Agregados

- **Uma expressão indicando o valor de um tipo composto**
- **Cada elemento do tipo composto tem o seu valor definido**
- **OTHERS**
  - identifica todos os elementos não especificados
  - deve ser a última associação na lista de associações

```
-- valor 00010 agregado notacao posicional
-- posicao:
--      4   3   2   1   0
s2 <= ('0','0','0','1','0');              -- equivalente: s2 <= "00010";


-- valor 00011 agregado associacao por nomes
--
s3 <= (1=>'1', 0=>'1', OTHERS=>'0');     -- equivalente: s3 <= "00011";
```

Definição de novos tipos (visão preliminar)

- **A linguagem permite a criação de novos tipos**
- **Aplicações**:
  - facilitar a leitura do código: estados de uma máquina
  - definir novos tipos físicos: resistência, capacitância etc.
  - novos tipos compostos: definição de memórias
- **Declaração**: palavra reservada TYPE
- **Exemplo**:

```
TYPE estado IS (parado, inicio, caso_1, caso_2, caso_3);
SIGNAL abc : estado := parado;
```

- é definido um novo tipo estado

```
TYPE estado IS (parado, inicio, caso_1, caso_2, caso_3);
```

- declaração de um sinal do tipo estado

```
SIGNAL abc : estado := parado;
```

- valores possíveis para o sinal abc: parado, inicio, caso_1, caso_2, caso_3

Operadores

- **Divididos em classes**:
  - as classes definem a precedência dos operadores
  - operadores de uma mesma classe: igual precedência
- **Maior precedência**: classe diversos
- **Menor precedência**: classe lógicos
- **Operador not**: operador lógico; está na classe diversos devido à precedência

classe	operadores
lógicos	and or nand nor xor xnor
relacionais	= /= < <= > >=
deslocamento	sll srl sla sra rol ror
adição	+ – &
sinal	+ –
multiplicação	* / mod rem
diversos	** abs not

Operadores lógicos

- Operandos: tipos bit e boolean
  vetores unidimensionais compostos de elementos bit e boolean
  (exemplo: bit_vector)
- Os vetores devem ter o mesmo tamanho
- Operação executada entre elementos de mesma posição

operadores	operando L	operando R	retorna
not and or	bit	bit	bit
nand nor xor xnor	boolean	boolean	boolean
Nota: o operador not pertence à classe diversos			

- Tipo bit: 0,1 tipo boolean: 0 = false 1 = true

L	R	not L	L and R	L nand R	L or R	L nor R	L xor R	L xnor R
1	1	0	1	0	1	0	0	1
1	0	0	0	1	1	0	1	0
0	1	1	0	1	1	0	1	0
0	0	1	0	1	0	1	0	1

Operadores relacionais

- Igualdade e desigualdade: qualquer tipo
  - escalares x = y verdadeiro: x mesmo valor de y
  - compostos x = y verdadeiro: são iguais cada elemento de mesma posição de x e y

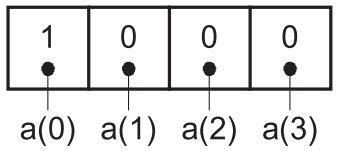
operadores	operando L	operando R	retorna
= /=	qualquer tipo	mesmo tipo de L	boolean

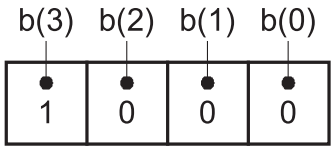
- Ordenação: tipos escalares (bit, boolean, character, integer, real, time)

operadores	operando L	operando R	retorna
> < >= <=	qualquer tipo escalar	mesmo tipo de L	boolean

- Exemplo de valores tipo bit_vector iguais:

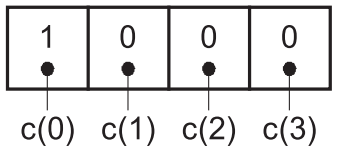
CONSTANT a: BIT_VECTOR(0 TO 3) := "1000"





CONSTANT b: BIT_VECTOR(3 DOWNTO 0) := "1000"

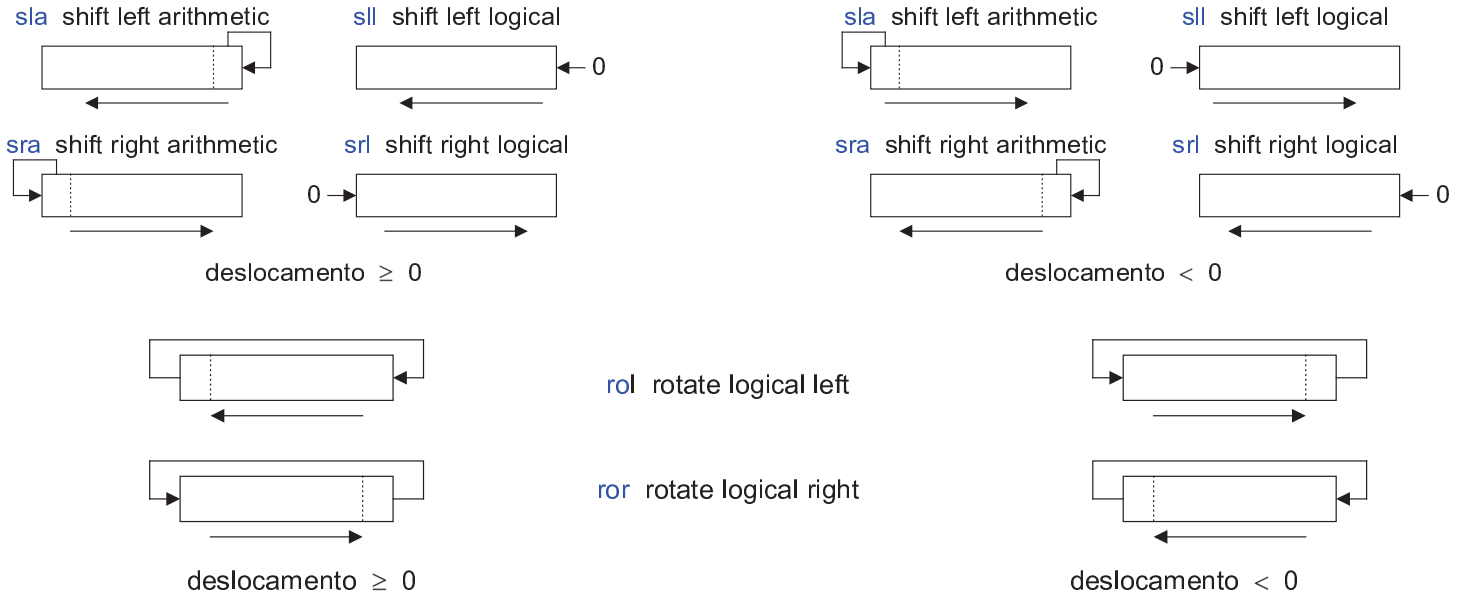
CONSTANT c: BIT_VECTOR(0 TO 3) := "1000"



Operadores de deslocamento e rotação

- Nota: não suportado pela versão VHDL-1987

operadores	operando L	operando R	retorna
sll srl sla sra rol ror	vetor unidimensional com elementos bit ou boolean	integer	o mesmo tipo de L



Operadores de adição

- Adição e subtração: tipo numérico

operadores	operando L	operando R	retorna
+ -	tipo numérico	o mesmo tipo de L	mesmo tipo

- Concatenação: vetor unidimensional e elementos (mesmo tipo)

operadores	operando L	operando R	retorna
&	vetor	elemento	vetor mesmo tipo
	elemento	vetor	vetor mesmo tipo
	elemento	elemento	vetor

- **Exemplos**:

```
a <= b & c;   -- ”a” bit_vector 8 elementos,”b”,”c” bit_vetor 4 elementos
x <= y & ’1’; -- ”x” bit_vector 5 elementos,    ”y” bit_vetor 4 elementos
```

Operadores aritméticos – identidade, negação

- Possuem o mesmo significado dos operadores matemáticos equivalentes
- Operandos: qualquer tipo numérico

operadores	operando R	retorna
+ -	qualquer tipo numérico	mesmo tipo

Operadores diversos – absoluto, exponenciação

- Possuem o mesmo significado dos operadores matemáticos equivalentes

operadores	operando L	operando R	retorna
abs	qualquer tipo numérico	-	mesmo tipo
**	qualquer tipo integer	integer	mesmo tipo de L
	qualquer tipo real	integer	mesmo tipo de L

Operadores de multiplicação

- Multiplicação e divisão: operandos tipo inteiro e real
- mod e rem: operandos tipo inteiro

operadores	operando L	operando R	retorna
* /	qualquer tipo integer	mesmo tipo	mesmo tipo
	qualquer tipo real	mesmo tipo	mesmo tipo
mod rem	qualquer tipo integer	mesmo tipo	mesmo tipo

Operadores de multiplicação – tipos físicos

- Multiplicação: operandos tipo (físico – real) e (físico – inteiro)
- Divisão: operando L tipo físico, operando R (inteiro, real, físico)

operadores	operando L	operando R	retorna
*	qualquer tipo physical	integer	mesmo tipo de L
	qualquer tipo physical	real	mesmo tipo de L
	integer	qualquer tipo physical	mesmo tipo de R
	real	qualquer tipo physical	mesmo tipo de R
/	qualquer tipo physical	integer	mesmo tipo de L
	qualquer tipo physical	real	mesmo tipo de L
	qualquer tipo physical	mesmo tipo	integer

Exemplo – Atribuição de valores em sinais, tipos INTEGER e REAL

- Operação: valor 11 atribuído a todas as portas de saída

```
 9 ARCHITECTURE teste OF int_real IS
10   CONSTANT i1 : INTEGER  :=  11;              -- valor 11, base 10
11   CONSTANT i2 : INTEGER  :=  10#11#;          -- valor 11, base 10
12   CONSTANT i3 : INTEGER  :=  2#01011#;        -- valor 11, base 2
13   CONSTANT i4 : INTEGER  :=  2#01_01_1#;      -- valor 11, base 2
14   CONSTANT i5 : INTEGER  :=  5#21#;           -- valor 11, base 5
15   CONSTANT i6 : NATURAL  :=  8#13#;           -- valor 11, base 8
16   CONSTANT i7 : POSITIVE :=  16#B#;           -- valor 11, base 16
17
18   CONSTANT r1 : REAL     :=  11.0;            -- valor 11, base 10
19   CONSTANT r2 : REAL     :=  1.1E01;          -- valor 11, base 10 formato nn.nExx
20   CONSTANT r3 : REAL     :=  2#01011.0#;      -- valor 11, base 2
21   CONSTANT r4 : REAL     :=  8#1.3#E01;       -- valor 11, base 8  formato nn.nExx
22   CONSTANT r5 : REAL     :=  16#B.0#;         -- valor 11, base 16
23 BEGIN
24   ci1 <= i1; ci2 <= i2; ci3 <= i3; ci4 <= i4; ci5 <= i5; ci6 <= i6; ci7 <= i7;
25   cr1 <= r1; cr2 <= r2; cr3 <= r3; cr4 <= r5;
26 END teste;
```

Exemplo – Atribuição de valores em sinais, tipos BIT_VECTOR

- Operação: valor 1011 atribuído a todas as portas de saída
- Diferentes bases de representação
  - tipos integer, real, formato: 16#B# 16#B.0# (vide exemplo anterior)
  - tipos bit_vector, formato: X"B"
- Linha 12: caractere em 01_0_11 melhora a leitura do valor
- Linhas 14 e 15: valor definido para uma parte do vetor pal. res. DOWNTO

```
 5 ARCHITECTURE teste OF std_a IS
 6   CONSTANT c1   : BIT_VECTOR(4 DOWNTO 0) := "01011";  -- constante
 7   CONSTANT zero : BIT := '0';
 8   CONSTANT um   : BIT := '1';
 9 BEGIN
10   s1 <= c1;                -- valor atraves de constante
11   s2 <= "01011";           -- valor (01011) direto - base binaria
12   s3 <= B"01_0_11";        -- valor (01011) direto - base binaria com separadores
13   s4 <= '0' & X"B";        -- bit (0) concatenado com valor hexadecimal (1011)
14   s5(4 DOWNTO 3) <= "01";           -- valor (01), parte do vetor
15   s5(2 DOWNTO 0) <= zero & um & um;  -- valor (010), parte com concatenacao
16 END teste;
```

Exemplo – Atribuição de valores em sinais, tipos BIT_VECTOR – agregados

- s2 s4 → atribuição de valor: notação posicional
- s3 s5 → atribuição de valor: associação de nomes

```
 1 ENTITY std_a1 IS
 2  PORT(s2, s3, s4, s5   : OUT BIT_VECTOR(4 DOWNTO 0));
 3 END std_a1;
 4
 5 ARCHITECTURE teste OF std_a1 IS
 6   CONSTANT zero : BIT := '0';
 7   CONSTANT um   : BIT := '1';
 8 BEGIN
 9   s2 <= ('0','0','0','1','0');            --  00010, agregado notacao posicional
10   s3 <= (1=>'1', 0=>'1', OTHERS=>'0');     --  00011, agregado associacao por nomes
11   s4 <= (zero, '0', um OR '0', '0', '0');        -- 00100, agregado com operacoes
12   s5 <= (4 DOWNTO 3 =>'0', 1=>'0', OTHERS=>'1'); -- 00101, agregado faixa discreta
13 END teste;
```

Exemplo – Atribuição de valores – expressões com operadores lógicos

- Linhas 8 a 14: comentário no código
- Linha 15: x nand y equivale a not (a and b) → ordem das operações importa

```
 1 ENTITY std_xa1 IS
 2   PORT( a, b, c, d          : IN  BIT;
 3         x1, x2, x3, x4, x5 : OUT BIT);
 4 END std_xa1;
 5
 6 ARCHITECTURE exemplo OF std_xa1 IS
 7 BEGIN
 8    x1 <= a OR NOT b;              -- Certo:  operador NOT tem precedencia
 -                                   --         mais elevada
 9    x2 <= a AND b AND c;           -- Certo:  operadores iguais
10 -- x3 <= a AND b  OR c;           -- Errado: expressao ambigua x3=(a.b)+c
 -                                   --                         ou x3=a.(b+c) ?
11    x3 <=(a AND b) OR c;           -- Certo:  empregando parentesis
12 -- x4 <= a AND b  OR  c AND d;    -- Errado: expressao ambigua, operadores
 -                                   --         com mesma precedecia
13    x4 <=(a AND b) OR (c AND d);   -- Certo:  x4 = a.b + c.d
14 -- x5 <= a NAND b  NAND c;        -- Errado: operadores com negacao
 -                                   --         necessitam parentesis
15    x5 <=(a NAND b) NAND c;        -- Certo:  operador com negacao entre
 -                                   --         parentesis
16 END exemplo;
```

Exemplo – Operadores de classe adição

- Linhas 10 e 11: operação de concatenação de dois vetores (tipos bit_vector)
- Linha 12: soma de dois tipos inteiros

```
 1 ENTITY std_xc IS
 2   PORT (bv_a,  bv_b  : IN  BIT_VECTOR(1 DOWNTO 0);
 3         int_a, int_b : IN  INTEGER RANGE -32 TO 31;
 4         bv_c,  bc_d  : OUT BIT_VECTOR(3 DOWNTO 0);
 5         int_c        : OUT INTEGER RANGE -64 TO 63);
 6 END std_xc;
 7
 8 ARCHITECTURE teste OF std_xc IS
 9 BEGIN
10   bv_c  <= bv_a & bv_b;
11   bc_d  <= bv_a & '1' & '0';
12   int_c <= -int_a +int_b;
13 END teste;
```