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

entity my_adder3_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
            -- ready to accept an address and associated control signals.
        S_AXI_ARREADY   : out std_logic;
        -- Read data (issued by slave)
        S_AXI_RDATA : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
        -- Read response. This signal indicates the status of the
            -- read transfer.
        S_AXI_RRESP : out std_logic_vector(1 downto 0);
        -- Read valid. This signal indicates that the channel is
            -- signaling the required read data.
        S_AXI_RVALID    : out std_logic;
        -- Read ready. This signal indicates that the master can
            -- accept the read data and response information.
        S_AXI_RREADY    : in std_logic
    );
end my_adder3_v1_0_S00_AXI;

architecture arch_imp of my_adder3_v1_0_S00_AXI is

    -- AXI4LITE signals
    signal axi_awaddr   : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_awready  : std_logic;
    signal axi_wready   : std_logic;
    signal axi_bresp    : std_logic_vector(1 downto 0);
```

```
    signal axi_bvalid  : std_logic;
    signal axi_araddr  : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
    signal axi_arready : std_logic;
    signal axi_rdata   : std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal axi_rresp   : std_logic_vector(1 downto 0);
    signal axi_rvalid  : std_logic;

    -- Example-specific design signals
    -- local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
    -- ADDR_LSB is used for addressing 32/64 bit registers/memories
    -- ADDR_LSB = 2 for 32 bits (n downto 2)
    -- ADDR_LSB = 3 for 64 bits (n downto 3)
    constant ADDR_LSB  : integer := (C_S_AXI_DATA_WIDTH/32)+ 1;
    constant OPT_MEM_ADDR_BITS : integer := 1;
    ------------------------------------------------
    ---- Signals for user logic register space example
    ------------------------------------------------
    ---- Number of Slave Registers 4
    signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal slv_reg_rden : std_logic;
    signal slv_reg_wren : std_logic;
    signal reg_data_out :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
    signal byte_index   : integer;

signal adder_out : std_logic_vector(31 downto 0);

    component adder
    port(
       clk: in std_logic;
       a: in std_logic_VECTOR(15 downto 0);
       b: in std_logic_VECTOR(15 downto 0);
       p: out std_logic_VECTOR(15 downto 0));
    end component;

begin
    -- I/O Connections assignments

    S_AXI_AWREADY   <= axi_awready;
    S_AXI_WREADY    <= axi_wready;
    S_AXI_BRESP <= axi_bresp;
    S_AXI_BVALID    <= axi_bvalid;
    S_AXI_ARREADY   <= axi_arready;
    S_AXI_RDATA <= axi_rdata;
```

```
S_AXI_RRESP <= axi_rresp;
S_AXI_RVALID   <= axi_rvalid;
-- Implement axi_awready generation
-- axi_awready is asserted for one S_AXI_ACLK clock cycle when both
-- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
-- de-asserted when reset is low.

process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_awready <= '0';
    else
      if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') then
        -- slave is ready to accept write address when
        -- there is a valid write address and write data
        -- on the write address and data bus. This design
        -- expects no outstanding transactions.
        axi_awready <= '1';
      else
        axi_awready <= '0';
      end if;
    end if;
  end if;
end process;

-- Implement axi_awaddr latching
-- This process is used to latch the address when both
-- S_AXI_AWVALID and S_AXI_WVALID are valid.

process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_awaddr <= (others => '0');
    else
      if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') then
        -- Write Address latching
        axi_awaddr <= S_AXI_AWADDR;
      end if;
    end if;
  end if;
end process;

-- Implement axi_wready generation
-- axi_wready is asserted for one S_AXI_ACLK clock cycle when both
```

```
    -- S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
    -- de-asserted when reset is low.

    process (S_AXI_ACLK)
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          axi_wready <= '0';
        else
          if (axi_wready = '0' and S_AXI_WVALID = '1' and S_AXI_AWVALID = '1') then
              -- slave is ready to accept write data when
              -- there is a valid write address and write data
              -- on the write address and data bus. This design
              -- expects no outstanding transactions.
              axi_wready <= '1';
          else
            axi_wready <= '0';
          end if;
        end if;
      end if;
    end process;

    -- Implement memory mapped register select and write logic generation
    -- The write data is accepted and written to memory mapped registers when
    -- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
strobes are used to
    -- select byte enables of slave registers while writing.
    -- These registers are cleared when reset (active low) is applied.
    -- Slave register write enable is asserted when valid address and data are
available
    -- and the slave is ready to accept the write address and write data.
    slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID ;

    process (S_AXI_ACLK)
    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
    begin
      if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
          slv_reg0 <= (others => '0');
          slv_reg1 <= (others => '0');
          slv_reg2 <= (others => '0');
          slv_reg3 <= (others => '0');
        else
          loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
          if (slv_reg_wren = '1') then
            case loc_addr is
```

```
            when b"00" =>
              for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                if ( S_AXI_WSTRB(byte_index) = '1' ) then
                  -- Respective byte enables are asserted as per write strobes

                  -- slave registor 0
                  slv_reg0(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                end if;
              end loop;
            when b"01" =>
              for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                if ( S_AXI_WSTRB(byte_index) = '1' ) then
                  -- Respective byte enables are asserted as per write strobes

                  -- slave registor 1
                  slv_reg1(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                end if;
              end loop;
            when b"10" =>
              for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                if ( S_AXI_WSTRB(byte_index) = '1' ) then
                  -- Respective byte enables are asserted as per write strobes

                  -- slave registor 2
                  slv_reg2(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                end if;
              end loop;
            when b"11" =>
              for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
                if ( S_AXI_WSTRB(byte_index) = '1' ) then
                  -- Respective byte enables are asserted as per write strobes

                  -- slave registor 3
                  slv_reg3(byte_index*8+7 downto byte_index*8) <=
S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
                end if;
              end loop;
            when others =>
              slv_reg0 <= slv_reg0;
              slv_reg1 <= slv_reg1;
              slv_reg2 <= slv_reg2;
              slv_reg3 <= slv_reg3;
          end case;
```

```
        end if;
      end if;
    end if;
  end process;

  -- Implement write response logic generation
  -- The write response and response valid signals are asserted by the slave
  -- when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
  -- This marks the acceptance of address and indicates the status of
  -- write transaction.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_bvalid  <= '0';
        axi_bresp   <= "00"; --need to work more on the responses
      else
        if (axi_awready = '1' and S_AXI_AWVALID = '1' and axi_wready = '1' and
S_AXI_WVALID = '1' and axi_bvalid = '0'  ) then
          axi_bvalid <= '1';
          axi_bresp  <= "00";
        elsif (S_AXI_BREADY = '1' and axi_bvalid = '1') then   --check if bready is
asserted while bvalid is high)
          axi_bvalid <= '0';                                 -- (there is a
possibility that bready is always asserted high)
        end if;
      end if;
    end if;
  end process;

  -- Implement axi_arready generation
  -- axi_arready is asserted for one S_AXI_ACLK clock cycle when
  -- S_AXI_ARVALID is asserted. axi_awready is
  -- de-asserted when reset (active low) is asserted.
  -- The read address is also latched when S_AXI_ARVALID is
  -- asserted. axi_araddr is reset to zero on reset assertion.

  process (S_AXI_ACLK)
  begin
    if rising_edge(S_AXI_ACLK) then
      if S_AXI_ARESETN = '0' then
        axi_arready <= '0';
        axi_araddr  <= (others => '1');
      else
        if (axi_arready = '0' and S_AXI_ARVALID = '1') then
```

```
        -- indicates that the slave has acceped the valid read address
        axi_arready <= '1';
        -- Read Address latching
        axi_araddr  <= S_AXI_ARADDR;
      else
        axi_arready <= '0';
      end if;
    end if;
  end if;
end process;

-- Implement axi_arvalid generation
-- axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
-- S_AXI_ARVALID and axi_arready are asserted. The slave registers
-- data are available on the axi_rdata bus at this instance. The
-- assertion of axi_rvalid marks the validity of read data on the
-- bus and axi_rresp indicates the status of read transaction.axi_rvalid
-- is deasserted on reset (active low). axi_rresp and axi_rdata are
-- cleared to zero on reset (active low).
process (S_AXI_ACLK)
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      axi_rvalid <= '0';
      axi_rresp  <= "00";
    else
      if (axi_arready = '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_arready and S_AXI_ARVALID and (not axi_rvalid) ;

process (slv_reg0, adder_out, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN,
slv_reg_rden)
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 <= adder_out;
          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
    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_arready),
            -- 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
        adder_0 : adder
        port map (
            clk => S_AXI_ACLK,
            a => slv_reg0(31 downto 16),
            b => slv_reg0(15 downto 0),
            p => adder_out(15 downto 0));
    -- User logic ends

end arch_imp;
```