

Profissionalizando o retrogaming: estrutura de verificação para o MiSTer FPGA

Victor Hugo Lopes da Silva

Resumo—A indústria microeletrônica movimenta bilhões de dólares anualmente, e a integridade dos circuitos digitais é essencial para evitar falhas que podem resultar em perdas significativas. Para garantir a conformidade com as especificações, são empregadas técnicas avançadas de verificação formal. Este trabalho apresenta um estudo de caso focado no projeto MiSTer FPGA, um sistema complexo desenvolvido por entusiastas, que reproduz consoles de videogame clássicos. Apesar de sua sofisticação, o projeto original não incluía uma estrutura formal de verificação. Para suprir essa necessidade, foram realizadas modificações no código, viabilizando a verificação formal com ferramentas como o JasperGold, além de profissionalizar o projeto. A nova estrutura foi validada com sucesso, assegurando a consistência lógica do projeto e contribuindo para seu desenvolvimento futuro.

Index Terms—Sistemas Digitais, FPGA, Verificação Formal.

I. INTRODUÇÃO

O avanço da tecnologia e o crescimento da indústria da microeletrônica trouxeram desafios cada vez mais complexos para o projeto de sistemas digitais, impulsionados pela demanda por aplicações mais poderosas. Sistemas modernos operam com 64 bits e processadores como o AMD Ryzen Threadripper 3970X, que possuem até 32 núcleos, executam múltiplas tarefas em paralelo [1]. Além disso, o Intel Core i9 de 14ª geração alcança velocidades de até 6,0 GHz, proporcionando processamento extremamente rápido [6].

Devido à complexidade e aos altos custos de produção, assegurar a funcionalidade dos circuitos é essencial. Erros em projetos podem resultar em grandes prejuízos, como no caso do Pentium FDIV bug, que custou à Intel cerca de \$475 milhões para corrigir [4]. Esse exemplo mostra que, mesmo empresas com vastos recursos, podem cometer erros em projetos complexos, justificando o fato de que até 75% do tempo de fabricação de um chip é dedicado à verificação [3].

A verificação é uma etapa fundamental no desenvolvimento de circuitos, garantindo que o sistema funcione conforme as especificações e detectando possíveis falhas. Embora os testes e simulações sejam importantes, eles não asseguram que um chip esteja totalmente livre de erros. Para lidar com essa limitação, a indústria microeletrônica adota técnicas avançadas, como a verificação formal, que utiliza métodos matemáticos para provar que o circuito se comporta corretamente em todos os possíveis estados operacionais. Essa abordagem explora o espaço de estados de forma simbólica,

o que reduz significativamente o esforço computacional ao representar múltiplos estados de uma só vez [2].

Embora a verificação formal seja amplamente utilizada na indústria microeletrônica devido à sua eficácia em detectar falhas complexas, ela ainda é pouco explorada nos cursos de graduação em engenharia elétrica.

A fim de estudar e aplicar conceitos de verificação formal, este trabalho tem como objetivo fornecer um estudo de caso prático, demonstrando como essa metodologia pode ser implementada efetivamente. Para isso, escolheu-se o projeto o MiSTer FPGA como base para desenvolver uma estrutura formal de verificação.

O MiSTer FPGA é uma plataforma de código aberto, baseada em FPGA (Field-Programmable Gate Arrays), que reproduz funcionalmente consoles clássicos de videogames, além de sistemas de computação legados. Voltado para a comunidade de retrogaming — movimento relacionado à prática de jogar e colecionar videogames antigos, muitas vezes em consoles e computadores clássicos — o projeto permite que esses sistemas sejam recriados de forma precisa em nível de hardware, reproduzindo o comportamento original dos dispositivos.

O MiSTer FPGA é altamente modular e pode ser reconfigurado em diferentes “cores”, que são essencialmente os circuitos eletrônicos que definem cada sistema reproduzido. Sendo assim, essa plataforma é amplamente utilizada por entusiastas de retrogaming e retrocomputação, sendo uma ferramenta poderosa para preservar a história da computação e dos jogos eletrônicos.

Embora seja um projeto de código aberto descrito linguagens de descrição de hardware (HDL), o MiSTer enfrenta barreiras significativas para colaboração e modificação, como a falta de documentação e a participação limitada de colaboradores. Além disso, o MiSTer é um projeto extremamente grande mas feito de forma completamente artesanal por poucas pessoas, o que inviabiliza qualquer tipo de modificação (que é extremamente desejável para que essa plataforma funcione em outras FPGAs). Em várias partes do código fonte do circuito, não há quase nenhuma legibilidade, o que dificulta severamente modificações.

Este trabalho visa mudar essa realidade, proporcionando uma metodologia estruturada de verificação formal que facilita a experimentação e modificação dentro do projeto MiSTer. A introdução de uma estrutura formal de verificação não só melhora a colaboração, mas também garante que as alterações e inovações realizadas no projeto sejam funcionalmente corretas. Com isso, espera-se que o MiSTer se torne mais acessível para

a comunidade de desenvolvedores, promovendo um ambiente de desenvolvimento mais robusto e confiável.

Como o projeto MiSTer reproduz os circuitos de uma variedade de consoles, optou-se por focar em apenas um para este estudo: o Nintendo Entertainment System (abreviado como NES, e popularmente conhecido como Nintendinho no Brasil). A escolha do NES se justifica por seu status icônico e sua popularidade duradoura. Além disso, sua arquitetura, desenvolvida na década de 1980, apresenta uma complexidade suficiente para criar um caso de estudo interessante. O 'core' do NES no MiSTer, por exemplo, contém 47.908 linhas de código e 181 módulos (para outros "cores" mais complexos esses números são ainda maiores), o que proporciona um ambiente ideal para experimentar ferramentas de verificação, que são padrões na indústria, mas ainda pouco exploradas durante a graduação.

Este projeto proporciona um equilíbrio entre complexidade e viabilidade, justificando a escolha do NES. O MiSTer, sendo um projeto de código aberto de grande escala, serve como um exemplo real e robusto para a aplicação de técnicas de verificação formal, permitindo a exploração prática e o aprendizado aprofundado dessas ferramentas.

Este trabalho desenvolveu uma estrutura de verificação formal para o core do NES no projeto MiSTer FPGA, criando scripts que permitem que ferramentas de verificação formal elaborem e validem propriedades do projeto. Isso facilita e torna possíveis modificações futuras extremamente necessárias neste projeto, como a refatoração do código para torná-lo mais portátil, permitindo sua execução em diferentes plataformas e FPGAs.

O restante deste artigo está organizado da seguinte forma. Na seção II é apresentado um embasamento teórico de todos os conceitos necessários para entendimento total deste trabalho. A seção III apresenta a metodologia utilizada, e na seção IV mostra de fato o que foi feito, bem como os respectivos resultados. E por fim, a seção V resume e conclui o trabalho.

II. FUNDAMENTAÇÃO TEÓRICA

A. Projeto de Circuitos Digitais

Circuitos digitais são sistemas eletrônicos que operam com base na lógica binária, sendo projetados para processar e manipular informações na forma de bits (0s e 1s) e executam funções como adição, subtração, multiplexação, memória e controle. Os circuitos digitais são construídos para processar, armazenar e transmitir dados digitais de forma precisa e eficiente, sendo, portanto, a base para a construção de computadores, sistemas embarcados, e diversos dispositivos eletrônicos modernos.

O projeto desses circuitos digitais é uma área complexa que envolve várias etapas para transformar uma ideia inicial em um circuito elétrico funcional. Este processo é ilustrado na Figura 1, sendo dividido em diferentes níveis principais de abstração: Nível de Sistema, Nível Lógico e Nível Físico. Cada nível de abstração oferece uma perspectiva distinta e utiliza ferramentas e técnicas específicas para garantir que o circuito final atenda às especificações e requisitos de desempenho.

Circuitos digitais operam com lógica binária, processando informações na forma de bits (0s e 1s) para executar funções como aritmética, controle e memória. Eles são fundamentais para a construção de computadores e dispositivos eletrônicos modernos. O projeto desses circuitos é complexo e envolve várias etapas (ilustradas na Figura 1), organizadas em diferentes níveis de abstração: Nível de Sistema, Nível Lógico e Nível Físico, cada um com ferramentas e técnicas específicas para garantir que o circuito atenda às especificações de desempenho."

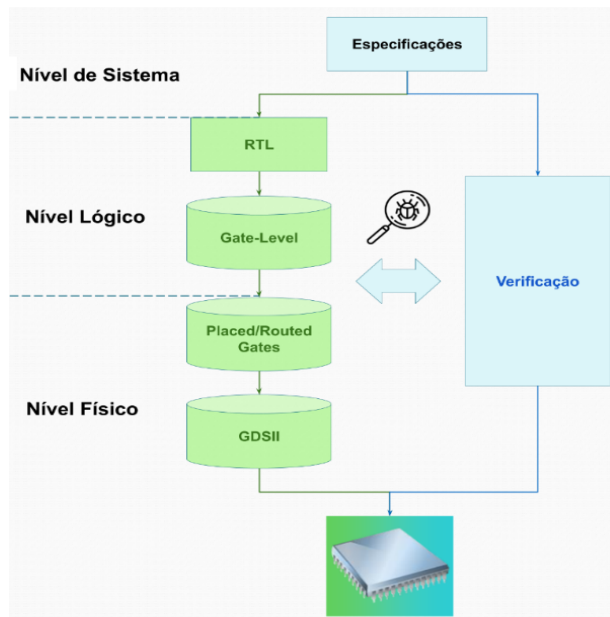


Figura 1. Fluxo de Projeto de um circuito digital

1) Nível de Sistema:

No nível de sistema, o foco está na definição dos requisitos gerais e das especificações funcionais do circuito. Este é o ponto de partida onde se compreende o que o circuito deve fazer e como ele deve se comportar em diferentes condições operacionais. É nesta etapa que são elaboradas as especificações iniciais, incluindo a funcionalidade desejada, interfaces e restrições de desempenho.

Para exemplificar essa etapa, podemos pensar em um projeto de um circuito cujo objetivo é controlar a cor de um LED RGB. E para este projeto, tem-se as seguintes especificações:

- O estado do LED poderá ser alterado por um botão ou por um sinal de clock.
- Estados possíveis do LED
 - Estado 0: LED azul
 - Estado 1: LED vermelho
 - Estado 2: LED amarelo
- Transições de estado
 - O sistema começa no Estado 0
 - A cada ciclo de clock, o circuito transita para o próximo estado.

– Após o Estado 2, o LED retorna ao Estado 0.

O funcionamento e a transição de todos os estados do circuito podem ser ilustrados pelo diagrama de estados mostrado na Figura 2.

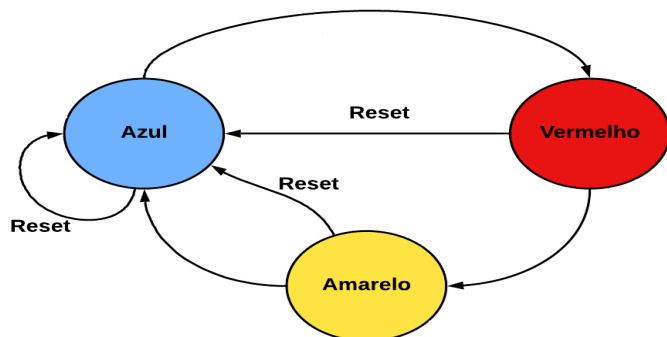


Figura 2. Diagramas de estados do Circuito. Nota-se que pelo diagrama e as especificações, é impossível o LED ir direto do estado Azul para o Amarelo.

2) Nível Lógico:

O nível lógico envolve o desenvolvimento da Representação em Nível de Registradores (RTL, do inglês Register Transfer Level) do circuito. O RTL é uma abstração que descreve a operação de um circuito digital em termos de fluxos de dados entre registradores e as operações lógicas realizadas nesses dados. Os registradores são elementos de armazenamento que mantêm os dados ao longo dos ciclos de clock, enquanto as operações lógicas incluem funções como AND, OR, NOT e aritméticas.

Para modelar circuitos digitais no nível RTL, linguagens de descrição de hardware (HDL) como Verilog, System Verilog e VHDL são utilizadas. As linguagens HDL conseguem criar representações de alto nível de um circuito, permitindo que engenheiros descrevam a estrutura e o comportamento do circuito de maneira textual, possibilitando a organização e leitura de projetos grandes e complexos, além de facilitar a simulação, verificação e síntese do sistema projetado.

Voltando ao exemplo do circuito controlador de luminosidade de um LED, a representação em RTL pode ser dada por meio da linguagem verilog como pode ser observado no Quadro 1.

```

1 module fsm_led (
2     input clk,
3     input reset,
4     output reg [1:0] state_led
5 );
6
7 always @(posedge clk or posedge reset) begin
8     if (reset) begin
9         state_led <= 2'b00;
10    end else begin
11        case (state_led)
12            2'b00: begin
13                state_led <= 2'b01; //vermelho
14            end
15            2'b01: begin
16                state_led <= 2'b10; //amarelo
17            end
18            2'b10: begin

```

```

19                state_led <= 2'b00; //azul
20            end
21        endcase
22    end
23 end
24 endmodule

```

Quadro 1. Circuito de estados do LED em Verilog

No código foram definidas duas entradas: uma de clock (que poderia representar o botão) e um reset para o LED voltar ao estado inicial. E a saída foi declarada como reg (registrador) para manter o estado atual, sendo alterado sempre na borda positiva do sinal de clock.

3) Nível Físico:

O código HDL é convertido em uma netlist de portas lógicas, que consiste em uma lista detalhada dos componentes eletrônicos do circuito e os nós aos quais estão conectados, formando os elementos fundamentais no nível de portas (Gate Level). Em seguida, essas portas são posicionadas e interconectadas no chip, garantindo que todas as conexões físicas sejam realizadas corretamente. Essa etapa, conhecida como Place and Route, é ilustrada na Figura 3.

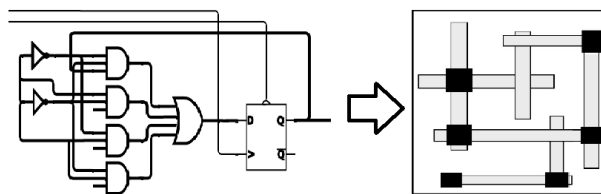


Figura 3. Na etapa de place and route os componentes e as conexões do circuito são mapeados para a fabricação do Chip (Imagem ilustrativa do processo).

Após o mapeamento dos componentes do circuito, é gerado o arquivo GDSII, um formato binário padrão da indústria que representa o layout físico do projeto, transformando o projeto lógico em uma forma que pode ser fabricada. A implementação também pode ser realizada em FPGAs, dispositivos reconfiguráveis compostos por blocos lógicos e interconexões programáveis. A configuração dos FPGAs é feita por ferramentas de síntese, que traduzem a descrição do hardware em um mapa lógico, definindo como os recursos do FPGA serão interconectados para implementar o circuito.

B. Verificação de Circuitos Digitais

A verificação é uma etapa essencial para garantir que o sistema funcione conforme as especificações, identificando e corrigindo falhas. Tradicionalmente, isso é feito por meio de testes e simulações que avaliam o comportamento do circuito. Quando falhas são encontradas, o circuito passa por depuração até alcançar confiabilidade satisfatória. Entretanto, esse tipo de verificação enfrenta desafios em termos de exaustividade, controlabilidade e observabilidade, dificultando a cobertura de todos os cenários possíveis. Para superar essas limitações,

projetistas têm adotado a verificação formal, que permite explorar exhaustivamente os estados e comportamentos do sistema, sem a necessidade de testar múltiplas combinações de estímulos, identificando erros que a simulação tradicional pode não detectar.

A aplicação de técnicas formais de verificação não é trivial, exigindo um conhecimento aprofundado de métodos matemáticos e uma compreensão detalhada do projeto a ser verificado. Profissionais precisam se capacitar adequadamente para aplicar essas técnicas de maneira eficaz. Além disso, a verificação formal demanda um alto esforço computacional, pois envolve a exploração exhaustiva de todos os estados e comportamentos possíveis do sistema.

Para facilitar essa tarefa e minimizar a introdução de erros humanos, foram criados os provadores de teoremas (ou verificadores de propriedades). Esses assistentes de prova são softwares projetados para auxiliar os verificadores durante a execução de seus testes, automatizando a verificação de propriedades específicas e provendo um alto grau de confiança na correção do sistema. Os provadores de teoremas utilizam algoritmos avançados para realizar provas matemáticas de maneira eficiente, reduzindo a carga de trabalho manual e permitindo que os verificadores se concentrem em aspectos mais críticos do projeto.

1) Verificação Funcional x Verificação Formal:

Como mencionado anteriormente, a verificação funcional é a abordagem mais tradicional na validação de projetos de circuitos digitais. Nesse método, são gerados estímulos para um modelo do circuito, geralmente descrito em HDL, e o comportamento resultante é observado e comparado com os resultados esperados. A simulação é uma ferramenta essencial, pois permite a detecção de diversas falhas no projeto antes de sua implementação física. No entanto, sua eficácia é limitada, especialmente em circuitos com um espaço de estados muito grande, ou seja, com uma quantidade significativa de bits de entrada e elementos lógicos, como flip-flops. Nessas situações, a simulação pode não cobrir todas as possíveis combinações de estados, deixando potenciais falhas não identificadas.

Como alternativa à simulação, existe a verificação formal, uma técnica avançada de validação que utiliza métodos matemáticos rigorosos para garantir que o circuito funcione conforme o esperado. Ao contrário da simulação, a abordagem formal verifica sistematicamente todas as possíveis condições operacionais do circuito.

A diferença entre a verificação formal e a simulação pode ser ilustrada pela Figura 4. No diagrama, o espaço de estados do circuito é representado de forma circular. A simulação começa a partir de um estado inicial e, a cada ciclo de clock, explora novos estados, encontrando tanto estados corretos (livres de falhas) quanto, eventualmente, estados com falhas. No entanto, como mostrado na figura, algumas simulações não conseguem detectar todas as falhas do projeto, pois cobrem apenas uma fração do espaço de estados. Por isso, é comum realizar dezenas de milhares de simulações para aumentar

as chances de encontrar bugs. Contudo, essa abordagem não garante a identificação de todas as falhas, pois não é possível testar todos os cenários.

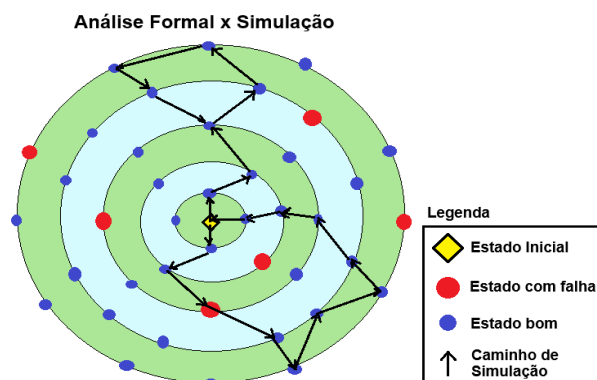


Figura 4. A simulação pode ser vista como caminhos de um estado para outro, enquanto que a análise formal envolve todos os estados por meio de círculos concêntricos cujo diâmetro (espaços de estados verificados) aumenta a cada ciclo de clock.

A verificação formal explora todo o espaço de estados representado pelas circunferências no diagrama da figura, e sua cobertura se expande conforme o número de ciclos de clock aumenta (representado pelo diâmetro das circunferências). Embora seja um processo exaustivo, ele é capaz de identificar todas as falhas do projeto, garantindo uma análise completa do circuito.

A verificação formal não apenas garante uma exploração exaustiva do espaço de estados do circuito, mas também se baseia na definição precisa de propriedades que o projeto deve atender. Essas propriedades descrevem o comportamento esperado do circuito em termos de regras ou condições que devem ser sempre verdadeiras. Essas propriedades são descritas em linguagens especializadas, como SystemVerilog Assertions (SVA), para definir os comportamentos esperados e invariantes do circuito.

Uma ferramenta de verificação formal prova uma propriedade com menor esforço computacional do que uma simulação porque utiliza abstração e análise simbólica para explorar de maneira inteligente e exaustiva o espaço de estados do sistema. Em vez de simular cada possível cenário individualmente, a verificação formal representa e analisa conjuntos de estados e transições simultaneamente. Técnicas como model checking e prova automática de teoremas evitam a necessidade de examinar cada estado separadamente, identificando rapidamente contradições ou confirmando a validade de propriedades, o que reduz significativamente o tempo de processamento comparado à simulação tradicional [8].

2) Verificação formal baseada em propriedades:

As propriedades descritas em SystemVerilog Assertions permitem que o projetista defina explicitamente as condições que o circuito deve satisfazer ao longo de sua operação. Por

exemplo, uma propriedade pode especificar que um sinal de controle deve estar ativado em um determinado ciclo de clock após a ocorrência de um evento específico.

A linguagem SVA contém as seguintes diretivas:

- **Property:** Um comportamento de circuito definido como relações lógicas e temporais entre expressões booleanas e sequenciais;
- **Assert:** Indica que a ferramenta deve verificar se uma propriedade é sempre verdadeira;
- **Assume:** Estabelece uma condição/restrrição para a propriedade;
- **Cover:** Utilizado para demonstrar que determinada propriedade possa ser verdadeira.

Ferramentas de verificação, verificam essas diretivas da linguagem SVA de maneira avançada, contendo várias “engines”, que utilizam diferentes técnicas para explorar o espaço de estados do circuito. Ele compara uma representação do sistema descrito em HDL com a representação formal da propriedade a ser verificada, e avalia se o projeto atende a essa propriedade em todos os cenários possíveis.

Dessa forma, a metodologia de verificação formal pode ser ilustrada por meio de um diagrama, como mostrado na Figura 5. As ferramentas especializadas em verificação realizam a síntese do RTL para gerar um modelo formal do circuito, que é então utilizado na verificação de propriedades. Após a criação desse modelo, a ferramenta verifica as propriedades definidas ao realizar provas matemáticas rigorosas. Se as propriedades forem comprovadas, isso indica que o projeto foi validado com sucesso. Caso contrário, o verificador identifica uma falha e fornece um contraexemplo que demonstra onde o projeto não atende às especificações. Essas falhas podem então ser corrigidas no próprio RTL, seguido de uma nova verificação para garantir que o projeto esteja em conformidade com todas as propriedades definidas.

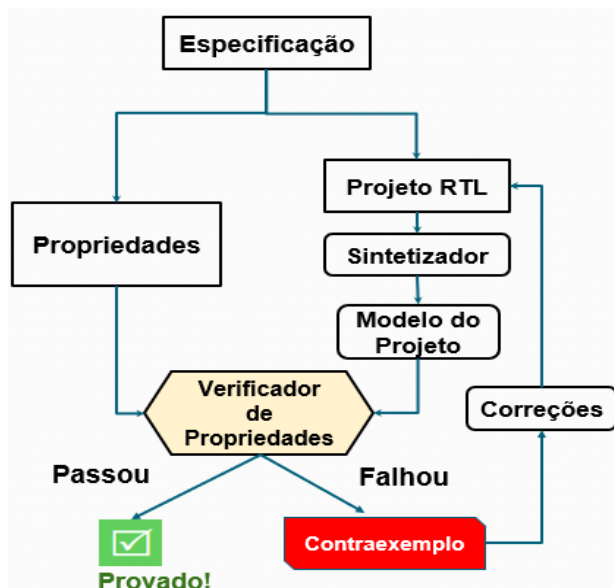


Figura 5. Metodologia de verificação formal baseada em propriedades

Voltando ao exemplo do circuito do LED, suponha-se que se deseja verificar o ciclo de funcionamento do circuito, isto é, se o estado do LED seguirá a sequência correta. Para isso, pode-se criar no próprio HDL uma propriedade em SystemVerilog que verifique esta especificação, mostrada no quadro 2.

```

1 property ciclo;
2   @(posedge clk)
3   disable iff (reset == 1)
4   ((state_led == 2'b00) | => (state_led == 2'b01)
5     ##1 (state_led == 2'b10) ##1 (state_led == 2'
6       b11));
7 endproperty
8 assert property (ciclo);
  
```

Quadro 2. Especificação de propriedade que verifica o ciclo de estados do circuito

A escrita da propriedade em SystemVerilog, deve obedecer a seguinte sintaxe:

- A palavra-chave **property** é usada para definir uma propriedade, que é uma regra que o circuito deve seguir. Neste caso, a propriedade foi nomeada como **ciclo**.
- **@(posedge clk)**: Essa linha indica que a propriedade será verificada em cada borda de subida do sinal de clock (clk). O operador **@** especifica o evento que desencadeia a verificação, e **posedge** refere-se à transição de 0 para 1 do clock.
- **disable iff (reset == 1)**: Aqui, a expressão **disable iff** significa “desabilitar se e somente se”. Isso significa que a verificação da propriedade é suspensa sempre que o sinal **reset** for igual a 1. Em outras palavras, quando o sistema estiver em estado de reset, a propriedade não será verificada.
- O operador **|=>** é conhecido como implicação sequencial. Ele afirma que se a condição à esquerda for verdadeira (**state_led == 2'b00**), então a condição à direita (**state_led == 2'b01**) deve ser verdadeira no próximo ciclo de clock.
- O operador **##1** indica um atraso de um ciclo de clock. Isso significa que, após a transição para 01, o estado do LED deverá alterar no próximo ciclo de clock.
- A palavra-chave **assert** é usada para verificar se a propriedade definida é sempre verdadeira. Se a propriedade **ciclo** for violada durante a verificação, o sistema sinaliza um erro, indicando que o circuito não está se comportando conforme o esperado.

Para fins de demonstração, foi um incluído propositalmente um erro na propriedade, ‘**state_led == 2'b11**’ ao invés de ‘**state_led == 2'b00**’ no último estado, para ver como a ferramenta de verificação indica a falha através da forma de onda (Figura 6).

3) BlackBox:

Em projetos complexos, onde os circuitos possuem múltiplos módulos interconectados, a verificação formal pode se tornar desafiadora devido à grande quantidade de estados e interações entre os componentes. Para lidar com essa com-

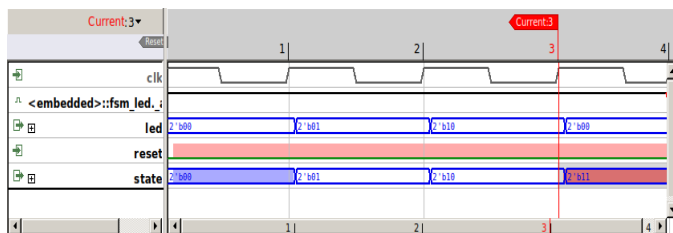


Figura 6. Falha indicada na saída 'state_led' pelo JasperGold durante a transição do terceiro para o quarto ciclo.

plexidade, o uso da técnica de "blackboxing" se torna muito importante.

O conceito de "blackbox" refere-se a tratar certos módulos de um circuito como caixas pretas, abstraindo seu comportamento interno e focando apenas nas entradas e saídas. Essa abordagem simplifica a análise formal ao reduzir o espaço de estados a ser explorado, permitindo que a verificação se concentre em partes críticas do circuito.

Na verificação de circuitos, "blackboxing" isola partes do projeto que não precisam de uma análise detalhada ou cujas funcionalidades já são confiáveis, otimizando recursos computacionais e aumentando a eficiência. Essa técnica é especialmente útil quando módulos não possuem código RTL disponível, como em módulos de terceiros ou IPs.

Por exemplo, se o circuito de LED controlasse um motor de passo e o código RTL desse motor não está disponível, o módulo do motor pode ser tratado como uma "blackbox". O circuito do LED foi verificado usando essa técnica, e um diagrama de conexões (Figura 7) ilustra a interação entre o circuito do LED e o módulo "blackbox", sem necessidade de acessar o código interno do módulo.

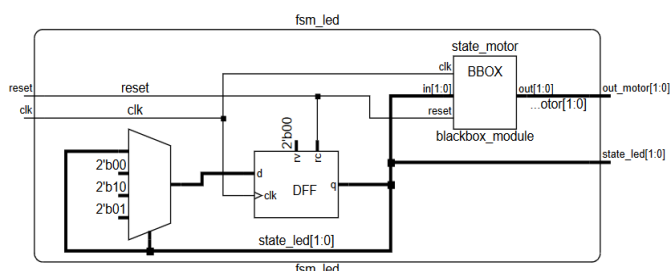


Figura 7. Esquema do circuito elaborado pelo JasperGold

III. METODOLOGIA

A metodologia adotada neste trabalho difere do processo tradicional utilizado na indústria. Normalmente, na indústria, antes do início de um projeto, é elaborada uma especificação detalhada (requisitos do projeto). A partir dessa especificação, engenheiros de desenvolvimento realizam a implementação, enquanto uma equipe separada de engenheiros de verificação escreve propriedades baseadas nesses requisitos, evitando qualquer influência da implementação. No caso do MiSter, essa especificação clara não estava disponível, o que exigiu

um esforço adicional para analisar os módulos do projeto, uma vez que não havia uma documentação formal prévia.

Neste trabalho foram criados scripts necessários para que a ferramenta de verificação formal, JasperGold, pudesse analisar e elaborar o projeto MiSter FPGA. A primeira etapa consistiu em uma análise detalhada do interfaceamento entre todos os blocos e módulos descritos em Verilog e VHDL, com o intuito de mapear as dependências entre eles. Isso permitiu determinar a ordem correta em que os blocos deveriam ser analisados, uma vez que muitos módulos dependiam de outros submódulos já descritos no projeto.

Para simplificar a verificação de módulos cuja implementação interna não estava disponível ou era irrelevante para a análise, foi utilizada a técnica de blackbox, que trata esses módulos como caixas pretas, permitindo que a verificação focasse apenas nos sinais de entrada e saída. Isso facilitou a análise de partes críticas do projeto sem a necessidade de inspecionar os detalhes de cada submódulo.

Em seguida, foi utilizada uma ferramenta de linting da Cadence Design Systems, disponível no JasperGold, para identificar erros e violações na linguagem de descrição de hardware. Com base nas análises dessa ferramenta, foram realizadas modificações no código-fonte do MiSter, corrigindo as violações críticas que impediam a ferramenta de verificação de elaborar o projeto corretamente.

Após essas correções, propriedades específicas foram definidas para serem verificadas, funcionando como um padrão de comportamento esperado para o projeto. Por fim, os experimentos foram realizados em um notebook equipado com processador Intel Core i7 8550U de 8ª geração e 8GB de RAM, utilizando o sistema operacional Ubuntu 21.04 LTS.

IV. DESENVOLVIMENTO

Para usar ferramentas de verificação formal em um projeto digital, é crucial compreender o circuito. No caso do projeto MiSter, foi necessário estudar sua arquitetura, que é desenvolvida na placa DE10-Nano da Intel (anteriormente Altera). Esta placa compacta e versátil contém um FPGA Intel Cyclone V, que permite a implementação de emuladores de videogames clássicos, como o NES. Além do FPGA, a DE10-Nano possui um processador ARM Cortex-A9 (Hard Processor System), que suporta Linux, e oferece diversas interfaces e periféricos, como memória DDR3, portas USB, cartão SD e interface HDMI, fornecendo flexibilidade para o projeto MiSter.

A. Documentação da Arquitetura MiSter

O projeto MiSter FPGA consiste em um framework genérico desenvolvido em Verilog/SystemVerilog que facilita o desenvolvimento e a implementação de cores de consoles de videogame e computadores clássicos na DE10 Nano. Os cores, que são emuladores de sistemas específicos, são criados por colaboradores da comunidade e devem aderir a diretrizes e regras estabelecidas na documentação oficial do projeto para garantir compatibilidade e funcionalidade adequada.

Para incorporar um core ao sistema, é necessário compilar o código-fonte utilizando o software Quartus 17.0.2, seguindo

as configurações fornecidas nos modelos disponibilizados. Isso resulta na geração de um arquivo .rbf, que deve ser transferido para a pasta /_Console no cartão SD configurado da DE10 Nano [7].

Um estudo aprofundado do código fonte do Mister e das referências disponíveis permitiu ilustrar de forma bastante simplificada a arquitetura projeto Mister, como pode ser visto na Figura 8. Embora a Figura 8 se concentre no core do NES, a FPGA pode ser reconfigurada para emular qualquer console compatível, (o que muda é apenas o módulo destacado em vermelho). O core é apenas uma parte do projeto, que inclui diversos circuitos auxiliares necessários para seu funcionamento.

O módulo sys_top é o componente mais alto na hierarquia do MiSTer, integrando todos os circuitos implementados na FPGA Cyclone V e fazendo a interface com componentes externos, como clock, memórias, e I/O. Esse módulo inclui circuitos de PLL (Phase-Locked Loop) para sincronização e o HPS responsável por tarefas de gerenciamento.

Dentro do sys_top, o bloco emu integra o core específico do console (no caso, o NES) com os circuitos auxiliares para vídeo, áudio, e outras funcionalidades. O HPS lê os arquivos .rbf dos cores no cartão SD, configurando a FPGA e gerenciando a transferência de dados para o core, incluindo o controle de jogos e dispositivos USB. O módulo game_loader carrega os jogos na SDRAM, permitindo que o core acesse as informações durante a execução. O core do NES gera sinais de saída de vídeo (em termos de linhas de varredura, ciclos de clock e sinais de controle) para circuitos de vídeo que processam esses sinais convertendo-os em sinais VGA ou HDMI.

B. Criação da Estrutura de Verificação

O desenvolvimento da estrutura de verificação começou com a criação de um fork do repositório do MiSTer para o core NES no GitHub, permitindo que as modificações necessárias fossem realizadas em um repositório próprio. Para o desenvolvimento da estrutura de verificação, o software JasperGold da Cadence Design Systems apresenta-se como a ferramenta de verificação ideal para este trabalho, devido a sua capacidade de lidar com projetos grandes e complexos, além de ser uma ferramenta profissional utilizada na indústria, sendo uma solução de verificação poderosa e versátil.

A fim de configurar o ambiente de desenvolvimento e integrar a ferramenta JasperGold, foi necessário criar um arquivo “.tcl”. TCL (Tool Command Language) é uma linguagem de script amplamente utilizada em ambientes de projetos eletrônicos para automatizar tarefas e configurar ferramentas EDA (Electronic Design Automation). Nesse arquivo “.tcl” são incluídos comandos que a ferramenta JasperGold irá ler e interpretar durante o processo de verificação formal. Entre os comandos essenciais estão “analyze” e “elaborate”.

O comando “analyze” é responsável por ler os arquivos HDL do projeto, verificando se há erros de sintaxe e possíveis violações da linguagem. Já o comando “elaborate” realiza a síntese e constrói a hierarquia do projeto, preparando-o para as etapas subsequentes de verificação.

Esses comandos foram aplicados conforme mostrado no Quadro 3. Como o MiSTer é escrito em três diferentes linguagens HDL, o comando analyze precisa ser executado separadamente para cada linguagem, especificada logo após o comando. Além disso, o projeto contém muitos arquivos distribuídos em diferentes diretórios. Para manter a organização, foi utilizado o argumento “-f”, que instrui a ferramenta a analisar todos os arquivos listados em um arquivo “.f”.

A elaboração do script apresentado no Quadro 3 não foi simples. Os comandos precisam ser executados em uma ordem específica para que a ferramenta construa corretamente a hierarquia, dado que existe uma dependência entre estes módulos. Por exemplo, os módulos “savestates” são unidades fundamentais do projeto escritas em VHDL e são instanciadas em módulos mais complexos, portanto, devem ser processados primeiro. Além disso, a ferramenta requer que o código VHDL seja analisado inicialmente. Assim, foi essencial executar o comando de análise primeiro para os módulos VHDL que representam a CPU do NES (T65) e todos os módulos que são instanciados por ela.

```
1 analyze -vhdl ../rtl/statemanager.vhd
2 analyze -vhdl ../rtl/bus_savestates.vhd
3 analyze -vhdl ../rtl/savestates.vhd
4 analyze -vhdl ../rtl/t65/T65_Pack.vhd
5 analyze -vhdl ../rtl/t65/T65_MCode.vhd
6 analyze -vhdl ../rtl/t65/T65_ALU.vhd
7 analyze -vhdl ../rtl/t65/T65.vhd
8 analyze -sv ../rtl/regs_savestates.sv
9 analyze -sv -f files_sv.f
10 analyze -sv -f files_mappers_sv.f
11
12 elaborate -top sys_top -bbox_m DSP48A1 -bbox_m
    dpram -bbox_m EEPROM_24C0x\
```

Quadro 3. Lista de comandos para portar o projeto para o JasperGold

O comando “elaborate” é utilizado com o argumento -top para indicar qual módulo será o nível mais alto da hierarquia, ou seja, o módulo que instanciará os outros. Este comando é seguido pelo nome do módulo que ocupará essa posição.

Como o MiSTer é desenvolvido usando ferramentas da Intel, especificamente o Quartus 17.0.2, já que o projeto é implementado em um FPGA de mesmo fabricante, esse ambiente de desenvolvimento incorpora módulos exclusivos dessas ferramentas, como circuitos de PLL, memória, e o subsistema HPS (Hard Processor System). Devido a essa dependência de componentes específicos do FPGA que o MiSTer é implementado, o uso da técnica de blackbox nesses módulos específicos é essencial para adequar a plataforma à ferramenta JasperGold. O processo de “blackboxing” é feito usando o argumento -bbox_m e o nome do módulo logo após a declaração do comando elaborate, como mostrado no Quadro 3, que mostra alguns módulos sendo considerados como caixa preta.

O JasperGold possui uma ferramenta conhecida como Superlint, projetada para identificar e categorizar violações na linguagem HDL. O Superlint verifica o projeto HDL em busca de erros, advertências e práticas inadequadas, classificando essas violações em diferentes tipos. Além disso, o Superlint realiza diversas verificações, como a identificação de construções que

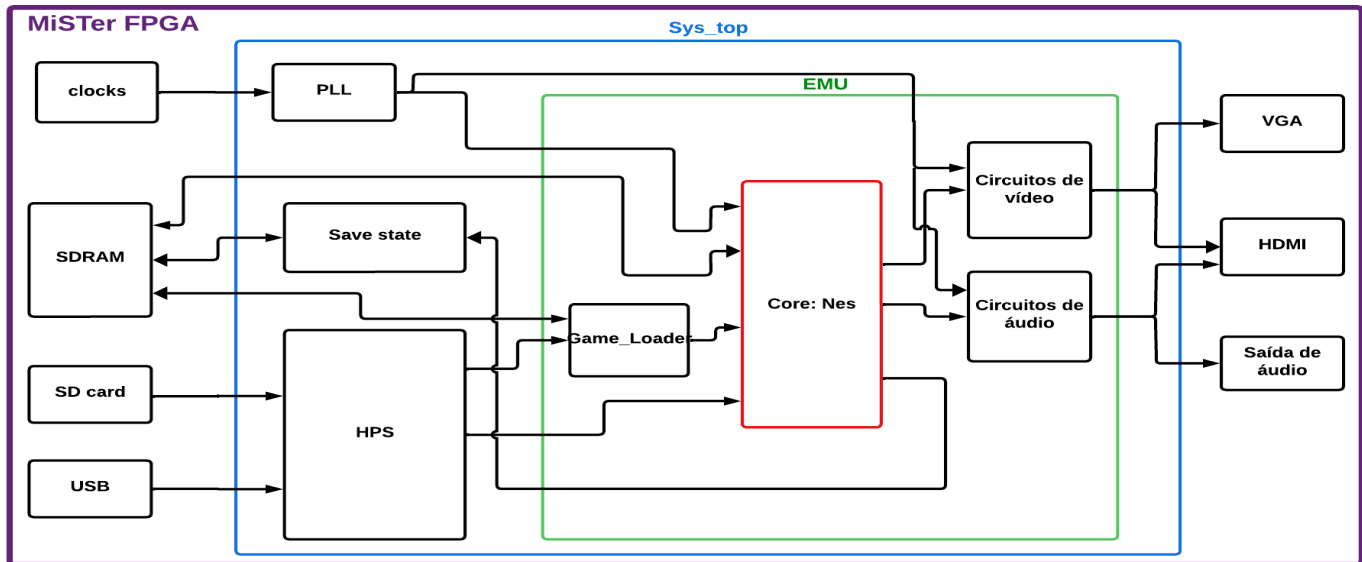


Figura 8. Arquitetura MiSTer representada em diagrama de blocos. Os blocos representam os módulos que dentro dele possuem outros submódulos

não são sintetizáveis - por exemplo, o uso da instrução "initial" para inicializar sinais em código Verilog, que pode resultar em comportamentos imprevisíveis na síntese.

A ferramenta também verifica estruturas de projeto que podem causar problemas funcionais ou em ferramentas posteriores, como Flip-Flops sem resets. Além disso, o Superlint analisa problemas semânticos e funcionais no código, como portas não conectadas, variáveis não utilizadas ou não atribuídas, e sinais não utilizados.

Dessa forma, quando o script .tcl foi executado pela primeira vez, o JasperGold identificou numerosos erros e violações na linguagem, além de uma grande quantidade de warnings. Esses problemas impediam que o Jasper elaborasse o circuito do projeto e prosseguisse com a análise formal. Como resultado, foi necessário corrigir muitos desses erros e warnings para garantir que o projeto estivesse em conformidade e pudesse ser corretamente verificado.

Uma das violações mais comuns encontradas no código foi o uso inadequado de variáveis do tipo "wire" em atribuições procedurais, ou seja, dentro de blocos "always". Em Verilog, variáveis "wire" são destinadas a conexões contínuas e não podem ser usadas para armazenar valores dentro de blocos procedurais, pois não têm a capacidade de reter estados. Para corrigir, as variáveis "wire" foram alteradas para "reg", que é apropriado para atribuições procedurais e permite o armazenamento de valores entre ciclos de clock.

Além disso, um erro grave foi identificado na interface do módulo de memória SDRAM, onde uma porta do tipo "inout" (que pode atuar tanto como entrada quanto como saída) foi incorretamente declarada como reg. Embora o Intel Quartus, utilizado no projeto MiSTer, tenha sido capaz de sintetizar o circuito dessa forma, essa violação impediria que qualquer outra ferramenta, como o JasperGold, realizasse a síntese corretamente.

O objetivo dos desenvolvedores era que a porta inout denominada "SDRAM_DQ" fosse utilizada como entrada quando determinados sinais de controle estivessem em nível lógico baixo, permitindo que os dados de entrada fossem direcionados para um registrador chamado "data_reg". Por outro lado, quando os sinais de controle estivessem em nível lógico alto, a porta "SDRAM_DQ" deveria funcionar como saída, permitindo que os dados armazenados no registrador data fossem transmitidos para fora. A declaração inadequada dessa porta comprometeu a flexibilidade necessária para essa operação.

Para resolver essa situação, foi adotada a estratégia ilustrada na Figura 9. Foi criada uma variável do tipo "reg" chamada Controle, que assume o valor 0 ou 1 dependendo dos sinais de controle. A porta inout "SDRAM_DQ" foi então declarada como "wire". Dessa forma, quando Controle está em nível lógico baixo, "SDRAM_DQ" é usada como entrada para data_reg. Quando Controle está em nível lógico alto, "SDRAM_DQ" transmite os dados armazenados em "data".

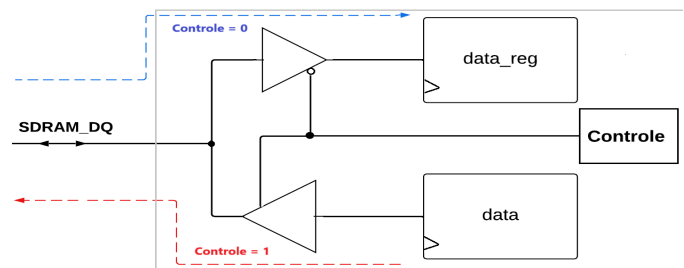


Figura 9. Modificação realizada para resolver o problema da porta "inout" no circuito de SDRAM

Mesmo após corrigir as violações necessárias para permitir que o JasperGold analisasse e elaborasse o projeto, a ferramenta ainda apresentou muitos avisos (warnings) de

violações menos críticas, categorizadas em diferentes tipos durante as etapas de analyze e elaborate. Alguns desses avisos foram resolvidos, como: identificadores estavam sendo usados antes de serem declarados, a presença de valores iniciais que não eram atribuídos posteriormente, e discrepâncias entre o comprimento de bits real e o esperado para uma porta.

Essas modificações reduziram consideravelmente o número de avisos gerados pelo sistema (redução de 331 violações), mas ainda assim alguns warnings persistem. No total, foram adicionadas 1.377 linhas de código e removidas 1.047 linhas em 79 arquivos diferentes. E embora esses avisos remanescentes não impeçam o processo de verificação formal, eles indicam áreas que podem ser aprimoradas para otimizar o projeto e garantir maior conformidade com as melhores práticas de desenvolvimento.

Essas modificações permitiram que o projeto MiSTer fosse elaborado pelo Jasper, criando assim uma estrutura para verificação no mister [5], fazendo com que agora seja possível realizar técnicas de verificação formal no projeto.

C. Validação da Estrutura de Verificação

Com o projeto MiSTer estruturado e formalizado, o próximo passo foi criar propriedades a partir do projeto e verificá-las fazendo o uso do verificador de propriedades do JasperGold. Essa etapa consistiu em analisar os módulos estudados da arquitetura e selecionar alguns sinais específicos, que não fossem tão complexos de serem analisados, para fazer tal verificação.

A primeira propriedade criada foi relacionada aos sinais utilizados nos circuitos de geração de vídeo do MiSTer. Em um desses circuitos o contador horizontal “hcpt” utilizado no processo de varredura de tela tem que funcionar corretamente, incrementando seu valor a cada ciclo de clock quando as condições apropriadas são atendidas. Para verificar esse comportamento, foi criada a propriedade em SystemVerilog mostrada no Quadro 5.

```
1 assume property (@(posedge CLK_VIDEO) (hcpt >= 0 &&
2   hcpt < 4095));
3
4 property hcpt_increment;
5   @(posedge CLK_VIDEO)
6   ((VGA_DE_IN && CE_PIXEL) | => (hcpt == $past(hcpt
7     , 1) + 1));
8 endproperty
9
10 assert property (hcpt_increment);
```

Quadro 4. Propriedade para verificação do incremento do contador “hcpt” do circuito de vídeo

Como o projeto não possui um reset para todos os registradores utilizados, O desenvolvimento dessa propriedade começou com a necessidade de garantir que o valor do contador hcpt estivesse sempre dentro do intervalo esperado na condição inicial (como o hcpt possui 12 bits, o seu valor deve ficar entre 0 e 4095). Isso foi feito usando a diretiva “assume”, que estabelece essa condição como premissa para a verificação.

A propriedade “hcpt_increment” foi então criada para verificar o comportamento de incremento do contador. A lógica implementada verifica que, em cada borda de subida do sinal de clock (CLK_VIDEO), se as condições de controle (VGA_DE_IN e CE_PIXEL) forem verdadeiras, o valor do contador hcpt deve aumentar em 1 em relação ao valor do ciclo anterior. O operador “|=>” é utilizado para estabelecer essa relação causal, enquanto a função \$past recupera o valor anterior do contador para comparar com o valor atual.

Essa propriedade é então assegurada usando a diretiva assert, que instrui a ferramenta de verificação a monitorar o comportamento do contador hcpt a verificação formal, garantindo que essa propriedade é sempre verdadeira.

Outra propriedade criada baseou-se no fato de que quando o processador gráfico do NES precisa de buscar dados de novos sprites para geração de imagem, o barramento de endereços fica com o valor 16’h4014 e sinaliza que a CPU deve ser interrompida a partir da flag mw_int. Assim, no próximo ciclo de clock a CPU deve ser pausada (indicada pelo sinal pause_cpu). Essa propriedade é descrita em SystemVerilog no Quadro 6.

```
1 property dma_cpu;
2   @(posedge clk)
3   disable iff(reset==1)
4   ((addr == 'h4014 && mw_int) && (cpu_ce == 1'b1)
5     | => (pause_cpu == 1));
6 endproperty
7
8 dma_assert: assert property (dma_cpu);
```

Quadro 5. Propriedade que verifica o sinal de pause da CPU quando o processador gráfico precisa receber elementos gráficos

Essa propriedade foi provada pelo Jasper, isto é, esta condição sempre é verdadeira. E o comportamento desses sinais analisados pode ser visto pelo gráfico gerado pelo Jasper na Figura 10.

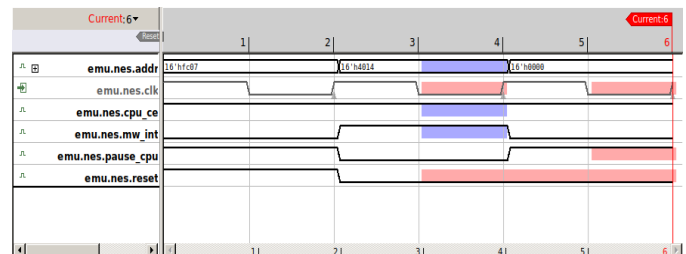


Figura 10. Sinal de pause_cpu indo para alto no próximo ciclo de clock quando as condições são atendidas

Além dessas propriedades, foram incluídos alguns covers ao longo do projeto para a identificação de possíveis falhas. Um desses covers está descrito no Quadro 7.

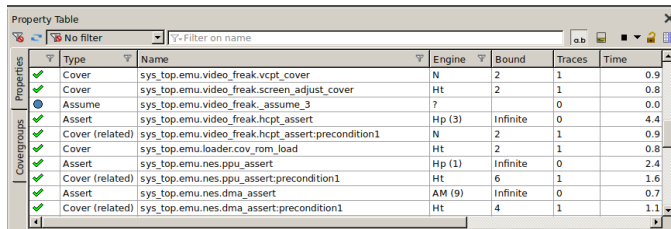
O cover cov_rom_load é utilizado para verificar que, quando determinados sinais como downloading e pelo menos um dos tipos de jogo (type_fds, type_nes, type_nsf) são atendidos, o sinal rom_loaded é ativado, indicando que o jogo foi carregado corretamente no sistema. Essa verificação é importante para garantir que o processo de carregamento de ROMs esteja funcionando conforme o esperado. Além disso, essa abordagem é

extremamente útil para identificar possíveis erros no circuito, especialmente se o sinal rom_loaded nunca for ativado, o que indicaria uma falha no processo de carregamento.

```
cov_rom_load: cover property (@(posedge clk) (
    downloading && (type_fds || type_nes || type_nsf)
)) |-> (rom_loaded == 1));
```

Quadro 6. Cover para Verificação do Carregamento de ROM no Sistema

Essas propriedades foram verificadas com sucesso pelo JasperGold, conforme mostrado na Figura 11. Isso mostra que que a estrutura de verificação criada para o projeto MiSTer funciona, abrindo espaço para criação de outras propriedades para outras validações a respeito do projeto.



Type	Name	Engine	Bound	Traces	Time
Cover	sys_top.emu.video_freak_vcpt_cover	N	2	1	0.9
Cover	sys_top.emu.video_freak_screen_adjust_cover	Ht	2	1	0.8
Assume	sys_top.emu.video_freak_assume_3	?		0	0.0
Assert	sys_top.emu.video_freak_hcpt_assert	Hp (3)	Infinite	0	4.4
Cover (related)	sys_top.emu.video_freak_hcpt_assert precondition1	N	2	1	0.9
Cover	sys_top.emu.loader.cov_rom_load	Ht	2	1	0.8
Assert	sys_top.emu.nes.ppu_assert	Hp (1)	Infinite	0	2.4
Cover (related)	sys_top.emu.nes.ppu_assert precondition1	Ht	6	1	1.6
Assert	sys_top.emu.nes.dma_assert	AM (9)	Infinite	0	0.7
Cover (related)	sys_top.emu.nes.dma_assert precondition1	Ht	4	1	1.1

Figura 11. Propriedades criadas para validação da estrutura sendo verificadas

Além de mostrar o tipo de propriedade verificada (assertion ou cover), bem como o nome dela, a tabela mostrada na Figura 11 possui uma coluna de Engine que indica qual motor da ferramenta foi responsável por mudar o status da propriedade (por exemplo, provada ou refutada) e com qual esforço (um número que representa a quantidade de trabalho ou iterações necessárias). A coluna Bound representa o número mínimo de pulsos de clock necessários para verificar a propriedade dentro do espaço de estados analisado pela ferramenta. O termo "Infinite" significa que a propriedade foi provada para todos os possíveis traços sem encontrar falhas. A coluna trace indica o número de sequência de estados e transições ao longo do tempo que mostra como o circuito evolui de um estado para outro em resposta a eventos, como pulsos de clock. Quando o JasperGold encontra um trace, ele está essencialmente identificando uma sequência de eventos que leva a um determinado resultado — seja uma violação de uma propriedade (no caso de uma assertion) ou a ocorrência de um evento esperado (no caso de uma cover).

V. CONCLUSÃO

A verificação é uma etapa essencial no desenvolvimento de circuitos digitais, desempenhando um papel muito importante na detecção de falhas e na garantia de que o projeto esteja em conformidade com as especificações estabelecidas. Além de assegurar a funcionalidade correta, a verificação também é vital para evitar prejuízos significativos que podem ocorrer devido a falhas não detectadas, como aconteceu em diversos casos históricos na indústria microeletrônica. A importância dessa etapa é evidenciada pelo fato de que muitas empresas dedicam uma parte substancial de seus recursos a processos de verificação, utilizando ferramentas avançadas para minimizar riscos e garantir a confiabilidade dos sistemas.

Neste trabalho, foi adotada uma abordagem de estudo de caso para explorar o uso de ferramentas de verificação amplamente utilizadas na indústria microeletrônica, com ênfase em técnicas de verificação formal, em um projeto grande e complexo como o MiSTer, tal como é na indústria. Nesse contexto, as modificações implementadas no MiSTer não apenas demonstram a aplicação prática dessas ferramentas em um ambiente real, mas também mostram como o projeto pode ser aprimorado para atender aos rigorosos padrões industriais.

As modificações realizadas no projeto MiSTer foram fundamentais para formalizar o projeto, elevando-o a um nível mais profissional. Agora, o projeto pode ser utilizado com ferramentas de verificação formal avançadas, como o JasperGold, o que amplia significativamente sua confiabilidade e aplicabilidade. Além disso, essas mudanças permitem que o projeto seja portado para outras plataformas, além da FPGA Cyclone V, aumentando sua flexibilidade e potencial de uso em diferentes contextos de hardware.

A estrutura de verificação criada foi devidamente validada, o que assegura que outros projetistas possam realizar otimizações e modificações no projeto sem comprometer sua funcionalidade. Essa infraestrutura viabiliza uma refatoração completa do projeto, garantindo que, mesmo após mudanças significativas, o projeto permanecerá logicamente consistente e funcionalmente equivalente, graças ao suporte das ferramentas de verificação formal.

REFERÊNCIAS

- [1] Geek 360. *Melhores Processadores*. Acessado: 02 de setembro de 2024. URL: <https://geek360.net/melhores-processadores/>.
- [2] Jerry R Burch et al. "Symbolic model checking: 1020 states and beyond". Em: *Information and computation* 98.2 (1992), pp. 142–170.
- [3] Cadence. *Digital IC Design Fundamentals*. Accessed: 2024-08-16. 2024. URL: https://www.cadence.com/en_US/home/training/all-courses/86305.html.
- [4] Matt Hanson. *Pentium FDIV: The processor bug that shook the world*. Accessed: 2024-09-02. 2021. URL: <https://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773>.
- [5] Victor Hugo. *NES_MiSTerCadence*. https://github.com/victorhug97/NES_MiSTerCadence. Acessado em setembro de 2024. 2024.
- [6] Intel. *14th Gen Intel® Core™ Desktop Processors Brief*. Acessado em 02 de setembro de 2024. 2024. URL: <https://www.intel.com.br/content/www/br/pt/products/docs/processors/core/core-14th-gen-desktop-brief.html>.
- [7] MiSTer Development Team. *MiSTer Wiki*. Acessado em setembro de 2024. 2024. URL: https://mister-devel.github.io/MkDocs_MiSTer/.
- [8] Erik Seligman, Tom Schubert Schubert e M V Achutha Kiran Kuma. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. Morgan Kaufmann, 2015.