

# Introdução ao VHDL

Circuitos Lógicos

DCC-IM/UFRJ

Prof. Gabriel P. Silva

Original por  
Ayman Wahba

# VHDL

- **É uma linguagem de descrição de “hardware”, ou seja, uma forma estruturada para a descrição de circuitos digitais.**
- **Essa linguagem permite que o circuito eletrônico seja descrito com sentenças, tais como em uma linguagem de programação, possibilitando que seja simulado e sintetizado, isto é, transformado em portas lógicas.**
- **Very High Speed ASIC Description Language**

# História do VHDL

- \* 1981: Iniciada pelo Departamento de Defesa dos EUA para resolver a crise do ciclo de vida dos projetos eletrônicos.**
- \* 1983-85: Desenvolvimento da linguagem básica pela empresa Intermetrics, IBM e Texas Instruments.**
- \* 1986: Todos os direitos transferidos para o IEEE.**
- \* 1987: Publicação do padrão IEEE – VHDL 87.**
- \* 1994: Padrão revisado (VHDL 93)**

# Porquê VHDL?

**Aumenta dramaticamente a sua produtividade.**

**É uma forma muito mais rápida para projetar circuitos digitais.**

# Porquê VHDL?

**Permite que o mesmo código seja usado com diversas tecnologias.**

**Isso garante portabilidade e longevidade para seu projeto.**

# Porquê VHDL?

**É possível testar o seu código em diversos níveis, garantindo maior confiabilidade nos resultados.**

# Como o VHDL é Utilizado?

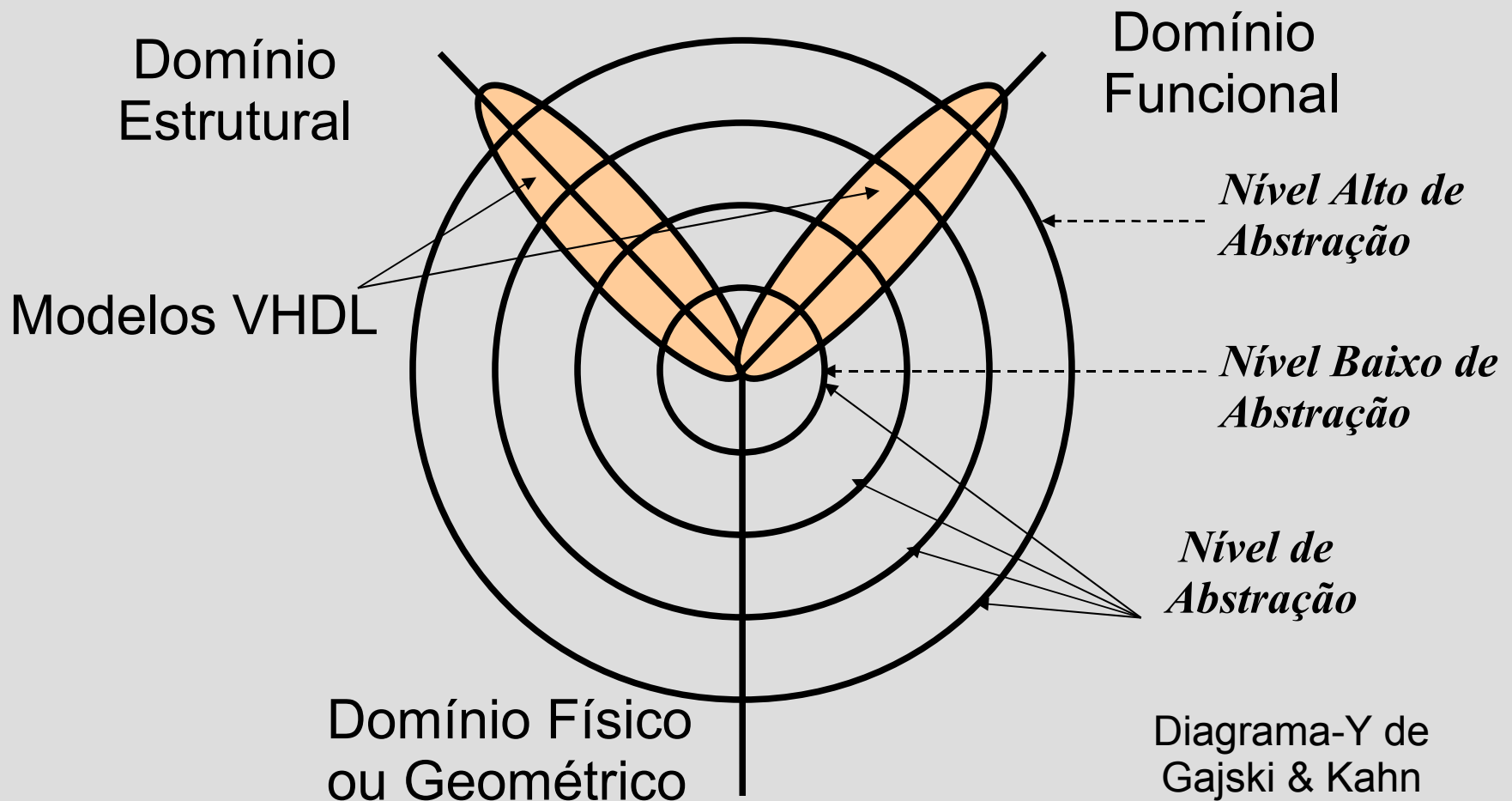
- \* Para a especificação do projeto;**
- \* Para a captura do projeto;**
- \* Para a simulação do projeto;**
- \* Para documentação do projeto;**
- \* Como uma alternativa ao esquemático;**
- \* Como uma alternativa às linguagens proprietárias.**

# VHDL

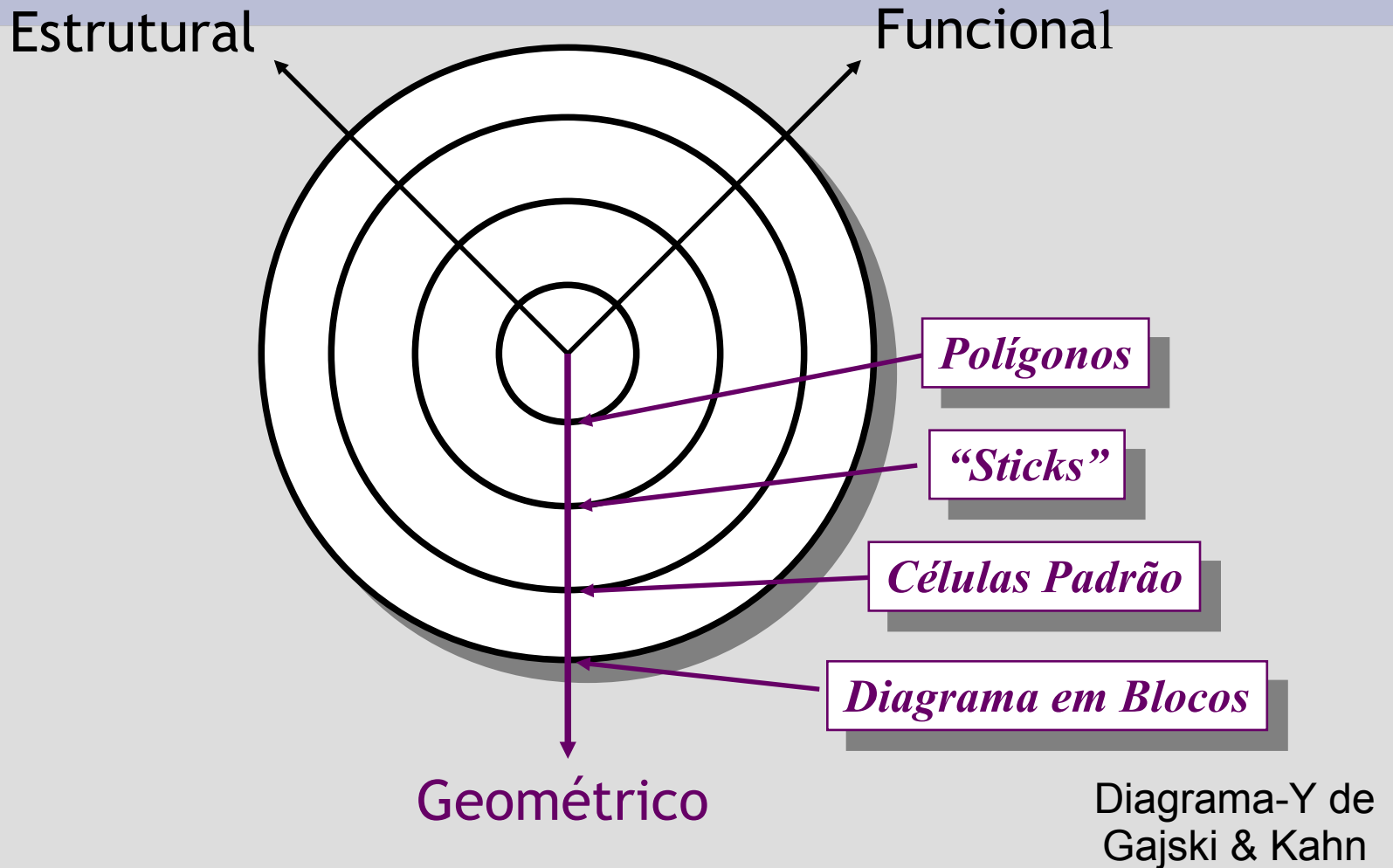
- **Em realidade o VHDL permite que um circuito seja descrito tanto no eixo estrutural como no eixo comportamental (funcional).**
- **Quanto maior for o nível de abstração, mais dependente das ferramentas de síntese fica o projeto.**
- **Em contrapartida, mais flexível é a sua aplicação e possibilita o uso de diversas tecnologias com que pode ser implementado.**



# Domínios de Descrição de um Circuito



# Domínios e Níveis de Modelagem



# Domínios e Níveis de Modelagem

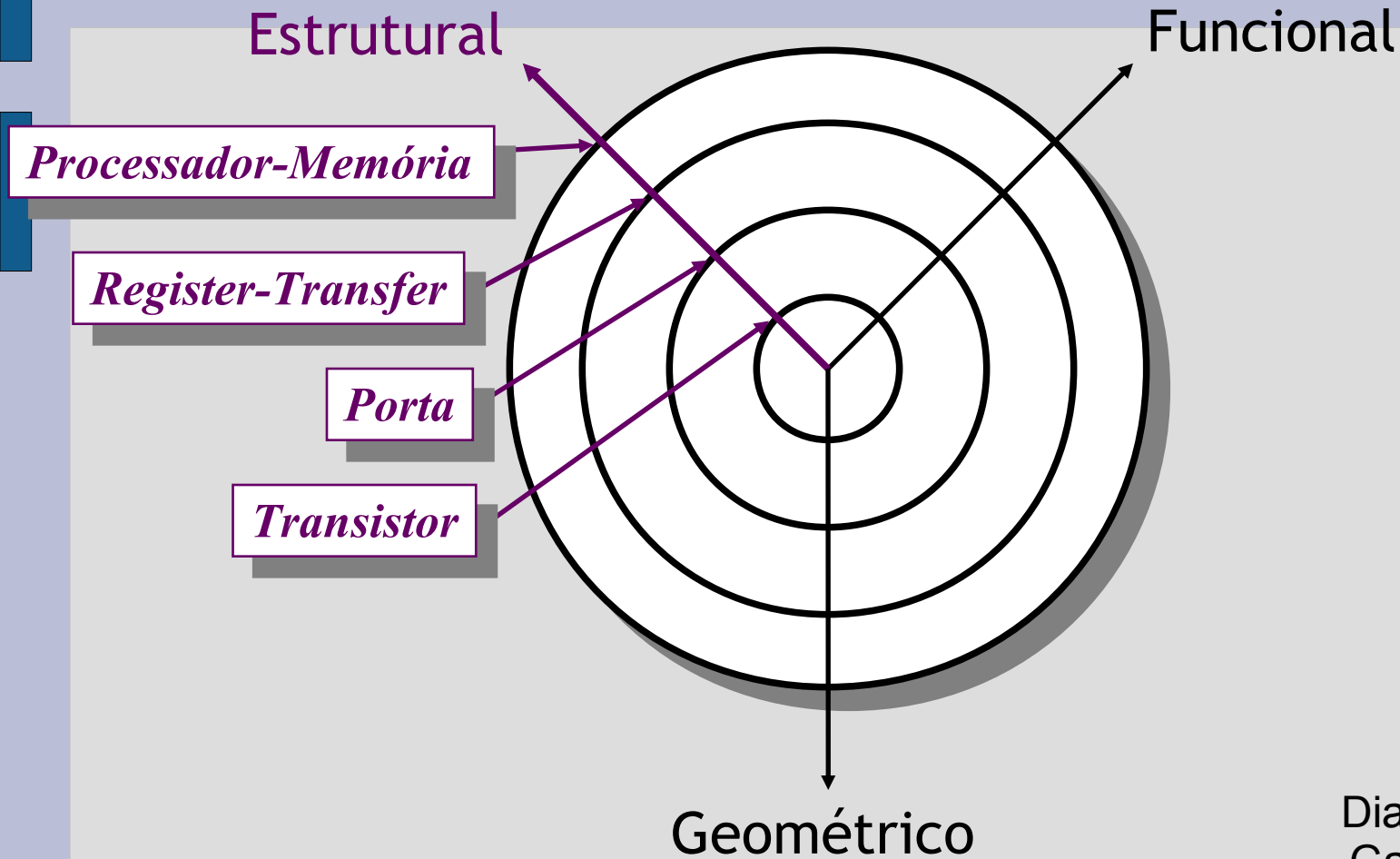
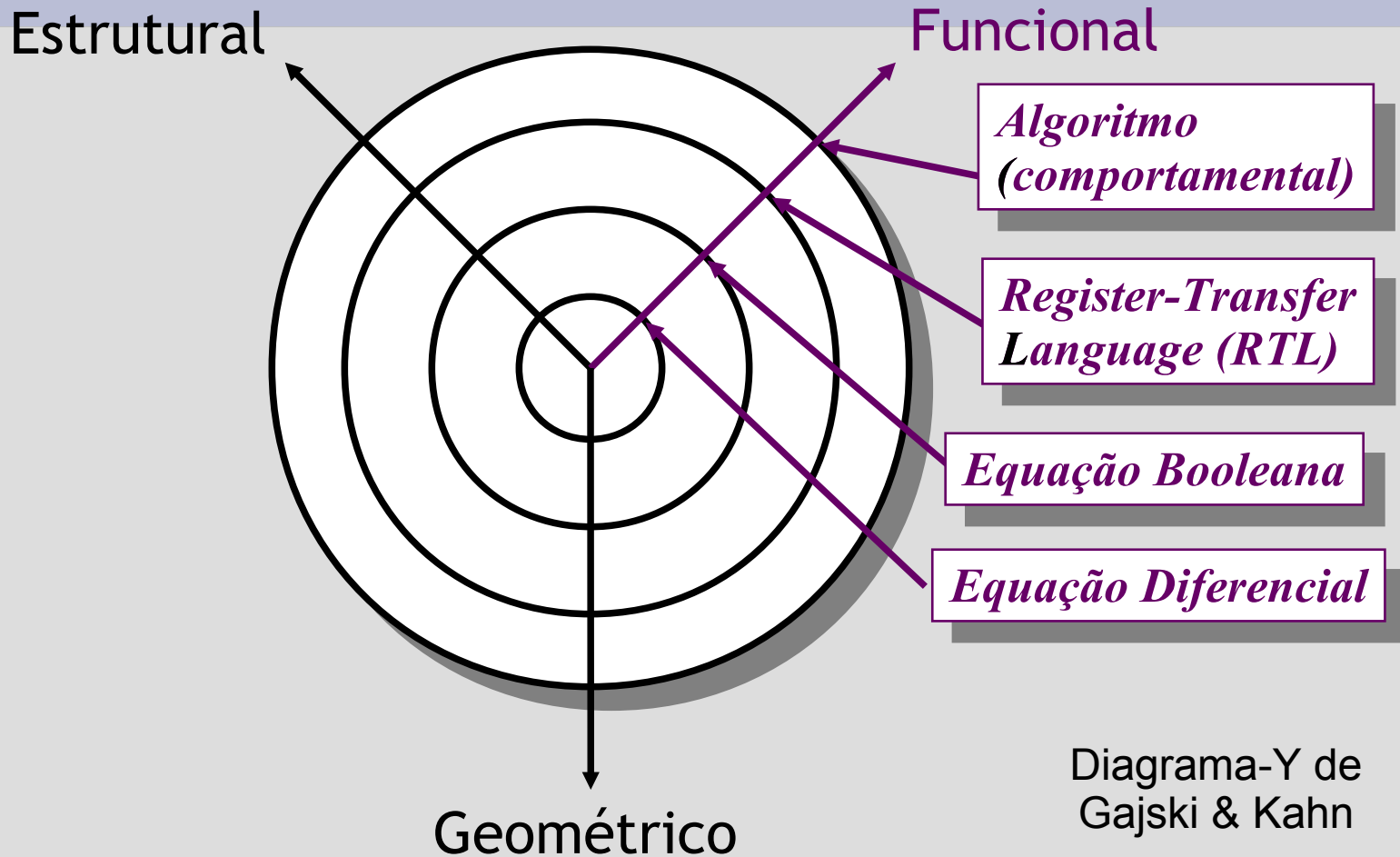
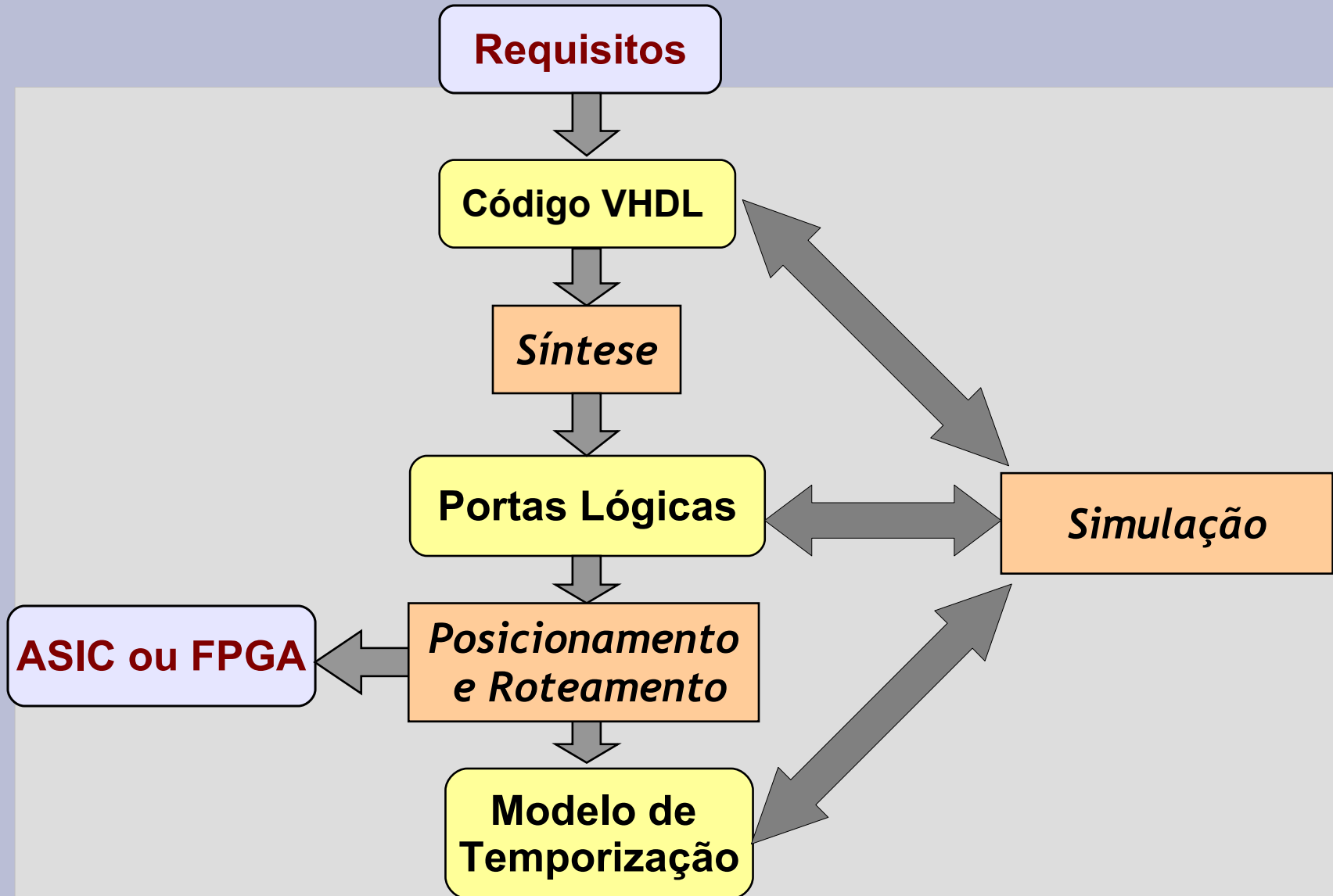


Diagrama-Y de Gajski & Kahn

# Domínios e Níveis de Modelagem



# Metodologia Básica de Projeto



# Exemplo de Processo de Projeto

## \* Problema:

**Projetar um meio somador de um bit com vai\_um e habilita.**

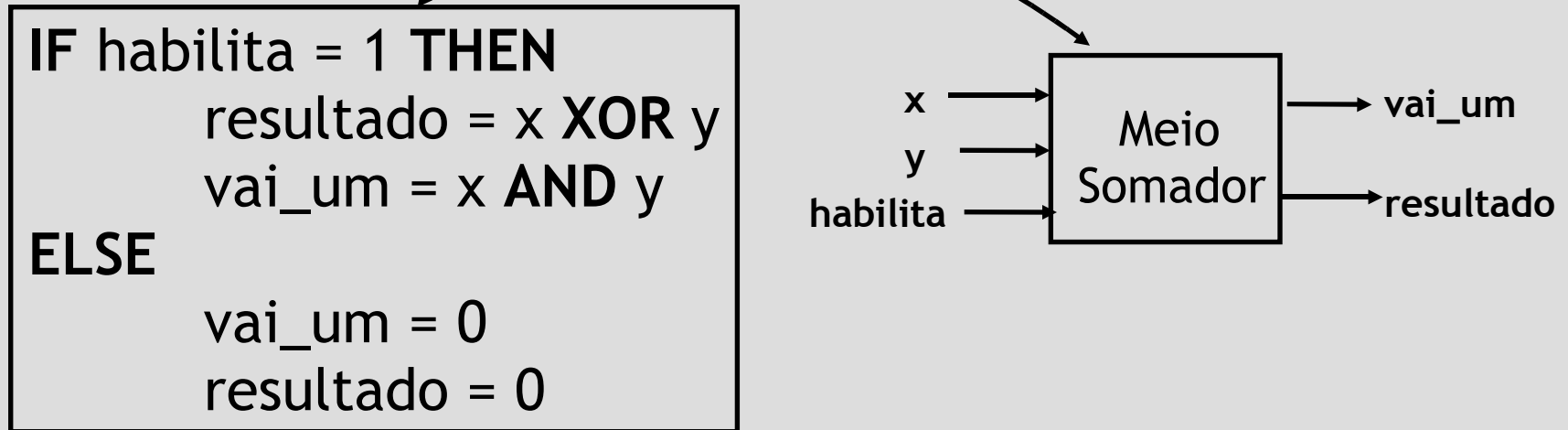
## \* Especificações:

- **Passa o resultado apenas se habilita for igual a '1'.**
- **Resultado é zero se habilita for igual a '0'.**
- **Resultado recebe  $x + y$**
- **Vai\_um recebe o vai\_um, se houver, de  $x + y$**



# Projeto Comportamental

\* Iniciando com um algoritmo, uma descrição de alto nível do somador é criada:

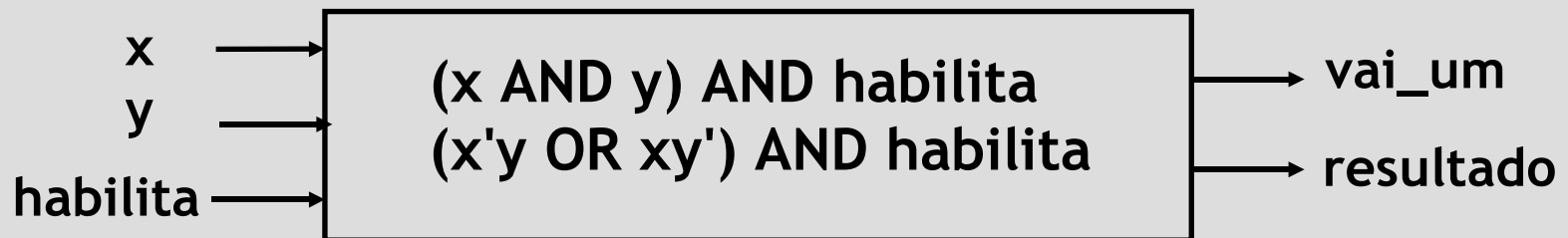


\* O modelo pode ser agora simulado nesse nível de descrição para verificar o correto entendimento do problema.

# Projeto Fluxo de Dados

- \* Com a descrição de alto nível confirmada, equações lógicas descrevendo o fluxo de dados são então criadas.

$\text{vai\_um} = (x \text{ AND } y) \text{ AND } \text{habilita}$   
 $\text{resultado} = (x'y \text{ OR } xy') \text{ AND } \text{habilita}$

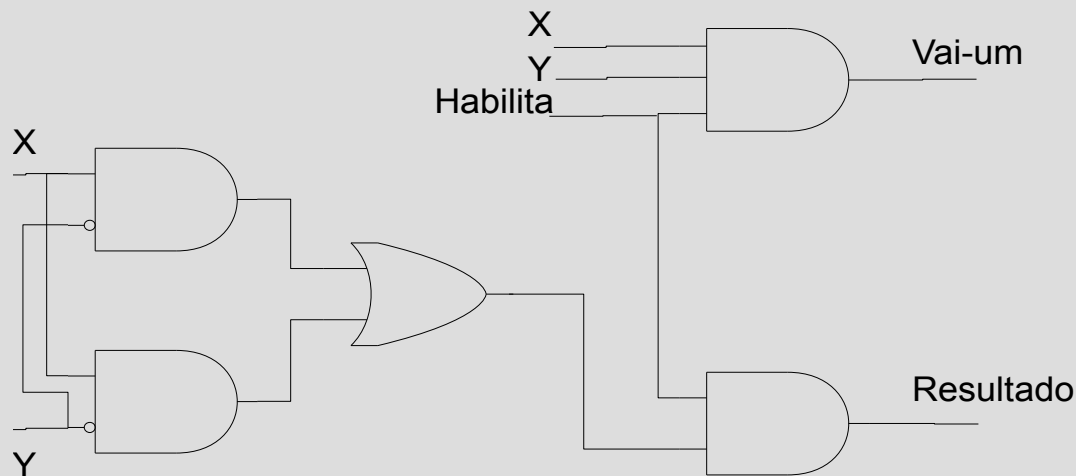


- \* Novamente, o modelo pode ser simulado neste nível para confirmar as equações lógicas.



# Projeto Lógico

**\* Finalmente, uma descrição estruturada é criada no nível de portas.**



**\* Essas portas podem ser obtidas de uma biblioteca de componentes.**

# Características do VHDL

- \* **Suporte para sentenças concorrentes:**
  - no projeto real de sistemas digitais todos os elementos do sistema estão ativos simultaneamente e realizam suas tarefas ao mesmo tempo
- \* **Suporte para Bibliotecas:**
  - Primitivas definidas pelo usuário e pré-definidas pelo sistema podem residir em uma biblioteca.
- \* **Sentenças Seqüenciais**
  - Dá controle sequencial como em um programa comum (isto é, case, if-then-else, loop, etc.)

# Características do VHDL

- \* **Suporte a Projeto Hierárquico**

- \* **Projeto Genérico:**

- **Descrições genéricas são configuráveis em tamanho, características físicas, temporização, condições de operação, etc.**

- \* **Uso de subprogramas:**

- **Habilidade de definir e usar funções e procedimentos;**
- **Subprogramas são utilizados para a conversão explícita de tipos, redefinição de operadores, etc.**

# Características do VHDL

## \* Declaração de tipo e uso:

- **uma linguagem de descrição de hardware em vários níveis de abstração não pode estar limitada a tipos de dados como Bit ou Booleanos.**
- **o VHDL permite tipos inteiros, de ponto flutuante, enumerados, assim como tipos definidos pelos usuários.**
- **possibilidade de definição de novos operadores para os novos tipos.**

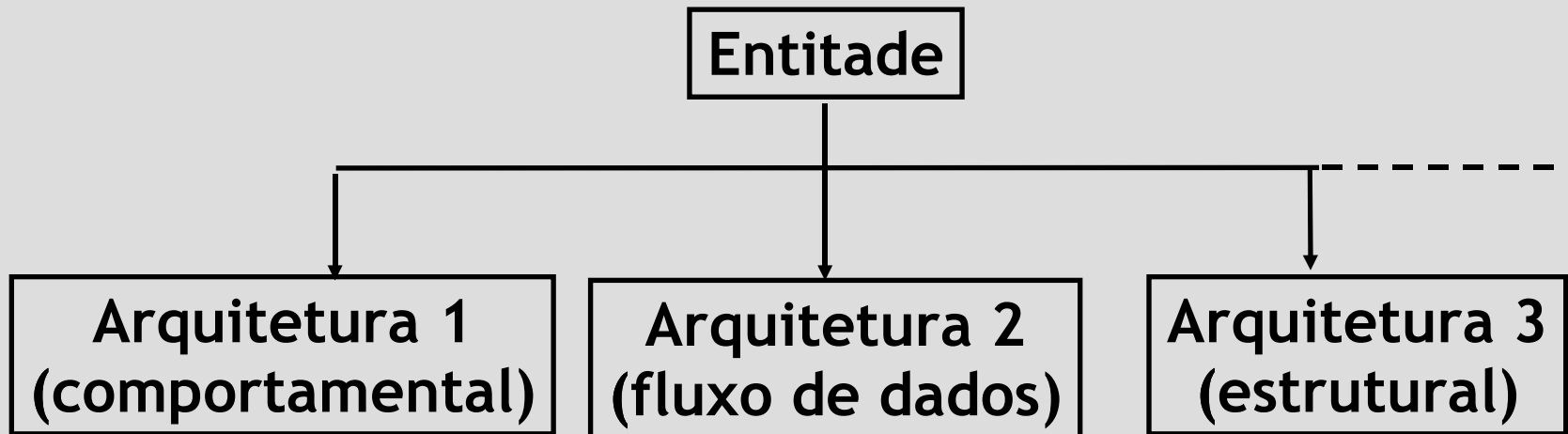
# Características do VHDL

## \* Controle de Temporização:

- Habilidade para especificar temporização em todos os níveis.
- Esquema de distribuição do sinal de relógio é totalmente por conta do usuário, já que a linguagem não tem um esquema pré-definido para isso.
- Construções para detecção de rampa do sinal (subida ou descida), especificação de atraso, etc. estão disponíveis.

## \* Independente de Tecnologia

# Processo de Projeto VHDL



# Declaração de Entidade

- \* **Uma declaração de entidade (ENTITY) descreve a interface do componente.**
- \* **Uma cláusula PORT indica as portas de entrada e saída.**
- \* **Uma entidade pode ser pensada com um símbolo para um componente.**

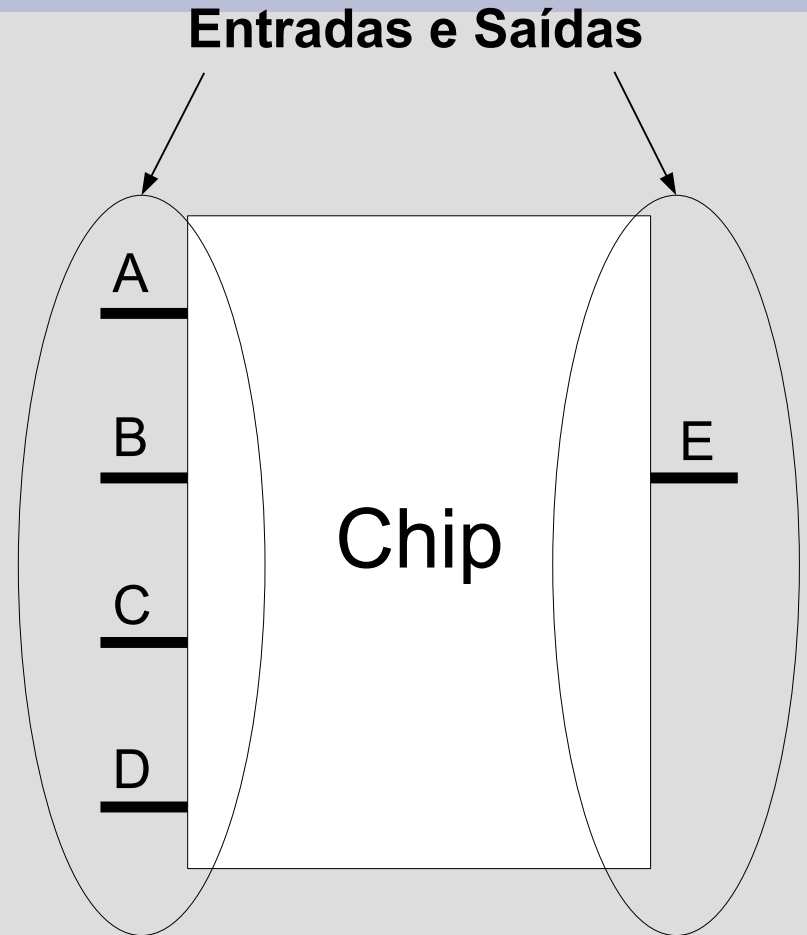
# Entidade

- Define entradas e saídas
- Exemplo:

Entity test is

```
Port ( A,B,C,D: in std_logic;  
      E: out std_logic);
```

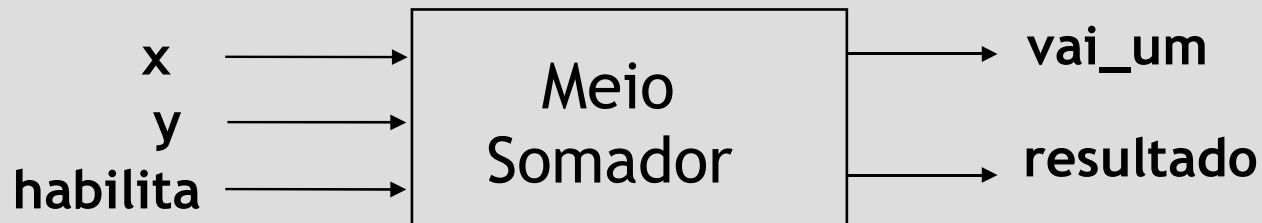
```
End test;
```





# Declaração de Entidade

```
ENTITY meio_somador IS  
  PORT (x, y, habilita: IN bit;  
        vai_um, resultado: OUT bit);  
END meio_somador;
```



# Declaração de Porta

- \* **Uma declaração de porta (PORT) estabelece a interface entre o componente e o mundo externo**
- \* **Há três partes na declaração PORT**
  - **Nome**
  - **Modo**
  - **Tipos de Dados**

```
ENTITY teste IS  
    PORT (<nome> : <modo> <tipos_dados>);  
END teste;
```

# Nome

Qualquer identificador legal em VHDL

- \* Apenas letras, dígitos e sublinhados podem ser usados;
- \* O primeiro caractere deve ser uma letra;
- \* O último caractere não pode ser um sublinhado;
- \* Não são permitidos dois sublinhados consecutivos.

<u>Nomes Legais</u>	<u>Nomes Ilegais</u>
rs_clk	_rs_clk
ab08B	sinal#1
A_1023	A__1023
	rs_clk_

# Nome

- Não é sensível à “Caixa Alta ou Baixa”
  - **inputa, INPUTA e InputA** se referem à mesma variável.
- Comentários
  - ‘--’ marca um comentário até o final da linha atual
  - Se você deseja comentar múltiplas linha, um ‘--’ precisa ser colocado no início de cada linha.
- As sentenças são terminadas por ‘;’
- Atribuição de valores aos sinais: ‘<=’
- Atribuição de valores às variáveis: ‘:=’

# Modo de Porta

- \* **O modo da porta de interface descreve o sentido do fluxo de dados tomando com referência o componente.**
- \* **Os cinco tipos de fluxo de dados são:**
  - **IN:** os dados entram nesta porta e podem apenas ser lidos (é o padrão).
  - **OUT:** os dados saem por essa porta e podem apenas serem escritos.
  - **BUFFER:** similar a Out, mas permite realimentação interna.
  - **INOUT:** o fluxo de dados pode ser em qualquer sentido, com qualquer número de fontes permitido (barramento)
  - **LINKAGE:** o sentido do fluxo de dados é desconhecido

# Tipos de Dados

- \* Os tipos de dados que passam através de uma porta devem ser especificados para completar a interface.
- \* Os dados podem ser de diferentes tipos, dependendo do pacote e bibliotecas utilizados.
- \* Alguns tipos de dados definidos no padrão IEEE são:
  - Bit, Bit\_vector
  - Boolean
  - Integer
  - std\_ulogic, std\_logic

# Tipos de Dados

- **bit values:** '0', '1'
- **boolean** values: TRUE, FALSE
- **integer** values:  $-(2^{31})$  to  $+(2^{31} - 1)$
- **std\_logic** values: 'U','X','1','0','Z','W','H','L','-'
  - U' = uninitialized
  - 'X' = unknown
  - 'W' = weak 'X'
  - 'Z' = floating
  - 'H'/'L' = weak '1'/'0'
  - '-' = don't care
- **std\_logic\_vector** (n downto 0);
- **std\_logic\_vector** (0 upto n);

# Architecture

- \* **Declarações do tipo Architecture descrevem a operação do componente.**
- \* **Muitas arquiteturas podem existir para uma mesma entidade, mas apenas pode haver uma delas ativa por vez.**



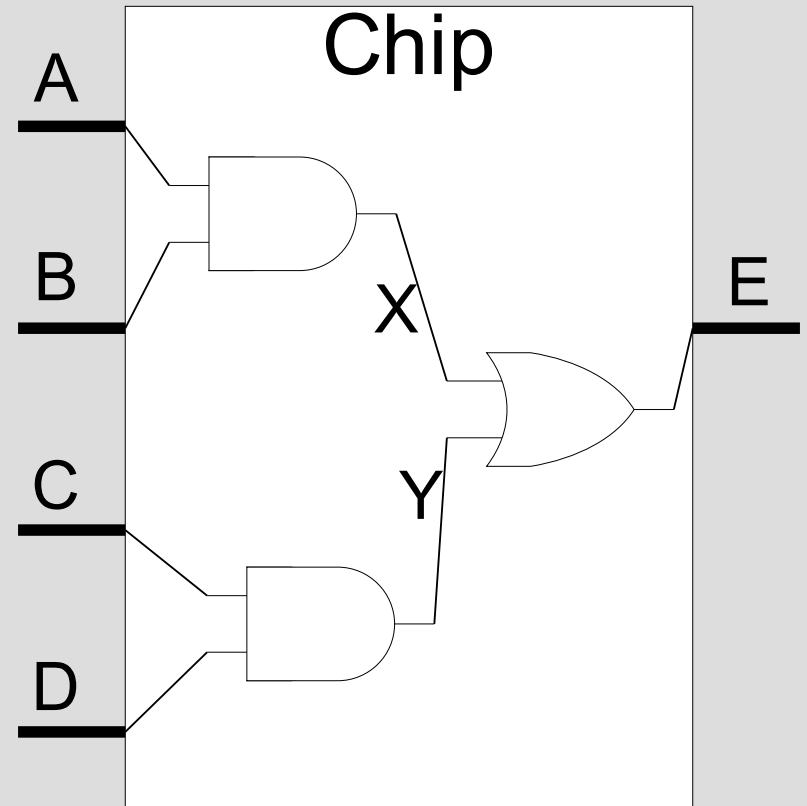
# Architecture

- **Define a funcionalidade do circuito**

$X \leq A \text{ AND } B;$

$Y \leq C \text{ AND } D;$

$E \leq X \text{ OR } Y;$



# Architecture Body # 1

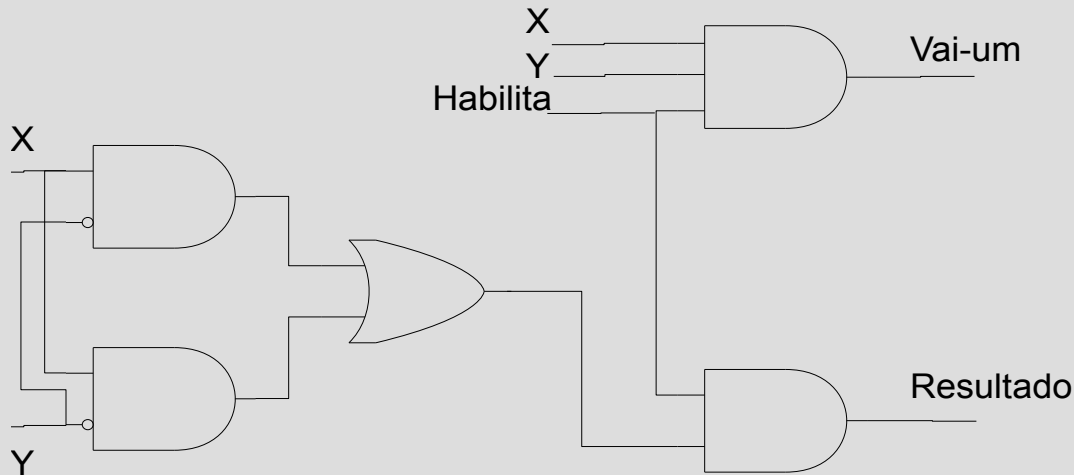
```
ARCHITECTURE behavior1 OF meio_somador IS
BEGIN
    PROCESS (habilita, x, y)
    BEGIN
        IF (habilita = '1') THEN
            resultado <= x XOR y;
            vai_um <= x AND y;
        ELSE
            vai_um <= '0';
            resultado <= '0';
        END PROCESS;
    END behavior1;
```

# Architecture Body # 2

```
ARCHITECTURE data_flow OF meio_somador IS
BEGIN
    vai_um <= (x AND y) AND habilita;
    resultado <= (x XOR y) AND habilita;
END data_flow;
```

# Architecture Body # 3

- \* Para fazer a arquitetura estrutural, nós precisamos primeiro definir as portas a serem utilizadas.
- \* No exemplo a seguir, nós precisamos definir as portas NOT, AND, e OR.



# Architecture Body # 3

```
ENTITY not_1 IS
    PORT (a: IN bit; output: OUT bit);
END not_1;
```

```
ARCHITECTURE data_flow OF not_1 IS
BEGIN
    output <= NOT(a);
END data_flow;
```

```
ENTITY and_2 IS
    PORT (a,b: IN bit; output: OUT bit);
END not_1;
```

```
ARCHITECTURE data_flow OF and_2 IS
BEGIN
    output <= a AND b;
END data_flow;
```

# Architecture Body # 3

```
ENTITY or_2 IS
    PORT (a,b: IN bit; output: OUT bit);
END or_2;
```

```
ARCHITECTURE data_flow OF or_2 IS
BEGIN
    output <= a OR b;
END data_flow;
```

```
ENTITY and_3 IS
    PORT (a,b,c: IN bit; output: OUT bit);
END and_3;
```

```
ARCHITECTURE data_flow OF and_3 IS
BEGIN
    output <= a AND b AND c;
END data_flow;
```

# Architecture Body # 3

ARCHITECTURE structural OF meio\_somador IS

```
COMPONENT and2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT and3 PORT(a,b,c: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT or2 PORT(a,b: IN bit; output: OUT bit); END COMPONENT;  
COMPONENT not1 PORT(a: IN bit; output: OUT bit); END COMPONENT;
```

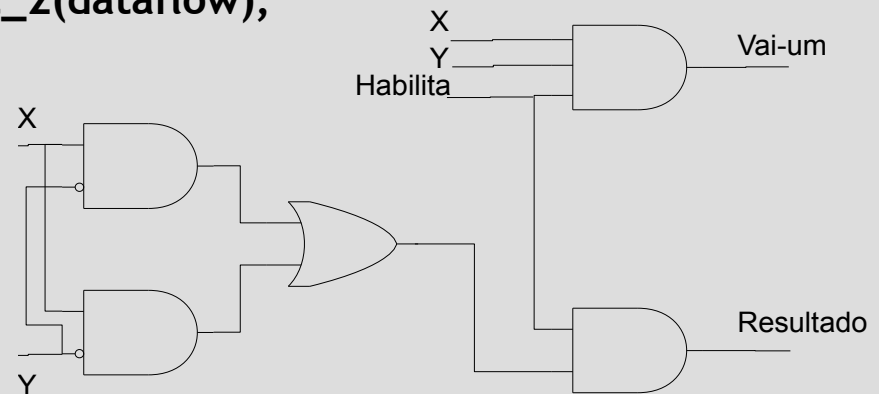
```
FOR ALL: and2 USE ENTITY work.and_2(dataflow);  
FOR ALL: and3 USE ENTITY work.and_3(dataflow);  
FOR ALL: or2 USE ENTITY work.or_2(dataflow);  
FOR ALL: not1 USE ENTITY work.not_2(dataflow);
```

```
SIGNAL v,w,z,nx,nz: BIT;
```

BEGIN

```
c1: not1 PORT MAP (x,nx);  
c2: not1 PORT MAP (y,ny);  
c3: and2 PORT MAP (nx,y,v);  
c4: and2 PORT MAP (x,ny,w);  
c5: or2 PORT MAP (v,w,z);  
c6: and2 PORT MAP (habilita,z,result);  
c7: and3 PORT MAP (x,y,habilita,vai_um);
```

END structural;



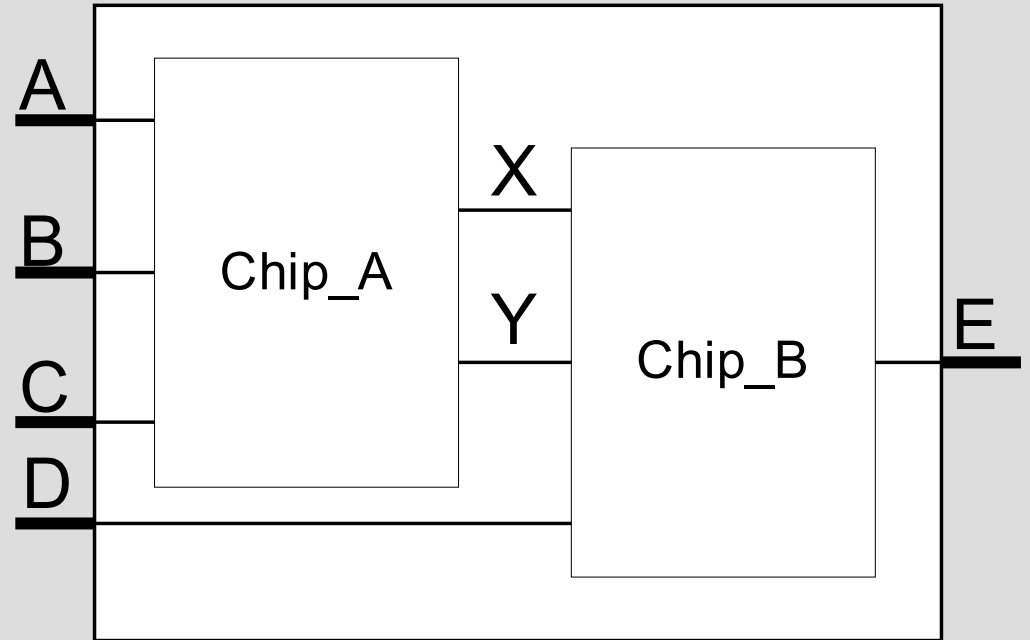
# Port Map

Chip1 : Chip\_A

Port map (A,B,C,X,Y);

Chip2 : Chip\_B

Port map (X,Y,D,E);





# Exemplo Port Map

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY TEST IS
PORT (A,B,C,D : IN STD_LOGIC;
      E       : OUT STD_LOGIC);
END TEST;

ARCHITECTURE BEHAVIOR OF TEST IS

SIGNAL X,Y : STD_LOGIC;

COMPONENT Chip_A
PORT (L,M,N : IN STD_LOGIC;
      O,P   : OUT STD_LOGIC);
END COMPONENT;
```

```
COMPONENT Chip_B
PORT (Q,R,S : IN STD_LOGIC;
      T     : OUT STD_LOGIC);
END COMPONENT;

BEGIN

Chip1 : Chip_A
PORT MAP (A,B,C,X,Y);

Chip2 : Chip_B
PORT MAP (X,Y,D,E);

END BEHAVIOR;
```

# Exemplo AND

```
entity and2 is  
    port ( a, b : in bit; y : out bit );  
end entity and2;
```

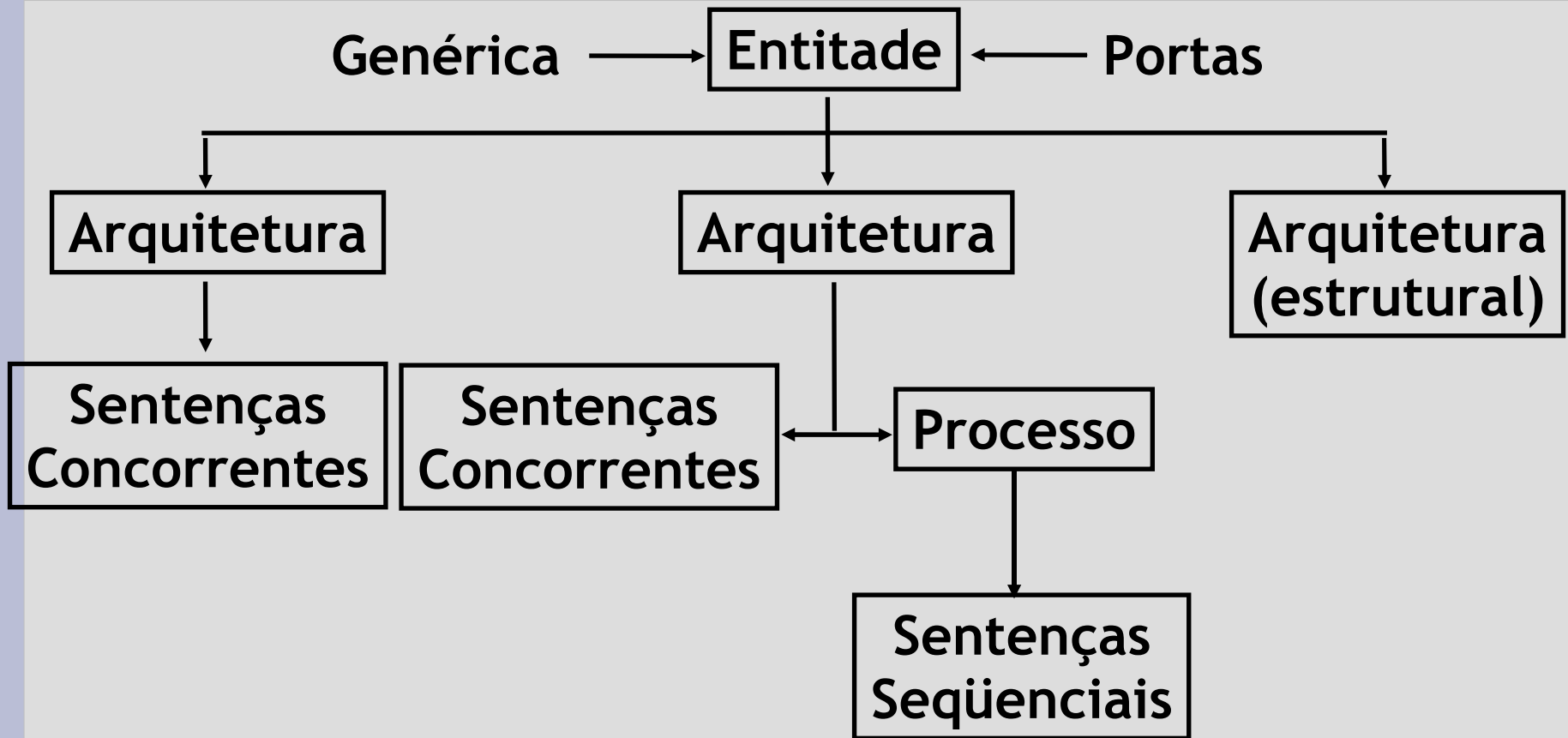
```
architecture basic of and2 is  
begin  
    and2_behavior : process is  
        begin  
            y <= a and b after 2 ns; v z  
            wait on a, b;  
        end process and2_behavior; w  
end architecture basic;
```

# Somador de 1 bit

```
1  ENTITY add1 IS
2  PORT(
3      cin    :IN BIT;
4      a     :IN BIT;
5      b     :IN BIT;
6      s     :OUT BIT;
7      cout  :OUT BIT);
8  END add1;
9
10 ARCHITECTURE a OF add1 IS
11 BEGIN
12
13     s      <= a XOR b XOR cin;
14     cout   <= (a AND b) OR (a AND cin) OR (b AND cin);
15 END a;
```

**FIGURA 6.25**  
Somador completo de 1 bit em VHDL.

# Resumo



# VHDL

- \* **Objetos de Dados**
- \* **Tipos de Dados**
- \* **Tipos e Subtipos**
- \* **Atributos**
- \* **Sentenças Concorrentes e Sequenciais**
- \* **Procedimentos e Funções**
- \* **Pacotes e Bibliotecas**
- \* **Generics**
- \* **Tipos de Atraso**

# Objetos em VHDL

**\* Há quatro tipos de objetos em VHDL:**

- Constantes**
- Sinais**
- Variáveis**
- Arquivos**

**\* Declarações de arquivo tornam um arquivo disponível para uso em um projeto.**

**\* Arquivos podem ser abertos para leitura e escrita.**

**\* Arquivos fornecem um maneira de um projeto em VHDL se comunicar com o ambiente do computador hospedeiro.**

# Constantes

**\*Aumentam a legibilidade do código**

**\*Permitem fácil atualização**

```
CONSTANT <constant_name> : <type_name> := <value>;
```

```
CONSTANT PI : REAL := 3.14;
```

```
CONSTANT WIDTH : INTEGER := 8;
```

# Sinais

- \* **Sinais são utilizados para comunicação entre componentes.**
- \* **Sinais podem ser interpretados com fios físicos, reais.**

```
SIGNAL <nome_sinal> : <tipo> [:= <valor>];
```

```
SIGNAL enable : BIT;
```

```
SIGNAL output : bit_vector(3 downto 0);
```

```
SIGNAL output : bit_vector(3 downto 0) := "0111";
```



# Variáveis

- \* **Variáveis são usadas apenas em processos e subprogramas (funções e procedimentos)**
- \* **Variáveis usualmente não estão disponíveis para múltiplos componentes e processos**
- \* **Todas as atribuições de variáveis tem efeito imediato.**

```
VARIABLE <nome_variavel> : <tipo> [:= <valor>];
```

```
VARIABLE opcode : BIT_VECTOR (3 DOWNT0 0) := "0000";  
VARIABLE freq : INTEGER;
```

# Sinais x Variáveis

- \* Uma diferença fundamental entre variáveis e sinais é o atraso da atribuição

```
ARCHITECTURE signals OF test IS
    SIGNAL a, b, c, out_1, out_2 : BIT;
BEGIN
    PROCESS (a, b, c)
    BEGIN
        out_1 <= a NAND b;
        out_2 <= out_1 XOR c;
    END PROCESS;
END signals;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0

# Sinais x Variáveis

```
ARCHITECTURE variables OF test IS
  SIGNAL a, b, c: BIT;
  VARIABLE out_3, out_4 : BIT;
BEGIN
  PROCESS (a, b, c)
  BEGIN
    out_3 := a NAND b;
    out_4 := out_3 XOR c;
  END PROCESS;
END variables;
```

Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	1

# Escopo dos Objetos

- \* O VHDL limita a visibilidade dos objetos, dependendo de onde eles são declarados.**
- \* O escopo dos objetos é definido como a seguir:**
  - Objetos declarados em um pacote são globais para todas as entidades que usam aquele pacote.**
  - Objetos declarados em uma entidade são globais para todas as arquiteturas que utilizam aquela entidade.**

# Escopo dos Objetos

- **Objetos declarados em um arquitetura são disponíveis a todas as sentenças naquela arquitetura.**
  - **Objetos declarados em um processo são disponíveis apenas para aquele processo.**
- \* Regras de escopo se aplicam a constantes, variáveis, sinais e arquivos.**

# Tipos de Dados



# Tipos Escalares

## \* Tipos Inteiros

- A variação mínima definida pelo padrão é:  
**-2,147,483,647 a +2,147,483,647**

```
ARCHITECTURE test_int OF test IS
BEGIN
    PROCESS (X)
    VARIABLE a: INTEGER;
    BEGIN
        a := 1; -- OK
        a := -1; -- OK
        a := 1.0; -- bad
    END PROCESS;
END TEST;
```

# Tipos Escalares

## \* Tipos Reais

- A faixa mínima definida pelo padrão é: **-1.0E38**

a **+1.0E38**

```
ARCHITECTURE test_real OF test IS
BEGIN
    PROCESS (X)
    VARIABLE a: REAL;
    BEGIN
        a := 1.3; -- OK
        a := -7.5; -- OK
        a := 1; -- bad
        a := 1.7E13; -- OK
        a := 5.3 ns; -- bad
    END PROCESS;
END TEST;
```



# Tipos Escalares

## \* Tipos Enumerados

- É uma faixa de valores definida pelo usuário

```
TYPE binary IS ( ON, OFF );
....
ARCHITECTURE test_enum OF test IS
BEGIN
    PROCESS (X)
    VARIABLE a: binary;
    BEGIN
        a := ON; -- OK
        ....
        a := OFF; -- OK
        ....
    END PROCESS;
END TEST;
```

# Tipos Escalares

## \* Tipos Físicos:

- Podem ter os valores definidos pelo usuário.

```
TYPE resistance IS RANGE 0 to 1000000
UNITS
    ohm; -- ohm
    Kohm = 1000 ohm; -- 1 K
    Mohm = 1000 kohm; -- 1 M
END UNITS;
```

- Unidades de tempo são os únicos tipos físicos pré-definidos em VHDL.

# Tipos Escalares

As unidades de tempo pré-definidas são:

```
TYPE TIME IS RANGE -2147483647 to 2147483647
UNITS
    fs;                -- femtosegundo
    ps = 1000 fs;     -- picosegundo
    ns = 1000 ps;     -- nanosegundo
    us = 1000 ns;     -- microsegundo
    ms = 1000 us;     -- milisegundo
    sec = 1000 ms;    -- segundo
    min = 60 sec;     -- minuto
    hr = 60 min;      -- hora
END UNITS;
```

# Tipos Compostos

## \* Tipo Array:

- Usados para colecionar um ou mais elementos de um mesmo tipo em uma única construção.
- Elementos podem ser de qualquer tipo VHDL.

```
TYPE data_bus IS ARRAY (0 TO 31) OF BIT;
```

```
0 ...element numbers...31
```

```
0 ...array values...1
```

```
SIGNAL X: data_bus;
```

```
SIGNAL Y: BIT;
```

```
Y <= X(12); -- Y recebe o valor do 13o elemento
```

# Tipos Compostos

- \* **Outro exemplo de vetor uni-dimensional (usando a ordenação DOWNTO)**

```
TYPE register IS ARRAY (15 DOWNTO 0) OF BIT;  
15 ...element numbers... 0  
0 ...array values... 1
```

```
Signal X: register;
```

```
SIGNAL Y: BIT;
```

```
Y <= X(4); -- Y recebe o valor do 5o elemento
```

- \* **A palavra DOWNTO ordena os elementos da esquerda para a direita, com elementos de índice decrescente.**

# Tipos Compostos

- \* **Arranjos bi-dimensionais são úteis para a descrição de tabelas da verdade.**

```
TYPE truth_table IS ARRAY(0 TO 7, 0 TO 4) OF BIT;  
CONSTANT full_adder: truth_table := (  
    "000_00",  
    "001_01",  
    "010_01",  
    "011_10",  
    "100_01",  
    "101_10",  
    "110_10",  
    "111_11");
```

# Tipos Compostos

## \* Tipos Record

- Usados para colecionar um ou mais elementos de diferentes tipos em uma única construção
- Elementos podem ser qualquer tipo VHDL
- Os elementos são acessados através no nome do campo

```
TYPE binary IS ( ON, OFF );  
TYPE switch_info IS  
RECORD  
    status : binary;  
    IDnumber : integer;  
END RECORD;  
VARIABLE switch : switch_info;  
    switch.status := on; -- estado da chave  
    switch.IDnumber := 30; -- número da chave
```

# Tipo Access

## \* Access

- Similar aos ponteiros em outras linguagens
- Permitem a alocação dinâmica de memória
- Úteis para a implementação de filas, fifos, etc.



# Subtipos

## \* Subtipos

- Permitem o uso de restrições definidas pelo usuário em um certo tipo de dado.
- Podem incluir a faixa inteira de um tipo básico
- Atribuições que estão fora da faixa definida resultam em um erro.

```
SUBTYPE <name> IS <base type> RANGE <user range>;
```

```
SUBTYPE first_ten IS integer RANGE 1 to 10;
```

# Subtipos

```
TYPE byte IS bit_vector(7 downto 0);  
  
signal x_byte: byte;  
signal y_byte: bit_vector(7 downto 0);  
  
IF x_byte = y_byte THEN ...
```

← O compilador gera um erro.

O compilador não gera nenhum erro.



```
SUBTYPE byte IS bit_vector(7 downto 0)  
  
signal x_byte: byte;  
signal y_byte: bit_vector(7 downto 0);  
  
IF x_byte = y_byte THEN ...
```

# Somador de 4 bits

```
1  ENTITY fig6_22 IS
2  PORT(
3      cin    :IN BIT;
4      a      :IN BIT_VECTOR(3 DOWNTO 0);
5      b      :IN BIT_VECTOR(3 DOWNTO 0);
6      s      :OUT BIT_VECTOR(3 DOWNTO 0);
7      cout   :OUT BIT);
8  END fig6_22;
9
10 ARCHITECTURE a OF fig6_22 IS
11     SIGNAL c :BIT_VECTOR (4 DOWNTO 0); -- carries exigem matrizes de 5 bits
12
13     BEGIN
14         c(0) <= cin;           -- Lê carry de entrada na matriz de bits
15         s <= a XOR b XOR c(3 DOWNTO 0); -- Gera soma dos bits
16         c(4 DOWNTO 1) <=
17             OR (a AND b)
18             OR (a AND c(3 DOWNTO 0))
19             OR (b AND c(3 DOWNTO 0));
20         cout <= c(4);        -- leva para a saída o carry do MSB.
21     END a;
```

**FIGURA 6.22**  
Somador em VHDL.

# Resumo

- \* **O VHDL tem diversos tipos de dados disponíveis para o projetista.**
- \* **Tipos enumerados são definidos pelo usuário**
- \* **Tipos físicos representam quantidades físicas**
- \* **Os arranjos contém um número de elementos do mesmo tipo ou subtipo.**
- \* **Os records podem conter um número de elementos de diferentes tipos ou subtipos.**
- \* **O tipo access é basicamente um ponteiro.**
- \* **Subtipos são restrições definidas pelo usuário para um tipo básico.**

# Atributos

- \* **Atributos definidos na linguagem retornam informação sobre certos tipos em VHDL.**
  - **Tipos, subtipos**
  - **Procedimentos, funções**
  - **Sinais, variáveis, constantes**
  - **Entidades, arquiteturas, configurações, pacotes**
  - **Componentes**
- \* **O VHDL tem diversos atributos pré-definidos que são úteis para o projetista.**
- \* **Atributos podem ser definidos pelo usuários para lidar com registros definidos pelo usuário, etc.**

# Atributos de Sinal

\* A forma geral de um atributo é:

`<nome> ' <identificador_de_atributo>`

\* Alguns exemplos de atributos de sinal

X'EVENT -- avaliado como VERDADEIRO quando um evento no sinal X acabou de ocorrer

X'LAST\_VALUE - retorna o último valor do sinal X

X'STABLE(*t*) - avaliado com VERDADEIRO quando nenhum evento ocorreu no sinal X há pelo menos *t* segundos.

# Atributos

'LEFT - retorna o valor mais a esquerda de um tipo

'RIGHT -- retorna o valor mais a direita de um tipo

'HIGH -- retorna o maior valor de um tipo

'LOW -- retorna o menor valor de um tipo

'LENGTH - retorna o número de elementos de um vetor

'RANGE - retorna a faixa de valores de um vetor

# Exemplos de Atributos

**TYPE count is RANGE 0 TO 127;**

**TYPE states IS (idle, decision, read, write);**

**TYPE word IS ARRAY(15 DOWNT0 0) OF bit;**

<b>count'left = 0</b>	<b>states'left = idle</b>	<b>word'left = 15</b>
<b>count'right = 127</b>	<b>states'right = write</b>	<b>word'right = 0</b>
<b>count'high = 127</b>	<b>states'high = write</b>	<b>word'high = 15</b>
<b>count'low = 0</b>	<b>states'low = idle</b>	<b>word'low = 0</b>
<b>count'length = 128</b>	<b>states'length = 4</b>	<b>word'length = 16</b>

**count'range = 0 TO 127**

**word'range = 15 DOWNT0 0**



# Exemplo de Registrador

- \* Este exemplo mostra como atributos podem ser usados na descrição de um registrador de 8 bits.
- \* Especificações
  - Disparado na subida do relógio
  - Armazena apenas se ENABLE for alto.
  - Os dados tem um tempo de "setup" de 5 ns.

```
ENTITY 8_bit_reg IS
  PORT (enable, clk : IN std_logic;
        a : IN std_logic_vector (7 DOWNTO 0);
        b : OUT std_logic_vector (7 DOWNTO 0);
  END 8_bit_reg;
```

# Exemplo de Registrador

- \* Um sinal do tipo `std_logic` pode assumir os seguintes valores: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'
- \* O uso de `'STABLE` detecta violações de "setup"

```
ARCHITECTURE first_attempt OF 8_bit_reg IS
    BEGIN
        PROCESS (clk)
            BEGIN
                IF (enable = '1') AND a'STABLE(5 ns) AND
                    (clk = '1') THEN
                    b <= a;
                END IF;
            END PROCESS;
        END first_attempt;
```

- \* O que acontece se `clk` for 'X'?

# Exemplo de Registrador

\* O uso de 'LAST\_VALUE assegura que o relógio está saindo de um valor '0'

```
ARCHITECTURE behavior OF 8_bit_reg IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (enable ='1') AND a'STABLE(5 ns) AND
            (clk = '1') AND (clk'LASTVALUE = '0') THEN
            b <= a;
        END IF;
    END PROCESS;
END behavior;
```

# Sentenças Seqüenciais e Concorrentes

- \* **O VHDL provê dois tipos de execução: *Sequencial e Concorrente.***
- \* **Tipos diferentes de execução são úteis para a modelagem de circuitos reais.**
- \* **As sentenças sequenciais enxergam os circuitos do ponto de vista do programador.**
- \* **As sentenças concorrentes tem ordenação independente e são assíncronas.**

# Sentenças Concorrentes

Três tipos de sentenças concorrentes usados em descrições de fluxo de dados

```
graph TD; A[Três tipos de sentenças concorrentes usados em descrições de fluxo de dados] --> B[Equações Booleanas]; A --> C[with-select-when]; A --> D[when-else];
```

**Equações Booleanas**

Para atribuições concorrentes de sinais

**with-select-when**

Para atribuições seletivas de sinais

**when-else**

Para atribuições condicionais de sinais

# Equações Booleanas

```
entity control is port(mem_op, io_op, read, write: in bit;  
                        memr, memw, io_rd, io_wr:out bit);  
end control;
```

```
architecture control_arch of control is  
begin  
    memw <= mem_op and write;  
    memr <= mem_op and read;  
    io_wr <= io_op and write;  
    io_rd <= io_op and read;  
end control_arch;
```

# With-select-when

```
entity mux is port(a,b,c,d: in std_logic_vector(3 downto 0);  
                  s: in std_logic_vector(1 downto 0);  
                  x: out std_logic_vector(3 downto 0));  
end mux;
```

```
architecture mux_arch of mux is  
begin  
  with s select  
    x <= a when "00",  
        b when "01",  
        c when "10",  
        d when others;  
end mux_arch;
```

# with-select-when

```
architecture mux_arch of mux is  
begin  
with s select
```

```
    x <= a when "00",
```

```
        b when "01",
```

```
        c when "10",
```

```
        d when "11",
```

```
        "--" when others;
```

```
end mux_arch;
```

*Possíveis valores  
de s*





# when-else

```
architecture mux_arch of mux is
begin
    x <= a when (s = "00") else
        b when (s = "01") else
        c when (s = "10") else
        d;
end mux_arch;
```

Pode ser  
qualquer  
*condição*  
simples



# Operadores Lógicos

**AND**

**OR**

**NAND**

**XOR**

**XNOR**

**NOT**

\* Pré-definidos para os tipos:

- Bit e boolean.
- Vetores unidimensionais de bits e boolean.

\* Operadores lógicos NÃO TEM ordem de precedência:

$X \leq A \text{ or } B \text{ and } C$   
resultará em erro de compilação.

# Operadores Relacionais

=

<=

<

/=

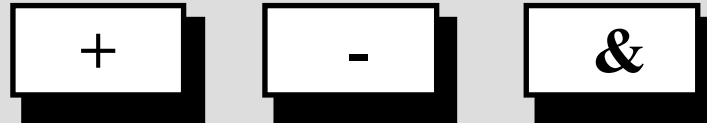
>=

>

- \* Usados para testar igualdade, diferença e ordenamento.
- \* (= and /=) são definidos para todos os tipos.
- \* (<, <=, >, and >=) são definidos para tipos escalares.
- \* Os tipo de operando em uma operação relacional devem ser iguais.

# Operadores Aritméticos

## Operadores de Adição



## Operadores de Multiplicação



## Outros



# Sentenças Seqüenciais

Sentenças seqüenciais são contidas em processos, funções ou procedimentos.

Dentro de um processo a atribuição de um sinal é seqüencial do ponto de vista da simulação.

A ordem na qual as atribuições de sinais são feitas AFETAM o resultado.

# Exemplos

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Reg IS
  PORT(Data_in : IN  STD_LOGIC_VECTOR;
        Data_out: OUT STD_LOGIC_VECTOR;
        Wr      : IN  STD_LOGIC ;
        Reset   : IN  STD_LOGIC ;
        Clk     : IN  STD_LOGIC);
END Reg;

ARCHITECTURE behavioral OF Reg IS
BEGIN
  PROCESS(Wr,Reset,Clk)
    CONSTANT Reg_delay: TIME := 2 ns;
    VARIABLE BVZero: STD_LOGIC_VECTOR(Data_in'RANGE):= (OTHERS => '0');
```

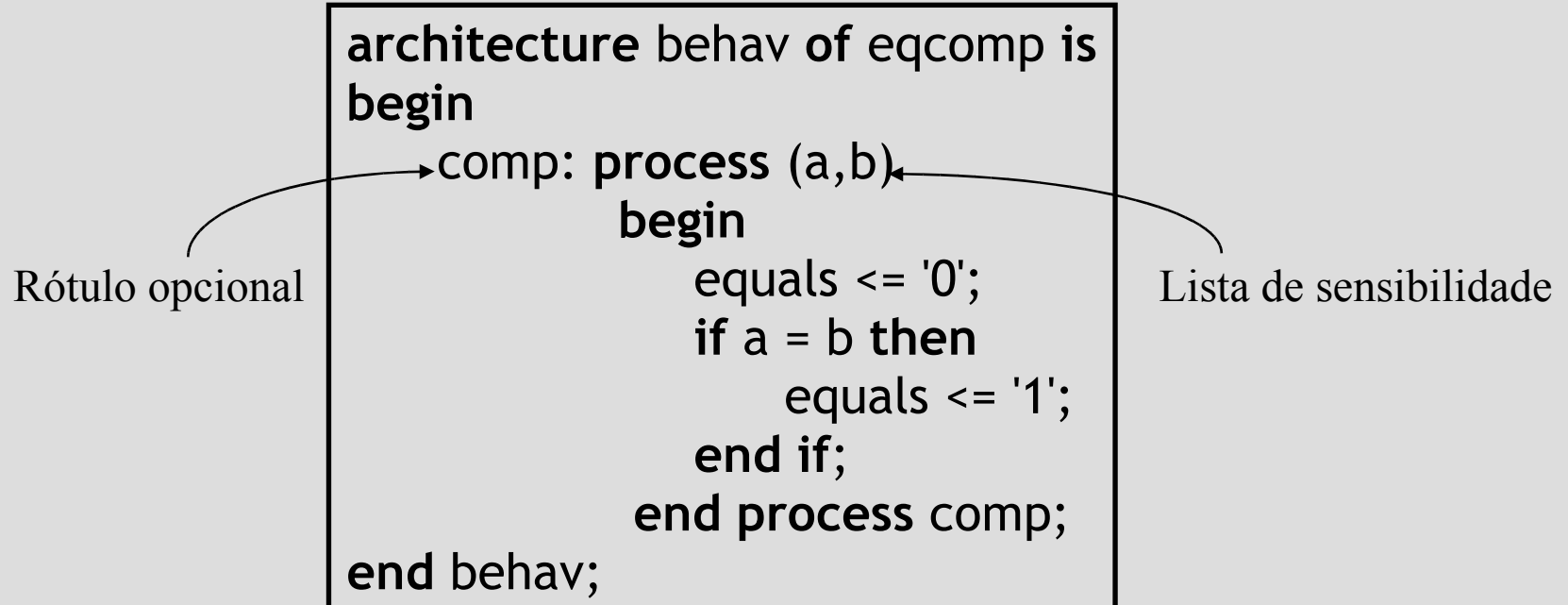
# Exemplos

```
BEGIN
  IF (Reset = '1') THEN
    Data_out <= BVZero AFTER Reg_delay;
  END IF;

  IF (Clk'EVENT AND Clk = '1' AND Wr = '1') THEN
    Data_out <= Data_in AFTER Reg_delay;
  END IF;
END PROCESS;
END behavioral;
```

# Processo

- \* Um processo é uma construção em VHDL que guarda algoritmos
- \* Um processo tem uma lista de sensibilidade que identifica os sinais cuja variação irão causar a execução do processo.





# Processo

O uso do comando wait

```
Proc1: process (a,b,c)
begin
  x <= a and b and c;
end process;
```

Equivalentes



```
Proc2: process
begin
  x <= a and b and c;
  wait on a, b, c;
end process;
```

# Sentenças Seqüenciais

Quatro tipos de sentenças seqüenciais  
são usadas em descrições comportamentais

```
graph TD; A[Quatro tipos de sentenças seqüenciais são usadas em descrições comportamentais] --> B[if-then-else]; A --> C[case-when]; A --> D[for-loop]; A --> E[while-loop];
```

**if-then-else**

**case-when**

**for-loop**

**while-loop**

# if-then-else

```
signal step: bit;  
signal addr: bit_vector(0 to 7);  
    ⋮  
p1: process (addr)  
    begin  
        if addr > x"0F" then  
            step <= '1';  
        else  
            step <= '0';  
        end if;  
    end process;
```

```
signal step: bit;  
signal addr: bit_vector(0 to 7);  
    ⋮  
p2: process (addr)  
    begin  
        if addr > x"0F" then  
            step <= '1';  
        end if;  
    end process;
```

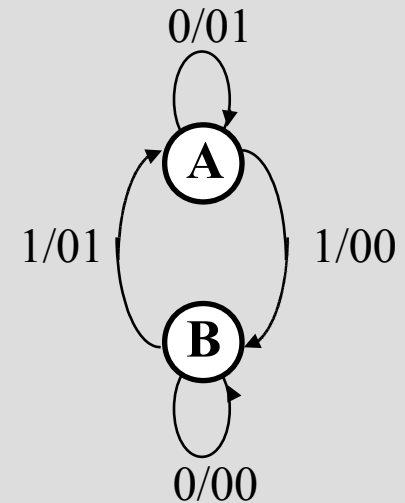
**P2 tem uma memória implícita**

# if-then-else

```
architecture mux_arch of mux is
begin
mux4_1: process (a,b,c,d,s)
    begin
        if s = "00" then
            x <= a;
        elsif s = "01" then
            x <= b;
        elsif s = "10" then
            x <= c;
        else
            x <= d;
        end if;
    end process;
end mux_arch;
```

# case - when

```
case present_state is
  when A => y <= '0'; z <= '1';
    if x = '1' then
      next_state <= B;
    else
      next_state <= A;
    end if;
  when B => y <= '0'; z <= '0';
    if x = '1' then
      next_state <= A;
    else
      next_state <= B;
    end if;
end case;
```



entradas: x  
saídas: y,z

# Detector de Moeda

```
ENTITY    fig4_64 IS
PORT( q, d, n:IN BIT;                -- quarter, dime, nickel
      cents :OUT INTEGER RANGE 0 TO 25); -- valor binário das moedas
END fig4_64;
ARCHITECTURE detector of fig4_64 IS
  SIGNAL  moedas:BIT_VECTOR(2 DOWNT0 0);-- grupo de sensores de moedas
  BEGIN
    moedas <= (q & d & n);            -- atribui sensores para o grupo
    PROCESS (moedas)
      BEGIN
        CASE (moedas) IS
          WHEN "001" => cents <= 5;
          WHEN "010" => cents <= 10;
          WHEN "100" => cents <= 25;
          WHEN others => cents <= 0;
        END CASE;
      END PROCESS;
    END detector;
```

FIGURA 4.64

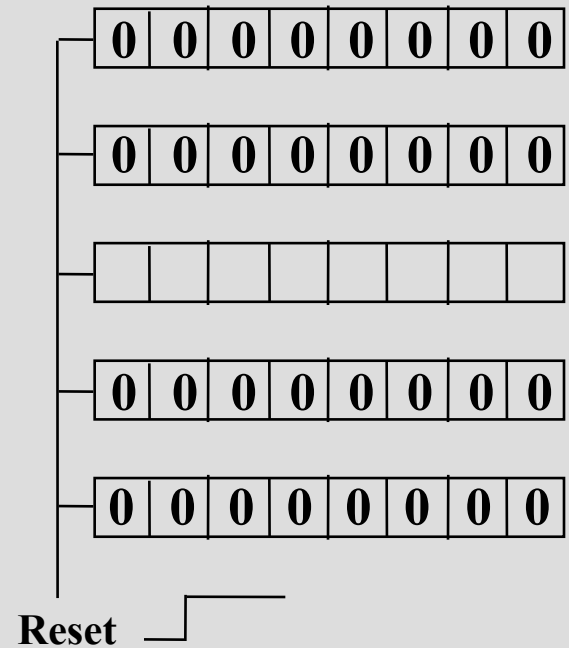
Um detector de moeda em VHDL.



# while-loop

```
type register is bit_vector(7 downto 0);
type reg_array is array(4 downto 0) of register;
signal fifo: reg_array;

process (reset)
  variable i: integer := 0;
begin
  if reset = '1' then
    while i <= 4 loop
      if i /= 2 then
        fifo(i) <= (others => '0');
      end if;
      i := i + 1;
    end loop;
  end if;
end process;
```





# Funções e Procedimentos

**\* Construções de alto nível que são comumente utilizadas para:**

- Conversões de tipo**
- "Operator overloading"**
- Alternativa à instanciação de componentes**
- Qualquer outra definição feita pelo usuário**

**\* Os subprogramas de uso mais freqüente são pré-definidos nos seguintes padrões:**

- IEEE 1076, 1164, 1076.3**

# Funções

## Conversão de Tipo

```
function bv2I (bv: bit_vector) return integer is
  variable result, onebit: integer := 0;
begin
  myloop: for i in bv'low to bv'high loop
    onebit := 0;
    if bv(i) = '1' then
      onebit := 2**(i-bv'low);
    end if;
    result := result + onebit;
  end loop myloop;
  return (result);
end bv2I;
```

# Funções

## Conversão de Tipo

\* As sentenças dentro de uma função devem ser sequenciais.

---

\* Os parâmetros das funções devem ser apenas de entrada e não podem ser modificados.

---

\* Nenhum sinal novo pode ser declarado em uma função, mas as variáveis podem.

# Funções

## Componentes Simples

```
function inc (a: bit_vector) return bit_vector is
  variable s: bit_vector (a'range);
  variable carry: bit;
begin
  carry := '1';
  for i in a'low to a'high loop
    s(i) := a(i) xor carry;
    carry := a(i) and carry;
  end loop
  return (s);
end inc;
```

**\* Funções são restritas para substituir componentes com apenas uma saída.**

# Funções

## Overloading

```
function "+" (a,b: bit_vector) return
bit_vector is
    variable s: bit_vector (a'range);
    variable c: bit;
    variable bi: integer;
begin
    carry := '0';
    for i in a'low to a'high loop
        bi := b'low + (i - a'low);
        s(i) := (a(i) xor b(bi)) xor c;
        c := ((a(i) or b(bi)) and c) or
            (a(i) and b(bi));
    end loop;
    return (s);
end "+";
```

```
function "+" (a: bit_vector; b: integer)
return bit_vector is
begin
    return (a + i2bv(b,a'length));
end "+";
```

# Usando Funções

## Funções podem ser definidas:

- \* Na região declarativa de uma arquitetura (visível apenas para aquela arquitetura)
- \* Em um pacote (package) (é visível para quem usar o pacote)

```
use work.my_package.all
architecture myarch of full_add is
begin
  sum <= a xor b xor c_in;
  c_out <= majority(a,b,c_in)
end;
```

```
use work.my_package.all
architecture myarch of full_add is
  . } ← Colocamos aqui a
  . } definição da função
begin
  sum <= a xor b xor c_in;
  c_out <= majority(a,b,c_in)
end;
```

# Procedimientos

```
entity flop is port(clk: in bit;  
    data_in: in bit_vector(7 downto 0);  
    data_out, data_out_bar: out bit_vector(7 downto 0));  
end flop;
```

architecture design of flop is

```
procedure dff(signal d: bit_vector; signal clk: bit;  
    signal q, q_bar: out bit_vector) is  
begin  
    if clk'event and clk = '1' then  
        q <= d; q_bar <= not(d);  
    end if;  
end procedure;
```

```
begin  
    dff(data_in, clk, data_out, data_out_bar);  
end design;
```

# Bibliotecas e Pacotes

- \* **Usados para declarar e armazenar:**
  - **Componentes**
  - **Declarações de tipo**
  - **Funções**
  - **Procedimentos**
  
- \* **Pacotes (packages) e bibliotecas fornecem a habilidade de reuso das construções em várias entidades e arquiteturas.**



# Bibliotecas

- \* **A biblioteca é um lugar no qual as unidades de projeto podem ser compiladas.**
- \* **Existem duas bibliotecas pré-definidas que são as bibliotecas IEEE e WORK.**
- \* **A biblioteca padrão IEEE contém as unidades de projeto padrão do IEEE . (por exemplo os pacotes: std\_logic\_1164, numeric\_std).**
- \* **WORK é a biblioteca padrão.**
- \* **O VHDL só conhece a biblioteca pelo nome lógico**

# Bibliotecas

- \* Uma biblioteca é tornada visível com uso da cláusula ***library***.

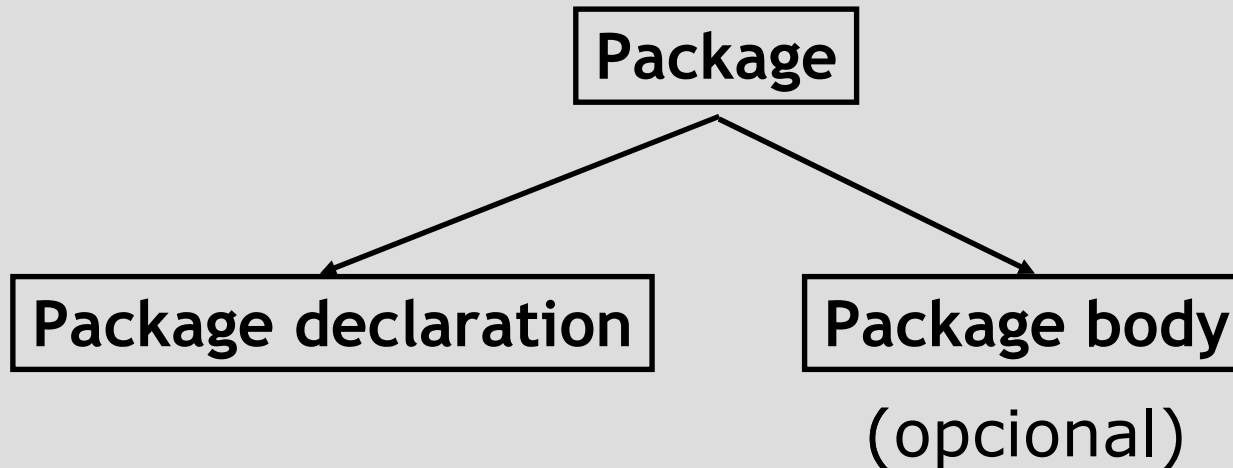
```
library ieee;
```

- \* Unidades de projeto dentro da biblioteca podem também ficar visíveis com o uso da cláusula ***use*** .

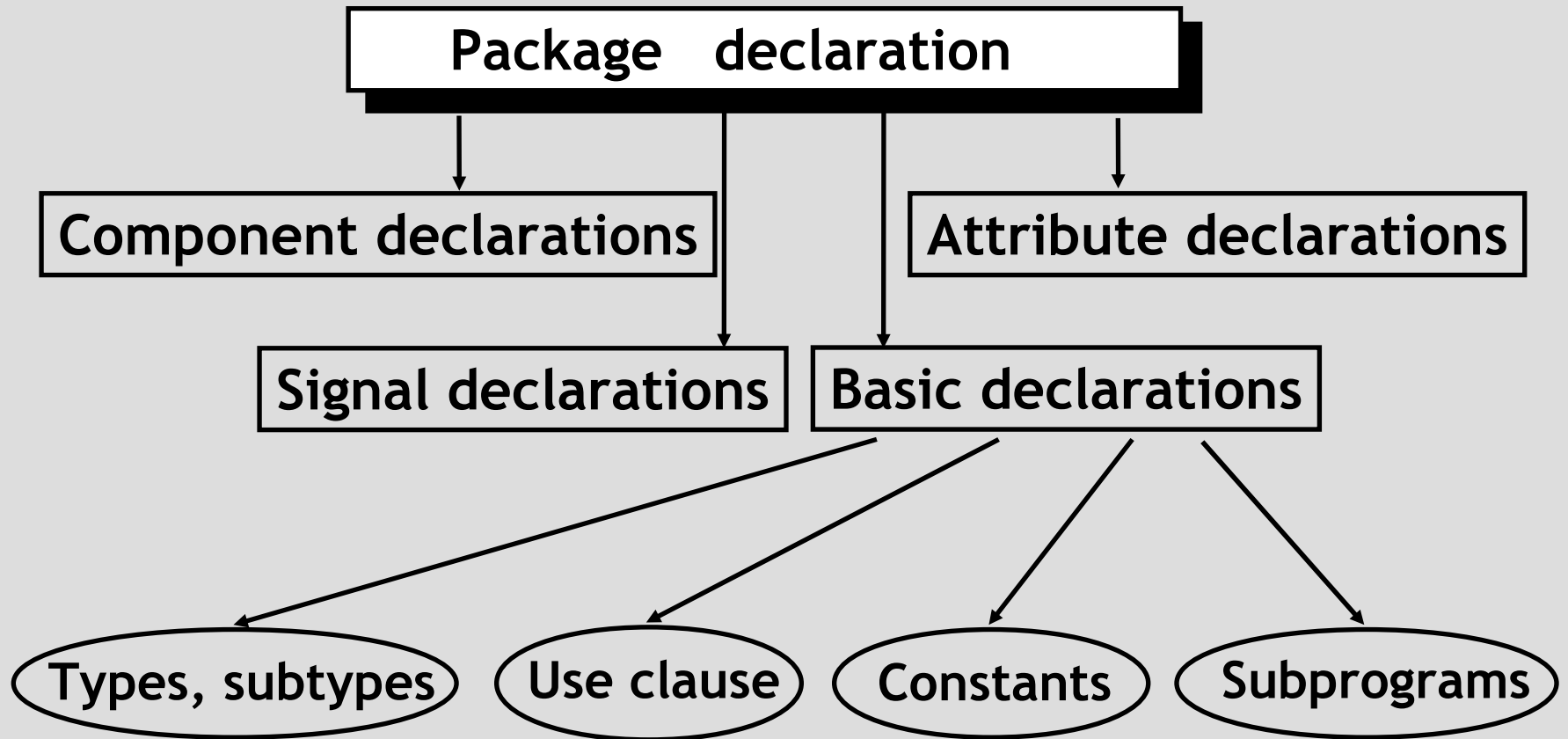
```
for all: and2 use entity work.and_2(dataflow);  
for all: and3 use entity work.and_3(dataflow);  
for all : or2 use entity work.or_2(dataflow);  
for all : not1 use entity work.not_2(dataflow);
```

# Pacotes

**\* Pacotes são usados para fazer as suas construções visíveis para outras unidades de projeto.**




# Packages



# Declaração Package

```
package my_package is
  type binary is (on, off);
  constant pi : real := 3.14;
  procedure add_bits3 (signal a, b, en : in bit;
    signal temp_result, temp_carry : out bit);
end my_package;
```



**O corpo do procedimento é definido no "package body"**

# Package Body

- \* A declaração de um pacote contém apenas as declarações de vários itens
- \* O package body contém subprogram bodies e outras declarações que não serão usadas por outras entidades em VHDL.

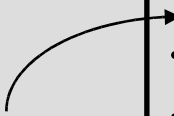
```
package body my_package is
  procedure add_bits3 (signal a, b, en : in bit;
    signal temp_result, temp_carry : out bit) is
  begin
    temp_result <= (a xor b) and en;
    temp_carry <= a and b and en;
  end add_bits3;
end my_package;
```

# Package

Como usar ?

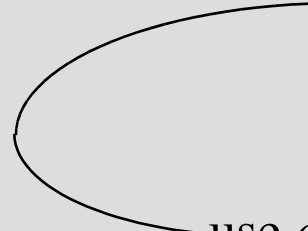
\* Um pacote é tornado visível com uso da cláusula use .

```
use my_package.binary, my_package.add_bits3;  
... entity declaration ...  
... architecture declaration ...
```



usa as declarações *binary* e *add\_bits3*

```
use my_package.all;  
... entity declaration ...  
... architecture declaration ...
```



use *all* para todas as declarações no pacote *my\_package*

# Somador/Subtrator de n bits

```
1  PACKAGE const IS
2      CONSTANT number_of_bits :INTEGER:=8;  -- estabelece número total de bits
3      CONSTANT n :INTEGER:= number_of_bits - 1;  -- número de índice do MSB
4  END const;
5
6  USE work.const.all;
7
8  ENTITY fig6_24 IS
9  PORT(
10     add      :IN BIT;  -- controle da soma
11     sub      :IN BIT;  -- controle da subtração e carry de entrada do LSB
12     a        :IN BIT_VECTOR(n DOWNT0 0);
13     bin      :IN BIT_VECTOR(n DOWNT0 0);
14     s        :OUT BIT_VECTOR(n DOWNT0 0);
15     carryout :OUT BIT);
16  END fig6_24;
17
18  ARCHITECTURE a OF fig6_24 IS
19  SIGNAL c :BIT_VECTOR (n+1 DOWNT0 0);  -- define carries intermediários
20  SIGNAL b :BIT_VECTOR (n DOWNT0 0);    -- define operandos intermediários
21  SIGNAL bnot :BIT_VECTOR (n DOWNT0 0);
22  SIGNAL mode :BIT_VECTOR (1 DOWNT0 0);
23  BEGIN
24     bnot <= NOT bin;
25     mode <= add & sub;
26
27     WITH mode SELECT
28         b <= bin          WHEN "10",    -- add
29         bnot              WHEN "01",    -- sub
30         "0000"           WHEN OTHERS;
31
32     c(0) <= sub;          -- lê carry_in para matriz de bits
33     s <= a XOR b XOR c(n DOWNT0 0);  -- gera bits da soma
34     c(n+1 DOWNT0 1) <= (a AND b) OR
35                       (a AND c(n DOWNT0 0)) OR
36                       (b AND c(n DOWNT0 0));  --gera carries
37     carryout <= c(n+1);  -- leva à saída o carry do MSB
38
39  END a;
```

FIGURA 6.24

Descrição de um somador/subtrator de n bits em VHDL



# Generics

- \* A cláusula **Generics** pode ser adicionada para legibilidade manutenção ou configuração

```
entity half_adder is
  generic (prop_delay : time := 10 ns);
  port (x, y, enable: in bit;
        carry, result: out bit);
end half_adder;
```

O valor padrão quando `half_adder` é usado, se nenhum outro valor for especificado,

Neste caso, uma propriedade genérica chamada `prop_delay` foi adicionada à entidade e definida como 10 ns.

# Generics

```
architecture data_flow of half_adder is
begin
    carry = (x and y) and enable after prop_delay;
    result = (x xor y) and enable after prop_delay;
end data_flow;
```

# Generics

```
architecture structural of two_bit_adder is
  component adder generic( prop_delay: time);
    port(x,y,enable: in bit; carry, result: out bit);
  end component;
```

```
  for c1: adder use entity work.half_adder(data_flow);
  for c2: adder use entity work.half_adder(data_flow);
```

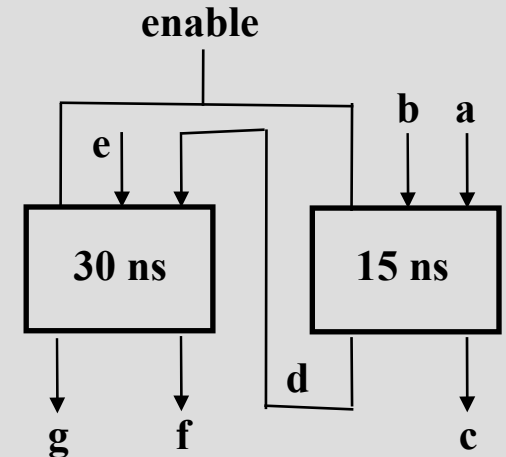
```
  signal d: bit;
```

```
begin
```

```
  c1: adder generic map(15 ns) port map (a,b,enable,d,c);
```

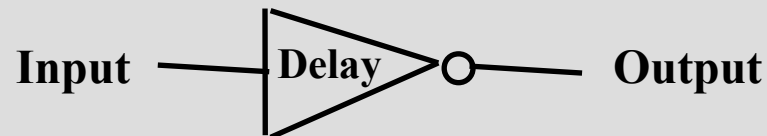
```
  c2: adder generic map(30 ns) port map (e,d,enable,g,f);
```

```
end structural;
```



# Tipos de Atraso (Delay)

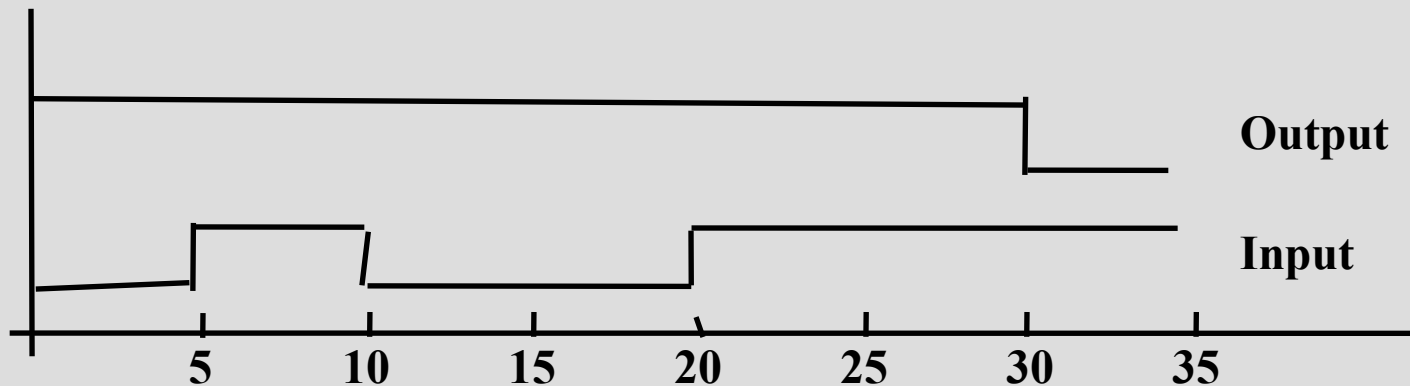
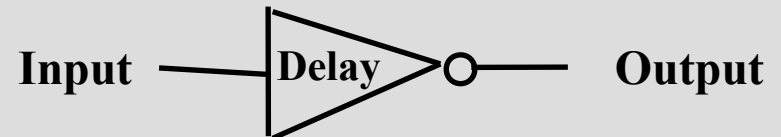
- \* O atraso é criado escalonando uma atribuição de sinal para um tempo futuro
- \* Há dois tipos principais de atraso suportados em VHDL
  - Inercial (Inertial)
  - Transporte (Transport)



# Inertial Delay

- \* O atraso inercial é o padrão para o tipo de atraso
- \* Ele absorve pulsos com duração menor que o atraso especificado

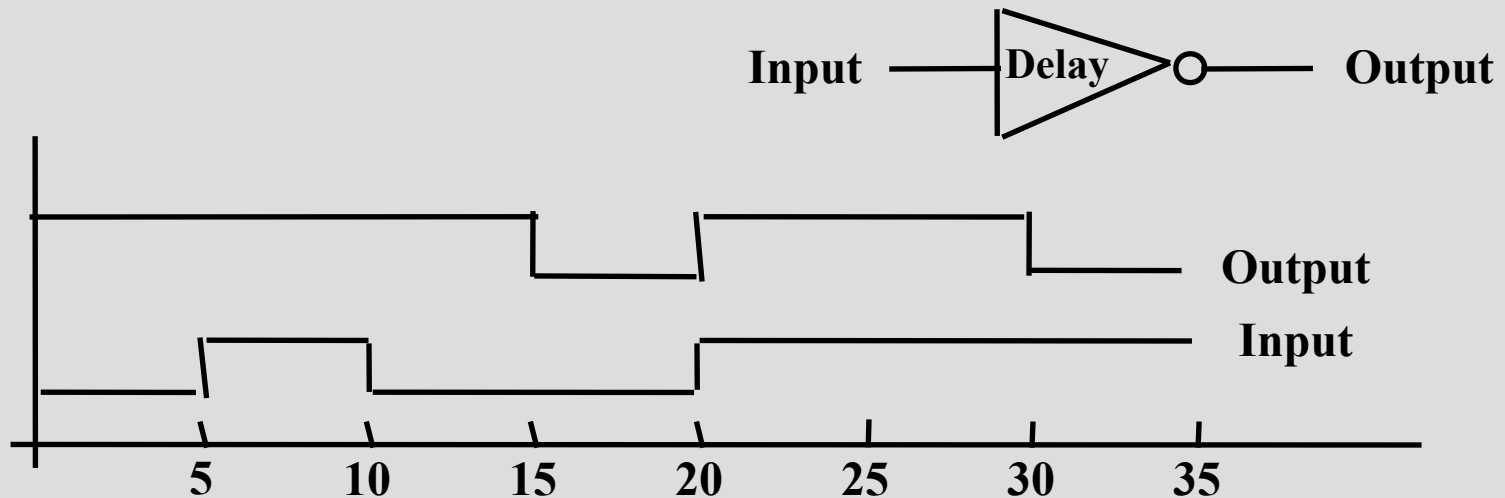
-- Inertial é o padrão  
Output  $\leq$  not Input after 10 ns;



# Transport Delay

- \* Deve ser especificamente especificado pelo usuário
- \* Passa todas as transições de entrada com atraso

```
-- TRANSPORT deve ser especificado  
Output <= transport not Input after 10 ns;
```



# Resumo

- \* **O VHDL é um padrão mundial para a descrição e a modelagem de circuitos lógicos.**
- \* **O VHDL dá ao projetista muitas maneiras diferentes de descrever um circuito.**
- \* **Ferramentas de programação estão disponíveis para projetos simples ou complexos.**
- \* **Os modos de execução sequenciais e concorrentes servem para um grande variedade de necessidade de projetos.**
- \* **Pacotes e bibliotecas permitem o reuso de componentes e o gerenciamento mais adequado do projeto.**

# Palavras Reservadas em VHDL

ABS	DISCONNECT	IN	OF	RETURN	VARIABLE
ACCESS	DOWNTO	INERTIAL	ON		
AFTER		INOUT	OPEN	SELECT	WAIT
ALIAS	ELSE	IS	OR	SEVERITY	WHEN
ALL	ELSIF		OTHERS	SIGNAL	WHILE
AND	END	LABEL	OUT	SHARED	WITH
ARCHITECTURE	ENTITY	LIBRARY	PACKAGE	SLA	
ARRAY	EXIT	LINKAGE	PORT	SLL	XNOR
ASSERT	FILE	LITERAL	POSTPONED	SRA	XOR
ATTRIBUTE	FOR	LOOP	PROCEDURE	SRL	
	FUNCTION		PROCESS	SUBTYPE	
BEGIN		MAP	PURE		
BLOCK	GENERATE	MOD		THEN	
BODY	GENERIC		RANGE	TO	
BUFFER	GROUP	NAND	RECORD	TRANSPORT	
BUS	GUARDED	NEW	REGISTER	TYPE	
		NEXT	REM		
CASE	IF	NOR	REPORT	UNAFFECTED	
COMPONENT	IMPURE	NOT	ROL	UNITS	
CONFIGURATION		NULL	ROR	UNTIL	
CONSTANT				USE	



# Referências

**D. R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.**

**R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.**

**Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*, McGraw-Hill, 1993.**

***IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993.**

# Referências

**J. Bhasker, *A VHDL Primer*, Prentice Hall, 1995.**

**Perry, D.L., *VHDL*, McGraw-Hill, 1994.**

**K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley, 1996**