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

entity esempio_v1_0_S00_AXI is
    generic (
        -- Users to add parameters here

        -- User parameters ends
        -- Do not modify the parameters beyond this line

        -- Width of S_AXI data bus
        C_S_AXI_DATA_WIDTH  : integer    := 32;
        -- Width of S_AXI address bus
        C_S_AXI_ADDR_WIDTH  : integer    := 4
    );
    port (
        -- Users to add ports here

        -- User ports ends
        -- Do not modify the ports beyond this line

        -- Global Clock Signal
        S_AXI_ACLK  : in std_logic;
        -- Global Reset Signal. This Signal is Active LOW
        S_AXI_ARESETN    : in std_logic;
        -- Write address (issued by master, acceped by Slave)
        S_AXI_AWADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Write channel Protection type. This signal indicates the
            -- privilege and security level of the transaction, and whether
            -- the transaction is a data access or an instruction access.
        S_AXI_AWPROT    : in std_logic_vector(2 downto 0);
        -- Write address valid. This signal indicates that the master signaling
            -- valid write address and control information.
        S_AXI_AWVALID   : in std_logic;
        -- Write address ready. This signal indicates that the slave is ready
            -- to accept an address and associated control signals.
        S_AXI_AWREADY   : out std_logic;
        -- Write data (issued by master, acceped by Slave)
        S_AXI_WDATA : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Write strobes. This signal indicates which byte lanes hold
            -- valid data. There is one write strobe bit for each eight
            -- bits of the write data bus.
        S_AXI_WSTRB : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
        -- Write valid. This signal indicates that valid write
            -- data and strobes are available.
        S_AXI_WVALID    : in std_logic;
        -- Write ready. This signal indicates that the slave
            -- can accept the write data.
        S_AXI_WREADY    : out std_logic;
        -- Write response. This signal indicates the status
            -- of the write transaction.
        S_AXI_BRESP : out std_logic_vector(1 downto 0);
        -- Write response valid. This signal indicates that the channel
            -- is signaling a valid write response.
        S_AXI_BVALID    : out std_logic;
        -- Response ready. This signal indicates that the master
            -- can accept a write response.
        S_AXI_BREADY    : in std_logic;
        -- Read address (issued by master, acceped by Slave)
        S_AXI_ARADDR    : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
        -- Protection type. This signal indicates the privilege
            -- and security level of the transaction, and whether the
            -- transaction is a data access or an instruction access.
        S_AXI_ARPROT    : in std_logic_vector(2 downto 0);
        -- Read address valid. This signal indicates that the channel
            -- is signaling valid read address and control information.
        S_AXI_ARVALID   : in std_logic;
        -- Read address ready. This signal indicates that the slave is
```

```
70              -- ready to accept an address and associated control signals.
71          S_AXI_ARREADY   : out std_logic;
72          -- Read data (issued by slave)
73          S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
74          -- Read response. This signal indicates the status of the
75              -- read transfer.
76          S_AXI_RRESP : out std_logic_vector(1 downto 0);
77          -- Read valid. This signal indicates that the channel is
78              -- signaling the required read data.
79          S_AXI_RVALID    : out std_logic;
80          -- Read ready. This signal indicates that the master can
81              -- accept the read data and response information.
82          S_AXI_RREADY    : in std_logic
83      );
84  end esempio_v1_0_S00_AXI;
85
86  architecture arch_imp of esempio_v1_0_S00_AXI is
87
88      -- AXI4LITE signals
89      signal axi_awaddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
90      signal axi_awready  : std_logic;
91      signal axi_wready   : std_logic;
92      signal axi_bresp    : std_logic_vector(1 downto 0);
93      signal axi_bvalid   : std_logic;
94      signal axi_araddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
95      signal axi_arready  : std_logic;
96      signal axi_rdata    : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
97      signal axi_rresp    : std_logic_vector(1 downto 0);
98      signal axi_rvalid   : std_logic;
99
100     -- Example-specific design signals
101     -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
102     -- ADDR_LSB is used for addressing 32/64 bit registers/memories
103     -- ADDR_LSB = 2 for 32 bits (n downto 2)
104     -- ADDR_LSB = 3 for 64 bits (n downto 3)
105     constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
106     constant OPT_MEM_ADDR_BITS : integer := 1;
107     ------------------------------------------------
108     ---- Signals for user logic register space example
109     ------------------------------------------------
110     ---- Number of Slave Registers 4
111     signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
112     signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
113     signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
114     signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
115     signal slv_reg_rden : std_logic;
116     signal slv_reg_wren : std_logic;
117     signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
118     signal byte_index   : integer;
119     signal aw_en    : std_logic;
120
121 begin
122     -- I/O Connections assignments
123
124     S_AXI_AWREADY   <= axi_awready;
125     S_AXI_WREADY    <= axi_wready;
126     S_AXI_BRESP <= axi_bresp;
127     S_AXI_BVALID    <= axi_bvalid;
128     S_AXI_ARREADY   <= axi_arready;
129     S_AXI_RDATA <= axi_rdata;
130     S_AXI_RRESP <= axi_rresp;
131     S_AXI_RVALID    <= axi_rvalid;
132     -- Implement axi_awready generation
133     -- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
134     -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
135     -- de-asserted when reset is low.
136
137     process (S_AXI_ACLK)                                    --PROC 01
138     begin
```

```
139         if rising_edge(S_AXI_ACLK) then
140           if S_AXI_ARESETN = '0' then
141             axi_awready <= '0';
142             aw_en <= '1';
143           else
144             if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en
                = '1') then
145               -- slave is ready to accept write address when
146               -- there is a valid write address and write data
147               -- on the write address and data bus. This design
148               -- expects no outstanding transactions.
149                  axi_awready <= '1';
150                  aw_en <= '0';
151               elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then
152                  aw_en <= '1';
153                  axi_awready <= '0';
154             else
155               axi_awready <= '0';
156             end if;
157           end if;
158         end if;
159       end process;
160
161       -- Implement axi_awaddr latching
162       -- This process is used to latch the address when both
163       -- S_AXI_AWVALID and S_AXI_WVALID are valid.
164
165       process (S_AXI_ACLK)                                          --PROC 02
166       begin
167         if rising_edge(S_AXI_ACLK) then
168           if S_AXI_ARESETN = '0' then
169             axi_awaddr <= (others => '0');
170           else
171             if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' and aw_en
                = '1') then
172               -- Write Address latching
173               axi_awaddr <= S_AXI_AWADDR;
174             end if;
175           end if;
176         end if;
177       end process;
178
179       -- Implement axi_wready generation
180       -- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
181       -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
182       -- de-asserted when reset is low.
183
184       process (S_AXI_ACLK)                                          --PROC 03
185       begin
186         if rising_edge(S_AXI_ACLK) then
187           if S_AXI_ARESETN = '0' then
188             axi_wready <= '0';
189           else
190             if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1' and aw_en =
                 '1') then
191                -- slave is ready to accept write data when
192                -- there is a valid write address and write data
193                -- on the write address and data bus. This design
194                -- expects no outstanding transactions.
195                axi_wready <= '1';
196             else
197               axi_wready <= '0';
198             end if;
199           end if;
200         end if;
201       end process;
202
203       -- Implement memory mapped register select and write logic generation
204       -- The write data is accepted and written to memory mapped registers when
```

```
205       -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
          strobes are used to
206       -- select byte enables of slave registers while writing.
207       -- These registers are cleared when reset (active low) is applied.
208       -- Slave register write enable is asserted when valid address and data are available
209       -- and the slave is ready to accept the write address and write data.
210       slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID ;
211
212       process (S_AXI_ACLK)                                          --PROC 04
213       variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
214       begin
215         if rising_edge(S_AXI_ACLK) then
216           if S_AXI_ARESETN = '0' then
217             slv_reg0 <= (others => '0');
218             slv_reg1 <= (others => '0');
219             slv_reg2 <= (others => '0');
220             slv_reg3 <= (others => '0');
221           else
222             loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
223             if (slv_reg_wren = '1') then
224               case loc_addr is
225                 when b"00" =>
226                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
227                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
228                       -- Respective byte enables are asserted as per write
                          strobes
229                       -- slave registor 0
230                       slv_reg0(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(
                          byte_index*8+7 downto byte_index*8);
231                     end if;
232                   end loop;
233                 when b"01" =>
234                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
235                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
236                       -- Respective byte enables are asserted as per write
                          strobes
237                       -- slave registor 1
238                       slv_reg1(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(
                          byte_index*8+7 downto byte_index*8);
239                     end if;
240                   end loop;
241                 when b"10" =>
242                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
243                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
244                       -- Respective byte enables are asserted as per write
                          strobes
245                       -- slave registor 2
246                       slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(
                          byte_index*8+7 downto byte_index*8);
247                     end if;
248                   end loop;
249                 when b"11" =>
250                   for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
251                     if ( S_AXI_WSTRB(byte_index) = '1' ) then
252                       -- Respective byte enables are asserted as per write
                          strobes
253                       -- slave registor 3
254                       slv_reg3(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(
                          byte_index*8+7 downto byte_index*8);
255                     end if;
256                   end loop;
257                 when others =>
258                   slv_reg0 <= slv_reg0;
259                   slv_reg1 <= slv_reg1;
260                   slv_reg2 <= slv_reg2;
261                   slv_reg3 <= slv_reg3;
262               end case;
263             end if;
264           end if;
```

```
265         end if;
266       end process;
267
268       -- Implement write response logic generation
269       -- The write response and response valid signals are asserted by the slave
270       -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
271       -- This marks the acceptance of address and indicates the status of
272       -- write transaction.
273
274       process (S_AXI_ACLK)                                              --PROC 05
275       begin
276         if rising_edge(S_AXI_ACLK) then
277           if S_AXI_ARESETN = '0' then
278             axi_bvalid  <= '0';
279             axi_bresp   <= "00"; --need to work more on the responses
280           else
281             if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and
                S_AXI_WVALID = '1' and axi_bvalid = '0'  ) then
282               axi_bvalid <= '1';
283               axi_bresp  <= "00";
284             elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then   --check if bready is
                asserted while bvalid is high)
285               axi_bvalid <= '0';                                 -- (there is a
                possibility that bready is always asserted high)
286             end if;
287           end if;
288         end if;
289       end process;
290
291       -- Implement axi_arready generation
292       -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
293       -- S_AXI_ARVALID is asserted. axi_awready is
294       -- de-asserted when reset (active low) is asserted.
295       -- The read address is also latched when S_AXI_ARVALID is
296       -- asserted. axi_araddr is reset to zero on reset assertion.
297
298       process (S_AXI_ACLK)                                               --PROC 06
299       begin
300         if rising_edge(S_AXI_ACLK) then
301           if S_AXI_ARESETN = '0' then
302             axi_arready <= '0';
303             axi_araddr  <= (others => '1');
304           else
305             if (axi_arready = '0' and S_AXI_ARVALID = '1') then
306               -- indicates that the slave has acceped the valid read address
307               axi_arready <= '1';
308               -- Read Address latching
309               axi_araddr  <= S_AXI_ARADDR;
310             else
311               axi_arready <= '0';
312             end if;
313           end if;
314         end if;
315       end process;
316
317       -- Implement axi_arvalid generation
318       -- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
319       -- S_AXI_ARVALID and axi_arready are asserted. The slave registers
320       -- data are available on the axi_rdata bus at this instance. The
321       -- assertion of axi_rvalid marks the validity of read data on the
322       -- bus and axi_rresp indicates the status of read transaction.axi_rvalid
323       -- is deasserted on reset (active low). axi_rresp and axi_rdata are
324       -- cleared to zero on reset (active low).
325       process (S_AXI_ACLK)                                                 --PROC 07
326       begin
327         if rising_edge(S_AXI_ACLK) then
328           if S_AXI_ARESETN = '0' then
329             axi_rvalid <= '0';
330             axi_rresp  <= "00";
```

```
          else
            if (axi_aready = '1' and S_AXI_ARVALID = '1' and axi_rvalid = '0') then
              -- Valid read data is available at the read data bus
              axi_rvalid <= '1';
              axi_rresp  <= "00"; -- 'OKAY' response
            elsif (axi_rvalid = '1' and S_AXI_RREADY = '1') then
              -- Read data is accepted by the master
              axi_rvalid <= '0';
            end if;
          end if;
        end if;
      end process;

      -- Implement memory mapped register select and read logic generation
      -- Slave register read enable is asserted when valid address is available
      -- and the slave is ready to accept the read address.
      slv_reg_rden <= axi_aready and S_AXI_ARVALID and (not axi_rvalid) ;

      process (slv_reg0, slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
      slv_reg_rden)    --PROC 08
      variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
      begin
          -- Address decoding for reading registers
          loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
          case loc_addr is
            when b"00" =>
              reg_data_out <= slv_reg0;
            when b"01" =>
              reg_data_out <= slv_reg1;
            when b"10" =>
              reg_data_out <= slv_reg2;
            when b"11" =>
              reg_data_out <= slv_reg3;
            when others =>
              reg_data_out  <= (others => '0');
          end case;
      end process;

      -- Output register or memory read data
      process( S_AXI_ACLK ) is                                                --PROC 09
      begin
        if (rising_edge (S_AXI_ACLK)) then
          if ( S_AXI_ARESETN = '0' ) then
            axi_rdata  <= (others => '0');
          else
            if (slv_reg_rden = '1') then
              -- When there is a valid read address (S_AXI_ARVALID) with
              -- acceptance of read address by the slave (axi_aready),
              -- output the read dada
              -- Read address mux
                axi_rdata <= reg_data_out;     -- register read data
            end if;
          end if;
        end if;
      end process;


      -- Add user logic here

      -- User logic ends

  end arch_imp;

```