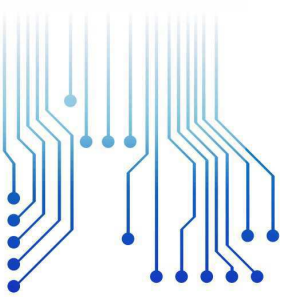


CURSO DE ENGENHARIA ELÉTRICA

MATHEUS ANDRADE DE ALMEIDA

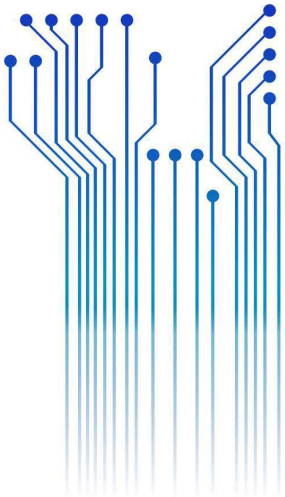
Universidade Federal
de Campina Grande

Centro de Engenharia
Elétrica e Informática

TRABALHO DE CONCLUSÃO DE CURSO

VERIFICAÇÃO FORMAL PARA HARDWARE

Departamento de
Engenharia Elétrica



CAMPINA GRANDE
2018

MATHEUS ANDRADE DE ALMEIDA

VERIFICAÇÃO FORMAL PARA HARDWARE

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de concentração: Microeletrônica

Orientador:
Professor Gutemberg Gonçalves dos Santos Júnior, D.Sc.

CAMPINA GRANDE
2018

MATHEUS ANDRADE DE ALMEIDA

VERIFICAÇÃO FORMAL PARA HARDWARE

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de concentração: Microeletrônica

Aprovado em: 21/12/2018

Professor Marcos Ricardo Alcântara Morais, D.Sc.
Universidade Federal de Campina Grande
Avaliador

Professor Gutemberg Gonçalves dos Santos Júnior, D.Sc.
Universidade Federal de Campina Grande
Orientador

CAMPINA GRANDE
2018

Dedico este trabalho à Deus por toda a força e orientação que me concedeu ao longo desse anos.

AGRADECIMENTOS

Agradeço a Deus, em primeiro lugar, pela minha vida e pelo dom da perseverança, que me permitiu concluir este trabalho.

Agradeço também aos meus pais, Antonio e Fatima, por terem se esforçado tanto para me proporcionar uma boa educação, por terem me alimentado com saúde, força e coragem, as quais que foram essenciais para superação de todas as adversidades ao longo desta caminhada.

Agradeço também a minha noiva, Natalya, por ser a pessoa maravilhosa que ela é na minha vida, por todo apoio e força que ela me deu em todos os momentos, fáceis e difíceis nesses anos.

Agradeço também a toda minha família, pelo auxilio ao longo dos meus estudos sem contar todo carinho e apoio, não medidindo esforços para eu chegar a esta etapa da minha vida.

Aos meus queridos amigos do grupo dos "Normais" que sempre estiveram presentes em todos os momentos bons e ruins, desde antes como depois da graduação.

Agradeço também aos meus amigos advindos do meio universitário, em particular, Jesney, Thiago e Yago pois sem eles não teria chegado até aqui.

A todos os membros do projeto de Excelência em Microeletrônica do Nordeste, por todos os momentos que passamos durante o período no laboratório tanto os de trabalho como os de vadiagem.

Aos professores Marcos Morais e Gutemberg Júnior, por todos os ensinamentos, orientação e oportunidades concedidas, permitindo que eu descobrisse áreas de interesse da minha formação e por acreditarem na microeletrônica no Brasil.

Enfim, agradeço a todos que de alguma forma, passaram pela minha vida e contribuíram para a construção de quem sou hoje.

"Tudo o que fizeres, façam de todo o coração, como para o Senhor, e não para os homens, sabendo que receberão do Senhor a recompensa da herança. É a Cristo, o Senhor, que estão servindo."

Colossenses 3:23-24

RESUMO

A verificação formal tem grande importância no mundo empresarial de tecnologia, devido ao aumento da complexidade dos sistemas de hardware e software o que acarentou em uma maior quantidade de falhas encontradas nos projetos. Por isso que a utilização da verificação formal vem se tornando cada dia mais presente no mercado, isso se dá devido a otimização do tempo de produção que esse método providencia, gerando assim menos gastos durante a confecção dos produtos. Por causa disso é perceptível a necessidade de treinar cada vez mais equipes de verificação em métodos formal, sendo necessário para isso a criação de mais documentações que auxiliem os verificadores na utilização das logicas formais. Tendo esse problema em mente, esse trabalho foi desenvolvido para servir de guia pratico para um verificador poder realizar uma verificação formal em um hardware, fazendo uso da linguagem *SystemVerilog Assertions* em conjunto com um ambiente *open source* que faz uso de UVM, o SVAUnit.

Palavras-chave: Verificação Formal, SystemVerilog Assertions, SVAUnit.

ABSTRACT

The formal verification has great importance in the business world of technology, due to the increase in the complexity of hardware and software systems which has led to a greater number of fail found in the projects. That is why the use of formal verification is becoming more and more present in the market, this is due to optimization of time of production that this method provides, thus generating less expenses during the confection of the products. Because of this, there is a perceived need to train more and more verification teams in formal methods, and it is necessary to create more documentation that will assist verifiers in the use of formal logic. Having this problem in mind, this work was developed to serve as a practical guide for a verifier to be able to perform a formal verification on a hardware, making use of the SystemVerilog Assertions language in conjunction with an open source environment that makes use of UVM, the SVAUnit.

Keywords: Formal Verification, SystemVerilog Assertions, SVAUnit.

LISTA DE FIGURAS

Figura 1 – Visão Geral da Verificação de Alto Nível. 8
Figura 2 – Árvore de Decisão Ordenada. 13
Figura 3 – Diagrama de Decisão Binaria Ordenados e Reduzido. 14
Figura 4 – Os dois casos para um caminho limitado, (a) sem loop, (b) com loop. . . . 15
Figura 5 – Um esboço do sistema STeP. 18
Figura 6 – Camadas do PSL 20
Figura 7 – Exemplo Funcionamento de uma Assertions 24
Figura 8 – Estrutura SVAUnit 30
Figura 9 – Diagrama de fluxo SVAUnit 31
Figura 10 – Estrutura do PULPino 33
Figura 11 – Estrutura da Unidade de Eventos 36
Figura 12 – Bits de eventos 36
Figura 13 – Fluxograma de Verificação Formal 37
Figura 14 – Fluxo de dados analisado no teste do registro PADDIR 39
Figura 15 – Formas de onda da simulação referente a configuração o registrador PADDIR 40
Figura 16 – Formas de onda da simulação com falha para a configuração do registrador PADDIR 41
Figura 17 – *Log* da simulação com falha referente a configuração o registrador PADDIR 41
Figura 18 – Fluxograma dos dados de escritos nos PAD da GPIO até a memoria 42
Figura 19 – Formas de onda das SVAs da operação de escrita na GPIO 43
Figura 20 – *Log* das SVAs da operação de escrita na GPIO 43
Figura 21 – Fluxograma dos dados de leitura nos PAD da GPIO vindo da memoria . . . 45
Figura 22 – Fluxograma dos dados de leitura nos PAD da GPIO vindo da memoria . . . 45
Figura 23 – Fluxograma dos dados de conexão dos barramentos com os periféricos . . . 46
Figura 24 – Mapa de Memoria do PULPino 48
Figura 25 – Fluxograma dos dados com a memoria e da "resposta"do barramento 49

LISTA DE QUADROS

Quadro 1 – Expressão booleana. 20
Quadro 2 – Declaração de *clock* padrão. 21
Quadro 3 – Exemplo de declaração de *clock*. 21
Quadro 4 – Declaração de uma sequência. 22
Quadro 5 – Exemplo declaração de uma sequência. 22
Quadro 6 – Exemplo comparativo de uma assert com if. 24
Quadro 7 – Exemplo declaração de uma assert. 25
Quadro 8 – Exemplo declaração de modalidade de falhas. 25
Quadro 9 – Exemplo declaração de assert property. 26
Quadro 10 – Exemplo declaração de assert property com sequência. 26
Quadro 11 – Exemplo declaração de assert property com sequência em funções separada. 26
Quadro 12 – Exemplo declaração de assert property com disable iff 27
Quadro 13 – Exemplo declaração de cover property. 28
Quadro 14 – Assertion utilizada para verificar a configuração do registrador PADDIR. . 40
Quadro 15 – Assertion utilizada para verificar entrada de dados na GPIO. 42
Quadro 16 – Assertion utilizada para verificar saída de dados na GPIO. 44
Quadro 17 – Assertion utilizada para verificar os barramentos AXI e APB durante uma leitura. 47
Quadro 18 – Assertion utilizada para verificar os barramentos AXI e APB durante uma escrita. 47
Quadro 19 – Assertion utilizada para verificar os sinais de "resposta"do AXI durante uma leitura. 49
Quadro 20 – Assertion utilizada para verificar os sinais de "resposta"do AXI durante uma escrita. 50

LISTA DE TABELAS

Tabela 1 – Quadro comparativo entre simulação e a verificação formal. 3
Tabela 2 – Quadro explicativo dos principais operadores de LTL. 6
Tabela 3 – Quadro explicativo dos principais operadores de CTL 7
Tabela 4 – Diretivas da camada de verificação 23
Tabela 5 – Sintaxe dos operadores de SVA 27
Tabela 6 – Sintaxe dos operadores de SVA combinacionais 28
Tabela 7 – Sinais Externos da UART 33
Tabela 8 – Sinais Externos da GPIO 34
Tabela 9 – Sinais do Mestre SPI 34
Tabela 10 – Sinais do I^2C 35

LISTA DE ABREVIATURAS E SIGLAS

ACL2	*A Computational Logic for Applicative Common Lisp*;
AP	*Atomic Proposition*;
APB	*Advanced Peripheral Bus*;
API	*Application Programming Interface*;
ASIC	*Application Specific Integrated Circuits*;
AXI	*Advanced eXtensible Interface*;
BDD	*Binary Decision Diagrams*;
Coq	*Coenzyme Q10*;
CTL	*Computational Tree Logic*;
ESL	*Electronic System-Level*;
GPIO	*General-Purpose Input/Output*;
HOL	*Higher Order Logic*;
HLV	*High-Level Verification*;
I^2C	*Inter–Integrated Circuit*;
JTAG	*Joint Test Action Group*;
LTL	*Linear Temporal Logic*;
PSL	*Property Specification Language*;
PVS	*Prototype Verification System*;
RTL	*Register Transfer Level*;
SoC	*System On Chip*;
SMT	*Satisfiability Modulo Theories*;
SMC	*Symbolic Model Checking*;
SPI	*Serial Peripheral Interface*
STeP	*The Stanford Temporal Prover*;
SVA	*SystemVerilog Assertions*;
ROBDD	*Reduced Ordinary Binary Decision Diagrams*;
TL	*Temporal Logic*;
UART	*Universal Asynchronous Receiver/Transmitter*;
UVM	*Universal Verification Methodology*.

SUMÁRIO

1 – Introdução **1**
1.1 Objetivos 2
1.1.1 Gerais 2
1.1.2 Específicos 2

2 – Revisão Bibliográfica **3**
2.1 Especificações da Verificação de Hardware 4
2.1.1 Especificações na Lógica Temporal 5
2.1.1.1 Lógica Temporal de Tempo Linear - LTL 5
2.1.1.2 A lógica da árvore de computação - CTL 6
2.1.2 Especificação com modelos de alto nível 8
2.1.2.1 Mecanismos de Abstração 9
2.1.2.2 Explicação da Lógica 10
2.1.2.3 Refinamento 11
2.2 Técnicas de Verificação 11
2.2.1 Verificação de Modelo 12
2.2.1.1 Verificação de Modelo Simbólico - SMC 12
2.2.1.2 Verificação de Modelo Limitado - BMC 14
2.2.2 Métodos dedutivos 15
2.2.2.1 Sistemas de Prova de Teorema 16
2.2.3 Combinando verificação de modelo e raciocínio dedutivo 17
2.3 Linguagens de Verificação Formal 19
2.3.1 Property Specification Language - PSL 19
2.3.1.1 Estrutura da Linguagem 19
2.3.2 SystemVerilog Assertions - SVA 23
2.3.2.1 Estrutura da Linguagem 24

3 – Ambiente de Verificação Formal em Hardware **29**
3.1 SVAUnit 29
3.1.1 SVAUnit Testbench 30
3.1.2 SVAUnit Test 30
3.1.3 SVAUnit Test Suite 31
3.1.4 Fluxo 31

4 – Descrição do Hardware **32**
4.1 Parallel Ultra Low Power Processor - PULPino 32
4.1.1 UART 33
4.1.2 GPIO 33
4.1.3 Mestre SPI 34
4.1.4 I^2C 35
4.1.5 Timer 35
4.1.6 Event Unit 35
4.1.7 Controle do SoC 36

5 – Guia Para Realizar Verificação Formal em Hardware **37**

5.1 Definição . . . 37
5.2 Planejamento de Verificação . . . 38
5.3 Ambientação . . . 38
5.4 Simulação . . . 38

6 – ANÁLISE E DISCUSSÃO DOS RESULTADOS . . . 39
6.1 Plano de Verificação . . . 39
6.1.1 GPIO . . . 39
6.1.2 Barramentos AXI-APB . . . 45
6.1.3 Teste de Acesso Reservado . . . 48
6.2 Discursão . . . 50

7 – Conclusão . . . 51

Referências . . . 52

Anexos 54

ANEXO A – SVAUnit Testbench para múltiplas interfaces parametrizadas . . . 55

ANEXO B – SVAUnit Test . . . 56

ANEXO C – SVAUnit Test Suit . . . 57

1 Introdução

Com o avanço da tecnologia e o aumento na complexidade dos sistemas de hardware e software se tornou cada vez maior a probabilidade de erros sutis no desenvolvimento ocorrerem. Além disso, alguns desses erros podem causar perda catastrófica de dinheiro, tempo ou até mesmo da vida humana, dessa forma a necessidade de garantir o funcionamento adequado dos equipamentos se tornou cada vez mais importante. Partindo desse preceito os engenheiros de verificação ganharam cada vez mais espaço e importância dentro das industrias, tornando-se ferramentas indispensáveis no processo de produção.

A metodologia mais utilizada pelos verificadores na realização de suas atividades é a verificação funcional, método esse que consiste em desenvolver modelos do sistema, os quais simulam o funcionamento do produto analisado. Entretanto esse modelo apresenta algumas limitações, sendo o mais agravante no que diz respeito ao design de sistemas complexos. Simulações de design são muitos custosas e demandam tempo, além de ser quase impossível possibilitar uma simulação completa e ideal do projeto. Atualmente, como solução para esses problemas, os projetistas começaram a usar métodos formais para realizar a verificação na maioria dos produtos, entretanto, ainda há uma grande lacuna para a verificação de projetos de grande porte que possam ser desenvolvidos. Estes projetos não podem ser verificados completamente tendo em vista a complexidade dos problemas inerentes ao sistema. Tal lacuna fez com que o mundo acadêmico e empresarial enfrentassem os desafios de reduzi-la ou até mitigá-la, possibilitando soluções novas e engenhosas para especificar, projetar, estruturar e aplicar casos de testes usando a verificação formal Edgar e David (2014).

A verificação formal consiste em técnicas que aplicam raciocínio e linguagem matemática no projeto, com a finalidade de verificar se as especificações iniciais permanecem as mesmas durante a implementação do hardware. A exatidão proporcionada pelo formalismo matemático reduz a ambiguidade, erros e inconsistências do projeto, além de viabilizar a automação de testes e até a geração de código. Com essa detecção é possível superar os desafios da simulação, pois todos os valores de entrada possíveis podem ser explorados matematicamente ou exaustivamente, o que significa dizer que para conseguir um alto grau de observação do produto não é necessário exagerar no design ou na criação de múltiplos cenários para verificação. O uso de métodos formais não garante, a priori, a correção, No entanto, eles podem aumentar muito a nossa compreensão de um sistema, revelando inconsistências, ambiguidades e incompletudes que poderiam passar despercebidas.

A formalização matemática se dá por meio da especificação do sistema e de provas de correção da sua funcionalidade. Trata-se de um trabalho que apresenta certa complexidade e, por ser bastante minucioso e exaustivo, demanda muito tempo, porém, os resultados o tornam justificável e viável. A fim de facilitar tal tarefa e evitar que sejam introduzidos erros oriundos de falha humana, foram criados os provadores de teorema, ou assistentes de prova, programas os quais servem para auxiliar o verificação durante a execução de seus testes.

Tendo em vista o ganho obtido ao fazer uso da verificação formal em um projeto de desenvolvimento de hardware, percebe-se a necessidade de ter acesso um guia de fácil compreensão e entendimento do procedimento necessário para a criação e utilização de um ambiente de verificação formal, para que os verificadores possam realizá-la sem demandar um longo tempo de estudo, otimizando o tempo de projeto.

1.1 Objetivos

A proposta do presente trabalho consiste em aplicar os conhecimentos adquiridos nas disciplinas de Arquitetura de Sistemas Digitais, Arquitetura Avançadas para Computadores e Circuitos Lógicos, a fim de apresentar o modelo de verificação formal em hardwares.

1.1.1 Gerais

Propor um guia do procedimento e a metodologia que se deve seguir para se realizar uma verificação formal em hardware, utilizado como análise um Processador Paralelo de Ultra Baixa Potência (*Parallel Ultra Low Power Processor* - PULP) mais especificamente PULPino: Um SoC RISC-V de Núcleo Único (*A Single-Core RISC-V SoC*) e seus periféricos.

1.1.2 Específicos

- Apresentar a verificação formal em hardware;
- Apresentar as linguagens de programação utilizadas para se realizar uma verificação formal;
- Definir qual a linguagem mais adequada de acordo com a sua complexidade e sua eficiência;
- Descrever as etapas que se deve realizar para poder executar uma verificação formal;
- Desenvolver o código exemplificativos de verificação formal com o PULPino e seus periféricos por meio da linguagem escolhida;
- Analisar os resultados da verificação formal e discutir a respeito de sua viabilidade em um projeto.

2 Revisão Bibliográfica

No contexto de sistemas de hardware e software, a verificação formal é um processo sistemático que usa o raciocínio matemático para verificar se a especificação do projeto é preservada na implementação (RTL). Pois esta explora de maneira algorítmica e exaustiva, todos os possíveis valores de entrada ao longo do tempo, podendo dessa forma superar todos os três desafios de simulação mostrados na Tabela 1.

Tabela 1 – Quadro comparativo entre simulação e a verificação formal.

	Simulação	**Verificação Formal**
Exaustão (Capacidade de observar todos os possíveis cenários de entrada)	* Não é possível simular todos os estados possíveis em um design, mesmo com 100 s de CPUs e meses de simulação. * Foco em cenários e asserções para quebrar o design.	* Explorar todos os estados possíveis. * Resulta em RTL de alta qualidade. * Mudar o foco no comportamento funcional correto da intenção.
Controlabilidade (Capacidade de ativar, estimular ou sensibilizar um ponto específico dentro do projeto)	* Deve-se conceber cenários de vetores para "adequadamente" simular o design. * Probabilidade de perder os cenários de caso de canto.	* Nenhum estímulo requerido. * Começar cedo no ciclo de design. * Todos os erros de caso de especificos são possiveis de ser capturados.
Observabilidade (Capacidade de observar os efeitos de um ponto específico, interno e estimulado dentro do projeto)	* Deve propagar erros para os pinos de saída ou inserir asserções locais para expor erros e ajudar a depurar.	* Isolar automaticamente a causa raiz de bugs. * Visualizar comportamentos incorretos e corrigi-los.

Fonte: (SANGHAVI, 2010)

A verificação é feita fornecendo-se uma prova formal através de um modelo matemático abstrato do sistema, sendo possível por meio de diversas abordagens sendo uma delas a verificação de modelo (*Model checking*). Esse método consiste em uma exploração sistematicamente exaustiva do modelo matemático, isso é possível tanto para modelos finitos, como também para alguns modelos infinitos onde conjuntos infinitos de estados podem ser efetivamente representados finitamente usando abstração ou aproveitando-se de simetria.

Dessa maneira as propriedades a serem verificadas são frequentemente descritas em lógicas temporais, como lógica temporal linear (*Linear temporal logic* - LTL), Linguagem de Especificação de Propriedade (*Property Specification Language* - PSL), Asserções SystemVerilog (*SystemVerilog Assertions* - SVA), ou lógica de árvore computacional (Computational tree logic - CTL). A grande vantagem da verificação de modelos é que muitas vezes é totalmente automática e sua principal desvantagem é que, em geral, não é de viável para para sistemas de grande escala.

Outra abordagem é a verificação dedutiva que consiste em gerar a partir do sistema e de suas especificações uma coleção de obrigações de prova matemática, cuja veracidade implica a conformidade do sistema com sua especificação e o cumprimento dessas obrigações usando provadores de teoremas interativos, provadores de teoremas automáticos ou solucionadores de teorias de módulo de satisfazibilidade (*Satisfiability modulo theories* - SMT). Essa abordagem

tem a desvantagem de normalmente exigir que o usuário entenda detalhadamente seu funcionamento do sistema. Para que o mesmo passa realizar um projeto de verificação que funcione corretamente e transmita as informações para o sistema, seja na forma de uma sequência de teoremas a serem comprovados ou na forma de especificações de componentes do sistema.

Por meio dessa breve explanação é possível ver a complexidade que a verificação formal possui bem como a grande variedade de metodologias que podem ser empregadas em um projeto, portante apresentaremos mais detalhadamente esses métodos assim como as suas linguagens de programação formal, ao longo desse trabalho iniciando nossas atividades com uma introdução estruturas que são adequadas para a formalização de especificações e descrições de implementação de projetos de hardware, seguindo com a analise de metodologias, posteriormente sera apresentado um guia de qual o procedimento a se seguir para realizar uma verificação formal, juntamente com um exemplo de estudo de caso, por fim sera discutido os resultados do trabalho bem como uma conclusão a cerca do uso da verificação formal em hardwares.

2.1 Especificações da Verificação de Hardware

Dentre as especificações da verificação de hardware exites duas abordagens que apresentam maior destaque com relação as demais. Sendo a primeira que diz respeito a condições de propriedades para o design, sabendo que a verificação formal é geralmente relacionada com propriedades temporais, ou seja, elas pertencem a atributos que variam de acordo com o *clock* do sistemas e não de maneira estática. As lógicas temporais são uma estrutura unificadora para expressar tais propriedades, dessa maneira a verificação tem o objetivo de comprovar se todo o sistema apresenta comportamentos que satisfazem as propriedades temporais da sua especificação.

A segunda abordagem por outro lado é baseada na especificação em termos de um modelo de alto nível do sistema. Nesse modelo as condições validas do sistema são dadas pelo conjunto de todos os comportamentos do modelo de alto nível, em vez de um conjunto de propriedades temporais. A verificação, por sua vez, tem como objetivo mostrar que cada comportamento possível da implementação em RTL do sistema é consistente com algum comportamento de sua especificação de alto nível.

No fluxo de um projeto de verificação formal não é obrigatório escolher uma das duas metodologias para se seguir, uma vez que elas, não são excludentes entre se. Dessa maneira as abordagens geralmente são usadas em conjunto, sendo primeiro mostrado um modelo de alto nível do design satisfazendo um conjunto de propriedades temporais desejadas. Em seguida, uma série de especificações cada vez mais detalhadas é desenvolvida, cada uma das quais é uma implementação da especificação no próximo nível superior.

Em uma estrutura técnica apropriada, as propriedades temporais do modelo de nível mais alto são preservadas pelas etapas de refinamento e, portanto, são satisfeitas pelo nível mais baixo e mais detalhado. Nesse contexto, o primeiro tipo de verificação também é chamado de verificação de design ou propriedade, enquanto o segundo formulário é conhecido como verificação de implementação. Note que a distinção entre as duas formas é apenas conceitual. Ambos os tipos de verificação são instâncias do mesmo problema: a especificação define alguma restrição sobre os comportamentos permitidos de um sistema, e a verificação requer mostrar que a implementação atende a essa restrição.

Nesta seção, introduzimos formalismos para ambas as abordagens. A Seção 2.1.1 trata da especificação de propriedades temporais, enquanto a Seção 2.1.2 descreve a especificação em termos de modelos de alto nível.

2.1.1 Especificações na Lógica Temporal

Lógicas temporais (*Temporal Logic* - TL) são lógicas modais projetada para especificar relações temporais entre eventos que ocorrem ao longo do tempo Blackburn, Rijke e Venema (2001). A TL pode expressar condições especificas que tenham relação temporal, ou seja, que apresentem uma marcação de uma ação que depende de um tempo. Como por exemplo "a propriedade p vale sempre" ou "se *p* é válido em algum instante no tempo, *q* deve eventualmente se manter em algum momento posterior". Proposições desse tipo podem ser empregadas para especificar as propriedades desejadas dos sistemas, como "este controlador de barramento sempre concederá no máximo uma solicitação para o barramento por vez" e "todas as solicitações serão eventualmente concedidas".

Existem muitas formas diferentes de lógica temporal, em particular, há várias opções com relação ao modelo de tempo subjacente, como tempo de ramificação versus tempo linear, tempo discreto versus contínuo, ou o uso de operadores temporais em pontos versus intervalos no tempo. Nesta seção, primeiro apresentamos a lógica temporal de tempo linear (*Linear temporal logic* - LTL), como um exemplo representativo de lógicas temporais e demonstramos como ela pode ser usada para propósitos de especificação. Mais tarde, examinamos outras variantes da lógica temporal proposicional mais brevemente.

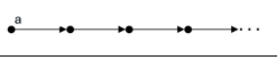
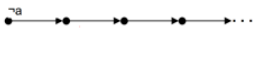
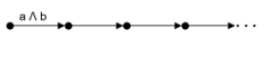
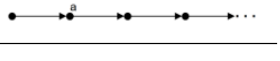
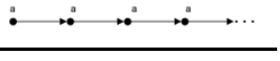
2.1.1.1 Lógica Temporal de Tempo Linear - LTL

A lógica temporal de tempo linear, como o próprio nome já indica, analisa o sistema de forma sequencial e linear. Assim, uma fórmula temporal de tempo linear é uma afirmação sobre uma sequência particular de estados, para raciocinar sobre sistemas não determinísticos. Tendo essa definição como norte podemos apresentar a sintaxe da LTL a qual contém variáveis booleanas proposicionais **V**, operadores temporais e os operadores propositivos usuais, incluindo negação $\neg$ e conjunção $\wedge$. Operadores temporais típicos são os operadores: "próxima vez" **X**, "finalmente" **F**, "até que" **U**, "para ir até" **W** e o operador "globalmente" **G**.

Sabendo da sintaxe da lógica temporal de tempo linear, podemos aprofundar nossa definição apresentando uma básica explanação da semântica de uma fórmula LTL. Esta é definida em relação a uma estrutura de tempo linear $M = (S, x, L)$, onde S é um conjunto de estados, $x = x_0, x_1, ...$ é uma sequência infinita de estados, e $L : S \rightarrow 2^\rho$ é uma rotulagem de estados com proposições atômicas em ρ Emerson (1990).

Assim uma fórmula LTL pode ser satisfeita por uma sequencia infinita de avaliações das variáveis de proposições atômicas (*Atomic Proposition* - AP). Essas sequencias podem ser vistas como uma palavra em um caminho de estrutural de Kripke Kripke (1963). Sabendo que uma estrutura de Kripke é uma variação do sistema de transição, originalmente proposto por Saul Kripke, usado na verificação de modelos para representar o comportamento de um sistema. É basicamente um gráfico nos quais seus nós representam os estados alcançáveis do sistema e suas arestas representam transições de estado. Uma função de rotulagem mapeia cada nó para um conjunto de propriedades que são mantidas no estado correspondente. Formalmente as logicas temporais são interpretadas em função da estrutura de Kripke, dessa maneira foi montada a Tabela 2 a qual apresenta os operadores lógicos e temporais da formula LTL.

Tabela 2 – Quadro explicativo dos principais operadores de LTL.

Textual	Expresão	Explicação	Diagrama
a	$\pi \models a \equiv a \in L(\pi[0])$	Uma proposição atômica é verdadeira em um caminho, se ela se mantém no primeiro estado do caminho.	
¬a	$\pi \models a \equiv a \notin L(\pi[0])$	A negação de uma proposição atômica mantém um caminho π se, e somente se, a proposição atômica não se mantiver no primeiro estado do caminho.	
a ∧ b	$\pi \models a \wedge b \equiv$ $\pi \models a$ e $\pi \models b$	Dadas duas fórmulas LTL "a" e "b", a conjunção se mantém em um caminho π se, e somente se, ambas as fórmulas se mantêm no caminho	
Xa	$\pi \models \mathbf{X}a \equiv \pi[1...] \models a$	Dada a proposição "a", a fórmula Xa se mantém em um caminho se ele se mantém no próximo estado.	
a **U** b	$\pi \models a\ \mathbf{U}\ b \equiv$ $\exists i \geq 0.\ \pi[i...] \models b$ e $\forall\ 0 \leq j < i.\ \pi[j...] \models a$	O operador Until especifica que uma fórmula é verdadeira até que outra seja verdadeira.	
Fa	$\pi \models \mathbf{F}a \equiv$ $\exists i \geq 0.\ \pi[i...] \models a$	O operador Finally especifica que a proposição "a" eventualmente tem que se manter, em algum lugar no caminho subsequente.	
Ga	$\pi \models \mathbf{G}a \equiv$ $\forall i \geq 0.\ \pi[i...] \models a$	O operador Globally tem que manter a proposição "a" por todo o caminho subsequente.	

Fonte: Feita pelo autor tomando como base (SCHMALTZ, 2017)

2.1.1.2 A lógica da árvore de computação - CTL

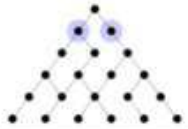
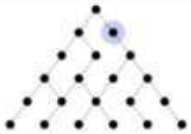
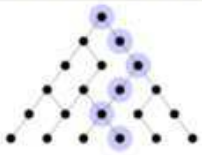
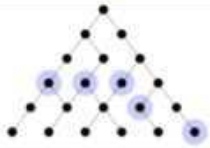
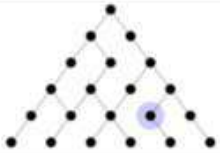
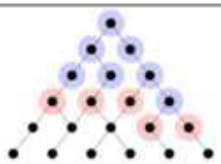
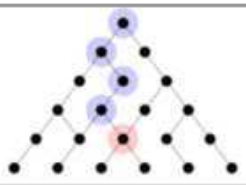
Diferindo da LTL que apresenta um caminho linear e um único futuro definido, a lógica da arvore de computação (*Computation Tree Logic* - CTL) é uma lógica proposicional de tempo de ramificação, isto é, utiliza da lógica proposicional como base e usa um modelo discreto de tempo em que, a cada instante, o tempo pode se dividir em mais de um futuro possível Emerson, Sistla e Clarke (1986). Esta é usada na verificação formal de sistemas de software ou hardware, geralmente utilizado com a metodologia da verificação de modelo, apresentada na Seção 2.2.1. Por exemplo, a propriedade de segurança pode ser verificada por um verificador de modelo que explora todas as transições possíveis fora dos estados do programa que satisfazem a condição inicial e garante que todas essas execuções satisfaçam a propriedade.

Primeiro introduzimos a sintaxe do CTL, que assim como a LTL apresenta variáveis booleanas como Verdadeiro (*True*) e falso (*False*), bem como os demais operadores lógicos: negação ¬, conjunção ∧, disjunção ∨, implicação material ⇒, implicação inversa ⇐ e a bicondicional ⇔. Também temos operadores temporais, sendo estes divididos em quantificadores sobre caminho: "ao longo de todos os caminhos" **A** e ao menos existe um caminho" **E** e os quantificadores específicos de caminho: "próxima vez" **X**, "finalmente" **F**, "até que" **U**, "tem que manter até" **W** e o operador "globalmente" **G**.

Por sua vez, agora, que temos o conhecimento da sintaxe da lógica da arvore de computação é possível fazer uma explicação a respeito da sua semântica. Sendo assim para uma fórmula CTL a semântica é definida com relação a uma estrutura temporal M = (S, R, L), onde S é um conjunto de estados, R ⊆ S × S é uma relação binária total (i.e., $\forall_{s \in S} \exists_{t \in S}$(s,t) ∈ R), e L: S → 2^{ρ} é uma marcação de estados com as proposições atômicas em ρ que são verdadeiras em um dado estado. R é a relação de próximo estado da estrutura, ou seja, se o sistema estiver no estado s em um determinado instante de tempo, ele estará em qualquer um dos estados no conjunto {t ∈ S|(s,t) ∈ R} no instante seguinte. O requisito de totalidade para

R está incluído porque as fórmulas de CTL não têm interpretação sensata para estados sem sucessores. Um caminho é definido como uma sequência infinita de estados $s_0, s_1, ...$ tal que $\forall_{i \geq 0}(s_t, s_{i+1}) \in$ R. Podemos ver na Tabela 3 uma explicação acerta de algumas aplicações do CTL.

Tabela 3 – Quadro explicativo dos principais operadores de CTL.

Textual	Expresão	Explicação	Diagrama
AXa	s |= **AX**a, se $\forall \pi$ **s.t.** $s_0 = s.\ S_1 \models a$	Mantem o valor para todos os proximos estados de da proporsição "a".	
EXa	s |= **EX**a, se $\exists \pi$**s.t.** $s_0 = s.\ s_1 \models a$	Existe um proximo estado no qual a proporsição "a" mantem o valor.	
AGa	s |= **AG**a, se $\forall \pi$**s.t.** $s_0 = s.\ \forall i.s_i \models a$	Para todos os caminhos, para todos os estados ao longo deles, proporsição "a" mantém.	
EGa	s |= **EG**a, se $\exists \pi$**s.t.** $s_0 = s.\ \forall i.s_i \models a$	Existe um caminho tal que, para todos os estados ao longo dele, contenham a proporsição "a".	
AFa	s |= **AF**a, se $\forall \pi$**s.t.** $s_0 = s.\ \exists i.s_i \models a$	Para todos os caminhos, existe um estado futuro em que a proposição "a" é válida.	
EFa	s |= **EF**a, se $\exists \pi$**s.t.** $s_0 = s.\ \exists i.s_i \models a$	Aqui existe um caminho com um estado futuro onde a proposição "a" é válida.	
A[a **U** b]	s |= **A**[a **U** b], se $\forall \pi$**s.t.** $s_0 = s.\forall i.s_i \models b$ e $\forall j < i.\ S_j \models a$	Para todos os caminhos, a proposição "b" eventualmente se mantém, e a proposição "a" vale para todos os estados anteriormente.	
E[a **U** b]	s |= **E**[a **U** b], se $\exists \pi$**s.t.** $s_0 = s.\ \exists i.s_i \models b$ e $\forall j < i.\ S_j \models a$	Existe um caminho onde a proposição "b" eventualmente se mantém, e "a" se mantém estados anteriores.	

Fonte: Feita pelo autor tomando como base (FLEURIOT, 2015)

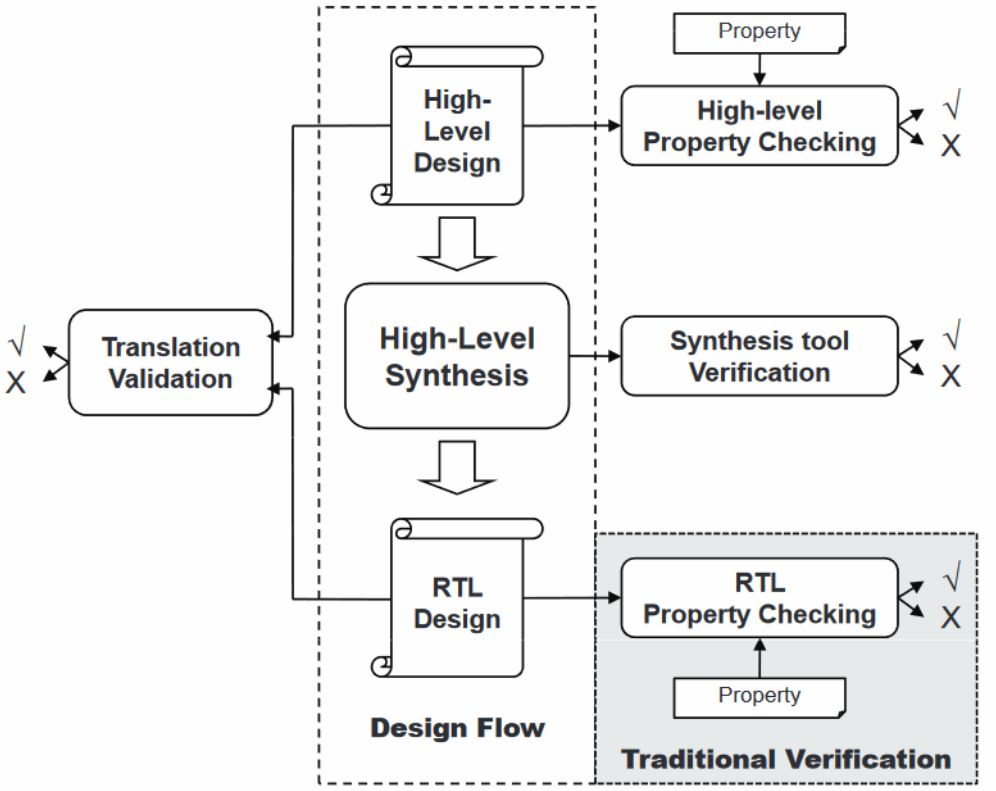
2.1.2 Especificação com modelos de alto nível

Verificação de modelos de alto nível (*High-Level Verification* - HLV) também conhecida como verificação de nível de sistema eletrônico (*Electronic System-Level* - ESL) é um estilo alternativo de especificação que usado para estipular o comportamento permitido de um sistema. Nessa estrutura, a verificação implica o raciocínio sobre o relacionamento entre o modele de alto nível (M_s), conhecido como especificação, e um modelo de nível inferior (M_i), chamado de implementação.

O projeto de hardware digital eletrônico evolui de abstração de baixo nível, nível do gate (*Gate-Level*), para nível de registro de transferência (RTL) e o nível de abstração acima da RTL é comumente chamado de nível alto, ESL ou nível comportamental. O foco principal dessa abordagem é a noção de abstração, que permite que detalhes desnecessários sejam ocultados em alto nível, geradas através de uma série de abstrações da descrição detalhada da implementação.

O processo HLV consiste em realizar transformações passo a passo de uma especificação comportamental em uma implementação estrutural (RTL). O principal benefício do HLS é que ele fornece um tempo mais rápido para a RTL e um tempo de verificação mais rápido. A Figura 1 mostra os vários componentes envolvidos na verificação de alto nível e como eles interagem. O fluxo de design da especificação de alto nível para a RTL é mostrado junto com várias tarefas de verificação. Essas tarefas podem ser classificadas da seguinte forma: Verificação de propriedades de alto nível (*High-level property checking*), Validação de tradução (*Translation validation*), Verificação da ferramenta de síntese (*Synthesis tool verification*) e Verificação da propriedade RTL (*RTL property checking*)

Figura 1 – Visão Geral da Verificação de Alto Nível.



Fonte: (KUNDU, 2009)

Tradicionalmente, os projetistas iniciam seus esforços de verificação diretamente para projetos de RTL. No entanto, com a popularidade do HLS, esses esforços estão se movendo mais em direção às suas contrapartes de alto nível. Isso é particularmente interessante porque permite um tempo de verificação funcional e formal mais rápido, quando comparado a uma implementação de RTL de baixo nível mais detalhada. Além disso, permite uma exploração do espaço de design mais elaborada, o que, por sua vez, leva a uma melhor qualidade de design.

Nesta seção, primeiro revisamos o papel da abstração e, em seguida, exploramos técnicas para cada uma das três áreas descritas acima, ou seja, verificação de propriedades de alto nível, validação de tradução e verificação de ferramentas de síntese. Por fim, analisamos várias formalizações sobre o que significa que "o M_i implementa o M_s".

2.1.2.1 Mecanismos de Abstração

Para exemplificar os mecanismos de abstrações sera usado como base a definição dado por Melham (1990), o qual identifica quatro tipos diferentes de abstração que prevalecem na verificação de hardware. Para facilitar a compreensão sera ilustrado cada categoria usando um simples contador de 3 bits, como exemplo.

Primeiro a abstração estrutural (*Structural abstraction*) que suprime detalhes sobre a estrutura interna da implementação na especificação. A especificação fornece uma visão de "caixa preta" do design que reflete o comportamento externamente observável do sistema sem restringir seu design interno. Por exemplo, o contador pode ser descrito como um componente com duas entradas uma para o *clock* e outra para o controle de direção, e uma saída para o valor da contagem, um refinamento estrutural desta descrição poderia descrever o contador usando flip-flops e *gates*.

Após a abstração estrutural podemos definir um outro modelo a abstração comportamental (*Behavioural abstraction*) esta suprime detalhes sobre o que o componente faz sob condições operacionais que nunca devem ocorrer. Por exemplo, uma descrição comportamental do contador de três bits pode omitir uma especificação do que o contador faz se for incrementado além da sua contagem máxima ou decrementado além da sua contagem mínima. A abstração comportamental também pode indicar condições "não se importam". Por exemplo, se vários dispositivos estiverem solicitando acesso a um barramento e todos eles tiverem a mesma prioridade, o controlador de barramento poderá conceder as solicitações em uma ordem arbitrária. Isso dá ao projetista maior flexibilidade para otimizar a implementação, já que a especificação é mais abstrata, no sentido de que seus comportamentos são um superconjunto dos comportamentos de implementação.

Seguindo com os modelos de abstração temos a abstração de dados (*Data abstraction*) a qual relaciona os sinais na implementação aos sinais na especificação quando eles têm diferentes representações. Por exemplo, a especificação do contador pode especificar que a saída é um número inteiro entre zero e sete, e a implementação pode ter uma saída que consiste em três sinais com valor booleano. A abstração de dados requer um mapeamento que determina como os estados ou sinais da implementação devem ser interpretados no domínio semântico da especificação.

Por fim podemos citar a abstração temporal (*Temporal abstraction*) a qual relaciona as etapas de tempo da implementação às etapas de tempo da especificação. Por exemplo, o contador pode ser especificado em termos do que acontece para cada ciclo de *clock* enquanto a implementação usa um *clock* de duas fases. Frequentemente, as especificações de um microprocessador usam a execução de uma única instrução como a unidade básica de tempo. Uma implementação, no entanto, basearia sua noção de tempo na execução de uma instrução de microcódigo, um ciclo de *clock* ou uma fase de relógio. A abstração temporal requer que

os estados de execução correspondentes nas duas escalas de tempo sejam identificados. Isso é complicado pela possibilidade de que uma unidade de tempo de especificação nem sempre corresponda necessariamente ao mesmo número de etapas de implementação.

2.1.2.2 Explicação da Lógica

A primeira categoria de métodos a ser analisada é a verificação de propriedades de alto nível, a qual permite que as propriedades importantes que os componentes de alto nível precisam satisfazer tenham sido verificadas, várias outras técnicas são usadas para provar que a tradução do projeto de alto nível para RTL de baixo nível está correta, garantindo também que as propriedades importantes de os componentes são preservados.

Especificando o design de alto nível, usamos técnicas de verificação de modelo para verificar se o design satisfaz uma determinada propriedade, como a ausência de *deadlocks* ou violações de asserção. A verificação do modelo em sua forma pura sofre do bem conhecido problema de explosão do estado como sera explanado na seção 2.2.1. Para lidar com esse problema, alguns sistemas desistem da conclusão da pesquisa e concentram-se nos recursos de descoberta de bugs da verificação de modelo. Essa linha de pensamento leva à abordagem de verificação de modelo baseada em execução, que, para uma dada entrada e profundidade de teste, explora sistematicamente todos os possíveis comportamentos do projeto, devido à simultaneidade assíncrona.

O benefício mais impressionante da abordagem de verificação de modelo baseada em execução é que ela pode analisar linguagens de programação ricas em recursos, como C++, pois evita a necessidade de representar formalmente a semântica da linguagem de programação como uma relação de transição. Outro aspecto importante dessa abordagem é a ideia de busca sem estado, o que significa que ela não armazena representações de estado na memória, mas apenas informações sobre quais transições foram executadas até o momento. Embora a pesquisa sem estado reduza os requisitos de armazenamento, um desafio significativo para essa abordagem é como lidar com o número exponencial de caminhos no programa. Para resolver isso, é possível usar técnicas dinâmicas de redução de ordem parcial (*partial-order-reduction* - POR) para evitar a geração de dois caminhos que tenham o mesmo efeito no comportamento do design. Intuitivamente, as técnicas de POR exploram a independência entre *threads* paralelas para pesquisar um conjunto reduzido de caminhos e ainda são comprovadamente suficientes para detectar *deadlocks* e violações de asserções Kundu (2009).

A segunda categoria é validação de tradução a qual inclui técnicas que tentam mostrar, para cada conversão que a ferramenta HLV executada, que o programa de saída produzido pela ferramenta tem o mesmo comportamento que o programa original. Embora essa abordagem não garanta que a ferramenta HLV esteja livre de erros, ela garante que quaisquer erros na tradução serão detectados quando a ferramenta for executada, evitando que esses erros se propaguem mais no processo de fabricação de hardware.

A verificação de ferramenta de síntese de terceira categoria consiste em técnicas cujo objetivo é provar automaticamente que uma determinada ferramenta HLS de otimização está correta. Embora, essas técnicas tenham o mesmo objetivo da validação de tradução, ou seja, garantir que uma determinada ferramenta HLS produza resultados corretos, essas técnicas são diferentes porque podem comprovar a correção de partes da ferramenta HLS de uma vez por todas, antes de serem executadas. Ao contrário da validação de tradução, essa abordagem prova a correção de uma ferramenta HLS de uma vez por todas, antes que ela seja executada.

2.1.2.3 Refinamento

A noção de que M_I implementa M_S é formalizada em termos de uma relação de refinamento $\subseteq$. Nesta seção, descrevemos várias definições dessa relação, que têm sido usadas na literatura. É geralmente aceito que $M_I \subseteq M_S$ deve implicar que cada comportamento observável de M_I é também um comportamento observável de M_S GERTH (1989). Geralmente, a equivalência de comportamentos observáveis é entendida com relação a um mapeamento de abstração apropriado que relaciona aspectos correspondentes de M_I e M_S.

Conforme a descrição de ABADI e LAMPORT (1991), uma noção muito geral de refinamento é baseada na inclusão de traços. Cada estado tem dois componentes: uma parte "externa",isto é, visível, e uma parte "interna", isto é, oculta. Seja E denotar o conjunto de estados visíveis externamente da implementação e da especificação. A abstração de dados entre os dois níveis é tratada pelo mapeamento de estados de implementação externos em seus estados de especificação correspondentes. Se I_I e I_s denotar os estados internos de M_I e M_S e $S_I = E \times I_I$ e $S_S = E \times I_S$ ser os espaços de estado dos dois modelos. Sabendo que X_I e X_S denotar os conjuntos de todas as sequências de estados permitidos por M_I e M_S são definidos como sendo um refinamento de M_S se assim $\subseteq$ se para cada comportamento da implementação. Existe um comportamento de especificação com os mesmos estados visíveis externamente, a abstração temporal é acomodada permitindo que para cada $\sigma \in X_S$, todas as sequências obtidas substituindo estados por repetições finitas, ou removendo repetições, também estão em X_S.

Outra definição é dada por MARETTI (1994)a qual introduz uma generalização da inclusão de traços onde o estado visível externamente do sistema está sujeito a um protocolo de interface. Considere, por exemplo, um dispositivo que envia dados em um barramento e indica a disponibilidade de dados válidos no barramento, definindo um sinal válido como verdadeiro. Um protocolo de interface pode ser definido especificando que o barramento deve ser observado apenas, isto é, visível externamente se *valid* = verdadeiro. Uma implementação do dispositivo que coloca valores arbitrários no barramento enquanto *valid* = falso ainda pode ser considerada um refinamento da especificação com relação ao protocolo de interface, mesmo que os valores no barramento não sejam consistentes com a especificação se o protocolo de interface for desconsiderado.

2.2 Técnicas de Verificação

Tendo introduzido vários conceitos importantes sobre lógicas aplicadas dentro das metodologias formais, os quais tem a função de descrever o sistemas e afirmar sua conformidade com uma especificação, agora focamos nossa atenção em métodos para verificar tais asserções.

Um aspecto importante é o grau de automação oferecido por uma técnica de verificação; a ferramenta de verificação "ideal", dada uma descrição do sistema e uma especificação, decidirá, dentro de um período de tempo "aceitavelmente curto", se a especificação é atendida ou não. Infelizmente, a criação de tal ferramenta é muitas vezes irreal na prática, se não impossível. Pode-se, de fato, argumentar que não é razoável esperar sempre a verificação totalmente automática de projetos que resultam de um processo intelectual envolvendo engenheiros altamente qualificados.

Atualmente, as técnicas disponíveis variam de métodos completamente automáticos para verificar especificações de lógica temporal como a verificação de modelo e demonstração de teoremas interativos usando cálculos lógicos usamos o método de verificação dedutiva. Finalmente, a seção apresentamos algumas abordagens que integram as abordagens anteriores.

2.2.1 Verificação de Modelo

A verificação de modelo (*Model Checking*), é uma técnica que verifica as propriedades em relação a um modelo para provar que o projeto está de acordo com as especificações. De forma mais clara, a verificação é executada como uma pesquisa de espaço de estado exaustiva que é garantida para terminar, uma vez que o modelo é finito. Com isso a principal vantagem dessa técnica é que ela relata caminhos ilegais, conhecidos como contraexemplos, que suportam a correção de bugs. Um problema inerente a essa técnica, no entanto, é a explosão do espaço de estados, uma vez que o número de estados usados para modelar o sistema cresce exponencialmente com o número de variáveis McMillan (1992).

Usando a exploração de estado, torna o *Model Checking* uma ferramenta poderosa de verificação formal, sendo amplamente usada em hardware. Todavia, seu método explicito de enumeração de estado é altamente consumidor de recursos. Portanto, para sistemas com milhares de estados, é necessário limitar o número de estados possíveis em que o verificador de modelo pode trabalhar ou pode proibir sua criação. O crescimento exponencial no espaço de estados tem um efeito colateral conhecido como explosão do espaço de estados, diretamente conectado ao número de variáveis do sistema e sinais de entrada e saída Burch et al. (1992).

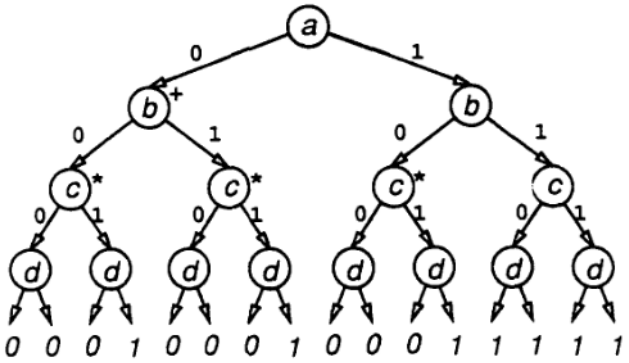
A verificação de modelos como uma técnica de verificação possui três características fundamentais. Primeiro, é automático; Não depende de interação complicada com o usuário para comprovação de propriedade incremental. Se uma propriedade não for válida, o verificador de modelo gera um rastreio de contraexemplo automaticamente. Segundo, os sistemas que estão sendo verificados são considerados finitos. Exemplos típicos de sistemas finitos, para os quais a verificação de modelo foi aplicada com sucesso, são circuitos sequenciais digitais e protocolos de comunicação. Finalmente, a lógica temporal é usada para especificar as propriedades do sistema. Assim, a verificação de modelos pode ser resumida como uma técnica algorítmica para verificação de propriedades temporais de sistemas finitos.

2.2.1.1 Verificação de Modelo Simbólico - SMC

Dessa maneira é necessário utilizar formas de se evitar a explosão de estados, sendo uma delas a verificação de modelo simbólico (*Symbolic Model Checking* - SMC). No lugar de enumerar explicitamente todos os estados, o SMC trabalha com conjuntos de símbolos, utilizando do diagrama de decisão binaria (*Binary decision diagrams* - BDDs) para representar conjuntos de estados e trabalhar com eles em operações de sistemas mais complexos.

Os BDDs representam funções booleanas de forma canônica onde um caminho pode ser rastreado desde o nó raiz até qualquer um dos nós folha, assim, estes encontraram aplicação em muitas tarefas de projeto auxiliadas por computador, incluindo verificação simbólica de lógica combinacional. Um BDD é semelhante a uma árvore de decisão binária, exceto que sua estrutura é um gráfico acíclico direcionado em vez de uma árvore, e há uma ordem total colocada na ocorrência de variáveis quando se percorre todo o gráfico. Considere, por exemplo, a Figura 2. Ela representa a fórmula $(a * b) + (c * d)$, usando a variável ordenando $a < b < c < d$. Dada uma atribuição de valores booleanos às variáveis a, b, c e d, pode-se decidir se a atribuição satisfaz a fórmula percorrendo o gráfico começando na raiz, ramificando em cada nó com base no valor atribuído da variável que rotula esse nó. Por exemplo, a atribuição $< a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1 >$ leva a um nó folha rotulado 1, portanto, essa atribuição satisfaz a fórmula.

Figura 2 – Árvore de Decisão Ordenada.



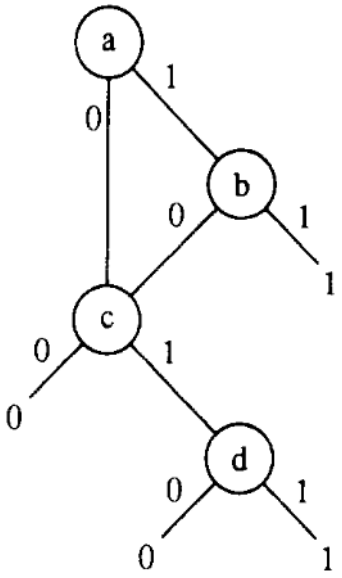
Fonte: (MCMILLAN, 1992)

Dessa maneira é perceptível que os BDDs são altamente dependentes da ordenação de variáveis, o que afeta diretamente seu tamanho. Portanto, é necessário aplicar boas estratégias para otimizá-las corretamente. Os BDDs Ordenados Reduzidos (*Reduced Ordered BDDs* - ROBDDs) são um passo adiante, com a remoção de todos os caminhos redundantes, compartilhando todos os nós possíveis e usando heurísticas de ordenação de variáveis para que tenha um tamanho mínimo.

Os ROBDDs são uma forma de decisão reduzida que fornece uma representação canônica compacta para fórmulas booleanas. A representação canônica do ROBDD para uma função booleana pode ser derivada pela redução de uma estrutura relacionada chamada de árvore de decisão ordenada. Em uma árvore de decisão ordenada, o valor da função é obtido descendo a árvore da raiz para uma folha. Em casa nó ao longo do caminho, um desce para o filho esquerdo se o valor da variável rotular o nó for 0 e, para o filho direito, o valor for 1. Cada folha da árvore é rotulada com um valor 0 ou 1, o que dá o resultado de a função. Diz-se que a árvore é ordenada se as variáveis sempre ocorrerem na mesma ordem ao longo de qualquer caminho, da raiz à folha. Neste caso, lendo as folhas da esquerda para a direita, obtém-se a tabela verdade da função.

Utilizando o exemplo, como ilustração de redução para forma canônica com a árvore de decisão ordenada para a função $(a * b) + (c * d)$ descrita na Figura 2. Os três nós marcados com " $*$ " são raízes de subárvores isomórficas. Dessa maneira. eles podem ser combinados em uma única subárvore, além do que, a partir do nó marcado " $+$ ", chega-se à mesma subárvore ao descer para a esquerda ou para a direita, ou seja, independentemente do valor de b. Portanto, este vértice não afeta o valor da função e pode ser eliminado, resultando na árvore da Figura 3 que é uma representação ROBDDs. Observe a redução significativa no número de vértices, resultante essencialmente do reenvio na tabela verdade da função.

Figura 3 – Diagrama de Decisão Binaria Ordenados e Reduzido.



Fonte: (BURCH et al., 1992)

As falhas encontradas com o modelo de verificação simbólico, quando falha ao verificar a entrada é um contraexemplo, que traça caminhos desde o estado inicial até o estado da falha da proposição. Com esse rastreamento, um desenvolvedor pode recriar a falha e descobrir o que a causou. Uma característica importante do contraexemplo é seu comprimento mínimo, em contraste com os traçados de simulação. Enquanto uma execução de simulação pode precisar de centenas de milhares de ciclos para atingir um estado de falha, um mecanismo formal pode atingir esse mesmo estado com apenas dezenas de ciclos.

2.2.1.2 Verificação de Modelo Limitado - BMC

Outra técnica que tenta superar o problema causado pela explosão do espaço de estados do SMC é a verificação de modelo limitado (Bounded Model Checking - BMC) Biere et al. (1999). O BMC usa procedimentos de decisão proposicional ou verificação de satisfação (Satisfiability Checking - SAT) para modelar o sistema, que, como BDDs, também são baseados em expressões booleanas, mas não tentam construir uma estrutura de dados canônica. Por causa disso, o SAT pode lidar com procedimentos de satisfazibilidade proposicional com milhares de variáveis.

A motivação original da verificação de modelos limitados foi alavancar o sucesso do SAT na solução de fórmulas booleanas para modelar a verificação. Dessa forma possibilitando aos verificadores de modelos simbólicos com BDDs, formas de verificar sistemas com mais do que algumas centenas de latches através do uso do modelo SAT.

Na prática, o BMC é usado principalmente para verificar testes de "falsificação", os quais são preocupado com violações de propriedades temporais Biere et al. (2009). Duas outras aplicações relacionadas, nas quais o BMC se torna cada vez mais importante, são, a geração automática de casos de teste para o fechamento de furos de cobertura e a redundância dos projetos.

A ideia básica do BMC é representar um contraexemplo de comprimento simbólico e verificar a fórmula proposicional resultante com uma solução do SAT. Se a fórmula for

satisfatória o caminho é viável, o resultado retornado pelo solucionador SAT pode ser traduzido em um contraexemplo concreto, apresentando caminhos que mostram qual a propriedade que está violada, caso contrário, o limite é aumentado e o processo repetido.

A primeira década de pesquisa em verificação de modelos testemunhou um debate acalorado para decidir qual o formalismo de especificação é mais apropriado: lógica do tempo linear (LTL) ou lógica de árvore de cálculo , separando dessa forma os modelos de verificação do SMC e BMC. Sendo o BMC baseado na LTL, podemos definir a lógica para esse modelo da seguinte forma, as fórmulas LTL são definidas em todos os caminhos, dessa maneira encontrar contraexemplos corresponde à pergunta se existe um caminho que os contradiga. Se encontrarmos tal vestígio, chamamos isso de testemunha da propriedade.

A ideia básica da verificação de modelo limitado, como foi explicado antes, é considerar apenas um prefixo finito de um caminho que pode ser uma testemunha de um problema de verificação de modelo existencial, dessa maneira restringimos o comprimento do prefixo por algum "k" limitado. Na prática, aumentamos progressivamente o limite, procurando testemunhas em caminhos mais longos. Uma observação crucial é que, embora o prefixo de um caminho seja finito, ele ainda pode representar um caminho infinito se houver um loop de retorno do último estado do prefixo para qualquer um dos estados anteriores, como na Figura 4 (b). Se não existe tal loop de retorno, como na Figura 4 (a), então o prefixo não diz nada sobre o comportamento infinito do caminho além do estado s_k. Por exemplo, apenas um prefixo com um loop de retorno pode representar uma testemunha para o G_p. Mesmo se p mantiver ao longo de todos os estados de s_0 para s_k, mas não há loop de volta de s_k para um estado anterior, não podemos concluir que encontramos uma testemunha para G_p, pois p pode não ser mantido em s_{k+1} Biere et al. (2003).

Figura 4 – Os dois casos para um caminho limitado, (a) sem loop, (b) com loop.

Fonte: (BIERE et al., 2003)

2.2.2 Métodos dedutivos

Como apresentado anteriormente outra metodologia é o método dedutivo, o qual se baseia nas especificações do sistemas para gerar um conjunto de obrigações e provas matemáticas, cuja veracidade implica na conformidade do sistema com o que é previsto. Para realizar isso temos o auxilio de um conjunto de provadores de teoremas como HOL, ACL2, Isabelle e PVS. Dessa maneira a verificação usando método dedutivo verifica se uma implementação atende à sua especificação em tal estrutura sendo equivalente a provar um teorema na lógica subjacente. Em princípio, esta prova pode ser realizada manualmente, no entanto, as provas de tais teoremas são frequentemente longas e bastante exaustivas na prática, tornando provável que contenham erros.

O uso de um sistema de prova de teoremas mecanizado pode garantir a solidez e reduzir os equívocos, automatizando partes da prova. A discussão até agora se refere a provas manuais rigorosas, tanto quanto a sistemas automatizados. Isso significa que não há procedimento automático (ou algoritmo) que, dada uma fórmula, sempre possa determinar se existe uma derivação da fórmula na lógica. Assim, o uso bem-sucedido do teorema que prova a derivação de teoremas não triviais tipicamente envolve interação com um usuário treinado. No entanto, os provadores de teoremas ainda são ferramentas valiosas para a verificação formal. Algumas das coisas que um provador de teoremas pode fazer de acordo com Ray (2010) são.

Primeiro Um provador de teoremas pode verificar mecanicamente uma prova, isto é, verificar se uma sequência de fórmulas corresponde a uma derivação legal no sistema de prova. Isso geralmente é um assunto fácil, o teorema do provador precisa apenas verificar se cada derivação na sequência é um axioma ou se segue dos anteriores por uma aplicação legal das regras de inferência.

Como segundo aspecto provador de teorema pode ajudar o usuário na construção de uma prova. A maioria dos provadores de teoremas na prática implementar várias heurísticas para pesquisa de prova. Tais heurísticas incluem a generalização da fórmula para aplicação de indução matemática, usando instanciação apropriada de teoremas previamente comprovados, aplicação criteriosa de reescrita de termos e assim por diante.

Por fim se a fórmula a ser provada como um teorema é expressável em algum fragmento decidível bem identificado da lógica, então o teorema pode invocar um procedimento de decisão para o fragmento determinar se é um teorema. A maioria dos provadores de teoremas integra procedimentos de decisão para vários fragmentos lógicos. Por exemplo, o PVS tem procedimentos para decidir fórmulas na "aritmética de Presburger"e fórmulas sobre listas finitas, e o ACL2 tem procedimentos para decidir as desigualdades lineares sobre os racionais.

Os sistemas de provas mecanizadas são importantes na prática, uma vez que eles podem lidar com muitos casos "desinteressantes" automaticamente e, assim, ajudar na realização das provas. Agora iremos apresentar alguns dos provadores de teoremas que são usados na verificação formal.

2.2.2.1 Sistemas de Prova de Teorema

Nessa seção como já comentado sera apresentado quatro teoremas de prova usados para verificação sendo eles HOL, ACL2, Isabelle e PVS.

Começaremos com o provador lógica de ordem superior (*Higher Order Logic* - HOL) sendo descrito em detalhes no livro Gordon (1993). O HOL Theorem Prover é um programa de computador geral e amplamente utilizado para a construção de especificações e provas formais em lógica de ordem superior. O sistema é usado na indústria e na academia para suportar raciocínio formal em muitas áreas diferentes, incluindo design e verificação de hardware, raciocínio sobre segurança, provas sobre sistemas em tempo real, semântica de linguagens de descrição de hardware, verificação de compilação, correção de programas, simultaneidade de modelagem e refinamento do programa. HOL também é usado como uma plataforma aberta para pesquisa de demonstração de teoremas gerais e como uma plataforma para matemática formalizada.

O teorema interativo do HOL é um assistente de provas para lógica de ordem superior: um ambiente de programação no qual os teoremas podem ser provados e ferramentas de prova implementadas. Procedimentos internos de decisão e provadores de teoremas podem estabelecer automaticamente muitos teoremas simples (os usuários podem ter que provar os próprios teoremas difíceis!) Um mecanismo oracle dá acesso a programas externos, como motores SMT e BDD. HOL é particularmente adequado como uma plataforma para implementar

combinações de dedução, execução e verificação de propriedades.

Continuando, *A Computational Logic for Applicative Common Lisp* - (ACL2) é uma linguagem lógica e de programação na qual você pode modelar sistemas de computador, junto com uma ferramenta para ajudá-lo a provar as propriedades desses modelos. "ACL2"indica "Uma Lógica Computacional para o Lisp Comum Aplicativo".

Segundo Slobodova (2017) a lógica da ACL2 é uma lógica de primeira ordem baseada em uma linguagem de programação aplicativa, em particular, um subconjunto de aplicativos estendido do Common Lisp. O sistema ACL2, incluindo seu provador de teoremas e ambiente de desenvolvimento de sistemas, é amplamente codificado nessa mesma linguagem de programação. O ACL2 está disponível no formato de código aberto, sem custo, sob os termos de uma licença BSD de 3 cláusulas.

Outro sistema é o Isabelle o qual é uma assistente de provas genérica. Ele permite que fórmulas matemáticas sejam expressas em uma linguagem formal e fornece ferramentas para provar essas fórmulas em um cálculo lógico. Isabelle foi originalmente desenvolvido na Universidade de Cambridge e Technische Universität München, mas agora inclui inúmeras contribuições de instituições e indivíduos em todo o mundo. Sendo a instância mais difundida de Isabelle atualmente é a Isabelle/HOL, que fornece um ambiente de demonstração de teoremas lógicos de ordem superior que está pronto para uso em grandes aplicações.

Para provas, Isabelle incorpora algumas ferramentas para melhorar a produtividade do usuário. Em particular, o raciocinador clássico de Isabelle pode realizar longas cadeias de passos de raciocínio para provar fórmulas. O simplificador pode argumentar com e sobre equações. Fatos aritméticos lineares são provados automaticamente, vários procedimentos de decisão algébrica são fornecidos. Provadores externos de primeira ordem podem ser invocados através de marreta.

Por fim apresentaremos o sistema de verificação de protótipo (*Prototype Verification System* - PVS) sendo descrito por Shankar. (1992). O PVS é um sistema de verificação: isto é, uma linguagem de especificação integrada com ferramentas de suporte e um provador de teoremas. Destina-se a capturar o estado-da-arte em métodos formais mecanizados e a ser suficientemente robusto para poder ser utilizado para aplicações significativas. O PVS é um protótipo de pesquisa: evolui e melhora à medida que desenvolvemos ou aplicamos novas capacidades, e como o estresse do uso real expõe novos requisitos.

O provador de teorema PVS fornece uma coleção de procedimentos de inferência primitivos poderosos que são aplicados interativamente sob orientação do usuário dentro de uma estrutura de cálculo sequencial. As inferências primitivas incluem regras proposicionais e quantificadoras, indução, reescrita, simplificação usando procedimentos de decisão para igualdade e aritmética linear, abstração de dados e predicados e verificação de modelo simbólico. As implementações dessas inferências primitivas são otimizadas para grandes provas: por exemplo, a simplificação proposicional usa BDDs e os auto-reescritos são armazenados em cache para fins de eficiência.

2.2.3 Combinando verificação de modelo e raciocínio dedutivo

Nas seções anteriores, foi apresentado metodologias de verificação formal, desde abordagens altamente automatizadas, baseadas na exploração do estado, até os métodos baseados na melhoria da qualidade, que exigem uma quantidade considerável de esforço do verificador. Realizar a escolha entre uma essas duas técnicas acaba se tornando um processo difícil para o programador, que pode acarentar em um maior esforço e desgasto para a realização do projeto.

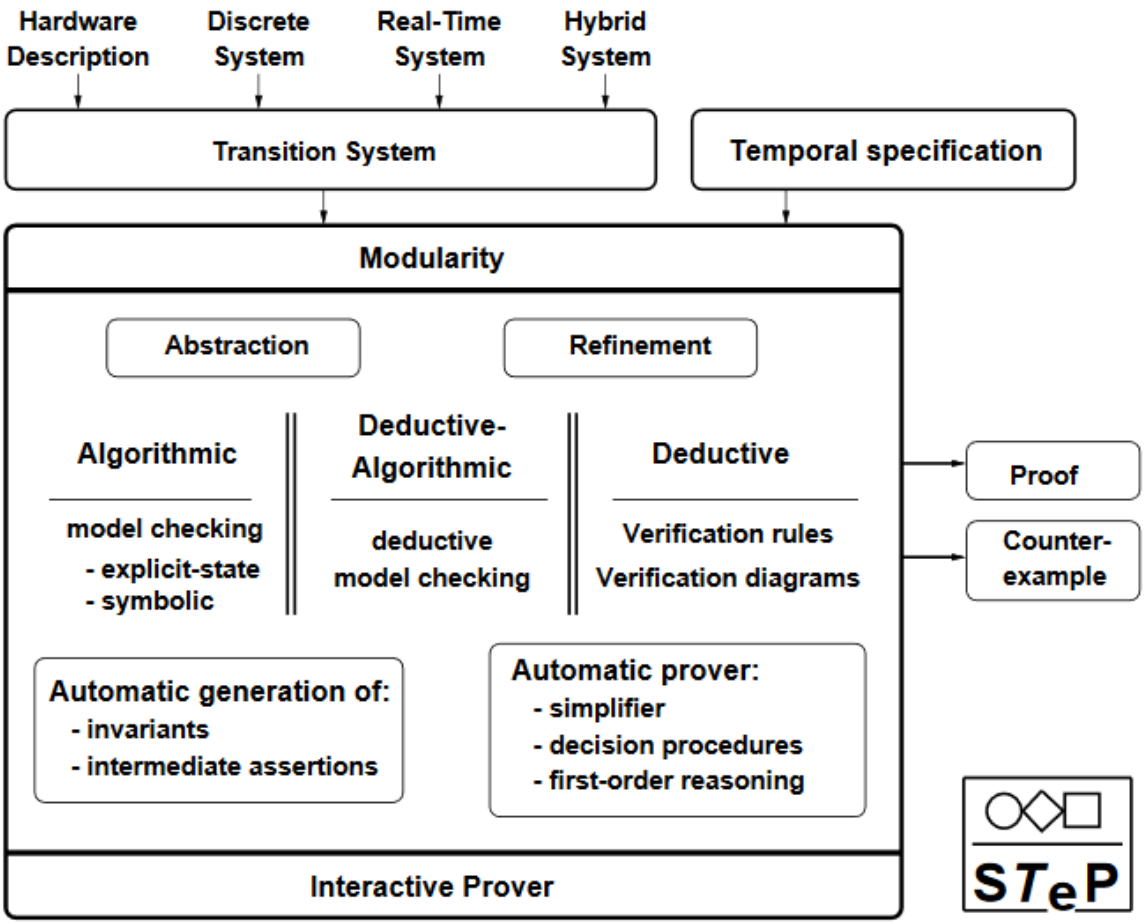
No entanto, essa duvida pode ser evitada em certos casos em que é possível aplicar

técnicas automatizadas a subsistemas ou obrigações mais simples e, em seguida, usar abordagens dedutivas para combinar os resultados assim obtidos em um resultado geral de correção. Nesta seção, apresentamos uma ferramenta baseadas nessa abordagem. Note que é um exercício não trivial adicionar um procedimento de decisão de verificação de modelo a um teorema de propósito geral já existente, pois geralmente requer formalizar a semântica da lógica temporal dentro da lógica do teorema do provador.

O Provedor Temporal de Stanford (*Stanford Temporal Prover* - STeP), é uma ferramenta para a verificação formal assistida por computador de sistemas reativos, incluindo sistemas híbridos e em tempo real, com base em suas especificações temporais. O STeP integra métodos para verificação dedutiva e algorítmica, incluindo verificação de modelo, prova de teorema, geração de invariante automática, abstração e raciocínio modular.

A Figura 5 apresenta um esboço de uma linha do sistema STeP. Os principais insumos são um sistema reativo e uma propriedade a ser comprovada para ele, expressa como uma fórmula de lógica temporal. O sistema pode ser uma descrição de hardware ou software e inclui componentes híbridos e em tempo real. A verificação é realizada por verificação do modelo ou meios dedutivos, ou uma combinação dos dois.

Figura 5 – Um esboço do sistema STeP.



Fonte: (MANNA et al., 1999)

2.3 Linguagens de Verificação Formal

A partir de todo o conteúdo abordado até o momento sobre verificação formal, chegamos ao ponto de apresentar as duas principais linguagens de programação usadas para realizar verificação formal em hardware. Portanto nesta seção iremos apresentar o conceito sobre e alguns exemplos da sintaxe dessas linguagens que são bastantes usadas por verificadores no âmbito empresarial, sendo elas *Property Specification Language* (PSL) e *SystemVerilog Assertions* (SVA).

2.3.1 Property Specification Language - PSL

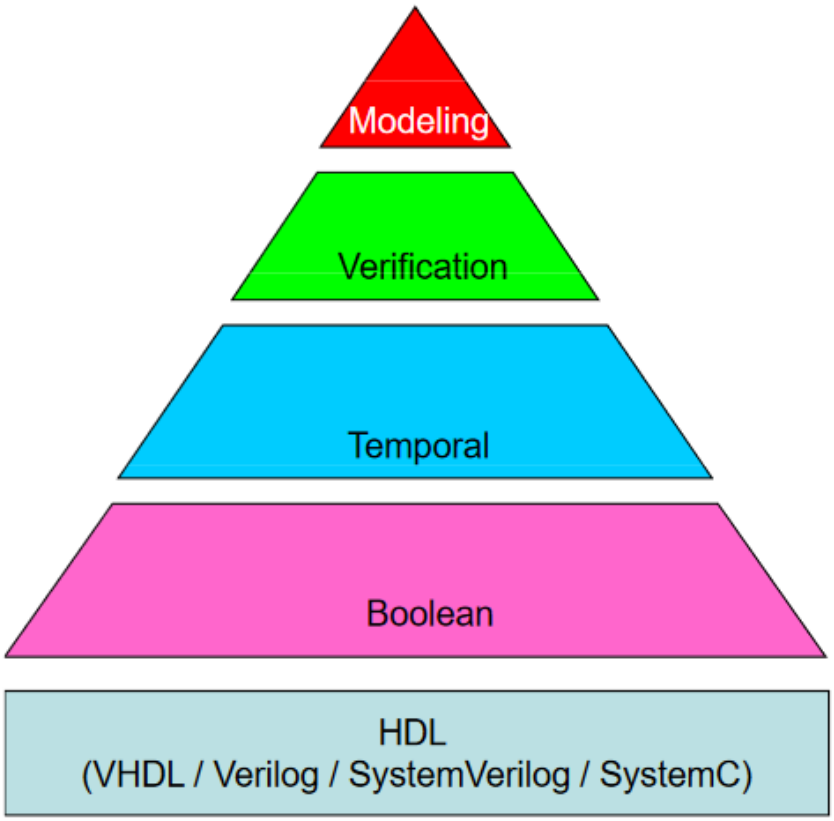
A linguagem de especificação de propriedade da Accellera (*Property Specification Language* - PSL) é uma lógica temporal que estende a lógica temporal linear com uma gama de operadores para facilitar a expressão e melhorar o poder expressivo. É amplamente utilizado na indústria de design e verificação de hardware, onde ferramentas de verificação formais e ferramentas de simulação lógica são usadas para provar ou refutar que determinada fórmula de PSL é válida em um dado design. Neste momento, a PSL trabalha em conjunto com um projeto escrito em VHDL ou Verilog/SystemVerilog e SystemC, mas no futuro a PSL pode ser estendida para trabalhar com outras linguagens. As propriedades escritas em PSL podem ser incorporadas ao código HDL como comentários ou podem ser colocadas em um arquivo separado ao lado do código HDL.

O PSL fornece ao arquiteto de design um meio padrão de especificar propriedades de design usando uma sintaxe concisa com semântica formal definida de forma clara. Da mesma forma, ele permite que o implementador de RTL capture a intenção do projeto de forma verificável, enquanto habilita o engenheiro de verificação a validar se a implementação satisfaz sua especificação através de meios de verificação dinâmicos por meio de simulações e estáticos por meio de formais. Além disso, fornece um meio de medir a qualidade do processo de verificação através da criação de modelos de cobertura funcional baseados em propriedades formalmente especificadas. Além disso, ele fornece um meio padrão para projetistas de hardware e engenheiros de verificação documentarem rigorosamente a especificação do projeto.

2.3.1.1 Estrutura da Linguagem

O PSL é definido em quatro camadas segundo Accellera (2003), sendo elas, a camada booleana (*Boolean layer*), a camada temporal (*temporal layer*), a camada de modelagem (*modeling layer*) e a camada de verificação (*verification layer*). A partir dessas camadas o PSL fornece um meio para escrever especificações que são fáceis de ler e matematicamente precisas. As quais Destinam-se a ser usadas para especificação funcional por um lado e como entrada para ferramentas de verificação funcional por outro. Assim, uma especificação de PSL é a documentação executável de um projeto de hardware.

Figura 6 – Camadas do PSL



Fonte: (GHEORGHE, 2010)

Primeiro falaremos sobre a camada booleana que é usada para descrever um estado atual do design e é expressa usando um dos HDLs acima mencionados. Essa camada é usada para criar expressões que, por sua vez, são usadas pelas outras camadas. Embora contenha expressões de muitos tipos, ela é conhecida como camada booleana porque é o fornecedor de expressões booleanas para o coração da linguagem da camada temporal. Expressões booleanas são avaliadas em um único ciclo de avaliação. Consiste em expressões booleanas, mostradas no Quadro 1, cuja sintaxe e semântica são ditadas pelo *flavor* do PSL sendo usado. A camada booleana também inclui certas expressões PSL que são do tipo booleano, onde sub-expressões de uma expressão booleana podem ser de qualquer tipo suportadas pelo HDL correspondente

Quadro 1 – Expressão booleana.

Boolean ::=
*boolean*_HDL_or_PSL_Expression

Fonte: (ACCELLERA, 2003)

Por exemplo, booleanos podem ser usados como expressões de *clock*, que indicam quando outras expressões booleanas são avaliadas. No *flavor* Verilog, qualquer expressão que o Verilog permita ser usada como condição em uma instrução *if* pode ser usada como uma expressão de *clock*. Além disso, qualquer expressão de evento Verilog permitida pela camada

de modelagem pode ser usada como uma expressão de *clock*, como vista no Quadro 2 o qual especifica uma expressão de *clock* para diretivas que possuem uma propriedade ou sequência externa que não possui uma expressão de *clock* explícita.

Quadro 2 – Declaração de *clock* padrão.

PSL_Declaration ::=
Clock_Declaration
Clock_Declaration ::=
default clock DEF_SYM Boolean;

Fonte: (ACCELLERA, 2003)

Representando em código:

Quadro 3 – Exemplo de declaração de *clock*.

Exemplo:
default clock = (posedge clk1);
assert always (req → next ack);
cover req; ack; !req; !ack;
Sendo equivalente a:
assert always (req → next ack) @(posedge clk1);
cover req; ack; !req; !ack @(posedge clk1);

Fonte: (ACCELLERA, 2003)

Dando continuidade as definições temos a camada temporal a qual consiste nos operadores temporais usados para descrever cenários que se estendem ao longo do tempo, possivelmente sobre um número ilimitado de unidades de tempo. Essa camada é o coração da linguagem, é usado para descrever as propriedades do design. É conhecida como a camada temporal porque, além de propriedades simples, como "sinais a e b são mutuamente exclusivos", também pode descrever propriedades envolvendo relações temporais complexas entre sinais, como "se o sinal c for afirmado, então o sinal d deve ser declarado antes que o sinal e seja declarado, mas não mais que oito ciclos de clock mais tarde". As expressões temporais são avaliadas em uma série de ciclos de avaliação.

A camada temporal é usada para definir propriedades, que descrevem o comportamento ao longo do tempo. As propriedades podem descrever o comportamento do design ou o comportamento do ambiente externo. No Quadro 4 temos uma declaração de sequência que pode especificar uma lista de parâmetros formais que podem ser referenciados na sequência.

Quadro 4 – Declaração de uma sequência.

```
PSL_Declaration ::=
  Sequence_Declaration
Sequence_Declaration ::=
  sequence Name [(Formal_Parameter_List)] DEF_SYM Sequence;
Formal_Parameter_List ::=
  Formal_Parameter  ; Formal_Parameter
Formal_Parameter ::=
  sequence_ParamKind Name , Name
sequence_ParamKind ::=
  const | boolean | sequence
```

Fonte: (ACCELLERA, 2003)

Um exemplo de código, apresentada no Quadro 5, para uma sequência nomeada BusArb representa uma sequência de arbitragem de barramento genérica envolvendo parâmetros formais br (solicitação de barramento) e bg (concessão de barramento), bem como um parâmetro n que especifica o atraso máximo no recebimento da concessão de barramento.

Quadro 5 – Exemplo declaração de uma sequência.

```
sequence BusArb (boolean br, bg; const n) = { br; (br && !bg)[0:n]; br && bg };
```

Fonte: (ACCELLERA, 2003)

Por sua vez a camada de verificação consiste em diretivas para uma ferramenta de verificação. Essa camada é usada para informar às ferramentas de verificação o que fazer com as propriedades descritas pela camada temporal. Por exemplo, a camada de verificação contém diretivas que informam uma ferramenta para verificar se uma propriedade é válida ou para verificar se uma sequência especificada é coberta por algum caso de teste.

A camada de verificação fornece diretivas que informam às ferramentas de verificação o que fazer com as propriedades especificadas. A camada de verificação também fornece construções que agrupam diretivas relacionadas e outras instruções PSL. Sendo essas diretivas, *assert*, *assume*, *assume_guarantee*, *restrict*, *restrict_guarantee*, *cover*, *fairness and strong fairness*. De forma resumida apresentando as declarações das diretivas da camada de verificação na Tabela 4.

Tabela 4 – Diretivas da camada de verificação.

Diretivas	Declaração	Código
Assert	Assert_Statement ::= **assert** Property**;**	assert always (ack -> next !ack);
Assume	Assume_Statement ::= **assume** Property**;**	assume always (ack -> next !ack);
Assume Guarantee	Assume_Guarantee_Statement ::= **assume_guarantee** Property **;**	assume_guarantee always (ack -> next !ack);
Restrict	Restrict_Statement ::= **restrict** Sequence**;**	restrict {!rst;rst[*3];!rst[*]};
Restrict Guarantee	Restrict_Guarantee_Statement ::= **restrict_guarantee** Sequence **;**	restrict_guarantee {!rst;rst[*3];!rst[*]};
Cover	Cover_Statement ::= **cover** Sequence **;**	cover {start_trans;!end_trans[*]; start_trans & end_trans};
Fairness and Strong Fairness	Fairness_Statement ::= **fairness** Boolean**;** | **strong fairness** Boolean, Boolean**;**	fairness p; assume GF p; strong fairness p, q; assume (GF p) -> (GF q);

Fonte: Feita pelo autor tomando como base (ACCELLERA, 2003)

Por fim temos a camada de modelagem que pode ser usada para descrever máquinas de estado auxiliares de maneira processual. Essa camada é usada para modelar o comportamento de entradas de design, para ferramentas, como ferramentas de verificação formais, que não usam casos de teste e para modelar hardware auxiliar que não faz parte do design, mas é necessário para verificação.

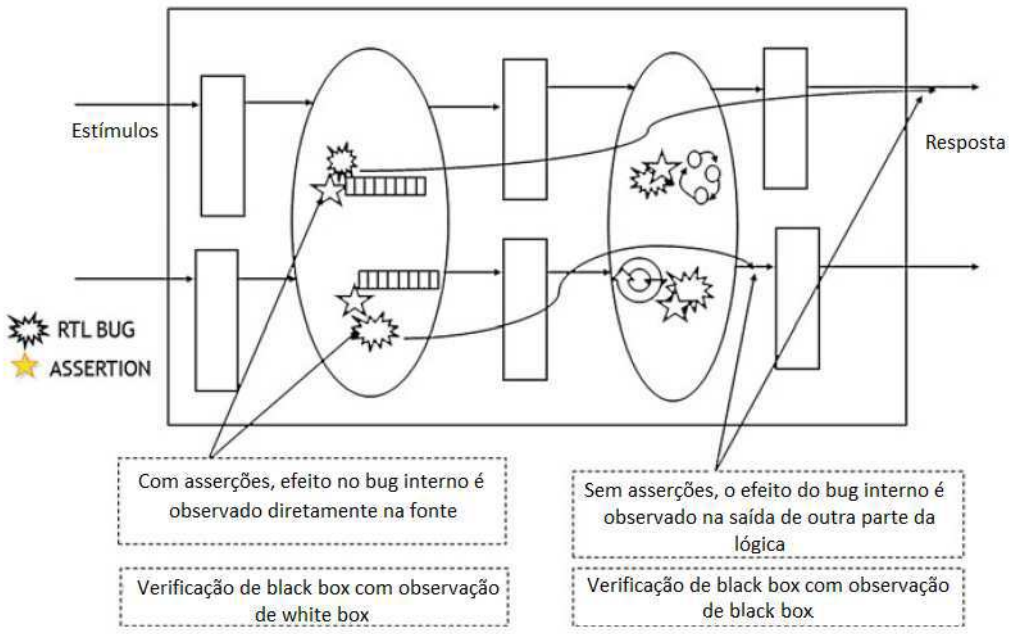
A camada de modelagem fornece um meio de modelar o comportamento de entradas de design, para ferramentas como ferramentas de verificação formal nas quais o comportamento não é especificado de outra forma e para declarar e fornecer comportamento a sinais e variáveis auxiliares.

2.3.2 SystemVerilog Assertions - SVA

As Asserções SystemVerilog *(SystemVerilog Assertions* - SVA) formam um subconjunto importante do SystemVerilog e, como tal, podem ser introduzidas nos fluxos de projeto Verilog e VHDL existentes, na Figura 7 apresenta um fluxo de compilação com assertion.

As *assertions* são usadas principalmente para validar o comportamento de um design. Levando perguntas como, "Está funcionando corretamente?", Elas também podem ser usadas para fornecer informações de cobertura funcional para um design, "Quão bom é o teste?". Você pode adicionar *assertions* ao seu código RTL à medida que você escreve "estes formulários"comentários ativos "que documentam o que você escreveu e que suposições você fez. As *assertions* também podem ser usadas como linguagem de especificação formal, tornando os requisitos claros e não ambíguos, e possibilitando automatizar a validação do projeto em relação à especificação.

Figura 7 – Exemplo Funcionamento de uma Assertions



Fonte: (MEHTA, 2014)

Uma *assertions* é uma instrução para uma ferramenta de verificação para verificar uma propriedade. As propriedades podem ser verificadas dinamicamente por simuladores, ou estaticamente por uma ferramenta de verificação de propriedades separada.

2.3.2.1 Estrutura da Linguagem

No SystemVerilog existem dois tipos de *assertions*, afirmação imediata (*assert*) e afirmação simultâneas (*assert property*). As declarações de cobertura (*cover property*) são simultâneas e têm a mesma sintaxe das *concurrent assertions*, assim como as *assume property* , que são usadas principalmente por ferramentas formais. Por fim, *expect* é uma declaração procedural que verifique se alguma atividade especificada (*property*) ocorre. Os três tipos de declaração de asserção concorrente e a declaração *expect* fazem uso de sequências que descrevem o comportamento temporal do projeto, ou seja, o comportamento ao longo do tempo, conforme definido por um ou mais *clocks*.

As afirmações imediatas (*Immediate Assertions*) são declarações processuais e são usadas principalmente na simulação. Uma afirmação é basicamente uma declaração de que algo deve ser verdadeiro, semelhante à instrução *if*. A diferença é que uma declaração *if* não afirma que uma expressão é verdadeira, ela simplesmente verifica se é verdadeira, por exemplo, temos no Quadro 6.

Quadro 6 – Exemplo comparativo de uma assert com if.

```
if (A == B) ... // Simplesmente verifica se A é igual a B
assert (A == B); // Asserts que A é igual a B; se não, um erro é gerado
```

Fonte: (DOULOS, 2014)

Se a expressão condicional da *Immediate Assertions* for avaliada como X, Z ou 0, a asserção falhará e o simulador gravará uma mensagem de erro. Desde que o verificador não tenha utilizado o comparador que leva em conta a comparação com X ou Z.

Uma afirmação imediata pode incluir uma declaração de aprovação e/ou uma declaração de falha. no Quadro 6, a instrução *PASS* é omitida, portanto, nenhuma ação é executada quando a expressão *assert* é verdadeira. Se a instrução de passagem existir, a *assert* ficara como no Quadro 7 nesse exemplo também temos a declaração associada a um *else*, que é chamada de declaração de *FAIL* e é executada se a asserção falhar.

Quadro 7 – Exemplo declaração de uma assert.

```
assert (A == B) $display("OK. A é igual a B");
else $error("Está errado");
```

Fonte: (DOULOS, 2014)

A falha de uma afirmação tem uma gravidade associada a ela. Existem três tarefas do sistema de gravidade que podem ser incluídas na declaração de falha para especificar um nível de gravidade: **$fatal**, **$error**, a gravidade padrão, e **$warning**. Além disso, a tarefa do sistema **$info** indica que a falha de declaração não possui gravidade específica.

As instruções de *PASS* e *FAIL* podem ser qualquer declaração procedural SystemVerilog legal. Eles podem ser usados, por exemplo, para escrever uma mensagem, definir um sinalizador de erro, incrementar uma contagem de erros ou sinalizar uma falha para outra parte do testbench, no Quadro 8 apresentamos um exemplo dessa lógica.

Quadro 8 – Exemplo declaração de modalidade de falhas.

```
ReadCheck: assert (data === correct_data)
else $error("Erro de leitura de memória");
Igt10: assert (I > 10)
else $warning("Eu sou menor ou igual a 10");
AeqB: assert (a === b)
else begin
  contador_error++;
  $error("A deve igual B");
end
```

Fonte: (DOULOS, 2014)

O comportamento de um design pode ser especificado usando instruções semelhantes a estas, "Os sinais de leitura e gravação nunca devem ser declarados juntos"ou "Um Pedido deve ser seguido por um Reconhecimento ocorrendo não mais do que dois *clocks* após o Pedido ser declarado". Afirmações simultâneas (*Concurrent assertions*) são usadas para verificar comportamento como este. Estas são instruções que afirmam que as propriedades especificadas devem ser verdadeiras. Por exemplo observe o Quadro **??**, que afirma que a expressão *Read* && *Write* nunca é verdadeira em nenhum momento durante a simulação.

Quadro 9 – Exemplo declaração de assert property.

```
assert property (!(Read && Write));
```

Fonte: (DOULOS, 2014)

As propriedades são construídas usando sequências. onde *Req*, presente no Quadro 10, é uma sequência simples e ## [1:2] *Ack* é uma expressão de sequência mais complexa, o que significa que *Ack* é verdadeiro no próximo relógio ou no seguinte, ou em ambos. |-> é o operador de implicação, portanto, essa asserção verifica que, sempre que *Req* é declarado, *Ack* deve ser declarado no próximo *clock* ou no seguinte *clock*.

Quadro 10 – Exemplo declaração de assert property com sequência.

```
assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

Fonte: (DOULOS, 2014)

Afirmações simultâneas como essas são verificadas durante a simulação. Eles geralmente aparecem fora de qualquer bloco inicial ou sempre em módulos, interfaces e programas. O exemplo Quadro 9 de asserção acima não contém um relógio. Portanto, ele é verificado em todos os pontos da simulação. A afirmação do Quadro 10 só é verificada quando uma borda de subido do *clock* ocorre, os valores de *Req* e *Ack* são amostrados na borda de subida do *clock*.

Nestes exemplos que estamos usando, as propriedades que estão sendo declaradas são especificadas nas próprias declarações da propriedade assert. As propriedades também podem ser declaradas separadamente, como mostrado no Quadro 11

Quadro 11 – Exemplo declaração de assert property com sequência em funções separada.

```
sequence request
  Req;
endsequence

sequence acknowledge
  ##[1:2] Ack;
endsequence

property handshake;
  @(posedge Clock) request |-> acknowledge;
endproperty

assert property (handshake);
```

Fonte: (DOULOS, 2014)

No Quadro 12 a seguir, a cláusula *disable iff* permite que uma redefinição assíncrona seja especificada. O *not* nega o resultado da sequência seguinte. Então, esta afirmação significa que se *reset* se tornar verdade a qualquer momento durante a avaliação da sequência, então a tentativa de p1 é um sucesso. Caso contrário, a sequência b ## 1 c nunca deve ser avaliada como verdadeira.

Quadro 12 – Exemplo declaração de assert property com disable iff .

```
property p1;
  @(posedge clk) disable iff (Reset) not b ##1 c;
endproperty

assert property (p1);
```

Fonte: (DOULOS, 2014)

Após toda essas definições é necessário realizar uma explicação a respeito de algumas sintaxes que ainda não foram explanadas, portanto na tabela 5 é apresentada algumas lógicas sequenciais e na Tabela 6 é a apresentada as lógicas combinacionais, as quais representam um resumo das mais relevantes sintaxes de SVA a serem analisadas.

Tabela 5 – Sintaxe dos operadores de SVA.

Operação	Sintaxe	Descrição
Implicação sobreposta	s1 |-> s2;	Para uma implicação sobreposta, se houver uma correspondência para a expressão de sequência antecedente, o primeiro elemento da expressão de sequência consequente é avaliado na mesma subida de *clock*.
Implicação não sobreposta	s1 |=> s2;	Para uma implicação não sobreposta, o primeiro elemento da consequente expressão de sequência é avaliado no próximo pulso de clock.
Operador de atraso	a ## N b	"a" deve ser verdadeiro no pulso de *clock* atual e "b" no "N" pulso do *clock*.
Operador de atraso variado	a ##[1:4] b	"a" deve ser verdadeiro no *clock* atual e "b" em algum relógio entre o primeiro e o quarto após o *clock* atual.
Operador de repetição	a ## 1 b [* 3] ## 1 c	O operador * é usado para especificar uma repetição consecutiva do operando do lado esquerdo. Equivalente: a ## 1 b ## 1 b ## 1 b ## 1 c
Operador de indeterminação	a ## 1 b [* 1: $] ## 1 c	O operador $ pode ser usado para estender uma janela de tempo para um intervalo finito, mas ilimitado. Significa: a b b b b c
Operador de sequência não consecutiva	a ## 1 b [-> 1: 3] ## 1 c	O operador [-> ou "*goto repetition*" especifica uma sequência não consecutiva. Isso significa que a é seguido por qualquer número de *clocks*, onde c é false, e b é verdadeiro entre 1 e 3 vezes, sendo a última vez que o *clock* antes de c é verdadeiro. Significa: a! b b b! b! b b c
Operador de repetição não consecutivo	a ##1 b [=1:3] ##1 c	O operador de repetição [= ou não consecutivo é semelhante a repetição goto, mas a expressão (b neste exemplo) não precisa ser verdadeira no ciclo de clock antes de c ser verdadeira. Significa: a !b b b !b !b b !b !b c

Fonte: Feita pelo autor tomando como base (DOULOS, 2014)

Tabela 6 – Sintaxe dos operadores de SVA combinacionais.

Operação	Sintaxe	Descrição
Operador and	s1 and s2	O operador binário "and" é usado quando se espera que as expressões de dois operandos sejam bem-sucedidas, mas os tempos de encerramento das expressões dos operandos podem ser diferentes.
Operador intersect	s1 intersect s2	O operador binário "intersect" é usado quando se espera que as expressões de dois operandos sejam bem-sucedidas, e os tempos de encerramento das expressões de operandos devem ser os mesmos.
Operador or	s1 or s2	O operador "or" é usado quando pelo menos uma das duas sequências de operandos deve corresponder. A sequência corresponde sempre que pelo menos um dos operandos é avaliado como verdadeiro.

Operador first match	`sequence fms;`
	`  first_match(s1 ##[1:2] s2);`
	`endsequence`
		O operador "first_match" corresponde apenas à primeira correspondência de possivelmente várias correspondências para uma tentativa de avaliação de uma expressão de sequência. Isso permite que todas as correspondências subsequentes sejam descartadas da consideração.

Fonte: Feita pelo autor tomando como base (DOULOS, 2014)

Dando continuidade na explicação da semântica do SVA apresentaremos o método para monitorar sequências e outros aspectos comportamentais de um projeto para cobertura funcional, instruções de *cover property* podem ser usadas. A sintaxe destes é a mesma que a de *assert property*. O simulador mantém uma contagem do número de vezes que a propriedade na declaração de propriedade da capa é válida ou falha. Isso pode ser usado para determinar se certos aspectos da funcionalidade do projeto foram ou não exercidos, no Quadro 13 apresentamos o código para usar o *cover property*.

Quadro 13 – Exemplo declaração de cover property.

```
module Amod2(input bit clk);
  bit X, Y;
  sequence s1
    @(posedge clk) X ##1 Y;
  endsequence
  CovLavel: cover property (s1);
...
endmodule
```

Fonte: (DOULOS, 2014)

3 Ambiente de Verificação Formal em Hardware

A partir de todo o embasamento teórico apresentado no capitulo 2, é necessário, agora, descidimos como sera realizado a ambientação de verificação formal, na qual sera realizado a comunicação entre os testes formais e o nosso hardware.

Para isso foi preferível utilizar um ambiente *open source* em UVM disponibilizado pela AMIQ (2015), sendo esse o SVAUnit. O sistema foi projetado especificamente para realizar uma verificação formal com SVA apresentando todo uma ambientação em UVM o qual otimiza o trabalho do verificador, já que o mesmo torna todo o relatório formal das ferramentas mais agradável e mais didático para o usuário. Iniciaremos esse capítulos com uma breve apresentação do ambiente SVAUnit, apresentando seu funcionamento e características básicas necessárias para sua utilização.

3.1 SVAUnit

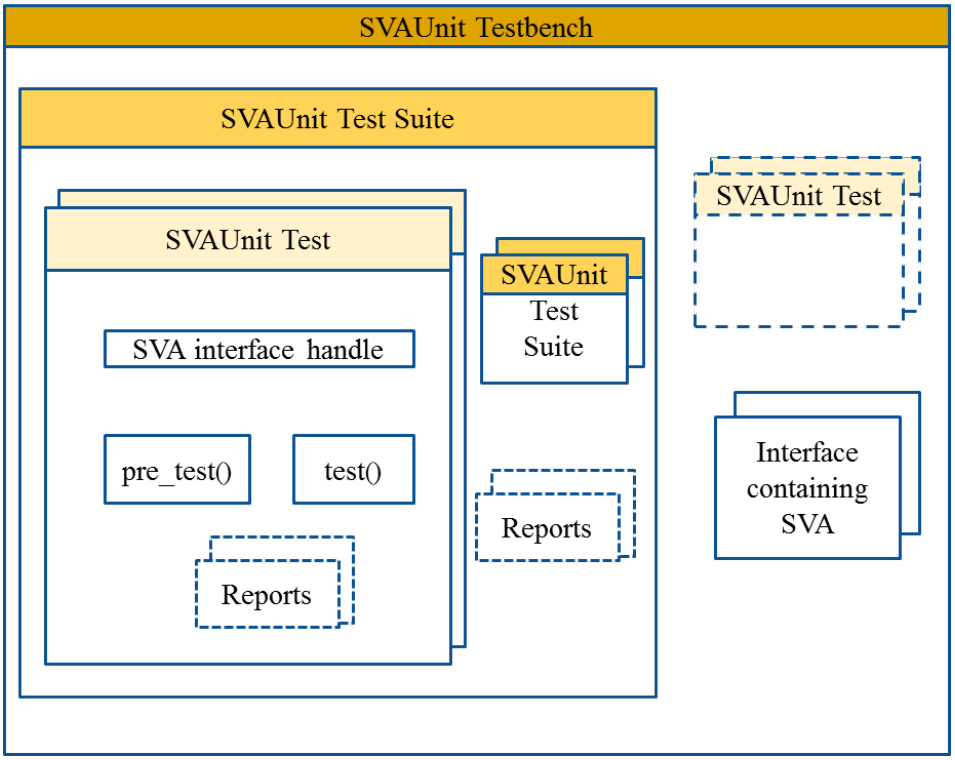
Como definido pela própria AMIQ (2015), o SVAUnit combina o paradigma de teste de unidade do mundo do software com o poderoso recurso de asserções de linguagens de verificação de hardware, como o SystemVerilog. O SVAUnit representa uma estrutura desenvolvida para testes unitários que permite ao usuário separar o código de validação de *assertions* do código de definição.

O ambiente fornece a capacidade de lidar com a integridade da verificação nas etapas iniciais do processo de desenvolvimento de código. As *assertions* podem ser validadas à medida que são escritas, sem ter que recorrer a uma lógica de verificação demorada e não reutilizável, uma vez que, o SVAUnit é um pacote compatível com UVM escrito em SystemVerilog. Ele fornece uma classe base para desenvolver testes de unidade para provar que as asserções são executadas conforme pretendido.

O SVAUnit fornece os meios para encapsular cada cenário de teste do SVA dentro de um teste de unidade. Ambientes cada vez mais complexos exigem reutilização, dessa forma com o uso do ambiente que fornece a capacidade de reutilizar cenários e estender a verificação do SVA para vários testes na mesma simulação por meio dos conjuntos de testes. Toda a funcionalidade pode ser facilmente controlada e supervisionada usando uma API simples. Embora o pacote SVAUnit possa ser facilmente integrado a um ambiente de verificação existente, o único requisito real é uma interface contendo os SVAs a serem verificados.

A estrutura do SVAUnit Testbench é um módulo que instancia uma ou mais interfaces que contêm SVA a serem testadas. O teste herda o uvm_test e contém a lógica de geração e verificação de estímulos. O SVAUnit Test Suite herda o $svaunit\_test$ e é usado para agregar mais testes de unidade e/ou *suites* de teste. Os conceitos são representados em grandes detalhes na Figura 8.

Figura 8 – Estrutura SVAUnit



Fonte: (AMIQ, 2015)

O pacote SVAUnit contém modelos de código que permitem ao usuário se concentrar no cenário de teste em vez de construir a infraestrutura. Dessa maneira o verificador deve se concentrar apenas em realizar a configuração dos principais blocos do ambiente.

3.1.1 SVAUnit Testbench

O SVAUnit Testbench representa um módulo SystemVerilog onde o pacote SVAUnit é usado. A estrutura do SVAUnit é ativada assim que se instancia o "'SVAUNIT_UTILS", que lidará com todo o trabalho de configuração do UVM de uma maneira que seja transparente da perspectiva do usuário.

Após a inicialização do SVAUnit Testbench, o usuário deve realizar a instanciação da interface de SVA e realizar uma referência de interface virtual deve ser configurada no "uvm_config_db"para ter acesso a ela posteriormente.

A utilização de interfaces parametrizadas ou múltiplas interfaces de uma só vez é totalmente suportado. O anexo A apresenta um exemplo de código mostra um banco de testes SVAUnit contendo várias instâncias de uma interface parametrizada.

3.1.2 SVAUnit Test

A classe SVAUnit Test herda o "uvm_test", o que significa que se beneficiará dos recursos de teste de base do UVM. O teste SVAUnit é usado para descrever e implementar um cenário que verifica um ou mais aspectos de uma *assertion*. A interface que contém os SVAs em teste é acessível através do "uvm_config_db"desde que foi definida a partir do banco de testes SVAUnit.

O SVAUnit Test contém dois métodos importantes: "pre_test() e test()". O método "pre_test()"deve conter a inicialização do cenário de verificação, enquanto que o cenário de verificação deve ser definido dentro do método "test()". O cenário contém a geração de estímulos SVA e a verificação do estado das *assertion*. Um exemplo de um teste SVAUnit é fornecido no anexo B.

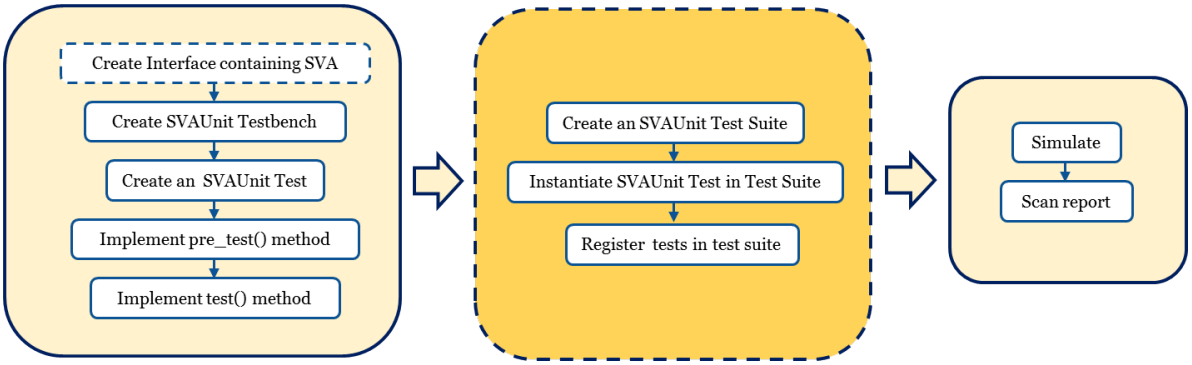
3.1.3 SVAUnit Test Suite

O SVAUnit Test Suite é útil quando são necessários mais de um teste SVAUnit. Ele herda a classe "svaunit_test"e um conjunto de execuções para outros grupos de teste. Os testes podem ser facilmente adicionados dentro do conjunto de testes usando o método "add_test()"após serem instanciados e criados. Eles serão executados na mesma ordem em que foram adicionados dentro da *Test suite*. Um teste pode ser excluído da execução, desabilitando-o através do método "disable_test()". Todas essas ações devem ser feitas dentro do corpo do "build_phase()"no anexo C temos um exemplo de código para SVAUnit Test Suite.

3.1.4 Fluxo

O fluxo de compilação do SVAUnit Test pode ser visto na Figura 9, e a partir dela podemos perceber que o fluxo de funcionamento do SVAUnit é simples e fácil de ser compreendido por parte do verificador, o que reduz o tempo gasto para se aprender a utilizar o SVAUinit otimizando assim o tempo de projeto, uma vez que, como é reduzido o período de estudo/montagem do ambiente de verificação formal, isso acarenta em um menor tempo gasto na realização do projeto, gerando menos gastos para a empresa.

Figura 9 – Diagrama de fluxo SVAUnit



Fonte: (AMIQ, 2015)

4 Descrição do Hardware

Com a necessidade de exemplificar toda a teoria proposta anteriormente foi utilizado o processador RISC-V:PULPino, para demonstrar a aplicação da verificação formal em um SoC, e auxiliar na compreensão de como esse modelo de verificação deve ser utilizado na pratica. Portanto nesse capitulo iremos apresentar as características e funcionalidades do PULPino, para que a partir do conhecimento a respeito do seu funcionamento seja possível planejas quais testes formais são adequados para a demonstração.

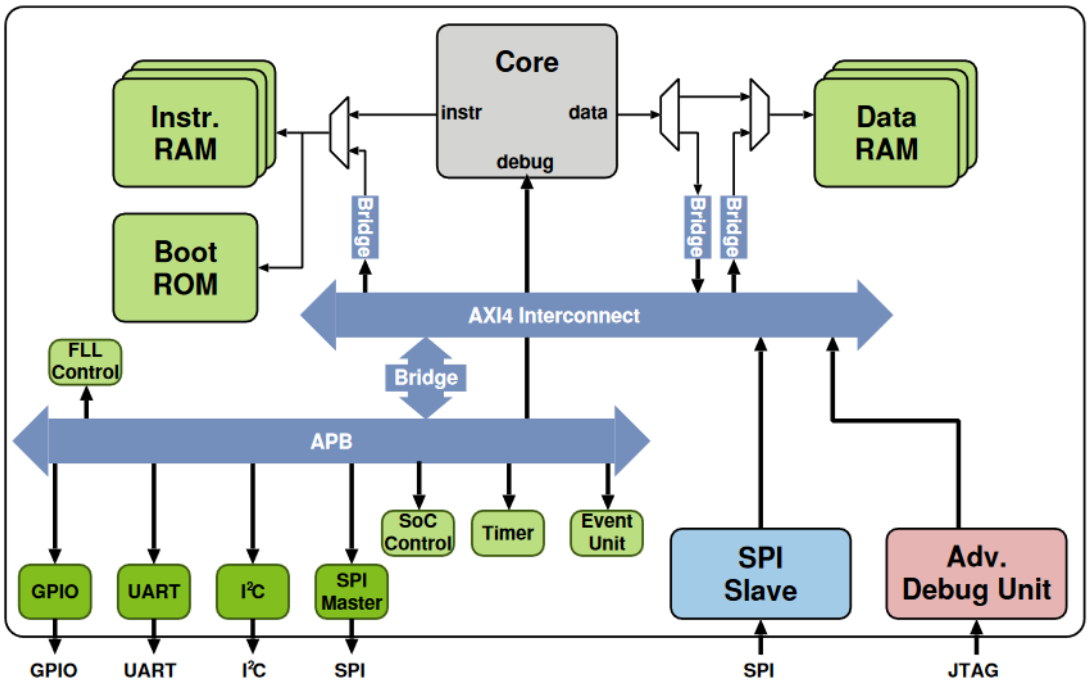
4.1 Parallel Ultra Low Power Processor - PULPino

O PULPino é um SoC, *open source*, de um único núcleo construído para o RISC-V sendo criada pela Universidade de Bologna e a ETH Zürich no ano de 2013 com objetivo de explorar novas e eficientes arquiteturas para processamento de baixo consumo de energia Traber e Gautschi (2017). O processador reutiliza a maioria dos componentes do seu antecessor o PULP, fazendo uso de RAMs de dados e instruções de uma porta única separadas, incluindo uma ROM de inicialização que contém um gerenciador de inicialização o qual pode carregar um programa via uma interface periférica serial (*Serial Peripheral Interface* - SPI) a partir de um dispositivo externos de uma *flash*.

O SoC usa um barramento AXI como principal interconexão, utilizando de uma ponte APB para periféricos simples. Os barramentos AXI e APB possuem largura de canais de dados com 32 bits. Para fins de teste, o SoC inclui uma unidade de *debugger* avançada que permite o acesso aos registradores principais, as duas RAMs e IOs mapeados pela memória via JTAG. Ambas as RAMs estão conectadas ao barramento AXI através de adaptadores de barramento, o esquemático do SoC pode ser visto na Figura 10.

Como explicado, por meio do barramento AXI-APB é possível se comunicar com os periféricos do PULPino sendo eles: GPIO, UART, I2C, *SPI Master*, *SoC Control*, *Timer* e *Event Unit*. Os quais serão descritos descritos nas seções posteriores. O PULPino foi desenvolvido voltado principalmente para a simulação RTL e ASICs, embora, também, haja uma versão para FPGA. As versões de FPGA não são especificamente otimizadas em termos de desempenho, pois são usadas principalmente como uma plataforma de emulação em vez de uma plataforma independente Traber e Gautschi (2017), nesse trabalho não sera analisado a versão para FPGA na realização da verificação.

Figura 10 – Estrutura do PULPino



Fonte: (TRABER; GAUTSCHI, 2017)

4.1.1 UART

Um UART, receptor/transmissor assíncrono universal é responsável por realizar a tarefa principal em comunicações seriais com computadores. O dispositivo muda informações paralelas para dados seriais que podem ser enviados em uma linha de comunicação. A UART usada no PULPino é compatível com um 16750. Ela possui todos os sinais UART típicos, apresentado ba Tabela 7, além de alguns sinais adicionais definidos pelo 16750.

Tabela 7 – Sinais Externos da UART.

Sinais	Direção	Descrição
uart_tx	Saída	Transmite dados
uart_rx	Entrada	Recebe dados
uart_rts	Saída	Solicitar envio
uart_cts	Entrada	Limpar para enviar
uart_dtr	Saída	Terminal de dados pronto
uart_dsr	Entrada	Grupo de dados pronto

Fonte: Feita pelo autor tomando como base (TRABER; GAUTSCHI, 2017)

4.1.2 GPIO

O GPIO, "entrada / saída para fins gerais", é um tipo de pino encontrado em um circuito integrado que não possui uma função específica. Embora a maioria dos pinos tenha um

propósito específico, como enviar um sinal para um determinado componente, a função de um pino GPIO é personalizável e pode ser controlada por software. No PULPino o GPIO possui nove registradores de 32 bits. Destacam-se os registradores PADDIR, PADIN e PADOUT. O primeiro controla a direção dos dados de cada um dos pinos de GPIO, o segundo é utilizado para os pinos de entrada e o último para os de saída, a Tabela 8 mostra os principais sinais do GPIO.

Tabela 8 – Sinais Externos da GPIO.

Sinais	Direção	Descrição
gpio_in[31:0]	Entrada	Transmitir Dados
gpio_out[31:0]	Saída	Receber Dados
gpio_dir[31:0]	Saída	Requerimento de envio
gpio_padcfg[5:0][31:0]	Saída	Configuração de Pad
interrupt	Saída	Interrupção

Fonte: Feita pelo autor tomando como base (TRABER; GAUTSCHI, 2017)

4.1.3 Mestre SPI

Interface serial periférica (SPI) é um barramento de interface comumente usado para enviar dados entre microcontroladores e pequenos periféricos, como registradores de deslocamento, sensores e cartões SD. Ele usa linhas de *clock* e dados separadas, juntamente com uma linha de seleção para escolher o dispositivo com o qual você deseja conversar. O SPI no PULPino assim como na GPIO apresenta nove registradores de 32 bits e os sinais internos do mesmo se encontram na Tabela 9.

Tabela 9 – Sinais do Mestre SPI.

Sinais	Direção	Descrição
spi_clk	Saída	Clock Mestre
spi_csn0	Saída	Seletor do Chip 0
spi_csn1	Saída	Seletor do Chip 1
spi_csn2	Saída	Seletor do Chip 2
spi_csn3	Saída	Seletor do Chip 3
spi_mode[1:0]	Saída	Modo SPI
spi_sdo0	Saída	Saída da linha 0
spi_sdo1	Saída	Saída da linha 1
spi_sdo2	Saída	Saída da linha 2
spi_sdo3	Saída	Saída da linha 3
spi_sdi0	Entrada	Entrada da linha 0
spi_sdi1	Entrada	Entrada da linha 1
spi_sdi2	Entrada	Entrada da linha 2
spi_sdi3	Entrada	Entrada da linha 3
events_o[1:0]	Saída	Eventos/Interrupção

Fonte: Feita pelo autor tomando como base (TRABER; GAUTSCHI, 2017)

4.1.4 I^2C

O protocolo I^2C, barramento de circuito inter-integrado, é notável por algumas características menos que simples: você não apenas conecta alguns pinos IC juntos e deixa o hardware de baixo nível assumir enquanto você lê ou grava no buffer apropriado, como é mais ou menos o caso com SPI ou um UART. O I^2C por sua vez apresenta seis registradores de 32 bits e os sinais internos do mesmo se encontram na Tabela 10.

Tabela 10 – Sinais do I^2C.

Sinais	Direção	Descrição
scl_pad_i	Entrada	Entrada SCL
scl_pad_o	Saída	Saída SCL
scl_padoen_o	Saída	Direção do pad SCL
sda_pad_i	Entrada	Entrada SDA
sda_pad_o	Saída	Saída SDA
sda_padoen_o	Saída	Direção do pad SDA
interrupt_o	Saída	Evento/Interrupção

Fonte: Feita pelo autor tomando como base (TRABER; GAUTSCHI, 2017)

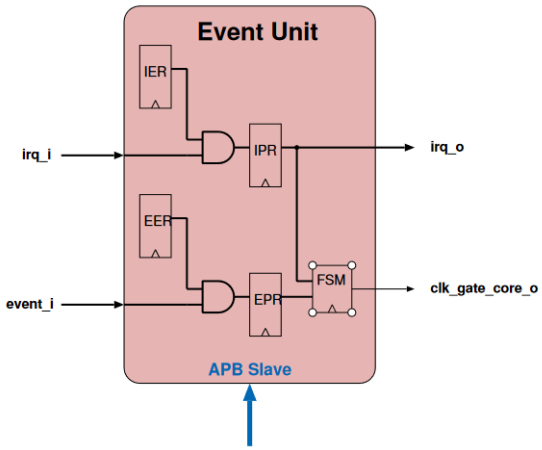
4.1.5 Timer

O módulo do timer possui dois temporizadores e cada um deles possui três registradores: TIMER, CTRL e CMP. O primeiro armazena o valor do temporizador e quando chega em 0xFFFFFFFF uma interrupção acontece. O segundo controla tanto quando é inicializado quanto um *prescaler* que determina o intervalo no qual o temporizador é incrementado. O último determina um limite, menor que 0xFFFFFFFF, para comparar o valor do módulo para que ocorra uma interrupção.

4.1.6 Event Unit

O PULPino possui uma unidade de interrupções e eventos que suporta 32 interrupções vetorizadas e acionamento de eventos de até 32 linhas de entrada. As linhas de interrupção e evento são mascaradas e armazenadas separadamente, o esquema do circuito da unidade e evento é representado na Figura 11.

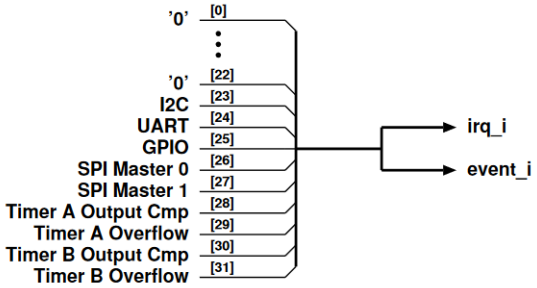
Figura 11 – Estrutura da Unidade de Eventos



Fonte: (TRABER; GAUTSCHI, 2017)

As interrupções e eventos são organizadas como apresentado na Figura 12. Vale destacar que irq_i e event_i possuem o mesmo valor.

Figura 12 – Bits de eventos



Fonte: (TRABER; GAUTSCHI, 2017)

Este módulo possui dez registradores de 32 bits. Nos quais os quatro registros de interrupções: IER é responsável por habilita as instruções por bits, IPR é o registrador de leitura e escrita de interrupções por linha, ISP é utilizado para estabelecer a interrupção no registrador IPR, e ICP ao colocar um bit, fara com que a interrupção correspondente em IPR seja zerada. Com os demais registradores de eventos: EER, EPR, ESP e ECP estes funcionam de maneira similar aos de interrupção.

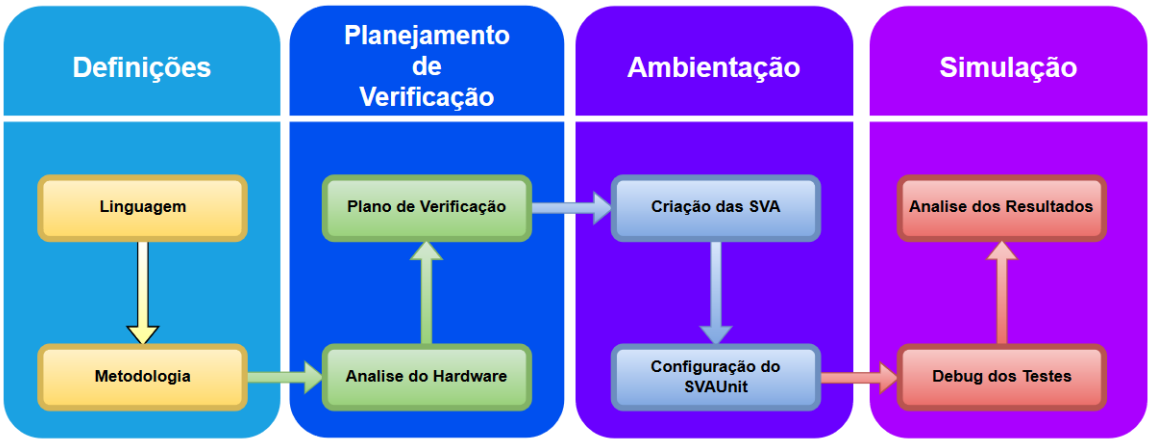
4.1.7 Controle do SoC

O PULPino possui um periférico APB pequeno e simples que fornece informações sobre a plataforma e fornece os meios para o preenchimento de bloco no ASIC. Possuindo seis registradores de 32 bits, nos quais se destacam o *PADMUx* no qual seu conteúdo pode ser usado para multiplexar *pads* ao direcionar um ASIC e o *Boot Address* que neste registrador contém o endereço de inicialização, é possível inicializar a partir de uma ROM ou diretamente da memória de instruções.

5 Guia Para Realizar Verificação Formal em Hardware

Uma vez definido nosso ambiente bem como qual metodologia e linguagem sera utilizada podemos, finalmente, estruturar um guia didático das etapas que devem ser seguidas para que seja possível realizar uma verificação formal em um hardware. Nesse trabalho o guia sera esquematizado de acordo com a avaliação do autor, que por meio de suas analises e experiencia com a verificação formal, definiu como linguagem de programação o SVA, juntamente com a metodologia da verificação de modelo. A partir dessas escolhas foi criado o fluxograma representativo do guia o qual pode se observar na Figura 13.

Figura 13 – Fluxograma de Verificação Formal



Fonte: Feito pelo Próprio Autor

5.1 Definição

Iniciamos o processo de verificação formal a partir da escolha da linguagem a ser utilizada, essa definição é importante pois toda a etapa de ambientação é depende dela. Cada linguagem de programação apresenta sua sintaxe própria bem como seu próprio ambiente de copilação, portanto iniciar o trabalho escolhendo qual linguagem será utilizada, fara com que seu trabalho seja direcionado, fazendo com que o verificador se concentre no que é mais importante, ou seja, realizar a verificação do hardware.

Para esse trabalho foi escolhida a linguagem de *SystemVerilog Assertion*, essa escolha foi feita devido a mair familiaridade do autor com a mesma, bem como, para facilitar o uso do ambiente em UVM do SVAUnit.

A definição da metodologia muitas vezes sera feita automaticamente a partir da escolha de qual linguagem e copilador sera utilizado, uma vez que, as linguagens de verificação formal são baseadas em suas metodologias, e os copiladores que as executam realizam os testes analisando o formalismo matemático que esta intrínseco na copilação.

Portanto nesse trabalho utilizamos a metodologia da verificação de modelo que é presente na linguagem de SVA. Após escolher essas duas definições iniciais é possivel da inicio as etapas verdadeiras da verificação formal, começando com o planejamento.

5.2 Planejamento de Verificação

A parte inicial do planejamento de verificação formal não difere da realizada na verificação funcional. A primeira etapa sempre sera realizar um estudo acerca do funcionamento do hardware a ser analisado, para que seja possível definir quais casos devem ser analisador funcionalmente, bem como quais os casos devem ser verificados formalmente. Após essa definição deve ser estudado um pouco mais a fundo o funcionamento do IP/SoC para que se tenha um entendimento, por parte do verificador, mais aprofundado e assim seja possível realizar a criação de assertivas que estejam de acordo com o funcionamento previsto pela documentação.

Após esse estudo mais aprofundado por parte do verificador pode-se da inicio ao planejamento de verificação, o qual deve conter quais funcionalidades ele ira verificar formalmente, e como sera realizado os estímulos para valida-las. Esse elaboração tem como objetivo auxiliar o verificador no seu trabalho, uma vez que, dessa maneira o mesmo não ira se "perder"durante o trabalho, e dificultara que ele esqueça algum caso que deva ser testado.

5.3 Ambientação

Finalmente damos incio, de fato, ao trabalho de verificação formal. A parte de ambientação tem como objetivo realizar a conexão entre o hardware e os testes a serem realizados. Iniciamos essa etapa com a criação das *assertions*e instanciação do SoC dentro de uma interface do SVAUnit, criado todas as assertivas o verificador pode da inicio a configuração do resto do ambiente do SVAUnit, como representado no fluxo ilustrado na Figura 9.

Após toda a ambientação com a criação das *assertions* e a configuração do SVAUnit para realizar a comunicação com o hardware, é possivel passar para a ultima etapa de verificação que é a fase de simulação.

5.4 Simulação

A etapa de simulação consiste na copilação de todo o ambiente de teste, para que seja possível iniciar a fase de *debug* das *assertions*, corrigindo possíveis erros de sintaxe, bem como analisando mais visualmente as logicas inseridas nas assertivas, validando se as mesmas estão em conformidade com a documentação.

Finalizando a verificação é feita a analise dos resultados e somado os dados colhidos no relatório final do projeto, podendo assim informar possíveis *bugs*, encontrados durante as simulação, para a equipe de *desig*. Melhorando dessa maneira a qualidade do produto final, e otimizando o trabalho da equipe.

6 ANÁLISE E DISCUSSÃO DOS RESULTADOS

Utilizando das diretrizes descritas no do capitulo 5, foi possível estruturar um planejamento adequado para realizar a verificação formal do PULPino, como exemplificação das teorias apresentadas nesse trabalho. Para isso, após a etapa de definições ser concluída foi iniciada a fase de planejamento, resultando no plano de verificação apresentado nesse capitulo.

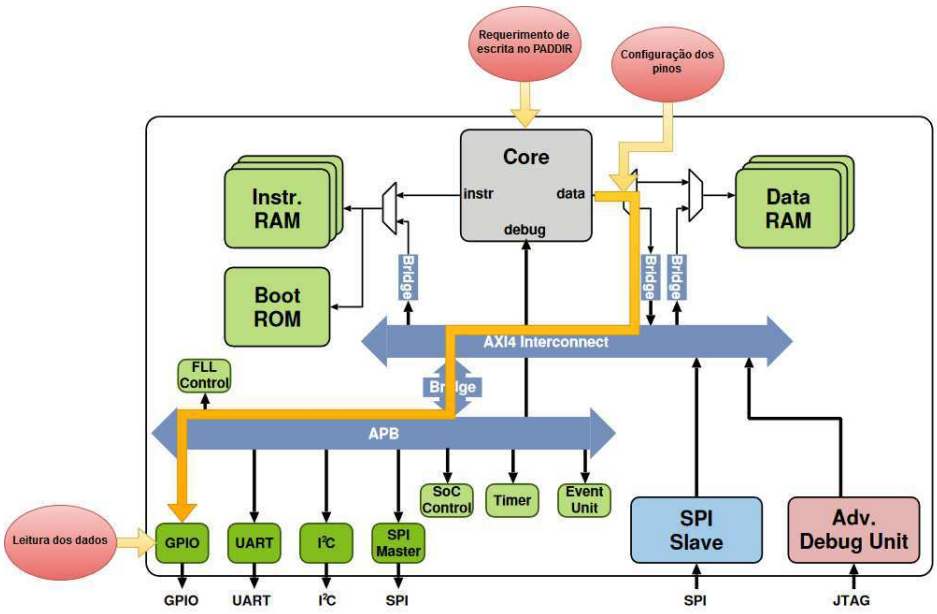
6.1 Plano de Verificação

Para realizar a verificação formal do PULPino foi definido alguns *corner cases*, que do ponto de vista do autor são validos de se verificar formalmente. Iniciamos nossas atividades alguns casos pontuais para serem verificados, sendo eles, uma verificação no GPIO que focada em analisar o procedimento de requerimento e gravação de dados de entradas no pinos, bem como para um requerimento e envio de dados de saída pelos pinos. Outro caso julgado relevante foi verificar a conexão dos barramentos AXI-APB com os periféricos internos do SoC, por fim como ultimo teste foi definido realizar um teste voltado a acessar endereçamentos reservados da memoria a partir do *CORE*.

6.1.1 GPIO

A verificação realizada no GPIO iniciou com a configuração do ambiente do SVAUinit como explicado na Seção 3.1, após essa configuração foi realizada a conexão com SoC e as assertivas. Com o ambiente pronto foi iniciado a fase de simulações na GPIO, que consistiu em primeiro configurar os pinos como entrada, a partir do registrador PADDIR, seguindo de uma escrita de um dado arbitrário no registrador PADIN, verificando se esse dado foi gravado e direcionado corretamente para as memorias pelo barramento. Na Figura 14 é apresentado o caminho dos dados verificados do teste.

Figura 14 – Fluxo de dados analisado no teste do registro PADDIR



Fonte: Feito pelo Próprio Autor

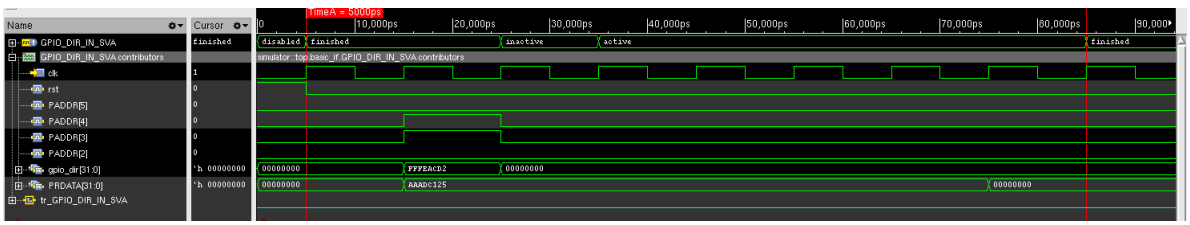
Tendo conhecimento desses dados foi criada a *assertion* responsável por garantir que esse caminho estava funcionando de acordo com o esperado. No Quadro 14 é apresentado o código da SVA implementada no teste, logo após é apresentado as formas de onda obtidas na simulação, representadas na figura 15.

Quadro 14 – Assertion utilizada para verificar a configuração do registrador PADDIR.

```
sequence reg_paddir
  (PADDR[5:2] == REG_PADDIR);
endsequence

sequence config_pad
  (gpio_dir == e_data_in);
endsequence

sequence read_data
  (PRDATA == gpio_dir);
endsequence

property gpio_dir_in_property;
  @(posedge clk)
    disable iff (rst)
      (reg_paddir |-> config_pad |=> ##[0:5]read_data);
endproperty

GPIO_DIR_IN_SVA: assert property(gpio_dir_in_property)
                 else `uvm_error("GPIO_DIR_IN_SVA",
                 "Configuracao dos PADs como entrada falhou");
```

Fonte: Feito pelo próprio autor.

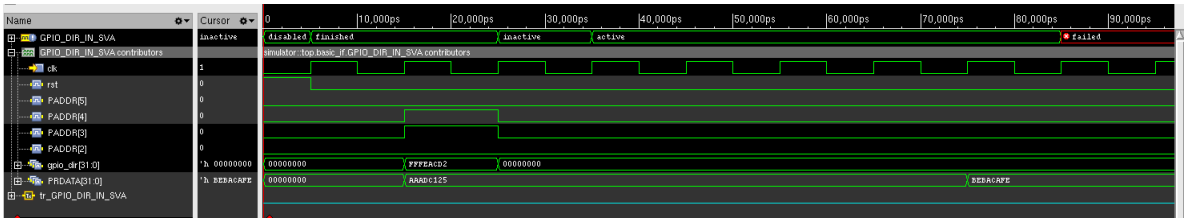
Figura 15 – Formas de onda da simulação referente a configuração o registrador PADDIR



Fonte: Feito pelo Próprio Autor

Como forma de exemplificar o *degub* de SVA foi feito um teste no qual era esperando uma falhar na *assertion*, dessa forma demonstrando assim o relatório das SVAs na forma de onda, apresentado na Figura 16, bem como o *log* do SVAUnit quando é inserida um erro proposital, na Figura 17.

Figura 16 – Formas de onda da simulação com falha para a configuração do registrador PADDIR



Fonte: Feito pelo Próprio Autor, Usando a Ferramenta SimVison da Cadence

Figura 17 – *Log* da simulação com falha referente a configuração o registrador PADDIR

```
------------------- PULPino_svaunit_gpio_test : Status statistics -------------------

        PULPino_svaunit_gpio_test SVAUNIT_PASS (5/5 SVAUnit checks PASSED)

UVM_INFO  @ 675000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Exercised SVAs -------------------

        1/4 SVA were exercised
                top.gpio_if.GPIO_DIR_IN_SVA

        3/4 SVA were not exercised
                top.gpio_if.GPIO_IN_SVA
                top.gpio_if.GPIO_DIR_OUT_SVA
                top.gpio_if.GPIO_OUT_SVA

UVM_INFO  @ 675000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Checks status summary -------------------

        CHECK_SVA_EXISTS 3/3 PASSED
        CHECK_SVA_ENABLED 1/1 PASSED
        CHECK_SVA_FAILED 1/1 PASSED

UVM_INFO  @ 675000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Checks for each SVA statistics -------------------

        GPIO_DIR_IN_SVA 5/5 checks PASSED
                CHECK_SVA_EXISTS 3/3 PASSED
                CHECK_SVA_ENABLED 1/1 PASSED
                CHECK_SVA_FAILED 1/1 PASSED

UVM_INFO  @ 675000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : ALL SVAs passed -------------------
```

Fonte: Feito pelo Próprio Autor Usando o Relatório do Ambiente SVAUnit

Dando continuidade aos testes da GPIO, após a configuração dos PADs como entrada é feita uma escrita de um dado arbitrário, e verificado, pelas SVAs apresentadas no Quadro 15, se esse dado foi devidamente acessado pelo CORE e guardado na memoria, o diagrama representativo do fluxo de dados dessa operação é apresentado na figura 18, seguido da Figura

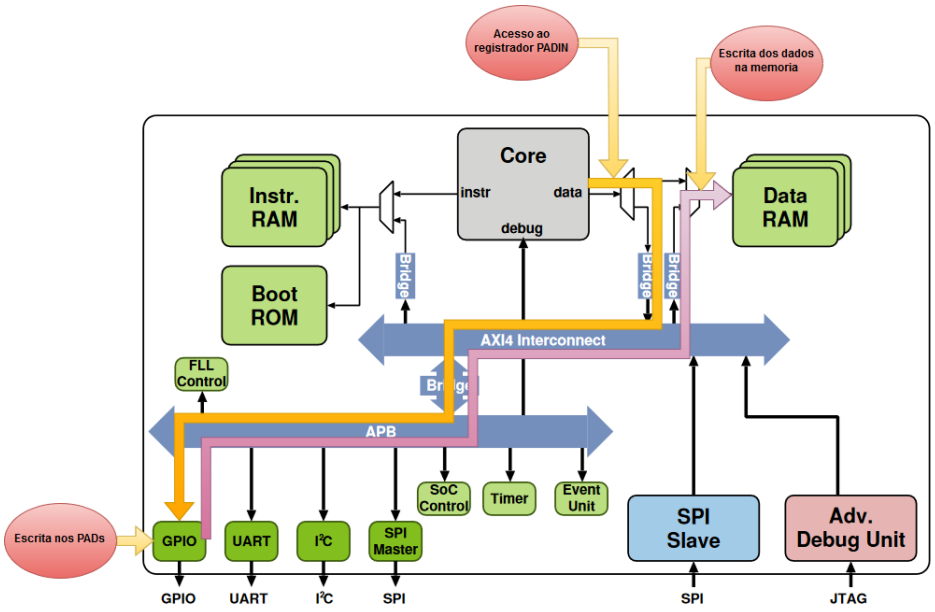
20 que representa as formas de ondas das *assertions* tando da configuração PADDIR como da operação de escrita, também apresentando o *log* obtido nessa simulação, na Figura 21.

Quadro 15 – Assertion utilizada para verificar entrada de dados na GPIO.

```
property gpio_in_property;
    @(posedge clk)
        disable iff (rst)
        ((PADDR[5:2]==REG_PADIN)|->##[0:5](PRDATA==gpio_in));
endproperty
GPIO_IN_SVA: assert property(gpio_in_property)
             else `uvm_error("GPIO_IN_SVA",
                             "Dado de Entrada do PAD falhou");
```

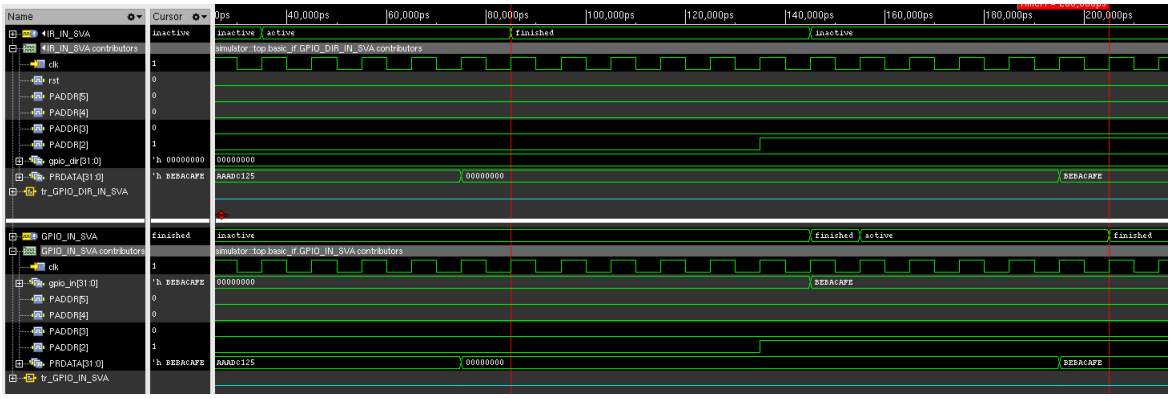
Fonte: Feito pelo próprio autor.

Figura 18 – Fluxograma dos dados de escritos nos PAD da GPIO até a memoria



Fonte: Feito pelo Próprio Autor.

Figura 19 – Formas de onda das SVAs da operação de escrita na GPIO



Fonte: Feito pelo Próprio Autor, Usando a Ferramenta SimVison da Cadence.

Figura 20 – *Log* das SVAs da operação de escrita na GPIO

```
------------------- PULPino_svaunit_gpio_test : Status statistics --------------------

        PULPino_svaunit_gpio_test SVAUNIT_PASS (20/20 SVAUnit checks PASSED)

UVM_INFO  @ 795000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Exercised SVAs --------------------

        2/4 SVA were exercised
                top.gpio_if.GPIO_DIR_IN_SVA
                top.gpio_if.GPIO_IN_SVA

        3/4 SVA were not exercised
                top.gpio_if.GPIO_DIR_OUT_SVA
                top.gpio_if.GPIO_OUT_SVA

UVM_INFO  @ 795000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Checks status summary --------------------

        CHECK_SVA_EXISTS 6/6 PASSED
        CHECK_SVA_ENABLED 2/2 PASSED
        CHECK_SVA_PASSED 2/2 PASSED

UVM_INFO  @ 795000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : Checks for each SVA statistics --------------------

        GPIO_DIR_IN_SVA 5/5 checks PASSED
                CHECK_SVA_EXISTS 3/3 PASSED
                CHECK_SVA_ENABLED 1/1 PASSED
                CHECK_SVA_PASSED 1/1 PASSED

        GPIO_IN_SVA 5/5 checks PASSED
                CHECK_SVA_EXISTS 3/3 PASSED
                CHECK_SVA_ENABLED 1/1 PASSED
                CHECK_SVA_PASSED 1/1 PASSED

UVM_INFO  @ 795000 ns [PULPino_svaunit_gpio_test]:

------------------- PULPino_svaunit_gpio_test : ALL SVAs passed --------------------
```

Fonte: Feito pelo Próprio Autor, Usando o Relatório do Ambiente SVAUnit.

Assim como foi feito para uma escrita no PADs da GPIO, também foi realizado um teste do caminho inverso, ou seja, foi realizado teste de leitura. Iniciamos com a configuração da registrador de PADDIR, que é feita da mesma maneira do fluxo da Figura 14, diferindo apenas no valor do dado escrito no registro que no lugar de 0, que configura como entrada, usamos 1, que configura como saída.

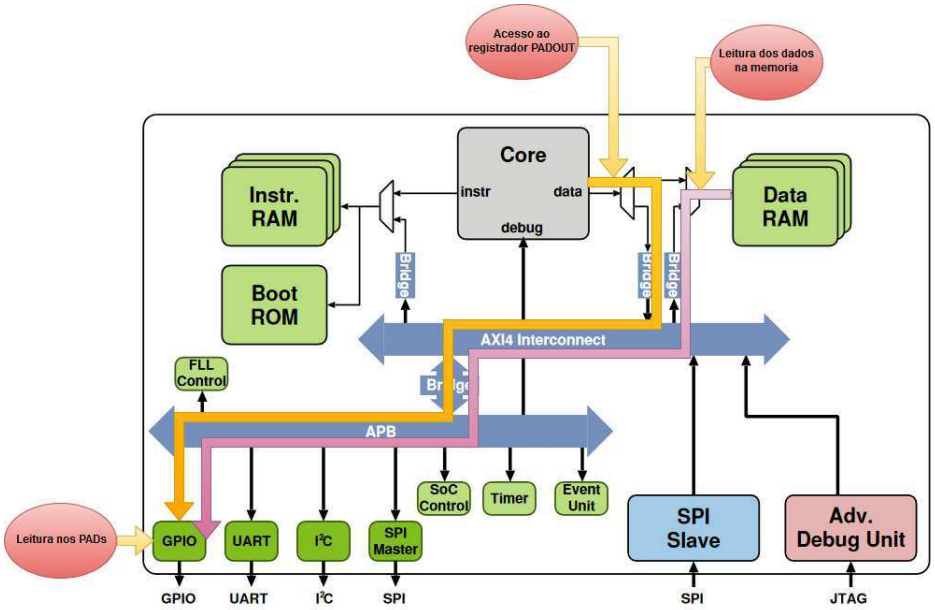
O procedimento de leitura dos dados no PAD, é feita utilizando o registrador interno do GPIO que realiza uma leitura na memoria para acessar os dados salvos, e informa-los aos PADs. Esse fluxo de dados esta representado na Figura 21, assim como as *assertions* utilizadas nesse testes se encontram no Quadro 16 e as formas de onda na Figura 22.

Quadro 16 – Assertion utilizada para verificar saída de dados na GPIO.

```
sequence reg_paddir
  (PADDR[5:2] == REG_PADDIR);
endsequence

sequence config_pad
  (gpio_dir == e_data_in);
endsequence

sequence read_data
  (PRDATA == gpio_dir);
endsequence

property gpio_dir_out_property;
  @(posedge clk)
    disable iff (rst);
    ((reg_paddir && config_pad) |-> ##[0:5] read_data);
endproperty

GPIO_DIR_OUT_SVA: assert property(gpio_dir_out_property)
                  else `uvm_error("GPIO_DIR_OUT_SVA",
                  "Configuracao dos PADs como saida falhou");

property gpio_out_property;
  @(posedge clk)
    disable iff (rst)
    ((PADDR[5:2]==4'b0010)|->##[0:5](gpio_out==PWDATA));
endproperty

GPIO_OUT_SVA: assert property(gpio_out_property)
              else `uvm_error("GPIO_OUT_SVA",
                        "Dado de Saida do PAD falhou");
```

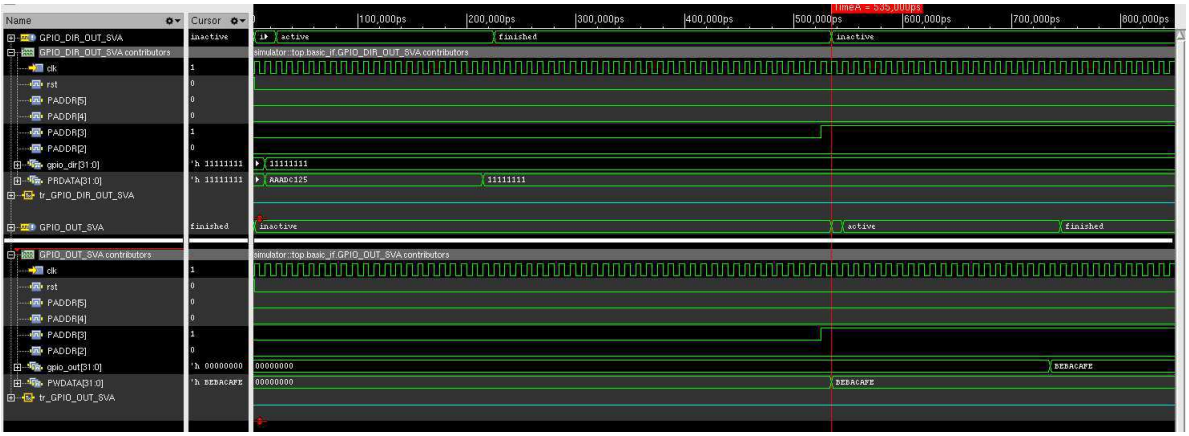
Fonte: Feito pelo próprio autor.

Figura 21 – Fluxograma dos dados de leitura nos PAD da GPIO vindo da memoria



Fonte: Feito pelo Próprio Autor.

Figura 22 – Fluxograma dos dados de leitura nos PAD da GPIO vindo da memoria



Fonte: Feito pelo Próprio Autor, Usando a Ferramenta SimVison da Cadence.

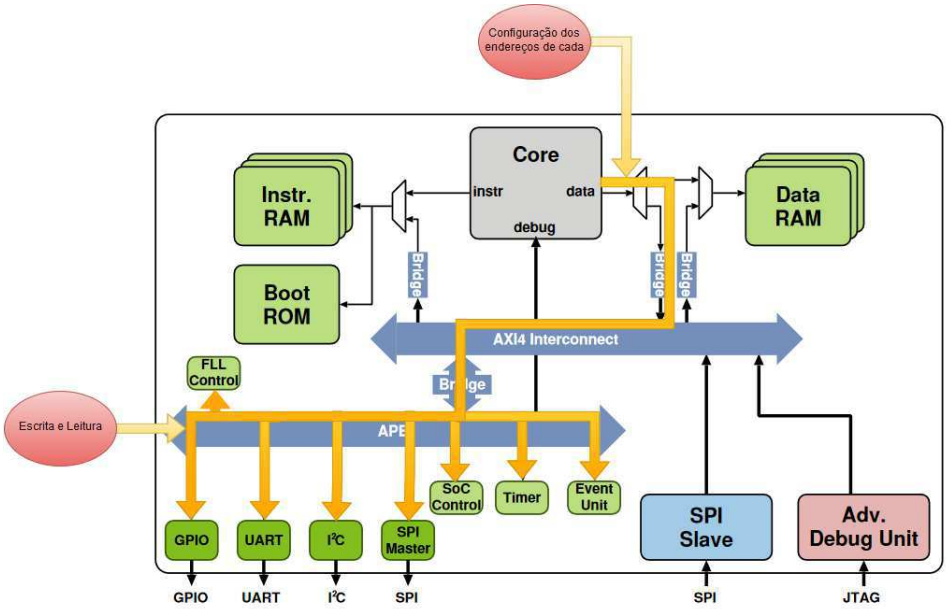
6.1.2 Barramentos AXI-APB

Realizar testes de conexão de barramento é uma pratica de verificação fundamental dentro de um SoC, uma vem que o mesmo é responsável por todas as conexões do hardware. Garantir o bom funcionamento desse componente proporciona para o verificador, a possibilidade de verificar os periféricos sem a preocupação de que os dados não estejam chegando ao seu destino.

Para o PULPino esse teste foi voltado para o objetivo de garantir que todos os IPs periféricos estavam sendo conectados ao CORE por meio dos barramentos AXI-APB, sem que ocorresse possíveis falhas de conexão durante o trajeto. Dessa maneira o padrão teve como objetivo estimular separadamente cada um dos periféricos por meio de uma simples

requerimento de escrita e leitura, o fluxo de dados previsto para esse teste esta representado na Figura 23.

Figura 23 – Fluxograma dos dados de conexão dos barramentos com os periféricos



Fonte: Feito pelo Próprio Autor.

O teste foi realizado configurando inicialmente cada um dos endereços dos periféricos de acordo com o mapa de memoria mostrado na Figura 24, após o endereçamento de cada periférico, foi realizado uma escreta seguida de uma leitura em cada um dos IPs. O resultado da simulação comprovou o esperado, ou seja, todos os periféricos testados responderam corretamente ao requerimento realizado pelo CORE.

As *assertions* desenvolvidas nessa verificação visavam monitorar os sinais de conexão dos barramento que eram responsáveis por transmissão dos dados tanto de escrita como de leitura, o sinal de requerimento de escrita ou leitura, bem como o sinal de endereçamento, o código desenvolvido se encontra nos Quadros 17 e 18.

Quadro 17 – Assertion utilizada para verificar os barramentos AXI e APB durante uma leitura.

```
property P_AXI_APB_R(trigger_valid, trigger_ready,
                     dado_core, dado_periferico,
                     ip_addrs, addrs_esperado, read_write);
  @(posedge clk)
    disable iff ((rst) || (ip_addrs != addrs_esperado)
                 || (read_write))
    ($rose(trigger_valid) |-> ##[0:$] $rose(trigger_ready)
    |-> ##[0:TOLERNACIA] (dado_core == dado_periferico));
endproperty

A_AXI_APB_R: assert property (P_AXI_APB_R(ARVALID, ARREADY,
                              PRDATA, ARADDR, PADDR, PWRITE))
             else `uvm_error("A_AXI_APB_R",
                             "Tentativa de Leitura Falha");
```

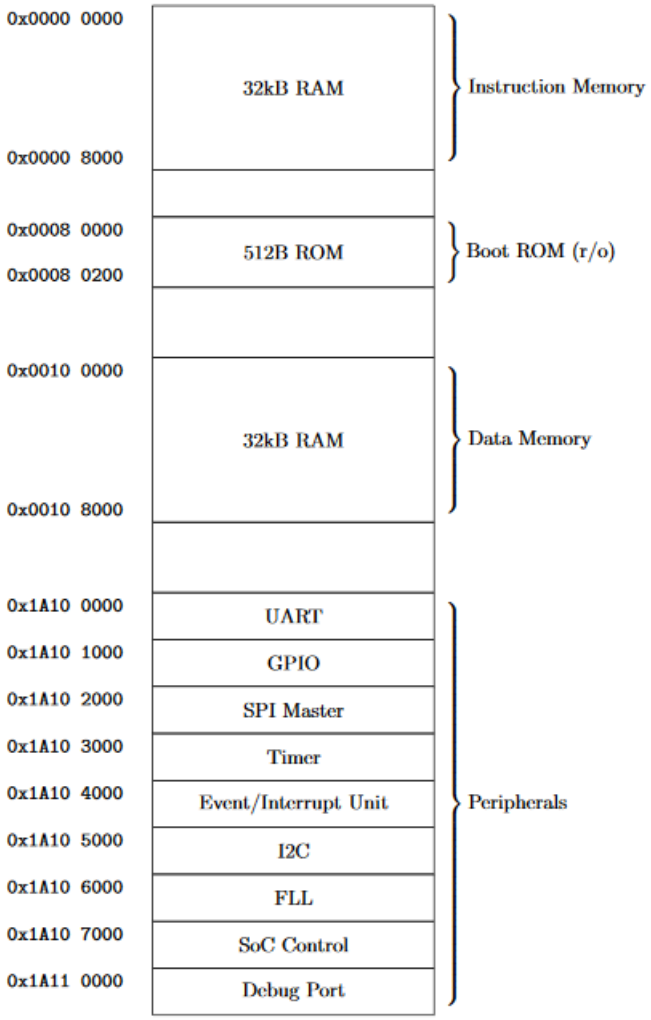
Fonte: Feito pelo próprio autor.

Quadro 18 – Assertion utilizada para verificar os barramentos AXI e APB durante uma escrita.

```
property P_AXI_APB_W(trigger_valid, trigger_ready,
                     dado_core, dado_periferico,
                     ip_addrs, addrs_esperado, read_write);
  @(posedge clk)
    disable iff ((rst) || (ip_addrs != addrs_esperado)
                 ||(!read_write))
    ($rose(trigger_valid) |-> ##[0:$] $rose(trigger_ready)
    |-> ##[0:TOLERNACIA] (dado_core == dado_periferico));
endproperty

A_AXI_APB_W: assert property (P_AXI_APB_W(ARVALID, AWREADY,
                              PWDATA, AWADDR, PADDR, PWRITE))
         else `uvm_error("A_AXI_APB_W",
                         "Tentativa de Escrita Falha");
```

Fonte: Feito pelo próprio autor.

Figura 24 – Mapa de Memoria do PULPino



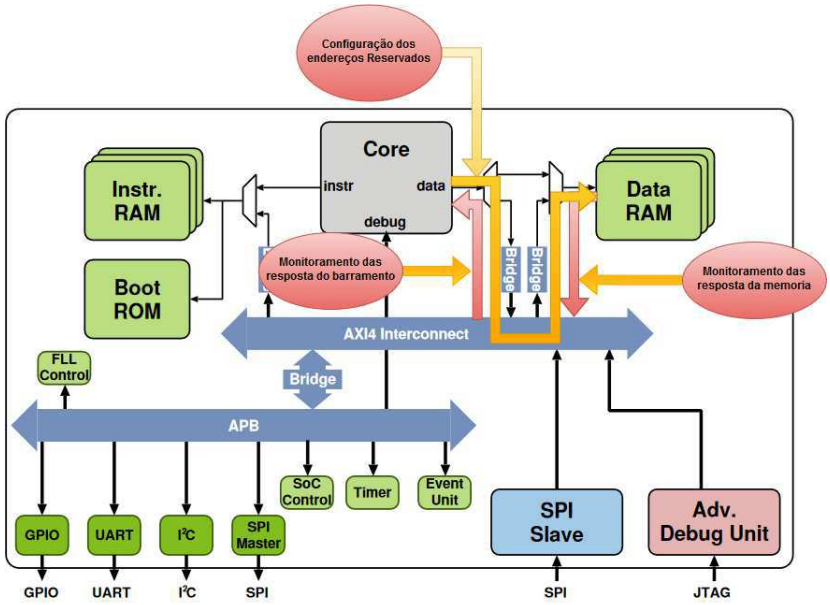
Fonte: (TRABER; GAUTSCHI, 2017)

6.1.3 Teste de Acesso Reservado

Por fim como ultimo teste realizado, foi feito uma verificação de acesso reservado a memoria. Esse tipo de teste é comum de ser realizado no ambiento de um projeto de SoC o mesmo tem como objetivo realizar a cobertura dos sinais de resposta dos barramentos que normalmente não são estimulados pelos outros tipos de testes, dessa maneira foi desenvolvido um padrão que visava garantir que o sinal de "resposta"do barramento AXI estava funcionando de acordo com o esperado, quando era tentado por parte do CORE acessar um endereço reservado do SoC.

Esse teste foi feito de forma que o CORE realizava uma escrita e uma leitura nos endereços reservados, que são apresentados na Figura 24, e era esperado que esse processo falhasse retornando uma "resposta"de erro por parte do barramento AXI, garantindo assim a variação do sinal, o que gera um *toggle* validando-o no relatório de cobertura do projeto. O fluxograma dos dados é apresentado na Figura 25.

Figura 25 – Fluxograma dos dados com a memoria e da "resposta"do barramento



Fonte: Feito pelo Próprio Autor.

A *assertion* criada para essa verificação tinha como função monitoras os sinais de erro de leitura RRESP e o de erro de escrita BRESP. O código gerado para esse teste se encontra nos quadros 19 e 20

Quadro 19 – Assertion utilizada para verificar os sinais de "resposta"do AXI durante uma leitura.

```
property P_AXI_RESP_R(trigger_valid, trigger_ready,
                      dado_core, dado_periferico,
                     ip_addrs, addrs_esperado,
                     read_write, sinal_resposta);
  @(posedge clk)
    disable iff (rst || (ip_addrs != addrs_esperado)
                 || (read_write))
    ($rose(trigger_valid) |-> ##[0:$] $rose(trigger_ready)
    |-> ##[0:TOLERNACIA] ((dado_core != dado_periferico)
    && ($rose(sinal_resposta))));
endproperty

A_AXI_RESP_R: assert property (P_AXI_RESP_R(ARVALID, ARREADY,
                               PRDATA, ARADDR, PADDR, PWRITE,
                               RRESP))
        else `uvm_error("A_AXI_RESP_R",
                        "Sinal de Resposta N o levantou");
```

Fonte: Feito pelo próprio autor.

Quadro 20 – Assertion utilizada para verificar os sinais de "resposta"do AXI durante uma escrita.

```
property P_AXI_RESP_W(trigger_valid, trigger_ready,
                      dado_core, dado_periferico,
                      ip_addrs, addrs_esperado,
                      read_write, sinal_resposta);
  @(posedge clk)
    disable iff (rst || (ip_addrs != addrs_esperado)
                 || (~read_write))
     ($rose(trigger_valid) |-> ##[0:$] $rose(trigger_ready)
     |-> ##[0:TOLERNACIA] ((dado_core != dado_periferico)
     && ($rose(sinal_resposta))));
endproperty

A_AXI_RESP_W: assert property (P_AXI_RESP_W(ARVALID, AWREADY,
                                PWDATA, AWADDR, PADDR, PWRITE,
                                BRESP))
        else `uvm_error("A_AXI_RESP_W",
                        "Sinal de Resposta Nao levantou");
```

Fonte: Feito pelo próprio autor.

6.2 Discursão

A partir dos resultados obtidos com os testes foi possível exemplificar praticamente como é realizada uma verificação formal em um hardware, o que contribui como melhoria para o desenvolvimento geral do projeto. Alguns podem se questionar se esses casos analisados são os únicos casos validos para se realizar uma verificação formal no PULPino, a resposta é não. Como esse trabalho não tinha como foco realizar a verificação formal do processador, mas sim apresentar um guia pratico e exemplificado de como se realizar um projeto de verificação formal.

Existem diversos outros *corner cases* que são factíveis de se verificar formalmente dentro do PULPino, uma vez que o que foi apresentado nesse trabalho não chega a representar os 100% de cobertura desejados num projeto, num ambiente empresarial é recomendado realizar em conjunto a verificação funcional com a formal, proporcionando assim um trabalho mais robusto e confiável o que gerara para empresa um retorno financeiro maior devido a maior confiabilidade do seu produto.

7 Conclusão

A utilização de verificação formal no âmbito empresarial, se comprova totalmente viável mesmo se que seja levado em conta a complexidade por trás da utilização desses métodos. Isso se deve ao fato da mesma otimizar o tempo e qualidade da verificação do projeto, gerando para a empresa um maior garantia da qualidade do seu produto bem como um menor gasto devido a redução dos prazos de produção.

Portanto esse trabalhou cumpriu com seu primeiro objetivo, uma vez que, foi realizada uma apresentação de como se realizar uma verificação formal em um hardware mostrando para o verificador quais etapas devem ser seguidas e levadas em conta durante um projeto. Essa etapa foi apresentada de forma didática e exemplificada na qual se torna viável para ser usada por verificadores em seus projetos.

O segundo e terceiro objetivos também foram compridos, já que durante o trabalho foi apresentando duas importantes linguagens de verificação formal e posteriormente foi selecionada qual delas seria mais adequada para ser explanada nesse trabalho, juntamente com o ambiente de verificação utilizado.

Por fim os demais objetivos foram compridos com a criação do guia pratico que apresenta quais são as etapas que um verificador deve seguir para realizar uma verificação formal em um hardware. Exemplificando essas informações por meio de testes realizados no processador RISC-V:PULPino, os quais além de apresentar as assertivas da linguagem SVA durante a discussão dos resultados, também foi apresentado os dados obtidos durante a algumas simulações, tornando mais visível para os leitores o método de *debug* que pode ser utilizado durante um projeto.

Referências

ABADI, M.; LAMPORT, M. The existence of refinement mappings. Theoretical Computer Science, New York, v. 82, p. 253–284., 1991. Citado na página 11.

ACCELLERA. Property specification language: Reference manual. Accellera: Reference Manual, Napa, 2003. Citado 5 vezes nas páginas 19, 20, 21, 22 e 23.

AMIQ. **How to Verify SystemVerilog Assertions with SVAUnit**. 2015. Disponível em: <https://www.amiq.com/consulting/2015/04/29/how-to-verify-systemverilog-assertions-with-svaunit/>. Acesso em: 28 de Novembro de 2018. Citado 6 vezes nas páginas 29, 30, 31, 55, 56 e 57.

BIERE, A. et al. Symbolic model checking without bdds. in tools and algorithms for the construction and analysis of systems. **London, UK**, Cleaveland, p. 193–207, 1999. Citado na página 14.

BIERE, A. et al. Bounded model checking. **Advances in Computers**, Switzerland, v. 58, 2003. Citado na página 15.

BIERE, A. et al. **Handbook of Satisfiability**. 1. ed. Amsterdam: IOS Press, 2009. Citado na página 14.

BLACKBURN, P.; RIJKE, M. de; VENEMA, Y. **Handbook of Satisfiability**. 1. ed. Cambridge: Cambridge University Press, 2001. Citado na página 5.

BURCH, J. et al. Symbolic model checking: An approach to the sate explosion problem. **Inf. Comput**, Pittsburgh, n. 98, p. 142–170, 1992. Citado 2 vezes nas páginas 12 e 14.

DOULOS. **BibTeX.org**. 2014. Disponível em: <https://www.doulos.com/knowhow/sysverilog/tutorial/assertions/>. Acesso em: 28 de Novembro de 2018. Citado 5 vezes nas páginas 24, 25, 26, 27 e 28.

EDGAR, S.-M.; DAVID, M.-V. State of the art in the research of formal verification: Estado del arte de la investigación en verificación forma. **Ingeniería Investigación y Tecnología**, XV, n. 4, p. 615–623, 2014. Citado na página 1.

EMERSON, E. Temporal and modal logic. **Handbook of Theoretical Computer Science**, Cambridge, B, p. 995–1072, 1990. Citado na página 5.

EMERSON, E. A.; SISTLA, A. P.; CLARKE, E. M. Automatic verification of finite-state concurrent systems using temporal logic specifications. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, New York, v. 8, p. 244–263, 1986. Citado na página 6.

FLEURIOT, J. **Lecture 5: Computation Tree Logic (CTL)**. 2015. Disponível em: <http://www.inf.ed.ac.uk/teaching/courses/ar/slides/>. Acesso em: 28 de Novembro de 2018. Citado na página 7.

GERTH, R. Foundations of compositional program refinement: Safety properties. in stepwise refinement of distributed systems. Lecture Notes in Computer Science, New York, v. 430, p. 777–808, 1989. Citado na página 11.

GHEORGHE, S. Aadl and ieee - 1850 property specification language. Canadá, 2010. Citado na página 20.

GORDON, T. F. M. M. J. C. **Introduction to HOL: A theorem proving environment for higher order logic**. 1. ed. Cambridge: Cambridge University Press, 1993. Citado na página 16.

KRIPKE, S. A. Semantical considerations on modal logic. **Acta Philosophica Fennica**, Finland, v. 16, p. 83–94, 1963. Citado na página 5.

KUNDU, S. High-level verification of system designs. UNIVERSITY OF CALIFORNIA, SAN DIEGO, SAN DIEGO, p. 1–34, 2009. Citado 2 vezes nas páginas 8 e 10.

MANNA, Z. et al. An up date on step: Deductive-algorithmic verification of reactiv e systems. Computer Science Department Stanford Universit, Stanford, 1999. Citado na página 18.

MARETTI, N. Mechanized verification of refinement. In Proceedings of the Second International Conference on Theorem Provers in Circuit Design (TPCD'94). Lecture Notes in Computer Science, New York, v. 901, p. 185–202, 1994. Citado na página 11.

MCMILLAN, K. Symbolic model checking: 1020 states and beyond. **PhD Thesis, Carnegie Mellon University**, Pittsburgh, 1992. Citado 2 vezes nas páginas 12 e 13.

MEHTA, A. B. **SystemVerilog Assertions and Functional Coverage**. 1. ed. Berlim: Springer, 2014. Citado na página 24.

MELHAM, T. F. Aformalizing abstraction mechanisms for hardware verification in higher order logic. **Technical Report - University of Cambridge**, Cambridge, v. 201, p. 30–47, 1990. Citado na página 9.

RAY, S. **Scalable Techniques for Formal Verification**. 1. ed. New York: Springer US, 2010. Citado na página 16.

SANGHAVI, A. What is formal verification? **EE Times-Asia**, n. 2, 2010. Citado na página 3.

SCHMALTZ, J. **2IX20 Software Specification**. 1. ed. Eindhoven: Eindhoven University of Technology, 2017. Citado na página 6.

SHANKAR., S. O. D. J. R. D. N. Pvs: A prototype verification system. 11th International Conference on Automated Deduction (CADE), Saratoga, v. 11, p. 748–752, 1992. Citado na página 17.

SLOBODOVA, W. A. H. M. K. J. S. M. A. Industrial hardware and software verification with acl2. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, Londres, 2017. Citado na página 17.

TRABER, A.; GAUTSCHI, M. Pulpino: Datasheet. Universidade de Bologna, Zürich, 2017. Citado 6 vezes nas páginas 32, 33, 34, 35, 36 e 48.

Anexos

ANEXO A – SVAUnit Testbench para múltiplas interfaces parametrizadas

```
module top;
  ...

  my_if#(100) dut_if(.clk(clock));

  initial begin
    uvm_config_db#(virtual my_if#(100))::set(uvm_root::get(),
    "*",         "VIF", dut_if);
  end
  ...
endmodule

module top;
  ...

  generate
    genvar if_param;

    for(if_param = 100; if_param < 200; if_param++) begin
      my_if#(if_param) dut_if(.clk(clock));

      initial begin
        uvm_config_db#(virtual
        my_if#(.if_param(if_param)))::set(uvm_root::get(),
        "*", $sformatf("vif%0d", if_param), dut_if);
      end
    end
  endgenerate

  ...
endmodule
```

Fonte: (AMIQ, 2015)

ANEXO B – SVAUnit Test

```
class ut1 extends svaunit_test;
  virtual my_if vif;

  function void build_phase(input uvm_phase phase);
    if (!uvm_config_db#(virtual my_if)::get(this, "", "vif", vif))
      `uvm_fatal("UT1_NO_VIF_ERR", "SVA IF is not set!")

      // The test is configured to run by default
      disable_test();
  endfunction

  task pre_test();
    // Initialize signals
  endtask

  task test();
    // Create scenarios for AN_SVA
  endtask
endclass
```

Fonte: (AMIQ, 2015)

ANEXO C – SVAUnit Test Suit

```
class uts extends svaunit_test_suite;
  ut1 unit_test1;
  ...
  ut2#(virtual my_if#(100)) unit_test2;

  function void build_phase(input uvm_phase phase);
    unit_test1 = ut1::type_id::create("unit_test1", this);
    ...
    unit_test2 = ut2#(virtual
    my_if#(100)))::type_id::create("unit_test2", this);

    add_test(unit_test1);
    ...
    add_test(unit_test2);
    unit_test2.disable_test();
  endfunction
endclass
```

Fonte: (AMIQ, 2015)