TRABALHO DE GRADUAÇÃO

Análise de Physical Unclonable Functions baseadas em osciladores em anel em FPGA

Gabriel Vinícius Trevisan

Brasília, Julho de 2014

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

Análise de Physical Unclonable Functions baseadas em osciladores em anel em FPGA

Gabriel Vinícius Trevisan

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro Eletricista

Banca Examinadora

Prof. José Edil Guimarães Medeiros, ENE/UnB _____
Orientador

Prof. Sandro Augusto Pavlik Haddad, ENE/UnB _____
FGA/UnB

Msc. Heider Marconi, ENE/UnB _____
DFChip

Agradecimentos

Gostaria de agradecer a todos, encarnados ou não, que me ajudaram, direta ou indiretamente com esse trabalho. Gostaria de agradecer à minha família pelo apoio e aos meus amigos pela paciência. Gostaria de agradecer especialmente ao meu orientador, prof. Edil pelo apoio e ao Heider e ao prof. Sandro pelas sugestões ao meu trabalho.

Gabriel Vinícius Trevisan

RESUMO

O presente texto apresenta a implementação de uma Physical Unclonable Function (PUF) baseada em osciladores em anel em FPGA. Neste trabalho, é verificado como as saídas do circuito variam com o tempo e como essas respostas variam ao se utilizar componentes diferentes dentro da mesma FPGA. O trabalho mostra que caso os mesmos componentes sejam usados, o erro absoluto entre duas respostas em instantes de tempo diferentes não passa de 1%. Porém caso diferentes componentes da mesma FPGA sejam usados, o erro absoluto entre duas respostas que usam componentes diferentes chega a ser maior do que 20%. Esse resultado indica que além de osciladores em anel serem componentes promissores na fabricação de PUFs, ele indica que FPGAs são boas plataformas para abrigar tais PUFs, pois a unicidade da PUF depende não só da FPGA, mas também da região na qual o circuito foi mapeado. Pode-se sugerir em trabalhos futuros a verificação de tais resultados para uma população mais ampla de FPGAs em condições diferentes de temperatura e tensão de alimentação.

ABSTRACT

The following text introduces an FPGA based Physical Unclonable Function implementation. This paper shows the PUF outputs for both same bitstream configuration file, and different bitstream configuration file. These outputs are then compared and it can be seen that absolute errors from same bitstream PUFs are less than 1%, while absolute errors from PUFs using different bitstreams, even though they dwell in the same FPGA, can be higher than 20%. This result indicates that not only ring oscillator are promissing devices in PUF design, but FPGAs are also interesting platforms to host ring oscillator PUFs. For future work, different temperature and power supply conditions should be investigated, along with tests in a wider population of FPGAs.

SUMÁRIO

LISTA DE FIGURAS

Capítulo 1

Introdução

1.1 Contextualização

Com a portabilização e redução de custos de equipamentos eletrônicos, o conceito de dispositivos pessoais começou a surgir, inicialmente com os computadores. Estes computadores, com a universalização da internet, passaram a realizar funções para aumentar a comodidade de seus usuários, como por exemplo o correio eletrônico, acesso a sistemas bancários, transmissão de arquivos etc. Com isso, uma nova questão surge: como saber se uma pessoa é quem ela diz ser através de um computador? Ou seja, como levar o reconhecimento de assinaturas e rostos, que por anos compuzeram os protocolos de segurança de bancos, por exemplo, para o mundo digital, garantindo que as pessoas possam ser identificadas quando necessário?

Nesse contexto, muitas soluções foram propostas para se assegurar identidades, entre elas, podemos citar a autenticação de operações por senhas e por certificados instalados nos equipamentos do cliente. Porém, um atacante que tivesse informações sobre a senha ou sobre o certificado instalado no equipamento poderia, sem a necessidade de possuir o equipamento físico em si, impersonar o usuário em suas operações. Com isso, seria muito útil a existência de uma maneira de identificar o dispositivo que o cliente está usando sem que fosse possível que um atacante pudesse forjar tal dispositivo.

Assim, nesse contexto de identificação do usuário, surgem as funções fisicamente inclonáveis, ou PUFs (Physical Unclonable Functions). Esse tipo de dispositivo faz uso de pequenas variações dos processos de fabricação para criar entidades que funcionam de forma análoga às impressões digitais em pessoas. Assim, mesmo que uma PUF sofra uma tentativa de clonagem utilizando o mesmo design e o mesmo método de fabricação, cada PUF será, idealmente, única.

As PUFs podem ser entendidas como um tipo de função matemática. Às entradas, damos o nome de desafios e as saídas denominamos respostas. As entradas são definidas pelo usuário, porém as saídas são geradas de acordo com parâmetros intrínsecos da PUF e sua saída não pode, idealmente, ser prevista com base na entrada ou pelo menos deve ter uma modelagem complexa o suficiente para que o atacante não tenha todas as informações necessárias para predizer a resposta com precisão.

Cada PUF, idealmente, apresenta respostas diferentes para os mesmos desafios, ou seja, os pares desafio-resposta, *challenge-response pairs (CRP)* são únicos. Assim, conhecendo-se todas as respostas para todos os desafios, conhece-se a PUF. Por esta razão diz-se que uma PUF é definida pelo seu conjunto de CRPs.

1.2 Definição do problema

As PUFs utilizam parâmetros aleatórios da fabricação de dispositivos que podem ser elétricos ou não. Dentre os parâmetros que podem ser explorados em sistemas elétricos, mais especificamente digitais, temos o atraso de portas. Ele tem componentes aleatórios pois depende das capacitâncias parasitas, mobilidade de portadores de carga e outros parâmetros de difícil avaliação precisa para todos os dispositivos.

Similarmente a medir-se o atraso de portas, pode ser medir a frequência de um oscilador em anel, pois essa frequência é baseada nos mesmos parâmetros desses atrasos. Assim, o problema abordado nesse trabalho é a verificação da possibilidade do uso de osciladores em anel em utilizações em PUFs em FPGAs (Field Programmable Gate Arrays), que são dispositivos reconfiguráveis que implementam circuitos digitais.

1.3 Objetivos do projeto

Os objetivos deste projeto são:

- Criar uma implementação de uma PUF baseadas em osciladores em FPGAs para futuros estudos. Essa implementação é composta por um conjunto de arquivos que incluem:
  - Códigos em VHDL dos componentes;
  - Testbenches para a simulação dos componentes;
  - Arquivos do Signal Tap para verificação do arquivo na FGPA;
  - Diagramas esquemáticos dos blocos que foram testados diretamente em FPGA.
- Analisar os dados obtidos da implementação e avaliar seu uso como PUF.

1.4 Apresentação do manuscrito

No capítulo 2 é feita uma revisão bibliográfica sobre o tema de estudo. Em seguida, o capítulo 3 descreve a metodologia empregada no desenvolvimento do projeto. O desenvolvimento da implementação é discutido em 4. Resultados experimentais são discutidos no capítulo 5, seguido das conclusões no capítulo 6. Os anexos contém material complementar.

Capítulo 2

Revisão Bibliográfica

2.1 PUFs

As PUFs são dispositivos físicos que podem ser entendidos como uma função matemática. As entradas dessa função, chamadas de desafios, são definidas pelo usuário, porém as saídas, chamadas de respostas, dependem da estrutura física da PUF. Por sua vez, a estrura física da PUF sofre variações aleatórias em seu processo de fabricação, fazendo com que a resposta da PUF também varie aleatóriamente com o processo de fabricação da PUF.

Uma vez que a resposta da PUF varia com a estrutura física da própria PUF e essa estrutura física varia com o processo de fabricação, não se pode garantir que duas PUFs, mesmo que tenham sido criadas da mesma forma e a partir do mesmo projeto, se comportem da mesma maneira. Dependendo do projeto da PUF, a saída é mais ou menos sensível ao processo de fabricação e pode variar mais ou menos de acordo com o processo de fabricação, portanto pode ser mais ou menos provável que se obtenha duas PUFs que funcionem de forma idêntica a partir do mesmo processo. Em outras palavras, cada projeto de PUF é capaz de gerar PUFs mais ou menos únicas.

Digamos que haja um esquema de verificação de PUFs dado da seguinte forma: Possuímos as PUFs A e B, porém não sabemos a princípio qual é a PUF "A"e qual é a PUF "B", mas conhecemos os CRPs de ambas as PUFs. Lançamos, assim, um desafio para uma das PUFs esperando que essa seja a PUF "A"e analisamos a resposta. Caso o a resposta de "A"e de "B"sejam parecidas o suficiente para o mecanismo de autenticação, é possível que ele identifique a PUF "A"como sendo a PUF "B", e assim temos a validação de um falso-positivo. Dessa forma, quanto maior a diferença entre as respostas de duas PUFs, mais improvável é a autenticação de falso-positivos e melhor é a PUF quanto à unicidade. A medida de quão parecidas duas respostas de **PUFs diferentes** ao **mesmo desafio** é chamada de *distânca inter-classe* (μ_{inter}). A Figura 2.1 exemplifica o comportamento inter-classe das PUFs.

De forma semelhante, é interessante analisar o número de falso-negativos autenticados, ou seja, seguindo o exemplo acima, o mecanismo de autenticação gera um desafio esperando que a PUF escolhida seja a PUF "A". A PUF "A"então recebe o desafio e gera a resposta, porém a resposta não é interpretada pelo mecanismo de autenticação como sendo da PUF "A". Esse tipo

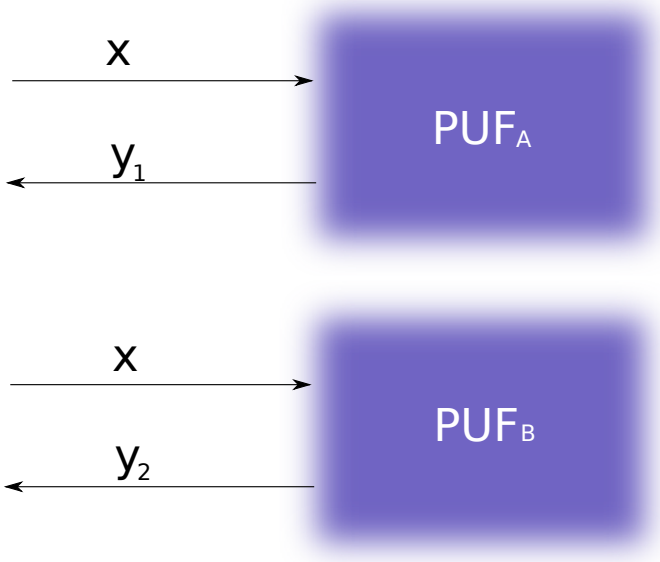


Figura 2.1: Sistema com duas PUFs distintas A e B. Espera-se que as respostas de ambas as PUFs sejam o mais distintas possível.

de erro é causado por por fatores externos como temperatura ambiente e tensão de alimentação [1] ou variações no processo de medição, como o encontrado em PUFs ópticas [2]. Analogamente à distância inter-classe, quanto mais próximas forem as respostas da mesma PUF em diferentes instantes de tempo, menor a probabilidade de se obter um falso-negativo. Assim, denominamos a medida de similaridade entre respostas para o mesmo desafio de uma **mesma PUF** em **instantes de tempo diferentes** de *distância intra-classe* (μ_{intra}). A Figura 2.2 demonstra o comportamento intra-classe das PUFs.

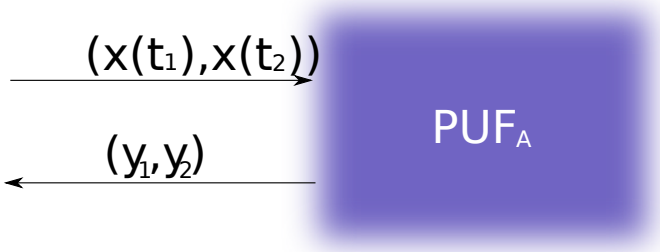


Figura 2.2: Sistema com uma PUF. Espera-se que as saídas para instantes de tempo diferentes sejam o mais próximas possível

Uma das métricas mais comuns para se avaliar as distâncias intra-classe e inter-classe é apresentada em [1], que é um trabalho que busca reunir e analisar várias formas de implementação de PUFs, é a distância de Hamming. A distância de Hamming é uma medida que conta a quantidade de bits diferentes em duas sequências binárias e no caso das PUFs, essas sequências de bits são as saídas das PUFs. Assim, digamos que a sequência A seja $A = 010010$ e a sequência B seja 010100, a distância de Hamming entre essas duas sequências será igual a 2 bits, pois só o quarto e

o quinto bit da sequencia são diferentes. Essa distância também pode ser dada de forma relativa dividindo-se esse valor pelo tamanho da sequência. Dessa forma, usando o exemplo anterior, temos uma distância de Hamming de 2 bits ou 33.3%. Se as sequências binárias avaliadas com a distância de Hamming forem respostas geradas pelo **mesmo desafio na mesma PUF**, o valor da média de suas distâncias é a *distância intra-classe* da PUF. Se as sequências forem as respostas geradas **pelo mesmo desafio porém em PUFs diferentes**, a média dessas distâncias será a *distância inter-classe*. Vale ressaltar que, quando se trata de distâncias inter-classes, o melhor resultado é quando metade da quantidade total de bits é trocada, ou seja, uma distância de Hamming de 50%. Isso pode parecer contra-intuitivo, porém caso a distância de Hamming seja igual a 100%, todos os bits são diferentes, bastando invertê-los todos e assim obter a sequência de bits desejada.

A distância de Hamming, porém, nem sempre é a melhor métrica para se medir distâncias intra e inter-classe. Imagine uma situação em que uma PUF seja formada de osciladores cujas frequências variam com seu processo de fabricação e a saída desta PUF é o valor em base binária da frequência desses osciladores. Suponha, então, que tenhamos três ociladores A, B e C que possuem valores de frequência em base decimal da seguinte forma: a frequência de $A = 15MHz$, a frequência de $B = 14MHz$ e a frequência de $C = 7MHz$. Em base binária, teremos $A = 1111_2MHz$, $B = 1110_2MHz$ e $C = 0111_2MHz$. A distância de Hamming entre A e B e entre A e C são ambas iguais a 1 bit, porém as diferenças entre as frequências são iguais a 1MHz e 8MHz. Neste caso, uma outra métrica é necessária, ou então, uma codificação binária que permita o uso da distância de Hamming.

A distância inter-classe indica quão diferente um clone de uma PUF é da PUF original, dado que o clone seja produzido pelo mesmo processo de fabricação da PUF original. Porém, caso alguém mal-intencionado (aqui referido por atacante), ao invés de tentar produzir uma nova PUF, tente supor as respostas de uma PUF, seria útil uma avaliação do quão difícil seria para ele se passar pelo cliente dessa forma. Para isso, trazemos da Teoria da Informação a noção de entropia de Shannon. A entropia de Shannon mede o grau de incerteza de uma variável aleatória baseado na distribuição de probabilidade desta variável aleatória. Quando se trata de PUFs, quanto maior a entropia de uma PUF, maior a incerteza que se tem sobre sua saída. Imaginemos, por exemplo, duas PUFs A e B cuja saída é apenas um bit. Devido ao processo de fabricação de A, sua saída tem 50% de chance de ser 1 e 50% de chance de ser 0. A PUF B, por sua vez, tem 80% da sua saída ser 1 e 20% de ser 0. Dessa forma, se o atacante quiser predizer a resposta da PUF A, ele sempre terá 50% de chance de acertar, porém caso ele tente predizer a saída da PUF B, ele poderá ter 80% de chance de acertar. Dessa forma, a entropia de saída é uma característica importante em PUFs e há trabalhos que visam calcular sua entropia de forma teórica [3].

Outra característica importante nas PUFs é que PUFs sejam *tamper evidents*, ou seja, que caso haja uma violação (*tampering*) na PUF, tal violação faça com que a PUF não se comporte da forma habitual [1].

2.2 Implementações de PUFs

As PUFs podem ser baseadas em inúmeros fenômenos. Um resumo dessas topologias foi feito em [1] e em [2]. Os tipos mais relevantes para o entendimento desse trabalho são apresentados nesta seção.

É importante que vários projetos de PUF sejam estudados, pois ainda não há um estado da arte bem definido para PUFs e já surgiram trabalhos que dizem ser possível clonar PUFs baseadas em SRAM, um tipo que parecia muito promissor [4].

2.2.1 PUFs não eletrônicas

Aqui descreveremos PUFs cujo o funcionamento é baseado em parâmetros não elétricos. PUFs não eletrônicas, em geral, tem um pouco mais de dificuldade de serem integradas, sendo o termo integrável entendido por PUFs cujo equipamento de medida possa ser incluído no mesmo circuito do circuito que gera o desafio desse dispostivo, ou seja, não necessitando de equipamentos externos.

Um dos conceitos que podem ser utilizados é o da PUF óptica, que teve seu surgimento com padrões de reflexão ópticos [2]. Já PUFs baseadas em meios transparentes foram propostas em [5] e utilizam o padrão de espalhamento de pequenos token ópticos para seu funcionamento. Esses tokens são atingidos por um laser e esse laser será espalhado de acordo com a estrutura física do token. Esse padrão é captado por uma câmera e o sinal é o processado digitalmente. Essa implementação, porém, possui alto custo e tem sua medição complicada pois mesmo rotações de um sessenta-avos de grau causava um padrão de refração totalmente diferente [2], nos dando uma distância intra-classe de algo próximo de $\mu_{intra} = 25.25\%$[1]. Similarmente, temos a PUF de papel [6], a qual tem o seu funcionamento baseado no escaneamento da estrutura caótica das fibras em um papel.

Já a construção do chamado RF-DNA foi introduzida na patente [7] e utiliza fios de cobre finos em um token. Em seguida é verificado o espalhamento de campo próximo quando o campo é submetido a ondas na faixa entre 5 e 6GHz. A entropia dessa topologia é bem alta, cerca de 50000 bits no mínimo, e para sua medição, é necessário o uso de uma matriz de antenas de RF.

Por fim, temos as PUFs baseadas em mídias de armazanamento, como a PUF magnética [8], que utiliza a unicidade intrínseca das fitas ou tarjas magnéticas e as PUFs de CD [9], que fazem uso das variações de comprimento de vales em um CD, que mesmo sendo pequenas, são detectadas pelo leitor. As PUFs de CD podem chegar a uma distância interclasse de $\mu_{inter} = 54\%$ porém sua distância intra-classe chega próxima de $\mu_{intra} = 8\%$[1].

2.2.2 PUFs eletrônicas

Nessa seção, os princípio de funcionamento das PUFs levam em consideração características elétricas, porém analógicas. Um dos exemplos mais simples é a PUF de V_t [10], a qual tem o seguinte funcionamento: uma série de transistores é criada identicamente e de forma endereçável.

Em seguida, um transistor é selecionado e este transistor é posto para fornecer corrente para uma carga resistiva e, devido a processos aleatórios da fabricação do dispositivo, a corrente será parcialmente aleatória. A tensão sobre a carga é então medida e transformada num trem de bits.

Similarmente, temos a PUF de potência [11], cuja proposta foi baseada nas variações de resistência na rede de um chip. Quedas de tensão e resistências equivalentes são mensuradas na distribuição de potência utilizando instrumentos externos e é observado novamente que esses parâmetros elétricos são afetados por processos aleatórios em sua manufatura.

Temos também um tipo de PUF que utiliza como base a capacitância gerada por capacitores em formato de pente cobertos por um spray de substâncias dielétricas [12]. Essa estratégia, além de utilizar a variação intrínseca na manufatura do dispositivo, ainda introduz outra parcela de aleatoriedade ao pulverizar uma camada dielétrica sobre os pentes. Tal método também torna evidente a adulteração da PUF, uma vez que alteração na camada dielétrica geraria um acúmulo de carga diferente, ou seja, uma capacitância diferente.

Para finalizar as PUFs analógicas, temos as PUFs LC [13], que consistem em um filtro LC utilizando um pequeno prato de vidro com uma placa de metal em cada lado, formando um capacitor, conectado em serie com uma bobina de metal. A idéia é medir a quantidade de energia absorvida pelo circuito quando uma fonte de RF faz uma varredura na frequencia.

2.2.2.1 PUFs baseadas em atraso

As PUFs baseadas em atraso são circuitos que utilizam o atraso de portas lógicas em seu funcionamento. As PUFs baseadas nesses processos são principalmente as PUFs de osciladores e as PUFs de árbitro.

Para a PUF de árbitro [14], a idéia é gerar uma condição de corrida entre dois caminhos, teoricamente idênticos, e um circuito no fim que decida qual dos dois caminhos é mais rápido. Para a introdução dos desafios nesse tipo de PUF, é usado um bloco de circuito chamado switchblock, que são blocos parecidos com interruptores four-way que irão alterar os caminhos do circuito. Para entendermos melhor, digamos que cada switchblock tenha duas entradas: in_a e in_b e as saídas out_a e out_b. Caso o valor da variável de controle do switchbox seja zero, temos a entrada in_a conectada à saída out_a e a entrada in_b conectada à saída out_b. Caso o valor da variável de controle mude para 1, in_a se conectará com a saída out_b e a entrada in_b se conectará à entrada out_a. Dessa forma, colocando várias switchboxes em série, o trem de bits usado nas switchbox é o desafio e o resultado da corrida é a resposta.

Dentro do conceito de PUF de atraso, temos também as PUFs baseadas em osciladores em anel. Esse tipo de PUF usa como característica principal a frequência de um oscillador em anel. Uma vez conhecendo os atrasos na linha realimentada do oscilador em anel, sabemos sua frequência e vice versa. Portanto, devido a pequenas diferenças na construção das portas na linha de realimentação, teremos frequências diferentes em cada oscilador. Mais detalhes sobre esta implementação serão discutidos na seção 2.3.

2.2.2.2 PUFs baseadas em memórias

Existem, também, PUFs baseadas em memórias. Tais PUFs são baseadas no estado no qual um elemento de memória tende a se estabilizar. Em outras palavras, quando um dispositivo de memória é ligado, suas células de memória se estabilizam em um estado aleatório, porém esses estados possuem uma certa preferência, e essa preferência varia de acordo com a forma com que a memória é construída.

O problema desse tipo de PUF foi demonstrado em [4] onde, com a ajuda de equipamentos de laboratório amplamente disponíveis, conseguiu-se ler os valores de preferência de blocos de SRAM, reduzindo a intensidade de pesquisas com esse tipo de topologia.

2.3 PUFs baseadas em osciladores em anel

PUFs baseadas em osciladores em anel, ou *ring oscillators* (ROs) foram propostas em [15]. Elas são um tipo de PUF baseada em atraso pois o valor de suas saídas varia com o atraso do circuito. Denominamos de circuitos de atraso os elementos do circuito responsáveis por atrasar a saída com relação à entrada. A saída dos circuito de atraso é realimentada em sua entrada. Isso irá fazer com que o sinal lógico da saída tenha valor contrário ao da saída após um determinado instante de tempo. Assim, cada oscilador irá oscilar em uma frequência igual ao inverso do atraso total do circuito.

Osciladores em anel podem ser facilmente implementados com portas digitais. Isso pode ser obtido conectando-se um número ímpar de portas inversoras em série e conectando sua saída em sua entrada. Dessa forma, a saída dessa ligação irá mudar de valor lógico após o tempo correspondente ao atraso de todas as inversoras. Uma vez que a saída está conectada à entrada, esse processo irá se repetir, fazendo o circuito oscilar. Na Figura 2.3, temos uma representação comum de um RO baseado em portas inversoras.

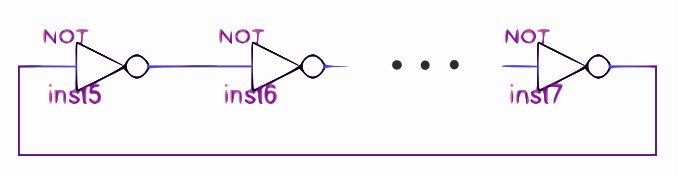
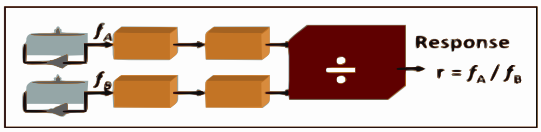


Figura 2.3: Diagrama esquemático de um oscilador em anel

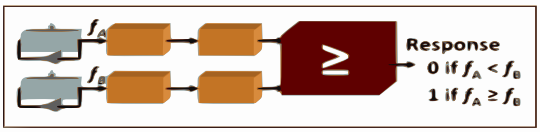
Uma das questões dessa implementação é a variação de frequência que os osciladores sofrem naturalmente devido à variações de temperatura, tensão de alimentação e outros parâmetros externos e portanto, se pensou em maneiras de se compensar esse inconveniente. Uma das formas foi utilizando uma divisão entre os valores obtidos de frequência [15]. Nessa implementação, dois circuitos de atraso são escolhidos para serem usados no oscilador. Suas frequências são amostradas e o resultado mostrado é a razão entre as duas frequências. Isso é útil nos casos em que a frequência dos osciladores varie de forma linear com a temperatura e/ou tensão de alimentação, fazendo com que a razão entre duas frequências seja um valor fixo. Ainda de acordo com [15] o

valor obtido para as distâncias inter-classe e intra-classe com esses métodos são $\mu_{inter} \approx 10 \cdot 10^{-3}$ e $\mu_{intra} \approx 0.1 \cdot 10^{-3}$[1].

Um método mais simples apresentado por [16] é baseado na comparação de frequências de dois osciladores diferentes. Nessa implementação, os osciladores da PUF são divididos em 2 grupos e a frequência de dois osciladores, um de cada grupo, é amostrada e em seguida comparada seguindo um teste lógico, que pode ser, por exemplo, que a saída seja 0 caso a frequência do oscilador A seja maior do que a do oscilador B e que a saída seja 1 em caso contrário. Utilizando esse mecanismo de compensação e o método chamado de "1-out-of-8 masking", que a cada 8 comparações entre osciladores, apenas 1 é utilizada na saída da PUF, foram obtidas distâncias intra e inter-classe de 0.48% e 46.15% [1]. Na Figura 2.4, temos os dois exemplos compensação citados em [15] e [16]



(a) Ring oscillator PUF with division compensation.



(b) Ring oscillator PUF with comparator compensation.

Figura 2.4: Esquemas de compensação de ruído. [1]

Na implementação de [16], embora as distâncias intra-classe e inter-classe possuam valores próximos de 0 e 50%, respectivamente, a implementação tem como saída apenas 1 bit por oscilador. Além disso, a implementação utiliza o que é chamado de "1-out-of-8 masking", que descarta 7 bits a cada 8, fazendo com que o número de bits utilizável na PUF diminua ainda mais. Além disso, a PUF é dividida em dois grupos de osciladores que serão comparados entre si. Dessa forma, dado um certo número de osciladores, apenas metade deles será utilizada na resposta da PUF, em seguida dessas respostas, apenas 1 em 8 será utilizada e cada resposta terá apenas 1 bit. Isso faz esse modelo de PUF ineficiente em área, embora seja de fácil implementação. Portanto esse trabalho é inspirado nessa implementação, porém visa reduzir o problema de entropia de saída.

Capítulo 3

Metodologia

Este capítulo descreve detalhes de como a implementação foi feita e analisada. Ele descreve como foi feita a implementação e a sequência de medição e análise dos resultados do trabalho.

3.1 Setup experimental

A implementação dessa PUF baseada em osciladores em anel foi feita em uma FPGA (Field Programmable Gate Array). FPGAs são dispositivos reconfiguráveis que podem ter suas conexões internas e as chamadas look-up tables configuradas. Assim, hardware de diferentes tipos podem ser realizados em uma FPGA.

A FPGA utilizada no projeto é a FPGA Cyclone II da Altera. A placa na qual a FPGA é montada é o modelo Cyclone II DSP Development Board. Nessa placa, a entrada JTAG é conectada em uma porta USB do computador utilizando o cabo Byteblaster, que permite a configuração da FPGA por um computador, que roda o sistema operacional Ubuntu 12.04 LTS. Para fazer a comunicação com a placa, o software Altera Quartus II Web Edition vs 13.0sp1 foi utilizado. O Quartus II inclui o editor de código, o software que se comunica com a FPGA e a programa e módulos como o SignalTap II, que permite o mapeamento de um analisador lógico dentro da FPGA e o Chip Planner, que mostra as regiões da FPGA que foram utilizadas e permite mover alguns blocos lógicos manualmente. Na Figura 3.1 mostramos uma representação esquemática da ligação.

Como pode ser observado na Figura 3.1, as setas que ligam os componenetes são bidirecionais. Isso é devido ao fluxo de bits serem divididos em dois caminhos: do computador para a FPGA e da FPGA para o computador, ambos passando pela conexão JTAG. O fluxo do computador para a FPGA é o fluxo de programação da FPGA, que inclui o design da PUF e o analisador lógico (SignalTap II). Uma vez tendo a FPGA programada, temos o fluxo da FPGA para o computador. Nesse fluxo de bits, os dados gerados pela PUF são capturados pelo analisador lógico que foi mapeado dentro da FPGA pelo SignalTap II, que envia esses dados também pela interface JTAG para a entrada USB do computador, onde são processados.

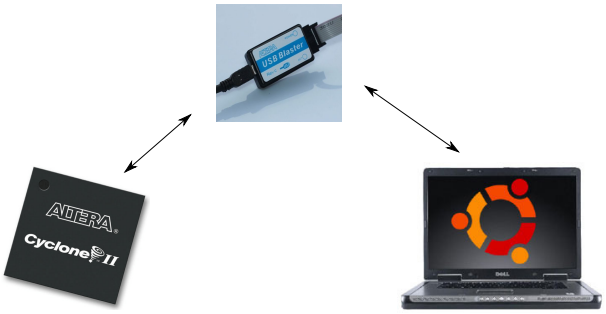


Figura 3.1: Diagrama esquemático do setup experimental do trabalho

3.2 Sequência de medição

A medição das saídas da PUF foi feita seguindo o seguinte fluxo:

- O analisador lógico espera que uma determinada condição de gatilho seja atingida de forma que a PUF possa ser resetada. Essas condições serão discutidas no capítulo 4.
- O primeiro oscilador é selecionado.
- A frequência desse oscilador é medida 'n' vezes. No caso desse experimento, foram escolhidas 42 medições por oscilador.
- Após o mesmo oscilador ter sido medido 'n' vezes, o próximo oscilador é selecionado e também é medido 'n' vezes.
- Esse processo se repete até que todos os 'm' osciladores tenham suas frequências medidas.
- Os dados são armazenados num vetor no computador. Ou seja, cada medição de frequência, independente do oscilador, são adicionadas num vetor de números.

A Figura 3.2 mostra a sequência de medição. Os dois quadrados externos representam um laço de repetição, similares aos laços do programa Labview. Dentro do quadrado menor, temos uma representação de um oscilador, que é o círculo com um ciclo de senoide dentro dele e um medidor de frequência. As letras 'L' e 'K' no canto inferior direito do oscilador e do medidor de frequência mostram a qual laço de repetição cada componente está associado. Assim, 'n' medidas de frequência serão feitas para cada um dos 'm' osciladores, e cada uma delas é armazenada em um vetor numérico no computador.

A esse fluxo de medição daremos o nome nesse trabalho de sequência de medição. Assim, quando é dito que foi feita uma sequência de medição em uma PUF, o que se quer dizer é que a frequência de todos os osciladores foi medida de acordo com o procedimento descrito acima.

3.2.1 Análise dos dados

Diversas sequências de medição foram feitas nesse trabalho, com PUFs de configuração diferentes e os dados, após obtidos, foram processados utilizando-se um código em Python vs 2.7.3,

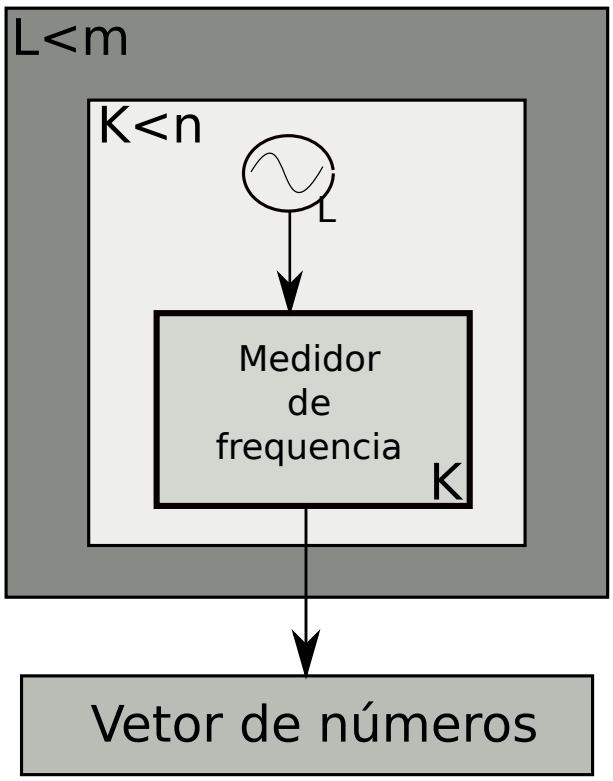


Figura 3.2: Esquemático de uma sequência de medição

o qual prepara os dados para a apresentação em gráficos. O código necessita de ajustes manuais dependendo dos dados que se deseja. Esse código faz o seguinte procedimento para cada sequência de medição:

- Lê o arquivo no qual os dados foram gravados,
- Separa os dados correspondentes para cada oscilador, ou seja, ele informa qual medida de frequência corresponde a qual oscilador
- Calcula a média e o desvio padrão da frequência de cada oscilador nessa sequência.
- Após obter os valores médios das frequências de cada oscilador em uma sequência de medição, ele compara o valor das **médias das frequências** para os mesmos osciladores em sequências de medição diferentes.

Assim, os objetivos dessas medições são ver o quanto a medição de frequência de um oscilador varia em medições diferentes, ver o quão diferente as médias das frequências de vários osciladores implementados da mesma forma são entre si e em seguida, ver o quanto as médias das frequências dos osciladores variam para sequências de medição diferentes.

Capítulo 4

Desenvolvimento

4.1 Implementação da PUF

A PUF implementada contém um banco de osciladores cujas frequências serão medidas. A implementação escolhida foi implementação de um banco de 256 osciladores com 5 e posteriormente 10 células de atraso que são habilitados conforme a necessidade de medição e é inspirada em [17] e [18], porém não se utiliza a compensação por comparação de duas frequências. A entrada dessa PUF é o número do oscilador no qual se quer medir a frequência porém a saída é a string binária que corresponde à frequência desse oscilador escolhido. As saídas, porém, não serão idênticas mesmo que os osciladores tenham sido projetados de forma idêntica, pois as pequenas variações internas da FPGA farão com que cada oscilador tenha uma frequência individual.

Para a implementação da PUF em si, Figura 4.1, dividiremos esta em duas componentes: uma chamada subPUF e uma outra que chamaremos de estimador. A subPUF é o banco de osciladores e é a componente responsável por habilitar e selecionar o oscilador cuja frequência será medida. Sua entrada é uma sequência de bits correspondente ao oscilador desejado. Esse oscilador é então habilitado e seu sinal é levado para a saída da subPUF, que então o leva para a entrada do estimador.

O estimador, por sua vez, recebe o sinal do oscilador da subPUF e avalia a sua frequência em relação ao clock interno da FPGA. Essa frequência é então levada à sua saída, que então é levada à saída da PUF.

Dessa forma, para o circuito descrito, dado um desafio (oscilador), temos uma resposta (frequência do oscilador). Os parâmetros que regulam a frequência dos osciladores, porém, variam de acordo com os componentes internos do oscilador e variam conforme a FPGA. Dessa forma, para o mesmo projeto de osciladores, espera-se ter osciladores diferentes em FPGAs diferentes. Esse tipo de comportamento onde dispositivos diferentes geram saídas diferentes dado o mesmo projeto é promissor na fabricação de PUFs. Esse tipo de PUF é citado em [2].

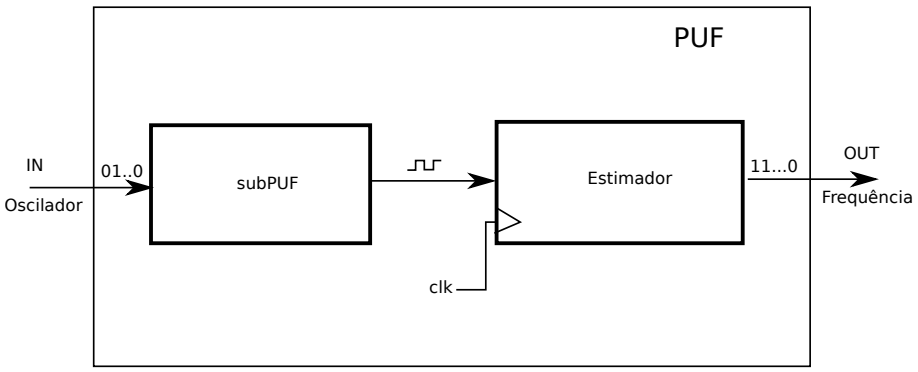


Figura 4.1: Diagrama esquemático da PUF

4.1.1 subPUF

A subPUF consiste em um decodificador, um multiplexador e um banco de osciladores. O sinal de enable de cada RO (Ring Oscillator ou oscilador em anel) é conectada em uma saída do decodificador e cada saída do RO é conectada a uma entrada do multiplexador e os sinais de seleção do decodificador e do multiplexador são conectados no mesmo nó, dessa forma, o mesmo oscilador habilitado pelo decodificador será selecionado pelo multiplexador. Um diagrama esquemático desse circuito pode ser observado na Figura 4.2. Isso permite que apenas o oscilador escolhido esteja ativo no momento da medição, diminuindo a potência consumida [18]. Devido a ativação dos osciladores apenas quando estes forem ser medidos, é possível que haja um comportamento transitório do oscilador que faça com que a medição não seja correta, porém é considerado nesse trabalho que a medição precisa da frequência do oscilador não é importante desde que ele mantenha sua unicidade e o estudo de transitórios do oscilador foi deixado para trabalhos futuros.

Quanto ao número de osciladores no banco, esse parâmetro pode ser configurado no momento de sua instanciação, utilizando os chamados parâmetros genéricos, ou apenas genéricos. Isso facilita o projeto pois permite a configuração do circuito sem a necessidade de acesso ao seu código fonte e permite que dois circuitos do mesmo tipo sejam instanciados com parâmetros diferentes.

O circuito da subPUF foi simulado antes de ser sintetizado na FPGA apenas para a visualização de seu funcionamento. Nessa simulação, quatro osciladores com frequências visualmente diferentes foram simulados e conectados ao decodificador e ao multiplexador. Na PUF a ser implementada, os osciladores são criados de forma idêntica, porém nesse caso, os osciladores simulados tem frequências diferentes. Os resultados são mostrados na Figura 4.3. Nessa figura, temos 3 grupos: A, B e C. A primeira linha do grupo A corresponde à sequência binária que corresponde à entrada da subPUF e consequentemente, da própria PUF. Na segunda linha, temos a saída da subPUF. Podemos ver nesse sinal a saída dos osciladores selecionados e, consequentemente a saída da subPUF. No grupo B, temos a saída do decodificador, sendo a primeira linha a representação binária das saídas e em seguida cada saída individualmente. No grupo C, temos a entrada do multiplexador, sendo a primeira linha a representação binária do sinal e nas demais, temos cada entrada individualmente.

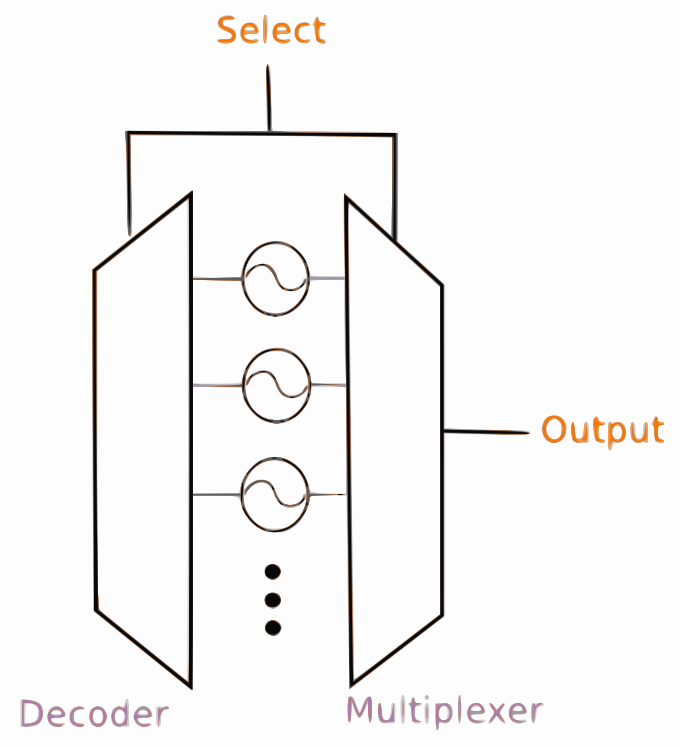


Figura 4.2: Circuito da subPUF

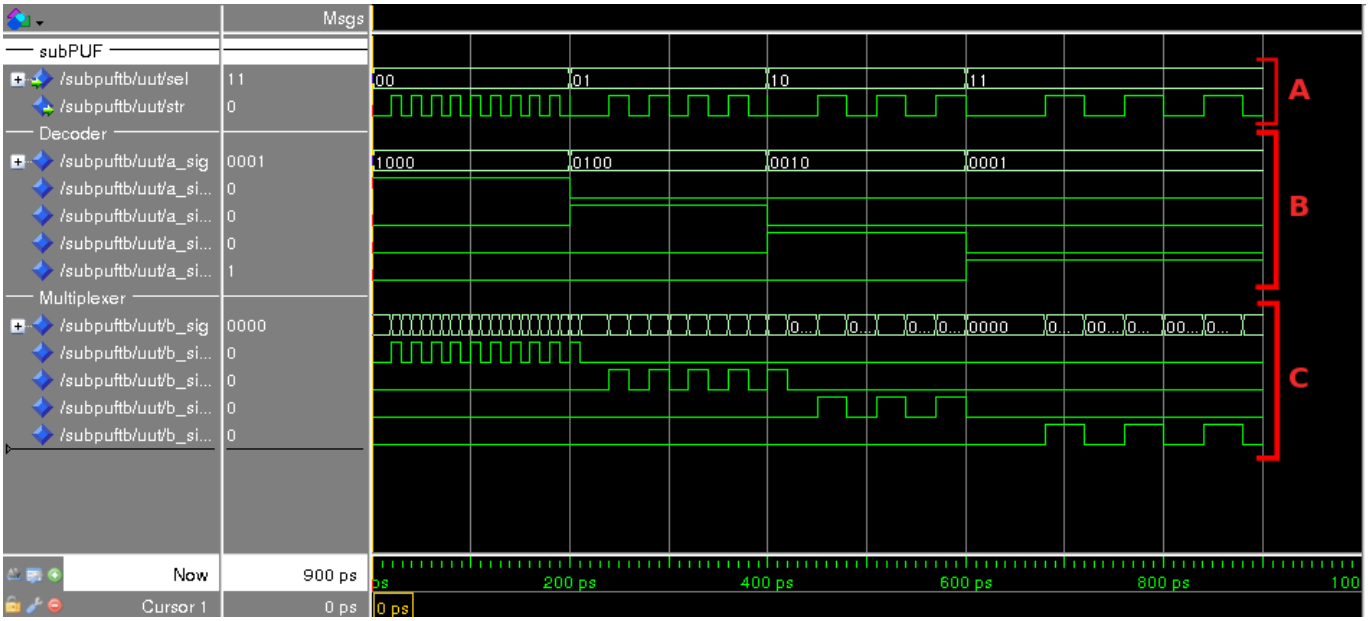


Figura 4.3: Simulação subPUF. No grupo A, temos a entrada e saída da subPUF; no grupo B, temos a palavra binária correspondente às saídas do decodificador, seguida pelas saída individuais; no grupo C, temos a a palavra binária correspondente às entradas do multiplexador, seguida das entradas individuais

4.1.1.1 Oscilador

Cada RO (Ring Oscillator - Oscilador em anel) possui uma entrada e uma saída. A entrada é chamada de enable e é responsável por ativar o oscilador através de uma porta XOR conectada à

linha de atraso do circuito. A saída do RO produz um sinal oscilante de uma frequência específica caso o enable seja igual a 1 e produz saída igual a 0 caso o enable seja 0.

Os osciladores geralmente são construídos utilizando-se portas inversoras conectadas em série, sendo a saída da última porta conectada à entrada da primeira, formando uma malha fechada. Quando se trata da implementação em FPGA, no entanto, o compilador VHDL tende a otimizar o código, simplificando a cadeia de inversoras em apenas uma porta inversora, impedindo a oscilação. Dessa forma, o circuito de fato implementado se torna ligeiramente diferente, utilizando uma porta XOR e uma sequência de buffers específicos, que são usados apenas para gerar atraso no circuito. Esses buffers são chamados de Lcell no ambiente de programação das placas Altera e não são otimizados pelo compilador, tornando possível a implementação dos osciladores.

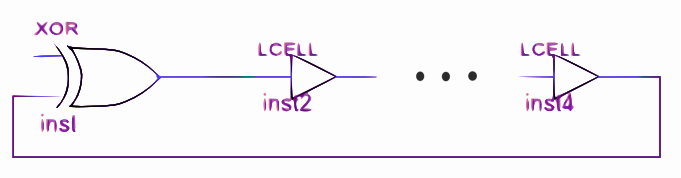


Figura 4.4: Oscilador de fato implementado

O número de células de atraso no circuito dos osciladores é um parâmetro genérico, tornando a implementação flexível para testes com diferentes números de células de atraso.

4.1.2 Estimador

O estimador possui como entrada o sinal do oscilador escolhido na subPUF e, como saída, a frequência desse oscilador. Para medir a frequência dos osciladores, o clock interno da FPGA é utilizado como referência para medição. Em seguida, verifica-se a relação entre a frequência do oscilador interno da placa e a frequência do sinal que vem da subPUF. Uma vez obtida essa relação e sabendo-se a frequência do clock interno da FPGA, estima-se a frequência do oscilador escolhido na subPUF.

Tanto o clock interno da placa quanto a saída da subPUF são ligados a um subcircuito chamado timer. Cada timer é um contador que conta as bordas de subida em sua entrada *str* e ativa um sinal *done* quando seu contador atinge um valor determinado. Cada timer também possui uma entrada *hold and update* que interrompe a contagem e mostra o valor do contador na saída e uma entrada *clear* que zera seu contador.

O timer do clock conta um valor pré definido de bordas e então ativa sua saída *done*. Quando esse sinal done é percebido pelo timer conectado ao oscilador, o timer do oscilador interrompe sua contagem e mostra o valor de seu contador na saída do timer. Essa saída é proporcional à frequência do oscilador e o valor do contador é então conectada à saída da PUF para ser processada. Na Figura 4.5, vemos a conexão dos timers na construção do estimador. Nessa figura, o timer de baixo é o timer conectado ao clock interno da FPGA e funciona como referência de tempo. Quando ele atinge um determinado valor, a saída *done* é ativada e ativa a entrada *hold_upd8* do timer de

cima, que é o timer conectado à subPUF. Quando essa entrada é ativada, a saída do estimador, e consequentemente a entrada da PUF é atualizada com o valor do contador interno do bloco timer.

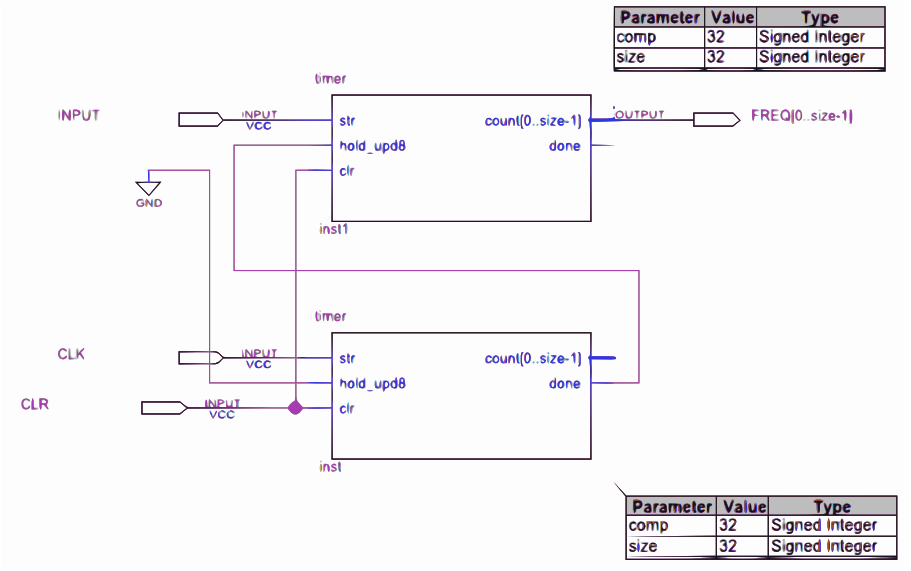


Figura 4.5: Arquitetura do estimador

Essa arquitetura apresentada pelo estimador possui algumas limitações. Primeiramente, ela depende de uma referência de clock. Caso essa referência não seja confiável, a medição também não o será, pois mesmo que a frequência do oscilador não varie, se a referência variar, o resultado será alterado. Uma outra limitação é sua precisão. Uma vez que o circuito trabalha apenas com o número de ciclos completos em um instante de tempo, ele não identificará valores fracionários de frequência. Essa precisão varia com a frequência do clock de referência e com o número de ciclos de clock utilizados na realização da medida e o número de bits menos significativos descartados.

A implementação do circuito estimador também possui parâmetros genéricos. É possível configurar o tamanho do vetor de saída do circuito, configurar o número de ciclos de clock que o circuito usa para estimar a frequência e é possível configurar o número de bits menos significativos descartados na saída.

Para a simulação desse circuito, um oscilador 9 vezes mais rápido que o clock foi criado e usado como entrada do estimador. O número de ciclos de clock utilizados para fazer a medição foi igual a 1024 ciclos e o tamanho da saída foi de 16 bits.

A simulação é mostrada na Figura 4.6. Na primeira linha dessa simualação, no sinal chamado Input, temos a simulação da entrada do sinal do oscilador. O sinal seguinte mostra o contador do timer acrescentando seu valor em cada borda de subida da entrada Input e interrompe a contagem quando o quarto sinal, Clock Counter, atinge o valor de 1024. No terceiro sinal, sob o nome de Clock, vemos a simulação do sinal de clock e no sinal de baixo, vemos o contador do Clock incrementar em uma unidade a cada borda de subida do clock. No quinto sinal, sob o nome Done, vemos que, quando o contador do clock atinge o número pré-definido, que neste caso é igual a 1024, ele muda de valor lógico, mudando de 0 para 1. Isso faz com que a saída chamada Output e a saída com o número ajustável de bits, chamada de Matched Output, atualizem seus valores de

zero para o valor em hexadecimal do contador do oscilador, neste caso, o valor do contador é igual a 9212 em decimal ou 23FC em hexadecimal.

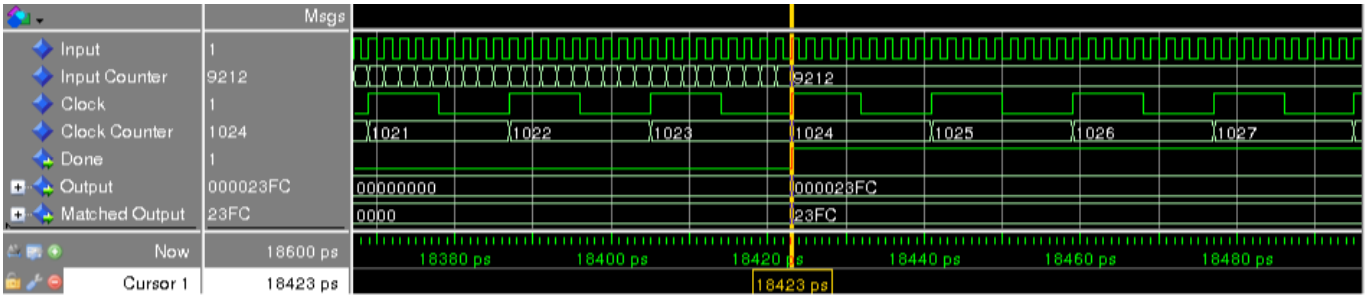


Figura 4.6: Simulação do estimador

Dividindo-se o valor obtido pela saída por 1024, que o número de ciclos de clock pré-definido, temos o valor igual a 8.99. Sabendo que o oscilador simulado foi feito para ser 9 vezes mais rápido que o clock, vemos que o circuito é capaz de medir a relação entre o clock de referência e o clock da subPUF.

4.2 Testbench

Todos os circuitos que foram simulados, tiveram primeiramente de ser montados no que chamamos de *testbench*, ou seja, uma bancada de testes. No testbench, criamos as condições propícias para que o circuito possa ser testado em termos de sinais de entrada e saída. No circuito da PUF, para que o circuito pudesse ser testado, um testbench também foi criado ao seu redor.

Essa testbench faz o que é descrito no capítulo 3, ou seja, captura várias medidas de frequência de cada oscilador em cada sequência de medição após o estimador ser resetado, conforme as condições de gatilho descritas no capítulo 3. A Figura 4.7 mostra o diagrama de blocos do circuito que foi usado para implementar a testbench. Essa figura pode ser diretamente associada à Figura 3.2, no capítulo de metodologia. Assim, temos o terceiro e o quarto bloco da Figura 4.7 são divisores de frequência e são utilizados para se implementar os laços de repetição interno e externo respectivamente da Figura 3.2. A relação entre divisores de frequência e laços de repetição é feita no sentido que o divisor de frequência tem sua saída ativada apenas quando ele detecta um determinado número de bordas em seu sinal de entrada. Dessa forma, se associarmos a cada iteração dentro do laço com uma borda no sinal de entrada do divisor de frequência, podemos usar o divisor de frequência como laço de repetição. Após os laços de repetição, o oscilador presente na Figura 3.2 corresponde ao circuito da subPUF em conjunto com o circuito do seletor na Figura 4.7. O circuito seletor é incluído pois é esse bloco de circuito que controla a ativação e desativação dos osciladores. Por fim, o bloco counter2 na Figura 4.7 corresponde ao medidor de frequencia na Figura 3.2.

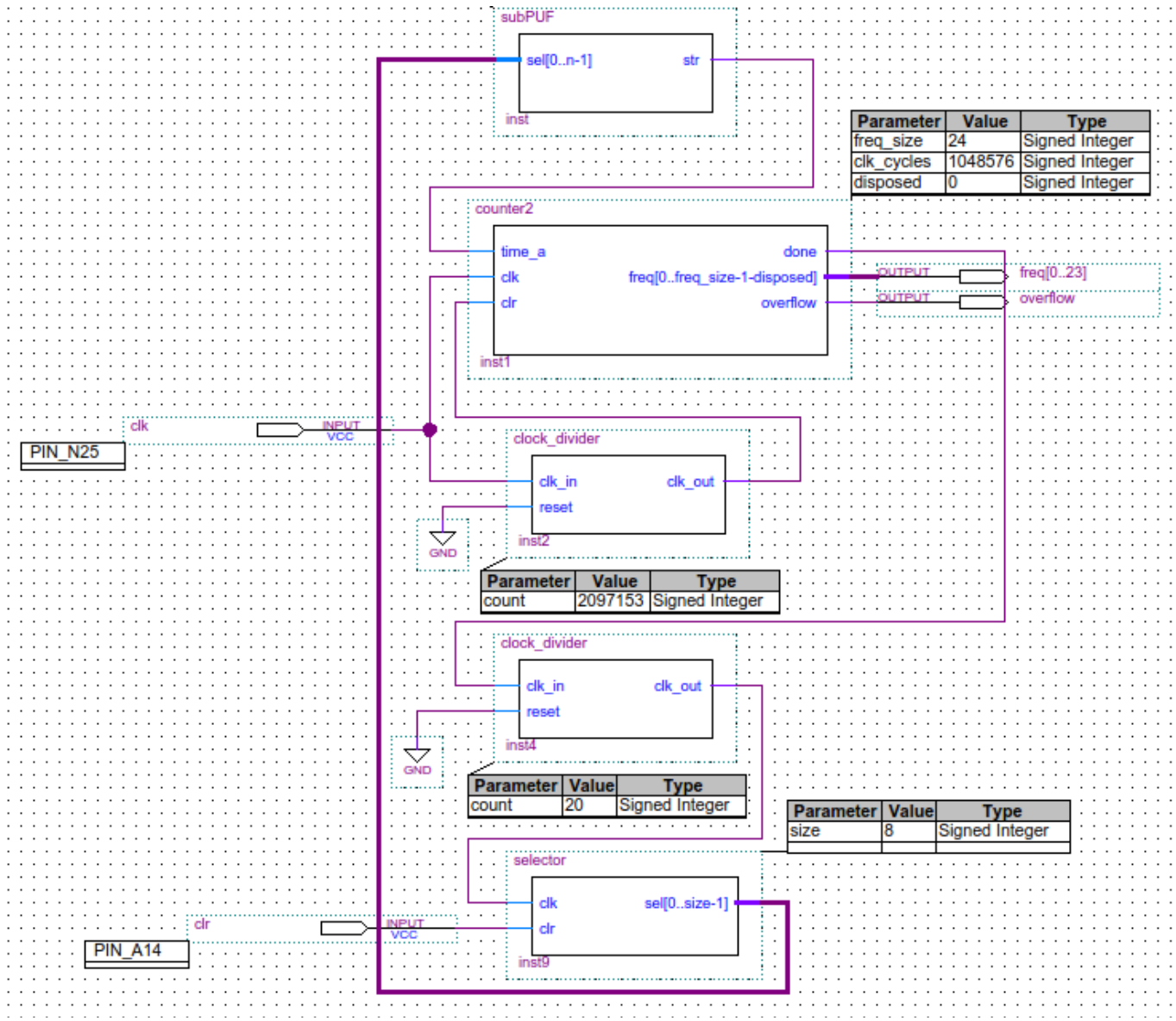


Figura 4.7: Implementação da testbench. Onde o rótulo counter2 corresponde ao estimador de frequência

Capítulo 5

Resultados Experimentais

5.1 Dados experimentais

Os dados obtidos nesta seção foram extraídos através do analisador lógico SingalTap II, que é embutido no ambiente de desenvolvimento Quartus II, e os dados processados por um script feito em Python.

Os valores obtidos para as frequências a seguir foram calculados da seguinte forma: iniciamos dois contadores simultaneamente, sendo um ativado pelas bordas de subida do clock interno e outro ativado pelas bordas de subida do sinal a ter a frequência medida. Quando o contador do clock interno atinge um valor pré determinado, ele dispara um sinal que interrompe o contador ligado ao sinal medido. Desta forma, a frequência do oscilador será o valor do contador do sinal a ser medido dividido pelo valor pré-determinado de ciclos de clock usados na medição e tudo isso é multiplicado pela frequência do clock da placa.

A primeira questão a ser investigada é o quão diferentes são as medidas de um mesmo oscilador em instantes de tempo diferentes. Para isso, realizamos várias **sequências de medição** e coletamos os dados referentes a um só oscilador. Como cada sequência de medição captura de 42 medições de frequência (amostras) para todos os 256 osciladores de uma vez (seção 3.2), após as sequências de medição, apenas os dados de um oscilador foram armazenados. Na Figura 5.1 temos a frequência de um mesmo oscilador em 6 sequências de testes diferentes. Desta forma, uma vez que cada sequência de medição captura 42 amostras de frequência de cada oscilador, cada curva da Figura 5.1 possui 42 amostras.

Uma vez que as medidas são independentes entre si e medem a frequênca do mesmo oscilador, espera-se algum tipo de distribuição independente da sequência de teste. Porém como pode ser observado na Figura 5.1, pode ser identificada uma certa tendência, onde as medições feitas na mesma sequência de medição são mais próximas entre si do que as medições de frequências em diferentes sequências de medição. Podemos ver isso, por exemplo, entre as curvas das sequências de testes 1 e 5, onde as faixas de variação destas duas curvas nem mesmo se interceptam.

A mesma questão também foi testada quando a PUF possui osciladores com mais células de

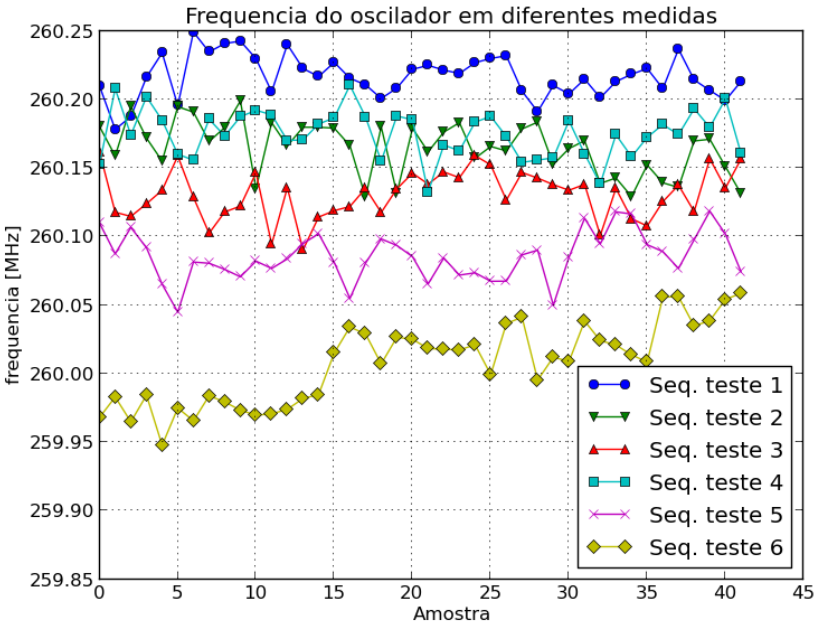


Figura 5.1: Distribuição de frequência para um oscilador com 5 células de atraso em diferentes sequências de medição

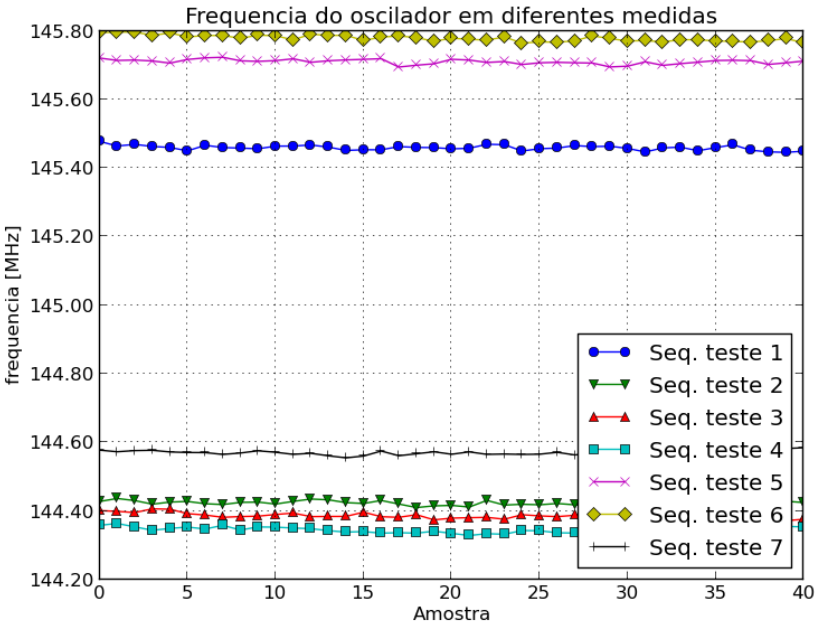


Figura 5.2: Distribuição de frequência para um oscilador com 10 células de atraso em diferentes sequências de medição

atraso. O teste é mostrado na Figura 5.2, na qual se pode ver a tendência de concentração destas frequências ao redor de uma frequência central de forma ainda mais acentuada.

Uma vez que os dados não foram todos capturados no mesmo dia, é possível que diferentes condições de temperatura, tensão de alimentação, condição de interconexões sejam diferentes, causando tal diferença. Porém, é considerado neste trabalho que a faixa de variação da frequência dos osciladores é suficientemente pequena para que um sistema de autenticação possa identificar qual a frequência do oscilador através de várias medições.

Após termos uma amostra da variação da frequência de um oscilador em diferentes instantes de tempo nas Figuras 5.1 e 5.2, o próximo passo é ver o quão próximas são as frequências de osciladores diferentes dentro de uma mesma PUF. Para isso, a frequência média de cada oscilador em cada sequência de medição foi calculada, juntamente com o seu desvio padrão percentual, o qual foi normalizado pela média. Na Figura 5.3 é mostrado as frequências médias e os desvios padrão percentual, normalizados pela frequência média de cada oscilador, para 256 osciladores em 6 sequências de medição diferentes. Todos os osciladores da Figura 5.3 têm 5 células de atraso.

Na Figura 5.3, várias sequências de medição foram feitas, sendo cada uma delas uma curva do gráfico. Porém as médias das frequências dos osciladores não variam o suficiente com diferentes medições para que as curvas possam ser discernidas visualemente. Por esta razão, foi feito uma ampliação em alguns pontos para que se possa ver que de fato há diferentes curvas no gráfico, cada uma sendo uma sequências de medição diferente, embora sejam semelhantes. Foi feito também o mesmo teste, só que desta vez com osciladores com 10 células de atraso. Estes resultados são mostrados na Figura 5.4.

O primeiro ponto a ser observado nas Figuras 5.3 e 5.4 é que embora os osciladores tenham sido concebidos com o mesmo projeto, há uma faixa de variação aproximadamente entre 220MHz e 300MHz, ou seja, de 36% com relação ao mínimo, para os osciladores com 5 células de atraso e uma faixa entre aproximadamente 140MHz a 155MHz, ou seja de aproximadamente de 10% com relação ao mínimo, para os osciladores com 10 células de atraso. Isso é um indicativo de que o número de células de atraso dos osciladores influencia na entropia de saída da PUF, ou seja, na distribuição das frequências nos osciladores da mesma PUF. Segundo, quando aumentamos o número de células de atraso de 5 para 10, o desvio padrão percentual diminui. Este resultado havia sido indicado nas Figuras 5.1 e 5.2 quando o aumento de células concentrava as medições de frequência em torno de uma frequência central, porém agora nas Figuras 5.3 e 5.4 temos um resultado quantitativo também. Outro fato interessante mostrado nos gráficos é que o desvio padrão não possui uma distribuição uniforme. Há certas regiões em que o desvio padrão assume valores menores do que em outros e isso indica que cada oscilador pode ser mais ou menos sucetível a variações, ou seja é mais sensível a condições externas. Isso pode ser um fator complicador nos esquemas de compensação térmica. Em [19], o autor cita os chamados *bit flips*, que acontecem quando dado dois osciladores A e B a frequência de A é maior do que a de B em determinada temperatura mas é menor do que B em outra faixa de temperatura. O ideal seria ter a relação entre frequência e temperatura para todos os osciladores individualmente, uma vez que suposições de que os osciladores variam de forma idêntica não são precisas.

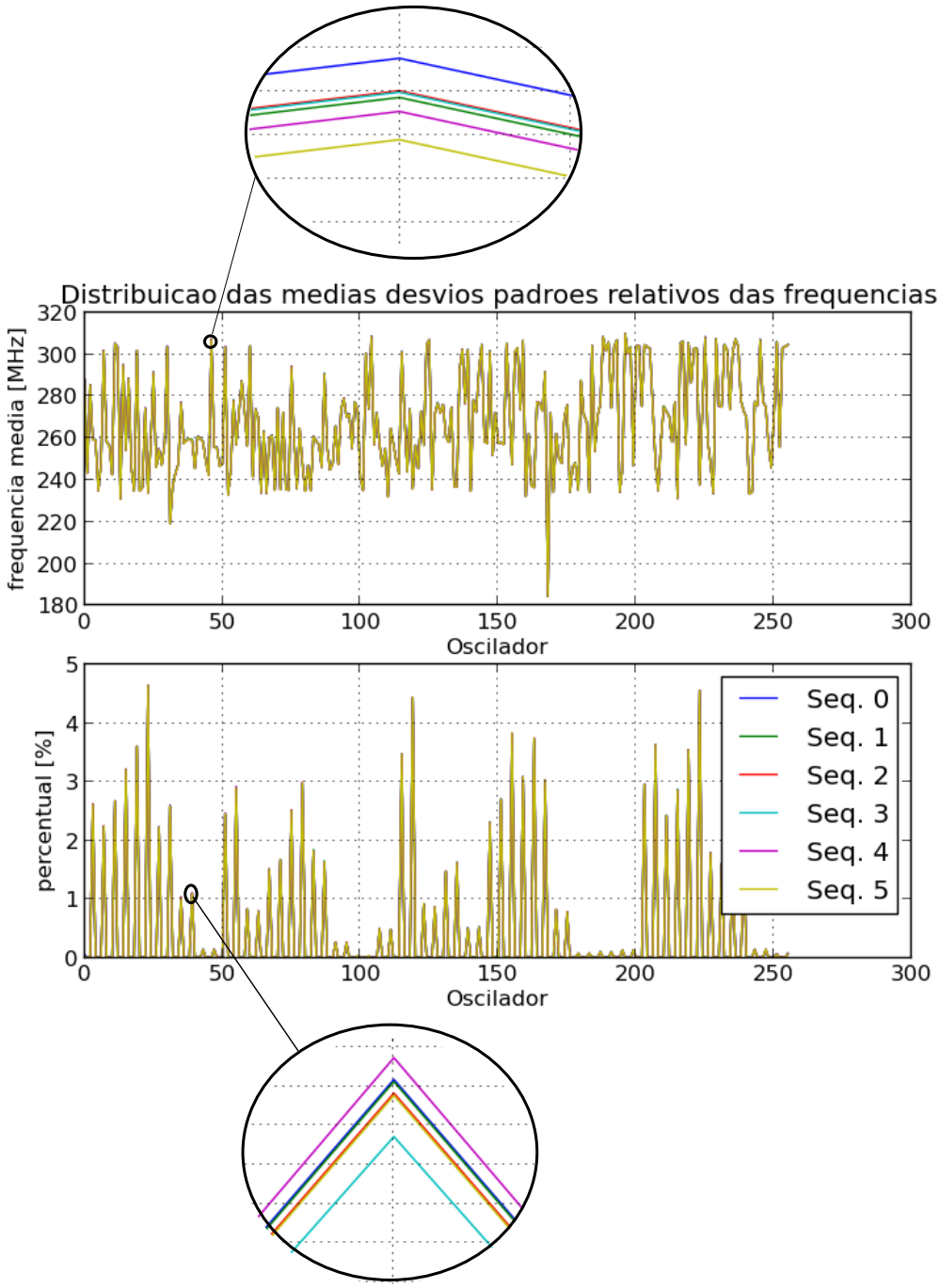


Figura 5.3: Média (frequência) e desvio padrão percentual (percentual) em PUF de osciladores com 5 células de atraso

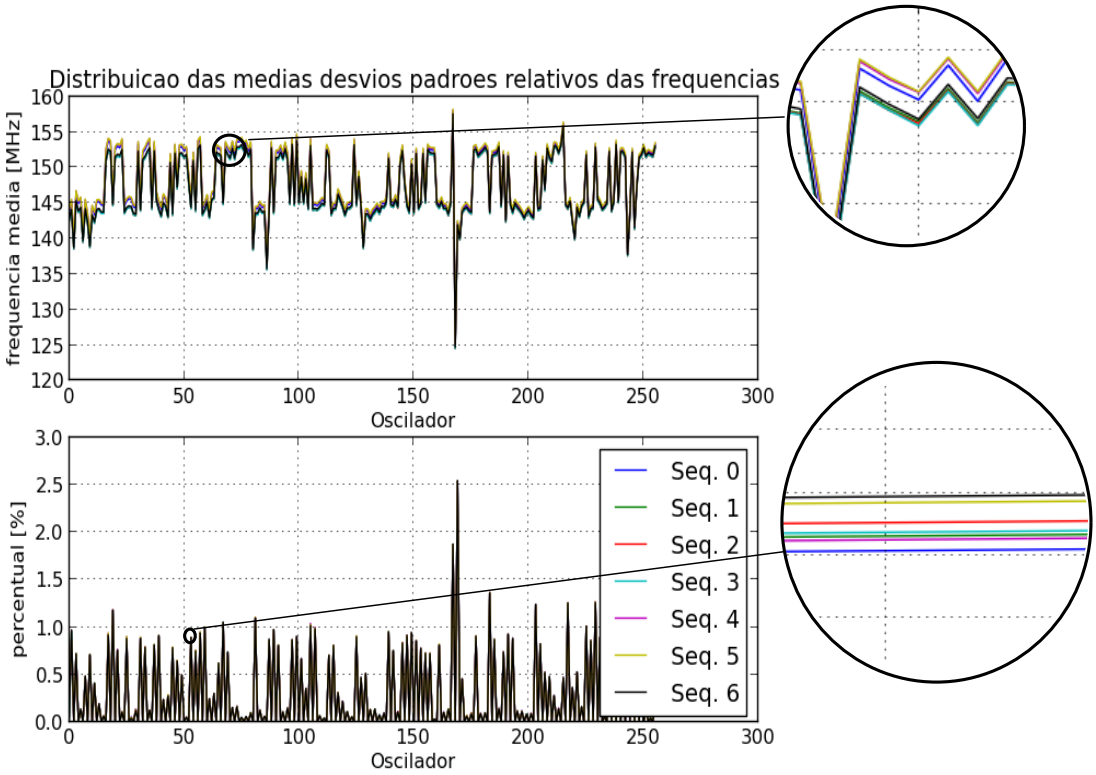


Figura 5.4: Média (frequência) e desvio padrão percentual (percentual) em PUF de osciladores com 10 células de atraso

As Figuras 5.3 e 5.4, então, fornecem informações importantes com respeito a uma PUF, como o valor médio de frequência de cada oscilador, a sensibilidade de cada oscilador a parâmetros externos (desvio padrão), a distribuição das frequências para cada sequência de medição de uma PUF. Por esta razão, daremos o nome de *assinatura* de uma sequência de medição para tais curvas, de modo a simplificar referências futuras.

Uma vez que é indicado que sequências de medição diferentes em instantes de tempo diferentes dos mesmos osciladores geram assinaturas próximas entre si, a próxima pergunta a ser feita é o quão próximas duas assinaturas são entre si. Este resultado está associado com a distância intra-classe da PUF, pois será analisado o quão parecido serão as respostas para os mesmos desafios para a mesma PUF. A métrica usada para calcular a distância entre duas assinaturas é o erro absoluto normalizado entre estas assinaturas, ou seja, duas assinaturas são subtraídas uma da outra e o valor absoluto é calculado e em seguida, esse valor de diferença é normalizado pelo valor dos elementos de uma das assinaturas. Na figura 5.5, tomou-se a assinatura de uma sequência de medição como referência para ser comparada com as outras e essa assinatura também será usada na normalização dos valores. Na figura, todos os osciladores da PUF são compostos de 5 células de atraso.

É mostrado na Figura 5.5 os valores correspondentes aos erros absolutos de algumas sequências de medição em comparação com uma referência. Devido à aglomeração de curvas na imagem, uma ampliação foi feita para auxiliar na visualização. Nessa Figura, os erros assumem, pelo menos

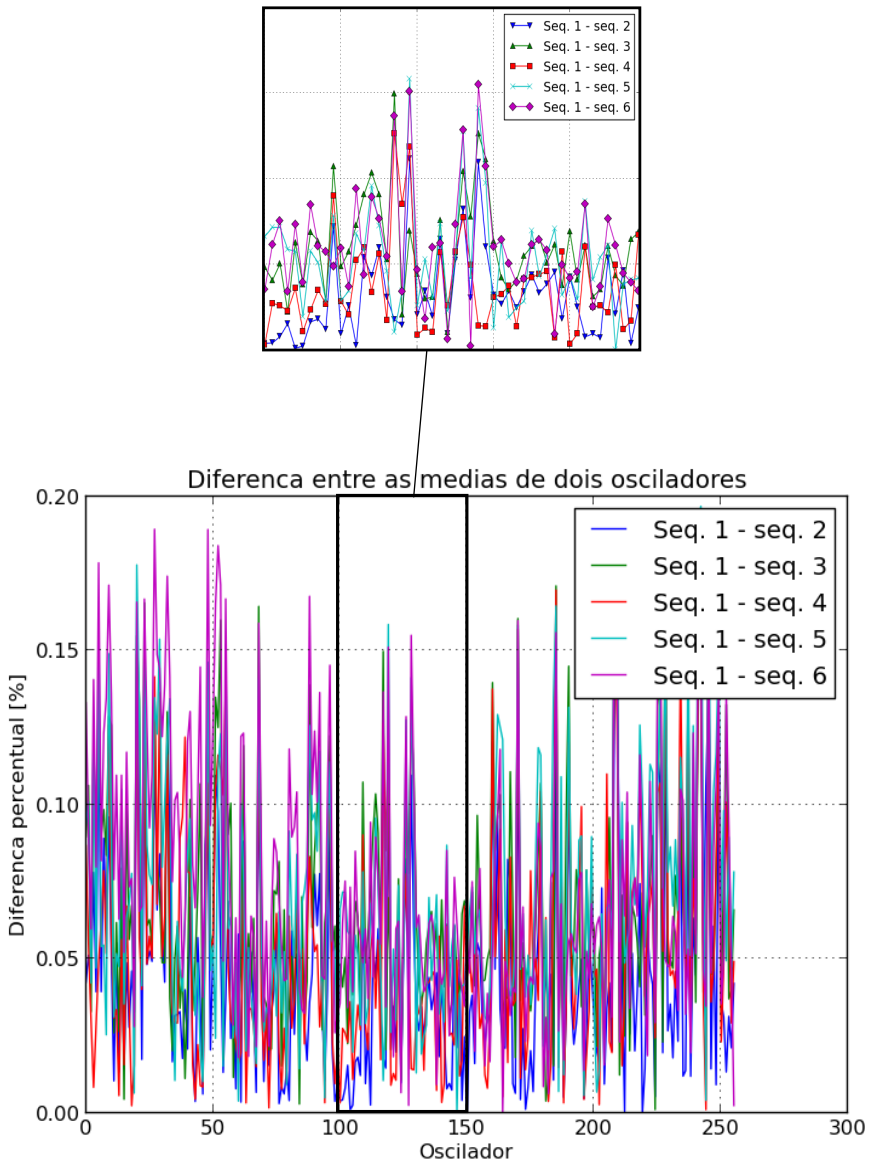


Figura 5.5: Diferença das assinaturas de duas sequências de medição para PUF com 5 células de atraso

visualmente, uma distribuição uniforme. Pode-se notar também que os erros não ultrapassam 0.2%.

Os erros também foram calculados para o caso em que os osciladores da PUF são formados por 10 células de atraso. Os resultados podem ser vistos na Figura 5.6

Pode-se notar com base na Figura 5.6 que quando o número de células de atraso aumenta para 10 células, há a formação de um padrão decrescente, mostrando que o erro em muitos casos é maior nos primeiros osciladores. Uma vez que a sequência de testes começa com os primeiros osciladores, há a possibilidade de algum efeito transitório na medição dos valores de frequência da PUF. Uma outra possibilidade é a formação de algum tipo de gradiente de temperatura ou de tensão dentro da FPGA e então pode estar relacionado com a distribuição dos componentes lógicos dentro da

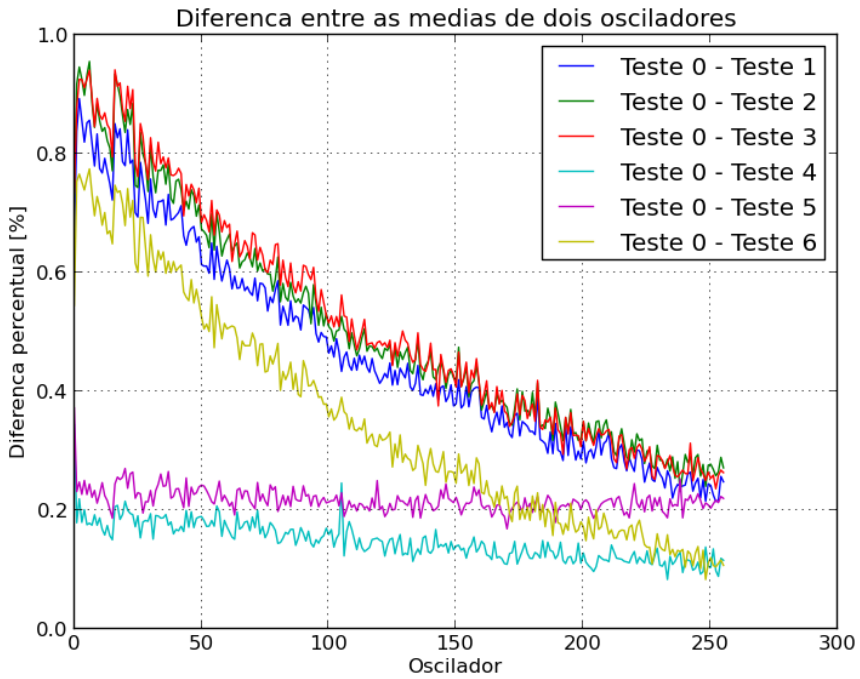


Figura 5.6: Diferença das assinaturas de duas sequências de medição para PUF com 10 células de atraso

FPGA. Tais questões, portanto, serão investigadas em trabalhos futuros.

Mesmo com a formação de um padrão decrescente, vemos que o erro obtido entre duas assinaturas é pequeno quando o mesmo arquivo de configuração é usado. Porém, quando diferentes arquivos de configuração sejam utilizados, é esperado que a PUF tenha seu comportamento alterado, uma vez que diferentes componentes internos são utilizados. Em [18], embora o modelo de PUF utilizado não seja idêntico ao desse trabalho mas também é baseado em osciladores em anel, o autor mostra resultados que quando diferentes regiões da FPGA são utilizadas, as respostas da PUF chegam a ter uma distância intra-classe de quase 50%, ou seja, quase como se uma PUF diferente fosse utilizada.

Para utilizarmos elementos diferentes, devemos utilizar diferentes *bitstreams*, ou seja, diferentes arquivos de configuração e para gerar outro bitstream, o código foi recompilado e então mapeado na FPGA e o mesmo procedimento de captura de dados foi feito. Para se ter uma indicação melhor da tendência do erro, os testes foram feitos também para PUFs com 20 células de atraso. As assinaturas de cada uma dessas PUFs para arquivos de configuração (files) diferentes pode ser Vista na Figura 5.7.

Na Figura 5.7 são mostradas as diferentes assinaturas para cada PUF. Note que há 3 pares de curvas e em cada par há uma linha contínua e uma linha tracejada. Cada par de curvas mostra um par de assinaturas de PUFs feitas com o mesmo projeto, porém possuem bitstreams diferentes. É possível notar que, em comparação com as Figuras 5.3 e 5.4, as curvas contínuas e tracejadas não se sobrepoem tão claramente. Isso é um indicativo de que o erro entre tais curvas será maior do que

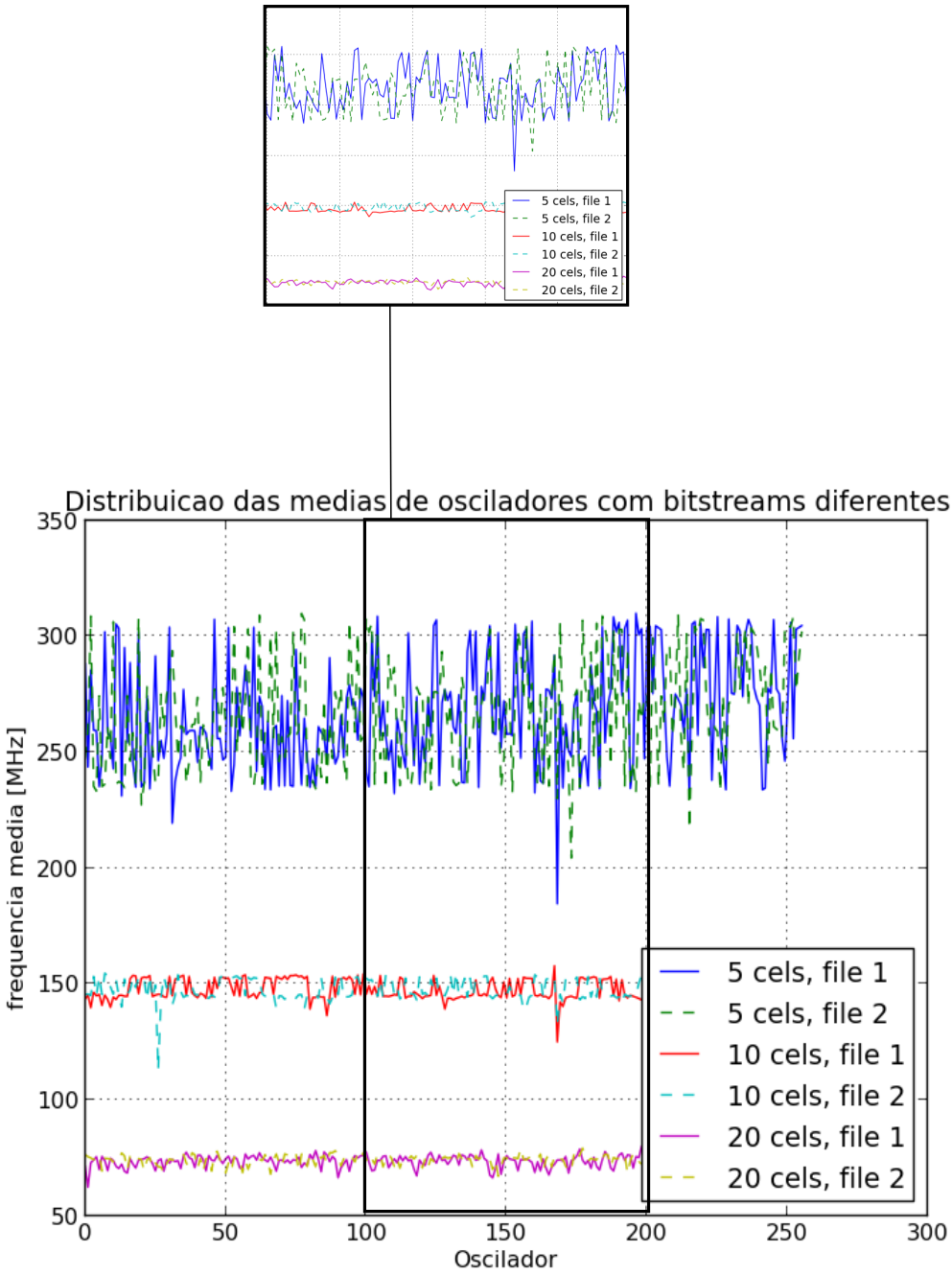


Figura 5.7: Médias das frequência entre dois testes com bitstreams diferentes

o mostrado nas Figuras 5.5 e 5.6. Para que esse resultado possa ser confirmado quantitativamente, o erro absoluto das assinaturas foi computado da mesma maneira que foi computado para quando se usou o mesmo bitstream de configuração. O resultado desse teste pode ser visto na Figura 5.8

Nas Figuras 5.5 e 5.6 que os erros não ultrapassam 1%. Porém, quando usamos dois bitstreams diferentes, conforme pode ser visto na Figura 5.8, os erros em alguns casos ultrapassam 20% nos casos mais extremos, sendo que o máximo valor obtido foi de quase 35%, e a maioria deles fica acima de pelo menos 5%. Isso indica que mesmo utilizando-se a mesma FPGA, caso a região

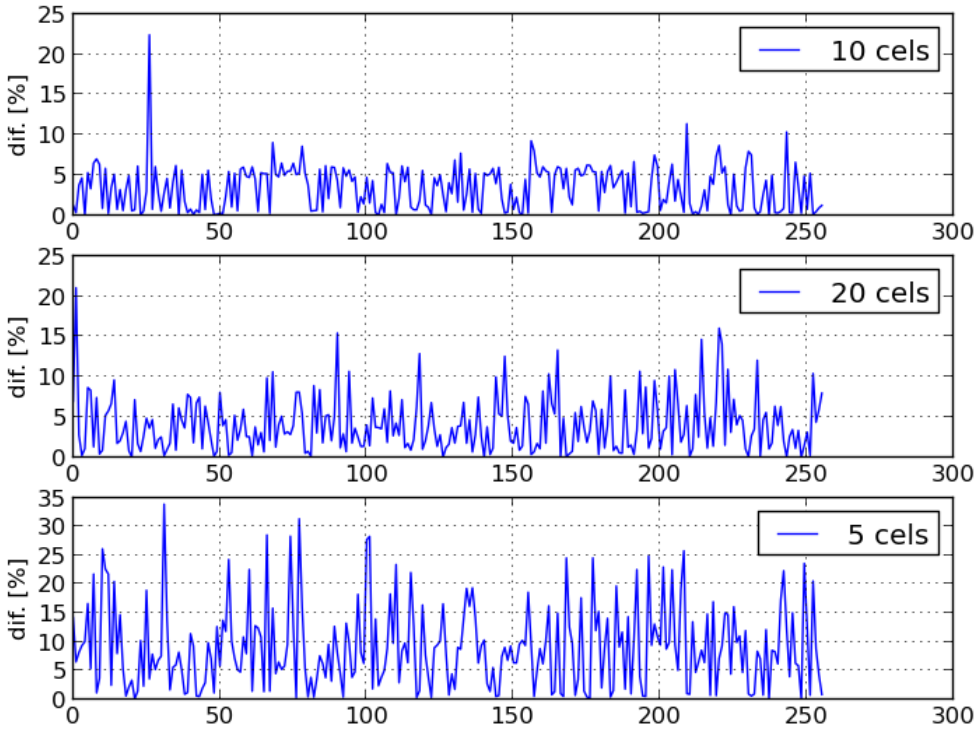


Figura 5.8: Diferença das assinaturas de duas sequências de medição com bitstreams diferentes

não seja bem definida, os resultados irão variar consideravelmente, o que é desejável numa PUF, pois caso esta sofra uma tentativa de ataque por modelagem, é necessário que o atacante modele diferentes regiões da FPGA.9

Capítulo 6

Conclusões

Testes foram conduzidos para se checar a possibilidade de se utilizar um banco de osciladores em FPGA como uma PUF. Para isso, foram testadas a variação das frequências desses osciladores em diferentes instantes de tempo, a variação da média das frequências dos osciladores dentro da mesma da PUF e a variação da média das frequências dos osciladores para arquivos de configuração diferentes dentro da mesma FPGA.

Quanto à variação das frequências dos osciladores em diferentes instantes de tempo, uma dos fatos observados é que a distribuição do desvio padrão dos osciladores não é uniforme, ou seja, há osciladores mais sensíveis a condições externas do que outros, mesmo o projeto sendo idêntico. Isso mostra que um mecanismo de compensação térmica, por exemplo, não pode ser implementado supondo que a frequência dos osciladores são igualmente sensíveis a condições ambientes.

Um outro resultado observado é que quando o número de células de atraso nos osciladores aumenta de 5 células para 10 células, as frequências tendem a ter uma variação menor com relação ao tempo, ou seja, bancos de osciladores de 5 células são mais robustos do que bancos de osciladores de 10 células, pelo menos quanto a diferentes instantes de tempo. Isso quer dizer que o valor da média das frequências dos osciladores pode ser obtida com mais precisão utilizando-se o mesmo número de amostras em um banco com osciladores com 10 células do que um banco com 5 células. Em trabalhos futuros, pode ser feito uma investigação sobre o número ótimo de células que compoem um banco de osciladores.

Não apenas a distribuição das frequências em diferentes instantes de tempo mas a variação da média dos osciladores em uma PUF também varia com o número de células utilizadas nos osciladores. Quando aumentamos o número de células de 5 para 10, é possível notar que as médias das frequências se concetram em torno de um frequência central. Dessa forma, dependendo da precisão da medição de frequência, isso pode levar a uma diminuição na entropia de saída.

Quanto ao uso de bitstreams de configurações diferentes na mesma FPGA, o resultado mostrado é que mesmo quando o mesmo projeto é utilizado na mesma FPGA, os resultados diferem consideravelmente. Isso indica que o uso de FGPAs como plataformas para PUFs é promissora, pois os pares desafio resposta dependem do projeto da PUF, da FPGA em si e da região da FPGA na qual o circuito foi mapeado, fazendo com que caso um atacante tente impersonar um usuário

de uma PUF ele precisará dessas três informações. Portanto, o trabalho conclui que as PUFs baseadas em osciladores em anel mapeadas em FPGA são um tópico promissor para dispositivos de segurança.

Como trabalhos futuros, podemos citar um sistema de compensação térmica para melhorar a robustez da PUF. Tal sistema tem que levar em consideração o fato de os osciladores variarem não uniformemente, portanto uma sugestão é que durante a fase de autenticação da PUF, dois valores de média sejam adquiridos para cada oscilador, cada um em uma temperatura diferente. Com esses dois valores, é possível fazer uma interpolação linear e, uma vez conhecendo a temperatura ambiente, é possível prever qual será o comportamento do oscilador. Esse tipo de mecanismo é interessante pois não diminui a entropia de saída da PUF, não ocupa mais espaço no circuito e não descarta osciladores da PUF. Tem como desvantagem, porém, para o lado do mecanismo de autenticação, que necessitará de duas vezes mais memória para armazenar os dados de cada PUF.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] KATZENBEISSER, S. et al. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. In: *Cryptographic Hardware and Embedded Systems–CHES 2012*. [S.l.]: Springer, 2012. p. 283–301.

[2] MAES, R.; VERBAUWHEDE, I. Physically unclonable functions: A study on the state of the art and future research directions. In: *Towards Hardware-Intrinsic Security*. [S.l.]: Springer, 2010. p. 3–37.

[3] BERG, R. van den. *Entropy analysis of Physical Unclonable Functions*. Tese (Doutorado) — MSc. thesis, Eindhoven University of Technology, 2012.

[4] HELFMEIER, C. et al. Cloning physically unclonable functions. In: IEEE. *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*. [S.l.], 2013. p. 1–6.

[5] PAPPU, R. et al. Physical one-way functions. *Science*, v. 297, n. 5589, p. 2026–2030, 2002. Disponível em: <http://www.sciencemag.org/content/297/5589/2026.abstract>.

[6] BUCHANAN, J. D. et al. Fingerprinting documents and packaging. *Nature*, Nature Publishing Group, v. 436, n. 28, p. 475, 2005.

[7] DEJEAN, G.; KIROVSKI, D. *Radio frequency certificates of authenticity*. Google Patents, mar. 16 2010. US Patent 7,677,438. Disponível em: <http://www.google.com/patents/US7677438>.

[8] INDECK, R.; MULLER, M. *Method and apparatus for fingerprinting magnetic media*. Google Patents, nov. 15 1994. US Patent 5,365,586. Disponível em: <http://www.google.com/patents/US5365586>.

[9] HAMMOURI, G.; DANA, A.; SUNAR, B. Cds have fingerprints too. In: *Cryptographic Hardware and Embedded Systems-CHES 2009*. [S.l.]: Springer, 2009. p. 348–362.

[10] LOFSTROM, K.; DAASCH, W. R.; TAYLOR, D. Ic identification circuit using device mismatch. In: IEEE. *Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International*. [S.l.], 2000. p. 372–373.

[11] HELINSKI, R.; ACHARYYA, D.; PLUSQUELLIC, J. A physical unclonable function defined using power distribution system equivalent resistance variations. In: ACM. *Proceedings of the 46th Annual Design Automation Conference*. [S.l.], 2009. p. 676–681.

[12] TUYLS, P. et al. Read-proof hardware from protective coatings. In: *Cryptographic Hardware and Embedded Systems-CHES 2006*. [S.l.]: Springer, 2006. p. 369–383.

[13] GUAJARDO, J. et al. Anti-counterfeiting, key distribution, and key storage in an ambient world via physical unclonable functions. *Information Systems Frontiers*, Springer, v. 11, n. 1, p. 19–41, 2009.

[14] LIM, D. et al. Extracting secret keys from integrated circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, IEEE, v. 13, n. 10, p. 1200–1205, 2005.

[15] GASSEND, B. et al. Silicon physical random functions. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2002. (CCS '02), p. 148–160. ISBN 1-58113-612-9. Disponível em: <http://doi.acm.org/10.1145/586110.586132>.

[16] SUH, G. E.; DEVADAS, S. Physical unclonable functions for device authentication and secret key generation. In: ACM. *Proceedings of the 44th annual Design Automation Conference*. [S.l.], 2007. p. 9–14.

[17] MAITI, A. et al. A large scale characterization of ro-puf. In: *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*. [S.l.: s.n.], 2010. p. 94–99.

[18] COSTEA, C. et al. Analysis and enhancement of ring oscillators based physical unclonable functions in fpgas. In: IEEE. *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*. [S.l.], 2010. p. 262–267.

[19] QU, G.; YIN, C.-E. Temperature-aware cooperative ring oscillator puf. In: IEEE. *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*. [S.l.], 2009. p. 36–42.

ANEXOS

I. CÓDIGOS

I.1 Clock Divider

```
-----------------
--Clock Divider--
-----------------
library ieee;
use ieee.std_logic_1164.all;

entity clock_divider is
generic (count : integer := 100);
port(
clk_in: in std_logic;
clk_out: out std_logic;
reset: in std_logic
);
end clock_divider;

architecture clock_divider_behv of clock_divider is
signal temporal: std_logic;
signal counter: integer range 0 to count := 0;
begin

frequency_divider: process(reset,clk_in)
begin
if (reset = '1') then
temporal <= '0';
counter <= 0;
elsif rising_edge(clk_in) then
if(counter=count) then
temporal <= not(temporal);
counter <= 0;
else
counter <= counter+1;
end if;
end if;
end process;
```

```
clk_out <= not temporal;
end clock_divider_behv;
```

I.2 Estimador

```
--------------
--Counter v2--
--------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter2 is
generic(freq_size : integer := 16; -- Tamanho da palavra de saida
clk_cycles : integer := 8; -- Numero de ciclos de clock para amostragem
disposed : integer := 4);
port(time_a : in std_logic; -- Sinal a ser amostrado
clk : in std_logic; -- Sinal de referencia
clr : in std_logic; -- Clear
done : out std_logic := '0'; -- Indica se a amostragem foi concluida
freq : out std_logic_vector(0 to freq_size-1-disposed); -- palavra de saida
overflow : out std_logic := '0'); -- Indica se houve overflow
end entity;

architecture bhv of counter2 is

component timer is
generic(comp : integer := 32; -- Numero a ser comparado
size : integer := 32); -- Tamanho do comparador
port(str : in std_logic; -- Sinal de entrada
hold_upd8 : in std_logic; -- Interrompe a contagem e atualiza a saida
clr : in std_logic; -- Clear
count : out std_logic_vector(0 to size-1) := (others => '0'); -- Valor do contador interno
done : out std_logic := '0'); -- Indica se o contador interno atingiu o valor a ser comparado
end component;

signal count_signal : std_logic_vector(0 to 31); -- Sinal interno entre timer e media
signal done_signal : std_logic; -- Sinal interno da saida done

signal vcc : std_logic := '1';
```

```
signal gnd : std_logic := '0';

begin

clk_timer : timer -- Conta o numero de pulsos de clock e emite um sinal quando esse numero eh
generic map(comp => clk_cycles,size => 5)
port map(str => clk, hold_upd8 => gnd, clr => clr,count => open,done => done_signal);

sample_timer: timer
generic map(comp => 2147483647,size => 32)
port map(str => time_a, hold_upd8 => done_signal, clr => clr, count => count_signal, done => o

process(count_signal)
  begin
    freq <= count_signal(32-freq'length-disposed to 31-disposed);
 done <= done_signal;
--    freq <= std_logic_vector(to_unsigned(count_signal,freq'length));
  end process;

end architecture;
```

I.3 Decodificador Genérico

```
-----------
--Decoder--
-----------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gen_decoder is
generic (n : integer := 2); -- numero de bits da entrada
port(
din : in std_logic_vector(0 to n-1);
dout : out std_logic_vector(0 to (2**n-1)) -- numero de saidas
);
end gen_decoder;

architecture behv of gen_decoder is
```

```
begin
process(din)
begin
dout <= (others => '0');
dout(to_integer(unsigned(din))) <= '1';
end process;
end behv;
```

I.4 Decodificador Genérico - Testbench

```
-------------
--subPUF TB--
-------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder_tb is
end entity;

architecture bhv of decoder_tb is
constant n : integer := 2;

component gen_decoder is
generic (n : integer := 2); -- numero de bits da entrada
port(
din : in std_logic_vector(0 to n-1);
dout : out std_logic_vector(0 to (2**n-1)) -- numero de saidas
);
end component;

signal sel_sig : std_logic_vector(0 to n-1);
signal output : std_logic_vector(0 to 2**n-1);

begin

uut: gen_decoder
generic map(n => 2)
port map(sel_sig,output);
```

```
process
begin
for i in 0 to (2**n)-1 loop
sel_sig <= std_logic_vector(to_unsigned(i,n));
wait for 200 ps;
end loop;
wait;
end process;

end bhv;
```

I.5 Mux Genérico

```
-----------
--Gen Mux--
-----------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gen_mux is
generic (nsel : integer := 2); -- numero de bits do select
port(
sel : in std_logic_vector(0 to nsel-1) := (others => '0'); -- tamanho do select
input : in std_logic_vector(0 to 2**nsel-1) := (others => '0'); -- numero de entradas
output : out std_logic
);
end gen_mux;

architecture behv of gen_mux is
signal aux : integer := 0;
begin
process(sel)
begin
    aux <= to_integer(unsigned(sel));
  end process;

  process(input,aux)
  begin
```

```
output <= input(aux);
end process;
end behv;
```

I.6 Mux Genérico - Testbench

```
--------------
--Gen_mux TB--
--------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gen_mux_tb is
end gen_mux_tb;

architecture bhv of gen_mux_tb is

component gen_mux is
generic (nsel : integer := 2); -- numero de bits do select
port(
sel : in std_logic_vector(0 to nsel-1); -- tamanho do select
input : in std_logic_vector(0 to 2**nsel-1); -- numero de entradas
output : out std_logic
);
end component;

constant n : integer := 2;
signal sel_sig : std_logic_vector(0 to n-1) := (others => '0');
signal input_sig : std_logic_vector(0 to 2**n-1) := ("0101");
signal output_sig : std_logic;

begin

uut: gen_mux
generic map (nsel => n)
port map (sel_sig,input_sig,output_sig);

process
begin
```

```
for i in 0 to 2**n-1 loop
sel_sig <= std_logic_vector(to_unsigned(i,n));
wait for 100 ps;
end loop;
wait;
end process;
end bhv;
```

I.7 Ring Oscilator

```
-------------------
--Ring Oscillator--
-------------------

library altera_mf;
use altera_mf.altera_mf_components.all;

library ieee;
use ieee.std_logic_1164.all;

entity RO is
generic (n : integer := 7);
port( en: in std_logic;
str: out std_logic);
end RO;

architecture behv of RO is

signal i : integer := 0;

signal wire : std_logic_vector(0 to n+1);

component lcell
    port (
        a_in : in std_logic;
        a_out : out std_logic);
end component;

begin
gen_lcell:
for i in 0 to n generate
```

```
ro_array: lcell port map
(wire(i), wire(i+1));
end generate;
wire(0)<= en xor wire(n+1);
str <=wire(n+1);
end behv;
```

I.8 Seletor

```
------------
--Selector--
------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity selector is
generic(size: integer := 8);
port(clk : in std_logic := '0';
--en : in std_logic := '0';
clr : in std_logic := '0';
sel : out std_logic_vector (0 to size-1)
);
end entity;

architecture bhv of selector is

signal clk_signal, en_signal, clr_signal, sel_signal : std_logic;
signal count : integer := 0;
begin

process(clk,clr)
begin
if clr = '1' then
if rising_edge(clk) and clk = '1' then
sel <= std_logic_vector(to_unsigned(count,sel'length));
count <= count + 1;
end if;
else
```

```
count <= 0;
sel <= std_logic_vector(to_unsigned(count,sel'length));
end if;
end process;

end architecture;
```

I.9 Seletor - Testbench

```
--------------
--Seletor_tb--
--------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity seletor_tb is
end entity;

architecture bhv of seletor_tb is

component selector is
generic(size: integer := 8);
port(clk : in std_logic := '0';
--en : in std_logic := '0';
clr : in std_logic := '0';
sel : out std_logic_vector (0 to size-1)
);
end component;

constant period : time := 10 ps;

signal clr_signal : std_logic := '1';
signal clk_signal : std_logic := '0';
signal sel_probe : std_logic_vector(0 to 7);

begin

uut: selector
generic map(8)
```

```
port map(clk_signal,clr_signal,sel_probe);

clk_prcs: process
begin
wait for period;
clk_signal <= not clk_signal;
end process;

end architecture;
```

I.10 subPUF

```
----------
--SubPUF--
----------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity subPUF is
generic (n : integer := 2;
cells : integer := 7); -- tamanho do select
port (sel : in std_logic_vector(0 to n-1);
str: out std_logic);
end entity;

architecture bhv of subPUF is

component gen_decoder is
generic (n : integer := 2); -- numero de bits da entrada
port(
din : in std_logic_vector(0 to n-1);
dout : out std_logic_vector(0 to (2**n-1)) -- numero de saidas
);
end component;

component gen_mux is
generic (nsel : integer := 2); -- numero de bits do select
port(
sel : in std_logic_vector(0 to nsel-1); -- tamanho do select
```

```
input : in std_logic_vector(0 to 2**nsel-1); -- numero de entradas
output : out std_logic
);
end component;

--component osc_test is
--generic(t : time := 10 ns);
--port(en: in std_logic := '0';
--str: out std_logic := '0');
--end component;

component RO is
generic (n : integer := 7);
port( en: in std_logic;
str: out std_logic);
end component;

--signal sel_sig: std_logic_vector(0 to n-1);
signal a_sig : std_logic_vector(0 to 2**n-1);
signal b_sig : std_logic_vector(0 to 2**n-1);
signal str_signal : std_logic;

begin

decoder: gen_decoder
generic map (n => n)
port map(sel,a_sig);

mux: gen_mux
generic map (n)
port map (sel,b_sig,str);

Osc:
for i in 0 to 2**n-1 generate
Osc_i: RO
generic map (n => cells)
port map(a_sig(i),b_sig(i));
end generate;

end bhv;
```

I.11 subPUF - Testbench

```
-------------
--subPUF TB--
-------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity subPUFtb is
end entity;

architecture bhv of subPUFtb is
constant n : integer := 2;

component subPUF is
generic (n : integer := 2); -- tamanho do select
port (sel : in std_logic_vector(0 to n-1);
str: out std_logic);
end component;

signal sel_sig : std_logic_vector(0 to n-1);
signal str_sig : std_logic;

begin

uut: subPUF
generic map(n => 2)
port map(sel => sel_sig, str => str_sig);

process
begin
for i in 0 to 2**n-1 loop
sel_sig <= std_logic_vector(to_unsigned(i,n));
wait for 200 ps;
end loop;
wait;
end process;

end bhv;
```

I.12 Timer

```
---------
--Timer--
---------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity timer is
generic(comp : integer := 32; -- Numero a ser comparado
size : integer := 32); -- Tamanho do comparador em bits
port(str : in std_logic; -- Sinal de entrada
hold_upd8 : in std_logic; -- Interrompe a contagem e atualiza a saida
clr : in std_logic; -- Clear
count : out std_logic_vector(0 to size-1) := (others => '0'); -- Contador interno
done : out std_logic := '0'); -- Indica se o contador e maior que o numero a ser comparado
end entity;

architecture bhv of timer is
signal count_signal: integer := 0;
begin
process(str, clr, hold_upd8)
begin
if clr = '0' then -- Clear control
if hold_upd8 = '0' then -- Update control
if rising_edge(str) and str = '1' then -- Edge counting
count_signal <= count_signal+1; -- Increase counter
end if;
else
count <= std_logic_vector(to_unsigned(count_signal,size)); -- Hold control
end if;
else
  count_signal <= 0;
count <= (others => '0');
end if;
end process;

process(count_signal,clr)
begin
   if clr = '0' then
```

```
     if count_signal >= comp then -- check done
     done <= '1';
     end if;
   else
     done <= '0';
 end if;
  end process;

end architecture;
```

I.13 Timer - Testbench

```
------------
--Timer TB--
------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity timer_tb is
end entity;

architecture bhv of timer_tb is
component timer is
generic(comp : integer := 32; -- Numero a ser comparado
size : integer := 32); -- Tamanho do comparador em bits
port(str : in std_logic; -- Sinal de entrada
hold_upd8 : in std_logic; -- Interrompe a contagem e atualiza a saida
clr : in std_logic; -- Clear
count : out std_logic_vector(0 to size-1) := (others => '0'); -- Contador interno
done : out std_logic := '0'); -- Indica se o contador e maior que o numero a ser comparado
end component;

signal clk_signal : std_logic := '0';
signal upd8_signal : std_logic := '0';
signal clr_signal : std_logic := '0';

constant size_constant : integer := 4;
```

```
begin

uut: timer
generic map(comp => (2**size_constant), size => size_constant)
port map(clk_signal,upd8_signal,clr_signal,open,open);

clk_process: process
begin
wait for 2 ps;
clk_signal <= not clk_signal;
end process;

ctrl_process: process
begin
wait for 200 ps;
clr_signal <= '1';
wait for 200 ps;
clr_signal <= '0';
wait for 200 ps;
upd8_signal <= '1';
wait for 200 ps;
upd8_signal <= '0';
end process;


end architecture;
```

I.14 SignalTap

Para que o SignalTap possa funcionar de forma adequada, as condições de gatilho devem ser configuradas na aba setup. As configurações utilizadas foram:

- O clock utilizado para o SignalTap II foi o clock interno da placa
- Data > Sample depth : 16k
- Data > Storage qualifier > Type: Conditional
- Trigger > Trigger flow control : Sequential
- Trigger position > Pre trigger position
- Trigger condition > 1

Os sinais escolhidos para o circuito da PUF foram: a saída de select do seletor, a frequência de saída, a entrada de clear do estimador, a saída do clock divider conectado ao seletor, o sinal de clear do seletor e o sinal done, que indica quando o processo de estimação da frequencia está completo.

Para que os sinais sejam adquiridos sempre da mesma forma, as condiçoes de gatilho foram definidas da seguinte forma:

- Para que seja os dados sejam gravados apenas quando o sinal done for ativado, o gatilho *storage qualifier* foi configurado para ser ativado nas bordas de subida do sinal done.
- Para que a simulação iniciasse no momento adequado, a condição de gatilho foi configurada para borda de descida do sinal done e nível lógico 1 para o sinal de clear do seletor.

II. DESCRIÇÃO DO CONTEÚDO DO CD

- Relatório em PDF
- Resumo e palavras chave