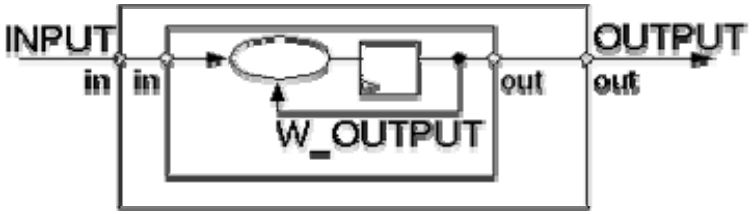
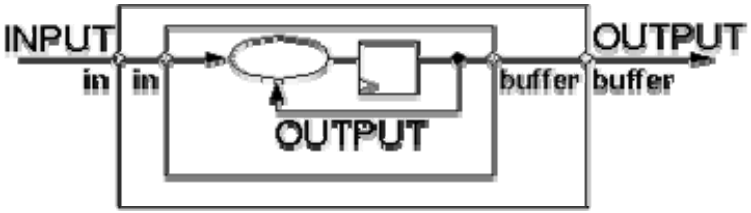
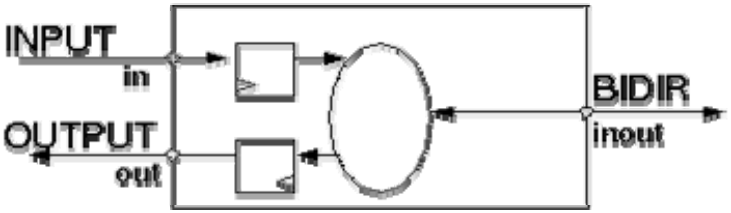
Modos dos pinos da Entity e de Sinais Internos

Modos dos pinos da entidade







in:

Valor do sinal do pino é read-only

out:

Valor do sinal do pino é write-only

buffer:

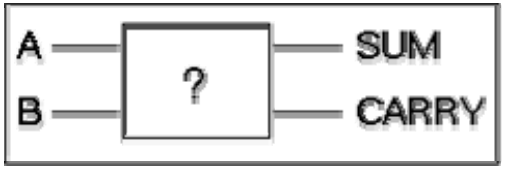
O sinal pode ser tambem lido da saída

inout:

Pino bidirecional

Exemplos de entidades

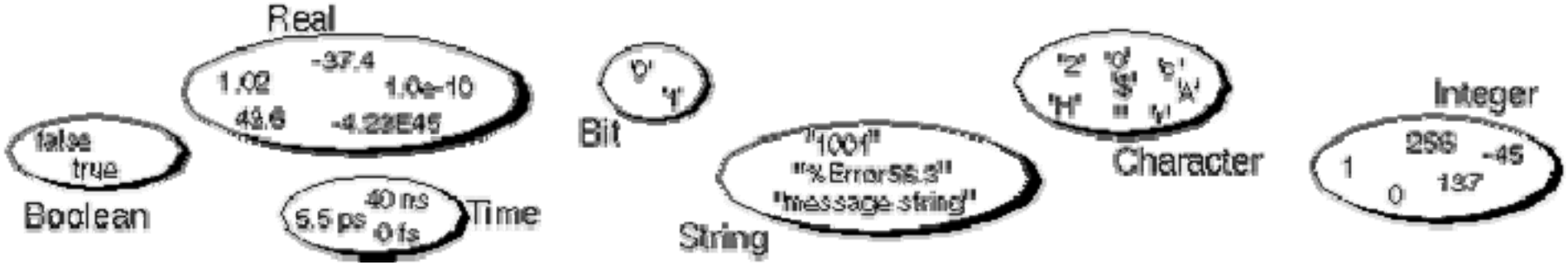
```
entity HALFADDER is
   port(
      A, B:              in   std_logic;
      SUM, CARRY: out std_logic);
end HALFADDER;
-- VHDL'93: end entity HALFADDER
```




```
entity ADDER is
    port(
      A, B:              in      std_logic_vector(3 downto 0;
      SUM:               out   std_logic_vector(3 downto 0);
      CARRY:             out   std_logic );
end ADDER;
```

Tipos de dados de pinos e sinais e declaração de sinais

Standard Data Types




```
package STANDARD is
   type BOOLEAN is (FALSE,TRUE);
   type BIT is (`0`,`1`);
   type CHARACTER is (-- ascii set);
   type INTEGER is range
                        -- implementation defined
   type REAL is range
                        --  implementation defined
   -- BIT_VECTOR, STRING, TIME
end STANDARD;
```

Tipo de dado: TIME

- **Uso:**
  - **testbenches**
  - **Atraso de portas**
- **Unidades de tempo disponíveis:**
  - **fs,**
  - **ps,**
  - **ns,**
  - **us,**
  - **ms,**
  - **sec,**
  - **min,**
  - **hr**

```
architecture EXAMPLE of TIME_TYPE is
  signal CLK : bit := `0';
  constant PERIOD : time := 50 ns;

begin
  process
  begin
    wait for 50 ns;
    . . .
    wait for  PERIOD ;
    . . .
    wait for 5 *  PERIOD ;
    . . .
    wait for  PERIOD  * 5.5;
  end process;

. . .

  -- concurrent signal assignment
  CLK <= not CLK after 0.025 us;
  -- or with constant time
  -- CLK <= not CLK after PERIOD/2;
end EXAMPLE;
```

Tipo de dados: Bit

- **O tipo de dado deve estar especificado no:**
  - **Port**
  - **Declaração dos sinais**
- **Os tipos devem ser compativeis no assinalamento.**

```
entity FULLADDER is
   port(A, B, CARRY_IN: in   bit;
         SUM, CARRY:      out bit);
end FULLADDER;

architecture MIX of FULLADDER is

begin
   SUM <= A xor B xor CARRY_IN;
   CARRY <= (A and B) or (A and CARRY_IN)
or (B and CARRY_IN);

end MIX;
```

Problemas com o tipo: BIT

type BIT is ('0', '1')

- **Valores `0` e `1`, apenas**
  - **Valor padrão `0`**
- **Para simulação e sintese é preciso outros valores como:**
  - **Não inicializado**
  - **Alta impedancia**
  - **Não definido**
  - **Não interessa (X)**
  - **Diferentes correntes**

Tipo Binário com multiplos valores: STD_LOGIC

- **IEEE-standard**
- **9 valores padronizados pela IEEE**
- **standard IEEE 1164 (STD_LOGIC_1164)**

IEEE Standard Logic Type

```
type STD_ULOGIC is (
        ` U `,   -- uninitialized
        ` X `,   -- strong 0 or 1 (= unknown)
        ` 0 `,   -- strong 0
        ` 1 `,   -- strong 1
        ` Z `,   -- high impedance
        ` W `,   -- weak 0 or 1 (= unknown)
        ` L `,   -- weak 0
        ` H `,   -- weak 1
        ` - `,   -- don`t care);
```

Sinais

- 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 : tipo [restrição] [:=expressão];
- exemplo
  - `signal cont : integer range 50 downto 1;`
  - `signal ground : bit := ‘0’;`
  - `signal bus : bit_vector;`

Exemplo de declaração de sinais

signal <nome> **:** tipo **:=** inicialização;

```
Library ieee;
Use ieee.std_logic_1164.all;

entity FULLADDER is
   port(A, B, CARRY_IN: in   std_logic;
         SUM, CARRY:      out std_logic);
end FULLADDER;

architecture MIX of FULLADDER is
signal x : std_logic;
begin
   x <= A xor B;
   SUM <= x xor CARRY_IN;
   CARRY <= (A and B) or (A and CARRY_IN)
or (B and CARRY_IN);

end MIX;
```

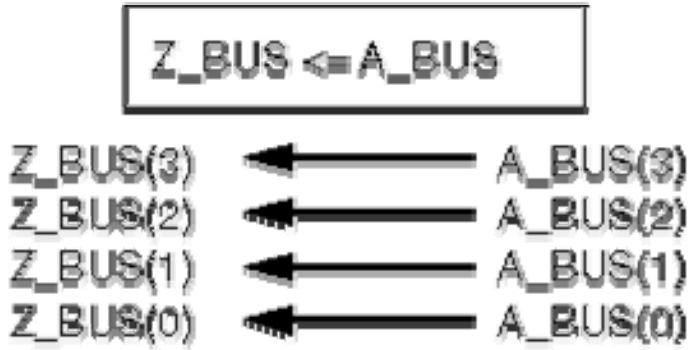
Tipo de dados: Std_logic

- **O tipo de dado deve estar especificado no:**
  - **port**
  - **Declaração dos sinais**
- **Os tipos devem ser compativeis no assinalamento.**

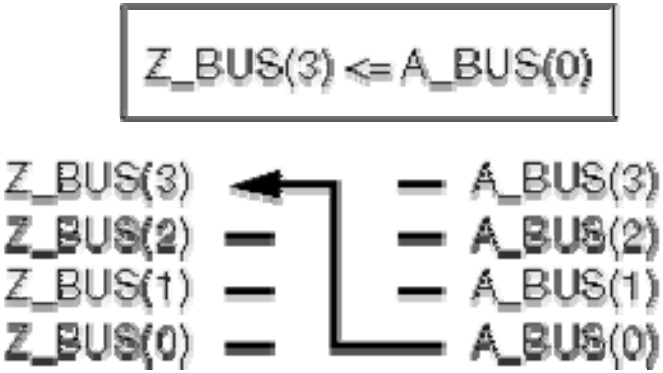
```
Library ieee;
Use ieee.std_logic_1164.all;

entity FULLADDER is
   port(A, B, CARRY_IN: in   std_logic;
         SUM, CARRY:      out std_logic);
end FULLADDER;

architecture MIX of FULLADDER is

begin
   SUM <= A xor B xor CARRY_IN;
   CARRY <= (A and B) or (A and CARRY_IN)
or (B and CARRY_IN);

end MIX;
```

Definição de Array

- **Coleção de sinais do mesmo tipo**
- **Array pre-definido**
  - **bit_vector (array of bit)**
  - **string (array of character)**
- **A definição do tamanho do array é dada na declaração do sinal ou do pino.**



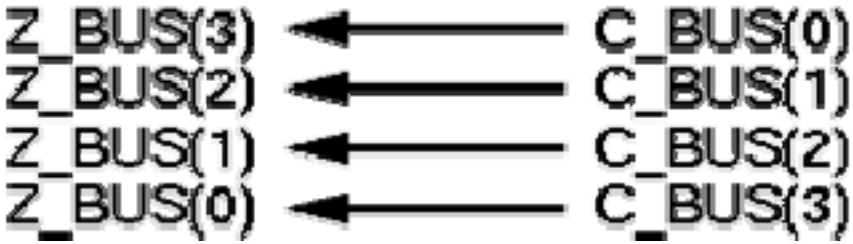
signal A_BUS, Z_BUS : bit_vector (3 downto 0);



signal A_BUS, Z_BUS : std_logic_vector(3 downto 0);

Assignments with Array Types

```
architecture EXAMPLE of ARRAYS is
   signal Z_BUS : std_logic_vector (3 downto 0);
   signal C_BUS : std_logic_vector (0 to 3);
begin
   Z_BUS <= C_BUS;
end EXAMPLE;
```



Cuidado ao usar (X downto 0) ou (0 to X)

Tipo Integer, Signed e Unsigned

- Os numeros inteiros podem variar de $-2^{31} + 1$ to $+2^{31} - 1$
- (-2147483647 to +2147483647).
- A representação padrão é em decimal.
- Para utilizar uma outra representação é preciso mostrar explicitamente:

binary	2#...#
Octal	8#...#
Hexadecimal	16#...#

Tipos de assinalamento para 'std_logic'

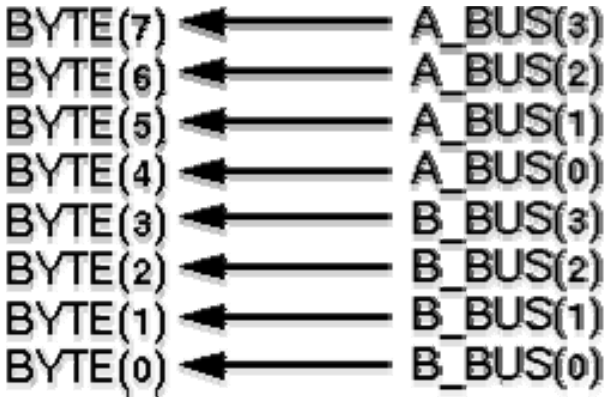
```
architecture EXAMPLE of ASSIGNMENT is

   signal Z_BUS :     std_logic_vector (3 downto 0);
   signal BIG_BUS : std_logic_vector (15 downto 0);

begin

   -- legal assignments:
   Z_BUS(3) <= `1`;
   Z_BUS      <= ``1100``;

   Z_BUS      <=  b`` 1100 ``;
   Z_BUS      <=  x`` c ``;
   Z_BUS      <=  X`` C ``;
   BIG_BUS  <=  B` `0000 _ 0001_0010_0011 ``;

end EXAMPLE;
```

Simples para um digito

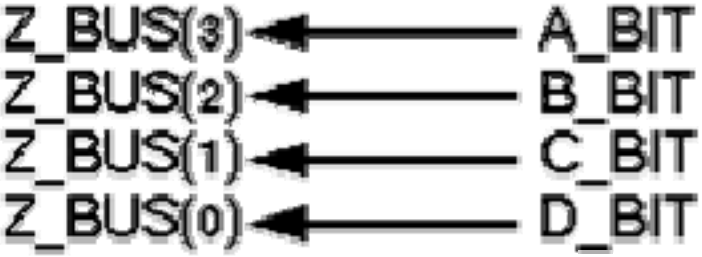
Aspas duplas para multiplos digitos

Concatenação (&)

```
architecture EXAMPLE_1 of CONCATENATION is
   signal BYTE              : std_logic_vector (7 downto 0);
   signal A_BUS, B_BUS : std_logic_vector (3 downto 0);
begin

   BYTE   <= A_BUS & B_BUS;

end EXAMPLE;
```




```
architecture EXAMPLE_2 of CONCATENATION is
   signal Z_BUS : std_logic_vector (3 downto 0);
   signal A_BIT, B_BIT, C_BIT, D_BIT : bit;
begin

   Z_BUS <= A_BIT & B_BIT & C_BIT & D_BIT;

end EXAMPLE;
```



Lado direito do assinalamento

Agregação

```
architecture EXAMPLE of AGGREGATES is
   signal BYTE :  std_logic_vector (7 downto 0);
   signal Z_BUS : std_logic_vector (3 downto 0);
   signal A_BIT, B_BIT, C_BIT, D_BIT : std_logic;
begin
   Z_BUS <= ( A_BIT, B_BIT, C_BIT, D_BIT ) ;
    ( A_BIT, B_BIT, C_BIT, D_BIT ) <= std_logic'(``1011``);
    ( A_BIT, B_BIT, C_BIT, D_BIT ) <= BYTE(3 downto 0);

   BYTE <= ( 7 => `1`, 5 downto 1 => `1`, 6 => B_BIT, others => `0`) ;
end EXAMPLE;
```

Utilizando Porções de Arrays

```
architecture EXAMPLE of SLICES is
   signal BYTE : std_logic_vector (7 downto 0);
   signal A_BUS, Z_BUS : std_logic_vector (3 downto 0);
   signal A_BIT : bit;
begin
   BYTE (5 downto 2) <= A_BUS;
   --BYTE (5 downto 0) <= A_BUS;              -- wrong

   Z_BUS (1 downto 0) <= `0` & A_BIT;
   Z_BUS                      <= BYTE (6 downto 3);
   --Z_BUS (0 to 1)       <= `0` & B_BIT;      -- wrong

   A_BIT   <= A_BUS (0);
end EXAMPLE;
```

Tipos Enumerados

```
architecture EXAMPLE of ENUMERATION is

   type T_STATE is (RESET, START, EXECUTE, FINISH);

  signal CURRENT_STATE, NEXT_STATE : T_STATE ;
  signal TWO_BIT_VEC : std_logic_vector(1 downto 0);

begin

  -- valid signal assignments
  NEXT_STATE        <= CURRENT_STATE;
  CURRENT_STATE <= RESET;

  -- invalid signal assignments
  CURRENT_STATE <= "00";
  CURRENT_STATE <= TWO_BIT_VEC;

end EXAMPLE;
```

Multidimensional Arrays

```
architecture EXAMPLE of ARRAY is

    type  INTEGER_VECTOR  is
        array (1 to 8) of integer;

  -- 1 --
    type  MATRIX_A  is
        array(1 to 3)  of  INTEGER_VECTOR ;
  -- 2 --
  type  MATRIX_B  is
        array(1 to 4, 1 to 8)  of  integer ;

   signal MATRIX_3x8 : MATRIX_A;
   signal MATRIX_4x8 : MATIRX_B;

begin

    MATRIX_3x8(3)(5)  <= 10;  --array of array

    MATRIX_4x8(4, 5)   <= 17;  -- 2 dim array

end EXAMPLE;
```

2 possibilities
- **array of array**
- **multidimensional array**

- **Different referencing**

- **Barely supported by synthesis tools**

```
architecture EXAMPLE of AGGREGATE is

    type INTEGER_VECTOR is
        array (1 to 8) of integer;
    type MATRIX_A is
        array(1 to 3) of INTEGER_VECTOR;

    type MATRIX_B is
        array(1 to 4, 1 to 8) of integer;

   signal MATRIX3x8 : MATRIX_A;
   signal MATRIX4x8 : MATIRX_B;
   signal VEC0, VEC1,
            VEC2, VEC3  :  INTEGER_VECTOR;
begin

   MATRIX3x8 <= (VEC0, VEC1, VEC2);
   MATRIX4x8 <= (VEC0, VEC1, VEC2, VEC3);

   MATRIX3x8 <= (others => VEC3);
   MATRIX4x8 <= (others => VEC3);

   MATRIX3x8 <= (others => (others => 5));
   MATRIX4x8 <= (others => (others => 5));

end EXAMPLE;
```

Type Conversion

```
architecture EXAMPLE of CONVERSION is
    type MY_BYTE is array (7 downto 0) of std_logic;

    signal VECTOR:      std_logic_vector(7 downto 0);
    signal SOME_BITS:  bit_vector(7 downto 0);
    signal BYTE:        MY_BYTE;

begin

    SOME_BITS <= VECTOR;                    -- wrong
    SOME_BITS <= Convert_to_Bit( VECTOR ) ;

    BYTE <= VECTOR;                         -- wrong
    BYTE <= MY_BYTE( VECTOR ) ;

end EXAMPLE;
```

Conversions

Operator	Argument (arg)	Result
conv_integer(arg)	std_logic std_logic_vect or signed unsigned	integer
conv_unsigned(arg, size: integer) size: number of bits of the final result	std_logic signed unsigned integer	unsigned
conv_signed(arg, size: integer) size: number of bits of the final result	std_logic signed unsigned integer	signed
conv_std_logic_vector(arg, size: integer) size: number of bits of the final result	std_logic signed unsigned integer	std_logic_vector

Subtypes

```
architecture EXAMPLE of SUBTYPES is
    type MY_WORD is array (15 downto 0) of std_logic;
    subtype SUB_WORD is std_logic_vector (15 downto 0);

    subtype MS_BYTE is integer range 15 downto 8;
    subtype LS_BYTE is integer range 7 downto 0;

    signal VECTOR:     std_logic_vector(15 downto 0);
    signal SOME_BITS: bit_vector(15 downto 0);
    signal WORD_1: MY_WORD;
    signal WORD_2: SUB_WORD;

begin
    SOME_BITS <= VECTOR;                    -- wrong
    SOME_BITS <= Convert_to_Bit(VECTOR);

    WORD_1 <= VECTOR;                       -- wrong
    WORD_1 <= MY_WORD(VECTOR);

    WORD_2 <= VECTOR;                       -- correct!

    WORD_2(LS_BYTE) <= "11110000";
end EXAMPLE;
```

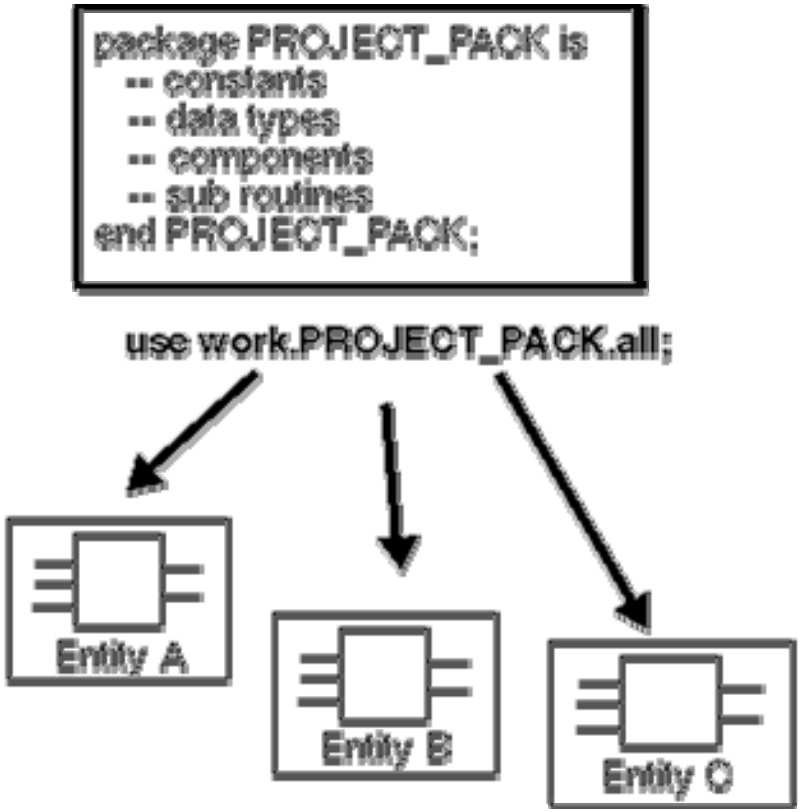
UFRGS

Aliases

```
architecture EXAMPLE of ALIAS is
    signal DATA is bit_vector(9 downto 0);

    alias STARTBIT :   bit is DATA(9) ;
    alias MESSAGE:  bit_vector(6 downto 0) is DATA (8 downto 2);
    alias PARITY:       bit is DATA(1);
    alias STOPBIT:    bit is DATA(0);
    alias REVERSE:  bit_vector(1 to 10) is DATA;

    function calc_parity(data: bit_vector) return bit is
    . . .

begin
    STARTBIT        <= '0';
    MESSAGE        <= "1100011";
    PARITY          <= calc_parity(MESSAGE);
    REVERSE(10) <= '1';

end EXAMPLE;
```

Package

- Coleção de definições, tipos de dados e sub programas ou funções
- Referencia feita pelo grupo de projeto
- Vantagem:
  - Qualquer mudança é vista por todo o time rapidamente.
  - Mesmo tipo de dado e uso ("downto vs. to")
  - Funções para todosl



Declaração de sinais e constantes

architecture name_type **of** name_entity **is**

signal A, B, C : std_logic := ‘0’;

signal D : std_logic_vector(3 downto 0) := “0000”;

signal F: integer range 0 to 4;

constant G : integer := 9;

inicialização

BEGIN

Operadores e Comandos

Expressões

- 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 ?

Operadores Lógicos

Prioridade

- **not (maior prioridade)**
- **and, or, nand, nor, xor, xnor (mesma prioridade)**

Pre-definidos para

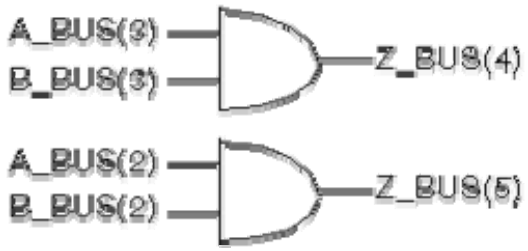
- **bit, bit_vector**
- **boolean**

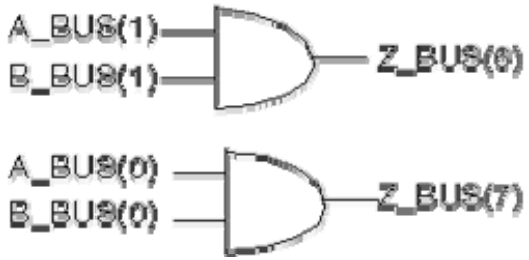
- **Para std_logic precisa usar a library ieee.std_logic_1164.all;**

```
entity LOGIC_OP is
   port (A, B, C, D : in   bit;
          Z1:              out bit;
          EQUAL :    out boolean);
end LOGIC_OP;

architecture EXAMPLE of LOGIC_OP is
begin

   Z1 <= A  and (B or (not C xor D)));

   EQUAL <= A xor B;                    -- wrong

end EXAMPLE;
```

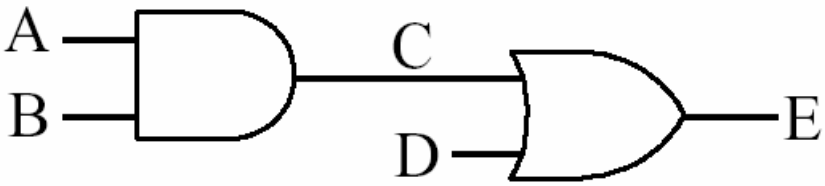
Operados Lógicos em Arrays

```
architecture EXAMPLE of LOGICAL_OP is
   signal A_BUS, B_BUS : std_logic_vector (3 downto 0);
   signal Z_BUS :        std_logic_vector (4 to 7);
begin
   Z_BUS <= A_BUS and B_BUS;
end EXAMPLE;
```





Comandos concorrentes



Comandos concorrentes:

C <= A **and** B **after** 5 ns;

E <= C **or** D **after** 5 ns;

Se um atrado não é especificado, um “delta” atraso é assumido

C <= A **and** B;

E <= C **or** D;

A ordem do comando concorrente não é importante

E <= C **or** D;

C <= A **and** B;

Esse comando executa repedidamente

CLK <= **not** CLK **after** 10 ns;

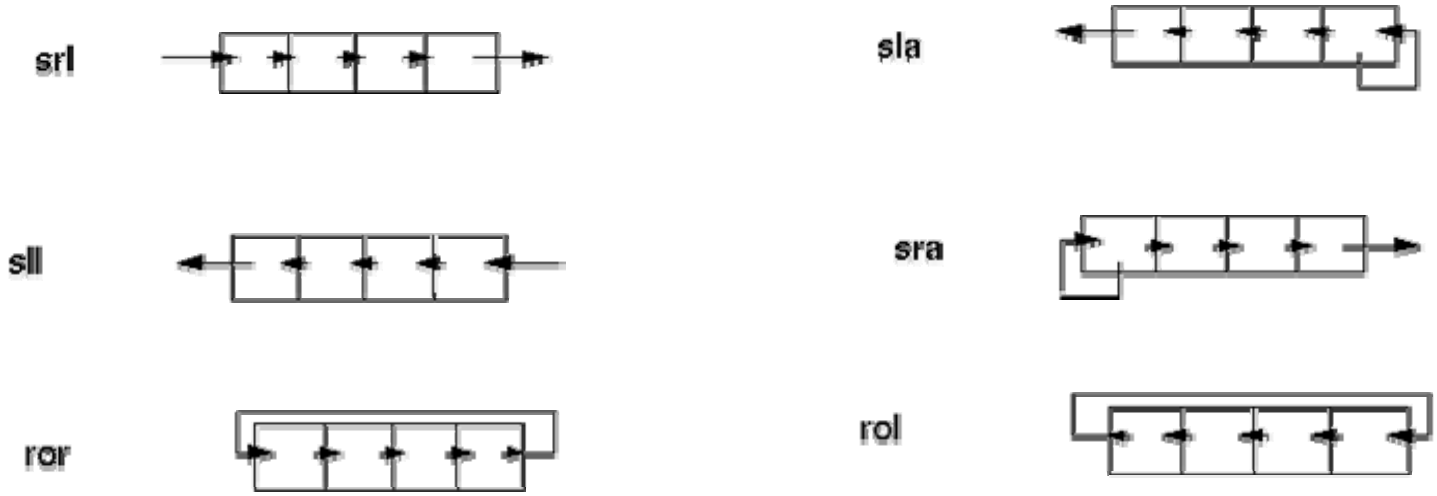
Este comando causa um erro na simulação

CLK <= **not** CLK;

Operadores de Deslocamento: só funcionam com bit_vector

```
signal A_BUS, B_BUS, Z_BUS : bit_vector (3 downto 0);

Z_BUS <= A_BUS sll 2;
Z_BUS <= B_BUS sra 1;
Z_BUS <= A_BUS ror 3;
```

At the end, the first value of the type is used for filling up



Assinalamento condicional

Comandos concorrentes

```
TARGET <= VALUE_1 when CONDITION_1 else
          VALUE_2 when CONDITION_2 else
          . . .
          VALUE_n;
```

Assinalamento seletivo

Comandos concorrentes

```
with EXPRESSION select

   TARGET <= VALUE_1 when CHOICE_1,
             VALUE_2 when CHOICE_2 | CHOICE_3,
             VALUE_3 when CHOICE_4 to CHOICE_5,
             VALUE_n when others;
```

Comandos concorrentes

- ATRIBUIÇÃO DE SINAIS

  alu_result <= op1 + op2;

- ATRIBUIÇÃO SELETIVA DE SINAIS

```
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;
```

- Tem semântica similar ao comando CASE em um processo

Operadores de relação

```
BEGIN

S_out <= "00" when A <=B else
          "11" when A="1100" else
          "01";

END;
```

<=	less or equal
<	less than
=	equal
/=	unequal
>=	greater or equal
>	greater

Importante

- Comandos seletivos são puramente combinacionais e por isso, todas as possibilidades devem estar cobertas no comando.

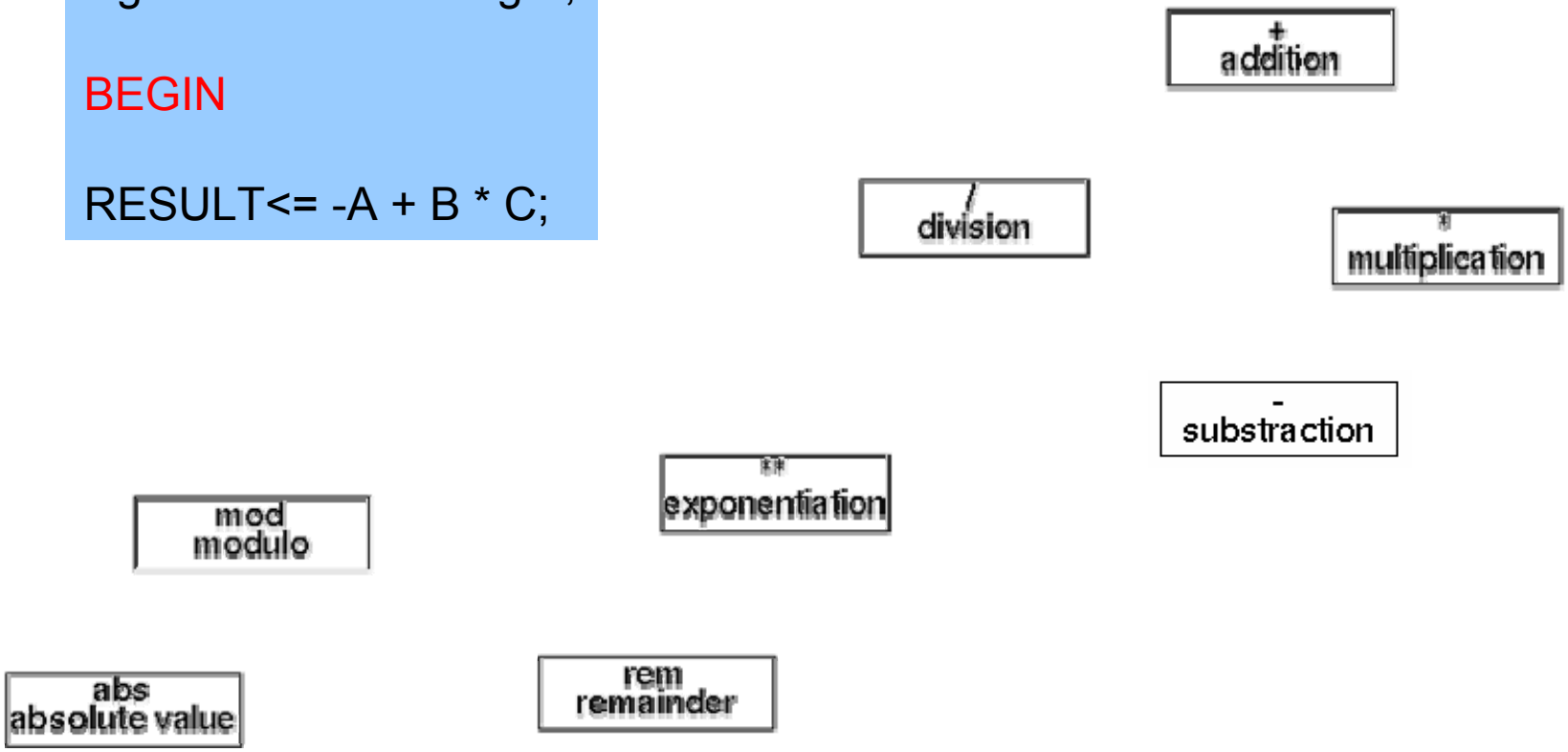
```
TARGET <= VALUE_1 when CONDITION_1 else
          VALUE_2 when CONDITION_2 else
          VALUE_n;
```

```
with EXPRESSION select

  TARGET <= VALUE_1 when CHOICE_1,
            VALUE_2 when CHOICE_2 | CHOICE_3,
            VALUE_3 when CHOICE_4 to CHOICE_5,
            VALUE_n when others;
```

- Caso isso não aconteça, surgirão latches e flip-flops indesejados na sintese o que irá comprometer o desempenho do circuito.

Operadores Aritméticos

```
signal A, B, C:    integer;
signal RESULT: integer;

BEGIN

RESULT<= -A + B * C;
```



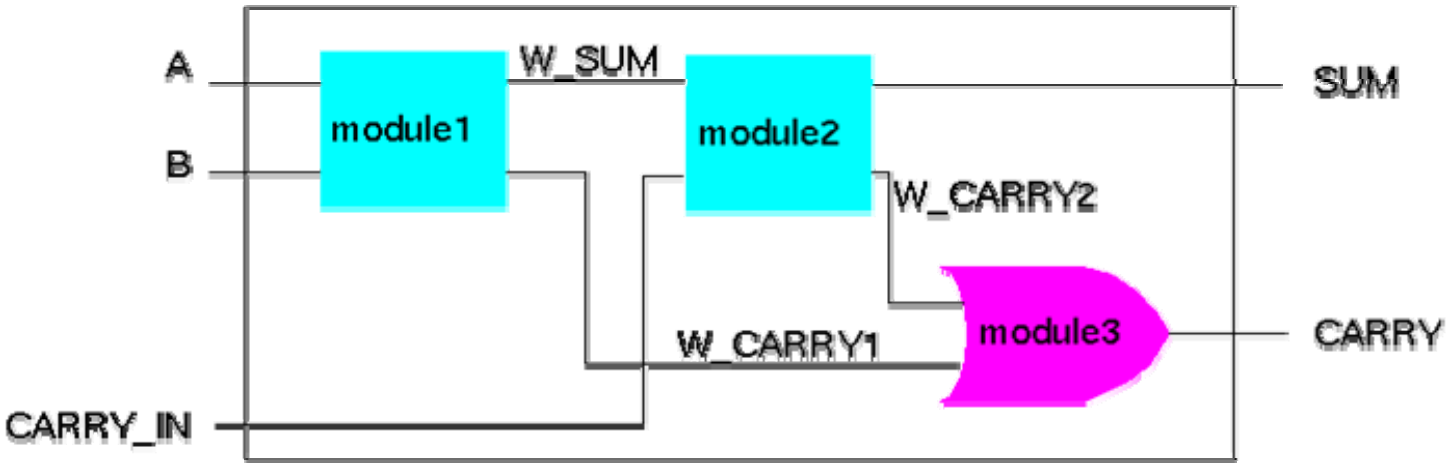
Uso da biblioteca ieee.std_logic_arith.all;

Declaração e Instanciação de Componentes

Declaração de componente

```
entity FULLADDER is
   port (A,B, CARRY_IN: in  std_logic;
          SUM, CARRY:     out  std_logic);
end FULLADDER;

architecture STRUCT of FULLADDER is
   signal W_SUM, W_CARRY1, W_CARRY2 :  std_logic;

    component  HALFADDER
      port (A, B :                in    std_logic;
             SUM, CARRY : out  std_logic);
    end component;

    component  ORGATE
      port (A, B : in    std_logic;
             RES : out  std_logic);
    end component;

begin
. . .
```

Modelo Hierarquico



Full adder: 2 halfadders + 1 OR-gate

Instanciação de Componente

```
architecture STRUCT of FULLADDER is
  component HALFADDER
    port (A, B :              in    std_logic;
          SUM, CARRY : out  std_logic);
  end component;
  component ORGATE
    port (A, B : in   std_logic;
          RES : out  std_logic);
  end component;

  signal W_SUM, W_CARRY1, W_CARRY2: std_logic;

begin

    MODULE1: HALFADDER
     port map( A, B, W_SUM, W_CARRY1 );

    MODULE2: HALFADDER
     port map ( W_SUM, CARRY_IN,
                SUM, W_CARRY2 );

    MODULE3: ORGATE
     port map ( W_CARRY2, W_CARRY1, CARRY );

end STRUCT;
```

Assinalamento dos sinais (ports) pela ordem dos pinos na interface do componente

Instanciação de Componente: associação de nomes dos ports

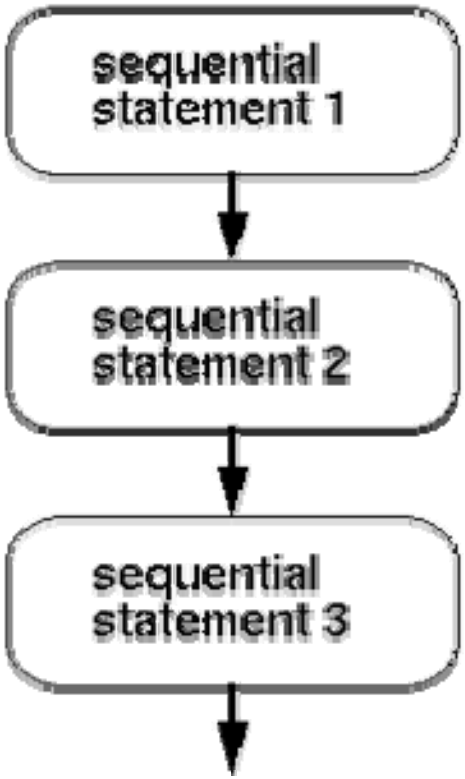

```
entity FULLADDER is
  port (A,B, CARRY_IN: in    std_logic;
        SUM, CARRY:    out  std_logic);
end FULLADDER;

architecture STRUCT of FULLADDER is

  component HALFADDER
    port (A, B :              in  std_logic;
          SUM, CARRY : out  std_logic);
  end component;
. . .
  signal W_SUM, W_CARRY1, W_CARRY2 :  std_logic;

begin

   MODULE1: HALFADDER
              port map (  A        => A,
                          SUM      => W_SUM,
                          B        => B,
                          CARRY    => W_CARRY1 );
  . . .
end STRUCT;
```

Assinalamento explicito

7 e 8
Comandos Sequencias

Comandos Sequenciais

- Execução de acordo com a ordem com que os comandos sequenciais aparecem.
- Permitido apenas dentro da estrutura **process**
- Usado para representar algoritmos.



Process

- Contem comandos sequenciais
- Existe apenas dentro da arquitetura
- Todos os **process** rodam ao mesmo tempo de maneira concorrente.
- A execução dos **process** são controladas por:
  - Lista de sensibilidade (sinais de trigger para a execução do **process**), ou
  - Comandos de **wait**
- O label do process é opcional

```
entity AND_OR_XOR is
  port (A,B :                          in    bit;
          Z_OR, Z_AND, Z_XOR : out bit);
end AND_OR_XOR;

architecture RTL of AND_OR_XOR is
begin

   A_O_X:  process  (A, B)
    begin
      Z_OR   <= A or    B;
      Z_AND <= A and B;
      Z_XOR <= A xor  B;
     end process  A_O_X ;

end RTL;
```

Process (3)

- Resides in the architecture's body
- A process is like a circuit part, which can be
  a) active (known as *activated*)
  b) inactive (known as *suspended*)
- It's statements will be executed sequentially top-down until the end of the process
  – Written order of statements matters, unlike in concurrent statements
- However, all signal assignments take place *when process exits*
  – Forgetting this is a Top-3 mistake for beginners

```
b <= 1; -- b was 5
c <= b; -- c gets the old value of b, i.e. 5
```

  – Last assignment to a signal will be kept

Process's Sensitivity List

- A process is activated when any of the signals in the *sensitivity list* changes its value
- Process must contain either sensitivity list or *wait* statement(s), but NOT both
  - Similar behavior, but sensitivity list is much more common
- General format:

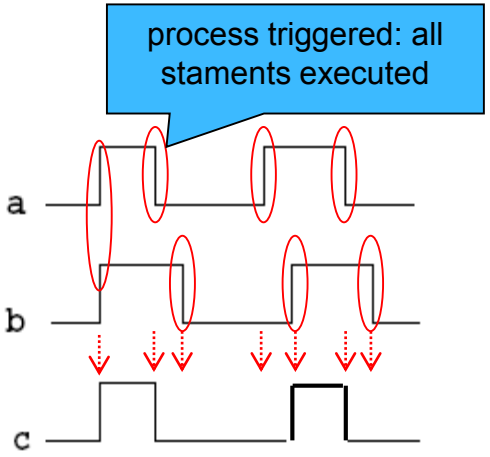
```
label: PROCESS[(sensitivity_list)]
   process_declarative_part
BEGIN
   process_statements
   [wait_statement]
END PROCESS;
```

Either but not both

Example Sensitivity List

- Process with sensitivity list:

```
ex_p: PROCESS(a,b)
BEGIN
   c <= a AND b;
END PROCESS ex_p;
```



- Process is executed when value of `a` or `b` changes
  - Type of `a` and `b` can be arbitrary: scalar, array, enumeration, or record
  - `ex_p` is a user defined label (recommended)

Example (2)

- The same process with wait statement:

```
PROCESS
BEGIN
   WAIT ON a,b;
   c <= a AND b;
END PROCESS;
```

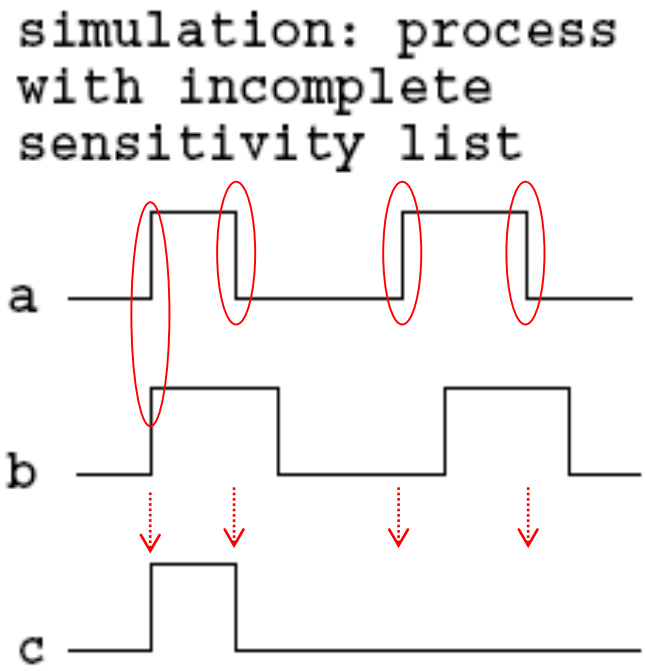
Wait for change on a or b, as in prev slide

- Bad process with incomplete sensitivity list:

```
PROCESS(a)
BEGIN
   c <= a AND b;
END PROCESS;
```

Trigger only when `a` changes

Not evaluated when `b` changes (simulation does not match synthesis !!!). superbad.

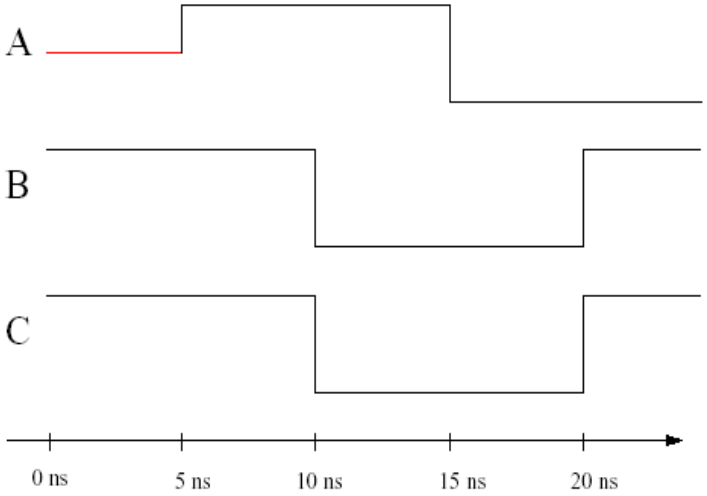


Example: Last Assignment Is Kept

```
ENTITY Pitfall1 IS
END Pitfall1;
ARCHITECTURE behav OF
     Pitfall1 IS
     SIGNAL A, B, C :
     std_logic;
BEGIN
     A <= '1' AFTER 5
     ns,
       '0' AFTER 15 ns,
       '1' AFTER 25 ns;
     B <= '1' AFTER 0
     ns,
       '0' AFTER 10 ns,
       '1' AFTER 20 ns;
     PROCESS (A, B)
     BEGIN -- process
       C <= A;
         ...
       C <= B;
     END PROCESS;
END behav;
```

Input wave generation, (sim only)

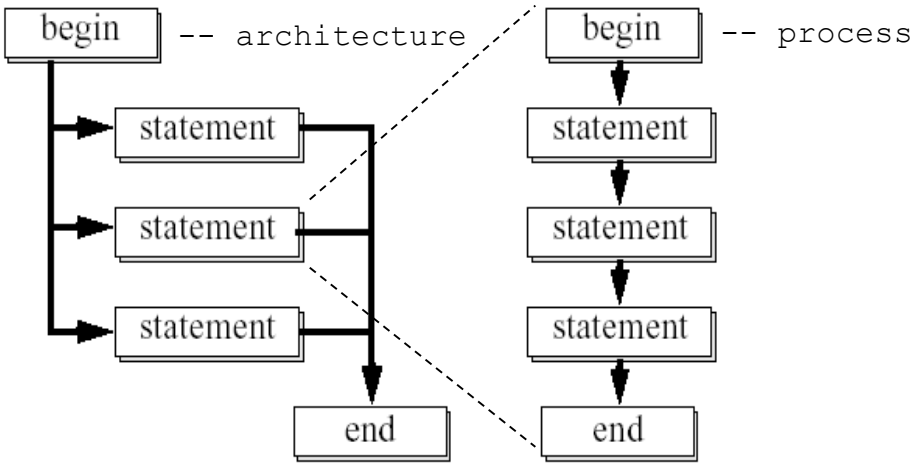
OK



Only the last assignment, `C <= B;` ,is kept.

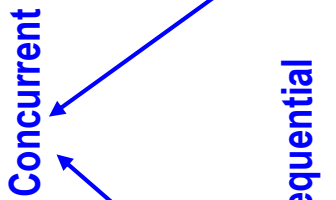
However, this is also useful. In a complex process, designer can assign a default value at first and then overrride it later in some branch.

Concurrent vs. Sequential VHDL



Modeling style	Concurrent	Sequential
Location	Inside architecture	Inside process or function
Example statements	process, component instance, concurrent signal assignment	if, for, switch-case, signal assignment, variable assignment

Concurrent vs. Sequential VHDL: Example




```
ARCHITECTURE rtl OF rotate_left IS
  SIGNAL rot_r : STD_LOGIC_VECTOR(7 DOWNTO 0);
  BEGIN
    shift : PROCESS(rst, clk)
    BEGIN
     IF (rst = '1') THEN
          rot_r <= (others =>'0'); -- reset the register

     ELSIF (clk = '1' AND clk'EVENT) THEN
          IF (load_en_in = '1') THEN
               rot_r <= data_in; -- store new value
          ELSE
               rot_r (7 DOWNTO 1)<= rot_r(6 DOWNTO 0);
               rot_r (0)         <= rot_r(7);
          END IF;
     END IF;
    END PROCESS;
   q_out <= rot_r; -- connect DFF's output to output port
END concurrent_and_sequential;
```

Comando: IF then else .

```
if CONDITION then
   -- sequential statements
end if;


if CONDITION then
   -- sequential statements
else
   -- sequential statements
end if;


if CONDITION then
   -- sequential statements
elsif CONDITION then
   -- sequential statements
. . .
else
   -- sequential statements
end if;
```

A condição é uma expressão booleana

Opcional:

-elsif

-else

IF Statement: Examplo


```
entity IF_STATEMENT is
    port (A, B, C, X : in   bit_vector (3 downto 0);
          Z          : out bit_vector (3 downto 0);
end IF_STATEMENT;
```

```
architecture EXAMPLE1 of IF_STATEMENT is
begin
  process (A, B, C, X)
  begin
    Z <= A;
    if  (X = "1111")  then
      Z <= B;
    elsif  (X > "1000")  then
      Z <= C;
    end if;
  end process;
end EXAMPLE1;
```

```
architecture EXAMPLE2 of IF_STATEMENT is
begin
  process (A, B, C, X)
  begin

    if  (X = "1111")  then
      Z <= B;
    elsif  (X > "1000")  then
      Z <= C;
    else
      Z <= a;
    end if;
  end process;
end EXAMPLE2;
```

Exemplo de “if”

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

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

Comando: CASE is WHEN .


```
case  EXPRESSION  is

   when  VALUE_1  =>
     -- sequential statements

   when  VALUE_2 | VALUE_3   =>
     -- sequential statements

   when  VALUE_4 to VALUE_N  =>
     -- sequential statements

   when   others =>
     -- sequential statements

end   case ;
```

- Opções não podem ser coincidentes.

- Todas as opções devem ser cobertas:
  - valores simples
  - intervalo de valores
  - seleção de valores por ("|" que significa "or")
  - uso obrigatorio de "when others" para cobrir a(s) ultima(s) opções.

```
entity CASE_STATEMENT is
   port (A, B, C, X : in   integer range 0 to 15;
          Z             : out integer range 0 to 15;
end CASE_STATEMENT;

architecture EXAMPLE of CASE_STATEMENT is
begin
   process (A, B, C, X)
   begin
      case  X  is
        when  0  =>
          Z <= A;
        when  7  |  9  =>
          Z <= B;
        when  1 to 5  =>
          Z <= C;
        when others =>
          Z <= 0;
      end case;
   end process;
end EXAMPLE;
```

```
entity RANGE_2 is
port (A, B, C, X : in    bit_vector(3 downto 0);
        Z            : out  bit_vector(3 downto 0);
end RANGE_2;

architecture EXAMPLE of RANGE_2 is
begin
  process (A, B, C, X)
  begin
    case X is
      when "0000" =>
        Z <= A;
      when "0111" | "1001" =>
        Z <= B;
      when "0001" to "0101" =>        -- wrong
        Z <= C;
      when others =>
        Z <= 0;
    end case;
  end process;
end EXAMPLE;
```

A sequencia de valores é indefinida para arrays.

```
entity CONDITIONAL_ASSIGNMENT is
   port (A, B, C, X : in   bit_vector (3 downto 0);
          Z_CONC : out bit_vector (3 downto 0);
          Z_SEQ    : out bit_vector (3 downto 0));
end CONDITIONAL_ASSIGNMENT;

architecture EXAMPLE of CONDITIONAL_ASSIGNMENT is
begin
   -- Concurrent version of conditional signal assignment
   Z_CONC <= B when X = "1111" else
                  C when X > "1000" else
                  A;

   -- Equivalent sequential statements
   process (A, B, C, X)
   begin
      if  (X = "1111")  then
        Z_SEQ <= B
      elsif  (X > "1000")  then
        Z_SEQ <= C;
     else
        Z_SEQ <= A;
      end if;
   end process;
end EXAMPLE;
```

Loop - FOR

- 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;
```

Comando: FOR Loops


```
entity FOR_LOOP is
   port (A : in    integer range 0 to 3;
          Z  : out bit_vector (3 downto 0));
end FOR_LOOP;

architecture EXAMPLE of FOR_LOOP is
begin
   process (A)
   begin
      Z <= "0000";
      for  I  in  0 to 3  loop
        if (A = I) then
           Z(I) <= `1`;
        end if;
      end loop;
   end process;
end EXAMPLE;
```

Se o LOOP é para ser sintetizado, o intervalo do loop não pode depender do valor de um sinal ou variavel, ou seja, deve ser totalmente estático o intervalo.

Loop Sintaxe


```
[LOOP_LABEL :]
for IDENTIFIER in DISCRETE_RANGE loop
   -- sequential statements
end loop [LOOP_LABEL] ;
```

```
[LOOP_LABEL :]
while CONDITION loop
   -- sequential statements
end loop [LOOP_LABEL] ;
```

```
entity CONV_INT is
    port (VECTOR: in   bit_vector(7 downto 0);
          RESULT:  out integer);
end CONV_INT;
```

```
architecture A of CONV_INT is
begin
  process(VECTOR)
    variable TMP: integer;

  begin
    TMP := 0;

    for I in 7 downto 0 loop
      if (VECTOR(I)='1') then
        TMP := TMP + 2**I;
      end if;
    end loop;

    RESULT <= TMP;
  end process;
end A;
```

```
architecture B of CONV_INT is
begin
  process(VECTOR)
    variable TMP: integer;

  begin
    TMP := 0;

    for I in VECTOR'range loop
      if (VECTOR(I)='1') then
        TMP := TMP + 2**I;
      end if;
    end loop;

    RESULT <= TMP;
  end process;
end B;
```

```
architecture C of CONV_INT is
begin
  process(VECTOR)
    variable TMP: integer;
    variable I      : integer;
  begin
    TMP := 0;
    I := VECTOR'high;
    while (I >= VECTOR'low) loop
      if (VECTOR(I)='1') then
        TMP := TMP + 2**I;
      end if;
      I := I - 1;
    end loop;
    RESULT <= TMP;
  end process;
end C;
```

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;
```

Comando: WAIT

O comando wait' para a execução do process

- **O process é continuado quando a instrução é completada.**

- **wait para um especifico tempo**
- **wait por um evento do sinal**
- **wait por uma condição verdadeira (necessita de um evento do sinal)**
- **indefinido (process não é mais ativado)**

WAIT Statement: Examplos

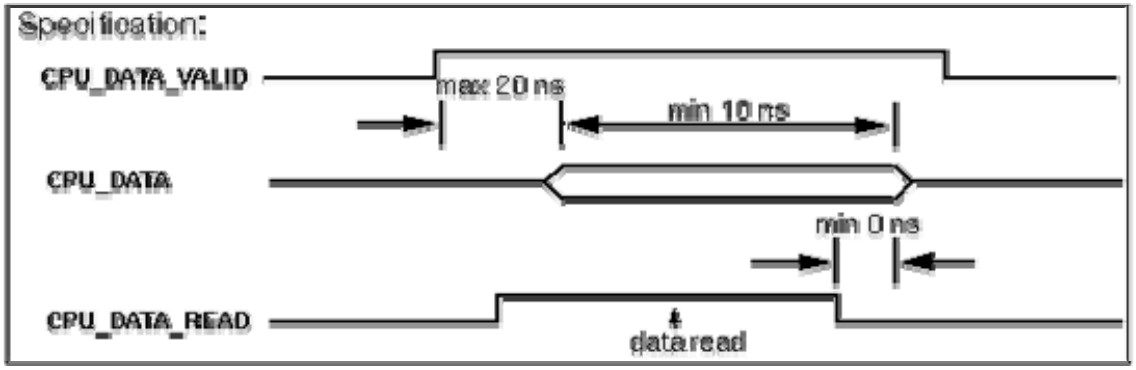

```
entity FF is
    port (D, CLK : in   bit;
          Q          : out bit);
end FF;
```

```
architecture BEH_1 of FF is
begin
  process
  begin
    wait on CLK;
    if (CLK = '1') then
      Q <= D;
    end if;
  end process;
end BEH_1;
```

```
architecture BEH_2 of FF is
begin
  process
  begin
    wait until CLK=`1`;

    Q <= D;

  end process;
end BEH_2;
```

WAIT Statement: Examplos


```
STIMULUS: process
begin
    SEL     <= `0`;
    BUS_B <= "0000";
    BUS_A <= "1111";
    wait for 10 ns;

    SEL <= `1`;
    wait for 10 ns;

    SEL <= `0`;
    wait for 10 ns;

    wait;
end process STIMULUS;
```



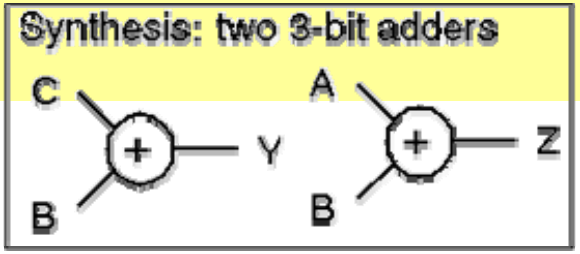

```
READ_CPU : process
begin
    wait until CPU_DATA_VALID = `1`;
    CPU_DATA_READ <= `1`;
    wait for 20 ns;
    LOCAL_BUFFER <= CPU_DATA;
    wait for 10 ns;
    CPU_DATA_READ <= `0`;
end process READ_CPU;
```

Uso de Variaveis em VHDL

Variaveis

- **Variaveis são usadas apenas em processes**
  - **São declaradas antes do begin do process**
  - **Conhecidas apenas localmente no process onde foram declaradas**
- **VHDL 93: variaveis globais**
  - Não sintetizavel
- **Assinalamento global**
  - **signal to variable**
  - **variable to signal**
  - **types have to match**

```
architecture RTL of XYZ is
    signal A, B, C : integer range 0 to 7;
    signal Y, Z :    integer range 0 to 15;
begin
    process (A, B, C)
      variable M, N : integer range 0 to 7;
    begin
      M := A;
      N := B;
      Z <= M + N;
      M := C;
      Y <= M + N;
    end process;
end RTL;
```



Variáveis

- Utilizada em processos, sem temporização. Atribuição imediata.
- Sintaxe:

```
variable var_id : tipo [restrição][:=expressão];
```

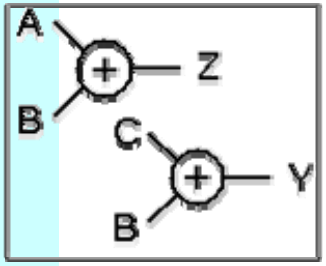
- Exemplos

```
variable  índice : integer range 1 to 50 := 50;
variable  ciclo : time range 10 ns to 50 ns := 10ns;
variable  memória : bit_vector (0 to 7)
variable  x, y : integer;
```

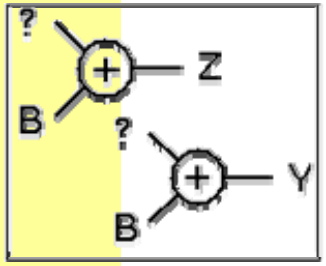
Variaveis vs. Sinais

- **Valores de sinais são assinalados depois da execução do process.**
- **Apenas o ultimo assinalamento é levado em consideração**
- **M <= A;**
  é sobre escrito por M <= C;
- **A segunda entrada do somador é conectado a C.**

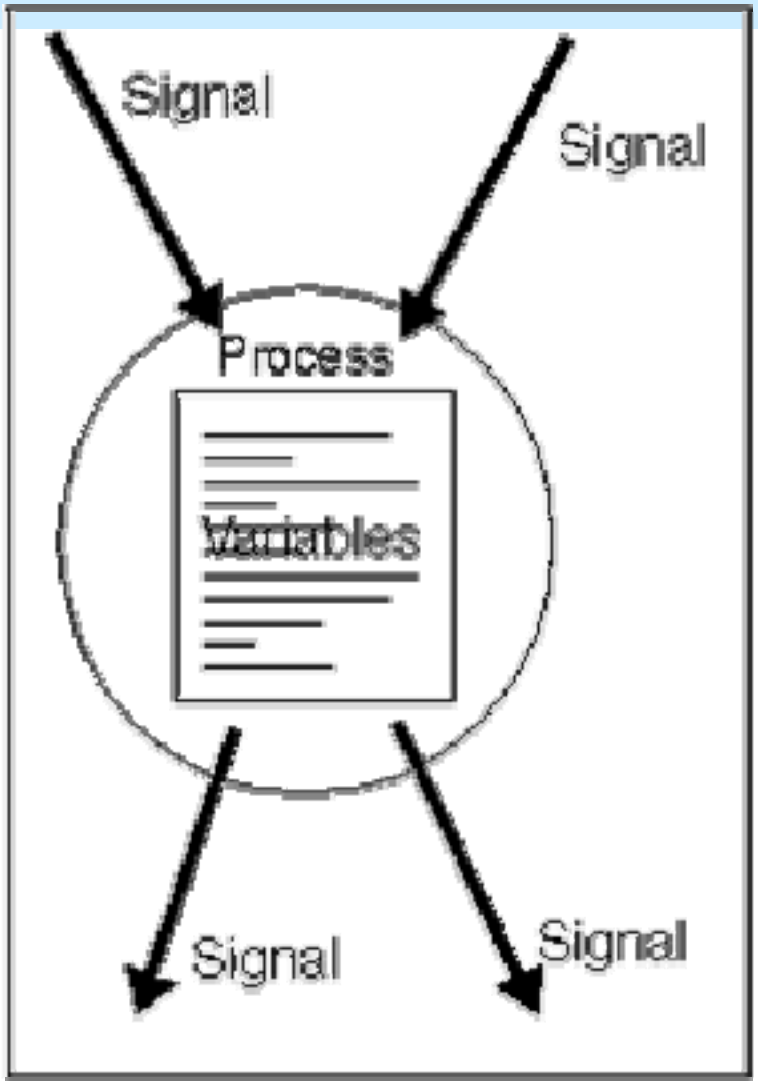
```
signal A, B, C, Y, Z : integer;

begin
    process (A, B, C)
        variable M, N : integer;
    begin
        M  :=  A;
        N  :=  B;
        Z <= M + N;
        M  :=  C;
        Y <= M + N;
    end process;
```



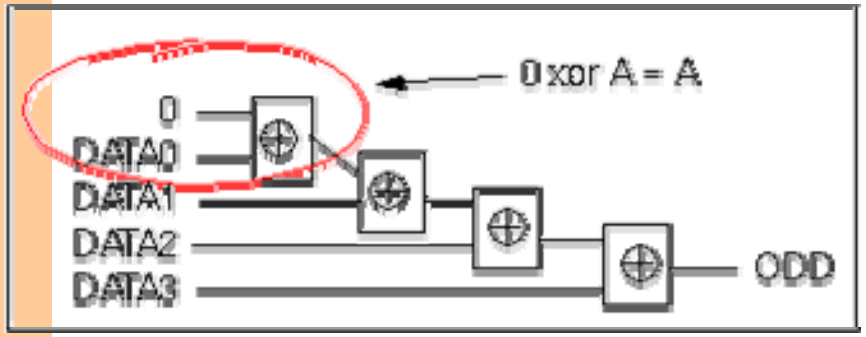

```
signal A, B, C, Y, Z : integer;
signal M, N          : integer;
begin
    process (A, B, C, M, N)

    begin
        M  <=  A;
        N  <=  B;
        Z <= M + N;
        M  <=  C;
        Y <= M + N;
    end process;
```



Uso de Variaveis



```
entity PARITY is
     port (DATA: in   bit_vector (3 downto 0);
            ODD : out bit);
end PARITY;


architecture RTL of PARITY is
begin
    process (DATA)
      variable TMP : bit;
    begin
      TMP := `0`;

      for I in  DATA`low to DATA`high  loop
        TMP := TMP xor DATA(I);
      end loop;

      ODD <= TMP;
    end process;
end RTL;
```



Clock

```
-- 100 MHz
clock_100 <= not clock_100 after 5 ns;

-- 200 MHz
clock_200 <= not clock_200 after 2.5 ns;
```

Latch tipo D sensível a nível lógico 1

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

entity latch_d is
port (clk, D: in std_logic;
      Q: out std_logic);
end latch_d;

architecture behavioral of latch_d is
begin
  process(clk, D)
  begin
      if clk = '1' then
            Q <= D;
      end if;
  end process;
end architecture behavioral;
```

Flipflop tipo D disparado por borda de subida

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

entity ffd is
port (clk, D: in std_logic;
      Q: out std_logic);
end ffd;

architecture behavioral of ffd is
begin
  process(clk)
  begin
      if clk`event AND clk = '1' then
            Q <= D;
      end if;
  end process;
end architecture behavioral;
```

FF D disp. por borda de subida c/ RESET asínc.

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

entity ffd is
port (clk,rst, D: in std_logic;
      Q: out std_logic);
end ffd;

architecture behavioral of ffd is
begin
  process(clk)
  begin
      if rst='1' then
            Q <= '0';
      elsif clk`event AND clk = '1' then
            Q <= D;
      end if;
  end process;
end architecture behavioral;
```

REGISTRADOR (2/4)

Exemplo de registrador com largura de palavra parametrizável e com habilitação (sinal “ce”, do inglês *chip enable*):

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 para o módulo

Uso:

```
rx: regnbit generic map(8)
     port map(ck => ck, rst => rst,  ce => wen,
                  D => RD, Q => reg);
```

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 sob controle de um *clock*, serão sintetizados como saídas de *flip-flops*
- Sinais fora de processos ou em processos sem variável de sincronismo (*clock*) serão, em geral, sintetizados como lógica combinacional

Erros comuns ao se descrever um *process*!!

1. Duplo driver (fazer atribuições de um mesmo sinal em dois comandos *process* distintos)

Errado

```
p1: process(clock, reset)
    begin
        if reset='1' then
                x <= '0';
        elsif clock'event and clock='1' then
        if  hab='0' then
                x <= '1';
            end if;
end process;

p2: process(clock)
    begin
        if clock'event and clock='1' then
        if  ctr='1' then
                x <= '1';
            end if;
end process;
```

Certo

```
p1: process(clock, reset)
begin
    if reset='1' then
            x <= '0';
    elsif clock'event and clock='1' then
        if  hab='0'  or ctr='1' then
        x <= '1';
        end if;
end process;
```

2. Lista de sensitividade incompleta

3. Escrever código VHDL que cause inferência de "latches" **(ver lâmina 14)**

4. Realizar lógica junto com o teste de borda do sinal de clock

Errado: elsif clock'event and clock='1' and ce='1' then…

Certo: elsif clock'event and clock='1' then
if ce='1' then…

5. Não atentar para que sinais geram registradores **(ver lâmina 18)**

Importante: todo sinal controlado por um *event* implica criar um registrador

Problemas em Comandos `Process`

- O problema de inferência de "latches"em comandos `process`
  - Algumas descrições VHDL podem provocar a inferência de meios de armazenamento para garantir que um hardware se comporta da mesma forma que o modelo de simulação
  - Latches inferidos costumam causar problemas. EVITEM ELES!!
- Exemplo

Esta descrição VHDL escreve em B somente quando A vale "1010", e deve (pela semântica do process) manter o valor de B sempre que A for diferente deste valor. Assim, a síntese de hardware infere um latch para armazenar o valor de B.

```
entity gera_latch is
   Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
          B : out  STD_LOGIC_VECTOR (3 downto 0));
end gera_latch;

architecture Behavioral of gera_latch is
 signal int_B: STD_LOGIC_VECTOR (3 downto 0):="0000";
begin
B <= int_B;
gera_latch: process (A)
   begin
            if (A="1010") then
               int_B<= not A;
            end if;
   end process;
end Behavioral;
```