```
--------------------------------------------------------------------------
-- HEIG-VD, Haute Ecole d'Ingenierie et de Gestion du canton de Vaud
-- Institut REDS, Reconfigurable & Embedded Digital Systems
--
-- File         : axi4lite_slave.vhd
-- Author       : E. Messerli    27.07.2017
-- Description  : slave interface AXI  (without burst)
-- used for     : SOCF lab
--| Modifications |-------------------------------------------------------
-- Ver  Date       Auteur  Description
-- 1.0  26.03.2019  EMI    Adaptation du chablon pour les etudiants
-- 1.1  03.04.2020  ISS    Complète le chablon pour le laboratoire 5
--------------------------------------------------------------------------------

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

entity axi4lite_slave is
    generic (
        -- Users to add parameters here

        -- User parameters ends

        -- Width of S_AXI data bus
        AXI_DATA_WIDTH  : integer   := 32;  -- 32 or 64 bits
        -- Width of S_AXI address bus
        AXI_ADDR_WIDTH  : integer   := 12
    );
    port (
        -- AXI4-Lite
        axi_clk_i       : in  std_logic;
        axi_reset_i     : in  std_logic;

        -- Write Address Channel
        axi_awaddr_i    : in  std_logic_vector(AXI_ADDR_WIDTH-1 downto 0);
        axi_awprot_i    : in  std_logic_vector( 2 downto 0);    -- not used
        axi_awvalid_i   : in  std_logic;
        axi_awready_o   : out std_logic;

        -- Write Data Channel
        axi_wdata_i     : in  std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
        axi_wstrb_i     : in std_logic_vector((AXI_DATA_WIDTH/8)-1 downto 0);
        axi_wvalid_i    : in  std_logic;
        axi_wready_o    : out std_logic;

        -- Write Response Channel
        axi_bresp_o     : out std_logic_vector(1 downto 0);
        axi_bvalid_o    : out std_logic;
        axi_bready_i    : in  std_logic;

        -- Read Address Channel
        axi_araddr_i    : in  std_logic_vector(AXI_ADDR_WIDTH-1 downto 0);
        axi_arprot_i    : in  std_logic_vector( 2 downto 0);    -- not used
        axi_arvalid_i   : in  std_logic;
        axi_arready_o   : out std_logic;

        -- Read Data Channel
        axi_rdata_o     : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
        axi_rresp_o     : out std_logic_vector(1 downto 0);
        axi_rvalid_o    : out std_logic;
        axi_rready_i    : in  std_logic;

        -- User input-output
        switch_i        : in  std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
        key_i           : in  std_logic_vector(AXI_DATA_WIDTH-1 downto 0);

        leds_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);

```

```
70          hex0_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
71          hex1_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
72          hex2_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
73          hex3_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
74          hex4_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
75          hex5_o          : out std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
76
77
78          -- Interruption
79          irq_o           : out std_logic
80      );
81  end entity axi4lite_slave;
82
83  architecture rtl of axi4lite_slave is
84
85      signal reset_s : std_logic;
86
87      -- local parameter for addressing 32 bit / 64 bits, cst: AXI_DATA_WIDTH
88      -- ADDR_LSB is used for addressing word 32/64 bits registers/memories
89      -- ADDR_LSB = 2 for 32 bits (n-1 downto 2)
90      -- ADDR_LSB = 3 for 64 bits (n-1 downto 3)
91      constant ADDR_LSB           : integer := (AXI_DATA_WIDTH/32)+ 1;
92
93      --------------- SIGNAUX AXI 4 LIGHT -------------------
94
95      --signal for the AXI slave
96      --intern signal for output
97      signal axi_awready_s        : std_logic;
98      signal axi_arready_s        : std_logic;
99
100     signal axi_wready_s         : std_logic;
101     signal axi_rready_s         : std_logic;
102
103     signal axi_rvalid_s         : std_logic;
104     signal axi_rresp_s          : std_logic_vector(1 downto 0);
105     signal axi_rdata_mem_s      : std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
106
107     -- write enable
108     signal axi_data_wren_s      : std_logic;
109
110      --intern signal for the axi interface
111     signal axi_waddr_mem_s      : std_logic_vector(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
112     signal axi_araddr_mem_s     : std_logic_vector(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
113
114     signal axi_wdata_mem_s      : std_logic_vector(AXI_DATA_WIDTH-1 downto 0);
115     signal axi_wstrb_mem_s      : std_logic_vector((AXI_DATA_WIDTH/8)-1 downto 0);
116     -- signal axi_araddr_mem_s    : std_logic_vector(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
117
118     signal axi_bresp_s          : std_logic_vector(1 downto 0);
119     signal axi_bvalid_s         : std_logic;
120
121
122     --------------- SIGNAUX ENTREES / SORTIES ---------------
123
124     constant registre_cst_mem   : std_logic_vector(AXI_DATA_WIDTH-1 downto 0):=
        x"deedbeef";
125     signal registre_test_mem    : std_logic_vector(AXI_DATA_WIDTH-1 downto 0):=
        x"12345678";
126
127     -- signal for registre input (switch / key)
128     signal registre_switch_mem  : std_logic_vector(9 downto 0) := (others => 'X');
129     signal registre_key_mem     : std_logic_vector(3 downto 0) := (others => 'X');
130
131     -- signal for registre leds
132     signal registre_led_mem     : std_logic_vector(9 downto 0) := (others => 'X');
133
134     -- signal for registre 7 seg
135     signal registre_hex0_mem    : std_logic_vector(6 downto 0) := (others => 'X');
136     signal registre_hex1_mem    : std_logic_vector(6 downto 0) := (others => 'X');
```

```
137       signal registre_hex2_mem    : std_logic_vector(6 downto 0) := (others => 'X');
138       signal registre_hex3_mem    : std_logic_vector(6 downto 0) := (others => 'X');
139       signal registre_hex4_mem    : std_logic_vector(6 downto 0) := (others => 'X');
140       signal registre_hex5_mem    : std_logic_vector(6 downto 0) := (others => 'X');
141
142       --------------- SIGNAUX GESTION IRQ ---------------
143       signal irq_s                : std_logic;
144       signal irq_source           : std_logic_vector(3 downto 0) := (others => '0');
145       signal key_val_save         : std_logic_vector(3 downto 0) := (others => '1');
146       -- par défaut, toutes les irq actives
147       signal key_irq_mask         : std_logic_vector(3 downto 0) := (others => '0');
148
149   begin
150
151        -- mise à jour des entrées
152       reset_s  <= axi_reset_i;
153
154       registre_switch_mem <= switch_i(9 downto 0);
155       registre_key_mem    <= key_i(3 downto 0);
156
157
158
159   ---------------------------------------------------------
160   -- Write address channel
161
162       process (reset_s, axi_clk_i)
163       begin
164           -- En cas de reset
165           if reset_s = '1' then
166               -- Valeur par défaut
167               axi_awready_s <= '0';
168               axi_waddr_mem_s <= (others => '0');
169           elsif rising_edge(axi_clk_i) then
170               -- Si une adresse d'écriture est valide
171               if (axi_awready_s = '0' and axi_awvalid_i = '1')  then --and axi_wvalid_i =
                  '1') then  modif EMI 10juil2018
172                   -- slave is ready to accept write address when
173                   -- there is a valid write address
174                   axi_awready_s <= '1';
175                   -- Write Address memorizing
176                   axi_waddr_mem_s <= axi_awaddr_i(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
177               else
178                   axi_awready_s <= '0';
179                   axi_waddr_mem_s <= (others => '0');
180               end if;
181           end if;
182       end process;
183       axi_awready_o <= axi_awready_s;
184
185
186   ---------------------------------------------------------
187   -- Write data channel
188
189       -- Implement axi_wready generation
190       process (reset_s, axi_clk_i)
191       begin
192           -- En cas de reset
193           if reset_s = '1' then
194               -- Valeur par défaut
195               axi_wready_s    <= '0';
196               axi_wdata_mem_s <= (others => '0');
197               axi_wstrb_mem_s <= (others => '0');
198           elsif rising_edge(axi_clk_i) then
199               -- Si les données d'écriture est valide
200               if (axi_wready_s = '0' and axi_wvalid_i = '1')  then
201                   -- slave is ready to accept write data when
202                   -- there is a valid write data
203                   axi_wready_s <= '1';
204
```

```
                -- Read axi_wstrb_i
                axi_wstrb_mem_s <= axi_wstrb_i((AXI_DATA_WIDTH/8)-1 downto 0);


                -- Mémorisation des données à écrire en fonction du paramètre strobe
                axi_wdata_mem_s <= (others => '0');

                if (axi_wstrb_i(0) = '1') then
                    axi_wdata_mem_s(7 downto 0) <= axi_wdata_i(7 downto 0);
                end if;
                if (axi_wstrb_i(1) = '1') then
                    axi_wdata_mem_s(15 downto 8) <= axi_wdata_i(15 downto 8);
                end if;
                if (axi_wstrb_i(2) = '1') then
                    axi_wdata_mem_s(23 downto 16) <= axi_wdata_i(23 downto 16);
                end if;
                if (axi_wstrb_i(3) = '1') then
                    axi_wdata_mem_s(31 downto 24) <= axi_wdata_i(31 downto 24);
                end if;

                -- Test sans la fonctionnalité strobe
                -- axi_wdata_mem_s <= axi_wdata_i;

            else
                axi_wready_s <= '0';
                axi_wdata_mem_s <= (others => '0');
                axi_wstrb_mem_s <= (others => '0');

            end if;
        end if;
    end process;

    -- Met à jour la sortie
    axi_wready_o <= axi_wready_s;


    -- condition to write data : si on est prêt à écrire
    axi_data_wren_s <= '1' when axi_wready_s = '1' else
                       '0';


    process (reset_s, axi_clk_i)
        --number address to access 32 or 64 bits data
        variable int_waddr_v : natural;
    begin
        if reset_s = '1' then
            -- Valeur par défaut : RESET
            registre_test_mem <= x"12345678";
            registre_led_mem  <= "0101010101";
            registre_hex0_mem <= "1000000" ;
            registre_hex1_mem <= "1111001";
            registre_hex2_mem <= "0100100";
            registre_hex3_mem <= "0110000";
            registre_hex4_mem <= "0011001";
            registre_hex5_mem <= "0010010";

            key_irq_mask      <= "0000";

        elsif rising_edge(axi_clk_i) then
            -- Si une écriture est active
            if axi_data_wren_s = '1' then
                -- convertie l'adresse d'écriture en integer
                int_waddr_v   := to_integer(unsigned(axi_waddr_mem_s));
                case int_waddr_v is
                    -- offset 0 : constante
                    when 0    =>
                    -- offset 4 : registre de test
                    when 1    =>
                        registre_test_mem <= axi_wdata_mem_s;
```

```
274
275                     -- offset 64 : leds
276                     when 64   =>
277                         registre_led_mem <= axi_wdata_mem_s(9 downto 0);
278
279                     -- offset 130 : mask irq key
280                     when 130   =>
281                         key_irq_mask <= axi_wdata_mem_s(3 downto 0);
282
283                     -- offset 256 - 276 : afficheur 7 seg
284                     when 256   =>
285                         registre_hex0_mem <= axi_wdata_mem_s(6 downto 0);
286                     when 260   =>
287                         registre_hex1_mem <= axi_wdata_mem_s(6 downto 0);
288                     when 264   =>
289                         registre_hex2_mem <= axi_wdata_mem_s(6 downto 0);
290                     when 268   =>
291                         registre_hex3_mem <= axi_wdata_mem_s(6 downto 0);
292                     when 272   =>
293                         registre_hex4_mem <= axi_wdata_mem_s(6 downto 0);
294                     when 276   =>
295                         registre_hex5_mem <= axi_wdata_mem_s(6 downto 0);
296
297
298                     when others => null;
299                 end case;
300             end if;
301         end if;
302     end process;
303
304
305 -----------------------------------------------------------
306 -- Write response channel
307
308     process (reset_s, axi_clk_i)
309     begin
310         -- En cas de reset
311         if reset_s = '1' then
312             -- Valeur par défaut
313             axi_bresp_s    <= "00";
314             axi_bvalid_s   <= '0';
315         elsif rising_edge(axi_clk_i) then
316             -- Si le master est pret à lire la réponse
317             if (axi_bvalid_s = '0' and axi_bready_i = '1')  then
318                 -- slave is ready to accept write data when
319                 -- there is a valid write data
320                 axi_bvalid_s <= '1';
321                 -- Write response
322                 axi_bresp_s    <= "00";
323             else
324                 axi_bvalid_s <= '0';
325                 axi_bresp_s    <= "--";
326
327             end if;
328         end if;
329     end process;
330     -- Met à jours les sorties
331     axi_bresp_o <= axi_bresp_s;
332     axi_bvalid_o <= axi_bvalid_s;
333
334
335
336 -----------------------------------------------------------
337 -- Read address channel
338
339     process (reset_s, axi_clk_i)
340     begin
341         -- en cas de reset
342         if reset_s = '1' then
```

```
343             -- valeur par défaut
344            axi_aready_s    <= '0';
345            axi_araddr_mem_s <= (others => '1');
346         elsif rising_edge(axi_clk_i) then
347             -- Si une adresse de lecture est valide
348             if axi_aready_s = '0' and axi_arvalid_i = '1' then
349                 -- indicates that the slave has acceped the valid read address
350                 axi_aready_s    <= '1';
351                 -- Read Address memorizing
352                 axi_araddr_mem_s <= axi_araddr_i(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
353             else
354                 axi_aready_s    <= '0';
355             end if;
356         end if;
357     end process;
358     -- Met à jour la sortie
359     axi_aready_o <= axi_aready_s;
360 
361 ---------------------------------------------------------
362 -- Read data channel
363 
364     -- Implement axi_wready generation
365     process (reset_s, axi_clk_i)
366     --number address to access 32 or 64 bits data
367         variable int_raddr_v : natural;
368     begin
369 
370         -- En cas de reset
371         if reset_s = '1' then
372             -- valeur par défaut
373             axi_rvalid_s    <= '0';
374             axi_rdata_mem_s <= (others => '0');
375             axi_rresp_s     <= "00";
376 
377             irq_source <= "0000";
378             irq_s <= '0';
379 
380         elsif rising_edge(axi_clk_i) then
381             -- Gestion des interruptions
382             if (key_val_save(0) /= registre_key_mem(0) and registre_key_mem(0) = '0'
                and key_irq_mask(0) = '0') then
383                 irq_source(0) <= '1';
384                 irq_s <= '1';
385             elsif (key_val_save(1) /= registre_key_mem(1) and registre_key_mem(1) = '0'
                and key_irq_mask(1) = '0') then
386                 irq_source(1) <= '1';
387                 irq_s <= '1';
388             elsif (key_val_save(2) /= registre_key_mem(2) and registre_key_mem(2) = '0'
                and key_irq_mask(2) = '0') then
389                 irq_source(2) <= '1';
390                 irq_s <= '1';
391             elsif (key_val_save(3) /= registre_key_mem(3) and registre_key_mem(3) = '0'
                and key_irq_mask(3) = '0') then
392                 irq_source(3) <= '1';
393                 irq_s <= '1';
394             end if;
395             -- Met à jour l'ancienne valeur des keys
396             key_val_save <= registre_key_mem;
397 
398             -- Si une lecture est faite
399             if (axi_aready_s = '1' and axi_rvalid_s = '0')  then
400 
401                 -- Pré-charge une lecture à 0
402                 axi_rdata_mem_s <= (others => '0');
403 
404                 -- slave is ready to accept write data when
405                 -- there is a valid write data
406                 axi_rvalid_s <= '1';
407 
```

```
408                 -- read Data go
409                 int_raddr_v   := to_integer(unsigned(axi_araddr_mem_s));
410                 axi_rresp_s    <= "00";
411 
412                 -- En fonction de l'adresse qu'on souhaite lire
413                 case int_raddr_v is
414                     -- Lecture de la constante
415                     when 0   =>
416                         axi_rdata_mem_s <= registre_cst_mem;
417                     -- Lecture du registre de test
418                     when 1   =>
419                         axi_rdata_mem_s <= registre_test_mem;
420                     -- Lecture des leds
421                     when 64   =>
422                         axi_rdata_mem_s(9 downto 0) <= registre_led_mem;
423                      -- Lecture des keys
424                     when 128   =>
425                         axi_rdata_mem_s(3 downto 0) <= registre_key_mem;
426                     -- lecture de la source d'interruption et acquitement
427                     when 129   =>
428                         axi_rdata_mem_s(3 downto 0) <= irq_source;
429                         irq_s <= '0';
430                         irq_source <= "0000";
431 
432                     -- lecture des masque des irq
433                     when 130   =>
434                         axi_rdata_mem_s(3 downto 0) <= key_irq_mask;
435                     -- Lecture des switches
436                     when 192   =>
437                         axi_rdata_mem_s(9 downto 0) <= registre_switch_mem;
438 
439                     -- Lecture d'un afficheur 7 seg (256 - 276)
440                     when 256   =>
441                         axi_rdata_mem_s(6 downto 0) <= registre_hex0_mem;
442                     when 260   =>
443                         axi_rdata_mem_s(6 downto 0) <= registre_hex1_mem;
444                     when 264   =>
445                         axi_rdata_mem_s(6 downto 0) <= registre_hex2_mem;
446                     when 268   =>
447                         axi_rdata_mem_s(6 downto 0) <= registre_hex3_mem;
448                     when 272   =>
449                         axi_rdata_mem_s(6 downto 0) <= registre_hex4_mem;
450                     when 276   =>
451                         axi_rdata_mem_s(6 downto 0) <= registre_hex5_mem;
452 
453 
454                     when others =>
455                         axi_rresp_s    <= "00";
456                 end case;
457 
458             else
459                 axi_rvalid_s <= '0';
460                 axi_rresp_s    <= "--";
461 
462             end if;
463         end if;
464     end process;
465 
466     -- Mise à jour de la ligne l'interruption
467     irq_o <= irq_s;
468 
469     -- Mise à jour de la validité de lecture
470     axi_rvalid_o <= axi_rvalid_s;
471 
472     -- Mise à jour des données lues
473     axi_rdata_o <= axi_rdata_mem_s;
474 
475     -- Mise à jour de la réponse de lecture
476     axi_rresp_o <= axi_rresp_s;
```

```
477
478
479         -- Mise à jour des sorties
480         leds_o(9 downto 0)      <=  registre_led_mem;
481
482         hex0_o(6 downto 0)      <=  registre_hex0_mem;
483         hex1_o(6 downto 0)      <=  registre_hex1_mem;
484         hex2_o(6 downto 0)      <=  registre_hex2_mem;
485         hex3_o(6 downto 0)      <=  registre_hex3_mem;
486         hex4_o(6 downto 0)      <=  registre_hex4_mem;
487         hex5_o(6 downto 0)      <=  registre_hex5_mem;
488
489
490     end rtl;
491
```