```
--
-- Produce a clock signal for the A0 system A square wave and sine wave are output
-- with a variable phase relatonship.
-- The sine wave is a poor sine wave comprising only 16 samples per cycle
-- or 5 digitisisation values. However after fltering by the membrane mirror
-- and possibly a simple rc filter this should provide an adequate sine wave.
-- A square wave (zeros) tracks the zero crossing of the sine wave and
-- a second square wave (strobe) is offset by a known phase from the sin wave.
-- The operation frequency is set by a 13 bit number (divide) which specifies the
-- number of clocks that should comprise each 1/16 sin output.
-- The phase is set by a second number (phase), 16 bits long, which specifies the
-- phase offset of the (strobe) transition in cycles from the start of each
-- (zeros) transition. Note that the number represinting any given phase offset
-- is proportional to the frequency divider number (divide), so will have to
-- be changed each time the frequency is updated.
-- Note that the true sine wave and the (zeros) and (stobe) output are offser
-- by 1/32 od a period, which should be accounted for by adjusting the phase
-- apropriately.
--
-- The 5 levels of sin output can also be programmed externally. This allows for
-- the output levels to be changed, or even the generation of a different waveform.
--
-- On power up the device shoud be reset by holding the reset pin high and simultaniously
-- strobing the data_strobe line
--

library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

--
-- Generates an arbitrary programmable waveform with an offsettavle
-- phase signal. This code assumes that the DAC is set up in transparent
-- mode so that changes to the DAC output will propagate immediately
-- if we need to write data directly to the DAC then the output from the
-- sinphase generator mus be momentatily disabled.
--
entity sinphase is
  port(clk: in std_logic;
       reset: in std_logic;
       phase1:  in unsigned(12 downto 0);
       phase2:  in unsigned(12 downto 0);
       steplen: in unsigned(9 downto 0);
       val0: in std_logic_vector(7 downto 0);
       val1: in std_logic_vector(7 downto 0);
       val2: in std_logic_vector(7 downto 0);
       val3: in std_logic_vector(7 downto 0);
       val4: in std_logic_vector(7 downto 0);
       val5: in std_logic_vector(7 downto 0);
       val6: in std_logic_vector(7 downto 0);
       val7: in std_logic_vector(7 downto 0);
       predev: in unsigned(3 downto 0);
       sinphase_zeros:out std_logic;
       dac_req: out std_logic;
       dac_ack:in std_logic;
       dac: buffer std_logic_vector(7 downto 0)
     );
end sinphase;

architecture arch_sinphase of sinphase is
type kdac_type is (kd_idle,kd_ack,kd_guard);
signal kdac : kdac_type;
signal current_phase: unsigned(12 downto 0);
signal sample: unsigned(2 downto 0);
signal current_count: unsigned(9 downto 0);
signal kick_dac : std_logic;
signal iclk : std_logic;
signal precount : unsigned(3 downto 0);
begin
  --
  -- run the various counters.
  --
  with sample select
    dac(7 downto 0) <=
    val0 when "000",
    val1 when "001",
    val2 when "010",
    val3 when "011",
    val4 when "100",
    val5 when "101",
    val6 when "110",
    val7 when "111",
    val0 when others;

  -- purpose: predivide the clock for the sinphase doober
  -- type   : sequential
  -- inputs : clk, reset
  -- outputs:
  pre: process (clk, reset, precount)
  begin  -- process pre
    if reset = '1' then                 -- asynchronous reset (active low)
    elsif clk'event and clk = '1' then  -- rising clock edge
      if (precount = predev) then
        iclk <= not iclk;
        precount <= (others => '0');
      else
        precount <= precount+1;
      end if;
    end if;
  end process pre;

  count: process(iclk,reset, steplen, phase1, phase2 ) begin
    if (rising_edge(iclk)) then
      kick_dac <= '0';
      if (current_count = steplen) then
        current_count <= (others => '0');
        kick_dac <= '1';
        if (sample = "111") then
          --
          -- this means we are at the end of the sequence
          --
          sample <= (others => '0');
          current_phase <= (others => '0');
        else
          sample <= sample+1;
          current_phase <= current_phase +1;
        end if;
      else
        if current_phase = phase1 then
          sinphase_zeros <= '1';
        else
          if (current_phase = phase2) then
            sinphase_zeros <= '0';
```

```
124           end if;
125         end if;
126         current_phase <= current_phase +1;
127         current_count <= current_count +1;
128       end if;
129     end if;
130   end process count;
131
132   dacit: process (clk, reset)
133   begin  -- process
134     if reset = '1' then                 -- asynchronous reset (active low)
135
136     elsif clk'event and clk = '1' then  -- rising clock edge
137       dac_req <= '0';
138       case (kdac) is
139         when kd_idle =>
140           if (kick_dac = '1') then
141             dac_req <= '1';
142             kdac <= kd_ack;
143           end if;
144         when kd_ack =>
145           -- Dont bother to check for lowering of ack
146           if (dac_ack = '1') then
147             dac_req <= '0';
148             kdac <= kd_guard;
149           else
150             dac_req <= '1';
151           end if;
152         when kd_guard =>
153           -- Dont return to idle untill the current request goes away.
154           if (kick_dac = '0') then
155             kdac <= kd_idle;
156           end if;
157         when others => kdac <= kd_idle;
158       end case;
159     end if;
160   end process dacit;
161
162 end arch_sinphase;
163
164
165 library ieee;
166 use ieee.std_logic_1164.all;
167 use work.numeric_std.all;
168
169 entity write_to_dac is
170   port (
171     clk : in std_logic;
172     reset : in std_logic;
173     sp_req : in std_logic;
174     sp_ack : out std_logic;
175     pc_req : in std_logic;
176     pc_ack : out std_logic;
177     sp_data : in std_logic_vector(7 downto 0);
178     sp_addr : in std_logic_vector(2 downto 0);
179     pc_data : in std_logic_vector(12 downto 0);
180     pc_addr : in std_logic_vector(3 downto 0);
181     dac_addr : out std_logic_vector(2 downto 0);
182     dac_data : out std_logic_vector(12 downto 0);
183     n_dac_wr : out std_logic;
184     n_dac_clr : out std_logic;
185     n_dac_cs : out std_logic;
186     n_dac_ldab : out std_logic;
187     n_dac_ldcd : out std_logic;
188     n_dac_ldef : out std_logic;
189     n_dac_ldgh : out std_logic;
190     shift : in unsigned(2 downto 0));
191 end write_to_dac;
192
193 architecture write_to_dac_arch of write_to_dac is
194 type dac_type is (idle,pc_ack_st,sp_ack_st,sp_shift,write_dac);
195 signal dac_state : dac_type;
196 signal timeout : unsigned(2 downto 0);
197 -- Have to keep a separate copy of this otherwise we end up reading the address lines
198 -- as inputs, which has the undesirable side-effect of removing the drive!
199 signal dac_addr_copy: std_logic_vector(2 downto 0);
200 signal dac_hold : std_logic_vector(12 downto 0);
201 begin  -- write_to_dac_arch
202
203   n_dac_ldab <= '0' when (dac_addr_copy(2 downto 1) = "00") else '1';
204   n_dac_ldcd <= '0' when (dac_addr_copy(2 downto 1) = "01") else '1';
205   n_dac_ldef <= '0' when (dac_addr_copy(2 downto 1) = "10") else '1';
206   n_dac_ldgh <= '0' when (dac_addr_copy(2 downto 1) = "11") else '1';
207   dac_addr <= dac_addr_copy;
208   -- purpose: clock data to the dac.
209   -- type   : sequential
210   -- inputs : clk, reset
211   -- outputs:
212   dacwr: process (clk, reset)
213   begin  -- process dacwr
214     if reset = '1' then                 -- asynchronous reset
215       n_dac_clr <= '0';
216       n_dac_cs <= '0';
217     elsif clk'event and clk = '1' then  -- rising clock edge
218       n_dac_wr <= '1';
219       n_dac_clr <= '1';
220       n_dac_cs  <= '0';
221       sp_ack <= '0';
222       pc_ack <= '0';
223       case (dac_state) is
224         when idle =>
225           timeout <= (others => '0');
226           --
227           -- PC takes presidence because sinphase may be requesting transferrs
228           -- continuously until we get some rigeisters loaded
229           --
230           if (pc_req = '1') then
231             dac_addr_copy <= pc_addr(2 downto 0);
232             dac_hold <= pc_data;
233             pc_ack <= '1';
234             dac_state <= pc_ack_st;
235           else
236             if (sp_req = '1') then
237               dac_addr_copy <= sp_addr;
238               dac_hold(7 downto 0) <= sp_data;
239               dac_hold(12 downto 8) <= (others => '0');
240               sp_ack <= '1';
241               dac_state <= sp_shift;
242             end if;
243           end if;
244         when sp_shift =>
245           --
246           -- shift the data as indicated by the value of the shift register
```

```
247           -- allows us to use the full dynamic range of the DAC while still
248           -- using only 8 bit registers for the waveform values.
249           --
250           if (timeout = shift) then
251             timeout <= (others => '0');
252             dac_state <= sp_ack_st;
253             --
254             -- Following clause sets the MSB to 1 for all shifts less than
255             -- 5 (i'm assuming that software disallows shofts > 5). This
256             -- keeps the signal closer to zero mean  without doing fancy
257             -- arithmetic. Still need AC coupling to eliminate dc bas though.
258             --
259             if (shift = "101") then
260             else
261               dac_hold(12) <= '1';
262             end if;
263           else
264             timeout <= timeout+1;
265             for i in 11 downto 0 loop
266               dac_hold(i+1) <= dac_hold(i);
267             end loop;  -- i
268             dac_hold(0) <= '0';
269           end if;
270         when sp_ack_st =>
271           sp_ack <= '1';
272           if (sp_req = '0') then
273             dac_state <= write_dac;
274           end if;
275         when pc_ack_st =>
276           dac_data <= dac_hold;
277           pc_ack <= '1';
278           if (pc_req = '0') then
279             dac_state <= write_dac;
280           end if;
281         when write_dac =>
282           dac_data <= dac_hold;
283           n_dac_wr <= '0';
284           if (timeout = "0100") then
285             dac_state <= idle;
286           else
287             timeout <= timeout+1;
288           end if;
289         when others =>
290           dac_state <= idle;
291       end case;
292     end if;
293   end process dacwr;
294 end write_to_dac_arch;
295
296 library ieee;
297 use ieee.std_logic_1164.all;
298 use work.numeric_std.all;
299
300
301 entity read_from_ibus is
302
303   port (
304     clk       : in     std_logic;
305     reset     : in     std_logic;
306     il_req    : in     std_logic;
307     il_ack    : out    std_logic;
308     il_io     : in     std_logic_vector(15 downto 0);
309     data      : buffer std_logic_vector(15 downto 0);
310     address:buffer std_logic_vector(3 downto 0);
311     req       : out    std_logic;
312     ack       : in     std_logic;
313     i_am_remote:in     std_logic;
314     my_data_address:in    std_logic_vector(3 downto 0);
315     my_ctrl_address:in    std_logic_vector(3 downto 0);
316     --request_reset:out     std_logic;
317     il_i_am_addressed: buffer std_logic;
318     phase1     : out unsigned(12 downto 0);  -- Phase to switch sinphase_
319                                              -- strobe from 0 to 1
320     phase2 : out unsigned(12 downto 0);  -- Phase to switch sinphase
321                                          -- strobe from 1 to 0
322     steplen : out unsigned(9 downto 0);
323     val0 : out std_logic_vector(7 downto 0);
324     val1 : out std_logic_vector(7 downto 0);
325     val2 : out std_logic_vector(7 downto 0);
326     val3 : out std_logic_vector(7 downto 0);
327     val4 : out std_logic_vector(7 downto 0);
328     val5 : out std_logic_vector(7 downto 0);
329     val6 : out std_logic_vector(7 downto 0);
330     val7 : out std_logic_vector(7 downto 0);
331     divider: out unsigned(3 downto 0);
332     shift: out unsigned(2 downto 0);
333     digital_out: out std_logic_vector(7 downto 0)
334     );
335
336 end read_from_ibus;
337
338 architecture read_from_ibus_arch of read_from_ibus is
339 type il_read_type is (idle,am_i_addressed,il_hs_for_address,il_hs_for_data,
340                       il_use_ctrl_data,internal_ack1
341            --           ,internal_ack2
342                       );
343 signal il_read_state : il_read_type;
344 signal this_is_a_ctrl_transaction : std_logic;
345 begin  -- read_from_ibus_arch
346
347   -- purpose: read from the on-board bus
348   -- type   : sequential
349   -- inputs : clk, reset
350   -- outputs:
351   read_ibus: process (clk, reset)
352   begin  -- process read_ibus
353     if reset = '1' then                 -- asynchronous reset (active low)
354       il_i_am_addressed <= '0';
355       this_is_a_ctrl_transaction <= '0';
356     elsif clk'event and clk = '1' then  -- rising clock edge
357       il_ack <= '0';
358       req <= '0';
359       --request_reset <= '0';
360       case (il_read_state) is
361         when idle =>
362           if (il_req = '1') then
363             data <= il_io;
364             il_read_state <= am_i_addressed;
365           end if;
366         when am_i_addressed =>
367           il_read_state <= idle;
368           if (data(15) = '1') then -- this is an address
369             il_i_am_addressed <= '0';   -- any addtress transaction should
```

```
                                        -- reset select bits on all chips
            this_is_a_ctrl_transaction <= '0';
            --Dont need to stuff this because this chip is never present on local end
            -- when there is a possibility of forwarding data over the fiber.
            --if (data(12) = i_am_remote) then
              if (data(11 downto 8) = my_ctrl_address) then
                this_is_a_ctrl_transaction <= '1';
                il_i_am_addressed <= '1';
                address <= data(3 downto 0);
                il_read_state <= il_hs_for_address;
              end if;
              if (data(11 downto 8) = my_data_address) then
                il_i_am_addressed <= '1';
                address <= data(3 downto 0);
                il_read_state <= il_hs_for_address;
              end if;
            --end if;
          else
            --
            -- This is not an address, but if this chip is already addressed
            -- then this will be a data transaction to complete the previous
            -- address transaction
            --
            if (il_i_am_addressed = '1') then
              il_read_state <= il_hs_for_data;
            else
              il_read_state <= idle;
            end if;
          end if;
        when il_hs_for_address =>
          il_ack <= '1';
          if (il_req = '0') then
            --
            -- Only do an ack cycle if we are in loopback mode
            --
            il_read_state <= idle;
          end if;
        when il_hs_for_data =>
          il_ack <= '1';
          if (il_req = '0') then
            --
            -- things to do once the external transaction has completed
            --
            if (this_is_a_ctrl_transaction = '1') then
              il_read_state <= il_use_ctrl_data;
            else
              -- Just send the data to the dac output
              il_read_state <= internal_ack1;
            end if;
          end if;
        when il_use_ctrl_data =>
          --
          -- What we do with the control data is completely dependent on the
          -- chip all of the rest of this code should be fairly generic, if
          -- a littl over elaborate in most cases
          --
          il_read_state <= idle;
          case address(3 downto 0) is
            when "0001" =>
              for i in 12 downto 0 loop
                phase1(i) <= data(i);
              end loop;
            when "0010" =>
              for i in 12 downto 0 loop
                phase2(i) <= data(i);
              end loop;
            when "0011" =>
              for i in 9 downto 0 loop
                steplen(i) <= data(i);
              end loop;  -- i
            when "0100" =>
              val0 <= data(7 downto 0);
            when "0101" =>
              val1 <= data(7 downto 0);
            when "0110" =>
              val2 <= data(7 downto 0);
            when "0111" =>
              val3 <= data(7 downto 0);
            when "1000" =>
              val4 <= data(7 downto 0);
            when "1001" =>
              val5 <= data(7 downto 0);
            when "1010" =>
              val6 <= data(7 downto 0);
            when "1011" =>
              val7 <= data(7 downto 0);
            when "1100" =>
              for i in 3 downto 0 loop
                divider(i) <= data(i);
              end loop;  -- i
            when "1101" =>
              for i in 2 downto 0 loop
                shift(i) <= data(i);
              end loop;  -- i
            when "1110" =>
              --Hmm enabling this makes us run out of resources.
              --digital_out(3 downto 0) <= data(3 downto 0);
              --digital_out <= data(7 downto 0);
            when others => null;
          end case;
         when internal_ack1 =>
           req <= '1';
           if (ack = '1') then
             il_read_state <= idle;
           end if;
        when others =>
          il_read_state <= idle;
      end case;
    end if;
  end process read_ibus;
end read_from_ibus_arch;

library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;

entity tout is
  port (led1 : out std_logic;
       clk : in std_logic;
       id: in std_logic_vector(15 downto 0);
       --
       -- global
       --
```

```
493       reset: in std_logic;
494       --
495       -- meanings of these signals are reversed since they are named in the
496       -- master CPLD
497       --
498       sinphase_bus_request: buffer std_logic;
499       sinphase_bus_grant: in std_logic;
500       sinphase_strobe: in std_logic;   -- Switch counter channels
501       sinphase_zeros: out std_logic;     -- Mark zero trnsitions (phase shifted
502       sinphase_mids: in std_logic;      -- Mark mid-point transisitons.
503       external_zeros: in std_logic;     -- External phase transitions.
504       AO_FROM_PC_STROBE:in std_logic;
505       AO_FROM_PC_ACK:out std_logic;
506       DIO_TO_FIBER_READ_REQ:in std_logic;
507       AO_TO_PC_STROBE:in std_logic;
508       AO_TO_PC_ACK:in std_logic;
509       SOFT_RESET:in std_logic;
510       --
511       -- DA control
512       --
513       shdn: buffer std_logic_vector(2 downto 0);
514       da_not_clr: out std_logic;
515       da_not_wr: out std_logic;
516       da_not_cs: out std_logic;
517       da_not_ldab: out std_logic;
518       da_not_ldcd: out std_logic;
519       da_not_ldef: out std_logic;
520       da_not_ldgh: out std_logic;
521       dad: out std_logic_vector(12 downto 0);
522       daaddr: out std_logic_vector(2 downto 0);
523       dio: out std_logic_vector(7 downto 0);
524       dio_dir: buffer std_logic;
525       dio_tristate: buffer std_logic
526     );
527 end tout;
528
529
530 -- purpose: Collect together preceeding modules
531 architecture tout_arch of tout is
532     signal write_oe : std_logic;
533     signal read_oe : std_logic;
534     signal ibus_read_frame_flag: std_logic;
535     signal in_strobe_int : std_logic;
536     signal in_is_frame_int : std_logic;
537     signal phase1: unsigned(12 downto 0);  -- Phase to switch sinphase_
538                                            -- strobe from 0 to 1
539     signal phase2 : unsigned(12 downto 0);  -- Phase to switch sinphase
540                                            -- strobe from 1 to 0
541     signal steplen : unsigned(9 downto 0);
542     signal val0 : std_logic_vector(7 downto 0);
543     signal val1 : std_logic_vector(7 downto 0);
544     signal val2 : std_logic_vector(7 downto 0);
545     signal val3 : std_logic_vector(7 downto 0);
546     signal val4 : std_logic_vector(7 downto 0);
547     signal val5 : std_logic_vector(7 downto 0);
548     signal val6 : std_logic_vector(7 downto 0);
549     signal val7 : std_logic_vector(7 downto 0);
550     signal shift : unsigned(2 downto 0);
551     signal predev : unsigned(3 downto 0);
552     signal sinphase_dac: std_logic(7 downto 0);
553     signal to_dac_from_pc_req : std_logic;
554     signal to_dac_from_pc_ack : std_logic;
555     signal to_dac_from_sp_req : std_logic;
556     signal to_dac_from_sp_ack : std_logic;
557     signal dac_data_from_pc : std_logic_vector(15 downto 0);
558     signal dac_from_sp : std_logic_vector(7 downto 0);
559     signal il_ack: std_logic;
560     signal this_chip_selected : std_logic;
561 --    signal request_reset : std_logic;
562     signal sp_req: std_logic;
563     signal sp_ack: std_logic;
564     signal address_from_pc : std_logic_vector(3 downto 0);
565 begin
566     shdn(0) <= '0';
567     shdn(1) <= '0';
568     shdn(2) <= '0';
569     dio_tristate <= '1';
570
571 dacwr : write_to_dac port map (
572   clk     => clk,
573   reset   => reset,
574   sp_req  => sp_req,
575   sp_ack  => sp_ack,
576   pc_req  => to_dac_from_pc_req,
577   pc_ack  => to_dac_from_pc_ack,
578   sp_data => dac_from_sp,
579   sp_addr => "000",
580   pc_data => dac_data_from_pc(12 downto 0),
581   pc_addr => address_from_pc,
582   dac_addr => daaddr,
583   dac_data => dad,
584   n_dac_wr  => da_not_wr,
585   n_dac_cs => da_not_cs,
586   n_dac_clr => da_not_clr,
587   n_dac_ldab => da_not_ldab,
588   n_dac_ldcd => da_not_ldcd,
589   n_dac_ldef => da_not_ldef,
590   n_dac_ldgh => da_not_ldgh,
591   shift => shift);
592
593 -- purpose: Takes care of tristating the internal lower bus ack line which is
594 -- multi-driven by several chips. This is a clocked process because we wish to
595 -- make sure that the ack signal is set to zero before the output is tristated.
596 -- In this way the output holding latch always holds the line low in the
597     AO_FROM_PC_ACK <= il_ack when (this_chip_selected = '1') else 'Z';
598
599     rio: read_from_ibus port map (
600       clk => clk,
601       reset => reset,
602       il_req => AO_FROM_PC_STROBE,
603       il_ack => il_ack,
604       il_io => id,
605       data => dac_data_from_pc,
606       address => address_from_pc,
607       req => to_dac_from_pc_req,
608       ack => to_dac_from_pc_ack,
609       i_am_remote => '0',
610       my_ctrl_address => "0101",
611       my_data_address => "0100",
612 --      request_reset => request_reset,
613       il_i_am_addressed => this_chip_selected,
614       phase1 => phase1,
615       phase2 => phase2,
```

```
616       steplen => steplen,
617       val0 => val0,
618       val1 => val1,
619       val2 => val2,
620       val3 => val3,
621       val4 => val4,
622       val5 => val5,
623       val6 => val6,
624       val7 => val7,
625       divider => predev,
626       shift => shift,
627       digital_out => dio
628       );
629
630      spg : sinphase port map (
631        clk     => clk,
632        reset   => reset,
633        phase1  => phase1,
634        phase2  => phase2,
635        steplen => steplen,
636        val0    => val0,
637        val1    => val1,
638        val2    => val2,
639        val3    => val3,
640        val4    => val4,
641        val5    => val5,
642        val6    => val6,
643        val7    => val7,
644        predev  => predev,
645        sinphase_zeros  => sinphase_zeros,
646        dac_req => sp_req,
647        dac_ack => sp_ack,
648        dac     => dac_from_sp);
649
650 end  tout_arch;
651
652
653
654
655
656
657
658
659
660
661
662
663
664
```