CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA

FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO EM HARDWARE E SISTEMAS EMBARCADOS DE ALGORITMOS DE CRIPTOGRAFIA LEVE PARA APLICAÇÃO EM IOT

FERNANDA MAYUMI OHNUMA TACHIBANA

ORIENTADOR(A): PROF. DR. FÁBIO DACÊNCIO PEREIRA

Marília - SP
Dezembro/2017

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA

FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IMPLEMENTAÇÃO EM HARDWARE E SISTEMAS EMBARCADOS DE ALGORITMOS DE CRIPTOGRAFIA LEVE PARA APLICAÇÃO EM IOT

FERNANDA MAYUMI OHNUMA TACHIBANA

Monografia apresentada ao Centro Universitário Eurípides de Marília como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador(a): Prof. Dr. Fábio Dacêncio Pereira

Marília - SP

Dezembro /2017

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA - UNIVEM
MANTIDO PELA FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Fernanda Mayumi Ohnuma Tachibana

Implementação em Hardware e Sistemas Embarcados de Algoritmos de Criptografia Leve para Aplicação em IoT

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 9,5 (nove e meio)

Orientador: Fábio Dacêncio Pereira

1º.Examinador: Rodolfo Barros Chiaramonte

2º.Examinador: Claudio Roberto Costa

Marília, 27 de novembro de 2017.

AGRADECIMENTO

Agradeço primeiramente a Caroline Ferreira pelo apoio durante todo o ano. Agradeço também a todos os professores, especialmente ao Fábio Dacêncio por todos os anos de orientação e também pelo crescimento tanto pessoal quanto acadêmico. Agradeço aos meus amigos de laboratório, Valdir e Thiago e ao professor Rodolfo pelo apoio e disposição em ajudar sempre que precisei.

RESUMO

Recentemente, sistemas computacionais como Internet das Coisas e dispositivos móveis têm atraído muita atenção, alimentada em grande parte pelo crescente interesse em soluções inteligentes e especializadas. Tais sistemas visam abordar a conexão entre objetos físicos, entre objetos virtuais ou ambos, permitindo a coleta, troca, armazenamento de uma grande quantidade de dados que, quando processadas tornam-se informações e serviços. Com o avanço tecnológico nas áreas de sensores e sistemas embarcados, as soluções envolvendo o cenário de Internet das Coisas tornaram-se factíveis. Porém, com a facilidade de acesso a estas informações, surgem grandes desafios como: questões relacionadas à segurança das informações e à privacidade. Estes desafios obrigam o desenvolvimento de um conjunto de protocolos e algoritmos criptográficos específicos para a segurança destes sistemas computacionais, garantindo a confiabilidade, integridade e privacidade das informações. Neste sentido, foram propostos neste trabalho a implementação e avaliação de diferentes algoritmos criptográficos leves, analisando o desempenho e posteriormente comparando-os com outros trabalhos encontrados na literatura.

Palavras-chave: Cifra de Blocos Leves, Segurança, Sistemas Embarcados Críticos, Internet das Coisas.

ABSTRACT

Recently, computer systems like Internet of Things and mobile devices have attracted much attention, fueled in large part by the growing interest in intelligent and specialized solutions. Such systems aim to address the connection between physical objects, between virtual objects or both, allowing the collection, exchange, and storage of a large amount of data that, when processed, becomes information and services. With the technological advance in the areas of sensors and embedded systems, the solutions involving Internet of Things became feasible. However, with the ease of access to this information, great challenges arise, such as: issues related to information security and privacy. These challenges require the development of a set of protocols and cryptographic algorithms specific to the security of these computer systems, guaranteeing the reliability, integrity and privacy of the information. In this sense, we proposed the implementation and evaluation of different lightweight cryptography algorithms, analyzing the performance and comparing them with other works found in the literature.

Keywords: Lightweight Block Cipher, Security, Embedded Critical Systems, Internet of Things.

LISTA DE FIGURAS

Figura 1.1 - Relação de algoritmos de cifras leves dos últimos cinco anos 15

Figura 2.1 - Diagrama de blocos de um microcontrolador – Fonte BANNATYNE; VIOT,1998 19

Figura 2.2 - Arquitetura básica de um UCP – Fonte: GRIDLING et al., 2007.....20

Figura 2.3 - Arquitetura da aplicação JavaCard – Fonte: BAENTSCH et al.,1999....22

Figura 2.4 - Comparação entre UCP e GPU – Fonte: APPLICATIONS; INNOVATION,2012 24

Figura 2.5 - Arquitetura Maxwell – Fonte: SMITH, R.; ANANDTECH, 2014.....25

Figura 2.6 - Arquitetura de um FPGA – Fonte: FAROOQ; MARRAKCHI; MEHREZ, 2012 27

Figura 2.7 - Elemento lógico básico (BLE) – Fonte: FAROOQ; MARRAKCHI; MEHREZ,2012 28

Figura 2.8 - Relação de algoritmos de cifras leves dos últimos cinco anos 29

Figura 2.9 - Estrutura da rede de Feistel.....30

Figura 2.10 - Estrutura de uma rodada da cifra Simon – Fonte: BEAULIEU et al.,2013 31

Figura 2.11 - Estrutura de uma rodada da cifra Speck – Fonte: BEAULIEU et al.,2013 32

LISTA DE TABELAS

Tabela 1 - Relações dos algoritmos que poderiam fazer parte do portfólio de padronização do NIST 38

Tabela 2 - Resultados da implementação em FPGA dos algoritmos Simon e Speck 41

Tabela 3 - Resultados da implementação em microcontrolador dos algoritmos Simon e Speck 42

Tabela 4 - Resultados da implementação em FPGA dos algoritmos Simon e Speck 43

Tabela 5 - Resultados da implementação em smart card dos algoritmos Simon e Speck 44

Tabela 6 - Resultados da implementação em microcontrolador PIC32 dos algoritmos Simon e Speck 46

Tabela 7 - Análise de área e desempenho de Simon 47

Tabela 8 - Análise de área e desempenho de Speck 48

Tabela 9 - Tempo de execução de Simon em GPGPU 49

Tabela 10 - Tempo de execução de Speck em GPGPU 49

Tabela 11 - Comparação da vazão de Simon e Speck com trabalhos correlatos 49

Tabela 12 - Comparação do custo de Simon e Speck com trabalhos correlatos 50

Tabela 13 - Comparação entre tecnologias das implementações Simon e Speck 51

LISTA DE ABREVIATURAS E SIGLAS

AFR - *Async Full Rounds*

APDU - *application protocol data unit*

API - *Application Programming Interface*

ASIC - *Application Specific Integrated Circuits*

BLE - *Basic Logic Element*

CLB - *configurable logic block*

E/S- *Entrada e Saída*

EEPROM- *Electrically-Erasable Programmable Read-Only Memory*

FPGA - *Field programmable Gate Arrays*

GFN - *Generalized Feistel Networks*

GPC - *Graphics Processing Cluster*

GPGPU - *General Purpose Graphics Processing Unit*

IOB - *input/output block*

IoT – *Internet of Things*

ISO - *International Organization for Standardization*

JCRMI - *Java Card Remote Method Invocation*

JVM - *Java Virtual Machine*

LIFO – "*Last In, First Out"*

LUT – *Look-up Table*

NIST - *National Institute of Standards and Technology*

NSA - *National Security Agency*

RAM - *Random Access Memory*

ROM - *Read only memory*

RFID - *Radio-Frequency IDentification*

SFU - *Special Function Units*

SFR - *Sync Full Rounds*

SIMD - *Single Instruction, Multiple Data*

SIMT - *single-instruction multiple-thread*

SIMM - *Maxwell Streaming Multiprocessor*

SP- *Streaming Processors*

SPN- *substitution–permutation network*

SRAM- *Static Random Access Memory*

SSR- *Sync Single Rounds*

UCP- *Unidade central de processamento*

UI - *Unidade de Instrução*

ULA- *Unidade lógica e aritmética*

VHDL - *VHSIC Hardware Description Language*

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO 13

1.1 Motivação e Objetivos 15
1.2 Metodologia de Desenvolvimento do Trabalho 16
1.3 Organização do Trabalho 17

CAPÍTULO 2 - CONTEXTUALIZAÇÃO 18

2.1 Microcontroladores 18
2.1.1 Arquitetura do microcontrolador 20
2.2 Smart Card 21
2.2.1 Arquitetura do JavaCard 22
2.3 GPGPU 23
2.3.1 Arquitetura Maxwell 24
2.4 FPGA 26
2.4.1 Arquitetura do FPGA 26
2.5 Cifras de Bloco Leve 28
2.5.1 Simon 30
2.5.2 Speck 31
2.6 Considerações finais do capítulo 32

CAPÍTULO 3 - TRABALHOS CORRELATOS 33

3.1 Padronização da NIST 33
3.1.1 Cifras que poderiam fazer parte do portfólio 36
3.2 Simon e Speck e a NSA 39
3.3 “Simon and Speck Block Ciphers for the Internet of Things” 41
3.4 “Optimization of Block Cipher with SIMON” 43
3.5 Considerações finais do capítulo 43

CAPÍTULO 4 - RESULTADOS 44

4.1 Smart card 44
4.2 Microcontrolador 45
4.3 FPGA 46

4.3.1 Arquiteturas implementadas....46
4.3.2 Resultados algoritmo Simon....47
4.3.3 Resultados algoritmo Speck....48
4.4 GPGPU....48
4.5 Comparação com trabalhos correlatos....49
CAPÍTULO 5 - CONCLUSÃO....51
5.1 Lições aprendidas....52
5.2 Trabalhos Futuros....53
5.3 Agradecimentos....53
REFERÊNCIAS....54
APÊNDICE A....59
APÊNDICE B....61
APÊNDICE C....63
APÊNDICE D....65
APÊNDICE E....67
APÊNDICE F....70
APÊNDICE G....73
APÊNDICE H....81
APÊNDICE I....83
APÊNDICE J....86
APÊNDICE K....90
APÊNDICE L....93

Capítulo 1

INTRODUÇÃO

Atualmente, há uma evolução significativa de tecnologias como dispositivos móveis, RFIDs, nós sensores, meios de comunicação sem fio, sistemas embarcados entre outros. Estes compõem um sistema computacional que, nos últimos anos, está em evidência: a Internet das Coisas (do inglês, *Internet of Things* - IoT).

Segundo a Forbes (2014), a crescente disponibilidade da Internet banda larga, a redução do custo deste, houve um aumento expressivo sobre o desenvolvimento de produção de dispositivos com capacidade e sensores com Wi-Fi acoplados. Além disso, os custos de tecnologias e a eclosão do uso de smartphones, todas estas coisas favoreceram o surgimento da Internet das Coisas.

O conceito baseia-se em conectar qualquer tecnologia à internet para a comunicação entre elas, desde a smartphones, a eletrodomésticos, relacionando pessoas a pessoas, pessoas a coisas e coisas a coisas.

A empresa de análise Gartner diz que até 2020 haverá mais de 26 bilhões de dispositivos conectados, isto é, são muitas conexões (alguns até mesmo estimaram que esse número seja muito maior, mais de 100 bilhões). (FORBES,2014).

Como a proposta deste sistema é a conexão de tudo e qualquer coisa, cada dispositivo portará e transmitirá dados em seus dispositivos, tanto pessoais quanto protocolos que fazem com que tudo funcione. Assegurar a proteção destes dados e informações torna-se imprescindível. (WEBER,2010).

Para a proteção deste sistema computacional, são utilizadas técnicas de segurança como, por exemplo, a criptografia, onde assegura-se alguns serviços de proteção sobre essa massa de dados, como: confidencialidade, integridade e autenticidade.

Algoritmos criptográficos tradicionais como AES, RSA, SHA-3 são reconhecidamente seguros para uma série de ataques existentes, sendo indicados fortemente para proteger todos os sistemas computacionais. Porém, em alguns cenários onde os dispositivos possuem recursos limitados, estes algoritmos não são adequados, pois exigem uma quantidade significativa de memória, processamento e energia.

Em cenários onde os recursos do hardware são limitados em relação à capacidade de processamento, memória e consumo de energia, é usual separar as informações da massa de dados. A classificação é realizada de acordo com o grau de valor ou significado que o conjunto de dados possui, concedendo a estas informações a proteção criptográfica.

As cifras de blocos leve surgiu, com a possibilidade de dispensar a classificação de dados, sendo uma boa solução para suprir a necessidade de algoritmos criptográficos adequados para proteger integralmente os dispositivos com hardware limitado, sem degradar seu desempenho computacional.

As cifras de blocos são baseadas nas funções de rede de Feistel e substituição-permutação. Este tipo de cifra, funciona com a execução de uma função de mapeamento de blocos possuindo n-bits de texto não cifrado para blocos de n-bits de texto cifrado, onde n é comprimento do bloco. Parametrizada por k-bits, a função da chave K contém o mesmo tamanho do bloco, impossibilitando a expansão dos dados (HANDBOOK. ,2001).

Nos últimos cinco anos surgiram várias cifras de bloco leve, tais como: Simon (BEAULIEU et al.,2013)., Speck (BEAULIEU et al.,2013)., SIMECK (YANG et al., 2015)., RoadRunneR (BAYSAL; SAHIN,2015)., Chaskey (MOUHA et al.,2014)., entre outros.

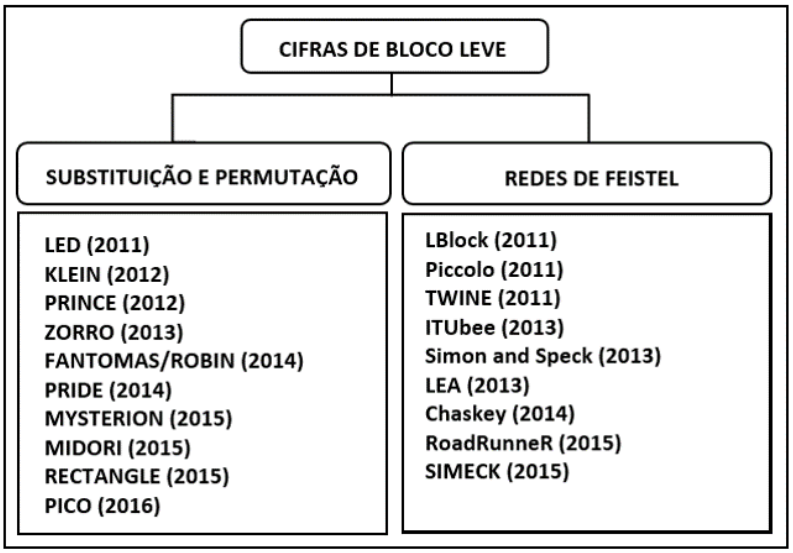

Figura 1.1 - Relação de algoritmos de cifras leves dos últimos cinco anos

Na figura 1.1, apresenta-se a relação dos algoritmos de criptografia leve levantados na revisão bibliográfica, propostos nos últimos 5 anos. Essa evolução tem como principal justificativa o avanço da Internet das Coisas e a crescente necessidade de proteção destes sistemas computacionais.

1.1 Motivação e Objetivos

Com a evolução expressiva em tecnologias como dispositivos móveis, Internet das Coisas (IoT), nós sensores, entre outros, é imprescindível assegurar proteção ao conteúdo digital armazenado nestes tipos de sistemas, porém neste cenário, onde os recursos do hardware são limitados, torna-se inadequado a utilização de criptografias tradicionais, onde a execução do algoritmo poderia ter um alto custo, degradando o desempenho computacional e o alto consumo de energia.

Com o advento das cifras de bloco leve, tornou-se possível a proteção integral dos dados destes sistemas computacionais limitados, sem degradar o desempenho e/ou consumo de energia. Algumas questões devem ser levantadas para destacar as motivações do trabalho:

- Quais os algoritmos de criptografia leve são mais adequados para os cenários de IoT?

- Qual o comportamento de cada algoritmo de cifra de bloco leve implementado em diferentes tecnologias em hardware e software?
- Qual a melhor técnica de implementação de cifras leves considerando desempenho?

Então, o objetivo principal foi explorar a implementação de algoritmos de cifra de blocos leves em tecnologias limitadas (microcontroladores e smart cards) em relação à capacidade de processamento em memória e em tecnologias onde a capacidade de processamento é reconhecidamente avançada como em circuitos programáveis FPGA e GPGPUs.

Neste sentido, estes algoritmos foram implementados e avaliados em arquiteturas de alto desempenho (FPGA e GPGPUs) e também em arquitetura que possui recursos limitados (microcontroladores e smart cards). Ao final, foram mapeados os algoritmos adequados para cenários de IoT. Seguem os objetivos específicos:

- Criação de modelos de arquitetura para avaliação dos resultados;
- Criação de uma biblioteca de implementações de algoritmos criptográficos leves; Definição das cifras de bloco leve que serão desenvolvidas.
- Análise e comparação das arquiteturas com as existentes no estado da arte;
- Mapeamento dos resultados e classificação dos melhores algoritmos de criptografia leve para diferentes cenários de IoT.

1.2 Metodologia de Desenvolvimento do Trabalho

A metodologia de desenvolvimento foi dividida em 4 etapas, sendo:

1. **Pesquisar sobre o avanço e aprimoramento da especificação das Cifras de Blocos Leves (revisão sistemática do tema):** Nesta etapa foram relacionados e estudados os algoritmos mais recentes encontrados no estado da arte no cenário de criptografia de cifras leves para a construção de uma biblioteca de implementações, reunindo os algoritmos mais recentes do estado da arte.

2. **Implementação e validação dos algoritmos criptográficos selecionados:** Foram feitas implementações nas diferentes arquiteturas computacionais das cifras de blocos leves selecionadas na anterior, assim como sua verificação e validação. Para a implementação dos algoritmos, será utilizado VHDL para a plataforma FPGA, linguagem C para microcontrolador e GPGPU e linguagem Java para smart card.
3. **Estatísticas de uso:** Geração das estatísticas de cada algoritmo de criptografia leve para caracterizá-los em cada tecnologia proposta para implementação. As estatísticas geradas por benchmarks ou pelo próprio software de desenvolvimento.
4. **Comparação com arquiteturas existentes:** A partir das estatísticas obtidas, foi possível a comparação com outros algoritmos implementados por outros autores correlacionados na revisão sistemática do tema.

1.3 Organização do Trabalho

Este trabalho está organizado da seguinte forma. No capítulo 2 é apresentado a contextualização das tecnologias utilizadas e sobre cifras de bloco leve, onde são expostas as arquiteturas de cada plataforma e classificação da criptografia leve e apresentação de duas delas: Simon e Speck. No capítulo 3, são referidos trabalhos encontrados no estado na arte com a finalidade de construir uma base referencial sólida para o trabalho. No capítulo 4, descrevemos as arquiteturas propostas e suas especificações para os algoritmos Simon e Speck projetadas e implementadas em diversas tecnologias e seus respectivos resultados.

Capítulo 2

CONTEXTUALIZAÇÃO

Para uma implementação eficiente requer conhecimento das arquiteturas das tecnologias, como também dos designs dos algoritmos criptográficos.

Este capítulo apresenta uma introdução sobre as tecnologias que possui limitações de hardware como: Microcontrolador e smart card; como também as tecnologias que possuem alto desempenho, como: GPGPU e FPGA. Além disso, apresenta as cifras de bloco leve Simon e Speck que foram implementadas neste trabalho.

2.1 Microcontroladores

Microcontrolador é um circuito integrado que une hardware e software para execução de um objetivo específico. Geralmente programados na linguagem C ou Assembly, esta plataforma pode ser utilizada para basicamente qualquer finalidade que o desenvolvedor desejar.

Como pode ser observado na figura 2.1, esta plataforma é composta por elementos básicos como: entrada e saída, *clock*, UCP e memória. Geralmente o microcontrolador possui componentes adicionais como unidades de série e timer. (BANNATYNE; VIOT,1998)

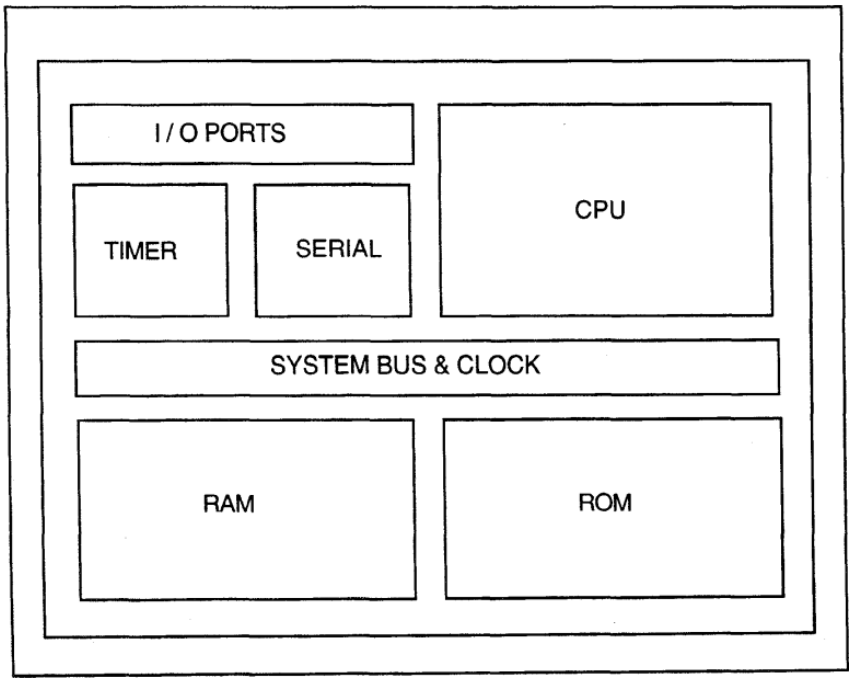

Figura 2.1 - Diagrama de blocos de um microcontrolador – Fonte BANNATYNE; VIOT,1998

Em janeiro de 2016, a empresa Microchip comprou a Atmel, as duas conhecidas pelas linhas de microcontroladores PIC e AVR (LIMA, T., 2016). Elas produzem estas plataformas para uso geral de diversos tipos de configurações, geralmente os tamanhos são 8 bits, 16 bits e 32 bits. Estes são designados para serem flexíveis a diversas aplicações.

Este tipo de plataforma possui um controle de interruptor permitindo a interrupção quando determinados eventos internos ou externos acontecem, possuindo funções como hibernar (*Sleep/Wait*). Além de atuar com uma frequência menor aos computadores comuns, estas características fazem que o consumo de energia seja pequeno. (GRIDLING et al.,2007)

Os microcontroladores atualmente são muito utilizados em áreas como Internet das Coisas, devido ao consumo de energia e custo. O custo diminui devido sua flexibilidade em relação a agilidade no desenvolvimento do projeto, além da plataforma permitir a integração de periféricos através de suas portas de entrada e saída.

Neste trabalho foi utilizado especificamente o microcontrolador Microchip PIC32MX795F512L para implementação das cifras.

2.1.1 Arquitetura do microcontrolador

Como pode ser observado na figura 2.2, a arquitetura de um microcontrolador baseia-se em uma UCP usual, possuindo componentes como a unidade lógica aritmética, registradores, unidade de controle, entre outros.

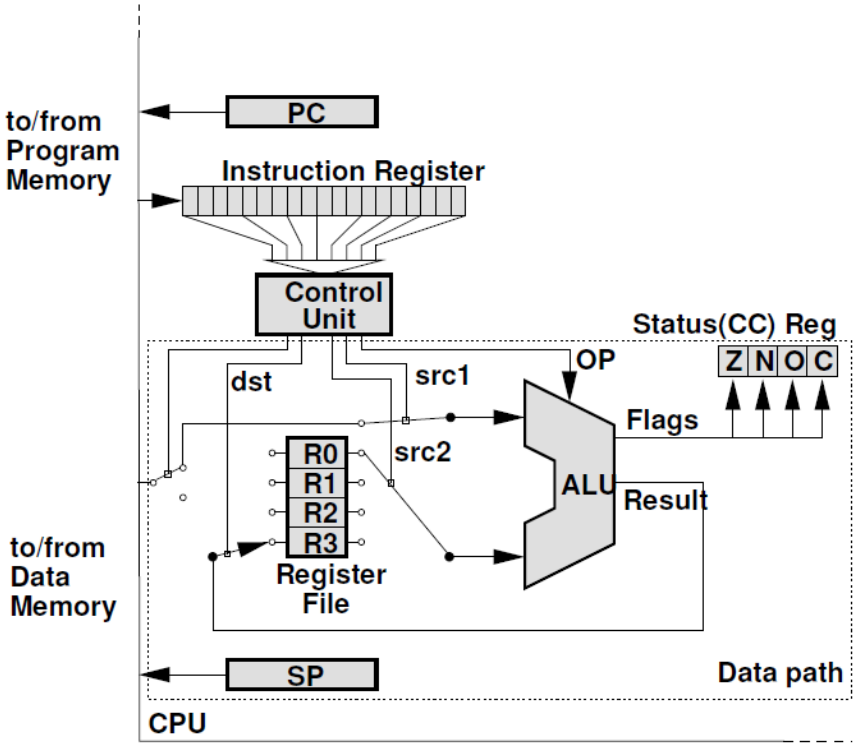

Figura 2.2 - Arquitetura básica de um UCP – Fonte: GRIDLING et al., 2007

A Unidade Lógica Aritmética (ULA) consiste em um circuito combinacional responsável pela execução de cálculos lógicos e aritméticos da UCP (inteiros, lógico, deslocamento de bits), sendo assim, uma peça fundamental da arquitetura.

Os registradores são circuitos combinacionais e sua função é armazenar palavras (conjunto de bits) temporariamente. Na UCP podemos encontrar um conjunto de registradores (acumulador, instrução, gerais, status, etc.), todos possuem uma função particular dentro da arquitetura.

A pilha (LIFO), chamada assim pelo modo em que as palavras são inseridas e removidas ("*Last In, First Out*"), é uma estrutura de memória que permite a escrita ou remoção dos dados através dos comandos de acesso PUSH (inserir) e POP (remover). O local do armazenamento do dado dentro da pilha é definido pelo ponteiro da pilha, assim, os dados vão sendo dispostos em endereços de forma decrescente e retirados de forma inversa, removendo o último inserido.

A memória de pilha é utilizada sempre que o fluxo do programa não é contínuo, armazenando o endereço de retorno para retomar a execução posteriormente. (GRIDLING et al.,2007)

A unidade de controle coordena a UCP interpreta a instrução do registrador de instrução, decodificando-o e gerando os comandos necessários para cada componente da UCP, coordena os componentes da UCP para executar as instruções nas sequências necessárias contidas no programa.

2.2 Smart Card

O smart card consiste em um cartão de plástico contendo um circuito integrado que armazena e processa dados, semelhante a cartões bancários. Esta plataforma transporta dados de uma forma segura, autenticação e acesso seguro a sistemas que exigem um grau elevado de segurança. (ORTIZ,2013)

O padrão ISO 7816 foi estabelecida em 1987 e atualizada em 2003, neste é definido aspectos como características físicas, contatos físicos, sinais eletrônicos, protocolos de transmissão, comandos, arquitetura de segurança, identificadores de aplicação, etc.

Os smart cards mais avançados possuem microprocessadores e memória, sua fonte de energia é fornecida externamente por não possuir uma bateria integrada, sendo assim, só é possível ter acesso ao dispositivo quando conectado a um leitor, este faz uma solicitação de um aplicativo cliente. A plataforma processa e armazena os dados de uma forma mais segura, permitindo o uso de aplicações criptográficas. Alguns smart cards possuem um coprocessador que permite a implementação de algoritmos como RSA, AES e DES. (ORTIZ,2013)

A comunicação é de forma serial e pode ser realizada através da interface de contato ou sem contato, e também permitindo ser ambas. O processamento de dados é realizado pelo sistema operacional do Smart card, atualmente os mais conhecidos são JavaCard e MultOS. (ORTIZ,2013)

Para implementação dos algoritmos criptográficos foi utilizado especificamente o smart card Gemalto ICP D72 FXR1.

2.2.1 Arquitetura do JavaCard

A plataforma JavaCard é caracterizada por um smart card que possui em sua ROM a Máquina Virtual Java (MVJ), possuindo as seguintes características físicas:

- Uma UCP integrada de 8, 16 ou 32 bits (alto desempenho) com 3,7MHz de frequência;
- 1 Kbyte de memória RAM;
- 16 Kbytes de memória EEPROM ou FLASH;

Como pode ser observado na figura 2.3, a arquitetura da aplicação JavaCard consiste na comunicação entre a aplicação do terminal e o cartão. A aplicação do terminal consiste na aplicação do host e o meio de validação de cartão, onde este envia comandos, o JavaCard processa e retorna o resultado.

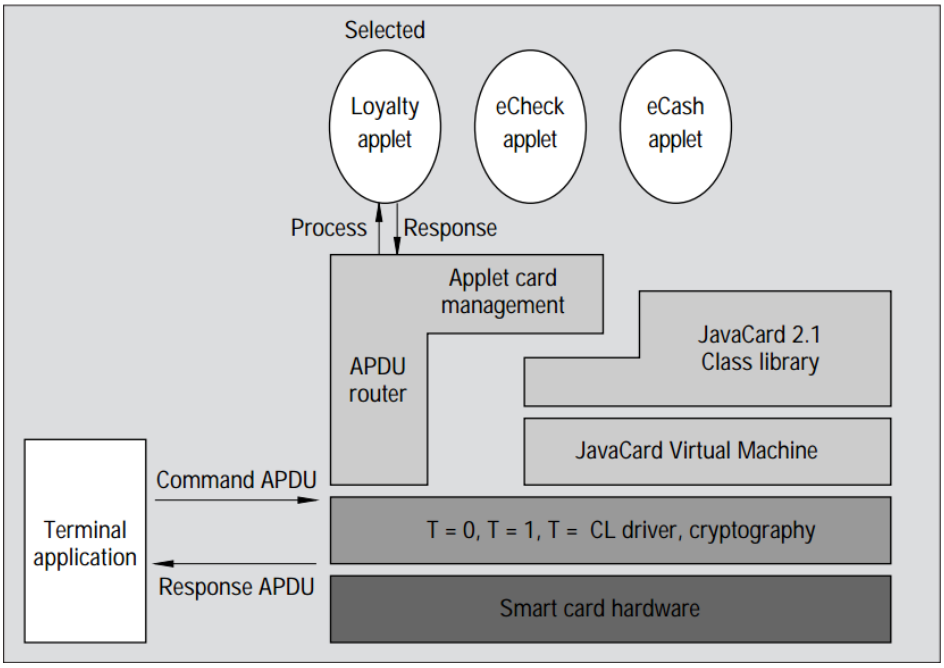

Figura 2.3 - Arquitetura da aplicação JavaCard – Fonte: BAENTSCH et al.,1999

O meio de validação de cartão pode ser um dispositivo individual como um leitor de cartão ou pode estar integrado a um computador. Existem dois modelos de comunicação, o modelo de passagem de mensagem e Método de Invocação Remota JavaCard (JCRMI) entre o host e o JavaCard.

As APDUs (Aplicação de Protocolo de Unidade de Dados) estabelecem a comunicação assíncrona entre o host e a plataforma JavaCard. Este modelo consiste

em um pacote de dados lógicos essencial para a arquitetura da aplicação JavaCard, pois possibilita realizar tanto a comunicação interna pelo leitor e a comunicação entre o leitor e o cartão. Este pacote de dados pode ser tanto orientado por bytes (T = 0 ou através de blocos (T = 1), onde applets apropriados definido pelo Framework JavaCard, processam as APDUs de comando e retornam as APDUs de resposta.

O Método de Invocação Remota consiste no encapsulamento de mensagens dentro do objeto APDU, pois este modelo fornece um recurso de objeto distribuído em cima do modelo de APDUs.

A Máquina Virtual Java (JVM) controla todos os recursos presentes no dispositivo. Segundo (BAENTSCH et al.,1999), a JVM interna do JavaCard é inicializada quando o terminal do *smart card* estimula o hardware e retorna uma reposta, estabelecendo uma comunicação. O sistema entra então em um loop, aguardando comandos do terminal do *smart card*. Quando um comando chega, ele é decodificado e seleciona um *applet* específico ou é encaminhado para um *applet* selecionado anteriormente. No último caso, o controle passa para o código Java, invocando o método *process* () do *applet*. A JVM então executa qualquer sequência de operações programada no *applet*. Isso pode incluir a leitura de dados adicionais do terminal do *smart card*, a descodificação de alguns dados usando as chamadas JavaCard Cryptography API ou o armazenar dados em objetos alocados na EEPROM. Quando o método termina, o ambiente de tempo de execução do JavaCard retorna uma palavra de *status* ao terminal, juntamente com todos os dados que o *applet* escreveu para a interface de E/S.

2.3 GPGPU

Neste trabalho os algoritmos criptográficos foram implementados na GPU GTX 970 da nVidia. A GPGPU é uma plataforma que contém milhares de unidades de processamento gráfico, que ao invés de ser utilizadas para processamento gráfico, são utilizadas para cálculos de propósito geral. Por incentivo da própria empresa nVidia, nos anos 2000, alavancou o mercado de GPGPU pelo interesse da empresa sobre computação de alto desempenho.

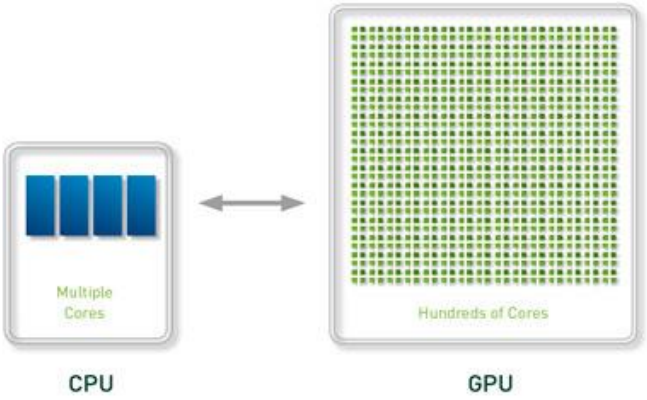

Figura 2.4 - Comparação entre UCP e GPU – Fonte: APPLICATIONS; INNOVATION,2012

As GPUs tiveram sua origem quando necessitaram de um componente de hardware que processasse imagens de uma forma mais eficiente do que uma UCP. No início o processamento era dedicado a modelos 2D e resolução do monitor, mas ao longo do tempo, com a popularização dos modelos 3D, começaram a ser desenvolvidas APIs gráficas para jogos e visualização de dados. (SOLLER, 2011)

Antigamente, todo efeito gráfico possuía uma API específica implementada fixa, o que limitava os programadores gráficos. Para mudar esse cenário limitado, a arquitetura da GPU foi modificada tornando-se menos específica para cada elemento de processamento gráfico (textura, pixels, cálculo de vértices, etc.), com uma arquitetura mais refinada (geralmente chamada de *Unified Shader Architecture*) esta plataforma tornou-se flexível e eficiente. (SOLLER, 2011)

2.3.1 Arquitetura Maxwell

Esta é conhecida como SIMT executando uma única instrução em múltiplos threads de forma paralela (CLUA, E. G. W.; ZAMITH, M, 2015). A arquitetura é composta por 4 *Cluster* de Processamento Gráfico (GPC) dispostas em paralelo. Os GPCs são constituídos por 4 *Streaming Processors* (SP), são eles os núcleos e são similares a UCP padrão por terem suporte para *multi-threading* simultâneo. Já o SFUs (*Special Function Units*), são úteis para computação rápida de funções como exponenciais, logaritmos, seno e coseno. Os GPCs são compostos também por Unidade de Instrução (UI) e *Maxwell Streaming Multiprocessor* (SMM), eles são compostos por blocos de threads (1024 threads por bloco), onde dentro de cada SMM, os blocos podem compartilhar de dados localmente e utilizar memória global para troca de informações entre eles. A SMM é fisicamente próxima dos núcleos e tem um

acesso muito rápido, mas é de tamanho pequeno (128KB em Maxwell). O número de SMM pode variar de acordo com a geração e o modelo da GPU (CLUA, E. G. W.; ZAMITH, M, 2015). A figura 2.5 mostra a arquitetura Maxwell da segunda geração.

Figura 2.5 - Arquitetura Maxwell – Fonte: SMITH, R.; ANANDTECH, 2014

As arquiteturas das GPUs evoluíram com base na computação paralela com foco na vazão, processando um grande número de dados. Todas as GPUs modernas utilizam técnicas de processos de *multi-threading* de hardware e SIMD (instrução única, dados múltiplos). (FATAHALIAN; HOUSTON, 2008)

A arquitetura *multicore* utiliza o conceito de programação paralela que permite o processamento de dois ou mais programas em múltiplos núcleos simultaneamente partilhando da mesma memória, isto propicia a utilização de vários processos e threads aumentando o desempenho da plataforma. (FARIAS; SEGUNDO; BATISTA, 2015)

O processamento SIMD é um método que aplica a mesma instrução simultaneamente a uma gama de dados regulares. Esta técnica aumenta o desempenho, amenizando a complexidade da decodificação de um fluxo de instruções e do custo das estruturas de controle de uma ULA em múltiplas ULAs, resultando em uma execução de chip eficiente em termos de energia e área. (FATAHALIAN; HOUSTON, 2008)

As GPGPUs utilizam o *multi-threading* para ocultar o acesso a memória e as latências de pipeline de instruções. Em tempo de execução, a GPGPU particiona o armazenamento do chip para ter o máximo de threads possíveis. Esta é uma estratégia que intercala a execução de múltiplos segmentos para aumentar a vazão na arquitetura. (FATAHALIAN; HOUSTON, 2008)

2.4 FPGA

Os FPGAs (*Field programmable Gate Arrays*) é uma plataforma composta por portas lógicas reprogramáveis e reconfiguráveis, entrada e saída e pode ser praticamente qualquer circuito ou sistema que o projetista deseja em menos tempo e custo comparado a um projeto em ASIC (*Application Specific Integrated Circuits*). É uma tecnologia que está bastante presente na indústria onde há necessidade de alto desempenho, principalmente em aplicações em tempo real.

Atualmente existem Xilinx e a Altera que são as duas grandes empresas que produzem FPGAs, sendo a Xilinx a maior fabricante e líder deste mercado (LIMA, T., 2015), em segundo encontra-se a empresa Altera que recentemente foi comprada pela Intel. (COMPUTERWORLD, 2015)

Para a implementação das cifras foi utilizado a tecnologia FPGA Xilinx Virtex-7 XC7VX330T.

2.4.1 Arquitetura do FPGA

Como mostra da figura 2.6, o FPGA é composto por basicamente três componentes: Blocos de entrada e saída (IOB), blocos lógicos configuráveis (CLB) e

Switch Matrix, além de memórias (SRAM e EEPROM). (FAROOQ; MARRAKCHI; MEHREZ, 2012)

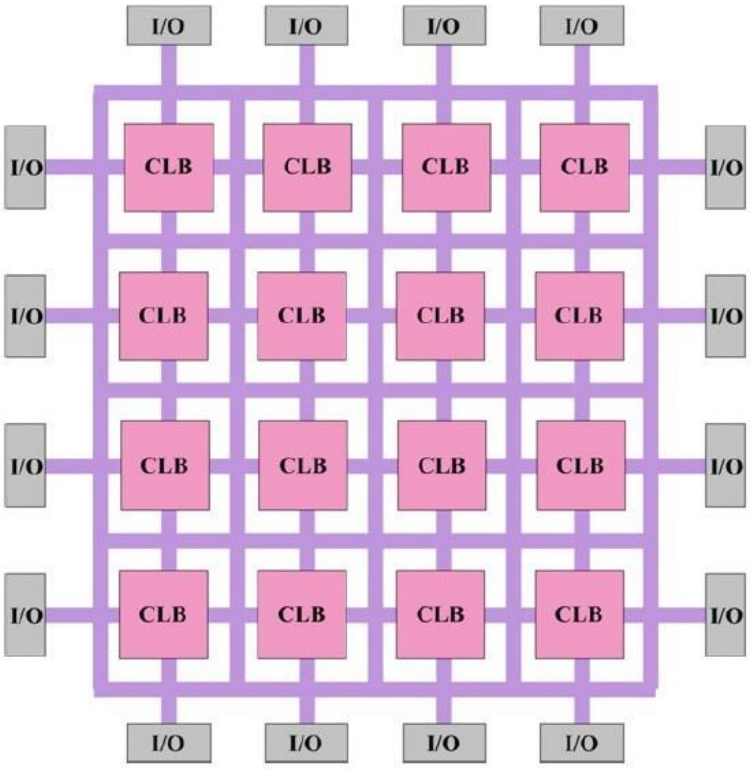

Figura 2.6 - Arquitetura de um FPGA – Fonte: FAROOQ; MARRAKCHI; MEHREZ, 2012

Os CLBs são compostos por células SRAM, estes blocos são utilizados para implementar funções lógicas. Alguns blocos são baseados em LUT que é um bloco de memória volátil que possui uma lógica básica e funcionalidade de armazenamento de apenas um bit (0 ou 1). Um bloco lógico pode ser composto por um elemento lógico básico (BLE) ou por um conjunto de BLEs. Um elemento lógico básico possui um LUT e um Flip-Flop. A Figura 2.7 mostra um BLE simples composto de um Flip-Flop tipo D e LUTS de 4 entradas. (FAROOQ; MARRAKCHI; MEHREZ,2012)

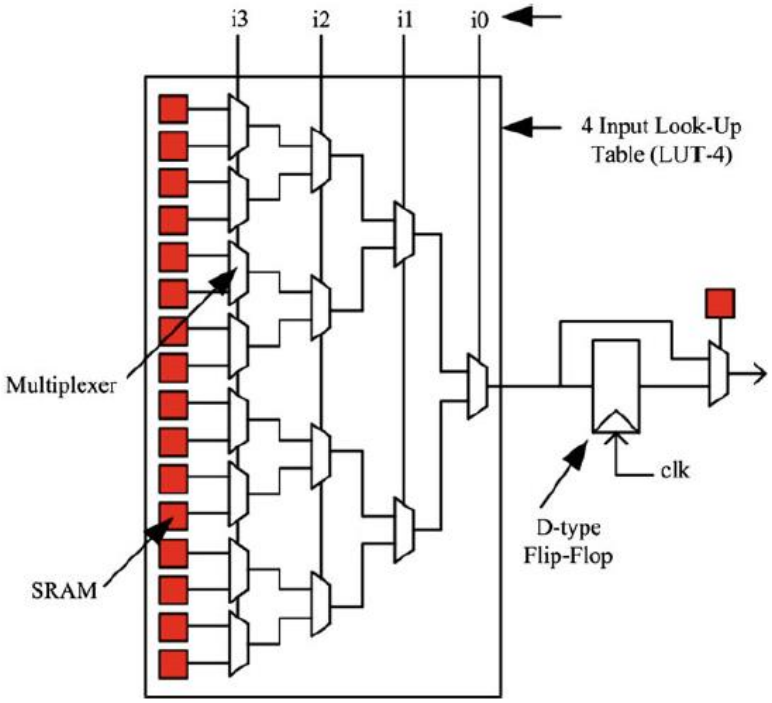

Figura 2.7 - Elemento lógico básico (BLE) – Fonte: FAROOQ; MARRAKCHI; MEHREZ,2012

O FPGA possui conexões de roteamento entre os blocos lógicos e os blocos de entrada e saída. Estas conexões são constituídas por fios e switches programáveis, onde as configurações de roteamento entre os CLBs e IOBs são estabelecidas de acordo com as funções lógicas e suas conexões internas. (FAROOQ; MARRAKCHI; MEHREZ,2012)

Após apresentadas as tecnologias utilizadas, a partir da seção seguinte serão apresentados a classificação dos algoritmos criptográficos leves e suas especificações.

2.5 Cifras de Bloco Leve

Para compreender os algoritmos a serem implementados, requer o conhecimento das cifras e a base de seus designs. A cifra de Bloco Leve é uma criptografia simétrica em bloco. O seu termo "leve" não se refere à segurança da cifra, e sim, porque elas são desenvolvidas para dispositivos que possuem recursos limitados em relação a processamento, memória e energia. Como mostra a figura 2.8,

este tipo de cifra possui dois designs principais: Redes de Feistel e redes de Substituição-Permutação.

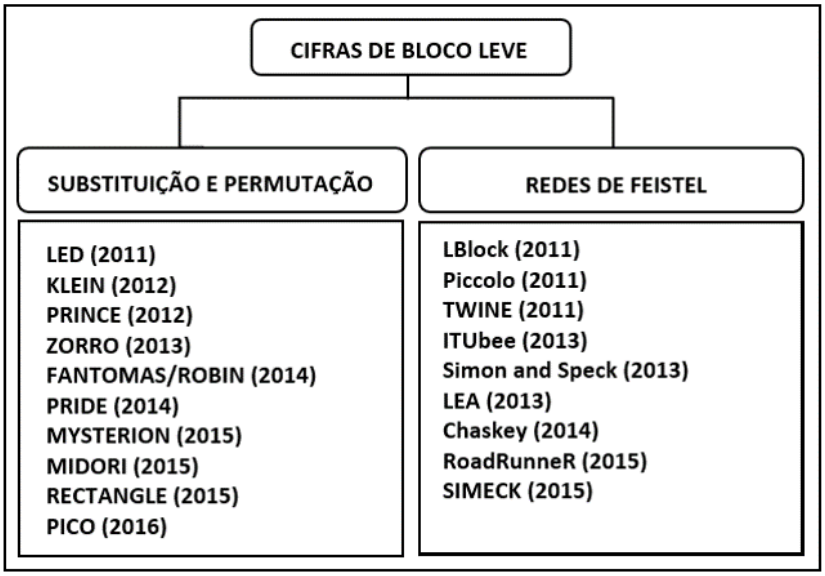

Figura 2.8 - Relação de algoritmos de cifras leves dos últimos cinco anos

A rede de Substituição-Permutação (SPN) foi proposta em 1949 por Shannon, que nos dias de hoje são a base da maioria das cifras como o AES. Esta rede é baseada em duas primitivas criptográficas: Substituição (S-Box) e Permutação (P-Box). Estas duas primitivas provêm do objetivo de Shannon de confundir e difundir.

Horst Feistel desenvolvedor de cifras da IBM, propôs a rede de Feistel em 1973, base de algumas cifras de blocos leves. A rede divide em duas palavras (*Left e Right*) que recebe várias iterações ao passar pelo *round function*. Para cada rodada (*round*) é gerada uma sub-chave por um algoritmo gerador de chaves.

Em cada rodada a palavra Ri-1 faz uma iteração com uma determinada função, depois realiza-se uma soma com a palavra Li-1, no fim, acontece a permuta entre os lados, como demonstra a figura 2.9:

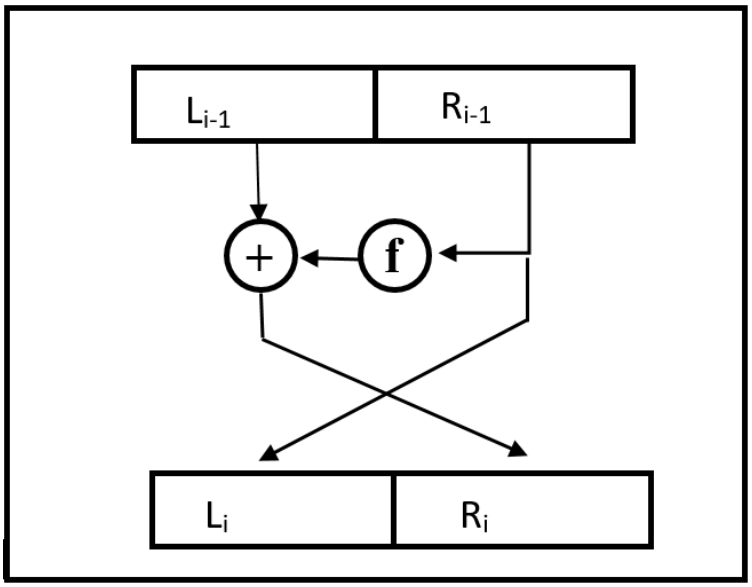

Figura 2.9 - Estrutura da rede de Feistel

2.5.1 Simon

A cifra Simon (BEAULIEU et al.,2013) é baseada na rede de Feistel e foi projetado para ser muito pequeno para aplicações em hardware. Denotada com tamanho de bloco 2n e tamanhos de chave mn, onde n é o tamanho da metade do bloco e pode assumir tamanho 16, 24, 32, 48 ou 64. Quando é escrito Simon64/128 refere-se a versão da cifra Simon com bloco tamanho 64 bits e tamanho de chave 128 bits. Para cifrar e decifrar, esta cifra utiliza operações como xor, and e deslocamento de bits.

A *round function* é dada por:

$$R_k(x, y) = (y \oplus f(x) \oplus k, x), \quad (2.1)$$

Onde f (x) é a função e k é a sub-chave. O inverso do *round function*, usada para decifrar, é:

$$R_k^{-1}(x, y) = (y, x \oplus f(y) \oplus k), \quad (2.2)$$

A figura 2.10 representa a equação de rodada de Simon apresentada acima.

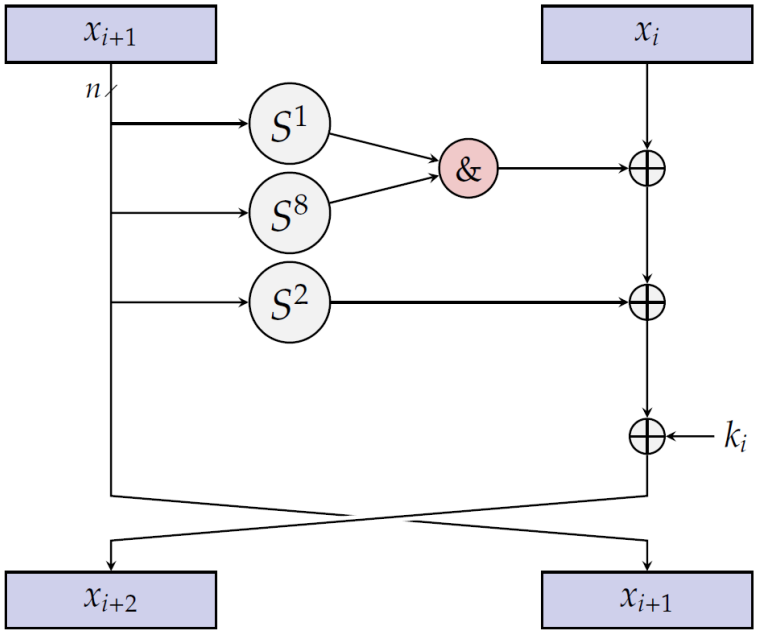

Figura 2.10 - Estrutura de uma rodada da cifra Simon – Fonte: BEAULIEU et al.,2013

A quantidade de rodadas T para cifrar e a quantidade de sub-chaves geradas a partir da chave principal depende do tamanho de n (T = 32, 36, 42, 44, 52, 54, 68, 69, 72).

2.5.2 Speck

A cifra Speck (BEAULIEU et al.,2013) é baseada na rede de Feistel e foi projetado para ser eficiente tanto em hardware quanto em software, porém foi otimizada para possuir um desempenho melhor em aplicações em software. A sua representação de versão é a mesma que Simon. Para cifrar e decifrar, esta cifra utiliza operações como xor, adição modular e deslocamento circular de bits.

A função de cifrar é dada por:

$$R_k(x,y) = (S^{-\alpha}x + y) \oplus k, S^{\beta}y \oplus (S^{-\alpha}x + y) \oplus k), \quad (2.3)$$

A função inversa para decifrar:

$$R_k^{-1}(x,y) = \left(S^{\alpha}\left((x \oplus k) - S^{-\beta}(x \oplus y)\right), S^{-\beta}(x \oplus y)\right), \quad (2.4)$$

A figura 2.11 representa a equação de rodada de Speck apresentada acima.

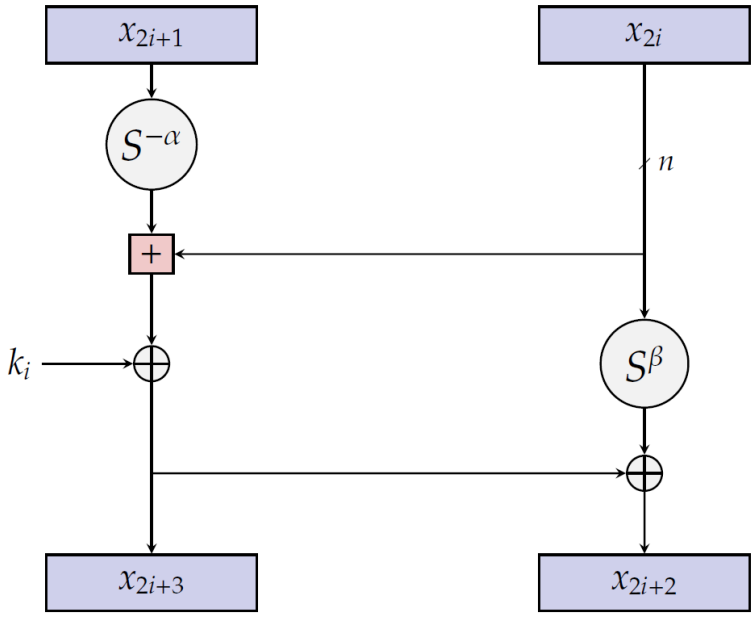

Figura 2.11 - Estrutura de uma rodada da cifra Speck – Fonte: BEAULIEU et al.,2013

2.6 Considerações finais do capítulo

Todas as tecnologias e conceitos discutidos neste capítulo são de suma importância para o melhor entendimento das tecnologias trabalhadas e dos algoritmos criptográficos, desta forma aplicando os conhecimentos de forma eficiente e correta.

O próximo capítulo trata-se da apresentação dos trabalhos correlatos com ênfase em cifra de blocos leves que são discutidos nesse trabalho.

Capítulo 3

TRABALHOS CORRELATOS

Esta seção tem como objetivo abordar alguns artigos encontrados no estado da arte, os quais tratam sobre contextualização, padronização, portfólio de algoritmos de cifras de blocos leves e por fim, a seleção de algoritmos realizados neste trabalho.

Além disso, foram abordados artigos encontrados no estado da arte, os quais tratam sobre implementações dos algoritmos de cifra leve Simon e Speck, ambos com ênfase nas plataformas FPGA e microcontrolador. São apresentados dois trabalhos, em que os resultados de suas implementações são comparados em relação ao desempenho dos algoritmos em cada tecnologia. É importante ressaltar que não foi encontrado no estado da arte sobre trabalhos que destacam implementações das cifras Simon e Speck em smart card e GPGPU que mensurem o desempenho, porém existem implementações em GPGPU de outras cifras leves nos trabalhos de (PARK, MOO KYU, AND JI WON YOON, 2015) e (QIU, WEIDONG, ET AL, 2016)

3.1 Padronização da NIST

Dispositivos com capacidades limitadas estão se tornando cada vez mais evidentes. Com essa visibilidade, esses dispositivos têm recebido mais atenção dos pesquisadores e do NIST (Instituto Nacional de Padrões e Tecnologia).

O NIST é um instituto dos Estados Unidos, onde um dos seus objetivos é a padronização, promovendo a inovação e a competitividade industrial americana. Para padronizar a segurança de dispositivos computacionais, o instituto já promoveu concursos, selecionando algoritmos criptográficos com perfis que garantam a

confidencialidade, integridade e autenticidade. O primeiro vencedor deste concurso foi a cifra de bloco AES em 2001 e a segunda foi a função hash SHA-3 no ano de 2015.

Com a expansão do cenário de dispositivos que possuem recursos limitados como bateria e processamento, o NIST iniciou em 2013 um projeto que visa estudar criptografia leve (MCKAY et al.,2016) para entender a necessidade desses algoritmos criptográficos em cada cenário em que os recursos computacionais são restritos e, em seguida, projetar um processo de padronização destes. Em 2015 e 2016, o instituto realizou workshops sobre criptografia leve, onde reuniu a comunidade científica para exposição e discussão de requisitos e recursos para aplicações reais de cifras de blocos leves.

O NIST atualmente reconhece dois algoritmos de cifra de bloco, AES e TDEA; No entanto, para a implementação em cenários onde os dispositivos são limitados apenas a cifra de bloco AES é recomendada; seu design é específico para ser usado em software, com três versões AES-128, AES-192 e AES-256. No entanto, para fins onde necessita-se de cifras mais leves, recomenda-se usar o AES-128, pelo tamanho das chaves e o número de rodadas cifração e decifração. Porém deve ser ressaltado que a cifra AES só deve ser implementada quando seu desempenho for aceitável dentro da aplicação.

O cenário de criptografia leve tem crescido rapidamente, nos últimos anos vários algoritmos foram desenvolvidos tanto derivados de cifras existentes tanto totalmente novas. O esforço da comunidade científica para desenvolver uma cifra que seja adequada para dispositivos limitados é essencial, mas não há padronização para este tipo de algoritmo. Nesse sentido, o NIST planeja desenvolver uma coletânea versátil de algoritmos leves, onde novas cifras podem ser incluídas ou excluídas de acordo com suas características. Estas cifras de bloco leve terão que cumprir as métricas estabelecidas pela ISO/IEC 29192, Lightweight Cryptography. (MCKAY et al.,2016)

O NIST fará a coletânea de criptografias leves devido à velocidade da produção de cifras atualmente. Uma competição de padronização leva anos, como é o caso do SHA-3, que levou oito anos para ser anunciado como o vencedor do concurso. Com a quantidade de cifras desenvolvidas anualmente, até o final da competição a cifra de bloco vencedora poderia estar desatualizada. (MCKAY et al.,2016)

Para a avaliação das criptografias, o NIST utilizará parâmetros específicos que variam de acordo com a aplicação, além disso, precisam estar dentro da maioria dos requisitos gerais de design, são eles:

- **Força de segurança**: Qualquer algoritmo selecionado para a coletânea deve fornecer uma segurança de pelo menos 112 bits para ser considerada adequada.
- **Flexibilidade**: Deve ser possível implementar eficientemente uma cifra em várias plataformas. O algoritmo também deve suportar variações de implementações em uma única plataforma. É desejável algoritmos que permitem a flexibilidade nos parâmetros, tais como o tamanho do bloco e tamanho da chave pois utiliza menos recursos, suportando vários aplicativos.
- **Baixa sobrecarga para múltiplas funções**: de preferência, as cifras devem ter cifração e decifração das funções de rodada semelhantes, reduzindo assim o custo de implementação de vários algoritmos no mesmo dispositivo.
- **Expansão de texto cifrado**: Algoritmos criptográficos leves que não expandem significativamente seu texto cifrado são desejáveis, de modo que não se tornem custoso para a aplicação.
- **Canal lateral e ataques de falha**: Prefere-se cifras de bloco que fornecem facilidade de proteção contra-ataques de canal lateral e injeção de falhas. Implementações podem deixar informações importantes vulneráveis em várias vertentes, particularmente sobre a chave ou texto simples. O ataque de canal lateral usa propriedades de implementação durante a execução de operações criptográficas, como tempo, consumo de energia e emissão eletromagnética, para descobrir informações confidenciais. A injeção de falhas recupera informações sensíveis introduzindo erros na computação. No caso de dispositivos pervasivos, isto é particularmente perceptível pois os atacantes podem ter acesso físico aos dispositivos e a disposição contra tais ataques pode não ser pré-enviada devido a recursos limitados.
- **Limites no número de pares texto-texto cifrado**: Pode ser viável para os projetistas de algoritmos projetar uma quantidade maior em relação ao número de texto simples/cifra usando a mesma chave, uma vez que isso é justificado para implementações em dispositivos com baixo desempenho. O designer deve estar ciente de ataques criptoanalíticos sobre a chave. No entanto, o

projeto de um gerador que produz chaves individuais também pode sofrer ataques, bem como ataques multichave. Portanto, recomenda-se a obtenção de um equilíbrio entre a arquitetura do algoritmo e a aplicação na qual ele será incorporado.

- **Ataques de chaves relacionadas**: Este tipo de ataque visa descobrir a chave, observando as operações sobre os textos com diferentes chaves. Quando as chaves não são escolhidas aleatoriamente, gera uma certa dependência entre essas chaves, criando um relacionamento entre elas, tornando-as vulneráveis a ataques mal-intencionados. É preferível algoritmos onde chaves são geradas aleatoriamente e independentemente um do outro.

3.1.1 Cifras que poderiam fazer parte do portfólio

Nesta subseção é apresentado uma lista de algoritmos de cifra de bloco leve que poderiam estar presentes no portfólio da NIST, onde neste trabalho os algoritmos criptográficos foram estudados e selecionados de acordo com as características que atendem aos requisitos de seleção exigido.

- **HIGHT (HONG et al.,2006)**: O algoritmo é classificado como baseado em ARX, foi projetado para hardware que possui poucos recursos, como dispositivos ubíquos. HIGHT suporta comprimento de bloco de 64 bits e tamanho de chave de 128 bits.
- **CLEFIA (SHIRAI et al.,2007)**: Esta cifra suporta apenas tamanho de 128 bits por bloco e tamanho de 128, 192 e 256 bits por chave. Ele tem a flexibilidade de ser implementado em software e hardware, mas é mais eficiente quando implementado em hardware. É classificado como *Generalized Feistel Networks* (GFN) e é padronizado, tornando-se parte da ISO-29192.
- **PRESENT (BOGDANOV et al.,2007)**: Com o surgimento dos dispositivos limitados, o AES tornou-se não adequado para aplicações em alguns dispositivos, assim surgiu a motivação para projetar a cifra PRESENT, para garantir a proteção de RFIDs e sensores. Este algoritmo é classificado como a Rede Substituição-Permutação (SPN), permitindo o tamanho de bloco de 64 e 80 bits, comprimento de chaves de 128 bits. PRESENT foi padronizado com a cifra CLEFIA, ambos fazem parte da norma ISO- 29192.

- **GOST revisited (POSCHMANN; LING; WANG,2010)**: GOST é um antigo algoritmo criptográfico soviético, padronizado pela agência russa em 1989. Nesta versão otimizada do algoritmo, o consumo de área foi reduzido e suporta tamanho de bloco de 64 bits e tamanho de chave de 256 bits. Este algoritmo é categorizado como Feistel e é mais eficiente para implementações em hardware.
- **Piccolo (SHIBUTANI et al.,2011)**: Essa cifra suporta blocos de 64 bits e comprimentos de chave de 80 bits e 128 bits, sendo eficiente e compacta quando implementada em hardware, adequada para dispositivos limitados, classificada como GFN (*Generalized Feistel Networks*).
- **LED (GUO et al.,2011)**: Apesar de serem projetados para hardware, os autores tentaram manter um perfil para que sua cifra não perca tanto desempenho quando implementado em software. Este algoritmo é classificado como Substituição-Permutação de Rede (SPN) tem tamanho de bloco de 64 bits, comprimento de chave de 64 e 128 bits.
- **TWINE (SUZAKI et al.,2011):** Esta cifra permite a implementação em hardware do algoritmo consumindo uma área pequena, também é eficiente quando implementado em software incorporado. Baseado na rede Feistel, ele suporta blocos de tamanhos de 64 bits e Chaves de 80 e 128 bits.
- **LEA (HONG et al.,2014):** O primeiro autor de HIGHT é também o primeiro autor do LEA, classificado como baseado em ARX, a cifra tem comprimento de bloco de 128 bits e comprimentos de chave de 128, 192, 256 bits. Implementado nas plataformas Intel, AMD, ARM e ColdFire, o algoritmo mostra ser mais rápido que o AES.
- **SIMON and SPECK (BEAULIEU et al.,2013):** Os autores foram motivados pelo problema de uma série de cifras sendo produzidas para um único dispositivo, porém quando implementadas em outros dispositivos perdiam seu desempenho. Nesse sentido, foram criados o SIMON e SPECK, propostos para atender todas as plataformas de hardware e software, respectivamente. Esta cifra é classificada como uma rede Feistel e suporta blocos de 32, 48, 64, 96 e 128 bits e 64, 72/96, 96/128, 96/144 e 128/192/256-bit tamanhos de chaves, correspondentes aos tamanhos de bloco, respectivamente.

- **Rectangle (ZHANG et al.,2015):** Esta cifra funciona bem em hardware e software e é muito competitivo em todas as plataformas pois foi projetado para ser flexível. Classificada como Substituição-Permutação de rede (SPN), suporta tamanhos de bloco de 64 bits e 80, tamanho de chave de 128 bits.
- **RoadRunner (BAYSAL; SAHIN,2015):** Esta cifra foi projetada devido ao problema de um grande número de cifras de bloco leve que estavam sendo produzidas no ambiente científico, onde a maioria deles são deficientes em segurança daqueles designados para o software restrito. Portanto, o RoadRunner é uma cifra de redes de Feistel, eficiente para software de 8 bits, garantindo a segurança dos dados. A cifra suporta comprimento de bloco de 64 e 80 bits, tamanho de chave de 128 bits.
- **SIMECK (YANG et al.,2015):** Baseado em dois blocos de código de criptografia leve, SIMON e SPECK, o SIMECK combina as funções de rodadas de SIMON e a programação de chaves SPECK, projetando um algoritmo mais eficiente e compacto em hardware. Classificada como rede de Feistel, esta cifra suporta blocos de 32, 48 e 64 bits e chaves de 64, 96 e 128 bits correspondentes aos tamanhos de bloco, respectivamente.
- **SPARX (DINU et al.,2016):** Baseado em ARX foi projetado de acordo com a Estratégia *Long Trail* para implementações software, sua execução sendo uma das mais rápidas existentes no estado da arte. A cifra suporta blocos de tamanho 64 e 128 bits e chaves de 128 e 128/256 bits de acordo com o tamanho do bloco, respectivamente.

As relações dos algoritmos apresentados foram reunidas na tabela 1 com seus principais pontos.

Tabela 1 - Relações dos algoritmos que poderiam fazer parte do portfólio de padronização do NIST

Apresentação			**Propriedades da cifra**			
Nome	**Designers**	**Ano**	**Tamanhos de blocos**	**Tamanhos de chave**	**Estrutura**	**Melhor desempenho**
CLEFIA	Shirai et al.	2007	128	128 192 256	GFN	Hardware
GOST revisited	Poschmann et al.	2010	64	256	Feistel	Hardware
HIGHT	Hong et al.	2006	64	128	GFS	Hardware
PRESENT	Bogdanov et al.	2007	64	80 128	SPN	Hardware

Apresentação			Propriedades da cifra			
Nome	**Designers**	**Ano**	**Tamanhos de blocos**	**Tamanhos de chave**	**Estrutura**	**Melhor desempenho**
Piccolo	Shibutani et al.	2011	64	80	GFN	Hardware
				128		
LED	Guo et al.	2011	64	64	SPN	Hardware
				128		
TWINE	Suzaki et al.	2011	64	80	GFN	Hardware e Software
				128		
LEA	Hong et al	2013	128	128	ARX	Hardware
				192		
				256		
Simon	Beaulieu et al.	2013	32	64	Feistel	Hardware
			48	72/96		
			64	96/128		
			96	96/144		
			128	128/ 192/ 256		
Speck	Beaulieu et al.	2013	32	64	ARX	Software
			48	72/96		
			64	96/128		
			96	96/144		
			128	128/ 192/ 256		
Rectangle	Zhang et al.	2015	64	80	SPN	Hardware e Software
				128		
RoadRunner	Baysal et. al.	2015	64	80	Feistel	Software
				128		
SIMECK	Yang et al.	2015	32	64	Feistel	Hardware
			48	96		
			64	128		
SPARX	Dinu et al.	2016	64	128	SPN (ARX)	Software
			128	128/256		

Fonte: Elaborado pelo próprio autor

3.2 Simon e Speck e a NSA

Em novembro de 2016, a NSA (AGENCY,2016) decidiu publicar um documento onde retrata qual a intenção da agência de segurança com as cifras Simon e Speck. Por essas cifras terem repercutido bastante na comunidade científica, a NSA esclarece algumas questões sobre as cifras.

Em 2013, a NSA lançou um par de algoritmos Simon e Speck para a comunidade criptográfica analisar e rever. O objetivo para Simon e Speck é fornecer opções que poderiam permitir que dispositivos futuros em ambientes altamente restritos sejam protegidos, em particular onde as opções atuais não são viáveis e onde soluções classificadas não são possíveis.

A agência considera importante a questão da evolução rápida da Internet das Coisas (IoT), uma explosão de dispositivos conectados que irão afetar profundamente nossas vidas diárias e mudar a maneira como interagimos com o mundo físico ao nosso redor. A IoT expande a Internet em uma rede de objetos inteligentes ("coisas") que têm a capacidade de se comunicar uns com os outros e com recursos centralizados. As tecnologias de IoT proeminentes incluem hoje o RFID e as redes wireless do sensor, cujas as aplicações variam da automatização de casa a controlar de tráfego à monitoração médica.

Quando considerado o rápido aumento dos dispositivos conectados, é evidente que estamos enfrentando desafios sem precedentes para a segurança e privacidade, afirma a NSA. Para aplicações muito restritas, os algoritmos criptográficos escolhidos devem ser eficientes o suficiente para caber dentro dos recursos escassos disponíveis. A pesquisa está atualmente em andamento no crescente campo da criptografia “leve” que se esforça para proteger a comunicação nesses ambientes onde energia, área de chip, RAM, ROM, etc. disponíveis são limitadas, sem sacrificar a segurança.

Para enfrentar esses desafios, o NIST anunciou recentemente sua intenção de criar um portfólio de primitivos leves. Algoritmos serão recomendados para uso somente no contexto de perfis, que descrevem características físicas, desempenho e segurança. Esses perfis destinam-se a capturar requisitos de algoritmos criptográficos impostos por dispositivos e aplicações onde a criptografia leve é necessária.

A NSA planeja apresentar variantes apropriadas de Simon e Speck para consideração no processo do NIST. Como o tamanho de chave mínimo exigido pelo NIST é de 112 bits, somente as variantes com tamanho de chave de pelo menos 128 bits são esperadas para ser apropriado para o processo de seleção de primitivo leve NIST.

Um elemento-chave da concepção de Simon e Speck é a sua flexibilidade, ou seja, a sua capacidade de executar bem em uma gama completa de plataformas de hardware e software. O objetivo é apoiar implementações em redes heterogêneas, onde o bom desempenho pode ser necessário em muitos tipos diferentes de dispositivos. Com isso dito, Simon é otimizado para aplicações de hardware e Speck para aplicações de software. Assim, para uma aplicação principalmente em dispositivos de hardware (respectivamente software), Simon (ou Speck) é a melhor escolha.

A flexibilidade de Simon e Speck é um subproduto da sua simplicidade. Eles são construídos usando um conjunto de operações muito simples (AND, XOR, adição, deslocamento circular) que pode ser facilmente executado em praticamente qualquer dispositivo capaz de computação.

No outono de 2014, uma emenda foi proposta para incluir Simon e Speck estão passando pelo processo de padronização com a Organização Internacional de Padronização (ISO). O padrão de criptografia leve ISO (ISO / IEC 29192-2) atualmente contém PRESENT (que tem um bloco de 64 bits e permite chaves de 80 ou 128 bits) e CLEFIA (que tem um bloco de 128 bits e permite chaves de 128, 192 ou 256 bits). Além disso, Simon e Speck foram propostos para o padrão de interface RFID ISO, ISO/ IEC 29167.

3.3 “Simon and Speck Block Ciphers for the Internet of Things”

As cifras de bloco leve Simon e Speck foram desenvolvidas pela NSA, com o objetivo de apoiar implementações em redes heterogêneas, onde o bom desempenho pode ser necessário em muitos tipos diferentes de dispositivos. Com isso dito, Simon é otimizado para aplicações de hardware e Speck para aplicações de software.

Logo no trabalho de BEAULIEU, et al. (BEAULIEU et al.,2015) foram realizadas implementações dos algoritmos Simon e Speck em plataformas como circuitos integrados de aplicação especifica (ASICs), FPGAs e microcontroladores, com a finalidade de mensurar o desempenho destas cifras em diversos cenários para aplicação em sistemas como a Internet das Coisas (IoT).

Entre as implementações realizadas nas três plataformas citadas no trabalho de BEAULIEU, et al (BEAULIEU et al.,2015), destacam-se as implementações de tamanho 64/128 em FPGA relacionadas na tabela 2 com o intuito de parametrizar os resultados com o nosso trabalho.

Tabela 2 - Resultados da implementação em FPGA dos algoritmos Simon e Speck

Tamanho (bloco/chave)	Algoritmo	Area (Slice LUTs)	Vazão (Mb/s)
64/128	Simon	138	512
	Speck	153	416

Fonte: BEAULIEU et al.,2015

Os autores realizaram dois tipos de implementação em microcontrolador: Alta vazão/baixo consumo de energia e minimização de RAM.

Alta vazão/baixo consumo de energia: essas implementações desenrolam a função de rodada para alcançar uma vazão com cerca de 3% de uma implementação totalmente desenrolada. A chave, armazenada em flash, é usada para gerar as sub-chaves das rodadas que posteriormente são armazenadas na RAM. (BEAULIEU et al.,2015)

Minimização de RAM: Essas implementações evitam o uso de RAM para armazenar as sub-chaves, incluindo as chaves pré-expandidas na memória flash. Não há programação de chave para atualizar a chave expandida, tornando essas implementações adequadas para aplicativos onde a chave é estática. (BEAULIEU et al.,2015)

Os algoritmos implementados em microcontrolador, foram implementados em assembly, especificamente nas plataformas AVR ATmega128 e MSP430. Os resultados foram relacionados na tabela 3, com tamanho 64/128 de ambos algoritmos.

Tabela 3 - Resultados da implementação em microcontrolador dos algoritmos Simon e Speck

AVR ATmega128				
Alta vazão/baixo consumo de energia				
Tamanho (bloco/chave)	**Algoritmo**	**ROM (bytes)**	**RAM (bytes)**	**Custo (cyc/byte)**
64/128	Speck	628	108	122
	Simon	436	176	221
Minimização de RAM				
64/128	Speck	218	0	154
	Simon	290	0	253
MSP430				
Alta vazão/baixo consumo de energia				
64/128	Speck	556	0	89
	Simon	324	0	153
Minimização de RAM				
64/128	Speck	204	0	98
	Simon	280	0	177

Fonte: BEAULIEU et al.,2015

Segundo os autores, esses dois algoritmos são ideais para uso em redes heterogêneas, onde possuem a flexibilidade para serem otimizados para aplicações específicas.

3.4 “Optimization of Block Cipher with SIMON”

Com um intuito de implementar a cifra Simon em uma arquitetura de baixo custo, o trabalho de Shylaja C. e Shreekanth T. (SHYLAJA, C., & SHREEKANTH, T., 2015), o realizou na plataforma FPGA Spartan-3 serializando bits, reduzindo o tamanho da área do algoritmo de tamanho de bloco 128 e 128 de tamanho da chave. Os resultados das implementações nas propostas arquiteturas que reduz a área e que aumenta a vazão foram relacionados na tabela 4.

Tabela 4 - Resultados da implementação em FPGA dos algoritmos Simon e Speck

Cifra	Área (Slice LUTs)	Vazão (Mbps)
Simon128/128	36	3,6
Simon128/128	399	216,8
Simon128/128	766	392.4

Fonte: SHYLAJA, C., & SHREEKANTH, T., 2015

Finalmente, pode-se concluir que o SIMON mostra 50% da redução da área em comparação com a AES para a sua implementação ASIC e também mostrou que, com a redução de 86% do SIMON, é uma alternativa mais forte ao AES para aplicações FPGA de baixo custo.

3.5 Considerações finais do capítulo

Todos os trabalhos correlatos discutidos neste capítulo foram importantes para entender melhor o estado da arte das cifras de bloco leves. Devido ao cenário que se encontra, foram selecionados para ser implementados neste trabalho os algoritmos criptográficos Simon e Speck.

Os resultados da implementação em FPGA e microcontrolador apresentados pelos trabalhos contidos nesta seção serão utilizados para comparação com os resultados obtidos neste trabalho.

Capítulo 4

RESULTADOS

Nesta seção é apresentado em cada subseção as implementações dos algoritmos em Java, C, VHDL e CUDA, sendo SimonN/T, SpeckN/T com N = tamanho do bloco e T = tamanho da chave, utilizando as tecnologias smart card Gemalto ICP D72 FXR1, microcontrolador PIC32MX795F512L, FPGA Xilinx Virtex-7 XC7VX330T e GPU GTX970 Nvidia, sucedendo a comparação destes sobre desempenho, área e vazão.

4.1 Smart card

As implementações de Simon 32/64 e Speck 32/64 foram realizadas em smart card na arquitetura Java Card, particularmente na família Gemalto ICP D72 FXR1, utilizando a plataforma Eclipse Mars 2 e Java Card Platform 2.2.2, utilizando a linguagem Java.

Em relação à utilização de memória, tem-se os dados representados na tabela 5. Estes valores foram retirados com o uso de comandos dentro do cartão, disponibilizados pelo Java Card.

Tabela 5 - Resultados da implementação em smart card dos algoritmos Simon e Speck

Cifras	Tamanho da Função (bytes)	EEPROM (bytes)	RAM (bytes)
Simon	10625,125	9068	198
Speck	8977,125	8830	25

Fonte: Elaborado pelo próprio autor

O algoritmo Simon ocupa 10625,125 bytes na memória EEPROM do cartão, onde este valor representa apenas o tamanho da função carregada na memória. Para utilização nas operações da função, são utilizados 9068 bytes da memória EEPROM e 198 bytes da memória RAM.

Logo a cifra Speck ocupa 8977,125 bytes na memória EEPROM do cartão, onde este valor representa apenas o tamanho da função carregada na memória. Para utilização nas operações da função, são utilizados 8830 bytes da memória EEPROM e 25 bytes da memória RAM.

4.2 Microcontrolador

As implementações de Simon 64/128 e Speck 64/128 foram realizadas em microcontrolador, particularmente na família PIC32MX795F512L, utilizando MPLAB X, utilizando a linguagem C.

Na função de rodada de Simon são realizados 29 ciclos para execução de uma rodada. Para cifrar os blocos são necessárias 44 rodadas, logo levará cerca de 29x44 = 1276 ciclos. A quantidade de ciclos será dividida por 1 byte, para o cálculo do custo gerando o resultado 1276/byte = 159,5.

Na função de rodada de Speck são realizados 21 ciclos para execução de uma rodada. Para cifrar os blocos são necessárias 27 rodadas, logo levará cerca de 21x27 = 567 ciclos. A quantidade de ciclos será dividida por 1 byte, para o cálculo do custo gerando o resultado 567/byte = 70,875.

As implementações de Simon 64/128 e Speck 64/128 foram realizadas em microcontrolador, particularmente na família PIC32MX795F512L, utilizando MPLAB X, utilizando a linguagem C.

Simon utilizou 2512 bytes de ROM o que corresponde cerca de 0,5% da quantidade de ROM total. Quanto a RAM, foram utilizados 260 bytes, o que corresponde cerca de 0,2% da quantidade total.

Speck utilizou 2324 bytes de ROM o que corresponde cerca de 0,4% da quantidade de ROM total. Quanto a RAM, foi utilizado 0 bytes.

Os resultados obtidos como quantidades utilizadas de ROM, RAM e custo de Simon e Speck foram relacionados na tabela 6.

Tabela 6 - Resultados da implementação em microcontrolador PIC32 dos algoritmos Simon e Speck

Cifras	ROM (bytes)	RAM (bytes)	Custo (cyc/byte)	Vazão (Mb/s)
Simon	2512	260	159,5	5,979
Speck	2324	0	70,875	1,345

Fonte: Elaborado pelo próprio autor

4.3 FPGA

Nesta subseção são apresentadas as arquiteturas em que Simon e Speck foram implementados, suas especificações e resultados.

4.3.1 Arquiteturas implementadas

Os algoritmos implementados foram simulados em FPGA, particularmente na família Xilinx Virtex-7 XC7VX330T, utilizando VIVADO 2016.3, utilizando a linguagem de descrição VHDL. As arquiteturas foram desenvolvidas e publicadas em "*Análise de Metodologias de Implementação e Desempenho em FPGA dos Algoritmos Criptográficos Leves Simon, Speck e Simeck, 2017*" e foram categorizadas em: Sync Single Rounds (COSTA, C. et al, 2017), Sync Full Rounds (COSTA, C. et al, 2017) e Async Full Rounds (COSTA, C. et al, 2017).

Sync Single Rounds (SSR): Na arquitetura síncrona composta por sinais, os bits são atualizados no final de cada iteração (*clock*), resultando no aumento significativo de iterações, porém há ganho em relação à área utilizada dentro do circuito e a frequência de operação.

Sync Full Rounds (SFR): Na arquitetura síncrona composta por variáveis, são utilizados laços repetitivos (*for*), realizando apenas uma única iteração do circuito (*clock*) para gerar as chaves e operações de cifrar ou decifrar. No entanto, esse tipo de metodologia consome uma maior área do circuito digital.

Async Full Rounds (AFR): A arquitetura assíncrona, descrita com circuitos combinacionais, ocorre a associação de portas lógicas e suas operações podem ser especificadas por meio de um conjunto de equações booleanas. O tempo de iteração

de cara round está associado diretamente ao tempo de atraso de propagação dos valores entre as entradas das portas lógicas até a saída.

Vazão denota a arquitetura com melhor desempenho em relação ao volume de informações cifradas por segundo. Para a avaliação da vazão de cada arquitetura em ambos os algoritmos, é realizado o cálculo a partir da divisão do tempo base (1 segundo), pelo período de execução que é o tempo de propagação da lógica do circuito implementado, multiplicado pela quantidade iterações necessárias para cada implementação e por fim multiplicado pelo tamanho da palavra (64 bits). Esse cálculo é representado na equação 4.1.

$$Vazão = \frac{1}{Período * QtdeIterações} * 64 \quad (4.1)$$

A relação área/vazão indica o melhor custo benéfico entre área e vazão (equação 4.2). A variável área é situada pela a quantidade de Slice LUTs consumida por cada implementação.

$$VA = \frac{Área}{Vazão} \quad (4.2)$$

4.3.2 Resultados algoritmo Simon

As implementações das arquiteturas no algoritmo Simon 64/128 foram relacionadas na tabela 7.

Tabela 7 - Análise de área e desempenho de Simon

Simon	Período (ns)	Área (Slice LUTs)	Vazão (Mb/s)	Área/Vazão
SFR	43,617	4224	1467	2,88
SSR	10,335	2293	69	33,23
AFR	29,802	1408	2147	0,65

Fonte: Elaborado pelo próprio autor

Em relação a área, destaca-se a arquitetura AFR, que obteve o menor consumo, com 1408 Slices LUTs. Quando comparado com as arquiteturas SFR e SSR houve ganho de 200% e 64,85%, respectivamente.

Com relação a vazão, a arquitetura que se destacou foi a AFR com 2147 Mb/s, sendo superior 46,35% e 3011,59% quando comparada com as arquiteturas SFR e SSR respectivamente.

Em relação a área/vazão, a arquitetura com melhor resultado é a AFR, sendo 343,07% e 5012,30% mais vantajosa que as arquiteturas SFR e SSR respectivamente

4.3.3 Resultados algoritmo Speck

As implementações das arquiteturas no algoritmo Speck 64/128 foram relacionados na tabela 8.

Tabela 8 - Análise de área e desempenho de Speck

Speck	Período (ns)	Área (Slice LUTs)	Vazão (Mb/s)	Área/Vazão
SFR	74,148	4861	863	5,63
SSR	4,067	1789	286	6,25
AFR	37,638	1056	1700	0,62

Fonte: Elaborado pelo próprio autor

Em relação a área, destaca-se a arquitetura AFR, que obteve o menor consumo, com 1056 Slices LUTs. Quando comparado com as arquiteturas SFR e SSR tem-se um ganho de 360,32% e 69,41%, respectivamente.

Com relação a vazão, a arquitetura que se destacou foi a AFR com 1700 Mb/s, sendo superior 96,98% e 494,40% quando comparada com as arquiteturas SFR e SSR respectivamente.

Em relação a área/vazão, a arquitetura com melhor resultado é novamente a AFR, sendo 808,06% e 908,06% mais vantajosa que as arquiteturas SFR e SSR respectivamente.

4.4 GPGPU

As implementações de Simon 64/128 e Speck 64/128 foram realizadas em GPGPU, particularmente na família Nvidia GTX 970, utilizando CUDA Toolkit, utilizando a linguagem C.

As cifras de bloco leve Simon e Speck foram executadas 100 vezes, a partir disso foram calculados a média e o desvio padrão dos tempos de upload, download e tempo de execução do *kernel*. Do tempo de execução do *kernel*, foi possível estimar a vazão de Simon e Speck. Para melhor visualização, os dados dispostos em uma tabela, o primeiro de Simon (Tabela 9) e segundo de Speck (Tabela 10).

Tabela 9 - Tempo de execução de Simon em GPGPU

Média de tempo de upload (us)	**Média de tempo de download (us)**	**Média de tempo de execução (us)**	**Vazão (Mb/s)**
12,56188	11,76224	6432,036	0,006432036
Desvio Padrão	**Desvio Padrão**	**Desvio Padrão**	
0,0987503	0,060554528	79,47045405	

Fonte: Elaborado pelo próprio autor

As cifras Simon e Speck executaram aproximadamente 1 Mb de texto plano em 1 bloco com tamanho de 1024 threads. Foram cifrados 131072 blocos na GPGPU onde foi implementada apenas a função de cifrar e as sub-chaves foram geradas na CPU.

Tabela 10 - Tempo de execução de Speck em GPGPU

Média de tempo de upload (us)	**Média de tempo de download (us)**	**Média de tempo de execução (us)**	**Vazão (Mb/s)**
12,58079	11,76416	1231,265	0,001231265
Desvio Padrão	**Desvio Padrão**	**Desvio Padrão**	
0,295206455	0,068474683	8,971980795	

Fonte: Elaborado pelo próprio autor

4.5 Comparação com trabalhos correlatos

Na comparação com artigos correlatos, a intenção é comparar o desempenho das criptografias leves entre FPGAs. A tabela 11 apresenta os resultados atingidos pelos trabalhos correlatos e por este trabalho para as implementações em FPGA de Simon e Speck.

Tabela 11 - Comparação da vazão de Simon e Speck com trabalhos correlatos

Autor	**Algoritmo**	**FPGA**	**Vazão(Mb/s)**
BEAULIEU et al (2015)	Simon	Spartan-3	512
BEAULIEU et al (2015)	Speck	Spartan-3	416
Shylaja e Shreekanth (2015)	Simon	Spartan-3	392.4
AFR (2017)	Simon	Virtex-7	2147
AFR (2017)	Speck	Virtex-7	1700

Fonte: Elaborado pelo próprio autor

BEAULIEU, et al (BEAULIEU et al.,2015), são autores dos algoritmos Simon e Speck, como pode-se observar as implementações de AFR/Simon e AFR/Speck foram os melhores em relação a vazão entres os trabalhos correlatos apontados.

Na comparação com artigos correlatos, a intenção é comparar o custo das criptografias leves entre microcontroladores. A tabela 12 apresenta os melhores resultados atingidos pelos trabalhos correlatos e por este trabalho para as implementações em microcontrolador de Simon e Speck.

Tabela 12 - Comparação do custo de Simon e Speck com trabalhos correlatos

Autor	Algoritmo	Microcontrolador	Custo (cyc/byte)
BEAULIEU et al (2015)	Simon	MSP430	153
BEAULIEU et al (2015)	Speck	MSP430	89
BEAULIEU et al (2015)	Simon	ATmega128	221
BEAULIEU et al (2015)	Speck	ATmega128	122
TACHIBANA et al (2017)	Simon	PIC32	159,5
TACHIBANA et al (2017)	Speck	PIC32	70,875

Fonte: Elaborado pelo próprio autor

Como pode-se observar a implementação de Speck com o microcontrolador PIC32 teve o menor custo entre os trabalhos correlatos apontados. Já para a implementação de Simon, a referência BEAULIEU, et al (BEAULIEU et al.,2015) atingiu o melhor resultado.

Em relação ao smart card, Simon e Speck foram implementados na versão 32/64 devido as limitações de hardware. Não foi encontrado nenhum trabalho no estado da arte em que as cifras leves foram implementadas neste tipo de tecnologia.

Na plataforma GPGPU também não foi encontrado trabalhos correlatos da implementação de algoritmos criptográficos leves neste tipo de tecnologia.

Capítulo 5

CONCLUSÃO

Este trabalho compreendeu-se em implementar as cifras de bloco leve em quatro tecnologias diferentes: *smart card*, microcontrolador, FPGA e GPGPU. Teve como objetivo avaliar o comportamento desses algoritmos em dispositivos distintos, visto que a Internet das Coisas é uma rede heterogênea.

Primeiramente foi feito a pesquisa de trabalhos correlatos no estado da arte sobre cifras de blocos leves no contexto de Internet das Coisas, caracterizando-os de acordo com seus designs de implementação, para então ter parâmetros para a definição de quais algoritmos serão selecionadas para posteriormente serem desenvolvidos. A partir do documento publicado pelo NIST e pela NSA, foram selecionados os algoritmos Simon e Speck para serem implementados neste trabalho.

As implementações deste trabalho tiveram início com implementações na linguagem C para melhor compreensão dos designs dos algoritmos, posteriormente projetadas em arquiteturas específicas do microcontrolador e GPGPU, realizadas também depois de transcritos para as tecnologias FPGA e *smart card*.

Os objetivos deste trabalho foram alcançados com os resultados das implementações apresentados na tabela 13, onde foi possível avaliar o comportamento em relação a vazão final de cada cifra em três tecnologias.

Tabela 13 - Comparação entre tecnologias das implementações Simon e Speck

Tecnologia	**Cifra**	**Vazão (Mb/s)**
PIC32MX795F512L	Simon	5,979
PIC32MX795F512L	Speck	1,345

Tecnologia	Cifra	Vazão (Mb/s)
Virtex-7 XC7VX330T	Simon	2147
Virtex-7 XC7VX330T	Speck	1700
Nvidia GTX 970	Simon	0,006432036
Nvidia GTX 970	Speck	0,001231265

Fonte: Elaborado pelo próprio autor

Como pode-se observar entre as implementações, Simon e Speck obteve melhor desempenho na plataforma FPGA com vazão de 2147 Mb/s e 1700 Mb/s respectivamente.

Os resultados do *smart card* não foram relacionados na comparação devido ao software não fornecer dados sobre a frequência de operação.

Ainda assim, dependendo do cenário da Internet das Coisas onde será realizada a aplicação da solução, pode-se escolher entre a melhor implementação conforme os requisitos relacionados ao desempenho, área, consumo de memórias (ROM e RAM) e custo da aplicação.

5.1 Lições aprendidas

Este trabalho vem sendo desenvolvido há dois anos, onde implementações com FPGA na linguagem de descrição de hardware VHDL tiveram bastante ênfase. Então foi proposto a implementações em mais 3 tecnologias sendo elas: *smart card*, microcontrolador e GPGPU.

Houve o aprendizado das tecnologias *smart card* e GPGPU, por se tratar de plataformas que ainda não havia oportunidade de contato. A abrangência de tecnologias aprendidas, suas arquiteturas e implementações de algoritmos criptográficos, acrescentou positivamente para a minha experiência acadêmica.

5.2 Trabalhos Futuros

Para trabalhos futuros é proposto realizar estudos relacionados ao consumo de energia dessas implementações na plataforma FPGA, GPGPU, *smart card* e microcontrolador.

5.3 Agradecimentos

Agradeço à FAPESP e ao CNPq que fomentou minhas pesquisas durante a graduação. Agradeço também ao UNIVEM e COMPSI por ter oferecido todo o suporte para realização das iniciações científicas e publicações.

REFERÊNCIAS

AGENCY, N. S. **Algorithms to Support the Evolution of Information Assurance Needs**. In:. [S.l.]: NSA, 2016.

APPLICATIONS, S. S.; INNOVATION. **GPGPU (General Purpose Graphics Processing Unit)**. 2012. Disponível em:<http://www.hpc.cineca.it/content/gpgpu-general-purpose-graphics-processing-unit>.

BAENTSCH, M. et al. JavaCard—**From Hype to Reality**. In:. [S.l.]: IBM Zurich Research Laboratory and IEEE, 1999.

BANNATYNE, R.; VIOT, G. **Introduction to Microcontrollers - Part 1**. In: Transp. Syst. Group, Motorola Inc., USA. [S.l.]: IEEE, 1998.

BAYSAL, A.; SAHIN, S. **RoadRunneR: A Small And Fast Bitslice Block Cipher For Low Cost 8-bit Processors**. In: LightSec 2015. [S.l.: s.n.], 2015.

BEAULIEU, R. et al. **The Simon and Speck Families of Lightweight Block Ciphers**. Cryptology ePrint Archive, 2013.

BEAULIEU, R., SHORS, D., SMITH, J., TREATMAN-CLARK, S., WEEKS, B., & WINGERS, L. (2015). **Simon and Speck: Block Biphers for the Internet Of Things**. Iacr cryptology eprint archive, 2015, 585.

BOGDANOV, A. et al. **PRESENT: An ultra-lightweight block cipher**. In:. [S.l.]: In Cryptographic Hardware and Embedded Systems-CHES 2007 - Springer Berlin Heidelberg, 2007. p. 450–466.

COMPUTERWORLD. **Intel anuncia a conclusão da compra da Altera por US$ 16,7 bilhões**. 2015. Disponível em:< http://computerworld.com.br/intel-anuncia-conclusao-da-compra-da-altera-por-us-167-bilhoes>.

COSTA, C., TACHIBANA, F. M. O., MORENO, E. D. and PEREIRA, F. D. "**Análise De Metodologias De Implementação e Desempenho em FPGA dos Algoritmos Criptográficos Leves Simon, Speck e Simeck."** Lascas - Prime – Iberchip, 2017

CLUA, E. G. W.; ZAMITH, M. **Programming in CUDA for Kepler and Maxwell Architecture**. Revista de Informática Teórica e Aplicada - seer ufrgs. Volume 22, Número 2, 2015.

DINU, D. et al. **Design Strategies for ARX with Provable Bounds: SPARX and LAX**. In:. [S.l.]: In Advances in Cryptology–ASIACRYPT 2016 - Springer Berlin Heidelberg, 2016. p. 484–513.

FARIAS, L.; SEGUNDO, M.; BATISTA, A. **Multicore**. In:. [S.l.]: Universidade Federal Rural de Pernambuco - UFRPE, 2015.

FAROOQ, U.; MARRAKCHI, Z.; MEHREZ, H. **FPGA Architectures: An Overview. In: Tree-based Heterogeneous FPGA Architectures**. [S.l.]: Springer-Verlag New York, 2012. p. 7–48. ISBN 978-1-4614-3594-5.

FATAHALIAN, K.; HOUSTON, M. **A closer look at GPUs**. In:. [S.l.]: Communications of the ACM, 2008

FORBES. **A Simple Explanation Of 'The Internet Of Things**. 2014. Disponível em:<http://www.forbes.com/sites/jacobmorgan/2014/05/13/ simple-explanation-internet-things-that-anyone-can-understand/>.

GRIDLING et al. **Introduction to Microcontrollers**. In: Courses 182.064 182.074. [S.l.]: Vienna University of Technology, Institute of Computer Engineering, Embedded Computing Systems Group, 2007.

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

HANDBOOK of Applied Cryptography. 2001. Disponível em:<www.cacr.math.uwaterloo.ca/hac/>.

HONG, D. et al. **LEA: A 128-bit block cipher for fast encryption on common processors**. In: . [S.l.]: Information Security Applications - Springer International Publishing, 2014. p. 3–27.

HONG, D. et al. **Hight: A new block cipher suitable for low-resource device**. In: . [S.l.]: In Cryptographic Hardware and Embedded Systems-CHES 2006 - Springer Berlin Heidelberg, 2006. p. 46–59.

LIMA, T. **XILINX está trabalhando em tecnologia de 7nm para FPGAs**. 2015. Disponível em:< https://www.embarcados.com.br/xilinx-tecnologia-7nm-fpgas/>.

LIMA, T. **Microchip anuncia compra da Atmel**. 2016. Disponível em:< https://www.embarcados.com.br/microchip-compra-atmel/>.

MCKAY, K. A. et al. **Report on Lightweight Cryptography**. In: Computer Security Division Information Technology Laboratory. [S.l.]: NIST, 2016.

MOUHA, N. et al. **Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers**. In: In Selected Areas in Cryptography – SAC 2014. [S.l.]: Springer International Publishing, 2014. p. 306–323.

ORTIZ, C. E. **An Introduction to Java Card Technology - Part 1**. 2013. Disponível em: <http://www.oracle.com/technetwork/java/javacard/javacard1-139251.html>.

PARK, MOO KYU, AND JI WON YOON. "**Optimization Of Lightweight Encryption Algorithm (Lea) Using Threads And Shared Memory Of Gpu."** Journal Of The Korea Institute Of Information Security And Cryptology 25.4 (2015): 719-726.

POSCHMANN, A.; LING, S.; WANG, H. 2**56 bit standardized crypto for 650 GE – GOST revisited.** In:. [S.l.]: In Cryptographic Hardware and Embedded Systems-CHES 2010 - Springer Berlin Heidelberg, 2010. p. 219–233.

QIU, WEIDONG, ET AL. **"Protecting Lightweight Block Cipher Implementation In Mobile Big Data Computing."** Peer-To-Peer Networking And Applications (2016): 1-13.

SHIBUTANI, K. et al. **Piccolo: an ultra-lightweight blockcipher**. In:. [S.l.]: 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:. [S.l.]: In Fast software encryption - Springer Berlin Heidelberg, 2007. p. 181–195.

SHYLAJA, C., & SHREEKANTH, T. **Optimization Of Block Cipher With Simon**. International Journal Of Computer Applications (0975 – 8887), 2015.

SMITH, R. **The NVIDIA GeForce GTX 980 Review: Maxwell Mark 2**. September 18, 2014. Disponível em: <https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/3>

SOLLER, S. **GPGPU origins and GPU hardware architecture**. In:. [S.l.]: Stuttgart Media University, 2011.

SUZAKI, T. et al. **Twine: A lightweight, versatile block cipher**. In:. [S.l.]: ECRYPT Workshop on Lightweight Cryptography, 2011. p. 146–169.

WEBER, R. H. **Internet of Things – New security and privacy challenges. Computer law security review** 26. In:. [S.l.: s.n.], 2010. p. 23–30.

Yang, G. Et Al. **The Simeck Family Of Lightweight Block Ciphers.** In: Cryptographic Hardware And Embedded Systems - Ches. [S.L.: S.N.], 2015. V. 9293, P. 307—329.

Apêndice A

IMPLEMENTAÇÃO APPLET SIMON 32/64

```
package simon.jc;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;
import javacard.framework.JCSystem;


public class Simon extends Applet {

    protected Simon()
    {
        register();
    }

        public static void install(byte[] bArray, short bOffset,
byte bLength)
    {
        new Simon();
    }

        public void process(APDU apdu)
    {
     byte[] z =
                {1,1,1,1,1,0,1,0,0,0,1,0,0,1,0,1,0,1,1,0,0,0,0,1,1,1
                ,0,0,1,1,0,1,1,1,1,1,0,1,0,0,0,1,0,0,1,0,1,0,1,1,0,0
                ,0,0,1,1,1,0,0,1,1,0};
        short n = 16;
        short m = 4;
        short rounds = 32;
        short i;
        short left;
        short right;
        short[] plaintext = {0x2064, 0x656b};
        short[] key = new short[rounds];
        short auxkey;
        short auxenc;

        key[3] = 0x0302;
```

```
           key[2] = 0x0b0a;
           key[1] = 0x1312;
           key[0] = 0x1b1a;
        //key expansion
        for (i = m; i < rounds; i++){
            auxkey = RotateRight(key[(short) (i-1)], (short)3);
            if (m == 4){
                auxkey ^= key[(short)(i-3)];            }
            auxkey ^= RotateRight(auxkey,(short)1);
            key[i] = (short)(~key[(short)(i-m)] ^ auxkey ^
z[(short)(i-m)] ^ (short)3);
        }
        //encrypt
           left = plaintext[0];
           right = plaintext[1];
           for (i = 0; i < rounds; i++){
            auxenc = left;
            left = (short) (right ^ (RotateLeft(left,(short)1) &
RotateLeft(left,(short)8)) ^ RotateLeft(left,(short)2) ^ key[i]);
            right = auxenc;
           }
        //decrypt
           for (i = 0; i < rounds; i++){
            auxenc = right;
            right = (short) (left ^ (RotateLeft(right,(short)1) &
RotateLeft(right,(short)8)) ^ RotateLeft(right,(short)2) ^
key[(short)((rounds-1)-i)]);
            left = auxenc;
           }
    }
           public short RotateRight(short x, short bits){
               short result = (short) ((x << bits) | (x >> (16-
bits)));
               return result;
           }
           public short RotateLeft(short x, short bits){
               short result = (short)((x >> bits) | (x << (16-
bits)));
               return (short) result;
           }

}
```

Apêndice B

IMPLEMENTAÇÃO APPLET SPECK 32/64

```
package speck.jc;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;

public class Speck extends Applet {

    protected Speck()
    {

        register();
    }

     public static void install(byte bArray[], short bOffset, byte
bLength) throws ISOException {
           new Speck();
     }

     public void process(APDU apdu) throws ISOException {
           short n = 16;
        short m = 4;
        short rounds = 22;
        short i;
           short left;
        short right;
        short[] plaintext = {0x2064, 0x656b};
        short[] key = new short[rounds];
        short[] l = new short[rounds];
        short auxenc;

        l[2] = 0x0302;
           l[1] = 0x0b0a;
           l[0] = 0x1312;
           key[0] = 0x1b1a;
      //key expansion
        for (i = 0; i <= (short)(rounds - 2); i++){
           l[(short)(i+m-1)] = (short)((key[i] + RotateRight(l[i],
(short) 7)) ^ (short) i);
```

```
            key[(short)(i+1)] = (short)(RotateLeft(key[i],(short)2) ^
l[(short)(i+m-1)]);
      }
       //encrypt
      left = plaintext[0];
      right = plaintext[1];
      for (i = 0; i < rounds; i++){
            right = (short)((RotateRight(right,(short)7) + left) ^
key[i]);
            left = (short)(RotateLeft(left,(short)2) ^ right);
      }
      //decrypt
      for (i = 0; i < rounds; i++){
            auxenc = (short)(left ^ right);
            left = (short)(RotateRight(auxenc,(short)2));

            auxenc = (short)((right ^ key[(short)((rounds-1)-i)]) -
left);
            right = (short)(RotateLeft(auxenc,(short)7));
      }


      }
       public short RotateRight(short x, short bits){
                short result = (short) ((x << bits) | (x >> (16-
bits)));
                return result;
            }
            public short RotateLeft(short x, short bits){
                short result = (short)((x >> bits) | (x << (16-
bits)));
                return (short) result;
            }

}
```

Apêndice C

IMPLEMENTAÇÃO PIC32 SIMON 64/128

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#define ROTATE_LEFT_64(x,bits) ( (x << bits) | (x >> (32-bits)) )
#define ROTATE_RIGHT_64(x,bits) ( (x >> bits) | (x << (32-bits)))

main(){
      static uint8_t z[62] =
      {1,1,0,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,0,1,1,1,1,0,0,0,0,
0,0,1,0,0,1,0,0,0,1,0,1,0,0,1,1,1,0,0,1,1,0,1,0,0,0,0,1,1,1,1};

      static uint8_t n = 32;
      static uint8_t m = 4;
      static uint8_t rounds = 44;

      uint32_t plaintext[2], left, right;
      uint32_t key[rounds];
      uint32_t auxkey = 0;
      uint32_t auxenc = 0;

      int i = 0;

      plaintext[1] = 0x20646e75;
      plaintext[0] = 0x656b696c;

      key[3] = 0x03020100;
      key[2] = 0x0b0a0908;
      key[1] = 0x13121110;
      key[0] = 0x1b1a1918;
      //key expansion
      for (i = m; i < rounds; i++){
            auxkey = ROTATE_RIGHT_64(key[i-1],3);
            if (m == 4){
                  auxkey ^= key[i-3];
            }
            auxkey ^= ROTATE_RIGHT_64(auxkey,1);
            key[i] = ~key[i-m] ^ auxkey ^ z[(i-m) % 62] ^ 3;
      }
      //encrypt
      left = plaintext[0];
      right = plaintext[1];
```

```
      for (i = 0; i < rounds; i++){
            auxenc = left;
            left = right ^ (ROTATE_LEFT_64(left,1) &
ROTATE_LEFT_64(left,8)) ^ ROTATE_LEFT_64(left,2) ^ key[i];
            right = auxenc;
      }
      //decrypt
      for (i = 0; i < rounds; i++){
            auxenc = right;
            right = left ^ (ROTATE_LEFT_64(right,1) &
ROTATE_LEFT_64(right,8)) ^ ROTATE_LEFT_64(right,2) ^ key[(rounds-1)-
i];
            left = auxenc;
      }
}
```

Apêndice D

IMPLEMENTAÇÃO PIC32 SPECK 64/128

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#define ROTATE_LEFT_64(x,bits) ( (x << bits) | (x >> (64-bits)) )
#define ROTATE_RIGHT_64(x,bits) ( (x >> bits) | (x << (64-bits)))

main(){
      static uint8_t n = 32;
      static uint8_t m = 4;
      static uint8_t rounds = 27;

      int i = 0;

      uint32_t plaintext[2], left, right;
      uint32_t key[rounds];
      uint32_t l[rounds];
      uint32_t auxenc, tmp;

      plaintext[1] = 0x7475432d;
      plaintext[0] = 0x3b726574 ;

      l[2] = 0x03020100;
      l[1] = 0x0b0a0908;
      l[0] = 0x13121110;
      key[0] = 0x1b1a1918;

      //key expansion
      for (i = 0; i <= rounds - 2; i++){
            l[i+m-1] = (key[i] + ROTATE_RIGHT_64(l[i],8)) ^ i;
            key[i+1] = ROTATE_LEFT_64(key[i],3) ^ l[i+m-1];
      }
      //encrypt
      left = plaintext[0];
      right = plaintext[1];
      for (i = 0; i < rounds; i++){
            right = (ROTATE_RIGHT_64(right,8) + left) ^ key[i];
            left = ROTATE_LEFT_64(left,3) ^ right;
      }

      //decrypt
      for (i = 0; i < rounds; i++){
```

```
            auxenc = left ^ right;
            left = ROTATE_RIGHT_64(auxenc,3);

            auxenc = (right ^ key[(rounds-1)-i]) - left;
            right = ROTATE_LEFT_64(auxenc,8);
      }
}
```

Apêndice E

IMPLEMENTAÇÃO FPGA SSR SIMON 64/128

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity Simon_variable is
     Port( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0);
           key   : in STD_LOGIC_VECTOR (127 downto 0));
end Simon_variable;

architecture Behavioral of Simon_variable is
constant n : integer := 32;
constant m : integer := 4;
constant Z3 : STD_LOGIC_VECTOR(61 downto 0) :=
"11011011101011000110010111100000010010001010011100110100001111";
constant rounds : integer := 44;

type MEM_ROM is array (0 to 44) of STD_LOGIC_VECTOR (31 downto 0);

begin
process (clock, reset)

--Variaveis roundkey
 variable ky_tmp : STD_LOGIC_VECTOR (31 downto 0);
 variable ky_aux : STD_LOGIC_VECTOR (31 downto 0);
 variable ky_S1  : STD_LOGIC_VECTOR (31 downto 0);
 variable ky_S3  : STD_LOGIC_VECTOR (31 downto 0);
 variable ky : MEM_ROM;

--Variaveis rounds
 variable S1  : STD_LOGIC_VECTOR (31 downto 0);
 variable S2  : STD_LOGIC_VECTOR (31 downto 0);
 variable S8  : STD_LOGIC_VECTOR (31 downto 0);
 variable tmp : STD_LOGIC_VECTOR (31 downto 0);
 variable left : STD_LOGIC_VECTOR(31 downto 0);
 variable right : STD_LOGIC_VECTOR(31 downto 0);

begin

--RESET
```

```
 if reset = '1' then
    for i in 0 to 31
    loop
       ky(i) := (others => '0');
    end loop;
    dout <= (others => '0');
 elsif rising_edge (clock) then

--KEY EXPANSION
    ky(0) := key(31 downto 0);
    ky(1) := key(63 downto 32);
    ky(2) := key(95 downto 64);
    ky(3) := key(127 downto 96);
    left := din(63 downto 32);
    right := din(31 downto 0);

    for i in m to (rounds-1)
       loop
          ky_S3 := ky(i-1)(2 downto 0) & ky(i-1)(31 downto 3);--
S(?3) de k[i-1]
          ky_tmp := ky_S3;--S(?3) de k[i-1]
          if m = 4 then
             ky_aux := ky_tmp;
             ky_tmp := ky_aux XOR ky(i-3);
          end if;
          ky_aux := ky_tmp;
          ky_S1 := ky_tmp(0) & ky_tmp(31 downto 1);
          ky_tmp :=  ky_aux xor ky_S1;

          ky(i) := (not ky(i-M)) xor ky_tmp xor
"00000000000000000000000000000000"&Z3(((i-m) mod 62)) xor
"00000000000000000000000000000011";
       end loop;

--Encrypt
    if ed = '0' then
         for j in 0 to 44
         loop
            S1 := left(30 downto 0) & left(31);
            S8 := left(7 downto 0) & left(31 downto 8);
            S2 := left(29 downto 0) & left(31 downto 30);

            tmp := left;
            left := right XOR (S1 AND S8) XOR S2 XOR ky(j);
            right := tmp;
         end loop;
       dout <= left & right;

--Decrypt
    elsif ed = '1' then
           for z in 44 downto 0
           loop
              S1 := right(30 downto 0) & right(31);
              S8 := right(7 downto 0)  & right(31 downto 8);
              S2 := right(29 downto 0) & right(31 downto 30);
```

```
            tmp := right;
            right := left xor (S1 and S8) xor S2 xor ky(z);
            left := tmp;
         end loop;
         dout <= left & right;
      end if;
 end if;
end process;
end Behavioral;
```

Apêndice F

IMPLEMENTAÇÃO FPGA SFR SIMON 64/128

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity Simon32 is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0);
           key   : in STD_LOGIC_VECTOR (127 downto 0));
end Simon32;

architecture Behavioral of Simon32 is
constant n : integer := 32;
constant m : integer := 4;
constant Z3 : STD_LOGIC_VECTOR(61 downto 0) :=
"11011011101011000110010111100000010010001010011100110100001111";
constant rounds : integer := 44;

type MEM_ROM is array (0 to 44) of STD_LOGIC_VECTOR (31 downto 0);

type STATUS is (DISPONIVEL, WORK, ESPERA);
signal ESTADO : STATUS;
signal skey : STD_LOGIC_VECTOR(63 downto 0);
signal ky : MEM_ROM;
signal left : STD_LOGIC_VECTOR(31 downto 0);
signal right : STD_LOGIC_VECTOR(31 downto 0);
signal count : integer range 0 to 44;
signal kcount : integer range 0 to 44;

begin
process (clock, reset)
 variable k_tmp : STD_LOGIC_VECTOR (31 downto 0);
 variable L : STD_LOGIC_VECTOR (31 downto 0);
 variable R : STD_LOGIC_VECTOR (31 downto 0);
begin

--reset
if reset = '1' then
    ky(0) <= (others => '0');
    ky(1) <= (others => '0');
    ky(2) <= (others => '0');
```

```
    ky(3) <= (others => '0');
    left <= (others => '0');
    right <= (others => '0');
    count <= 0;
    kcount<= 4;
    dout <= (others => '0');
    ESTADO <= DISPONIVEL;

--begin
elsif rising_edge (clock) then
   case ESTADO is
   when DISPONIVEL =>
           ky(0) := key(31 downto 0);
           ky(1) := key(63 downto 32);
           ky(2) := key(95 downto 64);
           ky(3) := key(127 downto 96);
      left <= din(63 downto 32);
      right <= din(31 downto 0);
      if ed = '0' then
         count <= 0;
      elsif ed = '1' then
         count <= 31;
      end if;
      kcount<= m;
      dout <= (others => '0');
      ESTADO <= WORK;
   when WORK =>

--keyexpansion
    if (kcount <= 63) then
        k_tmp := ky(kcount-1)(2 downto 0) & ky(kcount-1)(31 downto
3);
        if m=4 then
           k_tmp := k_tmp XOR ky(kcount-3);
        end if;
        k_tmp :=  k_tmp XOR (k_tmp(0) & k_tmp(31 downto 1));

        ky(kcount) <= (not ky(kcount-M)) XOR k_tmp XOR
"00000000000000000000000000000000"&Z3(((kcount-m) mod 62)) xor
"00000000000000000000000000000011";
           kcount <= kcount +1;
     end if;

--encrypt
    if ed = '0' then
        if (count <= 43) then
           L := left;
           R := right;
           left <= right XOR ((left(30 downto 0) & left(31)) AND
(left(7 downto 0) & left(31 downto 8))) XOR (left(29 downto 0) &
left(31 downto 30)) XOR ky(count);
           right <= L;
           --dout <= X & Y;
           if (count < 43) then
              count <= count + 1;
           elsif (count = 43) then
```

```
              ESTADO <= ESPERA;
            end if;
         end if;
    elsif ed = '1' then
              if (kcount = 44) then
                 R := right;
                 right <= left XOR ((right(30 downto 0) & right(31))
AND (right(7 downto 0)  & right(31 downto 8))) XOR (right(29 downto
0) & right(31 downto 30)) XOR ky(count);
                 left <= R;
                 if (count > 0) then
                  count <= count - 1;
                 elsif (count = 0) then
                  ESTADO <= ESPERA;
                 end if;
              end if;
           end if;
       when ESPERA =>
           kcount <= m;
           count <= 0;
           dout <= left & right;
           ESTADO <= DISPONIVEL;
       when others =>
           ESTADO <= DISPONIVEL;
       end case;
     end if;
    end process;
    end Behavioral;
```

Apêndice G

IMPLEMENTAÇÃO FPGA AFR SIMON 64/128

```
entity Simon_asc is
 Port ( --ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0)
           --key   : in STD_LOGIC_VECTOR (127 downto 0)
        );
end Simon_asc;

architecture Behavioral of Simon_asc is

--signals of encryption
signal leftEnc : STD_LOGIC_VECTOR(31 downto 0);
signal left0 : STD_LOGIC_VECTOR(31 downto 0);
signal left1 : STD_LOGIC_VECTOR(31 downto 0);
signal left2 : STD_LOGIC_VECTOR(31 downto 0);
signal left3 : STD_LOGIC_VECTOR(31 downto 0);
signal left4 : STD_LOGIC_VECTOR(31 downto 0);
signal left5 : STD_LOGIC_VECTOR(31 downto 0);
signal left6 : STD_LOGIC_VECTOR(31 downto 0);
signal left7 : STD_LOGIC_VECTOR(31 downto 0);
signal left8 : STD_LOGIC_VECTOR(31 downto 0);
signal left9 : STD_LOGIC_VECTOR(31 downto 0);
signal left10 : STD_LOGIC_VECTOR(31 downto 0);
signal left11 : STD_LOGIC_VECTOR(31 downto 0);
signal left12 : STD_LOGIC_VECTOR(31 downto 0);
signal left13 : STD_LOGIC_VECTOR(31 downto 0);
signal left14 : STD_LOGIC_VECTOR(31 downto 0);
signal left15 : STD_LOGIC_VECTOR(31 downto 0);
signal left16 : STD_LOGIC_VECTOR(31 downto 0);
signal left17 : STD_LOGIC_VECTOR(31 downto 0);
signal left18 : STD_LOGIC_VECTOR(31 downto 0);
signal left19 : STD_LOGIC_VECTOR(31 downto 0);
signal left20 : STD_LOGIC_VECTOR(31 downto 0);
signal left21 : STD_LOGIC_VECTOR(31 downto 0);
signal left22 : STD_LOGIC_VECTOR(31 downto 0);
signal left23 : STD_LOGIC_VECTOR(31 downto 0);
signal left24 : STD_LOGIC_VECTOR(31 downto 0);
signal left25 : STD_LOGIC_VECTOR(31 downto 0);
signal left26 : STD_LOGIC_VECTOR(31 downto 0);
signal left27 : STD_LOGIC_VECTOR(31 downto 0);
signal left28 : STD_LOGIC_VECTOR(31 downto 0);
signal left29 : STD_LOGIC_VECTOR(31 downto 0);
signal left30 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal left31 : STD_LOGIC_VECTOR(31 downto 0);
signal left32 : STD_LOGIC_VECTOR(31 downto 0);
signal left33 : STD_LOGIC_VECTOR(31 downto 0);
signal left34 : STD_LOGIC_VECTOR(31 downto 0);
signal left35 : STD_LOGIC_VECTOR(31 downto 0);
signal left36 : STD_LOGIC_VECTOR(31 downto 0);
signal left37 : STD_LOGIC_VECTOR(31 downto 0);
signal left38 : STD_LOGIC_VECTOR(31 downto 0);
signal left39 : STD_LOGIC_VECTOR(31 downto 0);
signal left40 : STD_LOGIC_VECTOR(31 downto 0);
signal left41 : STD_LOGIC_VECTOR(31 downto 0);
signal left42 : STD_LOGIC_VECTOR(31 downto 0);
signal left43 : STD_LOGIC_VECTOR(31 downto 0);

signal rightEnc : STD_LOGIC_VECTOR(31 downto 0);
signal right0 : STD_LOGIC_VECTOR(31 downto 0);
signal right1 : STD_LOGIC_VECTOR(31 downto 0);
signal right2 : STD_LOGIC_VECTOR(31 downto 0);
signal right3 : STD_LOGIC_VECTOR(31 downto 0);
signal right4 : STD_LOGIC_VECTOR(31 downto 0);
signal right5 : STD_LOGIC_VECTOR(31 downto 0);
signal right6 : STD_LOGIC_VECTOR(31 downto 0);
signal right7 : STD_LOGIC_VECTOR(31 downto 0);
signal right8 : STD_LOGIC_VECTOR(31 downto 0);
signal right9 : STD_LOGIC_VECTOR(31 downto 0);
signal right10 : STD_LOGIC_VECTOR(31 downto 0);
signal right11 : STD_LOGIC_VECTOR(31 downto 0);
signal right12 : STD_LOGIC_VECTOR(31 downto 0);
signal right13 : STD_LOGIC_VECTOR(31 downto 0);
signal right14 : STD_LOGIC_VECTOR(31 downto 0);
signal right15 : STD_LOGIC_VECTOR(31 downto 0);
signal right16 : STD_LOGIC_VECTOR(31 downto 0);
signal right17 : STD_LOGIC_VECTOR(31 downto 0);
signal right18 : STD_LOGIC_VECTOR(31 downto 0);
signal right19 : STD_LOGIC_VECTOR(31 downto 0);
signal right20 : STD_LOGIC_VECTOR(31 downto 0);
signal right21 : STD_LOGIC_VECTOR(31 downto 0);
signal right22 : STD_LOGIC_VECTOR(31 downto 0);
signal right23 : STD_LOGIC_VECTOR(31 downto 0);
signal right24 : STD_LOGIC_VECTOR(31 downto 0);
signal right25 : STD_LOGIC_VECTOR(31 downto 0);
signal right26 : STD_LOGIC_VECTOR(31 downto 0);
signal right27 : STD_LOGIC_VECTOR(31 downto 0);
signal right28 : STD_LOGIC_VECTOR(31 downto 0);
signal right29 : STD_LOGIC_VECTOR(31 downto 0);
signal right30 : STD_LOGIC_VECTOR(31 downto 0);
signal right31 : STD_LOGIC_VECTOR(31 downto 0);
signal right32 : STD_LOGIC_VECTOR(31 downto 0);
signal right33 : STD_LOGIC_VECTOR(31 downto 0);
signal right34 : STD_LOGIC_VECTOR(31 downto 0);
signal right35 : STD_LOGIC_VECTOR(31 downto 0);
signal right36 : STD_LOGIC_VECTOR(31 downto 0);
signal right37 : STD_LOGIC_VECTOR(31 downto 0);
signal right38 : STD_LOGIC_VECTOR(31 downto 0);
signal right39 : STD_LOGIC_VECTOR(31 downto 0);
signal right40 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal right41 : STD_LOGIC_VECTOR(31 downto 0);
signal right42 : STD_LOGIC_VECTOR(31 downto 0);
signal right43 : STD_LOGIC_VECTOR(31 downto 0);

signal tmp0  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp1  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp2  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp3  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp4  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp5  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp6  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp7  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp8  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp9  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp10  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp11  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp12  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp13  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp14  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp15  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp16  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp17  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp18  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp19  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp20  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp21  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp22  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp23  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp24  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp25  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp26  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp27  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp28  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp29  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp30  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp31  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp32  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp33  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp34  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp35  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp36  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp37  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp38  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp39  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp40  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp41  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp42  : STD_LOGIC_VECTOR(31 downto 0);
signal tmp43  : STD_LOGIC_VECTOR(31 downto 0);
signal cipher : STD_LOGIC_VECTOR(63 downto 0);

begin

--encrypt
    leftEnc <= din(63 downto 32);
    rightEnc <= din(31 downto 0);
```

```
    --round 0
       tmp0 <= leftEnc;
       left0 <= rightEnc XOR ((leftEnc(30 downto 0) & leftEnc(31))
AND (leftEnc(7 downto 0) & leftEnc(31 downto 8))) XOR (leftEnc(29
downto 0) & leftEnc(31 downto 30)) XOR
"00000000000000000000000000000000";
       right0 <= tmp0;
    --round 1
       tmp1 <= left0;
       left1 <= right0 XOR ((left0(30 downto 0) & left0(31)) AND
(left0(7 downto 0) & left0(31 downto 8))) XOR (left0(29 downto 0) &
left0(31 downto 30)) XOR "00000000000000000000000000000000";
       right1 <= tmp1;
    --round 2
       tmp2 <= left1;
       left2 <= right1 XOR ((left1(30 downto 0) & left1(31)) AND
(left1(7 downto 0) & left1(31 downto 8))) XOR (left1(29 downto 0) &
left1(31 downto 30)) XOR "00000000000000000000000000000000";
       right2 <= tmp2;
    --round 3
       tmp3 <= left2;
       left3 <= right2 XOR ((left2(30 downto 0) & left2(31)) AND
(left2(7 downto 0) & left2(31 downto 8))) XOR (left2(29 downto 0) &
left2(31 downto 30)) XOR "00000000000000000000000000000000";
       right3 <= tmp3;
    --round 4
       tmp4 <= left3;
       left4 <= right3 XOR ((left3(30 downto 0) & left3(31)) AND
(left3(7 downto 0) & left3(31 downto 8))) XOR (left3(29 downto 0) &
left3(31 downto 30)) XOR "00000000000000000000000000000000";
       right4 <= tmp4;
    --round 5
       tmp5 <= left4;
       left5 <= right4 XOR ((left4(30 downto 0) & left4(31)) AND
(left4(7 downto 0) & left4(31 downto 8))) XOR (left4(29 downto 0) &
left4(31 downto 30)) XOR "00000000000000000000000000000000";
       right5 <= tmp5;
    --round 6
       tmp6 <= left5;
       left6 <= right5 XOR ((left5(30 downto 0) & left5(31)) AND
(left5(7 downto 0) & left5(31 downto 8))) XOR (left5(29 downto 0) &
left5(31 downto 30)) XOR "00000000000000000000000000000000";
       right6 <= tmp6;
    --round 7
       tmp7 <= left6;
       left7 <= right6 XOR ((left6(30 downto 0) & left6(31)) AND
(left6(7 downto 0) & left6(31 downto 8))) XOR (left6(29 downto 0) &
left6(31 downto 30)) XOR "00000000000000000000000000000000";
       right7 <= tmp7;
    --round 8
       tmp8 <= left7;
       left8 <= right7 XOR ((left7(30 downto 0) & left7(31)) AND
(left7(7 downto 0) & left7(31 downto 8))) XOR (left7(29 downto 0) &
left7(31 downto 30)) XOR "00000000000000000000000000000000";
       right8 <= tmp8;
    --round 9
```

```
       tmp9 <= left8;
       left9 <= right8 XOR ((left8(30 downto 0) & left8(31)) AND
(left8(7 downto 0) & left8(31 downto 8))) XOR (left8(29 downto 0) &
left8(31 downto 30)) XOR "00000000000000000000000000000000";
       right9 <= tmp9;
    --round 10
       tmp10 <= left9;
       left10 <= right9 XOR ((left9(30 downto 0) & left9(31)) AND
(left9(7 downto 0) & left9(31 downto 8))) XOR (left9(29 downto 0) &
left9(31 downto 30)) XOR "00000000000000000000000000000000";
       right10 <= tmp10;
    --round 11
       tmp11 <= left10;
       left11 <= right10 XOR ((left10(30 downto 0) & left10(31)) AND
(left10(7 downto 0) & left10(31 downto 8))) XOR (left10(29 downto 0)
& left10(31 downto 30)) XOR "00000000000000000000000000000000";
       right11 <= tmp11;
    --round 12
       tmp12 <= left11;
       left12 <= right11 XOR ((left11(30 downto 0) & left11(31)) AND
(left11(7 downto 0) & left11(31 downto 8))) XOR (left11(29 downto 0)
& left11(31 downto 30)) XOR "00000000000000000000000000000000";
       right12 <= tmp12;
    --round 13
       tmp13 <= left12;
       left13 <= right12 XOR ((left12(30 downto 0) & left12(31)) AND
(left12(7 downto 0) & left12(31 downto 8))) XOR (left12(29 downto 0)
& left12(31 downto 30)) XOR "00000000000000000000000000000000";
       right13 <= tmp13;
    --round 14
       tmp14 <= left13;
       left14 <= right13 XOR ((left13(30 downto 0) & left13(31)) AND
(left13(7 downto 0) & left13(31 downto 8))) XOR (left13(29 downto 0)
& left13(31 downto 30)) XOR "00000000000000000000000000000000";
       right14 <= tmp14;
    --round 15
       tmp15 <= left14;
       left15 <= right14 XOR ((left14(30 downto 0) & left14(31)) AND
(left14(7 downto 0) & left14(31 downto 8))) XOR (left14(29 downto 0)
& left14(31 downto 30)) XOR "00000000000000000000000000000000";
       right15 <= tmp15;
    --round 16
       tmp16 <= left15;
       left16 <= right15 XOR ((left15(30 downto 0) & left15(31)) AND
(left15(7 downto 0) & left15(31 downto 8))) XOR (left15(29 downto 0)
& left15(31 downto 30)) XOR "00000000000000000000000000000000";
       right16 <= tmp16;
    --round 17
       tmp17 <= left16;
       left17 <= right16 XOR ((left16(30 downto 0) & left16(31)) AND
(left16(7 downto 0) & left16(31 downto 8))) XOR (left16(29 downto 0)
& left16(31 downto 30)) XOR "00000000000000000000000000000000";
       right17 <= tmp17;
    --round 18
       tmp18 <= left17;
```

```
       left18 <= right17 XOR ((left17(30 downto 0) & left17(31)) AND
(left17(7 downto 0) & left17(31 downto 8))) XOR (left17(29 downto 0)
& left17(31 downto 30)) XOR "00000000000000000000000000000000";
       right18 <= tmp18;
    --round 19
       tmp19 <= left18;
       left19 <= right18 XOR ((left18(30 downto 0) & left18(31)) AND
(left18(7 downto 0) & left18(31 downto 8))) XOR (left18(29 downto 0)
& left18(31 downto 30)) XOR "00000000000000000000000000000000";
       right19 <= tmp19;
    --round 20
       tmp20 <= left19;
       left20 <= right19 XOR ((left19(30 downto 0) & left19(31)) AND
(left19(7 downto 0) & left19(31 downto 8))) XOR (left19(29 downto 0)
& left19(31 downto 30)) XOR "00000000000000000000000000000000";
       right20 <= tmp20;
    --round 21
       tmp21 <= left20;
       left21 <= right20 XOR ((left20(30 downto 0) & left20(31)) AND
(left20(7 downto 0) & left20(31 downto 8))) XOR (left20(29 downto 0)
& left20(31 downto 30)) XOR "00000000000000000000000000000000";
       right21 <= tmp21;
    --round 22
       tmp22 <= left21;
       left22 <= right21 XOR ((left21(30 downto 0) & left21(31)) AND
(left21(7 downto 0) & left21(31 downto 8))) XOR (left21(29 downto 0)
& left21(31 downto 30)) XOR "00000000000000000000000000000000";
       right22 <= tmp22;
    --round 23
       tmp23 <= left22;
       left23 <= right22 XOR ((left22(30 downto 0) & left22(31)) AND
(left22(7 downto 0) & left22(31 downto 8))) XOR (left22(29 downto 0)
& left22(31 downto 30)) XOR "00000000000000000000000000000000";
       right23 <= tmp23;
    --round 24
       tmp24 <= left23;
       left24 <= right23 XOR ((left23(30 downto 0) & left23(31)) AND
(left23(7 downto 0) & left23(31 downto 8))) XOR (left23(29 downto 0)
& left23(31 downto 30)) XOR "00000000000000000000000000000000";
       right24 <= tmp24;
    --round 25
       tmp25 <= left24;
       left25 <= right24 XOR ((left24(30 downto 0) & left24(31)) AND
(left24(7 downto 0) & left24(31 downto 8))) XOR (left24(29 downto 0)
& left24(31 downto 30)) XOR "00000000000000000000000000000000";
       right25 <= tmp25;
    --round 26
       tmp26 <= left25;
       left26 <= right25 XOR ((left25(30 downto 0) & left25(31)) AND
(left25(7 downto 0) & left25(31 downto 8))) XOR (left25(29 downto 0)
& left25(31 downto 30)) XOR "00000000000000000000000000000000";
       right26 <= tmp26;
    --round 27
       tmp27 <= left26;
```

```
       left27 <= right26 XOR ((left26(30 downto 0) & left26(31)) AND
(left26(7 downto 0) & left26(31 downto 8))) XOR (left26(29 downto 0)
& left26(31 downto 30)) XOR "00000000000000000000000000000000";
       right27 <= tmp27;
    --round 28
       tmp28 <= left27;
       left28 <= right27 XOR ((left27(30 downto 0) & left27(31)) AND
(left27(7 downto 0) & left27(31 downto 8))) XOR (left27(29 downto 0)
& left27(31 downto 30)) XOR "00000000000000000000000000000000";
       right28 <= tmp28;
    --round 29
       tmp29 <= left28;
       left29 <= right28 XOR ((left28(30 downto 0) & left28(31)) AND
(left28(7 downto 0) & left28(31 downto 8))) XOR (left28(29 downto 0)
& left28(31 downto 30)) XOR "00000000000000000000000000000000";
       right29 <= tmp29;
    --round 30
       tmp30 <= left29;
       left30 <= right29 XOR ((left29(30 downto 0) & left29(31)) AND
(left29(7 downto 0) & left29(31 downto 8))) XOR (left29(29 downto 0)
& left29(31 downto 30)) XOR "00000000000000000000000000000000";
       right30 <= tmp30;
    --round 31
       tmp31 <= left30;
       left31 <= right30 XOR ((left30(30 downto 0) & left30(31)) AND
(left30(7 downto 0) & left30(31 downto 8))) XOR (left30(29 downto 0)
& left30(31 downto 30)) XOR "00000000000000000000000000000000";
       right31 <= tmp31;
   --round 32
      tmp32 <= left31;
      left32 <= right31 XOR ((left31(30 downto 0) & left31(31)) AND
(left31(7 downto 0) & left31(31 downto 8))) XOR (left31(29 downto 0)
& left31(31 downto 30)) XOR "00000000000000000000000000000000";
      right32 <= tmp32;
 --round 33
        tmp33 <= left32;
        left33 <= right32 XOR ((left32(30 downto 0) & left32(31))
AND (left32(7 downto 0) & left32(31 downto 8))) XOR (left32(29
downto 0) & left32(31 downto 30)) XOR
"00000000000000000000000000000000";
        right33 <= tmp33;
 --round 34
      tmp34 <= left33;
      left34 <= right33 XOR ((left33(30 downto 0) & left33(31)) AND
(left33(7 downto 0) & left33(31 downto 8))) XOR (left33(29 downto 0)
& left33(31 downto 30)) XOR "00000000000000000000000000000000";
      right34 <= tmp34;
 --round 35
        tmp35 <= left34;
        left35 <= right34 XOR ((left34(30 downto 0) & left34(31))
AND (left34(7 downto 0) & left34(31 downto 8))) XOR (left34(29
downto 0) & left34(31 downto 30)) XOR
"00000000000000000000000000000000";
        right35 <= tmp35;
--round 36
        tmp36 <= left35;
```

```
        left36 <= right35 XOR ((left35(30 downto 0) & left35(31))
AND (left35(7 downto 0) & left35(31 downto 8))) XOR (left35(29
downto 0) & left35(31 downto 30)) XOR
"00000000000000000000000000000000";
        right36 <= tmp36;
--round 37
        tmp37 <= left36;
        left37 <= right36 XOR ((left36(30 downto 0) & left36(31))
AND (left36(7 downto 0) & left36(31 downto 8))) XOR (left36(29
downto 0) & left36(31 downto 30)) XOR
"00000000000000000000000000000000";
        right37 <= tmp37;
--round 38
        tmp38 <= left37;
        left38 <= right37 XOR ((left37(30 downto 0) & left37(31))
AND (left37(7 downto 0) & left37(31 downto 8))) XOR (left37(29
downto 0) & left37(31 downto 30)) XOR
"00000000000000000000000000000000";
        right38 <= tmp38;
--round 39
        tmp39 <= left38;
        left39 <= right38 XOR ((left38(30 downto 0) & left38(31))
AND (left38(7 downto 0) & left38(31 downto 8))) XOR (left38(29
downto 0) & left38(31 downto 30)) XOR
"00000000000000000000000000000000";
        right39 <= tmp39;
--round 40
        tmp40 <= left39;
        left40 <= right39 XOR ((left39(30 downto 0) & left39(31))
AND (left39(7 downto 0) & left39(31 downto 8))) XOR (left39(29
downto 0) & left39(31 downto 30)) XOR
"00000000000000000000000000000000";
        right40 <= tmp40;
--round 41
        tmp41 <= left40;
        left41 <= right40 XOR ((left40(30 downto 0) & left40(31))
AND (left40(7 downto 0) & left40(31 downto 8))) XOR (left40(29
downto 0) & left40(31 downto 30)) XOR
"00000000000000000000000000000000";
        right41 <= tmp41;
--round 42
        tmp42 <= left41;
        left42 <= right41 XOR ((left41(30 downto 0) & left41(31))
AND (left41(7 downto 0) & left41(31 downto 8))) XOR (left41(29
downto 0) & left41(31 downto 30)) XOR
"00000000000000000000000000000000";
        right42 <= tmp42;
--round 43
        tmp43 <= left42;
        left43 <= right42 XOR ((left42(30 downto 0) & left42(31))
AND (left42(7 downto 0) & left42(31 downto 8))) XOR (left42(29
downto 0) & left42(31 downto 30)) XOR
"00000000000000000000000000000000";
        right43 <= tmp43;
dout <= right43 & left43;
end Behavioral;
```

Apêndice H

IMPLEMENTAÇÃO FPGA SSR SPECK 64/128

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use ieee.numeric_std.all;

entity Speck_Variable is
  Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0);
           key   : in STD_LOGIC_VECTOR (127 downto 0)
  );
end Speck_Variable;

architecture Behavioral of Speck_Variable is
constant n : integer := 32;
constant m : integer := 4;
constant rounds : integer := 27;
--alpha 8 beta 3

type MEM_ROM is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
begin
process (clock, reset)
 variable left : STD_LOGIC_VECTOR (31 downto 0);
 variable right : STD_LOGIC_VECTOR (31 downto 0);
 variable ky : MEM_ROM;
 variable l : MEM_ROM;

--Variaveis da key
 variable ky_tmp : STD_LOGIC_VECTOR (31 downto 0);
 variable ky_alfa : STD_LOGIC_VECTOR (31 downto 0);
 variable ky_beta : STD_LOGIC_VECTOR (31 downto 0);
 variable aux : STD_LOGIC_VECTOR (31 downto 0);
 --variaveis enc dec

 variable tmp : STD_LOGIC_VECTOR (31 downto 0);
 variable alfa : STD_LOGIC_VECTOR (31 downto 0);
 variable beta : STD_LOGIC_VECTOR (31 downto 0);
begin
    if reset = '1' then
        for i in 0 to 26
```

```
        loop
           ky(i) := (others => '0');
           l(i) := (others => '0');
        end loop;
       dout <= (others => '0');
       left := (others => '0');
       right := (others => '0');
    elsif clock'event and clock = '1' then
        ky(0) := key(31 downto 0);
        l(0) := key(63 downto 32);
        l(1) := key(95 downto 64);
        l(2) := key(127 downto 96);
        left := din(63 downto 32);
        right := din(31 downto 0);
--keyexpansion
    for i in 0 to (rounds-2)
          loop
             ky_alfa := l(i)(7 downto 0) & l(i)(31 downto 8);
             ky_tmp := ky(i) + ky_alfa;
             aux := std_logic_vector(to_unsigned(i, aux'length));
             l(i+m-1) := ky_tmp XOR aux;

             ky_beta := ky(i)(28 downto 0) & ky(i)(31 downto 29);
             ky(i+1) := ky_beta XOR l(i+m-1);
          end loop;
--encrypt
    if ed = '0' then
              for j in 0 to (rounds - 1)
              loop
                 alfa := left(7 downto 0) & left(31 downto 8);
                 left := (alfa + right) xor ky(j);

                 beta := right(28 downto 0) & right(31 downto 29);
                 right := beta xor left;
              end loop;
              dout <= left & right;
--decrypt
    elsif ed = '1' then

             for Z in (rounds-1) downto 0
             loop
                tmp := right XOR left;
                right := tmp(2 downto 0) & tmp(31 downto 3);

                aux := ((left xor ky(z)) - right);
                left := aux(23 downto 0) & aux(31 downto 24);
            end loop;
            dout <= left & right;
          end if;
       end if;
    end process;
end Behavioral;
```

Apêndice I

IMPLEMENTAÇÃO FPGA SFR SPECK 64/128

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use ieee.numeric_std.all;

entity Speck_Signal is
  Port (clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0);
           key   : in STD_LOGIC_VECTOR (127 downto 0)
   );
end Speck_Signal;

architecture Behavioral of Speck_Signal is
constant n : integer := 32;
constant m : integer := 4;
constant rounds : integer := 27;

type MEM_ROM is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);

type STATUS is (DISPONIVEL, WORK, ESPERA);
signal ESTADO : STATUS;
signal skey : STD_LOGIC_VECTOR(63 downto 0);
signal ky : MEM_ROM;
signal l : MEM_ROM;
signal left : STD_LOGIC_VECTOR(31 downto 0);
signal right : STD_LOGIC_VECTOR(31 downto 0);
signal count : integer range 0 to 27;
signal kcount : integer range 0 to 27;

begin
process (clock, reset)
 variable ky_tmp : STD_LOGIC_VECTOR (31 downto 0);
variable ky_alfa : STD_LOGIC_VECTOR (31 downto 0);
variable ky_beta : STD_LOGIC_VECTOR (31 downto 0);
variable aux : STD_LOGIC_VECTOR (31 downto 0);
 variable tmp : STD_LOGIC_VECTOR (31 downto 0);
begin
    if reset = '1' then
        ky(0) <= (others => '0');
        ky(1) <= (others => '0');
```

```
        ky(2) <= (others => '0');
        ky(3) <= (others => '0');
        left <= (others => '0');
        right <= (others => '0');
        count <= 0;
        kcount<= 4;
        dout <= (others => '0');
        ESTADO <= DISPONIVEL;
    case ESTADO is
       when DISPONIVEL =>
            ky(0) <= key(31 downto 0);
               l(0) <= key(63 downto 32);
               l(1) <= key(95 downto 64);
               l(2) <= key(127 downto 96);
               left <= din(63 downto 32);
               right <= din(31 downto 0);
          if ed = '0' then
             count <= 0;
          elsif ed = '1' then
             count <= 31;
          end if;
          kcount<= m;
          dout <= (others => '0');
          ESTADO <= WORK;
       when WORK =>
       --keyexpansion
           if (kcount <= 25) then
               ky_alfa := l(kcount)(7 downto 0) & l(kcount)(31
downto 8);
                ky_tmp := ky(kcount) + ky_alfa;
                aux := std_logic_vector(to_unsigned(kcount,
aux'length));
                l(kcount+m-1) <= ky_tmp XOR aux;

                ky_beta := ky(kcount)(28 downto 0) & ky(kcount)(31
downto 29);
                ky(kcount+1) <= ky_beta XOR l(kcount+m-1);

                kcount <= kcount +1;
           end if;
--encrypt
       if ed = '0' then
           if (count <= 26) then
              left <= (left(7 downto 0) & left(31 downto 8) + right)
xor ky(count);
             right <= right(28 downto 0) & right(31 downto 29) xor
left;

             if (count < 26) then
               count <= count + 1;
            elsif (count = 26) then
               ESTADO <= ESPERA;
            end if;
           end if;
        elsif ed = '1' then
           if (kcount = 25) then
```

```
                tmp := right XOR left;
                right <= tmp(2 downto 0) & tmp(31 downto 3);

                aux := ((left xor ky(kcount)) - right);
                left <= aux(23 downto 0) & aux(31 downto 24);
                if (count > 0) then
                  count <= count - 1;
                elsif (count = 0) then
                  ESTADO <= ESPERA;
                end if;
              end if;
           end if;
            when ESPERA =>
                kcount <= m;
                count <= 0;
                dout <= left & right;
                ESTADO <= DISPONIVEL;
            when others =>
                ESTADO <= DISPONIVEL;
            end case;
         end if;
        end process;

end Behavioral;
```

Apêndice J

IMPLEMENTAÇÃO FPGA AFR SPECK 64/128

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use ieee.numeric_std.all;

entity Speck_combinational is
  Port ( clock : in STD_LOGIC;
           ed    : in STD_LOGIC;
           din   : in STD_LOGIC_VECTOR (63 downto 0);
           dout  : out STD_LOGIC_VECTOR (63 downto 0);
           key   : in STD_LOGIC_VECTOR (127 downto 0)
  );
end Speck_combinational;

architecture Behavioral of Speck_combinational is
type MEM_ROM is array (0 to 26) of STD_LOGIC_VECTOR (31 downto 0);
signal ky : MEM_ROM;
signal l : MEM_ROM;

signal leftEnc : STD_LOGIC_VECTOR(31 downto 0);
signal left0 : STD_LOGIC_VECTOR(31 downto 0);
signal left1 : STD_LOGIC_VECTOR(31 downto 0);
signal left2 : STD_LOGIC_VECTOR(31 downto 0);
signal left3 : STD_LOGIC_VECTOR(31 downto 0);
signal left4 : STD_LOGIC_VECTOR(31 downto 0);
signal left5 : STD_LOGIC_VECTOR(31 downto 0);
signal left6 : STD_LOGIC_VECTOR(31 downto 0);
signal left7 : STD_LOGIC_VECTOR(31 downto 0);
signal left8 : STD_LOGIC_VECTOR(31 downto 0);
signal left9 : STD_LOGIC_VECTOR(31 downto 0);
signal left10 : STD_LOGIC_VECTOR(31 downto 0);
signal left11 : STD_LOGIC_VECTOR(31 downto 0);
signal left12 : STD_LOGIC_VECTOR(31 downto 0);
signal left13 : STD_LOGIC_VECTOR(31 downto 0);
signal left14 : STD_LOGIC_VECTOR(31 downto 0);
signal left15 : STD_LOGIC_VECTOR(31 downto 0);
signal left16 : STD_LOGIC_VECTOR(31 downto 0);
signal left17 : STD_LOGIC_VECTOR(31 downto 0);
signal left18 : STD_LOGIC_VECTOR(31 downto 0);
signal left19 : STD_LOGIC_VECTOR(31 downto 0);
signal left20 : STD_LOGIC_VECTOR(31 downto 0);
signal left21 : STD_LOGIC_VECTOR(31 downto 0);
signal left22 : STD_LOGIC_VECTOR(31 downto 0);
```

```
signal left23 : STD_LOGIC_VECTOR(31 downto 0);
signal left24 : STD_LOGIC_VECTOR(31 downto 0);
signal left25 : STD_LOGIC_VECTOR(31 downto 0);
signal left26 : STD_LOGIC_VECTOR(31 downto 0);

signal rightEnc : STD_LOGIC_VECTOR(31 downto 0);
signal right0 : STD_LOGIC_VECTOR(31 downto 0);
signal right1 : STD_LOGIC_VECTOR(31 downto 0);
signal right2 : STD_LOGIC_VECTOR(31 downto 0);
signal right3 : STD_LOGIC_VECTOR(31 downto 0);
signal right4 : STD_LOGIC_VECTOR(31 downto 0);
signal right5 : STD_LOGIC_VECTOR(31 downto 0);
signal right6 : STD_LOGIC_VECTOR(31 downto 0);
signal right7 : STD_LOGIC_VECTOR(31 downto 0);
signal right8 : STD_LOGIC_VECTOR(31 downto 0);
signal right9 : STD_LOGIC_VECTOR(31 downto 0);
signal right10 : STD_LOGIC_VECTOR(31 downto 0);
signal right11 : STD_LOGIC_VECTOR(31 downto 0);
signal right12 : STD_LOGIC_VECTOR(31 downto 0);
signal right13 : STD_LOGIC_VECTOR(31 downto 0);
signal right14 : STD_LOGIC_VECTOR(31 downto 0);
signal right15 : STD_LOGIC_VECTOR(31 downto 0);
signal right16 : STD_LOGIC_VECTOR(31 downto 0);
signal right17 : STD_LOGIC_VECTOR(31 downto 0);
signal right18 : STD_LOGIC_VECTOR(31 downto 0);
signal right19 : STD_LOGIC_VECTOR(31 downto 0);
signal right20 : STD_LOGIC_VECTOR(31 downto 0);
signal right21 : STD_LOGIC_VECTOR(31 downto 0);
signal right22 : STD_LOGIC_VECTOR(31 downto 0);
signal right23 : STD_LOGIC_VECTOR(31 downto 0);
signal right24 : STD_LOGIC_VECTOR(31 downto 0);
signal right25 : STD_LOGIC_VECTOR(31 downto 0);
signal right26 : STD_LOGIC_VECTOR(31 downto 0);


begin
   leftenc <= din(63 downto 32);
   rightenc <= din(31 downto 0);
--round 0
   left0 <= ((leftenc(7 downto 0) & leftenc(31 downto 8)) +
rightenc) XOR "00000000000000000000000000000000";
   right0 <= (rightenc(28 downto 0) & rightenc(31 downto 29)) XOR
left0;
--round 1
   left1 <= ((left0(7 downto 0) & left0(31 downto 8)) + right0) XOR
"00000000000000000000000000000000";
   right1 <= (right0(28 downto 0) & right0(31 downto 29)) XOR left1;
--round 2
   left2 <= ((left1(7 downto 0) & left1(31 downto 8)) + right1) XOR
"00000000000000000000000000000000";
   right2 <= (right1(28 downto 0) & right1(31 downto 29)) XOR left2;
 --round 3
   left3 <= ((left2(7 downto 0) & left2(31 downto 8)) + right2) XOR
"00000000000000000000000000000000";
   right3 <= (right2(28 downto 0) & right2(31 downto 29)) XOR left3;
--round 4
```

```
   left4 <= ((left3(7 downto 0) & left3(31 downto 8)) + right3) XOR
"00000000000000000000000000000000";
   right4 <= (right3(28 downto 0) & right3(31 downto 29)) XOR left4;
--round 5
   left5 <= ((left4(7 downto 0) & left4(31 downto 8)) + right4) XOR
"00000000000000000000000000000000";
   right5 <= (right4(28 downto 0) & right4(31 downto 29)) XOR left5;
--round 6
   left6 <= ((left5(7 downto 0) & left5(31 downto 8)) + right5) XOR
"00000000000000000000000000000000";
   right6 <= (right5(28 downto 0) & right5(31 downto 29)) XOR left6;
--round 7
   left7 <= ((left6(7 downto 0) & left6(31 downto 8)) + right6) XOR
"00000000000000000000000000000000";
   right7 <= (right6(28 downto 0) & right6(31 downto 29)) XOR left7;
 --round 8
   left8 <= ((left7(7 downto 0) & left7(31 downto 8)) + right7) XOR
"00000000000000000000000000000000";
   right8 <= (right7(28 downto 0) & right7(31 downto 29)) XOR left8;
--round 9
   left9 <= ((left8(7 downto 0) & left8(31 downto 8)) + right8) XOR
"00000000000000000000000000000000";
   right9 <= (right8(28 downto 0) & right8(31 downto 29)) XOR left9;
--round 10
   left10 <= ((left9(7 downto 0) & left9(31 downto 8)) + right9) XOR
"00000000000000000000000000000000";
   right10 <= (right9(28 downto 0) & right9(31 downto 29)) XOR
left10;
--round 11
   left11 <= ((left10(7 downto 0) & left10(31 downto 8)) + right10)
XOR "00000000000000000000000000000000";
   right11 <= (right10(28 downto 0) & right10(31 downto 29)) XOR
left11;
--round 12
   left12 <= ((left11(7 downto 0) & left11(31 downto 8)) + right11)
XOR "00000000000000000000000000000000";
   right12 <= (right11(28 downto 0) & right11(31 downto 29)) XOR
left12;
 --round 13
   left13 <= ((left12(7 downto 0) & left12(31 downto 8)) + right12)
XOR "00000000000000000000000000000000";
   right13 <= (right12(28 downto 0) & right12(31 downto 29)) XOR
left13;
--round 14
   left14 <= ((left13(7 downto 0) & left13(31 downto 8)) + right13)
XOR "00000000000000000000000000000000";
   right14 <= (right13(28 downto 0) & right13(31 downto 29)) XOR
left14;
--round 15
   left15 <= ((left14(7 downto 0) & left14(31 downto 8)) + right14)
XOR "00000000000000000000000000000000";
   right15 <= (right14(28 downto 0) & right14(31 downto 29)) XOR
left15;
--round 16
   left16 <= ((left15(7 downto 0) & left15(31 downto 8)) + right15)
XOR "00000000000000000000000000000000";
```

```
   right16 <= (right15(28 downto 0) & right15(31 downto 29)) XOR
left16;
--round 17
   left17 <= ((left16(7 downto 0) & left16(31 downto 8)) + right16)
XOR "00000000000000000000000000000000";
   right17 <= (right16(28 downto 0) & right16(31 downto 29)) XOR
left17;
 --round 18
   left18 <= ((left17(7 downto 0) & left17(31 downto 8)) + right17)
XOR "00000000000000000000000000000000";
   right18 <= (right17(28 downto 0) & right17(31 downto 29)) XOR
left18;
--round 19
   left19 <= ((left18(7 downto 0) & left18(31 downto 8)) + right18)
XOR "00000000000000000000000000000000";
   right19 <= (right18(28 downto 0) & right18(31 downto 29)) XOR
left19;
--round 20
   left20 <= ((left19(7 downto 0) & left19(31 downto 8)) + right19)
XOR "00000000000000000000000000000000";
   right20 <= (right19(28 downto 0) & right19(31 downto 29)) XOR
left20;
--round 21
   left21 <= ((left20(7 downto 0) & left20(31 downto 8)) + right20)
XOR "00000000000000000000000000000000";
   right21 <= (right20(28 downto 0) & right20(31 downto 29)) XOR
left21;
--round 22
    left22 <= ((left21(7 downto 0) & left21(31 downto 8)) + right21)
XOR "00000000000000000000000000000000";
    right22 <= (right21(28 downto 0) & right21(31 downto 29)) XOR
left22;
--round 23
    left23 <= ((left22(7 downto 0) & left22(31 downto 8)) + right22)
XOR "00000000000000000000000000000000";
    right23 <= (right22(28 downto 0) & right22(31 downto 29)) XOR
left23;
--round 24
    left24 <= ((left23(7 downto 0) & left23(31 downto 8)) + right23)
XOR "00000000000000000000000000000000";
    right24 <= (right23(28 downto 0) & right23(31 downto 29)) XOR
left24;
--round 25
    left25 <= ((left24(7 downto 0) & left24(31 downto 8)) + right24)
XOR "00000000000000000000000000000000";
    right25 <= (right24(28 downto 0) & right24(31 downto 29)) XOR
left25;
--round 26
    left26 <= ((left25(7 downto 0) & left25(31 downto 8)) + right25)
XOR "00000000000000000000000000000000";
    right26 <= (right25(28 downto 0) & right25(31 downto 29)) XOR
left26;
   dout <= left26 & right26;

end Behavioral;
```

Apêndice K

IMPLEMENTAÇÃO GPGPU SIMON 64/128

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <cuda_runtime.h>

const int rounds_h = 44;
const int m_h = 4;

__constant__ int rounds;
__constant__ int m;
__constant__ int key[44];

int z[62] =
{1,1,0,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,1,0,1,1,1,1,0,0,0,0,0,0,1
,0,0,1,0,0,0,1,0,1,0,0,1,1,1,0,0,1,1,0,1,0,0,0,0,1,1,1,1};

#define ROTATE_RIGHT_64(x,bits) ( (x >> bits) | (x << (32-bits)))

#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void cudaDevAssist(cudaError_t code, int line, bool
abort=true)
{
      if (code != cudaSuccess)
      {
            fprintf(stderr,"cudaDevAssistant: %s %d\n",
cudaGetErrorString(code), line);
            if (abort) exit(code);
      }
}

__global__ void cudaRunner(int *pt)
{
      int i = 0, j = 0, left = 0, right = 0, ERL1 = 0, ERL8 = 0, ERL2
= 0, auxenc = 0;

      for(i = 0; i < 262144; i+=2){

            left = pt[i];
            right = pt[i+1];

            ERL1 = (left << 1) || (left >> (32-1));
            ERL8 = (left << 8) || (left >> (32-8));
```

```
            ERL2 = (left << 2) || (left >> (32-2));

            #pragma unroll 44
            for (j = 0; j < 44; j++){
                  auxenc = left;
                  left = right ^ (ERL1 & ERL8) ^ ERL2 ^ key[j];
                  right = auxenc;
            }
            pt[i] = left;
            pt[i+1] = right;
      }
}

void K_Exp(int* key, int* skey)
{
      int i = 0, auxkey = 0;
      for (i = m; i < rounds; i++){
            auxkey = ROTATE_RIGHT_64(key[i-1],3);
            auxkey ^= key[i-3];
            auxkey ^= ROTATE_RIGHT_64(auxkey,1);
            skey[i] = ~key[i-m] ^ auxkey ^ z[(i-m) % 62] ^ 3;
      }
}

int main()
{

      int blocks[262144], result = 0;

      FILE *arq;

      arq = fopen("ArqTeste.dat", "rb");

      result = fread (&blocks[0], sizeof(int), 262144, arq);

      printf("Nro de elementos lidos: %d\n", result);

      int inkey[4], outkey[rounds];
      inkey[3] = 0x03020100;
      inkey[2] = 0x0b0a0908;
      inkey[1] = 0x13121110;
      inkey[0] = 0x1b1a1918;

      K_Exp(inkey, outkey);

      //send constants to GPU
      cudaSetDevice(0);
      cudaDevAssist(cudaMemcpyToSymbol(rounds, &rounds_h,
sizeof(int), 0, cudaMemcpyHostToDevice), 250, true); //mudar numero
da linha depois
      cudaDevAssist(cudaMemcpyToSymbol(m, &m_h, sizeof(int), 0,
cudaMemcpyHostToDevice), 543, true);
      cudaDevAssist(cudaMemcpyToSymbol(key, &outkey, 44*sizeof(int),
0, cudaMemcpyHostToDevice), 823, true);
      cudaThreadSynchronize();
```

```
      int *gpublocks = NULL;
      cudaDevAssist(cudaMalloc((void**)&gpublocks,
result*sizeof(int)), 425, true);

           cudaDevAssist(cudaMemcpy(gpublocks, blocks,
result*sizeof(int), cudaMemcpyHostToDevice), 426, true);
           cudaDevAssist(cudaDeviceSynchronize(), 268, true);
           cudaRunner<<<1,1024>>>(gpublocks);

           cudaDevAssist(cudaDeviceSynchronize(), 270, true);
           cudaDevAssist(cudaMemcpy(blocks, gpublocks,
result*sizeof(int), cudaMemcpyDeviceToHost), 431, true);

      cudaFree(gpublocks);
      cudaDeviceReset();
      fclose(arq);
      return 0;
}
```

Apêndice L

IMPLEMENTAÇÃO GPGPU SPECK 64/128

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <cuda_runtime.h>

const int rounds_h = 27;
const int m_h = 4;

__constant__ int rounds;
__constant__ int m;
__constant__ int key[27];
__constant__ int l[27];

#define ROTATE_LEFT_64(x,bits) ( (x << bits) | (x >> (32-bits)) )
#define ROTATE_RIGHT_64(x,bits) ( (x >> bits) | (x << (32-bits)))

#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void cudaDevAssist(cudaError_t code, int line, bool
abort=true)
{
      if (code != cudaSuccess)
      {
            fprintf(stderr,"cudaDevAssistant: %s %d\n",
cudaGetErrorString(code), line);
            if (abort) exit(code);
      }
}

__global__ void cudaRunner(int *pt)
{
      int i = 0, j = 0, left = 0, right = 0, ERL3 = 0, DRR8 = 0;

      for(i = 0; i < 262144; i+=2){

            //encrypt
      left = pt[i];
      right = pt[i+1];

      ERL3 = (left << 3) || (left >> (32-3));
            DRR8 = (right << 8) || (right >> (32-8));

      #pragma unroll 27
```

```
      for (j = 0; j < 27; j++){
              right = (DRR8 + left) ^ key[j];
              left = ERL3 ^ right;
      }
      pt[i] = left;
           pt[i+1] = right;
      }
}

void K_Exp(int* lkey, int* key,  int* skey)
{
      int i = 0;
      skey[0] = key[0];
      for (i = 0; i <= rounds - 2; i++){
           lkey[i+m-1] = (skey[i] + ROTATE_RIGHT_64(lkey[i],8)) ^ i;
           skey[i+1] = ROTATE_LEFT_64(skey[i],3) ^ lkey[i+m-1];
      }
}

int main()
{

      int blocks[262144], result = 0;

      FILE *arq;

      arq = fopen("ArqTeste.dat", "rb");

      result = fread (&blocks[0], sizeof(int), 262144, arq);

      printf("Nro de elementos lidos: %d\n", result);

      int inkey[1], outkey[rounds];
      l[2] = 0x03020100;
      l[1] = 0x0b0a0908;
      l[0] = 0x13121110;
      inkey[0] = 0x1b1a1918;

      K_Exp(l, inkey, outkey);

      //send constants to GPU
      cudaSetDevice(0);
      cudaDevAssist(cudaMemcpyToSymbol(rounds, &rounds_h,
sizeof(int), 0, cudaMemcpyHostToDevice), 250, true); //mudar numero
da linha depois
      cudaDevAssist(cudaMemcpyToSymbol(m, &m_h, sizeof(int), 0,
cudaMemcpyHostToDevice), 543, true);
      cudaDevAssist(cudaMemcpyToSymbol(key, &outkey, 27*sizeof(int),
0, cudaMemcpyHostToDevice), 823, true);
      cudaThreadSynchronize();

      int *gpublocks = NULL;
      cudaDevAssist(cudaMalloc((void**)&gpublocks,
result*sizeof(int)), 425, true);
```

```
            cudaDevAssist(cudaMemcpy(gpublocks, blocks,
result*sizeof(int), cudaMemcpyHostToDevice), 426, true);
            cudaDevAssist(cudaDeviceSynchronize(), 268, true);
            cudaRunner<<<1,1024>>>(gpublocks);

            cudaDevAssist(cudaDeviceSynchronize(), 270, true);
            cudaDevAssist(cudaMemcpy(blocks, gpublocks,
result*sizeof(int), cudaMemcpyDeviceToHost), 431, true);

      cudaFree(gpublocks);
      cudaDeviceReset();
      fclose(arq);
      return 0;
}
```