DO VHDL AO VHDL-AMS EM 1 DIA

Tiago da Silva Almeida e Rafael Lima de Carvalho

EDUFT
UNIVERSIDADE FEDERAL DO TOCANTINS

Tiago da Silva Almeida e Rafael Lima de Carvalho

DO VHDL AO VHDL-AMS EM 1 DIA

EDUFT
UNIVERSIDADE FEDERAL DO TOCANTINS

Palmas- TO
2020

Universidade Federal do Tocantins

Diagramação e capa: Gráfica Movimento
Arte de capa: Gráfica Movimento

Dados Internacionais de Catalogação na Publicação – CIP

A447v

Almeida, Tiago da Silva.
Do VHDL ao VHDL-AMS em 1 dia . / Tiago da Silva Almeida; Rafael Lima de Carvalho . – Palmas, TO: EDUFT, 2020.
85 p. : il. ; 21 x 29,7 cm.

ISBN 978-65-89119-01-2
Inclui referências.

1. Circuitos eletrônicos. 2. Projetos, execução. 3. Projetos, elaboração. 4. Operadores lógicos. 5. Modelagem sequencial. I. Rafael Lima de Carvalho. II. Título.

CDD – 025.3

SÚMARIO

1. Introdução à arte 11
1.1. Introdução ao processo de síntese 13
1.2. Síntese de circuitos eletrônicos e linguagens de descrição de hardware 17
1.3. Análise, elaboração e execução de projetos 19
1.4. Mais sobre projeto top-down 20

2. Introdução ao VHDL 24
2.1. Construção de modelos 24
2.1.1. Entidade 24
2.1.2. Arquitetura 25

3. Todo dado tem um tipo 27
3.1. Tipos enumerados 27
3.2. Tipos com intervalos 27
3.3. Tipos físicos 28
3.4. Tipo vector 28
3.5. Tipo enumerado definido pelo usuário 29
3.6. Tipo array 29
3.7. Subtipos 29

4. Operadores: o básico de tudo 30
4.1. Operadores lógicos 30
4.2. Operadores numéricos 31
4.3. Operadores relacionais 31
4.4. Operador de deslocamento 31
4.5. Operador de concatenação 32

5. Toda linha é concorrente 33
5.1. Declaração de sinais concorrentes 33
5.2. Declarações condicionais 33
5.3. Atribuição de sinais com select 34
5.4. Atribuição de sinais com atraso 35
5.4.1. Atraso inercial 35
5.4.2. Atraso de transporte 35

6. Projeto hierárquico 36
6.1. Componentes 36

7. Abstraindo: modelagem sequencial 37
7.1. Básico de processos 37
7.1.1. Lista de sensibilidade 37
7.1.2. Declaração wait 38
7.1.3. Declaração de sinais sequenciais 39
7.1.4. Variáveis 39
7.2. Programação condicional 40
7.2.1. Declaração if / then 40
7.2.2. Declaração case 41
7.2.3. Laços infinitos 41
7.2.4. Laço while 42
7.2.5. Laço for 42
7.3. Atributos de sinais 43

8. Aproveitando as facilidades: pacotes 45
8.1. O pacote STD_LOGIC_1164 45
8.1.1. Resolução de funções 46
8.1.2. Operadores lógicos 47
8.1.3. Funções de detecção de borda 47
8.1.4. Funções de conversão de tipos 47
8.2. O pacote NUMERIC_STD 48
8.2.1. Funções aritméticas 48
8.2.2. Funções lógicas 49
8.2.3. Operador de comparação 49
8.2.4. Funções de detecção de borda 49
8.2.5. Funções de conversão 49
8.2.6. Casting de tipos 50
8.3. Outros pacotes 50
8.3.1. O pacote numeric_std_unsigned 50
8.3.2. O pacote math_real 51
8.3.3. O pacote math_complex 53

9. Introdução ao VHDL-AMS 55
9.1. Domínios e níveis de modelagem 56

10. Sinais de outras naturezas 58
10.1. Quantidades livres 59
10.2. Quantidades ramificadas e terminais 59
10.3. Declaração break e descontinuidade 60
10.4. Especificação de limite de passos 60
10.5. Procedimentos 61
10.5.1. Declaração return em procedimentos 61
10.5.2. Parâmetros de procedimentos 61
10.6. Funções 62
10.6.1. Função now 62
10.7. Declaração procedural simultânea 63

11. Modelagem baseada em frequência 64
11.1. Quantidades de fonte espectral 64
11.2. Modelagem de ruídos 65
11.3. Função de transferência de Laplace 65
11.4. Amostras e funções de transferência discreta 66
11.4.1. Amostras de ordem zero (zoh) 66
11.4.2. Função de transferência z 66
11.5. Sinais resolvidos básicos 67

12. Mais sobre estruturas 68
12.1. Instancias de configuração de componentes 68
12.2. Estrutura iterativa generate 69
12.3. Estrutura generate condicional 70
12.4. Declaração generate de configurações 70
12.5. Sinais guardados e desconexão 71
12.6. Tipos de acesso 71
12.7. Estruturas de dados ligadas 72

13. Mais sobre atributos pré-definidos 73
13.1. Atributos definidos pelo projetista 77
13.2. Grupos 78
13.3. Processos adiados 78

14. Simulação robusta: arquivos 80
14.1. O pacote textio 80

15. Verificação com *testbench*. *83*
15.1. Checagem automática. 83
15.1.1. Declaração report. 83
15.1.2. Declaração assert. 84
15.2. Usando entrada e saída 84

Referências bibliográficas 85

Prefácio

Linguagens de descrição de hardware são elementos indispensáveis em qualquer projeto ou pesquisa sobre hardware. E muitas vezes é trabalhoso aprender a gramática de uma nova linguagem, ou mesmo nos esquecemos de certos detalhes. Pensando nisso, decidimos produzir esse material para auxiliar os iniciantes, ou mesmo servir como material de consulta para usuários intermediários.

O VHDL é a escolha mais óbvia para ser aprendida devido a sua simplicidade de sintaxe e sua ampla utilização nas universidades e centros de pesquisa. Já o VHDL-AMS é uma extensão muito poderosa da linguagem VHDL e não poderia ser deixado de lado nessa obra.

Claro que todo o contexto de projeto é demasiado complexo para ser apresentado em uma obra tão diminuta. Contudo, é indispensável que façamos uma breve introdução sobre o assunto, o qual é apresentado no Capítulo 1.

Já no Capítulo 2 começamos a explorar, de maneira "*straightforward*" a sintaxe do VHDL, a começar pela estrutura básica de qualquer arquivo VHDL ou VHDL-AMS, pelos tipos de dados básicos (Capítulo 3) e pelos operadores básicos das linguagens (Capítulo 4). Lembrando mais uma vez, que o VHDL-AMS é uma extensão, automaticamente as mesmas regras se aplicam a ele.

Linguagens de descrição de hardware, como VHDL e VHDL-AMS, não trabalham da mesma forma que uma linguagem de programação tradicional. O que nos leva às descrições do Capítulo 5, deixando claro que naturalmente toda descrição é concorrente, dado a natureza física do hardware.

Felizmente, as linguagens não são limitadas à concorrência, e modelos abstratos e sequenciais podem ser facilmente criados, conforme a sintaxe apresentada no Capítulo 7. Além da criação de modelos hierárquicos em grandes projetos, o que pode ser feito aproveitando modelos existentes e outras facilidades com a utilização de componentes ou pacotes, descritos no Capítulo 6, 8 e 12.

As descrições feitas em VHDL são totalmente válidas para o VHDL-AMS, mas o inverso não é verdadeiro. A natureza do modelo é demasiadamente complexa e diferente, se comparada ao VHDL. As diferenças podem ser percebidas com a leitura do Capítulo 9, e já no Capitulo 10 e 11, podemos notar um pouco dessa complexidade com a descrição de tipos escalares, com outras naturezas e a modelagem de sistemas baseados em frequência.

Os Capítulos 12 e 13 destacam formas mais elaboradas para reutilização de código voltado para o VHDL-AMS, que além disso, ainda possui alguns atributos inexistentes ao VHDL descritos no Capítulo 10.

Para ambas as linguagens (VHDL e VHDL-AMS), a manipulação de arquivos, descrita no Capítulo 14, é indispensável para simulação de modelos mais complexos. Arquivos externos ao modelo são utilizados somente na simulação, já que o processo de síntese para uma tecnologia alvo é inviável. Esses modelos de simulação só são possíveis utilizando de *testbench*, descrito no Capítulo 15.

Essa obra foi feita com base principalmente em dois livros muito bons: *Quick start guide to VHDL* de B. J. Lameres, de 2019 e *System Designers Guide to VHDL-AMS* de J. P. Ashenden, G. D. Peterson e D. A. Teegarden, de 2003. O livro de B. J. Lameres é bastante objetivo e sucinto, a maneira como acreditamos que nossa obra também deve ser. E o livro de J. P. Ashenden, G. D. Peterson e D. A. Teegarden é uma obra incrível e completa. Qualquer leitor que queria se aprofundar no assunto deve estudar com afinco esse livro.

O "aprender em 1 dia" é uma provocação e um desafio ao leitor. Consideramos totalmente plausível o aprendizado em um dia, desde que haja foco e empenho nesse único dia. Ainda pensando na compactação do conteúdo, decidimos ilustrar somente as regras de sintaxe das linguagens e deixamos exemplos fora dessa obra. Essa não foi uma decisão fácil, já que o sentimento do escritor é de que algo está por fazer. E quando nos voltamos ao objetivo central, percebemos que era a decisão a ser feita.

Mas esse aprendizado em um dia será mais fácil se algumas disciplinas já forem familiarizadas ao leitor, como por exemplo (não exatamente nessa ordem): Sistemas Digitais I e II; Arquitetura de Computadores; Física I, II e III; Processamento Digital de Sinais; Cálculo; Lógica de Programação; Estrutura de Dados I e II; possivelmente Compiladores.

Pode parecer muita coisa, mas são disciplinas bases em qualquer curso de Engenharia da Computação, Engenharia Elétrica, Engenharia Mecatrônica, Engenharia Eletrônica, e é claro, Ciência da Computação. São para esses estudantes que essa obra é voltada. Ligue seu cronômetro, pare para as refeições e boa leitura.

1. INTRODUÇÃO À ARTE

A linguagem *VHSIC* (*Very-High-Speed Integrated Circuit*) *Hardware Description Language*, ou simplesmente VHDL[1], é uma linguagem de descrição de hardware muito comum na academia devido a sua simplicidade sintática. Se voltarmos à década de 1980, é bastante compreensível sua simplicidade, já que ela foi originalmente pensada como uma forma de documentação, ou uma linguagem única de comunicação, dos projetos de hardware[2].

Ao mesmo tempo, surge sua principal concorrente, a linguagem Verilog, a qual vem com um propósito até então novo: a sintetização de hardware. Logo, o VHDL também foi incorporado aos sintetizadores modernos, contudo, sem perder a sua simplicidade.

Mas afinal, qual o propósito de uma linguagem de descrição de hardware? A reposta já está na pergunta, ou seja, descrever o funcionamento de um certo hardware em um determinado projeto. Apesar de inicialmente simples, a resposta é mais complicada do que parece. Esse livro não tem o objetivo de abordar todos os pormenores envolvidos no contexto. E, sim, resumir a gramática da linguagem para que esse material sirva de guia e consulta para iniciantes ou mesmo usuários intermediários da linguagem.

Exploremos inicialmente a noção de hardware, isto é, todo elemento físico envolvido na computação moderna. Essa noção é bastante ampla e é evidente que o VHDL tem um foco. VHDL é amplamente utilizado na descrição, simulação e síntese de CPLDs (*Complex Programmable Logic Device*), FPGAs (*Field Programmable Gate Array*) e ASICs (*Application Specific Integrated Circuits*). Esses dispositivos são usados em projetos específicos e robustos (muitas vezes aplicações militares, aeroespacial etc.), diferentemente do hardware utilizado em computadores pessoais e celulares, que são pensados para serem genéricos e atenderem todo tipo de aplicação (como navegação na Internet, edição de textos, jogos etc.). Claro que o VHDL também pode ser utilizado para descrever o processador i7 da Intel ou ARM A9 da Qualcom, por exemplo.

A grande diferença dos dispositivos CPLDs e FPGAs[3] de um dispositivo de hardware tradicional é que estes dispositivos são reprogramáveis. Primeiro, o que significa um hardware reprogramável? Um computador tradicional também não é programável e reprogramável?

Para entender melhor o assunto, imaginemos o funcionamento de um computador tradicional. O processador i7, por exemplo, foi fabricado para que ele consiga “entender” as instruções que chegam até ele e agir de acordo com essas instruções. As instruções de ação são os programas de um computador. Cada programa (software) consiste de um conjunto de instruções do processador para executar ações específicas. Não é objetivo deste livro entrar a fundo na organização e arquitetura de computadores. Contudo, é importante que o leitor tenha em mente que, independentemente da finalidade do programa, o processador é o mesmo, não há alterações no seu funcionamento ou nos recursos internos que ele possui, ele foi fabricado dessa forma.

Já um dispositivo reprogramável é diferente. Ele possui um conjunto grande de conexões internas no formato de matriz (por exemplo, uma FPGA). Essa matriz de conexões permite que as ligações internas do hardware possam ser alteradas, desfeitas, ou mesmo feitas novas

conexões, para que novas construções ou novos hardwares, completamente diferentes, sejam criados. Ou seja, a noção tradicional de programação não é mais a mesma.

Mas como isso é feito? É nesse momento que entram as linguagens de descrição de hardware, como o VHDL, e os sintetizadores, ou ferramentas de síntese (que são ferramentas de software). A síntese é o processo de modelagem de um projeto em alto nível de abstração (uma descrição em VHDL, por exemplo) e a posterior transformação, ou tradução, para um nível mais baixo de abstração (por exemplo, a implementação física do projeto VHDL em FPGA).

E a linguagem VHDL-AMS (*Analog and Mixed Signal*)? O VHDL-AMS é uma extensão da linguagem VHDL, com exatamente o mesmo objetivo, com a adição de mais formas de representação de hardware do que simplesmente hardware digital. Com o VHDL-AMS é possível descrever, simular e sintetizar hardware no nível elétrico, mecânico, pneumático, rotacional etc.. Contudo, nem sempre o processo de síntese é simples ou mesmo possível com o VHDL-AMS. Mas ele continua sendo uma ferramenta muito importante na modelagem e simulação de hardware.

Dessa forma, o VHDL se mostra uma ferramenta muito importante dentro de todo esse processo, e dentro da ampla gama de modelagens e ferramentas que podem ser utilizadas em um projeto de hardware. Como esse material é introdutório, façamos ainda uma breve introdução do processo de síntese, antes de adentrar na sintaxe do VHDL, que é o objetivo principal deste livro.

1.1. Introdução ao processo de síntese

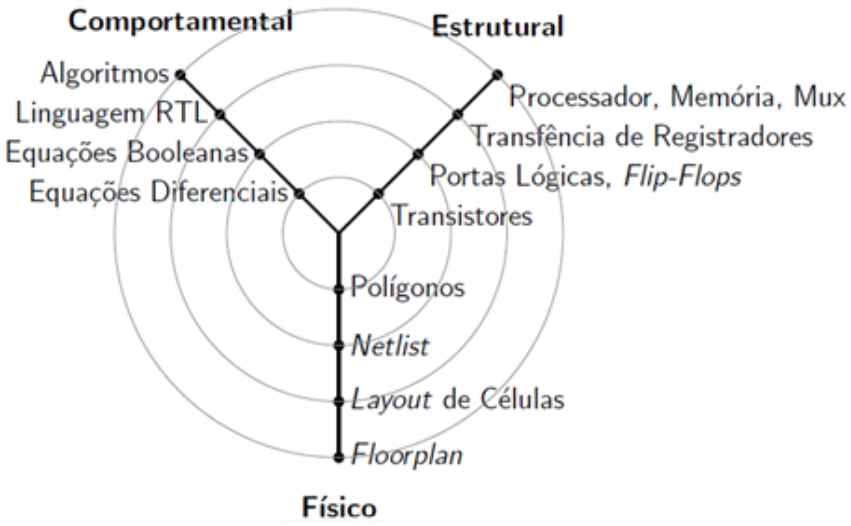
Em projeto de hardware, as fases de concepção, análise, implementação e testes são importantes para o desenvolvimento de tecnologias de qualidade. Um projeto de hardware elaborado de forma coerente permite a redução significativa de tempo, custo monetário e intelectual, ou mesmo de situações de alto risco, como falhas em sistemas críticos.

A sistematização das etapas de projeto nas fases de concepção, análise, implementação e testes é de vital importância para redução de custos. Nesse sentido, alguns trabalhos representaram um marco na sistematização de projetos de sistemas eletrônicos[4]. Dentro da área de Projeto de sistemas eletrônicos, é importante citar o trabalho de Gajski e Kuhn (1983). Dadas as diferentes representações de um determinado sistema eletrônico, o projeto pode ser dividido em três níveis: **comportamental**, **estrutural** e **físico**.

Os níveis de representação do sistema podem ser apresentados como eixos, onde cada eixo é dividido em níveis de abstração (cada nível de abstração sendo uma representação diferente do projeto). O eixo **comportamental** pode ser dividido do nível mais alto para o mais baixo, em: Algoritmos, Linguagem RTL (*Register Transfer Logic*), Equações Booleanas e Equações Diferenciais. O eixo **estrutural** pode ser divido em: Processador, Memória, Multiplexadores; Transferência de Registradores; Portas Lógicas, Flip-flops e Transistores. O eixo **físico** pode ser dividido em: *Floorplan*, *Layout* de Células, *Netlist* e Polígonos. Deixemos claro ao leitor que não iremos nos aprofundar em cada uma das representações.

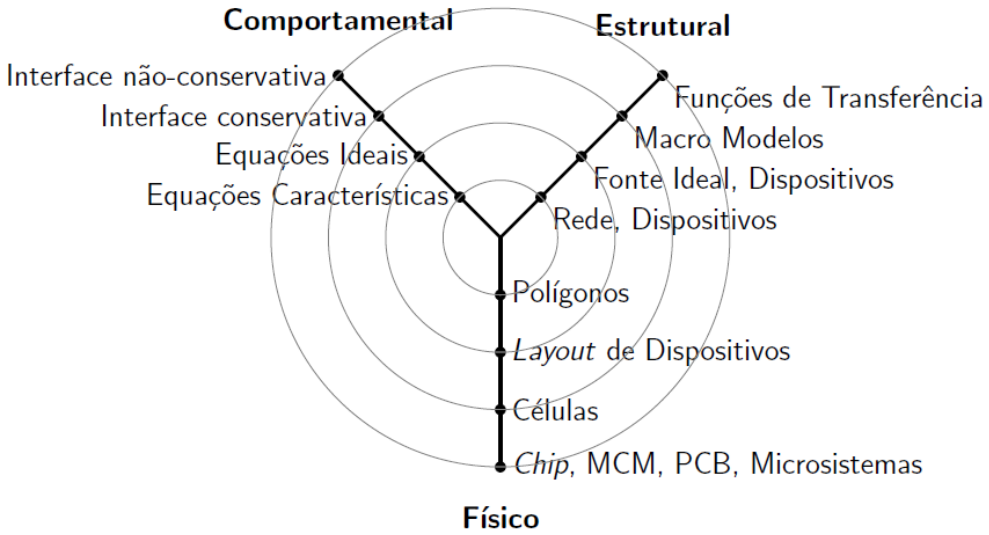
Gajski e Kuhn (1983) sistematizaram os projetos de sistemas eletrônicos de acordo com o contexto da época. Assim, Riesgo, Torroja e Torre (1999) atualizaram as etapas de projeto para as novas representações que surgiram ao longo dos anos, do qual concebeu-se a Figura 1.

Figura 1- Diagrama Y Digital.



A Figura 1 é conhecida como "Diagrama Y", dado o seu formato. De acordo com o que já foi discutido, conclui-se que o Diagrama Y da Figura 1 sistematiza as etapas de projetos de sistemas eletrônicos (hardwares) digitais. Entretanto, é possível obter a mesma sistematização para projeto de sistemas eletrônicos analógicos (ou com representação de diferentes domínios de energia), o qual é ilustrado na Figura 2. Os eixos também são divididos em: **comportamental**, **estrutural** e **físico**.

Figura 2 - Diagrama Y Analógico.



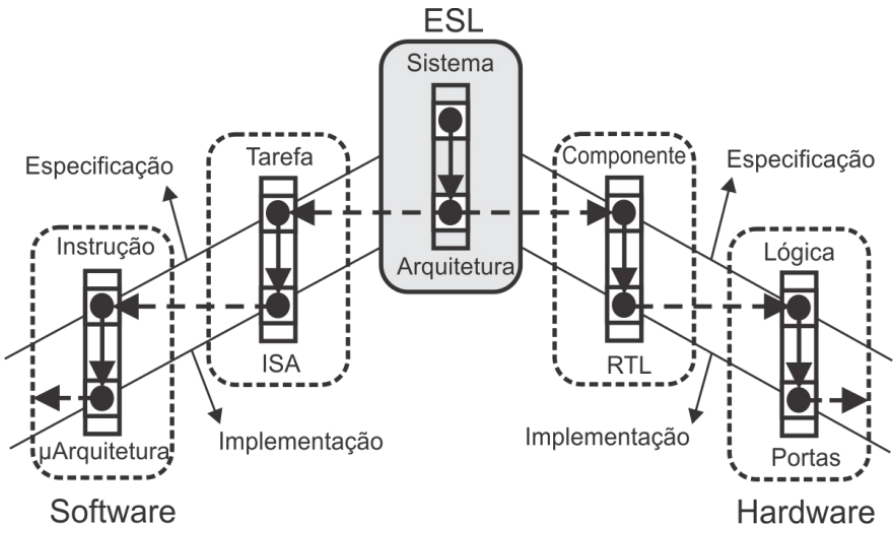
O eixo **comportamental** pode ser dividido do nível mais alto para o mais baixo em: Interface não-conservativa; Interface conservativa; Equações Ideais e Equações Características. O eixo **estrutural** pode ser dividido em: Funções de Transferência; Macro Modelos; Fonte Ideal e Dispositivos; e Rede e Dispositivos. O eixo **físico** pode ser dividido em: Chip, MCM (*Multi--Chip Module*), PCB (*Print Circuit Board*), Microsistemas; Células; *Layout* de Dispositivos e Polígonos.

A diferença entre as duas representações apresentadas nas Figura 1 e Figura 2 é mais significativa nos eixos comportamentais e estruturais, devido os eixos representarem um nível de abstração mais alto comparado ao eixo físico. Maiores detalhes sobre as formas de representação podem ser encontrados no livro de Ashenden, Peterson e Teegarden (2003).

A representação em níveis muito abstratos dificulta a implementação e fabricação de dispositivos eletrônicos. Por isso, deu-se origem a metodologia conhecida como *top-down*, descrita por Gajski e Kuhn (1983) e Riesgo, Torroja e Torre (1999). As etapas dessa metodologia são conhecidas como "processo de síntese", onde a partir de um determinado nível de abstração é obtido um nível mais baixo de abstração com a representação equivalente. O processo de síntese também se aplica a mudança de eixo, seja, do eixo comportamental para o estrutural ou do eixo estrutural para o físico.

Mesmo com uma metodologia de elaboração e análise de projetos sendo bem definida e atualizada por Riesgo, Torroja e Torre (1999), eventualmente, outras formas de representação surgiram para tentar automatizar os projetos de sistemas eletrônicos. Assim, Gerstlauer et al. (2009) propuseram uma generalização entre as formas de representação e síntese de sistemas eletrônicos. Essa generalização unifica as Figuras 1 e 2, as quais podem ser representadas pela Figura 3.

Figura 3 - Fluxo de projeto de sistemas eletrônicos, também conhecido como modelo de telhado duplo (*Double Roof Model*).



A Figura 3 generaliza e adiciona um novo paradigma aos projetos de sistemas eletrônicos. Essa adição é a representação em **software**[5], contudo, ainda existe a representação em **hardware**. A representação em **software** é mais evidente em sistemas embarcados[6] ou sistemas de aplicação específica, pois, atualmente existem muitas ferramentas computacionais proprietárias para automação de projetos. Dessa forma, não é necessária a construção do hardware, já que, muitas vezes ele é reprogramável ou mesmo um microprocessador convencional adicionado ao projeto como um todo.

A etapa de projeto representada pelo modelo da Figura 3 começa com uma especificação em nível de **ESL** (*Electronic System Level*). O nível **ESL** é um modelo comportamental que muitas vezes é algum tipo de rede de processos de comunicação. Além disso, é fornecido um

conjunto de restrições de mapeamento e restrições de execução (área máxima, o rendimento mínimo, etc.).

O modelo **ESL** é tipicamente um modelo estrutural constituído pelos componentes de arquitetura, tais como processadores, barramentos, memórias e aceleradores de hardware. A tarefa de síntese ESL é, então, o processo de seleção de uma arquitetura em uma plataforma apropriada, determinando o mapeamento do modelo comportamental em uma arquitetura, e gerando a correspondente implementação do comportamento executado na plataforma.

O resultado é um modelo refinado contendo todas as decisões de projeto e métricas de qualidade, tais como latência ou área. Se selecionado, os componentes deste modelo refinado são então usados como entrada para as etapas de projeto em níveis mais baixos de abstração, onde cada processador de hardware ou software é ainda implementado separadamente (GERSTLAUER et al., 2009).

Os processos de síntese em níveis mais baixos de abstração são semelhantes, em que uma especificação comportamental é refinada em uma especificação estrutural de nível mais baixo. No entanto, dependendo do nível de abstração, a granularidade de objetos manipulados durante a síntese difere e algumas tarefas podem ser mais importantes do que outras. Por exemplo, no nível da **tarefa** no lado do **software**, comunicação de processos ou de *threads* ao mesmo processador devem ser traduzidos para a arquitetura do conjunto de instruções (ISA - *Instruction-Set Architecture*) do processador.

Esta tarefa de síntese em **software** é tipicamente realizada utilizando um compilador e ferramentas de ligação[7] para o processador selecionado. No nível de **instrução**, o conjunto de instruções de processadores programáveis é então realizado em hardware através da aplicação da microarquitetura subjacente. Esta etapa resulta em um modelo estrutural da organização do processador, geralmente especificados como uma descrição RTL (GERSTLAUER et al. ,2009).

Por outro lado, no nível de **componente** do lado do **hardware**, processos selecionados para serem implementados como aceleradores de hardware são sintetizados para uma descrição RTL sob a forma de controladores de máquinas de estado finitos que direcionam um caminho de dados consistindo de unidades funcionais, memórias e interconexões. Este passo de refinamento é comumente conhecido como síntese comportamental. Finalmente, no nível da **lógica**, a granularidade dos objetos considerados durante a síntese lógica corresponde às expressões booleanas implementadas por portas lógicas e *flip-flops* (GERSTLAUER et al. ,2009).

Uma observação importante que pode ser feita a partir da Figura 3 é que, ao nível RTL, hardware e software se unem de novo em um processo lógico do projeto para a saída final (fabricação). Além disso, nota-se que um processo *top-down* de projeto ESL depende da disponibilidade dos fluxos de projeto, componente ou tarefa (e, eventualmente, lógica e instrução). Fluxos de nível mais baixo podem ser fornecidos tanto na forma de ferramentas de síntese correspondentes ou de componentes pré-concebidos para ser conectada a arquitetura do sistema.

A revolução tecnológica representada pela integração em muito larga escala abriu novas perspectivas nas metodologias de projeto digital e de sinais mistos. O tamanho e a complexidade de sistemas, como VLSI (*Very Large Scale Integration*) e ULSI (*Ultra Large Scale Integration*), aumentam a necessidade da eliminação de operações manuais e repetitivas, motivando o desen-

7 Ligação refere-se ao papel do "linker" em compiladores, utilizado para juntar bibliotecas externas e internas com o software do usuário para a criação de um único conjunto de instruções do processador.

volvimento de sistemas automáticos de projetos. Para a realização desses sistemas automáticos, o entendimento dos problemas e dos processos de um projeto são fundamentais. A automação de um determinado projeto requer a análise de seus algoritmos e que eles sejam rapidamente e facilmente implementados, conforme descrito por Breuer, Sarrafzdeh e Somenzi em 2000, o que continua sendo válido nos dias atuais.

A utilização da metodologia *top-down* em projetos de sistemas eletrônicos é um aspecto importante do desenvolvimento de projetos. Ela permite uma especificação rápida e eficiente, uma modelagem das funcionalidades do projeto em uma linguagem de descrição adequada, uma verificação por meio de simulação e a síntese do modelo em uma tecnologia alvo (RIESGO; TORROJA; TORRE, 1999; GAJSKI; KUHN, 1983).

A metodologia *top-down*, descrita inicialmente por Gajski e Kuhn (1983), diz que devido à complexidade de muitos projetos, ele pode ser dividido em subprojetos mais simples. A análise individual de cada um fornece novas soluções para o problema original como um todo e cada subproblema pode ser eficientemente resolvido e as soluções devem ser combinadas de forma eficaz. Esse fato deixa claro a necessidade do desenvolvimento de bibliotecas, dividindo o projeto em subprojetos, permitindo também a reutilização destes subprojetos em diferentes sistemas.

Alguns problemas de otimização de circuitos eletrônicos são descritos como não polinomiais completos, i.e., a solução ótima não é encontrada em tempo polinomial. O que sugere fortemente que certos problemas são muito difíceis de resolver eficientemente.

Segundo D'Amore (2005) e Riesgo, Torroja e Torre (1999), mesmo a representação RTL sendo um nível mais baixo de abstração, o modelo RTL não se prende à nenhuma tecnologia alvo. Evidentemente, tanto o VHDL quanto o VHDL-AMS se encaixam nesse modelo de representação. Como será vista na descrição da sintaxe das linguagens, diferentes níveis de abstração podem ser usados na descrição do sistema.

1.2. Síntese de circuitos eletrônicos e linguagens de descrição de hardware

Segundo D'Amore (2005), a linguagem VHDL ou VHDL-AMS permite que um mesmo circuito seja descrito em diversos níveis de abstração. O código gerado inicialmente pode conter estruturas muito abstratas que não permitem a síntese em uma primeira etapa.

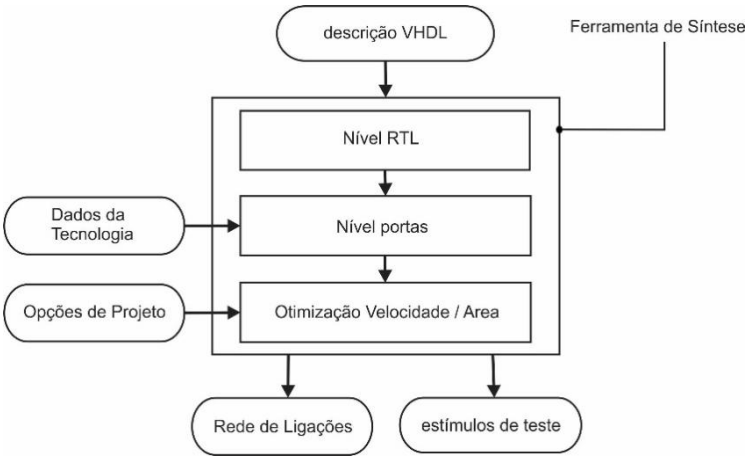
De um modo geral, as ferramentas de síntese realizam um processo iterativo de simulações e detalhamento das estruturas até atingir uma descrição que permita a síntese. Essas ferramentas fazem a verificação de erros de sintaxe da linguagem e posteriormente executa a inferência e interligação das estruturas necessárias para o circuito ser gerado a partir da descrição. Dessa forma, é gerado o circuito no nível RTL, empregando primitivas disponíveis na ferramenta como comparadores, somadores, registradores e portas lógicas. Nesse ponto, o circuito não está associado a nenhuma tecnologia de fabricação e não está necessariamente otimizado.

Em seguida, é gerado um novo circuito a partir da estrutura RTL empregando elementos disponíveis na tecnologia empregada na fabricação, como elementos lógicos disponíveis na tecnologia alvo, sendo necessário especificar o dispositivo para a realização

da síntese. Uma tentativa de minimização é executada nessa etapa levando em conta dois fatores conflitantes, a otimização de custo e de velocidade.

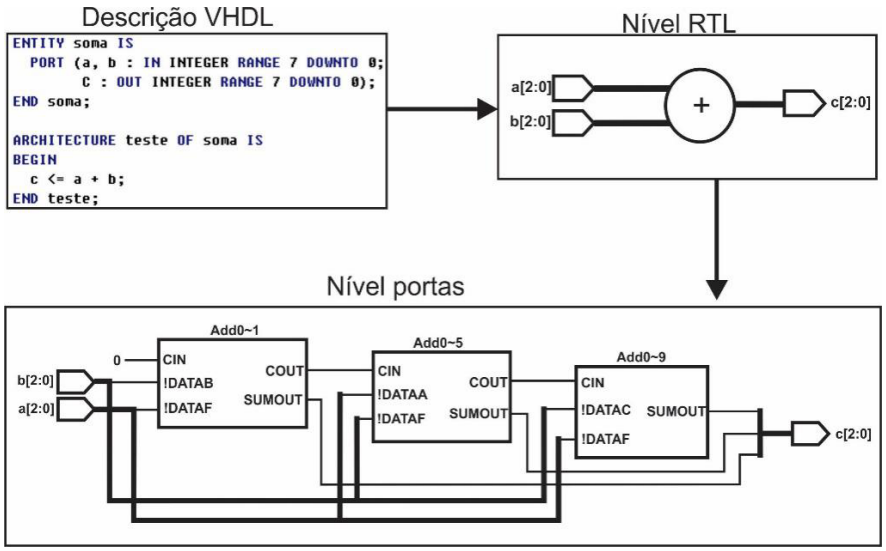
A Figura 4, retirada de D'Amore (2005), ilustra os passos realizados por muitas ferramentas de síntese da atualidade. Onde, após a otimização de velocidade e custo, a ferramenta gera o posicionamento e as interligações das primitivas geradas para o projeto. Nesse ponto, os circuitos são estimulados por testes para verificação da estrutura em relação a temporização (D'AMORE, 2005).

Figura 4 - Diagrama funcional dos passos realizados pelas ferramentas de síntese em geral.



Como exemplo, a Figura 5, também retirada de D'Amore (2005), ilustra o trabalho realizado pelas ferramentas de síntese em geral. Nesse exemplo, trata-se de um circuito somador de valores inteiros de 0 a 7, ou seja, um somador de três bits sem *carry in* e descartando o *carry out*.

Figura 5 - Exemplo de passos executados pelas ferramentas de síntese a partir de uma descrição na linguagem VHDL.



A partir da descrição em VHDL a ferramenta de síntese gera uma representação equivalente em nível RTL. Uma forma de evitar confusões sobre a representação, já que realmente

não possuem registradores nesse exemplo, é imaginar que o circuito de soma como um componente genérico. Evidentemente, esse componente possui as entradas e saídas descritas no modelo VHDL. Em seguida é construção um novo modelo em nível de portas, não necessariamente portas lógicas, como pode ser observado na Figura 5. O modelo de portas diz respeito aos elementos físicos disponíveis na tecnologia alvo de implementação, como um FPGA, por exemplo. O que pode realmente ser portas lógicas, como elementos pré-fabricados de soma disponíveis na tecnologia.

1.3. Análise, elaboração e execução de projetos

Uma das principais razões para escrever um modelo de um sistema é para nos permitir simulá-lo. A tarefa de modelagem e simulação envolve três estágios: análise, elaboração e execução. A Análise e a elaboração também são necessárias na preparação do modelo para outros fins, como por exemplo, a síntese.

Durante a fase de análise, a descrição em VHDL (ou VHDL-MAS) é examinada e erros semânticos são localizados. Todo o modelo de um sistema não precisa ser analisado de uma vez. É possível analisar unidades do projeto, como declarações de entidades e corpo de arquitetura separadamente. Se o analisador não encontra erros na unidade de projeto, ele cria uma representação intermediária da unidade e armazena-a em biblioteca.

A fase de elaboração atua através da hierarquia do projeto criando todos os objetos definidos nas declarações. O resultado da elaboração do projeto é uma coleção de processos interconectados por redes com a possibilidade de cada processo conter variáveis. Um modelo deve ser redutível para uma coleção de processos, sinais e expressões características para simulá-lo (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Na fase de execução do modelo, a passagem do tempo é simulada em passos discretos dependendo de quando um evento ocorra. Em algum tempo de simulação um processo pode ser estimulado pela mudança no valor do sinal ao qual ele é sensível. O processo é retornado e pode alocar novos valores para o sinal em um instante de tempo posterior. Isso é chamado de agendamento de transição de sinal. Se o novo valor for diferente do valor anterior, um evento ocorre, e outro processo sensível ao sinal pode ser executado (ASHENDEN, PETERSON e TEEGARDEN, 2003). Lembrando que nos referimos a um modelo simulado e o tempo de simulação, também tempo de simulação delta, é um modelo de tempo infinito e em cada instante de tempo existe mais infinitos instantes tempo. Toda essa lógica de tempo de simulação também pode variar ou ser limitada de acordo com a ferramenta de síntese utilizada[8].

Porções analógicas do sistema simulado são avaliadas por ferramentas de solução analógica incorporadas aos sintetizadores em tempo contínuo (no caso do VHDL-AMS). A ferramenta de solução usa três conjuntos de equações: equações estruturais, equações explícitas e equações com argumentos (ASHENDEN, PETERSON e TEEGARDEN, 2003). Esse ponto é basicamente o trabalho realizado pelos analisadores semânticos convencionais

8 Nessa obra optamos por não nos referir a nenhuma ferramenta de síntese especifica, mesmo existindo grandes variações de comportamento de uma para outra. Algumas até nem suportam todas as especificações disponíveis no VHDL ou VHDL-AMS, sendo também conhecido como um dialeto das linguagens.

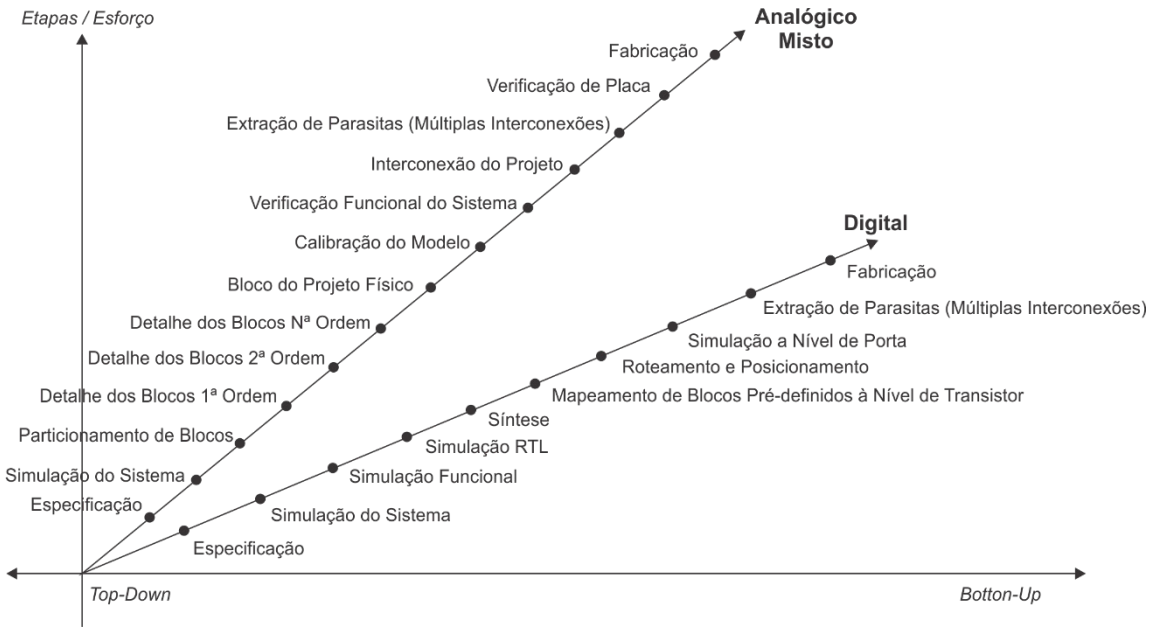
1.4. Mais sobre projeto *top-down*

A metodologia de projeto *top-down* tem um impacto profundo no projeto do sistema digital, por permitir a realização da especificação de forma rápida e eficiente, o projeto, a sintetização e verificação de projetos prontos até a fase de fabricação. Ao analisarmos o número de blocos de construção digitais necessários para a síntese, notamos que é relativamente pequeno, afinal a escala de valores possíveis é pequena. O mesmo não pode ser dito para o projeto analógico, projeto de sinais mistos e tecnologias mistas. Os blocos de construção para estes domínios de projeto são muito mais numerosos e sofisticados (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Até onde é possível afirmar, não existem ferramentas de síntese abrangentes para cada domínio analógico de projeto. No entanto, tem-se algumas ferramentas para aplicações específicas, como por exemplo o projeto de filtros de sinais. Tais ferramentas são semelhantes às ferramentas de síntese digital em que começam com uma especificação do comportamento de um sistema, contam com topologias de circuito fixas e constroem blocos RTL para implementação, são redirecionadas para diferentes tecnologias, tais como PSoCs (*Programmable System-on-Chip*), e os resultados são verificáveis por meio da simulação (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Na ausência de ferramentas de síntese mais generalizadas, evidentemente devemos recorrer ao refinamento manual do projeto. Nesse caso, um fluxo de projeto *top-down* é ainda mais importante para gerenciar a complexidade. A Figura 6 mostra a comparação de um fluxo de projeto digital com base no projeto *top-down* e na verificação de *botton-up*[9] com um possível fluxo para projeto analógico ou de sinal misto. Existem muitos paralelos e várias diferenças significativas entre os dois fluxos da Figura 6, onde o esforço ou quantidades de etapas são maiores no processo de projeto analógico ou de sinais mistos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Figura 6 - Representação do esforço exigido em projetos de sistemas analógicos em contraste com sistema digitais.



Adicionalmente, algumas boas práticas são recomendadas, entre elas surgem conceitos como **gerenciamento de configuração**, onde um indivíduo ou grupo gerencia e controla alterações no projeto. Isso garante um ambiente estável e previsível para o desenvolvimento de novos trabalhos, sem confusão causada por alterações simultâneas feitas por outros desenvolvedores (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O gerenciamento de configuração também inclui o conceito de **controle de revisão**, permitindo que uma sequência histórica de alterações de projeto seja capturada, gerenciada e, se necessário, recuperada (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Finalmente, o gerenciamento de configuração inclui a noção de integração ordenada e mesclagem de mudanças em um fluxo principal de trabalho cuidadosamente controlado, destinado a se tornar o produto entregue. Normalmente, a integração é feita por um painel de controle de alterações, que muitas vezes é feita por ferramentas automáticas de gerenciamento de projetos e versões, semelhante a plataforma Github[10].

Voltando à Figura 6, a primeira etapa do desenvolvimento do produto é uma definição clara dos requisitos do sistema e uma arquitetura do sistema que os implemente. Muitos projetos usam documentos informais de especificação. A manutenção de documentos à medida que o projeto evolui pode ser difícil, principalmente em projetos grandes ou complexos com requisitos em mudança. Além disso, documentos informais não são eficazes para comunicar alterações nas especificações ou para validar projetos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Uma abordagem alternativa é usar um modelo de arquitetura do sistema como ponto de partida para uma metodologia *top-down*. O modelo captura a estrutura e a funcionalidade do sistema proposto e atua como uma especificação executável. Ele define a decomposição do sistema em blocos funcionais e especifica como os blocos devem funcionar juntos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Depois de desenvolvermos o modelo de arquitetura do sistema, podemos dividir o sistema em partes que podem ser atribuídas a indivíduos ou equipes de projeto para aprimoramento. Particionar o projeto fornece um processo metódico para conduzir o desenvolvimento em paralelo, o que é particularmente importante para grandes projetos. Além disso, esta divisão também ajuda a identificar oportunidades para reutilizar os projetos existentes e / ou criar projetos reutilizáveis para o futuro (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Se os blocos de um sistema forem projetados em paralelo, é muito importante que as interfaces entre os blocos sejam bem definidas. A definição da interface é auxiliada por um modelo de projeto de sistema em evolução, baseado no modelo de arquitetura do sistema, com os detalhes do projeto adicionados à medida que se tornam disponíveis. O modelo de projeto do sistema e os casos de teste de simulação fornecem o contexto que permite o desenvolvimento independente de blocos. Cada equipe pode conectar seu trabalho ao modelo e receber um retorno fácil do comportamento do sistema, erros etc. (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Uma vez definidos os blocos básicos do sistema, estes podem ser explorados e projetados em detalhes, resultando nas informações necessárias para a implementação física dos blocos. A interconexão entre os blocos ainda pode ser considerada ideal neste momento, a menos que já exista conhecimento dos principais problemas de interconexão (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O refinamento de um bloco envolve a adição de modelos que fornecem progressivamente mais informações sobre o mesmo. Adicionalmente, podem ser tomadas decisões concretas que, em última análise, permitem que o projeto seja fabricado. As decisões de projeto são testadas, primeiro apenas no contexto restrito do bloco e depois no contexto maior do modelo do sistema. As decisões podem ser a edição de alterações no projeto, substituições de componentes ou alterações de valores de componentes. Equações em um modelo analógico ou lógica em um modelo digital podem ser editadas para alterar um algoritmo específico do sistema. Um modelo inteiramente novo pode ser criado, representando a adoção de uma nova abordagem para a implementação do bloco (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Todas essas mudanças estão nas revisões dos modelos, geradas automaticamente pelas ferramentas ou pela edição manual do modelo. As alterações também podem resultar em acréscimos ao conjunto de *testbench*, permitindo que alterações subsequentes sejam testadas, avaliadas e verificadas. Esses testes que se aplicam a modelos de blocos individuais são semelhantes aos testes unitários, descritos na maioria da literatura sobre desenvolvimento de software (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Algumas organizações já possuem equipes para caracterizar os processos de fabricação e criar modelos correspondentes para uso nas tarefas de projeto e verificação. Essa atividade pode ser estendida para incluir a calibração de modelos comportamentais. Algumas organizações também exigem modelos calibrados de seus fornecedores, para facilitar o uso e verificação baseados em modelos. Além disso, as ferramentas estão disponíveis para ajudar na criação e calibração de tipos específicos de blocos de projeto. Onde possível, os resultados da simulação devem ser combinados com os resultados reais medidos para calibrar completamente os modelos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O objetivo da verificação do sistema é garantir que o projeto fabricado atenda às especificações e aos requisitos estabelecidos. Uma boa prática é executar a verificação o mais cedo possível no processo de projeto, uma vez que o custo de descobrir e corrigir problemas aumenta nas fases posteriores do processo. Ashenden, Peterson e Teegarden (2003) apresentam três formas de verificação:

- Verificação funcional - determina que o projeto funcionará como planejado, assumindo a perfeita interconexão dos componentes. Verificação funcional é mais precisa quando executada usando os modelos mais precisos disponíveis, geralmente os modelos de implementação física. No entanto, simular o uso desses modelos é demorado, por isso é mais eficiente usar modelos de nível superior já calibrados;
- Verificação de interconexão - leva em consideração a natureza não ideal das interconexões entre componentes e pode descobrir erros não capturados pela verificação funcional.

O posicionamento relativo dos blocos e sua interconexão pode afetar significativamente o desempenho geral do sistema. O desempenho da interconexão é especialmente importante para o projeto de circuitos integrados de alta velocidade e para os circuitos implementados em placas de circuitos impressos. Em geral, o projeto da interconexão de componentes deve considerar a colocação de componentes, efeitos de carregamento, interconexão e correspondente velocidade de operação e efeitos de dimensionamento de interconexão. Geralmente, é necessário modelar os mecanismos de interconexão e incluí-los no modelo do sistema (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Dado todo o contexto, adentremos agora na gramática do VHDL no capítulo posterior. Começaremos pela estrutura básica, passando pelos tipos de dados básicos, semelhantes em várias linguagens programação tradicional.

2. INTRODUÇÃO AO VHDL

2.1. Construção de modelos

O projeto do VHDL descreve um único sistema em um único arquivo. O arquivo tem a extensão *.VHD. Dentro do arquivo, existem duas partes que descrevem o sistema: a entidade e a arquitetura. A entidade descreve a interface para o sistema (ou seja, as entradas e saídas) e a arquitetura descreve o comportamento.

As funcionalidades do VHDL (por exemplo, operadores, tipos de sinal, funções etc.) são definidas através de pacotes, que são agrupados em uma biblioteca. O IEEE (*Institute of Electrical and Electronics Engineers*)[11] define o conjunto básico de funcionalidades para VHDL no pacote padrão. Este pacote está contido em uma biblioteca chamada IEEE. A inclusão da biblioteca e do pacote é declarada no início de um arquivo VHDL antes da entidade e da arquitetura.

Funcionalidades adicionais podem ser utilizadas através da inclusão de outros pacotes, mas todos os pacotes são baseados na funcionalidade principal definida no pacote padrão. Como resultado, não é necessário declarar explicitamente que um projeto está usando o pacote padrão IEEE porque é inerente ao uso do VHDL.

A palavra-chave library é usada para indicar quais pacotes serão adicionados ao projeto da biblioteca especificada. O nome da biblioteca segue a palavra-chave library. Para incluir um pacote específico da biblioteca, uma nova linha é usada com a palavra-chave use seguida pelos detalhes do pacote. A sintaxe do pacote possui três campos separados por um ponto. O primeiro campo é o nome da biblioteca. O segundo campo é o nome do pacote. O terceiro campo é a funcionalidade específica do pacote a ser incluído.

Se todas as funcionalidades de um pacote forem necessárias ao projeto, a palavra-chave all será usada no terceiro campo. Exemplos de como incluir alguns dos pacotes mais usados da biblioteca IEEE são mostrados abaixo.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_textio.all;
```

2.1.1. Entidade

A entidade descreve as entradas e saídas do sistema. Estes são chamados de portos. Cada porto precisa ter seu nome, modo e tipo especificados. O nome é definido pelo usuário. O modo descreve a direção em que os dados são transferidos pelo porto e pode assumir valores de `in`, `out`, `inout` e `buffer`. Os nomes de portos com o mesmo modo e tipo podem ser listados na mesma linha, separados por vírgulas. A definição de uma entidade é dada abaixo.

```
entity nome_da_entidade is
    port (nomedoporto : <modo> <tipo>;
          nomedoporto : <modo> <tipo>);
end entity;
```

2.1.2. Arquitetura

A arquitetura descreve o comportamento de um sistema. Existem inúmeras técnicas para descrever o comportamento em VHDL que abrangem vários níveis de abstração. A forma de uma arquitetura genérica é fornecida abaixo.

```
architecture nome_arquitetura of <entidade_associada> is
-- declaração de tipos enumerados (opcional)
-- declaração de sinais (opcional)
-- declaração de constantes (opcional)
-- declaração de componentes (opcional)
begin
-- descrição comportamental do sistema
end architecture;
```

Um sinal é usado para conexões internas em um sistema e é declarado na arquitetura. Cada sinal deve ser declarado com um tipo. O sinal só pode ser usado para fazer conexões de tipos semelhantes. Um sinal é declarado com a palavra-chave `signal` seguido de um nome definido pelo usuário, dois pontos e o tipo.

Sinais do mesmo tipo podem ser declarados na mesma linha, separados por vírgula. Todos os tipos de dados descritos anteriormente podem ser usados para sinais. Os sinais representam fios dentro do sistema, ou seja, eles não têm uma direção ou modo. Os sinais não podem ter o mesmo nome que um porto no sistema em que residem. A sintaxe para uma declaração de sinal é:

```
signal nome : < tipo >;
```

O VHDL suporta uma abordagem de projeto hierárquico. Os nomes de sinais podem ser os mesmos dentro de um subsistema que aqueles em um nível superior, sem conflito.

Uma constante é útil para representar uma quantidade que será usada várias vezes na arquitetura. A sintaxe para declarar uma constante é a seguinte:

```
constant nome_da_constante : <tipo> := <valor>;
```

Um componente é o termo usado para um subsistema que é instanciado em um sistema de nível superior. Se um componente for usado em um sistema, ele deverá ser declarado na arquitetura antes da instrução `begin`. A sintaxe para uma declaração de componente é:

```
component nome_do_componente
    port (nome_do_porto : <modo> <tipo>;
          nome_do_porto : <modo> <tipo>);
end component;
```

As definições de porto do componente devem corresponder exatamente às definições de porto da entidade do subsistema. Como tal, essas linhas geralmente são copiadas diretamente da descrição da entidade dos sistemas de nível inferior. Uma vez declarado, um componente pode ser instanciado após a instrução `begin` na arquitetura quantas vezes for necessário.

3. TODO DADO TEM UM TIPO

No VHDL, todo sinal, constante, variável e função deve receber um tipo de dado. O pacote padrão IEEE fornece uma variedade de tipos de dados predefinidos. Alguns tipos de dados são sintetizáveis, enquanto outros são apenas para modelar o comportamento. A seguir, são apresentados os tipos de dados mais usados no pacote padrão (LAMERES, 2019).

3.1. Tipos enumerados

Um tipo enumerado é aquele no qual são definidos os valores exatos que o tipo pode assumir, conforme apresentado na Tabela 1.

Tabela 1 - Descrição dos tipos de dados enumerados.

Tipo	Valores que o tipo pode assumir
`bit`	0, 1
`boolean`	false, true
`character`	Qualquer dos 256 caracteres ASCII definidos no ISO 8859-1

O tipo `bit` é sintetizável, enquanto `boolean` e `character` não. Os valores escalares individuais são indicados colocando-os entre aspas simples, por exemplo, `'0'`, `'a'`, `'true'`.

3.2. Tipos com intervalos

Um tipo com intervalo é aquele que pode assumir qualquer valor dentro de um intervalo, conforme apresentado na Tabela 2.

Tabela 2 - Tipos de dados com intervalos pré-definidos.

Tipo	Valores que o tipo pode assumir
`integer`	Número entre e
`real`	Números fracionais entre e

O tipo `integer` é um número em complemento de dois, com sinal e de 32 bits, e é sintetizável. Se o intervalo completo de valores inteiros não for desejado, esse tipo pode ser limitado incluindo o intervalo `<min>` a `<max>`. O tipo real é um valor de ponto flutuante de 32 bits e não pode ser sintetizado diretamente, a menos que um pacote adicional seja incluído e defina o formato do ponto flutuante.

3.3. Tipos físicos

Um tipo físico é aquele que contém um valor e unidades de medida. No VHDL, o `time` é o principal tipo físico suportado, conforma apresentado na Tabela 3. Já o VHDL-AMS suporta outros tipos físicos, como será visto mais adiante.

Tabela 3 - Unidades de representação do tipo time.

Tipo	Valores que o tipo pode assumir	
`time`	Número entre -2.147.483.648 e +2.147.483.647	
	fs	Femtosegundo, 10^{-15}
	ps = 1000 fs	Picosegundo, 10^{-12}
	ns = 1000 ps	Microsegundo, 10^{-9}
	µs = 1000 ns	Milisegundo, 10^{-6}
	ms = 1000 µs	Microsegundo, 10^{-3}
	s = 1000 ms	Segundos
	min = 60 s	Minutos
	h = 60 min	Horas

A unidade base para o `time` é fs, o que significa que, se nenhuma unidade for fornecida, o valor será assumido em femtossegundos. O valor do tempo é mantido como um número com sinal de 32 bits e não é sintetizável.

3.4. Tipo `vector`

Um tipo `vector` é aquele que consiste em uma matriz linear de tipos escalares, conforme apresentado na Tabela 4.

Tabela 4 - Valores possíveis para o tipo vector.

Tipo	Valores que o tipo pode assumir
`bit_vector`	Um vetor linear do tipo bit
`string`	Um vetor linear do tipo character

O tamanho de um tipo `vector` é definido pela inclusão do índice máximo, da palavra-chave `downto` o índice mínimo. Por exemplo, se um sinal chamado `BUSA` recebesse o tipo `bit_vector` (7 `downto` 0), ele criaria um vetor de 8 escalares, cada um do tipo `bit`. O escalar mais à esquerda teria um índice de 7 e o escalar mais à direita teria um índice de 0. Cada um dos escalares individuais no vetor pode ser acessado, fornecendo o número do índice entre parênteses. Por exemplo, `BUSA` (0) acessaria o escalar na posição mais à direita. Os índices nem sempre precisam ter um valor mínimo de 0, mas isso é a abordagem de indexação mais comum no projeto. O tipo `bit_vector` é sintetizável, enquanto `string` não é. Os valores desses tipos são indicados colocando-os entre aspas duplas, por exemplo, `"0011"`, `"abcd"`.

3.5. Tipo enumerado definido pelo usuário

Um tipo enumerado definido pelo usuário é aquele em que o nome do tipo é especificado pelo usuário, além de todos os valores possíveis que o tipo pode assumir. A criação de um tipo enumerado definido pelo usuário é mostrada abaixo.

```
type nome is (valor1, valor2, . . .);
```

Tipos enumerados definidos pelo usuário são sintetizáveis.

3.6. Tipo `array`

Um `array` contém vários elementos do mesmo tipo. Os elementos dentro de uma matriz podem ser escalares ou vetores. Para usar um `array`, deve ser declarado um novo tipo que define a configuração do `array`. Depois que o novo tipo é criado, os sinais podem ser declarados com o tipo. O intervalo do `array` deve ser definido na declaração. O intervalo é especificado com números inteiros (mínimo e máximo) e com as palavras-chave `downto` ou `to`. A criação de um tipo de `array` é mostrada abaixo.

```
type nome is array (<variacao>) of <tipo_do_elemento>;
```

3.7. Subtipos

Um subtipo é uma versão ou subconjunto restrito de outro tipo. Os subtipos são definidos pelo usuário, embora alguns subtipos comumente usados sejam predefinidos no pacote padrão. A seguir, é apresentada a sintaxe para declarar um subtipo e dois exemplos de subtipos comumente usados (`NATURAL` e `POSTIVE`) definidos no pacote padrão.

```
subtype nome is <type> range <min> to <max>;
```

```
subtype NATURAL is integer range 0 to 255;
subtype POSTIVE is integer range 1 to 256;
```

4. OPERADORES: O BÁSICO DE TUDO

Há uma variedade de operadores predefinidos no pacote padrão IEEE. É importante observar que os operadores são definidos para trabalhar em tipos de dados específicos e que nem todos os operadores são sintetizáveis. Todas as operações devem executar em tipos semelhantes e o resultado deve ser atribuído ao mesmo tipo que as entradas de operação.

O VHDL usa <= para todas as atribuições de sinal e := para todas as atribuições de variável e inicialização. Esses operadores de atribuição trabalham em todos os tipos de dados. O destino da atribuição fica à esquerda desses operadores e os argumentos de entrada à direita.

4.1. Operadores lógicos

VHDL contém os seguintes operadores lógicos, apresentados na Tabela 5.

Tabela 5 - Operadores lógicos.

Operador	Operação
`not`	Negação lógica
`and`	AND lógico
`nand`	NAND lógico
`or`	OR lógico
`nor`	NOR lógico
`xor`	OR exclusivo lógico
`xnor`	NOR exclusivo lógico

Esses operadores trabalham sobre os tipos `bit`, `bit_vector` e `boolean`. Para operações no tipo `bit_vector`, os vetores de entrada devem ter o mesmo tamanho e ocorrerão de maneira bit a bit. Por exemplo, se dois barramentos de 8 bits chamados `BUSA` e `BUSB` fossem operandos da operação `and`, o `BUSA(0)` realizaria a operação `and` individualmente com o `BUSB(0)`, o `BUSA(1)` realizaria a operação `and` individualmente com o `BUSB(1)` etc.. O operador `not` é uma operação unária (ou seja, opera em uma única entrada) e a palavra-chave é colocada à esquerda do sinal operando. Todos os outros operadores têm duas ou mais entradas e são colocados entre os nomes das entradas (LAMERES, 2019).

A ordem de precedência é diferente da álgebra booleana. O operador `not` tem uma prioridade mais alta que todos os outros operadores. Todos os outros operadores lógicos têm a mesma prioridade e não têm precedência inerente. Isso significa que, o operador `and` não precederá a operação `or`, como na álgebra booleana. Parênteses são usados para descrever explicitamente a precedência. Se forem utilizados operadores que tiverem a mesma prioridade e parênteses não forem fornecidos, as operações ocorrerão nos sinais listados mais à esquerda, movendo-se da esquerda para a direita na atribuição do sinal.

4.2. Operadores numéricos

VHDL contém os seguintes operadores numéricos apresentados na Tabela 6.

Tabela 6 - Operadores numéricos.

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
mod	Módulo
rem	Resto da divisão
abs	Valor absoluto
**	Exponenciação

Esses operadores trabalham nos tipos inteiro e real. Observe que o padrão VHDL não suporta operadores numéricos nos tipos `bit` e `bit_vector`.

4.3. Operadores relacionais

Operadores relacionais comparam duas entradas do mesmo tipo e retornam o tipo `boolean` (ou seja, verdadeiro ou falso), e são apresentados na Tabela 7.

Tabela 7 - Operadores relacionais.

Operador	Retorna verdadeira se a comparação é:
=	Igual
/=	Diferente
<	Menor que
<=	Menor ou igual que
>	Maior que
>=	Maior ou igual que

4.4. Operador de deslocamento

Operadores de deslocamento trabalham nos tipos `bit_vector` e `string`, e são apresentados na Tabela 8.

Tabela 8 - Operadores de deslocamento de bits.

Operador	Operação
`sll`	Deslocamento lógico a esquerda
`srl`	Deslocamento lógico a direita
`sla`	Deslocamento aritmético a esquerda
`sra`	Deslocamento aritmético a direita
`rol`	Rotação a esquerda
`ror`	Rotação a direita

A sintaxe para usar uma operação de deslocamento é fornecer o nome do vetor seguido pelo operador de deslocamento desejado, seguido por um número inteiro indicando quantas operações de deslocamento devem ser executadas. O destino da atribuição deve ser do mesmo tipo e tamanho de entrada.

4.5. Operador de concatenação

Em VHDL, o `&` é usado para concatenar vários sinais. O destino desta operação deve ter o mesmo tamanho da soma dos tamanhos dos argumentos de entrada.

5. TODA LINHA É CONCORRENTE

É importante lembrar que o VHDL é uma linguagem de descrição de hardware, não uma linguagem de programação. Em uma linguagem de programação convencional, as linhas de código são executadas sequencialmente conforme aparecem no arquivo de origem. No VHDL, as linhas de código representam o comportamento do hardware real. Como resultado, todas as atribuições de sinal são executadas por padrão simultaneamente, a menos que seja especificado de outra forma (LAMERES, 2019).

5.1. Declaração de sinais concorrentes

As atribuições de sinal concorrente são realizadas simplesmente usando o operador <= após a instrução `begin` na arquitetura. Cada atribuição individual será executada simultaneamente e sintetizada como circuitos lógicos separados. Considere o seguinte exemplo:

```
X <= A;
Y <= B;
Z <= C;
```

Quando simuladas, essas três linhas farão três atribuições de sinal separadas ao mesmo tempo. Isso é diferente de uma linguagem de programação tradicional que atribui primeiro `A` a `X`, depois `B` a `Y` e, finalmente, `C` a `Z`. No VHDL, essa funcionalidade é idêntica a três fios separados e será sintetizada diretamente em dessa forma.

Abaixo está outro exemplo de como as atribuições de sinal simultâneo em VHDL diferem de uma linguagem de programação executada sequencialmente:

```
A <= B;
B <= C;
```

Em uma simulação, as atribuições de sinal de `C` a `B` e `B` a `A` ocorrerão ao mesmo tempo; no entanto, durante a síntese, o sinal `B` será eliminado do projeto, pois essa funcionalidade descreve dois fios em série (LAMERES, 2019).

5.2. Declarações condicionais

Operadores lógicos descrevem o comportamento de pequenos circuitos. No entanto, ainda será necessário criar manualmente a expressão lógica que descreve a funcionalidade do sistema. O interessante é quando o comportamento do sistema pode ser descrito completamente sem a necessidade de qualquer projeto manual. Uma atribuição de sinal condicional nos permite descrever uma atribuição de sinal simultâneo usando condições booleanas que afetam os valores do resultado. Em uma atribuição de sinal condicional, a palavra-chave `when` é usada

para descrever a atribuição de sinal para uma condição booleana específica. A palavra-chave `else` é usada para descrever as atribuições de sinal para outras condições. Várias condições booleanas podem ser usadas para descrever completamente a saída do circuito em todas as condições de entrada. Operadores lógicos também podem ser usados nas condições booleanas para criar condições mais sofisticadas. As condições booleanas podem ser incluídas entre parênteses para facilitar a leitura (LAMERES, 2019). A sintaxe para uma atribuição de sinal condicional é mostrada abaixo.

```
nome_do_sinal <= expressao_1 when condicao_1 else
      expressao_2 when condicao_2 else
      :
      expressao_n;
```

5.3. Atribuição de sinais com `select`

Uma atribuição de sinal com `select` fornece outra técnica para implementar atribuições de sinal simultâneas. Nesta abordagem, a atribuição do sinal é baseada em um valor específico no sinal de entrada. A palavra-chave `with` é usada para iniciar a atribuição do sinal. Em seguida, é inserido o nome da entrada que será usada para determinar o valor da saída. Somente um único nome de variável pode ser listado como entrada. Isso significa que, se a atribuição for baseada em várias variáveis, elas deverão ser concatenadas em um único nome de vetor antes de iniciar a atribuição do sinal. Após a entrada da lista, a palavra-chave `select` significa o início das atribuições do sinal. É feita uma atribuição a um sinal com base em uma lista de possíveis valores de entrada que seguem a palavra-chave `when`. Vários valores de entrada podem ser usados e são separados por vírgulas. A palavra-chave `others` é usada para cobrir quaisquer valores de entrada que não sejam explicitamente declarados (LAMERES, 2019). A sintaxe para uma atribuição de sinal selecionada é a seguinte:

```
with nome_da_entrada select
      nome_do_sinal <= expressao_1 when condicao_1,
                       expressao_2 when condicao_2,
                       :
                       expressao_n when others;
```

Uma característica da atribuição de sinal com `select` que torna sua forma ainda mais compacta do que outras técnicas é que vários sinais de entrada que correspondem à mesma atribuição de saída podem ser listados por meio do delimitador de linha (|). Por exemplo:

```
with AB select
      F3 <= '1' when "01" | "10",
            '0' when others;
```

`F3` receberá o valor `'1'`, quando `A=0` e `B=1` ou quando `A=1` e `B=0`. Para quaisquer valores diferentes destes F3 receberá o valor `'0'`.

5.4. Atribuição de sinais com atraso

5.4.1. Atraso inercial

O VHDL oferece a capacidade de atrasar uma atribuição de sinal simultâneo para modelar com mais precisão o comportamento de portas lógicas ou fios reais, os quais possuem atrasos de propagação de sinal, devido a características físicas dos matérias semicondutores. A palavra-chave `after` é usada para atrasar uma atribuição por um determinado período de tempo. A magnitude do atraso é fornecida como tipo `time` (LAMERES, 2019). A sintaxe para atrasar uma atribuição é a seguinte:

```
nome_do_sinal <= <expressao> after <time>;
```

5.4.2. Atraso de transporte

Ignorar pequenas pulsos de entrada modela com precisão o comportamento das portas no circuito. Quando o pulso de entrada é mais rápido que o atraso da porta, a saída da porta não tem tempo para responder. Como resultado, não haverá uma alteração lógica na saída. Se desejar que todos os pulsos nas entradas apareçam nas saídas ao modelar o comportamento de outros tipos de lógica digital, a palavra-chave `transport` é usada em conjunto com a instrução `after`. Isso é chamado de modelo de atraso de transporte (LAMERES, 2019). A sintaxe é:

```
nome_do_sinal <= transport <expressao> after <time>;
```

6. PROJETO HIERÁRQUICO

6.1. Componentes

Um subsistema é chamado de componente em VHDL. Para qualquer componente que será usado em uma arquitetura, ele deve ser declarado antes da instrução `begin`. Um componente específico precisa ser declarado apenas uma vez. Após a instrução `begin`, ele pode ser usado quantas vezes forem necessárias. Cada componente é executado simultaneamente. O termo instanciação refere-se ao uso, ou inclusão, do componente no sistema VHDL. Quando um componente é instanciado, ele precisa receber um nome de identificação exclusivo. Isso é chamado de nome da instância. Para instanciar um componente, o nome da instância é fornecido primeiro, seguido de dois pontos e depois o nome do componente. A última parte da instanciação de um componente é conectar sinais às suas portas. A maneira pela qual os sinais são conectados aos portos do componente é chamada de mapeamento de portos (LAMERES, 2019). A sintaxe para instanciar um componente é a seguinte:

```
nome_da_instancia : <component nome_do_componente >
       port map (<port conexoes>);
```

Existem duas formas para conectar sinais às portas do componente, mapeamento de porto explícito e mapeamento de porto posicional.

- No mapeamento de porto explícito, o nome de cada porto do componente é fornecido, seguido pelo indicador de conexão =>, seguido pelo sinal ao qual está conectado. As conexões de porto podem ser listadas em qualquer ordem, pois os detalhes da conexão (ou seja, nome do porto para o nome do sinal) são explícitos. Cada nome de conexão é separado por vírgula. A sintaxe para o mapeamento de porto explícito é a seguinte:

```
nome_da_instancia : <component nome_do_componente>
      port map (porto1 => sinal1, porto2 => sinal2, ...);
```

- No mapeamento de porto posicional, os nomes dos portos do componente não são explicitamente listados. Em vez disso, os sinais são listados na mesma ordem em que os portos do componente foram definidos. Cada nome de sinal é separado por vírgula. Essa abordagem requer menos texto para descrever, mas também pode levar a conexões incorretas devido a incompatibilidades na ordem dos sinais conectados. A sintaxe para o mapeamento de porto posicional é a seguinte:

```
nome_da_instancia : <component nome_do_componente>
       port map (sinal1, sinal2, ...);
```

7. ABSTRAINDO: MODELAGEM SEQUENCIAL

As técnicas de modelagem apresentadas foram apropriadas para a lógica combinacional porque esses tipos de circuitos têm saídas dependentes apenas dos valores atuais de suas entradas. Isso significa que um modelo que executa atribuições de sinal continuamente fornece um modelo preciso desse comportamento do circuito. Quando começamos a observar circuitos sequenciais (isto é, flip-flops, registradores, máquinas de estados finitos e contadores), esses dispositivos apenas atualizam suas saídas com base em um evento, geralmente na borda (ascendente ou descendente) de um sinal de relógio. Descrevemos as construções VHDL para modelar atribuições de sinal que são acionadas por um evento para modelar com precisão a lógica sequencial. Podemos então usar essas técnicas para descrever circuitos lógicos sequenciais mais complexos, como máquinas de estados finitos e sistemas de nível RTL (D'AMORE, 2005; LAMERES, 2019).

7.1. Básico de processos

O VHDL usa um processo para modelar atribuições de sinais baseadas em um evento. Um processo é uma técnica para modelar o comportamento de um sistema. Assim, um processo é colocado na arquitetura VHDL após a instrução `begin`. As atribuições de sinal em um processo têm características únicas que permitem modelar com precisão a lógica sequencial. Primeiro, as atribuições de sinal não ocorrem até que o processo termine ou seja suspenso. Segundo, as atribuições de sinal serão feitas apenas uma vez a cada vez que o processo for acionado. Por fim, as atribuições de sinal serão executadas na ordem em que aparecerem no processo. Esse comportamento de atribuição é chamado de atribuição de sinal sequencial. As atribuições sequenciais de sinal permitem que um processo modele o comportamento do nível RTL, em que um sinal pode ser usado como operando de uma atribuição e como destino de uma atribuição diferente dentro do mesmo processo. O VHDL fornece duas técnicas para acionar um processo, a lista de sensibilidade e a instrução de espera (D'AMORE, 2005; LAMERES, 2019).

7.1.1. Lista de sensibilidade

Uma lista de sensibilidade é um mecanismo para controlar quando um processo é acionado (ou iniciado). Uma lista de sensibilidade contém uma lista de sinais aos quais o processo é sensível. Se houver uma transição em qualquer um dos sinais da lista, o processo será acionado e as atribuições de sinal no processo serão feitas (LAMERES, 2019). A seguir, é apresentada a sintaxe para um processo que usa uma lista de sensibilidade.

```
nome_do_processo : process (<nome_do_sinal1>, <nome_do_sinal2>, ...)
-- declaração de variaveis
begin
       declaracao_sinal_sequencial_1
       declaracao_sinal_sequencial_2
       :
end process;
```

7.1.2. Declaração `wait`

Uma declaração `wait` é um mecanismo para suspender (ou interromper) um processo e permitir que as atribuições de sinal sejam executadas sem a necessidade do processo terminar. Ao usar uma instrução `wait`, uma lista de sensibilidade não é usada. Sem uma lista de sensibilidade, o processo será acionado imediatamente. Dentro do processo, a instrução `wait` é usada para parar e iniciar o processo. Existem três maneiras pelas quais as instruções de espera podem ser usadas. A primeira é uma espera indefinida. No exemplo a seguir, o processo não contém uma lista de sensibilidade, portanto, será acionado imediatamente. A palavra-chave `wait` é usada para suspender o processo. Quando essa declaração é alcançada, as atribuições de sinal para `Y1` e `Y2` serão executadas e o processo será suspenso indefinidamente (LAMERES, 2019).

```
Proc_Ex1 : process
begin
     Y1 <= '0';
     Y2 <= '1';
     wait;
end process;
```

A segunda maneira para usar uma instrução de espera para suspender um processo é usá-la em conjunto com a palavra-chave `for` e uma expressão de tempo. No exemplo a seguir, o processo será acionado imediatamente, pois não contém uma lista de sensibilidade. Quando o processo atingir a instrução `wait`, ele suspenderá e executará a primeira atribuição de sinal ao `CLK` (`CLK <= '0'`). Após 5 ns, o processo será iniciado novamente. Quando atingir a segunda instrução `wait`, suspenderá e executará a segunda atribuição de sinal para `CLK` (`CLK <= '1'`). Após outros 5 ns, o processo será iniciado novamente e será encerrado imediatamente devido à instrução final do processo. Após o término do processo, ele será acionado imediatamente novamente devido à falta de uma lista de sensibilidade e repetirá o comportamento descrito acima. Esse comportamento continuará indefinidamente. Este exemplo cria uma onda quadrada chamada `CLK` com um período de 10 ns (LAMERES, 2019).

```
Proc_Ex2 : process
begin
     CLK <= '0'; wait for 5 ns;
     CLK <= '1'; wait for 5 ns;
end process;
```

A terceira maneira para usar uma instrução `wait` para suspender um processo é usá-la em conjunto com a palavra-chave `until` e uma condição booleana. No exemplo a seguir, o processo será acionado imediatamente porque não há uma lista de sensibilidade. O processo será suspenso imediatamente e somente será retomado quando uma condição booleana se tornar verdadeira (ou seja, `Counter > 15`). Quando atingir a segunda instrução `wait`, executará a primeira atribuição de sinal ao `RollOver` (`RollOver <= '1'`). Após 1 ns, o processo será retomado. Quando o processo terminar, ele executará a segunda atribuição de sinal ao `RollOver` (`RollOver <= '0'`).

```
Proc_Ex3 : process
begin
     wait until (Counter > 15);       -- primeira declaração wait
     RollOver <= '1'; wait for 1 ns; -- segunda declaração wait
     RollOver <= '0';
end process;
```

7.1.3. Declaração de sinais sequenciais

Um dos conceitos mais confusos de um processo é como as atribuições sequenciais de sinais se comportam. As regras de atribuição de sinais em um processo são as seguintes:

- Os sinais não podem ser declarados dentro de um processo;
- As atribuições de sinal não ocorrem até o processo terminar ou suspender;

As atribuições de sinal são executadas na sequência em que aparecem no processo (quando o processo termina ou o processo é suspenso) (D'AMORE, 2005; LAMERES, 2019).

7.1.4. Variáveis

Há situações dentro dos processos em que é desejável que as atribuições sejam feitas instantaneamente, e não quando o processo é suspenso. Para essas situações, o VHDL fornece o conceito de uma variável. Uma variável possui as seguintes características:

- Variáveis existem apenas dentro de um processo;
- Variáveis são definidas em um processo antes da instrução `begin`;
- Quando o processo termina, as variáveis são removidas do sistema. Isso significa que atribuições a variáveis não podem ser feitas por sistemas fora do processo;
- As atribuições às variáveis são feitas usando o operador "`:=`";
- As atribuições às variáveis são feitas instantaneamente.

A sintaxe para declarar uma variável é a seguinte:

```
variable variable_name : <type> := <initial_value>;
```

7.2. Programação condicional

Um dos recursos mais poderosos que os processos fornecem no VHDL é a capacidade de usar construções de programação condicional, como cláusulas `if / then`, instruções `case` e laços de repetição. Essas construções estão disponíveis apenas em um processo, mas seu uso não se limita à modelagem da lógica sequencial. Isso fornece a capacidade de modelar lógica combinacional e sequencial usando as construções de linguagem de programação tradicional (LAMERES, 2019).

7.2.1. Declaração `if / then`

Uma instrução `if / then` fornece uma maneira de fazer atribuições de sinais condicionais com base nas condições booleanas. A parte `if` da instrução é seguida por uma condição booleana que, se avaliada como `TRUE`, fará a atribuição do sinal após a instrução `then`. Se a condição booleana for avaliada como `FALSE`, nenhuma atribuição será feita. O VHDL fornece várias variantes da instrução `if / then`. Uma instrução `if / then / else` fornece uma atribuição final de sinal que será feita se a condição booleana for avaliada como `FALSE`. Uma instrução `if / then / elsif` permite que várias condições booleanas sejam usadas (LAMERES, 2019). A sintaxe para as várias formas da instrução `if / then` é a seguinte:

```
if condicao_booleana then declaracao_sequencial
end if;


if condicao_booleana then declaracao_sequencial_1
else declaracao_sequencial_2
end if;


if condicao_booleana_1 then declaracao_sequencial_1
elsif condicao_booleana_2 then declaracao_sequencial_2
:
:
elsif condicao_booleana_n then declaracao_sequencial_n
end if;


if condicao_booleana_1 then declaracao_sequencial_1
elsif condicao_booleana_2 then declaracao_sequencial_2
:
:
elsif condicao_booleana_n then declaracao_sequencial_n
else sequential_statement_n+1
end if;
```

7.2.2. Declaração case

Uma declaração `case` é outra maneira para modelar atribuições de sinais com base em condições booleanas. Assim como na instrução `if / then`, uma instrução `case` pode ser usada apenas dentro de um processo. A instrução começa com a palavra-chave `case` seguida pelo nome do sinal de entrada do qual as atribuições serão baseadas. O nome do sinal de entrada pode estar opcionalmente entre parênteses para facilitar a leitura. A palavra-chave `when` é usada para especificar um valor (ou escolha) do sinal de entrada que resultará em atribuições de sinal sequenciais associadas. As atribuições são listadas após o símbolo => (LAMERES, 2019). A seguir está a sintaxe para uma instrução `case`.

```
case (nome_da_entrada) is
     when opcao_1 => declaracao_sequencial(s);
     when opcao_2 => declaracao_sequencial(s);
     :
     :
     when opcao_n => declaracao_sequencial(s);
end case;
```

Quando nem todas as condições de entrada possíveis (ou opções) são especificadas, uma cláusula `when others` é usada para fornecer atribuições de sinal para todas as outras condições de entrada. A seguir, é apresentada a sintaxe de uma instrução `case` que usa a cláusula `when others`.

```
case (nome_da_entrada) is
     when opcao_1 => declaracao_sequencial(s);
     when opcao_2 => declaracao_sequencial(s);
     :
     :
     when others => declaracao_sequencial(s);
end case;
```

Várias opções que correspondem às mesmas atribuições de sinal podem ser delimitadas por (|) na instrução `case`. A seguir, é apresentada a sintaxe para uma instrução `case` com opções delimitadas.

```
case (nome_da_entrada) is
     when opcao_1 | opcao_2 => declaracao_sequencial(s);
     when others => declaracao_sequencial(s);
end case;
```

7.2.3. Laços infinitos

Um laço no VHDL fornece um mecanismo para executar atribuições repetitivas infinitamente. Isso é útil em *testbenches* para criar estímulos, como relógios ou outras formas de onda

periódicas. Um laço pode ser usado apenas dentro de um processo. A palavras-chave `loop` é usada para representar o início do laço. As atribuições sequenciais de sinal são então inseridas. O final do laço é representado pelas palavras-chave `end loop`. Dentro do laço, as instruções `wait`, `wait until` e `after` são permitidas (LAMERES, 2019).

As atribuições de sinal dentro de um laço serão executadas repetidamente para sempre, a menos que uma instrução `exit` ou `next` seja encontrada. A cláusula de saída fornece uma condição booleana que forçará o laço a terminar se a condição for avaliada como verdadeira.

Ao usar a instrução `exit`, uma atribuição de sinal adicional geralmente é colocada após o laço para fornecer o comportamento desejado quando o laço não está ativo. O uso de instruções de controle de fluxo, como `wait for` e `wait after`, fornece um meio de evitar que o laço seja executado novamente imediatamente após a saída. A cláusula `next` fornece uma maneira de pular as atribuições de sinal restantes e iniciar a próxima iteração do laço. A seguir, é apresentada a sintaxe para um laço infinito (LAMERES, 2019).

```
loop
     exit when condicao_booleana; -- declaracao exit opcional
     next when condicao_booleana; -- declaracao next opcional
     declaracao_sequencial(s);
end loop;
```

7.2.4. Laço `while`

Um laço `while` fornece uma estrutura com uma condição booleana que controla sua execução. O laço será executado apenas enquanto sua condição for avaliada como verdadeira. A seguir, é apresentada a sintaxe para um laço `while`.

```
while condicao_booleana loop
     declaracao_sequencial(s);
end loop;
```

7.2.5. Laço `for`

Um laço `for` fornece a capacidade de criar um laço que executará um número predefinido de vezes. O intervalo do laço é especificado com números inteiros (`min`, `max`) no início do laço `for`. Uma variável de laço é declarada implicitamente no laço que aumentará (ou diminuirá) de `min` para `max` do intervalo. A variável de laço é do tipo `integer`. Se desejar que a variável do laço seja incrementada de `min` a `max`, a palavra-chave `to` será usada ao especificar o intervalo do laço. Se desejar que a variável do laço diminua de `max` para `min`, a palavra-chave `downto` é usada ao especificar o intervalo do laço.

A variável laço pode ser usada dentro do laço como um índice para vetores; assim, o loop `for` é útil para acessar e atribuir automaticamente vários sinais dentro de uma única estrutura de laço. A seguir, é apresentada a sintaxe para um laço `for`, no qual a variável do laço será incrementada de `min` a `max` do intervalo:

```
for variavel in min to max loop
     declaracao_sequencial(s);
end loop;
```

A seguir, é apresentada a sintaxe de um laço `for`, no qual a variável de laço diminuirá de `max` para `min` do intervalo:

```
for variavel in max downto min loop
     declaracao_sequencial(s);
end loop;
```

Os laços são sintetizáveis, desde que o comportamento completo do sistema desejado seja descrito pelo laço.

7.3. Atributos de sinais

Há situações em que queremos descrever um comportamento baseado em mais do que apenas o valor atual de um sinal. Por exemplo, um flip-flop real apenas atualiza suas saídas em um tipo específico de transição (ou seja, ascendente ou descendente). Para modelar esse comportamento, precisamos especificar mais informações sobre o sinal. Isso é realizado usando atributos (LAMERES, 2019).

Um atributo pode fornecer informações como valores passados, se uma atribuição foi feita a um sinal ou quando a última vez que uma atribuição resultou em uma alteração de valor. Um atributo de sinal é implementado colocando um apóstrofo (') após o nome do sinal e listando a palavra-chave do atributo.

Atributos diferentes resultarão em diferentes tipos de saída. Os atributos que produzem tipos de saída booleanos podem ser usados como entradas para condições de decisão booleanas para outras construções. Outros atributos podem ser usados para definir o intervalo de novos vetores, referenciando o tamanho dos vetores existentes ou definindo automaticamente o número de iterações em um laço (LAMERES, 2019).

Finalmente, alguns atributos podem ser usados para criar *testbenches* de auto verificação que monitoram o impacto dos atrasos do circuito na funcionalidade de um sistema. A seguir, é apresentada uma lista dos atributos de sinal predefinidos comumente usados. O nome do sinal de exemplo `A` é usado para ilustrar como os atributos escalares operam. O exemplo de sinal `B` é usado para ilustrar como os atributos vetoriais operam com o tipo `bit_vector (7 a 0)` (LAMERES, 2019). Todos são apresentados na Tabela 9.

Tabela 9 - Principais atributos de sinais.

Atributo	Informação retornada	Tipo retornado
`A'event`	Verdadeiro quando o sinal `A` muda, falso caso contrário	`boolean`
`A'active`	Verdadeiro quando uma atribuição é feita `A`, falso caso contrário	`boolean`
`A'last_event`	O tempo em que o sinal A mudou pela última vez	`time`
`A'last_active`	O tempo em que uma atribuição a `A` foi feita pela última vez	`time`
`A'last_value`	O valor anterior de `A`	O mesmo de `A`
`B'length`	O tamanho do vetor (por exemplo, 8)	`integer`
`B'left`	Elemento mais a esquerda do vetor (por exemplo, 7)	`integer`
`B'right`	Elemento mais à direita do vetor (por exemplo, 0)	`integer`
`B'range`	Intervalo do vetor (por exemplo, "`7 downto 0`"	`string`

8. APROVEITANDO AS FACILIDADES: PACOTES

Uma das desvantagens do pacote padrão é que ele fornece funcionalidade limitada em seus tipos de dados sintetizáveis. O `bit` e o `bit_vector`, embora sintetizáveis, carecem da capacidade de modelar com precisão muitas das topologias implementadas nos sistemas digitais modernos. Principalmente as topologias que envolvem vários drivers conectados a um único fio. O pacote padrão não permitirá esse tipo de conexão. Contudo, esse tipo de topologia é uma maneira comum de fazer interface com vários nós em uma interconexão compartilhada (LAMERES, 2019).

Além disso, o pacote padrão não fornece muitos recursos úteis para esses tipos, como *don't care state*, aritmética usando os operadores + e −, funções de conversão de tipo ou a capacidade de ler / gravar arquivos externos. Para aumentar a funcionalidade do VHDL, os pacotes são incluídos no projeto (D'AMORE, 2005; LAMERES, 2019).

8.1. O pacote `STD_LOGIC_1164`

No final dos anos 80, foi lançado o pacote com o padrão IEEE 1164 (IEEE,1993) que adicionava funcionalidade ao VHDL para permitir um sistema lógico de valores múltiplos (ou seja, um sinal pode assumir mais valores do que apenas 0 e 1). Esse padrão também forneceu um mecanismo para conectar vários drivers ao mesmo sinal. Uma versão atualizada em 1993, chamada IEEE 1164-1993, foi a atualização mais significativa para esse padrão e contém a maioria das funcionalidades usadas no VHDL atualmente. Quase todos os sistemas descritos em VHDL incluem o padrão IEEE 1164 como um pacote. Este pacote é incluído adicionando a seguinte sintaxe no início do arquivo (LAMERES, 2019).

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Este pacote define quatro novos tipos de dados: `std_ulogic`, `std_ulogic_vector`, `std_logic` e `std_logic_vector`. `Std_ulogic` e `std_logic` são enumerados, ou seja, tipos escalares que podem fornecer um sistema lógico de valores múltiplos. Os tipos `std_ulogic_vector` e `std_logic_vector` são tipos vetoriais que contêm uma matriz linear de tipos escalares `std_ulogic` e `std_logic`, respectivamente. Os tipos escalares podem assumir nove valores diferentes, conforme descrito na Tabela 10.

Tabela 10 - Tipos escalares disponível no pacote `std_logic_1164`.

Valor	Descrição
U	Não inicializado (valor inicial padrão)
X	Forçando desconhecido
0	Forçando 0
1	Forçando 1
Z	Alta impedância
W	Desconhecido fraco
L	0 fraco (pull-down)
H	1 fraco (pull-up)
-	Don't care (usado somente para síntese)

Esses valores podem ser atribuídos aos sinais colocando-os entre aspas simples (escalares) ou aspas duplas (vetores).

8.1.1. Resolução de funções

O `std_logic_1164` resolverá o conflito de sinal do tipo `std_logic` usando uma função de resolução. Sempre que houver um conflito, o simulador consultará a função de resolução para determinar o valor do sinal. A Figura 6 mostra as forças relativas dos nove possíveis valores de sinal fornecidos pelo pacote `std_logic_1164` e pela função de resolução (LAMERES, 2019).

Figura 7 - função de resolução de conflito do pacote `std_logic_1164`.

Valor 1 \ Valor 2	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

8.1.2. Operadores lógicos

O `std_logic_1164` também contém novas definições para todos os operadores lógicos (`and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`) para os tipos `std_ulogic` e `std_logic`. Isso é necessário, pois esses tipos de dados podem assumir mais valores lógicos do que apenas 0 ou 1, logo, as definições do operador lógico do pacote padrão não são suficientes (LAMERES, 2019).

8.1.3. Funções de detecção de borda

O `std_logic_1164` também fornece funções para a detecção de transições de subida ou descida em um sinal. As funções `rising_edge()` e `falling_edge()` fornecem uma forma mais legível dessa funcionalidade em comparação com a abordagem (`Clock'event and Clock ='1'`).

8.1.4. Funções de conversão de tipos

O pacote `std_logic_1164` também fornece funções para converter entre tipos de dados. Existem funções para converter entre `bit`, `std_ulogic` e `std_logic`. Também existem funções para converter entre as formas vetoriais desses tipos (`bit_vector`, `std_ulogic_vector` e `std_logic_vector`). As funções estão listadas na Tabela 11.

Tabela 11 - Funções de conversão de tipos do pacote `std_logic_1164`.

Nome	Tipo de entrada	Tipo retornado
`To_bit()`	`std_ulogic`	`bit`
`To_bitvector()`	`std_ulogic_vector`	`bit_vector`
`To_bitvector()`	`std_logic_vector`	`bit_vector`
`To_StdULogic()`	`bit`	`std_ulogic`
`To_StdULogicVector()`	`bit_vector`	`std_ulogic_vector`
`To_StdULogicVector()`	`std_logic_vector`	`std_ulogic_vector`
`To_StdLogicVector()`	`bit_vector`	`std_logic_vector`
`To_StdLogicVector()`	`std_ulogic_vector`	`std_logic_vector`

Ao usar essas funções, o nome da função e o sinal de entrada são colocados à direita do operador de atribuição e o sinal de destino é colocado à esquerda.

8.2. O pacote NUMERIC_STD

O pacote numeric_std fornece computação numérica para os tipos std_logic e std_logic_vector. Ao executar aritmética binária, os resultados das operações e comparações aritméticas variam muito, dependendo se o número binário é sem sinal ou com sinal. Como resultado, o pacote numeric_std define dois novos tipos de dados, unsigned e signed.

Um tipo unsigned é definido para ter seu MSB (*Most Significant Bit*) na posição mais à esquerda do vetor e o LSB (*Least Significant Bit*) na posição mais à direita do vetor. Um número signed usa a representação de complemento de dois, com o bit mais à esquerda do vetor sendo o bit de sinal. Ao declarar um sinal como um desses tipos, está implícito que eles representam a codificação de um tipo nativo subjacente de std_logic / std_logic_vector.

O uso de tipos unsigned / signed fornece a interpretação de como os operadores aritméticos, lógicos e de comparação serão executados. Isso também implica que o pacote numeric_std exige que o std_logic_1164 seja sempre incluído. Embora o pacote numeric_std inclua uma chamada de inclusão do pacote std_logic_1164, é comum incluir explicitamente os pacotes std_logic_1164 e numeric_std no arquivo principal. O sintetizador ignorará instruções de pacote redundantes (LAMERES, 2019). A sintaxe para incluir esses pacotes é a seguinte:

```
library IEEE;
use IEEE.std_logic_1164.all; -- defines types std_ulogic and std_logic
use IEEE.numeric_std.all; -- defines types unsigned and signed
```

8.2.1. Funções aritméticas

O pacote numeric_std fornece suporte para várias funções aritméticas para os tipos unsigned e signed. Isso inclui as funções +, *, /, mod, rem e abs. Essas operações aritméticas se comportam de maneira diferente para os tipos unsigned e signed, mas o sintetizador usará automaticamente a operação correta com base nos tipos dos argumentos de entrada (LAMERES, 2019).

A maioria das ferramentas de síntese suporta os operadores de adição, subtração e multiplicação neste pacote. Isso fornece um nível mais alto de abstração ao modelar circuitos aritméticos. Lembre-se de que o pacote padrão VHDL não suporta adição, subtração e multiplicação dos tipos bit / bit_vector usando os operadores +, – e *. O uso do pacote numeric_std permite modelar essas operações aritméticas com um tipo de dados sintetizável usando os operadores matemáticos mais familiares. As funções de divisão, módulo, restante e valor absoluto não são sintetizáveis diretamente deste pacote (LAMERES, 2019).

O pacote numeric_std permite a modelagem aritmética em um nível mais alto de abstração. Vejamos uma situação de implementação de um circuito somador usando o operador "+". Embora esse operador seja suportado para o tipo integer no pacote std_logic_1164, modelar a adição usando números inteiros podem ser onerosos devido aos vários níveis de conversão, verificação de intervalo e manipulação manual da execução. Uma abordagem mais simples para modelar o comportamento do somador é usar os tipos não unsigned / signed

e o operador "+" fornecido no pacote `numeric_std`. Sinais ou variáveis temporárias desses tipos são necessários para modelar o comportamento do somador com o sinal "+". Além disso, a conversão de tipo ainda é necessária ao atribuir os valores de volta aos portos de saída. Uma vantagem dessa abordagem é que a verificação de faixa de valores é eliminada porque a sobreposição é tratada automaticamente com esses tipos (LAMERES, 2019).

8.2.2. Funções lógicas

O pacote `numeric_std` fornece suporte para todos os operadores lógicos (`and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`) para tipos `unsigned` e `signed`. Ele também fornece duas novas funções de deslocamento: `shift_left()` e `shift_right()`. Essas funções de deslocamento preencherão a posição vaga no vetor após o deslocamento com um 0. Assim, esses são deslocamentos lógicos (LAMERES, 2019). Este pacote também fornece duas novas funções de rotação: `rotate_left()` e `rotate_right()`.

8.2.3. Operador de comparação

O pacote `numeric_std` fornece suporte para todas os operadores de comparação para tipos `unsigned` e `signed`. Isso inclui >, <, <=, >=, = e /=. Essas comparações retornam o tipo `boolean`.

8.2.4. Funções de detecção de borda

O `numeric_std` também fornece as funções `rising_edge()` e `falling_edge()` para a detecção de transição de borda ascendente ou descendente para tipos `unsigned` e `signed`.

8.2.5. Funções de conversão

O pacote `numeric_std` contém uma variedade de funções de conversão úteis. De particular utilidade são as funções entre o tipo `integer` e "de" ou "para" `unsigned` / `signed`. Isso permite que modelos comportamentais para contadores, somadores e subtratores sejam implementados usando o número inteiro do tipo mais legível (LAMERES, 2019).

Após a descrição da funcionalidade, uma conversão pode ser usada para transformar o resultado em tipos `unsigned` ou `signed` para fornecer uma saída sintetizável. Ao converter um número inteiro em um vetor, um argumento de tamanho é incluído. O argumento `size` é do tipo `integer` e fornece o número de bits no vetor em que o número inteiro será convertido. As funções existentes são descritas na Tabela 12.

Tabela 12 - Funções de conversão de tipo do pacote numeric_std.

Nome	Tipo de entrada	Tipo retornado
To_integer()	unsigned	integer
To_integer()	signed	integer
To_unsigned()	integer, <size>	unsigned (size-1 downto 0)
To_signed()	integer, <size>	signed (size-1 downto 0)

8.2.6. *Casting* de tipos

O VHDL contém um conjunto de operações de conversão de tipo internas comumente usadas com o pacote numeric_std para converter entre std_logic_vector e unsigned / signed. Como os tipos unsigned e signed são baseados no tipo subjacente std_logic_vector, a conversão é simplesmente conhecida como *casting*[12]. Na Tabela 13 são apresentados os recursos de *casting* de tipo.

Tabela 13 - Funções de casting de tipos do pacote numeric_std.

Nome	Tipo de entrada	Tipo retornado
std_logic_vector()	unsigned	std_logic_vector
std_logic_vector()	signed	std_logic_vector
unsigned()	std_logic_vector	unsigned
signed()	std_logic_vector	signed

8.3. Outros pacotes

8.3.1. O pacote numeric_std_unsigned

Ao usar o pacote numeric_std, os tipos de dados unsigned e signed devem ser usados para obter acesso aos operadores numéricos. Embora isso forneça controle final sobre o comportamento das operações e comparações de sinais, muitos projetos podem usar apenas tipos unsigned. Para fornecer um mecanismo para tratar todos os vetores como unsigned, deixando seu tipo como std_logic_vector, o pacote numeric_std_unsigned foi criado (LAMERES, 2019).

Quando este pacote é usado, todos os std_logic_vectors no projeto são tratados como unsigned. Este pacote requer que os pacotes std_logic_1164 e numeric_std sejam incluídos anteriormente. Quando usados, todos os sinais e portos podem ser declarados como std_logic / std_logic_vector e serão tratados como unsigned ao executar

operações aritméticas e comparações (LAMERES, 2019). A seguir, é apresentado um exemplo de como incluir este pacote.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.numeric_std_unsigned.all;
```

O pacote `numeric_std_unsigned` contém mais algumas conversões de tipo além do pacote `numeric_std`. Essas conversões adicionais são listadas na Tabela 14.

Tabela 14 - Funções de conversão de tipo do pacote `numeric_std_unsigned`.

Nome	Tipo de entrada	Tipo retornado
`To_integer()`	`std_logic_vector`	`integer`
`To_stdlogicvector()`	`unsigned`	`std_logic_vector`

8.3.2. O pacote `math_real`

O pacote `math_real` fornece computação numérica para o tipo `real`. O tipo `real` é o tipo VHDL usado para descrever um número de ponto flutuante de 32 bits. Nenhum dos operadores fornecidos no pacote `math_real` é sintetizado. Este pacote é usado principalmente para *testbench* ou descrições em VHDL-AMS. Este pacote é incluído adicionando a seguinte sintaxe no início do arquivo.

```
library IEEE;
use IEEE.math_real.all;
```

O pacote padrão do IEEE `math_real` e `math_complex` define constantes e funções matemáticas em números reais e complexos. Na Tabela 15 são listadas as constates do pacote `math_real`.

Tabela 15 - Constantes definidas no pacote `math_real`.

Constante	Valor	Constante	Valor
`math_e`	e	`math_log_of_2`	ln2
`math_1_over_e`	$\frac{1}{e}$	`math_log_of_10`	ln10
`math_pi`	π	`math_log2_of_e`	$\log_2 e$
`math_2_pi`	2π	`math_log10_of_e`	$\log_{10} e$
`math_1_over_pi`	$\frac{1}{\pi}$	`math_sqrt_2`	√2
`math_pi_over_2`	$\frac{\pi}{2}$	`math_1_over_sqrt_2`	$\frac{1}{\sqrt{2}}$
`math_pi_over_3`	$\frac{\pi}{3}$	`math_sqrt_pi`	√π
`math_pi_over_4`	$\frac{\pi}{4}$	`math_deg_to_rad`	$\frac{2\pi}{360}$
`math_3_pi_over_2`	$\frac{3\pi}{2}$	`math_rad_to_deg`	$\frac{360}{2\pi}$

As constantes definidas no pacote `math_real` têm precisão total de 32 bits. O pacote `math_real` fornece um conjunto de operadores de ponto flutuante comumente usados para o tipo real. Já na Tabela 16, são listadas as funções definidas no pacote `math_real`

Tabela 16 - Funções definidas no pacote `math_real`.

Função	Significado	Função	Significado
`ceil(x)`	Teto de x (menor inteiro ≤ x)	`sign(x)`	sinal de x (-1, 0 ou 1)
`floor(x)`	Piso de x (maior inteiro ≥ x)	`"mod"(x,y)`	Módulo de ponto flutuante de x/y
`round(x)`	x arredondado para o valor inteiro mais próximo	`realmax(x,y)`	Maior de x e y
`trunc(x)`	x truncado em direção à zero	`realmin(x,y)`	Menor de x e y
`sqrt(x)`	$\sqrt{x}$	`log(x)`	$\ln x$
`cbrt(x)`	$\sqrt[3]{x}$	`log2(x)`	$\log_2 x$
`"**"(n,y)`	n^y	`log10(x)`	$\log_{10} x$
`"**"(x,y)`	x^y	`log(x,y)`	$\log_y x$
`exp(x)`	e^x		
`sin(x)`	seno de x (radianos)	`arcsin(x)`	arcsin x
`cos(x)`	cosseno de x (radianos)	`arccos(x)`	arccos x
`tan(x)`	tangente de x (radianos)	`arctan(x)`	arctan x
		`arctan(y,x)`	Arco de tangente do ponto (x,y)
`sinh(x)`	seno hiperbólico de x	`arcsinh(x)`	Arco de seno hiperbólico de x
`cosh(x)`	cosseno hiperbólico de x	`arccosh(x)`	Arco de cosseno hiperbólico de x
`tanh(x)`	tangente hiperbólico de x	`arctanh(x)`	Arco de tangente hiperbólico de x

8.3.3. O pacote `math_complex`

O pacote `math_complex` fornece computação numérica para números complexos. Novamente, nada neste pacote é sintetizável e normalmente é usado apenas para *testbench* ou descrições em VHDL-AMS. Este pacote é incluído adicionando a seguinte sintaxe no início do arquivo:

```
library IEEE;
use IEEE.math_complex.all;
```

Este pacote define três novos tipos de dados: `complex`, `complex_vector` e `complex_polar`. O tipo `complex` é definido com dois campos, real e imaginário. O tipo `complex_vector` é uma matriz linear do tipo `complex`. O tipo `complex_polar` é definido com dois campos, magnitude e ângulo. Este pacote fornece um conjunto de operações comuns para uso com números complexos. Este pacote também suporta os operadores aritméticos `+`, `-`, `*` e `/` para tipo `complex`. A lista completa de operadores é listada na Tabela 17 e a Tabela 18 lista as funções definidas no pacote `math_complex`.

Tabela 17 - Operadores sobrepostos no pacote `math_complex`.

Operador	Operação	Operando Esquerdo	Operando Direito	Resultado
=	Igualdade	Polar Complexo	Polar Complexo	Booleano
/=	Desigualdade	Polar Complexo	Polar Complexo	Booleano
abs	Magnitude		Complexo	Real Positivo
			Polar Complexo	Real Positivo
-	Negação		Complexo	Complexo
			Polar Complexo	Polar Complexo
+	Adição	Complexo	Complexo	Complexo
-	Subtração	Real	Complexo	Complexo
*	Multiplicação	Complexo	Real	Complexo
/	Divisão	Polar Complexo	Polar Complexo	Polar Complexo
		Real	Polar Complexo	Polar Complexo
		Polar Complexo	Real	Polar Complexo

Tabela 18 - Funções definidas no pacote `math_complex`.

Função	Significado	Resultado
cmplc(x,y)	$x+jy$	Complexo
get_principal_value(x)	$x+2\pi k$ para algum	Valor Principal
complex_to_polar(c)	c na forma polar	Polar Complexo
polar_to_complex(p)	p na forma cartesiana	Valor Principal
arg(z)	arg(z) em radianos	Mesmo de z
conj(z)	Conjugado Complexo de z	Mesmo de z
sqrt(z)	$\sqrt{z}$	Mesmo de z
exp(z)	e^z	Mesmo de z
log(z)	$\ln z$	Mesmo de z
log2(z)	$\log_2 z$	Mesmo de z
log10(z)	$\log_{10} z$	Mesmo de z
log(z,y)	$\log_y z$	Mesmo de z
sin(z)	seno de z	Mesmo de z
cos(z)	cosseno de z	Mesmo de z
sinh(z)	seno hiperbólico de z	Mesmo de z
cosh(z)	cosseno hiperbólico de z	Mesmo de z

9. INTRODUÇÃO AO VHDL-AMS

A diferença fundamental entro do VHDL-AMS para o VHDL é a capacidade de descrição e simulação. Enquanto o VHDL é capaz de descrever somente sistemas eletrônicos digitais. O VHDL-AMS descreve sistemas analógicos e sistemas de sinais mistos. Mas o que são sistemas analógicos e mistos? São todos os sistemas que não são representados de maneira digital, ou seja, sistemas modelados de acordo com as características elétricas (por exemplo, em nível de tensão ou corrente), mecânicas (por exemplo, torque ou aceleração), rotacional (por exemplo, ângulos ou torque) e pneumáticos (por exemplo, pressão). E sistemas mistos são sistemas representados em mais de uma forma diferente na mesma modelagem (por exemplo a modelagem de um sistema microcontrolado que monitora a temperatura ambiente, ele possui os valores físicos do ambiente, o nível de tensão lido por um sensor analógicos, a conversão desse sinal para digital com um ADC (*Analag to Digital Converter*), e por fim o processamento desse sinal digitalizado pelo núcleo do microcontrolador).

Evidente que uma boa modelagem de sistemas em VHDL-AMS precisa de um conhecimento sobre a natureza dos fenômenos físicos, o qual não faz parte do escopo deste livro. Partimos do princípio que o leitor já possui um conhecimento mínimo nas disciplinas de Física I, II e III, Processamento de Sinais Digitais, Matemática Discreta, entre outras que podem ajudar a esclarecer sobre o assunto.

O leitor que já possui um conhecimento prévio de sistemas digitais pode ainda se perguntar, mas sistemas digitais descritos em VHDL, são em sua essência sistemas analógicos, já que são sistemas elétricos, então porque usar o VHDL-AMS? Sim, sistemas digitais são na verdade sistemas elétricos, contudo, as descrições em VHDL são representações discretas em um nível diferente de abstração, ou seja, os sinais recebem apenas uma quantidade limitada de valores. Além de que os sintetizadores não conseguem sintetizar as descrições mais abstratas que recebem valores como inteiros e reais, o que pode ser aperfeiçoado pelo VHDL-AMS. Outro ponto importante é que o VHDL-AMS estende em muito a capacidade de representação, e consequente descrição, dos sinais, aumentando em muito o leque de opções do estudante ou projetista.

Dado todo esse contexto, é necessário ter em mente que o processo de sintetização do VHDL-AMS é bem mais complexo do que VHDL. Já que, uma ferramenta de síntese que seja capaz de analisar uma descrição em VHDL-AMS, em seus vários tipos de sinais e abstrações, e transforma-la em um hardware equivalente para implementação, pode se tornar inviável dado o tempo necessário. Porém, não é impossível. Já que dispositivos modernos como PSoC (*Programmable System-on-Chip*), possuem em um mesmo encapsulamento, microcontroladores, FPGAs e FPAA (*Field-Programmable Analog Array*). Claro, existe uma limitação para essa possibilidade. O que leva o VHDL-AMS a ser usado principalmente na simulação de sistemas analógicos ou mistos mais robustos.

9.1. Domínios e níveis de modelagem

No VHDL-AMS pode haver diferentes modelos de um sistema, cada um focado em diferentes aspectos. Podemos classificar esses modelos em três domínios: **funcional**, **estrutural** e **geométrico**. O domínio funcional está relacionado às operações executadas pelo sistema. De certa forma, esse é o domínio mais abstrato da descrição, pois não indica como a função é implementada. O domínio estrutural lida com a forma como o sistema é composto de subsistemas interconectados, ambas similares ao VHDL. O domínio geométrico lida com a forma como o sistema é organizado no espaço físico (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O VHDL-AMS suporta a modelagem de sistemas de tempo contínuo descritos com conjuntos de equações simultâneas. Essas equações são equações diferenciais e algébricas com conjuntos de incógnitas que são funções analíticas contínuas no tempo. Podemos modelar a função de sistemas analógicos em diferentes níveis de abstração. Primeiro, podemos formar modelos no nível do sistema usando funções de transferência. Podemos decompor o sistema em subsistemas e empregar um conjunto de macromodelos para cada um. Finalmente, podemos decompor ainda mais os macromodelos em modelos no nível do dispositivo. A complexidade dos modelos aumenta à medida que modelos mais detalhados são desenvolvidos e as representações estruturais, funcionais e geométricas são refinadas (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Os aspectos estruturais de um sistema são refinados a partir de visões abstratas que podem consistir em um conjunto de blocos funcionais conectados ou funções de transferência de sistema, para uma coleção de macromodelos e redes, depois para fontes, dispositivos e interconexões ideais e, finalmente, para dispositivos primitivos, incluindo fios e canais. Da mesma forma, os aspectos funcionais de um sistema são refinados de modelos abstratos usando equações simples, para equações que representam componentes ideais e, em seguida, para equações de modelagem mais complexas que incluem efeitos de segunda ordem do dispositivo. Os aspectos físicos de um sistema podem abranger desde placas de circuito ou módulos multichip, até células de *floorplain*, depois para o *layout* do dispositivo e, finalmente, para polígonos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Podem ser utilizadas equações algébricas e diferenciais ordinárias como *framework* de modelagem de sistemas analógicos ou que acoplam múltiplos domínios de energia. Conjuntos de equações diferenciais e algébricas podem ser classificados como conservativas e não conservativas. Conjuntos conservativos representam sistemas no qual a energia é conservada, enquanto que conjuntos não conservativos representam sistemas onde a energia não é conversada (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Sistemas elétricos conservativos usualmente não possuem efeitos de campos elétricos. Sistemas mecânicos conservativos surgem de sistemas em equilíbrio dinâmico. Conjuntos não conservativos de equações podem ser usados para modelar sistemas em níveis abstratos, para modelar sistema onde os fenômenos físicos que governam o comportamento não são precisamente entendidos ou aperfeiçoar a velocidade de simulação quando aproximações não diminuem a velocidade do modelo (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Os modelos não precisam ser puramente estruturais ou puramente comportamentais. Muitas vezes, é útil especificar um modelo com algumas partes compostas de instâncias de componentes interconectadas e outras partes descritas usando processos ou instruções simultâneas. Utilizamos sinais, terminais e quantidades como meio de unir instâncias de componentes, processos e declarações simultâneas (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Podemos escrever esse modelo híbrido incluindo instância de componente, processo e instruções simultâneas no corpo de uma arquitetura. Instruções de instância e processo de componente são chamadas instruções simultâneas, pois, todos os processos correspondentes são executados simultaneamente quando o modelo é simulado (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10. SINAIS DE OUTRAS NATUREZAS

Usualmente, as propriedades de um sistema são modeladas em termos de nós do circuito. Em VHDL-AMS as propriedades são representadas por terminais. Um terminal é de uma natureza específica, a qual define valores contínuos associados com o terminal. A natureza representa o domínio de energia para o terminal.

Existem dois tipos que se referem à declaração natural, são: `across` e `through`. O tipo `across` modela o esforço ou a força associada com o domínio de energia representado e o tipo `through` modela o fluxo e velocidade. Cada um desses tipos deve ser do tipo ou subtipo ponto flutuante e pode incluir um grupo de tolerância (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O identificador imediatamente antes a palavra-chave `reference` chama-se terminal de referência para a declaração natural. A quantidade `across` do terminal é reservada para ser o terminal de referência de sua natureza, e podem ser declaradas como:

```
subtype tensao is real tolerance "default voltage";
subtype corrente is real tolerance "default current";
subtype eletrica is tensao through ref_eletrica reference;
```

A Tabela 19 ilustra os domínios de energia possíveis.

Tabela 19 - Domínios de energia.

Variável Generalizada	Elétrica	Tradução Mecânica	Rotação Mecânica	Fluido
Esforço ()	Tensão ()	Velocidade ()	Velocidade angular ()	Pressão ()
Fluxo ()	Corrente ()	Força ()	Torque ()	Taxa de fluxo volumétrico ()
Momento ()	Ligação de fluxo ()	Momento ()	Momento angular ()	Momento de pressão ()
Estado ()	Carga ()	Tradução ()	Ângulo ()	Volume ()
Energia ()	$\int_q Vdq$, $\int_\lambda Id\lambda$	$\int_x Fdx$, $\int_p Idp$	$\int_\theta \tau d\theta$, $\int_{p_\theta} \omega dp_\theta$	$\int_V PdV$, $\int_{p_v} \Phi dp_v$
Potência ()	V(t)×I(t)	F(t)×V(t)	τ(t)×ω(t)	P(t)×Φ(t)

A Tabela 19 anterior ilustra o esforço e fluxo para um número de domínios comuns e mostra como os tipos `across` e `through` podem ser combinados com outras variáveis generalizadas para modelar os domínios (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10.1. Quantidades livres

Existem três tipos de quantidades em VHDL-AMS: livre, ramificada e origem. Uma quantidade livre é um objeto analógico valorado que pode ser usado na modelagem de fluxo de sinal. Uma quantidade ramificada é similar, mas especialmente usada para modelar sistemas de energia conservativa. Uma quantidade origem é usada para modelar frequência e ruído. A regra de sintaxe para uma quantidade livre é:

```
quantity identificador {,...}: identificacao_subtipo [:=expressao];
```

Esta quantidade pode ser usada em equações sem aderir às leis de energia conservativa. Isto é útil para modelar o comportamento do fluxo de sinais ou para modelar um subconjunto de fenômenos físicos associados com um determinado domínio de energia (ASHENDEN, PETERSON e TEEGARDEN, 2003).

É possível incluir uma mistura de portos de sinais e quantidades para sinais mistos e modelos de tecnologia mista:

```
([signal] identificador {,...}: [modo] identificacao_subtipo [:=expressao]
 |quantity identificador {,...}: [in|out] identificacao_subtipo [:=expressao]
) {,...}
```

10.2. Quantidades ramificadas e terminais

Terminais é uma característica em VHDL-AMS para representar pontos de conexão físicos ou nós do circuito. Terminais são declarados para ser de várias naturezas que representam diferentes domínios de energia de um sistema. A regra de sintaxe é:

```
terminal identificador {,...}: identificacao_subnatureza;
```

Para referenciar aspectos `across` e `through` de terminais é necessário declarar quantidades ramificadas entre os terminais. As quantidades ramificadas são usadas em equações do modelo. A regra de sintaxe é:

```
declaracao_de_quantidade_ramificada <=
      quantity [aspecto_across][aspecto_through] aspecto_terminal;
aspecto_across <=
      identificador {,...} [tolerance expressao][:=expressao] across
aspecto_through <=
      identificador {,...} [tolerance expressao][:=expressao] through
aspecto_terminal <=
      nome_terminal_positivo [to nome_terminal_negativo]
```

10.3. Declaração break e descontinuidade

Quando valores de um sistema são modelados continuamente, às vezes existe a necessidade de introduzir um comportamento "descontínuo". Isso acontece em sistemas de sinais mistos onde um processo representando uma parte digital do sistema muda as condições de operação de uma parte analógica. É possível indicar a descontinuidade usando a declaração `break`. Usualmente, descontinuidade ocorre quando um ou mais sinais digitais mudam de valor. Porém, o inverso não é verdadeiro. Podem existir eventos em sinais digitais que não causam descontinuidade em quantidades analógicas, particularmente em sistemas predominantemente digitais. Se o simulador assumir que todos os eventos digitais produzem descontinuidade, a simulação pode ser muito ineficiente e vagarosa (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe para a declaração `break` sequencial é:

```
[rotulo:] break [elemento_de_quebra {,...}][when expressao_booleana];
elemento_de_quebra <=
      [for nome_da_quantidade use] nome_da_quantidade => expressao
```

VHDL-AMS também fornece uma declaração `break` concorrente para indicar a descontinuidade no modelo.

```
[rotulo:] break [elemento_de_quebra {,...}]
            [on nomde_do_sinal]
            [when expressao_boolena];
```

Uma declaração `break` concorrente é uma notação abreviada para um processo contendo uma declaração `break` sequencial com o mesmo elemento de quebra e cláusula de condições. A declaração sequencial `break` é seguida por uma declaração `wait` (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10.4. Especificação de limite de passos

Já que quantidades em um modelo analógico são funções de tempo contínuo, existe um número infinito de quantidades de valores em um intervalo de tempo. Um solucionador analógico em um simulador aproxima a quantidade de valores para resolver as equações do modelo em um número finito de passos de tempo. A solução em cada passo é chamada de ponto de solução analógica. A linguagem VHDL-AMS não especifica detalhes de como o tempo e frequência no qual os pontos de solução são calculados. Em alguns casos é desejável assegurar que os pontos de solução analógicos ocorram dentro de uma janela de tempo ou com a mesma frequência mínima. É possível fazer essa tarefa usando a especificação de limite de passos (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
limit (nome_da_quantidade {,...} | others | all): tipo with expressao_real;
```

10.5. Procedimentos

Quando escrevemos modelos comportamentais mais complexos, é útil dividir o código em seções, cada uma lidando com uma parte do comportamento. O VHDL-AMS fornece um recurso de criação de "subprogramas" para nos permitir fazer isso. Existem dois tipos de subprogramas: procedimentos e funções. A diferença entre os dois é que um procedimento encapsula uma coleção de instruções sequenciais que são executadas, semelhante aos processos, enquanto uma função encapsula uma coleção de instruções que calculam um determinado resultado. Assim, um procedimento é uma generalização de uma declaração, enquanto uma função é uma generalização de uma expressão (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe para declaração de um procedimento é:

```
subprograma_corpo <=
      procedure identificador [(lista_de_parametros)] is
     { parte declarativa }
      begin
     { declaração sequencial }
      end [ procedure ] [ identificador ];
```

10.5.1. Declaração `return` em procedimentos

Algumas vezes é útil habilitar o retorno de um procedimento antes que termine toda a sua execução como uma forma de atender a uma condição excepcional. A regra de sintaxe é:

```
retorna_declaracao <= [rotulo]: return;
```

O rótulo opcional permite identificar a declaração de retorno. O efeito na declaração `return` é que quando ela é executada em um procedimento ele é imediatamente terminado e o controle é transferido de volta ao chamador (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10.5.2. Parâmetros de procedimentos

Um procedimento parametrizado é muito mais geral em que ele pode executar seu algoritmo usando diferentes objetos de dados ou valores cada vez que ele é chamado. O objetivo é que o chamador passe parâmetros para o procedimento como parte da chamada ao procedimento. E o procedimento então execute suas declarações usando os parâmetros (ASHENDEN, PETERSON e TEEGARDEN, 2003).

A regra de sintaxe para a lista de procedimentos é:

```
lista_interface <=
      ([constant | variable | signal]
      identificador {,...}: [modo] indicacao_subtipo [:=expressao]) {,...}
modo <= in | out | inout
```

A lista de interfaces de parâmetros é similar à lista de portos na declaração de entidade. A principal diferença é que a lista de parâmetros não pode ser incluída quantidades ou terminais.

A terceira classe de objetos que podem ser especificados para parâmetros formais é o sinal, o qual indica que o algoritmo executado pelo procedimento invoca um sinal passado pelo chamador (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10.6. Funções

Pode-se pensar em funções como generalização de expressões. Uma função é uma forma de definir uma nova operação que pode ser usada em expressões. Podem ser definidas como novas as operações que trabalham escrevendo uma coleção de declarações sequenciais que calculam o resultado parâmetros (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
corpo_subprograma <=
      [pure | impure]
      function identificador [(lista_de_parametros)] return marca_de_tipo is
            {item_declarativo_de_subprograma}
      begin
            {declaracoes_sequenciais}
      end [function][identificador];
```

A lista de parâmetros de uma função tem a mesma forma que para o procedimento com duas restrições. Primeiro, os parâmetros de uma função não pode ser uma classe variável. Se a classe não for explicitamente mencionada, ela é assumida ser uma constante. Segundo o modo de cada parâmetro deve ser do modo `in` (entrada).

10.6.1. Função `now`

O VHDL-AMS fornece uma função pré-definida, `now`, que retorna o tempo de simulação atual quando ela é chamada. Como o VHDL-AMS é uma linguagem de modelagem de sinais mistos suportando o comportamento analógico de tempo contínuo e o comportamento digital de tempo discreto, existem duas versões da função `now`. Uma versão retorna o tempo como um valor discreto de tipo `delay_length` e a outra versão retorna o tempo de simulação como valor contínuo do tipo `real` (ASHENDEN, PETERSON e TEEGARDEN, 2003).

10.7. Declaração `procedural` simultânea

Uma declaração `procedural` permite expressar um comportamento analógico usando declarações sequencias. Diferente de procedimentos e funções, `procedural` são declaradas na parte declarativa de um modelo e não são invocadas usando mecanismos de chamada. Ela contrasta com o estilo de modelagem declarativo de outras declarações simultâneas, onde as igualdades são expressas restringindo quantidades, sem indicar diretamente como a quantidade pode ser calculada (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
declaracao_procedural_simultanea <=
      [rotulo:]
      procedural [is]
            {item_declarativo_de_subprograma}
      begin
            {declaracoes_sequenciais}
      end procedural [rotulo];
```

11. MODELAGEM BASEADA EM FREQUÊNCIA

Em sistemas elétricos, são comuns filtros que permitem passar ou amplificar certas faixas de frequência ou atenuam outras. Em sistemas mecânicos muitas vezes existem frequências ressonantes por oscilações. Sistemas de controle empregam realimentação no qual o assunto relacionado à frequência é um aspecto chave do comportamento do sistema (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O VHDL-AMS fornece a habilidade de caracterizar uma função de transferência no domínio da frequência. Dado um sinal de entrada, a função de transferência descreve a saída como uma função da magnitude e frequência senóide de entrada. Se o sinal de entrada é visualizado como uma soma de senóides, a função de transferência produz uma saída que é a sobreposição de cada senóide de entrada (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O VHDL-AMS não fornece um mecanismo de transformada de Laplace ou Fourier para traduzir entre o domínio do tempo e da frequência. As funções de transferência z e Laplace fornece um mecanismo para expressar o comportamento do sistema como uma função de frequência, mesmo quando o sistema de simulação precede em um domínio de tempo longo (ASHENDEN, PETERSON e TEEGARDEN, 2003).

A formulação matemática usada tem impacto significativo no tempo de simulação do modelo. Para reduzir o tempo de execução da simulação de pequenos sinais de frequência e a probabilidade de problemas na resolução do sinal, simuladores usam uma forma linearizada do circuito no ponto de operação CC (Corrente Contínua). O VHDL-AMS define formas linearizadas para uso em frequência de pequenos sinais e simulação de ruídos para permitir o uso de sobreposição (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.1. Quantidades de fonte espectral

Para simulações da frequência de pequenos sinais, simuladores VHDL-AMS usam fontes espectrais que são especificadas no modelo como quantidades de fonte espectral. Estas quantidades são estímulos para simulação no domínio da frequência. O sinal senoidal pode ser caracterizado pela equação

O simulador escolhe os valores para a frequência , e os valores são especificados para a magnitude e a fase na declaração. Durante a simulação no domínio da frequência, o simulador inclui equações características derivadas da declaração da fonte espectral no conjunto de argumentos no domínio da frequência (ASHENDEN, PETERSON e TEEGARDEN, 2003). As regras de sintaxe simplificadas são:

```
declaracao_de_quantidade <=
         quantity identificador {,...}: subtipo aspecto_fonte;
aspecto_fonte <=
         spectrum expressao_de_magnitude, expressao_de_fase
```

As expressões de magnitude e fase em uma declaração de quantidade de fonte espectral deve ser do mesmo tipo que a quantidade fonte. Isto assegura que existem valores de magnitude e fase correspondente a cada elemento escalar da quantidade fonte. As expressões não são obrigatoriamente estáticas, para que elas possam se referir à quantidade e sinais definidos no modelo (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.2. Modelagem de ruídos

Quando o comportamento de um sistema é modelado, frequentemente é necessário analisar o impacto de dispositivos geradores de ruídos em um circuito ou a sensibilidade do sistema a ruídos de entrada. O VHDL-AMS fornece uma segunda forma de quantidades fontes, uma quantidade fonte de ruído, para permitir executar a análise do ruído (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe completa é:

```
aspecto_fonte <=
        spectrum expressao_de_magnitude, expressao_de_fase
      | noise expressao_de_potencia
```

Um tipo comum de ruído é o ruído térmico, que surge devida a movimentos térmicos aleatórios dos elétrons. Ruído térmico é altamente dependente da temperatura de um resistor ou dispositivo ativo, mas não depende de um campo elétrico (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.3. Função de transferência de Laplace

Muitas vezes é conveniente especificar as funcionalidades do sistema em termos de função de transferência de tempo continuo. A função de transferência de Laplace implementa uma relação de dois polinômios no operador de Laplace. No domínio do tempo, isso corresponde a equações diferenciais ordinárias com coeficientes constantes.

Em VHDL-AMS a função de transferência de Laplace é dada pelo atributo `'ltf`. Dado uma quantidade escalar, o atributo `Q'ltf(num,den)` produz uma quantidade cujo tipo base de e cujo valor é a transformada de Laplace de . Os polinômios numerador e denominador, `num` e `den`, respectivamente, são especificados como vetores do tipo `real_vector`, com a exigência de que o primeiro elemento do vetor denominador não deve ser zero. Os vetores polinomiais numerador e denominador devem ser constantes para permitir uma transformação analítica das expressões características resultantes no domínio da frequência (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.4. Amostras e funções de transferência discreta

Em muitos sistemas é importante modela-los utilizando funções de transferência discretas. Em VHDL-AMS existem três atributos de quantidade `'zoh`, `'ztf` e `'delayed` para suportar essa atividade de modelagem. Os atributos `'zoh` e `'ztf` são usados especificamente para a modelagem discreta, enquanto `'zoh`, `'delayed` é um atributo de atraso de propósito geral (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.4.1. Amostras de ordem zero (zoh)

O atributo `'zoh` atua como uma função “amostra” e “segura”. Ele permite a amostragem periódica de uma quantidade. O atributo é uma primitiva útil cujo comportamento não pode ser descrito utilizando a linguagem VHDL-AMS, já que ele é fornecido como um atributo construído dentro da linguagem. Por exemplo, dada uma quantidade , o atribuito `Q'zoh(T, atraso_inicial)` produz uma quantidade que o valor de amostrado no tempo determinado por `atraso_inicial` e em intervalos . A quantidade pode ser qualquer tipo e o resultado do atributo é do mesmo tipo de . O intervalo de amostragem é uma expressão estática do tipo `real`, desde que seu valor é maior que zero. Da mesma forma, o atraso inicial é uma expressão estática do tipo `real`, desde que seu valor seja não negativo. Pode-se omitir a expressão `atraso_inicial`, escrevendo `Q'zoh(T)`, caso em que a primeira amostra é tomada no momento zero (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.4.2. Função de transferência z

O atributo `'ztf` ou função de transferência no domínio z, implementa uma relação de dois polinômios na variável , o que representa um atraso por uma constante .

Como a função de transferência de Laplace, a função de transferência no domínio z é definida como um atributo que produz uma quantidade. Por exemplo, dado um escalar qualquer, o atributo `Q'ztf(num, den, T, atraso_inicial)` produz uma quantidade cujo tipo é o mesmo de e cujo valor é a transformada de no domínio . Os polinômios do numerador e denominador, `num` e `den`, respectivamente, são especificados como vetores do tipo `real_vector`, com a exigência de que o primeiro elemento do vetor denominador não deve ser zero.

Os vetores polinomiais numerador e denominador devem ser constantes, para permitir uma transformação analítica das expressões características resultantes no domínio da frequência. A quantidade é amostrada no tempo dado por `atraso_inicial` em intervalos . Tal como acontece com o atributo `'zoh`, a expressão `atraso_inicial` deve ser estática e produzir um valor não negativo `real`. Pode-se omitir a expressão `atraso_inicial`, caso em que a primeira amostra é tomada no momento zero (ASHENDEN, PETERSON e TEEGARDEN, 2003).

11.5. Sinais resolvidos básicos

A abordagem adotada pelo VHDL-AMS para modelagem digital é uma forma muito geral, como: a linguagem exige que o projetista especifique precisamente que valores são conectados às múltiplas saídas. Ele faz isso através de “sinais resolvidos”, que são uma extensão de sinais básicos. Um sinal resolvido inclui na sua definição uma função, chamada “função de resolução”, que é usada para calcular o valor final do sinal a partir dos valores de todas as suas fontes (ASHENDEN, PETERSON e TEEGARDEN, 2003).

12. MAIS SOBRE ESTRUTURAS

A primeira coisa que é necessária fazer para descrever uma interconexão de subsistemas em um projeto é descrever os diferentes tipos de componentes utilizados. É possível fazer isso escrevendo declarações de entidade para cada um dos subsistemas. Cada declaração de entidade é uma unidade de projeto separada e tem arquiteturas correspondentes que descrevem implementações. Uma abordagem alternativa é escrever "declarações de componentes" na parte declarativa da arquitetura (ASHENDEN, PETERSON e TEEGARDEN, 2003).

12.1. Instancias de configuração de componentes

Uma vez que temos descrito a estrutura de um nível de um projeto utilizando componentes e instancias de componentes, ainda será preciso aprofundar a implementação hierárquica para cada instancia de componente.

Pode-se fazer isso escrevendo uma "declaração de configuração" para o projeto. Esse é um caso mais elaborado e robusto do que a declaração e utilização de componentes já explicada anteriormente. Nele, especifica-se qual interface de entidade e arquitetura correspondente deverá ser usada para cada uma das instancias de componente. Isso é chamado de "ligação" a instancias de componentes para entidades de projeto. Note que não é especificada qualquer informação vinculada para uma declaração de instanciação direta de componente de uma entidade, uma vez que entidade e a arquitetura são especificadas na declaração de instanciação de componentes (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O caso mais simples surge quando as entidades às quais estão vinculadas as instâncias de componentes são implementadas com arquitetura comportamental. Neste caso existe apenas um nível de hierarquia (ASHENDEN, PETERSON e TEEGARDEN, 2003). A sintaxe simplificada é:

```
declaracao_de_configuracao <=
      configuration identificador of nome_da_entidade is
            for nome_da_arquitetura
                  {for especificacao_do_componente
                        indicacao_de_ligacao;
                  end for;}
            end for;
      end [configuration][identificador];
especificacao_do_componente <=
      (rotulo_instanciacao {,...} | others | all): nome_do_componente
indicacao_de_ligacao <=
      use entity nome_da_entidade [(identificador_da_arquitetura)]
```

Os componentes no nível superior têm arquiteturas que contem instancias de componentes que devem ser configuradas. As arquiteturas vinculadas a esses componentes de segundo nível também podem conter instancias de componentes, e assim por diante (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
indicacao_de_ligacao <=
       use configuration nome_configuracao
```

Pode-se fazer uso direto de uma entidade de projeto totalmente configurado dentro de uma arquitetura escrevendo uma declaração de instanciação de componente que diretamente nomeia a configuração (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe alternativa para as declarações de instanciação de componentes que expressa essa possibilidade é:

```
declaracao_de_instancia_de_componente <=
     [rotulo:]
      configuration nome_da_configuracao
     [generic map (lista_de_associacao_generica)]
     [port map (lista_de_associacao_de_portos)];
```

Agora se volta a um aspecto muito poderoso de configuração de componentes: a inclusão de mapeamento genéricos e mapeamento de portos nas indicações de ligação. Esta facilidade fornece uma grande flexibilidade quando uma instancia de componente é ligada a entidade de projeto (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe estendida para uma indicação de ligação que mostra como os mapeamentos genéricos e os mapeamentos de portos podem ser incluídos:

```
indificacao_de_ligacao <=
       use (entity nome_da_entidade [(identificador_arquitetura)]
             | configuratiom nome_configuracao)
      [generic map (lista_de_associacao_generica)]
      [port map (lista_de_associacao_de_portos)]
```

Existe ainda uma terceira opção para especificar a instanciação de componente, que é deixar desconectada a instancia do componente e adiar vincula-lo em uma etapa posterior do projeto. A regra de sintaxe é:

```
indificacao_de_ligacao <=
    use open;
```

12.2. Estrutura iterativa generate

Se for necessário replicar um subsistema, pode-se usar uma declaração `generate`. Esta é uma declaração simultânea contendo outras declarações concorrentes e simultâneas que devem ser replicadas. Declarações `generate` são particularmente uteis se o número de vezes que é necessário replicar as declarações simultâneas não é fixo, por exemplo, a partir do valor de uma constante genérica (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
declaracao_generate <=
      [rotulo:]
      for identificador in faixa_discreta generate
            [{item_de_declaracao_de_blocos}
      begin]
            {declaracao_concorrente | declaracao_simultanea }
      end generate [rotulo];
```

12.3. Estrutura generate condicional

Em alguns projetos, existem células especificas que precisam ser tratadas de forma diferente. Isso muitas vezes ocorre onde as células são conectadas a seus vizinhos. As células em cada extremidade não têm vizinhos de ambos os lados, mas estão ligados a sinais, quantidades, terminais ou portos na arquitetura em questão. Pode-se lidar com esses casos especiais dentro de uma estrutura `generate` usando uma instrução condicional (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
declaracao_generate <=
      [rotulo:]
      if expressao_booleana generate
            [{item_de_declaracao_de_blocos}
      begin]
            {declaracao_concorrente | declaracao_simultanea}
      end generate [rotulo];
```

12.4. Declaração generate de configurações

Se um projeto inclui uma declaração iterativa `generate`, é necessário identificar as células individuais de iteração, a fim de configura-las. Se o projeto inclui uma declaração `generate` condicional, é necessário ser capaz de incluir informações de configuração que devem ser usadas somente se a célula está incluída no projeto. A fim de lidar com esses casos, usa-se uma forma estendida de configuração de bloco (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
configuracao_de_bloco <=
      for (nome_da_arquitetura
            | rotulo_de_declaracao_do_bloco
            | rotulo_de_declaracao_generate
                  [((faixa_discreta | expressao_estatica ))])
            {configuracao_do_bloco
            | for especificacao_do_componente
                  [indicacao_de_ligacao;]
                  [configuracao_do_bloco]
            end for;}
      end for;
```

A nova parte nesta regra é a alternativa que permite configurar uma declaração `generate` escrevendo seu rotulo. A parte opcional após o rotulo é usado apenas para a declaração `generate`. Esta parte nos permite escrever uma expressão cujo valor seleciona uma célula em particular a partir da estrutura iterativa ou um intervalo de valores que selecionam um conjunto de células. Uma vez que foi identificado a declaração `generate`, as informações de configuração restantes dentro do bloco de configuração especifica como as instruções simultâneas dentro das células geradas serão configuradas (ASHENDEN, PETERSON e TEEGARDEN, 2003).

12.5. Sinais guardados e desconexão

Se um sistema está sendo modelado em um nível muito alto de abstração, pode ser necessário usar um tipo de dado mais abstrato como um tipo `integer` ou um tipo pouco simples para representar sinais. Em tais casos, não é apropriado incluir o estado de alta impedância como um valor, então o VHDL-AMS proporciona uma abordagem alternativa, usando sinais guardados. Estes são sinais resolvidos para qual dos drivers que podem ser desconectados (ASHENDEN, PETERSON e TEEGARDEN, 2003). A regra de sintaxe é:

```
declaracao_do_sinal <=
       signal identificador {,...}: identificacao_de_subtipo [register | bus]
           [:= expressao];
```

12.6. Tipos de acesso

Tipos de dados de acesso são semelhantes aos encontrados em tipos de ponteiro em muitas linguagens de programação tradicional. Em VHDL-AMS, tipos de acesso são usados principalmente em alto nível de modelos comportamentais e raramente em modelos de baixo nível.

É possível declarar um tipo de acesso usando uma nova forma de definição de tipo:

```
definicao_tipo_acesso <=
      access indificacao_de_subtipo
```

Uma vez que foi declarado um tipo de acesso, uma variável pode ser declarada com o tipo dentro de um processo ou subprograma. Inicialmente a variável possui o valor `null`. Depois se pode criar um novo objeto de dados. Faz-se isso usando um alocador escrito de acordo com a regra de sintaxe:

```
primario <=
     new indificacao_de_subtipo | new expressao_qualificada
```

Agora que existe uma variável de acesso para apontar um objeto de dados em memória, pode-se usar o valor do objeto acessando-o através da variável. Este uso da variável é a razão para ser ter “tipo de acesso” e “variável de acesso”. O objeto é acessado usando a palavra `all` como um sufixo após o nome da variável de acesso (ASHENDEN, PETERSON e TEEGARDEN, 2003).

12.7. Estruturas de dados ligadas

Suponha que é necessário armazenar uma lista de valores a serem utilizados para estimular um sinal durante uma simulação. Uma abordagem possível seria definir uma matriz ou vetor de valores de estímulos. No entanto, surge um problema se não se sabe o quão grande deve ser feita a matriz. A abordagem alternativa é a utilização de tipos de acesso e criar valores quando eles forem necessários. Os valores podem ser ligados com ponteiros para formar uma estrutura de dados extensível. Existem várias organizações possíveis para estruturas ligadas, mas será demonstrado um dos mais simples, uma "lista ligada" (ASHENDEN, PETERSON e TEEGARDEN, 2003). Como por exemplo:

```
type value_cell;
type value_ptr is access value_cell;
type value_cell is record
     value: real_vector (0 to 3);
     next cell: value_ptr;
end record value_cell;
```

13. MAIS SOBRE ATRIBUTOS PRÉ-DEFINIDOS

Atributos pré-definidos são usados para recuperar informações sobre tipos, objetos e outros itens em um modelo. Resumiremos os atributos e descrevemos completamente os atributos pre-definidos restantes, agora disponíveis somente no VHDL-AMS. O primeiro grupo de atributos predefinidos fornece informações sobre os valores em um tipo escalar, os quais são listados na Tabela 20.

Tabela 20 - Os atributos predefinidos dando informações sobre os valores em um tipo.

Atributo	Tipo de T	Tipo de Resultado	Resultado
`T'left`	Algum tipo escalar ou subtipo	Mesmo de T	Valor mais é esquerda em T
`T'right`	Algum tipo escalar ou subtipo	Mesmo de T	Valor mais à direita em T
`T'low`	Algum tipo escalar ou subtipo	Mesmo de T	Menor valor em T
`T'high`	Algum tipo escalar ou subtipo	Mesmo de T	Maior valor em T
`T'ascending`	Algum tipo escalar ou subtipo	Booleano	Verdadeiro se T é um intervalo ascendente
`T'image(x)`	Algum tipo escalar ou subtipo	Caractere	Uma representação textual do valor x do tipo T
`T'values(s)`	Algum tipo escalar ou subtipo	Tipo de base de T	Valor em T representado pela string s
`T'pos(s)`	Qualquer tipo discreto ou físico ou subtipo	Inteiro	Número da posição de x em T
`T'val(x)`	Qualquer tipo discreto ou físico ou subtipo	Tipo de base de T	Valor na posição x em T
`T'succ(x)`	Qualquer tipo discreto ou físico ou subtipo	Tipo de base de T	Valor em uma posição maior do que x em T
`T'pred(x)`	Qualquer tipo discreto ou físico ou subtipo	Tipo de base de T	Valor em uma posição maior do que x em T
`T'leftof(x)`	Qualquer tipo discreto ou físico ou subtipo	Tipo de base de T	Valor em uma posição para a esquerda de x em T
`T'rightof(x)`	Qualquer tipo discreto ou físico ou subtipo	Tipo de base de T	Valor em uma posição para a direita de x em T
`T'base`	Qualquer tipo ou subtipo	Tipo de base de T	Tipo de base de T, apenas para uso como prefixo de um outro atributo

O segundo grupo de atributos predefinidos fornece informações sobre os valores de natureza ou subnatureza escalar e estão resumidos na Tabela 21.

Tabela 21 – Os atributos predefinidos dando informações sobre os valores de uma natureza.

Atributo	Natureza de N	Resultado
`N'across`	Qualquer natureza ou sub-natureza	Tipo across de N
`N'through`	Qualquer natureza ou sub-natureza	Tipo through de N

O terceiro grupo de atributos predefinidos fornece informações sobre os valores de índice de um objeto, tipo ou matriz e estão resumidos na Tabela 22. O prefixo `T` na tabela refere-se a um tipo ou subtipo de matriz restrita, a uma natureza ou sub-natureza de matriz restrita, a um objeto de matriz ou a uma parte de uma matriz. Se `T` é uma variável de um tipo de acesso apontando para um objeto de matriz, o atributo se refere ao objeto de matriz, não ao valor do ponteiro. Cada um dos atributos opcionalmente usa um argumento que seleciona uma das dimensões de índice da matriz. Observe que, se o prefixo `T` for um alias para um objeto de matriz, os atributos retornarão informações sobre os valores de índice declarados para o alias, não sobre os declarados para o objeto original. (ASHENDEN, PETERSON e TEEGARDEN, 2003)

Tabela 22 – Os atributos predefinidos dando informações sobre o intervalo de índice de um vetor.

Atributo	Resultado
`T'left(n)`	Valor na faixa mais à esquerda do índice de dimensão n
`T'right(n)`	Valor na faixa mais à direita do índice de dimensão n
`T'low(n)`	Menor valor do índice na faixa de dimensão n
`T'high(n)`	Maior valor do índice na faixa de dimensão n
`T'range(n)`	Faixa de índice de dimensão n
`T'reverse_range(n)`	Faixa de índice de dimensão n invertida em direção e limites
`T'length(n)`	Comprimento da faixa de índice de dimensão n
`T'ascending`	Verdadeiro se faixa de índice de dimensão n é ascendente

O quarto grupo de atributos predefinidos fornece informações sobre sinais ou define novos sinais implícitos derivados de sinais explicitamente declarados e estão resumidos na Tabela 23. O prefixo `S` na tabela refere-se a qualquer sinal estaticamente nomeado. Três dos atributos opcionalmente usam um argumento não negativo `t` do tipo `time`. O atributo `'ramp` opcionalmente utiliza um argumento de tempo ascendente do tipo `time` e um argumento de tempo descendente opcional do tipo `time`. O atributo `'slew` opcionalmente aceita um argumento de sinal ascendente, que é uma expressão estática do tipo `real`, bem como um argumento opcional de de sinal descendente, também uma expressão estática do tipo `real` (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Tabela 23 - Os atributos predefinidos dando informações sobre o intervalo de índice de um vetor.

Atributo	Tipo de Resultado	Resultado
`S'delayed(t)`	Tipo de base de S	Sinal implícito, com o mesmo valor de S, mas foi adiado por unidades de tempo t (t ≥ 0 ns)
`S'stable(t)`	Booleano	Sinal implícito, verdadeiro quando nenhum evento ocorreu em S para unidades de tempo t (t ≥ 0 ns)
`S'quiet(t)`	Booleano	Sinal implícito, verdadeiro quando nenhuma transação ocorreu em S para unidades de tempo t (t ≥ 0 ns)
`S'transaction`	Bit	Sinal implícito, mudanças de valor em ciclos de simulação em que uma transação ocorre em S
`S'event`	Booleano	Verdadeiro se um evento ocorreu em S no ciclo de simulação
`S'active`	Booleano	Verdadeiro se uma transação ocorreu em S no ciclo de simulação
`S'last event`	Time	Tempo desde a último evento ocorrido em S, ou `time'high` se nenhum evento ocorreu ainda
`S'last active`	Time	Tempo desde a última transação ocorrida em S, ou `time'high` se nenhuma transação ocorreu ainda
`S'last value`	Tipo de base de S	Valor de S antes do evento ocorrido pela última vez sobre ele
`S'driving`	Booleano	Verdadeiro se o processo está direcionando S (ou a cada elemento de um sinal composto S), ou falso se o processo desconectou seu driver de S (ou qualquer elemento de S), com uma transição nula
`S'driving value`	Tipo de base de S	Valor contribuído pelo driver para S no processo
`S'ramp(trise, tfall)`	Tipo de base de S	Quantidade que cada subelemento escalar segue o subelemento correspondente de S com uma mudança linear de valor em unidades de tempo `trise` (se houver) para valores crescentes e, em todas as unidades de tempo `tfall` (se houver) para os valores em queda
`S'slew(rslope, fslope)`	Tipo de base de S	Quantidade que cada subelemento escalar segue o subelemento correspondente de S com uma mudança linear de valor governado por uma inclinação máxima de aumento `rslope` (se houver) e uma inclinação máxima queda de `fslope` (se houver)

O quinto grupo de atributos predefinidos fornece informações sobre terminais e estão resumidos na Tabela 24. O prefixo `T` na tabela refere-se a qualquer terminal.

Tabela 24 - Os atributos predefinidos dando informações sobre terminais.

Atributo	Tipo de Resultado	Resultado
`T'reference`	Tipo `across` da natureza de T	Quantidade de referência de T
`T'contribution`	Tipo `through` da natureza de T	Quantidade de contribuição de T
`T'tolerance`	Caracteres	Grupo de tolerância de T

O sexto grupo de atributos predefinidos fornece informações sobre quantidades e estão resumidos na Tabela 25. O prefixo `Q` na tabela refere-se a qualquer quantidade. Três dos atributos opcionalmente usam um argumento não negativo `t` do tipo `time`.

Tabela 25 - Os atributos predefinidos dando informações sobre quantidades.

Atributo	Tipo de Resultado	Resultado
`Q'tolerance`	Caracteres	Grupo de tolerância de Q
`Q'dot`	Mesmo tipo de Q	Derivada com respeito ao tempo de Q
`Q'integ`	Mesmo tipo de Q	Integral do tempo de Q desde o tempo zero
`Q'delayed(t)`	Mesmo tipo de Q	Quantidade igual a Q mas atrasada por t
`Q'above(E)`	Booelano	Verdadeiro se Q > E
`Q'zoh(t, delay)`	Tipo de base de Q	Uma quantidade cujo valor é o de Q amostrados inicialmente em atraso e cada intervalo `t`
`Q'ltf(num, den)`	Tipo de base de Q	Função de transferência Laplace de cada subelemento escalar de Q com `num` como numerador e `den` como os coeficientes do polinômio denominador
`Q'ztf(num,den,t,delay)`	Tipo de Q	Função de transferência z-domínio de cada subelemento escalar de Q com `num` como numerador e `den` como os coeficientes do polinômio denominador, `t` como o período de amostragem, e com `delay` o tempo da primeira amostragem
`Q'slew(rslope, fslope)`	Tipo de base de Q	Quantidade que cada subelemento escalar segue o subelemento escalar correspondente de Q, mas com sua derivada em relação ao tempo limitado pelos slope especificado

Os demais atributos predefinidos são aplicados a qualquer item declarado e retornam uma representação de sequência do nome do item. Esses atributos estão resumidos na Tabela 26. O prefixo `X` na tabela refere-se a qualquer item declarado. Se o item for um alias, o atributo retornará o nome do próprio alias, não o item alias. O atributo `'simple_name` retorna uma representação em `string` do nome de um item (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Tabela 26 - Os atributos predefinidos que fornecem nomes de itens declarados.

Atributo	Resultado
`X'simple_name`	`String` representando o símbolo de caractere identificador, ou operador definido na declaração do item `X`
`X'path_name`	`String` descrevendo o caminho através da hierarquia do projeto elaborado, da entidade de alto nível ou pacote para o item `X`
`X'instance_name`	`String` similar ao produzido pela `X'path_name`, mas incluindo os nomes da entidade e arquitetura ligada a cada instância de componente no caminho

13.1. Atributos definidos pelo projetista

O VHDL-AMS fornece uma maneira de adicionar informações de escolha do projetista de itens nos modelos, através de atributos definidos pelo ele. Estes atributos podem ser usados para adicionar informações de projeto físico, tais como alocação de células, as restrições de *layout*, como atraso máximo de fio e inclinação "*interwire*" ou informações para síntese como codificações de tipos de enumerados e dicas sobre alocação de recursos. Em geral as informações de natureza não estruturais e não comportamentais podem ser adicionadas usando atributos e processados utilizando ferramentas de software que operam no banco de dados do projeto (ASHENDEN, PETERSON e TEEGARDEN, 2003).

O primeiro passo na definição de um atributo é declarar o nome e o tipo de um atributo, usando uma declaração de atributo. A regra de sintaxe é:

```
declaracao_de_atributo <=
     attribute identificador: marca_do_tipo;
```

Uma declaração de atributo simplesmente define o identificador como representando um atributo definido pelo utilizador que pode assumir valores de tipo especificado. O tipo pode ser qualquer tipo VHDL-AMS, exceto a um tipo de acesso, arquivo, protegido ou um tipo composto com um sub-elemento, que pode ser de acesso, arquivo ou protegido.

Uma vez que tenhamos definido um nome de atributo e tipo, ele é usado para declarar itens dentro do projeto. As especificações de atributos são escritas com itens de nomeação que o atributo pode assumir com valores particulares (ASHENDEN, PETERSON e TEEGARDEN, 2003). As regras de sintaxe para uma especificação de atributo são:

```
especificacao_de_atributo <=
      attribute identificador of lista_nome_entidade: classe_entidade
            is expressao;
lista_nome_entidade <=
      ((identificador | caractere_literal | simbolo_operador)[assinatura]
{,...}
      | others | all
classe_entidade <=
      entity          | architecture    | configuration    | package
      | procedure | function  | type           | subtype
      | constant  | signal    | variable  | file
      | component | label          | literal   | units
      | group          | nature    | subnature | quantity
      | terminal
```

13.2. Grupos

O VHDL-AMS fornece um mecanismo de agrupamento para identificar uma coleção de itens sobre os quais mantem alguma relação. As informações sobre a relação se expressam como um atributo do grupo.

A primeira etapa nos itens de grupos é definir um modelo para as classes de itens que podem ser incluídos no grupo. Faz-se isso com uma declaração de modelo de grupo, para o qual a regra de sintaxe é:

```
declaracao_do_grupo <=
     group identificador is ((classe_entidade [<>]) {,...});
```

Pode-se usar este modelo para definir uma serie de grupos com base em declarações do grupo. A regra de sintaxe é:

```
declaracao_do_grupo <=
     group identificador: nome_do_grupo
          ((nome | caractere_literal) {,...});
```

13.3. Processos adiados

O VHDL-AMS fornece uma facilidade, processos adiados, que é útil em modelos de atraso delta (ASHENDEN, PETERSON e TEEGARDEN, 2003). Um processo é feito adiado incluindo a palavra-chave `postponed`, como mostrado pela regra de sintaxe completa:

```
declaracao_de_processo <=
      [rotulo_processo:]
      [postponed] process [(nome_do_sinal {,...})][is]
            {itens_declarados_no_processo}
      begin
            {declaracao_sequencial}
      end [postponed] process [rotulo_processo];
```

Um processo adiado é disparado, mas não executa no mesmo ciclo de simulação.

14. SIMULAÇÃO ROBUSTA: ARQUIVOS

Um arquivo VHDL-AMS é uma classe de objeto usado para armazenar dados. Assim, como acontece com outras classes de objetos, devem-se incluir definições de tipo de arquivo nos modelos. A regra de sintaxe para definir um tipo de arquivo é:

```
definicao_de_tipo_de_arquivo <=
     file of marca_do_tipo;
```

Um arquivo só pode conter um tipo de objeto, mas este tipo pode ser quase qualquer tipo VHDL-AMS, incluindo os tipos escalares, estrutura de dados ligadas e matrizes unidimensionais. Os únicos tipos que não podem ser armazenados em arquivos são matrizes multidimensionais, tipos de acesso e tipos protegidos (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Uma vez que se tem definido um tipo de arquivo, pode-se declarar objetos de arquivos. Faz-se isso com uma nova forma de declaração do objeto, descrito pela regra de sintaxe:

```
declaracao_de_arquivo <=
      file identificador {,...}: indificacao_de_subtipo
           [[open expressao_do_tipo_de_arquivo_aberto] is expressao_de_
texto];
```

14.1. O pacote textio

O pacote `textio` oferece a capacidade de ler e gravar de ou para entrada / saída externa. E / S externa refere-se a itens como arquivos ou a entrada / saída padrão de um computador. Este pacote contém funções que permitem que os valores de sinais e variáveis sejam lidos e gravados, além de `strings`. Isso permite que mensagens de saída mais sofisticadas sejam criadas em comparação apenas com a declaração `report`, que pode apenas gerar sequências de caracteres.

A capacidade de ler valores de um arquivo permite que padrões de teste sofisticados sejam criados fora do VHDL ou do VHDL-AMS e depois sejam lidos durante a simulação para testar um sistema. É importante ter em mente que o termo “E / S” se refere a arquivos externos ou à janela de transcrição, não às entradas e saídas de um modelo de sistema (LAMERES, 2019).

O pacote `textio` não é sintetizável e é usado apenas em *testbenches*. O pacote `textio` está dentro da biblioteca `STD` e é incluído em um projeto usando a seguinte sintaxe.

```
library STD;
use STD.textio.all;
```

Este pacote, por si só, suporta leitura e gravação dos tipos `bit`, `bit_vector`, `integer`, `character` e `string`. Como a maioria dos projetos sintetizáveis usa os tipos `std_logic` e

std_logic_vector, foi criado um pacote adicional que adicionou suporte para esses tipos. O pacote é chamado std_logic_textio e está localizado na biblioteca IEEE. A sintaxe para incluir este pacote está abaixo (LAMERES, 2019).

```
library IEEE;
use IEEE.std_logic_textio.all;
```

O pacote textio define dois novos tipos de interface com E / S externa. Esses tipos são file e line. O tipo file é usado para identificar ou criar um arquivo para leitura ou gravação no projeto. A sintaxe para declarar um arquivo é a seguinte:

```
file file_handle : <arquivo_tipo> open <arquivo_modo> is <"arquivo_nome">;
```

A declaração de um arquivo abrirá automaticamente o arquivo e o manterá aberto até o final do processo que o está usando. O file_handle é um identificador exclusivo para o arquivo usado em procedimentos subsequentes. O nome do identificador de arquivo é definido pelo projetista. Um identificador de arquivo elimina a necessidade de especificar o nome completo do arquivo sempre que um procedimento de acesso a arquivos é chamado. O arquivo_tipo descreve as informações dentro do arquivo. Existem dois tipos de arquivos suportados, TEXT e INTF (LAMERES, 2019).

Um arquivo TEXT é aquele que contém cadeias de caracteres. Esse é o tipo mais comum de arquivo usado, pois existem funções que podem ser convertidas entre os tipos string, bit ou bit_vector e std_logic ou std_logic_vector. Isso permite que todas as informações no arquivo sejam armazenadas como caracteres, o que torna o arquivo legível por outros programas.

Um tipo de arquivo INTF contém apenas valores inteiros e as informações são armazenadas como um número binário com sinal de 32 bits. O arquivo_modo descreve se o arquivo será lido ou gravado. Existem dois modos suportados, WRITE_MODE e READ_MODE.

O nome do arquivo é fornecido entre aspas duplas e é definido pelo projetista. É comum inserir uma extensão no arquivo para que ele possa ser aberto por outros programas (por exemplo, output.txt). A declaração de um arquivo sempre ocorre dentro de um processo antes da instrução de início do processo (LAMERES, 2019). A seguir, exemplos de como declarar arquivos.

```
file Fout: TEXT open WRITE_MODE is "output_file.txt";
file Fin: TEXT open READ_MODE is "input_file.txt";
```

As informações em um arquivo são acessadas (lidas ou gravadas) usando o conceito de uma linha. No pacote textio, um arquivo é interpretado como uma sequência de linhas, cada uma contendo uma sequência de caracteres ou um valor inteiro. A tipo line é usado como um *buffer* temporário ao acessar uma linha dentro do arquivo. Essa variável do tipo line é usada para armazenar as informações lidas de uma linha no arquivo ou para armazenar as informações que devem ser gravadas em uma linha do arquivo. Uma variável é necessária para esse comportamento, pois as atribuições de ou para o arquivo devem ser feitas imediatamente. Como tal, uma variável line é sempre declarada dentro de um processo antes do início da instrução do processo (LAMERES, 2019). A sintaxe para declarar uma variável do tipo linha é a seguinte:

```
variable <variavel_nome> : line;
```

Existem dois procedimentos que permitem que informações sejam transferidas entre uma variável `line` e uma linha em um arquivo. Esses procedimentos são `readline()` e `writeline()`. A sintaxe deles é a seguinte:

```
readline(<file_handle>, <variavel_nome>);
writeline(<file_handle>, <variavel_nome>);
```

A transferência de informações entre uma variável `line` e uma linha em um arquivo usando esses procedimentos é realizada em toda a linha. Não há mecanismo para ler ou gravar apenas uma parte da linha em um arquivo. Depois que um arquivo é aberto ou criado usando uma declaração de arquivo, as linhas são acessadas na ordem em que aparecem no arquivo. O primeiro procedimento a ser chamado (`readline()` ou `writeline()`) acessará a primeira linha do arquivo. Na próxima vez que um procedimento for chamado, ele acessará a segunda linha do arquivo. Isso continuará até que todas as linhas tenham sido acessadas. O pacote `textio` fornece uma função para indicar quando o final do arquivo foi atingido ao executar `readline()`. Essa função é chamada `endfile()` e retorna o tipo `boolean`. Esta função retornará `TRUE` quando o final do arquivo for atingido (LAMERES, 2019).

Dois procedimentos adicionais são fornecidos para adicionar ou recuperar informações de ou para a variável `line` dentro do *testbench*. Esses procedimentos são `read()` e `write()`. A sintaxe para esses procedimentos é a seguinte:

```
read(<variavel_nome>, <destino_variavel>);
write(<variavel_nome>, <fonte_variavel>);
```

Ao usar o procedimento `read()`, as informações na variável `line` são tratadas como delimitadas por espaço. Isso significa que cada procedimento `read()` recuperará as informações da variável `line` até atingir o caractere de espaço. Isso permite que vários procedimentos `read()` sejam usados para analisar as informações em nomes separados de `destino_variavel`. O `destino_variavel` deve ser do tipo e tamanho adequados das informações que estão sendo lidas no arquivo. Por exemplo, se o campo na linha que está sendo lida for uma sequência de 4 caracteres (`"wxyz"`), uma variável de destino deverá ser definida do tipo sequência de `string(1 to 4)`. Se o campo que está sendo lido for um `std_logic_vector` de 2 bits, uma variável de destino deverá ser definida do tipo `std_logic_vector(1 to 0)` (LAMERES, 2019).

O procedimento `read()` ignorará o caractere de espaço. Ao usar o procedimento `write()`, supõe-se que fonte e destino sejam do tipo `bit`, `bit_vector`, `integer`, `std_logic` ou `std_logic_vector`. Se desejar inserir uma sequência de texto diretamente, a função `string` será usada com formato `string'<"caracteres...">`. Vários procedimentos `write()` podem ser usados para inserir informações na variável `line`. Cada procedimento de gravação subsequente anexa as informações ao final da sequência. Isso permite que diferentes tipos de informações sejam intercalados (por exemplo, texto, valor do sinal, texto etc.) (LAMERES, 2019).

15. VERIFICAÇÃO COM *TESTBENCH*

A verificação funcional dos projetos de VHDL ou VHDL-AMS é realizada através de simulação usando um *testbench*. Mas afinal, o que é um *testbench*? Um *testbench*[13] é um sistema em VHDL ou VHDL-AMS que instancia o sistema a ser testado como um componente e gera os padrões de entrada e observa as saídas. Existe uma variedade de recursos para projetar *testbenches* que podem automatizar a geração de estímulos e fornecer verificação de saída automatizada. Esses recursos podem ser expandidos incluindo pacotes que tiram vantagem da leitura ou gravação em E / S externas. Este capítulo fornece os detalhes dos recursos internos que permitem a criação de *testbench*.

Criar a estratégia de teste para um projeto é uma parte crítica do processo de projeto eletrônico. Nos testes, o sistema que está sendo testado costuma ser chamado de dispositivo em teste (DUT – *Device Under Test*). *Testbench*es são usadas apenas para simulação, para que possamos usar técnicas de modelagem abstrata que não são sintetizáveis para gerar os padrões de estímulo. Existe também funcionalidades específicas para relatar o status de um teste e também verificar automaticamente se as saídas estão corretas (LAMERES, 2019).

Normalmente, é necessário testar um DUT sob todas as condições de entrada possíveis para verificar a funcionalidade. Testar sob todas as condições de entrada pode exigir um grande número de condições de entrada. Como um estudo de caso, considere um somador de bits. Para testar um somador de bits em cada condição de entrada numérica, serão necessários vetores de teste. Para um somador simples de 4 bits, isso equivale a 256 padrões de entrada. Mesmo para um circuito pequeno como esse, o grande número de padrões de entrada impede o uso de atribuições de sinais manuais no *testbench* para estimular o circuito. Uma abordagem para gerar automaticamente um conjunto exaustivo de padrões de teste de entrada é usar aninhados para laços (LAMERES, 2019).

15.1. Checagem automática

15.1.1. Declaração `report`

A declaração `report` pode ser usada em um *testbench* para fornecer o status do teste atual. Uma declaração `report` imprimirá uma sequência na janela de transcrição da ferramenta de simulação. A saída do relatório também contém um nível de gravidade opcional. Existem quatro níveis de gravidade (`ERROR`, `WARNING`, `NOTE` e `FAILURE`). O nível de gravidade `FAILURE` interromperá uma simulação, enquanto os níveis `ERROR`, `WARNING` e `NOTE` permitirá que a simulação continue. Se o nível de gravidade for omitido, o relatório será considerado um nível de gravidade `NOTE` (ASHENDEN, PETERSON e TEEGARDEN, 2003). A sintaxe para usar uma declaração de relatório é a seguinte:

```
report "string a ser impressa" severity <nivel>;
```

15.1.2. Declaração `assert`

A declaração `assert` fornece um mecanismo para verificar uma condição booleana antes de usar a declaração `report`. Isso permite que as saídas do `report` sejam impressas seletivamente com base nos valores dos sinais no sistema em teste.

Isso pode ser usado para imprimir a operação bem-sucedida ou a falha de um sistema. Se a condição booleana associada à declaração `assert` for avaliada como verdadeira, ela não executará a declaração `report` subsequente. Se a condição booleana for avaliada como falsa, ela executará a instrução `report` sequente. A declaração `assert` é sempre usada em conjunto com a declaração `report` (ASHENDEN, PETERSON e TEEGARDEN, 2003). A seguir está a sintaxe para a declaração `assert`.

```
assert condicao report "string" severity <nivel>;
```

15.2. Usando entrada e saída

Quando se deseja relatar grandes quantidades de dados, a gravação na transcrição se torna impraticável e é necessário um arquivo externo. Para gravar em um arquivo externo a partir de um *testbench*, são necessários os pacotes `textio` e `std_logic_textio`. Esse modo de saída é idêntico ao modo como a instrução `report` funciona, mas o uso do pacote `textio` permite mais funcionalidades no texto de saída. A saída padrão de um computador recebe um identificador de arquivo reservado chamado `OUTPUT`. Ao usar esse identificador de arquivo, um novo arquivo não precisa ser declarado no *testbench*, pois já está definido como parte do pacote `textio`. O nome do identificador de arquivo reservado `OUTPUT` pode ser usado diretamente no procedimento `writeline()` (ASHENDEN, PETERSON e TEEGARDEN, 2003).

Referências bibliográficas

ASHENDEN, P. J.; PETERSON, G. D.; TEEGARDEN, D. A. **The System Designers Guide to VHDL-AMS**. California, USA: Elsevier Science, 2003.

BREUER, M. A.; SARRAFZADEH, M.; SOMENZI, F. **Fundamental CAD algorithms**. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, v. 19, n. 12, p. 1449-1475, 2000.

D'AMORE, R. **VHDL: Descrição e síntese de circuitos digitais**. São Paulo: LTC, 2005. 259 p.

GAJSKI, D. D.; KUHN, R. H. **Guest editors' introduction new VLSI tools**. IEEE Computer, v. 16, n. 12, p. 11-14, 1983.

GERSTLAUER, A.; HAUBELT, C.; PIMENTEL, A. D.; STEFANOV, T. P.; GAJSKI, D. D.; TEICH, J. **Electronic system-level synthesis methodologies**. IEEE Transactions on Computer--aided Design of Integrated Circuits and Systems, IEEE, USA, v. 28, n. 10, p. 1517-1530, 2009.

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE STD 1164-1993**: IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164). USA: IEEE, 1993. 24 p.

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. **IEEE STD 1076.1-2017**: IEEE Standard VHDL Analog and Mixed-Signal Extensions. USA: IEEE, 2007. 672 p.

LAMERES, Brock J.. **Quick start guide to VHDL**. Bozeman, MT, USA: Springer, 2019. 215 p.

RIESGO, T.; TORROJA, Y.; TORRE, E. de la. **Design methodologies based on hardware description language**s. IEEE Transactions on Industrial Electronics, v. 46, n. 1, p. 3-12, 1999.


EDUFT
UNIVERSIDADE FEDERAL DO TOCANTINS