Uma Abordagem Hardware/Software para Implementação de Criptografia Baseada em Identidades

Este exemplar corresponde à redação final da dissertação corrigida e defendida por Leonardo Scanferla Amaral e aprovada pela Banca Examinadora.

Campinas, 9 de dezembro de 2009.

Prof. Dr. Guido Costa Souza de Araújo
(Orientador)

Prof. Dr. Julio Cesar López Hernández
(Co-orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Amaral, Leonardo Scanferla
Am13a Uma abordagem hardware/software para implementação de criptografia baseada em identidades/Leonardo Scanferla Amaral--Campinas, [S.P. : s.n.], 2009.

Orientador : Guido Costa Souza de Araújo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Arquitetura de computador. 2.Criptografia de chave-pública. I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Hardware/software co-design approach for identity-based encryption

Palavras-chave em inglês (Keywords): 1.Computer architecture. 2.Public key cryptography.

Área de concentração: Arquitetura e Sistemas de Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Guido Costa Souza de Araújo (IC-UNICAMP)
Profa. Dra. Nahri Moreano (DCT-UFMS)
Prof. Dr. Ricardo Dahab (IC-UNICAMP)

Data da defesa: 09/12/2009

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 09 de dezembro de 2009, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Nahri Balesdent Moreano
DCT / UFMS

Prof. Dr. Ricardo Dahab
IC / UNICAMP

Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Uma Abordagem Hardware/Software para Implementação de Criptografia Baseada em Identidades

Leonardo Scanferla Amaral[1]

Dezembro de 2009

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof.[a] Dr.[a] Nahri Moreano (DCT-UFMS)
- Prof. Dr. Ricardo Dahab (IC-UNICAMP)
- Prof. Dr. Paulo Cesar Centoducatte (IC-UNICAMP) (Suplente)
- Prof. Dr. Ricardo Santos (DCT-UFMS) (Suplente)

[1]Trabalho financiado pela FAPESP (processo 2007/02087-5)

Resumo

A Criptografia Baseada em Identidades vem sendo cada vez mais aceita como uma alternativa à criptografia assimétrica em alguns cenários. O cálculo eficiente de emparelhamentos sobre curvas elípticas é imprescindível para o desempenho dos sistemas de Criptografia Baseada em Identidades. Nesse trabalho apresentaremos um estudo comparativo entre diferentes técnicas de implementação em hardware da aritmética em F_3^m para o cálculo do emparelhamento η_T, em uma plataforma de baixo custo. Nossa implementação hardware/software utiliza o processador Nios II da Altera como plataforma. Através de um mapeamento da execução do emparelhamento foram identificadas as operações aritméticas que consomem mais tempo durante o cálculo do emparelhamento; essas operações foram então implementadas como instruções/módulos especializadas em FPGA e adicionadas ao processador. Em seguida, o processador especializado foi sintetizado em FPGA e o software da aplicação de emparelhamento foi modificado para utilizar as novas instruções especializadas para o emparelhamento. Experimentos comprovam que um ganho considerável de desempenho é alcançado quando essa abordagem é comparada à abordagem de software inicial. Além disso, vamos mostrar que a abordagem Hardware/Software se mostra competitiva com relação a outras soluções.

Abstract

Identity-Based Cryptography has been gradually accepted as an effective way of implementing asymmetric cryptography. The calculation of cryptographically-suitable pairings is crucial for the performance of pairing based protocols. In this work we present a comparative study of hardware implementation techniques for computing the η_T pairing over the finite field F_{3^m} using a low-cost platform based on then Altera Nios II processor. Using code profiling we identify critical field operations which concentrate most of the execution time; these operations were implemented as specialized FPGA instructions/modules and added to the processor. The specialized processor was synthesized and the application was tailored to the new hardware. Experimental results show that a considerable speedup can be achieved when compared to the baseline software-only approach. Moreover, we show that such Hardware/Software co-design approach is competitive with other solutions.

Epígrafe

There are only two ways to live your life.
One is as though nothing is a miracle.
The other is as though everything is a miracle.

Albert Einstein (1879-1955)

Dedicatória

Dedico esse trabalho primeiramente aos meus pais Nelson e Lourdes; pelo carinho, dedicação e incentivo, em todos os momentos da minha vida. À minha namorada Bianca, por todo o seu carinho, compreensão e incentivos, mesmo nos momentos em que não estive presente. Aos meus familiares e amigos, sem os quais a vida não teria graça. E por fim à Deus, pois sem ele, nada seria possível.

Agradecimentos

Meus profundos agradecimentos aos meus pais, por me apoiarem em todos os momentos da minha vida, por todo o carinho, amor, dedicação, e pela infinita compreensão. Agradeço ainda pela confiança depositada em mim, por todos os ensinamentos transmitidos em todas as fases da minha vida, por sempre me apoiarem na busca de meus sonhos e por me presentearem com o maior presente que se pode receber, a vida. Obrigado.

Meus profundos agradecimentos à minha amada namorada Bianca, por todo amor, confiança, carinho, e apoio dedicados a mim. Agradeço também pela paciência e compreensão que apresentou em todos os momentos em que eu não pude estar presente e peço desculpas pelas longas esperas pelas quais passou. Gostaria de dizer que sou um homem de sorte por ter te encontrado.

Agradeço a minha irmã Melissa, por toda motivação e ajuda que me prestou durante todo o tempo. Especialmente pelos momentos de distração e risadas que você sempre me proporcionou.

Agradeço sinceramente a minha tia Sonilda, ao meu tio Marcio e aos meus primos Raphael e Gabriel por todo carinho, atenção e apoio que me deram durante todos os momentos de dificuldades.

Agradeço também a minha avó Dirce e ao meu tio Zé por todos os ensinamentos transmitidos, pelo carinho inesgotável e por todos os momentos que tivemos juntos.

Minha eterna gratidão ao professor Guido Araujo por toda atenção despendida durante a orientação do meu mestrado, assim como por todos os conselhos, incentivos e por sempre acreditar em mim. Obrigado.

Especial agradecimento ao professor Julio López pela excepcional co-orientação em meu mestrado, por todas as longas conversas que tivemos e por me mostrar um mundo novo de conhecimentos.

Agradeço aos meus amigos de Bilac por todos os momentos de descontração e alegria que sempre me proporcionaram. A todos os encontros da turma nos sábados à tarde e por todo incentivo despendido.

Meus sinceros agradecimentos a todos os amigos da republica dos Malditos por me receberem de braços abertos, pelo convívio harmônico e pela amizade fomentada durante todo o tempo da realização deste trabalho.

Agradeço também aos amigos do laboratório LSC da Unicamp por todas as conversas e discussões inspiradoras e pelos inumeros cafés que tomamos juntos.

Agradeço a fundação FAPESP pela ajuda financeira fornecida durante a realização deste trabalho e por tornar esse sonho possível.

E por fim, mas não menos importante, manifesto minha infinita gratidão à Deus por me proporcionar a oportunidade de desenvolver esse trabalho em conjunto com profissionais brilhantes e por me conceder forças para realizá-lo.

Abreviações e Notações Utilizadas

$\wedge$: Operação & lógico (*AND*)

$\vee$: Operação OU lógico (*OR*)

$\neg$: Operação Não lógico (*NOT*)

$\oplus$: Operação OU-Exclusivo (*XOR*)

HW/SW : Hardware/Software

FPGA : Do inglês *Field Programmable Gate Array*

IDE : Do inglês *Integrated Development Environment*

MMU : Do inglês *Memory Management Unit*

MPU : Do inglês *Memory Protection Unit*

ISA : Do inglês *Instruction Set Architecture*

DMIPS : Do inglês *Dhrystone Million Instructions Per Second*

RISC : Do inglês *Reduced Instruction Set Computer*

GPL : Do inglês *General Public License*

ALU : Do inglês *Arithmetic Logic Unit*

TLB : Do inglês *Translation Lookaside Buffer*

IRQ : Do inglês *Interrupt Request*

ROM : Do inglês *Read Only Memory*

LCD : Do inglês *Liquid Crystal Display*

ALUT : Do inglês *Adaptive Look-up Table*

IBE : Do inglês *Identity-Base Encryption*

PKI : Do inglês *Public Key Infrastructure*

PKG : Do inglês *Public Key Generator*

NONCE : Do inglês *Number used ONCE*

MSE : Do inglês *Most-Significant Element*

VHDL : Do inglês *Very High Speed Integrated Circuits Hardware Description Language*

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

O sigilo nas comunicações é uma necessidade antiga, que existe desde os primórdios da humanidade. Atualmente, com a expansão da tecnologia da informação e dos meios de comunicação, é corriqueiro realizar atividades de caráter sigiloso (ou privado) através de equipamentos eletrônicos interconectados, no entanto, nem sempre esses equipamentos oferecem a segurança necessária à realização dessas atividades. Isto resulta em uma preocupação constante com relação à segurança no transporte e armazenamento de informações no contexto computacional. Nesse cenário a criptografia se apresenta como solução para suprir essa necessidade de segurança.

A comunicação no meio computacional costuma ocorrer através de equipamentos desconhecidos interconectados por meio de canais de comunicações inseguros. Um exemplo de tal configuração é a Internet. Sob tal contexto é essencial que as informações enviadas entre as estações comunicantes sejam manipuladas de tal forma a torná-las irreconhecíveis a qualquer indivíduo que não seja o verdadeiro destinatário da informação, assegurando assim o sigilo e a confiabilidade da informação. O ato de realizar essa manipulação da informação é chamado de encriptação/decriptação de dados e o conjunto das diferentes técnicas de encriptação/decriptação de dados define o termo criptografia.

A criptografia se subdivide em duas categorias: sistemas criptográficos simétricos e sistemas criptográficos assimétricos. Nos sistemas criptográficos simétricos um par de comunicantes contam com uma chave secreta pré-estabelecida, comum aos comunicantes. Essa chave é utilizada no processo de encriptação dos dados da comunicação, gerando assim um conjunto de dados irreconhecíveis, o qual retorna a seu formato inicial quando é decifrado utilizando a mesma chave utilizada na encriptação. Esse procedimento está ilustrado na Figura 1.1.

Já o sistema criptográfico assimétrico (também conhecido como sistema criptográfico de chave pública) utiliza um par de chaves matematicamente relacionadas para encriptar/decriptar as mensagens: chave privada e chave pública. A chave pública é distribuída livremente para todos os correspondentes, enquanto a chave privada deve ser conhecida apenas pelo seu dono,

conceito este que foi introduzido por Whitfield Diffie e Martin Hellman em 1976 [1]. Esse procedimento está ilustrado na Figura 1.2.

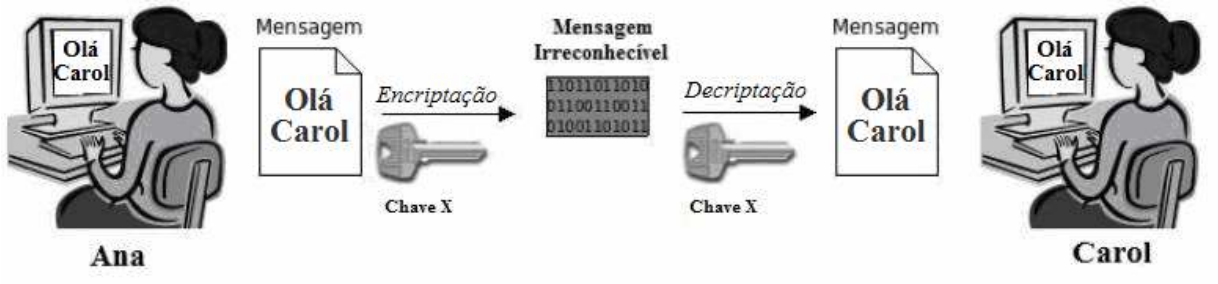

Figura 1.1: Enviando uma mensagem na Criptografia Simétrica

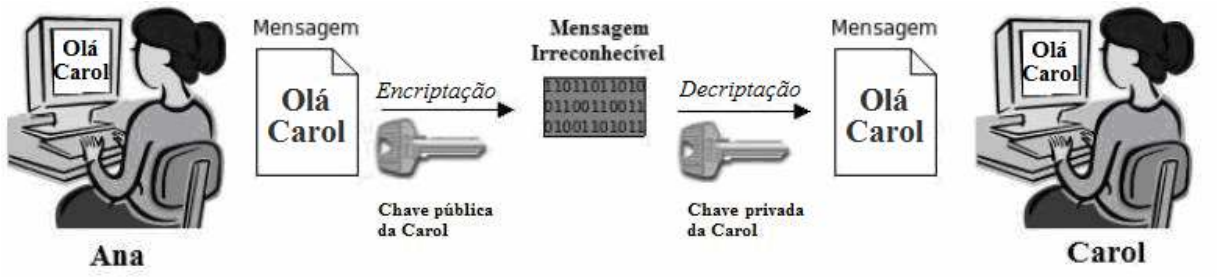

Figura 1.2: Enviando uma mensagem na Criptografia Assimétrica

Em um algoritmo criptográfico de chave pública, uma mensagem criptografada com a chave pública pode somente ser decriptografada pela sua chave privada correspondente. A segurança desse sistema se deve ao fato de ser computacionalmente inviável deduzir a chave privada a partir da chave pública.

A grande vantagem da criptografia assimétrica em relação à criptografia simétrica é não necessitar que as partes envolvidas na comunicação compartilhem previamente uma chave secreta, contudo, a criptografia simétrica proporciona maior velocidade na encriptação/decriptação de dados quando comparada a criptografia assimétrica (a criptográfica assimétrica necessita de grandes quantidades de cálculos matemáticos para ser realizada). Devido a esses fatos, é comum o emprego de ambos os métodos durante uma comunicação de dados. Inicialmente a criptografia assimétrica é utilizada para se estabelecer uma chave comum entre os comunicantes e após isso é utilizada a criptografia simétrica na encriptação/decriptação dos dados.

Outro problema relacionado aos sistemas criptográficos assimétricos é o fato de ele necessitar de uma complexa infra-estrutura de gerenciamento de chaves para que o detentor de uma chave pública possa provar sua identidade por meio de certificados digitais e autoridades certificadoras. Caso a identidade dos comunicantes não seja validada corretamente, um adversário pode se passar por um dos comunicantes e obter acesso ao dados contidos na

comunicação. Esse procedimento está ilustrado na Figura 1.3, nesse exemplo o adversário se faz passar pela comunicante Carol, interceptando a mensagem enviada de Ana para Carol (ataque passivo do intermediário).

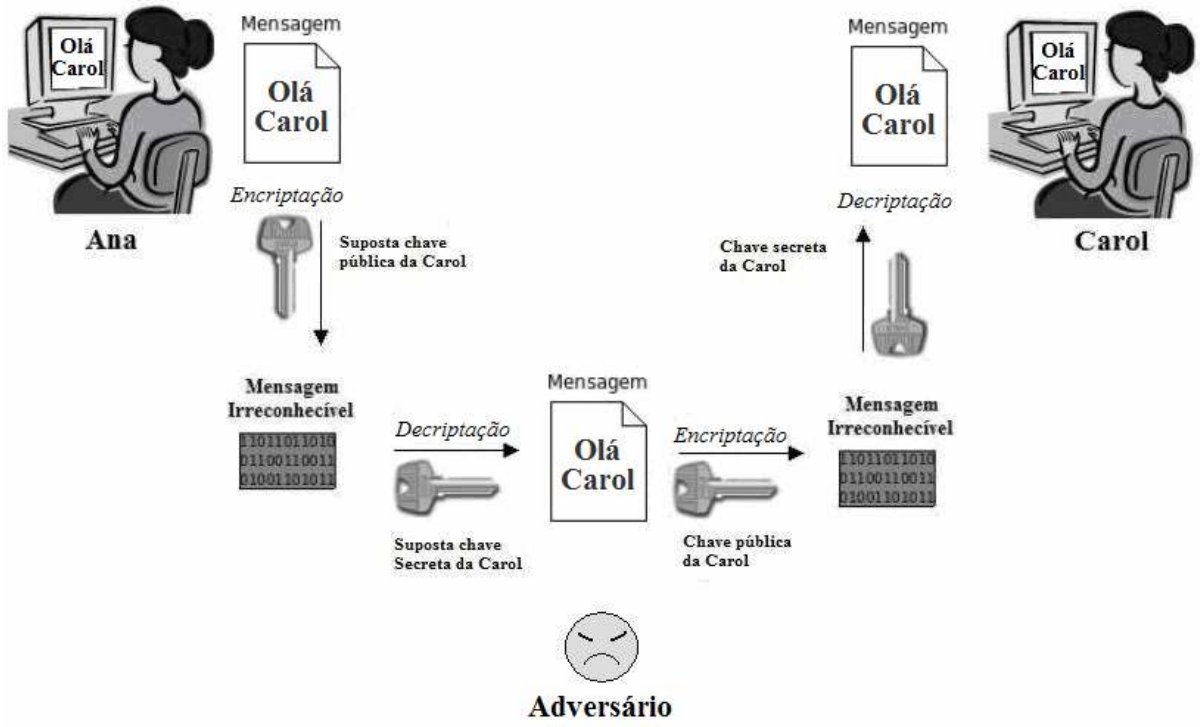

Figura 1.3: Ataque do Intermediário

Em 2001, o emprego de emparelhamento de curvas elípticas em criptografia permitiu o desenvolvimento de um novo esquema de criptografia assimétrica, esquema esse que não sofre das mesmas restrições encontradas na criptografia de chave pública tradicional. Esse esquema de criptografia assimétrica é conhecido como Criptografia Baseada em Identidades (*Identity-Base Encryption* - IBE) e foi proposto inicialmente por Shamir em 1984 [2]. Nesse novo esquema a chave pública de um usuário não é uma seqüência aleatória de bits, como no esquema anterior, e sim um identificador que caracteriza esse usuário de forma única, como por exemplo, seu número de RG ou seu endereço eletrônico (*e-mail*). Sendo assim, não é necessária a implementação da complexa infra-estrutura de autenticação presente na criptografia de chave pública. Os conceitos e características da Criptografia Baseada em Identidades serão apresentados em detalhes no capítulo 3.

1.1 Trabalhos Relacionados

Como será apresentado posteriormente, a operação mais custosa envolvida na realização da IBE é o emparelhamento sobre curvas elípticas. Várias técnicas já foram propostas para aceleração do emparelhamento utilizado na IBE, muitas dessas técnicas seguem a abordagem puramente hardware [5][6][7] ou puramente software [8][9][10]. A maioria das abordagens puramente hardware implementam todo o emparelhamento sobre curvas elípticas em hardware e são normalmente desenvolvidos como módulos de segurança caros e proprietários, ou co-processadores.

A abordagem puramente hardware atinge os melhores resultados no quesito desempenho, contudo ela necessita de maior esforço para ser produzida e oferece baixa flexibilidade quando comparada a outras abordagens. Por esses motivos é mais difícil empregar a abordagem puramente hardware em cenários do mundo real. Contudo, se um sistema especializado necessita computar milhares de emparelhamentos por segundo, essa abordagem se torna necessária, além do mais, ela também oferece um incremento no quesito segurança, uma vez que um ataque de criptoanálise diretamente em hardware é um processo mais caro e complexo de se realizar.

Já a abordagem puramente software (que não utiliza nenhum tipo de instrução customizada ou hardware especializado) é lenta quando comparada à abordagem puramente hardware. Contudo, ela se apresenta como uma solução de baixo custo com alta flexibilidade [11].

Ao longo dessa dissertação será apresentada uma terceira alternativa baseada em uma combinação da abordagem puramente hardware e puramente software. Nessa terceira abordagem o emparelhamento em software conta com ajuda de módulos especializados em hardware para computar partes criticas do emparelhamento. Tal abordagem é subdividida em duas categorias.

Na primeira, a qual será referenciada nesta dissertação como abordagem hardware/software especializada, o hardware possui instruções ou módulos altamente especializados utilizados na resolução de operações criticas do emparelhamento. Na segunda, a qual será referenciada nesta dissertação como abordagem hardware/software genérica, o hardware possui algumas instruções customizadas genéricas inseridas no processador. Essas instruções são utilizadas para realizar o cômputo de algumas partes das operações críticas do emparelhamento. A primeira abordagem apresenta desempenho superior em relação à segunda, enquanto que a segunda apresenta uma flexibilidade maior que a primeira.

A Tabela 1.1 apresenta um resumo da comparação entres os diferentes tipos de abordagens citados.

Abordagem	Característica	Desempenho	Flexibilidade
Puramente hardware	Implementação totalmente em hardware	1º	Nenhuma
Hardware/software especializada	A implementação em software conta com hardware especializado para sua aceleração	2º	Baixa
Hardware/software genérica	A implementação em software conta com hardware genérico para sua aceleração	3º	Média
Puramente software	Não possui otimizações em hardware	4º	Alta

Tabela 1.1: Tipos de Abordagens

Na Tabela 1.1 os números da coluna de desempenho representam a ordem de desempenho esperado para cada uma das abordagens, contudo nem sempre essa ordem é mantida. As implementações em hardware também podem ser subdivididas em implementações em hardware reconfigurável (*FPGA*) ou em *ASIC*.

Serão apresentados agora alguns trabalhos importantes que apareceram em cada uma das abordagens citadas.

1.1.1 Abordagem Puramente Hardware

Beuchat *et al* apresentou em [6] uma modificação do algoritmo de emparelhamento η_T , sobre curvas elípticas supersingulares sobre F_3^m, que não necessitam de nenhum cálculo de raízes cúbicas. Ele também descreve um acelerador para esse emparelhamento totalmente em hardware, prototipado em uma plataforma baseada na FPGA Cyclone II da Altera. Este acelerador é dez vezes mais rápido que a implementação em FPGA anteriormente conhecida. Em [5], Beuchat *et al* apresenta vários algoritmos para computar o emparelhamento η_T na característica três e também são sugeridas várias melhorias ao acelerador em hardware.

1.1.2 Abordagem Puramente Software

Ahmadi *et al* apresentou em [8] alguns resultados para bases normais Gaussianas em F_3^m e usou esses resultados para desenvolver um algoritmo de multiplicação rápido em software. Ele também comparou a velocidade de encriptação e decriptação para o esquema de Criptografia Baseado em Identidades de Boneh-Franklin e Sakai-Kasahara no nível de segurança de 128 bits, nos casos em que são empregadas curvas elípticas supersingulares com graus de mergulho 2, 4 e 6. O trabalho apresentado em [10] implementa uma versão software do emparelhamento η_T usando Java que é executado em um telefone móvel.

1.1.3 Abordagem Hardware/Software

Em [11] Vejda *et al* expõem uma implementação de hardware/software genérica, onde o conjunto de instruções de um processador de propósito geral (um processador SPARC V8) é expandido através de instruções aritméticas customizadas. Diferente do nosso trabalho, que se limita a computação do emparelhamento η_T em característica 3, as instruções em [11] investigam a criação de uma unidade aritmética integrada capaz de realizar operações em três diferentes campos (F_p, F_2^m e F_3^m).

1.2 Objetivos

O objetivo desta dissertação é estudar a Criptografia Baseada em Identidade, compreendendo todas as fases envolvidas na sua execução, assim como desenvolver um sistema de baixo custo capaz de realizar sua execução de forma satisfatória (em tempo satisfatório). Para que seja possível a execução da Criptografia Baseada em Identidades em um sistema de baixo custo (com recursos limitados) é necessário realizar a aceleração do emparelhamento sobre curvas elípticas, que é a operação mais crítica envolvida durante a execução da Criptografa Baseada em Identidades, portanto nosso objetivo é a aceleração do emparelhamento.

Um estudo bibliográfico foi realizado com o intuito de identificar as principais técnicas envolvidas na aceleração do emparelhamento sobre curvas elípticas. A partir desse estudo foram escolhidas as abordagens mais promissoras para serem empregadas no nosso sistema de baixo custo, buscando torná-lo capaz de realizar a computação da Criptografia Baseada em Identidades de forma eficaz. Nosso objetivo é comparar essas diferentes abordagens de aceleração do emparelhamento, buscando identificar os pontos fortes e fracos em cada abordagem. Para isso uma implementação do emparelhamento sobre curvas elípticas que utilize essas diferentes abordagens foi confeccionada e diferentes métricas foram utilizadas na avaliação essas abordagens.

1.3 Organização

Esse trabalho se encontra organizado da seguinte maneira. No capítulo 2 apresentaremos os conceitos matemáticos essenciais para a compreensão do restante do trabalho. Nesse capítulo encontram-se os conceitos básicos de grupo, corpo, curvas elípticas e emparelhamentos. No capítulo 3 apresentamos os conceitos e características da Criptografia Baseada em Identidades, assim como os procedimentos necessários para criptografar e decriptografar as mensagens nesse sistema. O capítulo 4 apresenta a nossa plataforma de trabalho, vislumbrando as características do processador Nios II, assim como estabelecendo a padronização da plataforma utilizada. Já no capítulo 5 os aspectos algorítmicos do sistema são apresentados, assim como as características criptográficas escolhidas para o sistema. As operações envolvidas no emparelhamento são discutidas, e a forma como foram implementadas e validadas é apresentada. Também são apresentados os resultados do mapeamento da execução da implementação do emparelhamento em software. Nos capítulos 6 e 7 as implementações das diferentes abordagens hardware/software desenvolvidas são apresentadas. O capítulo 8 apresenta a comparação entre as diversas abordagens desenvolvidas, mostrando os resultados obtidos e realizando uma análise de cada abordagem. No capítulo 9 as conclusões e observações do nosso trabalho são expostas e por fim, no capítulo 10 apresentamos os possíveis trabalhos complementares que podem ser realizados no futuro em relação a este trabalho.

Capítulo 2

Fundamentos Matemáticos

Este capítulo apresenta os fundamentos matemáticos básicos que serão utilizados no decorrer dessa dissertação. Apresentaremos uma breve introdução a grupos, corpos, curvas elípticas e emparelhamentos sobre curvas elípticas. Definições mais concisas e aprofundadas podem ser encontradas em [12], [13] e [16].

2.1 Grupo

Um Grupo pode ser definido como um conjunto não vazio G, dotado de uma operação binária "o" tal que G o G $\rightarrow$ G satisfaz as seguintes propriedades [14]:

- Elemento *identidade*: $\exists n \in G : \forall a \in G : a \circ n = n \circ a = a$;
- Elemento *inverso*: $\forall a \in G : \exists \neg a \in G : a \circ \neg a = n$;
- Propriedade *associativa*: $\forall a, b, c \in G : (a \circ b) \circ c = a \circ (b \circ c)$;
- *Fechamento*: $\forall a, b \in G : a \circ b \in G$.

Para ser considerado um grupo *abeliano* (também conhecido com grupo *comutativo*) um grupo deve possuir uma propriedade adicional, a *comutatividade*:

$$\forall a, b \in G : a \circ b = b \circ a.$$

Quando a operação "o" equivale à operação de adição "+", dizemos que o grupo é um grupo *aditivo*. Nesse cenário o elemento identidade é definido como 0 e o elemento inverso de a é definido como $-a$.

A *multiplicação por um escalar* de um número inteiro positivo x por $a \in G$ é definida como $xa = a + a + ... + a$; ou seja, x somas do elemento a.

Um grupo *aditivo* é dito cíclico se existe um elemento $P \in G$ tal que qualquer elemento $Q \in G$ pode ser escrito como múltiplo escalar de P, ou seja, $Q = xP$ para algum inteiro x. Um elemento P com tal propriedade é definido como gerador de G.

Quando a operação "o" equivale à operação de multiplicação "·", dizemos que o grupo é um grupo *multiplicativo*. Nesse cenário o elemento identidade é definido como 1 e o elemento inverso de a é definido como a^{-1}. A *exponenciação* a^x, onde x é número inteiro positivo e $a \in G$, é definida como x multiplicações de a, ou seja $a^x = a \cdot a \cdot ... \cdot a$; com x termos.

Quando o número de elementos de G é finito, esse número é chamado de *ordem* de G.

2.2 Corpo

Um corpo F é uma estrutura algébrica que consiste de um conjunto F e duas operações binárias [14].

- Adição ("+"): $F + F \rightarrow F$;
- Multiplicação ("·"): $F \cdot F \rightarrow F$.
- Satisfazendo as seguintes propriedades:
- (F, +) é um grupo *abeliano*, com identidade aditiva denotada por 0;
- (F\{0}, ·) é um grupo *abeliano*, com identidade multiplicativa denotada por 1;
- O elemento identidade aditiva é diferente do elemento identidade multiplicativa ($0 \neq 1$);
- Distributividade: $x \cdot (y + z) = x \cdot y + x \cdot z$, $\forall x, y, x \in F$.

Se o conjunto F é finito, então ele é chamado de um corpo *finito* (ou corpo de *Galois*). A *ordem* de um corpo *finito* é o número de elementos no corpo. Existe um corpo *finito* F de *ordem* q se e somente se q é uma potência prima, isto é, $q = p^m$, onde p é um número primo (chamado característica de F), e m é um inteiro positivo [15].

2.3 Curvas Elípticas

Esta seção é baseada nos trabalhos [14][16][17][3] e apresenta uma breve introdução a curvas elípticas. Informalmente, curvas elípticas são curvas "planas", isto é, definidas por equações em duas variáveis.

Seja F um corpo e os elementos a, b, c, d, $e \in$ F. Uma curva elíptica sobre F, denotada E(F), é o conjunto de pares $(x, y) \in$ F x F que satisfaz uma equação da forma:

$$y^2 + axy + by = x^3 + cx^2 + dx + e$$

Acrescenta-se por conveniência um ponto extra $\mathcal{O}$, chamado de *ponto no infinito*. O número de pontos de uma curva elíptica E (isto é, o número de soluções da equação da curva mais o *ponto no infinito*) é chamado de a *ordem* da curva, denotada por #E. Para corpos finitos F_q, o *Teorema de Hasee* diz que a *ordem* da curva satisfaz:

$$\#E \leq q + 1 + t$$

Onde $|t| \leq 2\sqrt{q}$ é conhecido na literatura como traço de *Frobenius* da curva. Um subgrupo G de uma curva elíptica $E(F_q)$ é dito ter grau de mergulho k se sua ordem r divide $q^k - 1$, mas não divide $q^i - 1$ para todo $0 < i < k$. Em outras palavras, k é o menor inteiro tal que r | $(q^k - 1)$.

O grupo $E(F_q)$ é isomorfo ao grupo $E(F_{q^k})$. Sendo assim, se P é um ponto de ordem r em E(F_q), $E(F_{q^k})$ contém um ponto Q de mesma ordem, linearmente independente a P. Um alto grau de mergulho é importante para que possamos ter um elevado nível de segurança em sistemas criptográficos baseados em curvas elípticas. Por outro lado, é possível fazer importantes simplificações usando curvas com pequeno grau de mergulho.

A equação da curva pode ser simplificada por meio de mudanças de coordenadas. Se a característica do corpo F for 2, a equação reduz-se a uma das formas:

$$y^2 + xy = x^3 + ax^2 + b$$
$$y^2 + cy = x^3 + ax + b$$

Para corpos de característica 3, a equação pode ser reescrita como:

$$y^2 = x^3 + ax^2 + bx + c$$

Por último, para corpos de característica diferente de 2 e de 3, a equação de ser transformada em:

$$y^2 = x^3 + ax + b \qquad (1)$$

Para que os pontos da curva elíptica formem um grupo, precisamos definir uma operação binária sobre eles e especificar os elementos identidade e inverso. A construção da lei de grupos para curvas sobre R tem caráter geométrico, com base em retas secantes e tangentes à curva. Em certos tipos de curva não é possível a construção de um grupo. Isso ocorre quando o polinômio f(x) = $x^3 + ax + b$ possui raízes múltiplas, ou seja, o discriminante ! $\equiv 4a^3 + 27b^2$ é igual a zero.

Sobre R, existem as chamadas curvas *singulares* ou *degeneradas*, em que o discriminante sempre é zero; elas possuem a forma geral $y^2 = x^3 - 3t^2x + 2t^3$.

Quando o corpo é finito, o discriminante da equação da curva deve ser diferente de zero (módulo #*E*) para que seja possível formar um grupo de pontos da curva sobre os pontos da curva.

As curvas elípticas da forma (1) são simétricas em relação ao eixo X. Portanto, define-se o elemento inverso de um ponto $P = (x, y)$ como sendo o ponto simétrico $-P = (x, -y)$. O elemento identidade é o ponto no infinito $\mathcal{O}$. Em curvas com discriminante diferente de zero, a operação binária do grupo é a soma de dois pontos *P* e *Q*, definida da seguinte maneira:

- $P + \mathcal{O} = \mathcal{O} + P = P, \forall P$;
- Se $P = -Q$, então $P + Q = \mathcal{O}$;
- Se $P \neq Q$, traça-se uma reta secante à curva, pelo ponto *P* e *Q*. Pode-se provar que essa reta sempre intercepta a curva num terceiro ponto *R*; define-se então $P + Q = -R$;
- Se $P = Q$, traça-se uma reta tangente à curva pelo ponto *P*. Pode-se provar que essa reta sempre intercepta a curva em um segundo ponto *R*; define-se então $P + Q = P + P = 2P = -R$;

Uma classe de curvas que merece destaque em criptografia são as curvas ditas *supersingulares*. Uma curva é dita *supersingular* se o seu traço de *Frobenius* é múltiplo da característica *p* do corpo sob a qual a curva está definida, ou seja se $t \mid p$. Um fato importante em curvas supersingulares é que nelas o grau de mergulho *k* é sempre menor ou igual a 6.

Os parâmetros das curvas elípticas *supersingular* devem ser escolhidos cuidadosamente, para não sofrerem ataques como o MOV, contudo, esse tipo de curva se mostra muito eficiente em relação ao desempenho [3].

2.4 Emparelhamentos

Os emparelhamentos sobre curvas elípticas foram inicialmente introduzidos à criptografia com o intuito de realizar quebra de códigos criptográficos. Posteriormente a isso, identificou-se suas propriedades construtivas, que passaram a ser o coração de muitos sistemas criptográficos [5].

Desde o fundamental trabalho de Joux [39], um número cada vez maior de sistemas baseados em emparelhamentos estão surgindo. Veremos agora uma breve definição do emparelhamento sobre curvas elípticas.

O emparelhamento de curvas elípticas pode ser definido da seguinte forma: Sejam G_1 e G_2 dois grupos de ordem q, para algum número primo q muito grande. Definimos emparelhamento (*pairing*) como um mapeamento ê entre esses grupos, isto é:

$$\hat{e}(G_1, G_1) : G_1 \times G_1 \rightarrow G_2$$

Segundo Boneh e M. Franklin [4], para que tais mapeamentos possam ser utilizados em sistemas criptográficos, eles devem satisfazer as seguintes propriedades:

- Ser bilinear: dizemos que um mapeamento ê: $G_1 \times G_1 \rightarrow G_2$ é bilinear se: $\hat{e}(aP, bQ) = \hat{e}(P, Q)^{ab} \ \forall P, Q \in G_1$ e $\forall$ a, b $\in Z$;
- Ser não-degenerado: um mapeamento é dito não-degenerado se não mapeia todos os pares $G_1 \times G_1$ para a identidade em G_2. Observe que como G_1 e G_2 são grupos de ordem prima, isto implica que se P é um gerador de G_1, então $\hat{e}(P, P)$ é um gerador de G_2;
- Ser computável: dizemos que um mapeamento é computável se existe um algoritmo eficiente (i.e., tempo polinomial) para computar $\hat{e}(P, Q)$ para todo $P,Q \in G_1$.

Um mapeamento que satisfaça tais propriedades é dito um mapeamento *admissível*. Os emparelhamentos de *Weil*, *Tate* e η_T são exemplos de mapeamentos admissíveis para os grupos G_1 e G_2 e podem ser utilizados na implementação de sistemas Criptográficos Baseados em Identidades.

Como exemplo, segue agora uma definição mais formal de um emparelhamento *Tate* extraído de [5]. Um emparelhamento bilinear *Tate* sobre curvas elípticas pode ser definido da seguinte forma:

- Tome E como uma curva elíptica supersingular sobre F_{p^m} (nesta dissertação $F_{p^m} = F_{3^{97}}$), onde p é um número primo e m um inteiro positivo; $E(F_{p^m})$ denota um grupo desses pontos que pertencem à curva E;
- Defina $\ell > 0$ como um inteiro relativamente primo à p (ou seja, não existem divisores comuns entre ℓ e p);
- Tome k como o grau de mergulho tal que k seja o menor inteiro positivo que satisfaz $p^{km} \equiv 1 \pmod{\ell}$;
- $E(F_{p^m})[\ell]$ define o subgrupo *ℓ-torsion* de $E(F_{p^m})$, isto é, o conjunto de elementos P de $E(F_{p^m})$ que satisfaz $[\ell]P = \mathcal{O}$, onde $\mathcal{O}$ é o ponto no infinito da curva elíptica;
- Tome $P \in E(F_{p^m})[\ell]$ e $\in E(F_{p^{km}})[\ell]$, defina $f_{\ell, P}$ como uma função racional sobre a curva com divisor $\ell(P) - \ell(\mathcal{O})$, tal que exista um divisor D_Q equivalente à $(Q) - (\mathcal{O})$, com um suporte separado do suporte de $f_{\ell, P}$;
- Finalmente o emparelhamento *Tate* de ordem ℓ é um mapeamento ê: $E(F_{p^m})[\ell] \times E(F_{p^{km}})[\ell] \rightarrow F^*_{p^{km}}$ definido por $\hat{e}(P,Q) = f_{\ell, P}(D_Q)^{\left(p^{km}-1\right)/\ell}$.

O tipo de elevação que ocorre nessa definição é chamada de *Exponenciação Final*, ela torna possível obter um valor em um subgrupo multiplicativo de $F^*_{p^{km}}$ (que é requerido pela maioria das aplicações criptográficas) ao invés de um subgrupo multiplicativo de um quociente de $F^*_{p^{km}}$.

Em [18] Barreto *et al* prova que este emparelhamento pode ser computado como $\hat{e}(P,Q) = f_{\ell,P}(Q)^{\left(p^{km}-1\right)/\ell}$, onde $f_{\ell,P}$ é calculado em um ponto ao invés de um divisor. Graças ao mapa de distorção $\psi : E(F_{p^m})[\ell] \rightarrow E(F_{p^{km}})[\ell]$ é possível definir uma modificação de emparelhamento *Tate* como $\hat{e}(P, Q) = e(P, \psi(Q))$ para todo P e $Q \in E(F_{p^m})[\ell]$.

Barreto *et al* apresenta em [19] um aperfeiçoamento da técnica Duursma-Lee [40]. Isso tornou possível calcular de forma eficiente e emparelhamento *Tate*, criando assim o emparelhamento η_T. No capítulo 5 apresentaremos o emparelhamento η_T.

Capítulo 3

Criptografia Baseada em Identidades

O conceito de Criptografia Baseada em Identidades foi introduzido em 1984 por Shamir em [2]. Nesse artigo, o autor propôs um novo modelo criptográfico, que permitiria a qualquer par de usuários se comunicarem de forma segura, sem que fosse necessário o pré-estabelecimento de chaves secretas, como ocorre na criptografia simétrica, e sem que fosse preciso utilizar certificados digitais e autoridades certificadoras para a autenticação das chaves públicas (*Public Key Infrastructure* - PKI), como ocorre na criptografia assimétrica tradicional.

O esquema proposto por Shamir baseia-se no esquema de criptografia assimétrica tradicional, sendo que em vez de termos um par de chaves representadas por uma seqüência de bits sem significado intrínseco, teremos como chave pública um identificador, ou seja, uma característica que identifica o usuário de forma única. Como exemplos de identificadores, poderíamos citar o número do RG ou o endereço eletrônico (*e-mail*) de um individuo. A grande vantagem deste esquema é que, ao contrário da criptografia assimétrica tradicional, não há necessidade de se fazer um mapeamento entre uma chave pública e seu dono, haja vista que, nesse caso, a chave pública já identifica inequivocamente o seu dono.

Outra vantagem é que, por não ser mais um número aleatório, um usuário não necessita reservar espaço adicional para guardar as chaves públicas das pessoas com quem ele deseja se comunicar, sendo possível usar sua própria lista de endereços eletrônicos como chaves. Tais características fazem com que a criptografia assimétrica baseada em identidades se assemelhe ao correio físico, ou seja, se você conhece o endereço de uma pessoa, você pode enviar-lhe uma mensagem, de modo que somente ela poderá ler [3].

Nesse esquema criptográfico, existe uma entidade definida como o Gerador de Chaves Particulares (*Private Key Generator* - PKG), cuja principal função é manter a custódia de chaves secretas dos usuários geradas através da chave-mestra da PKG (que apenas o PKG conhece) e transmiti-las aos seus respectivos donos por um canal seguro. Naturalmente, antes de gerar e distribuir essa chave, o PKG fará uma cuidadosa investigação com o objetivo de autenticar o solicitante, da mesma forma que ocorre em uma verificação de identidade para emissão de certificados na criptografia assimétrica tradicional. Embora a entidade PKG seja indesejável em certos ambientes, ela é muito útil em ambientes hierárquicos.

A Figura 3.1 apresenta um cenário onde uma comunicante Ana envia uma mensagem para a comunicante Carol utilizando para isso a IBE. Observe que na encriptação da mensagem são utilizados o identificador de Carol (ID_{Carol}) e a chave pública da PKG.

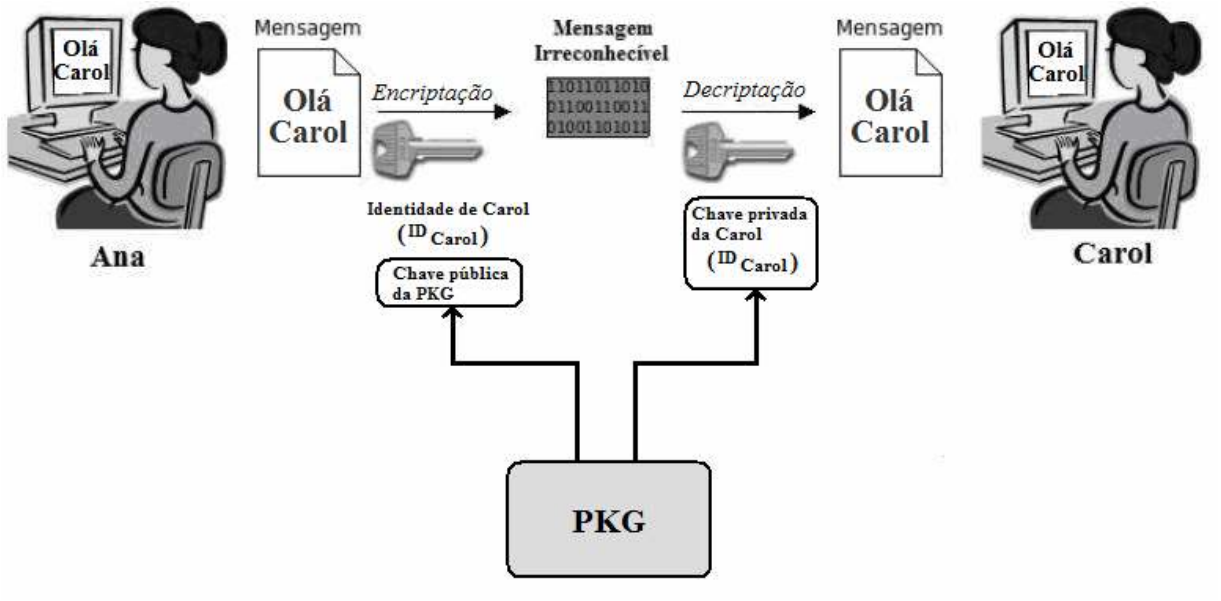

Figura 3.1: Criptografia Baseada em Identidades

Uma característica relacionada ao IBE é a possibilidade de gerar mensagens com datas definidas, uma vez que é possível concatenar uma data na chave usada para cifrar certa mensagem para um dado usuário, nessas condições o usuário destinatário só terá acesso a essa informação após a data definida na chave. Essa característica pode ser utilizada em leilões digitais, onde os lances só devem ser acessados após certa data.

Em [2], Shamir propôs um esquema de assinatura baseada em identidades, cuja segurança está na dificuldade de fatoração de números inteiros grandes, assim como no RSA. Nesse artigo Shamir considerou como um problema em aberto um esquema de Criptografia Baseada em Identidades. Somente com o esquema proposto por Boneh e Franklin em 2001 [4], que emprega emparelhamentos de curvas elípticas na implementação do IBE, que se conseguiu uma solução satisfatória para o IBE. Devido a essa nova abordagem, o problema central do IBE passou a ser a implementação eficiente do emparelhamento sobre curvas elípticas. Com isso, diversos esquemas para a implementação do emparelhamento sobre curvas elíptica foram propostos, nesta dissertação iremos apresentar alguns deles.

3.1 Funções Hash Utilizadas

Uma função hash é uma função de transformação que recebe como entrada uma cadeia *x* com número arbitrário de bits e retorna como saída uma cadeia *y* com número fixo de bits. A cadeia *y* é conhecida como resumo de *x*. Para que uma função hash possa ser utilizada em criptografia ela deve satisfazer algumas propriedades [3]:

- Dado um x qualquer, é computacionalmente fácil calcular H(x), onde H é a função hash e x é uma cadeia arbitrária de bits;
- Resistência à primeira inversão. É computacionalmente fácil calcular *y* tal que H(*x*) = *y*. Contudo, é computacionalmente inviável recuperar o valor de *x* através de *y*;
- Resistência à segunda inversão, isto é, dado um valor *x*, é computacionalmente inviável encontrar um valor $z \neq x$ tal que H(*x*) = H(*z*);
- Resistência à colisões, onde é computacionalmente inviável encontrar quaisquer dois valores *x* e *z* tais que H(*x*) = H(*z*).

Nos esquema IBE que iremos ver ao longo desta dissertação são utilizadas duas funções hash. Essas funções são definidas da seguinte forma:

- $H_1 : \{0, 1\}^* \rightarrow E(F_p^m)$
- $H_2 : E(F_p^{6m}) \rightarrow \{0, 1\}^*$

Ou seja, H_1 é uma função hash que tem como entrada uma seqüência de bits de tamanho arbitrário e saída um ponto da curva elíptica; H_2 é uma função hash que tem como entrada o resultado de um emparelhamento entre dois pontos da curva e como saída uma seqüência de bits de tamanho arbitrário.

Em [37] é apresentada uma maneira eficiente (evitando o uso de raízes quadradas) de se implementar a função H_1 para o caso de F_3^m. Já a função H_2 pode ser implementada através da utilização dos bits que compõem o ponto em F_p^{6m}, uma vez que essa função será utilizada apenas para gerar um OU-Exclusivo com a mensagem [3].

3.2 Formação das chaves

Na IBE dois tipos de chaves são usadas: chaves públicas (usadas como parâmetro público) e chaves pessoais. Vamos agora definir cada um desses tipos.

3.2.1 Chaves Públicas

Sejam $R \in E(F_{p^m})$, $s \in F_p$ e P um ponto gerador de $E(F_{p^m})$ e de conhecimento público. Temos que: $R = sP$. Note que apesar dos valores de R e P serem públicos, é computacionalmente inviável calcular o valor de s dados R e P, uma vez que esse cálculo está protegido pelo Problema de Logaritmos Discreto sobre Curvas Elípticas [3].

3.2.2 Chaves Pessoais

Sejam Q_{ID}, $S_{ID} \in E(F_{p^m})$, chaves pública e secreta de um dado usuário e suponha que exista uma autoridade confiável (em nosso caso a PKG) com um par de chaves (R_{AC}, s) de modo que valem as seguintes relações:

- $S_{ID} = sQ_{ID}$
- $Q_{ID} = H_1(ID)$

Onde ID é o identificador único do usuário (como o seu *e-mail* ou RG). Note que apesar dos valores de Q_{ID} e S_{ID} serem públicos, é computacionalmente inviável calcular o valor de s dados Q_{ID} e S_{ID}, uma vez que esse cálculo esta protegido pelo Problema de Logaritmos Discreto sobre Curvas Elípticas.

Após identificar corretamente o usuário do ID, o cálculo da chave S_{ID} é realizado pelo PKG e é posteriormente enviado para o usuário por meio de um canal seguro. A PKG tem conhecimento das chaves secretas de todos os usuários e a segurança do sistema se encontra na obscuridade da chave secreta s (ou chave mestra) da PKG [3].

3.3 Encriptação

Suponha que um usuário denominado Beto deseje enviar uma mensagem (*men*) para Ana usando para isso um identificador de Ana. Sob tal contexto, Beto deve proceder da seguinte forma:

1) Inicialmente Beto gera aleatoriamente um elemento $r \in F_p$. O elemento r é conhecido na literatura como *NONCE* (*Number used ONCE*);
2) Beto gera então a mensagem $M_1 = rP$, onde P é um parâmetro público conhecido por todos, tal que $P \in E(F_{p^m})$ e a chave $R_{AC} = sP$;
3) Beto gera a mensagem $M_2 = men \oplus H_2(\hat{e}(rQ_{ANA}, R_{AC}))$; (3.1)
4) Finalmente Beto envia para Ana as mensagens M_1 e M_2.

3.4 Decriptação

Após receber as mensagens M_1 e M_2 de Beto, Ana realiza o seguinte cálculo para obter a mensagem original cifrada por Beto:

$$men = M_2 \oplus H_2(\hat{e}(S_{ANA}, M_1)) \qquad (3.2)$$

Para a demonstração de que o cálculo da fórmula definida em (3.2) realmente chega à mensagem original *men*, procedemos da seguinte forma:

$M_2 \oplus H_2(\hat{e}(S_{ANA}, M_1)) = M_2 \oplus H_2(\hat{e}(S_{ANA}, rP))$, uma vez que $M_1 = rP$;
$= M_2 \oplus H_2(\hat{e}(sQ_{ANA}, rP))$, dado que $S_{ANA} = sQ_{ANA}$;
$= M_2 \oplus H_2(\hat{e}(Q_{ANA}, P)^{sr})$, pela propriedade de bilinearidade de ê;
$= M_2 \oplus H_2(\hat{e}(rQ_{ANA}, sP))$, pela propriedade de bilinearidade de ê;
$= M_2 \oplus H_2(\hat{e}(rQ_{ANA}, R_{AC}))$, pois $R_{AC} = sP$;
$= men \oplus H_2(\hat{e}(rQ_{ANA}, R_{AC})) \oplus H_2(\hat{e}(rQ_{ANA}, R_{AC}))$, por (3.1)
$= men$, pelas propriedades do operando OU-Exclusivo.

Como podemos observar, a Criptografia Baseada em Identidade tem como parâmetros públicos os seguintes elementos: p, m, P, R_{AC}, ê, H_1 e H_2.

Capítulo 4

A Plataforma Utilizada

O desenvolvimento de uma solução HW/SW para o cálculo do emparelhamento requer uma plataforma de *FPGA* (*Field Programmable Gate Array*) flexível que seja capaz de permitir a fácil interação entre o programa em software e os módulos em hardware. A plataforma escolhida foi um kit de desenvolvimento do processador Nios II da Altera [20], baseado em um *FPGA Stratix II* [21].

Nios II é um processador de 32 bits da Altera que permite a inserção de novas instruções em seu conjunto de instruções, ou seja, ele é uma plataforma flexível para o teste de novos hardwares. Ele conta também com um barramento *Avalon* que permite a inserção de novos módulos ao barramento de dados do processador.

Tal plataforma possui o auxilio de software especializado como o *Altera Quartus II* [22] e um compilador *GNU GCC* capaz de gerar código executável otimizado a partir de códigos em *C/C++*.

A linguagem *VHDL* (*Very High Speed Integrated Circuits Hardware Description Language*) foi usada na implementação dos módulos e instruções especializadas em hardware. A ferramenta *SOPC Builder*, que é parte integrante do *Altera Quartus II*, foi utilizada para gerar a inserção do hardware especializado diretamente no processador. Essa ferramenta gera automaticamente bibliotecas em *C*, as quais contêm as assinaturas das chamadas utilizadas no acesso ao hardware especializado, que foi adicionado ao processador, simplificando assim o acesso aos módulos e instruções desenvolvidos.

O processador Nios II é capaz de calcular um emparelhamento sobre curvas elípticas puramente em software, contudo, com baixo desempenho.

4.1 Processador Nios II

Como apresentado em [20], o processador Nios II é um processador RISC de propósito geral. O sistema ao qual o processador Nios II pertence é equivalente a um microcontrolador ou a um *computer on a chip*, que inclui um processador, memória e uma combinação de diferentes

periféricos em um único chip. O sistema a qual o processador Nios II pertence consiste de um núcleo do processador Nios II, uma memória *on-chip*, um conjunto de periféricos *on-chip*, uma interface para memória *off-chip*; tudo implementado em um simples dispositivo *FPGA* da Altera. Outras características que o processador Nios II possui são:

- Conjunto de instruções, *datapath* e espaço de endereçamento em 32 bits;
- 32 fontes externas de interrupção;
- Multiplicador de 32 x 32 e divisor com resultado de 32 bits;
- Instrução dedicada para computação de produtos de multiplicação de 64 bits e 128 bits;
- Instruções para operação em ponto flutuante de precisão simples;
- Instrução de *barrel shifter*;
- Acesso a uma variedade de periféricos *on-chip*, e interface para periféricos e memórias *off-chip*;
- Módulo assistente de debug em hardware, possibilitando gerar comandos de *start*, *stop*, *step* e *trace* através de um ambiente de desenvolvimento integrado (*integrated development environment* - IDE);
- Suporte a unidade de gerenciamento de memória (*memory management unit* - MMU), para dar suporte a sistemas que o requerem;
- Suporte a unidade de proteção de memória (*memory protection unit* - MPU);
- Ambiente de desenvolvimento de software baseado na *GNU C/C++* e *Eclipse* IDE;
- Conjunto de instruções da arquitetura compatível (*Instruction set architecture* - ISA) com todos os sistemas Nios II;
- Desempenho de mais de 250 DMIPS (*Dhrystone Million Instructions Per Second*).

Na prática, muitos projetos em *FPGA* implementam alguns dispositivos lógicos extras além do sistema do processador Nios II. Por isso o processador Nios II provê flexibilidade para adicionar elementos e melhorias de desempenho no sistema do processador Nios II.

Convenientemente, é possível também eliminar elementos desnecessários do sistema do processador, tornando assim o sistema mais enxuto e capaz de executar em dispositivos de baixo custo. A Figura 4.1 apresenta um exemplo de configuração possível do processador Nios II em um kit de desenvolvimento da Altera e a Figura 4.2 apresenta um exemplo de configuração possível do núcleo do processador Nios II.

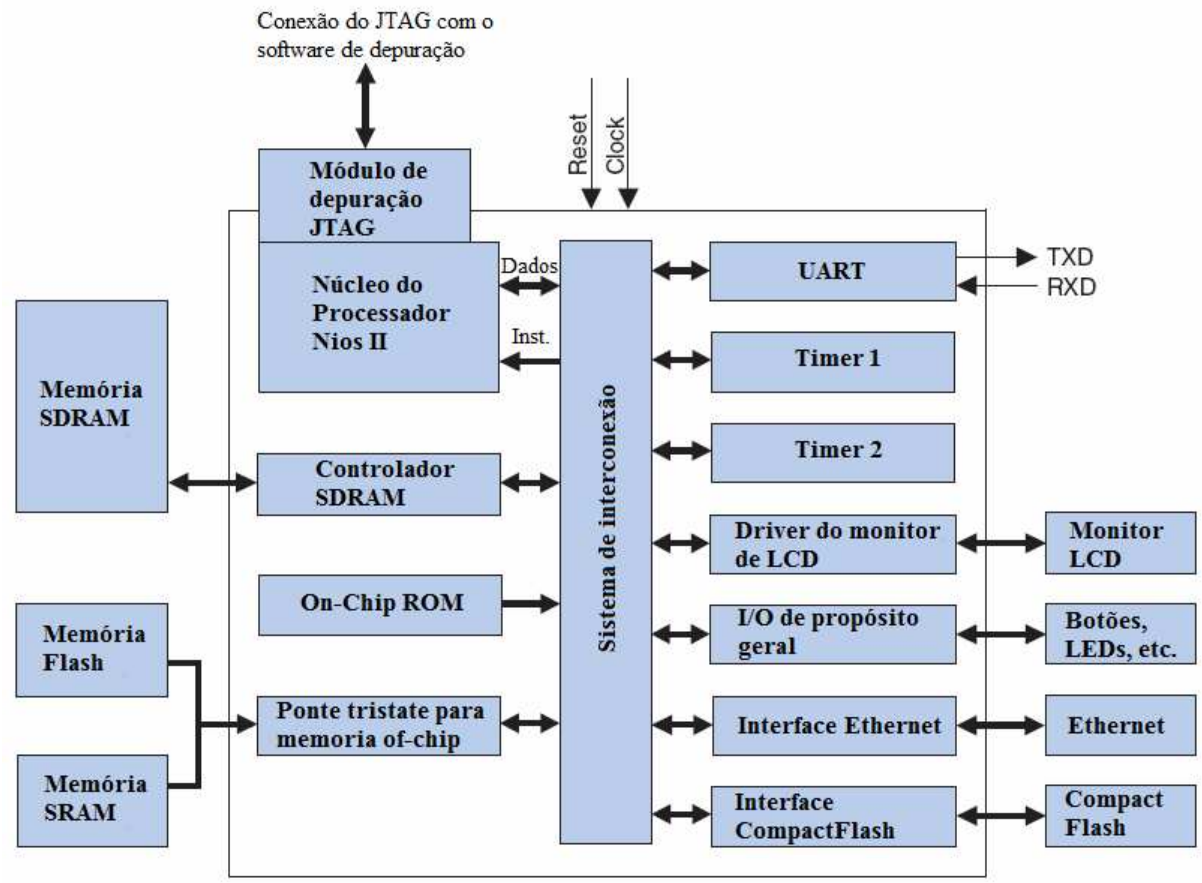

Figura 4.1: Exemplo de Configuração de um Sistema do Processador Nios II (extraído de [20])

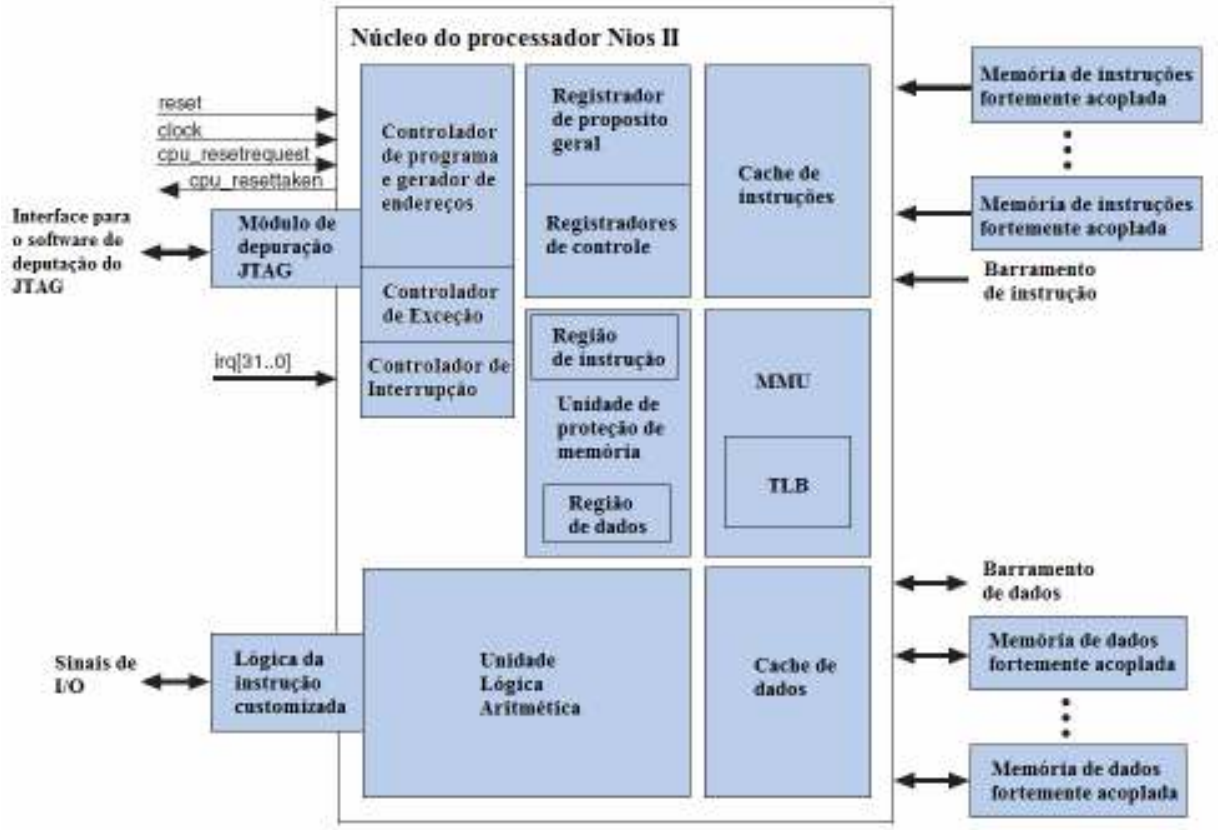

Figura 4.2: Exemplo de Configuração do Núcleo do Processador Nios II (extraído de [20])

4.1.1 Módulos Customizados

É possível criar periféricos (ou módulos) *customizados* e integrados ao sistema do processador Nios II. Para elementos críticos do sistema que gastam muitos ciclos de processamento executando partes especificas de código, uma técnica comum é criar periféricos customizados que implementam a mesma funcionalidade em hardware, acelerando assim a execução do sistema. Essa abordagem apresenta dois benefícios de desempenho: a implementação em hardware é mais rápida que a em software; e o processador fica livre para realizar outras operações em paralelo enquanto o periférico *customizado* opera.

4.1.2 Instruções Customizadas

Como os módulos *customizados*, as instruções *customizadas* possibilitam alcançar um ganho de desempenho por meio de adição de hardware customizado no processador. A natureza soft-core do processador Nios II permite a integração de lógicas *customizadas* na unidade de lógica aritmética (ALU) do processador.

Análogo as instruções nativas do Nios II, as instruções customizada podem receber como entrada dois registradores como fonte e escrever o resultado em um registrador de destino. Como o processador é implementável em *FPGAs*, engenheiros de software e hardware podem trabalhar juntos para interativamente aperfeiçoarem o hardware e testar os resultados da execução do software no hardware.

4.2 Padronização do Sistema

No sentido de comparar diferentes abordagens estudadas nesta dissertação o mais precisamente quanto possível, foi utilizada a mesma plataforma baseada no Nios II em todos os experimentos. A configuração usada possui um processador *pipelined* de seis estágios executando a 50MHz (*clock* do kit de desenvolvimento do processador Nios II da Altera), com *branch prediction* dinâmico e com o auxilio de multiplicador em hardware.

Buscando diminuir o número de *caches misses*, que podem interferir nos resultados entre as diferentes abordagens, são utilizadas *caches* de tamanho relativamente grande, 16 KB e 32 KB respectivamente para *cache* de dados e instruções.

Em complemento à *cache*, uma memória SRAM de 128 KB interna ao *FPGA* foi usada para armazenar os códigos e dados do programa. Para assegurar a eficácia das medições no sistema, todas as operações são funções, não importando se implementadas em hardware ou software. Essas funções são chamadas múltiplas vezes usando uma estrutura em repetição (*loop*) durante a realização das medições do sistema.

Para a sintetização dos hardwares e a programação do *FPGA* (*Stratix II*) foi utilizado o software *Quartus II* versão 6.1 da Altera. Na compilação dos códigos em C para o processador Nios II foi utilizado o compilador *GCC* da ferramenta Nios II IDE versão 6.1, utilizando o parâmetro de compilação –O3.

Capítulo 5

Aspectos Algorítmicos do Sistema

5.1 A Implementação

Em geral, as implementações dos algoritmos para IBE se concentram em três tipos de emparelhamentos (*Weil*, *Tate* e η_T) e na otimização das operações associadas a esses emparelhamentos, buscando a diminuição do tempo necessário para calculá-los. Isso se deve ao fato do emparelhamento sobre curvas elípticas ser a operação dominante durante a execução da encriptação/decriptação de mensagens na IBE.

Em nosso trabalho, o tipo de emparelhamento escolhido foi o η_T e o algoritmo escolhido para implementação foi o proposto por Beuchat *et al* em [5]. Uma modificação do algoritmo livre de raízes cúbicas apresentado em [5] foi escolhido como base para a nossa implementação do emparelhamento.

Uma implementação didática em software desse algoritmo foi realizada na linguagem de programação *C* e posteriormente testada através de suas propriedades matemáticas. A implementação do emparelhamento pode ser decomposta em diversas operações, tais como: multiplicação, elevação ao cubo, inversão de elementos, etc.

A forma como as operações foram implementadas na linguagem de programação *C* se aproxima muito da forma como são implementadas em hardware, apesar de isso tornar a implementação mais lenta e trabalhosa, o mapeamento da versão em software para a versão em hardware é mais simples e natural. A seguir serão apresentadas todas as operações que compõem o emparelhamento, assim como a abordagem utilizada para a validação dessas operações.

5.2 Algoritmo do Emparelhamento

Como dito antes, em nosso trabalho, o tipo de emparelhamento escolhido foi o η_T e o algoritmo escolhido para implementação foi o proposto por Beuchat *et al* em [5]. O emparelhamento η_T pode ser definido da seguinte forma:

- Defina E como uma curva elíptica *supersingular* de acordo com a equação $E: y^2 = x^3 - x + b$, onde $b \in \{-1, 1\}$;
- Considerando um inteiro positivo m co-primo a 6, o número de pontos racionais da curva E sobre o corpo finito $F_3{}^m$ é dado por: $N = \#E(F_3{}^m) = 3^m + 1 + \mu b 3^{\frac{m+1}{2}}$. Com $\mu = +1$ se $1, 11 \equiv (\text{mod } 12)$ ou $\mu = -1$ se $5, 7 \equiv (\text{mod } 12)$;
- O grau de mergulho de E é então 6;
- Escolhendo $T = 3^m - N = -\mu b 3^{\frac{m+1}{2}} - 1$ e um inteiro ℓ que divide N;
- Nós definimos o emparelhamento η_T entre dois pontos P e Q como uma ℓ-torção $E(F_3{}^m)[\ell]$:

$$\eta_T(P, Q) = f_{T, P}(\psi(Q)) \text{ se } T > 0 \text{ (ou seja } \mu b = -1); \text{ ou}$$
$$f_{-T, -P}(\psi(Q)) \text{ se } T < 0 \text{ (ou seja } \mu b = 1)$$

Onde:

- ψ é um mapa de distorção de $E(F_3{}^m)[\ell]$ para $E(F_3{}^{6m})[\ell]$ definido como $\psi(x, y) = (\rho - x, y\sigma)$ para todo $(x, y) \in E(F_3{}^{97})[\ell]$, como dado em [18], onde ρ e σ são elementos em $E(F_3{}^{6m})[\ell]$ satisfazendo a equação $\rho^3 - \rho - b = 0$ e $\sigma^2 + 1 = 0$. Isso permite representar $F_3{}^{6*m}$ como uma extensão de $F_3{}^m$ usando a base $(1, \sigma, \rho, \sigma\rho, \rho^2, \sigma\rho^2)$: $F_3{}^{6*97} = F_3{}^{97}[\rho, \sigma] \cong F_3{}^{97}[X, Y] / (X^3 - X - b, Y^2 + 1)$. Conseqüentemente, toda computação em $F_3{}^{6m}$ pode ser substituída por computação em $F_3{}^m$.
- $f_{n, P}$ para $n \in \mathrm{N}$ e $P \in E(F_3{}^m)[\ell]$ é uma função racional definida sobre $E(F_3{}^{6m})[\ell]$ com divisor $(f_{n, P}) = n(P) - ([n]P) - (n - 1)(\mathcal{O})$.

Para garantir que os valores obtidos do emparelhamento pertencem ao grupo da ℓ-ésima raiz da unidade $F^*{}_3{}^{6m}$, temos que calcular o emparelhamento η_T reduzido, que é definido como $\eta_T(P, Q)^{\mathrm{M}}$, onde:

$$M = \frac{3^{6m}-1}{N} = (3^{3m}-1)\cdot(3^m+1)\cdot(3^m+1-\mu b 3^{(m+1)/2})$$

Esse último passo é definido como Exponenciação Final. Maiores detalhes sobre o emparelhamento η_T podem ser encontrados em [19] e [5].

Uma modificação do algoritmo livre de raízes cúbicas apresentado em [5] (Algoritmo 5.1) foi selecionado como base para a nossa implementação do emparelhamento. Algumas características desse algoritmo podem ser observadas na Tabela 1. Emparelhamentos com essas características vêm sendo amplamente utilizados na literatura [5][6][10]. Um emparelhamento com tais características também já foi implementado em *ASIC* [24].

Emparelhamento η_T

Entrada: $P,Q \in E(F_3^{97})[\ell]$.
Saída: $\eta_T(P,Q) \in F_3^{6*97}$.
1: $y_P \leftarrow \mu y_P$;
2: $x_Q \leftarrow x_Q^3$; $y_Q \leftarrow y_Q^3$;
3: $t \leftarrow x_P + x_Q$;
4: $R \leftarrow (\lambda y_P t - \lambda y_Q \sigma - \lambda y_P \rho)\cdot(t^2 + y_P y_Q \sigma - t\rho - \rho^2)$;
5: $d \leftarrow 1$;
6: **for** $j \leftarrow 1$ **to** 48 **do**
7: $R \leftarrow R^3$;
8: $d \leftarrow d - 1$ **mod** 3
9: $x_Q \leftarrow x_Q^9$; $y_Q \leftarrow -y_Q^9$;
10: $t \leftarrow x_P + x_Q + d$;
11: $S \leftarrow -t^2 + y_P y_Q \sigma - t\rho - \rho^2$;
12: $R \leftarrow R \cdot S$;
13: **end for**;
14: **return** R;

Algoritmo 5.1: Emparelhamento η_T

Característica	Valor Usado
Corpo	F_3^m , onde m é primo relativo à 6 e m é 97
Curva	E: $y^2 = x^3 - x + 1$, $b = 1$
Polinômio irredutível de grau m	$f(x) = x^{97} + x^{12} + 2$
Número de pontos da curva	1908805632340782707542448628761560269267064896 3
Grau de mergulho	$k = 6$
ℓ	272686518905826101077496079813497618717146272 1
Mapa de distorção	$\psi : E(F_3^{97})[\ell] \rightarrow E(F_3^{6*97})[\ell] \setminus E(F_3^{97})[\ell]$ (x, y) α $(\rho - x, y\sigma)$ com $\rho \in F_3^{3*97}$ e $\sigma \in F_3^{2*97}$ satisfazendo $\rho^3 = \rho + 1$ e $\sigma^2 = -1$
Tower field	$F_3^{6*97} = F_3^{97}[\rho, \square] \cong F_3^{97}[X, Y] / (X^3 - X - b, Y^2 + 1)$
Exponenciação final	$M = (3^{3*97} - 1) \cdot (3^{97} + 1) \cdot (3^{97} + 1 - \mu 3^{(97+1)/2})$
Parâmetros	λ e $v = -1$

Tabela 5.1: Características Criptográficas

Diferentes corpos podem ser utilizados na criptografia, os mais comuns são: F_p, F_2^m e F_3^m. Em nosso trabalho escolhemos o campo F_3^m, com $m = 97$. Como apresentado em [11], uma possível correspondência em termos de grau de segurança entre o corpo escolhido e os outros corpos seria:

$$\begin{aligned} q = 3^m &: m = 97,\ k = 6 &&\rightarrow \log_2(q^k) \sim 922 \\ q = 2^m &: m = 233,\ k = 4 &&\rightarrow \log_2(q^k) \sim 932 \\ q = p &: \log_2(p) \sim 160,\ k = 6 &&\rightarrow \log_2(q^k) \sim 960 \end{aligned}$$

Esses valores são ligeiramente menores que os níveis de segurança tipicamente aceitos (1024 bits do RSA), contudo, poderíamos futuramente expandir a segurança do trabalho utilizando um dos níveis de segurança como: F_3^{163}, F_3^{193}, F_3^{239}, F_3^{353} e F_3^{509}.

Como podemos observar no Algoritmo 5.1, um emparelhamento η_T é composto por diferentes sub-operações, como multiplicação em F_3^{97}, elevação ao cubo em F_3^{97}, etc. Detalharemos a seguir todas as funções necessárias para realizar a implementação do emparelhamento η_T.

5.2.1 Soma, Subtração e Negação de Elementos em F_3

Cada elemento pertencente a F_3 é representado por 2 bits, o bit *high* (H) e o bit *low* (L). As operações de adição, subtração e negação sobre esses elementos podem ser eficientemente

implementadas em hardware, utilizando pequenos circuitos combinacionais. O *gate delay* desses circuitos é baixo, possibilitando que a latência destas operações em hardware seja desprezível. Contudo, em software não é possível trabalhar com bits de forma eficiente, então cada elemento em F_3 é mapeado na menor palavra que a arquitetura comporta, no nosso caso o byte (isso em uma implementação não otimizada em software). A soma de elementos em F_3 é eficientemente calculada através da seguinte expressão:

$$R_H \leftarrow (a_L \vee b_L) \oplus ((a_L \vee b_H) \oplus (a_H \vee b_L))$$
$$R_L \leftarrow (a_H \vee b_H) \oplus ((a_L \vee b_H) \oplus (a_H \vee b_L))$$

Onde *a* e *b* são os elementos de entrada em F_3 e *R* é o resultado da soma ($R_{HL} = a_{HL} + b_{HL}$). Todos formados pelos bits H e L.

A negação de um elemento em F_3 pode ser realizada apenas invertendo-se os bits H e L que compõem o elemento. Dada essa premissa, a operação de subtração de elementos pode ser realizada utilizando-se a função de soma de elementos em F_3, ou seja, basta apenas realizar a inversão do segundo elemento antes de realizar a operação de soma: $R_{HL} = a_{HL} + b_{LH}$.

5.2.2 Multiplicação de Elementos em F_3

A multiplicação de elementos em F_3 é eficientemente calculada através da seguinte expressão:

$$R_H \leftarrow (a_L \wedge b_H) \vee (a_H \wedge b_L)$$
$$R_L \leftarrow (a_L \wedge b_L) \vee (a_H \wedge b_H)$$

Onde *a* e *b* são os elemento de entrada em F_3 e *R* é o resultado da multiplicação ($R_{HL} = a_{HL} \cdot b_{HL}$), todos formados pelos bits H e L. No capítulo 6 veremos essa operação detalhadamente.

5.2.3 Multiplicação de Elementos em F_3^{97}

Em nosso trabalho, usamos uma modificação do algoritmo MSE (*Most-Significant Element*) de multiplicação em F_3^{97} proposto em [5]. Basicamente o algoritmo recebe como entrada dois elementos em F_3^{97}, calcula a multiplicação entre os dois, reduz o resultado pelo polinômio irredutível de grau *m* e retorna como resultado um elemento em F_3^{97}. Esse esquema pode ser observado no Algoritmo 5.2.

Multiplicação de Elementos em F_3^{97}

Entrada: $a(x)$ and $b(x) \in F_3^{97}$.
Saída: $p(x) = a(x).\ b(x)$ **mod** $f(x)$.
1: $p(x) \leftarrow 0$;
2: **for** $i \leftarrow \lceil 97 / D \rceil$ - 1 **downto** 1 **do**
3: $s(x) \leftarrow \sum_{j=0}^{D-1} \left(a_{Di+j}.\ b(x)\ .\ x^j \right)$;
4: $p(x) \leftarrow (s(x) + (p(x)\ .\ x^D))$ **mod** $f(x)$;
5: **end for**;
6: **return** $p(x)$;

Algoritmo 5.2: Multiplicação de Elementos em F_3^{97}

O algoritmo apresentado é executado 32 vezes para o caso de $D = 3$, onde D é um parâmetro que define o número de coeficientes de $a(x)$ que serão processados a cada ciclo de *clock* (ele é chamado de MSE, pois começa multiplicando os coeficientes de mais alta ordem para os de menor ordem). Ele também chama uma subfunção denominada *mod* $f(x)$, que realiza a redução pelo polinômio irredutível de grau m.

O algoritmo apresentado realiza um número menor de chamadas à subfunção de redução pelo polinômio irredutível de grau m, quando comparado ao algoritmo apresentado em [5] (32 vezes contra 66 vezes). Contudo note que o polinômio $s(x)$ possui o grau um pouco maior ($m + D - 2$).

Um melhor entendimento desse algoritmo poderá ser encontrado no capítulo 7, onde descreveremos o funcionamento de uma multiplicação.

5.2.4 Redução pelo Polinômio Irredutível de Grau *m*

Como o nosso polinômio irredutível de grau m é um trinômio ($x^{97} + x^{12} + 2$), exploramos o fato de $x^{97} \equiv -(x^{12} + 2)$ *mod* $f(x)$ na simplificação da operação, tornando assim a operação de redução simples e rápida.

O Algoritmo 5.3 implementa a execução dessa operação. Ele basicamente remove do polinômio de entrada $x^{12} + 2$ para cada elemento que estiver acima do grau 97 (no algoritmo consideramos o elemento com o grau de entrada de $97 + D - 1$), respeitando o grau de cada elemento com relação a 97. Um melhor entendimento desse algoritmo poderá ser encontrado no capítulo 6, onde descreveremos a implementação desse algoritmo em hardware.

Redução pelo Polinômio Irredutível de Grau *m*

Entrada: $a(x) \in F_3^{97+D-1}$.
Saída: $p(x) = a(x)$ **mod** $f(x)$.
1: $i \leftarrow 97 + D - 1$;
2: **while** $i >= 97$
3: **if** $a(i) \neq 0$
4: $p(12 + (i - 97)) \leftarrow a(12 + (i - 97)) - a(i)$;
5: $p(i - 97) \leftarrow a(i - 97) + a(i)$;
6: $p(i) \leftarrow 0$;
7: **end if**;
8: $i \leftarrow i - 1$;
9: **end while**;
10: **return** *p(x);*

Algoritmo 5.3: Redução pelo Polinômio Irredutível de Grau *m*

5.2.5 Multiplicação de Elementos em F_3^{6*97}

Gorla *et al* apresentam em [41] um esquema de multiplicação de elementos em F_3^{6*97} onde a multiplicação pode ser representada como cinco multiplicações em F_3^{2*97} e o esquema de *Karatsuba-Ofman* é utilizado para computar cada multiplicação em F_3^{2*97} através de três multiplicações em F_3^{97}. Assim, para realizar uma multiplicação de dois elementos em F_3^{6*97}, precisamos de 15 multiplicações em F_3^{97}. A implementação desse esquema pode ser observada no Algoritmo 5.4.

Multiplicação de Elementos em F_3^{6*97}

Entrada: U e V $\in F_3^{6*97}$ onde U = $u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho^2 + u_5\sigma\rho^2$ e V = $v_0 + v_1\sigma + v_2\rho + v_3\sigma\rho + v_4\rho^2 + v_5\sigma\rho^2$.
Saída: W = U · V $\in F_3^{6*97}$.
1: $r_0 \leftarrow u_0 + u_4$; $a_0 \leftarrow r_0 + u_2$; $a_{12} \leftarrow r_0 - u_2$;
2: $r_0 \leftarrow v_0 + v_4$; $a_3 \leftarrow r_0 + v_2$; $a_{15} \leftarrow r_0 - v_2$;
3: $r_0 \leftarrow u_0 - u_4$; $a_6 \leftarrow r_0 - u_3$; $a_{18} \leftarrow r_0 + u_3$;
4: $r_0 \leftarrow v_0 - v_4$; $a_9 \leftarrow r_0 - v_3$; $a_{21} \leftarrow r_0 + v_3$;
5: $r_0 \leftarrow u_1 + u_5$; $a_1 \leftarrow r_0 + u_3$; $a_{13} \leftarrow r_0 - u_3$;
6: $r_0 \leftarrow v_1 + v_5$; $a_4 \leftarrow r_0 + v_3$; $a_{16} \leftarrow r_0 - v_3$;
7: $r_0 \leftarrow u_1 - u_5$; $a_7 \leftarrow r_0 + u_2$; $a_{19} \leftarrow r_0 - u_2$;
8: $r_0 \leftarrow v_1 - v_5$; $a_{10} \leftarrow r_0 + v_2$; $a_{22} \leftarrow r_0 - v_2$;
9: $a_2 \leftarrow a_0 + a_1$; $a_5 \leftarrow a_3 + a_4$; a_8 $a_6 + a_7$;
10: $a_{11} \leftarrow a_9 + a_{10}$; $a_{14} \leftarrow a_{12} + a_{13}$; $a_{17} \leftarrow a_{15} + a_{16}$;
11: $a_{20} \leftarrow a_{18} + a_{19}$; $a_{23} \leftarrow a_{21} + a_{22}$;
12: $a_{24} \leftarrow u_4 + u_5$; $a_{25} \leftarrow v_4 + v_5$;
13: $m_0 \leftarrow a_0 \cdot a_3$; $m_1 \leftarrow a_2 \cdot a_5$; $m_2 \leftarrow a_1 \cdot a_4$;
14: $m_3 \leftarrow a_6 \cdot a_9$; $m_4 \leftarrow a_8 \cdot a_{11}$; $m_5 \leftarrow a_7 \cdot a_{10}$;
15: $m_6 \leftarrow a_{12} \cdot a_{15}$; $m_7 \leftarrow a_{14} \cdot a_{17}$; $m_8 \leftarrow a_{13} \cdot a_{16}$;
16: $m_9 \leftarrow a_{18} \cdot a_{21}$; $m_{10} \leftarrow a_{20} \cdot a_{23}$; $m_{11} \leftarrow a_{19} \cdot a_{22}$;
17: $m_{12} \leftarrow u_4 \cdot v_4$; $m_{13} \leftarrow a_{24} \cdot a_{25}$; $m_{14} \leftarrow u_5 \cdot v_5$;
18: $t_0 \leftarrow m_0 + m_4 + m_{12}$; $t_1 \leftarrow m_2 + m_{10} + m_{14}$;
19: $t_2 \leftarrow m_6 + m_{12}$; $t_3 \leftarrow -m_8 - m_{14}$;
20: $t_4 \leftarrow m_7 + m_{13}$; $t_5 \leftarrow t_3 + m_2$;
21: $t_6 \leftarrow t_2 - m_0$; $t_7 \leftarrow t_3 - m_2 + m_5 + m_{11}$;
22: $t_8 \leftarrow t_2 + m_0 - m_3 - m_9$;
23: $w_0 \leftarrow -t_0 + t_1 - m_3 + m_{11}$;
24: $w_1 \leftarrow t_0 + t_1 - m_1 + m_5 + m_9 - m_{13}$;
25: $w_2 \leftarrow t_5 + t_6$;
26: $w_3 \leftarrow t_5 - t_6 + t_4 - m_1$;
27: $w_4 \leftarrow t_7 + t_8$;
28: $w_5 \leftarrow t_7 - t_8 + t_4 + m_1 - m_4 - m_{10}$;
29: **return** $w_0 + w_1\sigma + w_2\rho + w_3\sigma\rho + w_4\rho^2 + w_5\sigma\rho^2$;

Algoritmo 5.4: Multiplicação de Elementos em F_3^{6*97}

5.2.6 Elevação ao Cubo de um Elemento em F_3^{97}

Devido ao fato de estarmos trabalhando em característica 3, a operação de elevação ao cubo poder ser simplificada (qualquer valor multiplicado por 3 é zero em F_3), o que nos leva a seguinte expressão para o cálculo da elevação ao cubo de um elemento em F_3^{97}:

$$\text{Se } d(x) = \sum_{i=0}^{m-1} d_i x^i \text{ então } d(x)^3 \bmod f(x) \equiv \sum_{i=0}^{m-1} d_i x^{3i} \bmod f(x).$$

Como exemplo vamos realizar a elevação ao cubo do polinômio $x + 1$:

$$(x+1)^3 = x^3 + 3x^2 + 3x + 1 = x^3 + 1$$

Basicamente, essa função recebe como entrada um elemento em F_3^{97}, computa sua elevação ao cubo, reduz pelo polinômio irredutível de grau m e retorna outro elemento em F_3^{97} como resultado. O algoritmo que foi implementado pode ser obtido a partir da aplicação da técnica de *loop-unrolling* na expressão apresentada.

O cálculo de cada elemento do polinômio resultante é obtido através da realização de algumas somas (em F_3) e permutações dos coeficientes do polinômio de entrada, o que pode ser realizado em paralelo (a multiplicação de um elemento em F_3 por dois, pode ser realizada apenas invertendo a posição dos dois bits que compõem cada elemento em F_3, como já foi dito). Note que essa operação poderia ser classificada como "divisível", como definido em [26]. Contudo, para manter a simplicidade da implementação, a operação de elevação ao cubo foi tratada como "não divisível". O resultado da aplicação da técnica de *loop-unrolling* pode ser observado no Algoritmo 5.5. No capítulo 6 apresentaremos o resultado completo da operação de *loop-unrolling*.

Elevação ao Cubo de um Elemento em F_3^{97}

Entrada: $d(x)$ □ F_3^{97}.
Saída: $d(x)^3$ mod $f(x)$.
1: Result (0) ← d(0) + d(89) + d(93);
2: Result (1) ← d(65) + 2 · d(61);
3: Result (2) ← d(33);
4: Result (3) ← d(1) + d(90) + d(94);
5: Result (4) ← d(66) + 2 · d(62);
6: Result (5) ← d(34);
...
92: Result (91) ← d(87) + d(91) + d(95);
93: Result (92) ← d(63) + 2 · d(59);
94: Result (93) ← d(31);
95: Result (94) ← d(88) + d(92) + d(96);
96: Result (95) ← d(64) + 2 · d(60);
97: Result (96) ← d(32);
98: **return** Result;

Algoritmo 5.5: Elevação ao Cubo de um Elemento em F_3^{97}

5.2.7 Elevação ao Cubo de um Elemento em F_3^{6*97}

Como definido em [5] a elevação ao cubo de um elemento em F_3^{6*97} pode ser realizada da seguinte maneira: Defina $U = u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho^2 + u_5\sigma\rho^2 \in F_3^{6*97}$, $U^3 \in F_3^{6*97}$ é dado pela seguinte expressão: $U^3 = {u_0}^3 + {u_1}^3\sigma^3 + {u_2}^3\rho^3 + {u_3}^3(\sigma\rho)^3 + {u_4}^3(\rho^2)^3 + {u_5}^3(\sigma\rho^2)^3$. Onde:

$$\rho^3 = \rho + b;$$
$$(\rho^2)^3 = \rho^2 + b\rho + 1;$$
$$\sigma^3 = -\sigma,$$
$$(\sigma\rho)^3 = -\sigma\rho - b\sigma,$$
$$(\sigma\rho^2)^3 = -\sigma\rho^2 - b\sigma\rho - \sigma.$$

Simplificando as expressões chegamos aos seguintes coeficientes para $V = U^3$:

$$v_0 = {u_0}^3 + b{u_2}^3 + {u_4}^3;$$
$$v_1 = -{u_1}^3 - b{u_3}^3 - {u_5}^3;$$
$$v_2 = {u_2}^3 - b{u_4}^3;$$
$$v_3 = -{u_3}^3 + b{u_5}^3;$$
$$v_4 = {u_4}^3;$$
$$v_5 = -{u_5}^3.$$

5.3 Exponenciação Final

O emparelhamento η_T deve ser reduzido em ordem de grandeza para ser unicamente definido, não apenas elevado à ℓ potência. Essa redução é realizada através da exponenciação final, na qual o emparelhamento $\eta_T(P, Q)$ é elevado à potência de M, onde:

$$M = (3^{3*97} - 1) \cdot (3^{97} + 1) \cdot (3^{97} + 1 - \mu 3^{(97+1)/2}).$$

A Figura 5.6 apresenta a implementação da exponenciação final. Esse algoritmo recebe como entrada $U \in F^*_{3^{6*97}}$. Primeiramente o algoritmo calcula $U^{(3^{3\square 97}-1)}$ através do Algoritmo 5.7, então é chamado o Algoritmo 5.11 para obtermos $U^{(3^{3\square 97}-1)(3^{97}+1)}$.

Através de sucessivas elevações ao cubo obtemos $W = U^{(3^{3\square 97}-1)(3^{97}+1)3^{(97+1)/2}}$, então é chamado novamente o Algoritmo 5.11, obtendo $V = U^{(3^{3\square 97}-1)(3^{97}+1)(3^{97}+1)}$. Por fim é realizada a multiplicação de W por V. Algumas operações especiais são necessárias para o cálculo da exponenciação final. Essas operações serão discutidas a seguir.

Exponenciação Final

Entrada: $U = u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho^2 + u_5\sigma\rho^2 \in F_{3^{6*97}}$.
Saída: $U^M \in T_2(F_{3^{3*97}}) \subset F^*_{3^{6*97}}$.
1: $V \leftarrow U^{3^{3\square 97}-1}$;
2: $V \leftarrow U^{3^{97}+1}$;
3: $W \leftarrow V$;
4: **for** $i \leftarrow 1$ **to** 49 **do**
5: $W \leftarrow W^3$;
6: **end for;**
7: $V \leftarrow U^{3^{97}+1}$;
8: **return** $V \cdot W$;

Algoritmo 5.6: Exponenciação Final

5.3.1 Cálculo de $U^{3^{3\square 97}-1}$

Na exponenciação final, tomamos $U = \eta_T(P, Q) \in F^*_{3^{6*97}}$, inicialmente é calculado $U^{3^{3\square 97}-1}$. Escrevendo U como $U_0 + U_1\sigma$, onde U_0 e $U_1 \in F^*_{3^{3*97}}$ e dados que [5]:

$$U^{3^{3\square 97}-1} = U_0 - U_1\sigma;$$
$$U^{-1} = \frac{(U_0^2 - U_1^2) + U_0U_1\sigma}{U_0^2 + U_1^2}$$

É obtida a seguinte expressão para $U^{3^{3\square 97}-1}$:

$$U^{3^{3\square 97}-1} = \frac{(U_0^2 - U_1^2) + U_0U_1\sigma}{U_0^2 + U_1^2}$$

Como definido em [5], $U^{3^{3\square 97}-1}$ é de fato um elemento em $T_2(F_3{}^{3*97})$, onde $T_2(F_3{}^{3*97}) = \{X_0 + X_1\sigma \in F^*_{3^{6*97}} : X_0^2 + X_1^2 = 1\}$ é um *torus* como o introduzido por Granger *et al* para o caso do emparelhamento *Tate* em [32]. Esse é um ponto importante, pois a aritmética em no *torus* $T_2(F_3{}^{3*97})$ é muito mais simples que a aritmética em $F^*_{3^{6*97}}$. O Algoritmo 5.7 implementa a expressão $U^{3^{3\square 97}-1}$. Observe que duas novas operações (elevação de elementos ao quadrado em $F_3{}^{3*97}$ e inversão de elementos em $F_3{}^{3*97}$) são necessárias para implementação de tal algoritmo.

Cálculo de $U^{3^{3\square 97}-1}$

Entrada: $U = u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho^2 + u_5\sigma\rho^2 \in F^*_{3^{6*97}}$.
Saída: $V = U^{3^{3\square 97}-1} \in T_2(F_3{}^{3*97})$.
1: $m_0 \leftarrow (u_0 + u_2\rho + u_4\rho^2)^2$;
2: $m_1 \leftarrow (u_1 + u_3\rho + u_5\rho^2)^2$;
3: $m_2 \leftarrow (u_0 + u_2\rho + u_4\rho^2) \cdot (u_1 + u_3\rho + u_5\rho^2)$;
4: $a_0 \leftarrow m_0 - m_1$; $a_1 \leftarrow m_0 + m_1$;
5: $i \leftarrow a_1^{-1}$;
6: $V_0 \leftarrow a_0 \cdot i$;
7: $V_1 \leftarrow m_2 \cdot i$;
8: **return** $V_0 + V_1\sigma$,

Algoritmo 5.7: Cálculo de $U^{3^{3\square 97}-1}$

5.3.2 Elevação de Elementos ao Quadrado em F_3^{3*97}

Defina $U = u_0 + u_1\rho + u_2\rho^2 \in F_3^{3*97}$, com $u_i \in F_3^{97}$, $0 \leq i \leq 2$. $V = U^2$ é dado por:

- $v_0 = u_0^2 - bu_1u_2$;
- $v_1 = bu_2^2 - u_0u_1 - u_1u_2$;
- $v_2 = (u_0 + u_1) \cdot (u_0 + u_1 + u_2) - u_0^2 + u_0u_1 + u_1u_2$.

A implementação de tais expressões pode ser observada no Algoritmo 5.8.

Elevação de Elementos ao Quadrado em F_3^{3*97}

Entrada: $U = u_0 + u_1\rho + u_2\rho^2 \in F_3^{3*97}$.
Saída: $V = U^2 \in F_3^{3*97}$.
1: $a_0 \leftarrow u_0 + u_1$; $a_1 \leftarrow a_0 + u_2$;
2: $m_0 \leftarrow u_0^2$; $m_1 \leftarrow u_0 \cdot u_1$; $m_2 \leftarrow u_1 \cdot u_2$;
3: $m_3 \leftarrow u_2^2$; $m_4 \leftarrow a_1^2$;
4: $a_2 \leftarrow m_1 + m_2$;
5: $v_0 \leftarrow m_0 - bm_2$;
6: $v_1 \leftarrow bm_3 - a_2$;
7: $v_2 \leftarrow m_4 + a_2 - m_0$;
8: **return** $v_0 + v_1\rho + v_2\rho^2$;

Algoritmo 5.8: Elevação de Elementos ao Quadrado em F_3^{3*97}

5.3.3 Inversão de Elementos em F_3^{3*97}

Tome $V = v_0 + v_1\rho + v_2\rho^2 \in F_3^{3*97}$ como sendo o inverso multiplicativo de $U = u_0 + u_1\rho + u_2\rho^2 \in F_3^{3*97}$, sendo $U \neq 0$, onde $u_i, v_i \in F_3^{97}$, $0 \leq i \leq 2$. Dado que $V \cdot U = 1$, obtemos:

$$u_0v_0 + bu_2v_1 + bu_1v_2 = 1,$$
$$u_1v_0 + (u_0 + u_2)v_1 + (u_1 + bu_2)v_2 = 0,$$
$$u_2v_0 + u_1v_1 + (u_0 + u_2)v_2 = 0.$$

A solução desse sistema de equações é dada por:

$$\begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = w^{-1} \begin{bmatrix} u_0^{\,2} - \left(u_1^{\,2} - u_2^{\,2}\right) - u_2\left(u_0 + bu_1\right) \\ bu_2^{\,2} - u_0u_1 \\ u_1^{\,2} - u_2^{\,2} - u_0u_2 \end{bmatrix}$$

Onde $w = u_0^{\,2}(u_0 - u_2) + u_1^{\,2}(-u_0 + bu_1) + u_2^{\,2}(-(-u_0 + bu_1) + u_2) \in F_3{}^{97}$. A implementação dessa expressão é apresentada no Algoritmo 5.9. Observe para a implementação desse algoritmo é necessária a operação de inversão de elementos em $F_3{}^{97}$.

Inversão de Elementos em $F_3{}^{3*97}$

Entrada: $U = u_0 + u_1\rho + u_2\rho^2 \in F_3{}^{3*97}$.
Saída: $V = U^{-1} \in F_3{}^{3*97}$.
1: $a_0 \leftarrow u_0 + bu_1$; $a_1 \leftarrow u_0 - u_2$;
2: $a_2 \leftarrow -u_0 + u_1$; $a_3 \leftarrow -a_2 + u_2$;
3: $m_0 \leftarrow u_0^{\,2}$ $m_1 \leftarrow u_1^{\,2}$ $m_2 \leftarrow u_2^{\,2}$;
4: $m_3 \leftarrow u_0 \cdot u_1$; $m_4 \leftarrow u_0 \cdot u_2$; $m_5 \leftarrow u_2 \cdot a_0$;
5: $m_6 \leftarrow m_0 \cdot a_1$; $m_7 \leftarrow m_1 \cdot a_2$; $m_8 \leftarrow m_2 \cdot a_3$;
6: $w \leftarrow m_6 + m_7 + m8$;
7: $i \leftarrow w^{-1}$;
8: $a_4 \leftarrow m_1 - m_2$; $a_5 \leftarrow -a_4 + m_0 - m_5$;
9: $a_6 \leftarrow bm_2 - m_3$; $a_7 \leftarrow a_4 - m_4$;
10: $v_0 \leftarrow i \cdot a_5$; $v_1 \leftarrow i \cdot a_6$; $v_2 \leftarrow i \cdot a_7$;
11: **return** $v_0 + v_1\rho + v_2\rho^2$;

Algoritmo 5.9: Inversão de Elementos em $F_3{}^{3*97}$

5.3.4 Inversão de Elementos em $F_3{}^{97}$

A exponenciação final envolve uma inversão de elementos em $F_3{}^{97}$. A inversão de elementos pode ser realizada através de operações de multiplicações e elevações ao cubo em $F_3{}^{97}$. A inversão de elementos $F_3{}^{97}$ é definida da seguinte forma: defina um elemento $d \in F_3{}^{97}$, tal que $d \neq 0$. Inicialmente elevamos d a potências de 3 para obter $r = d^{(3^{(97-1)}-1)/2}$. Realizando a elevação de r ao cubo e posteriormente multiplicando por o resultado por d, encontramos $u = d^{(3^{97}-3)/2}$ e $v = d^{(3^{97}-1)/2}$. Por fim realizamos a multiplicação $u \cdot v = d^{(3^{97}-3)/2} \cdot d^{(3^{97}-1)/2} = d^{(3^{97}-2)} = d^{-1}$. A implementação dessa expressão é apresentada no Algoritmo 5.10.

Inversão de Elementos em F_3^{97}

Entrada: $d \in F_3^{97}$.
Saída: $d^{-1} \in F_3^{97}$.
1: $r \leftarrow d^{(3^{(97-1)}-1)/2}$;
2: $u \leftarrow r^3$;
3: $v \leftarrow u \cdot d$;
4: **return** $u \cdot v$;

Algoritmo 5.10: Inversão de Elementos em F_3^{97}

Como apresentado em [33] e [34], cadeias de adição se mostram excelentes na elevação de elementos em F_3^m a certas potências, como a potência $3^{(97-1)}-1$, necessária em nosso algoritmo de inversão de elementos em F_3^{97}. A seguir será apresentada a cadeia de adição Brauer-type, que pode ser utilizada para o cálculo da potência $d^{(3^{(97-1)}-1)/2}$.

Como apresentado em [5], um cadeia de adição Brauer-type C de tamanho l é uma seqüência de l inteiros $S = (j_1, ..., j_l)$ tal que $0 \leq j_i < i$, para todo $1 \leq i \leq l$. Podemos então construir uma outra seqüência $(n_0, ..., n_l)$ que satisfaça:

$$n_0 = 1;$$
$$n_i = n_{i-1} + n_{j_i}, \text{ para todo } 1 \leq i \leq l.$$

Uma propriedade adicional pode ser obtida em [35], para todo $1 \leq l' \leq l$:

$$\sum_{i=1}^{l'} n_{j_i} = n_{j'} - 1$$

Além disso, é possível observar que para todo $n \leq n'$:

$$d^{(3^{n+n'}-1)/2} = d^{(3^n-1)/2} \cdot (d^{(3^{n'}-1)/2})^{3^n}$$

Conseqüentemente, dada uma cadeia de adição Brauer-type C de tamanho l para m-1, é possível calcular a expressão requerida $d^{(3^{(97-1)}-1)/2}$ como apresentado no Algoritmo 5.11. Esse algoritmo garante que para cada iteração i, temos $z_i = d^{(3^{n_i}-1)/2}$, onde $(n_0, ..., n_l)$ é a seqüência de inteiros associada com a cadeia de adição C. Com $n_l = m - 1$, são necessárias l multiplicações em F_3^{97} e m–2 elevações ao cubo em F_3^{97}. Para nossa configuração $l = 7$.

Cálculo de $d^{(3^{(97-1)}-1)/2}$

Entrada: $d \in \mathrm{F}_3{}^{97}$.
Saída: $d^{-1} \in \mathrm{F}_3{}^{97}$.
1: $z_0 \leftarrow d$;
2: **for** $i \leftarrow 1$ **to** l **do**
3: $z_i \leftarrow z_{j_i} \cdot z_{i-1}^{3^{n_{j_i}}}$;
4: **end for;**
5: **return** z_l;

Algoritmo 5.11: Cálculo de $d^{(3^{(97-1)}-1)/2}$

5.3.5 Cálculo de $\mathrm{U}^{3^{97}+1}$

A implementação de $\mathrm{U}^{3^{97}+1}$ pode ser observada no Algoritmo 5.12.

Cálculo de $\mathrm{U}^{3^{97}+1}$

Entrada: $\mathrm{U} = u_0 + u_1\sigma + u_2\rho + u_3\sigma\rho + u_4\rho^2 + u_5\sigma\rho^2 \in \mathrm{F}^*{}_3{}^{6*97}$.
Saída: $\mathrm{V} = \mathrm{U}^{3^{97}+1} \in T_2(\mathrm{F}_3{}^{3*97})$.
1: $a_0 \leftarrow u_0 + u_1$; $a_1 \leftarrow u_2 + u_3$; $a_2 \leftarrow u_4 - u_5$;
2: $\mathrm{m}_0 \leftarrow u_0 \cdot u_4$; $\mathrm{m}_1 \leftarrow u_1 \cdot u_5$; $\mathrm{m}_2 \leftarrow u_2 \cdot u_4$;
3: $\mathrm{m}_3 \leftarrow u_3 \cdot u_5$; $\mathrm{m}_4 \leftarrow a_0 \cdot a_2$; $\mathrm{m}_5 \leftarrow u_1 \cdot u_2$;
4: $\mathrm{m}_6 \leftarrow u_0 \cdot u_3$; $\mathrm{m}_7 \leftarrow a_0 \cdot a_1$; $\mathrm{m}_8 \leftarrow a_1 \cdot a_2$;
5: $a_3 \leftarrow \mathrm{m}_5 + \mathrm{m}_6 - \mathrm{m}_7$; $a_4 \leftarrow -\mathrm{m}_2 - \mathrm{m}_3$;
6: $a_5 \leftarrow -\mathrm{m}_2 + \mathrm{m}_3$; $a_6 \leftarrow -\mathrm{m}_0 + \mathrm{m}_1 + \mathrm{m}_4$;
7: $v_0 \leftarrow 1 + \mathrm{m}_0 + \mathrm{m}_1 + ba_4$;
8: $v_1 \leftarrow b\mathrm{m}_5 - b\mathrm{m}_6 + a_6$;
9: $v_2 \leftarrow -a_3 + a_4$;
10: $v_3 \leftarrow \mathrm{m}_8 + a_5 - ba_6$;
11: $v_4 \leftarrow -ba_3 - ba_4$;
12: $v_5 \leftarrow b\mathrm{m}_8 + ba_5$;
13: **end if**
14: **return** $v_0 + v_1\sigma + v_2\rho + v_3\sigma\rho + v_4\rho^2 + v_5\sigma\rho^2$;

Algoritmo 5.12: Cálculo de $\mathrm{U}^{3^{97}+1}$

5.3.1 Multiplicação de Elementos em F_3^{3*97}

Como apresentado em [5], a multiplicação de elementos em F_3^{3*97} pode ser construída através de multiplicações em F_3^{97}. Assuma que U e V $\in F_3^{3*97}$ e são definidos da seguinte forma: U = $u_0 + u_1\rho + u_2\rho^2$ e V = $v_0 + v_1\rho + v_2\rho^2$, onde u_i, $v_i \in F_3^{97}$, $0 \leq i \leq 2$. O produto W = U · V é dado por:

$$w_0 = b(bu_1 + u_2)(v_1 + bv_2) + u_0v_0 - u_1v_1 - u_2v_2;$$

$$w_1 = (u_0 + u_1)(v_0 + v1) + (bu_1 + u_2)(v_1 + bv_2) - u_0v_0 - (b + 1)\, u_1v_1;$$

$$w_2 = (u_0 + u_2)(v_0 + v_2) - u_0v_0 + u_1v1.$$

É possível observar que os elementos w_0, w_1 e $w_2 \in F_3^{97}$ e podem ser calculados em paralelo. O Algoritmo 5.13 implementa a multiplicação de elementos em F_3^{3*97}.

Multiplicação de Elementos em F_3^{3*97}

Entrada: U = $u_0 + u_1\rho + u_2\rho^2$ e V = $v_0 + v_1\rho + v_2\rho^2$.
Saída: W = U · V $\in F_3^{3*97}$.
1: $a_0 \leftarrow u_0 + u_1$; $a_1 \leftarrow u_0 + u_2$; $a_2 \leftarrow bu_1 + u_2$;
2: $a_3 \leftarrow v_0 + v_1$; $a_4 \leftarrow v_0 + v_2$; $a_5 \leftarrow v_1 + bv_2$;
3: $m_0 \leftarrow u_0 \cdot v_0$; $m_1 \leftarrow u_1 \cdot v_1$; $m_2 \leftarrow u_2 \cdot v_2$;
4: $m_3 \leftarrow a_0 \cdot a_3$; $m_4 \leftarrow a_1 \cdot a_4$; $m_5 \leftarrow a_2 \cdot a_5$;
5: $a_6 \leftarrow m_0 - m_1$;
6: $w_0 \leftarrow a_6 - m_2 + bm_5$
7: $w_1 \leftarrow -a_6 + m_3 + m_5$;
8: $w_2 \leftarrow -a_6 + m_4$;
9: **return** $w_0 + w_1\rho + w_2\rho^2$;

Algoritmo 5.13: Multiplicação de Elementos em F_3^{3*97}

5.4 Validação das Operações

Diferentes técnicas foram empregadas para verificar a corretude do sistema. Inicialmente pequenas porções do sistema foram testadas individualmente, e posteriormente o sistema como um todo foi testado. Algumas operações que compõem o emparelhamento podem ter sua corretude comprovada através de suas propriedades matemáticas básicas. Veremos agora alguns exemplos:

Multiplicação de Elementos

As propriedade básicas da multiplicação de elementos foram utilizadas para testar as implementações das funções de multiplicação desenvolvidas. Polinômios aleatoriamente escolhidos foram utilizados em todos os testes realizados. As seguintes propriedades foram testadas nas operações de multiplicação de elementos:

- Propriedade associativa. $\forall a, b \text{ e } c \in \mathrm{F}_3{}^m : a \cdot (b \cdot c) = (a \cdot b) \cdot c$;
- Propriedade distributiva. $\forall a, b \text{ e } c \in \mathrm{F}_3{}^m : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Elevação de Elementos ao Cubo

Após ter validado a operação de multiplicação, foi possível utilizá-la para testar a corretude de outras operações, como as operações de exponenciação de elementos. Para tanto, polinômios aleatoriamente escolhidos foram utilizados para testar a seguinte propriedade:

$$\forall\ a \in \mathrm{F}_3{}^m : a \cdot a \cdot a = a^3 \bmod f(x)$$

Abordagem parecida foi utilizada para testar a operação de elevação ao quadrado de elementos.

Inversão de Elementos

Para testar a operação de inversão de elementos foi utilizada a propriedade de que a multiplicação de um elementos pelo seu inverso multiplicativo tem como resultado o valor 1, ou seja:

$$\forall\ a \in \mathrm{F}_3{}^m : a \cdot a^{-1} \bmod f(x) = 1$$

5.5 Validação do Emparelhamento

Para realizar a validação do emparelhamento, um segundo sistema teve de ser implementado para fornecer uma entrada válida ao algoritmo de emparelhamento. Para um emparelhamento ser válido, temos que fornecer como entrada dois pontos pertencentes à curva escolhida (no caso, $y^2 = x^3 - x + 1$).

O segundo sistema teve que implementar um algoritmo capaz de buscar dois pontos pertencentes à curva. Para garantir que um ponto realmente pertence à curva, implementamos uma função capaz de realizar a operação de raiz quadrada em $F_3{}^m$.

Contudo, não basta apenas pertencer à curva para um ponto ser válido para o cálculo do emparelhamento. Segundo [5] o ponto tem que satisfazer a seguinte propriedade para ser válido: P é um ponto válido se $P \in E(F_q)[\ell]$, ou seja P é definido sobre a curva escolhida e $\ell P = \mathcal{O}$, onde $\mathcal{O}$ é um ponto no infinito.

Para encontrar ℓ foi necessário encontrar o número total de pontos da curva (que é dado por $\#E(F_3{}^m) = 3^m + 1 + \mu b 3^{\frac{m+1}{2}} = 1908805632340782707542448628761560269267064896 3$ pontos) e então fatorá-lo. O número de pontos da curva é constituído pela multiplicação de dois primos ($\#E(F_3{}^m) = n \cdot s$), um pequeno (no caso $n = 7$) e outro muito grande (no caso $s = 272686518905826101077496079813497618717146272 1$). Para realizar tal tarefa foi necessária a implementação do algoritmo de fatoração utilizando uma biblioteca GMP [36] de manipulação de números grandes. Tal biblioteca utiliza o algoritmo de *Miller-Rabin* para determinar se um número é um possível primo ou não.

5.5.1 Encontrando um Ponto da Curva

O Algoritmo 5.14 foi utilizado para encontrar um ponto pertencente à curva escolhida, tal que $\ell P = \mathcal{O}$. Encontrar os pontos da curva é necessário para poder testar o emparelhamento utilizando parâmetros validos. O Algoritmo 5.14 utilizada duas operações novas (raiz quadrada de elementos e soma de pontos de uma curva elíptica) que serão apresentas a seguir.

Encontrando um Ponto da Curva

Saída: $P.x$ e $P.y \in F_3^{97}$ tal que P satisfaz a equação da curva $y^2 = x^3 - x + 1$ e $\ell P = \mathcal{O}$.

```
1:  while (1)
2:          P.x ← gera um elemento aleatório em F3^97;
3:          R ← (P.x)^3 – A.x + 1;
4:          if ( √R ) //obs.: esse if verifica se a raiz quadrada existe
5:                  P.y ← √R ;
6:                  Z ← P;
7:                  K ← #ℓ [];
8:                  while K > 0
9:                          K ← K – 1;
10:                         Z ← Z + Z;
11:                         if ℓ [K] = 1
12:                                 Z ← Z + P;
13:                         end if;
14:                 end while;
15:         end if;
16:         if Z = O
17:                 return P;
18;         end if;
19: end while;
```

Algoritmo 5.14: Encontrando um Ponto da Curva

Da linha 7 a 14 podemos observar a implementação do algoritmo inteligente de multiplicação escalar (ou *double-and-add*, como é definido em [18]). Nesse algoritmo $\#\ell$ [] é o número de bits utilizados na representação de ℓ e ℓ [K] equivale ao bit da posição K de ℓ. Nas linhas 10 e 12 o operador "+" equivale à operação de soma de pontos da curva elíptica.

5.5.2 Raiz Quadrada de Elementos em F_3^{97}

A operação de radiciação de elementos foi utilizada para encontrar o elemento y da curva, dado um x aleatório. Conforme apresentado em [18], assumindo a como o resíduo quadrático, então temos que $x^2 = a$ e que $x = a^{(3^{97}+1)/4}$. Para o nosso caso, uma vez que:

$$\frac{3^{97}+1}{4} = \left[6\sum_{i=0}^{47} (9)^i + 1 \right]$$

Temos então:

$$a^{(3^{97}+1)/4} = \left[a^{\sum_{i=0}^{47}(9)^i} \right]^6 \square a$$

5.5.3 Soma de Pontos de uma Curva

A operação de soma de pontos da curva $y^2 = x^3 - x + 1$ foi utilizada para encontrar um elemento P tal que $\ell P = \mathcal{O}$. A implementação deste esquema pode ser observada no Algoritmo 5.15.

Soma de Pontos de uma Curva Elíptica em F_3^{97}

Entrada: $P.x$, $P.y$, $Q.x$ *e* $Q.y \in F_3^{97}$ tal que P e Q satisfazem a equação da curva $y^2 = x^3 - x + 1$.
Saída: $R.x$ *e* $R.y \in F_3^{97}$: R = $P + Q$ e R satisfaz a equação da curva $y^2 = x^3 - x + 1$.

```
1: if P = O
2:          return Q;
3: end if;
4: if Q = O
5:          return P;
6: end if;
7: if P = − Q
8:          return O;
9: end if;
10: if P = Q
11:         λ ← (P.y)^−1     //obs.: É necessária a operação de inversão em F3^97
12:         R.x ← P.x + λ^2; R.y ← (P.y + λ^3);
13:         return R;
14: end if;
15: if P ≠ − Q
16:         λ ← (Q.y − P.y) · (Q.x − P.x)^−1
17:         R.x ← λ^2 − (Q.x +P.x); R.y ← P.y + Q.y − λ^3;
18:         return R;
19: end if;
```

Algoritmo 5.15: Soma de Pontos de uma Curva Elíptica em F_3^{97}

5.5.3 Testando o Emparelhamento

A propriedade de bilinearidade dos emparelhamentos sobre curvas elípticas foi utilizada para validar a implementação do emparelhamento desenvolvida. Basicamente a validação do emparelhamento é realizada através do seguinte teste:

$$\forall\ P, Q \in \mathrm{F}_3{}^{97} \text{ e } \forall\ \mathrm{a} \in \mathrm{Z} \rightarrow \eta_T(cP, Q) = \eta_T(P, Q)^c$$

Para realizar esse teste, inicialmente encontramos através do Algoritmo 5.14 os pontos P e Q pertencentes à curva $y^2 = x^3 - x + 1$, definimos aleatoriamente o valor de uma constante c e realizamos a operação de soma de pontos para obter cP. Então calculamos η_T (cP, Q). Posteriormente a isso calculamos η_T (P, Q) e elevamos o resultado a c, utilizando a multiplicação de elementos em $\mathrm{F}_3{}^{6*97}$. Os valores encontrados serão idênticos, caso o emparelhamento esteja correto.

5.6 Mapeamento da Execução do Emparelhamento

O algoritmo de emparelhamento selecionado foi implementado usando a linguagem de programação C, como visto no capítulo anterior. A execução do emparelhamento em software foi mapeada usando a ferramenta *Oprofile* [25] em um processador *Intel Pentium 4*. O objetivo deste experimento foi identificar o subgrupo das operações que consomem a maior fatia de tempo para serem computadas durante o cálculo de um emparelhamento. Através desse experimento foi possível determinar o melhor subconjunto de operações que devem ser aceleradas em hardware usando uma abordagem HW/SW para o emparelhamento.

Analisando os resultados da execução da ferramenta *Oprofile* (Tabela 5.2) foi possível identificar que as duas operações que possuem maior contribuição no tempo de execução do emparelhamento são as funções de multiplicação e elevação ao cubo em $\mathrm{F}_3{}^{97}$. Essas operações consomem juntas aproximadamente 98% do tempo necessário para computar um emparelhamento sobre curvas elípticas. Tais funções foram escolhidas para serem implementadas em hardware, almejando assim obter um ganho no desempenho do emparelhamento.

Operações do emparelhamento	**Porcentagem do tempo total de execução**
Multiplicação em F_3^{97}	96,3228%
Multiplicação em F_3^{3*97}	0,0057%
Multiplicação em F_3^{6*97}	0,4298%
Elevação ao cubo em F_3^{97}	2,2137%
Elevação ao cubo em F_3^{3*97}	0,1555%
Elevação ao cubo em F_3^{6*97}	0,3442%
Inversão em F_3^{97}	0,0099%
Elevação ao quadrado em F_3^{3*97}	0,0028%
Outras	0,5156%

Tabela 5.2: Mapeamento do Emparelhamento

Capítulo 6

Abordagem Hardware/Software Especializada

Como previamente mencionado, as funções de multiplicação em F_3^{97} e elevação ao cubo em F_3^{97} foram as candidatas selecionadas para serem implementadas em hardware. A linguagem de descrição de hardware escolhida para essa implementação foi a linguagem *VHDL* (*Very High Speed Integrated Circuits Hardware Description Language*). Duas abordagens distintas foram usadas para a implementação, dependendo das características de cada função. A primeira abordagem consiste em implementar a função como uma instrução *customizada* (estendendo assim o conjunto de instruções do processador) no processador Nios II, enquanto que a segunda consiste em implementar a função como um módulo acoplado ao barramento *Avalon* do processador Nios II.

A elevação ao cubo em F_3^{97} foi implementada como uma instrução customizada diretamente no *datapath* do processador Nios II. Essa decisão se deve ao fato de a implementação da função de elevação ao cubo ser um circuito lógico combinacional relativamente pequeno, assim o seu tempo de execução pode ser acomodado no estágio de execução do pipeline do processador, sem impactar o ciclo de *clock*.

Já a multiplicação sobre F_3^{97} foi implementada como um módulo acoplado ao barramento *Avalon* do processador Nios II. Essa decisão se deve ao fato de a implementação da função de multiplicação ser um circuito lógico combinacional relativamente grande (sua execução necessita de múltiplos ciclos de *clock*), assim como sua latência de execução, tornando assim inviável sua implementação como instrução *customizada* dentro do *datapath* do processador.

6.1 Elevação ao Cubo em F_3^{97}

Devido a natureza *soft-core* do processador Nios II, os projetistas podem integrar lógicas extras à unidade lógica aritmética (ULA) do processador, como mostrado na Figura 6.1, criando assim instruções customizadas (ou especializadas).

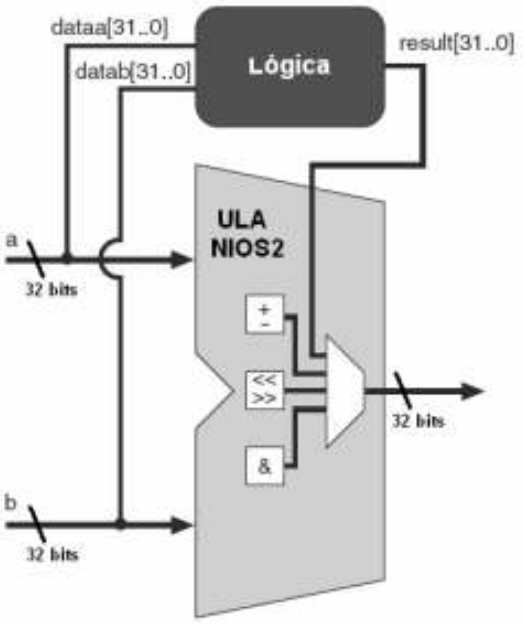

Figura 6.1: Unidade Lógica Aritmética (ULA) do Processador Nios II

Como dito antes, a elevação ao cubo em F_3^{97} foi implementada como uma instrução *customizada* no processador Nios II (existem diferentes tipos de instruções customizadas [20]; o tipo escolhido para nossa implementação foi o estendido). Essa nova instrução recebe como entrada dois operando de 32 bits e um parâmetro de 8 bits (*n*), que são passados a cada chamada da instrução para selecionar qual subfunção da instrução está sendo requisitada. A nova instrução retorna como resultado um elemento de 32 bits, que corresponde a uma fração do elemento resultante.

Como dito anteriormente, em nosso sistema cada elemento em F_3 é representado com dois bits em hardware, ou seja, um elemento em F_3^{97} é representado por 194 bits em hardware, ou seja, 7 palavras de 32 bits.

A interface utilizada para acoplar a instrução *customizada* a ALU do processador Nios II pode ser observada no Algoritmo 6.1. O significado de cada porta do módulo é apresentado na Tabela 6.1

Interface em VHDL da Instrução *Customizada*

```
Entity Instruction is
port(
    signal  clk: in std_logic;
    signal  reset: in std_logic;
    signal  clk_en: in std_logic;
    signal  start: in std_logic;
    signal  done: out std_logic;
    signal  n: in std_logic_vector(7 downto 0);
    signal  dataa: in std_logic_vector(31 downto 0);
    signal  datab: in std_logic_vector(31 downto 0);
    signal  result: out std_logic_vector(31 downto 0)
);
end Entity Instruction;
```

Algoritmo 6.1: Interface em VHDL da Instrução *Customizada*

Elemento	Função
clk	Clock do sistema – 1 bit
reset	Reset assíncrono do sistema (ativo em *high*) – 1 bit
clk_en	Habilita a operação da instrução – 1 bit
start	Sinal ativo em *high*, é usado para especificar que as entradas estão validas – 1 bit
done	Sinal ativo em *high*, é usado para notificar a CPU de que o resultado é válido – 1 bit
n	Seletor de subfunção da instrução (*n*) – 8 bits
dataa	Entrada de dados do operando A – 32 bits
datab	Entrada de dados do operando B – 32 bits
result	Resultado da instrução – 32 bits

Tabela 6.1 – Elementos da Interface da Instrução *Customizada*

Um exemplo de como a instrução funciona está presente na Figura 6.2. Nessa figura vemos as formas de ondas utilizadas para realizar a operação de elevação ao cubo. Através da figura é possível identificar as três fases de operação da instrução (que serão vistas a seguir).

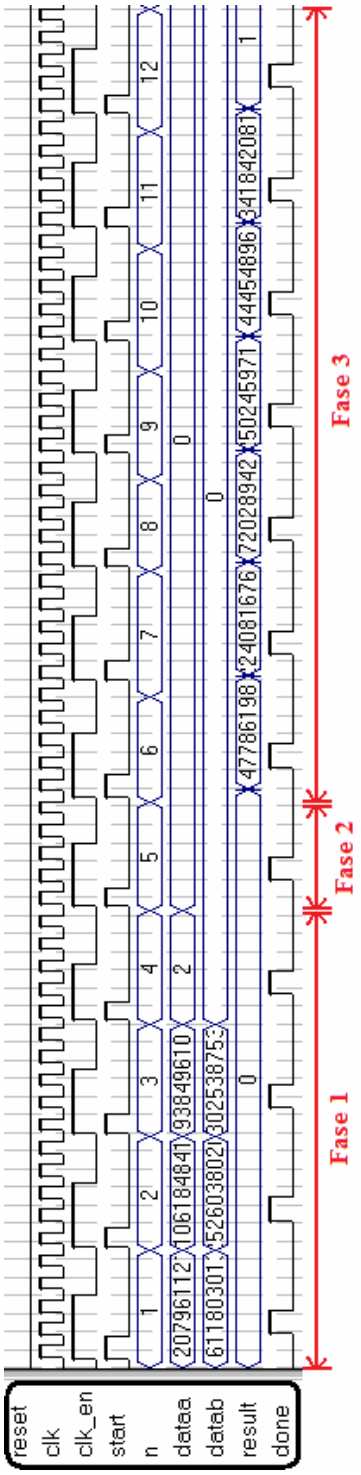

Figura 6.2: Formas de Ondas para a Instrução de Elevação ao Cubo

Uma instrução customizada no processador Nios II pode receber apenas 64 bits como entrada e retornar 32 como saída, portanto, é necessário dividir o cálculo da elevação ao cubo em F_3^{97} em três fases:

- Na primeira fase a instrução é chamada quatro vezes, sendo que em cada chamada 64 bits do elemento de entrada são transferidos para um registrador interno da instrução. Na última chamada apenas os últimos 2 bits do elemento de entrada são passados.
- Na segunda fase a computação da elevação ao cubo é realizada. Para isso, a instrução é novamente chamada e o código que indica tal tarefa é fornecido na entrada *n*.
- Na terceira e última fase o resultado da operação é retornado. Como é possível retornar apenas 32 bits por chamada da função, são necessárias 7 chamadas para retornar todo o resultado.

Portanto são necessárias 12 chamadas à instrução para realizar a elevação ao cubo em F_3^{97}. Todo esse procedimento pode ser observado na Figura 6.3.

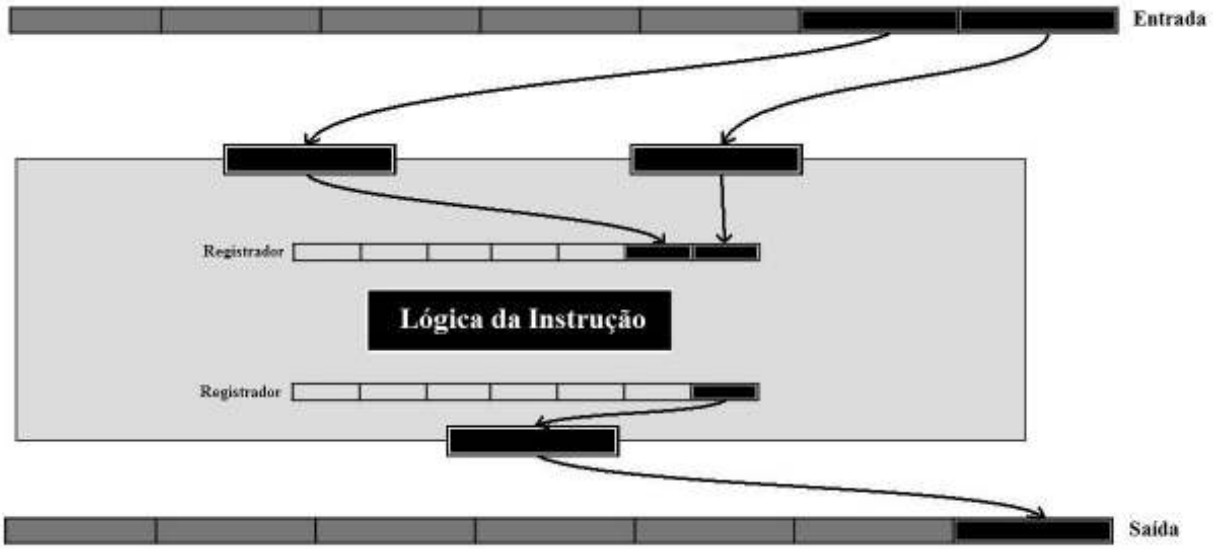

Figura 6.3: Instrução *Customizada* de Elevação ao Cubo

O fragmento de código C a seguir (Algoritmo 6.2) funciona como uma interface para acessar a instrução customizada de elevação ao cubo através do programa em C:

Interface para a Instrução *Customizada* de Elevação ao Cubo

```
ALT_CI_CUBING(n1, Input 1, Input 2); //Início da fase um
ALT_CI_CUBING(n2, Input 3, Input 4);
ALT_CI_CUBING(n3, Input 5, Input 6);
ALT_CI_CUBING(n4, Input 7, Input 8); //Fim

ALT_CI_CUBING(n5,A,B); //Fase dois

result1=(unsigned int) ALT_CI_CUBING(n6,A,B); //Início da fase três
result2=(unsigned int) ALT_CI_CUBING(n7,A,B);
result3=(unsigned int) ALT_CI_CUBING(n8,A,B);
result4=(unsigned int) ALT_CI_CUBING(n9,A,B);
result5=(unsigned int) ALT_CI_CUBING(n10,A,B);
result6=(unsigned int) ALT_CI_CUBING(n11,A,B);
result7=(unsigned int) ALT_CI_CUBING(n12,A,B); //Fim da fase três
```

Algoritmo 6.2: Interface em C para Acessar a Instrução *Customizada* de Elevação ao Cubo

Onde *n[1 ..12]* é a sub-função solicitada a cada chamada da instrução, *Input[1 ..8]* representa o elemento de entrada e *result[1 ..7]* o elemento resultante. A função *ALT_CI_CUBING* é usada para ler e enviar informação para a instrução de elevação ao cubo. Essa função é previamente definida na biblioteca *system.h*, que é automaticamente gerada pela ferramenta *SOPC Builder* do *Quartus II*.

O Algoritmo 6.3 apresenta o esquema completo que foi implementado como instrução. Como dito antes, esse algoritmo foi gerado através da técnica de *loop-unrolling* aplicada à seguinte expressão:

$$\text{Se } d(x) = \sum_{i=0}^{m-1} d_i x^i \text{ então } d(x)^3 \bmod f(x) \equiv \sum_{i=0}^{m-1} d_i x^{3i} \bmod f(x).$$

Elevação ao Cubo de um Elemento em F_3^{97}

Entrada: $d(x) \in F_3^{97}$.
Saída: $d(x)^3 \bmod f(x)$.

1: Result (0) ← d(0) + d(89) + d(93);
2: Result (1) ← d(65) + 2 · d(61);
3: Result (2) ← d(33);
4: Result (3) ← d(1) + d(90) + d(94);
5: Result (4) ← d(66) + 2 · d(62);
6: Result (5) ← d(34);

```
7:   Result (6) ← d(2) + d(91) + d(95);
8:   Result (7) ← d(67) + 2 · d(63);
9:   Result (8) ← d(35);
10:  Result (9) ← d(3) + d(92) + d(96);
11:  Result (10) ← d(68) + 2 · d(64);
12:  Result (11) ← d(36);
13:  Result (12) ← d(4) + d(93) – (d(89) + d(93));
14:  Result (13) ← d(65) + d(69) + d(61);
15:  Result (14) ← 2 · d(33) + d(37);
16:  Result (15) ← d(5) + d(94) – (d(90) + d(94));
17:  Result (16) ← d(66) + d(70) + d(62);
18:  Result (17) ← 2 · d(34) + d(38);
19:  Result (18) ← d(6) + d(95) – (d(91) + d(95));
20:  Result (19) ← d(67) + d(71) + d(63);
21:  Result (20) ← 2 · d(35) + d(39);
22:  Result (21) ← d(7) + d(96) – (d(92) + d(96));
23:  Result (22) ← d(68) + d(72) + d(64);
24:  Result (23) ← 2 · d(36) + d(40);
25:  Result (24) ← d(8) – d(93);
26:  Result (25) ← d(65) + d(69) + d(73);
27:  Result (26) ← 2 · d(37) + d(41);
28:  Result (27) ← d(9) – d(94);
29:  Result (28) ← d(66) + d(70) + d(74);
30:  Result (29) ← 2 · d(38) + d(42);
31:  Result (30) ← d(10) – d(95);
32:  Result (31) ← d(67) + d(71) + d(75);
33:  Result (32) ← 2 · d(39) + d(43);
34:  Result (33) ← d(11) – d(96);
35:  Result (34) ← d(68) + d(72) + d(76);
36:  Result (35) ← 2 · d(40) + d(44);
37:  Result (36) ← d(12);
38:  Result (37) ← d(69) + d(73) + d(77);
39:  Result (38) ← 2 · d(41) + d(45);
40:  Result (39) ← d(13);
41:  Result (40) ← d(70) + d(74) + d(78);
42:  Result (41) ← 2 · d(42) + d(46);
43:  Result (42) ← d(14);
44:  Result (43) ← d(71) + d(75) + d(79);
45:  Result (44) ← 2 · d(43) + d(47);
46:  Result (45) ← d(15);
47:  Result (46) ← d(72) + d(76) + d(80);
48:  Result (47) ← 2 · d(44) + d(48);
49:  Result (48) ← d(16);
```

50: Result (49) ← d(73) + d(77) + d(81);
51: Result (50) ← 2 · d(45) + d(49);
52: Result (51) ← d(17);
53: Result (52) ← d(74) + d(78) + d(82);
54: Result (53) ← 2 · d(46) + d(50);
55: Result (54) ← d(18);
56: Result (55) ← d(75) + d(79) + d(83);
57: Result (56) ← 2 · d(47) + d(51);
58: Result (57) ← d(19);
59: Result (58) ← d(76) + d(80) + d(84);
60: Result (59) ← 2 · d(48) + d(52);
61: Result (60) ← d(20);
62: Result (61) ← d(77) + d(81) + d(85);
63: Result (62) ← 2 · d(49) + d(53);
64: Result (63) ← d(21);
65: Result (64) ← d(78) + d(82) + d(86);
66: Result (65) ← 2 · d(50) + d(54);
67: Result (66) ← d(22);
68: Result (67) ← d(79) + d(83) + d(87);
69: Result (68) ← 2 · d(51) + d(55);
70: Result (69) ← d(23);
71: Result (70) ← d(80) + d(84) + d(88);
72: Result (71) ← 2 · d(52) + d(56);
73: Result (72) ← d(24);
74: Result (73) ← d(81) + d(85) + d(89);
75: Result (74) ← 2 · d(53) + d(57);
76: Result (75) ← d(25);
77: Result (76) ← d(82) + d(86) + d(90);
78: Result (77) ← 2 · d(54) + d(58);
79: Result (78) ← d(26);
80: Result (79) ← d(83) + d(87) + d(91);
81: Result (80) ← 2 · d(55) + d(59);
82: Result (81) ← d(27);
83: Result (82) ← d(84) + d(88) + d(92);
84: Result (83) ← 2 · d(56) + d(60);
85: Result (84) ← d(28);
86: Result (85) ← d(85) + d(89) + d(93);
87: Result (86) ← 2 · d(57) + d(61);
88: Result (87) ← d(29);
89: Result (88) ← d(86) + d(90) + d(94);
90: Result (89) ← 2 · d(58) + d(62);
91: Result (87) ← d(30);
92: Result (91) ← d(87) + d(91) + d(95);

93: Result (92) ← d(63) + 2 · d(59);
94: Result (93) ← d(31);
95: Result (94) ← d(88) + d(92) + d(96);
96: Result (95) ← d(64) + 2 · d(60);
97: Result (96) ← d(32);

Algoritmo 6.3: Algoritmo da Instrução de Elevação ao Cubo

Esse algoritmo pode ser implementado utilizando apenas a operação de soma, uma vez que a operação de subtração pode ser realizada usando a operação de soma, como veremos mais adiante. Iremos agora apresentar os elementos que juntos compõem a instrução de elevação ao cubo.

6.1.1 Soma de Elementos em F_3

A operação de adição pode ser eficientemente implementada em hardware, utilizando pequenos circuitos combinacionais. O *gate delay* desse circuito é baixo, possibilitando que a execução desta operação em hardware seja muito rápida. Como discutido anteriormente, a soma de elementos em F_3 é eficientemente calculada através da seguinte expressão [23]:

$$R_H \leftarrow (a_L \vee b_L) \oplus ((a_L \vee b_H) \oplus (a_H \vee b_L))$$
$$R_L \leftarrow (a_H \vee b_H) \oplus ((a_L \vee b_H) \oplus (a_H \vee b_L))$$

Onde a e b são os elementos de entrada em F_3 e R é o resultado da soma ($R_{HL} = a_{HL} + b_{HL}$), todos formados pelos bits H e L. Podemos observar o circuito resultante na Figura 6.4. O resultado da soma sofre a operação de módulo por 3, assim sendo a soma de dois elementos em F_3 gera outro elemento em F_3, ou seja, não existe *carry* em uma operação de soma. A Tabela 6.2 apresenta todos os valores possíveis encontrados na operação de soma em F_3.

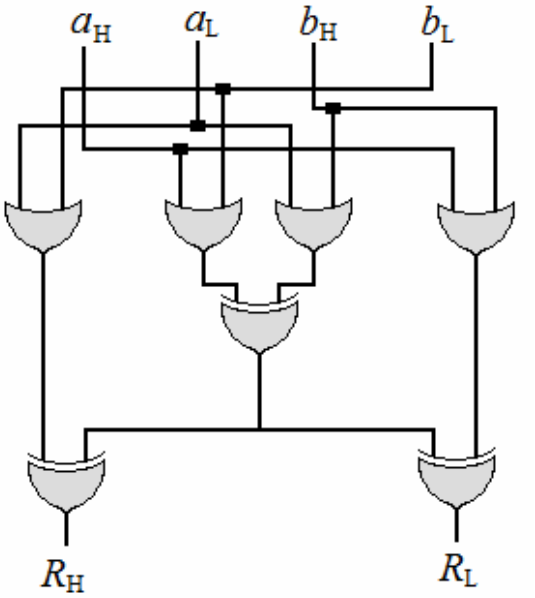

Figura 6.4: Circuito Lógico da Soma de Elementos em F_3

+	**00**	**01**	**10**
00	00	01	10
01	01	10	00
10	10	00	01

Tabela 6.2 – Valores da Soma Binária

6.1.2 Caminho Crítico do Circuito

O caminho crítico do circuito é o caminho mais longo que um sinal percorre para que a operação seja completada. Ou seja, o menor período de *clock* para que o circuito funcione de maneira correta tem que ser maior ou igual ao tempo necessário para um sinal percorrer todo o caminho crítico do circuito, assim sendo, a velocidade de *clock* a que podemos submeter o circuito está ligado à latência do caminho crítico, isto é o "tamanho" do caminho crítico. Quanto menor é o caminho crítico, maior é a freqüência de *clock* em que o circuito pode trabalhar e menor é o tempo que ele gasta para realizar a sua tarefa.

Como dito anteriormente, cada elemento do polinômio resultante da elevação ao cubo é calculado paralelamente. Nesse cenário, o pior dos caminhos acontece quando temos que calcular a soma entre 4 elementos em F_3, como acontece na linha 13, 16, 19 e 22 do Algoritmo 6.3. Observe que essas quatro linhas necessitam de 3 somas devido à redução pelo polinômio irredutível de grau *m*. A Figura 6.5 apresenta um exemplo do circuito que é o

caminho crítico da instrução de elevação ao cubo. Nessa figura o registrador 1 armazena o elemento $a(x)$ de entrada e R(x) armazena o resultado da operação de elevação ao cubo.

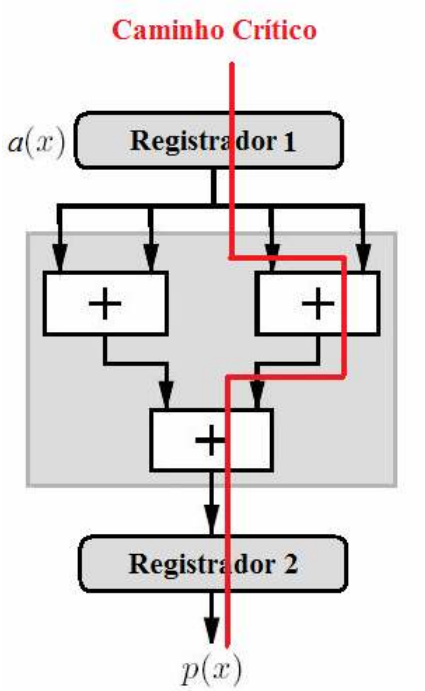

Figura 6.5: Caminho Crítico da Instrução de Elevação ao Cubo

6.2 Multiplicação em F_3^{97}

Em nosso trabalho usamos uma modificação do algoritmo MSE (*Most-Significant Element*) de multiplicação em F_3^{97}, apresentado em [5]. Basicamente o algoritmo recebe como entrada dois elementos em F_3^{97}, calcula a multiplicação entre os dois, reduz o resultado pelo polinômio irredutível de grau m e retorna como resultado um elemento em F_3^{97}. O algoritmo que implementa esta operação pode ser observado no Algoritmo 5.2.

O algoritmo é executado 32 vezes para o caso de $D = 3$, onde D é um parâmetro que define o número de coeficientes de $a(x)$ que serão processados a cada ciclo de *clock*. Ele também chama uma subfunção denominada *mod f(x)*, a qual realiza a redução pelo polinômio irredutível de grau m.

Análogo a chamada de instrução *customizada* no processador Nios II, a chamada a um módulo também apresenta algumas restrições. Um módulo acoplado ao barramento *Avalon* do processador Nios II recebe como entrada um operador de 32 bits e retorna 32 bits como saída.

Ele também utiliza um parâmetro de 8 bits (*n*) que é passado a cada chamada ao módulo, com o intuito de selecionar qual subfunção do módulo está sendo requisitada.

A interface utilizada no módulo para se acoplar ao barramento *Avalon* pode ser observada no Algoritmo 6.4, o significado de cada porta do módulo é apresentado na Tabela 6.3

Interface em VHDL do Módulo de Multiplicação

```
Entity Multiplicador is
port(
    clk : in std_logic;
    reset : in std_logic;
    chipselect : in std_logic;
    address : in std_logic_vector(7 downto 0);
    write : in std_logic; -- leitura de dados prota
    writedata : in std_logic_vector(31 downto 0);
    read : in std_logic; -- saida pronta
    readdata : out std_logic_vector(31 downto 0);
    waitrequest : out std_logic
);
end Entity Multiplicador;
```

Algoritmo 6.4: Interface em VHDL do Módulo de Multiplicação

Elemento	Função
Clk	Clock de entrada do módulo – 1 bit
Reset	Reset de entrada do módulo (ativo em high) – 1 bit
Chipselect	Habilita a operação do módulo (ativo em high) – 1 bit
Address	Endereçamento, ou seletor de subfunção do módulo (n) – 8 bits
Write	Habilita a leitura de dados pronta (ativo em high) – 1 bit
Writedata	Entrada de dados – 32 bits
Read	Habilita a saída de dados (ativo em high) – 1 bit
Readdata	Saída de dados – 32 bits
Waitrequest	Faz o processador esperar até que a operação termine (ativo em high) – 1 bit

Tabela 6.3 – Elementos da Interface do Módulo

Um exemplo de como o módulo deve funcionar é mostrado na Figura 6.6. Nessa figura vemos as formas de ondas utilizadas para realizar a multiplicação entre dois elementos (1 e 1). Inicialmente um reset é desferido ao módulo, então são realizadas as três fases de operação (que serão vistas a seguir).

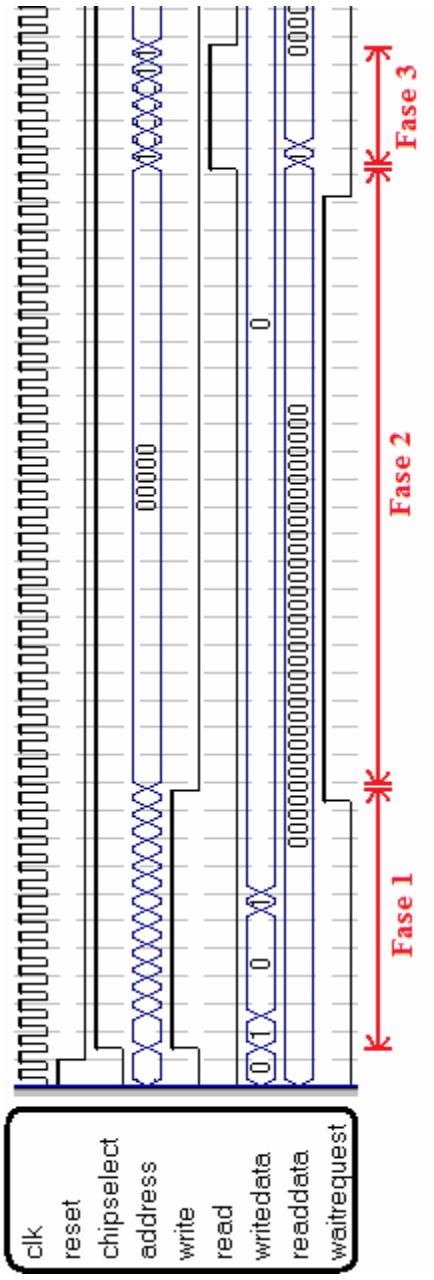

Figura 6.6: Formas de Ondas para o Módulo de Multiplicação

Devido às restrições de acesso ao módulo, a operação de multiplicação é subdividida em três fases:

- Na primeira fase o módulo de multiplicação é chamado 14 vezes para passar os dois elementos de entrada através do operando de 32 bits de entrada do módulo. Os dois elementos são armazenados em dois registradores internos ao módulo.
- Na segunda fase a multiplicação é realizada. Essa fase se inicia durante a última chamada ao módulo ocorrida na fase anterior, ou seja, ela não necessita de uma nova chamada ao módulo de multiplicação para ser executada.
- Na terceira e última fase, 7 chamadas ao módulo de multiplicação são realizadas para retornar o resultado obtido na multiplicação.

Portanto, no total são realizadas 21 chamadas ao modulo de multiplicação para que a operação de multiplicação de elementos em F_3^{97} seja realizada. Uma ilustração desse procedimento pode ser observada na Figura 6.7. Uma otimização interessante que poderia ser futuramente implementada é passar inicialmente todo polinômio multiplicando para o módulo de multiplicação e intercalar os 32 ciclos necessário para realizar a multiplicação com a transferência do polinômio multiplicador (a partir do recebimento de uma das 7 palavras que compõem o multiplicador, já é possível iniciar a multiplicação). Essa otimização conseguiria diminuir um pouco o número excessivo de ciclos gastos na comunicação, contudo, ela não foi implementada em nosso trabalho pela questão de manter a simplicidade.

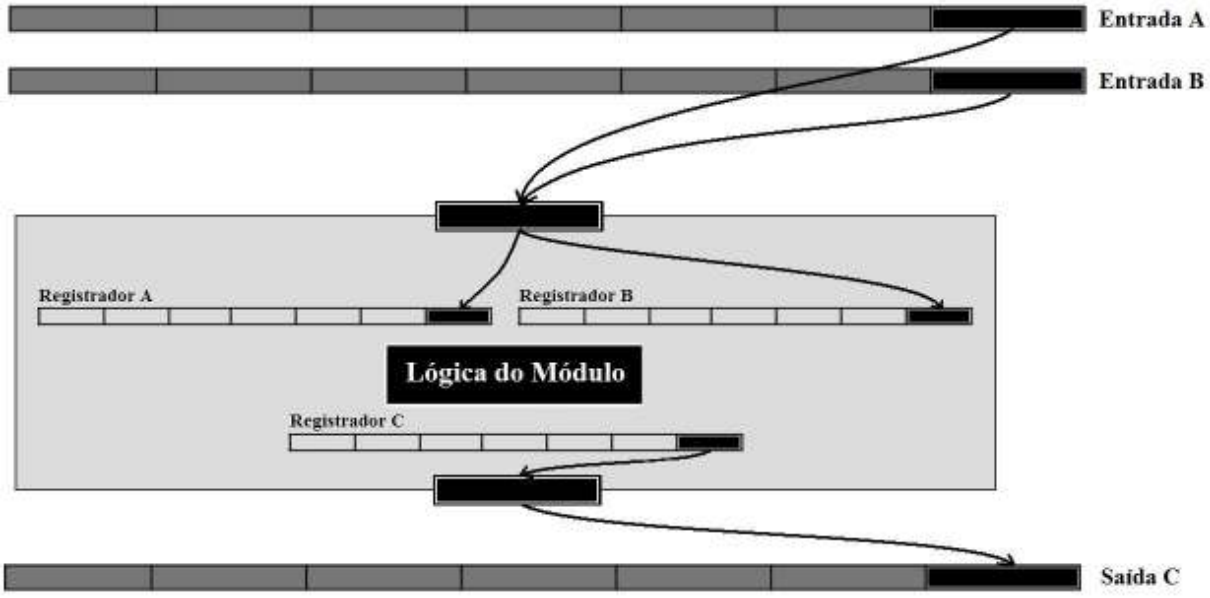

Figura 6.7: Módulo de Multiplicação

O fragmento de código C a seguir (Algoritmo 6.5) funciona como uma interface para acessar o módulo de multiplicação através do programa em C.

Interface em C para a Instrução *Customizada* de Multiplicação

```
IOWR(MULTIPLICATION_BASE, n1, InputA_1); //Início da fase um
IOWR(MULTIPLICATION_BASE, n2, InputA_2);
IOWR(MULTIPLICATION_BASE, n3, InputA_3);
IOWR(MULTIPLICATION_BASE, n4, InputA_4);
IOWR(MULTIPLICATION_BASE, n5, InputA_5);
IOWR(MULTIPLICATION_BASE, n6, InputA_6);
IOWR(MULTIPLICATION_BASE, n7, InputA_7);
IOWR(MULTIPLICATION_BASE, n8, InputB_1);
IOWR(MULTIPLICATION_BASE, n9, InputB_2);
IOWR(MULTIPLICATION_BASE, n10, InputB_3);
IOWR(MULTIPLICATION_BASE, n11, InputB_4);
IOWR(MULTIPLICATION_BASE, n12, InputB_5);
IOWR(MULTIPLICATION_BASE, n13, InputB_6);
IOWR(MULTIPLICATION_BASE, n14, InputB_7); //Fim da fase um. Fase dois

result1 = IORD(MULTIPLICATION_BASE, n15); //Início da fase três
result2 = IORD(MULTIPLICATION_BASE, n16);
result3 = IORD(MULTIPLICATION_BASE, n17);
result4 = IORD(MULTIPLICATION_BASE, n18);
result5 = IORD(MULTIPLICATION_BASE, n19);
result6 = IORD(MULTIPLICATION_BASE, n20);
result7 = IORD(MULTIPLICATION_BASE, n21); //Fim da fase três
```

Algoritmo 6.5: Interface em C para a Instrução *Customizada* de Multiplicação

Na Figura 7.12, *n[1 ..21]* identifica a subfunção solicitada do módulo, *InputA_[1..7]* e *InputB_[1..7]* são os coeficientes dos respectivos elementos de entrada A e B. O elemento resultante da multiplicação é retornado através do *result[1 ..7]*. As funções *IOWR* e *IORD* são definidas na biblioteca *io.h*, a qual é gerada automaticamente pela ferramenta *SOPC Builder*. A função *IOWR* realiza a operação de escrita no módulo endereçado pela constante *MULTIPLICATION_BASE* (também é gerada automaticamente pela ferramenta *SOPC Builder*), já a função *IORD* realiza a leitura do módulo, também endereçado pela constante *MULTIPLICATION_BASE*.

A Figura 6.8 apresenta a micro-arquitetura do módulo de multiplicação desenvolvido em hardware (com $D = 3$). Para o caso de $D = 3$, três circuitos são necessários para calcular os produtos parciais.

Após o cálculo dos produtos parciais, cada produto parcial é deslocado por zero, $D - 2$ ou $D - 1$ elementos (as operações de deslocamentos são realizadas utilizando apenas fios). Depois de realizados os deslocamentos, a soma dos produtos parciais é executada e por último a

redução pelo polinômio irredutível de grau *m* é realizada. O resultado de tal circuito realimenta a soma na próxima iteração. Esse circuito requer apenas 32 *clocks* para calcular uma multiplicação de elementos em F_3^{97}.

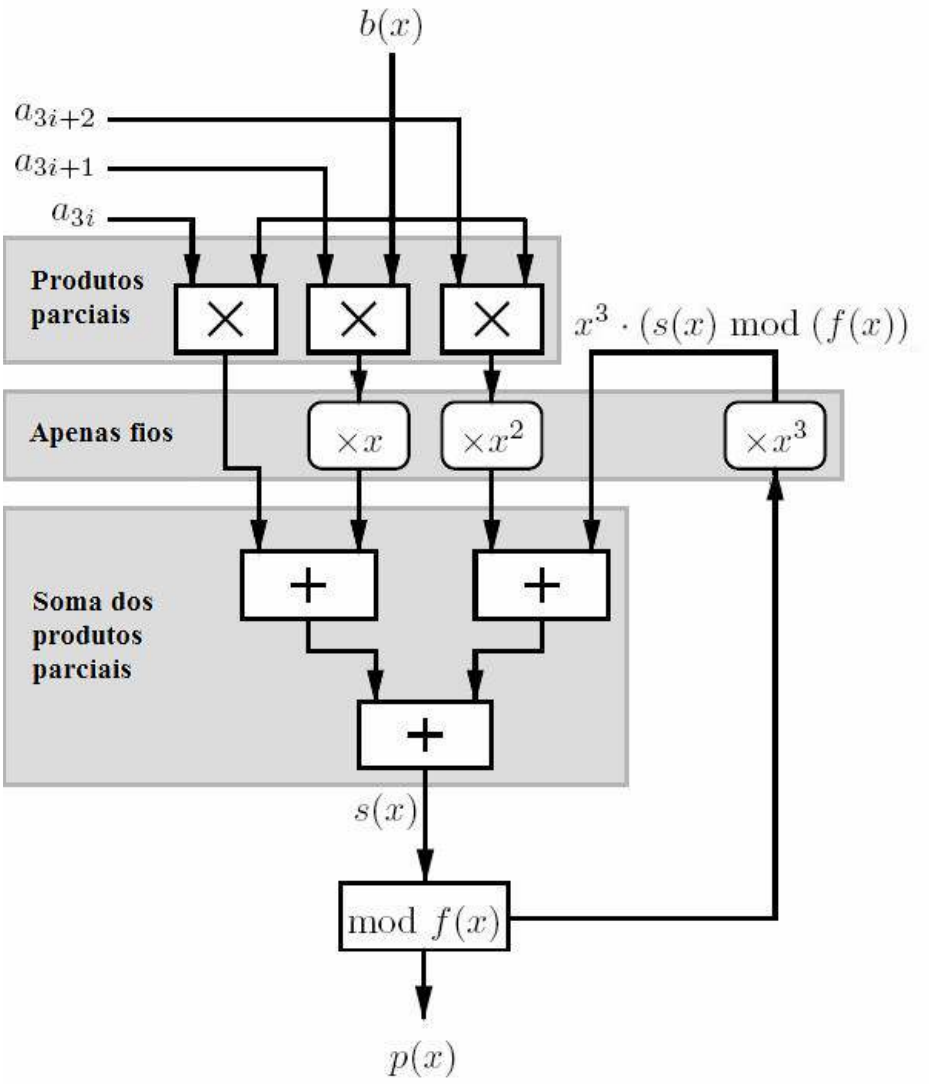

Figura 6.8: Micro-Arquitetura do Módulo de Multiplicação

Iremos agora apresentar os elementos que juntos compõem o módulo de multiplicação.

6.2.1 Multiplicação de Elementos em F_3

Cada elemento pertencente a F_3 é representado por 2 bits em hardware, o bit *high* (H) e o bit *low* (L). Como vimos no capítulo 5, a multiplicação de elemento em F_3 é eficientemente calculada através da seguinte expressão [5]:

$$R_H \leftarrow (a_L \wedge b_H) \vee (a_H \wedge b_L)$$
$$R_L \leftarrow (a_L \wedge b_L) \vee (a_H \wedge b_H)$$

Onde a e b são os elementos de entrada em F_3 e R é o resultado da multiplicação ($R_{HL} = a_{HL} \cdot b_{HL}$), todos formados pelos bits H e L. O mapeamento desta operação para o hardware é facilmente realizado, poucos elementos lógicos são necessários na sua implementação, ou seja, o *gate delay* desse circuito é baixo, possibilitando uma alta velocidade na execução desta operação.

Podemos observar o circuito resultante na Figura 6.9. O resultado da multiplicação sofre a operação de módulo por 3, assim sendo a multiplicação de dois elementos em F_3 gera outro elemento em F_3. A Tabela 6.4 apresenta todos os valores possíveis encontrados na multiplicação.

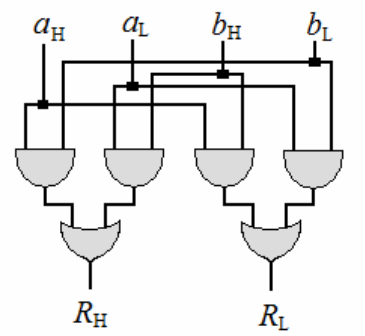

Figura 6.9: Circuito Lógico da Multiplicação de Elementos em F_3

X	**00**	**01**	**10**
00	00	00	00
01	00	01	10
10	00	10	01

Tabela 6.4 – Valores da Multiplicação Binária

6.2.2 Geração dos Produtos Parciais

Um produto parcial é gerado através da multiplicação de um elemento em F_3 do polinômio multiplicador por todo o polinômio que compõe o multiplicando. O nosso módulo multiplicador possui 3 elementos lógicos que realizam o cálculo de 3 produtos parciais em paralelo a cada *clock*, uma vez que nosso parâmetro *D* escolhido foi 3.

Alguns trabalhos [6][24] já apresentaram estudos comparativos entre diversos tamanhos para *D*. Quanto maior o tamanho de *D*, menos ciclos de *clock* são necessários para calcular uma multiplicação, uma vez que calculamos mais produtos parciais por *clock*. Contudo, quanto maior o tamanho de *D*, maior é o tamanho do circuito necessário para implementar a multiplicação.

O circuito lógico que implementa a geração do produto parcial é composto por 97 sub-elementos que realizam a operação de multiplicação de elementos em F_3. Ele possui como entrada 194 bits para o operando multiplicando e 2 bits para o elemento em F_3 do polinômio multiplicador. Como saída é retornado um elemento em F_3^{97}. É interessante observar que devido às propriedades da multiplicação de elementos em F_3, ou seja, por estarmos trabalhando em aritmética modular, não existe *carry* entre a multiplicação de um elemento e outro, assim todas as multiplicações de elementos em F_3 podem ser realizadas em paralelo.

Um esquema do circuito lógico que implementa o cálculo do produto parcial pode ser observado na Figura 6.10. Nessa figura vemos a geração de um produto parcial resultante da multiplicação do elemento $a \in F_3^{97}$ e do elemento $b \in F_3$.

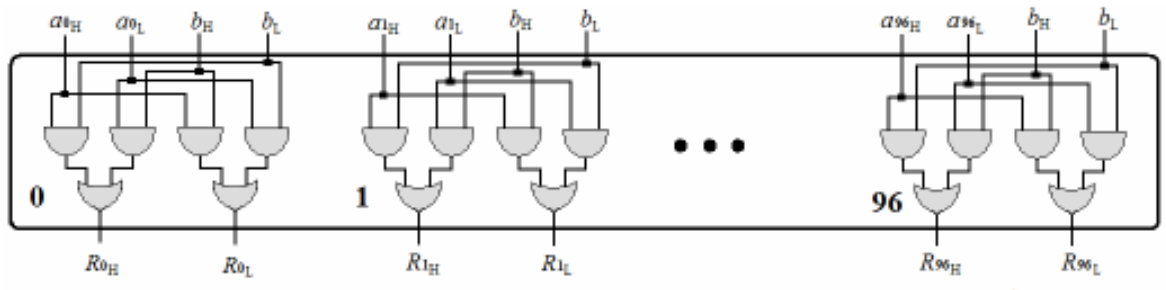

Figura 6.10: Circuito Lógico da Geração dos Produtos Parciais

No capítulo 7 apresentaremos uma maneira mais eficiente de realizar a geração dos produtos parciais.

6.2.3 Shifters

São utilizados três *shifters* na implementação do módulo de multiplicação. Os *shifters* deslocam um, dois ou três elementos em F_3 (os elementos resultantes ficam um, dois ou três graus maior que o grau 96 de entrada), inserindo zeros nas posições de menor ordem (deslocadas).

Esses *shifters* são implementando utilizando apenas fios, nenhum elemento lógico é necessário para essa implementação. A Figura 6.11 apresenta o esquema de implementação para o *shifter* de 3 posições. Os outros dois shifters seguem o mesmo padrão.

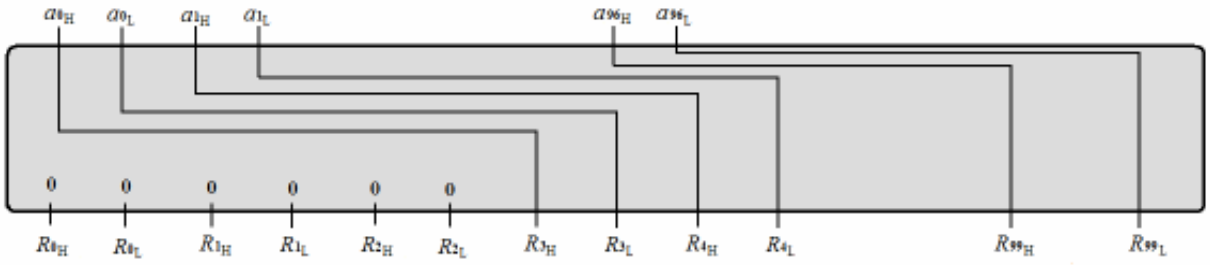

Figura 6.11: *Shifter* de Três Elementos

6.2.5 Soma dos Produtos Parciais

Após os produtos parciais sofrerem o *shift*, eles são então somados. Como podemos observar na Figura 6.8, são realizadas 3 somas. Na primeira soma, um produto parcial em F_3^{97} é somado com outro produto em F_3^{98}. Na segunda soma, um produto parcial em F_3^{99} é somado com outro produto parcial em F_3^{100} (esse produto é a realimentação do circuito; na primeira iteração ele tem valor zero). Na terceira e última soma, os resultados das somas anteriores são somados, ou seja, um elemento em F_3^{98} e um em F_3^{100}. Resultando assim em um elemento final de grau F_3^{100} ($m + D - 1$).

Três diferentes circuitos são necessários para implementar as 3 somas dos produtos parciais. Cada um desses circuitos é composto por um conjunto de sub-elementos que realizam a soma entre elementos em F_3. Um exemplo de um desses circuitos pode ser observado na Figura 6.12. Nessa figura é apresentada a soma entre um elemento $a \in F_3^{97}$ e um $b \in F_3^{98}$. Como os elementos são de graus diferentes, o último elemento não é somando, passando direto para o resultado. Esse mesmo esquema é utilizado nos outros dois somadores.

É interessante observar que devido às propriedades da soma de elementos em F_3, ou seja, por estarmos trabalhando em aritmética modular, não existe *carry* entre as somas de um elemento e outro, assim todas as somas de elementos em F_3 podem ser realizadas em paralelo.

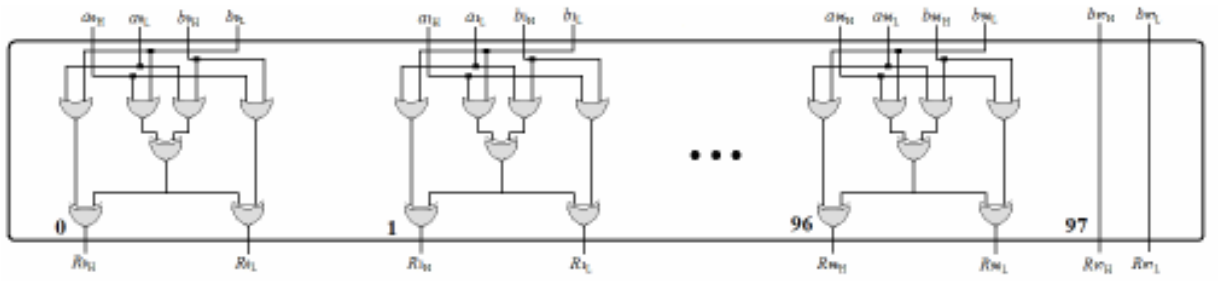

Figura 6.12: Circuito Lógico da Soma dos Produtos Parciais

6.2.5 Redução pelo Polinômio Irredutível de Grau *m*

Como mencionado anteriormente, nosso polinômio irredutível de grau *m* é um trinômio ($x^{97} + x^{12} + 2$), portanto podemos explorar o fato de $x^{97} \equiv -(x^{12} + 2)\ mod\ f(x)$, na simplificação da operação. Tornando assim a operação de redução simples e rápida.

O Algoritmo 5.3 implementa a execução dessa operação. Ele basicamente remove do elemento de entrada $x^{12} + 2$ para cada elemento que estiver acima do grau 97, respeitando o grau de cada elemento com relação a 97.

O elemento resultante das somas dos produtos parciais possui no pior caso o grau 99, ou seja, no pior caso precisamos reduzir três graus do elemento. Seja $P = p_0 + p_1x + p_2x^2 + ... + p_{99}x^{99} \in F_3{}^{100}$ o elemento resultante das somas dos produtos parciais, a redução pelo polinômio irredutível de grau *m* é realizada da seguinte forma:

- Grau 99: $P - p_{99}(x^{12+2} + 2x^2) - p_{99}$
- Grau 98: $P - p_{98}(x^{12+1} + 2x) - p_{98}$
- Grau 97: $P - p_{97}(x^{12} + 2) - p_{97}$

A negação de um elemento em F_3 pode ser realizada apenas invertendo-se os bits H e L que compõem o elemento. Dada essa premissa, a operação de subtração de elementos pode ser realizada utilizando-se a função de soma de elementos em F_3, ou seja, basta apenas realizar a inversão do segundo elemento antes de realizar a operação de soma ($R_{HL} = a_{HL} + b_{LH}$) [5]. A Tabela 6.5 apresenta todos os valores possíveis encontrados na operação de subtração em F_3 (linha – coluna).

Outra questão importante é que a multiplicação de um elemento em F_3 por dois também pode ser realizada apenas invertendo-se os bits do multiplicando (observe a Tabela 6.4). Através disso, podemos simplificar a expressão:

$$p_ix - 2p_jx = p_ix + p_jx$$

Essa simplificação foi utilizada na implementação da redução pelo polinômio irredutível de grau *m*. O circuito lógico que implementa a redução pelo polinômio irredutível de grau *m*

pode ser observado na Figura 6.13. Nessa figura vemos a redução de um polinômio $P = p_0 + p_1x + p_2x^2 + ... + p_{99}x^{99} \in F_3^{100}$ pelo polinômio irredutível de grau m, obtendo $R \in F_3^{97}$ como resultado.

–	**00**	**01**	**10**
00	00	10	01
01	01	00	01
10	10	01	00

Tabela 6.5 – Valores da Subtração Binária

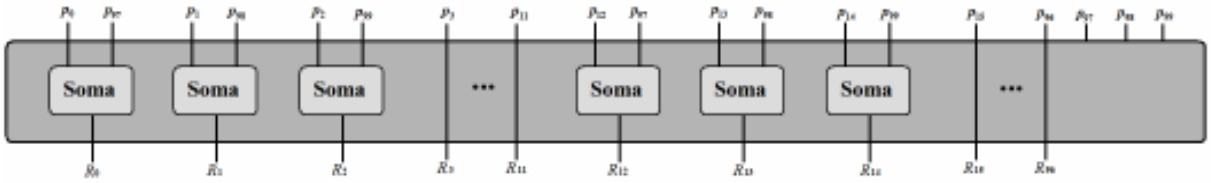

Figura 6.13: Circuito Lógico da Redução pelo Polinômio Irredutível de Grau *m*

6.2.6 Registradores Internos

Existem três grandes registradores internos na micro-arquitetura do módulo de multiplicação. A posição em que eles se encontram pode ser observada na Figura 6.14. Os registradores 1 e 2 armazenam os valores dos operando de entrada $a(x)$ e $b(x)$, ou seja, dois elementos em F_3^{97}. Já o registrador 3 armazena o valor do elemento obtido a cada iteração do circuito, e após a última iteração ele contém o valor resultante da operação de multiplicação entre $a(x)$ e $b(x)$. É interessante observar que o registrador 3 realimenta o circuito com o valor da iteração anterior e que na primeira iteração o valor contido nesse registrador é zero.

Além desses registradores, existem outros registradores menores que são utilizados para o controle da operação da multiplicação, como a contagem do número de iterações.

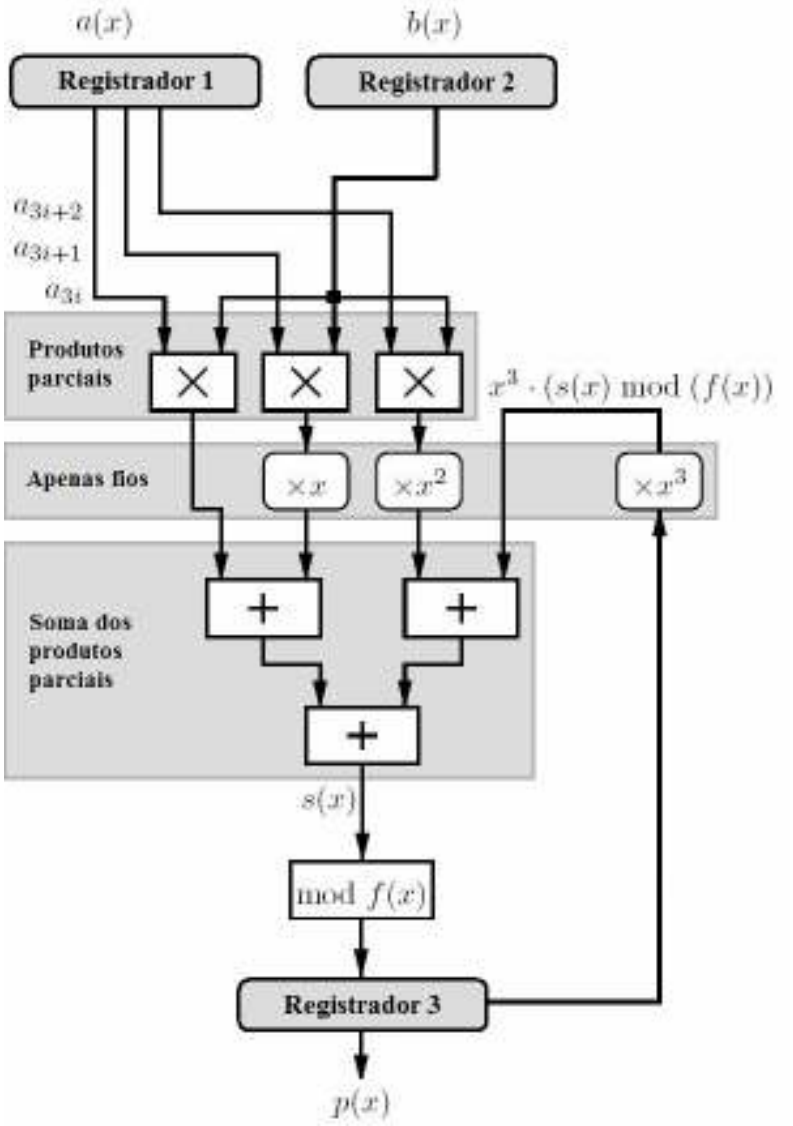

Figura 6.14: Registradores Internos do Módulo de Multiplicação

É sabido que quanto menor é o caminho crítico de um circuito, maior é a freqüência de *clock* em que o circuito pode trabalhar e menor é o tempo que ele gasta para realizar a sua operação. Assim sendo é importante tentar diminuir e otimizar o caminho crítico da implementação. A Figura 6.15 apresenta o caminho crítico do módulo de multiplicação.

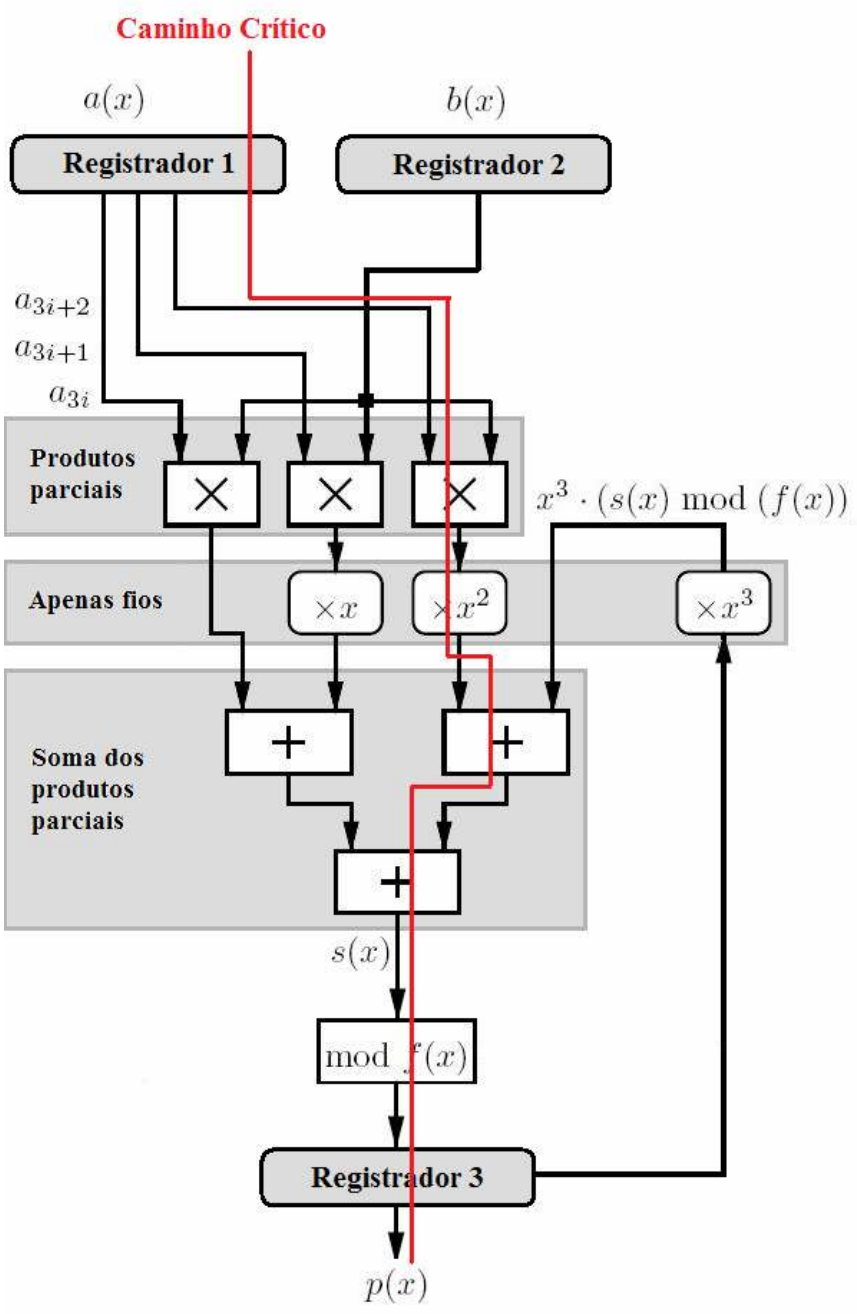

Figura 6.15: Caminho Crítico do Módulo de Multiplicação

Capítulo 7

Abordagem Hardware/Software Genérica

Assim como a abordagem HW/SW especializada, a abordagem HW/SW genérica implementa em software o emparelhamento sobre curvas elípticas, contudo ela usa algumas instruções "genéricas" em hardware para acelerar o cálculo do emparelhamento.

Diferentemente das instruções desenvolvidas na abordagem especializada (que implementa toda uma operação em hardware), as instruções empregadas na abordagem genérica são utilizadas para acelerar certas operações críticas envolvidas no emparelhamento, contudo, apenas pequenas partes genéricas dessas operações são implementadas. Dessa forma a implementação genérica é mais flexível e pode ser expandida posteriormente, apenas alterando a implementação em software da operação, ou seja, sem necessitar de alterações no hardware [11].

Duas instruções foram escolhidas para serem implementadas como instruções customizadas no processador Nios II. São elas a instrução de soma e a instrução de multiplicação de elementos em F_3^{16}. Essas instruções têm por objetivo a aceleração da multiplicação de elementos envolvida no emparelhamento de curvas elípticas. Usando essas duas instruções é possível acelerar o cálculo de uma multiplicação em F_3^m.

7.1 Multiplicação de Elementos em F_3^{16}

Essa instrução recebe como entrada dois elementos em F_3^{16} (representado por uma palavra de 32 bits) e retorna como resultado um elemento em F_3^{31} (representado por duas palavras de 32 bits). Ela utiliza a mesma interface apresentada no Algoritmo 6.1 para se acoplar ao processador Nios II. Como é possível retornar apenas 32 bits a cada chamada de instrução, são necessárias duas chamadas à instrução para obter todos os 64 bits do resultado (na primeira chamada são retornados os 32 bits menos significantes, e na segunda são retornados os 32 bits mais significantes). A operação realizada por essa instrução pode ser observada na Figura 7.1.

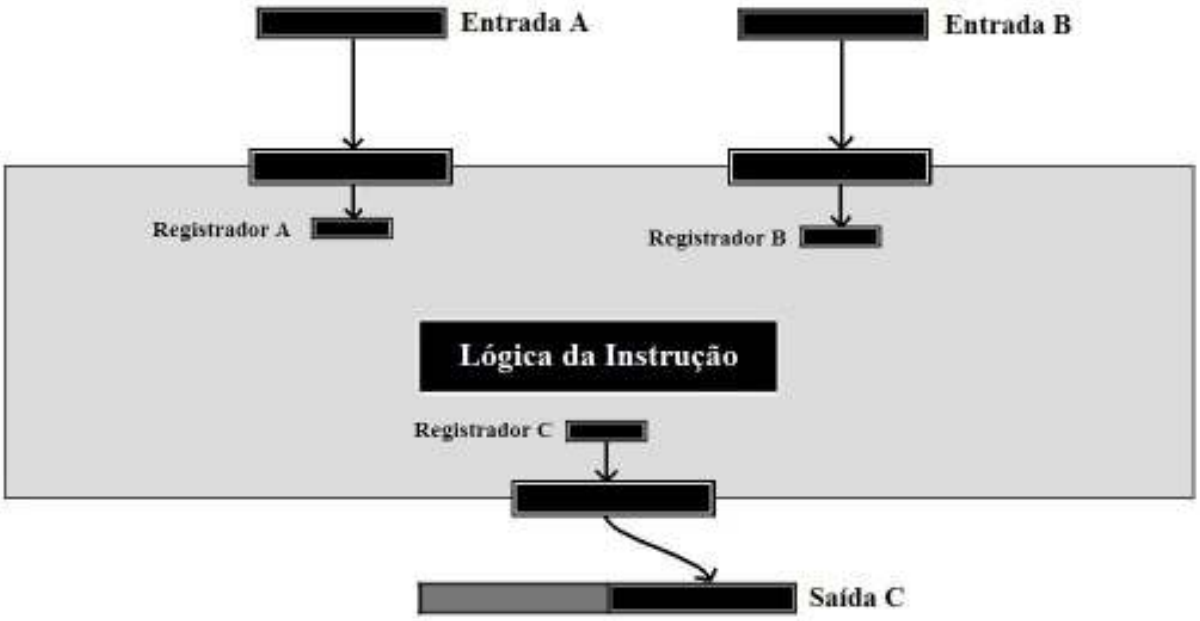

Figura 7.1: Instrução de Multiplicação Genérica

A instrução de multiplicação genérica realiza uma multiplicação completa entre dois elementos em F_3^{16}, contudo a redução pelo polinômio irredutível de grau *m* não é realizada, gerando assim como resultado um elemento em F_3^{31}.

Para exemplificar o princípio de funcionamento da instrução de multiplicação genérica, vamos tomar como exemplo a multiplicação realizada na Figura 7.2. Nessa figura observamos a realização de uma multiplicação comum entre dois elementos (*a* e *b*) formados por 10 elementos em F_3 (quadrados em cinza), ou seja, dois elementos em F_3^{10}.

Considerando que estamos de posse de uma instrução genérica capaz de realizar a multiplicação entre elementos em F_3^3, poderíamos subdividir cada polinômio em 4 sub-elementos pertencentes a F_3^3 e então realizar a multiplicação através de 16 multiplicações em F_3^3. Através do padrão de cores podemos observar qual parte da multiplicação cada uma das 16 multiplicações em F_3^3 está realizando.

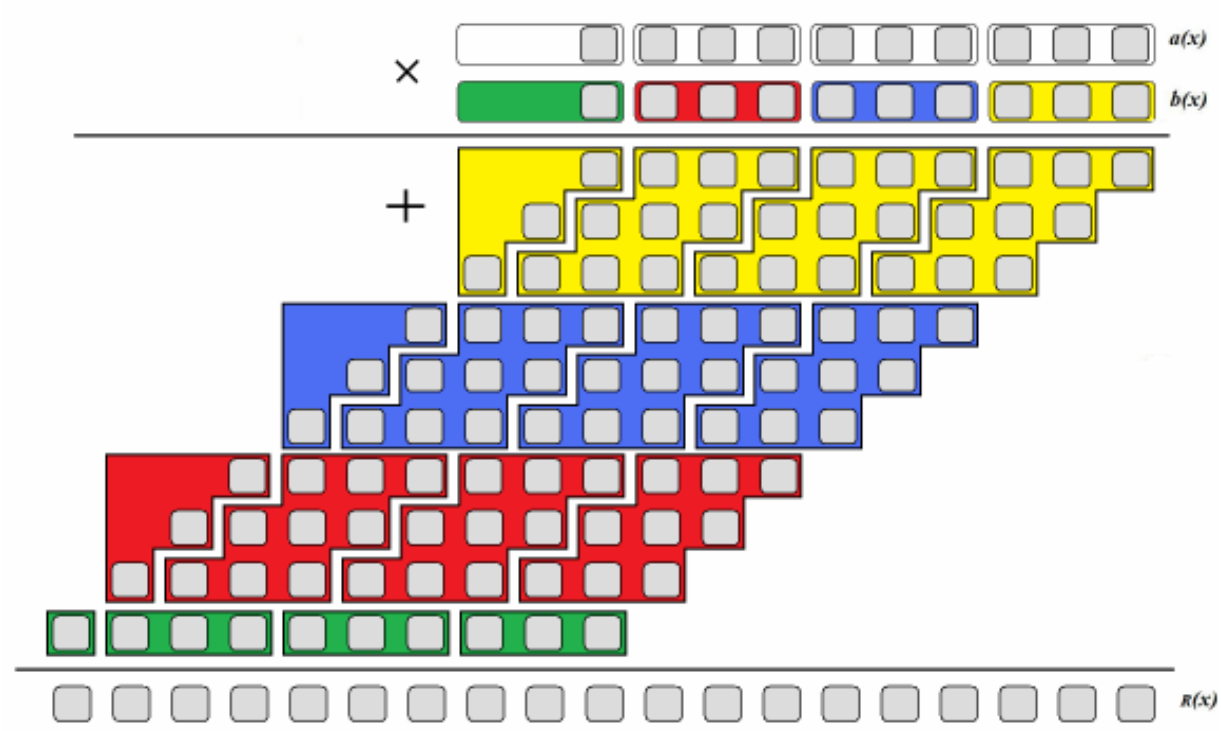

Figura 7.2: Exemplo do Funcionamento da Instrução de Multiplicação Genérica

A Figura 7.3 apresenta a micro-arquitetura da instrução de multiplicação genérica desenvolvida em hardware. Para que essa instrução seja completada em um ciclo de *clock*, temos que estipular que o valor de *D* seja o mesmo que o valor do elemento de entrada, no caso 16. Para o caso de $D = 16$, 16 circuitos são necessários para calcular os produtos parciais. Após o cálculo dos produtos parciais, cada produto parcial é deslocado 0..15 elementos e posteriormente são todos somados.

Uma nova estratégia para realizar a geração, soma e deslocamento dos produtos parciais foi desenvolvida. Esse circuito requer apenas um *clock* para calcular uma multiplicação entre elementos em F_3^{16}.

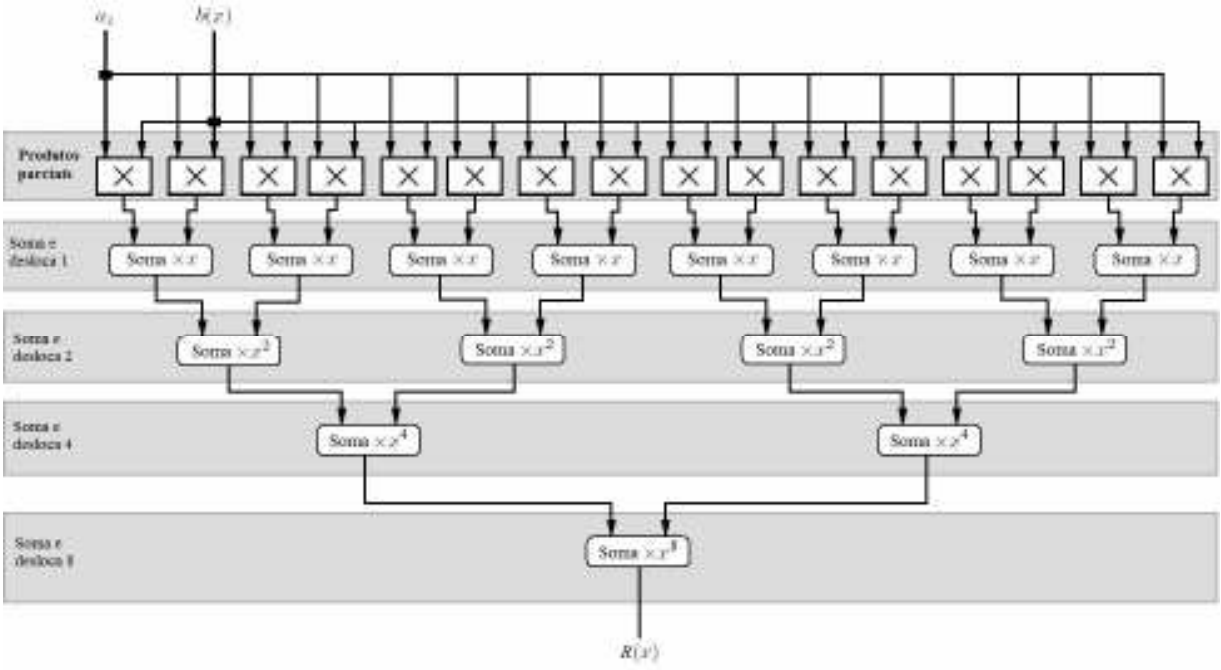

Figura 7.3: Micro-Arquitetura da Instrução de Multiplicação Genérica

Iremos agora apresentar os elementos que juntos compõem a instrução de multiplicação genérica e mostrar os detalhes da nova estratégia.

7.1.1 Geração dos Produtos Parciais

Um produto parcial é gerado através da multiplicação de um elemento em F_3 do polinômio multiplicador por todo o polinômio que compõe o multiplicando. A nossa implementação da instrução de multiplicação genérica possui 16 circuitos lógicos que realizam o cálculo de 16 produtos parciais em paralelo a cada *clock*, uma vez que nosso parâmetro *D* escolhido foi 16.

Na implementação desses elementos utilizamos uma estratégia diferente da utilizada na implementação dos mesmos elementos na abordagem especializada. Nesse novo esquema usamos um multiplexador para calcular o valor correspondente ao invés de sub-elementos que realizam a multiplicação de elementos em F_3. Esse multiplexador se vale de algumas características para realizar o cálculo do produto parcial. Essa nova abordagem é mais simples e elegante que a abordagem apresentada anteriormente.

Sejam $P = p_0 + p_1x + p_2x^2 + ... + p_{m-1}x^{m-1} \in F_3^m$ tal que $p_i \in F_3$ e $q \in F_3$. Temos então que:

- Se $q = 0$ então $P \cdot q = 0$;
- Se $q = 1$ então $P \cdot q = P$;
- Se $q = 10$ então $P \cdot q = P^*$.

Onde P^* equivale a P com os dois bits de todos p_i de invertidos. Observe que essa inversão pode ser realizada apenas com fios. Um esquema do circuito lógico que implementa o cálculo do produto parcial pode ser observado na Figura 7.4. Nessa figura vemos a geração de um produto parcial resultante da multiplicação dos elementos $b \in F_3^{16}$ e do elemento $a \in F_3$. As três situações possíveis de ocorrer são apresentadas.

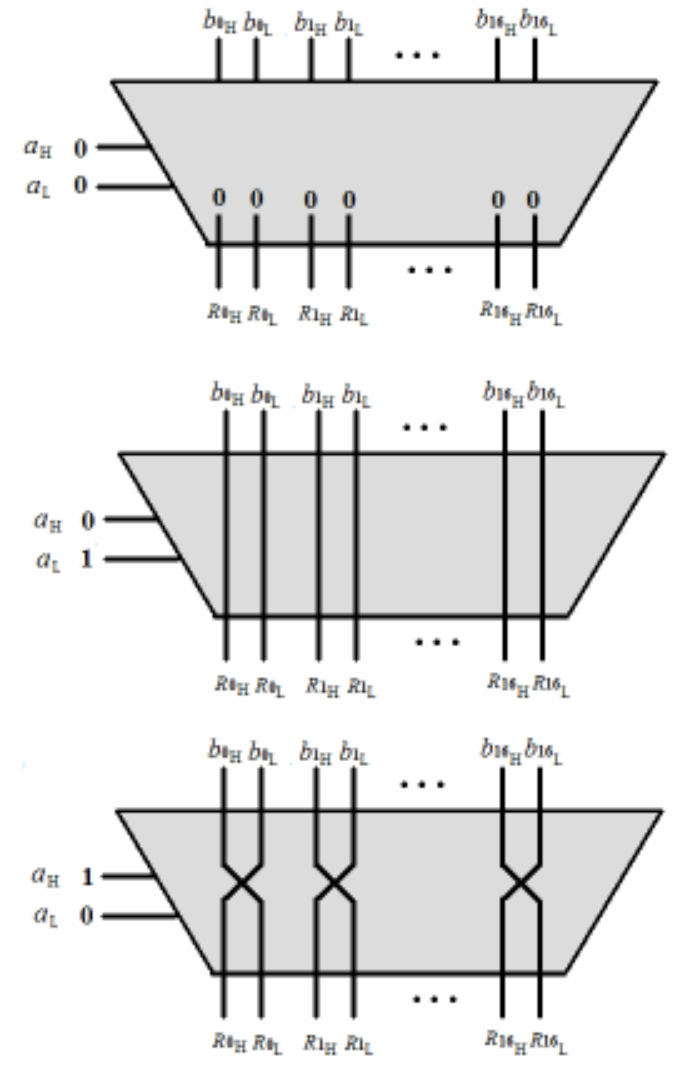

Figura 7.4: Circuito Lógico da Geração dos Produtos Parciais

7.1.2 Soma dos Produtos Parciais

Diferentemente da implementação do módulo de multiplicação especializada, na implementação da instrução genérica os *shifters* ficam embutidos nos elementos que realizam a soma dos produtos parciais. Assim temos 4 tipos diferentes de somadores:

- Somador com um deslocamento – soma dois elementos em F_3^{16} e retorna um elemento em F_3^{17};
- Somador com dois deslocamentos – soma dois elementos em F_3^{17} e retorna um elemento em F_3^{19};
- Somador com quatro deslocamentos – soma dois elementos em F_3^{19} e retorna um elemento em F_3^{23};
- Somador com oito deslocamentos – soma dois elementos em F_3^{23} e retorna um elemento em F_3^{31};

Cada um desses circuitos é composto por um conjunto de sub-elementos que realizam a soma entre elementos em F_3. Um exemplo de um desses circuitos pode ser observado na Figura 7.5. Nessa figura vemos a soma entre dois elementos a e $b \in F_3^{16}$. O primeiro elemento do resultado $R \in F_3^{17}$ é o primeiro elemento de a e o último elemento de R é o último elemento de b. Esse mesmo esquema é utilizado nos outros somadores. Podemos observar na Figura 7.3 que as somas seguem um padrão de árvore invertida, e que a cada soma o elemento resultante é incrementado de 2^i elementos ao polinômio de entrada inicial, onde i é a profundidade da árvore, começando com 0.

Esse mesmo esquema poderia ter sido utilizado para a implementação do módulo de multiplicação especializada, porém o tamanho do circuito seria exorbitante.

Como já dito antes, devido às propriedades da soma de elementos em F_3, ou seja, por estarmos trabalhando em aritmética modular, não existe *carry* entre as somas de um elemento e outro, assim todas as somas de elementos em F_3 podem ser realizadas em paralelo.

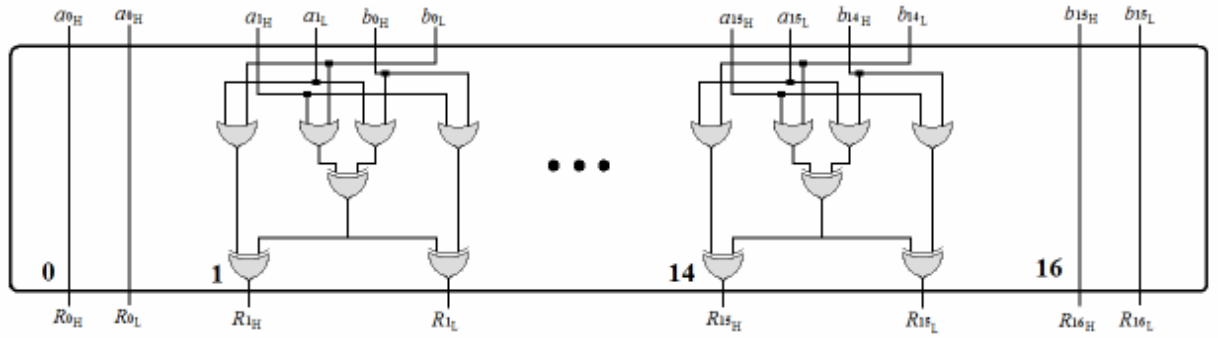

Figura 7.5: Circuito Lógico da Soma dos Produtos Parciais

7.1.3 Registradores Internos

Existem três grandes registradores internos na micro-arquitetura da instrução de multiplicação genérica. A posição em que eles se encontram pode ser observada na Figura 7.6. Os registradores 1 e 2 guardam o valor dos operando de entrada $a(x)$ e $b(x)$, ou seja, dois elementos em F_3^{16}. Já o registrador 3 guarda o valor do elemento obtido como resultado, ou seja um elemento em F_3^{31}. Após o término da operação, a parte menos significativa do registrador 3 é retornado para o processador e outra chamada a instrução é necessária para retornar a parte mais significativa ao processador. Além desses registradores, existem outros registradores menores que são utilizados para o controle da operação da multiplicação.

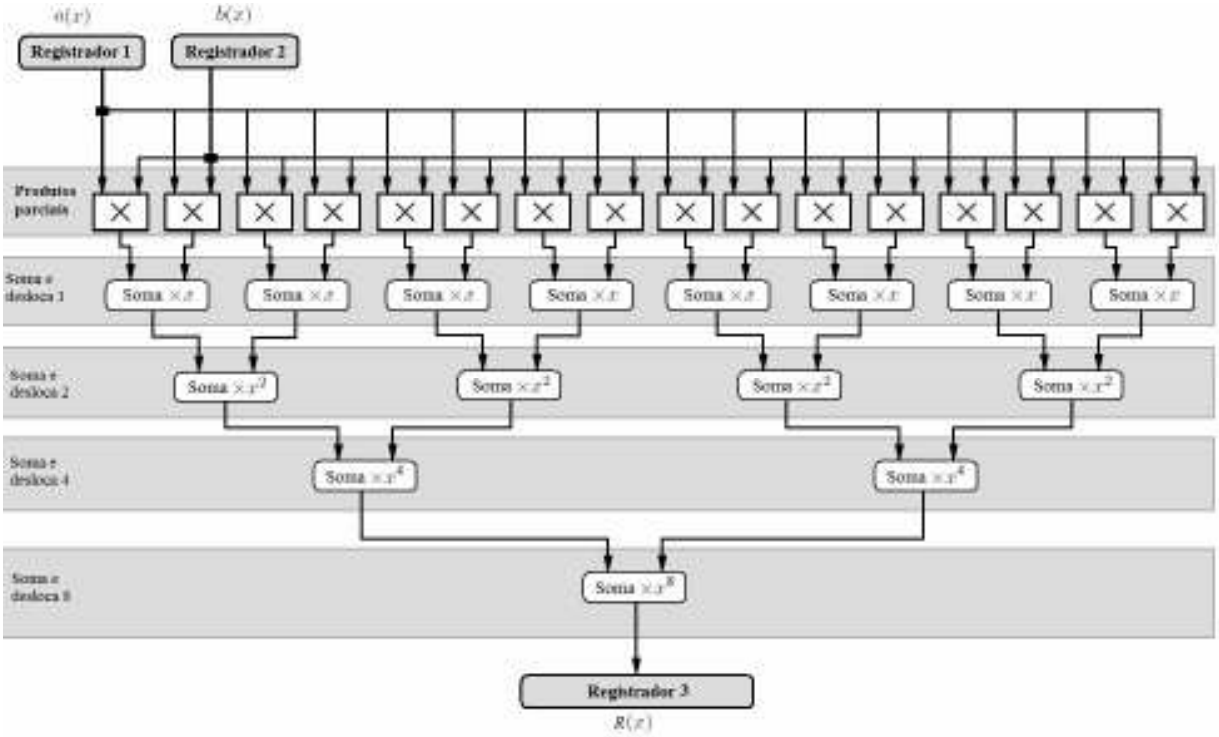

Figura 7.6: Registradores Internos da Instrução de Multiplicação Genérica

A Figura 7.7 apresenta o caminho crítico da instrução de multiplicação genérica.

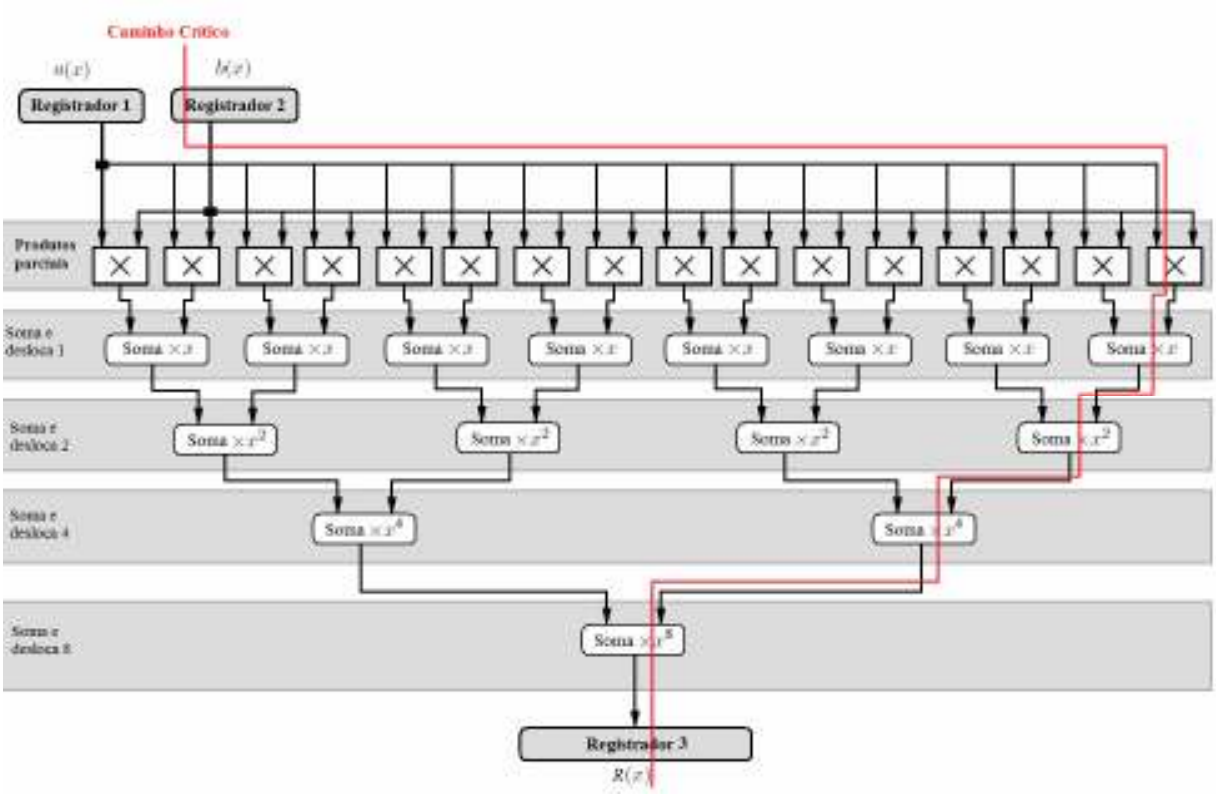

Figura 7.7: Caminho Crítico da Instrução de Multiplicação Genérica

7.2 Soma de Elementos em F_3^{16}

Essa instrução recebe como entrada dois elementos em F_3^{16} (representados por uma palavra de 32 bits) e retorna como resultado um elemento em F_3^{16} (representado por uma palavra de 32 bits). Tal instrução retorna penas 32 bits como resultado, assim é necessária apenas uma chamada à instrução para realizar a operação de soma. A operação realizada por essa instrução pode ser observado na Figura 7.8.

Em [11] é apresentada uma arquitetura parecida, que une a instrução de multiplicação genérica e a de soma em uma só. Contudo optamos por dividí-las em nossa implementação, para tornar a implementação mais genérica.

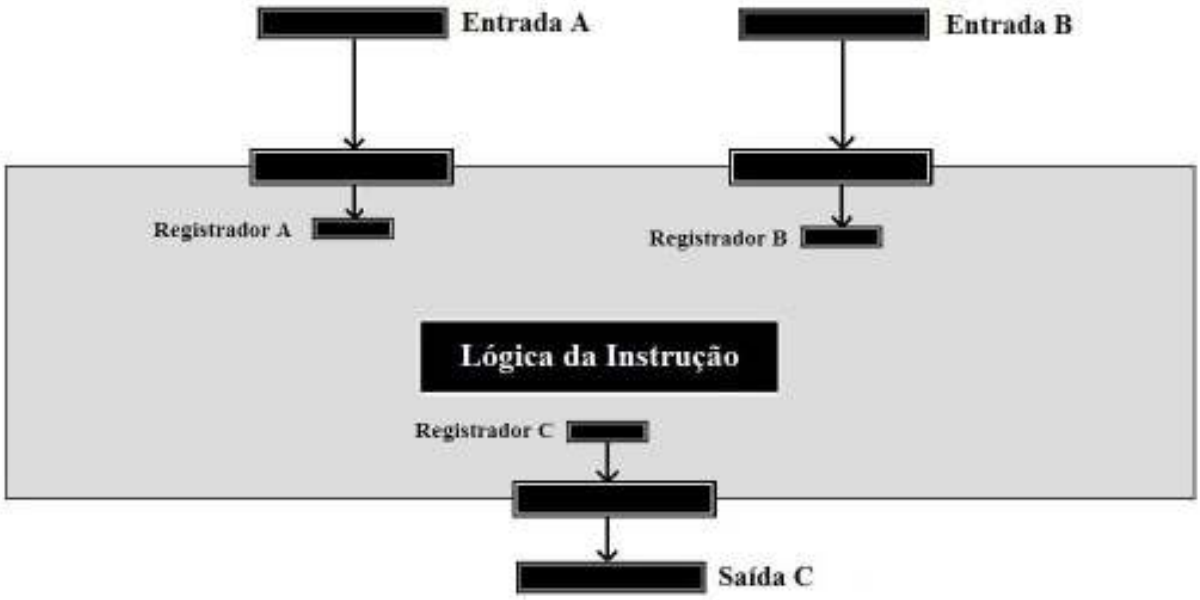

Figura 7.8: Instrução de Soma Genérica

O circuito de soma é composto por um conjunto de sub-elementos que realizam a soma entre elementos em F_3. Um exemplo desse circuito pode ser observado na Figura 7.9. Nessa figura vemos a soma entre um elemento $a \in F_3^{16}$ e um $b \in F_3^{16}$. Como resultado obtemos um elemento $R \in F_3^{16}$. Como já dito antes, devido às propriedades da soma de elementos em F_3, ou seja, por estarmos trabalhando em aritmética modular, não existe *carry* entre as somas de um elemento e outro, assim todas as somas de elementos em F_3 podem ser realizadas em paralelo.

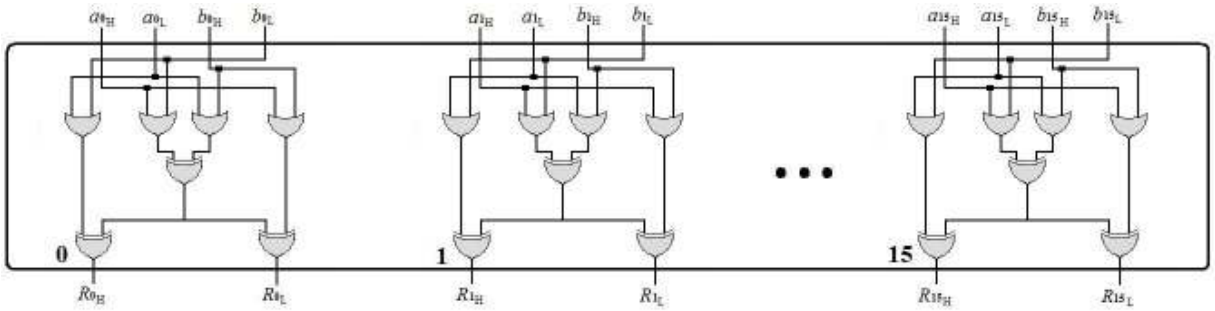

Figura 7.9: Circuito Lógico da Soma de Elementos em F_3^{16}

A Figura 7.10 apresenta o caminho crítico da instrução de soma genérica. Observe que nessa figura também estão presentes os registradores internos da instrução. São três registradores, os registradores 1 e 2 armazenam os elementos de entrada e o registrador 3 o elemento de saída.

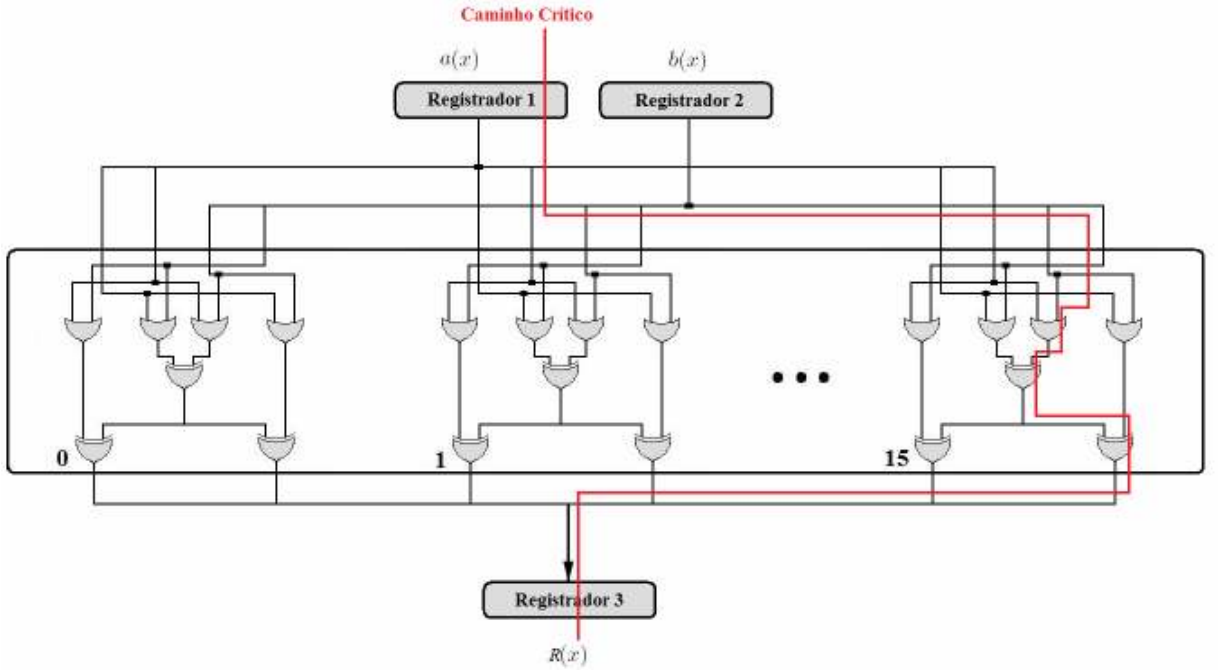

Figura 7.10: Caminho Crítico da Instrução de Multiplicação Genérica

7.3 Multiplicação em F_3^{97} Usando as Instruções Genéricas

A multiplicação em software usando instruções genéricas pode ser dividida em três fases. Durante a primeira fase, os produtos parciais são computados (podendo ser em paralelo), na segunda fase é realizada a soma dos produtos parciais (também podendo ser em paralelo) e finalmente, na última fase é realizada a redução pelo polinômio irredutível de grau *m*. Dependendo do tipo de implementação do algoritmo de multiplicação em software, a ordem em que ocorre cada fase poder ser alterada, ou mesmo ocorrer simultaneamente. A seguir vamos listar como essa multiplicação é realizada em nossa implementação usando o método de multiplicação *Schoolbook*.

7.3.1 Geração dos Produtos Parciais

Na primeira fase os dois elementos de entrada, *a(x)* e *b(x)* (ambos elementos em F_3^{97}), são subdivididos em 7 elementos em F_3^{16}, cada um representado por uma palavra de 32 bits (o último elemento é apenas um elemento em F_3, uma vez que 97 não é perfeitamente divisível por 16).

Cada um dos 7 elementos que compõem *b(x)* é multiplicado por cada elemento dos 7 elementos que compõem *a(x)*, gerando assim 49 produtos parciais (cada produto parcial é um elemento em F_3^{31}, menos nos casos de multiplicação envolvendo os últimos elementos, que são elementos em F_3. A multiplicação de um dos 7 elementos que formam *b(x)* por todo *a(x)* pode ser observada na Figura 7.11. Observe que nesse caso cada produto parcial é um elemento em F_3^{31} e é representado por duas palavras de 32 bits.

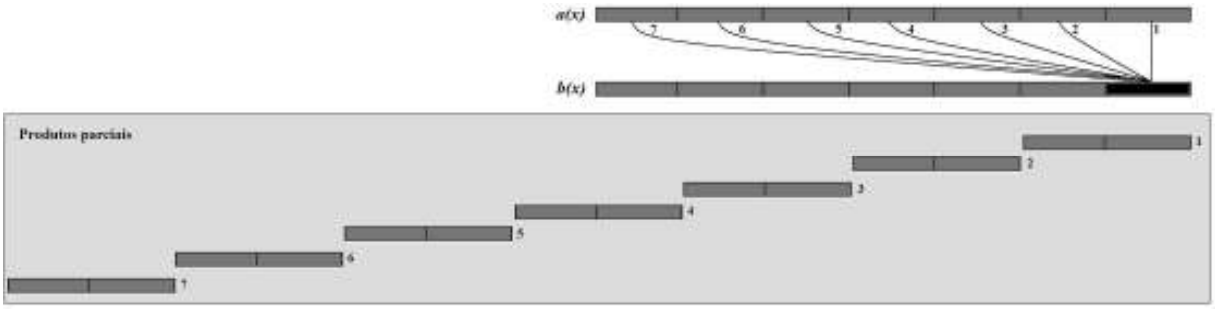

Figura 7.11: Multiplicação Usando Instruções Genéricas

Devido à irregularidade da divisão dos polinômios em palavras de 32 bits, três tipos de produtos parciais são obtidos:

- Multiplicação entre elementos em F_3^{16} – resulta em um elemento em F_3^{31}.
- Multiplicação entre um elemento em F_3^{16} e um elemento em F_3 – resulta em um elemento em F_3^{16}.
- Multiplicação entre dois elementos em F_3 – resulta em um elemento em F_3.

Independente do tipo de resultado obtido, todos eles são armazenados em palavras de 32 bits, que são o retorno da instrução de multiplicação genérica. No caso do resultado ser um elemento em F_3^{31}, temos uma palavra de 32 bits contendo os 16 elementos menos significativos em uma palavra e outra palavra de 32 bits contendo os 15 elementos mais significativos do polinômio.

7.3.2 Soma Produtos Parciais

Na segunda fase é realizada a soma dos produtos parciais gerados na primeira fase. Utilizando a instrução de soma genérica para realizar essa tarefa. Nessa fase o deslocamento intrínseco de cada produto parcial deve ser respeitado na hora de realizar a soma. Um esquema de como esses deslocamentos ficam está representado na Figura 7.12. Essa figura mostra que são gerados 49 produtos parciais e que devido à irregularidade da divisão dos polinômios em pedaços de 32 bits, três tipos de produtos parciais são obtidos (F_3^{31}, F_3^{16} e F_3). Os produtos que estão em uma mesma coluna vertical devem ser somados. A soma pode ser otimizada através

da utilização dos registradores do processador Nios II no armazenamento dos produtos parciais durante a realização das somas e apenas armazenando em memória o resultado final de cada posição.

O resultado da soma entre todos os produtos parciais é um elemento em F_3^{193}. É interessante observar que não é necessário usar deslocamentos para realizar as operações de soma, apenas a instrução de soma genérica é necessária. Um esquema de soma binária é utilizado para acelerar a soma dos produtos parciais.

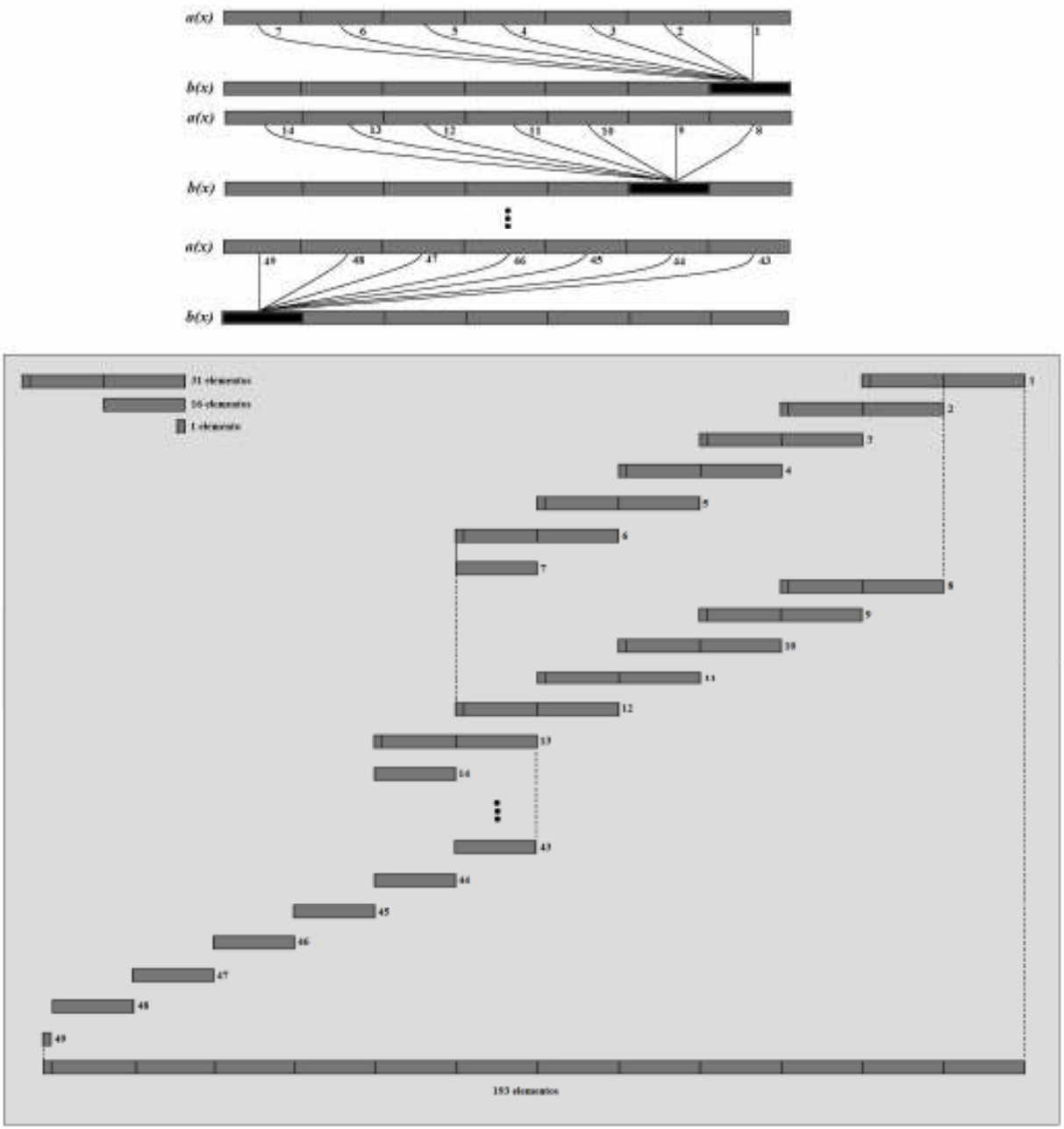

Figura 7.12: Soma dos Produtos Parciais Usando a Instrução Genérica

7.3.3 Redução pelo Polinômio Irredutível de Grau *m*

Finalmente, na última fase é realizada a redução pelo polinômio irredutível de grau *m*. Como o resultado da operação de soma dos produtos parciais gera um elemento em F_3^{193}, devemos reduzir esse elemento em 96 graus (ver algoritmo 5.3).

Capítulo 8

Comparação entre as Diferentes Abordagens

Duas métricas são usadas para avaliar as diferentes abordagens desenvolvidas (abordagem puramente software, abordagem hardware/software especializada e abordagem hardware/software genérica). A primeira métrica é o número de ALUTs utilizado em cada entidade implementada, esses números são apresentados na Tabela 8.1. Observando a tabela é possível constatar que a implementação da abordagem HW/SW necessita de uma pequena quantidade de hardware, quando comparada a abordagem puramente hardware (em [6] são gastos 14895 *slices* em em [7] 55616 *slices* na implementação da abordagem puramente hardware) e pouco hardware a mais que a abordagem puramente software (considerando apenas o tamanho do processador Nios II).

Entidade	Total de ALUTs
Processador Nios II	2293
Módulo de Multiplicação	1746
Instrução de Multiplicação mais Instrução de Soma Genérica	843
Instrução de Elevação ao Cubo	261

Tabela 8.1 – Número de ALUTs por Entidade de Hardware

A segunda métrica utilizada é o tempo necessário para o cálculo do emparelhamento, multiplicação e elevação ao cubo nas diferentes abordagens implementadas. Essas métricas são apresentadas na Tabela 8.2, esses valores consideram todos os tempos de atrasos (execução real), como tempos de transmissão dos barramentos. Nessa tabela podemos observar que o tempo necessário para calcular o emparelhamento em software, desprezando-se o tempo do cálculo da operação de multiplicação (aceleração ideal da multiplicação), cai para ≈2% quando comparado com a abordagem puramente software. Em outras palavras, o mapeamento do emparelhamento realizado pela ferramenta *Oprofile* é coerente com o resultado obtido no experimento.

Notamos ainda que o tempo necessário para calcular a operação de multiplicação usando um processador *Pentiun 4* é ≈10 vezes menor quando comparado ao tempo no processadorNios II e o tempo necessário para calcular o emparelhamento segue a mesma proporção. Ou seja, a melhoria de desempenho que obtivemos no processador Nios II seguirá a mesma proporção caso for empregada em processadores *high-end*.

Abordagem	**Emparelhamento η_T**	**Multiplicação**	**Elevação ao cubo**
Processador *Pentiun 4* à 2.4GHz (sem nenhuma otimização)	7×10^{2}	$8{,}779 \times 10^{-1}$	$1{,}96 \times 10^{-3}$
Processador Nios II à 50MHz (sem nenhuma otimização)	$7{,}299 \times 10^{3}$	8,656	$4{,}1 \times 10^{-2}$
Processador Nios II à 50MHz (usando o módulo de multiplicação especializado)	$2{,}172 \times 10^{2}$	$5{,}199 \times 10^{-2}$	$4{,}1 \times 10^{-2}$
Processador Nios II à 50MHz (usando o módulo de multiplicação e a instrução de elevação ao cubo especializados)	$2{,}147 \times 10^{2}$	$5{,}199 \times 10^{-2}$	$3{,}60 \times 10^{-2}$
Processador Nios II à 50MHz (usando a instrução de multiplicação e soma genéricas)	$3{,}024 \times 10^{2}$	$1{,}4899 \times 10^{-1}$	$4{,}1 \times 10^{-2}$
Processador Nios II à 50MHz (despresando-se o tempo gasto na multiplicação; essa é a aceleração ideal do sistema)	$1{,}484 \times 10^{2}$	0	$4{,}1 \times 10^{-2}$

Tabela 8.2 – Tempos de Execuções (em milissegundos)

A Tabela 8.3 apresenta a comparação da nossa implementação em hardware da multiplicação em F_3^{97} com diversas outras implementações encontradas na literatura. O multiplicador em *pipeline* apresentado em [27] é atualmente (na data em que esse trabalho foi realizado) o mais rápido encontrado na literatura, necessitando de apenas 11,467 ns para computar uma multiplicação em F_3^{97}.

É interessante notar que nossa implementação não está em formato de um *pipeline*. Caso nossa implementação venha a tornar-se um pipeline, é natural esperar que ocorra um ganho considerável de desempenho.

Multiplicadores	Plataforma	Ciclos	Tempo (ns)	Latência (ns)	Freqüência (MHz)	Área (Slices/ Aluts)
Bertoni *et al* [28]	Virtex II Pro	7	10.6	74.15	94.4	3, 561
Ronan *et al* [29]	Virtex II Pro	7	16.23	113.6	61.6	3, 737
Nosso trabalho	Stratix II	32	5.915	186.28	169	1746
Grabher *et al* [30]	Virtex II Pro	28	6.67	186.6	150	946
Beuchat *et al* [31]	Cyclone II	33	6.711	221.46	149	700
Kerings *et al* [7]	Virtex II Pro	25	34.129	853.22	29.3	1821

Tabela 8.3 – Comparações da Multiplicação em Hardware entre Diversos Trabalhos

Em [5] é apresentada uma arquitetura onde o cálculo da multiplicação e elevação ao cubo em F_3^{97} são realizados por apenas um circuito. Em nosso trabalho decidimos não optar por essa abordagem, buscando gerar circuitos menores e mais rápidos. Quando unimos os circuitos envolvidos nos cálculos de multiplicação e elevação ao cubo, acabamos gerando um circuito maior e mais complexo, sem dizer que fica inviável a comparação entre diferentes abordagens usando esse esquema. Além disto, desenvolvendo a técnica de *loop-unrolling* para implementar a elevação ao cubo, conseguimos chegar a uma implementação muito mais eficiente (realiza a elevação ao cubo em apenas um ciclo de *clock*).

Uma grande dificuldade encontrada na realização da comparação do nosso trabalho com outros trabalhos presentes na literatura foi a não homogeneidade das plataformas de trabalho, o que dificulta uma real comparação entre as diferentes implementações.

Outro problema encontrado é a forma de testar o sistema. Alguns autores apresentam como resultado apenas o valores mensurados a partir dos elementos em hardware, não apresentando os valores reais encontrados quando se une os elementos em hardware com o resto do sistema (possivelmente em software). Esses valores sim representam o verdadeiro desempenho do sistema, pois levam em conta todos os elementos envolvidos na computação. Em muitos casos, por mais rápida que seja a implementação, o resultado final não é satisfatório, pois algum elemento externo a implementação pode funcionar como um gargalo (tempo de acesso ao barramento, tempo de acesso à memória), caindo por terra o ganho de desempenho alcançado com a implementação.

Muitas implementações criam dispositivos em hardware específicos para acelerar partes integrantes da exponenciação final [38]. O nosso tipo de aceleração também pode ser utilizado na aceleração da exponenciação final, uma vez que ela também é composta por muitas multiplicações e elevações ao cubo.

Entre todos os trabalhos da literatura, talvez o que mais se aproxime do nosso trabalho (pelo menos em princípios) é o apresentado em [11]. Nesse trabalho Vejda *et al* expõem uma implementação de hardware/software genérica, onde o conjunto de instruções de um

processador de propósito geral (um processador LEON SPARC V8) é expandido através de instruções aritméticas customizadas.

Diferente do nosso trabalho, que se limita a computação do emparelhamento η_T em característica 3, as instruções em [11] investigam a criação de uma unidade aritmética integrada capaz de realizar operações em três diferentes campos (F_p, F_2^m e F_3^m). Ele utiliza a forma *Redundant Signed Digit* para representar os elementos dos três diferentes campos e implementa um *Multiply-Accumulate Unit* em *pipeline* para realizar a operação de multiplicação entre duas palavras de 32 bits.

Duas abordagens foram implementadas, uma com um multiplicador de (32 × 16) bits e outra menor com um multiplicador de (32 × 8) bits. Essas abordagens gastam respectivamente dois e quatro ciclos de *clock* para realizar a multiplicação entre duas palavras de 32 bits e necessitam de um ciclo de *clock* de 12 a 22 ns respectivamente (dependendo o tamanho da implementação), gerando assim um multiplicador capaz de trabalhar com freqüências de 45 a 83 MHz. A multiplicação é realizada utilizando o método *Comba* [42].

Em comparação, nossa instrução de multiplicação genérica realiza a multiplicação entre duas palavras de 32 bits em um ciclo de *clock* e necessita de um ciclo de *clock* de apenas 2 ns, possibilitando assim trabalhamos em uma freqüência de até 500 MHz, que é o *clock* interno máximo que a FPGA Stratix II suporta (ver Tabela 8.4).

Portando, considerando o tempo de execução, nossa instrução de multiplicação genérica entre duas palavras de 32 bits é 16 vezes mais eficiente que a melhor implementação apresentada em [11].

Entidade	**Período do *Clock* (ns)**	***Clock* Máximo (MHz)**
Módulo de Multiplicação	5,915	169
Instrução de Multiplicação Genérica	2	500 (Clock máximo da FPGA)
Instrução de Soma Genérica	2	500 (Clock máximo da FPGA)
Instrução de Elevação ao Cubo	3,220	310,56

Tabela 8.4 – Período do *Clock* e *Clock* Máximo por Entidade de Hardware (Stratix II)

Capítulo 9

Conclusões

Analisando os resultados obtidos, podemos notar que a abordagem HW/SW especializada conseguiu um ganho de desempenho de ≈3300%, sendo assim a melhor opção quando se necessita de mais desempenho, mantendo o baixo custo. Contudo, a abordagem HW/SW genérica fica pouco atrás, conseguindo um ganho de performance de ≈2400% usando um circuito ≈51% menor que a abordagem HW/SW especializada (considerando-se apenas a implementação da multiplicação em F_3^{97}). Um dos motivos dessa proximidade nos resultados é o fato do acesso ao módulo de multiplicação na abordagem HW/SW especializada ser por meio do barramento *Avalon* do processador Nios II, o que é custoso.

A implementação HW/SW genérica necessita apenas de ≈36% a mais de hardware para ser implementada quando comparado ao hardware de uma implementação puramente software (apenas o processador). Assim, podemos concluir que ambas as abordagens HW/SW apresentaram um ganho considerável de desempenho a um custo pequeno de implementação. Também concluímos que a abordagem HW/SW genérica é a melhor candidata a implementação em dispositivos de baixo custo, uma vez que apresenta um ganho satisfatório de desempenho a um baixo custo de implementação e ainda apresenta grande flexibilidade para acomodar mudanças nos algoritmos criptográficos.

Um fato importante que os resultados expuseram, é que a instrução de elevação ao cubo não conseguiu contribuir de forma significativa para a melhora da execução do emparelhamento. Isso se deve ao fato da implementação em software dessa operação já ser otimizada, assim a instrução customizada pouco teve a ganhar.

Um ponto a se ponderar é o fato de que os resultados obtidos podem servir de exemplo de como otimizar um sistemas computacional onde grande parte da computação é gasto fazendo-se poucas operações (a maioria das aplicações possui essa característica).

Os resultados também expuseram os princípios da Lei de *Amdahl*, uma vez que o tempo de execução final dos emparelhamentos acelerados por hardware acabou sendo dominado pelo tempo de execução da parte não otimizada do emparelhamento. Em outras palavras, o limite para a aceleração de um sistema otimizado da forma HW/SW é o tempo gasto na computação da parte não otimizada.

Capítulo 10

Trabalhos Futuros

Nesta dissertação vislumbramos o ganho de desempenho que se pode obter quando se implementa em hardware partes críticas de um sistema em software. Em nosso caso um emparelhamento η_T sobre o corpo finito F_3^{97} em uma plataforma de baixo custo foi utilizado. Contudo, essa abordagem poderia ser empregada em outros sistemas, inclusive em sistemas *high-end*. Uma contribuição importante seria aplicar as técnicas de aceleração usando as abordagens hardware/software para acelerar uma aplicação em um ambiente *high-end* e posteriormente comparar os resultados obtidos.

Outra contribuição interessante seria a implementação das abordagens totalmente em hardware e em software otimizada do emparelhamento, utilizando para isso a mesma plataforma empregada na realização deste trabalho, criando um ambiente ideal para a comparação entre as diversas abordagens existentes.

Embora, na atualidade, a utilização de F_3^{97} seja aceitável na grande maioria dos casos, devemos ter em mente a crescente demanda por maiores níveis de segurança, o que pode ser traduzido em chaves e parâmetros de maior tamanho. Assim sendo, uma expansão imediata do trabalho seria tratar a implementação e análise dos mesmos algoritmos para diferentes tamanhos de corpos binários.

Os módulos e instruções *customizadas* desenvolvidas neste trabalho podem sofrer um grande aumento de desempenho se a abordagem de *pipeline* for empregada em sua implementação, gerando assim a possibilidade de explorar o paralelismo existente no emparelhamento.

A instrução de multiplicação genérica poderia ser modificada para funcionar como um multiplicador com acumulador interno, facilitando assim a implementação do esquema de multiplicação *Comba* para a multiplicação em F_3^m, e conseqüentemente alcançando um melhor desempenho.

Neste trabalho apenas as avaliações de tempo e área do sistema foram considerados. Um grande complemento do trabalho seria gerar as análises de consumo de potência de cada uma

das abordagens, bem como estudar a resistência a *side-channel attacks* de cada uma das abordagens implementadas.

Outra contribuição interessante seria a implementação de um sistema de comunicação que utiliza Criptografia Baseada em Identidades para prover segurança ao meio de comunicação. Para tanto, tal sistema seria prototipado em um FPGA e usaria um processador com o hardware *customizado* desenvolvidos neste trabalho para acelerar a comunicação. Tal plataforma possibilitaria o estudo e desenvolvimento de técnicas empregadas à Criptografia Baseada em Identidades no gerenciamento de uma aplicação real, assim como a análise de desempenho das abordagens implementadas nesse trabalho em uma aplicação real.

Referências Bibliográficas

[1] W. Diffie, M. E. Hellman, "New Directions in Cryptography," IEEE Transactions on Information Theory, vol. IT-22, n. 6, pp. 644-654, Nov. 1976.

[2] A. Shamir, "Identity-based cryptosystems and signature schemes," in Proc. of the Crypto'84 on Advances in Cryptology, 1985, p. 47-53.

[3] W. D. Benit, R. Terada "Sistemas criptográficos baseados em identidades pessoais," Dissertação apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo para obtenção do grau de Mestre em Ciência da Computação, São Paulo - novembro - 2003.

[4] D. Boneh, M. Franklin, "Identity-Based Encryption from the Weil Pairing," SIAM Journal on Computing, vol. 32, n. 3, pp. 586-615, 2003.

[5] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, M. Shirase, T. Takagi. "Algorithms and arithmetic operators for computing the η_T pairing in characteristic three," IEEE Transactions on Computers, vol. 57, n. 11, pp. 1454-1468, Nov. 2008.

[6] J.-L. Beuchat, M. Shirase, T. Takagi, E. Okamoto, "An algorithm for the η_T pairing calculation in characteristic three and its hardware implementation," in Proc. of the 18th IEEE Symposium on Computer Arithmetic, 2007, p. 97–104.

[7] T. Kerins, W. P. Marnane, E. M. Popovici, P. S. L. M. Barreto, "Efficient hardware for the Tate pairing calculation in characteristic three," in Proc. of the Cryptographic Hardware and Embedded Systems – CHES'05, 2005, p. 412–426.

[8] O. Ahmadi, D. Hankerson, A. Menezes "Software implementation of arithmetic in GF(3^m)," in Proc. of the 1st international Workshop on Arithmetic of Finite Fields, vol. 4547, pp. 85-102, Jun. 2007.

[9] J.-L. Beuchat, E. López-Trejo. L. Martínez-Ramos, S. Mitsunari, F. Rodríguez-Henríquez, "Multi-core implementation of the Tate pairing over supersingular elliptic curves," in Cryptology ePrint Archive, 2009, Report 2009/276.

[10] Y. Kawahara, T. Takagi, E. Okamoto, “Efficient implementation of Tate pairing on a mobile phone using Java,” in Proc. of the CIS 2006, NAI 4456, 2007, p. 396–405.

[11] T. Vejda, D. Page, J. Großschädl,“Instruction Set Extensions for Pairing-Based Cryptography,” in Proc. of the Pairing 2007, 2007, p. 208-224.

[12] A. J. Menezes, P.C. Van Oorschot, and S.A. Vanstone, “Handbook of Applied Cryptography,” CRC Press, October 1996.

[13] D. Hankerson, A.J. Menezes, and S. Vanstone, “Guide to Elliptic Curve Cryptography,” Springer-Verlag, 1st edition, January 2004.

[14] N. Koblitz, “Elliptic curve cryptosytems,” in Mathematics of Computation, v. 48, n. 177, p. 203-209, 1987.

[15] M. Juliato, G. Araujo, J. López, “Especialização de Arquiteturas para Criptografia em Curvas Elípticas,” Dissertação apresentada ao Instituto computação da Universidade Estadual de Campinas para obtenção do grau de Mestre em Ciência da Computação, Campinas - Agosto - 2006.

[16] P. Barreto, “Curvas Elípticas e Criptografia: Conceitos e Algoritmos,” 1999. Disponível em http://www.larc.usp.br/~pbarreto/my_publications.html, 2009.

[17] D. H. Goya, R. Terada, “Proposta de Esquama de Criptografia e de Assinatura sob Modelo de Criptografia de Chave Publica sem Certificado,” Dissertação apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo para obtenção do grau de Mestre em Ciência da Computação, São Paulo - junho - 2006.

[18] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott, “Efficient algorithms for pairing-based cryptosystems,” in Advances in Cryptology – CRYPTO 2002, ser. Lecture Notes in Computer Science, M. Yung, Ed., no. 2442. Springer, 2002, pp. 354–368.

[19] P. S. L. M. Barreto, S. D. Galbraith, C. ´O h ´ Eigeartaigh, and M. Scott, “Efficient pairing computation on supersingular Abelian varieties,” in Designs, Codes and Cryptography. Springer Netherlands, Mar. 2007, vol. 42(3), pp. 239–271.

[20] Altera Nios II processor website. Disponível em: http://www.altera.com/products/ip/processors/nios2/ni2-index.html, 2009.

[21] Altera Stratix II FPGA website. Disponível em: http://www.altera.com/products/devices/stratix-fpgas/stratix-ii/stratix-ii/st2-index.jsp, 2009.

[22] Altera Quartus II Subscription edition software website. Disponível em: http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html, 2009.

[23] K. Harrison, D. Page, and N.P. Smart, "Software implementation of finite fields of characteristic three, for use in pairing-based cryptosystems," in LMS Journal of Computation and Mathematics, 2002, 5:181-193.

[24] J.-L. Beuchat et al, "FPGA and ASIC implementations of the η_T Pairing in Characteristic Three," in Cryptology ePrint Archive, 2008, Report 2008/280.

[25] Oprofile website. Available: http://oprofile.sourceforge.net/about/, 2009.

[26] M. Juliato, G. C. S. Araujo, J. C. López, R. Dahab, "Custom Instruction Approach for Hardware and Software Implementations of Finite Field Arithmetic over F2163 using Gaussian Normal Bases," in Proc. International Conference on Field-Programmable Technology 2005 – FPT'05, 2005, p. 5-12.

[27] N. Cortez-Duarte, F. Rodríiguez-Henríquez, J.-L. Beuchat, E. Okamoto, "A pipelined Karatsuba-Ofman multiplier over GF(397)," in Cryptology ePrint Archive, 2008, Report 2008/127.

[28] G. Bertoni, J. Guajardo, S. S. Kumar, G. Orlando, C. Paar, T. J. Wollinger. "Efficient GF(p^m) arithmetic architectures for cryptographic applications," In Topics in Cryptology - CT RSA 2003, 2003, p. 158–175.

[29] R. Ronan, C. Murphy, T. Kerins, C. Ó' Héigeartaigh, P. S. L. Barreto, "A flexible processor for the characteristic 3 η_T pairing," International Journal of High Performance Systems Architecture, vol. 1, n. 2, pp. 79-88, 2007.

[30] P. Grabher, D. Page, "Hardware acceleration of the Tate pairing in characteristic three," In Proc. of the Cryptographic Hardware and Embedded Systems – CHES'05, 2005, p. 398–411.

[31] J.-L. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, "Arithmetic operators for pairing-based cryptography," In Proc. of the 9th international workshop on Cryptographic Hardware and Embedded Systems – CHES'07, 2007, p. 239-255.

[32] R. Granger, D. Page, and M. Stam, “On small characteristic algebraic tori in pairing-based cryptography,” LMS Journal of Computation and Mathematics, vol. 9, pp. 64–85, Mar. 2006.

[33] J. von zur Gathen and M. Nocker, “Computing special powers in finite fields,” Mathematics of Computation, vol. 73, no. 247, pp. 1499–1523, 2003.

[34] F. Rodríguez-Henríquez, G. Morales-Luna, N. A. Saqib, and N. Cruz-Cortés, “A parallel version of the Itoh-Tsujii multiplicative inversion algorithm,” in Reconfigurable Computing: Architectures, Tools and Applications – Proceedings of ARC 2007, ser. Lecture Notes in Computer Science, P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. M. P. Cardoso, Eds., no. 4419. Springer, 2007, pp. 226–237.

[35] D. E. Knuth, “The Art of Computer Programming,” 3rd ed. Addision-Wesley, 1998, vol. 2, Seminumerical Algorithms.

[36] GNU Multiple Precision Arithmetic Library website Disponível em: http://gmplib.org/, Report 05/03/2009, 2009.

[37] P. Barreto and H. Kim, “Fast Hashing onto Elliptic Curves over Fields of Characteristic 3,” in Cryptology ePrint Archive, 2001, Report 2001/098.

[38] J.-L. Beuchat, N. Brisebarre, M. Shirase, T. Takagi and E. Okamoto, “A Coprocessor for the Final Exponentiation of the η_T Pairing in Characteristic Three,” in Proceedings of the 1st international workshop on Arithmetic of Finite Fields, 2007, pp. 25-39.

[39] A. Joux, “A one round protocol for tripartite Diffie-Hellman,” in Algorithmic Number Theory – ANTS IV, ser. Lecture Notes in Computer Science, W. Bosma, Ed., no. 1838. Springer, 2000, pp. 385–394.

[40] I. Duursma and H. S. Lee, “Tate pairing implementation for hyperelliptic curves $y^2 = x^p - x + d$,” in Advances in Cryptology – ASIACRYPT 2003, 2003, pp. 111–123.

[41] E. Gorla, C. Puttmann and J. Shokrollahi, “Explicit formulas for efficient multiplication in $F_3{}^{6m}$,” in Selected Areas in Cryptography – SAC 2007, 2007, pp. 173–183.

[42] P. G. Comba, “Exponentiation cryptosystems on the IBM PC,” IBM Systems Journal 29(4), 1990, pp 526–538.