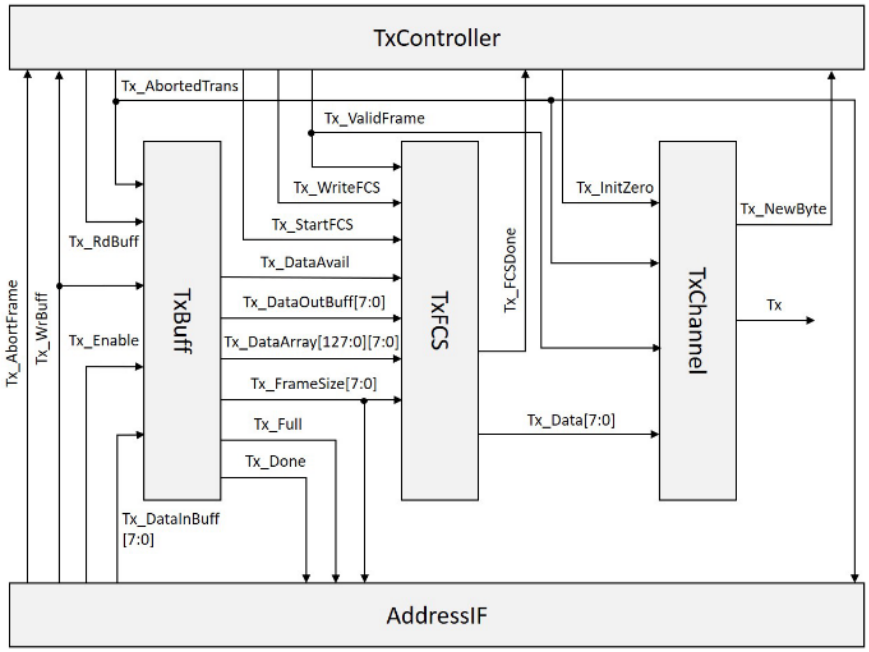

Figure 3: Block diagram of TX logic [1]

1.2 Verification of the HDLC module

To ensure that the design can be integrated it must meet certain specifications. This report uses SytemVerilog Assertions to verify the design according to the 18 specification listed below:

1. Correct data in RX buffer according to RX input. The buffer should contain up to 128 bytes (this includes the 2 FCS bytes, but not the flags).
2. Attempting to read RC buffer after aborted frame, frame error or dropped frame should result in zeros.
3. Correct bits set in RX status/control register after receiving frame. Remember to check all bits. I.e. after an abort the *Rx_Overflow* bit should be 0, unless an overflow also occured.
4. Correct TX output according to written TX buffer.
5. Start and end of frame pattern generation (Start and end flag : 0111_1110).
6. Zero insertion and removal for transparent transmission.
7. Idle pattern generation and checking (1111_1111 when not operating).
8. Abort pattern generation and checking (1111_1110). Remember that the 0 must be sent first.
9. When aborting frame during transmission, Tx_AbortedTrans should be asserted.
10. Abort pattern detected during valid frame should generate Rx_AbortSignal.
11. CRC generation and Checking.
12. When a whole RX frame has been received, check if end of frame is generated.
13. When receiving more than 128 bytes, Rx_Overflow should be asserted.
14. Rx_FrameSize should equal the number of bytes received in a frame, (max. 126 bytes = 128 bytes in buffer - 2 FCS bytes).
15. Rx_Ready should indicate byte(s) in RX buffer is ready to be read.
16. Non-byte aligned data or error in FCS checking should result in frame error.
17. Tx_Done should be asserted when the entire TX buffer has been read for transmisson.
18. Tx_Full should be asserted after writing 126 or more bytes to the TX buffer (overflow).

2 Assertions

An assertion is s statement that validates assumptions or check the condition in a program. Assertions can notify you if some legal or illegal combination of values of internal program variables has occured[2].

In pseudo-code, an assertion can look like this:

```
if (condition) then
    take action
```

An assertion observes the state of a program and can block further execution of a code, but never alter the program itself. With the hardware descriptive language SystemVerilog, assertions can be written inline with other language constructs without the need to create special pragmas or similar restrictions. This report uses the to main types of SystemVerilog assertions: immediate and concurrent assertions.

2.1 Immediate assertion

Immediate assertions check if an expression in a procedural block is true at any given instance of time. It the block is true a "pass block" will be executed while if the block is false a "fail block" is executed.

An immediate assertion has the following form:

```
assert (expression)
    pass_block;
else
    fail_block;
```

In this report immediate assertion has been used to verify specification: 1, 2, 3, 4 and 11.

2.2 Concurrent assertion

Concurrent assertion detects a behavior over a period of time and while expression for immediate assertion is evaluated whenever the expression changes value, evaluation for a concurrent assertion happens right before the clock edge. Evaluating an assertion just before a clock edge ensures that all variables have attained their stable and race free values.

As described in section 2.1 immediate assertions occurs within a procedural block. Concurrent assertions differ from immediate assertions where a concurrent assertions can be done within a procedural block or within a module.

Since concurrent assertions are clock edge driven, their timing model is based on "clock ticks". A clock tick represents an active edge of a clock and the time interval between clock tick is the active period of the associated clock[2].

To explain how a concurrent assertion is implemented lets assume there are three signals: reset, req and ack. We want to make a assertion that detects the following sequences of events.

- Sequence 1 : req should be asserted five clocks after reset is de-asserted.
- Once req is asserted, ack is asserted two clocks later.

This assertion in pseudo-code has the following form:

```
if (Sequence 1 occurs followed by Sequence 2)
    execute pass_block;
else
    execute fail_block;
```

In this report concurrent assertion has been used on the specification: 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17 and 18.

3 Specification

3.1 First specification

Correct data in the RX buffer according to RX input. The buffer should contain up to 128 bytes (this includes the 2 FCS bytes, but not the flags)

The task VerifyNormalReceive verifies the correct value in Rx status/control. The verification verifies that Rx_ready is true, and that all other RO (read only) bits in RX_SC are false. VerifyNormalReceive also asserts that Rx_buff has the correct data. It is correct if the Rx data buffer is not full. This is solved by the use of a for-loop, where it checks every bit in the Rx data buffer.

```
task VerifyNormalReceive(logic [127:0][7:0] data, int Size);
    logic [7:0] ReadData;
    wait(uin_hdlc.Rx_Ready);

    ReadAddress('h02, ReadData);

    a_VerifyNormalReceive_0: assert(ReadData[0] == 1 )
            $display("PASS: VerifyNormalReceive PASSED. Rx_Buff has data to read");
    else begin
            $display("FAILED: VerifyNormalReceive FAILED. Value in Rx_SC is %h",
            ↪  ReadData);
            TbErrorCnt++;
    end

    a_VerifyNormalReceive_2: assert(ReadData[2] == 0 )
        $display("PASS: VerifyNormalReceive PASSED. No Frame error");
    else begin
        $display("FAILED: VerifyNormalReceive FAILED. Value in Rx_SC is %h",ReadData);
        TbErrorCnt++;
    end

    a_VerifyNormalReceive_3: assert(ReadData[3] == 0 )
        $display("PASS: VerifyNormalReceive PASSED. No Abort signal");
    else begin
        $display("FAILED: VerifyNormalReceive FAILED. Value in Rx_SC is %h", ReadData);
        TbErrorCnt++;
    end

    a_VerifyNormalReceive_4: assert(ReadData[4] == 0 )
        $display("PASS: VerifyNormalReceive PASSED. No overflow signal");
    else begin
        $display("FAILED: VerifyNormalReceive FAILED. Value in Rx_SC is %h", ReadData);
        TbErrorCnt++;
    end

    for (int i =0; i< Size; i++) begin
        ReadAddress('h03,ReadData);
                a_Rx_Buff_not_full: assert (ReadData == data[i])
                $display("PASS: VerifyNormalReceive PASSED. Rx_Buff has correct data");
                else begin
                    $display("FAIL: VerifyNormalReceive FAILED. Value in Rx_Buff is
                    ↪  %h", ReadData);
                      TbErrorCnt++;
                end
    end

  endtask
```

3.2 Second specification

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

The second specification if solved by the tasks VerifyAbortReceive, VerifyFCSErrorReceive and VeryfiDropReceive.

3.2.1 RX, aborted frame

The task VerifyAbortReceive verifies correct value in the Rx status/control register, and checks that the Rx data buffer is zero after abort. It checks that the RX buffer has no data, no frame error, abort signal is asserted and no overflow signal is detected, this is done by setting the related bits in Rx_Sc, using assertions. Additionally the task asserts that Rx_buff is empty ('b0000 0000), by using the ReadAddress task together with the address 0x3 (Rx_Databuffer) and the ReadData byte. If one assertion fails a fail-message will occur and the error count will increase by one.

```
task VerifyAbortReceive(logic [127:0][7:0] data, int Size);
    logic [7:0] ReadData;
        ReadAddress('h02,ReadData); //Rx_SC address is 0x2

        //assert that rx_sc is correct value for abort, all RO bits except abort should
        ↪  be 0. We dont know what value the WO bits will have

    a_Correct_Val_0: assert (ReadData[0]==0)
        $display("PASS: VerifyAbortReceive PASSED, Rx_Buff has no data ");
    else begin
        $display("FAIL: VerifyAbortReceive FAILED,  wrong value in ReadData:
        ↪  %h",ReadData);
        TbErrorCnt++;
    end

    a_Correct_Val_2: assert (ReadData[2]==0)
        $display("PASS: VerifyAbortReceive PASSED, No frame error ");
    else begin
        $display("FAIL: VerifyAbortReceive FAILED,  wrong value in ReadData:
        ↪  %h",ReadData);
        TbErrorCnt++;
    end

    a_Correct_Val_3: assert (ReadData[3]==1)
        $display("PASS: VerifyAbortReceive PASSED, Abort signal asserted ");
    else begin
        $display("FAIL: VerifyAbortReceive FAILED,  wrong value in ReadData:
        ↪  %h",ReadData);
        TbErrorCnt++;
    end

    a_Correct_Val_4: assert (ReadData[4]==0)
        $display("PASS: VerifyAbortReceive PASSED, No overflow signal ");
    else begin
        $display("FAIL: VerifyAbortReceive FAILED,  wrong value in ReadData:
        ↪  %h",ReadData);
        TbErrorCnt++;
    end
```

```
    //assert that rx_buff is empty
    ReadAddress('h03,ReadData); //Rx_Buff address is 0x3

    a_DataBuf_zero: assert (ReadData=='b00000000)
        $display("PASS: VerifyAbortReceive PASSED, Rx_Buff is empty");
    else begin
        $display("FAIL: VerifyAbortReceive FAILED, Rx databuf is not zero: %h",ReadData
);
        TbErrorCnt++;
    end

endtask
```

3.2.2 RX, frame error

The task VerifyFCSErrorReceive verifies that error in FCS-checking results in a frame error. This is done by first setting the Rx_FCSen bit in RX_SC, put ReadData into RX_SC, and lastly check whether ReadData is equal to 00000100, this means that Rx_FrameError is the only bit set.

Further, we have to make sure that RX_DataBuf is zero. This is done by placing ReadData into Rx_Buf using the readaddress task, and then check if ReadData==00000000.

Note that this assertion was not met as the assertion did not get a pass or fail in the simulation log. The group thinks the reason for this is that the stimuli written did not function as expected such that the group were not able to verify that the assertion was correct or not.

```
task VerifyFCSErrorReceive(logic [127:0][7:0] data, int Size);
    logic [7:0] ReadData;

    //Check all RO bits: Should have only Rx_FrameError set

    WriteAddress('h02,'h10); //Rx_FCSen should be set in RXSC
    ReadAddress('h02,ReadData); //Rx_SC address is 0x2

    //FCSerr=1;

    a_VerifyErr_0: assert (ReadData == 'b00000100)
        $display("PASS: VerifyFrameErrorReceive PASSED. Rx_Ready not set");
    else begin
        $display("FAILED: VerifyFrameErrorReceive FAILED. Rx_Ready set");
        TbErrorCnt++;
    end

    //And Error in FCS checking results in frameerror
    //Check that RX_databuf is zero
    ReadAddress('h03,ReadData);

    a_VerifyFrameErrorReceive_zero: assert (ReadData=='b00000000)
        $display("PASS: VerifyFrameErrorReceive PASSED, Rx_Buff is empty");
    else begin
        $display("FAIL: VerifyFrameErrorReceive FAILED, Rx databuf is not zero:
        ↪  %h",ReadData);
        TbErrorCnt++;
    end
endtask
```

3.2.3 RX, dropped frame

The task VerifyDropReceive verifies correct value in the Rx status/control register such that the Rx data buffer is zero after a frame is dropped. This is done similarly as the task above, where we set Rx_Drop in RX_SC, then put ReadData into the Rx_Buf, and then check whether ReadData is zero.

```
task VerifyDropReceive(logic [127:0][7:0] data, int Size);
        logic [7:0] ReadData;

    WriteAddress('h02,'h02); //RX_SC addr is 0x2 and its value should be 0x2, since
    ↪  RX_DROP should be set
    ReadAddress('h03,ReadData);
        a_Drop_DataBuf_zero: assert (ReadData=='b00000000)
        $display("PASS: VerifyDropReceive PASSED, Rx_Buff is empty");
    else begin
        $display("FAIL: VerifyDropReceive FAILED, Rx databuf is not zero: %h",ReadData);
        TbErrorCnt++;
    end
endtask
```

3.3 Third specification

Correct bits set in RX status/control register after receiving frame. Remember to check all bits. I.e. after an abort the Rx_Overflow bit should be 0, unless an overflow also occurred.

The task VerifyOverflowReceive verifies that rx_buff has data (Rx_ready is set), no frame error(Rx_FrameError is 0), no abort signal (Rx_AbortSignal is 0) and overflow signal is set (Rx_Overflow is 1) in the Rx_SC register. If one of the assertion fails it will display a fail-message and increase the error count.

```
task VerifyOverflowReceive(logic [127:0][7:0] data, int Size);
    logic [7:0] ReadData;
    wait(uin_hdlc.Rx_Ready);

    ReadAddress('h02, ReadData);

    a_Verify_Overflow_0: assert ( ReadData[0]==1)
        $display("PASS: VerifyOverflowRecive PASSED. Rx_Buff has data to read");
    else begin
        $display("FAIL: VerifyOverflowReceived FAILED.");
        TbErrorCnt++;
    end

    a_Verify_Overflow_2: assert (ReadData[2]==0)
        $display("PASS: VerifyOverflowReceive PASSED. No frame error.");
    else begin
        $display("FAIL: VerifyOverflowReceived FAILED.");
        TbErrorCnt++;
    end

    a_Verify_Overflow_3: assert (ReadData[3]==0)
      $display("PASS: VerifyOverflowReceive PASSED. No abort signal.");
    else begin
      $display("FAIL: VerifyOverflowReceived FAILED.");
      TbErrorCnt++;
    end
```

```
    a_Verify_Overflow_4: assert (ReadData[4]==1)
        $display("PASS: VerifyOverflowReceive PASSED. Overflow signal asserted.");
    else begin
        $display("FAIL: VerifyOverflowReceived FAILED.");
        TbErrorCnt++;
    end

  endtask
```

3.4 Fourth specification

Correct TX output according to written TX buffer

The task TxDataCorrect verifies that the TX output is right according to the written TX buffer. This is done by generating random values to the Tx_Data array. That array is then written to the Tx_Buffer by the use of the task WriteAddress.

To find out if the the output is correct Tx_Enable must be set in Tx_Sc. Finally check that uin_hdlc.Tx_DataOutBuff==Tx_Data. If that is true, then the value in Tx_value is the same as the Tx buffer.

```
task TxDataCorrect();
    logic [7:0] Tx_Data;

    for (int i = 0; i < $size(Tx_Data); i++) begin
        Tx_Data[i]= $urandom();
        WriteAddress('h01, Tx_Data[i]); //write value of data to txbuffer
    end

    //Tx_Data=£urandom();
    @(posedge uin_hdlc.Clk);
    WriteAddress('h00,8'b00000010); //TX_SC is h'00, set enabl
    @(posedge uin_hdlc.Clk);

    a_TxDataCorrect: assert (uin_hdlc.Tx_DataOutBuff==Tx_Data)
        $display("PASS: Tx_data=Tx_Buffer");
    else begin
        $error("Fail: Tx_Data is not equal to Tx_Buffer");
        TbErrorCnt++;
    end

    repeat(25) //let task finish
    @(posedge uin_hdlc.Clk);

  endtask
```

3.5 Fifth specification

Start and end of frame pattern generation (Start and end flag: 0111_1110)

For RX, if a RX flag is detected, the pattern 0111_1110, seen on the left side of the implication operator, is generated and the Rx_FlagDetect signal will be set.

For TX, if the Tx_ValidFrame signal rises and Tx_AbortFrame has not been previously detected, the flag-sequence will be generated. This is disabled if reset is set.

```
property RX_FlagDetect;
    @(posedge Clk) !Rx ##1  Rx [*6] ##1 !Rx |->  ##2 Rx_FlagDetect;
endproperty

property TX_FlagGenerate;
    disable iff (!Rst)
    @(posedge Clk) !Tx_AbortFrame ##2 $rose(Tx_ValidFrame) |-> ##[0:2] !Tx ##1 Tx[*6]
    ↪  ##1 !Tx;
endproperty

//assertions:
RX_FlagDetect_Assert : assert property (RX_FlagDetect) begin
    $display("PASS: start and end flag detected");
end else begin
    $error("start and end flag not detected");
    ErrCntAssertions++;
end

TX_FlagGenerate_Assert : assert property (TX_FlagGenerate) begin
    $display("PASS: Start and end frame generated");
end else begin
    $error("TX Flag sequence did not generate start and end frame");
    ErrCntAssertions++;
end
```

3.6 Sixth specification

Zero insertion and removal for transparent transmission

In the HDLC behavior it is specified that in order to avoid the data in the information field to accidentally generate a flag sequence or an abort pattern, a zero should be inserted whenever 5 continuous ones are being transmitted. This is done by seeing if there ever is a case where we have the pattern 111110 anywhere while there is a new byte incoming for TX, the next bit in Tx should be 0.

On the receiving side, whenever 5 continuous ones are received, the following zero should be removed. This is done by checking if the pattern 0111110 can be detected for Rx signal, and that the Rx_ValidFrame has been high for 6 consecutive samples, then in the next 9:17 clock cycles, when there is a new incoming byte (Rx_NewByte), we should only be left with 5 consecutive incoming ones, without the zero in Rx_Data.

```
property TX_ZeroInsertion;
    disable iff (!Rst || !Tx_ValidFrame)
    @(posedge Clk)  $rose(Tx_NewByte) and Tx_Zero_Check |->  !Tx;
endproperty

property RX_ZeroRemoval;
    disable iff (!Rst)
    @(posedge Clk) (!Rx ##1 Rx[*5] ##1 !Rx and Rx_ValidFrame [*6]) |-> ##[9:17]
    ↪  Rx_NewByte ##1 Rx_Data[*5];
```

```
endproperty


//assertions:
TX_ZeroInsertion_Assert: assert property (TX_ZeroInsertion) begin
    $display("PASS: TX Insertion passed");
  end else begin
    $error("TX Insertion failed");
    ErrCntAssertions++;
  end

RX_ZeroRemoval_Assert: assert property (RX_ZeroRemoval) begin
   $display("PASS: RX zero removal passed");
   end else begin
    $error("RX zero removal failed");
    ErrCntAssertions++;
  end
```

3.7 Seventh specification

Idle pattern generation and checking (1111_1111 when not operating).

The idle pattern should be generated if the Tx_ValidFrame is low, if Tx_AbortedTrans is low and the transmission frame size= 0000_0000. The idle pattern is 1111_1111, we check if Tx is set for 8 consecutive clk cycles on the right side of the implication operator.

Idle pattern checking is done by checking if Rx-signal has been high for 8 consecutive clk cycles, if so: One clock cycle later, the Rx_FlagDetect should go low.

```
//generation
property TX_GenerateIdlePattern;
    disable iff(!Rst)
    @(posedge Clk) !Tx_ValidFrame and Tx_FrameSize==8'b0 and !Tx_AbortedTrans |-> Tx[*8
];
endproperty

//Checking
property RX_CheckIdlePattern;
    disable iff (!Rst)
    @(posedge Clk) Rx[*8] |-> ##1 !Rx_FlagDetect;
endproperty

//assertions:
TX_GenerateIdlePattern_Assert: assert property (TX_GenerateIdlePattern) begin
    $display("PASS: Idle pattern generated");
end else begin
    $error("FAIL: Idle pattern not generated");
    ErrCntAssertions++;
end

RX_CheckIdlePattern_Assert: assert property (RX_CheckIdlePattern) begin
    $display("PASS: Idle pattern observed");
end else begin
    $error("FAIL: Idle pattern not observed");
    ErrCntAssertions++;
end
```

3.8 Eighth specification

Abort pattern generation and checking (1111_1110). Remember that the 0 must be sent first.

To generate the abort pattern, we first have to see if the signal TX_AbortFrame is high while we are transmitting (while TX_ValidFrame is high), if so, we have to generate the pattern itself. This is done by first sending a 0, then 7 consecutive ones.

In order to check for abort pattern, we look for the sequence 1111 1110 in signal Rx. If this happens, Rx_AbortDetect is to be set two clk cycles later.

```
//generation
property TX_GenerateAbortPattern;
    disable iff (!Rst)
    @(posedge Clk) Tx_AbortFrame and Tx_ValidFrame |-> ##4 !Tx ##1 Tx [*7];
endproperty

//checking
property RX_CheckAbortPattern;
    disable iff (!Rst)
    @(posedge Clk) !Rx ##1 Rx[*7] |-> ##2 $rose(Rx_AbortDetect);
endproperty

//assertions:
RX_CheckAbortPattern_Assert: assert property (RX_CheckAbortPattern) begin
    $display("PASSED CheckAbortPattern: Abort pattern observed");
end else begin
    $error("FAIL in CheckAbortPattern: Abort pattern not observed");
    ErrCntAssertions++;
end

TX_GenerateAbortPattern_Assert: assert property (TX_GenerateAbortPattern) begin
    $display("PASSED GenerateAbortPattern: Abort pattern generated");
end else begin
    $error("FAIL in GenerateAbortPattern: Abort pattern not generated");
    ErrCntAssertions++;
end
```

3.9 Ninth specification

When aborting frame during transmission, TX_AbortedTrans should be asserted.

In order to check this, it is necessary to see if Abort_Frame signal is set while we also have data available, (TX_DataAvail is set), and then one clock cycle later, TX_AbortFrame goes low. If all this is true, we check if TX_AbortedTrans signal goes high one clock cycle later.

```
property TX_TransmissionAbort;
disable iff (!Rst)
    @(posedge Clk)  Tx_AbortFrame && Tx_DataAvail ##1 $fell(Tx_AbortFrame) |=>
    ↪  $rose(Tx_AbortedTrans);
endproperty

//asserion:
TX_TransmissionAbort_Assert: assert property (TX_TransmissionAbort) begin
    $display("PASSED TransmissionAbort: Tx_AbortedTrans asserted");
end else begin
    $error("FAIL TransmissionAbort: Tx_AbortedTrans not asserted");
    ErrCntAssertions++;
end
```

3.10 Tenth specification

Abort pattern detected during valid frame should generate RX_AbortSignal.

Verifies correct RX_AbortSignal behavior. If abort is detected during valid frame, then abort signal should go high. If RX_AbortSignal is asserted we then display a pass message, if not we give an error message and increase error count.

```
property RX_AbortSignal;
    @(posedge Clk) (Rx_ValidFrame && Rx_AbortDetect) |=> Rx_AbortSignal;
endproperty

//assertion:
RX_AbortSignal_Assert : assert property (RX_AbortSignal) begin
    $display("PASS: Abort signal");
end else begin
    $error("AbortSignal did not go high after AbortDetect during validframe");
    ErrCntAssertions++;
end
```

3.11 Eleventh specification

CRC generation and Checking.

Not covered.

3.12 Twelfth specification

When a whole RX frame has been received, check if end of frame is generated

This is checked by seeing if the RX_ValidFrame has fallen, and then if the signal RX_EoF rises one clock cycle afterwards. If this is the case, we will end up with a pass message.

```
property Rx_EndOfFile;
disable iff(!Rst)
    @(posedge Clk) $fell(Rx_ValidFrame) |-> ##1 $rose(Rx_EoF);
endproperty

//assertion:
RX_EoF_Assert: assert property (Rx_EndOfFile) begin
    $display("PASS: RX End Of file is generated");
end else begin
    $error("FAIL: RX End Of file is not generated");
    ErrCntAssertions++;
end
```

3.13 Thirteenth specification

When receiving more than 128 bytes, RX_Overflow should be asserted.

Here, it has to be checked that if we have gotten 129 or more bytes, that the overflow signal is set. To check if we have gotten a byte, we look at the RX_NewByte signal. Since we do not have to get all 129 bytes consecutively, we use the non-consecutive GoTo repetition operator. We also have to make sure that there is a valid frame.

```
property RX_Overflow;
disable iff (!Rst || !Rx_ValidFrame)
    @(posedge Clk) $rose(Rx_ValidFrame) and ($rose(Rx_NewByte) [->129])  |=>
    ↪  $rose(Rx_Overflow);
endproperty

//assertion:
RX_Overflow_Assert: assert property (RX_Overflow) begin
    $display ("PASS: RX overflow asserted");
end else begin
    $error("FAIL: RX overflow not asserted");
    ErrCntAssertions++;
end
```

3.14 Fourteenth specification

Rx_FrameSize should equal the number of bytes received in a frame (max. 126 bytes = 128 bytes in buffer – 2 FCS bytes).

In order to check this, we have to go through a whole frame. It starts whenever RX_ValidFrame rises. After this we have count the number of bytes received, one is added for each time RX_NewByte is received. Maximum can be 128 bytes in buffer, so we have to check for 128 consecutive cycles. In the end, when RX_EoF is asserted, we have to see that the FrameSize is equal to the number of bytes we counted minus two.

```
property FrameSizeNumOfBytes;
    int numBytes=0;
    disable iff(!Rst)
    @(posedge Clk) $rose(Rx_ValidFrame) and
    ↪  (##[7:9]$rose(Rx_NewByte),numBytes++)[*1:128] ##5 Rx_EoF |=> Rx_FrameSize ==
    ↪  (numBytes-2);
endproperty

//assertion:
FrameSizeNumOfBytes_Assert: assert property (FrameSizeNumOfBytes) begin
    $display("PASS FrameSizeNumOfBytes: Framesize is equal to number of bytes received"
);
end else begin
    $error("FAIL FrameSizeNumOfBytes: Framesize is not equal to number of bytes
    ↪  received ");
    ErrCntAssertions++;
end
```

3.15 Fifteenth specification

Rx_Ready should indicate byte(s) in RX buffer is ready to be read.

When Rx_Ready rises we check in the next clock cycle that we have reached the end of the file (Rx_EoF rises), and that the frame is no longer valid, indicating that the RX buffer is ready to be read.

```
property RX_Ready;
    disable iff(!Rst)
    @(posedge Clk) $rose(Rx_Ready) |-> !Rx_ValidFrame and $rose(Rx_EoF)
endproperty

//assertion:
RX_Ready_Assert: assert property (RX_Ready) begin
    $display("PASS: Rx_Ready set: Buffer ready to be read");
end else begin
    $error("FAIL: RX_Ready not set");
    ErrCntAssertions++;
end
```

3.16 Sixteenth specification

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

As the group was not able to write a correct stimuli this specification was not met as it is uncertain if the assertion was written correctly or not. However, the groups thought process was that if Rx_FCSen was set, we would have to check whether Rx_FrameError rises on the next clock cycle.

```
property RX_FrameErrorCheck;
    disable iff (!Rst)
    @(posedge Clk) Rx_FCSen |=> $rose(Rx_FrameError);
endproperty

//assertion:
RX_FrameErrorCheck_Assert: assert property (RX_FrameErrorCheck) begin
    $display("PASS FrameErrorCheck: FrameError signal gone high");
end else begin
    $error("FAIL FrameErrorCheck: FrameError signal not gone high");
    ErrCntAssertions++;
end
```

3.17 Seventeenth specification

Tx_Done should be asserted when the entire TX buffer has been read for transmission.
The entire TX buffer has been read for transmission whenever the Tx_DataAvail signal is no longer high, if Tx_DataAvail falls, we then have to check whether the Tx_Done signal is set, indicating that the transmission is done.

```
property TX_TransmissionDone;
    disable iff (!Rst)
    @(posedge Clk) $fell(Tx_DataAvail) |-> Tx_Done;
endproperty

//assertion:
TX_TransmissionDone_Assert: assert property (TX_TransmissionDone) begin
    $display("PASS: TransmissionDone: Tx_Done asserted");
end else begin
    $error("FAIL TransmissionDone: Tx_Done not asserted");
    ErrCntAssertions++;
end
```

3.18 Eighteenth specification

Tx_Full should be asserted after writing 126 or more bytes to the TX buffer (overflow).

If Tx_Done signal goes low, and we are writing to the TX buffer for 126 non-consecutive clock cycles, we have overflow in TX buffer. Therefore we have to check if Tx_Full was set in the previous clock cycle, since it should have been set after writing for 125 clock cycles.

```
property TX_Overflow;
    disable iff (!Rst )
    @(posedge Clk) $fell(Tx_Done) and ((Tx_WrBuff) [->126]) |-> $past(Tx_Full);
endproperty

//assertion:
TX_Overflow_Assert: assert property (TX_Overflow) begin
    $display("PASS: Overflow: Tx_Overflow asserted");
end else begin
    $error("FAIL TransmissionDone: Tx_Overflow not asserted");
    ErrCntAssertions++;
end
```

4 Coverage report

For our coverage report, three different cover groups where made: one for the register interface; one for receive signals; and one for the transmit signals. Cover points were made for all signals used in the assertions. The binary signals have two bins, covering true and false, while the data and size signals have 128 bins, with one for every bit, covering their total length. The coverage report obtained a total coverage of 74.11%. Total coverage of 100% has not been obtained due to some stimulus not being generated and thus some specifications have not been met.

References

[1] Karianne Krokan Kragseth. HDLC Module Design Description. 2017.

[2] Project veriPage. SystemVeriolog Assertion.