UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMATICA
Aula 31/1/23 - Sistemas Digitais
Prof. Fernanda Kastensmit

TRABALHO FINAL AHMES
Pontuação: 10 pontos (vale 1/4 da nota do semestre)

Nome: Giordano Souza de Paula Cartão UFRGS: 00308054

Objetivo: projetar e descrever em VHDL o processador Ahmes, implementar 2 programas em sua memória e mostrar através de simulação lógica sem e com atraso o funcionamento.

PASSO 1: 3 pontos

Descrever o DATAPATH do processador AHMES em VHDL em uma entidade apenas chamada de datapath_ahmes.

Cole aqui o código completo em VHDL do datapath

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

entity datapath is
   port(
      CLOCK: in std_logic;
      RESET: in std_logic;

      -- register control inputs
      inc_PC: in std_logic;
      load_PC: in std_logic;
      load_AC: in std_logic;
      load_MA: in std_logic; -- memory address register
      load_MD: in std_logic; -- memory data register
      load_I: in std_logic;  -- instruction register
      load_N: in std_logic;
      load_Z: in std_logic;
      load_V: in std_logic;
      load_C: in std_logic;
      load_B: in std_logic;

      -- selectors
      sel_MUX_MAR: in std_logic_vector(0 downto 0);  -- selects which data reg_MA should recieve (between reg_PC or reg_MD)
      sel_MUX_MDR: in std_logic_vector(0 downto 0);  -- selects which data reg_MD should recieve (between reg_MEM or reg_AC)
      sel_ALU: in std_logic_vector(3 downto 0);  -- selects which arithmetic operation ALU should execute
```

```
        -- memory control
        mem_read: in std_logic_vector(0 downto 0);
        mem_write: in std_logic_vector(0 downto 0);
                mem_out: out std_logic_vector(7 downto 0);

        -- ALU flags
        reg_N: out std_logic;
        reg_Z: out std_logic;
        reg_V: out std_logic:= '0';
        reg_C: out std_logic;
        reg_B: out std_logic:= '0';

        -- instruction flags (from reg_I)
        instruction_flags: out std_logic_vector(23 downto 0));
    end datapath;

architecture Behavioral of datapath is
    -- registersf
    signal reg_PC: std_logic_vector(7 downto 0):= "00000000";
    signal reg_AC: std_logic_vector (7 downto 0):= "00000000";
    signal reg_MA: std_logic_vector(7 downto 0):= "00000000";
    signal reg_MD: std_logic_vector(7 downto 0):= "00000000";
          signal reg_I: std_logic_vector(7 downto 0):= "00000000";
    signal MAR_MUX_out: std_logic_vector(7 downto 0):= "00000000";
    signal MDR_MUX_out: std_logic_vector(7 downto 0):= "00000000";
    signal ALU_out: std_logic_vector(7 downto 0):= "00000000";
    signal omem_out: std_logic_vector(7 downto 0):= "00000000";
    signal IR_DECOD_out: std_logic_vector(23 downto 0):= "000000000000000000000000";

    -- AC flags (for upkeeping internal values until the register update on rising edge)
    signal flag_N: std_logic:= '0';
    signal flag_Z: std_logic:= '0';
    signal flag_V: std_logic:= '0';
    signal flag_C: std_logic:= '0';
    signal flag_B: std_logic:= '0';

    -- ALU
    signal ALU_X: std_logic_vector(7 downto 0):= "00000000"; -- recieves signals from
reg_AC
    signal ALU_Y: std_logic_vector(7 downto 0):= "00000000"; -- recieves signals from
reg_RDM;
          signal ALU_op: std_logic_vector(8 downto 0):= "000000000"; -- ALU operational
signal
    -- the extra bit is used for the overflow(V) flag

    -- ALU operation constants
          constant ALUNOP: std_logic_vector(3 downto 0):= "0000";
          constant ALUADD: std_logic_vector(3 downto 0):= "0001";
          constant ALUOR:  std_logic_vector(3 downto 0):= "0010";
          constant ALUAND: std_logic_vector(3 downto 0):= "0011";
          constant ALUNOT: std_logic_vector(3 downto 0):= "0100";
```

```
        constant ALUSUB: std_logic_vector(3 downto 0):= "0101";
        constant ALUSHR: std_logic_vector(3 downto 0):= "0110";
        constant ALUSHL: std_logic_vector(3 downto 0):= "0111";
        constant ALUROR: std_logic_vector(3 downto 0):= "1000";
        constant ALUROL: std_logic_vector(3 downto 0):= "1001";
        constant ALUY:   std_logic_vector(3 downto 0):= "1010";

-- intruction number constants to be used with instruction_flags slices
        constant iNOP: std_logic_vector(7 downto 0):= "00000000";
        constant iSTA: std_logic_vector(7 downto 0):= "00010000";
        constant iLDA: std_logic_vector(7 downto 0):= "00100000";
        constant iADD: std_logic_vector(7 downto 0):= "00110000";
        constant iOR:  std_logic_vector(7 downto 0):= "01000000";
        constant iAND: std_logic_vector(7 downto 0):= "01010000";
        constant iNOT: std_logic_vector(7 downto 0):= "01100000";
        constant iSUB: std_logic_vector(7 downto 0):= "01110000";
        constant iJMP: std_logic_vector(7 downto 0):= "10000000";
        constant iJN:  std_logic_vector(7 downto 0):= "10010000";
        constant iJP:  std_logic_vector(7 downto 0):= "10010100";
        constant iJV:  std_logic_vector(7 downto 0):= "10011000";
        constant iJNV: std_logic_vector(7 downto 0):= "10011100";
        constant iJZ:  std_logic_vector(7 downto 0):= "10100000";
        constant iJNZ: std_logic_vector(7 downto 0):= "10100100";
        constant iJC:  std_logic_vector(7 downto 0):= "10110000";
        constant iJNC: std_logic_vector(7 downto 0):= "10110100";
        constant iJB:  std_logic_vector(7 downto 0):= "10111000";
        constant iJNB: std_logic_vector(7 downto 0):= "10111100";
        constant iSHR: std_logic_vector(7 downto 0):= "11100000";
        constant iSHL: std_logic_vector(7 downto 0):= "11100001";
        constant iROR: std_logic_vector(7 downto 0):= "11100010";
        constant iROL: std_logic_vector(7 downto 0):= "11100011";
        constant iHLT: std_logic_vector(7 downto 0):= "11110000";

-- BRAM memory component (mem_ahmes)
component mem_ahmes
          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;

begin  -- datapath behavioral start

    -- hw connections (not clock dependent)
                mem_out <= omem_out;

    mem: mem_ahmes
        port map(
            clka => CLOCK,
            wea => mem_write,
```

```
            addra => reg_MA,
            dina => reg_MD,
            douta => omem_out);

        PC: process(CLOCK, RESET)  -- program counter register
            -- standard register, has functions to reset, load values and increment internal value
by 1
            -- mainly recieves values from reg_MD
            -- asynchronous reset
            begin
                if(RESET = '1') then
                    reg_PC <= "00000000";
                elsif(rising_edge(CLOCK)) then
                    if(load_PC = '1') then
                        reg_PC <= reg_MD;
                    elsif(inc_PC = '1') then
                        reg_PC <= std_logic_vector(unsigned(reg_PC) + 1);
                    end if;
                end if;
            end process;

        AC: process(CLOCK, RESET, load_AC)  -- accumulator register
            -- standard register, has functions to reset and load values
            -- mainly recieves values from reg_ALU
            -- asynchronous reset
            begin
                if(RESET = '1') then
                    reg_AC <= "00000000";
                elsif(rising_edge(CLOCK)) then
                                        ALU_X <= reg_AC;   -- retrieves ALU X from reg_AC
(accumulator)
                                        if(load_AC = '1') then
                                                reg_AC  <=  std_logic_vector(ALU_out(7  downto
0));
                                        end if;
                end if;
            end process;

        -- ALU flag registers
        regN: process(CLOCK, RESET, load_N)  -- negative flag register
            -- standard register, has functions to reset and load values
            -- mainly recieves values from internal signal flag_N
            -- asynchronous reset
            begin
                if(RESET = '1') then
                    reg_N <= '0';
                elsif(rising_edge(CLOCK) and load_N = '1') then
                    reg_N <= flag_N;
                end if;
            end process;

        regZ: process(CLOCK, RESET, load_Z)  -- zero flag register
```

```
    -- standard register, has functions to reset and load values
    -- mainly recieves values from internal signal flag_Z
    -- asynchronous reset
    begin
        if(RESET = '1') then
            reg_Z <= '0';
        elsif(rising_edge(CLOCK) and load_Z = '1') then
            reg_Z <= flag_Z;
        end if;
    end process;

regV: process(CLOCK, RESET, load_V) -- overflow flag register
    -- standard register, has functions to reset and load values
    -- mainly recieves values from internal signal flag_V
    -- asynchronous reset
    begin
        if(RESET = '1') then
            reg_V <= '0';
        elsif(rising_edge(CLOCK) and load_V = '1') then
            reg_V <= flag_V;
        end if;
    end process;

regC: process(CLOCK, RESET, load_C)  -- carry flag register
    -- standard register, has functions to reset and load values
    -- mainly recieves values from internal signal flag_C
    -- asynchronous reset
    begin
        if(RESET = '1') then
            reg_C <= '0';
        elsif(rising_edge(CLOCK) and load_C = '1') then
            reg_C <= flag_C;
        end if;
    end process;

regB: process(CLOCK, RESET, load_B)  -- borrow flag register
    -- standard register, has functions to reset and load values
    -- mainly recieves values from internal signal flag_B
    -- asynchronous reset
    begin
        if(RESET = '1') then
            reg_B <= '0';
        elsif(rising_edge(CLOCK) and load_B = '1') then
            reg_B <= flag_B;
        end if;
    end process;

IR: process (CLOCK, RESET, load_I) -- instruction register
    -- standard register, has functions to reset and load values
    -- recieves values from reg_MD
    -- asynchronous reset
    begin
```

```
            if(RESET = '1') then
               reg_I <= "00000000";
            elsif(rising_edge(CLOCK) and load_I = '1') then
               reg_I <= reg_MD;
            end if;
         end process;

      MAR: process(CLOCK, RESET, load_MA) -- memory address register
         -- standard register, has functions to reset and load values
         -- recieves values from MA_MUX_out
         -- asynchronous reset
         begin
            if (RESET = '1') then
               reg_MA <= "00000000";
            elsif (rising_edge(CLOCK) and load_MA = '1') then
               reg_MA <= MAR_MUX_out;
            end if;
         end process;

      MDR: process(CLOCK, RESET, load_MD) -- memory data register
         -- standard register, has functions to reset and load values
         -- recieves values from MD_MUX_out
         -- asynchronous reset
         begin
            if(RESET = '1') then
               reg_MD <= "00000000";
            elsif(rising_edge(CLOCK)) then
                                    ALU_Y <= reg_MD;   -- retrieves ALU Y from reg_MD
(memory data)
                                    if(load_MD = '1') then
                                             reg_MD <= MDR_MUX_out;
                                    end if;
            end if;
         end process;

      MAR_MUX: process(sel_MUX_MAR, reg_PC, reg_MD)  -- MAR 2x1 multiplexer
         -- 2x1 MUX, selecting between reg_PC and reg_MD
         begin
            if(sel_MUX_MAR = "0") then
               MAR_MUX_out <= reg_PC;
            else
               MAR_MUX_out <= reg_MD;
            end if;
         end process;

      MDR_MUX: process(sel_MUX_MDR, oMEM_out, reg_AC, mem_read)  -- MDR 2x1
multiplexer
         -- 2x1 MUX, selecting between MEM_out and reg_AC
         begin
                                 -- requires 'mem_read' signal to enable memory reading
            if(sel_MUX_MDR = "0" and mem_read = "1") then
               MDR_MUX_out <= oMEM_out;
```

```
            else
               MDR_MUX_out <= reg_AC;
            end if;
         end process;

      ALU: process(sel_ALU, ALU_X, ALU_Y, flag_C, ALU_op) -- arithmetic logic unit
         -- Performs arithmetic and logical operations on two input operands
         -- Output result is stored in ALU_op
         -- Operation selected based on sel_ALU input
         -- Available operations:
         --  ADD - adds ALU_X and ALU_Y
         --  OR - logical OR of ALU_X and ALU_Y
         --  AND - logical AND of ALU_X and ALU_Y
         --  NOT - bitwise complement of ALU_X
         --  SUB - subtracts ALU_Y from ALU_X
         --  SHR - right shift of ALU_X by 1 bit
         --  SHL - left shift of ALU_X by 1 bit
         --  ROR - right rotate of ALU_X by 1 bit
         --  ROL - left rotate of ALU_X by 1 bit
         --  NOP - no operation, ALU_op receives ALU_Y
         begin
            if(unsigned(ALU_op(7 downto 0)) = 0) then
               flag_Z <= '1';
            else
               flag_Z <= '0';
            end if;

                              ALU_out <= std_logic_vector(ALU_op(7 downto 0));  -- updates
ALU output with 7 lsb of ALU_op
                              flag_V <= '1';
                              flag_B <= '1';
                              flag_N <= ALU_op(7); -- accumulator msb
                              flag_C <= ALU_op(8);  -- carry flag
            case sel_ALU is
               when ALUADD =>  -- ADD
                                          ALU_op    <=    std_logic_vector(unsigned('0'    &
ALU_X) + unsigned('0' & ALU_Y));

                                          -- checks for overflow
                  if (ALU_X(7) = '0' and ALU_Y(7) = '0' and ALU_op(7) = '1') then
                     flag_V <= '1';
                  else
                     flag_V <= '0';
                  end if;

                                          -- resets other flag signals
                                          flag_B <= '0';
               when ALUOR =>  -- OR
                  ALU_op <= std_logic_vector(('0' & ALU_X) or ('0' & ALU_Y));

                                          -- resets other flag signals
                                          flag_B <= '0';
```

```
                                        flag_V <= '0';
            when ALUAND =>  -- AND
               ALU_op <= std_logic_vector(('0' & ALU_X) and ('0' & ALU_Y));

                                        -- resets other flag signals
                                        flag_B <= '0';
                                        flag_V <= '0';
            when ALUNOT =>  -- NOT
               ALU_op <= std_logic_vector(not('0' & ALU_X));

                                        -- resets other flag signals
                                        flag_B <= '0';
                                        flag_V <= '0';
            when ALUSUB =>  -- SUB
               ALU_op  <=  std_logic_vector(unsigned('0'  &  ALU_X)  -  unsigned('0'  &
ALU_Y));

                                        -- checks for overflow
               if (ALU_X(7) = '1' and ALU_Y(7) = '1' and ALU_op(7) = '0') then
                  flag_V <= '1';
               else
                  flag_V <= '0';
               end if;

                                        -- borrow flag
               flag_B <= ALU_op(7) and ALU_op(8);
            when ALUSHR => -- SHR
               ALU_op(8) <= ALU_X(0);  -- ALU_op, and by consequence carry recieves
reg_AC lsb
               ALU_op(7) <= '0'; -- ULA_out msb recieves 0
               ALU_op(6) <= ALU_X(7);
               ALU_op(5) <= ALU_X(6);
               ALU_op(4) <= ALU_X(5);
               ALU_op(3) <= ALU_X(4);
               ALU_op(2) <= ALU_X(3);
               ALU_op(1) <= ALU_X(2);
               ALU_op(0) <= ALU_X(1);

                                        -- resets other flag signals
                                        flag_B <= '0';
                                        flag_V <= '0';
            when ALUSHL => -- SHL
               ALU_op(8) <= ALU_X(7);  -- ALU_op, and by consequence carry recieves
reg_AC msb
               ALU_op(7) <= ALU_X(6);
               ALU_op(6) <= ALU_X(5);
               ALU_op(5) <= ALU_X(4);
               ALU_op(4) <= ALU_X(3);
               ALU_op(3) <= ALU_X(2);
               ALU_op(2) <= ALU_X(1);
               ALU_op(1) <= ALU_X(0);
               ALU_op(0) <= '0'; -- ULA_out lsb recieves 0
```

```
                                    -- resets other flag signals
                                    flag_B <= '0';
                                    flag_V <= '0';
            when ALUROR => -- ROR
               ALU_op(8) <= ALU_X(0);
               ALU_op(7) <= flag_C;
               ALU_op(6) <= ALU_X(7);
               ALU_op(5) <= ALU_X(6);
               ALU_op(4) <= ALU_X(5);
               ALU_op(3) <= ALU_X(4);
               ALU_op(2) <= ALU_X(3);
               ALU_op(1) <= ALU_X(2);
               ALU_op(0) <= ALU_X(1);

                                    -- resets other flag signals
                                    flag_B <= '0';
                                    flag_V <= '0';
            when ALUROL => -- ROL
                                    ALU_op(8) <= ALU_X(7);
               ALU_op(7) <= ALU_X(6);
               ALU_op(6) <= ALU_X(5);
               ALU_op(5) <= ALU_X(4);
               ALU_op(4) <= ALU_X(3);
               ALU_op(3) <= ALU_X(2);
               ALU_op(2) <= ALU_X(1);
               ALU_op(1) <= ALU_X(0);
               ALU_op(0) <= flag_C;

                                    -- resets other flag signals
                                    flag_B <= '0';
                                    flag_V <= '0';
            when ALUY => -- ULAY
               ALU_op <= std_logic_vector('0' & ALU_Y);

                                    -- resets other flag signals
                                    flag_B <= '0';
                                    flag_V <= '0';
            when others => -- NOP (ULAX - reg_AC)
               ALU_op <= '0' & ALU_X;
         end case;
      end process;

   IR_DECOD: process(reg_I, IR_DECOD_out)   -- IR_DECOD, used for setting instruction_flags
                  -- decodes reg_I signal setting a 24 bits vector, with flags for each of
      -- the processor instructions
      begin
                        instruction_flags <= IR_DECOD_out;
                        IR_DECOD_out <= "000000000000000000000000";
                        case reg_I is
```

```
                                    when        iNOP        =>        IR_DECOD_out        <=
"000000000000000000000001"; -- 00 NOP
                                    when        iSTA        =>        IR_DECOD_out        <=
"000000000000000000000010"; -- 16 STA
                                    when        iLDA        =>        IR_DECOD_out        <=
"000000000000000000000100"; -- 32 LDA
                                    when        iADD        =>        IR_DECOD_out        <=
"000000000000000000001000"; -- 48 ADD
                                    when        iOR          =>       IR_DECOD_out        <=
"000000000000000000010000"; -- 64 OR
                                    when        iAND        =>        IR_DECOD_out        <=
"000000000000000000100000"; -- 80 AND
                                    when        iNOT        =>        IR_DECOD_out        <=
"000000000000000001000000"; -- 96 NOT
                                    when        iSUB        =>        IR_DECOD_out        <=
"000000000000000010000000"; -- 112 SUB
                                    when        iJMP        =>        IR_DECOD_out        <=
"000000000000000100000000"; -- 128 JMP
                                    when        iJN          =>       IR_DECOD_out        <=
"000000000000001000000000"; -- 144 JN
                                    when        iJP          =>       IR_DECOD_out        <=
"000000000000010000000000"; -- 148 JP
                                    when        iJV          =>       IR_DECOD_out        <=
"000000000000100000000000"; -- 152 JV
                                    when        iJNV        =>        IR_DECOD_out        <=
"000000000001000000000000"; -- 156 JNV
                                    when        iJZ          =>       IR_DECOD_out        <=
"000000000010000000000000"; -- 160 JZ
                                    when        iJNZ        =>        IR_DECOD_out        <=
"000000000100000000000000"; -- 164 JNZ
                                    when        iJC          =>       IR_DECOD_out        <=
"000000001000000000000000"; -- 176 JC
                                    when        iJNC        =>        IR_DECOD_out        <=
"000000010000000000000000"; -- 180 JNC
                                    when        iJB          =>       IR_DECOD_out        <=
"000000100000000000000000"; -- 184 JB
                                    when        iJNB        =>        IR_DECOD_out        <=
"000001000000000000000000"; -- 188 JNB
                                    when        iSHR        =>        IR_DECOD_out        <=
"000010000000000000000000"; -- 224 SHR
                                    when        iSHL        =>        IR_DECOD_out        <=
"000100000000000000000000"; -- 225 SHL
                                    when        iROR        =>        IR_DECOD_out        <=
"001000000000000000000000"; -- 226 ROR
                                    when        iROL        =>        IR_DECOD_out        <=
"010000000000000000000000"; -- 227 ROL
                                    when        iHLT        =>        IR_DECOD_out        <=
"100000000000000000000000"; -- 240 HLT
                                    when        others       =>       IR_DECOD_out        <=
"100000000000000000000000";  -- 00 HLT
            end case;
                        end process;
```

end Behavioral;

Qual componente FPGA escolheste para a síntese? Xc7a100t-3csg324

Quantos registradores tem o datapath do RAMSES? 10
Quantas operações diferentes tem a ULA? 11

A área do DATAPTH em # LUTs: 160 e #ffps: 66

main Project Status (03/17/2023 - 03:24:23)			
Project File:	Ahmes.xise	**Parser Errors:**	
Module Name:	main	**Implementation State:**	
Target Device:	xc7a100t-3csg324	• **Errors:**	
Product Version:	ISE 14.7	• **Warnings:**	
Design Goal:	Balanced	• **Routing Results:**	
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	
Environment:	System Settings	• **Final Timing Score:**	

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utiliz**
Number of Slice Registers	66	126,800	
Number used as Flip Flops	66		
Number used as Latches	0		
Number used as Latch-thrus	0		
Number used as AND/OR logics	0		
Number of Slice LUTs	160	63,400	
Number used as logic	159	63,400	
Number using O6 output only	148		
Number using O5 output only	1		
Number using O5 and O6	10		

PASSO 2: 3 pontos

Descrever a parte de controle do AHMES em VHDL como uma maquina de estados.

Dada as tabelas com as instruções do Neander por estado da máquina de estrados

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	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa 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	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	
t5	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	sel=1, carga REM	
t6	carga RDM	Read	Read	Read	Read	
t7	Write, goto t0	UAL(Y), carga AC, carga NZ, goto t0	UAL(ADD), carga AC, carga NZ, goto t0	UAL(OR), carga AC, carga NZ, goto t0	UAL(AND, carga AC, carga NZ, goto t0	

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	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa PC	Read, incrementa 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	incrementa PC, goto t0	sel=0, carga REM	incrementa PC, goto t0	goto t0	Halt
t4	Read	Read		Read			
t5	carga PC, goto t0	carga PC, goto t0		carga PC, goto t0			
t6							
t7							

Código	Instrução	Significado
0000 xxxx	NOP	nenhuma operação
0001 xxxx	STA end	MEM(end) ← AC
0010 xxxx	LDA end	AC ← MEM(end)
0011 xxxx	ADD end	AC ← AC + MEM(end)
0100 xxxx	OR end	AC ← AC **OR** MEM(end) *("ou" bit-a-bit)*
0101 xxxx	AND end	AC ← AC **AND** MEM(end) *("e" bit-a-bit)*
0110 xxxx	NOT	AC ← **NOT** AC (complemento de 1)
0111 xxxx	SUB end	AC ← AC - MEM(end)
1000 xxxx	JMP end	PC ← end (desvio incondicional)
1001 00xx	JN end	IF N=1 THEN PC ← end
1001 01xx	JP end	IF N=0 THEN PC ← end
1001 10xx	JV end	IF V=1 THEN PC ← end
1001 11xx	JNV end	IF V=0 THEN PC ← end

Código	Instrução	Significado
1010 00xx	JZ end	IF Z=1 THEN PC ← end
1010 01xx	JNZ end	IF Z=0 THEN PC ← end
1011 00xx	JC end	IF C=1 THEN PC ← end
1011 01xx	JNC end	IF Z=0 THEN PC ← end
1011 10xx	JB end	IF B=1 THEN PC ← end
1011 11xx	JNB end	IF Z=0 THEN PC ← end
1110 xx00	SHR	C ← AC(0); AC(i-1) ← AC(i); AC(7) ← 0
1110 xx01	SHL	C ← AC(7); AC(i) ← AC(i-1); AC(0) ← 0
1110 xx10	ROR	C ← AC(0); AC(i-1) ← AC(i); AC(7) ← C
1110 xx11	ROL	C ← AC(7); AC(i) ← AC(i-1); AC(0) ← C
1111 xxxx	HLT	término da execução (halt)

Completar a tabela a seguir com as instruções AHMES que não tem no NEANDER

Tempo	SHR	SHL	ROR	ROL	SUB		
T0							
T1							
T2							
T3							
T4							
T5							
T6							
T7							

E as instruções novas de Desvio

Tempo							
T0							
T1							
T2							
T3							
T4							
T5							
T6							
T7							

Cole aqui o VHDL da parte de controle usando FSM com dois process.

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

entity control is
    Port(
            CLOCK: in std_logic;
       RESET: in std_logic;

                instruction_flags: in std_logic_vector(23 downto 0);

                -- ALU flags
                reg_N: in std_logic;
       reg_Z: in std_logic;
       reg_V: in std_logic;
       reg_C: in std_logic;
       reg_B: in std_logic;

                -- register control outputs
            inc_PC: out std_logic;
                load_AC: out std_logic;
       load_PC: out std_logic;
                load_MA: out std_logic;
                load_MD: out std_logic;
       load_I: out std_logic;
                load_N: out std_logic;
                load_Z: out std_logic;
                load_V: out std_logic;
                load_C: out std_logic;
                load_B: out std_logic;

                -- selectors
                sel_MUX_MAR: out std_logic_vector(0 downto 0);
                sel_MUX_MDR: out std_logic_vector(0 downto 0);
                sel_ALU: out std_logic_vector(3 downto 0);

                -- memory control
                mem_read: out  std_logic_vector(0 downto 0);
                mem_write: out std_logic_vector(0 downto 0));
        end control;

architecture Behavioral of control is
    type state_type is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11);
        signal next_state, current_state: state_type;

        -- intruction signals to be used with instruction_flags slices
        signal oNOP: std_logic;
        signal oSTA: std_logic;
        signal oLDA: std_logic;
        signal oADD: std_logic;
```

```
    signal oOR: std_logic;
    signal oAND: std_logic;
    signal oNOT: std_logic;
    signal oSUB: std_logic;
    signal oJN: std_logic;
    signal oJP: std_logic;
    signal oJV: std_logic;
    signal oJNV: std_logic;
    signal oJZ: std_logic;
    signal oJNZ: std_logic;
    signal oJC: std_logic;
    signal oJNC: std_logic;
    signal oJB: std_logic;
    signal oJNB: std_logic;
    signal oSHR: std_logic;
    signal oSHL: std_logic;
    signal oROR: std_logic;
    signal oROL: std_logic;
    signal oHLT: std_logic;

    -- ALU operation constants
    constant ALUNOP: std_logic_vector(3 downto 0):= "0000";
    constant ALUADD: std_logic_vector(3 downto 0):= "0001";
    constant ALUOR: std_logic_vector(3 downto 0):= "0010";
    constant ALUAND: std_logic_vector(3 downto 0):= "0011";
    constant ALUNOT: std_logic_vector(3 downto 0):= "0100";
    constant ALUSUB: std_logic_vector(3 downto 0):= "0101";
    constant ALUSHR: std_logic_vector(3 downto 0):= "0110";
    constant ALUSHL: std_logic_vector(3 downto 0):= "0111";
    constant ALUROR: std_logic_vector(3 downto 0):= "1000";
    constant ALUROL: std_logic_vector(3 downto 0):= "1001";
    constant ALUY: std_logic_vector(3 downto 0):= "1010";

begin
        -- hw connections (slices instruction_flags to internal instruction signals)
        oNOP <= instruction_flags(0);
        oSTA  <= instruction_flags(1);
        oLDA  <= instruction_flags(2);
        oADD <= instruction_flags(3);
        oOR          <= instruction_flags(4);
        oAND <= instruction_flags(5);
        oNOT <= instruction_flags(6);
        oSUB <= instruction_flags(7);
        oJN          <= instruction_flags(9);
        oJP          <= instruction_flags(10);
        oJV          <= instruction_flags(11);
        oJNV  <= instruction_flags(12);
        oJZ          <= instruction_flags(13);
        oJNZ  <= instruction_flags(14);
        oJC          <= instruction_flags(15);
        oJNC  <= instruction_flags(16);
        oJB          <= instruction_flags(17);
```

```
oJNB  <= instruction_flags(18);
oSHR <= instruction_flags(19);
oSHL  <= instruction_flags(20);
oROR <= instruction_flags(21);
oROL  <= instruction_flags(22);
oHLT  <= instruction_flags(23);

SC: process(CLOCK, RESET) -- FSM state control
   begin
                if(RESET = '1') then
                        current_state <= S0;
                elsif(rising_edge(CLOCK)) then
                        current_state <= next_state;
                end if;
        end process;

FSM: process(
        current_state,

        -- value flags
        reg_N,
        reg_Z,
        reg_V,
        reg_C,
        reg_B,

        -- instruction flags
        oNOP,
        oSTA,
        oLDA,
        oADD,
        oOR,
        oAND,
        oNOT,
        oSUB,
        oJN,
        oJP,
        oJV,
        oJNV,
        oJZ,
        oJNZ,
        oJC,
        oJNC,
        oJB,
        oJNB,
        oSHR,
        oSHL,
        oROR,
        oROL,
        oHLT)
        -- FSM description
        begin
```

```
-- resets signals
inc_PC <= '0';
load_AC <= '0';
load_PC <= '0';
load_MA <= '0';
load_MD <= '0';
load_I <= '0';
load_N <= '0';
load_Z <= '0';
load_V <= '0';
load_C <= '0';
load_B <= '0';
sel_MUX_MAR <= "0";
sel_MUX_MDR <= "0";
mem_write <= "0";
mem_read <= "0";
sel_ALU <= ALUNOP;  -- default ALU operation

case current_state is
	when S0 =>  -- POINTS
		-- updates memory cursor
		sel_MUX_MAR <= "0"; -- reg_MA <= reg_PC
		load_MA <= '1';  -- updates MAR

		-- goto S1
		next_state <= S1;
	when S1 =>  -- SKIP

		-- goto S2
		next_state <= S2;
	when S2 =>  -- READS
		-- reads byte from memory updating MD
		mem_read <= "1";
		sel_MUX_MDR <= "0"; -- reg_MD <= mem_out
		load_MD <= '1';

		-- increments reg_PC
		inc_PC <= '1';   -- adjusts memory cursor and
program counter

		-- goto S3
		next_state <= S3;
	when S3 =>  -- FETCH
		-- fetches new instruction

		load_I <= '1';  -- reg_I <= reg_MD

		-- goto s3
		next_state <= S4;
	when S4 =>  -- INS/POINT
		if(oNOP = '1') then
			-- goto S0
```

```
                next_state <= S0;
            elsif(oNOT = '1') then
                -- updates ALU operation
                sel_ALU <= ALUNOT;

                -- updates AC value
                load_AC <= '1';

                -- updates flag signals
                load_N <= '1';
                load_Z <= '1';
                load_V <= '1';
                load_C <= '1';
                load_B <= '1';

                -- goto S0
                next_state <= S0;
            elsif(oJN = '1' and reg_N = '0') then -- JN if n=0
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJP = '1' and reg_N = '1') then -- JP if n=1
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJV = '1' and reg_V = '0') then -- JV if v=0
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJNV = '1' and reg_V = '1') then -- JNV if v=1
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJZ = '1' and reg_Z = '0') then -- JZ if z=0
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJNZ = '1' and reg_Z = '1') then -- JNZ if z=1
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
```

```
                next_state <= S0;
            elsif(oJC = '1' and reg_C = '0') then -- JC if c=0
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJNC = '1' and reg_C = '1') then -- JNC if c=1
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJB = '1' and reg_B = '0') then -- JB if b=0
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oJNB = '1' and reg_B = '1') then -- JNB if b=1
                -- conditions does not match for jumping to
given addres
                -- goes to the next instruction, going to S0
                inc_PC <= '1';
                next_state <= S0;
            elsif(oSHR = '1') then  -- SHR
                -- updates ALU instruction
                sel_ALU <= ALUSHR;

                -- updates accumulator value
                load_AC <= '1';

                -- updates logic flag signals
                load_N <= '1';
                load_Z <= '1';
                load_V <= '1';
                load_C <= '1';
                load_B <= '1';

                -- goes to S0
                next_state <= S0;
            elsif(oSHL = '1') then  -- SHL
                -- updates ALU instruction
                sel_ALU <= ALUSHL;

                -- updates accumulator value
                load_AC <= '1';

                -- updates logic flag signals
                load_N <= '1';
                load_Z <= '1';
                load_V <= '1';
```

```
                load_C <= '1';
                load_B <= '1';

                -- goes to S0
                next_state <= S0;
            elsif(oROR = '1') then  -- ROR
                -- updates ALU instruction
                sel_ALU <= ALUROR;

                -- updates accumulator value
                load_AC <= '1';

                -- updates logic flag signals
                load_N <= '1';
                load_Z <= '1';
                load_V <= '1';
                load_C <= '1';
                load_B <= '1';

                -- goes to S0
                next_state <= S0;
            elsif(oROL = '1') then  -- ROL
                -- updates ALU instruction
                sel_ALU <= ALUROL;

                -- updates accumulator value
                load_AC <= '1';

                -- updates logic flag signals
                load_N <= '1';
                load_Z <= '1';
                load_V <= '1';
                load_C <= '1';
                load_B <= '1';

                -- goes to S0
                next_state <= S0;
            elsif(oHLT = '1') then -- HLT
                -- goto S8 (loop state)
                next_state <= S8;
            else
                -- for any sucessful jumping instruction and
others

                -- sets memory cursor position to current
reg_PC value

                sel_MUX_MAR <= "0"; -- reg_MA <=
reg_PC

                load_MA <= '1';  -- updates reg_MA with
reg_PC


                sel_MUX_MDR <= "1";
                next_state <= S5;
```

```
                end if;
            when S5 =>  -- SKIP

                -- goto S6
                next_state <= S6;
            when S6 =>  -- SKIP
                -- reads a new byte from memory
                mem_read <= "1";
                sel_MUX_MDR <= "0";  -- reg_MD <= mem_out
                load_MD <= '1'; -- updates memory data register


                -- increments reg_PC for any of the following instructions
                if(oSTA = '1' or oLDA = '1' or oADD = '1' or oOR = '1' or oAND = '1' or oSUB = '1') then
                    inc_PC <= '1';
                else
                    inc_PC <= '0';
                end if;

                -- goes to S7
                next_state <= S7;
            when S7 =>  -- POINT
                if(oSTA = '1' or oLDA = '1' or oADD = '1' or oOR = '1' or oAND = '1' or oSUB = '1') then
                    sel_MUX_MAR <= "1";    -- reg_MA <= reg_MD
                    load_MA <= '1';  -- updates reg_MA value

                    -- goto S8
                    next_state <= S8;
                else  -- for any other instruction (jumps)
                    -- updates program counter
                    load_PC <= '1';

                    -- goto S0
                    next_state <= S0;
                end if;
            when S8 =>  -- SKIP
                next_state <= S9;
            when S9 =>  -- READ
                if(oSTA = '1') then  -- STA
                    sel_MUX_MDR <= "1";    -- reg_MD <= reg_AC
                    load_MD <= '1';  -- updates reg_MD value
                else
                    -- for any other instruction
                    mem_read <= "1";
                    sel_MUX_MDR <= "0";    -- reg_MD <= mem_out
                    load_MD <= '1';  -- updates MD value
```

```
                end if;

                -- goto s10
                next_state <= S10;
            when S10 =>  -- INS
                if(oSTA = '1') then
                    mem_write <= "1";

                    -- goto S0
                    next_state <= S0;
                elsif(oLDA = '1') then
                    -- updates ULA_out and by consequence the accumulator with reg_MD
                    load_AC <= '1';

                    -- updates value flag registers
                    load_N <= '1';
                    load_Z <= '1';
                    load_V <= '1';
                    load_C <= '1';
                    load_B <= '1';

                    -- updates ALU instruction
                    sel_ALU <= ALUY;

                    -- goto S0
                    next_state <= S0;
                elsif(oADD = '1') then
                    -- updates ALU instruction
                    sel_ALU <= ALUADD;

                    -- updates the accumulator with ULA_out
                    load_AC <= '1';

                    -- updates value flag registers
                    load_N <= '1';
                    load_Z <= '1';
                    load_V <= '1';
                    load_C <= '1';
                    load_B <= '1';

                    -- goto S0
                    next_state <= S0;
                elsif(oOR = '1') then
                    -- updates ALU instruction
                    sel_ALU <= ALUOR;

                    -- updates the accumulator with ULA_out
                    load_AC <= '1';

                    -- updates value flag registers
                    load_N <= '1';
```

```
                                        load_Z <= '1';
                                        load_V <= '1';
                                        load_C <= '1';
                                        load_B <= '1';

                                        -- goto S0
                                        next_state <= S0;
                                    elsif(oAND = '1') then
                                        -- updates ALU instruction
                                        sel_ALU <= ALUAND;

                                        -- updates the accumulator with ULA_out
                                        load_AC <= '1';

                                        -- updates value flag registers
                                        load_N <= '1';
                                        load_Z <= '1';
                                        load_V <= '1';
                                        load_C <= '1';
                                        load_B <= '1';

                                        -- goto S0
                                        next_state <= S0;
                                    elsif(oSUB = '1') then
                                        -- updates ALU instruction
                                        sel_ALU <= ALUSUB;

                                        -- updates the accumulator with ULA_out
                                        load_AC <= '1';

                                        -- updates value flag registers
                                        load_N <= '1';
                                        load_Z <= '1';
                                        load_V <= '1';
                                        load_C <= '1';
                                        load_B <= '1';

                                        -- goto S0
                                        next_state <= S0;
                                    else
                                        -- invalid instruction
                                        next_state <= S11;
                                    end if;
                                when S11 =>  -- HALT
                                    -- loop state (HALT)
                                    next_state <= S10;
                                when others =>  -- HALT
                                    next_state <= S10;
                            end case;
                        end process;
end Behavioral;
```

PASSO 3: 1 ponto

Descrever o programa em Assembly do AHMES que realize a multiplicação de dois números inteiros positivos de 8 bits por Deslocamento e soma em binário e colocar no arquivo .COE na memória BRAM.

Inserir aqui o programa em Assembly com explicação

```
; programa multiplicação 10x2
LDA 128
SHL
HLT

ORG 128
Cte_10:
        DB 10
```

Por deslocamento:

```
memory_initialization_radix=10;
memory_initialization_vector=0,32,128,225,240,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
```

```
; programa multiplicação 10x2
LDA 128
ADD cte_10
HLT

ORG 128
Cte_10:
        DB 10
```

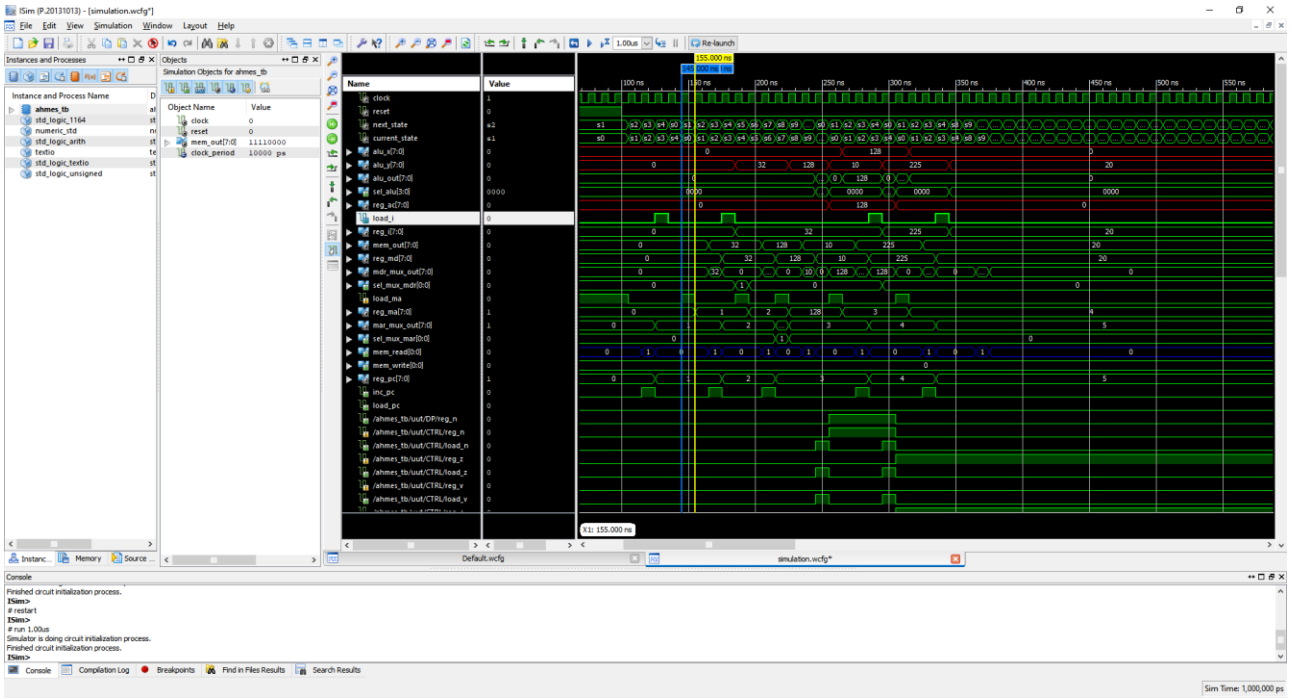
Por somas sucessivas:

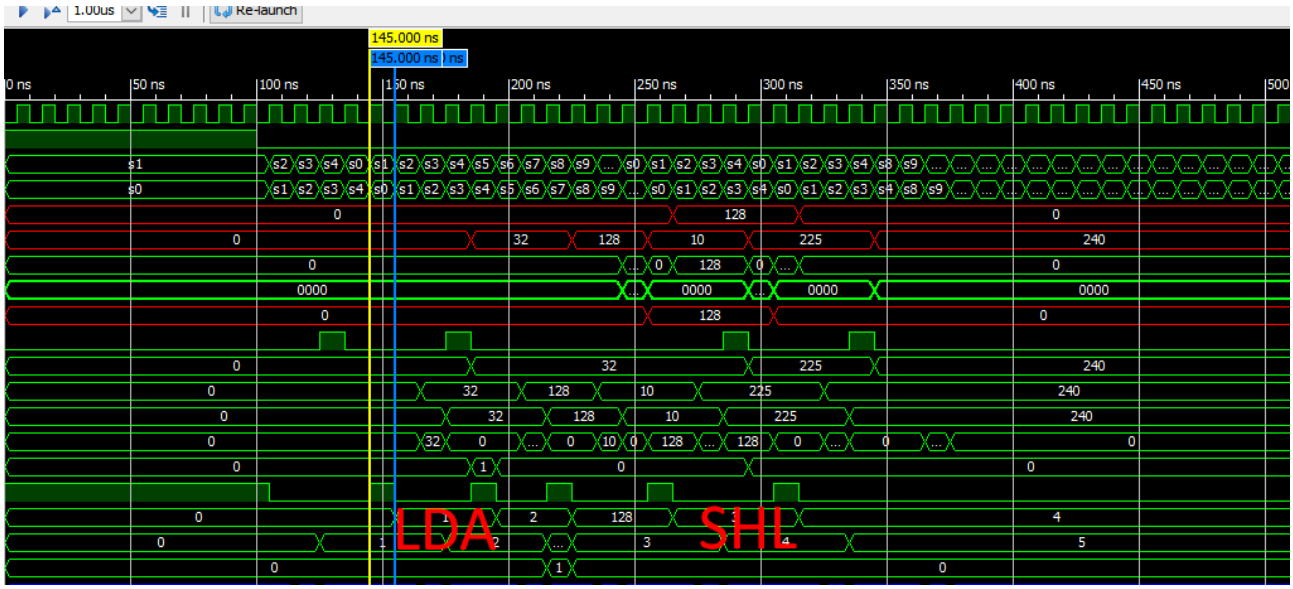
```
memory_initialization_radix=10;
memory_initialization_vector=0,32,128,48,128,240,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
```

PASSO 4: 3 pontos

Simular sem atraso o AHMES com um programa teste a ser feito pelo aluno e depois que testado e funcionando, simular com o programa do passo 3. Depois de tudo funcionando, simular também com atraso.
Lembrem-se que deve ser feito um testbench para a simulação.

Colar aqui o programa teste e simulações (.JPG)





Quantos ciclos de relógio foram necessários para a execução do programa de multiplicação no AHMES? 38 c.c (programa anterior – até atingir o HLT)