

Organização de Computadores

VHDL

Linguagem de Descrição de Hardware
(do inglês, “*Hardware Description Language*”)

23/outubro/2003

-- VHDL PARTE 1 --

Apresentação da Linguagem

SUMÁRIO PARTE I

1. Introdução

2. Estrutura de um programa VHDL

3. Elementos primitivos da linguagem VHDL

4. Comandos seqüenciais

5. Funções e procedimentos

6. Estruturas concorrentes

Plano para VHDL

- **Próximas partes**
 - circuitos básicos: de multiplexadores a máquinas de estado
 - estudos de caso: calculadora, comunicação assíncrona e a arquitetura Cleópatra
- **Atividade extra-classe**
 - rodar os exemplos vistos em aula no simulador

Bibliografia

- **BIBLIOTECA**

- Mazor, Stanley; Langstraat, Patricia. “*A guide to VHDL*”. Kluwer Academic Publishers, Boston, 1996. ca250p. [005.133V M476g]
- Ashenden, Peter J. “*The Designers Guide to VHDL*”. Morgan Kaufmann, San Francisco, CA. 1996. 688p.
- Ott, Douglas E.; Wilderotter, Thomas J. “*A designer's guide to VHDL synthesis*”. Kluwer Academic Publishers, Boston, 1996. 306p. [005.133V O89d]
- Bergé, Jean-Michel et al. “*VHDL designer's reference*”. Kluwer Academic Publishers, Dordrecht, 1996. 455p. [005.133V V533v]
- “*IEEE Standard VHDL language : reference manual*”. IEEE Computer Society Press, New York, NY, 1988. 150p. [005.133V I22i]
- Chang, K. C.: “*Digital design and modeling with VHDL and synthesis*”. IEEE Computer Society Press, Los Alamitos, CA, 1997. 345 pp.

- **REDE**

- Ashenden, Peter. “*The VHDL Cookbook*”. **Livro completo, disponível em formato Word.** Disponível na área de download da disciplina, baixado originalmente de <ftp://ftp.cs.adelaide.edu.au/pub/VHDL>
<http://www.inf.pucrs.br/~calazans/undergrad/orgcomp/vhdlcook.exe> (126kb). Disponível

Mais informações sobre VHDL

- **Web sites sobre VHDL e assuntos relacionados**
 - <http://www.stefanvhdl.com/vhdl/htm/index.html>
excelente material para verificação
 - <http://www.sdsmt.edu/syseng/ee/courses/ee741/vhdl-links.html>
site com links diversos para VHDL, incluindo software (gratuito e comercial)
 - <http://www.eda.org/>
padrões de várias HDLs
 - <http://freehdl.seul.org>
simulador VHDL free para LINUX
 - www.esperan.com
empresa de treinamento sobre linguagens em HDLs

Introdução

- **VHDL: Uma linguagem para descrever sistemas digitais**
- **Outras linguagens de descrição de hardware**
 - SystemC, VERILOG, Handel-C, SDL, ISP, Esterel, ... (existem dezenas)
- **Originalmente para especificar hardware, hoje, simulação e síntese, também!**
- **Origem**
 - Linguagem para descrever hardware, no contexto do programa americano “Very High Speed Integrated Circuits” (VHSIC), iniciado em 1980.
 - VHDL → VHSIC Hardware Description Language
 - Padrão IEEE em 1987 (Institute of Electrical and Electronics Engineers), revisado em 1993
 - Linguagem utilizada mundialmente por empresas de CAD (simulação, síntese, propriedade intelectual). Verilog muito usada nos EUA.

Benefícios / Desvantagens

- **Benefícios**

- **Especificação do sistema digital:**

- Projetos independentes da tecnologia (implementação física é postergada)
 - Ferramentas de CAD compatíveis entre si
 - Flexibilidade: re-utilização, escolha de ferramentas e fornecedores
 - Facilidade de atualização dos projetos
 - Permite explorar, em um nível mais alto de abstração, diferentes alternativas de implementação
 - Permite, através de simulação, verificar o comportamento do sistema digital
 - **Nível físico:**
 - Reduz tempo de projeto (favorece níveis abstratos de projeto)
 - Reduz custo
 - Elimina erros de baixo nível
 - **Conseqüência:** reduz “time-to-market” (tempo de chegada de um produto ao mercado)

- **Desvantagens**

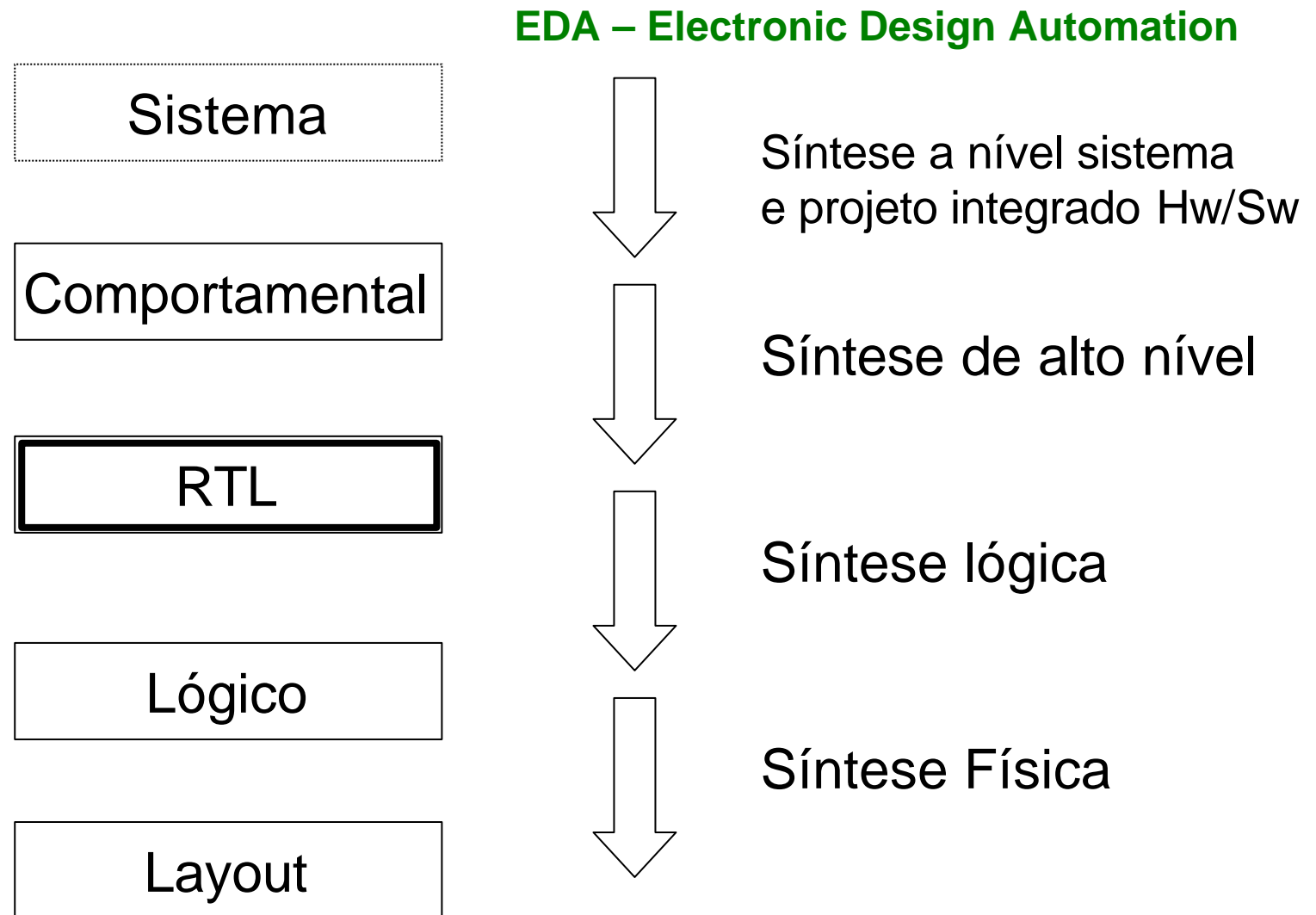
- **Hardware gerado é menos otimizado**
 - **Controlabilidade/Observabilidade de projeto reduzidas**
 - **Falta de pessoal treinado para lidar com a linguagem**

Níveis de abstração (1/3)

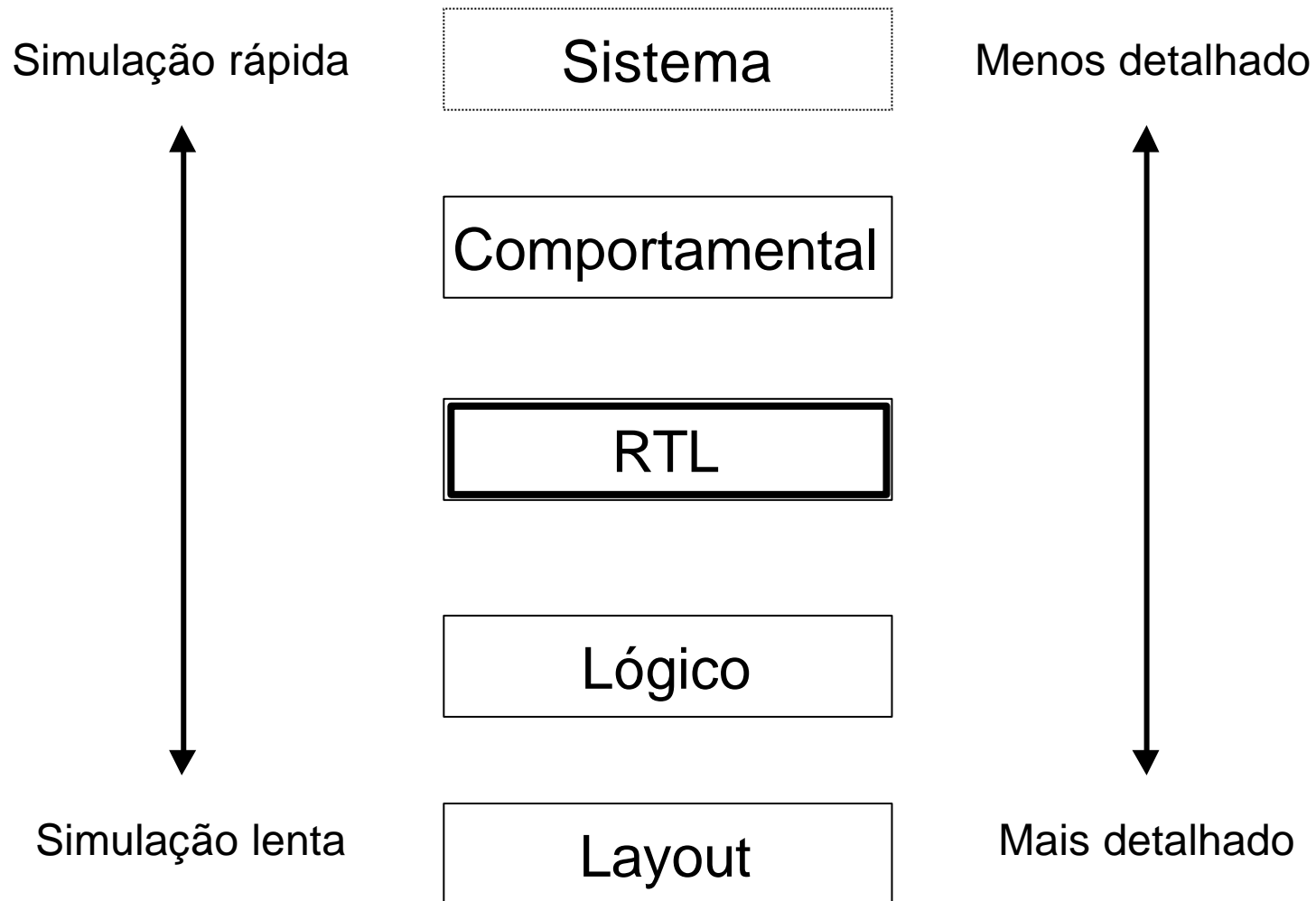
- **Permite descrever hardware em diversos níveis de abstração**
 - Algorítmico, ou Comportamental
 - Transferência entre registradores (RTL)
 - Nível lógico com atrasos unitários
 - Nível lógico com atrasos arbitrários
- **Favorece projeto descendente (“top-down design”)**
 - Projeto é inicialmente especificado de forma abstrata, com detalhamento posterior dos módulos
 - Exemplo : $A \leq B + C$ after 5.0 ns;

A forma de realizar a soma pode ser decidida no momento da implementação (e.g. propagação rápida de vai-um, ou não, paralelo ou série, etc)

Níveis de abstração: ferramentas EDA (2/3)



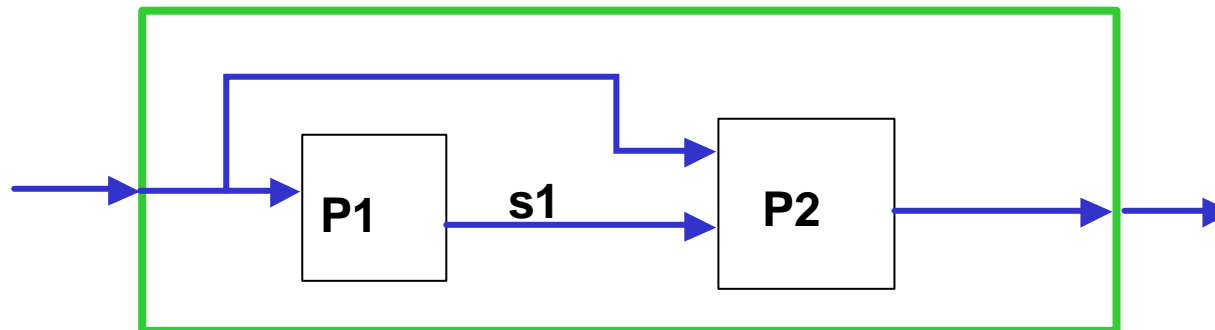
Relação entre os níveis de abstração (3/3)



VHDL é uma linguagem de programação ? (1/3)

- **Quase ...**

- Paralelismo entre componentes de um circuito digital
- Comunicação entre processos paralelos



- Processos P1 e P2 rodam em paralelo (podem ser simplesmente tanto duas portas lógicas, como dois módulos arbitrariamente complexos), com algum sinal sincronizando a comunicação entre eles (ex. S1 na Figura).
- Temporização entre os elementos
- Componentes e instâncias → netlist (descrição estrutural)
- Controle de versões dos módulos através de “configurações”

VHDL é uma linguagem de programação ? (2/3)

- **Relações temporais**

```
A <= B + C after 5.0 ns;           -- Programa para o futuro um evento sinal A!  
D <= A + E;                       -- D recebe o valor antigo de A (antes de B+C)!!
```

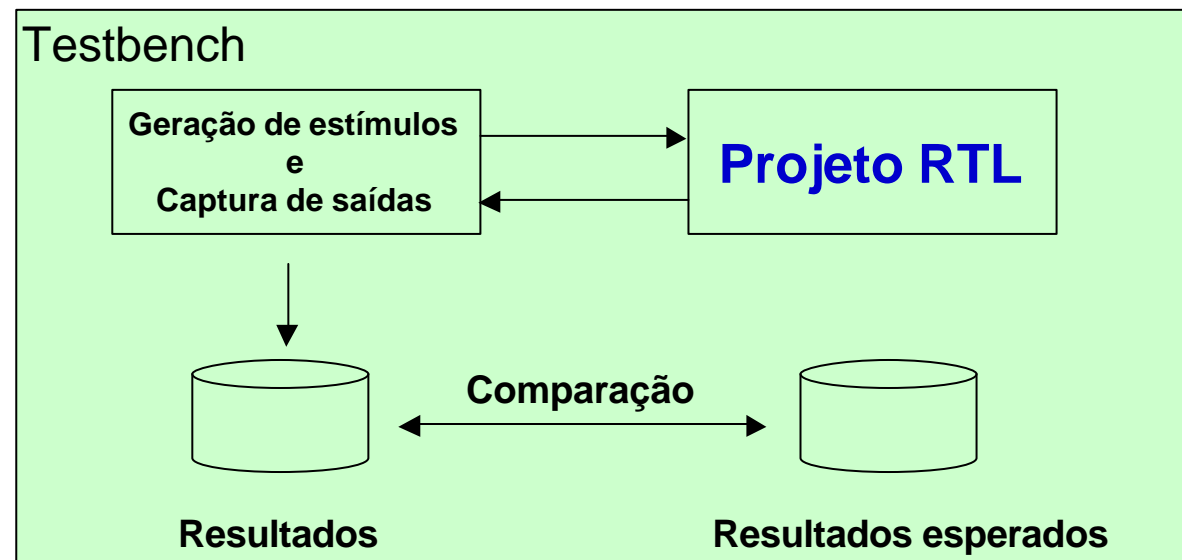
- **Temporização de operações**

```
architecture a1 of meucircuito is   -- o que estas linhas realizam?  
    signal x, y: std_logic;  
begin  
    process(clock, reset)  
    begin  
        if clock'event and clock='1' then  
            x <= y;  
            y <= x;  
        end if;  
    end process;  
end a1;
```

- Sinais: temporizados
- Variáveis: sem temporização – linguagem de programação

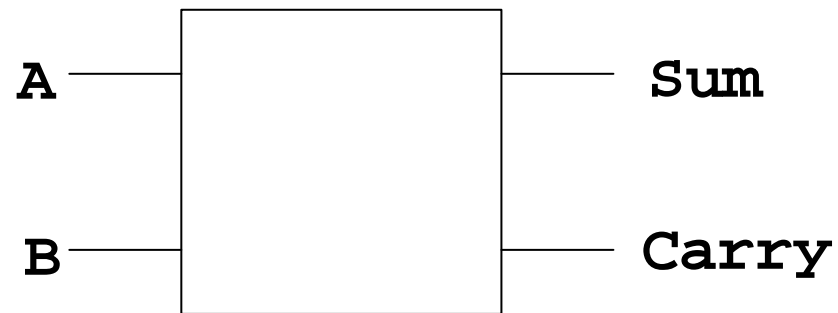
VHDL é uma linguagem de programação ? (3/3)

- **Código é executado em um simulador**
 - não há um compilador de VHDL, não há um código executável
- **Projeto do usuário**
 - especificado no estilo RTL
- **Testbench: descrição VHDL para teste do projeto em desenvolvimento**
 - especificação comportamental
 - interage com o projeto
 - portátil



Descrição de módulos de hardware (1/2)

- **INTERFACE EXTERNA: *entity***
 - Especifica somente a interface
 - Não contém definição do comportamento



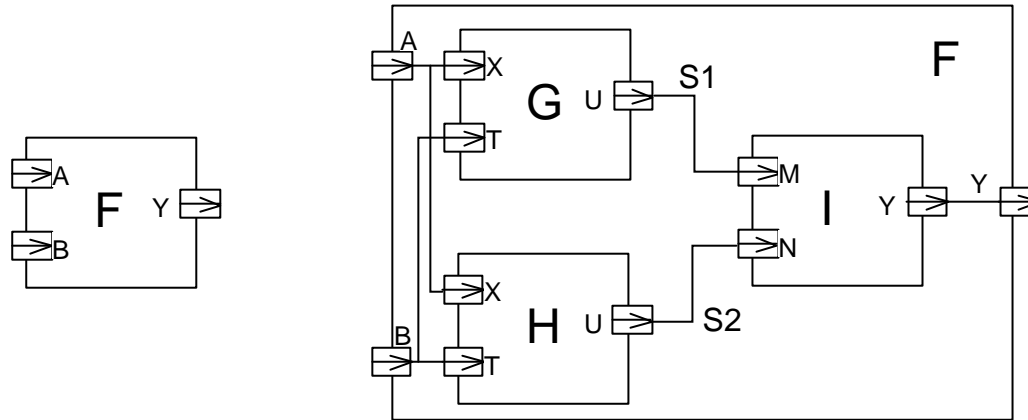
```
entity halfadd is  
port (A, B: in std_logic;  
       Sum, carry: out std_logic);  
end halfadd;
```

Descrição de módulos de hardware (2/2)

- **COMPORTAMENTO : architecture**
 - Especifica o comportamento da *entity*
 - Deve ser associada a uma *entity* específica
 - Uma *entity* pode ter associada a si várias *architecture* (diferentes formas de implementar um mesmo módulo)

```
architecture comp of halfadd is
begin
    Sum <= A xor B;
    Carry <= A and B;
end comp;
```


Descrivendo estrutura (1/3)



- **Módulos compostos por sub-módulos (instâncias) conectados por sinais**
 - Na figura temos o módulo F, composto pelos módulos G,H,I; com entradas A e B; e sinais internos s1 e s2.
- **Em VHDL:**
 - Instâncias: `component` (declaração opcional)
 - Sinais: `signal` (de um determinado tipo)
 - Relação entre sinais e conectores das instâncias: `port map`

Descrivendo estrutura (2/3)

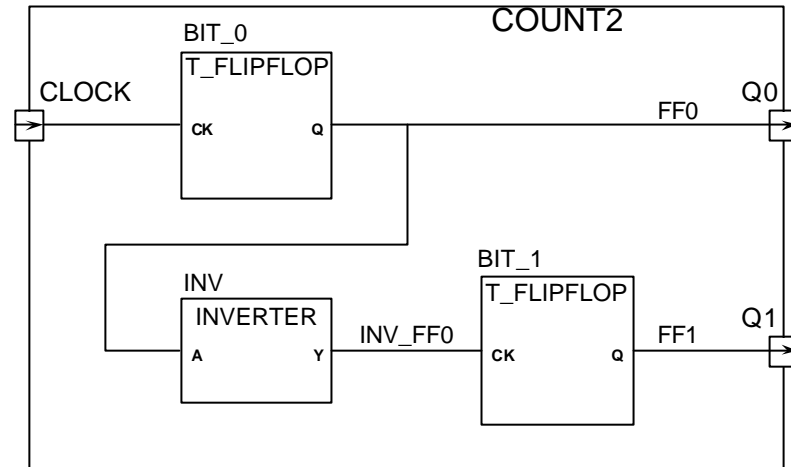
```
library IEEE;  
use IEEE.std_logic_1164.all;
```

```
entity contador is  
    generic(prop_delay : Time := 2 ns);  
    port( clock : in bit; q1, q0 : out bit);  
end contador;
```

Declaração da Interface Externa

```
architecture estrutural of contador is  
    signal ff0, ff1, inv_ff0 : std_logic;  
begin  
    bit_0:  entity work.Tflip_flop  
            port map( ck=> clock, q => ff0);  
    inv:    entity work.inversor  
            port map( a=> ff0, y => inv_ff0);  
    bit_1:  entity work.Tflip_flop  
            port map( ck=> inv_ff0, q => ff1);  
    q0 <= ff0;  
    q1 <= ff1;  
end estrutural;
```

Descrição Estrutural do Módulo



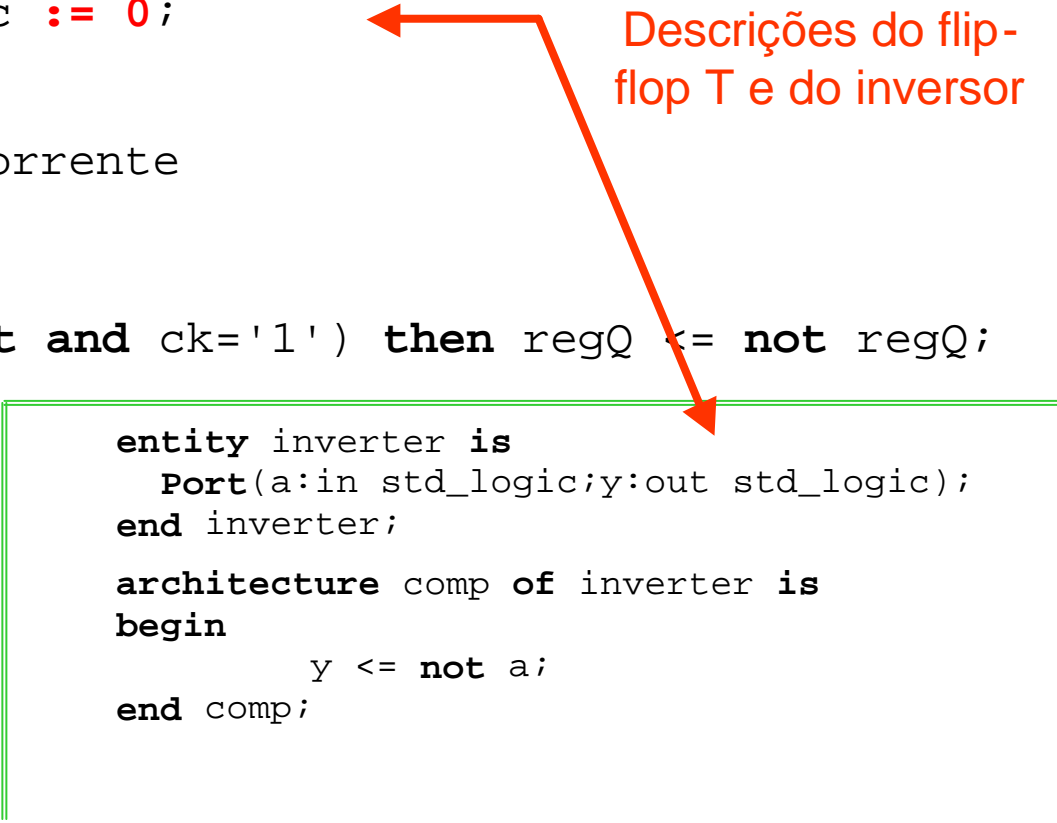
Obs. Notação posicional:

```
bit_0: Tflip_flop port map(clock, ff0);
```

Descrivendo estrutura (3/3)

```
entity Tflip_flop is
    Port(ck : in std_logic; q : out std_logic);
end Tflip_flop;

architecture comp of Tflip_flop is
    signal regQ: std_logic := 0;
begin
    q<=regQ;    -- concorrente
    process (ck)
    begin
        if (ck'event and ck='1') then regQ <= not regQ;
        end if;
    end process;
end comp;
```



```
entity inverter is
    Port(a:in std_logic;y:out std_logic);
end inverter;

architecture comp of inverter is
begin
    y <= not a;
end comp;
```

Descrições do flip-flop T e do inversor

Recomenda-se inicializar o flip-flop com zero, pois não há sinal explícito de reset

Descrevendo comportamento (1/2)

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

```
entity contador is  
    generic(prop_delay : Time := 2 ns);  
    port( clock : in bit; q1, q0 : out bit);  
end contador;
```

Declaração da Interface Externa

```
architecture comportamental of contador is  
    signal q : std_logic_vector(1 downto 0);  
begin  
    count_up: process (clock)  
        variable count_value : integer := 0;  
    begin  
        if clock = '1' then  
            count_value := (count_value + 1) mod 4;  
            q <= CONV_STD_LOGIC_VECTOR(count_value, 2) after prop_delay;  
        end if;  
    end process count_up;  
    q0 <= q(0);  
    q1 <= q(1);  
end comportamental;
```

Sinal

Variável

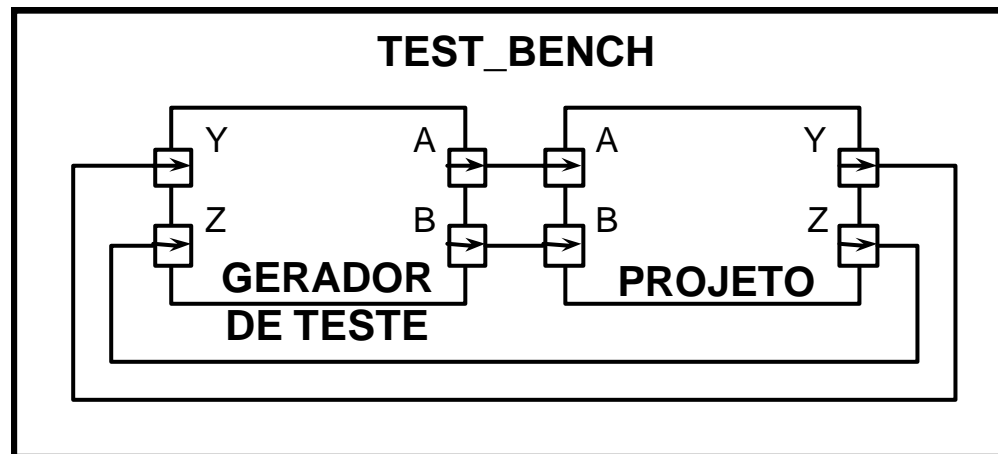
Descrição Comportamental do Módulo

Descrevendo comportamento (2/2)

- **Primitiva de base (concorrência): *process***
- **Observar diferença entre variável e sinal:**
 - **Variável:** interna ao processo, do tipo natural, atribuição **IMEDIATA**
 - **Sinal:** global, com atribuição ao término do processo
- **Notar que na declaração do processo há a variável “clock”**
 - **Significado:** o processo aguarda transição no sinal “clock”

Validação por simulação (1/3)

- Utilizar um circuito de teste: *test_bench*
 - Contém um processo “gerador de teste” e uma instância do projeto
 - O *test_bench* não contém portas de entrada/saída



Validação por simulação (2/3)

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

```
entity tb_cont is  
end tb_cont;
```

} **O test_bench não contém interface externa**

```
architecture teste of tb_cont is
```

```
signal ck, q1, q0 : std_logic;
```

```
begin
```

Seleção da arquitetura



```
c1: entity work.contador(comportamental) port map(clock=>ck, q0=>q0, q1=>q1);
```

} **Instanciação do projeto**

```
-- gerador de clock
```

```
process
```

```
begin
```

```
ck <= '1' after 10ns, '0' after 20ns;
```

```
wait for 20ns;
```

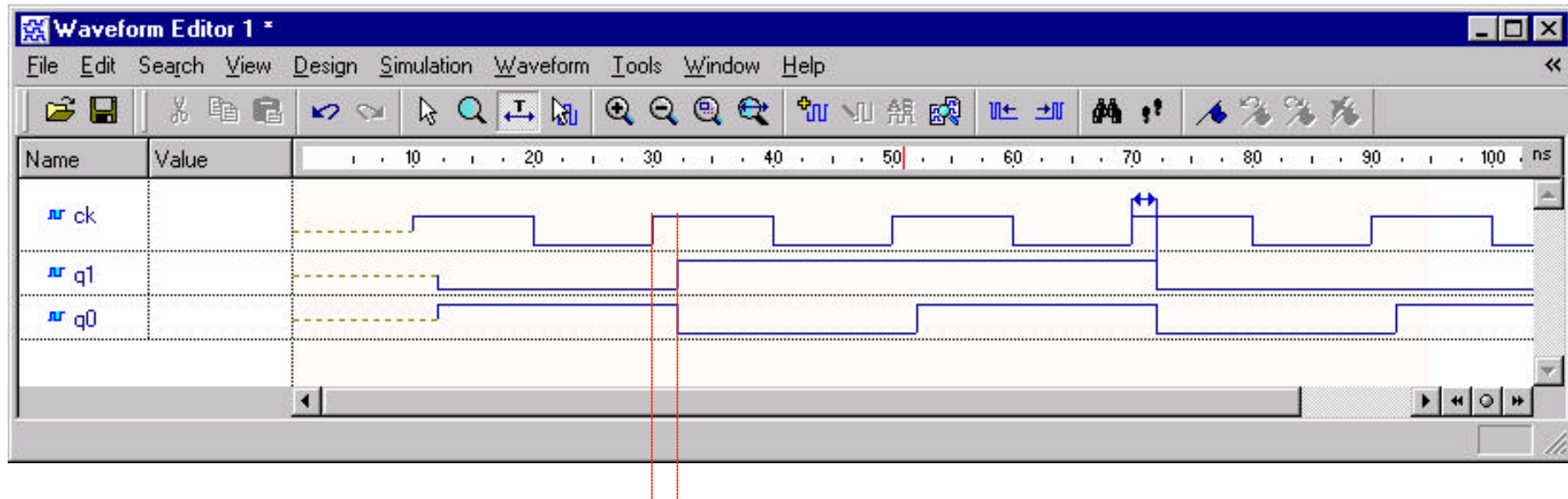
```
end process;
```

} **Geração do clock**

```
end teste;
```

Validação por simulação (3/3)

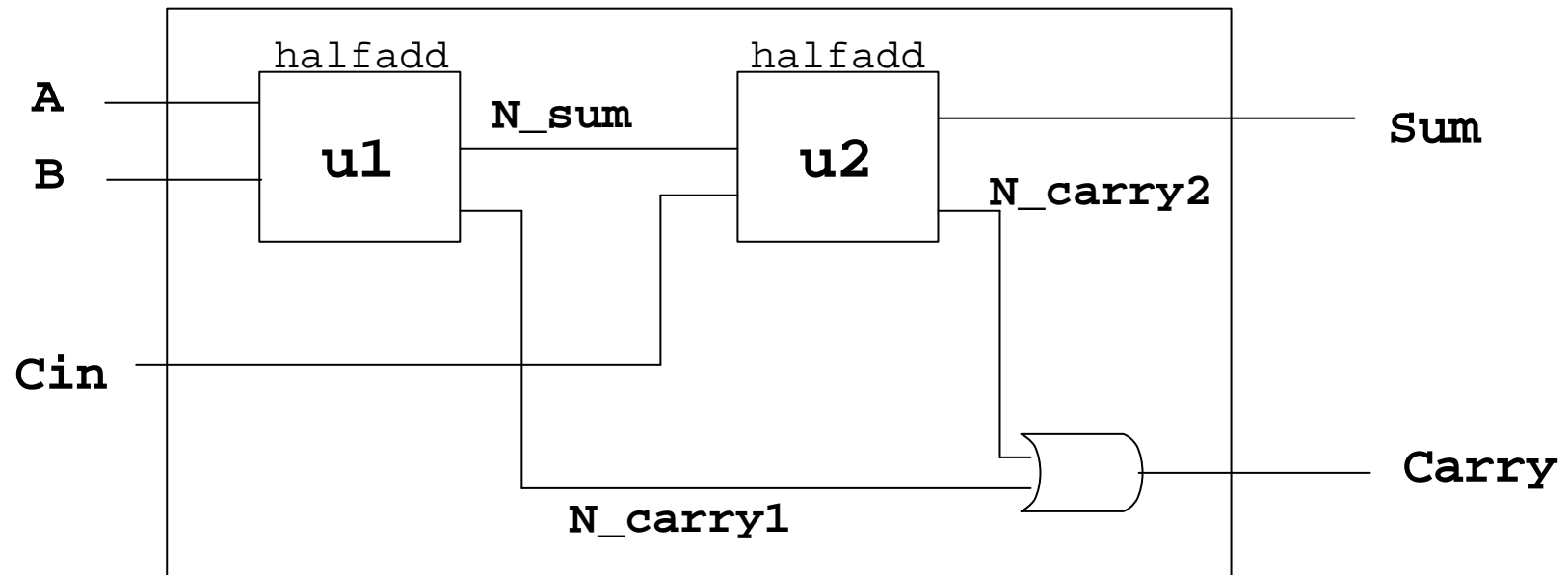
- Resultado da simulação, para implementação comportamental:



Observar o atraso de q0 e q1 em relação ao sinal de clock (2ns)

Exercício

Descrever o sistema *full-adder*, abaixo, em VHDL:



Usar o componente:

```
entity halfadd is
port (A, B: in std_logic;
      Sum, Carry: out std_logic);
end halfadd
```

Solução possível

```
entity fulladd is
port (A, B, Cin: in std_logic;
       Sum, Carry: out std_logic);
end fulladd;

architecture structural of fulladd is
    signal N_sum, N_carry1, N_carry2: std_logic;

begin

    u1: entity work.halfadd port map(A, B, N_sum, N_carry1);

    u2: entity work.halfadd port map(N_sum, Cin, Sum, N_carry2);

    carry <= N_carry2 or N_carry1;

end structural;
```

Resumindo ...

Até agora:

- ☺ Introdução à linguagem - comportamento e estrutura
- ☺ Diferenças em relação à linguagem de programação
- ☺ Simulação com “test_bench”

A seguir:

- ☺ Estrutura de um programa VHDL
- ☺ Tipos primitivos (escalares, objetos, expressões)
- ☺ Exercícios

SUMÁRIO PARTE I

1. Introdução

2. Estrutura de um programa VHDL

3. Elementos primitivos da linguagem VHDL

4. Comandos seqüenciais

5. Funções e procedimentos

6. Estruturas concorrentes

Estrutura de um programa VHDL

Projeto VHDL

Arquivos VHDL

Package:

Declara constantes, tipos de dados, subprogramas.
Objetivo: reutilização de código

Architecture:

define a(s) implementação (ões) do projeto

Entity:

declara as interfaces do projeto (pinos de entrada/saída)

Configuration:

declara qual das arquiteturas será utilizada
(no próprio port map)
ver lâmina 23

Estrutura de um programa VHDL

- Cada módulo tem sua própria “entity” e “architecture”.
- As arquiteturas podem ser descritas tanto no nível comportamental quanto estrutural ou uma mistura disto.
- Toda a comunicação ocorre através das portas declaradas em cada entity, observando-se o tipo, tamanho, se se trata de sinal ou barramento e a direção.
- Várias funções e tipos básicos são armazenados em bibliotecas (library). A biblioteca “IEEE” sempre é incluída.
- Biblioteca do usuário (default): work. Todos os arquivos contidos no diretório de trabalho fazem parte da biblioteca do usuário.

Arquitetura

- A função de uma “entity” é determinada pela sua “architecture”
- Organização:

Architecture
Declarações <i>signal</i> - sinais de comunicação entre processos concorrentes sinais que comunicação entre processos concorrentes e os pinos de E/S <i>type</i> - novos tipos <i>constant</i> - constantes <i>component</i> - componentes (para descrever estrutura) <i>function</i> - Subprogramas (<i>apenas a declaração destes</i>)
Begin (declarações concorrentes)
Atribuição a sinais Chamadas a “functions” e a “procedures” Instanciação de Componentes Processos: descrição de algoritmo
End

Package (1/3)

- Permite a reutilização de código já escrito
- Armazena:
 - Declaração de subprogramas
 - Declaração de tipos
 - Declaração de constantes
 - Declaração de arquivos
 - Declaração de “alias” (sinônimos, por exemplo, para mnemônicos)

```
package minhas_definicoes is
    function max(L, R: INTEGER) return INTEGER;
    type UNSIGNED is array (NATURAL range <>) of STD_ULOGIC;
    constant unit_delay : time := 1ns;
    file outfile :Text is Out "SIMOUT.DAT";
    alias C : Std_Ulogic is grayff (2);
end minhas_definicoes;
```


Package (2/3)

- Um “package” pode ser dividido em duas partes: definição e corpo.
 - Corpo: opcional, detalha especificações incompletas na definição.
- Exemplo completo:

```
package data_types is
  subtype address is bit_vector(24 downto 0);
  subtype data is bit_vector(15 downto 0);
  constant vector_table_loc : address;
  function data_to_int(value : data) return integer;
  function int_to_data(value : integer) return data;
end data_types;

package body data_types is
  constant vector_table_loc : address := X"FFFF00";
  function data_to_int(value : data) return integer is
    body of data_to_int
  end data_to_int;
  function int_to_data(value : integer) return data is
    body of int_to_data
  end int_to_data;
end data_types;
```

Detalhes de implementação omitidos, corpo necessário

Package (3/3)

- Utilização do “package” no programa que contém o projeto:

- Via utilização do prefixo do package

```
variable PC : data_types.address;  
int_vector_loc := data_types.vector_table_loc + 4*int_level;  
offset := data_types.data_to_int(offset_reg);
```

- Via declaração, antes de iniciar a unidade de projeto “entity”, indicação para utilizar todos os tipos declarados em determinado “package”

```
use data_types.all;
```

- Praticamente todos os módulos escritos em VHDL iniciam com:

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

- utilizar a biblioteca IEEE, que contém a definição de funções básicas, subtipos, constantes; e todas as definições dos packages incluídos nesta biblioteca.

SUMÁRIO PARTE I

1. Introdução

2. Estrutura de um programa VHDL

3. Elementos primitivos da linguagem VHDL

4. Comandos seqüenciais

5. Funções e procedimentos

6. Estruturas concorrentes

Elementos primitivos da linguagem VHDL

- VHDL é uma linguagem fortemente tipada
(*integer 1* ¹ *real 1.0* ¹ *bit '1'*)
 - auxilia a detectar erros no início do projeto
 - exemplo: conectar um barramento de 4 bits a um barramento de 8 bits
- Tópicos
 - Escalares / Arrays
 - Objetos
 - Expressões

Escalares / Arrays

- Escalar é o oposto ao array, é um único valor
- Tipos básicos da linguagem VHDL
 - bit / boolean / real / integer / character / physical
- Bit
 - Assume valores '0' e '1'
 - bit não tem relação com o tipo boolean.
 - bit_vector: tipo que designa um conjunto de bits. Exemplo: "001100" ou x"00FF"
- Boolean
 - Assume valores *true* e *false*.
 - Útil apenas para descrições abstratas, onde um sinal só pode assumir dois valores

Escalares / Arrays

– Real

- Utilizado durante desenvolvimento da especificação
- Exemplos: -1.0 / +2.35 / 37.0 / -1.5E+23

– Inteiros

- Exemplos: +1 / 1232 / -1234
- NÃO é possível realizar operações lógicas sobre inteiros (deve-se realizar a conversão explícita)

– Character

- VHDL não é “case sensitive”, exceto para caracteres.
- valor entre aspas simples: ‘a’, ‘x’, ‘0’, ‘1’, ...
- string: tipo que designa um conjunto de caracteres. Exemplo: “vhdl”.

– Physical

- Representam uma medida: voltagem, capacitância, tempo
- Tipos pré-definidos: fs, ps, ns, um, ms, sec, min, hr

Arrays

– Intervalos (range)

- sintaxe: *range valor_baixo to valor_alto*
range valor_alto downto valor_baixo
- integer range 1 to 10 **NÃO** integer range 10 to 1
- real range 1.0 to 10.0 **NÃO** integer range 10.0 to 1.0
- declaração sem range declara todo o intervalo
- declaração range<> : declaração postergada do intervalo

– Enumerações

- Conjunto ordenando de nomes ou caracteres.
- Exemplos:
 type logic_level is ('0', '1', 'X', 'Z');
 type octal is ('0', '1', '2', '3', '4', '5', '6', '7');

Arrays

- **coleção de elementos de mesmo tipo**

 - type word is array (31 downto 0) of bit;

 - type memory is array (address) of word;

 - type transform is array (1 to 4, 1 to 4) of real;

 - type register_bank is array (byte range 0 to 132) of integer;

- **array sem definição de tamanho**

 - type vector is array (integer range <>) of real;

- **exemplos de arrays pré definidos:**

 - type string is array (positive range <>) of character;

 - type std_logic_vector is array (natural range <>) of bit;

- **preenchimento de um array: posicional ou por nome**

 - type a is array (1 to 4) of character;

 - posicional: ('f', 'o', 'o', 'd')

 - por nome: (1 => 'f', 3 => 'o', 4 => 'd', 2 => 'o')

 - valores default: ('f', 4 => 'd', others => 'o')

Declaração e Atribuição de Arrays

```
signal z_bus : std_logic_vector (3 downto 0);  
signal c_bus : std_logic_vector (0 to 3);
```

```
z_bus <= c_bus;
```

```
z_bus(3) ← c_bus(0)  
z_bus(2) ← c_bus(1)  
z_bus(1) ← c_bus(2)  
z_bus(0) ← c_bus(3)
```

```
z_bus(3) <= c_bus(2);
```

Conexão fio a fio

Observação:

- tamanho dos arrays deve ser o mesmo
- elementos são atribuídos por posição, pelo número do elemento

Declaração e Atribuição de Arrays

```
signal a_bus, b_bus, z_bus: std_logic_vector (3 downto 0);
```

```
signal a_bit, b_bit, c_bit, d_bit : bit;
```

```
signal byte: std_logic_vector (7 downto 0);
```

```
z_bus <= (a_bit, b_bit, c_bit, d_bit);
```

```
byte <= (7 => '1', 5 downto 1 => '1', 6 => b_bit, others => '0');
```

Tipo padrão para síntese: `std_logic` (1/3)

– `standard_ulogic`

- enumeração com 9 níveis lógicos
- tipo não “resolvido”

```
-----fonte: stdlogic.vhd-----  
-- logic state system (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  
                    );
```

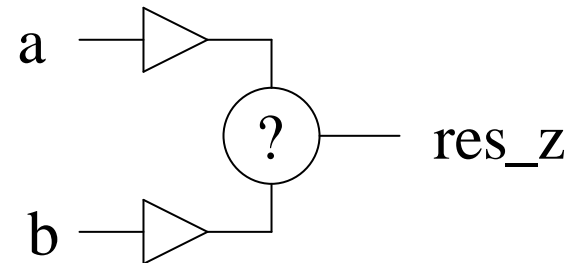
Tipo padrão para síntese: `std_logic` (2/3)

```
signal a, b, c :<tipo pré-definido, não “resolvido”>;  
signal res_z : <tipo_resolvido>;
```

```
z <= a;  
z <= b;
```

X

Erro de “*multiple drivers*”,
ou seja, curto-circuito



```
res_z <= a;  
res_z <= b;
```

Resolução por tabela de resolução

Tipo padrão para síntese: `std_logic` (3/3)

– `standard_logic`

- tipo “resolvido” - permite por exemplo implementação de barramentos
- tipo mais utilizado em descrições de hardware

```
SUBTYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
-----
-- resolution function
-----
CONSTANT resolution_table : stdlogic_table := (
--
-- | U   X   0   1   Z   W   L   H   -   |
--
-- |-----|
-- ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
-- ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
-- ( 'U', 'X', '0', 'X', '0', '0', '0', '0', '0', 'X' ), -- | 0 |
-- ( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
-- ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
-- ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
-- ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
-- ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
-- ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | - |
--
-- );
```

Records (1/2)

- estruturas semelhantes a “struct” em linguagem C, ou “record” em Pascal
- coleção de elementos com **tipos diferentes**

```
type instruction is
  record
    op_code : processor_op;
    address_mode : mode;
    operand1, operand2: integer range 0 to 15;
  end record;
```

- declaração: `signal instrução : instruction;`
- referência a um campo: `instrução.operando1`

Records - exemplo (2/2)

```
type t_packet is record
    byte_id : std_logic;
    parity : std_logic;
    address : integer range 0 to 3;
    data : std_logic_vector(3 downto 0);
end record;

signal tx_data, rx_data: t_packet;
...
rx_data <= tx_data;
tx_data <= ('1', '0', 2, "0101");
tx_data.address <= 3;
```

Objetos: Constantes (1/3)

- nome dado a um valor fixo
- consiste de um nome, do tipo, e de um valor (opcional, com declaração posterior)
- sintaxe:
 - **constant identificador : tipo [:=expressão];**
 - **correto**
constant gnd: real := 0.0;
 - **incorreto**
gnd := 4.5; -- atribuição a constante fora da declaração
- constantes podem ser declaradas em qualquer parte, porém é aconselhável declarar as freqüentemente utilizadas em um package

Objetos: Variáveis (2/3)

- utilizadas em **processos**, sem temporização, atribuição imediata

- sintaxe:

variable identificador (es) : tipo [restrição] [:=expressão];

- exemplo:

variable indice : **integer range** 1 to 50 := 50;

variable ciclo_de_maquina : **time range** 10 ns to 50 ns := 10ns;

variable memoria : **std_logic_vector** (0 to 7)

variable x, y : **integer**;

Objetos: Sinais (3/3)

- Comunicação entre módulos
- Temporizados
- Podem ser declarados em entity, architecture ou em package
- Não podem ser declarados em processos, podendo serem utilizados no interior destes
- Sintaxe:

signal identificador (es) : tipo [restrição] [:=expressão];

- Exemplo

signal cont : integer range 50 downto 1;

signal ground : std_logic := '0';

signal bus : std_logic_vector (5 downto 1);

Expressões (1/2)

– Expressões são fórmulas que realizam operações sobre objetos de mesmo tipo.

- Operações lógicas: and, or, nand, nor, xor, not
- Operações relacionais: =, /=, <, <=, >, >=
- Operações aritméticas: - (unária), abs
- Operações aritméticas: +, -
- Operações aritméticas: *, /
- Operações aritméticas: mod, rem, **
- Concatenação

Menor

PRIORIDADE

Maior



– Questão: o que a seguinte linha de VHDL realiza?

`X <= A <= B`

– E se X, A e B fossem variáveis?

Expressões (2/2)

Observações:

- Operações lógicas são realizadas sobre tipos bit e boolean
- Operadores aritméticos trabalham sobre inteiros e reais. Incluindo-se o package **“use ieee.STD_LOGIC_UNSIGNED.all”** pode-se somar vetores de bits (std_logic_vector)
- Todo tipo físico pode ser multiplicado/dividido por inteiro ou ponto flutuante
- Concatenação é aplicável sobre caracteres, strings, bits, vetores de bits e arrays

Exemplos: “ABC” & “xyz”

resulta em: “ABCxyz”

“1001” & “0011”

resulta em: “10010011”

Resumo de elementos primitivos

- VHDL é uma linguagem fortemente tipada
- Escalares são do tipo:
bit, boolean, real, integer, physical, character.
- Há a possibilidade de se declarar novos tipos: enumeração e record
- Objetos podem ser constantes, variáveis e sinais
- Expressões são fórmulas cujos operadores devem ser exatamente do mesmo tipo

Exercício

Qual/quais das linhas abaixo é/são incorreta/s? Justifique a resposta.

```
variable A, B, C, D : std_logic_vector (3 downto 0);
```

```
variable E,F,G : std_logic_vector (1 downto 0);
```

```
variable H,I,J,K : std_logic;
```

```
[ ] A := B xor C and D ;
```

```
[ ] H := I and J or K;
```

```
[ ] A := B and E;
```

```
[ ] H := I or F;
```

Exercício

Quais linhas abaixo estão incorretas?

```
signal c_bus : std_logic_vector (0 to 3);  
signal a_bus, b_bus, z_bus : std_logic_vector (3 downto 0);  
signal a_bit, b_bit, c_bit, d : std_logic;  
signal byte : std_logic_vector (7 downto 0);  
type t_int_array is array (0 to 3) of integer;  
signal int_array : t_int_array;  
...  
byte <= (others => '1');  
z_bus <= c_bus;  
z_bus <= ('1', b_bit, '0');  
int_array <= "0123";
```

Solução:

```
int_array <= (0, 41, 25, 1);
```

Instalação do simulador

➤ buscar a versão demo na homepage

- excelente documentação VHDL disponível, tutorial Evita, interativo, disponível no mesmo local (versão resumida evita.zip, versão completa evita.exe)
- existem *templates* prontos para comandos VHDL
- existem programas exemplos prontos
- simulador funciona com depuração de código fonte

SUMÁRIO PARTE I

1. Introdução

2. Estrutura de um programa VHDL

3. Elementos primitivos da linguagem VHDL

4. Comandos seqüenciais

5. Funções e procedimentos

6. Estruturas concorrentes

Comandos seqüenciais

- Comandos utilizados em:
 - **processos**
 - **functions / procedures**
- Conjunto de ações seqüenciais, executadas passo a passo. É um estilo de descrição semelhante a outras linguagens de programação.
- Comandos seqüenciais:
 - **atribuição de variáveis**
 - **if**
 - **case**
 - **for**
 - **while**
 - **NULL**

Atribuição de variáveis (1/2)

- Atribuição de variáveis

variable_assignment_statement ::= target := expression ;

target ::= name | aggregate

- Variáveis não passam valores fora do processo na qual foram declaradas, são **locais**. Elas sequer existem fora de um processo.
- As atribuições são seqüenciais, ou seja, a ordem delas importa.
- Exemplo:

```
function conv_to_sdt_vector ( letra:linha_sdh; pos: integer) return nibble is
  variable bin: nibble;
begin
  case (letra(pos)) is
    when '0' => bin := "0000";
    when '1' => bin := "0001";
    ....
    when others => bin := "0000";
  end case;
  return bin;
end conv_to_sdt_vector;
```

Atribuição de variáveis (2/2)

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

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

```
architecture tb of tb is
```

```
    type par is record
        a,b : integer;
    end record;
```

```
    signal rr : par;
begin
```

```
    process
```

```
        variable r,s : par;
```

```
    begin
```

```
        r.a:= 10;          r.b:= 20;          -- inicialização imediata
```

```
        rr.a <= 10;       rr.b <= 20;       -- inicialização após wait
```

```
        (a=>s.b, b=>s.a) := r;              -- posição s.b recebe r.a
```

```
        (b => r.a, a => r.b) := r;          -- (r.b, r.a) := r;
```

```
        wait for 10 ns;
```

```
        (a=>rr.b, b=>rr.a) <= rr;
```

```
    end process;
```

```
end tb;
```

Descreva a funcionalidade do código ao lado, diferenciando atribuição a variáveis e a sinais.

Comando If (1/2)

```
if_statement ::=  
  if condition then  
    sequence_of_statements  
  { elsif condition then          -- 0 ou n ocorrências  
    sequence_of_statements }  
  [ else                          -- 0 ou 1 ocorrência  
    sequence_of_statements ]  
end if ;
```

IMPORTANTE

- teste de borda de subida: if clock'event and clock='1' then ...
- teste de borda de descida: if clock'event and clock='0' then ...
- a seqüência na qual estão definidos os ´ifs´ implica na prioridade das ações.

Exemplos de if (2/2)

- Exemplo onde a atribuição à variável T tem maior prioridade:

```
if (x) then T:=A; end if;  
if (y) then T:=B; end if;  
if (z) then T:=C; end if;
```

⇨ **equivalente** ⇨

```
if (z) then T:=C;  
  elsif (y) then T:=B;  
  elsif (x) then T:=A;  
end if;
```

- Qual a implementação em hardware da seguinte seqüência de comandos ?

```
process(A, B, control)  
begin  
  if( control='1') then  
    Z <= B;  
  else  
    Z <= A;  
  end if;  
end process;
```

Comando Case (1/3)

- Preferível ao *if* se a condição for simples
- Sintaxe:

```
case_statement ::=
    case expression is
        case_statement_alternative
        { case_statement_alternative }
    end case ;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements

choices ::= choice { | choice }

choice ::=
    simple_expression
    | discrete_range
    / element_simple_name
    | others
```

Case (2/3)

```
case element_colour is
  when red =>                                -- escolha simples
    statements for red;
  when green | blue =>                       -- ou
    statements for green or blue;
  when orange to turquoise =>              -- intervalo
    statements for these colours;
end case;
```

```
case opcode is
  when X"00" => perform_add;
  when X"01" => perform_subtract;
  when others => signal_illegal_opcode;
end case
```


Case - construção de máquinas de estado (3/3)

```
process(EA, i)
begin
  case EA is
    when Sidle => PE <= Sfetch;

    when Sfetch => PE <= Sreg;

    when Sreg =>
      if i=halt or i=invalid_instruction then
        PE <= Shalt;

        elsif ((i=jprgr or i=jsrmi) and jump='0') or i=stmsk then
          PE <= Supdatepc;

        else PE <= Salu;


        end if;

    when .....

    when Supdatepc | Sstack | Spop | Swrbk => PE <= Sfetch;

  end case;
end process;
```

type type_state is
(Sidle, Sfetch, Sreg, Shalt, Salu, Sdmem, Supdatepc, Sstack, Spop, Swrbk);



Laços - For (1/3)

- útil para descrever comportamento / estruturas regulares
- o “for” declara um objeto, o qual é alterado somente durante o laço
- internamente o objeto é tratado como uma constante e não deve ser alterado.

```
for item in 1 to last_item loop  
    table(item) := 0;  
end loop;
```

Loop - For (2/3)

- **next:** interrompe a iteração corrente e inicia a próxima

```
outer_loop : loop
  inner_loop : loop
    do_something;
    next outer_loop when temp = 0;
    do_something_else;
  end loop inner_loop;
end loop outer_loop;
```

- **exit:** termina o laço

```
for i in 1 to max_str_len loop
  a(i) := buf(i);
  exit when buf(i) = NUL;
end loop;
```

Loop - For (3/3)

- **Qual a função do laço abaixo ?**

```
function conv (byte : word8) return integer
is
    variable result : integer := 0;
    variable k : integer := 1;
begin
    for index in 0 to 7 loop
        if ( std_logic'(byte(index))='1')
            then result := result + k;
        end if;
        k := k * 2;
    end loop;
    return result;
end conv ;
```

- **Exercício: faça a conversão ao contrário.**

Loops - While

EXEMPLO 1:

```
while index < length and str(index) /= ' ' loop
    index := index + 1;
end loop;
```

EXEMPLO 2:

```
while NOT (endfile(ARQ)) loop
    readline(ARQ, ARQ_LINE);    -- read line of a file
    read(ARQ_LINE, line_arq);
```

```
.....
for w in 1 to 9 loop
    case line_arq(w) is
        when '0' => bin := "0000";
        when '1' => bin := "0001";
        when '2' => bin := "0010";
        .....
        when 'E' => bin := "1110";
        when 'F' => bin := "1111";
        when others => null;
    end case;
end loop;
```

```
.....
```

```
end loop;
```

CASE } **FOR** } **WHILE**

Null

- serve, por exemplo, para indicar “faça nada” em uma condição de case.

case controller_command **is**

when forward => engage_motor_forward;

when reverse => engage_motor_reverse;

when idle => **null**;

end case;

SUMÁRIO PARTE I

1. Introdução
2. Estrutura de um programa VHDL
3. Elementos primitivos da linguagem VHDL
4. Comandos seqüenciais
5. Funções e procedimentos
6. Estruturas concorrentes

Funções e procedimentos

- Simplificam o código, pela codificação de operações muito utilizadas
- Funções e procedures são declaradas entre a entity e o begin, ou no corpo de um determinado package.
- Utilizam os comandos seqüenciais para a execução do programa
- **Procedures:** permitem o retorno de vários sinais, pela passagem de parâmetros.

`mult(A,B, produto);`

- **Functions:** retornam apenas um valor, utilizando o comando return


`produto <= mult(A,B);`

Funções e procedimentos

Exemplo de procedure:

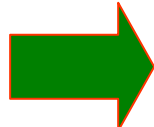
```
procedure mpy (    signal a, b : in std_logic_vector (3 downto 0);
                  signal prod : out std_logic_vector (7 downto 0))
is
    variable p0, p1, p2, p3 : std_logic_vector (7 downto 0);    -- produtos parciais
    constant zero : std_logic_vector := "00000000";
begin
    if b(0) = '1' then    p0 := ( "0000" & a);        else    p0 := zero;    end if;
    if b(1) = '1' then    p1 := ( "000" & a & '0');    else    p1 := zero;    end if;
    if b(2) = '1' then    p2 := ( "00" & a & "00");    else    p2 := zero;    end if;
    if b(3) = '1' then    p3 := ( '0' & a & "000");    else    p3 := zero;    end if;
    prod <= ( p3 + p2 ) + ( p1 + p0 );
end mpy;
```

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



```
package calcHP is  
    subtype ...  
    type ...  
    constant ...  
    procedure somaAB ( signal A,B: in  regsize; signal S:  out regsize);  
end calcHP;
```

A procedure é declarada no package

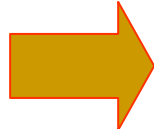


```
package body calcHP is  
    procedure somaAB ( signal A,B: in  regsize; signal S:  out regsize);  
    is  
        variable carry : STD_LOGIC;  
    begin  
        ....  
    end procedure somaAB;  
end package body calcHP;
```

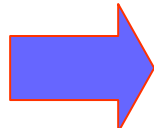
A procedure é implementada no package body

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

Significa: utilizar todas as declarações do package calcHP



```
entity calculadora is  
    port( clock : in bit;  saida : out regsize; flag : out std_logic);  
end;
```



```
architecture rtl of calculadora is  
begin  
    ...  
    somaAB( opA, opB, cin, soma, cout);  
    ...  
end rtl;
```

A procedure é utilizada na architecture

SUMÁRIO PARTE I

1. Introdução
2. Estrutura de um programa VHDL
3. Elementos primitivos da linguagem VHDL
4. Comandos seqüenciais
5. Funções e procedimentos
6. Estruturas concorrentes

Estruturas concorrentes - PROCESS

- Conjunto de ações seqüenciais
- **Wait: suspende** o processo, até que as condições nele incluídas sejam verdadeiras:

wait [sensitivity_clause] [condition_clause] [timeout_clause] ;

sensitivity_clause ::= **on** signal_name { , signal_name }

condition_clause ::= **until** condition

timeout_clause ::= **for** time_expression

- Exemplo:

```
muller_c_2 : process
```

```
  begin
```

```
    wait until a = '1' and b = '1';
```

```
    q <= '1';
```

```
    wait until a = '0' and b = '0';
```

```
    q <= '0';
```

```
  end process muller_c_2 ;
```

- Não são permitidos componentes dentro de processos.

Estruturas concorrentes - PROCESS

- **Sensitivity list:** caso haja uma lista de sinais no início do processo, isto é equivalente a um wait no final do processo
- Havendo sensitivity list no processo, **nenhum** wait é permitido no processo
- Exemplo:

```
process(clock, reset)
begin
    if reset='1' then
        x <= '0';
    elsif clock'event and clock='1' then
        x <= din;
    end if;
end process;
```

Sintaxe do comando PROCESS

```
process_statement ::=
    [ process_label : ]
    process [ ( sensitivity_list ) ]
    process_declarative_part
    begin
    process_statement_part
    end process [ process_label ] ;

process_declarative_part ::= { process_declarative_item }
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration
    | use_clause

process_statement_part ::= { sequential_statement }
sequential_statement ::=
    wait_statement
    | assertion_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

Estruturas concorrentes - ATRIBUIÇÕES (1/2)

ATRIBUIÇÃO DE SINAIS

`alu_result <= op1 + op2;`

ATRIBUIÇÃO DE SINAIS COM ESCOLHA

- fora de processos
- exemplo:

with **alu_function** select

alu_result <= op1 + op2	when alu_add alu_incr ,
op1 – op2	when alu_subtract ,
op1 and op2	when alu_and ,
op1 or op2	when alu_or ,
op1 and not op2	when alu_mask ;

- escreva a atribuição de “alu_function” em um processo com comando case

Estruturas concorrentes - ATRIBUIÇÕES (2/2)

ATRIBUIÇÃO CONDICIONAL DE SINAIS

- fora de processos
- construção é análoga a um processo com sinais na *sensitivity list* e um “if-then-else”

```
mux_out <=  in_2'  when h = '1' and sel="00" else
            in_0   when h = '1' and sel="01" else
            'Z';
```

- escreva a atribuição de “mux_out” em um processo com *if-then-else*

-- VHDL PARTE 2 --

*Circuitos básicos e
representação em VHDL*

VHDL

Circuitos básicos e representação em VHDL

- ❑ **Exemplos de circuitos combinacionais**
 - Codificador
 - Decodificador / Codificador
 - Comparadores
 - Geradores de paridade
 - Multiplexador
 - Somador / Subtrator
 - ULA
 - Multiplicadores / Divisores
 - PLAs
- ❑ **ROM**
- ❑ **RAM**
- ❑ **Exemplos de circuitos seqüenciais**
 - Registradores (deslocamento, carga paralela, acumulador, serial-paralelo)
 - Contadores (binário, BCD, Johnson, Gray / up, down, up-down)
 - Máquina de Estados
 - Geradores de clock
 - Seqüenciadores

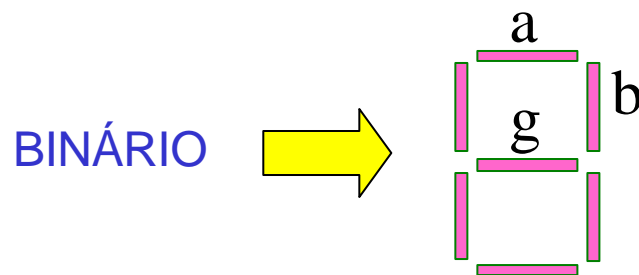
CODIFICADOR

- Em um codificador a saída é uma função combinacional da entrada.
- O comando 'with' é utilizado para atribuir um dado valor a um sinal, em função de um sinal de controle.
- O exemplo abaixo ilustra um codificador BCD para sete segmentos.
- **Relacione o estado dos 7 segmentos 'DISPB' com o estado do número binário 'showb'**

with showb select

DISPB <=

```
"0000001" when "0000",  
"1001111" when "0001",  
"0010010" when "0010",  
"0000110" when "0011",  
"1001100" when "0100",  
"0100100" when "0101",  
"0100000" when "0110",  
"0001111" when "0111",  
"0000000" when "1000",  
"0001100" when "1001",  
"0001000" when "1010",  
"1100000" when "1011",  
"0110001" when "1100",  
"1000010" when "1101",  
"0110000" when "1110",  
"0111000" when "1111";
```



CODIFICADOR COM PRIORIDADE

– Codificador com prioridade

- Em um codificador com prioridade se o bit menos significativo for '1' a saída é '0', se o bit seguinte for 1, independentemente do anterior, a saída é '1'; e assim sucessivamente.
- Exemplo (s(3) tem maior prioridade) :

```
Y <= "11"   when s(3) = '1'   else  
      "10"   when s(2) = '1'   else  
      "01"   when s(1) = '1'   else  
      "00";
```

Importante haver condição *default* em atribuições e estruturas similares:
NÃO HAVENDO ESTA CONDIÇÃO IMPLICA EM HAVER MEMORIZAÇÃO DO SINAL - diferente de software! (warning *latch infered*)

DECODIFICADOR

- O decodificador é utilizado basicamente para acionar uma saída em função de um determinado endereço
- Mesma construção que o codificador
- Exemplo para um decodificador 3→8

with endereço **select**

```
saída <= "00000001" when "000",  
         "00000010" when "001",  
         "00000100" when "010",  
         "00001000" when "011",  
         "00010000" when "100",  
         "00100000" when "101",  
         "01000000" when "110",  
         "10000000" when "111";
```

- Como fica o codificador para escrita dos registradores do bloco de Dados da Cleópatra?

MULTIPLEXADOR (1/2)

- Em um multiplexador uma dentre várias entradas é colocada na saída em função de uma variável de controle.
- Os comando de seleção (índice de array, if, case) são na maioria das vezes implementados com multiplexadores.

```
(a)      architecture A of nome_da_entidade is
          begin
              OUTPUT <= vetor(índice)
          end A

(b)      process(A, B, control)
          begin
              if( control='1')      then      Z <= B;
                                      else      Z <= A;
          end if;
          end process;
```

MULTIPLEXADOR (2/2)

(c) process(A, B, C, D, escolha)
begin

 case escolha is

 when IS_A => Z<=A;

 when IS_B => Z<=B;

 when IS_C => Z<=C;

 when IS_D => Z<=D;

 end case;

end process;

(d) with IntCommand select

 MuxOut <= InA when 0 | 1,

-- OU

 InB when 2 to 5,

-- intervalo

 InC when 6,

 InD when 7,

 'Z' when others;

-- default

SOMADOR (1/4)

- Utilizar para soma/subtração a operação '+'/ '-' entre dois operandos de mesmo tipo.
- O pacote IEEE permite a soma entre std_logic_vector, via redefinição do operador '+'. Incluir:

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


SOMADOR (2/4)

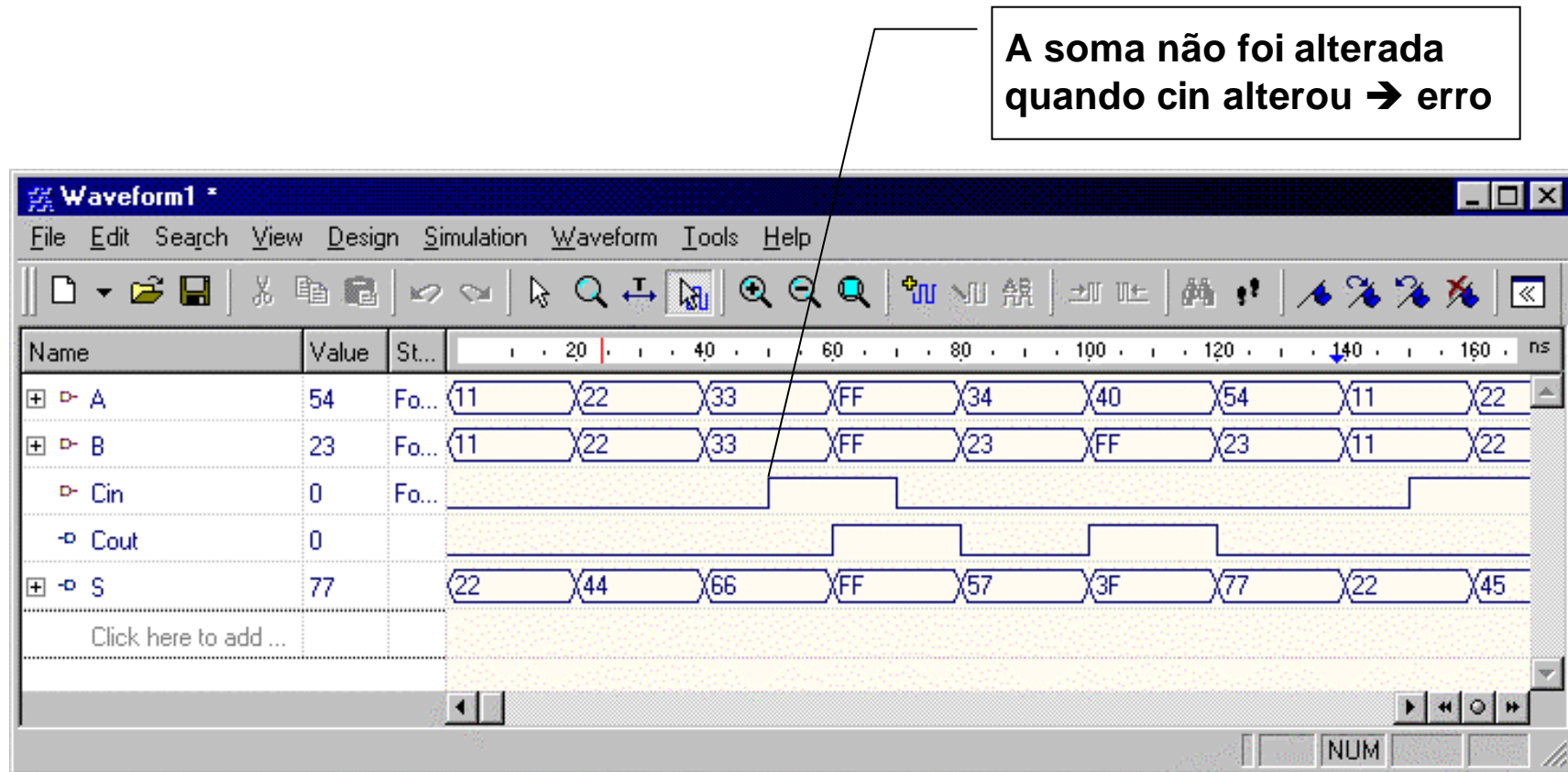
- Exemplo de implementação estrutural em um laço (loop)

```
architecture somador of somador is
begin
  realiza_soma : process(A,B)
    variable carry : STD_LOGIC;
    begin
      for w in 0 to 7 loop
        if w=0 then carry:=cin; end if;
        S(w) <= A(w) xor B(w) xor carry;
        carry := (A(w) and B(w)) or (A(w) and carry) or (B(w) and carry);
      end loop;
      cout <= carry;
    end process;
end somador;
```

- 1) A ordem dentro do for é importante ?
- 2) Qual é a entity desta arquitetura?
- 3) Quando o processo realiza_soma é executado?
- 4) Porque a variável carry é necessária ? Não daria para utilizar o sinal Cout?
- 5) O Cin deveria ou não estar na lista de variáveis do process ? Por quê ?

SOMADOR (3/4)

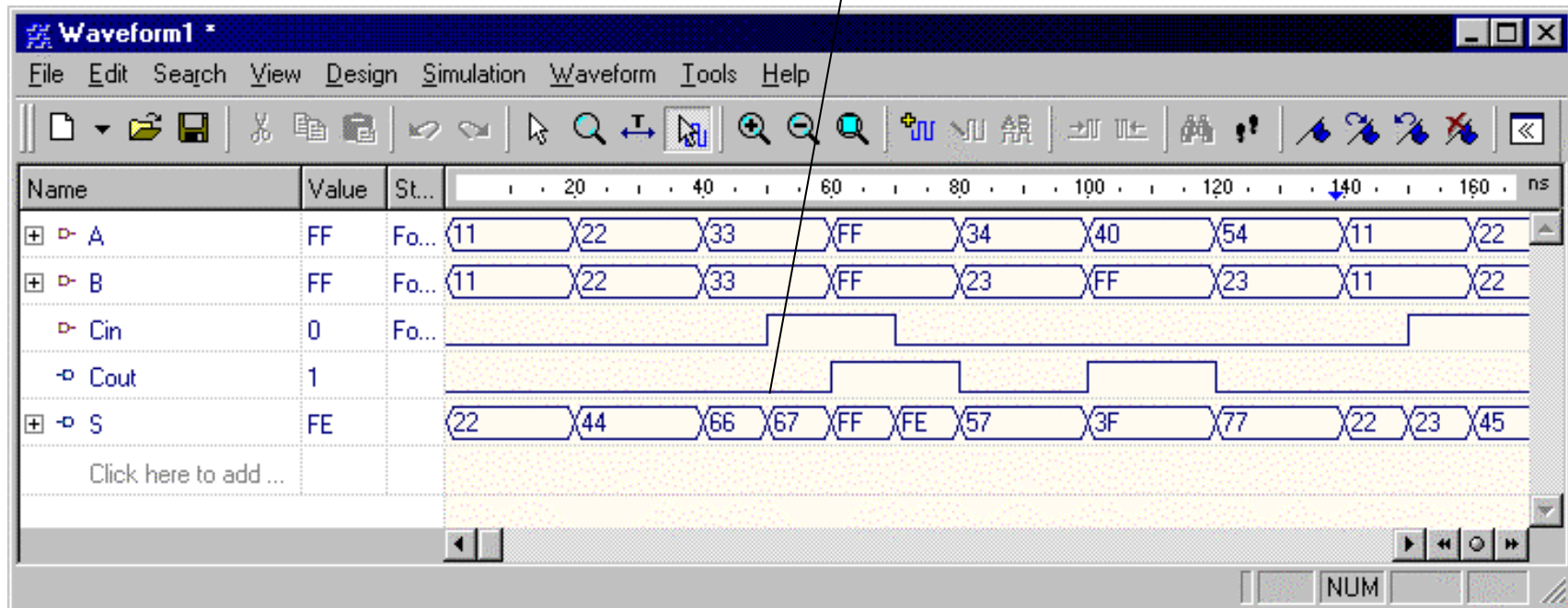
- Simulação incorreta, quando o cin não está incluído na lista de sensibilidade



SOMADOR (4/4)

- Simulação correta, quando o Cin está incluído na lista de sensibilidade

A soma É alterada quando cin altera → OK



Unidade Lógico Aritmética - ULA (1/2)

– Implementação 1:

Utilização de atribuição para selecionar a saída:

<code>outalu_int <= opB</code>	<code>when op_alu=st</code>	<code>else</code>
<code>opA</code>	<code>when op_alu=mov</code>	<code>else</code>
<code>opA and opB</code>	<code>when op_alu=and_i</code>	<code>else</code>
<code>opA or opB</code>	<code>when op_alu=or_i</code>	<code>else</code>
<code>opA xor opB</code>	<code>when op_alu=xor_i</code>	<code>else</code>
<code>opB(15 downto 8) & opA(7 downto 0)</code>	<code>when op_alu=ldli</code>	<code>else</code>
<code>opA - opB</code>	<code>when op_alu=sub</code>	<code>else</code>
<code>not opA</code>	<code>when op_alu=not_i</code>	<code>else</code>
<code>opA(14 downto 0) & '0'</code>	<code>when op_alu=sl</code>	<code>else</code>
<code>'0' & opA(15 downto 1)</code>	<code>when op_alu=sr</code>	<code>else</code>
<code>opB + apA;</code>		

Unidade Lógico Aritmética - ULA (2/2)

Implementação 2: via utilização de *process*

```
process(M,cin,OPCODE,OPERA,OPERB)
begin
```

```
  if (M='1') then                                -- modo 1 é lógico
    case OPCODE is
      when "0000" => saida <= not(OPERA);
      when "0001" => saida <= not(OPERA and OPERB);
      when "0010" => saida <= (not(OPERA)) or OPERB;
      when "0011" => saida <= "0001";
      ..... continuem as outras operações
    end case;
```

```
  else                                           -- modo 0 é aritmético
    case OPCODE is
      when "0000" =>      tempA <= OPERA;          tempB <= OPERB;
      when "0001" =>      tempA <= not OPERA;      tempB <= OPERB;
      when "0010" =>      tempA <= OPERA;          tempB <= "1111";
      ..... continuem as outras operações
    end case;
```

```
    SUM(tempA, tempB, cin, saida, C4);
```

```
  end if;
```

```
end process;
```

Por que na na parte aritmética, utilizou-se apenas um somador, após a seleção dos operandos?

REGISTRADOR (1/4)

- registradores são basicamente sinais declarados em processos com sinal de sincronismo (exemplo: clock). Para efeito de síntese e simulação, é aconselhável introduzir um reset assíncrono.

```
process (clock, reset)
begin
  if reset = '1' then
    reg <= (others => '0');           -- portátil;
  elsif clock 'event and clock='1' then
    reg <= barramento_A;
  end if;
end process;
```

- 1) Como introduzir um sinal de “enable” no registrador, para habilitar a escrita?
- 2) Como implementar um registrador “tri-state” controlado por um sinal “hab”?

REGISTRADOR (2/4)

- Registrador com largura de palavra parametrizável, com 'ce':

```
library ....
```

```
entity regnbit is
```

```
  generic(N : integer := 16);
```

```
  port( ck, rst, ce : in std_logic;
```

```
        D : in STD_LOGIC_VECTOR (N-1 downto 0);
```

```
        Q : out STD_LOGIC_VECTOR (N-1 downto 0) );
```

```
end regnbit;
```

```
architecture regn of regnbit is
```

```
begin
```

```
  process(ck, rst)
```

```
  begin
```

```
    if rst = '1' then
```

```
      Q <= (others => '0');
```

```
    elsif ck'event and ck = '0' then
```

```
      if ce = '1' then
```

```
        Q <= D;
```

```
      end if;
```

```
    end if;
```

```
  end process;
```

```
end regn;
```

generic
define um
parâmetro do
módulo

Uso:

```
rx: regnbit generic map(8)
```

```
  port map(ck => ck, rst => rst, ce => wen,  
           D => RD, Q => reg);
```

REGISTRADOR (3/4)

- exemplo de registrador de deslocamento:

```
process (clock, reset)
begin
    if reset = '1' then
        A <= 0;   B <= 0;   C <= 0;
    elsif clock'event and clock='1' then
        A <= entrada;
        B <= A;
        C <= B;
    end if;
end process;
```

- 1) Desenhe o circuito acima utilizando flip-flops
- 2) A ordem das atribuições (A,B,C) é importante ? O que ocorreria se fosse uma linguagem de programação tipo C?
- 3) Escreva o código para um registrador com deslocamento à esquerda e a direita

REGISTRADOR (4/4)

- Atribuição dentro/fora de process:

```
process (clock, reset)
begin
    if clock'event and clock='1' then
        A <= entrada;
        B <= A;
        C <= B;
        Y <= B and not (C);           -- dentro do process
    end if;
end process;
X <= B and not (C);                 -- fora do process
```

Qual a diferença de comportamento nas atribuições à X e a Y?

- Conclusão:
 - sinais atribuídos em processos, com controle de clock, serão sintetizados com flip-flops.
 - Sinais fora de processos ou em processos sem variável de sincronismo (clock) serão sintetizados com lógica combinacional.

CONTADOR (1/3)

```
entity contup is
  port (
    clock, reset, Load, Enable: In std_logic;
    DATABUS : In Std_logic_Vector (5 downto 0);
    Upcount2 : Out Std_logic_Vector (5 downto 0));
end contup;
```

```
architecture RTL of contup is
  Signal Upcount : std_logic_Vector (5 downto 0);
begin
  Upcount2 <= Upcount;

  Upcounter : Process (clock, reset)
  begin
    if reset = '1' then
      Upcount <= "000000";
    elsif clock'event and clock='1' then
      if ENABLE = '1' then
        if LOAD = '1' then Upcount <= DATABUS;
        else Upcount <= Upcount + 1;
        end if;
      end if;
    end if;
  end process Upcounter;
end RTL;
```

- (1) Determine o comportamento deste contador, fazendo um diagrama de tempos.
- (2) O reset é prioritário em relação ao clock? Por quê?
- (3) Como modificar o contador para realizar contagem crescente/decrescente?

CONTADOR (2/3)

- Código gray: seqüência onde de um estado para outro há apenas a variação de um bit:
000 → 001 → 011 → 010 → 110 → 111 → 101 → 100 → 000 → ...
- Uma forma de implementar este código, que não apresenta uma seqüência regular, é utilizar uma técnica tipo “máquina de estados”, onde em função do estado atual do contador, determina-se o próximo estado.

```
architecture RTL of graycounter is
    signal clock, reset : std_logic; signal graycnt : std_logic_vector (2 downto 0);
begin
    gray : process (clock,reset)
    begin
        if reset = '1' then    graycnt <= "000";           -- reset assíncrono
        elsif clock'event and clock='1' then
            case graycnt is
                when "000" => graycnt <= "001";
                when "001" => graycnt <= "011";
                when "010" => graycnt <= "110";
                when "011" => graycnt <= "010";
                when "100" => graycnt <= "000";
                when "101" => graycnt <= "100";
                when "110" => graycnt <= "111";
                when "111" => graycnt <= "101";
                when others => null;
            end case; end if;
        end process gray;
    end RTL;
```

(1) Implemente um contador JOHNSON utilizando esta técnica. Algoritmo para n bits: bit(i+1) <= bit(i) e bit(0) <= not bit(n-1)

CONTADOR (3/3)

- Outra forma de implementar o contador JOHNSON, é utilizando um registrador de deslocamento:

```
if reset = '1' then
```

```
    john <= "000";
```

```
elseif clock'event and clock='1' then
```

```
    john <= john(1 downto 0) & not (john(2));
```

```
end if;
```

-- CONCATENAÇÃO

ROM (1/4)

– ROM → conjunto de constantes escolhidas por um endereço

- observação: ROMs são implementadas com portas lógicas nas ferramentas de síntese lógica.
- exemplo: aplicação na síntese de um contador com estados não consecutivos (13 estados: 12, 12, 4, 0, 6, 5, 7, 12, 4, 0, 6, 5, 7)

```
package ROM is -- definição de uma rom 13x4
  constant largura : integer := 4;
  subtype palavra is std_logic_vector(1 to largura);
  subtype tamanho is integer range 0 to 12;
  type mem_rom is array (0 to 12) of palavra;
  constant ROM1 : mem_rom := "1100", "1100","0100","0000", "0110","0101","0111",
    "1100","0100","0000","0110","0101","0111");
end ROM;
```



- (1) Como implementar uma RAM ?
- (2) Como inicializar uma RAM ?

ROM (2/4)

– Módulo contador

```
use work.ROM.all;
```

```
entity contador is
```

```
    port( clock, reset : in bit;  
          waves : out palavra);
```

```
end;
```

```
architecture A of contador is
```

```
    signal step : tamanho := 0;
```

```
begin
```

```
    waves <= ROM1(step);
```

-- conteúdo da ROM na saída

```
    process
```

```
        begin
```

```
            wait until clock'event and clock='1';
```

```
            if reset='1' then
```

```
                step <= 0;
```

-- primeiro estado

```
            elsif step = tamanho'high then
```

```
                step <= tamanho'high;
```

-- tranca !

```
            else
```

```
                step <= step + 1;
```

-- avança 1 passo

```
            end if;
```

```
        end process;
```

```
end A;
```

(1) Observe que utilizou-se o atributo *high* para especificar o limite superior do tipo.

(2) O que fazer para a contagem tornar-se cíclica? [Atributo low]

ROM (3/4)

– Simulação do contador utilizando a ROM:

```
entity rom_tb is  
end rom_tb;
```

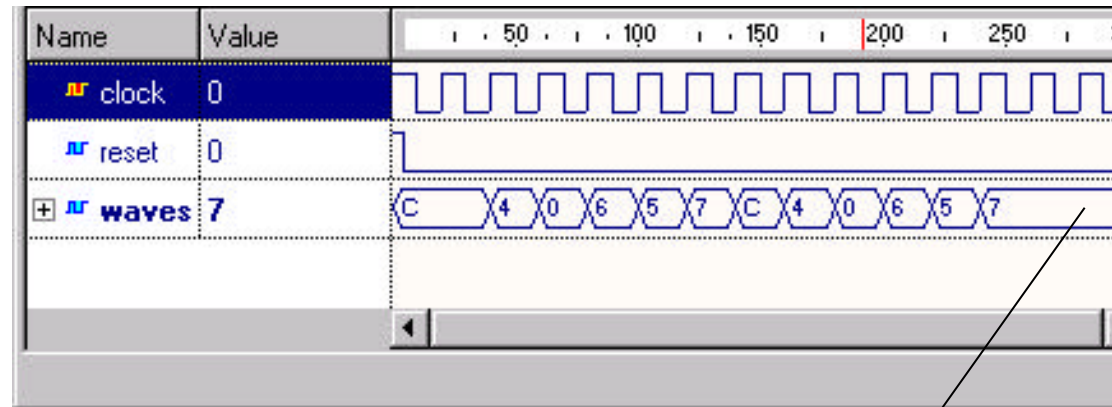
```
architecture t1 of rom_tb is  
    signal clock, reset: std_logic;  
    signal waves: palavra;  
begin
```

```
    UUT : entity work.contador  
        port map (clock => clock, reset => reset, waves => waves );
```

```
    reset <= '1', '0' after 5 ns;
```

```
    process  
    begin  
        clock <= '1', '0' after 10 ns;  
        wait for 20 ns;  
    end process;
```

```
end t1;
```



Observar que tranca no último estado, só saindo com reset

ROM (4/4)

– Técnica muito útil para test bench

```
control : process
  variable contador : integer := 0;
  constant rom : mem_rom := mem_rom'( "0101", "1111", "1010", "1001", "0111", "1011", "0010",
    "0001", "1101", "1111", "1110", "0001", "0111", "0011", "0010", "1001", others=>"0000");
  begin

    wait until reset'event and reset='0';

    -- envia 16 palavras de 4 bits, ou seja, 4 palavras de 16 bits
    for i in 0 to 15 loop
      entrada <= rom(contador);
      contador := contador + 1;
      receive <= '1' after delay;
      wait until acpt='1';
      receive <= '0' after delay;
      wait until acpt='0';
    end loop;

    .....
  end process;
```


MÁQUINA DE ESTADOS (1/2)

```
entity MOORE is port(X, clock : in std_logic; Z: out std_logic); end;
```

```
architecture A of MOORE is
```

```
    type STATES is (S0, S1, S2, S3);           -- tipo enumerado
```

```
    signal scurrent, snext : STATES;
```

```
begin
```

```
    controle: process(clock, reset)
```

```
    begin
```

```
        if reset='1' then
```

```
            scurrent <= S0;
```

```
        elsif clock'event and clock='1' then
```

```
            scurrent <= snext;
```

```
        end if;
```

```
    end process;
```

```
    combinacional: process(scurrent, X)
```

```
    begin
```

```
        case scurrent is
```

```
            when S0 => Z <= '0';
```

```
                if X='0' then snext<=S0; else snext <= S2; end if;
```

```
            when S1 => Z <= '1';
```

```
                if X='0' then snext<=S0; else snext <= S2; end if;
```

```
            when S2 => Z <= '1';
```

```
                if X='0' then snext<=S2; else snext <= S3; end if;
```

```
            when S3 => Z <= '0';
```

```
                if X='0' then snext<=S3; else snext <= S1; end if;
```

```
        end case;
```

```
    end process;
```

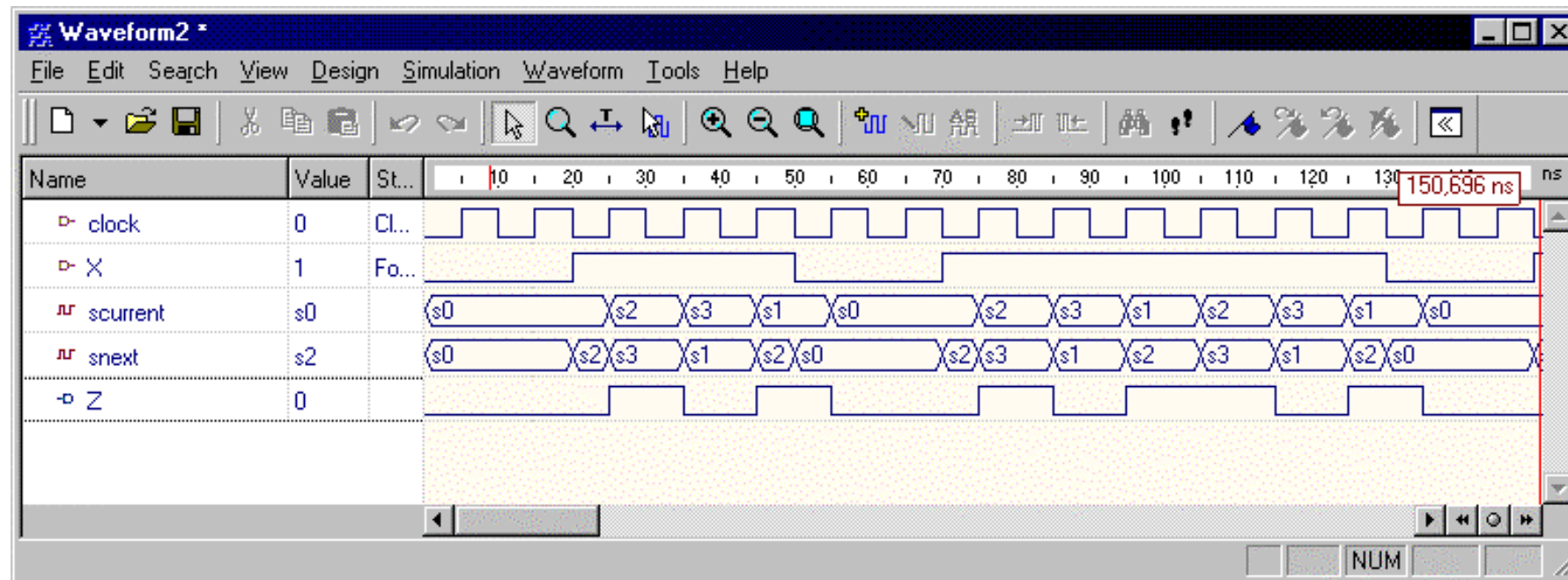
```
end A;
```

□ *Moore* → saídas são calculadas apenas à partir do ESTADO ATUAL

MÁQUINA DE ESTADOS (2/2)

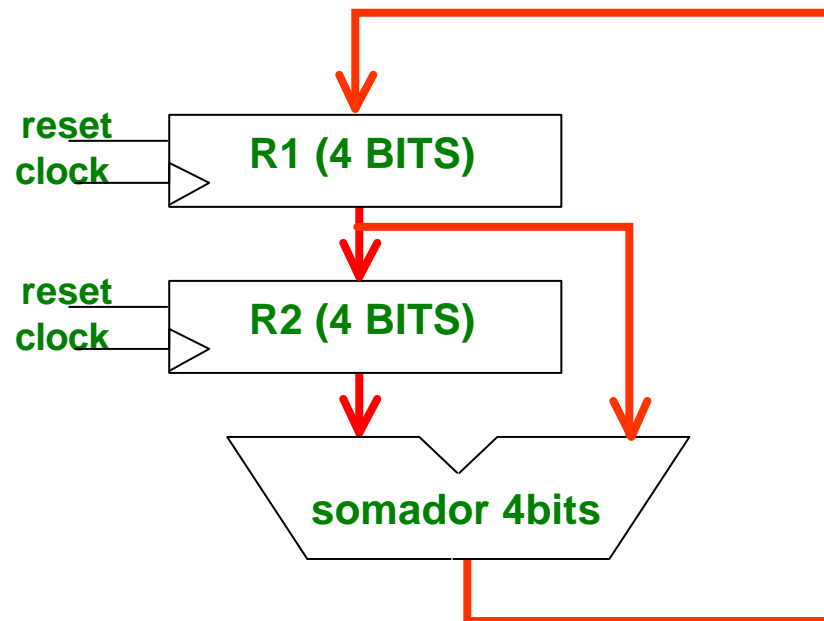
– Mealy → saídas são calculadas à partir do ESTADO ATUAL e ENTRADAS

- 1) Por que dois processos ?
- 2) Daria para implementar com apenas um processo ?
- 3) O tipo “state” está bem especificado ? Não precisa definir quem é S0,S1,S2,S3?
- 4) O que deve ser alterado no código anterior para transformar Moore em Mealy ?



EXERCÍCIO 1

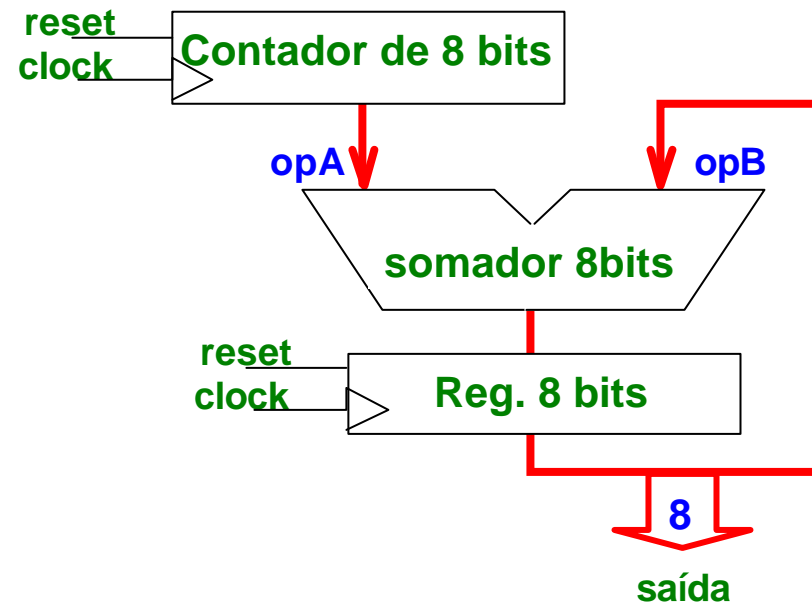
- Quando o sinal de reset for '1', os registradores R1 e R2 armazenam "0001" e "0000" respectivamente. Determinar o conteúdo de R1 e R2 para os 6 primeiros ciclos de relógio.



Descreva este circuito em VHDL.

EXERCÍCIO 2 (1/3)

- **Descreva o circuito abaixo em VHDL:**
 - Um só processo, pois as variáveis de controle são as mesmas



EXERCÍCIO 2 (descrição completa) (2/3)

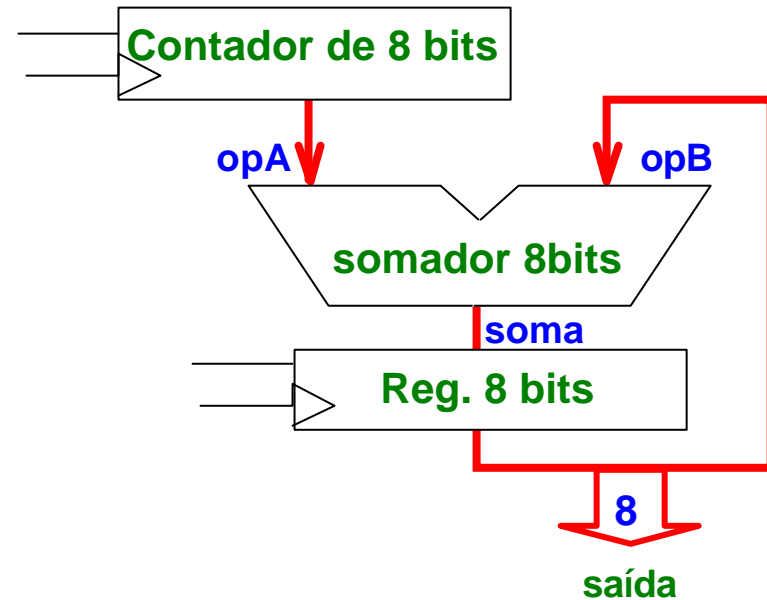
```
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_Logic_unsigned.all;

entity exemplo is
end;

architecture a1 of exemplo is
    signal opA, opB, soma : std_logic_vector(3 downto 0);
    signal clock, reset, cin, cout: std_logic;
begin

    soma <= opA + opB;

    process(reset, clock)
    begin
        if reset='1' then
            opA<=(others=>'0');
            opB<="0001";
        elsif clock'event and clock='1' then
            opA <= opB;
            opB <= soma;
        end if;
    end process;
end a1;
```

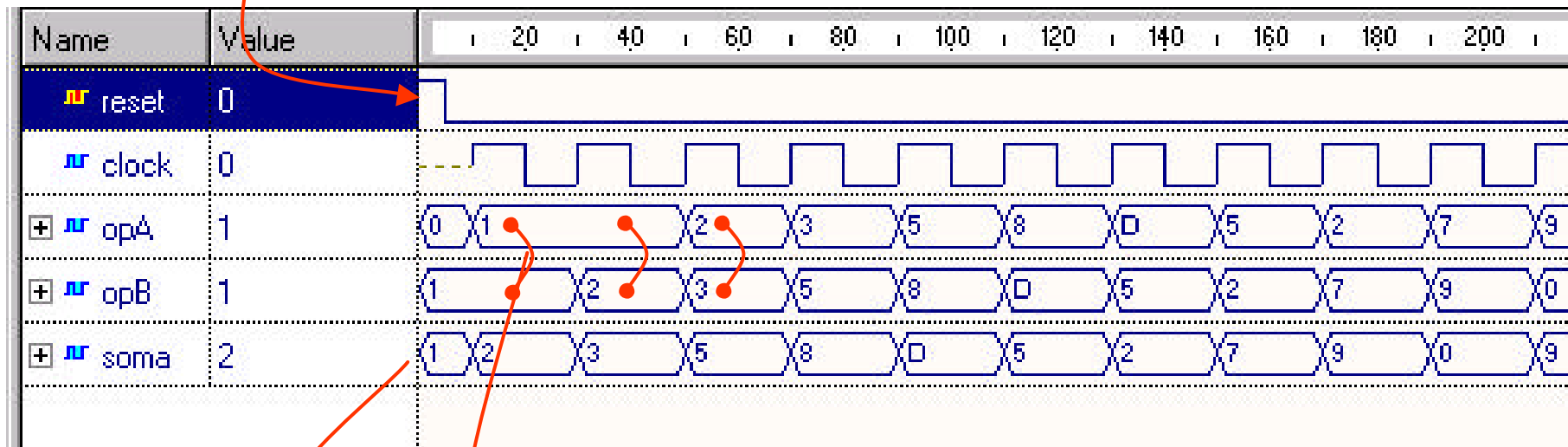


```
-- geração do clock e o reset --
reset <= '1', '0' after 5ns;
process
begin
    clock <= '1' after 10ns, '0' after 20ns;
    wait for 20ns;
end process;
```

end a1;

EXERCÍCIO 2 (simulação) (3/3)

Pulso de reset: `reset <= '1', '0' after 5ns;`



→ Soma de opA com opB resulta na soma

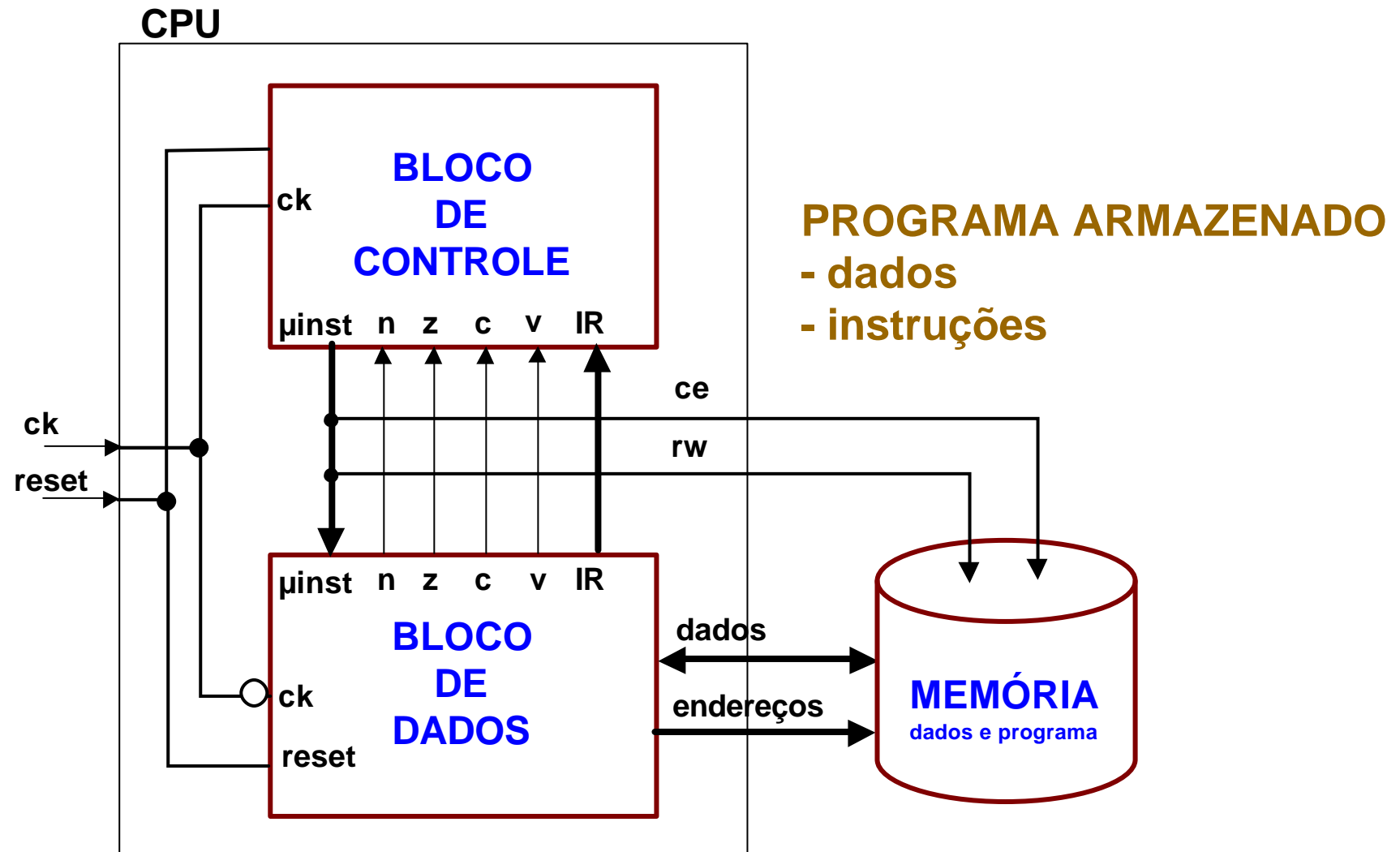
→ Saída do contador

ESTUDOS DE CASO

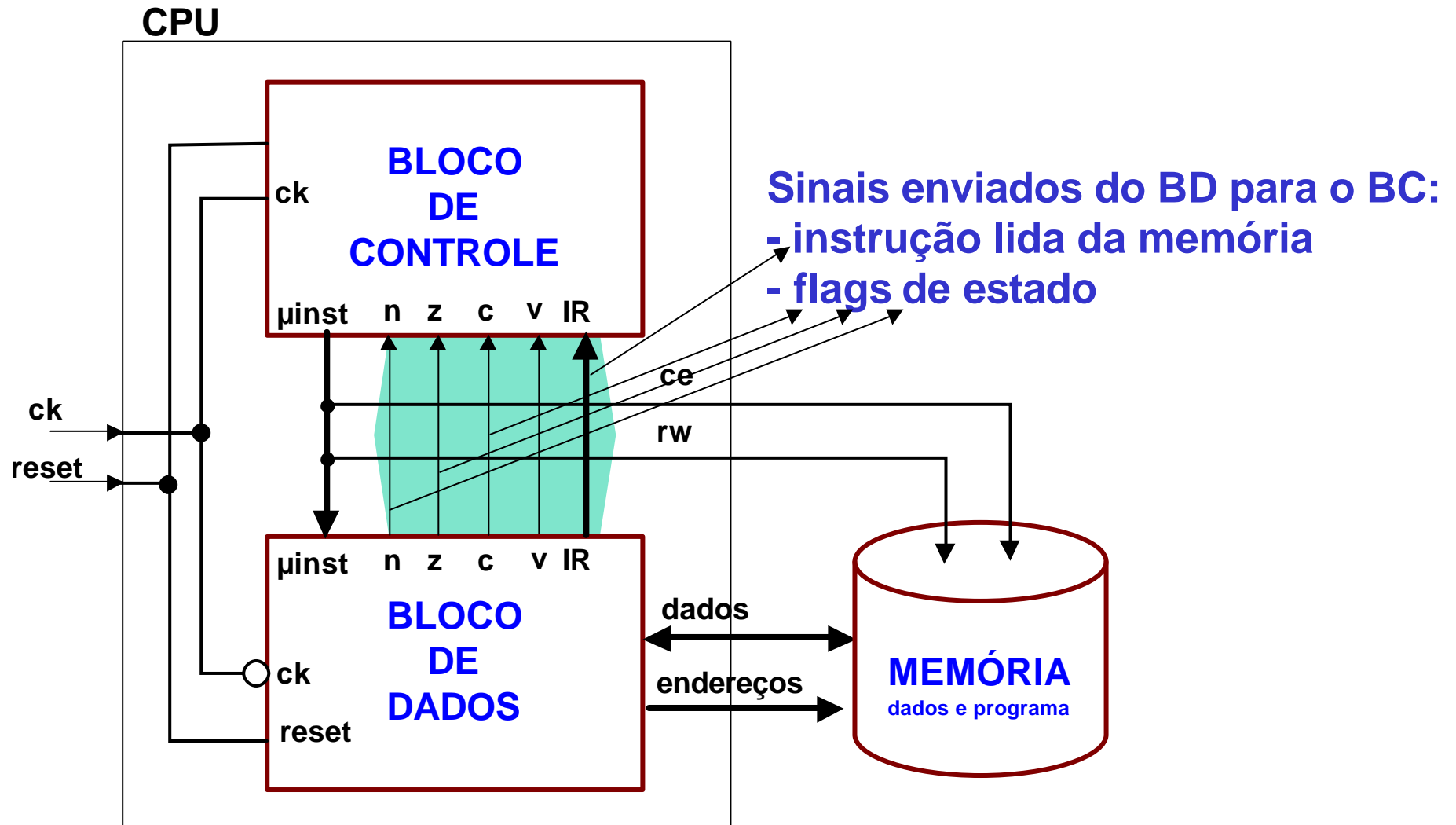
---- PARTE 3 ----

- **ARQUITETURA CLEÓPATRA**
- **COMUNICAÇÃO ASSÍNCRONA**
- **CALCULADORA**

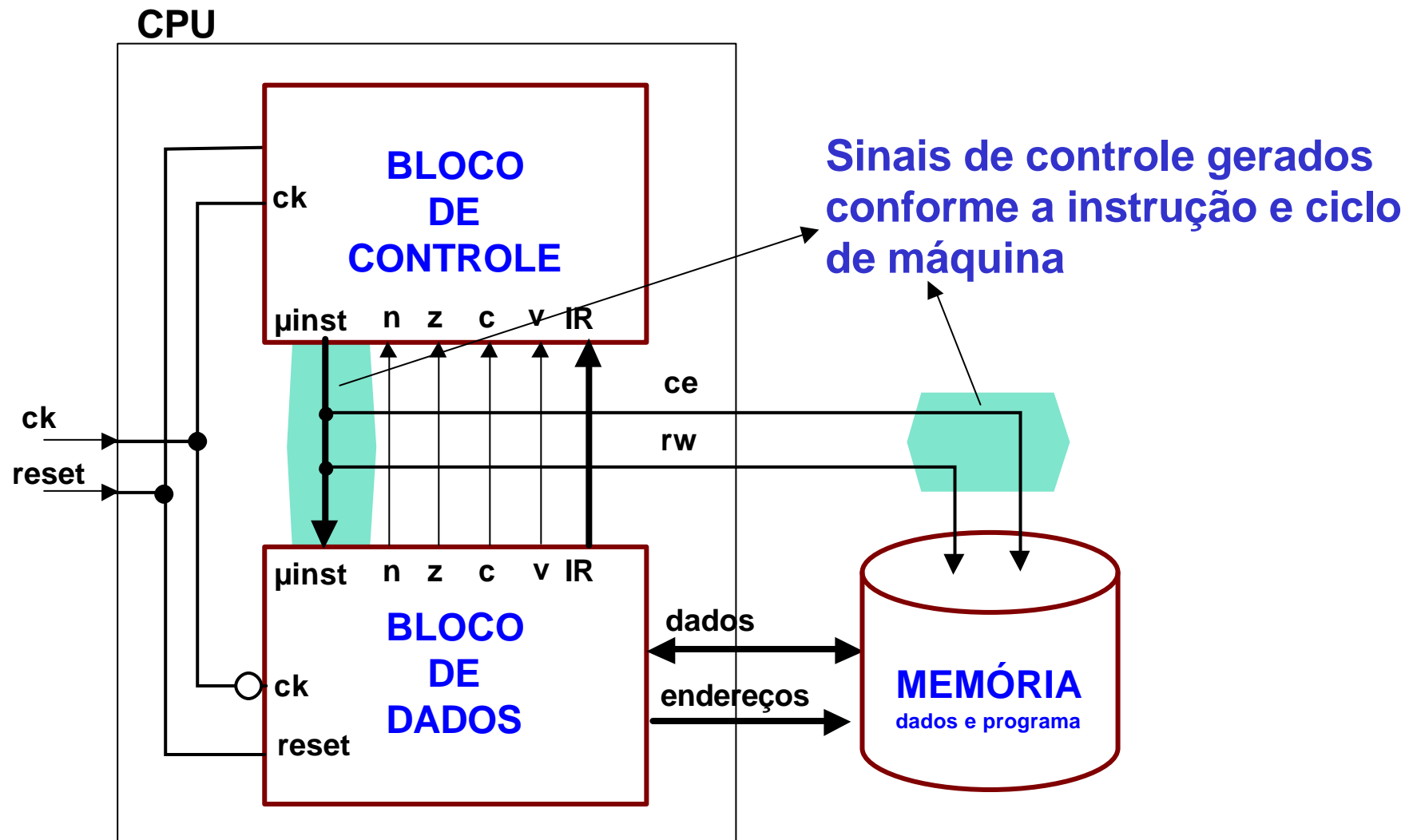
ORGANIZAÇÃO DA ARQUITETURA CLEÓPATRA



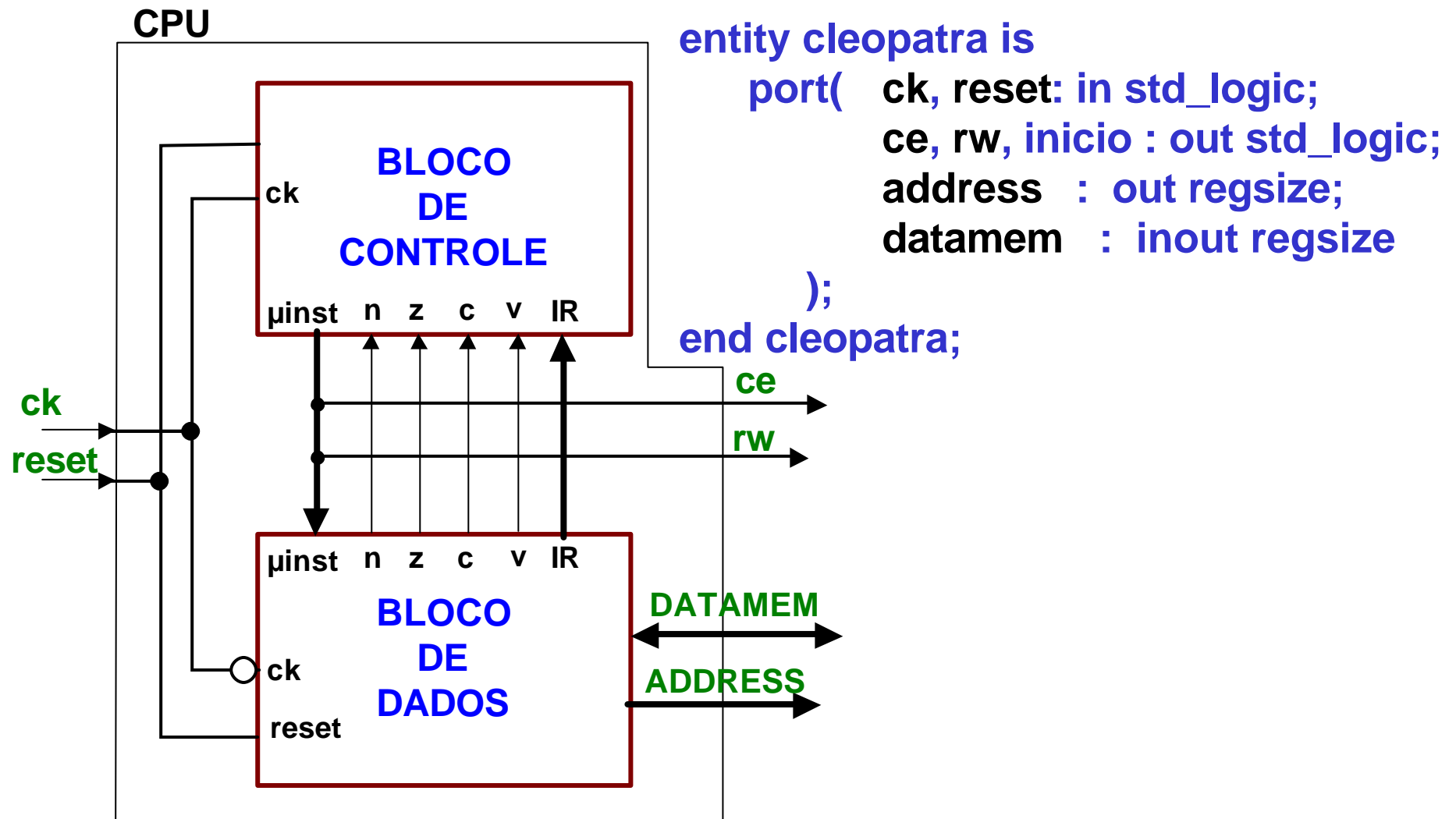
Arquitetura CLEÓPATRA



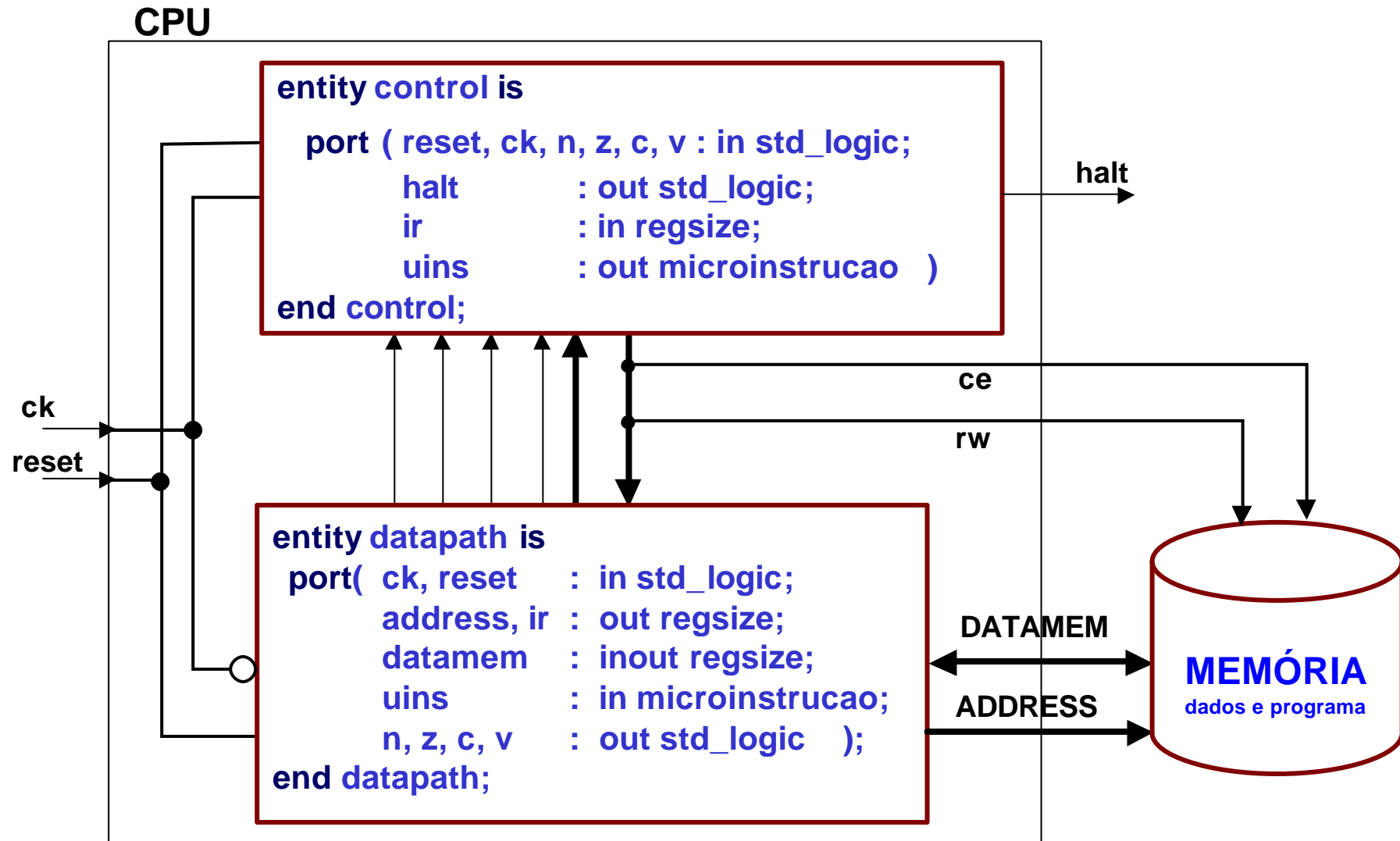
Arquitetura CLEÓPATRA



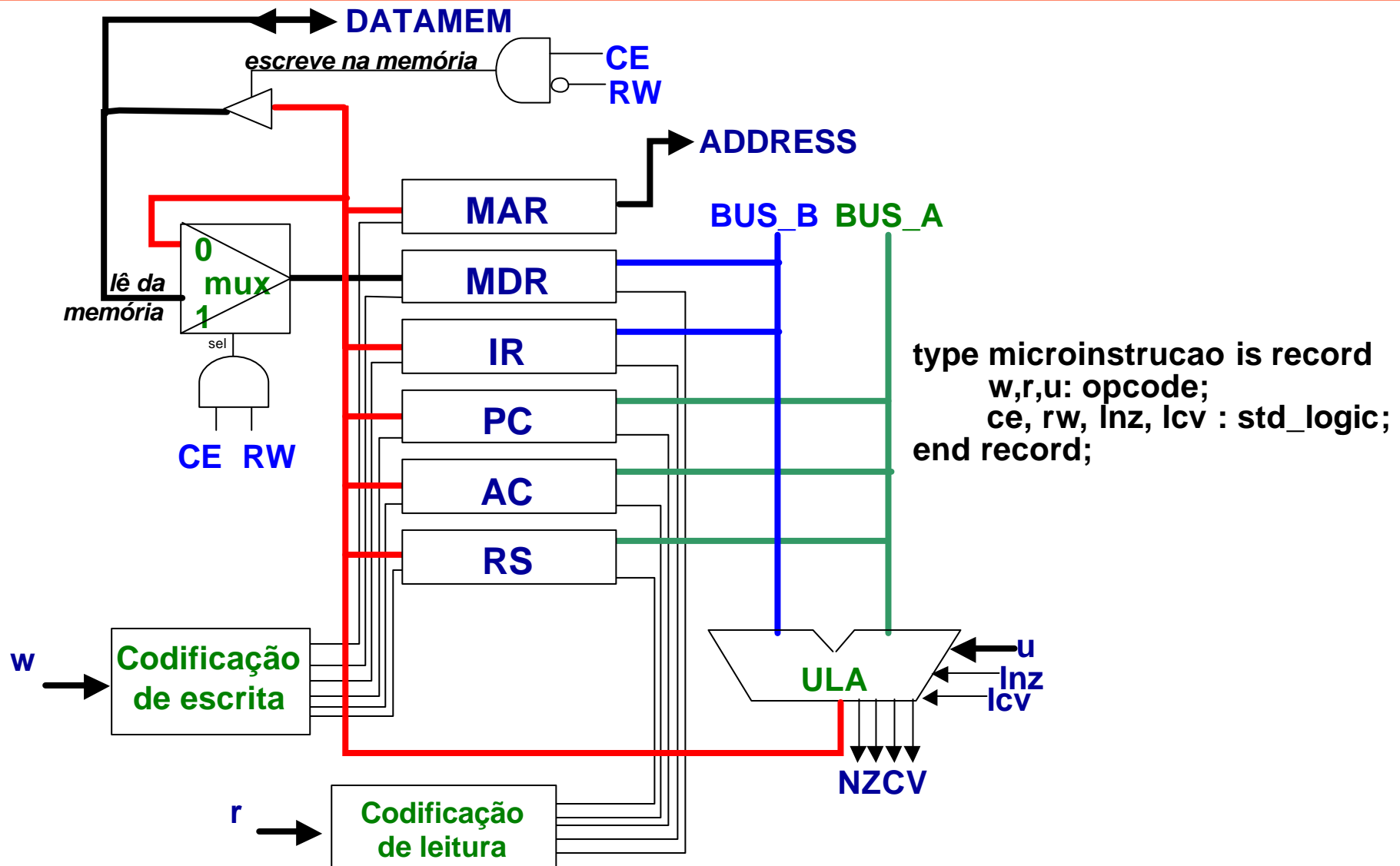
Arquitetura CLEÓPATRA



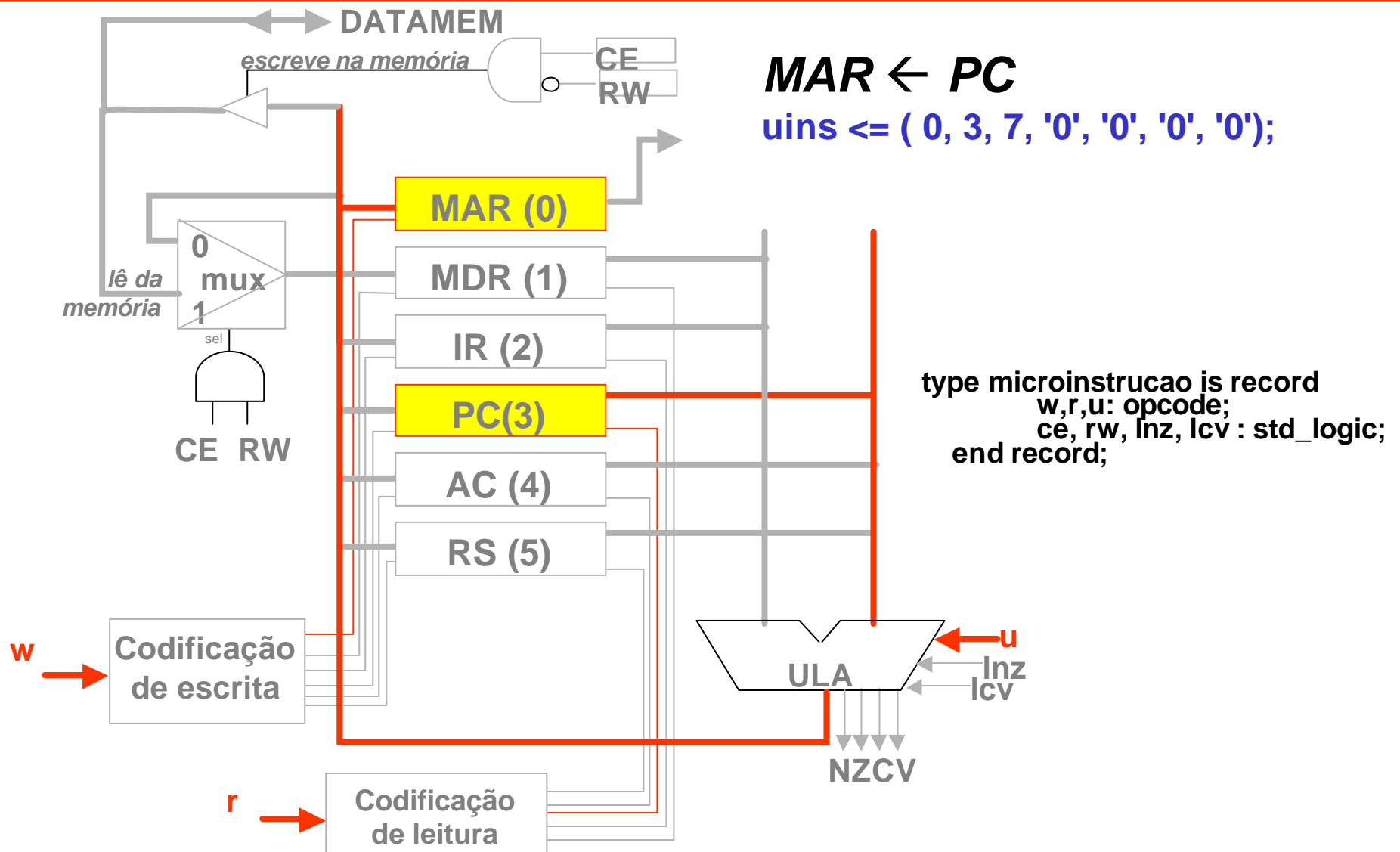
Arquitetura CLEÓPATRA



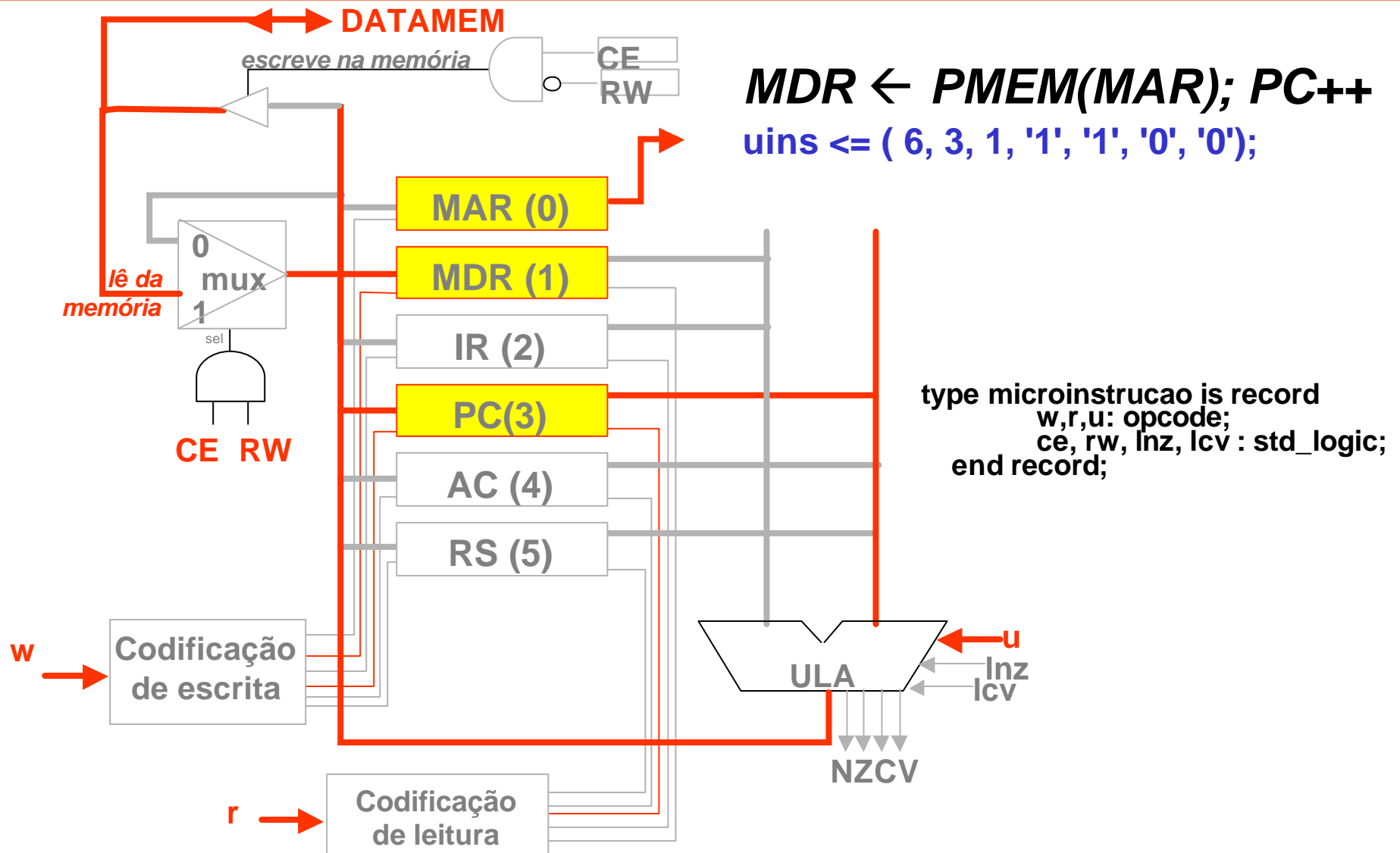
MICROINSTRUÇÃO => PALAVRA DE CONTROLE



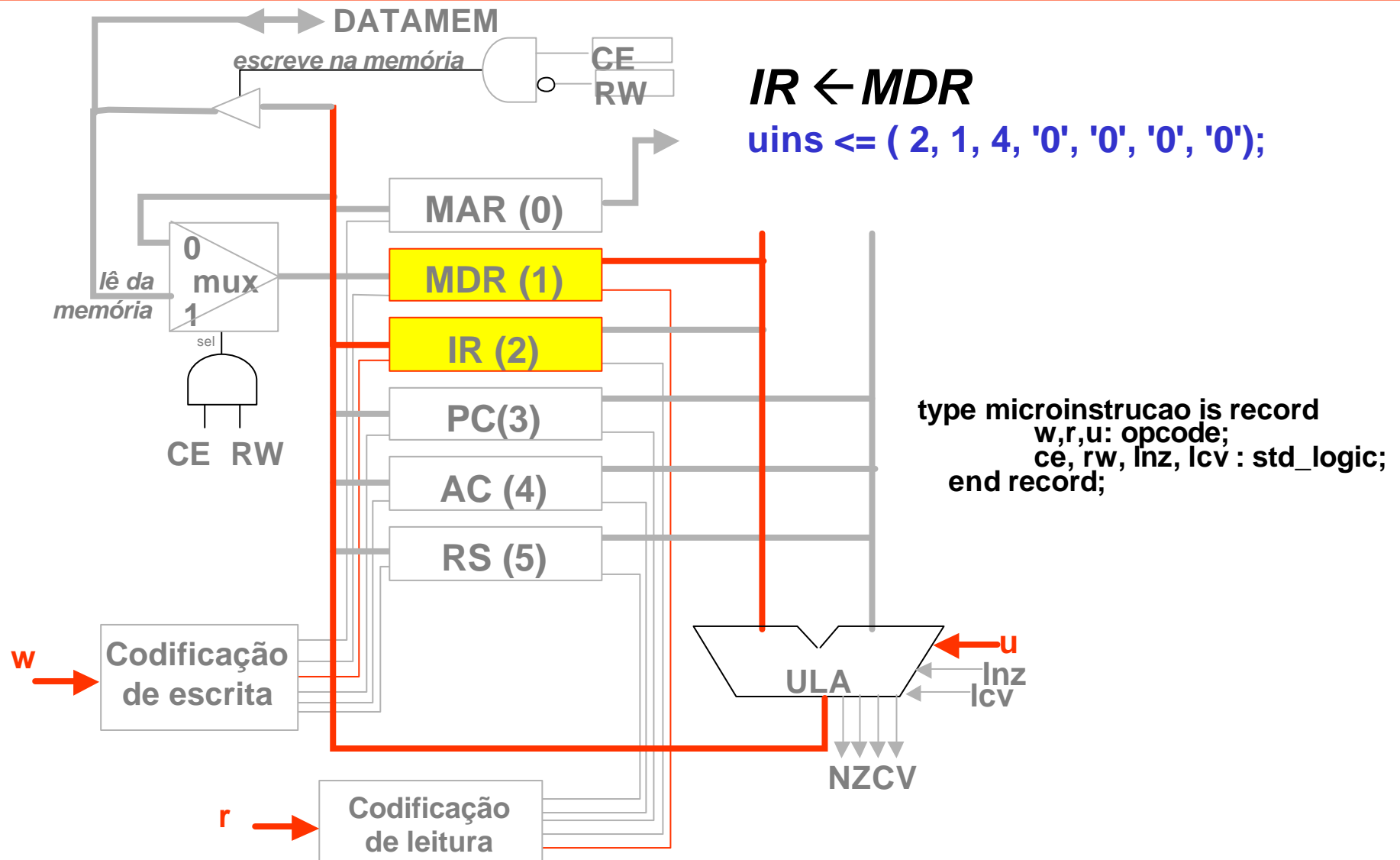
FETCH 1/3



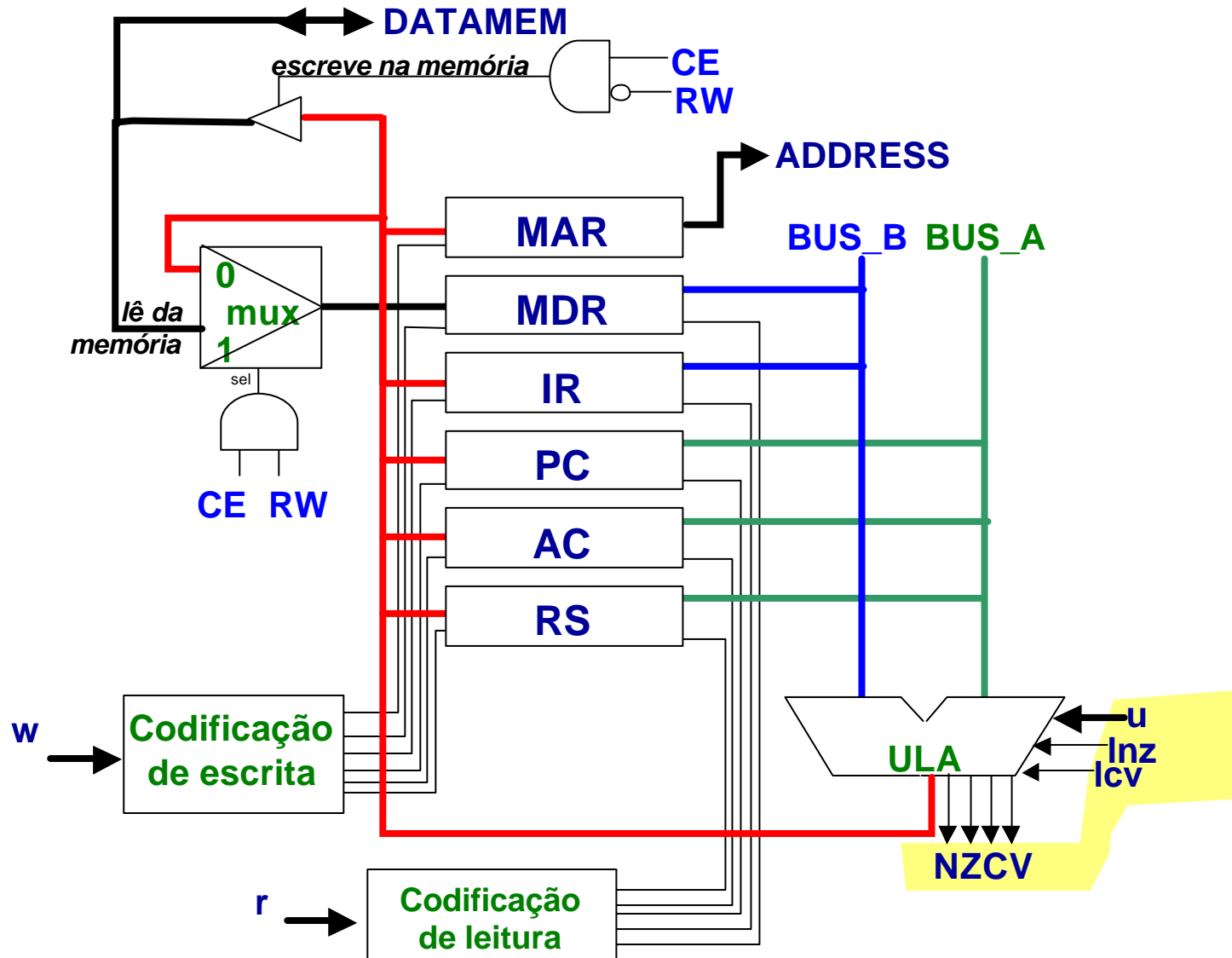
FETCH 2/3



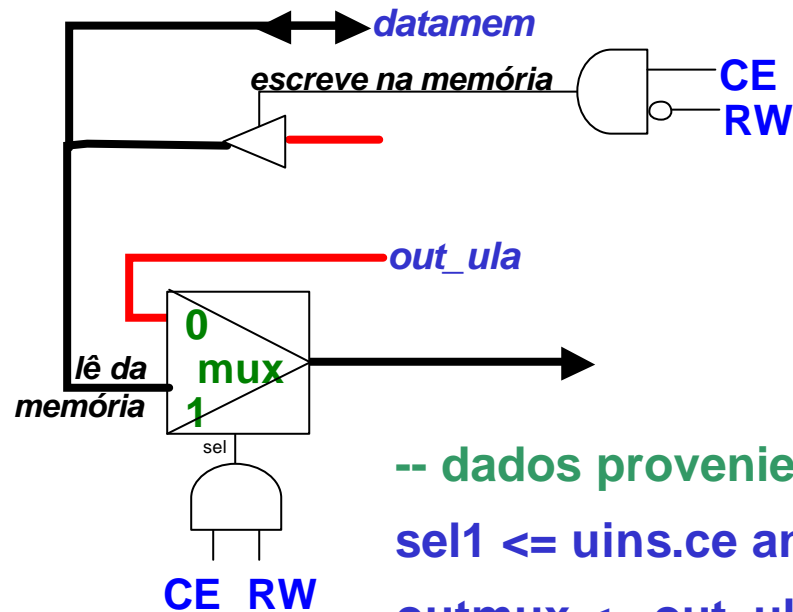
FETCH 3/3



BLOCO DE DADOS - COMPONENTES (7)



BD - ACESSO À MEMÓRIA



-- dados provenientes ou da memória ou da ULA

```
sel1 <= uins.ce and uins.rw;
```

```
outmux <= out_ula when sel1='0' else datamem;
```

-- escrita para a memória

```
sel2 <= uins.ce and (not uins.rw);
```

```
datamem <= out_ula when sel2='1' else "ZZZZZZZZ";
```

BD - REGISTRADORES

`ir <= reg_ir;` -- instrução corrente, a ser utilizada no bloco de controle

R1: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wmar**, D=>**out_ula**, Q=>**address**);

R2: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wmdr**, D=>**outmux**, Q=>**mdr**);

R3: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wir**, D=>**out_ula**, Q=>**reg_ir**);

R4: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wpc**, D=>**out_ula**, Q=>**pc**);

R5: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wac**, D=>**out_ula**, Q=>**ac**);

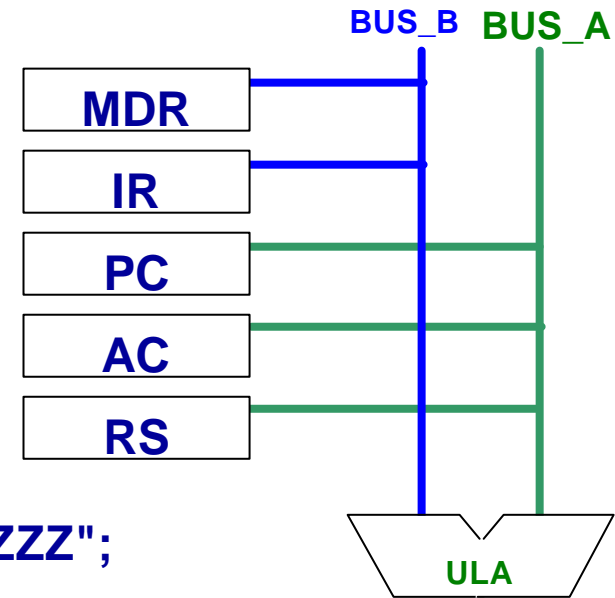
R6: **reg8clear** port map

(clock=>**ck**, reset=>**reset**, ce=>**wrs**, D=>**out_ula**, Q=>**rs**);



BD - ACESSO AOS BARRAMENTOS

Uso de tri-states:



```
busB <= mdr      when rmdr='1'      else "ZZZZZZZZ";  
busB <= reg_ir   when rir='1'       else "ZZZZZZZZ";  
busA <= pc       when rpc='1'       else "ZZZZZZZZ";  
busA <= ac       when rac='1'       else "ZZZZZZZZ";  
busA <= rs       when rrs='1'       else "ZZZZZZZZ";
```

BD - CODIFICAÇÃO DE ESCRITA

wmar <= '1'	when uins.w=0	else '0';
wmdr <= '1'	when uins.w=1 or uins.w=6	else '0';
wir <= '1'	when uins.w=2	else '0';
wpc <= '1'	when uins.w=3 or uins.w=6	else '0';
wac <= '1'	when uins.w=4	else '0';
wrs <= '1'	when uins.w=5	else '0';



BD - CODIFICAÇÃO DE LEITURA

rmdr <= '1' when uins.r =1 or uins.r=6 or uins.r=7 else '0';

rir <= '1' when uins.r =2 else '0';

rpc <= '1' when uins.r =3 or uins.r=7 else '0';

rac <= '1' when uins.r =4 or uins.r=6 else '0';

rrs <= '1' when uins.r=5 else '0';



BD - ULA

```
um <= "00000001";
zero <= '0';
process(uins.u,busA,busB)
begin
    case uins.u is
        when 0 => somaAB( busA, busB, zero, out_ula, cout);
        when 1 => somaAB( busA, um,  zero, out_ula, cout);
        when 2 => out_ula <= not busA;
        when 4 => out_ula <= busB;
        when 5 => out_ula <= busA or busB;
        when 6 => out_ula <= busA and busB;
        when 7 => out_ula <= busA;
        when others => null;
    end case;
end process;
```



BD - FLAGS DE ESTADO (falta v)

```
process(ck,reset,uins)
begin
  if (reset='1') then
    c <= '0'; n <= '0'; z <= '0';
  elsif ck'event and ck='0' then
    if uins.c='1' then c <= cout; end if;
    if uins.nz='1' then
      n <= out_ula(7);
      z <= is_zero(out_ula);
    end if;
  end if;
end process;
```

Chamada de FUNÇÃO



BLOCO DE CONTROLE

- **Função:** gerar os sinais de controle para o bloco de dados, em função da instrução corrente e dos flags de estado.
- **Estrutura básica do bloco de controle:**

```
process
```

```
begin
```

```
wait until ck'event and ck='1';
```

← **Espera o clock**

```
-- fetch --
```

```
uins <= mar_pc;      wait until ck'event and ck='1';
```

```
uins <= mdr_MmarP;  wait until ck'event and ck='1';
```

```
uins <= ir_mdr;     wait until ck'event and ck='1';
```

MICROINS E ESPERA

```
case ir is -- seleção pelo opcode
```

```
  when nota => uins <= ac_ac;
```

```
  when others => null;
```

```
end case;
```

← **SELECIONA**

```
end process;
```

VOLTA

VER NO CÓDIGO DISPONÍVEL AS CONSTANTES PARA MICROINSTRUÇÃO

BLOCO DE CONTROLE

Vantagens deste estilo de descrição:

- **Simple de descrever o controle: fetch seguido de case para seleção da operação.**
- **Fácil de realizar a temporização: basta inserir após cada microinstrução uma espera por borda de clock.**
- **Atenção: após a última microinstrução do ciclo de instrução não vai wait. Razão: antes do fetch já tem inserido um wait.**
- **Esta temporização permite instruções com número diferente de ciclos para execução, como é o caso da arquitetura proposta.**

BC - Exemplo de instrução (1)

- De acordo com a especificação LDA, ADD, OR, AND são praticamente iguais

when `ldaim | andim | orim | addim =>`

`uins <= mar_pc; wait until ck'event and ck='1';`

`uins <= mar_MmarP; wait until ck'event and ck='1';`

`sel_op (ir(7 downto 4), uins);`

t0: MAR ← PC
t1: MDR ← PMEM(MAR); PC++
t2: AC ← AC op MDR
setar flags

Função para escolha do microcomando em função dos 4 bits mais significativos

(1) continuação

- *Função para escolha do microcomando para LDA/ADD/OR/AND*
- *Inserir a função ou no package ou antes do begin*

```
procedure sel_op (signal ir: in std_logic_vector(3 downto 0);  
                 signal uins : out microinstrucao ) is
```

```
begin
```

```
  case ir is
```

```
    when x"4" => uins <= (4, 1, 4, '0','0', '1','0'); -- ac <- mdr
```

```
    when x"5" => uins <= (4, 6, 0, '0','0', '1','1'); -- ac <- ac + mdr
```

```
    when x"6" => uins <= (4, 6, 5, '0','0', '1','0'); -- ac <- ac or mdr
```

```
    when x"7" => uins <= (4, 6, 6, '0','0', '1','0'); -- ac <- ac and mdr
```

```
    when others => null;
```

```
  end case;
```

```
end sel_op;
```



flags

BC - Exemplo de instrução (2)

- *Micro código para os jumps (endereçamento direto)*
- *Trata-se todos os jumps juntos, no mesmo caso*

when jcdir | jndir | jzdir =>

uins <= mar_pc; **wait until** ck'event **and** ck='1';

uins <= mdr_MmarP; **wait until** ck'event **and** ck='1';

if (((jc **and** c)='1') **or** ((jn **and** n)='1') **or** ((jz **and** z)='1')) **then**

 uins <= pc_mdr;

else uins <= nop;

end if;

```
t0: MAR ← PC
t1: MDR ← PMEM(MAR);
t2: if(flag) then PC ← MDR
     else PC++;
```

BC - Exemplo de instrução (3)

- *Micro código para o HALT:*
 - *implementa através de uma espera pelo reset*

when hlt =>

```
while reset='0' loop
    wait until ck'event and ck='1';
end loop;
```

Crítica à implementação apresentada:

As seqüências **mar_pc**, **mdr_MmarP**, e **mdr_Mmar** são repetidas inúmeras vezes. Poder-se-ia ter escrito um código mais estruturado.

ENTIDADE CPU

entity cleopatra is

port(ck, reset: **in** std_logic;
address: **out** regsize;

end cleopatra;

ce, rw, inicio : **out** std_logic;
datamem: **inout** regsize);

architecture cleopatra of cleopatra is

component datapath is

port(uins : **in** microinstrucao;
ir, address : **out** regsize;
n, z, c, v : out std_logic);

end component datapath;

ck, reset: **in** std_logic;
datamem : **inout** regsize;

component control is

port(ir : **in** regsize;
uins : **out** microinstrucao;
);

end component control;

n, z, c, v, ck, reset: **in** std_logic

signal uins : microinstrucao;
signal ir : regsize;

signal n,z,c,v : std_logic;

begin

ENTIDADE CPU

begin

```
ce <= uins.ce;  
rw <= uins.rw;
```

SINAIS PARA A MEMÓRIA



```
dp: datapath port map ( uins=>uins, ck=>ck, reset=>reset, ir=>ir,  
address=>address, datamem=>datamem, n=>n, z=>z, c=>c, v=>v);
```

```
ctrl: control port map ( ir=>ir, n=>n, z=>z, c=>c, v=>v, ck=>ck,  
reset=>reset, uins=>uins);
```

end cleopatra;

TEST BENCH (1)

- *Módulo responsável por gerar os vetores de teste para a simulação*
- **AÇÕES:**
 - 1 -- *incluir a CPU no test_bench*
 - 2 -- *gerar o clock*
 - 3 -- *gerar o reset*
 - 4 -- *ler da memória*
 - 5 -- *escrever na memória, de maneira síncrona, como nos registradores*
 - 6 -- *realizar a carga na memória quando acontece o reset*

TEST BENCH (2)

- **IMPLEMENTAÇÃO:**

architecture tb of tb is

signal ck, reset, ce, rw, inicio: std_logic;

signal address, data : regsize;

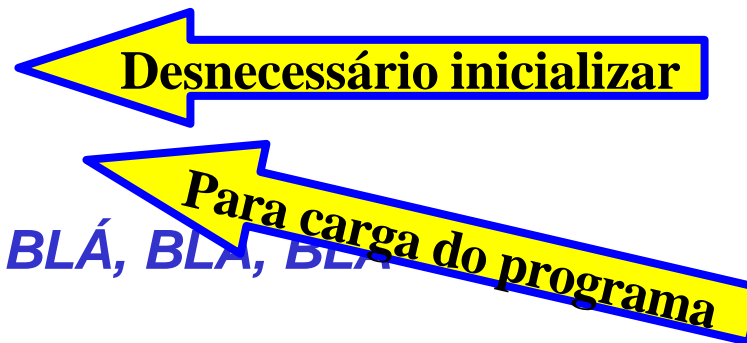
file INFILE : TEXT open READ_MODE is "program.txt";

signal memoria : ram;

signal ops, endereco : integer;

begin

end tb



TEST BENCH (3)

1 -- incluir a CPU no test_bench

```
cpu : cleopatra port map(ck=>ck, reset=>reset, ce=>ce, rw=>rw,  
                        address=>address, datamem=>data);
```

2 -- gerar o clock

```
process  
begin  
    ck <= '1', '0' after 10ns;  
    wait for 20ns;  
end process;
```

3 -- gerar o reset

```
reset <= '1', '0' after 5ns ;
```

TEST BENCH (4)

A MEMÓRIA É UM ARRAY, QUE É LIDO OU ESCRITO CONFORME OS SINAIS CE E RW.

4 -- ler da memória

```
data <= memoria(CONV_INTEGER(address)) when ce='1' and rw='1'  
    else "ZZZZZZZZ";
```



TEST BENCH (4 bis)

5 -- escrever na memória, de maneira síncrona, como nos registradores

- **PROBLEMA para escrita - duas fontes de escrita: inicialização e Cleóptara.**
- **Solução:**

```
process(go, ce, rw, ck)
```

```
begin
```

```
  if go'event and go='1' then
```

```
    if endereco>=0 and endereco <= 255 then
```

```
      memoria(endereco) <= conv_std_logic_vector(ops,8);
```

```
    end if;
```

```
  elsif ck'event and ck='0' and ce='1' and rw='0' then
```

```
    if CONV_INTEGER(address)>=0 and CONV_INTEGER(address) <= 255 then
```

```
      memoria(CONV_INTEGER(address)) <= data;
```

```
    end if
```

```
  end if;
```

```
end process;
```

escrita pelo test_bech

*escrita pela
Cleópatra*

Importante: testar os limites da RAM



TEST BENCH (5)

O PROGRAMA ARMAZENADO NA MEMÓRIA É CARREGADO QUANDO O RESET ESTÁ ATIVO

6 -- realizar a carga na memória quando acontece o reset

process

variable IN_LINE : LINE; -- pointer to string

variable linha : string(1 to 5);

begin

wait until reset = '1';

SUBIDA DO RESET

while NOT(endfile(INFILE)) loop -- end file checking

readline(INFILE,IN_LINE); -- read line of a file

read(IN_LINE, linha);

decodifica a linha e gera o sinal "go"

end loop;

end process;

LAÇO DE LEITURA

TEST BENCH (6)

- **COMO CONVERTER A LINHA EM ENDEREÇO E DADO E GERAR “GO” :**

```
case linha(1) is
  when '0' => endereco <= 0;
  when '1' => endereco <= 1;

  when 'F' => endereco <=15;
  when others => null;
end case;
wait for 1 ps;

case linha(2) is
  when '0' => endereco <= endereco*16 + 0;
  when '1' => endereco <= endereco*16 + 1;

  when 'F' => endereco <= endereco*16 + 15;
  when others => null;
end case;

-- linha (3) é espaço em branco
```

```
case linha(4) is
  when '0' => ops <= 0;
  when '1' => ops <= 1;

  when 'F' => ops <=15;
  when others => null;
end case;
wait for 1 ps;

case linha(5) is
  when '0' => ops <= ops*16 + 0;
  when '1' => ops <= ops*16 + 1;

  when 'F' => ops <= ops*16 + 15;
  when others => null;
end case;
wait for 1 ps;
go <= '1';
wait for 1 ps;
go <= '0';

```

*Pulso em “go”
gera escrita na
memória*

Fazer uma função para converter um char em inteiro

SIMULAÇÃO (1) - PROGRAMA

- **PROGRAMA (em código objeto)**

00 40 ; endereço 00 LDA #

01 98 ; endereço 01 H98

02 54 ; endereço 02 ADD

03 10 ; endereço 03 H10

04 24 ; endereço 04 STA

05 11 ; endereço 05 H11

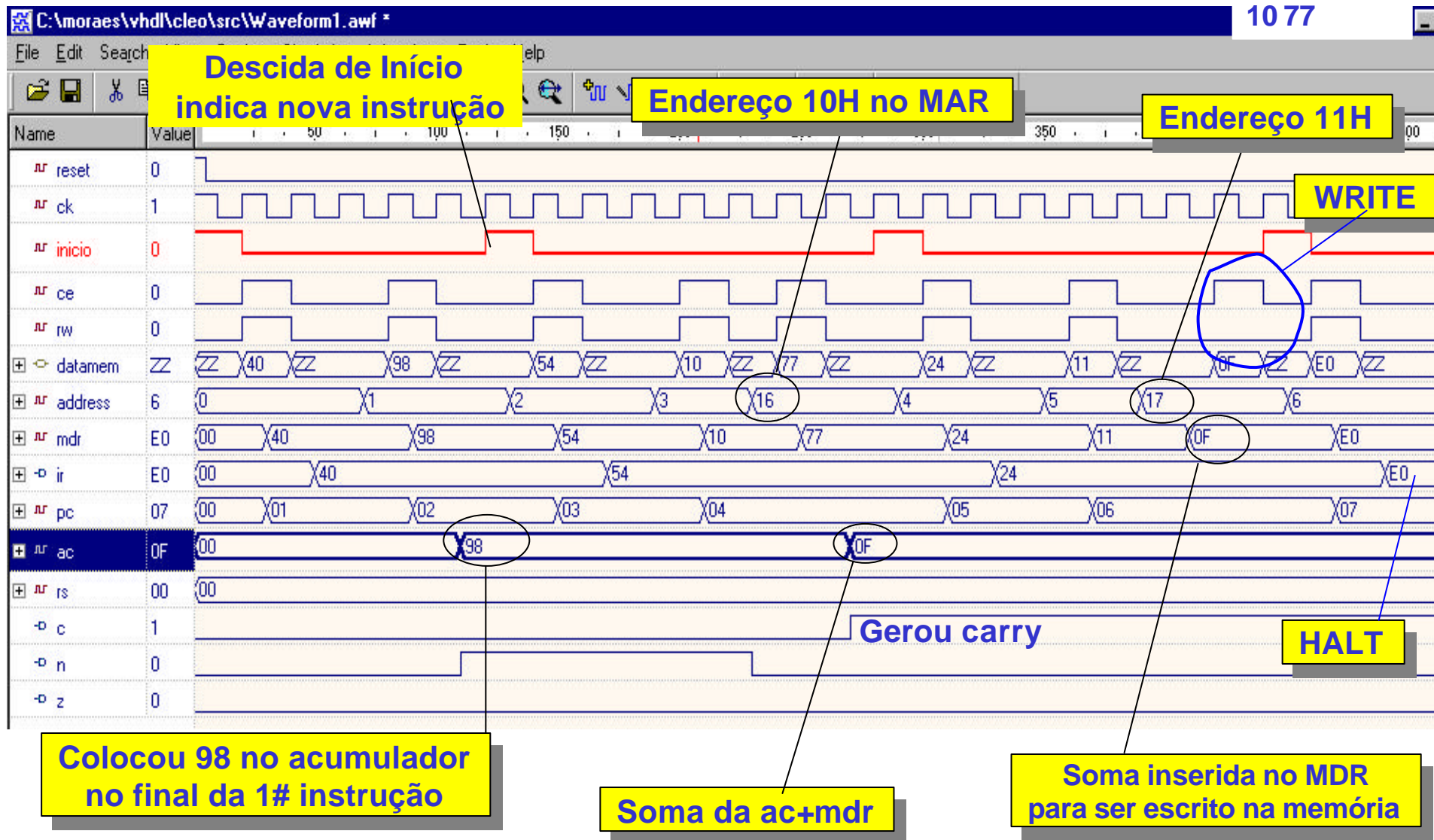
06 E0 ; endereço 06 HALT

10 77 ; endereço 10 H77

- **FUNÇÃO DO PROGRAMA:** somar a constante H98 ao conteúdo do endereço H10 e depois gravar o resultado em H11

SIMULAÇÃO (2)

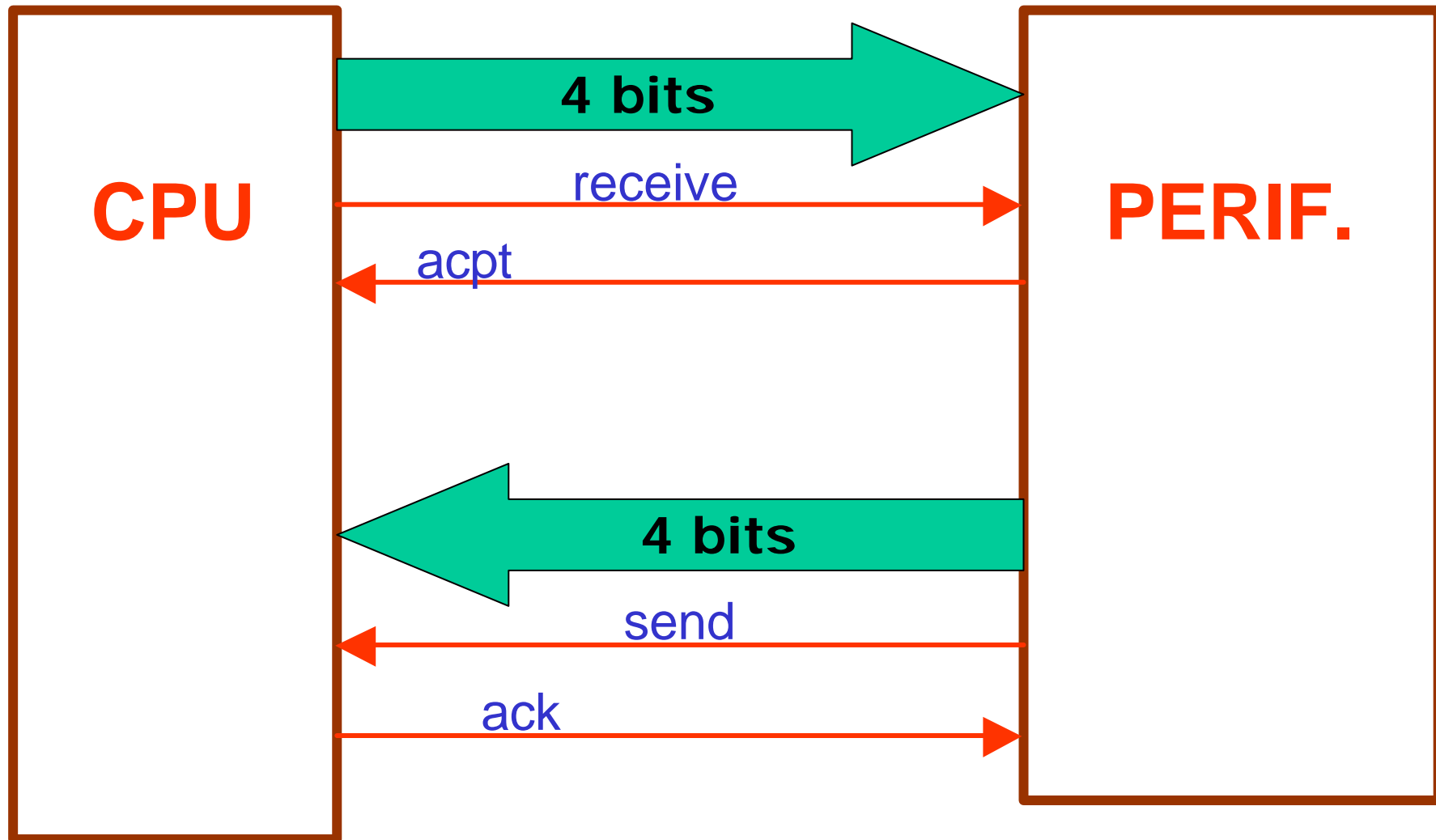
LDA # H98
ADD H10
STA H11
HALT
10 77



ESTUDOS DE CASO

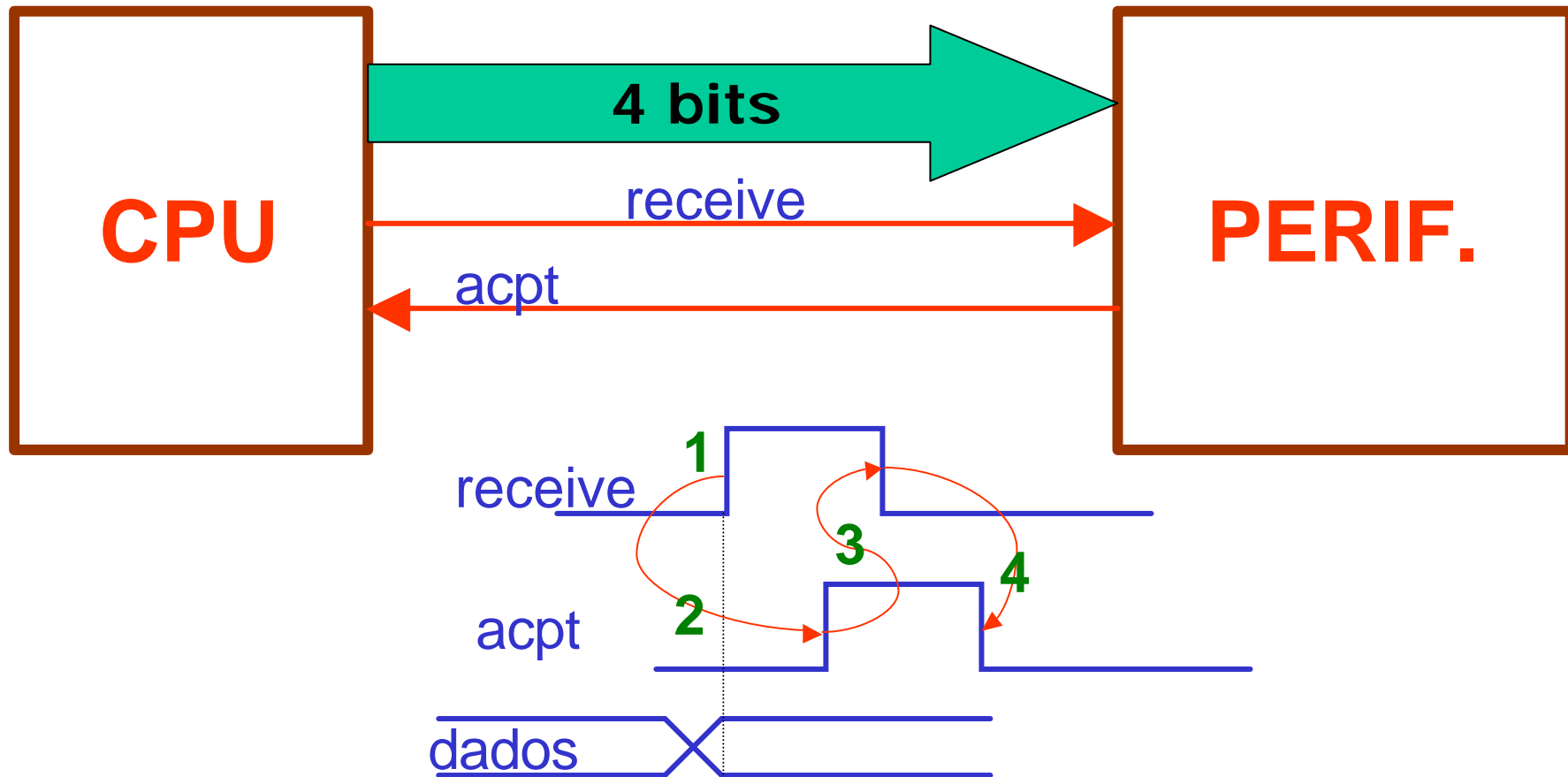
- **ARQUITETURA CLEÓPATRA**
- **COMUNICAÇÃO ASSÍNCRONA**
- **CALCULADORA**

COMUNICAÇÃO ASSÍNCRONA



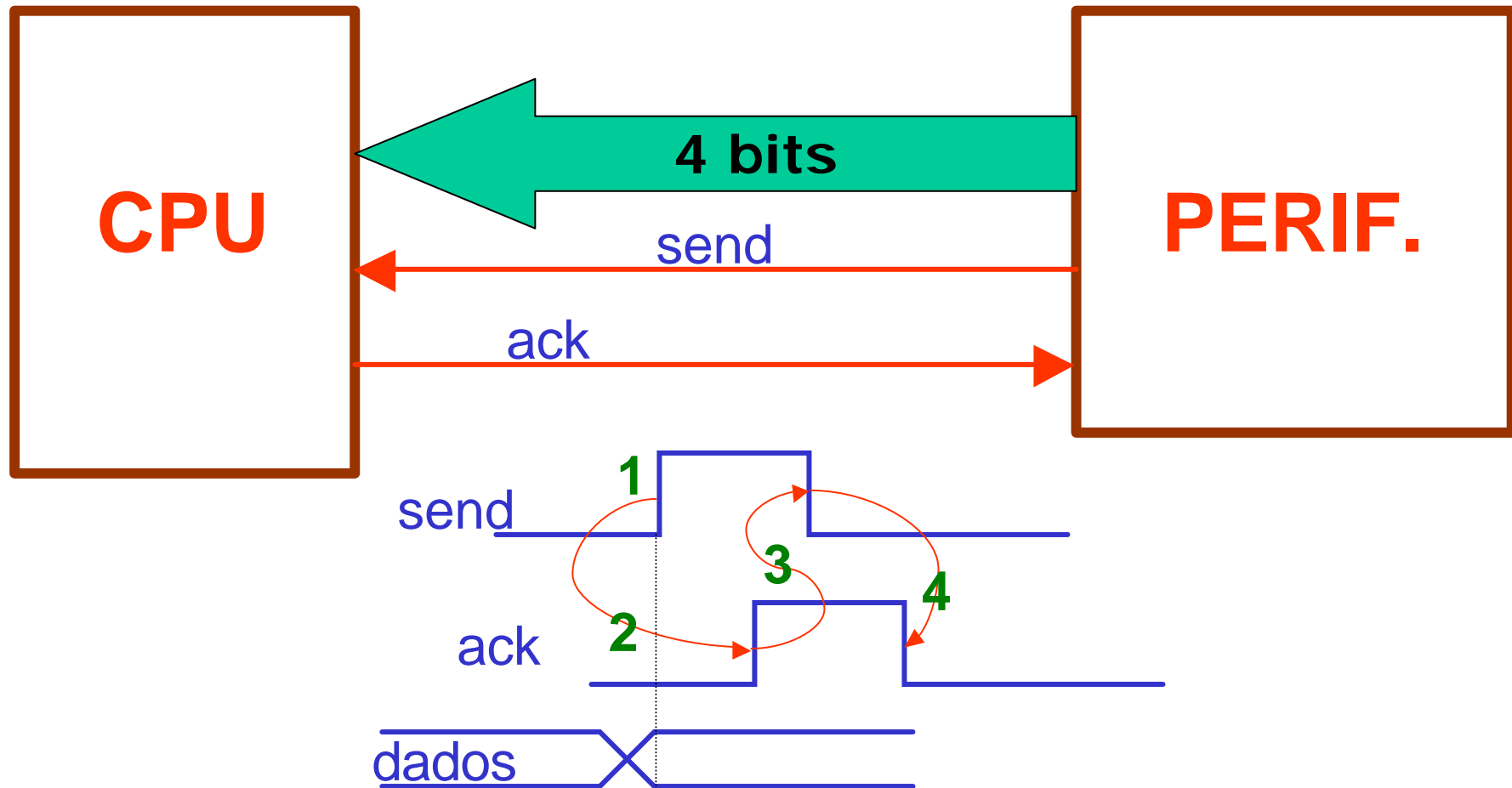
COMUNICAÇÃO ASSÍNCRONA

- Envio de dados da CPU para o periférico



COMUNICAÇÃO ASSÍNCRONA

- Envio de dados do periférico para a CPU



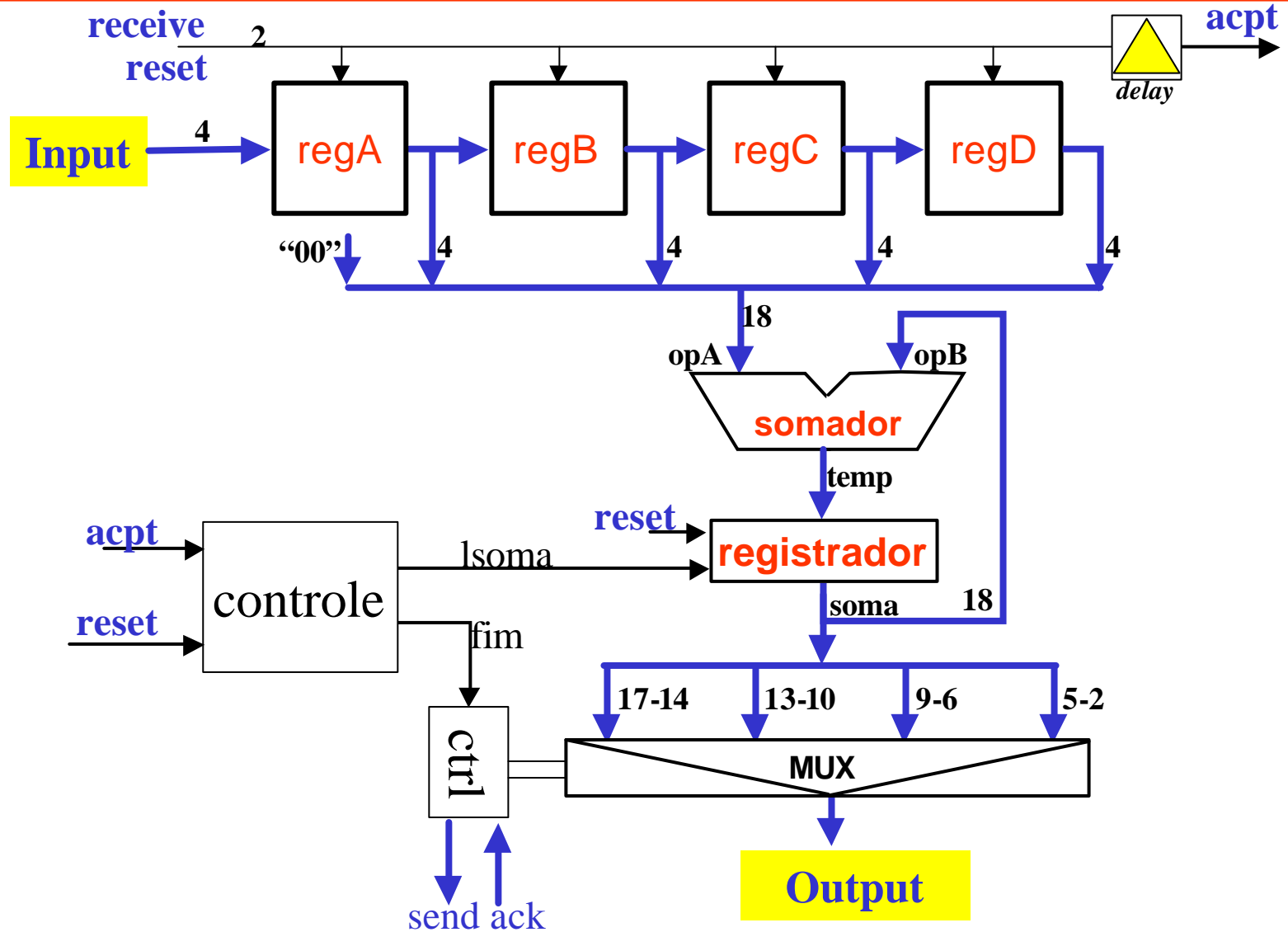
Função do periférico

- *Receber 4 palavras de 16 bits*
 - para isto a CPU deve enviar 16 palavras de 4 bits
- *Somar as 4 palavras de 16 bits, sem perder precisão*
 - para isto o somador deve ter 18 bits
- *Calcular a média aritmética das 4 palavras, sem utilizar divisão*
 - emprego de deslocamento à direita
- *Enviar para a CPU a média (16 bits) em pacotes de 4 bits*

Implementação do periférico

- *5 módulos:*
 - **registrador de deslocamento de entrada**
 - conversão serial para paralelo
 - **somador de 18 bits**
 - **registrador para armazenar a soma**
 - **multiplexador de saída**
 - conversão paralelo para serial
 - **controle**

Implementação



MÓDULO 1

- MODULO 1 : recepção dos dados por palavras de 4 bits
- reset é assíncrono, quando sobe o receive os registradores são
- deslocados e o acpt sobe. Depois de receive descer o acpt e' retirado

```
acpt <= aceite;  
entrada : process
```

```
begin
```

```
wait on reset, receive;
```

```
if reset = '1' then
```

```
regA <= zero; regB <= zero; regC <= zero; regD <= zero;
```

```
elsif receive'event and receive='1' then
```

```
regA <= input;
```

```
regB <= regA;
```

```
regC <= regB;
```

```
regD <= regC;
```

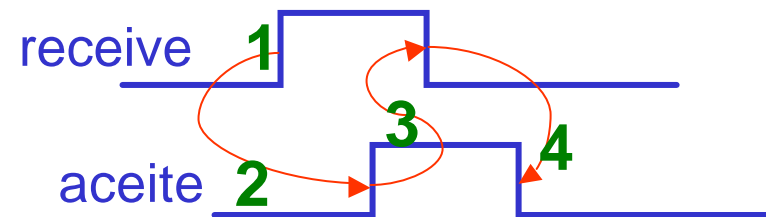
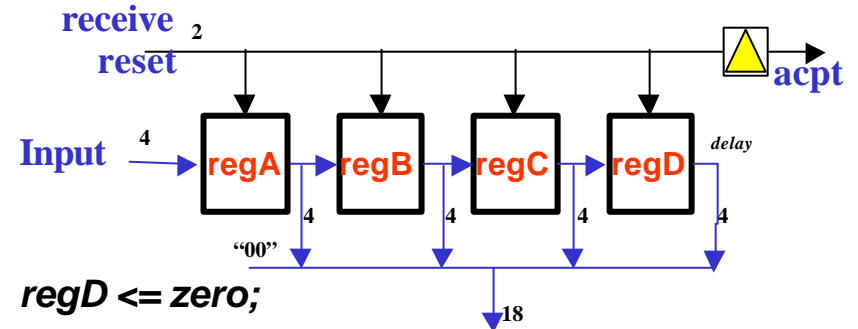
```
aceite <= '1' after 10 ns;
```

```
wait until receive'event and receive='0';
```

```
aceite <= '0' after 10 ns;
```

```
end if;
```

```
end process;
```



MÓDULO 2

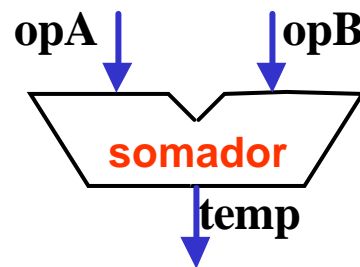
-- MODULO 2 : soma duas palavras de 18 bits

***opA* <= "00" & *regD* & *regC* & *regB* & *regA*;**

***opB* <= *soma*;**

***cin* <= '0';**

***somaAB*(*opA*, *opB*, *cin*, *temp*, *cout*);**



MÓDULO 3

-- MODULO 3 : armazena o resultado da soma, quando vem o sinal lsoma

store_soma: process (lsoma, reset)

begin

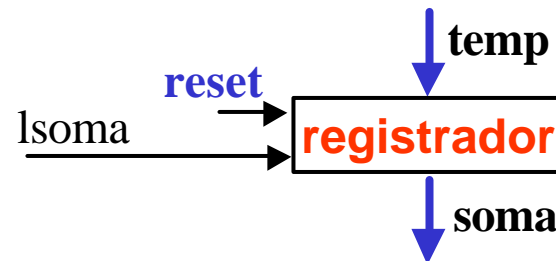
if reset='1' then soma <= "000000000000000000";

elsif (lsoma'event and lsoma='1')

then soma <= temp;

end if;

end process;



MÓDULO 4

-- MODULO 4 : realiza o envio dos dados, multiplexando a soma

saida : process

begin

wait on fim;

if fim'event and fim='1' then

-- envia: 17..14, 13..10,

for i in 4 downto 1 loop

-- 9..6 e 5..2

*output <= soma(i*4+1 downto i*4-2); -- ou seja*

send <= '1' after 10 ns;

-- deslocou 2 a esquerda

wait until ack'event and ack='1';

-- dividindo por 4

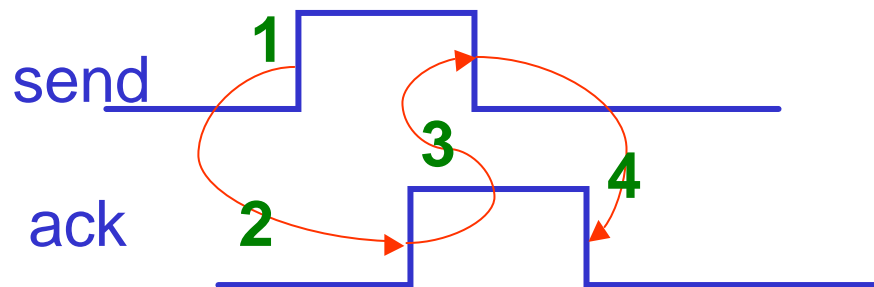
send <= '0' after 10 ns;

wait until ack'event and ack='0';

end loop;

end if;

end process;



MÓDULO 5

-- MODULO 5 : realiza o controle do numero de palavras

controle : process

variable cont, total : integer := 0;

begin

wait on aceite, reset;

if reset='1' then

Isoma<='0';

fim <= '0';

elsif aceite'event and aceite='0' then

if cont=3 then Isoma<='1'; cont:= 0;

else Isoma<='0'; cont := cont+1;

end if;

if total=15 then fim<='1' after 10ns; total:= 0;

else fim <='0' after 10ns; total := total+1;

end if;

end if;

end process;



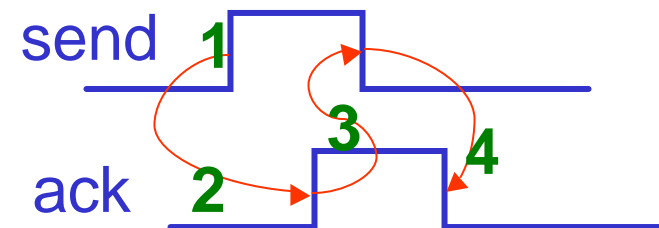
IMPLEMENTAÇÃO DO MÓDULO 4

```
wait on fim;  
if fim'event and fim='1' then  
  for i in 4 downto 1 loop  
    output <= soma(i*4+1 downto i*4-2);  
    send <= '1' after 10 ns;  
    wait until ack'event and ack='1';  
    send <= '0' after 10 ns;  
    wait until ack'event and ack='0';  
  end loop;  
end if;
```

- *Quantos estados diferentes tem este loop?*

– **Resposta: 8**

- *Implementação:*
– **Máquina de Estados**



IMPLEMENTAÇÃO DO MÓDULO 4

wait on fim;

if fim'event and fim='1' then

for i in 4 downto 1 loop

*output <= soma(i*4+1 downto i*4-2);*

send <= '1' after 10 ns;

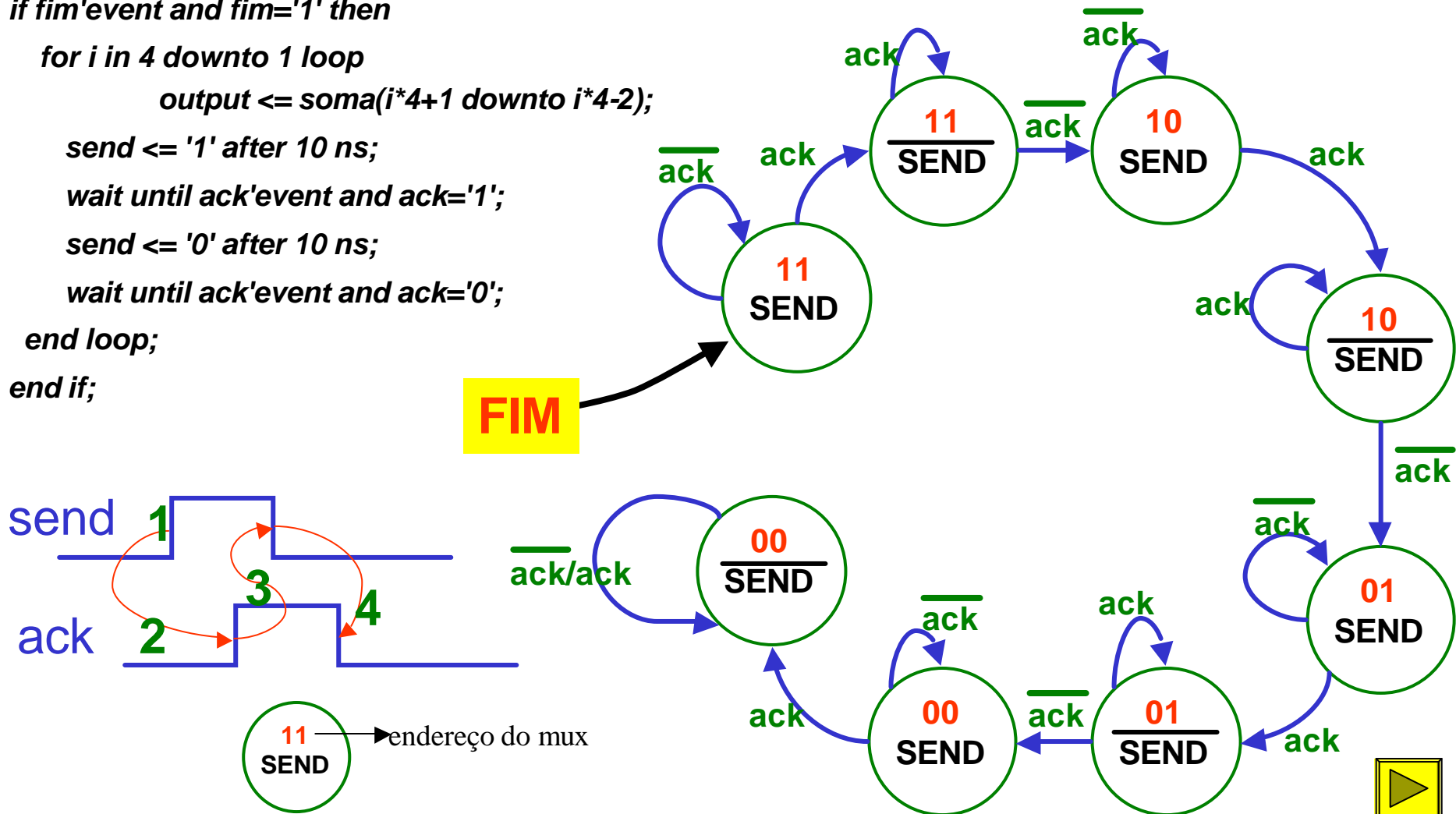
wait until ack'event and ack='1';

send <= '0' after 10 ns;

wait until ack'event and ack='0';

end loop;

end if;



IMPLEMENTAÇÃO DO MÓDULO 4

Exercício:

***IMPLEMENTE A MÁQUINA DE ESTADOS DA
TRANSPARÊNCIA ANTERIOR EM VHDL***

MÓDULO de TESTE : CPU (1)

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

```
architecture tb of tb is
```

```
component periferico is
```

```
port( reset, receive, ack : in std_logic;
```

```
acpt, send : out std_logic;
```

```
end component periferico;
```

```
input: in opsize;
```

```
output: out opsize );
```

```
signal entrada, saida : opsize;
```

```
signal reset, receive, ack, acpt, send : std_logic;
```

```
signal data : std_logic_vector(15 downto 0);
```

```
begin
```

MÓDULO de TESTE : CPU (2)

begin

-- SINAL DE RESET --

reset <= '1', '0' after 10 ns; -- reset da maquina

-- INSTANCIÇÃO DO MÓDULO PERIFÉRICO --

*perif1 : periferico port map(reset=>reset, receive=>receive, ack=>ack,
input=>entrada, acpt=>acpt, send=>send, output=>saida);*

-- IMPLEMENTAÇÃO DA PARTE RELATIVA AO COMPORTAMENTO DA CPU --

control :

process

variable contador : integer := 0;

*constant rom : mem_rom := mem_rom'("0101", "1111", "1010", "1001",
"0111", "1011", "0010", "0001", "1101", "1111", "1110", "0001",
"0111", "0011", "0010", "1001", others=>"0000");*

begin

wait until reset'event and reset='0';

MÓDULO de TESTE : CPU

-- envia 16 palavras de 4 bits, ou seja, 4 palavras de 16 bits

for i in 0 to 15 loop

 entrada <= rom(contador);

 contador := contador + 1;

 receive <= '1' after 10 ns;

 wait until acpt='1';

 receive <= '0' after 10 ns;

 wait until acpt='0';

end loop;



16 PALAVRAS DE 4 BITS
ou
4 palavras de 16 bits

for i in 4 downto 1 loop

 wait until send'event and send='1'; -- recebe do periferico os dados

 data(i*4-1 downto (i-1)*4) <= saida; -- 15..12, 11..8, 7..4, 3..0

 ack <= '1' after 10 ns;

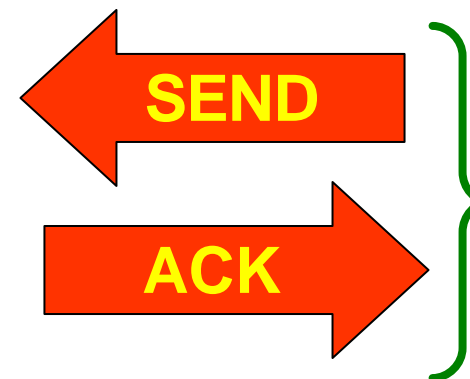
 wait until send'event and send='0';

 ack <= '0' after 10 ns;

end loop;

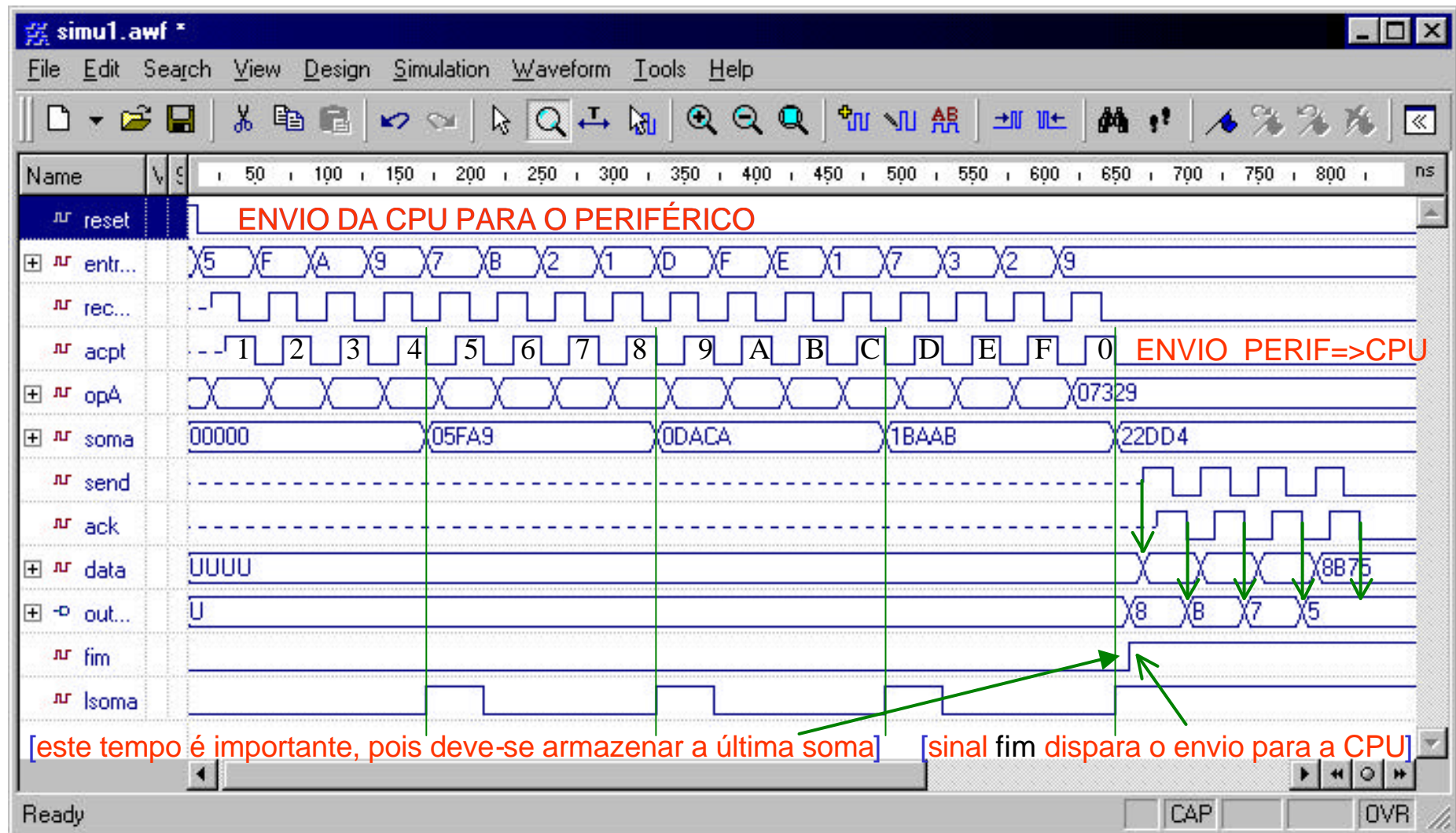
end process;

end tb;



4 PALAVRAS

SIMULAÇÃO DA COMUNICAÇÃO

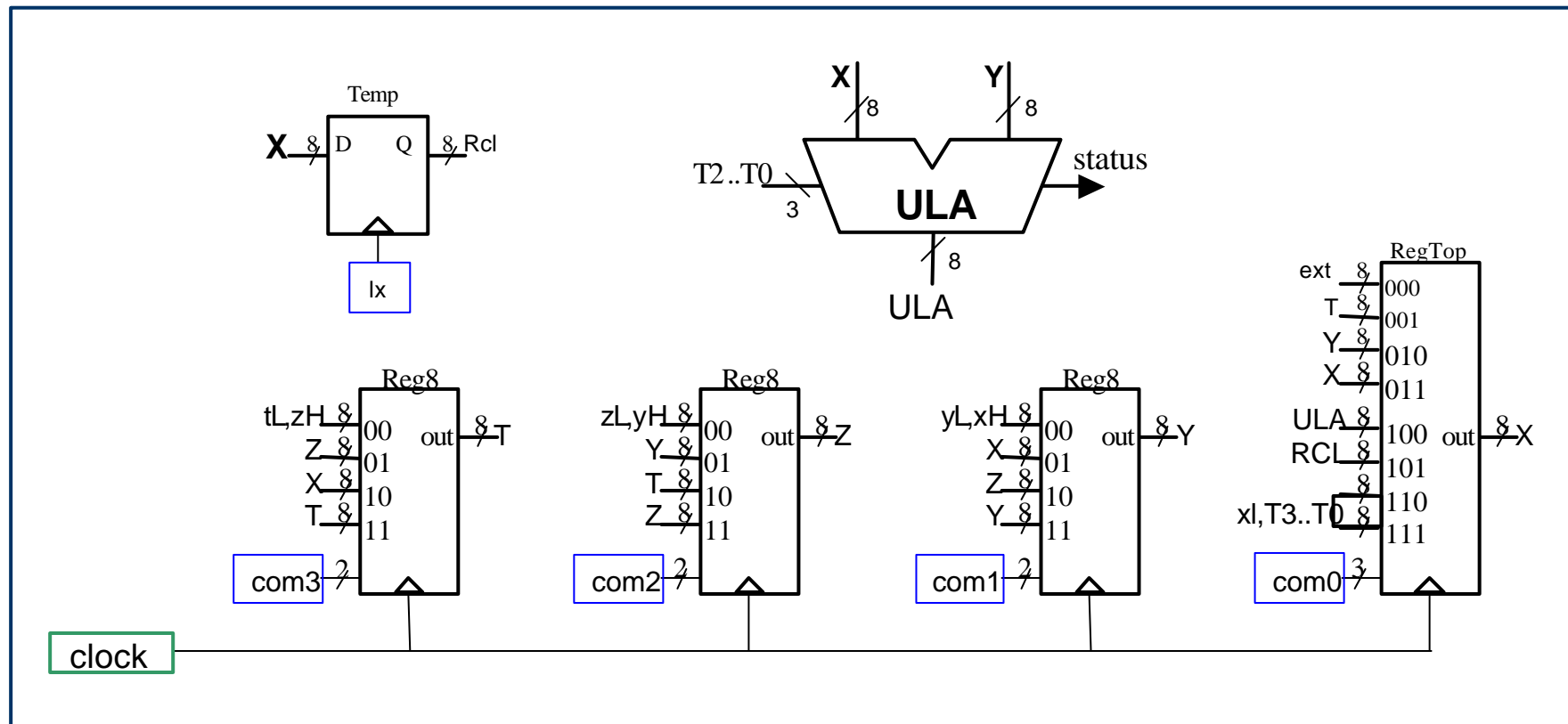


ESTUDOS DE CASO

- **ARQUITETURA CLEÓPATRA**
- **COMUNICAÇÃO ASSÍNCRONA**
- **CALCULADORA**

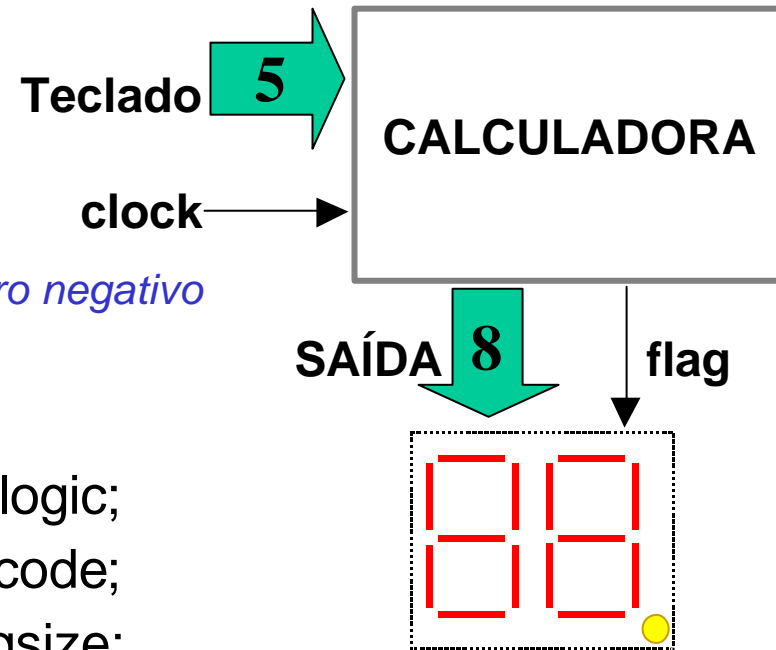
ESTUDO DE CASO 3 - CALCULADORA

- Calculadora tipo pilha, com estrutura das operações similar às calculadoras HP
- Comandos com[0,3] controlam os deslocamentos



CALCULADORA - Interface externa

- **Teclado:** entrada de 5 bits (palavra de 4 bits)
 - bit mais significativo igual a 0: instrução
 - bit mais significativo igual a 1: dado
- **Clock**
- **Saída:** 8 bits, correspondente ao regX
- **Flag :** indica transbordo (overflow) ou número negativo



entity calculadora **is**

```
port( clock : in std_logic;  
      teclado: in opcode;  
      saida : out regsize;  
      flag : out std_logic);
```

end;

CALCULADORA - package

- **Define os tipos básicos e constantes**

package calcHP is

*subtype **opcode** is std_logic_vector(4 downto 0);*

*subtype **regsize** is std_logic_vector(7 downto 0);*

*type **optxt** is (iadd, isub, iinc, idec, ilog, irs, ista, ircl, iup, idown, ixy, icpx, key);*

*type **mem_rom** is array (0 to 127) of opcode;*

*constant **add** : opcode := "00000";*

-- correspondente à especificação original

*constant **sub** : opcode := "00001";*

.....

*constant **cpx** : opcode := "01111";*

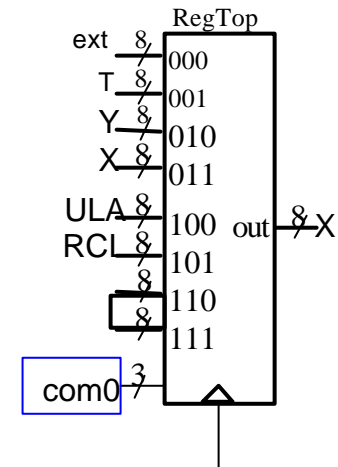
*procedure **somaAB** (signal A,B: in regsize; signal Cin: in STD_LOGIC;*

signal S: out regsize; signal Cout:out STD_LOGIC);

end calcHP;

CALCULADORA - Implementação 1

- *Estrutural, como na definição da calculadora*
- *Há um conjunto de registradores, comandados pelos sinais com0 a com3*
- *Esta implementação conterá 3 blocos:*
 - *registradores com atribuição síncrona ao relógio*
 - *geração dos sinais de comando, sincronamente ao relógio - CONTROLE*
 - *unidade lógico/aritmética combinacional*



Implementação 1 - registradores

```
process(clock)
```

```
begin
```

```
if clock'event and clock='0' then
```

```
  case com0 is
```

```
    when "001" =>
```

```
    when "010" =>
```

```
    when "011" =>
```

```
    when "100" | "000" =>
```

```
    when "101" =>
```

```
    when "110" | "111" =>
```

```
    when others =>
```

```
      regX <= regT;
```

```
      regX <= regY;
```

```
      regX <= regX;
```

```
      regX <= ULA;
```

```
      regX <= cte;
```

```
      regX <= regX(3 downto 0) & teclado(3 downto 0);
```

```
      regX <= "00000000";
```

```
  end case;
```

```
  case com1 is
```

```
    when "00" => regY <= regY(3 downto 0) & regX(7 downto 4);
```

```
    when "01" => regY <= regX;
```

```
    when "10" => regY <= regZ;
```

```
    when others => regY <= regY; -- caso 11 --
```

```
  end case;
```

```
-- o mesmo para reg Z e reg T
```

```
if (instruction=add and ( conv_integer(regX) + conv_integer(regY) )>255) or
```

```
(instruction=sub and ( conv_integer(regX) + conv_integer(regY) )<0)
```

```
then flag<='1'; else flag<='0'; end if;
```

```
end if;
```

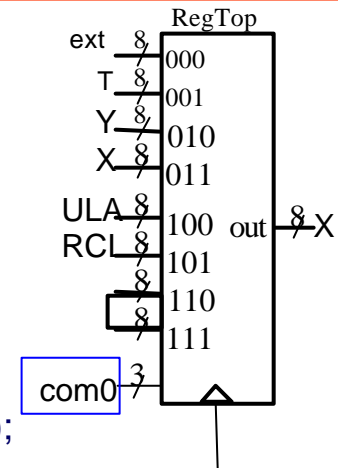
```
end process;
```

Borda de descida

Entrada via teclado

comportamental

função disponível



Implementação 1- ULA

- *Utilização de soma e subtração de forma comportamental, sem especificar o algoritmo*

-- ula nao depende do clock, e' um bloco combinacional

with teclado select

<i>ULA <=</i>	<i>regX + regY</i>	<i>when add,</i>	} comportamental
	<i>regY - regX</i>	<i>when sub,</i>	
	<i>regX + 1</i>	<i>when inc,</i>	
	<i>regX - 1</i>	<i>when dec,</i>	
	<i>regX and regY</i>	<i>when e,</i>	} operações de set/reset foram para a ULA
	<i>regX or regY</i>	<i>when ou,</i>	
	<i>regX xor regY</i>	<i>when oux,</i>	
	<i>not regX</i>	<i>when neg,</i>	
	<i>"00000000"</i>	<i>when resx,</i>	
	<i>"11111111"</i>	<i>when setx,</i>	
	<i>"00000000"</i>	<i>when others;</i>	

Uso de constantes

Implementação 1 - sinais de comando

-- parte de controle

process(clock)

begin

if clock'event and clock='1' then

Borda de subida



case teclado is

when add | sub | inc | dec | e | ou | oux | neg =>

com0 <= "100"; com1 <= "10"; com2 <= "10"; com3 <= "11";

when setx | resx =>

com0 <= "000"; com1 <= "11"; com2 <= "11"; com3 <= "11";

when sta =>

com0 <= "011"; com1 <= "11"; com2 <= "11"; com3 <= "11";

cte <= regX;

variável
temporária



when cxx =>

com0 <= "011"; com1 <= "01"; com2 <= "01"; com3 <= "01";

--- entrar todas as outras instruções

when others =>

com0 <= "111"; com1 <= "00"; com2 <= "00"; com3 <= "00";

-- entrada via teclado

end case;

end if;

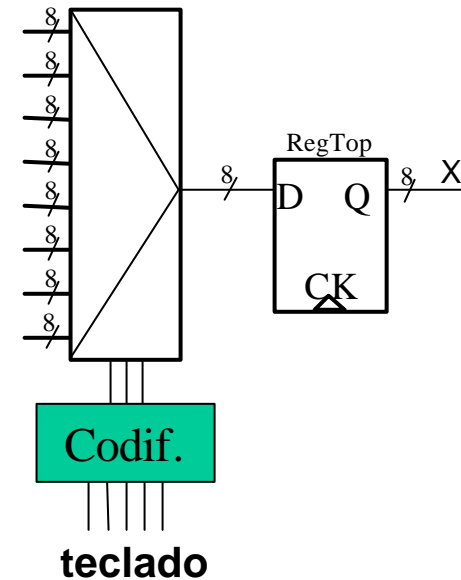
end process;

Implementação 1 - Críticas

- *Utilização de soma e subtração de forma comportamental, complica a geração dos sinais de controle, tipo flag (carry out)*
- *A codificação é complicada, pois à partir da instrução corrente gera-se um sinal de controle para ser utilizado nas atribuições.*

CALCULADORA - Implementação 2

- *Estrutural*
- *Codificação direta do sinais de comando*
 - o hardware é praticamente o mesmo, registrador com multiplexador na entrada
 - diferença: codificador do teclado na entrada do mux
- *Esta implementação conterà 2 blocos:*
 - registradores com atribuição síncrona ao relógio
 - unidade lógico/aritmética combinacional e estrutural



Implementação 2 - registradores

```
process(clock)
begin
  if clock'event and clock='0' then

    case teclado is
      when up =>          regX <= regT;
      when down | xy =>   regX <= regY;
      when sta | cpx =>   regX <= regX;
      when add | sub | inc | dec => regX <= soma;
      when e =>          regX <= regX and regY;
      when ou =>         regX <= regX or regY;
      when oux =>        regX <= regX xor regY;
      when neg =>        regX <= not regX;
      when setX =>       regX <= "11111111";
      when resX =>       regX <= "00000000";
      when rcl =>        regX <= cte;
      when others =>     regX <= regX(3 downto 0) & instruction(3 downto 0);
    end case;

    case teclado is
      when rcl | up | xy | cpx          => regy <= regX;
      when add | sub | e | ou | oux | neg | down => regY <= regZ;
      when inc | dec | setX | resX | sta  => regY <= regY;
      when others => regY <= regY(3 downto 0) & regX(7 downto 4);
    end case;

    ... continua
  end if;
end process;
```

Borda de descida

Saída da ULA

Entrada via teclado

Implementação 2 - registradores

... continuação

```
case teclado is
  when rcl | up | cpx          => regZ <= regY;
  when add | sub | e | ou | ous | neg | down => regZ <= regT;
  when inc | dec | setX | resX | sta | xy   => regZ <= regZ;
  when others => regZ <= regZ(3 downto 0) & regY(7 downto 4);
end case;
```

```
case teclado is
  when up | cpx | rcl          => regT <= regZ;
  when down                    => regT <= regX;
  when add | sub | inc | dec | e | ou | ous |
    neg | setX | resX | sta | xy => regT <= regT;
  when others => regT <= regT(3 downto 0) & regZ(7 downto 4);
end case;
```

```
case teclado is
  when sta => cte <= regX;
  when others => null;
end case;
```

```
if (teclado=add and cout='1') or (teclado=sub and cout=0)
  then flag<='1';
  else flag<='0';
end if;
```

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

Armazenamento da constante junto
aos registradores

Utilizando a procedure
somaAB, o controle do
flag é simplificado

Implementação 2 - ULA

- *A ula se resume ao somador e à seleção dos operandos*

-- somador nao depende do clock, e' um bloco combinacional

somaAB(opA, opB, cin, soma, cout);



estrutural
(procedure definida no package)

-- determina os operandos para o somador da ULA (opA, opB e cin)

with teclado select

*opA <= regY when sub,
regX when others;*

with teclado select

*opB <= not(regX) when sub,
"00000000" when inc,
"11111111" when dec,
regY when others;*

with teclado select

*Cin <= '1' when sub | inc,
'0' when others;*

Test bench - entidade para simulação

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

-- sem interface externa

```
architecture estrutural of tb is
```

```
component calculadora is
```

```
port( clock : in std_logic;          teclado: in opcode  
      saida : out regsize; flag : out std_logic);
```

```
end component;
```

```
signal instruction : opcode;
```

```
signal ck, flag : std_logic;
```

```
signal saida : regsize;
```

```
signal debug : optxt;
```


-- exibe o texto da instrução corrente

```
signal cin, cout : std_logic;
```

-- programa : testa todos os operando

```
constant ROM1 : mem_rom := mem_rom'(setx, cpx, cpx, cpx, add,  
  "1000", "11000", cpx, cpx, add, cpx, add, add,  
  "10001", "11000", xy, sub, xy, setx, up, setx,  
  down, sta, setx, resx, rcl, others=>"00000");
```

instância da
calculadora



ROM com programa
armazenando



```
begin
```

-- continua ...

Test bench - entidade para simulação

begin

```
calc : calculadora port map(clock=>ck, teclado=>instruction, saida=>saida, flag=>flag);
```

process

```
variable contador : integer range 0 to 255 := 0;
```

begin

```
ck <= '1' after 10ns, '0' after 20ns;
```

```
instruction <= rom1(contador);
```

```
wait for 20ns;
```

```
contador := contador + 1;
```

case instruction is

```
when add => debug <= iadd;
```

```
when inc => debug <= iinc;
```

```
when e | ou | oux | neg => debug <= ilog; when setx | resx => debug <= irs;
```

```
when sta => debug <= ista;
```

```
when up => debug <= iup;
```

```
when xy => debug <= ixy;
```

```
when others => debug <= key;
```

end case;

end process;

end estrutural;

instância da calculadora

gera o clock e lê da ROM
-- avança uma posição na ROM

debug, exibe a instrução corrente no simulador

Configuração

- *Já temos descritas duas arquiteturas, um test_bench, a entidade e um package.*
- *Falta agora indicar ao simulador qual das arquiteturas serão utilizadas.*
- *A configuração só é necessária quando há mais de uma arquitetura.*

```
configuration conf1 of tb is                                -- conf1: nome da calculadora
    for estrutural;                                       -- estrutural: nome da arquitetura da conf1
        for calc : calculadora;                          -- calc:calculadora: instância e componente
            use entity work.calculadora(rtl);             -- arquitetura da instância
        end for;
    end for;
end conf1;
```

SIMULAÇÃO

(1) flag: $t=100ns$ overflow, $t=360ns$ número negativo

(2) cte: variável auxiliar

