```
--
--  txstate.vhd
--  Serial Transceiver Transmit interface
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity flowstate is
  port (CLK: in STD_LOGIC;
        Data_in: in unsigned (15 downto 0);
        Data_Ready: in BOOLEAN;
        Data_new: in BOOLEAN;
          DATA_addr: in unsigned (31 downto 0);

          iCTS: in BOOLEAN;
          iXON: in BOOLEAN;
          iXOFF: in BOOLEAN;
          CTS: in BOOLEAN;
          XON: in BOOLEAN;
          XOFF: in BOOLEAN;

          iCTS_clear: out BOOLEAN;
          iXON_clear: out BOOLEAN;
          iXOFF_clear: out BOOLEAN;
          CTS_clear: out BOOLEAN;

        Data_out: out unsigned (15 downto 0);
        DIN_ACK: out BOOLEAN;
        EN: out BOOLEAN;
        ER: out BOOLEAN);
end;

architecture flowstate_arch of flowstate is


-- SYMBOLIC ENCODED state machine: TX_state
type TX_state_type is (siCTS, siXoff, siXon, sINT1, sINT2, sINT3, sADDRh,
sADDRl, sCTS_wait, sIDLE, sINT, sRTS, sXoff, sXon);
signal TX_state: TX_state_type;
signal ReturnState: TX_state_type;

signal inew: BOOLEAN;
signal inewclear: BOOLEAN;


constant dIDLE : unsigned(15 downto 0) := "0000000000000000";
constant dCTS : unsigned(15 downto 0) := "0000000000000001";
constant dRTS : unsigned(15 downto 0) := "0000000000000010";
constant dXON : unsigned(15 downto 0) := "0000000000000101";
constant dXOFF : unsigned(15 downto 0) := "0000000000000110";

begin


TX_state_machine: process (CLK)

begin

if CLK'event and CLK = '1' then

   case TX_state is
```

```
when siCTS =>
   -- go back from whence we came
   EN <= TRUE;
   ER <= FALSE;
   DATA_OUT <= dCTS;
   DIN_ACK <= FALSE;
   TX_state <= ReturnState;
   if ReturnState = sXon then
      DIN_ACK <= TRUE;
   end if;

when sINT1 =>
   EN <= FALSE;
   ER <= TRUE;
   DATA_OUT <= dIDLE;
   DIN_ACK <= FALSE;
   TX_state <= siCTS;

when siXON =>
   -- go back from whence we came
   EN <= TRUE;
   ER <= FALSE;
   DATA_OUT <= dXON;
   DIN_ACK <= FALSE;
   TX_state <= ReturnState;
   if ReturnState = sXon then
      DIN_ACK <= TRUE;
   end if;

when sINT2 =>
   EN <= FALSE;
   ER <= TRUE;
   DATA_OUT <= dIDLE;
   DIN_ACK <= FALSE;
   TX_state <= siXON;

when siXOFF =>
   -- go back from whence we came
   EN <= TRUE;
   ER <= FALSE;
   DATA_OUT <= dXOFF;
   DIN_ACK <= FALSE;
   TX_state <= ReturnState;
   if ReturnState = sXon then
      DIN_ACK <= TRUE;
   end if;

when sINT3 =>
   EN <= FALSE;
   ER <= TRUE;
   DATA_OUT <= dIDLE;
   DIN_ACK <= FALSE;
   TX_state <= siXOFF;

when sADDRh =>
   EN <= TRUE;
   ER <= FALSE;
   DATA_OUT <= DATA_addr(31 downto 16);
   DIN_ACK <= FALSE;
   inewclear <= FALSE;
   TX_state <= sADDRl;

when sADDRl =>
   EN <= TRUE;
   ER <= FALSE;
   DATA_OUT <= DATA_addr(15 downto 0);
```

```
    DIN_ACK <= FALSE;
    TX_state <= sCTS_wait;

when sCTS_wait =>
    EN <= FALSE;
    ER <= FALSE;
    DATA_OUT <= dIDLE;
    DIN_ACK <= FALSE;
    ReturnState <= sCTS_wait;
    if iCTS then
       TX_state <= sINT1;
    elsif iXON then
       TX_state <= sINT2;
    elsif iXOFF then
       TX_state <= sINT3;
    elsif CTS then
       TX_state <= sXon;
       DIN_ACK <= TRUE;
    end if;
when sIDLE =>
    EN <= FALSE;
    ER <= FALSE;
    DATA_OUT <= dIDLE;
    DIN_ACK <= FALSE;
    ReturnState <= sIDLE;
    if iCTS then
       TX_state <= sINT1;
    elsif Data_ready then
       TX_state <= sINT;
    end if;

when sINT =>
    ER <= TRUE;
    EN <= FALSE;
    DATA_OUT <= dIDLE;
    DIN_ACK <= FALSE;
    TX_state <= sRTS;

when sRTS =>
    ER <= FALSE;
    EN <= TRUE;
    DATA_OUT <= dRTS;
    DIN_ACK <= FALSE;
    inewclear <= TRUE;
    TX_state <= sADDRh;

when sXoff =>
    ER <= FALSE;
    EN <= FALSE;
    DATA_OUT <= dIDLE;
    DIN_ACK <= FALSE;
    ReturnState <= sXoff;
    if iCTS then
       TX_state <= sINT1;
    elsif iXON then
       TX_state <= sINT2;
    elsif iXOFF then
       TX_state <= sINT3;
    elsif Xon then
       TX_state <= sXon;
       DIN_ACK <= TRUE;
    end if;

when sXon =>
    ER <= FALSE;
    EN <= TRUE;
    DATA_OUT <= DATA_IN;
```

```
            DIN_ACK <= TRUE;
            ReturnState <= sXon;
            if Xoff then
               TX_state <= sXoff;
            elsif iXON then
               TX_state <= sINT2;
            elsif iXOFF then
               TX_state <= sINT3;
            elsif iCTS then
               TX_state <= sINT1;
            elsif inew then
               TX_state <= sINT;
            elsif NOT Data_Ready then
               TX_state <= sIDLE;
            end if;

         when others =>
            TX_STATE <= sIDLE;
      end case;
   end if;
   end process;

   process (tx_state)
   begin
      iCTS_clear <= (tx_state = siCTS);
      iXON_clear <= (tx_state = siXON);
      iXOFF_clear <= (tx_state = siXOFF);
      CTS_clear <= (tx_state = sXON);
   end process;

   process (data_new, inewclear)
   begin
      if data_new then
         inew <= TRUE;
      elsif inewclear then
         inew <= FALSE;
      end if;
   end process;
   end flowstate_arch;
```

```
--
--  rxstate.vhd
--  Serial Transceiver Receive interface
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity rxstate is
  port (CLK: in STD_LOGIC;
        PCI_en: buffer BOOLEAN;
        LINK_en: buffer BOOLEAN;
        Data_ready: out BOOLEAN;
        Data_out: out unsigned (15 downto 0);
          DATA_addr: buffer unsigned (31 downto 0);
          iCTS: out BOOLEAN;
          iXON: out BOOLEAN;
          iXOFF: out BOOLEAN;
          RTS: out BOOLEAN;
          CTS: out BOOLEAN;
          XON: out BOOLEAN;
          XOFF: out BOOLEAN;
        LocalMin : in unsigned (31 downto 22);
        LocalMax : in unsigned (31 downto 22);
        link_ready: in BOOLEAN;
        pci_ready:in BOOLEAN;
        Data_in: in unsigned (15 downto 0);
        EN: in BOOLEAN;
        ER: in BOOLEAN);
end;

architecture rxstate_arch of rxstate is


-- SYMBOLIC ENCODED state machine: RX_state
type RX_state_type is (sCTS, sADDRh, sADDRl, sCTS_wait, sIDLE, sINT, sRTS,
sXoff, sXon);
signal RX_state: RX_state_type;
signal ReturnState: RX_state_type;


constant dIDLE : unsigned(15 downto 0) := "0000000000000000";
constant dCTS : unsigned(15 downto 0) := "0000000000000001";
constant dRTS : unsigned(15 downto 0) := "0000000000000010";
constant dXON : unsigned(15 downto 0) := "0000000000000101";
constant dXOFF : unsigned(15 downto 0) := "0000000000000110";

begin


RX_state_machine: process (CLK)

begin

if CLK'event and CLK = '1' then


   case RX_state is

      when sXON =>
         DATA_OUT <= DATA_IN;
         ReturnState <= sXON;
```

```
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            CTS <= FALSE;
            Data_Ready <= EN and not ER;
            if (EN and not ER) then
               DATA_ADDR <= DATA_ADDR+2;
            end if;
            if (ER and not EN) then
               RX_state <= sINT;
            elsif (EN and ER) then
               -- error - handle?
               RX_state <= sIDLE;
            elsif (PCI_en and not PCI_ready) or (LINK_EN and not LINK_ready)
then
               iXOFF <= TRUE;
               RX_state <= sXOFF;
            end if;

         when sXOFF =>
            DATA_OUT <= DATA_IN;
            ReturnState <= sXOFF;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            CTS <= FALSE;
            Data_Ready <= EN and not ER;
            if (ER and not EN) then
               RX_state <= sINT;
            elsif (EN and ER) then
               -- error - handle?
               RX_state <= sIDLE;
            elsif (PCI_en and PCI_ready) or (LINK_EN and LINK_ready) then
               iXON <= TRUE;
               RX_state <= sXON;
            end if;

         when sADDRh =>
            DATA_OUT <= DATA_IN;
            Data_Ready <= FALSE;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            RTS <= TRUE;
            DATA_addr (31 downto 16) <= DATA_IN;
            if ((DATA_IN(15 downto 8) >= LocalMin) and (DATA_IN(15 downto 8) <
LocalMax)) then
               PCI_en <= TRUE;
               LINK_en <= FALSE;
            else
               PCI_en <= FALSE;
               LINK_en <= TRUE;
            end if;
            RX_state <= sADDRl;

         when sADDRl =>
            DATA_OUT <= DATA_IN;
            Data_Ready <= FALSE;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            RTS <= FALSE;
            DATA_addr (15 downto 0) <= DATA_IN -2;
            RX_state <= sCTS_wait;

         when sCTS_wait =>
            DATA_OUT <= dIDLE;
```

```
            ReturnState <= sCTS_wait;
            Data_Ready <= FALSE;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            CTS <= FALSE;
            if (ER and not EN) then
               RX_state <= sINT;
            elsif (PCI_en and PCI_ready) or (LINK_en and LINK_ready) then
               RX_state <= sXON;
               iCTS <= TRUE;

            end if;

         when sIDLE =>
            DATA_OUT <= dIDLE;
            Data_Ready <= FALSE;
            ReturnState <= sIDLE;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            CTS <= FALSE;
            if (ER and not EN) then
               RX_state <= sINT;
            end if;

         when sINT =>
            DATA_OUT <= dIDLE;
            Data_Ready <= FALSE;
            iXON <= FALSE;
            iXOFF <= FALSE;
            iCTS <= FALSE;
            if (DATA_IN = dRTS) then
               RX_state <= sADDRh;
            elsif (DATA_IN = dCTS) then
               RX_STATE <= ReturnState;
               CTS <= TRUE;
            elsif (DATA_IN = dXON) then
               RX_STATE <= ReturnState;
               XON <= TRUE;
               XOFF <= FALSE;
            elsif (DATA_IN = dXOFF) then
               XOFF <= TRUE;
               XON <= FALSE;
               RX_STATE <= ReturnState;
            end if;



         when others =>
            RX_STATE <= sIDLE;
      end case;
   end if;
   end process;
   end rxstate_arch;
```

```
--
--  lbus.vhd
--  PLX 9054 interface
--

library IEEE;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;



entity lbus is
  port (CLK: in STD_LOGIC;

          reset: in BOOLEAN;

          LHOLD: in BOOLEAN;
          LHOLDA: buffer BOOLEAN;

          dir0min: buffer unsigned (31 downto 22);
          dir0max: buffer unsigned (31 downto 22);
          dir1min: buffer unsigned (31 downto 22);
          dir1max: buffer unsigned (31 downto 22);


          DATAOUT: buffer unsigned (31 downto 0);
          ADDROUT: buffer unsigned (31 downto 0);
          dir1: buffer BOOLEAN;
          dir0: buffer BOOLEAN;
          dir1_in_ready: in BOOLEAN;
          dir0_in_ready: in BOOLEAN;
          data_in_new: out BOOLEAN;
          data_in_next: out BOOLEAN;

          data_out_ready: in BOOLEAN;
          data_out_new: in BOOLEAN;
          data_out_addr: in unsigned (31 downto 0);
          data_out_data: in unsigned (31 downto 0);
          data_out_next: out BOOLEAN;

       ADSIn: in BOOLEAN;
       ADSOut: out BOOLEAN;
       ADSEn: out BOOLEAN;
       BLASTIn: in BOOLEAN;
       BLASTOut: out BOOLEAN;
       BLASTEn: out BOOLEAN;
       LBEin: in STD_LOGIC_VECTOR (3 downto 0);
       LBEout: out STD_LOGIC_VECTOR (3 downto 0);
       LBEEn: out BOOLEAN;
       LWRIn: in BOOLEAN;
       LWROut: out BOOLEAN;
       LWREn: out BOOLEAN;
       LADin: in unsigned (31 downto 0);
       LADout: out unsigned (31 downto 0);
       LADEn: out BOOLEAN;
       READYIn: in BOOLEAN;
       READYOut: buffer BOOLEAN;
       READYEn: out BOOLEAN;
       BTERMIn: in BOOLEAN;
       BTERMOut: out BOOLEAN;
       BTERMEn: out BOOLEAN);
end;
```

```
architecture lbus_arch of lbus is

type Lbus_state_type is (sRX_ADDR, sRX_DATA, sIDLE, sRECOVER, sTX_ADDR,
sTX_DATA, sTX_LAST, sConfig1, sConfig2);
signal Lbus_state: Lbus_state_type;

constant cConfig1: unsigned (31 downto 0) := X"FFFF0000";
constant cConfig2: unsigned (31 downto 0) := X"FFFF0001";


begin


process (dir0min, dir0max, dir1min, dir1max, addrout)
begin


 if (dir0min < dir0max) then     -- no addr boundary in range, normal compare
   if (ADDROUT >= dir0min and ADDROUT < dir0max) then
      dir0 <= TRUE;
   else
      dir0 <= FALSE;
   end if;
 else               -- addr boundary in direction's range
   if (ADDROUT >= dir0min or ADDROUT < dir0max) then
      dir0 <= TRUE;
   else
      dir0 <= FALSE;
   end if;
 end if;


 if (dir1min < dir1max) then     -- no addr boundary in range, normal compare
   if (ADDROUT >= dir1min and ADDROUT < dir1max) then
      dir1 <= TRUE;
   else
      dir1 <= FALSE;
   end if;
 else               -- addr boundary in direction's range
   if (ADDROUT >= dir1min or ADDROUT < dir1max) then
      dir1 <= TRUE;
   else
      dir1 <= FALSE;
   end if;
 end if;
end process;


process (data_out_ready, lhold)
begin

 if lhold and not data_out_ready then
   lholda <= TRUE;
 else
   lholda <= FALSE;
 end if;

end process;


Lbus_state_machine: process (CLK)
begin

if rising_edge(CLK) then

       ADSEn <= FALSE;
```

```
    BLASTEn <= FALSE;
    LBEEn <= FALSE;
    LWREn <= FALSE;
    LADEn <= FALSE;
    READYEn <= FALSE;
    BTERMEn <= FALSE;
    ADSout <= FALSE;
    BLASTout <= FALSE;
    LBEout <= "0000";
    LWRout <= FALSE;
    LADout <= X"00000000";
    READYout <= FALSE;
    BTERMout <= FALSE;

        data_in_new <= FALSE;
        data_in_next <= FALSE;
        data_out_next <= FALSE;


if reset then

  lbus_state <= sIDLE;

else

case Lbus_state is

    when sIDLE =>


        if data_out_ready then
            -- we have data destined for this board
            Lbus_state <= sTX_ADDR;

        elsif (ADSin and LHOLDA) then
        -- 9054 owns the bus and is sending data
            READYout <= TRUE;
            READYen <= TRUE;
            data_in_new <= TRUE;

            if (LADin = cConfig1) then
                Lbus_state <= sCONFIG1;
            elsif (LADin = cConfig2) then
                Lbus_state <= sCONFIG2;
            else
                Lbus_state <= sRX_ADDR;
            end if;
        else
            Lbus_state <=sIDLE;
        end if;

    when sTX_ADDR =>
        LADout <= data_out_addr;
        LADen <= TRUE;
        ADSout <= TRUE;
        ADSen <= TRUE;
        BLASTout <= FALSE;
        BLASTen <= TRUE;
        Lbus_state <= sTX_DATA;

    when sTX_DATA =>
        ADSout <= FALSE;
        ADSen <= TRUE;
        LADout <= data_out_data;
        LADen <= TRUE;
        LWRout <= TRUE;
        LWREn <= TRUE;
```

```
   LBEout <= "1111";
   LBEen <= TRUE;
   BLASTout <= FALSE;
   BLASTen <= TRUE;

   data_out_next <= READYin;

   if BTERMin or not data_out_ready then
      Lbus_state <= sTX_LAST;
   else
      Lbus_state <= sTX_DATA;
   end if;


when sTX_LAST =>

   ADSout <= FALSE;
   ADSen <= TRUE;
   LADout <= X"00000000";
   LADen <= TRUE;
   LWRout <= TRUE;
   LWRen <= TRUE;
   LBEout <= "0000";
   LBEen <= TRUE;
   BLASTout <= TRUE;
   BLASTen <= TRUE;

   Lbus_state <= sRECOVER;


when sRX_ADDR =>
   -- first Data word is desitnation Address
   ADDROUT <= LADin - 2; --- not 2 because of way 32to16 works
   READYout <= FALSE;
   READYen <= TRUE;
   BTERMout <= FALSE;
   BTERMen <= TRUE;


   Lbus_state <= sRX_DATA;

when sRX_DATA =>

   DATAOUT <= LADin;
   data_in_next <= READYout;

   if (READYout) then
      ADDROUT <= ADDROUT+4;
   end if;

   READYout <= (dir0 and dir0_in_ready) or (dir1 and dir1_in_ready);
   READYen <= TRUE;

   BTERMout <= data_out_ready;
   BTERMen <= TRUE;

   if BLASTin then
      Lbus_state <= sIDLE;
   else
      Lbus_state <= sRX_DATA;
   end if;

when sRECOVER =>
   Lbus_state <= sIDLE;

when sCONFIG1 =>
   BTERMout <= TRUE;
```

```
            BTERMen <= TRUE;
            READYout <= FALSE;
            READYen <= TRUE;

            dir0min (31 downto 22) <= LADin (9 downto 0);
            dir0max (31 downto 22) <= LADin (25 downto 16);

            lbus_state <= sIDLE;

         when sCONFIG2 =>
            BTERMout <= TRUE;
            BTERMen <= TRUE;
            READYout <= FALSE;
            READYen <= TRUE;

            dir1min (31 downto 22) <= LADin (9 downto 0);
            dir1max (31 downto 22) <= LADin (25 downto 16);

            lbus_state <= sIDLE;

         when others =>
            lbus_state <= sIDLE;

      end case;
      end if;
   end if;
   end process;


   end lbus_arch;
```

```
--
--  direction.vhd
--  decides which direction to send incoming data in the ring
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity direction is
    port (
   clk: in STD_LOGIC;
        iaddr: in unsigned (31 downto 0);
        dir1: out STD_LOGIC
        dir2: out STD_LOGIC
    );
end direction;

architecture direction_arch of direction is

signal dir0min: unsigned (31 downto 22);
signal dir0max: unsigned (31 downto 22);
signal dir1min: unsigned (31 downto 22);
signal dir1max: unsigned (31 downto 22);

begin

process (CLK)
begin

if(rising_edge(clk)) then

 if (dir0min < dir0max) then     -- no addr boundary in range, normal compare
   if (iaddr > dir0min and iaddr < dir0max) then
      dir0 <='1';
   else
      dir0 <='0';
   end if;
 else               -- addr boundary in direction's range
   if (iaddr > dir0min or iaddr < dir0max) then
      dir0 <='1';
   else
      dir0 <='0';
   end if;
 end if;


 if (dir1min < dir1max) then     -- no addr boundary in range, normal compare
   if (iaddr > dir1min and iaddr < dir1max) then
      dir1 <='1';
   else
      dir1 <='0';
   end if;
 else               -- addr boundary in direction's range
   if (iaddr > dir1min or iaddr < dir1max) then
      dir1 <='1';
   else
      dir1 <='0';
   end if;
 end if;


end if;

end process;
end direction_arch;
```

```
--
--  fiforeg.vhd
--  Stores fifo state information
--

library IEEE;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity FIFOreg is
    port (
   inclk: in STD_LOGIC;
   outclk: in STD_LOGIC;
        iin: in STD_LOGIC;
        iout: in STD_LOGIC;
        oaddr: buffer unsigned (7 downto 0);
        iaddr: buffer unsigned (7 downto 0);
        full: out BOOLEAN;
        empty: buffer BOOLEAN;
      count: buffer unsigned (7 downto 0)
    );
end FIFOreg;

architecture FIFOreg_arch of FIFOreg is


begin

process (iaddr, oaddr)
begin

       count <= iaddr - oaddr;

end process;

process (inCLK)
begin

if(rising_edge(inclk)) then

   if(iin = '1') then
      iaddr <= iaddr + 1;
   end if;



end if;
end process;

process (outclk)
begin

if(falling_edge(outclk)) then

   if(iout = '1' and not empty) then
      oaddr <= oaddr + 1;
   end if;



end if;
end process;


process (count)
```

```
begin
       full <= (count = 240);
       empty <= (count = 0);
end process;

end FIFOreg_arch;
```

```
--
--  addrreg.vhd
--  Stores address associated with data in buffer
--

library IEEE;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ADDRreg is
    port (
          clk: in STD_LOGIC;
          reg: in BOOLEAN;

          count: in unsigned (7 downto 0);

          oaddr: out unsigned (31 downto 0);
          iaddr: in unsigned (31 downto 0)
    );
end ADDRreg;

architecture ADDRreg_arch of ADDRreg is

signal inaddr: unsigned (31 downto 0);
begin


process (CLK)
begin

if(rising_edge(clk)) then

   if reg then
      inaddr <= iaddr;
   end if;


end if;
end process;

process (count, inaddr)
begin

   oaddr(31 downto 1) <= inaddr(31 downto 1) - count  + 1;
   oaddr(0) <= '0';

end process;

end ADDRreg_arch;
```

```
--
--  PCI_togl.vhd
--  arbitrates fifo pair for fairness
--

library IEEE;
use IEEE.std_logic_1164.all;

entity PCI_TOGL is
    port (
        CLK: in STD_LOGIC;

        A_EN: out BOOLEAN;
        B_EN: out BOOLEAN;

        A_NEW: in BOOLEAN;
        A_READY: in BOOLEAN;

        B_NEW: in BOOLEAN;
        B_READY: in BOOLEAN;

        OUT_NEW: out BOOLEAN;
        OUT_READY: out BOOLEAN
    );
end PCI_TOGL;

architecture PCI_TOGL_arch of PCI_TOGL is
signal a: boolean;

begin
process (CLK)
begin
   if rising_edge(CLK) then

      if a then
      -- A is active
         A_EN <= TRUE;
         B_EN <= FALSE;

         OUT_READY <= A_READY;
         OUT_NEW <= A_NEW;

         if B_READY and A_NEW then
            a <= FALSE;
         end if;
      else
         A_EN <= FALSE;
         B_EN <= TRUE;

         OUT_READY <= B_READY;
         OUT_NEW <= B_NEW;

         if A_READY and B_NEW then
            a <= FALSE;
         end if;
      end if;
   end if;
end process;

end PCI_TOGL_arch;
```

```
--
--  srlatch.vhd
--  SR latch
--

library IEEE;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity latch is
  port ( Set: in BOOLEAN;
         Clear: in BOOLEAN;
         d: out BOOLEAN;
         g: out BOOLEAN);

end;

architecture latch_arch of latch is
begin

process (Set, Clear)
begin

if set then
   d <= true;
   g <= true;
elsif clear then
   d <= false;
   g <= true;
else
   g <= false;
end if;



end process;


end latch_arch;
```

```
--
--  parity.vhd
--  bytewise parity calculator
--

library IEEE;
use IEEE.std_logic_1164.all;

entity parity is
    port (
        din: in STD_LOGIC_VECTOR (31 downto 0);
        dout: out STD_LOGIC_VECTOR (3 downto 0)
    );
end parity;

architecture parity_arch of parity is
begin

dout(3) <= din(31) xor din(30) xor din(29) xor din(28) xor din(27) xor din(26)
xor din(25) xor din(24);
dout(2) <= din(23) xor din(22) xor din(21) xor din(20) xor din(19) xor din(18)
xor din(17) xor din(16);
dout(1) <= din(15) xor din(14) xor din(13) xor din(12) xor din(11) xor din(10)
xor din(9) xor din(8);
dout(0) <= din(7) xor din(6) xor din(5) xor din(4) xor din(3) xor din(2) xor
din(1) xor din(0);


end parity_arch;
```

```
--
--  16to32.vhd
--  converts two 16 bit words to one 32 bit word
--

library IEEE;
use IEEE.std_logic_1164.all;

entity to32 is
    port (CLK: in STD_LOGIC;
        din: in STD_LOGIC_VECTOR (15 downto 0);
        dout: out STD_LOGIC_VECTOR (31 downto 0)
    );
end to32;

architecture to32_arch of to32 is

signal state: BOOLEAN;

begin

process (CLK)
begin
   if rising_edge(CLK) then
      if state then
         state <= FALSE;
         dout(15 downto 0) <= din ;
      else
         state <= TRUE;
         dout(31 downto 16) <= din ;
      end if;
   end if;
end process;

end architecture;
```

```
--
--  32to16.vhd
--  converts one 32 bit word to two 16 bit words
--

library IEEE;
use IEEE.std_logic_1164.all;

entity to16 is
    port (CLK: in STD_LOGIC;
        din: in STD_LOGIC_VECTOR (31 downto 0);
        dout: out STD_LOGIC_VECTOR (15 downto 0)
    );
end to16;

architecture to16_arch of to16 is

signal state: BOOLEAN;

begin

process (CLK)
begin
   if rising_edge(CLK) then
      if state then
         state <= FALSE;
         dout <= din (15 downto 0);
      else
         state <= TRUE;
         dout <= din (31 downto 16);
      end if;
   end if;
end process;

end architecture;
```