HDLC Controller Design

ECE 551 Final Project Report

Jeng-Liang Tsai
Jie Zhang
Jui-Ning Cheng
Suniti Avadhesh Mandelia

12/12/2001

INDEX

1 Introduction

In this project we are to implement a high-level data link controller (HDLC) using verilog HDL. Layer two of the OSI model is the data link layer, and the most commonly used protocol is the HDLC protocol. The protocol provide reliable data link to the application by using zero-insertion and frame check sequence. Below is the basic frame structure.

7	E	DATA Bits	2	or 4	CRC	Byte	7	E

2 Block Diagram

The implementation is separated into two parts, the transmitter and the receiver. Figure 1-1 is the design hierarchy of the transmitter and Figure 1-2 through Figure 1-8 gives more detail about the transmitter. Figure 2-1 gives the design hierarchy of the receiver and Figure 2-2 through Figure 2-7 gives more detail about the receiver.

2.1 Transmitter

2.1.1 Design Hierarchy

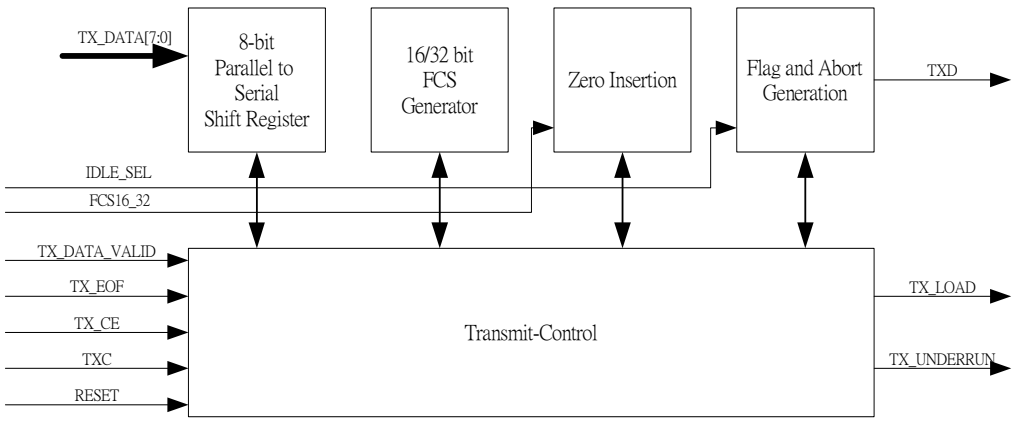


Figure 1-1

2.1.2 8-bit Parallel to Serial Shift Register

Figure 1-2 shows the 8-bit Parallel to Serial Shift Register. When the control unit detects a new data, it asserts LOAD and the 8-bit register load the data into the register. After it loads the data, it shifts the content at every clock cycle and fill in a 0 into the MSB, and the output is connected to the FCS generator through S_DATA pin.

When the transmitter finishes transmitting a frame, or aborts a frame because the data is not valid, the control unit will not assert the LOAD signal. Because we fill in 0s into the MSB every time when we do the shift operation, the content of the registers will be reset to zero after translating every byte, thus no reset is needed when we reinitialize another transfer. The same argument applies to the FCS generation circuit as well.

Note that we use a gated clock, G_TXC, as the clock signal. This is because when the zero-insertion circuit performs an insertion, the prior stages should be stalled by one clock cycle. Using G_TXC can reduce the control complexity as well as power consumption. Because the insertion signal used to generate the gated clock is synchronized, it will guarantee correct function.

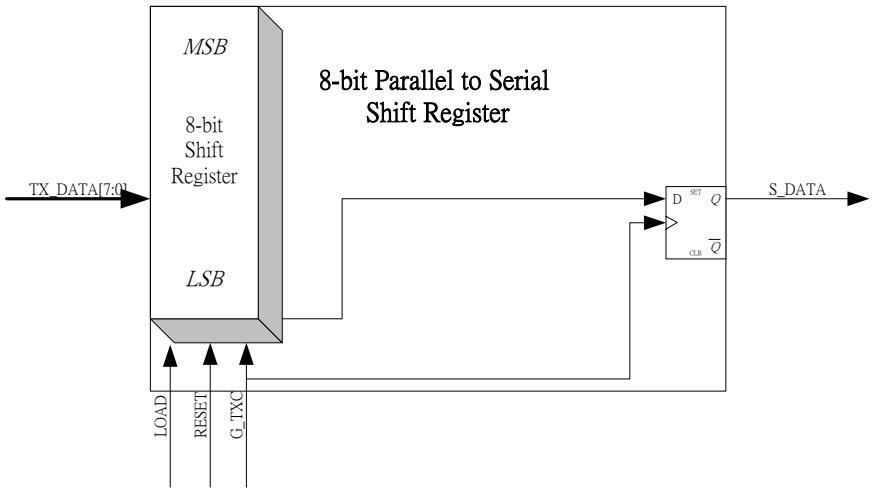
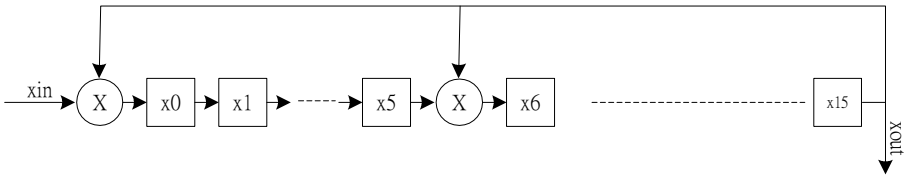


Figure 1-2

2.1.3 16/32 bit FCS Generator

Figure 1-3 shows the 16/32-bit FCS Generator. We use the shift register with distributed XOR operation to generate the FCS. The structure of the 16-bit FCS circuit is as follow.



There are two FCS generators, and only one of them is active at a given time. Control unit calculate the timing carefully and assert the SEND_FCS signal to start sending out the FCS. A counter is used to calculate when the transfer is finished. The counter is triggered by the gated clock signal G_TXC so that it will not inform the control unit too early if the complemented FCS contains consecutive 1s and triggers zero-insertion operation.

While sending FCS to the next stage, the FCS Generators do not perform XOR operation. It serves as a shift register when SEND_FCS is asserted.

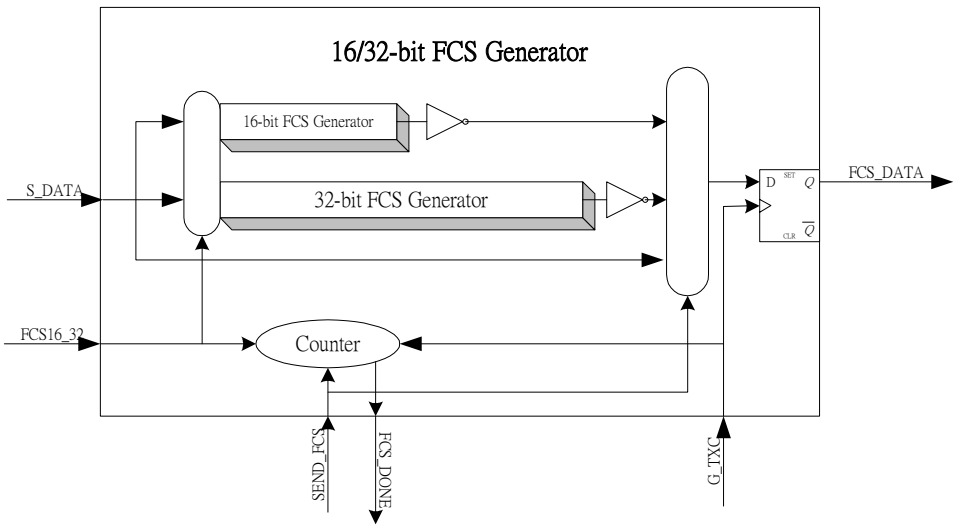


Figure 1-3

2.1.4 Zero Insertion

Figure 1-4 shows the zero-insertion circuit. We use 4-bit shift register to record the bit history. When the incoming bit and the 4 bits in the register are all 1s, we will stuff a 0 into the bit stream next clock cycle. Thus we inform the control unit through STUFF signal.

Zero insertion block and the following Flag/Abort Circuit do not need to stall, thus the clock comes directly from TXC.

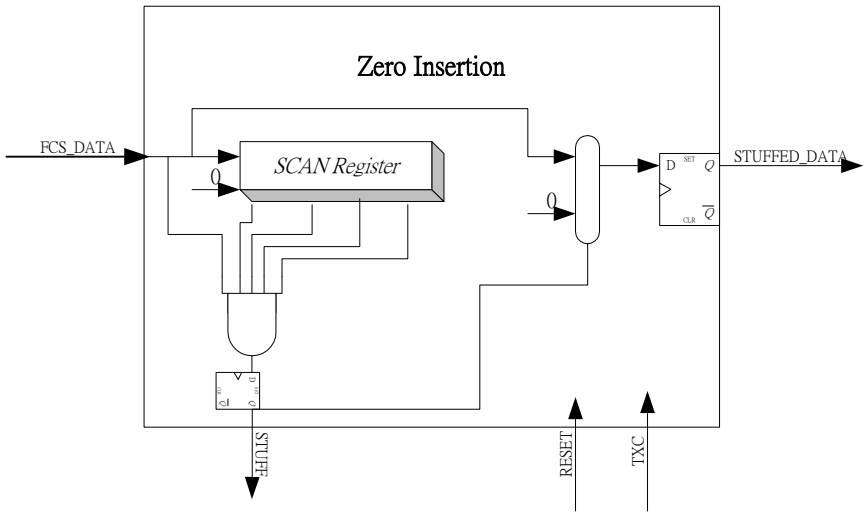


Figure 1-4

2.1.5 Flag and Abort Generation

Figure 1-5 shows the Flag and Abort Generation circuit. Figure 1-6 shows the state diagram of the Flag and Abort unit.

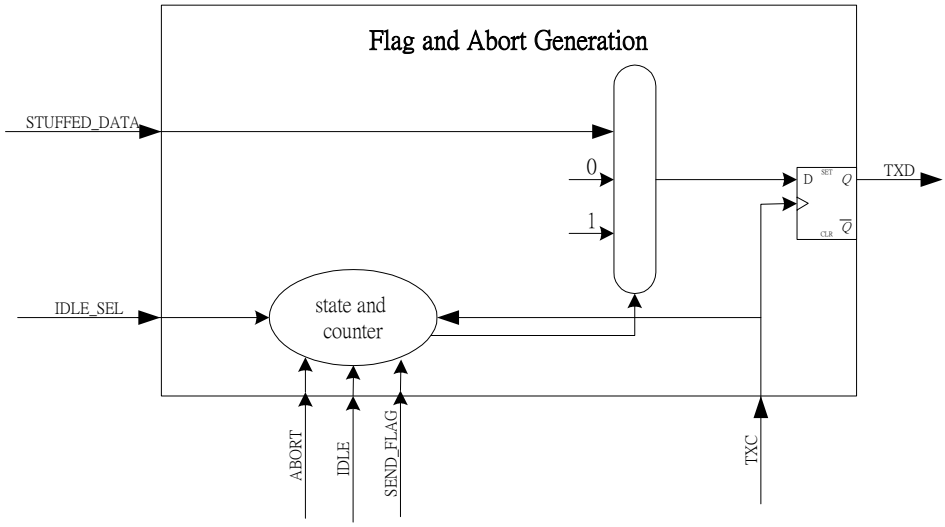


Figure 1-5

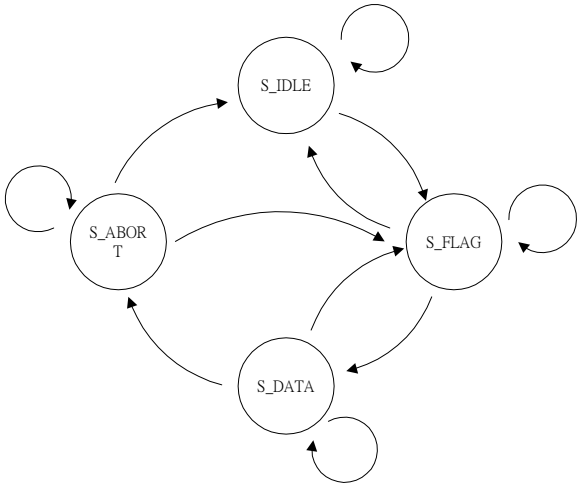


Figure 1-6

2.1.6 Transmit Control

Figure 1-7 shows the Transmit Control circuit. Figure 1-8 shows the state diagram of the Transmit Control unit.

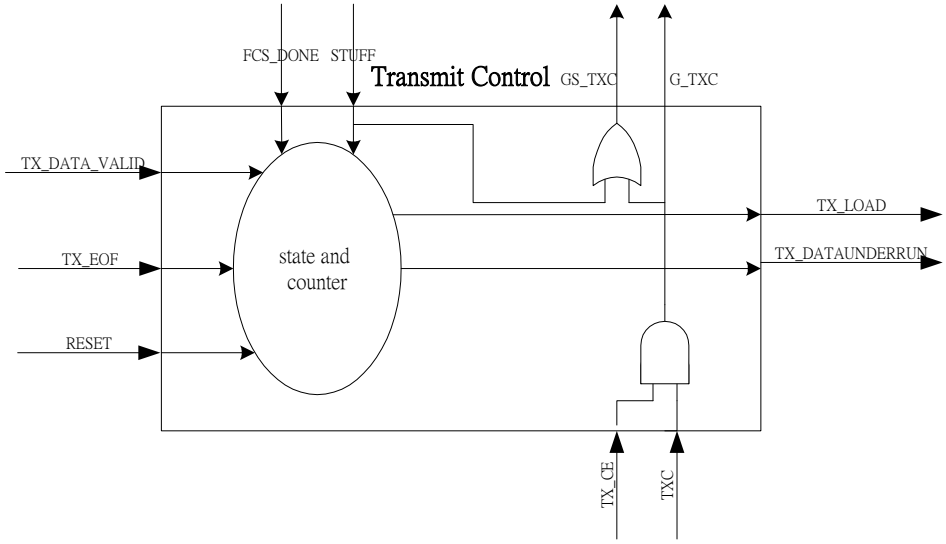


Figure 1-7

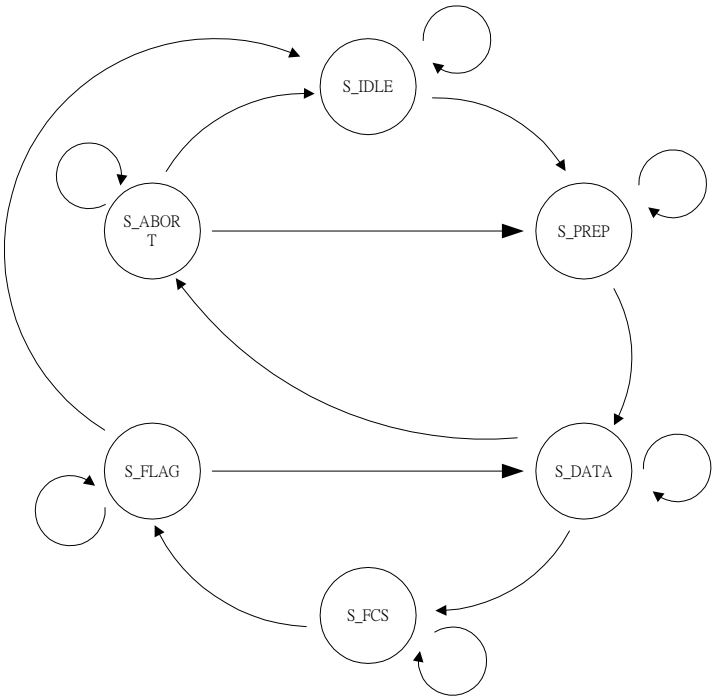


Figure 1-8

2.2 Receiver

2.2.1 Design Hierarchy

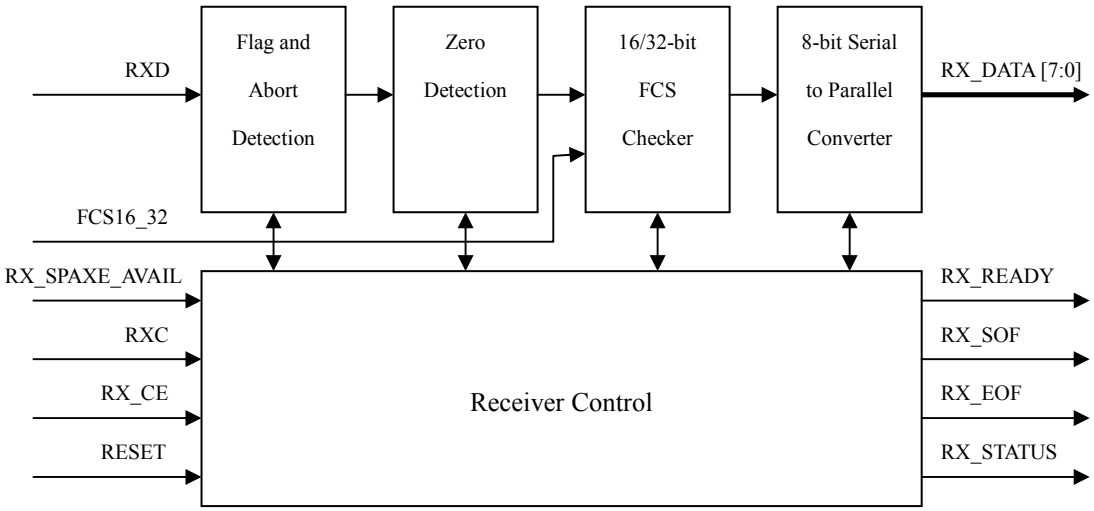


Figure 2-1

2.2.2 Flag and Abort Detection

Figure 2-2 shows the Flag and Abort Detection circuit. It compares the input and the register content with the flag and abort pattern every clock cycle, and report the detection through FLAG and ABORT signal to control unit.

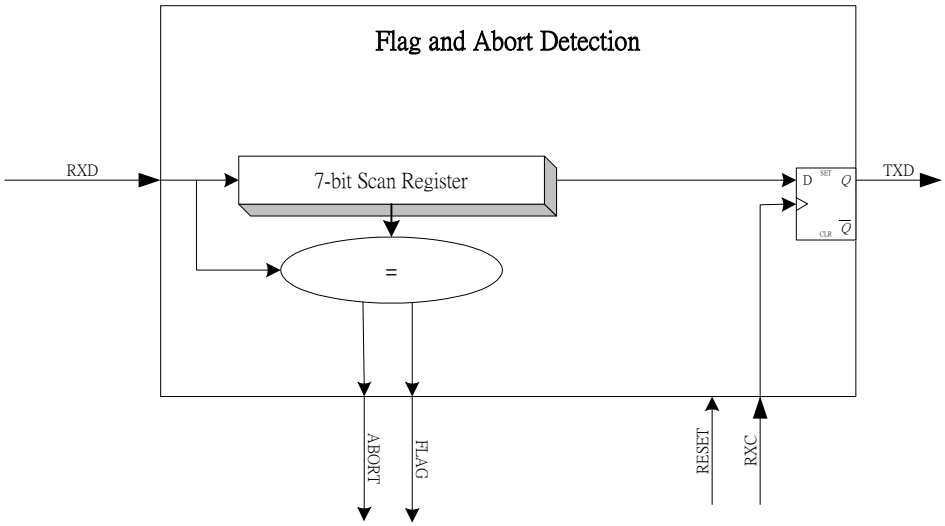


Figure 2-2

2.2.3 Zero Detection

Figure 2-3 shows the Zero Detection circuit. When five consecutive 1s are received, the ZERO signal is asserted to notify the control unit that the coming data is an inserted zero. Control unit then gated the clock of FCS checker and serial-to-parallel unit to ensure correct output. The PASS signal is used to pass a 0 to the next stage during IDLE state.

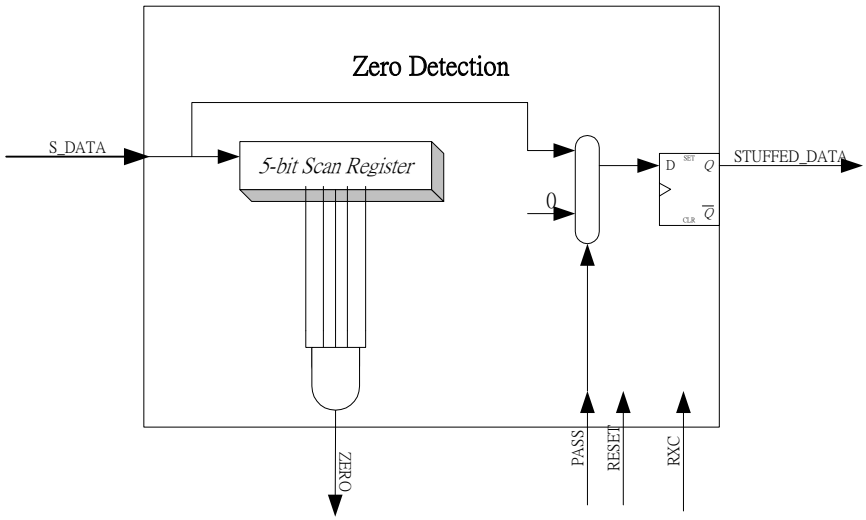


Figure 2-3

2.2.4 16/32 bit FCS Checker

Figure 2-4 shows the FCS Checker unit. The unit consists of a 32-bit shift register, a 16-bit FCS generator and a 32-bit FCS generator. The data is shifted into the register from zero-detection stage, and the 16^{th} bit is shifted into 16-bit FCS generator if FCS16_32 is 0. Otherwise the 32^{nd} bit is shifted into 32-bit FCS generator.

If an inserted zero is detected in prior stage, the control unit gated the clock for one cycle and prevents the FCS data from corrupted.

The circuit compares the values of FCS generator and scan register every clock cycle and send the result to control unit through FCS_ERROR signal. The control unit has to sample the result at the right time to get the correct result.

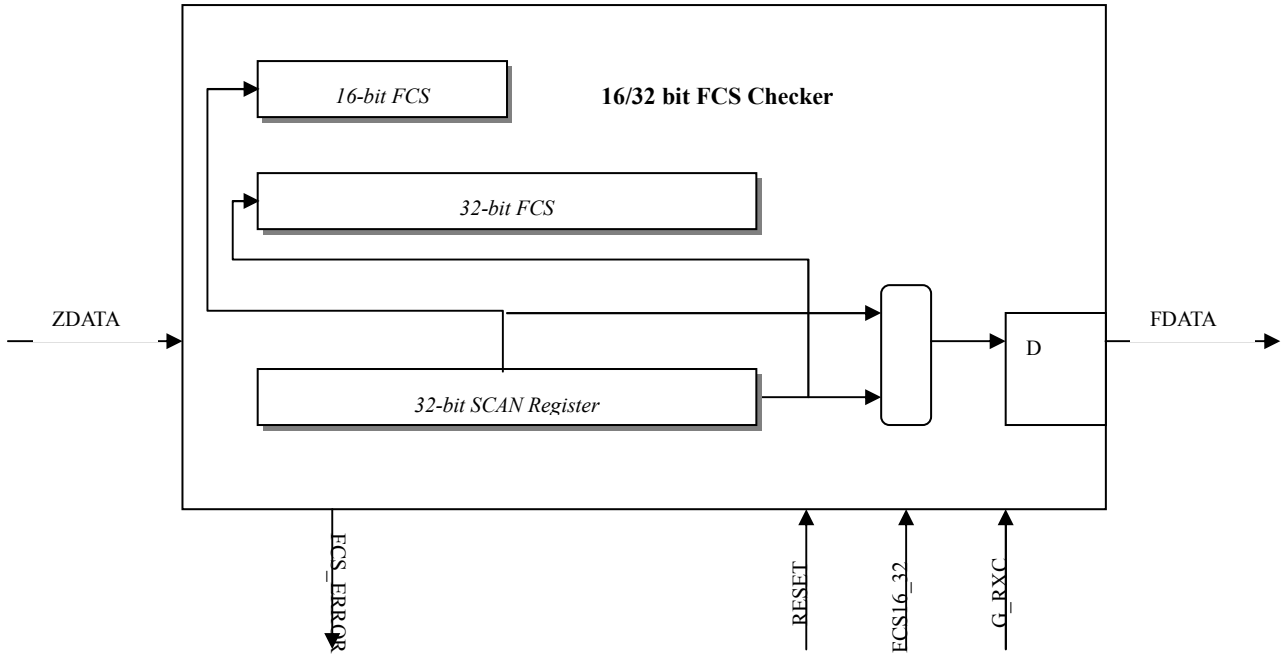


Figure 2-4

2.2.5 8-bit Serial to Parallel Converter

Figure 2-5 shows the 8-bit Serial to Parallel Converter. The converter has its own counter and loads a byte to the output every 8 clock cycles. When it receives a LOAD signal, it loads the status signals to the output instead. If there are remaining data in the internal shift register, an octet error is detected and the corresponding status bit is set.

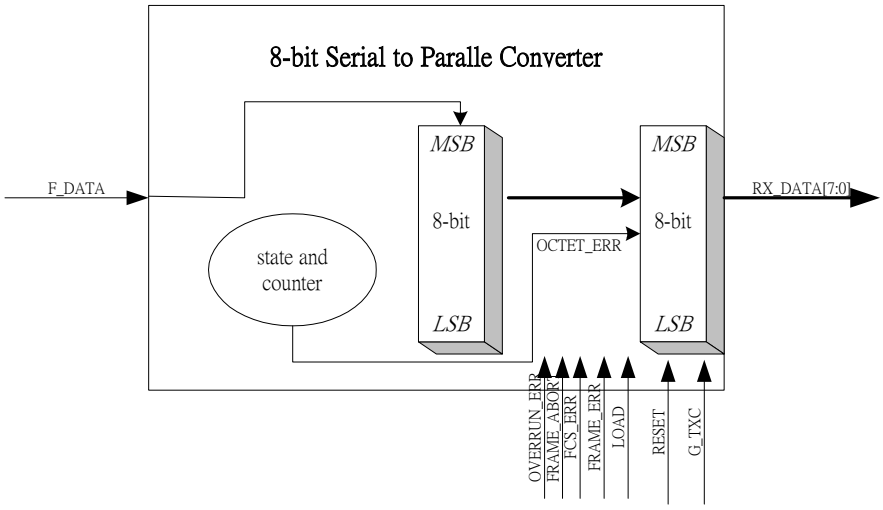


Figure 2-5

2.2.6 Receive Control

Figure 2-6 shows the receive control. Figure 2-7 shows the state machine of the control unit.

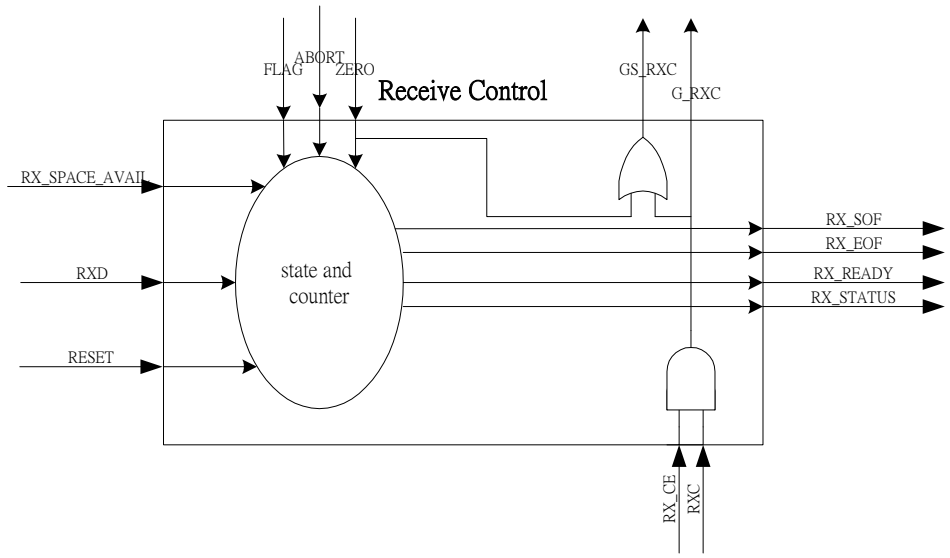


Figure 2-6

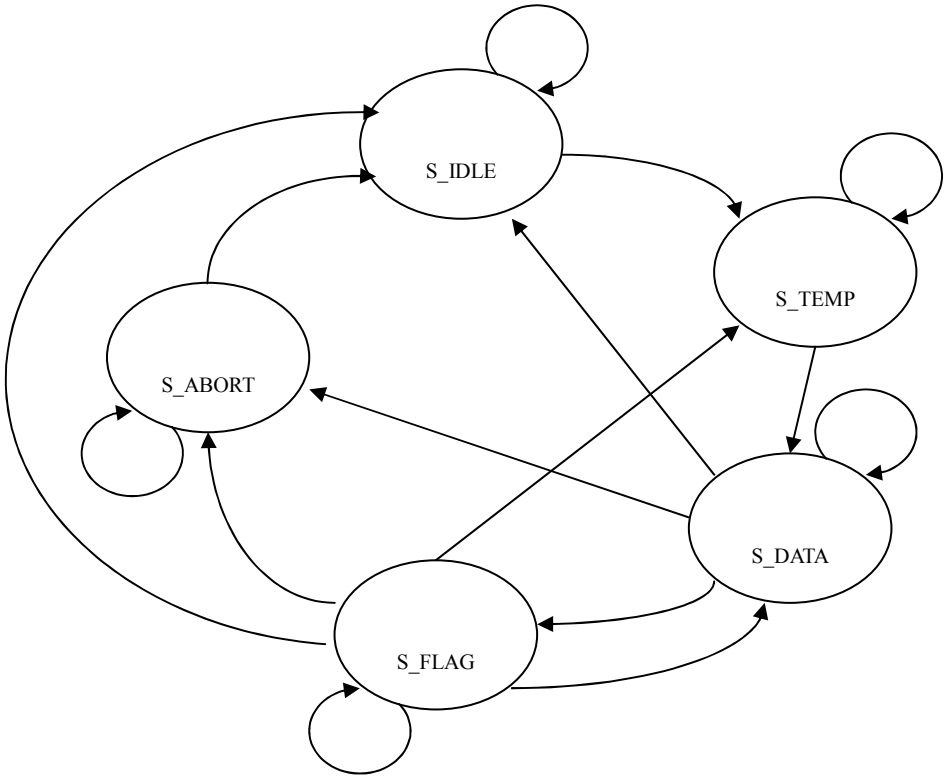


Figure 2-7

3 Verilog Code

The final version of the Verilog code we use for both simulation and synthesis are given below.

3.1 Transmitter

3.1.1 Design Hierarchy

```
`timescale 1ns/100ps

module Transmitter(TXD, TX_LOAD, TX_UNDERRUN, TX_DATA, TX_DATA_VALID, TX_EOF, IDLE_SEL,
FCS16_32, RESET, TX_CE, TXC);
     output TXD, TX_LOAD, TX_UNDERRUN;
     input [7:0] TX_DATA;
     input TX_DATA_VALID, TX_EOF, IDLE_SEL, FCS16_32, RESET, TX_CE, TXC;

     Parallel_Serial ps(S_DATA, TX_DATA, LOAD, STALL, B_RESET, G_TXC);

     FCS_Generator fg(FCS_DATA, FCS_DONE, S_DATA, SEND_FCS, FCS16_32, B_RESET, GS_TXC);

     Zero_Insertion zi(STUFFED_DATA, STUFF, FCS_DATA, B_RESET, G_TXC);

     Flag_Abort fa(TXD, STUFFED_DATA, IDLE_SEL, IDLE, SEND_FLAG, ABORT, RESET, G_TXC);

     Transmit_Control tc(TX_LOAD, TX_UNDERRUN,
          LOAD, STALL, SEND_FCS, SEND_FLAG, IDLE, ABORT, GS_TXC, G_TXC, B_RESET, // Internal
output
          STUFF, FCS_DONE, // Internal input
          TX_DATA_VALID, TX_EOF, IDLE_SEL, RESET, TX_CE, TXC);

endmodule
```

3.1.2 8-bit Parallel to Serial Shift Register

```
`timescale 1ns/100ps

module Parallel_Serial(S_DATA, TX_DATA, LOAD, STALL, RESET, TXC);
     output S_DATA;
     input [7:0] TX_DATA;
     input LOAD, STALL, RESET, TXC;

     reg [7:0] DATA;
     reg S_DATA;

     always@(posedge TXC)
     begin
          if(RESET)
          begin
               S_DATA <= #3 0;
               DATA <= #3 0;
          end

          else if(~STALL)
          begin
               S_DATA <= #3 DATA[0];
               if(LOAD)
                    DATA <= #3 TX_DATA;
               else
                    DATA <= #3 (DATA >> 1);
          end
```

```
     end

     specify

       ( TX_DATA, LOAD, STALL, RESET, TXC *> S_DATA ) = 3 ;

     endspecify

endmodule
```

3.1.3 16/32 bit FCS Generator

```
`timescale 1ns/100ps

`define P16 16'b0001_0000_0010_0001
`define P32 32'b0000_0100_1100_0001_0001_1101_1011_0111
`define S_DATA 1'b0
`define S_FCS  1'b1

module FCS_Generator(FCS_DATA, FCS_DONE, S_DATA, SEND_FCS, FCS16_32, RESET, TXC);
     output FCS_DATA, FCS_DONE;
     input S_DATA, SEND_FCS, FCS16_32, RESET, TXC;

     reg [15:0] R16;
     reg [31:0] R32;
     reg [4:0] counter;
     reg done, FCS_DATA, FCS_DONE, state;


     always@(posedge TXC)
     begin

          if(RESET)
          begin
               R16 <= #3 0;
               R32 <= #3 0;
               counter <= #3 0; done <= #3 0;
               FCS_DATA <= #3 0; FCS_DONE <= #3 0; state <= #3 `S_DATA;
          end

          else
          begin
               casex({state,FCS16_32})
                    2'b11:
                         begin
                              counter <= #3 counter +1;
                              {state, done} <= #3 (counter == 31) ? {`S_DATA, 1'b1} :
                                    {`S_FCS, 1'b0};
                              R32 <= #3 { R32[30:0], 1'b0};
                              FCS_DATA <= #3 ~R32[31];
                         end
                    2'b10:
                         begin
                              counter <= #3 (counter == 15) ? 0 : counter +1;
                              {state, done} <= #3 (counter ==15) ? {`S_DATA, 1'b1} :
                                   {`S_FCS, 1'b0};
                              R16 <= #3 { R16[14:0], 1'b0};
                              FCS_DATA <= #3 ~R16[15];
                         end
                    2'b01:
                         begin
                              state <= #3 (SEND_FCS && ~(done || FCS_DONE)) ? `S_FCS :
                                    `S_DATA;
                              R32 <= #3 (R32[31]) ? { R32[30:0], S_DATA} ^ `P32 :
                                   { R32[30:0], S_DATA};
                              FCS_DATA <= #3 S_DATA;
                              {FCS_DONE, done} <= #3 {done, 1'b0};
                         end
                    2'b00:
                         begin
                              state <= #3 (SEND_FCS && ~(done || FCS_DONE)) ? `S_FCS :
                                    `S_DATA;
                              R16 <= #3 (R16[15]) ? { R16[14:0], S_DATA} ^ `P16 :
                                   { R16[14:0], S_DATA};
```

```
                              FCS_DATA <= #3 S_DATA;
                              {FCS_DONE, done} <= #3 {done, 1'b0};
                         end
                endcase
           end

     end

     specify

          ( S_DATA, SEND_FCS, FCS16_32, RESET, TXC *> FCS_DATA, FCS_DONE) = 3;

     endspecify

endmodule
```

3.1.4 Zero Insertion

```
`timescale 1ns/100ps

module Zero_Insertion(STUFFED_DATA, STUFF, FCS_DATA, RESET, TXC);
     output STUFFED_DATA, STUFF;
     input FCS_DATA, RESET, TXC;


     reg [3:0] SCAN;
     reg STUFF, STUFFED_DATA;

     always@(posedge TXC)
     begin
          if(RESET)
               {SCAN, STUFF, STUFFED_DATA} <= #3 3'b000;

          else
          begin
               STUFF <= #3 (STUFF) ? 0 : ({ FCS_DATA, SCAN} == 5'b11111 ) ? 1 : 0;
               SCAN <= #3 { SCAN[2:0], (STUFF) ? 1'b0 : FCS_DATA };
               STUFFED_DATA <=     #3 (STUFF) ? 0 : FCS_DATA;
          end
     end

     specify

          ( FCS_DATA, RESET, TXC *> STUFFED_DATA, STUFF) = 3;

     endspecify

endmodule
```

3.1.5 Flag and Abort Generation

```
`timescale 1ns/100ps

`define FLAG 8'b01111110
`define FILL 8'b11111111
`define S_IDLE 2'b00
`define S_FLAG 2'b01
`define S_DATA 2'b10
`define S_ABORT 2'b11

module Flag_Abort(TXD, STUFFED_DATA, IDLE_SEL, IDLE, SEND_FLAG, ABORT, RESET, TXC);
     output TXD;
     input STUFFED_DATA, IDLE_SEL, IDLE, SEND_FLAG, ABORT, RESET, TXC;

     reg [2:0] counter;
     reg TXD;
     reg [1:0] state;

     always@(posedge TXC)
     begin
          if(RESET)
```

```
          begin
               counter <= #3 0;
               TXD <= #3 0;
               state <= #3 `S_IDLE;
          end

          else
          begin
               case(state)
                    `S_IDLE:
                    begin
                         TXD <= #3 (SEND_FLAG) ? 1'b0 : (IDLE_SEL) ? 1'b1 :
                              (counter ==0 || counter == 3'b111) ? 1'b0 : 1'b1;
                         if(IDLE_SEL)
                         begin
                              state <= #3 (SEND_FLAG) ? `S_FLAG : `S_IDLE;
                              counter <= #3 (SEND_FLAG) ? 1 : counter +1;
                         end
                         else
                         begin
                              state <= #3 (counter== 3'b111 && ~IDLE) ? `S_DATA : `S_IDLE;
                              counter <= #3 counter + 1;
                         end
                    end
                    `S_FLAG:
                    begin
                         TXD <= #3 (counter ==0 || counter == 3'b111) ? 0 : 1;

                         state <= #3 (counter == 3'b111)    ? (IDLE) ? `S_IDLE : `S_DATA :
                               `S_FLAG;
                         counter <= #3 counter +1;
                    end
                    `S_DATA:
                    begin
                         TXD <= #3 (SEND_FLAG) ? 0 : STUFFED_DATA;

                         state <= #3 (SEND_FLAG) ? `S_FLAG : (ABORT)  ? `S_ABORT :
                               `S_DATA;
                         counter <= #3 (SEND_FLAG) ? 1 : counter;
                    end
                    `S_ABORT:
                    begin
                         TXD <= #3 (SEND_FLAG) ? 0 : 1;

                         state <= #3 (counter == 3'b111) ? (SEND_FLAG) ? `S_FLAG : `S_IDLE :
                               `S_ABORT;
                         counter <= #3 (SEND_FLAG) ? 1 : counter +1;
                    end
               endcase
          end
     end


     specify

          ( STUFFED_DATA, IDLE_SEL, IDLE, SEND_FLAG, RESET, TXC *> TXD ) = 3 ;

     endspecify

endmodule
```

3.1.6 Transmit Control

```
`timescale 1ns/100ps

`define S_IDLE  6'b000001
`define S_PREP  6'b000010
`define S_DATA  6'b000100
`define S_FCS   6'b001000
`define S_FLAG  6'b010000
`define S_ABORT 6'b100000
`define B_DELAY 3

module Transmit_Control(TX_LOAD, TX_UNDERRUN,
```

```
     LOAD, STALL, SEND_FCS, SEND_FLAG, IDLE, ABORT, GS_TXC, G_TXC, B_RESET, // Internal
     STUFF, FCS_DONE, // Internal input
     TX_DATA_VALID, TX_EOF, IDLE_SEL, RESET, TX_CE, TXC);

output TX_LOAD, TX_UNDERRUN;
output LOAD, STALL, SEND_FCS, SEND_FLAG, IDLE, ABORT, GS_TXC, G_TXC, B_RESET;
input STUFF, FCS_DONE;
input TX_DATA_VALID, TX_EOF, IDLE_SEL, RESET, TX_CE, TXC;

reg TX_LOAD, TX_UNDERRUN;
reg LOAD, STALL, SEND_FCS, SEND_FLAG, IDLE, ABORT;
reg eof, safe, txce;
reg [5:0] state;
reg [2:0] counter;

assign GS_TXC = TXC | STUFF | ~txce;
assign G_TXC = TXC | ~txce;
assign B_RESET = RESET | ABORT;

always@(posedge TXC)
begin
     txce <= #3 TX_CE;
     if(RESET)
     begin
          TX_LOAD <= #3 0; TX_UNDERRUN <= #3 0;
          LOAD <= #3 0; STALL <= #3 0;
          SEND_FLAG <= #3 0; SEND_FCS <= #3 0;
          IDLE <= #3 1; ABORT <= #3 0;
          state <= #3 `S_IDLE;
          eof <= #3 0; safe <= #3 0;
          counter <= #3 0;
     end

     else if(txce) // Output Logic
     begin
          STALL <= #3 STUFF;
          case(state)
               `S_IDLE:
                    begin
                         if(IDLE_SEL)
                         begin
                              safe <= #3 (counter == 3'b110) ? 1 : 0;
                              {state, counter} <= #3 (TX_DATA_VALID && safe)
                                   ? {`S_PREP, 3'b000} : {`S_IDLE, counter+3'b001} ;
                              {SEND_FLAG, IDLE} <= #3 (TX_DATA_VALID && safe) ?
                                   2'b10 : 2'b01;
                         end

                         else
                         begin
                              {state, counter} <= #3 ( TX_DATA_VALID &&
                                   counter==`B_DELAY)? {`S_DATA, 3'b000} : {`S_IDLE,
                                   counter+3'b001};
                              {IDLE, LOAD} <= #3 (TX_DATA_VALID &&
                                   counter==`B_DELAY) ? 2'b01 : 2'b10;
                         end
                    end

               `S_PREP:
               begin
                    state <= #3 (counter == `B_DELAY) ? `S_DATA : `S_PREP;
                    counter <= #3 (counter == `B_DELAY) ? 0 : counter + 1;
                    SEND_FLAG <= #3 0;
                    if(counter == `B_DELAY)
                         LOAD <= #3 1;
                    if(LOAD)
                         {LOAD, TX_LOAD} <= #3 2'b01;
                    if(TX_LOAD)
                         TX_LOAD <= #3 0;
               end

               `S_DATA:
               begin
                    if(counter == 7 && eof)
                    begin
                         eof <= #3 0;
                         state <= #3 `S_FCS;
```

```
                    counter <= #3 (STUFF) ? counter : counter + 1;
               end

               else if(counter == 7 && ~eof)
               begin
                    state <= #3 (TX_DATA_VALID) ? `S_DATA : `S_ABORT;
                    counter <= #3 (STUFF) ? counter : counter + 1;
                    if(~TX_DATA_VALID)
                         counter <= #3 0;
                    eof <= #3 (TX_EOF && TX_DATA_VALID && ~STUFF) ? 1 : 0;
                    {LOAD, TX_UNDERRUN, ABORT} <= #3 (TX_DATA_VALID) ?
                         3'b100 : 3'b011;
               end

               else
                    counter <= #3 (STUFF) ? counter : counter + 1;

               SEND_FLAG <= #3 0;
               if(counter == 0 && LOAD == 1)
                    {LOAD, TX_LOAD} <= #3 (STUFF) ? 2'b10 : 2'b01;
               if(TX_LOAD ==1)
                    TX_LOAD <= #3 0;
          end

          `S_ABORT:
               begin
                    state <= #3 (counter == 7) ? ( (TX_DATA_VALID) ? `S_PREP :
                         `S_IDLE ) : `S_ABORT;
                    counter <= #3 counter + 1;

                    ABORT <= #3 (counter == 7) ? 1 : 0;
                    SEND_FLAG <= #3 (TX_DATA_VALID && counter == 7) ? 1 : 0;
                    IDLE <= #3 (~TX_DATA_VALID && counter ==7)? 1 : 0;
                    TX_UNDERRUN <= #3 0;
               end

          `S_FCS:
          begin
               if(SEND_FCS ==0)
                    counter <= #3 (STUFF) ? counter : counter + 1;
               else
                    {state, counter} <= #3 (FCS_DONE && ~STUFF) ? {`S_FLAG,
                         3'b000} : {`S_FCS, counter};

               if(counter == 0)
                    SEND_FCS <= #3 1;
               if(counter == 1)
                    {SEND_FCS, SEND_FLAG} <= #3 (FCS_DONE && ~STUFF) ?
                         2'b01 : 2'b10;
          end

          `S_FLAG:
          begin
               state <= #3 (counter == `B_DELAY && TX_DATA_VALID) ? `S_DATA :
                    (counter == 7) ? `S_IDLE : `S_FLAG;
               counter <= #3 (counter == `B_DELAY && TX_DATA_VALID) ? 0 :
                    counter + 1;

               SEND_FLAG <= #3 0;
               if(counter == 3 && TX_DATA_VALID)
                    LOAD <= #3 1;
               if(counter == 4 && ~LOAD)
                    IDLE <= #3 1;
          end

     endcase

   end
end


endmodule
```

3.2 Receiver

3.2.1 Design Hierarchy

```
`timescale 1ns/100ps

module Receiver (RX_DATA, RX_READY, RX_SOF, RX_EOF, RX_STATUS, RXD, FCS16_32,
RX_SPACE_AVAIL, RESET, RX_CE, RXC);

output [7:0] RX_DATA;
output RX_READY, RX_SOF, RX_EOF, RX_STATUS;
input RXD, FCS16_32, RX_SPACE_AVAIL, RXC, RX_CE, RESET;

flag_abort_detect fa (SDATA, FLAG, ABORT, RXD, RESET, RXC);

zero_detection zd (ZDATA, ZERO, SDATA, PASS, B_RESET, RXC);

FCS_checker fcs (FDATA, FCS_ERROR, ZDATA, FCS16_32, B_RESET, G_RXC);

serial_parallel sp (RX_DATA, FDATA, FCS_ERR, FRAME_ERR, FRAME_ABORT, OCTET_ERR, OVERRUN_ERR,
LOAD, B_RESET, G_RXC);

receiver_control rc (RX_READY, RX_SOF, RX_EOF, RX_STATUS, FCS_ERR, FRAME_ERR, FRAME_ABORT,
OCTET_ERR, OVERRUN_ERR, LOAD, PASS, B_RESET, G_RXC, FLAG, ABORT, ZERO, FCS_ERROR,
RX_SPACE_AVAIL, FCS16_32, RESET, RX_CE, RXC);

endmodule
```

3.2.2 Flag and Abort Detection

```
`timescale 1ns/100ps

module flag_abort_detect(SDATA, FLAG, ABORT, RX_DATA, RESET, RXC);

output SDATA, FLAG, ABORT;
input RX_DATA, RESET, RXC;

reg SDATA, FLAG, ABORT;
reg [6:0] buffer;

always@(posedge RXC)

begin
     if (RESET)
          begin
               buffer <= #3 7'b0000000;
               SDATA <= #3 0;
               FLAG <= #3 0;
               ABORT <= #3 0;
          end

     else
          begin
               SDATA <= #3 buffer[6];
               ABORT <= #3 ( {buffer, RX_DATA} == 8'b11111111) ? 1 : 0;
               FLAG <= #3 ( {buffer, RX_DATA} == 8'b01111110 ) ? 1 : 0;
               buffer <= #3 { buffer[5:0], RX_DATA};
          end
     end
endmodule
```

3.2.3 Zero Detection

```
`timescale 1ns/100ps

module zero_detection (ZDATA, ZERO, SDATA, PASS, RESET, RXC);

input SDATA, PASS, RESET, RXC;
output ZDATA, ZERO;
reg [4:0] buffer;
reg ZDATA, ZERO;

always @ (posedge RXC)
begin
     if(RESET)
          begin
               ZERO <= #3 0;
               ZDATA <= #3 0;
               buffer <= #3 5'b00000;
          end

     else
          begin
               buffer <= #3 {buffer[3:0], SDATA};
               ZDATA <= #3 (PASS) ? SDATA : 0;
               ZERO <= #3 ( buffer == 5'b11111 ) ? 1 : 0;
          end
end
endmodule
```

3.2.4 16/32 bit FCS Checker

```
`timescale 1ns/100ps

`define P16 16'b0001_0000_0010_0001
`define P32 32'b0000_0100_1100_0001_0001_1101_1011_0111

module FCS_checker (FDATA, FCS_ERROR, ZDATA, FCS16_32, RESET, G_RXC);
     output FDATA, FCS_ERROR;
     input ZDATA, FCS16_32, RESET, G_RXC;
     reg [15:0] R16;
     reg [31:0] buffer, R32;
     reg FDATA, FCS_ERROR;


always@(posedge G_RXC)
begin
     if(RESET)
     begin
          R16 <= #3 16'h0000;
          R32 <= #3 32'h00000000;
          buffer <= #3 32'h00000000;
          FDATA <= #3 0;
          FCS_ERROR <= #3 0;
     end

     else
     begin

          buffer <= #3 {buffer[30:0], ZDATA} ;
          FDATA <= #3 (FCS16_32) ? buffer[31] : buffer[15];
          if(FCS16_32)
          begin
               R32 <= #3 (R32[31]) ? { R32[30:0], buffer[31]} ^ `P32 : { R32[30:0],
                    buffer[31]};
               FCS_ERROR <= #3 ~&(buffer ^ R32);
          end

          else
          begin
               R16 <= #3 (R16[15]) ? { R16[14:0], buffer[15]} ^ `P16 : { R16[14:0],
```

```
                    buffer[15]};
               FCS_ERROR <= #3 ~&(buffer[15:0] ^ R16);
          end
     end
end
endmodule
```

3.2.5 8-bit Serial to Parallel Converter

```
`timescale 1ns/100ps

`define B_DELAY 1

module serial_parallel (PDATA, FDATA, FCS_ERR, FRAME_ERR, FRAME_ABORT, OCTET_ERR,
OVERRUN_ERR,
                                                    LOAD, RESET, G_RXC);

output [7:0] PDATA;
input FDATA, FCS_ERR, FRAME_ERR, FRAME_ABORT, OCTET_ERR, OVERRUN_ERR, LOAD, RESET, G_RXC;

reg [2:0] counter;
reg [7:0] PDATA, buffer;

always @ (posedge G_RXC)
begin
     if (RESET)
     begin
          counter <= 3'b000;
          buffer <= #3 8'h00;
    PDATA <= #3 8'h00;
     end

     else if(LOAD)
          PDATA <= #3 {3'b000, OVERRUN_ERR, OCTET_ERR , FRAME_ABORT, FRAME_ERR, FCS_ERR};

     else
     begin
          buffer <= #3 {FDATA, buffer[7:1]};
          counter <= #3 counter +1;
          if(counter==`B_DELAY)
               PDATA <= #3 buffer;
  end
end

endmodule
```

3.2.6 Receive Control

```
`timescale 1ns/100ps
`define S_IDLE   5'b00001
`define S_TEMP   5'b00010
`define S_DATA   5'b00100
`define S_ABORT  5'b01000
`define S_FLAG   5'b10000
`define B_DELAY  1

module receiver_control(RX_READY, RX_SOF, RX_EOF, RX_STATUS, FCS_ERR, FRAME_ERR,
FRAME_ABORT, OCTET_ERR, OVERRUN_ERR, LOAD, PASS, B_RESET, G_RXC, FLAG, ABORT, ZERO,
FCS_ERROR, RX_SPACE_AVAIL, FCS16_32, RESET, RX_CE, RXC);

output RX_READY, RX_SOF, RX_EOF, RX_STATUS, FCS_ERR, FRAME_ERR, FRAME_ABORT, OCTET_ERR,
OVERRUN_ERR, LOAD, PASS, B_RESET, G_RXC;
input FLAG, ABORT, ZERO, FCS_ERROR, RX_SPACE_AVAIL, FCS16_32, RESET, RX_CE, RXC;

reg  RX_READY, RX_SOF, RX_EOF, RX_STATUS, FCS_ERR, FRAME_ERR, FRAME_ABORT, OCTET_ERR,
OVERRUN_ERR, LOAD, PASS, CLEAN;
reg [5:0] state;
reg [2:0] counter;
reg [6:0] f_counter;
reg octet_check, sof, abort_sent, rxce;
```

```
wire G_RXC;

assign G_RXC = RXC | ~rxce | ZERO;
assign B_RESET = RESET | CLEAN;

always@(posedge RXC)
begin
     rxce <= #3 RX_CE;
     if (RESET)
          begin
               RX_READY <= #3 0;
               RX_SOF <= #3 0; RX_EOF <= #3 0;
               RX_STATUS <= #3 0;
               LOAD <= #3 0; PASS <= #3 0; CLEAN <= #3 1;
               FCS_ERR <= #3 0; FRAME_ERR <= #3 0;
               FRAME_ABORT <= #3 0;
               OCTET_ERR <= #3 0;
               OVERRUN_ERR <= #3 0;
               state <= #3 `S_IDLE;
               counter <= #3 3'b000;
               f_counter <= #3 7'b0000000;
               octet_check <= #3 0; sof <= #3 0;
               abort_sent <= #3 0;
          end

     else if (rxce) // Output Logic
          begin
               case(state)
                    `S_IDLE:
                    begin
                         counter <= #3 0;
                         f_counter <= #3 0;
                         abort_sent <= #3 0;
                         state <= #3 (FLAG) ? `S_TEMP : `S_IDLE;

                         {PASS, CLEAN} <= #3 2'b01;
                    end

                    `S_TEMP:
                    begin
                         counter <= #3 counter + 1;
                         state <= #3 (counter == 7 && ~FLAG) ? `S_DATA : `S_TEMP;
                         f_counter <= #3 (counter == 7 && ~FLAG) ? 7'b0001000 : 7'b0000000 ;

                         {PASS, CLEAN} <= #3 (counter == 7 && ~FLAG) ? 2'b10 : 2'b01;
                    end

                    `S_DATA:
                    begin
                         counter <= #3 (ZERO) ? counter : counter + 1;
                         f_counter <= #3 (f_counter == 7'b1111111) ? 7'b1111111 : ( (ZERO) ?
                              f_counter : f_counter + 1);
                         state <= #3 ((FLAG || ABORT) && ((FCS16_32 && f_counter<=48) ||
                              (~FCS16_32 && f_counter<=32))) ? `S_IDLE :
                               ((FLAG) ? `S_FLAG : ((ABORT) ? `S_ABORT : `S_DATA));

                         if( (counter == `B_DELAY) && ((FCS16_32 && f_counter > 48) ||
                              (~FCS16_32 && f_counter > 32)))
                         begin
                              if(RX_SPACE_AVAIL)
                              begin
                                   RX_READY <= #3 1;
                                   RX_SOF <= #3 ( (FCS16_32 && f_counter==(48+`B_DELAY))
                                   || (~FCS16_32 && f_counter==(32+`B_DELAY)) ) ? 1 : 0;
                              end
                              else
                                   OVERRUN_ERR <= #3 1;
                         end
                         FRAME_ABORT <= #3 (ABORT) ? 1 : 0;
                         if(RX_READY)
                              {RX_READY, RX_SOF} <= #3 2'b00;
                    end

                    `S_ABORT:
                    begin
                         if(abort_sent)
                         begin
```

```
                              counter <= #3 counter + 1;
                              state <= #3 (counter ==2) ? `S_IDLE : `S_ABORT;

                              if(counter == 0)
                                   LOAD <= #3 1;
                              if(counter == 1)
                                   {LOAD, RX_READY, RX_STATUS} <= #3 3'b011;
                              if(counter == 2)
                                   {RX_READY, RX_STATUS, FRAME_ABORT, OVERRUN_ERR} <=
                                        #3 4'b0000;
                         end
                         else
                         begin
                              abort_sent <= #3 1;
                              counter <= #3 0;
                         end
                    end

                    `S_FLAG:
                    begin
                         counter <= #3 (octet_check) ? counter + 1 : 1;
                         if(octet_check && counter==`B_DELAY && ~RX_SPACE_AVAIL)
                              state <= #3 `S_ABORT;

                         if(counter == 7 && octet_check)
                         begin
                              {state, octet_check} <= #3 (FLAG) ? {`S_TEMP, 1'b0} :
                                   (ABORT) ? {`S_IDLE, 1'b0} : {`S_DATA, 1'b0};
                              f_counter <= #3 (FLAG) ? 7'b0001000 : (ABORT) ?
                                   7'b0000000 : 7'b0001000;
                         end
                         else f_counter <= #3 (f_counter == 7'b1111111) ? 7'b1111111 :
                                   f_counter + 1;

                         if(~octet_check)
                         begin
                              octet_check <= #3 1;
                              OCTET_ERR <= #3 (counter == 0) ? 0 : 1;
                              PASS <= #3 0;
                         end

                         else
                         begin
                              if(counter == (`B_DELAY))
                              begin
                                   if(RX_SPACE_AVAIL)
                                   begin
                                        {RX_READY, RX_EOF} <= #3 2'b11;
                                        FRAME_ERR <= #3 ( (FCS16_32 && f_counter < 96) ||
                                             (~FCS16_32 && f_counter < 64) ) ? 1 : 0;
                                        FCS_ERR <= #3 FCS_ERROR;
                                   end
                                   else
                                        OVERRUN_ERR <= #3 1;
                              end
                              if(counter == (`B_DELAY+1))
                                   {RX_READY, RX_EOF, LOAD} <= #3 3'b001;
                              if(counter == (`B_DELAY+2))
                                   {LOAD, RX_READY, RX_STATUS} <= #3 3'b011;
                              if(counter == (`B_DELAY+3))
                                   {RX_READY, RX_STATUS, CLEAN} <= #3 3'b001;
                              if(counter == 7)
                                   {PASS, CLEAN, FRAME_ERR, FCS_ERR} <= #3 4'b1000;
                         end
                    end
               endcase
          end
end


endmodule
```

4 Simulation Results

The testbench we use for simulation is based on the template that TA provides. We also add some statements to generate different situations.

In first part of the simulation, we change the first frame to contain 20 consecutive 1s and verify the zero-insertion/zero-detection function. We also set the second and the third frame as back-to-back frame. Both IDLE_SELECT modes are also tested, and all of the cases are verified with 16-bit FCS and 32-bit FCS.

In second part of the simulation, we stimulate various kinds of errors and proved that both transmitter and receiver can handle those errors correctly.

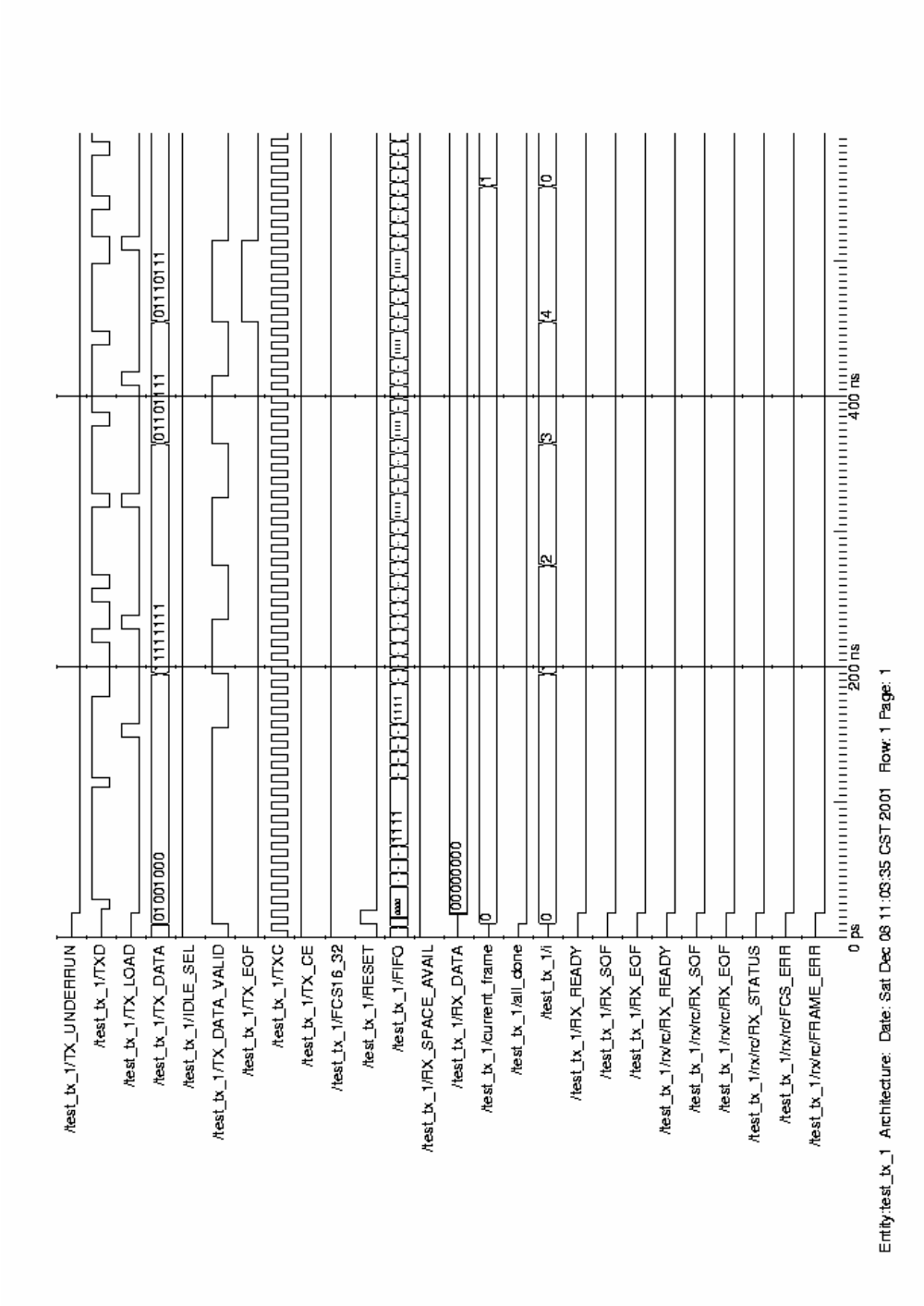
The modified testbench with annotations is given at the end of this section.

4.1 Single frame and back to back frames

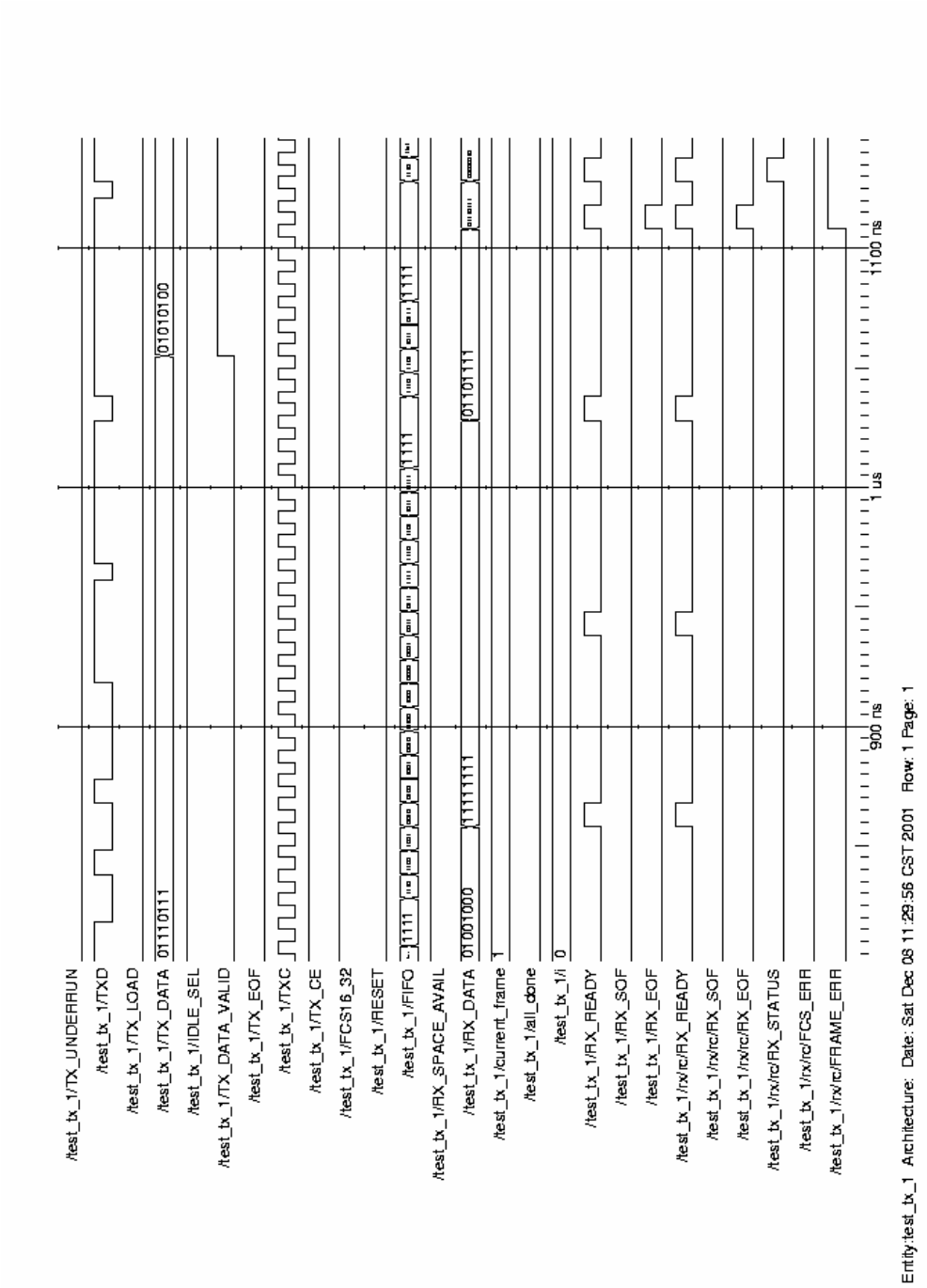
4.1.1 Single Frame with 16/32 FCS

Frame 1: The following is the simulation result for the first frame. It consists of 5 bytes. Hence it shows a frame error for the 32 bit FCS but no frame error for the 16 bit FCS.

Transmitter(32 bits FCS):

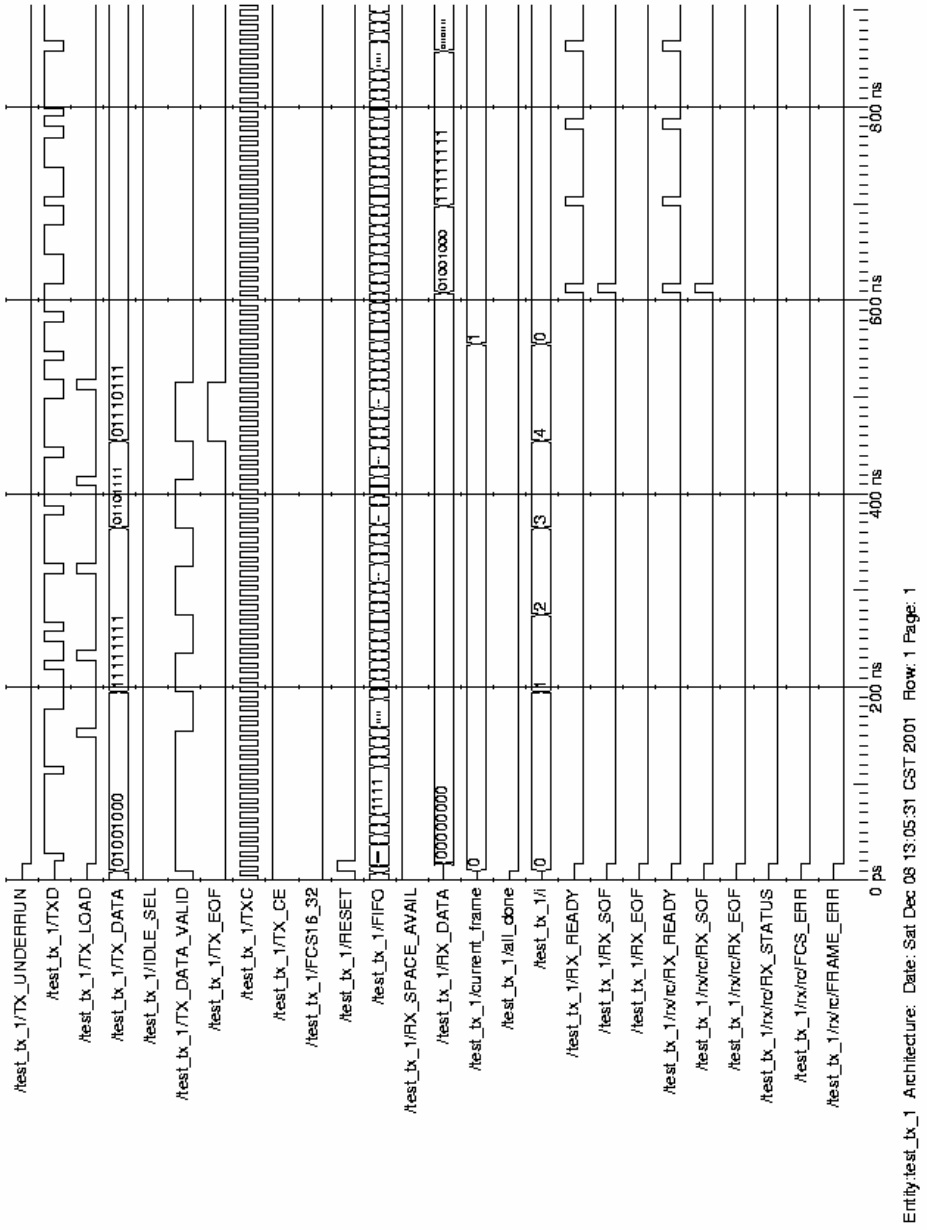


Receiver (32 bit FCS):

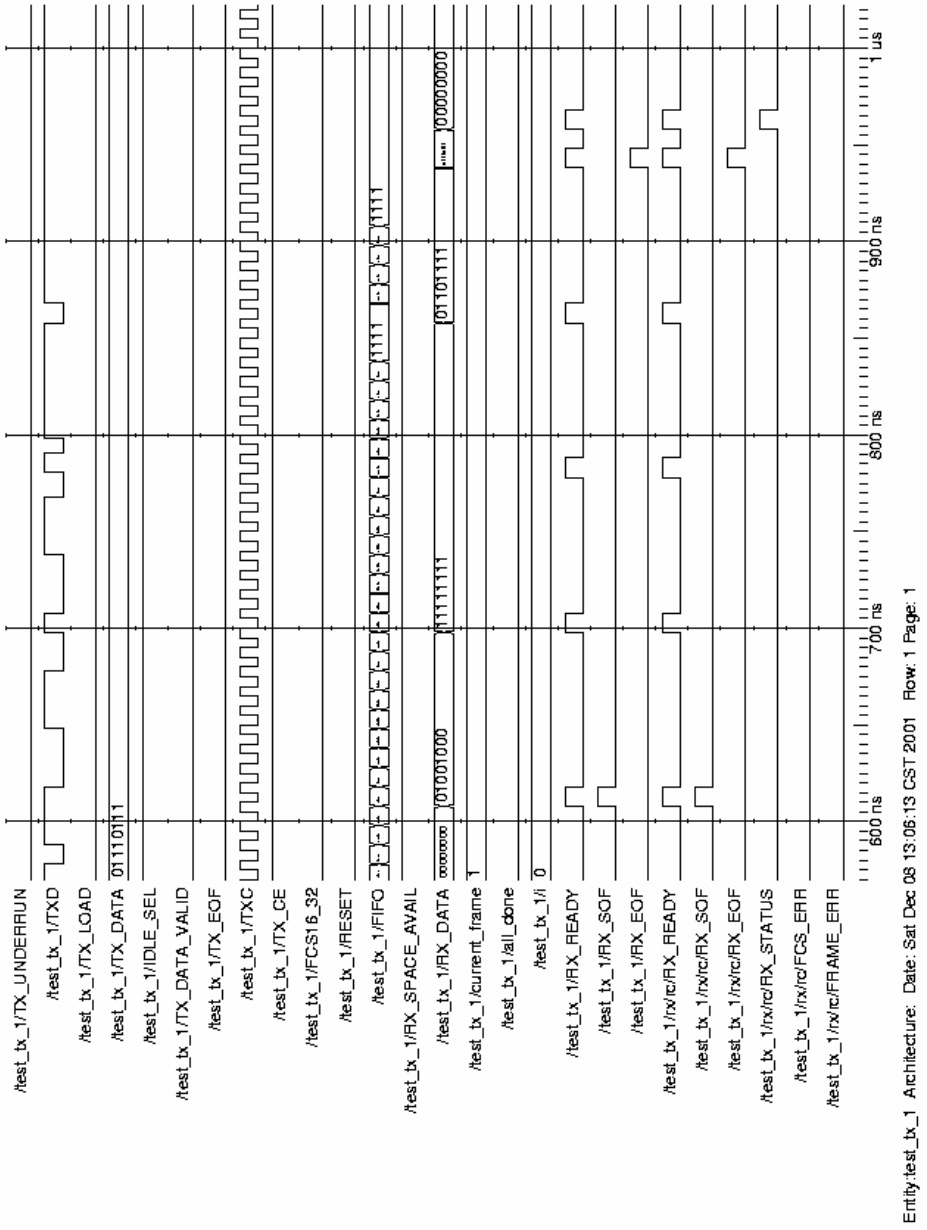


As we can see, the status byte indicates a frame error for 32 bit FCS.

Transmitter (16 bit FCS):



Receiver (16 bit FCS) :

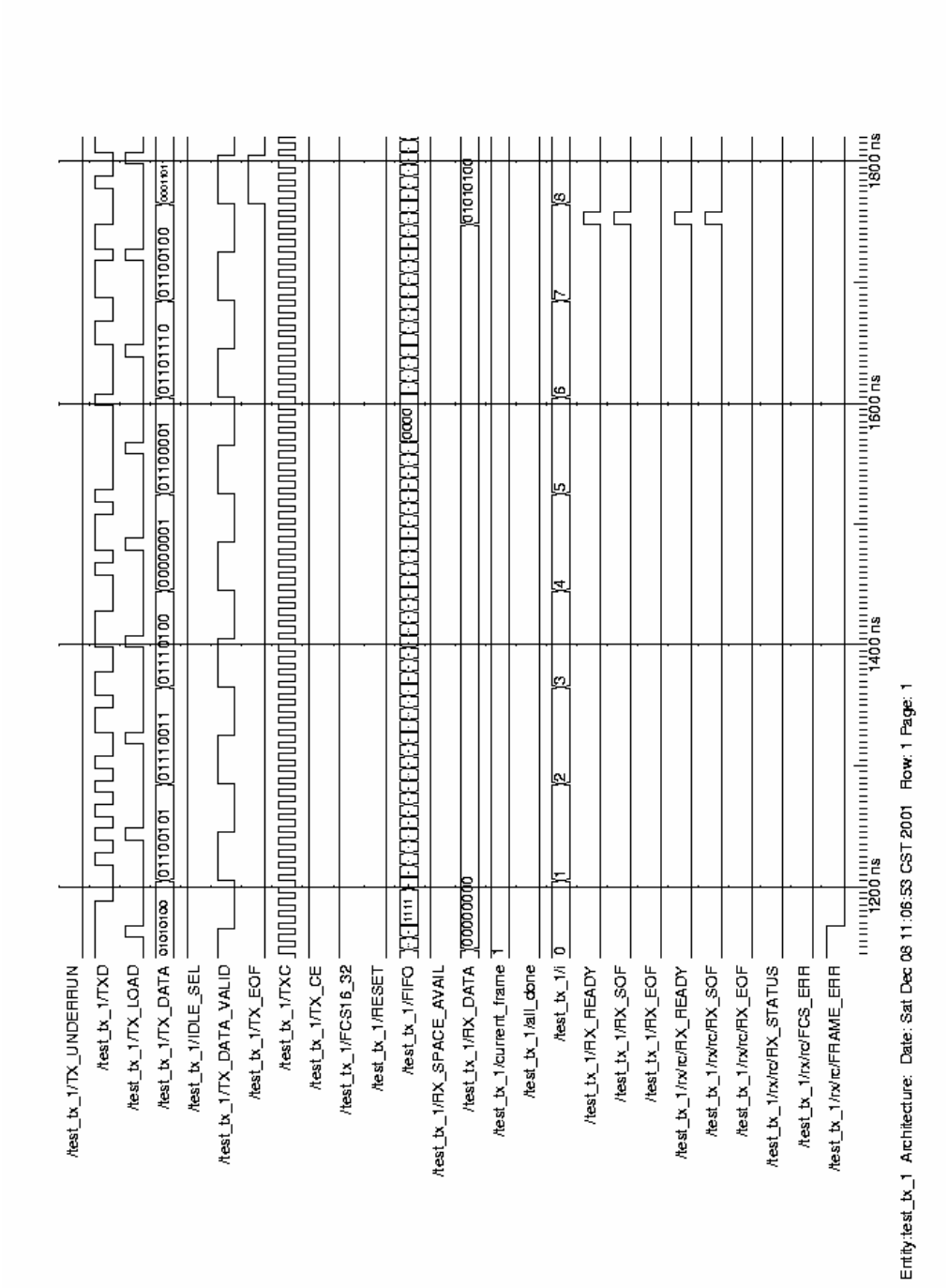


As we can see, the status byte indicates no frame error for the 16 bit FCS.

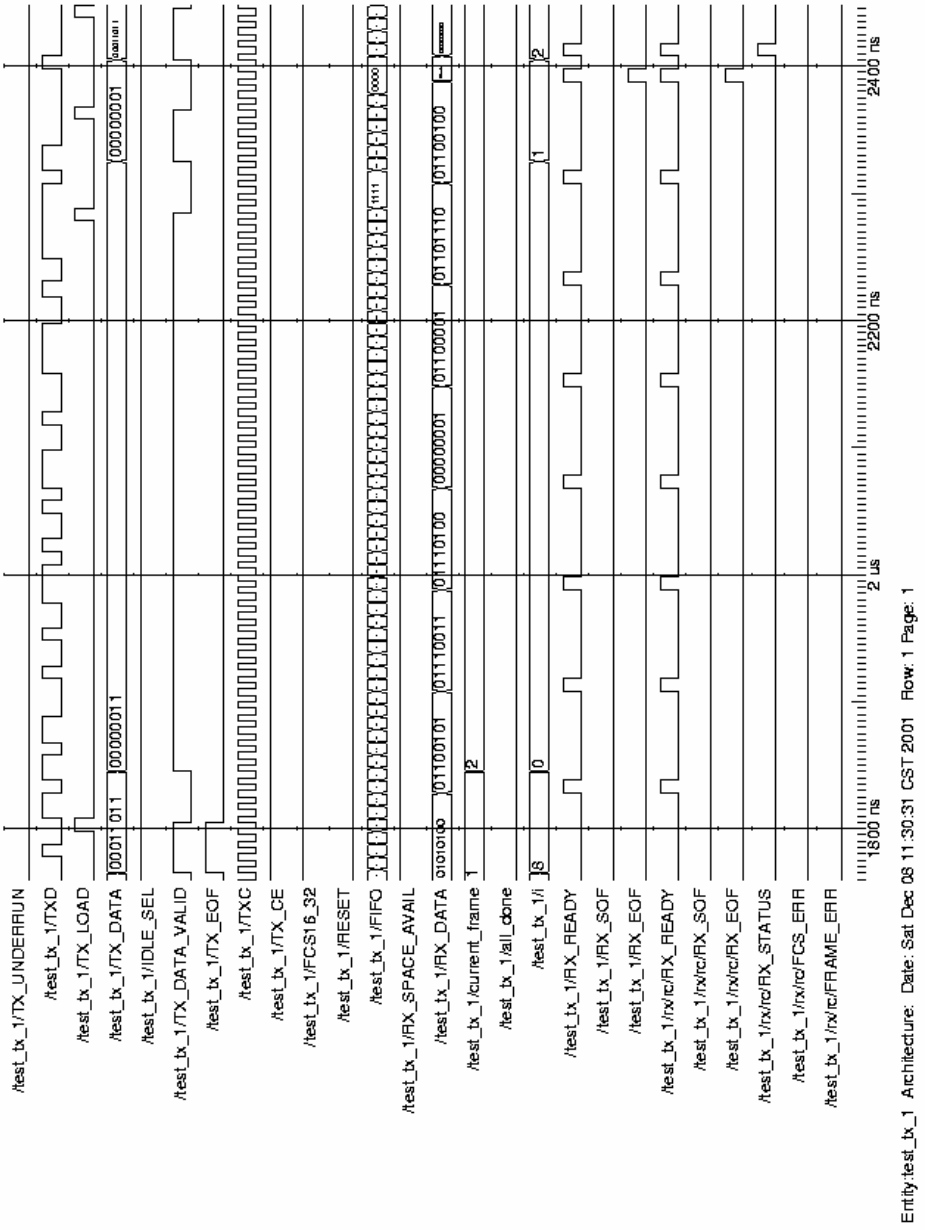
4.1.2 Back to back frames with 16/32 FCS

Frame 2: The frame 2 is an error free frame for the 16 and 32 bit FCS

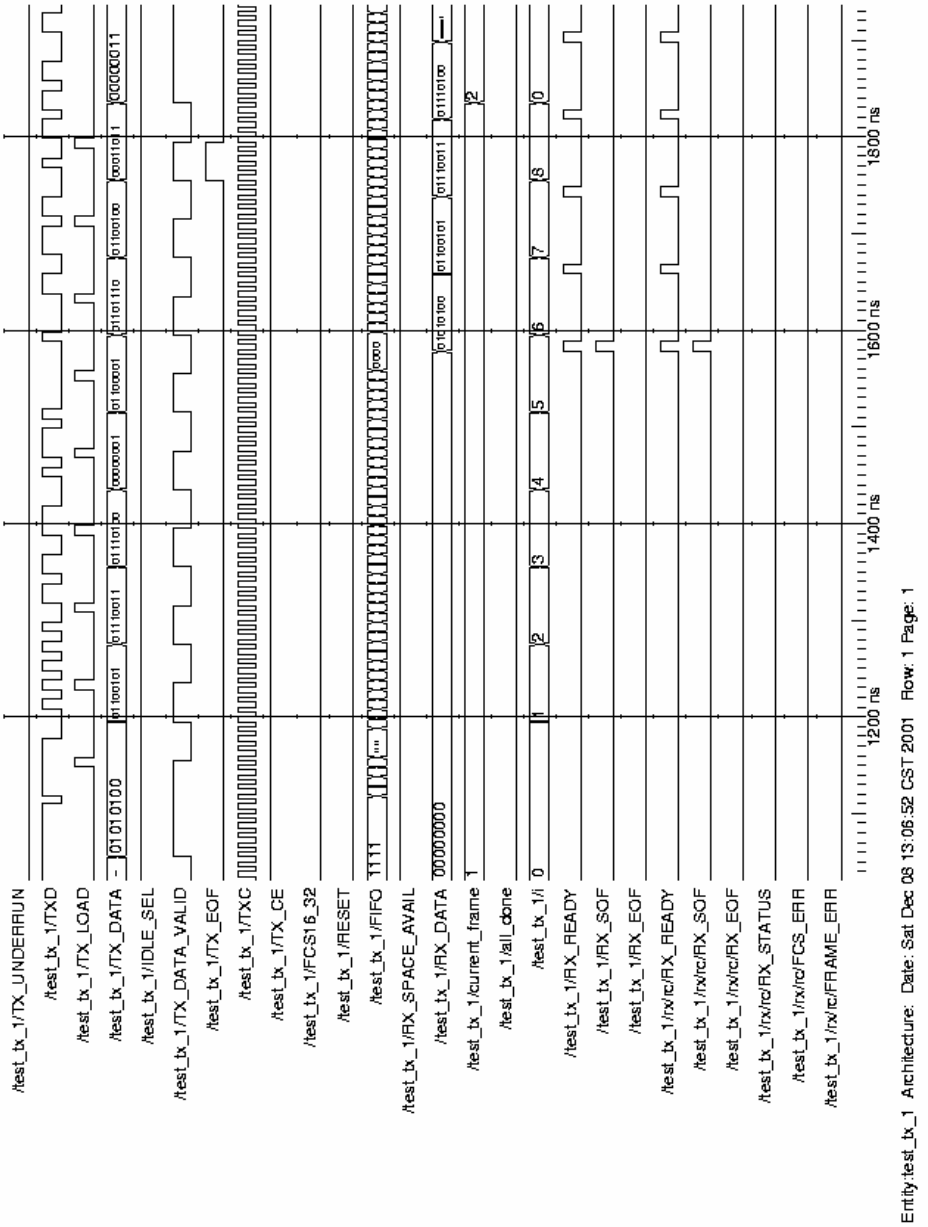
Transmitter (32 bit FCS) :



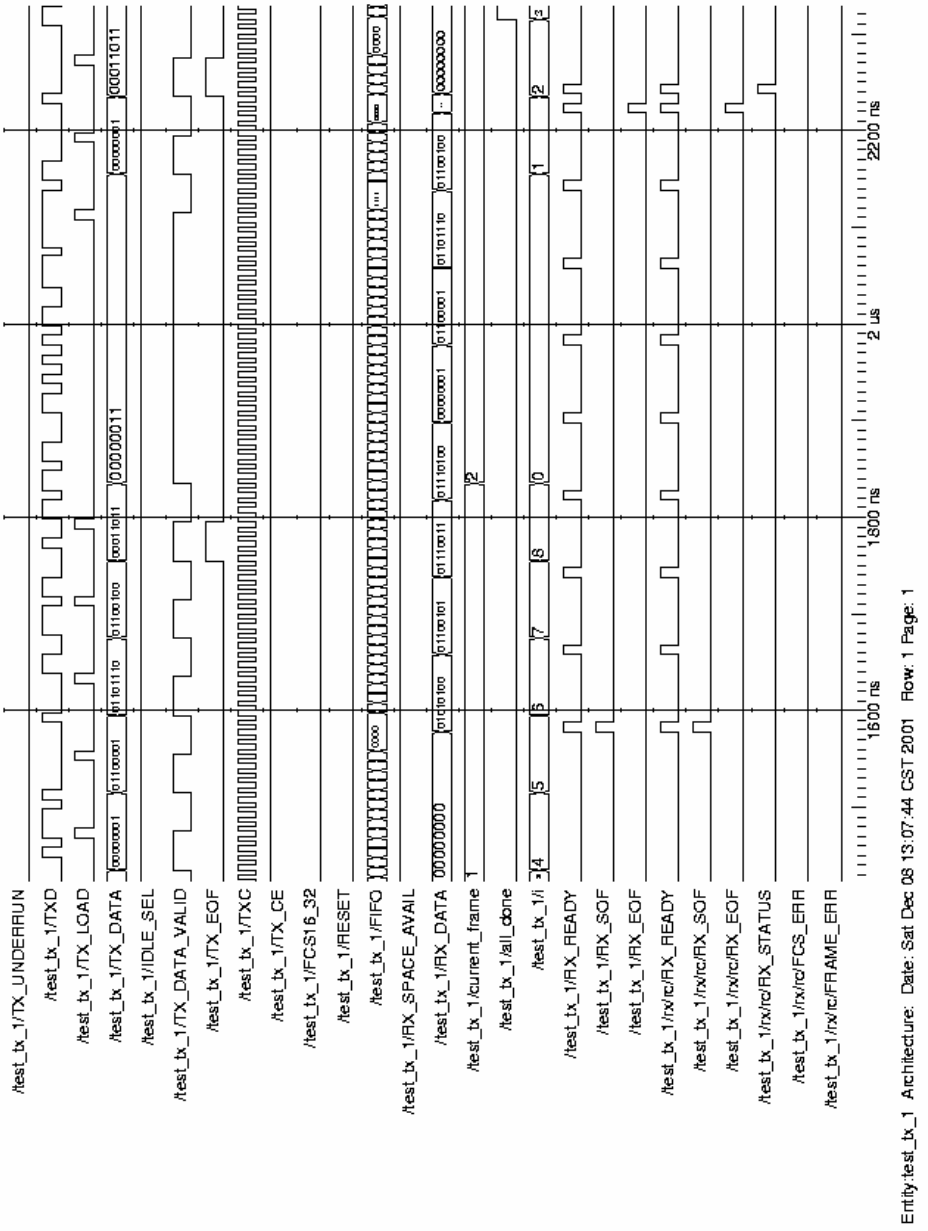
Receiver (32 bit FCS) :



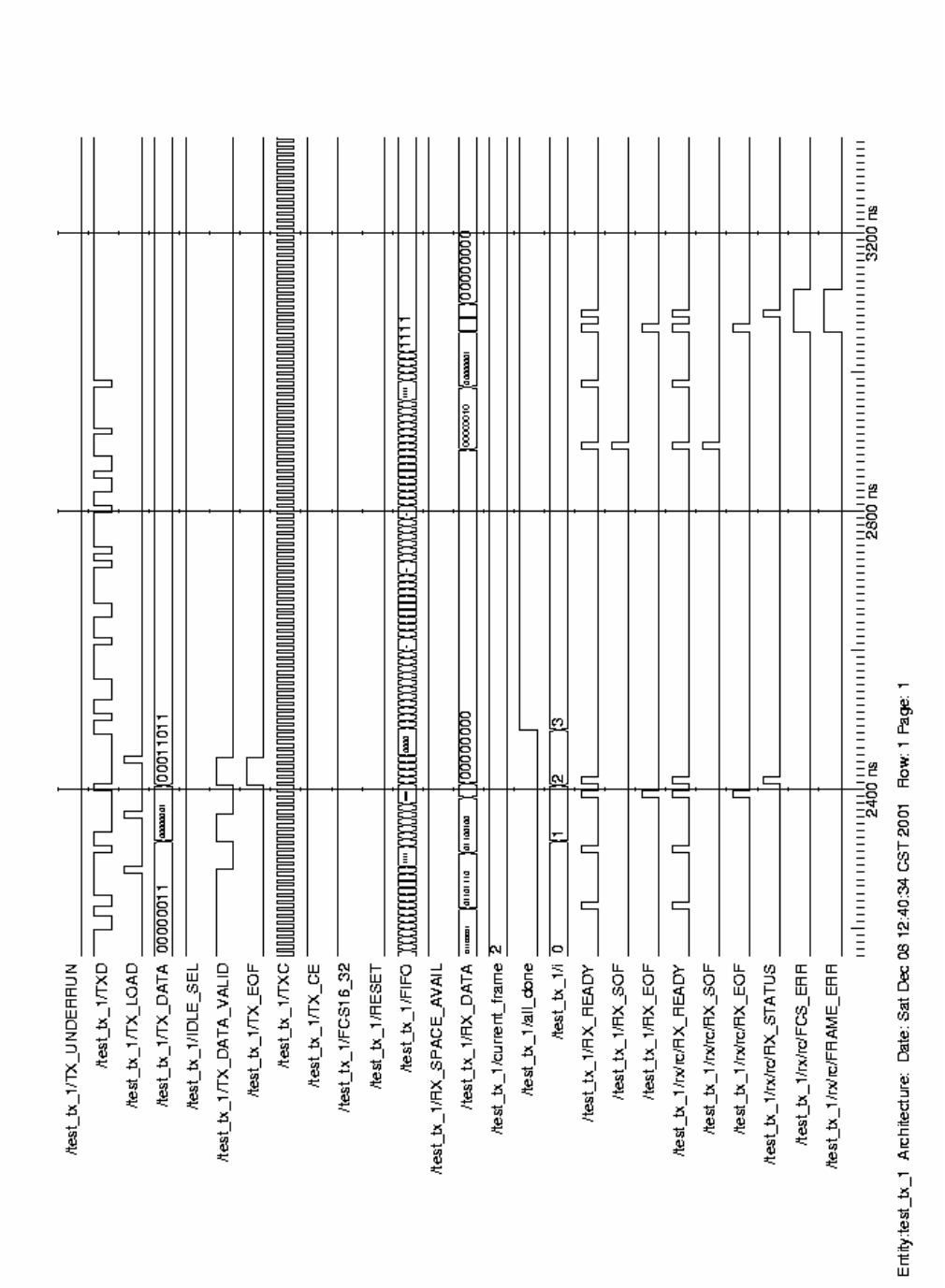
Transmitter (16 bit FCS) :



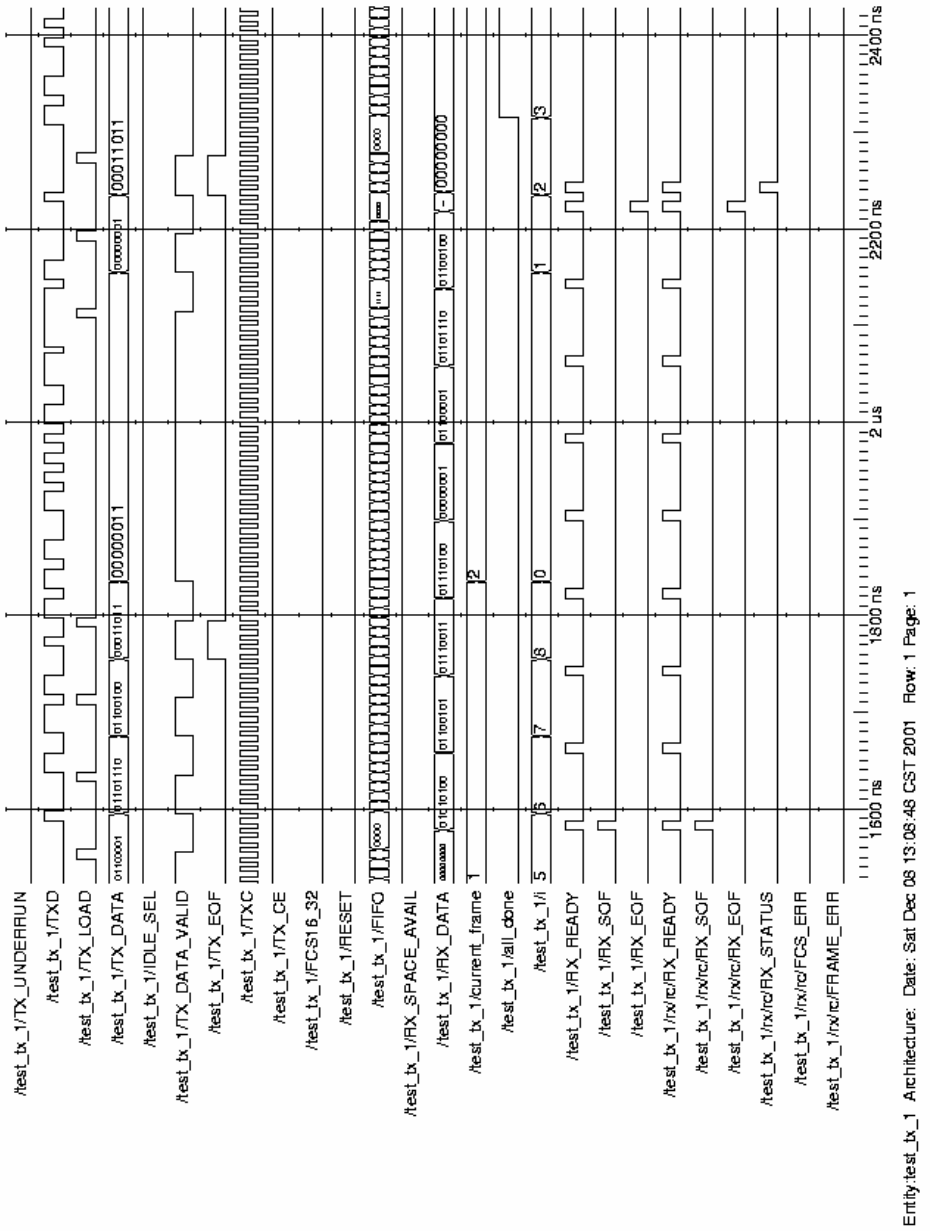
Receiver (16 bit FCS) :



Frame 3: This is a **back to back** frame with Frame 2. The following trace shows the region between the frame 2 and frame 3 for 32 bit FCS.

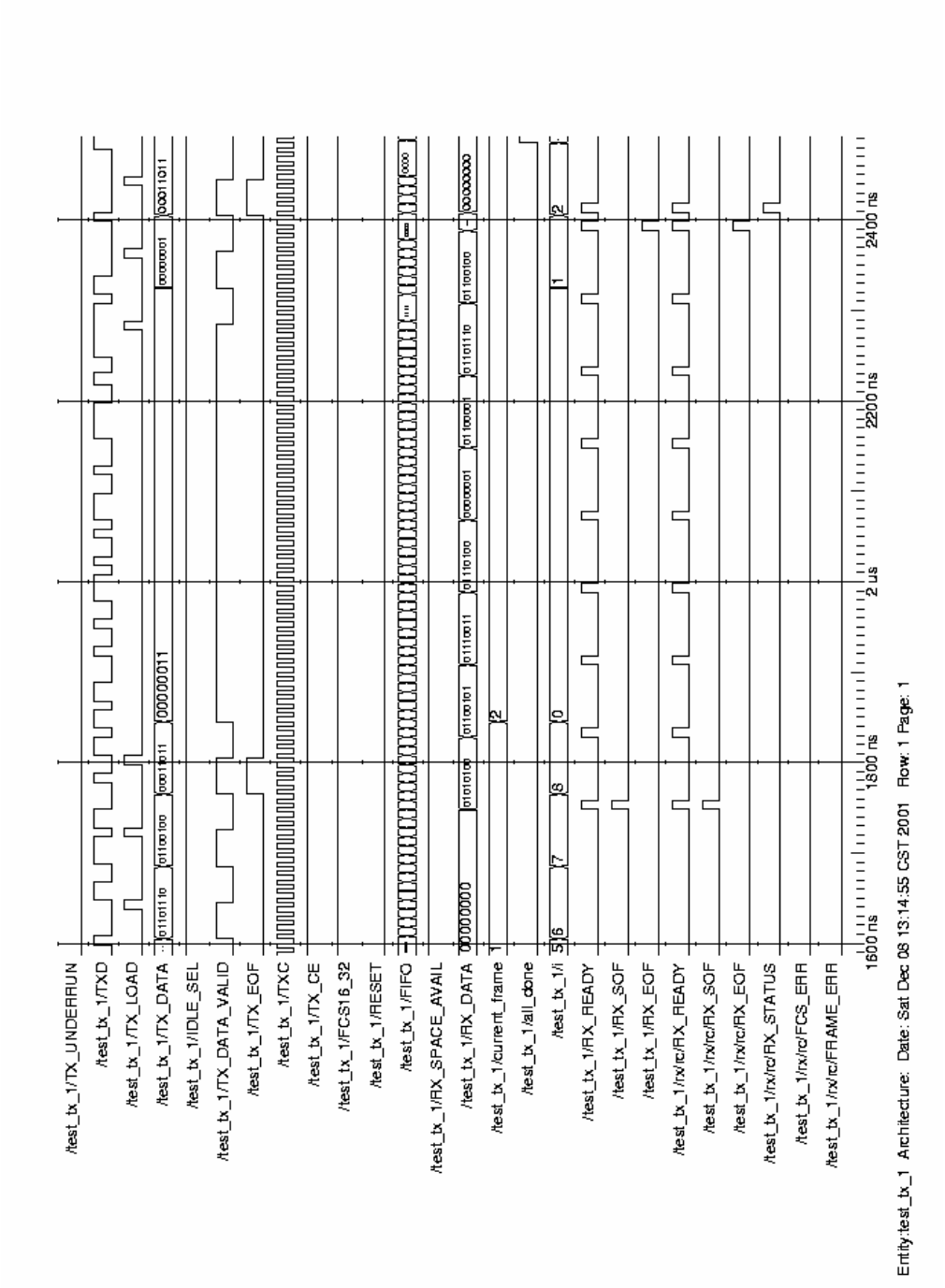


Frame 3: This is a **back to back** frame with Frame 2. The following trace shows the region between the frame 2 and frame 3 for 16 bit FCS.

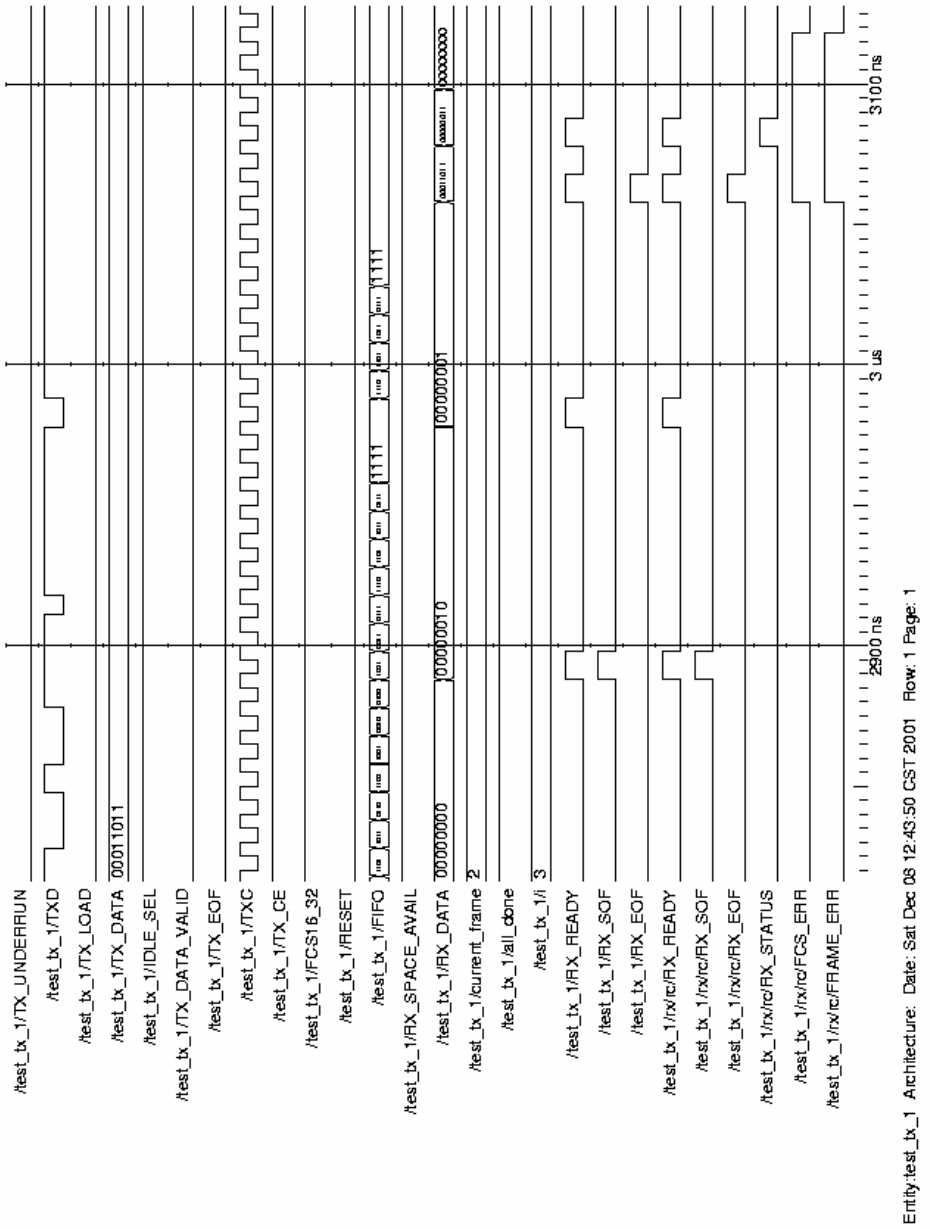


4.1.3 Frame Error in back-to-back frame

Transmitter (32 bit FCS) : This frame is a 3 byte frame. Hence it shows a frame error as well as an FCS error for the 32 bit FCS.

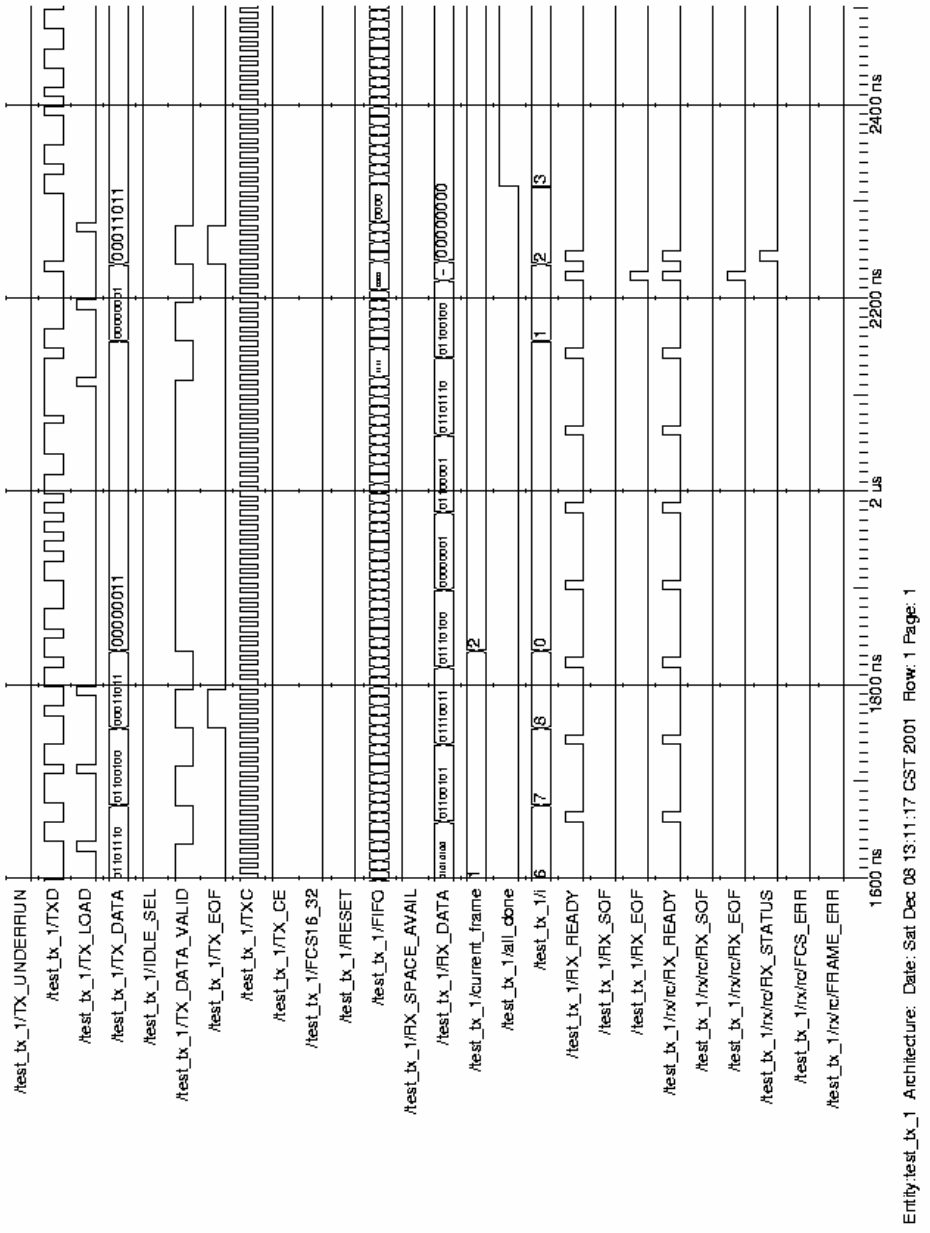


Receiver (32 bit FCS):

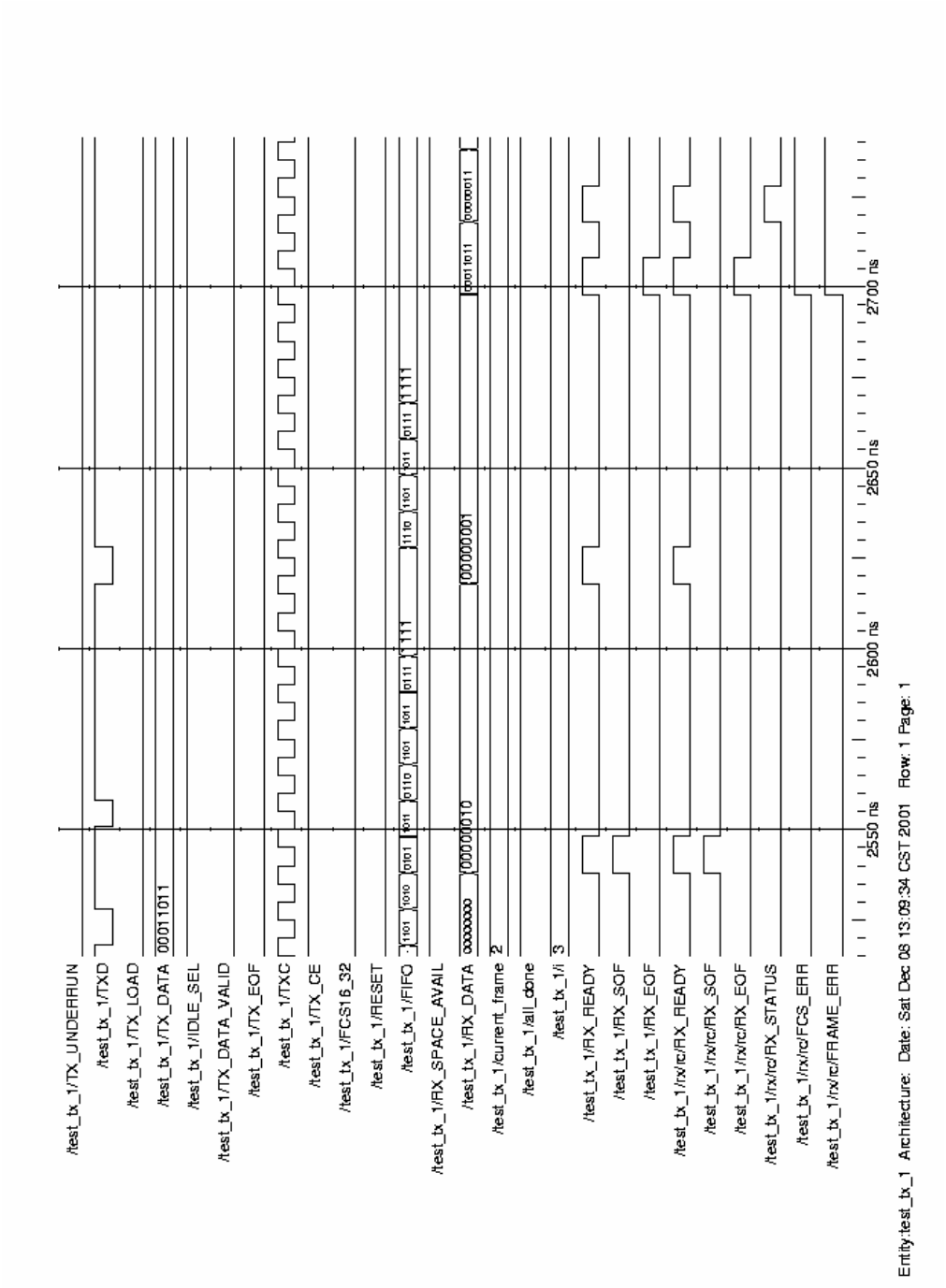


As can be seen from the status byte, the frame exhibits frame error and FCS error.

Transmitter (16 bit FCS) : This is a 3 byte frame. Hence it depicts a frame error and an FCS error for the 16 bit FCS. The reason why there is an FCS error along with frame error is that in our design we do not reset the FCS registers if we receive a back-to-back frame. If the second frame is too short, then the obsolete values that reside in the registers are not shifted out. Thus the comparison will indicate an error. Although this can be fixed easily, it is not an issue. Since the FCS of a frame that is shorter than the length of FCS is meaningless.



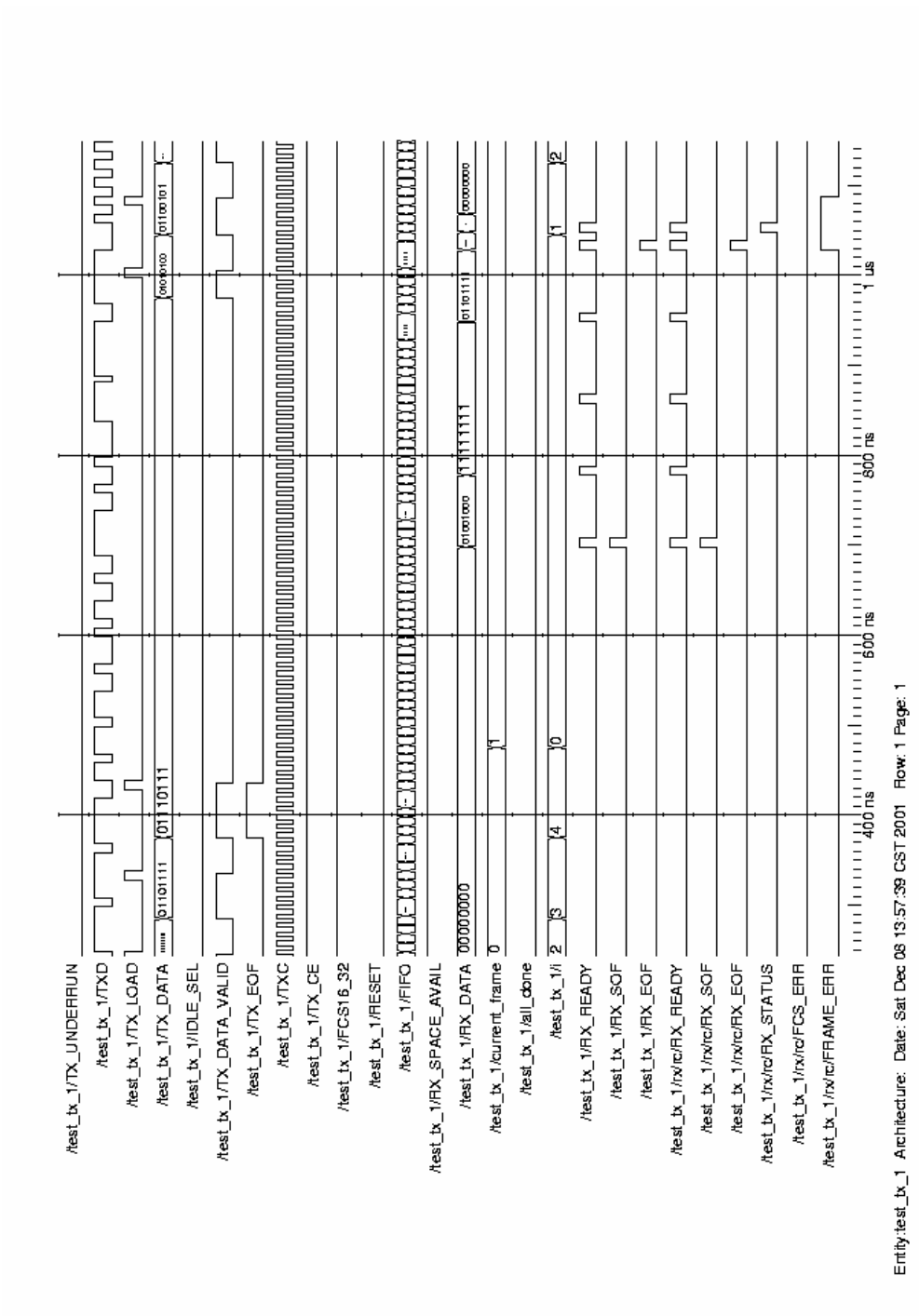
Receiver (16 bit FCS):



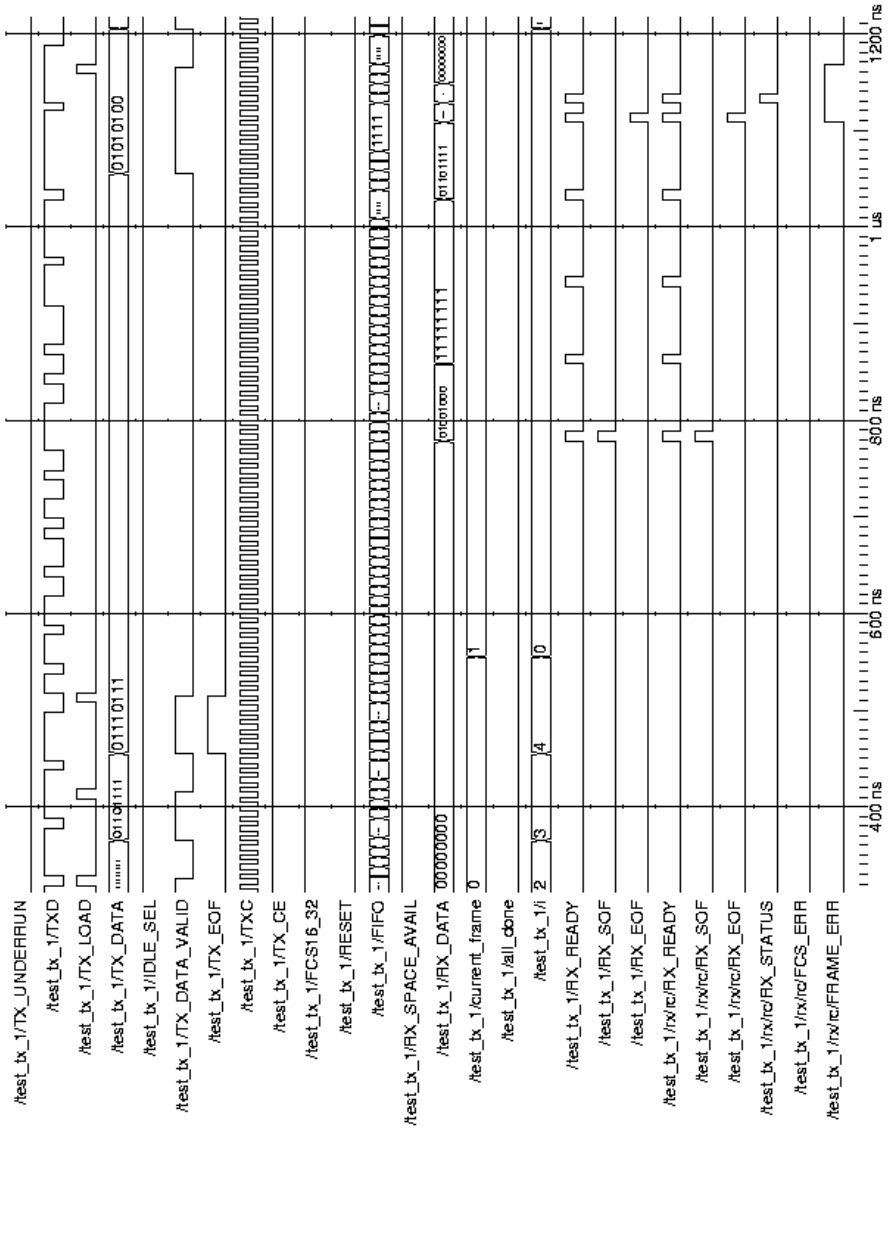
As can be seen from the status byte, the frame exhibits a frame error and an FCS error.

4.1.4 IDLE_SELECT modes with 16/32 FCS

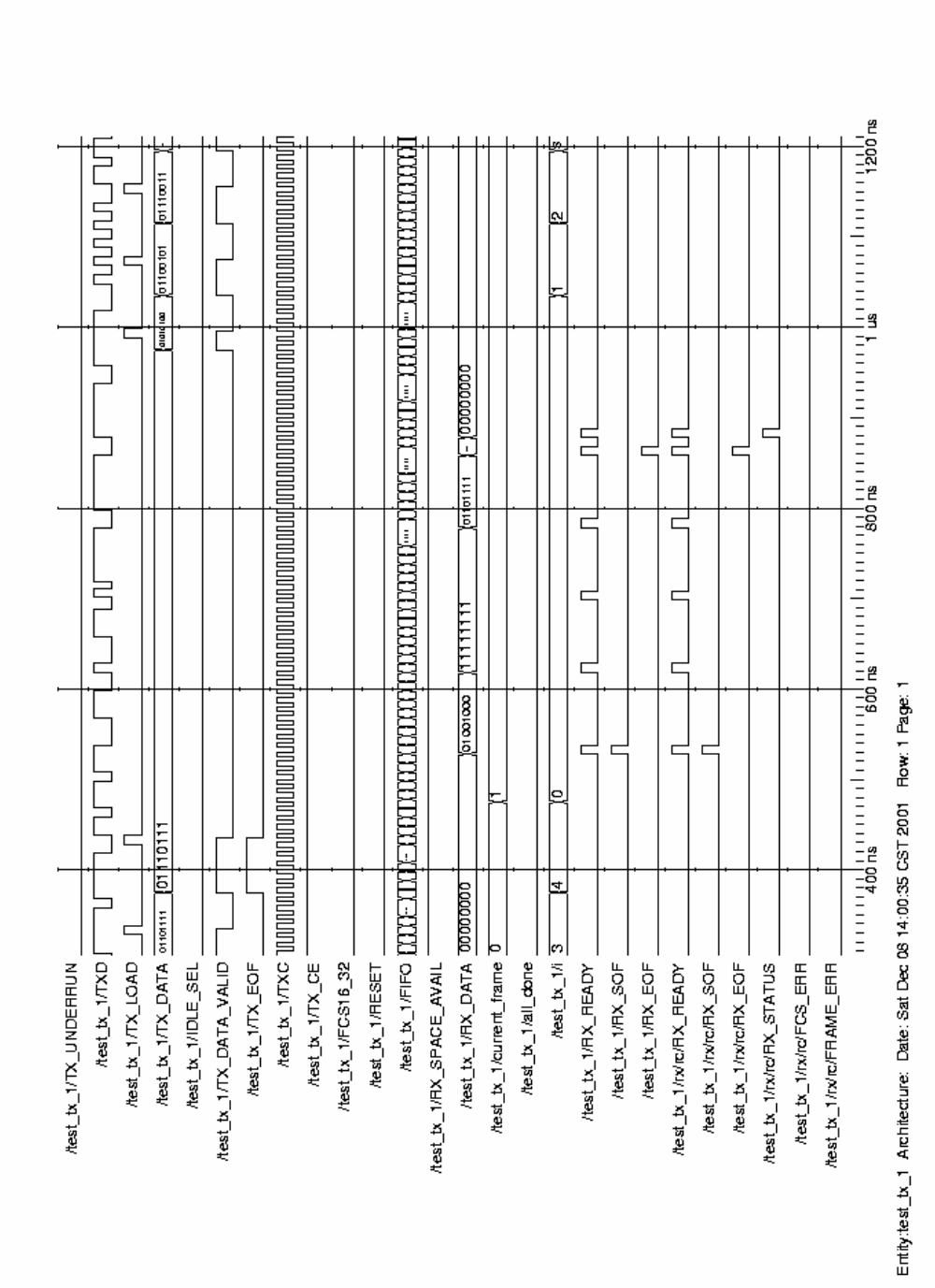
Idle select 0: for the 32 bit FCS case, with idle select = 0, between the frames 1 and 2, flags will be transmitted.



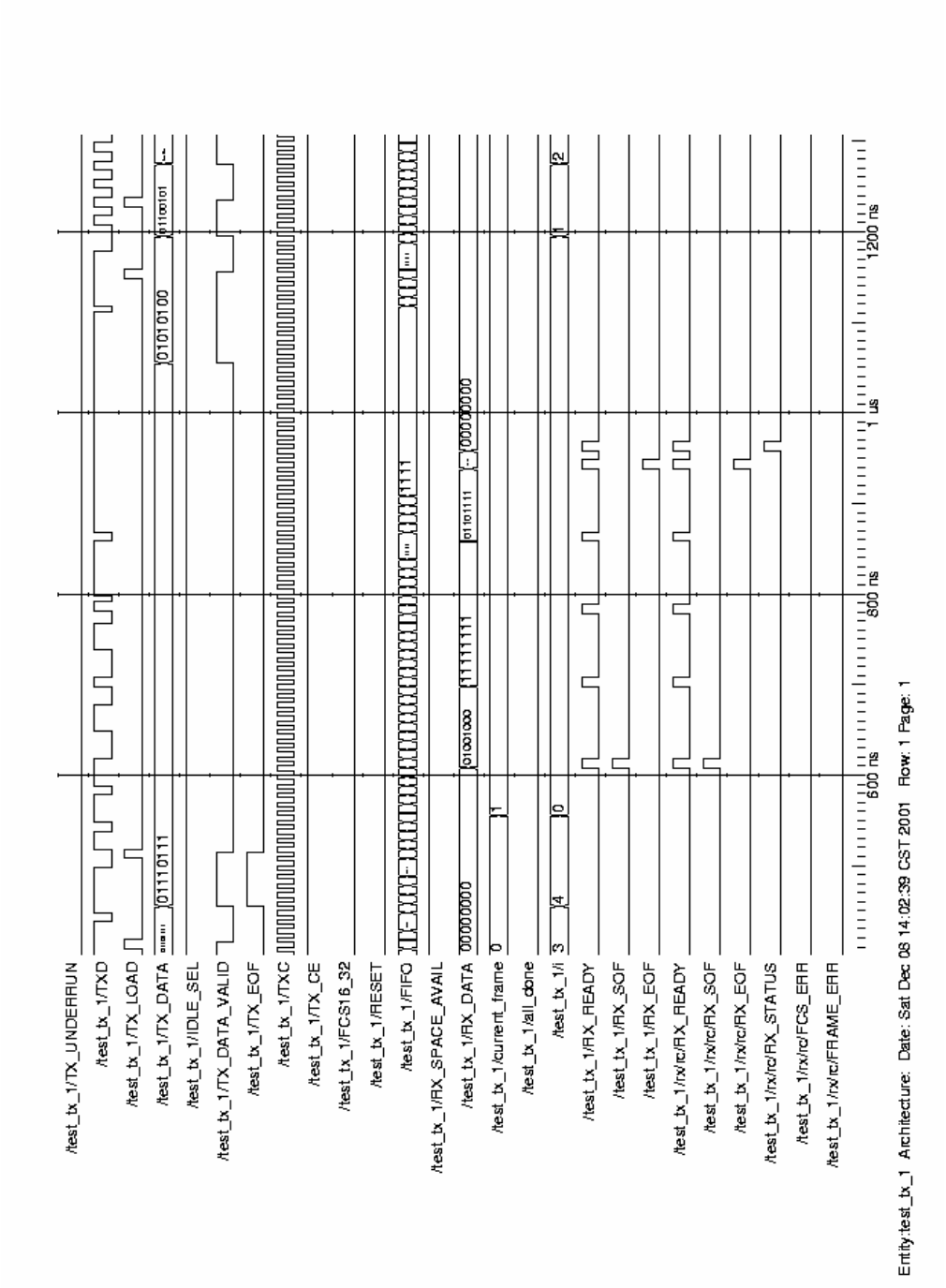
Idle Select 1: For the 32 bit FCS case, with idle select = 1, between frames 1 and 2 1's will be transmitted.



Idle Select 0: For the 16 bit FCS case, with idle select = 0, between frames 1 and 2 flags will be transmitted.



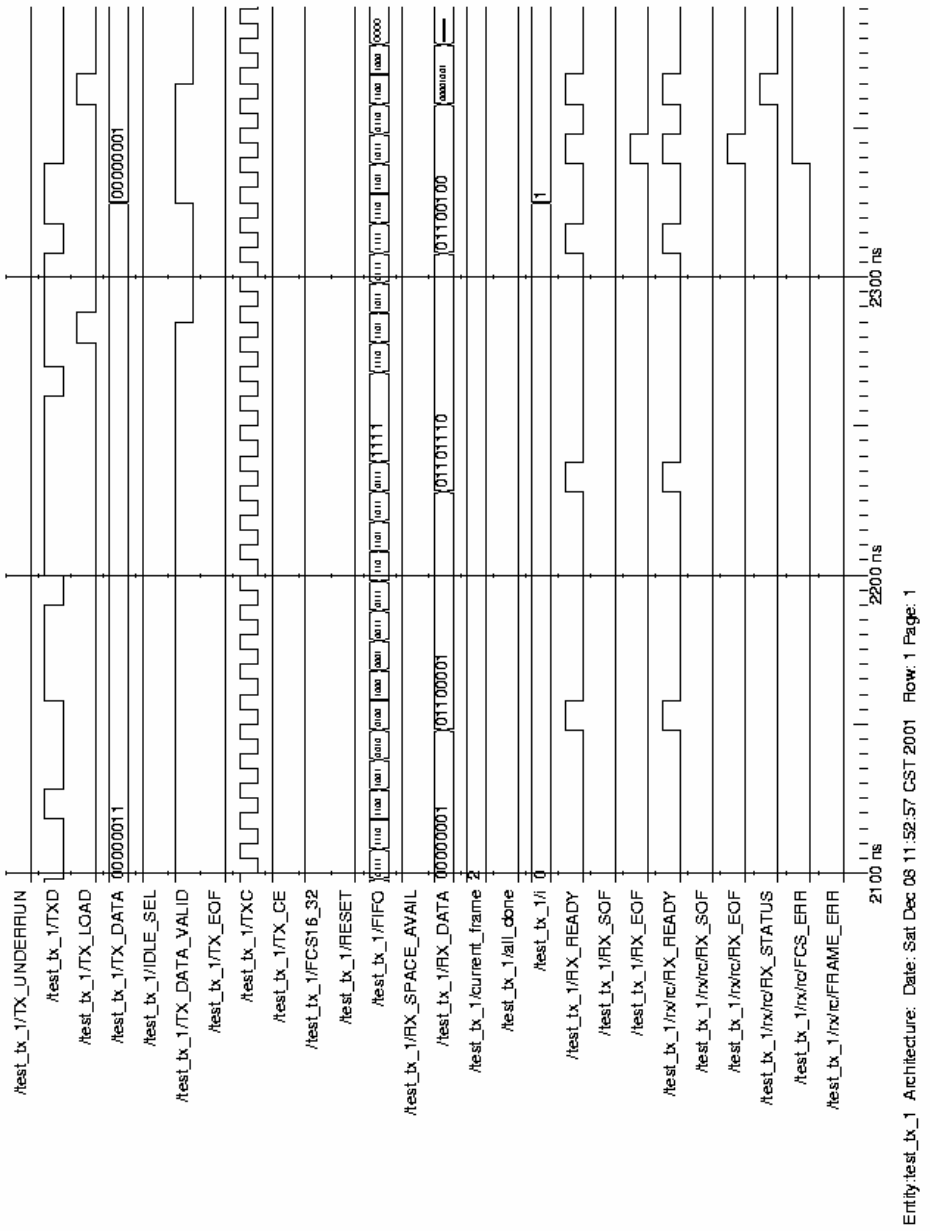
Idle Select 1: For the 16 bit FCS case, with idle select = 1, between frames 1 and 2 1's will be transmitted.



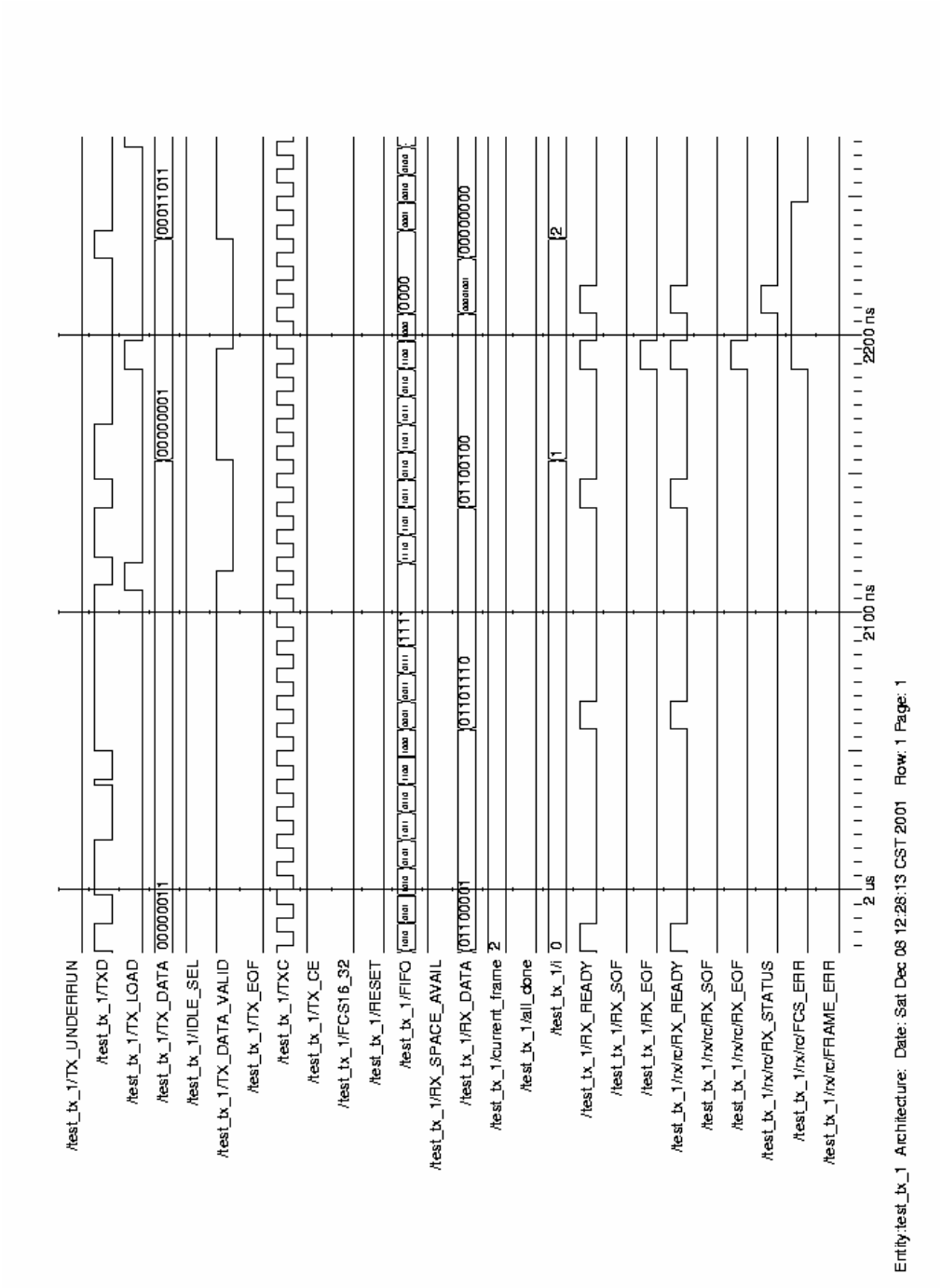
4.2 Error detection

4.2.1 Octet Error

Octet Error : The following trace shows an octet error for the 32 bit FCS case. This occurs when the receiver receives a flag at an odd bit boundary, i.e. between a byte.

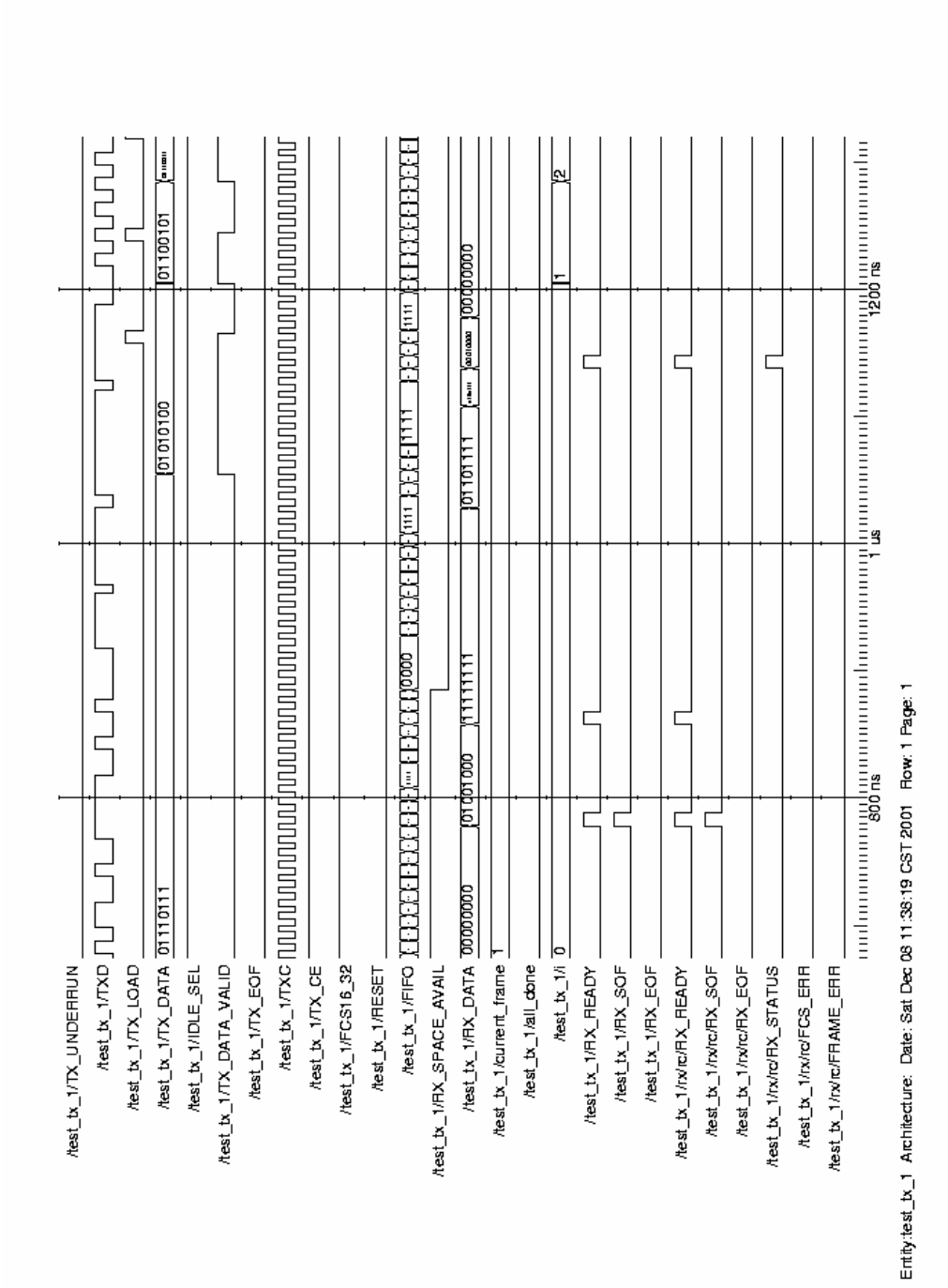


Octet Error : The following trace shows an octet error for the 16 bit FCS case.

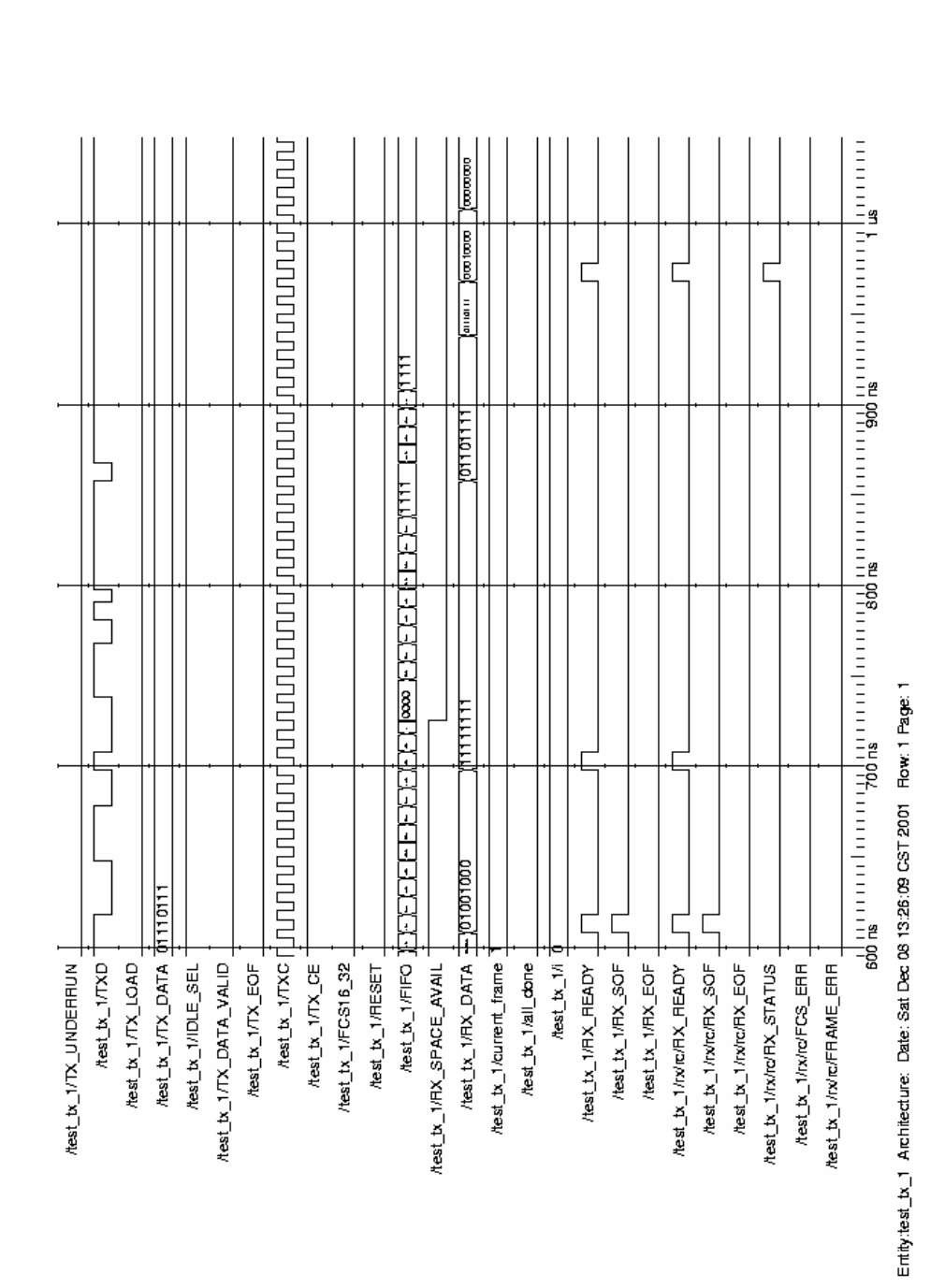


4.2.2 Overrun Error

Overrun Error : The following trace shows an overrun error for the 32 bit FCS case. This occurs when the receiver has data to be sent to the user but the user deasserts the RX_SPAC_AVAIL signal, indicating that it cannot accept more data.



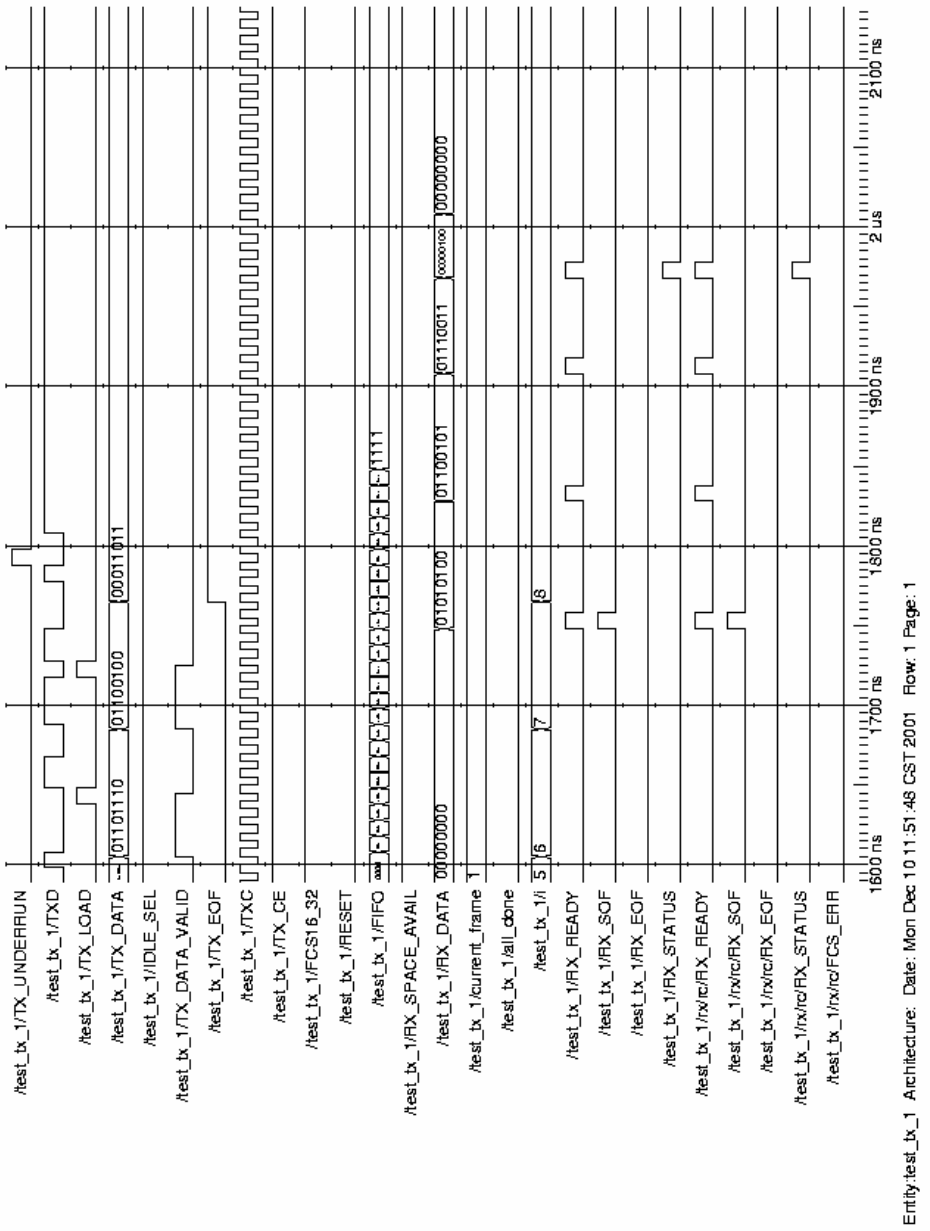
Overrun Error : The following trace shows an overrun error for the 16 bit FCS case.



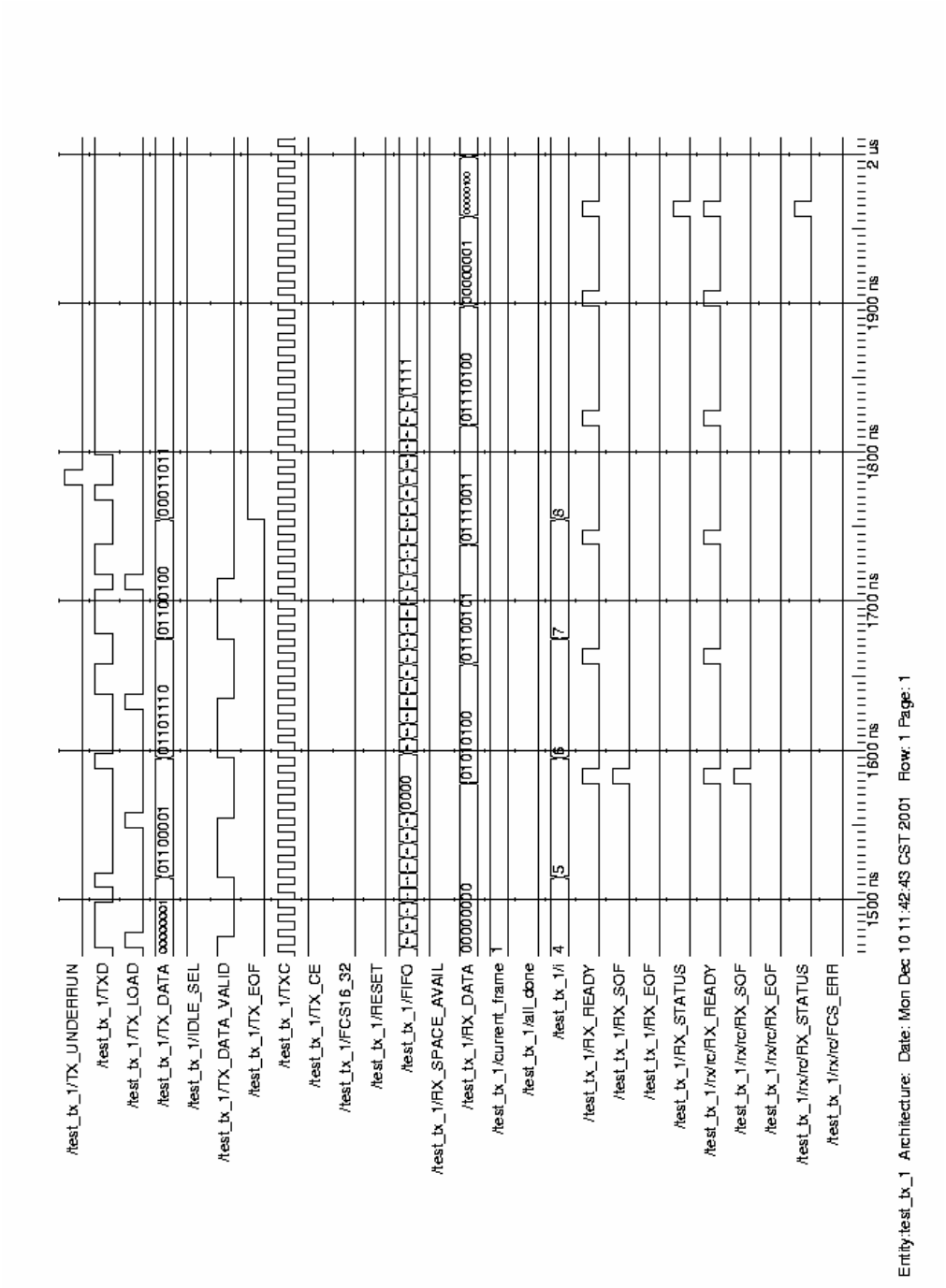
4.2.3 Abort Error

Abort Error : The following trace shows an abort error for the 32 bit FCS case. This occurs when the TX_DATA_VALID signal gets deasserted for a byte in the transmitter.

It causes the Transmitter to send a string of 1's to the receiver, which gets detected as Abort error by the receiver.



Abort Error : The following trace shows an abort error for the 16 bit FCS case.



4.3 Testbench

```
`timescale 1ns/100ps

/*     ECE551 DIGITAL SYSTEM DESIGN AND SYNTHESIS
                    FALL 2001
              David Leonard & Chuck Kime
*/
       `define clock_period  10
       `define num_frames    3

module test_tx_1;

// Output Signals from Transmitter

wire TX_UNDERRUN, TXD;

/*The following declaration is for normal operation; for testing of test_tx_1, it should
be
commented out.*/

wire TX_LOAD;

// Input Signals to Transmitter
   reg [7:0] TX_DATA;
   reg IDLE_SEL, TX_DATA_VALID, TX_EOF, TXC, TX_CE, FCS16_32, RESET;

//   FIFO to simulate channel between Transmitter and Receiver
   reg [3:0] FIFO;
   reg RX_SPACE_AVAIL;
   wire [7:0] RX_DATA;


// Frame Buffer
   reg [7:0] frame0 [4:0];
   reg [7:0] frame1 [8:0];
   reg [7:0] frame2 [4:0];
   integer frame_size [(`num_frames-1):0];
   integer inter_frame_delay [(`num_frames-1):0];
   integer current_frame;
   reg all_done;
   integer i;


     Transmitter tx(TXD, TX_LOAD, TX_UNDERRUN, TX_DATA, TX_DATA_VALID, TX_EOF, IDLE_SEL,
FCS16_32, RESET, TX_CE, TXC);
     Receiver rx(RX_DATA, RX_READY, RX_SOF, RX_EOF, RX_STATUS, FIFO[3], FCS16_32,
RX_SPACE_AVAIL, RESET, TX_CE, TXC);

// Initialize Transmitter Control Signal Values
   initial
     begin
        IDLE_SEL     = 1'b1;        // Change Idle Select
        TX_DATA      = 8'b0;
        TX_DATA_VALID = 1'b0;
        TX_EOF       = 1'b0;
        TXC         = 1'b0;
        TX_CE        = 1'b1;
        FCS16_32     = 1'b1;        // Change FCS Select
        RESET        = 1'b0;
// reset all circuitry
        #(`clock_period) RESET = 1'b1;
        #(`clock_period) RESET = 1'b0;
    end

// Initialize Receiver Control Signal Values
     initial
          begin
    RX_SPACE_AVAIL = 1;
    FIFO = 0;
      # 1750 force TX_DATA_VALID = 0;    // Force Underrun Condition for 16 bit FCS
      # 50   release TX_DATA_VALID ;
```

```
      // #885 RX_SPACE_AVAIL = 0;     // Force Overrun Error in Receiver for 32 bit FCS
      // #725 RX_SPACE_AVAIL = 0;     // Force Overrun Error in Receiver for 16 bit FCS


    // # 2040  force TXD = 0;  // Force octet error for 16 bit FCS
    // # 10 force TXD = 1;
    // # 10 force TXD = 1;
    // # 10 force TXD = 1;
    // # 10 force TXD = 1;
    // # 10 force TXD = 1;
    // # 10 force TXD = 1;
    // # 10 force TXD = 0;
    // # 10 release TXD ;

    //  # 2190 force TXD = 0;  // Force octet error for 32 bit FCS
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 1;
    //  # 10 force TXD = 0;
    //  # 10 release TXD ;
         end

    always@(posedge TXC)
         begin
              if(RESET)
                   FIFO = 0;
              else
                   FIFO <= #3 {FIFO[2:0], TXD};
         end

  initial
    begin
    // Data contains long 1's       // Frame error for 32 bit FCS
              frame0[0] = 8'h48;
              frame0[1] = 8'hff;
              frame0[2] = 8'hff;
              frame0[3] = 8'h6f;
              frame0[4] = 8'h77;

     frame_size[0] = 5;
     inter_frame_delay[0] = 0;

    // init ram1
       frame1[0] = 8'h54;
       frame1[1] = 8'h65;
       frame1[2] = 8'h73;
       frame1[3] = 8'h74;
       frame1[4] = 8'h01;
       frame1[5] = 8'h61;
       frame1[6] = 8'h6e;
       frame1[7] = 8'h64;
       frame1[8] = 8'h1b;
       frame_size[1] = 9;
       inter_frame_delay[1] = 50;


    // init ram2           //Frame error and FCS error for 16 bit and 32 bit FCS
        frame2[0] = 8'h03;
        frame2[1] = 8'h01;
        frame2[2] = 8'h1b;

        frame_size[2] = 3;
        inter_frame_delay[2] = 0;
    end
    // End Frame Buffer

// Generate Transmitter Clock
  always
    #(`clock_period/2) TXC = ~TXC;

// Generate the Rest of the Transmitter Control Signals (Dynamic)
  always@(posedge RESET or posedge TXC)
    begin
```

```
        if(RESET) // reset condition
          begin
             current_frame  = 0;
             all_done     = 0;
             i           = 0;
          end

// ...if all frames have NOT been sent out
        if(~all_done)
          begin
             // FIRST BYTE OF FRAME
             if(i == 0)    // first byte of frame?
               begin
                 if(inter_frame_delay[current_frame] == 0)
             // no delay?
                begin
                      // select appropriate frame and its data
                      case(current_frame)
                         0: TX_DATA = frame0[i];
                         1: TX_DATA = frame1[i];
                         2: TX_DATA = frame2[i];
                      endcase
                      TX_DATA_VALID = 1'b1;
                 end
                else // if so, decrement delay
                   begin
                       inter_frame_delay[current_frame] =
inter_frame_delay[current_frame] - 1;
                   end
                end
             // ALL OTHER BYTES OF FRAME
               if(TX_LOAD) // loaded previously-provided byte?
                  begin
                 TX_DATA_VALID = 0;
               TX_EOF = 0;
                  # (`clock_period + `clock_period + `clock_period + `clock_period)
                   i = i + 1;
                   // in frame?
                   if(i < frame_size[current_frame])
                     begin
                     //select appropriate frame and its data
                        case(current_frame)
                           0: TX_DATA = frame0[i];
                           1: TX_DATA = frame1[i];
                           2: TX_DATA = frame2[i];
                        endcase
                     // last byte of frame?
                        if(i == (frame_size[current_frame] - 1))
                          begin
                          //signal end of frame
                           TX_EOF = 1'b1;
                          end
                        TX_DATA_VALID = 1'b1;
                     end
                   // if not (in frame), deassert TX_DATA_VALID
                   else
                     begin
                        TX_DATA_VALID = 1'b0;
                   TX_EOF = 0;
                        // last frame?
                        if(current_frame == (`num_frames - 1))
                          begin
                             all_done = 1;
                          end
                        else // if not, go to next frame
                          begin
                             current_frame = current_frame + 1;
                             i = 0;
                          end
                     end
                   end
            else
                  TX_DATA = TX_DATA;
         end // end of "all_done" if statement
      end // end of always statement
endmodule
```

5 Synthesis Results

In this section we first give the summary of our synthesis result. Then more detailed simulation setup and results are listed.

5.1 Performance Summary

Table 5-1 gives the performance summary of our synthesis result. We set our target speed to 400MHz and try to minimize our area. Although our design can run up to 500MHz, those data are not included.

	Before Optimization			**After** Optimization		
	Area	Frequency	Merit	Area	Frequency	Merit
Transmitter (Normal Case)	7113	400MHz	0.0562	7008	400MHz	0.0571
Transmitter (Worst Case)	7775	400MHz	0.0514	7319	400MHz	0.0547
Receiver (Normal Case)	11235	400MHz	0.0356	10890	400MHz	0.0367
Receiver (Worst Case)	11600	400MHz	0.0345	11289	400MHz	0.0354

Table 5-1

Table 5-2 list the attributes and constraints we use throughout our synthesis process.

Parameter	Value
Clock period	2.5ns
Input Drive(All but RXC/TXC)	Drive_of(lcbg11io/BUFDR/Z)
Input Drive(RXC/TXC)	Drive_of(lcbg11io/CLK2I/Z)
Input Delay	1.0ns
Output load	Load_of(lcbg11io/B28NTLDR/A)
Output delay	1.0ns

Table 5-2

5.2 Normal condition Transmitter synthesis

Table 5-3 lists the attributes we use for synthesis under normal condition and table 5-4 lists the area report for out optimized design.

Optimization	Ungroup	Flatten	Structure (Timing,Boolean)	Boundary Optimization	Area	Slack	Figure of Merit
Original					7113	0.00	0.0562
			√, √	√	7043	0.00	0.0568
			√, X	√	7055	0.00	0.0567
		√	√, X	√	7057	0.00	0.0567
		√	√, √	√	7051	0.00	0.0567
	√	√	√, √		7031	0.00	0.0569
	√	√	√, X		7010	0.00	0.0571
*	√	√	√, X	√	7008	0.00	0.0571
	√	√	√, √	√	7051	0.00	0.0567

Table 5-3

```
****************************************
Report : area
Design : Transmitter
Version: 2000.11
Date   : Mon Dec 10 10:52:01 2001
****************************************

Library(s) Used:
   lcbg11p (File:
/afs/engr.wisc.edu/apps/eda/flexstream.2.0/lsi_fs_2.0/lib3p/synopsys/lcbg11p/lcbg11p_n
om.db)
```

```
Number of ports:            18
Number of nets:             34
Number of cells:             7
Number of references:        7
Combinational area:      2111.021973
Noncombinational area:   3263.312500
Net Interconnect area:   1633.835938

Total cell area:         5374.334473
Total area:              7008.170410
```

Table 5-4

5.3 Worst-case Transmitter synthesis

Table 5-5 lists the attributes we use for synthesis under worst-case and table 5-6 lists the area report for out optimized design.

Optimization	Ungroup	Flatten	Structure (Timing,Boolean)	Boundary Optimization	Area	Slack	Figure of Merit
Original					7775	0.00	0.0514
			√, √	√	7585	0.00	0.0527
			√, X	√	7419	0.00	0.0539
		√	√, X	√	7418	0.00	0.0539
		√	√, √	√	7372	0.00	0.0543
	√	√	√, √		7496	0.00	0.0534
	√	√	√, X		7565	0.00	0.0529
	√	√	√, X	√	7537	0.00	0.0531
*	√	√	√, √	√	7319	0.00	0.0547

Table 5-5

```
****************************************
Report : area
Design : Transmitter
Version: 2000.11
Date   : Mon Dec 10 11:28:32 2001
****************************************

Library(s) Used:

   lcbg11p (File:
/afs/engr.wisc.edu/apps/eda/flexstream.2.0/lsi_fs_2.0/lib3p/synopsys/lcbg11p/lcbg11p_l
si_wc.db)
Number of ports:            18
Number of nets:             36
Number of cells:             9
Number of references:        9
Combinational area:      2312.162598
Noncombinational area:    3275.315674
Net Interconnect area:    1732.108032

Total cell area:         5587.478516
Total area:             7319.586426
```

Table 5-6

5.4 Normal condition Receiver synthesis

Table 5-7 lists the attributes we use for synthesis under normal condition and table 5-8 lists the area report for out optimized design.

Optimization	Ungroup	Flatten	Structure (Timing,Boolean)	Boundary Optimization	Area	Slack	Figure of Merit
Original					11235	0.00	0.0356
			√, √	√	10961	0.00	0.0365
			√, X	√	10931	0.00	0.0366
		√	√, X	√	10916	0.00	0.0366
		√	√, √	√	10914	0.00	0.0366
	√	√	√, √		10900	0.00	0.0367
	√	√	√, X		10931	0.00	0.0366
*	√	√	√, X	√	10890	0.00	0.0367
	√	√	√, √	√	10908	0.00	0.0367

Table 5-7

```
****************************************
Report : area
Design : Receiver
Version: 2000.11
Date   : Mon Dec 10 10:59:31 2001
****************************************
Library(s) Used:

    lcbg11p (File:
/afs/engr.wisc.edu/apps/eda/flexstream.2.0/lsi_fs_2.0/lib3p/synopsys/lcbg11p/lcbg11p_n
om.db)
Number of ports:            18
```

```
Number of nets:              35
Number of cells:              6
Number of references:         6
Combinational area:      3418.523682
Noncombinational area:    4590.479492
Net Interconnect area:    2881.200195

Total cell area:         8009.002930
Total area:             10890.203125
```

Table 5-8

5.5 Worst-case Receiver synthesis

Table 5-9 lists the attributes we use for synthesis under worst-case and table 5-10 lists the area report for out optimized design.

Optimization	Ungroup	Flatten	Structure (Timing,Boolean)	Boundary Optimization	Area	Slack	Figure of Merit
Original					11600	0.00	0.0345
			√, √	√	11672	0.00	0.0343
			√, X	√	11640	0.00	0.0344
		√	√, X	√	11541	0.00	0.0347
		√	√, √	√	11596	0.00	0.0345
	√	√	√, √		11362	0.00	0.0352
*	√	√	√, X		11289	0.00	0.0354
	√	√	√, X	√	11634	0.00	0.0344
	√	√	√, √	√	11538	0.00	0.0347

Table 5-9

```
****************************************
```

```
Report : area
Design : Receiver
Version: 2000.11
Date   : Mon Dec 10 11:29:01 2001
****************************************
Library(s) Used:
   lcbg11p (File:
/afs/engr.wisc.edu/apps/eda/flexstream.2.0/lsi_fs_2.0/lib3p/synopsys/lcbg11p/lcbg11p_l
si_wc.db)
Number of ports:            18
Number of nets:             38
Number of cells:             9
Number of references:        7

Combinational area:      3551.589355
Noncombinational area:    4891.472656
Net Interconnect area:    2846.436035

Total cell area:         8443.062500
Total area:             11289.498047
```

Table 5-10

5.6 Discussion

Under both normal and worst cases, all types of optimization constraints will improve the design performance. The synthesis techniques we applied to our design include “ungroup”, “flatten”, “boundary optimization”, “structure timing optimization” and “structure Boolean optimization”. Since our design is a one-level non-combinational circuit, all the techniques mentioned above will improve the synthesis performance. From the quantitative table, we also found that for different design (i.e. Transmitter or Receiver) under different working conditions (i.e. normal case or worst case), the best result comes from different optimization constraints combination, but the difference is small enough to be ignored in terms of the figure of merit. It is because that for our simple design there is no such a difference among those techniques.

6 Post-Synthesis Simulation

During the synthesis process, we found out that sometimes synthesis tool is not very smart. For example, we usually separate the next-state and output logic into two different always blocks. But when a register is reset in one always block and assigned value in another block, Synopsis will consider them as wired-AND logic even if those

operations are exclusive. We also use gated clock in our design, and the timing simulation might not reflect the problems. Thus we want to simulate our design after synthesis.

After we spent some time collecting information and hacked some problems, we are able to simulate our design and proven our design functions correctly after synthesis. Following are the steps to do post-synthesis simulation with ModelSim and LSI logic in CAE environment.

6.1 Synthesized Verilog netlist extraction

After you finish the synthesis process in Synopsis, you can save your design in various formats. The default format is .db, but you can change it to .v and you will get the synthesized verilog netlist. The netlist should look like this:

```
module Parallel_Serial ( S_DATA, TX_DATA, LOAD, STALL, RESET, TXC );
input  [7:0] TX_DATA;
input  LOAD, STALL, RESET, TXC;
output S_DATA;
   wire \DATA[6] , \DATA[2] , \DATA[0] , \DATA[4] , \DATA48[3] , \DATA48[7] ,
      \DATA48[5] , \DATA48[1] , \DATA48[0] , \DATA48[4] , \DATA48[6] ,
      \DATA48[2] , \DATA[5] , \DATA[3] , \DATA[1] , \DATA[7] , n65, n66, n67,
      n68, n69, n70, n71, n72, n73, n74, n75, n76, n77, n78, n79, n80, n81,
      n82, n83, n84, n85, n86, n87, n88;
   NR2A U19 ( .Z(\DATA48[7] ), .A(n66), .B(n69) );
   N1A U20 ( .Z(n69), .A(TX_DATA[7]) );
   N1A U21 ( .Z(n72), .A(TX_DATA[6]) );
   N1A U22 ( .Z(n74), .A(TX_DATA[5]) );
   N1A U23 ( .Z(n76), .A(TX_DATA[4]) );
   N1A U24 ( .Z(n78), .A(TX_DATA[3]) );
   N1A U25 ( .Z(n80), .A(TX_DATA[2]) );
   N1A U26 ( .Z(n82), .A(TX_DATA[1]) );
   N1C U27 ( .Z(n65), .A(n68) );
   ND2B U28 ( .Z(n68), .A(LOAD), .B(n87) );
   N1A U29 ( .Z(n83), .A(TX_DATA[0]) );
   N1A U30 ( .Z(n86), .A(LOAD) );
   FD1LQA \DATA_reg[7]  ( .Q(\DATA[7] ), .D(\DATA48[7] ), .CP(TXC), .LD(n85)
       );
……
……
Endmodule
```

6.2 LSI logic Verilog modules

In the above netlist, you can see instantiations of several LSI logic gates, such as NR2A, N1A, …etc. To simulate your design, you have to get those modules as well. Those Verilog files are located in {/afs/engr.wisc.edu/apps/eda/flexstream.2.0/lsi_fs_2.0/lib3p/verilog/} and might disperse in {lcbg11p/, lcbg11p_hpb/, lcbg11p_io/, udps/} sub-directories. Copy them in your directory.

6.3 Some tricks

Although our design does not make use of negative clock edge, some flip-flops the synthesis tools choose for you are double edge-triggered. Thus the RESET period have to cover both positive and negative edge. The testbench that TA provides has a short RESET signal and you have to double it.

Another problem is trickier. Look at the following code:

```
`resetall
`timescale 1 ns / 10 ps
`celldefine

module FD1QC(Q,D,CP);
  output  Q;
  input   D, CP;

  reg notifier1, notifier2;
  wire DD, DCP;

……
`ifdef approximate
      ( posedge CP => ( Q +: D ) ) = (0.2143:0.2143:0.2143, 0.2233:0.2233:0.2233);

`else
      ( posedge CP => ( Q +: D ) ) = ( 100:100:100, 100:100:100);

`endif
……..
```

The module FD1QC, which describes a flip-flop, uses a parameter *approximate* to select the timing limit. But the first line of the module reset all the parameters! I wrote a small script that add a line {`define approximate} following {`resetall} in every file to use the correct timing information.

6.4 Compile and Simulate

Now you can put your post-synthesized verilog netlist, LSI logic gate modules, and the testbench together and recompile all of them. Make use of the .do file of ModelSim, because the LSI modules your design uses might be around a hundred, and using scripts can save you lots of time.

Figure 6-1, Figure 6-2, and Figure 6-3 shows the post-synthesis simulation of our design.

In Figure 6-1, it shows back-to-back frames are sent from transmitter and received correctly by receiver.

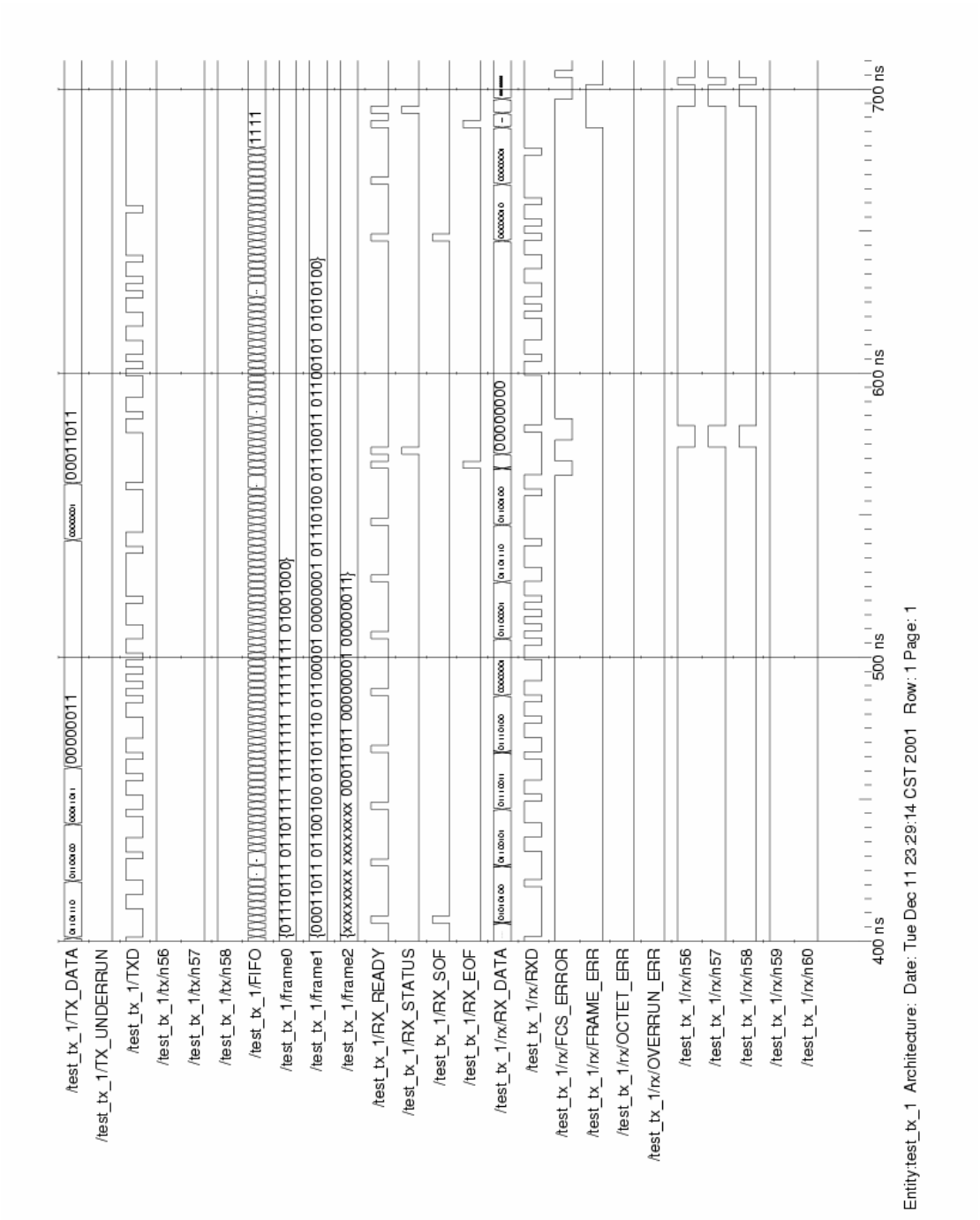


Figure 6-1

Figure 6-2 is a close look of the ending of frame two. The status byte is 00000000, which indicates a correct reception.

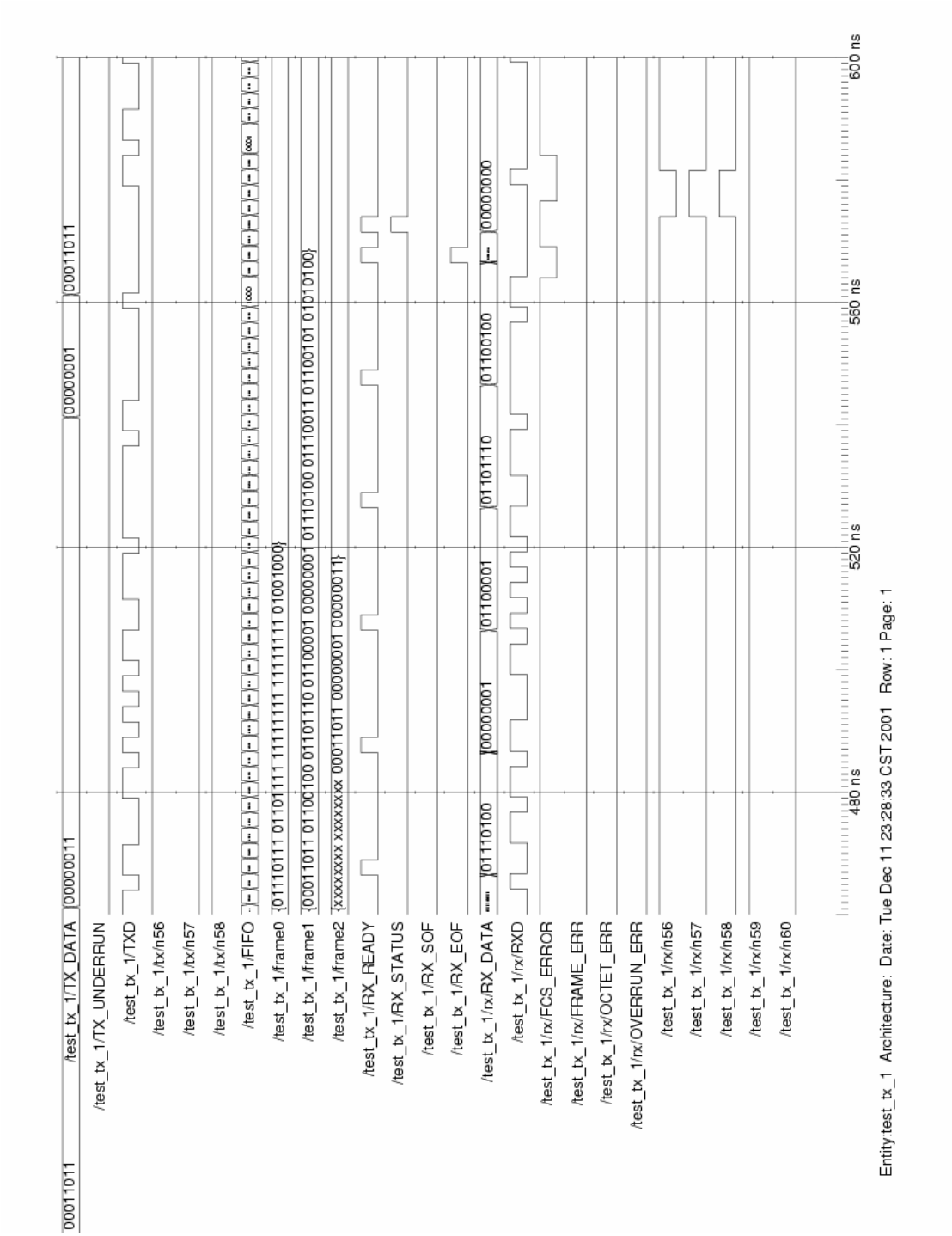


Figure 6-2

Figure 6-3 shows the end of frame three. The third frame has only three bytes, thus the status byte is 00000011, which indicates FRAME_ERROR and FCS_ERROR. The result is the same as in Chapter 4, and the reason for the FCS_ERROR is explained in Chapter 4 as well.

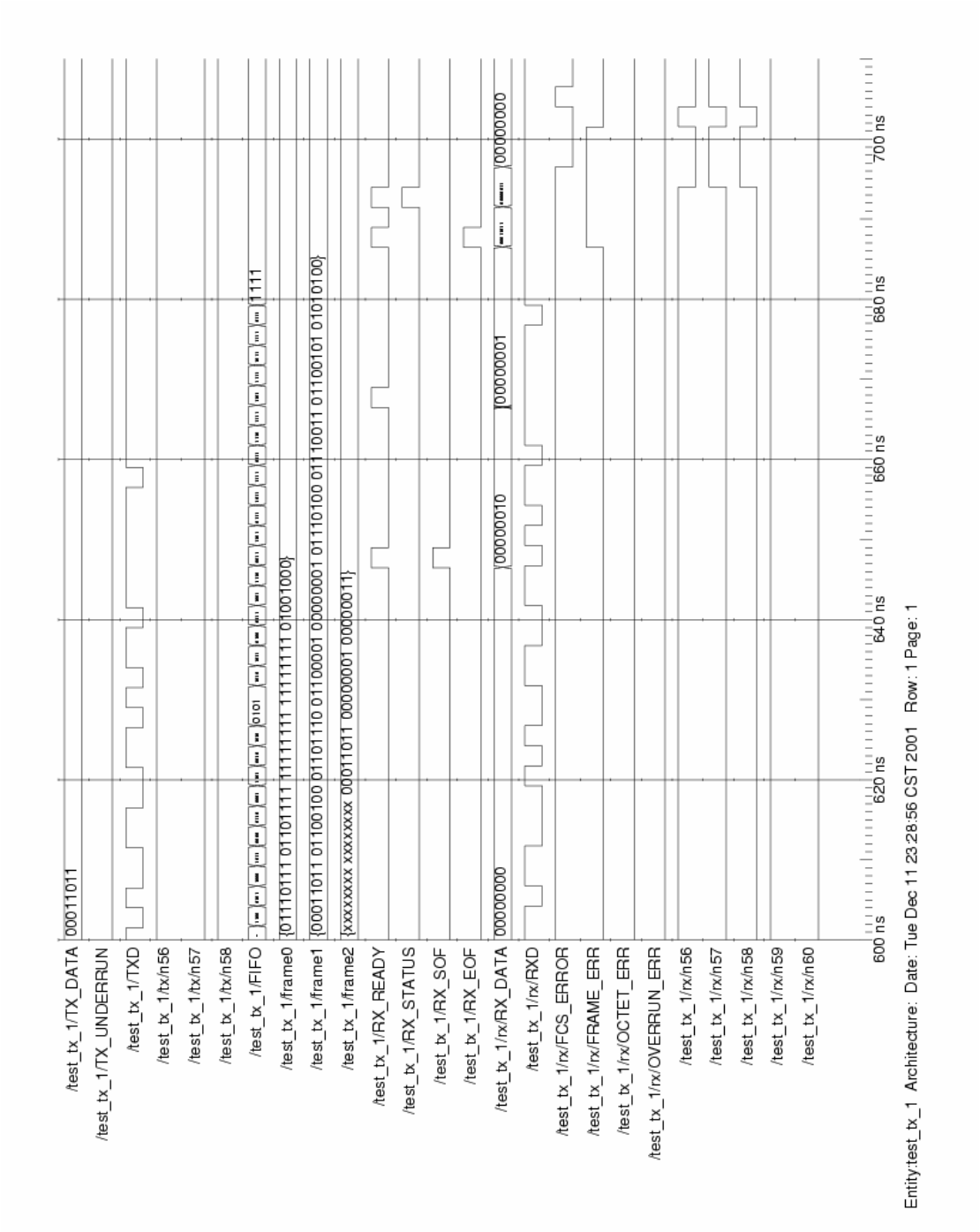


Figure 6-3

7 Discussion

In this project, we learned how to start from the specification and build a design step by step. We chose to build the function blocks first because they are the easier part. However, we try to think ahead how those blocks might communicate with the control unit and choose the more flexible way to implement their function. After the function blocks are set, we start to build the state machine of the control unit. Sometimes we find that what we reserved in the function blocks are redundant and might need to remove them, and sometimes we need to add functions to both control unit and function blocks.

We designed two sets of transmitter and receivers, and we synthesized both of them. By comparing the result, we found that different coding style can affect the speed and area of a design. There are several ways to write a more concise code. For example, reduce the control signals and storage elements. By doing so, you get a simpler logic and less storage elements, which in turn enhance the speed and reduce the area.

This project is not a big design, thus setting different attributes does not make a huge difference in term of speed and area. Detailed discussion is in Chapter 5.

It is important to do the post-synthesis simulation because simulation result and synthesized circuit do not always behave the same. At the beginning, we got some wire-AND warning from Synopsis, and we used Xilinx FPGA Express to verify our design but found that the post-synthesis result is wrong. After doing some modification on the code, we get the same results from pre-synthesis and post-synthesis.

8 Contribution

Each member contributes a lot to this project, and it is not able to make distinctions.