DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Implementação em FPGA de um mecanismo de encapsulamento de chave pós-quântico utilizando HLS

Renata Colares Policarpo

Brasília, junho de 2023

Programa de Pós-Graduação Profissional em Engenharia Elétrica
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Implementação em FPGA de um mecanismo de encapsulamento de chave pós-quântico utilizando HLS

Renata Colares Policarpo

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Alexandre Solon Nery, Ph.D, FT/UnB _____
Orientador

Prof. Robson de Oliveira Albuquerque, Ph.D, FT/UnB _____
Coorientador

Prof. João José Costa Gondim, Ph.D, FT/UnB _____
Examinador interno

Pesquisador Evander Pereira de Rezende, Ph.D, CEPESC _____
Examinador externo

FICHA CATALOGRÁFICA

POLICARPO, RENATA COLARES
Implementação em FPGA de um mecanismo de encapsulamento de chave pós-quântico utilizando HLS [Distrito Federal] 2023.
xvi, 73 p., publicação: PPEE.MP.045, 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2023).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Criptografia pós-quântica	2. Mecanismo de encapsulamento de chave
3. FPGA	4. HLS
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

POLICARPO, R. C. (2023). *Implementação em FPGA de um mecanismo de encapsulamento de chave pós-quântico utilizando HLS*. Dissertação de Mestrado Profissional, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 73 p., publicação: PPEE.MP.045

CESSÃO DE DIREITOS

AUTOR: Renata Colares Policarpo
TÍTULO: Implementação em FPGA de um mecanismo de encapsulamento de chave pós-quântico utilizando HLS.
GRAU: Mestre em Engenharia Elétrica ANO: 2023

Renata Colares Policarpo
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

Dedico esse trabalho aos meus pais, José Carlos e Terezinha, que nunca mediram esforços para alavancar meu desenvolvimento pessoal e acadêmico.

AGRADECIMENTOS

Primeiramente agradeço a Deus pelo dom da vida, por me capacitar e permitir a conclusão de mais essa etapa. À minha família, pelo apoio incondicional e pela compreensão nos momentos de ausência. Ao meu orientador por toda paciência, pela tranquilidade transmitida e pela ajuda na construção do conhecimento, sem as quais essa tarefa não teria sido finalizada. Ao meu coorientador, pelo incentivo desde as etapas iniciais desse processo. Aos meus colegas de trabalho que foram suporte nos momentos de dúvidas. Aos meus amigos que, de forma direta ou indireta, me ajudaram nessa caminhada. À instituição em que trabalho, à Universidade de Brasília, em especial ao PPEE, e aos demais docentes que contribuíram com a minha formação, o meu sincero agradecimento.

RESUMO

Essa dissertação apresenta a especificação de um acelerador para o CRYSTALS-Kyber, o primeiro mecanismo de encapsulamento de chaves (KEM) padronizado pelo *National Institute of Standards and Technology* (NIST) como criptografia pós-quântica (PQC). O acelerador, que foi desenvolvido com síntese de alto nível (HLS), é composto pelas operações de cifração e decifração presentes nos algoritmos de encapsulamento e desencapsulamento do KEM Kyber. A arquitetura desenvolvida faz uso de 33733 LUTs, 22810 FFs e 151 DSPs, sendo implementada em uma FPGA de baixo custo PYNQ-Z1 (XC7Z020-1 CLG400C). Em uma simulação de troca de chaves realizada com a ferramenta Vitis HLS, o acelerador gastou o tempo total de aproximadamente 3,81 milissegundos, operando a 100MHz. Nessa mesma simulação, a arquitetura desenvolvida teve um consumo estimado de 2,243W de potência. Com a implementação do acelerador na FPGA, o tempo observado para realização das operações de cifração e decifração foi de 5,01 e 2,24 milissegundos, respectivamente. O consumo de energia nesse processo foi de aproximadamente 6,2 Joules.

Palavras-chave: Criptografia pós-quântica; PQC; CRYSTALS-Kyber; acelerador; FPGA; HLS.

ABSTRACT

This dissertation presents the specification of an accelerator for CRYSTALS-Kyber, the first Key Encapsulation Mechanism (KEM) standardized by the *National Institute of Standards and Technology* (NIST) as Post-Quantum Cryptography (PQC). The accelerator was developed with high-level synthesis (HLS) and it is composed of the encryption and decryption operations present in the KEM Kyber encapsulation and decapsulation algorithms. The developed architecture makes use of 33733 LUTs, 22810 FFs and 151 DSPs, being implemented in a low cost FPGA PYNQ-Z1 (XC7Z020-1 CLG400C). In a key exchange simulation performed with the Vitis HLS tool, the accelerator spent a total time of approximately 3.81 milliseconds, operating at 100MHz. In this simulation, the architecture developed had an estimated consumption of 2.243W of power. With the implementation of the accelerator in the FPGA, the observed time to perform the encryption and decryption operations was 5.01 and 2.24 milliseconds, respectively. The energy consumption in this process was approximately 6.2 Joules.

Keywords: Post-Quantum Cryptography; PQC; CRYSTALS-Kyber; accelerator; FPGA; HLS.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AES	*Advanced Encryption Standard*
ALU	Unidade Lógica e Aritmética
BRAM	*Block Random Access Memory*
CCA	Ataque por texto cifrado escolhido
CCA2	Ataque adaptável por texto cifrado escolhido
CLB	Bloco Lógico Configurável
CPA	Ataque por texto em claro escolhido
CPU	Unidade Central de Processamento
CRYSTALS	*Cryptographic Suite for Algebraic Lattices*
CVP	*Closest Vector Problem*
DDR	*Double Data Rate*
DSP	*Digital Signal Processor*
ECC	Criptografia de Curva Elíptica
EDVAC	*Electronic Discrete Variable Automatic Computer*
FF	*Flip-Flop*
FIPS	*Federal Information Processing Standards*
FPGA	*Field-Programmable Gate Array*
GPGPU	Unidade de Processamento Gráfico para Propósito Geral
GPU	Unidade de Processamento Gráfico
HDL	Linguagem de Descrição de Hardware
HLS	Síntese de Alto Nível
IAS	*Institute for Advanced Study*
IEEE	*Institute of Electrical and Electronics Engineers*
IND	Indistinguibilidade
IND-CCA2	Indistinguibilidade sob ataque adaptativo por texto cifrado escolhido
IND-CPA	Indistinguibilidade sob ataque por texto em claro escolhido
KDF	Função de Derivação de Chave
KEM	Mecanismo de Encapsulamento de Chaves
KEM.Dec	Algoritmo de Desencapsulamento do KEM
KEM.Enc	Algoritmo de Encapsulamento do KEM
KEM.KeyGen	Algoritmo de Geração de Chaves do KEM
LUT	*Look-Up Table*
LWE	*Learning With Errors*
MLWE	*Module Learning With Errors*
MSIS	*Module Short Integer Solution*
MUX	Multiplexador
NM	Não-maleabilidade
NIST	*National Institute of Standards and Technology*

NTT	Transformada Numérica de Fourier
PKE	Cifração de Chave Pública
PKE.Dec	Algoritmo de Decifração da PKE
PKE.Enc	Algoritmo de Cifração da PKE
PKE.KeyGen	Algoritmo de Geração de Chaves da PKE
PLD	Dispositivo Lógico Programável
PQC	Criptografia Pós-Quântica
PRF	Função Pseudo-aleatória
QROM	Modelo do Oráculo Aleatório Quântico
RLWE	*Ring Learning With Errors*
ROM	Modelo do Oráculo Aleatório
RSA	*Rivest–Shamir–Adleman*
RSIS	*Ring Short Integer Solution*
RTL	Nível de Transferência entre Registradores
SHA3	*Secure Hash Algorithm 3*
SHAKE	*Secure Hash Algorithm and Keccak*
SIMD	Única Instrução - Múltiplos Dados
SIS	*Short Integer Solution*
SIVP	*Shortest Independent Vector Problem*
SVP	*Shortest Vector Problem*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
XOF	Função de Saída Extensível

1 INTRODUÇÃO

A dinâmica do mundo moderno, com o contínuo desenvolvimento de novas tecnologias, fez com que a quantidade de informações que trafegam por rede aumentasse consideravelmente nos últimos anos. A pesquisa realizada pelo Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação, publicada em [1], corrobora com essa constatação. Ela aponta que o distanciamento social, medida adotada para a contenção da disseminação da COVID-19, intensificou o acesso e o uso da internet pelos brasileiros. Essa demanda por conectividade veio, entre outros fatores, para viabilizar o acesso a serviços que antes eram executados presencialmente e passaram a ser realizados à distância, de forma virtual.

Com isso, há uma quantidade crescente de dados sensíveis sendo transmitidos a cada instante por meio de diferentes operações, tais como transações bancárias, teleconferências, comércio eletrônico, redes sociais, dentre outras. O avanço da tecnologia fez com essas operações pudessem ser realizadas por diversos dispositivos, como celulares, computadores e tablets. Essa facilidade de acesso à rede expõe a necessidade de garantir a segurança e privacidade dessas informações sensíveis que são compartilhadas, conforme apontado em [2].

Nesse cenário, a criptografia torna-se uma ferramenta essencial pois, como definido em [3], a criptografia permite que que haja comunicação de forma segura em um canal inseguro. Sendo assim, ela é uma ciência que consiste no desenvolvimento de algoritmos, técnicas e protocolos com a finalidade de garantir a segurança das informações.

Os requisitos de segurança da informação, tais como a confidencialidade, autenticidade e integridade de dados, podem ser alcançados com a utilização de protocolos criptográficos, seja enquanto os dados trafegam em rede ou quando são armazenados digitalmente [4]. Isso impede pessoas não autorizadas de acessarem ou fazerem alterações indesejadas nas informações. O estudo publicado em [5] divide a criptografia em três tipos principais: de chave secreta ou simétrica, de chave pública ou assimétrica, e funções de hash. A Figura 1.1 apresenta alguns exemplos de algoritmos desenvolvidos segundo o tipo de criptografia ao qual pertencem.

A criptografia é uma área em constante evolução, pois os algoritmos podem se tornar vulneráveis e obsoletos à medida que a tecnologia evolui e novos ataques são desenvolvidos por especialistas e estudiosos do mundo inteiro [6]. Para assegurar que os protocolos criptográficos desempenharão sua função de resguardar informações sensíveis, faz-se necessário que o algoritmo utilizado em sua construção seja robusto e confiável.

O *National Institute of Standards and Technology* (NIST) é um instituto americano que estabelece padrões a serem adotados em inúmeros produtos e serviços. Na área de criptografia, o NIST testa, valida e padroniza algoritmos, com o auxílio da indústria, do governo e da academia, e também estabelece documentos com diretrizes para a utilização desses algoritmos [7].

Há vários algoritmos já aprovados e recomendados pelo NIST como, por exemplo, o *Advanced Encryption Standard* (AES) de chave simétrica [8] , o *Rivest-Shamir-Adleman* (RSA) de chave pública [9] e o *Secure Hash Algorithm-3* (SHA-3) como função de hash [10], dentre outros. Essa padronização possibi-

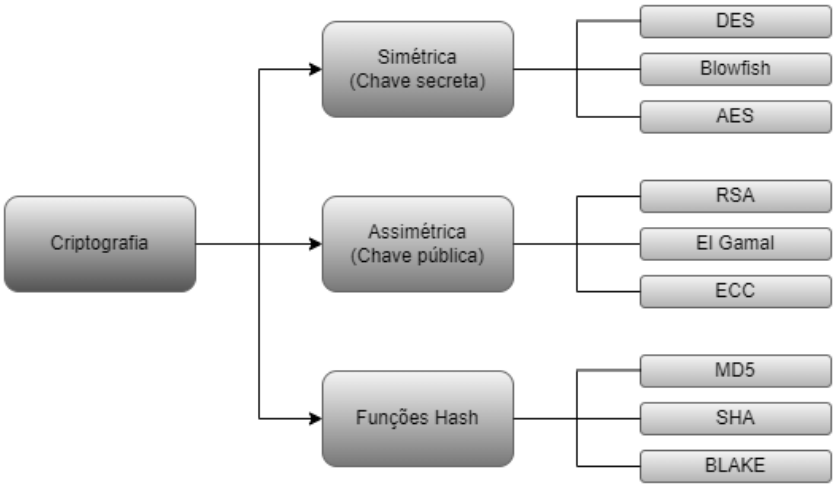

Figura 1.1: Exemplos de algoritmos criptográficos classificados por tipo. (Fonte: o autor)

lita a implantação de esquemas criptográficos efetivos e robustos que permitem comunicação e aplicativos seguros.

No entanto, os recentes avanços nas pesquisas e no desenvolvimento da computação quântica representam uma ameaça aos padrões criptográficos atuais [11]. Há estudos da década de 90 que mostram que o computador quântico é capaz de fragilizar os algoritmos presentes nos sistemas criptográficos mais utilizados atualmente.

Grover, em [12], propôs um algoritmo quântico capaz de reduzir o nível de segurança de sistemas criptográficos que utilizam chave simétrica e de funções hash. Na presença dos computadores quânticos, esses sistemas continuariam seguros desde que houvesse um aumento significativo no número de bits das chaves utilizadas.

Shor traz em [13] um estudo com consequências mais danosas, uma vez que o algoritmo sugerido por ele é capaz de quebrar a segurança dos sistemas criptográficos assimétricos utilizados atualmente. A criptografia de curva elíptica (*Elliptic Curve Cryptography* - ECC) e o RSA, por exemplo, são algoritmos de chave pública baseados nos problemas matemáticos do logaritmo discreto e da fatoração de grandes números, respectivamente. Esses problemas, que são de difícil resolução nos computadores convencionais, podem ser facilmente solucionados por um computador quântico, com o algoritmo de Shor.

Diante desse cenário, em dezembro de 2016, o NIST começou um processo público para avaliar e definir um novo padrão para algoritmos de chave pública com criptografia pós-quântica (*Post-Quantum Cryptography* - PQC), que é a criptografia que permanece segura mesmo diante do surgimento e da utilização dos computadores quânticos para a realização de ataques. Esse processo do NIST tem como objetivo estabelecer algoritmos que possam ser utilizados para cifrações, assinaturas digitais e em mecanismos de encapsulamento de chaves (*Key Encapsulation Mechanism* - KEM).

Segundo o documento de chamada para propostas do processo de padronização de criptografia pós-quântica [14], haviam algumas restrições para a aceitação da submissão dos algoritmos candidatos, sendo que uma delas era a utilização de componentes considerados inseguros contra computadores quânticos. Em relação a especificação dos algoritmos, que deveria estar inclusa no pacote de submissão, existia a

necessidade de listar suas vantagens e limitações, sua compatibilidade com as redes e os protocolos de comunicação existentes, a viabilidade de implementação em dispositivos com diferentes recursos, dentre outros. Os principais critérios utilizados para avaliação foram a segurança, a performance e as características de implementação, tais como simplicidade e flexibilidade.

Dos 82 algoritmos submetidos, 69 atendiam aos requisitos mínimos exigidos para a primeira rodada de avaliação. Desses, 26 passaram para a segunda rodada e apenas 7 foram considerados finalistas para a rodada seguinte. Em julho de 2022, com os resutados obtidos na terceira rodada, foram divulgados em [15] os quatro primeiros algoritmos selecionados para padronização , onde três deles (CRYSTALS-Dilithium, Falcon, and SPHINCS+) têm como funcionalidade as assinaturas digitais e apenas um (CRYSTALS-Kyber) foi escolhido para cifração e mecanismo de encapsulamento de chaves.

Uma quarta rodada de avaliação está em andamento, com a abertura para novas propostas de algoritmos para assinatura digital e com análises aprofundadas de outros quatro candidatos que foram previamente submetidos para funcionalidade de cifração e encapsulamento de chaves.

É notório que há uma grande preocupação em garantir a continuidade da segurança das informações diante do avanço da computação quântica. Se faz necessário que os estudos e as pesquisas na área de criptografia avancem, com o intuito de encontrar melhores e mais seguras ferramentas para atender aos requisitos exigidos pela nova tecnologia. É essencial o desenvolvimento de meios para facilitar a implementação da criptografia pós-quântica, fazendo com que ela possa estar inserida nos mais diversos ambientes e dispositivos.

Com o propósito de otimizar o desempenho dos algoritmos pós-quânticos, a utilização de *Field-Programmable Gate Array* (FPGA) se mostra uma opção viável para a aceleração da execução das funções desses algoritmos. Fazer uso da síntese de alto nível (*High-Level Synthesis* - HLS) facilita e reduz significativamente o tempo de desenvolvimento do código *Register Transfer Level* (RTL) a ser implementado na FPGA.

O estudo que segue contribui com o desenvolvimento de uma implementação eficaz, em FPGA de baixo custo, do algoritmo pós-quântico CRYSTALS-Kyber, definido pelo NIST como padrão para encapsulamento de chaves. Utilizando ferramentas HLS, foi possível desenvolver um acelerador para as funções de cifração e decifração que faz uso de 33733 *Look-Up Tables* (LUTs), 22810 *Flip-Flops* (FFs) e 151 *Digital Signal Processors* (DSPs), executando essas operações em 5,01 e 2,24 milissegundos, respectivamente.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O objetivo geral dessa pesquisa é fornecer um acelerador com implementação de criptografia pós-quântica em FPGA de baixo custo, utilizando o mecanismo de encapsulamento de chaves CRYSTALS-Kyber, que permita uma aceleração da execução das operações criptográficas do algoritmo, com a utilização da síntese de alto nível para o desenvolvimento da descrição RTL.

1.1.2 Objetivos específicos

- Detalhar a FPGA escolhida para o desenvolvimento do trabalho, em relação aos recursos disponíveis para implementação dos algoritmos do mecanismo Kyber;
- Descrever as operações criptográficas e o embasamento matemático do mecanismo de encapsulamento de chaves CRYSTALS-Kyber;
- Desenvolver o código RTL do acelerador em ferramentas específicas para desenvolvimento de síntese de alto nível;
- Sintetizar na FPGA o código desenvolvido e coletar os resultados obtidos para avaliar a eficácia do acelerador proposto;

1.2 PRINCIPAIS CONTRIBUIÇÕES

Esse estudo traz uma implementação dos algoritmos do mecanismo de encapsulamento de chaves pós-quântico CRYSTALS-Kyber em uma FPGA de baixo custo. Além do detalhamento do Kyber, com a apresentação do embasamento matemático de seus algoritmos, é possível encontrar nessa dissertação explicações a respeito do funcionamento e da composição da FPGA PYNQ-Z1. É apresentado ainda o código HLS que deu origem à arquitetura RTL utilizada para configuração do acelerador desenvolvido com os algoritmos do Kyber.

Pode-se destacar também a publicação do artigo científico [16] intitulado "Quantum-resistant Cryptography in FPGA" nos anais do *2022 Workshop on Communication Networks and Power Systems* (WCNPS), que foi realizado na Universidade Federal do Ceará (UFC), em Fortaleza/CE, nos dias 17 e 18 de novembro de 2022.

1.3 ORGANIZAÇÃO

A pesquisa que segue está organizada da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, com conceitos e definições necessárias para o entendimento do estudo, e os principais trabalhos relacionados desenvolvidos; o Capítulo 3 descreve a construção proposta para o acelerador Kyber; os resultados obtidos com o acelerador desenvolvido são apresentados no Capítulo 4; por fim, são apresentadas as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS CORRELATOS

As seções seguintes apresentam definições necessárias para a compreensão dessa dissertação. São abordados conceitos matemáticos e criptográficos que servem como base para a construção e a definição da segurança dos algoritmos que compõem o mecanismo de encapsulamento de chaves CRYSTALS-Kyber. Na Seção 2.2 é apresentado o detalhamento desse mecanismo, no qual esse estudo é baseado. Noções a respeito de arquiteturas de alto desempenho e síntese de circuitos são apresentadas nas seções que seguem. Por fim, tem-se os resultados encontrados nos principais trabalhos relacionados, especificando as limitações e contribuições para essa área de pesquisa.

2.1 FUNDAMENTOS CRIPTOGRÁFICOS E MATEMÁTICOS

2.1.1 Cifração e decifração

O processo de cifração e decifração de dados é um dos princípios mais importantes da criptografia. Ele auxilia na salvaguarda dos dados, seja no armazenamento local ou na transferência de informações digitais através das redes de computadores, como aponta Alenezi et al. em [17].

É por meio da cifração que o texto em claro se transforma em texto cifrado, tornando-se ininteligível, visando garantir que apenas pessoas autorizadas tenham acesso ao conteúdo da mensagem. A decifração realiza a operação inversa, recuperando o texto original a partir do texto cifrado.

Para realizar a cifração e decifração de dados, os algoritmos criptográficos fazem uso de chaves. Como apresentado em [18], dependendo do modo de distribuição das chaves que utilizam, os algoritmos podem ser classificados em simétricos (chave secreta) ou assimétricos (chave pública).

Os algoritmos de chave simétrica são aqueles que utilizam a mesma chave no processo de cifração e decifração. O remetente e o destinatário devem compartilhar uma chave em comum para realizar a comunicação, conforme exemplificado na Figura 2.1.

A Figura 2.2 traz um exemplo de uso dos algoritmos de chave assimétrica, que possuem chaves distintas para os processos de cifração e decifração. O par de chaves pública e privada utilizadas por esses algoritmos são matematicamente relacionadas.

Na criptografia simétrica a chave precisa ser compartilhada por meio de um canal seguro, pois o vazamento da chave facilitaria o acesso de pessoas não autorizadas às informações e comprometeria a segurança da comunicação. A criptografia assimétrica, considerada mais segura, permite o estabelecimento de comunicação sem a necessidade do canal seguro, uma vez que não há o compartilhamento da chave, como exposto em [19].

Embora a criptografia assimétrica sejam considerada mais segura, sua utilização se torna complexa e

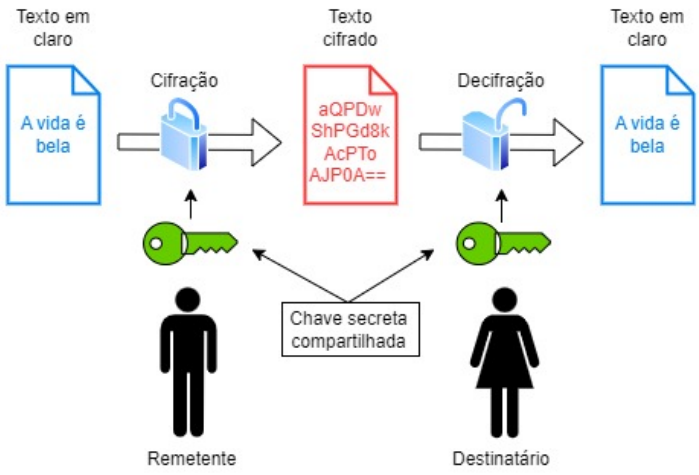

Figura 2.1: Esquema de cifração com algoritmo simétrico. (Fonte: o autor)

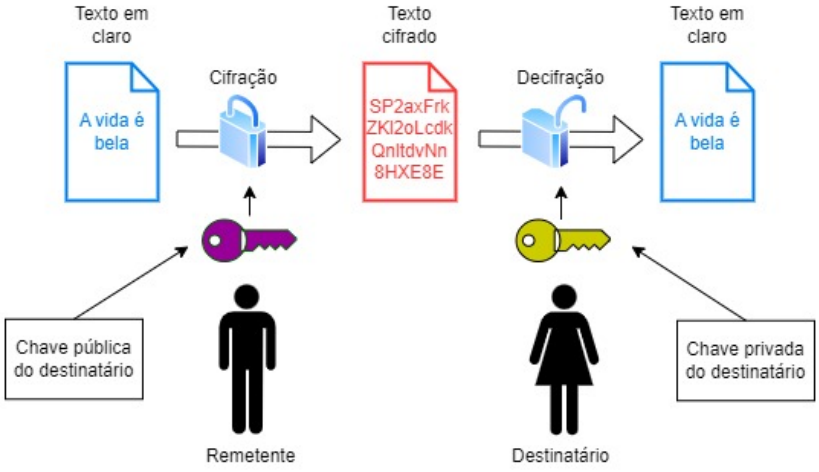

Figura 2.2: Esquema de cifração com algoritmo assimétrico. (Fonte: o autor)

inviável para uma grande quantidade de dados. Em [20], tem-se que o tempo de processamento e recursos computacionais gastos por algoritmos de chave pública em operações de cifração e decifração são maiores em comparação com os resultados obtidos pelos algoritmos de chave simétrica.

2.1.2 Esquemas criptográficos de chave pública - PKE e KEM

De forma geral, a cifração de chave pública (*Public-Key Encryption* - PKE) consiste em um esquema criptográfico onde uma mensagem é cifrada pelo remetente com a chave pública (*Public Key* - P_k) do destinatário enquanto a decifração é realizada pelo destinatário com sua chave privada (*Secret Key* - S_k). Conforme descrito em [21], esse esquema é composto pelos três algoritmos que seguem:

- Geração de chaves (*Key Generation* - PKE.KeyGen()): algoritmo probabilístico que retorna um par de chaves P_k e S_k;
- Cifração (*Encryption* - PKE.Enc()): de forma geral, esse algoritmo, que pode ou não ser probabi-

lístico, recebe a chave pública P_k e uma mensagem m como parâmetros de entrada e produz um cifrado c;

- Decifração (*Decryption* - PKE.Dec()): algoritmo determinístico que tem como parâmetros de entrada a chave secreta S_k e o texto cifrado c, resultando na mensagem m como saída.

A Figura 2.3 ilustra uma simulação de troca de mensagem utilizando os algoritmos da cifração de chave pública. Alice executa o algoritmo PKE.Keygen() e obtém o par P_k e S_k. Bob gera uma mensagem m e executa o algoritmo PKE.Enc() com m e P_k de Alice, como parâmetros de entrada, para obter o texto cifrado c. Com o cifrado c e sua chave S_k, Alice consegue obter m utilizando o algoritmo PKE.Dec().

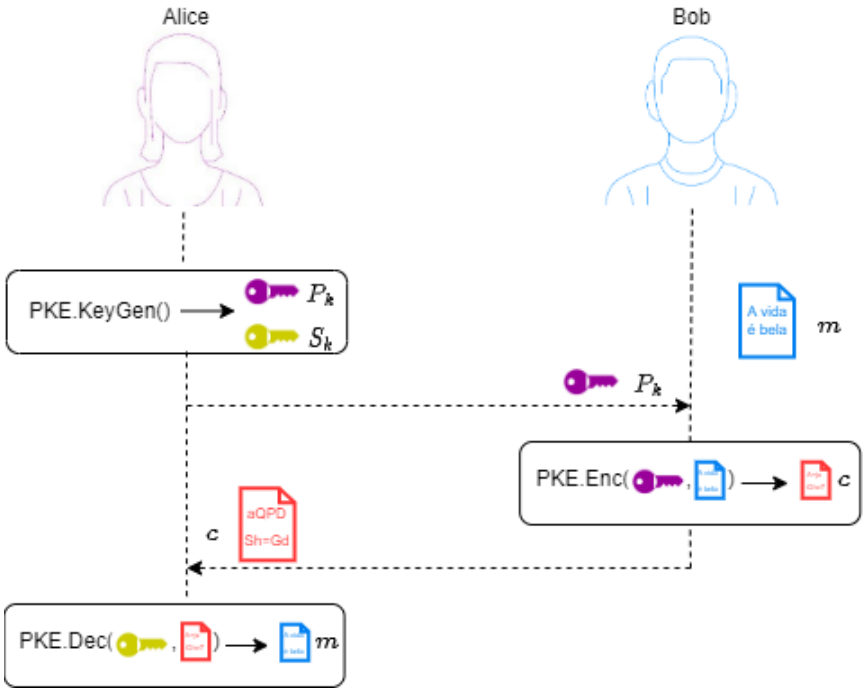

Figura 2.3: Protocolo de troca de mensagem, entre Alice e Bob, utilizando algoritmos PKE. (Fonte: o autor)

Shoup, em [22], define o mecanismo de encapsulamento de chaves (*Key Encapsulation Mechanism* - KEM) como um protocolo que utiliza criptografia de chave pública para encapsular uma chave simétrica. Esse mecanismo é utilizado como a primeira etapa em sistemas criptográficos híbridos, que usam a segurança da criptografia assimétrica para o compartilhamento da chave a ser utilizada posteriormente em algoritmos simétricos, que possuem um menor tempo de processamento das operações de cifração e decifração. Os algoritmos do KEM podem ser descritos como:

- Geração de chaves (*Key Generation* - KEM.KeyGen()): tem como função gerar um par de chaves pública P_k e privada S_k;
- Encapsulamento (*Encapsulation* - KEM.Enc()): o algoritmo de encapsulamento recebe como parâmetro de entrada a chave pública P_k e entrega na saída uma chave K e um cifrado c;
- Desencapsulamento (*Decapsulation* - KEM.Dec()): nesse algoritmo, os parâmetros de entrada são a chave secreta S_k e o cifrado c e retorna no fim a chave K.

A Figura 2.4 mostra um protocolo simplificado de compartilhamento de chaves utilizando as funções de um KEM. Alice executa o algoritmo KEM.KeyGen() e obtém o par P_k e S_k. Tendo a chave pública de Alice, Bob executa o KEM.Enc() para obter a chave K e o cifrado c, a ser enviado à Alice. Com o cifrado c e sua chave privada S_k, Alice consegue obter K por meio do algoritmo KEM.Dec(). Ao fim do processo, Alice e Bob possuem a mesma chave K que pode ser usada para comunicação segura utilizando algoritmos simétricos.

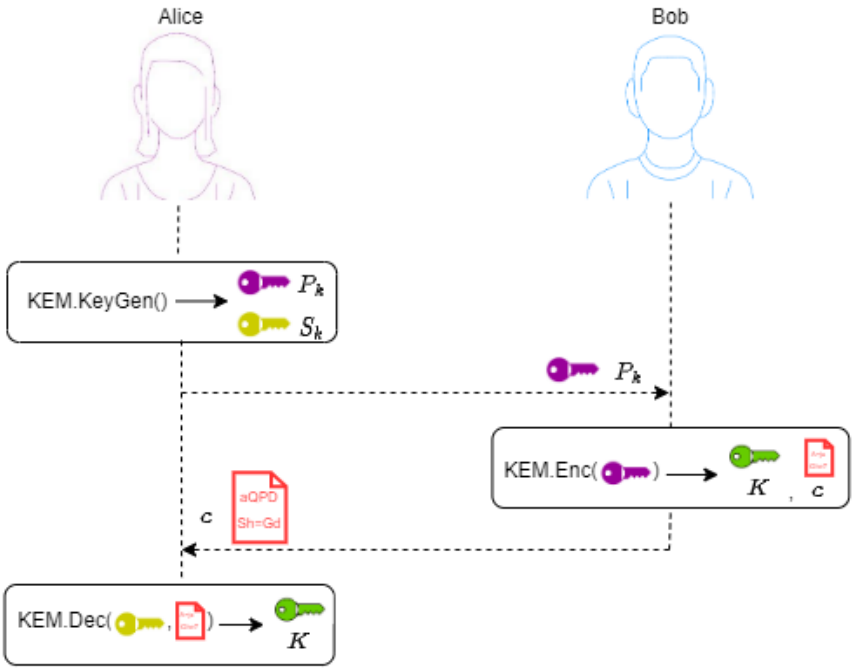

Figura 2.4: Protocolo de troca de chave, entre Alice e Bob, utilizando algoritmos KEM. (Fonte: o autor)

Conforme as definições acima, é possível observar que os esquemas criptográficos PKE e KEM são similares e ambos fazem uso de par de chaves pública e privada em seus algoritmos. A diferença entre eles é que o algoritmo de encapsulamento do KEM tem como função intrínseca a geração de uma chave aleatória K e a cifração da mesma, não recebendo nenhum parâmetro além da chave pública do destinatário, enquanto que no PKE o algoritmo de cifração tem a necessidade de receber como parâmetro a mensagem a ser cifrada, não sendo necessariamente uma chave.

Em [23], Cramer e Shoup afirmam que, embora seja possível usar os algoritmos do PKE para o compartilhamento de chaves, gerando uma chave aleatória separadamente e cifrando-a com PKE, utilizar os algoritmos do KEM é uma forma mais eficiente de realizar esse processo.

2.1.3 Noções de segurança em criptografia de chave pública

As noções clássicas de segurança em esquemas criptográficos de chave pública, apontadas por Bellare et al. em [24], podem ser divididas em dois objetivos e três modelos de ataque. Cada definição de segurança é constituída por um par, com a combinação de um objetivo e um modelo de ataque.

Como objetivos principais, são considerados:

- Indistinguibilidade (*Indistinguishability* - IND): introduzida por Goldwasser e Micali, em [25], a indistinguibilidade consiste na incapacidade de um adversário obter qualquer informação sobre um texto em claro m, tendo um texto cifrado c como referência,

- Não-maleabilidade (*Non-Malleability* - NM): definida em [26], a não-maleabilidade parte do pressuposto de que dado um texto cifrado c, gerado a partir de um texto em claro m, o adversário é incapaz de gerar outro texto cifrado c' de tal modo que sua decifração m' seja significativamente relacionada a m.

Entre os modelos de ataques, tem-se os três abaixo definidos, sendo relacionados pelo grau de sua força, do menor para o maior:

- Ataque por texto em claro escolhido (*Chosen-Plaintext Attack* - CPA): nesse caso de ataque, o adversário conhece a chave pública P_k e pode escolher textos em claro para obter os respectivos cifrados.

- Ataque por texto cifrado escolhido (*Chosen-Ciphertext Attack* - CCA): estabelecido por Naor e Yung em [27], as circunstâncias para esse ataque consistem em o adversário conhecer a chave pública P_k e possuir acesso ao oráculo de decifração[1] até receber o cifrado c que ele é desafiado a decifrar.

- Ataque adaptativo por texto cifrado escolhido (*adaptive Chosen-Ciphertext Attack* - CCA2): nesse ataque, formalizado por Rackoff e Simon em [28], além de possuir a chave pública P_k, o adversário pode consultar o oráculo de decifração[1] a qualquer tempo, com a restrição de não poder solicitar a decifração do cifrado c que tenha recebido como desafio.

Com base nessas explanações, é possível obter as definições de segurança IND-CPA, IND-CCA, IND-CCA2, NM-CPA, NM-CCA e NM-CCA2. Bellare et al. afirmam em [24] que a IND-CCA2 é a noção que deve ser considerada no desenvolvimento de protocolos criptográficos de chave pública. Se um protocolo tem a segurança IND-CCA2, implica que também terá as demais noções de segurança acima relacionadas.

As relações entre essas noções são demonstradas na Figura 2.5. As setas são as implicações e as setas com traços são as separações provadas em [24]. Se $A \rightarrow B$, diz-se que se um esquema de cifração possui a noção de segurança A, então também terá a noção B. Se $A \nrightarrow B$, significa que se o esquema tem noção de segurança A, então não terá necessariamente a noção B.

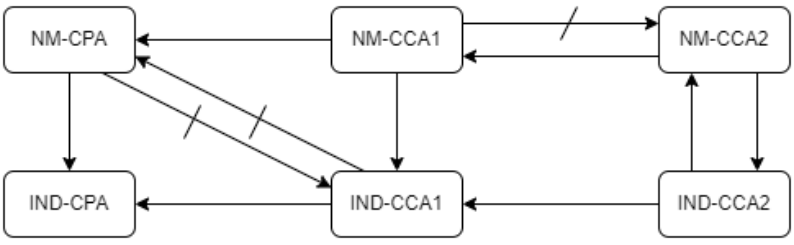

Figura 2.5: Relação entre as noções de segurança para esquemas de cifração de chave pública. (Fonte: adaptada de [24])

[1]Oráculo de decifração: consiste em uma função que fica a disposição do adversário e retorna o texto em claro resultante da decifração de qualquer cifrado de sua escolha, com exceção do texto cifrado do desafio.

Essas definições de segurança são avaliadas através de desafios. Para essa dissertação, é de interesse o detalhamento da indistinguibilidade sob ataque por texto em claro escolhido (IND-CPA) e da indistinguibilidade sob ataque adaptativo por texto cifrado escolhido (IND-CCA2).

No desafio IND-CPA o desafiante gera um par de chaves P_k e S_k e envia a primeira chave para o adversário, mantendo a segunda em sigilo. Possuindo a chave pública, o adversário pode cifrar quantos textos em claro ele quiser e obter os respectivos cifrados. O adversário computa duas mensagens m_0 e m_1, de mesmo tamanho, e envia para o desafiante que escolhe aleatoriamente uma das mensagens para cifrar e devolve o resultado c ao adversário. O adversário vence o desafio se descobrir se c provém da cifração de m_0 ou m_1.

Um sistema criptográfico de chave pública é considerado IND-CPA seguro se o adversário não conseguir vencer o desafio acima descrito com probabilidade significativamente maior que $1/2$.

Similarmente ao desafio anterior, no desafio IND-CCA2, o desafiante gera um par de chaves P_k e S_k, envia P_k para o adversário e mantém S_k em segredo. O adversário entrega duas mensagens m_0 e m_1 de mesmo tamanho para o desafiante, que escolhe aleatoriamente uma das mensagens para cifrar e devolve o resultado c ao adversário. A qualquer momento o adversário pode solicitar ao oráculo de decifração[1] os textos em claro correspondente aos cifrados que ele escolher, com exceção do texto cifrado c. Se descobrir que c é resultado da cifração de m_0 ou m_1, o adversário vence o desafio.

Caso o adversário não consiga vencer o desafio acima descrito com probabilidade significativamente maior que $1/2$, o sistema criptográfico de chave pública é considerado IND-CCA2 seguro.

2.1.4 Modelo do Oráculo Aleatório Clássico e Quântico

O modelo do oráculo aleatório (*Random Oracle Model* - ROM), introduzido por Bellare e Rogaway em [29], é um modelo teórico utilizado para provar a segurança de algoritmos e protocolos criptográficos. Em [30], Bleumer define o ROM como uma forma de modularizar provas de segurança, avaliando se o algoritmo ou protocolo criptográfico é seguro considerando que as funções pseudo-aleatórias utilizadas por ele sejam substituídas por um oráculo aleatório que, ao ser acionado, retorna valores verdadeiramente aleatórios.

Bellare e Rogaway em [29] definem um oráculo aleatório R como um mapeamento de $\{0,1\}^*$ para $\{0,1\}^\infty$ escolhido, selecionando cada bit de $R(x)$ uniformemente e independentemente para cada x, onde $\{0,1\}^*$ denota o espaço de cadeias binárias finitas e $\{0,1\}^\infty$ o espaço infinito.

A Figura 2.6 ilustra um oráculo aleatório que pode ser imaginado como uma caixa preta que recebe como entrada uma sequência de bits x e, quando solicitado, responde com uma sequência de bits $R(x)$, sem que seja conhecida a função interna por ele executada.

O resultado obtido no modelo do oráculo aleatório é a garantia de que o protocolo criptográfico é seguro a depender da forma como ele faz suas escolhas aleatórias, mesmo que o adversário tenha acesso ao oráculo aleatório. Na prática, substitui-se o oráculo aleatório por funções hash, que são funções determinísticas que retornam valores indistinguíveis de cadeias verdadeiramente aleatórias sob determinadas condições, que foram explicitadas por Preneel em [31]. Ao fazer essa substituição, assume-se que em um ataque o

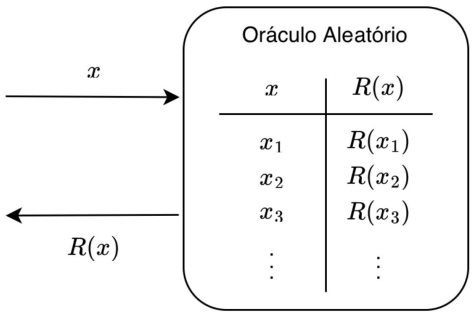

Figura 2.6: Oráculo aleatório visto como uma caixa preta. (Fonte: o autor)

adversário não se beneficia de nenhuma fraqueza que possa existir na função hash utilizada no protocolo ou algoritmo criptográfico.

No artigo [32], publicado em 2011, Boneh et al. introduziram o conceito do modelo do oráculo aleatório quântico (*Quantum Random Oracle Model* - QROM), de modo similar ao ROM clássico. Ele é utilizado para provar que o protocolo criptográfico é resistente a ataques quânticos, diante do cenário em que o adversário tem acesso quântico ao oráculo aleatório.

2.1.5 Transformação Fujisaki-Okamoto

No desenvolvimento de protocolos criptográficos, é desejável que a noção de segurança alcançada seja IND-CCA2, por ser a mais segura e implicar que as demais noções de segurança também serão atendidas. Porém, provar que um protocolo possui essa segurança é mais difícil do que provar a segurança IND-CPA, conforme afirma Hofheinz et al. em [33]. Com isso, várias transformações foram sugeridas a fim de fazer com que esquemas de cifração de chave pública com noções de segurança fracas se tornem IND-CCA2 seguros.

Uma dessas transformações foi proposta por Fujisaki e Okamoto, em [34]. Essa é uma transformação genérica que dá origem a um esquema de cifração assimétrico IND-CCA2 seguro, no modelo do oráculo aleatório, por meio da integração de esquemas de cifração simétrico e assimétrico que possuem fracas noções de segurança.

O estudo apresentado em [33] apresenta uma variação da transformação Fujisaki-Okamoto. Hofheinz et al. mostram que é possível obter, por meio de transformações em módulos, um mecanismo de encapsulamento de chaves IND-CCA2 seguro a partir de um esquema de cifração de chave pública IND-CPA seguro.

A figura 2.7 mostra, de forma simplificada, a construção dos algoritmos de um mecanismo de encapsulamento de chaves IND-CCA2 seguro para uma das variações da transformada de Fujisaki-Okamoto proposta por Hofheinz et al. em [33]. Esses algoritmos utilizam as funções de geração, cifração e decifração de um PKE em conjunto com as funções de hash H e G.

Geração

1 (p_k, s_k) ← Geração
2 $s \leftarrow M$
3 s'_k:= (s_k, s)
4 retorna (p_k, s'_k)

Encapsulamento(p_k)

1 $m \leftarrow M$
2 c := Cifração(p_k, m; G(m))
3 K := H(m, c)
4 retorna (K, c)

Desencapsulamento(s'_k = (s_k, s), c)

1 m' := Decifração(s_k, c)
2 if c ≠ Cifração(p_k, m'; G(m')) ou $m' = \perp$
3 retorna K := H(s, c)
4 else retorna K := H(m', c)

Figura 2.7: Algoritmos do KEM IND-CCA2 seguro. (Fonte: adaptado de [33])

2.1.6 Reticulados

Micciancio e Regev, em [35], definem os reticulados como um conjunto de pontos em um espaço de dimensão n com uma estrutura periódica. A base de um reticulado é composta por n vetores linearmente independentes $\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n} \in \mathbb{R}^n$, que dão origem ao reticulado $\mathcal{L}(\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n})$ pelo conjunto de vetores definido por

$$\mathcal{L}(\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b_i} : x_i \in \mathbb{Z} \right\}.$$

O reticulado gerado por uma matriz é definido como $\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$, onde $\mathbf{B} = [\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}] \in \mathbb{R}^{n \times n}$ é a matriz com colunas compostas pelos vetores da base e $\mathbf{Bx}$ é a multiplicação entre matriz e vetor.

Dadas as definições acima, é possível constatar que existem várias bases que representam um mesmo reticulado. A Figura 2.8 exemplifica um reticulado de dimensão 2, com duas possíveis base.

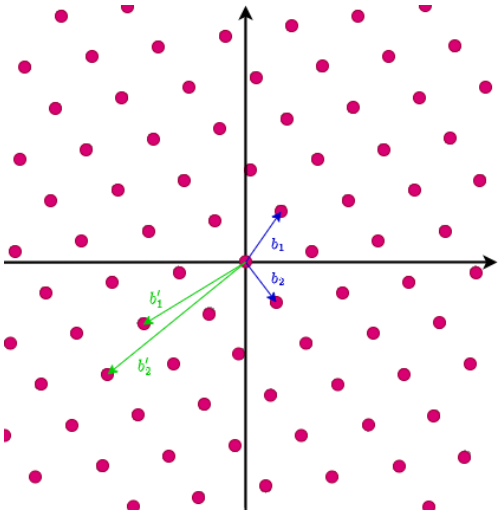

Figura 2.8: Exemplo de um reticulado de dimensão 2 com duas bases representadas. (Fonte: adaptado de [35])

Inicialmente pouco se conhecia sobre a complexidade dos problemas em reticulados. A popularização do uso de reticulados na área da criptografia se deu como ferramenta de criptoanálise, com a técnica da redução de reticulados. Essa técnica foi considerada, naquela ocasião, um problema de fácil solução e os algoritmos de redução passaram a ser utilizados para quebrar a segurança de esquemas criptográficos de chave púbica com sucesso, como apontam Nguyen e Stern, em [36].

Como na criptografia a dificuldade do problema a ser resolvido é o que dá a garantia da segurança dos esquemas criptográficos, se faz necessário que os esquemas desenvolvidos sejam baseados em problemas que tenham evidências de serem de difícil solução. Logo, os reticulados não eram vistos como opção para o desenvolvimento de protocolos criptográficos, já que eram considerados um problema fácil.

Estudos apresentados por Ajtai, em [37] e [38], mostraram resultados significativos sobre a complexidade dos problemas em reticulados. Além de fomentar outros estudos sobre a dificuldade dos problemas associados aos reticulados, os resultados obtidos por Ajtai viabilizaram o desenvolvimento de esquemas criptográficos seguros, tendo os reticulados como base.

Os problemas computacionais mais conhecidos em reticulados são:

- *Shortest Vector Problem* (SVP): busca encontrar o menor vetor não nulo em um dado reticulado;
- *Closest Vector Problem* (CVP): dados um reticulado e um vetor v (que pode ou não pertencer ao reticulado), deseja-se encontrar o ponto do reticulado que seja mais perto de v;
- *Shortest Independent Vector Problem* (SIVP): consiste em encontrar uma nova base que resulte no mesmo reticulado mas minimize o comprimento do vetor mais longo;
- *Short Integer Solution* (SIS): tendo vetores uniformemente aleatórios, busca-se encontrar um vetor não nulo de inteiros com coeficientes pequenos tal que a soma da combinação entre eles seja igual a zero;
- *Learning With Errors* (LWE): consiste em encontrar um vetor s dada uma sequência aleatória de equações lineares aproximadas em s, acrescidas de ruídos (erros).

Alguns dos problemas acima listados, tais como SIS e LWE, possuem variantes. Quando o problema de interesse e o reticulado são definidos sobre um anel, tem-se o *Ring Short Integer Solution* (RSIS) e o *Ring Learning With Error* (RLWE). Se eles forem definidos sobre um módulo, obtem-se o *Module Short Integer Solution* (MSIS) e o *Module Learning With Error* (MLWE). Como o mecanismo de encapsulamento de chaves Kyber é baseado no problema MLWE, mais informações e detalhamentos sobre esse problema são descritos na Seção 2.1.7.

Os esquemas criptográficos baseados nos problemas computacionais sobre reticulados entraram em evidência nos últimos anos pois eles são considerados seguros mesmo diante do desenvolvimento e da utilização dos computadores quânticos. Além disso, Nejatollahi et al. mostram em [39] que muitos deles são eficientes e de fácil implementação.

2.1.7 Problema LWE e suas variantes

Definido por Regev em [40], o problema LWE consiste em encontrar o vetor $\mathbf{s} \in \mathbb{Z}_q^m$ dado o par $(\mathbf{A}, \mathbf{b})$, onde $\mathbf{b} \leftarrow \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \pmod{q}$, com $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ e $\mathbf{e} \in \mathbb{Z}_q^m$ com uma distribuição de probabilidade χ. Esse problema é conhecido como o problema da busca.

Um segundo problema LWE, equivalente ao definido acima, é conhecido como problema de decisão. Ele consiste em distinguir equações lineares que possuem distribuição verdadeiramente uniforme de outras equações lineares aleatórias, que foram alteradas por um erro / ruído. Ou seja, dado $(\mathbf{A}, \mathbf{b})$, com $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ e $\mathbf{b} \in \mathbb{Z}_q^n$, determinar se $\mathbf{b} \leftarrow \mathbf{A} \cdot \mathbf{s} + \mathbf{e} \pmod{q}$ ou se $\mathbf{b}$ é escolhido em $\mathbb{Z}_q^n$ uniformemente ao acaso.

Regev afirma que a segurança de sistemas criptográficos de chave pública baseados em LWE decorre do fato de ser computacionalmente inviável realizar a distinção do problema de decisão ou a recuperação de s no problema da busca. No decorrer dos anos, o LWE se mostrou bastante versátil, servindo como base não só para cifração de chave pública, mas também para várias outras construções criptográficas.

Entretanto, Lyubashevsky et al. apontam, em [41], que a maior desvantagem do LWE é que ele tende a não ser eficiente para aplicações práticas, exigindo grande tempo computacional para sua execução e grandes tamanhos de chaves. Para contornar essa ineficiência, é definido o RLWE como solução.

O RLWE é similar ao LWE, porém ao invés de utilizar elementos inteiros, ele realiza suas operações com todas as suas variáveis sendo polinômios em algum anel[2] $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. Realizar operações de multiplicação polinomial em reticulados, ao invés de multiplicações entre matriz e vetor, como no LWE, aumenta a eficiência e reduz a complexidade.

Ainda que o RLWE ofereça melhor eficiência em termos de velocidade e tamanho das chaves, Langlois e Stehlé citam em [42] que há desvantagem do RLWE em comparação com o LWE, em termos de segurança, pois a inclusão da estrutura algébrica pode facilitar ataques maliciosos. Dessa forma, eles definem o MLWE como um equilíbrio entre o LWE e o RLWE.

O MLWE realiza operações em um módulo sobre um anel. Seu problema pode ser entendido como um problema RLWE com a dimensão do módulo maior que 1. Ou seja, estando os elementos do MLWE em R_q^d, se $d = 1$, então o problema MLWE é um RLWE. Sendo menos estruturado, o MLWE oferece um melhor nível de segurança, com uma menor chance de ataques, mas mantendo um desempenho similar ao do RLWE.

2.2 CRYSTALS-KYBER

O mecanismo de encapsulamento de chaves Kyber, que compõe o pacote *Cryptographic Suite for Algebraic Lattices* (CRYSTALS), teve sua descrição inicial apresentada em [43]. Sua segurança é baseada na dificuldade de resolver o problema *Learning With Errors* quando o reticulado é definido sobre um módulo (MLWE). Ele foi o primeiro KEM escolhido pelo NIST para ser padronizado como criptografia pós-quântica.

[2] Estrutura algébrica que possui duas operações, adição e multiplicação.

O Kyber é um mecanismo IND-CCA2 seguro, construído a partir de um esquema de cifração de chave pública IND-CPA seguro, que cifra mensagens de tamanho fixo, com 32 bytes. Para essa construção, foi utilizada uma das variações da transformação Fujisaki-Okamoto, explicitada na Seção 2.1.5.

Os algoritmos que compõem o KEM Kyber são descritos nos algoritmos 1, 2 e 3, sendo o algoritmo de geração de chaves, de encapsulamento e de desencapsulamento, respectivamente. É possível observar que, em sua construção, os algoritmos descritos integram as funções da cifração de chave pública (PKE) com as funções de hash G e H e com a *Key-Derivation Fuction* - (KDF).

Algoritmo 1: KEM.KeyGen()

Entrada: -
Saída: Chave pública : $p_k \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Chave privada: $s_k \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$

$z \leftarrow \mathcal{B}^{32}$
$(p_k, s'_k) :=$ PKE.KeyGen()
$s_k := (s'_k \parallel p_k \parallel \mathrm{H}(p_k) \parallel z)$
retorna (p_k, s_k)

Algoritmo 2: KEM.Enc(p_k)

Entrada: Chave pública: $p_k \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
Saída: Cifrado: $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Chave compartilhada: $K \in \mathcal{B}^*$

$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$
$(\bar{K}, r) := \mathrm{G}(m \parallel \mathrm{H}(p_k))$
$c :=$ PKE.Enc(p_k, m, r)
$K :=$ KDF($\bar{K} \parallel \mathrm{H}(c)$)
retorna (c, K)

Algoritmo 3: KEM.Dec(c, s_k)

Entrada: Cifrado: $c \in \mathcal{B}^{d_u \cdot k \cdot n/8+d_v \cdot n/8}$
Chave privada: $s_k \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
Saída: Chave compartilhada: $K \in \mathcal{B}^*$

$p_k := s_k + 12 \cdot k \cdot n/8$
$h := s_k + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z := s_k + 24 \cdot k \cdot n/8 + 64$
$m' :=$ PKE.Dec(s_k, c)
$(\bar{K}', r') := \mathrm{G}(m' \parallel h)$
$c' :=$ PKE.Enc(p_k, m', r')
se $c = c'$ **então**
    retorna $K := KDF\ (\bar{K}' \parallel H(c))$
senão
    retorna $K := KDF\ (z \parallel H(c))$
fim
retorna K

Os algoritmos utilizados nas especificações do Kyber foram retirados do padrão estabelecido pelo

NIST na publicação *Federal Information Processing Standards* 202 (FIPS-202) [10], sendo utilizado o SHA3-512 como a função de hash G, o SHA3-256 como a função de hash H e o SHAKE-256 como KDF.

Outras primitivas criptográficas da publicação FIPS-202 são utilizadas pelos algoritmos do PKE, são eles o SHAKE-128 como função de saída extensível (*Extendable Output Function* - XOF) e o SHAKE-256(s, b) como função pseudo-aleatória (*Pseudorandom Function* - PRF). A descrição dos algoritmos PKE podem ser encontradas em [44].

Foram apresentadas três versões para o mecanismo Kyber, de acordo com o nível de segurança definido pelo NIST em [14]. A Tabela 2.1 mostra as versões construídas, seus respectivos níveis de segurança e o algoritmo simétrico de comparação que o NIST adotou para a classificação. Como exemplo, observa-se que o Kyber512 foi classificado em nível de segurança 1 por exigir que qualquer ataque que quebre uma definição de segurança importante exija recursos computacionais iguais ou superiores aos necessários para uma busca de chaves no AES-128.

Tabela 2.1: Níveis de segurança das versões do KEM Kyber.

Versão	Kyber512	Kyber768	Kyber1024
Nível de Segurança	1	3	5
Correspondência NIST	AES-128	AES-192	AES-256

O que diferencia as versões do Kyber são os parâmetros n, k, q, η_1, η_2 d_u e d_v utilizados em seus algoritmos. Os valores inicialmente propostos em [43] foram atualizados durante o concurso de PQC do NIST. A Tabela 2.2 apresenta os valores finais que foram adotados para esse conjunto de parâmetros em cada uma das versões do Kyber, com base na documentação oficial disponibilizada em [44]. A definição dos parâmetros é a que segue:

- n: número de bits de entropia da chave encapsulada;
- k: determina a dimensão do reticulado;
- q: número primo pequeno que satifaz a condição $n \mid (q-1)$, necessário para agilizar a multiplicação polinomial;
- η_1, η_2, d_u, d_v: definidos para estabelecer o equilíbrio entre a segurança, o tamanho do texto cifrado e a probabilidade de falha δ dos algoritmos.

Tabela 2.2: Conjuntos de parâmetros utilizados pelos algoritmos das versões do KEM Kyber.

Versão	n	k	q	η_1	η_2	(d_u, d_v)	δ
Kyber512	256	2	3329	3	2	(10, 4)	2^{-139}
Kyber768	256	3	3329	2	2	(10,4)	2^{-164}
Kyber1024	256	4	3329	2	2	(11,5)	2^{-174}

Todos os conjuntos de parâmetros das versões do Kyber apresentam uma probabilidade de falha na decifração. Como essas falhas são motivo de preocupação para a segurança, a chance delas ocorrerem deve ser pequena. Os valores observados no parâmetro δ, da Tabela 2.2, mostram a probabilidade de falha

no desencapsulamento de um texto cifrado válido, que contém uma chave encapsulada. Essa probabilidade de falha é menor para a versão do algoritmo de maior nível de segurança.

Os diferentes parâmetros das versões do Kyber também interferem nos tamanhos das principais variáveis que seus algoritmos retornam, são eles: o par de chaves pública e privada e o arquivo cifrado. A Tabela 2.3 mostra o tamanho em bytes dessas principais variáveis para cada versão do algoritmo. Pode-se notar que quanto maior o nível de segurança da versão, maior será o tamanho de suas variáveis.

Tabela 2.3: Tamanho das variáveis retornadas pelos algoritmos do KEM Kyber, em bytes.

Variável	Kyber512	Kyber768	Kyber1024
Chave privada - S_k	1632	2400	3168
Chave pública - P_k	800	1184	1568
Cifrado - c	768	1088	1568

2.3 ARQUITETURAS DE ALTO DESEMPENHO

2.3.1 Arquitetura von Neumann

Os primeiros computadores desenvolvidos eram máquinas de programa fixo, feitos para realizar uma tarefa específica. Para que eles resolvessem um problema diferente do inicialmente projetado, era necessário que um complexo e demorado processo fosse efetuado de forma manual, como citado em [45]. Para contornar esse problema, o conceito de programa armazenado foi apresentado por von Neumann na proposta de construção do computador EDVAC *(Electronic Discrete Variable Automatic Computer)* [46].

O conceito de programa armazenado teve o propósito de permitir que as intruções do programa sejam armazenadas na memória da mesma forma que os dados. Uma vez armazenadas, as instruções podem ser facilmente alteradas, permitindo a execução de diferentes tarefas em um curto espaço de tempo. Outra vantagem desse conceito, apontada por Eigenmann e Lilja em [47], é a de permitir que programas possam gerar outros programas, possibilitando a automatização de tarefas que, até então, eram realizadas manualmente.

Em 1946, von Neumann e outros pesquisadores iniciaram um projeto para o desenvolvimento do computador de programa armazenado IAS *(Institute for Advanced Study)*. Descrito na documentação [48] e [49], o projeto do IAS deu origem à arquitetura von Neumann e serviu como inspiração para a arquitetura de quase todos os computadores que foram projetados posteriormente. A arquitetura von Neumann, ilustrada pela Figura 2.9, é composta pelos seguintes componentes:

- Unidade Central de Processamento (*Central Processing Unit* - CPU) - sendo o componente principal da arquitetura, realiza interações com a memória e com os dispositivos de entrada e saída. É a unidade responsável por buscar instruções e dados armazenados na memória e coordenar a execução sequencial dessas instruções e o processamento dos dados. Ela é constituída por:
  - Unidade de Controle - interpreta e executa as instruções que são obtidas da memória, determinando a ordem em que elas serão executadas. Ela coordena todo o sistema, controlando a Unidade Lógica e Aritmética e a interface com os demais componentes do sistema.

  - Unidade Lógica e Aritmética (*Arithmetic Logic Unit* - ALU) - unidade capaz de operar dados binários, combinando-os e transformando-os por meio da realização de operações aritméticas básicas, tais como adição, subtração, multiplicação e divisão, e também de operações lógicas, como negação, conjunção e disjunção.

- Memória - a memória consiste em diversos espaços de armazenamento com quantidade de bits definidos, chamados de palavras. Cada palavra possui um endereço exclusivo que é utilizado para identificar a posição de um determinado conteúdo. Tanto os dados como as instruções são nela armazenados de forma binária.

- Dispositivos de entrada/saída - controlados pela unidade de controle, esses dispositivos possibilitam a interação do computador com o mundo exterior. É por meio dos dispositivos de entrada que programas e dados podem ser inseridos no sistema. Já os dispositivos de saída permitem que o sistema mostre resultados após a conclusão das instruções.

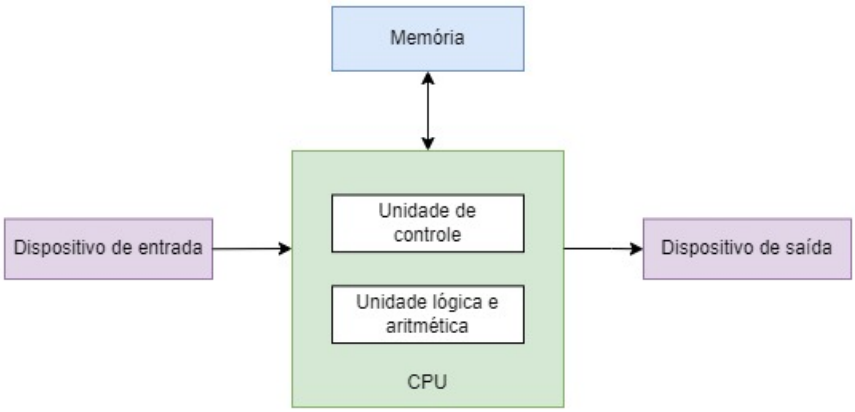

Figura 2.9: Componentes da arquitetura de von Neumann. (Fonte: adaptado de [45])

A arquitetura von Neumann é utilizada como base para a maioria dos computadores da atualidade. Ao longo dos anos essa arquitetura foi sendo otimizada para obter um maior desempenho computacional, potencializando o fluxo de execução das tarefas sequenciais para maximizar a vazão de instruções. Essa arquitetura proporciona um bom desempenho para a maioria dos programas de propósito geral.

2.3.2 Aceleradores em GPU

A Unidade de Processamento Gráfico (*Graphics Processing Unit* - GPU) foi desenvolvida com a finalidade de realizar operações em dados gráficos de forma paralela. Conforme explica Stallings em [50], ela é originalmente utilizada para otimizar a codificação e renderização gráfica bem como o processamento de vídeos. O aumento no realismo de gráficos e imagens de jogos foi um propulsor para o crescimento da capacidade de processamento das GPUs.

A CPU e GPU possuem arquiteturas e finalidades diferentes. A Figura 2.10 mostra a comparação entre os componentes da CPU e GPU em relação a área que ocupam. Na CPU a maior parte é ocupada pela unidade de controle e a memória cache. Na GPU tem-se a arquitetura de Única Instrução - Múltiplos Dados (*Single Instruction - Multiple Data* - SIMD), com foco principal na execução de operações matemáticas.

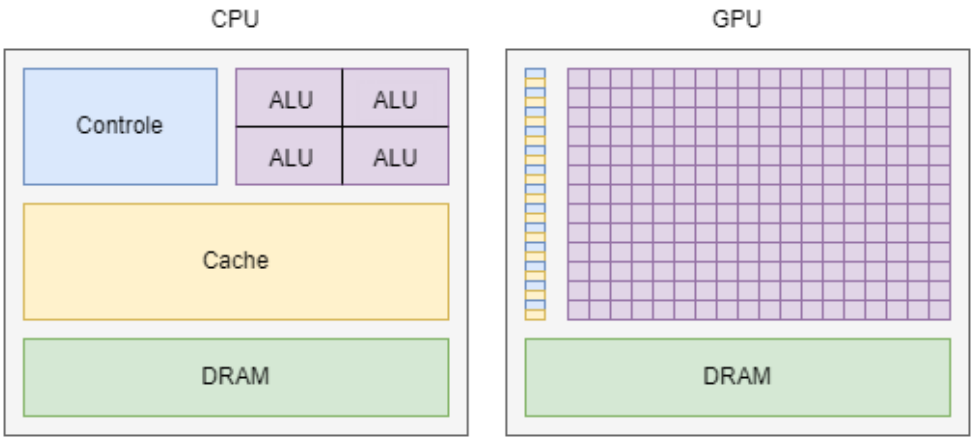

Figura 2.10: Comparação entre os componentes da CPU e GPU. (Fonte: adaptada de [50])

Enquanto que a CPU é versátil, com seu poder de processamento capaz de realizar diversas tarefas ordenadas de forma sequencial, a GPU tem seu processamento focado em possibilitar a resolução de problemas complexos dividindo-os em partes menores que podem ser resolvidas simultaneamente, com execuções em paralelo em seus vários núcleos. Brodtkorb et al., em [51], afirmam que o paralelismo é uma forma eficiente de aumentar o desempenho de aplicações que tenham trechos paralelizáveis em seu código.

O rápido aumento da capacidade de processamento juntamente com melhorias em sua programabilidade fizeram com que as GPUs modernas evoluíssem para um mecanismo eficiente no processamento de aplicações que, segundo Owens et al. [52], possuem como características: grandes requisitos computacionais, sejam altamente paralelizáveis e onde a vazão de operações seja prioritária em relação à latência. Essa constatação viabilizou a utilização da GPU para além de seu propósito de processamento gráfico, originando a chamada Unidade de Processamento Gráfico para Propósito Geral (*General-Purpose computing on the GPU* - GPGPU).

Dessa forma, é possível utilizar aceleradores desenvolvidos em GPUs para otimizar o desempenho de uma variedade de aplicações que exigem computações repetitivas, tais como inteligência artificial, processamento de áudio e de sinal, mineração de criptomoedas, machine learning, modelagem estatística, criptografia, dentre outras.

2.3.3 Aceleradores em FPGA

A computação reconfigurável ganhou espaço na computação moderna devido ao potencial de acelerar uma variedade de aplicações. Segundo Compton e Hauck, em [53], ela tem como característica a capacidade de otimizar o desempenho de operações realizando-as em hardware, mas mantendo grande parte da flexibilidade de uma solução em software. Os sistemas reconfiguráveis modernos são construídos pela combinação de um microprocessador de propósito geral e uma lógica reprogramável, atribuída tipicamente ao acrônimo FPGA (*Field Programmable Gate Array*)

Introduzida em meados dos anos 80, a tecnologia FPGA é um Dispositivo Lógico Programável (*Programmable Logic Device* - PLD) que pode ser reconfigurado pelo usuário. É por isso que chama-se *Field-*

Programmable, pois ela pode ser programada "em campo" (em ambiente de desenvolvimento, homologação e/ou produção), ou seja, não é programada na fábrica. Em [54], Brown et al. descrevem a FPGA como uma matriz bidimensional de Blocos Lógicos Configuráveis (*Configurable Logic Block* - CLB) conectados por recursos gerais de interconexão que, em conjunto, facilitam a implementação de um grande número de circuitos lógicos digitais. A FPGA conta ainda com uma estrutura de blocos de entrada/saída, conforme representado na Figura 2.11.

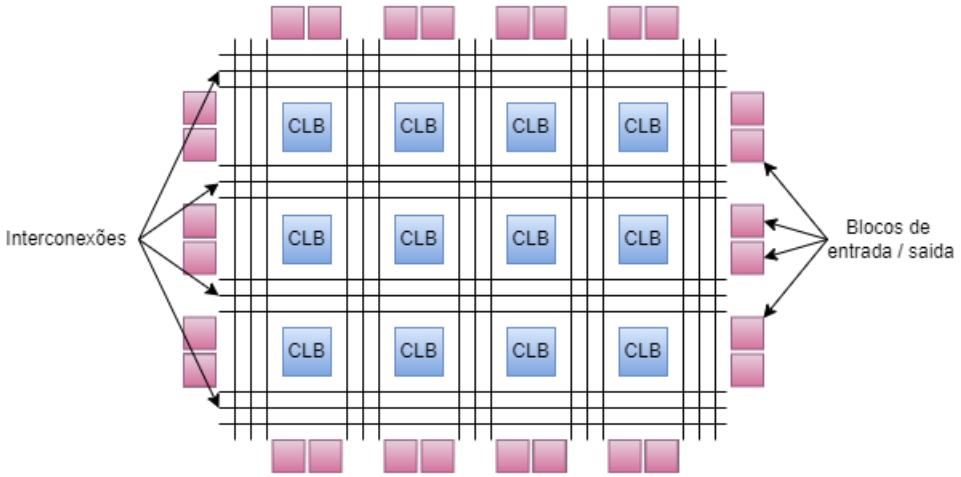

Figura 2.11: Arquitetura genérica de uma FPGA. (Fonte: adaptada de [54])

O bloco lógico configurável é a unidade lógica básica da FPGA que, segundo Kuon et al. [55], tem como objetivo fornecer a computação básica e os elementos de armazenamento utilizados em sistemas lógicos digitais. De forma geral, o CLB se comporta como uma tabela verdade, no qual é possível produzir determinada saída por meio de uma entrada determinada. Como descrito em [56], ele é compostos por três elementos básicos: *Look-Up Table* (LUT), *Flip-flop* (FF) e multiplexador (MUX). A Figura 2.12 mostra, de forma simplificada, a estrutura de um CLB.

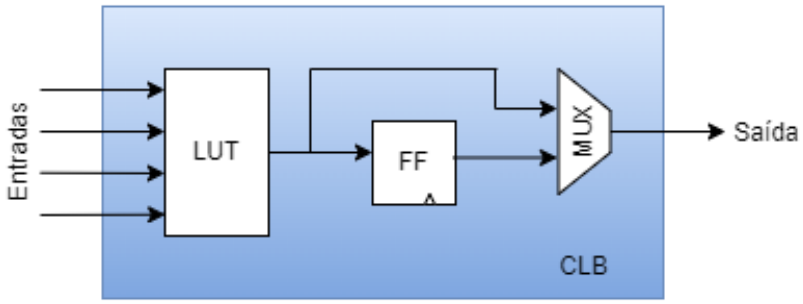

Figura 2.12: Exemplo simplificado de um CLB. (Fonte: adaptada de [55])

A implementação de funções lógicas, em cada CLB, é feita utilizando LUTs. Uma vez que as LUTs geram apenas lógica combinacional, onde a saída depende exclusivamente da entrada, são utilizados *flip-flops* para viabilizar a implementação da lógica sequencial, onde a saída depende do estado anterior. O multiplexador é utilizado para fazer a seleção da saída entre a lógica combinacional e sequencial.

Conforme apontam Kuon et al., em [55], os recursos gerais de interconexão (ou arquitetura de roteamento) são compostos por segmentos de fios e chaves programáveis. Essa arquitetura viabiliza a conexão

entre blocos de lógica configuráveis e entre blocos de entrada/saída, como descrevem Huffmire et al. em [57]. São estruturas também responsáveis pela distribuição dos pulsos de *clock*[3] por toda FPGA. Na Figura 2.13, tem-se a representação simplificada de uma arquitetura de interconexão em FPGA.

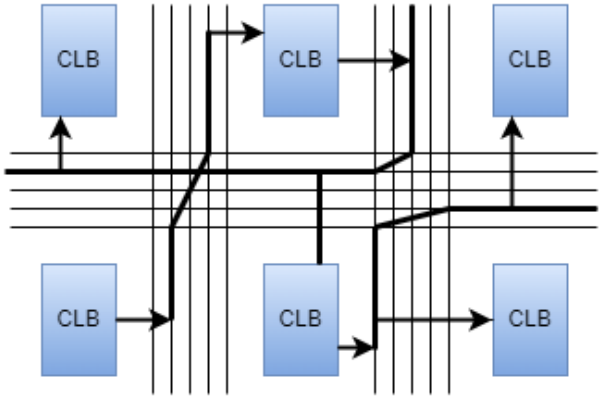

Figura 2.13: Arquitetura de interconexão FPGA. (Fonte: adaptada de [57])

O objetivo dos blocos de entrada/saída é fornecer interface do usuário para a arquitetura da FPGA. Em [55] tem-se que são essas estruturas que permitem a comunicação entre a FPGA e vários dispositivos externos, que possuem velocidades e tensões diferentes. É por meio desse blocos que sinais capazes de alterar as rotas de interconexão e o funcionamento lógico de cada CLB podem ser enviados.

Em [58], Boutros e Betz afirmam que, com o passar do tempo, os dispositivos FPGAs passaram a apresentar maior capacidade lógica, viabilizando a implementação de sistemas maiores. As necessidades desses sistemas fizeram com que as FPGAs se desenvolvessem e, em versões mais modernas, passassem a oferecer recursos mais complexos, tais como *Block Random Access Memory* (BRAM)[4] e *Digital Signal Processor* (DSP)[5].

A flexibilidade na arquitetura da FPGA viabiliza a construção de aceleradores para otimizar o desempenho da execução de aplicações, por meio da implementação de circuitos lógicos específicos para determinadas tarefas dessas aplicações.

Em um projeto, uma vez definidas as operações que serão realizadas pelo acelerador, é fundamental informar para a FPGA as configurações necessárias que devem ser carregadas em seus componentes. Como pontuado em [55], esse processo de programação da FPGA deve ser executado toda vez que a FPGA for energizada pois, na maioria dos dispositivos modernos, sua memória é volátil e essa informação não fica armazenada quando ocorre uma interrupção no fornecimento de energia ou quando ela é desligada.

Os dados de configuração que devem ser carregados na FPGA para a implementação da funcionalidade desejada são compilados em um arquivo binário (*bitstream*), criado por ferramentas de síntese de circuitos específicas para FPGA. Esse arquivo pode ser obtido a partir de uma descrição em Nível de Transferência entre Registradores (*Register Transfer Level* - RTL), descrita por meio de uma Linguagem de Descrição de Hardware (*Harware Description Language* - HDL). Os detalhes a respeito de HDL e RTL são abordados na seção de síntese de circuitos FPGA, nos itens 2.4.1 e 2.4.2, respectivamente.

[3] Sinal lógico utilizado para sincronizar ações de circuitos digitais.
[4] Blocos de memória utilizados para armazenar dados na FPGA.
[5] Blocos especializados em processamento de sinais digitais.

Das arquiteturas detalhadas na Seção 2.3, tem-se que as CPUs são excelentes para a execução de programas sequênciais, enquanto que a GPU tem seu foco no paralelismo de dados, mas com um alto consumo de energia. A FPGA aparece como uma opção intermediária em comparação às demais, pois sua customização possibilita a construção de arquiteturas com alto desempenho, mas com baixo consumo de energia. Como principais vantagens do uso de FPGAs para a construção de aceleradores é possível destacar ainda a sua possibilidade de reprogramação e a flexibilidade de sua arquitetura, que viabiliza sua utilização em aplicações das mais diversas áreas.

2.4 SÍNTESE DE CIRCUITOS FPGA

2.4.1 HDL

Inicialmente, os métodos clássicos para projetar circuitos integrados dependiam de esquemas e métodos manuais. Ciletti, em [59], pontua que com o crescimento do tamanho e da complexidade dos projetos essa prática se tornou inviável. Isso fez com que a utilização de Linguagem de Descrição de Hardware (HDL) se tornasse uma ferramenta importante para o projeto e gerenciamento desses circuitos.

As Linguagens de Descrição de Hardware permitem descrever as funcionalidades e características importantes de um sistema lógico na forma de um programa. Segundo Vahid, em [60], por meio dessas linguagens é possível descrever não só as interconexões estruturais entre componentes mas também o comportamento desses componentes. Como características das HDLs, pode-se citar:

- Possibilidade de descrever uma lógica complexa com códigos de descrição sucintos;
- Uma vez desenvolvidos, os códigos que descrevem um componente podem ser modificados ou reutilizados;
- Os códigos descritos são portáveis e independem do dispositivo no qual o projeto possa vir a ser sintetizado.

Em [61], Massoumi destaca que as HDLs mais comuns permitem a descrição da funcionalidade de um circuito digital em três diferentes níveis de abstração. Do nível mais baixo para o mais alto, são eles: o estrutural, onde se descreve o circuito em nível de portas lógicas, o de fluxo de dados, que faz a descrição em nível de transferência entre registradores (*Register Transfer Level* - RTL), e o comportamental, que é usado para descrever a funcionalidade do design sem detalhar sua implementação.

Na década de 80, o Departamento de Defesa dos Estados Unidos iniciou o programa de pesquisa *Very High Speed Integrated Circuits* (VHSIC). Esse programa tinha como objetivo a pesquisa e o desenvolvimento de circuitos eletrônicos de altíssima velocidade. Ele deu origem à linguagem de descrição de hardware VHDL (*VHSIC Hardware Description Language*) por conta da necessidade de criação de uma ferramenta de projeto e de documentação padrão que fosse compatível com equipamentos de diversos fornecedores. Ela foi a primeira HDL padronizada pelo *Institute of Electrical and Electronics Engineers* (IEEE 1076 [62]), no ano de 1987.

A linguagem Verilog HDL, que também foi desenvolvida nos anos 80, foi inicialmente definida como um produto de verificação e simulação propietário da empresa *Gateway Design Automation*. Anos mais tarde passou a ser utilizada como ferramenta de síntese comportamental e lógica de circuitos. Após ter se tornado pública, ela foi padronizada pelo *Institute of Electrical and Electronics Engineers* (IEEE 1364 [63]) no ano de 1995.

Dentre as diversas Linguagens de Descrição de Hardware existentes, pode-se citar a VHDL e a Verilog HDL como as mais utilizadas. Embora cada linguagem tenha particularidades em seu estilo, ambas podem ser utilizadas para modelar uma estrutura de hardware. Legíveis pela máquina e pelos humanos, elas são definidas, em suas respectivas documentações de padronização, como notações formais que suportam todas as fases da criação de sistemas eletrônicos: o desenvolvimento, a verificação, a síntese e o teste de projetos de hardware.

2.4.2 RTL

O Nível de Transferência entre Registradores (RTL) é definido em [64] como um nível de descrição que apresenta o comportamento do *clock* em termos de transferências de dados entre elementos de armazenamento em lógica sequencial e lógica combinacional. Nesse nível de abstração, o circuito é descrito em termos de registradores e a transferência de dados entre eles ocorre através da lógica combinacional. A Figura 2.14 ilustra a abstração RTL.

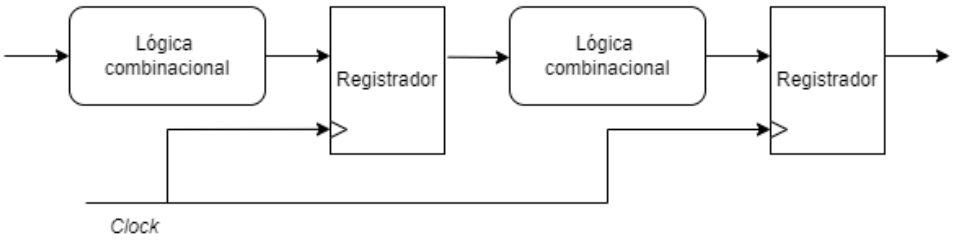

Figura 2.14: Representação do nível de abstração RTL. (Fonte: adaptada de [65])

Quando um código RTL é processado geralmente é apresentada sua visualização esquemática que, segundo Pedroni [65], é útil para verificar se o projeto está correto e, caso contrário, auxiliar na correção dos erros presentes no código. Nesse nível de abstração é possível realizar otimizações de desempenho, área ou potência por meio do controle da quantidade e do tipo de registradores e de portas lógicas que são utilizados no circuito.

A linguagem em nivel RTL é capaz de descrever as operações de sistemas síncronos. Em [66], Wakerly afirma que, sendo uma descrição clara das interconexões e operações lógicas, o código RTL é útil para o desenvolvimento de projetos em FPGA, pois define as instruções individuais em linguagem de máquina como sequência de etapas mais primitivas. Isso faz com que a arquitetura descrita em RTL seja sintetizável. As ferramentas de síntese são responsáveis por transformar o código RTL em um layout físico do circuito, em nível de portas lógicas, por meio da sintetização lógica.

O processo de síntese gera um *Netlist*, que é uma relação dos componentes do circuito, explicitando a forma como eles estão conectados. Em um fluxo de projeto, de forma simplificada, a etapa seguinte é

a implementação, onde são posicionados os elementos da *Netlist* na FPGA, considerando o fabricante e modelo do dispositivo escolhido. Concluídas essas etapas, é possível gerar o arquivo *bitstream* com as informações necessárias para configuração da FPGA com a funcionalidade desejada. A Figura 2.15 mostra o esquema desse fluxo de projeto que foi especificado.

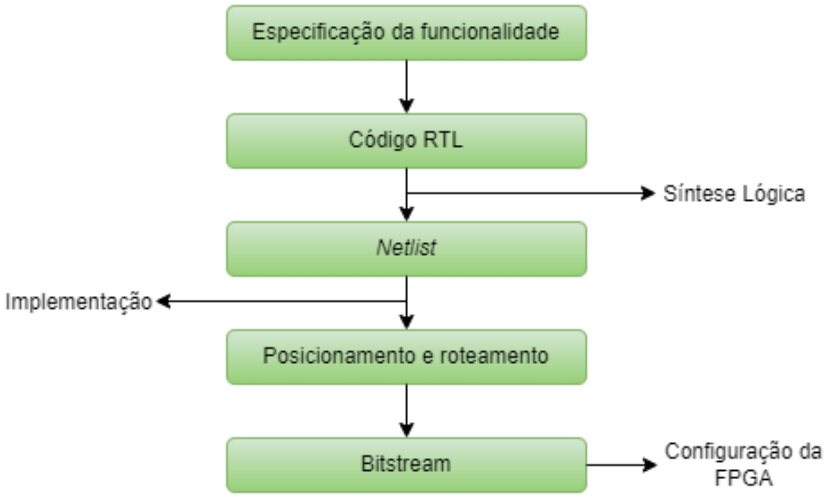

Figura 2.15: Fluxo de projeto simplificado para configuração de uma FPGA. (Fonte: o autor)

2.4.3 HLS

A Síntese de Alto Nível (*High-Level Synthesis* - HLS), conforme aponta Coussy et al. em [67], surge da necessidade de aumentar o nível de abstração para acelerar a automação dos processos de síntese e de verificação, dada a evolução da capacidade dos circuitos integrados e o aumento da complexidade de seus projetos.

Em [68], Nane et al. definem HLS como o processo que permite um alto nível de abstração ao utilizar um programa de software para especificar uma funcionalidade de hardware. As ferramentas HLS utilizam códigos desenvolvidos em determinadas linguagens de alto nível, tais como C, C++ ou *SystemC*, e produzem automaticamente a correspondente especificação do circuito em HDL. A especificação gerada é uma arquitetura RTL que viabiliza a implementação da funcionalidade em um dispositivo FPGA, por exemplo.

A Figura 2.16 mostra de forma simplificada o fluxo de projeto para configuração de uma FPGA utilizando HLS como metodologia, destacando o processo de obtenção do código RTL por meio da síntese de um código de linguagens de alto nivel. Uma vez que a arquitetura RTL é obtida, o fluxo ocorre da mesma forma que é especificado na Seção 2.4.2.

No desenvolvimento de um projeto de hardware em FPGA utilizando HDLs é necessário possuir conhecimento em eletrônica digital e há um gasto maior de tempo para a construção da descrição do projeto, aumentando também o seu custo. Por outro lado, utilizar HLS como método permite o desenvolvimento de projetos complexos de forma eficiente em um menor tempo, facilitando o processo de verificação, permitindo a realização de alterações de maneira mais simples e possibilitando a análise de várias alternativas de implementação para a funcionalidade desejada.

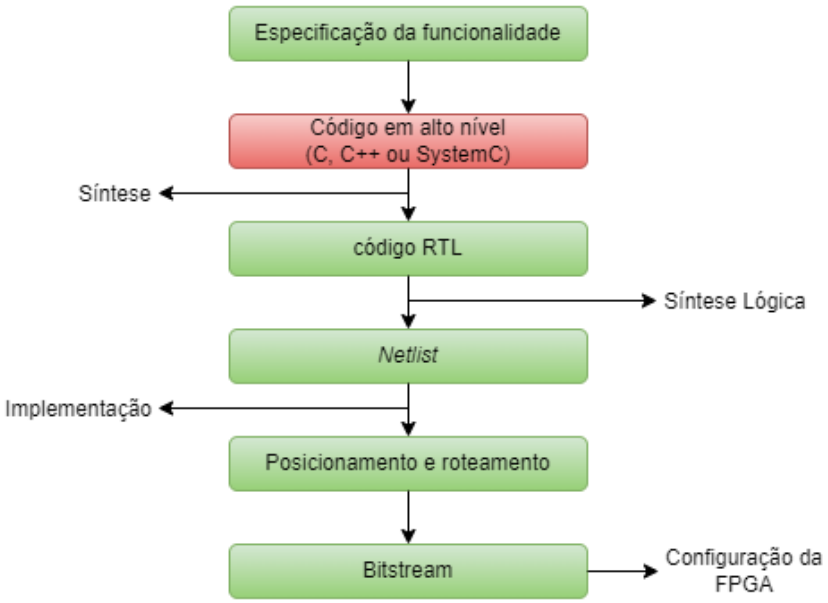

Figura 2.16: Fluxo de projeto simplificado para configuração de uma FPGA utilizando HLS. (Fonte: o autor)

Outro ponto que pode ser destacado como vantagem na utilização de HLS é a facilidade em realizar a portabilidade do código desenvolvido. Uma vez escolhida a FPGA a ser utilizada, a ferramenta HLS se encarrega de otimizar a arquitetura RTL a fim de aperfeiçoar a utilização dos recursos disponíveis na FPGA em questão.

Cong et al., em [69], apontam que a implantação da tecnologia HLS amadureceu significantemente na última década, possibilitando a utilização de ferramentas HLS para o desenvolvimento de aceleradores em FPGA nas mais diversas áreas, tais como *machine learning*, processamento de dados, bioinformática, criptografia, processamento de imagens e videos, dentre outras.

2.5 TRABALHOS CORRELATOS

O estudo publicado por Shor em [13] e os avanços no desenvolvimento do computador quântico fomentaram o interesse dos pesquisadores em procurar novas soluções criptográficas capazes de garantir a segurança das informações diante dessa nova tecnologia. A divulgação do processo do NIST para a padronização de algoritmos pós-quânticos fez com que os estudos nessa área se intensificassem nos últimos anos e os algoritmos que foram sendo aprovados nas fases desse processo ficaram em maior evidência.

Com relação ao Kyber, mecanismo de encapsulamento explorado nessa dissertação, observa-se um crescente número de publicações em relação a segurança, eficácia e melhorias em suas funções. Os estudos que apresentam projetos de aceleradores construídos em hardware se concentram principalmente em otimizações da função que possui um alto custo computacional para o Kyber, as suas operações de multiplicação polinomial.

Considerando implementações de funcionalidades completas para o KEM Kyber em dispositivos FPGA, tem-se os resultados que seguem, sendo apenas um utilizando as ferramentas de síntese de alto nível en-

quanto os demais são implementações em hardware desenvolvidos utilizando linguagem de descrição de hardware.

Uma comparação entre os candidatos da segunda fase do processo do NIST é apresentada por Basu et al. em [70]. A arquitetura dos algoritmos é desenvolvida utilizando HLS, com o uso da plataforma FPGA Virtex-7, e discute-se técnicas de otimizações para diminuir a latência dos algoritmos pós-quânticos. Para os resultados comparativos, foi utilizada a versão Kyber512, apontado como um dos algoritmos com menor latência tanto na cifração quanto na decifração.

Em [71] é apresentada uma implementação desenvolvida em Verilog, com maximização dos recursos e reutilização dos módulos comuns entre os processos de encapsulamento e desencapsulamento de chave. Foi utilizada a FPGA Virtex-7 (VC707 - XC7VX485T) com frequência de 192 MHz para a versão Kyber1024 e o modelo Xilinx Artix-7 (AC701 - XC7A200T), operando na frequência de 155 MHz, para as demais versões do Kyber. Houve uma aceleração de até 129 vezes para a cifração e decifração, em comparação com o desempenho desses algoritmos no processador Cortex-M4. Huang et al. apontam que a maior dificuldade da implementação em hardware é a grande quantidade de operações matemáticas, tais como as transformadas numéricas e as operações de deslocamento realizadas em várias matrizes.

Xing e Li, em [72], utilizaram a FPGA Xilinx Artix-7 de modelo XC7A12TCPG238-1, com frequência de 161 MHz, para implementação da arquitetura proposta e mostram que as operação que mais consumiram área do dispositivo foram as funções de hash e as transformações necessárias para a multiplicação de polinômios presentes nos algoritmos.

Outra proposta foi apresentada por Bisheh-Niasar et al. em [73] com a utilização da FPGA Xilinx Artix-7 XC7A100T-3 com frequência de 115 MHz. Com uma arquitetura de tempo constante, visando evitar ataques de canal lateral, os autores apresentaram um conjunto de instruções para otimizar a amostragem de polinômios, a transformação para multiplicação polinomial e também a multiplicação de vetores.

Guo et al. mostraram os resultados da implementação da arquitetura proposta por eles em uma FPGA Xilinx Artix-7 com 159MHz de frequência em [74]. O design apresentado é compatível para as três versões do Kyber. Eles afirmam que, utilizando o algoritmo de redução modular e os endereços de dados reconfiguráveis propostos, é possível reduzir o tempo de execução atingindo alto grau de paralelismo.

A Tabela 2.4 mostra o resumo dos principais resultados encontrados nas pesquisas acima descritas, explicitando as funções que foram implementadas e os recursos utilizados para cada arquitetura desenvolvida.

Tabela 2.4: Síntese dos resultados obtidos nos trabalhos relacionados.

Trabalhos correlatos	Funções implementadas	HLS	LUT	FF	DSP	BRAM	Frequência (MHz)
Basu et al. [70][1]	KEM.Enc / KEM.Dec	Yes	1977896	194126	-	-	-
Huang et al. [71]	KEM.Enc / KEM.Dec	No	110260	-	292	202	155
Xing e Li [72]	KEM.KeyGen / KEM.Enc / KEM.Dec	No	7412	4644	2	3	161
Bisheh-Niasar et al. [73]	KEM.KeyGen / KEM.Enc / KEM.Dec	No	16000	6000	9	16	115
Guo et al. [74]	KEM.KeyGen / KEM.Enc / KEM.Dec	No	7900	3900	4	16	159

[1] versão Kyber512

O acelerador proposto e detalhado no Capítulo 3 mostra a implementação em dispositivo FPGA de uma arquitetura desenvolvida com HLS, com o propósito de otimizar a execução dos algoritmos do Kyber,

em sua versão Kyber768. Até o momento, não foram encontrados na literatura trabalhos similares, que tenham adotado ferramentas HLS para o desenvolvimento da arquitetura do acelerador e que apresentem resultados de desempenho, área e potência para as versões do Kyber.

3 ACELERADOR KYBER EM FPGA

Esse capítulo descreve a metodologia utilizada para esta pesquisa, a implementação proposta para o acelerador Kyber em relação a arquitetura escolhida, as especificações HLS que foram desenvolvidas no processo e as especificações Python utilizadas para configuração e teste da FPGA.

3.1 METODOLOGIA

Esse projeto foi desenvolvido utilizando uma placa PYNQ-Z1 da Digilent, que tem como base um chip Xilinx ZYNQ-7000 que combina lógica programável de uma FPGA (modelo XC7Z020-1CLG400C) com um processador ARM Cortex-A9 dual-core de 650 MHz. Ela é uma placa voltada para prototipação de projetos FPGA de baixo custo e acompanha uma imagem linux com módulos Python que permitem a rápida programação da FPGA em tempo de execução.

Além da placa PYNQ-Z1, foi utilizado o aparelho USB meter, de modelo UM24C, do fabricante Hangzhou Ruideng Technology. Dentre outras medidas, esse aparelho apresenta a tensão de alimentação e a corrente consumida por dispositivos USB. Com essas medidas foi possível obter informações de potência e energia do acelerador durante sua execução. Esses resultados são apresentados na Seção 4.3.

A Figura 3.1 mostra a imagem da PYNQ-Z1 que foi utilizada no desenvolvimento do acelerador acoplada ao aparelho USB meter durante a execução de suas operações.

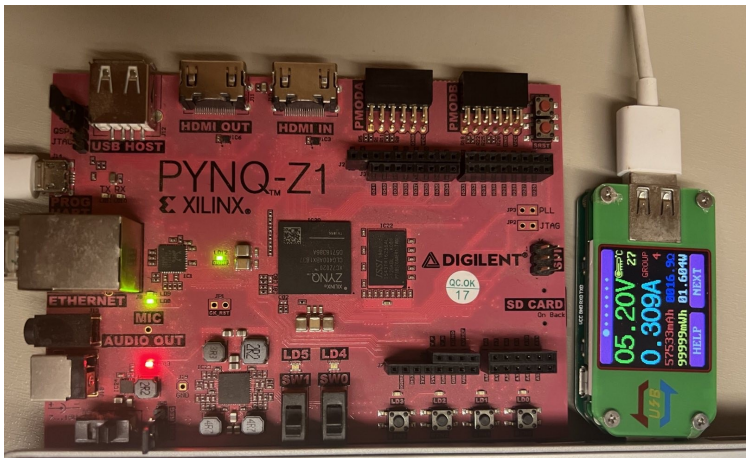

Figura 3.1: Dispositivos utilizados para o desenvolvimento dos resultados. (Fonte: o autor)

Uma vez especificada a funcionalidade a ser desempenhada pelo acelerador Kyber, foi desenvolvido o código em linguagem C que atendesse às especificações, utilizando como base o código oficial de referência do Kyber, disponibilizado no repositório público git [75]. Esse código do acelerador é descrito na Seção 3.3.

A descrição RTL proposta é resultante da síntese executada pela ferramenta Vitis HLS (v2022.1) [76], da empresa Xilinx, a partir do código em C. A linguagem de descrição de hardware selecionada para o

desenvolvimento do RTL foi a Verilog, definida por padrão pela síntese HLS.

O código RTL desenvolvido foi exportado pelo Vitis, viabilizando a importação da descrição gerada para a ferramenta Vivado (v2022.1)[77], também desenvolvida pela empresa Xilinx. Com essa ferramenta foi feita a implementação da arquitetura considerando o modelo da FPGA utilizada nesta dissertação. Após o processo de implementação, foi possível realizar a geração do arquivo *bitstream*, necessário para a configuração da FPGA.

O sistema operacional Linux presente na placa possui um servidor SSH que viabiliza a transferência do arquivo *bitstream* para a FPGA. Com a biblioteca PYNQ, que utiliza o Python como linguagem, foi possível configurar a FPGA, em tempo de execução, para realizar as operações inicialmente definidas para o acelerador.

A Figura 3.2 mostra de forma simplificada e estruturada as fases executadas no processo de desenvolvimento do acelerador Kyber, especificando as ferramentas utilizadas em cada etapa desse processo.

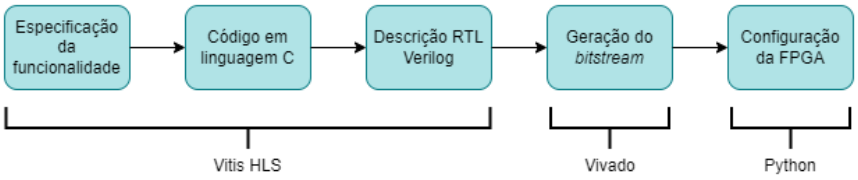

Figura 3.2: Fases do desenvolvimento da arquitetura do acelerador Kyber na FPGA. (Fonte: o autor)

Para validar o acelerador construído e obter os resultados oriundos da sua execução, foram realizados dois testes. Um deles foi desenvolvido e processado pela ferramenta de software Vitis HLS e consiste na realização de uma simulação de compartilhamento de chaves, com uso do acelerador e do processador ARM executando as operações dos algoritmos KEM. O outro, realizado na FPGA, executa apenas as operações de cifração (PKE.Enc()) e decifração (PKE.Dec()) de mensagem, inicialmente definidas como funcionalidades do acelerador.

3.2 ARQUITETURA ACELERADOR KYBER

A arquitetura estabelecida para o acelerador Kyber é um projeto de codesign, com a utilização do processador ARM e a FPGA. Assim sendo, devido a restrições de recursos da FPGA, foi definido que parte do código continuaria sendo executado pelo ARM, enquanto que a FPGA seria responsável por executar em paralelo o maior número possível de operações que são custosas para serem executadas pelo processador.

De forma geral, as funções hash e as multiplicações de polinômios são operações que se beneficiam do paralelismo em sua execução. Os algoritmos que compõem a cifração de chave pública, presentes no KEM Kyber, foram escolhidos para compor a arquitetura do acelerador na FPGA uma vez que eles possuem em suas construções essas operações que são paralelizáveis.

Na interface da arquitetura é utilizado o protocolo AXI (*Advanced eXtensible Interface*) para viabilizar a comunicação e, por consequência, a troca de informações entre o processador e a FPGA. Esse protocolo,

definido em [78], foi introduzido pela ARM de acordo com o padrão AMBA (*Advanced Microcontroller Bus Architecture*) e é utilizado em projetos de sistemas de alta frequência e alto desempenho. O compilador HLS é o responsável por produzir automaticamente essa interface.

A Figura 3.3 mostra de forma simplificada o diagrama de blocos com as interfaces do acelerador Kyber. As linhas verde e amarela correpondem as interfaces *axi-master* e *axi-lite*, respectivamente. Optou-se por utilizar essa configuração pois o barramento *axi-master* oferece uma alta performance com a possibilidade da transferência de dados por meio de rajadas, enquanto que o *axi-lite* transfere apenas um dado por vez.

O módulo *ps7_axi periph* permite a conexão e gerenciamento dos periféricos pelo sistema de processamento, sendo utilizado para lidar com sinais de controle e dados de registro mapeados em memória. O módulo *axi_mem intercon* permite o acesso à memória externa e é usado para rápida transferência de dados entre as BRAMs do acelerador e a memória externa *Double Data Rate* (DDR).

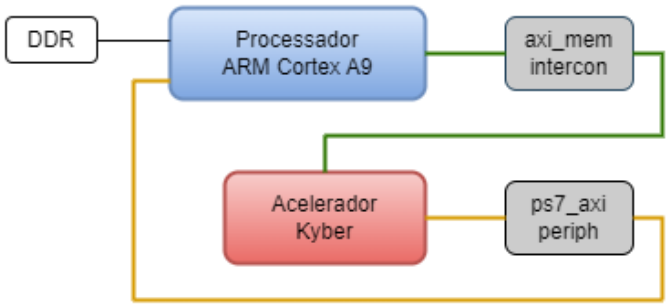

Figura 3.3: Arquitetura e interface do acelerador Kyber. (Fonte: o autor)

Para acessar os parâmetros do acelerador, na FPGA, é necessário mapear os endereços de memória de cada um deles. A Tabela 3.1 mostra os endereços que foram definidos para os parâmetros m, ct, sk, pk, $coins$ e op, que são utilizados na construção das operações do acelerador. Esses valores são fundamentais para a execução do teste do Kyber na FPGA, detalhado na Seção 3.4.

Tabela 3.1: Endereço dos registradores dos parâmetros do acelerador.

Endereço do registrador	Descrição	Bit
0x00	Sinais de controle	0 : ap_start 1 : ap_done 2 : ap_idle 3 : ap_ready
0x10	Sinal de dados de m	0 - 31
0x18	Sinal de dados de ct	0 - 31
0x20	Sinal de dados de sk	0 - 31
0x28	Sinal de dados de pk	0 - 31
0x30	Sinal de dados de $coins$	0 - 31
0x38	Sinal de dados de op	0 - 31

3.3 ESPECIFICAÇÃO KYBER HLS

O acelerador proposto teve sua implementação baseada no código de referência do Kyber, que é disponibilizado no repositório público git [75], na linguagem C. Algumas modificações foram necessárias para viabilizar a geração da arquitetura RTL pela ferramenta de síntese Vitis HLS.

Devido a limitações na disponibilidade dos recursos da FPGA utilizada, foram escolhidas as funções dos algoritmos PKE.Enc() e PKE.Dec() para serem implementadas no acelerador. Como o algoritmo responsável pela geração das chaves, o PKE.KeyGen(), possui em sua implementação na linguagem C uma função com chamada de sistema para a geração de bytes aleatórios, ele não foi considerado para ser implementado na FPGA. Para facilitar o detalhamento da implementação adotada, o código é explicado a seguir, em blocos.

Definida como *acelerador_kyber* e detalhada no Código 3.1, a função que descreve o acelerador recebe como um dos parâmetros um inteiro op que define a operação que será realizada por ele. Se op for igual a 1 é realizada a operação PKE.Enc() e se for igual a 2 realiza-se a PKE.Dec(). Os demais parâmetros, a mensagem m, o cifrado ct, a chave privada sk e a chave pública pk, podem ser de entrada ou de saída, dependendo da operação que o acelerador executa.

Nas linhas de 5 a 16 da função *acelerador_kyber* são especificadas as definições do protocolo de comunicação estabelecido entre o acelerador e os componentes externos. A interface *m_axi*, que segue o padrão AXI, viabiliza a transferência de dados em rajadas, permitindo a rápida entrada e saída de dados da memória BRAM da FPGA, enquanto que a interface *s_axilite* mapeia um mecanismo de controle.

As declarações das variáveis necessárias para receber os valores dos dados que são passados para o acelerador são encontradas nas linhas de 19 a 23. Sendo que as variáveis m_hls, ct_hls, sk_hls, pk_hls e $coins\_hls$ são definidas para receberem 32, 1088, 1152, 1184 e 32 bytes, respectivamente.

Código 3.1: Acelerador Kyber HLS

```
1  void acelerador_kyber(volatile uint8_t *m, volatile uint8_t *ct,
2                        volatile uint8_t *sk, volatile uint8_t *pk,
3                        volatile uint8_t *coins, int op)
4  {
5      #pragma HLS INTERFACE mode=m_axi depth=32 port=m offset=slave
6      #pragma HLS INTERFACE mode=m_axi depth=1088 port=ct offset=slave
7      #pragma HLS INTERFACE mode=m_axi depth=1152 port=sk offset=slave
8      #pragma HLS INTERFACE mode=m_axi depth=1184 port=pk offset=slave
9      #pragma HLS INTERFACE mode=m_axi depth=32 port=coins offset=slave
10     #pragma HLS INTERFACE mode=s_axilite port=m bundle=CONTROL
11     #pragma HLS INTERFACE mode=s_axilite port=ct bundle=CONTROL
12     #pragma HLS INTERFACE mode=s_axilite port=sk bundle=CONTROL
13     #pragma HLS INTERFACE mode=s_axilite port=pk bundle=CONTROL
14     #pragma HLS INTERFACE mode=s_axilite port=coins bundle=CONTROL
15     #pragma HLS INTERFACE mode=s_axilite port=op bundle=CONTROL
16     #pragma HLS INTERFACE mode=s_axilite port=return bundle=CONTROL
17
18     unsigned int i;
```

```
19      uint8_t m_hls[KYBER_INDCPA_MSGBYTES_hls];
20      uint8_t ct_hls[KYBER_INDCPA_BYTES_hls];
21      uint8_t sk_hls[KYBER_INDCPA_SECRETKEYBYTES_hls];
22      uint8_t pk_hls[KYBER_INDCPA_PUBLICKEYBYTES_hls];
23      uint8_t coins_hls[KYBER_SYMBYTES_hls];
```

Entre as linhas 24 e 70 estão as funções executadas pela FPGA quando a operação de cifração é selecionada. Nesse caso, a mensagem m, a chave pública pk e o aleatório *coins* são os parâmetros de entrada, sendo o cifrado ct o parâmetro de saída, retornado pelo acelerador. Nas linhas 27 a 34, 68 e 69 tem-se as definições necessárias para as transferências de dados entre a FPGA e o processador.

A definição do valor da constante *KYBER_K_hls*, que aparece nas linhas 38, 45, 48 e 54, é vinculada ao nível de segurança do mecanismo de encapsulamento de chaves Kyber e, consequentemente, à sua versão. Como optou-se por utilizar a versão Kyber768, o valor da constante é definido como 3.

```
24      if(op == 1) //cifracao
25      {
26          //copiar dados para o acelerador
27          for (i = 0; i < KYBER_INDCPA_MSGBYTES_hls; i++)
28              m_hls[i] = m[i];
29
30          for (i = 0; i < KYBER_INDCPA_PUBLICKEYBYTES_hls; i++)
31              pk_hls[i] = pk[i];
32
33          for (i = 0; i < KYBER_SYMBYTES_hls; i++)
34              coins_hls[i] = coins[i];
35
36          uint8_t seed_hls[KYBER_SYMBYTES_hls];
37          uint8_t nonce_hls = 0;
38          polyvec sp_hls, pkpv_hls, ep_hls, at_hls[KYBER_K_hls], x_hls;
39          poly v_hls, k_hls, epp_hls;
40
41          unpack_pk_hls(&pkpv_hls, seed_hls, pk_hls);
42          poly_frommsg_hls(&k_hls, m_hls);
43          gen_at_hls(at_hls, seed_hls);
44
45          for(i=0;i<KYBER_K_hls;i++)
46             poly_getnoise_eta1_hls(sp_hls.vec+i, coins_hls, nonce_hls++);
47
48          for(i=0;i<KYBER_K_hls;i++)
49             poly_getnoise_eta2_hls(ep_hls.vec+i, coins_hls, nonce_hls++);
50
51          poly_getnoise_eta2_hls(&epp_hls, coins_hls, nonce_hls++);
52          polyvec_ntt_hls(&sp_hls);
53
54          for(i=0;i<KYBER_K_hls;i++)
55             polyvec_basemul_acc_montgomery_hls(&x_hls.vec[i], &at_hls[i], &sp_hls);
56
57          polyvec_basemul_acc_montgomery_hls(&v_hls, &pkpv_hls, &sp_hls);
58          polyvec_invntt_tomont_hls(&x_hls);
```

```
59          poly_invntt_tomont_hls(&v_hls);
60          polyvec_add_hls(&x_hls, &x_hls, &ep_hls);
61          poly_add_hls(&v_hls, &v_hls, &epp_hls);
62          poly_add_hls(&v_hls, &v_hls, &k_hls);
63          polyvec_reduce_hls(&x_hls);
64          poly_reduce_hls(&v_hls);
65          pack_ciphertext_hls(ct_hls, &x_hls, &v_hls);
66
67          //devolver os dados computados para o processador principal
68          for (i = 0 ; i < KYBER_INDCPA_BYTES_hls ; i++)
69              ct[i] = ct_hls[i];
70      }
```

Quando o acelerador é determinado para realizar uma decifração, são executadas as operações descritas entre as linhas 71 e 95. O acelerador recebe como parâmetros de entrada o cifrado ct e a chave privada sk, retornando como resultado a mensagem decifrada m. Na decifração, as transferências de dados entre o processador e a FPGA são descritas nas linhas de 74 a 78, 93 e 94.

```
71      else if(op == 2) //decifracao
72      {
73          //copiar dados para o acelerador
74          for (i = 0 ; i < KYBER_INDCPA_BYTES_hls ; i++)
75              ct_hls[i] = ct[i];
76
77          for (i = 0 ; i < KYBER_INDCPA_SECRETKEYBYTES_hls ; i++)
78              sk_hls[i] = sk[i];
79
80          polyvec x_hls, skpv_hls;
81          poly y_hls, mp_hls;
82
83          unpack_ciphertext_hls(&x_hls, &y_hls, ct_hls);
84          unpack_sk_hls(&skpv_hls, sk_hls);
85          polyvec_ntt_hls(&x_hls);
86          polyvec_basemul_acc_montgomery_hls(&mp_hls, &skpv_hls, &x_hls);
87          poly_invntt_tomont_hls(&mp_hls);
88          poly_sub_hls(&mp_hls, &y_hls, &mp_hls);
89          poly_reduce_hls(&mp_hls);
90          poly_tomsg_hls(m_hls, &mp_hls);
91
92          //devolver os dados computados para o processador principal
93          for (i = 0 ; i < KYBER_INDCPA_MSGBYTES_hls ; i++)
94              m[i] = m_hls[i];
95      }
96  }
```

Todas as funções mencionadas acima também foram incluídas na arquitetura da FPGA e suas implementações são detalhadas na Seção I.1 do Apêndice. De forma geral, tanto na cifração quanto na decifração, essas operações consistem em Transformadas Numéricas de Fourier (*Number Theoretic Transform* - NTT), reduções, multiplicações, somas e subtrações realizadas com polinômios e com vetores de

polinômios.

Os algoritmos de cifração e decifração do PKE, aqui implementados são descritos em [44]. O detalhamento sobre cada uma das funções utilizadas pela implementação dos algoritmos acima referidos podem ser encontradas no código de referência disponibilizado em [75].

3.4 ESPECIFICAÇÃO PYNQ PARA TESTE KYBER FPGA

Para validar o acelerador construído utilizando as ferramentas de software da Xilinx Vitis HLS e Vivado, foi desenvolvido um código em Python, utilizando a biblioteca PYNQ, para realizar a configuração da FPGA com o carregamento do *bitstream* construído pelas ferramentas e a execução das suas operações de cifração e decifração. O Código 3.2 mostra o detalhamento desse código e foi separado em blocos para facilitar o entendimento de seus comandos.

Nas linhas de 1 a 5 são feitas as importações dos módulos que são utilizados ao longo do código. A função definida como *load_overlay*, entre as linhas 7 a 12, é utilizada para configurar a FPGA, carregando o design a partir do arquivo *bitstream* gerado pela ferramenta Vivado.

Código 3.2: Configuração da FPGA e execução das operações de cifração e decifração

```
1  from pynq import Xlnk
2  from pynq import Overlay
3  import time
4  import sys
5  import numpy as np
6
7  def load_overlay(path):
8      overlay = Overlay(path)
9      print(overlay.ip_dict)
10     kyber_acc = overlay.kyber_accelerator_0
11     print('overlay loaded')
12     return kyber_acc
```

Iniciada na Linha 13, a função principal começa com a chamada da função *load_overlay*, na Linha 16, para realizar a configuração da FPGA de forma a habilitá-la a realizar as operações das funcionalidades de cifração e decifração definidas na construção do acelerador.

Em seguida, tem-se a definição do *cifrado_esperado*, uma lista que contém o valor esperado para o texto cifrado resultante da operação de cifração sendo realizada com os parâmetros de entrada *coins_tmp*, *m_tmp* e *pk_tmp*, definidos nas linhas 19, 20 e 21, respectivamente. Na Linha 22 tem-se a definição dos valores da chave secreta *sk_tmp*, utilizada como pârametro de entrada para a operação de decifração. Os valores completos definidos para os parâmetros aqui mencionados são expostos no Apêndice I.2.

```
13 def main():
14
```

```
15      kyber_acc = load_overlay('/home/xilinx/kyber_final/kyber_final.bit')
16
17      cifrado_esperado = [171,94,...,48,18]
18
19      coins_tmp = [229, 84, ..., 53, 98]
20      m_tmp = [24, 106, ..., 100, 27]
21      pk_tmp = [126, 185, ..., 211, 64]
22      sk_tmp = [99, 172, ..., 88, 35]
```

Para viabilizar o armazenamento dos valores definidos para cada um dos parâmetros, nas linhas de 24 a 29 são instanciados *arrays* do tipo uint8 com diferentes tamanhos em bytes, sendo 32 para m , 1184 para pk, 1152 para sk, 1088 para ct e 32 para $coins$. Entre as linhas 30 e 40 os *arrays* são preenchidos com os valores dos parâmetros.

```
23      xlnk = Xlnk()
24      m = xlnk.cma_array(shape=(32,), dtype=np.uint8)
25      pk = xlnk.cma_array(shape=(((3 * 384) + 32),), dtype=np.uint8)
26      sk = xlnk.cma_array(shape=((3 * 384),), dtype=np.uint8)
27      ct = xlnk.cma_array(shape=(((3 * 320) + 128),), dtype=np.uint8)
28      coins = xlnk.cma_array(shape=(32,), dtype=np.uint8)
29
30      for i in range(0, len(m_tmp)):
31          m[i] = m_tmp[i]
32
33      for i in range(0, len(pk_tmp)):
34          pk[i] = pk_tmp[i]
35
36      for i in range(0, len(sk_tmp)):
37          sk[i] = sk_tmp[i]
38
39      for i in range(0, len(coins_tmp)):
40          coins[i] = coins_tmp[i]
```

As linhas entre 41 e 48 são utilizadas para escrever nos registradores que foram previamente mapeados e são descritos na Seção 3.2. Na Linha 41 é inicializado o acelerador, enquanto que na 42 é definida a operação que ele deve realizar, sendo nesse caso a cifração correspondente ao valor 1. Entre as linhas 44 e 48 são passados para os registradores de cada parâmetro os ponteiros com os endereços dos seus respectivos *arrays*.

Após esses procedimentos, o acelerador começa a funcionar a partir do comando da Linha 53 e utilizou-se $t1$ e $t2$ para calcular o tempo gasto na execução da cifração até que aconteça a parada, com o fim das operações. Nas linhas de 59 a 62 foi comparado o resultado obtido pela cifração executada pelo acelerador Kyber e armazenada em ct com o valor esperado para validar o funcionamento do acelerador.

```
41      kyber_acc.write(0x0,0)
42      kyber_acc.write(0x38,1) #1 enc, 2 dec
43
44      kyber_acc.write(0x10,m.physical_address)
```

```
45      kyber_acc.write(0x18,ct.physical_address)
46      kyber_acc.write(0x20,sk.physical_address)
47      kyber_acc.write(0x28,pk.physical_address)
48      kyber_acc.write(0x30,coins.physical_address)
49
50      val = 0
51      print('start encrypt')
52
53      kyber_acc.write(0x0,1) #start
54      t1 = time.time()
55      while(val != 4):
56          val = kyber_acc.read(0x0)
57      t2 = time.time()
58
59      for i in range(0,len(ct)):
60          if ct[i] != cifrado_esperado[i]:
61              print("erro no resultado")
62              break
63
64      print()
65      print('time encrypt=%f seconds' % (t2-t1))
```

Como na decifração o parâmetro m torna-se de saída, optou-se por preencher seu *array* com zeros, nas linhas 66 e 67, para garantir que o valor que estivesse nesse parâmetro no momento da validação seria o calculado pela operação de decifração e não o valor anteriormente definido e copiado para esse *array* antes da operação de cifração.

Em seguida, os procedimentos definidos são similares ao utilizados no caso da cifração, tendo como diferença o estabelecimento da operação de decifração na Linha 70, ao escrever 2 no registrador. Para validar o resultado, comparou-se a mensagem obtida com a decifração utilizando o acelerador e armazenada em m com a mensagem original definida em m_tmp.

```
66      for i in range(0,len(m)):
67          m[i] = 0
68
69      kyber_acc.write(0x0,0)
70      kyber_acc.write(0x38,2) #1 enc, 2 dec
71
72      val = 0
73      print('start decrypt')
74
75      kyber_acc.write(0x0,1) #start
76      t1 = time.time()
77      while(val != 4):
78          val = kyber_acc.read(0x0)
79      t2 = time.time()
80
81      for i in range(0,len(m)):
82          if m[i] != m_tmp[i]:
83              print("erro no resultado")
```

```
84                 break
85
86         print('time decrypt=%f seconds' % (t2-t1))
87
88 if __name__ == '__main__':
89         main()
```

4 RESULTADOS E ANÁLISES

Aqui são apresentados os resultados obtidos ao utilizar o acelerador Kyber, descrito no Capítulo 3, executando a versão Kyber768, que possui uma segurança pós-quântica estimada de mais de 128 bits. Nas seções que seguem, os resultados estão separados em relação ao desempenho, à área do circuito e à potência. Foi considerada a FPGA XC7Z020-1CLG400C (PYNQ-Z1) da Xilinx como dispositivo.

4.1 ANÁLISE DE DESEMPENHO DA SIMULAÇÃO E DA IMPLEMENTAÇÃO

4.1.1 Análise de desempenho da simulação na ferramenta HLS

Utilizando os algoritmos do KEM Kyber em um compartilhamento de chaves como o da Figura 2.4, as funções de cifração (PKE.Enc()) e decifração (PKE.Dec()) são empregadas três vezes, sendo uma PKE.Enc() no encapsulamento e as outras duas vezes, uma PKE.Dec() e outra PKE.Enc(), no desencapsulamento, conforme demonstrado nos algoritmos 2 e 3, da Seção 2.2.

Realizando uma simulação de compartilhamento de chaves na ferramenta Vitis HLS, utilizando o acelerador Kyber desenvolvido, obtêm-se os resultados estimados de desempenho, medidos em ciclos de relógio. Como valores médios, a latência teve como resultado 127.051, o intervalo foi de 112.605 e o tempo total de execução do acelerador foi de 381.154 ciclos, considerando as três vezes em que ele é acionado para a realização das operações de cifração e decifração durante o encapsulamento e o desencapsulamento da chave.

Como o acelerador foi sintetizado a 100MHz, depreende-se que o tempo gasto pelo acelerador durante a simulação proposta foi de aproximadamente 3,81 milissegundos. Esse valor corresponde a soma de tempo da execução dos algoritmos de duas cifrações e uma decifração.

4.1.2 Análise de desempenho da execução da implementação na FPGA

Ao realizar o segundo teste, com a execução das operações na placa FPGA, os valores medidos foram de aproximadamente 5,01 milissegundos para a cifração (PKE.enc()) e 2,24 milissegundos para a decifração (PKE.dec()).

Pode-se estimar que em uma simulação de compartilhamento de chaves, onde no encapsulamento a operação de cifração é realizada uma vez e no desencapsulamento são realizadas uma vez a operação de cifração e uma vez a de decifração, o tempo total gasto pelo acelerador na FPGA seria de aproximadamente 12,26 milissegundos, um tempo superior ao estimado pela ferramenta Vitis HLS.

Esse comportamento é esperado dado que no teste da FPGA, além do tempo gasto para o processamento das operações pelo acelerador, é também considerado o tempo para transferência de dados entre o processador ARM e a FPGA, tanto para a entrada dos dados na FPGA quanto para a saída do resultado.

Além disso, deve-se considerar que essa passagem dos dados é limitada pela frequência da FPGA, que é de 100 MHz, mesmo que a frequência do processador seja maior, de 650 MHz.

4.2 ÁREA DO CIRCUITO

Desenvolvida pela ferramenta Vivado, após a fase de implementação, a Figura 4.1 mostra a configuração da FPGA com o processo de posicionamento e roteamento da arquitetura do acelerador Kyber no dispositivo. Pela imagem é possível observar que a arquitetura proposta ocupou grande parte dos recursos disponíveis no modelo de FPGA utilizada para essa dissertação, a PYNQ-Z1.

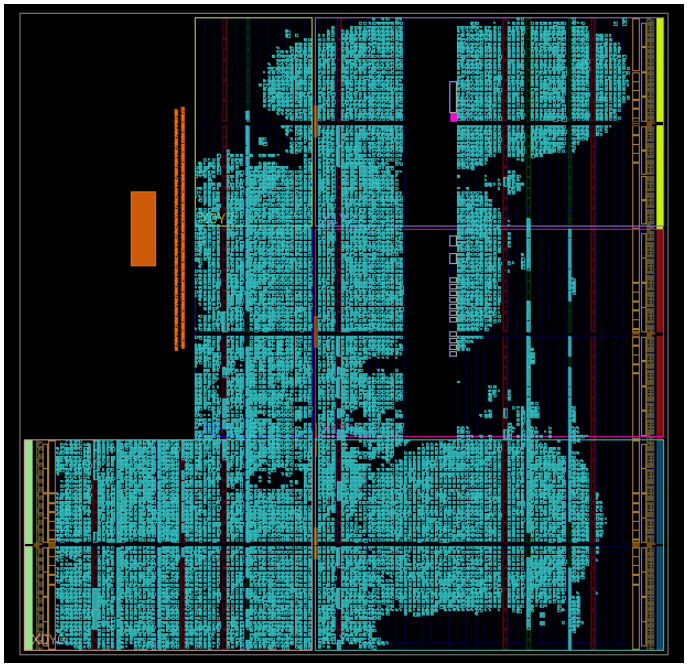

Figura 4.1: Ocupação da FPGA após processo de implementação. (Fonte: o autor)

O resumo dos recursos utilizados pelo acelerador após o processo de implementação realizado pelo Vivado é mostrado na Tabela 4.1. Ela traz informações a respeito da quantidade de recursos disponíveis na FPGA, os valores referentes ao total utilizado de cada um dos recursos, bem como o percentual que esses valores representam.

Tabela 4.1: Utilização de recursos da FPGA.

Recurso	Disponível	Utilizado	Percentual
LUT	53200	33733	63
LUTRAM	17400	423	2
FF	106400	22810	21
BRAM	140	23,5	17
DSP	220	151	69

Os recursos com maiores utilizações foram os DSPs e as LUTs, com aproximadamente 69% e 63%, respectivamente. Isso mostra que o compilador HLS foi capaz de aproveitar os DSPs para viabilizar o

paralelismo de operações, acelerando a execução das mesmas. O recurso menos utilizado foi a LUTRAM, com apenas 2%, visto que o projeto desenvolvido para o acelerador lida com poucas porções pequenas de dados.

As funções que mais se beneficiaram das DSPs foram a *polyvec_basemul_acc_montgomery_hls*, que realiza a multiplicação de vetores de polinômios, e a *poly_basemul_montgomery_hls*, que calcula a multiplicação de polinômios. As funções de hash SHAKE-128, utilizada como função de saída extensível dentro na função *gen_at_hls*, e SHAKE-256, usada como função pseudo-aleatória e que é chamada pelas funções *poly_getnoise_eta1_hls* e *poly_getnoise_eta2_hls*, são as funções que mais utilizam os recursos de LUT da FPGA.

4.3 ANÁLISE DE POTÊNCIA DA SIMULAÇÃO E DA IMPLEMENTAÇÃO

4.3.1 Análise de potência da simulação na ferramenta HLS

O consumo de potência em FPGAs pode ser dividido em dinâmico e estático. O dinâmico é a potência consumida pela transição de sinais no dispositivo enquanto o estático é a energia que continua a ser consumida mesmo na ausência de transições de sinal.

A ferramenta Vivado disponibiliza uma estimação do consumo da potência dos componentes que compõem a arquitetura desenvolvida. A Tabela 4.2 apresenta essas informações de forma detalhada, considerando o teste de simulação de compartilhamento de chaves similar ao da Figura 2.4, utilizando os algoritmos do KEM Kyber, na versão Kyber768.

A análise apresentada mostra, em Watts, que o consumo total de potência estimado é de aproximadamente 2,243 W, sendo que desse total 0,156 W é consumo estático, ou seja, na ausência de transição de sinais. O maior consumo observado, de 1,527 W, é feito pelo processador ARM, para a realização das operações que não estão inclusas no acelerador.

O consumo relacionado às funções de cifração e decifração implementadas na parte programável da FPGA podem ser interpretadas como a soma das potências consumidas pelos componentes dinâmicos *clocks*, sinais, lógica, BRAM e DSP. Dessa forma, é estimado que o acelerador consuma aproximadamente 0,56 W de potência.

Tabela 4.2: Distribuição da potência consumida pela arquitetura implementada.

	Componente	Potência (W)
Dinâmico	*Clocks*	0,032
	Sinais	0,251
	Lógica	0,186
	BRAM	0,022
	DSP	0,069
	Processador	1,527
Estático	-	0,156
Total	-	2,243

4.3.2 Análise de potência e consumo de energia da execução da implementação na FPGA

Através do segundo teste, executado na FPGA, foi possível obter as medidas da tensão de alimentação e da corrente consumida pela placa durante a execução das operações estabelecidas, com o auxílio do aparelho USB meter mostrado na Figura 3.1. As medições de corrente em função do tempo são mostradas no gráfico da Figura 4.2. Uma vez que a tensão foi de 5,2 Volts e a média dos valores da corrente foi de 0,31 Amperes, fora da execução das operações, é possível concluir que a potência estática foi de aproximadamente 1,61 W.

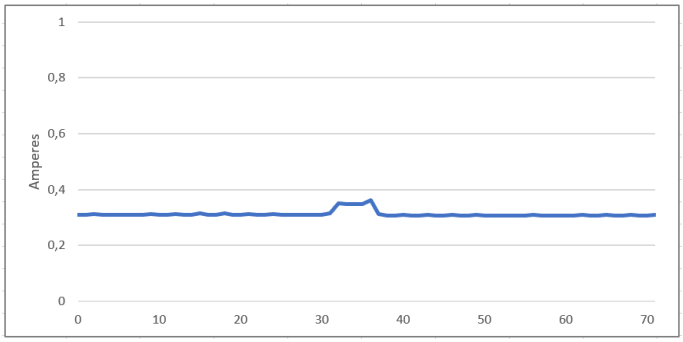

Figura 4.2: Medida do consumo de corrente pela FPGA. (Fonte: o autor)

O aumento da corrente, observado entre o instante 30 e 40, se dá quando há o carregamento do arquivo *bitstream*, a configuração da FPGA, a transferência de dados e a consequente execução das operações de cifração e decifração realizadas pelo acelerador. Com esses dados é possível calcular a energia gasta pelo acelerador, que foi de aproximadamente 6,2 Joules.

5 CONCLUSÕES E TRABALHOS FUTUROS

Devido a ameaça que o avanço da computação quântica representa, é importante entender o papel da criptografia na segurança da informação e o quão necessária se torna a evolução constante dessa área para garantir a continuidade da segurança de dados.

Uma vez que a criptografia de chave pública atual é vulnerável em relação a essa nova tecnologia, se faz necessário o estabelecimento de novos padrões criptográficos pós-quânticos. É imprescindível ainda que esses novos padrões possuam implementações que otimizem o desempenho dos seus algoritmos e viabilizem seu funcionamento em um maior número de aplicações e em diversos dispositivos.

Nessa dissertação foram apresentadas as principais características dos algoritmos que compõem o CRYSTALS-Kyber, o primeiro mecanismo de encapsulamento de chaves pós-quântico padronizado pelo NIST que é baseado no problema MLWE em reticulados e possui segurança IND-CCA2.

Foi proposto um acelerador para o Kyber desenvolvido por meio de ferramentas de síntese de alto nível (HLS) que traduzem especificações em linguagem C para descrição em nível de transferência entre registradores (RTL), com a linguagem de descrição de hardware Verilog. O acelerador foi sintetizado, implementado e configurado em um FPGA da Xilinx, modelo XC7Z020-1CLG400C, conhecida como (PYNQ-Z1).

O modelo da FPGA utilizada é de baixo custo e possui recursos limitados. Devido a isso, a arquitetura proposta utilizou boa parte dos recursos disponíveis somente com a implementação das funções de cifração e decifração que estão presentes nos algoritmos de encapsulamento e desencapsulamento do mecanismo Kyber. Esse fato inviabilizou a inserção de outras funções significativas na arquitetura, para serem executadas pelo acelerador. Ainda assim o acelerador foi eficaz na realização das operações propostas.

Os resultados obtidos com a implementação da versão Kyber768 mostraram que os recursos mais utilizados foram os DSPs e as LUTs, com 69% e 63%, respectivamente. Os DSPs foram, em sua maioria, utilizados para viabilizar o paralelismo das operações de multiplicações polinomiais. As funções de hash presentes nos algoritmos do KEM Kyber foram as responsáveis pela maior parte da utilização dos recursos disponíveis na FPGA.

Em uma simulação de compartilhamento de chave, realizada pela ferramenta Vitis HLS, onde o acelerador é acionado três vezes, o tempo gasto por ele para execução de duas operações de cifração e uma de decifração foi de aproximadamente 3,81 milissegundos, com um consumo de potência estimado de apenas 2,243 W.

Considerando o teste efetuado na FPGA, com a execução das operações de cifração e decifração apenas, observou-se que o tempo gasto no processamento dessas operações foi bem superior ao estimado pela ferramenta Vitis na simulação do compartilhamento de chaves, sendo de aproximadamente 5,01 milissegundos para a cifração e 2,24 milissegundos para a decifração.

Esse fato já era esperado pois a ferramenta Vitis HLS considera apenas o tempo de execução das operações no acelerador, enquanto que o tempo medido no teste utilizando a placa FPGA também leva em

conta o tempo gasto para a transferência de dados entre a FPGA e o processador ARM.

O consumo de energia utilizado pelo acelerador durante a execução das operações de cifração e decifração foi de aproximadamente 6,2 Joules, confirmando que a FPGA é uma boa opção para a construção de aceleradores de alto desempenho, mantendo um baixo consumo de energia.

5.1 TRABALHOS FUTUROS

Como trabalho futuro, tem-se a possibilidade de implementação das demais versões do mecanismo de encapsulamento Kyber, o Kyber512 e o Kyber 1024, para comparar com os resultados obtidos com a versão implementada (Kyber768) e assim poder fazer inferências a respeito da versão que melhor se adequa de acordo com os recursos disponíveis no dispositivo alvo.

Como a arquitetura sugerida foi construída a partir do código C padrão disponibilizado pelo Kyber, sem otimizações, é possível que outras implementações em linguagem C possam ser encontradas, principalmente das funções hash que são utilizadas pelos algoritmos. Essas outras implementações podem ser testadas com a finalidade de encontrar uma que resulte em um código RTL otimizado e que ocupe uma menor área do dispositivo, a fim de viabilizar a inclusão de outras operações significativas na FPGA.

Outra abordagem a ser desenvolvida é estudar meios de incluir a geração das chaves na FPGA, contornando as chamadas de sistema utilizadas para a geração dos bits aleatórios, mas certificando-se que essa geração continue com as propriedades de aleatoriedade necessárias para garantir a segurança do mecanismo.

Por fim, deve-se realizar uma simulação de compartilhamento de chaves executando os algoritmos do Kyber exclusivamente no processador ARM Cortex-A9 e uma outra simulação utilizando a arquitetura de codesign, com o uso do processador e da parte programável desenvolvida para a FPGA. A comparação entre essas duas simulações viabilizará a análise de eficiência do acelerador.

Referências Bibliográficas

1 CETIC.BR | NIC.BR. *Resumo executivo - Pesquisa TIC domicílio 2021*. Disponível em: <https://cetic.br/media/docs/publicacoes/2/20221121125804/resumo_executivo_tic_domicilios_2021.pdf>. Acesso em: 04 fev. 2023.

2 ALSHEHRI, J.; ALHAMED, A. A review paper for the role of cryptography in network security. In: IEEE. *2022 4th International Conference on Electrical, Control and Instrumentation Engineering (ICECIE)*. [S.l.], 2022. p. 1–5.

3 CORON, J.-S. What is cryptography? *IEEE security & privacy*, IEEE, v. 4, n. 1, p. 70–73, 2006.

4 KESSLER, G. C. An overview of cryptography (updated version, 3 march 2016). 2016.

5 GENÇOĞLU, M. T. Importance of cryptography in information security. *IOSR J. Comput. Eng*, v. 21, n. 1, p. 65–68, 2019.

6 ZULKIFLI, M. Z. W. M. Evolution of cryptography. Citeseer, v. 8, n. 06, 2007.

7 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *NISTIR 7977. NIST Cryptographic Standards and Guidelines Development Process*. 2016. Disponível em: <http://dx.doi.org/10.6028/NIST.IR.7977>. Acesso em: 04 fev. 2023.

8 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *FIPS PUB. 197. Advanced Encryption Standard (AES)*. 2001. Disponível em: <https://doi.org/10.6028/NIST.FIPS.197>. Acesso em: 04 fev. 2023.

9 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *FIPS PUB. 186-5. Digital Signature Standard (DSS)*. 2019. Disponível em: <https://doi.org/10.6028/NIST.FIPS.186-5>. Acesso em: 04 fev. 2023.

10 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *FIPS PUB. 202. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015. Disponível em: <http://dx.doi.org/10.6028/NIST.FIPS.202>. Acesso em: 04 fev. 2023.

11 BERNSTEIN, D. J.; LANGE, T. Post-quantum cryptography. *Nature*, Nature Publishing Group UK London, v. 549, n. 7671, p. 188–194, 2017.

12 GROVER, L. K. A fast quantum mechanical algorithm for database search. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1996. p. 212–219.

13 SHOR, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In: IEEE. *Proceedings 35th annual symposium on foundations of computer science*. [S.l.], 1994. p. 124–134.

14 NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. 2016. Disponível em: <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>. Acesso em: 04 fev. 2023.

15 ALAGIC, G.; APON, D.; COOPER, D.; DANG, Q.; DANG, T.; KELSEY, J.; LICHTINGER, J.; MILLER, C.; MOODY, D.; PERALTA, R. et al. Status report on the third round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022.

16 POLICARPO, R. C.; NERY, A. S.; ALBUQUERQUE, R. d. O. Quantum-resistant cryptography in fpga. In: *2022 Workshop on Communication Networks and Power Systems (WCNPS)*. [S.l.: s.n.], 2022. p. 1–5.

17 ALENEZI, M. N.; ALABDULRAZZAQ, H.; MOHAMMAD, N. Q. Symmetric encryption algorithms: Review and evaluation study. *International Journal of Communication Networks and Information Security*, Kohat University of Science and Technology (KUST), v. 12, n. 2, p. 256–272, 2020.

18 CHANDRA, S.; PAIRA, S.; ALAM, S. S.; SANYAL, G. A comparative survey of symmetric and asymmetric key cryptography. In: IEEE. *2014 international conference on electronics, communication and computational engineering (ICECCE)*. [S.l.], 2014. p. 83–93.

19 ABUTAHA, M.; FARAJALLAH, M.; TAHBOUB, R.; ODEH, M. Survey paper: cryptography is the science of information security. International Journal of Computer Science and Security (IJCSS), 2011.

20 MAQSOOD, F.; AHMED, M.; ALI, M. M.; SHAH, M. A. Cryptography: a comparative analysis for modern techniques. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 8, n. 6, 2017.

21 BELLARE, M.; ROGAWAY, P. Introduction to modern cryptography. *Ucsd Cse*, v. 207, p. 207, 2005.

22 SHOUP, V. A proposal for an iso standard for public key encryption. *Cryptology ePrint Archive*, 2001.

23 CRAMER, R.; SHOUP, V. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, SIAM, v. 33, n. 1, p. 167–226, 2003.

24 BELLARE, M.; DESAI, A.; POINTCHEVAL, D.; ROGAWAY, P. Relations among notions of security for public-key encryption schemes. In: SPRINGER. *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*. [S.l.], 1998. p. 26–45.

25 GOLDWASSER, S.; MICALI, S. Probabilistic encryption. *Journal of Computer and System Sciences*, Elsevier, v. 28, n. 2, p. 270–299, 1984.

26 DOLEV, D.; DWORK, C.; NAOR, M. Non-malleable cryptography. In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1991. p. 542–552.

27 NAOR, M.; YUNG, M. Public-key cryptosystems provably secure against chosen ciphertext attacks. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1990. p. 427–437.

28 RACKOFF, C.; SIMON, D. R. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In: SPRINGER. *Advances in Cryptology—CRYPTO'91: Proceedings*. [S.l.], 2001. p. 433–444.

29 BELLARE, M.; ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. [S.l.: s.n.], 1993. p. 62–73.

30 BLEUMER, G. Random oracle model. In: ____. *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011. p. 1027–1028. ISBN 978-1-4419-5906-5. Disponível em: <https://doi.org/10.1007/978-1-4419-5906-5_220>.

31 PRENEEL, B. Hash functions. In: ____. *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011. p. 543–553. ISBN 978-1-4419-5906-5. Disponível em: <https://doi.org/10.1007/978-1-4419-5906-5_580>.

32 BONEH, D.; DAGDELEN, Ö.; FISCHLIN, M.; LEHMANN, A.; SCHAFFNER, C.; ZHANDRY, M. Random oracles in a quantum world. In: SPRINGER. *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*. [S.l.], 2011. p. 41–69.

33 HOFHEINZ, D.; HÖVELMANNS, K.; KILTZ, E. A modular analysis of the fujisaki-okamoto transformation. In: SPRINGER. *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*. [S.l.], 2017. p. 341–371.

34 FUJISAKI, E.; OKAMOTO, T. Secure integration of asymmetric and symmetric encryption schemes. In: SPRINGER. *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*. [S.l.], 1999. p. 537–554.

35 MICCIANCIO, D.; REGEV, O. Lattice-based cryptography. *Post-quantum cryptography*, Springer, p. 147–191, 2009.

36 NGUYEN, P. Q.; STERN, J. The two faces of lattices in cryptology. In: SPRINGER. *Cryptography and Lattices: International Conference, CaLC 2001 Providence, RI, USA, March 29–30, 2001 Revised Papers*. [S.l.], 2001. p. 146–180.

37 AJTAI, M. Generating hard instances of lattice problems. In: *Proceedings of the 28th annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1996. p. 99–108.

38 AJTAI, M. The shortest vector problem in l2 is np-hard for randomized reductions. In: *Proceedings of the 30th annual ACM symposium on Theory of Computing*. [S.l.: s.n.], 1998. p. 10–19.

39 NEJATOLLAHI, H.; DUTT, N.; RAY, S.; REGAZZONI, F.; BANERJEE, I.; CAMMAROTA, R. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 6, p. 1–41, 2019.

40 REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 56, n. 6, p. 1–40, 2009.

41 LYUBASHEVSKY, V.; PEIKERT, C.; REGEV, O. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 60, n. 6, p. 1–35, 2013.

42 LANGLOIS, A.; STEHLÉ, D. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, Springer, v. 75, n. 3, p. 565–599, 2015.

43 BOS, J.; DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHANCK, J. M.; SCHWABE, P.; SEILER, G.; STEHLÉ, D. Crystals-kyber: a cca-secure module-lattice-based kem. In: IEEE. *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. [S.l.], 2018. p. 353–367.

44 AVANZI, R.; BOS, J.; DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHANCK, J.; SCHWABE, P.; SEILER, G.; STEHLÉ, D. *CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.02)*. 2021. <https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf>.

45 O'REGAN, G. Von neumann architecture. In: ____. *The Innovation in Computing Companion: A Compendium of Select, Pivotal Inventions*. Cham: Springer International Publishing, 2018. p. 257–259.

46 NEUMANN, J. von. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, v. 15, n. 4, p. 27–75, 1993.

47 EIGENMANN, R.; LILJA, D. J. Von neumann computers. *Wiley Encyclopedia of Electrical and Electronics Engineering*, Wiley, v. 23, p. 387–400, 1998.

48 BURKS, A. W.; GOLDSTINE, H. H.; NEUMANN, J. V. *Preliminary discussion of the logical design of an electronic computer instrument*. [S.l.], 1946.

49 GOLDSTINE, H. H.; NEUMANN, J. V. Planning and coding of problems for an electronic computing instrument. Institute for Advanced Study Princeton, NJ, 1947.

50 STALLINGS, W. *Arquitetura e Organização de Computadores 10a Edição*. [S.l.]: São Paulo: Pearson Education do Brasil, 2017.

51 BRODTKORB, A. R.; HAGEN, T. R.; SÆTRA, M. L. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, Elsevier, v. 73, n. 1, p. 4–13, 2013.

52 OWENS, J. D.; HOUSTON, M.; LUEBKE, D.; GREEN, S.; STONE, J. E.; PHILLIPS, J. C. Gpu computing. *Proceedings of the IEEE*, IEEE, v. 96, n. 5, p. 879–899, 2008.

53 COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, ACM New York, NY, USA, v. 34, n. 2, p. 171–210, 2002.

54 BROWN, S. D.; FRANCIS, R. J.; ROSE, J.; VRANESIC, Z. G. *Field-programmable gate arrays*. [S.l.]: Springer Science & Business Media, 1992. v. 180.

55 KUON, I.; TESSIER, R.; ROSE, J. et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, Now Publishers, Inc., v. 2, n. 2, p. 135–253, 2008.

56 GANDHARE, S.; KARTHIKEYAN, B. Survey on fpga architecture and recent applications. In: IEEE. *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*. [S.l.], 2019. p. 1–4.

57 HUFFMIRE, T.; IRVINE, C.; NGUYEN, T. D.; LEVIN, T.; KASTNER, R.; SHERWOOD, T. *Handbook of FPGA design security*. [S.l.]: Springer Science & Business Media, 2010.

58 BOUTROS, A.; BETZ, V. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, IEEE, v. 21, n. 2, p. 4–29, 2021.

59 CILETTI, M. D. *Advanced digital design with the Verilog HDL*. [S.l.]: Prentice hall Upper Saddle River, 2003. v. 1.

60 VAHID, F. *Digital design with RTL design, VHDL, and Verilog*. [S.l.]: John Wiley & Sons, 2010.

61 MASSOUMI, M. M. Hardware description. *Computer Science and Engineering*, EOLSS Publications, v. 15, p. 151, 2009.

62 IEEE Standard for VHDL Language Reference Manual. *IEEE Std 1076-2019*, p. 1–673, 2019.

63 IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, p. 1–590, 2006.

64 IEC/IEEE International Standard - VHDL Register Transfer Level (RTL) Synthesis. *IEC 62050 Ed. 1 (IEEE Std 1076.6-2004)*, p. 1–128, 2005.

65 PEDRONI, V. A. *Circuit design with VHDL*. [S.l.]: MIT press, 2020.

66 WAKERLY, J. F. *Digital Design: Principles and Practices (5th Edition)*. 5th. ed. [S.l.]: Pearson, 2017. ISBN 013446009X.

67 COUSSY, P.; GAJSKI, D. D.; MEREDITH, M.; TAKACH, A. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, IEEE, v. 26, n. 4, p. 8–17, 2009.

68 NANE, R.; SIMA, V.-M.; PILATO, C.; CHOI, J.; FORT, B.; CANIS, A.; CHEN, Y. T.; HSIAO, H.; BROWN, S.; FERRANDI, F. et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 35, n. 10, p. 1591–1604, 2015.

69 CONG, J.; LAU, J.; LIU, G.; NEUENDORFFER, S.; PAN, P.; VISSERS, K.; ZHANG, Z. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, ACM New York, NY, v. 15, n. 4, p. 1–42, 2022.

70 BASU, K.; SONI, D.; NABEEL, M.; KARRI, R. Nist post-quantum cryptography-a hardware evaluation study. *Cryptology ePrint Archive*, 2019.

71 HUANG, Y.; HUANG, M.; LEI, Z.; WU, J. A pure hardware implementation of crystals-kyber pqc algorithm through resource reuse. *IEICE Electronics Express*, The Institute of Electronics, Information and Communication Engineers, p. 17–20200234, 2020.

72 XING, Y.; LI, S. A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, p. 328–356, 2021.

73 BISHEH-NIASAR, M.; AZARDERAKHSH, R.; MOZAFFARI-KERMANI, M. Instruction-set accelerated implementation of crystals-kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, IEEE, v. 68, n. 11, p. 4648–4659, 2021.

74 GUO, W.; LI, S.; KONG, L. An efficient implementation of kyber. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 69, n. 3, p. 1562–1566, 2022.

75 BOS, J.; DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHANCK, J. M.; SCHWABE, P.; SEILER, G.; STEHLÉ, D. *PQ-CRYSTALS/Kyber*. [S.l.]: online on GitHub, 2018. <https://github.com/pq-crystals/kyber>.

76 XILINX. *Vitis HLS*. Disponível em: <https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html>. Acesso em: 04 ago. 2022.

77 XILINX. *Vivado*. Disponível em: <https://www.xilinx.com/products/design-tools/vivado.html>. Acesso em: 04 ago. 2022.

78 ARM. *AMBA® AXI Protocol Specification*. Disponível em: <https://developer.arm.com/documentation/ihi0022/latest/>. Acesso em: 05 mai. 2023.

APÊNDICES

I.1 CÓDIGO DAS FUNÇÕES IMPLEMENTADAS NO ACELERADOR KYBER

Código 1: Funções executadas pelo acelerador Kyber

```
int16_t montgomery_reduce_hls(int32_t a)
{
  int16_t t;

  t = (int16_t)a*QINV_hls;
  t = (a - (int32_t)t*KYBER_Q_hls) >> 16;
  return t;
}

int16_t fqmul_hls(int16_t a, int16_t b)
{
  return montgomery_reduce_hls((int32_t)a*b);
}

void ntt_hls(int16_t r[256])
{
  unsigned int len, start, j, k;
  int16_t t, zeta;

  k = 1;
  for(len = 128; len >= 2; len >>= 1) {
    for(start = 0; start < 256; start = j + len) {
      zeta = zetas_hls[k++];
      for(j = start; j < start + len; j++) {
        t = fqmul_hls(zeta, r[j + len]);
        r[j + len] = r[j] - t;
        r[j] = r[j] + t;
      }
    }
  }
}

int16_t barrett_reduce_hls(int16_t a)
{
  int16_t t;
  const int16_t v = ((1<<26) + KYBER_Q_hls/2)/KYBER_Q_hls;

  t  = ((int32_t)v*a + (1<<25)) >> 26;
  t *= KYBER_Q_hls;

  return a - t;
}

void poly_reduce_hls(poly *r)
{
  unsigned int i;
```

```
47
48    for(i=0;i<KYBER_N_hls;i++)
49      r->coeffs[i] = barrett_reduce_hls(r->coeffs[i]);
50  }
51
52  void poly_ntt_hls(poly *r)
53  {
54    ntt_hls(r->coeffs);
55    poly_reduce_hls(r);
56  }
57
58  void polyvec_ntt_hls(polyvec *r)
59  {
60    unsigned int i;
61
62    for(i=0;i<KYBER_K_hls;i++)
63      poly_ntt_hls(&r->vec[i]);
64  }
65
66  void poly_add_hls(poly *r, const poly *a, const poly *b)
67  {
68    unsigned int i;
69
70    for(i=0;i<KYBER_N_hls;i++)
71      r->coeffs[i] = a->coeffs[i] + b->coeffs[i];
72  }
73
74  void basemul_hls(int16_t r[2], const int16_t a[2], const int16_t b[2],
75                   int16_t zeta)
76  {
77    r[0]  = fqmul_hls(a[1], b[1]);
78    r[0]  = fqmul_hls(r[0], zeta);
79    r[0] += fqmul_hls(a[0], b[0]);
80    r[1]  = fqmul_hls(a[0], b[1]);
81    r[1] += fqmul_hls(a[1], b[0]);
82  }
83
84  void poly_basemul_montgomery_hls(poly *r, const poly *a, const poly *b)
85  {
86    unsigned int i;
87
88    for(i=0;i<KYBER_N_hls/4;i++) {
89      basemul_hls(&r->coeffs[4*i], &a->coeffs[4*i], &b->coeffs[4*i],
90                  zetas_hls[64+i]);
91      basemul_hls(&r->coeffs[4*i+2], &a->coeffs[4*i+2], &b->coeffs[4*i+2],
92                  -zetas_hls[64+i]);
93    }
94  }
95
96  void polyvec_basemul_acc_montgomery_hls(poly *r, const polyvec *a,
97                                          const polyvec *b)
98  {
```

```
99    unsigned int i;
100   poly t;
101
102   poly_basemul_montgomery_hls(r, &a->vec[0], &b->vec[0]);
103
104   for(i=1;i<KYBER_K_hls;i++) {
105     poly_basemul_montgomery_hls(&t, &a->vec[i], &b->vec[i]);
106     poly_add_hls(r, r, &t);
107   }
108
109   poly_reduce_hls(r);
110 }
111
112 void poly_tomont_hls(poly *r)
113 {
114   unsigned int i;
115
116   const int16_t f = (1ULL << 32) % KYBER_Q_hls;
117   for(i=0;i<KYBER_N_hls;i++)
118     r->coeffs[i] = montgomery_reduce_hls((int32_t)r->coeffs[i]*f);
119 }
120
121 void polyvec_reduce_hls(polyvec *r)
122 {
123   unsigned int i;
124
125   for(i=0;i<KYBER_K_hls;i++)
126     poly_reduce_hls(&r->vec[i]);
127 }
128
129 void polyvec_add_hls(polyvec *r, const polyvec *a, const polyvec *b)
130 {
131   unsigned int i;
132
133   for(i=0;i<KYBER_K_hls;i++)
134     poly_add_hls(&r->vec[i], &a->vec[i], &b->vec[i]);
135 }
136
137 void invntt_hls(int16_t r[256])
138 {
139   unsigned int start, len, j, k;
140   int16_t t, zeta;
141   const int16_t f = 1441; // mont^2/128
142
143   k = 127;
144   for(len = 2; len <= 128; len <<= 1) {
145     for(start = 0; start < 256; start = j + len) {
146       zeta = zetas_hls[k--];
147       for(j = start; j < start + len; j++) {
148         t = r[j];
149         r[j] = barrett_reduce_hls(t + r[j + len]);
150         r[j + len] = r[j + len] - t;
```

```
151           r[j + len] = fqmul_hls(zeta, r[j + len]);
152         }
153       }
154     }
155
156     for(j = 0; j < 256; j++)
157       r[j] = fqmul_hls(r[j], f);
158   }
159
160   void poly_invntt_tomont_hls(poly *r)
161   {
162     invntt_hls(r->coeffs);
163   }
164
165   void polyvec_invntt_tomont_hls(polyvec *r)
166   {
167     unsigned int i;
168
169     for(i=0;i<KYBER_K_hls;i++)
170       poly_invntt_tomont_hls(&r->vec[i]);
171   }
172
173   void poly_tomsg_hls(uint8_t msg[KYBER_INDCPA_MSGBYTES_hls], const poly *a)
174   {
175     unsigned int i,j;
176     uint16_t t;
177
178     for(i=0;i<KYBER_N_hls/8;i++) {
179       msg[i] = 0;
180       for(j=0;j<8;j++) {
181         t  = a->coeffs[8*i+j];
182         t += ((int16_t)t >> 15) & KYBER_Q_hls;
183         t  = (((t << 1) + KYBER_Q_hls/2)/KYBER_Q_hls) & 1;
184         msg[i] |= t << j;
185       }
186     }
187   }
188
189   void poly_sub_hls(poly *r, const poly *a, const poly *b)
190   {
191     unsigned int i;
192
193     for(i=0;i<KYBER_N_hls;i++)
194       r->coeffs[i] = a->coeffs[i] - b->coeffs[i];
195   }
196
197   void polyvec_decompress_hls(polyvec *r,
198                               const uint8_t a[KYBER_POLYVECCOMPRESSEDBYTES_hls])
199   {
200     unsigned int i,j,k;
201
202   #if (KYBER_POLYVECCOMPRESSEDBYTES_hls == (KYBER_K_hls * 352))
```

```
203   uint16_t t[8];
204   for(i=0;i<KYBER_K_hls;i++) {
205     for(j=0;j<KYBER_N_hls/8;j++) {
206       t[0] = (a[0] >> 0) | ((uint16_t)a[ 1] << 8);
207       t[1] = (a[1] >> 3) | ((uint16_t)a[ 2] << 5);
208       t[2] = (a[2] >> 6) | ((uint16_t)a[ 3] << 2) | ((uint16_t)a[4] << 10);
209       t[3] = (a[4] >> 1) | ((uint16_t)a[ 5] << 7);
210       t[4] = (a[5] >> 4) | ((uint16_t)a[ 6] << 4);
211       t[5] = (a[6] >> 7) | ((uint16_t)a[ 7] << 1) | ((uint16_t)a[8] << 9);
212       t[6] = (a[8] >> 2) | ((uint16_t)a[ 9] << 6);
213       t[7] = (a[9] >> 5) | ((uint16_t)a[10] << 3);
214       a += 11;
215
216       for(k=0;k<8;k++)
217         r->vec[i].coeffs[8*j+k] = ((uint32_t)(t[k] & 0x7FF)*KYBER_Q_hls + 1024)
218                                   >> 11;
219     }
220   }
221 #elif (KYBER_POLYVECCOMPRESSEDBYTES_hls == (KYBER_K_hls * 320))
222   uint16_t t[4];
223   for(i=0;i<KYBER_K_hls;i++) {
224     for(j=0;j<KYBER_N_hls/4;j++) {
225       t[0] = (a[0] >> 0) | ((uint16_t)a[1] << 8);
226       t[1] = (a[1] >> 2) | ((uint16_t)a[2] << 6);
227       t[2] = (a[2] >> 4) | ((uint16_t)a[3] << 4);
228       t[3] = (a[3] >> 6) | ((uint16_t)a[4] << 2);
229       a += 5;
230
231       for(k=0;k<4;k++)
232         r->vec[i].coeffs[4*j+k] = ((uint32_t)(t[k] & 0x3FF)*KYBER_Q_hls + 512)
233                                   >> 10;
234     }
235   }
236 #else
237 #error "KYBER_POLYVECCOMPRESSEDBYTES needs to be in {320*KYBER_K, 352*KYBER_K}"
238 #endif
239 }
240
241 void poly_decompress_hls(poly *r, const uint8_t a[KYBER_POLYCOMPRESSEDBYTES_hls])
242 {
243   unsigned int i;
244
245 #if (KYBER_POLYCOMPRESSEDBYTES_hls == 128)
246   for(i=0;i<KYBER_N_hls/2;i++) {
247     r->coeffs[2*i+0] = (((uint16_t)(a[0] & 15)*KYBER_Q_hls) + 8) >> 4;
248     r->coeffs[2*i+1] = (((uint16_t)(a[0] >> 4)*KYBER_Q_hls) + 8) >> 4;
249     a += 1;
250   }
251 #elif (KYBER_POLYCOMPRESSEDBYTES_hls == 160)
252   unsigned int j;
253   uint8_t t[8];
254   for(i=0;i<KYBER_N_hls/8;i++) {
```

```
255       t[0] = (a[0] >> 0);
256       t[1] = (a[0] >> 5) | (a[1] << 3);
257       t[2] = (a[1] >> 2);
258       t[3] = (a[1] >> 7) | (a[2] << 1);
259       t[4] = (a[2] >> 4) | (a[3] << 4);
260       t[5] = (a[3] >> 1);
261       t[6] = (a[3] >> 6) | (a[4] << 2);
262       t[7] = (a[4] >> 3);
263       a += 5;
264
265       for(j=0;j<8;j++)
266         r->coeffs[8*i+j] = ((uint32_t)(t[j] & 31)*KYBER_Q_hls + 16) >> 5;
267     }
268   #else
269   #error "KYBER_POLYCOMPRESSEDBYTES needs to be in {128, 160}"
270   #endif
271   }
272
273   static void unpack_ciphertext_hls(polyvec *b, poly *v,
274                                     const uint8_t c[KYBER_INDCPA_BYTES_hls])
275   {
276     polyvec_decompress_hls(b, c);
277     poly_decompress_hls(v, c+KYBER_POLYVECCOMPRESSEDBYTES_hls);
278   }
279
280   void poly_frombytes_hls(poly *r, const uint8_t a[KYBER_POLYBYTES_hls])
281   {
282     unsigned int i;
283
284     for(i=0;i<KYBER_N_hls/2;i++) {
285       r->coeffs[2*i]   = ((a[3*i+0] >> 0) | ((uint16_t)a[3*i+1] << 8)) & 0xFFF;
286       r->coeffs[2*i+1] = ((a[3*i+1] >> 4) | ((uint16_t)a[3*i+2] << 4)) & 0xFFF;
287     }
288   }
289
290   void polyvec_frombytes_hls(polyvec *r, const uint8_t a[KYBER_POLYVECBYTES_hls])
291   {
292     unsigned int i;
293
294     for(i=0;i<KYBER_K_hls;i++)
295       poly_frombytes_hls(&r->vec[i], a+i*KYBER_POLYBYTES_hls);
296   }
297
298   static void unpack_sk_hls(polyvec *sk,
299                             const uint8_t packedsk[KYBER_INDCPA_SECRETKEYBYTES_hls])
300   {
301     polyvec_frombytes_hls(sk, packedsk);
302   }
303
304   static void unpack_pk_hls(polyvec *pk, uint8_t seed[KYBER_SYMBYTES_hls],
305                             const uint8_t packedpk[KYBER_INDCPA_PUBLICKEYBYTES_hls])
306   {
```

```
  size_t i;

  polyvec_frombytes_hls(pk, packedpk);
  for(i=0;i<KYBER_SYMBYTES_hls;i++)
    seed[i] = packedpk[i+KYBER_POLYVECBYTES_hls];
}

void poly_frommsg_hls(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES_hls])
{
  unsigned int i,j;
  int16_t mask;

#if (KYBER_INDCPA_MSGBYTES_hls != KYBER_N_hls/8)
#error "KYBER_INDCPA_MSGBYTES must be equal to KYBER_N/8 bytes!"
#endif

  for(i=0;i<KYBER_N_hls/8;i++) {
    for(j=0;j<8;j++) {
      mask = -(int16_t)((msg[i] >> j)&1);
      r->coeffs[8*i+j] = mask & ((KYBER_Q_hls+1)/2);
    }
  }
}

static const uint64_t KeccakF_RoundConstants_hls[NROUNDS_hls] = {
  (uint64_t)0x0000000000000001ULL,
  (uint64_t)0x0000000000008082ULL,
  (uint64_t)0x800000000000808aULL,
  (uint64_t)0x8000000080008000ULL,
  (uint64_t)0x000000000000808bULL,
  (uint64_t)0x0000000080000001ULL,
  (uint64_t)0x8000000080008081ULL,
  (uint64_t)0x8000000000008009ULL,
  (uint64_t)0x000000000000008aULL,
  (uint64_t)0x0000000000000088ULL,
  (uint64_t)0x0000000080008009ULL,
  (uint64_t)0x000000008000000aULL,
  (uint64_t)0x000000008000808bULL,
  (uint64_t)0x800000000000008bULL,
  (uint64_t)0x8000000000008089ULL,
  (uint64_t)0x8000000000008003ULL,
  (uint64_t)0x8000000000008002ULL,
  (uint64_t)0x8000000000000080ULL,
  (uint64_t)0x000000000000800aULL,
  (uint64_t)0x800000008000000aULL,
  (uint64_t)0x8000000080008081ULL,
  (uint64_t)0x8000000000008080ULL,
  (uint64_t)0x0000000080000001ULL,
  (uint64_t)0x8000000080008008ULL
};

static void KeccakF1600_StatePermute_hls(uint64_t state[25])
```

```
359 {
360         int round;
361 
362         uint64_t Aba, Abe, Abi, Abo, Abu;
363         uint64_t Aga, Age, Agi, Ago, Agu;
364         uint64_t Aka, Ake, Aki, Ako, Aku;
365         uint64_t Ama, Ame, Ami, Amo, Amu;
366         uint64_t Asa, Ase, Asi, Aso, Asu;
367         uint64_t BCa, BCe, BCi, BCo, BCu;
368         uint64_t Da, De, Di, Do, Du;
369         uint64_t Eba, Ebe, Ebi, Ebo, Ebu;
370         uint64_t Ega, Ege, Egi, Ego, Egu;
371         uint64_t Eka, Eke, Eki, Eko, Eku;
372         uint64_t Ema, Eme, Emi, Emo, Emu;
373         uint64_t Esa, Ese, Esi, Eso, Esu;
374 
375         //copyFromState(A, state)
376         Aba = state[ 0];
377         Abe = state[ 1];
378         Abi = state[ 2];
379         Abo = state[ 3];
380         Abu = state[ 4];
381         Aga = state[ 5];
382         Age = state[ 6];
383         Agi = state[ 7];
384         Ago = state[ 8];
385         Agu = state[ 9];
386         Aka = state[10];
387         Ake = state[11];
388         Aki = state[12];
389         Ako = state[13];
390         Aku = state[14];
391         Ama = state[15];
392         Ame = state[16];
393         Ami = state[17];
394         Amo = state[18];
395         Amu = state[19];
396         Asa = state[20];
397         Ase = state[21];
398         Asi = state[22];
399         Aso = state[23];
400         Asu = state[24];
401 
402         for(round = 0; round < NROUNDS_hls; round += 2) {
403             //    prepareTheta
404             BCa = Aba^Aga^Aka^Ama^Asa;
405             BCe = Abe^Age^Ake^Ame^Ase;
406             BCi = Abi^Agi^Aki^Ami^Asi;
407             BCo = Abo^Ago^Ako^Amo^Aso;
408             BCu = Abu^Agu^Aku^Amu^Asu;
409 
410             //thetaRhoPiChiIotaPrepareTheta(round, A, E)
```

```
411             Da = BCu^ROL_hls(BCe, 1);
412             De = BCa^ROL_hls(BCi, 1);
413             Di = BCe^ROL_hls(BCo, 1);
414             Do = BCi^ROL_hls(BCu, 1);
415             Du = BCo^ROL_hls(BCa, 1);
416
417             Aba ^= Da;
418             BCa = Aba;
419             Age ^= De;
420             BCe = ROL_hls(Age, 44);
421             Aki ^= Di;
422             BCi = ROL_hls(Aki, 43);
423             Amo ^= Do;
424             BCo = ROL_hls(Amo, 21);
425             Asu ^= Du;
426             BCu = ROL_hls(Asu, 14);
427             Eba =   BCa ^((~BCe)&  BCi );
428             Eba ^= (uint64_t)KeccakF_RoundConstants_hls[round];
429             Ebe =   BCe ^((~BCi)&  BCo );
430             Ebi =   BCi ^((~BCo)&  BCu );
431             Ebo =   BCo ^((~BCu)&  BCa );
432             Ebu =   BCu ^((~BCa)&  BCe );
433
434             Abo ^= Do;
435             BCa = ROL_hls(Abo, 28);
436             Agu ^= Du;
437             BCe = ROL_hls(Agu, 20);
438             Aka ^= Da;
439             BCi = ROL_hls(Aka,  3);
440             Ame ^= De;
441             BCo = ROL_hls(Ame, 45);
442             Asi ^= Di;
443             BCu = ROL_hls(Asi, 61);
444             Ega =   BCa ^((~BCe)&  BCi );
445             Ege =   BCe ^((~BCi)&  BCo );
446             Egi =   BCi ^((~BCo)&  BCu );
447             Ego =   BCo ^((~BCu)&  BCa );
448             Egu =   BCu ^((~BCa)&  BCe );
449
450             Abe ^= De;
451             BCa = ROL_hls(Abe,  1);
452             Agi ^= Di;
453             BCe = ROL_hls(Agi,  6);
454             Ako ^= Do;
455             BCi = ROL_hls(Ako, 25);
456             Amu ^= Du;
457             BCo = ROL_hls(Amu,  8);
458             Asa ^= Da;
459             BCu = ROL_hls(Asa, 18);
460             Eka =   BCa ^((~BCe)&  BCi );
461             Eke =   BCe ^((~BCi)&  BCo );
462             Eki =   BCi ^((~BCo)&  BCu );
```

```
            Eko =   BCo ^((~BCu)&  BCa );
            Eku =   BCu ^((~BCa)&  BCe );

            Abu ^= Du;
            BCa = ROL_hls(Abu, 27);
            Aga ^= Da;
            BCe = ROL_hls(Aga, 36);
            Ake ^= De;
            BCi = ROL_hls(Ake, 10);
            Ami ^= Di;
            BCo = ROL_hls(Ami, 15);
            Aso ^= Do;
            BCu = ROL_hls(Aso, 56);
            Ema =   BCa ^((~BCe)&  BCi );
            Eme =   BCe ^((~BCi)&  BCo );
            Emi =   BCi ^((~BCo)&  BCu );
            Emo =   BCo ^((~BCu)&  BCa );
            Emu =   BCu ^((~BCa)&  BCe );

            Abi ^= Di;
            BCa = ROL_hls(Abi, 62);
            Ago ^= Do;
            BCe = ROL_hls(Ago, 55);
            Aku ^= Du;
            BCi = ROL_hls(Aku, 39);
            Ama ^= Da;
            BCo = ROL_hls(Ama, 41);
            Ase ^= De;
            BCu = ROL_hls(Ase,  2);
            Esa =   BCa ^((~BCe)&  BCi );
            Ese =   BCe ^((~BCi)&  BCo );
            Esi =   BCi ^((~BCo)&  BCu );
            Eso =   BCo ^((~BCu)&  BCa );
            Esu =   BCu ^((~BCa)&  BCe );

            //    prepareTheta
            BCa = Eba^Ega^Eka^Ema^Esa;
            BCe = Ebe^Ege^Eke^Eme^Ese;
            BCi = Ebi^Egi^Eki^Emi^Esi;
            BCo = Ebo^Ego^Eko^Emo^Eso;
            BCu = Ebu^Egu^Eku^Emu^Esu;

        //thetaRhoPiChiIotaPrepareTheta(round+1, E, A)
            Da = BCu^ROL_hls(BCe, 1);
            De = BCa^ROL_hls(BCi, 1);
            Di = BCe^ROL_hls(BCo, 1);
            Do = BCi^ROL_hls(BCu, 1);
            Du = BCo^ROL_hls(BCa, 1);

            Eba ^= Da;
            BCa = Eba;
            Ege ^= De;
```

```
515             BCe = ROL_hls(Ege, 44);
516             Eki ^= Di;
517             BCi = ROL_hls(Eki, 43);
518             Emo ^= Do;
519             BCo = ROL_hls(Emo, 21);
520             Esu ^= Du;
521             BCu = ROL_hls(Esu, 14);
522             Aba =   BCa ^((~BCe)&  BCi );
523             Aba ^= (uint64_t)KeccakF_RoundConstants_hls[round+1];
524             Abe =   BCe ^((~BCi)&  BCo );
525             Abi =   BCi ^((~BCo)&  BCu );
526             Abo =   BCo ^((~BCu)&  BCa );
527             Abu =   BCu ^((~BCa)&  BCe );
528
529             Ebo ^= Do;
530             BCa = ROL_hls(Ebo, 28);
531             Egu ^= Du;
532             BCe = ROL_hls(Egu, 20);
533             Eka ^= Da;
534             BCi = ROL_hls(Eka, 3);
535             Eme ^= De;
536             BCo = ROL_hls(Eme, 45);
537             Esi ^= Di;
538             BCu = ROL_hls(Esi, 61);
539             Aga =   BCa ^((~BCe)&  BCi );
540             Age =   BCe ^((~BCi)&  BCo );
541             Agi =   BCi ^((~BCo)&  BCu );
542             Ago =   BCo ^((~BCu)&  BCa );
543             Agu =   BCu ^((~BCa)&  BCe );
544
545             Ebe ^= De;
546             BCa = ROL_hls(Ebe, 1);
547             Egi ^= Di;
548             BCe = ROL_hls(Egi, 6);
549             Eko ^= Do;
550             BCi = ROL_hls(Eko, 25);
551             Emu ^= Du;
552             BCo = ROL_hls(Emu, 8);
553             Esa ^= Da;
554             BCu = ROL_hls(Esa, 18);
555             Aka =   BCa ^((~BCe)&  BCi );
556             Ake =   BCe ^((~BCi)&  BCo );
557             Aki =   BCi ^((~BCo)&  BCu );
558             Ako =   BCo ^((~BCu)&  BCa );
559             Aku =   BCu ^((~BCa)&  BCe );
560
561             Ebu ^= Du;
562             BCa = ROL_hls(Ebu, 27);
563             Ega ^= Da;
564             BCe = ROL_hls(Ega, 36);
565             Eke ^= De;
566             BCi = ROL_hls(Eke, 10);
```

```
567             Emi ^= Di;
568             BCo = ROL_hls(Emi, 15);
569             Eso ^= Do;
570             BCu = ROL_hls(Eso, 56);
571             Ama =   BCa ^((~BCe)&  BCi );
572             Ame =   BCe ^((~BCi)&  BCo );
573             Ami =   BCi ^((~BCo)&  BCu );
574             Amo =   BCo ^((~BCu)&  BCa );
575             Amu =   BCu ^((~BCa)&  BCe );
576 
577             Ebi ^= Di;
578             BCa = ROL_hls(Ebi, 62);
579             Ego ^= Do;
580             BCe = ROL_hls(Ego, 55);
581             Eku ^= Du;
582             BCi = ROL_hls(Eku, 39);
583             Ema ^= Da;
584             BCo = ROL_hls(Ema, 41);
585             Ese ^= De;
586             BCu = ROL_hls(Ese, 2);
587             Asa =   BCa ^((~BCe)&  BCi );
588             Ase =   BCe ^((~BCi)&  BCo );
589             Asi =   BCi ^((~BCo)&  BCu );
590             Aso =   BCo ^((~BCu)&  BCa );
591             Asu =   BCu ^((~BCa)&  BCe );
592         }
593 
594         //copyToState(state, A)
595         state[ 0] = Aba;
596         state[ 1] = Abe;
597         state[ 2] = Abi;
598         state[ 3] = Abo;
599         state[ 4] = Abu;
600         state[ 5] = Aga;
601         state[ 6] = Age;
602         state[ 7] = Agi;
603         state[ 8] = Ago;
604         state[ 9] = Agu;
605         state[10] = Aka;
606         state[11] = Ake;
607         state[12] = Aki;
608         state[13] = Ako;
609         state[14] = Aku;
610         state[15] = Ama;
611         state[16] = Ame;
612         state[17] = Ami;
613         state[18] = Amo;
614         state[19] = Amu;
615         state[20] = Asa;
616         state[21] = Ase;
617         state[22] = Asi;
618         state[23] = Aso;
```

```
619         state[24] = Asu;
620 }
621
622 static uint64_t load64_hls(const uint8_t x[8])
623 {
624   unsigned int i;
625   uint64_t r = 0;
626
627   for(i=0;i<8;i++)
628     r |= (uint64_t)x[i] << 8*i;
629
630   return r;
631 }
632
633 static void keccak_absorb_once_hls(uint64_t s[25], unsigned int r,
634                                    const uint8_t *in, size_t inlen, uint8_t p)
635 {
636   unsigned int i;
637
638   for(i=0;i<25;i++)
639     s[i] = 0;
640
641   while(inlen >= r) {
642     for(i=0;i<r/8;i++)
643       s[i] ^= load64_hls(in+8*i);
644     in += r;
645     inlen -= r;
646     KeccakF1600_StatePermute_hls(s);
647   }
648
649   for(i=0;i<inlen;i++)
650     s[i/8] ^= (uint64_t)in[i] << 8*(i%8);
651
652   s[i/8] ^= (uint64_t)p << 8*(i%8);
653   s[(r-1)/8] ^= 1ULL << 63;
654 }
655
656 void shake128_absorb_once_hls(keccak_state *state, const uint8_t *in, size_t inlen)
657 {
658   keccak_absorb_once_hls(state->s, SHAKE128_RATE_hls, in, inlen, 0x1F);
659   state->pos = SHAKE128_RATE_hls;
660 }
661
662 void kyber_shake128_absorb_hls(keccak_state *state,
663                                const uint8_t seed[KYBER_SYMBYTES_hls], uint8_t x,
664                                uint8_t y)
665 {
666   uint8_t extseed[KYBER_SYMBYTES_hls+2];
667
668   memcpy(extseed, seed, KYBER_SYMBYTES_hls);
669   extseed[KYBER_SYMBYTES_hls+0] = x;
670   extseed[KYBER_SYMBYTES_hls+1] = y;
```

```
671
672   shake128_absorb_once_hls(state, extseed, sizeof(extseed));
673 }
674
675 static void store64_hls(uint8_t x[8], uint64_t u) {
676   unsigned int i;
677
678   for(i=0;i<8;i++)
679     x[i] = u >> 8*i;
680 }
681
682 static void keccak_squeezeblocks_hls(uint8_t *out, size_t nblocks, uint64_t s[25],
683                                      unsigned int r)
684 {
685   unsigned int i;
686
687   while(nblocks) {
688     KeccakF1600_StatePermute_hls(s);
689     for(i=0;i<r/8;i++)
690       store64_hls(out+8*i, s[i]);
691     out += r;
692     nblocks -= 1;
693   }
694 }
695
696 void shake128_squeezeblocks_hls(uint8_t *out, size_t nblocks, keccak_state *state)
697 {
698   keccak_squeezeblocks_hls(out, nblocks, state->s, SHAKE128_RATE_hls);
699 }
700
701 static unsigned int rej_uniform_hls(int16_t *r, unsigned int len,
702                                     const uint8_t *buf, unsigned int buflen)
703 {
704   unsigned int ctr, pos;
705   uint16_t val0, val1;
706
707   ctr = pos = 0;
708   while(ctr < len && pos + 3 <= buflen) {
709     val0 = ((buf[pos+0] >> 0) | ((uint16_t)buf[pos+1] << 8)) & 0xFFF;
710     val1 = ((buf[pos+1] >> 4) | ((uint16_t)buf[pos+2] << 4)) & 0xFFF;
711     pos += 3;
712
713     if(val0 < KYBER_Q_hls)
714       r[ctr++] = val0;
715     if(ctr < len && val1 < KYBER_Q_hls)
716       r[ctr++] = val1;
717   }
718
719   return ctr;
720 }
721
722 void gen_matrix_hls(polyvec *a, const uint8_t seed[KYBER_SYMBYTES_hls],
```

```
723                     int transposed)
724 {
725   unsigned int ctr, i, j, k;
726   unsigned int buflen, off;
727   uint8_t buf[GEN_MATRIX_NBLOCKS_hls*XOF_BLOCKBYTES_hls+2];
728   xof_state state;
729
730   for(i=0;i<KYBER_K_hls;i++) {
731     for(j=0;j<KYBER_K_hls;j++) {
732       if(transposed)
733         xof_absorb_hls(&state, seed, i, j);
734       else
735         xof_absorb_hls(&state, seed, j, i);
736
737       xof_squeezeblocks_hls(buf, GEN_MATRIX_NBLOCKS_hls, &state);
738       buflen = GEN_MATRIX_NBLOCKS_hls*XOF_BLOCKBYTES_hls;
739       ctr = rej_uniform_hls(a[i].vec[j].coeffs, KYBER_N_hls, buf, buflen);
740
741       while(ctr < KYBER_N_hls) {
742         off = buflen % 3;
743         for(k = 0; k < off; k++)
744           buf[k] = buf[buflen - off + k];
745         xof_squeezeblocks_hls(buf + off, 1, &state);
746         buflen = off + XOF_BLOCKBYTES_hls;
747         ctr += rej_uniform_hls(a[i].vec[j].coeffs + ctr, KYBER_N_hls
748                               - ctr, buf, buflen);
749       }
750     }
751   }
752 }
753
754 void shake256_absorb_once_hls(keccak_state *state, const uint8_t *in, size_t inlen)
755 {
756   keccak_absorb_once_hls(state->s, SHAKE256_RATE_hls, in, inlen, 0x1F);
757   state->pos = SHAKE256_RATE_hls;
758 }
759
760 void shake256_squeezeblocks_hls(uint8_t *out, size_t nblocks, keccak_state *state)
761 {
762   keccak_squeezeblocks_hls(out, nblocks, state->s, SHAKE256_RATE_hls);
763 }
764
765 static unsigned int keccak_squeeze_hls(uint8_t *out, size_t outlen, uint64_t s[25],
766                                        unsigned int pos, unsigned int r)
767 {
768   unsigned int i;
769
770   while(outlen) {
771     if(pos == r) {
772       KeccakF1600_StatePermute_hls(s);
773       pos = 0;
774     }
```

```
775     for(i=pos;i < r && i < pos+outlen; i++)
776       *out++ = s[i/8] >> 8*(i%8);
777     outlen -= i-pos;
778     pos = i;
779   }
780 
781   return pos;
782 }
783 
784 void shake256_squeeze_hls(uint8_t *out, size_t outlen, keccak_state *state)
785 {
786   state->pos = keccak_squeeze_hls(out, outlen, state->s, state->pos,
787                     SHAKE256_RATE_hls);
788 }
789 
790 void shake256_hls(uint8_t *out, size_t outlen, const uint8_t *in, size_t inlen)
791 {
792   size_t nblocks;
793   keccak_state state;
794 
795   shake256_absorb_once_hls(&state, in, inlen);
796   nblocks = outlen/SHAKE256_RATE_hls;
797   shake256_squeezeblocks_hls(out, nblocks, &state);
798   outlen -= nblocks*SHAKE256_RATE_hls;
799   out += nblocks*SHAKE256_RATE_hls;
800   shake256_squeeze_hls(out, outlen, &state);
801 }
802 
803 void kyber_shake256_prf_hls(uint8_t *out, size_t outlen,
804                             const uint8_t key[KYBER_SYMBYTES_hls], uint8_t nonce)
805 {
806   uint8_t extkey[KYBER_SYMBYTES_hls+1];
807 
808   memcpy(extkey, key, KYBER_SYMBYTES_hls);
809   extkey[KYBER_SYMBYTES_hls] = nonce;
810 
811   shake256_hls(out, outlen, extkey, sizeof(extkey));
812 }
813 
814 static uint32_t load32_littleendian_hls(const uint8_t x[4])
815 {
816   uint32_t r;
817   r  = (uint32_t)x[0];
818   r |= (uint32_t)x[1] << 8;
819   r |= (uint32_t)x[2] << 16;
820   r |= (uint32_t)x[3] << 24;
821   return r;
822 }
823 
824 static void cbd2_hls(poly *r, const uint8_t buf[2*KYBER_N_hls/4])
825 {
826   unsigned int i,j;
```

```
827   uint32_t t,d;
828   int16_t a,b;
829
830   for(i=0;i<KYBER_N_hls/8;i++) {
831     t  = load32_littleendian_hls(buf+4*i);
832     d  = t & 0x55555555;
833     d += (t>>1) & 0x55555555;
834
835     for(j=0;j<8;j++) {
836       a = (d >> (4*j+0)) & 0x3;
837       b = (d >> (4*j+2)) & 0x3;
838       r->coeffs[8*i+j] = a - b;
839     }
840   }
841 }
842
843 static uint32_t load24_littleendian_hls(const uint8_t x[3])
844 {
845   uint32_t r;
846   r  = (uint32_t)x[0];
847   r |= (uint32_t)x[1] << 8;
848   r |= (uint32_t)x[2] << 16;
849   return r;
850 }
851
852 static void cbd3_hls(poly *r, const uint8_t buf[3*KYBER_N_hls/4])
853 {
854   unsigned int i,j;
855   uint32_t t,d;
856   int16_t a,b;
857
858   for(i=0;i<KYBER_N_hls/4;i++) {
859     t  = load24_littleendian_hls(buf+3*i);
860     d  = t & 0x00249249;
861     d += (t>>1) & 0x00249249;
862     d += (t>>2) & 0x00249249;
863
864     for(j=0;j<4;j++) {
865       a = (d >> (6*j+0)) & 0x7;
866       b = (d >> (6*j+3)) & 0x7;
867       r->coeffs[4*i+j] = a - b;
868     }
869   }
870 }
871
872 void poly_cbd_eta1_hls(poly *r, const uint8_t buf[KYBER_ETA1_hls*KYBER_N_hls/4])
873 {
874 #if KYBER_ETA1_hls == 2
875   cbd2_hls(r, buf);
876 #elif KYBER_ETA1_hls == 3
877   cbd3_hls(r, buf);
878 #else
```

```
879 #error "This implementation requires eta1 in {2,3}"
880 #endif
881 }
882
883 void poly_getnoise_eta1_hls(poly *r, const uint8_t seed[KYBER_SYMBYTES_hls],
884                             uint8_t nonce)
885 {
886   uint8_t buf[KYBER_ETA1_hls*KYBER_N_hls/4];
887   prf_hls(buf, sizeof(buf), seed, nonce);
888   poly_cbd_eta1_hls(r, buf);
889 }
890
891 void poly_cbd_eta2_hls(poly *r, const uint8_t buf[KYBER_ETA2_hls*KYBER_N_hls/4])
892 {
893 #if KYBER_ETA2_hls == 2
894   cbd2_hls(r, buf);
895 #else
896 #error "This implementation requires eta2 = 2"
897 #endif
898 }
899
900 void poly_getnoise_eta2_hls(poly *r, const uint8_t seed[KYBER_SYMBYTES_hls],
901                             uint8_t nonce)
902 {
903   uint8_t buf[KYBER_ETA2_hls*KYBER_N_hls/4];
904   prf_hls(buf, sizeof(buf), seed, nonce);
905   poly_cbd_eta2_hls(r, buf);
906 }
907
908 void polyvec_compress_hls(uint8_t r[KYBER_POLYVECCOMPRESSEDBYTES_hls],
909                           const polyvec *a)
910 {
911   unsigned int i,j,k;
912
913 #if (KYBER_POLYVECCOMPRESSEDBYTES_hls == (KYBER_K_hls * 352))
914   uint16_t t[8];
915   for(i=0;i<KYBER_K_hls;i++) {
916     for(j=0;j<KYBER_N_hls/8;j++) {
917       for(k=0;k<8;k++) {
918         t[k]  = a->vec[i].coeffs[8*j+k];
919         t[k] += ((int16_t)t[k] >> 15) & KYBER_Q_hls;
920         t[k]  = ((((uint32_t)t[k] << 11) + KYBER_Q_hls/2)/KYBER_Q_hls) & 0x7ff;
921       }
922
923       r[ 0] = (t[0] >>  0);
924       r[ 1] = (t[0] >>  8) | (t[1] << 3);
925       r[ 2] = (t[1] >>  5) | (t[2] << 6);
926       r[ 3] = (t[2] >>  2);
927       r[ 4] = (t[2] >> 10) | (t[3] << 1);
928       r[ 5] = (t[3] >>  7) | (t[4] << 4);
929       r[ 6] = (t[4] >>  4) | (t[5] << 7);
930       r[ 7] = (t[5] >>  1);
```

```
931         r[ 8] = (t[5] >>  9) | (t[6] << 2);
932         r[ 9] = (t[6] >>  6) | (t[7] << 5);
933         r[10] = (t[7] >>  3);
934         r += 11;
935       }
936     }
937   #elif (KYBER_POLYVECCOMPRESSEDBYTES_hls == (KYBER_K_hls * 320))
938     uint16_t t[4];
939     for(i=0;i<KYBER_K_hls;i++) {
940       for(j=0;j<KYBER_N_hls/4;j++) {
941         for(k=0;k<4;k++) {
942           t[k]  = a->vec[i].coeffs[4*j+k];
943           t[k] += ((int16_t)t[k] >> 15) & KYBER_Q_hls;
944           t[k]  = ((((uint32_t)t[k] << 10) + KYBER_Q_hls/2)/ KYBER_Q_hls) & 0x3ff;
945         }
946
947         r[0] = (t[0] >> 0);
948         r[1] = (t[0] >> 8) | (t[1] << 2);
949         r[2] = (t[1] >> 6) | (t[2] << 4);
950         r[3] = (t[2] >> 4) | (t[3] << 6);
951         r[4] = (t[3] >> 2);
952         r += 5;
953       }
954     }
955   #else
956   #error "KYBER_POLYVECCOMPRESSEDBYTES needs to be in {320*KYBER_K, 352*KYBER_K}"
957   #endif
958   }
959
960   void poly_compress_hls(uint8_t r[KYBER_POLYCOMPRESSEDBYTES_hls], const poly *a)
961   {
962     unsigned int i,j;
963     int16_t u;
964     uint8_t t[8];
965
966   #if (KYBER_POLYCOMPRESSEDBYTES_hls == 128)
967     for(i=0;i<KYBER_N_hls/8;i++) {
968       for(j=0;j<8;j++) {
969         // map to positive standard representatives
970         u  = a->coeffs[8*i+j];
971         u += (u >> 15) & KYBER_Q_hls;
972         t[j] = ((((uint16_t)u << 4) + KYBER_Q_hls/2)/KYBER_Q_hls) & 15;
973       }
974
975       r[0] = t[0] | (t[1] << 4);
976       r[1] = t[2] | (t[3] << 4);
977       r[2] = t[4] | (t[5] << 4);
978       r[3] = t[6] | (t[7] << 4);
979       r += 4;
980     }
981   #elif (KYBER_POLYCOMPRESSEDBYTES_hls == 160)
982     for(i=0;i<KYBER_N_hls/8;i++) {
```

```
983       for(j=0;j<8;j++) {
984         // map to positive standard representatives
985         u  = a->coeffs[8*i+j];
986         u += (u >> 15) & KYBER_Q_hls;
987         t[j] = ((((uint32_t)u << 5) + KYBER_Q_hls/2)/KYBER_Q_hls) & 31;
988       }
989
990       r[0] = (t[0] >> 0) | (t[1] << 5);
991       r[1] = (t[1] >> 3) | (t[2] << 2) | (t[3] << 7);
992       r[2] = (t[3] >> 1) | (t[4] << 4);
993       r[3] = (t[4] >> 4) | (t[5] << 1) | (t[6] << 6);
994       r[4] = (t[6] >> 2) | (t[7] << 3);
995       r += 5;
996     }
997   #else
998   #error "KYBER_POLYCOMPRESSEDBYTES needs to be in {128, 160}"
999   #endif
1000  }
1001
1002  static void pack_ciphertext_hls(uint8_t r[KYBER_INDCPA_BYTES_hls], polyvec *b,
1003                                  poly *v)
1004  {
1005    polyvec_compress_hls(r, b);
1006    poly_compress_hls(r+KYBER_POLYVECCOMPRESSEDBYTES_hls, v);
1007  }
```

I.2 VALORES DOS PARÂMETROS UTILIZADOS NO TESTE KYBER FPGA

cifrado_esperado = [171, 94, 218, 135, 17, 112, 96, 73, 27, 52, 171, 199, 255, 178, 40, 12, 150, 23, 158, 236, 141, 9, 197, 180, 66, 83, 18, 107, 125, 207, 123, 246, 138, 201, 239, 58, 12, 179, 176, 137, 22, 25, 53, 199, 12, 67, 101, 24, 57, 98, 135, 29, 245, 57, 97, 143, 215, 31, 112, 151, 45, 233, 84, 100, 144, 2, 41, 190, 225, 57, 197, 145, 24, 84, 187, 137, 32, 56, 155, 185, 76, 93, 223, 203, 42, 137, 228, 76, 82, 51, 255, 156, 36, 167, 81, 84, 174, 242, 117, 203, 83, 70, 89, 31, 7, 137, 184, 175, 35, 245, 24, 30, 134, 18, 105, 16, 122, 17, 34, 237, 167, 54, 165, 20, 193, 137, 122, 113, 221, 237, 197, 132, 218, 68, 18, 47, 94, 145, 53, 58, 245, 84, 0, 101, 1, 4, 79, 184, 64, 95, 121, 116, 207, 116, 132, 50, 232, 194, 220, 7, 4, 237, 89, 69, 131, 105, 170, 63, 105, 49, 132, 110, 255, 232, 169, 237, 12, 6, 11, 69, 112, 37, 45, 68, 13, 200, 78, 232, 130, 180, 122, 53, 76, 13, 20, 22, 79, 152, 15, 20, 139, 45, 251, 147, 212, 244, 237, 195, 73, 60, 39, 72, 152, 198, 115, 43, 175, 234, 117, 32, 131, 37, 21, 82, 164, 126, 159, 170, 130, 135, 82, 222, 205, 84, 164, 246, 79, 198, 246, 107, 134, 246, 75, 252, 173, 20, 128, 58, 119, 186, 180, 126, 187, 57, 157, 171, 235, 39, 130, 197, 146, 130, 49, 171, 18, 165, 32, 86, 156, 8, 105, 217, 184, 139, 120, 254, 219, 211, 59, 156, 205, 241, 186, 95, 240, 163, 175, 31, 228, 137, 84, 169, 15, 155, 169, 8, 94, 219, 99, 123, 74, 104, 61, 104, 193, 116, 171, 60, 169, 77, 84, 90, 42, 20, 0, 168, 0, 22, 21, 90, 101, 37, 127, 196, 129, 13, 11, 23, 78, 222, 79, 213, 101, 205, 53, 192, 50, 150, 176, 238, 78, 185, 216, 22, 111, 165, 101, 245, 149, 133, 218, 142, 154, 169, 118, 203, 113, 121, 103, 80, 170, 255, 242, 225, 179, 23, 138, 1, 84, 32, 47, 58, 156, 126, 28, 221, 196, 68, 126, 185, 12, 50, 151, 188, 172, 148, 54, 203, 232, 63, 222, 121, 215, 162, 49, 43,

8, 183, 84, 30, 54, 235, 194, 86, 156, 168, 44, 49, 164, 30, 244, 248, 17, 134, 80, 91, 73, 188, 68, 36, 28, 122, 29, 158, 125, 155, 196, 72, 11, 251, 211, 30, 148, 126, 60, 71, 212, 59, 62, 196, 102, 52, 104, 4, 249, 232, 117, 51, 76, 223, 100, 115, 70, 153, 26, 89, 38, 153, 132, 220, 112, 221, 235, 2, 198, 170, 232, 139, 148, 57, 160, 210, 23, 24, 51, 41, 164, 164, 243, 75, 247, 92, 7, 216, 189, 200, 113, 102, 199, 193, 141, 184, 69, 10, 212, 3, 129, 19, 99, 96, 7, 47, 136, 45, 154, 180, 181, 84, 178, 111, 71, 253, 66, 58, 49, 24, 8, 47, 155, 55, 124, 204, 199, 122, 195, 179, 28, 10, 161, 245, 223, 80, 27, 142, 124, 211, 47, 237, 109, 248, 150, 67, 37, 193, 143, 202, 32, 100, 14, 197, 3, 196, 68, 68, 69, 136, 72, 158, 128, 83, 53, 93, 141, 155, 35, 21, 97, 160, 136, 45, 89, 155, 100, 192, 138, 144, 120, 44, 136, 53, 61, 210, 245, 13, 17, 48, 233, 232, 204, 147, 77, 169, 47, 27, 110, 150, 100, 124, 139, 83, 7, 136, 252, 237, 160, 166, 233, 199, 127, 182, 174, 145, 219, 126, 6, 228, 83, 208, 190, 75, 181, 65, 88, 42, 56, 36, 244, 94, 80, 6, 223, 227, 103, 157, 201, 41, 59, 88, 203, 141, 73, 201, 89, 85, 27, 137, 84, 20, 57, 216, 148, 245, 183, 48, 107, 135, 166, 197, 107, 213, 63, 18, 166, 47, 152, 214, 111, 184, 129, 219, 223, 28, 234, 114, 97, 48, 194, 229, 141, 27, 158, 39, 52, 220, 171, 248, 136, 2, 255, 68, 38, 239, 242, 48, 101, 32, 64, 56, 164, 197, 181, 224, 32, 131, 111, 142, 47, 194, 143, 238, 125, 58, 239, 91, 161, 74, 124, 6, 12, 16, 195, 242, 196, 137, 246, 48, 162, 182, 55, 13, 214, 89, 153, 246, 130, 80, 228, 57, 121, 208, 215, 179, 134, 110, 141, 68, 234, 81, 144, 170, 182, 174, 220, 16, 195, 165, 32, 103, 27, 251, 233, 156, 125, 21, 68, 226, 4, 226, 116, 186, 34, 89, 244, 107, 66, 18, 149, 153, 81, 241, 51, 226, 201, 230, 175, 119, 237, 53, 40, 97, 109, 251, 227, 85, 42, 190, 127, 94, 56, 137, 59, 184, 240, 247, 66, 212, 253, 64, 105, 12, 31, 7, 207, 53, 121, 12, 219, 135, 208, 165, 47, 67, 195, 228, 197, 219, 111, 227, 3, 99, 201, 95, 13, 178, 21, 16, 120, 106, 202, 202, 226, 215, 197, 14, 215, 186, 19, 46, 46, 218, 88, 6, 50, 252, 189, 195, 152, 84, 181, 124, 220, 79, 76, 21, 117, 153, 172, 219, 73, 47, 162, 34, 145, 139, 211, 48, 230, 134, 225, 239, 223, 228, 143, 79, 52, 0, 170, 50, 233, 192, 95, 147, 118, 96, 200, 71, 43, 85, 145, 35, 248, 245, 226, 84, 168, 22, 228, 214, 139, 250, 13, 135, 2, 166, 244, 105, 149, 97, 148, 144, 5, 227, 87, 171, 115, 243, 151, 240, 19, 138, 191, 110, 159, 106, 57, 146, 11, 57, 40, 96, 19, 43, 23, 217, 40, 102, 93, 100, 189, 245, 206, 27, 128, 96, 68, 145, 232, 61, 177, 59, 216, 39, 250, 46, 59, 154, 103, 195, 91, 99, 24, 168, 97, 2, 176, 28, 182, 247, 219, 29, 138, 161, 9, 164, 245, 96, 3, 153, 147, 151, 184, 132, 69, 105, 162, 228, 111, 150, 10, 240, 240, 227, 177, 24, 64, 233, 215, 77, 225, 80, 186, 185, 253, 57, 147, 9, 13, 8, 119, 180, 152, 232, 55, 255, 252, 169, 5, 213, 151, 99, 191, 235, 31, 245, 58, 254, 197, 31, 207, 31, 201, 87, 27, 209, 19, 25, 134, 190, 58, 24, 132, 132, 178, 182, 219, 28, 142, 160, 143, 22, 101, 106, 151, 215, 177, 190, 226, 221, 151, 159, 39, 105, 190, 80, 132, 78, 115, 192, 211, 133, 124, 12, 250, 28, 186, 183, 48, 18]

coins_tmp = [229, 84, 147, 131, 207, 100, 8, 179, 43, 145, 245, 128, 107, 189, 226, 9, 239, 80, 173, 88, 106, 217, 75, 178, 235, 62, 146, 41, 42, 93, 53, 98]

m_tmp = [24, 106, 175, 191, 49, 237, 164, 80, 124, 95, 214, 221, 235, 92, 183, 174, 225, 223, 86, 207, 78, 124, 181, 14, 218, 184, 101, 15, 77, 195, 100, 27]

pk_tmp = [126, 185, 207, 239, 21, 142, 52, 243, 19, 131, 247, 15, 223, 154, 152, 206, 201, 89, 108, 139, 138, 22, 171, 134, 206, 134, 89, 39, 81, 83, 16, 115, 174, 157, 3, 97, 209, 72, 54, 55, 33, 133, 101, 163, 3, 59, 194, 13, 232, 121, 23, 17, 198, 152, 19, 58, 152, 180, 6, 39, 170, 167, 91, 219, 9, 159, 252, 235, 122, 145, 129, 135, 32, 166, 27, 105, 245, 61, 169, 245, 95, 64, 178, 58, 208, 52, 95, 225, 130, 193, 128, 28, 205, 209, 214, 83, 85, 106, 118, 202, 121, 28, 107, 54, 38, 13, 112, 100, 157, 230, 70, 7, 58, 159, 9, 195, 158, 63, 67, 134, 4, 137, 153, 27, 183, 24, 134, 160, 112, 162, 9, 153, 145, 148, 199, 150, 69, 73, 17, 60, 59, 216, 179, 41, 110, 195, 202, 197, 38, 41, 58, 149, 177, 118, 161, 3, 21, 58, 85, 29, 236, 162, 37,

149, 171, 194, 229, 202, 42, 9, 98, 239, 50, 84, 123, 241, 177, 207, 168, 43, 79, 28, 26, 219, 25, 10, 226, 137, 60, 158, 224, 197, 41, 48, 112, 42, 7, 32, 168, 105, 72, 29, 169, 25, 114, 231, 106, 135, 0, 49, 88, 64, 143, 121, 70, 176, 17, 176, 1, 43, 217, 40, 254, 138, 1, 2, 70, 191, 254, 228, 56, 163, 236, 193, 78, 66, 71, 73, 181, 70, 222, 91, 128, 119, 51, 22, 205, 246, 108, 27, 44, 161, 205, 54, 167, 18, 96, 33, 118, 36, 118, 111, 102, 21, 152, 198, 24, 90, 131, 85, 28, 197, 108, 60, 225, 194, 73, 42, 99, 91, 134, 139, 191, 69, 26, 115, 140, 83, 31, 244, 154, 218, 182, 106, 133, 69, 183, 24, 41, 204, 187, 250, 95, 18, 84, 117, 130, 55, 71, 132, 180, 147, 114, 21, 101, 229, 88, 86, 203, 214, 45, 157, 129, 16, 111, 84, 58, 16, 82, 77, 42, 123, 181, 22, 185, 39, 38, 9, 77, 153, 119, 170, 109, 139, 35, 91, 86, 16, 175, 220, 182, 71, 187, 102, 49, 83, 87, 170, 103, 70, 210, 0, 196, 23, 1, 110, 72, 40, 190, 100, 35, 168, 39, 249, 141, 222, 148, 12, 143, 182, 149, 156, 49, 160, 49, 199, 187, 225, 138, 185, 75, 8, 23, 209, 107, 88, 112, 22, 26, 74, 245, 155, 110, 180, 15, 40, 64, 62, 173, 129, 36, 19, 76, 145, 106, 17, 160, 139, 118, 73, 70, 59, 114, 68, 132, 33, 168, 98, 6, 42, 139, 170, 173, 122, 200, 28, 96, 202, 231, 130, 24, 10, 185, 202, 243, 9, 21, 30, 251, 32, 175, 241, 142, 49, 163, 50, 197, 196, 166, 175, 119, 24, 139, 21, 170, 15, 155, 186, 4, 161, 91, 158, 242, 19, 6, 10, 133, 57, 37, 195, 240, 26, 136, 31, 133, 125, 155, 160, 7, 131, 168, 0, 72, 241, 118, 184, 204, 185, 201, 208, 89, 198, 55, 33, 252, 58, 40, 79, 201, 94, 217, 16, 179, 171, 41, 7, 255, 34, 158, 110, 131, 91, 106, 177, 108, 135, 112, 30, 163, 8, 85, 107, 28, 76, 58, 64, 203, 144, 18, 34, 41, 240, 73, 55, 248, 117, 244, 148, 152, 145, 243, 155, 214, 28, 12, 114, 32, 26, 54, 148, 93, 60, 146, 140, 155, 182, 74, 62, 0, 141, 73, 96, 22, 39, 85, 10, 152, 217, 89, 30, 140, 101, 202, 16, 171, 9, 7, 10, 175, 186, 192, 235, 55, 104, 108, 203, 199, 28, 27, 165, 134, 229, 207, 250, 252, 86, 20, 55, 118, 23, 28, 70, 249, 130, 170, 151, 180, 41, 47, 199, 205, 5, 198, 136, 133, 194, 94, 242, 89, 68, 89, 233, 138, 88, 65, 61, 115, 10, 166, 87, 84, 28, 19, 27, 114, 99, 208, 206, 84, 83, 57, 44, 211, 131, 202, 183, 5, 74, 101, 92, 7, 8, 81, 132, 48, 125, 19, 150, 35, 75, 91, 119, 143, 104, 123, 60, 105, 84, 33, 101, 161, 228, 148, 74, 63, 75, 119, 116, 199, 141, 181, 200, 145, 169, 166, 152, 245, 108, 24, 233, 20, 136, 182, 168, 11, 19, 247, 12, 123, 5, 202, 120, 55, 67, 206, 186, 73, 252, 98, 69, 130, 214, 179, 28, 247, 99, 54, 172, 199, 216, 19, 78, 159, 132, 139, 245, 228, 112, 75, 218, 186, 46, 37, 137, 110, 135, 128, 84, 209, 19, 239, 161, 202, 116, 90, 92, 0, 56, 54, 91, 203, 166, 3, 57, 165, 35, 7, 92, 106, 241, 40, 191, 235, 168, 245, 9, 105, 249, 230, 68, 5, 136, 40, 5, 124, 50, 251, 42, 207, 76, 83, 25, 135, 7, 14, 221, 128, 75, 146, 4, 125, 119, 48, 206, 83, 181, 36, 93, 44, 207, 111, 133, 190, 171, 156, 105, 176, 166, 12, 190, 134, 108, 28, 65, 174, 204, 211, 202, 2, 84, 81, 23, 242, 49, 88, 163, 45, 170, 226, 163, 153, 11, 3, 19, 155, 205, 154, 241, 89, 28, 250, 23, 22, 7, 149, 40, 36, 114, 205, 56, 77, 89, 122, 64, 96, 216, 52, 61, 197, 200, 200, 183, 137, 190, 212, 103, 196, 74, 72, 57, 87, 196, 158, 230, 10, 13, 181, 41, 39, 49, 113, 176, 165, 99, 31, 218, 132, 228, 150, 19, 113, 121, 117, 147, 202, 26, 31, 228, 66, 89, 39, 139, 146, 40, 12, 183, 80, 147, 113, 231, 198, 17, 97, 150, 82, 196, 84, 255, 193, 103, 126, 147, 191, 230, 155, 121, 143, 121, 97, 155, 35, 178, 205, 16, 112, 154, 38, 15, 10, 152, 177, 154, 244, 201, 154, 172, 119, 31, 114, 55, 43, 162, 34, 32, 147, 5, 2, 71, 10, 113, 170, 192, 2, 122, 6, 124, 195, 86, 246, 40, 73, 187, 217, 147, 191, 114, 174, 225, 163, 17, 162, 64, 4, 225, 124, 28, 246, 200, 120, 202, 11, 96, 1, 104, 133, 119, 197, 42, 102, 163, 92, 87, 50, 41, 162, 72, 62, 151, 227, 127, 235, 23, 167, 116, 113, 197, 172, 113, 157, 136, 17, 192, 13, 251, 40, 23, 203, 67, 172, 181, 45, 124, 185, 38, 47, 123, 14, 218, 80, 147, 227, 6, 179, 137, 241, 150, 58, 114, 3, 192, 5, 36, 246, 233, 94, 209, 186, 149, 103, 121, 163, 81, 74, 58, 112, 179, 88, 9, 39, 96, 164, 146, 163, 186, 244, 154, 124, 69, 122, 166, 234, 199, 47, 225, 196, 74, 196, 2, 122, 20, 190, 62, 103, 160, 152, 38, 0, 18, 128, 68, 158, 54, 77, 167, 188, 16, 233, 203, 31, 210, 2, 69, 254, 59, 15, 63, 180, 53, 247, 181, 158, 157, 161, 156, 170, 98, 132, 59, 146, 206, 96, 131, 156, 244, 87, 62, 76, 166, 57, 164, 72, 95, 214, 19, 120, 142, 40, 155, 221, 102, 17, 167, 201, 32, 60, 198, 103, 247, 74, 26, 168, 52, 225, 114,

12, 108, 104, 182, 217, 192, 227, 76, 211, 64]

sk_tmp = [99, 172, 3, 219, 240, 29, 130, 36, 31, 195, 211, 39, 78, 164, 26, 151, 211, 153, 151, 48, 158, 19, 180, 120, 224, 40, 94, 170, 42, 47, 61, 34, 93, 247, 212, 135, 164, 247, 138, 87, 36, 162, 50, 214, 107, 213, 248, 71, 248, 89, 154, 206, 85, 30, 79, 115, 74, 250, 100, 141, 129, 88, 94, 166, 186, 81, 215, 216, 35, 62, 69, 45, 5, 185, 17, 202, 229, 191, 66, 66, 53, 44, 187, 175, 209, 132, 32, 42, 179, 102, 106, 56, 96, 239, 201, 167, 70, 225, 25, 18, 51, 144, 183, 17, 165, 3, 104, 19, 70, 250, 106, 219, 213, 161, 168, 227, 122, 15, 181, 10, 208, 137, 149, 32, 233, 52, 23, 57, 57, 224, 184, 0, 31, 104, 66, 170, 66, 45, 174, 182, 180, 72, 8, 14, 88, 19, 98, 87, 114, 161, 58, 1, 42, 234, 23, 35, 33, 20, 165, 1, 74, 14, 252, 55, 194, 182, 194, 105, 0, 241, 51, 174, 67, 155, 41, 23, 155, 158, 236, 130, 200, 58, 126, 235, 219, 103, 157, 123, 139, 22, 48, 68, 246, 246, 100, 197, 10, 53, 146, 232, 110, 239, 50, 121, 32, 53, 120, 90, 121, 138, 212, 76, 200, 24, 103, 67, 130, 172, 154, 84, 4, 15, 146, 68, 17, 15, 217, 102, 14, 36, 78, 145, 225, 178, 243, 37, 20, 127, 149, 16, 71, 87, 52, 223, 67, 124, 237, 96, 183, 130, 96, 76, 198, 129, 144, 44, 49, 118, 208, 82, 120, 221, 213, 156, 11, 168, 66, 184, 131, 63, 75, 208, 199, 185, 76, 84, 224, 217, 165, 191, 250, 145, 33, 182, 7, 179, 8, 1, 222, 87, 137, 139, 34, 161, 153, 166, 17, 48, 118, 180, 158, 43, 104, 202, 39, 119, 7, 218, 18, 161, 52, 184, 100, 82, 68, 134, 1, 118, 28, 121, 195, 46, 22, 195, 26, 155, 92, 101, 83, 156, 48, 124, 72, 34, 219, 95, 35, 42, 189, 38, 230, 84, 107, 102, 188, 46, 245, 48, 63, 92, 199, 148, 24, 12, 137, 148, 130, 52, 231, 50, 177, 12, 40, 97, 247, 72, 201, 181, 191, 7, 230, 200, 204, 188, 26, 172, 240, 40, 87, 153, 83, 254, 22, 20, 180, 116, 185, 226, 249, 6, 44, 192, 41, 65, 233, 79, 135, 208, 121, 6, 129, 65, 164, 67, 11, 32, 232, 188, 251, 242, 203, 57, 44, 117, 201, 164, 62, 217, 4, 118, 189, 20, 48, 251, 136, 82, 79, 136, 164, 87, 26, 6, 20, 135, 111, 204, 43, 11, 33, 52, 191, 170, 36, 5, 120, 248, 112, 202, 48, 132, 193, 171, 32, 0, 116, 117, 91, 3, 141, 23, 180, 194, 64, 87, 29, 223, 2, 8, 191, 202, 66, 158, 178, 82, 251, 16, 167, 151, 0, 89, 165, 37, 205, 81, 19, 50, 142, 53, 138, 85, 224, 190, 33, 192, 45, 167, 40, 14, 131, 122, 178, 33, 216, 98, 213, 170, 46, 138, 248, 132, 143, 171, 140, 30, 249, 152, 157, 26, 76, 43, 10, 8, 111, 172, 5, 158, 39, 205, 204, 146, 130, 58, 178, 3, 97, 252, 26, 57, 144, 17, 56, 74, 38, 114, 42, 81, 156, 117, 156, 164, 204, 158, 163, 106, 61, 90, 129, 140, 241, 32, 88, 110, 129, 26, 150, 53, 200, 231, 140, 75, 43, 234, 111, 212, 67, 38, 234, 186, 96, 102, 101, 81, 222, 24, 177, 106, 228, 112, 63, 85, 126, 36, 193, 117, 208, 162, 40, 195, 86, 137, 138, 161, 111, 230, 161, 168, 106, 4, 153, 115, 224, 89, 125, 199, 170, 96, 6, 178, 224, 150, 102, 51, 57, 119, 149, 149, 76, 69, 39, 118, 3, 228, 160, 200, 32, 190, 202, 129, 206, 168, 100, 168, 124, 32, 117, 145, 226, 96, 58, 104, 194, 220, 193, 37, 10, 72, 73, 216, 51, 94, 81, 148, 190, 150, 121, 188, 222, 131, 113, 80, 146, 196, 205, 236, 202, 233, 81, 69, 195, 198, 1, 98, 104, 197, 158, 232, 94, 179, 131, 80, 4, 165, 109, 126, 66, 168, 72, 117, 86, 218, 234, 6, 213, 188, 31, 34, 40, 65, 80, 89, 97, 210, 65, 87, 119, 55, 207, 201, 183, 59, 112, 76, 150, 145, 167, 166, 133, 182, 75, 195, 50, 0, 186, 165, 132, 58, 26, 66, 57, 147, 16, 109, 219, 142, 26, 60, 102, 51, 133, 7, 137, 116, 108, 149, 235, 45, 68, 164, 194, 85, 134, 164, 141, 40, 24, 102, 56, 119, 139, 229, 27, 140, 197, 79, 104, 52, 31, 138, 211, 140, 28, 7, 100, 161, 163, 105, 205, 112, 31, 221, 56, 123, 58, 123, 72, 33, 27, 32, 46, 54, 113, 215, 152, 87, 76, 251, 44, 153, 145, 132, 85, 194, 118, 131, 184, 51, 209, 213, 161, 45, 208, 177, 235, 233, 124, 96, 134, 77, 227, 5, 27, 222, 184, 86, 231, 139, 51, 0, 25, 142, 125, 219, 86, 82, 39, 136, 240, 148, 85, 177, 113, 199, 90, 243, 85, 135, 151, 111, 24, 146, 103, 9, 171, 125, 132, 210, 58, 211, 115, 152, 21, 182, 131, 203, 48, 57, 226, 106, 131, 219, 196, 126, 143, 99, 59, 20, 180, 78, 117, 164, 117, 128, 199, 119, 97, 150, 112, 244, 218, 108, 198, 34, 144, 230, 91, 54, 25, 171, 33, 93, 138, 107, 251, 251, 160, 191, 26, 48, 162, 56, 176, 93, 26, 143, 136, 81, 154, 161, 241, 159, 27, 107, 9, 82, 208, 170, 240, 167, 131, 42, 203, 126, 185, 98, 187, 180, 124, 111, 32, 163, 65, 68, 229, 85, 51, 228, 86, 128, 18, 10, 214, 98, 105, 233, 172, 206, 100, 200, 141, 96, 100, 6, 159, 51, 31, 41, 51,

90, 59, 26, 56, 140, 43, 49, 36, 49, 181, 98, 107, 148, 192, 228, 36, 161, 3, 190, 129, 138, 65, 31, 23, 184, 226, 129, 115, 160, 124, 156, 26, 117, 125, 107, 226, 178, 41, 16, 102, 116, 87, 172, 96, 215, 163, 144, 152, 200, 245, 227, 45, 177, 42, 97, 226, 169, 108, 119, 178, 68, 235, 9, 95, 155, 67, 135, 126, 156, 185, 40, 101, 197, 23, 82, 0, 129, 182, 9, 113, 202, 15, 253, 55, 76, 133, 226, 44, 1, 252, 173, 140, 208, 137, 36, 51, 181, 137, 96, 7, 3, 227, 12, 50, 19, 31, 196, 73, 18, 51, 155, 123, 149, 74, 105, 117, 12, 78, 216, 152, 39, 184, 225, 46, 196, 19, 64, 193, 58, 43, 96, 202, 37, 241, 229, 169, 162, 90, 137, 212, 198, 45, 249, 196, 57, 230, 132, 17, 204, 219, 171, 149, 156, 160, 198, 149, 201, 196, 244, 167, 0, 215, 52, 201, 101, 56, 175, 186, 15, 227, 151, 124, 142, 203, 131]