Sintaxe do VHDL

O VHDL (VHSIC Hardware Description Language) é uma linguagem de descrição de hardware usada para modelar sistemas eletrônicos. Sua sintaxe é estruturada e possui elementos específicos para descrever tanto o comportamento quanto a estrutura de circuitos digitais.

Estrutura Básica de um Código VHDL

Um design VHDL é tipicamente composto por uma ou mais **unidades de design**. As unidades de design mais fundamentais são:

1. **ENTITY (Entidade):** Define a interface do componente, ou seja, suas portas de entrada e saída. Pense nela como a "caixa preta" do seu circuito, especificando como ele se conecta com o mundo exterior.
    - **Sintaxe:**

      ```
      ENTITY nome_da_entidade IS
          PORT (
              nome_do_sinal_1 : MODO tipo_de_dado;
              nome_do_sinal_2 : MODO tipo_de_dado;
              ...
              nome_do_sinal_n : MODO tipo_de_dado
          );
      END ENTITY nome_da_entidade;
      ```

    - **MODO:** Pode ser `IN` (entrada), `OUT` (saída), `INOUT` (bidirecional) ou `BUFFER` (saída com realimentação interna).
2. **ARCHITECTURE (Arquitetura):** Descreve o funcionamento interno ou a estrutura do componente definido pela entidade. Uma entidade pode ter múltiplas arquiteturas, cada uma representando uma implementação diferente.
    - **Sintaxe:**

      ```
      ARCHITECTURE nome_da_arquitetura OF nome_da_entidade IS
          -- Declarações (sinais, constantes, componentes, tipos, etc.)
      BEGIN
          -- Lógica do circuito (instruções concorrentes)
      END ARCHITECTURE nome_da_arquitetura;
      ```

Elementos Chave da Sintaxe

Bibliotecas e Pacotes (Libraries and Packages)

- **LIBRARY:** Especifica um diretório lógico onde os pacotes compilados são armazenados. A biblioteca padrão mais comum é a `IEEE`.

  ```
  LIBRARY ieee;
  ```

- **USE:** Torna visível o conteúdo de um pacote (ou partes dele) dentro da biblioteca. Pacotes contêm declarações de tipos de dados, funções, componentes, etc.

  ```
  USE ieee.std_logic_1164.ALL; -- Torna todos os elementos do pacote std_logic_1164 visíveis
  USE ieee.numeric_std.ALL;    -- Para operações aritméticas com std_logic_vector
  ```

Identificadores (Identifiers)

Nomes dados a entidades, arquiteturas, sinais, variáveis, etc.

- Devem começar com uma letra.
- Podem conter letras, números e o caractere underscore (`_`).
- Não podem ser palavras reservadas do VHDL (ex: `ENTITY`, `PORT`, `SIGNAL`).
- VHDL não é case-sensitive para identificadores e palavras-chave (ex: `Signal1` é o mesmo que `signal1`).

Tipos de Dados (Data Types)

VHDL é uma linguagem fortemente tipada. Alguns tipos comuns incluem:

- **BIT:** Pode ter os valores `'0'` ou `'1'`.
- **BOOLEAN:** Pode ter os valores `TRUE` ou `FALSE`.
- **INTEGER:** Números inteiros (ex: `-5`, `0`, `42`). O intervalo pode ser especificado.
- **REAL:** Números de ponto flutuante (ex: `3.14`). Não são diretamente sintetizáveis para hardware na maioria dos casos.
- **STD_LOGIC:** Definido no pacote `ieee.std_logic_1164`. É o tipo mais usado para sinais em hardware digital. Possui 9 valores possíveis (`'U'`, `'X'`, `'0'`,

`'1'`, `'Z'`, `'W'`, `'L'`, `'H'`, `'-'`) para modelar diferentes estados de um sinal (não inicializado, desconhecido, nível baixo, nível alto, alta impedância, etc.).
- **STD_LOGIC_VECTOR:** Um vetor (array) de `STD_LOGIC`. Usado para representar barramentos de dados.

```
SIGNAL data_bus : STD_LOGIC_VECTOR(7 DOWNTO 0); -- Um barramento de 8 bits
```

- **Array Types:** O usuário pode definir seus próprios tipos de array.
- **Enumerated Types:** O usuário pode definir um tipo com uma lista de valores possíveis.

```
TYPE estado IS (ocioso, lendo, escrevendo);
SIGNAL maquina_estado : estado;
```

Sinais (Signals)

Representam fios ou conexões físicas em um circuito. Sinais mantêm seus valores ao longo do tempo e são usados para comunicação entre processos e componentes.

- **Declaração:** Dentro da seção declarativa de uma arquitetura ou pacote.

```
SIGNAL contador : INTEGER RANGE 0 TO 255;
SIGNAL enable   : STD_LOGIC;
```

- **Atribuição:** Usa o operador `<=` (atribuição de sinal). Atribuições de sinal são agendadas para ocorrer em um momento futuro no ciclo de simulação (geralmente após um "delta cycle" ou um tempo especificado).

```
saida_q <= entrada_d;
```

Variáveis (Variables)

Usadas dentro de processos, procedimentos e funções para computações sequenciais.

- **Declaração:** Dentro da seção declarativa de um processo, procedimento ou função.

```
PROCESS (clk)
    VARIABLE temp_data : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    -- ...
END PROCESS;
```

- **Atribuição:** Usa o operador `:=` (atribuição de variável). A atribuição é imediata, diferente da atribuição de sinal.

```
temp_data := "0000";
```

Processos (Processes)

Blocos de código sequencial dentro de uma arquitetura. Um processo é executado quando um sinal em sua **lista de sensibilidade** muda de valor. Se a lista de sensibilidade for omitida, o processo geralmente contém uma instrução `WAIT` para controlar sua execução.

- **Sintaxe:**

```
nome_do_processo : PROCESS (lista_de_sensibilidade)
    -- Declarações de variáveis locais ao processo
BEGIN
    -- Instruções sequenciais
    -- (IF, CASE, LOOP, atribuições de variáveis, atribuições de sinais)
END PROCESS nome_do_processo;
```

Exemplo com lista de sensibilidade:

```
PROCESS (clk, reset)
BEGIN
    IF reset = '1' THEN
        contador <= (OTHERS => '0');
    ELSIF RISING_EDGE(clk) THEN
        contador <= contador + 1;
    END IF;
END PROCESS;
```

Operadores (Operators)

- **Lógicos:** `AND`, `OR`, `NAND`, `NOR`, `XOR`, `XNOR`, `NOT`
- **Relacionais:** `=`, `/=` (diferente), `<`, `<=`, `>`, `>=`

- **Aritméticos:** `+`, `-`, `*`, `/`, `MOD` (módulo), `REM` (resto), `**` (exponenciação), `ABS` (absoluto)
- **Concatenação:** `&` (usado para juntar arrays ou elementos de array)

```
result <= "00" & input_vector(1 DOWNTO 0);
```

- **Deslocamento e Rotação:** `SLL` (shift left logical), `SRL` (shift right logical), `SLA` (shift left arithmetic), `SRA` (shift right arithmetic), `ROL` (rotate left), `ROR` (rotate right). (Geralmente do pacote `numeric_std`).

Declarações Concorrentes e Sequenciais

Instruções Concorrentes (Concurrent Statements)

São executadas "simultaneamente" dentro de uma arquitetura. A ordem em que aparecem no código não afeta a ordem de execução (exceto para atribuições de sinal com resolução).

- **Atribuição de Sinal Condicional:**

```
saida <= valor_a WHEN condicao_a ELSE
         valor_b WHEN condicao_b ELSE
         valor_c;
```

- **Atribuição de Sinal Selecionada:**

```
WITH seletor SELECT
    saida <= valor_x WHEN "00",
             valor_y WHEN "01",
             valor_z WHEN OTHERS;
```

- **Instanciação de Componente (Port Map):** Usada para descrever a estrutura hierárquica de um design, conectando instâncias de outros componentes.

```
-- Declaração do componente (geralmente em um pacote ou na arquitetura)
COMPONENT meu_componente
    PORT (
        entrada : IN  STD_LOGIC;
        saida   : OUT STD_LOGIC
    );
END COMPONENT;

-- Instanciação na arquitetura
u1 : meu_componente
    PORT MAP (
        entrada => sinal_interno_1,
        saida   => sinal_interno_2
    );
```

- **Generate Statement:** Permite criar instâncias repetitivas de código concorrente.

```
gen_bits : FOR i IN 0 TO N-1 GENERATE
    -- código concorrente a ser repetido
END GENERATE gen_bits;
```

- **Bloco (Block):** Agrupa instruções concorrentes e pode ter suas próprias declarações locais.
- **Chamada de Procedimento Concorrente.**

Instruções Sequenciais (Sequential Statements)

São executadas na ordem em que aparecem dentro de um processo, procedimento ou função.

- **Atribuição de Sinal (`<=`)**
- **Atribuição de Variável (`:=`)**
- **IF Statement:**

```
IF condicao THEN
    -- instruções
ELSIF outra_condicao THEN
    -- instruções
ELSE
    -- instruções
END IF;
```

- **CASE Statement:**

```
CASE expressao IS
    WHEN valor_1 =>
        -- instruções
    WHEN valor_2 | valor_3 => -- Múltiplos valores
        -- instruções
    WHEN OTHERS => -- Para todos os outros valores
        -- instruções
END CASE;
```

- **LOOP Statements:**
  - **LOOP (infinito, requer `EXIT`):**

    ```
    LOOP
        -- instruções
        EXIT WHEN condicao_de_saida;
    END LOOP;
    ```

  - **WHILE LOOP:**

    ```
    WHILE condicao LOOP
        -- instruções
    END LOOP;
    ```

  - **FOR LOOP:**

    ```
    FOR variavel_contadora IN intervalo LOOP
        -- instruções
    END LOOP;
    -- Exemplo: FOR i IN 0 TO 7 LOOP ...
    ```

- **WAIT Statement:** Suspende a execução de um processo.
  - `WAIT ON sinal;` (espera mudar um sinal)
  - `WAIT UNTIL condicao;` (espera uma condição ser verdadeira)
  - `WAIT FOR tempo;` (espera por um período de tempo - usado em simulação)
- **NEXT Statement:** Pula para a próxima iteração de um loop.
- **EXIT Statement:** Sai de um loop.
- **RETURN Statement:** Usado em funções para retornar um valor e em procedimentos para retornar imediatamente.
- **ASSERT Statement:** Verifica uma condição e reporta uma mensagem se for falsa (usado para depuração e verificação).

  ```
  ASSERT (contador < 256) REPORT "Erro: Estouro do contador!" SEVERITY ERROR;
  ```

- **REPORT Statement:** Gera uma mensagem.
- **NULL Statement:** Nenhuma ação.

Comentários

Linhas que começam com dois hifens (`--`) são comentários e são ignoradas pelo compilador.

```vhdl -- Este é um comentário de uma linha.
```