VHDL

Mini-curso de Computação Híbrida Reconfigurável*

Vitor C. F. Gomes, Andrea S. Charão, Haroldo F. C. Velho

Última atualização: 20 de setembro de 2009

Universidade Federal de Santa Maria / Instituto Nacional de Pesquisas Espaciais
VHDL / Mini-curso de Computação Híbrida Reconfigurável

---

1 Visão Geral

VHDL (VHSIC[1] Hardware Description Language) é uma linguagem de descrição de hardware desenvolvida pelo departamento de defesa estadunidense para documentar o comportamento de circuitos integrados que eram vendidos a este departamento. O objetivo era permitir que o funcionamento de um componente fosse descrito com melhor clareza e que não restringisse a portabilidade.

Com o desenvolvimento de ferramentas que utilizavam a VHDL como linguagem para o a descrição textual de circuitos para a documentação, descrição, simulação, teste e síntese, esta linguagem tornou-se popular neste segmento, levando a sua padronização pelo IEEE em 1987. A partir desta época foram feitas algumas revisões, sendo a versão mais recente de 2008.

Uma descrição em VHDL pode ser usada para simular o comportamento de um circuito eletrônico ou ainda para ser implementado em uma tecnologia, como em FPGA ou ASIC. O processo de compilar a descrição em VHDL para a tecnologia apropriada é chamado de síntese. Diferente do processo de simulação, nem todas as construções em VHDL podem ser sintetizadas em todas as tecnologias, existindo um subconjunto de construções sintetizáveis que devem ser usadas como uma boa prática de programação para interessados na implementação da descrição.

---

*Este mini-curso é uma ação vinculada ao convênio INPE/UFSM e tem por objetivo oferecer uma introdução a Computação Híbrida Reconfigurável

[1] Very High Speed Integrated Circuits

1.1 Estrutura

Em um arquivo VHDL podemos identificar uma estrutura básica que é parecida com o código de uma linguagem de programação de software. Na Listagem 1 temos um exemplo de uma estrutura básica, onde:

- as palavras destacadas são **palavras reservadas** da linguagem ou **tipos de dados**;
- linhas que iniciam com dois hifens adjacentes (--) são comentários e, portanto, descartadas no processo de síntese;
- podemos observar que VHDL não é sensível a maiusculização, ou melhor não é *case-sensitive*. Entretanto, é recomendável que se mantenha um mesmo estilo de programação para aumentar a legibilidade do código.

Nesta descrição, podemos identificar três regiões, que caracterizam um arquivo VHDL básico:

1. a primeira, linha 2 e 3, representa a **inclusão de bibliotecas**. Primeiramente é indicada a biblioteca a ser utilizada (ieee), e então o(s) pacote(s) que devem ser carregados durante a síntese/simulação. Este tipo de declaração é semelhante ao utilizado na linguagem Java.
2. a **entidade**, da linha 6 até a 10, define a interface do sistema, onde são indicados os sinais de entrada e saída. Comparando com uma linguagem de programação, seria o protótipo de uma função, onde indicamos os argumentos e o retorno. Toda entidade deve ter um nome, sendo que a entidade da Listagem 1 é identificada por *porta_and*.
3. a última região, que inicia na linha 13, é a **arquitetura**, onde é feita a implementação da entidade. Neste bloco é definido o comportamento dos sinais e a interação entre eles. Comparando com uma linguagem de programação, seria o corpo de uma função. Para uma mesma entidade, podem existir diversas arquiteturas, devendo possuir nomes diferentes. Na linha 13, pode-se ver o início da arquitetura *arch1* da entidade *porta_and*.

Listagem 1: Estrutura básica de um arquivo em VHDL

```
-- biblioteca                                   1
library ieee;                                   2
use ieee.std_logic_1164.all;                    3
                                                4
-- entidade                                     5
entity porta_and is                             6
   port(a: in std_logic;                        7
        B: IN STD_LOGIC;                        8
        c: out std_logic);                      9
end entity porta_and;                           10
                                                11
-- arquitetura                                  12
architecture arch1 of porta_and is              13
begin                                           14
   c <= a and b;                                15
end architecture arch1;                         16
```

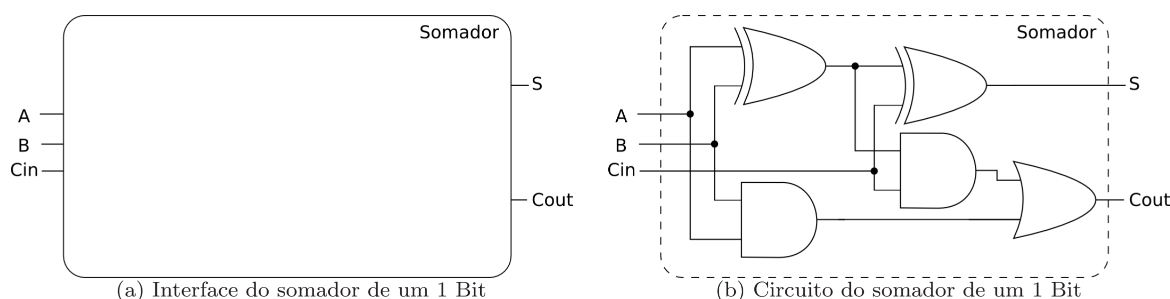
1.2 Exemplo Simples: somador de 1 bit

Para facilitar a compreensão da configuração de um sistema em VHDL usaremos um exemplo simples partindo de seu circuito lógico. Usaremos para este propósito um somador de 1 bit com carry.

Um método para organizar o desenvolvimento em VHDL é elaborar um desenho com o componente a ser implementado, a fim de estabelecer os sinais que farão parte da interface deste componente. Na Figura 1a temos a representação do nosso somador de 1 bit, onde indicamos somente as entradas e as saídas. Neste momento já é possível criar a *entity* no nosso arquivo VHDL. Na Listagem 2 vemos a declaração desta entidade, onde temos os sinais **a**, **b** e **cin** como sinais de entrada e **s** e **cout** como saída.

O próximo passo é implementar a *architecture* da nossa descrição. Para isso é necessário estabelecermos o comportamento do nosso componente. Como já dito anteriormente, usaremos um circuito lógico do somador de 1 bit. Na Figura 1b representamos a interação entre os sinais de entrada para a obtenção dos sinais de saída. Neste momento podemos descrever o comportamento do nosso sistema dentro da nossa arquitetura, que no nosso exemplo é chamada de *simples*.

Na *architecture* da Listagem 2 podemos ver como pode ser descrito o comportamento do nosso circuito. Na linha 14 podemos ver a declaração de um novo sinal que conterá o resultado da operação lógica *xor* entre as entradas **a** e **b**. O uso deste sinal não é essencial nesta descrição, mas facilita a legibilidade do código. Entre as linhas 16 e 18 é descrito as demais operações de nossa descrição.



(a) Interface do somador de um 1 Bit (b) Circuito do somador de um 1 Bit

Figura 1: Somador de 1 bit com carry

Listagem 2: Somador de 1 bit com carry

```
library ieee;                                   1
use ieee.std_logic_1164.all;                    2
                                                3
entity somador1bit is                           4
  port(                                         5
    a    : in  std_logic;                       6
    b    : in  std_logic;                       7
    cin  : in  std_logic;                       8
    s    : out std_logic;                       9
    cout: out std_logic);                       10
end entity somador1bit;                         11
                                                12
architecture simples of somador1bit is          13
  signal s_axorb : std_logic;                   14
begin                                           15
```

```
    s_axorb <= a xor b;                                    16
    s       <= (s_axorb xor cin);                          17
    cout    <= ((a and b) or (s_axorb and cin));           18
end architecture simples;                                  19
```

2 Elementos da Linguagem

2.1 Tipos

A Linguagem VHDL disponibiliza tipos básicos e meios de criar tipos compostos. Tipos básicos são tipos que não tem elemento ou estrutura interna, incluem números inteiros, reais, quantidades físicas e enumerações. Sendo que todos seus valores devem ser possíveis de ser ordenados e estão numa faixa de valores pré-definida. Alguns tipos básicos podem ser vistos na Tabela 1.

Tabela 1: Tipos básicos. Adaptado de [Universidade Federal de Itajubá]

Tipo	Exemplo de valores	Descrição
Boolean	true, false	Diferente de bit
Character	'0', '9', 'A', 'Z', '; '-'	Letras, Números e Caracteres
Integer	0, 1, 123145, -1, -123123	Números inteiros (de $-2^{32}-1$ até $2^{32}-1$)
Real	0.0, 3.14, 5.22E-5	Números em ponto flutuante (de $-1.0E38$ até $1.0E38$)
Bit	'0', '1'	Valores lógicos '0' e '1'

O tipo real pode ser usado em simulações, mas não pode ser sintetizado diretamente. Para projetos que necessitam deste tipo de representação são usadas bibliotecas que dão suporte a operações desta natureza. Mais informações sobre este caso será visto no item 2.1.2.

A declaração de um sinal utilizando um tipo pode ser vista na Listagem 3, tendo sua formulação genérica na primeira linha e um exemplo na linha seguinte.

Listagem 3: Declaração de um sinal

```
-- signal <nome do sinal> : <tipo>;      1
signal meu_sinal : integer;              2
```

Tipos compostos possuem uma estrutura interna, como é o caso de vetores. Vetores podem ser utilizados através de formatos pré-definidos como o caso de *bit_vector* e *string*, onde após o nome do tipo deve ser informada a quantidade de elementos do vetor. Na Listagem 4 temos a declaração de um sinal usando *bit_vector*. Entre as linhas 8 e 15, temos algumas opções para atribuir um valor ao sinal *barramento*. Quando é atribuído um conjunto (mais que 1) de valores, utiliza-se aspas duplas(") e para valores com um caractere, aspas simples ('). Os dados podem ser informados em formato binário (linha 8), octal (linha 9), hexadecimal (linha 10) ou ainda acessando cada elemento do vetor (linhas 12 até 15).

Um segundo modo de estruturar vetores é através da definição de um novo tipo. Na quarta linha da Listagem 4 é mostrado um exemplo da criação de um vetor de bits. A atribuição de valores ao sinal *barramento2* pode ser feita da mesma forma que foi realizada para *barramento*.

Listagem 4: Uso de Vetores

```
-- Usando vetor pre-definido
signal barramento: bit_vector(11 downto 0);
-- Definindo um novo tipo
type meu_vetor is array (11 downto 0) of bit;
signal barramento2: meu_vetor;

-- Atribuicao de valores
barramento <= "101010111100"; -- binario
barramento <= o"5274"; -- octal
barramento <= x"ABC"; -- hexadecimal

barramento(11) <= '1';
barramento(10) <= '0';
...
barramento(0) <= '0';
```

Além dos tipos nativos suportados pela linguagem, existem ainda bibliotecas que definem novos tipos e suas operações, como a *std_logic_1164* que define a representação de valores lógicos padronizados pela IEEE 1164. Esta representação define 9 estados possíveis de um sinal. Esta definição vai em desencontro com a idéia de dois estados lógicos de um sinal ('0' e '1'), entretanto representa os estados físicos reais de um sinal em uma tecnologia CMOS. Os estados definidos são:

- **U**: não inicializado. Útil para detectar possíveis erros em simulações;
- **X**: '0' ou '1' fortes. Garante o sinal em '0' ou '1', mas não em qual destes estados;
- **0**: nível lógico baixo;
- **1**: nível lógico alto;
- **Z**: alta impedância. Representa o terceiro estado de buffers tri-state. Útil para o uso em barramentos de dados;
- **W**: '0' ou '1' fracos;
- **L**: '0' fraco;
- **H**: '1' fraco;
- **-**: Não interessa (*don't care*).

Para a declaração de um sinal usando este padrão pode ser utilizado o *std_logic* e o *std_logic_vector* para vetores. A biblioteca *std_logic_1164* possui ainda fuções para para o tipo *std_logic*, as principais são *falling_edge* e *rising_edge*, que são usadas para detectar a transição de descida ou subida dos sinais especificados. O uso de *rising_edge(clk)* é equivalente aos comandos *clk'event and clk='1'*. Na Listagem 5 são mostradas essas duas formas de uso. A condição do comando *if* somente será satisfeita quando houver a mudança do *clk* para o estado '1', indicando uma borda de subida no *clk*.

Listagem 5: Uso de rising_edge

```
if ( rising_edge(clk) ) then
```

```
    a <= b;
end if;

-- equivalente
if (clk'event and clk = '1') then
    a <= b;
end if;
```

2.1.1 Enumeráveis

O uso de tipos enumeráveis é muito utilizado na descrição de uma máquina de estados. Nesta situação, são criados estados através de palavras que os identificarão. Esta comando é semelhante ao uso do comando *enum* em linguagem C. Sua definição, a declaração e uso são mostrados na Listagem 6. Durante o processo de síntese as palavras **esperar**, **verificar**, **processar** e **terminar** serão transformadas em valores inteiros com quantos bits forem necessários para identificá-las, por isso, não devem existir estados com mesmo nome em tipos diferentes de enumeráveis.

Listagem 6: Enumeráveis

```
type estados is (esperar, verificar, processar, terminar);
signal s_estado : estados;

s_estado <= esperar;

if (s_estado = processar) then
    s_estado <= terminar;
end if;
```

2.1.2 Ponto Flutuante

Números em ponto flutuante podem ser representados com o tipo real. Entretanto este formato não é sintetizável, sendo usado somente no caso de simulação de componentes. Se for necessário o uso deste tipo e de suas operações devem seu usadas bibliotecas que implementam componentes que realizam a manipulação deste formato. Existem algumas bibliotecas de código aberto que podem ser utilizadas, ou ainda podem ser utilizados módulos disponíveis nas ferramentas de desenvolvimento, como no Ise Foundation da Xilinx. Esta ferramenta possui implementações deste tipo de operação otimizadas para os dispositivos que comercializa. Como as operações não são nativas da linguagem, é necessária a utilização de componentes, a modularização com componentes é vista no item 2.9 deste material.

2.2 Operadores Lógicos

O sinais usados em VHDL podem ser combinados através de operadores lógicos. Os operadores *and*, *nand*, *or*, *nor*, *xor*, *xnor* exigem dois operando e o operador *not* apenas um. Essas operações podem ser utilizadas com os tipos *bit*, *boolean*, *bit_vector*, *std_logic*, *std_logic_vector*, etc, desde que os operandos sejam do mesmo tipo. No caso de vetores estas operações são efetuadas bit a bit.

2.3 Operadores Numéricos

Em VHDL estão disponíveis os operadores de adição (+), subtração (-), multiplicação (*), divisão (/), módulo (mod), valor absoluto (abs), resto (rem) e potência (**) para os tipos *integer* e *real*. Para estes operadores numéricos, os operandos devem ser do mesmo tipo. Vale lembrar que nem todas as construções em VHDL são sintetizáveis e, portanto, servirão somente para documentação ou simulação.

2.4 Operadores de Comparação

A comparação entre dois objetos pode ser feita usando os operadores: igual (=), diferente (/=), menor (¡), menor ou igual (¡=), maior (¿) ou maior ou igual (¿=). Os elementos devem ser do mesmo tipo e em caso de vetores com dimensões diferentes, o maior será justificado para o tamanho do menor para a comparação.

2.5 Indexação e Concatenação

Itens de vetores podem ser indexados individualmente através indicação do índice dentro de parenteses () adjacentes ao nome do sinal. Um subvetor pode ser indexado através do uso de *to* e *downto*. Com isso é possível inverter a ordem dos bits de um barramento (vetor). Na Listagem 7 pode ser visto alguns exemplos de indexação de vetores, sendo que na linha 8 o sinal *s_barramento1* recebe os dados de *s_barramento0* em ordem inversa. Na última linha é possível ver o uso do operador de concatenação & para atribuir valores a um vetor.

Listagem 7: Exemplo de indexação e concatenação

```vhdl
signal s_bit1, s_bit1: std_logic;
signal s_barramento0: std_logic_vector(3 downto 0);
signal s_barramento1: std_logic_vector(3 downto 0);
signal s_barramento2: std_logic_vector(3 downto 0);

s_bit1 <= s_barramento(1);
s_barramento0(0) <= s_bit0;
s_barramento1 <= s_barramento0(0 to 3);
s_barramento2 <= s_bit0 & '0' & s_bit1 & '1';
```

2.6 Entidade

A *entity* funciona como a interface do componente a ser implementado. Deve definir os sinais de saída e entrada e seus tipos. Analogamente, seria o protótipo de uma função de uma linguagem de programação, onde são definidos os argumentos e o retorno.

Além dos sinais de entrada e saída, uma entidade pode ter ainda opcionalmente o comando *generic*, que serve para descrever o valor de constantes. Estas constantes podem ser usadas para a confecção de componentes que possam ser modificados através da alteração destas constantes. Uma entidade que utiliza uma constante N para determinar o tamanho dos vetores de entrada pode ser vista na Listagem 8.

Listagem 8: Entidade

```
entity somador is                                        1
   generic(N: positive := 8);                            2
   port(                                                 3
      a    : in  std_logic_vector(N-1 down 0);           4
      b    : in  std_logic_vector(N-1 down 0);           5
      cin  : in  std_logic;                              6
      s    : out std_logic_vector(N-1 down 0);           7
      cout : out std_logic);                             8
end entity somador;                                      9
```

2.7 Arquitetura

A *architecture* é a implementação de uma *entity*, onde é definido o comportamento e iteração dos sinais. É a implementação da função, quando comparada a um programa. Múltiplas *architectures* podem ser definidas para uma mesma entidade, bastando que tenham nomes diferentes.

Um exemplo de *architecture* pode ser visto na Listagem 2, onde é apresentada uma arquitetura que possui 3 linhas, cada uma com uma operação de atribuição. Uma característica importante a ser observada nesta arquitetura, é que a 'execução' destas linhas não acontece sequencialmente na ordem em que aparecem no código, sendo processados em paralelo, visto que representam conexões entre sinais e não instruções a serem executadas em um processador. Este é exatamente o comportamento esperando com o circuito da Figura 1b. As operações serão computadas de forma imediata à alterção de um dos sinais, levando somente o tempo de propagação do sinal pelo condutor. Para testar o funcinamento, troque a ordem das linhas 16, 17 ou 18 e veja o comportamento em um simulador.

2.8 Processos

O comportamento do sistema descrito em uma *architecture* acontece em paralelo, sem que haja um fluxo sequencial de execução. Entretanto, em algumas situações é necessário uma ordenação e a utilização de fluxos de controle. Para estes casos são utilizados *process*. Processos são declarados dentro de *architecture* e a operação de cada linha é executada sequencialmente. Na Listagem 9 temos o exemplo de uma entidade implementado usando processo, que está entre as linhas 16 e 25. A declaração deve ser feita iniciando pela nome do processo, seguido por dois pontos (:) e pela palavra reservada *process*. Dentro dos parenteses vão os sinais que ativam a execução do *process*, ou seja, o processo do nosso exemplo somente será executado quando houver alguma alteração no estado de *clk*. Na ausência de sinais de ativação, o processo será executado novamente a cada ciclo do FPGA. Os sinais que sofrem alterações dentro de um processo, somente terão seus valores alterados efetivamente na finalização do processo.

Ainda na Listagem 9 podemos ver um segundo processo, chamado *segundo_processo*. Considerando que que todos os sinais, de s_a até s_c, estejam com o valor 1 e ocorra uma borda de subida no *clk*, após a execução deste processo os sinais terminam com os valores: s_a = 2, s_b = 2, s_c = 1. A atribuição de 5 ao s_a é descartada pois somente a ultima atribuição é efetivada. O s_b não recebe o valor 6, como poderia ser a intenção, pois s_a somente teria o valor 5 ao final do processo. Em situações onde é necessário que a atribuição ocorra sem atraso, é possível o uso de *variables*, pois o valor atribuído a uma *variable* é válido

imediatamente após a atribuição. No terceiro processo do nosso exemplo, é feita a implementação usando *variables*, e considerando o mesmo estado inicial anterior, no final deste processo teremos: s_a2 = 3, s_b2 = 6 e s_c2 = 3. Observe que foram utilizados outros sinais dentro deste processo, pois os sinais estão no escopo da arquitetura e são compartilhados pelos processos. Uma variável somente pode ter atribuição dentro de um processo, e leituras em quantos necessário. Na Tabela 2 é feita uma entre sinais e variáveis.

Uma última observação a ser feita no código da Listagem 9 é quanto à execução dos processos. Como todos eles são sensíveis ao sinal *clk*, quando este sinal sofrer uma variação, os três processos entrarão em operação concorrentemente, como se fossem processos independentes.

Tabela 2: Comparação entre *Signals* e *Variables*. Adaptado de [Universidade Federal de Itajubá]

a	*Signal*	*Variable*
Declaração	Em *architecture* ou como *port* na *entity*	Dentro de *process*
Escopo	Toda a *architecture*	Processo de declaração
Atribuição	Recebe o valor atribuído ao final do *process*. Somente a última atribuição é valida	Rebebe o valor atribuído imediatamente. Toda atribuição é válida.
Atraso	Inercial e Transporte	Não há

Listagem 9: Uso de processos

```
library ieee;                                                    1
use ieee.std_logic_1164.all;                                     2
                                                                 3
entity entidade is                                               4
   port(                                                         5
      entrada0   : in  std_logic;                                6
      entrada1   : in  std_logic;                                7
      seletor    : in  std_logic;                                8
      clk        : in std_logic;                                 9
      saida      : out std_logic);                               10
end entity entidade;                                             11
                                                                 12
architecture arch1 of entidade is                                13
   signal s_a, s_b, s_c: integer;                                14
   signal s_a2, s_b2, s_c2: integer;                             15
begin                                                            16
   primeiro_processo: process(clk)                               17
   begin                                                         18
      if (clk'event and clk='1') then                            19
         if (seletor = '0') then                                 20
            saida <= entrada0;                                   21
         else                                                    22
            saida <= entrada1;                                   23
         end if;                                                 24
      end if;                                                    25
   end process primeiro_processo;                                26
                                                                 27
   segundo_processo: process(clk)                                28
```

```
    begin
        s_a <= 5;
        s_b <= s_b + s_a;
        s_a <= s_a + 1;
        s_c <= s_a;
    end if;
    end process segundo_processo;

    terceiro_processo: process(clk)
        variable v_a, v_b, v_c: integer;
    begin
        v_a := 5;
        v_b := s_b2 + v_a;
        v_a := s_a2 + 1;
        v_c := v_a;
        s_a2 <= v_a;
        s_b2 <= v_b;
        s_c2 <= v_c;
    end process terceiro_processo;

end architecture arch1;
```

2.8.1 Controles Sequenciais

Como já pode ser observado em exemplos anteriores, dentro de processos podem ser utilizados alguns controles de fluxo. Este controles, assim como em softwares, servem para selecionar as operações que deverão ser executadas.

2.8.2 If

Este controle serve para restringir a execução de algumas operações em certas condições. Na Listagem 10 é exibido um exemplo de uso de *if* usando todas as suas opções. Assim como em linguagens de programação, o *if* também pode ser usado sem o *elsif* e/ou *else*.

Listagem 10: If

```
if (s_a = '1') then
    s_a <= '0';
elsif (s_a = '0') then
    s_a <= '1';
else
    s_b <= '1';
end if;
```

2.8.3 Case

O comando *case* é utilizado para selecionar operações assim como o *if*, entretanto, para muitas condições pode torna-se complicado a utilização deste último. Um está constantemente associado à implementação de máquinas de estado, onde a condição de avaliação é o estado atual da máquina. A Listagem 11 mostra o exemplo de um multiplexador de 4 entradas utilizando um selecionador de dois bits do tipo *std_logic*. Na última linha do *case* observamos a existência do *when others*, para situações não previstas pelas demais. Vale lembrar que *std_logic* possui 9 estados por bit, resultando mais que 4 estados possíveis quando considerados dois bits.

Listagem 11: Case

```
mux_processo: process(clk)                              1
begin                                                   2
  if (clk'event and clk='1') then                       3
    case (seletor) is                                   4
      when "00" => saida <= entrada0;                   5
      when "01" => saida <= entrada1;                   6
      when "10" => saida <= entrada2;                   7
      when "11" => saida <= entrada3;                   8
      when others => saida <= "ZZZZZZZZ";               9
    end case;                                           10
  end if;                                               11
end process mux_processo;                               12
```

2.8.4 For

O comando *for* repete uma sequência de operações por um número específico de vezes. Para isso, utiliza um contador que é incrementado ou decrementado a cada iteração. O contador não precisa ser declarado, sendo considerado uma constante existente somente dentro do laço. No uso desta estrutura é interessante observar que existem algumas construções do laço *for* que não podem ser sintetizados. Na configuração de um hardware não existem estruturas definidas para ser possível executar novamente um comando até que seja atingida uma condição. Esta propriedade é facilmente observável em processadores, mas não ocorrem naturalmente em FPGAs, por exemplo. Sendo assim, um *for* em uma descrição VHDL é 'desenrolado' em tempo de síntese, gerando uma sequência de N vezes a lista de operações que estão internas ao laço. Em casos em que não é possível definir a quantidade de vezes que será executado o laço, não será possível desenrolá-lo, e , portanto, a síntese do componente. Na Listagem 12 temos o exemplo de um laço *for*.

Listagem 12: For

```
for_processo: process(clk)                              1
begin                                                   2
  if (clk'event and clk='1') then                       3
    for indice in 7 downto 0 loop                       4
      meu_sinal(indice) <= '0';                         5
    end loop;                                           6
  end if;                                               7
end process for_processo;                               8
```

2.9 Componentes

Na construção de sistemas mais complexos, é conveniente estruturar a descrição do hardware na forma de componente que possuem sinais conectados. Esta organização permite desenvolver um sistema utilizando uma abordagem *top-down*, onde componentes de maior nível são formados por outros componentes.

Um componente é uma descrição VHDL de um módulo contendo *entity* e sua respectiva *architecture*, que pode ser invocado dentro da *architecture* de um outro componente. Na Listagem 13 é possível ver a declaração de um componente nas linhas 11 até 15, onde é declarada, através do comando *component*, a *entity* do componente desejado. A instanciação do componente acontece entre as linhas 21 e 24. Primeiramente é dado um nome para a instância, seguida de pois pontos e o nome do componente a ser instanciado. Na sequência, é feito a conexão dos sinais locais com os sinais do componente através da declaração do *port map*. Independente do sentido dos sinais (*in*, *out*, *inout*), a conexão sempre deve ser feita com ¡sinal do componente¿ =¿ ¡sinal local¿. Um exemplo com o uso de componentes pode ser visto na seção 3.1.

Listagem 13: Uso de componentes

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

entity aplicacao is
   port(saida : out std_logic_vector(3 downto 0));
end entity aplicacao;

architecture simples of aplicacao is
   -- declaracao do componente
   component comp is
      port(a   : in  std_logic_vector(3 downto 0);
           b   : in  std_logic_vector(3 downto 0);
           s   : out std_logic_vector(3 downto 0));
   end component;

   -- sinais
   signal s_a, s_b, s_s: std_logic_vector(3 downto 0);
begin
   -- instanciacao do componente
   instancia_comp: comp
      port map(a => s_a,
               b => s_b,
               s => s_s);
   -- atribuicao
   s_a <= "0101";
   s_b <= "0001";
   saida <= s_s;
end architecture simples;
```

3 Exemplo aprimorado: somador de 4 bits

A fim de mostrar o uso de mais recursos da VHDL é apresentado um exemplo mais aprimorado. Neste exemplo é apresentada a implementação de duas arquiteturas diferentes um somador de 4 bits com carry. A primeira utiliza uma organização estrutural e reutiliza o componente já implementado no primeiro exemplo (Listagem 2). A segunda descreve o comportamento do somador utilizando operações do pacote *std_logic_unsigned* da biblioteca *ieee*. A Listagem 14 apresenta esta implementação.

3.1 Abordagem modular

Na *architecture estrutural* da Listagem 14 é realizada a implementação do somador de 4 bits utilizando o somador de 1 bit implementado anteriormente. A organização desta implementação segue a estrutura mostrada na Figura 2. Acompanhando os sinais desta figura, fica mais fácil de realizar a conexão dos sinais na descrição desta arquitetura.

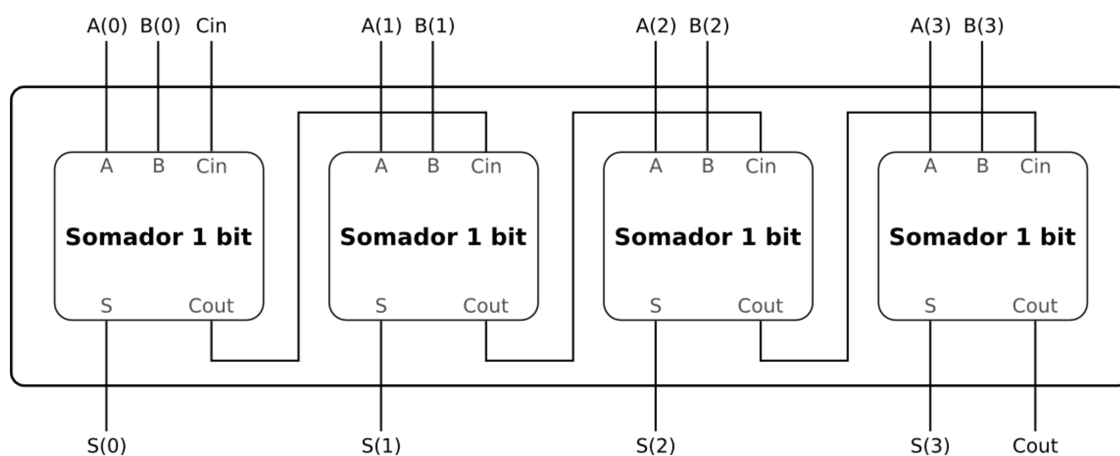


Figura 2: Somador 4 bits com carry

3.2 Usando operadores

O mesmo comportamento encontrado com a implementação da *architecture estrutural* pode ser obtido com a *architecture comportamental*. A diferença entre eles está na forma de implementação das operações. Na comportamental é utilizado a biblioteca *std_logic_unsigned*, a qual abstrai a realização de algumas operações entre sinais. Esta abordagem apesar de resultar em uma menor quantidade de código, não necessariamente representará menos recursos ocupados no hardware, pois cada soma será transformada em um circuito somador, provavelmente, muito semelhante ao implementado no *somador1bit*. Para esta soma com o bit de *carry*, foi realizada a operação considerando 5 bits e então ligado o bit mais significativo ao sinal de *carry out*.

Listagem 14: Somador de 4 bits

```
library ieee;                                     1
use ieee.std_logic_1164.all;                      2
```

```vhdl
use ieee.std_logic_unsigned.all;

entity somador4bits is
   port(
      entrada0 : in  std_logic_vector(3 downto 0);
      entrada1 : in  std_logic_vector(3 downto 0);
      cin      : in  std_logic;
      saida    : out std_logic_vector(3 downto 0);
      cout     : out std_logic);
end entity somador4bits;

architecture estrutural of somador4bits is
   -- componente somador1bit
   component somador1bit is
      port(a    : in  std_logic;
           b    : in  std_logic;
           cin  : in  std_logic;
           s    : out std_logic;
           cout : out std_logic);
      end component;

      signal s_carry: std_logic_vector(2 downto 0);
begin
   -- instancias do somador de 1 bit
   somador1: somador1bit
      port map(a     => entrada0(0),
               b     => entrada1(0),
               cin   => cin,
               s     => saida(0),
               cout  => s_carry(0));
   somador2: somador1bit
      port map(a     => entrada0(1),
               b     => entrada1(1),
               cin   => s_carry(0),
               s     => saida(1),
               cout  => s_carry(1));
   somador3: somador1bit
      port map(a     => entrada0(2),
               b     => entrada1(2),
               cin   => s_carry(1),
               s     => saida(2),
               cout  => s_carry(2));
   somador4: somador1bit
      port map(a     => entrada0(3),
               b     => entrada1(3),
               cin   => s_carry(2),
               s     => saida(3),
               cout  => cout);
end architecture estrutural;
```

```
architecture comportamental of somador4bits is                          54
   signal s_soma: std_logic_vector(4 downto 0);                         55
begin                                                                   56
   s_soma <= ('0' & entrada0) + ('0' & entrada1) + ("000" & cin);       57
   cout <= s_soma(4);                                                   58
   saida <= s_soma(3 downto 0);                                         59
end architecture comportamental;                                        60
```

4 Máquinas de Estado

Máquinas de estado são frequentemente encontradas em sistemas que precisam alterar seu estado de execução ou repetir uma quantidade variável de vezes algumas operações. A descrição de uma máquina de estados envolve a definição de estados através da definição de um *type* e o uso de *case*. Outra característica quando é necessário o uso de Máquinas de estado, é a utilização de um sinal de relógio e um sinal de restabelecimento (*reset*) para que a máquina de estados volte ao estado inicial. Uma implementação de uma máquina de estados que implementa a multiplicação através de somas sucessivas[2] de dois valores com 4 bits pode ser vista na Listagem 15.

Listagem 15: Máquina de Estados: multiplicador

```
library ieee;                                                           1
use ieee.std_logic_1164.all;                                            2
use ieee.std_logic_unsigned.all;                                        3
                                                                        4
entity multiplicador is                                                 5
   port(                                                                6
      entrada0  : in  std_logic_vector(3 downto 0);                     7
      entrada1  : in  std_logic_vector(3 downto 0);                     8
      clk       : in  std_logic;                                        9
      enable    : in std_logic;                                         10
      reset_n   : in  std_logic;                                        11
      saida     : out std_logic_vector(7 downto 0);                     12
      done      : out std_logic);                                       13
end entity multiplicador;                                               14
                                                                        15
architecture simples of multiplicador is                                16
   -- estados da maquina                                                17
   type estados is (desocupada, multiplicando);                         18
                                                                        19
   --sinais                                                             20
   signal s_estado : estados;                                           21
   signal s_operando0: std_logic_vector(3 downto 0);                    22
   signal s_operando1: std_logic_vector(3 downto 0);                    23
   signal s_produto: std_logic_vector(7 downto 0);                      24
   signal s_done: std_logic;                                            25
                                                                        26
   begin                                                                27
```

[2]Existem outros algoritmos para implementar a multiplicação binária de forma mais eficiente.

```
    mul_proc: process(reset_n, clk)
    begin
        if (reset_n = '0') then
            s_done <= '0';
            s_produto <= (others => '0');
            s_estado <= desocupada;
        elsif (clk'event and clk = '1') then
            -- maquina de estado
            case (s_estado) is
                when desocupada =>
                    if (enable = '1') then
                        s_produto <= (others => '0');
                        s_done <= '0';
                        s_estado <= multiplicando;
                        s_operando0 <= entrada0;
                        s_operando1 <= entrada1;
                    end if;
                when multiplicando =>
                    if (s_operando1 /= "0000") then
                        s_produto <= s_produto + ("0000" & s_operando0);
                        s_operando1 <= s_operando1 - 1;
                    else
                        s_done <= '1';
                        s_estado <= desocupada;
                        saida <= s_produto;
                    end if;
            end case;
        end if;
    end process mul_proc;
    done <= s_done;
end architecture simples;
```

Referências

[Cray Inc. 2005] Cray Inc. (2005). *Cray XD1 System Overview*. Mendota Heights, MN, USA.

[Universidade Federal de Itajubá] Universidade Federal de Itajubá. Tutorial de vhdl. Tutorial desenvolvido pelo grupo de microeletrônica.