

José Augusto Miranda Nacif
Orientador: Claudionor Nunes Coelho Jr.

Processador de asserções para depuração de circuitos integrados em tempo de execução

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Outubro, 2004

Belo Horizonte



UNIVERSIDADE FEDERAL DE MINAS GERAIS



FOLHA DE APROVAÇÃO

Processador de Asserções para Depuração de Circuitos Integrados em Tempo de
Execução

JOSÉ AUGUSTO MIRANDA NACIF

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

Prof. LUIGI CARRO
Departamento Engenharia Elétrica da UFRGS

DR. LUIZ FERNANDO ETRUSCO MOREIRA
I-Vision Ltda

Belo Horizonte, 07 de outubro de 2004.

Abstract

White-box verification is a technique that reduces observability problems by locating a failure during design simulation without the need to propagate the failure to the I/O pins. White-box verification in chip level designs can be implemented using assertion checkers to ensure the correct behavior of a design. With chip gate counts growing exponentially, today's verification techniques, such as white-box, can not always ensure a bug free design. This work proposes an assertion processor to be used with synthesized assertion checkers in released products to enable intelligent debugging of deployed designs. Extending white-box verification techniques to deployed products helps locate errors that were not found during simulation/emulation phases. We present results of the insertion of assertion checkers and an assertion processor in three different microprocessor cores. We also show that the insertion of these assertion checkers added minimal area and speed overheads to the design.

Resumo

Verificação caixa-branca é uma técnica que reduz problemas de observabilidade localizando um erro durante a simulação sem a necessidade de propagação da falha para os pinos de E/S. No desenvolvimento de circuitos integrados, a verificação caixa-branca pode ser implementada através de asserções. Asserções são monitores instanciados pelo projetista de forma a garantir o comportamento correto do circuito integrado. Com a complexidade dos circuitos integrados crescendo exponencialmente, as técnicas tradicionais de verificação como a verificação caixa-branca, nem sempre são suficientes para localizar todos os erros de um projeto. Este trabalho propõe um processador de asserções para ser usado conjuntamente com asserções sintetizadas de forma que um circuito integrado possa ser verificado depois de sua comercialização. A extensão de técnicas de verificação caixa-branca para circuitos integrados já comercializados permite a localização de erros não identificados nas etapas de simulação/emulação. Resultados da inserção do processador de asserções em três diferentes microprocessadores são apresentados. A inclusão destas asserções apresentou um custo mínimo de área e velocidade nestes microprocessadores.

Agradecimentos

A vida é curta, e a estrada é longa. Por mais este passo dado, agradeço a Deus; a meus pais, Paulo e Maria Déa, e meus irmãos, Paulo e Durandi, meus pilares de sustentação, pelo amor e cuidados constantes e por saberem entender minha ausência.

À minha namorada Soninha, por toda paciência, compreensão e apoio nos momentos mais difíceis desta longa caminhada.

Claudionor e Otávio, cujos conhecimento e experiência de vida sempre me auxiliaram, muito obrigado por serem a "luz no fim do túnel", o porto seguro contra as dúvidas do dia-a-dia, e, sobretudo, por confiarem na minha capacidade talvez mais até do que eu confio.

A Luiz Etrusco e Andréa, pela amizade e pela sempre enriquecedora convivência no DCC e na vida, e por aperfeiçoarem o profissional que sou, muito obrigado.

Aos professores Antônio Alfredo Loureiro, Diógenes Cecílio da Silva Jr., José Marcos Nogueira, José Monteiro da Mata, Linnyer Beatrys Ruiz, Mário Fernando Montenegro Campos, Newton José Vieira, Nivio Ziviani, Rodolfo Ferreira Resende e Sérgio Vale Campos toda minha gratidão pelos grandes ensinamentos.

Aos colegas da Engetron, da Scanitec, da pós, do LECOM e da i-VISION: Ana, Anício, Exo, Valdeci, Leonardo Alves, Fabrício Vivas, Paulinho, Flip, Flop, Leandro, Thiago, Makish, Vitorino, Luís Humberto, Mateus, Marcus, Claudiney, Martín, Felipe, Nakamura, Pinheiro, Daniel, Romanelli, Cristiana, Gurvan, Alla, Ruiter, Pio, Sica, Adriana, Fabiano, Thierso, Álvaro, Menoti, Guillermo, Otaviano, Márcia, Flávio Miana, Leandro Indrusiak e Guilherme, por me estenderem a mão sempre que necessitei, o meu muito obrigado.

João Camilo, Guilherme Correa, Guilherme Alves, Ronan, Alexandre, Ricardo Falcão, Fabiano, Carla Nacif, Cynthia Nacif, tios, primos e demais familiares, fico feliz por vocês fazerem parte, há muito, desse caminho que eu continuo a percorrer.

A todos os funcionários do DCC, que fazem parte da infra-estrutura desta instituição de sucesso, muito obrigado por tudo.

Ao CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico - pelo suporte financeiro, sob os processos PNM #830107/2002-9 e SensorNet #552111/2002-3.

Sumário

1	Introdução	1
2	Técnicas de verificação de circuitos integrados	4
2.1	Verificação funcional	5
2.1.1	Verificação caixa-preta	5
2.1.2	Verificação caixa-branca	7
2.2	Verificação formal	9
2.2.1	Verificação de modelos	10
2.2.2	Verificação de equivalência	13
2.3	Verificação Semi-Formal	14
2.4	Especificação de propriedade	16
2.4.1	OVL - Open Verification Library	16
2.4.2	Property Specification Language - PSL	19
2.4.3	SystemVerilog	22
3	Biblioteca de asserções para depuração em tempo de execução	24
3.1	Trabalhos relacionados	24
3.2	OVL - Versão modificada para síntese	25
3.3	Validação da versão modificada para síntese da OVL	29
4	Arquitetura de verificação em tempo de execução	32
4.1	Processador de Asserções	35
4.1.1	Processador de asserções usado para informar sobre condições de erro	36
4.1.2	Processador de asserções usado para recuperação de condições de erro	37
5	Resultados	39
5.1	Resultados de síntese das asserções da versão modificada da OVL	39
5.2	Estudos de caso	42
5.2.1	Microcontrolador 8051	42
5.2.2	Microprocessador μ RISC-II	43
5.2.3	Microprocessador Super Hitachi-2	45

5.3	Injeção de falhas	46
6	Conclusões e Trabalhos Futuros	50
A	Open Verification Library	57
A.1	assert_always	57
A.2	assert_always_on_edge	58
A.3	assert_change	58
A.4	assert_cycle_sequence	59
A.5	assert_decrement	59
A.6	assert_delta	59
A.7	assert_even_parity	60
A.8	assert_fifo_index	60
A.9	assert_frame	60
A.10	assert_handshake	61
A.11	assert_implication	61
A.12	assert_increment	62
A.13	assert_never	62
A.14	assert_next	62
A.15	assert_no_overflow	62
A.16	assert_no_transition	63
A.17	assert_assert_no_underflow	63
A.18	assert_odd_parity	63
A.19	assert_one_cold	63
A.20	assert_one_hot	64
A.21	assert_propositon	64
A.22	assert_quiescent_state	64
A.23	assert_range	64
A.24	assert_time	65
A.25	assert_transition	65
A.26	assert_unchange	65
A.27	assert_width	66
A.28	assert_win_change	66
A.29	assert_win_unchange	66
A.30	assert_window	66
A.31	assert_zero_one_hot	67

Lista de Tabelas

1.1	Lista de erros das principais famílias de processadores Intel	2
2.1	Tabela verdade para o DUV da Figura 2.3	6
2.2	Propriedades necessárias ao funcionamento correto do exemplo do controlador de sinal de trânsito	9
2.3	Asserções que compõem a biblioteca OVL	17
2.4	Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando asserções da OVL	18
2.5	Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando PSL	21
2.6	Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando SystemVerilog	23
3.1	Descrição dos sinais da Figura 3.1	26
4.1	Listagem da ordem de encadeamento para a Figura 4.1	33
4.2	Listagem da ordem de encadeamento para a Figura 4.2	34
5.1	Asserções cujas áreas são sensíveis apenas às expressões externas de teste .	40
5.2	Asserções cujas áreas são sensíveis ao sinal width	40
5.3	Asserções cujas áreas são sensíveis aos sinais num_cks, min_cks e max_cks .	41
5.4	Asserções cujas áreas são sensíveis aos sinais width e num_cks	41
5.5	Asserções cujas áreas são sensíveis a sinais presentes apenas nelas próprias	42
5.6	Asserções inseridas no microprocessador 8051	43
5.7	Resultados de síntese para o microprocessador 8051	43
5.8	Asserções inseridas no microprocessador μ RISC-II	44
5.9	Resultados de síntese para o microprocessador μ RISC-II	45
5.10	Asserções inseridas no microprocessador Super Hitachi-2	46
5.11	Resultados de síntese para o microprocessador Super Hitachi-2	46
A.1	Descrição dos principais sinais em comum da OVL	57

Lista de Figuras

2.1	Ciclo de desenvolvimento de um circuito integrado	4
2.2	Abordagem tradicional de verificação	5
2.3	Exemplo de circuito submetido ao processo de verificação caixa-preta . . .	6
2.4	Exemplo de circuito submetido ao processo de verificação caixa-branca . .	7
2.5	Estados alcançáveis através de um ponto fixo	11
2.6	Exemplo de um diagrama de decisão binário (BDD)	12
2.7	Verificação de equivalência utilizada para garantir que não existem erros na ferramenta de síntese	14
2.8	Erros identificados utilizando verificação funcional (simulação)	15
2.9	Erros identificados utilizando verificação formal	15
2.10	Erros identificados utilizando verificação semi-formal	16
3.1	(a) Interface típica de uma asserção OVL; (b) Interface de uma asserção OVL modificada para permitir encadeamento.	25
3.2	Diagrama de tempo com o comportamento da arquitetura proposta.	26
3.3	Trecho de código adicionado às asserções da OVL	27
3.4	Circuito sintetizado da asserção <code>assert_never</code> (versão original da OVL) . . .	28
3.5	Circuito sintetizado da asserção <code>assert_never</code> (versão modificada da OVL) .	28
3.6	Arquitetura de verificação utilizada para a validação do <code>assert_always</code> . . .	30
3.7	Trecho de código em Verilog usado para a validação do <code>assert_always</code> . . .	30
3.8	Comportamento de alguns sinais durante a simulação apresentado pelo vi- sualizador de formas de onda Gtkwave	31
4.1	Parte de um projeto hierárquico de um microcontrolador 8051	33
4.2	Parte de um projeto hierárquico de um bloco de comunicação IIC	34
4.3	Diagrama de blocos do funcionamento da ferramenta XRoach	35
4.4	Interface gráfica da ferramenta XRoach	36
4.5	Arquitetura proposta para o processador de asserções	36
4.6	Pseudo-código em Verilog HDL de um processador de asserções que informa sobre condição de erro	37
4.7	Pseudo-código em Verilog HDL de um processador de asserções que recupera de condições de erro	38
5.1	Diagrama de blocos do processador de asserções monitorando o μ RISC-II .	47

5.2 Diagrama de blocos das asserções inseridas no μ RISC-II 47

5.3 Resultado da simulação do μ RISC-II com uma falha na asserção A8 . . . 48

5.4 Resultado da simulação do μ RISC-II com uma falha na asserção A5 . . . 48

Capítulo 1

Introdução

O problema de verificação de circuitos integrados é historicamente complexo. É um problema de ordem $2^{(n+m)}$ onde n é o número de entradas e m o número de estados do circuito integrado. Este número de elementos cresce obedecendo à Lei de Moore, ou seja, duplica a cada 18 meses. Uma boa aproximação para o crescimento do esforço de verificação é o fato de que a duplicação do número de portas duplica o trabalho por ciclo de clock, e a complexidade extra no mínimo duplica o número de ciclos necessários à obtenção da cobertura desejada [MBC⁺00]. Assim, em uma previsão otimista, pode-se dizer que o esforço de verificação deve dobrar a cada 18 meses.

O mercado de desenvolvimento de circuitos integrados é extremamente competitivo, e não é possível que o tempo gasto em verificação dobre a cada 18 meses. Nos últimos 20 anos, a verificação tem sido feita em níveis cada vez mais altos de abstração: inicialmente em nível de porta lógica, passando pelo nível de transferência de registradores (do inglês RTL - *Register Transfer Level*), chegando hoje em nível de sistema. Ferramentas de verificação formal estão limitadas a complexidades menores se comparadas a simulação/emulação, mas fornecem uma cobertura muito maior.

Atualmente, a etapa de verificação é, sem dúvida, o gargalo no ciclo de desenvolvimento de um circuito integrado. A etapa de verificação é responsável por aproximadamente 50% a 80% do tempo total de desenvolvimento do circuito integrado [FKL04, Dre04a, BBN⁺04]. Mesmo utilizando-se tanto tempo, muitas vezes não é possível atingir o nível de cobertura desejado. Por exemplo, para simular apenas dois minutos de execução de um Pentium IV são necessários 200 bilhões de ciclos [Ben01], considerando-se que este processador está sendo executado a 1 GHz. Vários métodos têm sido propostos como alternativas aos métodos tradicionais de simulação [McM03]. Estes métodos têm maior eficácia se utilizados de maneira combinada, mas, mesmo utilizando diferentes métodos de verificação, é impossível garantir que não existe erro algum no circuito integrado.

A combinação dos fatores de aumento de complexidade e diminuição de tempo de desenvolvimento leva a situações indesejáveis tais como a comercialização de circuitos integrados contendo erros que se manifestam apenas depois da venda. O exemplo clássico

é o erro de divisão de ponto flutuante do processador Pentium da Intel [Bei95], o qual utiliza o algoritmo de divisão proposto em [Atk68] que depende dos valores de uma tabela com aproximadamente 2000 entradas. No caso da implementação no processador Pentium, cinco entradas desta tabela foram deixadas em branco. Este erro manifestava-se apenas em combinações muito específicas de divisor e dividendo, causando perda de precisão no quarto e no décimo-segundo dígitos do quociente. Apesar de não afetar diretamente a maioria dos usuários, a Intel viu-se obrigada a trocar todos os processadores defeituosos gratuitamente, tendo para tanto um custo aproximado de 475 milhões de dólares.

Apesar de a descoberta de erros em circuitos integrados já comercializados ser comum, nem sempre os fabricantes admitiam e publicavam tais erros. Isto mudou quando a MIPS Technologies Inc. começou a publicar sua lista de erros [MIP94]. Atualmente, a maioria dos fabricantes publica periodicamente uma lista de erratas acerca de seus processadores [AH99, CMH00]. A Tabela 1.1 apresenta os dados publicados nas listas de errata das principais famílias de processadores da Intel [Int99, Int02, Int03, Int04]. Em [AMD03a, AMD03c, AMD03b, AMD04, Fre04] podem ser encontradas erratas de outros grandes fabricantes de microprocessadores.

Tabela 1.1: Lista de erros das principais famílias de processadores Intel

Processador	Corrigido	A ser corrigido	Sem planos de correção	Total
Pentium®	98	0	43	141
Pentium® II	23	6	66	95
Pentium® III	28	2	58	88
Pentium® IV	50	8	34	92

O objetivo deste trabalho é propor nova metodologia para verificação de circuitos integrados nos estágios de pós-venda. Esta metodologia baseia-se em uma técnica largamente utilizada nos dias de hoje: a verificação baseada em asserções [FKL04]. Asserções podem ser definidas como monitores incluídos no circuito integrado ao longo do processo de desenvolvimento. Estes monitores devem garantir que as propriedades especificadas estejam presentes na implementação, auxiliando as ferramentas de verificação e/ou simuladores. Neste trabalho é proposta uma arquitetura que permite a inserção definitiva de asserções em um circuito integrado. Tal arquitetura permite que as asserções sejam habilitadas/desabilitadas e gerencia informações relativas a condições de erro ocorridas.

A arquitetura proposta neste trabalho pode ser utilizada de duas maneiras distintas. A primeira delas é durante a fase de emulação do circuito integrado. Neste caso, o circuito integrado pode ser emulado em situações reais, a uma velocidade impossível de ser alcançada durante o processo de simulação. As informações obtidas durante este processo podem ser muito úteis para a identificação de erros antes da comercialização do circuito integrado. A segunda aplicação da arquitetura proposta é o monitoramento remoto de um circuito

integrado. Isto pode ser feito com a adição de um circuito de comunicação, de forma que o circuito integrado possa informar condições de erro remotamente. Em se tratando de um circuito integrado em arquitetura reconfigurável (FPGA), o erro seria identificado e corrigido, e a FPGA seria novamente gravada. Esta possibilidade seria muito interessante em situações em que o custo de se ter acesso físico ao circuito integrado é muito alto como, por exemplo, na indústria espacial.

Este trabalho está organizado da seguinte forma: O Capítulo 2 faz uma revisão do estado da arte das técnicas de verificação; o Capítulo 3 trata dos trabalhos relacionados e apresenta a versão modificada para síntese de uma biblioteca de asserções. O Capítulo 4 apresenta uma ferramenta de roteamento automático das asserções e o processador de asserções, responsável pelo gerenciamento das asserções incluídas no circuito integrado. O Capítulo 5 apresenta os resultados e, finalmente, o Capítulo 6 trata de conclusões e trabalhos futuros.

Capítulo 2

Técnicas de verificação de circuitos integrados

O objetivo deste capítulo é apresentar as técnicas tradicionais de verificação de circuitos integrados. A Figura 2.1 ilustra as etapas de desenvolvimento de um circuito integrado. A etapa 1 consiste na especificação do funcionamento do circuito integrado em alto nível. Na etapa 2, esta especificação é implementada em nível de transferência de registradores (RTL, do inglês *Register Transfer Level*) através de uma linguagem de descrição de hardware, normalmente Verilog HDL [IEE01b] ou VHDL [IEE93] (do inglês *Very High Speed Integrated Circuit Hardware Description Language*). O processo de verificação acontece entre as etapas 1 e 2. Utilizando as técnicas descritas no decorrer deste capítulo, o comportamento do código RTL é confrontado com o comportamento descrito na especificação. Na etapa 3, a implementação em nível RTL é sintetizada por uma ferramenta que irá gerar *netlists*, as quais serão utilizadas na etapa 4, que é o processo de fabricação final do circuito integrado em silício. O processo de teste de um circuito integrado tem o objetivo de encontrar erros durante o processo de manufatura, através da comparação do comportamento das *netlists* com o circuito integrado (etapas 3 e 4) [Ber00]. Neste trabalho serão tratadas apenas questões relativas ao problema de verificação.

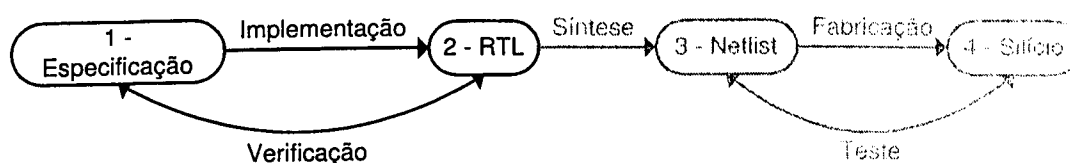


Figura 2.1: Ciclo de desenvolvimento de um circuito integrado

2.1 Verificação funcional

Um caso de teste (do inglês *testbench*) pode ser definido como um conjunto de estímulos aplicados às entradas de um dispositivo sob verificação (do inglês *Device Under Verification* - DUV), cujo comportamento é observado através das saídas do mesmo.

Dois conceitos são bastante importantes em verificação de circuitos digitais: controlabilidade e observabilidade [Ber00]. O conceito de controlabilidade refere-se à habilidade de estimular uma linha específica de código ou estrutura do projeto. Em um caso de teste tem-se baixa controlabilidade das estruturas internas do projeto. O conceito de observabilidade, por sua vez, refere-se à habilidade de visualizar os efeitos de um estímulo a linhas de código específicas ou estruturas internas. Pode-se concluir que um caso de teste possui observabilidade limitada, pois tem acesso apenas às interfaces de saída do dispositivo ou modelo. As estruturas internas não são acessíveis aos casos de teste.

2.1.1 Verificação caixa-preta

Tradicionalmente, a verificação de consistência entre implementação e especificação é feita através da abordagem caixa-preta, ilustrada na Figura 2.2 [Ber00]. Nesta abordagem, um caso de teste é criado e aplicado ao DUV. As saídas do DUV são, então, comparadas com um modelo de referência.

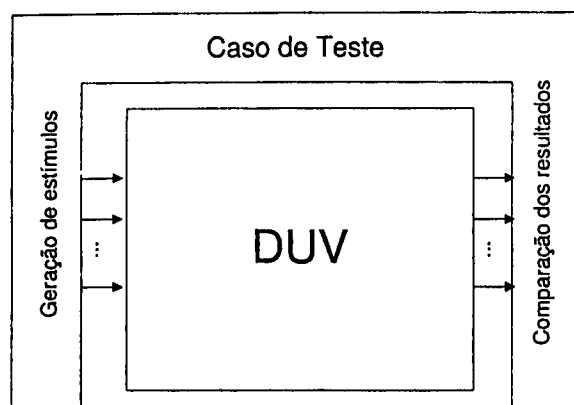


Figura 2.2: Abordagem tradicional de verificação

Atualmente, os casos de teste têm se tornado cada vez mais complexos, e normalmente são construídos através de uma linguagem de descrição de hardware combinando as seguintes características:

- Geração automática do vetor de testes;
- Validação do comportamento observado na saída;

- Análise de cobertura.

A Figura 2.3 apresenta uma parte combinacional de um circuito sequencial, sendo submetida à verificação caixa-preta. Este circuito é composto por duas portas lógicas E, (Y e Z) e uma porta lógica OU (W). As conexões destas portas lógicas estão representadas pelos pontos WZ e YZ. O comportamento esperado deste circuito é descrito na Tabela 2.1. De acordo com a abordagem de verificação caixa-preta, a controlabilidade do DUV limita-se à aplicação de estímulos às entradas *a*, *b* e *c*, e a observabilidade limita-se à comparação da saída *d* com o comportamento esperado.

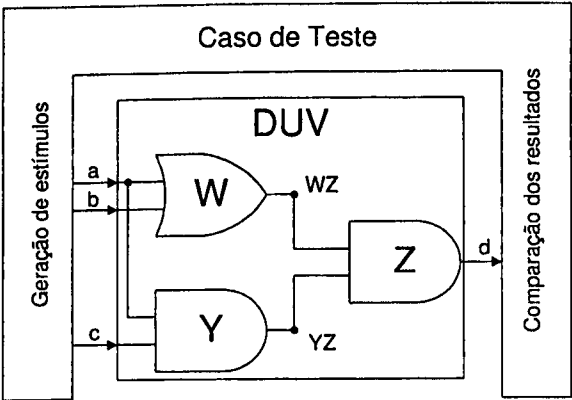


Figura 2.3: Exemplo de circuito submetido ao processo de verificação caixa-preta

Tabela 2.1: Tabela verdade para o DUV da Figura 2.3

Entrada a	Entrada b	Entrada c	Saída d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

A grande desvantagem da abordagem caixa-preta é a ausência de visibilidade das estruturas internas do DUV. Um comportamento interno incorreto pode não se propagar até

a interface de saída. Neste caso, apesar de o comportamento incorreto existir, ele não será identificado. É possível que para uma outra sequência de entrada este mesmo erro seja observável, mas a verificação de todas as propriedades de um DUV utilizando abordagem caixa-preta é inviável, especialmente quando a complexidade do DUV é grande. Para ilustrar esta limitação da abordagem de verificação caixa-preta, suponha que nível lógico do ponto WZ da Figura 2.3 seja sempre igual a zero. Este erro nunca será observável pela interface de saída d .

2.1.2 Verificação caixa-branca

A verificação caixa-branca complementa a verificação caixa-preta tradicional através da adição de conhecimento do funcionamento interno do DUV. Uma maneira de se implementar a verificação caixa-branca é utilizar asserções, as quais podem ser definidas como monitores que garantem que a especificação do projeto foi corretamente implementada. Estes monitores são instanciados pelo projetista durante a implementação. Uma abordagem caixa-branca para o exemplo da Figura 2.3 é apresentado na Figura 2.4. Neste exemplo uma asserção foi inserida para monitorar os pontos internos WZ e YZ . O objetivo desta asserção é garantir que WZ e YZ nunca serão verdadeiros ao mesmo tempo. Caso esta situação ocorra, um erro terá sido encontrado. Isto equivale a afirmar que a expressão $WZ + YZ > 1$ nunca será verdadeira.

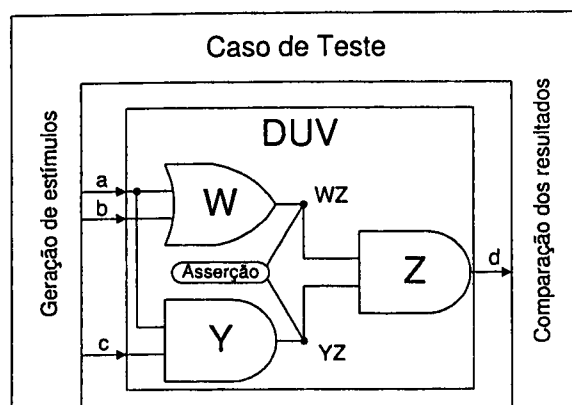


Figura 2.4: Exemplo de circuito submetido ao processo de verificação caixa-branca

Nos últimos anos, técnicas de verificação baseadas em asserções têm sido largamente utilizadas. Exemplos de empresas que vêm utilizando a verificação baseada em asserções são [FKL04]:

- Cisco Systems, Inc.;
- Digital Equipment Corporation;

- Hewlett-Packard Company;
- IBM Corporation;
- Intel Corporation;
- LSI Logic Corporation;
- Motorola, Inc;
- Silicon Graphics, Inc.

A popularidade da verificação baseada em asserções deve-se aos bons resultados alcançados em projeto de circuitos integrados reais. Alguns relatos de estatísticas de erros encontrados são:

- 34% de todos os erros do projeto do processador DEC Alpha 21164[KN96] foram encontrados por asserções;
- 17% de todos os erros do projeto do processador Cyrix M3(p1) foram encontrados por asserções[FKL04];
- 25% de todos os erros do projeto do processador DEC Alpha 21264 foram encontrados por asserções[TQB⁺98];
- 25% de todos os erros do projeto do processador Cyrix M3(p2) foram encontrados por asserções[FKL04];
- 85% de erros em vários projetos da HP foram encontrados por asserções [FC01];

Como vantagens do uso de verificação caixa-branca baseada em asserções pode-se citar:

- Maior observabilidade. O erro não precisa se propagar até a interface de saída para ser observado.
- Redução do tempo de depuração. Uma consequência direta do aumento da observabilidade é a capacidade de identificar o erro quando e onde ele ocorre, reduzindo consideravelmente o tempo de depuração do circuito integrado.
- Melhor integração através da verificação de uso correto de um bloco (do inglês *core*). Atualmente, prática comum no desenvolvimento de circuitos integrados são tanto a reutilização de blocos quanto a utilização de blocos de propriedade intelectual (IP, do inglês *intellectual property*) fornecidos por terceiros. Através da inserção de asserções nas interfaces destes módulos, pode-se garantir que os mesmos serão utilizados conforme especificações. Em outras palavras, as asserções garantem que eles não serão submetidos a uma condição imprevista.

- Melhoria de comunicação através de documentação. As asserções inseridas em um projeto podem ser vistas como uma documentação de como cada unidade deve se comportar. Neste caso, asserções podem ser encaradas como "comentários executáveis" que, caso não sejam respeitados, irão manifestar-se durante o processo de verificação.

Para ilustrar como asserções são utilizadas em um projeto real, será apresentado o exemplo clássico do controlador de sinal de trânsito [MC79]. Este exemplo constitui-se de um controlador de sinal para a interseção de duas estradas, sendo uma com quatro pistas e outra, estrada secundária. A estrada com quatro pistas é bastante movimentada, de modo que o controlador deve maximizar o tempo em que este sinal está verde. Este controlador possui um temporizador com duas saídas: os sinais *curto* e *longo*. A luz verde deve ficar acesa por, pelo menos, o intervalo de tempo *longo*. No fim deste período, caso haja um carro esperando na estrada secundária, o sinal da estrada de quatro pistas deve mudar e permanecer amarelo por um intervalo de tempo *curto* e, então, mudar para vermelho. O sinal deve permanecer vermelho até que todos os carros cruzem a estrada de quatro pistas, ou até que se passe o intervalo de tempo *longo*.

Para a verificação deste circuito, o projetista deve definir as propriedades que devem ser verificadas. Para que este circuito funcione adequadamente, as condições apresentadas na Tabela 2.2 devem ser satisfeitas.

Tabela 2.2: Propriedades necessárias ao funcionamento correto do exemplo do controlador de sinal de trânsito

Item	Descrição da propriedade
1	Em nenhum momento os dois sinais podem ficar verdes simultaneamente.
2	O valor <i>curto</i> do temporizador sempre deve ser menor do que o valor <i>longo</i> .
3	Não havendo carros e estando o sinal verde na entrada secundária, este sinal deve mudar para amarelo.
4	Um carro na entrada secundária não pode esperar para sempre.

Na Seção 2.4 serão apresentados exemplos de como especificar estas propriedades usando diferentes linguagens.

2.2 Verificação formal

Técnicas de verificação formal consistem em provar que uma implementação satisfaz sua especificação através de métodos matemáticos formais [Gup92]. A verificação é feita através de propriedades genéricas do projeto, e não mediante a simulação de conjuntos específicos

de entradas [McF93]. Utilizando-se métodos formais, pode-se provar que para todas as possíveis entradas de um projeto, uma propriedade P é satisfeita. Este conceito é análogo à diferença entre empregar leis da física e realizar experimentos. A maioria das técnicas de verificação formal utiliza diagramas de decisão binários (BDDs, do inglês *Binary Decision Diagrams*) [Bry86] ou resolvidores para o problema da satisfabilidade (do inglês, *SAT Solvers*) [BCCZ99, BCC⁺99]. As duas principais técnicas de verificação formal são: verificação de equivalência (do inglês *equivalence checking*) e verificação de modelos (do inglês *model checking*).

2.2.1 Verificação de modelos

O conceito de verificação de modelos surgiu da evolução da verificação por prova matemática de teoremas. Para o caso de sistemas concorrentes com estados finitos, a prova matemática formal não é necessária, podendo ser substituída por uma abordagem baseada em modelamento, a qual indicará se o sistema equivale ao que foi especificado em lógica temporal. Uma grande vantagem desta abordagem é o fato de a resposta ser dada automaticamente, sem a necessidade de qualquer interação por parte do usuário. Em [CES86] é apresentada uma implementação de verificação de modelos que utiliza estruturas finitas de Kripke [Kri63] para representar o grafo de estados do sistema concorrente e um algoritmo capaz de determinar se o sistema descrito no modelo satisfaz uma fórmula matemática específica. Este algoritmo era similar ao utilizado em otimização de código por compiladores.

Estruturas de Kripke modelam o comportamento de um circuito integrado utilizando um grafo, no qual um nó representa um estado, e uma aresta representa transições entre estados. Considerando que uma estrutura de Kripke $M = (S, S_0, R, L)$ é composta por:

- S é um conjunto finito de estados;
- S_0 é um conjunto de estados iniciais em que $S_0 \subseteq S$;
- $S \subseteq s \times S$ é um relação de transição em que, para cada estado $s \in S$, existe um estado $s' \in S$, em que a transição de estados $(s, s') \in R$;
- $L : S \longrightarrow 2^{AP}$, em que L é uma função que rotula cada estado com um conjunto de proposições atômicas, as quais são verdadeiras em um determinado estado.

Com o modelo construído, é necessário escrever as equações que representam o comportamento do circuito, ou seja, as transições entre os estados. Estas equações podem ser escritas em lógica temporal por desvio (do inglês *branching-time temporal logic*) ou em lógica temporal linear (do inglês *linear-time temporal logic*). Um exemplo de linguagem lógica temporal por desvio é a CTL (do inglês *Computational Tree Logic*) [CE82, BAMP83]. Os operadores temporais desta linguagem permitem descrever, a partir de um estado, todos os possíveis caminhos. Um exemplo de uma linguagem lógica temporal linear é a LTL

(do inglês *Linear-time temporal logic*) [Pnu77]. Através dos operadores temporais desta linguagem é possível informar os eventos em um caminho simples do grafo.

O passo final é o algoritmo automático de provas. Considerando um modelo formal de um projeto descrito em uma estrutura de Kripke $M = (S, S_0, R, L)$, e a fórmula lógico temporal f expressando um comportamento desejado do projeto, para se provar que o sistema está correto deve-se encontrar o conjunto de todos os estados em S que satisfaçam f :

$$\{s \in S \mid M, s \models f\}$$

em que $s \models f$ significa que o comportamento representado pela fórmula temporal f é válida para o estado s . Observe que o modelo formal satisfaz a especificação se, e somente se, todos os estados iniciais ($\forall s_i \in S_0$) pertencem ao conjunto de todos os estados que satisfazem f ($\{s \in S \mid M, s \models f\}$). A Figura 2.5 ilustra graficamente, em alto nível, um algoritmo de prova utilizado para encontrar o conjunto de estados em S que satisfaçam f .

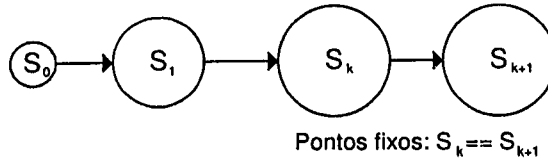


Figura 2.5: Estados alcançáveis através de um ponto fixo

O algoritmo de prova ilustrado começa com um conjunto de estados iniciais S_0 , conforme a Figura 2.5. Utilizando a relação de transição R , são calculados todos os estados alcançáveis a partir de S_0 em um ciclo de *clock*. No exemplo, o novo conjunto de estados alcançáveis é S_1 . Conjuntos de estados alcançáveis são gerados até que não exista mais qualquer estado alcançável. Este é o ponto fixo em que $S_{k+1} == S_k$.

Infelizmente, a implementação explícita de estruturas de Kripke é viável para representar apenas algumas dezenas de milhares de estados. Através da utilização de BDDs para representar o grafo de estados, foi possível a utilização de verificação de modelos em sistemas com 10^{20} estados [BCMD90], tendo, assim, a verificação de modelos aplicação direta na verificação de circuitos integrados. A representação através de BDDs consegue capturar as regularidades de estados determinadas pela lógica do caminho de dados. O algoritmo é baseado no processamento de pontos de funções de uns conjuntos de estados para outros conjuntos de estados (transformadores de predicados). Pode-se expressar os conjuntos de estados e as relações de transição através de BDDs. Por este motivo, a construção de todo o diagrama de estados do circuito não é necessária.

Um BDD é uma grafo direcionado acíclico. A Figura 2.6 apresenta um exemplo de um BDD representando a função booleana $f(a, b, c, d)$. A seguinte regra pode ser usada para mostrar que $f(1, 0, 1, 1) = 1$: trace um caminho da raiz do grafo até a folha e, em todo nó

escolha um desvio através do valor da variável correspondente. Esta regra pode ser usada para determinar a função representada pelo BDD. Restrições foram adicionadas à forma dos BDDs para que cada função tenha um único BDD. Uma destas restrições é o fato de que as variáveis da equação booleana devem estar ordenadas. A ordenação de variáveis da Figura 2.6 é $a < b < c < d$. A alteração na ordem das variáveis pode ter um grande impacto no tamanho do BDD necessário para representar uma função. Pode-se verificar em tempo constante se dois BDDs representam a mesma função. Maiores detalhes acerca do uso de BDDs em verificação de modelos podem ser encontrados em [CGP00, Kro00].

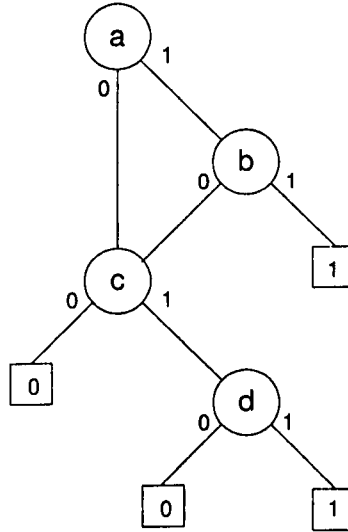


Figura 2.6: Exemplo de um diagrama de decisão binário (BDD)

Apesar de a utilização de BDDs em verificação de modelos ter representado um grande avanço, esta abordagem possui uma grande desvantagem: a importância da seleção da ordem correta das variáveis do BDD. A geração de uma ordem que resulte em BDDs pequenos é demorada e necessita de intervenção manual. Para muitas instâncias não existe uma ordem de variáveis eficiente em termos de espaço ocupado.

Nos últimos anos, a técnica conhecida como verificação de modelos com limite (do inglês *bounded model checking* - BMC) vem sendo muito utilizada [BCCZ99, BCC⁺99]. Nesta técnica, somente caminhos limitados a um tamanho k são considerados. Portanto, só serão encontrados erros e contra-exemplos de tamanho k . Dada uma especificação em lógica temporal e uma máquina de estados finitos, pode-se construir uma fórmula proposicional que será satisfeita se, e somente se, existir um contra-exemplo de tamanho k . Na prática, o tamanho máximo do contra-exemplo é incrementado, e após um certo número de iterações pode-se concluir que não existe contra-exemplo. Neste processo são utilizados algoritmos tradicionais de resolução do problema da satisfabilidade [DP60, G. 89]. Como principais vantagens de BMC, pode-se citar:

- Contra-exemplos são rapidamente encontrados. Esta é uma característica interessante pois encontrar contra-exemplos é o aspecto mais importante de verificação de modelos;
- Os contra-exemplos encontrados são de tamanho mínimo. Através desta característica o usuário consegue entender o contra-exemplo mais facilmente.
- Utiliza muito menos espaço que BDDs;
- A ordem das variáveis não precisa ser manualmente selecionada, além de não existir a demorada fase de reordenação dinâmica de variáveis.

Vários outros trabalhos recentes tratam de utilização de BDDs e de resolvers SAT para resolver o problema de verificação de modelos [KP03, McM03, SAV03, PICW04].

2.2.2 Verificação de equivalência

A verificação de equivalência tem o objetivo de garantir que duas descrições de circuitos são equivalentes. Estas descrições podem estar em diferentes níveis de abstração. Esta técnica de verificação é muito utilizada para testar ferramentas de síntese, ou seja, verificar se o circuito descrito em nível de transferência de registradores é equivalente ao circuito gerado pela ferramenta de síntese em nível de portas lógicas [SKKK04], conforme ilustra a Figura 2.7.

O problema de verificação de equivalência é conhecidamente co-NP completo [Mat96] e, portanto, não há esperanças de que seja encontrada uma solução eficiente para qualquer instância do problema. Os principais passos de um verificador de equivalência são [DH02, Dre04b]:

1. Traduzir os projetos para um formato interno;
2. Estabelecer a correspondência entre dois projetos na fase de casamento;
3. Provar equivalência ou não equivalência;
4. Em caso de não equivalência, um contra-exemplo é gerado e a fase de depuração se inicia.

Os algoritmos mais eficientes para a resolução da verificação de equivalência utilizam representações canônicas de diagramas booleanos, normalmente BDDs. Desta forma, os dois circuitos a serem comparados podem ser convertidos para esta forma canônica e, então, comparados. A grande desvantagem dos BDDs é a complexidade exponencial de espaço: se o BDD fica muito grande, o seu armazenamento e a sua manipulação começam a se tornar muito caros. Vários métodos têm sido propostos para diminuir o espaço ocupado por BDDs [KK97, CQ04].

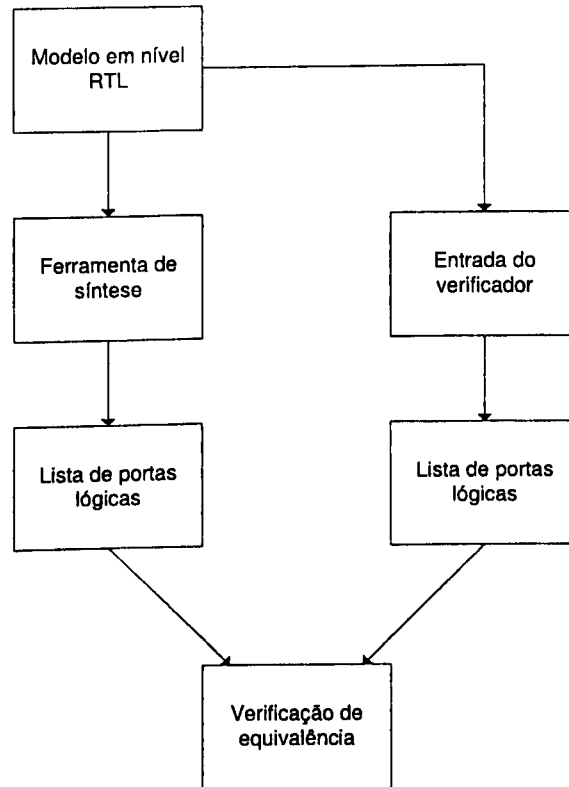


Figura 2.7: Verificação de equivalência utilizada para garantir que não existem erros na ferramenta de síntese

2.3 Verificação Semi-Formal

Uma abordagem intermediária entre verificação formal e verificação por simulação é a verificação semi-formal. Nas Figuras 2.8, 2.9 e 2.10 o espaço multi-dimensional de estados alcançados pelos processos de verificação está sendo representado de forma comprimida. A Figura 2.8 apresenta em alto nível um processo de verificação por simulação. Durante o processo de simulação, vetores de teste são aplicados às interfaces de entrada do circuito integrado. Estes estímulos são responsáveis pela mudança nos estados internos do circuito integrado até que um erro seja encontrado ou até que a simulação termine. Na Figura 2.8 o erro *E1* foi encontrado durante o processo de simulação, mas o erro *E2* não foi localizado, pois os estímulos gerados não foram suficientes para alcançar este estado. Esta é a principal limitação da verificação por simulação: é impossível aplicar todas as possibilidades de entradas do circuito.

A Figura 2.9 ilustra um processo de verificação formal. Nesta técnica o circuito integrado é modelado e todos os seus possíveis estados são exercitados através dos mecanismos

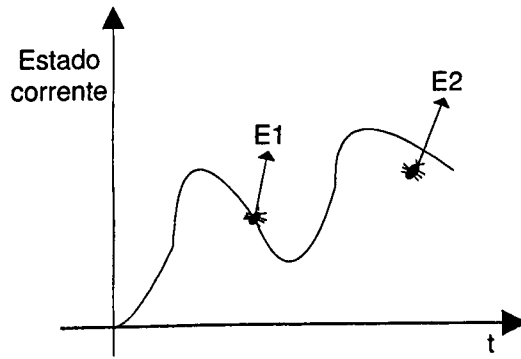


Figura 2.8: Erros identificados utilizando verificação funcional (simulação)

descritos na Seção 2.2.1. Observe que na Figura 2.9 os erros $E1$ e $E2$ foram encontrados. Infelizmente, devido a explosão de estados, esta abordagem não pode ser utilizada em circuitos integrados complexos.

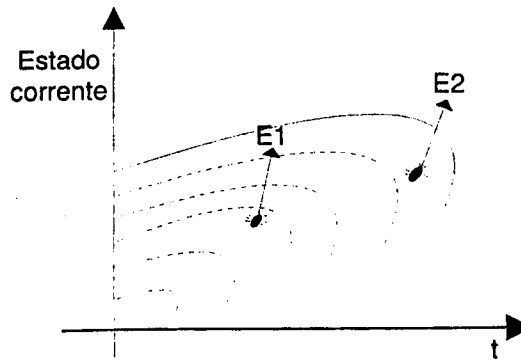


Figura 2.9: Erros identificados utilizando verificação formal

A Figura 2.10 apresenta a abordagem semi-formal de verificação. Nesta abordagem para determinados estados alcançados através de simulação é realizada verificação formal até que ocorra uma explosão de estados. Conseqüentemente, a cobertura é maior do que na simulação convencional. O grande problema é que não é possível saber o quanto esta cobertura é maior. No exemplo da Figura 2.10 os erros $E1$ e $E2$ foram identificados. Este tipo de abordagem, em que duas técnicas de verificação se complementam, é muito utilizada na indústria [GRS⁺03, Dil98, Lit04].

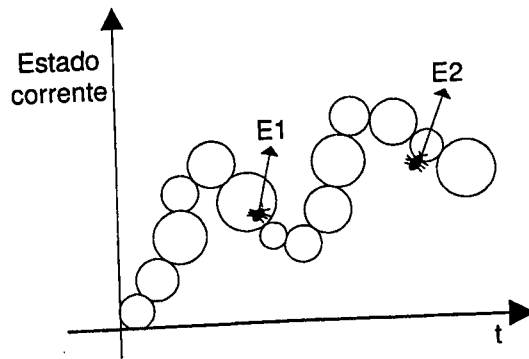


Figura 2.10: Erros identificados utilizando verificação semi-formal

2.4 Especificação de propriedade

Informalmente, *propriedade* pode ser definida como o comportamento geral que um projeto deve apresentar. Formalmente, pode-se definir *propriedade* como [FKL04]:

Um conjunto de relações lógico temporais entre expressões booleanas, expressões sequenciais e outras propriedades que agregadas representam o comportamento do projeto.

A composição destas *propriedades* pode ser dividida em quatro níveis:

- O nível booleano, representado através de expressões booleanas (por exemplo expressões em Verilog ou VHDL);
- O nível temporal, que descreve as relações entre as expressões booleanas através do tempo;
- O nível de modelamento, que fornece meios para modelar comportamentos complexos das entradas e saídas do projeto;
- O nível de verificação, que descreve como utilizar uma propriedade durante o processo de verificação.

Propriedades são largamente utilizadas tanto em verificação funcional por simulação quanto em verificação formal. Nas próximas subseções serão apresentadas três linguagens de especificação de propriedade: a OVL (do inglês *Open Verification Library*)[Acc03b], a PSL (do inglês *Property Specification Language*)[Acc04a] e a SystemVerilog[Acc04b].

2.4.1 OVL - Open Verification Library

A OVL [Acc03b] é uma biblioteca de asserções de código aberto com versões em Verilog e VHDL, e permite a especificação de propriedades em nível RTL através da instanciação

de monitores. A biblioteca de asserções OVL é composta por 31 asserções, conforme é apresentado na Tabela 2.3. Maiores detalhes sobre a funcionalidade de cada uma das asserções podem ser encontrados no Apêndice A.

Tabela 2.3: Asserções que compõem a biblioteca OVL

assert_always	assert_increment	assert_quiescent_state
assert_always_on_edge	assert_never	assert_range
assert_change	assert_next	assert_time
assert_cycle_sequence	assert_no_overflow	assert_transition
assert_decrement	assert_no_transition	assert_unchange
assert_delta	assert_no_underflow	assert_width
assert_even_parity	assert_odd_parity	assert_win_change
assert_fifo_index	assert_one_cold	assert_win_unchange
assert_frame	assert_one_hot	assert_window
assert_handshake	assert_proposition	assert_zero_one_hot
assert_implication		

A OVL provê as seguintes funcionalidades:

- Método de geração de relatórios unificado e sistemático que pode ser personalizado;
- Mecanismo comum para habilitação/desabilitação das asserções durante o processo de verificação
- Método sistemático de filtragem do relatório de uma violação específica de uma asserção através da limitação do número máximo de asserções disparadas.

Finalmente, como a OVL é escrita em nível RTL das linguagens Verilog/VHDL, não é necessário um pré-processamento e, muito menos, algum suporte específico de fabricante de ferramenta.

A OVL possui as seguintes macros globais de controle:

- ‘ASSERT_GLOBAL_RESET: Responsável pelo reset de todas as asserções do projeto. Sobrepuê o reset individual de cada asserção;
- ‘ASSERT_MAX_REPORT_ERROR: Todas as asserções possuem um registrador interno chamado `error_count`. A cada erro reportado, o valor de `error_count` é incrementado. Se o valor de `error_count` for maior que o desta macro global, a asserção não irá mais reportar erros;

- 'ASSERT_ON: Habilita as asserções;
- 'ASSERT_INIT_MSG: Mensagem inicial da asserção

Durante a simulação, uma asserção OVL irá reportar um erro caso alguma propriedade tenha sido violada. O erro é normalmente reportado na borda de subida do clock da asserção. A função `ovl_error()` é chamada para reportar esta violação.

A Tabela 2.3 demonstra a utilização deste método de especificação de propriedades, instanciando asserções OVL no controlador de sinais de trânsito apresentado na Seção 2.1. As asserções da Tabela 2.4 estão escritas em Verilog. Mais exemplos de utilização da versão VHDL da OVL podem ser encontrados em [Acc03b]. Para o melhor entendimento do exemplo, considere a seguinte descrição dos sinais RTL utilizados:

- `AmbosVerdes`, `CurtoMenorLongo`, `SemCarro`, `NaoEsperaParaSempre`: Nomes das instâncias das asserções;
- `LuzSec`: Luz ativa (Verde, Vermelha ou Amarela) no sinal da estrada secundária;
- `Luz4p`: Luz ativa (Verde, Vermelha ou Amarela) no sinal da estrada com quatro pistas;
- `Curto` e `Longo`: Intervalos de tempo disponíveis no temporizador do controlador;
- `TemCarro`: Indica se há algum carro na estrada secundária

Tabela 2.4: Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando asserções da OVL

Item	Asserções da OVL
1	<code>assert_never AmbosVerdes (clk, reset_n, (LuzSec == Verde && Luz4p == Verde));</code>
2	<code>assert_always CurtoMenorLongo (clk, reset_n, (Curto < Longo));</code>
3	<code>assert_time #(0,1) SemCarro (clk, reset_n, (LuzSec == Verde && ~TemCarro), (LuzSec == Amarelo));</code>
4	<code>assert_change #(0,2,32) NaoEsperaParaSempre (clk, reset_n, longo && Luz4p == Verde && TemCarro, Luz4p)</code>

A primeira asserção deve garantir que os dois sinais nunca estarão abertos simultaneamente, através de um asserção do tipo "nunca", que garante que uma expressão de teste nunca será verdadeira. A segunda asserção deve garantir que o tempo *curto* é sempre

menor que o tempo *longo*. Para tanto, é utilizada uma asserção do tipo "sempre", que garante que uma expressão sempre será verdadeira. A terceira asserção garante que, caso o sinal secundário esteja verde, e não haja carro algum, o sinal deve mudar seu estado para amarelo em uma unidade de tempo. A asserção usada neste caso é do tipo "tempo". A última asserção deve garantir que o carro na estrada secundária não ficará esperando para sempre. É usada uma asserção do tipo "mudança" para garantir que após o tempo máximo de 32 ciclos, o estado do sinal da estrada com quatro vias mudará.

2.4.2 Property Specification Language - PSL

A linguagem de especificação de propriedade (PSL) é uma linguagem formal criada pela Accellera [Acc04a] e é compatível com qualquer linguagem de descrição de hardware. A PSL é baseada na linguagem de especificação de propriedades Sugar [For00], desenvolvida pela IBM. Uma especificação de propriedade escrita em PSL é, ao mesmo tempo, fácil de entender e matematicamente precisa. Estas características tornam a PSL ideal tanto para especificação como para documentação. A PSL pode ser usada em ferramentas de simulação e/ou verificação formal.

Ao contrário das asserções OVL, que são usadas durante a implementação RTL, a linguagem PSL pode ser usada tanto na implementação quanto na etapa de especificação. Todas as propriedades da linguagem PSL podem ser verificadas em ferramentas formais, mas nem todas as propriedades PSL podem ser verificadas em simulação.

No nível booleano, a linguagem PSL faz referências a sinais e variáveis da linguagem de descrição de hardware que está sendo utilizada. No nível temporal, seqüências de condições booleanas que ocorrem sucessivamente podem ser definidas através de SEREs (do inglês *Sequential Extended Regular Expressions*). Seqüências e SEREs podem ser construídas das seguintes formas:

- b : Uma expressão booleana é uma SERE em sua forma mais simples;
- $\{SERE\}$: Uma seqüência construída por uma SERE;
- $SERE; SERE$: Uma SERE construída através da concatenação de duas SERES;
- $\{seqüência - seqüência\}$: Uma seqüência descrevendo alternativas;
- $\{seqüência \& seqüência\}$: Uma seqüência descrevendo duas seqüências paralelas que ocorrem ao mesmo tempo.

A PSL possui também o operador de repetição ($[]$), que descreve a concatenação repetida de uma mesma SERE. Por exemplo, dada uma SERE r e uma expressão booleana b :

- $r[*m:n]$: Uma seqüência de n ocorrências contínuas de r ;
- $b[=m:n]$: Qualquer seqüência contendo n ocorrências de b ;

- $b[->m:n]$: Qualquer seqüência que termine na n -ésima ocorrência de b ;
- $r[*]$: Zero ou mais ocorrências: $r[*0:inf]$;
- $r[+]$: Pelo menos uma ocorrência simples: $r ; r[*]$.

Além de todos os operadores padrão LTL, a linguagem PSL possui outros operadores. Por exemplo, considere as fórmulas temporais $f, f1, f2$:

- $!f$: f não é válida;
- $f1 \ \& \ f2$: $f1$ e $f2$ são válidas;
- $f1 \ | \ f2$: $f1$ ou $f2$ são válidas;
- $f1 \ -> \ f2$: $f1$ implica $f2$;
- $f1 \ <-> \ f2$: $f1 \ -> \ f2$ e $f2 \ -> \ f1$;
- **always** f : f é válida em todos os ciclos - $G \ f$;
- **never** f : f não é válida em nenhum ciclo - $G \ !f$;
- **next** f : f é válida no próximo ciclo, se houver - $X \ f$;
- **next** $!f$: f é válida no próximo ciclo - $X \ !f$;
- $f1 \ \text{until} \ f2$: $f1$ é válida até que $f2$ seja válida, se houver - $f1 \ W \ f2$;
- $f1 \ \text{until} \ !f2$: $f1$ é válida até que $f2$ seja eventualmente válida - $[f1 \ U \ f2]$;
- $f1 \ \text{before} \ f2$: $f1$ é válida antes que $f2$ seja válida;
- **within**($r1, b$) $r2$: $r2$ ocorre após $r1$ e antes de b ;
- **eventually** $!f$: f é válida em algum ciclo futuro - $F \ f$.

A linguagem PSL também suporta operadores que criam propriedades complexas através de SEREs:

- $\{r1\} \ |-> \ \{r2\}$: $r2$ inicia-se no último ciclo de $r1$;
- $\{r1\} \ |=> \ \{r2\}$: $r2$ inicia-se no primeiro ciclo após $r1$;
- $\{r\} \ (f)$: f é verdadeira no último ciclo de r .

O nível de verificação de PSL fornecem diretivas que indicam às ferramentas o que deve ser feito com as propriedades especificadas. As diretivas válidas são:

- assert;
- assume;
- assume_guarantee;
- restrict;
- restrict_guarantee;
- cover;
- fairness e strong fairness.

Em PSL, clocks podem ser especificados implicitamente através da seguinte declaração:

```
default clock = (posedge clk);
assert always !(en1 & en2);
```

Alternativamente, pode-se associar de forma explícita um clock com uma propriedade através da seguinte declaração:

```
assert always !(en1 & en2) @(posedge clk);
```

A Tabela 2.5 apresenta a especificação de propriedades utilizando PSL para o exemplo do controlador de sinais de trânsito da Seção 2.1. Maiores detalhes sobre a linguagem PSL podem ser encontrados em [Acc04a].

Tabela 2.5: Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando PSL

Item	Asserções PSL
1	<code>assert never ({LuzSec == Verde && Luz4p == Verde}) @(posedge clk);</code>
2	<code>assert always ({Curto < Longo}) @(posedge clk);</code>
3	<code>assert always ({LuzSec==Verde && ~TemCarro} => {LuzSec==Amarelo}) @(posedge clk);</code>
4	<code>assert always ({Longo && Luz4p==Verde && TemCarro, [*0:32]} => {Luz4p==Amarelo}) @(posedge clk);</code>

2.4.3 SystemVerilog

A linguagem SystemVerilog está sendo desenvolvida com o objetivo de prover níveis mais altos de abstração para a linguagem Verilog. Abaixo estão listadas as diversas versões da linguagem Verilog:

- Verilog 1.0 é o padrão IEEE 1394-1995 também conhecido como Verilog-1995;
- Verilog 2.0 é o padrão IEEE 1394-2001 também conhecido como Verilog 2001. Esta versão inclui diversas melhorias em relação ao Verilog-1995;
- Verilog 3.1 é a linguagem Verilog-2001 com a adição de várias extensões de abstração em alto nível, como camadas de verificação e integração com linguagem C.

As principais melhorias da linguagem SystemVerilog em relação a verificação são:

- Verificação de funcionalidade: Reusabilidade, tipos de dados e funções para geração de casos de teste;
- Sincronização: Mecanismos para criação dinâmica de processos, controle de processos e comunicação entre processos;
- Classes: Mecanismos orientados a objeto que fornecem capacidade de abstração e de encapsulamento;
- Memória dinâmica: Gerenciamento automático de memória;
- Funcionalidade baseada em ciclos: Domínios de clock e atributos baseados em ciclos que ajudam no desenvolvimento, manutenção e reusabilidade;
- Mecanismos de asserções com declarações de seqüências e propriedades.

Os principais operadores de SystemVerilog são:

- `s1 [* N:M]`: Repetição consecutiva de `s1` `N` ou entre os tempos `N` e `M`;
- `s1 [-> N:M]`: Repetição de `s1` (entre os tempos `N` e `M`) em ciclos não consecutivos, terminando em `s1`;
- `s1 [= N:M]`: Repetição de `s1` (entre os tempos `N` e `M`) em ciclos não consecutivos, talvez terminando em `s1`;
- `##[N:M]`: Atraso temporal. Concatenação de duas seqüências;
- `not p1`: Resultado inverso da avaliação da propriedade;
- `s1 and s2`: Ambas as propriedades ocorrem em um tempo qualquer;

- `s1 and p2`: Ambas as seqüências ocorrem em um tempo qualquer;
- `s1 or s2`: Qualquer seqüência ocorre;
- `p1 or p2`: Qualquer propriedade ocorre;
- `s1 intersect s2`: Ambas as seqüências ocorrem ao mesmo tempo;
- `if (expr) p1 else p2`: Baseado na avaliação de `expr`, avaliar a propriedade `p1` ou a propriedade `p2`;
- `b throughout s1`: A expressão booleana `b` deve ser verdadeira até a finalização da seqüência `s1`;
- `s1 within s2`: `s1` e `s2` devem ocorrer. A duração de `s1` deve ser menor ou igual a duração de `s2`;
- `s1.ended`: A seqüência `s1` termina neste tempo;
- `s1.matched`: A seqüência `s1` termina neste tempo (sincronizado por outro clock);
- `first_match(s1)`: Primeira ocorrência de uma seqüência;
- `s1 |-> p1`: Se `s1` ocorre, `p1` ocorre no mesmo ciclo;
- `s1 |=> p1`: Se `s1` ocorre, `p1` ocorre no próximo ciclo;

A Tabela 2.6 apresenta a especificação de propriedades, utilizando SystemVerilog para o exemplo do controlador de sinais de trânsito da Seção 2.1. Maiores detalhes sobre a linguagem SystemVerilog podem ser encontrados em [Acc04b].

Tabela 2.6: Especificação das propriedades do exemplo do controlador de sinais de trânsito utilizando SystemVerilog

Item	Asserções SystemVerilog
1	<code>assert property (@(posedge clk) not (LuzSec==Verde && Luz4p==Verde)) else \$error;</code>
2	<code>assert property (@(posedge clk) (Curto < Longo)) else \$error;)</code>
3	<code>assert property (@(posedge clk) (LuzSec==Verde && ~TemCarro) => (LuzSec==Amarelo)) else \$error;</code>
4	<code>assert property (@(posedge clk) (Longo && Luz4p==Verde && TemCarro) => ##[0:32] (Luz4p==Amarelo) else \$error;</code>

Capítulo 3

Biblioteca de asserções para depuração em tempo de execução

No Capítulo 2 foram apresentadas técnicas tradicionais de verificação de circuitos integrados. Ficou evidente que não existe uma metodologia eficaz para se garantir que um circuito integrado seja fabricado sem erros de projeto. Neste Capítulo será apresentada uma arquitetura para a inclusão de asserções no circuito integrado, de modo que o processo de verificação possa ser estendido à etapa pós-venda.

3.1 Trabalhos relacionados

Em [OH02] é proposta uma técnica de alto nível para especificação de monitores entre interfaces de blocos IP (do inglês *Intellectual Property*). O objetivo desta técnica é garantir que estes blocos IPs sempre sejam submetidos a entradas válidas e apresentem saídas corretas. Após a especificação dos monitores nas interfaces, os mesmos são traduzidos para linguagens de descrição de hardware Verilog/VHDL utilizando uma ferramenta protótipo. Infelizmente, as descrições dos monitores são apresentadas em um nível muito alto de abstração e são incluídas no processo de síntese, o que as torna difíceis de serem usadas em arquiteturas reconfiguráveis. Outra desvantagem desta técnica é a inexistência de ferramentas comerciais de verificação que permitam sua interação com os monitores e a descrição destes.

Outro trabalho relacionado [Dre03] propõe uma técnica para síntese de verificadores baseada na inclusão de flip-flops no circuito integrado. Por exemplo, considere uma propriedade em que, quando o sinal x se tornar 1, dois ciclos de clock depois, o sinal y deverá mudar seu valor para 0. Neste caso, são incluídos flip-flops suficientes para armazenar dois valores passados de x , sendo a saída destes flip-flops ligada em uma porta E juntamente com o sinal de y negado (pois ele deve ter o valor 0). Caso a saída da porta E seja 0, a propriedade terá sido violada. Em [GD04], são dados exemplos de como se implementar

esta técnica utilizando a linguagem SystemC. A abordagem apresentada nestes trabalhos é feita em muito baixo nível, pois os flip-flops devem ser instanciados pelo projetista. O ideal é que as propriedades sejam especificadas em um nível mais alto de abstração e automaticamente embutidas no processo de síntese.

3.2 OVL - Versão modificada para síntese

A versão em Verilog da biblioteca de asserções OVL foi modificada para permitir a inclusão no circuito integrado [NdPF+03]. A idéia básica é o encadeamento das asserções incorporadas ao circuito integrado em uma arquitetura similar ao padrão *boundary scan* [IEE01a]. Para que as asserções sejam encadeadas, foram adicionados pinos à interface tradicional da OVL (Figura 4.1 (a)), conforme ilustra a Figura 3.1 (b). A Tabela 3.1 apresenta a descrição dos sinais adicionados.

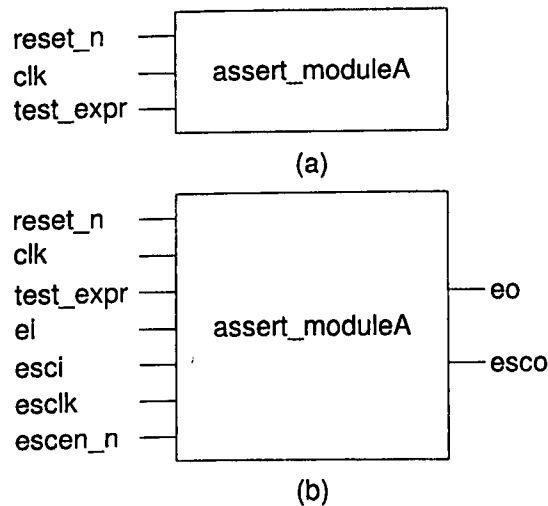


Figura 3.1: (a) Interface típica de uma asserção OVL; (b) Interface de uma asserção OVL modificada para permitir encadeamento.

A versão modificada para síntese da OVL foi estendida com os sinais *ei*, *esci*, *esclck*, *escen_n*, *eo* e *esco*. O sinal *eo*, saída de erro, é um circuito combinacional ativado quando qualquer uma das asserções presentes no circuito integrado reporta um erro. O sinal *escen_n* desabilita a avaliação da expressão de teste da asserção, de forma que a arquitetura de encadeamento possa ser exercitada. Os sinais *ei*, *esci*, *esco* e *esclck* são entrada de erro combinacional, entrada de erro sequencial, saída de erro sequencial e sinal de clock, respectivamente.

A Figura 3.2 apresenta o diagrama de tempo da ocorrência de erro em uma asserção incorporada ao circuito integrado. Nesta Figura, os sinais *eo* e *esco* referem-se à última

Tabela 3.1: Descrição dos sinais da Figura 3.1

Sinal	Descrição	Entrada/Saída
reset_n	Reset ativo em zero	Entrada
clk	Clock da asserção	Entrada
test_expr	Expressão de teste	Entrada
ei	Entrada de erro	Entrada
esci	Entrada de escaneamento de erro	Entrada
esclk	Clock do Erro	Entrada
escen_n	Habilitação do escaneamento de erro	Entrada
eo	Saída de erro	Saída
esco	Saída de escaneamento de erro	Saída

asserção encadeada no circuito integrado. Quando o sinal `eo` é ativado (em nível lógico 0), uma das n asserções presentes no circuito integrado acabou de reportar um erro. O sinal de habilitação de escaneamento, `escen_n`, é então ativado (também em nível lógico 0) e nenhuma asserção irá gerar novas falhas. A partir deste momento, a cada borda de subida do sinal `esclk`, o sinal de escaneamento de erro `esco` irá deslocar o seu valor para a asserção seguinte. Deste modo, após n ciclos de `esclk`, o sinal `esco` irá mudar para nível lógico zero, indicando que a n -ésima asserção falhou.

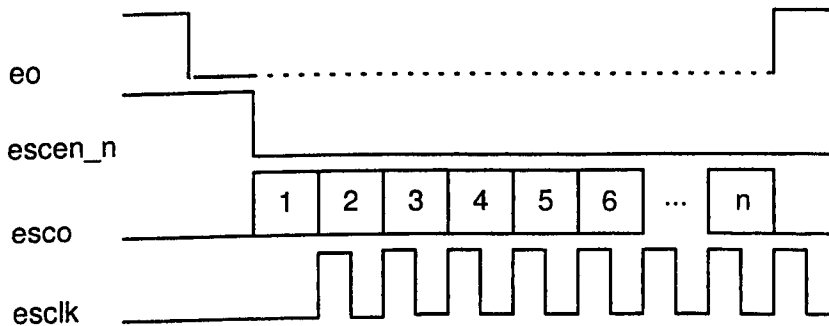


Figura 3.2: Diagrama de tempo com o comportamento da arquitetura proposta.

A Figura 3.3 apresenta o trecho de código em linguagem Verilog adicionado às asserções da biblioteca OVL. Nas linhas 01 e 02, observam-se as interfaces adicionais de Entrada/Saída. Na linha 03 é definido o registrador que armazena a condição de erro. Na versão original da OVL, quando ocorre um erro, a função `ovl_error` é chamada. Na versão

modificada da OVL, a chamada à função `ovl_error` é substituída por uma atribuição de 0 ao registrador `ovl_error`. Nas linhas 04 e 05, são declarados o registrador `esco`, e os fios `reset_scan` e `set_scan` que são utilizados no processo de escaneamento do sinal de erro. O sinal `eo` é uma operação booleana E dos sinais `ei` e `esco`, pois ambos são ativos em zero, conforme a linha 07. As linhas 08-18 definem o comportamento do sinal de escaneamento de erro `esco`. Quando o sinal `escen_n` está ativo, a asserção simplesmente copia o sinal `esci` para a saída `esco` a cada borda de subida de `esclk` (linha 16). Quando `escen_n` não está ativo, o valor do pino `esco` é o registrador `ovl_error`.

```

1  input  ei, esci, escen_n, esclk;
2  output eo, esco;
3  reg ovl_error;
4  reg esco;
5  wire reset_scan, set_scan;
6
7  assign eo = ei & esco;
8  assign reset_scan = ~(escen_n?ovl_error:1);
9  assign set_scan   = (escen_n?ovl_error:0);
10
11 always @(posedge esclk or posedge reset_scan or posedge set_scan)
12 begin
13     if(set_scan == 1)
14         esco <= 1;
15     else if (reset_scan == 1)
16         esco <= 0;
17     else
18         esco <= esci;
19 end

```

Figura 3.3: Trecho de código adicionado às asserções da OVL

A Figura 3.4 apresenta o circuito gerado através da síntese da asserção `assert_never`. É importante ressaltar que, para que uma ferramenta de síntese gere este circuito, é preciso fazer uma pequena alteração na asserção original da OVL. Esta alteração é a inclusão do pino de saída `result`. Caso contrário, a ferramenta irá simplificar todo o circuito. Na Figura 3.5, pode-se observar o circuito sintetizado da asserção `assert_never` modificada

para síntese/encadeamento. Para a representação deste circuito foram usados dois flip-flops D com clocks independentes e uma única saída.

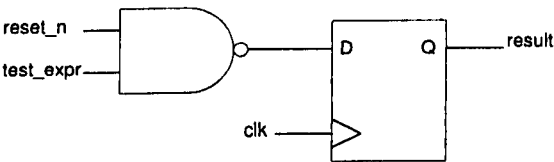


Figura 3.4: Circuito sintetizado da asserção `assert_never` (versão original da OVL)

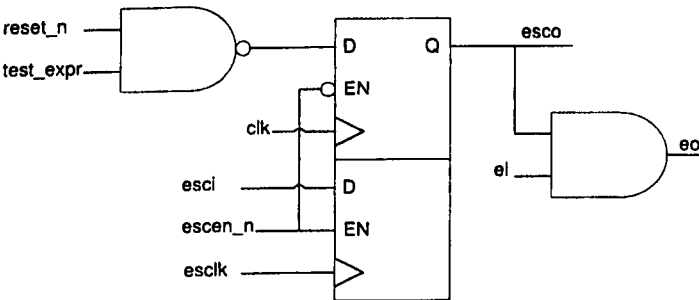


Figura 3.5: Circuito sintetizado da asserção `assert_never` (versão modificada da OVL)

Outras alterações realizadas na OVL original, que não afetam o processo de síntese são:

- Remoção da variável inteira (32 bits) `error_count`: As asserções modificadas para síntese/encadeamento não armazenam informação de quantas vezes falharam. Esta funcionalidade pode ser implementada no processador de asserções.
- Todas as propriedades PSL foram removidas, pois esta versão da biblioteca suporta apenas a linguagem Verilog.
- As funções do arquivo `ovl_task.h` (como por exemplo `ovl_error("")`) não são usadas. Portanto, este arquivo não é incluído.
- Mensagens de inicialização de asserções foram removidas.
- O pragma de ferramenta de síntese `"Synopsys translate on/off"` foi removido para que o circuito possa ser sintetizado.
- Inicializações do tipo `initial` não fazem sentido em ferramentas de síntese. Portanto, foram removidos.

- Algumas asserções possuem parâmetros que estão internamente definidos como inteiros de 32 bits. Para economia de área, foi criada uma macro que, com base no tamanho do parâmetro instanciado, define exatamente o tamanho necessário para armazená-lo.
- Algumas asserções como, por exemplo, `assert_handshake`, fornecem informações sobre a condição de erro ocorrida. No caso da versão modificada para síntese/encadeamento, esta informação não está disponível, sendo informada apenas a ocorrência da falha através da ativação dos sinais `eo` e `esco`.

3.3 Validação da versão modificada para síntese da OVL

Para garantir que a modificação na biblioteca de asserções não introduziu qualquer erro nas mesmas, é necessário verificá-las. A estratégia adotada para a validação é a verificação funcional caixa-preta. Através de um simulador, estímulos são gerados, exercitando as entradas do dispositivo sob teste (do inglês *Device Under Verification* - DUV). Esta técnica de verificação caixa-preta apresenta bons resultados em circuitos com baixa complexidade, como é o caso das asserções.

Um fator que precisa ser definido é como os estímulos são gerados. O ideal seria utilizar um teste exaustivo, em que todas as possibilidades de entrada fossem geradas, mas ter-se-ia um crescimento no tempo de simulação exponencial com o número de entradas do DUV. Uma solução muito adotada na indústria é a geração de estímulos aleatórios, os quais são aplicados simultaneamente ao DUV e a um modelo. As saídas dos dois módulos são então comparadas e, caso exista alguma incoerência de comportamento, um erro terá sido encontrado. A Figura 3.6 ilustra a arquitetura adotada para o exemplo da asserção `assert_always`. No arquivo `assert_always_test.v` são gerados os estímulos aleatórios. Neste mesmo arquivo estão instanciados o DUV, que é a versão modificada para síntese da OVL (`assert_always.v`), e o modelo de referência, que é a versão original da OVL (`assert_always_ref.v`). O comportamento das duas é comparado e, caso exista alguma incoerência, uma mensagem de erro será impressa.

A Figura 3.7 apresenta um trecho de código em Verilog do exemplo da validação da asserção `assert_always`. Nas linhas 1 – 4, tem-se uma função que gera números aleatórios no intervalo 0 e 15. Nas linhas 6 – 8, o DUV é instanciado de forma a reportar um erro caso o valor do contador seja maior que 9. Da mesma forma, o modelo de referência é instanciado nas linhas 10 – 12. Finalmente, na linha 14 é definido o sinal erro que é um *OU-EXCLUSIVO* das saídas das duas asserções, realizando a comparação entre elas.

Para a realização das simulações, foi utilizado o simulador da linguagem Verilog `cver` [Pra04]. Foi utilizado também o visualizador de formas de onda `Gtkwave` [Ton04]. A Figura 3.8 apresenta graficamente um trecho da simulação do código da Figura 3.7. A segunda forma de onda é o número gerado aleatoriamente dentro do intervalo 0 – 15. A penúltima

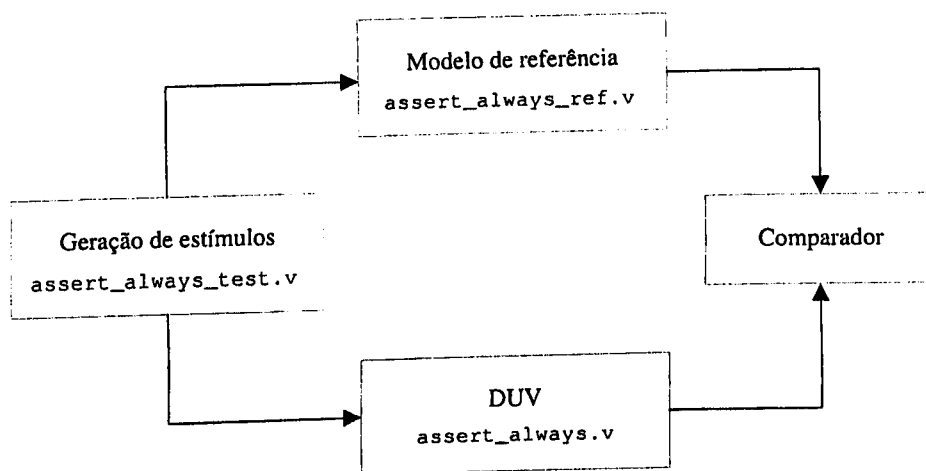


Figura 3.6: Arquitetura de verificação utilizada para a validação do assert_always

```

1  always @(posedge clk)
2  begin
3      count = $dist_uniform(seed,min,max);
4  end
5
6  assert_always #(32, 0, "")
7      modificada (clk, reset_n, (count >= 4'b0000) &&
8          (count <= 4'b1001), ei, esci, eo_ok, esco_ok, esclck, escen_n);
9
10 assert_always_ref #(64,0,"Erro ")
11     referencia (clk, reset_n, (count >= 4'b0000) &&
12         (count <= 4'b1001));
13
14 assign erro = (referencia.verificador ^ modificada.esco);

```

Figura 3.7: Trecho de código em Verilog usado para a validação do assert_always

e a última linhas são as saídas do DUV e do modelo de referência. Pode-se observar que, quando o valor do contador é maior que 10, as duas asserções indicam ocorrência de erro.

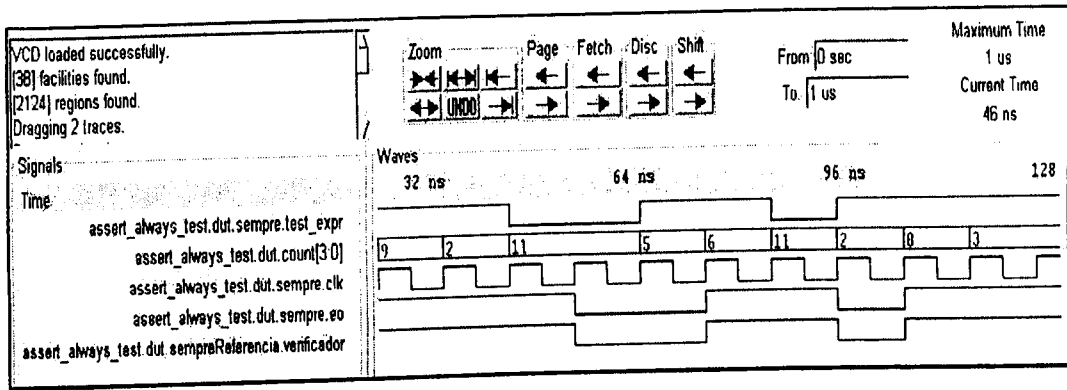


Figura 3.8: Comportamento de alguns sinais durante a simulação apresentado pelo visualizador de formas de onda Gtkwave

Este processo que foi descrito para a asserção `assert_always` foi repetido para as outras 30 asserções que compõem a OVL. Para cada tipo de asserção foi desenvolvido um caso de teste que conseguisse exercitar seu comportamento esperado. Cada tipo de asserção foi simulado durante 24 horas. Maiores detalhes podem ser encontrados em [OMN+04].

Capítulo 4

Arquitetura de verificação em tempo de execução

Este Capítulo apresenta os detalhes de como é realizado o encadeamento das asserções e do processador de asserções. Em se tratando de projetos hierárquicos sintetizáveis, cada módulo somente tem acesso às interfaces dos módulos de nível imediatamente inferior. Portanto, todos os módulos que possuem asserções devem ter suas interfaces alteradas com a adição dos sinais `ei`, `esci`, `esclk`, `escen_n`, `eo` e `esco`.

Para um projeto com n asserções, os sinais de saída `eo` e `esco` da asserção n devem ser conectados aos pinos de entrada `ei` e `esci` da asserção $n - 1$. Este processo deve se repetir até que os sinais `ei` e `esci` da asserção de número 1 sejam encadeados. Os sinais `eo` e `esco` desta asserção devem, então, ser ligados ao módulo de hierarquia mais alta.

A Figura 4.1 apresenta um exemplo de encadeamento de parte de um projeto de um microcontrolador 8051. O projeto original é representado pelos círculos brancos. As asserções, quatro no total, estão representadas por círculos cinza. No nível de hierarquia mais alto, o `alu_top`, está instanciada uma asserção tipo `assert_always(A4)`. No nível imediatamente inferior, o módulo `alu_divide`, estão instanciadas três asserções: `assert_always(A3)`, `assert_frame(A2)` e `assert_underflow(A1)`. Para a realização do encadeamento, os módulos `alu_top` e `alu_divide` têm adicionados às suas interfaces os sinais `ei`, `esci`, `esclk`, `escen_n`, `eo` e `esco`. Desta forma, os sinais `eo` e `esco` da asserção $A4$ podem ser ligados aos sinais `ei` e `esci` da asserção $A3$. Estes mesmos sinais são ligados da asserção $A3$ para a asserção $A2$ e da asserção $A2$ para a asserção $A1$. A Tabela 4.1 apresenta a ordem final de encadeamento das asserções.

A Figura 4.2 apresenta o encadeamento do bloco de comunicação I²C. O projeto original é composto pelos módulos `Master_top`, `Master_byte_ctrl` e `Master_bit_ctrl`. Ao módulo `Master_top` foram adicionadas três asserções: `assert_always(A5)`, `assert_never(A4)` e `assert_never(A3)`. Aos módulos `Master_byte_ctrl` e `Master_bit_ctrl` foi adicionada uma asserção `assert_one_hot(A2,A1)` em cada. O encadeamento final é apresentado na Tabela 4.2.

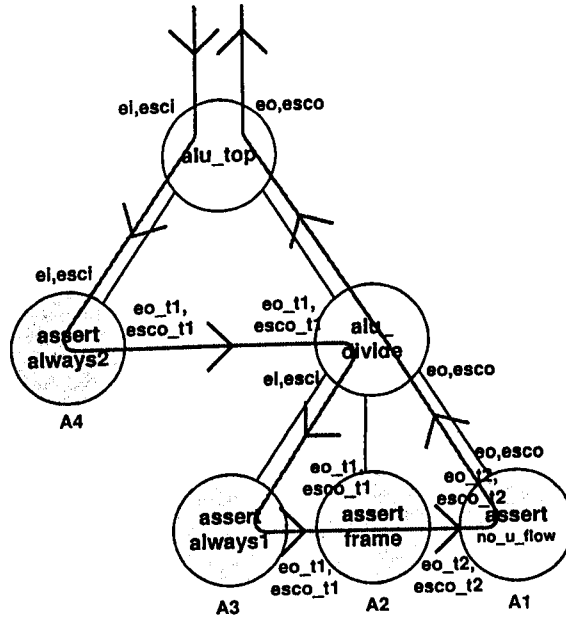


Figura 4.1: Parte de um projeto hierárquico de um microcontrolador 8051

Tabela 4.1: Listagem da ordem de encadeamento para a Figura 4.1

Número	Asserção
1	assert_no_underflow (A1)
2	assert_frame (A2)
3	assert_always (A3)
4	assert_always (A4)

Para o encadeamento automático de asserções, foi desenvolvida uma ferramenta protótipo [dONCF03] denominada XRoach, a qual é escrita em linguagem TK Perl. As entradas do XRoach são: um projeto hierárquico, com asserções originais da OVL instanciadas, e a versão modificada da OVL (Figura 4.3). O XRoach localiza todos os módulos do projeto que possuem asserções e incluem em suas interfaces os sinais `ei,esci`, `esclk`, `escen_n`, `eo` e `esco`. O próximo passo é o encadeamento das asserções localizadas. No caso das Figuras 4.1 e 4.2, o sinais (wires, em Verilog) `eo_ti` (onde *i* é o número de asserções por nível) são criados para realizar o encadeamento. As saídas da ferramenta XRoach são uma lista com a ordem em que as asserções foram encadeadas, e o novo projeto. Esta lista de ordem em que as asserções foram geradas é muito importante para o funcionamento do circuito que

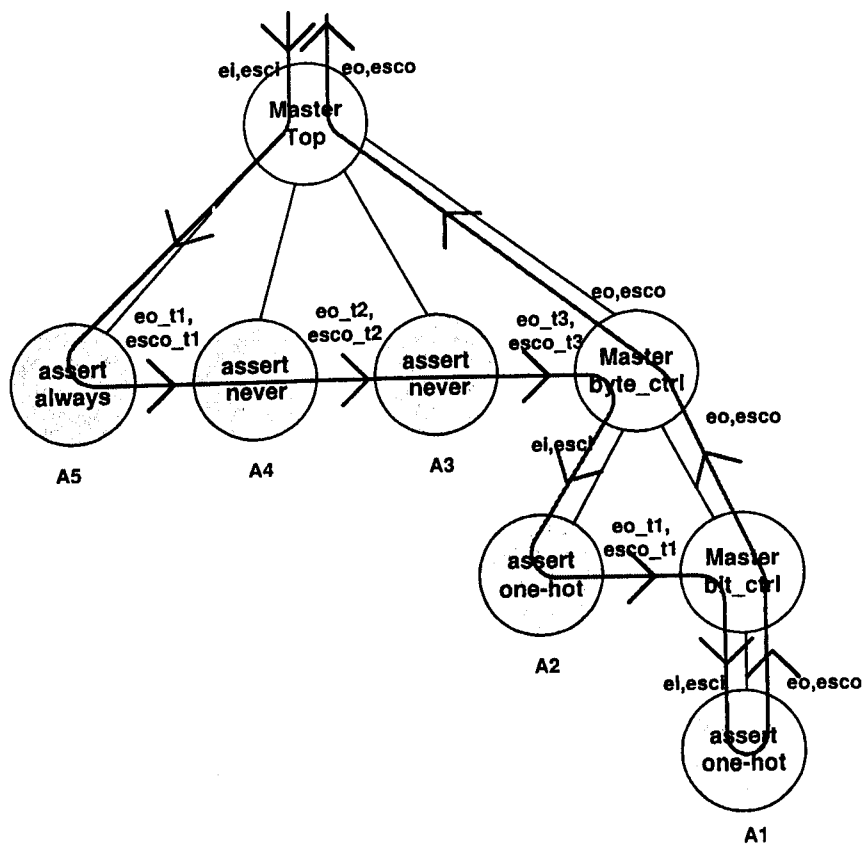


Figura 4.2: Parte de um projeto hierárquico de um bloco de comunicação IIC

Tabela 4.2: Listagem da ordem de encadeamento para a Figura 4.2

Número	Asserção
1	<code>assert_one-hot(A1)</code>
2	<code>assert_one-hot(A2)</code>
3	<code>assert_never(A3)</code>
4	<code>assert_never(A4)</code>
5	<code>assert_always(A5)</code>

gerencia as asserções: o processador de asserções.

Outra função da ferramenta XRoach é possibilitar a adição/remoção de asserções do

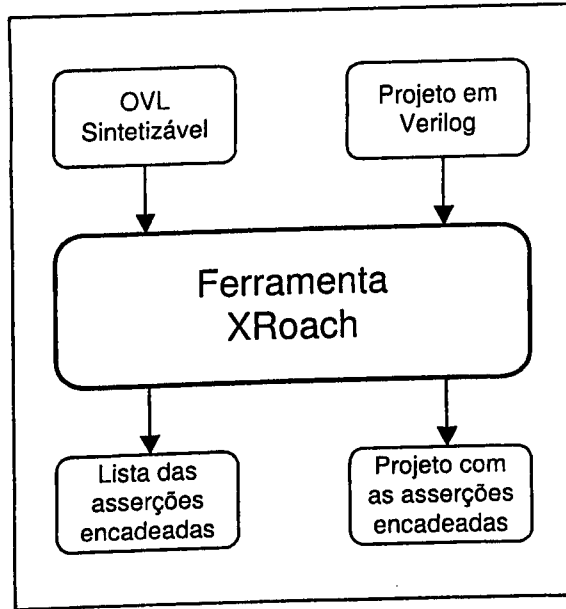


Figura 4.3: Diagrama de blocos do funcionamento da ferramenta XRoach

projeto, bem como definir severidades para tais asserções, através de uma interface gráfica. A Figura 4.4 apresenta a interface gráfica do XRoach sendo utilizada em um projeto de um microcontrolador 8051. A ferramenta XRoach permite, ainda, a geração da lista de ordem de encadeamento codificada em *one-hot* ou decimal.

4.1 Processador de Asserções

O processador de asserções é o circuito responsável pelo gerenciamento das asserções encadeadas. O processador de asserções gera os sinais *esclk*, *escen_n* baseado na leitura dos sinais *eo*, *esco*. A Figura 4.5 apresenta uma visão global da arquitetura proposta [NdPC⁺03, Nac04]. Podem ser observadas 10 asserções devidamente encadeadas através de seus pinos *ei*, *esci*, *eo* e *esco*. Os pinos *ei* e *esci* da última asserção (A10) estão ligados em nível lógico 1 (ausência de erros), e os pinos *eo* e *esco* da primeira asserção (A1) estão ligados no processador de asserções.

Um processador de asserções pode ser utilizado de duas maneiras distintas: informar sobre alguma asserção que falhou, ou realizar uma ação no próprio circuito integrado, de forma a se recuperar de uma situação de erro.

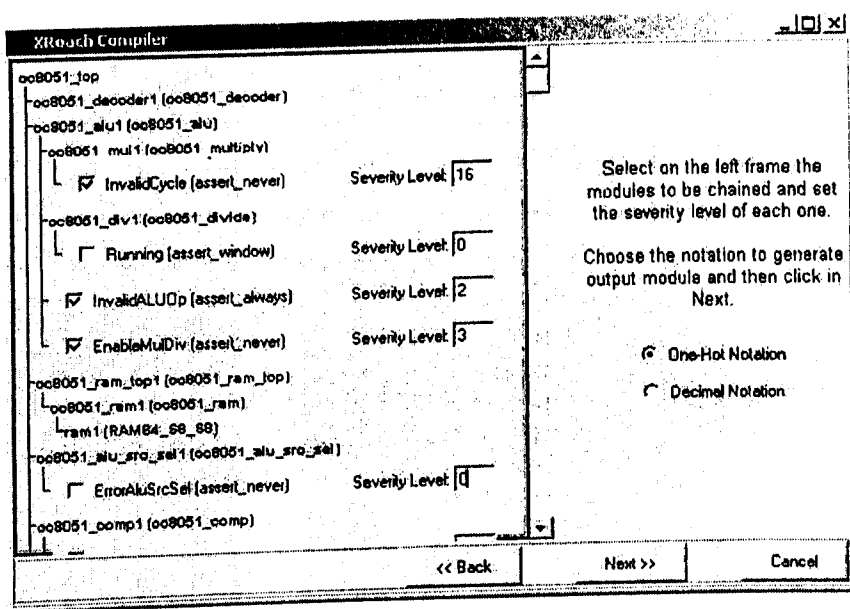


Figura 4.4: Interface gráfica da ferramenta XRoach

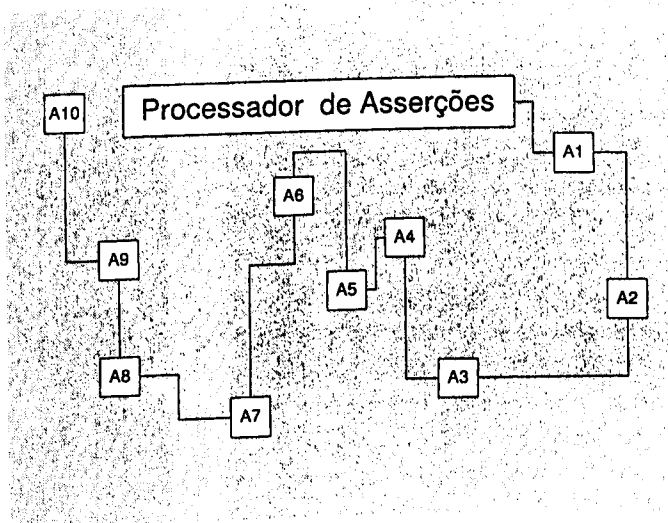


Figura 4.5: Arquitetura proposta para o processador de asserções

4.1.1 Processador de asserções usado para informar sobre condições de erro

A forma mais simples de informar sobre uma condição de erro é utilizando os pinos de E/S. A Figura 4.6 apresenta o pseudo-código de um processador de asserções que informa

a condição de erro através de pinos de E/S.

```
1 module ProcessadorAssercoes (...);  
2     output ['logNumeroAssercoes:0] NumeroAssercao;  
3     reg    ['logNumeroAssercoes:0] NumeroAssercao;  
4     // Detecção da condição de erro  
5     always@(posedge clk) begin  
6         if (Erro)  
7             begin  
8                 NumeroAssercao = NumeroAssercao + 1;  
9             end  
10    end  
11 endmodule
```

Figura 4.6: Pseudo-código em Verilog HDL de um processador de asserções que informa sobre condição de erro

Esta configuração do processador de asserções pode ser utilizada na fase de emulação do circuito integrado. Como foi discutido no Capítulo 2, é impossível simular um projeto aplicado-se todas as entradas exaustivamente. Neste caso, o circuito integrado estaria sendo emulado em situações reais, a uma velocidade inviável no processo de simulação. Nesta fase final de desenvolvimento, as informações sobre asserções que falharam podem ser muito úteis para a identificação de erros.

Outra aplicação para esta configuração de processador de asserções é o monitoramento remoto de um circuito integrado. Isto poderia ser feito com a adição de um circuito de comunicação, de forma que o processador de asserções pudesse informar condições de erro remotamente. Em se tratando de um circuito integrado em arquitetura reconfigurável (FPGA), o erro seria identificado e corrigido, e a FPGA seria novamente gravada. Esta possibilidade seria muito interessante em situações em que o custo de se ter acesso físico ao circuito integrado é muito alto como, por exemplo, na indústria espacial.

4.1.2 Processador de asserções usado para recuperação de condições de erro

O processador de asserções pode ser usado no desenvolvimento de circuitos integrados com capacidade de recuperação de condições de erro. Neste caso, as asserções seriam agrupadas por severidade, e cada severidade acionaria um pino do processador de asserções. A

Figura 5.3 apresenta o pseudo-código de um processador de asserções sendo utilizado para recuperar o circuito integrado de uma condição de erro.

```

1  module ProcessadorAssercoes (...);
2      output ['logNumeroAssercoes:0] Severidade;
3      reg    ['logNumeroAssercoes:0] Severidade;
4      reg    ['logNumeroAssercoes:0] NumeroAssercao;
5      // Detecção da condição de erro
6      always@(posedge clk) begin
7          if (Erro)
8              begin
9                  NumeroAssercao = NumeroAssercao + 1;
10             end
11         end
12         // Correção do erro
13         always@(posedge clk) begin
14             casex(NumeroAsserção)
15                 3'bxx1: Severidade = 3'b001 // Interromper a execução do CI
16                 3'bx1x: Severidade = 3'b010 // Reset de hardware
17                 3'b1xx: Severidade = 3'b100 // Interrupção
18             endmodule

```

Figura 4.7: Pseudo-código em Verilog HDL de um processador de asserções que recupera de condições de erro

Capítulo 5

Resultados

Neste Capítulo, serão apresentados os resultados de síntese das asserções, e do custo imposto pela adição das mesmas. Como estudo de caso, foram escolhidos três microprocessadores: o 8051[TS02], o μ RISC-II[Coe01] e o Super Hitachi-2 [Ait04].

5.1 Resultados de síntese das asserções da versão modificada da OVL

O objetivo desta Seção é apresentar uma referência de área ocupada por cada uma das asserções da OVL. Para tanto, as asserções foram agrupadas de forma que cada Tabela refira-se a asserções cujas áreas são sensíveis a um determinado grupo de sinais. Os resultados desta Seção foram obtidos utilizando a ferramenta de síntese Quartus II Web Edition 4.0, da Altera. A FPGA utilizada foi a EP1K100FC256-1, da família ACEX, e a síntese foi realizada com otimização para área. As métricas utilizadas na medição de área de cada asserção foram flip-flops e células lógicas ocupadas.

Na Tabela 5.1 são apresentadas asserções cujas áreas internas são constantes. No caso destas asserções, pode ocorrer uma variação de área ocupada devido à complexidade das expressões de teste: `test_expr`, `sampling_event`, `antecedent_expr`, `consequent_expr`, `start_event` e `end_event`.

A Tabela 5.2 apresenta as asserções cujas áreas são sensíveis principalmente ao sinal `width` o qual é utilizado para especificar a largura de sinais como `test_expr`, `state_state`, `end_state`, `state_expr` e `checkvalue`. Parâmetros como `value`, `min`, `max` pouco influenciam na área total utilizada pela asserção. Os resultados máximos e mínimos foram gerados com `width` igual a 1 e 32.

As asserções da Tabela 5.3 têm sua área sensível aos sinais `num_cks`, `min_cks` e `max_cks`. Os resultados máximos e mínimos foram gerados com `num_cks` igual a 1 e 32 para as asserções `assert_cycle_sequence`, `assert_next` e `assert_time`. No caso das asserções

Tabela 5.1: Asserções cujas áreas são sensíveis apenas às expressões externas de teste

Asserção	Flip-flops	LUTs
assert_always	2	5
assert_always_on_edge	2	5
assert_implication	2	5
assert_never	2	5
assert_propostion	1	4
assert_window	3	6

Tabela 5.2: Asserções cujas áreas são sensíveis ao sinal width

Asserção	Seq.(FF) (Min/Máx)	Comb. (LUTs) (Min/Máx)
assert_decrement	1 / 36	2 / 100
assert_delta	1 / 36	2 / 154
assert_even_parity	2 / 2	5 / 15
assert_increment	1 / 36	2 / 100
assert_no_overflow	3 / 3	6 / 29
assert_no_transition	5 / 67	11 / 126
assert_no_underflow	3 / 3	6 / 25
assert_odd_parity	2 / 2	5 / 15
assert_one_cold	2 / 2	5 / 64
assert_one_hot	2 / 2	5 / 64
assert_quiescent_state	3 / 3	7 / 24
assert_range	1 / 2	2 / 21
assert_win_change	5 / 36	12 / 60
assert_win_unchange	5 / 36	12 / 60
assert_transition	4 / 66	10 / 124
assert_zero_one_hot	1 / 2	2 / 55

assert_frame e assert_width, os sinais min_cks e max_cks foram alterados para os valores 1 e 32.

Tabela 5.3: Asserções cujas áreas são sensíveis aos sinais num_cks, min_cks e max_cks

Asserção	Seq.(FF) (Min/Máx)	Comb. (LUTs) (Min/Máx)
assert_cycle_sequence	2 / 498	3 / 508
assert_frame	2 / 10	5 / 31
assert_next	3 / 34	6 / 37
assert_time	3 / 8	5 / 19
assert_width	7 / 13	17 / 41

Na Tabela 5.4 observam-se as asserções sensíveis aos sinais width e num_cks. Para observar a influência destes sinais na área ocupada pelas asserções, os sinais foram variados da seguintes forma:

- A: width = 1, num_cks = 1;
- B: width = 1, num_cks = 32;
- C: width = 32, num_cks = 1;
- D: width = 32, num_cks = 32;

Pode-se observar que o sinal width tem um maior impacto na área ocupada pela asserção do que o sinal num_cks.

Tabela 5.4: Asserções cujas áreas são sensíveis aos sinais width e num_cks

Asserção	Seq.(FF) (Min/Máx)	Comb. (LUTs) (Min/Máx)
assert_change	5 / 10 / 36 / 41	12 / 27 / 60 / 75
assert_unchange	6 / 10 / 37 / 42	13 / 28 / 61 / 76

A Tabela 5.5 apresenta duas asserções mais específicas, cujas áreas variam em função de sinais presentes apenas nas mesmas. Para a asserção assert_fifo_index, os sinais push_width, pop_width e depth foram variados da seguinte forma:

- A: push_width, pop_width, depth = 8;
- B: push_width, pop_width, depth = 16;
- C: push_width, pop_width, depth = 32;

Para a asserção `assert_hand_shake`, os sinais `min_ack_cycle`, `max_ack_cycle`, `req_drop`, `deassert_count`, `max_ack` e `ack_lenght` foram variados da seguinte forma:

- A: `min_ack_cycle`, `max_ack_cycle`, `req_drop`, `ack_lenght` = 8;
- B: `min_ack_cycle`, `max_ack_cycle`, `req_drop`, `ack_lenght` = 16;
- C: `min_ack_cycle`, `max_ack_cycle`, `req_drop`, `ack_lenght` = 32;

Tabela 5.5: Asserções cujas áreas são sensíveis a sinais presentes apenas nelas próprias

Asserção	Seq.(FF) (Min/Máx)	Comb. (LUTs) (Min/Máx)
<code>assert_fifo_index</code>	10 / 10 / 10	114 / 171 / 204
<code>assert_hand_shake</code>	24 / 40 / 72	108 / 168 / 285

5.2 Estudos de caso

A adição das asserções nos microprocessadores foi feita com base na leitura e no entendimento da documentação disponibilizada pelo projetista. Apesar de o ideal ser a adição de asserções durante o processo de desenvolvimento, a adição de asserções em estágios mais avançados também é viável [FKL04].

5.2.1 Microcontrolador 8051

O processador 8051 é largamente utilizado nas mais diversas aplicações. Existem diversos fabricantes com diferentes configurações de periféricos e memórias. O bloco sintetizado possui dois temporizadores de 16 bits, quatro portas de E/S com oito bits cada, 4 Kbytes de memória de programa e 128 registradores de 1 byte. A memória de programa foi inferida pela ferramenta de síntese utilizando flip-flops da família da FPGA Virtex [Xil01]. Esta estrutura de memória é responsável por grande parte da área ocupada da FPGA. A Tabela 5.6 apresenta a descrição de algumas asserções inseridas no microcontrolador 8051. Foram adicionadas ao 8051 onze asserções no total.

A Tabela 5.7 apresenta a área e a velocidade do processador 8051 antes e após a inserção das asserções e do processador de asserções. Para a obtenção destes resultados, foi utilizada a ferramenta de síntese Free Web Pack 5.2i da Xilinx. A FPGA selecionada é a Virtex XCV 300 e a síntese foi realizada com alto esforço de otimização por área. O número de portas lógicas equivalentes aumentou de 3.19 % e a velocidade máxima do processador diminuiu 1.01 %. Neste caso, o custo associado a inserção das asserções é bastante pequeno, pois a

Tabela 5.6: Asserções inseridas no microprocessador 8051

Tipo de asserção	Funcionalidade
<code>assert_window</code>	Verifica a finalização de uma operação de divisão antes de uma nova habilitação
<code>assert_time</code>	Um sinal ACK de 4 ciclos deve acontecer após uma interrupção
<code>assert_over_flow</code>	Garante que não haverá estouro da pilha
<code>assert_always</code>	Garante que a ALU sempre receberá um <i>opcode</i> válido

maioria das asserções inseridas ocupa pouca área. Outro fator que também influenciou foi que as FPGAs Virtex utilizam células lógicas para inferir memória, e este processo consome bastante recursos.

Tabela 5.7: Resultados de síntese para o microprocessador 8051

Parâmetro	Estrutura Original	Estrutura Modificada	Custo
Flip-flops	838	943	12,53 %
LUTs	4.487	4.698	4,70 %
Portas lógicas equivalentes	68.141	70.313	3,19 %
Frequência máxima	12,99 MHz	12,86 MHz	1,01 %

5.2.2 Microprocessador μ RISC-II

O processador μ RISC-II é um processador de 16 bits com 8 registradores de 16 bits de uso geral. Seu conjunto de instruções é composto por 32 instruções. Instruções de 3 operandos em formato *Big endian*. A memória é endereçável a nível de palavra, ou seja, cada endereço de memória deve se referir a dois bytes. No total, o processador possui 64K endereços. A memória total do processador é de 128 kb. As instruções são executadas em um *pipeline* de 4 estágios:

- IF (do inglês *Instruction Fetch*): A próxima instrução a ser executada é buscada na memória e armazenada no registrador de instruções;
- ID (do inglês *Instruction Decode*): A instrução é decodificada e os operandos são buscados no banco de registradores;

- EX/MEM (do inglês *Execute and memory*): A instrução é executada e as condições dos resultados são marcadas para operações de ALU;
- WB (do inglês *Write Back*): Os resultados são escritos no banco de registradores.

Foram inseridas no microprocessador μ RISC-II 8 asserções, conforme ilustra a Tabela 5.8.

Tabela 5.8: Asserções inseridas no microprocessador μ RISC-II

Tipo de asserção	Funcionalidade
<code>assert_never</code>	Garante que nenhuma instrução de desvio escreve no banco de registradores, apenas a <i>jal</i> (<i>jump and link</i>).
<code>assert_never</code>	Escrita em memória, não pode ocorrer simultaneamente com a escrita no banco de registradores.
<code>assert_always</code>	Desvios incondicionais devem alterar o MUX do contador de programa.
<code>assert_change</code>	Instrução <i>jal</i> (<i>jump and link</i>) deve escrever no banco de registradores após 2 ciclos de clock.
<code>assert_delta</code>	Contador de programa não pode permanecer com mesmo valor.
<code>assert_change</code>	Desvios devem resetar o registrador de instruções.
<code>assert_no_overflow</code>	Verifica <i>overflow</i> no contador de programa.
<code>assert_proposition</code>	Verifica sinal <code>memtoreg</code> válido (0 a 2).

Os resultados apresentados na Tabela 5.9 foram obtidos utilizando a ferramenta de síntese Quartus II Web Edition 4.0, da Altera. A FPGA utilizada foi a EP1K100FC256-1, da família ACEX, e a síntese foi realizada com otimização para área. Apesar de o custo associado a inserção das asserções ter sido bem maior do que o exemplo anterior, isto pode ser facilmente explicado analisando as Tabelas da Seção 6.1. A asserção `assert_delta` apresenta um custo de 36 flip-flops e 154 células lógicas para monitorar um sinal de 32 bits. No caso específico da asserção instanciada no μ RISC-II, ela é responsável pelo monitoramento de um sinal de 16 bits e ocupa 84 células lógicas e 20 flip-flops. As Tabelas da Seção 6.1 se mostram uma boa referência para estimar a área ocupada por um determinado conjunto de asserções. O custo de velocidade associado a inserção das asserções se mostrou bem pequeno, da ordem de 2%.

Tabela 5.9: Resultados de síntese para o microprocessador μ RISC-II

Parâmetro	Estrutura Original	Estrutura Modificada	Custo
Flip-flops	180	227	26,11%
Células lógicas	658	821	24,77%
Frequência máxima	14,62 MHz	14,29 MHz	2,26%

5.2.3 Microprocessador Super Hitachi-2

O processador utilizado (Aquarius) é um clone do Processador SuperH-2 da Hitachi. Este processador tem um conjunto de instruções RISC e 16 registradores de 32 bits de propósito geral. Todas as instruções são de 16 bits. O processador SuperH-2 tem um *pipeline* de 5 estágios e a maioria das instruções é executada em 1 ciclo. Esta CPU possui também uma estrutura interna de 32 bits para funções de processamento avançadas como por exemplo multiplicação e acumulação. O Aquarius pode gerenciar requisições de interrupção como NMI (do inglês *Non maskable interrupt*) e IRQ (do inglês *Interrupt Request*) A prioridades das interrupções podem ser de 0 a 16.

Foram inseridas no microprocessador SuperH-2 8 asserções, conforme ilustra a Tabela 5.10.

Os resultados apresentados na Tabela 5.11 foram obtidos utilizando a ferramenta de síntese Quartus II Web Edition 4.0, da Altera. A FPGA utilizada foi a EP2S15F484C3, da família Stratix-II, e a síntese foi realizada com otimização balanceada de velocidade e área. A FPGA Stratix-II apresenta uma estrutura um pouco diferente da FPGA ACEX, usada no exemplo anterior. Sua célula lógica é denominada ALUT (do inglês *Adaptive Look-up Tables*) e pode ser configurada como combinacional ou sequencial. Uma célula lógica ALUT corresponde a 1.25 célula lógica tradicional (arquitetura baseada em LUTs de 4 entradas) [Alt04]. A ferramenta síntese inferiu 8 unidades de DSP (do inglês *Digital Signal Processing*), que são os multiplicadores do Super Hitachi-2. A FPGA Stratix-II, possui unidades de memória distribuídas. Neste projeto, a memória de programa é de 16 kbytes, inferidos em 131.072 bits. O custo de área mostrou-se pequeno, da ordem de 1 %. A explicação para este baixo custo é que foram utilizadas asserções de menor complexidade e o processador SuperHitachi2 é um projeto bem mais complexo que os exemplos anteriores. Quanto ao custo de velocidade, o projeto com as asserções teve um desempenho superior se comparado ao projeto original. De alguma forma, a inserção das asserções ajudou na heurística de roteamento da ferramenta de síntese.

Tabela 5.10: Asserções inseridas no microprocessador Super Hitachi-2

Tipo de asserção	Funcionalidade
<code>assert_never</code>	Garante que códigos de instrução (do inglês <i>opcodes</i> errados não serão executados.
<code>assert_implication</code>	<i>Pipeline</i> deve parar caso ocorra acesso simultâneo na memória e no banco de registradores.
<code>assert_always</code>	Garante coerência entre os sinais de controle e a largura da memória lida/escrita (16/32 bits).
<code>assert_implication</code>	Garante a sincronização entre os estados de espera da memória e o estágio de busca de instruções do <i>pipeline</i> .
<code>assert_no_overflow</code>	Monitoria estouro do contador de programa.
<code>assert_implication</code>	Garante que o <i>pipeline</i> irá parar enquanto a instrução de multiplicação estiver sendo executada.
<code>assert_never</code>	Garante que não ocorrerão transições inválidas na máquina de estados do controlador de memória.
<code>assert_always</code>	Garante que não ocorrerão acessos a regiões inválidas de memória.

Tabela 5.11: Resultados de síntese para o microprocessador Super Hitachi-2

Parâmetro	Estrutura Original	Estrutura Modificada	Custo
Células lógicas (Flip-flops)	3.864	3.922	1,50%
Células lógicas (Combinacional)	1.463	1.479	1,09%
Memória	131.072 bits	131.072 bits	0%
Elementos de DSP	8	8	0%
Frequência máxima	52,60 MHz	53,15 MHz	-1,04%

5.3 Injeção de falhas

O exemplo do μ RISC-II, apresentado na subseção 5.2.2, será utilizado para demonstrar o funcionamento do processador de asserções. O diagrama de blocos com os principais módulos da arquitetura implementada pode ser observado na Figura 5.1. As instruções

executadas pelo μ RISC-II estão armazenadas em uma memória de 64 K words. Os únicos sinais externos são clock e reset. Os sinais do processador de asserções, `reset_n`, `ei`, `esci`, `eo`, `esco`, `esclk`, `escen_n`, estão conectados ao μ RISC-II.

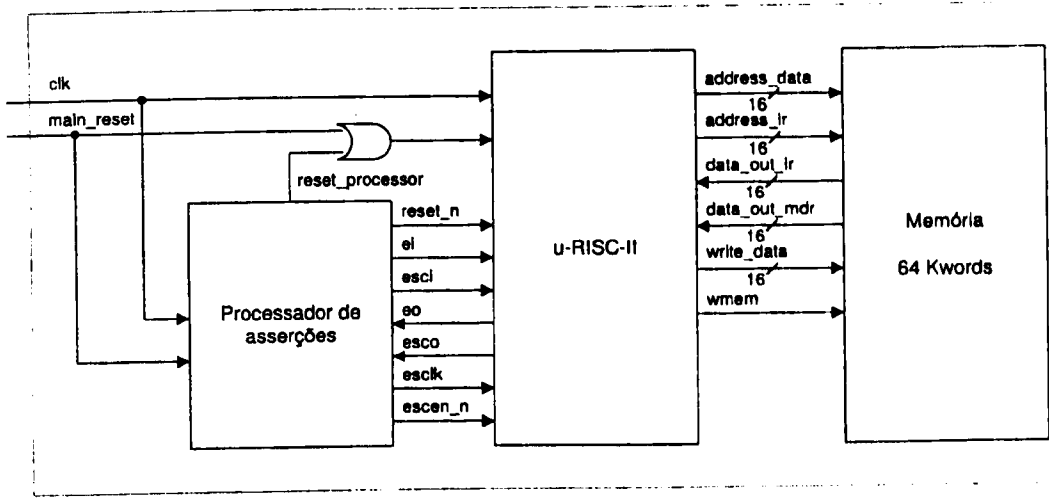


Figura 5.1: Diagrama de blocos do processador de asserções monitorando o μ RISC-II

No módulo μ RISC-II existem 10 asserções, conforme ilustra a Figura 5.2. Estas asserções estão encadeadas conforme a arquitetura apresentada no Capítulo 4.

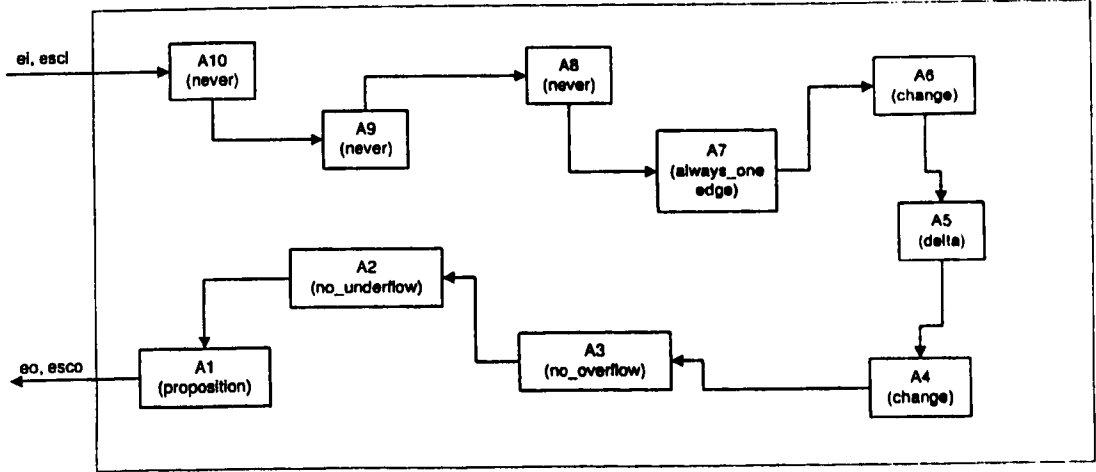


Figura 5.2: Diagrama de blocos das asserções inseridas no μ RISC-II

A asserção A8 é responsável por garantir que um desvio condicional e um desvio incondicional nunca estarão ativos ao mesmo tempo. A Figura 5.3 apresenta o resultado da

simulação do μ RISC-II quando a asserção A8 é ativada. O simulador utilizado é o evers [Pra04].

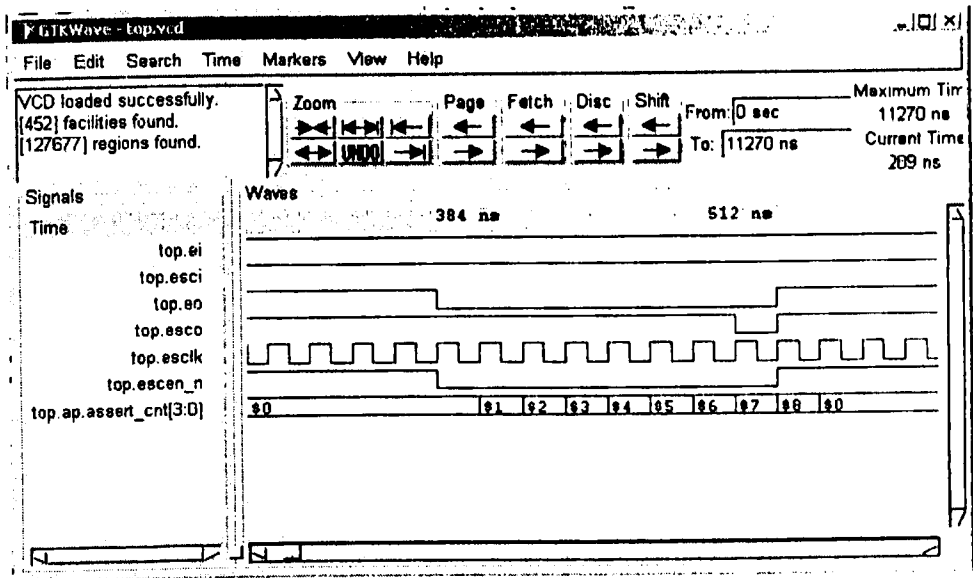


Figura 5.3: Resultado da simulação do μ RISC-II com uma falha na asserção A8

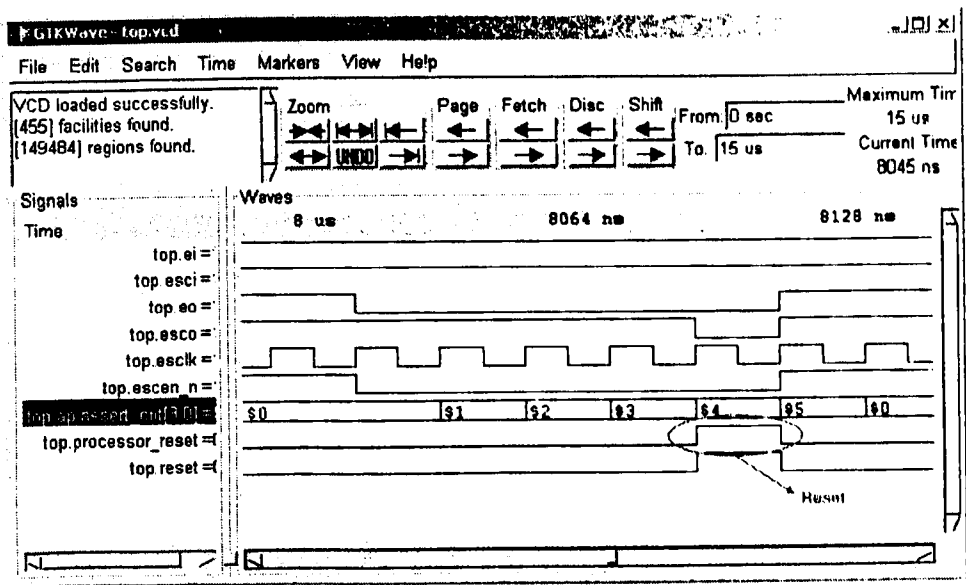


Figura 5.4: Resultado da simulação do μ RISC-II com uma falha na asserção A5

Os sinais apresentados na Figura 5.3 são vistos da entidade de mais alta hierarquia

(top). Pode-se observar que quando o sinal `eo` é ativado, o contador de asserções `assert_cnt` é zerado e incrementado a cada borda de subida do sinal de clock de escaneamento (`esclk`). Finalmente quando o sinal `esco` é ativado, tem-se em `assert_cnt` o valor da asserção que falhou.

A Figura 5.4 apresenta os resultados de simulação quando a asserção A5 falha. Esta asserção é responsável por garantir que o contador de programa sempre esteja dentro de um intervalo válido, ou seja, nunca execute uma instrução em um endereço inválido de memória. Neste caso, o processador de asserções está configurado para resetar o μ RISC se a asserção A5 falhar.

Capítulo 6

Conclusões e Trabalhos Futuros

Devido à crescente complexidade do esforço de verificação, as técnicas tradicionais de verificação não podem garantir que um circuito integrado seja projetado sem erros. Neste trabalho, foi apresentada uma arquitetura para a adição de asserções ao circuito integrado final. Esta arquitetura é baseada em uma versão modificada da biblioteca OVL. A versão modificada da OVL permite que as asserções inseridas no circuito integrado sejam encadeadas e gerenciadas pelo processador de asserções. Foi desenvolvida uma ferramenta protótipo para a realização automática do processo de encadeamento de asserções. Esta ferramenta tem como entrada um projeto com várias asserções da versão original da OVL instanciadas, e como saída, apresenta o projeto com as asserções da versão modificada da OVL já devidamente encadeadas e a lista de ordem de encadeamento. Foi apresentado também o processador de asserções que é o circuito responsável pelo gerenciamento das asserções adicionadas ao circuito integrado. O processador de asserções pode tanto informar quando uma asserção falhar através dos pinos de E/S, quanto interagir no próprio circuito integrado para que o mesmo se recupere de situações de erro. Foram apresentados resultados de adição de asserções em três microprocessadores. O custo de área mostrou-se pequeno, caso sejam usadas asserções menos complexas. O custo de velocidade mostrou-se desprezível.

Como trabalhos futuros, serão investigadas formas mais eficientes de fornecer informação acerca de asserções falhas como, por exemplo, uma interface de rede com agente SNMP (do inglês *Simple Network Management Protocol*) embutido. Igualmente será investigada a utilização de outras linguagens de especificação de propriedade, como PSL e System Verilog. É necessário também desenvolver uma metodologia para escolha de quais asserções serão definitivamente incorporadas ao circuito integrado, visando a uma boa relação custo/benefício. Para tanto, é necessário realizar um estudo sobre a cobertura alcançada pelas asserções.

Referências Bibliográficas

- [Acc03a] Accellera. Open Verification Library Assertion Monitor Reference Manual, 2003.
- [Acc03b] Accellera. Proposed Standard Open Verification Library Users Reference Manual, 2003.
- [Acc04a] Accellera. Proposed Standard Property Specification Language (PSL), March 2004.
- [Acc04b] Accellera. Proposed Standard System Verilog 3.1a, March 2004.
- [AH99] Algirdas Avizienis and Yutao He. *Dependable Computing for Critical Applications* 7, volume 12, chapter Microprocessor Entomology: A Taxonomy of Design Faults in COTS Microprocessors, pages 3–23. IEEE Computer Society, 1999.
- [Ait04] Thorn Aitch. Aquarius: A pipelined risc cpu (superh-ii isa compatible cpu core). Disponível em <http://www.opencores.org/projects/Aquarius>, 2004.
- [Alt04] Altera Corporation. Stratix Device Handbook, Volume 1, July 2004.
- [AMD03a] AMD. AMD Athlon TMProcessor Model 10 Revision Guide (C), October 2003.
- [AMD03b] AMD. AMD Athlon TMProcessor Model 6 Revision Guide (G), October 2003.
- [AMD03c] AMD. AMD Athlon TMProcessor Model 8 Revision Guide (E), October 2003.
- [AMD04] AMD. Revision Guide for AMD Athlon TM64 and AMD Opteron TMProcessors, June 2004.
- [Atk68] Daniel E. Atkins. Higher radix division using estimates of the divisor and partial remainder. *IEEE Transactions on Computers*, 17(10):925–934, 1968.
- [BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Acta Informatica* 20, 1983.
- [BBN⁺04] Bob Bentley, Kurt Baty, Kevin Normoyle, Makoto Ishii, and Einat Yogev. Verification: What Works and What Doesn't. In *Proceedings of Design Automation Conference*, 2004.

- [BCC⁺99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320. ACM Press, 1999.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer-Verlag, 1999.
- [BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 46–51. ACM Press, 1990.
- [Bei95] B. Beizer. The Pentium Bug - an Industry Watershed. Testing Techniques Newsletter, TTN, September 1995.
- [Ben01] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Proceedings of Design Automation Conference*, pages 244–248, 2001.
- [Ber00] Janick Bergeron. *Writing Testbenches*. Kluwer Academic Publishers, 2000.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1982.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
- [CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
- [CMH00] David Van Campenhout, Trevor Mudge, and John P. Hayes. Collection and analysis of microprocessors design errors. *IEEE Design & Test of Computers*, 17(4):51–60, 2000.
- [Coe01] Claudionor Nunes Coelho. Especificação do μ risc-ii. Disponível em <http://www.lecom.dcc.ufmg.br/urisc/urisc.html>, 2001.
- [CQ04] Gianpiero Cabodi and Stefano Quer. *Advanced Formal Verification*, chapter Advancements in mixed BDD and SAT techniques, pages 45–76. Kluwer Academic Publishers, 2004.

- [DH02] Rolf Drechsler and S. Höreth. Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment. In *Proceedings of International Workshop on Boolean Problems*, 2002.
- [Dil98] David L. Dill. What's between simulation and formal verification? In *Proceedings of the 35th ACM/IEEE conference on Design automation*, pages 328–329. ACM Press, 1998.
- [dONCF03] Márcia M. de Oliveira, José Augusto Nacif, Claudionor N. Coelho, and Antônio Otávio Fernandes. XRoach: A Tool for Generation of Embedded Assertions. In *Proceedings of Chip in Sampa Student Forum*, 2003.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [Dre03] R. Drechsler. Synthesizing checkers for on-line verification of system-on-chip designs,. In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages IV748–IV751, 2003.
- [Dre04a] Rolf Drechsler. *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.
- [Dre04b] Rolf Drechsler. Towards Formal Verification on the System Level. In *Proceedings of IEEE International Workshop on Rapid Systems Prototyping*, 2004.
- [FC01] H. Foster and C. Coelho. Assertions Targeting a Diverse Set of Verification Tools. In *Proceedings of International HDL Conference*, 2001.
- [FKL04] Harry Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2004.
- [For00] Formal Methods Group IBM Haifa Research Laboratory. Guide to Sugar Formal Specification Language, November 2000.
- [Fre04] Freescale Semiconductors. Mask Set Errata for MC9S12DP512 microcontroller, Mask 0L00M, August 2004.
- [G. 89] G. Stalmarck. A System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from a Formula. Swedish patent no. 467076(1992), U.S. patent no. 5276897(1994), European patent no. 0404454(1995), 1989.
- [GD04] Daniel Große and Rolf Drechsler. Checkers for SystemC Designs. In *Proceedings of 2nd ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2004.

- [GRS⁺03] Rajesh Gupta, Shishpal Rawat, Sandeep Shukla, Brian Bailey, Dan Beece, Masahiro Fujita, Carl Pixley, John O'Leary, and Fabio Somenzi. Formal verification - prove it or pitch it. In *Proceedings of the 40th conference on Design automation*, pages 710–711. ACM Press, 2003.
- [Gup92] Aarti Gupta. Formal hardware verification methods: a survey. *Form. Methods Syst. Des.*, 1(2-3):151–238, 1992.
- [IEE93] IEEE Standard 1076-1993. Vhdl language reference manual. IEEE Inc., 1993.
- [IEE01a] IEEE. *Standard 1149.1-2001*. IEEE Press, 2001.
- [IEE01b] IEEE Standard 1364-2001. Ieee standard hardware description language based on the verilog hardware description language. IEEE Inc., 2001.
- [Int99] Intel Corp. Intel Pentium Processor Specification Update, January 1999.
- [Int02] Intel Corp. Intel Pentium 2 Processor Specification Update, July 2002.
- [Int03] Intel Corp. Intel Pentium 3 Processor Specification Update, November 2003.
- [Int04] Intel Corp. Intel Pentium 4 Processor Specification Update, August 2004.
- [KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual conference on Design automation*, pages 263–268, 1997.
- [KN96] M. Kantrowitz and L. Noack. I'm done Simulating; Now What? Verification Coverage Analysis and Correctness Checking DEC chip 21164 Alpha Microprocessor. In *Procedings of Design Automation Conference*, pages 325–330, 1996.
- [KP03] Hyeong-Ju Kang and In-Cheol Park. Sat-based unbounded symbolic model checking. In *Proceedings of the 40th conference on Design automation*, pages 840–843. ACM Press, 2003.
- [Kri63] S. Kripke. Semantic considerations on model logic. In *Proceedings of a Colloquium: Modal an Many Valued Logics, volume 16 of Acta Philosophica Fennica*, pages 83–94, 1963.
- [Kro00] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 2000.
- [Lit04] Jay Littlefield. Using Formal Verification to Create Robust IP. EEdesign.com, July 2004.

- [Mat96] Yusuke Matsunaga. An efficient equivalence checker for combinational circuits. In *Proceedings of the 33rd annual conference on Design automation*, pages 629–634. ACM Press, 1996.
- [MBC⁺00] Don MacMillen, Michael Butts, Raul Camposano, Dwight Hill, and Thomas W. Williams. An industrial view of electronic design automation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 19(12):1428–1447, 2000.
- [MC79] Carver Mead and Lynn Conway. *Introduction to VLSI Design*. Addison-Wesley Publishers, 1979.
- [McF93] Michael C. McFarland. Formal verification of sequential hardware: a tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):633–654, 1993.
- [McM03] K. L. McMillan. Interpolation and sat based model checking. In *Proceedings of the 15th Conference on Computer Aided Verification*, pages 250–264. Springer-Verlag, 2003.
- [MIP94] MIPS Technologies Inc. MIPS R4000PC/SC Errata, Processor Rev. 2.2 and 3.0, 1994.
- [Nac04] José Augusto M. Nacif. Desenvolvimento de circuitos integrados tolerantes a falhas de projeto através de processadores de asserções. In *Proceedings of Workshop de Teses e Dissertações em Computação Tolerante a Falhas*, 2004.
- [NdPC⁺03] José Augusto Nacif, Flávio M. de Paula, Claudionor N. Coelho, Fernando C. Sica, Harry Foster, Antônio O. Fernandes, and Diógenes Cecílio da Silva Júnior. The Chip is Ready. Am I done? On-chip Verification using Assertion Processors. In *Proceedings of IFIP International Conference on Very Large Scale Integration System on a Chip VLSI-SoC*, pages 111–116, 2003.
- [NdPF⁺03] José Augusto Nacif, Flávio M. de Paula, Harry Foster, Claudionor N. Coelho, Fernando C. Sica, Diógenes C. da Silva, and Antônio O. Fernandes. An Assertion Library for On-Chip White-Box Verification at Run-Time. In *Proceedings of Latin-American Test Workshop*, 2003.
- [OH02] M. Oliveira and A. Hu. High-level specification and automatic generation of ip interface monitors. In *Proceedings of 39th Design Automation Conference*, pages 129–134, 2002.
- [OMN⁺04] Leonardo F. Otaviano, Leandro Maciel, José Augusto Nacif, Claudionor N. Coelho, and Antônio Otávio Fernandes. Validation of a Verilog Assertion Library. In *Proceedings of Chip in the Reefs Student Forum*, 2004.

- [PICW04] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang. Safety property verification using sequential sat and bounded model checking. *IEEE Design & Test of Computers*, 21(2):132-143, 2004.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46-57, 1977.
- [Pra04] Pragmatic-c. GPL CVER. Disponível em: <http://www.pragmatic-c.com/gpl-cver/index.htm>, 2004.
- [SAV03] D.G. Saab, J.A. Abraham, and V.M. Vedula. Formal verification using bounded model checking: Sat versus sequential atpg engines. In *Proceedings of the 16th International Conference on VLSI Design*, pages 315-319. IEEE CS Press, 2003.
- [SKKK04] Dominik Stoffel, Evgeny Karibaev, Irina Kufareva, and Wolfgang Kunz. *Advanced Formal Verification*, chapter Equivalence Checking of Arithmetic Circuits, pages 77-123. Kluwer Academic Publishers, 2004.
- [Ton04] Tony Bybell. GTKWave Electronic Waveform Viewer. Disponível em: <http://www.cs.man.ac.uk/apt/tools/gtkwave/>, 2004.
- [TQB⁺98] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional Verification of a Multiple-issue Out-of-order, Superscalar Alpha Processor - the DEC Alpha 21264 Microprocessor. In *Proceedings of Design Automation Conference*, pages 638-645, 1998.
- [TS02] Simon Teran and Jaka Simsic. 8051 core. Disponível em <http://www.opencores.org/projects/8051>, November 2002.
- [Xil01] Xilinx. VirtexTM2.5 V Field Programmable Gate Arrays Product Specification, April 2001.

Apêndice A

Open Verification Library

Neste Apêndice as asserções da OVL serão apresentadas. A Tabela A.1 apresenta a descrição dos sinais presentes na maioria das asserções. Maiores detalhes podem ser encontrados em [Acc03a].

Tabela A.1: Descrição dos principais sinais em comum da OVL

Sinal	Descrição
severity_level	Severidade da asserção. O valor padrão de severidade é 0 (maior prioridade).
msg	Mensagem de erro que será impressa pelo simulador quando uma asserção for violada.
options	Opção para ser usada na ferramenta EDA.
inst_name	Nome da instância da asserção.
clk	Clock da asserção. Normalmente a asserção é avaliada em toda borda de subida deste sinal.
reset_n	Sinal de inicialização, ativo em zero.

A.1 assert_always

```
assert_always [#(severity_level, options, msg)] inst_name (clk,  
reset_n, test_expr);
```

Esta assertção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. A expressão de teste pode ser qualquer expressão booleana. O objetivo é garantir que `test_expr` seja sempre verdadeira. Caso contrário, um erro será reportado.

A.2 `assert_always_on_edge`

```
assert_always_on_edge [#(severity_level, edge_type, options, msg)]  
inst_name (clk, reset_n, sampling_event, test_expr);
```

Esta assertção monitora continuamente `test_expr` a cada borda de subida do sinal `clk`, e também a cada borda especificada do sinal `sampling_event`. A borda válida do sinal `sampling_event` é definida pelo sinal `edge_type`, e os valores válidos são:

- 0 - Sem borda;
- 1 - Borda de subida;
- 2 - Borda de descida;
- 3 - Bordas de subida/descida.

A expressão `test_expr` sempre deve ser verdadeira na borda especificada de `sampling_event`. Se `test_expr` for falsa, a assertção reportará um erro.

A.3 `assert_change`

```
assert_change [#(severity_level, width, num_cks, flag, options,  
msg)] inst_name (clk, reset_n, start_event, test_expr);
```

Esta assertção monitora continuamente a expressão `start_event` em toda borda de subida de `clk`, ou a cada evento. Quando `start_event` se torna verdadeira, esta assertção assegura que `test_expr` irá mudar de valor em algum ponto, dentro de um certo número de ciclos de clock definidos por `num_cks`. Se `test_expr` mudar de valor neste intervalo e, após, `start_event` assumir o valor verdadeiro, a assertção é satisfeita, e uma nova verificação começa a monitorar `start_event` no ciclo de clock em que `test_expr` muda de valor. Assim, quando uma assertção é válida, uma nova verificação começa antes de atingir o número de ciclos determinado por `num_cks`.

A.4 assert_cycle_sequence

```
assert_cycle_sequence [#(severity_level, num_cks,  
necessary_condition, options, msg)] inst_name (clk, reset_n,  
event_sequence);
```

Esta asserção verifica as seguintes condições:

- Se `necessary_condition` é 2'b00: Assegura que se os `num_cks-1` eventos da sequência especificada em `event_sequence` forem verdadeiros, então a última deve ocorrer. A verificação é feita em modo *pipelined*.
- Se `necessary_condition` é 2'b01: Assegura que se o primeiro evento `num_cks-1` de `event_sequence` ocorrer, todos os eventos restantes devem ocorrer. A verificação é feita em modo *pipelined*.
- Se `necessary_condition` é 2'b10: Assegura que se o primeiro evento `num_cks-1` de `event_sequence` ocorrer, todos os eventos restantes devem ocorrer. A verificação é feita em modo não *pipelined*.

A.5 assert_decrement

```
assert_decrement [#(severity_level, width, value, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. O objetivo é garantir que a expressão de teste nunca será decrementada de um valor diferente de `value`. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.6 assert_delta

```
assert_delta [#(severity_level, width, min, max, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` nunca esteja fora do intervalo determinado pelos sinais `min` e `max`. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.7 assert_even_parity

```
assert_even_parity [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` sempre contenha um número par de bits setados. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.8 assert_fifo_index

```
assert_fifo_index [#(severity_level, depth, push_width, pop_width,  
options, msg)] inst_name (clk, reset_n, push, pop);
```

Esta asserção assegura que uma estrutura do tipo FIFO (do inglês *First In First Out*) nunca passará de um limite inferior ou superior (*underflow* ou *overflow*). Este tipo de monitoramento pode ser configurado para suportar múltiplas ações de leitura e escrita de uma estrutura FIFO em um determinado ciclo. Os principais sinais envolvidos são:

- `depth`: Especifica o número máximo de elementos da estrutura FIFO;
- `push_width`: Define o número de bits do argumento `push`. O valor padrão é 1.
- `pop_width`: Define o número de bits do argumento `pop`. O valor padrão é 1.
- `push`: Define o número de escritas que podem ocorrer na estrutura FIFO em um determinado ciclo. O valor padrão é uma única escrita por ciclo.
- `pop`: Define o número de leituras que podem ocorrer na estrutura FIFO em um determinado ciclo. O valor padrão é uma única leitura por ciclo.

A.9 assert_frame

```
assert_frame [#(severity_level, min_cks, max_cks, flag, options,  
msg)] inst_name (clk, reset_n, start_event, test_expr);
```

Esta asserção tem o objetivo de verificar as relações de temporização entre dois eventos. Quando o sinal `start_event` é verdadeiro, a expressão de teste `test_expr` deve também ser verdadeira dentro de um intervalo mínimo e máximo de ciclos definidos pelos sinais `min_cks` e `max_cks`. O sinal `flag` possui as seguintes opções:

- 0: Ignora eventos `start_event` ocorridos após o primeiro. Este é o valor padrão.
- 1: Reinicia o monitoramento de `test_expr` se o sinal `start_event` é ativado em ciclos subsequentes de clock.
- 2: Reporta um erro se um `start_event` é ativado em qualquer ciclo de clock, se `test_expr` estiver sendo monitorada.

A.10 `assert_handshake`

```
assert_handshake [(severity_level, min_ack_cycle, max_ack_cycle,
req_drop, deassert_count, max_ack_length, options, msg)] inst_name
(clk, reset_n, req, ack);
```

Esta asserção monitora continuamente os sinais `req` e `ack` a cada borda de subida do sinal de `clk`. O objetivo desta asserção é garantir que o protocolo de confirmação (do inglês *handshake*) funcione adequadamente, de acordo com os seguintes parâmetros:

- `min_ack_cycle`: Quando este parâmetro for maior que zero, a verificação é ativada. Ele verifica quando um `ack` ocorre num tempo igual ou superior a `min_ack_cycle` ciclos de clock.
- `max_ack_cycle`: Quando este parâmetro for maior que zero, a verificação é ativada. Ele verifica se um `ack` ocorre num tempo igual ou superior a `min_ack_cycle` ciclos de clock;
- `req_drop`: Quando este parâmetro for maior que zero, a verificação é ativada. Ele verifica se `req` mantém-se ativo por um ciclo completo ou até ocorrer um `ack`;
- `deassert_count`: Quando este parâmetro for maior que zero, a verificação é ativada. Ele verifica se `req` fica inativo (0) em `deassert_count` ciclos após um `ack`;
- `max_ack_length`: Quando este parâmetro for maior que zero, a verificação é ativada. Este parâmetro descreve o número máximo de ciclos de um `req` ou `ack`.

A.11 `assert_implication`

```
assert_implication [(severity_level, options, msg)] inst_name
(clk, reset_n, antecedent_expr, consequent_expr);
```

Esta asserção monitora continuamente `antecedent_expr` a cada borda de subida do sinal `clk`. Se esta expressão for verdadeira, a asserção irá verificar se `consequent_expr` também é verdadeira.

A.12 assert_increment

```
assert_increment [#(severity_level, width, value, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. O objetivo é garantir que a expressão de teste nunca será incrementada de um valor diferente de `value`. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.13 assert_never

```
assert_never [#(severity_level, options, msg)] inst_name (clk,  
reset_n, test_expr);
```

Esta asserção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. A expressão de teste pode ser qualquer expressão booleana. O objetivo é garantir que `test_expr` nunca será verdadeira. Caso contrário, um erro será reportado.

A.14 assert_next

```
assert_next [#(severity_level, num_cks, check_overlapping,  
only_if, options, msg)] inst_name (clk, reset_n, start_event, test_expr);
```

Esta asserção tem o objetivo de verificar as relações de temporização entre dois eventos. Quando o sinal `start_event` é verdadeiro, a expressão de teste `test_expr` deve também ser verdadeira dentro de um número exato de ciclos definido por `num_cks`. O parâmetro `check_overlapping` permite que `start_event` ocorra em paralelo com outra sequência. O parâmetro `only_if` garante que `test_expr` nunca seja verdadeira sem que antes `start_event` seja também verdadeira.

A.15 assert_no_overflow

```
assert_no_overflow [#(severity_level, width, min, max, options,  
msg)] inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. O objetivo desta asserção é garantir que a expressão de teste nunca

mudará seu valor de um valor máximo, definido pelo parâmetro `max`, para um valor maior que `max` ou menor ou igual a um valor mínimo, definido pelo parâmetro `min`. O parâmetro `width` é o tamanho máximo de `test_expr`.

A.16 `assert_no_transition`

```
assert_no_transition [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, test_expr, start_state, next_state);
```

Esta asserção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. Quando o valor de `test_expr` for igual ao valor de `start_event`, esta asserção garante que `test_expr` nunca mudará o seu valor para `next_state`. O parâmetro `width` é o tamanho máximo de `test_expr`.

A.17 `assert_assert_no_underflow`

```
assert_no_underflow [#(severity_level, width, min, max, options,  
msg)] inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. O objetivo desta asserção é garantir que a expressão de teste nunca mudará seu valor de um valor mínimo, definido pelo parâmetro `min`, para um valor menor que `min` ou maior ou igual a um valor máximo, definido pelo parâmetro `max`. O parâmetro `width` é o tamanho máximo de `test_expr`.

A.18 `assert_odd_parity`

```
assert_odd_parity [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` sempre contenha um número ímpar de bits setados. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.19 `assert_one_cold`

```
assert_one_cold [#(severity_level, width, inactive, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` sempre contenha um único bit em nível lógico 0. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.20 `assert_one_hot`

```
assert_one_hot [(severity_level, width, options, msg)] inst_name  
(clk, reset_n, test_expr);
```

Esta asserção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` sempre contenha um único bit em nível lógico 1. O sinal `width` é o número de bits da expressão de teste `test_expr`.

A.21 `assert_proposition`

```
assert_proposition [(severity_level, options, msg)] inst_name  
(reset_n, test_expr);
```

Esta asserção monitora continuamente `test_expr`. O objetivo desta asserção é garantir que a expressão de teste seja sempre verdadeira. A diferença desta asserção em relação a todas as outras asserções da OVL é que `assert_proposition` não possui clock, ou seja, trata-se de uma asserção completamente combinacional.

A.22 `assert_quiescent_state`

```
assert_quiescent_state [(severity_level, width, options, msg)]  
inst_name (clk, reset_n, state_expr, check_value, sample_event);
```

Esta asserção monitora o sinal `sample_event` em toda borda de subida de `clk`. O objetivo desta asserção é garantir que uma máquina de estados (representada pelo sinal `state_expr`) é igual a um valor determinado pelo parâmetro `check_value` na borda de subida do sinal `sample_event` e, opcionalmente, no final da simulação. O sinal `width` é o número de bits da expressão de teste `state_expr`.

A.23 `assert_range`

```
assert_range [(severity_level, width, min, max, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora o sinal `sample_event` em toda borda de subida de `clk`. O objetivo desta asserção é garantir que o valor de `test_expr` estará dentro de um intervalo especificado pelos parâmetros `min` e `max`. O sinal `width` é o número de bits da expressão de teste `state_expr`.

A.24 `assert_time`

```
assert_time [#(severity_level, num_cks, flag, options, msg)]  
inst_name (clk, reset_n, start_event, test_expr);
```

Esta asserção monitora o sinal `sample_event` em toda borda de subida de `clk`. Quando este sinal for verdadeiro, a asserção garante que `test_expr` será verdadeira nos próximos `num_cks` ciclos de clock. O sinal `flag` possui as seguintes opções:

- 0: Ignora eventos `start_event` ocorridos após o primeiro. Este é o valor padrão.
- 1: Reinicia o monitoramento de `test_expr` se o sinal `start_event` é ativado em ciclos subsequentes de clock.
- 2: Reporta um erro se um `start_event` é ativado em qualquer ciclo de clock, se `test_expr` estiver sendo monitorada.

A.25 `assert_transition`

```
assert_transition [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, test_expr, start_state, next_state);
```

Esta asserção monitora continuamente a expressão de teste `test_expr` a cada borda de subida do sinal `clk`. Quando o valor de `test_expr` for igual ao valor de `start_event`, esta asserção garante que se `test_expr` mudar de valor, mudará para o valor especificado por `next_state`. O parâmetro `width` é o tamanho máximo de `test_expr`.

A.26 `assert_unchange`

```
assert_unchange [#(severity_level, width, num_cks, flag, options,  
msg)] inst_name (clk, reset_n, start_event, test_expr);
```

Esta asserção monitora continuamente a expressão `start_event` em toda borda de subida de `clk` ou a cada evento. Quando `start_event` se torna verdadeira, esta asserção assegura que `test_expr` não irá mudar seu valor nos próximos ciclos de clock definidos por `num_cks`.

A.27 assert_width

```
assert_width [#(severity_level, min_cks, max_cks, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta asserção monitora continuamente a expressão `start_event` em toda borda de subida de `clk`. Quando a expressão é verdadeira, esta asserção garante que `test_expr` permanecerá verdadeira por um número mínimo de ciclos de clock, especificado pelo parâmetro `min_cks`, e não excederá um número máximo de ciclos de clock, especificado pelo parâmetro `max_cks`.

A.28 assert_win_change

```
assert_win_change [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, start_event, test_expr, end_event);
```

Esta asserção monitora continuamente a expressão `start_event` em toda borda de subida de `clk`. Quando este sinal for verdadeiro, a asserção garante que a expressão de teste `test_expr` tem o seu valor alterado antes que o sinal `end_event` se torne verdadeiro.

A.29 assert_win_unchange

```
assert_win_unchange [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, start_event, test_expr, end_event);
```

Esta asserção monitora continuamente a expressão `start_event` em toda borda de subida de `clk`. Quando este sinal for verdadeiro, a asserção garante que a expressão de teste `test_expr` não terá o seu valor alterado antes que o sinal `end_event` se torne verdadeiro.

A.30 assert_window

```
assert_window [#(severity_level, options, msg)] inst_name (clk,  
reset_n, start_event, test_expr, end_event);
```

Esta asserção monitora continuamente a expressão `start_event` em toda borda de subida de `clk`. Quando `start_event` se torna verdadeira, a asserção garante que `test_expr` será verdadeira em toda borda de subida de `clk` até que o sinal `end_event` seja verdadeiro.

A.31 assert_zero_one_hot

```
assert_zero_one_hot [#(severity_level, width, options, msg)]  
inst_name (clk, reset_n, test_expr);
```

Esta assertção monitora a expressão de teste `test_expr` em toda borda de subida de `clk`. Sua especificação determina que `test_expr` sempre contenha um único ou nenhum bit em nível lógico 1. O sinal `width` é o número de bits da expressão de teste `test_expr`.