8-bit Sequential Multiplier Solution

Learning Goal: A Simple VHDL Design.


1 Solution

1.1 Adder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Add is
  port(
    a : in std_logic_vector(7 downto 0);
    b : in std_logic_vector(7 downto 0);
    s : out std_logic_vector(8 downto 0)
  );
end Add;

architecture synth of Add is
begin
  -- A, B and S must have the same bitwidth
  s <= ('0' & a) + ('0' & b);
end synth;
```

1.2 And1x8

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

entity and1x8 is
  port(
    a : in std_logic;
    b : in std_logic_vector(7 downto 0);
    s : out std_logic_vector(7 downto 0)
  );
end;
```
8-bit Sequential Multiplier Solution

```
end and1x8;

architecture synth of and1x8 is
begin
    -- set 8 bits to 'a':
    s <= (7 downto 0 => a) and b;
end synth;

1.3 Multiplicand

library ieee;
use ieee.std_logic_1164.all;

entity multiplicand is
    port(
        clk : in std_logic;
        load : in std_logic;
        datain : in std_logic_vector(7 downto 0);
        dataout : out std_logic_vector(7 downto 0)
    );
end multiplicand;

architecture synth of multiplicand is
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
            if (load = '1') then
                dataout <= datain;
            end if;
        end if;
    end process;
end synth;

1.4 Multiplier

library ieee;
use ieee.std_logic_1164.all;

entity multiplier is
    port(
        clk : in std_logic;
        load : in std_logic;
        shift_right : in std_logic;
        datain : in std_logic_vector(7 downto 0);
        dataout : out std_logic
    );
end multiplier;
```
architecture synth of multiplier is
    signal data : std_logic_vector(7 downto 0);
beg

    dataout <= data(0);

    process(clk)
    begin
        if (rising_edge(clk)) then
            if (load = '1') then
                data <= datain;
            elsif (shift_right = '1') then
                data <= '0' & data(7 downto 1);
            end if;
        end if;
    end process;
end synth;

1.5 Product

Note that the reset signal is sent by the Controller and is therefore synchronous. It should initialize the internal register of Product in a synchronous way.

library ieee;
use ieee.std_logic_1164.all;

dep entity product is
    port(
        clk : in std_logic;
        reset : in std_logic;
        load : in std_logic;
        shift_right : in std_logic;
        datain : in std_logic_vector(8 downto 0);
        dataout : out std_logic_vector(15 downto 0)
    );
end product;

architecture synth of product is
    signal data : std_logic_vector(16 downto 0);
beg

    dataout <= data(15 downto 0);

    process(clk)
    begin
        if (rising_edge(clk)) then
            if (reset = '1') then
                data <= (others => '0');
            elsif (load = '1') then
                data(16 downto 8) <= datain;
            elsif (shift_right = '1') then
                data <= '0' & data(16 downto 1);
            end if;
        end if;
    end process;
end synth;
end if;
end if;
end process;

end synth;

1.6 Controller

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity controller is
  port(
    -- External signals
    clk : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    done : out std_logic;

    -- Control signals
    load_multiplier : out std_logic;
    shift_multiplier : out std_logic;
    load_multiplicand : out std_logic;
    reset_product : out std_logic;
    load_product : out std_logic;
    shift_product : out std_logic
  );
end controller;

architecture synth of controller is

  -- State Machine type
  type state_type is (S0, S1, S2, S3);
  signal state, nextstate : state_type;

  -- 3 bit counter register
  signal counter, nextcounter : std_logic_vector(2 downto 0);

begin

  -- synchronous process
  -- we update the state and counter registers
  -- at the rising edge of the clk
  process(reset, clk)
  begin
    -- In case of a reset we return to state S0
    if (reset = '1') then
      state <= S0;
      counter <= (others => '0');
    -- We update state and counter values
  end process;

end controller;
```vhdl
elsif (rising_edge(clk)) then
    state <= nextstate;
    counter <= nextcounter;
end if;
end process;

-- asynchronous process
-- Every combinatorial signal is defined here.
-- Each signal must be initialized with a default value or
-- defined for all each cases, otherwise we will get latches.
process(state, counter, start)
begin

    -- default values
    nextstate <= state;
    nextcounter <= counter;
    done <= '0';
    load_multiplier <= '0';
    load_multiplicand <= '0';
    shift_multiplier <= '0';
    reset_product <= '0';
    load_product <= '0';
    shift_product <= '0';

    -- case statement for the state
    case state is
        when S0 =>
            -- We wait for the start signals
            if (start = '1') then
                -- We go to state S1
                nextstate <= S1;
                -- We initialize the loop counter and registers
                nextcounter <= (others => '0');
                load_multiplier <= '1';
                load_multiplicand <= '1';
                reset_product <= '1';
            end if;
        when S1 =>
            -- nextstate always S2
            nextstate <= S2;
            -- counter is incremented
            nextcounter <= counter + 1;
            -- product register loads the Add result
            load_product <= '1';
        when S2 =>
            -- If counter = 0, it means we have counted from 1 to 8 and we go to S3
            -- Otherwise we go back to S1
            if (counter = 0) then
                nextstate <= S3;
            else
                nextstate <= S1;
            end if;
    end case;
end process;
```
8-bit Sequential Multiplier Solution

-- In each cases we shift the multiplier and product registers
shift_multiplier <= '1';
shift_product  <= '1';
when S3 =>
   -- We go back to S0
nextstate <= S0;
   -- Multiplication is finished, done is set
done <= '1';
end case;
end process;
end synth;

1.7 Clock Divider

The clock frequency is 24MHz. The most significant bit of a counter that has a bit-width of n will divide the clock by 2^n. To have something close to 1Hz, we need to divide the clock with a 25-bit counter (0.75Hz).

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity CLK_divider is
   port(
      clk       : in std_logic;
      clk_out   : out std_logic
   );
end CLK_divider;

architecture synth of CLK_divider is
   signal counter : std_logic_vector(24 downto 0) := (others => '0');
begnin
   -- most significant bit for the output
   clk_out <= counter(23);
   -- counter increment on rising edge
   process(clk)
   begin
      if (rising_edge(clk)) then
         counter <= counter + 1;
      end if;
   end process;
end synth;

1.8 Combinational MultCell

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity combinational_multcell is
  port(
    An : in std_logic;
    B  : in std_logic_vector(7 downto 0);
    Z  : in std_logic_vector(7 downto 0);
    Rm : out std_logic_vector(7 downto 0);
    R0 : out std_logic
  );
end combinational_multcell;

architecture combinational of combinational_multcell is
  signal R: std_logic_vector(8 downto 0);
begin
  R <= (('0' & Z) + ('0' & B)) when An = '1' else ('0' & Z);
  Rm <= R(8 downto 1);
  R0 <= R(0);
end combinational;

1.9 Combinational Multiplier

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity combinational_multiplier is
  port(
    A : in std_logic_vector(7 downto 0);
    B : in std_logic_vector(7 downto 0);
    overflow : out std_logic;
    P : out std_logic_vector(7 downto 0)
  );
end combinational_multiplier;

architecture combinational of combinational_multiplier is
  component combinational_multcell
    port(
      An : in std_logic;
      B : in std_logic_vector(7 downto 0);
      Z : in std_logic_vector(7 downto 0);
      Rm : out std_logic_vector(7 downto 0);
      R0 : out std_logic
    );
  end component;
8-bit Sequential Multiplier Solution

```vhdl

  type type_rest is array (7 downto 0) of std_logic_vector(7 downto 0);

  signal R: type_rest;

begin

  multcell_0: combinational_multcell port map (  
      An => A(0),  
      B => B,  
      Z => (others => '0'),  
      R0 => P(0),  
      Rm => R(0)  
    );

  multcell_1: combinational_multcell port map (  
      An => A(1),  
      B => B,  
      Z => R(0),  
      R0 => P(1),  
      Rm => R(1)  
    );

  multcell_2: combinational_multcell port map (  
      An => A(2),  
      B => B,  
      Z => R(1),  
      R0 => P(2),  
      Rm => R(2)  
    );

  multcell_3: combinational_multcell port map (  
      An => A(3),  
      B => B,  
      Z => R(2),  
      R0 => P(3),  
      Rm => R(3)  
    );

  multcell_4: combinational_multcell port map (  
      An => A(4),  
      B => B,  
      Z => R(3),  
      R0 => P(4),  
      Rm => R(4)  
    );

  multcell_5: combinational_multcell port map (  
      An => A(5),  
      B => B,  
      Z => R(4),  
      R0 => P(5),  
      Rm => R(5)  
    );

```
multcell_6: combinational_multcell port map (  
  An => A(6),  
  B => B,  
  Z => R(5),  
  R0 => P(6),  
  Rm => R(6)  
);

multcell_7: combinational_multcell port map (  
  An => A(7),  
  B => B,  
  Z => R(6),  
  R0 => P(7),  
  Rm => R(7)  
);

overflow <= '1' when R(7) /= (7 downto 0 => '0') else '0';
end combinational;