```
  1
  2 --Signal naming conventions
  3 -- ON- chip signals are not preficed
  4 -- Upper part of the on-board bus signale preficed with             iu_
  5 -- Lower part of the on-board bus signals preficed with             il_
  6 -- Upper part of pc bus                                            pu_
  7 -- Lower part of pc bus                                            pl_
  8 -- Backplane bus                                                   bp_
  9 -- fiber output                                                    fo_
 10 -- fiber input                                                     fi_
 11
 12
 13
 14 --
 15 -- take care of writing data to the onboard bus
 16 --
 17 library ieee;
 18 use ieee.std_logic_1164.all;
 19 use work.numeric_std.all;
 20
 21 entity write_to_ibus is
 22
 23    port (
 24     clk   : in  std_logic;
 25     reset : in  std_logic;
 26     iu_req   : out std_logic;
 27     iu_ack   : in  std_logic;
 28     iu_io    : out std_logic_vector(15 downto 0);
 29     data  : in  std_logic_vector(15 downto 0);
 30     req  : in  std_logic;
 31     ack  : out std_logic);
 32
 33 end write_to_ibus;
 34
 35 architecture write_to_ibus_arch of write_to_ibus is
 36 type iu_write_state is (idle,hs,iu_hs1,iu_hs2);
 37 signal iu_write : iu_write_state;
 38 signal timeout : unsigned(3 downto 0);
 39 begin  -- write_to_ibus_arch
 40
 41   iu_writer: process (clk, reset)
 42   begin  -- process iu_writer
 43     if reset = '1' then                 -- asynchronous reset (active low)
 44
 45     elsif clk'event and clk = '1' then  -- rising clock edge
 46       ack <= '0';
 47       iu_req <= '0';
 48       case iu_write is
 49         when idle =>
 50           if (req = '1') then
 51             iu_io <= data;
 52             ack <= '1';
 53             iu_write <= hs;
 54             timeout <= "0000";
 55           end if;
 56         when hs =>
 57           ack <= '1';
 58           iu_req <= '1';
 59           if (req = '0') then
 60             iu_write <= iu_hs1;
 61             timeout <= "0000";
 62           else
 63             if (timeout = "1111") then
 64               iu_write <= iu_hs1;
 65               timeout <= "0000";
 66             else
 67               timeout <= timeout+1;
 68             end if;
 69           end if;
 70         when iu_hs1 =>
 71           iu_req <= '1';
 72           if (iu_ack = '1') then
 73             iu_req <= '0';
 74             iu_write <= iu_hs2;
 75           else
 76             if (timeout = "1111") then
 77               iu_write <= 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 (iu_ack = '0') then
 86             iu_write <= idle;
 87           end if;
 88         when others =>
 89           iu_write <= idle;
 90       end case;
 91     end if;
 92   end process iu_writer;
 93 end write_to_ibus_arch;
 94
 95 library ieee;
 96 use ieee.std_logic_1164.all;
 97 use work.numeric_std.all;
 98
 99 --
100 -- This block reads data from the on board bus when addressed. It captures and
101 -- acts on command data and places other data or address info in the on
102 -- chip buffers.
103 -- Each transaction typically consists of two transfers, an address transfer
104 -- followed by a data transfer. There are a few exceptions where only an
105 -- address transfer is required, namelly a soft reset and switching to
106 -- loopback mode.
107 -- When the code detects an address transfer it check to see if it is being
108 -- addressed. If it is then enables itself. Thie chip stays enabled untill
109 -- another address transfer occurs. In this way the chip can accept follow
110 -- up data transfers.
111 --
112
113 entity read_from_ibus is
114
115   port (
116     clk       : in     std_logic;
117     reset     : in     std_logic;
118     il_req    : in     std_logic;
119     il_ack    : out    std_logic;
120     il_io     : in     std_logic_vector(15 downto 0);
121     data      : buffer std_logic_vector(15 downto 0);
122     address:buffer std_logic_vector(7 downto 0);
123     req       : out    std_logic;
```

```
124     ack       : in     std_logic;
125     i_am_remote:in     std_logic;
126     my_data_address:in    std_logic_vector(3 downto 0);
127     my_ctrl_address:in    std_logic_vector(3 downto 0);
128     loopback_state:buffer std_logic;
129     request_reset:out     std_logic;
130     counter_start:out     unsigned(7 downto 0);
131     counter_end:out       unsigned(7 downto 0);
132     il_i_am_addressed: buffer std_logic;
133     dip_switch:in         std_logic_vector(7 downto 0));
134
135 end read_from_ibus;
136
137 architecture read_from_ibus_arch of read_from_ibus is
138 type il_read_type is (idle,am_i_addressed,il_hs_for_address,il_hs_for_data,
139                       il_use_ctrl_data,internal_ack1,internal_ack2);
140 signal il_read_state : il_read_type;
141 signal timeout : unsigned(3 downto 0);
142 --signal il_i_am_addressed : std_logic;
143 signal this_is_a_ctrl_transaction : std_logic;
144 begin  -- read_from_ibus_arch
145
146   -- purpose: read from the on-board bus
147   -- type   : sequential
148   -- inputs : clk, reset
149   -- outputs:
150   read_ibus: process (clk, reset)
151   begin  -- process read_ibus
152     if reset = '1' then                 -- asynchronous reset (active low)
153       il_i_am_addressed <= '0';
154       this_is_a_ctrl_transaction <= '0';
155       loopback_state <= '0';
156       -- by default load counter
157       counter_start <= "11111111";
158       counter_end <= "11111111";
159       for i in 7 downto 0 loop
160         --counter_end(i) <= dip_switch(i);
161       end loop;  -- i
162     elsif clk'event and clk = '1' then  -- rising clock edge
163       il_ack <= '0';
164       req <= '0';
165       request_reset <= '0';
166       case (il_read_state) is
167         when idle =>
168 loopback_state <= '0';
169 -- disable loopback until things are working
170           if (il_req = '1') then
171             data <= il_io;
172             il_read_state <= am_i_addressed;
173             timeout <= "0000";
174           end if;
175         when am_i_addressed =>
176           il_read_state <= idle;
177           if (data(15) = '1') then -- this is an address
178             il_i_am_addressed <= '0';   -- any addtress transaction should
179                                         -- reset select bits on all chips
180             this_is_a_ctrl_transaction <= '0';
181             if (data(12) = i_am_remote) then
182               if (data(11 downto 8) = my_ctrl_address) then
183                 this_is_a_ctrl_transaction <= '1';
184                 il_i_am_addressed <= '1';
185 --                loopback_state <= data(4);
186 -- disable loopback until things are working
187                 request_reset <= data(5);
188                 address <= data(7 downto 0);
189                 il_read_state <= il_hs_for_address;
190               end if;
191               if (data(11 downto 8) = my_data_address) then
192                 il_i_am_addressed <= '1';
193                 address <= data(7 downto 0);
194                 il_read_state <= il_hs_for_address;
195               end if;
196             end if;
197           else
198             --
199             -- This is not an address, but if this chip is already addressed
200             -- then this will be a data transaction to complete the previous
201             -- address transaction
202             --
203             if (il_i_am_addressed = '1') then
204               il_read_state <= il_hs_for_data;
205             else
206               il_read_state <= idle;
207             end if;
208           end if;
209         when il_hs_for_address =>
210           il_ack <= '1';
211           if (il_req = '0') then
212             --
213             -- Only do an ack cycle if we are in loopback mode
214             --
215             if (loopback_state = '1') then
216               il_read_state <= internal_ack1;
217             else
218               il_read_state <= idle;
219             end if;
220           else
221             if (timeout = "1111") then
222               -- Hmm this timeout will lead to a spurious transaction, but not
223               -- to worry, there must be a serious problem somewhere else in
224               -- this case
225               il_read_state <= idle;
226             else
227               timeout <= timeout+1;
228             end if;
229           end if;
230         when il_hs_for_data =>
231           il_ack <= '1';
232           if (il_req = '0') then
233             --
234             -- things to do once the external transaction has completed
235             --
236             if (this_is_a_ctrl_transaction = '1') then
237               il_read_state <= il_use_ctrl_data;
238             else
239               il_read_state <= internal_ack1;
240               timeout <= "0000";
241             end if;
242           else
243             if (timeout = "1111")  then
244               --
245               -- If this timeout is triggered we are likely to keep looping.
246               -- By gong to an idle state we ensure that only 1 data transaction
```

```
247               -- can take place anyway, thought there is a possibility of
248               -- losing a transaction as well.
249               --
250               il_read_state <= idle;
251             else
252               timeout <= timeout+1;
253             end if;
254           end if;
255           if (this_is_a_ctrl_transaction = '0') then
256             --
257             -- Get a jump start on completing internal transactions
258             --
259             req <= '1';  -- attempt to transfer data out
260           end if;
261         when il_use_ctrl_data =>
262           --
263           -- What we do with the control data is completely dependent on the
264           -- chip all of the rest of this code should be fairly generic, if
265           -- a littl over elaborate in most cases
266           --
267           if (address(1 downto 0) = "01") then
268             c1: for i in 0 to 7 loop
269               counter_start(i) <= data(i);
270               counter_end(i) <= data(i+8);
271             end loop c1;
272           end if;
273           -- Only do an ack if this is a loopback
274       if (loopback_state = '1') then
275             il_read_state <= internal_ack1;
276           else
277             il_read_state <= idle;
278           end if;
279         when internal_ack1 =>
280           req <= '1';
281           if (ack = '1') then
282             timeout <= "0000";
283             req <= '0';
284             il_read_state <= internal_ack2;
285           else
286             if (timeout = "1111") then
287               il_read_state <= idle;
288             else
289               timeout <= timeout+1;
290             end if;
291           end if;
292         when internal_ack2 =>
293           req <= '0';
294           if (ack = '0') then
295             il_read_state <= idle;
296           else
297             if (timeout = "1111") then
298               il_read_state <= idle;
299             else
300               timeout <= timeout+1;
301             end if;
302           end if;
303         when others =>
304           il_read_state <= idle;
305       end case;
306     end if;
307   end process read_ibus;
308 end read_from_ibus_arch;
309
310 library ieee;
311 use ieee.std_logic_1164.all;
312 use work.numeric_std.all;
313
314 entity backplane_io is
315
316   port (
317     clk               : in  std_logic;
318     reset             : in  std_logic;
319     --broken_hva          : in  std_logic;
320     bp_ctrl_bus       : in  std_logic_vector(15 downto 0);
321     write_to_bp_addr  : in  std_logic_vector(7 downto 0);
322     write_to_bp_req   : in  std_logic;
323     write_to_bp_ack   : out std_logic;
324     bp_strobe         : buffer std_logic;
325     bp_data           : in  std_logic_vector(15 downto 0);
326     read_from_bp_addr : in  std_logic_vector(7 downto 0);
327     read_from_bp_data : out std_logic_vector(15 downto 0);
328     read_from_bp_req  : in  std_logic;
329     read_from_bp_ack  : out std_logic;
330     bp_addr_tristate  : out std_logic;
331     bp_addr           : out std_logic_vector(7 downto 0);
332     bp_data_tristate  : out std_logic;
333     bp_data_direction_is_write: buffer std_logic;
334     led1              : out std_logic);
335
336 end backplane_io;
337
338 architecture backplane_io of backplane_io is
339   type bp_access_type is (idle,
340                           write_wait,write_settle,write_strobe,write_ack,
341                           read_from_bp,read_strobe,read_ack);
342   signal bp_access_state : bp_access_type;
343   signal timeout : unsigned(2 downto 0);
344   signal propagation_delay : unsigned(2 downto 0);
345 begin  -- backplane_io
346
347   -- purpose: Read and write data to the backplane. Since data is multiplexed,
348   -- driving backplane through 1 process ensures no conflict can occur.
349   -- type   : sequential
350   -- inputs : clk, reset
351   -- outputs:
352   --
353   propagation_delay <= "010";           -- about 80ns
354   bp_access: process (clk, reset)
355   begin  -- process bp_access
356     if reset = '1' then                 -- asynchronous reset (active low)
357
358     elsif clk'event and clk = '1' then  -- rising clock edge
359       write_to_bp_ack <= '0';
360       read_from_bp_ack <= '0';
361       bp_data_direction_is_write <= '0';  -- default to zero since this implies
362                                           -- lower power dissipation
363       bp_data_tristate <= '1';
364       bp_strobe <= '0';
365       bp_addr_tristate <= '1';
366       led1 <= '1';
367       case (bp_access_state) is
368         when idle =>
369           timeout <= (others => '0');
```

```
370           if (write_to_bp_req = '1') then
371             bp_addr <= write_to_bp_addr;
372             -- let outputs settle before enabling drivers
373             --bp_data_tristate <= '0';
374             bp_data_direction_is_write <= '1';
375             bp_access_state <= write_wait;
376             timeout <= (others => '0');
377             led1 <= '0';
378             --bp_addr_tristate <= '0';
379           else
380             if (read_from_bp_req = '1') then
381               bp_addr <= read_from_bp_addr;
382               bp_data_tristate <= '0';
383               bp_access_state <= read_from_bp;
384               --bp_addr_tristate <= '0';
385             end if;
386           end if;
387         when write_wait =>
388           --
389           -- This waitstate is needed to give the pld time to activate its output
390           -- drive (which is done via the bp_data_direction_flag
391           --
392           bp_data_direction_is_write <= '1';
393           bp_addr_tristate <= '0';
394           bp_access_state <= write_settle;
395         when write_settle =>
396            led1 <= '0';
397            bp_data_direction_is_write <= '1';
398            bp_data_tristate <= '0';
399            bp_addr_tristate <= '0';
400            if (timeout = propagation_delay) then
401              bp_access_state <= write_strobe;
402              timeout <= "000";
403            else
404              timeout <= timeout+1;
405            end if;
406         when write_strobe =>
407           led1 <= '0';
408           bp_data_direction_is_write <= '1';
409           bp_data_tristate <= '0';
410           bp_addr_tristate <= '0';
411           bp_strobe <= '1';
412           if (timeout = propagation_delay)  then
413             bp_access_state <= write_ack;
414             write_to_bp_ack <= '1';
415             timeout <= (others => '0');
416           else
417             timeout <= timeout+1;
418           end if;
419         when write_ack =>
420           led1 <= '0';
421           write_to_bp_ack <= '1';
422           if (write_to_bp_req = '0') then
423             bp_access_state <= idle;
424           else
425             if (timeout = "111") then
426               bp_access_state <= idle;
427             else
428               timeout <= timeout+1;
429             end if;
430           end if;
431         when read_from_bp =>
432           bp_data_tristate <= '0';
433           bp_addr_tristate <= '0';
434           if (timeout = propagation_delay) then
435             bp_access_state <= read_strobe;
436             timeout <= timeout+1;
437           else
438             timeout <= timeout+1;
439           end if;
440         when read_strobe =>
441           bp_data_tristate <= '0';
442           bp_addr_tristate <= '0';
443           bp_strobe <= '1';
444           if (timeout = propagation_delay)  then
445             read_from_bp_data(14 downto 0) <= bp_data(14 downto 0);
446             bp_access_state <= read_ack;
447             read_from_bp_ack <= '1';
448           else
449             timeout <= timeout+1;
450           end if;
451         when read_ack =>
452            read_from_bp_ack <= '1';
453            if (read_from_bp_req = '0') then
454               bp_access_state <= idle;
455            else
456              if (timeout = "111") then
457                bp_access_state <= idle;
458              else
459                timeout <= timeout+1;
460              end if;
461            end if;
462         when others => bp_access_state <= idle;
463       end case;
464
465     end if;
466   end process bp_access;
467
468 end backplane_io;
469
470
471 library ieee;
472 use ieee.std_logic_1164.all;
473 use work.numeric_std.all;
474
475 --
476 -- Read the counter values on the backplane, taking into account that the
477 -- counters are gruped in 24's so every 4 the 8 byte chunk is missing.
478 --
479
480 entity read_counters is
481   port (
482     clk       : in  std_logic;
483     reset     : in  std_logic;
484     phase     : in  std_logic;
485     data      : buffer std_logic_vector(15 downto 0);
486     req       : out std_logic;
487     ack       : in  std_logic;
488     bp_address    : out std_logic_vector(7 downto 0);
489     bp_req   : out std_logic;
490     bp_ack : in  std_logic;
491     start_address:in unsigned(7 downto 0);
492     end_address:in unsigned(7 downto 0);
```

```
493     reset_counters:buffer std_logic);
494 end read_counters;
495 
496 architecture read_counters_arch  of read_counters is
497 type counter_read_type is (idle,valid_address,increment_address,test_end,
498                            start_fetch,complete_fetch,
499                            transfer_to_pc,complete_pc_transfer,
500                            do_reset_counters);
501 signal counter_read_state : counter_read_type;
502 signal old_phase : std_logic;
503 signal counter_address : unsigned(7 downto 0);
504 signal reset_count : unsigned(2 downto 0);
505 begin  -- read_counters_arch
506 
507   -- purpose: Take care of reading counter data from the backplane
508   -- type   : sequential
509   -- inputs : clk, reset
510   -- outputs:
511   process (clk, reset)
512   begin  -- process
513     if reset = '1' then                 -- asynchronous reset (active low)
514       old_phase <= phase;
515     elsif clk'event and clk = '1' then  -- rising clock edge
516       bp_req <= '0';
517       req <= '0';
518       old_phase <= phase;
519       reset_counters <= '0';
520       reset_count <= (others => '0');
521       case (counter_read_state) is
522         when idle =>
523           if ((old_phase = not phase)) then
524             --
525             -- Bus read triggered by change of phase.
526             --
527             if (start_address = "11111111") then
528               --
529               -- Special case (occurs after reset) which inhibits data read.
530               --
531             else
532               counter_address <= start_address;
533               counter_read_state <= valid_address;
534             end if;
535           end if;
536         when valid_address =>
537           if (counter_address(4 downto 3) = "11") then
538             -- This is not a valid address (only 24 counters per card)
539             counter_read_state <= test_end;
540           else
541             counter_read_state <= start_fetch;
542             for i in 0 to 7 loop
543               bp_address(i) <= counter_address(i);
544             end loop;  -- i
545           end if;
546         when start_fetch =>
547           bp_req <= '1';
548           if (bp_ack = '1') then
549             bp_req <= '0';
550             counter_read_state <= complete_fetch;
551             --
552             -- Set most significant bit to phase
553             --
554             data(15) <= not phase;
555           end if;
556         when complete_fetch =>
557           req <= '1';
558           if (bp_ack = '0') then
559             counter_read_state <= transfer_to_pc;
560           end if;
561         when transfer_to_pc =>
562           req <= '1';
563           if (ack = '1') then
564             req <= '0';
565             counter_read_state <= complete_pc_transfer;
566           end if;
567         when complete_pc_transfer =>
568           if (ack = '0') then
569             counter_read_state <= test_end;
570           end if;
571         when test_end =>
572           if (counter_address = end_address) then
573             counter_read_state <= do_reset_counters;
574           else
575             counter_read_state <= increment_address;
576           end if;
577         when increment_address =>
578           counter_address <= counter_address+1;
579           counter_read_state <= valid_address;
580         when do_reset_counters =>
581           reset_counters <= '1';
582           if (reset_count = "111") then
583             counter_read_state <= idle;
584           else
585             reset_count <= reset_count+1;
586           end if;
587         when others =>
588           counter_read_state <= idle;
589       end case;
590     end if;
591   end process;
592 
593 end read_counters_arch ;
594 
595 
596 library ieee;
597 use ieee.std_logic_1164.all;
598 
599 entity power_up is
600   port (clk: in std_logic;
601       fast : in std_logic;
602       slow : in std_logic;
603       led2: out std_logic;
604       led3: out std_logic;
605       reset: buffer std_logic
606         );
607 end power_up;
608 --
609 -- purpose: Handle graceful power-up, uses two RC filters on with
610 -- A short time constant, one with a long time constant. May replace
611 -- long time constant RC with counter driven by external oscillator.
612 --
613 -- The main requirement is that the A/D reference buffers be shutdown
614 -- until the negative power rail has had a chance to stabilise.
615 --
```

```
616 architecture power_up_arch of power_up is
617
618 begin  -- power_up_arch
619   led2 <= fast;
620   led3 <= slow;
621 --
622 -- Hold reset high during initial power up
623 --
624   with (fast and (not slow)) select
625     reset <= '1' when '1',
626     '0' when '0',
627     '0' when others;
628 end power_up_arch;
629 --
630 -- end of power up section
631 --
632
633 library ieee;
634 use ieee.std_logic_1164.all;
635 use work.numeric_std.all;
636 --
637 -- looks afte actual transactions on the data bus.
638 --
639
640 entity tout is
641   port (led1 : out std_logic;
642       led2 : out std_logic;
643       led3 : out std_logic;
644       clk : in std_logic;
645       fast : in std_logic;
646       slow : in std_logic;
647       --
648       -- on board data bus and associated control signals
649       --
650       d: inout std_logic_vector(31 downto 0);
651       --
652       -- global
653       --
654       reset: buffer std_logic;
655       -- Auxiliary interface
656       --aux_bus_request: in std_logic;
657       --aux_bus_grant: out std_logic;
658       AO_FROM_PC_STROBE: buffer std_logic;
659       AO_FROM_PC_ACK: buffer std_logic;
660       AO_TO_PC_STROBE: out std_logic;
661       AO_TO_PC_ACK: in std_logic;
662       in_strobe: in std_logic;
663       --periferal_ack: in std_logic;
664       --
665       -- backplane data bus and associated control signale
666
667       bpaddr: out std_logic_vector(7 downto 0);
668       bpctrl: buffer std_logic_vector(15 downto 0);
669       bpd: buffer std_logic_vector(15 downto 0);
670       bpd_h_dir: out std_logic;
671       bpd_h_tristate: out std_logic;
672       bpd_l_dir: out std_logic;
673       bpd_l_tristate: out std_logic;
674       bpaddr_dir: out std_logic;
675       bpaddr_tristate: out std_logic;
676       bpctrl_h_dir: out std_logic;
677       bpctrl_l_dir: out std_logic;
678       bpctrl_h_tristate: out std_logic;
679       bpctrl_l_tristate: out std_logic;
680       sw: in std_logic_vector(7 downto 0);
681       sinphase_zeros : in std_logic
682     );
683 end tout;
684
685
686 -- purpose: Collect together preceeding modules
687 architecture tout_arch of tout is
688 type tristate_state_type is (idle,active);
689 signal tristate_state : tristate_state_type;
690 signal bp_start_address : unsigned(7 downto 0);  -- first counter address
691 signal bp_end_address : unsigned(7 downto 0);  -- last counter address
692 signal il_tristate : std_logic;
693 signal il_ack: std_logic;
694 signal i_am_remote : std_logic;        -- set to 1 for remote board 0 for local
695 signal my_data_address : std_logic_vector(3 downto 0);
696 signal my_ctrl_address : std_logic_vector(3 downto 0);
697 signal from_pc_ack : std_logic;
698 signal from_pc_req : std_logic;
699 signal data_from_pc : std_logic_vector(15 downto 0);
700 signal address_from_pc : std_logic_vector(7 downto 0);
701 signal power_up_reset: std_logic;
702 signal loopback_state: std_logic;
703 signal soft_request_reset: std_logic;
704 signal data_to_pc : std_logic_vector(15 downto 0);
705 signal to_pc_ack : std_logic;
706 signal to_pc_req : std_logic;
707 signal this_chip_selected: std_logic;
708 signal bp_data_tristate : std_logic;
709 signal bp_addr_tristate: std_logic;
710 signal sensor_address : std_logic_vector(7 downto 0);
711 signal sensor_to_bp_read_req : std_logic;
712 signal sensor_to_bp_read_ack : std_logic;
713 signal sensor_to_ibus_req : std_logic;
714 signal sensor_to_ibus_ack : std_logic;
715 signal sensor_data: std_logic_vector(15 downto 0);
716 signal write_to_bp_req: std_logic;
717 signal write_to_bp_ack : std_logic;
718 -- Note this is a dummy variable to ifdef the "broken HVA" in
719 signal broken_hva_config : std_logic;
720
721
722 alias bp_direction_is_write : std_logic is bpctrl(0);
723 alias bp_phase : std_logic is bpctrl(1);
724 alias bp_strobe : std_logic is bpctrl(2);
725 alias bp_reset_counters : std_logic is bpctrl(3);
726 --
727 -- use this alias when dealing with the newer HVA board
728 --
729 -- alias backplane_data : std_logic_vector(15 downto 0) is bpd;
730 --
731 -- use this alias when dealing with the broken HVA board.
732 --
733
734
735
736 --
737 -- Following mods are needed when using the broken High voltege amplifier boards
738 --
```

```
739
740
741 begin
742   --
743   -- Following mods are needed for broken HVA board.
744   --
745   broken_hva_config <= '0'; -- for newer board
746   -- broken_hva_config <= '1'; -- for older board
747   --
748   -- Static definitions
749   --
750   i_am_remote <= '1';
751   my_data_address <= "0000";
752   my_ctrl_address <= "0001";
753   --
754   -- Use high control byte for outputs to attached periferals.
755   -- Note that this is the only driver that must be permenantly
756   -- switched on, since we cant let the strobe signals drift
757   -- around.
758   --
759   -- temporary initialisations that will be subsumed later
760   --
761   --
762   -- permenant settings
763   --
764   bpaddr_dir <= '0';            -- Permenantly set to output
765   --bpaddr_tristate <= '0';
766   bpctrl_l_dir <= '0';                  -- Permenently set to output
767   bpctrl_l_tristate <= '0';
768   pup:    power_up port map (clk,fast,slow,led2,led3,power_up_reset);
769   reset <= power_up_reset;
770
771   -- purpose: Takes care of tristating the internal lower bus ack line which is
772   -- multi-driven by several chips. This is a clocked process because we wish to
773   -- make sure that the ack signal is set to zero before the output is tristated.
774   -- In this way the output holding latch always holds the line low in the
775   -- absence of any logic drive.
776   -- type   : sequential
777   -- inputs : clk, reset,
778   process (clk, reset)
779   begin  -- process
780     if reset = '1' then                 -- asynchronous reset (active low)
781
782     elsif clk'event and clk = '1' then  -- rising clock edge
783       case (tristate_state) is
784         when idle =>
785       il_tristate <= '1';
786           if (this_chip_selected = '1') then
787            il_tristate <= '0';
788            tristate_state <= active;
789           end if;
790         when active =>
791           if (this_chip_selected = '0') then
792             il_tristate <= '0';      -- Lower output before tristating
793             tristate_state <= idle;
794           end if;
795         when others =>
796           tristate_state <= idle;
797       end case;
798     end if;
799   end process;
800   AO_FROM_PC_ACK <= il_ack when (il_tristate = '0') else 'Z';
801    --
802
803   --
804   -- deal with signal routing depending on setting of loopback bit
805   --
806   from_pc_ack <= to_pc_ack when (loopback_state = '1') else write_to_bp_ack;
807   to_pc_req <= from_pc_req when (loopback_state = '1') else sensor_to_ibus_req;
808   data_to_pc <= data_from_pc when (loopback_state = '1') else sensor_data;
809   write_to_bp_req <= from_pc_req when (loopback_state = '0') else '0';
810   sensor_to_ibus_ack <= to_pc_ack when (loopback_state = '0') else '0';
811
812    ibus_writer : write_to_ibus port map (
813      clk       => clk,
814      reset     => reset,
815      iu_req    => AO_TO_PC_STROBE,
816      iu_ack    => AO_TO_PC_ACK,
817      data      => data_to_pc,
818      iu_io     => d(31 downto 16),
819      req       => to_pc_req,
820      ack       => to_pc_ack);
821
822   ibus_reader : read_from_ibus port map (
823     clk             => clk,
824     reset           => reset,
825     il_req          => AO_FROM_PC_STROBE,
826     il_ack          => il_ack,
827     il_io           => d(15 downto 0),
828     data            => data_from_pc,
829     address         => address_from_pc,
830     req             => from_pc_req,
831     ack             => from_pc_ack,
832     i_am_remote     => i_am_remote,
833     my_ctrl_address => my_ctrl_address,
834     my_data_address => my_data_address,
835     loopback_state  => loopback_state,
836     request_reset   => soft_request_reset,
837     counter_start   => bp_start_address,
838     counter_end     => bp_end_address,
839     il_i_am_addressed => this_chip_selected,
840     dip_switch      => sw);
841
842
843 bp_io : backplane_io port map (
844   clk                   => clk,
845   reset                 => reset,
846   --broken_hva      => broken_hva_config,
847   bp_ctrl_bus       => bpctrl,
848   write_to_bp_addr      => address_from_pc,
849   write_to_bp_req       => write_to_bp_req,
850   write_to_bp_ack       => write_to_bp_ack,
851   bp_strobe             => bp_strobe,
852   bp_data_direction_is_write => bp_direction_is_write,
853   bp_data               => bpd, --backplane_data,
854   read_from_bp_addr     => sensor_address,
855   read_from_bp_req      => sensor_to_bp_read_req,
856   read_from_bp_ack      => sensor_to_bp_read_ack,
857   read_from_bp_data     => sensor_data,
858   bp_addr               => bpaddr,
859   bp_data_tristate      => bp_data_tristate,
860   bp_addr_tristate      => bp_addr_tristate,
861   led1                  => led1);
```

```
862
863   bpd_l_dir <= '0' when  (bp_direction_is_write = '1') else '1';
864   bpd_h_dir <= '0' when  (bp_direction_is_write = '1') else '1';
865   bpd_h_tristate <= bp_data_tristate;
866   bpd_l_tristate <= bp_data_tristate;
867   bpaddr_tristate <= bp_addr_tristate;
868   -- take care of dealing with backplane data io
869   --bp_dir_is_write <= bp_direction_is_write;
870   bp_phase <= sinphase_zeros;
871   --
872   -- Use this with the corrected HVA.
873   --
874   --bpd <= data_from_pc when (bp_direction_is_write = '1') else (others => 'Z');
875   --bpctrl_h_tristate <= '1';             -- permenantly disable
876   --
877   -- end of correct HVA code
878   --
879   --
880   -- Use ths with the broken HVA
881   --
882   bpctrl_h_tristate <= bp_data_tristate;
883   bpd(15 downto 10) <=
884     data_from_pc(5 downto 0) when (bp_direction_is_write = '1') else (others =>'Z');
885   bpctrl(8 downto 4) <=
886     data_from_pc(10 downto 6) when (bp_direction_is_write ='1') else (others =>'Z');
887   bpctrl(11 downto 10) <=
888     data_from_pc(12 downto 11) when (bp_direction_is_write ='1') else (others=>'Z');
889   bpctrl(14 downto 13) <=
890     data_from_pc(14 downto 13) when (bp_direction_is_write ='1') else (others=>'Z');
891   --
892   -- end of broken HVA code
893   --
894   --   read_from_bp_data(0) <= bp_data(10);
895   --   read_from_bp_data(1) <= bp_data(11);
896   --   read_from_bp_data(2) <= bp_data(12);
897   --   read_from_bp_data(3) <= bp_data(13);
898   --   read_from_bp_data(4) <= bp_data(14);
899   --  read_from_bp_data(5) <= bp_data(15);
900   --
901   --   read_from_bp_data(6) <= bp_ctrl_bus(4);
902   --   read_from_bp_data(7) <= bp_ctrl_bus(5);
903   --   read_from_bp_data(8) <= bp_ctrl_bus(6);
904   --   read_from_bp_data(9) <= bp_ctrl_bus(7);
905   --   read_from_bp_data(10) <= bp_ctrl_bus(8);
906   --
907   --   read_from_bp_data(11) <= bp_ctrl_bus(10);
908   --   read_from_bp_data(12) <= bp_ctrl_bus(11);
909   --
910   --   read_from_bp_data(13) <= bp_ctrl_bus(13);
911   --   read_from_bp_data(14) <= bp_ctrl_bus(14);
912
913
914
915
916 sp_read : read_counters port map (
917   clk           => clk,
918   reset         => reset,
919   phase         => sinphase_zeros,
920   --phase         => '0',               -- switch off counter reading.
921   data          => sensor_data,
922   req           => sensor_to_ibus_req,
923   ack           => sensor_to_ibus_ack,
924   bp_address    => sensor_address,
925   bp_req        => sensor_to_bp_read_req,
926   bp_ack        => sensor_to_bp_read_ack,
927   start_address => bp_start_address,
928   end_address   => bp_end_address,
929   reset_counters => bp_reset_counters);
930
931 end tout_arch;
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
```