```
library IEEE;
use IEEE.std_logic_1164.ALL ;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
-- use IEEE.std_logic_arith.all;
-- ver 2009_0311 v.fristot : Phelma Preorientation SEI

library lib_nanoproc;
USE lib_nanoproc.NANO_PKG.all;


entity PROC IS
port (      din                         : in std_logic_vector(len_data_bus-1 downto 0);
                  resetn, clk           : in std_logic;
                  write                    : out std_logic;
                  Rd_out                   : out std_logic_vector(len_data_bus-1 downto 0);
                  Rad_out                  : out std_logic_vector(len_addr_bus-1 downto 0));
end PROC;

architecture behavior of PROC is

component DEC3TO8
port (      w   : in std_logic_vector(2 downto 0);
                  y   : out std_logic_vector(0 to 7));
end component;

component ALU
port (      a,b            : in std_logic_vector(len_data_bus-1 downto 0);
                  alu_code    : in std_logic_vector(1 downto 0);
                  r              : out std_logic_vector(len_data_bus-1 downto 0));
end component;

type STATE is (fetch1,fetch2,exe_1,exe_2,exe_3);        -- Etats du processeur
type INST is array (0 to 13) of string(1 to 4);         -- liste jeu instructions


signal p_state,p_next_state : STATE;
-- pragma synthesis_off
signal cur_inst     : string(1 to 4);
-- pragma synthesis_on
signal alu_code     : std_logic_vector(1 downto 0);     -- choix code op alu
signal incr_pc,br_pc : std_logic;
signal offset_PC    : std_logic_vector(len_data_bus-1 downto 0); -- offset en cas de branch
signal res_alu      : std_logic_vector(len_data_bus-1 downto 0); -- resultat alu
signal Z_bit,G_bit : std_logic;                                  -- drapeaux resultat alu

-- definition registres
-- selections
signal R_ld             : std_logic_vector(0 to 7);             -- selection registres R()
signal Ra_ld,Rg_ld,Ri_ld : std_logic;
signal Rd_ld,Rad_ld : std_logic;

-- sorties registres
signal R_q,R_d      : REGISTER_FILE(0 to 7);
signal Ra_q,Ra_d    : std_logic_vector(len_data_bus-1 downto 0);
signal Rg_q,Rg_d    : std_logic_vector(len_data_bus-1 downto 0);
signal Ri_q,Ri_d    : std_logic_vector(9 downto 0);     -- registre instruction 10 bits
```

```
signal Rd_q,Rd_d    : std_logic_vector(len_data_bus-1 downto 0);
signal Rad_q,Rad_d  : std_logic_vector(len_addr_bus-1 downto 0);
signal Rw_q,Rw_d      : std_logic;                  -- registre simple pour signal write externe
signal Xreg,Yreg    : std_logic_vector(0 to 7);               -- selection registres

signal mux_sel  : std_logic_vector(3 downto 0); -- selection registres R(), A, G, I dans mux

signal I              : std_logic_vector(3 downto 0); -- instruction
signal rx,ry        : std_logic_vector(2 downto 0); -- champs Rx et Ry de l'instruction

signal nanobus  : std_logic_vector(len_data_bus-1 downto 0); -- bus data interne

-- constant zero        : natural :=0; -- test de nullite de G

constant i_mv       : std_logic_vector(3 downto 0) := "0000"; -- liste des codes instructions
constant i_ldi      : std_logic_vector(3 downto 0) := "0001";
constant i_add      : std_logic_vector(3 downto 0) := "0010";
constant i_sub      : std_logic_vector(3 downto 0) := "0011";
constant i_ld       : std_logic_vector(3 downto 0) := "0100";
constant i_st       : std_logic_vector(3 downto 0) := "0101";
constant i_mvnz     : std_logic_vector(3 downto 0) := "0110";
constant i_mvgt     : std_logic_vector(3 downto 0) := "0111";
constant i_and      : std_logic_vector(3 downto 0) := "1000";
constant i_bra      : std_logic_vector(3 downto 0) := "1001";
constant i_brnz     : std_logic_vector(3 downto 0) := "1010";
constant i_brgt     : std_logic_vector(3 downto 0) := "1011";
constant i_brz      : std_logic_vector(3 downto 0) := "1100";
constant i_brmi     : std_logic_vector(3 downto 0) := "1101";

constant i_name : INST := ("mv  ","ldi ","add ","sub ","ld  ","st  ","mvnz","mvgt","and ",
"bra ","brnz","brgt","brz ","brmi");

-- Table de constantes pour le multiplexeur
-- codes de 0000 à 0111 : nanobus <= R_q() (PC = 0111)
-- code         1000                : nanobus <= din
-- code         1001                : nanobus <= Rg
-- code              autres                 : nanobus <= 0

constant MUX_off        : std_logic_vector(3 downto 0) := "1111";
constant MUX_din        : std_logic_vector(3 downto 0) := "1000";
constant MUX_Rg         : std_logic_vector(3 downto 0) := "1001";
constant MUX_PC         : std_logic_vector(3 downto 0) := "0111";

BEGIN

-- mapping

decx: DEC3TO8   port map (rx,xreg); -- decodage pour chargement registres R_ld()
decy: DEC3TO8   port map (ry,yreg);
c_ALU : ALU port map (Ra_q,nanobus,alu_code,res_alu);

-- process synchrone de mise a jour des registres et fsm sequenceur
--
process (clk)
BEGIN
        if (clk'event and clk = '1') then
            if resetn='0' then  -- raz general
```

```
                Rw_q    <='0';
                Ra_q    <=(others=>'0');
                for i in 0 to 7 loop
                    R_q(i)<=(others=>'0');
                end loop;
                Rd_q    <=(others=>'0');
                Rad_q   <=(others=>'0');
                Ri_q    <=(others=>'0');
                Rg_q    <=(others=>'0');
                p_state     <=fetch1;
            else
                p_state     <=p_next_state;
                R_q     <=R_d;   -- prochain etat banc registre
                Ra_q    <=Ra_d;
                Rd_q    <=Rd_d;
                Rad_q   <=Rad_d;
                Ri_q    <=Ri_d;
                Rg_q    <=Rg_d;
                Rw_q    <= Rw_d;
            end if;
        end if;
END Process;

-- Process combinatoire registres du nano_proc
-- connexion des registres avec entrees de validation

reg_p : Process(DIN,nanobus,res_alu,incr_pc,R_ld,R_q,Ra_ld,Ra_q,Rw_q,br_pc,offset_pc,
        Rd_ld,Rd_q, Ri_ld,Ri_q, Rad_ld,Rad_q,Rg_ld,Rg_q)
Begin

    -- fonction banc de registres et autres registres
    for i in 0 to 7 loop
        R_d(i)<=R_q(i);          -- maintien par defaut
        if (R_ld(i)='1') then R_d(i)<=nanobus; end if;  -- charge nanobus si load actif
    end loop;
    if incr_pc='1' then R_d(7)<=R_q(7)+1; end if;    -- compteur PC
    if br_pc='1' then R_d(7)<=R_q(7)+offset_PC; end if;      -- compteur PC+offset de branch

    Ra_d        <= Ra_q;
    if Ra_ld ='1' then  Ra_d    <= nanobus; end if;
    Rd_d    <= Rd_q;
    if Rd_ld='1' then   Rd_d  <= nanobus; end if;
    Rad_d   <= Rad_q;
    if Rad_ld='1' then  Rad_d <= nanobus(len_addr_bus-1 downto 0); end if;
    Ri_d        <= Ri_q;
    if Ri_ld='1' then   Ri_d    <= din(9 downto 0); end if;
    Rg_d        <= Rg_q;
    if Rg_ld='1' then   Rg_d    <= res_alu; end if;

    Rd_out  <= Rd_q;
    Rad_out <= Rad_q;
    I           <= Ri_q(9 downto 6);
    write   <= Rw_q;
end process reg_p;


-- Process combinatoire : multiplexeur du nano_proc
```

```
-- Entrees du MUX : din, banc de registres R_q, Rg
-- Mux positionne nanobus, selection par signal mux_sel (voir constantes)

mux_p : Process (din, R_q, Rg_q, mux_sel)
begin
    if (mux_sel(3)='1') then -- Si n'est pas selection banc de registres
        case mux_sel is
            when MUX_din => nanobus<=din;
            when MUX_Rg => nanobus<=Rg_q;
      when others => nanobus <= (others=>'0');
    end case;
  else                                          -- Si selection banc de registres
    nanobus<=R_q(conv_integer(mux_sel(3 downto 0)) );
 end if;
end process mux_p;



-- Process combinatoire : drapeaux Z et G
-- Z_bit=1 si Rg_q = 0
-- G_bit = Rg_q(poids fort) c'est le bit de signe de Rg

flags_p : Process (Rg_q)
begin
    if (unsigned(Rg_q)/=0) then
        Z_bit<='0';
  else Z_bit<='1';
    end if;
    G_bit<=Rg_q(len_data_bus-1);

end process flags_p;

--------------------------------------------------------------------------------
-- Process combinatoire : Sequenceur
--

controlsignals: PROCESS (p_state, I, Ri_q, rx, ry,Rw_d,Z_bit,G_bit,alu_code)
BEGIN

-- pragma synthesis_off
    cur_inst <=i_name(conv_integer(I)); -- recupere instruction enum
-- pragma synthesis_on

    Ra_ld<='0';                 -- tous registres deselectionnes
    Rg_ld<='0';
    Ri_ld<='0';
    R_ld<=(others=>'0');
    mux_sel<=MUX_off;       -- pas de selection

    alu_code<=alu_add;  -- addition par defaut
    Rw_d <='0';                 -- pas d'ecriture
    incr_PC <='0';          -- pas d'increment
    br_PC   <='0';              -- pas de branch, offset sur PC
                                -- Separation des champs de l'instruction de Ri
    rx<=Ri_q(5 downto 3);
    ry<=Ri_q(2 downto 0);
    offset_PC(len_data_bus-1 downto 6) <= (others=>rx(2)); -- recopie signe de l'offset
    offset_PC(5 downto 0) <= rx & ry;           -- 6 bits de l'offset PC pour branch
```

```
    Rad_ld <='0';
    Rd_ld <='0';

-- Machine a etats :

    CASE p_state is
    when fetch1 =>
        mux_sel<=MUX_PC;      -- selection registre PC dans nanobus
        Rad_ld <= '1';        -- sortie PC pour instr fetch1
        incr_PC<='1';             -- incremente PC
        p_next_state <= fetch2;

    when fetch2 =>
        Ri_ld <= '1';             -- store din in Ri
        p_next_state <= exe_1;

    when exe_1 =>
        mux_sel<='0'& ry;     -- Ry dans nanobus
        p_next_state <= exe_2;

        case I is
        when i_mv =>
            R_ld<=Xreg;           -- selection registre destination
            p_next_state <= fetch1; -- fini
        when i_mvnz =>
            if (Z_bit='0') then
                R_ld<=Xreg;       -- selection registre destination
            end if;
            p_next_state <= fetch1;
        when i_mvgt =>
            if (G_bit='0') then
                R_ld<=Xreg;       -- selection registre destination
            end if;
            p_next_state <= fetch1;
        when i_ldi =>
            mux_sel<=MUX_PC; -- selection registre PC dans nanobus
            Rad_ld<='1';          -- selection nanobus dans address ROM
            incr_PC<='1';         -- avance PC pour data imm
        when i_add =>
            Ra_ld<='1';       -- ecriture dans accu A
        when i_sub =>
            Ra_ld<='1';       -- ecriture dans accu A
        when i_and =>
            Ra_ld<='1';       -- ecriture dans accu A
        when i_ld =>
            Rad_ld<='1';      -- ecriture dans adresse pour memoire externe
        when i_st =>
            Rad_ld<='1';      -- ecriture dans adresse pour memoire externe
        when i_bra =>
            br_pc<='1';       -- valide calcul PC + offset_PC
            p_next_state <= fetch1;
        when i_brnz =>
            if (Z_bit='0') then
                br_pc<='1';       -- valide calcul PC + offset_PC
            end if;
            p_next_state <= fetch1;
        when i_brgt =>
```

```
            if (G_bit='0') then
                br_pc<='1';      -- valide calcul PC + offset_PC
            end if;
            p_next_state <= fetch1;
        when i_brz =>
            if (Z_bit='1') then
                br_pc<='1';      -- valide calcul PC + offset_PC
            end if;
            p_next_state <= fetch1;
        when i_brmi =>
            if (G_bit='1') then -- N=1 pour branch minus
                br_pc<='1';      -- valide calcul PC + offset_PC
            end if;
            p_next_state <= fetch1;
        when others =>
        end case;

    when exe_2 =>
        p_next_state <= fetch1;
        case I is
        when i_ldi =>
            R_ld<=Xreg;          -- selection registre destination
            mux_sel<=MUX_din;    -- selection data DIN source
        when i_add =>
            mux_sel<='0' & rx;   -- selection registre X
            Rg_ld<='1';          -- ecriture dans REG G
            p_next_state <= exe_3;
        when i_sub =>
            mux_sel<='0' & rx;   -- selection registre X
            Rg_ld<='1';      -- ecriture Rx-A dans REG G
            alu_code<=alu_sub;   -- selection soustraction
            p_next_state <= exe_3;
        when i_ld =>
            R_ld<=Xreg;          -- selection registre destination
            mux_sel<=MUX_din;    -- selection data DIN source
        when i_st =>
            mux_sel<='0' & rx;   -- selection registre adresse source
            Rd_ld<='1';      -- ecriture dans REG data bus
            Rw_d <='1';      -- ecriture
        when i_and =>
            mux_sel<='0' & rx;   -- selection registre X
            Rg_ld<='1';      -- ecriture Rx-A dans REG G
            alu_code<=alu_and;   -- selection and
            p_next_state <= exe_3;
        when others =>
        end case;
    when exe_3 =>
        p_next_state <= fetch1;
        mux_sel<=MUX_Rg;             -- resultat Rg
        R_ld<=Xreg;              -- selection registre destination Rx
    end case;

END PROCESS;

END Behavior;
```