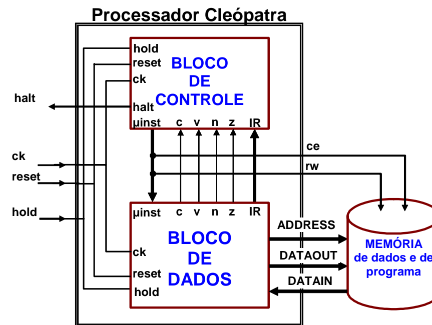
BLOCO DE DADOS E VHDL

Contexto:

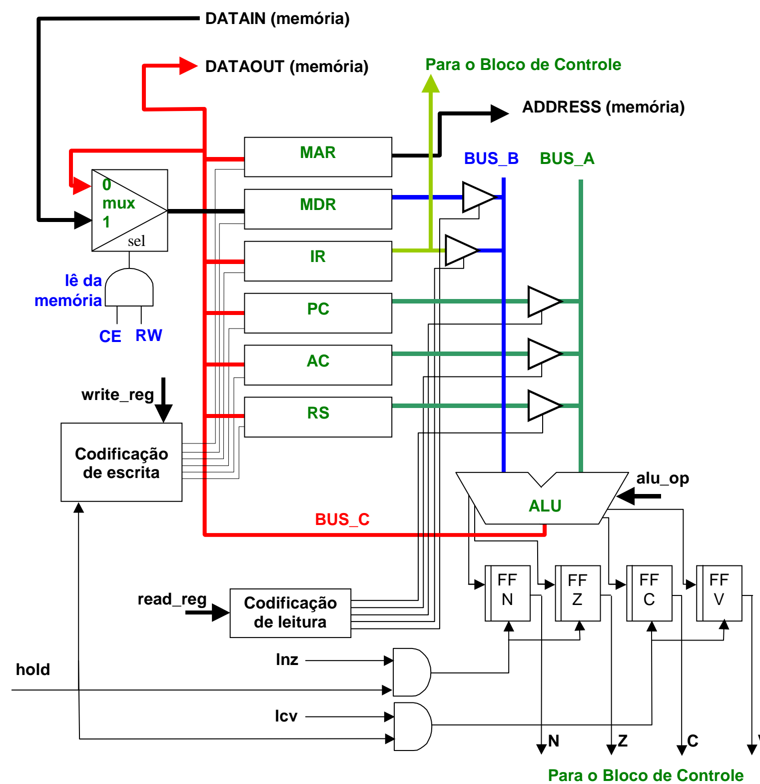
Os alunos já programaram várias aulas em linguagem de montagem (assembly) da Cleópatra (inclusive na terceira aula de Laborg) e tiveram uma introdução ao Bloco de Dados.

A estrutura geral do processador é:



Esta é a estrutura **mais** simples possível, pois é o primeiro contato com dos alunos com organização de computadores.

O bloco de dados é dado pela seguinte estrutura:



O objetivo será mostrar o bloco de dados em VHDL. A idéia é ensinar VHDL por modelos. É uma experiência que queremos fazer.

Aula 1:

Iniciar a aula desenhando de forma simplificada o bloco de dados acima ou usa o projetor (melhor o projetor). Depois de 1 ou 2 minutos revisando a estrutura geral, o que eu já fiz, iniciar o detalhamento da **micro-instrução**.

Olhando o desenho do bloco de dados temos que a micro-instrução proveniente do bloco de controle é formada pelo conjunto:

micro-instrução = { write_reg, read_reg, ALU_op, ce, rw, lnz, lcv }

A semântica destes sinais é:

- *Write_reg*: indica qual do(s) registradore(s) será(ão) escrito(s). Conformo o polígrafo temos a seguinte codificação (página 17):

Denominação	Codificação	Descrição
MAR	000	Escreve no registrador MAR.
MDR	001	Escreve no registrador MDR.
IR	010	Escreve no registrador IR.
PC	011	Escreve no registrador PC.
AC	100	Escreve no registrador AC.
RS	101	Escreve no registrador RS.
PC_MDR	110	Escreve simultaneamente nos registradores MDR e PC.
NULL	111	Não escreve em nenhum registrador.

- *Read_reg*: indica qual do(s) registradore(s) será(ão) lido(s). Conformo o polígrafo temos a seguinte codificação (página 17):

Denominação	Codificação	Descrição
NULL	000	Não coloca nada em BUS_A nem em BUS_B.
MDR	001	Lê o registrador MDR para BUS_B. NADA em BUS_A
IR	010	Lê o registrador IR para BUS_B. NADA em BUS_A
PC	011	Lê o registrador PC para BUS_A. NADA em BUS_B
AC	100	Lê o registrador AC para BUS_A. NADA em BUS_B
RS	101	Lê o registrador RS para BUS_A. NADA em BUS_B
AC_MDR	110	Lê o registrador AC para BUS_A. Lê o registrador MDR para BUS_B.
PC_MDR	111	Lê o registrador PC para BUS_A. Lê o registrador MDR para BUS_B.

- *ALU_op*: operação da Unidade Lógica Aritmética

Operação	Codificação	Descrição	
Soma	000	BUS_C ← BUS_A **+** BUS_B	Gera N, Z, C e V.
Inc	001	BUS_C ← BUS_A **+ 1**	Gera N, Z, C e V.
Não	010	BUS_C ← **not** BUS_A	Gera N, Z
PassaB	100	BUS_C ← BUS_B	Gera N, Z
Ou	101	BUS_C ← BUS_A **or** BUS_B	Gera N, Z
E	110	BUS_C ← BUS_A **and** BUS_B	Gera N, Z
PassaA	111	BUS_C ← BUS_A	Gera N, Z

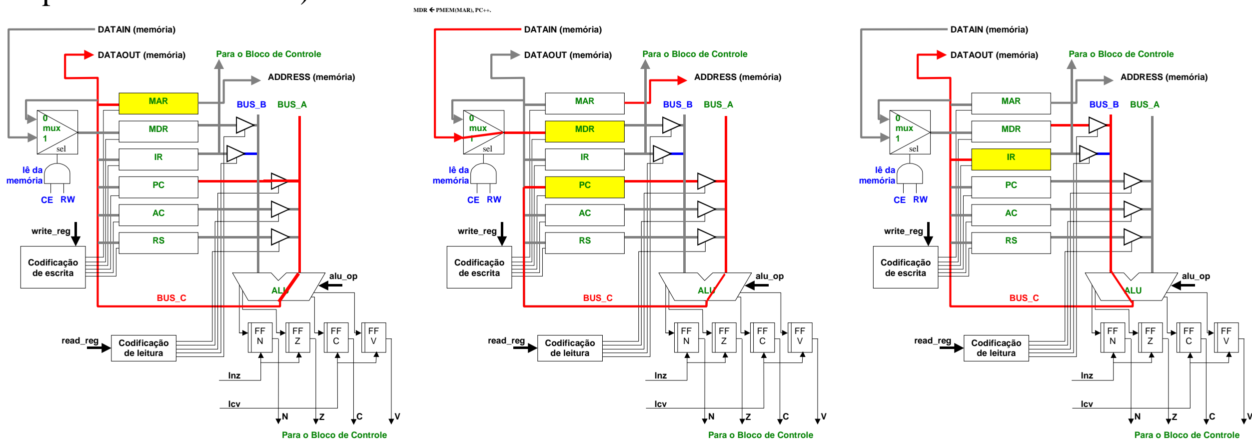
- *we, rw*: controlam a operação com a memória.
- *lnz*: comanda a carga dos flags N (negativo) e Z (zero). Em função da instrução *assembly* é feita esta carga. Por exemplo, ver o OR, que só carrega N/Z.
- *lcv*: comanda a carga dos flags C (carry) e V (overflow). Em função da instrução *assembly* é feita esta carga. Por exemplo, ver a soma, que só carrega os quatro flags.

Exemplo de execução de instrução assembly:

O que iremos mostrar chama-se **micro-simulação,** que ilustra ciclo-a-ciclo o que ocorre na máquina. Para isto, vamos utilizar o exemplo da página 21 do polígrafo. A instrução que será executada é **LDA #33H**. Significa que ao final da execução teremos o valor 33H no registrador acumulador.

CK	DIN	DOUT	ALU_op	WR_reg	RD_reg	lnz	lcv	ce	rw	ADDR	MDR	IR	PC	AC	RS	N Z C V	Microcódigo	*Ciclo de Máquina*
0	--	--	7	0	3	0	0	0	0	00	00	00	00	00	00	0 0 0 0	MAR ← PC.	**Busca LDA ,#**
1	**40**	--	1	6	3	0	0	1	1	00	40	00	01	00	00	0 0 0 0	MDR ← PMEM(MAR), PC++.	
2	--	--	4	2	1	0	0	0	0	00	40	40	01	00	00	0 0 0 0	IR ←MDR.	
3	--	--	7	0	3	0	0	0	0	01	40	40	01	00	00	0 0 0 0	MAR ← PC.	**Execução LDA ,#**
4	**33**	--	1	6	3	0	0	1	1	01	33	40	02	00	00	0 0 0 0	MDR ←PMEM(MAR), PC++.	
5	--	--	4	4	1	1	0	0	0	01	33	40	02	33	00	0 0 0 0	AC ← MDR, LNZ.	

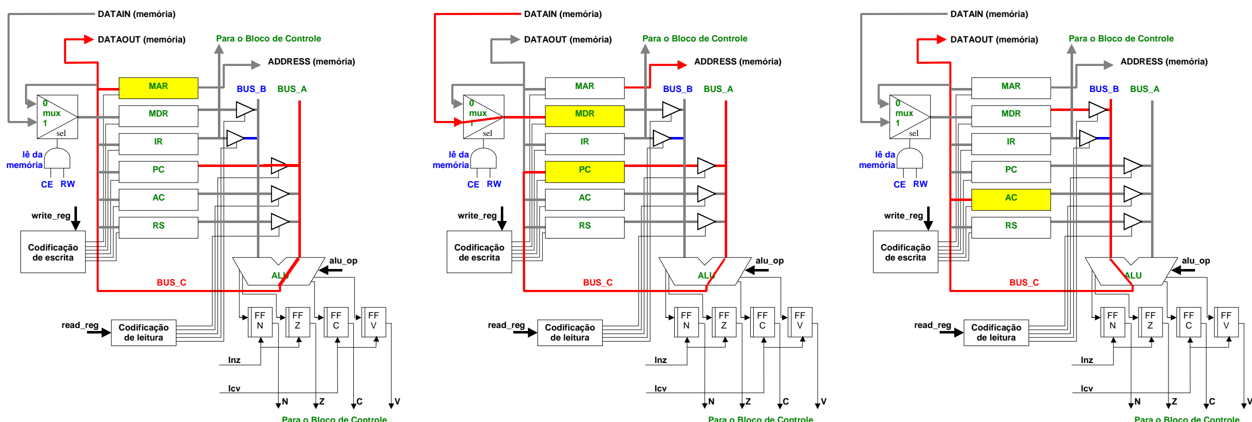
Ciclo a ciclo (normalmente eu desenho no quadro passo a passo o que ocorre para melhor compreensão dos alunos):



Ciclo 1: MAR ← PC.
Carrega o conteúdo do PC para o registrador MAR. [Lê do PC], [ALU opera em PASSA_A], e [grava no MAR]

Ciclo 2: MDR ← PMEM(MAR), PC++.
Operação na qual os alunos se enrolam. Explicar bem há **dois** caminhos paralelos, e que não há problema de ler e escrever o PC no mesmo ciclo (flip-flops mestre-escravo).

Ciclo 3: IR ← MDR
Terminou o ciclo de busca. O IR está carregado. Agora o bloco de controle decodifica e dispara as instruções de execução.



Ciclo 4: MAR ← PC.
Primeiro ciclo da execução .

Ciclo 5: MDR ← PMEM(MAR), PC++. Busca do operando na memória.

Ciclo 6: AC ← MDR.
Terminou a execução do LDA modo imediato de endereçamento.

Deves comentar:

- Este é o funcionamento do bloco de dados, com o bloco de controle enviando as micro-instruções.
- A partir deste momento iremos **construir** o bloco de dados.

Módulo 1 decodificador de leitura (aqui ficarás MUITO tempo, pois eles nunca viram VHDL antes na vida):

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity read_decoder is
    port(read_reg:in std_logic_vector(2 downto 0);
       r_mdr, r_ir, r_pc, r_ac, r_rs: out std_logic);
end read_decoder;

architecture read_decoder of read_decoder is
begin
    ---
    --- Códigos de leitura nos registradores (codificação de leitura):
    --- 1 MDR, 2 IR, 3 PC, 4 AC, 5 RS, 6 MDR/AC, 7 MDR/PC
    ---
    r_mdr <= '1' when read_reg="001" or read_reg="110" or read_reg="111" else '0';
    r_ir  <= '1' when read_reg="010"                                      else '0';
    r_pc  <= '1' when read_reg="011" or read_reg="111"                    else '0';
    r_ac  <= '1' when read_reg="100" or read_reg="110"                    else '0';
    r_rs  <= '1' when read_reg="101"                                      else '0';
end read_decoder;
```

DIZER QUE A ORDEM DAS INSTRUÇÕES NÃO INTERESSA

Entendo a linguagem:

1) Toda descrição VHDL inicia com a declaração de quais bibliotecas serão utilizadas. No nosso exemplo:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
```

A declaração **IEEE** e **std_logic** indicam que o circuito está usando **tipos** que **modelam** estados lógicos além do ‘1’ e do ‘0’. No caso std_logic temos 9 níveis lógicos (não entrar no que é tipo *resolved* e *unresolved*):

```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

2) Toda descrição VHDL é um par entidade-arquitetura.

   - Entidade: declara a **pinagem externa** do circuito
   - Arquitetura: detalha a **implementação**

   Aqui detalhar os sinais de entrada. Mostrar que há um barramento de entrada de 3 bits que especificam qual é a operação de leitura, e 5 sinais de saída, cada um conectado posteriormente ao CE de cada registrador.

3) Implementação.

   A forma de implementação para este circuito é através de **atribuições concorrentes**:

```
r_mdr <= '1' when read_reg="001" or read_reg="110" or read_reg="111" else '0';
```

Esta forma de implementação é adequada quando se deseja implementar circuitos combinacionais do tipo decodificador, multiplexador, comparadores.

E o teste deste circuito para sabermos se funciona?

Falar o que são test benchs.

Um VHDL simples é dado abaixo (rodar em aula):

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;


entity tb  is
end tb;

architecture a1 of tb is
    signal read: std_logic_vector(2 downto 0) := "000";   -- INICIALIZADO!
    signal r_mdr, r_ir, r_pc, r_ac, r_rs: std_logic;
begin

   R_DECOD: entity work.read_decoder port map(read_reg=>read,
            r_mdr=>r_mdr, r_ir=>r_ir, r_pc=>r_pc, r_ac=>r_ac, r_rs=>r_rs);

   process
    begin
        wait for 20 ns;
        read <= read + 1;
   end process;

end a1;
```

1) No início do TB (test bench) temos um novo componente (*IEEE.Std_Logic_unsigned.all*), que serve para fazer operações **aritméticas** entre vetores std_logic_vector. Na prática tem muito mais dentro deste pacote, não comentar ainda.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
```

2) O par *entidade-arquitetura* dos TB não tem pinos externos, pois o objetivo é testar o circuito.

```
entity tb  is
end tb;
```

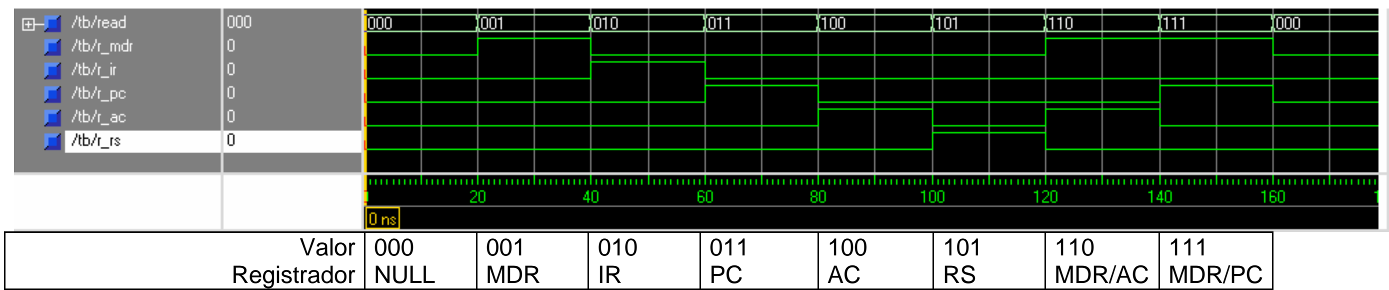
3) Notar que entre a *architecture* e o *begin* foram colocados sinais. Estes sinais serão utilizados pelo circuito que será testado.

4) A *architecture* tem duas partes: **instanciação *do circuito em teste* e o processo gerado de estímulos.**

   - Explicar o como utilizar *port maps*

   - Explicar o *process*. Mostrar aos alunos que o sinal foi inicialmente inicializado em “000”. O *process* fica parado 20ns com um dado valor, depois deste tempo o valor é incrementado (+1), e para novamente 20 ns. Este *process* é um laço infinito.

Se der simular em aula e comentar com eles o resultado:



Valor	000	001	010	011	100	101	110	111
Registrador	NULL	MDR	IR	PC	AC	RS	MDR/AC	MDR/PC

Aula 2: ULA e registradores

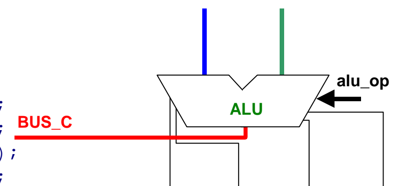
Parte I - ULA:

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

entity alu is
    port(A_bus, B_bus:  in std_logic_vector(7 downto 0);
         alu_op:        in std_logic_vector(2 downto 0);
         C_Bus:         out std_logic_vector(7 downto 0);
         n_inst, z_inst, c_inst, v_inst: out std_logic);
end alu;

architecture alu of alu is

   signal A_bus_int, B_bus_int, C_bus_int : std_logic_vector(8 downto 0);

begin

    A_bus_int <= '0' & A_bus;

    B_bus_int <= '0' & B_bus;

    C_bus <= C_bus_int (7 downto 0); -- toma n-1 bits menos significativos

     process(alu_op, A_bus_int, B_bus_int)
      begin
       case alu_op is
        when "000" =>  C_bus_int <= A_bus_int + B_bus_int;       -- soma
        when "001" =>  C_bus_int <= A_bus_int + 1;               -- inc
        when "010" =>  C_bus_int <= not A_bus_int;               -- not
        when "100" =>  C_bus_int <= B_bus_int;                   -- passa B
        when "101" =>  C_bus_int <= A_bus_int or B_bus_int;      -- ou lógico
        when "110" =>  C_bus_int <= A_bus_int and B_bus_int;     -- e lógico
        when "111" =>  C_bus_int <= A_bus_int;                   -- passa A
        when others => C_bus_int <= A_bus_int;                   -- default: passa A;
       end case;
     end process;

     n_inst <= C_bus_int(7);

     z_inst <= '1' when (C_bus_int(7 downto 0) = "00000000")    else '0';

     c_inst <= C_bus_int(8);

     v_inst <= '1' when (A_bus_int(7)=B_bus_int(7) and
                         A_bus_int(7)/=C_bus_int(7)) else '0';

end alu;
```

1) Bibliotecas – já explicado.

2) Interface externa:

```
entity alu is
    port(A_bus, B_bus:  in std_logic_vector(7 downto 0);
         alu_op:        in std_logic_vector(2 downto 0);
         C_Bus:         out std_logic_vector(7 downto 0);
         n_inst, z_inst, c_inst, v_inst: out std_logic);
end alu;
```



Temos 2 barramentos de 8 bits de entrada, A_BUS e B_BUS, um barramento de saída de 8 bits, C_BUS, um barramento de 3 bits para designar a operação, 4 bits de flags:

IMPORTANTE:

- os 4 bits de *flag* são **instantâneos**, ou seja, ainda não foram carregados em registradores.
- este é também um circuito puramente combinacional.

3) Sinais internos

```
architecture alu of alu is

   signal A_bus_int, B_bus_int, C_bus_int : std_logic_vector(8 downto 0);

begin
```

- Temos 4 barramentos de 9 (nove) bits. A razão para estende a largura dos barramentos é permitir detectar *carry* e *overflow*.

4) Implementação:

- Extensão dos barramentos A e B

```
A_bus_int <= '0' & A_bus;

B_bus_int <= '0' & B_bus;
```

- Barramento de saída com os 7 bits menos significativos do barramento C interno

```
C_bus <= C_bus_int (7 downto 0); -- toma n-1 bits menos significativos
```

- Implementação da ULA propriamente dita, via um case.

```
process(alu_op, A_bus_int, B_bus_int)      // lista de sensitividade ➔ explicar
 begin
  case alu_op is
   when "000" =>  C_bus_int <= A_bus_int + B_bus_int;       -- soma
   when "001" =>  C_bus_int <= A_bus_int + 1;               -- inc
   when "010" =>  C_bus_int <= not A_bus_int;               -- not
   when "100" =>  C_bus_int <= B_bus_int;                   -- passa B
   when "101" =>  C_bus_int <= A_bus_int or B_bus_int;      -- ou lógico
   when "110" =>  C_bus_int <= A_bus_int and B_bus_int;     -- e lógico
   when "111" =>  C_bus_int <= A_bus_int;                   -- passa A
   when others => C_bus_int <= A_bus_int;                   -- default: passa A;
  end case;
end process;
```

- Flag de negativo é o 8° (oitavo) bit. Insistir com os alunos que isto **não** é o bit de sinal, mas sim $-(2^7)$, ou –128 ➔ complemento de dois.

```
n_inst <= C_bus_int(7);
```

- Flag de zero: uso da **atribuição concorrente,** já visto antes, para implementar um comparador.

```
z_inst <= '1' when (C_bus_int(7 downto 0) = "00000000")    else '0';
```

- Flag de carry out: é o 9° (nono). Simples.

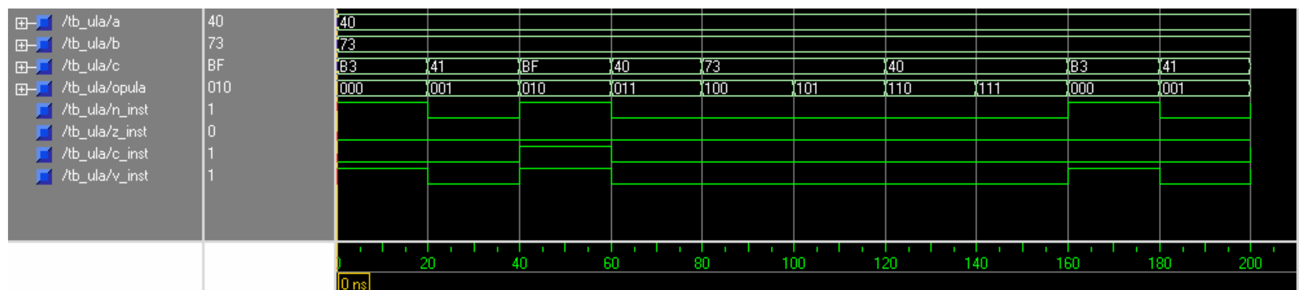
```
c_inst <= C_bus_int(8);
```

- Flag de overflow: é mais complicado... Também us da **atribuição concorrente** A idéia é a seguinte, quando o barramento A e B forem de mesmo sinal (ambos positivos ou ambos negativos) e o sinal da resposta for diferente, é indicação que ultrapassamos a capacidade de representação em complemento de dois.

```
v_inst <= '1' when (A_bus_int(7)=B_bus_int(7) and
                    A_bus_int(7)/=C_bus_int(7)) else '0';
```

TESTE DA ULA: PEDIR COMO EXERCÍCIO PARA ELES TENTAREM FAZER, E DEPOIS TU SIMULAS COM ELES.

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


entity tb_ula  is
end tb_ula;

architecture a1 of tb_ula is
    signal A, B, C : std_logic_vector(7 downto 0);
    signal opula: std_logic_vector(2 downto 0) := "000";
    signal n_inst, z_inst, c_inst, v_inst: std_logic;
begin


    ALU: entity work.alu port map(A_bus=>A, B_bus=>B, C_bus=>C,
                               alu_op=>opula,
                               n_inst=>n_inst,
                               z_inst=>z_inst,
                               c_inst=>c_inst,
                               v_inst=>v_inst);

   process
    begin
        wait for 20 ns;
        opula <= opula + 1;
   end process;

   A <=  x"40";   } ÚNICA DIFERENÇA PARA O TEST BENCH ANTERIOR
   B <=  x"73";

end a1;
```



Verificando:

Código 0: 40H + 73H = B3H   (soma de dois positivos, resultando em um negativo)

- `40H =   0100 0000`
- `73H =   0111 0011`
- `B3H =   1011 0011,`   só que em complemento de dois é: -128 + 32+16+1+2 = - 77!   Logo os *flags* de negativo e overflow estão corretamente ativados.

Código 1: Incrementa A (mostrar o 41H e que os flags foram para zero)

Código 2: NOT A

- `40H      =  0100 0000`
- `not 40H =  1011 1111` ➔ `BFH`

Código 3: não definido, ver a condição *default* do *case*, que é PASSA_A

Código 4: PASSA_B

Código 5: OU

- `40H =   0100 0000`
- `73H =   0111 0011`
- `OU:     0111 0011   ➔ 73H`

Código 6: AND

- `40H =   0100 0000`
- `73H =   0111 0011`
- `AND:    0100 0000   ➔ 40H`

Código 7: PASSA_A

Modificar o TB para gerar diferentes valores de A e B, como a instrução abaixo:

```
A <=  x"40",  x"FE" after 200 ns, X"00" after 400 ns;
```

Exemplo onde ocorre carry e não ocorre overflow:

C4H + 73H (1100 0100 + 0111 0011) = (1) 0011 0111

- Inteiros positivos: 196 + 115 = 55, o *carry out* indica corretamente erro.
- Em complemento de dois: -60 + 115 = 55 = 0011 0111, **sem overflow**

Parte II – registradores

- Agora vamos iniciar com circuitos seqüenciais. Apresentar o circuito abaixo e explicar o funcionamento. INSISTIR:
  1. no conceito de sensibilidade à borda (clock'event)
  2. no conceito de evento assíncrono (reset) e evento síncrono (clock)
  3. falar que o processo **só** é ativado quando alguém na lista de sensitividade mudar, ou seja, o reset ou o clock
  4. **não** precisa do D e o CE na lista de sensitividade, pois ambos estão sujeitos ao clock.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity reg_clear is
    port(clock, reset, ce:in std_logic;
          D:in std_logic_vector(7 downto 0);
          Q:out std_logic_vector(7 downto 0);
end reg_clear;

architecture reg_clear of reg_clear is
begin
  process (clock, reset)
    begin
    if reset='1' then
         Q <= (others=>'0');
    elsif (clock'event and clock='0')then    -- SENSIBILIDADE À BORDA DE DESCIDA
        if ce='1' then
            Q <= D;
```

```
        end if;
    end if;
  end process;
end reg_clear;
```

- Se fôssemos fazer um LATCH, o *process* seria:

```
process (clock, D)
  begin

  if  clock='0' then
          Q <= D;
  end if;

end process;
```

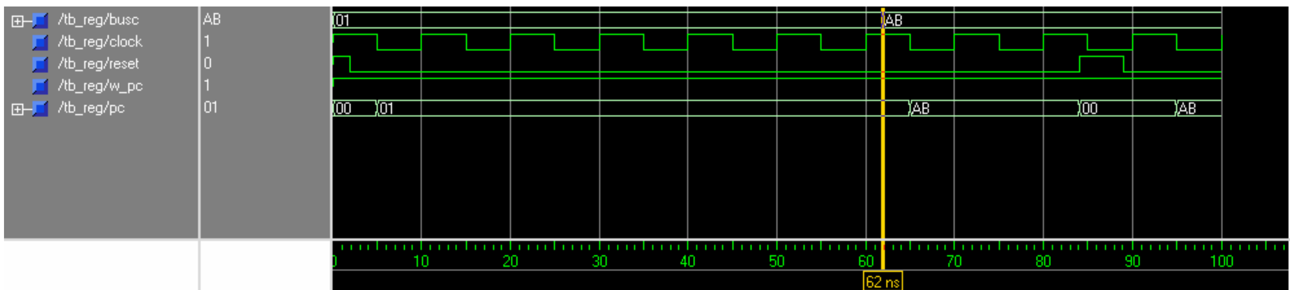
1. sempre que ou o clock ou o D mudarem o processo é ativo. Se o clock estiver em ‘0’ e o D mudar o Q muda ➔ conceito de sensibilidade à nível, e não a borda.
2. LATCHs devem ser evitadas, devido ao seu comportamento “passante”

VAMOS SIMULAR:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;

entity TB_REG  is
end TB_REG;

architecture a1 of TB_REG is
    signal busC, pc: std_logic_vector(7 downto 0);
    signal clock, reset, w_pc: std_logic;
begin

   REG_PC: entity work.reg_clear port map (clock=>clock, reset=>reset,
                ce=>w_pc,  D=>busC, Q=>pc);

   w_pc <=  '1';      -- sempre habilitado

   reset <= '1', '0' after 2 ns, '1' after 84 ns, '0' after 89 ns ;

   -- gera o clock
   process                                      
            begin                               
             clock <= '1', '0' after 5 ns;      }  IMPORTANTE
             wait for 10 ns;                    
   end process;                                 

   busC <= x"01", x"AB" after 62 ns;

end a1;
```

- Neste TB tem muita coisa a explicar, entre os pontos importantes reforçar:

1. Geração de clock com processo periódico
2. Mostrar a geração de w_pc, reset e busC com tempos
3. mostrar o correto funcionamento do circuito:



Mudar o clock para aumentar/diminuir a freqüência, e também mudar o ciclo de serviço (*duty cycle*)

Registradores de estado:

- Este módulo é muito simples. Entram os *flags instantâneos* (da ULA), e em função dos comandos *lnz* e *lcv* os flags são carregados.

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use work.cleo.all;

entity status_flags is
    port(n_inst, z_inst, c_inst, v_inst, lnz, lcv, clock, reset:in std_logic;
       n, z, c, v: out std_logic);
end status_flags;

architecture status_flags of status_flags is
begin
    process(clock,reset)
    begin
        if (reset='1') then
          n <= '0';
          z <= '0';
          c <= '0';
          v <= '0';
        elsif clock'event and clock='0' then
          if lcv='1' then
                                   c <= c_inst;
                                   v <= v_inst;
          end if;
          if lnz='1' then
                                   n <= n_inst;
                                   z <= z_inst;
          end if;
        end if;
    end process;
end status_flags;
```

Aula 3: "PUTTING IT ALL TOGETHER"

- Distribuir aos alunos a descrição do bloco de dados (VHDL) e comparar com a figura do mesmo no início deste documento.

- Um único detalhe a mencionar que ainda não foi dito: o sinal ***hold.*** Este sinal congela o bloco de dados quando está em um. Ele **só** entra no módulo *write_decoder.vhd*, impedindo escritas nos registradores.

- A descrição do bloco de dados é composta basicamente pelos três módulos já vistos: decodificador, ULA e registradores.

- Há um conjunto de atribuições concorrentes utilizadas para conectar os registradores nos barramentos:

```
busB <= mdr     when r_mdr='1' else (others=>'Z');
busB <= ir_int  when r_ir='1'  else (others=>'Z');
busA <= pc      when r_pc='1'  else (others=>'Z');
busA <= ac      when r_ac='1'  else (others=>'Z');
busA <= rs      when r_rs='1'  else (others=>'Z');
```

  Explicar a necessidade dos tri-states para evitar curto, assim como o cuidado que se dever ter no momento de criar o decodificador de leitura para que não hajam simultaneamente dois sinais em '1' para o mesmo barramento.

- Mostrar e comentar sobre o package. Porque um package? Explicar o novo tipo ***microinstrucao***.

```
library IEEE;
use IEEE.Std_Logic_1164.all;

package cleo is
    constant bus_size: integer :=8;
    subtype opcode  is std_logic_vector(2 downto 0);
    subtype internal_bus is std_logic_vector(bus_size-1 downto 0);
    type ram is array (0 to 255) of internal_bus;

    type microinstrucao is record
              w,r,u: opcode;
              ce, rw, lnz, lcv : std_logic;
    end record;

end cleo;
```

- Como ainda não vimos o bloco de controle, deveremos criar um test_bench apenas para validar o bloco de dados:

```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;
use work.cleo.all;

entity TB_BLOCO_DADOS  is
end TB_BLOCO_DADOS;

architecture a1 of TB_BLOCO_DADOS is

    signal clock, reset, hold: std_logic;

       signal uins : microinstrucao;
       signal n,z,c,v : std_logic;
       signal address,datain, dataout, ir: internal_bus;

begin
```

```
DP: entity work.datapath port map (clock=>clock, reset=>reset,
        hold=>hold, address=>address, datain=>datain, dataout=>dataout,
        ir=>ir, uins=>uins, n=>n, z=>z, c=>c, v=>v);
```

Bloco de dados

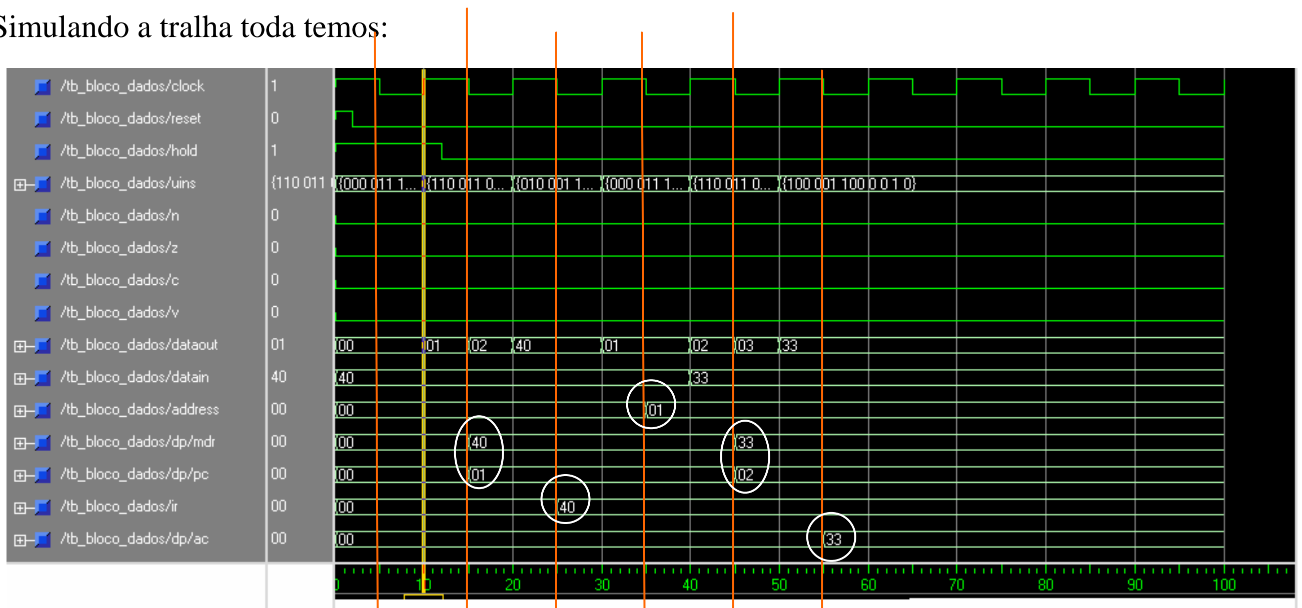
```
-- gera o clock
process
         begin
          clock <= '1', '0' after 5 ns;
          wait for 10 ns;
end process;

reset <= '1', '0' after 2 ns, '1' after 284 ns, '0' after 289 ns ;

hold <=  '1', '0' after 12 ns;      -- sempre em '1'
```

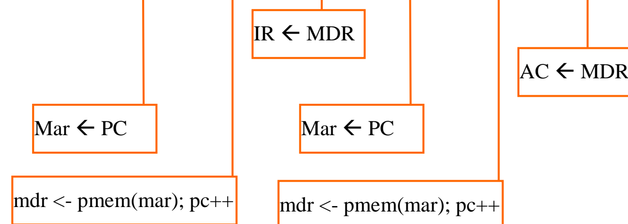
Geração dos sinais básicos

```
process
      begin
         uins <= ("000", "011", "111",'0','0','0','0');  -- mar <- pc
         wait for 10 ns;                                        -- 1 perído

         uins <= ("110", "011", "001",'1','1','0','0');  -- mdr <- pmem(mar); pc++
         wait for 10 ns;

         uins <= ("010", "001", "100",'0','0','0','0');  -- ir  <- mdr
         wait for 10 ns;

         uins <= ("000", "011", "111",'0','0','0','0');  -- mar <- pc
         wait for 10 ns;                                        -- 1 perído

         uins <= ("110", "011", "001",'1','1','0','0');  -- mdr <- pmem(mar); pc++
         wait for 10 ns;

         uins <= ("100", "001", "100", '0','0', '1','0'); -- ac <- mdr; lnz
         wait for 10 ns;

         wait for 10000 ns;       ---- PAROU
end process;

datain <= x"40", x"33" after 40 ns;

end a1;
```

Microinstruções para a microsimulação vista no início da parte do bloco de dados

- Simulando a tralha toda temos:





SUGESTÃO DE EXERCÍCIO: SIMULAR UM ADD EM MODO INDIRETO

Exercícios relativos ao Bloco de Dados

1. Dado o programa abaixo, calcule o número de ciclos de clock consumidos para sua execução.

programa			Solução do problema
	LDA	END1	
	ADD	END2,I	
	STA	END3	
	AND	#0F	
	JZ	FIM,R	
	NOT		
FIM:	HLT		
END1:	DB	#3AH	
END2:	DB	END4	
END3:	DB	#0	
END4:	DB	#16	

Resposta: 8+10+7+6+6+3= 40 ciclos de clock (não executa o NOT)

- Dado que o clock do processador é de 100 MHz, em quanto tempo (segundos) o programa é executado?
- Qual o CPI médio para este programa [introduzir aqui o conceito de ciclos por instrução]?

2. A tabela abaixo ilustra um exemplo de programa a ser executado pelo Processador Cleópatra.

Endereço na memória	Mnemônicos e/ou Dados	Código objeto (preencher)
0	**LDA #**	
1	**90H**	
2	**JN ,R**	
3	**02H**	
4	**ADD #**	
5	**F4H**	
6	**AND #**	
7	**87H**	
...	**...**	

Pede-se:

(a) Preencha o campo correspondente ao código objeto deste trecho de programa.

(b) Compute o número de ciclos de relógio para executar este programa.

(c) Supondo que o relógio seja de 100 MHz, quanto tempo este programa leva para ser executado?

(d) Complete na tabela abaixo o estado dos registradores a cada ciclo de clock e a micro-instrução correspondente para as **3** primeiras instruções executadas pelo programa. Assuma que todos os registradores inicialmente estão carregados com 0.

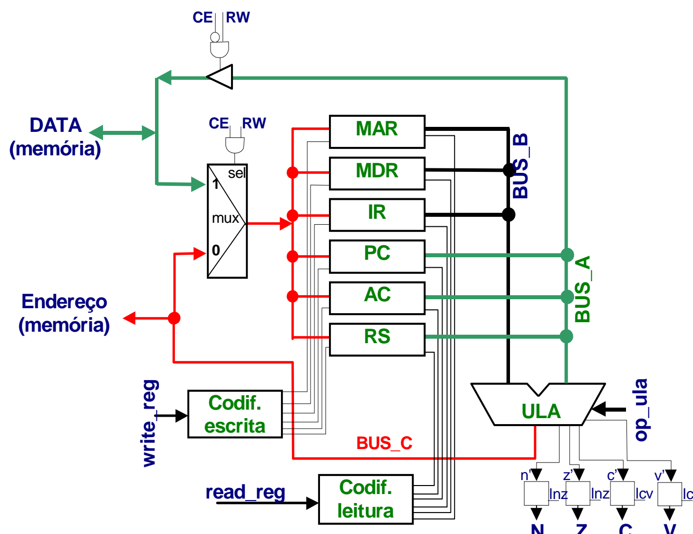
CK	DATA	MAR	MDR	IR	PC	AC	n/z/c/v	*Microinstrução*
0		0	0	0	0	0	0/0/0/0	
1								
2								
3								
...								

3. A arquitetura Cleópatra necessita muitos ciclos de relógio para executar uma determinada instrução. São necessários 3 ciclos para a busca e de 1 a 7 ciclos para a execução de uma instrução. A figura abaixo mostra uma organização alternativa para o Bloco de Dados da arquitetura Cleópatra, tendo o objetivo de tentar reduzir este número de ciclos.

a) Com esta modificação, que ganhos podem ser auferidos em termos de temporização dos ciclos de máquina do processador Cleópatra?

b) Diga quais e quantas seriam as microinstruções necessárias para realizar a instrução STA com modo de endereçamento indireto.

c) E para o LDA modo direto e para o LDA modo indireto?.



4. Na Figura abaixo é proposta uma organização alternativa para o Bloco de Dados da arquitetura Cleópatra. Pede-se, para esta nova organização:

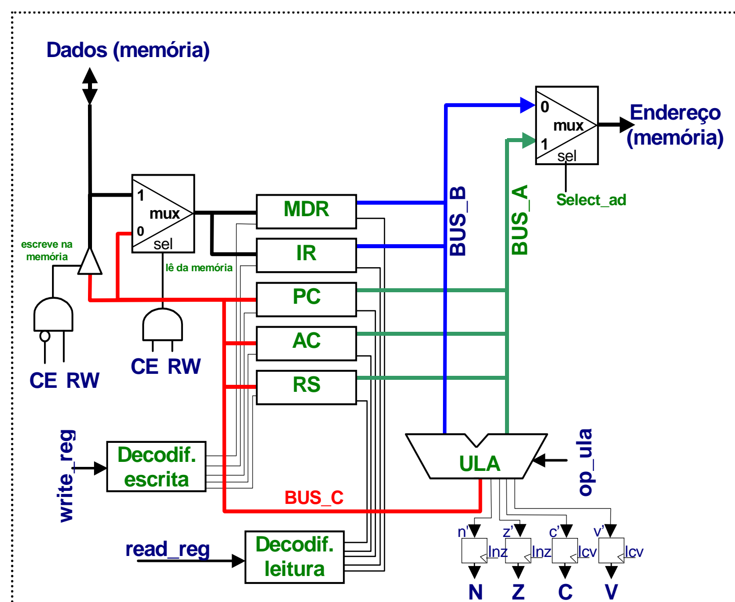
a) Detalhar o ciclo de busca de instruções em microinstruções. Seu detalhamento deve gastar o número <u>mínimo</u> absoluto de ciclos de relógio permitido por esta nova organização.

b) Detalhar o ciclo de execução em microinstruções para as instruções abaixo, consumindo o número <u>mínimo</u> absoluto de ciclos de relógio permitido por esta nova organização:

AND,I – *and*, modo de endereçamento indireto

STA (direto) - *store*, modo de endereçamento direto

JMP #label - *salto*, modo de endereçamento relativo



5. A tabela abaixo, na primeira coluna, ilustra um exemplo de programa a ser executado pelo Processador Cleópatra. Pede-se:

- Preencha a primeira tabela abaixo, gerando as informações correspondentes às 2 colunas (mnemônicos/operandos e código objeto correspondente).
- Supondo que o relógio seja de 100 MHz, quanto tempo este programa (**<u>inteiro</u>**, até a instrução *hlt*) leva para ser executado?
- Faça microssimulação para as primeiras **<u>três</u>** instruções do programa. Assuma que todos os registradores estão inicialmente carregados com 0.

Programa	End. Memória	Mnemônico/ Operando	Código Objeto (em hexadecimal)
`.code`	0		
`LDA #C8H`	1		
`ADD n`	2		
`JN xuxu,r`	3		
`NOT`	4		
`NOT`	5		
`xuxu: HLT`	6		
`.endcode`	7		
`.data`	8		
`n: DB #0C2H` `.enddata`	9		

6. Após a execução de um programa de teste sobre o processador Cleópatra obteve-se o seguinte relatório, relacionando o número de execuções por tipo de instrução:

Tipo de Instrução e Modo de Endereçamento	Número de Execuções no Programa
STA direto	**320**
LDA/Lógico/Aritmética (ADD/OR/AND) direto	**450**
LDA/Lógico/Aritmética (ADD/OR/AND) indireto	**370**
JMP/JZ/JC relativo (considere o pior caso de tempo de execução para os saltos condicionais)	**100**

Pede-se (justifique cada resposta):

a) Número de ciclos para a execução do programa.

b) Dado um clock de 100 MHz, quanto tempo, em segundos, o programa leva para ser executado ?