

INF01175 - NEANDER

Wellington M. Espindula



Descrição

O presente trabalho objetiva **implementar** e **testar** o processador hipotético, em VHDL, **NEANDER** descrito por WEBER (2012).

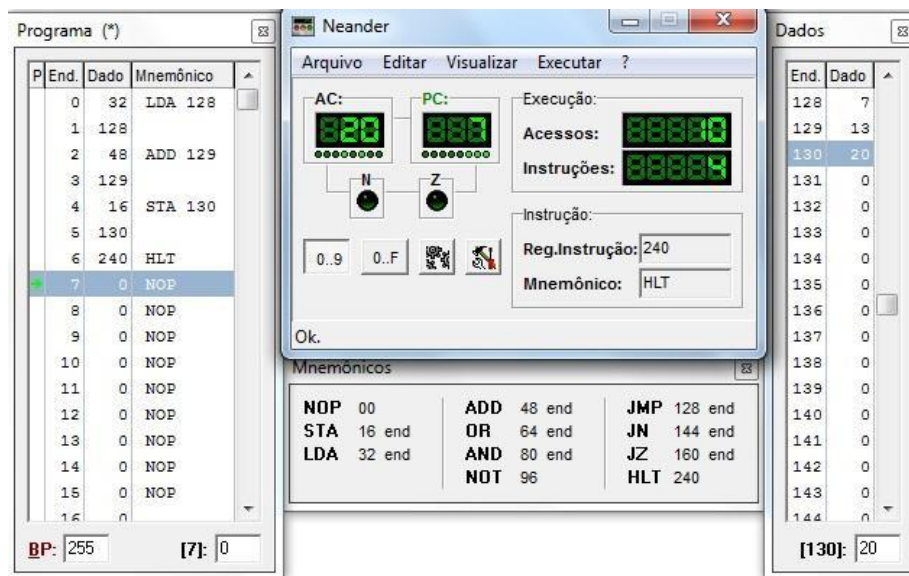


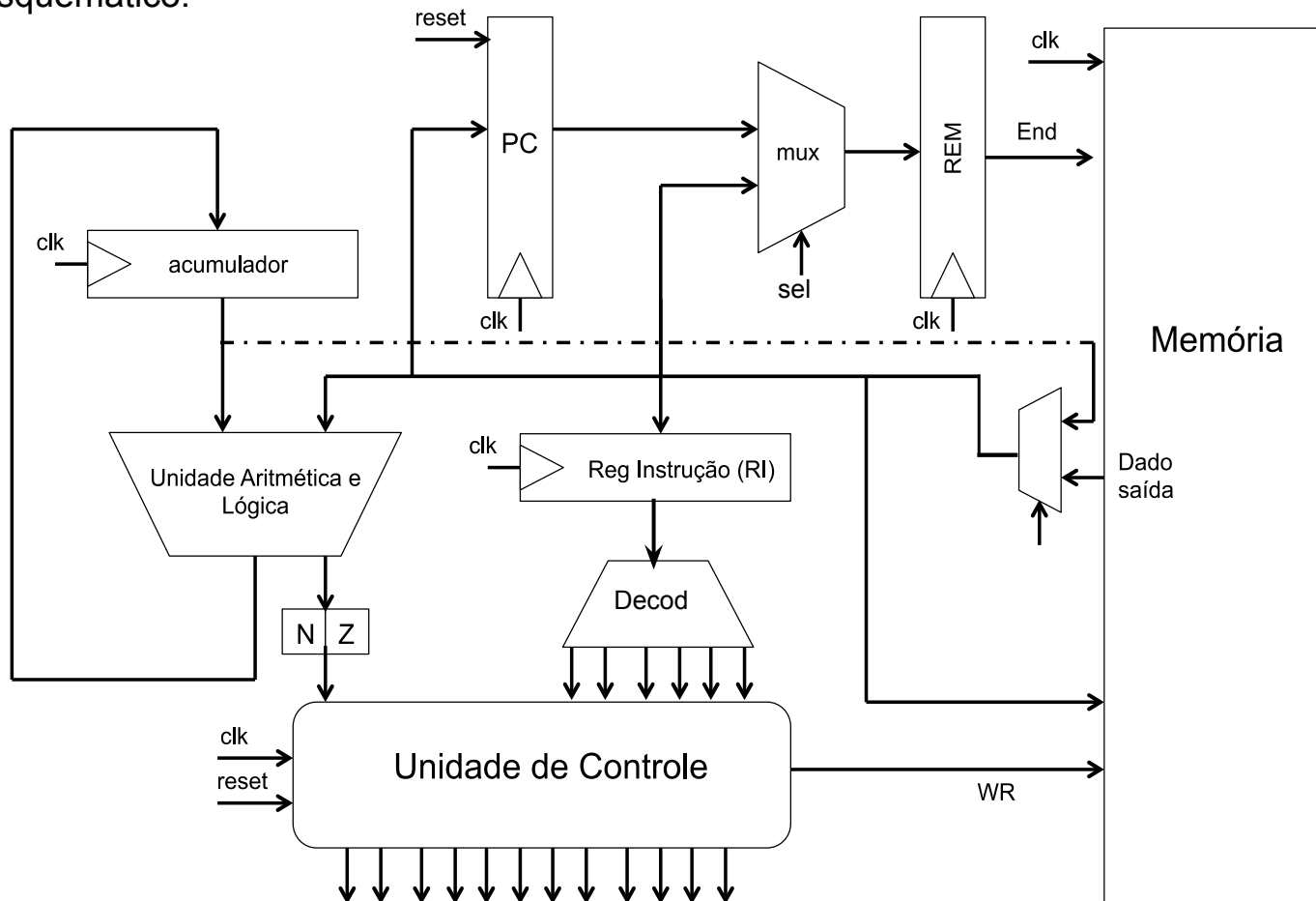
Fig. 1: Interface gráfica da implementação do simulador NEANDER. Fonte: <https://2.bp.blogspot.com/-3pujEqBLpx0/WpdmOF8cZwI/AAAAAAAAAFds/CWfopxkXKRoiipnxWmq0jMli_6V-S6NgCLcBGAs/s1600/2.jpg>

Referências

WEBER, R. F. Fundamentos de arquitetura de computadores. 4. ed. Porto Alegre: Bookman, 2012. 424 p. (Série Livros Didáticos Informática UFRGS, v. 8).

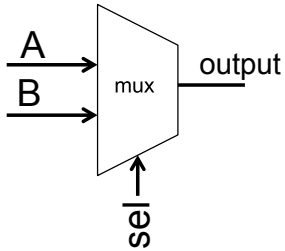
Construção - Datapath

Esquemático:



Circuitos Combinacionais: Mux 2x1

Componente:



VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2to1 is
    Generic (n : natural);
    Port ( A : in STD_LOGIC_VECTOR (n-1 downto 0);
          B : in STD_LOGIC_VECTOR (n-1 downto 0);
          sel : in STD_LOGIC;
          output : out STD_LOGIC_VECTOR (n-1 downto 0));
end mux2to1;

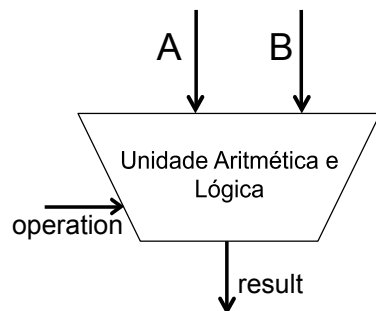
architecture Behavioral of mux2to1 is
begin

    output <= A when sel = '0' else
              B when sel = '1' else
              (others => 'X');

end Behavioral;
```

Circuitos Combinacionais: ULA

Componente:



Cod	# (h)	Op
000	0	ADD
001	1	AND
010	2	OR
011	3	NOT
100	4	B
110	6	SUB
111	7	XOR

VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ula is
    Generic ( width: natural := 8);
    Port ( A : in STD_LOGIC_VECTOR (width-1 downto 0);
          B : in STD_LOGIC_VECTOR (width-1 downto 0);
          operation : in STD_LOGIC_VECTOR (2 downto 0);
          result : out STD_LOGIC_VECTOR (width-1 downto 0);
          overflow : out STD_LOGIC);
end ula;

architecture Behavioral of ula is
begin

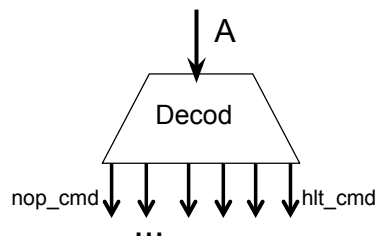
    process(A, B, operation) begin
        case operation is
            when "000" => result <= STD_LOGIC_VECTOR(SIGNED(A) + SIGNED(B));
            when "001" => result <= (A AND B);
            when "010" => result <= (A OR B);
            when "011" => result <= NOT(A);
            when "100" => result <= B;
            when "110" => result <= STD_LOGIC_VECTOR(SIGNED(A) - SIGNED(B));
            when "111" => result <= (A XOR B);
            when others => result <= (others => 'X');
        end case;
    end process;

    overflow <= '0'; -- not implemented

end Behavioral;
```

Circuitos Combinacionais: Decodificador

Componente:



Definição dos Comandos:

Código	# (h)	Instrução
0000	0	NOP
0001	1	STA
0010	2	LDA
0011	3	ADD
0100	4	OR
0101	5	AND
0110	6	NOT
0111	7	SUB
1000	8	JMP
1001	9	JN
1010	A	JZ
1011	B	XOR
1111	F	HLT

Circuitos Combinacionais: Decodificador

VHDL:

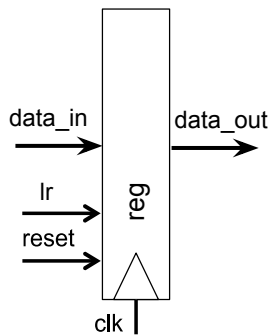
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decod is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
          nop_cmd : out STD_LOGIC;
          sta_cmd : out STD_LOGIC;
          lda_cmd : out STD_LOGIC;
          add_cmd : out STD_LOGIC;
          sub_cmd : out STD_LOGIC;
          or_cmd : out STD_LOGIC;
          and_cmd : out STD_LOGIC;
          xor_cmd : out STD_LOGIC;
          not_cmd : out STD_LOGIC;
          jmp_cmd : out STD_LOGIC;
          jn_cmd : out STD_LOGIC;
          jz_cmd : out STD_LOGIC;
          hlt_cmd : out STD_LOGIC);
end decod;
```

```
architecture Behavioral of decod is
begin
    process(A) begin
        nop_cmd <= '0';
        sta_cmd <= '0';
        lda_cmd <= '0';
        add_cmd <= '0';
        sub_cmd <= '0';
        or_cmd <= '0';
        and_cmd <= '0';
        xor_cmd <= '0';
        not_cmd <= '0';
        jmp_cmd <= '0';
        jn_cmd <= '0';
        jz_cmd <= '0';
        hlt_cmd <= '0';
        case A is
            when "0000" => nop_cmd <= '1'; -- NOP
            when "0001" => sta_cmd <= '1'; -- STA
            when "0010" => lda_cmd <= '1'; -- LDA
            when "0011" => add_cmd <= '1'; -- ADD
            when "0100" => or_cmd <= '1'; -- OR
            when "0101" => and_cmd <= '1'; -- AND
            when "0110" => not_cmd <= '1'; -- NOT
            when "0111" => sub_cmd <= '1'; -- SUB
            when "1000" => jmp_cmd <= '1'; -- JMP
            when "1001" => jn_cmd <= '1'; -- JN
            when "1010" => jz_cmd <= '1'; -- JZ
            when "1011" => xor_cmd <= '1'; -- XOR
            when "1111" => hlt_cmd <= '1'; -- HLT
            when others => null;
        end case;
    end process;
end Behavioral;
```

Circuitos Sequenciais: Registrador

Componente:



VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity reg is
    generic(n: natural := 8);
    Port ( clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          lr : in STD_LOGIC;
          data_in : in STD_LOGIC_VECTOR (n-1 downto 0);
          data_out : out STD_LOGIC_VECTOR (n-1 downto 0));
end reg;

architecture Behavioral of reg is
    signal output: STD_LOGIC_VECTOR (n-1 downto 0) := (others => '0');

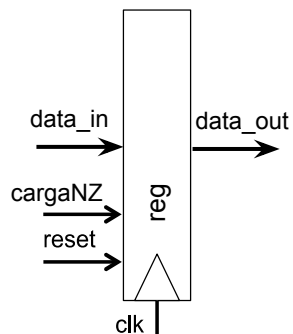
begin
    process(clk, reset, lr, data_in) begin
        if (reset = '1') then
            output <= (others=>'0');
        elsif rising_edge(clk) then
            if (lr='1') then
                output <= data_in;
            end if;
        end if;
    end process;

    data_out <= output;

end Behavioral;
```


Circuitos Sequenciais: Registrador NZ

Componente:



VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity regNZ is
    Generic (n : natural := 8);
    Port ( A : in STD_LOGIC_VECTOR(n-1 downto 0);
          cargaNZ : in STD_LOGIC;
          clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          out_n : out STD_LOGIC;
          out_z : out STD_LOGIC);
end regNZ;

architecture Behavioral of regNZ is
    constant zero_cte : STD_LOGIC_VECTOR (n-1 downto 0) := (others => '0');
    signal negative, zero : STD_LOGIC;
    signal vector_in_NZ : STD_LOGIC_VECTOR(1 downto 0);
    signal vector_out_NZ : STD_LOGIC_VECTOR(1 downto 0);

    begin
    process(A) begin
        negative <= '0';
        zero <= '0';

        if (A = zero_cte) then zero <= '1'; end if;
        if (A(n-1) = '1') then negative <= '1'; end if;
        vector_in_NZ <= negative & zero;
    end process;

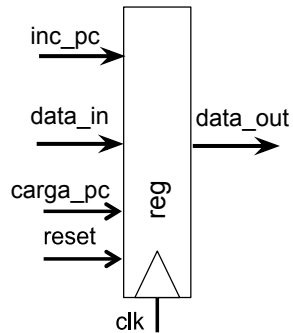
    reg2bits : entity work.reg
        generic map ( n=> 2)
        Port map ( clk => clk,
                  reset => reset,
                  lr => cargaNZ,
                  data_in => vector_in_NZ,
                  data_out => vector_out_NZ);

    out_n <= vector_out_NZ(1);
    out_z <= vector_out_NZ(0);

end Behavioral;
```

Circuitos Sequenciais: Contador

Componente:



VHDL:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity PC is
  Generic (n : natural := 8);
  Port ( data_in : in STD_LOGIC_VECTOR (n-1 downto 0);
        carga_pc : in STD_LOGIC;
        inc_pc : in STD_LOGIC;
        clk : in STD_LOGIC;
        reset : in STD_LOGIC;
        data_out : out STD_LOGIC_VECTOR (n-1 downto 0));
end PC;

architecture Behavioral of PC is
  constant init_const : std_logic_vector(7 downto 0) := (others => '0');
  signal count : std_logic_vector(n-1 downto 0) := init_const;

begin
  process(clk, reset) begin
    if (reset='1') then count <= init_const;
    elsif rising_edge(clk) then
      if carga_pc = '1' then count <= data_in;
      elsif inc_pc = '1' then count <= std_logic_vector(unsigned(count)+1);
      else count <= count;
      end if;
    end if;
  end process;

  data_out <= count;

end Behavioral;
```

Datapath: VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity neander is
    Port ( clk : in STD_LOGIC;
          reset_ext : in STD_LOGIC;
          start : in STD_LOGIC;
          hlt : out STD_LOGIC;
          -- suppressed debug signals
        );
end neander;

architecture Behavioral of neander is

    COMPONENT blk_mem_gen_0
    PORT (
        clka : IN STD_LOGIC;
        wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
        addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
    END COMPONENT;

    signal reset : STD_LOGIC;

    -- control unity signals
    -- input signals
    signal N, Z : STD_LOGIC;
    signal nop_cmd, sta_cmd, lda_cmd, add_cmd: STD_LOGIC;
    signal sub_cmd, or_cmd, and_cmd, xor_cmd, not_cmd :
    STD_LOGIC;
    signal jmp_cmd, jn_cmd, jz_cmd, hlt_cmd: STD_LOGIC;

    --output signals
    signal sel_mux : STD_LOGIC;
    signal inc_pc : STD_LOGIC;
    signal load_pc : STD_LOGIC;
    signal load_rem : STD_LOGIC;
    signal write_mem : STD_LOGIC_VECTOR(0 downto 0);
    signal load_rdm : STD_LOGIC;
    signal sel_ula : STD_LOGIC_VECTOR(2 downto 0);
    signal load_nz : STD_LOGIC;
    signal load_ac : STD_LOGIC;
    signal load_ri : STD_LOGIC;
    signal reset_int : STD_LOGIC;
    signal hlt : STD_LOGIC;

    -- components inputs and outputs signals
    signal out_pc : STD_LOGIC_VECTOR (7 downto 0);
    signal out_mux : STD_LOGIC_VECTOR(7 downto 0);
    signal out_rem : STD_LOGIC_VECTOR(7 downto 0);
    signal in_rdm : STD_LOGIC_VECTOR(7 downto 0);
    signal sel_rdm : STD_LOGIC;
    signal out_rdm : STD_LOGIC_VECTOR(7 downto 0);
    signal out_mem : STD_LOGIC_VECTOR(7 downto 0);
    signal opcode : STD_LOGIC_VECTOR(3 downto 0);
    signal out_ri : STD_LOGIC_VECTOR(7 downto 0);
    signal out_ac : STD_LOGIC_VECTOR(7 downto 0);
    signal out_ula : STD_LOGIC_VECTOR(7 downto 0);
    signal overflow_ula : STD_LOGIC;

begin
```

Datapath: VHDL

```
-- suppressed debug signals assignment
```

```
reset <= reset_ext or reset_int;
```

```
pc_impl : entity work.PC
```

```
    Generic Map (n => 8)
```

```
    Port Map (clk => clk, reset => reset,  
data_in=>out_rdm, inc_pc => inc_pc, carga_pc => load_pc,  
data_out => out_pc);
```

```
mux : entity work.mux2to1
```

```
    Generic Map (n => 8)
```

```
    Port Map (A => out_pc, B => out_rdm, sel => sel_mux,  
output => out_mux);
```

```
rem_impl : entity work.reg
```

```
    Generic Map (n => 8)
```

```
    Port Map (clk => clk, reset => reset, lr => load_rem,  
data_in => out_mux, data_out => out_rem);
```

```
memory : blk_mem_gen_0
```

```
    PORT MAP (
```

```
        clka => clk,  
        wea => write_mem,  
        addra => out_rem,  
        dina => out_rdm,  
        douta => out_mem
```

```
    );
```

```
in_rdm <= out_mem when sel_rdm = '0' else out_ac when
```

```
sel_rdm = '1' else out_mem;
```

```
out_rdm <= in_rdm;
```

```
ri_impl : entity work.reg
```

```
    Generic Map (n => 8)
```

```
    Port Map (clk => clk, reset => reset, lr => load_ri,  
data_in => out_rdm, data_out => out_ri);
```

```
opcode <= out_ri(7 downto 4);
```

```
decod_impl : entity work.decod
```

```
    Port Map (A => opcode, nop_cmd => nop_cmd, sta_cmd =>  
sta_cmd, lda_cmd => lda_cmd, add_cmd => add_cmd, sub_cmd =>  
sub_cmd, or_cmd => or_cmd, and_cmd => and_cmd, xor_cmd =>  
xor_cmd, not_cmd => not_cmd, jmp_cmd => jmp_cmd, jn_cmd =>  
jn_cmd, jz_cmd => jz_cmd, hlt_cmd => hlt_cmd);
```

```
ula : entity work.ula
```

```
    Generic Map (width => 8)
```

```
    Port Map (A => out_ac, B => out_rdm, operation => sel_ula,  
result => out_ula, overflow => overflow_ula);
```

```
ac : entity work.reg
```

```
    Generic Map (n => 8)
```

```
    Port Map (clk => clk, reset => reset, lr => load_ac, data_in  
=> out_ula, data_out => out_ac);
```

```
regNZ_impl : entity work.regNZ
```

```
    Generic Map (n => 8)
```

```
    Port Map (A => out_ula, cargaNZ => load_nz, clk => clk,  
reset => reset, out_n => N, out_z => Z);
```

```
uc : entity work.control_unity
```

```
    Port Map ( clk => clk, reset => reset_int, start => start,
```

```
--input signals
```

```
        N => N, Z => Z, nop_cmd => nop_cmd, sta_cmd =>  
sta_cmd, lda_cmd => lda_cmd, add_cmd => add_cmd, sub_cmd =>  
sub_cmd, or_cmd => or_cmd, and_cmd => and_cmd, xor_cmd =>  
xor_cmd, not_cmd => not_cmd, jmp_cmd => jmp_cmd, jn_cmd =>  
jn_cmd, jz_cmd => jz_cmd, hlt_cmd => hlt_cmd,
```

```
--output
```

```
        sel_mux => sel_mux, sel_rdm => sel_rdm, inc_pc =>  
inc_pc, load_pc => load_pc, load_rem => load_rem, write_mem =>  
write_mem, load_rdm => load_rdm, sel_ula => sel_ula, load_nz =>  
load_nz, load_ac => load_ac, load_ri => load_ri, reset_registers  
=> reset_int, hlt => hlt);
```

```
end Behavioral;
```

Construção: Unidade de Controle

Diagrama de Tempos:

Tempo	STA	LDA	ADD	OR	AND	NOT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	UAL(NOT), carga AC, carga NZ, goto t0
t4	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC	
t5	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	
t6		Espera leitura	Espera leitura	Espera leitura	Espera leitura	
t7	Write, goto t0	ULA(Y), carga AC, carga NZ, goto t0	ULA(ADD), carga AC, carga NZ, goto t0	ULA(OR), carga AC, carga NZ, goto t0	ULA(AND), carga AC, carga NZ, goto t0	

Construção: Unidade de Controle

Diagrama de Tempos:

Tempo	JMP	JN, N=1	JN, N=0	JZ, Z=1	JZ, Z=0	NOP	HLT
t0	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM	sel=0, carga REM
t1	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC	Inc PC
t2	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI	carga RI
t3	sel=0, carga REM	sel=0, carga REM	Inc PC, goto t0	sel=0, carga REM	Inc PC, goto t0	goto t0	Halt
t4	Espera leitura	Espera leitura		Espera leitura			
t5	carga PC, goto t0	carga PC, goto t0		carga PC, goto t0			
t6							
t7							

Unidade de Controle: VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity control_unity is
    Port ( clk : in STD_LOGIC;
          reset : in STD_LOGIC;
          start : in STD_LOGIC;
          N, Z: in STD_LOGIC;

          nop_cmd, sta_cmd, lda_cmd, add_cmd: in STD_LOGIC;
          sub_cmd, or_cmd, and_cmd, xor_cmd, not_cmd : in
STD_LOGIC;
          jmp_cmd, jn_cmd, jz_cmd, hlt_cmd: in STD_LOGIC;

          -- signals output
          sel_mux : out STD_LOGIC;
          sel_rdm : out STD_LOGIC;
          inc_pc : out STD_LOGIC;
          load_pc : out STD_LOGIC;
          load_rem : out STD_LOGIC;
          write_mem : out STD_LOGIC_VECTOR(0 downto 0);
          load_rdm : out STD_LOGIC;
          sel_ula : out STD_LOGIC_VECTOR (2 downto 0);
          load_nz : out STD_LOGIC;
          load_ac : out STD_LOGIC;
          load_ri : out STD_LOGIC;
          reset_registers : out STD_LOGIC;
          hlt : out STD_LOGIC);
end control_unity;
```

```
architecture Behavioral of control_unity is

    type t_state is (t0, t1, t2, t3, t4, t5, t6, t7);
    signal current_state, next_state : t_state;

    constant ADD_ULA : STD_LOGIC_VECTOR(2 downto 0) := "000";
    constant AND_ULA : STD_LOGIC_VECTOR(2 downto 0) := "001";
    constant OR_ULA : STD_LOGIC_VECTOR(2 downto 0) := "010";
    constant NOT_ULA : STD_LOGIC_VECTOR(2 downto 0) := "011";
    constant B_ULA : STD_LOGIC_VECTOR(2 downto 0) := "100";
    constant SUB_ULA : STD_LOGIC_VECTOR(2 downto 0) := "110";
    constant XOR_ULA : STD_LOGIC_VECTOR(2 downto 0) := "111";

begin

    -- State Reg
    process(clk, reset) begin
        if reset='1' then
            current_state <= t0;
        elsif (RISING_EDGE(clk)) then
            current_state <= next_state;
        end if;
    end process;
```

Unidade de Controle: VHDL - FSM (1)

-- FSM

```
process(N, Z, nop_cmd, sta_cmd, lda_cmd, add_cmd, sub_cmd,
or_cmd, and_cmd, xor_cmd, not_cmd, jmp_cmd, jn_cmd, jz_cmd,
hlt_cmd)
begin
    sel_mux <= '0'; sel_rdm <= '0'; inc_pc <= '0'; -- zera regs
    load_pc <= '0'; load_rem <= '0'; write_mem <= "0";
    sel_ula <= "000"; load_nz <= '0'; load_ac <= '0';
    load_ri <= '0'; reset_registers <= '0'; hlt <= '0';

    case current_state is
        when t0 =>
            if (hlt_cmd = '0' or start = '1') then
                sel_mux <= '0';
                load_rem <= '1';
                next_state <= t1;
            elsif (hlt_cmd = '1') then
                hlt <= '1';
            end if;
        when t1 =>
            inc_pc <= '1';
            next_state <= t2;

        when t2 =>
            load_ri <= '1';
            next_state <= t3;
```

```
        when t3 =>
            if (not_cmd = '1') then
                sel_ula <= NOT_ULA;
                load_ac <= '1';
                load_nz <= '1';
                next_state <= t0;
            elsif ((jn_cmd = '1' and N = '0') or
(jz_cmd = '1' and Z = '0')) then
                inc_pc <= '1';
                next_state <= t0;
            elsif (nop_cmd = '1') then
                next_state <= t0;
            elsif (hlt_cmd = '1') then
                hlt <= '1';
                next_state <= t0;
            else
                sel_mux <= '0';
                load_rem <= '1';
                next_state <= t4;
            end if;

            when t4 =>
                if (sta_cmd = '1' or lda_cmd = '1' or
and_cmd = '1' or or_cmd = '1' or xor_cmd = '1' or
add_cmd = '1' or sub_cmd = '1') then
                    inc_pc <= '1';
                end if;
                next_state <= t5;

            when t5 =>
                if (sta_cmd = '1' or lda_cmd = '1' or
and_cmd = '1' or or_cmd = '1' or xor_cmd = '1' or
add_cmd = '1' or sub_cmd = '1') then
                    sel_mux <= '1';
                    load_rem <= '1';
                    next_state <= t6;
                elsif (jmp_cmd = '1' or (jn_cmd = '1' and N
= '1') or (jz_cmd = '1' and Z = '1')) then
                    load_pc <= '1';
                    next_state <= t0;
                end if;
```


Unidade de Controle: VHDL - FSM (2)

```
when t6 =>
    next_state <= t7;

when t7 => if (sta_cmd = '1') then
    sel_rdm <= '1';
    write_mem <= "1";
    elsif (lda_cmd = '1') then
        sel_ula <= B_UULA;
        load_ac <= '1';
        load_nz <= '1';
    elsif (and_cmd = '1') then
        sel_ula <= AND_UULA;
        load_ac <= '1';
        load_nz <= '1';
    elsif (or_cmd = '1') then
        sel_ula <= OR_UULA;
        load_ac <= '1';
        load_nz <= '1';
    elsif (add_cmd = '1') then
        sel_ula <= ADD_UULA;
        load_ac <= '1';
        load_nz <= '1';
    elsif (sub_cmd = '1') then
        sel_ula <= SUB_UULA;
        load_ac <= '1';
        load_nz <= '1';
    elsif (xor_cmd = '1') then
        sel_ula <= XOR_UULA;
        load_ac <= '1';
        load_nz <= '1';
    end if;
    next_state <= t0;

when others =>
    reset_registers <= '1';
    next_state <= t0;

end case;
end process;

end Behavioral;
```

Testes

- Foram realizados *testbenches* unitários para os componentes internos (tb_pc, tb_ula, tb_reg, tb_regNZ)
- Ademais realizei testes usando uma memória de registradores sem latência (<https://github.com/WellingtonEspindula/INF01175-NEANDER/issues/1>)
- Por fim, com as memórias, realizei um *testbench* principal comparando os sinais com os sinais esperados dada a execução no simulador Hidra (modificado para suportar as instruções de SUB e XOR).



Testbench principal

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_neander is
  -- Port ( );
end tb_neander;

architecture Behavioral of tb_neander is
  library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;

  entity tb_neander is
    -- Port ( );
  end tb_neander;

  architecture Behavioral of tb_neander is

    constant clk_period : time := 20ns;

    signal clock, reset, start: STD_LOGIC;
    --output signals
    signal N, Z : STD_LOGIC;
    signal sel_mux : STD_LOGIC;
    signal inc_pc : STD_LOGIC;
    signal load_pc : STD_LOGIC;
    signal load_rem : STD_LOGIC;
    signal write_mem : STD_LOGIC;
    signal load_rdm : STD_LOGIC;
    signal sel_ula : STD_LOGIC_VECTOR(2 downto 0);
    signal load_nz : STD_LOGIC;
    signal load_ac : STD_LOGIC;
    signal load_ri : STD_LOGIC;
    signal reset_int : STD_LOGIC;
    signal hlt : STD_LOGIC;

```

```

    --- components inputs and outputs signals
    signal out_pc : STD_LOGIC_VECTOR (7 downto 0);
    signal out_mux : STD_LOGIC_VECTOR(7 downto 0);
    signal out_rem : STD_LOGIC_VECTOR(7 downto 0);
    signal in_rdm : STD_LOGIC_VECTOR(7 downto 0);
    signal sel_rdm : STD_LOGIC;
    signal out_rdm : STD_LOGIC_VECTOR(7 downto 0);
    signal out_mem : STD_LOGIC_VECTOR(7 downto 0);
    signal opcode : STD_LOGIC_VECTOR(3 downto 0);
    signal out_ri : STD_LOGIC_VECTOR(7 downto 0);
    signal out_ac : STD_LOGIC_VECTOR(7 downto 0);
    signal out_ula : STD_LOGIC_VECTOR(7 downto 0);
    signal overflow_ula : STD_LOGIC;

    begin
    neander_debug : entity work.neander
      Port Map ( clk => clock, reset_ext => reset,
        start => start, hlt => hlt,
        -- suppressed debugs assignments);

    process begin
      clock <= '1';
      wait for clk_period/2;
      clock <= '0';
      wait for clk_period/2;
    end process;

    process begin
      reset <= '1';
      wait for clk_period;
      reset <= '0';
      start <= '1';
      wait for 3*clk_period;
      reset <= '1';
      wait for clk_period;
      reset <= '0';
      wait for clk_period;
      start = '0'
      wait until hlt = '1';
      wait;
    end process;

    end Behavioral;

```

Programa 1: Descrição

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; 1 .Soma de duas matrizes A e B 2x2 com dados de 8 bits
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4  NOP
5
6  LDA A          ; carrega A[0]
7  ADD B          ; soma com B[0]
8  STA C          ; armazena em C[0]
9
10 LDA A+1        ; carrega A[1]
11 ADD B+1        ; soma com B[1]
12 STA C+1        ; armazena em C[1]
13
14 LDA A+2        ; carrega A[2]
15 ADD B+2        ; soma com B[2]
16 STA C+2        ; armazena em C[2]
17
18 LDA A+3        ; carrega A[3]
19 ADD B+3        ; soma com B[3]
20 STA C+3        ; armazena em C[3]
21
22 HLT
23
24 ORG 128
25 Zero: DB 0
26 Um: DB 1
27 Dois: DB 2
28 MenosUm: DB -1
29
30 A: DAB 1,2,3,4
31 B: DAB 3,3,3,3
32 C: DAB [4]
```

Programa 2: Descrição

```
1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2 ;; 2. Result (132d) = 5*Y-4*X
3 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4 NOP
5
6 LDA input1      ; carrega X
7 ADD input1      ; soma X (2*X)
8 ADD input1      ; soma X (3*X)
9 ADD input1      ; soma X (4*X)
10 STA input1X4    ; Armazena (4*X)
11 LDA input2      ; carrega Y
12 ADD input2      ; Soma Y...
13 ADD input2
14 ADD input2
15 ADD input2
16 STA input2X5    ; Armazena (5*Y)
17 SUB input1X4    ; subtrai (5*Y) - (4*X)
18 STA result2     ; Armazena resultado
19 HLT
20
21 ORG 128         ; Variaveis 2
22 input1:        DB 10
23 input2:        DB 20
24 input1X4:       DB 0
25 input2X5:       DB 0
26 result2:       DB 0
```

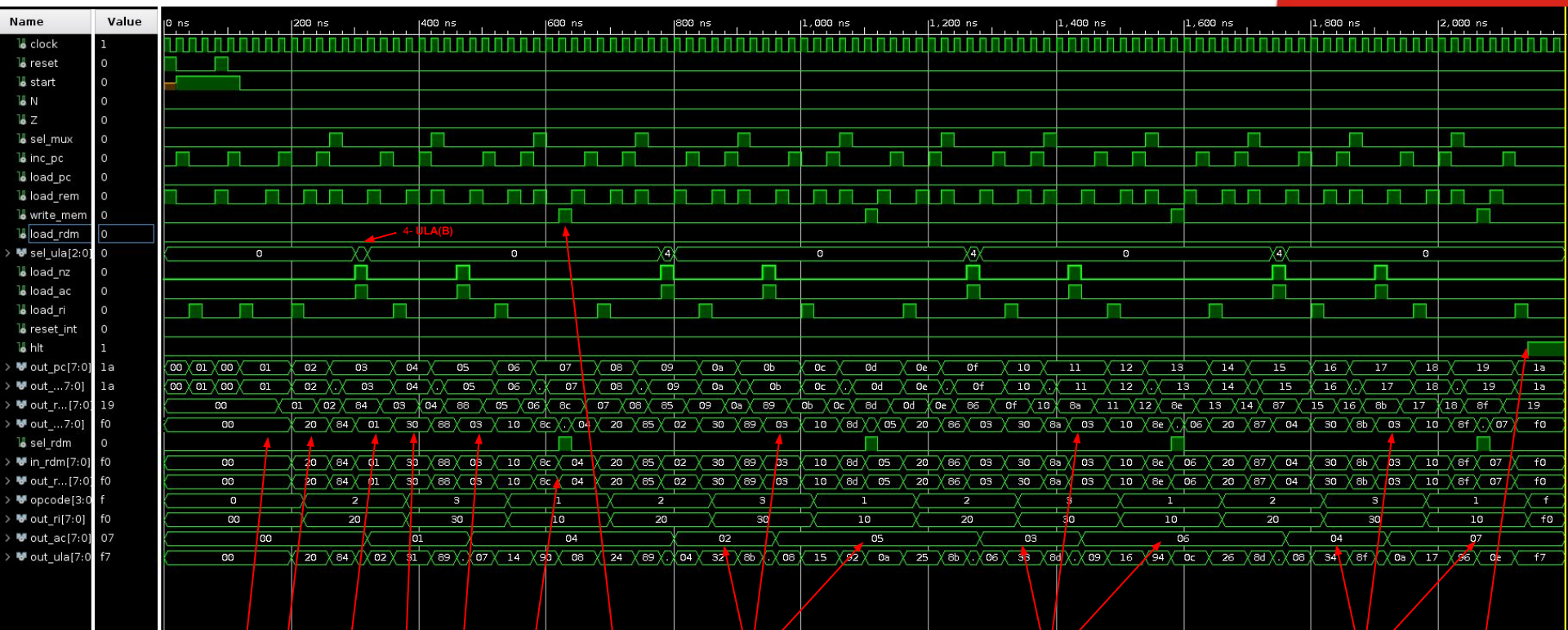
Programa 3: Descrição

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; 3. Programa que calcule a paridade par de um número de 8 bits
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4  NOP
5
6  LDA x          ; Carrega X
7  AND mascara_8  ; Limpa todos os bits exceto o 8º
8  JZ masc7       ; Se 0 então, vai pra mascara do 7º bit
9  LDA paridade_x ; Se != 0, carrega paridade_acumulada
10 XOR Um         ; Xor 1
11 STA paridade_x ; Salva
12
13 masc7:
14 LDA x          ; Mesma coisa pro 7º bit
15 AND mascara_7
16 JZ masc6
17 LDA paridade_x
18 XOR Um
19 STA paridade_x
20
21 masc6:
22 LDA x          ; Mesma coisa pro 6º bit
23 AND mascara_6
24 JZ masc5
25 LDA paridade_x
26 XOR Um
27 STA paridade_x
28
29 masc5:
30 LDA x          ; ... 5º bit
31 AND mascara_5
32 JZ masc4
33 LDA paridade_x
34 XOR Um
35 STA paridade_x
36
37 masc4:
38 LDA x          ; ...
39 AND mascara_4
40 JZ masc3
41 LDA paridade_x
42 XOR Um
43 STA paridade_x
44
45 masc3:
46 LDA x
47 AND mascara_3
48 JZ masc2
49 LDA paridade_x
50 XOR Um
51 STA paridade_x
52
53 masc2:
54 LDA x
55 AND mascara_2
56 JZ masc1
57 LDA paridade_x
58 XOR Um
59 STA paridade_x
60
61 masc1:
62 LDA x
63 AND mascara_1
64 JZ fim_prog3
65 LDA paridade_x
66 XOR Um
67 STA paridade_x
68
69 fim_prog3:
70 LDA paridade_x
71 HLT
72
73 ORG 128
74 Zero:      DB 0
75 Um:        DB 1
76 Dois:      DB 2
77 MenosUm:   DB -1
78
79 x:          DB 162
80 mascara_8:  DB H80
81 mascara_7:  DB H40
82 mascara_6:  DB H20
83 mascara_5:  DB H10
84 mascara_4:  DB H08
85 mascara_3:  DB H04
86 mascara_2:  DB H02
87 mascara_1:  DB H01
88 ORG 150
89 paridade_x: DB 0
90
```


Programa 4: Descrição

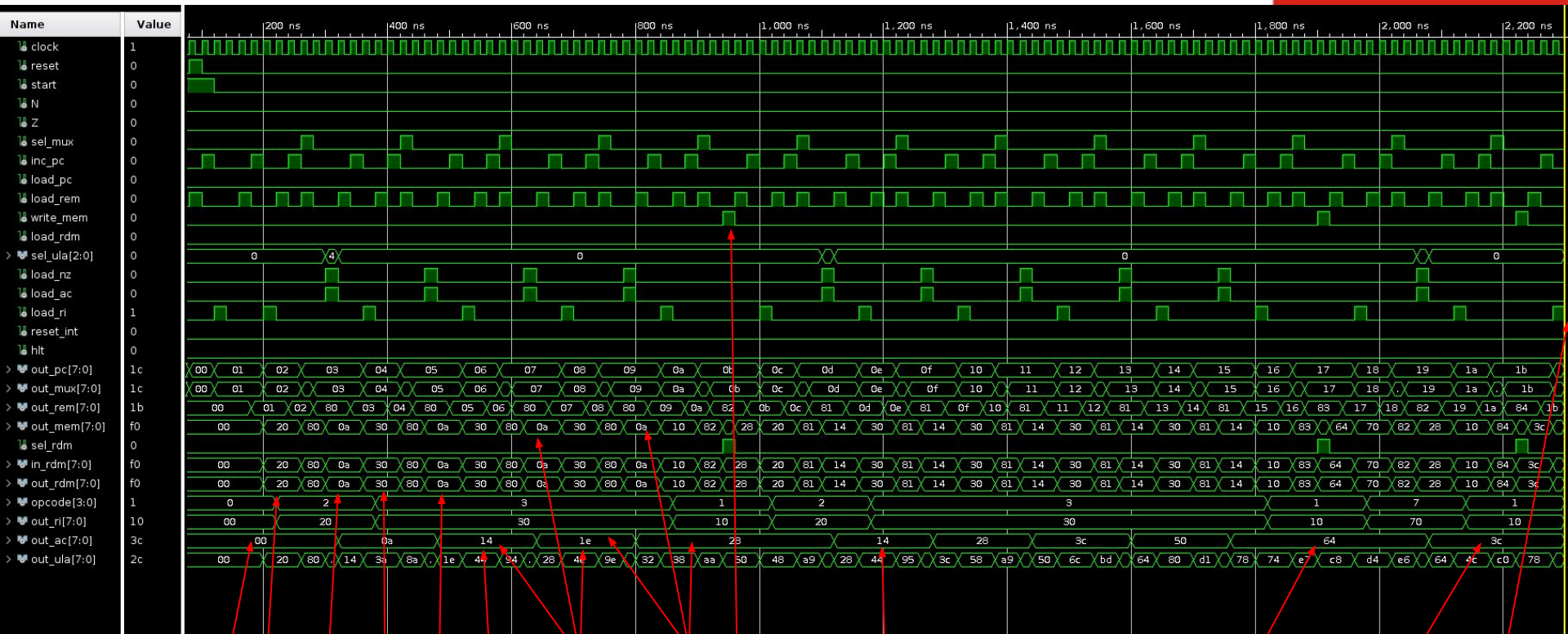
```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;; 4. Programa que enquanto selecao (132d) for positiva.
3  ;;     Se selecao for:
4  ;;     0. Faz swapping de A (133d) e B (134d) e termina
5  ;;     1. Faz o swapping de NOT(A) e NOT (B) e termina
6  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7  NOP
8
9  prog_4:
10 LDA sel_4      ; Carrega selecao
11 JN fim         ; se for negativo, termina
12 JZ op_0        ; Se for zero, faz op_0
13 SUB Um         ; Se for um, faz op_1
14 JZ op_1        ; Se for um, faz op_1
15 JMP prog_4     ; Caso contrário, volta pro inicio
16
17 op_0:
18 LDA a_4        ; Carrega A
19 XOR b_4        ; Xor B
20 STA xor_a_b    ; Salva máscara de Swapping
21 LDA xor_a_b    ; Carrega máscara
22 XOR b_4        ; Mascara Xor B = A
23 STA b_4        ; Salva no B
24 LDA xor_a_b    ; Carrega Mascara
25 XOR a_4        ; Mascara Xor A = B
26 STA a_4        ; Salva no A
27 JMP fim        ; Termina
28
29
30 op_1:
31 LDA a_4        ; nega A
32 NOT
33 STA a_4
34 LDA b_4        ; nega B
35 NOT
36 STA b_4
37 JMP op_0       ; faz o swapping
38
39 fim:
40 HLT
41
42 ORG 128
43 Zero: DB 0
44 Um:   DB 1
45 Dois: DB 2
46 MenosUm: DB -1
47
48 sel_4: DB 1
49 a_4:   DB 33
50 b_4:   DB 92
51 xor_a_b: DB 0
```

Programa 1: Simulação sem atraso



NOP
LDA
ADD
Mem(84h)=01h
Mem(88h)=03h
sel_rdm=1
out_ula = 04h
write_mem=1
02h+03h=05h
03h+03h=06h
04h+03h=07h
HLT

Programa 2: Simulação sem atraso



NOP

LDA

ADD

out_ula = 14h
10h+10h = 14h

14h+0Ah=1Eh

1Eh+0Ah=28h

write_mem

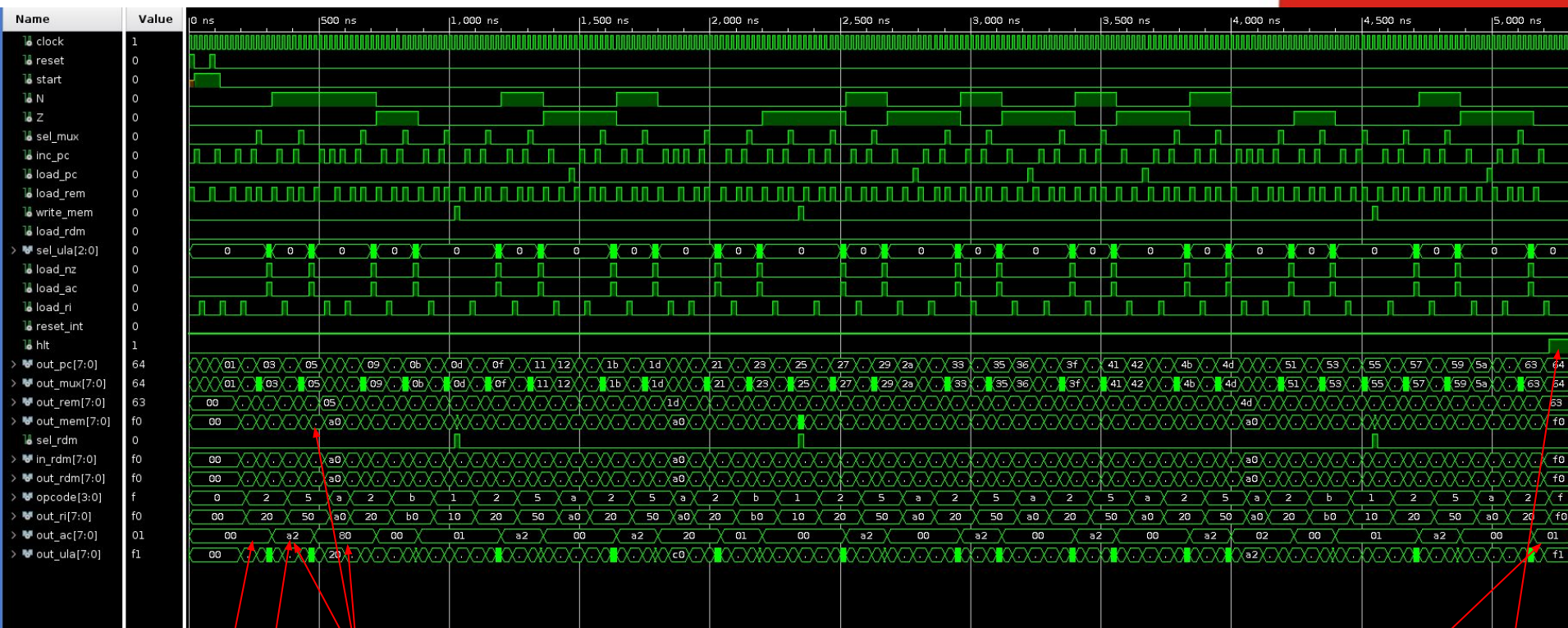
AC < 14h

5*14h = 64h

64h-28h=3Ch

HLT

Programa 3: Simulação sem atraso



NOP

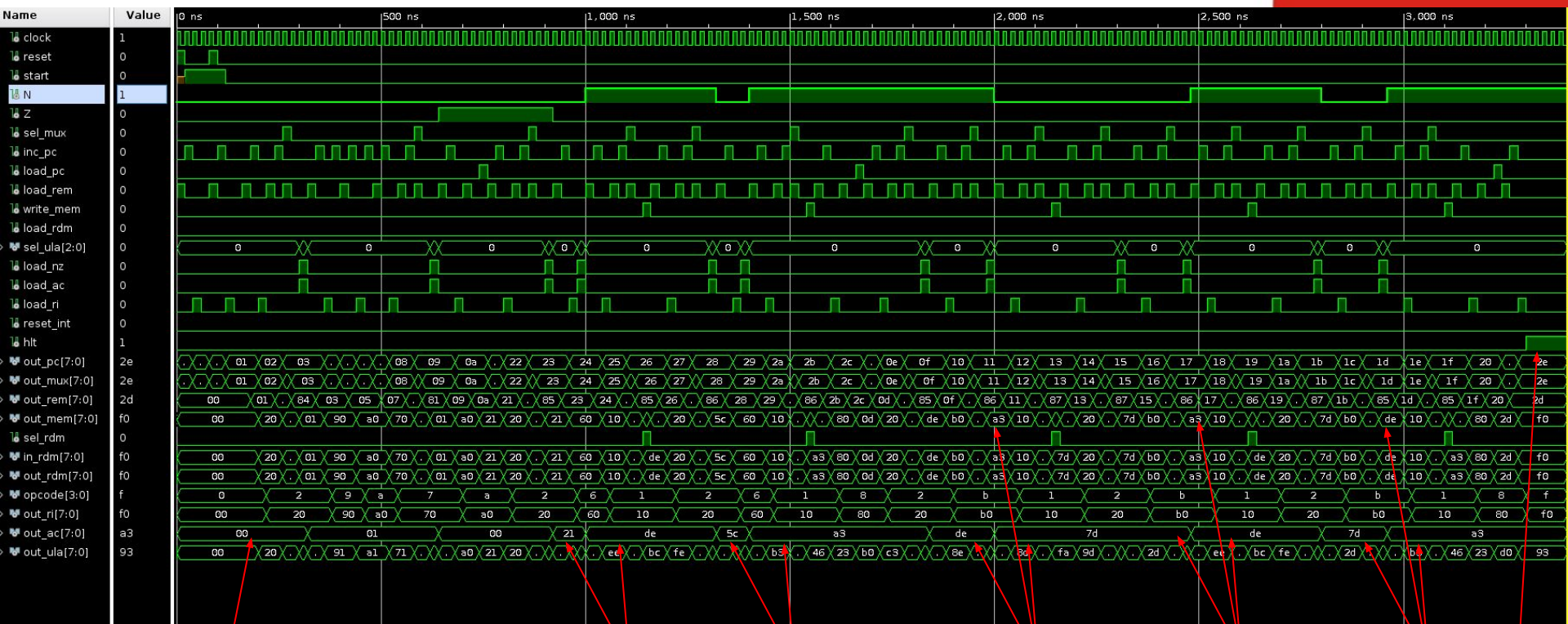
LDA

A2h xor 80h = 80h

paridade de
A2h = 1

HLT

Programa 4: Simulação sem atraso



NOP

NOT(21h) = DEh

NOT(5Ch) = A3h

DEh xor 5Ch = 7Dh

7Dh xor A3h = DEh

7Dh xor DEh = A3h

HLT



Dados de área

FPGA device: **xc7a12ticsg325-1L**

Número de 4-LUTs: **65**

Número de FFs: **36**

Número de BRAM: **1x 18K BRAM**

Número de MULT e ADD DSP: **0**



Dados de tempo de execução

Programa	Número de Instruções	Tempo de execução (ciclos de clock)	Tempo de execução (ns)
Prog. 1 - Soma de Matrizes	14	103	2.060
Prog. 2 - $5*A-4*B$	15	110	2.200
Prog. 3 - Paridade	50	257	5.140
Prog. 4 - Swapping (negado) com XOR	24	161	3.220

Instituto de Informática



Obrigado!

Todos os códigos desta apresentação foram disponibilizados na íntegra no Github: <<https://github.com/WellingtonEspindula/INF01175-NEANDER>>