```
--------------------------------------------------------------------------------
-- File         : siso_gen_calc_arch.vhd
-- Description  : "calculator" architecture for SISO
-- Author       : Sabih Gerez, University of Twente
-- Creation date: September 11, 2018
--------------------------------------------------------------------------------
-- $Rev: 1$
-- $Author: gerezsh$
-- $Date: Sat Sep 17 16:32:33 CEST 2022$
-- $Log$
--------------------------------------------------------------------------------


-- this architecture needs arithmetic functions
library ieee;
use ieee.numeric_std.all;

architecture calc_own of siso_gen is
  -- state for ALU
  -- two "read opcode (opc)" states, the first does not raise "ready"
  type state is (read_opc_init, read_opc_ready1, read_opc_ready2,
                 read_left1,    read_left2,      read_right);
  signal cur_state: state;
  signal nxt_state: state;

  -- internal registers for left and right ALU input and ALU
  -- output, all having the same width
  signal left_in_add_reg:  signed(word_length-1 downto 0);
  signal right_in_add_reg: signed(word_length-1 downto 0);
  signal left_in_mul_reg:  signed(word_length-1 downto 0);
  signal right_in_mul_reg: signed(word_length-1 downto 0);
  signal result_reg:   signed(2*word_length-1 downto 0);
  -- at most 16 different operations are supported by this ALU
  signal opcode_reg:   std_logic_vector(3 downto 0);

  -- and their next values
  signal left_in_add_nxt:  signed(word_length-1 downto 0);
  signal right_in_add_nxt: signed(word_length-1 downto 0);
  signal left_in_mul_nxt:  signed(word_length-1 downto 0);
  signal right_in_mul_nxt: signed(word_length-1 downto 0);
  signal result_nxt:   signed(2*word_length-1 downto 0);
  signal opcode_nxt:   std_logic_vector(3 downto 0);

  -- output of adder
  signal adder_out: signed(word_length-1 downto 0);

  -- output of multiplier
  signal mult_out: signed(2*word_length-1 downto 0);

  -- next value for ready
  signal ready_nxt: std_logic;

begin
  -- the next process is sequential and only sensitive to clk and reset
  seq: process(clk, reset)
  begin
    if (reset = '1')
    then
      left_in_add_reg  <= (others => '0');
      right_in_add_reg <= (others => '0');
      left_in_mul_reg  <= (others => '0');
      right_in_mul_reg <= (others => '0');
      opcode_reg   <= (others => '0');
      result_reg   <= (others => '0');
      ready        <= '0';
      cur_state    <= read_opc_init;
    elsif rising_edge(clk)
    then
      left_in_add_reg  <= left_in_add_nxt;
      right_in_add_reg <= right_in_add_nxt;
      left_in_mul_reg  <= left_in_mul_nxt;
      right_in_mul_reg <= right_in_mul_nxt;
      opcode_reg   <= opcode_nxt;
      result_reg   <= result_nxt;
      ready        <= ready_nxt;
      cur_state    <= nxt_state;
    end if;
  end process seq;

  -- combinational next-value process
  nxt: process(data_in, cur_state, left_in_add_reg, right_in_add_reg, left_in_mul_reg, right
_in_mul_reg,
               opcode_reg, adder_out, mult_out)
  begin
    case cur_state is
      when read_opc_init =>
        nxt_state    <= read_left1;
        left_in_add_nxt  <= left_in_add_reg;
        right_in_add_nxt <= right_in_add_reg;
  left_in_mul_nxt  <= left_in_mul_reg;
        right_in_mul_nxt <= right_in_mul_reg;
        opcode_nxt   <= data_in(3 downto 0);
        result_nxt   <= result_reg;
        ready_nxt    <= '0';
      when read_left1 =>
        nxt_state    <= read_right;
  case opcode_reg is
          when "0000" => -- the null result
            left_in_add_nxt  <= left_in_add_reg;
            right_in_add_nxt <= right_in_add_reg;
      left_in_mul_nxt  <= left_in_mul_reg;
            right_in_mul_nxt <= right_in_mul_reg;
          when "0001" => -- addition
            left_in_add_nxt  <= signed(data_in);
            right_in_add_nxt <= right_in_add_reg;
      left_in_mul_nxt  <= left_in_mul_reg;
            right_in_mul_nxt <= right_in_mul_reg;
          when "0010" => -- multiplication
            left_in_add_nxt  <= left_in_add_reg;
            right_in_add_nxt <= right_in_add_reg;
      left_in_mul_nxt  <= signed(data_in);
            right_in_mul_nxt <= right_in_mul_reg;
          when others => -- non-implemented codes behave like null command
            left_in_add_nxt  <= left_in_add_reg;
            right_in_add_nxt <= right_in_add_reg;
      left_in_mul_nxt  <= left_in_mul_reg;
            right_in_mul_nxt <= right_in_mul_reg;
        end case;
        opcode_nxt   <= opcode_reg;
        result_nxt   <= result_reg;
        ready_nxt    <= '0';
      -- read_left2 is identical to read_left1 except for ready_nxt
```

```
    when read_left2 =>
      nxt_state    <= read_right;
case opcode_reg is
        when "0000" => -- the null result
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
        when "0001" => -- addition
          left_in_add_nxt  <= signed(data_in);
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
        when "0010" => -- multiplication
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= signed(data_in);
          right_in_mul_nxt <= right_in_mul_reg;
        when others => -- non-implemented codes behave like null command
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
      end case;
      opcode_nxt   <= opcode_reg;
      result_nxt   <= result_reg;
      ready_nxt    <= '1';
    when read_right =>
      -- multiplication needs two cycles to output result
case opcode_reg is
        when "0000" => -- the null result
         nxt_state    <= read_opc_ready1;
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
        when "0001" => -- addition
         nxt_state    <= read_opc_ready1;
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= signed(data_in);
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
        when "0010" => -- multiplication
    nxt_state    <= read_opc_ready2;
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= signed(data_in);
          result_nxt <= mult_out;
        when others => -- non-implemented codes behave like null command
         nxt_state    <= read_opc_ready1;
          left_in_add_nxt  <= left_in_add_reg;
          right_in_add_nxt <= right_in_add_reg;
    left_in_mul_nxt  <= left_in_mul_reg;
          right_in_mul_nxt <= right_in_mul_reg;
      end case;
      opcode_nxt   <= opcode_reg;
      result_nxt   <= result_reg;
      ready_nxt    <= '0';
    when read_opc_ready1 | read_opc_ready2 =>
      -- multiplication needs two cycles to output result
        if (cur_state = read_opc_ready2)
        then
           nxt_state    <= read_left2;
        else
           nxt_state    <= read_left1;
        end if;
            left_in_add_nxt  <= left_in_add_reg;
            right_in_add_nxt <= right_in_add_reg;
      left_in_mul_nxt  <= left_in_mul_reg;
            right_in_mul_nxt <= right_in_mul_reg;
        case opcode_reg is
          when "0000" => -- the null result
            result_nxt <= (others => '0');
          when "0001" => -- addition
            result_nxt(word_length - 1 downto 0)             <= adder_out;
            result_nxt(2*word_length - 1 downto word_length) <= (others => '0');
          when "0010" => -- multiplication
            result_nxt <= mult_out;
          when others => -- non-implemented codes behave like null command
            result_nxt <= (others => '0');
        end case;
        opcode_nxt   <= data_in(3 downto 0);
        ready_nxt    <= '1';
    end case;
  end process nxt;

  -- adder, wrap around in case of overflow, so discard carry
  adder_out <= left_in_add_reg + right_in_add_reg;

  -- multiplier
  mult_out <= left_in_mul_reg * right_in_mul_reg;

  -- output register is lowest half of result_reg, except when second
  -- part of multiplication result needs to be output
  data_out <= std_logic_vector(result_reg(2*word_length - 1 downto word_length))
              when cur_state = read_left2
            else std_logic_vector(result_reg(word_length - 1 downto 0));

  -- this block should receive data in every clock cycle
  req <= '1';
end calc_own;
```