UNIVERSIDADE FEDERAL DE ITAJUBÁ - *CAMPUS* ITAJUBÁ

DETECÇÃO DE TROJANS DE HARDWARE EM CIRCUITOS COMBINACIONAIS NCL ATRAVÉS DO MÉTODO DE TRANSIÇÕES PROBABILÍSTICAS

JOÃO PEDRO PEREIRA MAGALHÃES

Itajubá - MG

27 de FEVEREIRO de 2026

UNIVERSIDADE FEDERAL DE ITAJUBÁ - *CAMPUS* ITAJUBÁ

JOÃO PEDRO PEREIRA MAGALHÃES

DETECÇÃO DE TROJANS DE HARDWARE EM CIRCUITOS COMBINACIONAIS NCL ATRAVÉS DO MÉTODO DE TRANSIÇÕES PROBABILÍSTICAS

Dissertação submetida à Universidade Federal de Itajubá - *Campus* Itajubá como parte dos requisitos para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Prof. Doutor Tales Cleber Pimenta
Coorientador: Prof. Doutor Diogo Ferreira Silva

Itajubá - MG

27 de FEVEREIRO de 2026

Agradecimentos

Gostaria de expressar meus profundos agradecimentos aos meus familiares que acreditaram, incentivaram e sempre foram o ponto de apoio durante toda a caminhada. Especialmente à minha mãe, ao pai e aos meus avôs, que me ajudaram a mudar em um mês e fazer com que o meu sonho se tornasse real.

Aos meus amigos que, nos momentos difíceis, estiveram me dando o apoio que precisava, oferecendo distrações e servindo como bons ouvintes e conselheiros. Especialmente aos meus amigos Vinícius Tadeu e Tchola.

Com o mesmo grau de importância, a todos os professores do Grupo de Microeletrônica da universidade por todo o suporte, pelos ensinamentos e por acreditarem na minha capacidade. Em especial ao meu orientador Tales, que aceitou e incentivou a jornada maluca que foi o desenvolvimento deste trabalho; ao meu coorientador Diogo, por acreditar na minha competência, ser um mentor e tornar o sonho do meu mestrado real; e aos professores Fanelli e Zoccal, por toda a consideração e ensinamentos ao longo dessa trajetória.

Por fim, agradeço a UNIFEI e a FAPEMIG por acreditarem no meu projeto, em minhas competências e fornecer todo o suporte estrutural e financeiro para que o projeto fosse realizado.

Sem a ajuda e a presença de cada um de vocês, nada disso seria possível. Por isso, a minha eterna e sincera gratidão a todos vocês.

"Eu via ao meu redor pessoas aflitas que para se salvarem esperavam apenas uma mão que as apoiasse, nada mais que isto. E Deus me dera duas mãos!"

(Érico Veríssimo)

Resumo

O interesse em circuitos digitais *Quasi Delay Insensitive*, em especial os circuitos *Null Convention Logic*, parte da não necessidade de um sinal de *clock* global para o controle das partes do circuito. Isso resulta em circuitos mais rápidos, energeticamente eficientes e mais robustos. Com o aumento na demanda desse tipo de circuito na indústria, modificações indesejadas advindas de pessoas mal-intencionadas são um fator de preocupação, considerando que o processo de produção em larga escala desses circuitos é dividido entre diversas empresas e colaboradores, o que dificulta o conhecimento de todo o seu processo produtivo. A partir desse contexto, este trabalho tem como objetivo o estudo e a análise do método de transições probabilísticas aplicado a circuitos *Null Convention Logic* combinacionais. O algoritmo recebe as informações sobre as interconexões do circuito original e do modificado e compara, a partir dos valores de cada elemento lógico, se houve ou não modificações entre os circuitos utilizando os casos de valores de entrada válidos ("0" e "10"). A partir das discussões contidas neste trabalho, têm-se os seguintes resultados: a descrição e simulações funcionais dos circuitos de teste; a implementação de um transpilador para conversão dos arquivos de saída do *software* Quartus para o padrão de *netlist* de entrada do algoritmo; e o desenvolvimento do próprio algoritmo de transições probabilísticas. Este último apresenta dois modos de operação: ao receber uma única *netlist*, calcula os valores de transição para todos os elementos, gerando uma assinatura exclusiva para o circuito; ao receber duas *netlists*, o programa realiza os cálculos individuais e executa uma comparação entre eles, informando eventuais divergências e seus respectivos locais de ocorrência.

Palavras-chaves: circuitos digitais. transição probabilística. null convention logic. trojans de hardware.

Abstract

The interest in Quasi Delay Insensitive digital circuits, especially Null Convention Logic circuits, stems from the absence of a need for a global clock signal to control the circuit's parts. This results in faster, energy-efficient, and more robust circuits. With the increasing demand for this type of circuit in the industry, undesirable modifications from malicious actors are a concern, given that the large-scale production process for these circuits is divided among various companies and collaborators, which makes it difficult to have complete knowledge of its entire production process. Given this context, this work aims to study and analyze the Probabilistic Transitions method applied to combinational Null Convention Logic circuits. The algorithm receives information about the interconnections of the original and modified circuits and compares, based on the values of each logic element, whether or not modifications occurred between the circuits, using the valid input value cases ("01" and "10"). Based on the discussions contained in this work, the following results are presented: functional descriptions and simulations of the test circuits; the implementation of a transpiler to convert Quartus software output files into the algorithm's required *netlist* format; and the development of the probabilistic transition algorithm itself. The algorithm operates in two modes: when provided with a single *netlist*, it calculates transition values for all elements to generate a unique circuit signature; when provided with two *netlists*, the program performs individual calculations and executes a comparative analysis, reporting potential divergences and their specific locations.

Keywords: digital circuits. transition probabilistic. null convention logic. hardware trojan.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IRDS	*International Roadmap for Devices and Systems*
ITRS	*International Technology Roadmap for Semiconductors*
MUX	Multiplexador
NCL	*Null Convention Logic*
QDI	*Quasi Delay Insensitive*
RTL	*Register Transfer Level*
TH	*Trojans de Hardware*
ULA	Unidade Lógica Aritmética
VLSI	*Very Large Scale Integration*

Códigos

*

Sumário

1 Introdução

Os *Trojans* de *Hardware* (TH) são modificações indesejadas inseridas em circuitos digitais com o intuito de afetar sua funcionalidade, seja vazando informações, alterando o funcionamento original do circuito ou inibindo por completo seu funcionamento (KARRI et al., 2010; CHAKRABORTY; NARASIMHAN; BHUNIA, 2009). No trabalho desenvolvido por(KARRI et al., 2010) é destacado a presença de roteadores falsos da empresa Cisco em redes de defesa, finanças e universidades nos Estados Unidos. Os roteadores adquiridos eram mal construídos, o que impactava em altas taxas de falha nos locais onde eram aplicados. Durante o processo investigativo do *Federal Bureau of Investigation* (FBI), foi detectado que os produtos eram adquiridos pelas empresas do governo de fontes não confiáveis, como revendedores não autorizados, leilões e empresas terceirizadas em outros países. Já (MITRA; WONG; WONG, 2015) destaca que, em 2007, um radar sírio não informou um ataque aéreo iminente. Dentre as causas apontadas para esse problema, estava a inserção de circuitos adicionais que alteravam o funcionamento do *chip*. Já (ADEE, 2008) destaca que, em conversa com um informante do Ministério da Defesa dos Estados Unidos da América (EUA), um fabricante europeu de *chips* desenvolveu microprocessadores que podiam ser desativados de maneira remota. Esses *chips*, ainda de acordo com o informante, foram aplicados em equipamentos militares franceses para que, caso esses circuitos caíssem na mão de pessoas indevidas, o exército francês fosse capaz de desativar esses circuitos. (PONUGOTI et al., 2022) apresenta o uso de TH durante a Guerra do Golfo em 1991, em que os *chips* utilizados em impressoras foram modificados pelos militares dos Estados Unidos para cópia e armazenamento de informações, sendo esse um dos primeiros casos de aplicação de *Trojans* de *Hardware*.

Anteriormente, a menor quantidade de *chips* a serem produzidos possibilitava que as próprias empresas os fabricassem. Porém, os altos custos para montagem de fábricas de ponta capazes de suprir a demanda e o custo elevado em pesquisa e desenvolvimento levaram à concentração dos processos produtivos em um nicho de países como a China, Coreia do Sul, Taiwan e Estados Unidos. Apesar do conhecimento sobre quais países fabricam os circuitos, não é possível garantir que modificações incluindo *hardwares* maliciosos, visando interesses pessoais, não estejam presentes em um lote inteiro ou em partes deles (MITRA; WONG; WONG, 2015).

1.1 Motivação e Justificativa

A escalabilidade dos circuitos digitais provoca a necessidade de circuitos cada vez mais otimizados, que, por sua vez, apresentam dificuldades em relação ao paradigma de

circuitos digitais síncronos. Manter um sinal de sincronismo comum a todo o circuito, em termos físicos, impacta a área e o consumo, visto que uma árvore de *clock* é utilizada para controlar o atraso do sinal de *clock* entre registradores do circuito; além disso, *buffers* são aplicados para manter a integridade do sinal. (OLIVEIRA et al., 2018) destaca que, em circuitos *Very Large Scale Integration* (VLSI), a sincronização global dos elementos lógicos representa uma tarefa custosa e complexa, sendo um gargalo de projeto.

Dado esse contexto, os circuitos digitais *Quasi Delay Insensitive* (QDI) destacam-se como uma alternativa aos circuitos tradicionais, haja vista suas características de robustez, eficiência energética, velocidade, etc. (FANT; BRANDT, 1996; KHODOSEVYCH; SAKIB, 2022). Além disso, a comunicação entre os blocos do circuito não é mais controlada por um *clock* global, sendo realizada pelo protocolo *handshake* local. Dentre as classes de circuitos QDI, a mais comum e a abordada neste trabalho é a *Null Convention Logic* (NCL).

O aumento no interesse pelos circuitos assíncronos é corroborado em 2013 pelo *International Technology Roadmap for Semiconductors* (ITRS) 2013, que previu a adoção da tecnologia assíncrona por 54% da indústria de semicondutores até 2027. De forma análoga, o *International Roadmap for Devices and Systems* (IRDS) (THE. . . , 2019) de 2018 destacou os circuitos assíncronos como uma das principais soluções para a redução do consumo de energia.

O crescente interesse no uso de circuitos NCL destaca a preocupação com a segurança e a confiabilidade. Contudo, durante o ciclo de vida do projeto, especialmente na fabricação, agentes mal-intencionados podem inserir modificações indesejadas, comprometendo a funcionalidade desses circuitos.

Portanto, o desenvolvimento de métodos para a identificação de Trojans de Hardware em circuitos NCL faz-se necessário. Abordagens comerciais baseads em vetores de teste são amplamente utilizadas. Contudo, a estrutura particular dos circuitos NCL, que operam em lógica Dual-Rail (exigindo dois sinais para representar um bit), introduz desafios significativos. O número dobrado de sinais em relação aos circuitos booleanos aumentam a complexidade da análise para esses algoritmos tradicionais tendo em vista que o número de possibilidades de entrada serão maiores, assim exigindo mais tempo computacional para resultados satisfatórios.

1.2 Definição do Problema e Abordagem Proposta

A realização do levantamento bibliográfico durante este trabalho apontou poucos estudos sobre métodos de identificação de TH em circuitos NCL. Nota-se a ausência de abordagens voltadas para a etapa pré-silício que sejam focadas na análise de dados válidos (valores de entradas iguais a “01” e “10”) mantendo, simultaneamente, um bom grau de

cobertura do circuito e um tempo de análise viável.

Analisando a literatura, verifica-se que autores como (GUIMARAES et al., 2018; GUAZZELLI et al., 2020; LODHI et al., 2014) propõem a identificação de *Trojans* de *Hardware* através da análise por canal lateral. Contudo, tais abordagens são aplicadas apenas na fase pós-silício, ou seja, após a fabricação do *chip*. Por outro lado, a proposta de (PONUGOTI et al., 2022) apresenta um método de análise pré-silício, mas restringe seu escopo aos casos de entrada inválida (estado "11"). Diante disso, este trabalho busca contribuir com uma forma de identificação focada especificamente em circuitos NCL combinacionais.

Nesse cenário, o objetivo deste trabalho consiste em analisar a viabilidade da aplicação do método de transições probabilísticas para a detecção de modificações maliciosas nos circuitos-alvo. A intenção é que a ferramenta desenvolvida apresente duas funcionalidades: (i) atue como um gerador de assinaturas para a saída de um circuito, as quais posteriormente podem ser comparadas com outras, e atue como um comparador, sendo capaz de identificar discrepâncias entre o circuito original e (ii) o circuito sob teste, apontando quais alterações foram realizadas e a localização dessas mudanças.

Portanto, a metodologia adotada baseia-se no desenvolvimento de um programa em linguagem C++. Este *software* utiliza o algoritmo proposto por (SALMANI; TEHRANIPOOR; PLUSQUELLIC, 2012) e (POPAT; MEHTA, 2016), modificado para analisar circuitos em lógica NCL e otimizar a apresentação dos resultados. O fluxo geral proposto é ilustrado na Figura 1, na qual as *netlists* são inicialmente processadas para a execução dos cálculos elemento a elemento. Por fim, realizam-se comparações para identificar possíveis divergências entre as descrições. Como exemplificado na Figura 1, o terceiro componente é alterado de uma porta AND no circuito original para uma XOR na versão modificada; essa mudança impacta os valores da tabela de verdade e propaga-se até as saídas, destacando a modificação do circuito. Para a etapa de descrição, utilizou-se a ferramenta Intel® Quartus® Prime Lite Edition (versão 23.1 std). Posteriormente, a simulação funcional é executada no simulador Questa - Intel FPGA Starter Edition 2023.3.

Uma vez validados os circuitos, as informações sobre as interconexões dos circuitos são extraídas no formato de *netlist*. Esses arquivos servem de entrada para o algoritmo de detecção desenvolvido, que processa os dados e fornece o veredito sobre a integridade do circuito. Os detalhes técnicos sobre a escolha da linguagem, a implementação dos circuitos, a inserção dos THs e a codificação do programa de detecção são aprofundados no Capítulo 3.

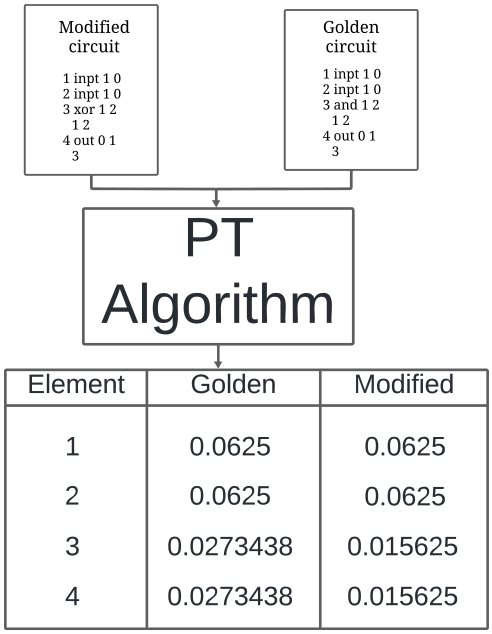


Figura 1 – Visão Macro do Funcionamento do Programa Desenvolvido.

1.3 Organização do Trabalho

O presente trabalho está organizado em 5 Capítulos. O Capítulo 1 fornece uma breve contextualização do trabalho, descreve a motivação e a justificativa, determina o problema a ser resolvido e os objetivos estabelecidos. Ainda faz uma breve explicação sobre a metodologia e discorre sobre a organização do texto.

O Capítulo 2 apresenta uma revisão teórica dos principais conhecimentos agregados no desenvolvimento do algoritmo e desenvolvimento dos circuitos, além de apresentar o estado da arte dos métodos de detecção.

O Capítulo 3 expõe os métodos de implementação dos circuitos a serem analisados, bem como os *trojans* que serão aplicados a eles. Descreve, ainda, em detalhes o método de implementação do algoritmo de transições probabilísticas, as funções principais do algoritmo, bem como a forma de exposição dos dados.

O Capítulo 4 traz os resultados obtidos com a implementação do algoritmo a partir da análise dos circuitos de teste, assim como a comparação com um método de verificação formal via *testbench*.

O Capítulo 5 apresenta as conclusões sobre o algoritmo a partir da perspectiva dos resultados obtidos e do desenvolvimento dele. Considerações para trabalhos futuros também são apresentadas e discutidas neste capítulo.

2 Referencial Teórico

2.1 Circuitos *Quasi Delay-Insensitive*

Atualmente, a indústria de circuitos digitais utiliza como padrão de projeto os circuitos síncronos, nos quais um sinal de controle global, o *clock*, é responsável por ditar o instante de atualização dos registradores em todo o circuito. Analisando o domínio do tempo, os circuitos síncronos operam com base em um modelo de atraso limitado (*bounded-delay*), onde diversas regras de tempo são consideradas. Dentre elas, a mais comum é a de que o período do *clock* deve ser maior que o atraso do caminho crítico do circuito, garantindo assim a transmissão de dados estáveis entre os blocos lógicos. Essa abordagem resulta em um desempenho atrelado ao pior caso de atraso.

No paradigma QDI, as restrições de tempo são praticamente eliminadas, pois assume-se que os atrasos em portas e fios são ilimitados. Para tornar esse modelo viável, utiliza-se a premissa de *isochronic wire forks*. Essa regra se aplica apenas dentro de componentes básicos (como somadores), onde o atraso dos fios é considerado muito menor que o das portas lógicas. Nas conexões entre esses componentes, os atrasos continuam sendo tratados como desconhecidos.

Essa premissa é necessária porque criar circuitos puramente insensíveis a atrasos inviabilizaria o projeto (WOJCICKI, 2014). Isso ocorre por dois motivos: primeiro, sem o *isochronic fork*, cada bifurcação de fio precisaria de seu próprio controle de fluxo (*handshaking*), o que aumentaria muito o custo de *hardware*. Segundo, o número de funções lógicas possíveis seria muito pequeno, o que impediria a construção de sistemas complexos (KHODOSEVYCH; SAKIB, 2022; SMITH; DI, 2009).

Dessa forma, para que o sistema funcione sem um *clock* global, o protocolo de *handshake* é usado em conjunto com a lógica *dual-rail* e a detecção de conclusão para informar de maneira confiável aos blocos seguintes quando um dado terminou de ser processado e está pronto para ser utilizado (WOJCICKI, 2014).

2.1.1 Lógica Dual-Rail

A lógica *dual-rail* caracteriza-se pelo uso de dois sinais de entrada A_0 e A_1 para a representação dos níveis lógicos Data0, Data1 e *NULL* (SMITH; DI, 2009; WOJCICKI, 2014). A Tabela 1 apresenta a relação entre os três níveis lógicos e sua representação em binário. O nível lógico *NULL* configura o momento em que o dado ainda não está pronto. Esse estado é importante para que haja a distinção de quando um dado pode ser transmitido ou não via o protocolo de *handshake*, e assim seja realizado corretamente.

Haja vista que se um estágio do circuito encontra-se em estado *NULL*, significa que este está em *stand by* e pronto para receber um novo dado. O nível lógico Data0 corresponde ao nível lógico 0 (ou baixo) na lógica booleana e o nível lógico Data1 é correspondente ao nível lógico 1 (alto) da lógica booleana. A lógica *dual-rail* é mutualmente exclusiva, ou seja, os sinais de entrada nunca apresentarão '1' simultaneamente. Esse é um estado de invalidez para circuitos QDI e não ocorre durante o funcionamento normal dos circuitos.

Tabela 1 – Representações em *dual-rail*.

Sinal	*NULL*	DATA0	DATA1	Inválido
A_0	0	1	0	1
A_1	0	0	1	1

2.2 Circuitos NCL

Propostos por (FANT; BRANDT, 1996), os circuitos NCL são uma subclasse de QDI que utilizam estados nulos e sinais de *handshaking* para validar dados de entrada dos registradores. O funcionamento desses circuitos baseia-se nas células M de N (identificadas como THmn), onde n é o número total de entradas e m indica quantas dessas entradas precisam ser '1' para que a saída seja '1'. A Figura 2 mostra a representação genérica de uma célula M de N.

Já a Figura 3 exibe um caso específico: a célula TH24w22. Ela possui entradas com pesos diferentes, indicados após a letra w. Neste exemplo, as entradas A e B têm peso 2, enquanto C e D têm peso 1. Portanto, a saída será ativada se A=1, ou B=1, ou se C e D forem ambos '1'. A lista com as 27 células básicas e suas expressões booleanas pode ser vista em (SMITH; DI, 2009).

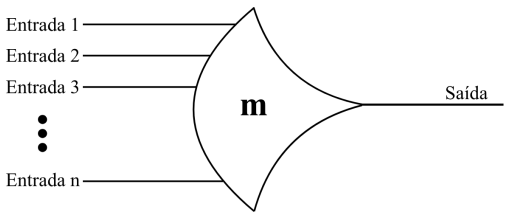


Figura 2 – Célula M de N genérica baseada em (SMITH; DI, 2009).

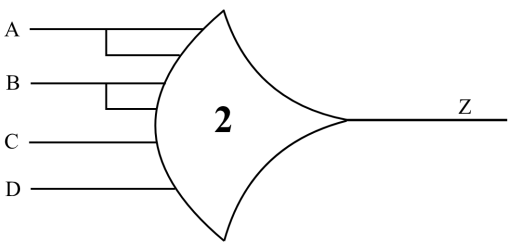


Figura 3 – Célula TH24w22: $Z = A + B + CD$.

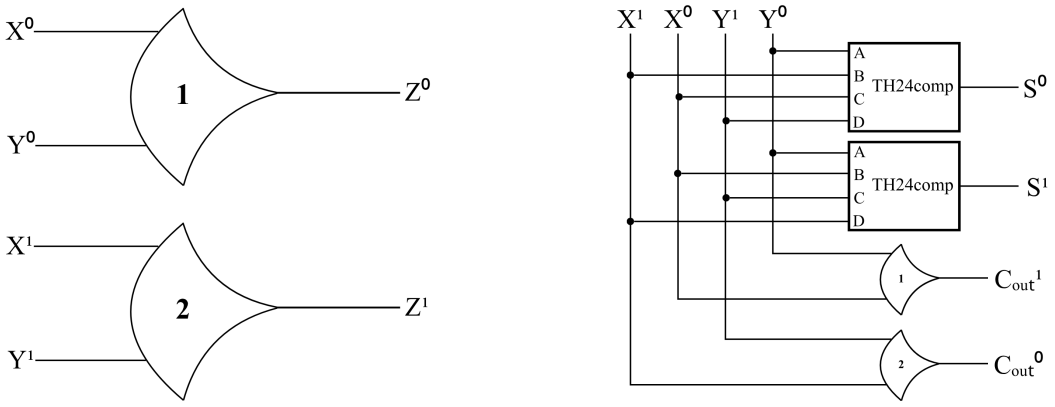
Dada a característica de completude de entrada destacada na Seção 2.2.1, as células NCL são projetadas para retenção de estado usando a condição de histerese. Dessa forma, a saída da célula só manterá o estado '1' até que todas as entradas sejam alteradas para o valor igual a '0'. Dessa forma, as células M de N apresentam um caráter de memória atrelado a elas, que garante a transição completa das entradas para o estado *NULL* antes que um próximo dado válido seja enviado para a próxima etapa.

2.2.1 Completude de Entrada e Observabilidade

As características de um circuito QDI, aplicadas aos circuitos NCL, baseiam-se nas propriedades de Completude de Entrada e Observabilidade (SMITH; DI, 2009). A completude de entrada define que as saídas de um circuito não podem mudar de *NULL* para um dado válido (*DATA*) até que todas as entradas tenham mudado de *NULL* para *DATA*. Da mesma forma, as saídas não podem voltar para *NULL* antes que todas as entradas tenham retornado para *NULL*.

A Figura 4a mostra um exemplo onde essa regra é violada, usando portas com asserções diferentes: uma porta TH12 e uma porta TH22. Nesse caso, a saída Z_0 é controlada pela porta TH12, que ativa a saída se apenas uma das entradas (X_0 ou Y_0) for '1'. Por exemplo, se a entrada X mudar para Data0 (ou seja, X_0 vira '1') enquanto a entrada Y ainda estiver em *NULL*, a saída Z_0 muda para '1' imediatamente. Como a saída mudou sem a presença de todas as entradas, esse circuito é considerado não completo de entrada.

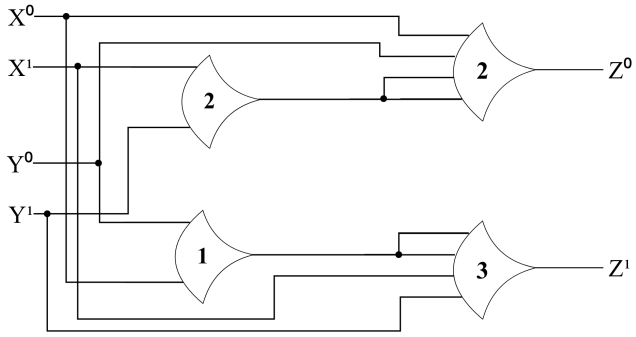
Por outro lado, a Figura 4b apresenta um Meio-Somador NCL. A lógica desse circuito é feita de modo que a saída de Soma (S) só muda para um valor válido (S_0 ou S_1) se ambas as entradas estiverem presentes. Isso garante que o conjunto total de saídas só seja validado quando todas as entradas chegarem, respeitando a completude de entrada e garantindo a insensibilidade a atrasos.



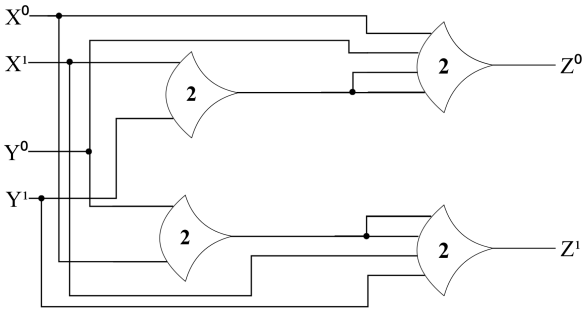
(a) (b)

Figura 4 – Circuito NCL AND (a) Incompletude de entrada e (b) Meio somador com completude de entrada. Reconstrução baseada em (SMITH; DI, 2009).

Por outro lado, a observabilidade é definida como: todo *gate* do circuito que sofre uma transição deve impactar a transição de pelo menos uma saída do circuito. A Figura 5a apresenta um circuito NCL com funcionalidade de uma XOR não observável, dado que para X e Y iguais a Data0, a saída $Z = Data0$ não ocorre. A célula 1 de 2 não é suficiente para alterar o estado da saída Z_1, haja vista que pelo menos 3 entradas devem ser '1' e a saída da célula 1 de 2 tem apenas o peso 2. Já na Figura 5b, para o caso de $X = Y = Data0$, a transição em qualquer uma das células M de N impacta na alteração das saídas Z_0 e Z_1.



(a)



(b)

Figura 5 – Circuito NCL XOR (a) Não observável e (b) Observável. Reconstrução baseada em (SMITH; DI, 2009).

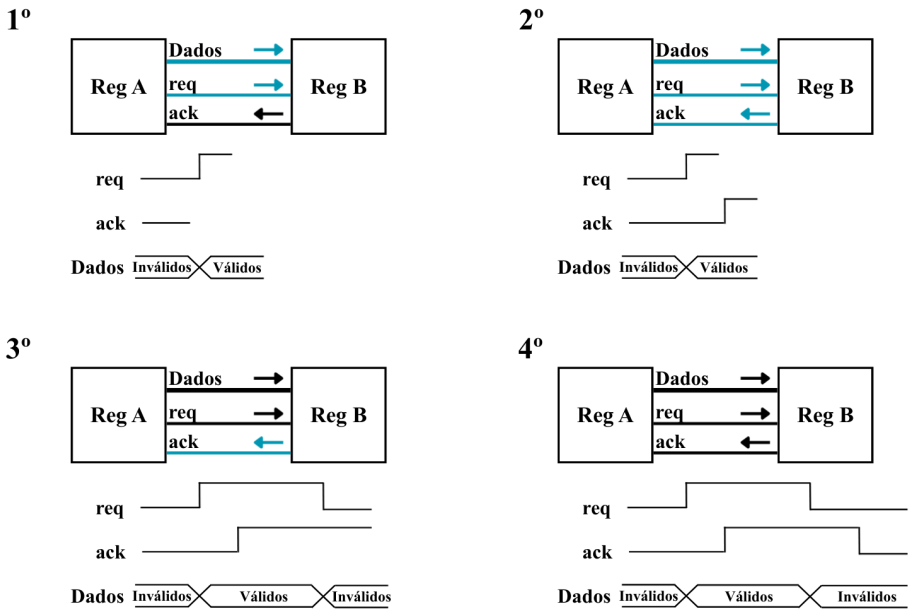
2.2.2 Protocolo *Handshaking*

O protocolo *handshaking* é um método de comunicação entre circuitos digitais NCL em que são utilizados sinais de *request* (req) e confirmação *acknowledge* (ack) que os dados foram enviados. Esse protocolo apresenta dois tipos de implementação: 2 fases e 4 fases.

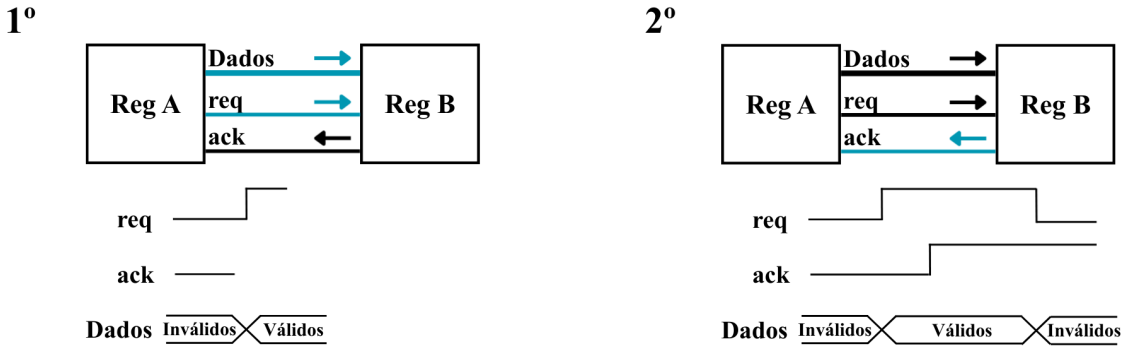
A Figura 6a apresenta a simulação do protocolo de 4 fases. Na primeira fase, o bloco que envia os dados fornece os dados válidos no barramento de comunicação e define o sinal de *request* como nível lógico alto. Já na segunda fase, o bloco receptor armazena as informações recebidas e define o sinal de *acknowledge* como '1'. Na terceira fase, essa

condição ao ser interpretada pelo bloco de envio define o sinal de *request* como nível lógico baixo, indicando o término do envio. Por fim, na quarta fase, o bloco que recebe os dados define o sinal de ack como nível lógico baixo para que uma nova comunicação seja realizada.

Já a Figura 6b destaca o protocolo de duas fases. Para a primeira fase o remetente adiciona os dados no barramento dados e define o sinal de req. Ao receber essa informação, na segunda fase, o bloco receptor altera o sinal de ack indicando que recebeu o dado enviado.



(a)



(b)

Figura 6 – *Handshake* de (a) 4 fases e (b) 2 fases.

2.2.3 Topologias de *Pipeline* Assíncrono para Circuitos NCL

A Figura 7a apresenta a descrição de um registrador NCL, onde I_0 e I_1 correspodem as entradas, O_0 e O_1 às saídas, K_o indica se o dado armazenado no registrador é válido e

K_i funciona como *enable* do registrador.

A Figura 7b ilustra a estrutura de um circuito NCL genérico em *pipeline*. Inicialmente, o dado é inserido no fluxo e armazenado no primeiro estágio de memória. Subsequentemente, a informação é processada pela lógica combinacional e propagada para a etapa seguinte. Devido à natureza assíncrona do projeto, o fluxo de dados é gerenciado por um circuito de detecção de conclusão. Este bloco é composto por células M de N do tipo $M = N$ em que $2 \leq M \geq 4$ (SMITH; DI, 2009). Esses elementos tem como entrada os sinais K_o dos registradores do estágio posterior e são encadeados até que seja gerada uma única saída.

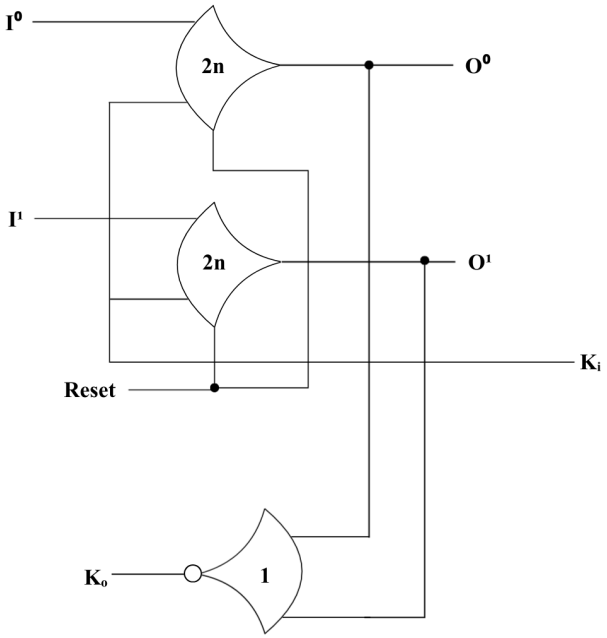
O sincronismo e o controle desse fluxo baseiam-se diretamente nos níveis lógicos de K_o. Um nível alto indica que o próximo estágio encontra-se no estado *NULL*, apto a receber novas informações. Após a recepção (transição para DATA0 ou DATA1), o sinal K_o altera-se para o nível baixo. Ao identificar essa mudança, o detector de conclusão inibe o envio de novos dados pelo estágio anterior, forçando o retorno dos componentes ao estado *NULL*. O ciclo é reiniciado assim que os registradores esvaziam, permitindo uma nova operação. Todo esse processo de comunicação é regido pelo protocolo de *handshake* de duas fases.

2.2.4 Descrição dos Circuitos Combinacionais

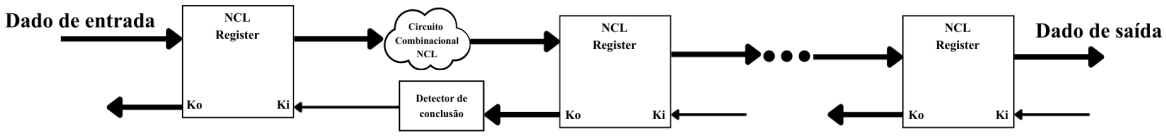
Nesta subseção são apresentados os métodos utilizados para a descrição das células M de N, bem como dos operandos lógicos NCL.

2.2.4.1 Descrição das Células M de N via Método de Huffman

Como discutido na Seção 2.2, os circuitos NCL são compostos por células M de N. Dadas as equações contidas em (SMITH; DI, 2009), a descrição de cada uma das células é realizada. Contudo, o processo de histerese não está atrelado à equação booleana fornecida, sendo necessário um método de garantir o comportamento no momento de descrição das células M de N. (OLIVEIRA et al., 2018) apresenta em seu trabalho o método próprio baseado na extração das funções de *set* e *reset* que, conectado a um *latch* de saída, infere o nível lógico alto ou baixo da célula. Além disso, outros três métodos são abordados ao longo do texto: Huffman (MYERS, 2001), *standard* RS (PARSAN; SMITH, 2012) e *modified* RS (OLIVEIRA et al., 2017). Dentre as possibilidades de descrição da célula M de N, o método de Huffman foi o escolhido dada a familiaridade com o método. A condição de *set* corresponde à equação booleana advinda das equações de cada uma das células, já a condição de histerese se dá por uma OR entre todas as entradas, seguida de uma AND com a saída atual (Out). As duas condições são unificadas em uma OR, gerando a saída $Out(t+1)$ da célula. Como discutido na Seção 2.2, os elementos NCL são inicializados no estado *NULL*, e, enquanto a condição de set não ocorre, estes permanecem inalterados. A



(a)



(b)

Figura 7 – a) Registrador NCL b) Circuito *pipeline* NCL. Reconstrução baseada em (SMITH; DI, 2009).

partir do momento em que a saída é afirmada, o circuito só volta para a saída igual a '0' quando todas as entradas também retornarem a '0'. Todas as células M de N empregadas na modelagem dos operadores lógicos dos circuitos de teste e dos *trojans* inseridos foram desenvolvidas pelo autor deste trabalho e encontram-se disponíveis no mesmo repositório dos projetos.

2.2.4.2 Operadores Lógicos NCL

Dada a natureza *dual-rail* dos circuitos NCL, os operadores lógicos básicos, como por exemplo OR, AND, XOR e NOT. em NCL também constituem de entradas e saídas em *dual-rail*. Por conta dessa característica, apesar de algumas células fornecerem a mesma condição lógica, ela só define parte da saída geral do operando. Por exemplo, o operando AND que, dadas as entradas A e B, tem como expressão lógica a Equação (2.1).

$$Out = A \cdot B \tag{2.1}$$

Dentre as opções de células disponíveis, a TH22 apresenta o mesmo comportamento da porta lógica *AND* booleana. No contexto da lógica NCL, contudo, esse comportamento representa a afirmação de apenas um dos sinais que compõem a saída em *dual-rail*. Dessa forma, é necessária uma segunda célula para controlar o sinal complementar do par. Neste caso, a célula THand0 (representada como THAnd na Figura 8) exerce essa função. Assim, a Figura 8 ilustra a estrutura da porta *AND* em NCL. Os subíndices `t` e `f` indicam a significância do bit em *dual-rail*, sendo `t` o mais significativo. Portanto, a modelagem prévia das portas lógicas básicas é a base para a subsequente descrição dos circuitos combinacionais bem como dos *trojans* aplicados nesse trabalho.

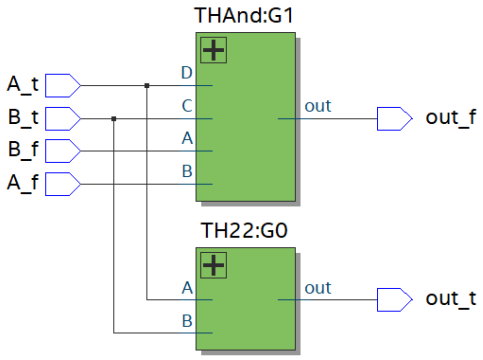


Figura 8 – Porta lógica AND NCL.

2.2.4.3 Uso de Operadores Lógicos NCL para Descrição de Circuitos Combinacionais

Como aponta (OLIVEIRA et al., 2018), os circuitos combinacionais NCL podem ser descritos a partir da função de minimização de circuitos booleanos. Esse tipo de abordagem facilita o processo de descrição de circuitos NCL, dada a familiaridade do projetista com a manipulação e compreensão dessas expressões. Como exemplo, seja a equação:

$$F = (A \mid B) \cdot (C \oplus B) \tag{2.2}$$

A descrição em NCL desse circuito se dá pela conversão das entradas em dual rail e o uso dos operandos lógicos NCL. Dessa forma, o circuito correspondente é apresentado na Figura 9. A partir desse método, todos os circuitos e os trojans utilizados neste trabalho serão descritos. Esse tipo de descrição também é vantajosa, dado que o método analisado nesse trabalho realiza a verificação baseado no tipo de porta lógica. Como os operandos lógicos em NCL manterão as mesmas características dos operandos booleanos, é possível utilizar o método para o processo de verificação de circuitos combinacionais NCL sem que haja alterações nas equações definidas na Seção 2.6.

2.3 *Trojans* de *Hardware*

Trojans de *Hardware* são modificações maliciosas em circuitos digitais e analógicos com intuito de (KARRI et al., 2010; CHAKRABORTY; NARASIMHAN; BHUNIA, 2009):

- Vazar informações;
- Inutilizar parcialmente ou totalmente o circuito;
- Alterar funcionalidades do circuito.

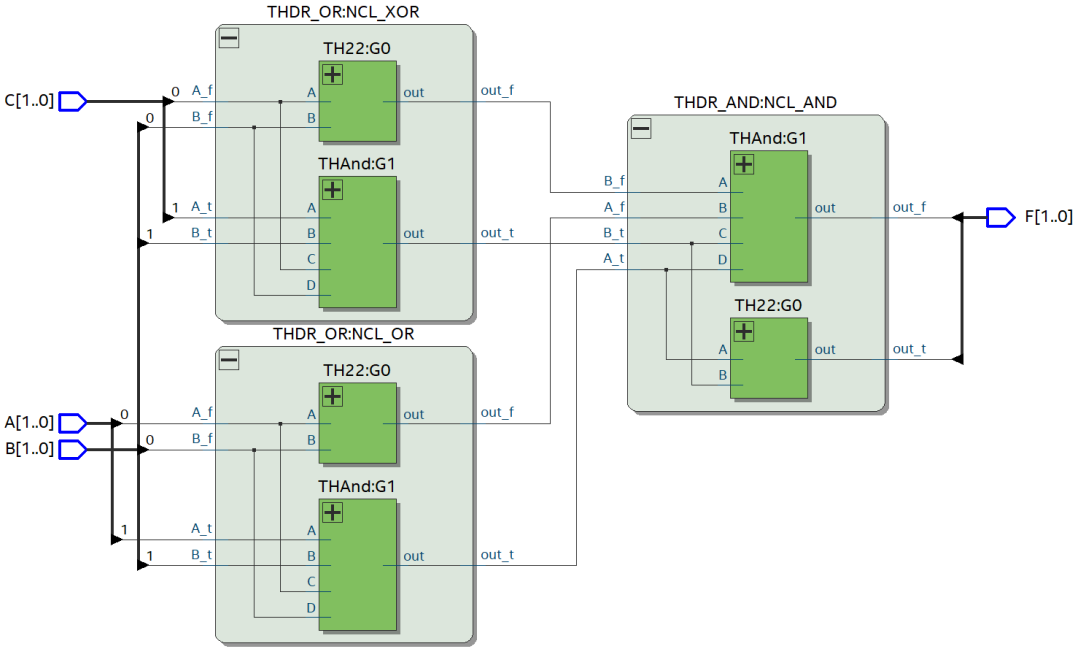


Figura 9 – Descrição da Função Lógica (2.2) em NCL.

As diversas possibilidades de alterações nos circuitos configuram uma vasta gama de THs. A Figura 10 apresenta a taxonomia de *Trojans* de acordo com 4 atributos diferentes. O primeiro atributo é referente à fase de inserção, que denota a possibilidade de inserção de THs em qualquer etapa do processo de fabricação dos circuitos. O momento em que o trojan será inserido depende de fatores como: acesso à etapa pelo usuário malicioso, conhecimento do funcionamento e aplicação do circuito, conhecimento sobre processos de fabricação, etc. Em seguida, o nível de abstração indica em que camada do circuito o trojan é inserido, seja no nível físico modificando as trilhas do circuito, área, capacitâncias ou potência até o nível de sistema onde podem ser utilizados para corrompimento de dados, vazamento de informações e afins. O mecanismo de ativação indica o momento em que os *trojans* serão ativos. Os *trojans* podem estar sempre ativos (caso menos comum, dado que a chance de serem detectados é maior), enquanto os casos de *trigger* (condição de ativação) dependem de uma condição específica para serem ativos.

Dentro desse grupo, ainda é possível destacar as condições para circuitos combinacionais ou sequenciais. Para circuitos combinacionais, os casos de *trigger* utilizam nós raros. Os nós raros são conjuntos de possibilidades de entrada e/ou interconexões entre elementos do circuito que são poucas vezes acionados. Já para circuitos sequenciais, diversas formas de *trigger* são possíveis. Por fim, o quarto atributo diz respeito ao *payload* ou carga útil, que é a parte executora do trojan, ou seja, ela é a responsável por modificar a resposta do circuito de acordo com as especificações do usuário.

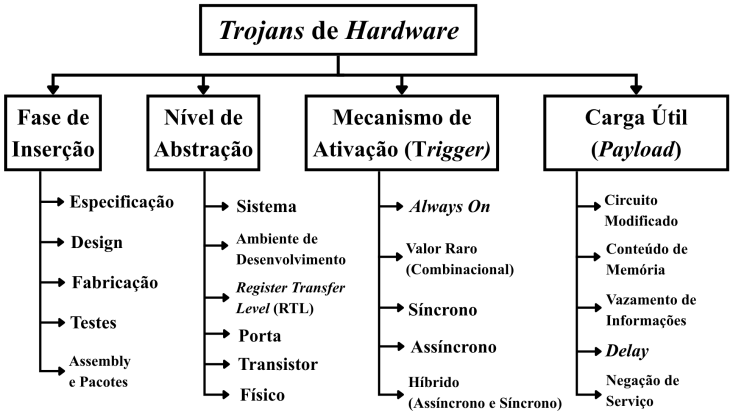


Figura 10 – Taxonomia *Trojans* de *Hardware*. Reconstrução baseada em (KARRI et al., 2010; CHAKRABORTY; NARASIMHAN; BHUNIA, 2009).

A Figura 11 apresenta três tipos de *trojans*: Combinacional, síncrono e assíncrono. O *trojan* combinacional, mostrada na Figura 11a, depende de uma condição de entradas específica para alteração na saída, neste caso, quando $A = B = 0$, o *trigger* é ativo e a carga útil do *trojan* passa a agir, sendo que nesta condição a saída $J_{original} = 0$. Dessa forma, a saída $J_{modificado}$ passa a ser '1', sendo que o estado deveria ser igual a '0', fornecendo um resultado errôneo. O TH síncrono, representado na Figura 11b, faz uso de um contador de N bits, que, quando o valor esperado é alcançado, o *trojan* é ativo, alterando o nó EJ. Já o *trojan* assíncrono, assim como o síncrono, utiliza contadores como *trigger*, como mostra a Figura 11c. Porém, o sinal de *clock* dos registradores, diferente do caso síncrono que utiliza o sinal do próprio sistema, o assíncrono utiliza uma condição em que $P = Q = 1$. Dessa forma, os contadores são atualizados em condições específicas, diferente do caso anterior, em que eles eram alterados a cada pulso de *clock* do sistema.

A versatilidade de inserção, forma de ativação, carga útil e funcionalidade torna a detecção de *trojans* uma tarefa complicada, haja vista que cada tipo de TH necessita de uma metodologia diferente para ser detectado.

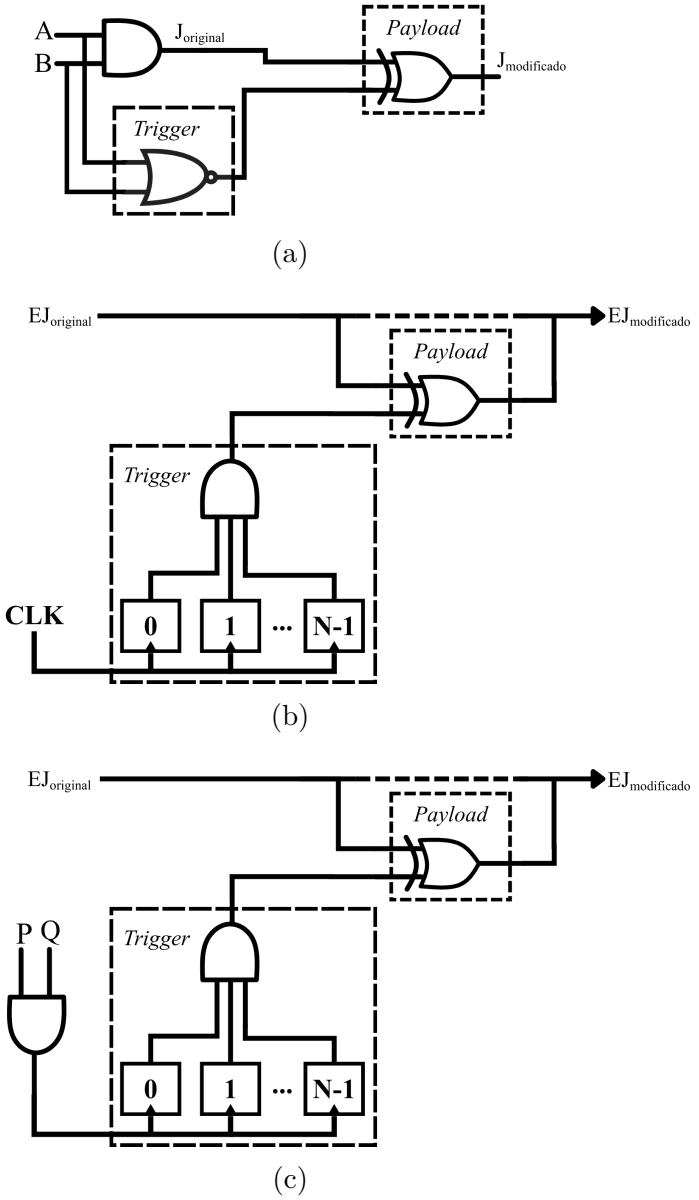


Figura 11 – (a) *Trojan* de *trigger* combinacional, (b) *Trojan* de *trigger* síncrono e (c) *Trojan* de *trigger* assíncrono. Reconstrução baseada em (CHAKRABORTY; NARASIMHAN; BHUNIA, 2009).

2.4 Taxonomia da Detecção de *Trojans* de *Hardware*

A detecção de THs em circuitos digitais é uma tarefa complexa, haja vista as diversas possibilidades de descrição desses circuitos, como visto na Seção 2.3.

A Figura 12 fornece a taxonomia para a detecção de THs. Os métodos não destrutivos são aqueles em que o circuito não é desmontado por completo ou em partes ao longo do processo de verificação. Essa abordagem é comumente utilizada, visto que o custo e o tempo e financeiro para o processo de verificação destrutivo.

As verificações invasivas e não invasivas são categorizadas pela necessidade ou não de modificação do circuito original para a detecção de trojans (CHAKRABORTY; NARASIMHAN; BHUNIA, 2009).

Já as verificações por tempo de teste levam em consideração apenas os momentos de teste durante as etapas de fabricação dos circuitos.

As detecções por teste lógico utilizam algoritmos computacionais que inferem um conjunto de vetores de entradas e analisam as respostas das saídas, verificando se as respostas geradas estão de acordo com os resultados esperados. A detecção de TH em circuitos booleanos e o método por trás de seus algoritmos são diversos, sendo atualmente baseados em métodos heurísticos aplicados à Inteligência Artificial, como por exemplo (HAYASHI; RUGGIERO, 2025) que analisa dois métodos de detecção: o primeiro baseado no Processamento Natural de Linguagem em conjunto com *Machine Learning* e o segundo a partir *Large Language Model* com *prompt* otimizado.

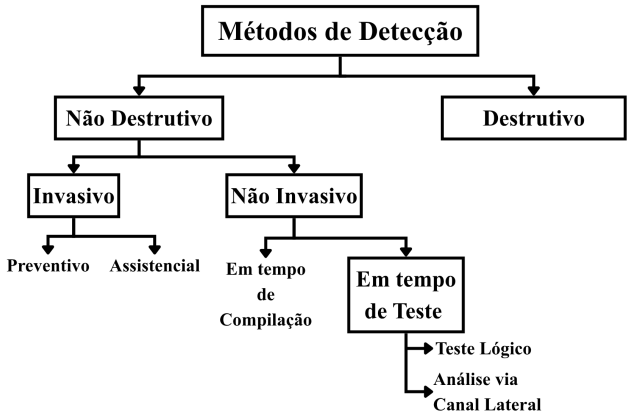


Figura 12 – Taxonomia de Detecção de *Trojans*. Reconstrução baseada em (CHAKRABORTY; NARASIMHAN; BHUNIA, 2009).

2.5 Detecção de *Trojans* de *Hardware* em Circuitos NCL

A detecção de THs é uma área em constante estudo, como visto na Seção 2.4. Analisando os circuitos NCL, dadas as divergências de construção desses circuitos em relação aos circuitos booleanos, testes como os de canal lateral e lógicos devem sofrer

alterações para a análise dos circuitos. Dessa forma, a área de detecção de trojans em circuitos NCL tem ganhado notoriedade.

2.5.1 Detecção por Análise de Atrasos de Caminho Global e Corrente Transiente

A Figura 13 destaca em formato de diagramas de blocos as representações de sistemas síncronos e assíncronos. Os circuitos QDI, como discutido anteriormente, não apresentam um sinal de *clock* para a sincronização dos dados entre as etapas do circuito, sendo utilizado, portanto, o protocolo *handshake*. Dessa forma, os dados são transmitidos de um bloco a outro apenas quando o bloco transmissor e o bloco receptor estiverem prontos. Dadas essas características dos circuitos assíncronos, (GUIMARAES et al., 2018) propuseram a detecção de THs em circuitos QDI via canal lateral, analisando os atrasos de caminho global e correntes transientes extraídos do gráfico da corrente de alimentação das partes do circuito.

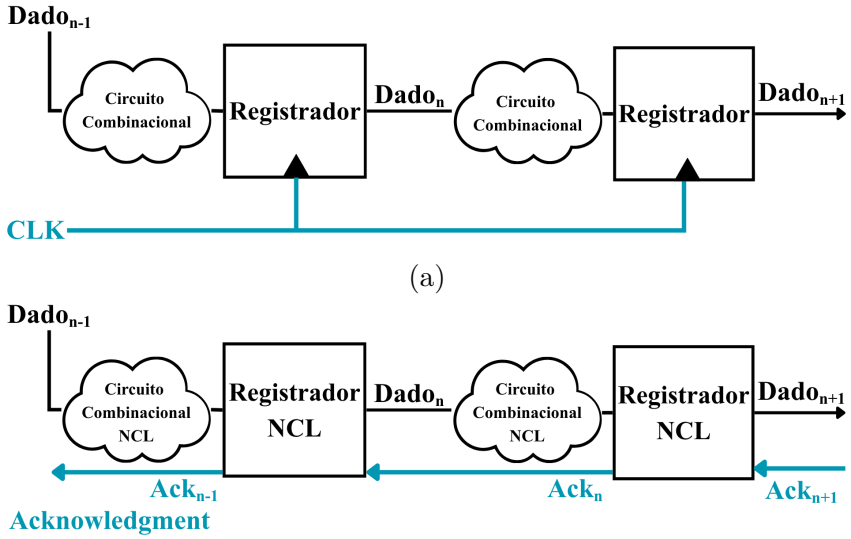
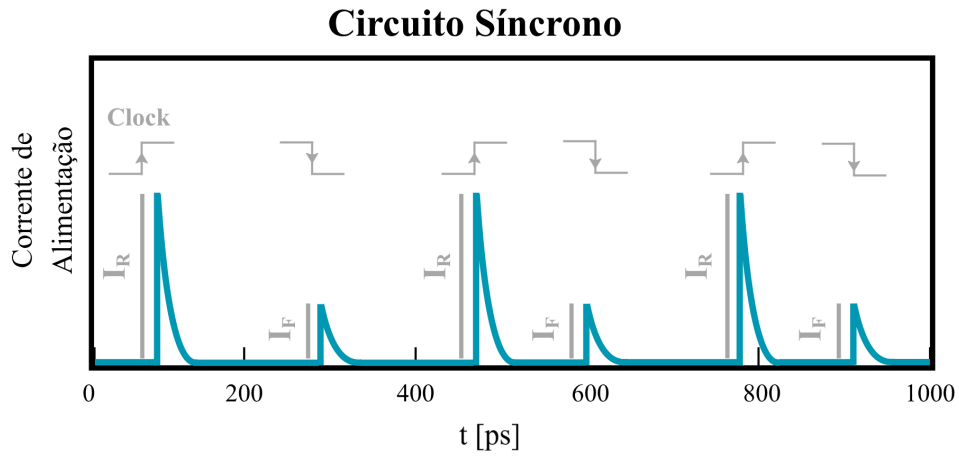
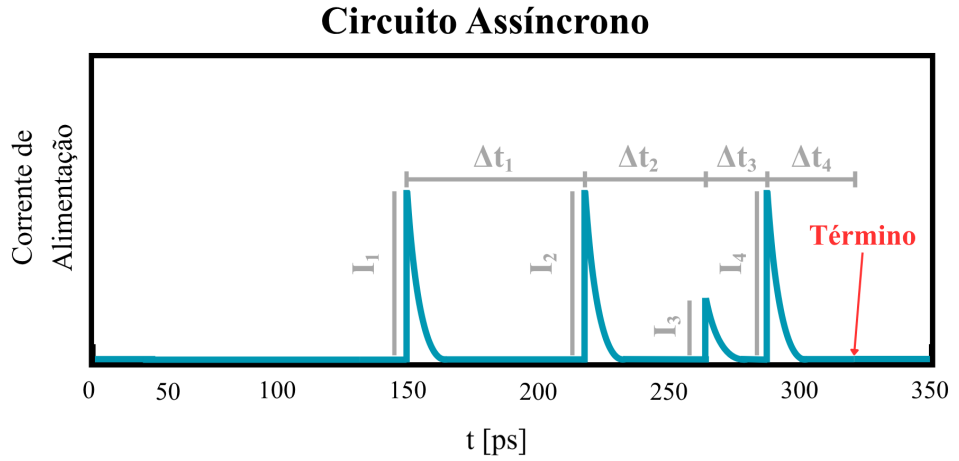


Figura 13 – Descrição Típica de circuitos a) síncronos e b) NCL assíncronos. Reconstrução baseada em (GUIMARAES et al., 2018).

A Figura 14a apresenta de maneira genérica o gráfico da corrente de alimentação de um circuito *pipeline* síncrono, enquanto a Figura 14b para o circuito assíncrono. Analisando o comportamento do gráfico do circuito síncrono, a cada borda do sinal de *clock* do sistema, como todos os blocos são controlados pelo mesmo sinal, o pico de corrente de subida ou de descida mensurada se dá pela contribuição de todos os estágios da *pipeline*. Dessa forma, independente do estágio em que o trojan é inserido, a alteração que ele causará na corrente de alimentação é vista de uma maneira geral, não sendo possível identificar em qual estágio ele foi inserido.



(a)



(b)

Figura 14 – Corrente de Alimentação do *pipeline* de circuitos a) síncronos e b) NCL assíncronos. Reconstrução baseada em (GUIMARAES et al., 2018).

Por outro lado, analisando o gráfico do circuito assíncrono, como os dados são transportados apenas quando requisitados, os picos de corrente são referentes a cada um dos estágios da *pipeline* separadamente. Assim, enquanto o dado é transportado do segundo para o terceiro estágio, o primeiro e o quarto estágio permanecem ociosos, não contribuindo com o pico de corrente. Essa característica dos circuitos QDI possibilita identificar em qual estágio o trojan foi inserido.

A partir do gráfico da corrente de alimentação, é possível extrair a corrente de transiente e o atraso de cada estágio separadamente. O atraso de cada estágio se dá pelo tempo entre dois picos de corrente. Dessa forma, o método de detecção se dá medindo a corrente de alimentação do circuito ouro (circuito sem a presença de trojans) e do circuito a ser analisado e comparando os parâmetros da corrente de transiente e do atraso global do sistema. Dentre os testes realizados pelos autores utilizando uma ULA de 8 bits em formato de *pipeline*, a alteração de corrente gerada por trojans de até 1,3% do tamanho total do circuito foram detectados com precisão superior a 90%.

2.5.2 Detecção a partir de Alterações Suaves em Circuitos *Pipeline* Macro Síncrono Micro Assíncrono

Em projeto de circuitos digitais, a mescla de circuitos síncronos e assíncronos associa robustez, baixa latência e consumo (FANT; BRANDT, 1996) dos circuitos assíncronos com a facilidade de sincronismo, descrição e implementação dos circuitos síncronos.

A partir de um modelo genérico de circuito Macro Síncrono Micro Assíncrono (MSMA) (LODHI et al., 2014), analisa a assinatura de tempo do protocolo de *handshake* da parte assíncrona em busca de modificações maliciosas. A análise tem como foco o nível de transistores, sendo capaz de analisar as fases pré e pós silício, assim, captando mudanças em espessuras no óxido, tensão de *threshold* e afins.

A Figura 15 apresenta o modelo genérico base. O *delay* (atraso) do estágio assíncrono se dá pela soma dos atrasos dos elementos que o compõem. Dado que a comunicação desses circuitos ocorre a partir do protocolo *handshake*, a assinatura de tempo dos sinais de aceito (ack) e requisição (req) do protocolo indica o *delay* de todos os blocos que compõem o estágio assíncrono. Dessa forma, comparando as assinaturas de tempo do circuito ouro e do circuito em análise, é possível detectar a presença ou não de Trojans no circuito em análise.

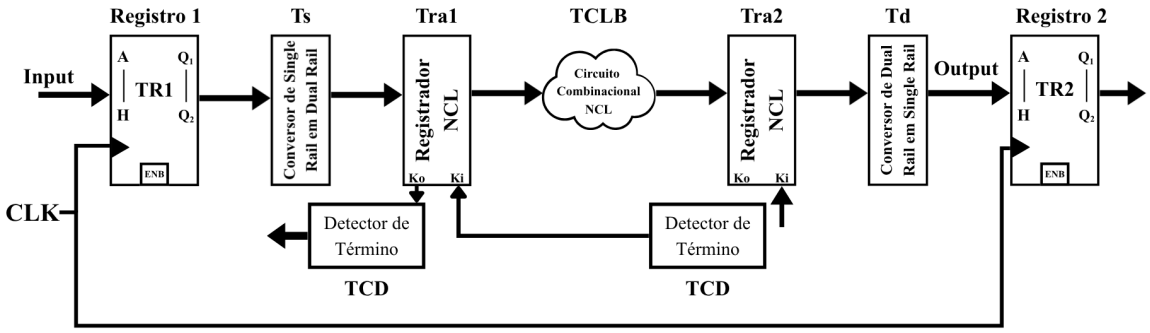


Figura 15 – *Macro Synchronous Micro Asynchronous Pipeline*. Reconstrução baseada em (LODHI et al., 2014).

2.5.3 Detecção por Verificação Formal via nuXmv

(LODHI et al., 2015) propõem um novo método de detecção de TH em circuitos MSMA baseado no processo de Verificação Formal utilizando o checador de modelos simbólico nuXmv. Neste caso, os autores mantêm a mesma base de circuito apresentado na Figura 15 para o processo de validação do método proposto.

Utilizando o nuXmv, os autores descrevem o circuito MSMA a partir de máquina de estados finitos. Em seguida, são formuladas as propriedades de verificação do circuito levando em consideração a parte funcional e a parte de tempo.

Para a parte funcional, as propriedades levam em consideração o estado inválido em que um sinal dual rail não pode apresentar, simultaneamente, o nível lógico alto em ambos os sinais que o compõem, a comunicação de dados só pode ocorrer se o protocolo de *handshake* (sendo representado pelos sinais de Ki e Ko dos registradores NCL) for corretamente realizado. Além disso, um dado só é considerado válido quando ele ocorre posterior a um estado *NULL* e, por fim, a cada transmissão de dado, o circuito NCL deve obrigatoriamente retornar ao estado *NULL*.

Já para a etapa de verificação de tempo, duas verificações são realizadas. Nelas, os autores utilizam os nomes dos módulos modelados no nuXmv para representar seus respectivos atrasos (`.delay`), onde: `sreg` representa um registrador síncrono, `areg` um registrador assíncrono (NCL), `add` o somador NCL, `sr2dr` o conversor de *single-rail* para *dual-rail*, e `dr2sr` o conversor de *dual-rail* para *single-rail*.

A primeira verificação diz respeito ao atraso de requisição da parte assíncrona (T*asynch*), que deve ser maior ou igual à soma dos atrasos dos componentes assíncronos (como registradores e o somador), conforme a Equação:

$$G(T_{\text{asynch}} \geq \text{areg0.delay} + \text{add0.delay} + \text{areg2.delay}) \tag{2.3}$$

Já a segunda verificação aponta que o tempo total da *pipeline* MSMA (T_{MSMA}) deve ser maior que a soma dos atrasos de todos os seus componentes (síncronos e assíncronos), como descrito na Equação:

$$\begin{aligned} G(T_{\text{MSMA}} > \text{sreg0.delay} + \text{sr2dr.delay} + \text{areg0.delay} \\ + \text{add0.delay} + \text{areg2.delay} \\ + \text{dr2sr.delay} + \text{sreg1.delay}) \end{aligned} \tag{2.4}$$

A partir da descrição do circuito em formato de máquina de estados e das verificações do circuito, os autores chegaram à conclusão de que o método proposto é capaz de identificar quais tipos de infecções cada elemento do circuito pode sofrer (seja na parte funcional ou de tempo), além de conseguir gerar como respostas contraexemplos que identificam os pontos de infecção de trojan na parte assíncrona da *pipeline*.

2.5.4 Detecção por Assinatura de Corrente via *One Class Support Vector Machine* (OC-SVM)

(GUIMARAES et al., 2018) analisam a presença de THs em circuitos QDI NCL a partir dos picos da corrente de transição e dos atrasos de caminho.

Analisando também a corrente do circuito, (GUAZZELLI et al., 2020) propõem um método a partir do uso de um algoritmo de *machine learning* em que toda a curva de

corrente é utilizada. O processo de detecção se dá, primeiramente, separando dois grupos de circuitos: os circuitos a serem analisados e os circuitos ouro.

A partir do estímulo de cada um dos circuitos por meio do envio de um *token* de dados, o percorrimento por toda a *pipeline* gera a assinatura de corrente transiente de cada um dos circuitos, que é extraída e separada posteriormente para ser utilizada no OC-SVM. Na etapa de treinamento, a ideia é utilizar as curvas das correntes de assinatura dos circuitos ouro de cada uma das partes da pipeline para treinar o algoritmo a reconhecer os padrões de corrente de cada estágio de acordo com as variações normais de processo de fabricação que possam existir. Dessa forma, a ideia é criar uma fronteira de classificação em que são determinados os limites de variabilidade advindos do processo de fabricação para a classificação das assinaturas de corrente dos circuitos analisados. Assim, caso as curvas de cada um dos estágios da *pipeline* obedeçam os limites, eles são considerados sem trojans, do contrário, há a presença de trojans.

Por fim, as correntes de assinatura dos circuitos a serem analisados passam pelo processo de classificação para que seja categorizado se houve ou não infecção.

Os resultados apresentados são comparados aos resultados apresentados em (GUIMARAES et al., 2018) e mostram que, para os estágios em que o algoritmo de (GUIMARAES et al., 2018) não obtinha 100% de acerto de acordo com o tamanho dos trojans, o método discutido conseguiu ser 100% assertivo e, além disso, conseguiu encontrar trojans com tamanhos ainda menores (entre 62 e 30 transistores) com a taxa de assertividade entre 82.9% e 100%.

2.5.5 Design e Detecção de *Trojans* de *Hardware* em Casos Ilegais em Circuitos NCL e *Multi-Threshold NULL Convention Logic* (MTNCL)

Até o momento, os métodos de detecção abordados analisam os momentos de transição entre o estado *NULL* e o de dados válidos para a detecção de THs.

(PONUGOTI et al., 2022), por outro lado, destaca os perigos de infecção de circuitos QDI no estado de dado inválido. Como discutido na Seção 2.1.1, a representação lógica por *dual-rail* resulta em um estado em que tanto os valores de entrada quanto de saída são descartados, dessa forma, sendo descartado do funcionamento desses circuitos. Porém, como os autores destacam, existem circuitos combinacionais que podem gerar casos de dados inválidos durante o seu funcionamento caso uma das entradas seja inválida também, como é o caso dos meios-somadores e somadores completos.

Dado que os THs devem atuar apenas nos casos de dados inválidos, três possíveis implementações são propostas: AND *Trojan*, NAND *Trojan* e MUX (Multiplexador) *Tree Trojan*. O AND Trojan trata-se de uma célula M de N 2 de 2 que emula parte do funcionamento de uma porta AND. Essa célula comporta-se como o *trigger* do Trojan, enquanto

o *payload* se dá por um mux, que alterna entre o dado a ser vazado e o funcionamento normal do circuito para a resposta do *trigger*.

O NAND *Trojan* também funciona como uma célula M de N 2 de 2, porém, emulando parte do funcionamento de uma porta NAND. Assim como o *Trojan* AND, o *trigger* se dá pela célula M de N enquanto o *payload* fica a cargo de um mux.

Por fim, o MUX *Tree Trojan* se dá por um conjunto de multiplexadores interconectados, sendo tanto o *trigger* quanto o *payload* compostos por mux.

Dadas as características de funcionamento de cada Trojan, os métodos de Observabilidade e Completude de Entrada não são suficientes para identificar todos os *Trojans*, dessa forma, os autores propõem condições de verificações específicas para cada um dos *Trojans*.

Em posse dessas condições de verificação, um circuito de descriptografia RSA (*Rivest–Shamir-Adleman*) NCL é utilizado para os testes de detecção, haja vista que os multiplicadores utilizados no processo utilizam somadores e meios-somadores para realização da operação. Para realização dos testes, os autores utilizaram o *software* de satisfabilidade Z3 SMT solver, que, a partir de um conjunto de regras (para este caso, as condições de verificação) e de um problema (neste caso o circuito RSA) é capaz de dizer se o conjunto de entradas é satisfatório ou não para o conjunto de regras proposto. Dessa forma, caso o conjunto de entradas gere respostas satisfatórias para o conjunto de regras dos *Trojans* no problema aplicado, é possível afirmar que há presença de modificações maliciosas no circuito.

Dentre os resultados apresentados, o método foi capaz de identificar com 100% de precisão todos os casos de *Trojans* aplicados aos circuitos independentemente de quantos estágios de somadores eram percorridos até gerar o estado ilegal. Além disso, o método proposto foi capaz de apresentar um comportamento polinomial à medida que o número de bits cresce e, o caso mais crítico testado para um RSA de 512 bits, a verificação levou 26 horas, sendo um valor rápido dada a complexidade do circuito.

2.6 Transições Probabilísticas

O processo de análise de circuitos digitais via Transições Probabilísticas (SALMANI; TEHRANIPOOR; PLUSQUELLIC, 2012; POPAT; MEHTA, 2016) propõe o equacionamento das probabilidades das saídas das portas lógicas que compõem o circuito serem iguais a ‘0’ ou ‘1’ a depender da probabilidade das entradas das portas lógicas. A Figura 16 destaca as equações para cada uma das portas lógicas primitivas que compõem os circuitos digitais. Os pares de entradas $\{IN_1(0), IN_1(1)\}$ e $\{IN_2(0), IN_2(1)\}$ correspondem aos valores probabilísticos das entradas IN_1 e IN_2 assumirem, respectivamente,

o nível lógico baixo $(IN_1(0), IN_2(0))$ ou alto $(IN_1(1), IN_2(1))$. Da mesma forma, o par de saídas $\{R(0), R(1)\}$ corresponde à probabilidade da saída de cada uma das portas lógicas serem iguais a '0' $(R(0))$ ou '1' $(R(1))$. Analisando R_0 e R_1, as equações descrevem o comportamento da tabela verdade de cada uma das portas lógicas de forma equacionada.

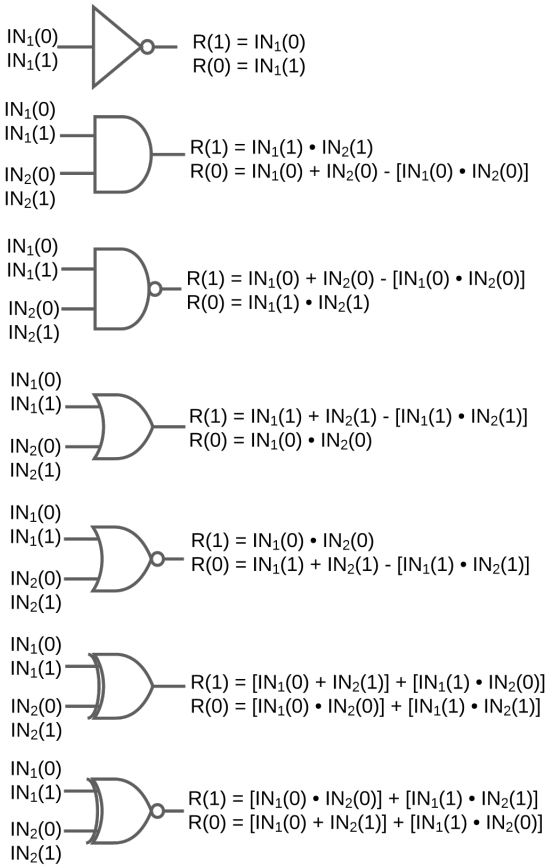


Figura 16 – Cálculo de Probabilidade das Portas Lógicas. Reconstrução baseada em (POPAT; MEHTA, 2016).

Como exemplo, considerem-se as equações para o cálculo da probabilidade de saída da porta lógica AND, conforme ilustrado na Figura 16. A probabilidade de a saída ser '1', denotada por $R(1)$, é calculada pelo produto das probabilidades de cada entrada ser '1': $R(1) = IN_1(1) \cdot IN_2(1)$. Isso reflete a lógica funcional da porta AND, cuja saída é '1' somente se ambas as entradas, IN_1 e IN_2, forem simultaneamente '1'. Por outro lado, a probabilidade de a saída ser '0', denotada por $R(0)$, é dada por $R(0) = IN_1(0) + IN_2(0) - (IN_1(0) \cdot IN_2(0))$. A saída de uma porta AND é '0' se a entrada IN_1 ou a entrada IN_2 for '0'. A equação representa a probabilidade da união desses dois eventos. O termo $IN_1(0) + IN_2(0)$ soma as probabilidades individuais de cada entrada ser '0', mas, nesse processo, o cenário em que ambas as entradas são '0' é contabilizado duas vezes. Portanto, o termo de subtração, $(IN_1(0) \cdot IN_2(0))$, que corresponde à probabilidade de ambas as entradas serem '0' ao mesmo tempo, é introduzido para corrigir essa contagem dupla, alinhando-se ao Princípio de Inclusão-Exclusão para eventos independentes.

Tendo em vista que cada porta lógica apresenta uma equação única que representa

o seu funcionamento, a multiplicação entre a probabilidade da saída ser '0' e a probabilidade da saída ser '1' gera a probabilidade de transição probabilística daquele elemento lógico em específico. Dado que o valor da transição probabilística será único para cada um dos elementos que compõem o circuito, ao analisar elemento a elemento entre dois circuitos digitais, eles serão iguais se e somente se todos os elementos que compõem ambos os circuitos forem os mesmos. Do contrário, ao analisar o valor da transição probabilística das saídas do circuito, o valor maior ou menor indica que uma ou mais portas lógicas do circuito foram alteradas ou houve a adição ou remoção de elementos lógicos. Como o cálculo é realizado porta lógica a porta lógica, é possível identificar a partir de que momento entre os circuitos houve alterações, facilitando o trabalho de identificação de alterações maliciosas.

3 Desenvolvimento

Tendo em vista o objetivo de identificação de THs por meio de transições probabilísticas, neste trabalho foi desenvolvido um programa que implementa o equacionamento de cada uma das portas lógicas e, por meio dos arquivos de simulação do circuito, é capaz de identificar cada um dos elementos e realizar o cálculo das transições probabilísticas elemento a elemento. Para que, em seguida, a partir da comparação entre os valores do circuito ouro e do circuito em análise, sejam geradas as conclusões de análise do circuito.

A descrição dos circuitos, bem como os testes de cada um, ocorreram na plataforma Intel® Quartus® Prime 23.1 versão *standard* (básica) em conjunto com o Questa*-Intel® FPGA. A escolha do conjunto de *software* se deu pela experiência no manuseio da plataforma e por serem *softwares* com versões gratuitas para o desenvolvimento dos circuitos.

O desenvolvimento do programa se deu em duas etapas: o gerador automático da *netlist* (lista de interconexões) e o analisador de transições probabilísticas. Essa abordagem foi escolhida pensando em ser um facilitador para o usuário, que, a partir de um arquivo de saída do próprio Quartus, será capaz de realizar a análise probabilística de todo o circuito. Para a implementação dos programas foi utilizada a linguagem de programação C++, tendo em vista a familiaridade com a linguagem, além de ser consolidada no mercado e conter uma base de desenvolvedores sólida.

3.1 Circuitos Teste para Validação Algoritmo

Como base de teste para o programa, três circuitos combinacionais foram selecionados: Codificador Gray NCL de 4 bits, *Barrel Shifter* NCL de 4 bits e uma Unidade Lógica Aritmética (ULA) de 5 bits. A descrição destes circuitos e dos trojans é realizada por meio de operandos lógicos NCL apresentados na Seção 2.2.4. Os testes apresentados nas subseções a seguir utilizam como *testbench* um conjunto limitado de entradas como exemplo de funcionamento [1].

3.1.0.1 Codificador Gray

O Codificador Gray segue o princípio de que a transição entre os valores de saída ocorre um bit por vez. Essa característica é útil em aplicações onde a mudança simultânea de múltiplos bits pode gerar valores errôneos, como por exemplo, em codificações de máquinas de estado (PIMENTA, 2017). O método de descrição do circuito em NCL se dá

[1] Acesso completo aos circuitos desenvolvidos (ouro e modificado) bem como aos *testbenchs* de cada um (de todas as possibilidades e dos casos específicos analisados) está disponível em: <https://github.com/JoaoPedro-P/Circuitos_Dissertacao>

a partir do processo apresentado na Seção 2.2.4.3. A Tabela 2 fornece o comportamento esperado das saídas dado o conjunto de entradas. Extraindo as expressões lógicas para cada uma das saídas via Mapa de Karnaugh, tem-se:

$$Out_3 = A \tag{3.1}$$
$$Out_2 = \overline{A}B + A\overline{B} \tag{3.2}$$
$$Out_1 = \overline{B}C + B\overline{C} \tag{3.3}$$
$$Out_0 = \overline{C}D + C\overline{D} \tag{3.4}$$
$$\tag{3.5}$$

Ou seja, as saídas Out_2, Out_1 e Out_0 são compostas por portas XOR. Como discutido na Seção 2.2.4.2, os operadores NCL são descritos por composições de células M de N. Neste caso, a porta lógica XOR NCL é composta por 2 células do tipo THxor0. A Figura 17 apresenta a simulação do circuito descrito pelo Código 3.1. Para o primeiro valor de entrada válido no circuito "10010101" a saída de acordo com a Tabela 2 equivale a saída "10100101", a qual é vista no mesmo instante de tempo da entrada (dado que a simulação é funcional e despreza os atrasos inerentes do circuito). Para o segundo caso "10100110" a saída correspondente é igual a "10101001", que pode ser vista no mesmo instante da entrada do segundo dado válido. Analisando a comparação entre a Tabela Verdade do circuito e o demais resultados da simulação do circuito a partir da conversão dos sinais para dual rail, infere-se que o circuito Codificador Gray NCL foi corretamente implementado.

```
1  module Encoder(A_t, A_f, B_t, B_f, C_t, C_f, D_t, D_f, out0_t, out0_f,
       out1_t, out1_f, out2_t, out2_f, out3_t, out3_f);
2
3  input A_t, A_f, B_t, B_f, C_t, C_f, D_t, D_f;
4
5  output out0_t, out0_f, out1_t, out1_f, out2_t, out2_f, out3_t, out3_f;
6
7  wire [5:0] outAnd_t, outAnd_f;
8
9  //Out 0
10 THDR_AND And0(C_f, C_t, D_t, D_f, outAnd_t[0], outAnd_f[0]);
11 THDR_AND And1(C_t, C_f, D_f, D_t, outAnd_t[1], outAnd_f[1]);
12 THDR_OR OR0(outAnd_t[0], outAnd_f[0], outAnd_t[1], outAnd_f[1], out0_t,
       out0_f);
13
14 //Out 1
15 THDR_AND And2(B_f, B_t, C_t, C_f, outAnd_t[2], outAnd_f[2]);
16 THDR_AND And3(B_t, B_f, C_f, C_t, outAnd_t[3], outAnd_f[3]);
17 THDR_OR OR1(outAnd_t[2], outAnd_f[2], outAnd_t[3], outAnd_f[3], out1_t,
       out1_f);
```

```
18
19 //Out 2
20 THDR_AND And4(A_f, A_t, B_t, B_f, outAnd_t[4], outAnd_f[4]);
21 THDR_AND And5(A_t, A_f, B_f, B_t, outAnd_t[5], outAnd_f[5]);
22 THDR_OR OR2(outAnd_t[4], outAnd_f[4], outAnd_t[5], outAnd_f[5], out2_t,
       out2_f);
23
24 //Out 3
25 assign out3_t = A_t;
26 assign out3_f = A_f;
27
28 endmodule
```

Código 3.1 – Descrição Codificador Gray NCL de 4 bits.

Tabela 2 – Tabela Verdade Codificador Gray 4 bits.

A	**B**	**C**	**D**	Out_3	Out_2	Out_1	Out_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

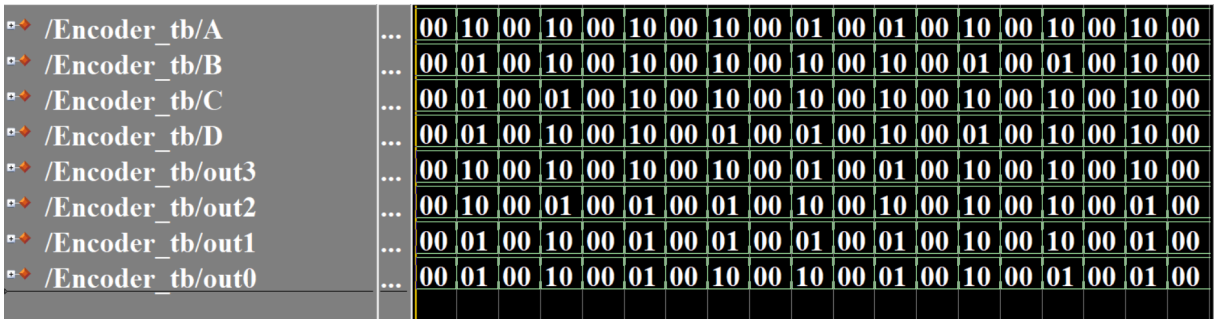


Figura 17 – Simulação via *testbench* Codificador Gray de 4 bits.

3.1.0.2 *Barrel Shifter*

O *Barrel Shifter* é um circuito combinacional capaz de deslocar e/ou rotacionar os bits de um sinal de entrada (PILLMEIER; SCHULTE; III, 2002). A quantidade de

rotações possíveis está relacionada com o número de bits de entrada do circuito, ou seja, para um circuito de 4 bits (nesse caso), o circuito pode rotacionar nenhuma, uma, duas ou três vezes. A direção em que o deslocamento será realizado (para a esquerda ou direita) dependerá do tipo de descrição escolhida pelo projetista. Sendo possível descrever circuitos deslocadores tanto uni quanto bidirecionais. Para fins de teste, um circuito deslocador para a direita foi escolhido. A Figura 18 fornece o esquemático geral do circuito proposto.

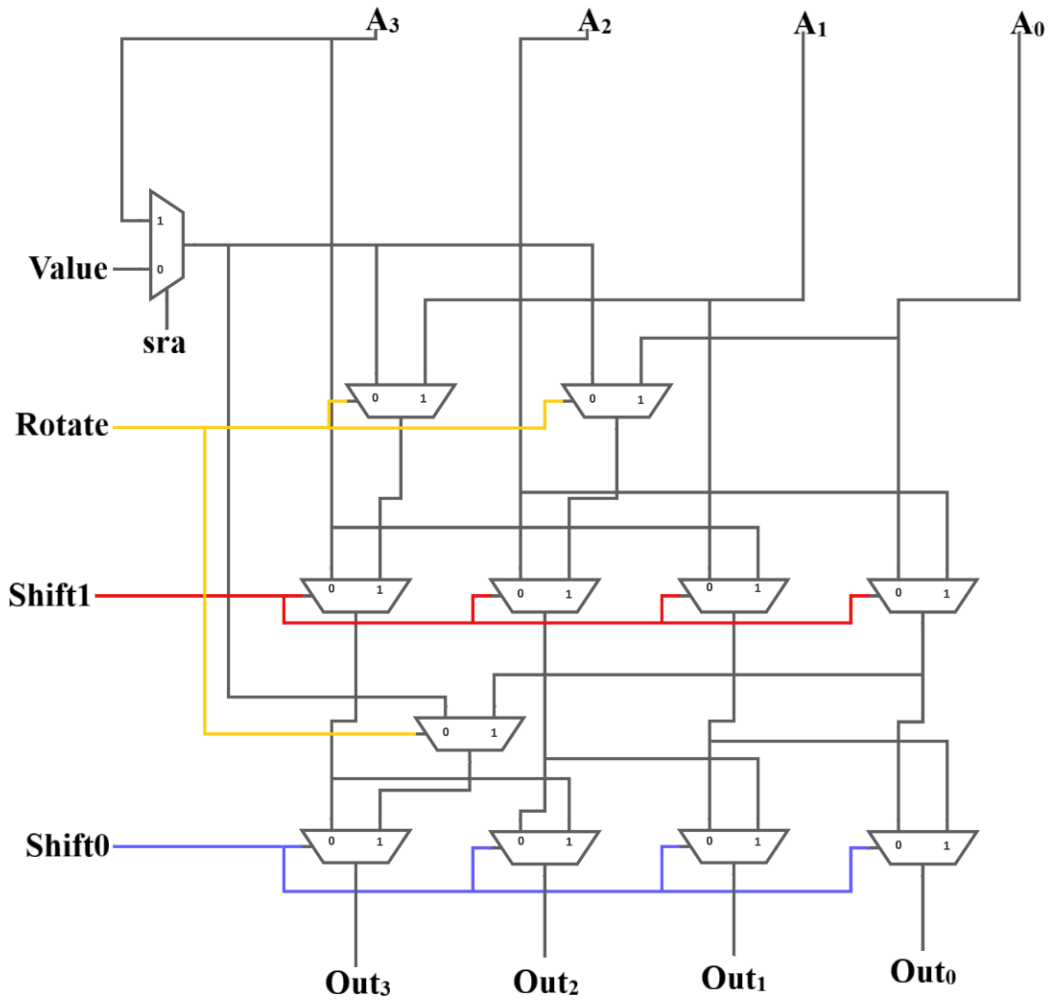


Figura 18 – Esquemático *Barrel Shifter* de 4 bits.

O funcionamento desses circuitos se dá pelo uso de vários mux (multiplexadores) 2 para 1, em que a partir do controle das chaves seletoras, é possível realizar os processos de deslocamento e rotação. A Equação (3.6) destaca a função lógica de um mux.

$$Out = (A \cdot hab) + (B \cdot \overline{hab}) \tag{3.6}$$

Como o circuito é composto por operadores AND e OR, estes devem ser descritos em NCL primeiro, como visto na Seção 2.2.4.3. A Figura 19 destaca a simulação do

multiplexador 2x1, na qual, para $hab = 10$, a saída encaminha o sinal de A para a entrada. Enquanto, para $hab = 01$, na saída é visto o valor B.

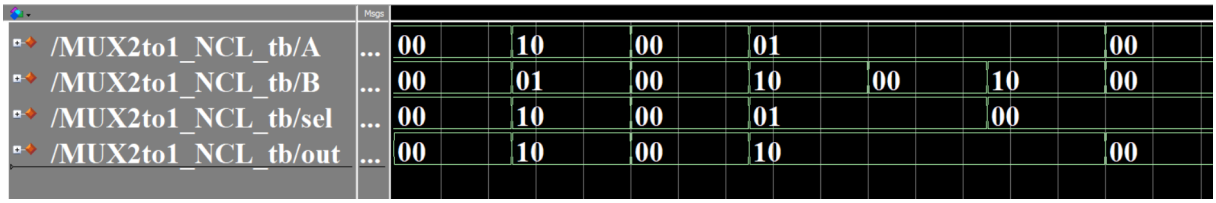


Figura 19 – Simulação MUX 2x1 NCL.

A partir da descrição do mux, o *Barrel Shifter* apresentado na Figura 18 é descrito a partir do Código 3.2. A simulação do circuito é vista na Figura 20, em que o valor de entrada "0111" (em dual rail "01101010") é submetido a uma rotação tripla, em seguida a dois deslocamentos e por fim a um deslocamento, validando, dessa forma, o funcionamento do circuito proposto.

```
1  module Barrel(A0_t, A0_f, A1_t, A1_f, A2_t, A2_f, A3_t, A3_f, sra_t,
       sra_f, rotate_t, rotate_f, shift1_t, shift1_f, shift0_t, shift0_f,
       value_t, value_f,
2              out0_t, out0_f, out1_t, out1_f, out2_t, out2_f, out3_t,
       out3_f);
3
4  input A1_t, A1_f, A2_t, A2_f, A3_t, A3_f, A0_t, A0_f, sra_t, sra_f,
       rotate_t, rotate_f, shift1_t, shift1_f, shift0_t, shift0_f, value_t,
       value_f;
5
6  output out0_t, out0_f, out1_t, out1_f, out2_t, out2_f, out3_t, out3_f;
7
8  wire       outMux_sra_t, outMux_sra_f, outMux_router2_t, outMux_router2_f
       ;
9  wire [1:0] outMux_router1_t, outMux_router1_f;
10 wire [3:0] outMux_shifter1_t, outMux_shifter1_f, outMux_shifter2_t,
       outMux_shifter2_f;
11
12 //value shifter
13 MUX2to1_NCL MUX_sra(sra_t, sra_f, A3_t, A3_f, value_t, value_f,
       outMux_sra_t, outMux_sra_f);
14
15 //first routers
16 MUX2to1_NCL MUX_router1_1(rotate_t, rotate_f, A1_t, A1_f, outMux_sra_t,
       outMux_sra_f, outMux_router1_t[0], outMux_router1_f[0]);
17 MUX2to1_NCL MUX_router1_2(rotate_t, rotate_f, A0_t, A0_f, outMux_sra_t,
       outMux_sra_f, outMux_router1_t[1], outMux_router1_f[1]);
18
19
20 //first shifters
```

```
21 MUX2to1_NCL MUX_shifter1_1(shift1_t, shift1_f, outMux_router1_t[0],
      outMux_router1_f[0], A3_t, A3_f, outMux_shifter1_t[0],
      outMux_shifter1_f[0]);
22 MUX2to1_NCL MUX_shifter1_2(shift1_t, shift1_f, outMux_router1_t[1],
      outMux_router1_f[1], A2_t, A2_f, outMux_shifter1_t[1],
      outMux_shifter1_f[1]);
23 MUX2to1_NCL MUX_shifter1_3(shift1_t, shift1_f, A3_t, A3_f, A1_t, A1_f,
      outMux_shifter1_t[2], outMux_shifter1_f[2]);
24 MUX2to1_NCL MUX_shifter1_4(shift1_t, shift1_f, A2_t, A2_f, A0_t, A0_f,
      outMux_shifter1_t[3], outMux_shifter1_f[3]);
25
26 //second router
27 MUX2to1_NCL MUX_router2_1(rotate_t, rotate_f, outMux_shifter1_t[3],
      outMux_shifter1_f[3], outMux_sra_t, outMux_sra_f, outMux_router2_t,
      outMux_router2_f);
28
29
30 //second shifters
31 MUX2to1_NCL MUX_shifter2_1(shift0_t, shift0_f, outMux_router2_t,
      outMux_router2_f, outMux_shifter1_t[0], outMux_shifter1_f[0], out3_t,
       out3_f);
32 MUX2to1_NCL MUX_shifter2_2(shift0_t, shift0_f, outMux_shifter1_t[0],
      outMux_shifter1_f[0], outMux_shifter1_t[1], outMux_shifter1_f[1],
      out2_t, out2_f);
33 MUX2to1_NCL MUX_shifter2_3(shift0_t, shift0_f, outMux_shifter1_t[1],
      outMux_shifter1_f[1], outMux_shifter1_t[2], outMux_shifter1_f[2],
      out1_t, out1_f);
34 MUX2to1_NCL MUX_shifter2_4(shift0_t, shift0_f, outMux_shifter1_t[2],
      outMux_shifter1_f[2], outMux_shifter1_t[3], outMux_shifter1_f[3],
      out0_t, out0_f);
35
36 endmodule
```

Código 3.2 – Descrição *Barrel Shifter* de 4 bits.

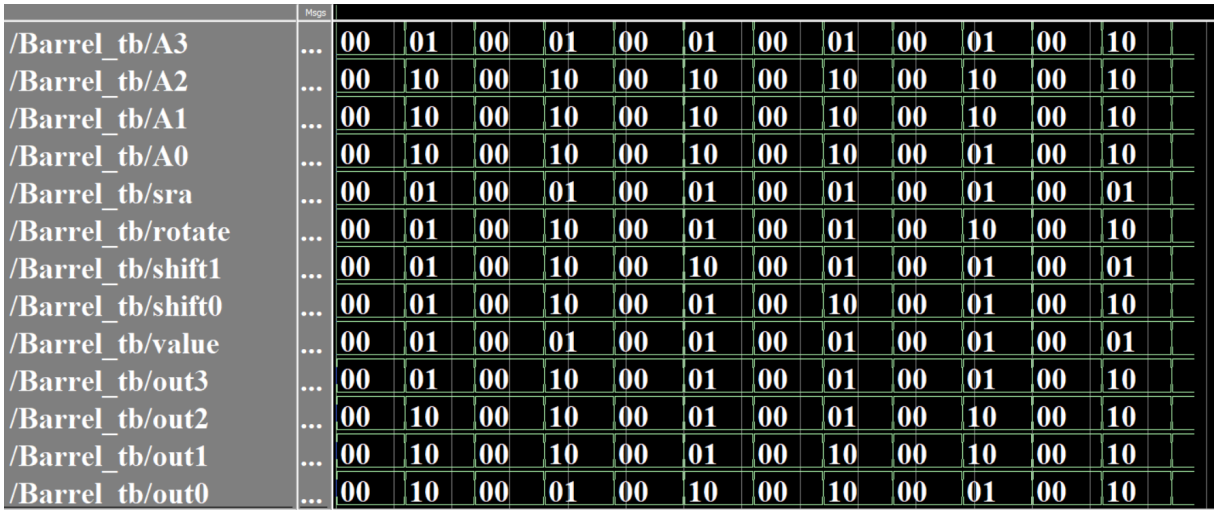


Figura 20 – Simulação *Barrel Shifter* NCL de 4 bits.

3.1.0.3 ULA

A ULA é o circuito responsável por realizar operações aritméticas entre um, dois ou mais operandos. Para o circuito de teste, a ULA recebe dois operandos de 5 bits, sendo o último bit de sinal, e realiza as operações de soma, subtração e XOR/AND bit a bit. A Tabela 3 apresenta o código de cada uma das operações da ULA.

Tabela 3 – Operações ULA.

Operação	**Seletora0**	**Seletora1**
Soma	01	01
Subtração	01	10
XOR bit a bit	10	01
AND bit a bit	10	10

Além das quatro operações, a ULA tem as *flags* de identificação para resultados nulos, negativos e caso haja *overflow*. Para as operações de soma (Op = '01') e subtração (Op = '10'), o circuito desenvolvido une as duas operações, sendo o controle realizado por uma chave seletora. As Equações (3.7) e (3.8) que representam, respectivamente, a resposta da operação e o bit de *carry* de saída foram obtidas via Mapa de Karnaugh a partir da Tabela Verdade.

$$Out = (A \cdot \overline{B} \cdot \overline{C_{in}}) + (A \cdot B \cdot C_{in}) + (\overline{A} \cdot \overline{B} \cdot C_{in}) + (\overline{A} \cdot B \cdot \overline{C_{in}}) \tag{3.7}$$

$$Carry_{out} = (B \cdot C_{in}) + (A \cdot \overline{Sel} \cdot C_{in}) + (\overline{A} \cdot Sel \cdot C_{in}) + (\overline{A} \cdot B \cdot Sel) + (A \cdot B \cdot \overline{Sel}) \tag{3.8}$$

Além das demais operações, o circuito ainda apresenta mux 2x1 que são responsáveis por atrelar a saída à operação desejada pelo usuário.

A *flag* de negativo está atrelada ao bit de sinal da resposta da operação de soma ou subtração. Como o bit de sinal está atrelado apenas às operações de soma/subtração, as demais operações devem se manter em '01'. Dessa forma, as chaves seletoras são utilizadas para afirmar a saída Neg se a operação realizada foi uma soma/subtração e se o último bit de sinal for '10'. Outro sinal a ser levado em consideração é o de não afirmar o sinal de resultado negativo dado que houve um *overflow* (estouro de campo). Essa condição acontece porque são utilizados 4 bits para a representação numérica. Logo, caso ocorra o *overflow*, o bit que deveria representar o 5 bit numérico será enviado para o bit de sinal, podendo gerar um resultado de número negativo incorretamente. Dado que são circuitos de teste, para esses casos os bits de sinal passam a ser considerados bits numéricos.

O sinal de *overflow* leva em consideração os sinais dos operandos e do resultado. Como discutido anteriormente, o bit de sinal pode ser alterado caso o *overflow* aconteça. Portanto, a condição de estouro de bit acontece quando, dado que os sinais dos operandos A e B são iguais, se o sinal do resultado for diferente, quer dizer que ocorreu um estouro de bit.

Por fim, a *flag* de zero sinaliza se a operação de soma ou subtração gerou um número nulo. Como os números são sinalizados, o resultado de uma operação ser igual a zero é só levado em consideração caso todos os bits do resultado sejam zeros também. Essa definição foi escolhida para não ocorrerem possíveis casos de *overflow* serem considerados como resultados iguais a zero também.

A Figura 21 destaca duas operações de soma realizadas. A primeira operação de soma é entre $A = B = 7 = 0101101010$ que resulta no valor 14 ("0110101001") sem que haja problemas de *overflow*. Já na segunda operação, $A = B = 15 = 0110101010$ o que resulta no valor 30 ("1010101001"), porém, como o bit mais significativo corresponde ao sinal do número, neste caso ocorreu um *overflow* como apontado pela *flag*.

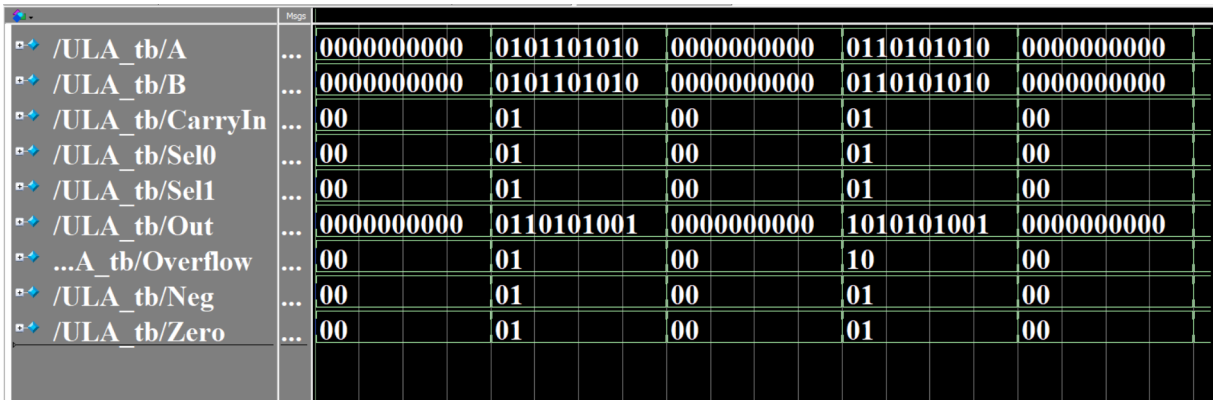


Figura 21 – Simulação ULA de 5 bits.

3.1.1 Inserção de *Trojans* de *Hardware* nos Circuitos de Teste

Como discutido na Seção 2.3, a composição dos THs e suas aplicabilidades são diversas. Dado que os circuitos de teste são circuitos combinacionais, serão adotados os *trojans* combinacionais para casos de nós raros. Os nós raros podem ser tanto combinações específicas de entradas como sinais internos do circuito. A seleção de cada um dos *trojans* abordados a seguir baseou-se na análise das tabelas-verdade e do modo de operação dos circuitos-alvo. Avaliou-se a probabilidade de ocorrência de cada conjunto de sinais, sejam eles de entrada ou internos, em relação a todos os estados possíveis do circuito.

3.1.1.1 Codificador Gray

Para o Codificador Gray, os seguintes *trojans* são aplicados:

1. Inversão do bit mais significativo da saída, alterando '10' para '01' quando a entrada for "10101010".
2. Inversão do segundo bit menos significativo da saída, alterando '01' para '10' para as entradas "01101001" e "01101010".
3. Inversão do bit menos significativo da saída, dadas as entradas "10011001" e "10011010".

Os *trojans* foram selecionados com base nas probabilidades de ocorrência dos sinais de entrada. Como as saídas do codificador não se repetem, a probabilidade de cada evento é sempre igual a 1/16, sendo 16 o número total de possibilidades de entrada circuito. Diante dessa distribuição uniforme, a escolha dos sinais para infecção foi feita de forma arbitrária. A Figura 22 apresenta a simulação do circuito Codificador Gray modificado. Para os casos em que houve a inserção de *trojans* (resultados circulados), os resultados foram divergentes do apresentado na simulação do circuito ouro da Figura 17, enquanto os demais permaneceram de acordo com a tabela, assim, indicando que a inserção foi realizada de maneira correta.

3.1.1.2 *Barrel Shifter*

Para o *Barrel Shifter*, os seguintes casos de *trojans* são inseridos:

1. Para que ocorra uma rotação, o sinal de $rotate$ e pelo menos um dos sinais de $shift$ devem ser ativos (ou seja, '10'). Portanto, um caso em que o sinal de $rotate$ esteja ativo enquanto nenhum dos sinais de $shift$ estejam, é considerado um caso raro. Assim, o trojan em questão deve substituir o valor original da saída Out_2 (neste caso, o nível lógico advindo da entrada A_2) pelo bit da entrada A_0.

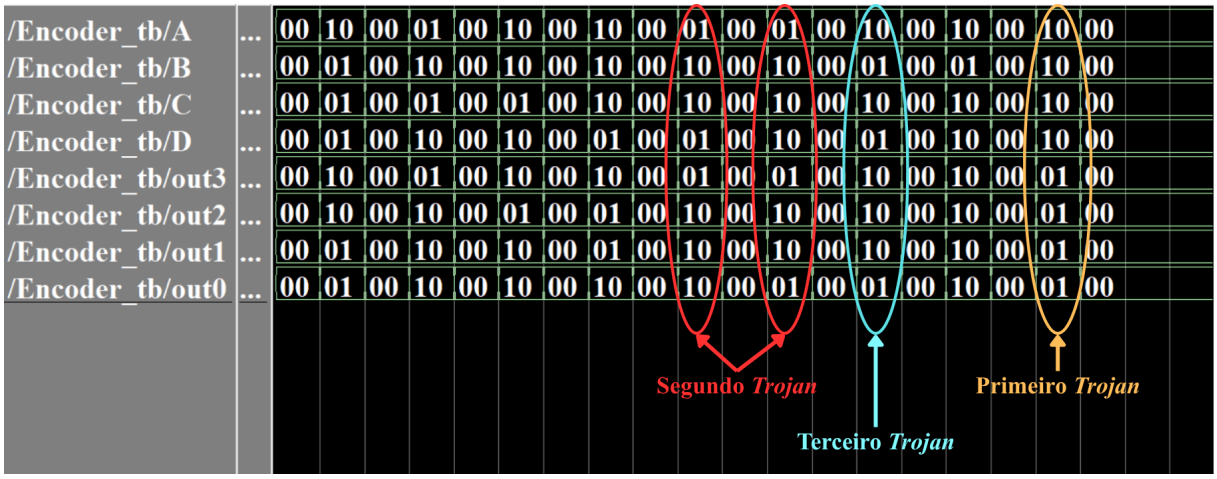


Figura 22 – Simulação Codificador Gray infectado.

2. O segundo trojan será ativo se, e somente se os valores dos nós internos do primeiro deslocador (controlados por $shifter1$) forem iguais a '10' e o sinal de $shifter0$ for igual a '10'. Neste caso, o trojan deve substituir o valor do segundo bit mais significativo da saída, pelo valor advindo do MUX controlado pela seletora sra.

A escolha dos THs propostos baseou-se na relação entre os casos de entrada e o funcionamento do circuito. Como a rotação dos bits também exige o seu deslocamento, um sinal de rotação sem deslocamento torna-se uma condição rara na prática. Além disso, a infecção altera a entrada A_2 para A_0, fazendo com que a saída do circuito divirja apenas quando $A_2 \neq A_0$. Essa situação apresenta uma probabilidade de 16/256, estabelecendo assim o primeiro *trojan*. Já o segundo *trojan* utiliza sinais internos para gerar uma nova condição rara, cuja probabilidade de ocorrência é de 10/256. Semelhante ao primeiro caso, a alteração na saída só é observada se $A_3 \neq A_1$. O resultado da simulação do circuito infectado é mostrado na Figura 23. Em comparação com o resultado da simulação do circuito ouro visto na Figura 20, para o penúltimo caso de teste, em que a entrada é igual a "01101001", para $rotate = 1$ o mesmo valor de entrada deveria ser visto na saída, dado que a rotação só acontece se houver deslocamentos. Porém, o resultado visto em simulação é "01011001", ou seja, o valor da saída Out_2 foi trocado pelo bit A_2, como esperado. Já o último resultado da simulação, o valor de entrada "10101010" deve sofrer rotação apenas uma vez (dado que $shifter0 = 10$), sendo assim, a saída deveria permanecer a mesma da entrada. Porém, ao analisar o resultado final da simulação, em Out_2 o valor visto é igual a '01'. Esse valor está atrelado ao valor de sra, que, caso fosse igual a '10', não alteraria o resultado (apesar de ativar a condição do *trojan*), criando um caso de falso negativo se apenas esse caso fosse testado. Analisando os casos específicos em que a saída foi modificada e os demais em que o comportamento normal do circuito foi atendido, é possível aferir que a implantação dos circuitos maliciosos foi realizada de maneira adequada.

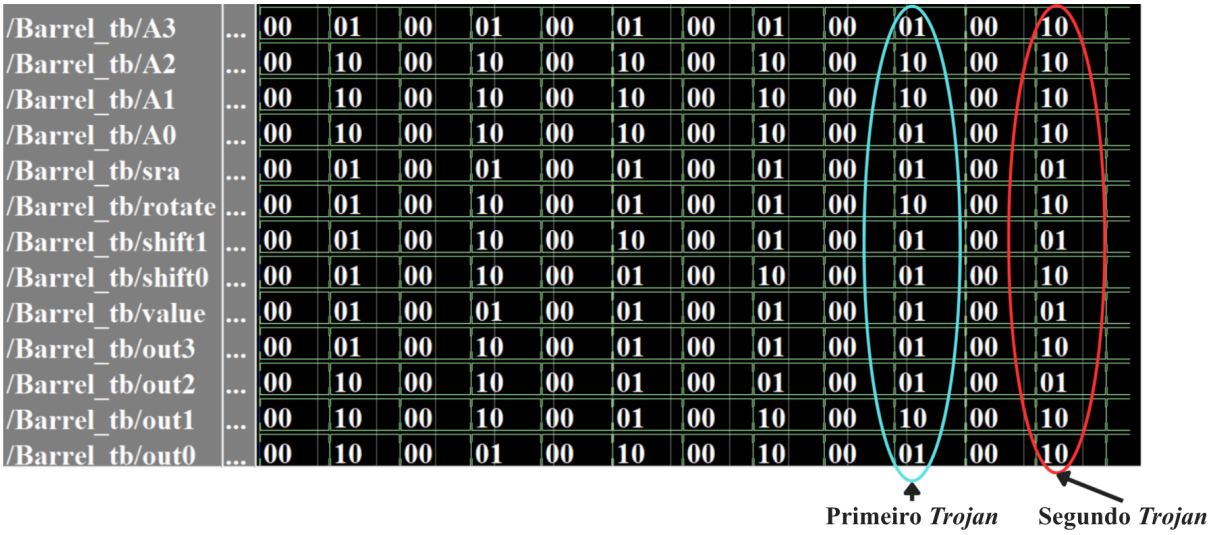


Figura 23 – Simulação *Barrel Shifter* infectado.

3.1.1.3 ULA

E, por fim, para a ULA, os seguintes THs são incluídos:

1. Seja B um número negativo, a operação a ser realizada seja uma subtração e que Cin seja igual a '10', o *trojan* deve definir a operação do primeiro somador/subtrator completo como uma soma;

2. Dado que o bit de *carry* de entrada vale '10' e a operação a ser realizada for uma XOR, a operação bit a bit entre o quarto bit mais significativo de ambos os operandos de entrada será uma AND e não uma XOR.

Assim como nos demais circuitos de teste, os *trojans* inseridos na ULA basearam-se em casos raros de entrada e na operação do circuito. O primeiro *trojan* decorre de uma entrada atípica, cuja probabilidade é de 256/8192. O segundo caso, por sua vez, explora o comportamento lógico do hardware. Como as operações lógicas bit a bit não dependem do *carry* de entrada, a presença do sinal $C_{in} = 10$ nesse contexto torna-se rara na prática, apresentando também uma probabilidade de 256/8192.. A Figura 24 representa a simulação das operações da ULA para alguns operandos. A primeira operação a ser realizada é a subtração $15 - (-15) - 1$ que deve gerar o resultado 29. Como a operação é o gatilho para o primeiro *trojan*, ele será ativo, invertendo a operação de subtração do primeiro bit por uma operação de soma. Porém, ao analisar o resultado, o valor de resposta continua 29. Esse comportamento acontece, pois, dada a Tabela Verdade 4 (em que $Operação = 01$ é soma e $Operação = 10$ é subtração) referente ao somador/subtrator completo, para as entradas $A = B = C_{in} = 10$, independentemente da operação, o resultado da operação e o bit de *carry* de saída são os mesmos. Essa condição se repete para outros casos na tabela que envolvam a ativação do *trojan*, sendo as duas únicas circunstâncias divergentes quando $A = B = 01$ e $C_{in} = 10$ ou $A = C_{in} = 10$ e $B = 01$.

Na primeira situação ($A = B = 01$ e $C_{in} = 10$), caso a operação a ser realizada seja uma soma, Out e C_{out} serão, respectivamente '10' e '01', enquanto para a subtração o par de saídas será $Out = C_{out} = 10$. Para a segunda situação ($A = C_{in} = 10$ e $B = 01$), em caso de soma $Out = 01$ e $C_{out} = 10$, enquanto para subtração $Out = C_{out} = 01$. Sendo assim, a segunda operação também tem o mesmo comportamento de o *trojan* ser ativo mas não impactar no resultado final, tendo em vista que o primeiro operando $A = 14 = 0110101001$ e o segundo $B = -3 = 1010100110$, o bit menos significativo de ambos e de C_{in} são, respectivamente, '01', '10' e '10'. Esses valores, tanto para a operação de soma quanto de subtração, geram os mesmos valores de saída e de *carry*, logo, não alterando o resultado final independentemente da operação realizada. Contudo, na terceira operação, em que $A = 15 = 0110101010$, $B = -10 = 1001101001$ e $C_{in} = 10$, a combinação entre os bits menos significativos e o bit de *carry* de entrada gera diferenças no resultado final, que impacta a resposta final do circuito somador/subtrator. Neste caso, ao invés da resposta ser $24 = 1010010101$, o resultado final foi $22 = 1001101001$. O mesmo comportamento pode ser visto na quarta operação, em que uma nova operação de subtração entre $A = -8 = 1010010101$, $B = -14 = 1001011001$ e $C_{in} = 10$ é realizada. O resultado esperado é $5 = 010101100110$, porém, dada a ativação do *trojan*, o resultado gerado é igual a $7 = 0101101010$. Por fim, a quinta operação ativa o segundo TH inserido. Neste caso, uma operação XOR bit a bit é realizada entre os valores $8 = 0110010101$ e $12 = 0110100101$, que deveria gerar o resultado $4 = 0101100101$, contudo, como o quarto bit mais significativo realiza uma operação AND, o resultado visto na saída será $12 = 0110100101$. Dados os resultados dos testes, é possível aferir que os THs foram corretamente inseridos e realizaram as modificações esperadas no circuito.

Tabela 4 – Operações somador/subtrator completo.

Operação	**A**	**B**	**C**	**Out**	$Carry_{out}$
01	01	01	01	01	01
01	01	01	10	10	01
01	01	10	01	10	01
01	01	10	10	01	10
01	10	01	01	10	01
01	10	01	10	01	10
01	10	10	01	01	10
01	10	10	10	10	10
10	01	01	01	01	01
10	01	01	10	10	10
10	01	10	01	10	10
10	01	10	10	01	10
10	10	01	01	10	01
10	10	01	10	01	01
10	10	10	01	01	01
10	10	10	10	10	10

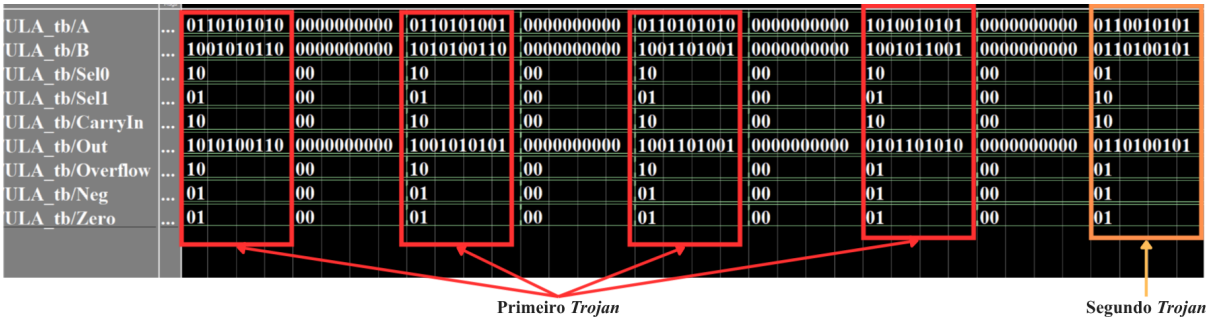


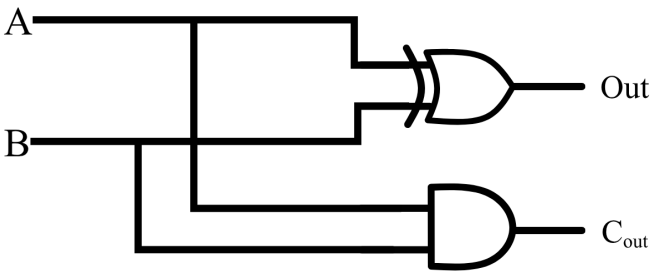
Figura 24 – Simulação ULA infectada.

3.2 Descrição Geral da Ferramenta

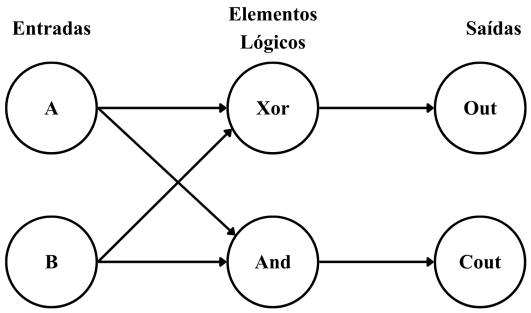
A ferramenta é composta por duas partes complementares (disponibilizadas via Github[2]). O primeiro programa tem como objetivo a criação automática da *netlist* a partir do arquivo de saída `.vo` (Verilog *Output*) do Quartus, que é utilizada no segundo algoritmo onde ocorrem as análises do circuito via o método de transições probabilísticas apresentado na Seção 2.6.

Apesar das funcionalidades distintas, ambos os programas utilizam como base um algoritmo de grafo para realização de suas tarefas. A escolha por esse tipo de algoritmo se dá pela facilidade de abstração dos circuitos digitais (grafos orientados). A Figura 25a apresenta um circuito meio somador que contém dois operandos de entrada A e B e duas saídas Out e C_{out}. Analisando o fluxo de dados, a informação primária chega pelas entradas, segue para os elementos lógicos e é encaminhada para as saídas. Como esse fluxo ocorre em todos os circuitos digitais, é possível abstrair esse comportamento em um grafo orientado (como indicado na Figura 25b). Os vértices são as conexões (entrada e saída) e os operadores lógicos (AND, OR, XOR, etc.) do circuito, enquanto as arestas são as conexões entre esses elementos.

[2] Repositório contendo os programas desenvolvidos: <https://github.com/JoaoPedro-P/Algoritmo_Transicao_Prob/tree/main>



(a)



(b)

Figura 25 – a) Circuito genérico meio somador e b) abstração em grafo orientado.

A Figura 26 apresenta as etapas dos algoritmos. O formato de disposição de dados na *netlist* não é padronizado, portanto cada empresa ou pesquisador adota o formato mais conveniente para o tipo de aplicação desenvolvida. Para a ferramenta desenvolvida, o padrão adotado será uma versão simplificada das *netlists* utilizadas para descrição dos circuitos de *benchmark* ISCAS-85 [3].

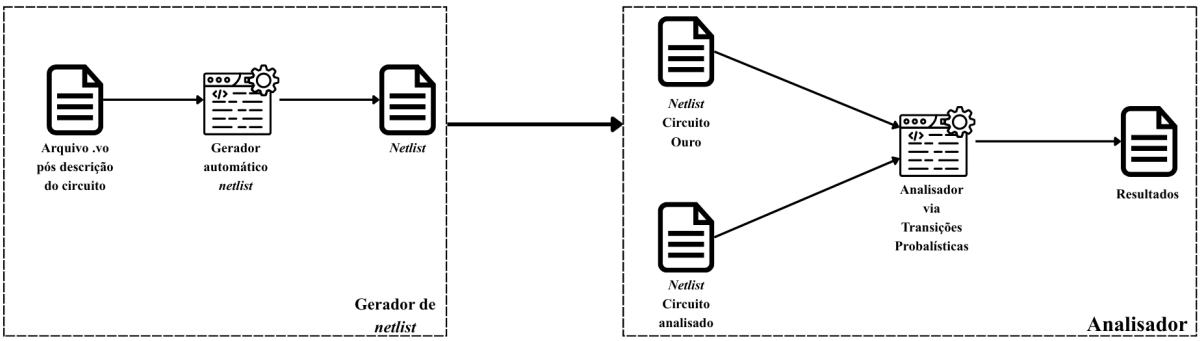


Figura 26 – Fluxo entre os programas.

A Figura 27a apresenta um exemplo de *netlist* no padrão `.isc`. A primeira informação corresponde ao identificador do elemento. Em seguida, acompanhado do número de identificação tem-se o tipo elemento: gat ou fan. O tipo gat sinaliza que o componente pode ser uma entrada ou operador lógico, enquanto o tipo fan indica que trata-se de um *buffer* de conexão de um componente do tipo gat. Para o tipo gat, a próxima informação

[3] Acesso completo das *netlists* disponível em: <https://pld.ttu.ee/~maksim/benchmarks/>

identifica se o elemento é uma entrada (inpt) ou um operador lógico (nand, or, xor, not, etc.), em contrapartida, para o tipo fan, a palavra from diz a quem aquele *buffer* está ligado. Os próximos dois números são, respectivamente, a quantidade de saídas e entradas que o componente tem para o tipo gat. Por outro lado, para o tipo fan há a identificação de qual elemento ele advém. Por fim, tem-se um marcador de fim de elemento. Como os operadores lógicos apresentam entradas, a definição de quais elementos constituem-se como entrada é identificada na linha abaixo da sua descrição. Como exemplo, tem-se o elemento 11 que é do tipo gat, que é uma nand com saída para duas outras partes do circuito e contém duas entradas. As entradas são os elementos 9 e 6, enquanto as saídas são compostas por dois *buffers* (14 e 15).

A Figura 27b apresenta o padrão simplificado do ISCAS-85 adotado. O uso de *buffers* foi removido, sendo os elementos interligados diretamente pelo número de identificação. Dessa forma, o uso de identificador de tipo do elemento também foi removido. Por fim, as tags no final de cada linha foram removidas e o tipo `out` foi acrescentado para facilitar a identificação das saídas do circuito, dado que no padrão original a saída do circuito estava atrelada aos últimos operadores lógicos declarados.

As seções seguintes descrevem o funcionamento dos algoritmos, bem como as funções mais importantes.

```
1    1gat inpt   1   0     >sa1
2    2gat inpt   1   0     >sa1
3    3gat inpt   2   0   >sa0 >sa1
8    8fan from    3gat     >sa1
9    9fan from    3gat     >sa1
6    6gat inpt   1   0     >sa1
10   10gat nand   1   2     >sa1
       1   8
11   11gat nand   1   2   >sa0 >sa1
       9   6
```

(a)

```
1   inpt   1   0
2   inpt   1   0
3   inpt   2   0
4   inpt   1   0
5   nand   1   2
       1   8
6   nand   1   2
       9   6
7   out    0   1
       5
8   out    0   1
       6
```

(b)

Figura 27 – *Netlist* padrão a) ISCAS-85 e b) adptado.

3.3 Gerador de *Netlist*

O processo de escrita manual das *netlists* torna-se uma tarefa exaustiva e suscetível a erros de conexões à medida que os circuitos crescem, podendo ser gerados falsos casos de *trojans* por falhas humanas. Portanto, um algoritmo transpilador é proposto, em que sua

entrada é o arquivo `.vo` e o resultado final é o padrão de *netlist* adotado. A Figura 28 ilustra o fluxo de execução completo do programa. Na etapa inicial, um conjunto de funções processa o arquivo de entrada para gerar um formato intermediário, o qual armazena em um arquivo `.txt` as informações referentes às conexões de entrada e saída, bem como aos elementos lógicos que compõem o circuito. Em seguida, esse arquivo gerado é submetido a um segundo módulo de funções, responsável por realizar a conversão dos dados para o padrão de *netlist* estrutural utilizado pelo algoritmo de transições probabilísticas.

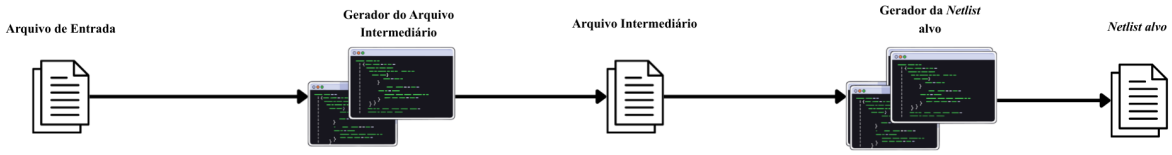


Figura 28 – Fluxograma do gerador de *netlist*.

O algoritmo tem duas funções principais: `generateIntermediateFile` e `generateSimplifiedNetlist`, as quais serão detalhadas posteriormente.

3.3.1 Configuração do Quartus para o arquivo `.vo`

Todos os projetos criados no Quartus geram o arquivo `.vo` após a simulação, porém, nativamente, o arquivo de saída fornece apenas a descrição do módulo topo e quais elementos serão utilizados para descrever o comportamento do circuito proposto. Dessa forma, uma alteração nas configurações do Quartus é necessária para que todos os módulos sejam instanciados e assim seja possível informar ao algoritmo gerador da *netlist* quais são os elementos lógicos que compõe aquele circuito. Para habilitar a configuração para manter a hierarquia original no arquivo de saída `.vo` tem-se o seguinte passo a passo:

1. Com o projeto já aberto no Quartus, acesse a aba `Assignments`
2. Em seguida, `Settings`
3. Na aba lateral `Category`, acesse em `EDA Tool Settings` a subcategoria `Simulation`
4. Clique no botão `More EDA Netlist Writer Settings`
5. Dentre as opções, habilite a `Maintain hierarchy`

Após habilitar a função `Maintain hierarchy` todos os módulos que foram instanciados no projeto estarão presentes no arquivo. Assim, recompile o projeto para que o arquivo `.vo` seja atualizado.

3.3.1.1 Geração do arquivo intermediário

O Código 3.3 destaca a função de geração do arquivo intermediário. Essa função tem como objetivo percorrer todo o arquivo `.vo` de maneira recursiva acessando cada um dos módulos em busca dos operandos lógicos e salvando as informações importantes como: a quem eles se conectam, a que módulo ele pertence, o tipo de operando, nome e demais informações relevantes. Além disso, essa função salva as informações sobre os sinais de entrada, saída e o nome geral do circuito.

Primeiramente, na linha 1 do código, a função `getTopModuleHeaderInfo` extrai as informações referentes ao nome, sinais de entrada e de saída do módulo topo.

Em seguida, na linha 2, a função `extractOutputConnections` obtém a informação de quais portas lógicas se conectam às saídas do circuito. Essa informação é retirada dentro do próprio arquivo, dado que existe uma instanciação específica para as saídas. O Código 3.4 fornece o exemplo da saída $out6_f$, em que a saída do operador lógico advindo de `MUX0|g2|G0` se conecta a ela.

Na linha 3 é iniciado o processo para extração dos operadores lógicos a partir da função `extractInstances`. Para compreensão do processo, o Código 3.5 referente a um somador/subtrator completo será usado como exemplo.

Primeiramente, a função começa a percorrer todo o arquivo `.vo` procurando chamadas de outros módulos ou definições de operandos lógicos. Para o caso de exemplo, a chamada do módulo somador/subtrator é encontrada (vide a partir da linha 1). Nele, há informações sobre o nome do módulo (Full_sum_sub), o nome da instância (SS0) e os sinais de controle (entradas, saídas, etc.). Como padrão de montagem do arquivo `.vo`, os sinais de entrada e saída do módulo são definidos nos espaços `.outX`, `.combY` ou em espaços com os nomes das variáveis do módulo topo. Existe a distinção, pois os sinais advindos de outros operadores são tratados como sinais internos e usam o padrão `.outX` ou `.combY`. Dessa forma, o algoritmo, ao se deparar com a chamada do módulo, armazena essas informações. Em seguida, o algoritmo irá procurar a instanciação do módulo ao longo do arquivo, e, quando for encontrada (vide linha 23), o algoritmo assimilará os sinais de `comb` e out armazenados anteriormente às definições dentro do módulo. Essa parte é necessária haja vista que os elementos internos utilizam os nomes das entradas como estão definidas nos módulos, ou seja, neste caso seria: `out1`, `out2`, `out3`, ... e, como o contexto de quem são esses sinais não está definido, o algoritmo não saberia posteriormente quem se liga a quem. Após essa etapa, o processo recursivo em busca dos operandos lógicos inicia-se.

Para que os operandos sejam encontrados, o algoritmo compara (linha 7 em diante do Código 3.3) todas as chamadas de módulos com os tipos básicos de operando (`THDR_AND`, `THDR_OR`, etc.) e, em caso de encontrar, o processo de armazenagem acon-

tece; do contrário, o algoritmo continua o percurso sobre os módulos encontrados. No caso utilizado como exemplo, dentro do módulo `Full_sum_sub`, o operando `THDR_AND_12` (linha 80) é o primeiro a ser encontrado. Por se tratar também de um módulo, a definição segue o mesmo aspecto do módulo `Full_sum_sub`. As entradas do módulo podem ser compostas por `comb`, `comb1`, `out4`, `out5`, `out7` e `out8`. Dado que o contexto de quem são cada um desses sinais, bem como o nome de instância do módulo foi salvo anteriormente, antes do operador ser armazenado são realizadas algumas modificações. Primeiramente, o nome do operador é convertido para uma palavra composta por duas partes: primeiramente pelo nome do módulo a que ele está atrelado (neste caso o módulo de nome $SS0$) e, após o separador (|) o nome específico do módulo ($Gout16$). Esse tipo de nomenclatura facilita em dois aspectos: o primeiro é em relação a que módulo pertence aquele operador e o segundo refere-se ao tipo de nome adotado pelo projetista. Em casos de descuido, um nome destoante do padrão adotado no projeto pode significar modificações inesperadas.

Em seguida, a partir do contexto de quem são cada um dos sinais de entrada do módulo `Full_sum_sub`, os sinais de controle de cada operando são substituídos pela conexão exata dele com outros elementos, facilitando o processo de montagem da *netlist* final.

O resultado de como o operando `THDR_AND_12` é visto no arquivo final é exposto no Código 3.6. Após todos os módulos serem analisados, os operandos são ordenados por ordem alfabética e numérica (a partir da linha 21 do Código 3.3) em relação ao módulo a que eles pertencem e, então, são salvos em um arquivo `.txt`, como apresentado no Código 3.7 e encaminhados para o próximo passo para a montagem da *netlist* final.

```
1    bool generateIntermediateFile(const string& vo_filename, const
     string& output_filename) {
2    auto [topModuleName, topInputs, topOutputs] = getTopModuleHeaderInfo
     (vo_filename);
3
4    auto outputConnections = extractOutputConnections(vo_filename);
5
6    vector<tuple<string, string, string>> topLevelInstances =
     extractInstances(vo_filename);
7    for (const auto& [instName, instType, instText] : topLevelInstances)
     {
8        if (isBaseCell(instType)) {
9            allFlattenedInstances.push_back({instType, instText});
10       } else {
11           auto parentConnections = extractPortMap(instText);
12           flattenAndResolve(instType, instName + "|",
     parentConnections, vo_filename, allFlattenedInstances);
13       }
14   }
```

```
15
16      ofstream outputFile(output_filename);
17
18
19      outputFile << "Instancia topo da hierarquia: " << topModuleName <<
    endl;
20      outputFile << "inputs:\n";
21      vector<string> sortedInputs(topInputs.begin(), topInputs.end());
22      sort(sortedInputs.begin(), sortedInputs.end());
23      for (const auto &in : sortedInputs) {
24          outputFile << in << " = " << in << endl;
25      }
26
27      outputFile << "\noutputs:\n";
28      vector<string> sortedOutputs;
29      for(const auto& pair : outputConnections) {
30          sortedOutputs.push_back(pair.first);
31      }
32      sort(sortedOutputs.begin(), sortedOutputs.end(), [](const string &a,
     const string &b) {
33          regex re(R"((.*?)\[(\d+)\]|(.+))");
34          smatch matchA, matchB;
35          string baseA, baseB;
36          int indexA = -1, indexB = -1;
37          if (regex_match(a, matchA, re)) {
38              baseA = matchA[1].matched ? matchA[1].str() : matchA[3].str
    ();
39               if (matchA[2].matched) indexA = stoi(matchA[2]);
40          }
41          if (regex_match(b, matchB, re)) {
42              baseB = matchB[1].matched ? matchB[1].str() : matchB[3].str
    ();
43              if (matchB[2].matched) indexB = stoi(matchB[2]);
44          }
45          if (baseA != baseB) return baseA < baseB;
46          return indexA < indexB;
47      });
48      for (const auto &outName : sortedOutputs) {
49          outputFile << outName << " = " << outputConnections[outName] <<
    endl;
50      }
51
52      outputFile << endl;
53
54      for (const auto &pair : allFlattenedInstances) {
55          outputFile << "// Instancia resolvida de " << pair.first << endl
    ;
```

```
56          outputFile << pair.second << endl;
57      }
58
59      outputFile.close();
60      cout << "Arquivo intermediario '" << output_filename << "' gerado
    com sucesso." << endl;
61      return true;
62  }
```

Código 3.3 – Função de geração do arquivo intermediário.

```
1  // Location: IOOBUF_X24_Y39_N9
2  fiftyfivenm_io_obuf \out6_f~output (
3    .i(\MUX0|g2|G0|out~0_combout ),
4    .oe(vcc),
5    .seriesterminationcontrol(16'b0000000000000000),
6    .devoe(devoe),
7    .o(out6_f),
8    .obar());
```

Código 3.4 – Definição da saída $out6_f$ no arquivo `.vo`.

```
1  //Chamada do modulo somador/subtrator completo
2  Full_sum_sub SS0(
3    .comb(\OR1|G1|comb~0_combout ),
4    .comb1(\OR1|G1|comb~1_combout ),
5    .out(\SS0|Gout10|G1|out~1_combout ),
6    .out1(\SS0|Gout10|G0|out~0_combout ),
7    .out2(\SS0|Gout20|G1|out~1_combout ),
8    .out3(\SS0|Gout20|G0|out~0_combout ),
9    .out4(\And2|G0|out~0_combout ),
10   .out5(\And3|G0|out~0_combout ),
11   .out6(\And4|G1|out~0_combout ),
12   .out7(\And2|G1|out~1_combout ),
13   .out8(\And3|G1|out~1_combout ),
14   .comb2(\And4|G1|comb~0_combout ),
15   .A_t(\A_t~input_o ),
16   .A_f(\A_f~input_o ),
17   .B_f(\B_f~input_o ),
18   .B_t(\B_t~input_o ),
19   .devpor(devpor),
20   .devclrn(devclrn),
21   .devoe(devoe));
22
23 //Instancia do modulo somador/subtrator completo
24 module Full_sum_sub (
25   comb,
26   comb1,
27   out,
```

```
28   out1,
29   out2,
30   out3,
31   out4,
32   out5,
33   out6,
34   out7,
35   out8,
36   comb2,
37   A_t,
38   A_f,
39   B_f,
40   B_t,
41   devpor,
42   devclrn,
43   devoe);
44 input    comb;
45 input    comb1;
46 output   out;
47 output   out1;
48 output   out2;
49 output   out3;
50 input    out4;
51 input    out5;
52 input    out6;
53 input    out7;
54 input    out8;
55 input    comb2;
56 input    A_t;
57 input    A_f;
58 input    B_f;
59 input    B_t;
60
61 // Design Ports Information
62
63 input    devpor;
64 input    devclrn;
65 input    devoe;
66
67 ...
68
69 wire \Gout8|G1|out~1_combout ;
70 wire \Gout9|G1|out~1_combout ;
71 wire \Gout9|G0|out~0_combout ;
72 wire \Gout8|G0|out~0_combout ;
73 wire \Gout19|G1|out~1_combout ;
74 wire \Gout17|G1|out~1_combout ;
```

```
75 wire \Gout17|G0|out~0_combout ;
76 wire \Gout19|G0|out~0_combout ;
77 ...
78
79
80 THDR_AND_12 Gout6(
81   .comb(comb),
82   .comb1(comb1),
83   .out(out4),
84   .out1(out5),
85   .out2(out7),
86   .out3(out8),
87   .out4(\Gout6|G0|out~0_combout ),
88   .out5(\Gout6|G1|out~1_combout ),
89   .devpor(devpor),
90   .devclrn(devclrn),
91   .devoe(devoe));
92
93 ...
```

Código 3.5 – Instância do Somador/Subtrator completo no arquivo `.vo`.

```
1 THDR_AND_12 SS0|Gout6 (
2   .comb(\OR1|G1|comb~0_combout ),
3   .comb1(\OR1|G1|comb~1_combout ),
4   .devclrn(devclrn),
5   .devoe(devoe),
6   .devpor(devpor),
7   .out(\And2|G0|out~0_combout ),
8   .out1(\And3|G0|out~0_combout ),
9   .out2(\And2|G1|out~1_combout ),
10  .out3(\And3|G1|out~1_combout ),
11  .out4(\SS0|Gout6|G0|out~0_combout ),
12  .out5(\SS0|Gout6|G1|out~1_combout )
13 );
```

Código 3.6 – Exemplo de Operador Lógico extraído pela função geradora do arquivo intermediário.

```
1 Instancia topo da hierarquia: Encoder
2 inputs:
3 A_f = A_f
4 A_t = A_t
5 B_f = B_f
6 B_t = B_t
7 C_f = C_f
8 C_t = C_t
9 D_f = D_f
```

```
10 D_t = D_t
11
12 outputs:
13 out0_f = \OR0|G0|out~0_combout
14 out0_t = \OR0|G1|out~1_combout
15 out1_f = \OR1|G0|out~0_combout
16 out1_t = \OR1|G1|out~1_combout
17 out2_f = \OR2|G0|out~0_combout
18 out2_t = \OR2|G1|out~1_combout
19 out3_f = \A_f~input_o
20 out3_t = \A_t~input_o
21 out4_f = \SS0|Gout10|G0|out~0_combout
22 out4_t = \SS0|Gout10|G1|out~1_combout
23 out5_f = \SS0|Gout20|G0|out~0_combout
24 out5_t = \SS0|Gout20|G1|out~1_combout
25 out6_f = \MUX0|g2|G0|out~0_combout
26 out6_t = \MUX0|g2|G1|out~1_combout
27
28 // Instancia resolvida de THDR_AND_16
29 THDR_AND_16 And0(
30   .out(\And0|G0|out~0_combout ),
31   .comb(\And0|G1|comb~0_combout ),
32   .out1(\And0|G1|out~1_combout ),
33   .comb1(\And0|G1|comb~1_combout ),
34   .C_f(\C_f~input_o ),
35   .D_t(\D_t~input_o ),
36   .D_f(\D_f~input_o ),
37   .C_t(\C_t~input_o ),
38   .devpor(devpor),
39   .devclrn(devclrn),
40   .devoe(devoe));
41
42 // Instancia resolvida de THDR_AND_17
43 THDR_AND_17 And1(
44   .out(\And1|G0|out~0_combout ),
45   .comb(\And0|G1|comb~0_combout ),
46   .out1(\And1|G1|out~1_combout ),
47   .C_f(\C_f~input_o ),
48   .D_t(\D_t~input_o ),
49   .D_f(\D_f~input_o ),
50   .C_t(\C_t~input_o ),
51   .devpor(devpor),
52   .devclrn(devclrn),
53   .devoe(devoe));
54
55 ...
```

Código 3.7 – Exemplo de arquivo intermediário.

3.3.1.2 Geração da *netlist*

Após o arquivo intermediário ser gerado (Seção 3.3.1.1), o algoritmo inicia o processo de formação da *netlist*. A função `generateSimplifiedNetlist` é a responsável pelo processo. O procedimento é dividido em três etapas complementares, sendo cada uma responsável por uma manipulação de dados da *netlist*. Essa abordagem foi adotada para facilitar o processo de depuração ao longo da implementação. A primeira etapa corresponde ao processo de *parsing* e criação dos nós. O objetivo desta etapa é utilizar os operandos lógicos do arquivo intermediário para criar uma estrutura de grafo orientado (utilizando `map`). Essa estrutura armazena, em pares chave-valor, as informações necessárias para a geração da *netlist* alvo. Nessa estrutura são armazenadas as informações sobre posição do elemento na *netlist*, tipo, elementos conectores de entrada e saída, etc.

```
struct CircuitNode {
    int id = 0;
    string name;
    string type; // Tipo simplificado (ex: "inpt", "and", "or", "out")

    // Conexoes logicas (grafo)
    vector<shared_ptr<CircuitNode>> fan_in_nodes;
    vector<shared_ptr<CircuitNode>> fan_out_nodes;

    // Armazenamento temporario durante o parsing
    vector<string> raw_input_signals;
    vector<string> raw_output_signals;
};
```

Código 3.8 – Estrutura das arestas do gerador automático de *netlist*.

O processo inicia percorrendo elemento a elemento do arquivo. Como os elementos podem ser `inputs`, `outputs` ou `gates`, o algoritmo atrela a variável `current_section` a cada elemento para identificar a qual tipo ele está atrelado. Em seguida, para cada tipo um tratamento para o armazenamento das informações será utilizado. Para os tipos `inputs` e `outputs`, outros dois `maps` são utilizados: `local_to_global_signal_map` e `submodule_output_map`. Esses maps são responsáveis por identificar a quais operandos esses elementos estarão ligados.

Para o caso dos operandos, um tratamento a mais em relação aos conectores é realizado. Tendo em vista o Código 3.6, identificar quais são os sinais de saída do operando é intuitivo, basta procurar entre as conexões, quais apresentam o mesmo nome do operando. Para esse caso, `.out4` e `.out5` são os sinais de saída. Porém, para os sinais de entrada, a mesma premissa não é verdadeira. Haja vista que `comb`, `comb1`, `out`, `out1`, `out2` e `out3` podem ser consideradas entradas do módulo. Portanto, a função `parseSignalName` é utilizada para identificação desses casos, dado que um sinal só será considerado entrada, caso: sejam pares de saída de mesmo operando ou do mesmo sinal

de entrada. Porém, para o primeiro evento, além de serem saídas do mesmo operando, a origem de dentro do operando deve ser diferente. Como exemplo, as conexões `comb` e `comb1` advêm do mesmo elemento `OR`, porém, são sinais da mesma origem `G1` de dentro do operando. Neste caso, esse sinal não pode ser considerado uma entrada do operando `THDR_AND_12`. Em contrapartida `out` e `out2` ou `out1` e `out3` são entradas desse operando, pois, são sinais advindos do mesmo operando (`And2` para `out` e `out2` e `And3` para `out1` e `out3`) porém de origens diferentes dentro dos respectivos elementos (`out` e `out1` correspondem à origem `G0`, enquanto `out2` e `out3` advêm de `G1`). Esse tipo de análise é importante para garantir a integridade das conexões nos próximos passos, haja vista que caso qualquer sinal fosse considerado, os operandos apresentariam mais conexões de entrada do que deveriam.

Para garantir que todos os sinais sem pares foram removidos dos elementos, a etapa um ponto cinco tem como objetivo uma segunda análise a cada um dos elementos aplicando essa verificação. Essa etapa é necessária, pois na primeira verificação os sinais de entrada por apresentarem o sufixo (`input_o`), diferente dos demais sinais (`comb 0/1_combout`) podem ser considerados de forma errônea. Portanto, neste momento cada elemento é analisado em busca dessas condições não contempladas no passo anterior para serem removidas.

Após a etapa 1 e 1,5 do algoritmo identificarem cada um dos elementos e suas respectivas conexões de entrada e saída, a etapa dois inicia o processo de conexão dos elementos utilizando a ideia de um grafo direcionado. O motivo desse comportamento foi discutido na Seção 3.2. Como cada elemento é definido por um conjunto de características definidas na `struct CircuitNode`, os elementos a quem eles se conectam são também armazenados nessa estrutura, no vetor de ponteiros inteligentes `fan_in_nodes` e `fan_out_nodes`. Essas variáveis são vetores de ponteiros inteligentes (são autolimpantes após o uso do programa para que não ocorra *overflow* de memória) que fornecem exatamente a quem o elemento analisado será conectado. Portanto, a etapa dois tem como objetivo percorrer cada um dos elementos e ir preenchendo esses vetores com a informação de a quem cada elemento se conecta.

Após todos os elementos serem corretamente definidos, a etapa três tem como objetivo a escrita da *netlist* final. A partir da iteração sobre todos os elementos, primeiramente são separados em vetores cada um dos elementos referentes aos tipos `inputs`, `gates` e `outputs`. Esse passo é importante para que no momento da escrita no arquivo de saída, sejam primeiramente posicionadas as entradas no topo, os elementos lógicos no meio e as saídas no fim. Em seguida, cada um dos vetores são ordenados por ordem alfabética e numérica. Por fim, a partir do vetor de `inputs`, o arquivo de saída começa a ser escrito. Sendo primeiramente posicionadas as entradas, e, a partir das informações de conexão advindas de `fan_in_nodes` e `fan_out_nodes` os demais elementos são escritos

no arquivo de saída. O Código 3.9 apresenta o resultado final da *netlist* do `Encoder` utilizado como exemplo. A cada elemento foi adicionado também o nome da instância logo a frente da definição na *netlist*, com o intuito de facilitar a identificação de cada elemento do circuito original.

```
1 inpt 7 0 //A
2 inpt 10 0 //B
3 inpt 4 0 //C
4 inpt 2 0 //D
5 and 3 2 //And0
  3 4
6 and 1 2 //And1
  3 4
7 and 6 2 //And2
  2 3
8 and 6 2 //And3
  2 3
9 and 1 2 //And4
  1 2
10 and 1 2 //And5
  1 2
11 and 1 2 //MUX0|g0
  5 27
12 and 1 2 //MUX0|g1
  5 37
13 or 1 2 //MUX0|g2
  11 12
14 or 1 2 //OR0
  5 6
15 or 1 2 //OR1
  7 8
16 or 1 2 //OR2
  9 10
17 and 1 2 //SS0|Gout0
  7 8
18 and 1 2 //SS0|Gout1
  2 17
19 and 2 2 //SS0|Gout2
  7 8
20 and 1 2 //SS0|Gout3
  2 19
21 and 2 2 //SS0|Gout4
  2 7
22 and 1 2 //SS0|Gout5
  8 21
23 and 2 2 //SS0|Gout6
  7 8
```

```
24 and 1 2 //SS0|Gout7
  2 23
25 or 1 2 //SS0|Gout8
  18 20
26 or 1 2 //SS0|Gout9
  22 24
27 or 2 2 //SS0|Gout10
  25 26
28 and 1 2 //SS0|Gout11
  2 8
29 and 1 2 //SS0|Gout12
  1 7
30 and 1 2 //SS0|Gout13
  2 29
31 and 1 2 //SS0|Gout14
  1 21
32 and 1 2 //SS0|Gout15
  1 23
33 and 1 2 //SS0|Gout16
  1 19
34 or 1 2 //SS0|Gout17
  28 30
35 or 1 2 //SS0|Gout18
  31 32
36 or 1 2 //SS0|Gout19
  33 35
37 or 2 2 //SS0|Gout20
  34 36
38 out 0 1 //out0
  14
39 out 0 1 //out1
  15
40 out 0 1 //out2
  16
41 out 0 1 //out3
  1
42 out 0 1 //out4
  27
43 out 0 1 //out5
  37
44 out 0 1 //out6
  13
```

Código 3.9 – *Netlist* final do circuito Encoder de exemplo.

3.4 Algoritmo de Transições Probabilísticas

Com as *netlists* dos circuitos geradas, o processo de análise via transições probabilísticas é realizado. A sequência de etapas para o funcionamento do algoritmo é descrita a seguir:

1. Definir a *netlist* original como a referência para a análise.
2. Percorrer a *netlist*, armazenando em uma estrutura de dados os seus elementos e respectivas propriedades: identificador, tipo (entrada, saída ou elemento lógico), conexões de entrada e os valores de probabilidade (saída 0, saída 1 e transição).
3. Configurar a probabilidade dos estados de entrada como 0.25. Dado que a lógica dual-rail possui quatro estados possíveis (*NULL*, DATA0, DATA1, INVALID), a probabilidade para cada um é de 1/4.
4. Percorrer a *netlist* das entradas até as saídas, calculando as probabilidades de saída 0 e 1 para cada elemento lógico a partir das equações da Seção 2.6. Os termos IN1(0), IN1(1), IN2(0) e IN2(1) são, respectivamente, a probabilidade da entrada 1 apresentar nível lógico 0, entrada 1 apresentar nível lógico 1, e assim por diante.
5. Calcular a probabilidade de transição de cada elemento a partir da equação: $P_{transicao} = P_{out=0} \cdot P_{out=1}$.
6. Definir a *netlist* a ser comparada com a original para a análise.
7. Repetir os passos de armazenamento e cálculo (itens 2 a 5) para a *netlist* sob teste.
8. Dispor em formato de tabela o identificador e o valor da probabilidade de transição dos elementos da *netlist* original e da analisada.
9. Percorrer todos os caminhos das saídas até as entradas do circuito original e do analisado, utilizando os valores da tabela para cada elemento.
10. Comparar se os caminhos percorridos apresentam os mesmos valores. Caso divirjam, o último elemento em que os valores divergiram antes que eles coincidam é identificado como o local de inserção do trojan.

A partir do passo a passo, o programa identificador foi implementado. O programa é dividido em três partes: extração das informações dos arquivos de entrada, cálculo das probabilidades dos elementos e análise das divergências entre os circuitos.

3.4.1 Análise do arquivo de entrada

Com a *netlist* fornecida ao programa, a função `parseNetlist` tem como objetivo identificar cada um dos elementos da *netlist* e salvá-los no formato `struct Element`, descrito no Código 3.10, que contém informações sobre a posição do elemento na *netlist*, o seu tipo, conexões de entrada e saída, além dos valores de probabilidades das saídas serem iguais a '0' ou '1', etc.

Durante o laço de repetição, os valores de `prob_0` e `prob_1` para as entradas são previamente definidos como 0.25, como discutido na definição do algoritmo.

Para cada elemento instanciado, as informações são salvas em um `map` que contém como valor cada um dos elementos instanciados no padrão `Element`. Dessa forma, a *netlist* se comporta como um grafo orientado, o que facilita o cálculo das probabilidades de transições para os elementos subsequentes às entradas.

Como padrão de projeto, algumas estruturas mais complexas como multiplexadores e somadores/subtratores completos são aceitos pelo algoritmo como modo de facilitar a descrição da *netlist* manualmente. Em relação à descrição dos demais tipos de elementos lógicos, os muxes utilizam uma segunda linha, a qual informa a chave seletora. Da mesma forma, o subtrator/somador completo utiliza a segunda linha para informar o bit de *carry* de entrada. Esses valores são armazenados na variável `selectors`. Da mesma forma, como os somadores/subtratores apresentam bit de *carry* de saída, além da saída aritmética, é necessário a definição da probabilidade dessas saídas serem '0' ou '1'. Como esses circuitos combinacionais apresentam duas saídas, o sufixo .1 e .2 são utilizados para identificar quais das saídas estão sendo conectadas ao elemento subsequente. Caso o sufixo seja .1, trata-se da saída aritmética, já o sufixo .2 informa a saída do bit de *carry*. O Código 3.11 apresenta a definição desses dois elementos em uma *netlist*.

```
1  struct Element {
2      int id;
3      string type;
4      vector<float> connections; // Conexoes gerais (entradas ou saidas)
5      vector<float> selectors;   // Conexoes especificas, como seletoras
       de mux ou op de sum_sub
6      double prob_0;           // Probabilidade de ocorrer nivel logico 0
7      double prob_1;           // Probabilidade de ocorrer nivel logico 1
8      double carry_out_prob_0; // Probabilidade de carry_out ser 0 (
       somente para sum_sub)
9      double carry_out_prob_1; // Probabilidade de carry_out ser 1 (
       somente para sum_sub)
10 };
```

Código 3.10 – Estrutura das arestas do algoritmo de Transições Probabilísticas.

```
1 1 inpt 16 0 //a
```

```
2 2 inpt 16 0 //b
3 3 inpt 12 0 //sel0
4 4 inpt 7 0 //sel1
5 5 inpt 1 0 //carry_in
6 6 sum_sub 1 4
7   1 2
8   5
9   3
10 7 sum_sub 1 4
11  1 2
12  6.2
13  3
14 8 sum_sub 1 4
15  1 2
16  7.2
17  3
18 ...
19 52 mux 1 3
20  10.1 10.1
21  3
22 53 mux 1 3
23  20 15
24  3
25 54 mux 1 3
26  52 53
27  4
28 ...
```

Código 3.11 – Estrutura das arestas do algoritmo de Transições Probabilísticas.

3.4.2 Cálculo das probabilidades

Após a instanciação de todos os elementos da *netlist*, o `map` de `Element` é enviado para a função `calculateProbabilities`. Nesta função é realizado o cálculo elemento a elemento das probabilidades de saída serem '0' e '1'. Para a realização do cálculo, primeiramente é analisado se cada elemento teve a sua probabilidade calculada, caso não tenha, o elemento e toda a *netlist* são enviados para a função `calculateElementProbability`, onde será identificado o tipo do elemento, quem são as suas conexões de entrada e será realizado o cálculo de acordo com as equações definidas na Figura 16. Esse processo é realizado duas vezes, pois podem ocorrer casos em que as entradas do elemento analisado ainda não tiveram seus valores de probabilidade calculados, assim, caso o algoritmo percorresse a *netlist* apenas uma vez, poderiam ocorrer casos em que alguns elementos não teriam seus valores calculados.

3.4.3 Divergência entre os circuitos

Como última etapa do algoritmo, o programa faz a comparação entre os valores de transição probabilística elemento a elemento do circuito a partir dos caminhos lógicos indo das saídas até as entradas do circuito.

A função `findPathsForOutputs` primeiramente extrai todos os caminhos lógicos possíveis de cada uma das saídas do circuito. Para, em seguida, a função `compareProbabilitiesWithPaths` realizar o percorrimento desses caminhos em busca de divergências como: valores de probabilidades diferentes no mesmo elemento, caminhos lógicos de tamanhos diferentes ou até mesmo saídas que estão presentes em um circuito e não estão presentes em outros.

Essas informações são enviadas para as funções: `displayOutputPaths` que disponibiliza em um arquivo `.txt` todos os caminhos lógicos dos circuitos e os valores de probabilidade de cada elemento, `saveDivergences` que também fornece em um arquivo `.txt` as divergências encontradas na função `compareProbabilitiesWithPaths` e, por fim, a função `saveTransitionProbabilities` que salva também em um arquivo `.txt` as transições probabilísticas de cada um dos elementos das *netlists* analisadas.

Para dispor os resultados, o algoritmo cria uma pasta topo `Results` a qual terá três pastas: `Divergences` (que contém as divergências entre os dois circuitos), `Outputs` (fornece os caminhos lógicos de cada um dos circuitos) e `Table_Transitions` (dispõe as tabelas de transições probabilísticas dos dois circuitos analisados).

A disposição dos resultados de três maneiras diferentes, além de fornecer uma visão completa da análise realizada, ela se completa na limitação que cada método tem. O Código 3.12 apresenta duas *netlists* exemplo. O circuito original é composto apenas por uma `and`, enquanto o circuito modificado tem a `and` substituída por uma `nand`.

```
Netlist Original:
1 inpt 1 0
2 inpt 1 0
3 and 1 2
  1 2
4 out 0 1
  3

Netlist Modificada:
1 inpt 1 0
2 inpt 1 0
3 nand 1 2
  1 2
4 out 0 1
  3
```

Código 3.12 – *Netlists* Exemplo.

A Tabela 5 fornece o resultado da transição probabilística entre os dois circuitos. Nota-se que os valores são iguais mesmo que o circuito tenha sido modificado. Dado que a transição probabilística é a multiplicação da probabilidade da saída do elemento lógico ser '0' pela probabilidade de ser '1', os elementos lógicos complementares (`and` e `nand` , `xor` e `xnor` , `or` e `nor`) apresentam como característica apenas a inversão das equações entre as probabilidades de serem '0' ou '1', como mostra a Figura 16. Dessa forma, o resultado em formato de tabela acusa um falso negativo para a presença de *trojans* no circuito.

Tabela 5 – Transições Probabilísticas Circuito Exemplo 3.12.

Elemento	Circuito Ouro	Circuito Modificado
1	0.0625	0.0625
2	0.0625	0.0625
3	0.0273438	0.0273438
4	0.0273438	0.0273438

Contudo, analisando os circuitos via caminhos lógicos das saídas é possível inferir que houve modificação entre os circuitos e em que elemento específico ocorreu essa modificação, como mostra o Código 3.13 em que o elemento 3 tem os valores das probabilidades de ser '0' e '1' invertidos entre as duas respostas.

```
1  Circuito Original:
2  Output 4:
3    Logical Path 1: 4 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.4375; 1: 0.0625)
       <- 1 (0: 0.25; 1: 0.25)
4    Logical Path 2: 4 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.4375; 1: 0.0625)
       <- 2 (0: 0.25; 1: 0.25)
5
6
7  Circuito Modificado:
8  Output 4:
9    Logical Path 1: 4 (0: 0.0625; 1: 0.4375) <- 3 (0: 0.0625; 1: 0.4375)
       <- 1 (0: 0.25; 1: 0.25)
10   Logical Path 2: 4 (0: 0.0625; 1: 0.4375) <- 3 (0: 0.0625; 1: 0.4375)
       <- 2 (0: 0.25; 1: 0.25)
```

Código 3.13 – Caminhos Lógicos das *Netlists* Exemplo 3.12.

Por fim, para os casos em que o circuito apresente diversos caminhos lógicos e saber qual é o elemento específico que foi modificado não seja um fator determinante, o resultado da análise pode ser visto pelas divergências encontradas, como mostra o Código 3.14 que indica que houve divergência entre as saídas analisadas.

```
1  Divergent Output: Output 4 from Netlist 1 (Prob 0: 0.4375, Prob 1:
       0.0625) diverges from Output 4 from Netlist 2 (Prob 0: 0.0625, Prob
       1: 0.4375).
```

Código 3.14 – Divergências entre as *Netlists* Exemplo 3.12.

4 Resultados

Este capítulo detalha os resultados das simulações realizadas para validar o algoritmo proposto. Os dados apresentados destacam a análise do circuito ouro em relação ao circuito modificado apresentado no Capítulo 3. Todas as *netlists* utilizadas durante as simulações advêm do algoritmo gerador de *netlist* descrito na Seção 3.3 e estão disponíveis no Apêndice A.

4.1 Codificador Gray

A Tabela 6 apresenta a comparação entre as transições probabilísticas do circuito ouro e do circuito modificado. Analisando a tabela, o primeiro fator divergente entre os dois circuitos é a quantidade de elementos entre eles. O circuito original apresenta ao todo 17 elementos (entradas, elementos lógicos e saídas), enquanto o circuito modificado apresenta 27. Esse aumento refere-se aos TH inseridos no circuito. Além disso, os treze primeiros elementos de ambos os circuitos apresentam o mesmo valor de transição probabilística. Porém, no elemento 13 ocorre a primeira divergência indicando que houve uma modificação em relação ao circuito original. Dada a quantidade de elementos lógicos a mais inseridos, a posição das saídas (que estão destacadas em negrito) também foi alterada na tabela. Para o primeiro circuito, as saídas correspondem aos elementos de 14 a 17, enquanto para o circuito modificado, são os elementos de 24 a 27. Em uma comparação entre as saídas, nota-se que apenas a saída 26 apresenta o mesmo valor que a saída 16. As demais estão com os valores alterados, o que é esperado dadas as modificações propostas.

O Código 4.1 fornece os caminhos lógicos das saídas de ambos os circuitos e os valores de probabilidade que cada elemento que compõe o caminho apresenta. Nota-se que os caminhos lógicos das saídas 14, 15 e 17 do circuito ouro divergem em relação às saídas 24, 25 e 27 do circuito modificado no quesito quantidade de caminhos lógicos para cada saída e quantidade de elementos que os compõem. Em contrapartida, assim como visto na Tabela 6, a saída 26 e a saída 16 apresentam o mesmo valor probabilístico, indicando a possibilidade da saída não ter sido modificada. Analisando os caminhos lógicos de ambas as saídas, a quantidade de caminhos, os elementos que a compõem e o valor de probabilidade das saídas serem iguais a ‘0’ ou ‘1’ desses elementos indicam que a saída não foi modificada.

Por fim, o Código 4.2 fornece a última análise do algoritmo o qual indica se existem mais ou menos saídas em um circuito em relação ao outro ou se existem divergências entre os valores das saídas. Como resultado, o algoritmo encontrou divergência entre 3 das 4 saídas que compõem o circuito, o que era esperado dadas as modificações propostas

anteriormente.

Tabela 6 – Transições Probabilísticas Codificador Gray NCL de 4 bits.

Elemento	Circuito Ouro	Circuito Modificado
1	0.0625	0.0625
2	0.0625	0.0625
3	0.0625	0.0625
4	0.0625	0.0625
5	0.0273438	0.0273438
6	0.0273438	0.0273438
7	0.0273438	0.0273438
8	0.0273438	0.0273438
9	0.0273438	0.0273438
10	0.0273438	0.0273438
11	0.0231781	0.0273438
12	0.0231781	0.0273438
13	0.0231781	0.00267029
14	**0.0231781**	0.0231781
15	**0.0231781**	0.0231781
16	**0.0231781**	0.0231781
17	**0.0625**	0.029541
18	-	0.00821584
19	-	0.0149195
20	-	0.0273438
21	-	0.0090332
22	-	0.0273438
23	-	0.0090332
24	-	**0.00821584**
25	-	**0.0149195**
26	-	**0.0231781**
27	-	**0.029541**

```
Circuito Ouro:
Output 14:
  Logical Path 1: 14 (0: 0.191406; 1: 0.121094) <- 11 (0: 0.191406; 1:
    0.121094) <- 5 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.25; 1: 0.25)
  Logical Path 2: 14 (0: 0.191406; 1: 0.121094) <- 11 (0: 0.191406; 1:
    0.121094) <- 5 (0: 0.4375; 1: 0.0625) <- 4 (0: 0.25; 1: 0.25)
  Logical Path 3: 14 (0: 0.191406; 1: 0.121094) <- 11 (0: 0.191406; 1:
    0.121094) <- 6 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.25; 1: 0.25)
  Logical Path 4: 14 (0: 0.191406; 1: 0.121094) <- 11 (0: 0.191406; 1:
    0.121094) <- 6 (0: 0.4375; 1: 0.0625) <- 4 (0: 0.25; 1: 0.25)

Output 15:
  Logical Path 1: 15 (0: 0.191406; 1: 0.121094) <- 12 (0: 0.191406; 1:
    0.121094) <- 7 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
  Logical Path 2: 15 (0: 0.191406; 1: 0.121094) <- 12 (0: 0.191406; 1:
    0.121094) <- 7 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.25; 1: 0.25)
  Logical Path 3: 15 (0: 0.191406; 1: 0.121094) <- 12 (0: 0.191406; 1:
    0.121094) <- 8 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
  Logical Path 4: 15 (0: 0.191406; 1: 0.121094) <- 12 (0: 0.191406; 1:
    0.121094) <- 8 (0: 0.4375; 1: 0.0625) <- 3 (0: 0.25; 1: 0.25)
```

```
13
14 Output 16:
15   Logical Path 1: 16 (0: 0.191406; 1: 0.121094) <- 13 (0: 0.191406; 1:
       0.121094) <- 9 (0: 0.4375; 1: 0.0625) <- 1 (0: 0.25; 1: 0.25)
16   Logical Path 2: 16 (0: 0.191406; 1: 0.121094) <- 13 (0: 0.191406; 1:
       0.121094) <- 9 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
17   Logical Path 3: 16 (0: 0.191406; 1: 0.121094) <- 13 (0: 0.191406; 1:
       0.121094) <- 10 (0: 0.4375; 1: 0.0625) <- 1 (0: 0.25; 1: 0.25)
18   Logical Path 4: 16 (0: 0.191406; 1: 0.121094) <- 13 (0: 0.191406; 1:
       0.121094) <- 10 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
19
20 Output 17:
21   Logical Path 1: 17 (0: 0.25; 1: 0.25) <- 1 (0: 0.25; 1: 0.25)
22
23
24
25   Circuito Modificado:
26   Output 24:
27   Logical Path 1: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 14 (0: 0.191406; 1: 0.121094) <- 5 (0: 0.4375; 1:
       0.0625) <- 3 (0: 0.25; 1: 0.25)
28   Logical Path 2: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 14 (0: 0.191406; 1: 0.121094) <- 5 (0: 0.4375; 1:
       0.0625) <- 4 (0: 0.25; 1: 0.25)
29   Logical Path 3: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 14 (0: 0.191406; 1: 0.121094) <- 6 (0: 0.4375; 1:
       0.0625) <- 3 (0: 0.25; 1: 0.25)
30   Logical Path 4: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 14 (0: 0.191406; 1: 0.121094) <- 6 (0: 0.4375; 1:
       0.0625) <- 4 (0: 0.25; 1: 0.25)
31   Logical Path 5: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 21 (0: 0.578125; 1: 0.015625) <- 3 (0: 0.25; 1: 0.25)
32   Logical Path 6: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 21 (0: 0.578125; 1: 0.015625) <- 20 (0: 0.4375; 1:
       0.0625) <- 1 (0: 0.25; 1: 0.25)
33   Logical Path 7: 24 (0: 0.112549; 1: 0.072998) <- 18 (0: 0.112549; 1:
       0.072998) <- 21 (0: 0.578125; 1: 0.015625) <- 20 (0: 0.4375; 1:
       0.0625) <- 2 (0: 0.25; 1: 0.25)
34
35 Output 25:
36   Logical Path 1: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
       0.134827) <- 15 (0: 0.191406; 1: 0.121094) <- 7 (0: 0.4375; 1:
       0.0625) <- 2 (0: 0.25; 1: 0.25)
37   Logical Path 2: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
       0.134827) <- 15 (0: 0.191406; 1: 0.121094) <- 7 (0: 0.4375; 1:
       0.0625) <- 3 (0: 0.25; 1: 0.25)
38   Logical Path 3: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
```

```
    0.134827) <- 15 (0: 0.191406; 1: 0.121094) <- 8 (0: 0.4375; 1:
    0.0625) <- 2 (0: 0.25; 1: 0.25)
39  Logical Path 4: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
    0.134827) <- 15 (0: 0.191406; 1: 0.121094) <- 8 (0: 0.4375; 1:
    0.0625) <- 3 (0: 0.25; 1: 0.25)
40  Logical Path 5: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
    0.134827) <- 23 (0: 0.578125; 1: 0.015625) <- 3 (0: 0.25; 1: 0.25)
41  Logical Path 6: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
    0.134827) <- 23 (0: 0.578125; 1: 0.015625) <- 22 (0: 0.4375; 1:
    0.0625) <- 1 (0: 0.25; 1: 0.25)
42  Logical Path 7: 25 (0: 0.110657; 1: 0.134827) <- 19 (0: 0.110657; 1:
    0.134827) <- 23 (0: 0.578125; 1: 0.015625) <- 22 (0: 0.4375; 1:
    0.0625) <- 2 (0: 0.25; 1: 0.25)
43
44 Output 26:
45  Logical Path 1: 26 (0: 0.191406; 1: 0.121094) <- 16 (0: 0.191406; 1:
    0.121094) <- 9 (0: 0.4375; 1: 0.0625) <- 1 (0: 0.25; 1: 0.25)
46  Logical Path 2: 26 (0: 0.191406; 1: 0.121094) <- 16 (0: 0.191406; 1:
    0.121094) <- 9 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
47  Logical Path 3: 26 (0: 0.191406; 1: 0.121094) <- 16 (0: 0.191406; 1:
    0.121094) <- 10 (0: 0.4375; 1: 0.0625) <- 1 (0: 0.25; 1: 0.25)
48  Logical Path 4: 26 (0: 0.191406; 1: 0.121094) <- 16 (0: 0.191406; 1:
    0.121094) <- 10 (0: 0.4375; 1: 0.0625) <- 2 (0: 0.25; 1: 0.25)
49
50 Output 27:
51  Logical Path 1: 27 (0: 0.171875; 1: 0.171875) <- 17 (0: 0.171875; 1:
    0.171875) <- 1 (0: 0.25; 1: 0.25)
52  Logical Path 2: 27 (0: 0.171875; 1: 0.171875) <- 17 (0: 0.171875; 1:
    0.171875) <- 13 (0: 0.683594; 1: 0.00390625) <- 11 (0: 0.4375; 1:
    0.0625) <- 1 (0: 0.25; 1: 0.25)
53  Logical Path 3: 27 (0: 0.171875; 1: 0.171875) <- 17 (0: 0.171875; 1:
    0.171875) <- 13 (0: 0.683594; 1: 0.00390625) <- 11 (0: 0.4375; 1:
    0.0625) <- 2 (0: 0.25; 1: 0.25)
54  Logical Path 4: 27 (0: 0.171875; 1: 0.171875) <- 17 (0: 0.171875; 1:
    0.171875) <- 13 (0: 0.683594; 1: 0.00390625) <- 12 (0: 0.4375; 1:
    0.0625) <- 3 (0: 0.25; 1: 0.25)
55  Logical Path 5: 27 (0: 0.171875; 1: 0.171875) <- 17 (0: 0.171875; 1:
    0.171875) <- 13 (0: 0.683594; 1: 0.00390625) <- 12 (0: 0.4375; 1:
    0.0625) <- 4 (0: 0.25; 1: 0.25)
```

Código 4.1 – Caminhos Lógicos das Saídas do Codificador Gray.

```
1 Divergent Output: Output 14 from Netlist 1 (Prob 0: 0.191406, Prob 1:
    0.121094) diverges from Output 24 from Netlist 2 (Prob 0: 0.112549,
    Prob 1: 0.072998).
2
3 ------------------------------------------------------
4
```

```
5 Divergent Output: Output 15 from Netlist 1 (Prob 0: 0.191406, Prob 1:
     0.121094) diverges from Output 25 from Netlist 2 (Prob 0: 0.110657,
     Prob 1: 0.134827).
6
7 ----------------------------------------------------------
8
9 Divergent Output: Output 17 from Netlist 1 (Prob 0: 0.25, Prob 1: 0.25)
     diverges from Output 27 from Netlist 2 (Prob 0: 0.171875, Prob 1:
     0.171875).
10
11 ----------------------------------------------------------
```

Código 4.2 – Divergências entre os circuitos Codificadores Gray Propostos.

4.2 *Barrel Shifter*

A Tabela 7 mostra as transições probabilísticas do circuito *Barrel Shifter* original e modificado. Dadas as modificações propostas na Seção 3.1.1, a coluna referente ao circuito modificado apresenta mais elementos e valores divergentes em relação ao circuito original, o que altera também a posição das saídas (que estão destacadas em negrito), que, no circuito original, iam da posição 46 ao 49 e passaram para 58 a 61.

Como apenas dois *trojans* foram inseridos, os valores divergentes de transição probabilística são notados nas saídas 47 (que corresponde à saída 59 do circuito modificado) e na saída 48 (que corresponde à saída 60 do circuito modificado).

Por fim, o Código 4.3 destaca as divergências encontradas entre as saídas do circuito. Como esperado, dado que os *trojans* inseridos afetam apenas as saídas 1 e 2, somente nas duas foi atestada a mudança.

```
1 Divergent Output: Output 47 from Netlist 1 (Prob 0: 0.147348, Prob 1:
     0.0505893) diverges from Output 59 from Netlist 2 (Prob 0: 0.12926,
     Prob 1: 0.0515247).
2
3 ----------------------------------------------------------
4
5 Divergent Output: Output 48 from Netlist 1 (Prob 0: 0.140178, Prob 1:
     0.0414613) diverges from Output 60 from Netlist 2 (Prob 0: 0.182379,
     Prob 1: 0.0219169).
6
7 ----------------------------------------------------------
```

Código 4.3 – Divergências entre os Circuitos *Barrel Shifter* Propostos.

Tabela 7 – Transições Probabilísticas *Barrel Shifter* 4 bits.

Elemento	**Circuito Ouro**	**Circuito Modificado**
1 - 9	0.0625	0.0625
10	0.0273438	0.0273438
11	0.0119143	0.0119143
12	0.015648	0.015648
13	0.0273438	0.0273438
14	0.0119143	0.0119143
15	0.015648	0.015648
16 - 17	0.0119143	0.0119143
18	0.00923587	0.0092358
19	0.00861408	0.0086140
20	0.0273438	0.0273438
21	0.0139001	0.0139001
22	0.00861408	0.00694118
23	0.0273438	0.0234499
24	0.0139001	0.00916823
...	...	...
46	**0.00923587**	7.89333e-06
47	**0.00745425**	5.34985e-06
48	**0.00581196**	7.91548e-06
49	**0.00487303**	0.0273438
...	-	...
58	-	**0.00923587**
59	-	**0.00666009**
60	-	**0.00399719**
61	-	**0.00487303**

4.3 ULA

Por fim, o último circuito proposto para análise é a ULA. A partir da inserção dos *trojans* destacadas na Seção 3.1.1, oito das nove saídas do circuito original são afetadas como mostra a Tabela 8, que destaca as transições probabilísticas do circuito original e do circuito modificado.

A inserção de mais elementos ocasiona um aumento da tabela entre os circuitos e a alteração na posição das saídas (que estão destacadas em negrito) na tabela, que passaram de 183 a 190 do circuito original para 189 a 196 no circuito modificado.

Por fim, a divergência entre as saídas é vista no Código 4.4 que aponta as alterações em oito das 9 saídas do circuito e os respectivos valores de probabilidade de ‘0’ e ‘1’ para cada uma delas no circuito original e no modificado.

Tabela 8 – Transições Probabilísticas ULA 5 bits.

Elemento	**Circuito Ouro**	**Circuito Modificado**
1 - 5	0.0625	0.0625
6 - 10	0.0273438	0.0273438
11	0.015625	0.0273438
12	0.015625	0.0090332
13	0.015625	0.0273438
14	0.015625	0.0090332
15	0.015625	0.0273438
16	0.0119143	0.0090332
17	0.0090332	0.0273438
18	0.0273438	0.0090332
19	0.0090332	0.010363
20	0.0273438	0.010363
21	0.0090332	0.00681986
22	0.0273438	0.0273438
23	0.0090332	0.0107422
24	0.010363	0.00396729
...	...	...
183	**0.00140665**	0.0107422
184	**0.00253991**	0.00921845
185	**0.00189103**	6.59524e-05
186	**0.0017094**	2.71701e-05
187	**0.00164734**	1.32038e-08
188	**0.00162527**	2.03053e-10
189	**3.08402e-05**	**0.00140653**
190	**3.46994e-10**	**0.00253991**
191	-	**0.00177757**
192	-	**0.00167099**
193	-	**0.00110108**
194	-	**0.00162038**
195	-	**3.06455e-05**
196	-	**2.03053e-10**

```
1 Divergent Output: Output 183 from Netlist 1 (Prob 0: 0.120345, Prob 1:
     0.0116884) diverges from Output 189 from Netlist 2 (Prob 0: 0.120336,
      Prob 1: 0.0116884).
2
3 ----------------------------------------------------------
4
5 Divergent Output: Output 185 from Netlist 1 (Prob 0: 0.124082, Prob 1:
     0.0152402) diverges from Output 191 from Netlist 2 (Prob 0: 0.123533,
      Prob 1: 0.0143895).
6
7 ----------------------------------------------------------
8
9 Divergent Output: Output 186 from Netlist 1 (Prob 0: 0.123314, Prob 1:
     0.0138622) diverges from Output 192 from Netlist 2 (Prob 0: 0.123263,
      Prob 1: 0.0135564).
10
11 ----------------------------------------------------------
12
```

```
13 Divergent Output: Output 187 from Netlist 1 (Prob 0: 0.123242, Prob 1:
      0.0133667) diverges from Output 193 from Netlist 2 (Prob 0: 0.130642,
       Prob 1: 0.00842819).
14
15 ----------------------------------------------------
16
17 Divergent Output: Output 188 from Netlist 1 (Prob 0: 0.123235, Prob 1:
      0.0131884) diverges from Output 194 from Netlist 2 (Prob 0: 0.123235,
       Prob 1: 0.0131487).
18
19 ----------------------------------------------------
20
21 Divergent Output: Output 189 from Netlist 1 (Prob 0: 0.23845, Prob 1:
      0.000129336) diverges from Output 195 from Netlist 2 (Prob 0:
      0.238409, Prob 1: 0.000128542).
22
23 ----------------------------------------------------
24
25 Divergent Output: Output 190 from Netlist 1 (Prob 0: 0.488275, Prob 1:
      7.10652e-10) diverges from Output 196 from Netlist 2 (Prob 0:
      0.492246, Prob 1: 4.12503e-10).
26
27 ----------------------------------------------------
```

Código 4.4 – Divergências entre os Circuitos ULA Propostos.

4.4 Comparação entre o Modelo de Transições Probabilísticas e a Análise via *Testbench*

Esta seção, apresenta a análise comparativa entre o método proposto e a análise via *testbench*.

A escolha do método via *testbench* se deu pela familiaridade com o método e pelas limitações de acesso a outros métodos de detecção.

A Tabela 9 destaca a comparação entre o algoritmo proposto e o método de detecção via *testbench* no quesito cobertura de circuito, tempo médio de execução. Ambas as métricas foram escolhidas para a análise pois em posse de ambos os dados é possível compreender o tempo necessário para que ambos os programas levam para analisar uma determinada quantidade de casos de cada circuito.

Já as métricas de desvio padrão e coeficiente de variação estimam a porcentagem de variabilidade entre as rodadas de testes realizadas para cada um dos algoritmos. Apesar do tempo de análise de cada circuito ser fixo, a variação nos tempos de simulação ocorre dada a presença de outros programas rodando simultaneamente com os algoritmos

Tabela 9 – Comparação entre o Algoritmo Proposto e a Simulação via Testbench.

	Testbench			**Transição Prob.**		
	Cod. Gray	***Barrel Shifter***	***ULA***	***Cod. Gray***	***Barrel Shifter***	***ULA***
T. Médio Exec. (ms)	133.264	610.758	5354.498	71.894	74.743	86.628
Desvio Padrão (ms)	9.359	54.536	147.755	7.772	7.884	8.961
Coef. Variação (%)	7.02	8.93	2.76	10.81	10.55	10.34
Cobertura (%)	100	100	100	100	100	100

de teste, o que implica no manejo do sistema operacional do tempo de recurso que cada um terá e que, consequentemente, afeta o tempo de compilação de cada algoritmo. Diante das variabilidades inerentes ao sistema, o menor coeficiente de variação alcançado experimentalmente foi de 11%. Por conseguinte, esse valor foi estipulado como o limite base para atestar a validade dos tempos médios de simulação em cada um dos métodos avaliados.

Os *scripts* utilizados para coleta das métricas estão disponíveis no Apêndice B. O programa utilizado para coleta das métricas via *testbench* foi o programa Questa - Intel FPGA Starter Edition 2023.3 (Quartus Prime Pro 23.1std).

A Tabela 9 fornece as métricas coletadas para cada um dos circuitos de teste propostos em ambos os métodos de detecção. Para ambos os casos cem por cento do circuito foram analisados, ou seja, todas as possibilidades de dados válidos foram cobertas. Analisando os resultados apresentados por cada um dos métodos, nota-se que o tempo de execução médio do método proposto em relação ao método via *testbench* é menor. Esse comportamento ocorre, haja vista o método de análise de cada um. Como o modelo via *testbench* analisa a resposta do circuito dado um conjunto de valores de entrada, e a cada nova entrada o número de possibilidades cresce 2^N, sendo N o número de entradas, o tempo para validação torna-se maior, quando comparado ao número de elementos lógicos que compõem o circuito como é o caso do método proposto.

5 Conclusão

A busca da indústria por novos circuitos mais eficientes energeticamente, mais rápidos e robustos impulsiona novas pesquisas e alternativas para descrição de circuitos digitais que consigam solucionar parcial ou totalmente as limitações dos circuitos digitais tradicionais.

Os circuitos QDI, com destaque para os circuitos NCL, vêm se demonstrando uma alternativa viável, sendo cada vez mais pesquisados e utilizados pela indústria. Dados o crescente estudo e aplicação desses circuitos, o presente trabalho teve como objetivo compreender como alterações maliciosas poderiam ser realizadas durante a etapa RTL (*Register Transfer Level*) em circuitos NCL combinacionais e como detectá-las utilizando o método de Transições Probabilísticas.

Nesse sentido, o trabalho apresentou os conceitos base sobre os circuitos QDI e NCL, detalhando características como o protocolo de *handshaking*, a completude de entrada e a observabilidade. Também foram discutidos os *trojans* de *hardware*, abordando suas classificações, mecanismos de ativação e carga útil, além de revisar os métodos de detecção já existentes, o que serviu de base para a escolha do método de transições probabilísticas para a análise na fase de pré-silício.

Na parte do desenvolvimento, foi criada um programa em linguagem C++ capaz de ler a descrição dos circuitos a partir dos arquivos de saída `.vo` do Quartus e gerar automaticamente as *netlists*. O algoritmo implementado realiza o cálculo das probabilidades de transição de cada elemento lógico, comparando o circuito original com o modificado. Para validar a proposta, foram descritos em NCL três circuitos de referência — um Codificador Gray, um *Barrel Shifter* e uma ULA — nos quais foram inseridos Trojans combinacionais ativados por casos raros, permitindo verificar se a ferramenta conseguia identificar as divergências nas saídas.

O método analisado mostrou-se eficiente para a verificação dos circuitos infectados, encontrando cem por cento das vezes os *trojans* inseridos. Além disso, em comparação com a verificação formal via *testbench*, o algoritmo de transições probabilísticas foi mais rápido, dada a verificação pelos elementos lógicos e não pelas possibilidades de entradas (tendo em vista que é uma das desvantagens dos circuitos NCL).

O modelo via transições probabilísticas mostrou-se ser um método eficiente de identificação de *trojans* aplicados às condições de dados válidos dos circuitos NCL combinacionais. Dessa forma, o modelo demonstrou ser um complemento aos demais métodos de identificação de th discutidos, aumentando a segurança no processo de fabricação desses circuitos.

Como ações futuras a esse trabalho, tem-se como objetivo a expansão do método para identificação de circuitos sequenciais e a aplicação do método para detecção de TH em *pipelines* NCL, a análise do método frente a outros programas de detecção, além da aplicação do método em outras etapas no projeto dos circuitos digitais.

Referências

ADEE, S. The hunt for the kill switch. **IEEE Spectrum**, v. 45, p. 34–39, 5 2008. ISSN 0018-9235. Disponível em: <http://ieeexplore.ieee.org/document/4505310/>.

CHAKRABORTY, R. S.; NARASIMHAN, S.; BHUNIA, S. Hardware trojan: Threats and emerging solutions. In: **2009 IEEE International High Level Design Validation and Test Workshop**. IEEE, 2009. p. 166–171. ISBN 978-1-4244-4823-4. Disponível em: <http://ieeexplore.ieee.org/document/5340158/>.

FANT, K.; BRANDT, S. Null convention logic/sup tm/: a complete and consistent logic for asynchronous digital circuit synthesis. In: **Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96**. IEEE Computer Soc. Press, 1996. p. 261–273. ISBN 0-8186-7542-X. Disponível em: <http://ieeexplore.ieee.org/document/542821/>.

GUAZZELLI, R. A. et al. Trojan detection test for clockless circuits. **Journal of Electronic Testing: Theory and Applications (JETTA)**, Springer, v. 36, p. 23–31, 2 2020. ISSN 15730727.

GUIMARAES, L. A. et al. Non-intrusive testing technique for detection of trojans in asynchronous circuits. In: **2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. IEEE, 2018. p. 1516–1519. ISBN 978-3-9819263-0-9. Disponível em: <http://ieeexplore.ieee.org/document/8342255/>.

HAYASHI, V. T.; RUGGIERO, W. V. Hardware trojan detection in open-source hardware designs using machine learning. **IEEE Access**, Institute of Electrical and Electronics Engineers Inc., v. 13, p. 37771–37788, 2025. ISSN 21693536.

KARRI, R. et al. Trustworthy hardware: Identifying and classifying hardware trojans. **Computer**, v. 43, p. 39–46, 10 2010. ISSN 0018-9162. Disponível em: <http://ieeexplore.ieee.org/document/5604161/>.

KHODOSEVYCH, D.; SAKIB, A. A. Evolution of null convention logic based asynchronous paradigm: An overview and outlook. **IEEE Access**, Institute of Electrical and Electronics Engineers Inc., v. 10, p. 78650–78666, 2022. ISSN 21693536.

LODHI, F. K. et al. Hardware trojan detection in soft error tolerant macro synchronous micro asynchronous (msma) pipeline. In: **2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)**. IEEE, 2014. p. 659–662. ISBN 978-1-4799-4132-2. Disponível em: <https://ieeexplore.ieee.org/document/6908501>.

_____. Formal analysis of macro synchronous micro asychronous pipeline for hardware trojan detection. In: **2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)**. IEEE, 2015. p. 1–4. ISBN 978-1-4673-6576-5. Disponível em: <http://ieeexplore.ieee.org/document/7364384/>.

MITRA, S.; WONG, H.-S. P.; WONG, S. The trojan-proof chip. **IEEE Spectrum**, v. 52, p. 46–51, 2 2015. ISSN 0018-9235. Disponível em: <http://ieeexplore.ieee.org/document/7024511/>.

MYERS, C. J. **Asynchronous circuit design**. [S.l.]: J. Wiley & Sons, 2001. 404 p. ISBN 9780471415435.

OLIVEIRA, D. L. et al. A novel k convention logic (ncl) gates architecture based on basic gates. In: **2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)**. IEEE, 2017. p. 1–4. ISBN 978-1-5090-6363-5. Disponível em: <http://ieeexplore.ieee.org/document/8079680/>.

OLIVEIRA, D. L. D. et al. Synthesis of qdi combinational circuits using null convention logic based on basic gates. **Advances in Science, Technology and Engineering Systems**, ASTES Publishers, v. 3, p. 308–317, 2018. ISSN 24156698.

PARSAN, F. A.; SMITH, S. C. Cmos implementation comparison of ncl gates. In: **2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)**. [S.l.]: IEEE, 2012. p. 394–397. ISBN 978-1-4673-2527-1.

PILLMEIER, M. R.; SCHULTE, M. J.; III, E. G. W. Design alternatives for barrel shifters. In: LUK, F. T. (Ed.). [S.l.: s.n.], 2002. p. 436.

PIMENTA, T. C. **Circuitos Digitais : Análise e Síntese Lógica: Aplicações em FPGA**. [S.l.]: Elsevier Editora, 2017. ISBN 9788535266030.

PONUGOTI, K. K. et al. Illegal trojan design and detection in asynchronous null convention logic and sleep convention logic circuits. **IET Computers and Digital Techniques**, John Wiley and Sons Inc, v. 16, p. 172–182, 9 2022. ISSN 1751861X.

POPAT, J.; MEHTA, U. Transition probabilistic approach for detection and diagnosis of hardware trojan in combinational circuits. In: **2016 IEEE Annual India Conference (INDICON)**. IEEE, 2016. p. 1–6. ISBN 978-1-5090-3646-2. Disponível em: <http://ieeexplore.ieee.org/document/7838895/>.

SALMANI, H.; TEHRANIPOOR, M.; PLUSQUELLIC, J. A novel technique for improving hardware trojan detection and reducing trojan activation time. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 20, p. 112–125, 1 2012. ISSN 10638210.

SMITH, S. C.; DI, J. **Designing Asynchronous Circuits using NULL Convention Logic (NCL)**. Springer International Publishing, 2009. ISBN 978-3-031-79799-6. Disponível em: <https://link.springer.com/10.1007/978-3-031-79800-9>.

THE International Roadmap for Devices International Roadmap for Devices and Systems 2018 Update Medical Devices Market Driver. [S.l.], 2019.

THE International Technology Roadmap for Semiconductors: 2013 International Technology Roadmap for Semiconductors 2013 Edition Design. [S.l.], 2013.

WOJCICKI, T. **VLSI**. CRC Press, 2014. ISBN 9781315215549. Disponível em: <https://www.taylorfrancis.com/books/9781315215549>.

Apêndices

APÊNDICE A – *Netlists*

Neste Apêndice são dispostos as *netlists* resultantes da geração automática dos circuitos utilizados como teste.

A.1 Netlists

A.1.1 Codificador Gray

A.1.1.1 *Netlist* do circuito ouro

```
1 inpt 3 0 //A
2 inpt 4 0 //B
3 inpt 4 0 //C
4 inpt 2 0 //D
5 and 1 2 //And0
  3 4
6 and 1 2 //And1
  3 4
7 and 1 2 //And2
  2 3
8 and 1 2 //And3
  2 3
9 and 1 2 //And4
  1 2
10 and 1 2 //And5
  1 2
11 or 1 2 //OR0
  5 6
12 or 1 2 //OR1
  7 8
13 or 1 2 //OR2
  9 10
14 out 0 1 //out0
  11
15 out 0 1 //out1
  12
16 out 0 1 //out2
  13
17 out 0 1 //out3
  1
```

Código A.1 – *Netlist* Codificador Gray Ouro

A.1.1.2 *Netlist* do circuito modificado

```
1 1 inpt 6 0 //A
2 2 inpt 7 0 //B
3 3 inpt 7 0 //C
4 4 inpt 3 0 //D
5 5 and 1 2 //And0
6   3 4
7 6 and 1 2 //And1
8   3 4
9 7 and 1 2 //And2
10   2 3
11 8 and 1 2 //And3
12   2 3
13 9 and 1 2 //And4
14   1 2
15 10 and 1 2 //And5
16   1 2
17 11 and 1 2 //And1TrojanOut3
18   1 2
19 12 and 1 2 //And2TrojanOut3
20   3 4
21 13 and 1 2 //And3TrojanOut3
22   11 12
23 14 or 1 2 //OR0
24   5 6
25 15 or 1 2 //OR1
26   7 8
27 16 or 1 2 //OR2
28   9 10
29 17 xor 1 2 //Or2TrojanOut3
30   1 13
31 18 xor 1 2 //OutTrojan0
32   14 21
33 19 or 1 2 //OutTrojan1
34   15 23
35 20 and 1 2 //trojan01
36   1 2
37 21 and 1 2 //trojan02
38   3 20
39 22 and 1 2 //trojan11
40   1 2
41 23 and 1 2 //trojan12
42   3 22
43 24 out 0 1 //out00
44   18
45 25 out 0 1 //out10
46   19
```

```
26 out 0 1 //out20
  16
27 out 0 1 //out30
  17
```

Código A.2 – *Netlist* Codificador Gray Modificado

A.1.2 *Barrel Shifter*

A.1.2.1 *Netlist* do circuito ouro

```
1 inpt 2 0 //A0
2 inpt 2 0 //A1
3 inpt 2 0 //A2
4 inpt 3 0 //A3
5 inpt 6 0 //rotate
6 inpt 8 0 //shift0
7 inpt 8 0 //shift1
8 inpt 2 0 //sra
9 inpt 1 0 //value
10 and 1 2 //MUX_router1_1|g0
  2 5
11 and 1 2 //MUX_router1_1|g1
  5 45
12 or 1 2 //MUX_router1_1|g2
  10 11
13 and 1 2 //MUX_router1_2|g0
  1 5
14 and 1 2 //MUX_router1_2|g1
  5 45
15 or 1 2 //MUX_router1_2|g2
  13 14
16 and 1 2 //MUX_router2_1|g0
  5 30
17 and 1 2 //MUX_router2_1|g1
  5 45
18 or 1 2 //MUX_router2_1|g2
  16 17
19 and 1 2 //MUX_shifter1_1|g0
  7 12
20 and 1 2 //MUX_shifter1_1|g1
  4 7
21 or 2 2 //MUX_shifter1_1|g2
  19 20
22 and 1 2 //MUX_shifter1_2|g0
  7 15
23 and 1 2 //MUX_shifter1_2|g1
  3 7
```

```
24 or 2 2 //MUX_shifter1_2|g2
  22 23
25 and 1 2 //MUX_shifter1_3|g0
  4 7
26 and 1 2 //MUX_shifter1_3|g1
  2 7
27 or 2 2 //MUX_shifter1_3|g2
  25 26
28 and 1 2 //MUX_shifter1_4|g0
  3 7
29 and 1 2 //MUX_shifter1_4|g1
  1 7
30 or 2 2 //MUX_shifter1_4|g2
  28 29
31 and 1 2 //MUX_shifter2_1|g0
  6 18
32 and 1 2 //MUX_shifter2_1|g1
  6 21
33 or 1 2 //MUX_shifter2_1|g2
  31 32
34 and 1 2 //MUX_shifter2_2|g0
  6 21
35 and 1 2 //MUX_shifter2_2|g1
  6 24
36 or 1 2 //MUX_shifter2_2|g2
  34 35
37 and 1 2 //MUX_shifter2_3|g0
  6 24
38 and 1 2 //MUX_shifter2_3|g1
  6 27
39 or 1 2 //MUX_shifter2_3|g2
  37 38
40 and 1 2 //MUX_shifter2_4|g0
  6 27
41 and 1 2 //MUX_shifter2_4|g1
  6 30
42 or 1 2 //MUX_shifter2_4|g2
  40 41
43 and 1 2 //MUX_sra|g0
  4 8
44 and 1 2 //MUX_sra|g1
  8 9
45 or 3 2 //MUX_sra|g2
  43 44
46 out 0 1 //out0
  42
47 out 0 1 //out1
```

```
  39
48 out 0 1 //out2
  36
49 out 0 1 //out3
  33
```

Código A.3 – *Netlist Barrel Shifter* Ouro

A.1.2.2 *Netlist* do circuito modificado

```
1 inpt 2 0 //A0
2 inpt 2 0 //A1
3 inpt 2 0 //A2
4 inpt 3 0 //A3
5 inpt 7 0 //rotate
6 inpt 9 0 //shift0
7 inpt 8 0 //shift1
8 inpt 2 0 //sra
9 inpt 1 0 //value
10 and 1 2 //MUX_router1_1|g0
  2 5
11 and 1 2 //MUX_router1_1|g1
  5 51
12 or 1 2 //MUX_router1_1|g2
  10 11
13 and 1 2 //MUX_router1_2|g0
  1 5
14 and 1 2 //MUX_router1_2|g1
  5 51
15 or 1 2 //MUX_router1_2|g2
  13 14
16 and 1 2 //MUX_router2_1|g0
  5 30
17 and 1 2 //MUX_router2_1|g1
  5 51
18 or 1 2 //MUX_router2_1|g2
  16 17
19 and 1 2 //MUX_shifter1_1|g0
  7 12
20 and 1 2 //MUX_shifter1_1|g1
  4 7
21 or 4 2 //MUX_shifter1_1|g2
  19 20
22 and 1 2 //MUX_shifter1_2|g0
  15 52
23 and 1 2 //MUX_shifter1_2|g1
  3 52
24 or 3 2 //MUX_shifter1_2|g2
```

```
  22 23
25 and 1 2 //MUX_shifter1_3|g0
  4 7
26 and 1 2 //MUX_shifter1_3|g1
  2 7
27 or 3 2 //MUX_shifter1_3|g2
  25 26
28 and 1 2 //MUX_shifter1_4|g0
  3 7
29 and 1 2 //MUX_shifter1_4|g1
  1 7
30 or 3 2 //MUX_shifter1_4|g2
  28 29
31 and 1 2 //MUX_shifter2_1|g0
  6 18
32 and 1 2 //MUX_shifter2_1|g1
  6 21
33 or 1 2 //MUX_shifter2_1|g2
  31 32
34 and 1 2 //MUX_shifter2_2|g0
  6 45
35 and 1 2 //MUX_shifter2_2|g1
  6 24
36 or 1 2 //MUX_shifter2_2|g2
  34 35
37 and 1 2 //MUX_shifter2_3|g0
  6 24
38 and 1 2 //MUX_shifter2_3|g1
  6 27
39 or 1 2 //MUX_shifter2_3|g2
  37 38
40 and 1 2 //MUX_shifter2_4|g0
  6 27
41 and 1 2 //MUX_shifter2_4|g1
  6 30
42 or 1 2 //MUX_shifter2_4|g2
  40 41
43 and 1 2 //MUX_shifter_trojan1|g0
  48 57
44 and 1 2 //MUX_shifter_trojan1|g1
  21 57
45 or 1 2 //MUX_shifter_trojan1|g2
  43 44
46 and 1 2 //MUX_shifter_trojan|g0
  51 57
47 and 1 2 //MUX_shifter_trojan|g1
  21 57
```

```
48 or 1 2 //MUX_shifter_trojan|g2
  46 47
49 and 1 2 //MUX_sra|g0
  4 8
50 and 1 2 //MUX_sra|g1
  8 9
51 or 4 2 //MUX_sra|g2
  49 50
52 or 2 2 //OutTrojan
  7 54
53 and 1 2 //trojan01
  6 7
54 and 1 2 //trojan02
  5 53
55 and 1 2 //trojan03
  21 24
56 and 1 2 //trojan04
  27 30
57 and 4 2 //trojan05
  55 56
58 out 0 1 //out00
  42
59 out 0 1 //out10
  39
60 out 0 1 //out20
  36
61 out 0 1 //out30
  33
```

Código A.4 – *Netlist Barrel Shifter* Modificado

A.1.3 ULA

A.1.3.1 *Netlist* do circuito ouro

```
1 inpt 36 0 //A
2 inpt 36 0 //B
3 inpt 6 0 //CarryIn
4 inpt 44 0 //Sel0
5 inpt 16 0 //Sel1
6 and 1 2 //A0
  1 2
7 and 1 2 //A1
  1 2
8 and 1 2 //A2
  1 2
9 and 1 2 //A3
  1 2
```

```
10 and 2 2 //A4
  1 2
11 xor 1 2 //AX0
  1 2
12 xor 1 2 //AX1
  1 2
13 xor 1 2 //AX2
  1 2
14 xor 1 2 //AX3
  1 2
15 xor 2 2 //AX4
  1 2
16 and 1 2 //Op0|SS0|Gout0
  1 2
17 and 1 2 //Op0|SS0|Gout1
  3 16
18 and 2 2 //Op0|SS0|Gout2
  1 2
19 and 1 2 //Op0|SS0|Gout3
  3 18
20 and 2 2 //Op0|SS0|Gout4
  1 3
21 and 1 2 //Op0|SS0|Gout5
  2 20
22 and 2 2 //Op0|SS0|Gout6
  1 2
23 and 1 2 //Op0|SS0|Gout7
  3 22
24 or 1 2 //Op0|SS0|Gout8
  17 19
25 or 1 2 //Op0|SS0|Gout9
  21 23
26 or 2 2 //Op0|SS0|Gout10
  24 25
27 and 1 2 //Op0|SS0|Gout11
  2 3
28 and 1 2 //Op0|SS0|Gout12
  1 4
29 and 1 2 //Op0|SS0|Gout13
  3 28
30 and 1 2 //Op0|SS0|Gout14
  4 20
31 and 1 2 //Op0|SS0|Gout15
  4 22
32 and 1 2 //Op0|SS0|Gout16
  4 18
33 or 1 2 //Op0|SS0|Gout17
```

```
  27 29
34 or 1 2 //Op0|SS0|Gout18
  30 31
35 or 1 2 //Op0|SS0|Gout19
  32 34
36 or 6 2 //Op0|SS0|Gout20
  33 35
37 and 1 2 //Op0|SS1|Gout0
  1 2
38 and 1 2 //Op0|SS1|Gout1
  36 37
39 and 2 2 //Op0|SS1|Gout2
  1 2
40 and 1 2 //Op0|SS1|Gout3
  36 39
41 and 2 2 //Op0|SS1|Gout4
  1 36
42 and 1 2 //Op0|SS1|Gout5
  2 41
43 and 2 2 //Op0|SS1|Gout6
  1 2
44 and 1 2 //Op0|SS1|Gout7
  36 43
45 or 1 2 //Op0|SS1|Gout8
  38 40
46 or 1 2 //Op0|SS1|Gout9
  42 44
47 or 2 2 //Op0|SS1|Gout10
  45 46
48 and 1 2 //Op0|SS1|Gout11
  2 36
49 and 1 2 //Op0|SS1|Gout12
  1 4
50 and 1 2 //Op0|SS1|Gout13
  36 49
51 and 1 2 //Op0|SS1|Gout14
  4 41
52 and 1 2 //Op0|SS1|Gout15
  4 43
53 and 1 2 //Op0|SS1|Gout16
  4 39
54 or 1 2 //Op0|SS1|Gout17
  48 50
55 or 1 2 //Op0|SS1|Gout18
  51 52
56 or 1 2 //Op0|SS1|Gout19
  53 55
```

```
57 or 6 2 //Op0|SS1|Gout20
  54 56
58 and 1 2 //Op0|SS2|Gout0
  1 2
59 and 1 2 //Op0|SS2|Gout1
  57 58
60 and 2 2 //Op0|SS2|Gout2
  1 2
61 and 1 2 //Op0|SS2|Gout3
  57 60
62 and 2 2 //Op0|SS2|Gout4
  1 57
63 and 1 2 //Op0|SS2|Gout5
  2 62
64 and 2 2 //Op0|SS2|Gout6
  1 2
65 and 1 2 //Op0|SS2|Gout7
  57 64
66 or 1 2 //Op0|SS2|Gout8
  59 61
67 or 1 2 //Op0|SS2|Gout9
  63 65
68 or 2 2 //Op0|SS2|Gout10
  66 67
69 and 1 2 //Op0|SS2|Gout11
  2 57
70 and 1 2 //Op0|SS2|Gout12
  1 4
71 and 1 2 //Op0|SS2|Gout13
  57 70
72 and 1 2 //Op0|SS2|Gout14
  4 62
73 and 1 2 //Op0|SS2|Gout15
  4 64
74 and 1 2 //Op0|SS2|Gout16
  4 60
75 or 1 2 //Op0|SS2|Gout17
  69 71
76 or 1 2 //Op0|SS2|Gout18
  72 73
77 or 1 2 //Op0|SS2|Gout19
  74 76
78 or 6 2 //Op0|SS2|Gout20
  75 77
79 and 1 2 //Op0|SS3|Gout0
  1 2
80 and 1 2 //Op0|SS3|Gout1
```

```
155   78 79
156 81 and 2 2 //Op0|SS3|Gout2
157   1 2
158 82 and 1 2 //Op0|SS3|Gout3
159   78 81
160 83 and 2 2 //Op0|SS3|Gout4
161   1 78
162 84 and 1 2 //Op0|SS3|Gout5
163   2 83
164 85 and 2 2 //Op0|SS3|Gout6
165   1 2
166 86 and 1 2 //Op0|SS3|Gout7
167   78 85
168 87 or 1 2 //Op0|SS3|Gout8
169   80 82
170 88 or 1 2 //Op0|SS3|Gout9
171   84 86
172 89 or 2 2 //Op0|SS3|Gout10
173   87 88
174 90 and 1 2 //Op0|SS3|Gout11
175   2 78
176 91 and 1 2 //Op0|SS3|Gout12
177   1 4
178 92 and 1 2 //Op0|SS3|Gout13
179   78 91
180 93 and 1 2 //Op0|SS3|Gout14
181   4 83
182 94 and 1 2 //Op0|SS3|Gout15
183   4 85
184 95 and 1 2 //Op0|SS3|Gout16
185   4 81
186 96 or 1 2 //Op0|SS3|Gout17
187   90 92
188 97 or 1 2 //Op0|SS3|Gout18
189   93 94
190 98 or 1 2 //Op0|SS3|Gout19
191   95 97
192 99 or 4 2 //Op0|SS3|Gout20
193   96 98
194 100 and 1 2 //Op0|SS4|Gout0
195   1 2
196 101 and 1 2 //Op0|SS4|Gout1
197   99 100
198 102 and 1 2 //Op0|SS4|Gout2
199   1 2
200 103 and 1 2 //Op0|SS4|Gout3
201   99 102
```

```
104 and 1 2 //Op0|SS4|Gout4
  1 99
105 and 1 2 //Op0|SS4|Gout5
  2 104
106 and 1 2 //Op0|SS4|Gout6
  1 2
107 and 1 2 //Op0|SS4|Gout7
  99 106
108 or 1 2 //Op0|SS4|Gout8
  101 103
109 or 1 2 //Op0|SS4|Gout9
  105 107
110 or 4 2 //Op0|SS4|Gout10
  108 109
111 and 1 2 //muxOut0|Mux0|gMUX0
  4 26
112 and 1 2 //muxOut0|Mux0|gMUX1
  4 26
113 or 1 2 //muxOut0|Mux0|gMUX2
  111 112
114 and 1 2 //muxOut0|Mux1|gMUX0
  4 6
115 and 1 2 //muxOut0|Mux1|gMUX1
  4 11
116 or 1 2 //muxOut0|Mux1|gMUX2
  114 115
117 and 1 2 //muxOut0|Mux2|gMUX0
  5 116
118 and 1 2 //muxOut0|Mux2|gMUX1
  5 113
119 or 2 2 //muxOut0|Mux2|gMUX2
  117 118
120 and 1 2 //muxOut1|Mux0|gMUX0
  4 47
121 and 1 2 //muxOut1|Mux0|gMUX1
  4 47
122 or 1 2 //muxOut1|Mux0|gMUX2
  120 121
123 and 1 2 //muxOut1|Mux1|gMUX0
  4 7
124 and 1 2 //muxOut1|Mux1|gMUX1
  4 12
125 or 1 2 //muxOut1|Mux1|gMUX2
  123 124
126 and 1 2 //muxOut1|Mux2|gMUX0
  5 125
127 and 1 2 //muxOut1|Mux2|gMUX1
```

```
249   5 122
250 128 or 2 2 //muxOut1|Mux2|gMUX2
251   126 127
252 129 and 1 2 //muxOut2|Mux0|gMUX0
253   4 68
254 130 and 1 2 //muxOut2|Mux0|gMUX1
255   4 68
256 131 or 1 2 //muxOut2|Mux0|gMUX2
257   129 130
258 132 and 1 2 //muxOut2|Mux1|gMUX0
259   4 8
260 133 and 1 2 //muxOut2|Mux1|gMUX1
261   4 13
262 134 or 1 2 //muxOut2|Mux1|gMUX2
263   132 133
264 135 and 1 2 //muxOut2|Mux2|gMUX0
265   5 134
266 136 and 1 2 //muxOut2|Mux2|gMUX1
267   5 131
268 137 or 2 2 //muxOut2|Mux2|gMUX2
269   135 136
270 138 and 1 2 //muxOut3|Mux0|gMUX0
271   4 89
272 139 and 1 2 //muxOut3|Mux0|gMUX1
273   4 89
274 140 or 1 2 //muxOut3|Mux0|gMUX2
275   138 139
276 141 and 1 2 //muxOut3|Mux1|gMUX0
277   4 9
278 142 and 1 2 //muxOut3|Mux1|gMUX1
279   4 14
280 143 or 1 2 //muxOut3|Mux1|gMUX2
281   141 142
282 144 and 1 2 //muxOut3|Mux2|gMUX0
283   5 143
284 145 and 1 2 //muxOut3|Mux2|gMUX1
285   5 140
286 146 or 2 2 //muxOut3|Mux2|gMUX2
287   144 145
288 147 and 1 2 //muxOut4|Mux0|gMUX0
289   4 110
290 148 and 1 2 //muxOut4|Mux0|gMUX1
291   4 110
292 149 or 1 2 //muxOut4|Mux0|gMUX2
293   147 148
294 150 and 1 2 //muxOut4|Mux1|gMUX0
295   4 10
```

```
151 and 1 2 //muxOut4|Mux1|gMUX1
  4 15
152 or 1 2 //muxOut4|Mux1|gMUX2
  150 151
153 and 1 2 //muxOut4|Mux2|gMUX0
  5 152
154 and 1 2 //muxOut4|Mux2|gMUX1
  5 149
155 or 2 2 //muxOut4|Mux2|gMUX2
  153 154
156 or 1 2 //negDetector|g9
  161 163
157 or 1 2 //negDetector|g10
  165 167
158 or 1 2 //negDetector|g11
  156 157
159 and 2 2 //negDetector|gNeg0
  110 175
160 and 1 2 //negDetector|gNeg1
  4 5
161 and 1 2 //negDetector|gNeg2
  159 160
162 and 1 2 //negDetector|gNeg3
  4 5
163 and 1 2 //negDetector|gNeg4
  159 162
164 and 1 2 //negDetector|gNeg5
  4 5
165 and 1 2 //negDetector|gNeg6
  15 164
166 and 1 2 //negDetector|gNeg7
  4 5
167 and 1 2 //negDetector|gNeg8
  10 166
168 xor 1 2 //over|gOver1
  1 178
169 xor 1 2 //over|gOver2
  1 110
170 and 2 2 //over|gOver3
  168 169
171 and 1 2 //over|gOver4
  4 5
172 and 1 2 //over|gOver5
  170 171
173 and 1 2 //over|gOver6
  4 5
174 and 1 2 //over|gOver7
```

```
343   170 173
344 175 or 2 2 //over|gOver8
345   172 174
346 176 and 1 2 //over|gOver0|gMUX0
347   2 4
348 177 and 1 2 //over|gOver0|gMUX1
349   2 4
350 178 or 1 2 //over|gOver0|gMUX2
351   176 177
352 179 and 1 2 //zero|gZero0
353   119 128
354 180 and 1 2 //zero|gZero1
355   137 146
356 181 and 1 2 //zero|gZero2
357   179 180
358 182 and 1 2 //zero|gZero3
359   155 181
360 183 out 0 1 //Neg0
361   158
362 184 out 0 1 //Out0
363   119
364 185 out 0 1 //Out1
365   128
366 186 out 0 1 //Out2
367   137
368 187 out 0 1 //Out3
369   146
370 188 out 0 1 //Out4
371   155
372 189 out 0 1 //Overflow0
373   175
374 190 out 0 1 //Zero0
375   182
```

Código A.5 – *Netlist* ULA Ouro

A.1.3.2 *Netlist* do circuito modificado

```
1 1 inpt 37 0 //A
2 2 inpt 37 0 //B
3 3 inpt 9 0 //CarryIn
4 4 inpt 42 0 //Sel0
5 5 inpt 16 0 //Sel1
6 6 and 1 2 //A0
7   1 2
8 7 and 1 2 //A1
9   1 2
10 8 and 1 2 //A2
```

```
  1 2
9 and 1 2 //A3
  1 2
10 and 2 2 //A4
  1 2
11 and 1 2 //Op0|SS0|Gout0
  1 2
12 and 1 2 //Op0|SS0|Gout1
  3 11
13 and 2 2 //Op0|SS0|Gout2
  1 2
14 and 1 2 //Op0|SS0|Gout3
  3 13
15 and 2 2 //Op0|SS0|Gout4
  1 3
16 and 1 2 //Op0|SS0|Gout5
  2 15
17 and 2 2 //Op0|SS0|Gout6
  1 2
18 and 1 2 //Op0|SS0|Gout7
  3 17
19 or 1 2 //Op0|SS0|Gout8
  12 14
20 or 1 2 //Op0|SS0|Gout9
  16 18
21 or 2 2 //Op0|SS0|Gout10
  19 20
22 and 1 2 //Op0|SS0|Gout11
  2 3
23 and 1 2 //Op0|SS0|Gout12
  1 107
24 and 1 2 //Op0|SS0|Gout13
  3 23
25 and 1 2 //Op0|SS0|Gout14
  15 107
26 and 1 2 //Op0|SS0|Gout15
  17 107
27 and 1 2 //Op0|SS0|Gout16
  13 107
28 or 1 2 //Op0|SS0|Gout17
  22 24
29 or 1 2 //Op0|SS0|Gout18
  25 26
30 or 1 2 //Op0|SS0|Gout19
  27 29
31 or 6 2 //Op0|SS0|Gout20
  28 30
```

```
32 and 1 2 //Op0|SS1|Gout0
  1 2
33 and 1 2 //Op0|SS1|Gout1
  31 32
34 and 2 2 //Op0|SS1|Gout2
  1 2
35 and 1 2 //Op0|SS1|Gout3
  31 34
36 and 2 2 //Op0|SS1|Gout4
  1 31
37 and 1 2 //Op0|SS1|Gout5
  2 36
38 and 2 2 //Op0|SS1|Gout6
  1 2
39 and 1 2 //Op0|SS1|Gout7
  31 38
40 or 1 2 //Op0|SS1|Gout8
  33 35
41 or 1 2 //Op0|SS1|Gout9
  37 39
42 or 2 2 //Op0|SS1|Gout10
  40 41
43 and 1 2 //Op0|SS1|Gout11
  2 31
44 and 1 2 //Op0|SS1|Gout12
  1 4
45 and 1 2 //Op0|SS1|Gout13
  31 44
46 and 1 2 //Op0|SS1|Gout14
  4 36
47 and 1 2 //Op0|SS1|Gout15
  4 38
48 and 1 2 //Op0|SS1|Gout16
  4 34
49 or 1 2 //Op0|SS1|Gout17
  43 45
50 or 1 2 //Op0|SS1|Gout18
  46 47
51 or 1 2 //Op0|SS1|Gout19
  48 50
52 or 6 2 //Op0|SS1|Gout20
  49 51
53 and 1 2 //Op0|SS2|Gout0
  1 2
54 and 1 2 //Op0|SS2|Gout1
  52 53
55 and 2 2 //Op0|SS2|Gout2
```

```
105   1 2
106 56 and 1 2 //Op0|SS2|Gout3
107   52 55
108 57 and 2 2 //Op0|SS2|Gout4
109   1 52
110 58 and 1 2 //Op0|SS2|Gout5
111   2 57
112 59 and 2 2 //Op0|SS2|Gout6
113   1 2
114 60 and 1 2 //Op0|SS2|Gout7
115   52 59
116 61 or 1 2 //Op0|SS2|Gout8
117   54 56
118 62 or 1 2 //Op0|SS2|Gout9
119   58 60
120 63 or 2 2 //Op0|SS2|Gout10
121   61 62
122 64 and 1 2 //Op0|SS2|Gout11
123   2 52
124 65 and 1 2 //Op0|SS2|Gout12
125   1 4
126 66 and 1 2 //Op0|SS2|Gout13
127   52 65
128 67 and 1 2 //Op0|SS2|Gout14
129   4 57
130 68 and 1 2 //Op0|SS2|Gout15
131   4 59
132 69 and 1 2 //Op0|SS2|Gout16
133   4 55
134 70 or 1 2 //Op0|SS2|Gout17
135   64 66
136 71 or 1 2 //Op0|SS2|Gout18
137   67 68
138 72 or 1 2 //Op0|SS2|Gout19
139   69 71
140 73 or 6 2 //Op0|SS2|Gout20
141   70 72
142 74 and 1 2 //Op0|SS3|Gout0
143   1 2
144 75 and 1 2 //Op0|SS3|Gout1
145   73 74
146 76 and 2 2 //Op0|SS3|Gout2
147   1 2
148 77 and 1 2 //Op0|SS3|Gout3
149   73 76
150 78 and 2 2 //Op0|SS3|Gout4
151   1 73
```

```
79 and 1 2 //Op0|SS3|Gout5
  2 78
80 and 2 2 //Op0|SS3|Gout6
  1 2
81 and 1 2 //Op0|SS3|Gout7
  73 80
82 or 1 2 //Op0|SS3|Gout8
  75 77
83 or 1 2 //Op0|SS3|Gout9
  79 81
84 or 2 2 //Op0|SS3|Gout10
  82 83
85 and 1 2 //Op0|SS3|Gout11
  2 73
86 and 1 2 //Op0|SS3|Gout12
  1 4
87 and 1 2 //Op0|SS3|Gout13
  73 86
88 and 1 2 //Op0|SS3|Gout14
  4 78
89 and 1 2 //Op0|SS3|Gout15
  4 80
90 and 1 2 //Op0|SS3|Gout16
  4 76
91 or 1 2 //Op0|SS3|Gout17
  85 87
92 or 1 2 //Op0|SS3|Gout18
  88 89
93 or 1 2 //Op0|SS3|Gout19
  90 92
94 or 4 2 //Op0|SS3|Gout20
  91 93
95 and 1 2 //Op0|SS4|Gout0
  1 2
96 and 1 2 //Op0|SS4|Gout1
  94 95
97 and 1 2 //Op0|SS4|Gout2
  1 2
98 and 1 2 //Op0|SS4|Gout3
  94 97
99 and 1 2 //Op0|SS4|Gout4
  1 94
100 and 1 2 //Op0|SS4|Gout5
  2 99
101 and 1 2 //Op0|SS4|Gout6
  1 2
102 and 1 2 //Op0|SS4|Gout7
```

```
199   94 101
200 103 or 1 2 //Op0|SS4|Gout8
201   96 98
202 104 or 1 2 //Op0|SS4|Gout9
203   100 102
204 105 or 4 2 //Op0|SS4|Gout10
205   103 104
206 106 and 1 2 //Op0|Trojan0
207   3 4
208 107 xor 4 2 //Op0|Trojan2
209   4 106
210 108 xor 1 2 //X0
211   1 2
212 109 xor 1 2 //X1
213   1 2
214 110 xor 1 2 //X2
215   1 2
216 111 xor 1 2 //X3
217   1 2
218 112 xor 2 2 //X4
219   1 2
220 113 and 1 2 //muxOut0|Mux0|g0
221   4 21
222 114 and 1 2 //muxOut0|Mux0|g1
223   4 21
224 115 or 1 2 //muxOut0|Mux0|g2
225   113 114
226 116 and 1 2 //muxOut0|Mux1|g0
227   4 6
228 117 and 1 2 //muxOut0|Mux1|g1
229   4 108
230 118 or 1 2 //muxOut0|Mux1|g2
231   116 117
232 119 and 1 2 //muxOut0|Mux2|g0
233   5 118
234 120 and 1 2 //muxOut0|Mux2|g1
235   5 115
236 121 or 2 2 //muxOut0|Mux2|g2
237   119 120
238 122 and 1 2 //muxOut1|Mux0|g0
239   4 42
240 123 and 1 2 //muxOut1|Mux0|g1
241   4 42
242 124 or 1 2 //muxOut1|Mux0|g2
243   122 123
244 125 and 1 2 //muxOut1|Mux1|g0
245   4 7
```

```
126 and 1 2 //muxOut1|Mux1|g1
  4 109
127 or 1 2 //muxOut1|Mux1|g2
  125 126
128 and 1 2 //muxOut1|Mux2|g0
  5 127
129 and 1 2 //muxOut1|Mux2|g1
  5 124
130 or 2 2 //muxOut1|Mux2|g2
  128 129
131 and 1 2 //muxOut2|Mux0|g0
  4 63
132 and 1 2 //muxOut2|Mux0|g1
  4 63
133 or 1 2 //muxOut2|Mux0|g2
  131 132
134 and 1 2 //muxOut2|Mux1|g0
  4 8
135 and 1 2 //muxOut2|Mux1|g1
  4 110
136 or 1 2 //muxOut2|Mux1|g2
  134 135
137 and 1 2 //muxOut2|Mux2|g0
  5 136
138 and 1 2 //muxOut2|Mux2|g1
  5 133
139 or 2 2 //muxOut2|Mux2|g2
  137 138
140 and 1 2 //muxOut3|Mux0|g0
  4 84
141 and 1 2 //muxOut3|Mux0|g1
  4 84
142 or 1 2 //muxOut3|Mux0|g2
  140 141
143 and 1 2 //muxOut3|Mux1|g0
  4 9
144 and 1 2 //muxOut3|Mux1|g1
  4 184
145 or 1 2 //muxOut3|Mux1|g2
  143 144
146 and 1 2 //muxOut3|Mux2|g0
  5 145
147 and 1 2 //muxOut3|Mux2|g1
  5 142
148 or 2 2 //muxOut3|Mux2|g2
  146 147
149 and 1 2 //muxOut4|Mux0|g0
```

```
293   4 105
294 150 and 1 2 //muxOut4|Mux0|g1
295   4 105
296 151 or 1 2 //muxOut4|Mux0|g2
297   149 150
298 152 and 1 2 //muxOut4|Mux1|g0
299   4 10
300 153 and 1 2 //muxOut4|Mux1|g1
301   4 112
302 154 or 1 2 //muxOut4|Mux1|g2
303   152 153
304 155 and 1 2 //muxOut4|Mux2|g0
305   5 154
306 156 and 1 2 //muxOut4|Mux2|g1
307   5 151
308 157 or 2 2 //muxOut4|Mux2|g2
309   155 156
310 158 and 1 2 //negDetector|g0
311   4 5
312 159 and 1 2 //negDetector|g1
313   158 169
314 160 and 1 2 //negDetector|g2
315   4 5
316 161 and 1 2 //negDetector|g3
317   160 169
318 162 and 1 2 //negDetector|g4
319   4 5
320 163 and 1 2 //negDetector|g5
321   112 162
322 164 and 1 2 //negDetector|g6
323   4 5
324 165 and 1 2 //negDetector|g7
325   10 164
326 166 or 1 2 //negDetector|g8
327   159 161
328 167 or 1 2 //negDetector|g9
329   163 165
330 168 or 1 2 //negDetector|g10
331   166 167
332 169 and 2 2 //negDetector|gNeg
333   105 177
334 170 and 1 2 //over|g0
335   4 5
336 171 and 1 2 //over|g2
337   4 5
338 172 xor 1 2 //over|over0
339   1 180
```

```
340 173 xor 1 2 //over|over1
341   1 105
342 174 and 2 2 //over|over2
343   172 173
344 175 and 1 2 //over|over3
345   170 174
346 176 and 1 2 //over|over4
347   171 174
348 177 or 2 2 //over|over5
349   175 176
350 178 and 1 2 //over|selSignalB|g0
351   2 4
352 179 and 1 2 //over|selSignalB|g1
353   2 4
354 180 or 1 2 //over|selSignalB|g2
355   178 179
356 181 and 1 2 //trojan1
357   1 2
358 182 and 1 2 //trojan0|g0
359   3 181
360 183 and 1 2 //trojan0|g1
361   3 111
362 184 or 1 2 //trojan0|g2
363   182 183
364 185 and 1 2 //zero|g0
365   121 130
366 186 and 1 2 //zero|g1
367   139 148
368 187 and 1 2 //zero|g2
369   185 186
370 188 and 1 2 //zero|g3
371   157 187
372 189 out 0 1 //Neg0
373   168
374 190 out 0 1 //Out0
375   121
376 191 out 0 1 //Out1
377   130
378 192 out 0 1 //Out2
379   139
380 193 out 0 1 //Out3
381   148
382 194 out 0 1 //Out4
383   157
384 195 out 0 1 //Overflow0
385   177
386 196 out 0 1 //Zero0
```

```
387 188
```

Código A.6 – *Netlist* ULA Modificada

APÊNDICE B – *Scripts*

Neste Apêndice são dispostos as *scripts* utilizados para extração das métricas de tempo de compilação, desvio padrão e afins.

B.1 *Script* Transições Probabilísticas

Para executar o *script* fornecido no Código B.1, crie um arquivo do tipo `Windows PowerShell Script` na mesma pasta em que está contido o arquivo `.cpp` do programa. A execução do B.1 deve ser realizada após a geração do arquivo `.exe` gerado pela compilação do algoritmo.

```
# --- 1. CONFIGURACAO ---
$num_runs = 100
$cpp_executable = "arquivo .exe gerado apos a compilacao do programa"

# --- 2. VERIFICACAO ---
if (-not (Test-Path $cpp_executable)) {
    Write-Host "Erro: Executavel '$cpp_executable' nao encontrado." -ForegroundColor Red
    Write-Host "Por favor, compile o programa C++ primeiro." -ForegroundColor Yellow
    exit
}

Write-Host "INFO: Iniciando o teste de performance com $num_runs execucoes..."
Write-Host "------------------------------------------------------------"

# --- 3. EXECUCAO E COLETA DE DADOS ---
$run_times_seconds = New-Object System.Collections.Generic.List[double]

for ($i = 1; $i -le $num_runs; $i++) {
    $stopwatch = [System.Diagnostics.Stopwatch]::StartNew()
    & $cpp_executable | Out-Null
    $stopwatch.Stop()

    $current_run_time_sec = $stopwatch.Elapsed.TotalSeconds
    $run_times_seconds.Add($current_run_time_sec)

    # Exibe o tempo da iteracao tambem em microssegundos.
    $current_run_time_us = $current_run_time_sec * 1000000
```

```
28      Write-Host ("INFO: Execucao {0} de {1} finalizada em {2:N2}
        microsegundos." -f $i, $num_runs, $current_run_time_us)
29  }
30
31  # --- 4. CALCULO E RESULTADO FINAL ---
32  Write-Host "------------------------------------------------------"
33  Write-Host "INFO: Todas as execucoes foram finalizadas. Calculando
        estatisticas..."
34
35  # --- Calculos em segundos ---
36  $total_time_sec = ($run_times_seconds | Measure-Object -Sum).Sum
37  $average_time_sec = $total_time_sec / $num_runs
38
39  $sum_of_squared_diffs = 0.0
40  foreach ($time in $run_times_seconds) {
41      $sum_of_squared_diffs += [Math]::Pow($time - $average_time_sec, 2)
42  }
43  $variance = $sum_of_squared_diffs / $num_runs
44  $std_dev_sec = [Math]::Sqrt($variance)
45
46  # Conversao dos resultados finais para microssegundos
47  $average_time_us = $average_time_sec * 1000000
48  $std_dev_us = $std_dev_sec * 1000000
49
50  # --- Calculo do Coeficiente de Variacao (nao muda, pois a unidade e
        cancelada) ---
51  $cv_percentage = 0.0
52  if ($average_time_sec -gt 0) {
53      $cv_percentage = ($std_dev_sec / $average_time_sec) * 100
54  }
55
56  # --- Exibicao dos Resultados ---
57  Write-Host " "
58  Write-Host "================== RESULTADO FINAL ==================="
59  Write-Host ("Numero de Execucoes:       {0}" -f $num_runs)
60  # ALTERADO: Exibe os valores em microssegundos
61  Write-Host ("Tempo Medio de Execucao:   {0:N2} microsegundos" -f
        $average_time_us)
62  Write-Host ("Desvio Padrao:             {0:N2} microsegundos" -f
        $std_dev_us)
63  Write-Host ("Coeficiente de Variacao:   {0:N2} %" -f $cv_percentage)
64  Write-Host "======================================================"
```

Código B.1 – *Script* para extração de métricas do algoritmo de Transições Probabilísticas

B.2 *Script* Transições Probabilísticas

Para utilização do *script* descrito no Código B.2, abra o circuito no Questa (ou Modelsim) e cole-o. A abertura do circuito pode ser realizada tanto via Quartus (a partir da simulação RTL) ou diretamente na ferramenta de simulação.

```
# ==========================================================================

# Script para executar a simulacao varias vezes e calcular

# o tempo medio, desvio padrao e coeficiente de variacao.

# ==========================================================================

# --- 1. CONFIGURACAO ---

# Defina o numero de vezes que a simulacao deve ser executada

set num_runs 200

# --- 2. INICIALIZACAO ---

set total_microseconds 0.0
set run_times_list [list]

# Mensagem inicial

puts "INFO: Iniciando o teste de performance com $num_runs execucoes..."
puts "-------------------------------------------------------------"

# --- 3. LACO DE EXECUCAO ---

# Loop que repetira o processo 'num_runs' vezes

for {set i 1} {$i <= $num_runs} {incr i} {

# Reinicia a simulacao para o tempo 0.
restart -f

# Executa a simulacao e mede o tempo
set time_output [time {run -all}]

# Extrai apenas o valor numerico da string
set current_run_time [lindex $time_output 0]

# Adiciona o tempo da execucao atual a nossa lista
lappend run_times_list $current_run_time

```

```
43 # Adiciona o tempo da execucao atual ao totalizador
44 set total_microseconds [expr {$total_microseconds + $current_run_time}]
45
46 # Mostra o resultado da iteracao atual
47 puts "INFO: Execucao $i de $num_runs finalizada em $current_run_time
       microsegundos."
48
49
50 }
51
52 --- 4. CALCULO E RESULTADO FINAL ---
53
54 puts "------------------------------------------------------"
55 puts "INFO: Todas as execucoes foram finalizadas."
56
57 --- Calculo da Media ---
58
59 set average_time [expr {double($total_microseconds) / $num_runs}]
60
61 --- Calculo do Desvio Padrao ---
62
63 set sum_of_squared_diffs 0.0
64
65 foreach time_value $run_times_list {
66 set difference [expr {$time_value - $average_time}]
67 set sum_of_squared_diffs [expr {$sum_of_squared_diffs + ($difference *
       $difference)}]
68 }
69
70 set variance [expr {$sum_of_squared_diffs / $num_runs}]
71 set std_deviation [expr {sqrt($variance)}]
72
73 --- Calculo do Coeficiente de Variacao (CV) ---
74
75 set cv_percentage 0.0
76
77 Evita divisao por zero caso o tempo medio seja 0
78
79 if {$average_time > 0} {
80 set cv_percentage [expr {(double($std_deviation) / $average_time) *
       100.0}]
81 }
82
83 --- Exibicao dos Resultados ---
84
85 puts " "
86 puts "================== RESULTADO FINAL ==================="
```

```
87 puts "Numero de Execucoes:        $num_runs"
88 puts "Tempo Medio de Execucao:    [format "%.2f" $average_time]
      microsegundos"
89 puts "Desvio Padrao:              [format "%.2f" $std_deviation]
      microsegundos"
90
91 ADICIONADO: Exibe o resultado do Coeficiente de Variacao
92
93 puts "Coeficiente de Variacao:    [format "%.2f" $cv_percentage] %"
94 puts "================================================="
95
96 O comando 'format "%.2f"' formata o numero para exibir apenas 2 casas
      decimais.
```

Código B.2 – *Script* para extração de métricas dos circuitos simulados via Questa