Memories Solution

Learning Goal: Memories in VHDL and on FPGA4U.

Requirements: Quartus II Web Edition and ModelSim-Altera.

1 32-bit Register File

1.1 Solution

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

entity register_file is
  port(
    clk    : in std_logic;
    aa     : in  std_logic_vector( 4 downto 0);
    ab     : in  std_logic_vector( 4 downto 0);
    aw     : in  std_logic_vector( 4 downto 0);
    wren   : in  std_logic;
    wrdata : in  std_logic_vector(31 downto 0);
    a      : out std_logic_vector(31 downto 0);
    b      : out std_logic_vector(31 downto 0)
  );
end register_file;

architecture synth of register_file is
  type reg_type is array (0 to 31) of std_logic_vector(31 downto 0);
  signal reg_array : reg_type := (others=>(others=>'0')); --intialized to 0
begin
  -- asynchronous read
  a <= reg_array(conv_integer(aa));
  b <= reg_array(conv_integer(ab));
  -- synchronous write
  process(clk)
  begin
    if(rising_edge(clk))then
      if(wren='1')then
        reg_array(conv_integer(aw)) <= wrdata;
      end if;
      -- This fixes reg_array(0) to 0 without any extra resource.
      reg_array(0) <= (others => '0');
    end if;
  end process;
end synth;
```

2 Synchronous memories

2.1 Decoder solution

We propose you here two implementations of the Decoder. The first one is simple to read, it consists of sequences of comparisons. The second one will slightly reduce the critical path of the circuit, since the comparisons are performed in parallel.

2.1.1 Simple Solution

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

entity decoder is
  port(
    address : in  std_logic_vector(15 downto 0);
    cs_LEDs  : out std_logic;
    cs_RAM  : out std_logic;
    cs_ROM  : out std_logic
    );
end decoder;

architecture synth of decoder is
begin
  -- unit selection
  process (address)
  begin
    cs_ROM <= '0';
    cs_RAM <= '0';
    cs_LEDs <= '0';
    if (address < X"1000") then
      cs_ROM <= '1';
    elsif (address < X"2000") then
      cs_RAM <= '1';
    elsif (address < X"2010") then
      cs_LEDs <= '1';
    end if;
  end process;
end synth;
```

2.1.2 Faster Solution

```
architecture synth of decoder is
begin
  -- unit selection
  cs_ROM <= '1' when address(15 downto 12) = "0000" else '0';
  cs_RAM <= '1' when address(15 downto 12) = "0001" else '0';
  cs_LEDs <= '1' when address(15 downto 4) = "001000000000" else '0';
end synth;
```

2.2 RAM solution

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

entity RAM is
  port(
    clk     : in  std_logic;
    cs      : in  std_logic;
    read    : in  std_logic;
    write   : in  std_logic;
    address : in  std_logic_vector(9 downto 0);
    wrdata  : in  std_logic_vector(31 downto 0);
    rddata  : out std_logic_vector(31 downto 0));
end RAM;

architecture synth of RAM is
  type mem_type is array(0 to 1023) of std_logic_vector(31 downto 0);
  signal mem         : mem_type;
  signal reg_address : std_logic_vector(9 downto 0);
  signal reg_read    : std_logic;
begin
  -- address register
  process (clk)
  begin
    if (rising_edge(clk)) then
      reg_read    <= cs and read;
      reg_address <= address;
    end if;
  end process;

  -- read in memory
  process (mem, reg_read, reg_address)
  begin
    rddata <= (others => 'Z');

    if (reg_read ='1') then
      rddata <= mem(conv_integer(reg_address));
    end if;
  end process;

  -- write in memory
  process (clk)
  begin
    if (rising_edge(clk)) then
      if (cs = '1' and write = '1') then
        mem(conv_integer(address)) <= wrdata;
      end if;
    end if;
  end process;
end synth;
```

2.3 ROM solution

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

entity ROM is
  port(
    clk     : in  std_logic;
    cs      : in  std_logic;
    read    : in  std_logic;
    address : in  std_logic_vector(9 downto 0);
    rddata  : out std_logic_vector(31 downto 0));
end ROM;

architecture synth of ROM is
  component ROM_Block is
    port(
      address : in  std_logic_vector(9 downto 0);
      clock   : in  std_logic;
      q       : out std_logic_vector(31 downto 0)
    );
  end component;

  -- internal signal for the ROM rddata
  signal in_rddata : std_logic_vector(31 downto 0);
  signal reg_read  : std_logic;

begin
  rom_block_0: ROM_Block port map(
    address => address,
    clock => clk,
    q => in_rddata);

  -- 1 cycle latency
  process(clk)
  begin
    if(rising_edge(clk))then
      reg_read <= read and cs;
    end if;
  end process;

  -- read in memory
  process(reg_read, in_rddata)
  begin
    rddata <= (others => 'Z');

    if(reg_read='1')then
      rddata <= in_rddata;
    end if;
  end process;
end synth;
```

2.4 LEDs solution

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

entity LEDs is
  port(
    -- bus interface
    clk       : in  std_logic;
    reset_n   : in  std_logic;
    cs        : in  std_logic;
    read      : in  std_logic;
    write     : in  std_logic;
    address   : in  std_logic_vector( 1 downto 0);
    rddata    : out std_logic_vector(31 downto 0);
    wrdata    : in  std_logic_vector(31 downto 0);

    -- external output
    LEDs      : out std_logic_vector(95 downto 0)
  );
end LEDs;

architecture synth of LEDs is
  constant REG_LED_0_31    : std_logic_vector(1 downto 0)  := "00";
  constant REG_LED_32_63   : std_logic_vector(1 downto 0)  := "01";
  constant REG_LED_64_95   : std_logic_vector(1 downto 0)  := "10";
  constant REG_DUTY_CYCLE  : std_logic_vector(1 downto 0)  := "11";

  signal reg_read     : std_logic;
  signal reg_address  : std_logic_vector( 1 downto 0);
  signal counter      : std_logic_vector( 7 downto 0);
  signal LEDs_reg     : std_logic_vector(95 downto 0);
  signal duty_cycle   : std_logic_vector( 7 downto 0);
begin

  LEDs <= LEDs_reg when counter < duty_cycle
          else (others => '0');

  -- registers
  process (clk, reset_n)
  begin
    if (reset_n='0') then
      reg_read     <= '0';
      reg_address  <= (others => '0');
      counter      <= (others => '0');

    elsif (rising_edge(clk)) then
      reg_read     <= cs and read;
      reg_address  <= address;

      if address /= REG_DUTY_CYCLE then
```

```
        counter   <= counter + 1;
      else
        counter   <= (others => '0');
      end if;
    end if;
  end process;

  -- read
  process (reg_read, reg_address, LEDs_reg, duty_cycle)
  begin
    rddata <= (others => 'Z');
    if (reg_read = '1') then
      rddata <= (others => '0');
      case reg_address is
        when REG_LED_0_31    => rddata <= LEDs_reg(31 downto  0);
        when REG_LED_32_63   => rddata <= LEDs_reg(63 downto 32);
        when REG_LED_64_95   => rddata <= LEDs_reg(95 downto 64);
        when REG_DUTY_CYCLE => rddata(7 downto 0) <= duty_cycle;
        when others =>
      end case;
    end if;
  end process;

  -- write
  process (clk, reset_n)
  begin
    if (reset_n='0') then
      LEDs_reg <= (others => '0');
      duty_cycle <= X"0F";
    elsif (rising_edge(clk)) then
      if (cs = '1' and write = '1') then
        case address is
          when REG_LED_0_31    => LEDs_reg(31 downto  0) <= wrdata;
          when REG_LED_32_63   => LEDs_reg(63 downto 32) <= wrdata;
          when REG_LED_64_95   => LEDs_reg(95 downto 64) <= wrdata;
          when REG_DUTY_CYCLE => duty_cycle <= wrdata(7 downto 0);
          when others => null;
        end case;
      end if;
    end if;
  end process;
end synth;
```

3 Sequential Design with Memories

3.1 Solution

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

entity controller is
```

```
  port(
    clk     : in  std_logic;
    reset_n : in  std_logic;
    read    : out std_logic;
    write   : out std_logic;
    address : out std_logic_vector(15 downto 0);
    rddata  : in  std_logic_vector(31 downto 0);
    wrdata  : out std_logic_vector(31 downto 0)
  );
end controller;

architecture synth of controller is
  type state_type is (S0, S1, S2, S3, S4, S5);
  signal state, nextstate : state_type;

  signal rdaddress, nextrdaddress : std_logic_vector(15 downto 0);
  signal wraddress, nextwraddress : std_logic_vector(15 downto 0);
  signal ROMaddr  , nextROMaddr   : std_logic_vector(15 downto 0);
  signal length   , nextlength    : std_logic_vector(15 downto 0);
begin

  process(state, rdaddress, wraddress, ROMaddr, length, rddata)
  begin
    nextstate     <= state;
    nextrdaddress <= rdaddress;
    nextwraddress <= wraddress;
    nextROMaddr   <= ROMaddr;
    nextlength    <= length;

    -- read in address ROMaddr by default
    address <= ROMaddr;
    read    <= '0';
    write   <= '0';
    -- wrdata is always equal to rddata
    wrdata  <= rddata;

    case state is
      -- read in the ROM and increment ROMaddr
      when S0 =>
        nextstate <= S1;
        read      <= '1';
        nextROMaddr <= ROMaddr + 4;
      -- read in the ROM and increment ROMaddr
      -- store the read value in the length register
      when S1 =>
        -- if length is 0, End of the program.
        if(rddata(15 downto 0) = 0)then
          nextstate <= S5;
        else
          nextstate <= S2;
        end if;
        read    <= '1';
        nextROMaddr <= ROMaddr + 4;
        nextlength  <= rddata(15 downto 0);
      -- store the read value in the rdaddress and wraddress registers
      when S2 =>
        nextstate <= S3;
        nextrdaddress <= rddata(31 downto 16);
        nextwraddress <= rddata(15 downto 0);
      -- read the rdaddress address
      when S3 =>
        -- if length is zero, return to S0
        if(length=0)then
          nextstate <= S0;
        else
          nextstate <= S4;
        end if;
        nextrdaddress <= rdaddress + 4;
        nextlength    <= length - 1;
        read          <= '1';
        address       <= rdaddress;
      -- write the wraddress address
      when S4 =>
        write <= '1';
        nextstate <= S3;
        address <= wraddress;
        nextwraddress <= wraddress + 4;
      -- dead end
      when S5 =>
        nextstate <= S5;

      when others =>
        nextstate <= S0;
      end case;
  end process;

  process(reset_n, clk)
  begin
    if(reset_n='0')then
      state   <= S0;
      ROMaddr <= (others =>'0');
    elsif(rising_edge(clk))then
      state     <= nextstate;
      ROMaddr   <= nextROMaddr;
      rdaddress <= nextrdaddress;
      wraddress <= nextwraddress;
      length    <= nextlength;
    end if;
  end process;
end synth;
```