UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO MULTIDICIPLINAR PAU DOS FERROS
CURSO DE CIÊNCIA E TECNOLOGIA

JANAILSON MACIEL DE QUEIROZ

ANÁLISE COMPARATIVA ENTRE LINGUAGENS DE DESCRIÇÃO DE *HARDWARE*: VERILOG E VHDL

PAU DOS FERROS
2016

JANAILSON MACIEL DE QUEIROZ

ANÁLISE COMPARATIVA ENTRE LINGUAGENS DE DESCRIÇÃO DE *HARDWARE*: VERILOG E VHDL

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Ciência e Tecnologia.

Orientador: Ernano Arrais Junior, Prof. Dr.

PAU DOS FERROS

2016

Q3a Queiroz., Janailson Maciel de.
ANÁLISE COMPARATIVA ENTRE LINGUAGENS DE DESCRIÇÃO DE HARDWARE: VERILOG E VHDL / Janailson Maciel de Queiroz.. - 2016.
80 f. : il.

Orientador: Ernano Arrais Junior.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de , 2016.

1. Linguagem. 2. Hardware. 3. Verilog. 4. VHDL. 5. FPGA. I. Arrais Junior, Ernano , orient. II. Título.

JANAILSON MACIEL DE QUEIROZ

ANÁLISE COMPARATIVA ENTRE LINGUAGENS DE DESCRIÇÃO DE HARDWARE: VERILOG E VHDL

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Ciência e Tecnologia.

Defendida em: 08 / 11 / 2016

BANCA EXAMINADORA

Prof. Dr. Ernano Arrais Junior
Presidente

Prof. Me. Francisco Carlos Gurgel da Silva Segundo
Membro Examinador

Prof. Ma. Verônica Maria Lima Silva
Membro Examinador

RESUMO

Ao longo dos anos os chips foram tornaram-se cada vez mais complexos, aumentando também a complexidade dos projetos eletrônicos. Dessa forma, as antigas ferramentas utilizadas pelos projetistas foram se tornando insuficientes, assim, as linguagens de descrição de *hardware* surgiram como uma solução, pois, de uma forma geral, propiciam o desenvolvimento de projetos independente da tecnologia que será empregada, além de facilitar a realização de testes e a atualização de projetos, reduzindo custos e tempo com a implementação. Existem diversas linguagens de descrição de *hardware*, entre elas VHDL (*VHSIC Hardware Description Language*), Verilog, System C, System Verilog, entre outras, merecendo destaque as linguagens VHDL e Verilog que são as mais difundidas. Também existem diversos trabalhos e discussões sobre qual das linguagens apresentam os melhores resultados de implementação, contudo, não existe um consenso sobre essa discussão. Este trabalho se propôs a análise das linguagens Verilog e VHDL, com o intuito de comparar quanto ao desempenho e otimização do hardware. Então, foi realizado um levantamento bibliográfico onde as linguagens de descrição de hardware, Verilog e VHDL, foram comparadas, verificando-se qual a melhor, sob a ótica de vários autores. Em seguida, foi realizada uma pesquisa de todas as características relevantes da sintaxe de Verilog e VHDL. Finalmente, foram implementados sistemas digitais básicos na plataforma FPGA (*Field Programmable Gate Array*), em ambas as linguagens, usando a ferramenta de síntese e simulação de linguagens de descrição de *hardware*, Quartus Prime, onde foi analisado tempo de compilação, tamanho de código, síntese, otimização, atraso de sinal, etc. Os resultados mostraram que a escolha de uma HDL para um projeto pode depender de outros fatores além da capacidade delas de descrever *hardwere* (como domínio de uma HDL ou preferencia, do projetista, por exemplo), já que se mostraram equivalentes em quase todos os critérios de comparação.

Palavras-chave: Linguagem, *Hardware*, Verilog, VHDL, FPGA.

ABSTRACT

Over the years the chips are becoming more complex, also increasing the complexity of the electronic design. Thus the old tools used by designers were becoming insufficient, so in this scenario the hardware description languages (HDL) have emerged as a solution because, in general, provide for the development of independent technology projects that will be used, besides facilitating the testing and updating projects, reducing costs and time to implementation. There are several hardware description languages, including VHDL (VHSIC Hardware Description Language), Verilog, System C, System Verilog, among others, with emphasis the VHDL and Verilog languages that are the most widespread. There are also several papers and discussions about which languages have the best results of implementation, however, there is no consensus on this discussion. This work proposes the analysis of Verilog and VHDL languages, in order to compare how the performance and optimization of the hardware. So it was performed a bibliographic reference survey where the hardware description languages, Verilog and VHDL were compared, verifying what the best from the perspective of various authors. Then he was made a survey of all relevant features of Verilog and VHDL syntax. Finally, basic digital systems have been implemented in the FPGA platform (Field Programmable Gate Array), in both languages, using the synthesis tool and simulation of hardware description languages, Quartus Prime where compile time was analyzed, code size, synthesis, optimization, signal delay, among others. The results showed that the choice of HDL for a design may depend on other factors besides their capacity to describe hardware (such as domain of an HDL or preferably designer, for example), since it showed equivalent in almost all criteria comparison. In other words, it is an issue of trade-off.

Keywords: Hardware, Languages, Verilog, VHDL, FPGA.

LISTA DE FIGURAS

Figura 1 – Porta AND.....52
Figura 2 – Porta OR.....52
Figura 3 – Porta NOT.....53
Figura 4 – Porta NAND.....54
Figura 5 – Porta NOR.....54
Figura 6 – XOR 1.....55
Figura 7 – Porta XOR.....55
Figura 8 – Porta XNOR.....56
Figura 9 – Máquina de Estados Fínitos.....58

LISTA DE QUADROS

Quadro 1 – Resumo do referencial Bibliográfico 21

Quadro 2 – Estrutura Básica de um Código em VHDL.....22

Quadro 3 – Operadores de VHDL.....28

Quadro 4 – Valores em Verilog.....39

Quadro 5 – Pesos para os valores em Verilog.....39

Quadro 6 – Operadores de Verilog.....40

Quadro 7 – Tabela Verdade AND.....52

Quadro 8 – Tabela Verdade OR.....52

Quadro 9 – Tabela Verdade NOT.....53

Quadro 10 – Tabela Verdade NAND.....54

Quadro 11 – Tabela Verdade NOR.....54

Quadro 12 – Tabela Verdade XOR.....55

Quadro 13 – Tabela Verdade XNOR.....56

LISTA DE TABELAS

Tabela 1 Resultados do Código da Porta AND….....…….....….....52

Tabela 2 Resultados do Código da Porta OR.....…….…….....53

Tabela 3 Resultados do Código da Porta NOT……..…….....53

Tabela 4 Resultados do Código da Porta NAND.....…….....54

Tabela 5 Resultados do Código da Porta NOR.....….....54

Tabela 6 Resultados do Código da Porta XOR.....….....55

Tabela 7 Resultados do Código da Porta XNOR.....56

Tabela 8 Resultados do Código de um Somador Completo.....57

Tabela 9 Resultados do Código de um Registrador.....….....57

Tabela 10 Resultados do Código de uma Memória ROM 8x8.....…...58

Tabela 11 Resultados do Código de uma Memória RAM 16x8.....….58

Tabela 12 Resultados do Código do Temporizador de um Laser.....….....59

LISTA DE ABREVIATURAS E SIGLAS

FPGA *Field-Programmable Gate Array* .

HDL *Hardware Description Language* (Linguagem de descrição de Hardware).

RAM *Random Access Memory*

ROM *Read-Only Memory*

VHDL *VHSIC Hardware Description Language*.

VHSIC *Very High Speed Itegrated Circuits*.

SUMÁRIO

1 INTRODUÇÃO 13

1.1 RELEVÂNCIA DO TEMA 13

1.2 MOTIVAÇÃO 16

1.3 OBJETIVOS 16

1.3.1 Objetivo Geral 16

1.3.2 Objetivos Específicos 16

1.3 METODOLOGIA 16

1.4 ORGANIZAÇÃO 17

2 ESTADO DA ARTE 19

3 VHDL 22

3.1 ESTRUTURA BÁSICA VHDL 22

3.1.1 Declaração de Bibliotecas e Pacotes 23

3.1.2 Entidade 24

3.2 TIPOS DE DADOS 25

3.3 CLASSES DE OBJETOS 27

3.3.1 Constant 27

3.3.2 Signal 27

3.3.3 Variable 28

3.3.4 File 28

3.4 OPERADORES 28

3.5 CÓDIGO CONCORRENTE 29

3.5.1 When 30

3.5.2 Select 30

3.5.3 Generate 31

3.6 CÓDIGO SEQUENCIAL 31

3.6.1 Process 31

3.6.2 If 32

3.6.3 Case 32

3.6.4 Loop 33

3.7 OUTROS COMANDOS IMPORTANTES 34

3.7.1 Package 34

3.7.2 Component 35

3.7.3 Function 36

3.7.4 Procedure 36

4 VERILOG 38

4.1 MÓDULOS 38

4.2 CONJUNTO DE VALORES 39

4.3 OPERADORES 40

4.4 NETS 40

4.5 VARIÁVEIS 41

4.5.1 Reg 41

4.5.2 Integer 41

4.6 ESPECIFICAÇÃO DE NÚMEROS 42

4.7 PARAMETERS 42

4.8 VETORES E ARRAYS 42

4.9 CÓDIGO CONCORRENTE 43

4.9.1 Instanciação de Portas Lógicas Primitivas 43

4.9.2 Atribuições contínuas 43

4.10 CÓDIGO SEQUENCIAL 44

4.10.1 Always 44

4.10.2 Initial 45

4.10.3 Declaração de If-Else 45

4.10.4 Declaração de Case 46

4.10.5 Declaração de Laços 46

4.10.6 Atribuições Blocking e Non-blocking 47

4.11 INSTANCIAÇÃO DE MODULOS 48

4.12 TASKS E FUNCTIONS..... 49

5 RESULTADOS E DISCUSSÕES 51

7 CONCLUSÃO.....60

REFERÊNCIAS 61

APÊNDICE A – CÓDIGO DE UMA PORTA AND EM VERILOG E EM VHDL 63

APÊNDICE B – CÓDIGO DE UMA PORTA OR EM VERILOG E EM VHDL 64

APÊNDICE C – CÓDIGO DE UMA PORTA NOT EM VERILOG E EM VHDL 65

APÊNDICE D – CÓDIGO DE UMA PORTA NAND EM VERILOG E EM VHDL 66

APÊNDICE E – CÓDIGO DE UMA PORTA NOR EM VERILOG E EM VHDL 67

APÊNDICE F – CÓDIGO DE UMA PORTA XOR EM VERILOG E EM VHDL 68

APÊNDICE G – CÓDIGO DE UMA PORTA XNOR EM VERILOG E EM VHDL 69

APÊNDICE H – CÓDIGO DE UM SOMADOR COMPLETO EM VERILOG E EM VHDL 70

APÊNDICE I – CÓDIGO DE UM REGISTRADOR DE 4 BITS EM VERILOG E EM VHDL 71

APÊNDICE J – CÓDIGO DE UMA MEMÓRIA ROM DE 8 PALAVRAS, 8 BITS CADA, EM VERILOG E EM VHDL 72

APÊNDICE L – CÓDIGO DE UMA MEMÓRIA RAM DE 16 PALAVRAS, 8 BITS CADA, EM VERILOG E EM VHDL 74

ANEXO A – CÓDIGO DA MAQUINA DE ESTADOS DO TEMPORIZADOR DE UM LASER, EM VERILOG E EM VHDL 75

1 INTRODUÇÃO

1.1 RELEVÂNCIA DO TEMA

Os circuitos eletrônicos se dividem em dois tipos: circuitos digitais e circuitos analógicos. A principal diferença entre eles se dá pelo tipo de sinais que eles processam, circuitos digitais processam sinais digitais, circuitos analógicos processam sinais analógicos. Sinais digitais são aqueles cujas informações são representadas em valores discretos, diferentemente dos sinais analógicos, cujas informações são dadas em forma contínua, apresentando infinitas possiblidades de representação (PEDRONI, 2010; TOCCI, 2011).

As principais vantagens de se usar as técnicas digitais são (TOCCI, 2011):

- A facilidade de projetar sistemas digitais e a praticidade no armazenamento de informações;
- Maior exatidão e precisão em tais circuitos, a operação do sistema pode ser programada;
- Redução quanto aos riscos de circuitos digitais serem afetados por ruídos;
- Uma quantidade superior de circuitos digitais pode ser inserida em um circuito integrado.

Por outro lado, tem como desafio o mundo ser em sua maioria analógico, o que faz com que os circuitos digitais usem conversores de sinais analógico-digitais e digitais-analógicos em suas entradas e saídas (TOCCI, 2011). Apesar desse desafio, os sistemas digitais são bastante utilizados atualmente, e existem vários tipos de métodos utilizadas em seus projetos, um importante método é com o uso de linguagens de descrição de *Hardware*.

Linguagem de Descrição de *Hardware* (*Hardware Description Language* - HDL) serve para modelar a estrutura e o comportamento de um hardware. As HDLs facilitam na descrição do *hardware* em dois importantes aspectos: o verdadeiro comportamento abstrato, em que as HDLs possuem estrutura para lhe dar suporte, podendo descrever o seu comportamento em vários níveis de abstração; o outro aspecto

é a sua estrutura, pois é possível modelar uma estrutura de um hardware independente do comportamento do circuito (ORDONEZ et al, 2003).

Além de descrever o funcionamento de um circuito que se deseja projetar, as HDLs são usadas para testar em um simulador se tal descrição alcança o funcionamento esperado no circuito, e tal código descritivo em uma HDL, consequentemente, pode ser sintetizado em um sintetizador lógico para configurar um dispositivo programável com o funcionamento desejado (R.UMA E R.SHARMILA, 2011).

Um código em HDL pode ser escrito em 3 estilos (GROUT, 2008):

- **Descrição estrutural**: descreve a estrutura de circuitos em termos de portas lógicas utilizadas e da fiação das interconexões entre as portas lógicas para formar uma lista de conexões do circuito.
- **Descrição de fluxo de dados**: descreve a transferência de dados de entrada para a saída, e entre os sinais.
- **Descrição comportamental**: descreve o comportamento do projeto em termos de circuitos ou sistema comportamental utilizando algoritmos. Esta descrição de alto nível usa construções de linguagem que se assemelham a uma linguagem de programação de *software* de alto nível.

Os níveis de abstração das descrições das HDLs são divididos em (OLMEDO E MARCALLA, 2014):

- Nível de interruptores (*switch level*): É utilizado para descrever o circuito em função de transistores e fios.
- Nível de porta (*gate level*): Descreve o circuito em função de portas lógicas e elementos de armazenamento, como flip-flops.
- Nível de fluxo de dados: Descreve o circuito em função do fluxo de dados entre registradores.
- Nível algoritmo: É semelhante a um programa em linguagem de alto nível como C. Possui instruções de alto nível como laços, decisões e outros.

As HDLs possuem, além do que já foi citado, as seguintes características: permitem que os módulos descrevam as ações que serão avaliadas sequencialmente (processual), sendo que cada modulo é visto como mais uma ação concorrente; a construção de uma estrutura hierárquica, em que é possível combinar as descrições

estruturais e de fluxo de dados com descrições de comportamento; modelar o conceito de “timing”, fundamental para a descrição de sistemas eletrônicos (OLMEDO E MARCALLA, 2014).

HDLs proporcionam as seguintes vantagens (RIESGO; TORROJA; LA TORRE, 1999):

- Permitem uma menor fase de desenvolvimento de projetos;
- Promovem uma continua checagem e verificação do desempenho do sistema e comportamento;
- Podem tornar o sistema independente da tecnologia final utilizada nas fases iniciais de implementação;
- Promovem uma interface comum a diferentes usuários, pessoas envolvidas em um projeto e também a diferentes designs e ferramentas de auxilio de design (CAD).

De acordo com Pedroni (2010) no projeto de sistemas digitais modernos de grande porte geralmente se usa uma HDL: Verilog ou VHDL. R. Uma e R. Sharmila (2011) confirmam que só essas duas HDLs são amplamente utilizadas e padronizadas, o Vahid(2008) também se refere a elas como as mais populares junto a SystemC. Consequentemente, focaremos o estudo nessas duas linguagens.

A linguagem VHDL (*VHSIC Hardware Description Language*, onde VHSIC quer dizer *Very High Speed Itegrated Circuits*) teve sua criação financiada pelo departamento de defesa dos Estados Unidos em 1980. A linguagem inicialmente era destinada a documentar o comportamento do *hardware* digital. Ela foi desenvolvida a partir das linguagens de programação ADA e Pascal. Passou por vários processos de incrementos ao longo dos anos até se tornar padrão 1076 do IEEE em 1987. Novas atualizações e melhorias desse padrão de 1076 IEEE foram criadas e novas continuam a serem desenvolvidas (PEDRONI, 2010; R.UMA E R.SHARMILA, 2011).

Já o Verilog, foi desenvolvido por Gateway Design Automation em 1983 como uma linguagem de modelagem de *hardware* para seus produtos simuladores. Foi desenvolvida a partir da linguagem C. A Cadence comprou o Verilog da Gateway Design Automation em 1989, então a linguagem ganhou muita popularidade. Em 1990 se tornou domínio público, desde então sofreu várias melhorias e em 1995 foi padronizada por IEEE como padrão IEEE 1364-1995. Depois disso, já houve diversos outros padrões e modificações do Verilog (R.UMA E R.SHARMILA, 2011).

1.2 MOTIVAÇÃO

Esse trabalho é motivado, primeiramente, pela relevância do tema na nossa sociedade, visto a grande utilização de linguagens de descrição de *hardware* para projetar novas tecnologias. Desse modo é necessário o estudo e comparação dessas linguagens para melhor utilizá-las, conseguindo maior eficiência e eficácia na implementação de projetos, sejam estes simples ou complexos. Contribuindo assim para comunidade especifica.

Em segundo lugar, essa pesquisa se faz importante também pela pouca quantidade de trabalhos científicos voltados para este tema no meio acadêmico nacional, dado que há pouca referência em português para embasamento teórico.

Dar-se também pela importância do conhecimento sobre HDLs em formações como engenharia elétrica e engenharia da computação.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

- Comparar as linguagens de descrição de *hardware* Verilog e VHDL.

1.3.2 Objetivos Específicos

- Realizar um estudo bibliográfico sobre as HDLs com foco em Verilog e VHDL;
- Estudar variáveis de referência para se comparar as duas linguagens;
- Realizar uma comparação qualitativa entre as HDLs, com base em testes práticos.

1.4 METODOLOGIA

O processo de produção dessa pesquisa foi divido em 3 etapas :

1. Iniciou-se com um levantamento de referencial bibliográfico onde as linguagens de descrição de hardware, Verilog e VHDL, foram comparadas, verificando-se qual a melhor, sob a ótica de vários autores.
2. Nessa etapa fez-se uma pesquisa de todas as características relevantes de Verilog e VHDL, como: declarações, tipos de dados suportados, tipos de descrição, sintaxe, entre outras. Logo após, foi analisado em quais dessas características uma HDL se sobressai em relação a outra.
3. Foram desenvolvidos códigos de circuitos em ambas as HDLs de portas lógicas (OR, AND, NOT, NAND, NOR, XOR XNOR), de memorias (ROM com 8 palavras de 8 bits cada, RAM com 16 palavras de 8 bits cada, e de um registrador de 4 bits), de um somador completo, e de uma maquina de estados para um temporizador de um laser. Para fazer as descrições, foi utilizada a ferramenta de síntese e simulação de sistemas digitais Quartus Prime 15.1, no qual foi analisado o tempo de simulação, tempo de compilação, tamanho de código, síntese, otimização, etc. O chip escolhido para a analise no Quartus Prime foi um 5CGXFC7C7CF23C8 da família Ciclone V.

1.5 ORGANIZAÇÃO

Este trabalho está organizado em 6 capítulos:

- O **Capítulo 1** abordou os objetivos e motivações para a elaboração deste trabalho, bem como uma pequena introdução e contextualização referente às linguagens de descrição de *hardware.*
- No **Capítulo** 2 será apresentado o estado da arte, o qual trata dos trabalhos de diversos autores em comparar HDLs, quais os resultados que eles obtiveram, seus critérios utilizados, o que faltou em seus trabalhos e etc.
- O **Capítulo 3** traz a fundamentação teórica acerca do VHDL**,** onde será abordada as principais características da sintaxe de tal HDL como: estrutura básica, tipos de objetos, tipos de dados, declarações de código concorrente, declarações de código sequencial, etc. Apresentando modelos de vários tipos de declarações de código.

- O **Capitulo** 4 traz a fundamentação teórica acerca do VERILOG, é semelhante ao capitulo 3, porém apresenta as principais característica da sintaxe de Verilog ao invés de VHDL.
- No **Capitulo 5** são apresentados os resultados e discursão. Obtidos a partir de testes comparativos.
- No **Capítulo 6** são apresentadas as conclusões gerais do trabalho, bem como perspectivas futuras.

2 ESTADO DA ARTE

Dada à importância das HDLs e de seu estudo comparativo, algumas pessoas já fizeram trabalhos se propondo a tal estudo e utilizando para isso diversos tipos de metodologias e critérios de comparação diferentes, dessa forma alcançando resultados variados. Aqui segue uma análise do que já foi feito referente ao tema, e servirá como base para o restante do trabalho, além de levantar hipóteses que poderão ser provadas ou refutadas sobre qual a melhor HDL neste trabalho.

Maginot (1992) comparou VHDL com outras 3 HDLs bastante conhecidas, Verilog, UDL/I (novo padrão japonês), e M (da Mentor Graphics). A comparação tinha dois principais interesses, um deles é que poderia ajudar um projetista a aprender VHDL a partir de uma dessas outras três linguagens com os conceitos que ele já está familiarizado, o outro interesse seria destacar as características fundamentais de VHDL demostrando suas faltas e vantagens em relação às outras linguagens.

Para essa comparação, Maginot (1992) usou vários parâmetros, abordando as características principais de cada linguagem paralelamente. Como resultado de sua pesquisa, Maginot (1992) concluiu que é possível descrever um hardware em qualquer uma das linguagens, pois apesar das diferenças sempre há uma forma alternativa para se alcançar o projeto desejado; e que quanto ao poder de modelagem o VHDL é menos eficiente em baixos níveis de descrição (analógico interruptor e porta), porém, mais eficiente em altos níveis de descrição (comportamental funcional). Além disso, verificou também que as principais diferenças entre as linguagens são metodológicas.

Coumeri e Thomas (1994) desenvolveram um conjunto de benchmarks (pode ser compreendido como método de comparação) para comparar a velocidade de simuladores de Verilog e VHDL. Eles serviriam tanto para comparar simuladores da mesma HDL, como para comparar o desempenho do simulador de um código HDL traduzido em relação ao simulado da outra HDL. Cada benchmark foi dividido em categorias de forma a avaliar o desempenho dos simuladores em vários aspectos. Como conclusão verificou-se que comparando simuladores de alto desempenho a descrição em VHDL gastou entre 1,93 a 46,62 vezes mais tempo na simulação.

Smith (1996) comparou Verilog e VHDL, demostrando suas características semelhantes e diferentes, além disso, como exemplo fez um modelo que calcula o Máximo Divisor Comum (MDC) entre dois números, esse modelo foi implementado em

nível de algoritmo em Verilog, VHDL e C para comparação. O modelo do MDC também foi implementado em nível RTL (nível de fluxo de dados entre registradores) em VHDL e Verilog e foi comparado. Ele concluiu que é possível se modelar um *hardware* em qualquer uma das linguagens e que a escolha de HDL consequentemente não é baseada na capacidade técnica, mas em: preferências pessoais, disponibilidade de ferramentas EDA e comercial, de negócios e questões de *marketing*.

Bailey (2005) compara Verilog, SystemVerilog e VHDL, analisando as vantagens e desvantagens de forte tipagem e várias outras características das linguagens. Ele questiona se com a recente popularidade de SystemVerilog vale a pena um projetista trocar outra linguagem por essa, pois em suas vantagens, complementa algumas falhas do Verilog comum e em certas partes não é tipada quanto VHDL, além de apresentar as características de Verilog que faltam em VHDL. Entretanto, afirma que SystemVerilog ainda não é um padrão e assim não há tantas ferramentas de suporte, entre outras vantagens e desvantagens dessa troca.

Observando o grande progresso no campo dos eletrônicos, havendo mudanças nas estruturas do projeto e nos princípios dos eletrônicos, e sabendo que no processo dos projetos as estruturas são definidas através de HDLs, R. Uma e R. Sharmila(2011) notaram a importância de se comparar as duas HDLs mais utilizadas, VHDL e Verilog, que possuem grandes diferenças em vários aspectos como, conteúdo, estrutura, reutilização, portabilidade, custo, entre outros, e essas diferenças de acordo com R. Uma e R. Sharmila (2011) podem gerar problemas de implementação. A análise comparativa também contribuiria para enquadrar a investigação futura no domínio da eletrônica. Consequentemente o estudo delas também determinaria a superioridade de uma linguagem sobre a outra.

Elas concluíram que Verilog é mais simples de se usar, logo uma vantagem para um projetista iniciante, melhor para modelar no nível de transistor e porta, menor ocupação de memória na simulação e maior velocidade gerando melhor desempenho, sua analise de tempo também supera VHDL, porém VHDL é melhor para modelar em altos níveis de abstração, além de ter o conceito de capacidade de reutilização, onde um projeto pode ser reutilizado em outro como se fosse uma função da biblioteca da linguagem.

Dada a importância das HDLs e a necessidade de seu ensino em cursos universitários ligados à área elétrica e digital, Olmedo e Marcalla (2014) fizeram um

estudo e comparação das linguagens VHDL e Verilog com objetivo de criar um guia nessa área e determinar a melhor linguagem com foco principal no critério de qual é mais fácil o aprendizado para um iniciante. As HDLs foram comparadas em vários parâmetros de sintaxe como: codificação de linha, simulação, bibliotecas, tipos de dados, sensibilidade e preferência de idioma; essa preferencia se refere aos autores de vários artigos que eles analisaram. Nesse processo vários códigos foram comparados. Concluíram que Verilog era melhor dada a facilidade para se aprender e o menor tamanho dos códigos, entre outros fatores.

Quadro 1: Resumo do referencial Bibliográfico

Autores	**Fez Simulações**	**Comparou códigos**	**Fez Analise de Tempo**	**Fez Síntese de Circuitos**	**Comparou a sintaxe**
Maginot (1992)					✓
Coumeri e Thomas(1994)	✓				
Smith (1996)		✓			✓
Bailey (2005)					✓
R.Uma e R.Sharmila(2011)					✓
Olmedo e Marcalla (2014)		✓			✓

Fonte: Próprio autor.

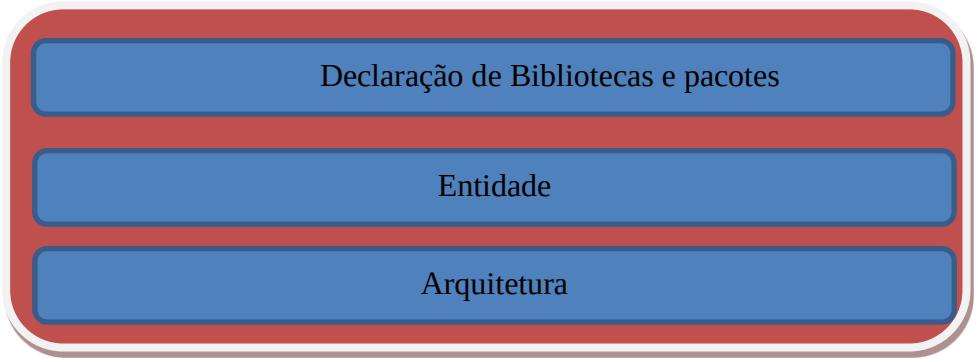
3 VHDL

A linguagem VHDL não é *case sensitive*, isso significa que a linguagem não diferencia letras maiúsculas de minúsculas, então aqui será adotada uma prática do Pedroni (2010) de apresentar as palavras reservadas da linguagem geralmente em maiúsculo. Em códigos de VHDL tudo que vem após "--" em uma linha é comentários sobre o código que não interagem com ele, isso foi utilizado para comentar os códigos. É de relevância relatar que a abordagem aqui buscará ser a mais direta possível com o objetivo de apresentar as características gerais de VHDL, sem grande aprofundamento, para dar base aos códigos que serão apresentados e dar uma noção do poder dessa HDL, para maior aprofundamento é aconselhável visitar as referências.

3.1 ESTRUTURA BÁSICA VHDL

Os elementos básicos de uma descrição em VHDL são: declaração de bibliotecas (algumas bibliotecas padrão são declaradas automaticamente em todo código, ou seja, estão implícitas e por isso não é necessário declara-las), entidade e arquitetura. Nos blocos abaixo está demostrado o formato básico de um código. Inicialmente se declara as bibliotecas de dados que serão utilizados no código, depois se inicia a entidade seguida pela arquitetura (PEDRONI, 2010).

Quadro 2: Estrutura Básica de um Código em VHDL.



Fonte: Adaptado do Pedroni (2004).

3.1.1 **Declaração de Bibliotecas e Pacotes**

A biblioteca *work e std* não precisam ser declaradas, porém, para se declarar outros tipos de bibliotecas e pacotes, primeiramente se declara o nome da biblioteca e depois o nome do pacote presente na biblioteca que se quer usar, com a seguinte sintaxe (PEDRONI, 2010):

```
LIBRARY nome_da_biblioteca;
USE  nome_da_biblioteca . nome_do_pacote . partes_do_pacote;
```

As principais bibliotecas com seus pacotes mais utilizados são descritos a seguir (GROUT, 2008; PEDRONI, 2010; PEDRONI, 2004):

Biblioteca *std* :

- Pacote standard: Predefine tipos de dados (BOOLEAN, BIT, BIT_VECTOR, INTEGER, NATURAL, POSITIVE, REAL, TIME, DELAY_LENGTH, etc) e funções básicas de VHDL.
- Pacote *textio*: Define operações com textos e arquivos.

Biblioteca IEEE :

- Pacote *std_logic_1164*: Possibilita o uso do tipo de dados STD_ULOGIC (com 9 tipos de valores) e seu subtipo STD_LOGIC (com 8 tipos de valores) , além de disponibilizar operadores lógicos e funções de conversão de tipos de dados.
- Pacote *numeric_bit*: Inclui vários tipos de operadores e define UNSIGNED e SIGNED tendo BIT como tipo base.
- Pacote *numeric_std*: Semelhante a *numeric_bit* porém com STD_LOGIC como tipo base.
- Pacote *std_logic_signed*: Define operadores para o uso de STD_LOGIC_VECTOR, operando como SIGNED.
- Pacote *std_logic_unsigned*: Define operadores para o uso de STD_LOGIC_VECTOR, operando como UNSIGNED.
- Pacote *std_logic_arith*: Define SIGNED e UNSIGNED tendo STD_LOGIC como tipo base, e vários operadores e funções que lidam com esses tipos de dados.

Biblioteca *work*: Ela tem a função de indicar onde estão armazenados os arquivos dos projetos.

3.1.2 **Entidade**

A entidade é responsável pela interface do projeto com o ambiente exterior. Seus dois principais componentes são GENERIC, onde pode introduzir informações estáticas no projeto, e PORT, onde se designa o modo e o tipo das suas portas de entrada e saída. Em todo projeto é necessário à declaração de PORT para síntese, GENERIC não é obrigatório, e quando está geralmente é designado antes de PORT. A entidade se inicia com a palavra reservada ENTITY e possui o seguinte modelo básico (D'AMORE, 2012; PEDRONI, 2010):

```
ENTITY nome_da_entidade IS
 GENERIC    (nome_de_constante    :    tipo_de_dado    :=
valor_do_dado;
                      -- Outras declarações se necessário
           )
PORT (nome_de_portas : modo_de_portas  tipo_de_dado;
                -- Oustras portas se necessário
              );
END nome_da_entidade;
```

O modo das portas representa a sua direção de transferência, existem 5 tipos (GROUT, 2008; D'AMORE, 2012):

IN: É usado para definir portas como unidirecionais de entrada.

OUT: É usado para definir portas como unidirecionais de saída.

INOUT: Define portas bidirecionais de entrada e saída.

BUFFER: Define portas de saída que podem ser referenciadas internamente pela arquitetura.

LINKAGE: Define portas que podem ser lidas e atualizadas.

3.1.3 Arquitetura

Na arquitetura, são definidas as interconexões internas do projeto com as entradas e saídas, assim como também determina o seu funcionamento desejado. A arquitetura possui duas partes, a primeira delas é onde é feito declarações de vários tipos como: sinais, constantes, componentes referenciados, corpo de subprogramas e tipos de dados locais. Na outra parte se descreve os comandos concorrentes ou sequenciais, o código naturalmente é concorrente, para criá-lo de forma sequencial é preciso invocar alguns comandos específicos que criam uma região sequencial que é concorrente com as outras regiões do código. O modelo básico da arquitetura está definido abaixo (D'AMORE, 2012; PEDRONI, 2010):

```
ARCHITECTURE nome_da_arquitetura OF nome_da_entidade IS
-- Parte onde é feita as declarações
BEGIN
-- Parte onde se descreve vários tipos de comandos
END;
```

3.2 TIPOS DE DADOS

Os tipos de dados escalares predefinidos no pacote padrão em VHDL são (D'AMORE, 2012; GROUT, 2008):

- BIT: Pode assumir os valores 1 e 0.
- BOOLEAN: pode assumir os valores verdadeiro e falso.
- CHARACTER: Pode assumir os valores dos caracteres do código ASCII estendida como: a, b, c, d, e, etc.
- INTEGER: Pode assumir valores inteiros entre $-2^{31} - 1$ *e* $2^{31} - 1$.
- NATURAL: Pode assumir valores entre 0 e $2^{31} - 1$.
- POSITIVE: Pode assumir valores entre 1 e $2^{31} - 1$.
- REAL: Pode assumir valores reais entre $-3{,}65 * 10^{47}$ e $3{,}65 * 10^{47}$, geralmente não é suportado pelas ferramentas de síntese.
- TIME: Pode assumir alguns valores de tempo, não é sintetizável.

Outro tipo de dado escalar sintetizável e que é considerado o padrão nas indústrias é o STD_LOGIC, ele pode assumir os seguintes valores: ‘U’(não inicializado), ‘X’ (desconhecido), ‘0’ (nível baixo), ‘1’ (nível alto), ‘Z’ (alta impedância), ‘W’ (desconhecido fraco), ‘L’ (nível baixo fraco), ‘H’ (nível alto fraco), ‘-‘ (“don’t care”) (PEDRONI, 2010).

Além dos escalares também há os tipos compostos (ARRAY), que podem ser vetores e matrizes dos valores escalares, no pacote padrão há dois tipos (D’AMORE, 2012; PEDRONI, 2004):

- BIT_VECTOR: formado por vetores com elementos do tipo BIT.
- STRING: define conjuntos de CARACTER.

Outros tipos sintetizáveis compostos são: STD_(U)LOGIC_VECTOR, UNSIGNED e SINGNED (PEDRONI, 2010).

Para passar valores de um objeto para outro é necessário que sejam do mesmo tipo de dados ou tenham o mesmo de tipo de dado base, por exemplo, um objeto cujo de tipo de dados é BIT_VECTOR poderá receber um valor de um objeto cujo tipo é BIT em uma de suas posições (D’AMORE, 2012).

A linguagem VHDL também permite a criação de novos tipos de dados que podem ser físicos, enumerados ou compostos, a sintaxe para criar um tipo é basicamente usar a palavra reservada TYPE, escrever o nome do tipo e então definir o seu intervalo de valores ou estados. Por exemplo, para se definir um tipo com valores inteiros (PEDRONI, 2010; D’AMORE, 2012; ORDONEZ et al., 2003):

```
TYPE nome_do_tipo IS RANGE 10 TO 300;
```

Para criar um tipo enumerado (PEDRONI, 2010):

```
TYPE nome_do_tipo IS (nome_de_estado1, nome_de_estado2,
nome_de_estadox);
```

Para criar um tipo de dados matriz ou vetores (PEDRONI, 2010):

```
TYPE nome_do_tipo IS ARRAY intervalo_do_array OF
tipo_de_dados_dos_elementos;
```

A linguagem VHDL também permite que se criem subtipos de dados, ou seja, subconjuntos de tipos já existentes, cuja vantagem de se criar um subtipo em vez de um

tipo novo é que é possível a troca de dados entre um subtipo e seu tipo de origem, um exemplo da sintaxe de um subtipo é (PEDRONI, 2010):

```
SUBTYPE nome_do_subtipo IS INTERGER RANGE 10 TO 50 ;
```

3.3 CLASSES DE OBJETOS

Objetos em VHDL são elementos que armazenam valores, nessa linguagem existe 4 classes de objetos(PEDRONI, 2010; D'AMORE, 2012): constant, signal, varable e file.

3.3.1 Constant

Um objeto do tipo constant pode ser declarado na entidade, arquitetura, bloco, processo e subprogramas. Como o nome sugere ele armazena um valor constante que não pode ser alterado no decorrer do código, semelhante a GENERIC. Tem o seguinte modelo (PEDRONI, 2010; D'AMORE, 2012) :

```
CONSTANT nome_da_constante : tipo_de_dado : = valor_da_constante;
```

3.3.2 Signal

O signal define I/Os e conexões entre fios. O seu valor pode ser modificado ao longo do código. Pode ser utilizado nos mesmo lugares que CONSTANT, podendo está presente em código sequencial ou concorrente, porém, em código concorrente ele só atualiza o seu valor no fim do processo e não a cada ciclo. Sua sintaxe (PEDRONI, 2010; D'AMORE, 2012):

```
SIGNAL nome_do_signal : tipo_de_dado;
```

Na atribuição de valor inicial pode-se usar ": =", porém, para outras atribuições se usa "<=".Pode ser criado um intervalo de valores para ele usando RANGE e também lhe atribuir valores iniciais, exemplo (PEDRONI, 2010; D'AMORE, 2012):

```
SIGNAL exemplo : INTEGER RANGE 0 TO 10;
SIGNAL exemplo2 : INTEGER : = 2;
```

3.3.3 Variable

O tipo varable é utilizado em regiões de código sequencial. Seu valor pode ser alterado a qualquer momento. Sua sintaxe é semelhante a do SIGNAL, porém começa com o nome VARIABLE em vez de SIGNAL e sua atribuição de valores sempre usa “: =” (PEDRONI, 2010).

3.3.4 File

Objetos do tipo file geralmente são usados para simulação e estão associados à criação de arquivos (PEDRONI, 2010; D’AMORE, 2012).

3.4 OPERADORES

A linguagem VHDL possui as seguintes categorias de operadores, que estão no quadro logo abaixo, que servem para operar com alguns dos tipos de dados já citados, e os operadores também dependem do pacote que se está usando (PEDRONI, 2010).

Quadro 3: Operadores de VHDL (Início).

Tipo de Operador	**Símbolo do Operador**	**Operação realizada**
Lógicos	NOT	NOT lógico
	AND	AND lógico
	NAND	NAND lógico
	OR	OR lógico
	NOR	NOR lógico
	XOR	XOR lógico
	XNOR	XNOR lógico
Aritméticos	+	Adição
	-	Subtração
	*	Multiplicação
	/	Divisão
	**	Exponenciação
	ABS	Valor absoluto
	REM	Resto de a/b com sinal de a
	MOD	Resto de a/b com sinal de b

Quadro 3: Operadores de VHDL (Continuação).

Tipo de Operador	Símbolo do Operador	Operação realizada
De comparação	=	Igual
	/=	Diferente
	>	Maior
	<	Menor
	>=	Maior ou igual
	<=	Menor ou igual
De deslocamento	SLL	Os dados são deslocados para a esquerda, com "0"s nas posições vazias.
	SRL	Os dados são deslocados para a direita com "0's nas posições vazias.
	SLA	Dados deslocados à esquerda, com bit extremo direito nas posições vazias.
	SRA	Dados deslocados à direita, com bit extremo esquerdo nas posições vazias.
	ROL	Deslocamento circular para a esquerda.
	ROR	Deslocamento circular para direita.
De concatenação	&	Utilizado para fazer concatenação (agrupamento) de valores.
Operadores de Atribuição	<=	Para atribuição de valores a sinais.
	:=	Para atribuição de valores a variáveis e constantes ou valores iniciais (default) de sinais e variáveis.
	=>	Para atribuição de valores a elementos individuais de conjuntos de bits.

Fonte: Adaptado do Pedroni (2010).

3.5 CÓDIGO CONCORRENTE

O código concorrente é adequado para modelar circuitos combinacionais, enquanto o código sequencial é utilizado tanto para modelar circuitos combinacionais quanto para modelar circuitos sequenciais. Para criar um código sequencial e suas instruções é preciso criar regiões. Já as instruções de código sequencial só podem ser

usadas no código sequencial enquanto que instruções de código concorrente só podem ser usadas fora do código sequencial (PEDRONI, 2010).

Em um código concorrente, as instruções ocorrem todas ao mesmo tempo de forma que a ordem como o código é escrito é irrelevante. Em um código concorrente pode ser usando os operadores já citados e as seguintes instruções: WHEN, SELECT e GENERATE (PEDRONI, 2010).

3.5.1 When

A instrução WHEN é utilizada para fazer uma atribuição condicional, possui a seguinte sintaxe (PEDRONI, 2010; D'AMORE, 2012):

```
Sinal_que_recebe <= expressao1 WHEN condiçao1 ELSE
                    expressao2 WHEN condicao2 ELSE
                    expressaoN;
```

Para cada condição, o sinal receberá uma expressão diferente, e caso nenhuma condição seja atendida, também haverá uma determinada expressão para o sinal receber. (PEDRONI, 2010; D'AMORE, 2012)

3.5.2 Select

Um sinal recebe uma expressão entre várias opções de acordo com a condição de valor, da expressão de escolha, que for verdadeira, nenhuma das condições possui prioridade sobre as outras. A sintaxe é a seguinte (PEDRONI, 2010; D'AMORE, 2012):

```
WITH expressao_de_escolha SELECT
Sinal_que_recebe <= expressao1 WHEN condição1,
                    expressao2 WHEN condição2,
                     expressaoN WHEN OTHERS;
```

Todas as condições possíveis devem ser satisfeitas, logo, se a expressão de escolha não for igual a nenhuma das condições testadas, pode-se usar a palavra reservada OTHERS que engloba todas as condições não citadas, definindo que expressão atribuir ao sinal em tal situação (PEDRONI, 2010; D'AMORE, 2012; ORDONEZ et al., 2003).

3.5.3 Generate

O GENERATE cria um laço que repete uma tarefa tantas vezes quanto for permitido pelos limites de um identificador, ou enquanto uma condição for verdadeira, a forma de parada do laço depende de qual sintaxe de GENERATE se usar, há duas, FOR GENERATE e IF GENERATE, a forma de FOR GENERATE é (PEDRONI, 2010; D'AMORE, 2012; PEDRONI, 2004):

```
rotulo: FOR indentificador IN (limites de execução) GENERATE
          -- Comandos da tarefa
        END GENERATE rotulo; -- o rotulo aqui é opcional
```

A instrução IF GENERATE tem a seguinte sintaxe:

```
rotulo: IF condição GENERATE
        -- Comandos da tarefa
      END GENERATE rotulo;-- o rotulo aqui é opcional
```

3.6 CÓDIGO SEQUENCIAL

Para usar código sequencial é preciso criar um processo (PROCESS) ou subprogramas (FUNCTION ou PROCEDURE), PROCESS é uma instrução de código concorrente, enquanto que os subprogramas são criados geralmente para serem armazenados e usados a partir de uma biblioteca. As principais instruções de código sequencial são: IF, LOOP, CASE e WAIT (PEDRONI, 2010; D'AMORE, 2012; PEDRONI, 2004). Pela importância de PROCESS para o código sequencial ele está sendo tratado aqui ao invés de estar no tópico sobre código concorrente.

3.6.1 Process

O PROCESS pode ser escrito dentro da arquitetura, e é ativado sempre que um SIGNAL em sua lista de sensibilidade muda (exceto quando usa o WAIT, nesse caso não se usa lista de sensibilidade, PROCESS se inicia sempre que uma determinada condição for satisfeita), dentro dele, então, é executado sequencialmente as operações e

instruções lá definidas, quando elas encerrarem o PROCESS acaba e só retorna novamente caso as exigências ditas antes voltem a ocorrer. Sua sintaxe é a seguinte (PEDRONI, 2004; D'AMORE, 2012):

```
rotulo: PROCESS (Lista de sensibilidade)
         -- pode se fazer declarações de vários tipos
         BEGIN
          -- tarefas diversas
           END PROCESS rotulo;
        -- Uso de rotulo não é obrigatório.
```

3.6.2 If

Determinadas atribuições são executadas de acordo com a condição satisfeita, possui a seguinte sintaxe (PEDRONI, 2004; D'AMORE, 2012):

```
rotulo: IF condiçao_1  THEN atribuicoes_1 ;
              -- rotulo não é obrigatório
         ELSIF condiçao_2 THEN atribuicoes_2;
         ELSIF condicao_3 THEN atribuicoes_3;
          -- ...
          ELSE atribuição_n;
```

3.6.3 Case

Atribuições serão feitas de acordo com a condição que satisfaz a expressão de escolha (PEDRONI, 2004; D'AMORE, 2012):

```
CASE expressão_de_escolha IS
WHEN condiçao_1 => atribuiçoes_1;
WHEN condição_2 => atribuição_2;
 -- ...
END CASE;
```

3.6.4 Wait

O WAIT em a capacidade de suspender um PROCESS, e como já foi mencionado, quando se usa WAIT, o PROCESS não possui uma lista de sensibilidade. Geralmente, é utilizado em entidades que servem para testar outras entidades. Há três variações desse comando, WAIT ON, que suspende um PROCESS até que certo sinal mude de valor, WAIT UTIL que suspende um PROCESS até que uma condição seja atendida e WAIT FOR que suspende PROCESS até que um período de tempo decorra (esse ultimo não é sintetizável). Abaixo a sintaxe dos 3 (o rótulo também é opcional) (PEDRONI, 2004; D'AMORE, 2012) :

```
WAIT ON sinais;
WAIT UTIL condicoes;
WAIT FOR expressao_de_tempo;
```

3.6.5 Loop

O LOOP um laço que executa instruções enquanto não ocorrem certas condições de parada. As condições de parada podem ser limites de execução determinados ou condições. Há dois tipos de LOOP: o FOR LOOP e WHILE LOOP, segue a sintaxe de FOR LOOP (os rótulos dos modelos desse tópico podem ser usados ou não) (PEDRONI, 2004; D'AMORE, 2012):

```
rotulo: FOR indentificador IN limites_de_execucao LOOP
-- Tarefas
END LOOP rotulo;
```

Já o WHILE LOOP (PEDRONI, 2004); D'AMORE, 2012):

```
rotulo: WHILE condicao LOOP
-- Tarefas
END LOOP rotulo;
```

Há duas declarações que também influenciam no modo de operação do LOOP e são utilizadas dentro dele. Elas são: palavra reservada EXIT que é utilizada para

finalizar um LOOP, pode ter condição ou não, e a palavra reservada NEXT que também pode ter uma condição ou não e tem a capacidade de fazer o LOOP pular de tarefas, segue a sintaxe (PEDRONI, 2004; D'AMORE, 2012):

```
rotulo: EXIT rotulo WHEN condicao;
rotulo: NEXT rotulo_do_laco WHEN condicao;
```

3.7 OUTROS COMANDOS IMPORTANTES

Os comandos citados aqui estão de alguma forma relacionada com a biblioteca, reutilização e compartilhamento de estruturas de descrição, são eles: PACKAGE (pacote), COMPONENT (componente), FUNCTION (função) e PROCEDURE (procedimento) (PEDRONI, 2004; D'AMORE, 2012).

3.7.1 Package

O PACKAGE não é destinado ao código principal, é um meio para se armazenar em uma unidade diversos tipos de recursos (como, constantes, tipos de dados, funções, procedimentos, entre outros) para que possam ser compartilhadas e reutilizadas por projetistas a partir da biblioteca onde foram armazenadas. A estrutura de um PACKAGE se divide em duas partes, a primeira delas chamada de PACKAGE é onde se faz vários tipos de declarações (como tipos, subtipos, funções, procedimentos, constantes, entre outros), a segunda parte chamada de PACKAGE BODY só é usada se dentro da PACKAGE houver alguma descrição de FUNCTION ou PROCEDURE, então, nela será descrito toda a estrutura das declarações FUNCTION e PROCEDURE. Sua sintaxe (PEDRONI, 2004; D'AMORE, 2012; PEDRONI, 2010) :

```
PACKAGE nome_da_package  IS
-- Declarações diversas
END nome_da_package;
PACKAGE BODY nome_da_package IS
-- Descrições de funções e procedimentos diversos
END nome_da_package;
```

3.7.2 Component

Os componentes permitem utilizar uma entidade de um projeto dentro de outra entidade (aqui a palavra “entidade” se refere a todo o corpo de um código, contendo declarações de bibliotecas e pacotes, a entidade do modo como foi mencionada antes, arquitetura, entre outras estruturas). Isso é muito útil para não ser necessário ficar repetindo códigos em um projeto, e permite criar descrições hierárquicas, chamadas também de descrições estruturais. Em uma descrição estrutural pode-se criar estruturas de códigos mais simples e através delas, com o uso de componentes, criar estruturas mais complexas, por exemplo, é possível criar entidades para portas simples como AND e OR, utilizá-las como componentes em um projeto de um flip-flop. E o projeto do flip-flop pode também ser utilizado como componente para criar um registrador (PEDRONI, 2004; D’AMORE, 2012; PEDRONI, 2010).

Um componente pode ser declarado dentro de um pacote ou de uma arquitetura. Declará-lo dentro de um pacote torna mais simples o modo de usá-lo no projeto principal. Abaixo a sintaxe de um componente (PEDRONI, 2004; D’AMORE, 2012; PEDRONI, 2010):

```
COMPONENT nome_do_componente IS
-- declaração de genéricos e portas do componente, são os
-- mesmos da entidade da qual ele deriva.
END COMPONENT;
```

A sua instanciação é realizada através do PORT MAP, a associação entre as portas do projeto principal com as portas do componente, tem a seguinte sintaxe (PEDRONI, 2004; D’AMORE, 2012; PEDRONI, 2010):

```
(Rotulo: nome_do_componente PORT MAP ( associacao_das_portas);
```

Também pode haver um GENERIC MAP, que associa os genéricos da entidade principal com o genéricos dos componentes, sua sintaxe é semelhante a do PORT MAP (PEDRONI, 2004) (D’AMORE, 2012)(PEDRONI, 2010).

3.7.3 Function

A principal utilidade do FUNCTION, dentro da VHDL, é criar códigos sequenciais, que são usados com muita frequência, e colocá-los dentro de pacotes, que possam ser referenciados em bibliotecas, para serem usados no código principal. Dentro dela, pode-se utilizar todas as instruções de código sequencial citados aqui (LOOP, IF, CASE, WAIT), exceto o WAIT. Podem ser declaradas principalmente em pacotes e na parte declarativa da arquitetura. Em sua lista de parâmetros de entrada (que não é obrigatório ter) se aceita todos os tipos de objetos citados anteriormente, exceto VARIABLE. Ela retorna um único valor e sua sintaxe de declaração possui o seguinte modelo, abaixo descrito por elementos, e não um código propriamente (D'AMORE, 2012; PEDRONI, 2010):

```
FUNCTION nome_da_funcao  (parametros) RETURN tipo_de_dado IS
      -- Declarações de vários tipos como constantes e
      --  variáveis
       BEGIN
      -- comandos e operações sequenciais
      END nome_da_funcao;
```

A sintaxe da lista de parâmetros de entrada (*parametros*) pode ser dada por (PEDRONI, 2010):

```
(parametros) = CONSTANT nome_da_constante : tipo_da_constante;
-- ou
               = SIGNAL nome_do_sinal : tipo_do_sinal;
```

3.7.4 Procedure

A utilização de PROCEDURES é semelhante a das funções, possuindo como principais diferenças poder retornar mais de um valor, aceitar VARIABLE como entrada, ser uma instrução por si só enquanto que funções são usadas como partes de expressões. Exemplo de sua sintaxe (D'AMORE, 2012; PEDRONI, 2010):

```
PROCEDURE nome_do_procedimento (parâmetros) IS
-- Declarações de vários tipos
BEGUIN
-- Código sequencial
END nome_do_procedimento;
```

Os parâmetros podem ser do tipo VARIABLE, SIGNAL ou CONSTANT, e podem ter o modo IN, INOUT ou OUT, com as seguintes sintaxes (D'AMORE, 2012; PEDRONI, 2010):

```
CONSTANT nome_da_constante: modo      tipo;
SIGNAL nome_do_signal: modo   tipo;
VARIABLE nome da variavel: modo    tipo:
```

4 VERILOG

A abordagem aqui utilizada é semelhante a que foi usada na descrição das características gerais de VHDL, porém Verilog é *case sensitive*, logo as palavras reservadas da linguagem só podem ser escritas de uma única forma e o artificio didático usado com VHDL não pode ser usado aqui, além disso, as definições para cada linguagem são distintas, ou seja, algo com um nome em VHDL não é equivalente a algo com mesmo nome em Verilog, apesar de isso poder dificultar a compreensão, foram mantidas tais definições porque são comumente usadas nas referências verificadas. Já os comentários em Velirog, quando ocupam menos de uma linha são precedidos por // e os que superam uma linha iniciam com /* e finalizam com */ (BROWN; VRANESIC, 2014).

4.1 MÓDULOS

O módulo é o bloco básico de construção em verilog, ele representa a descrição de um circuito ou um subcircuito. Em seu cabeçalho, é listado suas portas de entrada e saída criando a interface do projeto com o ambiente externo. Dentro do módulo pode haver declaração de portas, variáveis, nets, tasks, functions, always, etc. O código dentro do módulo é naturalmente concorrente, podendo se criar ambientes de código sequencial. O modelo de um módulo é definido a seguir (BROWN; VRANESIC, 2014; PALNITKAR, 2003; LEE, 2003):

```
module nome_do_modulo (lista_de_portas);
/* declarações e atribuições  diversas em código
concorrente */
endmodule
```

As portas podem ser de 3 tipos, *input*, que se refere as entradas, *output*, que se refere as saídas, e *inout*, que é uma porta bidirecional de entrada e saída. Uma porta com mais de um bit tem sua dimensão em bits definida dentro de um “[ ]”. Abaixo um exemplo da sintaxe das portas (BROWN; VRANESIC, 2014; PALNITKAR, 2003; LEE, 2003):

```
input [4:0] a, b; // “a” e “b” são entradas de 5 bits cada
output c, d; // “c” e “d” são saídas de um bit
inout e, f; // “e” e “f” são portas bidirecionais
```

4.2 CONJUNTO DE VALORES

Os sinais em Verilog podem possuir 4 tipos de valores diferentes, além disso há 8 pesos utilizados para resolver conflitos entre condutores de importância diferentes. Abaixo a tabela dos valores (PALNITKAR, 2003):

Quadro 4: Valores em Verilog.

Nível de Valor	Condição em Circuitos Digitais
0	Zero lógico, condição falsa
1	Um lógico, condição verdadeira
X	Valor desconhecido
Z	Alta impedância, estado flutuante

Fonte: Palnitkar (2003).

O valor de “x” e “z” também pode ser denotado pelas letras maiúsculas “X” e “Z” respectivamente (LEE, 2003). A seguir a tabela dos pesos para os valores de 0 e 1 (PALNITKAR, 2003):

Quadro 5: Pesos para os Valores em verilog.

Nível de peso	**Tipo**	**Degrau**
Supply	Transmissão	Maior prioridade
Strong	Transmissão	↑
Pull	Transmissão	
Large	Armazenamento	
Weak	Transmissão	
Médium	Armazenamento	
Small	Armazenamento	
Highz	Alta impedância	Menor prioridade

Fonte: Palnitkar (2003).

4.3 OPERADORES

Abaixo uma tabela com os operadores suportados por Verilog, alguns deles são bem similares a operadores da linguagem C (PALNITKAR, 2003).

Quadro 6: Operadores de Verilog (Inicio).

Tipo de Operador	Símbolo do Operador	Operação realizada	Número de Operando
Aritmético	*	Multiplicação	Dois
	/	Divisão	Dois
	+	Adição	Dois
	-	Subtração	Dois
	%	Módulo	Dois
	**	Potenciação	Dois
Lógico	!	NOT lógico	Um
	&&	AND lógico	Dois
	||	OR lógico	Dois
Relacional	>	Maior que	Dois
	<	Menor que	Dois
	>=	Maior ou igual que	Dois
	<=	Menor ou igual que	Dois
Igualdade	==	Igualdade	Dois
	!=	Desigualdade	Dois
	===	Igualdade Case	Dois
	!==	Desigualdade Case	Dois
Bit a Bit	~	NOT	Um
	&	AND	Dois
	|	OR	Dois
	^	XOR	Dois
	^~ or ~^	XNOR	Dois
Redução	~&	AND redutor	Um
	|	OR redutor	Um
	~|	NOR redutor	Um
	^	XOR redutor	Um
	^~ or ~^	XNOR redutor	Um
Deslocamento	>>	Deslocamento a direita.	Dois
	<<	Deslocamento a esquerda.	Dois
	>>>	Deslocamento a direita aritmético.	Dois
	<<<	Deslocamento a esquerda aritmético.	Dois
Concatenação	{ }	Concatenação	Qualquer número
Replicação	{ { } }	Replicação	Qualquer número
Condicional	?:	Condicional	Três

Fonte: Palnitkar (2003).

4.4 NETS

Os nets representam nós de circuitos, como há diversos tipos de nós em circuitos, também há diversos tipos de nets. O net mais utilizado se chama wire, ele

representa interconexões de circuitos, e é análogo há um fio em um hardware, porém, seu valor pode ser atualizado continuamente. Sua declaração tem a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003; LEE, 2003):

```
wire [largura_do_vetor] wire1, wire2, wireN;
```

A largura_do_vetor é utilizada caso se designe wires que recebem conjuntos de valores escalares, serve para designar a quantidade de valores escares que tais wires podem receber (BROWN; VRANESIC, 2014). Abaixo um exemplo:

```
wire [3:0] a, b; // declara dois vetores wire de 4 bits
```

4.5 VARIÁVEIS

Variáveis são utilizadas em descrições comportamentais, valores podem ser atribuídos a elas através de comandos e mudados a qualquer momento. Há dois tipos de variáveis em Verilog: integer e reg (BROWN; VRANESIC, 2014).

4.5.1 Reg

Esse tipo de variável representa elementos armazenadores de dados (PALNITKAR, 2003). São utilizados tanto para descrição de circuitos combinacionais quanto sequenciais, abaixo sua sintaxe (BROWN; VRANESIC, 2014):

```
reg [largura_do_vetor] reg1, reg2, regN;
```

4.5.2 Integer

São variáveis de propósito gerais utilizadas para manipulação de quantidades, são úteis em descrições comportamentais de um módulo, porém não correspondem diretamente a nós de circuitos. Possui a seguinte sintaxe (PALNITKAR, 2003; BROWN; VRANESIC, 2014):

```
integer nome1, nome2, nomeN; // Não se define o tamanho
```

4.6 ESPECIFICAÇÃO DE NÚMEROS

Os números atribuídos às variáveis podem ser dados por constantes que possuem a seguinte sintaxe (BROWN; VRANESIC, 2014):

```
<comprimento>’<base numérica><constante>
```

Exemplos:

```
4’b1000 // Número binário de 4 bits
12’habc //Número hexadecimal de 12 bits
16’d255// Número decimal de 16 bits
4’o5 // número octal de 4 bits
```

Como o exemplo mostra, as bases numéricas podem ser: octal, hexadecimal, binário e decimal. Caso não se defina a base, o número é entendido como decimal. Caso a constante declarada não tenha o <comprimento> que foi determinado, então os outros bits não determinados serão preenchidos com zero, exceto quando o primeiro valor da constante for “x” ou “z”, nesse caso o preenchimento será feito com tal valor. (BROWN; VRANESIC, 2014).

4.7 PARAMETERS

Os *parameters* são utilizados para definir constantes em Verilog. O valor de um *parameter* de um módulo pode ser substituído por outro valor quando tal módulo é instanciado dentro de outro. A seguir sua sintaxe (PALNITKAR, 2003; BROWN; VRANESIC, 2014):

```
parameter nome_da_constante = <valor numérico>;
```

4.8 VETORES E ARRAYS

Nos tópicos acima sobre nets e variáveis foi mostrado o uso de vetores em exemplos, um vetor de dados do tipo net ou variável é composto de um elemento formado por vários bits, já os *arrays* são formados por vários elementos que podem ter ou não vários bits, abaixo exemplos de *arrays* (PALNITKAR, 2003):

```
wire [2:0] a[3:0]; /* designa um wire “a” de 4 elementos com
3 bits cada */
reg b[9:0]; /* designa um reg b de 10 elementos de 1 bit cada
*/
```

4.9 CÓDIGO CONCORRENTE

O código em Verilog é escrito de forma concorrente, ou seja, as atribuições e declarações ocorrem paralelamente entre elas, exceto dentro de alguns tipos de blocos que permitem código sequencial. Abaixo duas formas de declarações em código concorrente: instanciação de portas lógicas primitivas e atribuições contínuas (BROWN; VRANESIC, 2014; LEE, 2003; PALNITKAR, 2003).

4.9.1 Instanciação de Portas Lógicas Primitivas

a linguagem Verilog permite descrever a estrutura de um circuito através da instanciação de portas lógicas usando a seguinte sintaxe (BROWN; VRANESIC, 2014):

```
nome_da_porta_lógica <rótulo> (saida, entrada1, entrada2,
entradaN):
```

Desta forma é possível descrever a estrutura de diversos tipos de portas lógicas e até mesmo transistores, porém nem todos são sintetizáveis. Quando a saída usada nesse tipo de instanciação não tem sido declarada antes, ela é definida como do tipo wire (LEE, 2003; BROWN; VRANESIC, 2014). Exemplo:

```
and  and1 (a, b, c);   /*entradas b e c conectadas a uma
porta lógica and com saída c */
```

4.9.2 Atribuições contínuas

Diferente da instanciação de portas lógicas primitivas, declaração de atribuição continua não representa descrição estrutural do hardware e sim da função de circuitos. Basicamente, através desse tipo de declaração, se atribui valores de nets, portas de entradas ou resultados de operações a outros nets ou portas de saídas, de forma continua, ou seja, o valor atribuído pode ser alterado a qualquer momento, desde que se

mude algo na expressão atribuidora. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; LEE, 2003; PALNITKAR, 2003):

```
assign  nome_de_wire_ou_output =<expressão a ser
atribuída> , <outras atribuições>;
/*Em uma declaração de assign pode ser feito diversas
atribuicões */
```

4.10 CÓDIGO SEQUENCIAL

Em Verilog, é permitido o uso de código sequencial dentro de alguns tipos de blocos, nessa forma de descrição as declarações não descrevem um comportamento paralelo, pelo contrário, cada declaração é executada na ordem que é escrita, ou seja, a ordem é relevante. O bloco *always* e *initial* são blocos que permitem o uso de código sequencial, o primeiro é sintetizável, o outro é utilizado em simulações e não é sintetizável (BROWN; VRANESIC, 2014).

4.10.1 Always

O bloco *Always* permite que declarações e atribuições sequenciais ocorram sempre que há uma mudança em um dos sinais da lista_de_sensibilidade. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
always @(lista_de_sensibilidade)
begin
// declarações em código sequêncial
end
```

O *begin* no início e *end* no fim só é necessário caso haja mais de um comando dento do bloco (BROWN; VRANESIC, 2014).

4.10.2 Initial

O bloco *initial* é semelhante ao *always*, porém, esse só executa uma única vez quando se inicia a simulação, segue sua sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
initial
begin
// declarações em código sequêncial
end
```

4.10.3 Declaração de If-Else

Com esse tipo de declaração, determinadas partes de código são executadas se uma expressão condicional tiver valor lógico verdadeiro, caso o valor lógico seja falso, com uso do else if ou else, é possível que outras condicionais sejam testadas e, dependendo de seus valores lógico, executadas, dessa forma é possível criar várias condições entrelaçadas. O uso do else if ou do else não é obrigatório, else if testa uma nova condição, o else determina o que será executado quando a condição do if ou else if anterior a ele for falsa, e as palavras begin e end só são obrigatórias se houver mais de um comando dentro das condicionais. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
if (expressão)
begin
// Declarações
end
else if (expressão2)
begin
// Declarações
end
else
begin
//Declarações
end
```

4.10.4 Declaração de Case

Uma expressão de controle é comparada com tantas alternativas se queira, de forma que comandos são executados caso o valor da alternativa seja equivalente ao da expressão. O *default* descreve o que ocorre caso nenhuma das alternativas dadas sejam equivalente à expressão de controle. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; LEE, 2003; PALNITKAR, 2003):

```
case (expressão)
alternativa1: begin
              //Declarações
               end
              // Outras alternativas que se queira
default: begin     // default não é obrigatório
              // Declarações
               end
endcase
```

Existem outras variações dessa declaração chamados de casez e casex, a diferença entre eles e o case convencional é que eles podem interpretar os valores x e z de forma diferente nas alternativas e na expressão de controle. No casez o valor z é tratado como uma condição que não importa. Já no casex tanto o valor x como o z são tratados como condição sem importância (BROWN; VRANESIC, 2014).

4.10.5 Declaração de Laços

Em Verilog, há os seguintes tipos de declarações de laço: for, while, repeat e forever. Somente o tipo for é sintetizável, ele permite que um evento repita enquanto uma variável de controle, que é incrementada, atende a uma determinada condição, possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
for (variavel_inicializada; condição;
incremento_da_variavel)
begin
// declarações em código sequencial
end
```

A instrução *repeat* permite que se repita um ciclo um número fixo de vezes. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
repeat (valor_contante)  /*determina o número de vezes que
se repitirá */
begin
// declarações em código sequencial
end
```

O laço *while* permite que um ciclo se repita enquanto uma expressão condicional for satisfeita, possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
while (expressão_condicional)
begin
// declarações em código sequencial
end
```

4.10.6 Atribuições Blocking e Non-blocking

Há dois tipos de atribuições em código sequencial blocking e non-blocking , nas atribuições blocking os valores são atribuídos as variáveis sequencialmente na ordem que se encontrarem e de imediato, porém em atribuições non-blocking o valor das variáveis só é atualizado no fim do ciclo. A seguir a sintaxe de ambos (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
variavel1 = expressao1 // atribuição blocking
variavel2 <= expressao2 // atribuição nos-blocking
```

4.11 INSTANCIAÇÃO DE MÓDULOS

A instanciação de um módulo (subcircuito) dentro de outro módulo é semelhante a instanciação de portas lógicas primitivas. Para instanciar um módulo em outro é necessário que ambos estejam no mesmo arquivo ou possam ser localizados pelo sintetizador (a forma como isso corre varia de sintetizador para sintetizador). Utilizando esse tipo de declaração é possível fazer grandes projetos hierárquicos. Possui a seguinte sintaxe (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
Nome_do_modulo #(substituicao_de_parameters)
nome_instanciacao
(.porta1(expressao),.porta2(expressao2),.portaN(expressaoN)
);
```

Cada instanciação deve possuir um nome único dentre os identificadores permitidos na linguagem, e é possível instanciar um modulo várias vezes em um projeto e cada vez a instanciação terá um nome diferente. O termo *substituicao_de_parameters* designa o local onde se pode atribuir novos valores aos *parameters* declarados dentro do modulo instanciado, nesse local, basta colocar os novos valores separados por virgula na ordem em que os *parameters* foram designados dentro do modulo instanciado, caso se coloque menos valores que a quantidade de *parameters* existentes no modulo instanciado, serão trocados somente os valores da quantidade que foi colocado na ordem. Outra forma de substituir os *parameters* é usando a declaração *defparam* (BROWN; VRANESIC, 2014; PALNITKAR, 2003).

No modelo acima os termos *porta1, porta2* e *portaN* se referem as portas do modulo instanciado, os termos *expressao, expressao2* e *expressaoN* se referem as conexões que as portas do modulo instanciado (nessa mesma ordem) terão, isso é chamado de conexão de portas por nome. É possível definir as conexões das portas do subcircuito com o modulo de outra forma, basta colocar as conexões das portas do modulo instanciado na ordem das respectivas portas que serão conectadas, isto é chamado de conexão de portas por ordem, a sintaxe seria a seguinte (BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
Nome_do_modulo #(substituicao_de_parameters)
nome_instanciacao  (expressao,expressao2,expressaoN);
```

4.12 TASKS E FUNCTIONS

Em uma descrição comportamental pode ser necessário usar uma funcionalidade em diversos lugares do código, para isso, pode ser relevante captar essas partes do código, que serão utilizadas muitas vezes, em rotinas, e então poder invocar essas rotinas sempre que necessário. Esses tipos de procedimentos podem ser utilizados em Verilog através da declaração de tasks e functions (PALNITKAR, 2003).

A função é definida dentro de um módulo, e é chamada ou em uma instrução de atribuição contínua ou em uma instrução de atribuição processual dentro deste módulo. Uma função pode ter mais do que uma entrada, mas não tem uma saída, porque o próprio nome da função serve como variável de saída (BROWN; VRANESIC, 2014). Dentro de uma função não se pode invocar tasks, porém pode se invocar outras functions (PALNITKAR, 2003). Abaixo a sua sintaxe (BROWN; VRANESIC, 2014):

```
function <tamanho>  integer  nome_da_function;
/* declaração de portas de entrada, parameters ou
variáveis locais */
begin
// declarações em código sequencial
end
endfunction
```

O <*tamanho*> e *integer* na sintaxe é opcional, são utilizados quando se quer determinar o tamanho do vetor de saída da *function*, caso não se coloque eles, a variável é definida como de um bit. Sua instanciação é semelhante à instanciação de um módulo, se declara o nome da function seguido das portas de entrada entre parênteses e depois ponto e virgula (BROWN; VRANESIC, 2014; PALNITKAR, 2003).

Uma *task* admite tanto argumentos de entrada como de saída e só pode ser instanciada dentro de um bloco *always* ou *initial*, possui a seguinte sintaxe(BROWN; VRANESIC, 2014; PALNITKAR, 2003):

```
task  nome_da_task;
// declaração de portas , parameters ou variáveis locais
begin
// declarações em código sequencial
end
endtask
```

5 RESULTADOS E DISCUSSÕES

No ambiente de descrição Quartus Prime foram implementados, em ambas as HDLs, Verilog e VHDL, os códigos das portas lógicas AND, OR, NAND, NOR, NOR, XOR e XNOR. Assim como os códigos de memórias RAM (Random Access *Memory*) 16x4 (16 palavras com 4 bits) , ROM (*Read-Only Memory*) 8x8 (8 palavras com 8 bits), registador de 4 bits. Um somador completo e uma maquina de estados do temporizador de um laser que cujo circuito descreve um *laser* que após ser iniciado por um sinal de entrada fica ligado durante 3 ciclos de *clock* voltando então para seu estado desligado, também possui uma entrada cujo sinal pode fazer a maquina do *laser* voltar ao estado de desligado a qualquer momento, esse código se encontra como um exemplo no Vahid(2008).

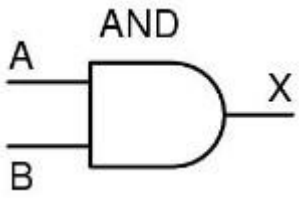
O Quartus Prime permite a analise e síntese de projetos em HDLs. Um projetista que o utiliza pode sintetizar com ele seus projetos, realizar analise de tempo, examinar diagramas de RTL, utilizar estímulos para analisar a resposta do que se é projetado, analisar os sinais do *hardware* em forma de onda, assim como configurar dispositivos lógicos programáveis, etc.

Os parâmetros utilizados para comparação entre as duas linguagens analisadas são: tempo de compilação, lógica utilizada, número de registadores utilizados, total de pinos utilizados, total de bits de bloco de memórias e total de PLLs, analise de tempo (esse parâmetro não foi utilizado na analise das implementações dos códigos das portas lógicas simples).

O dispositivo lógico programável escolhido para a analise de implementação, no Quartus Prime, foi um 5CGXFC7C7F23C8 da família Ciclone V. Os dispositivos lógicos programáveis possuem diversas portas lógicas, flip-flops, registradores e etc, cujas interconexões podem ser programadas criando sistemas com funcionalidades distintas, e também é possível reprograma-los. Em um projeto o ideal é escolher um dispositivo com os mecanismos adequados para sua funcionalidade.

Para dar alguma noção dos circuitos descritos nas HDLs, aqui será mostrado um pouco de como se comportam. Começando da porta AND que é extremamente simples, como pode ser visto na sua tabela verdade (Quadro 7). A sua representação em digrama pode ser vista na Figura 1 e os seus códigos se encontram no apêndice A.

Figura 1: Porta AND.



Fonte: Próprio Autor.

Quadro 7: tabela verdade AND.

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Fonte: Próprio Autor.

Dada a simplicidade da porta AND não surpreende que no resultado não haja grandes diferenças entre as HDLs para a implementação de seu código. Como pode ser visto na Tabela 1, o único parâmetro em que os resultados apresentam diferenças entre VHDL e Verilog é no tempo de compilação, porém não foi uma diferença significativa, e pode ser atribuída a outros fatores como a ocupação do processador do computador durante as compilações.

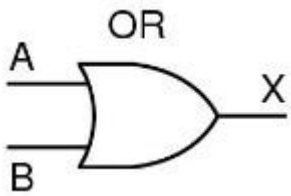
Tabela 1 - Resultado do Código da Porta AND.

Porta AND	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:03:28	00:02:34

Fonte: Próprio Autor.

A porta OR é uma das portas lógicas básicas de circuitos digitais assim como a porta AND e possui também a mesma quantidade de entradas e saídas como pode ser visto em sua representação na Figura 2, já a sua tabela verdade se encontra no Quadro 8 e seus códigos no apêndice B.

Figura 2: Porta AND.



Fonte: Próprio Autor.

Quadro 8: Tabela Verdade OR.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

Fonte: Próprio Autor.

Os resultados foram bem similares aos da porta AND, dado que não possuem grande complexidade. Segue abaixo seus resultados na Tabela 2.

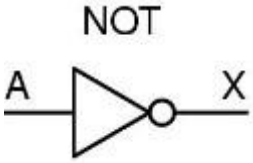
Tabela 2 - Resultado do Código da Porta OR.

Porta OR	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:50	00:03:30

Fonte: Próprio Autor.

A porta NOT também é bastante simples, porém só possui uma entrada. A sua operação lógica pode ser entendida pela sua tabela verdade no Quadro 9. A sua representação está na figura 3 e seus códigos no apêndice C.

Figura 3: Porta NOT.



Fonte: Próprio Autor.

Quadro 9: tabela verdade NOT.

A	X
0	1
1	0

Fonte: Próprio Autor.

Consequentemente a porta NOT utilizou menos pinos que a porta AND e OR na descrição em ambas as HDLs, quanto aos outros resultados foram similares. Seus resultados se encontram abaixo na tabela 3.

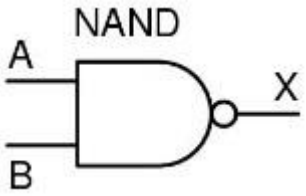
Tabela 3 – Resultado do Código da Porta NOT.

Porta NOT	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	2/268 (1%)	2/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:54	00:02:44

Fonte: Próprio Autor.

Uma porta NAND é um pouco mais complexa que as portas lógicas abordadas acima, pois é compreendida como a combinação de uma porta AND em série com uma porta NOT, a sua representação se encontra na Figura 4, sua tabela verdade no Quadro 10 e seus códigos no apêndice D.

Figura 4: Porta NAND.



Fonte: Próprio Autor.

Quadro 10: Tabela verdade NAND.

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Fonte: Próprio Autor.

Os resultados da porta AND foram similares aos da porta OR e da porta AND, ou seja, ambas as HDLs se mostraram igualmente boas nessa descrição, isso pode ser verificado na Tabela 4 abaixo.

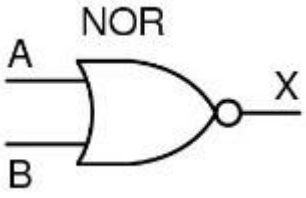
Tabela 4 - Resultado do Código da Porta NAND.

Porta NAND	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:30	00:02:49

Fonte: Próprio Autor.

Uma porta NOR é composta por uma porta OR em série com uma porta NOT, como mostra sua representação na Figura 5. A sua operação lógica pode ser verificada pela sua tabela verdade no Quadro 11 e seus códigos se encontram no apêndice E.

Figura 5: Porta NOR.



Fonte: Próprio Autor.

Quadro 11: tabela verdade NOR.

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

Fonte: Próprio Autor.

Como pode ser visto na tabela 5 abaixo, na implementação da porta NOR também não houve grande diferença entre as HDLs testadas.

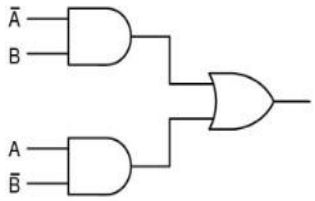
Tabela 5 - Resultado do Código da Porta NOR.

Porta NOR	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:43	00:03:24

Fonte: Próprio Autor.

Uma porta XOR pode ser compreendida como um tipo de combinação de algumas portas AND, OR e NOT, como mostra a Figura 6 (a barrinha em cima da letra A e B quer dizer que os seus valores estão negados, ou seja, conectados a uma porta NOT), consequentemente é um pouco mais complexa que as partes que a compõe, sua representação especifica está na Figura 7. Seu comportamento pode ser entendido melhor pela tabela verdade do Quadro 12 e seus códigos estão no apêndice F.

Figura 6: XOR 1.



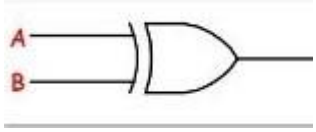
Fonte: Próprio Autor.

Quadro 12: Tabela Verdade XOR.

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Fonte: próprio Autor.

Figura 7: Porta XOR.



Fonte: Próprio Autor.

Em sua implementação a porta XOR não apresentou resultados muito distintos das outras portas lógicas, ambas as HDLs se mostraram com a mesma capacidade. O parâmetro de tempo de compilação, assim como nos outros códigos, demostrou resultados diferentes para cada HDL, as possíveis causas disso já foram mencionadas anteriormente. Na Tabela 6 abaixo se encontra os resultados para porta XOR.

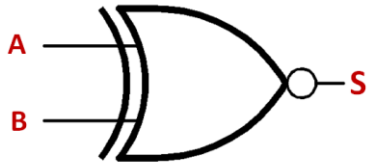
Tabela 6 - Resultado do Código da Porta XOR.

Porta XOR	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:55	00:02:19

Fonte: Próprio Autor.

A porta XNOR é dada como uma porta XOR em série com uma porta NOT, sua representação se encontra na Figura 8, sua tabela verdade no Quadro 13 e seus códigos no apêndice G.

Figura 8: Porta XNOR.



Fonte: Próprio Autor.

Quadro 13: Tabela Verdade XNOR.

A	B	S
0	0	1
0	1	0
1	0	0
1	1	1

Fonte: Próprio Autor.

Como pode ser notado vendo a Tabela 7 abaixo (com os resultados da porta XNOR) e os resultados das outras portas lógicas, não houve grandes diferenças em seus resultados nos critérios de comparação. O único critério que ainda apresentou alguma diferença de uma HDL para outra foi o tempo de compilação, logo, a diferença de complexidade entre as portas lógicas não produziu resultados diferentes quanto à comparação das HDLs.

Tabela 7 - Resultado do Código da Porta XNOR.

Porta XNOR	**Verilog**	**VHDL**
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	3/268 (1%)	3/268 (1%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:03:22	00:02:17

Fonte: Próprio Autor.

De todos os códigos cujos resultados foram apresentados e discutidos aqui, o do somador completo é o primeiro em que se é levado em conta o parâmetro de comparação das HDLs quanto a analise de tempo. Nesse critério se analisa quanto tempo de atraso tem as saídas do circuito em relação ao momento que sinais de entrada são inseridos. Nos códigos discutidos antes não se levava em conta esse parâmetro por serem mais simples e consequentemente não se esperava atrasos significativos ou notáveis. Em todos os próximos códigos, o parâmetro da análise de tempo será levado em consideração.

A explicação do funcionamento de um contador completo é um pouco mais complexa que das portas lógicas e por isso vai além do escopo desse trabalho. A sua utilidade geralmente se dá com mais de um deles associados entre si permitindo a execução da soma de números binários. Para melhor entendimento recomenda-se consultar alguma das referências citadas nesse trabalho. Na tabela 8 abaixo mostra, que

para o somador completo não houve diferença significativa nos resultados entre uma HDL e outra, e nem mesmo atraso no circuito descrito em ambas as HDLs. Os códigos do somador completo podem ser verificados no apêndice H.

Tabela 8 - Resultado do Código de um Somador Completo.

Somador Completo	**Verilog**	**VHDL**
Logica Utilizada	2/56480 (<1%)	2/56480 (<1%)
Registradores	0	0
Total de pinos	5/268 (2%)	5/268 (2%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:03:02	00:02:45
Atraso nas saídas (ns)	0	0

Fonte: Próprio Autor.

Registradores armazenam valores durante 1 ciclo de *clock*. O que foi implementado armazena 4 bits, e sua descrição em ambas as HDLs se encontra no apêndice I. Em relação aos outros códigos implementados (como pode ser visto nos resultados da Tabela 9), a sua complexidade é maior, necessita de registradores, utiliza mais pinos, porém também não se verificou diferença significativa entre Verilog e VHDL.

Tabela 9 - Resultado do Código de um Registrador.

Registrador 4 bits	Verilog	VHDL
Logica Utilizada	2/56480 (<1%)	2/56480 (<1%)
Registradores	4	4
Total de pinos	9/268 (3%)	9/268 (3%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:03:45	00:04:05
Atraso na saída (ns)	0	0

Fonte: Próprio Autor.

Uma memória que permite somente a leitura de seus dados é chamada de ROM, e os valores nela presentes lhe são colocados na sua fabricação, embora, atualmente já existam ROMs que fogem um pouco a essa definição. Os resultados da memória ROM, que se encontram na Tabela 10 abaixo, até então são os que mostram maior uso de registradores, porém, mesmo em seus códigos (que se encontram-no apêndice J) havendo descrições que não são equivalentes entre uma HDL e outra (no código em Verilog foi utilizado uma declaração initial para iniciar os valores da ROM diferente do código em VHDL em que os valores armazenados na memoria foram dados como constantes), na comparação entre Verilog e VHDL ela só fortalece a hipótese de que as duas linguagens possuem a mesma eficácia na descrição de circuitos.

Tabela 10 - Resultado do Código de uma memória ROM 8x8.

Memória ROM 8x8	Verilog	VHDL
Logica Utilizada	4/56480 (<1%)	4/56480 (<1%)
Registradores	7	7
Total de pinos	12/268 (3%)	12/268 (4%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:02:19	00:02:32
Atraso na saída (ns)	0	0

Fonte: Próprio Autor.

Uma memória RAM permite tanto a leitura como a escrita e reescrita de dados (os seus códigos se encontram no apêndice L). Como mostra a Tabela 11, o código da memoria RAM, apesar de apresentar resultados distintos dos outros códigos (pois é o único que usa blocos de memoria e também o que usa mais pinos), de um modo geral na comparação das HDLs não apresenta nada novo e só reafirma o qu e a implementação dos outros códigos mostra.

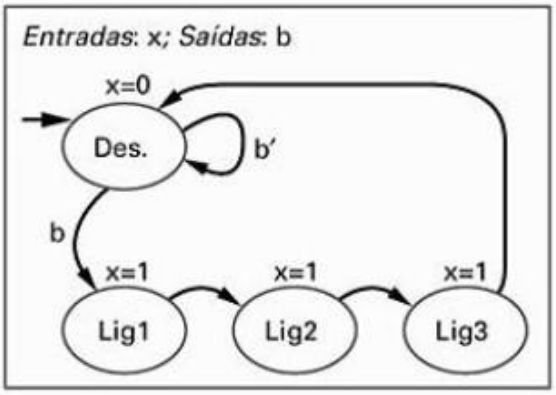
Tabela 11 - Resultado do Código de uma memória RAM 16x8.

Memoria RAM 16x8	Verilog	VHDL
Logica Utilizada	1/56480 (<1%)	1/56480 (<1%)
Registradores	0	0
Total de pinos	22/268 (8%)	22/268 (8%)
Total de bits de blocos de memoria	128/7024640 (<1%)	128/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:04:47	00:03:08
Atraso na saída (ns)	0	0

Fonte: Próprio Autor.

A maquina de estados finitos do temporizador de um laser basicamente descreve o funcionamento de um laser que, como já foi mencionado, depois de ligado fica ativo durante 3 ciclos de *clock* e depois retorna ao seu estado inicial. Tal máquina de estados finitos é um exemplo do Vahid(2008) e pode ser entendida pela Figura 9 abaixo. Cada seta ligando os estados representa a transição do *clock.* O b é o sinal que aciona o laser, x é o sinal que aciona o laser.

Figura 9: Máquina de Estados Finitos.



Fonte: Vahid (2008).

Os códigos do temporizador do laser foram os maiores descritos (se encontram no Anexo A) e os que apresentaram maior complexidade para o projetista, porém como é possível ver na Tabela 12, não houve diferenças significativas entre o código em Verilog em relação ao mesmo em VHDL, a única diferença é no tempo de compilação, porém tal diferença pode ser dada por outros fatores indiferentes as HDLs como já foi citado anteriormente.

Tabela 12 - Resultado do Código de um temporizador de laser.

Temporizador de laser	**Verilog**	**VHDL**
Logica Utilizada	2/56480 (<1%)	2/56480 (<1%)
Registradores	4	4
Total de pinos	4/268 (2%)	4/268 (2%)
Total de bits de blocos de memoria	0/7024640 (0%)	0/7024640 (0%)
Total PLLs	0/13 (0%)	0/13 (0%)
Tempo de Compilação (h:min:seg)	00:06:37	00:04:06
Atraso nas saídas (ns)	0	0

Fonte: Próprio Autor.

7 CONCLUSÃO

O estudo bibliográfico revelou que de fato não há um consenso entre os autores sobre qual a melhor HDL, já a análise das características gerais delas, revelou que possuem sintaxes bem distintas, porém, existem várias declarações similares entre elas sendo possível fazer às mesmas descrições em ambas as HDLs, e que em qualquer uma delas é possível descrever um hardware de várias formas.

No último passo deste trabalho onde vários códigos foram implementados tanto em Verilog quanto em VHDL, ambas as HDLs se mostraram igualmente eficientes, surgindo poucos resultados diferentes entre elas. Entretanto os códigos em Verilog geralmente são menores e pelo fato de sua sintaxe ter derivado da linguagem de programação C ela pode ser considerada mais fácil de aprender principalmente para programadores já acostumados com a linguagem C, já VHDL demostra ser mais organizada, pois, geralmente tudo que se instância ou se declara dentro do código possui uma interface com ele ou ambiente externo podendo dar mais clareza no entendimento dos projetos para um projetista que conhece tal HDL.

REFERÊNCIAS

BAILEY, Stephen. **Comparison of VHDL, Verilog and SystemVerilog.** 2005. Disponível em: <https://www.mentor.com/products/fv/resources/overview/comparison-of-vhdl-verilog-and-systemverilog-6884d3c1-e7a3-4425-b9ee-08ad281f867d>. Acesso em: 28 jul. 2016.

BROWN, Stephen; VRANESIC, Zvonko. **Fundamentals of Digital Logic with Verilog Design.** 3. ed. New York, Ny: Mcgraw-hill, 2014. 864 p

COUMERI, Sari L.; THOMAS, Donald E.. Benchmark descriptions for comparing the performance of Verilog and VHDL simulators. In: VERILOG HDL CONFERENCE, 1994., INTERNATIONAL, 1994, Santa Clara, Ca. **Verilog HDL Conference, 1994., International.** Santa Clara, Ca: Ieee, 1994. p. 37 - 42.

D'AMORE, Roberto. **VHDL:** Descrição e Síntese de Circuitos Digitais. 2. ed. Rio de Janeiro: Ltc, 2012.

GROUT, Ian. **Digital Systems Design with FPGAs and CPLDs.** Burlington: Elsevier Ltd, 2008.

LEE, Weng Fook. **Verilog Coding for Logic Synthesis.** New Jersey: John Wiley & Sons, 2003.

MAGINOT, Serge. Evaluation criteria of HDLs: VHDL compared to Verilog, UDL/I and M. In: DESIGN AUTOMATION CONFERENCE, 1992., EURO-VHDL '92, EURO-DAC '92, 1992, Hamburg. **Design Automation Conference, 1992., EURO-VHDL '92, EURO-DAC '92.** Hamburg: Ieee, 1992. p. 746 - 751.

OLMEDO, Aldo Alejandro Aparicio; MARCALLA, Neisser Fernando Ponluisa. **ESTUDIO COMPARATIVO DE LOS LENGUAJES HDL Y SU APLICACIÓN EN LA IMPLEMENTACIÓN DEL LABORATORIO DE SISTEMAS DIGITALES AVANZADOS MEDIANTE FPGAS EN LA EIE-CRI.** 2014. 285 f. Tese (Doutorado) - Curso de Escuela de IngenierÍa ElectrÓnica En Control y Redes Industriales, Escuela Superior PolitÉcnica de Chimborazo, Riobamba, 2014.

ORDONEZ, Edward David Moreno et al. **Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs).** Pompéia, S.p: Bless Gráfica e Editora Ltda, 2003. 300 p.

PALNITKAR, Samir. **Verilog HDL A guide to Digital Design and Synthesis**. 2. ed. California, U.s.a: Prentice Hall Ptr, 2003.

PEDRONI, Volei A.. **Circuit Design with VHDL.** London: Mit Press, 2004.

PEDRONI, Volei A.. **Eletrônica Digital Moderna e VHDL.** 5. ed. Rio de Janeiro: Elsevier, 2010. 621 p.

R.UMA; R.SHARMILA. Qualitative Analysis of Hardware Description Languages: VHDL and Verilog. **International Journal Of Computer Science And Information Security.** [s.i], p. 127-135. abr. 2011. Disponível em: <https://sites.google.com/site/ijcsis/>. Acesso em: 28 jul. 2016.

RIESGO, Teresa; TORROJA, Yago; LATORRE, Eduardo de. Design Methodologies Based on Hardware Description Languages. **Ieee Transactions On Industrial Electronics.** [s.l.], p. 3-12. fev. 1999.

SMITH, Douglas J.. VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C. In: DESIGN AUTOMATION CONFERENCE PROCEEDINGS 1996, 33RD, 1., 1996, Las Vegas, Nv. **Design Automation Conference Proceedings 1996, 33rd.** Las Vegas, Nv: Ieee, 1996. p. 771 – 776.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Sistemas Digitais: princípios e aplicações.** 11. ed. São Paulo: Prentice Hall, 2011. 840p.

VAHID, Frank. **Sistemas Digitais:** Projeto, Otimização e HDLs. Porto Alegre: Artemed, 2008. 560 p

APÊNDICE A – CÓDIGO DE UMA PORTA AND EM VERILOG E EM VHDL

```
// Porta AND em Verilog
module verilog_and (A, B, C);
input A, B;
output C;
assign
C= A & B;
endmodule
```

```
-- Porta AND em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_and is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_and;
architecture circ of VHDL_and is
begin
   C<= B and A;
end circ;
```

APÊNDICE B – CÓDIGO DE UMA PORTA OR EM VERILOG E EM VHDL

```
//Porta OR em Verilog
module verilog_or (A, B, C);
input A, B;
output C;
assign
C= A | B;
endmodule
```

```
-- Porta OR em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_or is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_or;
architecture circ of VHDL_or is
begin
   C<= B or A;
end circ;
```

APÊNDICE C – CÓDIGO DE UMA PORTA NOT EM VERILOG E EM VHDL

```
// Porta NOT em Verilog
module verilog_not (A, C);
input A;
output C;
assign
  C= ~ A;
endmodule
```

```
-- Porta NOT em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity vhdl_not is
   port (A : in bit; C : out bit);
end vhdl_not;
architecture cir5 of vhdl_not is
begin
    C<= not A;
end cir5;
```

APÊNDICE D – CÓDIGO DE UMA PORTA NAND EM VERILOG E EM VHDL

```
// Porta NAND em Verilog
module verilog_nand (A, B, C);
input A, B;
output C;
assign
  C= ~(A & B);
endmodule
```

```
-- Porta NAND em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_nand is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_nand;
architecture circ of VHDL_nand is
begin
   C<= B nand A;
end circ;
```

APÊNDICE E – CÓDIGO DE UMA PORTA NOR EM VERILOG E EM VHDL

```
// Porta NOR em Verilog
module verilog_nor (A, B, C);
input A, B;
output C;
assign
  C= ~(A | B);
endmodule
```

```
-- Porta NOR em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_nor is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_nor;
architecture circ of VHDL_nor is
begin
   C<= B nor A;
```

APÊNDICE F – CÓDIGO DE UMA PORTA XOR EM VERILOG E EM VHDL

```
// Porta XOR em Verilog
module verilog_xor (A, B, C);
input A, B;
output C;
assign
  C= A ^ B;
endmodule
```

```
-- Porta XOR em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_xor is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_xor;
architecture circ of VHDL_xor is
begin
   C<= B xor A;
end circ;
```

APÊNDICE G – CÓDIGO DE UMA PORTA XNOR EM VERILOG E EM VHDL

```
// Porta XNOR em Verilog
module verilog_xnor (A, B, C);
input A, B;
output C;
assign
  C= ~(A ^ B);
endmodule
```

```
-- Porta XNOR em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity VHDL_xnor is
   port (A : in bit; B : in bit; C : out bit);
end VHDL_xnor;
architecture circ of VHDL_xnor is
begin
   C<= B xnor A;
```

APÊNDICE H – CÓDIGO DE UM SOMADOR COMPLETO EM VERILOG E EM VHDL

```
// Somador Completo em Verilog
module som_completo ( A, B, vem1, S, vai1);
input A, B, vem1;
output S, vai1;
reg S, vai1;
always @ (A, B, vem1)
begin
    S= A ^ B ^ vem1;
     vai1 = ( A & B) | (A & vem1) | (B & vem1);
end
```

```
-- Somador Completo em VHDL
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY soma_completo IS
PORT ( A, B, vem1 : IN std_logic;
        vai1, S : out std_logic);
END soma_completo;

ARCHITECTURE comportamento_somador OF soma_completo IS
BEGIN
    PROCESS(A, B, vem1)
       BEGIN
       S <= A xor B xor vem1;
       vai1 <= (A and B) or (A and vem1) or (B and Vem1);
       end process;
END;
```

APÊNDICE I – CÓDIGO DE UM REGISTRADOR DE 4 BITS EM VERILOG E EM VHDL

```
// Registrador de 4 bits em Verilog
module registrador4 (E, CK, S);
input [3:0] E;
input CK;
output [3:0] S;
reg [3:0] S;
always @ (posedge CK)
begin
    S = E;
end
```

```
-- registrador de 4 bits em VHDL
library ieee;
use ieee.std_logic_1164.all;
entity regis is
    port ( I: in std_logic_vector (3 downto 0);
              Q: out std_logic_vector (3 downto 0);
                    clk: in std_logic
                );
end regis;
architecture com_reg of regis is
begin
   process(clk)
      begin
         if (clk='1' and clk'event) then
               Q<=I;
           end if;
      end process;
end com_reg;
```

APÊNDICE J – CÓDIGO DE UMA MEMÓRIA ROM DE 8 PALAVRAS, 8 BITS CADA, EM VERILOG E EM VHDL

```
// Memória ROM 8x8 em Verilog
module rom (input clk,
                input wire [2:0] endereco,
                output reg [7:0] Dados);
  //-- Memoria
  reg [7:0] rom [0:7];

  always @(negedge clk) begin
    Dados <= rom[endereco];
  end
//-- Inicialização da memoria.
  initial begin
    rom[0] = 8'b00000000;
    rom[1] = 8'b00000010;
    rom[2] = 8'b00000100;
    rom[3] = 8'b00001000;
    rom[4] = 8'b00010000;
    rom[5] = 8'b00100000;
    rom[6] = 8'b01000000;
    rom[7] = 8'b10000000;
   end
```

```
-- Memória RAM em VHDL
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
ENTITY rom IS
    GENERIC (bits: INTEGER :=8;  -- Número de bits por
words
                words: INTEGER :=8); --Número de words na
mémoria
      PORT (Endereco : IN INTEGER RANGE 0 TO words-1;
             CLK: IN std_logic;
             Dados : OUT STD_LOGIC_VECTOR (bits-1 DOWNTO
0));
END rom;
ARCHITECTURE circ_rom OF rom IS
    TYPE vector_array IS ARRAY (0 TO words-1) OF
          STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
       CONSTANT memoria: vector_array := ( "00000000",
                                           "00000010",
                                           "00000100",
                                           "00001000",
                                           "00010000",
                                            "00100000",
                                            "01000000",
                                            "10000000");
BEGIN
    PROCESS (CLK)
       BEGIN
       IF (CLK ='0') THEN
       Dados <= memoria(endereco);
       END IF;
       END PROCESS;
```

APÊNDICE L – CÓDIGO DE UMA MEMÓRIA RAM COM 16 PALAVRAS DE 8 BITS , EM VERILOG E EM VHDL

```
// Memória RAM em Verilog
module ram1 #(             //-- Parametros
         parameter AW = 4,   //-- Bits de endereço)
         parameter DW = 8, /*-- Bits de dados por palavra
(Data witdh) */
         parameter WR= 16)  //-- Número de words na memoria
       (        //-- Portas
         input clk,                       //-- clock
         input wire [AW-1: 0] endereco,      //endereços
         input wire rw,                   /*-- Modo leitura
(1) o escritura (0) */
         input wire [DW-1: 0] dados_in,   /*-- Dados de
entrada */
         output reg [DW-1: 0] dados_out); /*-- Dato de
saída */
  //-- Memoria
  reg [DW-1: 0] ram [0: WR-1];
  //-- Leitura da memoria
  always @(posedge clk) begin
    if (rw == 1)
    dados_out <= ram[endereco];
  end

  //-- Escritura da memoria
  always @(posedge clk) begin
    if (rw == 0)
     ram[endereco] <= dados_in;
  end
```

```
-- Memória RAM em VHDL
LIBRARY IEEE;
USE ieee.std_logic_1164.all;

ENTITY ram1 IS
    GENERIC (bits: INTEGER :=8;  -- Número de bits por
words
                words: INTEGER :=16); --Número de words na
mémoria
      PORT ( wr_ena, clk: IN STD_LOGIC;
             Endereco : IN INTEGER RANGE 0 TO words-1;
                   Dados_in  : IN STD_LOGIC_VECTOR (bits-1
DOWNTO 0);
             Dados_out : OUT STD_LOGIC_VECTOR (bits-1
DOWNTO 0));
END ram1;

ARCHITECTURE circ_ram1 OF ram1 IS
    TYPE vector_array IS ARRAY (0 TO words-1) OF
          STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
       SIGNAL memoria : vector_array;
BEGIN
    PROCESS (clk, wr_ena)
       BEGIN
       IF (wr_ena='1') THEN
          IF (clk'EVENT AND clk='1') THEN
                memoria (endereco) <= Dados_in;
             END IF;
    END IF;
       END PROCESS;
       Dados_out <= memoria (endereco);
END;
```

ANEXO A – CÓDIGO DA MAQUINA DE ESTADOS DO TEMPORIZADOR DE UM LASER, EM VERILOG E EM VHDL

Exemplo retirado do Vahid (2008).

```
// Temporizador de laser em Verilog
module tem_laser (b, x, clk, rst);
    input b, clk, rst;
     output x;
      reg x;
       parameter E_desligado = 2'b00,
                 E_ligado1 = 2'b01,
                 E_ligado2 = 2'b10,
                 E_ligado3 = 2'b11;
       reg [1:0] estado_atual;
       reg [1:0] prox_estado;
       // procedimento para o registador de estado
       always @(posedge rst or posedge clk)
       begin
          if (rst ==1) // estado inicial
                estado_atual <= E_desligado;
             else
                estado_atual <= prox_estado;
       end
       always @ (estado_atual or b)
       begin
          case (estado_atual)
                E_desligado: begin
                      x <= 0; // laser desligado
                         if (b==0)
                            prox_estado <= E_desligado;
                         else
                            prox_estado <= E_ligado1;
                  end
```

```
// Continuação do temporizador de laser em Verilog
                E_ligado1: begin
                     x <=1; // laser ligado
                        prox_estado <= E_ligado2;
                  end

                  E_ligado2: begin
                     x <= 1; //laser ainda ligado
                        prox_estado <= E_ligado3;
                  end
                  E_ligado3: begin
                        x <= 1; // laser ainda ligado
                        prox_estado <= E_desligado;
                  end
            endcase
      end
endmodule
```

```
// Temporizador de Laser em VHDL
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY temporizador_lazer IS
    PORT ( b: IN std_logic;
           x: OUT std_logic;
                clk, rst: IN std_logic);
END temporizador_lazer;

ARCHITECTURE comportamento OF temporizador_lazer IS
    TYPE tipo_estado IS
           (E_desligado, E_ligado1, E_ligado2, E_ligado3);
       SIGNAL estado_atual, prox_estado : tipo_estado;

BEGIN
    regis_estado: PROCESS (clk, rst)
       BEGIN
          IF (rst='1') THEN -- estado inicial
                estado_atual <= E_desligado;
             ELSIF (clk='1' and clk'EVENT) THEN
                estado_atual <= prox_estado;
             END IF;
       END PROCESS;
logica_comb: PROCESS (estado_atual, b)
       BEGIN
          CASE estado_atual IS
                WHEN E_desligado =>
                      x <='0'; --lazer desligado
                      IF (b='0') THEN
                         prox_estado <= E_desligado;
                      ELSE
                         prox_estado <= E_ligado1;
                      END IF;
```

```
-- Continuação do temporizador de laser em VHDL
                    WHEN E_ligado1 =>
                       x<= '1'; --lazer ligado;
                          prox_estado <= E_ligado2;
                    WHEN E_ligado2 =>
                       x<='1'; --lezer ainda ligado
                          prox_estado <= E_ligado3;
                    WHEN E_ligado3 =>
                       x<= '1'; --lazer ainda ligado
                          prox_estado <= E_desligado;
              END CASE;
       END PROCESS;
END;
```