PROJETO, IMPLEMENTAÇÃO E DESEMPENHO DOS ALGORITMOS CRIPTOGRÁFICOS AES, PRESENT E CLEFIA EM FPGA

William Pedrosa Maia

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica – PROEE, da Universidade Federal de Sergipe, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Edward David Moreno Ordonez

São Cristóvão-SE, Brasil
Agosto de 2017

PROJETO, IMPLEMENTAÇÃO E DESEMPENHO DOS ALGORITMOS CRIPTOGRÁFICOS AES, PRESENT E CLEFIA EM FPGA

William Pedrosa Maia

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA – PROEE DA UNIVERSIDADE FEDERAL DE SERGIPE COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA.

Examinado por:

_____

Prof. Dr. Eduardo Oliveira Freire (PROEE/UFS)
Presidente

_____

Profa. Dra. Edilayne Meneses Salgueiro (UFS)
Examinadora Externa

_____

Prof. Dr. Fábio Dacêncio Pereira (UNIVEM)
Examinador Externo

São Cristóvão-SE, Brasil
Agosto de 2017

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL
UNIVERSIDADE FEDERAL DE SERGIPE

Maia, William Pedrosa.
M217p Projeto, implementação e desempenho dos algoritmos criptográficos AES, PRESENT e CLEFIA em FPGA / William Pedrosa Maia ; orientador Edward David Moreno Ordonez. – São Cristóvão, 2017.
115 f. : il.

Dissertação (mestrado em Engenharia Elétrica) – Universidade Federal de Sergipe, 2017.

1. Criptografia. 2. Sistemas Embarcados. 3. Algoritmos. 4. Arranjos de lógica programável em campo. I. Ordonez, Edward David Moreno, orient. II. Título.

CDU 621.3

In memoriam de Benjamim Maxwell Maia, meu filho amado.

Agradecimentos

Agradeço a Deus, por me capacitar e ajudar a concluir este trabalho.
"*Mas, lembrem-se do Senhor, do seu Deus, pois é ele que lhes dá a capacidade de produzir riqueza*". (Dt. 8:18a)

A minha minha querida esposa, Lívian Maia, pelo amor, compreensão e incentivo, inclusive por ter me acompanhado em outra cidade para a realização deste trabalho.

Aos meus pais, Lázaro e Conceição, por todo amor, ensino, paciência e incentivo, e também aos meus irmãos, Neto e Lacione, pela amizade, apoio e carinho.

Ao meu orientador, Prof. Edward, pelo ensino, excelência na orientação, paciência e também pelo apoio e encorajamento em momentos de tristeza e dor.

Agradeço também aos professores do PROEE, em especial, Jugurta, Eduardo e Elyson, pelo ensino, apoio e ajuda em momentos de dificuldade.

Agradeço ainda, aos colegas discentes do programa, em especial a Marcos, pela amizade, apoio e ajuda em diversos momentos.

Ao Instituto Federal do Acre (IFAC), pelo incentivo à qualificação profissional.

Enfim, agradeço a todos os amigos e irmãos que oraram por mim, incentivaram e contribuíram de alguma forma para que este trabalho fosse realizado.

Resumo da Dissertação apresentada ao PROEE/UFS como parte dos requisitos necessários para a obtenção do grau de Mestre (Me.)

PROJETO, IMPLEMENTAÇÃO E DESEMPENHO DOS ALGORITMOS CRIPTOGRÁFICOS AES, PRESENT E CLEFIA EM FPGA

William Pedrosa Maia

Agosto/2017

Orientador: Prof. Dr. Edward David Moreno Ordonez

O desenvolvimento de sistemas dedicados de criptografia, para aplicações que exigem baixo custo e consumo tem sido enfoque atual de pesquisas. Este trabalho aborda o projeto e análise de desempenho dos algoritmos de criptografia AES-128 (padrão NIST), PRESENT-80 e CLEFIA-128 (padrão ISO/IEC para Criptografia Leve), implementados em FPGA (Basys 3 Artix-7 – tecnologia de 28 nm), utilizando VHDL. Foram analisadas e comparadas as métricas de desempenho: área ocupada no FPGA, velocidade de processamento (Mbps), eficiência (Mbps/slice), eficiência energética (Ws/bit) e consumo de corrente. As métricas foram obtidas através da ferramenta de síntese e implementação em FPGA, *Vivado Design Suites* (Xilinx), e por meio de um protótipo de medição de corrente, que utiliza a placa sensor Adafruit INA219 (sensor da Texas Instruments) e microcontrolador Arduino Uno (Atmega328 - Atmel). Foram analisadas também a representação gráfica do consumo de corrente através do modelo matemático baseado no periodograma de Welch, aplicado sobre as variáveis de consumo de corrente durante o processo de encriptação de dados. Os resultados mostram curvas de corrente que facilitam a identificação e comparação dos algoritmos. Os dados de consumo de área, velocidade processamento e eficiência no FPGA obtiveram desempenho satisfatório, em comparação com outras implementações existentes na literatura, além de fornecer informação relevante para escolha de um algoritmo de criptografia.

Palavras-Chave: Criptografia Leve, AES, PRESENT, CLEFIA, Sistemas Embarcados, FPGAs.

Abstract of Dissertation presented to PROEE/UFS as a partial fulfillment of the requirements for the degree of Master

DESIGN, IMPLEMENTATION AND PERFORMANCE OF CRYPTOGRAPHIC ALGORITHMS AES, PRESENT AND CLEFIA IN FPGA

William Pedrosa Maia

August/2017

Advisor: Prof. Dr. Edward David Moreno Ordonez

Program: Electrical Engineering

The development of dedicated cryptography systems for applications requiring low cost and consumption has been the current focus of research. This work addresses the design and performance analysis of cryptographic algorithms AES-128 (NIST standard), PRESENT-80 and CLEFIA-128 (ISO/IEC standard for Lightweight Cryptography), implemented in FPGA (Basys 3 Artix-7 - 28 nm technology) using VHDL. Performance metrics were analyzed and compared: occupied area in the FPGA, throughput (Mbps), efficiency (Mbps/slice), energy efficiency (Ws/bit) and current consumption. The metrics were obtained through the synthesis and implementation tool in FPGA, Vivado Design Suites (Xilinx), and by means of a current measurement prototype, which uses the Adafruit INA219 sensor board (Sensor from Texas Instruments) and microcontroller Arduino Uno (Atmega328 - Atmel). We also analyzed the graphical representation of current consumption through the mathematical model based on the Welch periodogram, applied on the current consumption variables during the data encryption process. The results show current curves that facilitate the identification and comparison of the algorithms. The data of area consumption, processing speed and efficiency in the FPGA obtained satisfactory performance in comparison with other implementations existing in the literature, besides providing relevant information to choose an algorithm of encryption.

Keywords: Lightweight Cryptography, AES, PRESENT, CLEFIA, Embedded Systems, FPGAs.

Sumário

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Capítulo 1

Introdução

Ná ultima década, houve um acentuado crescimento de dispositivos eletrônicos conectados a internet, isso devido principalmente à evolução da microeletrônica e da computação. De acordo Lucero *et al*. (2016), a quantidade estimada de dispositivos conectados à internet em 2015 era de aproximadamente 15,41 bilhões (incluindo smartphones, tablets e computadores). Ainda de acordo com a Lucero *et al*. (2016), a expectativa para 2020 é que a quantidade de dispositivos conectados à internet chegue a 30,73 bilhões. Esta tendência para uma maior interação de dispositivos na internet é denominada coletivamente como Internet das Coisas (IoT - *Internet of Things*) ou Internet de Objetos (CONVIGTON *et al*., 2013).

Neste universo de equipamentos interconectados, a tendência é o uso cada vez maior de dispositivos eletrônicos portáveis e de tamanho reduzido, para as mais diversas aplicações que venham a facilitar o dia a dia do ser humano, como por exemplo, pequenos sensores sem fio implantados no corpo de pessoas para monitoramento médico remoto, dispositivos instalados em lugares de difícil acesso para coletar informações geológicas ou de fauna e flora, encanamentos de água dotados de pequenos sensores para monitoração e controle de vazão, dentre muitas outras aplicações que venham conectar sistemas e/ou objetos com o objetivo principal de monitoramento, controle e automação, integrando os denominados "Sistemas Inteligentes", ou "Objetos Inteligentes", tendo impacto direto nos mais diversos setores da sociedade, mudando cada vez mais a maneira como conduzimos nosso dia a dia.

No entanto, o emprego de muitos destes dispositivos ainda oferece desafios a projetistas e pesquisadores, sendo que, dependendo da aplicação, existem limitações em termos de tamanho do circuito, memória, processamento, e principalmente de energia, onde o consumo requerido deve ser o menor possível. Um destes desafios é a segurança das informações armazenadas, transmitidas e processadas por estes dispositivos, tendo em vista que casos de ataques de intrusos com o objetivo de acessar informações sigilosas, e até mesmo de invadir e destruir sistemas são cada vez mais comuns.

Atualmente, uma das ferramentas amplamente utilizadas para proteger informações sigilosas é a proteção por criptografia, onde ocorre uma transformação de um texto legível em ilegível, protegendo assim a informação original, caso algum intruso tenha acesso aos dados. De acordo com Moreno *et al*. (2005) um algoritmo de criptografia é uma sequência de procedimentos que envolvem uma matemática capaz de cifrar e decifrar dados sigilosos.

Para manter a privacidade, algoritmos criptográficos fortes são necessários para impedir sistemas integrados de vazar informações confidenciais. Contudo, de acordo com

Manifavas *et al*. (2014) algoritmos criptográficos tradicionais são computacionalmente inviáveis para dispositivos com fortes restrições computacionais e/ou em sistemas embarcados de baixo consumo, como alguns dos dispositivos mencionados para aplicações em Internet das Coisas (IoT), devido os mesmos requererem uma quantidade de recursos além do que tais dispositivos dispõem para implementá-las. Moreno *et al*. (2015) aponta para implementações de soluções criptográficas específicas (leves) para aplicações em IoT. Estas implementações podem ser realizadas tanto em *hardware*, através de Circuitos Integrados de Aplicação Específica (ASIC) e Circuitos programáveis do tipo FPGAs; ou em *software*, através de *firmware* de sistemas embarcados comandados por microcontroladores de baixo custo de 4, 8 ou 16 *bits*.

Porém, para ambas as implementações há limitação de recursos computacionais. Em *hardware*, por exemplo, existe limitação no tamanho físico do circuito integrado e consumo de energia. Enquanto que implementações em *software* têm limitação na quantidade de memória tanto dinâmica (RAM) quanto só de leitura (ROM, *flash*) e também de consumo de energia.

Desta forma, tendo em vista a necessidade de solucionar o problema de segurança dos dados processados por dispositivos e sistemas embarcados de baixo consumo, algoritmos criptográficos especializados devem ser projetados para esse fim. A área da criptografia que trata da segurança nestes tipos de dispositivos é denominada de Criptografia Leve (EISENBARTH *et al*. (2007) e MORENO *et al*. (2015)).

1.1 Justificativa

Muitos trabalhos foram desenvolvidos com foco em algoritmos criptográficos com nível aceitável de confiabilidade e que utilizam de um menor espaço de memória e área de implementação quanto possível. Nos últimos anos, por exemplo, foram propostos vários algoritmos criptográficos leves do tipo simétricos e assimétricos, dentre os quais se destacam o CLEFIA (SHIRAI *et al*., 2007), PRESENT (BOGDANOV *et al*., 2007), HIGHT (HONG *et al*., 2006), HUMMINGBIRD (ENGELS *et al*., 2010), KATAN e KTANTAN (DE CANNIERE *et al*., 2009), KLEIN (GONG *et al*., 2012), LED (GUO *et al*., 2011), PICCOLO (SHIBUTANI *et al*., 2011), dentre outros. Outros trabalhos também foram desenvolvidos objetivando versões compactas e eficientes do Padrão Avançado de Criptografia (AES - *Advanced Encryption Standard*) (BOGDANOV *et al*., 2014; KUNDI *et al*., 2009; ZHANG *et al*., 2015).

A Organização Internacional para Padronização (ISO) em conjunto com a Comissão Eletrotécnica Internacional (IEC), através da normatização ISO/IEC 29192-2:2012, especificou duas cifras de bloco adequadas para aplicações que exigem implementações criptográficas leves: PRESENT - uma cifra leve de bloco, com um tamanho de bloco de 64 *bits* e um tamanho de chave de 80 ou 128 *bits*; CLEFIA: uma cifra leve de bloco, com um tamanho de bloco de 128 *bits* e um tamanho de chave de 128, 192 ou 256 *bits*.

Atualmente, implementações de algoritmos criptográficos leves diretamente em *Hardware* Embarcado tem sido enfoque principal de pesquisas na área, principalmente devido a um melhor desempenho destes quando comparado às soluções em *software*.

Moreno *et al*. (2005) comparando algumas implementações em *software* e em *hardware*, já indicava essa tendência para criptografia em *Hardware*, através do uso de circuitos programáveis (FPGAs), tecnologia acessível e que diminui de forma significativa o tempo e custo de realização de projetos e protótipos.

Yalla e Kaps (2009) realizaram em seu trabalho, implementações dos algoritmos leves PRESENT e HIGHT em plataforma FPGA (usando um chip da família Spartan 3), destacando o bom desempenho dos mesmos quando comparados com outras cifras leves implementadas por outros autores na mesma plataforma.

Comparações de desempenho dos algoritmos criptográficos leves recomendados pela norma ISO/IEC 29192-2:2012, PRESENT e CLEFIA, com versões compactas do algoritmo AES foram realizadas por Hanley e O'Neill (2012), utilizando as plataformas FPGA (Xilinx Virtex II e Virtex-5), obtendo dados importantes de área de implementação e eficiência destes algoritmos em *Hardware*, bem como fazendo uma comparação direta com o algoritmo AES, porém, dados de consumo de energia nos processos de encriptação e decriptação poderiam ser informados.

Uma arquitetura de *hardware* de multicriptografia baseado nas cifras padrões CLEFIA e AES é proposta por Resende e Chaves (2015). A implementação foi realizada em plataforma FPGA (Virtex-5 e Virtex-6), e foi desenvolvida através da combinação das estruturas dos algoritmos CLEFIA e AES, obtendo uma estrutura unificada (um único chip para as duas cifras). Os resultados do trabalho mostram um desempenho satisfatório da cifra dupla, quando comparados a outros trabalhos de *hardware* dedicado de cifra única (com um chip só para o AES e outro só para o CLEFIA).

Pelo exposto, a segurança dos dados armazenados, processados e transmitidos por dispositivos ou sistemas embarcados com limitação de recursos computacionais e de consumo de energia torna-se um desafio, sendo que os desenvolvimentos de *hardwares* dedicados de criptografia apresentam-se como uma boa ferramenta para a solução do problema, justificando assim um trabalho na área.

1.2 Objetivos

O objetivo principal desta dissertação de mestrado é o projeto e análise de desempenho de um *hardware* de criptografia dos algoritmos leves PRESENT e CLEFIA (Padrão ISO/IEC) e do algoritmo AES (padrão NIST), em plataforma FPGA, utilizando VHDL.

Objetivos Específicos

- Implementar e analisar o desempenho individual dos algoritmos PRESENT, CLEFIA e AES em FPGA;
- Comparar as respectivas implementações utilizando as métricas de desempenho: área ocupada no FPGA, velocidade de processamento (Mbps), eficiência (Mbps/slice), eficiência energética (Ws/bit) e consumo de corrente;
- Desenvolver um protótipo de medição de consumo de corrente, para comparar

as implementações destes algoritmos;

- Comparar as implementações realizadas com algumas existentes na literatura;

1.3 Metodologia

Para a consecução dos objetivos propostos neste trabalho, foi realizada uma revisão sistemática dos autores que tratam de soluções criptográficas em *hardware* FPGA, dos algoritmos AES, PRESENT e CLEFIA.

Com base nos pressupostos teóricos, os algoritmos foco deste trabalho foram devidamente implementados (primeiramente de forma individual) no FPGA utilizado neste trabalho, principalmente para fins de comparações de desempenho. Posteriormente um protótipo de medição de corrente foi desenvolvido, visando a obtenção de mais uma métrica de análise de desempenho e comparação entre as implementações. Alterações e otimizações nos códigos em VHDL dos algoritmos foco deste trabalho foram realizadas, com o objetivo de se obter uma estrutura unificada (de multicriptografia) para implementação em *hardware*. Também foram realizadas a análise de desempenho e comparação de resultados através de diversas métricas.

1.3.1 Materiais utilizados

Para a implementação do proposto na Metodologia, foram utilizados os seguintes recursos:

- **Placa FPGA (Basys 3™)**: plataforma FPGA para pesquisas e projetos de circuitos digitais.

- **Vivado™ *Design* Suite (v. 2016.3)**: ferramenta de *software* para programação em VHDL, simulação e síntese do código;

- **Placa Arduino Uno**: plataforma open sorce com microcontrolador Atmega328, utilizada no protótipo de medição de dados de consumo energético;

- **Placa Adafruit INA219**: placa com transdutor de corrente e tensão, utilizada como componente principal do protótipo de medição de corrente e tensão;

- **IDE Arduino (v. 1.8.2)**: *software* utilizado para a coleta e exibição dos dados de consumo de corrente e tensão, oriundos do protótipo de medição;

- **Matlab (v. R2017a)**: *software* matemático utilizado no tratamento e exibição de dados de consumo de energia;

- **PC (*Personal Computer*)**: necessária para a realização de pesquisas e utilização das ferramentas anteriormente citadas;

1.4 Organização da Dissertação

A dissertação está organizada em 7 capítulos:

O capítulo 1 traz a introdução sobre o tema proposto, justificativa, objetivos do trabalho e metodologia de desenvolvimento. O capítulo 2 descreve alguns conceitos sobre Criptografia Leve e os algoritmos criptográficos escolhidos para a implementação neste trabalho. O capítulo 3 aborda uma análise do Estado da Arte sobre implementações em *hardware* de algoritmos leves e compactos, com destaque para os algoritmos AES, PRESENT e CLEFIA. A metodologia proposta neste trabalho é descrita no capítulo 4. Os capítulos 5 e 6 trazem os detalhes das experimentações realizadas, análises de desempenho e comparação com outros trabalhos. Por fim, o capítulo 7 apresenta as conclusões do trabalho, destacando as principais contribuições e sugestões de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

As subseções a seguir têm por finalidade apresentar os conceitos de Criptografia Leve e algoritmos leves com foco em implementações em *hardware*, características dos algoritmos criptográficos escolhidos para o desenvolvimento deste trabalho.

2.1 Criptografia Leve

Criptografia leve é uma área da criptografia centrada no desenvolvimento de esquemas criptográficos confiáveis para dispositivos com recursos restritos no fornecimento de energia, conectividade, *hardware* e *software* (MANIFAVAS *et al*., 2014)

Uma importante relação entre implementações de algoritmos tradicionais em comparação com algoritmos leves está no tamanho da chave utilizada. De acordo com Moreno et al. (2005), em geral, quanto maior o tamanho da chave, maior é a segurança da cifra, ou seja, se torna mais difícil e custoso um ataque de força bruta bem-sucedido. Os algoritmos leves, em sua maioria, utilizam tamanhos de chaves menores do que algoritmos tradicionais, isto se deve principalmente ao fato de que, a utilização de chave maiores, requer maior tempo de processamento e/ou maior consumo de recursos, que é incompatível com as aplicações que requerem algoritmos leves.

Em *hardware* (ASIC - *Application Specific Integrated Circuit*), a métrica dominante utilizada na maioria das propostas de algoritmos leves é o número de Equivalente de Porta (GE – *Gate equivalent*), medida relacionada com a área de circuito integrado que a implementação da lógica do algoritmo (também chamada de cifra) requer, sem comprometer a exigência de fortes propriedades de segurança, ou seja, não vulnerável aos principais tipos de ataques. O número de GEs é obtido através da divisão da área de silício utilizada para uma cifra com uma determinada biblioteca de células padrão pela área de uma porta NAND de duas entradas (Batina *et al*., 2013). Outra métrica também utilizada em trabalhos na área utilizando *hardware* configurável FPGA é relacionada com os *Slices* (fatias) do dispositivo, onde cada *slice* é composto por *look-up-tables* (tabelas-de-consulta - *LUTs*) e flip-flops, sendo que a quantidade de *slices* difere de acordo com a família/fabricante do chip. A velocidade de processamento para encriptação e decriptação (eficiência) também é uma medida importante para análise de desempenho do algoritmo. Um exemplo de restrição de área de implementação são as *tags* (etiquetas) RFID, que de acordo com Bansod *et al*. (2015) possuem uma área para implementação entre 1.000-10000 GEs, sendo que apenas 300-2100 GEs estão disponíveis para os aspectos de segurança. Embora esses métodos para medir eficiência de uma arquitetura leve sejam úteis, os mesmos não

respondem a todas as questões relativas a cifras leves, sendo importante considerar outros aspectos, como medidas de potência, energia.

Em *software*, as métricas dominantes são basicamente quantidade de memória requerida para a implementação (RAM, ROM ou *flash*) e também velocidade de processamento (EISENBARTH *et al*., 2007).

Vale ressaltar que o Instituto Nacional de Padrões e Tecnologia – NIST (*National Institute of Standards and Technology*), publicou em 2017 um relatório que apresenta estratégias de padronização para Criptografia Leve (NISTIR 8114). O relatório descreve uma visão geral sobre Criptografia Leve, resume alguns algoritmos leves publicados, bem como apresenta estratégias para a padronização.

Nas próximas seções são apresentados alguns modelos de algoritmos criptográficos leves.

2.2 Criptografia Leve simétrica

A criptografia simétrica, ou criptografia de chave privada, utiliza uma mesma chave para cifrar (encriptar) e decifrar (decriptar) a mensagem, como pode ser visualizado na figura 2.1. Desta forma, o remetente (Alice), insere em um determinado sistema com um algoritmo de criptografia a mensagem (m) e a chave (k), onde o processo de criptografia é realizado (E (m) = c), em que c representa a mensagem criptografada. Já o destinatário (Bob), de posse da chave (k), utilizando o algoritmo de criptografia comum (entre Alice e Bob) realiza a decriptação de c, obtendo a mensagem original (D (c) = m). Neste sistema de criptografia, a chave deve ser mantida em segredo, ou seja, deve ser enviada através de um canal seguro entre o remetente e destinatário. Já a mensagem criptografada (c), pode ser transmitida através de um canal não seguro, sendo passível de interceptação por algum agente com fins maliciosos (Intruso).

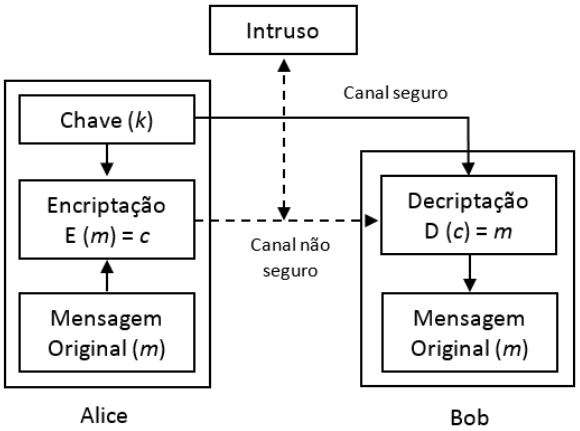


Figura 2.1. Exemplo de criptografia simétrica ou de chave privada.

(Fonte: Adaptado de Menezes *et al*. (1996).

A criptografia assimétrica, principalmente devido à rapidez de processamento, é amplamente utilizada em sistemas que necessitam de criptografia leve.

Assim como em sistemas criptográficos tradicionais, existem basicamente três grupos de criptografia leve do tipo simétrica: cifras de bloco e cifras de fluxo e também funções de hash. Estes tipos de criptografia são abordados a seguir.

2.2.1 Cifras leves de bloco

As cifras de bloco são as mais utilizadas por sistemas criptográficos modernos. Operam sobre bloco de dados (geralmente de 128 *bits*) e utilizam um modo de realimentação para suportar a encriptação de mensagens com tamanhos maiores do que o tamanho de bloco padrão da cifra. O modo mais comum de realimentação é a cifragem de blocos por encadeamento (CBC - *Cipher Block Chaining*), onde é realizada uma operação XOR (ou exclusivo) entre o bloco a ser cifrado e o bloco cifrado anteriormente. No caso do primeiro bloco, onde não há bloco cifrado anteriormente, é realizada uma operação XOR com um vetor de inicialização (IV - *initialization vector*), gerado de forma pseudoaleatória, de forma a evitar que uma mesma mensagem encriptada diversas vezes resulte em textos cifrados idênticos.

Boa parte das cifras leves de bloco foi derivada de cifras tradicionais, através de estudos e aplicações de métodos de redução de caixas de substituição (S-box) e de expansão e adição de chave, baseada na adição de portas XOR durante a execução das rodadas do algoritmo, dentre outras técnicas que visam à redução do consumo de recursos do dispositivo sem comprometer os requisitos de segurança. Algumas cifras leves de bloco são apresentadas a seguir.

As cifras leves DESL e DESXL, são exemplos de variantes da cifra tradicional DES. Segundo Eisenbarth *et al*. (2007) a cifra resultante DESL requer aproximadamente 20% a menos de área do que o DES (1850 GEs contra 2310 GEs). Já DESXL, resultante do aperfeiçoamento de DESL e com um nível de segurança mais elevado, requer 2170 GEs e criptografa um texto simples dentro de 144 ciclos de clock. Uma alternativa ao método de modificação de uma cifra existente para obtenção de uma variante leve é a concepção de uma estrutura completamente nova, projetada especificamente para aplicações restritas.

PRESENT é uma cifra leve de bloco projetada por Bogdanov *et al*. (2007), para uma implementação eficiente em *hardware*. Seu tamanho de bloco é de 64 *bits*, e suporta chaves de 80 ou 128 *bits*. PRESENT opera com uma rede de substituição-permutação (SPN - *Substitution-Permutation Network*) com 31 rodadas e de acordo com Bansod *et al*. (2015) suas diversas variantes requerem de 2520 a 3010 GEs para fornecer níveis de segurança adequados.

CLEFIA é uma cifra com tamanho de bloco de 128 *bits* e usa chaves de 128, 192 e 256 *bits*, que é compatível com os parâmetros do Padrão de Criptografia Avançado (AES). CLEFIA foi projetado pela SONY com alta eficiência em *hardware* e *software* (SHIRAI *et al*., 2007).

PRESENT e CLEFIA são os algoritmos padronizados pela norma ISO/IEC 29192-2:2012 e recomendados para aplicações que requerem criptografia leve, por isso os mesmo fazem parte dos algoritmos de criptografia foco deste trabalho.

HIGHT possui bloco de 64 *bits* com comprimento de chave de 128 *bits*. Usa a estrutura de construção *Feistel* generalizado com 32 rodadas com operações simples, como XOR, rotação bit a bit e modulação em grupo de 28 elementos. É adequado para aplicações de baixo-custo, baixa-potência e implementações leves (YALLA e KAPs, 2009).

KATAN e KTANTAN pertencem à família de algoritmos leves de cifragem de blocos projetados para serem eficientes em *hardware*. Usam chaves de 80 *bits* e blocos de tamanho de 32, 48 ou 64 *bits*. Katan com blocos de 64 *bits* e chaves de 80 *bits* precisa de 1054 GEs, enquanto o Ktantan equivalente requer apenas 688 GEs (MANIFAVAS *et al*., 2014).

SIMON e SPECK são algoritmos considerados ultra-leves (requer até 1.000 GEs de área) desenvolvidos pela Agência Nacional Americana (NSA). Ambas as cifras apresentam um bom desempenho em *software* e *hardware*. Suportam blocos de diversos tamanhos, de 32, 48, 64, 96 ou 128 *bits*. Cada tamanho de bloco suporta até três tamanhos de chaves, que podem ser de 64, 72, 96, 128, 144, 192 ou 256 *bits*. Devido à variedade de tamanho de blocos e chaves, os mesmos são indicados para aplicações em sistemas extremamente restritos, até os sistemas menos restritos e que necessitam de um nível de segurança adequado. As implementações em *hardware* do Simon e do Speck com blocos de 48 *bits* e chaves de 96 *bits* precisam respectivamente de 763 e 884 GEs de área de circuito integrado BEAULIEU *et al*., 2013).

Muitas outras cifras leves de bloco foram publicadas recentemente, tais como RC5, TWINE, KLEIN, LED, Lblock, PUFFIN-2, PICCOLO, NOEKEON e ITUbee, apresentando resultados satisfatórios para implementações de baixo consumo, porém não são alvo de estudo deste trabalho.

2.2.2 Cifras leves de fluxo

Um tipo alternativo de cifras simétricas são as cifras de fluxo. Os algoritmos de fluxo utilizam a técnica de criptografar a mensagem *bit* a *bit*, em um fluxo contínuo, através da combinação (geralmente uma operação XOR) dos *bits* da mensagem com *bits* da chave, obtidos a partir de um gerador de números pseudo-aleatórios (Moreno *et al*., 2005).

De acordo com Manifavas *et al*. (2014), para implementações em *hardware*, as cifras de fluxo Grain e Trivium são os mais indicados para aplicações em dispositivos com recursos restritos. Grain possui tamanho de chave de 80 ou 128 *bits*, e vetor de inicialização (IV) de 64 *bits* e requer cerca de 1300 GEs para implementação. Trivium utiliza chave de 80 *bits*, e um vetor de inicialização de 80 *bits* e requer cerca de 2600 GEs.

Apesar do esforço de evolução no domínio das cifras leves de fluxo, elas permanecem inferiores em comparação com as cifras leves de bloco, principalmente devido à vulnerabilidade a ataques (BANSOD *et al*., 2015).

2.2.3 Funções leves de hashing

Uma função criptográfica que gera uma saída de tamanho fixo (geralmente 128 a 256 *bits*) independentemente do tamanho da entrada é denominada de função de hashing. A saída recebe o nome de hash, como é comumente chamada este tipo de função.

Diversas funções leves de hash criptográfico têm sido propostos nos últimos anos. Muitos trabalhos se dedicaram em construir cifras leves de hashing com base no *design* do PRESENT, como por exemplo, C-PRESENT (requer 4600 GEs), H-PRESENT (requer 2330 GEs) e PRESENT -DM (1600 GEs) (MANIFAVAS *et al*., 2014).

SPONGENT é da família de algoritmos leves de hash criptográfico mais conhecidas até o momento. De acordo com Jungk (2014) Spongent utiliza as mesmas operações de uma rodada da rede de substituição-permutação (SPN) do PRESENT, com função denominada esponja, que utiliza permutação pseudo-aleatória. Os tamanhos de hash suportados são de 88, 128, 160, 224 ou 256 *bits* e apresentam uma resistência à colisão variando entre 40 a 128 *bits* de acordo com o tamanho do hash gerado. A implementação em *hardware* do Spongent com hash de 128 *bits* e com resistência à colisão de 64 *bits* precisa de pelo menos 1060 GEs.

Outras novas funções leves de hash leves com construções de esponja são SQUASH (6328 GEs), GLUON (2071 GEs), QUARK (1379 GEs), PHOTON (1120 GEs) sendo que a última utiliza função esponja baseada no AES.

2.3 Criptografia leve assimétrica

Algoritmos assimétricos ou de chave pública, pertencem a uma classe de algoritmos de criptografia que utilizam duas chaves (par), sendo uma secreta (ou privada) e outra pública. Quando a informação é codificada (processo de encriptação) com uma das chaves, somente a outra chave do par pode decodificá-la (processo de decriptação), sendo que a ordem das chaves a serem utilizadas depende da aplicação, se confidencialidade ou autenticação, integridade e não-repúdio.

A figura 2.2 exibe um exemplo de criptografia assimétrica para fins de confidencialidade (troca de mensagens entre Alice e Bob), onde a mensagem original (*m*) é encriptada com a chave pública (*e*) e decriptada com a chave privada (*d*), sendo que *c* representa a mensagem criptografada. Observa-se que nesta aplicação, tanto a mensagem criptografada (*c*), quanto a chave pública (*e*), podem ser transmitidas por meio de um canal não seguro, sendo passível de receptação por intrusos, porém, a mensagem é decriptada somente com o conhecimento da chave privada.

Estes tipos de algoritmos também podem ser adaptados para aplicações em dispositivos com as limitações de recursos, porém, devido os mesmos serem mais exigentes computacionalmente, não são tão explorados para aplicações leves quanto algoritmos simétricos.

Entre os algoritmos assimétricos, três famílias têm recebido destaque em aplicações de criptografia leve: Criptografia de Curva Elíptica (ECC - *Elliptic curve cryptography*), RSA (*Rivest-Shamir-Adleman*) e logaritmos discretos. Eisenbarth *et al*. (2007) considera

ECC a família mais atraente para sistemas embarcados por causa de seus comprimentos de operando menores e requisitos computacionais relativamente baixos.

RSA é um algoritmo bem conhecido em criptografia assimétrica e suporta chaves de tamanho variável, as mais usadas variam de 1.024 a 4.096 *bits*. O mesmo é usado como referência para os vários sistemas de criptografia assimétrica propostas por pesquisadores. No entanto devido à forte exigência em *hardware* levou os pesquisadores a procurar outros algoritmos para aplicações em dispositivos limitados.

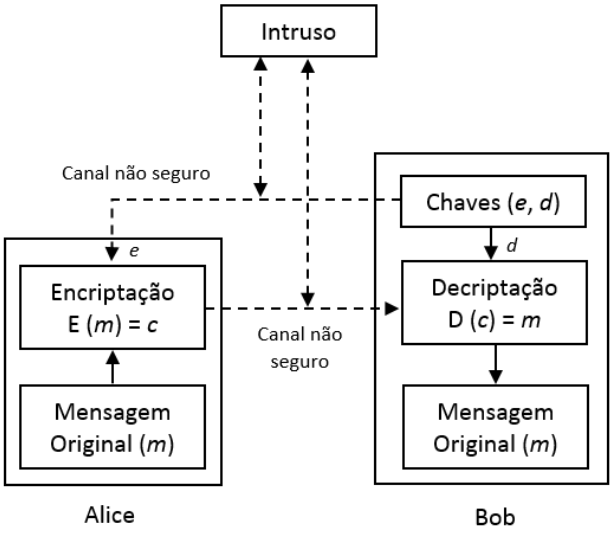


Figura 2.2. Exemplo de Criptografia assimétrica ou de chave pública.

(Fonte: Adaptado de Menezes *et al.* (1996).

Criptografia de Curva Hiperelíptica (HECC - *Hyperelliptic curve cryptography*) também é considera atraente para aplicações em *hardware* embarcado. HECC é uma generalização de Curvas Elípticas, e de acordo com Manifavas *et al*. (2014) possui tamanhos de operandos e desempenho melhor que em ECC.

Apesar da relevância de algoritmos assimétricos na área de segurança computacional, nesta dissertação de mestrado o foco será dado para criptografia simétrica leve. A seguir são abordadas características específicas dos algoritmos foco deste trabalho (AES, PRESENT e CLEFIA).

2.4 Algoritmo AES

O Padrão de Criptografia Avançado (AES - *Advanced Encryption Standard*) é fruto de um concurso realizado em 1997 pelo Instituto Nacional de Padrões e Tecnologia – NIST (*National Institute of Standards and Technology*), órgão do governo dos Estados Unidos, com o objetivo de encontrar um substituto para o até então padrão comercial de criptografia, o algoritmo DES, que devido ao desenvolvimento de tecnologias de computação e técnicas de criptoanálise, não conseguiria mais garantir a segurança adequada por muito tempo. O candidato que atendeu os principais requisitos e vencedor do concurso foi o algoritmo Rijndael, que posteriormente ficou conhecido como AES, sendo o mesmo publicado oficialmente pelo NIST em 2001 (Stallings, 2008).

O AES é uma cifra de bloco simétrica, onde o padrão atual opera em bloco de dados de 128 *bits*, que são organizados na forma de matriz quadrada de *bytes* de ordem quatro, denominada *State*, onde a ordenação dos *bytes* dentro da matriz ocorre por coluna. As chaves podem ser parametrizadas em tamanhos de 128, 192 e 256 *bits*. A cada iteração, ou rodada de encriptação sobre cada bloco de dados (essas rodadas podem variar de acordo com o tamanho da chave: 10, 12 e 14 rodadas, para chaves de 128, 192 e 256 *bits* respectivamente), são realizadas várias operações: Substituição de bytes (SubByte), Deslocamento de linhas (ShiftRow), Embaralhamento de colunas (MixColumns) e Adição de Chave da rodada (AddRoundKey), que ocorre sobre a aritmética no corpo finito GF (2^8), conhecida como Campo de Galois (GF - *Galois Field*), para a decriptação dos dados as operações matemáticas são invertidas). Outra operação importante a ser considerada é o processo de geração de subchaves, ou expansão da chave, onde a chave fornecida como entrada é expandida, através de um algoritmo específico, em um vetor de *n* palavras de 32 *bits*, onde o valor de *n* depende do tamanho da chave escolhida (44, 52, 60 palavras para 128, 192 e 256 *bits* de chave respectivamente). A cada rodada, 4 palavras distintas (128 *bits*) servem como chave para a operação de adição de chave (AddRoundKey).

A figura 2.3 exemplifica a estrutura geral do AES-128 (com chave de 128 *bits*).

Em síntese, as operações do processo de encriptação são (Moreno *et. al*., 2005):

- **Substituição de Bytes** (SubByte): os bytes de cada bloco da matriz (estado) são substituídos por seus equivalentes em uma tabela de substituição (S-BOX);

- **Deslocamento de linhas** (ShiftRow): uma permutação simples é realizada, onde os bytes são rotacionados em grupos de quatro bytes;

- **Embaralhamento de Colunas** (MixColumns): nesta etapa são realizadas multiplicações lineares em GF (2^8) sobre cada grupo de quatro bytes, proporcionando assim uma influência de cada byte do grupo sobre todos os outros bytes.

- **Adição de Chave da Rodada** (AddRoundKey): nesta fase, é realizada uma operação XOR bit a bit do bloco atual com uma parte da chave expandida;

É importante ressaltar que a última rodada de encriptação e decriptação (figura 2.3) é diferente das demais, não realizando a operação MixColumns.

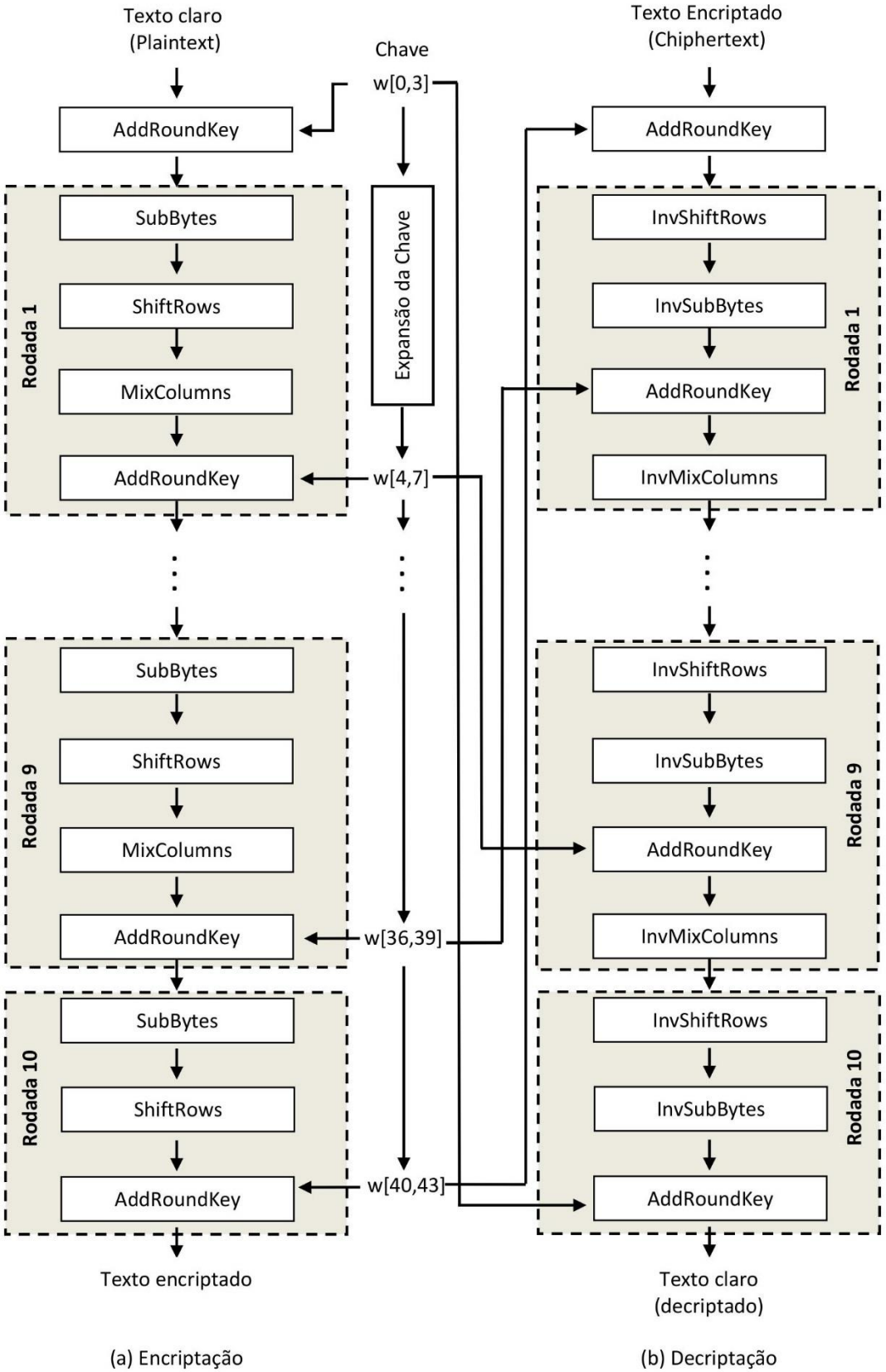


Figura 2.3. Algoritmo de Encriptação e Decriptação AES-128.

Fonte: Adaptado de Stallings (2008).

Como pode ser observado na figura 2.3 para processo de decriptação dos dados, operações inversas são aplicadas ao texto criptografado (InvSubByte, InvShiftRow, InvMixColumuns) bem como a operação AddRoundKey que é a própria inversa devido a propriedade da porta XOR, obtendo assim a mensagem original (texto claro).

Apesar do AES não ser considerado um algoritmo de criptografia leve, muitos trabalhos têm realizado implementações otimizadas do AES-128 (BOGDANOV *et al*., 2014; KUNDI *et al*., 2009; ZHANG *et al*., 2015).com bom desempenho e eficiência, em *hardware* e/ou em *software*, denominadas de AES compacto. Visando principalmente atender a necessidade da segurança dos dados em dispositivos e sistemas embarcados de baixo consumo.

2.5 Algoritmo PRESENT

PRESENT é uma cifra ultraleve de bloco, desenvolvida por Bogdanov *et al*. (2007) especialmente para ambientes com recursos limitados e com melhor eficiencia em *hardware*. Possui tamanho de bloco 64 *bits* e suporta chaves de 80 ou 128 *bits*. Possui uma estrutura de rede de substituição-permutação (SPN - *Substitution-Permutation Network*) e consiste em 31 rodadas. A figura 2.4 ilustra a descrição algorítmica e do fluxo de dados no processo de encriptação do PRESENT.

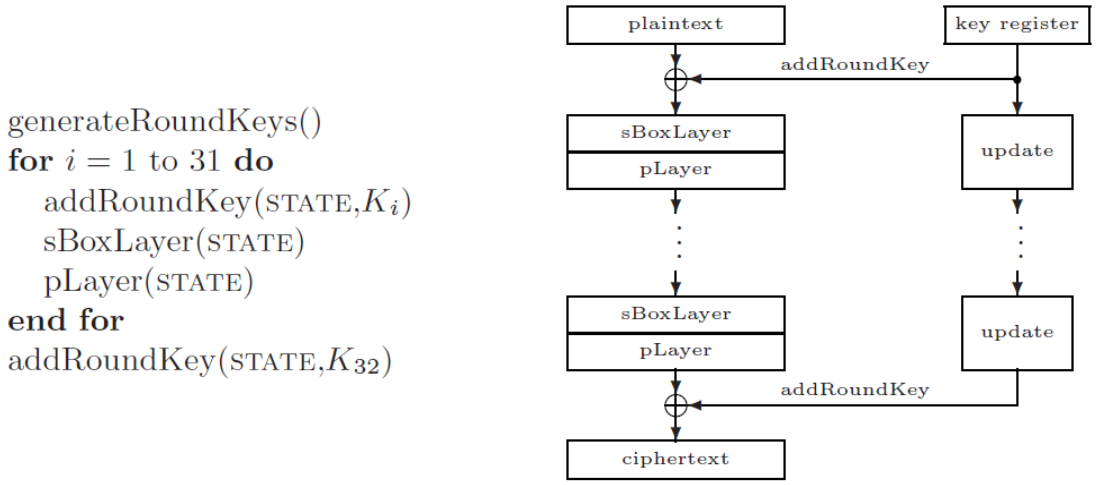


Figura 2.4. Descrição algorítmica e do fluxo de dados de encriptação do PRESENT.

Fonte: Bogdanov *et al*. (2007).

Cada uma das 31 rodadas (rounds) consiste de uma operação XOR para introduzir uma chave (addRoundKey) K_i para $1 \leq i \leq 32$, onde as repetidas iterações visam garantir um alto grau de segurança do bloco cifrado. O bloco de substituição (sBoxLayer), representado na figura 2.4, realiza operações não lineares de substituição, utilizando uma caixa de substituição 4 *bits* para 4 *bits* (S-box). Permutações lineares *bit* a *bit* são realizadas pelo bloco de permutação (pLayer), através de uma tabela de consulta. Detalhes das operações de encriptação do PRESENT-80 são descritas a seguir:

- **AddRoundKey** (adição de chave da rodada): neste processo é realizado uma operação XOR da chave da rodada K_i onde K_i=k_{63}^i ... k_0^i para $1 \leq i \leq 32$, obtida através de um procedimento específico de escalonamento de chave (descrito posteriormente), com o Estado Atual, denominado ESTATE, no caso do primeiro estado o ESTATE será o texto claro (plaintext) onde b_j= b_{63}... b_0 para $0 \leq j \leq 63$, sendo representado pela equação abaixo:

$$b_j \rightarrow b_j \oplus k_j^i \qquad \text{(Equação 2.1)}$$

- **sBoxLayer** (camada de substituição): nesta etapa é usada uma caixa de substituição (S-box) de 4 *bits* para 4 *bits* (expressa em hexadecimal na tabela 2.1), onde os 4 *bits* de entrada $(x_3||x_2||x_1||x_0)$ são representados por *x* e os quatro *bits* de saída são representados por S(*x*)=$(S_3(x)||S_2(x)||S_1(x)||S_0(x))$. Para a realização desta operação, o STATE (atual) b_{63}... b_0 (64 *bits)*, é subdividido em 16 palavras (words) de 4 *bits* w_{15}... w_0 sendo que a palavra de entrada w_i é obtida através da equação

$$w_i = b_{4*i+3}||b_{4*i+2}||b_{4*i+1}||b_{4*i+0} \qquad \text{(Equação 2.2)}$$

com $0 \leq i \leq 15$, gerando uma saída S(w_i) que, após obtido o valor equivalente na S-box, irá compor uma palavra do próximo STATE.

Tabela 2.1 Caixa de substituição (S-box) do Present (encriptação).

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Fonte: Bogdanov *et al*. (2007).

- **pLayer** (camada de permutação): realiza uma permutação simples *bit* a *bit*, conforme a tabela 2.2. O *bit* da posição *i* do STATE é movido para a posição do *bit P(i)*.

Tabela 2.2: Caixa de permutação (P-box) do Present (encriptação).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Fonte: Bogdanov *et al*. (2007).

- **Geração de RoundKeys** (geração de subchaves): o processo de geração de subchaves (chaves da rodada), que são utilizadas a cada iteração (addRoundKey)

funciona primeiramente através do armazenamento da chave fornecida pelo usuário em um registrador *K* (key register) e representada por $K=k_{79}k_{78} \ldots k_0$ (neste caso 80 *bits*). Na rodada *i* os 64 *bits* mais à esquerda do atual valor do registrador *K* são extraídos para compor o round key *i*, sendo $K_i=k_{79}k_{78} \ldots k_{16}$. Nas próximas rodadas a chave *K* é atualizada conforme os passos a seguir:

1. $[k_{79}k_{78} \ldots k_1k_0] = [k_{18}k_{17} \ldots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus$ round_counter

A figura 2.5 ilustra o processo de geração de subchaves. As posições dos 61 *bits* mais a esquerda (mais significativos) de *K* (atual) são invertidas com o restante dos *bits* (*bits* menos significativos), formando temporariamente o próximo estado de $K(i+1)$, sendo que uma operação de substituição (S-box) é realizada com os 4 *bits* mais significativos de $K(i+1)$ e uma operação XOR dos *bits* $(k_{19}k_{18}k_{17}k_{16}k_{15}]$ de $K(i+1)$ é realizada com o bit menos significativo que representa a rodada. Sendo que a chave da rodada (RoundKey), será K_i, será composta dos 64 *bits* mais à esquerda da chave *K* atualizada.

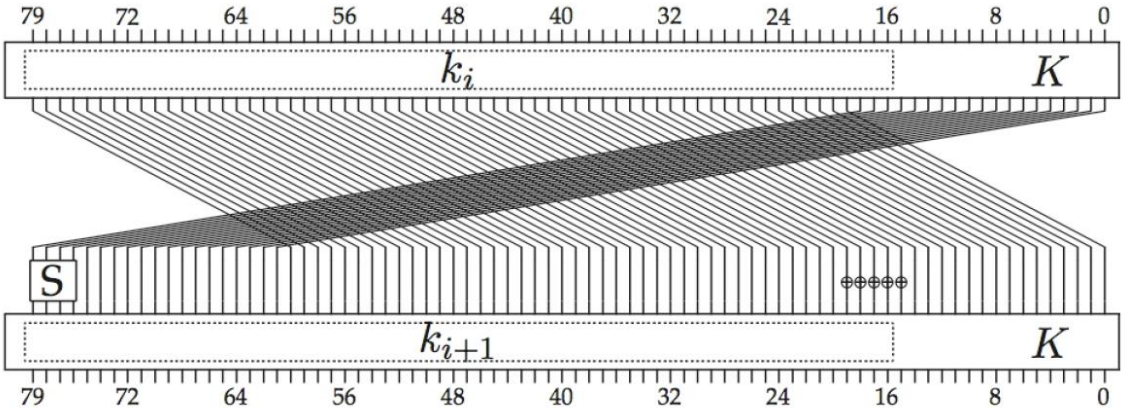


Figura 2.5: Processo de geração de RoundKey do PRESENT-80

Fonte: POSCHMANN (2009).

O processo de permutação e substituição (SPN) do PRESENT é ilustrado na figura 2.6. A descrição algorítmica e de fluxo de dados para o processo de decriptação dos dados é representada na figura 2.7. São executados processos invertidos da camada de substituição (invSBoxlayer), e da camada de permutação (invPLayer), com tabelas específicas para estes processos. Também é invertida a ordem de adição de chaves da rodada (addRoundKey), obtendo assim o texto claro.

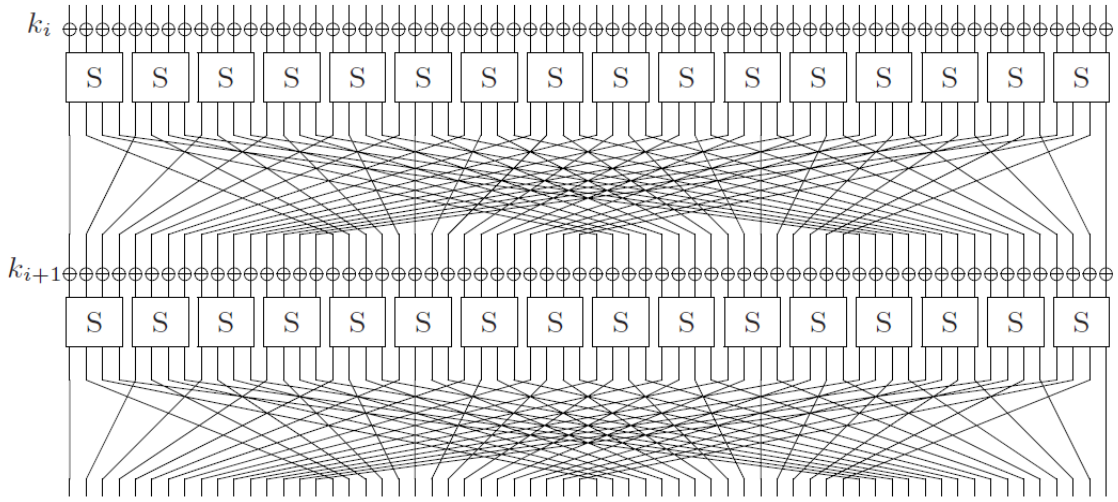


Figura 2.6: Ilustração da rede de permutação e substituição (SPN) do PRESENT

Fonte: POSCHMANN (2009).

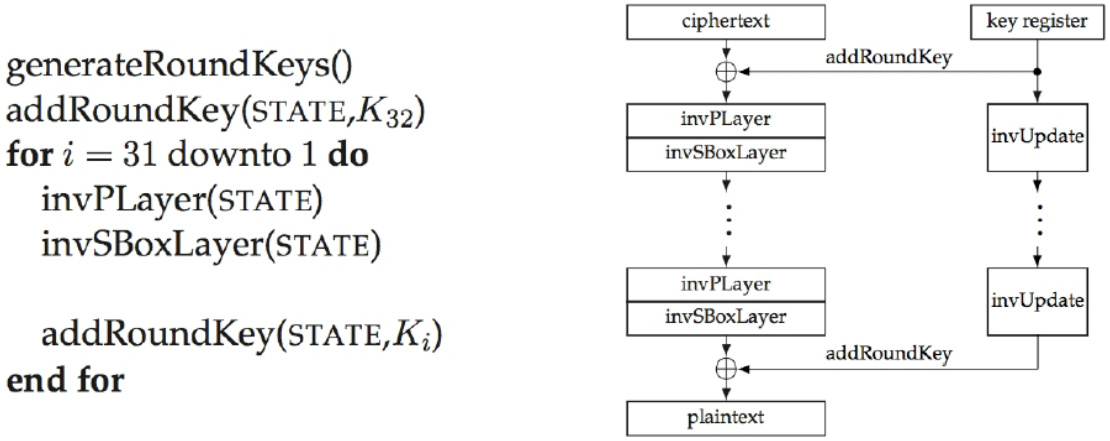


Figura 2.7: Descrição algorítimica e de fluxo de dados de decriptação do PRESENT

Fonte: Bogdanov *et al*. (2007).

2.6 Algoritmo CLEFIA

CLEFIA é uma cifra leve de bloco desenvolvida por projetistas da Sony Corporation (SHIRAI *et al*., 2007) para aplicações em dispositivos de baixo consumo e com boa eficiência em *hardware* e *software*. Opera com blocos de dados de 128 *bits* e comprimentos de chaves de 128, 192 e 256 *bits*, parâmetros que são compatíveis com a cifra padrão AES. A estrutura principal do CLEFIA consiste de uma generalização da rede de Feistel, com duas funções (F-functions) de 32 *bits* por rodada, conforme Figura 2.8. Os blocos de dados de 128 *bits* são organizados em uma matriz de 4 palavras de 32 *bits* e processados ao longo das *N* rodadas de instruções, que podem ser 18, 22 ou 26 rodadas, para chaves de 128, 192 ou 256 respectivamente. As instruções realizadas pelo CLEFIA são descritas

a seguir

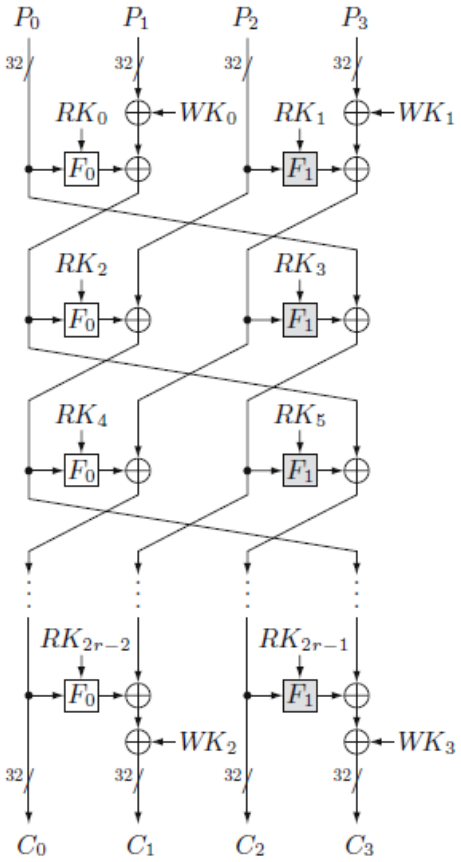


Figura 2.8: Fluxo de dados de encriptação do CLEFIA

Fonte: SHIRAI *et al*. (2007).

- **Funções-F**: para a realização da encriptação e decriptação dos dados, duas funções F diferentes são empregadas (F_0 e F_1). Estas funções realizam cálculos aritméticos (acréscimos) no corpo finito GF(2^8) entre os dados e chaves de cada rodada; operações de substituição, através das caixas de substituição S_0 e S_1 (S-boxes), e difusão matricial (M_0 e M_1) também são realizados com o objetivo de garantir um nível adequado de segurança e resistência a ataques.
  As funções F_0 e F_1 são representadas na figura 2.9.

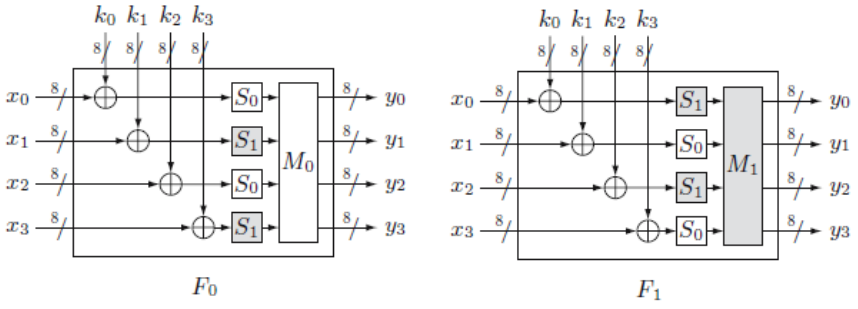


Figura 2.9: Representação das funções-*F* do CLEFIA

- **Processamento dos dados:** Primeiramente o bloco de dados de 128 *bits* é organizado em uma matriz de 4 palavras de 32 *bits* (P_0 , P_1 , P_2 , P_3). Então o processo dos dados (encriptação e decriptação) é realizado em uma sequência de rodadas, que é variável de acordo com o tamanho da chave adodata no processo (18, 22 ou 26 rodadas, para chaves de 128, 192 ou 256 respectivamente), consistindo em operações realizadas pelas funções-F e adições XOR. Quatro subchaves de 32 *bits*, denominadas chaves de clareamento (WK - *Whitening Keys*) obtidas através da chave original, são adicionadas ao processo, duas antes (WK_0 e WK_1) e duas ao final das rodadas (WK_2 e WK_3). Durante as iterações, as funções-*F* realizam suas operações com as chaves da rodada (*RK* – *roundKeys*), obtidas através de um processo específico de expansão de chave, juntamente com os dados de cada rodada. Operações XOR são realizadas com os dados de saída de cada função-*F* com os dados oriundos de processamentos anteriores, conforme ilustrado na figura 2.8.

- **Programação de Chave (expansão de chave):** o processo de expansão da chave fornecida pelo usuário, necessário para alimentar os processos durante as iterações do algoritmo, é definido como Programação de Chave *(Key Scheduling*). Caso a chave inicial utilizada for de 128 *bits*, as chaves de clareamento *(WK*) são obtidas diretamente da chave original, dividindo a chave de 128 *bits* em quatro palavras de 32 *bits*, compondo as quatro chaves de clareamento *(WK*). Caso sejam adotadas chaves de 192 ou 256 *bits*, as chaves de clareamento são obtidas através de outro processo *(WK*). As chaves da rodada *(RK*) são obtidas basicamente através de dois processos. No primeiro a chave original passa por uma função de processamento (GFN), que utiliza a mesma lógica estrutural da encriptação do CLEFIA (figura 8), sendo que para chave de 128 *bits* a função de geração de *RK* terá 4 ramificações (GFN_4), e para chaves de 192 ou 256 *bits* é usada uma função com 8 ramificações (GFN_8). Porteriormente é usada uma função de dupla-permutação *(DoubleSwap Function*), para permutação dos *bits* conforme figura 2.10, e constantes adicionais são acrescentadas. Através destes processos são obtidas as chaves de clareamento *(WK*) e chaves da rodada *(RK*), necessárias nas rodadas de encriptação e decriptação dos dados.

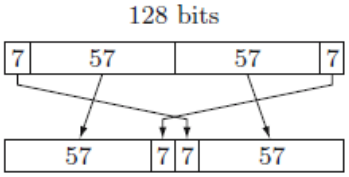


Figura 2.10: Função de dupla-permutação (*DoubleSwap*)

Fonte: SHIRAI *et al*. (2007).

Devido às características da estrutura da Rede de Feistel utilizada pelo CLEFIA, o processo de decriptação dos dados é semelhante ao de encriptação, sendo necessária apenas a inversão da ordem das operações, bem como da alimentação da chave da rodada.

2.7 Considerações finais do capítulo 2

Este capítulo abordou conceitos de criptografia e suas diversas classes, com foco na criptografia desenvolvida para dispositivos de baixo consumo, denominada Criptografia Leve.

Foram explanados ainda as cifras de bloco AES, PRESENT e CLEFIA, suas características e funcionamento, sendo que estes são detalhados no capítulo 4.

Capítulo 3

Análise de Trabalhos Correlatos

Diversas pesquisas sobre desempenho e segurança de implementações de algoritmos de criptografia leve em *hardware* e *software* vem sendo publicadas atualmente.

Nesta seção são apresentados análises e resultados de 10 trabalhos na área de criptografia leve, com destaque para implementações em *hardware*. Também são relatados alguns trabalhos que visam a análise do consumo energético de implementações de criptografia em hardware.

3.1 Criptografia leve em *hardware*

Para uma melhor compreensão dos resultados das implementações de algoritmos criptográficos leves em *hardware*, é importante o entendimento das diversas métricas de desempenho utilizadas nos trabalhos pesquisados.

Conforme já mencionado, uma métrica comum na maioria dos trabalhos é o número de Equivalente de Porta (GE – *Gate equivalent*), medida relacionada com a área de circuito integrado que a implementação da lógica do algoritmo requer, sem comprometer a exigência de fortes propriedades de segurança, ou seja, não vulnerável aos principais tipos de ataques. O número de GEs é obtido através da divisão da área de silício utilizada para uma cifra com uma determinada biblioteca de células padrão pela área de uma porta NAND de duas entradas (Batina *et al*. 2013). A quantidade de ciclos de relógio (ciclos de *clock*) que o algoritmo requer nos processos de encriptação ou decriptação também é considerada na maioria dos trabalhos. Em *hardware* configurável FPGA, outra métrica utilizada é relacionada com os *Slices* (fatias) do dispositivo, onde cada *slice* é composto por *look-up-tables* (tabelas-de-consulta - *LUTs*) e flip-flops, sendo que a quantidade de *slices* difere de acordo com a família/fabricante do chip. A velocidade de processamento para encriptação e decriptação (eficiência) também é uma medida importante para análise de desempenho do algoritmo.

Uma métrica diferente é utilizada por Manifavas *et al*. (2014), onde uma equação denominada de Figura de Mérito (FOM – *figure of merit*) é utilizada com o objetivo de comparar o desempenho entre diversas implementações e em diferentes dispositivos. A equação FOM é obtida através da diferença entre a taxa de transferência (Kbps), fixada em uma frequência de 100 KHz, pela área ao quadrado (GE^2) de cada implementação. Embora esses métodos para medir eficiência de uma arquitetura leve sejam úteis, os mesmos não respondem a todas as questões relativas a cifras leves, tendo em vista características específicas de cada fabricante/chip. Porém, outros aspectos de desempenho podem

ser considerados, tais como medidas de potência, energia.

3.2 Implementações em ASICs

Um importante trabalho na área de criptografia leve foi elaborado por Eisenbarth *et al*. (2007), onde foram realizadas comparações de diversas implementações de cifras leves em *hardware* e *software*, incluindo o algoritmo padrão do NIST (AES). A tabela 3.1 mostra os resultados das implementações das cifras leves em *hardware* ASIC, com destaque para o algoritmo PRESENT-80, com bom desempenho de área, equivalente a 46,17% da área total requerida pelo AES-128, bem como para o CLEFIA-128, que apesar da área requerida ser maior, apresentou uma taxa de transferência muito superior em comparação ao AES-128.

Tabela 3.1. Comparação de Cifras Leves

Algoritmo	Chave / Bloco (*bits*)	Ciclos por Bloco	Taxa de transferência a 100 KHz (Kbps)	Processo lógico	Área (GE)
Cifras de Bloco					
PRESENT	**80/64**	**32**	**200,00**	**0,18 µm**	**1570**
AES	**128/128**	**1032**	**12,40**	**0,35 µm**	**3400**
Hight	128/64	34	188,20	0,25 µm	3048
CLEFIA	**128/128**	**36**	**355,56**	**0,09 µm**	**4993**
mCrypton	96/64	13	492,30	0,13 µm	2681
DES	56/64	144	44,20	0,18 µm	2309
DESXL	184/64	144	44,20	0,18 µm	2168
Cifras de Fluxo					
Trivium	80	1	100	0,13 µm	2599
Grain	80	1	100	0,13 µm	1294

Fonte: Eisenbarth *et al*. (2007).

Batina *et al*. (2013) realizou uma análise de potência, energia e área de arquitetura de vários algoritmos leves de cifragem de blocos recentemente desenvolvidos. Foi utilizada a ferramenta *Cadence Encounter RTL Compiler* sintetizando em biblioteca de tecnologia Faraday UMC 0,13 μm e simulando através do *ModelSim*. As simulações foram executadas para uma frequência fixada em 100 KHz. A tabela 3.2 exibe alguns resultados destas implementações.

De acordo com Batina *et al*. (2013), nem sempre uma área de implementação menor indica um consumo de energia menor, esses valores dependem também da complexidade dos cálculos de encriptação/decriptação, da arquitetura utilizada, como também dos ciclos necessários para a execução dos processos referente a cada cifra.

Tabela 3.2. Dados de performance e energia de implementações de cifras leves de bloco

Arquitetura da Cifra	Bloco / Chave	Ciclos de encriptação	Área (GE)	Potência Estática (*μW*)	Potência Dinâmica (*μW*)	Energia por bit (*p*J/Bit)	Energia (*n*J)
CLEFIA	**128/128**	**18**	**6941**	**13,24**	**37,09**	**70,78**	**9,06**
Klein_paralelo	64/64	12	2760	4,88	2,18	13,24	0,85
Klein_serial	64/64	98	1432	2,56	1,48	61,86	3,96
LED	64/128	48	3194	5,62	2,34	59,70	3,82
mCryton	64/96	13	3197	5,80	2,50	16,86	1,08
PRESENT	**64/80**	**31**	**2195**	**3,75**	**1,14**	**23,96**	**3,82**
Prince_dobrado	64/128	12	2953	5,75	2,80	16,03	1,03
Prince_desdobrado	64/128	1	39938	16,13	120,20	21,30	1,36
Katan_32	32/80	254	801	1,52	0,43	154,78	4,94
Katan_48	48/80	254	925	1,71	0,49	116,42	5,60
Katan_64	64/80	254	1048	1,94	0,56	99,22	6,34

Fonte: Batina *et al*. (2013).

É importante ressaltar que os resultados de energia e potência dinâmica são informações relevantes que podem auxiliar na escolha de uma cifra leve para determinada aplicação.

Uma análise comparativa de diferentes implementações de cifras leves que requerem menos de 3.000 GEs é realizada por Manifavas *et al*. (2014). O mesmo utiliza uma métrica de comparação pouco comum nos trabalhos pesquisados, a Figura de Mérito (FOM), métrica mencionada anteriormente obtida através da diferença entre a taxa de transferência da cifra (Kbps) pela área ao quadrado (GE^2), com frequência fixada em 100 KHz. Nesta métrica, quanto maior for o valor de FOM, melhor é o desempenho.

Dentre alguns resultados importantes, destacam-se o PRESENT com tamanho de chave de 80 *bits*, que apresentou uma taxa de transferência de 200 Kbps e área de 1570 GEs resultando em uma FOM=811, a variante DM-PRESENT com chave de 64 *bits*, obteve 387,88 Kbps de taxa de transferência, área de 2530 GEs e FOM=605,95, um resultado inferior se comparado ao PRESENT tradicional. CLEFIA com chave de 128 *bits* apresentou uma taxa de transferência baixa de 39 Kbps, GE=2488 e FOM=63. O melhor resultado com base na métrica utilizada foi da cifra PRINTcipher, com chave de 80 bis, taxa de transferência de 100 Kbps, 503 GEs de área, resultando em uma FOM=3952.

Apesar de a métrica FOM destacar comparações de diferentes arquiteturas de cifras leves em diversos dispositivos de *hardware*, a mesmo desconsidera dados importantes, como ciclos de *clock* requeridos nos processos de encriptação e decriptação, bem como dados de consumo de energia, podendo provocar equívocos na escolha de determinadas cifras com base nessa métrica. Também é importante ressaltar que a quantidade de informação a ser criptoprocessada, bem como o nível de segurança requerido são requesitos importantes na escolha de determinado algoritmo.

Bansod *et al*. (2015) descreve os resultados de otimizações em cifras leves já existentes, desenvolvendo cifras leves compactas e com segurança adequada para aplicações como Internet das Coisas (IoT). As alterações nas cifras leves foram realizadas através do

incremento de instruções de GRP (operações de permutação de bit em grupo). Os algoritmos foram implementados em microcontrolador (chip LPC2129) de 32 *bits* da NXP (Philips) e verificados através do ambiente de simulação KEIL 4.0 ARM, também foram realizadas implementações em *hardware* utilizando a biblioteca de células padrão UMCL18G212T3, processo lógico 0,18 μm, Virtual Silicon.

A figura 3.1 exibe a comparação de área (GE) de algumas cifras com PRESENT-GRP. A nova variante GRP leva vantagem na comparação com outras cifras, porém quando comparado ao PRESENT tradicional, há uma desvantagem em termos de GEs, que segundo Bansod *et al*. (2015), essa diferença se deve a arquitetura mais complexa e segura de PRESENT-GRP.

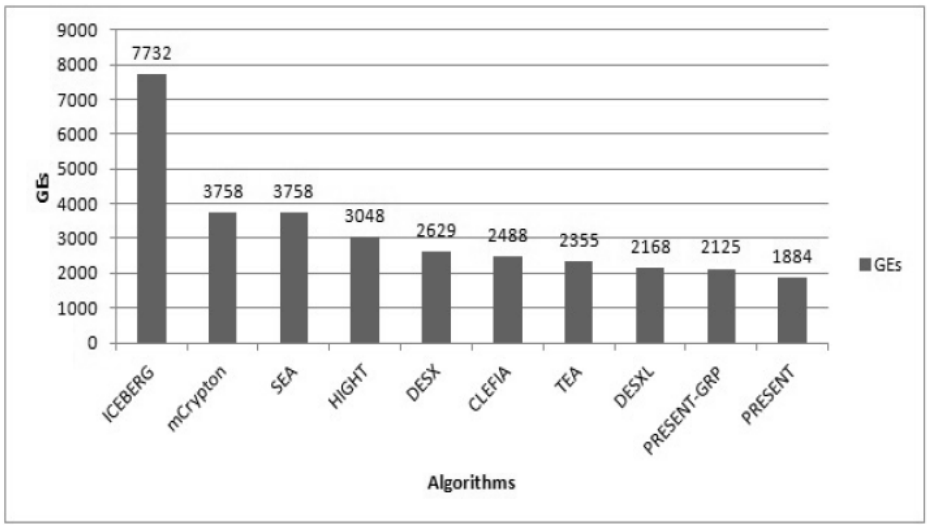


Figura 3.1. Comparação da cifra PRESENT-GRP com outras cifras
Fonte: Bansod *et al*. (2015).

Nas implementações em *software* (microcontrolador LPC2129), destaca-se o resultado satisfatório de PRESENT-GRP em comparação com o PRESENT padrão, requerendo uma quantidade menor de memória *Flash*.

3.3 Implementações em FPGAs

Os algoritmos criptográficos leves HIGHT e PRESENT foram implementados em FPGA por Yalla e Kaps (2009), os mesmos utilizaram o dispositivo XC3S50-5, da família Spartan-3, fabricante Xilinx, utilizando o *software* ISE 9.2i e ambiente de simulação Active-HDL 7.2. O trabalho também compara os resultados de suas implementações com os algoritmos Camellia, TinyXTEA-3, AES e a carteira cifras eSTREAM simulados em outros trabalhos que utilizaram o mesmo dispositivo FPGA (XC3S50-5).

As simulações foram realizadas com HIGHT e PRESENT com blocos de 64 *bits* e com chaves de força plena, ou seja, chave com comprimento de 128 *bits*, apesar de a criptografia leve considerar comprimento de chave de 80 *bits* como sendo suficientemente seguro.

Dentre alguns resultados destacam-se, PRESENT com 117 slices e HIGHT com 91 slices de área, contra 222 slices do AES (CHODOWIEC e GAJ, 2003). Com relação à eficiência na encriptação dos dados, AES apresentou resultado mais favorável que PRESENT e HIGHT, 129 Mbps contra 28,46 Mbps e 65,48 Mbps respectivamente.

Duas versões do algoritmo CLEFIA implementadas em FPGA foram propostas por Proença e Chaves (2011), uma versão adaptada do trabalho de Sugawara *et al*. (2008), que teve como objetivo uma implementação de alta performance do CLEFIA-128 em ASIC, e uma versão com foco em otimização estrutural do algoritmo e melhoria de eficiência. Ambas as versões podem ser parametrizadas para os tamanhos de chaves (128, 192 e 256 *bits*), sendo implementadas em três plataformas FPGA diferentes, Spartan 3E (dispositivo XC3S1200-4), Virtex 4 (dispositivo XC4LX200-11) e Virtex 5 (dispositivo XC5LX30-3), todos do fabricante Xilinx.

Para ambas as versões, técnicas de mesclagem das operações de substituição S_0 e S_1 (*S-boxes*), e difução matricial (M_0 e M_1) foram aplicadas, obtendo uma estrutura de equações de transformação denominadas *T-boxes*, reduzindo assim o caminho crítico do algoritmo. Na versão compacta do CLEFIA, otimizações nas estruturas das funções-*F* (F_0 e F_1) foram realizadas, obtendo uma estrutura unificada de operação, reduzindo consideravelmente os recursos de *hardware* necessários para a implementação. É importante destacar que nesta proposta, o processo de Programação de Chave (geração das chaves de clareamento - *WK*), adicionadas durante as rodadas de encriptação e decriptação, foram realizadas via *software* e incrementadas ao *hardware* no início do processo.

A tabela 3.3 mostra os resultados das implementações de CLEFIA proposta por Proença e Chaves (2011). Nota-se uma melhoria significativa na performance das implementações, comparando as diferentes plataformas utilizadas. Segundo o autor isso se deve a diferente tecnologia empregada na construção dos dispositivos, bem como uma frequência de operação maior. Na tabela 3.3, a implementação tipo 1 refere-se ao algoritmo CLEFIA adaptado de Sugawara *et al*. (2008), a tipo 2 refere-se ao algoritmo CLEFIA versão compacta.

Fan *et al*. (2010) realizou implementações do algoritmo de criptografia ultra-leve Hummingbird em FPGA, família Spartan-3 (dispositivo XC3S200-5) utilizando o *software* ISE *Design Suite* V11.1 da Xilinx, bem como realizou comparações com outras cifras leves existentes.

Observou- se neste experimento que Hummingbird com tamanho de chave de 256 *bits* e bloco de 16 *bits* ocupa uma área de 273 slices, e taxa de encriptação de 160,4 Mbps em uma frequência do dispositivo de 40,1 MHz, sendo a eficiência de 0.59 (Mbps/slice). Na comparação com XTEA, ICEBERG, SEA e AES, os resultados mostraram Hummingbird com um melhor desempenho, alcançando um maior rendimento com menor requisito de área. Porém, na comparação com PRESENT (POSCHMANN, 2009) o resultado foi inferior. PRESENT com chave de 80 *bits*, implementado em dispositivo da mesma família, requer 176 slices, possui taxa de encriptação de 516 Mbps e eficiência de 2,93 (Mbps/slice).

Implementações em *hardware* dos algoritmos leves padrão ISO/IEC 29192-2, PRESENT e CLEFIA, bem como do algoritmo AES, foram realizadas por Hanley e O'Neill (2012), tendo como objetivo fornecer uma comparação equitativa dos mesmos.

Para tal análise comparativa, os algoritmos foram implementados em duas plataformas FPGAs distintas, a Virtex-II (dispositivo XC2VP7 FG456-5), que usa LUTs de 4 entradas, e a plataforma Virtex-5 (dispositivo XC5VLX50 FF324-1), com LUTs de 6 entradas, ambos os dispositivos do fabricante Xilinx, e utilizando o *software* ISE *Webpack* 10.1.031 para simulação e geração dos resultados.

Tabela 3.3. Desempenho do CLEFIA em diferentes FPGAs

Plataforma	Tipo	Tamanho de chave	Ciclos de Clock	Mbps	Mbps/ slice
Spartan 3E	1	128	18	768	1,2
		192	22	628	1,0
		256	26	531	0,9
	2	128	36	658	2,4
		192	44	538	2,0
		256	52	455	1,7
Virtex 4	1	128	18	1273	2,0
		192	22	1052	1,7
		256	26	881	1,4
	2	128	36	1045	5,1
		192	44	855	4,2
		256	52	724	3,5
Virtex 5	1	128	18	1707	10,0
		192	22	1396	8,2
		256	26	1181	6,9
	2	128	36	1301	15,1
		192	44	1064	12,4
		256	52	900	10,5

Fonte: Proença e Chaves (2011).

Neste trabalho, duas abordagens para cada algoritmo são analisadas, sendo uma arquitetura iterativa (I), otimizada para velocidade de processamento, e outra serial (S), com foco na otimização de área. Para os algoritmos CLEFIA e AES, foram analisadas outras duas versões de cada arquitetura, baseadas em outros trabalhos que utilizaram diferentes técnicas de otimização de S-box. Também para o CLEFIA, versões iterativas e seriais com a programação de chave realizadas em *software* foram analisadas. Para ambos os algoritmos, o tamanho da chave adotada foi de 128 *bits*, e bloco de 128 *bits*, com excessão do PRESENT que opera com bloco de 64 *bits*, e os resultados foram coletados com base no processo de encriptação.

Diferente de outros trabalhos, Hanley e O'Neill (2012) mostram dados mais detalhados de suas implementações, como quantidade de Flip-Flops e LUTs que o algoritmo requer, e também taxa de transferência para uma frequência de 13,56 MHz, que é recomendada para smart cards (padrão ISO/IEC 14443-2). A tabela 3.4 mostra os resultados dos experimentos na plataforma Virtex-5.

Tay *et al*. (2015) propôs uma versão compacta do algoritmo PRESENT implementado em FPGA, utilizando portas lógicas AND e OR para construção da caixa de substituição (S-Box) através do mapeamento de Karnaugh. Expressões booleanas de soma e

produto (SOP - *sum-of-products*) foram obtidas para os *bits* de saída da S-box, e posteriormente foram utilizadas técnicas de fatoração das expressões booleanas em comum, com o objetivo de reduzir a quantidade de portas utilizadas na construção da S-Box e consequêntemente à área de implementação do algoritmo. Ao final da simplificação foram necessárias 26 portas AND e 17 portas OR para a construção na S-Box booleana.

Tabela 3.4. Resultados das implementações em FPGA Virtex-5 (Hanley e O'Neill, 2012)

Cifra	Tipo	S-Box	Flip-Flops	LUTs (6-I)	Slices	Max Freq (MHz)	Ciclos de latência	Tx. (Mbps)	Tx/slice (kbps/slice)	Tx[1] (Mbps)	Tx/slice[1] (kbps/slice)
AES	I	(Canright, 2005)	271	1391	456	96,04	26	472,81	1036,87	66,76	146,40
		LUT	271	1266	359	136,84	26	673,67	1876,53	66,76	185,95
	S	(Canright, 2005)	286	274	137	77,29	160	61,83	451,33	10,85	79,18
		LUT	286	258	113	113,25	160	90,60	801,77	10,85	96,00
CLEFIA	I	(Shirai *et al*., 2007)	409	816	243	108,66	46	302,36	1244,27	37,73	155,28
		LUT	409	809	267	267,00	46	742,96	2782,61	37,73	141,32
	S	(Shirai *et al*., 2007)	329	469	156	110,69	272	52,09	333,91	6,38	40,90
		LUT	329	467	155	93,96	272	44,22	285,27	6,38	41,17
CLEFIA[2]	I	(Shirai *et al*., 2007)	409	816	243	108,66	34	409,07	1683,43	51,05	210,08
		LUT	409	809	267	267,00	34	1005,18	3764,71	51,05	191,20
	S	(Shirai *et al*., 2007)	329	469	156	110,69	160	88,55	567,64	10,85	69,54
		LUT	329	467	155	93,96	160	75,17	484,95	10,85	69,99
PRESENT	I	LUT	200	285	87	250,89	47	341,64	3926,87	18,46	212,24
	S	LUT	203	237	70	245,76	295	53,32	761,68	2,94	42,03

[1]Usando frequência de 13.56 MHz (padrão ISO/IEC 14443-2 para smart cards)

[2]CLEFIA com programação de chave via *software*

A nova cifra PRESENT proposta por Tay *et al*. (2015) foi implementada em plataforma FPGA Virtex-5 (dispositivo XC5VLX50-1 FF324) e apresentou bons resultados, com relação a área de implementação, quando comparados a outras implementações do estado da arte, sendo necessário apenas 62 slices o que corresponde a 71.26% em comparação com a melhor implementação de PRESENT proposta por Hanley e O'Neill (2012), porém em relação a eficiência, apresentou 827,81 kpbs/slice o que corresponde a aproximadamente 4,7 vezes a eficiência da implementação de Hanley e O'Neill (2012).

Uma estrutura de cifra dupla dos algoritmos CLEFIA e AES em *hardware* FPGA é desenvolvida por Resende e Chaves (2015), através da unificação das estruturas comuns de operação de ambos os algoritmos, utilizando uma abordagem de otimização T-Box, bem como do uso adequado dos componentes existentes em tecnologias FPGA atuais, como bloco de memórias (BRAMs). A figura 3.2 ilustra o caminho de dados da estrutura de cifra AES/CLEFIA.

Foram realizadas implementações da cifra dupla AES/CLEFIA em FPGAs Virtex-5

(dispositivo XC5VLX30-3) e Virtex-6 (XC6VLX240T-3) da fabricante Xilinx, utilizando a ferramenta ISE *Design Suite* (v14.5). Tendo em vista que ainda não foram publicadas outras propostas de mais de uma cifra leve em um único *Core* (núcleo) FPGA, comparações de desempenho foram realizadas com implementações individuais do estado da arte para CLEFIA e AES. A tabela 3.5 destaca os resultados dessa abordagem para o processo de encriptação ou decriptação.

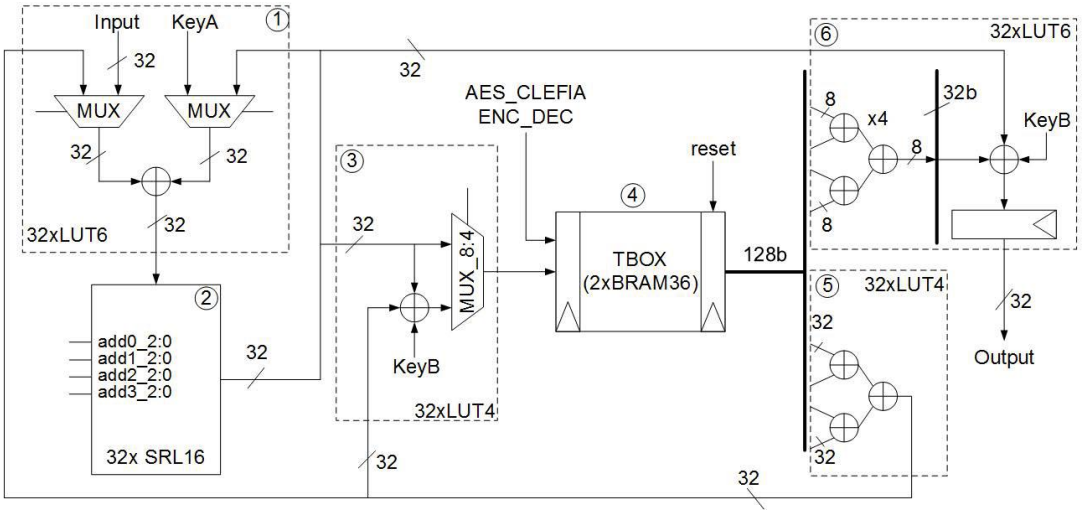


Figura 3.2. Caminho de dados da estrutura de cifra dupla CLEFIA/AES (Resende e Chaves, 2015)

Tabela 3.5. Resultados das implementações da cifra dupla AES/CLEFIA em FPGAs

Dispositivo	Algoritmo	Slices	BRAM	Freq. (MHz)	Taxa de Transferência (Gbps)	Eficiência (Mbps/slice)
Virtex-5 (XC5VLX30-3)	CLEFIA	123	3	352	1,707	8,72
	AES				0,850	6,91
Virtex-6 (XC6VLX240T-3)	CLEFIA	115	3	332	1,012	8,80
	AES				0,802	6,97

Fonte: Resende e Chaves (2015)

Observa-se que a área requerida para a cifra dupla é bem competitiva, requerendo apenas 123 slices e mais 3 BRAMs, sendo que no trabalho de Hanley e O'Neill (2012) os melhores resultados dos algoritmos AES e CLEFIA implementados em um dispositivo da mesma família Virtex-5 FPGA, necessita de 113 e 155 slices respectivamente. Porém é importante mencionar que as implementações de Hanley e O'Neill (2012), não utilizam BRAMs em sua arquitetura. Ressalta-se ainda, que na proposta de Resende e Chaves (2015) a Programação de Chave foi executada via *software* e as subchaves adicionadas ao *hardware* no início do processo.

3.4 Consumo de energia em dispositivos criptográficos

A modelagem de variáveis relacionadas ao consumo de energia em dispositivos de criptografia, especialmente aqueles com restrições em relação ao poder computacional, memória e energia, também tem sido alvo de pesquisas recentes, não só com o objetivo de comparação entre implementações, mas também visando a obtenção de informações importantes sobre determinado algoritmo, que podem inclusive levar a descoberta ("quebra"), da chave de criptografia, e consequentemente ao possível acesso a informações sigilosas.

No trabalho realizado por Singh *et al*. (2015) por exemplo, foram analisados ataques por um método denominado Análise Diferencial de Potência (DPA - *Differential Power Analysis*), em uma versão otimizada do algoritmo AES. Os resultados mostraram que a versão de baixo custo e consumo de AES é mais suscetível ao ataque DPA. O trabalho propôs ainda um *design* com reguladores de tensão tipo *Low-Drop-Out*, que são reguladores com baixa queda de tensão, para aumentar a resistência ao ataque DPA ao algoritmo AES compacto.

De acordo com Tang *et al*. (2012), um ataque por *Differential Power Analysis* (DPA) coleta informações sobre o consumo de energia de um sistema físico, que pode ser um *hardware* criptográfico, realizando a modelagem estatística destes dados com objetivo da obtenção de informações importantes para a quebra do sistema criptográfico.

Uma investigação sobre análises de consumo energia em ataques contra sistemas criptográficos é relatado por Kocher *et al*. (2011). De acordo com o autor o consumo de energia de um circuito integrado reflete a atividade agregada de seus elementos individuais, como por exemplo, a comutação dos transistores pode ser diferente conforme os tipos de dados, o que pode refletir em um consumo diferenciado. Neste contexto, a aplicação de técnicas estatísticas que buscam a correlação entre os dados de consumo de energia durante o processo de encriptação e outros dados já conhecidos, visando a obtenção da chave secreta é conhecido como ataque por *Differencial Power Analysis*.

Ataques por DPA em algoritmos de criptografia leve foram realizados por Yalla P. S. (2009). Diferentes arquiteturas otimizadas dos algoritmos AES, Camellia, xTEA, HIGHT and PRESENT foram submetidas aos ataques. As implementações foram feitas em *Hardware* FPGA Spartan-3 de baixo custo (Xilinx). Os resultados mostraram que arquiteturas que usam registros para armazenamento de dados e chaves são mais suscetíveis aos ataques. Os algoritmos que usavam uma quantidade maior de operações XOR também demonstraram maior vulnerabilidade aos ataques.

Batina *et al*. (2013) realizou um trabalho sobre análise comparativa de potência, energia e área de arquitetura de vários algoritmos leves de cifragem de blocos implementados em *Hardware* ASIC, os resultados mostraram que nem sempre uma área de implementação menor indica um consumo de energia menor, esses valores dependem de alguns fatores, como complexidade dos cálculos de encriptação, da arquitetura implementada, bem como da quantidade de rodadas necessárias para o processo de encriptação.

As pesquisas demonstraram que maioria desses trabalhos analisa variáveis do consumo de energia descrevendo o padrão no tempo de execução. Sendo que essas variáveis de consumo representam uma importante métrica nas implementações de criptografia, podendo ser relevantes tanto para fins de comparação entre implementações, como por exemplo, visando a obtenção de informações sigilosas através de análises estatísticas sobre estas variáveis.

3.5 Considerações finais do capítulo 3

Por se tratar de uma área relativamente nova e ainda em expansão, muitos desafios relativos a aspectos da criptografia leve estão em discussão. Porém, observou-se que a maioria dos trabalhos pesquisados tem por objetivo principal a otimização dos recursos computacionais dos algoritmos criptográficos padrão ISO/IEC 29192-2 (PRESENT e CLEFIA), como também do AES, sem comprometer aspectos de eficiência e segurança.

A tabela 3.6 ilustra as implementações de criptografia leve realizadas nos trabalhos pesquisados. Além de resumir os aspectos analisados, dispositivos e métricas utilizadas, as limitações de cada trabalho também são mencionadas.

De fato, é possível observar nos trabalhos pesquisados que a tendência é a otimização e implementação em *hardware* FPGA das cifras leves padrões AES, PRESENT e CLEFIA, avaliando e comparando através de diversas métricas.

Tabela 3.6. Resumo das implementações em *hardware* dos trabalhos pesquisados

Trabalho	**Algoritmos**	**Aspectos Analisados**	**Dispositivo / Ferramentas**	**Limitações**
EISENBARTH *et al*., 2007	PRESENT, CLEFIA, DES, DESXL, AES, Hight, mCrypton, Trivium, Grain	Área (GE); ciclos de clock; taxa de transferência (kbps)	ASIC (diferentes processos lógicos)	Não realizou implementação em FPGA; bem como medidas de energia
BATINA *et al*., 2013	AES, CLEFIA, PRESENT Klein, LED, mCrypton, , Prince, Katan	Área (GE), ciclos de clock; potência; energia	ASIC Simulação em *ModelSim;* RTL Compiler (UMC 0.13 µm)	Não realizou testes e medições reais de energia; analisar taxa de transferência (eficiência)
MANIFAVAS *et al*., 2014	Diversos Área < 3000 GEs	Eficiência através da métrica: FOM = taxa de transferência [Kbps] / área ao quadrado [GE²]	Não utilizou	Realizou apenas análise comparativa através da métrica FOM, desconsiderando os ciclos e consumo de energia
BANSOD; RAVAL; PISHAROTY, 2015	PRESENT-GRP, diversas cifras de bloco	Área (GE)	ASIC (biblioteca UMCL18G212T3 0.18µm) Virtual Silicon	Não incrementou GRP em outras cifras; não realizou análise de outras métricas, como: ciclos de clock, taxa de transferência e energia
HANLEY e O'NEILL, 2012	AES, PRESENT e CLEFIA (várias versões compactas)	Área (Slices, LUTs, Flip-Flops), ciclos de latência, taxa de transferência, eficiência (taxa de transferência / área)	FPGAs: Virtex-II (XC2VP7 FG456-5) Virtex-5 (SC5VLX50 FF324-1) ISE *Webpack* 10.1.031	Não analisou aspectos de consumo de energia.
YALLA; KAPS, 2009	PRESENT Hight	Área (slices), ciclos por bloco, atraso máximo (ns), taxa de transferência (Mbps)	FPGA – Spartan 3 (*XC3S50-5*) Xilinx ISE 9.2i Simulação Active-HDL 7.2	Não realizou comparação com a cifra leve CLEFIA e o AES compacto. Não detalhou consumo de área em LUTS e Flip-Flops
TAY *et al*., 2015	PRESENT (versão compacta S-box booleana)	Área (Slices, LUTs, Flip-Flops), ciclos de latência, taxa de transferência, eficiência (taxa de transferência / área)	FPGA – Virtex-5 (XC5VLX50-1 FF324)	Não realizou comparação com outras cifras compactas (CLEFIA e AES). Não analisou energia.
PROENÇA e CHAVES, 2011	CLEFIA (duas versões compactas)	Área (Slices, BRAMs), ciclos de latência, taxa de transferência, eficiência (Taxa de transferência / área)	FPGA: Spartan 3E (XC3S1200-4) Virtex 4 (XC4LX200-11) Virtex 5 (XC5LX30-3)	Não detalhou consumo de área em LUTS e Flip-Flops. Não analisou energia.
FAN *et al*., 2010	Hummingbird	Área (slices), taxa de transferência (Mbps) Eficiência (taxa de transferência / área)	FPGA: Spartan 3 (XC3S200-5) ISE *Design Suite* V11.1	Não analisou ciclos de clock (latência). Não comparou com a cifra leve CLEFIA.
Resende e Chaves, 2015	AES-CLEFIA (estrutura unificada)	Área (slices, BRAMs), taxa de transferência e eficiência (taxa de transferência / área)	FPGAs: Virtex-5 (XC5VLX30-3) e Virtex-6 (XC6VLX240T-3) ISE *Design Suite* V14.5	Não detalhou consumo de área em LUTs e Flip-Flops. Não realizou comparação com a cifra leve padrão PRESENT.
Nosso	**AES, PRESENT e CLEFIA**	**Área (FFs, LUTs, Slices), taxa de transferência (Mbps), eficiência (Mbps/Slice), eficiência energética (Ws/bit), Consumo de corrente.**	**FPGA: Artix-7 (xc7a35tcpg236) Vivado *Design Suites* V2016.3**	**Foram mensurados tanto o desempenho em termos de área, processamento e também energia.**

Capítulo 4

Metodologia para a simulação e coleta de dados

Nesta seção são apresentados métodos utilizados para a simulação e coleta dos dados de consumo das arquiteturas implementadas, tais como: dispositivo FPGA utilizado, protótipo para medição do consumo de corrente, arquitetura de simulação, bem como ferramentas utilizadas para coletar e analisar os dados de desempenho das implementações, sendo detalhadas a seguir.

4.1 Circuitos programáveis FPGA

O grande avanço da microeletrônica nas últimas décadas possibilitou o desenvolvimento de circuitos integrados reconfiguráveis, como o Arranjo de Portas Programável em Campo – FPGA (do inglês *Field Programmable Gate Array*), tecnologia que permite a implementação de circuitos e arquiteturas cada vez mais complexas, sem a necessidade do uso de grandes recursos de fundição de silício (MORENO *et al*., 2003). FPGAs são compostos basicamente de um conjunto de células lógicas ou blocos lógicos organizados em forma de uma matriz. O roteamento ou interconexão entre os blocos lógicos, como também as funcionalidades dos mesmos são configuráveis via *software*, através de uma linguagem de descrição de *hardware* (HDL - *Hardware Description Language*), sendo que esta característica traz inúmeros benefícios para o desenvolvimento de projetos de circuitos, como a rápida prototipagem e redução de custos.

Outra grande vantagem na utilização destes circuitos, em comparação com processadores de propósito gerais, é a opção dos processos ocorrem de forma não sequencial, possibilitando a realização de centenas de operações por ciclo.

Existem diferentes arquiteturas de FPGA, a depender do fabricante ou do dispositivo, porém alguns elementos fundamentais em comum podem ser destacados (MORENO *et al*., 2003):

- **CLB** (*Configurable Logic Block*): bloco lógico configurável, unidade lógica de um FPGA.

- **IOB** (*In/Out Block*): bloco de entrada e saída, localizado na periferia dos FPGAs, são responsáveis pela interface com o ambiente.

- **SB** (*Switch Box*): caixa de conexão, responsável pela interconexão entre os CLBs, através dos canais de roteamento.

4.1.1 Placa FPGA Basys 3™

A placa FPGA utilizada no trabalho foi a Basys 3™ da Digilent® (figura 4.1), que é uma plataforma de desenvolvimento de circuito digital baseado no chip Artix-7®, da fabricante Xilinx®. É uma placa preparada para a implementação de inúmeros projetos, que variam desde circuitos combinacionais básicos, até circuitos dedicados de alta complexidade. Já inclui diversos dispositivos adicionais, como LEDs, pinos de I/O, interruptores, conectores, display, dentre outros, com o objetivo de implantar/simular vários projetos sem a necessidade de dispositivos adicionais.

Algumas características específicas da Basys 3™ são citadas abaixo (DIGILENT, 2014):

- 33.280 células lógicas em 5.200 *slices* (onde cada *slice* contém 4 LUTs de 6 entradas e 8 flip-flops);
- 1.800 K*bits* de bloco RAM rápido (BRAM);
- 90 DSP slice;
- Clock interno de velocidade superior a 450MHz;
- Conversor analógico-digital on-chip (XADC);

Figura 4.1. Placa de desenvolvimento FPGA Basys 3

Fonte: Blog da Digilent Inc[1]

4.2 Linguagem de Descrição de Hardware

As linguagens de descrição de *hardware*, do inglês *Hardware Description Language* (HDL), são linguagens de programação utilizadas para configuração de disposivitos de *Hardware*, como FPGAs e ASICs por exemplo. Existem diversas HDLs utilizadas no mercado, como: VERILOG, VHDL, AHDL, Handel-C, SDL, ISP, ABEL dentre outras. Porém as de maior destaque são VHDL e VERILOG.

[1] Disponível em:< https://blog.digilentinc.com/index.php/small-solar-room-temperature-regulator/>. Acessado em 02/02/2016.

Para a consecução deste trabalho foi adotada a linguagem VHDL, pelo fato de ser uma linguagem consolidada e amplamente utilizada na descrição da maioria dos circuitos digitais reconfiguráveis.

4.2.1 VHDL

VHDL, do inglês *Very High Speed Integrated Circuit Hardware Description Language*, é uma linguagem padronizada pelo *Institute of Electrical and Electronics Engineers* (IEEE) para descrever componentes digitais. Foi criada pelo Departamento de Defesa Norte Americano (DoD) na década de 80 e atualmente é uma das HDLs mais utilizadas.

A descrição de um sistema utilizando VHDL pode ser feita de forma comportamental, onde é conhecido o comportamento e funcionamento do circuito que se deseja implantar ou de forma estrutural, onde é considerada a estrutura dos componentes empregados no circuito, como por exemplo, as portas lógicas (AND, OR, NOT) (MORENO *et al*., 2005).

Existem diversas características específicas da linguagem VHDL, porém não é o foco deste trabalho a descrição das mesmas.

4.3 Dados de área, taxa de transferência e eficiência no *hardware*

Para a realização da coleta de dados referente a área do projeto ocupada no FPGA e frequência máxima de clock das implementações deste trabalho, foi utilizada a ferramenta de *software* Vivado™ *Design Suite*, que é compatível com a Placa Basys 3™. Esta é uma ferramenta de alta performance da empresa Xilinx, que inclui muitos mecanismos e fluxos de *design* que facilitam e melhoram o desenvolvimento de projetos. As etapas de descrição em VHDL, simulação e validação, síntese, implementação e parte dos dados de consumo no FPGA foram realizadas através deste *software*.

Para uma melhor compreensão das métricas abordadas neste trabalho, é importante salientar as características do *slice* (fatia), sendo este um dado muito utilizado atualmente para análise e comparação de consumo de área de um dispositivo ou circuito em FPGA. Para a placa FPGA utilizada neste trabalho (Basys 3), cada *slice* contém 4 LUTs de 6 entradas (que também pode ser configurada como LUTs de 5 entradas), e 8 FFs, dentre outros circuitos multiplexadores e lógica combinatória. Dois slices, juntamente com outros circuitos, formam um Bloco Lógico Configurável (CLB – *Configurable Logic Block*).

A Figura 4.2 ilustra uma *slice* do FPGA (XC7A35TCPG236-1).

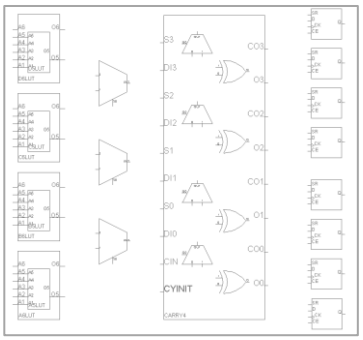


Figura 4.2. Ilustração de uma *Slice* do FPGA Placa Basys 3 (Chip: XC7A35TCPG236-1)

Os resultados de área ocupada no FPGA foram coletados e analisados através de duas perspectivas, disponíveis através do software Vivado™ *Design* Suite, sendo:

- **Synthesis** (Síntese): processo que converte o código VHDL, com base no *design* RTL (nível de transferência de registro), em uma representação a nível de portas lógicas, e gera estimativas do consumo de recursos de LUTs e Flip-Flops (FFs) para o FPGA escolhido;
- **Implementation** (Implementação): processo no qual a representação a nível de portas lógicas é devidamente interpretada, sendo realizada a alocação e roteamento dos circuitos lógicos no FPGA escolhido, de acordo com as restrições de pinos de entrada/saída, gerando dados reais do consumo de recursos (LUTs e Flip-Flops e Slices) e velocidade de clock para determinado projeto;

A síntese do projeto permite, além da análise da lógica de todos os componentes que formam a arquitetura do *hardware* criptográfico, uma análise estatística de área de cada módulo que compõe o sistema de uma forma geral, sem a necessidade de informar os pinos de entradas e saídas do *hardware*.

O processo de implementação do projeto no FPGA, permite verificar os dados reais de consumo do projeto, com os circuitos lógicos devidamente mapeados e com as rotas definidas. Somente nesta fase do projeto é possível coletar os dados de Slices (fatias) utilizados para implementação da lógica no projeto, pois este valor varia de acordo com forma com que os circuitos lógicos foram alocados e mapeados pela ferramenta de *software*, podendo inclusive, haver divergências entre os valores dos recursos de FFs e LUTs estimados na síntese, com os valores de recursos literalmente utilizados, uma vez que existem diversas possibilidades de alocação e roteamento destes circuitos lógicos no FPGA.

Neste trabalho, tanto no processo de síntese, quanto no processo de implementação, não foi utilizada nenhuma estratégia específica disponível pela ferramenta Vivado, visando otimização de área ou velocidade do projeto, sendo utilizada a opção padrão disponível no *software* (*Vivado Synthesis/Implementation Defaults*), que busca um equilíbrio entre área e caminho crítico do projeto.

Devido a restrições de pinos de entrada/saída (I/O) disponíveis pela placa FPGA utilizada neste trabalho (106 I/O disponíveis) e visando a devida coleta de dados do processo de implementação do projeto, foi agregado ao desenho de cada algoritmo de encriptação

alguns registradores de entrada e saída para atender a estas restrições, conforme modelo ilustrado na figura 4.3 onde os registradores de entrada (Texto e Chave), armazem 8 *bits* cada a cada ciclo de clock, até que a quantidade de bits dos blocos de texto e chave dos algoritmos sejam atendidas, no caso deste trabalho, os algoritmos AES e CLEFIA utilizam 128 bits para texto e 128 bits para chave, e o algoritmo PRESENT utiliza 64 bits para texto e 80 bits para chave. Já o registrador de saída recebe a quantidade de bits de acordo com cada algoritmo, e posteriormente externaliza estes dados, com vazão de 8 bits por ciclo, até que todos os bits encriptados sejam externalizados.

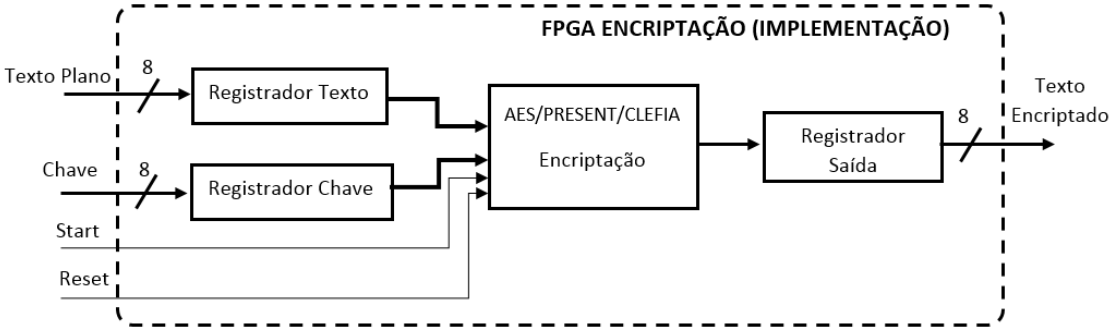


Figura 4.3. Arquitetura de implementação dos algoritmos de encriptação no FPGA

É importante salientar que a arquitetutura para implementação dos algoritmos de criptografia no dispositivo FPGA, foi utilizada apenas para adequação às restrições de portas de I/O do dispositivo, permitindo assim a alocação e roteamento do projeto, porém os recursos de área utilizados no projeto, foram extraídos com base somente nos algoritmos de encriptação, excluindo os registradores de entradas e saídas (ilustrados na figura 41). Isto foi possível através de relatórios detalhados que a ferramenta Vivado disponibiliza.

Desta forma, as métricas de área (LUTs, FFs e Slices) e clock máximo de operação (MHz) do *hardware* de criptografia foram extraídos através da ferramenta de *software* Vivado.

A métrica de taxa máxima de transferência de dados (Mbps – Mega bits por segundo) foi calculada de acordo com as especificações da quantidade de ciclos de clock para encriptação de um bloco de texto (latência) e o tamanho do bloco cifrado (em bits). A equação 4.1 detalha este cálculo.

$$\text{Taxa de Transferência (Mbps)} = \frac{Tamanho\ do\ Bloco}{Latência} \text{ x Clock Máximo (MHz)} \quad \text{(Equação 4.1)}$$

Já a métrica de eficiência do projeto é resultado da divisão da taxa máxima de transferência de dados (Mbps) pela quantidade de slices ocupados pela implementação do algoritmo de encriptação. A Equação 4.2 mostra o como esse cálculo é realizado.

$$\text{Eficiência (Mbps/Slices)} = \frac{Taxa\ máxima\ de\ transferência\ de\ dados\ (Mbps)}{Quantidade\ de\ Slices} \quad \text{(Equação 4.2)}$$

Outra métrica de eficiência utilizada neste trabalho está relacionada ao consumo de energia por bit, durante o processo de encriptação de dados. Esta métrica foi obtida a

partir da medição do consumo de corrente durante o processo de encriptação, que por motivo de restrições do protótipo de medição de corrente (descrito na subseção posterior), foi realizada a uma frequência de 10 kHz, que equivale 100 μs por ciclo. Com o valor médio de corrente consumida durante a encriptação, foi calculada a potência média (sendo adotada uma tensão de referência de 5 Volts (V), alimentação do FPGA). Também foi calculado o tempo de encriptação de um bloco de texto, que depende diretamente da quantidade de ciclos necessários para a encriptação (latência) e do tempo por ciclo de clock (neste trabalho 100 μs), desta forma o tempo total de encriptação de um bloco de texto foi calculado multiplicando-se a latência (quantidade de ciclos) do algoritmo pelo tempo de um ciclo de clock (em segundos). Com estas variáveis foi possível a realização do cálculo de eficiência energética, descrita na equação 4.3

$$\text{Eficiência Energética (Ws/bit)} = \frac{Potência\ (Watts)\ x\ Tempo\ (Segundos)}{Quantidade\ de\ bits\ em\ um\ bloco} \qquad \text{(Equação 4.3)}$$

Estas foram parte das métricas utilizadas neste trabalho, que foram devidamente comparadas com resultados de outros trabalhos, sendo apresentadas e discutidas no capítulo 6.

4.4 Protótipo de medição de corrente

Visando o atendimento aos pressupostos deste trabalho, um protótipo de medição de corrente elétrica e tensão com interface serial foi desenvolvido, para a realização de medições físicas do consumo de corrente do FPGA, configurado como *hardware* de criptografia dos algoritmos AES, PRESENT e CLEFIA.

Foram utilizados no desenvolvimento do protótipo, uma placa de monitoramento de corrente e tensão, distribuída pela empresa *Adafruit* e que utiliza o sensor INA219, desenvolvido pela empresa *Texas Instruments*, e também uma plataforma open-source *Arduino Uno*, que utiliza o microcontrolador ATmega328. Foi utilizado ainda um microcomputador para a recepção, armazenamento e tratamento destas informações.

O sensor INA219 possui interface de comunicação serial I2C, compatível com o *Arduino Uno*, além de um conversor analógico-digital interno de 12 bits (ADC) e também um amplificador de ganho programável (PGA). Com o devido ajuste do ganho do amplificador, a corrente máxima DC mensurada pelo sensor pode chegar a 400 mA, com resolução de 0.1mA e erro de aproximadamente 0.5% de acordo com as especificações disponibilizadas por *Texas Instruments* (2015).

A plataforma *Arduino Uno* foi utilizada para a realização da comunicação entre a placa *Adafruit* INA219 (sensor) e a porta serial conectada ao microcomputador, sendo utilizadas as bibliotecas com funções pré-definidas para esta comunicação e acesso aos registros das variáveis mensuradas (corrente e tensão), além do ajuste do ganho do amplificador (PGA) para alta precisão nas medições DC (resolução de 0.1 mA e erro aproximado de 0.5%).

A Figura 4.4 mostra o protótipo devidamente conectado com a alimentação de energia do FPGA (Basys 3) e pronto para as medições. Existe ainda a conexão de cinco portas

(Pmod JC) da placa FPGA com o Arduino, esta conexão visa a leitura de sinais que representam as rodadas de encriptação durante as simulações, conforme será explicado posteriormente.

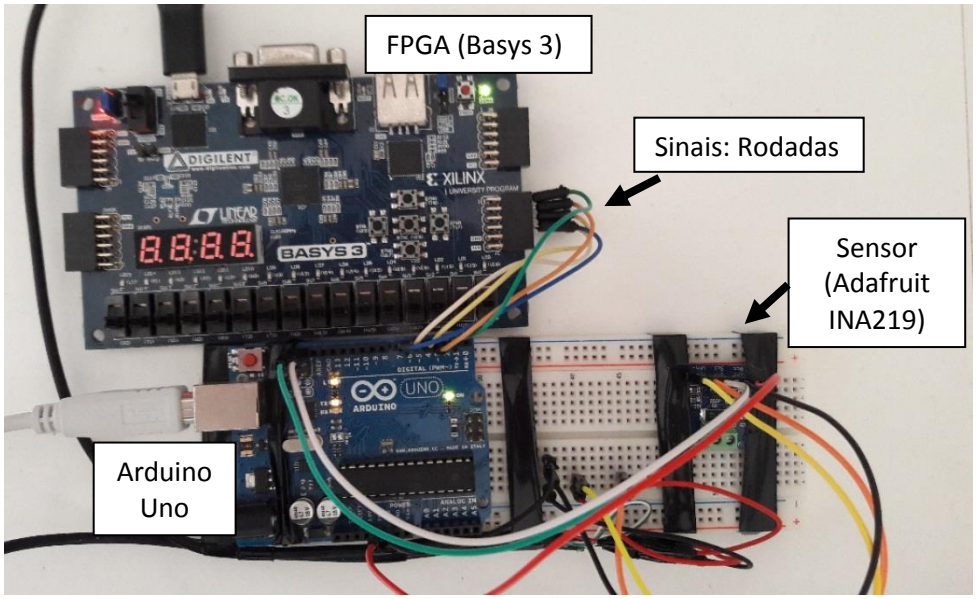


Figura 4.4. Protótipo de medição de corrente e tensão do FPGA

O recebimento destes dados de medição é realizado via serial do *Arduino*, conectado a um microcomputador que realiza a programação e exibe os dados através de um *software* (IDE do *Arduino*).

A Figura 4.5 mostra o sistema de medição completo durante uma simulação.

Portanto, neste trabalho, foi adotado uma medição de consumo de energia (corrente) através de uma implementação real de encriptação, e não através de ferramentas de software que estimam este consumo.

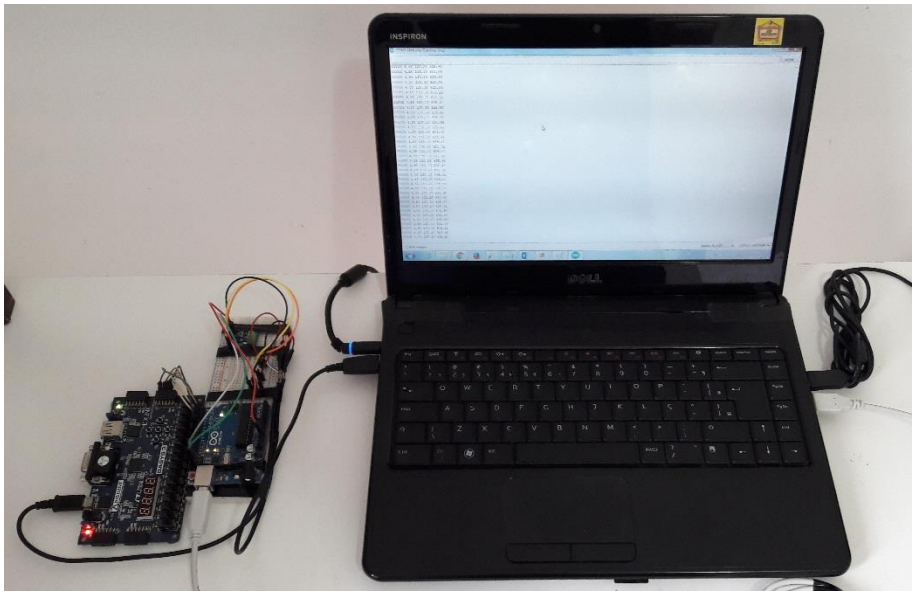


Figura 4.5. Sistema de medição recebendo dados durante simulação de encriptação

4.5 Arquitetura de simulação

Considerando as restrições de portas de entrada e saída do FPGA (I/O ports), bem como a necessidade de entrada de dados e controle das rodadas de encriptação, outros módulos foram incorporados a arquitetura do algorito de encriptação no FPGA, visando a execução da simulação com o objetivo de coletar os dados de consumo de energia. Os módulos complementares são descritos a seguir:

- **Gerador de sequência binária pseudo-aleatória (PRBS)**: módulo que gera sequências de bytes pseudo-aleatórios, com um período de 32 bits (2^{32}), que são utilizados como entrada para o *hardware* de criptografia (Texto plano e Chave), durante a simulação;
- **Registradores** (Reg_TextoPlano, Reg_Chave, Reg_TextoEnc): são modúlos registradores de dados, utilizados para carregar/descarregar os dados do Texto plano, Chave e Texto Encriptado. Dados estes que variam de acordo com o algoritmo de criptografia selecionado;
- **Controle**: módulo que realiza o controle da simulação;
- **Contador**: módulo contador para auxiliar o processo de controle da simulação;
- **Clock_gen**: módulo que gera, a partir do sinal de clock padrão do FPGA (100 MHz), um sinal de temporização (clock) adequado para a simulação;
- **Demux**: demultiplexador para auxiliar no carregamento dos registradores de Texto e Chave com os dados do PRBS;

A figura 4.6 ilustra a arquitetura utilizada para a realização da simulação de encriptação na placa FPGA.

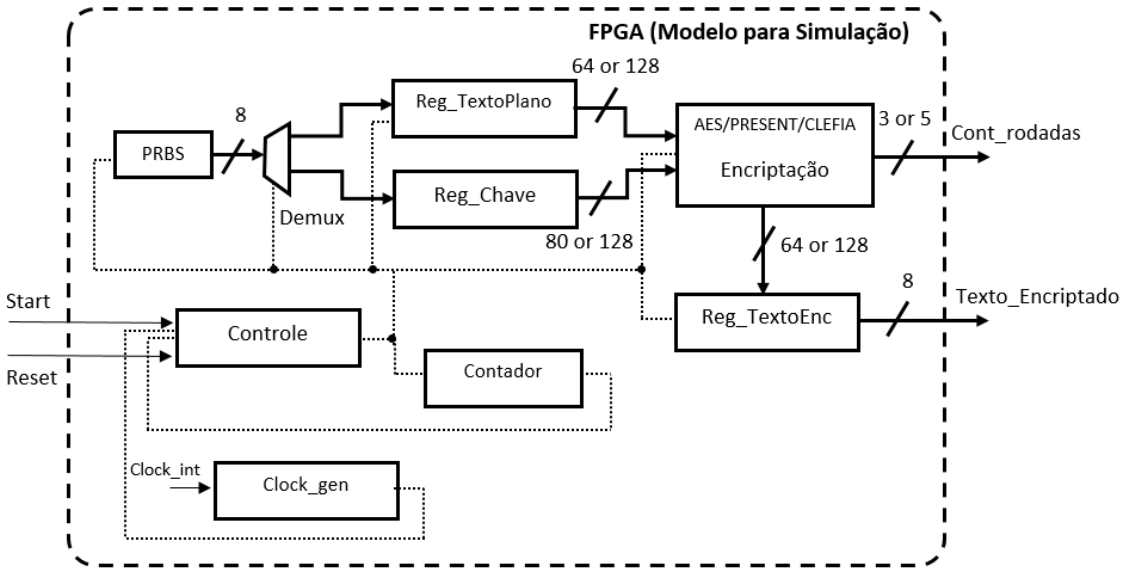


Figura 4.6. Modelo de simulação de encriptação de dados no FPGA

Visando obter o comportamento do consumo de corrente a cada rodada de encriptação, na arquitetura de simulação, os sinais que representam o contador de rodadas do

algoritmo de encriptação foram externalizados, para fins de leitura durante as medições.

Ainda para a realização da simulação, o módulo Clock_gen foi configurado para gerar sinais de clock de 10 KHz, isto para atender as características de tempo de leitura e transmissão de dados entre o protótipo de medição e o microcomputador, que coleta e trata estas informações.

4.5 Representação gráfica do consumo de corrente

Visando uma melhor análise e interpretação gráfica dos dados da curva de consumo de corrente e comparação entre as implementações, neste trabalho foi utilizado um método de análise no domínio da frequência denominado de Periodograma de Welch.

Este método baseia-se na estimativa da densidade espectral de potência (PSD) de um sinal, que é realizado pela divisão do tempo de sinal em blocos sucessivos, formando um periodograma, dado pelo quadrado da magnitude do resultado da transformada discreta de Fourier das amostras do processo, conforme mostra a Equação 4.4, para um dado sinal $x[n]$ de tamanho N.

$$Pxx(f) = \frac{|X(f)|^2}{F_s N}, \text{ onde } X(f) = \sum_{n=0}^{N-1} x[n] e^{-j\,2\,\pi\,f\,n/f_s} \qquad \text{(Equação 4.4)}$$

Uma das características do método de Welch é a capacidade de suavizar o espectro de um sinal na medida em que permite diminuir a variância entre os estimadores, com isso é possível a obtenção de uma melhor representação dos sinais obtidos.

As figuras 4.7 e 4.8 ilustram a diferença na representação gráfica dos sinais de consumo de corrente do algoritmo PRESENT durante a encriptação de dados, onde a figura 4.7 exibe o consumo de corrente no domínio do tempo, e a figura 4.8 exibe a curva que representa o consumo de corrente (em decibéis – dB) e a frequência normalizada, após a aplicação do método de Welch.

É notável a diferença de ambas as formas de representação do sinal de corrente (figuras 4.7 e 4.8), onde a representação através do método de Welch (figura 4.8), exibe uma curva suave, que pode auxiliar na identificação de uma curva padrão de consumo, bem como na comparação de diversas implementações.

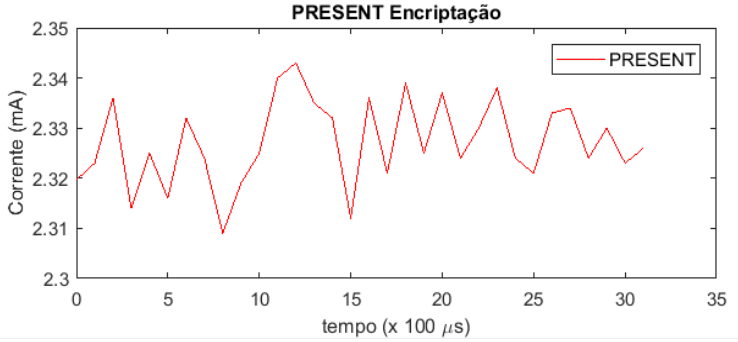


Figura 4.7. Consumo de corrente do algoritmo PRESENT amostrado no domínio do tempo

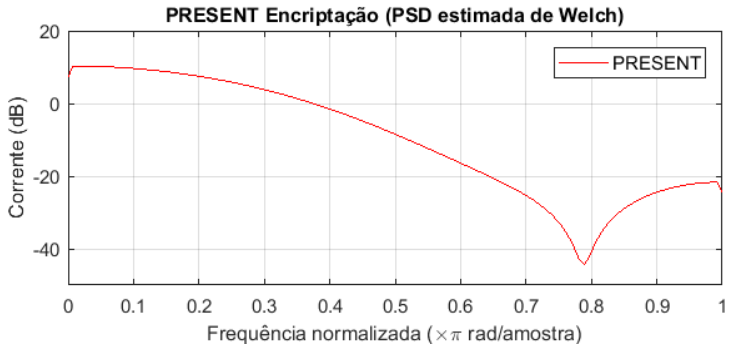


Figura 4.8. Representação do consumo de corrente do algoritmo PRESENT pelo método de Welch

Neste trabalho foram utilizadas funções especificas no *software* Matlab, que retornam a resposta em frequência do método de Welch (corrente em decibéis e frequência normalizada), para os dados de consumo de corrente.

4.6 Cenários de medição de corrente e coleta de dados

As medições de consumo de corrente foram realizadas de acordo com as condições abaixo:

• **Estático**: quando o FPGA está configurado com o algoritmo de simulação da encriptação, porém está no estado ocioso (não executando encriptação), sendo observado a corrente de fuga do circuito;

• **Dinâmico**: quando o FPGA está executando a simulação de encriptação;

Com o objetivo de se coletar dados de corrente apenas do processo de encriptação, foi realizada ainda uma medição da placa FPGA configurada sem o algoritmo de encriptação, pois devido a configuração padrão da placa FPGA Basys 3, LEDs e Display de 7 segmentos ficam ligados e, portanto, consumindo energia, não sendo interessante também mensurar estes dados de consumo. Também não é interessante mensurar os dados de consumo dos módulos PRBS, Registradores, e demais módulos agregados ao desenho de encriptação para a realização da simulação. Desta forma, o valor adotado para as medições nas condições Estático e Dinâmico, é a diferença entre a média aritmética de consumo do FPGA configurado com os algoritmos de encriptação (individualmente) e a média aritmética do consumo para o estado sem o desenho dos algoritmos de encriptação (AES, PRESENT, CLEFIA).

Os resultados coletados foram avaliados com base em uma quantidade significativa de amostras para cada condição de medição.

Para o cenário Dinâmico, foram coletadas 100 amostras do sinal de consumo de corrente, para cada rodada de encriptação, configurado com cada algoritmo de criptografia.

A tabela 4.1 exibe os dados de amostras selecionadas para o cenário dinâmico, bem como a quantidade de dados criptografados durante a simulação.

Já para o cenário Estático, foram coletadas 500 amostras para cada algoritmo de encriptação. E para o FPGA configurado sem o algoritmo de encriptação, que tem por objetivo apenas mensurar consumo de energia de LEDs e módulos incorporados para a simulação, 500 amostras também foram selecionadas.

Tabela 4.1. Amostras selecionadas e dados encriptados durante a simulação

Cenário	**Algoritmo de Encriptação**	**Dados Encriptados (Kbytes)**	**Quantidade de amostras selecionadas (por rodada de Encriptação)**
Dinâmico	AES	160	100
	CLEFIA	160	
	PRESENT	80	

Após a coleta dos dados de consumo de cada algoritmo, foram calculadas as médias aritméticas para cada cenário, e realizados os devidos cálculos de diferença dos cenários, Estático e Dinâmico, com o estado do FPGA configurado sem encriptação, sendo estes os dados apresentados e discutidos no capítulo a seguir.

Capítulo 5

Hardwares de Criptografia

Nesta seção são apresentadas as arquiteturas dos algoritmos AES, PRESENT e CLEFIA implementadas em FPGA, de forma individual, a simulação e validação destas implementações, bem como as ferramentas e dispositivos utilizados para o alcance dos objetivos propostos nesta dissertação.

5.1 Implementação AES

O *hardware* de criptografia do AES implementado neste trabalho foi baseado no modelo para encriptação com bloco e chave de 128 bits, descrito em Stallings (2008) e de acordo com os códigos em VHDL da implementação em FPGA cedida por Palmeira *et al*. (2014), com a reformulação e reorganização de algumas entidades do algoritmo cedido em VHDL, visando o correto funcionamento e uma melhor compreensão do mesmo, que serão descritas ao longo deste tópico.

A arquitetura implementada foi realizada de forma modular, onde cada módulo (ou entidade) realiza determinada função no processo de encriptação, sendo realizada a conexão destes módulos com a unidade de controle, de modo a formar a arquitetura geral do *hardware* criptográfico desenvolvido e implementado. A seguir são descritos o desenvolvimento e a funcionalidade de cada módulo que compõe esta arquitetura, bem como o teste e validação geral do *hardware*.

5.1.1 Módulo S-Box

Este módulo visa a realização das operações de substituição durante a encriptação do algoritmo, essas substituições são realizadas na forma de *byte* para *byte*, através de uma tabela de substituição (S-Box), onde os valores de entrada são mapeados por meio das variáveis *x* e *y* constantes na tabela, gerando assim novos valores a cada rodada de encriptação. A figura 5.1 (descrita em hexadecimal), mostra a tabela de substituição do algoritmo AES.

O algoritmo em VHDL da operação SubByte é realizada utilizando as atrições *WITH*, *SELECT*, *WHEN*, para os dados em binário. O Quadro 5.1 exibe parte desta descrição (linha 1 da S-Box).

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 5.1. Tabela de Substituição (S-Box) do AES (NIST, 2001)

```
40  architecture Behavioral of s_box is
41
42  begin
43
44      with s_in  (7 downto 0) select
45           s_out (7 downto 0) <=
46
47           -- primeira linha (S-box de AES)
48           "01100011" WHEN "00000000", --(X"63")
49           "01111100" WHEN "00000001", --(X"7C")
50           "01110111" WHEN "00000010", --(X"77")
51           "01111011" WHEN "00000011", --(X"7B")
52           "11110010" WHEN "00000100", --(X"F2")
53           "01101011" WHEN "00000101", --(X"6B")
54           "01101111" WHEN "00000110", --(X"6F")
55           "11000101" WHEN "00000111", --(X"C5")
56           "00110000" WHEN "00001000", --(X"30")
57           "00000001" WHEN "00001001", --(X"01")
58           "01100111" WHEN "00001010", --(X"67")
59           "00101011" WHEN "00001011", --(X"2B")
60           "11111110" WHEN "00001100", --(X"FE")
61           "11010111" WHEN "00001101", --(X"D7")
62           "10101011" WHEN "00001110", --(X"AB")
63           "01110110" WHEN "00001111", --(X"76")
```

Quadro 5.1. Parte do algoritmo em VHDL que compõe o módulo S-Box

Vale ressaltar que nesta implementação de AES, a cada rodada de encriptação, um vetor de dados de 128 bits (16 bytes) passa pela operação de substituição. Desta forma na arquitetura geral, foram instanciados 16 módulos S-Box, sendo que este instanciamento foi a principal alteração no código em VHDL, tendo em visto que, no código cedido havia apenas o instanciamento de 8 módulos S-Box, sendo capaz de substituir apenas 64 bits (8 bytes) a cada rodada de encriptação, o que influencia diretamente na latência do processo de encriptação.

5.1.2 Módulo ShiftRow

Neste módulo uma permutação simples é realizada, onde os bytes de uma linha da matriz são rotacionados em grupos de quatro bytes, a Figura 5.2 ilustra como esta permutação ocorre.

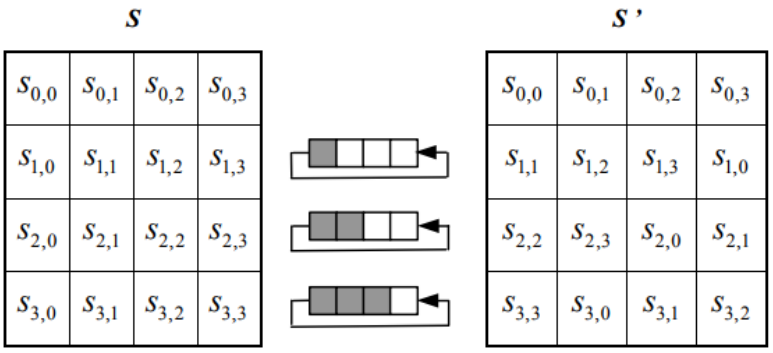


Figura 5.2. Deslocamento de linhas (ShiftRow) do AES (NIST, 2001)

No algoritmo do módulo ShiftRow, primeiramente o vetor de dados de 128 bits é organizado na forma de uma matriz de 16 bytes ordenados por coluna na matriz. Em seguida cada byte das linhas da matriz (com exceção da linha 1) sofrem as permutações (conforme Figura 5.2) e posteriormente são reorganizados na forma de um vetor de 128 bits. As alterações realizadas no código em VHDL deste módulo, foram apenas organizacionais, na declaração de sinais, tipos e subtipos, visando uma melhor compreensão dos mesmos.

O Quadro 5.2 descreve parte do algoritmo utilizado no módulo ShiftRow.

```
39  architecture Behavioral of shift_row is
40
41  SUBTYPE MIN_B is STD_LOGIC_VECTOR (7 downto 0);
42  TYPE MATRIX   is ARRAY (15 downto 0) of MIN_B;
43  SIGNAL box_columns, columns_out: MATRIX;
44
45  BEGIN
46      vector_to_matrix: -- converte o vetor em matriz
47          PROCESS(sr_in)
48          BEGIN
49              FOR i IN 15 DOWNTO 0 LOOP
50                  box_columns(15-i) <= sr_in(8*i+7 DOWNTO 8*i);
51              END LOOP;
52          END PROCESS vector_to_matrix;
53
54      --Deslocamento de linhas do AES (shift rows)...
65      columns_out(0)  <=  box_columns(0);
66      columns_out(1)  <=  box_columns(5);
67      columns_out(2)  <=  box_columns(10);
68      columns_out(3)  <=  box_columns(15);
69      -- 2
70      columns_out(4)  <=  box_columns(4);
71      columns_out(5)  <=  box_columns(9);
72      columns_out(6)  <=  box_columns(14);
73      columns_out(7)  <=  box_columns(3);
```

Quadro 5.2. Parte do algoritmo em VHDL do Módulo ShiftRow

5.1.3 Módulo MixColumns

A operação MixColumns realiza multiplicações lineares em GF(2^8) (*Galois Field*) sobre cada grupo de quatro bytes das colunas da matriz, proporcionando assim uma influência de cada byte do grupo sobre todos os outros bytes. A Figura 5.3 mostra o modelo para esta operação.

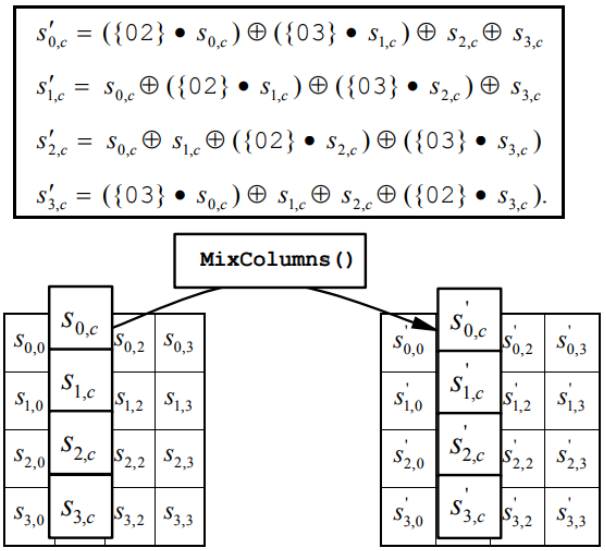


Figura 5.3. Ilustração da operação MixColumns do AES (NIST, 2001)

Na descrição em VHDL de MixColumns, semelhantemente ao módulo ShiftRow, o vetor de 128 bits é subdividido em 16 bytes e organizados em uma matriz ordenados por coluna. Porteriormente, os bytes de cada coluna passam por 2 processos que satisfazem as multiplicações por 2 e por 3 em GF (2^8) (*Galois Field*), respectivamente, seguido de operações XOR dos valores obtidos (conforme modelo da Figura 5.3) com os demais bytes da coluna que não sofrem a multiplicação. Por fim, os bytes resultantes são reorganizados na forma de um vetor de 128 bits.

Parte do código que descreve o módulo MixColumns é mostrado no Quadro 5.3.

```
67      --multiplicação por 2 (shift_left(X))
68      mul_2:
69          PROCESS(c_in, shifted_2)
70          BEGIN
71              FOR i IN  15 DOWNTO 0 LOOP
72                  shifted_2(i) <= c_in(i) & '0';  -- 2*X
73                  IF (shifted_2(i)(8)='1') THEN -- overflow
74                       mult_2(i) <= shifted_2(i)(7 downto 0) XOR "00011011"; -- deduction mod X^8+X^4+X^3+X^1+X^0
75                  ELSE
76                       mult_2(i) <= shifted_2(i)(7 downto 0);
77                  END IF;
78              END LOOP;
79          END PROCESS mul_2;
80
81      --multiplicação por 3 (shift_left(X) xor X)
82      mul_3:
83          PROCESS(c_in, shifted_3, xored)
84          BEGIN
85              FOR i IN  15 DOWNTO 0 LOOP
86                  shifted_3(i) <= c_in(i) & '0';
87                  xored(i) <= shifted_3(i) XOR '0' & c_in(i);
88                  IF (xored(i)(8)='1') THEN   -- overflow
89                       mult_3(i) <= xored(i)(7 downto 0) XOR "00011011";
90                  ELSE
91                       mult_3(i) <= xored(i)(7 downto 0);
92                  END IF;
93              END LOOP;
94          END PROCESS mul_3;
95
96          --Multiplicação de Colunas do AES (Mix Columns)...
08      c_out(0)  <= mult_2(0) XOR mult_3(1) XOR c_in(2) XOR c_in(3);
09      c_out(4)  <= mult_2(4) XOR mult_3(5) XOR c_in(6) XOR c_in(7);
10      c_out(8)  <= mult_2(8) XOR mult_3(9) XOR c_in(10) XOR c_in(11);
11      c_out(12) <= mult_2(12) XOR mult_3(13) XOR c_in (14) XOR c_in(15);
```

Quadro 5.3. Parte do algoritmo em VHDL do Módulo MixColumns

Apenas modificações na organização e declaração de sinais, tipos e subtipos foram realizadas no código em VHDL originalmente cedido.

5.1.4 Módulo Key Expansion

Este módulo visa atender as operações de expansão da chave original, gerando chaves para cada rodada de encriptação do algoritmo.

Neste processo, o vetor de bits da chave (128 bits) é divido em 4 palavras (Words) de 32 bits: Word(0), Word(1), Word(2) Word(3); cada palavra da chave que está sendo gerada para determinada rodada, passa por operações XOR com a palavra da chave da rodada anterior, sendo que a palavra Word(3) passa ainda por processos de permutação (RotWord), Substituição (SubWord), e cálculos de uma constante da rodada (Rcon), que utiliza constantes predefinidas pelos autores do algoritmo (armazenadas em uma memória ROM), e que ao final destes processos, também passam por operações XOR para geração da chave da rodada.

A figura 5.4 ilustra como o processo de Expansão da Chave é realizado.

O módulo Key Expansion descrito em VHDL utiliza 4 instâncias da operação Sub-Byte, descrita anteriormente, para realizar a operação de substituição em que a Word (3) é submetida. Um vetor T foi declarado na Key Expansion, visando atender a necessidade da operação Rcon, na qual apenas um byte da saída da SubWord é submetido a uma operação XOR com a constante da rodada já definida. O resultado da operação T é então inserido ao processo de expansão da chave, através da operação XOR com cada palavra chave da rodada anterior, conforme modelo descrito na figura 5.4.

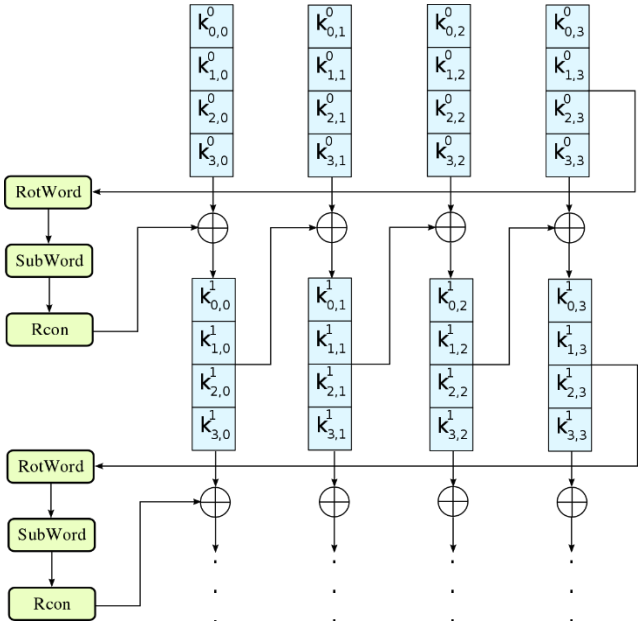


Figura 5.4. Processo de Expansão da Chave (Key Expansion) do AES.

Fonte: Wikipedia (Rijndael key schedule)

Parte do algoritmo do módulo Key Expansion é mostrado no Quadro 4.

```
60      --chave original dividida em quatro words
61      key_word(0) <= key_in (127 downto 96);
62      key_word(1) <= key_in (95 downto 64);
63      key_word(2) <= key_in (63 downto 32);
64      key_word(3) <= key_in (31 downto 0);
65
66      --RotWord - deslocamento a esquerda dos bytes (Key_Word 3)
67      temp_shift <= key_word(3)(23 downto 16) &
68                    key_word(3)(15 downto 8) &
69                    key_word(3)(7 downto 0) &
70                    key_word(3)(31 downto 24);
71
72      --passando pela S-Box (4 bytes)
73      sbox_lookup: s_box_n PORT MAP(s_word_in => temp_shift, s_word_out => temp_sbox);
74
75      --calculando RCON (Variável T)
76      upperbyte_trans <= temp_sbox (31 downto 24) XOR rcon;
77
78      --vetor T calculado
79      T <= upperbyte_trans & temp_sbox (23 downto 0);
80
81      --(XORing) calculando as próximas Key Words
82      next_key_word(0) <= key_word(0) XOR T;
83      next_key_word(1) <= key_word(1) XOR key_word(0) XOR T;
84      next_key_word(2) <= key_word(2) XOR key_word(1) XOR key_word(0) XOR T;
85      next_key_word(3) <= key_word(3) XOR key_word(2) XOR key_word(1) XOR key_word(0) XOR T;
86
87      --converting Words em vetor de 128 bits
88      key_out <= next_key_word(0) & next_key_word(1) & next_key_word(2) & next_key_word(3);
```

Quadro 5.4. Parte do algoritmo em VHDL do módulo Key Expansion

Para o algoritmo implementado, com chave de 128 bits, um total de 10 chaves das rodadas são geradas a partir da chave original, que são utilizadas durante o processo de encriptação.

A criação de uma pequena memória ROM para armazenamento das constantes Rcon utilizadas no cálculo da chave da rodada, e o devido instanciamento e interconexão dentro

do módulo Key Expansion, foram as principais modificações no código em VHDL originalmente cedido deste módulo. Antes, essas constantes eram provenientes do módulo de Controle, sendo enviadas a Key Expansion a cada rodada, conforme a mudança de estado.

5.1.5 Módulo de Controle

O módulo de controle é a unidade que gerencia os estados do algoritmo de encriptação, sendo baseada no modelo de Mealy de um circuito sequencial. Ao todo são quatro estados que compõe o módulo de controle, sendo estes:

- **P**: Parado ou Estático;
- **C**: Carrega os registradores com o Texto Plano (128 bits) e com a Chave (128 bits) e executa a operação inicial de adição da chave da rodada (AddRoundKey);
- **E**: Executa as rodadas de encriptação necessárias. No desenho deste algoritmo são executadas 10 rodadas de encriptação do texto;
- **L**: Sinaliza para a leitura do texto cifrado após o processo de encriptação;

A Figura 5.5 exibe o diagrama de estados do Módulo de Controle.

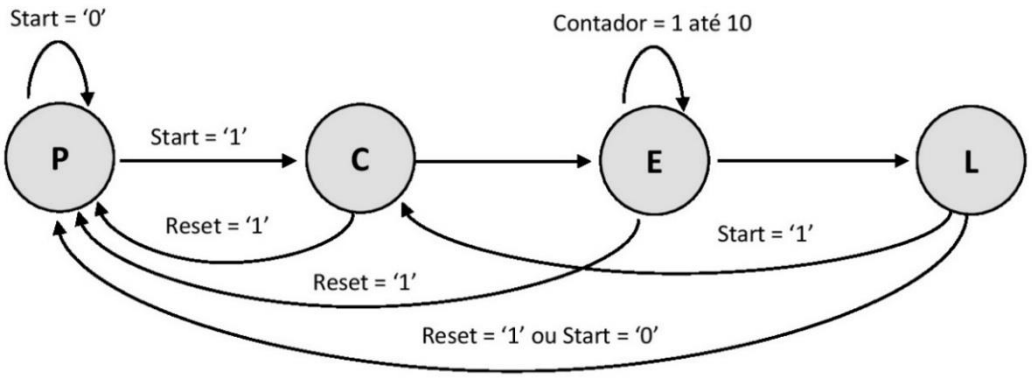


Figura 5.5. Diagrama de Estados do Módulo de Controle do AES

As variáveis externas que influênciam os estados são Start e Reset, e o contador de rodadas a variável interna. As saídas da unidade de controle, habilita/desabilita os registradores do texto e da chave, e ainda do contador, e gerenciam os seletores dos multiplexadores do texto e da chave. Mais detalhes do gerenciamento do módulo de controle podem ser observados através da Figura 5.5.

A descrição comportamental em VHDL do módulo de Controle foi reformulada, passando de 11 estados, no código originalmente cedido, para apenas 4 estados após as alterações. Isso foi possível devido a transferência das constantes Rcon para o módulo Key Expansion, bem como a implementação de um contador na arquitetura geral do algoritmo, visando o controle da encriptação (estado E).

Parte do algoritmo em VHDL do módulo de Controle pode ser observado no Quadro 5.5.

```
51              begin
52                  case state is
53                      -- Aguardando o início do processo
54                      when SM_STOP =>
55                          ready <= '0';
56                          reg_enable <= '0';
57                          ctrl_mux <= '0';
58                          sel_mux <= '0';
59                          if (start = '1') then
60                          next_state <= SM_LOAD;
61                          else
62                          next_state <= SM_STOP;
63                          end if;
64
65                      -- Carregando texto e chave
66                      when SM_LOAD =>
67                          ready <= '0';
68                          reg_enable <= '1';
69                          ctrl_mux <= '0';
70                          sel_mux <= '0';
71                          next_state <= SM_ENC;
72
73                      --Encriptação
74                      when SM_ENC =>
75                           ready <= '0';
76                           reg_enable <= '1';
77                           ctrl_mux <= '1';
78                           sel_mux <= '0';
79                           if (round = "0001") then
80                           next_state <= SM_ENC;
81                           elsif (round = "1010") then
82                           sel_mux <= '1';
83                           next_state <= SM_READY;
84                           else
85                           next_state <= SM_ENC;
86                           end if;
```

Quadro 5.5. Parte do algoritmo em VHDL do Módulo de Controle

5.1.6 Arquitetura AES Encriptação

A arquitetura geral do algoritmo de encriptação AES deste trabalho é formada pela interconexão dos módulos já mencionados anteriormente, com o acréscimo de alguns módulos e circuitos, visando o correto funcionamento da encriptação, sendo estes:

- **Módulo Registrador do Texto (Reg_Texto)**: registra o Texto Plano e o Texto Encriptado a cada rodada do algoritmo, com tamanho de 128 bits;
- **Módulo Registrador da Chave (Reg_Chave)**: registra a Chave original e as chaves geradas a cada rodada do algoritmo, com tamanho de 128 bits;
- **Contador**: contador sequencial de 4 bits para controle do processo de encriptação;

- **Mux 2:1 (1)**: responsável pela seleção do Texto Plano ou do texto encriptado a cada rodada;

- **Mux 2:1 (2)**: responsável pela seleção da Chave original ou da Chave Expandida a cada rodada;

- **Mux 2:1 (3)**: responsável pela seleção do texto em processo de encriptação após o módulo ShiftRow, ou após o módulo MixColumns, visando atender a rodada 10 do algoritmo, na qual a operação MixColumns não é selecionada;

- **AddRoundKey**: circuito responsável pela operação XOR (bit a bit) entre o texto e a chave a cada rodada de encriptação;

Através da Figura 5.6 é possível observar como esta interconexão entre os módulos e circuitos é realizada, as entradas (Texto Plano, Chave, Start e Reset) e as saídas (Texto Encriptado e Pronto) do FPGA, sendo que a saída (Pronto), sinaliza que o processo de encriptação foi realizado e que o texto encriptado já pode ser lido. Podendo ser observado ainda toda a interconexão interna do circuito.

Em VHDL a arquitetura geral do AES Encriptação é realizada através de um módulo, onde são declaradas as entradas e saídas do *Hardware* Criptográfico, e também é feito o instanciamento dos outros módulos (ou entidades) e circuitos que compõe o algoritmo, bem como o mapeamento das portas de cada módulo ou circuito, fazendo uso de conectores (sinais) para essa interconexão. Esse Módulo Geral do AES Encriptação foi nomeado de AES_ENC.

Parte do algoritmo descrito em VHDL do módulo AES_ENC pode ser visualizada no Quadro 5.6.

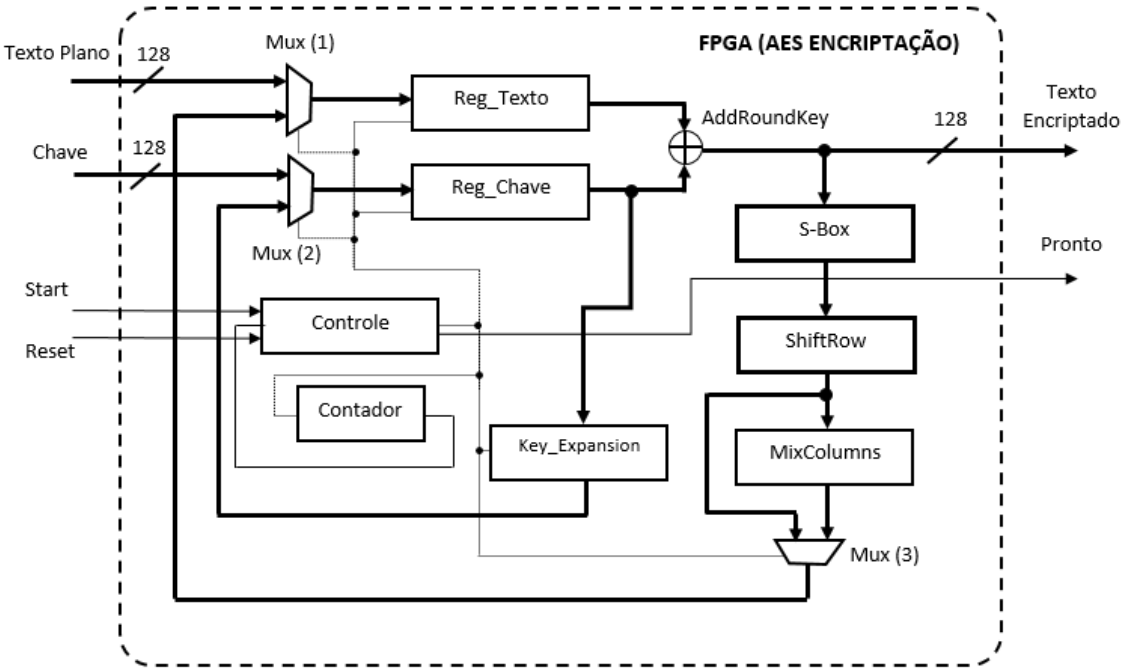


Figura 5.6. Arquitetura Geral AES-128 Encriptação.

```
179         SM: AES_MaqEst port map(
180             clk => clk,
181             reset => reset,
182             start => start,
183             ready => ready_sig,
184             round => sig_round,
185             ctrl_mux => mux_ctrl,
186             reg_enable => sig_enable,
187             sel_mux => sel
188         );
189 
190         key_S : key_shedule port map(
191             clk   => clk,
192             reset => reset,
193             en_rcon => sig_enable,
194             key_in => keyfout,
195             key_out => kupd
196         );
197 
198         RD : contador port map(
199             clk => clk,
200             reset => reset,
201             cnt_res => sig_enable,
202             round => sig_round
203         );
204 
205         --operação AddRoudKey
206         ciph <= toXor XOR keyfout; -- Texto XOR Chave rodada
207 
208         -- saída texto cifrado
209         ciphertext <= ciph;
210         ready <= ready_sig; -- pronto
211         round <= sig_round; -- contador de rodadas
```

Quadro 5.6. Parte do algoritmo em VHDL do módulo AES_ENC

5.1.7 Testbench AES

Nesta subseção é apresentado o Testbench (Teste de Bancada), onde é realizado uma simulação do circuito, utilizando o *software* Vivado *Design Suite* versão 2016.3 com o objetivo de verificar o correto funcionamento do algoritmo, com base em vetores de teste já predeterminados pelos autores do algoritmo.

Os vetores de teste do AES utilizados no Testbench estão baseados nas especificações do NIST (2001), para o algoritmo com chave de 128 bits e 10 rodadas de encriptação. Os vetores de teste completo (descritos em hexadecimal) podem ser visualizados nos Quadros 5.7 (parte 1) e 5.8 (parte 2).

1.			**Texto Plano**	**00112233445566778899aabbccddeeff**
2.			**Chave**	**000102030405060708090a0b0c0d0e0f**
3.	Rodada	0	Entrada Texto	00112233445566778899aabbccddeeff
4.			Chave	000102030405060708090a0b0c0d0e0f
5.	Rodada	1	Texto	000102030405060708090a0b0c0d0e0f0
6.			S-Box	63cab7040953d051cd60e0e7ba70e18c
7.			ShiftRows	6353e08c0960e104cd70b751bacad0e7
8.			MixColumns	5f72641557f5bc92f7be3b291db9f91a
9.			Key Shedule	d6aa74fdd2af72fadaa678f1d6ab76fe
10.	Rodada	2	Texto	89d810e8855ace682d1843d8cb128fe4
11.			S-Box	a761ca9b97be8b45d8ad1a611fc97369
12.			ShiftRows	a7be1a6997ad739bd8c9ca451f618b61
13.			MixColumns	ff87968431d86a51645151fa773ad009
14.			Key Shedule	b692cf0b643dbdf1be9bc5006830b3fe
15.	Rodada	3	Texto	4915598f55e5d7a0daca94fa1f0a63f7
16.			S-Box	3b59cb73fcd90ee05774222dc067fb68
17.			ShiftRows	3bd92268fc74fb735767cbe0c0590e2d
18.			MixColumns	4c9c1e66f771f0762c3f868e534df256
19.			Key Shedule	b6ff744ed2c2c9bf6c590cbf0469bf41
20.	Rodada	4	Texto	fa636a2825b339c940668a3157244d17
21.			S-Box	2dfb02343f6d12dd09337ec75b36e3f0
22.			ShiftRows	2d6d7ef03f33e334093602dd5bfb12c7
23.			MixColumns	6385b79ffc538df997be478e7547d691
24.			Key Shedule	47f7f7bc95353e03f96c32bcfd058dfd

Quadro 5.7. Vetores de Teste AES-128 Encriptação (Parte 1)

25.			Texto	247240236966b3fa6ed2753288425b6c
26.			S-Box	36400926f9336d2d9fb59d23c42c3950
27.	Rodada	5	ShiftRows	36339d50f9b539269f2c092dc4406d23
28.			MixColumns	f4bcd45432e554d075f1d6c51dd03b3c
29.			Key Shedule	3caaa3e8a99f9deb50f3af57adf622aa
30.			Texto	c81677bc9b7ac93b25027992b0261996
31.			S-Box	e847f56514dadde23f77b64fe7f7d490
32.	Rodada	6	ShiftRows	e8dab6901477d4653ff7f5e2e747dd4f
33.			MixColumns	9816ee7400f87f556b2c049c8e5ad036
34.			Key Shedule	5e390f7df7a69296a7553dc10aa31f6b
35.			Texto	c62fe109f75eedc3cc79395d84f9cf5d
36.			S-Box	b415f8016858552e4bb6124c5f998a4c
37.	Rodada	7	ShiftRows	b458124c68b68a014b99f82e5f15554c
38.			MixColumns	c57e1c159a9bd286f05f4be098c63439
39.			Key Shedule	14f9701ae35fe28c440adf4d4ea9c026
40.			Texto	d1876c0f79c4300ab45594add66ff41f
41.			S-Box	3e175076b61c04678dfc2295f6a8bfc0
42.	Rodada	8	ShiftRows	3e1c22c0b6fcbf768da85067f6170495
43.			MixColumns	baa03de7a1f9b56ed5512cba5f414d23
44.			Key Shedule	47438735a41c65b9e016baf4aebf7ad2
45.			Texto	fde3bad205e5d0d73547964ef1fe37f1
46.			S-Box	5411f4b56bd9700e96a0902fa1bb9aa1
47.	Rodada	9	ShiftRows	54d990a16ba09ab596bbf40ea111702f
48.			MixColumns	e9f74eec023020f61bf2ccf2353c21c7
49.			Key Shedule	549932d1f08557681093ed9cbe2c974e
50.			Texto	bd6e7c3df2b5779e0b61216e8b10b689
51.	Rodada	10	S-Box	7a9f102789d5f50b2beffd9f3dca4ea7
52.			ShiftRows	7ad5fda789ef4e272bca100b3d9ff59f
53.			Key Shedule	13111d7fe3944a17f307a78b4d2b30c5
54.			**Texto Encriptado**	**69c4e0d86a7b0430d8cdb78070b4c55a**

Quadro 5.8. Vetores de Teste AES-128 Encriptação (Parte 2)

Após a simulação com os vetores de teste acima mencionados, foi verificado ao final de cada rodada, se os valores na simulação correspondiam aos valores do vetor de teste, sendo confirmado a funcionalidade correta de cada módulo e da arquitetura geral do algoritmo AES Encriptação deste trabalho.

O Quadro 5.9 exibe os valores da simulação (com vetores descritos em hexadecimal) para as rodadas 1, 6, 10 e a rodada de leitura (11), na qual é sinalizada a finalização do processo de encriptação de um bloco de texto (128 bits), através da saída PRONTO. Na simulação os sinais (sig_sbox[127:0], sig_shiftrow [127:0], sig_mixcolumns [127:0]), representam as saídas dos módulos S-Box, ShiftRow e MixColumns, respectivamente.

Quadro 5.9. Parte da simulação AES-128 Encriptação

Portanto, a arquitetura geral do AES implementada neste trabalho, foi submetida a simulação e análise, onde comprovou-se o seu correto funcionamento.

5.2 Implementação PRESENT

Neste trabalho, o *hardware* de criptografia do algoritmo PRESENT, foi baseado no modelo de seus desenvolvedores, Bogdanov *et al*. (2007), para a versão com bloco de 64 bits e chave de 80 bits, e de acordo com os códigos em VHDL da implementação em FPGA, elaborada por Gajewski (2014), com pequenas alterações, visando uma melhor compreensão do código VHDL, que será descrito ao longo deste tópico.

Semelhante a implementação do AES, a arquitetura de PRESENT também foi realizada de forma modular, onde cada operação específica é realizada por um módulo ou componente, gerenciado pelo módulo de controle, que devidamente interconectados, formam um módulo unificado de encriptação do algoritmo PRESENT, nomeado de PRESENT_ENC. O desenvolvimento e a funcionalidade de cada módulo que compõe esta arquitetura, bem como a simulação e o teste e validação geral do *hardware* são descritos a seguir.

5.2.1 Módulo sBoxLayer

Este módulo visa a realização de operações de substituição, baseado em uma tabela de consulta. Estas operações ocorrem na ordem de *nibble* (4 bits para 4 bits). A Figura 5.7 mostra a tabela de consulta de PRESENT (descrita em hexadecimal), onde *x* representa o valor de entrada e S[*x*] representa o valor de substituição correspondente.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figura 5.7. Tabela de Substituição do PRESENT (Bogdanov et al., 2007)

O algoritmo em VHDL da operação de substituição utiliza as atribuições WHEN e ELSE, sendo que o código utilizado (Gajewski, 2014), não sofreu alteração para a implementação no FPGA objeto deste trabalho. Parte do algoritmo da operação sBoxLayer é exibida no Quadro 5.10 (em hexadecimal).

Visando atender a necessidade de substituição de um vetor de 64 bits de texto por rodada de encriptação, foram instanciados 16 módulos que realizam a operação de substituição para compor o módulo sBoxLayer.

```
entity sBoxLayer is
    port (
        input : in std_logic_vector(3 downto 0);
        output : out std_logic_vector(3 downto 0)
    );
end sBoxLayer;

architecture Behavioral of sBoxLayer is

    begin
        output <=    x"C" when input = x"0" else
                     x"5" when input = x"1" else
                     x"6" when input = x"2" else
                     x"B" when input = x"3" else
                     x"9" when input = x"4" else
                     x"0" when input = x"5" else
                     x"A" when input = x"6" else
                     x"D" when input = x"7" else
                     x"3" when input = x"8" else
                     x"E" when input = x"9" else
                     x"F" when input = x"A" else
                     x"8" when input = x"B" else
                     x"4" when input = x"C" else
                     x"7" when input = x"D" else
                     x"1" when input = x"E" else
                     x"2" when input = x"F" else
                     "ZZZZ";
    end Behavioral;
```

Quadro 5.10. Parte do algoritmo do módulo sBoxLayer

5.2.2 Módulo pLayer

Este módulo realiza operações de permutação simples (bit a bit) de um vetor de 64 bits, com os dados oriundos da sBoxLayer. A Figura 5.8 ilustra as mudanças de posições sofridas por cada bit através da passagem por este módulo.

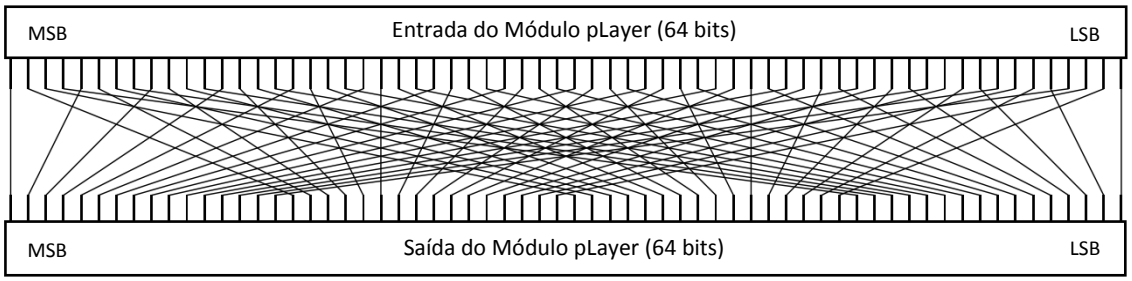


Figura 5.8. Operação do Módulo pLayer do PRESENT (Adaptado de Bogdanov et al., 2007)

Vale ressaltar que para a operação pLayer, a ordenação do bit mais significativo (MSB) para o bit menos significativo (LSB) do vetor de 64 bits, ocorre da esquerda para a direita, conforme Figura 5.8.

Parte do algoritmo do módulo pLayer descrito em VHDL é mostrado no Quadro 5.11. Sendo que o código original não sofreu alteração.

```
architecture Behavioral of pLayer is
begin
            output(0) <= input(0);
            output(16) <= input(1);
            output(32) <= input(2);
            output(48) <= input(3);
            output(1) <= input(4);
            output(17) <= input(5);
            output(33) <= input(6);
            output(49) <= input(7);
            output(2) <= input(8);
            output(18) <= input(9);
            output(34) <= input(10);
            output(50) <= input(11);
            output(3) <= input(12);
            output(19) <= input(13);
            output(35) <= input(14);
            output(51) <= input(15);
            output(4) <= input(16);
            output(20) <= input(17);
            output(36) <= input(18);
            output(52) <= input(19);
            output(5) <= input(20);
            output(21) <= input(21);
            output(37) <= input(22);
            output(53) <= input(23);
            output(6) <= input(24);
            output(22) <= input(25);
            output(38) <= input(26);
            output(54) <= input(27);
            output(7) <= input(28);
```

Quadro 5.11. Parte do algoritmo do Módulo pLayer

5.2.3 Módulo Keyupd

Este módulo realiza operações de atualização de chave original (geração de chave da rodada), visando atender a operação de adição da chave da rodada (AddRoundKey), que é realizada juntamente com o texto a cada rodada de encriptação.

A atualização da chave a cada rodada ocorre conforme os passos a seguir:

1. $[k_{79}k_{78} \ldots k_1k_0] = [k_{18}k_{17} \ldots k_{20}k_{19}]$
2. $[k_{79}k_{78}k_{77}k_{76}] = \textit{Substituição}\ [k_{79}k_{78}k_{77}k_{76}]$
3. $[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus$ rodada (bits)

Primeiramente, os 19 bits menos significativos (LSB) da chave são permutados e passam a ser os 19 bits mais significativos (MSB) da chave que está sendo atualizada (passo 1). Posteriormente, os 4 bits mais a esquerda passam por uma operação de substituição (sBoxLayer). Finalmente os bits da posição 19, 18, 17, 16, 15 da chave que está sendo atualizada, são submetidos a uma operação XOR com os bits de que representam a rodada. Após estas operações, a saída do módulo Keyupd recebe a chave atualizada.

Vale ressaltar que a operação da chave da rodada (AddRoundKey), é realizada através de uma operação XOR dos bits do bloco de texto (64 bits), com os 64 bits mais a esquerda da chave atualizada, a cada rodada de encriptação.

O código em VHDL do módulo Keyupd sofreu apenas alterações na declaração de sinais e em sua organização, visando uma melhor compreensão do mesmo. O Quadro 5.12 ilustra parte do algoritmo do módulo Keyupd.

```
35      component sBoxLayer is
36          port(
37              input : in std_logic_vector(3 downto 0);
38              output : out std_logic_vector(3 downto 0)
39          );
40      end component;
41
42      signal sbox_in   : std_logic_vector(3 downto 0);
43      signal sbox_out  : std_logic_vector(3 downto 0);
44      signal keytemp : std_logic_vector(79 downto 0);
45
46      begin
47
48          --chave em processo de atualização
49          --Permutação dos 19 bits LSB para MSB
50          keytemp <= key(18 downto 0) & key(79 downto 19);
51
52          --bits que passam pela substituição
53          sbox_in <= keytemp(79 downto 76);
54
55          -- instanciamento de uma sBoxLayer
56          s1: sBoxLayer port map(
57          input => sbox_in,
58          output => sbox_out
59          );
60
61          --Saída da chave atualizada
62          keyout(79 downto 76)<= sbox_out;
63          keyout(75 downto 20)<= keytemp(75 downto 20);
64          keyout(19 downto 15)<= keytemp(19 downto 15) xor round; -- XOR com contador de rodadas
65          keyout(14 downto 0) <= keytemp(14 downto 0);
66
67      end Behavioral;
```

Quadro 5.12. Parte do algoritmo do Módulo Keyupd

5.2.4 Módulo de Controle

Este módulo realiza o gerenciamento dos módulos e circuitos visando a habilitação/seleção correta dos dados a cada rodada de encriptação, bem como a devida sinalização ao final do processo de encriptação de um texto plano (64 bits).

Semelhante a arquitetura do módulo de Controle do AES, este módulo possui 4 estados, sendo estes:

- **P**: Parado ou Estático;
- **C**: Carrega os registradores com o Texto Plano (64 bits) e com a Chave (80 bits) e executa a operação inicial de adição da chave da rodada (AddRoundKey);
- **E**: Executa as rodadas de encriptação necessárias. Nesta implementação são executadas 31 rodadas de encriptação;
- **L**: Sinaliza para a leitura do texto cifrado após o processo de encriptação;

A Figura 5.9 ilustra o diagrama de estados deste módulo de controle.

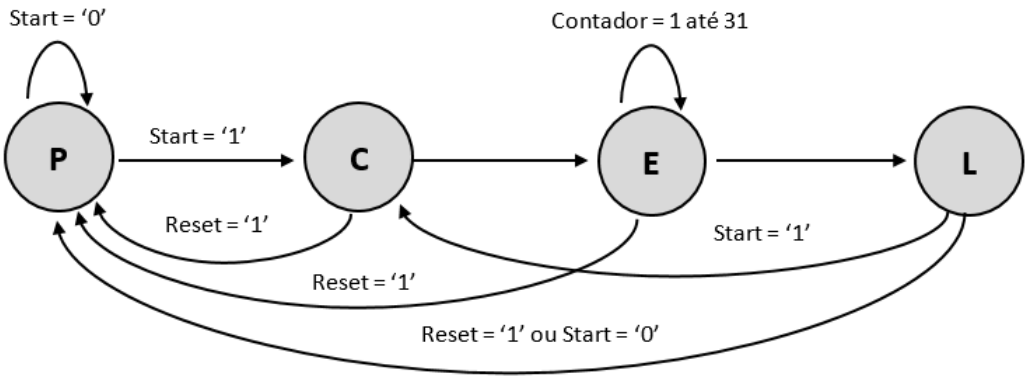


Figura 5.9. Diagrama de estados do Módulo de Controle do PRESENT

As diferenças entre o módulo de controle do PRESENT em relação ao AES, é a quantidade de rodadas em que o algoritmo permanece no estado E, tendo em vista o número maior de rodadas de encriptação, bem como uma quantidade menor de multiplexadores sendo gerenciados.

A principal alteração realizada no código em VHDL original foi o desmembramento do estado C, que antes estava incluído no estado E. Isto para uma melhor compreensão do processo de encriptação, e sem prejuízo com relação a consumo de recursos do FPGA.

Parte do algoritmo em VHDL do módulo de Controle pode ser observado no Quadro 5.13.

```
45              case state is
46                  -- Aguardando o início do processo
47                  when SM_STOP =>
48                      pronto <= '0'; -- Encriptação finalizada (Ler)
49                      cnt_res <= '0'; -- gatilho do contador
50                      ctrl_mux <= '0'; -- seletor dos mux
51                      RegEn <= '0'; -- habilita registradores
52                      if (start = '1') then
53                          next_state <= SM_ENC;
54                      else
55                          next_state <= SM_LOAD;
56                      end if;
57
58                  -- carregando dados
59                  when SM_LOAD =>
60                      pronto <= '0';
61                      RegEn <= '1';
62                      cnt_res <= '1'; -- gatilho do contador
63                      ctrl_mux <= '0';
64                      next_state <= SM_ENC;
65
66
67                  -- codificando
68                  when SM_ENC =>
69                      pronto <= '0';
70                      RegEn <= '1';
71                      cnt_res <= '1';
72                      ctrl_mux <= '1';
73                          if (round = "00001") then
74                              next_state <= SM_ENC;
75                          -- última rodada
76                          elsif (round = "11111") then
77                              next_state <= SM_READY;
78                          else
79                              next_state <= SM_ENC;
80                          end if;
```

Quadro 5.13. Parte do algoritmo do Módulo de Controle do PRESENT

5.2.5 Arquitetura PRESENT Encriptação

Visando a perfeita funcionalidade do algoritmo de encriptação PRESENT, outros módulos e circuitos complementares foram acrescentados ao desenho, e posteriormente interconectados de forma a satisfazer as características da cifra. Os módulos complementares são semelhantes ao desenho do AES, sendo estes:

- **Módulo Registrador do Texto (Reg_Texto)**: registra o Texto Plano e o Texto Encriptado a cada rodada do algoritmo, com tamanho de 64 bits;
- **Módulo Registrador da Chave (Reg_Chave)**: registra a Chave original e as chaves geradas a cada rodada do algoritmo, com tamanho de 80 bits;
- **Contador**: contador sequencial de 5 bits para controle do processo de encriptação;
- **Mux 2:1 (1)**: responsável pela seleção do Texto Plano ou do texto encriptado a cada rodada;
- **Mux 2:1 (2)**: responsável pela seleção da Chave original ou da Chave Expandida a cada rodada;
- **AddRoundKey**: circuito responsável pela operação XOR (bit a bit) entre o texto e a chave a cada rodada de encriptação (64 bits mais significativos da chave);

A arquitetura geral do algoritmo de encriptação do PRESENT pode ser visualizada na Figura 5.10.

É importante ressaltar que o modelo do PRESENT implementado neste trabalho, é a versão de encriptação, que opera em bloco de texto de 64 bits e chave de 80 bits.

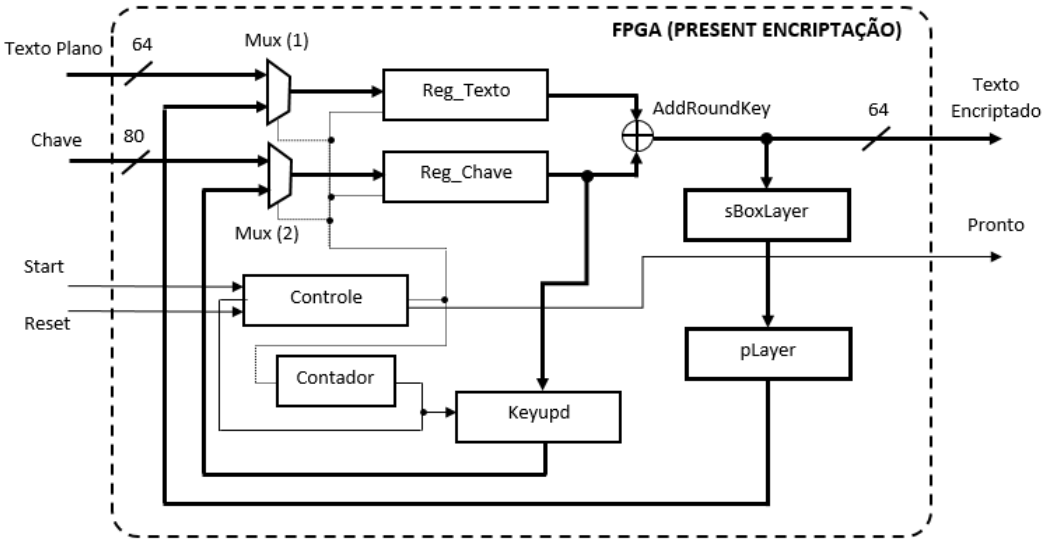


Figura 5.10. Arquitetura geral do algoritmo PRESENT-80 Encriptação

O *hardware* criptográfico do algoritmo PRESENT descrito em VHDL, realiza a interconexão entre os módulos utilizando a declaração de sinais que são mapeados entre as entradas e saídas dos módulos e circuitos mencionados, formando um módulo TOP, com as devidas entrada e saídas do FPGA, nomeado de PRESENT_ENC.

Parte do algoritmo descrito em VHDL do módulo PRESENT_ENC pode ser visualizada no Quadro 5.14.

```
142         p1: pLayer port map(
143             input => sig_sBoxLayer
144             output => sig_pLayer
145         );
146 
147         mixer: keyupd port map(
148             key => keyfout,
149             round => keyround,
150             keyout => sig_kupd
151         );
152 
153         SM: maqEstados port map(
154             start => start,
155             reset => reset,
156             ready => ready_sig,
157             cnt_res => cnt_res,
158             ctrl_mux => mux_ctrl,
159             clk => clk,
160             round => keyround,
161             RegEn => RegEn
162         );
163 
164         count: Contador port map(
165             clk => clk,
166             reset => reset,
167             cnt_res => cnt_res,
168             round => keyround
169         );
170 
171         ciph <= toXor XOR keyfout(79 downto 16); -- Texto Cifrado XOR Chave rodada
172 
173         texto_encriptado <= ciph;
174         pronto <= ready_sig;
```

Quadro 5.14. Parte do algoritmo do módulo PRESENT_ENC

5.2.6 Testbench PRESENT

Para fins de verificação da correta funcionalidade do algoritmo implementado, simulações e testes foram realizados, de acordo com as especificações dos autores Bogdanov *et al*. (2007).

Os vetores de teste do algoritmo de encriptação do PRESENT, com chave de 80 bits é mostrado no Quadro 5.15 (em hexadecimal).

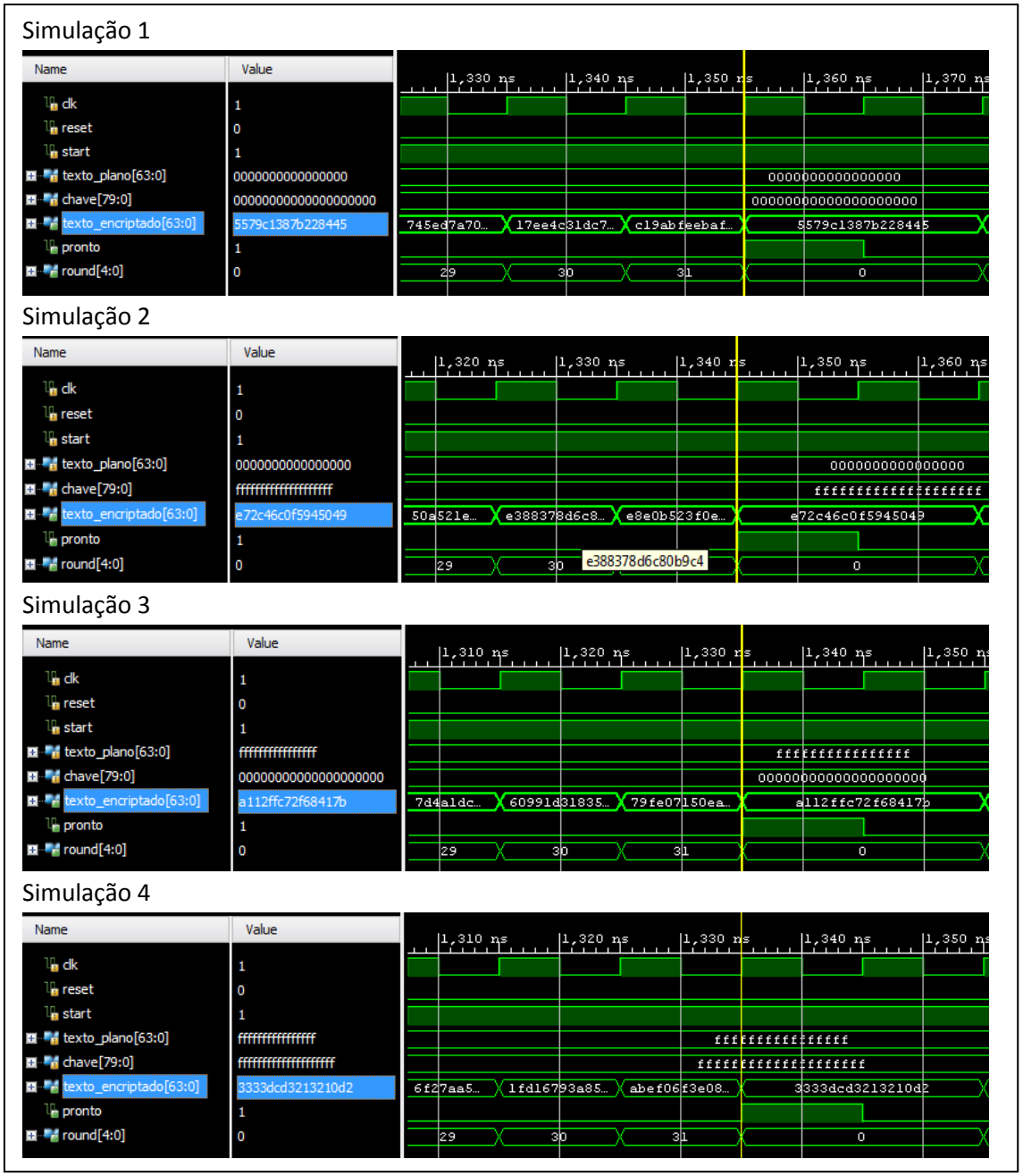
Teste 1	Texto Plano	00000000 00000000
	Chave	00000000 00000000 0000
	Texto Encriptado	**5579C138 7B228445**
Teste 2	Texto Plano	00000000 00000000
	Chave	FFFFFFFF FFFFFFFF FFFF
	Texto Encriptado	**E72C46C0 F5945049**
Teste 3	Texto Plano	FFFFFFFF FFFFFFFF
	Chave	00000000 00000000 0000
	Texto Encriptado	**A112FFC7 2F68417B**
Teste 4	Texto Plano	FFFFFFFF FFFFFFFF
	Chave	FFFFFFFF FFFFFFFF FFFF
	Texto Encriptado	**3333DCD3 213210D2**

Quadro 5.15. Vetores de teste do algoritmo PRESENT (Bogdanov et al., 2007)

No total quatro simulações de encriptação do PRESENT foram realizadas, com o objetivo de verificar a sua correta funcionalidade, comparando com os vetores dos 4 testes mostrados no Quadro 5.15.

A arquitetura implementada, realiza a encriptação de um texto plano (64 bits) em 32 rodadas. Ao final de cada processo de encriptação, o vetor de teste foi comparado com o resultado da simulação e constatado o funcionamento correto do algoritmo.

O Quadro 5.16 ilustra os resultados das simulações realizadas (em hexadecimal).



Quadro 5.16. Resultados das simulações do PRESENT-80 Encriptação

As simulações mostram que os resultados do texto encriptado correspondem aos resultados dos vetores de testes padrão do algoritmo PRESENT, sendo, portanto, confirmada a sua funcionalidade.

5.3 Implementação CLEFIA

O algoritmo em VHDL da cifra de bloco CLEFIA com chave de 128 bits, implementada em FPGA neste trabalho, foi desenvolvido com base nas especificações originais do algoritmo e em uma versão descrita em Verilog disponibilizada pela empresa Sony Corporation (2010).

CLEFIA emprega uma estrutura de Feistel Generalizada de quatro ramos (GF4N), onde possui duas funções F (F_0 e F_1), que realizam as operações de substituição e multiplicação sobre um corpo finito GF(2^8) (*Galois Field*), conforme já ilustrado na Figura 2.8, subseção 2.6.

Nesta subseção serão detalhadas todas as partes (módulos) que compõe a arquitetura do CLEFIA implementada.

5.3.1 Módulo *S0*

Este módulo foi desenvolvido com o objetivo de realizar parte das operações de substituição do algoritmo, e está inserido dentro dos módulos das funções F_0 e F_1.

As operações de substituição realizadas pelo módulo *S0* ocorrem por byte (8 bits para 8 bits), seguindo o padrão de uma tabela de substituição, conforme Figura 5.11 (descrita em hexadecimal).

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	57	49	d1	c6	2f	33	74	fb	95	6d	82	ea	0e	b0	a8	1c
1.	28	d0	4b	92	5c	ee	85	b1	c4	0a	76	3d	63	f9	17	af
2.	bf	a1	19	65	f7	7a	32	20	06	ce	e4	83	9d	5b	4c	d8
3.	42	5d	2e	e8	d4	9b	0f	13	3c	89	67	c0	71	aa	b6	f5
4.	a4	be	fd	8c	12	00	97	da	78	e1	cf	6b	39	43	55	26
5.	30	98	cc	dd	eb	54	b3	8f	4e	16	fa	22	a5	77	09	61
6.	d6	2a	53	37	45	c1	6c	ae	ef	70	08	99	8b	1d	f2	b4
7.	e9	c7	9f	4a	31	25	fe	7c	d3	a2	bd	56	14	88	60	0b
8.	cd	e2	34	50	9e	dc	11	05	2b	b7	a9	48	ff	66	8a	73
9.	03	75	86	f1	6a	a7	40	c2	b9	2c	db	1f	58	94	3e	ed
a.	fc	1b	a0	04	b8	8d	e6	59	62	93	35	7e	ca	21	df	47
b.	15	f3	ba	7f	a6	69	c8	4d	87	3b	9c	01	e0	de	24	52
c.	7b	0c	68	1e	80	b2	5a	e7	ad	d5	23	f4	46	3f	91	c9
d.	6e	84	72	bb	0d	18	d9	96	f0	5f	41	ac	27	c5	e3	3a
e.	81	6f	07	a3	79	f6	2d	38	1a	44	5e	b5	d2	ec	cb	90
f.	9a	36	e5	29	c3	4f	ab	64	51	f8	10	d7	bc	02	7d	8e

Figura 5.11. Tabela de substituição *S0* – CLEFIA (Sony, 2010)

O módulo *S0* descrito em VHDL fez uso das atribuições *WITH*, *SELECT*, *WHEN*, para a sua construção, utilizando os dados em binário. O Quadro 5.17 exibe parte do algoritmo em VHDL deste módulo (linha 1 da tabela de substituição *S0*).

```
42  begin
43
44      with s0_in  (7 downto 0) select
45           s0_out (7 downto 0) <=
46
47           "01010111"    when    "00000000", --(X"57")   0
48           "01001001"    when    "00000001", --(X"49")   1
49           "11010001"    when    "00000010", --(X"d1")   2
50           "11000110"    when    "00000011", --(X"c6")   3
51           "00101111"    when    "00000100", --(X"2f")   4
52           "00110011"    when    "00000101", --(X"33")   5
53           "01110100"    when    "00000110", --(X"74")   6
54           "11111011"    when    "00000111", --(X"fb")   7
55           "10010101"    when    "00001000", --(X"95")   8
56           "01101101"    when    "00001001", --(X"6d")   9
57           "10000010"    when    "00001010", --(X"82")   10
58           "11101010"    when    "00001011", --(X"ea")   11
59           "00001110"    when    "00001100", --(X"0e")   12
60           "10110000"    when    "00001101", --(X"b0")   13
61           "10101000"    when    "00001110", --(X"a8")   14
62           "00011100"    when    "00001111", --(X"1c")   15
```

Quadro 5.17. Parte do algoritmo em VHDL do módulo *S0* – CLEFIA

Ao todo, quatro módulos *S0* são instanciados no algoritmo, distribuído nas funções F, sendo dois na função F_0 e dois na F_1, o que equivale a substituição de 8 bytes de dados por rodada, o restante dos dados também são submetidos a um processo de substituição, porém pelo módulo *S1*, descrito a seguir.

5.3.2 Módulo *S1*

A função do módulo *S1* é a mesma do módulo *S0*, realizar operações de substituição de dados, porém, este módulo utiliza como referência outra tabela de substituição, exibida na Figura 5.12 (em formato hexadecimal).

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	6c	da	c3	e9	4e	9d	0a	3d	b8	36	b4	38	13	34	0c	d9
1.	bf	74	94	8f	b7	9c	e5	dc	9e	07	49	4f	98	2c	b0	93
2.	12	eb	cd	b3	92	e7	41	60	e3	21	27	3b	e6	19	d2	0e
3.	91	11	c7	3f	2a	8e	a1	bc	2b	c8	c5	0f	5b	f3	87	8b
4.	fb	f5	de	20	c6	a7	84	ce	d8	65	51	c9	a4	ef	43	53
5.	25	5d	9b	31	e8	3e	0d	d7	80	ff	69	8a	ba	0b	73	5c
6.	6e	54	15	62	f6	35	30	52	a3	16	d3	28	32	fa	aa	5e
7.	cf	ea	ed	78	33	58	09	7b	63	c0	c1	46	1e	df	a9	99
8.	55	04	c4	86	39	77	82	ec	40	18	90	97	59	dd	83	1f
9.	9a	37	06	24	64	7c	a5	56	48	08	85	d0	61	26	ca	6f
a.	7e	6a	b6	71	a0	70	05	d1	45	8c	23	1c	f0	ee	89	ad
b.	7a	4b	c2	2f	db	5a	4d	76	67	17	2d	f4	cb	b1	4a	a8
c.	b5	22	47	3a	d5	10	4c	72	cc	00	f9	e0	fd	e2	fe	ae
d.	f8	5f	ab	f1	1b	42	81	d6	be	44	29	a6	57	b9	af	f2
e.	d4	75	66	bb	68	9f	50	02	01	3c	7f	8d	1a	88	bd	ac
f.	f7	e4	79	96	a2	fc	6d	b2	6b	03	e1	2e	7d	14	95	1d

Figura 5.12. Tabela de substituição *S1* – CLEFIA (Sony, 2010)

A descrição em VHDL do módulo *S1* é semelhando ao do módulo *S0*, porém, obedecendo aos dados da tabela de substituição *S1*. Parte do algoritmo do módulo S1 é exibido no Quadro 5.18.

```
42  begin
43
44      with s1_in  (7 downto 0) select
45           s1_out (7 downto 0) <=
46
47           "01101100"    when    "00000000", --(X"6c")   0
48           "11011010"    when    "00000001", --(X"da")   1
49           "11000011"    when    "00000010", --(X"c3")   2
50           "11101001"    when    "00000011", --(X"e9")   3
51           "01001110"    when    "00000100", --(X"4e")   4
52           "10011101"    when    "00000101", --(X"9d")   5
53           "00001010"    when    "00000110", --(X"0a")   6
54           "00111101"    when    "00000111", --(X"3d")   7
55           "10111000"    when    "00001000", --(X"b8")   8
56           "00110110"    when    "00001001", --(X"36")   9
57           "10110100"    when    "00001010", --(X"b4")   10
58           "00111000"    when    "00001011", --(X"38")   11
59           "00010011"    when    "00001100", --(X"13")   12
60           "00110100"    when    "00001101", --(X"34")   13
61           "00001100"    when    "00001110", --(X"0c")   14
62           "11011001"    when    "00001111", --(X"d9")   15
```

Quadro 5.18. Parte do algoritmo do módulo S1 - CLEFIA

Um total de quatro módulos *S1* são instanciados no algoritmo, sendo dois na função F_0 e dois na F_1, resultando em 8 bytes de dados que passam por processo de substituição, atendidos por *S1*.

5.3.3 Módulo *M0*

Este módulo a realização de operações de multiplicação matricial realizados sobre GF(2^8) (*Galois Field*), que é definido por um polinômio primitivo $z^8 + z^4 + z^3 + z^2 + 1$ visando garantir um nível adequado de segurança e resistância a ataques.

O módulo *M0* tem como entrada os dados oriundos de dois módulos *S0* e dois módulos *S1*, totalizando 4 bytes de dados, que são organizados em forma de uma matriz e submetidos a uma multiplicação sobre GF (2^8), por uma matriz específica, exibida na Figura 5.13 (em hexadecimal).

$$M_0 = \begin{pmatrix} 0x01 & 0x02 & 0x04 & 0x06 \\ 0x02 & 0x01 & 0x06 & 0x04 \\ 0x04 & 0x06 & 0x01 & 0x02 \\ 0x06 & 0x04 & 0x02 & 0x01 \end{pmatrix}$$

Figura 5.13. Matriz de difusão *M0* do CLEFIA (Sony, 2010)

O algoritmo em VHDL do módulo *M0* implementado neste trabalho, utiliza uma versão onde a matriz M0 (Figura 26), é decomposta em três matrices. Sendo o resultado, a soma da multiplicação em GF (2^8), da matriz de dados com cada matriz específica, onde X_0, X_1, X_2, X_3, representam as entradas dos dados organizados na forma de matriz, Y_0, Y_1, Y_2, Y_3, representam a matriz resultante do processo de multiplicação, conforme exemplo ilustrado na Figura 5.14.

$$
\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 01 & 02 & 04 & 06 \\ 02 & 01 & 06 & 04 \\ 04 & 06 & 01 & 02 \\ 06 & 04 & 02 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

$$
= \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 00 & 02 & 00 & 02 \\ 02 & 00 & 02 & 00 \\ 00 & 02 & 00 & 02 \\ 02 & 00 & 02 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

$$
+ \begin{pmatrix} 00 & 00 & 04 & 04 \\ 00 & 00 & 04 & 04 \\ 04 & 04 & 00 & 00 \\ 04 & 04 & 00 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

Figura 5.14. Exemplo multiplicação matricial alternativa de *M0* – CLEFIA (Sony, 2010)

A multiplicação matricial exibida na Figura 5.14, resulta, após simplificação, no exibido na Figura 5.15, sendo que A_0, A_1, B_0, B_1, C_0, C_1, D_0, D_1, são variáveis que auxiliam no processo de multiplicação, sendo este o modelo utilizado para a descrição em VHDL do módulo *M0*.

$$
\begin{cases} A_0 = X_0 \oplus X_1 \\ A_1 = X_2 \oplus X_3 \end{cases}
\begin{cases} B_0 = X_0 \oplus X_2 \\ B_1 = X_1 \oplus X_3 \end{cases}
\begin{cases} C_0 = \{02\} \times B_0 \\ C_1 = \{02\} \times B_1 \end{cases}
\begin{cases} D_0 = \{04\} \times A_0 \\ D_1 = \{04\} \times A_1 \end{cases}
$$

$$
\begin{cases} Y_0 = C_1 \oplus D_1 \oplus X_0 \\ Y_1 = C_0 \oplus D_1 \oplus X_1 \\ Y_2 = C_1 \oplus D_0 \oplus X_2 \\ Y_3 = C_0 \oplus D_0 \oplus X_3 \end{cases}
$$

Figura 5.15. Modelo de cálculo para multiplicação por M0 – CLEFIA (Sony, 2010)

Na descrição em VHDL do módulo *M0*, duas funções (F2 e F4) que representam a multiplicação por 0x02 e 0x04 foram construídas. Parte da descrição do módulo *M0* pode ser visualizada no Quadro 5.19.

```
87      --entradas
88      X0 <= m0_in (31 downto 24);
89      X1 <= m0_in (23 downto 16);
90      X2 <= m0_in (15 downto 8);
91      X3 <= m0_in (7 downto 0);
92
93      --A0 = X0 + X1, A1 = X2 + X3
94      A0 <= X0 xor X1;
95      A1 <= X2 xor X3;
96
97      --B0 = X0 + X2, B1 = X1 + X3
98      B0 <= X0 xor X2;
99      B1 <= X1 xor X3;
100
101     --C0 = {02} x B0, C1 = {02} x B1
102     C0 <= F2 (B0);
103     C1 <= F2 (B1);
104
105     --D0 = {04} x A0, D1 = {04} x A1
106     D0 <= F4 (A0);
107     D1 <= F4 (A1);
108
109     --Y0 = C1 + D1 + X0
110     Y0 <= C1 xor D1 xor X0;
111     --Y1 = C0 + D1 + X1
112     Y1 <= C0 xor D1 xor X1;
113     --Y2 = C1 + D0 + X2
114     Y2 <= C1 xor D0 xor X2;
115     --Y3 = C0 + D0 + X3
116     Y3 <= C0 xor D0 xor X3;
117
118     m0_out <= Y0 & Y1 & Y2 & Y3;
```

Quadro 5.19. Parte do algoritmo em VHDL do módulo *M0*

5.3.4 Módulo *M1*

Este módulo possui a mesma função do módulo *M0*, a realização de cálculos de multiplicação matricial em GF(2^8), porém utilizando outra matriz de difusão, exibida na Figura 5.16.

$$M_1 = \begin{pmatrix} 0x01 & 0x08 & 0x02 & 0x0a \\ 0x08 & 0x01 & 0x0a & 0x02 \\ 0x02 & 0x0a & 0x01 & 0x08 \\ 0x0a & 0x02 & 0x08 & 0x01 \end{pmatrix}$$

Figura 5.16. Matriz de difusão M1 do CLEFIA (Sony, 2010)

Semelhante a matriz de difusão M0, a decomposição da matriz M1 para o processo de multiplicação também é possível, sendo exibida na Figura 29, onde X_0, X_1, X_2, X_3, representam uma matriz com os dados entrada e Z_0, Z_1, Z_2, Z_3, representam a matriz resultante do processo de multiplicação.

A resultante da multiplicação por M1, exibida na Figura 5.17, após simplificação, resulta no modelo exibido na Figura 5.18, sendo que A_0, A_1, B_0, B_1, C_0, C_1, D_0, D_1, são variáveis que auxiliam no processo de multiplicação.

$$
\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \begin{pmatrix} 01 & 08 & 02 & 0A \\ 08 & 01 & 0A & 02 \\ 02 & 0A & 01 & 08 \\ 0A & 02 & 08 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

$$
= \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 00 & 00 & 02 & 02 \\ 00 & 00 & 02 & 02 \\ 02 & 02 & 00 & 00 \\ 02 & 02 & 00 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

$$
+ \begin{pmatrix} 00 & 08 & 00 & 08 \\ 08 & 00 & 08 & 00 \\ 00 & 08 & 00 & 08 \\ 08 & 00 & 08 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}
$$

Figura 5.17. Exemplo para a multiplicação matricial alternativa de M1 - CLEFIA (Sony, 2010)

$$
\begin{cases} A_0 = X_0 \oplus X_1 \\ A_1 = X_2 \oplus X_3 \\ B_0 = X_0 \oplus X_2 \\ B_1 = X_1 \oplus X_3 \end{cases}
\quad
\begin{cases} C_0 = \{02\} \times A_0 \\ C_1 = \{02\} \times A_1 \\ D_0 = \{08\} \times B_0 \\ D_1 = \{08\} \times B_1 \end{cases}
\quad
\begin{cases} Z_0 = C_1 \oplus D_1 \oplus X_0 \\ Z_1 = C_1 \oplus D_0 \oplus X_1 \\ Z_2 = C_0 \oplus D_1 \oplus X_2 \\ Z_3 = C_0 \oplus D_0 \oplus X_3 \end{cases}
$$

Figura 5.18. Figura 28. Modelo de cálculo para multiplicação por M1 – CLEFIA (Sony, 2010)

Duas funções (F2 e F8) que representam a multiplicação por 0x02 e 0x08 foram descritas no algoritmo em VHDL do módulo *M0*. Parte deste algoritmo pode ser visualizada no Quadro 5.20.

```
88      --entradas
89      X0 <= m1_in (31 downto 24);
90      X1 <= m1_in (23 downto 16);
91      X2 <= m1_in (15 downto 8);
92      X3 <= m1_in (7 downto 0);
93
94      --A0 = X0 + X1, A1 = X2 + X3
95      A0 <= X0 xor X1;
96      A1 <= X2 xor X3;
97
98      --B0 = X0 + X2, B1 = X1 + X3
99      B0 <= X0 xor X2;
100     B1 <= X1 xor X3;
101
102     --C0 = {02} x A0, C1 = {02} x A1
103     C0 <= F2 (A0);
104     C1 <= F2 (A1);
105
106     --D0 = {08} x B0, D1 = {08} x B1
107     D0 <= F8 (B0);
108     D1 <= F8 (B1);
109
110     --Y0 = C1 + D1 + X0
111     Y0 <= C1 xor D1 xor X0;
112     --Y1 = C1 + D0 + X1
113     Y1 <= C1 xor D0 xor X1;
114     --Y2 = C0 + D1 + X2
115     Y2 <= C0 xor D1 xor X2;
116     --Y3 = C0 + D0 + X3
117     Y3 <= C0 xor D0 xor X3;
118
119     m1_out <= Y0 & Y1 & Y2 & Y3;
```

Quadro 5.20. Parte do algoritmo em VHDL do módulo *M1*

5.3.5 Módulo *F0*

O módulo *F0* é composto pela interconexão dos módulos *S0*, *S1* e *M0*. Possui entrada de 8 bytes, sendo 4 bytes de texto (x_0, x_1, x_2, x_3), que são submetidos a operações XOR com 4 bytes de chave (k_0, k_1, k_2, k_3), sendo que o resultado desta operação serve de entrada para quatro módulos de substituição (dois *S0* e dois *S1*), sendo as saídas dos módulos de substituição conectadas ao módulo *M0*, gerando por fim 4 bytes de saída do módulo *F0* (y_0, y_1, y_2, y_3).

A Figura 5.19 ilustra a estrutura do módulo *F0*.

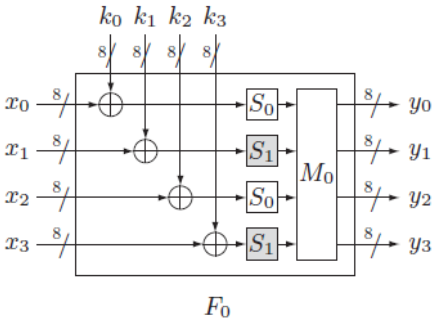


Figura 5.19. Estrutura da Função *F0* do CLEFIA (Sony, 2010)

O módulo *F0* descrito em VHDL possui 64 bits de entrada (32 de texto e 32 de chave) e 32 bits de saída. As entradas são subdivididas em 8 bytes (sendo 4 bytes de texto e 4 bytes de chave) a fim de proceder os cálculos específicos da função *F0*. São descritos ainda o instanciamento de dois módulos *S0* e dois módulos *S1*, e um módulo *M0*, interconectados seguindo o padrão da função *F0*, conforme Figura 31.

Parte do algoritmo do módulo *F0* é exibido no Quadro 5.21.

```
81      --Add round key
82      sb0_in <= (x0 xor k0) & (x2 xor k2);
83      sb1_in <= (x1 xor k1) & (x3 xor k3);
84
85      -- 2 instancias de S0 e conexões
86      s0_Boxs : for N in 1 downto 0 generate
87          s0_x: S0 port map(
88              s0_in => sb0_in(8*N+7 downto 8*N),
89              s0_out => sb0_out (8*N+7 downto 8*N)
90              );
91          end generate s0_Boxs;
92
93      -- 2 instancias de S1 e conexões
94      s1_Boxs : for N in 1 downto 0 generate
95          s1_x: S1 port map(
96              s1_in => sb1_in(8*N+7 downto 8*N),
97              s1_out => sb1_out (8*N+7 downto 8*N)
98              );
99          end generate s1_Boxs;
100
101     --entrada de M0
102     sig_m0_in <= sb0_out (15 downto 8) &
103                  sb1_out (15 downto 8) &
104                  sb0_out (7 downto 0)  &
105                  sb1_out (7 downto 0);
106
107     -- conexões de M0
108     m_0 : M0 port map(
109             m0_in => sig_m0_in,
110             m0_out => f0_out
111         );
```

Quadro 5.21. Parte do algoritmo em VHDL do módulo *F0*

5.3.6 Módulo *F1*

O módulo *F1* possui uma estrutura semelhante ao módulo *F0*, com entrada de 8 bytes, sendo 4 bytes de texto (x_0, x_1, x_2, x_3), que são submetidos a operações XOR com 4 bytes de chave (k_0, k_1, k_2, k_3), e as saídas conectadas as entradas dos módulos S (dois *S0* e dois *S1*). A principal diferente de *F1* para *F0* está na forma de distrituição dos módulos *S* (dois *S0* e dois *S1*), e na utilização do *M1*, em lugar ao módulo *M0*.

A Figura 5.20 ilustra a estrutura do módulo *F1*.

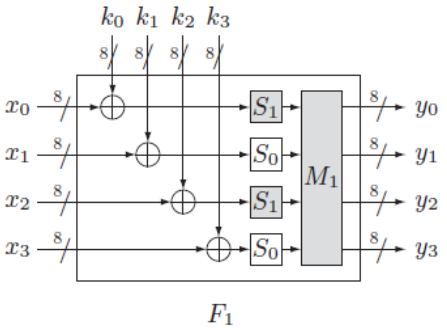


Figura 5.20. Estrutura da Função *F1* do CLEFIA (Sony, 2010)

Parte da descrição em VHDL do módulo F1 pode ser visualizada no Quadro 5.22.

```
81      --Add round key
82      sb0_in <= (x1 xor k1) & (x3 xor k3);
83      sb1_in <= (x0 xor k0) & (x2 xor k2);
84
85      -- 2 instancias de S0 e conexões
86      s0_Boxs : for N in 1 downto 0 generate
87          s0_x: S0 port map(
88              s0_in => sb0_in(8*N+7 downto 8*N),
89              s0_out => sb0_out (8*N+7 downto 8*N)
90              );
91          end generate s0_Boxs;
92
93      -- 2 instancias de S1 e conexões
94      s1_Boxs : for N in 1 downto 0 generate
95          s1_x: S1 port map(
96              s1_in => sb1_in(8*N+7 downto 8*N),
97              s1_out => sb1_out (8*N+7 downto 8*N)
98              );
99          end generate s1_Boxs;
100
101         --entrada de M1
102         sig_m1_in <= sb1_out (15 downto 8) &
103                      sb0_out (15 downto 8) &
104                      sb1_out  (7 downto 0) &
105                      sb0_out (7 downto 0);
106
107     m_1 : M1 port map(
108             m1_in => sig_m1_in,
109             m1_out => f1_out
110         );
```

Quadro 5.22. Parte do algoritmo em VHDL do módulo *F1*

5.3.7 Módulo GF4N

Este módulo foi desenvolvido com o objetivo de facilitar a implementação do CLEFIA em FPGA. Contempla a denominada Rede de Feistel Generalizada de quadro ramificações (GF4N), na qual é composta pelos módulos *F0* e *F1* e mais alguns módulos e circuitos, visando o atendimento das especificidades do algoritmo, sendo estes:

- **Módulo WK**: realiza as operações de clareamento de chave (WK), na qual duas palavras da chave original, são adicionadas através de uma operação XOR com duas palavras do texto, nas rodadas inicial e final do processo de encriptação;

- **Módulo SHIFT:** realiza as operações de permutação das palavras (deslocamento a esquerda);

- **Mux (1)**: responsável pela seleção dos dados oriundos da entrada de GF4N ou do módulo WK que serão submetidos aos módulos *F0*, *F1* e SHIFT.

- **Mux (2)**: seleciona os dados que são ser submetidos ao módulo WK, sendo que na rodada inicial de encriptação, os dados da entrada de GF4N são selecionados, e na última rodada de encriptação, a WK é realizada com os dados oriundo da saída das funções *F0* e *F1*;

- **Mux (3)**: visa atender a seleção de dados da última rodada de encriptação, em que não são submetidos ao módulo SHIFT;

- **Mux (4)**: responsável pela seleção dos dados de saída, em que na última rodada de encriptação os dados selecionados são derivados do módulo WK;

A Figura 5.21 exibe o comportamento da Rede de Feistel Generalizada de quatro ramificações, na qual o módulo GF4N foi desenvolvido, onde (P_0, P_1, P_2, P_3), representam as palavras do texto plano, (C_0, C_1, C_2, C_3), representam o texto encriptado, *RK* representa as chaves das rodadas, *WK* representa as chaves de clareamento (partes da chave original).

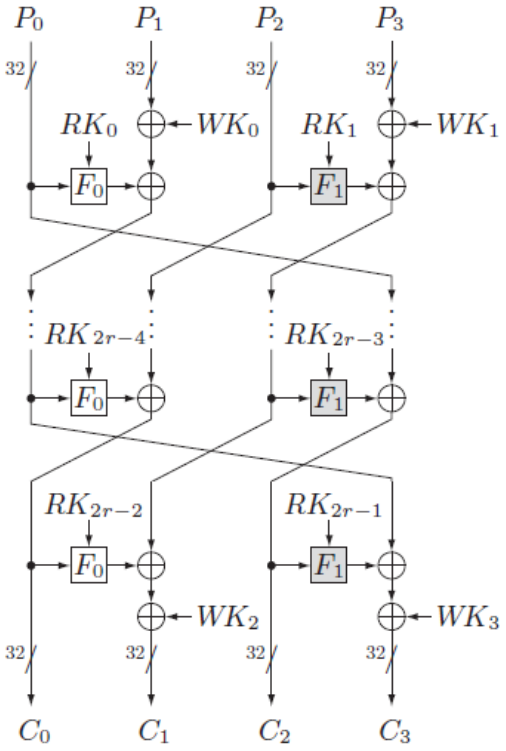


Figura 5.21. Comportamento da GF4N do CLEFIA (adaptado de Sony, 2010)

O módulo GF4N descrito em VHDL é formado pelas instancias dos módulos *F0*, *F1*, *WK*, *SHIFT* e circuitos multiplexadores, interconectados com o objetivo de executar as especificidades de cada rodada da rede que forma o algoritmo CLEFIA.

A Figura 5.22 exibe a arquitetura do módulo GF4N.

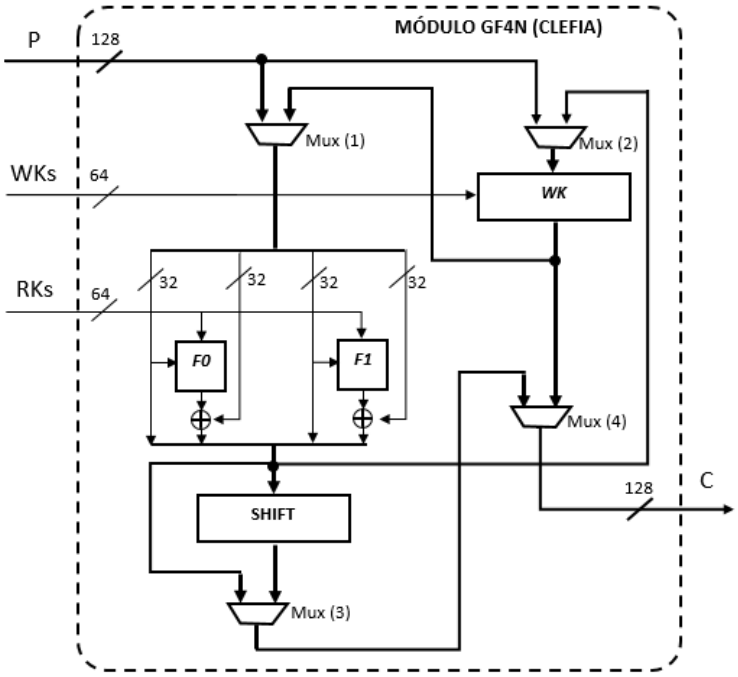


Figura 5.22. Arquitetura do módulo GF4N

O Quadro 5.23 exibe parte do algoritmo em VHDL do módulo GF4N deste trabalho.

```
112         -- mux de GF4N
113         WITH sel_mux_gf4n SELECT
114             sig_gf4n_in <= P WHEN '0',
115                            sig_wk_out WHEN OTHERS;
116
117         -- mux do WK
118         WITH sel_mux_wk SELECT
119             sig_wk_in <= P WHEN '1',
120                          sig_gf4n_out WHEN OTHERS;
121
122         -- mux da saída - C
123         WITH sel_mux_out SELECT
124             C <= sig_wk_out WHEN '1',
125                  sig_shift WHEN OTHERS;
126
127         -- mux do Shift
128         WITH sel_mux_shift SELECT
129             sig_shift <= sig_gf4n_out WHEN '1',
130                          sig_shift_out WHEN OTHERS;
131
132     --RKs (Chaves da rodada)
133     rk1 <= round_key (63 downto 32);
134     rk2 <= round_key (31 downto 0);
135
136     --p0,p1,p2,p3 (interno GF4N)
137     p0 <= sig_gf4n_in (127 downto 96);
138     p1 <= sig_gf4n_in (95 downto 64);
139     p2 <= sig_gf4n_in (63 downto 32);
140     p3 <= sig_gf4n_in (31 downto 0);
141
142     -- add F0 XOR "P1"
143     p1_out <= f0_out xor p1;
144     -- add F1 XOR "P3"
145     p3_out <= f1_out xor p3;
146
147     --saída de GF4N
148     sig_gf4n_out <= p0 & p1_out & p2 & p3_out;
```

Quadro 5.23. Parte do algoritmo em VHDL do módulo GF4N

5.3.8 Módulo Key Shedule

Este módulo visa atendar as operações de geração de chaves (RKs), que são utilizadas a cada rodada durante o processo de encriptação. A versão do CLEFIA implementado neste trabalho utiliza chave original com tamanho de 128 bits, sendo que para essa arquitetura, um total de 36 chaves de 32 bits são geradas, duas para cada rodada de encriptação, de um total de 18 rodadas de encriptação.

Para a geração das RKs, primeiramente é necessário uma chave intermediária de 128 bits, denominada L. Esta chave intermediária é gerada a partir da chave original, que é submetida a GF4N durante 12 rodadas (sem clareamento de chave WK), sendo que, para o processo de geração de L, as RKs são constantes predefinidas, que nesta implementação são oriundas de uma memória ROM. Ao final das 12 rodadas, a chave intermediária L é armazenada em um registrador e será utilizada para a geração das RKs do processo de encriptação.

Durante o processo de encriptação, as RKs são geradas a partir de operações XOR e permutações que utilizam a chave intermediária L, a chave original, e constantes predefinidas, de acordo como mostrado na Figura 5.23, onde K representa a chave original, L a chave intermediária, CON representam constantes (parte das constantes foram utilizadas na geração de L), e $\sum$ (sigma) representa uma função de permutação de L. WK, que é utilizada na primeira e última rodada de encriptação é gerada a partir da divisão da chave original (K) de 128 bits em quatro palavras (32 bits cada).

$$
\begin{array}{|c|c|c|c|l|}
\hline
WK_0 & WK_1 & WK_2 & WK_3 & \leftarrow K \\
\hline
RK_0 & RK_1 & RK_2 & RK_3 & \leftarrow L \oplus (CON_{24}^{(128)} | CON_{25}^{(128)} | CON_{26}^{(128)} | CON_{27}^{(128)}) \\
\hline
RK_4 & RK_5 & RK_6 & RK_7 & \leftarrow \Sigma(L) \oplus K \oplus (CON_{28}^{(128)} | CON_{29}^{(128)} | CON_{30}^{(128)} | CON_{31}^{(128)}) \\
\hline
RK_8 & RK_9 & RK_{10} & RK_{11} & \leftarrow \Sigma^2(L) \oplus (CON_{32}^{(128)} | CON_{33}^{(128)} | CON_{34}^{(128)} | CON_{35}^{(128)}) \\
\hline
RK_{12} & RK_{13} & RK_{14} & RK_{15} & \leftarrow \Sigma^3(L) \oplus K \oplus (CON_{36}^{(128)} | CON_{37}^{(128)} | CON_{38}^{(128)} | CON_{39}^{(128)}) \\
\hline
RK_{16} & RK_{17} & RK_{18} & RK_{19} & \leftarrow \Sigma^4(L) \oplus (CON_{40}^{(128)} | CON_{41}^{(128)} | CON_{42}^{(128)} | CON_{43}^{(128)}) \\
\hline
RK_{20} & RK_{21} & RK_{22} & RK_{23} & \leftarrow \Sigma^5(L) \oplus K \oplus (CON_{44}^{(128)} | CON_{45}^{(128)} | CON_{46}^{(128)} | CON_{47}^{(128)}) \\
\hline
RK_{24} & RK_{25} & RK_{26} & RK_{27} & \leftarrow \Sigma^6(L) \oplus (CON_{48}^{(128)} | CON_{49}^{(128)} | CON_{50}^{(128)} | CON_{51}^{(128)}) \\
\hline
RK_{28} & RK_{29} & RK_{30} & RK_{31} & \leftarrow \Sigma^7(L) \oplus K \oplus (CON_{52}^{(128)} | CON_{53}^{(128)} | CON_{54}^{(128)} | CON_{55}^{(128)}) \\
\hline
RK_{32} & RK_{33} & RK_{34} & RK_{35} & \leftarrow \Sigma^8(L) \oplus (CON_{56}^{(128)} | CON_{57}^{(128)} | CON_{58}^{(128)} | CON_{59}^{(128)}) \\
\hline
\end{array}
$$

Figura 5.23. Expansão da chave e geração das RKs do CLEFIA-128 (Sony, 2010)

Em síntese o módulo Key Shedule é composto pelos seguintes módulos e circuitos:

- **Módulo Constante**: um módulo de memória que armazena as constantes predefinidas pelos desenvolvedores do CLEFIA, sendo 60 constantes de 32 bits, que são utilizadas na geração de L e das RKs;

- **Módulo Reg_L**: registrador de L (128 bits);

- **Módulo Double_Swap**: este módulo realiza operações de permutação com os dados da saída do registrador L;

- **Módulo T**: este módulo realiza as operações XOR a cada rodada de encriptação com parte da chave intermediária L (64 bits), parte da chave original K (64 bits) alternadamente, e duas constantes (64 bits), conforme ilustrado Figura 36;

- **Mux (1)**: responsável pela seleção da entrada do registrador L, sendo que durante o processo de geração da chave intermediária L (12 rodadas), a seleção é proveniente da saída do módulo GF4N (GF4N_out), ou a entrada selecionada é oriunda da saída do módulo de permutação Double_Swap, durante o processo de geração das RKs;

- **Mux (2)**: seleciona a saída das RKs (chaves da rodada), que durante a geração de L é proveniente do módulo Constante, e durante a geração das RKs é proveniente do módulo T;

A Figura 5.24 ilustra a arquitetura do módulo Key Shedule.

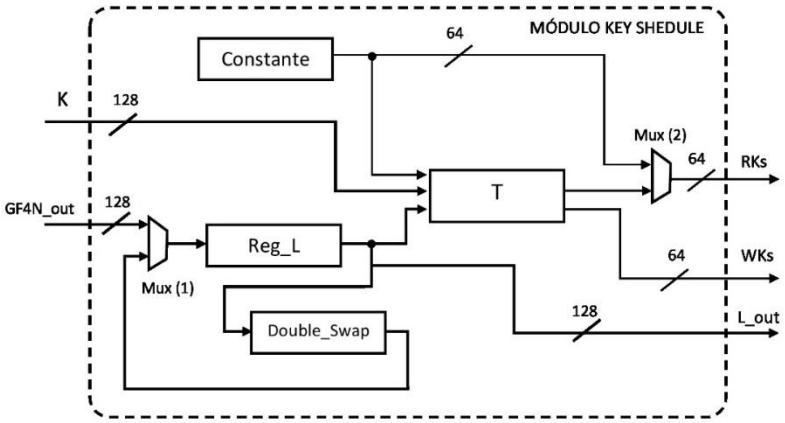


Figura 5.24. Arquitetura do Módulo Key Shedule do CLEFIA

Parte do algoritmo em VHDL do módulo Key Shedule é exibido no Quadro 5.24.

```
99      constant_K : const port map (
100             clk     => clk,
101             reset   => reset,
102             enable  => en_constant,
103             const_l => sig_const_out
104             );
105
106     reg_L : reg_kL  port map (
107             clk    => clk,
108             reset  => reset,
109             enable => en_regL,
110             kL_in  => sig_L_in,
111             kL_out => sig_L_out
112             );
113
114     D_W : double_swap port map (
115             ds_in  => sig_L_out,
116             ds_out => sig_DS_out
117             );
118
119     T_RK : T_key port map (
120             sel_KLC => sel_T,
121             sel_KL => sel_muxKL,
122             K_in   => K_in,
123             L_in   => sig_L_out,
124             C_in   => sig_const_out,
125             WK     => WK,
126             RK     => sig_T_out
127             );
128
129
130     -- mux RK (até rodada 12 RK <= constante, até rodada 30 RK <= T);
131     WITH sel_muxRK SELECT
132          RK <= sig_const_out WHEN '0',
133                sig_T_out WHEN OTHERS;
134
```

Quadro 5.24. Parte do algoritmo em VHDL do módulo Key Shedule

5.3.9 Módulo de Controle

O gerenciamento dos módulos e multiplexadores visando a habilitação/seleção correta dos dados a cada rodada, seja de geração de chave e/ou de encriptação é realizado por este módulo de Controle.

Este módulo possui ao todo 5 estados, sendo eles:

- **P**: Parado ou Estático;
- **C**: Carrega os registradores com o Texto Plano (128 bits) e com a Chave (128 bits);
- **KL**: Neste estado é gerada a chave intermediária L, que é utilizada no processo de geração de chaves para as rodadas, sendo que 12 rodadas (iterações), são executadas neste estado (contador 1 até 12);
- **E**: Executa as rodadas de encriptação necessárias. Nesta implementação são executadas 18 rodadas de encriptação (contador 13 até 30);
- **L**: Sinaliza para a leitura do texto cifrado após o processo de encriptação;

A Figura 5.25 ilustra o diagrama de estados deste módulo de controle.

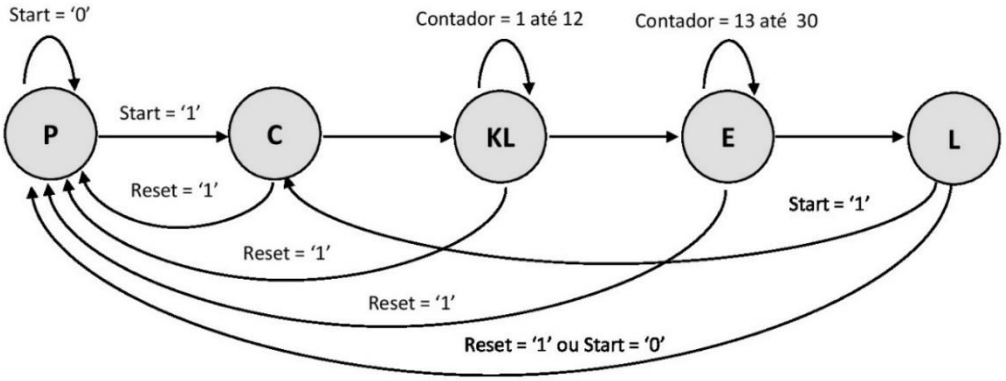


Figura 5.25. Diagrama de estados do Módulo de Controle do CLEFIA

A principal diferença dos módulos de controle dos algoritmos AES e PRESENT, sendo de 4 estados cada, para 5 estados do módulo de controle do CLEFIA é devida ao fato do processo de geração de chaves das rodadas, necessária para as rodadas de encriptação, depender primeiramente da chave intermediária L, sendo, portanto, um processo que não pode ocorrer simultaneamente com a encriptação.

Na descrição VHDL do módulo de controle do CLEFIA, visando atender as especificidades do algoritmo, que necessita de uma quantidade significativa de variáveis de controle para os multiplexadores e registradores, uma memória com essas variáveis para cada rodada foi elaborada a parte e instanciada no módulo de controle, sem prejuízo quanto ao consumo de recursos do FPGA.

Parte do algoritmo em VHDL do módulo de Controle pode ser observado no Quadro 5.25.

```
66              when SM_LOAD =>
67                  en_count <= '1';
68                  en_const <= '1';
69                  ready <=    '0';
70                  next_state <= SM_L_KEY;
71
72              when SM_L_KEY =>
73                  en_count <= '1';
74                  en_const <= '1';
75                  ready <=    '0';
76                  if (round = "01100") then -- 12 rodadas para geração de L_key
77                  next_state <= SM_ENC;
78                  else
79                  next_state <= SM_L_KEY;
80                  end if;
81
82              when SM_ENC =>
83                  en_count <= '1';
84                  en_const <= '1';
85                  ready <=    '0';
86                  if (round = "11110") then -- 30 (30 - 12 = 18) 18 rodadas de encriptação
87                  next_state <= SM_READY;
88                  else
89                  next_state <= SM_ENC;
90                  end if;
```

Quadro 5.25. Parte do algoritmo em VHDL do Módulo de Controle - CLEFIA

5.3.10 Arquitetura CLEFIA Encriptação

A interconexão dos módulos mencionados nos subtópicos anteriores, juntamente com módulos e circuitos complementares, formam a arquitetura geral do *hardware* de encriptação do algoritmo CLEFIA, que nesta implementação utiliza chave de 128 bits e é denominado de CLEFIA Encriptação. Os módulos complementares são descritos a seguir:

- **Módulo Registrador do Texto (Reg_Texto)**: registra o Texto Plano e o Texto Encriptado a cada rodada do algoritmo (128 bits);
- **Módulo Registrador da Chave (Reg_Chave)**: registra a Chave original e as chaves geradas a cada rodada do algoritmo (128 bits);
- **Contador**: contador sequencial de 5 bits para controle do processo de geração da chave intermediária e do processo de encriptação;
- **Mux 2:1 (1)**: responsável pela seleção do Texto Plano ou do texto encriptado a cada rodada de encriptação;
- **Mux 3:1 (2)**: responsável pela seleção da Chave original ou dos dados oriundos do módulo Key Shedule, que são utilizados no processo de geração da chave intermediária L; ou ainda pela selação do Texto duranto o processo de encriptação;

A arquitetura geral do algoritmo de encriptação do CLEFIA-128 pode ser visualizada na Figura 5.26.

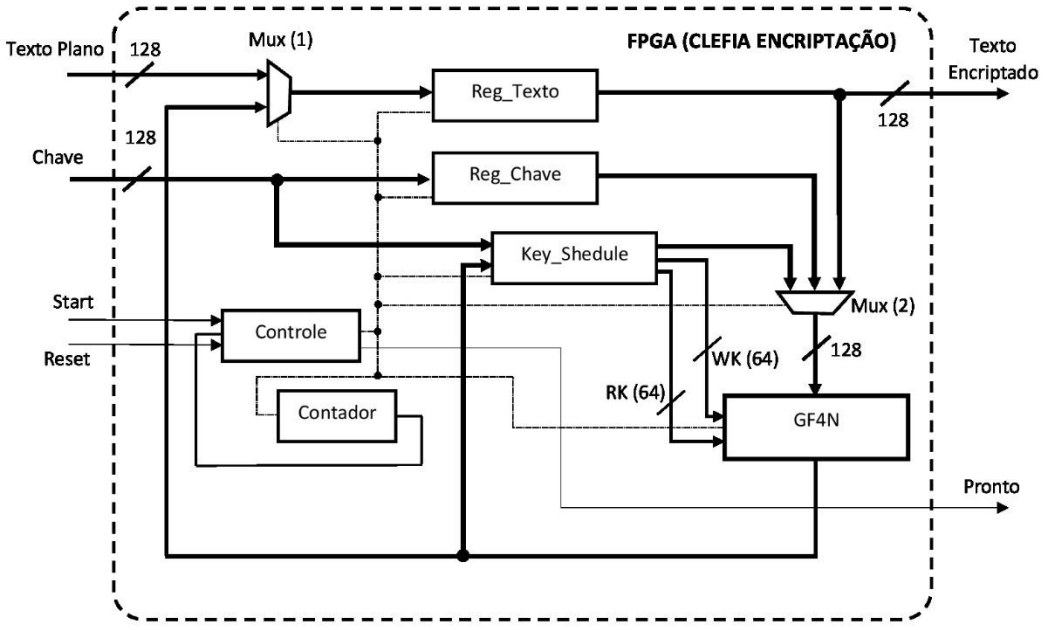


Figura 5.26. Arquitetura geral do algoritmo CLEFIA-128 encriptação

A descrição em VHDL do algoritmo CLEFIA-128 foi elaborada através do instanciamento dos módulos já mencionados, devidamente interconectados, de maneira a formar um módulo TOP denominado de CLEFIA_ENC.

Parte do algoritmo do módulo CLEFIA_ENC pode ser visualizado no Quadro 5.26.

```
223         reg_K : reg_key port map (
224             clk     => clk,
225             reset   => reset,
226             enable  => sig_en_K,
227             key_in  => key,
228             key_out => sig_K
229             );
230 
231         mux : mux4_1 port map (
232             input0 => sig_K,
233             input1 => sig_L,
234             input2 => sig_text_out,
235             ctrl   => sig_sel_P,
236             output => sig_P
237             );
238 
239         -- mux do registrador de texto
240         WITH sig_en_K SELECT
241             sig_text_in <= sig_C WHEN '0',
242                            plaintext WHEN OTHERS;
243 
244 
245     ciphertext <= sig_text_out; -- Texto Encriptado
246     round_count <= sig_round;   -- contador de rodadas
```

Quadro 5.26. Parte do algoritmo do módulo CLEFIA_ENC

5.3.11 Testbench CLEFIA

Semelhante as implementações dos algoritmos AES e PRESENT deste trabalho, simulações e testes também foram realizadas para o algoritmo CLEFIA, com o objetivo de verificar a correta funcionalidade da versão implementada, seguindo as especificações disponibilizadas pela empresa desenvolvedora da cifra, Sony Corporation (2010).

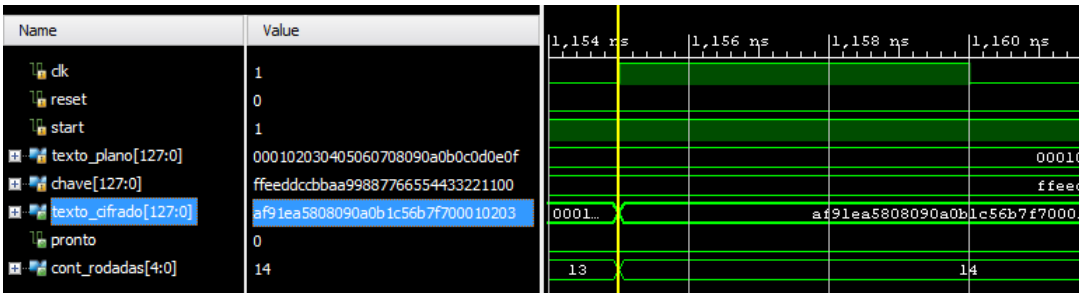
Os vetores de testes durante o processo de encriptação são detalhados no Quadro 5.27.

Texto Plano		**00010203**	**04050607**	**08090a0b**	**0c0d0e0f**
Chave Original		**ffeeddcc**	**bbaa9988**	**77665544**	**33221100**
Chave Intermediária L (12 rodadas para geração)		8f89a61b	9db9d0f3	93e65627	da0d027e
Rodadas de Encriptação					
1	depois do WK	00010203	fbebdbcb	08090a0b	b7a79787
		af91ea58	08090a0b	1c56b7f7	00010203
2		fd15e1b8	1c56b7f7	82dee144	af91ea58
3		c4896f29	82dee144	4ecf4244	fd15e1b8
4		376c6fd2	4ecf4244	4b49b022	c4896f29
5		673fc8b9	4b49b022	7a88be0e	376c6fd2
6		12d017bc	7a88be0e	3345dcfb	673fc8b9
7		1459a507	3345dcfb	b8ec058b	12d017bc
8		bfd8dde7	b8ec058b	81b85950	1459a507
9		5e3a5595	81b85950	79019cb3	bfd8dde7
10		bba357c7	79019cb3	066ca42f	5e3a5595
11		5196dce1	066ca42f	145d5524	bba357c7
12		f9d97f1d	145d5524	2bde6fe7	5196dce1
13		1e29190c	2bde6fe7	4da8e442	f9d97f1d
14		817ca7e4	4da8e442	3de82490	1e29190c
15		367c28af	3de82490	f4ebe9f7	817ca7e4
16		64664dd0	f4ebe9f7	4065c77b	367c28af
17		de2bf2fd	4065c77b	f1298555	64664dd0
18		de2bf2fd	ec12ff89	f1298555	76b685fd
	Depois do WK	de2bf2fd	9b74aacd	f1298555	459494fd
	Texto Cifrado	**de2bf2fd**	**9b74aacd**	**f1298555**	**459494fd**

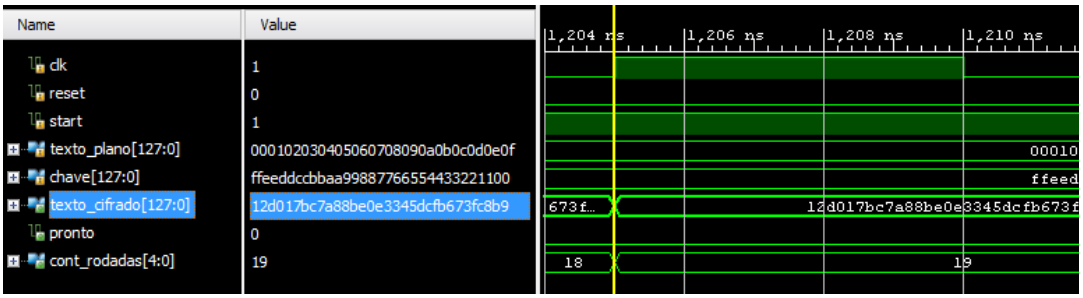
Quadro 5.27. Vetores de teste do algoritmo CLEFIA-128 (Sony, 2010)

Uma simulação foi então realizada com os valores de entrada constantes nos vetores de teste (Texto plano e chave original), sendo que ao final de cada rodada de encriptação, os dados de saída foram devidamente comparados com os dados constantes no Quadro 5.27. Parte desta simulação é exibida no Quadro 5.28.

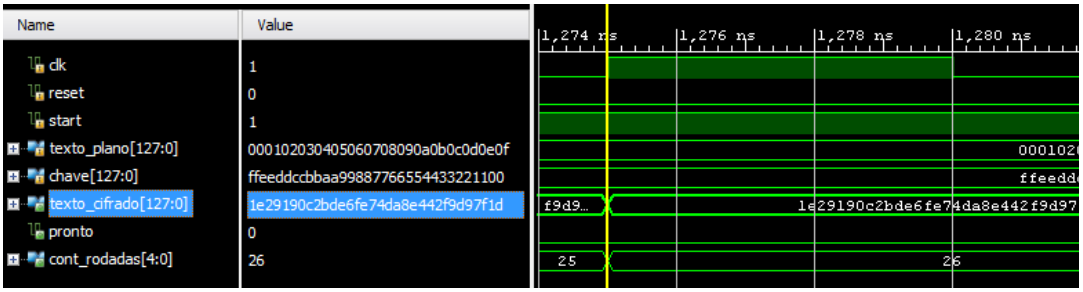
Final da Rodada de Encriptação 1



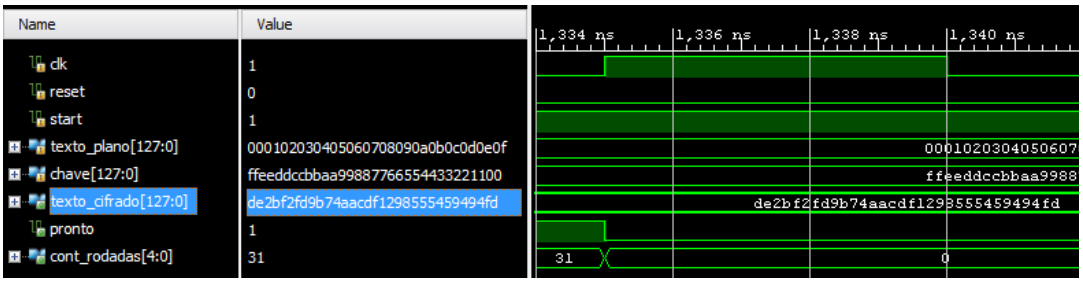
Final da Rodada de Encriptação 6



Final da Rodada de Encriptação 13



Final da Rodada de Encriptação 18 (Texto Cifrado)



Quadro 5.28. Parte dos resultados da simulação do CLEFIA-128 Encriptação

Os resultados da simulação mostraram que os dados correspondem aos resultados dos vetores de testes do algoritmo CLEFIA, confirmando assim correta funcionalidade.

Capítulo 6

Análise de Resultados

Esta seção aborda os resultados dos projetos em *hardware* dos algoritmos AES, PRESENT e CLEFIA, com foco no consumo de recursos do FPGA, consumo de corrente e eficiência, sendo primeiramente descritos os resultados para cada algoritmo, e posteriormente uma comparação entre os projetos é realizada.

6.1 Resultados do Algoritmo AES

Nesta subseção serão apresentados e discutidos os resultados do projeto de criptografia do algoritmo de encriptação AES no FPGA.

6.1.1 Estatísticas de área, taxa de transferência e eficiência

A arquitetura do algoritmo AES implementada neste trabalho, opera em blocos de 128 bits para Texto Plano e 128 bits para Chave. Realiza a encriptação de um bloco de texto em 12 ciclos de clock, ou seja, possui latência de 11 ciclos de clock.

O *hardware* criptográfico do algoritmo AES é formado por módulos que realizam operações específicas, conforme explanado na seção 5.1, onde cada módulo é interconectado de maneira a formar a arquitetura geral de Encriptação, que neste trabalho foi denominada de AES_ENC.

A Tabela 6.1 exibe as estatísticas de recursos de cada módulo do AES, analisado individualmente, e comparado com o módulo Top AES_ENC, através do processo de síntese no FPGA.

Tabela 6.1. Estatísticas da síntese do AES no FPGA (Artix-7)

Módulo/Entidade	Flip-Flops (FF)	% FF	LUTs	% LUTs
AES_ENC (TOP)	**275**	**—**	**1388**	**—**
S-Box	0	0	640	46,11
ShiftRow	0	0	0	0
MixColumns	0	0	128	9,22
Key Expansion	12	4,36	294	21,18
Controle	3	1.1	4	0,29
Circuitos Complementares	260	94,54	322	23,20

O processo de síntese no FPGA, conforme discutido na seção 4.3, realiza uma estimativa de recursos de FFs e LUTs com base no *design* RTL do projeto, conforme características de fabricação do FPGA selecionado.

Através da análise de dados da síntese (Tabela 6.1) é possível observar o percentual estimado de cada módulo do projeto, com relação a arquitetura geral de encriptação (interligação de todos os módulos), com destaque para o módulo S-Box, com consumo estimado de quase metade das LUTs total do projeto (44,11 %), um valor bem expressivo. Isto pode ser explicado devido o comportamento como a S-Box foi descrita em VHDL, utilizando tabelas de substituição para todos os 256 bytes possíveis (16 *x* 16 bytes), conforme mencionado na seção 5.1.1, consumindo uma grande quantidade de LUTs para implementação da lógica. Outra forma possível de sintetizar o módulo S-Box é através do uso de BRAMs, que são blocos de memórias configuráveis disponíveis em alguns FPGAs, porém, apesar do uso de BRAMs reduzir a quantidade de LUTs do projeto, a descrição utilizando BRAMs direciona o desempenho do algoritmo em uma determinada tecnologia de fabricação do FPGA, tendo em vista que a arquitetura de construção de BRAMs é específica para cada FPGA/fabricante. Desta forma, optou-se pela implementação com o uso de LUTs.

Nota-se ainda que a estimativa de consumo de FFs dos circuitos complementares é bem maior que os demais módulos, isto se deve principalmente aos circuitos Registradores (Texto e Chave) que utilizam FFs para armazenagem dos bits (128 bits para cada registrador), outros circuitos como multiplexadores, contador sequencial, e operações XOR do processo de adição de chave da rodada, também fazem parte dos Circuitos complementares (conforme mencionado na seção 5.1.6).

Após o processo de implementação em FPGA. Os valores de área (FFs, LUTs e Slices) e frequência máxima (MHz) de operação foram extraídos de relatórios disponíveis pela ferramentaVivado, e os cálculos de taxa de transferência (Mbps), eficiência (Mbps/slice) e eficiência energética (Ws/bit) foram realizados de acordo com a metodologia descrita na seção 5.3. A tabela 6.2 exibe estatísticas de área e performance do algoritmo AES devidamente implementado no FPGA.

Tabela 6.2. Estatísticas de área e performance do AES implementado no FPGA (Artix-7)

Algoritmo	**Flip-Flops (FFs)**	**LUTs**	**Slices**	**Latência**	**Freq. Max. (MHz)**	**Taxa de Transf. (Mbps)**	**Eficiência (Mbps/slice)**	**Eficiência Energética (*n*Ws/bit) [1]**
AES-128	275	1393	398	11	184,98	2.152,49	5,40	150,04

[1]Valor calculado com base em uma frequência de encriptação de 10 kHz

Analisando a tabela 11 (Implementação) e tabela 10 (síntese) é possível observar uma diferença na quantidade de LUTs entre a síntese e a implementação, essa diferença é justificada pela forma como a lógica foi implementada (mapeado e roteado) no FPGA, através da ferramenta Vivado *Design* Suites.

É possível observar ainda que o *design* implementado alcançou uma alta frequência

de operação (184,98 MHz) proporcionamento taxas de transferência também elevadas (2.152,49 Mbps). Outro valor importante é a medida de eficiência energética, resultando em 150,04 *n*Ws/bit.

Os resultados mostram ainda que a área total de slices necessários para a implementação do algoritmo de encriptação, representa 4,88% do total disponível no dispositivo. A figura 6.1 exibe o circuito do algoritmo AES devidamente mapeado no chip FPGA (área destacada na cor verde).

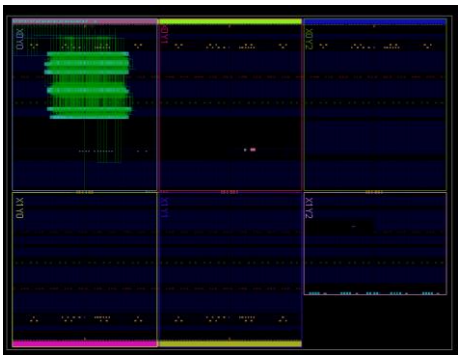


Figura 6.1. Algoritmo AES-128 mapeado no FPGA (XC7A35TCPG236-1)

6.1.2 Consumo de corrente do AES

As medições do consumo de corrente do algoritmo AES foram realizadas fisicamente, através de um protótipo de medição, descrito na seção 4.4, sendo mensurados dados para dois cenários: estático (ocioso) e dinâmico (encriptando dados), conforme descritos na seção 4.6 referente a metodologia de medição deste trabalho.

A figura 6.2 mostra o consumo médio de corrente do AES para ambos os cenários de medição.

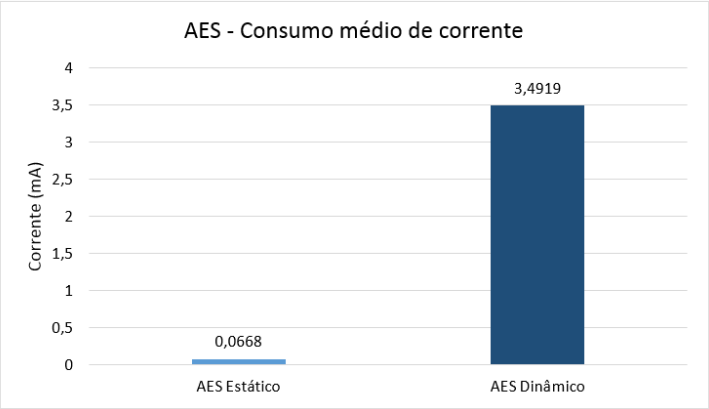


Figura 6.2. Consumo médio de corrente do algoritmo AES-128 no FPGA

O consumo médio de 0,0668 mA para o estado estático pode significar uma pequena corrente de fuga do circuito quando em estado ocioso.

As médias de consumo de corrente coletadas para cada rodada de encriptação do AES no FPGA são exibidas através da Figura 6.3.

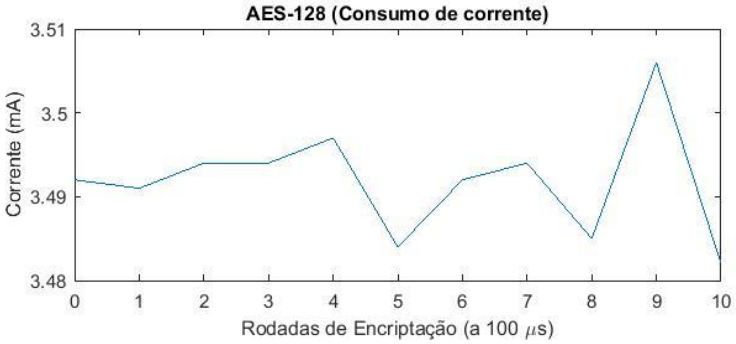


Figura 6.3. Dados de consumo de corrente do algoritmo AES-128 no FPGA

Os resultados mostram o consumo do algoritmo AES-128 durante as 11 rodadas de encriptação, concentrado em torno de 3,49 mA com pequenos desvios. Este consumo reduzido de corrente elétrica se deve principalmente ao avanço da tecnologia no desenvolvimento de dispositivos com o consumo de energia cada vez menor.

Foi realizada também uma análise do consumo de corrente no domínio da frequência, através do método de Welch (descrito na seção 4.5), visando auxiliar na identificação e comparação do algoritmo, através de um possível padrão de consumo. A figura 6.4 mostra a curva do consumo do algoritmo AES após a aplicação do método de Welch, com frequência normalizada e corrente em decibéis (dB).

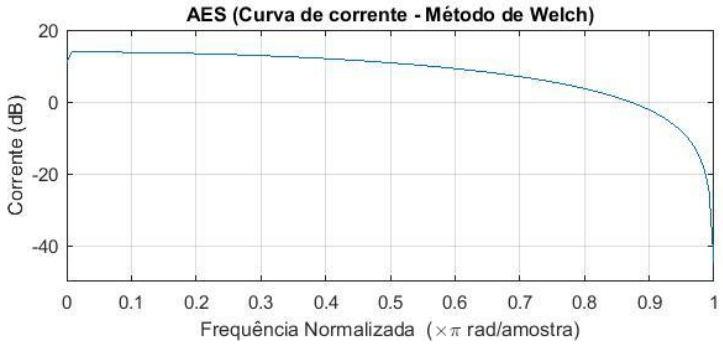


Figura 6.4. Curva do consumo de corrente do AES através do método de Welch

Como pode ser observado, a diferença entre ambas as representações gráficas do consumo de corrente do AES é notória (figuras 6.3 e 6.4), sendo que a representação pelo método de Welch (figura 6.4), mostra uma curva suave, que pode ser útil, tanto na identificação de um possível padrão de consumo, quanto na comparação entre implementações, que será discutida na subseção 6.4.

6.2 Resultados do Algoritmo PRESENT

Nesta subseção serão apresentados e discutidos os resultados do projeto de criptografia do algoritmo de encriptação PRESENT no FPGA.

6.2.1 Estatísticas de área, taxa de transferência e eficiência

O algoritmo PRESENT implementado neste trabalho opera em blocos de texto de 64 bits e blocos de 80 bits para chave. Realiza a encriptação de um bloco de texto em 32 rodadas.

Semelhante ao projeto do algoritmo AES, o *hardware* do algoritmo PRESENT é formado pela interconexão de módulos que realizam uma função específica, conforme descrito na seção 5.2. A Tabela 6.3 mostra os resultados de área de cada módulo, individualmente, em relação a arquitetura geral do PRESENT, após o processo de síntese da descrição em VHDL no FPGA (placa Basys 3, chip Artix-7).

Tabela 6.3. Estatísticas da síntese do PRESENT no FPGA (Artix-7)

Módulo/Entidade	Flip-Flops (FF)	% FF	LUTs	% LUTs
PRESENT_ENC (TOP)	**151**	**—**	**178**	**—**
SBoxLayer	0	0	32	17,98
pLayer	0	0	0	
Keyupd	0	0	7	3,93
Controle	2	1,32	3	1,69
Circuitos Complementares	149	98,68	136	76,40

Pode-se observar através da tabela 6.3 que o maior percentual de LUTs e FFs utilizados no projeto está concentrado nos Circuitos Complementares, que é composto por operações XOR do texto com a chave (AddRoundKey), multiplexadores e contador sequencial.

Tabela 6.4. Estatísticas de área e performance do PRESENT implementado no FPGA (Artix-7)

Algoritmo	Flip-Flops (FFs)	LUTs	Slices	Latência	Freq. Max. (MHz)	Taxa de Transf. (Mbps)	Eficiência (Mbps/slice)	Eficiência Energética (*n*Ws/bit) [1]
PRESENT-80	151	150	51	32	218,01	436,02	8,55	581,8

[1]Valor calculado com base em uma frequência de encriptação de 10 kHz

Os resultados de área e desempenho do PRESENT, confirmam a finalidade para o qual o algoritmo foi desenvolvido, para aplicações que exigem uma quantidade mínima de área, que nesta implementação, consumiu apenas 0.65% do total de slices (8150) no FPGA utilizado, resultando em uma relação de 8,55 Mbps/slice. Outro valor importante a ser destacado é referente ao consumo de energia por bit, que para PRESENT resultou em 581,8 *n*Ws/bit.

A Figura 6.5 mostra o algoritmo PRESENT mapeado no FPGA (área destacada na cor verde).

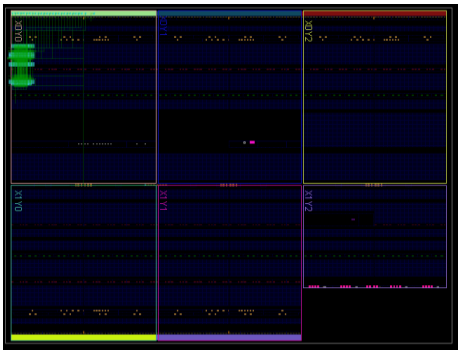


Figura 6.5. Algoritmo PRESENT-80 mapeado no FPGA (XC7A35TCPG236-1)

6.2.2 Consumo de corrente PRESENT

O consumo médio de corrente nos cenários Estático e Dinâmico do algoritmo PRESENT é exibido na Figura 6.6.

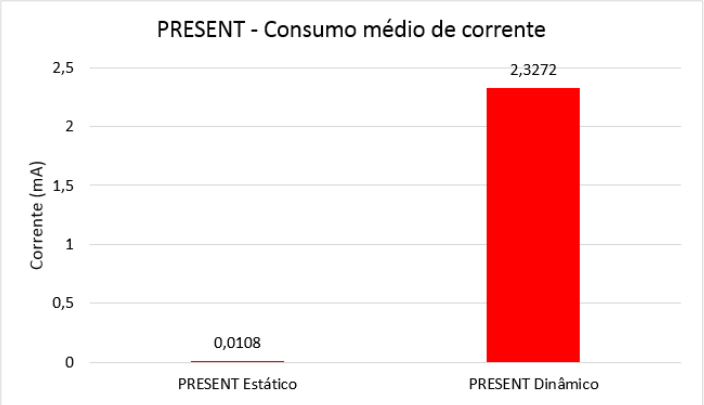


Figura 6.6. Consumo médio de corrente do algoritmo PRESENT-80 no FPGA

Observa-se que no estado Estático quase não há consumo de energia, sendo coletado um valor muito pequeno. Já para o consumo o estado Dinâmico, o consumo médio ficou em 2,32 mA.

As médias de consumo de durante as rodadas de encriptação do algoritmo são exibidas na figura 6.7.

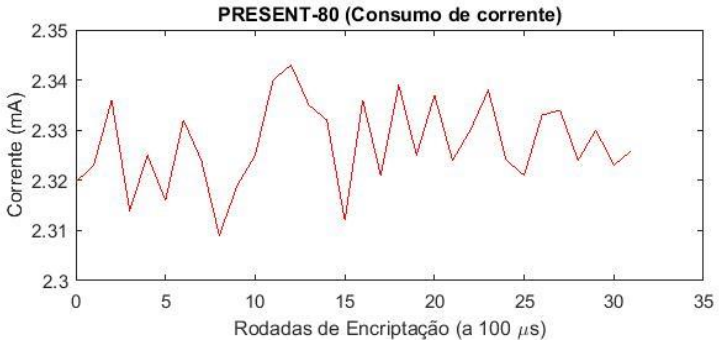


Figura 6.7. Dados de consumo de corrente do algoritmo PRESENT-80 no FPGA

Através da figura 6.7 é possível observar o comportamento do consumo médio de corrente do PRESENT durante as 32 rodadas de encriptação, onde a dispersão em torno

da média é muito pequena.

A curva referente a análise através da estimativa do método de Welch é exibida na figura 6.8.

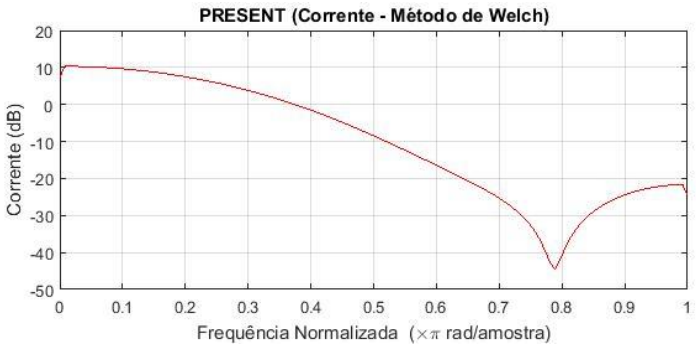


Figura 6.8. Curva do consumo de corrente do PRESENT através do método de Welch

Vale ressaltar que a curva obtida através do método de Welch, é com base em dados de consumo de corrente durante a encriptação de um bloco de texto.

6.3 Resultados do Algoritmo CLEFIA

Nesta subseção serão apresentados e discutidos os resultados do projeto de criptografia do algoritmo de encriptação CLEFIA no FPGA.

6.3.1 Estatísticas de área, taxa de transferência e eficiência

O algoritmo CLEFIA implementado neste trabalho opera em blocos de texto de 128 bits e blocos de 128 bits para chave. Realiza a encriptação de um bloco de texto em 31 rodadas.

Foram realizadas coleta de dados de cada módulo sintetizado no FPGA, e comparado com a arquitetura geral do algoritmo CLEFIA, também através do processo de síntese.

As estatísticas da síntese são exibidas na tabela 6.5.

Tabela 6.5. Estatísticas da síntese do PRESENT no FPGA (Artix-7)

Módulo/Entidade	Flip-Flops (FF)	% FF	LUTs	% LUTs
CLEFIA_ENC (TOP)	**461**	**–**	**1131**	**–**
GF4N	64	13,88	594	68,57
Key Shedule	243	52,71	282	24,93
Controle	22	4,77	25	2,21
Circuitos Complementares	132	28,64	230	20,33

Os resultados mostram que o módulo GF4N, que representa estrutura da Rede de Feistel generalizada de 4 ramificações, consumo um valor estimado de 68,57% de LUTs do projeto. É importante ressaltar ainda que o módulo Key Shedule, responsável pela geração da chave da rodada (em conjunto com o GF4N), representa cerca de 24,93% das LUTs e 52,71 % dos FFs do projeto.

Os valores detalhados do módulo GF4N também foram coletados, tendo em vista que podem representar informações importantes para possíveis otimizações e/ou reestruturação. Os valores de cada módulo/operação que compõe o módulo GF4N, foram comparados em termos de FFs e LUTs com a arquitetura geral, módulo TOP CLEFIA_ENC, sendo que os dados são exibidos na tabela 6.6.

Tabela 6.6. Estatísticas da síntese do módulo GF4N no FPGA (Artix-7)

Módulo/Entidade	Flip-Flops (FF)	% FF	LUTs	% LUTs
CLEFIA_ENC (TOP)	**461**	–	**1131**	–
GF4N	64	13,88	593	52,43
F0	–	–	230	20,33
S0 (2x)	–	–	80	7,07
S1 (2x)	–	–	80	7,07
M0	–	–	38	3,36
AddRK (XOR)	–	–	32	2,83
F1	–	–	235	20,78
S0 (2x)	–	–	80	7,07
S1 (2x)	–	–	80	7,07
M1	–	–	43	3,81
AddRK (XOR)	–	–	32	2,83
WK	–	–	128	11,32
SHIFT	–	–	–	–
Circuitos complementares	64	13,88	0	0

Através dos dados é possível observar uma diferença de 5 LUTs do módulo F1 para o F0, esta diferença se deve principalmente ao fato do módulo M1 (que compõe F1), realizar operações em GF(2^8) (*Galois Field*) de nível mais elevado que em M0, exigindo assim uma quantidade maior de recursos. Observa-se ainda que os módulos de substituição (S0 e S1), se somados, consomem cerca de 28,28% do total de LUTs do projeto.

Os valores de consumo de recursos de área e performance do algoritmo, após a implementação, são exibidos na Tabela 6.7.

Tabela 6.7. Estatísticas de área e performance do PRESENT implementado no FPGA (Artix-7)

Algoritmo	Flip-Flops (FFs)	LUTs	Slices	Latência	Freq. Max. (MHz)	Taxa de Transf. (Mbps)	Eficiência (Mbps/slice)	Eficiência Energética (*n*Ws/bit) [1]
CLEFIA-128	461	1097	324	31	177,40	732,49	2,26	393,89

[1]Valor calculado com base em uma frequência de encriptação de 10 kHz

Os resultados mostram CLEFIA com frequência máxima de operação em 177,40 MHz, um valor expressivo e que contribui para uma alta taxa de transferência (709,6 Mbps). Porém, acredita-se que com a reestruturação dos módulos Key Shedule e GF4N, visando reduzir os ciclos de latência para encriptação do texto, é possível alcançar valores mais expressivos de taxa de transferência de dados, em contrapartida com um custo maior de slices. Outra maneira de equilibrar o bom desempenho com a área implementada, seria

trabalhar os módulos S0 e S1, que juntos representam cerca de 28% do projeto, utilizando outras técnicas de descrição/síntese das operações de substituição.

Com relação a eficiência energética, CLEFIA apresentou cerca de 393,89 *n*Ws/bit.

A Figura 6.9 mostra a área utilizada pelo algoritmo CLEFIA, devidamente mapeada no FPGA.

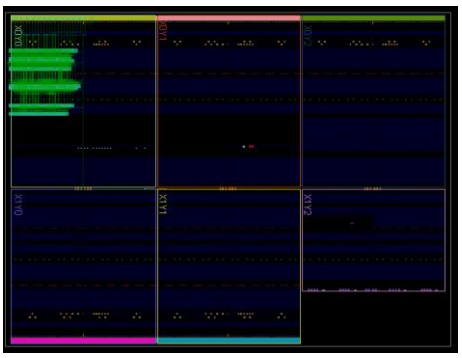


Figura 6.9. Algoritmo CLEFIA-128 mapeado no FPGA (XC7A35TCPG236-1)

A área do CLEFIA mapeado no FPGA representa cerca de 3.97% do total de slices disponíveis. A área total do chip FPGA utilizado neste trabalho é de 1 cm² (já encapsulado).

6.3.2 Consumo de corrente CLEFIA

O consumo médio de corrente nos cenários Estático e Dinâmico do algoritmo CLEFIA é exibido na Figura 6.10.

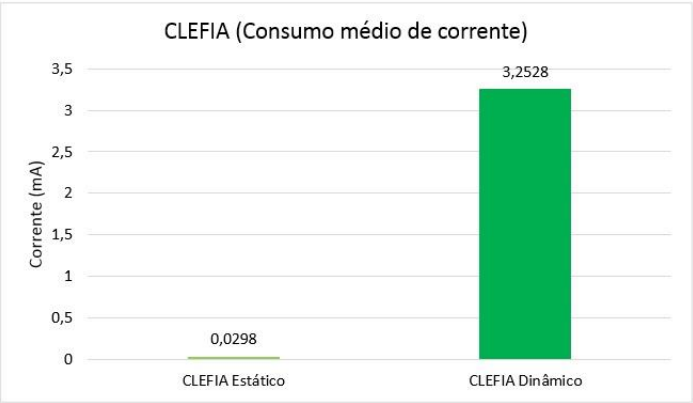


Figura 6.10. Consumo médio de corrente do algoritmo PRESENT-80 no FPGA

CLEFIA apresenta um consumo médio de corrente bem competitivo, cerca de 2,93 mA durante a encriptação, no estado ocioso, o circuito apresenta um consumo de corrente de 0,08 mA.

As representações gráficas do consumo de corrente do CLEFIA durante a encriptação, tanto no domínio do tempo quanto no domínio da frequência (aplicado o método de Welch), são exibidos nas Figuras 6.11 e 6.12, respectivamente.

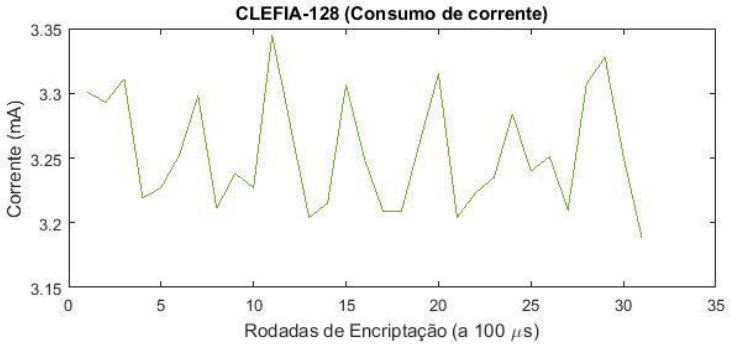


Figura 6.11. Dados de consumo de corrente do algoritmo CLEFIA-128 no FPGA

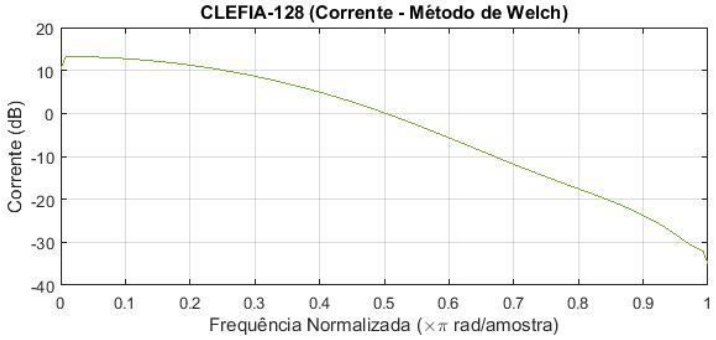


Figura 6.12. Curva do consumo de corrente do CLEFIA através do método de Welch

6.4 Análise comparativa das implementações

Nesta subseção, serão realizadas análises comparativas entre as implementações deste trabalho e outras existentes na literatura.

6.4.1 Comparação de área, taxa de transferência e eficiência

Para a realização da comparação das implentações de AES-128, CLEFIA-128 e PRESENT-80 deste trabalho, foram pesquisados na literatura trabalhos recentes e que utilizassem a tecnologia FPGA Artix-7, porém a maioria dos trabalhos publicados utilizam diferentes dispositivos para a implementação. Desta forma, a análise deste trabalho é mais útil em termos de informação sobre desempenho dos algoritmos foco deste trabalho, comparado com outras implementações, desenhados de diferentes maneiras, utilizando diversas técnicas.

A tabela 6.8 exibe informações da performance dos algoritmos deste trabalho, bem como de outras implementações existentes na literatura.

Tabela 6.8. Comparação AES, CLEFIA, PRESENT deste trabalho com outras implementações.

Algoritmo	Trabalho	Dispositivo	Slices	BRAM	Latência	Freq. Max. (MHz)	Taxa de Transf. (Mbps)	Eficiência (Mbps/slice)	Eficiência Energética (*n*Ws/bit) [1]
AES-128	**Nosso**	**Artix-7 xc7a35tcpg 236**	**398**	**0**	**11**	**184.98**	**2152,49**	**5,40**	**150,04**
AES-128	Hanley e O'Neill (2012)	Virtex-5 xc5vlx50	359	0	26	136,84	673,67	1,87	–
AES-128	El Mara-ghy (2013)	Virtex-5 xc5vlx50	303	10	–	425,0	1327	4,38	–
AES[2]	Resende e Chaves (2015)	Virtex-6 xc6vlx240t-3	115	3	–	332	802	6,97	–
CLEFIA-128	**Nosso**	**Artix-7 xc7a35tcpg 236**	**324**	**0**	**31**	**177,40**	**732,49**	**2,26**	**393,89**
CLEFIA-128	Hanley e O'Neill (2012)	Virtex-5 xc5vlx50	243	0	46	108,66	302,36	1,24	–
CLEFIA-128	Bittencourt *et al.* (2015)	Virtex-5 xc5vlx30	200	3	–	375	1333	6,7	–
CLEFIA[2]	Resende e Chaves (2015)	Virtex-6 xc6vlx240t-3	115	3	–	332	802	6.97	–
PRESENT-80	**Nosso**	**Artix-7 xc7a35tcpg 236**	**51**	**0**	**32**	**218,01**	**436,02**	**8,55**	**581,8**
PRESENT-80	Tay *et al.* (2015)	Virtex-5 xc5vlx50	62	0	295	236.57	51.32	0.87	–
PRESENT-80	Hanley e O'Neill (2012)	Virtex-5 xc5vlx50	87	0	47	250,89	341,64	3,92	–
PRESENT-80	Bogdanov *et al.* (2007)	Spartan-3 xc3s50	202	0	32	254	508	2,51	–

[1]Valor calculado com base em uma frequência de encriptação de 10 kHz
[2]*Design* otimizado dual AES-CLEFIA com chave programável via *software*

Realizando primeiramente uma análise das implementações deste trabalho, o AES alcançou a maior taxa de transferência de dados, chegando a 2.152,49 Mbps contra 436,02 Mbps de PRESENT, que obteve a menor taxa de transferência. Porém, analisando a métrica de eficiência, o algoritmo PRESENT obteve o melhor resultado, de 8,55 Mbps/slice. Já o algoritmo CLEFIA obteve desempenho intermediário na taxa de transferência de dados, sendo de 709,6 Mbps, já com relação a eficiência CLEFIA obteve 2,10 Mbps/slice.

Com relação a comparação de área das implementações, AES foi o algoritmo que

requereu mais slices, um total de 398 slices nesta implementação, CLEFIA requereu uma área menor do que AES, sendo de 324 slices, porém ficou em um nível intermediário, já que PRESENT necessitou apenas de 51 slices nesta implementação.

Outra métrica muito importante a ser destacada e comparada é a eficiência energética, ou seja, a relação de consumo de energia por bit encriptado de cada algoritmo. Apesar do AES obter um consumo de área maior, obteve o melhor resultado em termos de eficiência energética, sendo de 150,04 *n*Ws/bit, já o CLEFIA apresentou 393,90 *n*Ws/bit (262,52 % maior que AES), e o PRESENT, que apesar da menor área de implementação, obteve o maior consumo de energia por bit encriptado, sendo de 581,8 *n*Ws/bit (387,76% maior que AES). Uma justificativa para estes resultados esta relacionada ao tempo de processamente para um bloco de texto, e também o tamanho do bloco, tendo em vista que o AES realiza a encriptação de um bloco de texto (128 bits) em apenas 11 rodadas (ciclos de clock), enquanto que CLEFIA realiza em 31 rodadas (para 128 bits) e PRESENT em 32 rodadas (para 64 bits).

Porém, é importante ressaltar também que cada algoritmo é projetado com o objetivo de atender uma determinada gama de aplicações. O algoritmo AES por exemplo, não foi originalmente desenvolvido para aplicações que requeiram uma área menor, apesar de já desenvolverem versões compactas do mesmo. PRESENT e CLEFIA foram desenvolvidos com o objetivo de atender inúmeras aplicações que requeiram uma área e consumo menor, sendo inclusive padronizados como cifras para Criptografia Leve.

Com relação a comparação das implementações deste trabalho com outras existentes na literatura, as deste trabalho obtiveram desempenho satistatório para implementações que não utilizam BRAMs. BRAM é um módulo de memória que pode ser configurado de inúmeras maneiras, e pode melhorar o desempenho de um circuito. Porém, a utilização deste tipo de técnica, geralmente restringe a descrição em HDL para determinado dispositivo e/ou fabricante.

A implementação de Resende e Chaves (2015), por exemplo, trata-se de uma versão compacta e unificada de dois algoritmos, AES e CLEFIA, obtendo um ótimo desempenho. Porém, além da utilização de BRAMs, outras técnicas de otimização dos processos de substituição e multiplicação matricial foram aplicadas, denominada TBox, bem como todo o processo de geração das chaves das rodadas, tanto para o AES quanto para o CLEFIA, é realizado fora do FPGA, por meio de *software*. Esta também tem sido uma opção de implementação, onde determinadas funções de um algoritmo de criptografia são compartilhadas entre *hardware* e *software*, visando um alto nível de eficiência.

A implementação do CLEFIA, obteve uma eficiência de cerca de 56.62% melhor do que a versão de Hanley e O'Neill (2012), que não utiliza BRAMs e utiliza uma configuração de LUTs semelhantes (6 entradas) em sua arquitetura. Porém é importante ressaltar que se tratam de dispositivos diferentes, com tecnologias e sintetizadores também diferentes. Já em comparação com as versões de Bittencourt *et al*. (2015) e Resende e Chaves (2015), que utilizam BRAMs e técnica TBox, a relação Mbps/slice chega a ser 3 vezes menor.

A versão do PRESENT deste trabalho, obteve a melhor relação de eficiência (Mbps/slice), em comparação com as implementações de Bogdanov *et al*. (2007), Hanley e O'Neill (2012) e Tay et al. (2015). Esses valores podem ser interpretados, não só pela

diferença da tecnologia utilizada, mas também pela arquitetura implementada. Em Tay *et al*. (2015) por exemplo, foi implementada uma versão de 8 bits (entrada e saída), o que aumenta consideravelmente a latência do projeto, enquanto que a arquitetura deste trabalho opera com 64 bits de entrada/saída para texto e 80 bits de entrada para chave.

6.4.2 Comparação de consumo de corrente

Visando uma análise comparativa mais ampla entre as implementações deste trabalho, os dados de consumo médio de corrente para os cenários Estático e Dinâmico, de ambas os algoritmos, são exibidos na Figura 6.13.

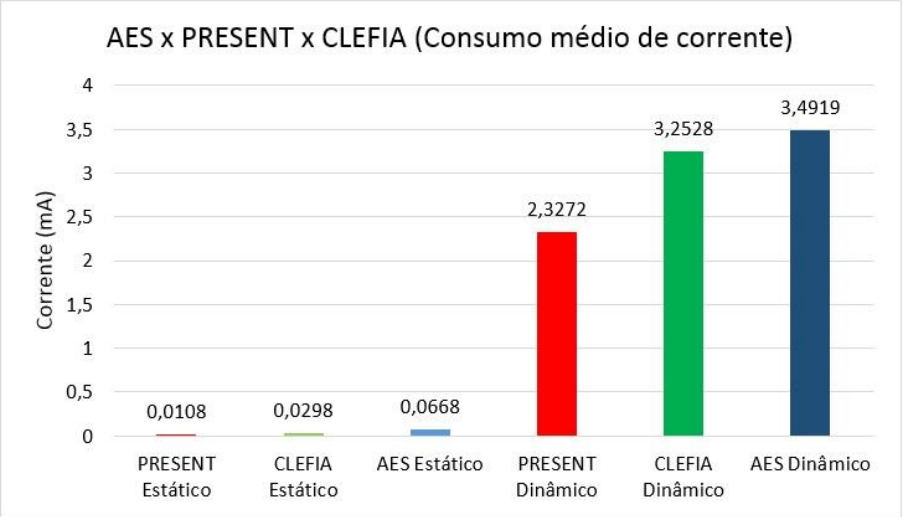


Figura 6.13. Consumo médio de corrente dos algoritmos AES, CLEFIA e PRESENT

Os resultados mostram CLEFIA com um consumo médio maior que AES e PRESENT para a condição Estático, já para a condição Dinâmico (encriptando dados), PRESENT apresentou o menor consumo entre as implementações, cerca de 2,32 mA, e AES o maior consumo, aproximadamente 3,49 mA. Já o CLEFIA, que nesta implementação apresentou consumo de recursos intermediário entre as implementações, porém não muito distante da quantidade de recursos utilizados pelo AES, obteve um consumo médio de 2,93, o que equivale a 83,95% do consumo do AES, e 126,29% do consumo de PRESENT.

Os resultados mostram CLEFIA com um consumo médio maior que AES e PRESENT para a condição Estático, já para a condição Dinâmico (encriptando dados), PRESENT apresentou o menor consumo entre as implementações, cerca de 2,32 mA, e AES o maior consumo, aproximadamente 3,49 mA. Já o CLEFIA, que nesta implementação apresentou consumo de recursos intermediário entre as implementações, porém não muito distante da quantidade de recursos utilizados pelo AES, obteve um consumo médio de 2,93, o que equivale a 83,95% do consumo do AES, e 126,29% do consumo de PRESENT.

As figuras 6.14 e 6.15 exibem o comportamento do consumo de corrente das implementações deste trabalho, através das representações gráficas no domínio do tempo e da frequência (aplicado o método de Welch).

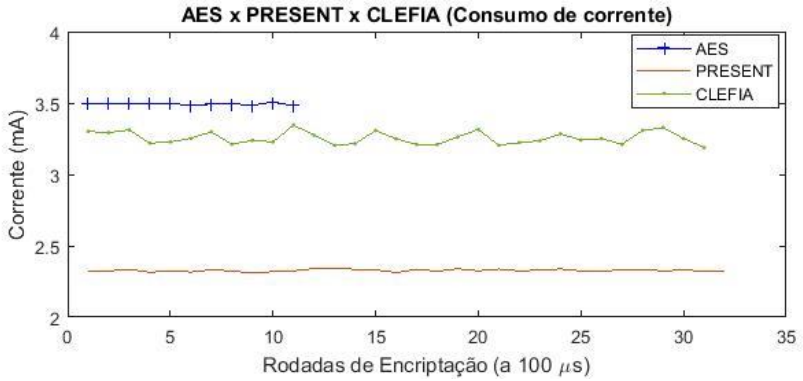


Figura 6.14. Comparação de curvas de corrente do AES, PRESENT e CLEFIA no FPGA

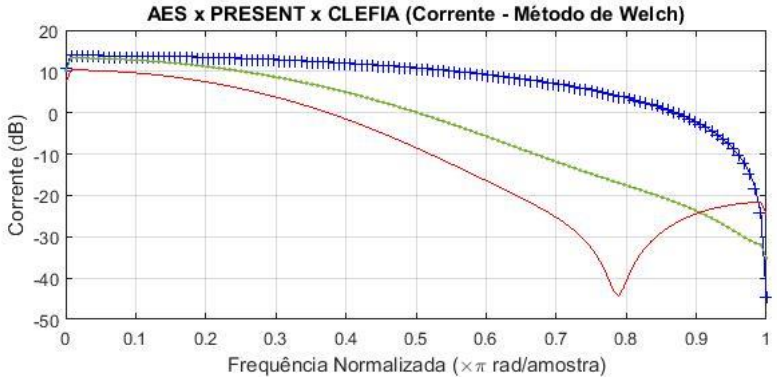


Figura 6.15. Comparação de curvas de corrente AES, PRESENT e CLEFIA através método de Welch

Analisando a figura 6.14 é possível observar com clareza o comportamento do consumo de corrente dos algoritmos durante a encriptação de um bloco de texto, sendo que o algoritmo AES se destaca com um consumo, enquanto CLEFIA apresenta um consumo intermediário, e PRESENT um consumo de corrente menor. Porém é importante observar também que, para as arquiteturas implementadas, AES finaliza a encriptação de um bloco de texto em apenas 11 rodadas, enquanto que PRESENT leva 32 rodadas e CLEFIA levam 31 rodadas.

Com relação a análise da curva de consumo pelo método de Welch (figura 6.15) é possível observar curvas que podem indicar um possível padrão de consumo para determinado algoritmo, o que seria importante para uma identificação fácil e ágil do algoritmo, com base no comportamento das variáveis de consumo de corrente, podendo inclusive contribuir com Ataques que utilizam análise de energia (DPA). Porém, é indicado mais estudos sobre a análise de sinal de corrente ou potência através do método de Welch, com a realização de comparações de consumo entre diferentes arquiteturas, com diferentes parâmetros de chaves de um mesmo algoritmo, visando uma análise minuciosa do comportamento e de um possível padrão, através deste método.

Capítulo 7

Conclusões e Trabalhos Futuros

No presente trabalho foram apresentadas desde a análise comportamental dos algoritos de Criptografia Leve PRESENT-80 e CLEFIA-128 e do algoritmo de criptografia AES-128, como também detalhes da descrição em VHDL do projeto, síntese e implementação em FPGA, com estatísticas de área, taxa de transferência (Mbps) e eficiência (Mbps/slice) eficiência energética (Ws/bit), comparando com outras implementações existentes na literatura, além de uma análise do consumo de corrente.

A seguir, são relatadas as principais contribuições deste trabalho, bem como sugestões de trabalhos futuros.

7.1 Principais contribuições

Dentre os principais resultados e contribuições resultantes da conclusão dos objetivos propostos, destacam-se:

- Implementação dos algoritmos (AES, PRESENT e CLEFIA) seguindo o padrão para blocos de texto do NIST (AES) e ISO/IEC 29192-2-2012 (PRESENT e CLEFIA);
- Descrição VHDL do algoritmo CLEFIA;
- Fornecimento de dados estatísticos de desempenho (inclusive de eficiência energética) dos algoritmos AES, PRESENT e CLEFIA, em plataforma FPGA com tecnologia Artix-7, que pode representar informação relevante para avaliação e escolha de um algoritmo de criptografia;
- Desenvolvimento de um protótipo de medição de consumo de corrente com interface serial I2C, capaz realizar medições de alta precisão (resolução de 0.1 mA);
- Comparação de dados de consumo de corrente dos algoritmos AES, PRESENT e CLEFIA, através de uma implementação real no dispositivo FPGA, bem como uma análise gráfica dos dados de consumo de corrente através do método de Welch, que pode contribuir para identificação de curvas padrão de consumo;
- Disponibilização dos códigos em VHDL dos algoritmos deste trabalho em: < https://github.com/wpmaia/AES-PRESENT-CLEFIA-VHDL>.

7.1.1 Artigos publicados

A partir da pesquisa e resultados obtidos com a realização deste trabalho, foram publicados alguns artigos, sendo estes:

- **Análise de Implementações de Criptografia Leve em Hardware**. Autores: W. P. Maia e E. D. Moreno. Capítulo do Livro: Segurança em Sistemas Embarcados Modernos: Desafios e Tendências, 2015;
- **Current Consumption Analysis of AES and PRESENT Encryption Algorithms in FPGA Using the Welch Method**. Autores: W. P. Maia e E. D. Moreno. In Communications in Computer and Information Science Series (CCIS), 2017. Fifth International Symposium on Security in Computing and Communications (SSCC'17) on. (aceito para publicação).

7.2 Trabalhos futuros

- Otimização de funções e unificação de operações entre os algoritmos AES, PRESENT e CLEFIA, visando o desenvolvendo de um circuito unificado, obtendo uma estrutura de *trial* compacta, que pode ser utilizada em um dispositivo embarcado de baixo consumo;
- Implementação e análise de variáveis de consumo de energia, de diversas arquiteturas dos algoritmos de criptografia, analisando graficamente através do método de Welch, objetivando a identificação de possíveis curvas padrão que podem ser facilmente interpretadas, e que pode auxiliar para uma fácil identificação de qual o algoritmo está sendo utilizado (por meio do consumo de energia).

Referências

BANSOD, G.; RAVAL, N.; PISHAROTY, N. Implementation of New Lightweight Encryption Design for Embedded Security. ***IEEE Transactions on Information Forensics and Security***, v. 10, n. 1, p. 142-151, 2015.

BATINA, L. *et al*. Dietary recommendations for lightweight block ciphers: Power, energy and area analysis of recently developed architectures. ***In: Radio Frequency Identification.*** Springer Berlin Heidelberg, 2013. p. 103-112.

BEAULIEU, Ray *et al*. The SIMON and SPECK Families of Lightweight Block Ciphers. ***IACR Cryptology ePrint Archive***, v. 2013, p. 404, 2013.

BITTERNCOURT, J. C., Resende, J. C., de Oliveira, W. L., & Chaves, R. (2015, August). ***CLEFIA Implementation with Full Key Expansion***. In Digital System Design (DSD), 2015 Euromicro Conference on (pp. 555-558). IEEE.

BOGDANOV, Andrey *et al*. PRESENT: An ultra-lightweight block cipher. ***Springer Berlin Heidelberg***, 2007.

BOGDANOV, Andrey. *et al*. ALE: AES-based lightweight authenticated encryption. ***In: Fast Software Encryption***. Springer Berlin Heidelberg, 2014. p. 447-466.

CANRIGHT, David. ***A very compact S-box for AES***. Springer Berlin Heidelberg, 2005.

CHODOWIEC, Paweł; GAJ, Kris. Very compact FPGA implementation of the AES algorithm. **In: Cryptographic *Hardware* and Embedded Systems-CHES 2003**. Springer Berlin Heidelberg, 2003. p. 319-333.

COVINGTON, Michael J.; CARSKADDEN, Rush. Threat implications of the internet of things. ***In: Cyber Conflict (CyCon), 2013 5th International Conference on***. IEEE, 2013. p. 1-12.

DE CANNIERE, Christophe; DUNKELMAN, Orr; KNEŽEVIĆ, Miroslav. KATAN and KTANTAN—a family of small and efficient *hardware*-oriented block ciphers. ***In: Cryptographic Hardware and Embedded Systems-CHES 2009***. Springer Berlin Heidelberg, 2009. p. 272-288.

DIGILENT. ***Basys 3 FPGA Board Reference Manual***. Digilent Inc. August, 2014. Disponível em:< http://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/>.

EISENBARTH, T.; KUMAR, S.; PAAR, C.; POSCHMANN, A.; & UHSADEL, L. A survey of lightweight cryptography implementations. ***IEEE Design & Test of Computers***, 24(6), 522-533, 2007.

EL MARAGHY, M., Hesham, S., ABD El Ghany, M.A.: ***Real-time efficient FPGA implementation of aes algorithm***. In: 2013 IEEE 26th International on SOC Conference (SOCC), pp. 203–208. IEEE. 2013.

ENGELS, Daniel. *et al*. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. ***In: Financial Cryptography and Data Security***. Springer Berlin Heidelberg, 2010. p. 3-18.

FAN, Xinxin *et al*. FPGA implementations of the Hummingbird cryptographic algorithm. ***In: Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on***. IEEE, 2010. p. 48-51.

GAJEWSKI, K. ***Present a lightweight block cipher***. Disponível em: Open Cores. <https://opencores.org/project,present>. 2014.

GONG, Zheng; NIKOVA, Svetla; LAW, Yee Wei. ***KLEIN: a new family of lightweight block ciphers***. Springer Berlin Heidelberg, 2012.

GUO, Jian. *et al*. The LED block cipher. ***In: Cryptographic Hardware and Embedded Systems–CHES 2011***. Springer Berlin Heidelberg, 2011. p. 326-341.

HANLEY, Neil; O'NEILL, Maire. *Hardware* comparison of the ISO/IEC 29192-2 block ciphers. ***In: VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on***. IEEE, 2012. p. 57-62.

HONG, Deukjo. *et al*. HIGHT: A new block cipher suitable for low-resource device. ***In: Cryptographic Hardware and Embedded Systems-CHES 2006***. Springer Berlin Heidelberg, 2006. p. 46-59.

JUNGK, Bernhard; LIMA, Leandro Rodrigues; HILLER, Matthias. A systematic study of lightweight hash functions on FPGAs. ***In:ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on***. IEEE, 2014. p. 1-6.

LUCERO, Sam *et al*. ***Iot platforms: enabling the internet of things***. IHS Technology. Retrieved from <https://cdn. ihs. com/www/pdf/enabling-IOT.pdf>, 2016.

KOCHER, Paul et al. ***Introduction to differential power analysis***. Journal of Cryptographic Engineering, v. 1, n. 1, p. 5-27, 2011.

KUNDI, Dur-e-Shahwar. *et al*. A compact AES encryption core on Xilinx FPGA. ***In: Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on***. IEEE, 2009. p. 1-4.

MANIFAVAS, C. *et al*. Lightweight cryptography for embedded systems – A comparative analysis. ***In: Data Privacy Management and Autonomous Spontaneous Security***. Springer Berlin Heidelberg, 2014. p. 333-349.

MENEZES, Alfred J.; VAN OORSCHOT, Paul C.; VANSTONE, Scott A. ***Handbook of applied cryptography***. CRC press, 1996.

MORENO, E. D. *et al*. ***Projeto, desempenho e aplicações de sistemas digitais em circuitos programáveis (FPGAs)***. Bless gráfica e editora, 2003.

MORENO, E. D., *et al*. ***Segurança em Sistemas Embarcados Modernos: desafios e tendências***. Aracaju: ArtNer Comunicação, 2015.

MORENO, E. D.; PEREIRA, PEREIRA, F. D.; CHIARAMONTE, R. B. ***Criptografia em Software e hardware***. São Paulo: Novatec, 2005.

NISTIR 8114. ***Report on Lightweight Cryptography***. NIST, 2017. Disponível em: <http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf>.

POSCHMANN, Axel York. Lightweight cryptography: cryptographic engineering for a pervasive world. ***In: Ph. D. Thesis***. 2009.

PROENÇA, Paulo; CHAVES, Ricardo. Compact CLEFIA implementation on FPGAs. ***In: Field Programmable Logic and Applications (FPL), 2011 International Conference on.*** IEEE, 2011. p. 512-517.

RESENDE, João Carlos; CHAVES, Ricardo. Dual CLEFIA/AES cipher core on FPGA. ***In: Applied Reconfigurable Computing***. Springer International Publishing, 2015. p. 229-240.

SHIBUTANI, Kyoji. *et al*. Piccolo: an ultra-lightweight blockcipher. ***In: Cryptographic Hardware and Embedded Systems–CHES 2011***. Springer Berlin Heidelberg, 2011. p. 342-357.

SHIRAI, T. *et al*. The 128-bit blockcipher CLEFIA. ***In: Fast software encryption***. Springer Berlin Heidelberg, 2007. p. 181-195.

SINGH, Arvind et al. ***Exploring power attack protection of resource constrained encryption engines using integrated low-drop-out regulators***. In: Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on. IEEE, 2015. p. 134-139.

SONY Corporation. ***The 128-bit Blockcipher CLEFIA Security and Performance Evaluations***. Disponível em <http://www.sony.net/Products/cryptography/clefia/download/index.html>. Junho, 2007.

SONY Corporation. ***The 128-bit Blockcipher CLEFIA Specification***. Disponível em <https://www.cryptrec.go.jp/english/method.html>. 2010.

SUGAWARA, Takeshi et al. High-performance ASIC Implementations of the 128-bit Block Cipher CLEFIA. ***In: Circuits and Systems, 2008. ISCAS 2008***. IEEE International Symposium on. IEEE, 2008. p. 2925-2928.

TANG, Ming et al. ***Evolutionary ciphers against differential power analysis and differential fault analysis***. Science China Information Sciences, p. 1-15, 2012.

TAY, J. J. *et al*. Compact FPGA implementation of PRESENT with Boolean S-Box. ***In: Quality Electronic Design (ASQED), 2015 6th Asia Symposium on***. IEEE, 2015. p. 144-148.

YALLA, Panasayya; KAPS, Jens-Peter. Lightweight cryptography for FPGAs. ***International Conference on Reconfigurable Computing and FPGAs***, 2009. ReConFig'09. IEEE, 2009. p. 225-230.

ZHANG, Xiaoqiang *et al*. *Hardware* Implementation of Compact AES S-box. ***IAENG International Journal of Computer Science***, v. 42, n. 2, 2015.