```
  1 --Signal naming conventions
  2 -- ON- chip signals are not preficed
  3 -- Upper part of the on-board bus signale preficed with             iu_
  4 -- Lower part of the on-board bus signals preficed with             il_
  5 -- Upper part of pc bus                                            pu_
  6 -- Lower part of pc bus                                            pl_
  7 -- Backplane bus                                                   bp_
  8 -- fiber output                                                    fo_
  9 -- fiber input                                                     fi_
 10 --
 11 --
 12 -- take care of reading data from the onboard bus
 13 -- direction: ibus_upper (BP data read) to FO_action
 14 --
 15 library ieee;
 16 use ieee.std_logic_1164.all;
 17 use work.numeric_std.all;
 18
 19 entity read_from_ibus is
 20
 21    port (
 22     clk   : in  std_logic;
 23     reset : in  std_logic;
 24     iu_req   : in std_logic;
 25     iu_ack   : buffer  std_logic;
 26     iu_io    : in std_logic_vector(15 downto 0);
 27     data  : out  std_logic_vector(15 downto 0);
 28     req  : out  std_logic;
 29     ack  : in std_logic);
 30
 31 end read_from_ibus;
 32
 33 architecture read_from_ibus_arch of read_from_ibus is
 34 type iu_read_state is (idle,hs,iu_hs1,iu_hs2);
 35 signal iu_read : iu_read_state;
 36 signal timeout : unsigned(3 downto 0);
 37 begin  -- write_to_ibus_arch
 38
 39   iu_reader: process (clk, reset)
 40   begin  -- process iu_writer
 41     if reset = '1' then                 -- asynchronous reset (active low)
 42
 43     elsif clk'event and clk = '1' then  -- rising clock edge
 44       req <= '0';
 45       iu_ack <= '0';
 46       case iu_read is
 47         when idle =>
 48           if (iu_req = '1') then
 49             data <= iu_io;
 50             iu_ack <= '1';
 51             iu_read <= hs;
 52             timeout <= "0000";
 53           else
 54             iu_read <= idle;
 55           end if;
 56         when hs =>
 57           iu_ack <= '1';
 58           req <= '1';
 59           if (iu_req = '0') then
 60             iu_read <= iu_hs1;
 61             timeout <= "0000";
 62           else
 63             if (timeout = "1111") then
 64               iu_read <= iu_hs1;
 65               timeout <= "0000";
 66             else
 67               timeout <= timeout+1;
 68             end if;
 69           end if;
 70         when iu_hs1 =>
 71           req <= '1';
 72           if (ack = '1') then
 73             req <= '0';
 74             iu_read <= iu_hs2;
 75           else
 76             if (timeout = "1111") then
 77               iu_read <= idle;
 78             else
 79               timeout <= timeout+1;
 80             end if;
 81           end if;
 82         when iu_hs2 =>                  -- no timeout here because by this
 83                                         -- stage we know there is something
 84                                         -- at the other end
 85           if (ack = '0') then
 86             iu_read <= idle;
 87           end if;
 88         when others =>
 89           iu_read <= idle;
 90       end case;
 91     end if;
 92   end process iu_reader;
 93 end read_from_ibus_arch;
 94
 95 -- This block reads data from the FI when addressed. It captures and
 96 -- acts on command data and places other data or address info in the on
 97 -- chip buffers.
 98 -- Each transaction typically consists of two transfers, an address transfer
 99 -- followed by a data transfer. There are a few exceptions where only an
100 -- address transfer is required, namelly a soft reset and switching to
101 -- loopback mode.
102 -- When the code detects an address transfer it check to see if it is being
103 -- addressed. If it is then enables itself. Thie chip stays enabled untill
104 -- another address transfer occurs. In this way the chip can accept follow
105 -- up data transfers.
106 --
107 library ieee;
108 use ieee.std_logic_1164.all;
109 use work.numeric_std.all;
110 entity read_from_FI is  -- to BP direction
111   -- note internal upper bus is used - from FI=PC=upper
112   port (
113     clk       : in     std_logic;
114     reset     : in     std_logic;
115     req        : in     std_logic;
116     ack        : out    std_logic;
117     data     : in     std_logic_vector(15 downto 0);
118     il_io      : out std_logic_vector(15 downto 0);
119     address   : buffer std_logic_vector(7 downto 0);
120     il_req       : buffer    std_logic;
121     il_ack       : in     std_logic;
122     i_am_remote:in     std_logic;
123     my_data_address:in    std_logic_vector(3 downto 0);
```

```
124     my_ctrl_address:in    std_logic_vector(3 downto 0);
125 loopback_state:buffer std_logic;
126     request_reset:out     std_logic;
127     FI_i_am_addressed: buffer std_logic;
128     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       -- by default load counter
151       -- counter_start <= "11111111";
152       -- counter_end <= "11111111";
153       -- for i in 7 downto 0 loop
154         --
155       -- end loop;  -- i
156     elsif clk'event and clk = '1' then  -- rising clock edge
157       ack <= '0';
158       il_req <= '0';
159       request_reset <= '0';
160       receive_enabled <= '1';   -- this side is always enabled
161       loopback_state <= '0';  -- disable loopback
162       case (FI_read_state) is
163         when idle =>
164           ack <= '0';
165           if (req = '1') then
166             il_io <= data;
167             FI_read_state <= am_i_addressed;
168             timeout <= "0000";
169           end if;
170         when am_i_addressed =>
171           FI_read_state <= idle;
172           if (data(15) = '1') then -- this is an address
173             FI_i_am_addressed <= '0';   -- any addtress transaction should
174                                         -- reset select bits on all chips
175 --            this_is_a_ctrl_transaction <= '0';
176             if (data(12) = i_am_remote) then
177               if (data(11 downto 8) = my_ctrl_address) then
178                 this_is_a_ctrl_transaction <= '1';
179 --                FI_i_am_addressed <= '1';
180 -- don't loopback                loopback_state <= data(4);
181                 request_reset <= data(5);
182 -- this side is always enabled -- receive_enabled <= data(0);
183                 address <= data(7 downto 0);
184                 FI_read_state <= FI_hs_for_address;
185               else
186                 if (data(11 downto 8) = my_data_address) then
187 --                FI_i_am_addressed <= '1';
188                 address <= data(7 downto 0);
189                 FI_read_state <= FI_hs_for_address;
190                 end if;
191               end if;
192             else
193             if (data(12) = not i_am_remote) then
194                 this_is_a_ctrl_transaction <= '0';
195             --
196             -- This is not an address, but if this chip is already addressed
197             -- then this will be a data transaction to complete the previous
198             -- address transaction
199             --
200 --            if (FI_i_am_addressed = '1') then
201               FI_read_state <= FI_hs_for_data;
202 --           else
203 --             FI_read_state <= internal_ack1; --
204 --           end if;
205             end if;
206           end if;
207         else
208         FI_read_state <= FI_hs_for_data;
209         end if;
210         when FI_hs_for_address =>
211           ack <= '1';
212           if (req = '0') then
213               FI_read_state <= idle;
214           else
215             if (timeout = "1111") then
216               -- Hmm this timeout will lead to a spurious transaction, but not
217               -- to worry, there must be a serious problem somewhere else in
218               -- this case
219               FI_read_state <= idle;
220             else
221               timeout <= timeout+1;
222             end if;
223           end if;
224         when FI_hs_for_data =>
225           ack <= '1';
226           if (req = '0') then
227             --
228             -- things to do once the external transaction has completed
229             --
230             if (this_is_a_ctrl_transaction = '1') then
231               FI_read_state <= FI_use_ctrl_data;
232             else
233               FI_read_state <= internal_ack1;
234               timeout <= "0000";
235             end if;
236           else
237             if (timeout = "1111")  then
238               --
239               -- If this timeout is triggered we are likely to keep looping.
240               -- By gong to an idle state we ensure that only 1 data transaction
241               -- can take place anyway, thought there is a possibility of
242               -- losing a transaction as well.
243               --
244               FI_read_state <= idle;
245             else
246               timeout <= timeout+1;
```

```
247             end if;
248           end if;
249           if (this_is_a_ctrl_transaction = '0') then
250             --
251             -- Get a jump start on completing internal transactions
252             --
253             il_req <= '1';  -- attempt to transfer data out
254           end if;
255         when FI_use_ctrl_data =>
256           --
257           -- What we do with the control data is completely dependent on the
258           -- chip all of the rest of this code should be fairly generic, if
259           -- a littl over elaborate in most cases
260           --
261           --if (address(1 downto 0) = "01") then
262             --c1: for i in 0 to 7 loop
263 --* put reset fifo here              --counter_start(i) <= data(i);
264               --counter_end(i) <= data(i+8);
265             --end loop c1;
266           --end if;
267             FI_read_state <= idle;
268       when internal_ack1 =>
269           il_req <= '1';
270           if (il_ack = '1') then
271             timeout <= "0000";
272             il_req <= '0';
273             FI_read_state <= internal_ack2;
274           else
275             if (timeout = "1111") then
276               FI_read_state <= idle;
277             else
278               timeout <= timeout+1;
279             end if;
280           end if;
281        when internal_ack2 =>
282           il_req <= '0';
283           if (il_ack = '0') then
284             FI_read_state <= idle;
285           else
286             if (timeout = "1111") then
287               FI_read_state <= idle;
288             else
289               timeout <= timeout+1;
290             end if;
291           end if;
292         when others =>
293           FI_read_state <= idle;
294       end case;
295     end if;
296   end process read_FI;
297 end read_from_FI_arch;
298
299
300 library ieee;
301 use ieee.std_logic_1164.all;
302 use work.numeric_std.all;
303
304 entity ibus_FO_action is
305     port (
306       clk : in std_logic;
307       reset : in std_logic;
308       il_i_am_addressed: in std_logic;
309       fo_req :  in std_logic;
310       fo_ack : buffer std_logic;
311       data :    in std_logic_vector(15 downto 0);
312       fiber_clk: in std_logic;
313       fo_d: out std_logic_vector(9 downto 0);
314       fo_ENA_l:  out std_logic
315     );
316
317 end ibus_FO_action;
318
319 architecture arch_ibus_FO_action of ibus_FO_action is
320   -- purpose: <description>
321    type states is (fo_idle,fo_byte1,fo_wait1,fo_byte2,fo_wait2);
322    signal state : states;
323    signal fiber_out_buf : std_logic_vector(15 downto 0);
324    signal byte_buf : std_logic_vector(7 downto 0);
325    signal fo_send : std_logic;      -- <comment>
326    alias sc : std_logic is fo_d(8);
327    alias svs : std_logic is fo_d(9);
328
329   -- purpose: performs the actual monitor action
330   -- type:    memorizing
331   -- inputs:  clk, reset
332   -- outputs:
333 begin
334   FO_action : process (clk, reset)
335   begin  -- process FO_action
336     if reset = '1' then
337     --  if reset_l = '0' then
338       -- activities triggered by rising edge of clock
339       elsif clk'event and clk = '1' then
340       fo_send <= '0';   -- enable needs to stay low for both bytes
341       case state is
342           when fo_idle =>
343           fo_d(8) <= '0';   -- command channel flag
344           sc <= '0';
345           svs <= '0';
346           fo_ack  <= '0';
347           if (fo_req = '1') then
348 --          if (il_i_am_addressed = '0') then
349             fo_d(7 downto 0) <= data(7 downto 0);
350 state <= fo_byte1;
351 --            if (fiber_clk = '1') then
352 --             -- if fiber clock is high it is now
353 --             -- transitioning to low, which
354 --             -- means data will be tranmitted
355 --             -- on next cycle. since that
356 --             -- will present a rising edge
357 --           --fo_not_strobe
358 --           state <= fo_wait1;
359 --           --
360 --           -- Hmm appears to be a bug in the component desciption
361 --           -- it appears that a low on ena any time that the
362 --           -- clock is high sends the data, rather than
363 --           -- the rising edge as written.
364 --           -- fo_send <= '1';
365 --         fo_send <= '1';
366 --          else
367 --          state <= fo_byte1;
368 --          end if;
369 --        else
```

```
370 --            state <= fo_idle;
371 --        end if;
372         end if;
373           when fo_byte1 =>
374 if (fiber_clk = '1') then
375 fo_ENA_l <= '0';
376 --          fo_send <= '1';
377             state <= fo_wait1;
378           fo_ack  <= '1';   -- so other side has time to see it
379 else
380 state <= fo_byte1;
381 end if;
382           when fo_wait1 =>
383 --          fo_send <= '1';
384 fo_ENA_l <= '0';
385             state <= fo_byte2;
386           when fo_byte2 =>
387 fo_ENA_l <= '0';
388           --
389           -- here fiber clock transitioning from high to low
390           --
391 --        fo_send <= '1';
392           fo_d(7 downto 0) <=  data(15 downto 8);
393           state <= fo_wait2;
394           when fo_wait2 =>
395           -- final low to high
396           --
397           --
398           -- Give this fiber strobe time tp get back to high
399           --
400 --        fo_send <= '1';
401           state <= fo_idle;
402           when others =>
403           state <= fo_idle;
404       end case;
405
406       end if;
407   end process FO_action;
408 --    fo_ENA_l <= '0' when (fo_send = '1') else '1';
409 end arch_ibus_FO_action;
410
411
412
413 library ieee;
414 use ieee.std_logic_1164.all;
415 use work.numeric_std.all;
416
417 entity ibus_FI_port is
418
419   port (
420     clk:        in     std_logic;
421     reset :     in std_logic;
422     data :      buffer std_logic_vector(15 downto 0);
423     req   : buffer std_logic;
424     ack   : in  std_logic;
425     fiber_to_ibus_buf :    in std_logic_vector(15 downto 0);
426     data_pump_word_ready   : in std_logic;
427     refill_ibus_output_buf  : out std_logic
428     );
429
430 end ibus_FI_port;
431
432 architecture arch_ibus_FI_port of ibus_FI_port is
433   type read_states  is (idle,read_req,end_cycle);
434   signal read_state: read_states;
435   signal timeout : unsigned(3 downto 0);
436 begin
437     ibus_FI_read : process (clk, reset)
438     begin
439     if reset = '1' then
440 --      refill_ibus_output_buf <= '1';
441 refill_ibus_output_buf <= '0';
442           -- asynchronous reset (active low)
443     elsif clk'event and clk = '1' then
444             req <= '0';
445 refill_ibus_output_buf <= '1';
446         case read_state is
447         when  idle =>
448          req <= '0';
449          timeout <= "0000";
450            if (data_pump_word_ready = '1') then
451                if (ack = '0') then
452                    data(15 downto 0) <= fiber_to_ibus_buf(15 downto 0);
453                    req <= '1';
454 --                   refill_ibus_output_buf <= '0';
455                    read_state <= read_req;
456                else
457                read_state <= idle;
458                end if;
459            end if;
460         when read_req =>
461             if (ack = '1') then
462             req <= '0';
463             read_state <= end_cycle;
464 else
465   if (timeout = "1111") then
466     read_state <= idle;
467   else
468     timeout <= timeout+1;
469     read_state <= read_req;
470   end if;
471 end if;
472         when end_cycle =>
473 refill_ibus_output_buf <= '1';
474             if (ack = '0') then
475 --          refill_ibus_output_buf <= '1';
476 refill_ibus_output_buf <= '0';
477                 read_state <= idle;
478             else
479                 if (timeout = "1111") then
480                 read_state <= idle;
481                 else
482                 timeout <= timeout+1;
483                 read_state <= end_cycle;
484                 end if;
485             end if;
486         when others =>
487            read_state <= idle;
488         end case;
489     end if;
490     end process ibus_FI_read;
491 end arch_ibus_FI_port;
492
```

```
493
494
495 library ieee;
496 use ieee.std_logic_1164.all;
497 use work.numeric_std.all;
498
499 entity fiber_rec is
500
501     port (clk : in std_logic;
502       fr_d:  in std_logic_vector(11 downto 0);  -- data bus from fiber rec
503       fr_RDY_l: in std_logic;                   -- from fiber rec RDY
504       fr_status: in std_logic;                  -- from fiber rec SO
505       receive_enabled: in std_logic;
506       increment_fifo_count: buffer std_logic;
507       violation_count: buffer unsigned(7 downto 0);
508       fifo_reset_l: buffer std_logic;           -- output to FIFO reset
509       fifo_D: out std_logic_vector(8 downto 0); -- output to FIFO d inputs
510       fifo_WRITE_l: out std_logic;              -- FIFO write strobe
511       reset_fifo_request: in std_logic
512       );
513 end fiber_rec;
514
515 -- purpose: manages recept of fiber data and its storage in the fifo
516 architecture arch_fiber_rec of fiber_rec is
517   type fr_states  is (idle,fifo_latch);-- handshake states
518   signal fr_state : fr_states;  -- handshake
519   alias code_violation : std_logic is fr_d(9);
520   alias control_byte : std_logic is fr_d(8);
521 begin  -- fiber_rec_arch
522     -- type:    memorizing
523     -- inputs:  clk, reset
524     -- outputs: <signal names>
525     fr : process (clk, fr_RDY_l)
526     begin  -- process fr
527     -- activities triggered by asynchronous reset (active low)
528     if clk'event and clk = '1' then
529         fifo_reset_l <= '1';
530         increment_fifo_count <= '0';
531         fifo_WRITE_l <= '1';
532         case fr_state is
533         when idle =>
534             if (reset_fifo_request = '1') then
535             fifo_reset_l <= '0';
536             violation_count <= (others => '0');
537             else
538             if (fr_status = '1' and receive_enabled = '1') then
539                 if (fr_RDY_l = '0') then
540 --                  if (code_violation = '1') then
541 --                  violation_count <= violation_count+1;
542 --                  end if;
543                 fifo_D <= fr_d(8 downto 0);
544                 fr_state <= fifo_latch;
545                 increment_fifo_count <= '1';
546                 end if;
547             end if;
548             end if;
549         when fifo_latch =>
550             if (reset_fifo_request = '1') then
551             fifo_reset_l <= '0';
552             else
553             fifo_WRITE_l <= '0';
554             end if;
555             fr_state <= idle;
556         when others =>
557             fr_state <= idle;
558         end case;
559     end if;
560     end process fr;
561 end arch_fiber_rec;
562
563
564 library ieee;
565 use ieee.std_logic_1164.all;
566 use work.numeric_std.all;
567
568 entity fifo_data_pump is
569
570     port (clk : in std_logic;
571       reset : in std_logic;
572       violation_count: in unsigned(7 downto 0);
573       fiber_to_ibus_buf: buffer std_logic_vector(15 downto 0);
574       fifo_OUT: in std_logic_vector(8 downto 0);
575       fifo_READ_l: out std_logic;
576       data_pump_word_ready: buffer std_logic;
577       refill_ibus_output_buf: in std_logic;
578       fifo_reset_l: in std_logic;
579       fifo_EMPTY_l: in std_logic
580       );
581 end fifo_data_pump;
582
583 -- purpose: pump data out of the fifo
584 architecture arch_fifo_data_pump of fifo_data_pump is
585   type pump_states  is (idle,fifo_strobe,fifo_read_data,fifo_wait_on_empty);-- handshake states
586   signal pump_state : pump_states;  -- handshake
587 signal count : unsigned(1 downto 0);
588 begin  -- fifo_data_pump_arch
589     -- purpose: <description>
590
591     pump : process (clk,reset)
592
593     begin  -- process pump
594     -- activities triggered by asynchronous reset (active low)
595 if reset = '1' then
596 fifo_READ_l <= '1';
597     elsif clk'event and clk = '1' then
598         --debug <= '0';
599 data_pump_word_ready <= '0';
600 --      data_pump_word_ready <= '1';
601         fifo_READ_l <= '1';
602         case pump_state is
603         when idle =>
604 count <= "00";
605 --          fifo_READ_l <= '1';
606 --          data_pump_word_ready <= '1';
607             if (fifo_reset_l = '1' or refill_ibus_output_buf = '1') then
608             data_pump_word_ready <= '0';
609             if (fifo_empty_l = '0') then
610                 -- This is an inverted signal so here there is
611                 -- no data in the fifo, must now wait for data
612                 -- to bcome available
613                 pump_state <= fifo_wait_on_empty;
614                 else
615                 pump_state <= fifo_strobe;
```

```
616                 end if;
617 else
618 pump_state <= idle;
619             end if;
620         when fifo_wait_on_empty =>
621             data_pump_word_ready <= '0';  --- may need to chsnge to 1
622             if (fifo_EMPTY_l = '0') then
623             pump_state <= fifo_wait_on_empty;
624             else
625             pump_state <= fifo_strobe;
626             end if;
627         when fifo_strobe =>
628             data_pump_word_ready <= '0';
629             fifo_READ_l <= '0';
630             pump_state <= fifo_read_data;
631         when fifo_read_data =>
632 if (fifo_out(8) = '1') then
633 count <= "00";
634 fifo_READ_l <= '1';
635 pump_state <= fifo_strobe;
636 else
637  fiber_to_ibus_buf(15 downto 8) <= fifo_out(7 downto 0);
638  fiber_to_ibus_buf(7 downto 0) <=
639                             fiber_to_ibus_buf(15 downto 8);
640 if (count  = "01") then
641 data_pump_word_ready <= '1';
642 pump_state <= idle;
643 else
644     count <= count +1;
645     fifo_READ_l <= '1';
646     if (fifo_empty_l = '0') then
647        pump_state <= fifo_wait_on_empty;
648     else
649        pump_state <= fifo_strobe;
650     end if;
651 end if;
652 end if;
653 --          data_pump_word_ready <= '0';
654 --          fiber_to_ibus_buf(15 downto 8) <= fifo_out(7 downto 0);
655 --          fiber_to_ibus_buf(7 downto 0) <=
656 --                                  fiber_to_ibus_buf(15 downto 8);         --
657 --          -- We use the control indicator to delimit word boundaries,
658 --          -- end of transfer occurs when we see the control bits,
659 --          -- no matter how many bytes are used.
660 --          --
661 --          if (fifo_out(8) = '1') then
662 --          pump_state <= idle;
663 --          else
664 --              if (fifo_empty_l = '0') then
665 --              pump_state <= fifo_wait_on_empty;
666 --              else
667 --              pump_state <= fifo_strobe;
668 --              end if;
669 --          end if;
670         when others =>
671             pump_state <= idle;
672         end case;
673
674     end if;
675     end process pump;
676 end arch_fifo_data_pump;
677
678
679
680 library ieee;
681 use ieee.std_logic_1164.all;
682 use work.numeric_std.all;
683
684 entity tout is
685   port (
686 --      led1 : out std_logic;
687 --       led2 : out std_logic;
688 --       led3 : out std_logic;
689       clk : in std_logic;
690       fast : in std_logic;
691       slow : in std_logic;
692       DEBUG : buffer std_logic;
693       --
694       -- on board data bus and associated control signals
695       --
696       id: inout std_logic_vector(31 downto 0);
697       --
698       -- global
699       --
700       reset: in std_logic;
701       AO_FROM_PC_STROBE: buffer std_logic;
702       AO_FROM_PC_ACK: in std_logic;
703       AO_TO_PC_STROBE: in std_logic;
704       AO_TO_PC_ACK: buffer std_logic;
705       in_strobe: in std_logic;
706       -- fiber reciever
707       --
708       fr_d: in std_logic_vector(11 downto 0);
709       fr_ref_clk: out std_logic;
710       fr_rf:      buffer std_logic;
711       fr_mode:    out std_logic;
712       fr_status:  in std_logic;
713       fr_RDY_l: in std_logic;
714       fr_ckr:     in std_logic;
715       --
716       -- fiber output section
717       --
718       fo_d: out std_logic_vector(9 downto 0);
719       fo_ENN_l: out std_logic;
720       fo_ENA_l: out std_logic;
721       fo_CKW:   buffer std_logic;
722       fo_mode:    out std_logic;
723       fo_foto:    out std_logic;
724       fo_RP_l:  in  std_logic;
725       --
726       -- fifo stuff
727       --
728       fifo_reset_l: buffer std_logic;   -- Declare as a buffer since we want
729       fifo_WRITE_l: out std_logic;
730       fifo_D:    out  std_logic_vector(8 downto 0);
731       fifo_READ_l:  out std_logic;
732       fifo_FULL_l:  in std_logic;
733       fifo_HALF_l:  in std_logic;
734       fifo_EMPTY_l: in std_logic;
735       fifo_OUT:   in std_logic_vector(8 downto 0)
736     );
737 end tout;
738
```

```
739
740 -- purpose: Collect together preceeding modules
741 architecture tout_arch of tout is
742
743     signal write_oe : std_logic;
744     signal read_oe : std_logic;
745     signal fo_data_strobe : std_logic;
746     signal violation_count : unsigned(7 downto 0);
747     signal decriment_fifo_count : std_logic;
748     signal increment_fifo_count : std_logic;
749     signal data_pump_word_ready:    std_logic;
750     signal refill_ibus_output_buf: std_logic;
751     signal fiber_to_ibus_buf : std_logic_vector(15 downto 0);
752     signal reset_fifo : std_logic;
753     signal rec_enabled : std_logic;
754
755     type tristate_state_type is (idle,active);
756 signal tristate_state : tristate_state_type;
757 signal il_tristate : std_logic;
758 signal il_ack: std_logic;
759 signal i_am_remote : std_logic;        -- set to 1 for remote board 0 for local
760 signal my_data_address : std_logic_vector(3 downto 0);
761 signal my_ctrl_address : std_logic_vector(3 downto 0);
762 signal from_FI_ack : std_logic;
763 signal from_FI_req : std_logic;
764 signal data_from_pc : std_logic_vector(15 downto 0);
765 signal address_from_pc : std_logic_vector(7 downto 0);
766 signal power_up_reset: std_logic;
767 signal soft_request_reset: std_logic;
768 signal data_to_FO : std_logic_vector(15 downto 0);
769 signal this_chip_selected: std_logic;
770 signal FI_address : std_logic_vector(7 downto 0);
771 signal FI_to_ibus_req : std_logic;
772 signal FI_to_ibus_ack : std_logic;
773 signal FI_data: std_logic_vector(15 downto 0);
774 signal write_to_FO_req : std_logic;
775 signal write_to_FO_ack : std_logic;
776 -- alias reset_fifo : std_logic is request_reset;
777
778 begin
779     --    reset <= '1';            -- negative logic
780     DEBUG <= AO_FROM_PC_STROBE;
781     i_am_remote <= '1';
782     my_data_address <= "1111";
783     my_ctrl_address <= "0010";
784   --
785     -- Fiber output section
786     --
787     outstrobe : process (clk,reset)
788     begin  -- process outstrobe
789     -- activities triggered by asynchronous reset (active low)
790     if clk'event and clk = '1' then
791         fo_data_strobe <= not fo_data_strobe;
792     end if;
793     end process outstrobe;
794     fo_CKW <= fo_data_strobe;
795     fo_mode <= '0';
796     fo_foto <= '0';
797     fo_ENN_l <= '1';
798   -- purpose: Takes care of tristating the internal lower bus ack line which is
799   -- multi-driven by several chips. This is a clocked process because we wish to
800   -- make sure that the ack signal is set to zero before the output is tristated.
801   -- In this way the output holding latch always holds the line low in the
802   -- absence of any logic drive.
803   -- type   : sequential
804   -- inputs : clk, reset,
805 --  bus_strobe : process (clk, reset)
806 --  begin  -- process
807 --    if reset = '1' then                -- asynchronous reset (active low)
808 --            fo_ENA_l <= '1';
809 --   elsif clk'event and clk = '1' then  -- rising clock edge
810 --     case (tristate_state) is
811 --       when idle =>
812 --    il_tristate <= '1';
813 --          if (this_chip_selected = '1') then
814 --           il_tristate <= '0';
815 --           tristate_state <= active;
816 --          end if;
817 --        when active =>
818 --          if (this_chip_selected = '0') then
819 --            il_tristate <= '0';      -- Lower output before tristating
820 --            tristate_state <= idle;
821 --          end if;
822 --        when others =>
823 --          tristate_state <= idle;
824 --      end case;
825 --    end if;
826 --  end process bus_strobe;
827 --  AO_FROM_PC_ACK <= il_ack when (il_tristate = '0') else 'Z';
828  --
829   -- deal with signal routing depending on setting of loopback bit
830   --
831   --from_pc_ack <= to_pc_ack when (loopback_state = '1') else write_to_FO_ack;
832   --to_pc_req <= from_pc_req when (loopback_state = '1') else FI_to_ibus_req;
833   --data_to_pc <= data_from_FI when (loopback_state = '1') else FO_data;
834   --write_to_FO_req <= from_pc_req when (loopback_state = '0') else '0';
835   --FI_to_ibus_ack <= to_pc_ack when (loopback_state = '0') else '0';
836 -- direction:  BP data read to ibus_upper >FO
837    ibus_reader : read_from_ibus port map (
838      clk       => clk,
839      reset     => reset,
840      iu_req    => AO_TO_PC_STROBE,  --  in
841      iu_ack    => AO_TO_PC_ACK,     --  out
842      iu_io     => id(31 downto 16),
843      data      => data_to_FO,
844      req       => write_to_FO_req,      -- out
845      ack       => write_to_FO_ack       -- in
846      );
847     -- direction: received FI=PC writes output to ibus_lower
848   FI_reader : read_from_FI port map (
849     clk             => clk,
850     reset           => reset,
851     req             => FI_to_ibus_req,  -- in
852     ack             => FI_to_ibus_ack,  -- out
853     data            => FI_data,
854     il_io           => id(15 downto 0),
855     address         => address_from_pc,
856     il_req          => AO_FROM_PC_STROBE,   -- out
857     il_ack          => AO_FROM_PC_ACK,      -- in
858     i_am_remote     => i_am_remote,
859     my_ctrl_address => my_ctrl_address,
860     my_data_address => my_data_address,
861     request_reset   => reset_fifo,
```

```
862     FI_i_am_addressed => this_chip_selected,
863     receive_enabled   => rec_enabled
864     );
865
866     --
867     ibus_FO: ibus_FO_action port map (
868         clk             => clk,
869         reset           => reset,
870         il_i_am_addressed => this_chip_selected,
871         fo_req          => write_to_FO_req, -- in
872         fo_ack          => write_to_FO_ack, -- out
873         data            => data_to_FO,
874         fiber_clk       => fo_CKW,
875         fo_d            => fo_d,
876         fo_ENA_l        => fo_ENA_l
877         );
878     --
879     -- fiber receiver stuff
880     --
881     fr_ref_clk <= fo_data_strobe;
882     fr_mode <= '0';
883     fr_rf <= '1';
884     ibus_FI: ibus_FI_port port map (
885         clk             => clk,
886         reset           => reset,
887         data            => FI_data,
888         req             => FI_to_ibus_req,  -- out
889         ack             => FI_to_ibus_ack,  -- in
890         fiber_to_ibus_buf       => fiber_to_ibus_buf,
891         data_pump_word_ready    => data_pump_word_ready,
892         refill_ibus_output_buf  => refill_ibus_output_buf
893         );
894
895     fr_imp: fiber_rec port map (
896         clk             => clk,
897         fr_d            => fr_d,
898         fr_RDY_l        => fr_RDY_l,
899         fr_status       => fr_status,
900         receive_enabled => rec_enabled,
901         increment_fifo_count    => increment_fifo_count,
902         violation_count         => violation_count,
903         fifo_reset_l            => fifo_reset_l,
904         fifo_D          => fifo_D,
905         fifo_WRITE_l    =>  fifo_WRITE_l,
906         reset_fifo_request      => reset_fifo
907                 );
908
909     pump: fifo_data_pump port map (
910         clk             => clk,
911         reset           => reset,
912         violation_count =>  violation_count,
913         fiber_to_ibus_buf =>  fiber_to_ibus_buf,
914         fifo_OUT        => fifo_OUT,
915         fifo_READ_l     => fifo_READ_l,
916         data_pump_word_ready => data_pump_word_ready,
917         refill_ibus_output_buf => refill_ibus_output_buf,
918         fifo_reset_l    =>  fifo_reset_l,
919         fifo_EMPTY_l => fifo_EMPTY_l
920                 );
921     --
922     end  tout_arch;
923
924
925
926
927
928
929
930
931
932
933
934
935
936
```