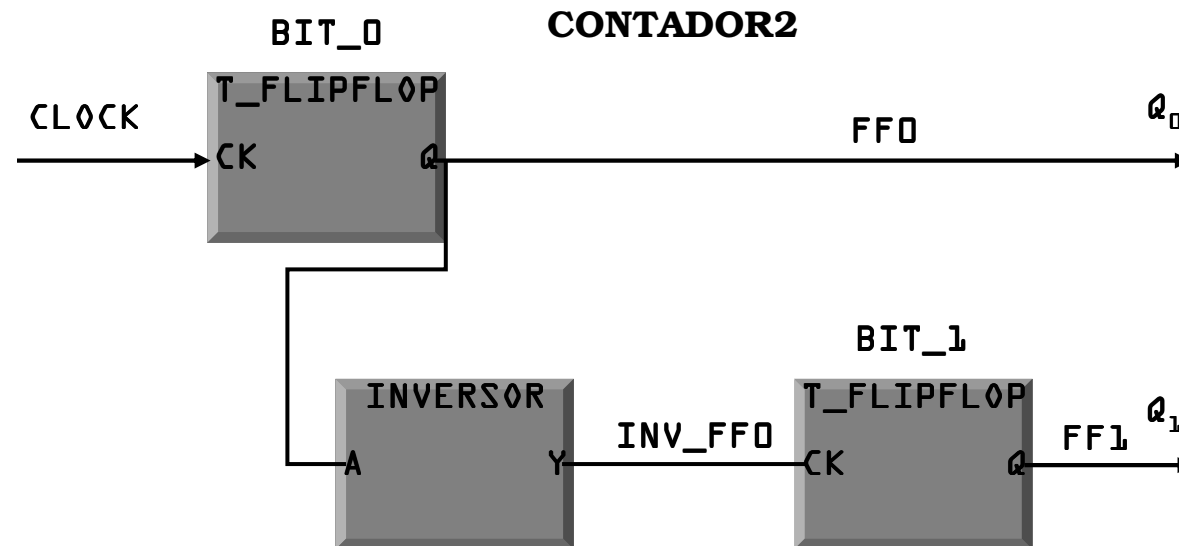


Modelagem de Sistemas com VHDL

A abstração empregada por projetistas de sistemas digitais pode ser expressa em dois domínios:

No domínio estrutural, um componente é descrito em termos de um conjunto de componentes mais primitivos, interconectados.



No domínio comportamental, um componente é descrito em termos de entradas e saídas.

```
architecture comportamento of contador2 is
begin
    process (clock)
        variable conta : natural := 0;
        begin
            if clock = '1' then
                conta := (conta + 1) mod 4;
                q0 <= bit'val (conta mod 2) after atraso;
                q1 <= bit'val (conta / 2) after delay;
            end if;
        end process;
    end comportamento;
```

Uma hierarquia da abstração é um conjunto de níveis de representação interrelacionados, que permitem que um sistema seja representado em diferentes detalhes.

Os níveis de representação comumente usados em projeto de sistemas digitais são:

nível	comportamental	estrutural
sistema	especificações de desempenho	computador, disco
chip	algoritmo	microprocessador, RAM, ROM
registro	fluxo de dados	registro, ALU, multiplexador
porta	equação Booleana	AND, OR, NOT, FF
circuito	equações diferenciais	transistor, resistor, indutor, capacitor
layout	nenhuma	formas geométricas

A quantidade de detalhes requerida para representar um projeto cresce à medida que a hierarquia decresce.

É importante que um projeto seja desenvolvido num nível que tenha detalhes suficientes, mas não excessivos. Detalhes insuficientes produzem resultados imprecisos, ao passo que detalhes demais tornam o projeto caro.

A representação de um projeto pode ser gráfica ou textual. A primeira inclui diagramas em bloco, esquemas lógicos, diagramas de tempo e diagramas de estado. A segunda forma inclui linguagens naturais (e.g., Português), equações (Booleanas ou diferenciais) e linguagens de computador.

Linguagens de programação com estruturas especiais para modelagem de hardware são chamadas **linguagens de descrição de hardware** (*hardware description languages - hdl*).

Geralmente, *texto é melhor para representar comportamentos complexos e gráficos são melhores para ilustrar interações.*

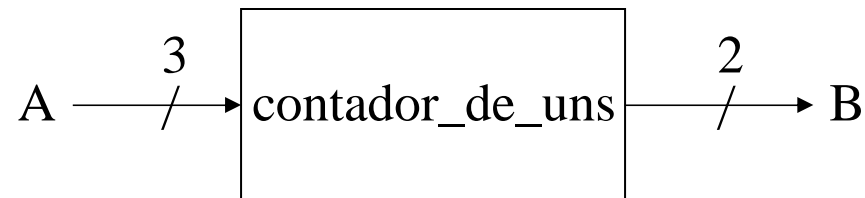
A linguagem de descrição de hardware VHDL

VHDL (*Very high speed integrated circuit Hardware Description Language*) foi originalmente desenvolvida como um método para documentar projetos de circuitos integrados de altíssima velocidade (VHSIC), financiados pelo Departamento de Defesa Americano. Em 1987, foi adotada como um padrão pelo IEEE (IEEE 1076).

Como todas as linguagens de descrição de hardware, possui duas principais aplicações: **documentação** e **modelagem de um projeto**. Boa documentação ajuda a garantir a precisão e portabilidade de um projeto. Modelagem é usada para validar um projeto. Validação através de **simulação** vem rapidamente substituindo o custoso processo de **prototipagem** de um projeto.

Em VHDL, um circuito lógico é representado por uma entidade de projeto denominada **entity**. O circuito pode ser tão complicado quanto um processador ou tão simples quanto uma porta lógica. Uma entidade de projeto consiste de uma *descrição da interface* e um ou mais *corpos de arquitetura*.

Ex: Considere a descrição de interface de um circuito que conta o número de 1's presentes nas suas três entradas:



```
entity contador_de_uns is
port (A : in bit_vector (2 downto 0);
      B : out bit_vector (1 downto 0));
end contador_de_uns;
```

A descrição da interface define o nome da entidade e descreve suas entradas e saídas. A descrição dos sinais da interface inclui tanto o **modo do sinal** quanto o seu **tipo**.

O que é requerido agora é um meio de especificar o comportamento da entidade. Isto é feito através de um **corpo de arquitetura**.

Uma descrição **comportamental**, em linguagens de descrição de hardware, normalmente toma duas formas:

Algorítmica - o procedimento que define a entrada/saída não é feito para representar qualquer implementação física.

Fluxo de dados - as dependências de dados na descrição correspondem àsquelas na implementação real.

No começo do processo de projeto, deve-se usar uma descrição algorítmica.

Durante o estágio de projeto lógico, pode-se descrever o comportamento do contador usando equações Booleanas. Neste caso, a arquitetura algorítmica poderia ser substituída por uma de fluxo de dados.

Enquanto essa descrição implica no uso de estruturas de portas padrão usando AND, OR e NOT, ela é, como a arquitetura algorítmica, ainda **comportamental** no sentido de que especifica as entradas e saídas da entidade, sem exatamente especificar a estrutura interna.

Considerando a saída B como dois sinais, poderíamos decompor o contador de uns em dois componente.

Devem existir descrições de interface e corpos de arquitetura para todos os componentes utilizados dentro do corpo de arquitetura desses dois componentes.

Assim como em outras formas de programação, uma vez que os modelos estejam descritos, eles devem ser testados. Uma forma de fazê-lo é criar uma entidade de mais alto nível denominada **teste**.

Elementos léxicos

Um **comentário** começa com dois hífen adjacentes e se estende até o fim da linha. Ex: `C<=A and B; -- comentário`

Um **identificador** é o nome de um objeto ou é uma palavra reservada. Identificadores devem começar com uma letra e conter somente letras, dígitos e sub-linha. O último caracter deve ser uma letra ou dígito. Não há distinção entre maiúsculas e minúsculas.

Um caracter **literal** é formado a partir de um caracter simples entre apóstrofes. Ex: ‘A’, ‘b’, ‘6’

Um literal do tipo **cadeia de caracteres** é formado a partir de uma seqüência de caracteres entre aspas. Ex: “Uma cadeia”

Um literal do tipo **cadeia de bits** consiste de uma cadeia de dígitos entre aspas e precedida pelo especificador da base. Este pode ser uma letra B (binário), O (octal) ou X (hexadecimal). Ex: B“0110”

Há duas classes de números literais: decimal e baseados. Os **decimais** são expressos na notação decimal padrão, usada em muitas linguagens de programação de alto nível. Ex: 5, 3E3, 23e0 (inteiros); 2.5, 24.65e4 (reais)

Os **baseados** consistem de uma base (entre 2 e 16), seguida de uma sequência de dígitos significativos e um expoente. A base e o expoente são especificados na base 10 e os dígitos significativos devem ser precedidos e seguidos pelo caracter #. Ex: 16#FfF#, 2#1111 (inteiros); 16#0.fff#e3, 2#1.1111#e11.

Tipos de dados e objetos

O tipo de dados de um objeto especifica que valores o objeto pode ter e limita o tipo de operação que pode ser realizada com ele. Em VHDL, há quatro classes de tipos de dados: **escalar**, **composto**, **de acesso**, **arquivo**.

Tipo de dados **escalar** tem valores simples e únicos, e incluem inteiros, reais, quantidades físicas e enumeração.

Tipo de dados **compostos** tem valores complexos e incluem vetores e registros.

Tipo de dados de **acesso** (ponteiros) são usados em conjunto com estruturas de dados dinâmicas, tais como listas encadeadas e filas.

Tipo de dados **arquivo** são usados para armazenar dados usados em programas VHDL. Vetores de teste para um modelo são frequentemente armazenados em arquivos.

Um tipo de dado **inteiro** é um conjunto de valores inteiros, dentro de um limite definido. Ex: `type byte_int is range 0 to 255;`

O tipo inteiro predefinido **integer** tem um limite que depende da implementação, mas deve incluir -2147483647 a +2147483647.

Um tipo de **ponto flutuante** representa uma aproximação discreta de um conjunto de números reais dentro de um limite especificado. Ex: `type signal_level is range -10.00 to 10.00;`

O tipo ponto flutuante predefinido **real** está dentro do limite $-1e38$ a $+1e38$.

Um tipo de dados **físico** é um tipo de dados numérico escalar com um sistema de unidades associado, que é usado para representar alguma quantidade física, tal como massa, comprimento, tempo, voltagem. Uma declaração do tipo físico inclui a especificação de uma **unidade de base** e, possivelmente, unidades secundárias que podem ser múltiplos da unidade base, e.g., o tipo de dados físico predefinido **time**:

```
type time is range definido_pela_implementation
  units
    fs;                -- femtosegundo
    ps = 1000 fs;     -- picosegundo
    ns = 1000 ps;     -- nanosegundo
    us = 1000 ns;     -- microsegundo
    ms = 1000 us;     -- milisegundo
  end units;
```

Um tipo **enumeração** é um conjunto ordenado de identificadores e caracteres.

```
Ex: type cor is (vermelho, verde, azul);  
      type tristate is ('z', '0', '1');  
      type estado (so, s1, s2, s3, s4);
```

Há um número de tipos enumerados predefinidos.

```
Ex: type bit is ('0', '1');  
      type boolean is (false, true);
```

Um tipo **array** é uma coleção de elementos indexados do mesmo tipo.

```
Ex: type palavra is array (31 downto 0) of bit;  
      type memória is array (address) of palavra;  
      type array_2d is array (0 to 15, 0 to 15) of bit;
```

Exemplos de tipos de array predefinidos:

```
type string is array (positivo range <>) of
character;
```

```
type bit_vector is array (natural range <>) of bit;
```

O alcance do índice para o tipo de dados *string* é do tipo positivo. A notação $\langle \rangle$ significa que o alcance é ilimitado e pode ser especificado quando um objeto desse tipo for declarado.

Um tipo **registro** é uma coleção de elementos nominados, possivelmente de tipos diferentes.

```
Ex: type data is record
    dia: integer range 1 to 31;
    mês: nome_do_mês;
    ano: integer range 0 to 3000;
end record;
```

```
Ex: type instrução is record
    código_op: op_processador;
    fonte: integer range 0 to 15;
    destino: integer range 0 to 15;
end record;
```

Um **subtype** é um tipo juntamente com uma restrição. Um valor pertence ao *subtype* se é um valor legítimo para o tipo e satisfaz à restrição. Há dois subtipos inteiros predefinidos:

```
Ex: subtype natural is integer range 0 to
    inteiro_mais_alto;
    subtype positivo is integer range 1 to
    inteiro_mais_alto;
```

O tipo **base** de um subtipo é o tipo do qual foi construído, e.g., o tipo base do subtipo *natural* é *inteiro*.

Um objeto de dados é um repositório no qual valores de um tipo definido podem ser armazenados.

Há três classes de objetos de dados: **constantes**, **variáveis** e **sinais**.

Objetos de dados devem ser declarados antes de serem referenciados em uma declaração de atribuição.

Uma **constante** é um objeto cujo valor é especificado em tempo de compilação e não pode ser modificada durante a simulação.

Ex: `constant nível: probabilidade := 0.75;`
`constant estado_inicial: estado := s0;`
`constant mensagem: string := "Alô";`

Uma **variável** é um objeto cujo valor corrente pode ser modificado por uma declaração VHDL.

```
Ex: variable A, B, C: bit;  
      variable raio: real := 0.0;  
      variable tempo_de_evento: tempo := 1 ns;  
      variable registro: bit_vector (3 downto 0) := B"1010";
```

Variáveis só podem ser declaradas dentro de *processos* ou *subprogramas*.

Um **signal** é um objeto de dados que tem uma dimensão de tempo. Valores futuros podem ser atribuídos ao objeto de dados sem afetar o valor corrente.

```
Ex: signal s1, s2, s3 : bit;  
      signal bus: bit_vector (3 downto 0) := B"0000";
```

Sinais podem não ser declarados dentro de *processos* ou *subprogramas*.

Tipo e objetos declarados numa descrição VHDL podem ter informação adicional, chamada *atributos*, associadas com eles.

Associado com qualquer tipo ou subtipo escalar T tem-se os seguintes atributos:

T'left	limite esquerdo de T
T'right	limite direito de T
T'low	limite inferior de T
T'high	limite superior de T

Para qualquer tipo ou subtipo físico ou discreto T, X um membro de T, e N um inteiro, pode-se usar:

T'pos(X)	posição de X em T
T'val(N)	valor da posição N em T
T'leftof(X)	valor em T que é uma posição à esquerda de X

Há também atributos associados a *arrays*, tais como:

$A'_{range}(N)$ alcance do índice de dimensão N de A ;

$A'_{length}(N)$ comprimento do alcance do índice de dimensão N de A .

e a sinais, tais como:

S'_{active} valor lógico indicando presença de uma transição em S durante o ciclo atual de simulação;

S'_{event} valor lógico indicando a presença de um evento em S durante o ciclo atual de simulação.

Declaração de Atribuição

Um declaração de atribuição a uma variável substitui o valor atual de uma variável por um novo valor determinado por uma expressão. A variável assume seu novo valor instantaneamente. Exemplo:

```
estado_atual := s4;           Registro := B"1111";  
A := '1';                     data_atual.ano := 1993;  
Registro (0) := '0';
```

Declarações de atribuição a sinal escala um novo valor para um sinal a ser realizado em um momento no futuro. O valor atual de um sinal nunca muda. Se nenhum valor definido de tempo é especificado, o valor padrão é um tempo infinitesimal no futuro, conhecido como tempo *delta*. Exemplo:

```
s1<='1' after 10 ns;  
sr1<=5 after 5 ns;  
s2<='0' after 10 ns, '1' after 20 ns;  
reset <= '0';
```

Operadores e Expressões

VHDL comporta os seguintes operadores:

- genéricos **, abs, not
- multiplicação *, /, mod, rem
- sinal +, -
- adição +, -, &
- relacional =, /=, <, <=, >, >=
- lógico and, or, nand, nor, xor

Operadores na mesma linha tem mesma precedência. Operadores numa determinada linha tem maior precedência que os operadores na linha inferior.

Operadores **lógicos** operam sobre variáveis ou sinais do tipo *bit* ou *Boolean*. Eles também podem operar sobre arranjos uni-dimensionais, cujos elementos são desses tipos.

Operadores **relacionais** são usados para comparar valores de operandos do mesmo tipo base, produzindo resultados Booleanos.

Operadores **aditivos** são usados para somar e subtrair operandos numéricos ou para concatenar arranjos uni-dimensionais do mesmo tipo base.

Operadores de **sinal** são usados para mudar o sinal dos operandos numéricos.

Operadores de **multiplicação** são usados para multiplicar e dividir operandos inteiros ou de ponto flutuante.

Declarações de Controle Seqüencial

As declarações condicionais podem ser do tipo “if” ou “case”. Ex:

```
if A < 0 then nível := 1;  
elsif A > 1000 then nível := 3;  
else nível := 2;  
end if;
```

```
case A + B is  
  when 0           => X <= “zero”;  
  when (1 to 20) => X <= “positivo”;  
  when others     => X <= “negativo”;  
end case;
```

Há três tipos de instruções de repetição. Exemplo:

```
while A < B loop
    A := A + 1;
end loop;

for I in 1 to 10 loop
    A(I) := A(I) + 1;
end for;

loop
    A := A + 1;
    exit when X > 10;
end loop;
```

A declaração *next* termina a execução da iteração atual e começa a subsequente. A declaração *exit* termina a execução da iteração atual e do *loop*. Exemplo: `next [etiqueta do loop] [when condição]`

`exit [etiqueta do loop] [when condição]`

Declarações Concorrentes

Declarações do tipo *process* são de duas formas. A primeira contém um lista sensitiva. Exemplo:

```
com_lista: process (s1, s2)
            declaração de constantes e de variáveis;
begin
            comandos seqüenciais;
end process com_lista;
```

Esse tipo de processo é executado uma vez no começo da simulação e, em seguida, somente quando um sinal na lista sensitiva muda de valor.

A outra forma de comando do tipo *process* não tem lista sensitiva.

Exemplo:

```
sem_lista: process
            declaração de constantes e de variáveis;
begin
            comandos sequenciais;
            wait on s1, s2;
end process sem_lista;
```

Processos desse tipo são executados uma vez, no começo da simulação, e continuam a ser executados até que um comando **wait** seja encontrado. Quando o último comando sequencial é executado, o processo recomeça automaticamente a partir do primeiro comando sequencial.

Atrasos

Em VHDL, atrasos podem ser especificados de duas formas. Ex:

```
Y <= X; -- atraso delta
Y <= X after 10 ns; -- atraso de uma unidade de tempo
```

No primeiro comando, o sinal Y recebe o valor atual de X depois de um atraso *delta*. Este atraso corresponde a um período de tempo maior do que zero, mas menor do que qualquer unidade de tempo padrão. No segundo comando, Y recebe o valor de X depois de um atraso determinado pela unidade de tempo.