```
--Signal naming conventions
-- ON- chip signals are not preficed
-- Upper part of the on-board bus signale preficed with             iu_
-- Lower part of the on-board bus signals preficed with             il_
-- Upper part of pc bus                                            pu_
-- Lower part of pc bus                                            pl_
-- Backplane bus                                                   bp_
-- fiber output                                                    fo_
-- fiber input                                                     fi_
--
--
-- take care of reading data from the onboard bus
-- direction: ibus_upper (BP data read) to FO_action
--
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity read_from_ibus is

   port (
    clk   : in  std_logic;
    reset : in  std_logic;
    iu_req   : in std_logic;
    iu_ack   : buffer  std_logic;
    iu_io    : in std_logic_vector(15 downto 0);
    data  : out  std_logic_vector(15 downto 0);
    req  : out  std_logic;
    ack  : in std_logic);

end read_from_ibus;

architecture read_from_ibus_arch of read_from_ibus is
type iu_read_state is (idle,hs,iu_hs1,iu_hs2);
signal iu_read : iu_read_state;
signal timeout : unsigned(3 downto 0);
begin  -- write_to_ibus_arch

  iu_reader: process (clk, reset)
  begin  -- process iu_writer
    if reset = '1' then                 -- asynchronous reset (active low)

    elsif clk'event and clk = '1' then  -- rising clock edge
      req <= '0';
      iu_ack <= '0';
      case iu_read is
        when idle =>
          if (iu_req = '1') then
            data <= iu_io;
            iu_ack <= '1';
            iu_read <= hs;
            timeout <= "0000";
          else
            iu_read <= idle;
          end if;
        when hs =>
          iu_ack <= '1';
          req <= '1';
          if (iu_req = '0') then
            iu_read <= iu_hs1;
            timeout <= "0000";
          else
            if (timeout = "1111") then
              iu_read <= iu_hs1;
              timeout <= "0000";
            else
              timeout <= timeout+1;
            end if;
          end if;
        when iu_hs1 =>
          req <= '1';
          if (ack = '1') then
            req <= '0';
            iu_read <= iu_hs2;
          else
            if (timeout = "1111") then
              iu_read <= idle;
            else
              timeout <= timeout+1;
            end if;
          end if;
        when iu_hs2 =>                  -- no timeout here because by this
                                        -- stage we know there is something
                                        -- at the other end
          if (ack = '0') then
            iu_read <= idle;
          end if;
        when others =>
          iu_read <= idle;
      end case;
    end if;
  end process iu_reader;
end read_from_ibus_arch;

-- This block reads data from the FI when addressed. It captures and
-- acts on command data and places other data or address info in the on
-- chip buffers.
-- Each transaction typically consists of two transfers, an address transfer
-- followed by a data transfer. There are a few exceptions where only an
-- address transfer is required, namelly a soft reset and switching to
-- loopback mode.
-- When the code detects an address transfer it check to see if it is being
-- addressed. If it is then enables itself. Thie chip stays enabled untill
-- another address transfer occurs. In this way the chip can accept follow
-- up data transfers.
--
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity read_from_FI is  -- to BP direction
  -- note internal upper bus is used - from FI=PC=upper
  port (
    clk       : in     std_logic;
    reset     : in     std_logic;
    req        : in     std_logic;
    ack        : out    std_logic;
    data     : in     std_logic_vector(15 downto 0);
    il_io      : out std_logic_vector(15 downto 0);
    address   : buffer std_logic_vector(7 downto 0);
    il_req       : buffer    std_logic;
    il_ack       : in     std_logic;
    i_am_remote:in     std_logic;
    my_data_address:in    std_logic_vector(3 downto 0);
    my_ctrl_address:in    std_logic_vector(3 downto 0);
loopback_state:buffer std_logic;
    request_reset:out     std_logic;
    FI_i_am_addressed: buffer std_logic;
    receive_enabled   : buffer std_logic
```

```
129     );
130
131 end read_from_FI;
132
133 architecture read_from_FI_arch of read_from_FI is
134 type FI_read_type is (idle,am_i_addressed,FI_hs_for_address,FI_hs_for_data,
135                       FI_use_ctrl_data,internal_ack1,internal_ack2);
136 signal FI_read_state : FI_read_type;
137 signal timeout : unsigned(3 downto 0);
138 signal this_is_a_ctrl_transaction : std_logic;
139 begin  -- read_from_FI_arch
140
141   -- purpose: read from the fiber receiver
142   -- type   : sequential
143   -- inputs : clk, reset
144   -- outputs:
145   read_FI: process (clk, reset)
146   begin  -- process read_ibus
147     if reset = '1' then                 -- asynchronous reset (active low)
148       FI_i_am_addressed <= '0';
149       this_is_a_ctrl_transaction <= '0';
150     elsif clk'event and clk = '1' then  -- rising clock edge
151       ack <= '0';
152       il_req <= '0';
153       request_reset <= '0';
154       receive_enabled <= '1';   -- this side is always enabled
155       loopback_state <= '0';  -- disable loopback
156       case (FI_read_state) is
157         when idle =>
158           ack <= '0';
159           if (req = '1') then
160             il_io <= data;
161             FI_read_state <= am_i_addressed;
162             timeout <= "0000";
163           end if;
164         when am_i_addressed =>
165           FI_read_state <= idle;
166           if (data(15) = '1') then -- this is an address
167             FI_i_am_addressed <= '0';   -- any addtress transaction should
168                                         -- reset select bits on all chips
169             this_is_a_ctrl_transaction <= '0';
170             if (data(12) = i_am_remote) then
171               if (data(11 downto 8) = my_ctrl_address) then
172                 this_is_a_ctrl_transaction <= '1';
173                 --                  FI_i_am_addressed <= '1';
174                 -- don't loopback                 loopback_state <= data(4);
175                 request_reset <= data(5);
176                 -- this side is always enabled -- receive_enabled <= data(0);
177                 address <= data(7 downto 0);
178                 FI_read_state <= FI_hs_for_address;
179               else
180                 if (data(11 downto 8) = my_data_address) then
181                 --                  FI_i_am_addressed <= '1';
182                 address <= data(7 downto 0);
183                 FI_read_state <= FI_hs_for_address;
184                 end if;
185               end if;
186             --
187             -- This is not an address, but if this chip is already addressed
188             -- then this will be a data transaction to complete the previous
189             -- address transaction
190             --
191             --            if (FI_i_am_addressed = '1') then
192               FI_read_state <= FI_hs_for_data;
193             --          else
194             --            FI_read_state <= internal_ack1; --
195             --          end if;
196
197           end if;
198         else
199         FI_read_state <= FI_hs_for_data;
200         end if;
201         when FI_hs_for_address =>
202           ack <= '1';
203           if (req = '0') then
204               FI_read_state <= idle;
205           else
206             if (timeout = "1111") then
207               -- Hmm this timeout will lead to a spurious transaction, but not
208               -- to worry, there must be a serious problem somewhere else in
209               -- this case
210               FI_read_state <= idle;
211             else
212               timeout <= timeout+1;
213             end if;
214           end if;
215         when FI_hs_for_data =>
216           ack <= '1';
217           if (req = '0') then
218             --
219             -- things to do once the external transaction has completed
220             --
221             if (this_is_a_ctrl_transaction = '1') then
222               FI_read_state <= FI_use_ctrl_data;
223             else
224               FI_read_state <= internal_ack1;
225               timeout <= "0000";
226             end if;
227           else
228             if (timeout = "1111")  then
229               --
230               -- If this timeout is triggered we are likely to keep looping.
231               -- By gong to an idle state we ensure that only 1 data transaction
232               -- can take place anyway, thought there is a possibility of
233               -- losing a transaction as well.
234               --
235               FI_read_state <= idle;
236             else
237               timeout <= timeout+1;
238             end if;
239           end if;
240           if (this_is_a_ctrl_transaction = '0') then
241             --
242             -- Get a jump start on completing internal transactions
243             --
244             il_req <= '1';  -- attempt to transfer data out
245           end if;
246         when FI_use_ctrl_data =>
247           --
248           -- What we do with the control data is completely dependent on the
249           -- chip all of the rest of this code should be fairly generic, if
250           -- a littl over elaborate in most cases
251           --
252           --if (address(1 downto 0) = "01") then
253             --c1: for i in 0 to 7 loop
254 --* put reset fifo here              --counter_start(i) <= data(i);
255               --counter_end(i) <= data(i+8);
256             --end loop c1;
```

```
257           --end if;
258             FI_read_state <= idle;
259       when internal_ack1 =>
260           il_req <= '1';
261           if (il_ack = '1') then
262             timeout <= "0000";
263             il_req <= '0';
264             FI_read_state <= internal_ack2;
265           else
266             if (timeout = "1111") then
267               FI_read_state <= idle;
268             else
269               timeout <= timeout+1;
270             end if;
271           end if;
272        when internal_ack2 =>
273           il_req <= '0';
274           if (il_ack = '0') then
275             FI_read_state <= idle;
276           else
277             if (timeout = "1111") then
278               FI_read_state <= idle;
279             else
280               timeout <= timeout+1;
281             end if;
282           end if;
283         when others =>
284           FI_read_state <= idle;
285       end case;
286     end if;
287   end process read_FI;
288 end read_from_FI_arch;
289
290
291 library ieee;
292 use ieee.std_logic_1164.all;
293 use work.numeric_std.all;
294
295 entity ibus_FO_action is
296     port (
297       clk : in std_logic;
298       reset : in std_logic;
299       il_i_am_addressed: in std_logic;
300       fo_req :  in std_logic;
301       fo_ack : buffer std_logic;
302       data :    in std_logic_vector(15 downto 0);
303       fiber_clk: in std_logic;
304       fo_d: out std_logic_vector(9 downto 0);
305       fo_ENA_l:  out std_logic
306     );
307
308 end ibus_FO_action;
309
310 architecture arch_ibus_FO_action of ibus_FO_action is
311   -- purpose: <description>
312    type states is (fo_idle,fo_byte1,fo_wait1,fo_byte2,fo_wait2);
313    signal state : states;
314    signal fiber_out_buf : std_logic_vector(15 downto 0);
315    signal byte_buf : std_logic_vector(7 downto 0);
316    signal fo_send : std_logic;      -- <comment>
317    alias sc : std_logic is fo_d(8);
318    alias svs : std_logic is fo_d(9);
319
320   -- purpose: performs the actual monitor action
321   -- type:    memorizing
322   -- inputs:  clk, reset
323   -- outputs:
324 begin
325   FO_action : process (clk, reset)
326   begin  -- process FO_action
327     if reset = '1' then
328     --  if reset_l = '0' then
329       -- activities triggered by rising edge of clock
330       elsif clk'event and clk = '1' then
331       fo_send <= '0';   -- enable needs to stay low for both bytes
332       case state is
333           when fo_idle =>
334           fo_d(8) <= '0';   -- command channel flag
335           sc <= '0';
336           svs <= '0';
337           fo_ack  <= '0';
338           if (fo_req = '1') then
339 --          if (il_i_am_addressed = '0') then
340               fo_d(7 downto 0) <= data(7 downto 0);
341               state <= fo_byte1;
342 --            if (fiber_clk = '1') then
343 --            -- if fiber clock is high it is now
344 --            -- transitioning to low, which
345 --            -- means data will be tranmitted
346 --            -- on next cycle. since that
347 --            -- will present a rising edge
348 --           --fo_not_strobe
349 --           state <= fo_wait1;
350 --            --
351 --            -- Hmm appears to be a bug in the component desciption
352 --            -- it appears that a low on ena any time that the
353 --            -- clock is high sends the data, rather than
354 --            -- the rising edge as written.
355         end if;
356           when fo_byte1 =>
357             if (fiber_clk = '1') then
358                 fo_ENA_l <= '0';
359                 state <= fo_wait1;
360             fo_ack  <= '1'; -- so other side has time to see it
361             else
362             state <= fo_byte1;
363             end if;
364           when fo_wait1 =>
365             fo_ENA_l <= '0';
366             state <= fo_byte2;
367           when fo_byte2 =>
368             fo_ENA_l <= '0';
369           --
370           -- here fiber clock transitioning from high to low
371           --
372           fo_d(7 downto 0) <=  data(15 downto 8);
373           state <= fo_wait2;
374           when fo_wait2 =>
375           -- final low to high
376           --
377           --
378           -- Give this fiber strobe time tp get back to high
379           --
380           state <= fo_idle;
381           when others =>
382           state <= fo_idle;
383       end case;
384       end if;
```

```
  385   end process FO_action;
  386 end arch_ibus_FO_action;
  387
  388
  389
  390 library ieee;
  391 use ieee.std_logic_1164.all;
  392 use work.numeric_std.all;
  393
  394 entity ibus_FI_port is
  395
  396   port (
  397     clk:        in     std_logic;
  398     reset :     in std_logic;
  399     data :      buffer std_logic_vector(15 downto 0);
  400     req   : buffer std_logic;
  401     ack   : in  std_logic;
  402     fiber_to_ibus_buf :    in std_logic_vector(15 downto 0);
  403     data_pump_word_ready   : in std_logic;
  404     refill_ibus_output_buf  : out std_logic
  405     );
  406
  407 end ibus_FI_port;
  408
  409 architecture arch_ibus_FI_port of ibus_FI_port is
  410   type read_states  is (idle,read_req,end_cycle);
  411   signal read_state: read_states;
  412   signal timeout : unsigned(3 downto 0);
  413 begin
  414     ibus_FI_read : process (clk, reset)
  415     begin
  416     if reset = '1' then
  417 --      refill_ibus_output_buf <= '1';
  418 refill_ibus_output_buf <= '0';
  419           -- asynchronous reset (active low)
  420     elsif clk'event and clk = '1' then
  421             req <= '0';
  422 refill_ibus_output_buf <= '1';
  423         case read_state is
  424         when  idle =>
  425          req <= '0';
  426          timeout <= "0000";
  427             if (data_pump_word_ready = '1') then
  428                 if (ack = '0') then
  429                     data(15 downto 0) <= fiber_to_ibus_buf(15 downto 0);
  430                     req <= '1';
  431 --                  refill_ibus_output_buf <= '0';
  432                     read_state <= read_req;
  433                 else
  434                 read_state <= idle;
  435                 end if;
  436             end if;
  437         when read_req =>
  438             if (ack = '1') then
  439             req <= '0';
  440             read_state <= end_cycle;
  441 else
  442   if (timeout = "1111") then
  443     read_state <= idle;
  444   else
  445     timeout <= timeout+1;
  446     read_state <= read_req;
  447   end if;
  448 end if;
  449         when end_cycle =>
  450 refill_ibus_output_buf <= '1';
  451             if (ack = '0') then
  452 --          refill_ibus_output_buf <= '1';
  453 refill_ibus_output_buf <= '0';
  454                 read_state <= idle;
  455             else
  456                 if (timeout = "1111") then
  457                 read_state <= idle;
  458                 else
  459                 timeout <= timeout+1;
  460                 read_state <= end_cycle;
  461                 end if;
  462             end if;
  463         when others =>
  464            read_state <= idle;
  465         end case;
  466     end if;
  467     end process ibus_FI_read;
  468 end arch_ibus_FI_port;
  469
  470
  471
  472 library ieee;
  473 use ieee.std_logic_1164.all;
  474 use work.numeric_std.all;
  475
  476 entity fiber_rec is
  477
  478     port (clk : in std_logic;
  479     reset :     in std_logic;
  480       fr_d:  in std_logic_vector(11 downto 0);  -- data bus from fiber rec
  481       fr_RDY_l: in std_logic;                   -- from fiber rec RDY
  482       fr_status: in std_logic;                  -- from fiber rec SO
  483       receive_enabled: in std_logic;
  484       increment_fifo_count: buffer std_logic;
  485       violation_count: buffer unsigned(7 downto 0);
  486       fifo_reset_l: buffer std_logic;           -- output to FIFO reset
  487       fifo_D: out std_logic_vector(8 downto 0); -- output to FIFO d inputs
  488       fifo_WRITE_l: out std_logic;              -- FIFO write strobe
  489       reset_fifo_request: in std_logic
  490       );
  491 end fiber_rec;
  492
  493 -- purpose: manages recept of fiber data and its storage in the fifo
  494 architecture arch_fiber_rec of fiber_rec is
  495   type fr_states  is (idle,fifo_latch);-- handshake states
  496   signal fr_state : fr_states;  -- handshake
  497   alias code_violation : std_logic is fr_d(9);
  498   alias control_byte : std_logic is fr_d(8);
  499 begin  -- fiber_rec_arch
  500     -- type:    memorizing
  501     -- inputs:  clk, reset
  502     -- outputs: <signal names>
  503 -- had to just pass thru receiver data to FIFO input and RDY to write strobe
  504 -- to meet FIFO WR strobe min of 30nsec. Receiver has 60/40% duty cycle.
  505 -- Otherwise we would have a 20nsec write strobe.  Code could be cleaned
  506 -- up later, but leaving state machine in for now....
  507 fifo_D <= fr_d(8 downto 0);
  508 fifo_WRITE_l <= fr_RDY_l;
  509     fr : process (clk, fr_RDY_l,reset_fifo_request,reset)
  510     begin  -- process fr
  511      if (reset_fifo_request = '1' or reset = '1') then
  512         fifo_reset_l <= '0';
```

```
513         -- activities triggered by asynchronous reset (active low)
514     elsif clk'event and clk = '1' then
515         case fr_state is
516         when idle =>
517             if (reset_fifo_request = '1') then
518             fifo_reset_l <= '0';
519             violation_count <= (others => '0');
520             else
521             if (fr_status = '1' and receive_enabled = '1') then
522                 if ( fr_RDY_l = '0') then
523                 fr_state <= fifo_latch;
524                 increment_fifo_count <= '1';
525                 fifo_reset_l <= '1';
526                 end if;
527             end if;
528             end if;
529         when fifo_latch =>
530             if (reset_fifo_request = '1') then
531             fifo_reset_l <= '0';
532             end if;
533             fr_state <= idle;
534         when others =>
535             fr_state <= idle;
536         end case;
537     end if;
538     end process fr;
539 end arch_fiber_rec;
540
541
542 library ieee;
543 use ieee.std_logic_1164.all;
544 use work.numeric_std.all;
545
546 entity fifo_data_pump is
547
548     port (clk : in std_logic;
549       reset : in std_logic;
550       violation_count: in unsigned(7 downto 0);
551       fiber_to_ibus_buf: buffer std_logic_vector(15 downto 0);
552       fifo_OUT: in std_logic_vector(8 downto 0);
553       fifo_READ_l: out std_logic;
554       data_pump_word_ready: buffer std_logic;
555       refill_ibus_output_buf: in std_logic;
556       fifo_reset_l: in std_logic;
557       fifo_EMPTY_l: in std_logic
558       );
559 end fifo_data_pump;
560
561 -- purpose: pump data out of the fifo
562 architecture arch_fifo_data_pump of fifo_data_pump is
563   type pump_states  is (idle,fifo_strobe,fifo_read_data,fifo_wait_on_empty);-- handshake states
564   signal pump_state : pump_states;  -- handshake
565 signal count : unsigned(1 downto 0);
566 begin  -- fifo_data_pump_arch
567     -- purpose: <description>
568
569     pump : process (clk,reset)
570
571     begin  -- process pump
572     -- activities triggered by asynchronous reset (active low)
573     if reset = '1' then
574         fifo_READ_l <= '1';
575     elsif clk'event and clk = '1' then
576         data_pump_word_ready <= '0';
577         fifo_READ_l <= '1';
578         case pump_state is
579         when idle =>
580             count <= "00";
581             if (fifo_reset_l = '1' or refill_ibus_output_buf = '1') then
582                 data_pump_word_ready <= '0';
583                 if (fifo_empty_l = '0') then
584                 -- This is an inverted signal so here there is
585                 -- no data in the fifo, must now wait for data
586                 -- to become available
587                     pump_state <= fifo_wait_on_empty;
588                 else
589 --                  pump_state <= fifo_strobe;
590                     pump_state <= fifo_wait_on_empty;   -- add a delay to make sure FIFO is ready
591                 end if;
592             else
593             pump_state <= idle;
594             end if;
595         when fifo_wait_on_empty =>
596             data_pump_word_ready <= '0';  --- may need to chsnge to 1
597             if (fifo_EMPTY_l = '0') then
598             pump_state <= fifo_wait_on_empty;
599             else
600             pump_state <= fifo_strobe;
601             end if;
602         when fifo_strobe =>
603 if (fifo_empty_l = '0') then
604 pump_state <= fifo_wait_on_empty;
605 else
606             data_pump_word_ready <= '0';
607             fifo_READ_l <= '0';
608             pump_state <= fifo_read_data;
609 end if;
610         when fifo_read_data =>
611             if (fifo_out(8) = '1') then
612                 count <= "00";
613                 fifo_READ_l <= '1';
614                 pump_state <= fifo_strobe;
615             else
616                 fiber_to_ibus_buf(15 downto 8) <= fifo_out(7 downto 0);
617                 fiber_to_ibus_buf(7 downto 0) <=
618                         fiber_to_ibus_buf(15 downto 8);
619                 if (count  = "01") then
620                 data_pump_word_ready <= '1';
621                 pump_state <= idle;
622                 else
623                     count <= count +1;
624                     fifo_READ_l <= '1';
625                     if (fifo_empty_l = '0') then
626                         pump_state <= fifo_wait_on_empty;
627                     else
628                         pump_state <= fifo_strobe;
629                     end if;
630                 end if;
631             end if;
632         when others =>
633             pump_state <= idle;
634         end case;
635 end if;
636     end process pump;
637 end arch_fifo_data_pump;
638
639
640
```

```
641 library ieee;
642 use ieee.std_logic_1164.all;
643 use work.numeric_std.all;
644
645 entity tout is
646   port (
647       clk : in std_logic;
648       fast : in std_logic;
649       slow : in std_logic;
650       DEBUG : buffer std_logic;
651       --
652       -- on board data bus and associated control signals
653       --
654       id: inout std_logic_vector(31 downto 0);
655       --
656       -- global
657       --
658       reset: in std_logic;
659       AO_FROM_PC_STROBE: buffer std_logic;
660       AO_FROM_PC_ACK: in std_logic;
661       AO_TO_PC_STROBE: in std_logic;
662       AO_TO_PC_ACK: buffer std_logic;
663       in_strobe: in std_logic;
664       -- fiber reciever
665       --
666       fr_d: in std_logic_vector(11 downto 0);
667       fr_ref_clk: out std_logic;
668       fr_rf:      buffer std_logic;
669       fr_mode:    out std_logic;
670       fr_status:  in std_logic;
671       fr_RDY_l: in std_logic;
672       fr_ckr:     in std_logic;
673       --
674       -- fiber output section
675       --
676       fo_d: out std_logic_vector(9 downto 0);
677       fo_ENN_l: out std_logic;
678       fo_ENA_l: out std_logic;
679       fo_CKW:   buffer std_logic;
680       fo_mode:    out std_logic;
681       fo_foto:    out std_logic;
682       fo_RP_l:  in  std_logic;
683       --
684       -- fifo stuff
685       --
686       fifo_reset_l: buffer std_logic;   -- Declare as a buffer since we want
687       fifo_WRITE_l: out std_logic;
688       fifo_D:    out  std_logic_vector(8 downto 0);
689       fifo_READ_l:  out std_logic;
690       fifo_FULL_l:  in std_logic;
691       fifo_HALF_l:  in std_logic;
692       fifo_EMPTY_l: in std_logic;
693       fifo_OUT:   in std_logic_vector(8 downto 0)
694     );
695 end tout;
696
697
698 -- purpose: Collect together preceeding modules
699 architecture tout_arch of tout is
700
701     signal write_oe : std_logic;
702     signal read_oe : std_logic;
703     signal fo_data_strobe : std_logic;
704     signal violation_count : unsigned(7 downto 0);
705     signal decriment_fifo_count : std_logic;
706     signal increment_fifo_count : std_logic;
707     signal data_pump_word_ready:    std_logic;
708     signal refill_ibus_output_buf: std_logic;
709     signal fiber_to_ibus_buf : std_logic_vector(15 downto 0);
710     signal reset_fifo : std_logic;
711     signal rec_enabled : std_logic;
712
713     type tristate_state_type is (idle,active);
714 signal tristate_state : tristate_state_type;
715 signal il_tristate : std_logic;
716 signal il_ack: std_logic;
717 signal i_am_remote : std_logic;        -- set to 1 for remote board 0 for local
718 signal my_data_address : std_logic_vector(3 downto 0);
719 signal my_ctrl_address : std_logic_vector(3 downto 0);
720 signal from_FI_ack : std_logic;
721 signal from_FI_req : std_logic;
722 signal data_from_pc : std_logic_vector(15 downto 0);
723 signal address_from_pc : std_logic_vector(7 downto 0);
724 signal power_up_reset: std_logic;
725 signal soft_request_reset: std_logic;
726 signal data_to_FO : std_logic_vector(15 downto 0);
727 signal this_chip_selected: std_logic;
728 signal FI_address : std_logic_vector(7 downto 0);
729 signal FI_to_ibus_req : std_logic;
730 signal FI_to_ibus_ack : std_logic;
731 signal FI_data: std_logic_vector(15 downto 0);
732 signal write_to_FO_req : std_logic;
733 signal write_to_FO_ack : std_logic;
734 -- alias reset_fifo : std_logic is request_reset;
735
736 begin
737     --    reset <= '1';             -- negative logic
738     DEBUG <= AO_FROM_PC_STROBE;
739     i_am_remote <= '1';
740     my_data_address <= "1111";
741     my_ctrl_address <= "0010";
742   --
743     -- Fiber output section
744     --
745     outstrobe : process (clk,reset)
746     begin  -- process outstrobe
747     -- activities triggered by asynchronous reset (active low)
748     if clk'event and clk = '1' then
749         fo_data_strobe <= not fo_data_strobe;
750     end if;
751     end process outstrobe;
752     fo_CKW <= fo_data_strobe;
753     fo_mode <= '0';
754     fo_foto <= '0';
755     fo_ENN_l <= '1';
756
757    ibus_reader : read_from_ibus port map (
758      clk       => clk,
759      reset     => reset,
760      iu_req    => AO_TO_PC_STROBE,  --  in
761      iu_ack    => AO_TO_PC_ACK,     --  out
762      iu_io     => id(31 downto 16),
763      data      => data_to_FO,
764      req       => write_to_FO_req,       -- out
765      ack       => write_to_FO_ack        -- in
766      );
767     -- direction: received FI=PC writes output to ibus_lower
768   FI_reader : read_from_FI port map (
```

```
769     clk             => clk,
770     reset           => reset,
771     req             => FI_to_ibus_req,  -- in
772     ack             => FI_to_ibus_ack,  -- out
773     data            => FI_data,
774     il_io           => id(15 downto 0),
775     address         => address_from_pc,
776     il_req          => AO_FROM_PC_STROBE,   -- out
777     il_ack          => AO_FROM_PC_ACK,      -- in
778     i_am_remote     => i_am_remote,
779     my_ctrl_address => my_ctrl_address,
780     my_data_address => my_data_address,
781     request_reset   => reset_fifo,
782     FI_i_am_addressed => this_chip_selected,
783     receive_enabled   => rec_enabled
784     );
785
786     --
787     ibus_FO: ibus_FO_action port map (
788         clk             => clk,
789         reset           => reset,
790         il_i_am_addressed => this_chip_selected,
791         fo_req          => write_to_FO_req, -- in
792         fo_ack          => write_to_FO_ack, -- out
793         data            => data_to_FO,
794         fiber_clk       => fo_CKW,
795         fo_d            => fo_d,
796         fo_ENA_l        => fo_ENA_l
797         );
798     --
799     -- fiber receiver stuff
800     --
801     fr_ref_clk <= fo_data_strobe;
802     fr_mode <= '0';
803     fr_rf <= '1';
804     ibus_FI: ibus_FI_port port map (
805         clk             => clk,
806         reset           => reset,
807         data            => FI_data,
808         req             => FI_to_ibus_req,  -- out
809         ack             => FI_to_ibus_ack,  -- in
810         fiber_to_ibus_buf       => fiber_to_ibus_buf,
811         data_pump_word_ready    => data_pump_word_ready,
812         refill_ibus_output_buf  => refill_ibus_output_buf
813         );
814
815     fr_imp: fiber_rec port map (
816         clk             => clk,
817         reset           => reset,
818         fr_d            => fr_d,
819         fr_RDY_l        => fr_RDY_l,
820         fr_status       => fr_status,
821         receive_enabled => rec_enabled,
822         increment_fifo_count    => increment_fifo_count,
823         violation_count         => violation_count,
824         fifo_reset_l            => fifo_reset_l,
825         fifo_D          => fifo_D,
826         fifo_WRITE_l    =>  fifo_WRITE_l,
827         reset_fifo_request      => reset_fifo
828                 );
829
830     pump: fifo_data_pump port map (
831         clk             => clk,
832         reset           => reset,
833         violation_count =>  violation_count,
834         fiber_to_ibus_buf =>  fiber_to_ibus_buf,
835         fifo_OUT        => fifo_OUT,
836         fifo_READ_l     => fifo_READ_l,
837         data_pump_word_ready => data_pump_word_ready,
838         refill_ibus_output_buf => refill_ibus_output_buf,
839         fifo_reset_l    =>  fifo_reset_l,
840         fifo_EMPTY_l => fifo_EMPTY_l
841                 );
842     --
843     end  tout_arch;
844
845
846
847
848
849
850
851
852
853
854
855
856
857
```