ALU Solution

Learning Goal: Testbench, arithmetic operations

Requirements: Quartus II Web Edition and ModelSim.

1 Solution

You will find in the following subsections, a VHDL solution for each unit of the ALU.

1.1 Logic Unit

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

entity logic_unit is
  port(
    a  : in  std_logic_vector(31 downto 0);
    b  : in  std_logic_vector(31 downto 0);
    op : in  std_logic_vector( 1 downto 0);
    r  : out std_logic_vector(31 downto 0)
  );
end logic_unit;

architecture synth of logic_unit is
begin

process(a, b, op)
begin
  case op is
    when "00" => r <= a nor b;
    when "01" => r <= a and b;
    when "10" => r <= a or b;
    when "11" => r <= a xor b;
    when others => null;
  end case;
end process;

end synth;
```

1.2 Add/Sub Unit

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

entity add_sub is
  port(
    a        : in  std_logic_vector(31 downto 0);
    b        : in  std_logic_vector(31 downto 0);
    sub_mode : in  std_logic;
    carry    : out std_logic;
    zero     : out std_logic;
    r        : out std_logic_vector(31 downto 0)
  );
end add_sub;

architecture synth of add_sub is
  signal b_s    : std_logic_vector(31 downto 0);
  signal result : std_logic_vector(32 downto 0);
begin

  -- b_s is the output of the xor between b and sub_mode.
  -- here a vector of 32 bits of sub_mode has been created
  b_s <= b xor (31 downto 0 => sub_mode);

  -- The result of the adder will be on 33 bits to keep the carry.
  result <= ('0' & a) + ('0' & b_s) + sub_mode;

  -- The carry is extracted from the most significant bit of the result
  carry <= result(32);

  -- the r output is the 32 least significant bits of result
  r <= result(31 downto 0);

  -- zero is 1 when r=0. As r is an output, it can't be read.
  -- Therefore we compare the 32 least significant bits of result
  zero  <= '1' when result(31 downto 0)=0 else '0';
end synth;
```

1.3 Comparator

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

entity comparator is
  port (
    a_31    : in  std_logic;
    b_31    : in  std_logic;
    diff_31 : in  std_logic;
```

```
    carry   : in  std_logic;
    zero    : in  std_logic;
    op      : in  std_logic_vector( 2 downto 0);
    r       : out std_logic
  );
end comparator;

architecture synth of comparator is

begin

process (op, zero, carry, a_31, b_31, diff_31)
begin
  -- By default, r will be equal comparison
  r <= zero;
  case op is
    -- >=
    when "001" =>
      -- take care to surround all logical operation with parenthesis:
      -- there's no priority between them.
      r <= (not a_31 and b_31) or ((not a_31 xor b_31) and not diff_31);
    -- <
    when "010"  =>
      r <= (a_31 and not b_31) or ((not a_31 xor b_31) and diff_31);
    -- !=
    when "011" => r <= not zero;
    -- == (optional as it is the default result)
    when "100" => r <= zero;
    -- >= unsigned
    when "101" => r <= carry;
    -- < unsigned
    when "110" => r <= not carry;
    when others =>
  end case;
end process;
end synth;
```

1.4 Multiplexer

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

entity multiplexer is
  port(
    i0  : in  std_logic_vector(31 downto 0);
    i1  : in  std_logic_vector(31 downto 0);
    i2  : in  std_logic_vector(31 downto 0);
    i3  : in  std_logic_vector(31 downto 0);
    sel : in  std_logic_vector( 1 downto 0);
    o   : out std_logic_vector(31 downto 0)
  );
```

```
end multiplexer;

architecture synth of multiplexer is
begin

process(i0, i1, i2, i3, sel)
begin
  case sel is
    when "00" => o <= i0;
    when "01" => o <= i1;
    when "10" => o <= i2;
    when "11" => o <= i3;
    when others =>
  end case;
end process;

end synth;
```

1.5 Shift/Rotate Unit

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

entity shift_unit is
  port(
    a  : in  std_logic_vector(31 downto 0);
    b  : in  std_logic_vector( 4 downto 0);
    op : in  std_logic_vector( 2 downto 0);
    r  : out std_logic_vector(31 downto 0)
  );
end shift_unit;

architecture synth of shift_unit is
  signal rotate, shift_left, shift_right : std_logic_vector(31 downto 0);
begin

-- selection between the operations
sel: process(op, rotate, shift_left, shift_right)
begin
  case op(1 downto 0) is
    when "00" | "01" => r <= rotate;
    when "10" => r <= shift_left;
    when "11" => r <= shift_right;
    when others =>
  end case;
end process;

-- rotate left or right
ror_rol: process(a, b, op)
  variable b_s : std_logic_vector( 4 downto 0);
  variable v   : std_logic_vector(31 downto 0);
```

```
begin
  -- we invert b if we want to rotate to the right:
  -- (a rol b <=> a ror (-b))
  -- When we rotate to the right op(0)='1'
  -- conditional invertion of b:
  b_s := (b xor (4 downto 0 => op(0))) + op(0);
  v   := a;
  for i in 0 to 4 loop
    if(b_s(i)='1')then
      v := v(31-2**i downto 0) & v(31 downto 32-2**i);
    end if;
  end loop;
  rotate <= v;
end process;

-- shift_right
srl_sra: process(a, b, op)
  variable sign : std_logic;
  variable v : std_logic_vector(31 downto 0);
begin
  -- if op(2)='1' we have to replicate the sign of the operand a.
  sign := op(2) and a(31);
  v    := a;
  for i in 0 to 4 loop
    if(b(i)='1')then
      v := ((2**i)-1 downto 0 => sign) & v(31 downto 2**i);
    end if;
  end loop;
  shift_right<= v;
end process;

-- shift_left
sh_left: process(a, b)
  variable v : std_logic_vector(31 downto 0);
begin
  v := a;
  for i in 0 to 4 loop
    if(b(i)='1')then
      v := v(31-2**i downto 0) & ((2**i)-1 downto 0 => '0');
    end if;
  end loop;
  shift_left<= v;
end process;

end synth;
```