FABRÍCIO VIVAS ANDRADE

UMA ARQUITETURA PARA VERIFICAÇÃO DE BLOCOS DE COMPUTAÇÃO GRÁFICA EM HARDWARE

Belo Horizonte

19 de agosto de 2005

FABRÍCIO VIVAS ANDRADE

UMA ARQUITETURA PARA VERIFICAÇÃO DE BLOCOS DE COMPUTAÇÃO GRÁFICA EM HARDWARE

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

Belo Horizonte

19 de agosto de 2005

Resumo

O presente trabalho propõe, analisa e valida uma arquitetura inédita para verificação de blocos de computação gráfica em hardware. A verificação automática e a verificação em alto e baixo nível de abstração são os dois pilares que dão suporte aos quatro estágios que compõe esta arquitetura. O primeiro estágio estabelece a definição e implementação de uma especificação executável em alto nível de abstração para o bloco de computação gráfica em hardware. O segundo estágio corresponde a implementação do mesmo bloco em RTL (do Inglês, *Register Transfer Level*) instanciando as asserções. Os dois últimos estágios são responsáveis pela verificação por sub-blocos e pelaa verificação em nível de sistema do bloco em hardware. Para auxiliar o estágio de verificação, este trabalho apresenta uma ferramenta para visualização gráfica da simulação em RTL (GVT) e uma ferramenta para a verificação automática em alto e baixo nível de abstração (V^2T). A validação da arquitetura foi determinada através de um estudo de caso que incluiu o projeto, a verificação e a prototipação de um bloco de computação gráfica para uma plataforma de inspeção ótica.

Abstract

This work presents, analyzes and validates a novel verification architecture for computer graphics cores. This architecture has four stages and it is supported by two techniques: the automatic verification and the high and low level verification. The first stage defines and implements a high level executable especification for the computer graphics core. The second stage implements the Register Transfer Level (RTL) implementation and assertions are written. The last two stages are responsible for the sub-block verification and the system level verification of the design. In order to improve the verification stage performance, this work presents a graphical tool (GVT) and a Computer Aided Design tool (V^2T) that provides automatic verification at a higher and lower level of abstraction. Finally, a case study is perfomed to validate the proposed architecture of verification. This case study includes the design, verification and prototyping of a computer graphics core for an automatic optical inpection platform.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Este trabalho propõe, analisa e valida uma arquitetura para verificação de blocos de computação gráfica em hardware[1]. A verificação automática e a verificação em alto nível de abstração são os dois pilares que dão suporte a esta arquitetura e que a tornam eficiente e eficaz. Além disso, este trabalho apresenta uma ferramenta CAD (V^2T) para a verificação automática em alto nível de abstração, e mecanismos de verificação entre especificações executáveis e modelos em linguagem de descrição de hardware.

Dentre os diversos tipos de projetos de circuitos integrados existentes, os blocos de computação gráfica foram escolhidos como objeto de estudo deste trabalho porque são bastante populares em diversos dispositivos eletrônicos como celulares e PDAs (do Inglês, *Personal Digital Assistent*). Também porque, até o hoje, nenhuma metodologia foi proposta especificamente para a verificação destes blocos em hardware. Por isso, os mesmos ainda são verificados por meio de metodologias tradicionais que, por muitas vezes, são inadequadas para esta tarefa.

Um bloco de computação gráfica em hardware geralmente possui funcionalidades básicas de desenho em duas dimensões como os desenhos de círculos, linhas e preenchimento de polígonos, estes são denominados de primitivas de saída [1]. A metodologia mais freqüentemente utilizada para verificação destes blocos é a verificação funcional em conjunto com a verificação por inspeção visual das primitivas obtidas [2]. Contudo, a inspeção visual possui grandes limitações como a dificuldade em se garantir que uma primitiva foi desenhada com o mesmo tom da cor especificada ou, em se determinar se alguns pixels foram desenhados nas coordenadas especificadas ou em coordenadas adjacentes. Outro ponto negativo é que a eficiência desta técnica pode variar de acordo o estado psicológico de quem faz a inspeção. Além disso, não há a garantia de um padrão de qualidade e esta técnica de verificação pode demandar muito tempo. Por isso, a arquitetura proposta executa a verificação de forma automática, eliminando

[1] O termo *bloco de Computação Gráfica em hardware* será utilizado como tradução do termo em Inglês *Computer Graphics core*. No entanto, outras traduções como *módulo gráfico de propriedade intelectual* ou *core de Computação Gráfica* também seriam possíveis.

estas limitações.

Problemas de verificação de circuitos integrados, como o tratado no presente trabalho, são muito freqüentes nos dias de hoje. Principalmente porque houve um crescimento acelerado na utilização de pequenos dispositivos de computação embutida nas últimas duas décadas. Segundo Mishra et al. [3], dois fatores foram os principais motivadores para este aumento: os avanços tecnológicos e a crescente demanda por dispositivos eletrônicos.

Em relação ao primeiro destes ítens, de acordo com a Lei de Moore [4], existe um crescimento exponencial do número de elementos em um circuito integrado. Assim, é de se esperar que o esforço para se projetar e, principalmente, verificar estes circuitos também cresçam exponencialmente. Para o segundo ítem, a necessidade de se colocar um produto novo no mercado em períodos de tempo cada vez menores leva freqüentemente ao lançamento de produtos com defeitos.

Com o intuito de se evitar defeitos como estes que geram enormes prejuízos para as empresas desenvolvedoras, a tarefa de verificação tornou-se a parte mais importante e a que mais demanda tempo em um projeto de circuito integrado. Nos dias de hoje, esta tarefa já representa de 50% a 80% do tempo total de desenvolvimento do circuito integrado [5, 6, 7].

Apesar de toda essa dedicação à tarefa de verificação, existem vários projetos de circuitos integrados em que os erros foram apenas detectados após a entrada do produto no mercado. Um dos casos mais famosos é o defeito na unidade de divisão de ponto flutuante do Pentium [8]. Este acontecimento obrigou a Intel a gastar aproximadamente 475 milhões de dólares na substituição dos processadores defeituosos.

Ainda sobre os projetos da Intel, segundo Schubert [9], os defeitos lógicos encontrados antes de fazer o primeiro circuito integrado para o Pentium foram de 800, para o Pentium Pro foram de 2240, para o Pentium IV foram de 7855. Esta tendência mostra um crescimento de 300% a 400% de defeitos em cada geração de processadores, por simples extrapolação, pode-se argumentar que a próxima geração de processadores deverá conter cerca de 25000 defeitos.

Outro exemplo clássico de falha em um sistema que causou milhões de dólares em prejuízos e ainda a morte de algumas pessoas foi a explosão do foguete espacial Ariane 5 em 4 de julho de 1996. Segundo Clarke [10], a explosão deste foguete aconteceu após apenas 40 segundos do seu lançamento e, após investigações, constatou-se que o erro aconteceu devido a uma falha no trecho do software que era responsável por calcular o movimento do foguete. Apesar da falha que causou este desastre não ter acontecido no hardware, este exemplo é muito útil para se mostrar a importância das técnicas de verificação.

Para lidar com a crescente dificuldade do problema de projeto e verificação de circuitos integrados, a solução mais eficaz foi a utilização de metodologias que aumen-

taram o nível de abstração. Desde a década de 30, as técnicas utilizadas no projeto e verificação de circuitos integrados evoluíram rapidamente. Passando por projetos utilizando equações em Álgebra Booleana na década de 30, Diagrama de Blocos nos anos 40, Linguagem de Transferência de Registros (RTL) na década de 60, passando por Linguagens de Descrição de Hardware (HDL, do Inglês *Hardware Description Language*) nos anos 80, Linguagens para Verificação de Hardware nos anos 90 e, finalmente, Linguagens de Propriedades e Asserções a partir do final dos anos 90 [5]. Seguindo esta tendência, a arquitetura de verificação proposta no presente trabalho é baseada na verificação em alto nível de abstração.

Atualmente, as metodologias para projetos de circuitos integrados são inadequadas para desenvolver projetos em ASICs (do Inglês, *Application Specific Integrated Circuits*) com milhões de portas lógicas [11]. Para oferecer um diretriz a esse problema, a metodologia de desenvolvimento em plataforma, chamada de SoC, (do Inglês, *System-on-a-Chip*) foi proposta.

Esta metodologia baseia-se no reuso de blocos de propriedade intelectual em hardware para reduzir o tempo gasto no desenvolvimento do projeto de circuito integrado. Isto torna o problema de verificação de circuitos integrados ainda mais importante e vital. Já que, com essa metodologia, um bloco de propriedade intelectual com um erro afetará tantos projetos quanto o número de vezes que o mesmo for reutilizado.

Devido a vital importância da tarefa de verificação de circuitos integrados e a grande tendência na utilização de blocos de computação gráficas em hardware, o presente trabalho propõe, analisa e valida uma arquitetura para verificação destes blocos de forma automática e em alto nível de abstração.

O presente trabalho foi organizado da seguinte forma: o capítulo 2 fornece uma revisão bibliográfica das metodologias e técnicas utilizadas atualmente para a verificação de circuitos integrados; o capítulo 3 mostra a arquitetura de verificação proposta para verificação de blocos de computação gráfica em hardware; o capítulo 4 detalha a implementação de um bloco de computação gráfica em diferentes níveis de abstração; o capítulo 5 mostra os resultados obtidos utilizando a arquitetura proposta em um estudo de caso; e, finalmente, o capítulo 6 ressalta as conclusões e possíveis extensões do trabalho atual.

Capítulo 2

Revisão Bibliográfica

Este capítulo fornece uma breve revisão bibliográfica sobre os métodos de verificação de circuitos integrados atualmente utilizados. Ao longo dos anos, vários autores utilizaram termos como *verificação* com significados diferentes, por isso, uma conceituação da terminologia utilizada faz-se necessária.

A definição do termo *verificação* foi extraída de Bergeron [2] que afirma que verificação é o processo usado para demonstrar que um projeto está funcionalmente correto. Segundo o mesmo autor, o processo de verificação pode ser dividido em duas grandes vertentes: a verificação formal e a verificação funcional. A primeira contempla os métodos de verificação por equivalência, verificação de modelos e prova de teoremas. Já a segunda abrange os métodos de verificação por caixa branca, caixa preta e caixa cinza. Uma outra vertente de verificação que não é destacada pelo autor é a verificação semi-formal [5] que é uma técnica intermediária entre a verificação formal e a funcional.

Outra definição importante do vocabulário utilizado neste trabalho vem dos termos: *teste* e *verificação*. Bergeron [2] define o termo *teste* como o propósito de se garantir que a transformação da descrição de *netlists* de um projeto de circuito integrado para uma representação no silício foi efetuada corretamente. Já o termo *verificação* é o processo que visa garantir que a transformação da especificação para a implementação foi executada de forma correta.

Estes termos servirão de tópicos para as próximas seções; e, apesar de existirem outras definições para os mesmos, as definições previamente citadas estão mais de acordo critérios adotados neste trabalho. Além disso, esta divisão possibilita uma exposição mais didática sobre os assuntos abordados.

2.1 O Problema de Verificação

A crescente complexidade dos projetos de circuitos integrados, devido aos avanços obtidos pela tecnologia VLSI (do Inglês, *Very Large Scale Integrated Circuit*), tornou o esforço de verificação bastante elevado. A maior demanda do mercado por dispositivos

com cada vez mais funcionalidades, diminuiu a janela de tempo entre produtos, fazendo com que o problema de se garantir a exatidão dos projetos torne-se cada vez mais árduo. Além disso, uma detecção tardia de erros nestes projetos podem invalidá-los, gerando enormes prejuízos para as empresas desenvolvedoras de circuitos integrados.

Para se afirmar que existe um erro em algum projeto de circuito integrado, é necessário possuir uma referência clara e correta do mesmo para confrontá-la com o projeto elaborado. No contexto de verificação de circuitos integrados, conceitos denominados de *implementação* e *especificação* e o conceito do termo *satisfazer* são muito importantes, e segundo Gupta [12]:

- a *implementação* refere-se ao projeto de circuito integrado a ser verificado;
- a *especificação* refere-se a propriedades sobre as quais a exatidão está para ser determinada; e
- *satisfazer* refere-se a relação que garante que a implementação possui as propriedades definidas na especificação.

Assim, um projeto de circuito integrado é dito verificado se a implementação satisfaz a especificação. Um problema que parece evidente após estas definições é de como garantir que a especificação está correta. Isto é, se um projeto basear-se em uma especificação incorreta, não é possível se obter uma implementação correta, mesmo que a implementação satisfaça a especificação.

O problema de se garantir que a especificação significa realmente o que esta deveria significar é muito vago e por isso foi deixado de lado. Assume-se que as especificações fornecidas indicam de forma exata e correta o que deve ser realizado. Partindo deste princípio, vários métodos estão disponíveis para garantir que a implementação satisfaz ou não a especificação, e estes métodos são detalhados nas seções a seguir.

2.2 Verificação Funcional

O processo de verificação funcional consiste em garantir que a implementação obedece às propriedades definidas na especificação. A forma mais comum de fazê-la, explicada de forma bem sucinta, é composta por três passos: cria-se um modelo do dispositivo a ser verificado (DUV - do Inglês, *Device under Verification*), aplicam-se estímulos em sua entrada e monitora-se a sua saída. A saída obtida deve ser comparada com a saída esperada, especificada ou calculada por um modelo de referência do projeto do circuito integrado. Este mecanismo é conhecido como caso de teste e a Figura 2.1 mostra como isto pode ser feito.

A verificação funcional pode ser alcançada por meio de três abordagens diferentes: a verificação por caixa preta, a verificação por caixa branca e a verificação por caixa

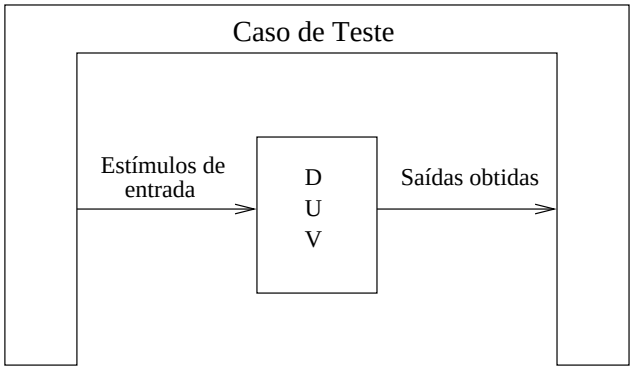

Figura 2.1: Caso de teste aplicado ao DUV.

cinza. Estas abordagens diferem-se principalmente em relação a observabilidade e a controlabilidade que cada uma fornece.

Segundo Foster et al. [5], o termo *observabilidade* refere-se a habilidade em se observar uma estrutura interna ou linha de código de um projeto de circuito integrado quando esta é estimulada. Já o termo *controlabilidade*, segundo o mesmo autor, refere-se a habilidade em se estimular uma linha específica de código ou uma estrutura desejada.

As três diferentes abordagem para a verificação funcional são detalhadas nas seções seguintes.

2.2.1 Verificação por Caixa Preta

O conceito de verificação por caixa preta talvez seja o mais comum e o mais intuitivo. Neste método de verificação, casos de testes são elaborados sem um conhecimento das estruturas internas do DUV. De uma forma mais detalhada, o primeiro passo consiste em desenvolver um modelo do hardware em uma linguagem de descrição de hardware. Em seguida, cria-se um caso de teste que instancia o modelo que corresponde ao DUV. O caso de teste gera estímulos na entrada do DUV e oferece meios para que as saídas em resposta a aqueles estímulos de entrada possam ser verificados.

As maiores dificuldades na utilização desta abordagem estão na limitação da controlabilidade e da observabilidade. Já que sem o conhecimento interno das estruturas, torna-se difícil selecionar partes mais críticas do projeto para se verificar. Além disso, mesmo que uma estrutura seja estimulada é possível que o erro não seja observável na saída. Outra grande dificuldade é que mesmo que os erros propaguem-se para a saída, isto pode acontecer após um longo atraso ou após alguns pulsos de clock, tornando ainda mais difícil a detecção da origem do problema.

Algumas vantagens podem ser destacadas quando uma verificação por caixa preta é utilizada. A primeira é que a definição dos casos de teste pode ser feita em paralelo com o desenvolvimento do modelo em linguagem de descrição de hardware. Isto acontece

porque o caso de teste não necessita de informações das estruturas internas do modelo para ser produzido. Outra conseqüência, com a mesma justificativa, é que mesmo que o modelo do DUV seja modificado, o caso de teste pode permanecer o mesmo, já que o modelo e o caso de teste são independentes na fase de elaboração. Outra vantagem é que estímulos aleatórios com uma dada distribuição também podem ser utilizados, pelos mesmos motivos apresentados anteriormente.

A verificação por caixa preta permite cobrir, na maior parte das vezes, apenas um pequena parte do projeto de circuito integrados Já que o total de estímulos a serem aplicados é demasiadamente grande: de 2^n para circuitos combinacionais, onde n é o número de entradas e de $2^{(n+m)}$ para circuitos seqüenciais, onde o novo termo m corresponde ao número de estados existentes.

Para exemplificar a abordagem de verificação por caixa preta, a Figura 2.2 mostra um pequeno trecho de circuito extraído de um projeto em circuito integrado complexo. Esta figura mostra um erro muito comum entre os projetistas que utilizam linguagem de descrição de hardware que é a inferência de *latches*. Isto acontece freqüentemente quando se deseja modelar um circuito combinacional, mas o desenvolvedor não faz o assinalamento da saída para todos os valores de entrada possíveis. A ferramenta de síntese assume que, se o valor não foi definido para uma determinada combinação de entrada, o valor anterior deve ser mantido. Isto gera um elemento armazenador de informação como um *latch*[1].

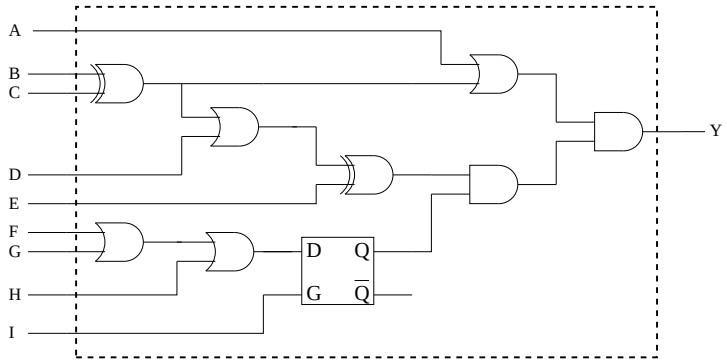

Figura 2.2: Exemplo de verificação por caixa preta.

A Figura 2.2 mostra o circuito a ser verificado, onde $A, B, C, D, E, F, G, H, I$ correspondem as entradas e Y corresponde a saída. A caixa delimitada pelas linhas pontilhadas definem o que é visível ao desenvolvedor, isto é, ele apenas possui informações a respeito das entradas e saídas, nenhuma estrutura interna é conhecida. A única informação disponível é que este trecho corresponderia a um circuito combinacional.

Aplicando estímulos nas entradas e monitorando a saída, o desenvolvedor deve identificar a não conformidade deste circuito em relação a especificação, ou seja, o *latch* inferido. Com a verificação por caixa preta, este trabalho pode tornar-se árduo.

[1]Uma discussão mais detalhada sobre este processo é descrita por Smith [13].

Pois, uma combinação de entradas pode não habilitar o *latch* evitando que o seu sinal de entrada propague-se para a saída, tornado difícil descobrir qual estímulo gerou o erro. Além disso, é possível que a cobertura feita pelos testes não seja suficiente para determinar se existe algum erro.

2.2.2 Verificação por Caixa Branca

A verificação funcional por meio de caixa branca visa oferecer uma total observabilidade das estruturas internas ou linhas de código em linguagem de descrição de hardware da implementação a ser verificada. Como o desenvolvedor tem total visibilidade interna do dispositivo, ele pode gerar casos de testes para estimular pontos específicos do projeto de circuito integrado. Isto possibilita exercitar funcionalidades em particular gerando combinações específicas de entradas o que não acontece em uma verificação por caixa preta.

Esta abordagem, no entanto, tem algumas desvantagens. A primeira delas é que o caso de teste só pode ser gerado após o término da implementação, já que o conhecimento das estruturas internas é necessário. Outro ponto negativo é que esta abordagem torna o processo de verificação muito depende da implementação, dificultando o reaproveitamento dos mesmos casos de teste em futuras implementações ou extensões do projeto do circuito integrado.

O mesmo exemplo abordado na Figura 2.2 pode ser retomado para uma verificação por caixa branca. A Figura 2.3 mostra com isto poderia ser feito. Com a verificação por caixa branca, todos os pontos do circuito integrado podem ser observados. Neste exemplo, os pontos inseridos foram p, q, r, s, t, u, v. Com o ponto t, pode-se observar que por ser um circuito combinacional, definindo-se as entradas F, G, H, a saída deveria também ser assinalada após um atraso de propagação. Isso não acontece porque um *latch* foi inferido.

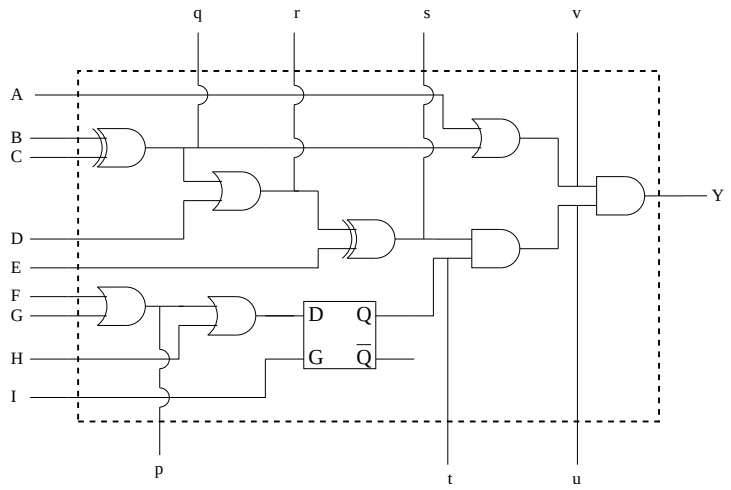

Figura 2.3: Exemplo de verificação por caixa branca.

Utilizando a abordagem de verificação por caixa branca, o erro da inferência do *latch* pode ser identificado rapidamente antes este se propague para o resto de circuito. Porém, como afirmado anteriormente, o processo de verificação ficaria muito dependente da implementação. Outra limitação que se torna evidente é que os outros pontos de observação como o p, q, r, s, u, v foram inseridos e não auxiliaram no processo de verificação, apenas aumentaram o número de variáveis a serem analisadas.

2.2.3 Verificação por Caixa Cinza

A verificação por caixa cinza é um caminho intermediário entre a verificação por caixa branca e a por caixa preta. Esta é utilizada para fornecer uma maior observabilidade e controlabilidade na criação dos casos de teste, mas não tão completas quanto as fornecidas pela verificação por caixa branca. Assim, a verificação por caixa cinza tenta estabelecer um compromisso entre os outros métodos de verificação.

A Figura 2.4 mostra como uma verificação por caixa cinza pode ser feita. Apenas alguns pontos no circuito são observados (r, s, t), e estes são selecionados de acordo com critérios estabelecidos pelo desenvolvedor. Neste exemplo, a inferência do *latch* poderia ser facilmente detectada com a inserção do ponto t, como aconteceu na verificação por caixa branca. Contudo, por meio do processo de verificação por caixa cinza este ponto poderia não ser escolhido, tornando este processo similar ao de verificação por caixa preta.

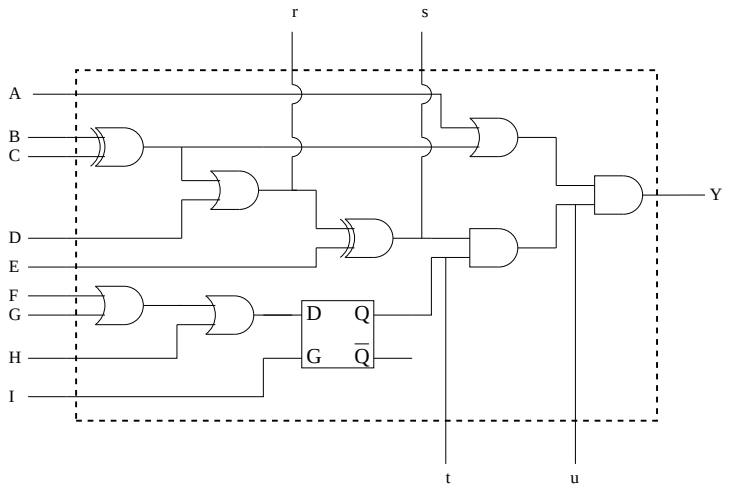

Figura 2.4: Exemplo de verificação por caixa cinza.

2.2.4 Técnicas Complementares a Verificação Funcional

Para todos os métodos de verificação funcional descritos, algumas ferramentas e técnicas complementares são necessários. Algumas destas serão descritas nesta seção para introduzir conceitos que são abordados novamente nos capítulos seguintes. Os conceitos

principais aqui descritos são o de simuladores, visualizadores de forma de onda e ferramentas de cobertura de código.

Segundo Bergeron [2], a ferramenta mais popular no processo de verificação é o simulador. Esta ferramenta é utilizada para imitar o comportamento do ambiente em que o DUV será submetido ao casos de teste. Simuladores são ferramentas dinâmicas, ou seja, necessitam de estímulos de entradas, geram valores de saídas e ainda necessitam que o desenvolvedor verifique se as saídas geradas correspondem aos valores esperados.

A ferramenta mais freqüentemente utilizada em conjunto com o simulador é o visualizador de forma de onda. Após uma simulação, geram-se valores de saídas que podem estar em um formato ASCII ou em um formato específico como VCD (do Inglês, *Value Changed Dump*) [14] que pode ser carregado em um visualizador de forma de onda [15]. Estes visualizadores auxiliam na interpretação das saídas geradas porque possibilitam a observação e a comparação de múltiplos sinais e transições no tempo. Além disso, tornam possível a medição de intervalos de tempo entre eventos e também fornecem dados em diferentes bases numéricas.

Como argumenta Bergeron [2], os visualizadores possibilitam ao desenvolvedor analisar informações produzidas e armazenadas após diversas horas de simulação de uma forma bastante rápida. No entanto, esta ferramenta apresenta apenas as saídas obtidas com a simulação, mas não fornece nenhum auxílio no intuito de garantir que os resultados obtidos estão corretos. Mesmo porque a tarefa de interpretação e comparação dos resultados cabe ao desenvolvedor.

Uma das maiores dificuldades quando métodos de verificação funcional são utilizados consiste em determinar quando o processo de verificação chegou ao fim. Mesmo após diversos casos de testes haverem sido aplicados e nenhum erro ou não conformidade com a especificação tiver sido encontrada, não significa que o DUV esteja funcionalmente correto. É possível que um erro exista mas ainda não tenha se manifestado pelas combinações de entrada já aplicadas.

Para este problema, a melhor solução são os métodos e as ferramentas de cobertura de código. Estes métodos já vêm sendo utilizado por muitos anos em técnicas de verificação de software e, nos últimos anos, na verificação de hardware. A ferramenta de cobertura de código inserem mecanismos de inspeção em pontos estratégicos do código fonte do DUV para detectar se funcionalidades ou construções particulares foram exercitadas.

Dentre as diversas possibilidades da análise de cobertura do código, diferentes técnicas podem ser utilizadas. A primeira consiste em verificar se determinados assinalamentos ou blocos foram estimulados. A segunda forma consiste em verificar se determinados caminhos, em um fluxo de execução, foram executados. Finalmente, pode-se verificar se as expressões utilizadas no códigos fonte foram cobertas. Após a execução da cobertura de código, pode-se obter um valor que indica a porcentagem co-

berta do código fonte. Na maior parte dos casos, uma cobertura de 100% do código não é viável. Esta técnica continua sendo amplamente pesquisada e diferentes abordagens já foram propostas e utilizadas com bastante sucesso [16, 17, 18, 19]

2.3 Verificação Formal

As limitações dos métodos e das técnicas de verificação funcional devido à coberturas cada vez menores dos projetos de circuitos integrados, motivaram o desenvolvimento de metodologias de verificação formal. Nos últimos anos, estas metodologias empregadas em ferramentas comerciais ocuparam uma lacuna deixada pelos métodos de verificação funcional.

Nos métodos de verificação funcional, a credibilidade e a confiança no projeto de circuito integrado elaborado surgem devido a execução de inúmeros casos de testes para aquele projeto. Ou seja, a medida que diferentes partes do projeto são exercitadas e erros não são aparentes, a probabilidade da existência de algum erro diminui e, portanto, maior é a expectativa de que todo o projeto esteja correto.

Nos métodos para verificação formal, métodos matemáticos são utilizados para examinar propriedades do projeto em uma enumeração de todos os estados alcançáveis para o sistema. Desta forma, é possível saber exatamente quando o ciclo verificação do circuito integrado já atingiu o seu fim.

Segundo Seger [20], apesar dos consideráveis esforços feitos na área de verificação funcional, a maior parte dos resultados obtidos concentrou-se apenas em aumentar a eficiência destes métodos melhorando o desempenho da simulação. Recentemente, algumas contribuições ainda são feitas nesta direção como em relação a formas de se alterar propriedades lógicas da simulação [21], em aumentar o desempenho dos mecanismos de simulação por simulação distribuída [22], ou ainda, projetando hardwares específicos para emulação [23, 24]. Seger qualifica este tipo de pesquisa como verificação por força-bruta e esta argumentação também é compartilhada por Bryant [25].

Dentre os métodos para verificação formal destacam-se a Prova de Teoremas, a Verificação de Modelos e a Verificação por Equivalência.

2.3.1 Verificação por Prova de Teoremas

Uma das formas mais antigas para a verificação formal de um circuito integrado é através da descrição da especificação e da implementação por meio de lógicas formais. Para garantir que o circuito está correto, faz-se a prova que a lógica formal que descreve a especificação é igual a que descreve a implementação.

Para se entender como o método de Prova de Teoremas é aplicado, é necessário explicar quais são as suas bases teóricas. A principal delas é a Teoria Formal como

descrito em [26]. Uma Teoria Formal S é dada por:

- Um alfabeto finito é dado e os símbolos desse alfabeto são os símbolos da teoria. Uma seqüência finita desses símbolos é chamada de *expressão* de *S*.
- Um subconjunto de expressões de S é chamado de *fórmula bem formada* de *S*.
- Um conjunto finito de *fórmulas bem formadas* de S é chamado de *axioma* de *S*.
- Um conjunto finito de *regras de inferência* é dado. Um regra de inferência permite deduzir novas *fórmulas bem formadas* a partir de um conjunto finito de *fórmulas bem formadas*.

Assim, uma *prova formal* em S é dada por uma seqüência de *fórmulas bem formadas*: $f_1, f_2, ..., f_n$ em que para cada i da fórmula f_i é um axioma ou pode ser deduzido através das regras de inferência a partir de um conjunto finito de fórmulas $f_1, f_2, ..., f_{i-1}$. A última fórmula obtida através das regras de inferência é chamada de *teorema* de *S*, e a prova formal é a *prova* deste teorema.

Para se utilizar a verificação por Prova de Teoremas pode-se usar vários tipos de lógicas e, segundo Gupta [12], as mais utilizadas são:

- lógica de predicado de primeira ordem,
- lógica computacional de Boyer-Moore e
- lógica de alta ordem.

A seguir, uma breve descrição de cada uma delas é fornecida.

Lógica de Predicado de Primeira Ordem

A lógica de predicado de primeira ordem pode ser considerada uma extensão da lógica proposicional (lógica de predicado de ordem zero). A lógica proposicional trabalha com conjunto de símbolos para variáveis e um conjunto padrão de conectivos booleanos como *negação* ($\neg$), *e* ($\wedge$), *ou* ($\vee$), *conseqüência lógica* ($\Rightarrow$) e *equivalência* ($\equiv$). A lógica proposicional foi a primeira a ser utilizada para especificar e verificar circuitos digitais, pois existe uma transposição direta entre a lógica proposicional e a implementação de circuitos combinacionais.

Porém, a lógica proposicional perdeu a sua força a medida em que circuitos seqüenciais tornaram-se mais populares já que esta não consegue especificá-los. Assim, houve a necessidade de escolher uma lógica de ordem superior e a escolha natural foi a lógica de predicado de primeira ordem. A extensão fornecida por esta lógica refere-se aos quantificadores *para todo* ($\forall$) e *existe* ($\exists$) e ao conceito de função que possibilita modelar circuitos seqüenciais colocando a variável tempo como argumento das mesmas.

Lógica Computacional de Boyer-Moore

A lógica computacional de Boyer-Moore [27] consiste em uma forma restrita da lógica de primeira ordem porque não possui quantificadores. Os conectivos lógicos *and*, *not* e *or* são definidos em termos de primitivas lógicas constantes, como *verdadeiro* e *falso*, a primitiva de conectividade *if* e a função *igual*. A sua sintaxe é bem semelhante a utilizada na linguagem LISP [28]. As regras de inferências utilizadas são as mesmas que servem para a lógica de primeira ordem, adicionando-se o princípio da indução.

A abordagem proposta por Boyer-Moore foi a primeira a fornecer um provador de teoremas. Este sistema provê a prova automática de teoremas com o mínimo de intervenção humana, a lógica em conjunto com o provador foram denominados de sistema Boyer-Moore [27]. Este sistema foi utilizado pela empresa Computational Logic Inc. [29] para verificar um computador completo (FM8501) tanto no nível de software quanto de hardware.

Os pontos fortes da utilização de métodos baseados nessa lógica para verificar projetos de circuitos integrados é evidenciado no trabalho de Hunt [30]. Nesse trabalho, fica aparente a eficácia da utilização de mecanismos automáticos como provadores de teoremas que, mesmo não sendo totalmente automáticos, são um passo definitivo no caminho da Verificação Formal. Segundo Gupta [12], isto acontece porque um enorme número de teoremas surge quando circuitos complexos e extensos como o de microprocessadores são verificados. Uma tentativa de se desenvolver provas não-automáticas (manuais) é um processo trabalhoso e ainda impossível para alguns casos. Portanto, a automação dos provadores de teoremas, seja em qualquer grau possível, é extremamente necessária.

Em relação aos pontos negativos, existe uma grande dificuldade em se lidar com o tempo quando se utiliza lógica de primeira ordem e, conseqüentemente, lógica de Boyer-Moore. Isto acontece porque todas as expressões tem que ser escritas em função do tempo[2].

Lógica de Alta Ordem

O sistema de lógica de alta ordem que é baseado em uma lógica propriamente dita e um provador de teoremas foi desenvolvido pelo Hardware Verification Group na Universidade de Cambridge. Este sistema foi proposto por Gordon [31] e foi proposto para verificar e especificar projetos de circuitos integrados.

A lógica proposicional, ou lógica de ordem zero, só admite conectivos e variáveis no domínio verdadeiro e falso. A lógica de primeira ordem amplia a lógica proposicional utilizando quantificadores para as variáveis do domínio. Uma extensão para a lógica

[2]Existem lógicas mais apropriadas para a se trabalhar com o tempo com a Lógica Temporal. Esta possui operadores que fornecem um representação implícita para o tempo. Esta lógica é descrita a seguir e é utilizada como base para outros métodos de verificação.

de primeira ordem é possibilitar quantificadores sobre variáveis (ex. predicados), esta é chamada de lógica de segunda ordem. Assim, a lógica de alta ordem permite a quantificação sobre predicados arbitrários. Isto habilita quantificar sobre símbolos de predicado fornecendo a lógica um alto poder de expressividade.

A lógica de alta ordem pode ser utilizada verificar circuitos integrados em diferentes níveis de abstração. Apesar de ser mais usada para verificação estrutural, a sua expressividade permite especificar e verificar em outros níveis de abstração como o de comportamento, de dados e o temporal.

O sistema de alta ordem foi utilizado para verificar integralmente um microprocessador chamado Viper [32]. Isso não seria novidade já que o sistema de Boyer-Moore já havia verificado o FM8501. Porém, a arquitetura do Viper foi projetada para uso comercial diferentemente do FM8501 que não passou de protótipos. Provavelmente, o Viper foi um dos primeiros produtos verificados formalmente a ser comercializado.

Segundo Gupta [12], o sistema de alta ordem é muito poderoso porque alia uma lógica de alta expressividade com a prova automática de teoremas. A facilidade de se expressar tanto o comportamento como a estrutura do hardware e a habilidade de se verificar diversos níveis de abstração, tornaram este sistema muito interessante para se verificar circuitos integrados muito extensos. Por outro lado, a maior desvantagem deste sistema é que as proposições que provam da especificação são escritas da seguinte forma:

$$\forall i_1 i_2 \dots i_m o_1 o_2 \dots o_n . Imp(i_1, i_2 \dots i_m, o_1, o_2 \dots o_n) \implies Spec(i_1, i_2 \dots i_m, o_1, o_2 \dots o_n)$$

onde, $i_1, i_2 \dots i_m$ são entradas e $o_1, o_2 \dots o_m$ são saídas. Escrevendo a proposição desta forma, implementação garantidamente falsa tornaria toda a proposição verdadeira. Assim, um predicado falso da implementação poderia satisfazer qualquer especificação. Mesmo parecendo um problema extremamente simples, formas de contorná-lo já foram propostas e a principal delas foi publicada pelo próprio grupo desenvolvedor do sistema de alta ordem. Para uma discussão mais detalhada, boas referências são Gupta [12] e Brock et al. [32].

2.3.2 Verificação de Modelos

Nos anos 80, uma abordagem para verificação verificação formal diferente da prova de teoremas foi proposta com intuito de reduzir o tempo gasto no ciclo de verificação e tornar este processo automático. Esta abordagem é denominada de verificação de modelos em lógica temporal e foi proposta de forma independente por Clarke e Emerson [33] e Quielle e Sifakis [34].

De forma sucinta, nesta abordagem, as especificações são escritas em lógica temporal e os circuitos são modelados através de um grafo de transições de estado. A partir

daí, um algoritmo de busca bastante eficiente é utilizado para se determinar a validade das especificações em cada um dos estados.

Os projetos em hardware são facilmente modelados porque estes podem ser vistos, em sua maior parte, como sistemas concorrentes de estados finitos. Estes estados podem ser determinados em um ponto em particular no tempo, examinando-se todos os estados do sistema. Em geral, cada estado possui um estado seguinte ou próximo estado. O próximo estado de um sistema pode ser definido pelo estado atual e o conjunto de entradas atuais. O par *estado atual - estado seguinte* representa uma relação de transição no sistema. Um caminho pode ser definido como uma seqüência de estados. Assim, pode-se definir um modelo para um sistema através de um grafo de transição de estados, o mais adequado é conhecido como estrutura de Kripke [35].

Uma estrutura de Kripke M é dada pela tupla de quatro $M = (S, S_0, R, L)$, onde:

- S é um conjunto de estados e S_0 é o estado inicial;
- R é o conjunto de transições entre estados dado por $R \subseteq S \times S$; e
- L é uma função que mapeia cada estado com uma proposição atômica, $L\colon S \rightarrow 2^{AP}$.

Esta estrutura é utilizada para descrever modelos extraídos diretamente dos níveis lógicos de circuitos integrados ou de modelos em um nível mais alto de abstração sem nenhum refinamento para a implementação.

A partir do modelo, é necessário definir a especificação que este deve obedecer. Isto pode ser feito através da definição de propriedades que são proposições sobre o que é esperado do modelo. Na verificação de modelos, estas propriedades são descritas em um forma de lógica temporal. A lógica proposicional descrita na seção anterior, adequa-se bem para especificações de propriedades que são independentes do tempo (estáticas). A lógica temporal é uma extensão da lógica proposicional que permite descrever propriedades em situações que variam no tempo.

Existe uma grande quantidade de lógicas temporais, dentre as quais destacam-se a lógica temporal em tempo linear proposta por Pnueli [36] e lógica temporal de desvios no tempo proposta por Clarke [33]. O segundo formalismo destacou-se mais por ser utilizado na verificação de modelos; principalmente um subconjunto deste formalismo denominado de CTL* (do Inglês, *Computation Logic Tree*) e por isso será detalhado.

CTL* é utilizada para definir fórmulas que descrevem propriedades de uma árvore que representa o modelo, neste caso, a estrutura de Kripke. Estas fórmulas são compostas por quantificadores de caminho e operadores temporais. Existem dois quantificadores, o primeiro é representado pelo símbolo A que significa *para todos os caminhos de computação* e o segundo é representado pelo símbolo E que significa *para algum caminho de computação*.

Em relação aos operadores temporais tem-se,

- o operador X que requer que uma propriedade seja válida no próximo estado;
- o operador F que determina que uma propriedade será válida em algum estado no futuro;
- o operador G que assegura que uma propriedade é válida em todo um caminho;
- o operador U que obriga que se existir um estado em que uma segunda propriedade é válida, o caminho até este estado deve ser válido para uma primeira propriedade; e finalmente,
- o operador R que requer que uma segunda propriedade seja válida em todo um caminho até o estado em que a primeira propriedade seja válida.

A partir destes operadores e quantificadores é possível definir propriedades que devem ser válidas para o modelo. A Figura 2.5 mostra a seguinte propriedade: para *algum* caminho a propriedade g é *sempre* válida, isto é, EGg. Observe que a estrutura abaixo corresponde a um estrutura de Kripke desenrolada.

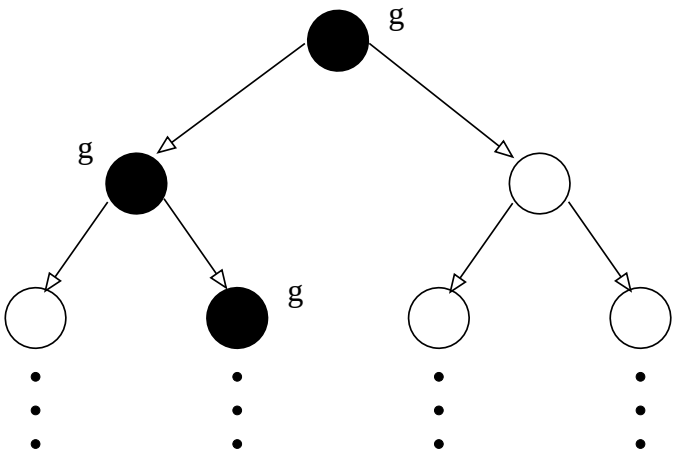

Figura 2.5: Exemplo da propriedade EGg em CTL*.

Clarke [33] propôs um sistema prático para verificação que combina lógica temporal (CTL*) com a representação de um circuito por meio de uma estrutura computacional (Kripke) com predicados anexados em vários pontos desta estrutura através de um verificador de modelos.

Com este verificador de modelos, aplica-se um algoritmo de prova para verificar se o modelo satisfaz as propriedades definidas. O algoritmo parte do estado inicial S_0 e procura pelos próximos estados alcançáveis, adicionando-os em uma lista. Cada novo estado desta lista é percorrido até que não existam mais estados a serem encontrados. O algoritmo de prova fornecerá um dos três tipos de resposta que são:

- sucesso - se todos estados alcançáveis foram atingidos e todas as propriedades foram satisfeitas; ou
- falha - antes que todos os estados fossem alcançados, alguma propriedade não foi satisfeita; ou ainda,
- sem decisão - se antes que todos os estados fossem alcançados, estes não puderem ser representados em memória devido ao seu número excessivo. Este fenômeno é conhecido como explosão de estados.

Apesar da verificação de modelos já ter sido utilizada na prática em aplicações reais, esta possui algumas limitações sérias. A primeira delas está na definição das propriedades a serem verificadas a partir de uma especificação. Não existe nenhuma metodologia para sua definição, e o processo de se julgar quais delas são importantes é geralmente subjetivo e propenso a erros, seja pela definição de propriedades erradas ou pela definição de propriedades demasiadamente simples. Além disso, para sistemas complexos, algumas fórmulas temporais podem ser extremamente difíceis de se entender. Tornando complicado o processo de se descobrir o que está realmente sendo verificado.

A segunda maior limitação é que para a maior parte dos problemas o espaço de soluções é extremamente grande. Isto leva o algoritmo de prova a condição de indecisão devido ao fenômeno da explosão de estados. Uma solução proeminente para este problema é a verificação simbólica de modelos [37].

Na implementação original do algoritmo de verificação de modelos, cada transição é representada através de seus estados adjacentes. Para projetos reais de circuitos integrados, este tipo de representação não podia ser feita devido ao enorme número de estados e transições existentes.

A solução para este problema foi proposta por McMillan [37] que sugeriu que se uma representação simbólica das transições existentes nos grafos de estados fosse utilizada, sistemas muito mais extensos e complexos poderiam ser verificados. A estrutura de dados escolhida para esta representação foi proposta por Bryant [38] e é denominada OBDD (do Inglês,*Ordered Binary Decision Diagram*).

Os OBDDs foram escolhidos porque provêm uma forma canônica para representação de fórmulas booleanas mais compactas do que as formas conjuntiva ou disjuntiva e podem ser mais facilmente manipuladas computacionalmente. Com esta representação foi possível verificar sistemas com 10^{20} estados. Após alguns refinamentos, com esta mesma abordagem já é possível verificar sistema com 10^{120} estados [39].

Nos últimos anos, outra técnica denominada verificação limitada de modelos (BMC, do Inglês, *Bounded Model Checking*) ofereceu resultados muito positivos [40, 41]. A idéia principal desta abordagem é limitar o tamanho dos caminhos a serem considerados, assim só os erros ou contra-exemplos de tamanhos determinados são analisados. O

principal algoritmo utilizado nesta abordagem são os mesmos que resolvem o problema da satisfabilidade [42, 43].

2.3.3 Verificação por Equivalência

Verificação por equivalência é uma forma de verificação formal e estática que usa técnicas matemáticas para determinar se duas versões do mesmo projeto, em diferentes estágios de desenvolvimento são equivalentes. Este tipo de verificação é mais freqüentemente utilizada para comparar e determinar a equivalência entre:

- uma descrição em RTL e uma descrição em nível de portas lógicas (*netlists*);
- duas descrições em nível de portas; ou
- duas descrições em RTL, onde uma pode ser um refinamento da outra;

O primeiro ítem desta enumeração é geralmente é utilizado para garantir que a síntese de modelos em linguagem de descrição de hardware para descrições em portas lógicas feita por uma ferramenta está correta. As ferramentas de síntese são programas muito complexos e propensos a erros. A maior parte dos projetistas confia nas ferramentas e por isso este tipo de verificação é, às vezes, deixada de lado. No entanto, engenheiros da Digital Equipament Corporation descobriram que a ferramenta de síntese utilizada produzia erros quando multiplicadores de mais de 48 bits eram sintetizados. A verificação por equivalência identificou este erros de forma eficaz e rápida [2].

O segundo ítem é bastante utilizado para garantir que um pós-processamento feito nas descrições em níveis de portas lógicas não alterou a funcionalidade do circuito. Este tipo de pós-processamento pode ser relativo a inserção de *scan-chains*, *clock-tree synthesis* ou modificação manual.

Já o último ítem não é tão utilizado como os anteriores. A verificação por equivalência entre duas descrições em RTL só é feita quando uma delas corresponde apenas a algumas alterações em requisitos não funcionais da outra. Assim, para se evitar a execução de longos testes de regressão, faz-se a equivalência entre as descrições.

O fluxo de um processo de verificação por equivalência pode ser dividido em quatro passos [44], estes são:

1. ler a descrição em RTL ou em *netlists* da referência e da implementação, segmentando-os em pequenos blocos lógicos chamados de cones lógicos;
2. efetuar o casamento entre estes cones lógicos entre a referência e a implementação;
3. verificar com métodos matemáticos se cada um dos cones da referência é equivalente aos da implementação; e

4. identificar cones lógicos não equivalentes e fornecer meios para depuração.

Os cones lógicos são pequenos grupos de registradores, caixas pretas e portas de entrada. Cada um deles possui um ponto denominado ponto de comparação. A partir destes, a ferramenta de verificação por equivalência mapeia os pontos de comparação da referência e da implementação, executando, em seguida, o segundo passo que é o casamento entre eles. No terceiro, passo estes pontos são determinados equivalentes ou não. Finalmente, os não equivalentes são identificados e exibidos.

A estrutura de dados necessária para mapear tanto a referência quanto a implementação é a mesma utilizada na verificação de modelos que é o OBDD. Assim, uma maior eficiência na representação dos dados utilizando estas estruturas tornaria tanto a verificação de modelos quanto a verificação por equivalente mais eficientes.

2.4 Verificação Semi-Formal

A tarefa de verificação consiste em confrontar o conjunto de documentos definidos na especificação contra a implementação obtida. A verificação por propriedades pode ser entendida como uma proposição que a implementação deve respeitar. Ou seja, essa proposição ou asserção inserida em uma implementação não contribui para a funcionalidade da implementação. Esta é apenas uma forma de se elaborar uma consistência entre a intenção do projetista com o que ele implementou. Este mecanismo é conhecido como verificação semi-formal.

Dentre as principais formas de se utilizar asserções em um projeto de circuito integrado destacam-se as bibliotecas:

- *Open Verification Library* (OVL)
- *PSL Property Specification Language*
- *System Verilog assertion constructs*

Segundo a própria documentação da OVL [45], esta biblioteca provê para os desenvolvedores uma interface padronizada de asserções para validação por meio simulação possibilitando a verificação semi-formal e a verificação formal.

De acordo com Foster et al. [5], a PSL é uma linguagem formal de propriedades criada pela Accellera e que é compatível com qualquer linguagem de descrição de hardware. Como uma de suas maiores qualidades, uma especificação escrita em PSL é fácil de se ler, além de ser matematicamente precisa, que a torna ideal tanto para a tarefa de verificação quanto para a documentação.

System Verilog é a versão mais atual da linguagem de descrição de hardware Verilog. Em sua definição foram adicionadas construções que possibilitam a inserção de

asserções sem o uso de bibliotecas externas, o que é o seu ponto forte. As asserções descritas em System Verilog assemelham-se com as descritas na biblioteca OVL.

Asserções podem ser utilizadas tanto para técnicas de verificação por simulação (verificação dinâmica), quanto para verificações formais e semi-formais (verificação estática). Além disso, as asserções provem mecanismos para se analisar a cobertura dos códigos do modelo a ser verificado. A linguagem PSL também pode ser utilizada para capturar propriedades criando uma especificação formal para um projeto de circuito integrado.

Estas asserções podem são utilizadas em quatro diferentes camadas que são:

- a camada Booleana que serve para se definir expressões que são usadas em outras camadas. Apesar de ser definida como camada Booleana, esta pode definir expressões muito além das Booleanas. No entanto, o seu principal papel é servir de base para as outras camadas. As expressões Booleanas são avaliadas em um único pulso de *clock*.

- A camada Temporal é usada para definir propriedades a respeito do projeto e é o núcleo principal de uma linguagem de asserções. Estas propriedades temporais são avaliadas em múltiplos ciclos de *clock*.

- A camada de Verificação serve para fazer uma interface entre a camada Temporal onde são descritas as propriedades e a ferramenta que irá avaliar as mesmas.

- A camada de Modelagem serve para modelar as entradas do DUV e ainda hardwares auxiliares que não foram desenvolvidos no projeto mas precisam ser verificados.

É importante notar que o termo verificação semi-formal adequa-se muito bem a esta metodologia. Pois esta consiste em um compromisso entre a verificação funcional e a verificação formal. A definição de um modelo, a inserção de propriedades definidas em uma lógica temporal e a verificação das mesmas pode ser vista como um processo de verificação formal. Porém, neste caso, o que a torna uma verificação semi-formal seria que a definição do modelo não precisa ser feita estritamente através de um grafo de transições de estados ou de uma estrutura de Kripke como detalhado na seção 2.3.2 (verificação formal). De forma menos rigorosa, o modelo pode ser dado por uma linguagem de descrição de hardware. As propriedades podem ser definidas de forma bem similar as utilizadas em uma verificação formal. Já a ferramenta que verifica as propriedades pode ser tão rígida quanto o verificador de modelos ou então ser apenas uma simulação criada como um caso de teste como os descritos na seção 2.2.

Capítulo 3

Arquitetura para Verificação de Blocos de Computação Gráfica em Hardware

3.1 Motivação

3.1.1 Blocos de Computação Gráfica em Hardware

O mercado de hardware para computação gráfica existente no final da década de 80 até o final da década de 90 foi muito lucrativo para as empresas que investiram no desenvolvimento de placas de video para computadores pessoais. Com a crescente popularização do mesmo, a tendência dos jogos eletrônicos baseados em computadores e a grande demanda por multimídia na internet em canais antes apenas textuais, impulsionou e estabeleceu várias empresas hoje líderes nesse mercado como a NVIDIA, ATI Technologies, Silicon Graphics, dentre outras.

A partir do final da década de 90, houve um crescimento extremamente elevado dos celulares e dos PDAs (do Inglês, *Personal Digital Assistent*). O celular, por exemplo, passou de um aparelho utilizado apenas para comunicação para um equipamento que provê informações e entretenimento.

No início do lançamento dos celulares e PDAs, a maior parte destes equipamentos possuia apenas funções básicas de processamento e de computação gráfica em duas dimensões com simples desenhos em formato de mapas de bits e telas orientadas a caracteres monocromáticos. Hoje em dia, coprocessadores gráficos complexos e telas com resoluções de 16 bits por pixel já são comuns neste mercado.

Diversas empresas como a ATI Technologies, MediaQ, Seiko-Epson e NeoMagic competem para conseguir a sua fatia nesse mercado de co-processadores gráficos para dispositivos móveis. Enquanto isso, a Sony, a Texas Instruments e a Intel fornecem as suas soluções integradas como a arquitetura OMAP e a arquitetura *Personal Internet*

Client. Tomando um caminho diferente, empresas como a Falanx [46] já disponibilizam blocos gráficos em hardware como propriedade intelectual para outras empresas integrarem com seus projetos no paradigma *System-on-Chip*.

Essa grande tendência para o desenvolvimento de aceleradores, co-processadores e blocos de computação gráfica em silício oferece uma vasta possibilidade de estudo e pesquisa. A aplicação dos conhecimentos de projetos de circuitos integrados ao desenvolvimento de blocos gráficos não é recente. Já que plataformas gráficas são criadas desde a década de 80. Todavia, este problema reaparece com novas restrições de tempo, com uma maior escala de integração do *chips*, com novas metodologias de projeto, e, principalmente, novas técnicas e métodos de verificação de circuitos integrados.

3.1.2 Trabalhos Relacionados

As metodologias utilizadas para se verificar blocos de computação gráfica em hardware são as mesmas utilizadas para qualquer outro tipo de bloco[1]. Nestes casos, a verificação funcional é utilizada seja por uma abordagem caixa branca, caixa preta ou caixa cinza. Na maior parte dos casos, apenas um visualizador da área de memória onde são desenhadas as primitivas é utilizado.

Por meio da verificação funcional, uma série de estímulos devem ser aplicados ao DUV e as suas saídas devem ser comparadas com um modelo de referência para garantir que o mesmo está correto. Como argumentado por Bergeron [2], decidir como aplicar os estímulos de entrada é um trabalho relativamente fácil, pois o desenvolvedor tem o controle dos dados a serem aplicados e da temporização. A tarefa mais difícil consiste em verificar as saídas, ou seja, garantir que a saída obtida equivale a resposta esperada de acordo com a especificação.

Os métodos tradicionais solucionam este problema de duas formas principais. A primeira consiste em uma inspeção visual dos resultados de saídas sejam eles numéricos ou em um formato padrão que possa ser observado através de um visualizador de formas de onda. A segunda forma consiste em criar módulos que possam fazer a verificação automática das respostas de saída. Para ambas abordagens, um conjunto de variáveis de saída é produzido por um modelo em um nível mais alto de abstração e, em seguida, comparado com os resultados obtidos pela implementação.

Para a verificação de blocos de computação gráfica em hardware, o problema de verificar a saída automaticamente torna-se mais complexo. O trabalho de se produzir um caso de teste automático para se verificar uma resposta que pode ser rapidamente reconhecida como certa ou errada por inspeção visual pode ser difícil [2]. Bergeron [2] ainda argumenta que nestes casos, uma inspeção visual pode ser muito mais eficiente

[1] Estes métodos estão descritos em maior detalhe na seção 2.2.

em reconhecer, por exemplo, se um círculo vermelho foi desenhado corretamente. Mas uma boa estratégia de verificação deve encontrar maneiras de se automatizar estes processo.

Assim, pode-se perceber que os blocos de computação gráfica em hardware ainda são verificados utilizando-se verificação funcional, com verificações automáticas para informações no nível RT e inspecionadas visualmente para níveis mais altos de abstração.

3.1.3 Verificação Automática em Alto Nível

O método citado anteriormente afirma que a verificação de uma saída que gera um círculo vermelho, por exemplo, pode ser feita de forma mais eficiente se esta for verificada por inspeção visual. Isto porque a criação de um caso de teste para verificar automaticamente esta saída é muito difícil. Contudo, algumas limitações da inspeção visual para uma saída de um bloco de computação gráfica em hardware podem ser destacadas:

- é extremamente difícil garantir que uma primitiva foi desenhada com o tom da cor especificada;
- o deslocamento de um ou mais pixel em uma direção são raramente identificados por inspeção visual;
- este tipo de verificação exige treinamento e a sua eficiência pode variar acordo com o estado psicológico do engenheiro de verificação.

Por estes motivos, o presente trabalho propõe um arquitetura para verificação automática de blocos de computação gráfica em hardware. Esta arquitetura visa obter vantagens das particularidades destes tipo de hardware para tornar o processo de verificação mais eficiente. Esta arquitetura foi dividida em quatro estágios para facilitar o processo de verificação e estes são detalhados nas seções seguintes.

3.2 Estágios da Arquitetura

A arquitetura de verificação proposta neste trabalho foi dividida em quatro estágios, estes são:

1. a implementação da especificação executável em alto nível de abstração;
2. a implementação do bloco em hardware em HDL;
3. a verificação funcional por sub-blocos; e

4. a verificação funcional em nível de sistema[2].

A especificação é o primeiro estágio de qualquer projeto de circuito integrado. Nesta fase, os projetistas definem o que o projeto de circuito integrado deve cumprir tanto em termos funcionais quanto em não funcionais. O primeiro estágio desta arquitetura contempla a definição da especificação para os requisitos funcionais do projeto. Esta especificação deve ser fornecida através de uma implementação em alto nível de abstração. Para os requisitos não funcionais como custo, potência, freqüência, dentre outros, a especificação em documento pode ser utilizada.

O segundo estágio consiste na implementação do bloco de computação de computação gráfica em linguagem de descrição de hardware. Neste estágio, asserções devem ser instanciadas a partir de propriedades da especificação executável.

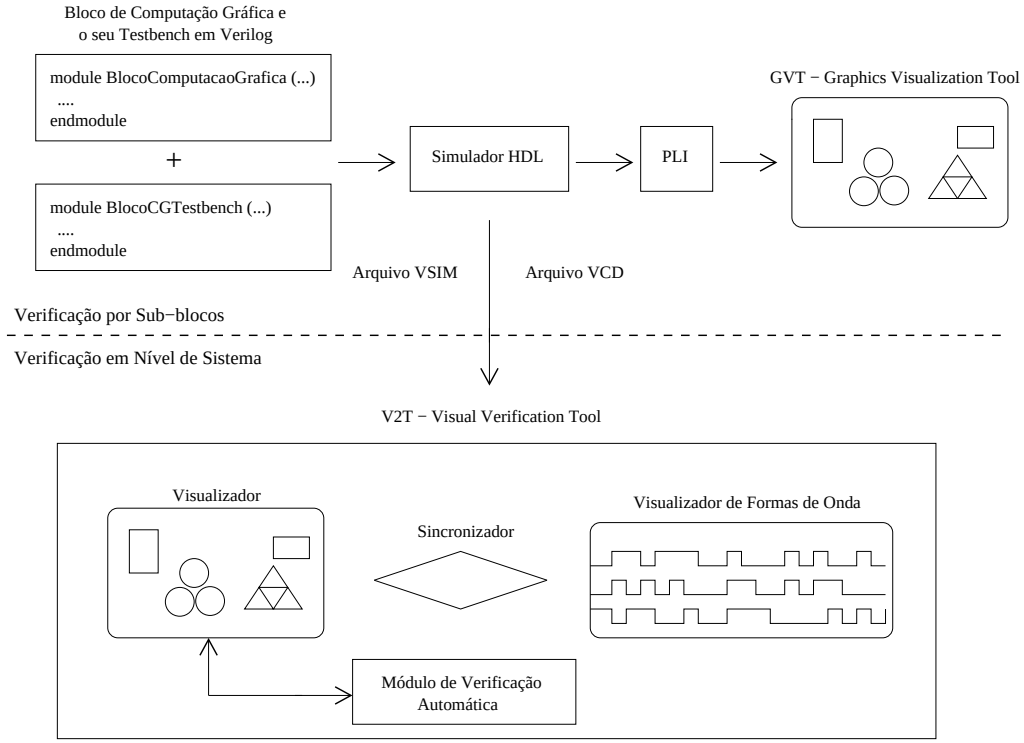

Figura 3.1: Estágios de verificação da arquitetura proposta.

Os próximos estágios da arquitetura são os níveis de abstração para a verificação. O primeiro nível visa garantir que um sub-bloco implementado em linguagem de descrição de hardware atende aos requisitos funcionais definidos na especificação. Já o segundo nível de verificação só se inicia ao término no primeiro. Ou seja, após garantir que cada um dos sub-blocos que formam o sistema está correto, pode-se verificar a interação

[2]O termo *verificação funcional em nível de sistema* geralmente é utilizado para denominar o processo de verificação de múltiplos blocos funcionais ou ASICs em um sistema. No presente trabalho, este termo têm um significado um pouco diferente pois refere-se ao processo de verificação da integração de sub-blocos funcionais.

entre os sub-blocos e verificar se o sistema como um todo atende os requisitos da especificação.

A Figura 3.1 mostra os estágios de verificação da arquitetura proposta. Esta figura foi dividida em duas partes delimitadas pela linha tracejada. A parte superior desta figura representa a verificação dos sub-blocos e a parte inferior representa a verificação em nível de sistema. Além disso, existe uma outra distinção importante. A verificação dos sub-blocos é feita pela análise dos resultados das primitivas de computação gráfica em uma ferramenta gráfica, denominada de *GVT* (do Inglês, *Graphics Visualization Tool*), no momento que a simulação é executada. Enquanto a verificação em nível de sistema é feita a partir de arquivos gerados na simulação e depois carregados em uma ferramenta CAD, denominada V^2T (do Inglês, *Visual Verification Tool*). As seções seguintes detalham cada um dos estágios da arquitetura.

3.3 Definição e Implementação da Especificação Executável

A definição da especificação executável é o estágio para projeto e verificação com a arquitetura proposta. A especificação executável desempenha dois papéis importantes. O primeiro destes é oferecer informações e antever dificuldades da futura implementação em RTL. O segundo, e talvez o mais importante, é servir de modelo de referência para a implementação em RTL e possibilitar uma verificação automática através da sua integração com a ferramenta V^2T.

Especificações executáveis são muito utilizadas em projetos de circuitos integrados. Segundo Chang et al. [47], uma especificação executável consiste em capturar os requisitos de um produto ou circuito integrado em termos de funcionalidade e desempenho[3]. Uma especificação executável pode ser entendida como um modelo abstrato para representar tanto o hardware quanto o software.

Estes modelos podem ser utilizados para diferentes atividades e em diversos níveis de abstração como os destacados abaixo:

- modelos de desempenho auxiliam no entendimento e no estabelecimento de requisitos funcionais e não-funcionais de um produto;
- modelos funcionais são elaborados para prover uma especificação executável para os produtos;
- modelos para projeto e síntese surgem de um refinamento e um compromisso entre os modelos de desempenho e os modelos funcionais; e

[3]Neste trabalho, apenas os requisitos funcionais serão analisados.

- os modelos para verificação e validação ajudam garantir que a implementação está de acordo com a especificação.

Assim, modelos ou especificações executáveis são freqüentemente utilizados pelos simples motivo de não se possuir, até o termino do ciclo de desenvolvimento, o produto[4].

As especificações executáveis são muito úteis para se descrever comportamentos funcionais na maior parte dos projetos. Para as especificações de circuitos integrados em um nível mais alto de abstração pode-se utilizar C, C++, SDL, System C, dentre outras. Além de servir como uma referência para se confrontar com a implementação, uma especificação executável permite aos projetistas verificar de forma prévia as funcionalidades básicas e as interfaces entre hardware e software.

Como a especificação executável é muito importante para a arquitetura de verificação proposta, foi necessário um cuidado maior na definição do seu modelo. Para o intuito deste trabalho, um modelo de um co-processador de computação gráfica foi desenvolvido. Optou-se por desenvolver o modelo em linguagem de programação C e a biblioteca gráfica bastante popular existente no sistema operacional Linux que é a biblioteca GTK [49]. A princípio, qualquer outra linguagem de programação em alto nível poderia ser utilizada e o mesmo vale para a biblioteca gráfica pois a arquitetura proposta não depende das mesmas. A Figura 3.2 mostra como é o fluxo na especificação executável. Estímulos de entrada são aplicados a especificação executável com a biblioteca gráfica e os resultados são visualizados por uma ferramenta gráfica.

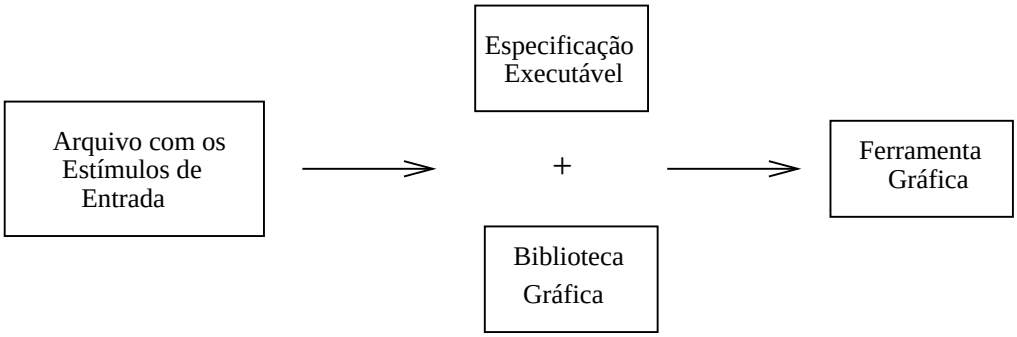

Figura 3.2: Fluxo para a especificação executável.

Esta especificação executável contém a implementação de todos sub-blocos do co-processador, como o bloco desenho de linhas, triângulos, retângulos e círculos, preenchimento de círculos, triângulos, retângulos, dentre outros.

[4]Uma discussão detalhada sobre o conceito de modelo pode ser encontrada em Janstch [48].

3.4 Implementação do Bloco de Computação Gráfica em Hardware

A implementação do bloco de Computação Gráfica em hardware representa o segundo estágio da arquitetura proposta. Nenhuma consideração especial ou restrição precisa ser feita neste estágio. Apenas que o bloco deve ser implementado em uma linguagem de descrição de hardware visando satisfazer a especificação executável implementada no estágio anterior[5].

3.5 Verificação por Sub-blocos

3.5.1 Extensão da Metodologia Tradicional

O modelo de verificação funcional descrito na seção 2.2 segue os seguintes passos:

1. criam-se casos de testes para cobrir a maior parte das funcionalidades do sub-bloco a ser verificado;
2. adiciona-se um mecanismo para verificar e relatar a cobertura do sub-bloco em questão;
3. aplica-se o caso de teste como entrada do DUV; e
4. adicionam-se blocos lógicos no próprio caso de testes ou externamente para se verificar a conformidade dos resultados de saída.

A arquitetura proposta neste trabalho amplia a metodologia tradicional em relação a verificação por sub-blocos. Além da execução de todos os passos enumerados anteriormente, uma verificação em um nível mais alto de abstração também é feita.

Em relação ao primeiro passo, se o bloco de computação gráfica for um co-processador, os estímulos podem ser gerados em um nível de abstração de conjunto de instruções. Com a criação de casos de teste em níveis mais altos de abstração, o processo de verificação torna-se mais rápido e ainda possibilita o seu reuso na verificação em nível de sistema. Mas nada impede que estímulos em um nível de abstração mais baixo sejam gerados para verificar blocos que não sejam co-processadores.

O mecanismo de cobertura de código é um passo importante neste processo de verificação. Todavia, diversas ferramentas comerciais já estão disponíveis para executar este trabalho como a VHDLCover, VeriSure, VeriCov, CoverMeter, dentre outras. Desta forma, esta discussão não será aprofundada.

[5] O capítulo 4 fornece uma descrição bem detalhada sobre o projeto deste bloco, mostrando inclusive os algoritmos implementados em linguagem de descrição de hardware.

O próximo passo consiste apenas na aplicação dos estímulos de entrada. Já o passo seguinte, consiste em verificar se as saídas geradas estão de acordo com a especificação. Isto pode ser feito através asserções definindo propriedades da especificação no projeto de circuito integrado.

O novo passo inserido pela arquitetura proposta no processo de verificação, aproveita-se das particularidades dos blocos de computação gráfica em hardware. Ou seja, a arquitetura fornece um mecanismo para a visualização das primitivas de computação gráfica geradas. Em detrimento de se verificar as saídas apenas em um nível mais baixo de abstração como é feito na arquitetura tradicional (passo 4), a ferramenta gráfica *GVT* desenha as primitivas de computação gráfica em tempo de simulação. Esta ferramenta extrai as informações do simulador através da interface PLI (do Inglês, *Programming Language Interface*) descrita na seção 3.5.2. Com esta extensão proposta pela arquitetura, é possível verificar por inspeção visual se as primitivas desenhadas correspondem às especificadas. Além disso, é possível executar o mesmo caso de testes na especificação executável e na implementação comparando-se graficamente os resultados obtidos.

A verificação dos sub-blocos implementados em linguagem de descrição de hardware segue um fluxo bastante simples. Este fluxo é mostrado na Figura 3.3. Primeiramente, criam-se casos de testes com o intuito de exercitar todo o sub-bloco. Em seguida, submete-se a implementação e os casos de testes a um simulador. Por meio das asserções, as propriedades são verificadas em um nível mais baixo de abstração. Através da interface PLI, a memória de tela é monitorada e funções são definidas para que estes dados possam ser alcançados por programas externos. Finalmente, a ferramenta gráfica *GVT* obtém os dados da simulação representando as primitivas gráficas. Isto possibilita uma verificação por inspeção visual em um nível de abstração mais alto.

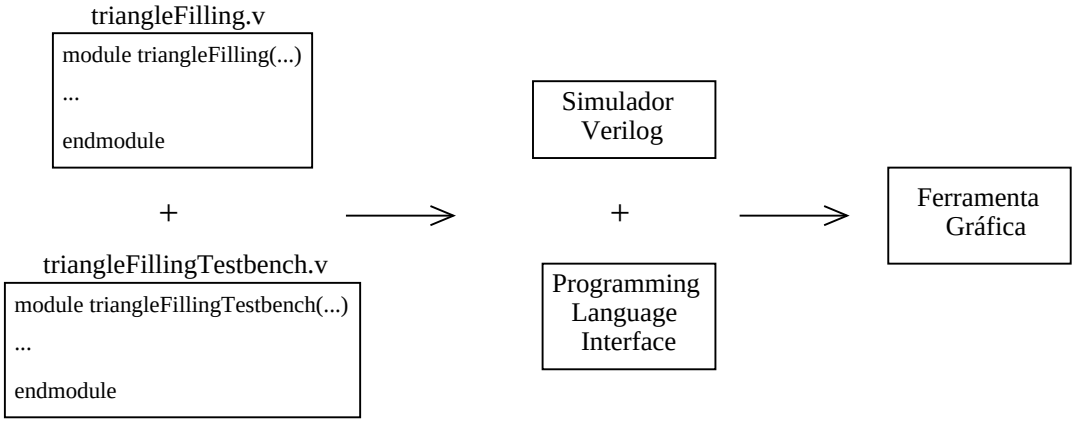

Figura 3.3: Fluxo para a verificação dos sub-blocos.

Devido a relevância da interface PLI e da ferramenta de visualização gráfica *GVT* neste estágio da arquitetura, breves introduções de ambas serão feitas nas próximas seções.

3.5.2 Interface de Linguagem de Programação - PLI

A linguagem de descrição de hardware Verilog tornou-se padrão do IEEE em 1995. Esta linguagem foi projetada com objetivos de ser simples, intuitiva e efetiva em diversos níveis de abstração. Esta foi projetada para ser utilizada nos diferentes estágios seja o de simulação, o de análise de tempo ou o de síntese lógica.

A linguagem Verilog foi projetada para ser extensível através de interfaces como a PLI e a VPI (do Inglês, *Verilog Procedural Interface*). A PLI e a VPI são coleções de rotinas que possibilitam funções externas obterem acesso a informações contidas no HDL do projeto de circuitos integrados tornando mais fácil a interação dinâmica com a simulação.

Estas interfaces existentes no Verilog possibilitam também a verificação dinâmica de propriedades com asserções definidas em PLI [5]. Isto possibilita uma maior flexibilidade na definição das asserções e um maior nível de abstração na definição das mesmas, mas pode impactar na simulação tornando-a de 2 a 10 vezes mais lenta. Enquanto asserções definidas em nível RTL geram um impacto de 10% a 30% no tempo total de simulação.

Neste trabalho, as interfaces existentes no Verilog foram utilizadas para fornecer, em um nível de abstração mais alto, uma visualização das primitivas gráficas através da ferramenta *GVT*.

3.5.3 Ferramenta de Visualização Gráfica - *GVT*

A ferramenta *GVT* é um software desenvolvido com a biblioteca gráfica GTK e a linguagem de programação C. Este software mostra em tempo real da execução do simulador cada pixel inserido na tela, a medida que estes são escritos no *frame-buffer*. Isto é possível porque uma *thread* é definida de forma a atualizar a tela sempre que o espaço de memória de tela for alterado. Este compartilhado de memória entre o simulador em tempo de execução só é possível através das interfaces PLI e VPI.

3.6 Verificação em Nível de Sistema

A verificação em nível de sistema tem como objetivo garantir que os sub-blocos já verificados não apresentam nenhuma inconsistência com os outros sub-blocos quando todos são verificados em conjunto. Geralmente, em grandes projetos, os sub-blocos são projetados por equipes diferentes. Estas equipes podem produzir sub-blocos baseando-se em uma mesma especificação, verificá-los individualmente (seção 3.5) e estes não interagirem entre si. Dessa forma, a verificação em nível de sistema aparentemente simples já que muitos dos sub-blocos já foram verificados, torna-se a mais trabalhosa e complexa. A Figura 3.4 mostra as fases de um projeto de forma simplificada supondo

que um projeto de circuito integrado será dividido em dois sub-blocos que poderão se interagir.

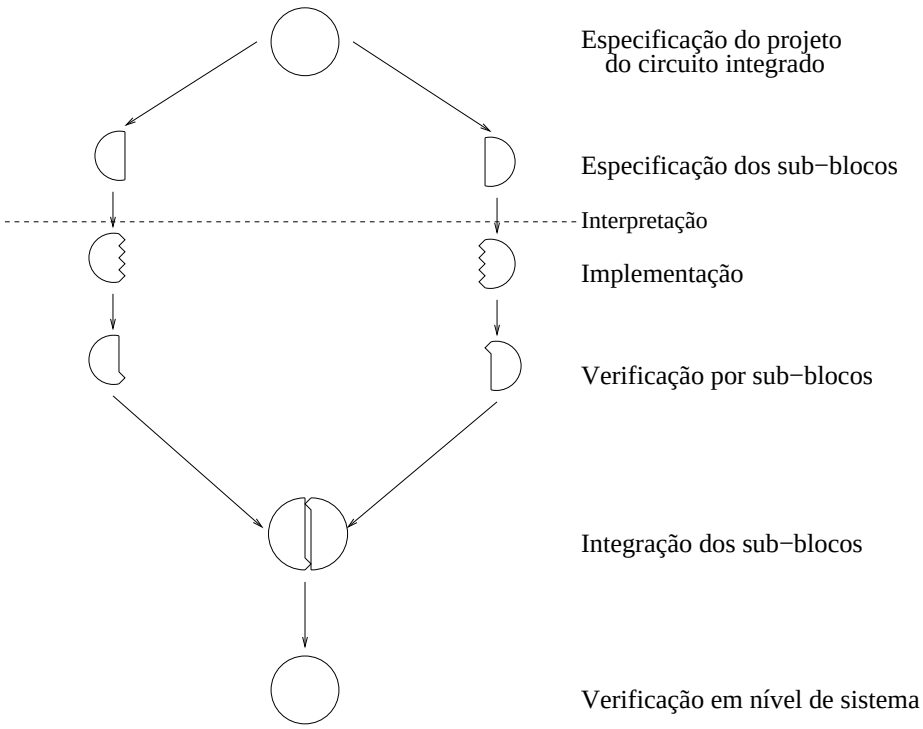

Figura 3.4: Fases do projeto para verificação por sub-blocos e em nível de sistema.

Na Figura 3.4 mostra a primeira fase que é a de especificação. A partir dela, dois grupos são designados para desenvolver dois sub-blocos do sistema e recebem um sub-conjunto da especificação correspondente ao seu sub-bloco. A partir destes sub-conjunto, cada grupo faz a sua devida interpretação do documento e elabora a implementação. Em seguida, os sub-blocos elaborados são verificados. O próximo passo consiste na integração dos sub-blocos para se formar o sistema. Finalmente, uma verificação em nível de sistema é feita.

Pode-se observar que as equipes fizeram interpretações do documento separadamente. Este é a principal motivaço da verificação em nível de sistema, ou seja, garantir que a interpretação em relação a interação dos sub-blocos feitos por duas ou mais equipes esteja correta. Este é um dos problema mais recorrente na verificação de circuitos integrados, daí a necessidade da verificação em nível de sistema.

3.6.1 Extensão da Metodologia Tradicional

A abordagem proposta nesta arquitetura difere-se bastante da abordagem feita pela metodologia tradicional de verificação funcional. Esta metodologia, como mostrado na seção 2.2, requer que um caso de teste seja gerado, aplicado ao DUV e que os resultados de saída sejam comparados automaticamente. Na Figura 2.1, tanto o gerador de estímulos para o DUV quanto o comparador dos resultados de saída estão inseridos

dentro do caso de teste por isso não são representados. Para explicitar a diferença da abordagem proposta, optou-se por retirar o comparador de resultados de saída da figura que representa o caso de testes.

Na abordagem proposta pela arquitetura de verificação do presente trabalho, os seguintes passos são seguidos:

1. estímulos são gerados para exercitar a maior parte das funcionalidades existentes no sistema;
2. mecanismos para verificar e relatar a cobertura são inseridos;
3. aplica-se o caso de teste como entrada do DUV;
4. uma comparação dos resultados de saída em um nível de abstração mais baixo é feita automaticamente; e
5. em alto nível de abstração, os resultados de saída também são comparados automaticamente.

A Figura 3.5 mostra os ítens descritos na enumeração anterior através de blocos. Além disso, a figura mostra o comparador das saídas em um nível de abstração mais baixo que é feito pelas asserções, de maneira similar a descrita na verificação por sub-blocos. A figura também mostra a verificação em alto nível de abstração que é feita através da comparação entre as primitivas gráficas geradas pela especificação executável e pela implementação.

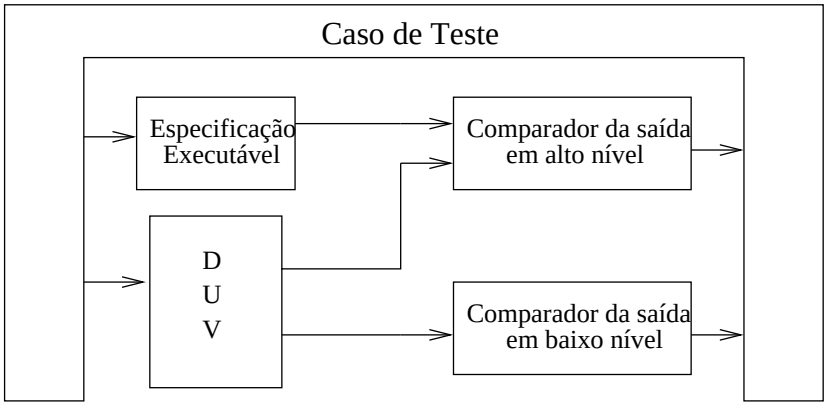

Figura 3.5: Verificação em nível de sistema em dois níveis de abstração.

De acordo com a metodologia tradicional de verificação, espera-se que todo o fluxo de informação descrito na Figura 3.3 ocorra em tempo de simulação. No entanto, uma abordagem diferente foi proposta. Isto acontece porque a execução do simulador de linguagem de descrição de hardware em paralelo com o comparador de saídas em alto nível deixaria o processo de simulação muito lento. Além disso, a interface PLI teria

que ser utilizada para fornecer informações ao comparador deixando o processo ainda mais lento.

Outro ponto a ser ressaltado é que não é necessária um comparação ciclo a ciclo para as primitivas de computação gráfica, ou seja, basta comparar funcionalmente o resultado final da primitiva. Além disso, é conveniente armazenar as informações da simulação porque se algum dos comparadores acusar algum erro será necessário voltar ciclos de *clock* na simulação para identificá-lo. Desta forma, optou-se por analisar as informações obtidas na simulação somente após o término da mesma.

3.6.2 Geração de Estímulos

Na verificação por sub-blocos, os casos de testes foram criados no nível de abstração do conjunto de instruções. Na verificação em nível de sistema, a mesma situação ocorre e esta reaproveita parte dos casos de testes criados na verificação por sub-blocos.

Os três principais objetivos da verificação em nível de sistema são o de exercitar *corner-cases*, verificar a interação e as interfaces entre sub-blocos e verificar situações não previstas através de estímulos aleatórios. Para alcançar estes objetivos, três grupos de casos de teste devem ser criados.

O primeiro grupo de estímulos deve ser criado para exercitar *corner-cases*. Estes estímulos tentam encontrar cenários complexos que são mais prováveis de levar o DUV a condições de erro. Para o bloco de computação gráfica em hardware em questão, deve-se criar, por exemplo, estímulos para que primitivas tenham coordenadas fora da tela, coordenadas idênticas para o desenho de triângulo, desenhos de círculos com o valor do raio igual a zero, dentre outras. Este processo de geração de caso de testes não pode ser automatizado, pois depende da criatividade do engenheiro de verificação.

O segundo grupo deve ser utilizado para fomentar a comunicação entre os sub-blocos. Como cada sub-bloco implementado geralmente relaciona-se com uma primitiva, diversas primitivas devem ser executadas em diferentes ordens para estimular troca de informações. Os sub-blocos que estabelecem comunicação com muitos outros devem ser bastante exercitados. Um destes é o de desenho de triângulo, que, se implementado de maneira eficiente, utilizará outros sub-blocos como o de desenho de linha. Assim, problemas entre as interfaces e a interação dos mesmos podem ser detectados. Como o primeiro grupo, estes estímulos não podem ser automatizados.

O último grupo de estímulos só é possível porque a arquitetura proposta possibilita uma verificação automática das saídas. O grupo de estímulos aleatórios visa exercitar cenários que os engenheiros de verificação não anteciparam nos dois primeiros grupos de estímulos. O principal objetivo deste grupo de estímulos é exercitar casos ainda não verificado e aumentar a cobertura de código do projeto. A geração destes estímulos pode ser totalmente automatizada. Por exemplo, se um co-processador de computação

gráfica for verificado, basta criar programas que gerem seqüências de instruções a partir do conjunto de instruções.

3.6.3 Verificação Automática das Saídas

Como argumentado anteriormente, gerar estímulos não é a tarefa mais difícil da verificação; mas sim garantir que as saídas geradas estão corretas. Na arquitetura proposta neste trabalho, a verificação é feita tanto em baixo nível de abstração quanto em alto nível de abstração.

Da mesma forma utilizada na verificação por sub-blocos, a verificação em nível de sistema é feita em um nível mais baixo de abstração através de propriedades extraídas da especificação com o auxílio das bibliotecas de asserções. Exemplos de propriedades a serem verificadas são dos tipos: se um padrão está dentro de uma faixa pré-determinada, detecção de bits não conectados, se um opcode inválido está no sub-bloco de controle, garantia de seqüência de padrões, dentre outros.

Já a verificação em um nível mais alto de abstração, visa prover uma melhor interpretação dos padrões a serem verificados. A arquitetura proposta fornece um mecanismo de verificação das primitivas de saída. Isto é feito através de uma comparação automática entre a especificação executável e a implementação.

Um exemplo deste processo seria: o engenheiro de verificação cria casos de teste em nível de conjunto de instruções para verificar a implementação. Neste nível de abstração, os estímulos podem ser aplicados tanto na especificação executável quanto na implementação. Se a primeira instrução for a de desenho de triângulo, por exemplo, tanto a implementação quando a especificação executável desenharão esta primitiva. A partir daí, basta comparar os resultados gráficos de ambas automaticamente.

Esta verificação em dois níveis de abstração não é feita em tempo de simulação, mas sim após a mesma por meio das informações coletadas, por vários motivos:

- sempre que um erro é detectado por algum dos mecanismos verificadores é possível reconstruir o cenário a partir dos dados coletados;
- a comparação ciclo a ciclo entre as primitivas desenhadas na especificação executável e as desenhadas na implementação não pode ser feita pois estes estão em dois níveis de abstração diferentes; e
- o processo de comparação automática em tempo de execução pode deixar o mecanismo de verificação lento.

Para se fazer uma verificação por meio das informações coletadas na simulação, a ferramenta V^2T foi desenvolvida. Além de fazer a verificação em dois níveis de abstração, a ferramenta fornece uma melhor visualização dos dados e esta é detalhada na próxima seção.

3.6.4 A Ferramenta V^2T

Funcionamento

A ferramenta V^2T foi desenvolvida com o intuito de auxiliar o processo de verificação em nível de sistema analisando as informações coletadas na simulação. Esta é formada por três módulos que são: o primeiro é responsável pela visualização de formas de onda, o segundo pela visualização das primitivas gráficas e o último é responsável por verificar automaticamente comparando as primitivas da implementação com as da especificação. Esta ferramenta necessita de três arquivos de entrada, são estes:

- um arquivo em um formato VCD com todos os sinais utilizados na verificação em baixo nível de abstração, como indicadores de asserções violadas, sinais vitais do *datapath*, dentre outros;
- um arquivo com as propriedades de cada pixel escrito na memória de tela e seus atributos vinculados com o tempo de simulação fornecido pelo simulador, denomindado de VSIM; e
- um arquivo com o caso de teste que gerou os dois primeiros arquivos para ser utilizado no modulo de verificação.

Por meio destes arquivos, a ferramenta fornece dois mecanismos importantes: a integração entre dois níveis de abstração e a verificação gráfica e automática.

A integração entre os dois níveis de abstração é possível porque os arquivos VSIM e VCD armazenam as primitivas e o sinais em RTL vinculados com o tempo, respectivamente, como mostra a Figura 3.6. Este mecanismo torna o processo de depuração muito mais eficiente. Já que, selecionada uma janela de tempo Δt, é possível detectar, por exemplo, qual pulso de *write enable* na memória dos sinais em RTL ocasionou o desenho de um pixel errado na primitiva gráfica (um caso real de verificação por meio deste mecanismo é mostrado na seção 5.2.1).

A Figura 3.6 mostra, além dos gráficos com as informações fornecidas pelos arquivos VSIM e VCD, um outro gráfico denominado de Asserções. Este gráfico mostra se alguma asserção foi violada durante a execução do caso de teste e se esta ocasionou um erro em uma primitiva gráfica (ver Figura 3.6). Este processo facilita ainda mais o processo de verificação do bloco de computação gráfica (ver detalhes na seção 5.2.2). É importante ressaltar que a violação de uma asserção pode ou não causar um erro na primitiva gráfica, isto dependerá do tipo de asserção instanciada.

O mecanismo de verificação gráfica e automática é executado quando o usuário o solicita através de uma ação na interface da ferramenta. Inicialmente, a ferramenta V^2T carrega, por meio do arquivo VSIM, as primitivas desenhadas na simulação em sua tela. Ao iniciar a verificação, cada instrução de cada primitiva do caso de teste é

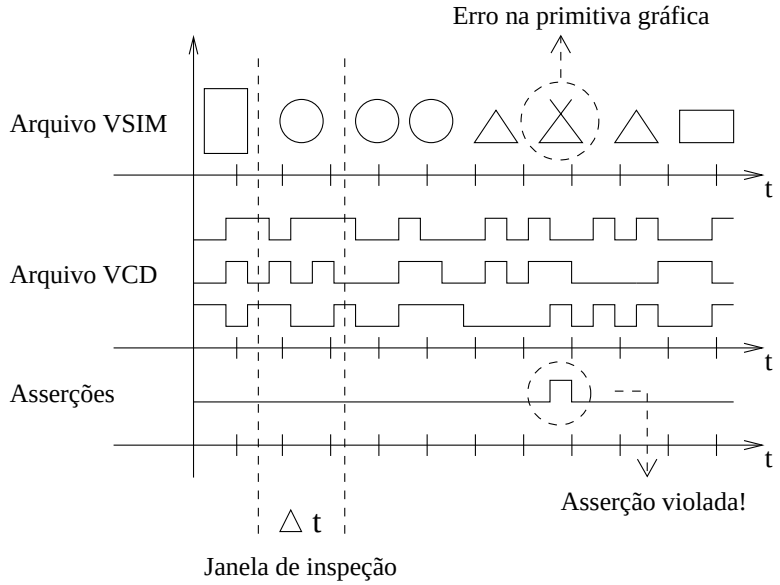

Figura 3.6: Mecanismo de integração entre dois níveis de abstração.

executada pela especificação executável embutida dentro da ferramenta V^2T. Como as instruções são primitivas gráficas, cada uma destas escreve na tela pixel por pixel algum desenho. Se durante a execução de uma destas instruções algum pixel não estiver na tela, significa que esta mesma instrução não foi executada de forma correta durante a simulação. Já que o desenho da tela é o resultado das primitivas gráficas da implementação em RTL. Por isso, a ferramenta mostra que esta instrução esta errada na implementação (ver detalhes na seção 5.2.3).

Além disso, após a verificação automática, a ferramenta possibilita que o usuário efetue diferenças visuais entre as primitivas da especificação e implementação facilitando a visualização dos pixels que não estão de acordo (ver detalhes na seção 5.2.3).

Implementação

Como descrito anteriormente, a ferramenta V^2T necessita de um visualizador de formas de onda e de um visualizador gráfico para mostrar as primitivas de saída do bloco de computação gráfica em hardware. Com o intuito de aprofundar no mérito do trabalho que é propor, analisar e validar a arquitetura de verificação proposta, optou-se por utilizar um visualizador de forma de onda já existente integrando-o a V^2T.

Para que este pudesse ser integrado com o visualizador das primitivas e o módulo verificador, foi necessário que o mesmo fosse código aberto. Dentre os visualizadores existentes com estes requisitos destacam-se o Dinotrace [50] e o GTKWave [15]. O GTKWave foi escolhido porque possui um código fonte mais bem documentado, o que facilita o trabalho de integração com os outros módulos. Além disso, este software possui uma equipe de desenvolvimento bem ativa o que facilita o intercâmbio de informações e a resolução de problemas.

O visualizador gráfico da ferramenta V^2T foi desenvolvido com a biblioteca gráfica GTK. Já o módulo verificador nada mais é do que a própria especificação executável desenvolvida no primeiro estágio da arquitetura.

Para a geração dos arquivos necessários para a ferramenta V^2T, o simulador escolhido foi o GPL CVER [51] que é um simulador de Verilog HDL distribuído sobre a licença GPL. Este simulador implementa todas as funcionalidades definidas pelo padrão IEEE 1364 de simuladores Verilog. Além disso, o mesmo implementa a maioria das características da revisão do padrão Verilog divulgada em 2001.

Esta seção fornece apenas uma introdução e conceituação da necessidade e algumas características da ferramenta V^2T. Exemplos de sua aplicação são mostrados no capítulo 5.

3.7 Benefícios da Arquitetura Proposta

Dentre os benefícios da arquitetura proposta, pode-se destacar:

- a divisão em dois níveis de verificação em sub-blocos e em nível de sistema reduz o tempo de verificação. Primeiramente, porque baseia-se na abordagem de dividir para conquistar, reduzindo, assim, a complexidade da verificação. Além disso, fornece uma verificação em nível de sistema para detectar principalmente erros relativos a interfaces e *corner-cases*.

- Melhora a interpretação dos dados, pois fornece a visualização das primitivas de computação gráfica tanto na verificação por sub-blocos quanto em nível de sistema.

- Fornece uma verificação automática em alto nível de abstração para a verificação em nível de sistema.

- Possibilita, através da ferramenta V^2T, que um erro encontrado automaticamente na visualização gráfica possa ser rapidamente encontrado no RTL.

- Suaviza a passagem da simulação para o protótipo pois as ferramentas V^2T e GVT disponibilizam as saídas das primitivas gráficas, tanto em tempo de simulação, quanto no término da simulação.

- Verifica em alto nível de abstração mantendo observabilidade e controlabilidade.

- Pode ser incluída no fluxo tradicional de verificação de projeto de circuitos integrados.

Capítulo 4

Bloco de Computação Gráfica em Hardware

Este capítulo tem como objetivo detalhar o desenvolvimento dos objetos utilizados para validar a arquitetura de verificação proposta. O principal objeto foi a implementação de um bloco de computação gráfica usando uma linguagem de descrição de hardware. Outro objeto implementado foi a especificação executável para este bloco. Cada seção deste capítulo representa os sub-blocos implementados tanto na especificação executável quanto na implementação.

4.1 Bloco de Computação Gráfica em Hardware

Para validar a arquitetura de verificação proposta, foi necessário especificar e implementar um bloco de computação gráfica em hardware. O objetivo desta implementação não foi o de se projetar um processador gráfico que contivesse todas as funcionalidades existentes em processadores utilizados em placas de vídeo de computadores pessoais. De forma bem sucinta, a proposta consiste em implementar funcionalidades existentes em celulares e PDAs (do Inglês, *Personal Digital Assistent*) como as primitivas de computação gráfica em duas dimensões.

Existem formas distintas de se implementar um bloco de computação gráfica em hardware. Pode-se optar por um processador completo, um acelerador ou um co-processador. Um processador completo implementaria todas as funcionalidades necessárias para o desenho das primitivas de computação gráfica e, ainda possibilitaria, a implementação de algoritmos para processamentos de imagens. Um acelerador gráfico trabalharia em conjunto com um processador principal recebendo tarefas deste último. O acoplamento entre ambos é fraco porque mecanismo de comunicação existente entre estes é geralmente feito através de um barramento. Co-processadores, diferentemente dos anteriores, são fortemente acoplados. Geralmente possuem acesso aos registradores

internos do processador principal compartilhando tarefas.

Optou-se por desenvolver um co-processador em detrimento das outras duas possibilidades: processador e acelerador. Pois, a implementação um processador gráfico com todas as funcionalidades existentes demandaria muito tempo; e a implementação de um acelerador geralmente necessita de um protocolo para comunicação com o processador principal e a verificação deste desviaria o foco do presente trabalho. Além disso, o bloco de computação gráfica em hardware aqui implementado é apenas um meio para se validar a arquitetura proposta. Por isso, optou-se pela forma mais simples, a implementação de um co-processador de primitivas de computação gráfica.

As principais motivações para se implementar um co-processador foram:

- a facilidade de comunicação entre o processador principal e o co-processador seja por meio de compartilhamento de registradores ou FIFO de instruções;
- facilidade de integração deste co-processador com blocos de processadores em hardware distribuídos pelas principais empresas dispositivos programáveis como Altera [52] e Xilinx [53]; e
- a maior eficiência na utilização dos recursos possibilitados pela implementação de um co-processador destinado apenas para tarefas de computação gráfica.

A empresa Altera fornece um bloco em hardware de um processador configurável, com uma arquitetura RISC e um conjunto de instruções de 16 bits ou 32 bits chamada Nios [52]. Segundo a própria empresa, este bloco em hardware pode trabalhar em uma freqüência de mais de 180 MHz ocupando apenas 1000 elementos lógicos em uma FPGA.

A empresa Xilinx também fornece um bloco em hardware de um processador RISC de 32 bits com pipeline e que pode executar em uma freqüência de até 150 MHz denominado Microblaze [53]. Este processador foi especialmente projetado para a utilização em telecomunicações e serviços de rede. Baseado em uma arquitetura Harvard, este processador possui memória de instruções separada da memória de dados ambas com barramentos de 32 bits.

Assim, para que o co-processador desenvolvido pudesse ser integrado com um processador configurável seja da Altera ou da Xilinx, optou-se por padronizar uma FIFO para a entrada de instruções, esta é descrita na seção 4.6.1.

4.2 Divisão em Sub-Blocos

O co-processador desenvolvido foi dividido em sub-blocos para facilitar a implementação e a verificação. As divisões foram feitas baseando-se em unidades funcionais como mos-

trado na Figura 4.1. Existem três divisões básicas em relação aos tipos de sub-blocos, estes são:

- os sub-blocos de primitivas de desenhos de computação gráfica em duas dimensões,
- os sub-blocos de primitivas de desenhos preenchidos, e
- os sub-blocos auxiliares como o banco de registradores e de controle.

Esses sub-blocos são mostrados na Figura 4.1 com linhas tracejadas, linhas pontilhadas e linhas contínuas, respectivamente.

Os blocos de desenho das primitivas de duas dimensões são compostos por desenhos de linhas, círculos, retângulos e triângulos. Os blocos de desenhos preenchidos são formados por preenchimento de círculos, retângulos e triângulos. Os blocos auxiliares são a FIFO de instruções, conjunto de registradores de uso geral e de propósito especiais, blocos de mapeamento de coordenadas para posições de memória do *frame buffer* e o bloco de controle. Esses blocos são explicados com mais detalhes nas seções seguintes. Neste contexto de Computação Gráfica alguns conceitos e definições são importantes, por isso faz-se necessário uma breve introdução os mesmos.

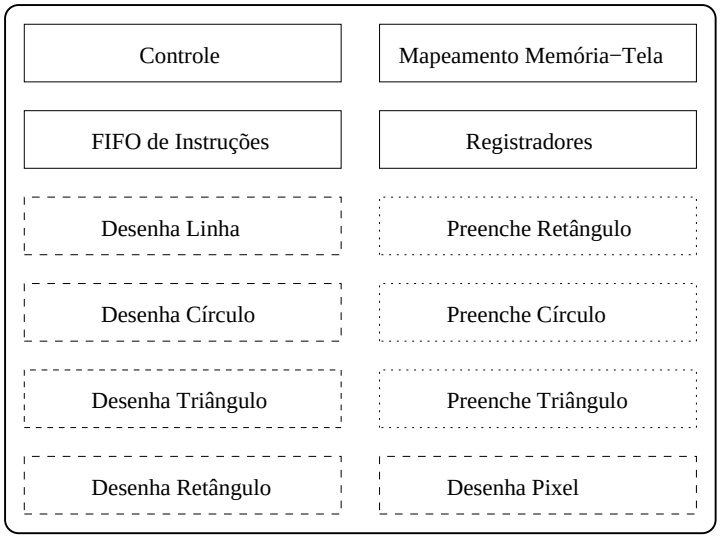

Figura 4.1: Visão em sub-blocos do co-processador implementado.

4.3 Introdução sobre Rasteirização

O termo rasteirização usado em computação gráfica surgiu devido aos tipos de hardware utilizados no início da década de 70, principalmente o *raster display*. Este teve

suas origens em um dispositivo de saída que o antecedeu chamado *vector display* e a arquitetura que o utilizava será explicada com intuito de contextualizar o co-processador desenvolvido no vocabulário da computação gráfica.

A arquitetura originada no meio da década de 60 e usada até o meio da década de 80 era conhecida por *vector system architecture* e deu origem ao modernos *raster displays* que hoje utilizamos. Essa arquitetura era composta por um processador, uma memória de tela e o tubo de raios catódicos. A memória armazenava uma lista de comandos que, quando executados, formavam os desenhos na tela. Estes comandos armazenados na memória eram lidos pelo processador que os convertia em sinais analógicos. Estes sinais eram usados para fazer deflexões no tubo produzindo as linhas, pontos e figuras desejadas. Para que o desenho permanecesse na tela, essa memória era lida em uma freqüência de aproximadamente 30 Hz.

Para substituir essa arquitetura, no inicio da década de 70, uma nova arquitetura baseada na tecnologia da televisão começou a ser utilizada e difundiu-se rapidamente porque não era tão cara quanto a anterior. Essa tecnologia chamada de *raster scan displays* foi a que mais contribuiu para fomentar o desenvolvimento na área, que em seguida, tornou-se conhecida como computação gráfica [1]. O sistema de varredura, baseado no mesmo mecanismo utilizado nas televisões, percorria a tela uma linha por vez da parte superior a inferior acendendo ou apagando um feixe de elétrons criando um padrão de pontos iluminados. O padrão a ser desenhado era lido da memória de tela que armazenava diretamente o formato do desenho. Assim, os desenhos de linhas, caracteres e áreas preenchidas eram armazenados na memória que antes continha operações de desenho destas mesmas figuras geométricas.

O mapeamento entre a memória e a tela pode ser feito de diferentes formas. Este varia de acordo com a quantidade de memória disponível, a definição da tela, o número de cores desejadas, a capacidade de varredura do tubo de raios catódicos, dentre outros. Para uma visualização em preto e branco, cada bit da memória de tela representa cada pixel da tela fazendo um mapeamento direto entre ambos. Os dois possíveis estados do bit indicam se o feixe deve estar aceso ou apagado naquela posição. Desta forma, pode-se representar um sistema de computação gráfica completo.

Fazendo um paralelo entre os termos utilizados no hardware de sistemas de computação gráfica e os termos correspondentes aos usados no presente trabalho em um sistema simulado, temos que:

- o *frame buffer* é uma área de memória de tela instanciada pelo co-processador;
- o co-processador efetua os desenhos das primitivas no *frame buffer* e esta memória é compartilhada com a *GVT* através da interface PLI (seção 3.5.3);
- a tela é criada pelo programa *GVT*; e

- a varredura e coerência entre o *frame buffer* e a tela é feita através de uma *thread*.

Contextualizados os termos do hardware de computação gráfica, pode-se focar no co-processador implementado que será detalhado nas próximas seções.

4.4 Primitivas de Desenho em Duas Dimensões

Uma imagem pode possuir diversos objetos dos mais simples aos mais complexos. No primeiro conjunto, estão as linhas, pontos, dentre outros, enquanto no segundo conjunto estão paisagens, pessoas, carros e etc. Acredita-se encontrar mais freqüentemente o segundo conjunto, porém essas estruturas complexas são formadas pelas primeiras. Ou seja, formas simples com linhas retas, pontos e cores podem representar qualquer imagem por mais complexa que esta seja. Desta forma, estes desenhos mais simples são utilizados como fundações para os mais complexos e por isso, são chamados de primitivas de desenhos em duas dimensões ou primitivas de saída.

As primitivas de desenhos em duas dimensões são as linhas, círculos, arcos, elipses, curvas cônicas e o preenchimento de figuras. Para se implementar um co-processador de computação gráfica, foi necessário implementar um sub-conjunto dessas primitivas. Em cada uma das sub-seções seguintes há uma explicação em nível algorítmico das funções de desenho (especificação executável) e das decisões e pressuposições necessárias para implementá-las em linguagem de descrição de hardware. Algumas sub-seções mereceram uma maior atenção já que são blocos chaves da implementação, como o desenho de linhas, ou porque a implementação feita difere-se da sugerida na literatura como o caso do preenchimento de triângulos.

Em relação a linguagem de descrição utilizada, optou-se por utilizar Verilog [14] por ser uma linguagem compacta e pela facilidade em se encontrar simuladores gratuitos para a mesma. No entanto, a arquitetura de verificação proposta neste trabalho poderia ter como base qualquer linguagem de descrição de hardware desde que um simulador apropriado da linguagem escolhida fosse utilizado.

4.4.1 Desenho de Linha por Meio de Algoritmo de Midpoint

Em um plano cartesiano, a equação de uma reta é:

$$y = m \cdot x + b \tag{4.1}$$

e m dado por

$$m = \frac{y2 - y1}{x2 - x1} = \frac{\Delta y}{\Delta x} \tag{4.2}$$

onde x e y são as ordenadas e abscissas, respectivamente; $x1$, $x2$, $y1$ e $y2$ ordenadas e abscissas de coordenadas (x_1, y_1) e (x_2, y_2) quaisquer; m é a inclinação da reta e b é o

ponto onde a mesma intercepta o eixo das ordenadas como mostra a Figura 4.2

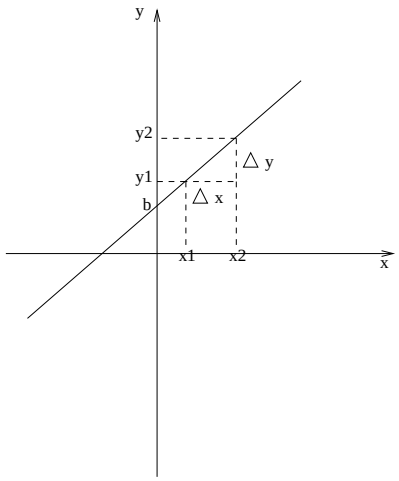

Figura 4.2: Definição de reta no plano cartesiano.

O termo linha, nesse contexto, dever ser entendido como um segmento de reta entre duas coordenadas bem definidas. Assim, a princípio, um sub-bloco para desenho de linhas seria apenas a implementação direta da Equação 4.1 em alguma linguagem de descrição de hardware.

No entanto, o algoritmo mais preciso e eficiente para desenho de linhas não necessita de multiplicação e é conhecido como MLA (do Inglês, *Midpoint Line Algorithm*). Este algoritmo usa apenas cálculos de incrementos inteiros para o cálculo das ordenadas e abscissas. Um outro algoritmo, também popular, mas menos eficiente que o MLA é o algoritmo DDA (do Inglês, *Digital Differential Analyzer*).

O algoritmo de DDA é um método mais eficiente do que utilizar diretamente a Equação 4.1 porque elimina a operação de multiplicação. Essa operação pode ser suprimida porque o DDA utiliza características de rasteirização, ou seja, incrementa de forma apropriada os valores das ordenas e abscissas, e a cada passo, indica a direção da linha desejada. Porém, o sucessivo acúmulo de erros de arredondamento existente nesse algoritmo, pode tornar o desenho da linha um pouco diferente da linha desejada em grandes caminhos. Além disso, o algoritmo de DDA necessita de operações de arredondamento e operações em ponto flutuante que consomem muito tempo e recurso lógicos.

O algoritmo MLA é um aprimoramento dos diferentes algoritmos propostos desde a publicação do algoritmo clássico proposto por Bresenham [54]. A principal alteração no algoritmo original foi publicada por Pitteway [55] e adaptada por Van Aken et al. [56] e fornece uma maior generalização da proposição inicial possibilitando, inclusive, a geração de figuras geométricas cônicas sem a operação de multiplicação.

O algoritmo MLA é mostrado na Listagem 4.1 e foi obtido de Foley et al. [57].

Listagem 4.1: Algoritmo MLA

```
1  procedure MidpointLine (x0, y0, x1, y1, value : integer);
2  var
3    dx, dy, incrE, incrNE, d, x, y : integer;
4  begin
5    dx := x1 - x0;
6    dy := y1 -y0;
7    d := 2 * dy - dx;
8    incrE := 2 * dy;
9    incrNE := 2 * (dy - dx);
10   x := x0;
11   y := y0;
12   WritePixel(x, y, value);
13   while x < x1 do
14     begin
15       if d <= 0 then
16         begin
17           d := d + incrE;
18           x := x + 1;
19         end
20       else
21         begin
22           d := d + incrNE;
23           x := x + 1;
24           y := y + 1;
25         end
26       WritePixel(x, y, value);
27     end
28 end;
```

De forma sucinta, o algoritmo mostrado na Listagem 4.1, executa os seguintes passos. Primeiramente, calcula o tamanho dos incrementos *delta* dados pelas variáveis incrE e incrNE. Em seguida, determina através da variável d para qual eixo os incrementos devem ser unitários e para qual eixo os incrementos *delta* devem ser utilizados. A partir daí, o laço de incrementos é executado do ponto inicial ao ponto final da reta, respeitando a condição definida pela variável d[1].

A linha desenhada pelo algoritmo é mostrada na Figura 4.3. A primeira linha (reta crescente) é das coordenadas $(6, 11)$ a $(16, 15)$ e a segunda linha (reta decrescente) é dos pontos $(7, 19)$ a $(11, 16)$.

A partir de uma especificação executável, similar a descrita na Listagem 4.1, um sub-bloco para o desenho de linhas foi em implementado em Verilog HDL. Resumidamente, este bloco é formado por uma máquina de estados com oito estados e utilizou 298 elementos lógicas na FPGA especificada no capítulo 5. É importante ressaltar que a função WritePixel(x, y, value);, em Verilog, corresponde a um sub-bloco para o mapea-

[1]Para uma visão aprofundada, consulte Foley et al. [57].

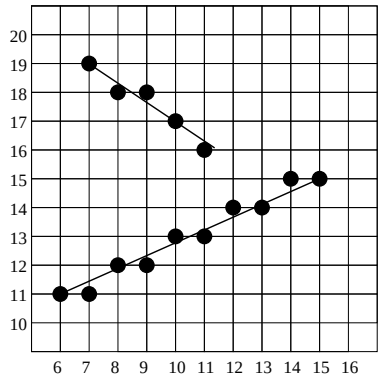

Figura 4.3: Desenho de duas linhas usando o MLA.

mento de coordenadas do mundo em endereços lineares da memória da tela e este será descrito na seção 4.6.2.

É interessante notar que contrariamente o que foi dito, o algoritmo possui três multiplicações. Mas basta observar que estas são todas multiplicadas por dois que na implementação em Verilog HDL foram substituídas por deslocamentos aritméticos. Estas foram deixadas na descrição do algoritmo apenas com o intuito de facilitar o entendimento do mesmo.

4.4.2 Desenho de Círculo por Meio de Algoritmo de Midpoint

Os círculos também são importantes primitivas de desenho em duas dimensões. A equação de um círculo centrado na origem é dada por:

$$x^2 + y^2 = R^2 \tag{4.3}$$

onde x e y são as abscissas e ordenadas do plano cartesiano e R é o raio. A implementação de um sub-bloco para desenho de círculo poderia aplicar diretamente a Equação 4.3, mas o algoritmo mais eficiente para o desenho desta primitiva é o Algoritmo de Midpoint para Círculo.

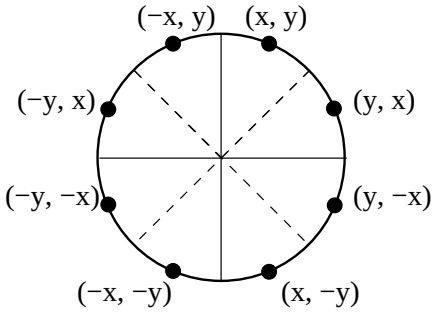

Figura 4.4: Pontos de simetria para desenho do círculo

Este algoritmo foi proposto por Bresenham [58] e usa o mesmo raciocínio do método

incremental desenvolvido para linhas. A primeira simplificação em relação ao uso direto da Equação 4.3 está no fato de usar simetria para calcular alguns pontos. O algoritmo efetua os cálculos para apenas 1/8 do círculo e os outros pontos são obtidos por simples troca de sinais e troca entre as variáveis x e y como mostra a Figura 4.4.

O algoritmo calcula os pontos através de um incremento e verifica qual destes está mais próximo do desenho do círculo. Esse desenho é dado por $F(x, y) = x^2 + y^2 - R^2$, se o resultado é $F(x, y) = 0$, então o ponto está no círculo, se $F(x, y) > 0$ então o ponto está fora do círculo e o ponto está dentro do círculo, se $F(x, y) < 0$. É mostrado por Foley et al. [57] que dado dois pixels p_0 e p_1 para se decidir qual está mais próximo do círculo, o seguinte método é usado. Se o ponto médio entre p_0 e p_1 está fora do círculo, o primeiro pixel está mais perto círculo e será preenchido. Se o ponto médio está dentro do círculo, o segundo está mais perto e será preenchido.

Listagem 4.2: Algoritmo de Midpoint para o Desenho de Círculo

```
1  procedure MidpointCircle (radius, value : integer);
2  var
3    x, y, d : integer;
4  begin
5    x := 0;
6    y := radius;
7    d := 1 - radius;
8    WritePixel(x, y, value);
9
10   while y > x do
11     begin
12       if d < 0 then
13         begin
14           d := d + 2 * x + 3;
15           x := x + 1;
16         end
17       else
18         begin
19           d := d + 2 * (x - y) + 5;
20           x := x + 1;
21           y := y - 1;
22         end
23       WritePixel(x, y, value);
24     end
25 end;
```

O algoritmo mostrado na Listagem 4.2 evidencia o raciocínio do parágrafo anterior. A variável d é chamada de variável de decisão e é responsável por definir se o ponto médio entre os pixels a serem escolhido está dentro ou fora do círculo. Como explicado anteriormente, se o $d < 0$, então o pixel mais próximo é o p_0 representado

pelo **if** d < 0 **then** ... **end**. Caso $d >= 0$, o pixel mais próximo é o p_1 e o **else** ... **end** é executado.

A partir da especificação executável do algoritmo mostrado na Listagem 4.2, uma máquina de estados com oito estados foi implementada em Verilog para representar o sub-bloco de desenho de círculo e exigiu 426 elementos lógicos da FPGA escolhida.

4.4.3 Desenhos de Triângulo e Retângulo

As primitivas de desenho de triângulo e retângulo também foram implementadas com o intuito de prover uma maior flexibilidade nos desenhos. A primitiva de triângulo exige três instruções do co-processador, pois é necessário fornecer os três vértices dessa figura geométrica que são armazenados nos registradores de propósito geral. Já para o desenho de retângulo, apenas duas instruções são necessárias.

Quando as instruções são buscadas na FIFO e decodificadas, o sub-bloco de controle inicia os sub-blocos que executarão aquela instrução. Os sub-blocos de desenho de retângulo e triângulo estão ligados diretamente ao bloco de desenho de linha. Dessa forma, o bloco de triângulo busca os três pontos nos registradores, os fornece dois a dois para o sub-bloco de desenho de linha e esse último se encarrega de desenhar a linha. A mesma seqüência ocorre com o desenho do retângulo, exceto que este necessita apenas de dois pontos: o ponto superior esquerdo e o inferior direito.

A implementação do desenho de triângulo utilizou 77 elementos lógicos e a implementação do retângulo utilizou 76 elementos lógicos.

4.5 Primitivas de Preenchimento em Duas Dimensões

Em pacotes de computação gráfica em software, a forma mais comum de se preencher primitivas é através de algoritmos de preenchimento de polígonos. Existem duas formas básicas de se preenchê-los: a primeira é através do desenho de linhas de preenchimento horizontais ou verticais limitadas pela borda da figura; e a segunda é fornecer um ponto interior ao desenho e preencher, por chamadas recursivas, pontos vizinhos até encontrar uma condição específica de borda. O algoritmo mais conhecido que se baseia na primeira forma de preenchimento é denominado de *Scan-line Fill* e os principais algoritmos para preenchimento baseados na segunda forma são o *Boundary Fill* e o *Flood Fill*. Por serem mais simples, dois últimos são mais populares.

Nos dois último algoritmos, a partir de um ponto inicial e interior ao desenho, os pontos adjacentes a este são percorridos. Em seguida, um ponto já preenchido é escolhido para ser o novo ponto inicial e assim sucessivamente. A principal diferença entre ambos está na condição de parada que, no primeiro, é a borda da figura e, no segundo,

é a existência de algum pixel com a cor antiga, ou seja, antes do preenchimento[2].

Estes algoritmos possuem implementações simples em linguagens de programação estruturadas, mas a implementação dos mesmos em linguagens de descrição de hardware é complexa. A primeira forma é simples porque existe a possibilidade de chamadas de procedimentos recursivos, o que não é diretamente implementado em hardware. Por isso, optou-se por uma abordagem diferente em relação as primitivas de preenchimento para a implementação em hardware. O foco foi direcionado principalmente para algoritmos mais relacionados com o *Scan-line Fill*.

Os algoritmos *Boundary Fill* e *Flood Fill* possibilitariam uma maior flexibilidade, pois podem preencher qualquer polígono. Mas escolhendo-se o *Scan-line Fill*, optou-se por focar em um maior desempenho e menor consumo de área. Como este último não é tão geral quanto os primeiros, determinou-se um subconjunto de algoritmos de preenchimentos a serem implementados. Este subconjunto é formado por algoritmos de preenchimento de retângulo, de triângulo e de círculo.

4.5.1 Preenchimento de Retângulo

O preenchimento da primitiva de desenho de retângulo é bastante simples e intuitiva como mostra a Listagem 4.3. Assume-se que x0 e y0 representam o canto superior esquerdo do retângulo e x1 e y1 o canto inferior direito do mesmo. Para o algoritmo abaixo assume-se que o eixo de coordenadas está centrado no canto superior esquerdo da tela.

Listagem 4.3: Algoritmo de Preenchimento de Retângulo

```
1  procedure rectangleFilling (x0, y0, x1, y1, value : integer);
2  var
3    x0, y0, x1, y1, value : integer;
4  begin
5    x := x0;
6    y := y0;
7
8    while y < y1 do
9      begin
10       while x < x1 do
11         begin
12           WritePixel(x, y, value);
13           x := x + 1;
14         end
15       x := x0;
16       y := y + 1;
17     end
18 end;
```

[2]Para uma visão mais detalhada, consulte Foley et al. [57].

O algoritmo apenas preenche com linhas de uma borda em uma extremidade a outra. É interessante notar que na implementação em Verilog, o anel mais interno do algoritmo representado pelo **while** x < x1 **do begin** ... **end** não existe. Isso porque este trecho foi substituído pela instância do sub-bloco de desenho de linha, que, para este caso em particular, faz o mesmo trabalho. Esta implementação utilizou 260 células lógicas.

4.5.2 Preenchimento de Triângulo

O algoritmo para preenchimento de triângulo utilizado foi proposto por Abbas et al. [59] e consiste em uma implementação específica para este objetivo, ou seja, este não faz o preenchimento de polígonos como o *Boundary Fill* e o *Flood Fill*.

A partir das três coordenadas oferecidas para o desenho do triângulo (x_1, y_1), (x_2, y_2) e (x_3, y_3) dois blocos do algoritmo de desenho de linha são iniciados com os seguintes parâmetros linha de (x_1, y_1) a (x_3, y_3) para o primeiro e linha de (x_2, y_2) a (x_3, y_3) para o segundo. A medida que o pontos de cada uma destas linhas é calculado, uma linha é traçada entre eles como mostra a Figura 4.5. O preenchimento termina quando as duas linhas terminam a sua execução.

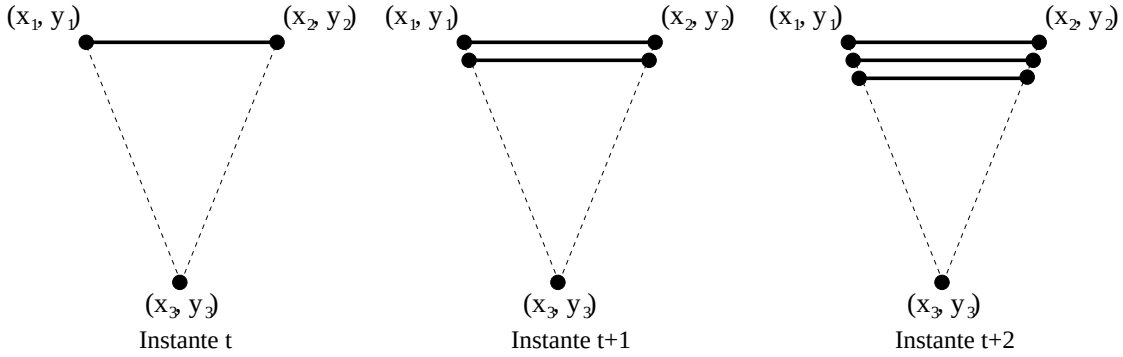

Figura 4.5: Instantes no preenchimento do triângulo.

Na Figura 4.5, as linhas tracejadas foram desenhadas apenas para facilitar a visualização do triângulo a ser preenchido. As linha sólidas representam o preenchimento feito entre duas coordenadas distintas obtidas pelos algoritmos de desenhos de linhas.

A implementação do algoritmo de preenchimento de triângulo é mostrado na Listagem 4.4

Listagem 4.4: Algoritmo de preenchimento de triângulo

```
1  procedure triangleFilling (x1, y1, x2, y2, x3, x3, value : integer);
2  var
3    xl, yl, x2, y2, x3, y3, value          : integer;
4    xLeft, yLeft, xRight, yRight, yFill    : integer;
5  begin
6    xLeft  = x1;
```

```
7       yLeft  = y1;
8       xRight = x2;
9       yRight = y2;
10      yFill  = y1;
11
12      do
13        begin
14          xLeft  = DrawLine2(xLeft, yLeft, x3, y3, value);
15          xRight = DrawLine2(xRight, yRight, x3, y3, value);
16
17          DrawLine(xLeft, yLeft, xRight, yRight, value);
18          yFill = yFill - 1;
19        end
20      while ( yFill <> y3 )
21    end;
```

Este algoritmo atende ao propósito de preenchimento de triângulos, mas alguns pontos devem ser ressaltados. Assume-se que a função Drawline2(...), ao ser chamada não desenha uma linha, mas apenas retorna o valor da abscissa para que a linha de preenchimento seja desenhada. Além disso, a condição de teste do laço só necessita considerar a variável yFill que é apenas decrementada para que as linhas horizontais de preenchimento sejam desenhadas como mostra a Figura 4.5.

A princípio, o algoritmo mostrado na Listagem 4.4 só serviria para casos bem específicos como o do triângulo isósceles e com uma base na horizontal como mostrado na Figura 4.5. Casos mais gerais como o da Figura 4.6 não seriam tratados.

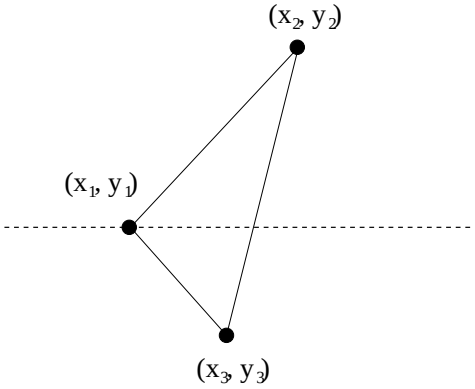

Figura 4.6: Triângulo qualquer dividido em partes.

Este representa o exemplo de um triângulo qualquer, sem particularidades. Contudo, se o triângulo mostrado na Figura 4.6 for dividido em duas partes teremos dois triângulos (inferior e superior) com a sua base na horizontal como um triângulo particular da Figura 4.5. Esta divisão pode ser feita criando-se uma linha a partir da coordenada oposta ao maior lado do triângulo como mostrado. Dessa forma, a única diferença seria que os triângulos divididos não possuem lados iguais, portanto o algo-

ritmo proposto ainda não seria adequado.

Esta limitação pode ser facilmente tratada colocando um inter-travamento entre as chamadas das duas funções Drawline2(...) existentes no algoritmo. Isto é, as linhas só serão desenhadas se as ordenadas yLeft e yRight possuírem o mesmo valor. Caso contrário, uma destas funções não é executada. Em resumo, utilizando estas características de rasteirização, é apenas necessário manter o sincronismo das coordenadas das linhas a serem desenhadas como proposto por Abbas et al. [59].

A implementação deste algoritmo em Verilog utilizou duas instâncias do módulo de desenho de linhas e consumiu 752 elementos lógicos.

4.5.3 Preenchimento de Círculo

O algoritmo para preenchimento de círculo não apresenta nenhuma novidade em relação aos algoritmos MLA e de desenho de círculo. Isso porque este apresenta grandes semelhanças com o algoritmo de desenho de círculo descrito na seção 4.4.2. De acordo com o algoritmo de desenho de círculo, apenas 1/8 do círculo é calculado através do algoritmo incremental descrito na Listagem 4.2. Os pontos dos outros octantes são obtidos por simples troca de sinais e troca entre as variáveis x e y como mostra a Figura 4.4.

Baseando-se nestas informações, para se implementar o preenchimento de círculo, basta calcular um ponto pelo algoritmo para desenho de círculo, definir o ponto de simetria e traçar uma linha entre estes pontos como mostra a Figura 4.7.

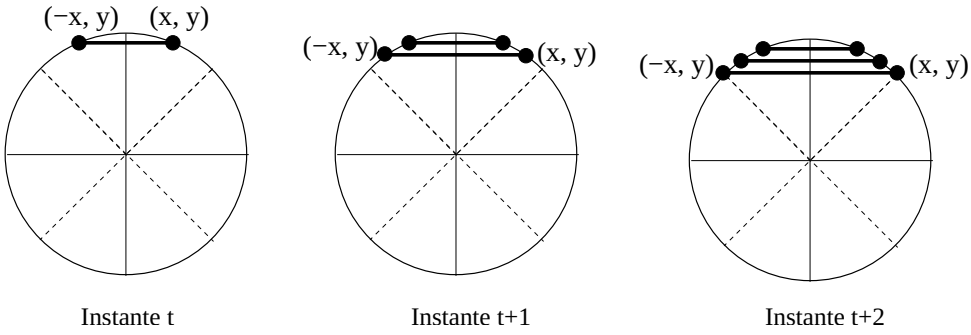

Figura 4.7: Instantes no preenchimento de círculo.

A Figura 4.7 mostra como o círculo é preenchido a medida em que o algoritmo é executado. A figura representa três instantes de tempo distintos e sucessivos. Em cada instante, a partir do algoritmo incremental de desenho de círculos a coordenada (x, y) é calculada e o ponto simétrico é definido como $(-x, y)$. Dados estes dois pontos, basta traçar uma linha que os interligue, preenchendo o interior do círculo. O algoritmo é detalhado na Listagem 4.5.

Listagem 4.5: Algoritmo de preenchimento de círculo

```
1  procedure MidpointCircleFilling (radius, value : integer);
2  var
3    x, y, d : integer;
4  begin
5    x := 0;
6    y := radius;
7    d := 1 - radius;
8    DrawLine(x, y, -x, y, value);
9
10   while y > x do
11     begin
12       if d < 0 then
13         begin
14           d := d + 2 * x + 3;
15           x := x + 1;
16         end
17       else
18         begin
19           d := d + 2 * (x - y) + 5;
20           x := x + 1;
21           y := y - 1;
22         end
23       DrawLine(x, y, -x, y, value);
24     end
25 end;
```

É importante observar que a única diferença entre o algoritmo de preenchimento de círculo e o de desenho de círculo é na utilização das variáveis x e y geradas. Enquanto, no primeiro, para cada uma destas variáveis desenha-se uma linha com a função DrawLine(x, y, −x, y, value), no segundo, apenas um pixel é desenhado com a função WritePixel(x, y, value).

A partir da Listagem 4.5, um bloco em Verilog foi implementado e este consumiu 449 elementos lógicos. Este bloco utilizou um máquina de estados com quatorze estados e possui conexão direta com o módulo de desenho de linhas o qual recebe os pontos das coordenas recém-calculadas.

4.6 Sub-blocos Auxiliares

Além dos sub-blocos de desenho de primitivas e preenchimento, outros sub-blocos também desempenham papel fundamental no co-processador e são descritos nas próximas subseções.

4.6.1 Sub-bloco da FIFO

O sub-bloco da FIFO contém apenas a fila de instruções a serem executadas no co-processador. As instruções podem ser inseridas pelo processador principal em uma ponta e retiradas pelo co-processador em outra. Apesar de não possuir nenhuma característica especial, essa tem extrema importância pois estabelece o único canal de comunicação entre o processador e o co-processador, já que não há compartilhamento de registradores ou comunicação através de nenhum protocolo. Este sub-bloco ocupou 245 elementos lógicos.

4.6.2 Sub-bloco de Mapeamento Tela-*Frame-Buffer*

O sub-bloco de mapeamento entre a tela e o *frame-buffer* executa uma função simples, mas muito importante. Todos os algoritmos mostrados nesse capítulo, efetuam os desenhos e escrevem na memória com a função WritePixel(x, y, value) ou através do DrawLine(x1, y1, x2, y2, value) que encapsula o primeiro.

Com o intuito de oferecer essa mesma abstração para os sub-blocos de desenho e preenchimento de primitivas em hardware, um bloco de mapeamento entre a tela e o *frame-buffer* foi implementado. Esse bloco recebe como variáveis de entrada as coordenadas e o valor a serem escritos (assim como na função); e fornece como variáveis de saída a posição de memória a ser escrita e o seu valor correspondente. A transformação é feita mapeando a coordenada (em duas dimensões) em endereço de memória (unidimensional) de acordo com o valor parametrizado do tamanho da tela. Este sub-bloco foi implementado em Verilog e utilizou apenas 20 elementos lógicos.

4.6.3 Sub-bloco de Controle

O sub-bloco de controle é responsável pela coordenação de todos os outros sub-blocos. Esse executa a busca de instruções na FIFO, as decodifica e dispara a execução nos sub-blocos. Assim que o sub-bloco responsável pela instrução termina a sua execução, este informa ao sub-bloco de controle que finaliza a instrução e busca uma nova instrução na FIFO.

4.7 Exemplos das Primitivas Gráficas Implementadas

A Figura 4.8 mostra exemplos de todas as primitivas implementadas na especificação executável e em Verilog. O desenho de linhas (seção 4.4.1) é utilizado para formar as bordas que estão coloridas de vermelho, verde, azul e branco. O desenho de círculo

(seção 4.4.2) e desenho de triângulo (seção 4.4.3) são executado três vezes como mostrado na figura. Além disso, dois retângulos são desenhos (seção 4.4.3) e posteriormente preenchidos (seção 4.5.1). Finalmente, três triângulos são preenchidos (seção 4.5.2) e três círculos são preenchidos (seção 4.5.3).

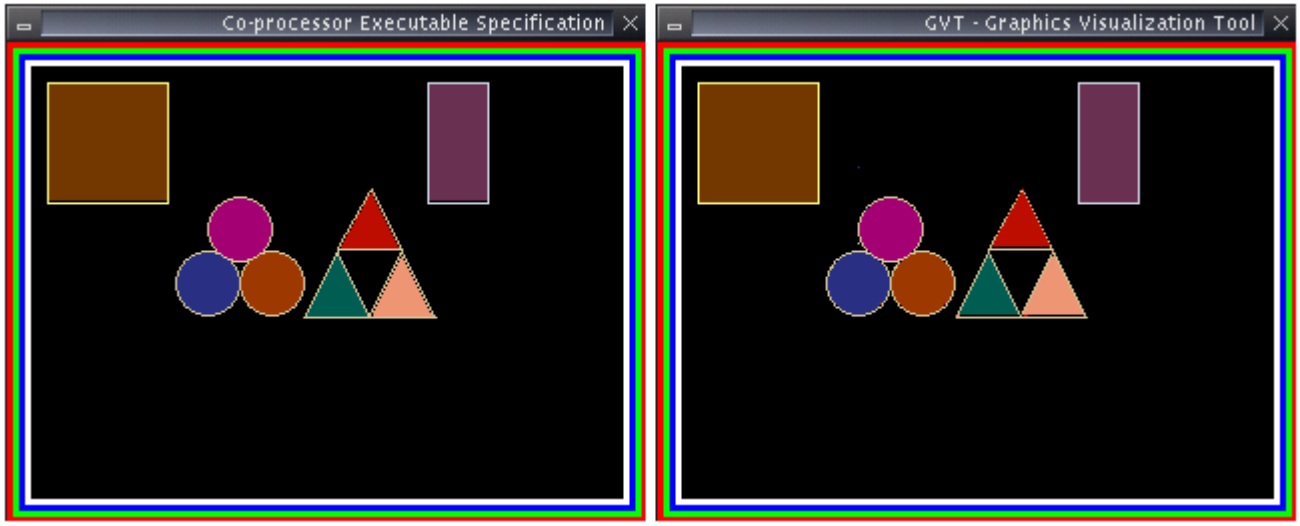

Figura 4.8: Exemplos das primitivas gráficas na especificação executável e em Verilog.

Capítulo 5

Resultados

Este capítulo tem por objetivo demonstrar e analisar os resultados obtidos com o uso da arquitetura de verificação proposta através de um estudo de caso.

5.1 Estudo de Caso

A princípio, a forma melhor de se avaliar a eficácia da arquitetura proposta seria compará-la diretamente com as outras metodologias detalhadas no capítulo 2. Para se fazer isto, seria necessário que equipes experientes em verificação formal, semi-formal e funcional verificassem um bloco de computação gráfica e uma outra equipe utilizasse a arquitetura proposta. Mesmo assim, isto ainda não garantiria que uma das metodologias é melhor para a verificação de blocos de computação gráfica. Talvez, este mecanismo apenas destacasse que uma equipe é mais eficiente que a outra, e não que uma metodologia é melhor.

Uma outra possibilidade para se demonstrar a eficiência da arquitetura proposta é através de um estudo de caso. Este artifício não tem o objetivo de mostrar que algum método é garantidamente melhor do que outro. No entanto, este pode destacar os principais benefícios e limitações de alguma técnica ou, para o presente trabalho, da arquitetura de verificação proposta.

É extremamente difícil encontrar blocos de computação gráfica em hardware disponíveis para verificação. Já que estes constituem parte da propriedade intelectual e, muitas vezes, da fonte de renda das empresas desenvolvedoras. Por isso, foi necessário desenvolver um bloco destes para que a arquitetura fosse validada.

Assim, a validação da arquitetura proposta será feita através de um estudo de caso que passa pelas seguintes etapas:

- inicia-se com o desenvolvimento de uma especificação executável para o bloco de computação gráfica em hardware;

- a implementação do mesmo em uma linguagem de descrição de hardware apropriada,
- a verificação deste bloco com a arquitetura proposta; e,
- finalmente, a prototipação.

O capítulo 4 fornece uma descrição detalhada dos dois primeiros estágios que são necessários para este estudo de caso. Já as seções seguintes detalham a utilização da arquitetura proposta mostrando os erros detectados e, em uma etapa posterior, os resultados da prototipação.

5.2 Resultados da Verificação Utilizando a Arquitetura Proposta

Esta seção tem por objetivo mostrar os principais erros encontrados por meio da arquitetura de verificação ressaltando os benefícios e limitações da mesma. Os erros foram divididos e analisados por subseções de acordo com a sua categoria.

5.2.1 Erros de Interfaces e Interação entre Sub-blocos

Erros nas interfaces entres os sub-blocos são muito recorrentes em projetos de circuitos integrados. Isto acontece porque, muitas vezes, estes são elaborados por equipes diferentes e que fizeram a sua própria interpretação sobre o que deveria ser implementado. Mesmo quando apenas uma equipe ou um desenvolvedor projeta dois sub-blocos que irão se interagir, estes erros podem aparecer. Pois, garantir que dois sub-blocos funcionam separadamente não significa que ambos funcionarão em conjunto.

Em um dos casos de testes propostos para verificar o bloco de computação gráfica desenvolvido, foi estimulada a interação entre os blocos de preenchimento de triângulo, desenho de triângulos e desenho de linhas. Neste caso de teste, um erro foi detectado na interação entre os dois primeiros sub-blocos.

O primeiro passo da arquitetura proposta neste trabalho é a verificação por sub-blocos. Por isso, cada um dos sub-blocos foi verificado separadamente em um nível mais baixo de abstração por meio de asserções e por um nível mais alto de abstração por meio da visualização das primitivas com a GVT. Todavia, nenhum erro foi encontrado, nem por violação de alguma asserção ou pelo desenho de alguma primitiva incorreta. Lembrando que o caso de teste foi dividido para que apenas um primitiva fosse verificada por vez.

O segundo passo consiste na verificação em nível de sistema com auxílio da ferramenta V^2T. Por meio da execução do caso de teste completo, foi possível perceber que

havia um erro entre a interação entre os sub-blocos de desenho de triângulo e os de preenchimento de triângulos. A Figura 5.1 mostra o erro na tela da ferramenta V^2T.

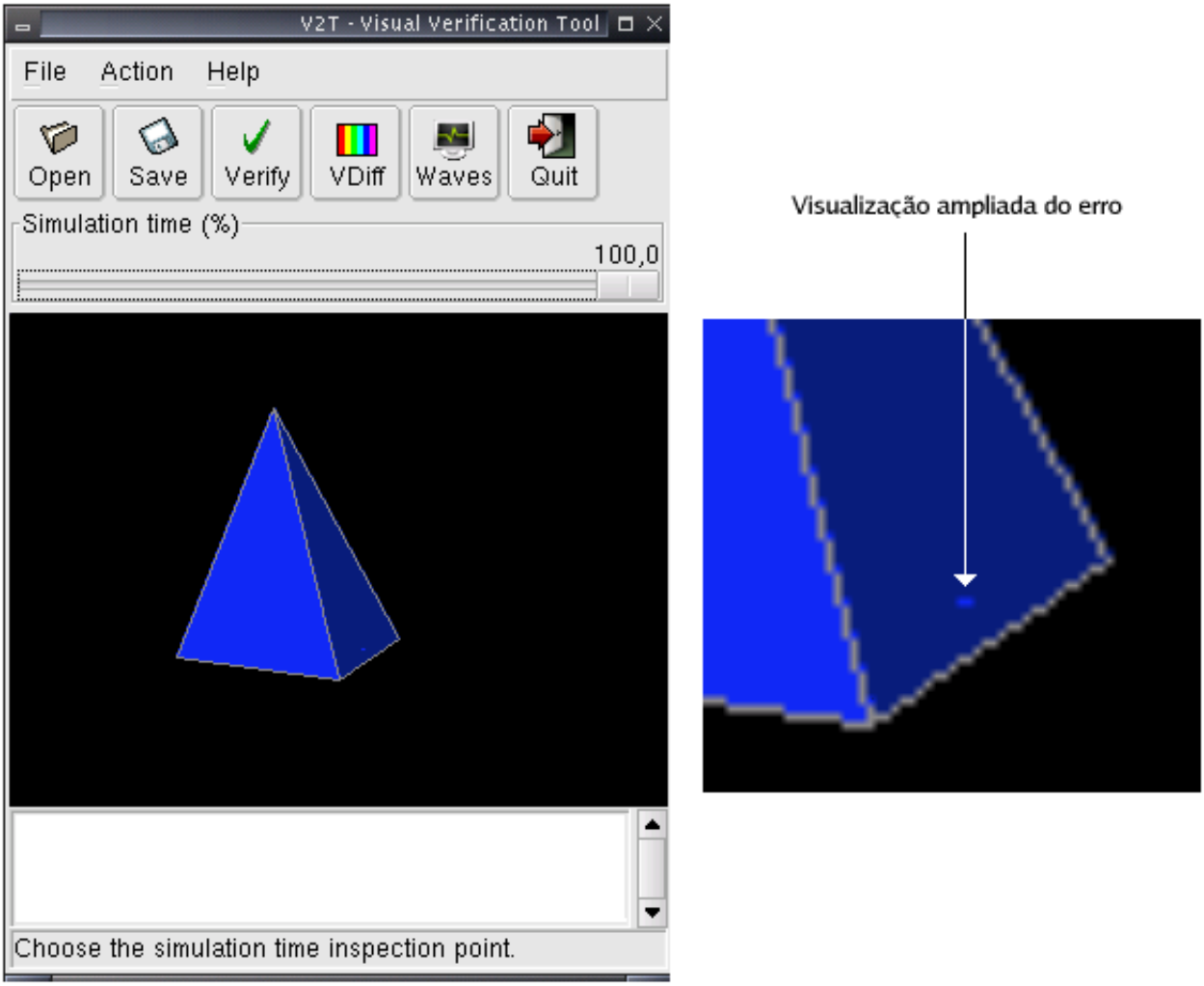

Figura 5.1: Visualização do erro do desenho de uma pirâmide através da ferramenta V^2T.

A simples visualização de uma primitiva com um erro não torna o processo de verificação mais eficiente. Por isso, a ferramenta V^2T possui um mecanismo de visualização em múltiplos níveis de abstração proposto por Andrade et al. [60]. Isto é, a ferramenta possui um visualizador das primitivas gráficas e um visualizador de formas de onda para os sinais em RTL. Além disso, ambos visualizadores estão sincronizados de forma que quando um ponto no tempo é selecionado em um deles o outro recompõe as informações do mesmo instante de tempo. A Figura 5.2 mostra como isto acontece e como o erro destas primitivas foi facilmente detectado.

Sabendo que alguns pixels haviam sido desenhados em pontos não apropriados como mostrado na Figura 5.1, foi necessário refazer o *trace* da simulação para se chegar ao ponto no tempo onde o erro ocorria. Utilizando a barra de rolagem de tempo na tela da ferramenta V^2T, foi possível detectar que os pixels errados eram desenhados em um

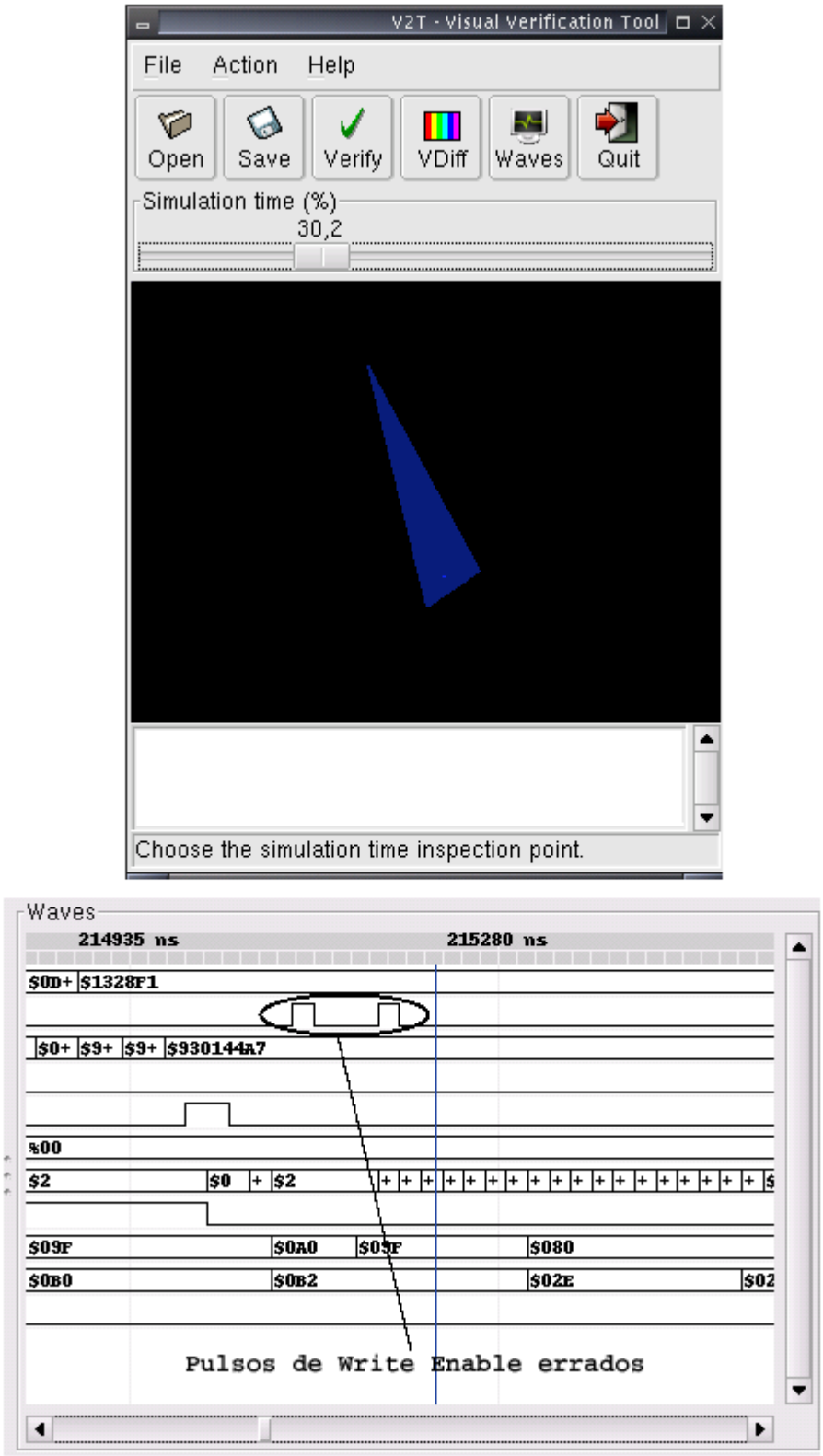

Figura 5.2: Ferramenta V^2T mostrando dados sincronizados de múltiplos níveis de abstração.

ponto no tempo que correspondia a $30,2\%$ do tempo total de simulação como mostra a Figura 5.2. Como o visualizador de formas de onda é sincronizado com o visualizador

de primitivas, o primeiro mostrou o instante de tempo correspondente a $30,2\%$ do tempo total que seria $215270ns$ no segundo e os sinais neste instante.

Destacando esta janela de tempo onde o erro ocorria, ficou fácil perceber que dois pulsos de *write enable* aconteciam na memória de tela devido a um erro de integração entre os sub-blocos de preenchimento e de desenho de triângulos como mostra a Figura 5.2. Este erro acontecia porque os sub-blocos não eram mantidos em estado de *reset* quando as instruções eram decodificadas. Desta forma, utilizando uma das funcionalidades da ferramenta V^2T, que é parte integrante da arquitetura de verificação proposta, o erro foi facilmente detectado e corrigido.

5.2.2 Erros Capturados por Meio de Asserções

Um dos erros principais encontrados através da verificação automática em um nível mais baixo de abstração aconteceu nas instruções que exigiam que desenhos de primitivas fossem feitos fora das coordenadas definidas para a tela. Os casos de testes que conseguiram detectar este erro foram definidos na arquitetura na seção 3.6.2 correspondentes a geração de estímulos para detectar *corner-cases*.

O erro foi detectado no primeiro passo de verificação da arquitetura proposta que é a verificação por sub-blocos. Através de propriedades da especificação, asserções foram instanciadas e para um caso de teste relativo ao desenho de triângulos, uma das asserções foi violada. Esta asserção foi definida no módulo de controle (ver seção 4.6.3).

Este módulo decodifica as instruções e a asserção verifica se as coordenadas codificadas na instrução estão dentro da faixa delimitada de acordo com as dimensões da tela. A Listagem 5.1 mostra as asserções definidas da linha 2 a linha 18 que garantem que nenhuma primitiva com pixel fora da tela será desenhada.

Listagem 5.1: Asserções instanciadas dentro do módulo de controle

```
1  ...
2  assert_always #(0, 0, "Ponto x de R1 fora da faixa")
3   ponto_x_r1(CLK, ~RESET, (r1[19:10] >= 0 ) && (r1[19:10] <= `WIDTH));
4
5  assert_always #(0, 0, "Ponto x de R2 fora da faixa")
6   ponto_x_r2(CLK, ~RESET, (r2[19:10] >= 0 ) && (r2[19:10] <= `WIDTH));
7
8  assert_always #(0, 0, "Ponto x de R3 fora da faixa")
9   ponto_x_r3(CLK, ~RESET, (r3[19:10] >= 0 ) && (r3[19:10] <= `WIDTH));
10
11 assert_always #(0, 0, "Ponto y de R1 fora da faixa")
12  ponto_y_r1(CLK, ~RESET, (r1[9:0] >= 0 ) && (r1[9:0] <= `HEIGHT));
13
14 assert_always #(0, 0, "Ponto y de R2 fora da faixa")
15  ponto_y_r2(CLK, ~RESET, (r2[9:0] >= 0 ) && (r2[9:0] <= `HEIGHT));
16
```

```
17   assert_always #(0, 0, "Ponto y de R3 fora da faixa")
18    ponto_y_r3(CLK, ~RESET, (r3[9:0] >= 0 ) && (r3[9:0] <= `HEIGHT));
19
20   ...
21   ST1:
22   begin
23     case (IR[`OPCODE])
24           `DLP1, `DLP2,  `SC,   `SP,  `DCC,  `DCR,
25           `DRP1, `DRP2, DTP1, `DTP2, `DTP3, `FRP1,
26           `FRP2,  `FCC, `FCR, `FTP1, `FTP2, `FTP3:
27            begin
28              case (IR[`REG_ADDR])
29                    `REG_0: r0 <= IR[`REG_DATA];
30                    `REG_1: r1 <= IR[`REG_DATA];
31                    `REG_2: r2 <= IR[`REG_DATA];
32                    `REG_3: r3 <= IR[`REG_DATA];
33               endcase
34            end
35      endcase
36
37   ...
```

A partir da violação de uma das asserções definidas, os dados gerados na simulação foram carregados no visualizador de formas de onda que é parte integrante da ferramenta V^2T para uma análise comparativa entre o erro em RTL e o erro em nível gráfico. A Figura 5.3 mostra como este erro se apresentaria no visualizador de forma de onda utilizado na ferramenta V^2T.

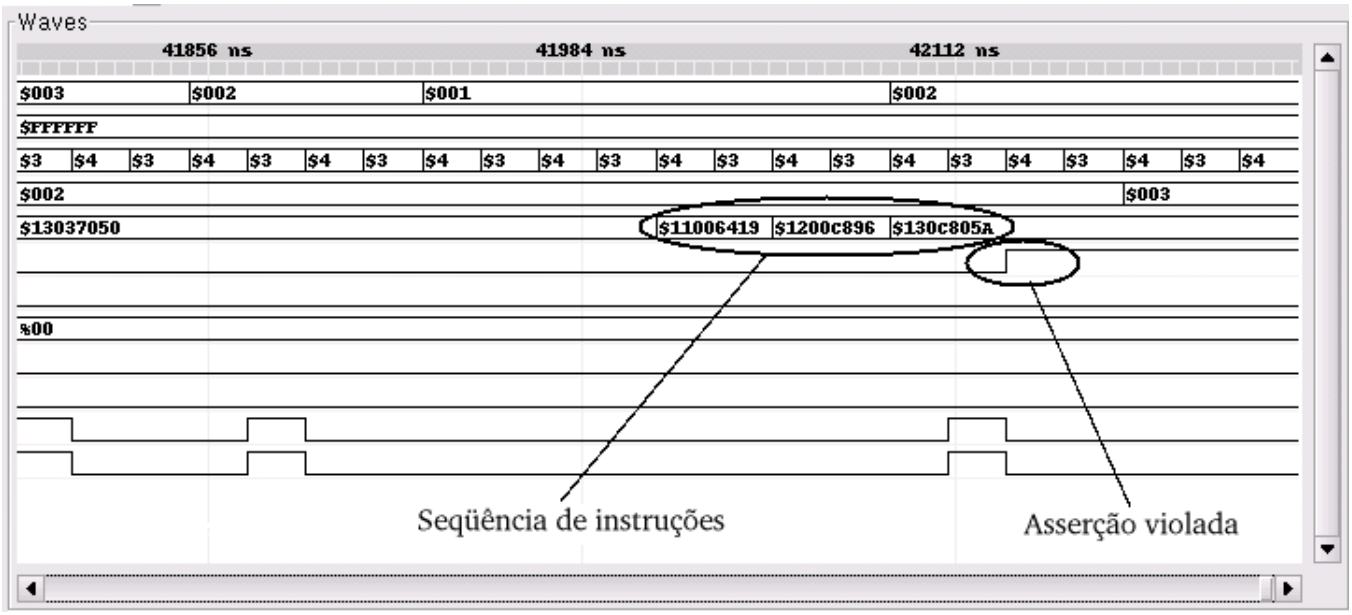

Figura 5.3: Asserção violada exibida no visualizador de forma de onda.

Observe que a asserção é violada no meio da decodificação da terceira instrução para

o desenho de triângulo que é dada pelo opcode 0x130C805A que corresponde a fixar o terceiro ponto da primitiva de desenho de triângulo no pixel $(800, 90)$. Ou seja, dado que os máximos valores para x e y para estes testes são de 320 e 240, respectivamente, o ponto x estaria fora da faixa. A Listagem 5.2 mostra a mensagem de erro indicada pela biblioteca OVL.

Listagem 5.2: Mensagem de asserção violada

```
1   ...
2  OVL_ERROR : ASSERT_ALWAYS : Ponto x de R3 fora da faixa :
3   : severity 0 : time 67730 : sim.GIB.CORE.control.ponto_x_r3.ovl_error
4   ...
```

Após a violação desta propriedade, o módulo que faz a interface com a memória foi alterado para evitar que erros como estes aconteçam. A solução encontrada a foi de inibir o desenho do ponto se o mesmo estiver fora das coordenadas especificadas sem interromper o fluxo de execução das instruções.

5.2.3 Erros Capturados por Meio de Verificação Gráfica

Os erros capturados por verificação gráfica podem ser divididos em duas categorias. A primeira delas corresponde aos erros de desenho de primitivas que não foram completadas. Isto é, quando o desenho da primitiva faltou algum pixel em alguma coordenada seja porque este foi desenhado em um ponto adjacente ou porque este não foi desenhado. A segunda categoria refere-se a erros que acontecem quando o desenho da primitiva possui todos os pixels que formam o seu desenho, mas além destes, outros que estão fora do lugares delimitados. As duas subseções a seguir exemplificam como cada um destes erros foi encontrado através da verificação automática em alto nível de abstração através da ferramenta V^2T.

Erro de Primitivas com Desenhos Incompletos

Este erro foi encontrado na verificação das instruções de desenho de círculos. A partir da especificação executável e da implementação detalhadas na seção 4.4.2, iniciou-se o processo de verificação proposto no presente trabalho

O primeiro passo da arquitetura de verificação proposta consiste na verificação por sub-blocos. Neste passo, o caso de teste deve ser dividido de forma a apenas possuir uma primitiva desenhada e verificada por vez. Neste caso de teste, tomou-se a liberdade de não se fragmentar o caso de teste já que apenas repetições da mesma primitiva são feitas. Assim, dentre os vários casos de testes gerados um deles, relativo ao desenho de círculos, apresentou um erro que dificilmente seria encontrado no primeiro passo da arquitetura de verificação ou por metodologias tradicionais.

A verificação por sub-blocos foi executada e o mecanismo de verificação automática no nível RTL (conjunto de asserções definidas a partir de propriedades da especificação) não acusou nenhum erro. Além disso, a comparação visual em nível mais alto de abstração fornecida pela *GVT* e os resultados obtidos pela especificação executável também não detectaram nenhum erro. A Figura 5.4 mostra os resultados obtidos comparando-se a especificação executável com os resultados obtidos na visualização da simulação em tempo de execução mostradas pela *GVT*.

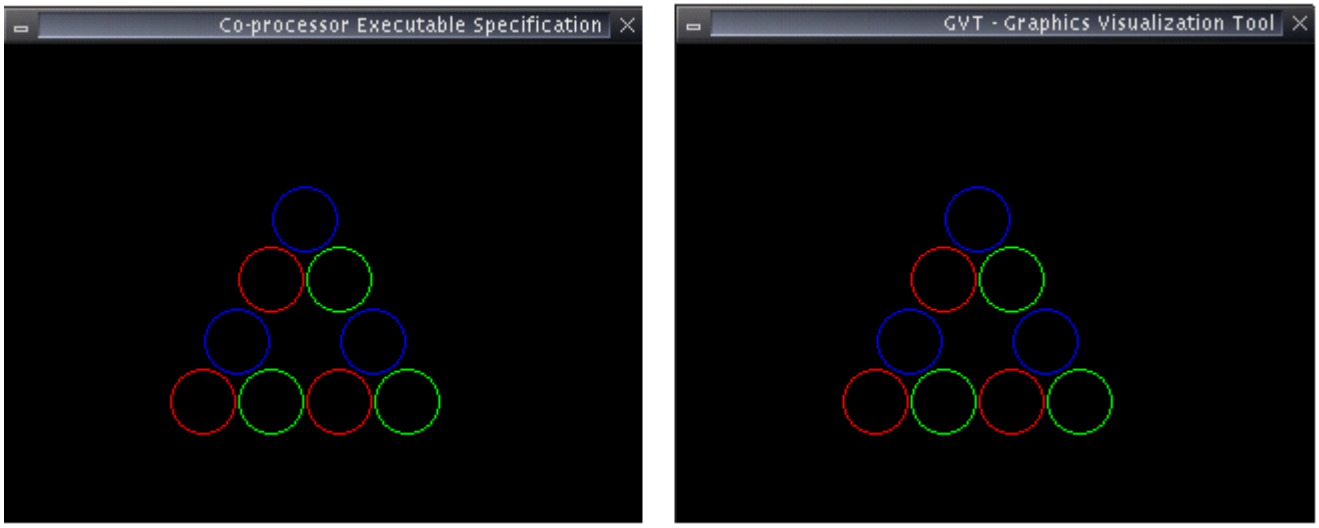

Figura 5.4: Especificação executável e visualização em tempo de simulação na verificação para verificação de desenho de círculos.

A partir daí, a análise em nível de sistema foi feita com a ferramenta V^2T e foi possível verificar que a instrução de desenho círculo não satisfazia a especificação executável. A Figura 5.5 mostra a ferramenta V^2T em dois momentos: o primeiro, quando apenas os dados da simulação foram carregados e o desenho apresenta-se como na Figura 5.4 e o segundo, quando a verificação automática e a diferença visual são executadas.

Apesar de nenhuma propriedade definidas pelas asserções ter sido violada. A verificação em um nível mais alto de abstração oferecida pela ferramenta V^2T mostrou que as instruções de desenho de círculo não foram executadas com sucesso. A Listagem 5.3 mostra a saída obtida com a verificação automática. Pode-se observar que as instruções de desenho de círculo mostradas nas linhas 5 e 6, 12 e 13, 19 e 20 estão incorretas segundo a ferramenta. Com auxílio da diferença visual, os erros naquelas instruções são facilmente observados. Na Figura 5.5 é possível ver que alguns pixels foram desenhados um pouco mais a esquerda ou a direita de onde os mesmos deveriam aparecer. Este tipo de erro é raramente observado por métodos tradicionais como a inspeção visual sugerida por Bergeron [2].

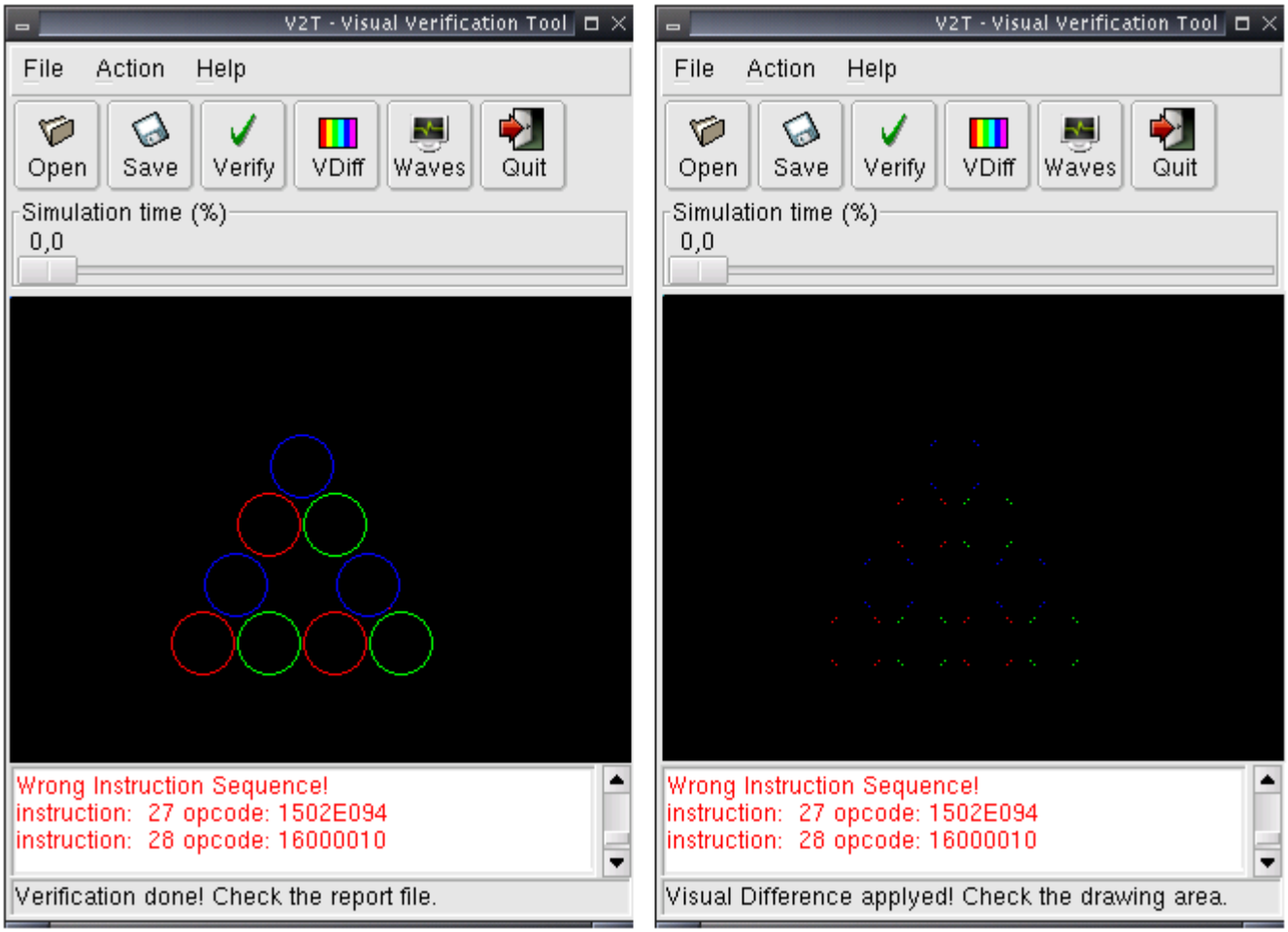

Figura 5.5: Ferramenta V^2T: antes e após a verificação automática e a diferença visual para a verificação de círculos.

Listagem 5.3: Resultado de Verificação Automática das Primitivas pela Ferramenta V^2T

```
1  Right Instruction Sequence
2  instruction: 0 opcode: 00ff0000
3
4  Wrong Instruction Sequence
5  instruction: 1 opcode: 15021475
6  instruction: 2 opcode: 16000010
7
8  Right Instruction Sequence
9  instruction: 3 opcode: 0000ff00
10
11 Wrong Instruction Sequence
12 instruction: 4 opcode: 15029c75
13 instruction: 5 opcode: 16000010
14
15 Right Instruction Sequence
16 instruction: 6 opcode: 000000ff
```

```
17
18 Wrong Instruction Sequence
19 instruction: 7 opcode: 15025857
20 instruction: 8 opcode: 16000010
21
22          ....
```

Após uma análise da implementação, foi observado que o sub-bloco de desenho de círculo em Verilog utilizava *regs* de 10 bits e a especificação executável utilizava *int* de 32 bits para as mesmas variáveis. Assim, nas operações efetuadas nos *regs* em Verilog, acontecia o truncamento deixando a implementação incorreta. Este problema foi resolvido adicionando-se algumas condições que verificam se havia estouro de capacidade nos *regs* de 10 bits.

Erro de Primitivas com Desenhos Completos

Este erro foi encontrado ao se verificar a instrução de preenchimento de retângulos. A arquitetura de verificação proposta neste trabalho foi utilizada e encontrou o erro de forma eficaz. Como primeiro passo, a verificação por sub-blocos foi executada não encontrando nenhuma inconformidade com a especificação. Isto é, a verificação feita pelas asserções em um nível mais baixo de abstração não indicou nenhum erro e muito menos a inspeção visual feita através da visualização das primitivas fornecidas pela especificação executável e pelo *GVT* como mostra a Figura 5.6. Ambos os resultados apresentam-se bastante similares em uma inspeção visual. É bom ressaltar que o caso de teste não foi divido em apenas uma primitiva pelos mesmos motivos apresentados na subseção anterior.

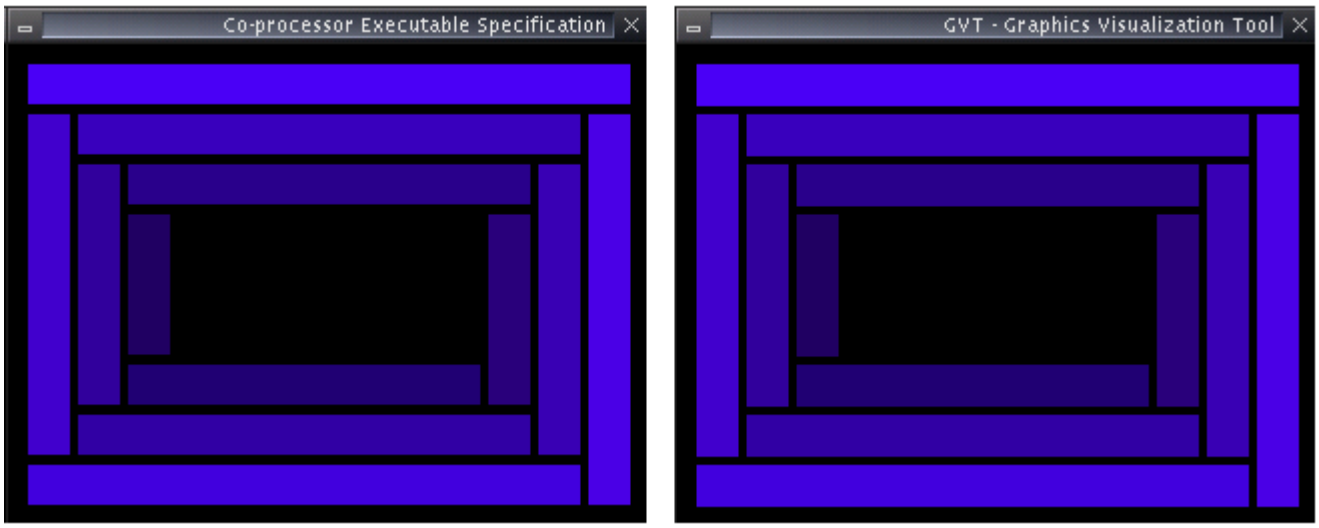

Figura 5.6: Especificação executável e visualização em tempo de simulação para preenchimento de retângulos.

Finalizada a etapa de verificação por sub-blocos, a verificação em nível de sistema que utiliza a ferramenta V^2T e os arquivos obtidos na simulação foi iniciada. A Figura 5.7 mostra a tela de saída da ferramenta V^2T antes e após a verificação e a diferença visual.

A ferramenta V^2T efetuou a verificação automática em um nível mais alto de abstração a partir das primitivas gráficas e foi possível detectar que algumas linhas a mais foram desenhas mostrando que a implementação não está de acordo com a especificação. A Listagem 5.4 mostra a saída obtida com a verificação automática das primitivas.

Listagem 5.4: Ferramenta V^2T: saída para a Verificação do Preenchimento de Retângulo

```
1  Right Instruction Sequence
2  instruction: 0 opcode: 004c00f2
3
4  Right Instruction Sequence
5  instruction: 1 opcode: 8d00280a
6  instruction: 2 opcode: 8e04d81e
7
8  Right Instruction Sequence
9  instruction: 3 opcode: 004800e5
10
11 Right Instruction Sequence
12 instruction: 4 opcode: 8d048823
13 instruction: 5 opcode: 8e04d8e6
14
15 Right Instruction Sequence
16 instruction: 6 opcode: 004400d8
17
18 Right Instruction Sequence
19 instruction: 7 opcode: 8d0474d2
20 instruction: 8 opcode: 8e0028e6
21
22         ....
```

É interessante notar que apesar de a verificação automática mostrar que todas as instruções foram executadas estão de acordo com a especificação, pode-se ver pela Figura 5.7 que há uma divergência entre ambas. Isto acontece porque o mecanismo de verificação automática verifica se o desenho determinado por uma instrução foi executado com sucesso. Neste caso, a ferramenta V^2T informa que as primitivas que deveriam ser desenhadas foram efetuadas com sucesso, mas o desenho como um todo está incorreto. Já que além da primitiva de desenho de retângulo, por exemplo, algumas linhas a mais foram desenhadas sem necessidade.

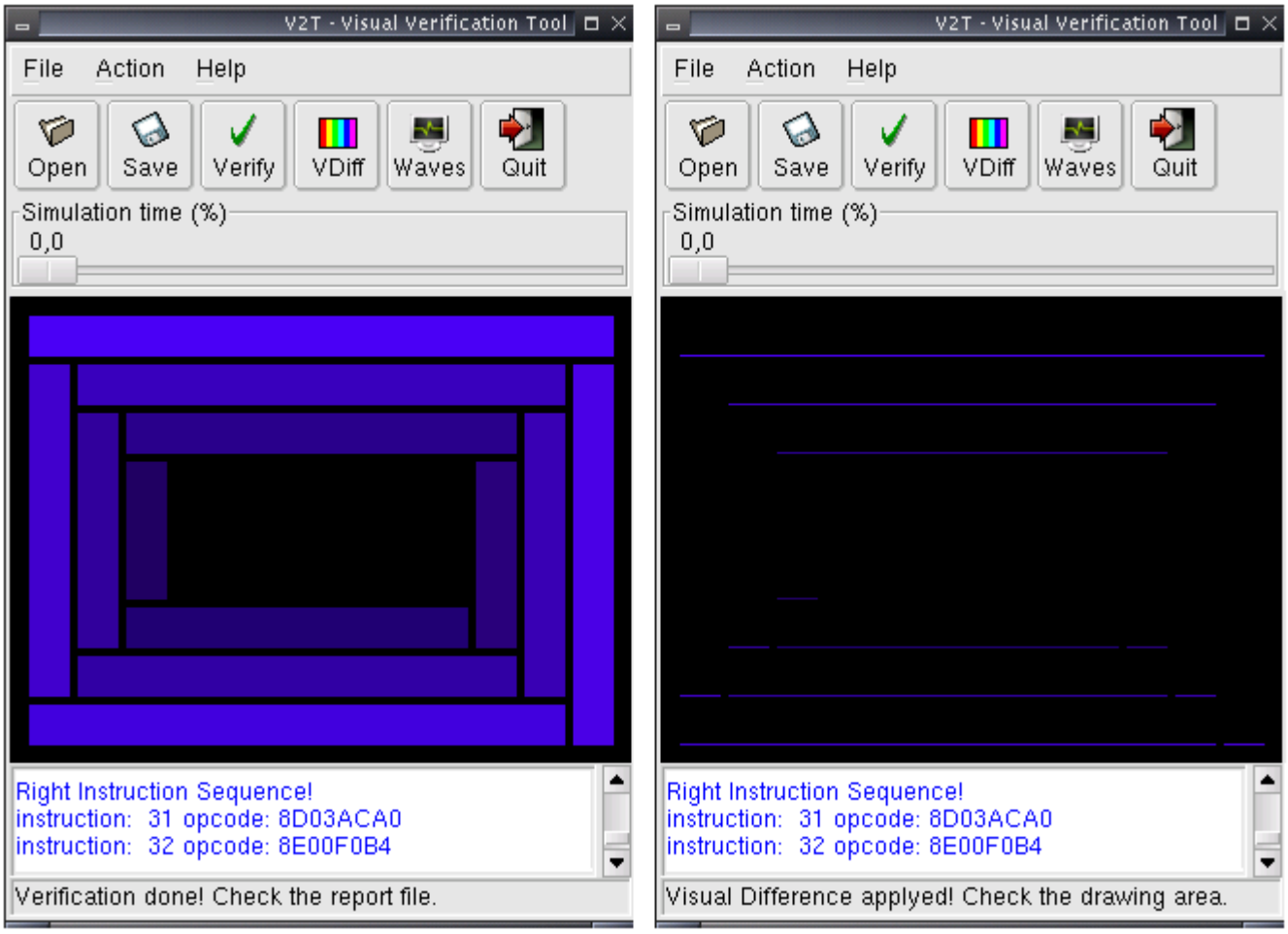

Figura 5.7: Ferramenta V^2T: Antes e após a verificação automática e a diferença visual para a verificação de preenchimento de retângulos.

5.3 Resultados da Prototipação

5.3.1 Plataforma para Prototipação

Na fase da prototipação, foi necessária a utilização de uma plataforma em que o bloco de computação gráfica pudesse ser inserido e que oferecesse saídas VGA ou XGA para a visualização das primitivas em um monitor.

A plataforma disponibilizada foi a proposta por Moreira et al. [61] e foi desenvolvida para inspeção ótica automática AOI (do Inglês, *Automatic Optical Inspection*). A Figura 5.8 mostra em detalhes esta plataforma.

Esta plataforma tem como núcleo principal de processamento uma FPGA. Já que, por ser inerentemente paralela, possibilita uma maior capacidade processamento das imagens capturadas na inspeção ótica. Esta FPGA também implementa toda a lógica de interface, de controle e ainda realiza processamento das imagens em tempo real [61]. O bloco de computação gráfica em hardware verificado foi inserido nesta FPGA.

Esta plataforma possui um sensor CMOS que é utilizado para adquirir, digitalizar

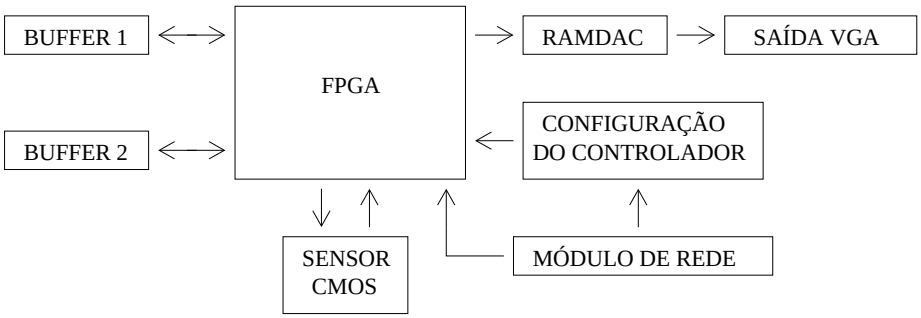

Figura 5.8: Plataforma utilizada para prototipação.

e enviar as imagens para a FPGA. Esta, por sua vez, escreve a imagem recebida em um dos buffers. Dois buffers são utilizados, o primeiro obtém e armazena a imagem mais atual, e o segundo buffer faz o tratamento da imagem obtida anteriormente. A plataforma ainda possui um RAMDAC que possibilita a visualização em um monitor comum das imagens disponíveis nos buffers. Um módulo de rede existente nesta plataforma possibilita a interação da mesma com outros dispositivos e também fornece alternativas para a configuração da mesma por meio do módulo de configuração do controlador.

Como afirmado anteriormente, o bloco de computação gráfica em hardware verificado foi inserido na FPGA. Isto possibilitou que, a medida que este desenhava as primitivas nos buffers, os outros módulos da plataforma se encarregavam de mostrá-las no monitor.

5.3.2 Visualização das Primitivas do Bloco Verificado

Após a detecção e correção de todos os erros exemplificados na subseção 5.2, iniciou-se a fase de prototipação na plataforma disponível. As Figuras 5.9, 5.10 e 5.11 mostram alguns casos de teste gerando primitivas que foram desenhadas de forma correta no protótipo. É bom ressaltar que todos os desenhos estão em tons de cinza devido a limitações da plataforma para resoluções XGA.

A Figura 5.9 mostram as primitivas de desenho de círculos e retângulos vazios. A Figura 5.10 mostram as primitivas de desenho de círculo e retângulos cheios. A Figura 5.11 mostra a primitiva de triângulo vazio. É importante ressaltar que a primitiva de preenchimento de triângulo não foi executada no protótipo porque o número de elementos lógicos da FPGA utilizada não era suficiente para testar todo o co-processador.

5.4 Análise Crítica dos Resultados

Este capítulo demonstrou a utilização da arquitetura de verificação proposta no presente trabalho através de um estudo de caso. Este estudo de caso passou pelas etapas de

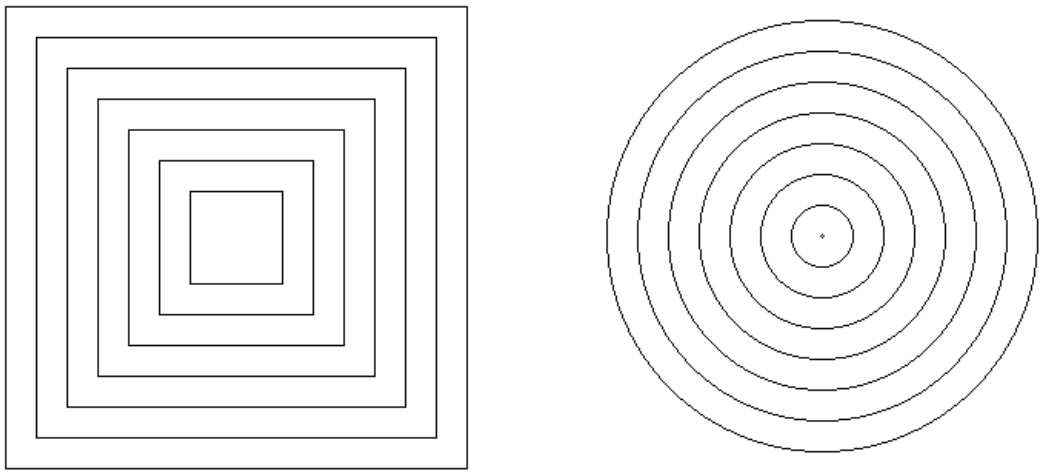

Figura 5.9: Primitivas de desenhos de círculo e retângulos vazios no protótipo.

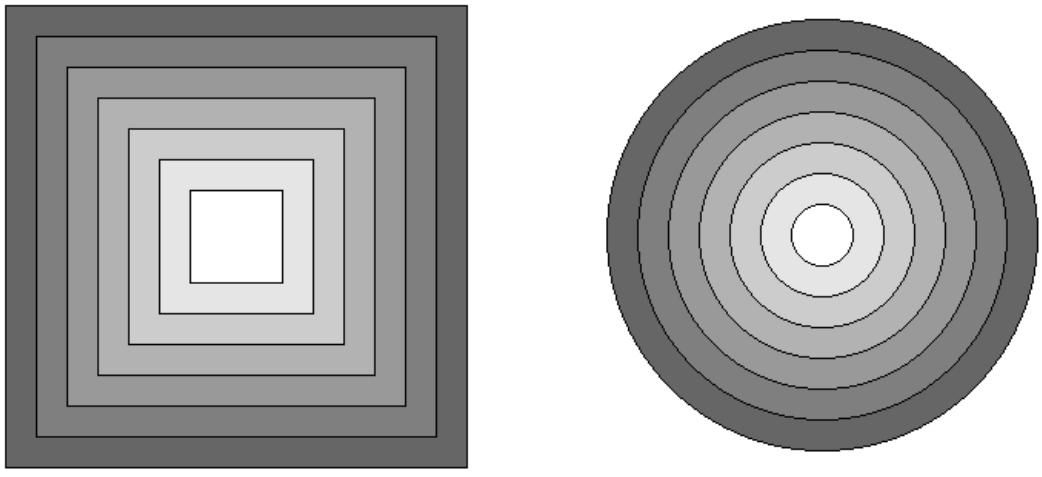

Figura 5.10: Primitivas de desenhos de círculo e retângulos cheios no protótipo.

projeto da especificação executável (capítulo 4), verificação por sub-blocos e verificação em nível de sistema (seção 5.2) e, finalmente, a prototipação do bloco de computação gráfica verificado (seção 5.3).

Grande parte das vantagens da verificação por meio da arquitetura proposta, enunciadas na seção 3.7, foram demonstradas. Isto fornece um indicativo de como esta arquitetura pode ser utilizada de forma eficaz encurtando o estágio de verificação de projetos de blocos de computação gráfica em hardware. No entanto, para uma análise justa, algumas limitações encontradas devem ser ressaltadas.

A primeira limitação acontece na elaboração da especificação executável proposta pela arquitetura. Um grande cuidado deve ser tomado para que esta especificação não

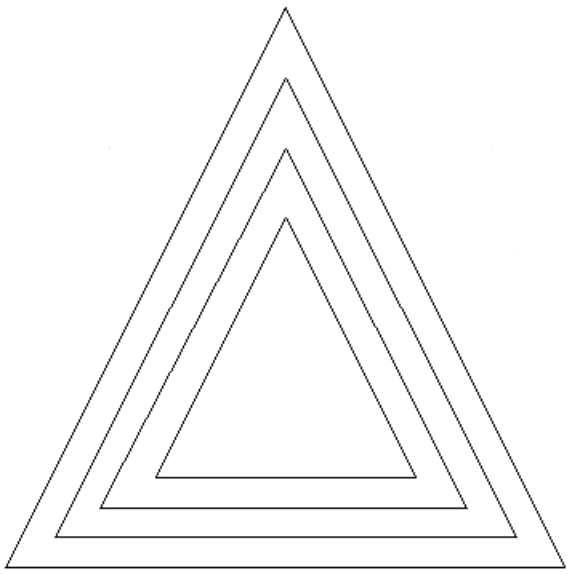

Figura 5.11: Primitivas de desenhos de triângulo vazio no protótipo.

seja feita em um nível muito alto de abstração, porque a eficiência da arquitetura proposta pode ser um pouco reduzida. Um exemplo é que um algoritmo de preenchimento de polígonos poderia ser utilizado em detrimento da divisão entre preenchimento de círculos, retângulos e triângulos feita neste trabalho. A utilização de uma especificação em um nível de abstração muito alto, não impediria que a arquitetura fosse utilizada, porém, informações e dificuldades enfrentadas no desenvolvimento do bloco em HDL poderiam não aparecer, perdendo-se, assim, parte importante do processo.

Outra limitação desta arquitetura surge porque a mesma baseia-se na metodologia de verificação funcional. Isto é, com esta metodologia não é possível saber quando o ciclo de verificação termina. Para isso, uma das melhores soluções é a utilização de ferramentas de cobertura de código como descrito na seção 2.2.4.

Além destas, outra restrição da arquitetura é que as ferramentas desenvolvidas (GVT e V^2T) estão muito vinculadas a implementação do bloco de computação gráfica e de sua especificação executável. Uma definição de padrões seria necessário para que estas ferramentas fossem utilizadas na verificação de blocos de computação gráficas quaisquer. Nesta etapa, uma análise do impacto de memória dos arquivos gerados na simulação também poderiam ser feitos.

Finalmente, é importante ressaltar que a arquitetura de verificação proposta não soluciona problemas muito comuns na prototipação. A garantia da exatidão, seja na verificação por sub-blocos, através das asserções e da ferramenta GVT; ou ainda, pela verificação automática em alto nível com a ferramenta V^2T, não implica em uma execução correta no protótipo. Já que a arquitetura propõe apenas a verificação para modelos funcionais. Uma análise por meio de modelos de tempo disponíveis nas ferramentas de desenvolvimento deve ser utilizada.

Capítulo 6

Conclusões e Trabalhos Futuros

A crescente demanda por produtos baseados em circuitos integrados, a necessidade de se colocar estes produtos no mercado em períodos de tempo cada vez menores e o crescimento exponencial do número de elementos em um circuito integrado tornou o processo de projeto e verificação dos mesmos muito complexo. Neste cenário, o lançamento de produtos com defeitos acaba sendo muito freqüente. Na maior parte dos casos, estes defeitos ocasionam sérios prejuízos financeiros para as empresas desenvolvedoras, ou ainda, em alguns casos, vidas são colocadas em risco. Por isso, as técnicas de verificação de circuitos integrados tornaram-se a parte mais importante e a que mais demanda tempo neste tipo de projeto.

O presente trabalho apresenta uma arquitetura para verificação de blocos de computação gráfica em hardware. Os projetos de hardware de computação gráfica, mesmo que muito simples, estão sendo criados desde a década de 80. Todavia, atualmente, este problema reaparece com maiores restrições de tempo, maior escala de integração do *chips*, e novas metodologias de projeto e de verificação de circuitos integrados. Apesar de toda esta experiência acumulada nos projetos de blocos de computação gráfica em hardware, até hoje, as metodologias utilizadas para verificá-los são as mesmas utilizadas para qualquer outro tipo de bloco.

O presente trabalho propõe um arquitetura de verificação específica para blocos de computação gráfica. Esta arquitetura é pode ser dividida em quatro estágios: a elaboração de uma especificação executável para o bloco, o projeto do bloco em linguagem de descrição de hardware, a verificação funcional por sub-blocos e a verificação funcional em nível de sistema. Para o primeiro estágio da verificação, cada um dos sub-blocos é verificado em dois níveis de abstração: nível o gráfico, oferecido por um visualizador integrado com uma interface PLI (GVT); e o nível-RT, oferecido por propriedades definidas por asserções no código. Para o segundo estágio, uma ferramenta CAD (V^2T) foi projetada, implementada e utilizada na verificação em nível de sistema. Esta ferramenta possibilita uma verificação automática em dois níveis de abstração. Em alto nível (graficamente), as primitivas de desenho são verificadas detectando se as ins-

truções que a geraram foram executadas de acordo com a especificação. Em nível baixo, através das asserções que representam propriedades da especificação. Além disso, esta ferramenta possui um visualizador de formas de ondas, um visualizador de primitivas gráficas e um mecanismo de verificação automática da implementação baseado-se na especificação executável.

Um bloco de computação gráfica foi implementado para ser o objeto de verificação da arquitetura proposta. Este bloco é composto por todas as primitivas de computação gráfica em duas dimensões que podem ser desenhadas sem o uso de artifícios complexos. Este bloco foi verificado pela arquitetura proposta e alguns erros de difícil detecção foram encontrados. Devido as vantagens oferecidas pela arquitetura como: a verificação por partes, a verificação em nível de sistema, a verificação automática em alto e baixo nível de abstração e a facilidade na visualização simultânea de dados em diferentes níveis de abstração, o processo de verificação tornou-se mais eficiente. Além disso, o bloco de computação gráfica foi prototipado para evidenciar que o mesmo havia sido amplamente verificado e apresentava-se funcionalmente correto, fornecendo um indicativo de validade da arquitetura de verificação proposta.

Em relação aos trabalhos futuros, pode-se notar que a arquitetura e as ferramentas propostas estão muito direcionadas para a implementação do bloco de computação gráfica desenvolvido. É necessário a definição de padrões como formatos de arquivos para que outros tipos de blocos de computação gráfica sejam verificados. Além disso, a ferramenta não verifica automaticamente nenhuma primitiva que tenha sobreposto o seu desenho sobre outra. Desta forma, faz-se necessário um extensão da ferramenta para tratar estes tipos de casos mais complexos, possibilitando, futuramente, a verificação inclusive de animações. Outra extensão do presente trabalho, pode ser feita através da generalização da metodologia proposta criando ferramentas de verificação em dois ou mais níveis de abstração para blocos de processadores de uso geral, blocos de rede, dentre outros.

Uma abordagem mais interessante para a utilização desta arquitetura seria através de uma série de refinamentos do mais alto nível até o mais baixo, fazendo com que níveis mais altos sirvam de especificação para os mais baixos. Algumas linguagens de descrição de hardware como SystemC possibilitam este tipo de refinamento. Desta forma, existiriam diversos modelos para o projeto do circuito integrado em diferentes níveis de abstração. A arquitetura proposta poderia estabelecer mecanismos de verificação entre dois níveis de abstração vizinhos em que os modelos de computação não são tão distintos. Isto evitaria um grande salto entre modelos de computação como o utilizado neste trabalho que parte de modelos em linguagens de alto nível diretamente para linguagens de descrição de hardware.

Referências Bibliográficas

[1] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. Pearson Prentice Hall, third edition, 2004.

[2] Janick Bergeron. *Writing Testbenches*. Kluwer Academic Publishers, 2000.

[3] Prabhat Mishra and Nikil Dutt. Funcional validation of programmable architectures. In *Proceedings of the EUROMICRO Systems on Digital Syste Design*, 2004.

[4] Gordon Moore. Cramming more componentes onto integrated circuits. *Electronics*, 38(8), 1965.

[5] Harry Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2004.

[6] Rolf Drechsler. *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[7] 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.

[8] B. Beizer. The Pentium Bug - an Industry Watershed. Testing Techniques Newsletter, TTN, September 1995.

[9] Tom Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of Design Automation Conference*, 2003.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[11] Michael Keating and Pierre Bricaud. *Reuse Methodology Manual - For System-on-a-Chip Designs*. Kluwer Academic Publishers, 2001.

[12] Aarti Gupta. Formal hardware verification methods: a survey. *Form. Methods Syst. Des.*, 1(2-3):151–238, 1992.

[13] Douglas J. Smith. *HDL chip design: A practical guide for designing, synthetizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone Publications, seventh edition, 2000.

[14] IEEE Standard 1364-2001. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language. IEEE Inc., 2001.

[15] Tony Bybell. GTKWave Electronic Waveform Viewer. Disponível em: http://www.cs.man.ac.uk/apt/tools/gtkwave/, 2005.

[16] A. Gupta, S. Malik, and P. Ashar. Toward formalizing a validation methodology using simulation coverage. In *Proceedings of Design Automation Conference*, june 1997.

[17] C. N Liu and J. Y. Jou. Efficient coverage analysis metric for HDL design validation. In *IEE Proceedings - Comput. Digital Technology*, volume 148, january 2001.

[18] Dinos Moundanos, Jacob A. Abraham, and Yatin V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, January 1998.

[19] T. H. Wang and C. G. Tan. Practical code coverage for verilog. In *Proceedings of the 1995 IEEE International Verilog HDL Conference*, pages 99–104. IEEE Computer Society, 1995.

[20] Carl-Johan Seger. An introduction to formal hardware verification. *ACM TOG*, 4(2):147–169, April 1985.

[21] Lionel Bening. A two-state methodology for RTL logic simulation. In *Proceedings of Design Automation Conference*, 1999.

[22] Rajeev Murgai and Masahiro Fujita. Some recent advances in software and hardware logic simulation. In *10th IEEE International Conference on VLSI Design*, January 1997.

[23] Gopi Ganapathy, Ram Narayan, Glenn Jorden, and Denzil Fernandez. Hardware emulation for functional verification of K5. In *Proceedings of Design Automation Conference*, 1996.

[24] Young-II Kim, Wooseung Yang, Young-Su Kwon, and Chong-Min Kyung. Communication-efficient hardware acceleration for fast funcional simulation. In *Proceedings of Design Automation Conference*, 2004.

[25] R. E. Bryant. Tutorial on formal verification of hardware. In *Proceedings of Design Automation Conference*, 1991.

[26] E. Mendelson. *Introductin to Mathematical Logic*. D. Van Nostrand Company, Inc., 1964.

[27] R. S. Boyer and J. S. Moore. Proof-checking, theorem-proving and program verification. *Contemporary Mathematics*, 29:119–132, 1984.

[28] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1996.

[29] Computational Logic Inc. Computational Logic Inc website. http://www.cli.com, 2005.

[30] Warren Alva Hunt. Microprocessor design verification. Computer Logic Inc. Technical Report, 1989.

[31] Mike Gordon. Proving a computer correct with the LCF_LSM hardware verification system. Technical Report 42, University of Cambridge Computer Laboratory, 1983.

[32] Bishop Brock and Warren A. Hunt. Report on the formal specification and partial verification of the VIPER microprocessor. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 91–98, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.

[33] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, pages 91–98. Springer, 1981.

[34] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *In Proceedings of the 5th International Symposium on Programming*, pages 337–350.

[35] 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.

[36] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th* IEEE *Symposium on Foundations of Computer Science*. IEEE Press, 1977.

[37] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of Design Automation Conference*, pages 46–51, 1990.

[38] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1985.

[39] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the 1991 International Conference on VLSI*, pages 49–58, August 1991.

[40] 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.

[41] 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.

[42] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[43] 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.

[44] Lonn Fiance and Robert Vallelunga. Complex FPGAs require equivalence checking. Xilinx XCell Journal, 2nd Quarter 2005.

[45] Accelera. *Accellera proposed standard Open Verification Library Users Reference Manual*. 2003.

[46] Inc Falanx Microsystems. Falanx Microsystems website. http://www.falanx.com, 2005.

[47] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surving the SOC revolution: a guide to platform-based design*. Kluwer Academic Publishers, 1999.

[48] Axel Jantsch. *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers, 2004.

[49] GTKTeam. Gimp Toolkit Library. http://www.gtk.org, 2005.

[50] Wilson Snyder. Dinotrace. Disponível em http://www.verypool.com/dinotrace, 2005.

[51] Pragmatic C Software Corp. GPL CVER Verilog Simulator. Disponível em: http://www.pragmatic-c.com/gpl-cver/, 2005.

[52] Altera. Altera website. http://www.altera.com, 2005.

[53] Xilinx. Xilinx website. http://www.xilinx.com, 2005.

[54] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[55] M. L. V. Pitteway. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *Computer Journal*, 10(3):282–289, November 1967.

[56] J. R. Van Aken and M. Novak. Curve-drawing algorithms for raster display. *ACM TOG*, 4(2):147–169, April 1985.

[57] J. D. Foley, A. van Dam, S. K. Freiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, second edition, 1990.

[58] J. E. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 1977.

[59] Ali Mohamed Abbas, László Szirmay-Kalos, Gábor Szijártó, Tamás Horváth, and Tibor Fóris. Quadratic interpolation in hardware phong shading and texture mapping. *IEEE*, 20(2):100–106, February 1977.

[60] F. V. Andrade, J. A. M. Nacif, C. N. Coelho Jr., A. O. Fernandes, Hao Chi Wong, L. F. Etrusco, and L. H. Barbosa. When bits are not bits and bytes are not bytes: Validating computer graphics cores at higher level of abstraction. In *Proceedings of the 6th IEEE Latin-American Test and Workshop*, April 2005.

[61] L. F. E. Moreira, J. L. Silvino, J. C. D de Melo, and C. N. Coelho Jr. Distributed network platform for automatic optical inspection. *IEE Eletronic Letters*, August 2003.