Apostila Básica de VHDL

Prof. Dr. Renato Giacomini

Objetivo

Esta apostila tem como objetivo introduzir conceitos básicos de VHDL, para alunos de Engenharia Eletrônica ou de Computação, envolvidos em atividades de projeto de hardware digital de alguma complexidade, que exija boa documentação e que busque independência de tecnologia de implementação. Para um aprendizado mais aprofundado do assunto, indicam-se as referências [1], [2] e [3], além das normas IEEE-STD-1076/1987 e IEEE-STD-1164/1993, que estabelecem os padrões internacionais para a linguagem. Outras referências citadas ao longo do texto permitem um aprofundamento em temas e estruturas específicas.

Definições e Abreviaturas

ASIC (Application Specific Integrated Circuits) Circuito integrado de aplicação específica.

CPLD (Complex Programmable Logic Devices) Dispositivo lógico programável complexo.

FPGA (Field Programmable Gate Array) Arranjo de portas programável em campo.

VHDL (VHSIC Hardware Description Language). A sigla é derivada de outra abreviatura - VHSIC (Very High Speed Integrated Circuits), já que seu objetivo inicial era voltado ao projeto de circuitos integrados de altíssima velocidade.

VHSIC (Very High Speed Integrated Circuits) Circuito integrado de altíssima velocidade.

Histórico

Nas décadas de 70 e 80 foi posto em prática um programa do Departamento de Defesa (DoD) americano, para desenvolvimento de circuitos integrados de alta velocidade, denominado VHSIC e originado da preocupação de domínio das tecnologias envolvidas. Desde o princípio, anteviu-se o problema de representação dos projetos segundo uma linguagem que fosse comum às várias empresas

envolvidas e que permitisse uma documentação fechada e clara para projetos de complexidade crescente. Nesta mesma época, já havia no mercado de computação a chamada crise do software, caracterizada pela dificuldade de gestão, documentação e sistematização do ciclo de vida do software (que envolve, em termos gerais, todas as atividades de relativas à sua criação e uso)[1]. O problema surgiu inicialmente no software porque era no software que se desenvolviam as aplicações, que tornavam o produto particular para um uso específico. Isso fazia com que um grande volume de projetos fossem conduzidos, quase sempre com muita complexidade e pouco método. No hardware a questão era mais bem controlada, pois os projetos eram, em geral, mais genéricos, especialmente para o hardware digital. Os projetos de microprocessadores, que eram os mais complexos, visavam na maior parte dos casos o uso geral. Um mesmo microprocessador seria utilizado em várias aplicações, seria reproduzido e testado aos milhões. Isso significava que o investimento em um único projeto poderia ser muito maior, inclusive em termos de especificação e testes. A evolução da metodologia de projeto de software foi então natural, em função dos argumentos sustentados pela crise. Hoje ainda é grande o desenvolvimento de novos métodos e a busca por uma linguagem descritiva universal, como o UML, para especificação e projeto de software orientado a objetos.

A questão preocupante quanto ao hardware, para o programa VHSIC americano, era que alguns fatos indicavam para uma evolução do mercado que faria surgirem problemas semelhantes aos apresentados pela crise de software. Entre estes fatos:

a) A fabricação de circuitos integrados deixava de ser exclusiva de alguns poucos fabricantes. Surgiam empresas especializadas na fabricação de projetos de terceiros, o que permitiria, em conjunto com o desenvolvimento de tecnologias de fabricação menos custosas, a criação de CIs de uso específico para certas aplicações.

b) As novas tecnologias de fabricação (abaixo de 2μm, na época) permitiam um aumento de densidade dos CIs, com conseqüente aumento de complexidade.

c) Surgiam componentes de lógica programável (PLDs e PALs), inicialmente de pequena densidade, mas que poderiam evoluir para densidades muito maiores rapidamente. Tais componentes poderiam criar, como de fato ocorreu posteriormente, um grande mercado de projetos de pequena escala e aplicações específicas, muito parecido com o mercado de projetos de software.

1 Os capítulos iniciais de Pressman tratam com clareza das causas e decorrências da crise de software.

A criação de uma linguagem de descrição de hardware patrocinada pelo programa e, em paralelo algumas outras linguagens criadas pela indústria, foi então uma decorrência natural. Em 1987, o VHDL foi normalizado pelo IEEE, tornando-se um padrão mundial, ao lado do Verilog, uma alternativa também expressiva no mercado de projetos de hardware. Hoje, praticamente todas as ferramentas de desenvolvimento de hardware computadorizadas aceitam essas linguagens como entrada, de forma que um projeto baseado em VHDL ou Verilog pode ser implementado com qualquer tecnologia.

Vantagens do Uso

A alternativa para uso de uma linguagem formal de descrição de hardware como o VHDL é a descrição por diagramas esquemáticos. O VHDL apresenta as seguintes vantagens:

a) Em sistemas seqüenciais, o detalhamento da lógica de controle é realizado pelas ferramentas de automação do projeto, o que evita a trabalhosa e limitada aplicação das técnicas manuais tradicionais;

b) O objetivo do projeto fica mais claro que na representação por esquemáticos, nos quais a implementação se sobrepõe à intenção do projeto;

c) O volume de documentação diminui, já que um código bem comentado em VHDL substitui com vantagens o esquemático e a descrição funcional do sistema;

d) O projeto ganha portabilidade, já que pode ser compilado em qualquer ferramenta e para qualquer tecnologia. É comum, na indústria, o uso de FPGAs e CPLDs para produções iniciais ou de menores escalas em projetos que posteriormente possam ser implementados em ASICs. Todas as implementações podem usar o mesmo código VHDL.

Ciclo de Projeto

O projeto de um sistema digital auxiliado por ferramentas computadorizadas segue normalmente as etapas descritas a seguir.

1- Especificação

Esta etapa visa determinar os requisitos e funcionalidade de projeto, incluindo timing dos sinais e

definido completamente as interfaces. É uma etapa comum e necessária a qualquer projeto, independentemente do uso do VHDL.

2- Codificação

Nesta etapa, o objetivo é descrever, de forma estruturada e bem documentada, em VHDL, todo o projeto, segundo seus padrões de sintaxe. Assim, formaliza-se a especificação da etapa anterior, numa implementação de alto nível, em que são descritos apenas os aspectos relevantes à solução do problema.

3- Simulação de Código Fonte

Nesta etapa, procura-se simular o código em uma ferramenta confiável, a fim de verificar, preliminarmente, o cumprimento da especificação. Esta simulação não considera ainda detalhes tecnológicos de implementação.

4- Síntese Otimização e Fitting

Síntese - É o processo de "tradução" ou compilação de um código VHDL para uma descrição abstrata, em linguagem mais próxima da implementação. Naturalmente, a síntese é ainda um processo independente da tecnologia. Basicamente, o resultado obtido é o que se chama de RTL (register-transfer level), em que se definem registros, suas entradas e saídas e a lógica combinacional entre elas.

Otimização - É o processo de seleção da melhor solução de implementação para uma dada tecnologia. Logicamente, as ferramentas EDA (Engineering Design Automation) são preparadas para aceitar diretivas de otimização, dependendo do resultado que se quer (minimização de área, consumo, tempo de resposta, etc).

Fitting - É o processo em que a lógica sintezada e otimizada é mapeada nos recursos oferecidos pela tecnologia.

5- Simulação do modelo pós-layout

A simulação realizada com o resultado do fitting permite resultados mais apurados de comportamento

e timing, porque considera as características da implementação definitiva na tecnologia, como, por exemplo, os tempos de propagação dos sinais.

6- Geração

É a fase de configuração das lógicas programáveis ou de fabricação de ASICs.

Entidades e Arquiteturas

O código abaixo representa um comparador binário para palavras de quatro bits:

```
-- comparador de 4 bits
entity comp4 is
    port (  a, b: in bit-vector (3 downto ∅);
            equals: out bit);
end comp4;

architecture arq1 of comp4 is
begin
    equals  <= '1'  when (a=b) else 'O';
end arq1;
```

Uma entidade (entity) é uma abstração que descreve um sistema, uma placa, um chip, uma função ou, até mesmo, uma porta lógica. Na declaração de uma entidade, descreve-se o conjunto de entradas e saídas. No exemplo dado, a entidade comp4 possui duas entradas, a e b, e uma saída, equals, que definem o port da entidade.

Os ports correspondem a pinos e são tratados como objetos de dados. Pode-se atribuir valores ou obtê-los de ports. Cada entrada ou saída possui um modo (mode) de operação. Os modos possíveis são:

in – entrada;

out – saída: os pinos definidos como saída não podem ser utilizados como entradas, nem seus valores utilizados na lógica interna;

buffer - saída com possibilidade de realimentação;

inout - substitui qualquer um dos outros, mas seu uso deve ser limitado aos casos em que o pino deva ser utilizado como entrada e saída, para clareza na descrição.

A arquitetura (architecture) descreve as funções realizadas pela entidade. No caso do exemplo é atribuído o valor '1 ' à saída equals, sempre que as entradas forem iguais e '0', caso contrário.

Corpo de Arquitetura

A arquitetura de uma entidade pode ser descrita de três formas distintas, mas que, em geral, conduzem a uma mesma implementação.

a) Descrição comportamental

Esta é a forma mais flexível e poderosa de descrição. São definidos processos concorrentes (process). A cada processo é associada uma lista de sensibilidade, que indica quais são as variáveis cuja alteração deve levar à reavaliação da saída. No simulador funcional, quando uma variável da lista é modificada, o processo é simulado novamente. O código abaixo ilustra a aplicação deste tipo de descrição ao comparador do exemplo.

```
-- comparador de 4 bits
entity comp4 is
    port (  a, b: in bit_vector (3 downto ∅);
            equals: out bit);
end comp4;

architecture comport of comp4 is
begin
  comp: process (a,b) -- lista de sensibilidade
    begin
      if a = b then
        equals  < = '1' ;
      else
        equals < = 'O' ;
      end if;
    end process comp;
end comport;
```

Uma arquitetura pode ter mais de um processo e eles serão executados concorrentemente entre si.

b) Descrição por fluxo de dados

Neste tipo de descrição, os valores de saída são atribuídos diretamente, através de expressões lógicas. Todas as expressões são concorrentes no tempo.

```
-- comparador de 4 bits
entity comp4 is
    port (  a, b: in bit_vector (3 downto ∅);
            equals: out bit);
end comp4;

architecture fluxo of comp4 is
begin
  equals < = '1' when (a=b) else 'O';
end fluxo;
```

c) Descrição estrutural

A descrição estrutural apresenta netlists e instanciação de componentes básicos, ou seja, é como se fosse uma lista de ligações entre componentes básicos pré-definidos.

```
-- comparador de 4 bits
entity comp4 is
    port (  a, b: in bit_vector (3 downto ∅);
            equals: out bit);
end comp4;

use work.gateskg.all;
architecture estrut of comp4 is
  signal x bit_vector (0 to 3);
begin
  U∅: xnor2 port map (a(∅), b(∅), x(∅));
  U1: xnor2 port map (a(1), b(1), x(1));
  U2: xnor2 port map (a(2), b(2), x(2));
  U3: xnor2 port map (a(3), b(3), x(3));
  U4: and4 port map (x(∅), x(1), x(2), x(3), equals);
end estrut;
```

Identificadores

Os identificadores são usados como referência a todos os objetos declarados no código. As regras para formação de nomes são:

- O primeiro caracter deve ser uma letra

- O último não pode ser underscore

- Não são permitidos 2 underscores em seqüência

- Maiúscula / Minúscula são equivalentes

Objetos de dados

Constantes - Assumem apenas um valor em todo o código

Ex: constant largura: integer: = 8 ;

Podem ser declaradas no níveis de package

entity

architecture

process

e valem apenas no contexto em que são declaradas

Sinais - Representam ligações (fios) que interligam componentes. Ports são exemplos de sinais.

Ex. signal valor-de-contagem: bit-vector (3 downto ∅):

Podem ser declarados na entidade ou na arquitetura.

Variáveis - São utilizadas em processos e subprogramas e devem ser declaradas neles. São atualizadas imediatamente e não correspondem à implementação física, como no caso dos sinais.

Ex: variable resultado: std-logic := '0'

Tipos de Dados

VHDL é strongly typed: objetos de dados de tipos diferentes não podem ser atribuídos um ao outro, sem conversão explícita.

Tipos escalares

a) Enumeration Types

Lista de valores que o objeto pode assumir.

Ex: type estado is (espera, erro, cálculo, transmitido);

Pode-se então declarar um sinal do tipo:

signal estado_atual: estado;

Os dados desse tipo são ordenados, ou seja

erro > = espera

Já definidos pela norma:

```
type boolean is (FALSE, TRUE);
type bit is (‘0’, ‘1’);
type std_ulogic is (  ‘U’, -- não inicializada
                      ‘X’, -- desconhecida forte
                      ‘0’, -- 0 forte
                      ‘1’, -- 1 forte
                      ‘Z’, -- alta impedância
                      ‘W’, -- desconhecida fraca
                      ‘L’, -- 0 fraco
                      ‘H’, -- 1 fraco
                      ‘-’  -- tanto faz
                        );
subtype std_logic is resolved std_ulogic
```

resolved → Existe uma função de resolução para os casos em que existe + de 1 driver para o sinal.

NOTA: para usar std_logic e std_ulogic devem-se acrescentar as seguintes linhas antes da entidade:

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

b) Integer Types

Inteiros na faixa de valores possíveis: - (2^{31} - 1) a (2^{31} - 1)

Exemplo:

variable attura: integer range 0 to 255;

c) Flooting Types

de -1.0E 38 a 1.0 E38

pouco utilizados, por envolverem grande quantidade de recursos.

d) Physical Types

Exemplo

```
type time in range - 2147483647to2147483647
units
fs;
ps = 1000 fs;
ns = 1000 ps;
µs = 1000 ns;
ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;
```

Este tipo relaciona-se diretamente a grandezas físicas

Tipos compostos

Array

Exemplo:

```
type ward is array (15 downto ∅) of bit;
signal b: ward;
```

Record

Exemplo:

```
type iocell is record
  buffer_inp: bitvector (7 downto ∅);
  enable: bit;
  bufer_out: bit-vector (7 downto ∅);
end record;
```

Exercícios

1 - Projetar e simular:

a) Biestável tipo JK com clock a borda de subida, com descrição comportamental.

b) Idem, com Preset e Clear assíncronos

c) Um full-adder de 1 bit comportamental

2 – Realizar o projeto de um contador síncrono, com uma seqüência pré-definida em VHDL e simular.

3 - Comentar os Exemplos:

```
--1 - VIGIA
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Vigia IS
  PORT(
    clk                      : IN  STD_LOGIC;
    SensorA, SensorB         : IN  STD_LOGIC;
    Multa_Vel, Multa_Comp     : OUT STD_LOGIC);
END Vigia;
ARCHITECTURE Intuitiva OF Vigia IS
  TYPE STATE_TYPE IS (Espera,
            Verificando_Velocidade,
            Verificando_Tamanho,
            Multa_Velocidade,
            Multa_Tamanho,
            Erro);
  SIGNAL Estado: STATE_TYPE;
  SIGNAL Cronometro: INTEGER RANGE 0 to 31;
BEGIN
  PROCESS (clk)
  BEGIN

    IF clk'EVENT AND clk = '1' THEN
      Multa_Vel <= '0';
      Multa_Comp <= '0';
      CASE Estado IS
        WHEN Espera =>
          IF SensorA = '1' THEN
            IF SensorB = '0' THEN
              Estado <= Verificando_Velocidade;
```

```
            ELSE
              Estado <= Erro;
            END IF;
          END IF;

        WHEN Verificando_Velocidade =>
          IF SensorA = '1' THEN
            IF SensorB = '0' THEN
              Cronometro <= Cronometro + 1;
            ELSE
              IF Cronometro < 8 OR Cronometro > 24 THEN
                Estado <= Multa_Velocidade;
              ELSE
                Estado <= Verificando_Tamanho;
              END IF;
            END IF;
          ELSE
            IF SensorB = '0' THEN
              Estado <= Espera;
              Cronometro <= 0;
            ELSE
              Estado <= Erro;
            END IF;
          END IF;

        WHEN Verificando_Tamanho =>
          IF SensorA = '1' THEN
            IF SensorB = '1' THEN
              Cronometro <= Cronometro -1;
              IF Cronometro = 0 THEN
                Estado <= Multa_Tamanho;
              END IF;
            ELSE
              Estado <= Erro;
            END IF;
          ELSE
            Estado <= Espera;
            Cronometro <= 0;
          END IF;
        WHEN Multa_Velocidade =>
          Multa_Vel <= '1';
        WHEN Multa_Tamanho =>
          Multa_Comp <= '1';
        WHEN Erro =>
          Multa_Vel <= '1';
          Multa_Comp <= '1';
      END CASE;
    END IF;
  END PROCESS;
END Intuitiva;


--2 - RECEPTOR
```

```
-- Receptor serial
entity Receptor is
  port( data_in, clock, limpa: in bit;
    pronto, ocupado: out bit;
    data_out: buffer bit_vector(7 downto 0));
end Receptor

architecture Receptor of Receptor is
  TYPE STATE_TYPE IS (  Espera,
          Start_bit,
          Recebendo,
          Pronto);
  SIGNAL Estado: STATE_TYPE;
  SIGNAL Cronometro: INTEGER RANGE 0 to 7;
  SIGNAL   Conta_bits: INTEGER RANGE 0 to 7;
  process
  begin
    if clock'event and clock = '1' then
    CASE Estado IS
        WHEN Espera =>
          IF Data_in = '0'then
            Cronometro <= 0;
            Ocupado <= '1';
            Estado <= Start_bit;
          ELSE
            Ocupado <= '0';
          END IF;

        WHEN Start_bit =>
          IF Cronometro < 4 then
            Cronometro = Cronometro+1;
          ELSE
            Cronômetro = 0;
            Conta_bits =0;
            Estado <= Recebendo;
          END IF;
        WHEN Recebendo =>
          IF Conta_bits < 6 then
            IF Cronometro < 3 then
              Cronometro = Cronometro+1;
            ELSE
              Cronômetro = 0;
              Conta_bits = Conta_bits+1;
              Data_out(0)<= Data_out(1);
              Data_out(1)<= Data_out(2);
              Data_out(2)<= Data_out(3);
              Data_out(3)<= Data_out(4);
              Data_out(4)<= Data_out(5);
              Data_out(5)<= Data_out(6);
              Data_out(6)<= Data_out(7);
              Data_out(7)<= Data_in;
            END IF;
          ELSE
            Estado <= Pronto;
```

```
            Pronto <= '1';
          END IF;
        WHEN Pronto =>
          IF Limpa = 0 Then
            Ocupado <= '0';
            Pronto <= '0';
            Estado <= Espera;
          END IF;
    END CASE;
    END IF;
  end process;
end Receptor;
```

Referências

[1] Mazor, S. Langstraat, P. A Guide to VHDL. Kluwer Academic Publishers, Boston, 1993.

[2] Scarpino, F. VHDL and AHDL Digital System Implementation. Prentice Hall, New Jersey, 1998.

[3] Skahill, K. VHDL for Programmable Logic. Addison Wesley, Reading, Massachusetts, 1996.