NTNU

Kunnskap for en bedre verden

Department of Electronic Systems

TFE4171 - Design of Digital Systems 2

Project (Final Submission)

Authors:
Andrew Glover Martey - 557813
Muhammad Naseer ud din - 557818
Tobi N. Jada - 554270

May, 2022

1 PART A

1. **Check for correct values after an abort in the VerifyAbortReceive task.**

```
// VerifyAbortReceive should verify correct value in the Rx status/control
// register, and that the Rx data buffer is zero after abort.
// Specification 2,1
task VerifyAbortReceive(logic [127:0][7:0] data, int Size);
  logic [7:0] ReadData;

//Read data in RX status/ control register (Rx_SC), address = 0x2
 ReadAddress(3'b010,ReadData);

//Check that RX buffer has no data to be read
  assert ( ReadData[0] != 1'b1) $display("PASS: VerifyAbortReceive:: Rx_Buff has no data");
  else begin
    $error("FAIL: VerifyAbortReceive:: Rx_Buff has data");
    TbErrorCnt++;
  end

//Check that there is no error in received RX frame
  assert ( ReadData[2] != 1'b1) $display("PASS: VerifyAbortReceive:: No Frame Error");
  else begin
    $error("FAIL: VerifyAbortReceive:: Frame Error");
    TbErrorCnt++;
  end
//Check that RX frame was aborted (that abort signal is asserted)
   assert ( ReadData[3] == 1'b1) $display("PASS: VerifyAbortReceive:: Abort Signal Asserted");
  else begin
    $error("FAIL: VerifyAbortReceive:: Value in RX_SC is not an abort signal");
    TbErrorCnt++;
  end
// Check that RX buffer has not overflown
  assert ( ReadData[4] != 1'b1) $display("PASS: VerifyAbortReceive:: No Overflow Signal");
  else begin
    $error("FAIL: VerifyAbortReceive:: Overflow Signal");
    TbErrorCnt++;
  end
// Check that Rx Data Buffer is empty (after abort signal is asserted)
  ReadAddress(3'b011,ReadData);
  assert(ReadData == 8'b0) $display("PASS: VerifyAbortReceive:: Rx Data Buffer is empty");
  else begin
    $error("FAIL: VerifyAbortReceive:: Rx Data Buffer is not Empty!");
    TbErrorCnt++;
  end


endtask
```

Figure 1: Immediate assertions checking for the correct value after an abort.

In the task VerifyAbortReceive(), we first use the ReadAddress task to read from the Rx status/control register using the address 3'b010 as specified in the data sheet - HDLC Module Design Descripition. This task reads from this register and gives an output, ReadData.To check for when the abort signal is high, we assert this by using ReadData[3] == 1'b1. Rx_AbortSignal is the fourth bit as per the field to check for abort signal going high. The $display function is used to print a pass message and the $error function is used to display the errors to demonstrate the severity during run-time.Using the ReadAddress task we read the buffer register to check if it is full or not. We assert this by using ReadData == 8'b0. We display the messages using the system functions display and error.

As an extra check we also verified that Rx_Ready, Rx_FrameError, and Rx_Overflow are not asserted by checking that the bit positions in the register are not high.

2. **Check for correct values after an overflow in the VerifyOverflowReceive task.**

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

 //Read data in RX status/ control register (Rx_SC), address = 0x2
  ReadAddress(3'b010, ReadData);

  //Check that RX buffer has some data to be read
  assert (ReadData[0] == 1'b1) $display("PASS: VerifyOverflowReceive:: Rx_Buff has data to read");
  else begin
    $error("FAIL: VerifyOverflowReceive:: Rx_Buff has NO data to read!");
    TbErrorCnt++;
  end

  // Read data in Frame length register (Rx_Len)
   ReadAddress(3'b100,datalength);
  assert(datalength == Size) $display("PASS: VerifyOverflowReceive :: No Frame Error");
  else begin
    $error("FAIL: VerifyOverflowReceive :: Frame Error");
    TbErrorCnt++;
  end

  //Check that RX frame was not aborted (that abort signal is not asserted)
  assert (ReadData[3] != 1'b1) $display("PASS: VerifyOverflowReceive :: No Abort Signal");
  else begin
    $error("FAIL: VerifyOverflowReceive:: Abort Signal Detected!");
    TbErrorCnt++;
  end

  // Check that RX buffer has  overflown
  assert (ReadData[4] == 1'b1) $display("PASS: VerifyOverflowReceive :: Overflow Signal Asserted");
  else begin
    $error("FAIL: VerifyOverflowReceive :: Overflow Signal NOT Asserted");
    TbErrorCnt++;
  end

endtask
```

Figure 2: Immediate assertions checking for the correct value after an overflow.

In the task VerifyOverflowReceive(), we first use the ReadAddress task to read from the Rx status/control register using the address 3'b010 as specified in the data sheet - HDLC Module Design Descripition. This task reads from this register and gives an output, ReadData.To check for when the overflow signal is high, we assert this by using ReadData[4] == 1'b1. Rx_Overflow is the fifth bit as per the register to check for when Rx_Overflow goes high. We check that the data in the Rx data buffer is correct by checking that there is no frame error. This is done using the ReadAddress task to read from the RX Frame length register at address 0x4. We compare this with the size of the data received when the VerifyOverflowReceive task was called.

3. **Check for correct values after normal operation in the VerifyNormalReceive task.**

   We use the task ReadAddress to get data from the register we pass as an argument to the task. The data is a byte stored in ReadData. We check the RX control status register for the data ready signal.

```
// VerifyNormalReceive should verify correct value in the Rx status/control
// register, and that the Rx data buffer contains correct data.
task VerifyNormalReceive(logic [127:0][7:0] data, int Size);
  logic [7:0] ReadData;
  logic [7:0] datalength;

  wait(uin_hdlc.Rx_Ready);

  //Read data in RX status/ control register (Rx_SC), address = 0x2
  ReadAddress(3'b010, ReadData);

  //Check that RX buffer has some data to be read
  assert (ReadData[0] == 1'b1) $display("PASS: VerifyNormalReceive:: Rx_Buff has data to read");
  else begin
    $error("FAIL: VerifyNormalReceive:: Rx_Buff has NO data to read!");
    TbErrorCnt++;
  end

  // Read data in Frame length register (Rx_Len)
  ReadAddress(3'b100,datalength);

  // Verify that data has same size as frame length register : Check that there is no frame error
  assert(datalength == Size) $display("PASS: VerifyNormalReceive:: No Frame error");
  else begin
    $error("FAIL: VerifyNormalReceive :: Frame error!");
    TbErrorCnt++;
  end

  //Check that RX frame was not aborted (that abort signal is not asserted)
  assert (ReadData[3] != 1'b1) $display("PASS: VerifyNormalReceive:: No Abort Signal");
  else begin
    $error("FAIL: VerifyNormalReceive:: Abort Signal Detected!");
    TbErrorCnt++;
  end

  // Check that RX buffer has not overflown
  assert (ReadData[4] != 1'b1) $display("PASS: VerifyNormalReceive:: No Overflow Signal");
  else begin
    $error("FAIL: VerifyNormalReceive:: Overflow Signal Detected!");
    TbErrorCnt++;
  end

  // For incoming data check that receive buffer has correct data
  for(int i= 0; i < Size; i++ )
  begin
  // Reaad data in RX data buffer
  ReadAddress (3'b011,ReadData);
  assert(ReadData == data[i]) $display("PASS: VerifyNormalReceive:: Rx_Buff has correct data");
  else begin
    $error("FAIL: VerifyNormalReceive :: Rx_Buff has not got correct data!");
    TbErrorCnt++;
  end
  end

endtask
```

Figure 3: Immediate assertions checking for the correct value after a normal operation.

We check the RX control status register and assert that there is no overflow signal. We check the RX frame length register to assert that there is no frame error when we compare to the size of the data received. We also check that for every incoming data in the RX data buffer, there is a match. i.e that the correct data is stored.

4. **Say whether the following statement is true or false: "assertions in the tasks mentioned above are all concurrent"**

False. None of the assertions have a property. There are no sequences of events that must be checked for an assertion to pass or fail.

5. **Explain in your report the difference between immediate and concurrent assertions.**

   Immediate assertions are non temporal, they only contain combinational logic, they are like if statements and are evaluated at the same time as they are being computed in code. That means that we always get an evaluation and then a result when an immediate assertion is encountered in code. Concurrent assertions are temporal and can contain sequential logic. They have a condition that must be true at the time of the sampling edge, the antecedent, and if it is not true they will not be evaluated.

6. **Show Rx_flag sequence (behaviour of Rx_FlagDetect). Explain it.**

```
/****************************************
 *  Verify correct Rx_FlagDetect behavior  *
 ****************************************/

sequence Rx_flag;
  // INSERT CODE HERE

  Rx == 0 ##1 Rx[*6] ##1 Rx == 0;
endsequence

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

RX_FlagDetect_Assert : assert property (RX_FlagDetect) begin
  $display("PASS: Flag detect");
end else begin
  $error("Flag sequence did not generate FlagDetect");
  ErrCntAssertions++;
end
```

Figure 4: Rx_flag sequence for flag identification.

The sequence, the antecedent, checks for the right flag sequence which is ”01111110” by requiring the first RX bit to be low, the six next consecutive ones to be high, and the last one to be low. If the antecedent is true Rx_FlagDetect asserted to be verified two clock cycles later.

7. **Show Rx_AbortSignal property. Explain it.**

```
/****************************************
 *  Verify correct Rx_AbortSignal behavior  *
 ****************************************/

//If abort is detected during valid frame. then abort signal should go high
property RX_AbortSignal;
  // INSERT CODE HERE
  @(posedge Clk)  Rx_AbortDetect && Rx_ValidFrame |=> Rx_AbortSignal;
endproperty

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
```

Figure 5: Rx_Abortsignal property for abort signal verification.

The antecedent in the property is true when both Rx_AbortDetect and Rx_ValidFrame are high. If the antecedent is true, the consequent Rx_AbortSignal will be checked at the next clock cycle.

2 PART B

2.1 specification 4

This assertion is from the task VerifyNoramlTransmit. We first wait for the read new byte signal then while data is being read from the buffer the data to be transmitted matches it.

```
for (int i = 0; i < Size - 1; i++)
    begin
       wait(uin_hdlc.Tx_RdBuff);
      assert( data[i] == uin_hdlc.Tx_DataOutBuff)
     $display("PASS: VerifyNoramlTransmit:: TX_Buff has correct data");
   else begin
     $display("FAIL: VerifyNoramlTransmit:: TX_Buff data incorrect!");
     TbErrorCnt++;
   end
      @(posedge uin_hdlc.Clk);
    end
```

2.2 specification 5

This assertion checks that if Tx_validFrame goes from low to high or from high to low and then Tx_AbortedTrans is low(not aborted frame) in the same clock cycle, then the StartEndFrameFlag sequence(start/stop frame) should follow within two next clock cycles(clock delay range).

```
property StartEndPatternGenerator;
@(posedge Clk) disable iff (!Rst) !$stable(Tx_ValidFrame)
##0 !Tx_AbortedTrans |-> ##[1:2] StartEndFrameFlag;
endproperty
```

2.3 specification 6

Concurrent assertions are used here. To check if zeros are inserted during transmission, the sequence StartEndFrameFlag and $Tx_V alidFrameofobservedshouldimplythatoneclockticklater, theTx_V alidFrameshou$

```
//Specification 6
property InsertZeros;
  @(posedge Clk) disable iff (!Rst) StartEndFrameFlag ##0 Tx_ValidFrame |=> (!Tx_ValidFrame[->1])
endproperty

property RemoveZeros;
 @(posedge Clk) disable iff (!Rst) ZeroRemove and Rx_ValidFrame  |-> ##[9:17] Rx_NewByte ##1 Real0
endproperty
```

2.4 specification 7

This assertion checks that 8 consecutive 1's are transmitted when a frame is not being transmitted(before and after frames), that is when Tx_ValidFrame is low and Tx_FrameSize is zero.

```
sequence IdlePatternGenerationAndChecking;
        Tx[*8];
endsequence
```

```
property Idle_Pattern_Generator_And_Checker;
@(posedge Clk) disable iff (!Rst) !Tx_ValidFrame
&& Tx_FrameSize == 8'b0 |-> IdlePatternGenerationAndChecking;
endproperty
```

2.5 specification 8

```
//Specification 8
property AbortPatternGeneratorAndCheckerRX;
@(posedge Clk) disable iff (!Rst) AbortPatternGenerationAndChecking
##0 Rx_ValidFrame |-> ##2 Rx_AbortDetect;
endproperty

property AbortPatternGeneratorAndCheckerTX;
    // @(posedge Clk) disable iff (!Rst) abort_pattern ##0 Tx_ValidFrame  |=> Tx_AbortFrame;
@(posedge Clk) disable iff (!Rst)  $rose(Tx_AbortedTrans) |-> ##2 abort_pattern;
endproperty
```

2.6 specification 9

We check from the Tx status/ control register and check bit position 3 for when it goes high. We check at least 2 clock cycles to assert that Tx_AbortedTrans gets asserted

```
for(int i = 0; i < 2; i++) begin
    ReadAddress(3'h0, ReadData);
    if (ReadData & (1 << 3))
        break;
end

  //Check that RX frame was aborted (that abort signal is asserted)
assert ( ReadData[3] == 1'b1)
    $display("PASS: VerifyAbortTransmit:: Tx_AbortedTrans Asserted");
else begin
    $error("FAIL: VerifyAbortTransmit:: Tx_AbortedTrans not asserted");
    TbErrorCnt++;
    end
```

2.7 specification 10

This assertion checks if Rx_AbortSignal is generated one clock cycle after if Rx_Validframe and Rx_AbortDetect are high.

```
//Specification 10
property GenerateRx_AbortSignal;
@(posedge Clk) disable iff (!Rst) Rx_ValidFrame && Rx_AbortDetect |=> Rx_AbortSignal;
endproperty
```

2.8 specification 11

2.8.1 Part A - Rx CRC Check

/This task checks that the checksum of the message+CRC equals Zero. It receives a "data" variable that is passed as an argument to the "GenerateFCSBytes" task. The checksum of the data variable is written into the "FCSBytes" variable, and then the "FCSBytes" variable is checked against 0. If it is not zero it will give an error.

```
//CRC Check for RX
task VerifyRxCRC(logic [127:0][7:0] data, int Size);
logic [15:0] FCSBytes;
    int i;
    int errorcount;
    errorcount = 0;
    for(i=0;i<Size;i++) begin
        $display("Size is - %d,  Value inside data is = %b", Size, data[i]);
    end

GenerateFCSBytes(data, Size, FCSBytes);

if(FCSBytes[7:0] != 8'B00000000 ) begin
   $error("Mismatch of FCA in Size byte: %b and data: %b", FCSBytes[7:0], data[Size]);
errorcount++;
end
if(FCSBytes[15:8] != 8'B00000000) begin
$error("Mismatch of FCS in Size byte: %b and data: %b", FCSBytes[15:8], data[Size+1]);
errorcount++;
end
if(errorcount==0)
$display("PASS: CRC check in RX passed with 0 errors");
endtask
```

2.8.2 Part B - Tx CRC Check

/This task checks for CRC generation. A message, transmit, is passed to the task "GenerateFCS-Bytes" which generates a remainder for the message, the fcs bytes thereof are placed in the variable "FCSBytes". Then transmit is written to the tx buffer before tx is enabled for transmission. The two for loops from line 183 checks if the data transmitted on tx equals stored in temp equals the two last bytes(FCS Bytes) in transmit.

```
// CRC Check for Tx
task VerifyTxCRC(int Size);
logic [15:0] FCSBytes;
logic [127:0][7:0] transmit;
logic [7:0] temp;
logic [7:0] Tx_Enable;
int i;
int j;
int errorcount;




for(i = 0; i<Size; i++) begin
```

```
transmit[i] = 8'b00100100;
end

//CRC Generation
transmit[Size] = 8'b00000000;
transmit[Size+1] = 8'b00000000;

GenerateFCSBytes(transmit, Size, FCSBytes);

transmit[Size] = FCSBytes[7:0];
transmit[Size+1] = FCSBytes[15:8];

Tx_Enable = 8'b00000010;

wait(!uin_hdlc.Tx_ValidFrame);

for(i=0;i<Size;i++) begin
WriteAddress(3'b001,transmit[i]);
end
WriteAddress(3'b000,Tx_Enable);

wait(uin_hdlc.Tx_ValidFrame);

repeat(9)
@(posedge uin_hdlc.Clk);

for(i=0;i<Size+2;i++) begin
for(j=0;j<8;j++) begin
@(posedge uin_hdlc.Clk);
temp[j] = uin_hdlc.Tx;
end
if(i==Size) begin
if(temp != transmit[Size]) begin
$error("Error in TX FCS where transmitted FCS %b but FCS is: %b",temp[j],transmit[Size]);
errorcount++;
end
end
if(i==Size+1) begin
if(temp != transmit[Size+1]) begin
$error("Error in TX FCS where transmitted FCS %b but FCS is: %b",temp[j],transmit[Size+1]);
errorcount++;
end
end
end

if(!errorcount) begin
$display("PASS: CRC check in TX passed with 0 error");
end
//$display("Error number for nonmatching TX CRC data is: %d.",errorcount);

endtask
```

2.9 specification 12

This assertion checks that Rx_EoF(end of frame) is generated 1 clock cycle after Rx_ValidFrame goes low(whole valid frame has been received).

```
//Specification 12
property EndOfFrameDetected;
@(posedge Clk) disable iff (!Rst) $fell(Rx_ValidFrame) |=> Rx_EoF;
endproperty
```

2.10 specification 13

This assertion checks that an overflow(Rx_Overflow) is generated when Rx_ValidFrame goes from low to high(rose) and Rx_NewByte is high 128+1 times (overflow at 129 times) by using the non consecutive repetition operator([-]).

```
//Specification 13
property Rx_OverflowAt129Byte;
@(posedge Clk) disable iff (!Rst|| !Rx_ValidFrame)( $rose(Rx_ValidFrame))
##0 ($rose(Rx_NewByte)[->129]) |=>$rose(Rx_Overflow);
endproperty
```

2.11 specification 14

We read the rx frame length register and place the value in ReadData. We then check if the size of ReadData is equal to Size, which means that the rx frame length/size is equal to the number of bytes received in a frame. Else we display an error message an increment TbErrorCnt.

```
    ReadAddress(3'b100, ReadData);

    assert (ReadData == Size)
      $display("PASS: Rx_FrameSize equals number of bytes in frame");
    else begin
      $display("ERROR: Rx_FrameSize is not equal to number of bytes in frame!");
      TbErrorCnt++;
    end
```

2.12 specification 15

This assertion checks that Rx_Eof goes from low to high and Rx_ValidFrame is low, meaning that the bytes in the RX buffer are ready to be read, when Rx_Ready is high.

```
//Specification 15
property RxBufferReadyToBeRead;
@(posedge Clk) disable iff (!Rst) $rose(Rx_Ready) |-> $rose(Rx_EoF) and !Rx_ValidFrame;
endproperty
```

2.13 specification 16

An immediate assertion was employed for this specification. task VerifyFrameError reads data in RX status/ control register (Rx_SC) and outputs it as ReadData.Both NonByteAligned and FCSerr signals call this task since they should result in a frame error. We assert this using Rx_FrameError signal which is at bit position 2 of address 0x2.

```
ReadAddress(3'b010, ReadData);
assert(ReadData[2] == 1'b1 ) $display("PASS: VerifyFrameError::FrameError Detected");
```

```
else begin
$error ("FAIL: VerifyFrameError::FrameError Detection Failed!");
```

2.14 specification 17

In Task Transmit, we first wait on the signal uin_hdlc.Tx_Done to be sure that data in the Tx buffer has been read. Using task ReadAddress, we now read from the TX status/control register to check its contents. ReadData is outputted. Using this data from the register we check that the first bit (Tx_Done) is high(asserted).

```
wait(uin_hdlc.Tx_Done);
    ReadAddress(3'h0, ReadData);
   assert(ReadData[0] == 1'b1)
     $display("PASS: TX_Done flag asserted");
   else begin
     $display("FAIL: TX_Done flag not asserted");
   end
```

2.15 specification 18

With this specification an immediate assertion was used. Task VerifyOverflowTransmit was written to assert this. First we read the TX status/control register using the ReadAddress task. We then check that the Size of the data if greater or equal to 126 sets the Tx_AbortedTrans bit high.

```
   ReadAddress(3'h0, ReadData);
   if (Size >= 126) begin
   assert(ReadData[4] == 1'b1)
     $display("PASS: VerifyOverflowTransmit:: TX_Full flag asserted");
   else begin
     $display("FAIL: VerifyOverflowTransmit:: TX_Full flag not asserted");
   end
   end
```

2.16 Coverage

Covergroups for the register, RX and TX registers were created.Coverpoints for the following were created in the cover groups accordingly.Address,WrEnable,RdEnable,DataIn,DataOut. Crosses were made on Address with WrEnable, RdEnable,DataIn DataOut.

Some bins were uncovered which resulted in a total coverage of 80.24%. If more stimuli was generated we could have achieved a much higher coverage.