Appendix A. PES NET FILES

A.1 FILE DESCRIPTION

Slave Node

Slave Node – project file

File: Rx2.gdf

This file should be loaded into the FPGA of the Slave node. The address of the node is specified in the file adr2.vhd. The file RX3.GDF is the same as RX2.GDF, except for the node address (adr3.vhd).

Control Block

File: RxTxCntr.vhd

The Control block is the heart of the communication node. It manages the work of both TAXIchip™s and serves as an interface to the rest of the FPGA logic blocks. It ensures the correctness of node operation following the defined protocol. Implemented logic generates fault signals for the specific, unexpected, events.

The algorithm is based on state-machine operation. There are five basic states and one Initialization state. The mode of the node operation depends on those states, the occurring events and suitable node actions. To understand the operation of this block, refer to Chapter 5.3, Protocol Implementation.

After the power-up and the local initialization (performed by the initialization Block, after which the ‘enable_control’ line becomes active), the node is only communicationaly active until it receives the Initial command. Following the reception of this command, the ‘enable_logic’ line is set to high.

It is possible to skip the initialization process by sending a ‘reset’ command. This will take the node from any state to the Wait state.

At this point the initialization is done in a rudimentary way. The Master sends an Initialization command. The slave node receives it, goes into the Wait state, activates the ‘enable_logic’ line, and passes on the command. After this command propagates through all of the nodes, it goes back to the master node, which then becomes aware that the ring is functional. Then, the DSP can start to send relevant data.

In the future, the initialization process is where the auto-assignment of the node addresses, dead-time download, and other features should be implemented.

Timer Block

File: Timer.vhd

This timer measures (counts) the time from the beginning of the communication cycle. It is reset to zero each time a synchronization command is received. It is used for the PWM Generator and the Watchdog timer.

One 20 kHz cycle corresponds to approximately 614 CLK cycles. For Watchdog timer, we need to have twice that value available. The number 2047 was chosen since it is equal (2^11-1)

PWM Registers

File: PWM_REG.gdf

This block has a memory function. Its purpose is to store the received information, which will be used for the generation of PWM pulses. The ‘count’ input specifies to which of the nine flip-flops (FFs) the input ‘byte’ information will be written.

PWM Generator

File: PWM_gen.vhd

The PWM_gen.vhd generates the PWM signal based on two one byte long time variables: tup and tdown. It compares their values with the current 'time' (the timer output).

When 'enable' is 0, the PWM output is always 0. 'Load' is set to '1' when received data is checked and is ready to be loaded into the PWM Generator. On reception of a 'load,’ the new time variables are loaded and memorized. They are used in the next cycle that begins with the reception of 'sync,’ whose reception is indicated by a timer. The timer is reset to zero on reception of 'sync,’ If no 'load' is supplied, old values of time variables are used.

One PWM Generator block should be used per switch. In that case, the dead time is already implemented in the Master. However, if this is not the case, the same PWM Generator may be used, and an additional block should be implemented that generates the signal for the complementary switch, and at the same time implements the dead-time.

Divider

File: Div3.vhd

This file slows down the CLK before it enters the timer. This is done to scale the operational frequency of the Master and Slave, because the crystal on the DSP board and Slave board are not the same.

In this case, it slows down the CLK three times.

Flag Register

File: Flag_reg.gdf

Any fault signal that appears within one communication cycle is reported to the Master. Flag registers keep track of such occurrences. Values of the flag registers are loaded into the SENS_REG as the first byte (last to be transmitted); they are loaded one CLK cycle before it is time to send the data. In the next CLK cycle, registers are reset.

Flag_reg.gdf consists of 8 D-FF. Their D-pins are tied to VCC. CLK-pins are tied to the fault-reporting lines. As the fault is reported, FF's is set to '1'. Reset-pins are active high and are tied to the 'clear' pin of the Flag_reg.

Sensor Registers

File: Sens_reg_v.vhd

This block has a memory function. Its purpose is to store the measured data until it is time to send them to the Master node.

This is the sensor memory. All ADC outputs are loaded at the same time when 'load' becomes high. When the 'load_flag_reg' is active, the flags stored in 'flag register' will be loaded in to the 8-bit FF, byteI. The 'CNT' specifies the 8-bit FF, whose output will be active at the output of this block, as 'data_to_send.'

Watchdog Timer

File: wdt.vhd

The watchdog timer alerts the Master when data is not received properly for the period of two converter's switching periods (Tsw).

This WDT will be activated when:

- there is no 'sync' for 2 Tsw
- 'sync' is received, but there is no valid data for 2 Tsw

Initialization Block

File: Initial.vhd

This file initializes (resets) the TAXIchip™s and all of the necessary logic blocks. After the file has been loaded it starts counting and sending reset and enable commands. After it has performed its functions, it becomes inactive.

Decoder block

File: Decoder.vhd

This block performs a decoder function and a count function. The control block counts the received bytes of data and provides that value while receiving. This is a 4-bit value, which the decoder block converts into the 11 bit value, where only one bit can be active at the time. After the '1000' has been received, this block starts performing additional counter function. It counts for three more CLK cycles. After that, it deactivates until the next data package is received.

The outputs of this block are used to activate different logical blocks in the node while data is being received (e.g. the loading of Flag registers into the Sensor register, the loading of PWM registers), as well as the activation of post-reception functions (e.g. CRC check, the loading of PWM Generators).

ADC Block

File: ADC.vhd

This file simulates the A/D converters. When sensors are implemented on the board, and ADC are connected, their outputs should be connected instead of the outputs of the ADC block.

Address Block

File: Adr2.vhd

This file stores the node address. If the self-initialization algorithm with auto-address assignment is implemented as part of the initialization process, this block becomes unnecessary.

Full Transmitter

Transmitter – project file

File: Tx1full.gdf

When this file is downloaded into the FPGA, the FPGA performs the negotiations with the DSP and Master transmit functions. If only this file is downloaded into Altera, the TAXIchip™ Receiver chip should not be plugged-in to the board. Instead of using the TAXIchip™ the pins CLK (Am7969 pin 19) and CNB (Am7969 pin 24) should be short-circuited.

When this file is used as part of the Master.gdf file, both transmitter and receiver TAXIchip™s should be plugged-in to the Master board.

Control Block

File: TxCntrl.vhd

This control block is simpler than the receivers control block. Command lines that come from the DSP indicate if an appropriate command should be sent. These lines are set to low when communication Altera sets ‘REG_CLR’ to high.

After the DSP has calculated all of the data, it stores it into the DSP-Altera registers and sets the ‘FLAG’ line to high. This line remains high as long as the data is being sent. It is set to low after the communication Altera (this one) sets ‘FLG_CLR’ to high for one cycle.

In this version, some command lines are disconnected.

Register Select

File: Reg_sel.vhd

This file selects the registers in the Altera that is connected to the DSP. This Altera has stored the data to be sent. The data is organized in data banks, where each of the data banks has the data for one phase leg. In this case, there are three banks. Each bank has 10 data bytes. The first one contains the node address (bank = 0, cnt = 10), followed by the data bytes from the highest to the lowest.

The CRC should also be stored into these data banks.

Master Node

Master Node – project file

File: Master.gdf

This file performs transmitting and receiving functions of the Master block. Transmitting and receiving functions are split between two files: TX1full.gdf and RX1FULL.gdf. While the transmitting part is done fully (DSP – local Altera – communication Altera interface), for the receiving part, only the communication Altera part is programmed fully. This is the case because there is not enough space in the existing local Altera.

When this file is downloaded into the FPGA, both TAXIchip™s should be in their place.

Receiver Part

File: Rx1full.gdf

This file performs the receiving functions. It does not contain all of the blocks in rx2.gdf, which is the file for the Slave nodes. It has the following blocks: Control, Initial,

Decoder, WDT, Flag register and Timer. Only the Control block has not been described previously.

Control block

File: RxCntr.vhd

This file is programmed to perform only the receiving part of the algorithm of the rxtxcntr.vhd. It operates based on the state machine concept (Chapter 5.3). The lines from this block lead to the DSP and indicate reception of specific commands and data packages.

Since the Master node does not have a node address (multiple master operation is not implemented), reception of each of the data frames activates the 'local_adr' line. The received address of the packet (rx_adr), along with the sequence of the data byte in the frame (count), determines where in the local Altera this data will be stored.

Test Transmitters

The four-byte frame test files – project file

File: Tx1.gdf

This file is used to generate test sequences, which are used to generate the sequences used to measure the node propagation delay (Fig. 6.5). The data sequence that is sent is stored in the file tx_memm.vhd. Four data frames are sent:

- address identifier, node address, and 2-byte data for node #2
- address identifier, node address, and 2-byte data for node #3
- address identifier, node address, and 2-byte data for node #2

- address identifier, node address, and 2-byte data for node #3.

Sending is initiated about every 3μs (count3s.gdf).

This file should be used with the old file, RX2.gdf (March 98), which accepts this frame format. At the point of reception, the LED display should write 'HI,' alternating with 'VP' (node #2), and 'EC' (node #3).

The 11-byte Test Transmitter – project file

File: TestTx1.gdf

This file imitates a fully operational Master node (with no CRC implemented). It sends 11-byte long frames, whose content is stored in txf_mem.vhd.

It utilizes the tx1full.gdf file.

It should be used with the fully operational Slave nodes (file rx2.gdf, the new one).

This transmitter file, along with the designed board, can be used as a digital signal generator at speeds up to 125 Mb/s for other purposes.

A.2 VHDL FILES

File: RxTxCntr.vhd

```
--*********************************************************************
----     This is the control part of the Slave Node on the PENet
--*********************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE PENet IS
   TYPE state_type   IS (initial_state, wait_state, sync_state,
```

```
address_state, active_state, shutdown_state);
   CONSTANT initial_identifier: STD_LOGIC_VECTOR(3 DOWNTO 0):= "0001";
   --  Unique Network Synchronization Identifier:
   CONSTANT sync_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0110";
   --  Unique Address Identifier preceedes adress field:
   CONSTANT addr_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0010";
   --  Unique Fault Identifier is sent when shut down is required
   CONSTANT shutdown_identifier: STD_LOGIC_VECTOR(3 DOWNTO 0):= "0101";
   --  Unique Reset Identifier is sent by master
   CONSTANT reset_identifier  : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0111";
   --TAXI Sync Command is used to keep TAXI-Rx and TAXI-Tx synchronized
   CONSTANT taxi_sync      : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0000";
   --  Broken line Identifier: when no optical signal is present,
   --  Taxi interprets it as a command "1111"
   CONSTANT broken_line    : STD_LOGIC_VECTOR(3 DOWNTO 0):= "1111";
   --  This specifies what data is written in the memory when invalid
   --  data is received
   CONSTANT invalid_data   : STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";
   CONSTANT invalid_command: STD_LOGIC_VECTOR(3 DOWNTO 0):= "0000";
   -- Number of bytes in a package minus one
   CONSTANT number_of_data_bytes: INTEGER RANGE 0 to 8   := 8;

END PENet;
--************************************************************************
--************************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.PENet.all;

--************************************************************************
--************************************************************************


ENTITY rxtxcntr IS
      PORT
   (  -- Clock Signal comes from the TAXI-Rx
      --       Tclk=byte-time
      clk            : IN  STD_LOGIC;
      node_address   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      enable         : IN  STD_LOGIC                     := '0';
      -- Data Interface to TAXI Rx
      rx_data        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      rx_command     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0)  := "0000";
      rx_dstrb       : IN  STD_LOGIC                     := '0';
      rx_cstrb       : IN  STD_LOGIC                     := '0';
      rx_vltn        : IN  STD_LOGIC                     := '0';
      -- Data Interface to TAXI Tx
      tx_data        : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      tx_command     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)  := "0000";
      tx_strb        : OUT STD_LOGIC                     := '0';
      -- Internal Data Interface
      data_to_send   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      count_bits     : OUT INTEGER  RANGE 0 TO number_of_data_bytes:=0;
      data_received  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      local_adr      : OUT STD_LOGIC                     := '0';
      synchronize    : OUT STD_LOGIC                     := '0';
```

```
      enable_logic   : OUT STD_LOGIC                    := '0';
      -- Fault Block Interface
      wdt            : IN  STD_LOGIC                    := '0';
      vltn_fault     : OUT    STD_LOGIC                 := '0';
      protocol_fault : OUT   STD_LOGIC                  := '0';
      corrupt_data   : OUT STD_LOGIC                    := '0';
      line_break     : OUT    STD_LOGIC                 := '0';
      reset          : OUT    STD_LOGIC                 := '0';
      shutdown       : OUT    STD_LOGIC                 := '0');
END rxtxcntr;

--*********************************************************************
--*********************************************************************


ARCHITECTURE a OF rxtxcntr IS
BEGIN
PROCESS(clk)
   -- This variable keeps track of in which state the system is in
   VARIABLE state       : state_type               := initial_state;
   VARIABLE count_data  : INTEGER RANGE 0 TO number_of_data_bytes :=0;

   BEGIN
      WAIT until clk ='1';
     IF (enable = '1') THEN

--
      -- On the rising edge of the clock signal following
      -- variables are reset. This could be done on the falling
      -- edge of the clock signal, but there is enough time
      -- for the signals to be reset, before the next data
      -- arrives

      protocol_fault <= '0';
      corrupt_data   <= '0';
      vltn_fault     <= '0';
      line_break     <= '0';
      synchronize    <= '0';
      reset          <= '0';
      local_adr      <= '0';
      tx_strb        <=  '0';
      tx_command     <= "0000";
      -- TAXI-Tx gives higher priority to commands. In order
      -- to send data, command input has to be all zeroes.
      --shutdown signal resets when the system receives 'reset' command

--*********************************************************************
--    In the case Watchdog timer is activated:
--*********************************************************************

      IF (wdt = '1') THEN
         tx_command  <= shutdown_identifier;
         tx_strb     <=    '1';
         shutdown <= '1';
         state    :=  shutdown_state;
         corrupt_data <=   '1';
```

```
--*****************************************************************
--    First we check if there was a VLTN error while TAXI-Rx was
-- receiving:
--*****************************************************************


      ELSIF (rx_vltn = '1') THEN
         CASE state IS
            WHEN active_state =>
               IF (count_data = number_of_data_bytes) THEN
                  state    := wait_state;
                  count_data  := 0;
                  data_received  <= invalid_data;
                  tx_data        <= data_to_send;
                  tx_strb        <=    '1';
                  vltn_fault     <= '1';
                  corrupt_data   <= '1';
               ELSE
                  count_data  := count_data + 1;
                  data_received  <= invalid_data;
                  tx_data        <= data_to_send;
                  tx_strb        <=    '1';
                  vltn_fault     <= '1';
                  corrupt_data   <= '1';
               END IF;
            WHEN shutdown_state  =>
               vltn_fault     <= '1';
            WHEN initial_state =>
               vltn_fault     <= '1';
            WHEN OTHERS =>
               state       := wait_state;
               vltn_fault     <= '1';
         END CASE;

--*****************************************************************
--    System has correctly received a command;
--    Control Block checks for the strobe signal from a TAXI receiver:
--*****************************************************************

      ELSIF  (rx_cstrb = '1') THEN
          CASE rx_command IS
  -- Depending on a state the system is in, Block does the appropriate:

            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
            --    Synchronization
            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            WHEN sync_identifier =>
               CASE state IS
                  WHEN wait_state   =>
                     state    :=    sync_state;
                     tx_command  <= rx_command;
                     tx_strb     <=    '1';
                  -- Fault cases:
                  WHEN sync_state =>
                     state       :=    sync_state;
                     tx_command     <= rx_command;
```

```
            tx_strb        <=    '1';
            protocol_fault    <=    '1';
         WHEN address_state =>
            state       :=    sync_state;
            tx_command     <= rx_command;
            tx_strb        <=    '1';
            protocol_fault    <=    '1';
         WHEN active_state =>
            IF (count_data = number_of_data_bytes) THEN
               state    := wait_state;
               count_data  := 0;
               data_received  <= invalid_data;
               tx_data        <= data_to_send;
               tx_strb        <=    '1';
               protocol_fault <= '1';
               corrupt_data   <= '1';
            ELSE
               count_data  := count_data + 1;
               data_received  <= invalid_data;
               tx_data        <= data_to_send;
               tx_strb        <=    '1';
               protocol_fault <= '1';
               corrupt_data   <= '1';
            END IF;
         WHEN OTHERS =>
            tx_command     <= rx_command;
            tx_strb        <=    '1';
         END CASE;

      -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
      --    Address Identifier
      -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

         WHEN addr_identifier =>
            CASE state IS
            WHEN wait_state =>
               tx_command  <= rx_command;
               tx_strb     <=    '1';
               state    := address_state;
            -- Fault cases:
            WHEN address_state   =>
               tx_command  <= rx_command;
               tx_strb     <=    '1';
               protocol_fault <= '1';
            WHEN sync_state =>
               tx_command  <= rx_command;
               tx_strb     <=    '1';
               state    := address_state;
               protocol_fault <= '1';
            WHEN active_state =>
               IF (count_data = number_of_data_bytes) THEN
                  state    := wait_state;
                  count_data  := 0;
                  data_received  <= invalid_data;
                  tx_data        <= data_to_send;
                  tx_strb        <=    '1';
                  protocol_fault <= '1';
```

```
            corrupt_data   <= '1';
         ELSE
            count_data  := count_data + 1;
            data_received  <= invalid_data;
            tx_data        <= data_to_send;
            tx_strb        <=    '1';
            protocol_fault <= '1';
            corrupt_data   <= '1';
         END IF;
      WHEN OTHERS =>
         tx_command     <= rx_command;
         tx_strb        <=    '1';
   END CASE;

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Shutdown
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN shutdown_identifier =>
   tx_command  <= shutdown_identifier;
   tx_strb     <=    '1';
   shutdown <= '1';
   state    :=  shutdown_state;
   corrupt_data <=   '1';

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Reset
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN reset_identifier =>
   tx_command     <= rx_command;
   tx_strb        <=    '1';
   reset       <= '1';
   shutdown    <= '0';
   vltn_fault     <= '0';
   corrupt_data   <= '0';
   protocol_fault <= '0';
   count_data     :=  0 ;
   state       := wait_state;

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Taxi Sync
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN taxi_sync =>
   tx_strb     <=    '0';

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Broken Line
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN broken_line =>
      CASE state IS
      WHEN initial_state   =>
         state       := initial_state;
      WHEN OTHERS =>
         shutdown    <= '1';
```

```
                  corrupt_data   <= '1';
                  line_break     <= '1';
                  state       := shutdown_state;
               END CASE;

            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
            --    Initialization
            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            WHEN initial_identifier =>
               tx_command     <= rx_command;
               tx_strb        <=    '1';
               enable_logic   <=    '1';
               state       :=  wait_state;

            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
            --  Unexpected Command
            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            WHEN OTHERS =>
               --protocol_fault  <= '1';
         END CASE;

--*****************************************************************
--    System receives data
--*****************************************************************


      ELSIF (rx_dstrb = '1') THEN
         CASE state IS
            WHEN wait_state =>
               tx_data     <= rx_data;
               tx_strb     <=    '1';
            WHEN sync_state   =>
               tx_data     <= rx_data;
               tx_strb     <=    '1';
               IF (rx_data = node_address) THEN
                  synchronize <= '1';
                  state    := wait_state;
               END IF;
            WHEN address_state =>
               IF (rx_data = node_address) THEN
                  state    := active_state;
                  tx_data     <= rx_data;
                  tx_strb     <=    '1';
                  local_adr   <= '1';
               ELSE
                  state    := wait_state;
                  tx_data     <= rx_data;
                  tx_strb     <=    '1';
               END IF;
            WHEN active_state =>
               IF (count_data = number_of_data_bytes) THEN
                  state    := wait_state;
                  count_data  := 0;
                  data_received  <= rx_data;
                  tx_data        <= data_to_send;
```

```
                  tx_strb        <=    '1';
               ELSE
                  state    := active_state;
                  count_data  := count_data + 1;
                  data_received  <= rx_data;
                  tx_data        <= data_to_send;
                  tx_strb        <=    '1';
               END IF;
            WHEN OTHERS =>
               tx_data     <= rx_data;
               tx_strb     <=    '1';
         END CASE;
      END IF; -- wdt, vltn, commands, data
--*****************************************************************

     END IF; -- enable
     count_bits   <= count_data;
END PROCESS;
END a;
```

File: Timer.vhd

```
-- This timer measures (counts) the time from the beginning
-- of the communication cycle. It is reset to zero each
-- time synchronization command is received.
-- It is used for PWM Generator and Watchdog Timer.

-- One 20 kHz cycle corresponds to approximately 614 clk cycles.
-- For watchdog timer, we need to have twice that value available.
-- 2047 was chosen, since it equals (2^11-1)


ENTITY timer IS

   PORT
   (
      clk      : IN  BIT;
      clear : IN  BIT;
      enable   : IN  BIT;
      q     : OUT    INTEGER RANGE 0 TO 2047
   );

END timer;

ARCHITECTURE a OF timer IS
BEGIN
   PROCESS (clk)
      VARIABLE cnt         : INTEGER RANGE 0 TO 2047;
   BEGIN
      IF clear = '1' THEN
            cnt := 0;
      ELSIF (clk'EVENT AND clk = '1') THEN
```

```
         IF enable = '1' THEN
               IF cnt = 2047 THEN
                  cnt := 0;
               ELSE
                  cnt := cnt + 1;
               END IF;
         END IF;
      END IF;

      q  <= cnt;

   END PROCESS;
END a;
```

File: PWM_gen.vhd

```
--
-- This file is the part of the rx.vhd

-- The PWM_gen.vhd generates the pwm signal based on two, one byte
-- long, time variables: tup and tdown. It compares the value of those
-- two with the current 'time' (output of the counter that is reset on
-- sync). One pwm_gen should be used per switch.
-- When 'enable' is 0, the pwm output is always 0
-- 'Load' is set to '1' when received data is checked and is ready to
-- be loaded into PWM Generator.
-- On reception of 'load' new time variables are loaded, and memorized.
-- They are used in the next cycle that begins with reception of
-- 'sync', whose reception is indicated by timer. Timer is reseted to
-- zero, on reception of 'sync'.
-- If no 'load' is supplied, old values of time variables are used.
--                                    Ivana 9/7/98

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY PWM_GEN IS

   PORT
   (  clk                  : IN STD_LOGIC;
      new_cycle            : IN STD_LOGIC;
      -- 'time' is the counter output
      time              : IN  STD_LOGIC_VECTOR(7 downto 0);
      -- If enable=0 outputs are always 0
      enable               : IN  STD_LOGIC   := '0';
      -- PWM parameters are loaded after the packet is correctly
      -- received
      load              : IN  STD_LOGIC   := '0';
      -- PWM parameters for each of the switches
      t_up           : IN  STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      t_down         : IN  STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      -- Generated PWM signals for switches
      pwm_out        : OUT STD_LOGIC   := '0';
      error_input    : OUT   STD_LOGIC := '0'
```

```
   );

END PWM_GEN;

--*********************************************************************
--*********************************************************************

ARCHITECTURE a OF PWM_GEN IS
-- PWM signal
   SIGNAL
      pwm               : STD_LOGIC;
-- Stored PWM parameters:
   SIGNAL
      t_up_l, t_down_l  : STD_LOGIC_VECTOR(7 DOWNTO 0);
-- PWM parameters for current cycle (values changed when time=0)
   SIGNAL
      t_u, t_d       : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
   PROCESS(clk)
      VARIABLE compare  : STD_LOGIC := '0';
   BEGIN
      WAIT UNTIL clk = '1';

--*********************************************************************
--*****************************************************************

      IF (enable = '0') THEN
         pwm   <= '0';
      ELSE

--*********************************************************************
--          New cycle begins
--*********************************************************************

         IF (new_cycle = '1') THEN
            IF (t_up_l = "00000000" and t_down_l = "00000000") THEN
               pwm      <= '0';
               compare := '0';
            ELSIF (t_up_l = "00000000" ) THEN
               pwm      <= '1';
               compare := '1';
            ELSIF (t_down_l = "00000000") THEN
               pwm      <= '0';
               compare := '1';
            ELSIF (t_up_l > t_down_l) THEN
               pwm      <= '1';
               compare := '1';
            ELSE
               pwm      <= '0';
               compare := '1';
            END IF;
            t_u      <= t_up_l;
            t_d      <= t_down_l;

--*****************************************************************
--       Regular operation (waiting for T-up and T-down)
--*****************************************************************
```

```
         ELSIF (time = t_u and compare = '1') THEN
             pwm  <= '1';
         ELSIF (time = t_d and compare = '1') THEN
            pwm   <=  '0';
         ELSE
            pwm   <= pwm;
         END IF;
      END IF;

--*****************************************************************
--       New T-up and T-down data has arrived:
--*****************************************************************

      IF (load = '1') THEN -- current input values are stored
         IF    (t_up = t_down) THEN
            error_input <=    '1'; -- in case of error, old values are
used
         ELSE
            error_input <= '0';
            t_up_l      <= t_up;
            t_down_l <= t_down;
         END IF;
      END IF;

--*****************************************************************
--*****************************************************************

   END PROCESS;
   pwm_out  <= pwm;
END a;
```

File: Div3.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY div3 IS
   PORT
   (
      clk_in               : IN  STD_LOGIC;
      clear             : IN  STD_LOGIC;
      clk_out              : OUT STD_LOGIC
   );
END div3;

ARCHITECTURE a OF div3 IS
BEGIN

   PROCESS (clk_in, clear)
      VARIABLE cnt   : INTEGER RANGE 1 TO 3  := 1;
   BEGIN

      IF clear = '1' THEN
```

```
         clk_out  <= '0';
         cnt      := 1;
      ELSIF (clk_in'EVENT AND clk_in = '1') THEN
         IF cnt = 3 THEN
            clk_out  <=    '1';
            cnt      := 1;
         ELSE
            clk_out  <= '0';
            cnt      := cnt + 1;
         END IF;
      END IF;

   END PROCESS;
END a;
```

File: Sens_reg_v.vhd

```
-- This is the sensor memory. All ADC outputs are loaded at
-- the same time when 'load' becomes high. When the 'load_flag_reg'
-- is active, the flags stored in 'flag register' will be loaded
-- in to the 8-bit FF, byteI. The 'CNT' specifies the
-- 8-bit FF, whose output will be active at the output of this block,
-- as 'data_to_send'.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sens_reg_v IS
   PORT
   (  clk            : IN  STD_LOGIC;
      cnt            : IN  STD_LOGIC_VECTOR(3 downto 0);
      load        : IN  STD_LOGIC;
      load_flag_reg  : IN  STD_LOGIC;
      byteI       : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteII         : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteIII        : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteIV         : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteV       : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteVI         : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteVII        : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteVIII    : IN  STD_LOGIC_VECTOR(7 downto 0);
      byteIX         : IN  STD_LOGIC_VECTOR(7 downto 0);
      data_to_send   : OUT    STD_LOGIC_VECTOR(7 downto 0));

END sens_reg_v;

ARCHITECTURE a OF sens_reg_v IS
   SIGNAL byteIs, byteIIs, byteIIIs, byteIVs, byteVs,
          byteVIs, byteVIIs, byteVIIIs, byteIXs :  STD_LOGIC_VECTOR(7
downto 0);
BEGIN
   PROCESS(clk)
   BEGIN
   IF (clk'EVENT and clk='1') THEN
```

```
      IF load = '1' THEN
         byteIXs     <= byteIX;
         byteVIIIs   <= byteVIII;
         byteVIIs <= byteVII;
         byteVIs     <= byteVI;
         byteVs      <= byteV;
         byteIVs     <= byteIV;
         byteIIIs <= byteIII;
         byteIIs  <= byteII;
      END IF;
      IF load_flag_reg = '1' THEN
         byteIs   <= byteI;
      END IF;
      CASE  cnt      IS
         WHEN "0000" =>
            data_to_send   <= byteIXs;
         WHEN "0001" =>
            data_to_send   <= byteVIIIs;
         WHEN "0010" =>
            data_to_send   <= byteVIIs;
         WHEN "0011" =>
            data_to_send   <= byteVIs;
         WHEN "0100" =>
            data_to_send   <= byteVs;
         WHEN "0101" =>
            data_to_send   <= byteIVs;
         WHEN "0110" =>
            data_to_send   <= byteIIIs;
         WHEN "0111" =>
            data_to_send   <= byteIIs;
         WHEN "1000" =>
            data_to_send   <= byteIs;
         WHEN OTHERS =>
            data_to_send   <= "00000000";
      END CASE;
      END IF;
   END PROCESS;

END a;
```

File: wdt.vhd

```
-- ******************************************************
-- ******            Watchdog Timer                  ******
-- ******           Ivana Milosavljevic            ******
-- ******************************************************

-- Watchdog timer alerts the master in the case when data
-- was not properly received for the period of two converter's
-- switching periods.
-- Tsw is the length of a converter's switching period.
-- This WDT will be activated when:
--   * there is no 'sync' for 2 Tsw
```

```
--  * 'sync' is received, but there was no valid data for 2 Tsw

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY wdt IS

   PORT
   (
      clk            : IN  STD_LOGIC;
      enable         : IN  STD_LOGIC   := '1';
      reset       : IN  STD_LOGIC   := '0';
      sync        : IN  STD_LOGIC   := '0';
      local_address  : IN  STD_LOGIC   := '0';
      error_input    : IN  STD_LOGIC   := '0';
      time        : IN  STD_LOGIC_VECTOR(10 downto 0) :=  "00000000000"
;
      wdt_out        : OUT STD_LOGIC   := '0'
   );

END wdt;

ARCHITECTURE a OF wdt IS

   SIGNAL   local_adr, missing_loc_adr, wdt : STD_LOGIC;

BEGIN

   PROCESS (clk)
   -- Tsw, switching period of the converter, is 614*Tclk
   -- Tsw2 is set to 1240*Tclk
   CONSTANT Tsw2   : STD_LOGIC_VECTOR(10 downto 0) := "10011011000";
   BEGIN
    IF (clk'EVENT AND clk = '1') THEN
     IF enable = '1' THEN

   --   * regular behaviour
      IF reset = '1' THEN
         local_adr <= '0';
         missing_loc_adr   <= '0';
         wdt   <= '0';
      END IF;
      IF local_address = '1' THEN
         local_adr   <= '1';
      END IF;
      IF (sync = '1' and local_adr = '1' and error_input = '0') THEN
         local_adr   <= '0';
         missing_loc_adr   <= '0';
      END IF;

   --   * there is no 'sync' for 2 Tsw
      IF time > Tsw2 THEN -- 'sync' didn't arrive
         wdt   <= '1';
      END IF;
   --  * 'sync' is received, but there was no valid data for 2 Tsw
      IF (sync = '1' and (local_adr = '0' or error_input = '1')) THEN
         IF missing_loc_adr = '0' THEN
```

```
            missing_loc_adr   <= '1';
         ELSE
            wdt   <= '1'; -- data is missing for 2 consecutive cycles
            missing_loc_adr   <= '0';
         END IF;
      END IF;

      END IF;
    END IF;
   END PROCESS;

   wdt_out <= wdt;

END a;
```

File: Initial.vhd

```
-- This file initializes (resets) the TAXIchips and
-- all of the necessary logic blocks.
-- After the file has been loaded it starts counting,
-- and sending reset and enable commands. After it has
-- performed its functions, it becomes inactive.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY initial IS

   PORT
   (
      clk                  : IN  STD_LOGIC;
      clear             : IN  STD_LOGIC   := '0';
      enable_control       : OUT STD_LOGIC;
      initial_reset        : OUT STD_LOGIC
   );

END initial;

ARCHITECTURE a OF initial IS

   SIGNAL   count: INTEGER RANGE 0 TO 15  := 0;

BEGIN

   PROCESS (clk, clear)
   BEGIN

      IF clear = '1' THEN
         count <= 0;
      ELSIF (clk'EVENT AND clk= '1') THEN
         IF count = 10 THEN
```

```
            initial_reset  <= '1';
         ELSIF count = 11 THEN
            initial_reset  <= '0';
         ELSIF count = 14 THEN
            enable_control <= '1';
         END IF;
         IF count < 15 THEN
            count <= count + 1;
         END IF;
      END IF;

   END PROCESS;

END a;
```

File: Decoder.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY decoder IS
   PORT
   (
      clk               : IN STD_LOGIC;
      s              : IN INTEGER RANGE 0 TO 8;
      count          : OUT STD_LOGIC_VECTOR(10 downto 0)
   );
END decoder;

ARCHITECTURE a OF decoder IS
BEGIN
PROCESS (clk)
   VARIABLE continue_count: INTEGER RANGE 0 to 3   := 0;

  BEGIN
    WAIT UNTIL clk = '0';
     CASE s IS
      WHEN 0   =>
         IF continue_count = 0 THEN
             count   <=    "00000000001";
         ELSIF continue_count = 3 THEN
            count <= "01000000000";
            continue_count := 2;
         ELSIF continue_count = 2 THEN
            count <= "10000000000";
            continue_count := 1;
         ELSIF continue_count = 1 THEN
            count <=    "00000000000";
            continue_count := 0;
         END IF;
      WHEN 1   =>
          count <= "00000000010";
      WHEN 2   =>
          count <= "00000000100";
```

```
      WHEN 3   =>
          count <= "00000001000";
      WHEN 4   =>
          count <= "00000010000";
      WHEN 5   =>
          count <= "00000100000";
      WHEN 6   =>
          count <= "00001000000";
      WHEN 7   =>
          count <= "00010000000";
      WHEN 8   =>
          count <= "00100000000";
         continue_count := 3;
     END CASE;
   END PROCESS;
END a;
```

File: ADR2.vhd

```
--
-- This file is assigns the address node.
--


ENTITY adr2 IS
   PORT
   (  qn       : OUT    INTEGER RANGE 0 TO 255);

END adr2;

ARCHITECTURE a OF adr2  IS
BEGIN

   qn <= 2;

END a;
```

File: ADC.vhd

```
--
-- This file is for testing purpose. It simulates the
-- outputs of A/D converters.
--


ENTITY adc IS
   PORT
   (  qnIX, qnVIII, qnVII, qnVI,
      qnV, qnIV, qnIII, qnII     : OUT    INTEGER RANGE 0 TO 255);

END adc ;
```

```
ARCHITECTURE a OF adc  IS
BEGIN
   qnIX  <= 2;
   qnVIII   <=    8;
   qnVII <= 248;
   qnVI  <= 100;
   qnV      <= 2;
   qnIV  <= 8;
   qnIII <= 248;
   qnII  <= 100;
END a;
```

File: Tx_Cntrl.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE PENet IS
   TYPE state_type   IS (initial_state, wait_state, sync_state,
address_state, active_state, shutdown_state);
   CONSTANT initial_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0)
   := "0001";
   --  Unique Network Synchronization Identifier:
   CONSTANT sync_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0110";
   --  Unique Address Identifier preceedes adress field:
   CONSTANT addr_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0010";
   --  Unique Fault Identifier is sent when shut down is required
   CONSTANT shutdown_identifier  : STD_LOGIC_VECTOR(3 DOWNTO 0):=
"0101";
   --  Unique Reset Identifier is sent by master
   CONSTANT reset_identifier  : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0111";
   --  TAXI Sync Command is used to keep TAXI-Rx and TAXI-Tx
   -- synchronized
   CONSTANT taxi_sync         : STD_LOGIC_VECTOR(3 DOWNTO 0):= "0000";
   --  Broken line Identifier: in the case of no optical signal
   -- present,
   --  Taxi interprets silence as a command "1111"
   CONSTANT broken_line    : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"1111";
   --  This specifies what data is written in the memmory when invalid
data is received
   CONSTANT invalid_data      : STD_LOGIC_VECTOR(7 DOWNTO 0)      :=
"00000000";
   CONSTANT invalid_command   : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"0000";
   -- Number of bytes in a package minus 1 (without CRC it is 8):
   CONSTANT number_of_data_bytes: INTEGER RANGE 0 to 8            := 8;

END PENet;

-- ************************************************************
-- ************************************************************
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.PENet.all;

ENTITY tx_cntrl IS
   PORT
   (  clk            : IN  STD_LOGIC;
      enable         : IN  STD_LOGIC   := '1';

      flag        : IN    STD_LOGIC := '0';
      flag_reset     : OUT    STD_LOGIC   := '0';
      last_byte      : IN  STD_LOGIC   := '0';
      enable_sel_reg : OUT STD_LOGIC;
      count       : IN  INTEGER range 0 to 10;

      reset       : IN    STD_LOGIC := '0';
      sync        : IN    STD_LOGIC := '0';
      shutdown    : IN    STD_LOGIC := '0';
      initialize     : IN  STD_LOGIC   := '0';

      data_in        : IN  STD_LOGIC_VECTOR(7 downto 0);

      tx_strb        : OUT    STD_LOGIC;
      data_to_send   : OUT    STD_LOGIC_VECTOR(7 downto 0);
      command_to_send   : OUT   STD_LOGIC_VECTOR(3 downto 0);
      com_reg_reset  : OUT STD_LOGIC);
-- ********************************************************************

-- 'flag'   is active high as data needs to be sent. It is reset after
-- all registers were emptied.
-- ********************************************************************
-- 'count'  is output of the counter that indicates the number of data
-- byte in a sequence. When counter is 0 and 'flag' is 1, address
-- identifier should be sent. Outside counter counts (and selects)
-- register banks (one per slave node)
-- ********************************************************************
-- 'shutdown', 'reset', 'sync' and 'initalize' are command lines. If
-- active, they indicate that a command should be sent down the line
-- ********************************************************************

END tx_cntrl;

--********************************************************************
--********************************************************************

ARCHITECTURE a OF tx_cntrl IS
   BEGIN
PROCESS(clk)

   VARIABLE cnt                              : INTEGER RANGE 0 to 3  :=
   0;
   VARIABLE do_reset, do_flag_reset, send_data     : STD_LOGIC := '0';

   BEGIN
         WAIT UNTIL clk = '1';
-- ********************************************************************
    IF (enable = '1') THEN
```

```
-- ********************************************************************

      IF (do_flag_reset = '1') THEN
         flag_reset     <=  '0';
         do_flag_reset  := '0';
         tx_strb     <= '0';
         command_to_send   <= "0000";
         data_to_send   <= "00000000";

      ELSIF (do_reset = '1') THEN
         com_reg_reset  <= '0';
         do_reset    := '0';
         tx_strb     <= '0';
         command_to_send   <= "0000";
         data_to_send   <= "00000000";

-- ********************************************************************
--    Master is requesting reset of the whole ring.
-- ********************************************************************

      ELSIF (reset = '1') THEN
         command_to_send   <= reset_identifier;
         data_to_send   <= "00000000";
         tx_strb     <= '1';
         com_reg_reset  <= '1';
         cnt            :=  0;
         do_reset       := '1';

--****************************************************************
--    Master is requesting converter shutdown.
--****************************************************************

      ELSIF (shutdown = '1') THEN
         command_to_send   <= shutdown_identifier;
         data_to_send   <= "00000000";
         tx_strb     <= '1';
         com_reg_reset  <= '1';
         cnt            :=  0;
         do_reset       := '1';

--********************************************************************
--       Master is requesting ring initialization.
--    Ring has to be initizlized, before it becomes operational.
-- ********************************************************************

      ELSIF (initialize = '1') THEN
         command_to_send   <= initial_identifier;
         data_to_send   <= "00000000";
         tx_strb     <= '1';
         com_reg_reset  <= '1';
         cnt            :=  0;
         do_reset       := '1';

-- ****************************************************************
--    The synchronization sequence is being sent (without implemented
-- delay); For the delay to be implemented cnt should be range 0 to 13,
-- and addresses would be sent on 13 (address of the third node), 7
```

```
--(address of the second)and 1 (address of the first node in a chain).
-- On each CLK, cnt would be decremented by one.
--*****************************************************************

      ELSIF (cnt > 0) THEN
         IF  cnt = 3 THEN
            data_to_send   <= "00000011";
            command_to_send   <= "0000";
            tx_strb        <= '1';
            cnt            :=  cnt - 1;
         ELSIF cnt = 2 THEN
            data_to_send   <= "00000010";
            com_reg_reset  <= '0';
            command_to_send   <= "0000";
            tx_strb        <= '1';
            cnt            :=  cnt - 1;
         ELSE
            data_to_send   <= "00000001";
            command_to_send   <= "0000";
            tx_strb        <= '1';
            cnt            :=  0;
         END IF;

-- ***************************************************************
--       Data is read from registers and send to TAXI
-- ***************************************************************

      ELSIF (send_data = '1') THEN
         IF last_byte = '1' THEN
            send_data      := '0';
            flag_reset     <= '1';
            tx_strb     <= '0';
            command_to_send   <= "0000";
            data_to_send   <= "00000000";
            do_flag_reset  := '1';
         ELSIF count = 10 THEN
            command_to_send   <= addr_identifier;
            data_to_send   <= "00000000";
            tx_strb     <= '1';
         ELSE
            data_to_send   <= data_in;
            tx_strb        <= '1';
            command_to_send   <= "0000";
         END IF;

--*****************************************************************
--       Master wrote the data for the next switching cycle into the
registers.
--    Now Alters should read data from master and send it to TAXI.
-- ***************************************************************

      ELSIF flag = '1' THEN
         send_data   := '1';

-- ***************************************************************
--    Master is initiating synchronizaiton sequence.
--*****************************************************************
```

```
      ELSIF sync = '1' THEN
         com_reg_reset  <= '1';
         cnt         :=     3;
         data_to_send   <= "00000000";
         command_to_send   <= sync_identifier;
         tx_strb     <= '1';

-- ************************************************************
--       If nothing else is active, then...
--************************************************************

      ELSE
         tx_strb     <= '0';
         command_to_send   <= "0000";
         data_to_send   <= "00000000";
      END IF;

-- ***************************************************
-- ***************************************************

     END IF;
   enable_sel_reg <= send_data;
   END PROCESS;
END a;
```

File: Reg_sel.vhd

```
-- This file selects the registers in the Altera that is connected
-- to the DSP. That Altera has stored the data to-be-sent .
-- The data is organized in data banks, where each of the data banks
-- has the data for one phase leg. In this case, there are three banks.
-- Each bank has 10 data bytes. First one contains the node address
-- (bank = 0, cnt = 10), followed by the data bytes from the highest to
-- the lowest.
-- The CRC should also be stored into these data banks.


ENTITY reg_sel IS

   PORT
   (
      clk         : IN  BIT;
      reset    : IN  BIT;
      enable      : IN  BIT;
      q        : OUT    INTEGER RANGE 0 to 10   := 10;
      bank_select : OUT    INTEGER RANGE 0 to 2 :=  0;
      last_byte   : OUT BIT                  := '0'
   );

END reg_sel;

ARCHITECTURE a OF reg_sel IS
BEGIN
```

```
   PROCESS (clk)
      VARIABLE cnt         : INTEGER RANGE 0 TO 10 := 10;
      VARIABLE    bank     : INTEGER RANGE 0 to 2  := 0;
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
         last_byte   <= '0';
         IF reset = '1' THEN
            cnt  := 10;
            bank := 0;
         ELSIF enable = '1' THEN
            IF cnt = 0 THEN
               IF bank = 2 THEN
                  bank  :=    0;
                  last_byte   <= '1';
                  cnt      := 10;
               ELSE
                  bank  := bank + 1;
                  cnt      := 10;
               END IF;
            ELSE
               cnt   := cnt - 1;
            END IF;
         END IF;
      END IF;
         bank_select <= bank;
         q        <= cnt;
   END PROCESS;
END a;
```

File: RxCntr.vhd

```
--*********************************************************************
----     This is the control part of the Slave Node on the PENet
--*********************************************************************

LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE PENet IS
   TYPE state_type   IS (initial_state, wait_state, sync_state,
address_state, active_state, shutdown_state);
   CONSTANT initial_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0)
   := "0001";
   --  Unique Network Synchronization Identifier:
   CONSTANT sync_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"0110";
   --  Unique Address Identifier preceedes adress field:
   CONSTANT addr_identifier   : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"0010";
   --  Unique Fault Identifier is sent when shut down is required
   CONSTANT shutdown_identifier  : STD_LOGIC_VECTOR(3 DOWNTO 0)   :=
"0101";
   --  Unique Reset Identifier is sent by master
   CONSTANT reset_identifier  : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
```

```
"0111";
   --  TAXI Sync Command is used to keep TAXI-Rx and TAXI-Tx
synchronized
   CONSTANT taxi_sync         : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"0000";
   --  Broken line Identifier: in the case of no optical signal
present,
   --  Taxi interprets silence as a command "1111"
   CONSTANT broken_line    : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"1111";
   --  This specifies what data is written in the memmory when invalid
data is received
   CONSTANT invalid_data      : STD_LOGIC_VECTOR(7 DOWNTO 0)      :=
"00000000";
   CONSTANT invalid_command   : STD_LOGIC_VECTOR(3 DOWNTO 0)      :=
"0000";
   -- Number of bytes in a package minus one
   CONSTANT number_of_data_bytes: INTEGER RANGE 0 to 8            := 8;

END PENet;
-- ****************************************************************
-- ****************************************************************


LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.PENet.all;

--****************************************************************
--****************************************************************


ENTITY rxcntr IS
      PORT
   (  -- Clock Signal comes from the TAXI-Rx
      --       Tclk=byte-time
      clk            : IN  STD_LOGIC;
      enable         : IN  STD_LOGIC                  := '0';
      -- Data Interface to TAXI Rx
      rx_data        : IN  STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      rx_command        : IN  STD_LOGIC_VECTOR(3 DOWNTO 0)  := "0000";
      rx_dstrb          : IN  STD_LOGIC                  := '0';
      rx_cstrb          : IN  STD_LOGIC                  := '0';
      rx_vltn           : IN  STD_LOGIC                  := '0';
      -- Data Interface to the DSP
      data_received  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)  := "00000000";
      count_bits     : OUT INTEGER  RANGE 0 TO 9      := 0;
      rx_adr         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0):=   "00000000";
      shutdown       : OUT    STD_LOGIC               :=  '0';
      initialize     : OUT STD_LOGIC                  :=  '0';
      -- Internal Data Interface
      local_adr      : OUT STD_LOGIC                  := '0';
      synchronize    : OUT STD_LOGIC                  := '0';
      enable_logic   : OUT    STD_LOGIC               := '0';
      -- Fault Block Interface
      wdt                  : IN  STD_LOGIC           := '0';
      vltn_fault           : OUT    STD_LOGIC        := '0';
```

```
      protocol_fault       : OUT   STD_LOGIC                := '0';
      corrupt_data         : OUT STD_LOGIC                  := '0';
      line_break           : OUT    STD_LOGIC               := '0';
      reset                : OUT    STD_LOGIC               := '0'
      );
END rxcntr;

--*********************************************************************
--*********************************************************************

ARCHITECTURE a OF rxcntr IS
BEGIN
PROCESS(clk)
   -- This variable keeps track of in which state the system is in
   VARIABLE state       : state_type      := initial_state;
   VARIABLE count_data  : INTEGER RANGE 0 TO number_of_data_bytes := 0;

   BEGIN
      WAIT until clk ='1';
     IF (enable = '1') THEN

--*************************************************************

      -- On the rising edge of the clock signal following
      -- variables are reseted. This could be done on the falling
      -- edge of the clock signal, but there is enough time
      -- for the signals to be reseted, before the next data
      -- arrives

      protocol_fault <= '0';
      corrupt_data   <= '0';
      vltn_fault     <= '0';
      line_break     <= '0';
      synchronize    <= '0';
      reset          <= '0';
      local_adr      <= '0';
--*************************************************************
--    In the case Watchdog timer is activated:
--*************************************************************

      IF (wdt = '1') THEN
         shutdown <= '1';
         state    :=  shutdown_state;
         corrupt_data <=   '1';

-- ************************************************************
-- First we check if there was a VLTN error while TAXI-Rx was receiving
-- ************************************************************

      ELSIF (rx_vltn = '1') THEN
         CASE state IS
            WHEN active_state =>
               state    := wait_state;
               count_data  := 0;
               rx_adr      <= "00000000";
               data_received  <= invalid_data;
               vltn_fault     <= '1';
```

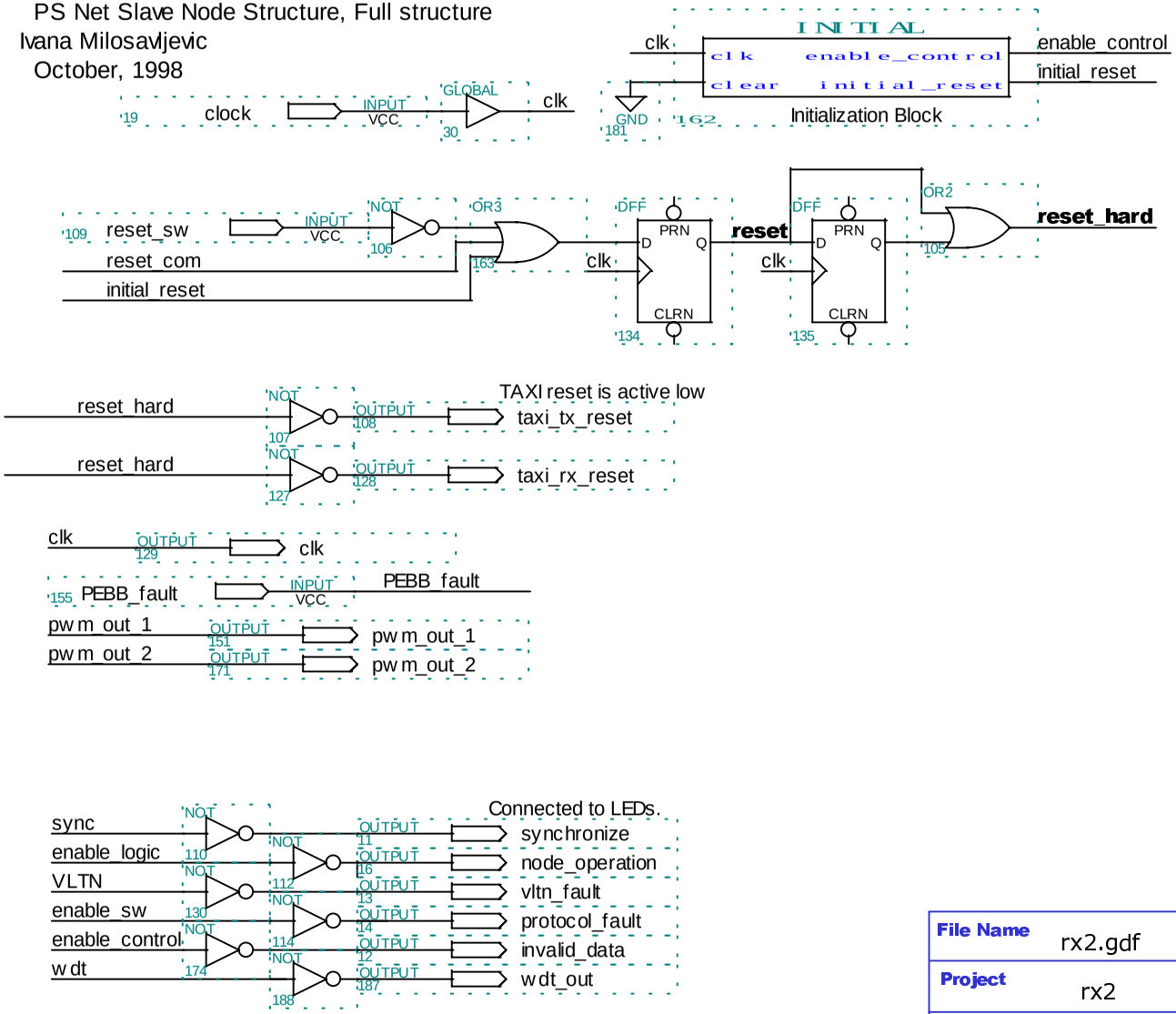
```
            corrupt_data   <= '1';
         WHEN shutdown_state  =>
            vltn_fault     <= '1';
         WHEN initial_state =>
            vltn_fault     <= '1';
         WHEN OTHERS =>
            state       := wait_state;
            vltn_fault     <= '1';
      END CASE;

-- ************************************************************
--    System has correctly received a command;
--    Control Block checks for the strobe signal from a TAXI receiver:
-- ************************************************************

   ELSIF  (rx_cstrb = '1') THEN
       CASE rx_command IS
   -- Depending on a state the system is in, Block does the appropriate

         -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
         --    Synchronization
         -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

         WHEN sync_identifier =>
            CASE state IS
               WHEN address_state =>
                  state       :=    wait_state;
                  protocol_fault    <=    '1';
                  synchronize <= '1';
               WHEN active_state =>
                  state    := wait_state;
                  count_data  := 0;
                  rx_adr      <= "00000000";
                  data_received  <= invalid_data;
                  protocol_fault <= '1';
                  corrupt_data   <= '1';
                  synchronize <= '1';
               WHEN OTHERS =>
                  synchronize <= '1';
               END CASE;

         -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
         --    Address Identifier
         -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            WHEN addr_identifier =>
               CASE state IS
               WHEN wait_state =>
                  state    := address_state;
               -- Fault cases:
               WHEN address_state   =>
                  protocol_fault <= '1';
               WHEN sync_state =>
                  state    := address_state;
                  protocol_fault <= '1';
               WHEN active_state =>
                  state    := wait_state;
```

```
         count_data  := 0;
         rx_adr      <= "00000000";
         data_received  <= invalid_data;
         protocol_fault <= '1';
         corrupt_data   <= '1';
      WHEN OTHERS =>
   END CASE;

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Shutdown
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN shutdown_identifier =>
   shutdown <= '1';
   state    :=  shutdown_state;
   corrupt_data <=   '1';

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Reset
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN reset_identifier =>
   reset       <= '1';
   shutdown    <= '0';
   vltn_fault     <= '0';
   corrupt_data   <= '0';
   protocol_fault <= '0';
   count_data     :=  0 ;
   rx_adr         <= "00000000";
   state       := wait_state;

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Taxi Sync
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN taxi_sync =>

-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Broken Line
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN broken_line =>
      CASE state IS
      WHEN initial_state   =>
         state       := initial_state;
      WHEN OTHERS =>
         shutdown    <= '1';
         corrupt_data   <= '1';
         line_break     <= '1';
         state       := shutdown_state;
   END CASE;
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
--    Initialization
-->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

WHEN initial_identifier =>
   enable_logic   <=    '1';
```

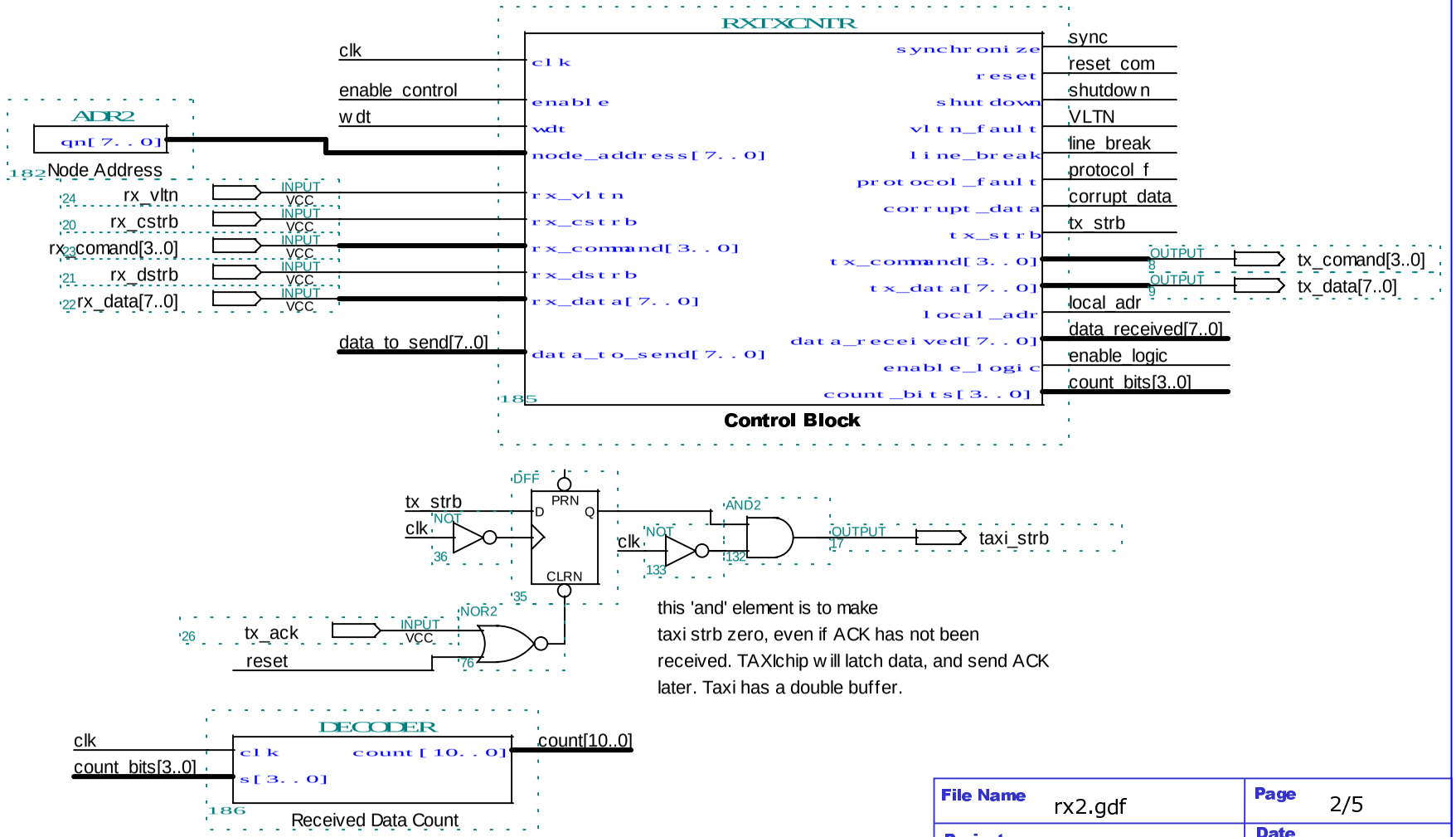
```
               initialize     <= '1';
               state       :=  wait_state;

            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
            --  Unexpected Command
            -->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            WHEN OTHERS =>
               --protocol_fault  <= '1';
         END CASE;

--*****************************************************************
--    System receives data
-- ***************************************************************

      ELSIF (rx_dstrb = '1') THEN
         CASE state IS
            WHEN address_state =>
               rx_adr      <= rx_data;
               local_adr   <= '1';
               count_data  :=     0;
               state    := active_state;
            WHEN active_state =>
               IF (count_data = number_of_data_bytes) THEN
                  state    := wait_state;
                  count_data  := count_data + 1;
                  data_received  <= rx_data;
               ELSE
                  data_received  <= rx_data;
                  state    := active_state;
                  count_data  := count_data + 1;
               END IF;
            WHEN OTHERS =>
         END CASE;
      END IF; -- wdt, vltn, commands, data
--*****************************************************************
     END IF; -- enable
     count_bits   <= count_data;
END PROCESS;
END a;
```

A.3 GRAPHIC DESIGN FILES

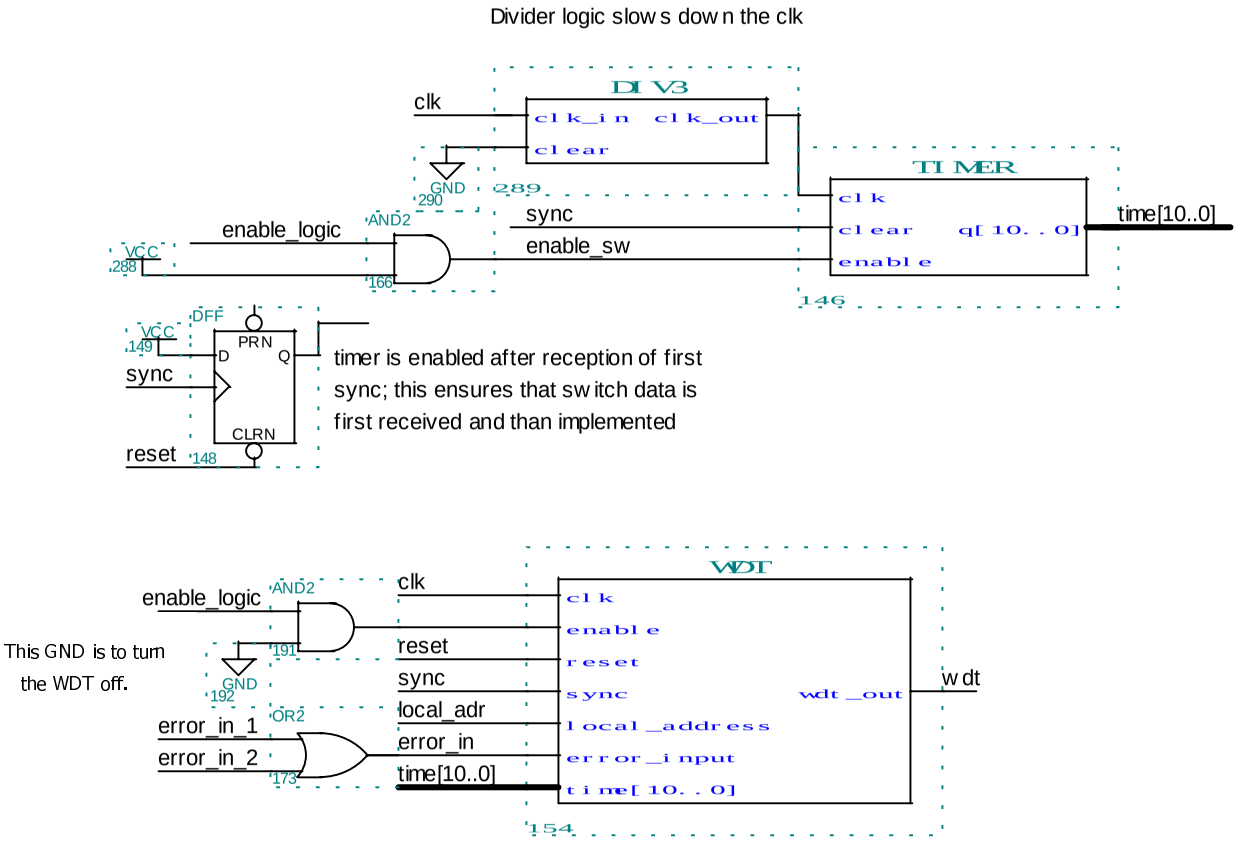
PS Net Slave Node Structure, Full structure
Ivana Milosavljevic
October, 1998



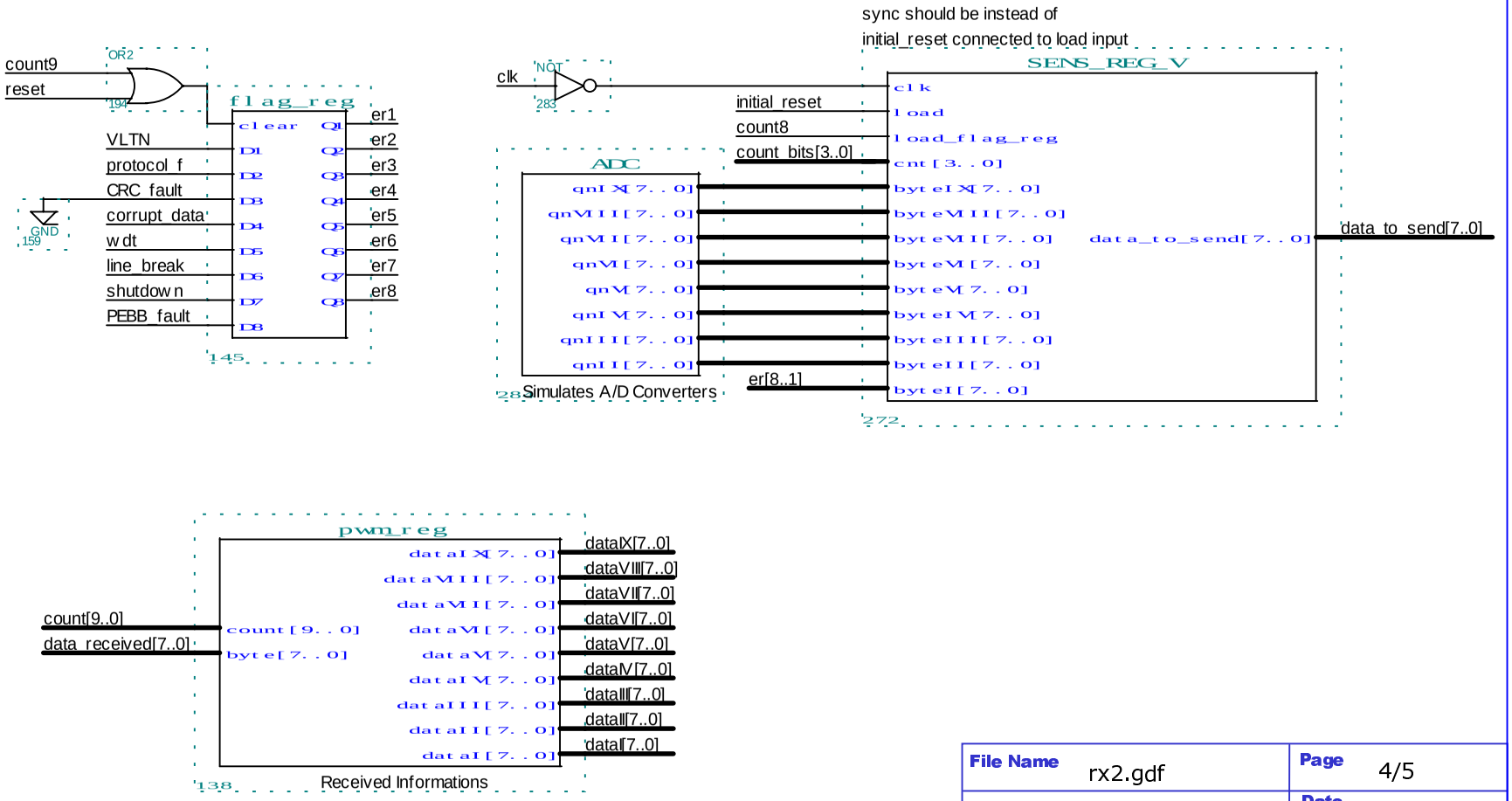
File Name	rx2.gdf	**Page**	1/5
Project	rx2	**Date**	October, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



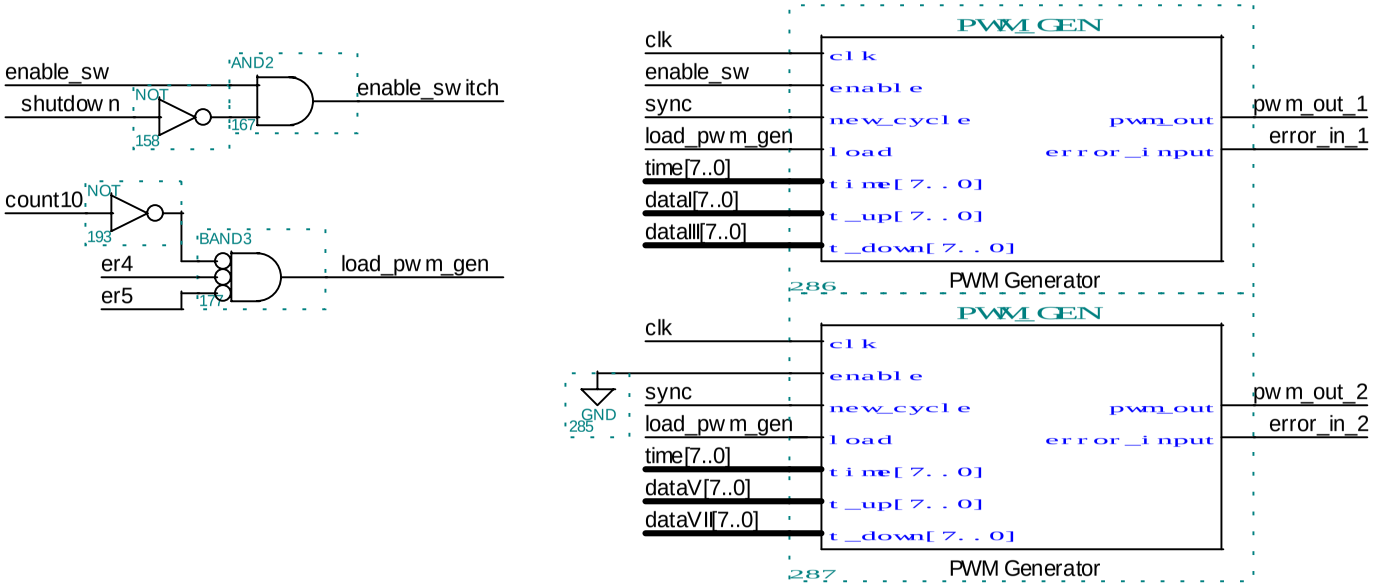
File Name	rx2.gdf	**Page**	2/5
Project	rx2	**Date**	October, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



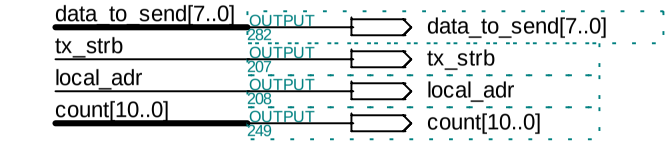
File Name	rx2.gdf	**Page**	3/5
Project	rx2	**Date**	October, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



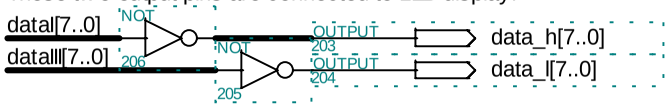
File Name	rx2.gdf	**Page**	4/5
Project	rx2	**Date**	Ovtober, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



TEST OUTPUTS:



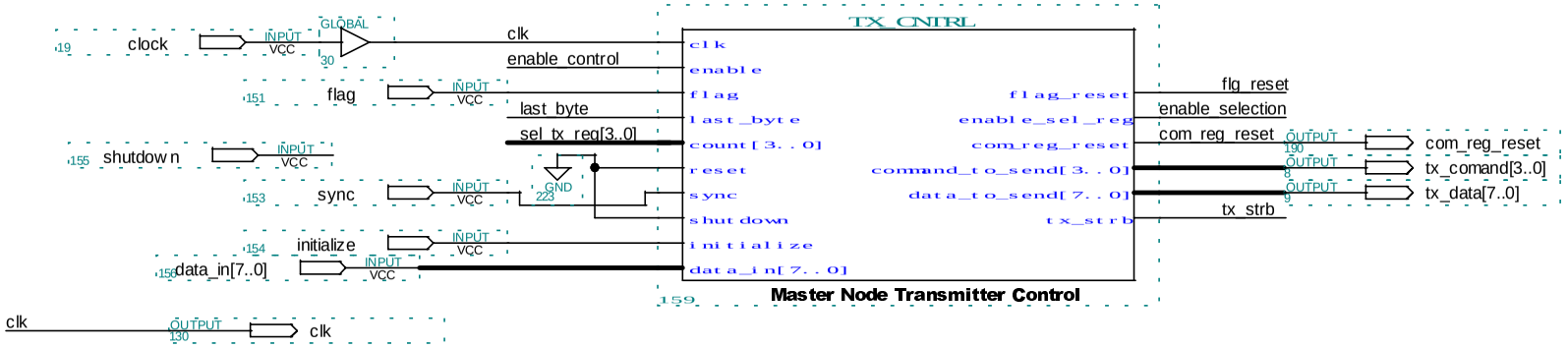
These two output pins are connected to LED display.

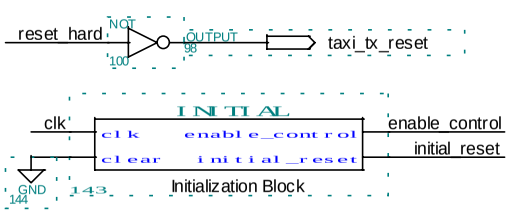


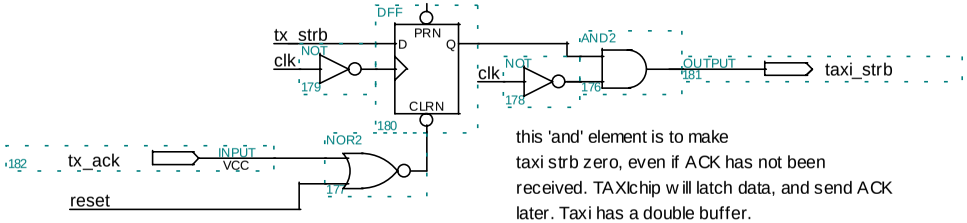
File Name	rx2.gdf	**Page**	5/5
Project	rx2	**Date**	October, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

CLOCK comes from the crystal on the board. It should be 10 MHz or less.
It may come from the output of TAXI transmitter.

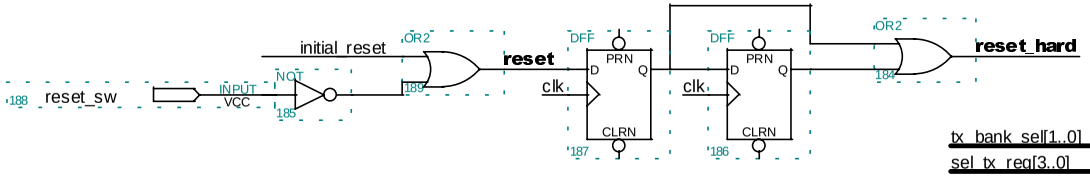
PES Net Transmitter Structure
Ivana Milosavljevic
March, 1998

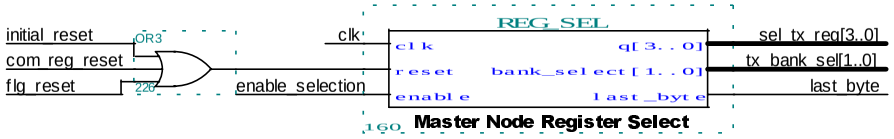




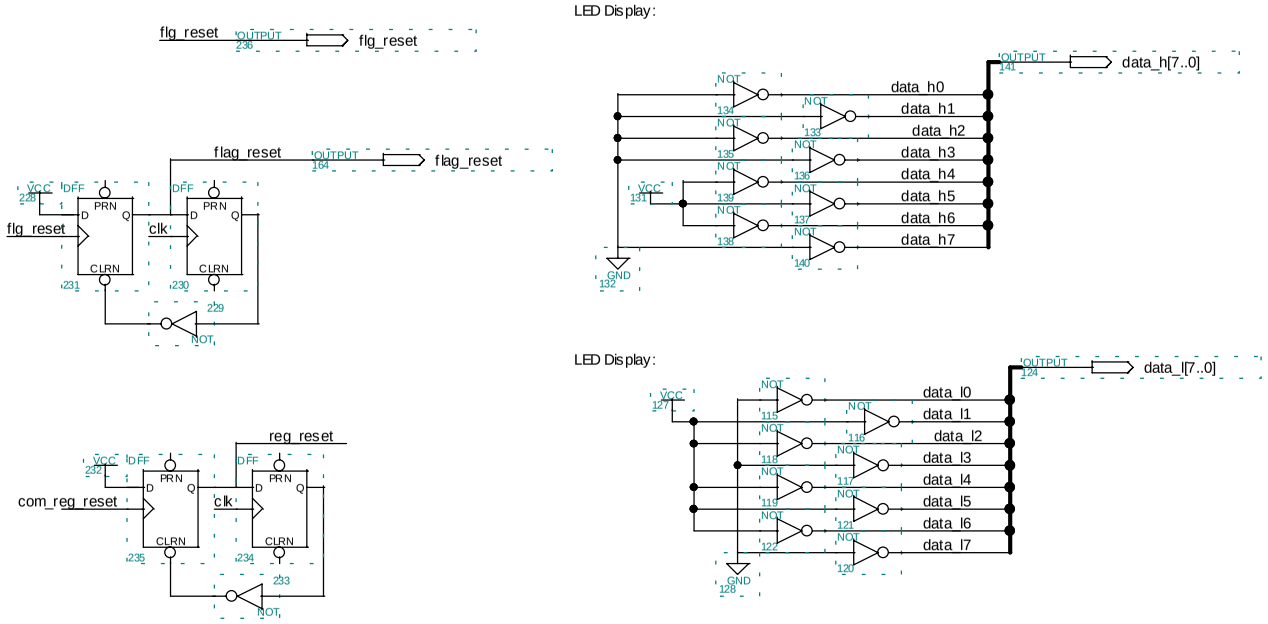


this 'and' element is to make taxi strb zero, even if ACK has not been received. TAXIchip will latch data, and send ACK later. Taxi has a double buffer.

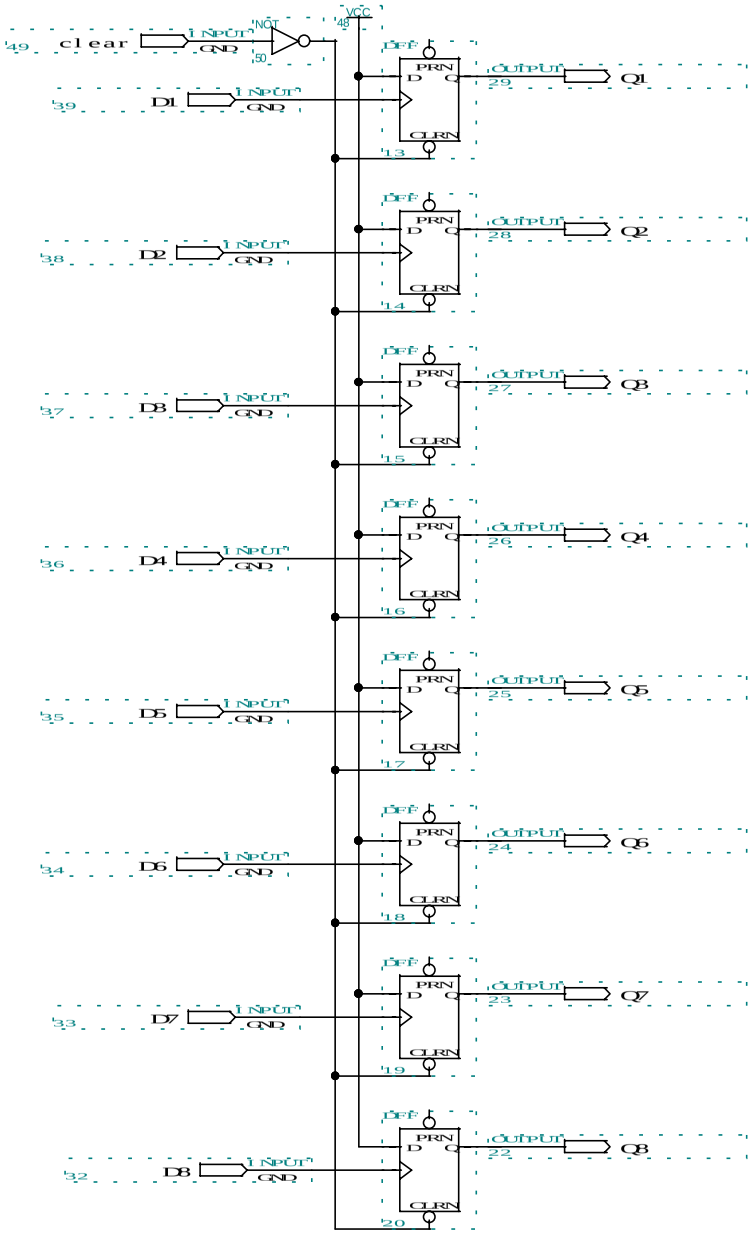




File Name	tx1full.gdf	**Page**	1/2
Project	tx1full	**Date**	March, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

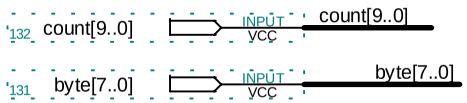


File Name tx1full.gdf	**Page** 2/2
Project tx1full	**Date** March, 1998
Software Altera MAX+plus II, Ver. 8.3	
Author Ivana Milosavljevic	



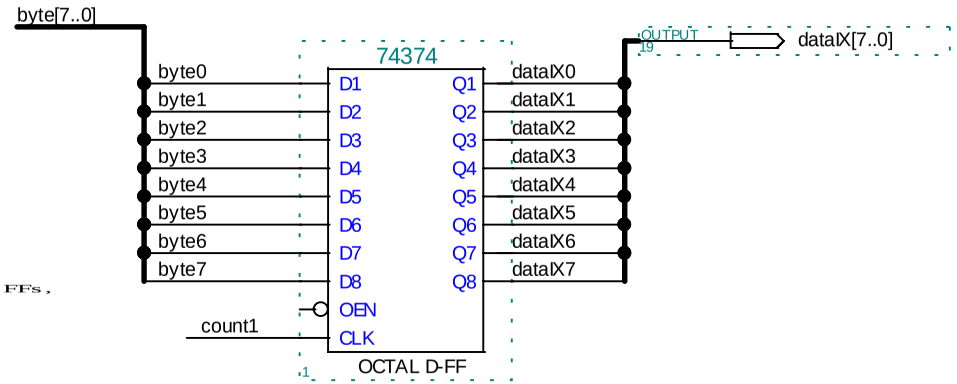
File Name	Flag_reg.gdf	**Page**	1/1
Project	rx2	**Date**	Oct., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

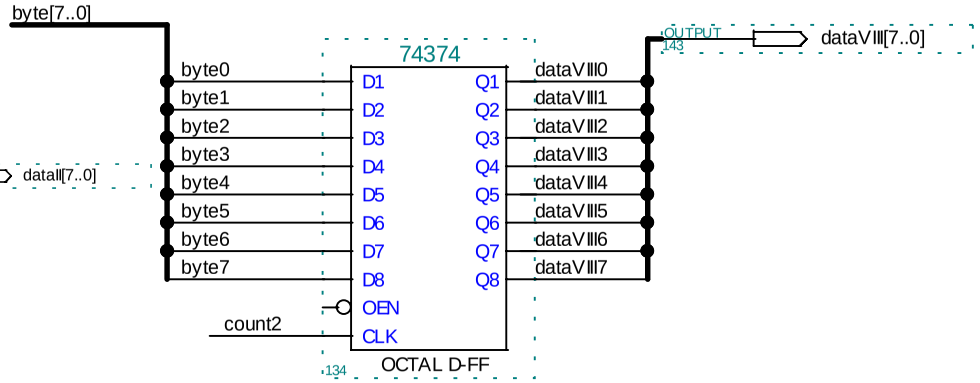
Received Information
Used for PWM Generation

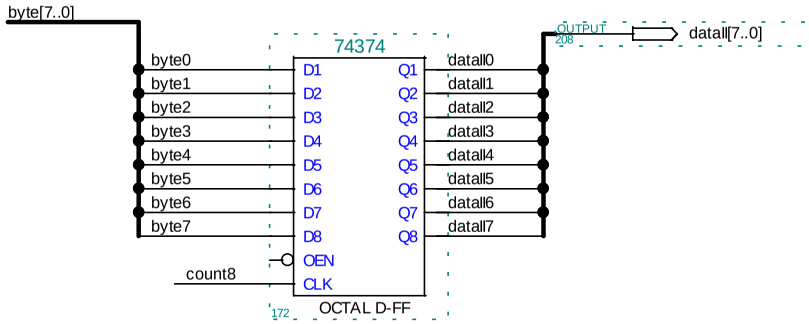


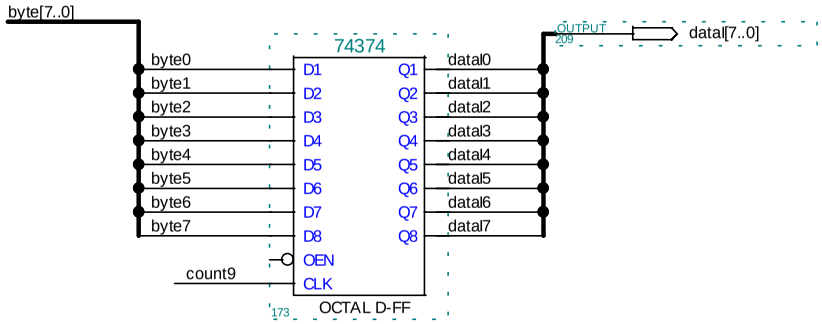
Here, The received information are stored. They will be used for generation of PWM pulses.

The 'count' input specifies to which of the nine FFs, the input 'byte' information will be written to.

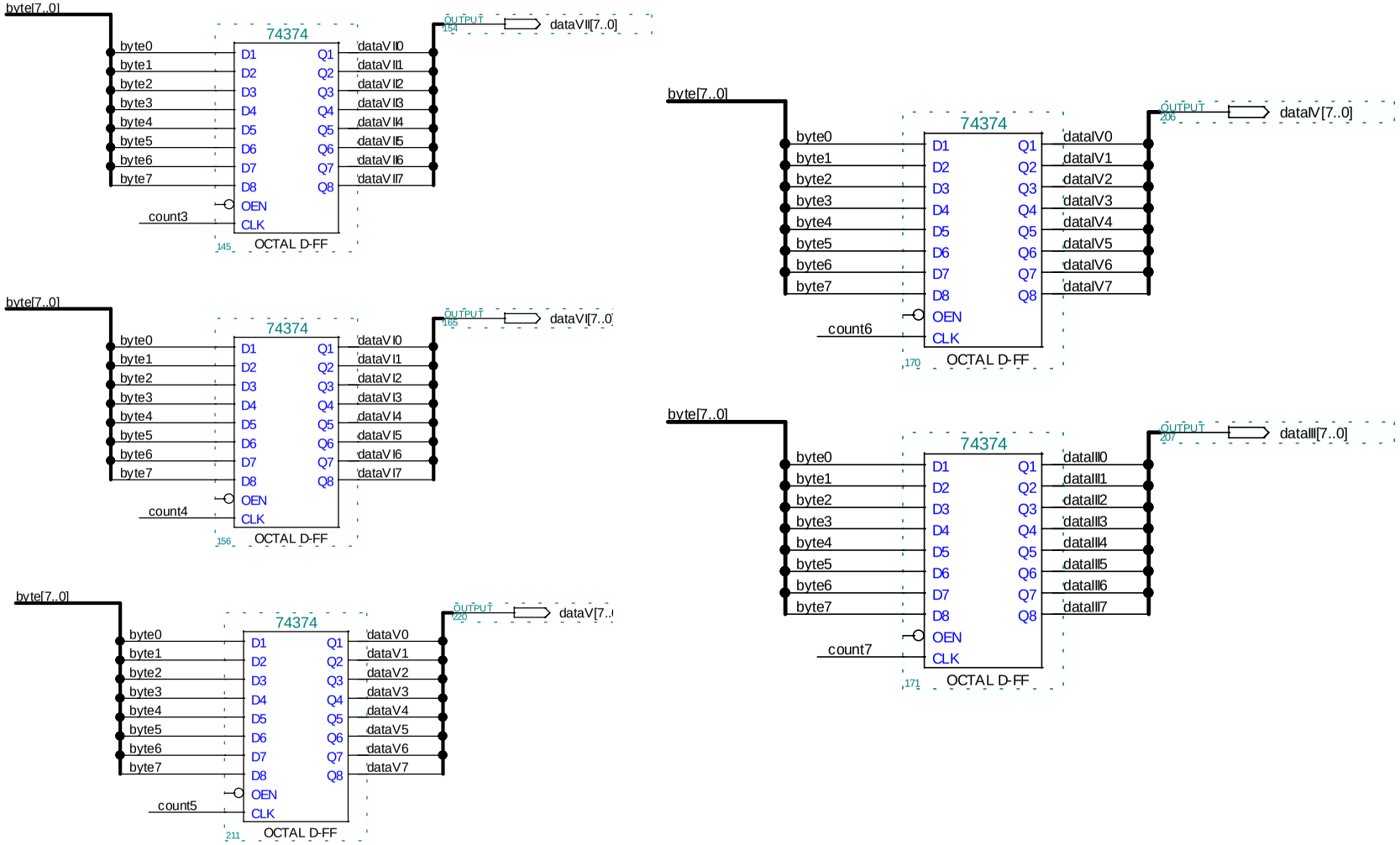




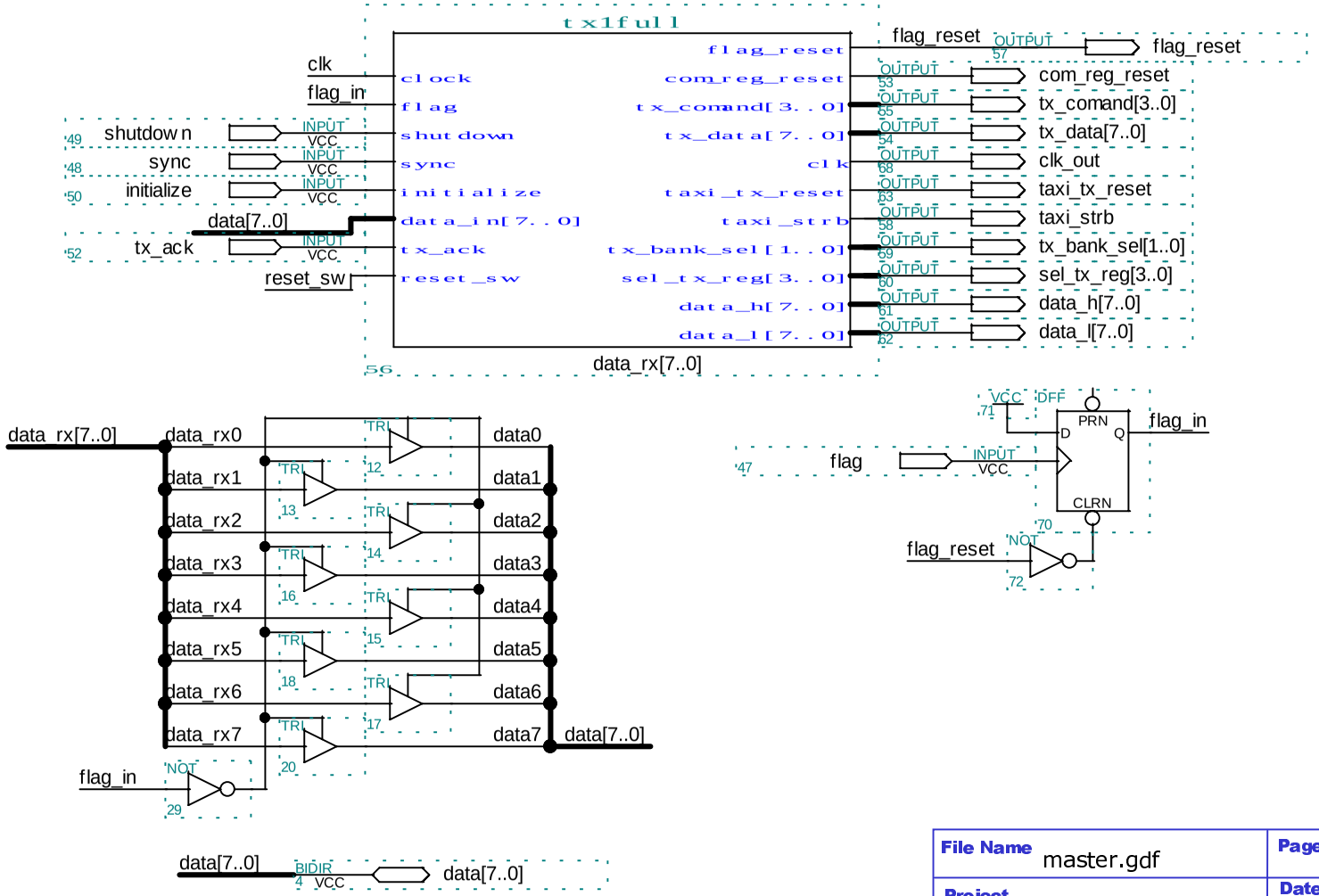




File Name	PWM_Reg.gdf	**Page**	1/2
Project	rx2	**Date**	Oct., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



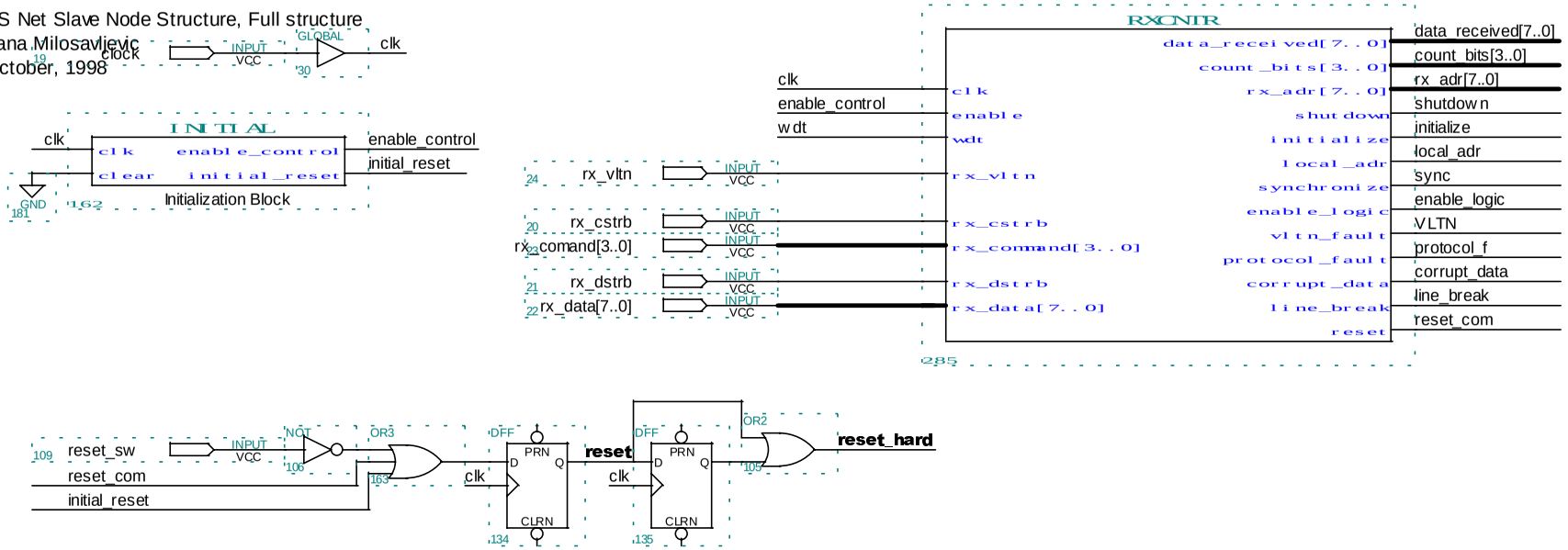
File Name	PWM_Reg.gdf	**Page**	2/2
Project	rx2	**Date**	Oct., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



A-141

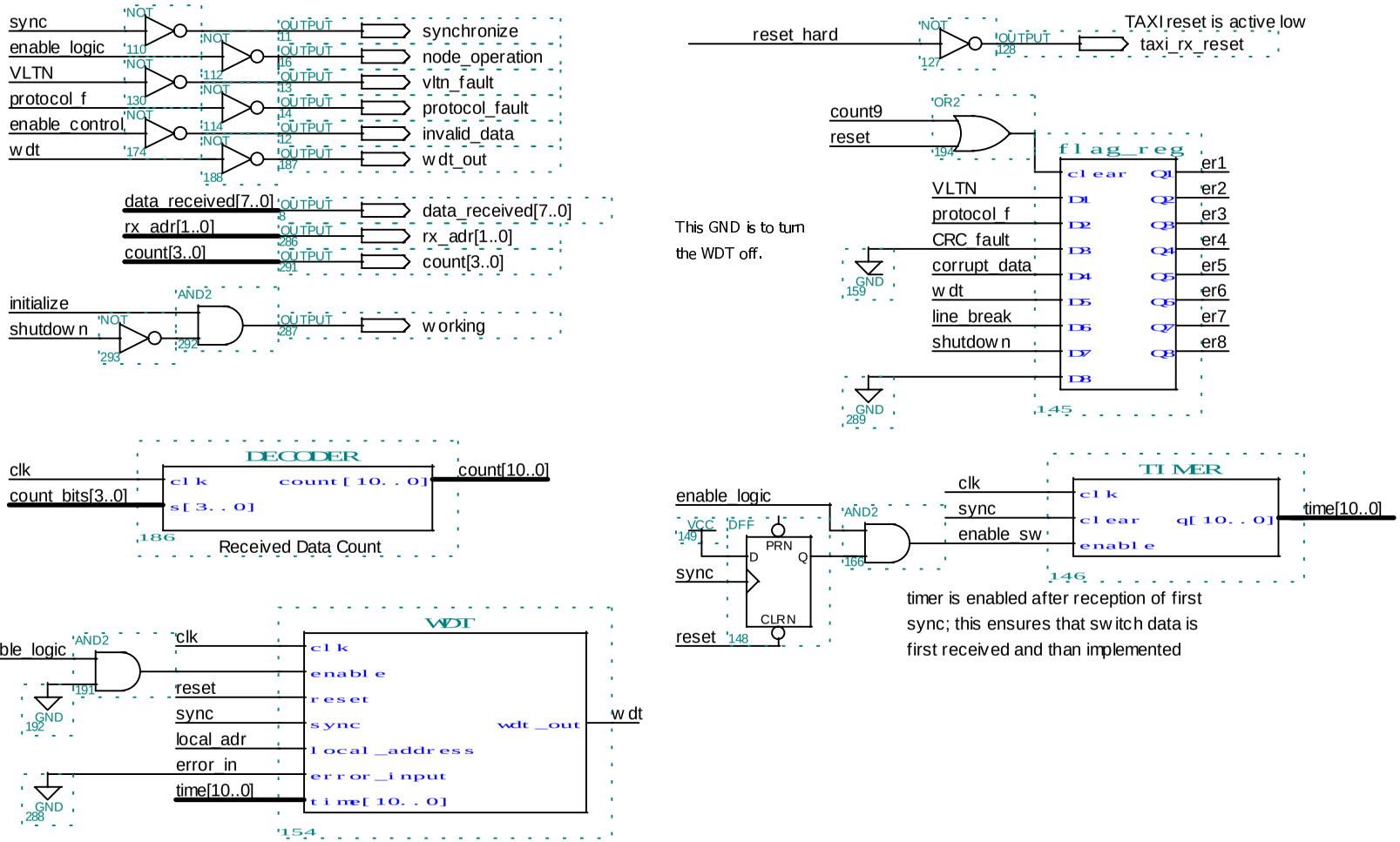
File Name	master.gdf	**Page**	2/2
Project	master	**Date**	Nov., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

PS Net Slave Node Structure, Full structure
Ivana Milosavljevic
October, 1998



A-142

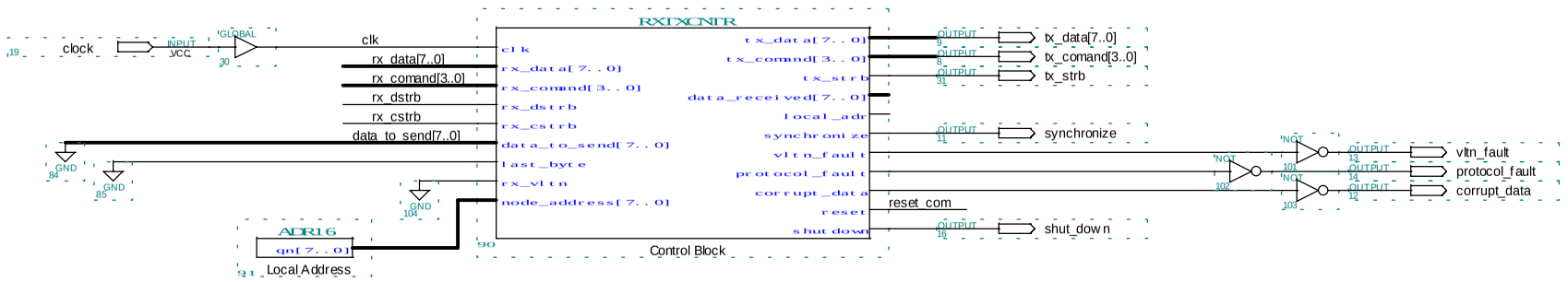
File Name	rx1full.gdf	**Page**	1/2
Project	master	**Date**	Nov., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



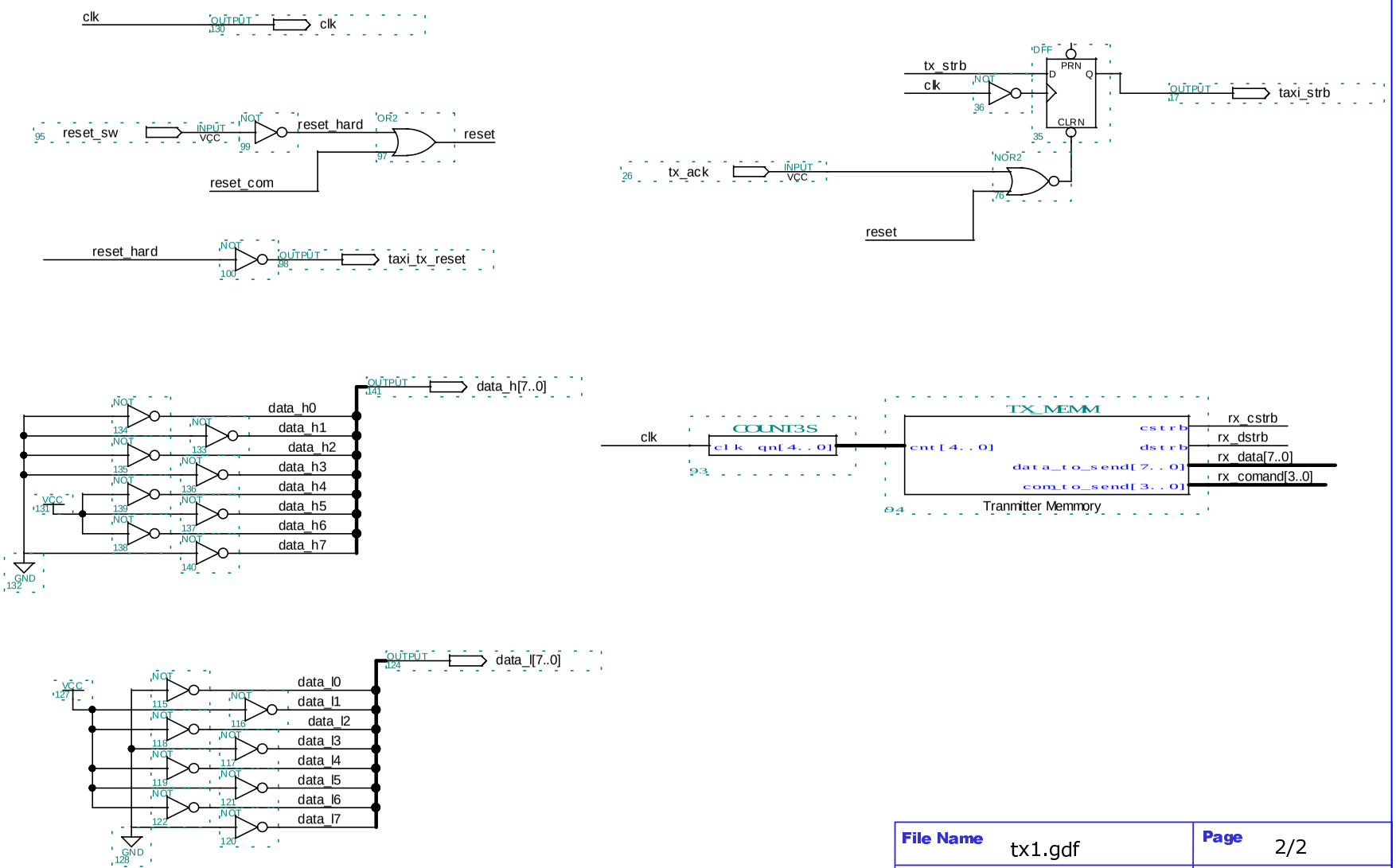
File Name	rx1full.gdf	**Page**	2/2
Project	master	**Date**	Nov., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

CLOCK comes from the crystal on the board. It should be 10 MHz or less.
It may come from the output of TAXI transmitter.

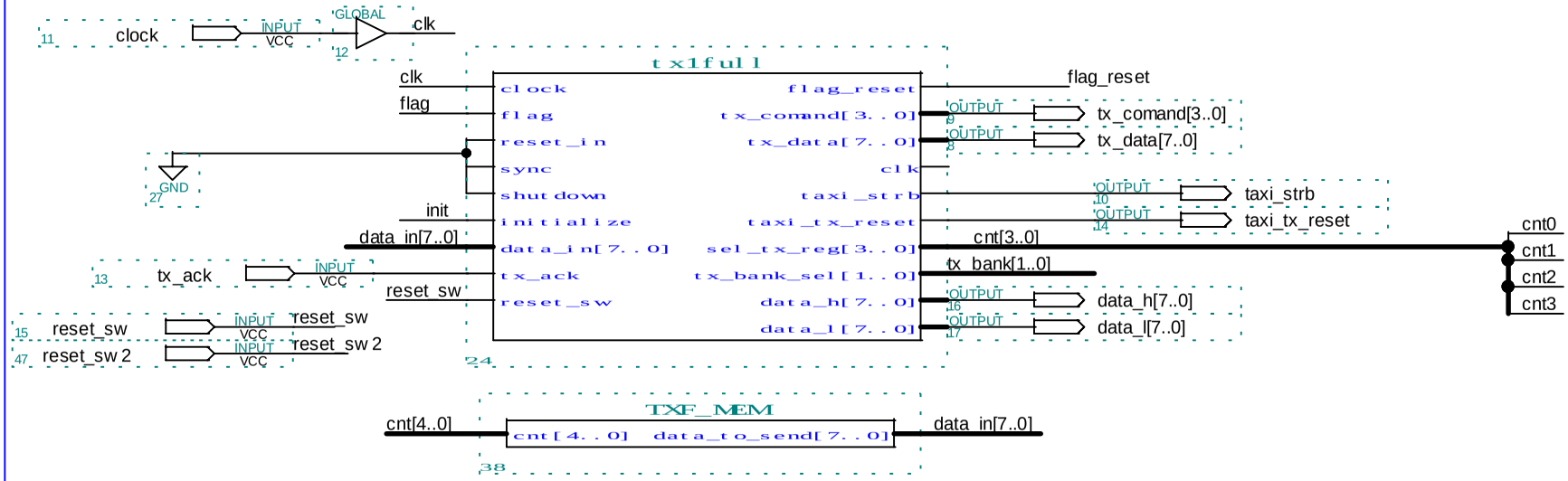
PES Net Transmitter Structure
Ivana Milosavljevic
March, 1998



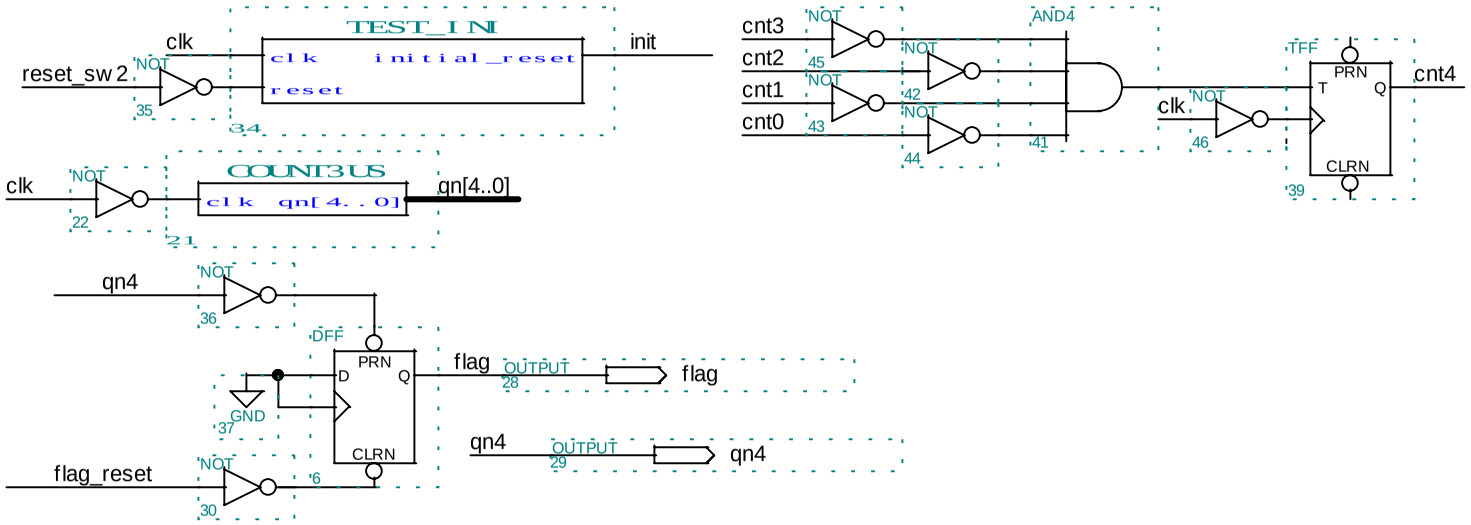
File Name	tx1.gdf	**Page**	1/2
Project	tx1	**Date**	March, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



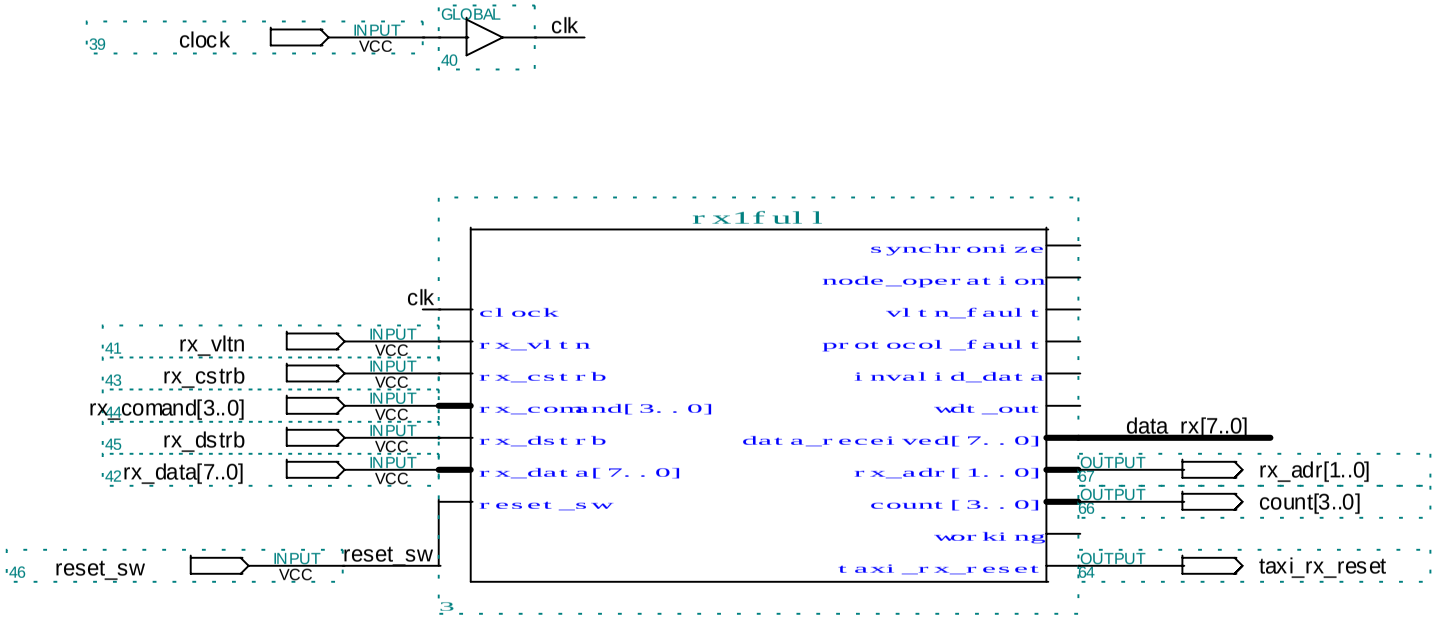
File Name	tx1.gdf	**Page**	2/2
Project	tx1	**Date**	March, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



File Name testtx1.gdf	**Page** 1/2
Project testtx1	**Date** Oct, 1998
Software Altera MAX+plus II, Ver. 8.3	
Author Ivana Milosavljevic	



File Name	testtx1.gdf	**Page**	2/2
Project	testtx1	**Date**	Oct, 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		



File Name	master.gdf	**Page**	1/2
Project	master	**Date**	Nov., 1998
Software	Altera MAX+plus II, Ver. 8.3		
Author	Ivana Milosavljevic		

REFERENCES

[A-1] Armstrong, J., R., Gray, F., G., *Structured Logic Design with VHDL*, New Jersey: Prentice Hall PTR, 1993.

[A-2] "Max+Plus® II Getting Started," *P25-04803-03*, Version 8.1, Altera Corporation, September 1997.

[A-3] "University Program Design Laboratory Package", *User Guide A-UG-UP1-01*, Version 1, Altera Corporation, August 1997.

[A-4] "Altera Data Book," *A-DB-0696-01*, Altera Corporation, June 1996.

[A-5] "TAXIchip™ Integrated Circuits," *Data Sheet and Technical Manual, Publication#07370*, Rev. F, Advanced Micro Devices, April 1994.

[A-6] Milosavljevic, I., Borojevic, D., "Modularized Control Architecture for Power Converters," *16th Annual VPEC Seminar Proceedings*, pp. 85-92, September 1998.

[A-7] Milosavljevic, I., Ye, Z., Borojevic, D., Holton, C., "Analysis of Converter Operation with a Phase-Leg in Daisy–Chained or Ring Type Structure", *IEEE PESC'99 Proceedings*, June 1999.