```
--------------------------------------------------------------------------
-- 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  26.04.2019  EMI    Adaptation du chablon pour les etudiants
-- 1.1  03.05.2020  ISS    Complète le chablon pour le laboratoire 5 Partie 2
-- 1.2  08.05.2020  ISS    Ajout de la fonctionnalité edge pour les irq
--------------------------------------------------------------------------------

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

```
137     signal registre_hex1_mem    : std_logic_vector(6 downto 0) := (others => 'X');
138     signal registre_hex2_mem    : std_logic_vector(6 downto 0) := (others => 'X');
139     signal registre_hex3_mem    : std_logic_vector(6 downto 0) := (others => 'X');
140     signal registre_hex4_mem    : std_logic_vector(6 downto 0) := (others => 'X');
141     signal registre_hex5_mem    : std_logic_vector(6 downto 0) := (others => 'X');
142
143     --------------- SIGNAUX GESTION IRQ ---------------
144     signal irq_s                : std_logic;
145     signal irq_source           : std_logic_vector(3 downto 0) := (others => '0');
146     signal key_val_save         : std_logic_vector(3 downto 0) := (others => '1');
147     -- par défaut, toutes les irq actives
148     signal key_irq_mask         : std_logic_vector(3 downto 0) := (others => '0');
149     -- par défaut, toutes les irq sur flanc descendant
150     signal key_irq_edge         : std_logic_vector(3 downto 0) := (others => '0');
151
152 begin
153
154      -- mise à jour des entrées
155     reset_s  <= axi_reset_i;
156
157     registre_switch_mem <= switch_i(9 downto 0);
158     registre_key_mem    <= key_i(3 downto 0);
159
160
161
162 -------------------------------------------------------------
163 -- Write address channel
164
165     process (reset_s, axi_clk_i)
166     begin
167         -- En cas de reset
168         if reset_s = '1' then
169             -- Valeur par défaut
170             axi_awready_s <= '0';
171             axi_waddr_mem_s <= (others => '0');
172         elsif rising_edge(axi_clk_i) then
173             -- Si une adresse d'écriture est valide
174             if (axi_awready_s = '0' and axi_awvalid_i = '1')  then --and axi_wvalid_i =
                '1') then  modif EMI 10juil2018
175                 -- slave is ready to accept write address when
176                 -- there is a valid write address
177                 axi_awready_s <= '1';
178                 -- Write Address memorizing
179                 axi_waddr_mem_s <= axi_awaddr_i(AXI_ADDR_WIDTH-1 downto ADDR_LSB);
180             else
181                 axi_awready_s <= '0';
182                 axi_waddr_mem_s <= (others => '0');
183             end if;
184         end if;
185     end process;
186     axi_awready_o <= axi_awready_s;
187
188
189 -------------------------------------------------------------
190 -- Write data channel
191
192     -- Implement axi_wready generation
193     process (reset_s, axi_clk_i)
194     begin
195         -- En cas de reset
196         if reset_s = '1' then
197             -- Valeur par défaut
198             axi_wready_s    <= '0';
199             axi_wdata_mem_s <= (others => '0');
200             axi_wstrb_mem_s <= (others => '0');
201         elsif rising_edge(axi_clk_i) then
202             -- Si les données d'écriture est valide
203             if (axi_wready_s = '0' and axi_wvalid_i = '1')  then
204                 -- slave is ready to accept write data when
```

```
                -- there is a valid write data
                axi_wready_s <= '1';

                -- 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";
            key_irq_edge      <= "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;

                    -- offset 64 : leds
                    when 64   =>
                        registre_led_mem <= axi_wdata_mem_s(9 downto 0);

                    -- offset 130 : mask irq key
                    when 130   =>
                        key_irq_mask <= axi_wdata_mem_s(3 downto 0);
                    -- offset 130 : mask irq key
                    when 131   =>
                        key_irq_edge <= axi_wdata_mem_s(3 downto 0);

                    -- offset 256 - 276 : afficheur 7 seg
                    when 256   =>
                        registre_hex0_mem <= axi_wdata_mem_s(6 downto 0);
                    when 260   =>
                        registre_hex1_mem <= axi_wdata_mem_s(6 downto 0);
                    when 264   =>
                        registre_hex2_mem <= axi_wdata_mem_s(6 downto 0);
                    when 268   =>
                        registre_hex3_mem <= axi_wdata_mem_s(6 downto 0);
                    when 272   =>
                        registre_hex4_mem <= axi_wdata_mem_s(6 downto 0);
                    when 276   =>
                        registre_hex5_mem <= axi_wdata_mem_s(6 downto 0);


                    when others => null;
                end case;
            end if;
        end if;
    end process;


-------------------------------------------------------------
-- Write response channel

    process (reset_s, axi_clk_i)
    begin
        -- En cas de reset
        if reset_s = '1' then
            -- Valeur par défaut
            axi_bresp_s    <= "00";
            axi_bvalid_s   <= '0';
        elsif rising_edge(axi_clk_i) then
            -- Si le master est pret à lire la réponse
            if (axi_bvalid_s = '0' and axi_bready_i = '1')  then
                -- slave is ready to accept write data when
                -- there is a valid write data
                axi_bvalid_s <= '1';
                -- Write response
                axi_bresp_s    <= "00";
            else
                axi_bvalid_s <= '0';
                axi_bresp_s    <= "--";

            end if;
        end if;
    end process;
    -- Met à jours les sorties
    axi_bresp_o <= axi_bresp_s;
    axi_bvalid_o <= axi_bvalid_s;
```

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

```
408                 -- Pré-charge une lecture à 0
409                 axi_rdata_mem_s <= (others => '0');
410 
411                 -- slave is ready to accept write data when
412                 -- there is a valid write data
413                 axi_rvalid_s <= '1';
414 
415                 -- read Data go
416                 int_raddr_v   := to_integer(unsigned(axi_araddr_mem_s));
417                 axi_rresp_s    <= "00";
418 
419                 -- En fonction de l'adresse qu'on souhaite lire
420                 case int_raddr_v is
421                     -- Lecture de la constante
422                     when 0   =>
423                         axi_rdata_mem_s <= registre_cst_mem;
424                     -- Lecture du registre de test
425                     when 1   =>
426                         axi_rdata_mem_s <= registre_test_mem;
427                     -- Lecture des leds
428                     when 64   =>
429                         axi_rdata_mem_s(9 downto 0) <= registre_led_mem;
430                      -- Lecture des keys
431                     when 128   =>
432                         axi_rdata_mem_s(3 downto 0) <= registre_key_mem;
433                     -- lecture de la source d'interruption et acquitement
434                     when 129   =>
435                         axi_rdata_mem_s(3 downto 0) <= irq_source;
436                         irq_s <= '0';
437                         irq_source <= "0000";
438 
439                     -- lecture des masque des irq
440                     when 130   =>
441                         axi_rdata_mem_s(3 downto 0) <= key_irq_mask;
442                     -- lecture des masque des irq
443                     when 131   =>
444                         axi_rdata_mem_s(3 downto 0) <= key_irq_edge;
445 
446                     -- Lecture des switches
447                     when 192   =>
448                         axi_rdata_mem_s(9 downto 0) <= registre_switch_mem;
449 
450                     -- Lecture d'un afficheur 7 seg (256 - 276)
451                     when 256   =>
452                         axi_rdata_mem_s(6 downto 0) <= registre_hex0_mem;
453                     when 260   =>
454                         axi_rdata_mem_s(6 downto 0) <= registre_hex1_mem;
455                     when 264   =>
456                         axi_rdata_mem_s(6 downto 0) <= registre_hex2_mem;
457                     when 268   =>
458                         axi_rdata_mem_s(6 downto 0) <= registre_hex3_mem;
459                     when 272   =>
460                         axi_rdata_mem_s(6 downto 0) <= registre_hex4_mem;
461                     when 276   =>
462                         axi_rdata_mem_s(6 downto 0) <= registre_hex5_mem;
463 
464 
465                     when others =>
466                         axi_rresp_s    <= "00";
467                 end case;
468 
469             else
470                 axi_rvalid_s <= '0';
471                 axi_rresp_s    <= "--";
472 
473             end if;
474         end if;
475     end process;
476 
```

```
477       -- Mise à jour de la ligne l'interruption
478       irq_o <= irq_s;
479
480       -- Mise à jour de la validité de lecture
481       axi_rvalid_o <= axi_rvalid_s;
482
483       -- Mise à jour des données lues
484       axi_rdata_o <= axi_rdata_mem_s;
485
486       -- Mise à jour de la réponse de lecture
487       axi_rresp_o <= axi_rresp_s;
488
489
490       -- Mise à jour des sorties
491       leds_o(9 downto 0)      <=  registre_led_mem;
492
493       hex0_o(6 downto 0)      <=  registre_hex0_mem;
494       hex1_o(6 downto 0)      <=  registre_hex1_mem;
495       hex2_o(6 downto 0)      <=  registre_hex2_mem;
496       hex3_o(6 downto 0)      <=  registre_hex3_mem;
497       hex4_o(6 downto 0)      <=  registre_hex4_mem;
498       hex5_o(6 downto 0)      <=  registre_hex5_mem;
499
500
501   end rtl;
502
```