Especialização de Arquiteturas para Criptografia em Curvas Elípticas

Marcio Rogério Juliato

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Especialização de Arquiteturas para Criptografia em Curvas Elípticas

Marcio Rogério Juliato[1]

Julho de 2006

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Manoel Eusébio de Lima
  Centro de Informática
  Universidade Federal de Pernambuco - UFPE
- Prof. Dr. Ricardo Dahab
  Departamento de Teoria de Computação
  Instituto de Computação
  Universidade Estadual de Campinas - UNICAMP
- Prof. Dr. Paulo Cesar Centoducatte (Suplente)
  Departamento de Sistemas de Computação
  Instituto de Computação
  Universidade Estadual de Campinas - UNICAMP

[1]**Trabalho financiado pela FAPESP (Processo 03/11674-0) e pela Intel Corporation**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Juliato, Marcio Rogério

J942e Especialização de arquiteturas para criptografia em curvas elípticas / Marcio Rogério Juliato -- Campinas, [S.P. :s.n.], 2006.

Orientadores : Guido Costa Souza de Araújo; Julio César López Hernández

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Criptografia de chave-pública. 3. Field programmable gate arrays. I. Araújo, Guido Costa Souza de. II. López Hernández, Julio César. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

(mjmr/imecc)

Título em inglês: Architecture specialization for elliptic curve cryptography.

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Public key cryptography. 3. Field programmable gate arrays.

Área de concentração: Arquitetura e Sistemas de Computação

Titulação: Mestre em Ciência da Computação

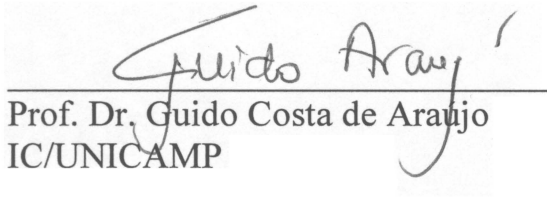
Banca examinadora: Prof. Dr. Manoel Eusébio de Lima (CIn-UFPE)
Prof. Dr. Ricardo Dahab (IC-UNICAMP)
Prof. Dr. Paulo Cesar Centoducatte (IC-UNICAMP)
Prof. Dr. Guido Costa Souza de Araújo (IC-UNICAMP)
Prof. Dr. Julio César López Hernández (IC-UNICAMP)

Data da defesa: 08/08/2006

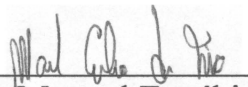
Programa de Pós-Graduação: Mestrado em Ciência da Computação

Termo de Aprovação

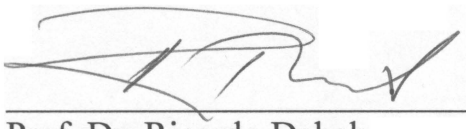
Dissertação defendida e aprovado em 08 de agosto de 2006, pela Banca Examinadora composta pelos Professores Doutores:



Prof. Dr. Guido Costa de Araújo
IC/UNICAMP



Prof. Dr. Manoel Eusébio de Lima
CIN/UFPE



Prof. Dr. Ricardo Dahab
IC/UNICAMP

Especialização de Arquiteturas para Criptografia em Curvas Elípticas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcio Rogério Juliato e aprovada pela Banca Examinadora.

Campinas, 08 de Agosto de 2006.

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

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

Resumo

O aumento na comunicação utilizando-se sistemas eletrônicos tem demandado a troca de informações cifradas, permitindo a comunicação entre dois sistemas desconhecidos através de um canal inseguro (como a Internet). Criptografia baseada em curvas elípticas (ECC) é um mecanismo de chave pública que requer apenas que as entidades, que desejam se comunicar, troquem material de chave que é autêntico e possuem a propriedade de ser computacionalmente infactível descobrir a chave privada somente com informações da chave pública. A principal operação de sistemas ECC é a multiplicação de ponto (kP) que gasta 90% de seu tempo de execução na multiplicação em corpos finitos. Assim, a velocidade de um sistema ECC é altamente dependente do desempenho das operações aritméticas em corpos finitos. Nesse trabalho, estudamos a especialização de um processador NIOS2 para aplicações criptográficas em curvas elípticas. Mais precisamente, implementamos operações em corpos finitos e a multiplicação de pontos sobre $\mathbb{F}_{2^{163}}$ como instruções especializadas e periféricos do NIOS2, e as analisamos em termos de área e *speedup*. Determinamos também, quais implementações são mais apropriadas para sistemas voltados a desempenho e para ambientes restritos. Nossa melhor implementação em *hardware* da multiplicação de pontos é capaz de acelerar o cálculo de kP em 2900 vezes, quando comparado com a melhor implementação em *software* executando no NIOS2. De acordo com a literatura especializada, obtivemos a mais rápida implementação da multiplicação de pontos sobre $\mathbb{F}_{2^{163}}$, comprovando que bases normais Gaussianas são bastante apropriadas para implementações em *hardware*.

Abstract

The increase in electronic communication has lead to a high demand for encrypted information exchange between unfamiliar hosts over insecure channels (such as the Internet). Elliptic curve cryptography (ECC) is a public-key mechanism that requires the communicating entities exchange key material that is authentic and has the property of being computationally infeasible to determine the private key from the knowledge of the public key. The fundamental ECC operation is the point multiplication (kP), which spends around 90% of its running time in the finite field multiplication. Therefore, the speed of an ECC scheme is highly dependent on the performance of its underlying finite field arithmetic. In this work, we studied the specialization of the NIOS2 processor for ECC applications. More precisely, we implemented the finite field operations and the point multiplication over $\mathbb{F}_{2^{163}}$ as NIOS2 custom instructions and peripherals, and thus, we analyzed them in terms of area and speedup. We also determined which implementations are best suited for performance-driven and area-constrained environments. Our best hardware implementation of the point multiplication is capable of accelerating the kP computation in 2900 times, when compared to the best software implementation running in the NIOS2. According to the literature, we obtained the fastest point multiplier in hardware over $\mathbb{F}_{2^{163}}$, proving that Gaussian normal bases are quite appropriate for hardware implementations.

Epígrafe

I do not know what I may appear to the world,
but to myself I seem to have been only like a boy playing on the sea-shore,
and diverting myself in now
and then finding a smoother pebble or a prettier shell than ordinary,
whilst the great ocean of truth
lay all undiscovered
before me.

Isaac Newton (1642-1727)

Dedicatória

Dedico esse trabalho a todas as pessoas que não puderam perseguir seus sonhos e atingir seus objetivos. Não por falta de capacidade ou motivação, dado que uma infinidade daquelas são certamente muito mais merecedoras que eu. Mas que não puderam ser felizes, viver dignamente e em paz, bem como estudar e trabalhar, por razões de guerras, despotismo, problemas sócio-econômicos, falta de saúde e alimentação adequadas, problemas de acessibilidade, catástrofes naturais, perseguições e discriminações em suas mais variadas formas, etc.

Agradecimentos

Meus mais sinceros agradecimentos aos meus pais, companheiros de todos os meus momentos, por todo o carinho, amor, dedicação, e pela infinita compreensão. Agradeço também pela confiança quando tudo era dúvida, e o futuro extremamente incerto. Saibam de minha admiração pelo jeito que fui criado, pelo ambiente de respeito e aprendizado em que estive imerso desde quando me conheço por gente. Meus mais profundos agradecimentos por terem fomentado minha curiosidade desde minha infância, e minha enorme gratidão pelo apoio financeiro e psicológico durante todos esses anos, sem os quais eu não poderia ter seguido meus sonhos e alcançado meus objetivos. Agradeço também por terem me ensinado a seguir a princípios e ter referências sólidas, por terem me apresentado à luz quando tudo era escuridão, e por terem me levantado em todas as vezes que caí. A vocês meus queridos, minha admiração, meu infinito respeito e amor, e eterna gratidão.

Meus profundos agradecimentos à minha amada esposa Rita, por todo amor, confiança, carinho e respeito. Considero-me uma pessoa de sorte por ter encontrado alguém tão especial e tão forte. Alguém que me completa e me faz ver o mundo de maneira mais humana, mais solidária, e menos egoísta. Minha gratidão por ter me compreendido e apoiado em todos os inúmeros momentos difíceis, por ter estado ao meu lado em todas as adversidades e batalhas travadas no passado, e por enfrentar de frente e sem medo os enormes desafios que vêm pela frente. A você, meu imenso amor, minha admiração e enorme gratidão e respeito.

Agradeço sinceramente às minhas irmãs Vanessa e Denise pela enorme compreensão e amizade. Minha gratidão por terem estado incondicionalmente do meu lado tanto nos momentos felizes quanto nos tristes. Obrigado por se orgulharem de minhas vitórias, e por terem sempre torcido por mim. Saibam que também me orgulho muito de ter irmãs como vocês e que sempre torcerei pelas suas felicidades, vitórias e sucesso.

Minha gratidão aos meus familiares que sempre se interessaram e se orgulharam de minhas conquistas, e que torceram por mim em todos os momentos. Meu muito obrigado aos muitos amigos (seria difícil citar todos os nomes aqui), pelas conversas, dicas, ajudas, trocas de idéias, risadas, e também pelas discordâncias. Pela companhia nos laboratórios, nas (infindáveis) horas de estudo na biblioteca e (intermináveis) listas de exercícios, e

pelos divertidos almoços no bandejão. Pelas competições e torneios que nos aventuramos, alguns vencemos, outros nada ganhamos mas muito aprendemos. Saibam que cresci muito com vocês, e agradeço pela convivência de todos nesses anos.

Agradeço ao professor Guido Araujo pela excelente orientação em meu mestrado, e por ter me mostrado o horizonte quando eu enxergava apenas um palmo adiante. Obrigado ainda por ter me apresentado diversas oportunidades interessantes, pelas trocas de idéias e dicas, e por ter confiado em mim, em meus conhecimentos e em minhas habilidades. Meus agradecimentos ao professor Paulo Centoducatte por ter amadurecido minha forma de pensar, e pela sabia orientação em nada menos que três iniciações científicas durante minha graduação. Obrigado também por ter confiado e me apoiado incondicionalmente. Agradeço ao professor Julio López pela brilhante co-orientação em meu mestrado, pela visão científica, pelas valiosas trocas de idéias e dicas, pela amizade criada nesse período de tempo em que trabalhamos juntos, e pelas tradicionais conversas de segunda-feira à tarde. Obrigado também ao professor Ricardo Dahab pelo apoio, pelas dicas, e por ter me auxiliado nos primeiros passos de minha caminhada no mundo da criptografia. A vocês, meus agradecimentos, minha admiração e meu mais profundo respeito.

Minha gratidão à Unicamp, meu maravilhoso sonho que se tornou realidade, por todo o imenso suporte e oportunidades nos meus anos de graduação em engenharia de computação e de mestrado em ciência da computação. Por ter aprimorado meus interesses e conhecimentos em ciência e engenharia. Por me mostrar o quão pequeno eu pensava, quão ignorante eu era, a infinidade de coisas ainda tenho de aprender, ou seja, por me apresentar ao infinito mundo do conhecimento. Obrigado também por ter me ensinado a aprender. Meus agradecimentos ao IC e à FEEC, pela magnífica estrutura de ensino, salas de aula, laboratórios, equipamentos, professores, e pessoal de apoio. Meus agradecimentos ao LSC, por ter me acolhido nesses últimos sete anos, e ter me provido de uma incomparável plataforma de trabalho em termos físicos, financeiros, e sociais. Devo dizer que é motivo de muito orgulho ter integrado esse laboratório.

Minha enorme gratidão a todas as instituições de ensino por que passei, especialmente à EMEI-Tangará, ao SESI, ao SENAI, à Politec, e à Universität Duisburg-Essen, por terem contribuído para minha formação, dando-me os alicerces para que eu pudesse chegar onde estou. Estendo meus agradecimentos a todos os mestres que solidificaram meu aprendizado e que me guiaram no mundo do saber, desde aqueles que me ensinaram a mexer com cola e papel, a escrever e a ler, a trabalhar com madeira, argila e eletricidade, a falar outras línguas, a passar no vestibular, a ter raciocínio lógico, até os que me ensinaram a admirar a beleza da natureza e a complexidade do universo.

Por fim, agradeço à FAPESP e à Intel por fomentar esse trabalho de mestrado e torná-lo possível. Agradeço também ao CNPq e DAAD que também contribuíram para minha formação anterior, provendo assim, bases para a concretização desse trabalho.

Abreviações e Notações Utilizadas

$\cdot$: Multiplicação em corpos finitos
$\odot$: Operação E (*And*)
$\oplus$: Operação OU-Exclusivo (*Xor*)
$\gg$: Rotação à direita
$\ll$: Rotação à esquerda
CI : Instrução Especializada. Do inglês *Custom Instruction*
CP : Periférico do Processador NIOS2. Do inglês *Custom Peripheral*
ECC : Do inglês *Elliptic Curve Cryptography*
ECDLP : Do inglês *Elliptic Curve Discrete Logarithm Problem*
ECDSA : Do inglês *Elliptic Curve Digital Signature Algorithm*
$\mathbb{F}_{2^m}$: Corpo finito binário com 2^m elementos
FPGA : Do inglês *Field Programmable Gate Array*
GNB: Do inglês *Gaussian normal basis*
HW : *Hardware*
kP : Multiplicação de Ponto (ou Multiplicação Escalar)
NB: Do inglês *Normal basis*
NIST : Do inglês *National Institute of Standards and Technology*
ONB: Do inglês *Optimal normal basis*
PB: Do inglês *Polynomial basis*
RFID: Do inglês *Radio Frequency Identification*
SOC : Do inglês *System-On-Chip*
SOPC : Do inglês *System-On-a-Programmable-Chip*
SSL : Do inglês *Secure Sockets Layer*
SW : *Software*
ULA : Unidade Lógica Aritmética
VHDL : Do inglês *VHSIC Hardware Description Language*
WEP : Do inglês *Wired Equivalent Privacy*

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com o crescente aumento nas comunicações utilizando-se sistemas eletrônicos tornou-se comum os ataques de vírus e *hackers*, como também de fraudes e espionagens eletrônicas. Nesse contexto, que envolve intensa conectividade, organizações e indivíduos têm demandado maior segurança para seus dados (que podem estar trafegando por algum meio de comunicação). Como resultado, a comunicação entre dois sistemas desconhecidos através de canais inseguros, como por exemplo a Internet, tem demandado a troca de informações cifradas.Para ambientes restritos como *Smart-Cards*, RFIDs, aplicações sem fio e móveis, além da segurança, faz-se necessário também considerar outros fatores como o custo de implementação e sua relação de compromisso em termos de velocidade, potência e área de modo a obter sistemas mais baratos e que possuam grande longevidade de suas baterias.

O principal objetivo de criptografia é fazer com que duas pessoas, usualmente nomeadas na literatura como Alice e Bob, possam se comunicar usando um canal inseguro de forma que um oponente, Malice, não possa entender o que está sendo dito. Esse canal pode ser, por exemplo, uma linha telefônica ou uma rede de computadores. A informação que Alice envia para Bob, nomeada *plaintext*, pode ser um texto em português, um dado numérico, ou qualquer outra coisa. Alice cifra o *plaintext*, usando uma chave pré-determinada e envia o resultado, *ciphertext*, pelo canal. Por espionagem, Malice pode ler o canal, mas não pode obter o *plaintext*. Bob, que também possui a chave de cifra-mento, pode decifrar o *ciphertext* e reconstruir o *plaintext*. Esse esquema é conhecido como sistema criptográfico de chaves simétricas.

Criptografia de chave pública foi proposta em 1976 por Whitfield Diffie e Martin Hellman [1]. Em contraste com criptografia de chaves simétricas, sistemas criptográficos de chaves públicas como o RSA [7, 25] requerem apenas que as entidades, que desejam se comunicar, troquem material de chave que é autêntico (mas não secreto). Cada entidade seleciona um par de chaves matematicamente relacionadas, denominadas chave pública e privada, que possuem a propriedade de ser computacionalmente infactível descobrir a

chave privada somente com informações da chave pública. Além disso, assinaturas digitais e algoritmos de estabelecimento de chaves têm sido utilizados para estabelecer canais cifrados. Mecanismos padrão para segurança de redes, como por exemplo, os protocolos IPSec, *Secure Sockets Layer* (SSL), e *Wired Equivalent Privacy* (WEP), também contam, de alguma forma, com criptografia de chave pública.

Criptografia baseada em curvas elípticas (ECC) é um mecanismo de chave pública, proposto independentemente por Victor Miller [2] e Neal Koblitz [3], o qual provê as mesmas funcionalidades do conhecido RSA. A segurança de sistemas utilizando ECC é baseada no problema do Logaritmo Discreto em Curvas Elípticas (ECDLP) [27], onde o melhor algoritmo que se conhece para resolver esse problema possui tempo de execução exponencial. Parâmetros significantemente menores podem ser usados em ECC, quando comparados com sistemas semelhantes tais como o RSA e o DSA; por exemplo, uma chave de 160 bits de um sistema ECC garante um mesmo nível de segurança que um sistema RSA utilizando uma chave de 1024 bits. Sistemas ECC podem ser implementados de forma eficiente e com pequena área, principalmente para domínios bastante restritos como *Smart-Cards*, RFIDs, e pequenos sensores.

O algoritmo de assinatura digital baseados em curvas elípticas (ECDSA) é constituído de uma plataforma que conta com um gerador de números aleatórios, uma função *hash*, aritmética modular e de números grandes, como mostrado na Figura 1.1. A aritmética de curva é a parte mais cara da computação do ECDSA. Em particular, a operação fundamental de ECC é a multiplicação de ponto (ou multiplicação escalar) kP, sendo responsável por aproximadamente 80% do tempo de execução do ECDSA. No caso de kP, k é um inteiro não-negativo e P um ponto na curva elíptica. Analisando a execução da multiplicação de ponto, percebe-se que em torno de 90% de seu tempo de execução é gasto na multiplicação em corpos finitos. Assim, a velocidade de um esquema de ECC é altamente dependente do desempenho das operações aritméticas em corpos finitos binários.

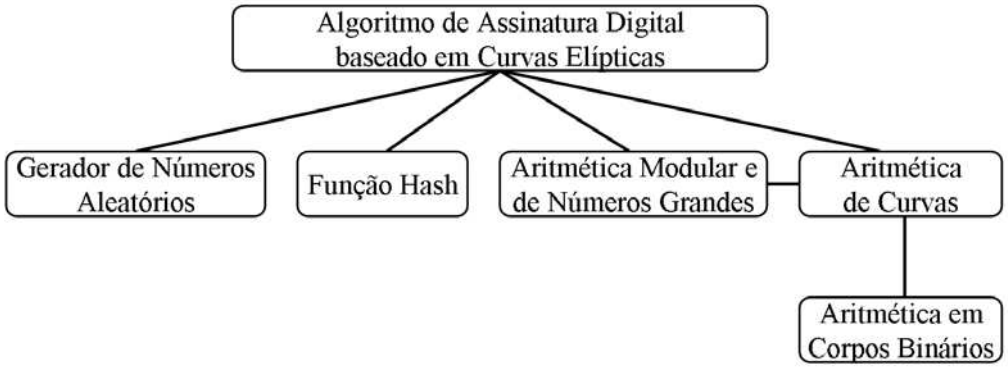


Figura 1.1: Elementos de Suporte ao ECDSA

Atualmente, o uso de processadores de 32 bits favorece implementações em *software*

das operações aritméticas necessárias em um sistema ECC utilizando-se bases polinomiais (PBs). Isso se deve ao bom mapeamento das operações em corpos finitos nas instruções de propósito geral de 32 bits desses processadores. Em geral, essas implementações resultam em desempenhos aceitáveis para os sistemas criptográficos da atualidade. Entretanto, devido à constante demanda por maior nível de segurança, o que pode ser traduzido em operandos maiores para essas operações aritméticas, faz-se necessário buscar por soluções que possuam maiores desempenhos que os da atualidade.

Alternativamente, bases normais (NBs) também podem ser utilizadas para implementar operações em corpos finitos. Porém, suas implementações em *software* possuem desvantagens, uma vez que o núcleo de várias operações aritméticas de corpo é baseado em permutações. Infelizmente, devido às essas permutações, as implementações em *software* dessas operações, não tem atingido desempenhos satisfatórios nas arquiteturas tradicionais de 32 bits dos processadores de propósito geral da atualidade.

1.1 Trabalhos Relacionados

Alguns trabalhos têm estudado o problema de se aproveitar a largura total do barramento dos processadores atuais quando implementando operações em corpos finitos. Em [14, 22] são considerados algoritmos rápidos para multiplicações em corpos finitos em bases normais sobre $\mathbb{F}_{2^m}$. Mais precisamente, são apresentados algoritmos que eliminam operações bit-a-bit em laços do algoritmo da multiplicação, resultando em um melhor aproveitamento das operações presentes no conjunto de instruções do processador. Já [9, 15] incluem algoritmos e técnicas para multiplicadores em corpos finitos em bases normais e bases normais ótimas (ONBs).

Novas abordagens para multiplicações em corpos finitos utilizando bases normais Gaussianas (GNBs) sobre $\mathbb{F}_{2^m}$ são mostradas em [47]. Os algoritmos apresentados nesse último trabalho resultam em métodos significantemente mais rápidos que todos os reportados anteriormente, empregando ao mesmo tempo menores áreas para armazenamento e processamento dos dados. Porém, afirma-se que multiplicações utilizando bases normais ainda são significantemente mais lentas que os melhores métodos que utilizam representações em bases polinomiais [27, 12]. Embora as operações de *trace*, quadrado, raiz quadrada, solução de equações quadráticas possuam as vantagens de serem rápidas e elegantes em bases normais, as penalidades impostas pelas suas implementações em *software* parecem ser suficientemente grandes para inviabilizar o seu uso em métodos de multiplicação de pontos. Entretanto, em [12], ressalta-se que essas conclusões são válidas apenas para implementações em *software*, ficando em aberto as conclusões para implementações em *hardware*.

Devido a essas razões, atualmente bases polinomiais são amplamente utilizadas para

implementações em *software* e em *hardware*. Por outro lado, existem alguns algoritmos de multiplicação em corpos finitos baseados em GNBs que fazem uso intenso de permutações. Dado que essas permutações podem ser construídas em *hardware* utilizando-se basicamente um conjunto de fios, espera-se que elas possam resultar em implementações bastante eficientes nessa tecnologia. Entretanto, diferentemente de implementações em *software*, pouca pesquisa tem sido feita em implementações de sistemas criptográficos em *hardware*, e dentre as existentes, pouquíssimas têm utilizado bases normais. Sendo assim, é difícil prever se a utilização de bases normais para implementações de sistemas criptográficos em *hardware* trazem vantagens reais sobre as bases polinomiais. Em outras palavras, até o momento, não se sabia se bases normais eram tão bem comportadas para *hardware* quanto às bases polinomiais o são para *software*.

Antes de citar alguns trabalhos relacionados à implementação de sistemas criptográficos utilizando-se bases normais, consideraremos implementações em *hardware* usando bases polinomiais. Em [16] um cartão PCI é apresentado para processar operações em curvas elípticas sobre $\mathbb{F}_{2^{191}}$. Esse trabalho afirma calcular uma multiplicação de ponto em torno de $1ms$ a uma freqüência de operação de 70MHz. Um aspecto interessante desse trabalho é a sua utilização em um sistema real, constituído por sistema operacional Windows e *device drivers*. Como veremos adiante, alguns trabalhos possuem restrições ao considerar somente o tempo de execução do módulo de *hardware*, ignorando por completo os atrasos presentes em uma aplicação real. Uma característica desse trabalho é de considerar otimizações para classes especiais de curvas elípticas, sendo assim, necessário reconfigurar a FPGA para que o sistema passe a operar em curvas genéricas.

O trabalho [28] propõe a extensão do conjunto de instruções de um processador MIPS32 de forma a acelerar operações aritméticas sobre $\mathbb{F}_{2^m}$ usando bases polinomiais. Resultados experimentais dessa abordagem mostram que a multiplicação de pontos sobre o corpo $\mathbb{F}_{2^{191}}$, com o processador operando a 33MHz, pode ser executada em $21ms$. Esse resultado, segundo os autores, é seis vezes mais rápido que a implementação do mesmo algoritmo em *software* em um processador MIPS32 convencional.

Similarmente, em [32], um processador SPARC V8 é especializado para execução de operações sobre corpos binários $\mathbb{F}_{2^m}$, através da inserção de instruções especializadas. Sua implementação otimizada para corpos $\mathbb{F}_{2^{191}}$, usando polinômios fixos para redução, acelerou a multiplicação de polinômios binários em até 90%, ao mesmo tempo em que reduziu o tamanho do código do programa.

Ainda utilizando bases polinomiais, um número de co-processadores criptográficos foram propostos em [20, 17, 18, 23, 30, 37, 38, 43]. Entretanto, ao invés de especializar os processadores, esses trabalhos consistem de implementações em *hardware* otimizadas para realizar multiplicações de ponto sobre corpos binários.

Em [18] é apresentado um acelerador configurável em *hardware* para multiplicação de

ponto sobre $\mathbb{F}_{2^m}$, onde $m \leq 255$. Esse acelerador pode operar tanto para curvas genéricas quanto para algumas curvas específicas. Considerando o caso de curvas genéricas sobre $\mathbb{F}_{2^{163}}$, os autores afirmam que kP pode ser calculado em 1,55ms, o que corresponde a um ganho de desempenho de duas vezes sobre sua implementação em *software*.

Outro trabalho baseando-se em implementações configuráveis é relatado em [17]. Essa implementação baseia-se no algoritmo de divisão-e-conquista de Karatsuba [27, 46], e é capaz de calcular uma multiplicação de ponto sobre $\mathbb{F}_{2^{113}}$ em 1,4ms. Além dessa, [20] descreve uma implementação que permite parametrizar (através da reconfiguração da FPGA) o polinômio de redução, a curva elíptica, e o tamanho do corpo finito. Nessa abordagem, o cálculo de kP leva 5,1ms sobre $\mathbb{F}_{2^{151}}$.

Ainda considerando bases polinomiais, tem-se em [30] um protótipo de co-processador operando a 66MHz capaz de calcular a multiplicação de ponto em 233μs para uma curva genérica, e em 75μs para uma curva de Koblitz, ambos sobre $\mathbb{F}_{2^{163}}$. Já em [23] uma arquitetura de *hardware* é proposta utilizando estratégias de paralelismo ao mesmo tempo em que procura reduzir a área de implementação. Nessa abordagem, os autores estimam que um cálculo de kP sobre $\mathbb{F}_{2^{191}}$ possa ser realizado em 80μs. Em [37] outro processador de curvas elípticas é proposto utilizando placas PCI. Esse processador utiliza-se de corpos primos de 192 bits, sendo capaz de realizar uma multiplicação de ponto em 99μs, em uma freqüência de 100MHz.

Recentemente, alguns trabalhos em bases polinomiais têm reduzido praticamente à metade o tempo de cálculo de kP quando comparamos com os trabalhos supracitados. Esses trabalhos utilizam o algoritmo de López-Dahab [10] para realizar a multiplicação de ponto.

Em [38] apresenta-se um módulo acelerador, operando sobre curvas recomendadas pelo NIST, e sobre os corpos finitos $\mathbb{F}_{2^{163}}$ e $\mathbb{F}_{2^{233}}$. Essa implementação faz uso de vários multiplicadores de corpos finitos operando em paralelo, aproveitando assim, as características de paralelismo do algoritmo de multiplicação de pontos utilizada. Nessa abordagem kP é realizada em 48μs sobre $\mathbb{F}_{2^{163}}$. A implementação mais rápida em *hardware* encontrada na literatura [43] realiza a multiplicação de ponto em 41,67μs sobre $\mathbb{F}_{2^{163}}$, utilizando uma freqüência de operação de 100MHz.

Diferentemente de bases polinomiais, encontram-se na literatura pouquíssimos trabalhos em bases normais. No melhor de nosso conhecimento, não temos notícias de trabalhos conclusivos se bases normais são, ou não são, apropriados para *hardware*. Essa falta de trabalhos em bases normais talvez possa ter sido influenciada negativamente pelo sucesso do uso de bases polinomiais em *software*, levando aos projetistas de *hardware* a também usar bases polinomiais para implementações em *hardware*. Temos assim, como um dos principais objetivos desse trabalho, a avaliação dos benefícios das bases normais Gaussianas para implementações em *hardware*.

Em [13, 19] processadores de curvas elípticas baseados em microcódigo são propostos para bases normais ótimas sobre $\mathbb{F}_{2^m}$. Esses processadores podem operar para m arbitrário, e neles o melhor tempo para cálculo de kP é de 3,7ms.

Já [31] apresenta um método, ainda utilizando bases normais ótimas, para produzir projetos de *hardware* para ECC sobre $\mathbb{F}_{2^m}$. Nesse caso, o melhor sistema multiplicador de ponto usando $\mathbb{F}_{2^{162}}$ realiza tal operação em 150μs. Nos três trabalhos acima mencionados, os módulos de *hardware* foram produzidos automaticamente por geradores, possibilitando o usuário, de forma rápida, determinar sistemas e experimentar relações de compromissos (*trade-offs*) como área, velocidade e segurança. Por outro lado, é fato comum em sistemas de *hardware*, que um projetista experiente possa explorar alguns detalhes de algoritmos e das tecnologias alvo, como também aproveitar características de um determinado tipo de base ou tamanho de corpo. Sendo assim, pode-se esperar melhores implementações em termos de desempenho e área dos módulos "escritos à mão"do que os gerados automaticamente por ferramentas, como nos trabalhos discutidos.

A única pesquisa realizada em bases normais Gaussianas que temos conhecimento é reportada em [40]. Nesse trabalho, um processador criptográfico para ECC sobre $\mathbb{F}_{2^{163}}$ é inteiramente implementado em *hardware*. Tal implementação baseia-se no algoritmo de cálculo da multiplicação de ponto proposta por López-Dahab, porém não explora as possibilidades de paralelismo desse algoritmo, e não otimiza os multiplicadores em corpos finitos de forma a obter maior velocidade. Como resultado, tem-se um desempenho baixo quando comparado com as melhores implementações em *hardware* utilizando bases polinomiais. Mais precisamente, esse processador calcula kP em 819μs.

1.2 Contribuição

A partir do contexto atual e dos resultados dos trabalhos relacionados, tem-se como primeiro objetivo desse trabalho a avaliação de implementações em *hardware* de operações aritméticas em corpos finitos e em curvas elípticas recomendados pelo NIST [11], utilizando para isso, bases normais Gaussianas sobre $\mathbb{F}_{2^{163}}$. Realizamos um estudo abrangente das operações baseando-nos em implementações otimizadas.

Muitos dos trabalhos mencionados anteriormente utilizam ferramentas de simulação para medir o desempenho *stand-alone* dos módulos. Em outras palavras, essas medidas não refletem o desempenho do módulo quando usado em uma aplicação real, utilizando-se, por exemplo, um programa escrito em uma linguagem de programação de alto nível como C. Assim, inúmeros atrasos (*overheads*) e detalhes do sistema são descartados, dificultando uma análise precisa dos ganhos de desempenho de tais implementações. Por outro lado, pode-se avaliar facilmente o desempenho de um sistema criptográfico implementado em *software*, dado que esses podem ser executados em sistemas reais, por exemplo em um PC,

onde todos os *overheads* são considerados. Por outro lado, nota-se que implementações em *hardware* necessitam de sistemas de medição de desempenho mais realistas, tornando, de certa maneira, mais justa as comparações com implementações em *software*.

Para a realização desse trabalho, utilizamos uma plataforma de prototipagem baseada em um processador *soft-core* NIOS2 [26], o qual nos permite experimentar implementações de operações em *hardware* e *software* utilizando *Field Programmable Gate Arrays* (FPGAs). No que diz respeito a implementações em *hardware* nesta arquitetura, podemos empregar duas abordagens distintas: instrução especializada (CI) e periférico do processador (CP). Na abordagem de instrução especializada o processador incorpora os módulos de *hardware* implementando uma determinada operação em seu *datapath*, formando assim novas instruções. Naturalmente, se fôssemos especializar um processador para ECC, mais de uma instrução poderia ser adicionada ao conjunto original de instruções do processador. Porém, de forma a se ter dados de áreas e desempenhos individualizados para cada uma das operações, apenas uma instrução é adicionada ao conjunto de instruções do NIOS2. Já na abordagem de periféricos, as operações são implementadas fora do *datapath* do processador, resultando em implementações mais flexíveis, que permitem, como por exemplo, usar diferentes freqüências de *clock* da utilizada pelo NIOS2. Similarmente às instruções especializadas, e devido às mesmas razões, cada processador possui apenas uma operação implementada como periférico.

De modo a tornar as análises mais abrangentes possíveis, consideramos nesse trabalho duas formas de medidas de rendimento: *stand-alone* e *end-to-end*. Na *stand-alone* considera-se apenas o tempo de processamento da operação pelo módulo de *hardware*, descartando-se os tempos necessários para escrita de argumentos e leitura de resultados. Com isso, podemos comparar nossos resultados com outras implementações que não consideram aplicações reais de seus módulos de *hardware*.

Como resultado, outro objetivo desse trabalho é comparar, de forma justa, implementações criptográficas em *software* e *hardware*. Para isso, tomamos como base uma plataforma real composta de processador, memória, temporizadores, etc., e adotamos o mesmo conjunto de parâmetros para nossos testes, entre eles, *clock* do processador, tamanhos de *caches* de dados e instruções, tipos e tamanhos de memória SRAMs, bem como um sistema de medição padronizado.

Já na *end-to-end* utilizamos a operação em uma aplicação real, onde não só as iterações para escrita/leitura de argumentos/resultados são consideradas, mas também outros fatores como transferência de dados entre memória e registradores do processador, e transferência entre os registradores do processador e os registradores internos do módulo da operação. Ou seja, nesse caso, temos uma medição mais precisa dos ganhos de desempenho resultantes do uso dos módulos de *hardware* em uma aplicação real.

De forma a salientar a diferença entre as medidas de desempenho de módulos *stand-*

alone e usando *end-to-end*, tomemos como exemplo o módulo multiplicador de ponto mais eficiente (3 NIST Modif. 27p). Esse módulo *stand-alone* é capaz de calcular kP em 36,30μs. Entretanto, quando implementamos esse módulo como um periférico do NIOS2, e utilizamos uma medição *end-to-end*, o cálculo de kP é realizado em 53μs. Comparando os dois resultados, é fácil observar uma diferença de 46% nos tempos das duas medições.

Dessa maneira, mostramos ser muito melhor nos basearmos em medições *end-to-end*, pois assim podemos ter maior precisão nos reais ganhos de desempenho resultantes de uma implementação em *hardware*. Trabalhos relacionados não incluem e nem estimam as perdas de desempenhos decorrentes do uso de seus módulos em uma aplicação real. Como resultado, pode-se esperar uma perda considerável de desempenho nos resultados obtidos através dos módulos *stand-alone*. Como resultado, podemos considerar bastante injusta a comparação de implementações em *software* (que considera todos esses *overheads*) com um implementações em *hardware* (desconsiderando todos esses *overheads*).

Contando com essa plataforma, abordagens *hardware/software* são amplamente exploradas em diferentes níveis: multiplicação e inversão em corpos finitos, e multiplicação de ponto. Além disso, comparamos os desempenhos das operações entre si dentro de uma mesma abordagem (instruções especializadas ou periféricos), bem como determinamos as vantagens e desvantagens de cada uma das duas abordagens para cada uma das operações. Através dessa metodologia, determinamos que as operações quadrado, raiz quadrada, *trace*, solução de equação quadrática, função β, e a multiplicação em corpos finitos resultam em maiores ganhos de desempenho quando implementadas como instruções especializadas. As operações mais complexas como a inversão e a multiplicação de pontos são mais apropriadas para serem implementadas como periféricos, e são candidatas a se tornarem co-processadores de sistemas de propósito geral.

Em todas nossas análises, comparamos os diversos algoritmos estudados em termos de área de implementação em *hardware*, utilizando FPGAs, seus ganhos de desempenho (*speedup*), e seus *speedups* por área implementada. Além disso, comparamos não só os *speedups* resultantes das implementações em *hardware* com suas respectivas implementações em *software* (o que resultaria em um *speedup* relativo), mas sim, comparamos tais implementações com os melhores algoritmos implementados em *software*, resultando em um *speedup* absoluto. Assim, determinamos quais algoritmos são mais apropriados para sistemas restritos em área, como por exemplo *Smart-Cards*, RFIDs, pequenos sensores, e quais são mais vantajosos para aplicações voltadas a desempenho como grandes servidores trabalhando com dados cifrados.

Utilizando o multiplicador López 18u, por exemplo, como instruções especializada em uma abordagem *hardware/software*, obtivemos que a multiplicação de pontos pode ser computada em 1,44ms, ou seja, aceleramos o cálculo de kP em 126,37 quando comparada com sua melhor implementação em *software* no NIOS2. Para ambientes restritos em área,

o multiplicador NIST Modif. 6p provê melhor custo/benefício entre desempenho e área de implementação, realizando o cálculo da multiplicação de pontos em 1,62ms.

Na abordagem de periféricos, mostramos que uma implementação totalmente em *hardware* da multiplicação de pontos, utilizando 3 multiplicadores finitos baseados no método modificado do NIST (3 NIST Modif. 27p), kP foi calculado em 53μs. Esse resultado é 2898,34 vezes mais rápido que a melhor implementação de kP em *software* no NIOS2. Observamos ainda que essa implementação é simultaneamente apropriada para sistemas voltados a desempenho e com restrições de área.

Ressaltamos também que até o momento não se tinha notícia de pesquisas afirmando que bases normais Gaussianas eram propícias para implementações em *hardware*. Através desse trabalho mostramos que se pode conseguir melhor desempenho utilizando bases normais Gaussianas do que os reportados utilizando-se bases polinomiais, dado que, de acordo com a literatura especializada, obtivemos a implementação mais rápida da multiplicação de pontos sobre $\mathbb{F}_{2^{163}}$.

Alguns resultados dos experimentos realizados nesse trabalho foram reportados nas seguintes publicações: IEEE 2005 International Conference on Field Programmable Technology (FPT'05) [35], e Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology [45].

1.3 Organização

O restante desse texto está organizado da seguinte maneira. No capítulo 2 apresentamos os fundamentos matemáticos necessários para a compreensão do trabalho em sua totalidade. Nesse capítulo encontram-se os conceitos básicos da aritmética em corpos finitos e em curvas elípticas, bem como uma breve introdução sobre bases normais Gaussianas. No capítulo 3 apresentamos os detalhes da plataforma de trabalho utilizada para a realização de nossos estudos e experimentos. Já no capítulo 4 apresentamos as implementações das operações aritméticas como instruções especializadas do processador NIOS2. É feita uma análise de desempenho dos módulos *stand-alone*, de suas utilizações dentro do *datapath* do NIOS2, bem como das diversas abordagens *hardware*/*software* baseadas em instruções especializadas. O capítulo 5 mostra a implementação das mesmas operações anteriores, só que agora utilizadas como periféricos do processador NIOS2. Mostramos também a análise de desempenho dos módulos *stand-alone*, como também seus desempenhos como periféricos. No final desse capítulo, comparamos as vantagens e desvantagens de se utilizar instruções especializadas e periféricos para implementar operações aritméticas em corpos finitos e em curvas elípticas. Depois de apresentar as mais diversas abordagens utilizadas neste estudo, apresentamos no capítulo 7 as nossas conclusões. Por fim, no capítulo 8, mostramos possíveis extensões deste trabalho que poderiam ser realizadas no futuro.

Capítulo 2

Fundamentos Matemáticos

De forma a prover o leitor de fundamentos matemáticos básicos que serão utilizados nas outras seções desse trabalho, apresentamos uma breve introdução a grupos e corpos, e a aritmética em corpos finitos e em curvas elípticas. Tratamentos mais rigorosos e aprofundados podem ser encontrados em [7], [24], [27], e [46].

2.1 Grupos e Corpos

Um grupo abeliano $(\mathbb{G}, *)$ consiste de um conjunto $\mathbb{G}$ com uma operação binária $*$ em $\mathbb{G}$ satisfazendo as seguintes propriedades:

- Associatividade: $a * (b * c) = (a * b) * c$ para todos $a, b, c \in \mathbb{G}$
- Identidade: Existe um elemento $e \in \mathbb{G}$, denominado elemento identidade, tal que $a * e = e * a = a$ para todo $a \in \mathbb{G}$
- Inverso: Para todo $a \in \mathbb{G}$, existe um elemento b, denominado inverso de a, tal que $a * b = b * a = e$
- Comutatividade: $a * b = b * a$ para todos $a, b \in \mathbb{G}$

A operação de grupo é usualmente a adição $(+)$ ou a multiplicação $(\cdot)$. No primeiro caso, o grupo é denominado de grupo *aditivo*, onde o elemento identidade é usualmente denotado por 0, e o inverso de a é denotado por $-a$. No segundo caso, o grupo é chamado de grupo *multiplicativo*, onde o elemento identidade é denotado por 1, e o inverso de a é denotado por a^{-1}. Se $\mathbb{G}$ é um conjunto finito, o grupo é denominado um grupo finito, cujo número de elementos em $\mathbb{G}$ é chamado de *ordem* de $\mathbb{G}$.

Um corpo $(\mathbb{F}, +, \cdot)$ é um conjunto de números $\mathbb{F}$ junto de duas operações, adição $(+)$ e multiplicação $(\cdot)$, que satisfazem as seguintes propriedades aritméticas:

- $(\mathbb{F}, +)$ é um grupo abeliano com identidade (aditiva) denotada por 0
- $(\mathbb{F} \setminus \{0\}, \cdot)$ é um grupo abeliano com identidade (multiplicativa) denotada por 1, de forma que $1 \cdot a = a \cdot 1 = a$ para todo $a \in \mathbb{F}$
- A lei distributiva é mantida, i.e., $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$ e $(b+c) \cdot a = (b \cdot a) + (c \cdot a)$ para todos $a, b, c \in \mathbb{F}$

Se o conjunto $\mathbb{F}$ 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 $\mathbb{F}$ 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 $\mathbb{F}$), e m é um inteiro positivo.

2.2 Aritmética em Corpos Finitos

Como explicado anteriormente, a eficiência da aritmética em corpos finitos é de grande importância para o desempenho de sistemas criptográficos em curvas elípticas. Três tipos de corpos são freqüentemente utilizados para implementar criptosistemas: corpos primos, corpos binários e corpos de extensão. Os elementos de um corpo são representados em termos de uma base, que por sua vez, pode ser polinomial ou normal. Nessa seção focaremos a aritmética de corpo finito usando bases normais Gaussianas sobre corpos binários, uma vez que nosso objetivo é utilizar esse tipo de base para implementar a aritmética finita sobre o corpo $\mathbb{F}_{2^m}$.

Uma base normal para $\mathbb{F}_{2^m}$ é definida como $\{\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}\}$, onde $\beta \in \mathbb{F}_{2^m}$. Assim, qualquer elemento $A \in \mathbb{F}_{2^m}$ é escrito como $A = \sum_{i=0}^{m-1} a_i \beta^{2^i}$, onde $a_i \in \{0, 1\}$, e pode ser representado como a seqüência binária $(a_0 a_1 a_2 ... a_{m-1})$. Seja $n = \lceil m/w \rceil$ e $s = wn - m$. Em *software*, A pode ser armazenado em um vetor de n palavras de w bits: $A = (a[0], a[1], ..., a[n-1])$. Os s bits mais a direita de $a[n-1]$ não são utilizados. Para corpos binários, o pequeno teorema de Fermat afirma que $A^{2^m} = A$, para todo $A \in \mathbb{F}_{2^m}$.

Uma generalização de bases normais ótimas para bases normais de baixa complexidade, conhecidas como bases normais Gaussianas, foi estudada por Ash, Blake, e Vanstone [5]. Para corpos binários, para os quais não existem bases normais ótimas, GNBs oferecem uma alternativa para implementar bases normais. O tipo T de uma base normal Gaussiana é um indicador da complexidade da multiplicação com respeito àquela base. Seja m um primo ímpar e T um inteiro positivo. Então uma GNB do tipo T para $\mathbb{F}_{2^m}$ existe se $p = Tm + 1$ é primo e $\text{mdc}(Tm/k, m) = 1$, onde k é a ordem multiplicativa de 2 módulo p. Bases normais ótimas são precisamente as GNBs de tipo $T = 1$ ou $T = 2$.

De forma a realizar uma ampla comparação entre abordagens *hardware* e *software*, implementamos as mais importantes operações em corpos finitos como módulos de *hardware*

que podem ser utilizados como instruções especializadas ou periféricos em um processador NIOS2. Entre elas, adição, quadrado, raiz quadrada, solução de equação quadrática, trace, três métodos de multiplicação em corpos finitos, e a multiplicação de ponto. As próximas seções descrevem essas operações em corpos finitos usando GNBs.

2.2.1 Adição

Dados dois elementos A e $B \in \mathbb{F}_{2^m}$, a soma $C = A + B$ é computada como o *xor* bit a bit de A com B, isto é, $c_i = a_i \oplus b_i$ para $i \in \{0, m-1\}$.

2.2.2 Quadrado e Raiz Quadrada

Dado o elemento $A \in \mathbb{F}_{2^m}$, o quadrado é calculado como $A^2 = \left(\sum_{i=0}^{m-1} a_i \beta^{2^i}\right)^2 = \sum_{i=0}^{m-1} a_i \beta^{2^{i+1}}$. Devido ao pequeno teorema de Fermat, $A^2 = a_{m-1} + \sum_{i=1}^{m-1} a_i \beta^{2^i}$, e assim,
$A^2 = (a_{m-1} a_0 a_1 a_2 ... a_{m-2})$. Dessa forma, o quadrado de um elemento finito é uma rotação a direita de sua representação vetorial. Similarmente, pode ser mostrado que a raiz quadrada de um elemento é uma rotação a esquerda de sua representação vetorial.

2.2.3 *Trace*

O trace de um elemento $A \in \mathbb{F}_{2^m}$ é definido como $Tr(A) = \sum_{i=0}^{m-1} A^{2^i} \in \{0, 1\}$. Para GNB, quando m é ímpar, o trace pode ser computado como $Tr(A) = \sum_{i=0}^{m-1} a_i$. Assim, o cálculo de $Tr(A)$ se reduz ao *xor* de todos os bits do vetor A.

2.2.4 Solução de Equação Quadrática

A solução da equação quadrática $X^2 + X = A$ pode ser determinada pelo Algoritmo 1. Quando m é ímpar, se X é uma solução, $X + 1$ também o é.

Algoritmo 1 Solução de $X^2 + X = A$

ENTRADA: $A = (a_0 a_1 ... a_{m-1}) \in \mathbb{F}_{2^m}$, m ímpar.
SAÍDA: $X = (x_0 x_1 ... x_{m-1}) \in \mathbb{F}_{2^m}$.

1. $x_0 \leftarrow a_0$.
2. Para i de 1 até $m - 2$ faça
   2.1. $x_i \leftarrow a_i \oplus x_{i-1}$.
3. $x_{m-1} \leftarrow 0$.
4. Retorna(X).

2.2.5 Multiplicação

A multiplicação em corpos finitos $\mathbb{F}_{2^m}$ é uma operação mais complexa do que as apresentadas até agora. Devido a sua importância para a multiplicação de ponto, ela deve ser calculada o mais eficientemente possível. Como citado anteriormente, vários algoritmos foram propostos para bases normais ótimas e bases normais Gaussianas. Esse trabalho considera três métodos para a multiplicação em corpos finitos usando GNBs: a proposta pelo NIST [11], a proposta por López [29], e uma versão modificada do algoritmo do NIST.

O algoritmo de multiplicação proposto pelo NIST é apresentado no Algoritmo 2. A função J é definida por $J(2^i u^j \bmod p) = i$, $i \in \{0, m-1\}$, $j \in \{0, T-1\}$, e $p = Tm + 1$.

Algoritmo 2 Multiplicação NIST

INPUT: $A, B \in \mathbb{F}_{2^m}, J(n), n \in \{1, p-1\}$
OUTPUT: $C = AB \in \mathbb{F}_{2^m}$
1. $C \leftarrow 0$.
2. Para i de 0 até $m-1$ faça
   2.1 Para n de 1 até $p-2$ faça
      2.1.1 $c_i \leftarrow c_i \oplus a_{J(n+1)} b_{J(p-n)}$.
   2.2 $A \leftarrow A \ll 1$, $B \leftarrow B \ll 1$.
3. Retorna (C).

Em [29], López introduziu um algoritmo para a multiplicação sobre $\mathbb{F}_{2^m}$ usando bases normais Gaussianas, o qual é baseado na observação que a operação $\beta \cdot B, B \in \mathbb{F}_{2^{163}}$, pode ser computada como a soma de T permutações de B, onde T é o tipo da GNB. Para $\mathbb{F}_{2^{163}}$ tem-se $T = 4$. Considere os elementos $\delta_i = \beta \cdot \beta^{2^i}$ para $i \in \{0, m-1\}$. Seja $n_i, i \in \{1, m-1\}$ o número de 1s na representação normal de δ_i, isto é, $\delta_i = \sum_{j=1}^{n_i} \beta^{2^{w_{ij}}}$. Então o produto $\beta \cdot B = \sum_{j=1}^{T} P_j(B)$, onde as permutações P_j são definidas como segue:

$P_1(B) = (b_1, b_{w_{11}}, b_{w_{21}}, ..., b_{w_{m-1,1}})$, e $P_j(B) = (0, b_{w_{1j}}, b_{w_{2j}}, ..., b_{w_{m-1,j}})$, $j > 1$.

Um método para calcular w_{ij}, em termos da função J é descrito no Algoritmo 3.

A descrição desse método para $\mathbb{F}_{2^m}$ é mostrado no Algoritmo 4. Note que a linha 2.2 mostra o cálculo de $S_P = \beta \cdot B$ (a soma de T permutações), denominada função β.

Um algoritmo alternativo no nível de bits para a multiplicação em $\mathbb{F}_{2^m}$ pode ser derivado dos Algoritmos 2 e 4. A descrição desse algoritmo, chamado multiplicação NIST modificada, é apresentada no Algoritmo 5. Seu núcleo de execução baseia-se na função β, na operação *and* ($\odot$), e um *trace*.

Algoritmo 3 Cálculo de w_{ij}

ENTRADA: $J(n), n = 1, 2, ..., p-1$, T par
SAÍDA: $w_{ij}, i \in \{1, m-1\}, j \in \{1, T\}$

1. Para i de 1 até $m-1$ faça
   1.1 Encontre as n_i soluções, $k_1, k_2, ..., k_{n_i}$, de
   $J(k+1) = i$ tal que $J(k_1) < J(k_2) < ... < J(k_{n_i})$.
   1.2 Para j de 1 até n_i faça $w_{ij} = J(k_j)$
   1.3 Se $n_i < T$ então $w_{ij} = m-1$ para $n_i < j \leq T$.
2. Retorne (w_{ij}).

Algoritmo 4 Multiplicação López

ENTRADA: $A, B \in \mathbb{F}_{2^m}, w_{ij}, i \in \{1, m-1\}, j \in \{1, T\}$
SAÍDA: $C = AB \in \mathbb{F}_{2^m}$

1. $C \leftarrow 0,\ B \leftarrow B \gg 1$.
2. Para i de $m-1$ até 0 faça
   2.1 $C \leftarrow C \gg 1$.
   2.2 $S_P \leftarrow \sum_{j=1}^{T} P_j(B)$.
   2.3 Se $a_i = 1$ então $C \leftarrow C \oplus S_P$.
   2.4 $B \leftarrow B \gg 1$.
3. Retorne (C).

Algoritmo 5 Multiplicação NIST Modificada

ENTRADA: $A, B \in \mathbb{F}_{2^m}, w_{ij}, i \in \{1, m-1\}, j \in \{1, T\}$
SAÍDA: $C = AB \in \mathbb{F}_{2^m}$

1. $C \leftarrow 0$.
2. Para i de 0 até $m-1$ faça
   2.1 $P_B \leftarrow \sum_{j=1}^{T} P_j(B)$.
   2.2 $c_i \leftarrow Trace(A \odot P_B)$.
   2.3 $A \leftarrow A \ll 1$, $B \leftarrow B \ll 1$.
3. Retorna (C).

2.2.6 Inversão

Um dos métodos mais eficientes para calcular o inverso de um elemento, em termos de somas, quadrados e multiplicações, é apresentado por Itoh and Tsujii [4]. Para $m = 163$ esse método, mostrado no Algoritmo 6, requer 9 multiplicações e 162 quadrados.

Algoritmo 6 Inversão (m ímpar)

ENTRADA: $A \neq 0 \in \mathbb{F}_{2^m}$
SAÍDA: $B = A^{-1}$
1. $A \leftarrow A^2, B \leftarrow 1, x \leftarrow (m-1)/2$.
2. Enquanto $x \neq 0$ faça
   2.1 $A \leftarrow A \cdot A^{2^x}$.
   2.2 Se x é par então $x \leftarrow x/2$.
   2.3 Ou então $B \leftarrow B \cdot A, A \leftarrow A^2, x \leftarrow (x-1)/2$.
3. Retorna (B).

2.3 Curvas Elípticas sobre $\mathbb{F}_{2^m}$

Apresentamos nessa seção uma breve introdução a curvas elípticas sobre $\mathbb{F}_{2^m}$; maiores detalhes podem ser encontrados em [6, 24, 27, 46, 11]. Uma curva elíptica não-supersingular E sobre $\mathbb{F}_{2^m}$ é definida como

$$y^2 + xy = x^3 + ax^2 + b \tag{2.1}$$

onde $a, b \in \mathbb{F}_{2^m}, b \neq 0$. Pode-se mostrar que o conjunto de pontos (x, y) que satisfaz 2.1, junto do ponto no infinito $\mathcal{O}$, formam um grupo finito abeliano sob uma apropriada operação de adição.

Em coordenadas afins, a adição de grupo elíptico é dada como segue [27]:

Seja $P = (x_1, y_1) \in E$; então $-P = (x_1, x_1+y_1)$. Para todo $P \in E, \mathcal{O}+P = P+\mathcal{O} = P$. Se $Q = (x_2, y_2) \in E$ e $Q \neq -P$, então $P + Q = (x_3, y_3)$, onde

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a, & P \neq Q \\ x_1^2 + \frac{b}{x_1^2}, & P = Q \end{cases} \tag{2.2}$$

e

$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1, & P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3, & P = Q. \end{cases} \tag{2.3}$$

As equações (2.2) e (2.3) requerem o computo de inversões em corpos finitos, as quais são custosas quando comparadas com a multiplicação em corpos finitos. Uma representação alternativa para reduzir o número de inversões é realizar a adição de pontos usando coordenadas projetivas [27]. O ponto afim $(X/Z, Y/Z)$ é representado como $(X : Y : Z), Z \neq 0$, e a equação projetiva da curva elíptica se torna:

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3. \tag{2.4}$$

O ponto no infinito $\mathcal{O}$ corresponde a $(0 : 1 : 0)$, enquanto o negativo de $(X : Y : Z)$ é $(X : X + Y : Z)$.

Porém, a adição de ponto pode ser calculada mais eficientemente usando coordenadas projetivas de López-Dahab [27, 10]. O ponto afim $(X/Z, Y/Z^2)$ é representado como $(X : Y : Z), Z \neq 0$, onde a equação projetiva da curva elíptica é

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4. \tag{2.5}$$

O ponto no infinito $\mathcal{O}$ corresponde a $(1 : 1 : 0)$, enquanto o negativo de $(X : Y : Z)$ é $(X : X+Y : Z)$. Em [27] as fórmulas aritméticas para a adição de ponto são apresentadas.

2.3.1 Multiplicação de Ponto (kP)

A operação central de um criptosistema de curvas elípticas é a multiplicação de ponto (kP), onde k é um inteiro e P um ponto em uma curva elíptica E definida sobre $\mathbb{F}$. A multiplicação de ponto é definida como

$$kP := \underbrace{P + P + \ldots + P}_{k \text{ vezes}}.$$

Um modo básico para calcular kP é o método binário, que usa a representação binária de k e requer na média t duplicações e $t/2$ adições, onde $t = \lfloor \log k \rfloor$. Um método mais eficiente para calcular kP é o algoritmo de López-Dahab [10], que é apresentado no Algoritmo 7. Esse método, que usa coordenadas projetivas, é baseado nas seguintes observações:

- A coordenada x da adição de dois pontos $Q_1 + Q_2$ pode ser calculada em termos das coordenadas x dos pontos Q_1, Q_2 e $Q_2 - Q_1$. Inicialmente, $Q_1 = P$ e $Q_2 = 2P$. Mais precisamente, sejam $Q_1 + Q_2 = (x_3, y_3)$, $Q_1 = (x_1, y_1)$, $Q_2 = (x_2, y_2)$, e $Q_1 - Q_2 = (x_4, y_4)$, com $Q_1 \neq \pm Q_2$, então x_3 pode ser calculado pela fórmula:

$$x_3 = x_4 + \frac{x_2}{x_1 + x_2} + \left(\frac{x_2}{x_1 + x_2}\right)^2. \tag{2.6}$$

- A coordenada x de $2Q_i = 2(x_i, y_i)$ é calculada em termos da coordenada x de Q_i pela fórmula:

$$x_i^2 + b/x_i^2. \tag{2.7}$$

- Em cada iteração i da Linha 3 do Algoritmo 7, calcula-se a coordenada x de $T_{i+1} = [2lP, (2l+1)P]$ ou $[(2l+1)P, (2l+2)P]$, dependendo do bit k_i, em termos de $T_i = [lP, (l+1)P]$, onde l é o inteiro representado pelos i bits mais à esquerda de k.

- Depois da última iteração, tendo calculado as coordenadas x de $kP = (x_1, y_1)$ e $(k+1)P = (x_2, y_2)$, a coordenada y de kP pode ser recuperada por:

$$y_1 = x^{-1}(x_1 + x)[(x_1 + x)(x_2 + x) + x^2 + y] + y. \tag{2.8}$$

Uma grande vantagem desse algoritmo é que ele não requer armazenamento extra. Além disso, em cada iteração, as mesmas operações são executadas (duplicação e adição de pontos), e como elas podem ser computadas em paralelo em *hardware*, aumenta-se a resistência a ataques baseados em análise de tempo e potência.

Algoritmo 7 Método de López-Dahab

ENTRADA: Um inteiro $k \geq 0, k = (1k_{t-2}...k_0)_2, P = (x, y) \in E$
SAÍDA: kP

1. $X_1 \leftarrow x, Z_1 \leftarrow 1$.
2. $X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$.
3. Para i de $t-2$ até 0 faça
   3.1 Se $k_i = 1$ então
   $T \leftarrow Z_1, Z_1 \leftarrow (X_1Z_2 + X_2Z_1)^2$.
   $X_1 \leftarrow xZ_1 + X_1X_2TZ_2, T \leftarrow X_2$.
   $X_2 \leftarrow X_2^4 + bZ_2^4, Z_2 \leftarrow T^2Z_2^2$.
   3.2 Ou então
   $T \leftarrow Z_2, Z_2 \leftarrow (X_1Z_2 + X_2Z_1)^2$.
   $X_2 \leftarrow xZ_2 + X_1X_2Z_1T, T \leftarrow X_1$.
   $X_1 \leftarrow X_1^4 + bZ_1^4, Z_1 \leftarrow T^2Z_1^2$.
4. $x_3 \leftarrow X_1/Z_1$.
5. $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2)+$.
   $(x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$.
6. Retorna (x_3, y_3).

Esse método pode ser representado de forma alternativa, como mostrado no Algoritmo 8. Para isso, toma-se como base as funções `Madd`, `Mdouble` e `Mxy`, mostradas nos Algoritmo 9, 10 e 11.

Algoritmo 8 Método de López-Dahab (Representação Alternativa)

ENTRADA: Um inteiro $k \geq 0, k = (1k_{t-2}...k_0)_2, P = (x, y) \in E$
SAÍDA: kP

1. Se $k = 0$ ou $x = 0$ então Retorna$(0, 0)$ e Termina.
2. $k \leftarrow (1k_{t-2}...k_1k_0)_2$.
3. $X_1 \leftarrow x, Z_1 \leftarrow 1, X_2 \leftarrow x^4 + b, Z_2 \leftarrow x^2$.
4. Para i de $t-2$ até 0 faça
   Se $k_i = 1$ então
   $Madd(X_1, Z_1, X_2, Z_2)$, $Mdouble(X_2, Z_2)$.
   Ou então
   $Madd(X_2, Z_2, X_1, Z_1)$, $Mdouble(X_1, Z_1)$.
5. Retorna $(Mxy(X_1, Z_1, X_2, Z_2))$.

A função `Madd` (Algoritmo 9) realiza a adição de dois pontos (P_1 e P_2) sobre a curva elíptica E, enquanto a função `Mdouble` (Algoritmo 10) se responsabiliza pela duplicação de um ponto P.

Algoritmo 9 *Madd*: Algoritmo de Adição de Pontos

ENTRADA: O corpo finito $\mathbb{F}_{2^m}$; os elementos finitos a e b definindo uma curva E sobre $\mathbb{F}_{2^m}$; as coordenadas x (X_1/Z_1 e X_2/Z_2) para os pontos $P_1 e P_2$.
SAÍDA: A coordenada x (X_1/Z_1) para o ponto $P_1 + P_2$.

1. $T_1 \leftarrow x$.
2. $X_1 \leftarrow X_1 \times Z_2$.
3. $Z_1 \leftarrow Z_1 \times X_2$.
4. $T_2 \leftarrow X_1 \times Z_1$.
5. $Z_1 \leftarrow Z_1 \times X_1$.
6. $Z_1 \leftarrow Z_1^2$.
7. $X_1 \leftarrow Z_1 \times T_1$.
8. $X_1 \leftarrow X_1 \times T_2$.
9. Retorna (X_1/Z_1).

Essas funções são chamadas múltiplas vezes dentro do laço mostrado na Linha 4 do Algoritmo 8. No final da computação, a função `Mxy` (Algoritmo 11) transforma o resultado de kP, que está representado utilizando-se coordenadas projetivas, para coordenadas afins. O Algoritmo 11 possui a vantagem de utilizar apenas uma inversão, ao invés de duas como mostrado na Linha 5 do Algoritmo 7. Mais precisamente, X_1/Z_1 e $(xZ_1Z_2)^{-1}$.

Algoritmo 10 *Mdouble*: Algoritmo de Duplicação de Pontos

ENTRADA: O corpo finito $\mathbb{F}_{2^m}$; os elementos finitos a e $c = b^{2^{m-1}} (c^2 = b)$ definindo uma curva E sobre $\mathbb{F}_{2^m}$; a coordenada x (X/Z) para um ponto P.
SAÍDA: A coordenada x (X/Z) para o ponto $2P$.

1. $T_1 \leftarrow c$.
2. $X \leftarrow X^2$.
3. $Z \leftarrow Z^2$.
4. $T_1 \leftarrow Z \times T_1$.
5. $Z \leftarrow Z \times X$.
6. $T_1 \leftarrow T_1^2$.
7. $X \leftarrow X^2$.
8. $X \leftarrow X + T_1$.
9. Retorna (X/Z).

Algoritmo 11 *Mxy*: Algoritmo de Transformação de Coordenadas Projetivas para Afins

ENTRADA: O corpo finito $\mathbb{F}_{2^m}$; as coordenadas afins do ponto $P = (x, y)$; as coordenadas x (X_1/Z_1 e X_2/Z_2) para os pontos P_1 e P_2.
SAÍDA: A coordenada afim $(x_k/y_k) = (X_2, Z_2)$ para o ponto P_1.

1. Se $Z_1 = 0$ então Retorna$(0, 0)$ e Termina.
2. Se $Z_2 = 0$ então Retorna$(x, x + y)$ e Termina.
3. $T_1 \leftarrow x$.
4. $T_2 \leftarrow y$.
5. $T_3 \leftarrow Z_1 \times Z_2$.
6. $Z_1 \leftarrow Z_1 \times T_1$.
7. $Z_1 \leftarrow Z_1 + X_1$.
8. $Z_2 \leftarrow Z_2 \times T_1$.
9. $X_1 \leftarrow Z_2 \times X_1$.
10. $Z_2 \leftarrow Z_2 + X_2$.
11. $Z_2 \leftarrow Z_2 \times Z_1$.
12. $T_4 \leftarrow T_1^2$.
13. $T_4 \leftarrow T_4 + T_2$.
14. $T_4 \leftarrow T_4 \times T_3$.
15. $T_4 \leftarrow T_4 + Z_2$.
16. $T_3 \leftarrow T_3 \times T_1$.
17. $T_3 \leftarrow inverso(T_3)$.
18. $T_4 \leftarrow T_3 \times T_4$.
19. $X_2 \leftarrow X_1 \times T_3$.
20. $Z_2 \leftarrow X_2 + T_1$.
21. $Z_2 \leftarrow Z_2 \times T_4$.
22. $Z_2 \leftarrow Z_2 + T_2$.
23. Retorna (X_2, Z_2).

Capítulo 3

Plataforma de Trabalho

3.1 O Processador NIOS2

Quando *system-on-chips* (SOCs) são projetados, é importante considerar alguns fatores como desempenho, custo, e a facilidade de integração com outros núcleos de *hardware* (*cores*). Além disso, pode-se considerar a possibilidade de se especializar a arquitetura para uma determinada classe de aplicações, como também a disponibilidade de um bom compilador para gerar código para o processador especializado. Em nosso caso, necessitávamos de uma plataforma reconfigurável baseada em FPGAs que nos permitisse investigar o uso de instruções especializadas e periféricos dedicados a aplicações criptográficas baseadas em curvas elípticas.

Levando em consideração essas restrições, e devido à sua flexibilidade, selecionamos o processador NIOS2 [26] como o elemento central de nossa plataforma. O NIOS2 é um processador *soft-core* de 32 bits que permite a especialização de sua arquitetura de *hardware* para aumentar o seu desempenho. Mais precisamente, a plataforma de desenvolvimento baseada no NIOS2 permite a criação e integração de módulos de *hardware* ao processador, seja como uma instrução especializada, seja como um periférico.

Essa plataforma conta também com um porte do compilador GCC para o NIOS2 capaz de gerar código para as instruções especializadas recém criadas, bem como, utilizar de forma simples e elegante os periféricos desse processador. A plataforma de trabalho utilizada baseia-se no kit de desenvolvimento NIOS2 da Altera [36, 42], que conta com memória, *caches* de dados e instruções configuráveis, e uma FPGA de alta capacidade Stratix II EPS260F672C5ES [41], permitindo uma alta reconfigurabilidade do sistema.

A idéia por trás da implementação de módulos de *hardware* como instruções especializadas e periféricos é a de se especializar o sistema para algoritmos criptográficos. Como será mostrado nos capítulos 4 e 5, tal especialização resulta em altos *speedups* sem grandes modificações nos programas originais. Mais precisamente, basta utilizar uma biblioteca C,

que implementa uma macro para a chamada da instrução especializada/periférico. Outro aspecto importante a ser considerado é o potencial reuso do processador especializado, ou do periférico (que pode ser pensado nesse caso, como um co-processador) em outro projeto de um SOC. Dessa forma, favorece-se o *time-to-market* de projetos posteriores envolvendo sistemas criptográficos para curvas elípticas.

Nessa plataforma, o programa inicia sua execução em *software*, ou seja, utilizando as instruções de propósito geral do processador. Se o programa necessita executar uma operação em corpos finitos de maneira eficiente, ele realiza uma seqüência de chamadas às instruções especializadas ou periféricos, que desviam a execução do algoritmo para *hardware*. Depois disso, a execução do programa finaliza em *software*.

3.2 Instruções Especializadas

Utilizando-se da natureza *soft-core* do processador NIOS2, projetistas podem integrar lógicas extras à unidade lógica aritmética (ULA) do processador, como mostrado na Figura 3.1, para formar instruções especializadas.

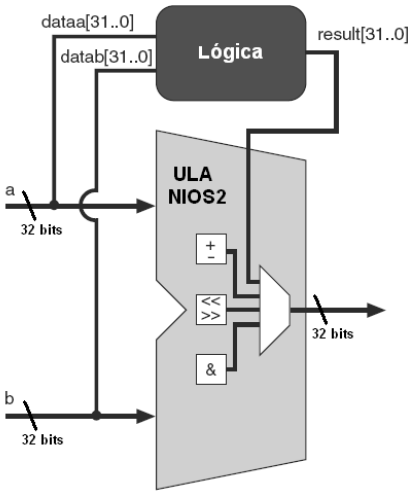


Figura 3.1: Integração de Lógica Extra à ULA do NIOS2

A Altera recomenda a utilização de instruções especializadas para operações que demandam poucos ciclos de *clock* e que são relativamente pequenas em área. Isso porque quando instruções especializadas são chamadas, o processador fica em um estado de espera, impedindo a realização de outras tarefas. Além disso, lógicas muito complexas demandarão maior área de implementação na FPGA, o que, por questões de roteamento, levaria em uma redução de sua freqüência de operação. Como resultado todo o sistema operaria de maneira mais lenta, dado que a instrução se situa dentro do *datapath* do processador.

Para criar uma instrução especializada do NIOS2, o projetista deve seguir uma interface pré-determinada pelas ferramentas da Altera. Para isso, existem quatro classes arquiteturais, como mostrado na Figura 3.2, que podem ser utilizadas simultaneamente para a criação de uma instrução especializada. São elas: Combinacional, Multi-Ciclo, Estendido, e Acesso a Registrador Interno.

A classe combinacional é empregada normalmente para instruções que executam em um único ciclo de *clock*, e a multi-ciclo, como o próprio nome diz, para operações mais demoradas que não podem ser realizadas em um único ciclo de *clock*. Já a classe estendido permite enviar um valor de 8 bits para a instrução especializada, possivelmente alterando seu comportamento de acordo com o comando recebido. Por fim, a classe de acesso a registrador interno pode ser utilizada para a leitura/escrita de unidades de armazenamento de dados internos ao módulo de *hardware*.

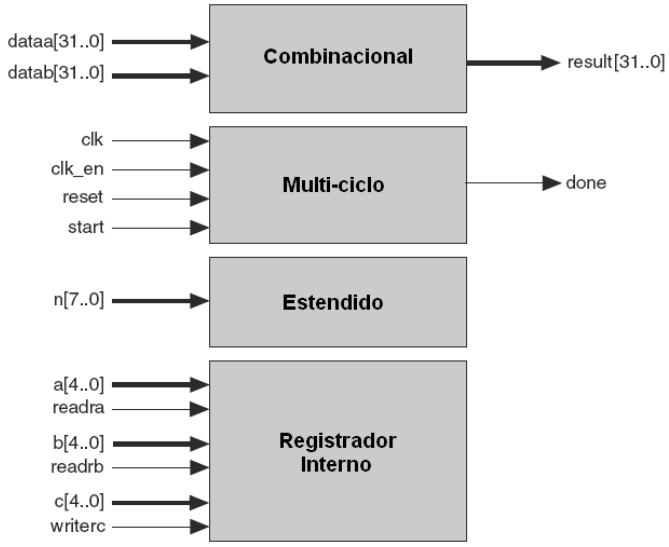


Figura 3.2: Classes Arquiteturais das Instruções Especializadas

Em nossas implementações utilizamos uma união das classes Combinacional, Multi-Ciclo e Estendido, dados que algumas operações são realizadas em um único ciclo, e outras não. Entretanto, todas usam a classe estendida para receber comandos que alteram sua lógica interna de operação. Uma interface em VHDL de instrução especializada, bastante utilizada em nosso trabalho e constituída pelas classes arquiteturais supracitadas, é mostrada na Figura 3.3. Os sinais de `clk`, `clk_en`, `reset`, e `start` habilitam o início e sincronizam a execução da instrução com o clock do processador. A entrada de dados é realizada por dois operandos de 32 bits `dataa` e `datab`, e quando necessário, um comando é enviado pela entrada `n`. Qualquer que seja o modo de operação da instrução especializada, informa-se o processador que ela terminou pelo sinal `done`, e retornam-se os 32 bits do resultado da operação através de `result`.

```
entity <CI_NAME> is
  port(
    clk    : in  std_logic;
    clk_en : in  std_logic;
    reset  : in  std_logic;
    start  : in  std_logic;
    dataa  : in  std_logic_vector(31 downto 0);
    datab  : in  std_logic_vector(31 downto 0);
    n      : in  std_logic_vector(7 downto 0);
    result : out std_logic_vector(31 downto 0);
    done   : out std_logic);
end <CI_NAME>;
```

Figura 3.3: Interface de Instrução Especializada em VHDL

Toda inserção de lógica na ULA do NIOS2 é realizada pela ferramenta SOPC Builder da Altera. Essa ferramenta automaticamente gera um novo processador com a instrução especializada em seu *datapath*, e simultaneamente, cria uma biblioteca (`system.h`) que contém as macros de chamadas à nova instrução do processador. Dessa maneira, cabe ao projetista apenas incluir essa biblioteca em seu código fonte e utilizar essa macro para fazer uso da instrução especializada. A Figura 3.4 exemplifica como é feita a chamada a uma instrução especializada a partir de uma linguagem de programação de alto nível como C. Tomando como exemplo a primeira chamada, tem-se duas palavras (`a[0]` e `b[0]`) sendo enviadas para o módulo de *hardware*, junto de um identificador (no caso `0`). Como retorno da função, tem-se outra palavra, que é armazenada em `c[0`. Esse esquema se repete até a obtenção de `c[5]`.

```
#include "system.h"
void main() {
  u32 a[6], b[6], c[6];
  ...

  // Chamadas à instrução especializada
  // result = <CI_NAME>(n, dataa, datab);
  c[0] = <CI_NAME>(0, a[0], b[0]);
  ...
  c[5] = <CI_NAME>(5, a[5], b[5]);
}
```

Figura 3.4: Chamada à uma Instrução Especializada usando Linguagem C

3.3 Periféricos

Similarmente às instruções especializadas, projetistas podem integrar seus módulos de *hardware* a sistemas baseados no processador NIOS2. Entretanto, como o módulo fica externo ao *datapath* do processador, temos maior grau de liberdade em sua implementação. Dessa forma, por exemplo, podem-se ter módulos muito maiores sem prejudicar o desempenho do sistema, ou mesmo ter o módulo operando em freqüências diferentes das utilizadas pelo NIOS2. No caso das instruções especializadas, isso era bastante difícil, uma vez que o módulo de *hardware* tinha necessariamente de operar na mesma freqüência do processador. Outra vantagem é poder adotar chamadas não-bloqueantes aos módulos, permitindo o processador executar outras tarefas em paralelo ao processamento dos dados no periférico.

Nessa abordagem, os periféricos podem ser qualquer lógica que adote uma interface de comunicação com o barramento Avalon [21, 33]. O Avalon consiste de sinais de controle, e barramentos de dados e endereços, sendo que os periféricos utilizam esse barramento para se comunicar com todos os outros componentes do sistema, como por exemplo, com o processador NIOS2. Essa situação é mostrada na Figura 3.5.

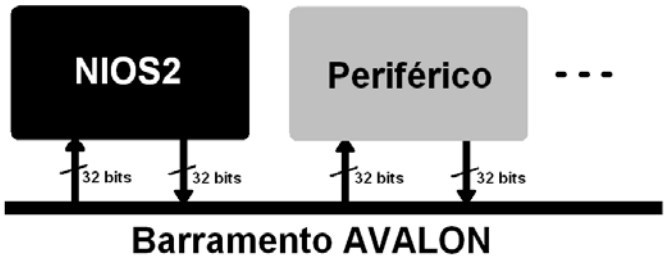


Figura 3.5: Integração do NIOS2 e Periféricos através do Barramento Avalon

Periféricos Avalon podem ser classificados como mestre (*master*) ou escravo (*slave*). Um periférico mestre é um periférico que pode iniciar transferências no barramento Avalon. Um periférico mestre possui pelo menos uma porta mestre que se conecta com o barramento Avalon. Porém, um periférico mestre pode também possuir portas escravas, as quais permitem o periférico receber transferências iniciadas por outros mestres presentes no barramento. Um periférico escravo é um periférico que apenas aceita transferências do barramento Avalon, e não pode inicia-las. Normalmente, periféricos escravos possuem apenas uma porta escrava, a qual se conecta com o barramento Avalon.

Nossos módulos foram implementados como periféricos escravos, sendo que não necessitamos que o periférico inicie transferências pelo barramento. Uma interface em VHDL desses periféricos escravos, amplamente utilizada em nosso trabalho, é mostrada na Figura 3.6.

```
entity <CP_NAME> is
  port(
    clk         : in  std_logic;
    reset       : in  std_logic;
    chipselect  : in  std_logic;
    address     : in  std_logic_vector(4 downto 0);
    write       : in  std_logic;
    writedata   : in  std_logic_vector(31 downto 0);
    read        : in  std_logic;
    readdata    : out std_logic_vector(31 downto 0);
    waitrequest : out std_logic);
end <CP_NAME>;
```

Figura 3.6: Interface de um Periférico Escravo em VHDL

Os sinais `chipselect` e `reset` habilitam o início de execução da operação, utilizando o sinal `clk` como *clock* do periférico. A entrada de dados é realizada por apenas um operando de 32 bits `writedata` quando habilitado por `write`. Já a saída de dados é composta por outro canal de 32 bits `readdata`, e habilitado pelo sinal `read`. Quando necessário endereçar registradores internos, ou enviar algum comando ao periférico utiliza-se o sinal `address`, também de 32 bits. Quando a operação realizada pelo periférico toma mais de um ciclo para ser finalizada, o sinal `waitrequest` é assertado, e permanece nesse estado até o final da operação. Nessa situação, o processador fica em um estado de espera, impossibilitado de executar outras operações. Dessa maneira, se o projetista deseja executar operações em paralelo no NIOS2, faz-se necessário adotar alguma estratégia de bufferização de argumentos e resultados, liberando o processador da espera.

Em nossas implementações adotamos chamadas bloqueantes, dado que não necessitamos dos serviços do processador em paralelo ao processamento do periférico. Entretanto, pretendemos melhorar os módulos no futuro de forma a possibilitar o processador escrever/ler dados no/do periférico ao mesmo tempo em que ele realiza o processamento de uma operação. Dessa maneira, um *pipeline* de escrita-execução-leitura seria formado, resultando em maior desempenho do sistema em chamadas sucessivas da mesma operação.

Toda inserção de periféricos do NIOS2 é realizada pela ferramenta SOPC Builder da Altera. Essa ferramenta automaticamente gera um novo sistema utilizando o módulo de *hardware* como periférico do processador NIOS2. A partir da descrição do módulo em alguma linguagem de descrição de *hardware*, como VHDL ou Verilog, a ferramenta SOPC Builder automaticamente gera a lógica de interconexão, incluindo decodificação de endereços, multiplexação do *datapath*, geração de *wait-states*, controle de interrupções, e correção de largura de dados dos barramentos.

Nesse trabalho não utilizamos interrupções. Isso para poder analisar o tempo gasto somente em uma operação simples. O uso de interrupções em um sistema real seria

interessante, pois o processador poderia se dedicar a outras tarefas enquanto a operação é executada. Como, nesse trabalho, o processador está dedicado somente a execução de uma operação específica, acreditamos que uma abordagem simplificada seria suficiente para a análise de desempenho. Da mesma maneira, pode-se pensar em acelerar o sistema com o uso de DMA. Entretanto isso demandaria um pouco mais de lógica no módulo de *hardware*, pois esse teria de se tornar um mestre do barramento Avalon, implicando em maior área de implementação. Como conseqüência, poderíamos ter uma perda de desempenho do módulo. Novamente, como estamos preocupados apenas com o núcleo de execução de cada operação, uma abordagem simplificada é suficiente para nossas análises.

Para interfacear com o módulo basta incluir uma biblioteca (`io.h`) que contém as macros de chamadas ao periférico. A Figura 3.7 exemplifica como é feita a chamada a um periférico utilizando a linguagem C. Diferentemente de como é feito nas chamadas de instruções especializadas, uma chamada de escrita a um periférico envia apenas uma palavra e um identificador. No caso da primeira chamada de escrita (`WR`) da Figura 3.7, tem-se o envio da palavra`a[0]` e do identificador `0` para o módulo de *hardware*. Somente depois das chamadas de escrita é que se pode obter algum resultado vindo do periférico através de chamadas de leitura (`RD`).

```
#include <io.h>
void main() {
  u32 a[6], b[6], c[6];
  ...

  // Chamadas de escrita no periférico
  // IOWR(CP_BASE_ADDRESS, offset, writedata);
  IOWR(<CP_NAME>_BASE, 0, a[0]);
  ...
  IOWR(<CP_NAME>_BASE, 5, a[5]);
  IOWR(<CP_NAME>_BASE, 6, b[0]);
  ...
  IOWR(<CP_NAME>_BASE, 11, b[5]);

  // Chamadas de leitura do periférico
  // readdata = (CP_BASE_ADDRESS, offset);
  c[0] = IORD(<CP_NAME>_BASE, 0);
  ...
  c[5] = IORD(<CP_NAME>_BASE, 5);
}
```

Figura 3.7: Chamada a um Periférico usando Linguagem C

3.4 Padronização do Sistema

Como nossas comparações *hardware-hardware* e *hardware-software* têm como intuito ser as mais justas possíveis, precisamos de uma plataforma única em todos os aspectos. Do contrário, pequenas alterações nas condições de teste poderiam comprometer a precisão dos resultados, e assim nossas análises. Dessa maneira, utilizamos nesse trabalho o mesmo sistema NIOS2, isto é, um processador com *pipeline* de seis estágios rodando a 120MHz, contando com predição dinâmica de *branch* e com um multiplicador de inteiros em *hardware*.

De forma a reduzir *cache misses*, o que interferiria nos resultados de forma a favorecer a implementação em *hardware*, utilizamos *caches* relativamente grandes. Sendo assim, contamos com *caches* de instruções e de dados com, respectivamente, 64KB e 32KB. Além disso, utilizamos uma memória SRAM de 64KB interna a FPGA para armazenamento dos dados do programa, e uma memória SRAM de 1MB externa a FPGA para armazenamento do código executável do programa. Como conseqüência, o código do programa teste pode ter tamanho relativamente grande e ainda ser acondicionado na *cache* de instruções reduzindo acessos à memória externa à FPGA, enquanto os dados necessários ao teste dos módulos são buscados rapidamente da memória interna.

Esse processador base recebeu instruções especializadas e periféricos, e dependendo do módulo de *hardware* tem-se uma grande área de implementação e um roteamento bastante complexo. Como conseqüência, espera-se que a freqüência de operação final de todo o sistema seja menor que as atingidas pelo processador quando sintetizado sozinho. De forma a evitar reduzir a freqüência de operação do processador por razões da inserção da instrução especializada/periférico, tivemos de analisar diversas configurações da ferramenta QuartusII [34] de forma a gerar o processador mais adequado para nossos experimentos. Mais precisamente, experimentamos com as técnicas de otimização de análise e síntese, e o esforço de síntese física daquela ferramenta. Vale a pena salientar que utilizamos a versão 5.0 do QuartusII para todas as implementações.

Pode-se observar pela Tabela 3.1 que o processador com maior freqüência de operação (131,10 MHz) foi obtido utilizando-se técnicas de otimização de análise e síntese voltadas para área, exigindo esforço extra da ferramenta de síntese. Com essa configuração, obteve-se um processador que ocupa uma área considerada satisfatória (3580 ALUTs) quando comparada com os resultados das outras configurações do QuartusII.

De forma a padronizar o sistema de medição, utilizamos todas as operações como funções, estejam elas implementadas em *hardware* ou em *software*, como mostrado nas Figuras 3.8a e 3.8b, respectivamente. Essas funções são então chamadas múltiplas vezes dentro de um laço. No interior de cada função, temos as macros que de fato realizam as chamadas às instruções especializadas e periféricos.

Téc. de Otimização Análise e Síntese	Esforço de Síntese Física	Área (ALUTs)	Freqüência (MHz)
Velocidade	Normal	4243	123,78
	Extra	4406	123,06
Balanceado	Normal	3347	124,36
	Extra	3723	126,26
Área	Normal	3358	126,87
	Extra	**3580**	**131,10**

Tabela 3.1: Determinação dos Parâmetros de Síntese do Processador NIOS2

Na entrada da execução de cada laço lê-se um contador do sistema que é incrementado na mesma freqüência de *clock* do processador. Nesse momento, inicia-se a contagem de tempo da execução da operação. No final da execução, toma-se novamente o valor desse contador. Dessa maneira, sabendo-se a diferença entre os valores lidos, e de posse da freqüência de operação do processador, determina-se o tempo de execução da operação.

```
// SOFTWARE EXECUTION
t1=alt_timestamp();
for(k = 0; k < NumTest; k++)
  for (j = 0; j < LOOPS; j++)
    SW_function(cs[j],a[j],b[j]);
t2=alt_timestamp();
```

(a)

```
// HARDWARE EXECUTION
t1=alt_timestamp();
for(k = 0; k < NumTest; k++)
  for (j = 0; j < LOOPS; j++)
    HW_function(ch[j],a[j],b[j]);
t2=alt_timestamp();
```

(b)

Figura 3.8: Padronização do Sistema de Medição

Além dos atrasos inseridos pelo laço (*for*), o compilador tem de traduzir cada chamada macro para uma seqüência de instruções *load* e *store*. Tais instruções possuem, respectivamente, as funções de carregar os registradores com os argumentos da chamada, e armazenar na memória o resultados da mesma. A Figura 3.9a ilustra a chamada a uma instrução especializada a partir de macros, a qual é transformada pelo compilador no código Assembly mostrado na Figura 3.9b.

Como resultado, mesmo se a operação utiliza apenas um ciclo de *clock* em *hardware*, não somos capazes de executá-la em um único ciclo de *clock* utilizando macros. Uma possível solução para esse problema seria otimizar o programa em nível Assembly, pagando porém, o preço da portabilidade do código fonte. Entretanto, ainda temos uma comparação bastante justa entre *hardware* e *software*, dado que visamos implementações reais e que tais atrasos ocorrem em ambas as implementações.

```
...                                 ...
                                    a8: 28c00017  ldw    r3,0(r5)
                                    ac: 30800017  ldw    r2,0(r6)
c[0]=<CI_NAME>(a[0],b[0]);          b0: 1887c032  custom 0,r3,r3,r2
                                    b4: 20c00015  stw    r3,0(r4)
                                    b8: 28800117  ldw    r2,4(r5)
                                    bc: 30c00117  ldw    r3,4(r6)
c[1]=<CI_NAME>(a[1],b[1]);          c0: 10c5c032  custom 0,r2,r2,r3
                                    c4: 20800115  stw    r2,4(r4)
...                                 ...
```

(a) (b)

Figura 3.9: Macro de Acesso ao Módulo de *Hardware* e Código Assembly Gerado pelo Compilador

3.5 Estratégias de Chamadas aos Módulos de *Hardware*

Quando operando sobre o corpo finito $\mathbb{F}_{2^{163}}$, as operações aritméticas normalmente utilizam um ou dois argumentos de 163 bits e retornam um resultado de 163 bits. Entretanto, arquiteturas de propósito geral trabalham usualmente com *datapaths* de 16, 32 e 64 bits, demandando, por conseqüência, implementações em *software* bastante otimizadas para lidar com o gargalo imposto pela largura do barramento. Se o tamanho da palavra de uma determinada arquitetura é w bits, um elemento de m bits utiliza $\lceil m/w \rceil$ palavras. Dessa maneira, quando empregando $m = 163$ e o processador NIOS2 ($w = 32$), um elemento $A \in \mathbb{F}_{2^{163}}$ necessitará de seis palavras de 32 bits para ser representado.

Na abordagem de instruções especializadas, a interface do módulo de *hardware* possui duas entradas de argumentos de 32 bits e uma saída de resultado de 32 bits. Já os periféricos possuem apenas uma entrada de argumento de 32 bits, e uma saída de resultado de 32 bits. Diante dessas restrições de projeto e almejando menores registradores internos ao módulo, como também maior eficiência, torna-se interessante adotar algumas estratégias de chamadas aos módulos.

Independente da abordagem utilizada (instrução especializada ou periférico), pode-se separar as operações em dois grupos, denominados *divisíveis* e *não-divisíveis*. As *divisíveis* podem ser partidas e executadas em blocos de 32 bits. Já as *não-divisíveis* requerem todos os 163 bits dos argumentos antes de iniciar os cálculos do resultado. Cabe salientar que operações divisíveis em na abordagem de instruções especializadas, o serão na de periféricos. O mesmo ocorre com as operações não-divisíveis.

3.5.1 Operações Divisíveis

As operações adição, quadrado, raiz quadrada, *trace* e resolução de equação quadrática recaem na categoria de divisíveis.

Considerando a abordagem de instrução especializada, e tomando os operandos `A` e `B` de 163 bits, cada chamada toma dois blocos de 32 bits dos argumentos e retorna um bloco de 32 bits do resultado. Assim, para se obter todo o resultado `C`, são necessárias seis chamadas às instruções especializadas, como ilustrado na Figura A.1 do Apêndice A. Como conseqüências dessa partição, em alguns casos, são necessários registradores internos para armazenar resultados intermediários, os quais serão utilizados na próxima execução de 32 bits.

Já na abordagem de periféricos, que possui apenas uma entrada de 32 bits, são necessárias ao todo dezoito chamadas ao módulo, como mostrado na Figura A.2. Na primeira chamada um bloco de 32 bits de `A` é enviado ao módulo de *hardware*, sendo então armazenado em um registrador interno. A segunda chamada, carregando um bloco de 32 bits de `B`, dispara a execução da operação. Em seguida, realiza uma terceira chamada para ler o bloco de 32 do resultado. Essa seqüência se repete até se obter todo o resultado `C`. Quando comparado com as instruções especializadas, é fácil notar que os periféricos demandam um número maior de chamadas, como também maiores registradores internos ao módulo. Esse é um forte indício que instruções especializadas são mais apropriadas para operações divisíveis.

3.5.2 Operações Não-Divisíveis

Diferentemente da categoria anterior, algumas operações requerem que todos os argumentos estejam disponíveis antes do início do processamento, sendo esse o caso da multiplicação e inversão em corpos finitos, algumas funções de permutações, e a multiplicação de ponto. Para contornar esse problema, utilizam-se três registradores internos de 163 bits, mais precisamente, dois para armazenamento dos argumentos e um para o resultado, nomeados respectivamente de `IRA`, `IRB`, e `IRC`. No caso da multiplicação de ponto são utilizados mais registradores, porém exemplifica-se a seguir, sem perda de generalidade, a estratégia adotada para o caso das operações em corpos finitos.

Na abordagem de instruções especializadas, são necessárias seis iterações para se escrever os argumentos em `IRA` e `IRB`, como mostrado na Figura A.3. Na última escrita, a instrução inicia imediatamente sua execução, que pode tomar múltiplos ciclos de *clock*. No momento de retorno da execução, a instrução especializada já retorna 32 bits do resultado. Assim, o restante do resultado deve ser lido do registrador interno `IRC`, sendo necessário para isso, mais cinco chamadas à instrução, as quais são representadas na Figura A.4.

No caso de periféricos, devido à única entrada de dados de 32 bits, são necessárias

seis chamadas para se escrever `IRA`, e mais seis para se escrever `IRB`, como mostrado na Figura A.5. A última iteração de escrita dispara a execução da operação, que pode tomar múltiplos ciclos de *clock*. Para se ler o resultado de `IRC` são necessárias mais seis iterações, como ilustrado na Figura A.6.

Capítulo 4

Instruções Especializadas

4.1 Implementação dos Módulos Aritméticos

Essa seção apresenta os detalhes de implementação das operações no corpo finito $\mathbb{F}_{2^{163}}$ usando a abordagem de instruções especializadas. De modo a melhorar o desempenho das operações, um conjunto de otimizações foram adotadas como desenrolamento de laços e paralelismo. Quando possível, direcionou-se as implementações tomando como base as características das operações divisíveis, permitindo assim, o uso mais eficiente do módulo quando implementado como instrução especializada.

4.1.1 Adição

Dado que a adição em corpos finitos é um *xor* bit a bit dos argumentos, essa operação pode ser dividida em blocos de 32 bits, como mostrado na Figura 4.1. Isso significa que múltiplas chamadas ao módulo são necessárias para realizar uma única operação, como mostrado na Figura 4.2. Como cada iteração retorna 32 bits do resultado, e os argumentos possuem 32-bits, seis chamadas a instrução especializada são necessárias. Essa operação não requer nenhum tipo de armazenamento interno ao módulo.

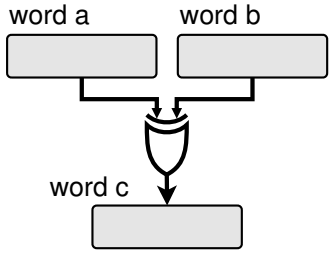


Figura 4.1: Diagrama em Blocos da Operação Adição

```
c[0] = ADDITION( a[0], b[0] );
c[1] = ADDITION( a[1], b[1] );
c[2] = ADDITION( a[2], b[2] );
c[3] = ADDITION( a[3], b[3] );
c[4] = ADDITION( a[4], b[4] );
c[5] = ADDITION( a[5], b[5] );
```

Figura 4.2: Chamadas à Instrução Especializada de Adição

4.1.2 Quadrado

Similarmente à adição, o quadrado pode ser dividido em blocos de 32 bits para a execução. Na verdade, o quadrado pode ser imaginado como uma rotação à direita do argumento de 163 bits, como mostrado na Figura 4.3. Nesse caso, as palavras $a[0]$ e $a[5]$ são enviadas ao módulo, como mostrado na Figura 4.4. Com isso, o bit mais a direita de $a[0]$ deve ser armazenado em um registrador interno para ser utilizado na segunda chamada. Além desses, os dois bits mais a esquerda de $a[5]$ devem também ser armazenados para que possam ser utilizados na última chamada. Um número é também enviado ao módulo para informá-lo em qual parte do argumento se está operando, e assim, quando utilizar os valores armazenados anteriormente.

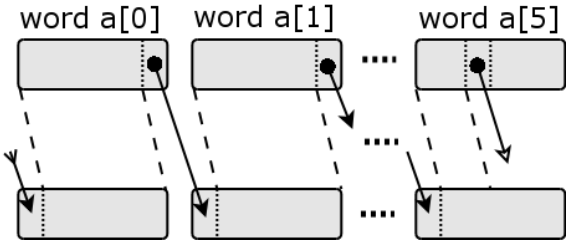


Figura 4.3: Diagrama em Blocos da Operação Quadrado

```
c[0] = SQUARE( 0, a[0], a[5] );
c[1] = SQUARE( 1, a[1], 0 );
c[2] = SQUARE( 2, a[2], 0 );
c[3] = SQUARE( 3, a[3], 0 );
c[4] = SQUARE( 4, a[4], 0 );
c[5] = SQUARE( 5, 0,    0 );
```

Figura 4.4: Chamadas à Instrução Especializada de Quadrado

4.1.3 Raiz Quadrada

Assim como o quadrado, a operação de raiz quadrada também pode ser imaginada como uma rotação. Entretanto, essa operação realiza uma rotação do argumento a esquerda, como mostrado na Figura 4.5. A raiz quadrada também necessita de armazenamento interno. Sua primeira chamada deve carregar as palavras $a[0]$ e $a[1]$ como se pode observar na Figura 4.6. Dessa maneira, o bit mais a esquerda de $a[0]$ deve ser mantido em um registrador interno para ser utilizado na última chamada, enquanto $a[1]$ deve ser armazenado para ser utilizado na segunda chamada (com exceção de seu bit mais a esquerda).

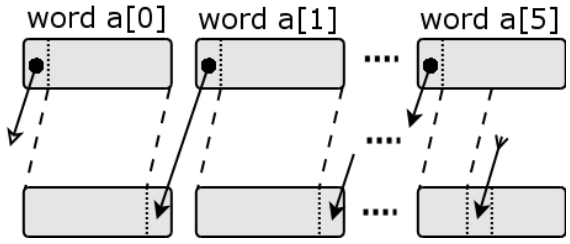


Figura 4.5: Diagrama em Blocos da Operação Raiz Quadrada

```
c[0] = SQUARE_ROOT( 0, a[0], a[1] );
c[1] = SQUARE_ROOT( 1, a[2], 0 );
c[2] = SQUARE_ROOT( 2, a[3], 0 );
c[3] = SQUARE_ROOT( 3, a[4], 0 );
c[4] = SQUARE_ROOT( 4, a[5], 0 );
c[5] = SQUARE_ROOT( 5, 0,    0 );
```

Figura 4.6: Chamadas à Instrução Especializada de Raiz Quadrada

A partir da análise das estratégias de chamadas, pode-se esperar que a operação raiz quadrada ocupe uma área maior que o quadrado, dado que ele precisa de registradores internos maiores para armazenar dados temporários.

4.1.4 *Trace*

A operação *trace* pode ser resumida como o *xor* de todos os 163 bits do argumento, como mostrado na Figura 4.7. Essa operação também pode ser calculado em blocos de 32 bits. Porém, se faz necessário armazenar o resultado parcial do cálculo em um registrador interno de 1 bit, o qual será utilizado nas próximas chamadas. Dado que o *trace* é uma operação unária, pode-se passar ao módulo por iteração, 64 bits do argumento. Como resultado, apenas três chamadas ao módulo são necessárias para computar o trace de um elemento, como mostrado na Figura 4.8.

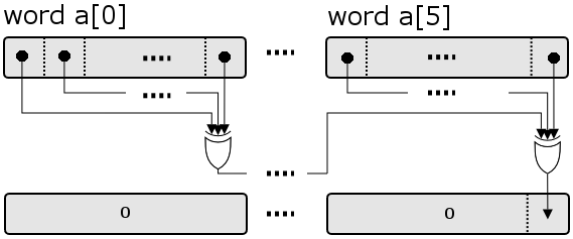


Figura 4.7: Diagrama em Blocos da Operação *Trace*

```
TRACE( 0, a[0], a[1] );
TRACE( 1, a[2], a[3] );
c = TRACE( 2, a[4], a[5] );
```

Figura 4.8: Chamadas à Instrução Especializada de *Trace*

4.1.5 Solução de Equação Quadrática

A solução da equação quadrática, calculada através do Algoritmo 1, é outra operação que pode ser computada em blocos de 32 bits, como mostrado na Figura 4.9. Para isso deve-se manter um bit de resultado da chamada anterior em um registrador interno. Assim, cada chamada ao módulo retorna 32 bits do resultado, como mostrado na Figura 4.10.

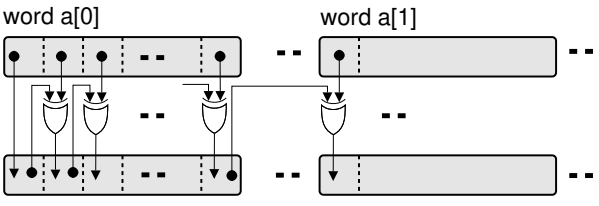


Figura 4.9: Diagrama em Blocos da Operação Solução de Equação Quadrática

```
c[0] = SOLVE_QE( 0, a[0], 0 );
c[1] = SOLVE_QE( 1, a[1], 0 );
c[2] = SOLVE_QE( 2, a[2], 0 );
c[3] = SOLVE_QE( 3, a[3], 0 );
c[4] = SOLVE_QE( 4, a[4], 0 );
c[5] = SOLVE_QE( 5, a[5], 0 );
```

Figura 4.10: Chamadas à Instrução Especializada de Solução de Equação Quadrática

4.1.6 Multiplicação

Os métodos de multiplicação implementados em *hardware* são apresentados nos Algoritmos 2, 4, e 5. Como a multiplicação necessita de todos os bits dos argumentos antes de iniciar a sua execução, fica difícil dividir a execução em blocos de 32 bits. Sendo assim, faz-se escrever os argumentos (completos) no módulo de *hardware*, para somente depois poder ler o resultado (também completo).

Independentemente do método de multiplicação utilizado, os mesmos parâmetros são utilizados para realizar as chamadas à instrução especializadas, mudando somente o nome da macro utilizada. A Figura 4.11 ilustra as chamadas à instrução de multiplicação.

Como se pode notar pela figura, faz-se necessário sinalizar para o módulo qual registrador interno está sendo escrito `0x1n` ou lido `0x0n`, onde `n` denota o número da palavra do registrador interno. Note que a sexta chamada dispara a execução multi-ciclo da multiplicação, que quando finalizada, deixa o resultado disponível no registrador interno IRC. Como se retorna 32 bits do resultado ($c[0]$) ao final da execução, cinco chamadas adicionais ao módulo são necessárias para que todo o resultado em IRC seja lido.

```
// Escrevendo os Argumentos
MULTIPLICATION( 0x10, a[0], b[0] );
MULTIPLICATION( 0x11, a[1], b[1] );
MULTIPLICATION( 0x12, a[2], b[2] );
MULTIPLICATION( 0x13, a[3], b[3] );
MULTIPLICATION( 0x14, a[4], b[4] );
c[0] = MULTIPLICATION( 0x15, a[5], b[5] );

// Lendo o Resultado
c[1] = MULTIPLICATION( 0x01, 0, 0 );
c[2] = MULTIPLICATION( 0x02, 0, 0 );
c[3] = MULTIPLICATION( 0x03, 0, 0 );
c[4] = MULTIPLICATION( 0x04, 0, 0 );
c[5] = MULTIPLICATION( 0x05, 0, 0 );
```

Figura 4.11: Chamadas à Instrução Especializada de Multiplicação

A multiplicação proposta pelo NIST computa um bit do resultado por iteração do algoritmo, como mostrado na Linha 2.1.1 do Algoritmo 2. Por razões de desempenho, aquelas operações foram transformadas em um circuito combinatório consistindo das operações *and* e *xor*. Essa característica nos permite explorar paralelismo de forma a acelerar o cálculo da multiplicação, dado que podemos calcular mais de um bit do resultado por ciclo de *clock*. O diagrama em blocos do algoritmo original de multiplicação proposto

pelo NIST é mostrado na Figura 4.12.

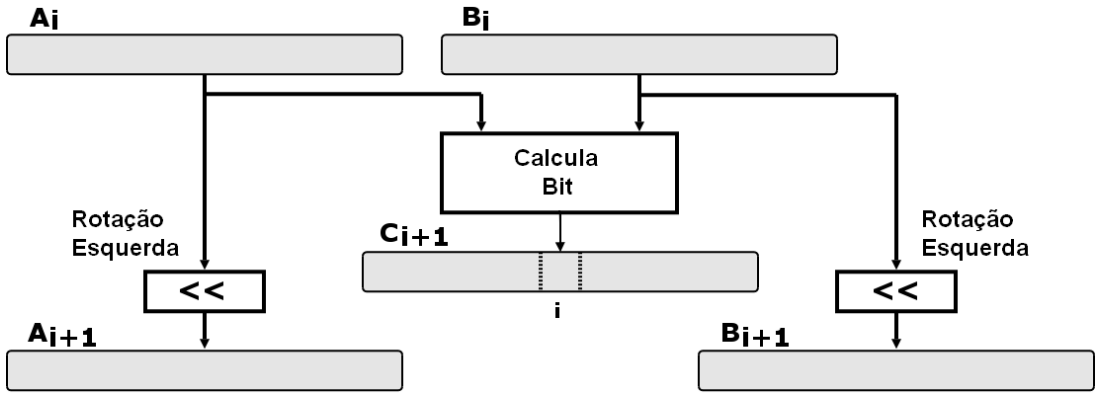


Figura 4.12: Diagrama em Blocos da Multiplicação NIST

Usando uma abordagem baseada na função β, discutida no Capítulo 2, cuja implementação em *hardware* é apresentada na Figura 4.13, a multiplicação López resulta em um bom mapeamento em *hardware* como mostrado na Figura 4.14. Devido a natureza do método, é possível explorar o desenrolamento de laços do algoritmo, fazendo com que mais de uma iteração seja executada por ciclo de *clock*.

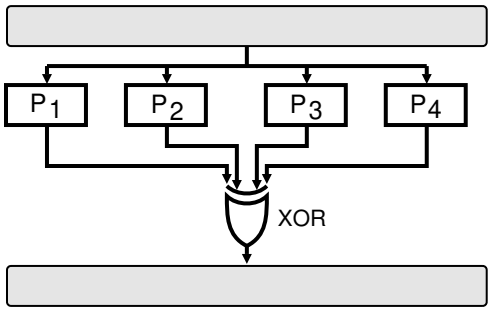


Figura 4.13: Diagrama em Blocos da Função β

Similarmente ao algoritmo de López, o algoritmo modificado do NIST se baseia na função β. Entretanto, ele ainda calcula um bit do resultado por iteração. Devido às permutações, esse método, mostrado na Figura 4.15, resulta em uma implementação mais eficiente em *hardware*, em termos de velocidade, que o algoritmo original do NIST. Além disso, esse método também permite explorar paralelismo com cômputo da multiplicação.

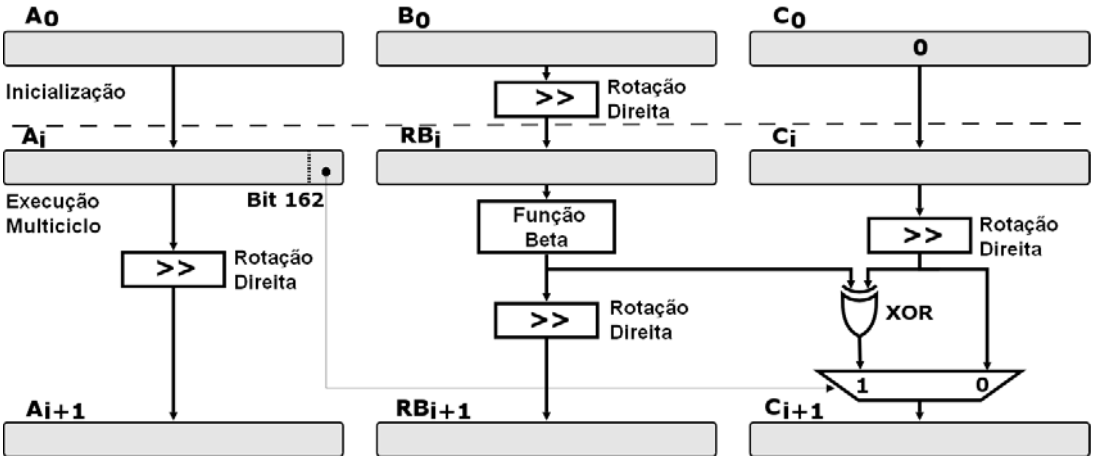


Figura 4.14: Diagrama em Blocos da Multiplicação López

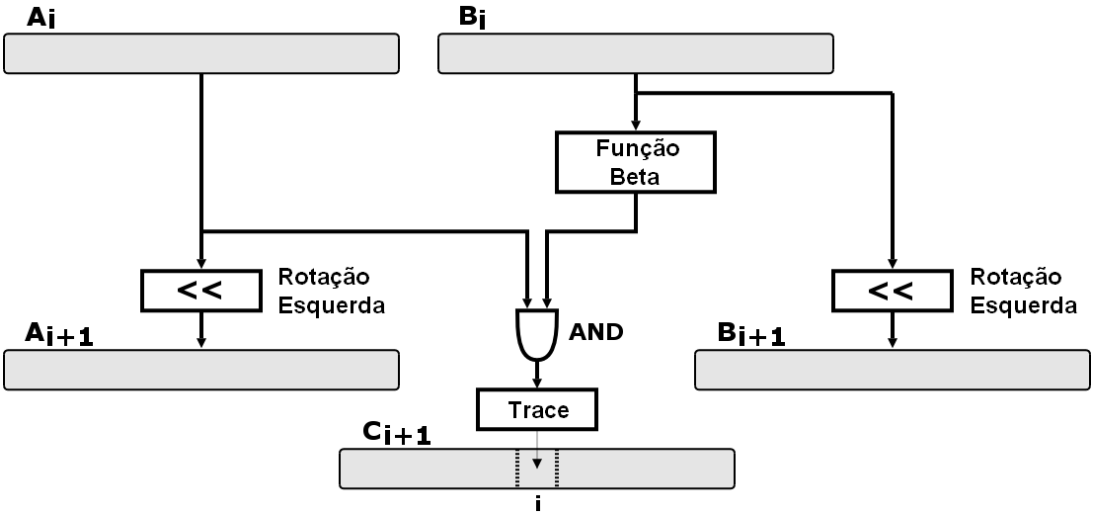


Figura 4.15: Diagrama em Blocos da Multiplicação NIST Modificada

4.1.7 Inversão

Tomando como base o método proposto por Itoh and Tsujii [4] (Algoritmo 6), o inverso de um elemento pode ser calculado em termos de somas, quadrados, e multiplicações em corpos binários, como mostrado na Figura 4.16. A implementação em *hardware* dessa operação consiste basicamente de uma máquina de estados controlando o fluxo dos dados através dos módulos de adição, quadrado e multiplicação. De forma a tornar a implementação em *hardware* mais eficiente, desenrolamos o laço da linha 2 do Algoritmo 6.

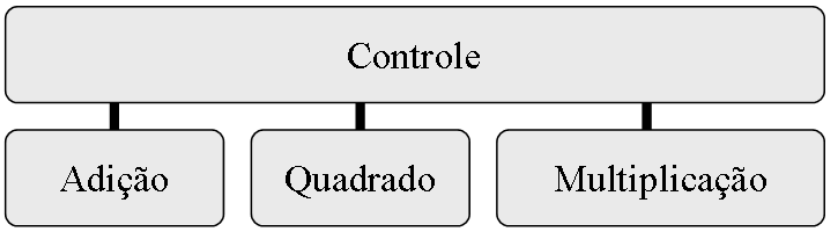


Figura 4.16: Diagrama em Blocos da Inversão

Devido ao grande número de multiplicadores que implementamos, tivemos a possibilidade de experimentar com várias configurações para os módulos inversores. Assim como os multiplicadores, a inversão não pode ser partida em blocos de 32 bits. Dessa forma, faz-se necessário realizar chamadas de escrita de parâmetros e de leitura do resultado.

Como detalhado adiante, os módulos inversores implementados não atingiram desempenho suficiente para serem inseridos no *datapath* do processador. Como conseqüência, tem-se um indício de que a inversão ultrapassou o limiar das operações mais apropriadas a se tornarem instruções em um processador especializado. Como resultado de seus tamanhos e freqüência de operação, a inversão em corpos finitos e a multiplicação de ponto serão exploradas no Capítulo 5 utilizando-se a abordagem de periféricos do processador NIOS2. No presente capítulo explora-se apenas os desempenhos *stand-alone* dos inversores, e abordagens *hardware*/*software* para o cálculo da inversão e de kP.

4.2 Análise dos Módulos *Stand-Alone*

Nessa seção avaliam-se os módulos *stand-alone* em termos de desempenho e área, quando os mesmos são implementados para seus usos posteriores como instruções especializadas. Consideramos aqui somente os módulos multiplicadores e inversores em corpos finitos, dado que seria difícil comparar os desempenhos dos módulos das operações divisíveis, uma vez que essas operam em blocos de 32 bits e não nos 163 bits dos argumentos. Dessa forma, postergamos essas análises para a Seção 4.3, ou seja, quando são usadas em um sistema real na forma de instruções especializadas.

Na avaliação de desempenho dos módulos *stand-alone* não consideramos os custos de passagem de parâmetros e de leitura/escrita de dados. Tal abordagem será tratada na Seção 4.3, onde todos os *overheads* são considerados em execuções de aplicações reais. Fizemos isso propositalmente para poder comparar os métodos de medida de desempenho (*stand-alone* e *end-to-end*), e posteriormente mostrar que é necessário ter cuidado com as medidas de desempenho, pois elas podem não se concretizar em aplicações reais.

4.2.1 Multiplicação

Iniciando nossas análises pelos multiplicadores, temos na Tabela 4.1 as áreas de implementação e os desempenhos de cada multiplicador implementado. Considerando a versão básica de cada método de multiplicação, isto é, sem explorar paralelismo nem desenrolar laços dos algoritmos, nota-se que as multiplicações de López e a NIST modificada resultam em melhores desempenhos que o método original proposto pelo NIST. De fato, o método López provê simultaneamente a menor área (845 ALUTs), a maior freqüência de operação (284,58MHz) e o melhor desempenho (572,77ns para calcular uma multiplicação).

Permitindo o uso de mais área, e mantendo os módulos acima de 120MHz, experimentamos com paralelismo e com desenrolamento de laços dos algoritmos para implementar os multiplicadores. De forma a denotar o uso de paralelismo, o nome do módulo carrega um número indicando o nível de paralelismo, seguido de um p. Por exemplo, NIST 9p indica que se utilizou o algoritmo proposto pelo NIST computando nove bits do resultado por ciclo de *clock*. No caso de desenrolar o algoritmo, o nome do algoritmo carrega o número de vezes que ele foi desenrolado seguido por um u, por exemplo, López 27u.

Retornando à Tabela 4.1, observa-se que o método NIST pode computar simultaneamente até 9 bits do resultado por ciclo de *clock*. Porém, a melhor implementação paralela resulta do método NIST modificado, dado que essa pode computar até 27 bits do resultado por ciclo de *clock*. Por sua vez, o laço do algoritmo de López pode ser desenrolado até 27 vezes, ou seja, executa 27 iterações do algoritmo por ciclo de *clock*.

Método de Multiplicação	Área (ALUTs)	Desempenho (ns)	Freqüência (MHz)
NIST	1036	645,16	252,65
NIST 6p	3750	223,84	125,09
NIST 9p	4826	151,73	125,22
López	845	572,77	284,58
López 6u	2623	157,16	178,16
López 9u	3494	108,02	175,90
López 18u	5833	70,89	141,06
López 27u	8075	53,54	130,74
NIST Modif.	1319	630,66	258,46
NIST Modif. 6p	2397	187,07	149,68
NIST Modif. 9p	3429	146,02	130,12
NIST Modif. 18p	5885	82,57	121,11
NIST Modif. 27p	7056	57,07	122,65
Gura et al. [18] †	-	900	66,4
Trujillo et al. [40] ‡	-	673	82,43

† Usando bases polinomiais e uma FPGA Xilinx Virtex XCV2000E-FG680-7

‡ Usando GNB e uma FPGA Altera EP2A70B724C7

Tabela 4.1: Instruções Especializadas: Módulos *Stand-Alone* dos Multiplicadores

Comparando os multiplicadores entre si, observa-se o fato de que, embora as multiplicações López 27u e NIST Modif. 27p requererem o mesmo número de ciclos de *clock* para computar uma multiplicação, a López 27u opera em freqüências mais altas, realizando a multiplicação em 53,54ns. Como resultado, tem-se esse módulo como o multiplicador mais eficiente.

Considerando a área de implementação, nota-se que o multiplicador López 27u é quase 10 vezes maior que sua versão básica, e roda mais de 10 vezes mais rápido que aquela. Com essa mesma análise, observa-se que o multiplicador modificado NIST 27p oferece melhores ganhos de desempenho por área, dado que ele é aproximadamente 5 vezes maior que sua versão básica, porém roda mais de 11 vezes mais rápido.

Comparando-se os multiplicadores implementados com as referências [18] e [40], concluímos que nossas implementações são mais eficientes em termos de desempenho. Infelizmente, não foi possível comparar com as áreas desses trabalhos relacionados.

4.2.2 Inversão

Da mesma maneira que os multiplicadores, iniciamos a análise dos inversores por suas versões básicas, isto é, sem explorar paralelismo nem desenrolar laços de algoritmos. Nesse caso, como mostrado na Tabela 4.2, o inversor baseado na multiplicação López tem o pior desempenho (8,76μs) e a maior área de implementação. Mais, o inversor usando a multiplicação NIST provê área de implementação satisfatória (2483 ALUTs), o melhor desempenho (8,53μs), e a melhor freqüência de operação (173,16 MHz) entre as versões básicas dos inversores.

Por outro lado, torna-se interessante analisar quão rápido um inversor pode ser quando utilizamos paralelismo. Sob esse ponto de vista, o inversor NIST Modif. 9p possui um bom nível de paralelismo, computando 9 bits do resultado por ciclo de *clock*, fazendo com que uma inversão seja calculada em 1,47μs, quando executando a 123,24MHz. Nesse caso, dobra-se aproximadamente o tamanho de sua versão básica enquanto executa-se somente 6 vezes mais rápido. Quando se utiliza o multiplicador López 18u, uma inversão é computada em 0,82μs quando executando a 121,33MHz. Essa implementação ocupa 3 vezes mais área que sua versão básica, porém roda mais de 10 vezes mais rápido. Apesar de esse ser o inversor mais eficiente, ele também foi o que ocupou maior área (7750 ALUTs).

Inversão usando Multiplicador	Área (ALUTs)	Desempenho (μs)	Freqüência (MHz)
NIST	2483	8,53	173,16
NIST 6p	-	-	*
NIST 9p	-	-	*
López	2694	8,76	168,63
López 6u	4691	1,77	147,67
López 9u	5466	1,22	147,95
López 18u	7750	0,82	121,33
López 27u	-	-	*
NIST Modif.	2309	8,61	171,64
NIST Modif. 6p	4022	1,97	132,75
NIST Modif. 9p	4874	1,47	123,24
NIST Modif. 18p	-	-	*
NIST Modif. 27p	-	-	*
Gura et al. [18] †	-	14,13	66,4
Trujillo et al. [40] ‡	-	10	82,43

* Módulo incapaz de rodar acima de 120MHz

† Usando bases polinomiais e uma FPGA Xilinx Virtex XCV2000E-FG680-7

‡ Usando GNB e uma FPGA Altera EP2A70B724C7

Tabela 4.2: Instruções Especializadas: Módulos *Stand-Alone* dos Inversores

Similarmente aos multiplicadores, nossos inversores são mais eficientes em termos de desempenho do que as implementações em [18] e [40]. Novamente, não foi possível comparar com as áreas desses trabalhos.

4.3 Análise dos Sistemas com Instruções Especializadas

De agora em diante nesse capítulo, consideraremos as operações sobre $\mathbb{F}_{2^{163}}$ implementadas como instruções especializadas em um processador NIOS2. Dessa maneira, devido ao aumento da área do sistema (NIOS2 mais instrução especializada), temos uma menor freqüência de operação quando comparando com os módulos *stand-alone*. Como fixamos a freqüência do processador em 120MHz, sistemas executando abaixo dessa freqüência serão descartados. Tomemos o exemplo do módulo *stand-alone* da multiplicação NIST Modif. 27p, que executa a 122,65MHz. Quando esse módulo é sintetizado como uma instrução especializada no interior do *datapath* do NIOS2, a freqüência de operação do sistema fica abaixo de 120MHz. Assim, tal operação é descartada.

A seção anterior e outros trabalhos analisam as implementações das operações em uma abordagem *stand-alone*, de forma que vários *overheads* de aplicações reais não são considerados. Consequentemente, as implementações em *hardware* são favorecidas em termos de desempenho, dado que uma situação *ideal* é considerada. Em nossa abordagem de instrução especializada, por estar utilizando apenas uma instrução por processador, podemos determinar precisamente as características (como por exemplo *speedup* e área do sistema) para cada uma das operações.

Para determinar os *speedups*, medimos os desempenhos das implementações *software* e *hardware* executando programas escritos na linguagem de programação C. É importante salientar que todos os programas de teste foram altamente otimizados e compilados com o compilador GCC (versão 3.4.1) portado para o processador NIOS2. Nesse experimentos utilizamos o nível de otimização `-O2`, que produziu, na maioria dos casos, os programas mais rápidos.

Em nossas medições de desempenho utilizamos uma abordagem *end-to-end*, não interessando se uma determinada operação sob teste foi implementada em *software* ou em *hardware*. Isso significa que todos os *overheads* são considerados, por exemplo, carga dos operandos da memória, comunicação com o módulo através de várias chamadas de instruções especializadas, escrita dos resultados na memória, etc. Como resultado, podemos comparar de forma mais justa implementações *software* e *hardware*, e assim, ter uma idéia mais precisa de quais serão os *speedups* de tais implementações, quando utilizadas em um criptosistema real.

Por fim, de modo a economizar área, experimentamos com abordagens mistas *soft-*

ware/hardware, em vários níveis de implementação (multiplicação e inversão em corpos finitos, e multiplicação de ponto), para determinar quais operações deveriam ser movidas para *hardware* e quais deveriam ser mantidas em *software*.

No caso de instruções especializadas não se consegue implementar toda a operação de inversão e a multiplicação de pontos (kP) em *hardware*, e ainda manter o sistema executando acima de 120Mhz. Dessa forma, para o caso da inversão e kP, utilizamos multiplicadores binários em *hardware* e o restante das operações sendo executada em *software*. Já para o caso dos periféricos, como será mostrado no Capítulo 5, consegue-se ter um sistema com toda a inversão e a multiplicação de ponto implementados em *hardware*. Entretanto, nesse caso, ainda experimentamos com abordagens *hardware/software* para o núcleo de kP que é a parte mais cara do algoritmo. Mais precisamente, implementamos os laços internos de kP em *hardware*, sendo o restante do algoritmo executado em *software*.

Analisamos o *speedup* do *hardware* sobre o *software*, a área de implementação, como também a razão *speedup*/área, sendo esse último um importante parâmetro de decisão para ambientes restritos. Aqui se justifica um segundo motivo de se utilizar apenas uma operação por processador; caso mais uma operação fosse utilizada, não poderíamos analisar o *speedup*/área das instruções individualmente.

4.3.1 Operações Elementares

Os desempenhos *end-to-end* das operações mais simples em corpos finitos são listados na Tabela 4.3. A partir dessa tabela nota-se que a operação de adição não proporciona nenhum ganho de desempenho. Na verdade, isso ocorre pois ela é uma função *xor* de 32 bits, e o processador NIOS2 já possui tal operação lógica.

As implementações em *hardware* das operações de quadrado e raiz quadrada também não resultaram em bons *speedups*. Embora essas operações, que são na verdade rotações, possam ser implementadas eficientemente em *hardware*. O ponto chave é que as implementações em *software* se baseiam nas instruções de deslocamentos a esquerda e direita presentes no conjunto de instruções original do NIOS2, produzindo eficiência satisfatória em *software*.

O *trace* é outro exemplo de baixo ganho de desempenho quando implementado em *hardware*. Como essa operação pode ser implementada em *software* utilizando-se apenas deslocamentos, *ands* e *xors*, suas implementações em *software* já apresentam desempenho satisfatório.

Porém, observa-se que o módulo que calcula a solução de equação quadrática resulta em um bom *speedup* devido à sua boa implementação em *hardware*, sendo executada 4,81 vezes mais rápido que sua versão em *software*. Essa operação opera bit a bit, e para

implementá-la em *software* deve-se fazer uso de operações de deslocamento, *ands*, *ors*, e de tabelas. Dessa maneira, tem-se dificuldade de se utilizar o conjunto de instruções original do NIOS2 para se obter desempenho satisfatório em *software*.

Operação	*Software* (μs)	*Hardware* (μs)	*Speedup*
Adição	0,44	0,44	1,00
Quadrado	0,60	0,51	1,18
Raiz Quadrada	0,52	0,50	1,03
Trace	0,43	0,28	1,50
Solução de E.Q.	2,53	0,53	4,81
Função β	27,25	0,51	53,61

Tabela 4.3: Instruções Especializadas: Desempenho e *Speedups* das Operações Elementares

Operação	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)
Adição	3596	2,78
Quadrado	3593	3,29
Raiz Quadrada	3851	2,68
Trace	3834	3,91
Solução de E.Q.	3497	13,75
Função β	4482	119,60

Tabela 4.4: Instruções Especializadas: Áreas de Sistema e *Speedups*/Área das Operações Elementares

Em contraste, a função β produz o melhor *speedup* (53,61) entre as operações mais simples, fato explicado pela maneira eficiente que as permutações podem ser implementadas em *hardware*, consistindo basicamente de um conjunto de fios e uma função *xor*. Sua implementação em *software* é relativamente lenta, uma vez que se fica restrito a uma enorme seqüência de deslocamentos, *and*s, *or*s e *xor*s.

Quando utilizando criptografia em ambientes restritos, tal como *smart-cards* ou pequenos sensores, a área de implementação se torna um ponto a ser considerado com atenção. Assim, a razão *speedup*/área pode ser considerado como um fator de decisão para o que deve ser mantido em *software* e o que deve ser movido para *hardware*. Nesse contexto, quanto mais alta a razão de *speedup*/área, mais propícia é a implementação da operação em *hardware*.

A Tabela 4.4 apresenta a área de implementação do sistema, composto pelo NIOS2 mais instrução especializada, e a razão *speedup*/área do sistema.

Devido aos reduzidos ganhos de desempenho das operações de adição, quadrado, raiz quadrada e *trace*, além de suas áreas de sistema bastante semelhantes, suas razões *speedup*/área do sistema não ultrapassaram (4×10^{-4}). Em outras palavras, essas operações não são apropriadas para ambientes restritos em área.

A operação de solução de equação quadrática não resulta em um alto *speedup*. Porém ela possui uma razoável razão *speedup*/área de sistema ($13{,}75\times10^{-4}$) devido à sua reduzida área de implementação.

Por fim, analisando a função β notamos que, embora ocupe maior área que as outras operações elementares, ela provê uma boa razão ($119{,}60\times10^{-4}$), resultante de seu bom *speedup*.

É evidente que a implementação de uma operação em *hardware* leva a um ganho de desempenho, porém é importante atentar para o caso de uma implementação em *hardware* prover *speedup* satisfatório pelo baixo desempenho em *software*, e não por uma implementação eficiente em *hardware*. Além disso, para ambientes restritos, sugere-se a utilização de operações que possuam altas razões *speedup*/área de implementação. Caso contrário, um sistema pode prover altíssimo *speedup* e ainda ter seu uso inviabilizado em um sistema criptográfico com restrições de área devido à sua grande área de implementação.

4.3.2 Multiplicação

As operações apresentadas até agora, exceto a função β, não resultaram em bons *speedups*. Porém, essas não são operações tão críticas para a multiplicação de ponto, como é a multiplicação em corpos finitos. De forma a analisar os ganhos de desempenho de nossas implementações, compararemos nossos multiplicadores em *hardware* com sua respectiva implementação em *software*.

O algoritmo de DHHLLM [47] é o método mais rápido para implementações em *software* de multiplicações usando bases normais Gaussianas. Diga-se de passagem que somente implementamos esse algoritmo em *software*, dado que o seu mapeamento para *hardware* não é muito apropriado. Isso porque, diferentemente dos métodos de NIST, López, e NIST modificado, ele não apresenta muita regularidade de operações, dificultando assim, o reaproveitamento de circuitos combinacionais, e por conseqüência, resultando em áreas grandes de implementação.

Por outro lado, comparando os desempenhos de nossas implementações em *hardware* com o desempenho de DHHLLM em *software* rodando no NIOS2, teremos a indicação de

qual implementação em *hardware* provê o melhor *speedup* em face da melhor multiplicação em *software*. Denominados esse *speedup* relativo ao DHHLLM como *speedup* normalizado. Dessa forma, garantimos que esses *speedups* não são provenientes de um algoritmo de desempenho deficiente em *software*, mas sim de uma boa implementação em *hardware*.

Multiplicador	*Software* (μs)	*Hardware* (μs)	*Speedup*	*Speedup* Normalizado
DHHLLM	156,33	-	-	-
López *	2309,77	178,78	12,92	0,87
NIST Modif. *	4233,08	205,67	20,58	0,78
NIST	7222,03	2,04	3537,32	76,57
NIST 6p	7222,03	0,91	7950,86	172,10
López	2309,77	2,03	1135,95	76,88
López 6u	2309,77	0,90	2566,41	173,69
López 9u	2309,77	0,83	2799,72	189,48
López 18u	2309,77	0,75	3079,69	208,43
NIST Modif.	4233,08	2,03	2090,41	77,20
NIST Modif. 6p	4233,08	0,90	4703,42	173,69

* Multiplicador implementado utilizando somente a função β em *hardware*

Tabela 4.5: Instruções Especializadas: Desempenho e *Speedups* dos Multiplicadores

Como apresentado na Tabela 4.5, o método DHHLLM realiza a multiplicação em 156,33μs quando executando em um processador NIOS2 a 120MHz. Como os algoritmos de López e NIST modificado utilizam a função β, adotamos inicialmente para esses algoritmos uma abordagem *hardware/software*. Nessa abordagem, de forma a economizar área, apenas a função β é executada em *hardware*, sendo o restante do algoritmo executado em *software*. Como resultado, os métodos López e NIST Modif. realizaram uma multiplicação em, respectivamente, 178,78μs and 205,67μs.

Se analisarmos os seus *speedups* vemos que são razoavelmente bons, 12,92 e 20,58, respectivamente. Porém, analisando os *speedups* normalizados, observamos com desapontamento que esses multiplicadores são mais lentos que o DHHLLM. Isso nos leva a concluir que não vale a pena utilizar a função β como instrução especializada dentro desses métodos para multiplicação. Ou seja, vale mais a pena manter o NIOS2 em sua forma original e utilizar o algoritmo DHHLLM, do que gastar mais área na esperança de acelerar os algoritmos de López e NIST modificado.

Implementando os multiplicadores inteiramente em *hardware*, notamos que os *speedups* aumentam significativamente, sendo todos eles superiores a 1100. O maior *speedup* é obtido pelo multiplicador NIST 6p (7950,86) devido ao baixo desempenho de seu algoritmo em *software*. Esse multiplicador é mais de 173 vezes mais rápido que o DHHLLM.

Interessantemente, devido a seu bom desempenho em *software*, o multiplicador López 18u resulta em um menor *speedup* (3079,69) que o NIST 6p. Porém seu *speedup* normalizado revela que esse multiplicador possui o melhor desempenho em *hardware* dado que ele é mais de 208 vezes mais rápido que o DHHLLM. Mais especificamente, ele computa uma multiplicação em 0,75μs. Em contraste, a Tabela 4.1 mostra que a sua versão *stand-alone* executa a mesma operação em 53,54ns, porém, sem considerar nenhum *overhead*. Devido a essas enormes diferenças, ficam claros os riscos de se selecionar algoritmos para implementação de sistemas em abordagens *stand-alone*.

Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
López *	4482	28,82	1,95
NIST Modif. *	4482	45,92	1,70
NIST	4891	7232,31	156,55
NIST 6p	7110	11182,65	242,05
López	4626	2455,58	166,19
López 6u	6549	3918,78	265,22
López 9u	7176	3901,50	264,05
López 18u	9649	3191,72	216,02
NIST Modif.	4356	4798,92	177,22
NIST Modif. 6p	5978	7867,88	290,56

* Multiplicador implementado utilizando somente a função β em *hardware*

Tabela 4.6: Instruções Especializadas: Áreas de Sistema e *Speedups*/Área dos Multiplicadores

Pensando em ambientes restritos, temos que a menor área é obtida quando se utiliza o multiplicador NIST Modif., ou seja, 4356 ALUTs. Porém, pensando na relação de compromisso entre o ganho de desempenho e a área de implementação, observa-se que a melhor razão *speedup*/área é obtida quando utilizamos o multiplicador NIST 6p, como mostrado na Tabela 4.6. Essa razão ($11182{,}65\times10^{-4}$) é mais de 3 vezes maior que a razão do eficiente multiplicador López 18u ($3191{,}72\times10^{-4}$).

Por outro lado, tomando o *speedup* normalizado/área, vemos que o multiplicador NIST 6p resulta em uma razão de $242{,}05\times10^{-4}$. Já López 18u e a modificada NIST 6p, possuem respectivamente razões $216{,}02\times10^{-4}$ e $290{,}56\times10^{-4}$. Dessa forma, concluí-se que a multiplicação NIST Modif. 6p é mais adequada para sistemas com restrições de área.

4.3.3 Inversão

Essa seção destina-se à avaliação da operação de inversão em corpos finitos. Infelizmente, quando sintetizamos os módulos inversão como instruções especializadas no processador NIOS2, nenhum dos sistemas foi capaz de rodar acima de 120MHz. Porém, para tornar a análise da inversão possível, utilizamos a abordagem *hardware/software*.

Inversão usando Multiplicador	*Software* (μs)	*Hardware/Software* (μs)	*Speedup*	*Speedup* Normalizado
DHHLLM	1456,43	-	-	-
López *	20009,13	1696,00	11,80	0,86
NIST Modif. *	38135,97	1907,74	19,99	0,76
NIST	65018,64	75,44	861,84	19,31
NIST 6p	65018,64	65,32	995,44	22,30
López	20009,13	75,44	265,23	19,31
López 6u	20009,13	65,23	306,77	22,33
López 9u	20009,13	64,56	309,94	22,56
López 18u	20009,13	63,88	313,25	22,80
NIST Modif.	38135,97	75,46	505,39	19,30
NIST Modif. 6p	38135,97	65,31	583,94	22,30

* Inversor implementado em *software* usando somente a função β em *hardware*

Tabela 4.7: Instruções Especializadas: Desempenho e *Speedups* dos Inversores utilizando Abordagem HW/SW

Iniciou-se com a função β implementada em *hardware* como instrução especializada, e o restante da inversão em *software*. Nesse caso, observa-se na Tabela 4.7 que, como na multiplicação, o uso da função β resulta em implementações com *speedups* satisfatórios. Mais precisamente, quando se utilizou os algoritmos López e NIST Modif. no algoritmo de inversão, obteve-se *speedups* de 11,80 e 19,99, respectivamente. Porém, analisando-se os *speedups* normalizados, nota-se que o cômputo da inversão, com somente a função β em *hardware*, foi mais lenta do que se tivesse utilizado o método DHHLLM e o processador NIOS2 original. Utilizando o método DHHLLM, a inversão é computada em 1456,43μs.

Usando o multiplicador implementado como uma instrução especializada, tem-se a inversão alcançando o maior *speedup* (995,44) quando utilizando o multiplicador NIST 6p. Porém, esse valor é resultado da implementação deficiente em *software* desse método (65,32ms), pois seu *speedup* normalizado (22,30) não o aponta como o melhor método a ser utilizado para inversão.

Tomando-se o multiplicador mais rápido, López 18u, para calcular a inversão, se obtêm um *speedup* menor (313,25) que o obtido com o multiplicador NIST 6p. Porém, López 18u provê o melhor *speedup* normalizado (22,80), dentre os outros multiplicadores. Como resultado, tem-se novamente López 18u como mais rápido multiplicador em *hardware*, sendo que uma inversão usando esse multiplicador em uma abordagem *hardware/software* é executada em 63,88μs.

Inversão usando Multiplicador	Área do Sistema (ALUTs)	*Speedup*/Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
López *	4482	26,32	1,92
NIST Modif. *	4482	44,60	1,70
NIST	4891	1762,09	39,47
NIST 6p	7110	1400,05	31,36
López	4626	573,34	41,73
López 6u	6549	468,42	34,10
López 9u	7176	431,91	31,44
López 18u	9649	324,65	23,63
NIST Modif.	4356	1160,22	44,31
NIST Modif. 6p	5978	976,81	37,30

* Inversor implementado em *software* usando somente a função β em *hardware*

Tabela 4.8: Instruções Especializadas: Áreas de Sistema e *Speedups*/Área dos Inversores utilizando Abordagem HW/SW

Analisando a Tabela 4.8, nota-se que a inversão com melhor razão speedup/área é obtida quando usando o multiplicador NIST. Sua razão (1762,09$\times 10^{-4}$) é mais de 5 vezes maior que a do López (324,65$\times 10^{-4}$).

Entretanto, quando considerando o *speedup* normalizado/área do sistema, o maior valor (44,31$\times 10^{-4}$) é obtido com o inversor usando a multiplicação NIST Modif. Novamente, no caso da inversão, o método modificado NIST se apresenta como uma boa opção para sistema com restrições de área.

4.3.4 Multiplicação de Ponto (kP)

Para implementar a multiplicação de ponto, selecionamos o algoritmo de López-Dahab [10] (Algoritmo 7), sendo esse um dos métodos mais rápidos conhecidos e baseado em coordenadas projetivas. Um dos aspectos mais importantes a ser analisado aqui é o quão rápido pode-se tornar o cálculo de kP, uma vez que se tem um multiplicador em corpos finitos eficientemente implementado em *hardware*.

Quando a multiplicação de ponto é inteiramente implementada em *software* usando o método DHHLLM, ela computa kP em 181,59ms, como mostrado na Tabela 4.9. Considerando-se o caso de se implementar somente a função β em *hardware*, nota-se que o *speedup* normalizado novamente indica que tais implementações são menos eficientes que usar DHHLLM junto do NIOS2 original.

Observa-se que, apesar da eficiente implementação em *hardware* da função β, as chamadas de instruções especializadas deterioram enormemente seu desempenho. Consequentemente, concluímos que não é vantajoso usar a função β em nenhum dos níveis (multiplicação e inversão em corpos finitos, e multiplicação de pontos), dado que seus desempenhos não são melhores do que quando se utiliza DHHLLM em *software*.

kP usando Multiplicador	*Software* (ms)	*Hardware* / *Software* (ms)	*Speedup*	*Speedup* Normalizado
DHHLLM	181,59	-	-	-
López *	2521,22	210,43	11,98	0,86
NIST Modif. *	4929,42	240,10	20,53	0,76
NIST	8409,08	2,94	2861,93	61,80
NIST 6p	8409,08	1,63	5170,94	111,67
López	2521,22	2,93	859,39	61,90
López 6u	2521,22	1,61	1564,38	112,68
López 9u	2521,22	1,52	1653,99	119,13
López 18u	2521,22	1,44	1754,52	126,37
NIST Modif.	4929,42	2,93	1680,31	61,90
NIST Modif. 6p	4929,42	1,62	3039,77	111,98

* Multiplicação de ponto implementado em *software* usando somente a função β em hardware

Tabela 4.9: Instruções Especializadas: Desempenho e *Speedups* da Multiplicação de Ponto utilizando Abordagem HW/SW

Quando se utiliza os multiplicadores em corpos finitos inteiramente em *hardware* o *speedup* cresce significativamente, como mostrado na Tabela 4.9. Nesse caso, observamos que o maior *speedup* (5170,94) é obtido novamente quando o multiplicador NIST 6p é

usado. Mas, da mesma forma que as análises anteriores, esse resultado é fruto de um desempenho não satisfatório *em software*.

Observa-se ainda que o multiplicador López 18u, embora não possuindo o melhor *speedup* (1754,52), oferece como uma melhor opção em desempenho para calcular kP, dado que possui o maior *speedup* normalizado (126,37). Usando esse multiplicador kP pode ser calculado em 1,44ms, isto é, 126,37 vezes mais rápido quando se utiliza DHHLLM.

Como regra geral, mostramos que se o objetivo é desempenho, não importa em qual nível, o multiplicador em corpos finitos López 18u se apresenta indubitavelmente como a melhor opção.

kP usando Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
López *	4482	26,73	1,93
Modif. NIST *	4482	45,81	1,69
NIST	4891	5851,41	126,36
NIST 6p	7110	7272,78	157,05
López	4626	1857,74	133,80
López 6u	6549	2388,73	172,05
López 9u	7176	2304,90	166,01
López 18u	9649	1818,34	130,97
NIST Modif.	4356	3857,45	142,10
NIST Modif. 6p	5978	5084,93	187,32

* Multiplicação de ponto implementado em *software* usando somente a função β em hardware

Tabela 4.10: Instruções Especializadas: Áreas de Sistema e *Speedups*/Área da Multiplicação de Ponto utilizando Abordagem HW/SW

Assim como na multiplicação e na inversão em corpos finitos, a melhor razão speedup/área do sistema ($7272{,}78\times10^{-4}$) para a multiplicação de pontos é obtida quando se utiliza o multiplicador em *hardware* NIST 6p, como mostrado na Tabela 4.10. Essa razão é mais de 4 vezes maior que a do multiplicador López 18u. Porém, a multiplicação NIST Modif. 6p possui novamente a melhor razão de *speedup* normalizado/área ($187{,}32\times10^{-4}$).

Dessa maneira, para sistemas restritos, o método NIST Modif. é recomendado para ser utilizado como instruções especializadas, não importa se ele vai ser usado na multiplicação ou inversão em corpos finitos, ou ainda na multiplicação de pontos.

4.3.5 Resumo da Análise das Operações

Essa seção resume as discussões sobre as implementações dos módulos de *hardware* como instruções especializadas do processador NIOS2. A Tabela 4.11 mostra os tempos gastos nas execuções das operações, e as áreas de implementação dos sistemas. No caso das operações que utilizam multiplicações em corpos finitos, o método de multiplicação é informado.

Observando essa tabela, conclui-se que a instrução especializada López 18u é a mais apropriada para sistemas voltados a velocidade, enquanto a NIST Modif 6p, no geral, é a mais indicada para sistemas com restrições de área.

Operação		Sistemas	
		Velocidade	Restritos
Adição	Desempenho	0,44μs	
	Área	3596 ALUTs	
Quadrado	Desempenho	0,51μs	
	Área	3593 ALUTs	
Raiz Quadrada	Desempenho	0,50μs	
	Área	3851 ALUTs	
Trace	Desempenho	0,28μs	
	Área	3834 ALUTs	
Solução de E.Q.	Desempenho	0,53μs	
	Área	3497 ALUTs	
Função β	Desempenho	0,51μs	
	Área	4482 ALUTs	
Multiplicação	Método	López 18u	NIST Modif. 6p
	Desempenho	0,75μs	0,90μs
	Área	9649 ALUTs	5978 ALUTs
Inversão (HW/SW)	Método	López 18u	NIST Modif.
	Desempenho	63,88μs	75,46μs
	Área	9649 ALUTs	4356 ALUTs
kP (HW/SW)	Método	López 18u	NIST Modif. 6p
	Desempenho	1,44ms	1,62ms
	Área	9649 ALUTs	5978 ALUTs

Tabela 4.11: Instruções Especializadas: Resumo das Análises das Operações

Capítulo 5

Periféricos

5.1 Implementação dos Módulos Aritméticos

Nessa seção apresentamos os detalhes de implementação das operações no corpo finito $\mathbb{F}_{2^{163}}$ usando a abordagem de periféricos sob a plataforma de trabalho padronizada. Na verdade, realizamos algumas alterações nos módulos de *hardware* que estavam implementados para serem utilizados como instrução especializada, de modo que esses pudessem ser utilizados como periféricos do NIOS2. Mais precisamente, alteramos as interfaces dos módulos das instruções especializadas (Figura 3.3) para periféricos (Figura 3.6), e devido a essa modificação, tivemos de adicionar alguns registradores internos para armazenamento temporário de dados.

Dessa forma, mantivemos as otimizações (desenrolamento de laços e paralelismo) utilizados anteriormente. Em alguns casos, por conta do maior grau de liberdade que os periféricos proporcionam, conseguimos até explorar mais paralelismo, tornando assim, os módulos mais rápidos. Similarmente às instruções especializadas, direcionamos as modificações tomando como base as características das operações divisíveis e não-divisíveis. Os Capítulos 2 e 4 mostram os algoritmos utilizados e maiores detalhes da implementação dos módulos de *hardware* que foram aqui reutilizados.

5.1.1 Adição

A adição implementada como periférico ainda é capaz de operar em blocos de 32 bits, como mostrado na Figura 4.1, e através de sucessivas iterações, realizar a soma dos operandos de 163 bits. Observa-se pela Figura 5.1 que o primeiro argumento (`CP_ADDITION_BASE`) das chamadas de escrita (`IOWR`) e leitura (`IORD`) é o endereço base do periférico no espaço de endereçamento do NIOS2.

Como o periférico possui apenas uma entrada de 32 bits para escrita dos argumen-

tos, são necessárias duas chamadas para enviar os dois argumentos para o periférico. Na primeira chamada, como mostrado na Figura 5.1, os 32 bits do argumento $a[0]$ são armazenados em um registrador interno ao periférico. No momento da segunda chamada, o periférico toma o argumento $b[0]$, e realiza a operação de *xor* com o valor de $a[0]$ armazenado internamente. Nota-se aqui que se faz necessário informar à lógica interna do periférico quando se está enviado os argumentos `a` e `b`. Em outras palavras, quando o segundo argumento das chamadas de escrita (`IOWR`) é `0`, o periférico armazena o valor em um registrador interno; quando é `1`, ele faz o *xor* entre o valor passado como argumento e o valor armazenado no registrador interno.

Em seguida à operação de *xor*, o resultado de 32 bits é armazenado temporariamente no registrador interno, ficando à espera da chamada de leitura (`IORD`), que coloca o valor em $c[0]$. Esse processo de duas escritas e uma leitura é realizado 6 vezes de forma a realizar a soma de 163 bits. Salientamos que essa estratégia é utilizada em todas as operações divisíveis, variando apenas, de acordo com a operação, o tamanho do registrador interno para armazenamento temporário de dados.

```
IOWR( CP_ADDITION_BASE, 0, a[0] );
IOWR( CP_ADDITION_BASE, 1, b[0] );
c[0] = IORD( CP_ADDITION_BASE, 0 );
IOWR( CP_ADDITION_BASE, 0, a[1] );
IOWR(CP_ADDITION_BASE, 1, b[1] );
c[1] = IORD( CP_ADDITION_BASE, 0 );
IOWR( CP_ADDITION_BASE, 0, a[2] );
IOWR( CP_ADDITION_BASE, 1, b[2] );
c[2] = IORD( CP_ADDITION_BASE, 0 );
IOWR( CP_ADDITION_BASE, 0, a[3] );
IOWR( CP_ADDITION_BASE, 1, b[3] );
c[3] = IORD( CP_ADDITION_BASE, 0 );
IOWR( CP_ADDITION_BASE, 0, a[4] );
IOWR( CP_ADDITION_BASE, 1, b[4] );
c[4] = IORD( CP_ADDITION_BASE, 0 );
IOWR( CP_ADDITION_BASE, 0, a[5] );
IOWR( CP_ADDITION_BASE, 1, b[5] );
c[5] = IORD( CP_ADDITION_BASE, 0 );
```

Figura 5.1: Chamadas ao Periférico de Adição

5.1.2 Quadrado

O quadrado, que pode ser pensado como uma rotação à direita do argumento de 163 bits, também mantém a característica de ser divisível quando implementado como periférico. De forma a reutilizar a implementação mostrada na Figura 4.3, e utilizá-la como periférico, faz-se necessário modificar sua interface, dado que periféricos contam com apenas uma entrada de dados de 32 bits.

A Figura 5.2 mostra a estratégia de chamadas ao periférico em blocos de 32 bits. O primeiro argumento das chamadas (`CP_SQUARE_BASE`) é o endereço do periférico. Já o segundo argumento indica sob qual bloco de 32 bits do argumento se está operando, orientando assim o módulo no armazenamento interno dos dados. Na primeira chamada envia-se ao periférico a palavra $a[5]$. Apenas três bits dessa palavra contém informação relevante, sendo que dois deles serão utilizados somente na última chamada. Dessa forma, implementamos o módulo com dois registradores: um registrador armazena o bit a ser utilizado na segunda chamada, enquanto o outro registrador armazena o valor (2 bits) a ser utilizado na última iteração.

Na segunda chamada, a palavra $a[0]$ é enviada ao módulo. O bit mais a direita de $a[0]$ é então armazenado para uso na quarta chamada, e os 31 bits mais à esquerda dessa palavra, junto de um bit proveniente de $a[5]$, são armazenados em um registrador de resultado de 32 bits. Esse registrador é lido na terceira chamada (`IORD`), e então armazenado em $c[0]$. Essa seqüência se sucede até a última chamada, onde $c[5]$ recebe os dois bits de $a[5]$, junto de mais um bit proveniente de $a[4]$, completando assim a operação de quadrado. Nota-se que o quadrado exige o uso de 35 bits de armazenamento interno quando implementado como periférico.

```
IOWR( CP_SQUARE_BASE, 5, a[5] );
IOWR( CP_SQUARE_BASE, 0, a[0] );
c[0] = IORD( CP_SQUARE_BASE, 0 );
IOWR( CP_SQUARE_BASE, 1, a[1] );
c[1] = IORD( CP_SQUARE_BASE, 1 );
IOWR( CP_SQUARE_BASE, 2, a[2] );
c[2] = IORD( CP_SQUARE_BASE, 2 );
IOWR( CP_SQUARE_BASE, 3, a[3] );
c[3] = IORD(CP_SQUARE_BASE, 3 );
IOWR( CP_SQUARE_BASE, 4, a[4] );
c[4] = IORD( CP_SQUARE_BASE, 4 );
c[5] = IORD( CP_SQUARE_BASE, 5 );
```

Figura 5.2: Chamadas ao Periférico de Quadrado

5.1.3 Raiz Quadrada

A raiz quadrada é um operação muito semelhante ao quadrado pois se trata de uma rotação do argumento de 163 bits, só que dessa vez, à esquerda. Essa operação também mantém a característica de ser divisível quando implementada como periférico. Novamente reutilizamos o núcleo do módulo de *hardware* apresentado na Figura 4.5. Para isso, reimplementando sua interface de comunicação com o barramento Avalon e também incluímos registradores internos para armazenamento temporário de dados entre as chamadas.

A Figura 5.3 mostra a estratégia de chamadas ao periférico de raiz quadrada. Identicamente a todas as outras operações divisíveis, exceto a adição, deve-se utilizar em toda chamada, o endereço do periférico, mais um identificador de qual bloco de 32 bits está sendo operado.

Na primeira chamada envia-se $a[0]$ para o módulo, sendo seus 31 bits mais à direita armazenados em um registrador de dados temporários. Já o seu bit mais à esquerda é armazenado para ser utilizado na última chamada. Na segunda chamada envia-se $a[1]$ para o módulo, sendo que seu bit mais à esquerda é concatenado com os 30 bits provenientes de $a[0]$ e assim armazenados no registrador de resultados. Os 30 bits restantes de $a[1]$ são armazenados no registrador de dados temporários. Na terceira chamada (`IORD`) lê-se o valor do registrador de resultado, sendo esse armazenado em $c[0]$. Essa seqüência de chamadas se repete até que se tenha operado sobre os 163 bits. Nota-se que, embora a raiz quadrada seja similar ao quadrado por se tratarem de rotações, a raiz quadrada utiliza registradores internos maiores. Mais precisamente, essa operação requer que a capacidade de armazenamento interno no periférico seja de 64 bits.

```
IOWR( CP_SQUARE_ROOT_BASE, 0, a[0] );
IOWR( CP_SQUARE_ROOT_BASE, 1, a[1] );
c[0] = IORD( CP_SQUARE_ROOT_BASE, 0 );
IOWR( CP_SQUARE_ROOT_BASE, 2, a[2] );
c[1] = IORD( CP_SQUARE_ROOT_BASE, 1 );
IOWR( CP_SQUARE_ROOT_BASE, 3, a[3] );
c[2] = IORD( CP_SQUARE_ROOT_BASE, 2 );
IOWR( CP_SQUARE_ROOT_BASE, 4, a[4] );
c[3] = IORD( CP_SQUARE_ROOT_BASE, 3 );
IOWR( CP_SQUARE_ROOT_BASE, 5, a[5] );
c[4] = IORD( CP_SQUARE_ROOT_BASE, 4 );
c[5] = IORD( CP_SQUARE_ROOT_BASE, 5 );
```

Figura 5.3: Chamadas ao Periférico de Raiz Quadrada

5.1.4 *Trace*

A operação *trace* também recai no grupo das operações divisíveis, e assim como as operações anteriores, também tivemos de realizar pequenas modificações na interface do módulo de *hardware* (Figura 4.7) para utilizá-lo como periférico. Como a nova interface possui apenas 32 bits de entrada de dados, tivemos de adotar uma nova estratégia de chamadas ao periférico, sendo essa mostrada na Figura 5.4.

Assim como as operações anteriores, utiliza-se em toda chamada o endereço do periférico, e informa-se o módulo de *hardware* em qual dos blocos de 32 bits, do argumento de 163 bits, se está operando. Na primeira chamada, envia-se $a[0]$ para o periférico, que por sua vez computa o *trace* desses 32 bits, armazenando o resultado de um bit em um registrador interno ao módulo. Na segunda chamada, calcula-se o *trace* do bit de resultado da primeira chamada, com os 32 bits de $a[1]$, resultando em mais um bit de resultado, o qual sobrescreve o valor armazenado anteriormente. Essa seqüência de chamadas é realizada até o envio de $a[5]$ ao periférico. Em seguida, realiza-se uma operação de leitura (`IORD`) para ler o bit de resultado, sendo esse armazenado em c.

```
IOWR( CP_TRACE_BASE, 0, a[0] );
IOWR( CP_TRACE_BASE, 1, a[1] );
IOWR( CP_TRACE_BASE, 2, a[2] );
IOWR( CP_TRACE_BASE, 3, a[3] );
IOWR( CP_TRACE_BASE, 4, a[4] );
IOWR( CP_TRACE_BASE, 5, a[5] );
c = IORD( CP_TRACE_BASE, 0 );
```

Figura 5.4: Chamadas ao Periférico do *Trace*

5.1.5 Solução de Equação Quadrática

A solução de equação quadrática é outra operação divisível considerada nesse trabalho. Seu módulo de *hardware*, mostrado na Figura 4.9 pode também ser utilizado como periférico do NIOS2. Sendo assim, pode-se enviar ao módulo 32 bits de dados para serem processados por iteração, uma vez que a interface do periférico possui uma única entrada de dados de 32 bits.

Observa-se (Figura 5.5) que na primeira chamada envia-se $a[0]$ ao periférico, que por sua vez realiza o cômputo da solução parcial do resultado, que é então armazenado em um registrador interno de 32 bits. Em seguida, executa-se uma chamada de leitura, sendo o resultado armazenado em $c[0]$. Na terceira iteração, escreve-se $a[1]$ no periférico, e, aproveitando o bit de resultado do cômputo anterior, calcula-se mais 32 bits do resultado, os

quais são novamente armazenados em um registrador interno. Essa seqüência de escritas e leituras ocorre até se obter o valor de $c[5]$. Nota-se assim, que essa operação necessita de um registrador interno de 32 bits para que a operação possa ser partida e executada em blocos de 32 bits.

```
IOWR( CP_SOLVE_QE_BASE, 0, a[0] );
c[0] = IORD( CP_SOLVE_QE_BASE, 0 );
IOWR( CP_SOLVE_QE_BASE, 1, a[1] );
c[1] = IORD( CP_SOLVE_QE_BASE, 1 );
IOWR( CP_SOLVE_QE_BASE, 2, a[2] );
c[2] = IORD( CP_SOLVE_QE_BASE, 2 );
IOWR( CP_SOLVE_QE_BASE, 3, a[3] );
c[3] = IORD( CP_SOLVE_QE_BASE, 3 );
IOWR( CP_SOLVE_QE_BASE, 4, a[4] );
c[4] = IORD( CP_SOLVE_QE_BASE, 4 );
IOWR( CP_SOLVE_QE_BASE, 5, a[5] );
c[5] = IORD( CP_SOLVE_QE_BASE, 5 );
```

Figura 5.5: Chamadas ao Periférico de Solução de Equação Quadrática

5.1.6 Multiplicação

Assim como sua implementação como instrução especializada, a multiplicação em corpos finitos não pode ser partida em blocos de 32 bits. Na verdade, utilizamos os mesmos módulos multiplicadores das instruções especializadas, sendo que agora suas interfaces de comunicação utilizam o padrão exigido para periféricos. Os diagramas em blocos dos multiplicadores NIST, López e NIST modificado, são mostrados respectivamente nas Figuras 4.12, 4.14, e 4.15. Experimentamos ainda em nossas implementações com paralelismo e desenrolamento de laços desses algoritmos de multiplicação.

Antes de iniciar a execução da multiplicação, deve-se enviar todos os 163 bits dos argumentos para o periférico, sendo assim necessários três registradores internos de 163 bits (dois para argumentos e um para resultado). Observa-se na Figura 5.6 que as doze primeiras chamadas (`IOWR`) escrevem os argumentos no periférico, sendo que na última escrita, o periférico inicia o cálculo da multiplicação, bloqueando a chamada da macro. Quando a operação é finalizada, mais seis chamadas (`IORD`) são necessárias para se ler o resultado da multiplicação.

```
// Escrita dos argumentos
IOWR( CP_MULTIPLICATION_BASE, 0, a[0] );
...
IOWR( CP_MULTIPLICATION_BASE, 5, a[5] );
IOWR( CP_MULTIPLICATION_BASE, 6, b[0] );
...
IOWR( CP_MULTIPLICATION_BASE, 11, b[5] );
// Leitura do resultado
c[0] = IORD( CP_MULTIPLICATION_BASE, 0 );
...
c[5] = IORD( CP_MULTIPLICATION_BASE, 5 );
```

Figura 5.6: Chamadas ao Periférico de Multiplicação

5.1.7 Inversão

No Capítulo 4 mencionamos o problema de se utilizar freqüências de operações abaixo de 120MHz para o processador NIOS2, quando inserimos os módulos inversores como instruções especializadas. Na abordagem de periféricos temos *clocks* independentes para o processador e para o periférico, permitindo utilizar freqüências mais baixas para os módulos inversores, e assim, executar nossos experimentos com o NIOS2 executando a 120MHz.

Assim como a multiplicação, a inversão é uma operação não-divisível. Dessa maneira, necessita-se de seis chamadas ao periférico (`IOWR`) para se escrever o argumento de 163 bits, como mostrado na Figura 5.7. A operação de inversão inicia na sexta escrita de dados, tomando múltiplos ciclos de *clock*, e mantendo a chamada bloqueada durante sua execução. Quando a operação termina, mais seis operações de leitura (`IORD`) são necessárias para se obter o resultado.

```
// Escrita do argumento
IOWR( CP_INVERSION_BASE, 0, a[0] );
...
IOWR( CP_INVERSION_BASE, 5, a[5] );
// Leitura do resultado
c[0] = IORD( CP_INVERSION_BASE, 0 );
...
c[5] = IORD( CP_INVERSION_BASE, 5 );
```

Figura 5.7: Chamadas ao Periférico de Inversão

Devido ao grande número de multiplicadores em corpos finitos que implementamos, tivemos a possibilidade de experimentar com várias configurações para os módulos inver-

sores. Mais precisamente, utilizamos multiplicadores baseados no algoritmo NIST, López e NIST modificado, em suas versões básicas e também explorando desenrolamento de laços/paralelismo.

5.1.8 Multiplicação de Ponto (kP)

Assim como mostrado no Capítulo 4, utilizamos para a implementação na versão periféricos, o algoritmo de López-Dahab [10] (Algoritmo 7).

Uma maneira diferente de apresentar esse algoritmo, em termos das funções `Madd`, `Mdouble`, e `Mxy` é mostrada nos Algoritmos 8, 9, 10, 11, apresentados no Capítulo 2. Diferentemente do Algoritmo 7, o Algoritmo 8 necessita de apenas uma inversão para calcular kP. De agora em diante, discutiremos as implementações em *hardware* tomando como base somente os Algoritmos 8, 9, 10, e 11.

Observando o Algoritmo 8, pode-se notar que a Linha 4 é responsável pela maior parte do cômputo de kP, onde as funções `Madd` e `Mdouble` podem ser executadas até $t-1$ vezes, sendo $t = \lfloor \log k \rfloor$. Em outras palavras, essa é a parte fundamental de kP que pode ser movida para *hardware* de forma a acelerar seu processamento, e ainda ter menor área do que se todo o algoritmo fosse implementado em *hardware*. Nessa versão *hardware/software*, nomeamos esse módulo de "Núcleo de kP". De forma a ter uma maior abrangência nos resultados da implementação de kP em *hardware*, implementamos todo o algoritmo em *hardware*, sendo essa implementação nomeada "kP Completo".

Devido à independência de dados nas funções `Madd` e `Mdouble`, algumas de suas operações podem ser executadas em paralelo, entre elas a multiplicação. Dessa maneira, visando aumentar o desempenho dos módulos "Núcleo de kP" e "kP Completo", realizamos as implementações de cada um desses módulos utilizando 1, 2, e 3 multiplicadores em corpos finitos, que são usados pelos módulos de kP de forma paralela.

De agora em diante nessa seção, adotamos uma nomenclatura padrão onde é mostrado, em frente do nome do módulo de kP, o número de multiplicadores em corpos finitos utilizados em sua implementação. Como exemplo, "3 NIST" indica que foram utilizados três multiplicadores em corpos finitos implementados segundo o método proposto pelo NIST.

Além disso, de forma complementar, implementamos os módulos "Núcleo de kP" e "kP Completo" utilizando paralelismo e desenrolamento de laços internos aos multiplicadores em corpos finitos, obtendo assim, módulos ainda mais rápidos. A notação utilizada neste caso é a mesma das seções anteriores, ou seja, o nome "2 Lopez 27u" indica que foram utilizados dois multiplicadores em corpos finitos a partir do algoritmo de López, os quais executam 27 iterações do algoritmo por ciclo de *clock*.

Núcleo de kP

Como mostrado nas Linhas 1, 2 e 3 do Algoritmo 8, a multiplicação de ponto possui uma pequena parte de inicialização de variáveis. Sendo esses passos compostos apenas por operações elementares, não requerendo área significativa, incluímos essas inicializações na implementação em *hardware*.

Dessa maneira, o módulo "Núcleo de kP" deve receber o inteiro k, o parâmetro b da curva, e as coordenadas x e y do ponto P. Dado que cada elemento em $\mathbb{F}_{2^{163}}$ necessita de seis palavras de 32 bits, são necessárias ao todo 24 chamadas de escrita no periférico, sendo essas mostradas na Figura 5.8.

```
// Escrita de k
IOWR(CP_KP_LD_CORE_BASE, 0, k[0]);
...
IOWR(CP_KP_LD_CORE_BASE, 5, k[5]);
// Escrita de b
IOWR(CP_KP_LD_CORE_BASE, 6, b[0]);
...
IOWR(CP_KP_LD_CORE_BASE, 11, b[5]);
// Escrita de x
IOWR(CP_KP_LD_CORE_BASE, 12, x[0]);
...
IOWR(CP_KP_LD_CORE_BASE, 17, x[5]);
// Escrita de y
IOWR(CP_KP_LD_CORE_BASE, 18, y[0]);
...
IOWR(CP_KP_LD_CORE_BASE, 23, y[5]);
```

Figura 5.8: Chamadas de Escrita ao Periférico "Núcleo de kP"

A última chamada ao periférico automaticamente dispara a execução do cômputo do núcleo de kP. Em seguida, executam-se mais 24 chamadas de leitura do periférico, como mostrado na Figura 5.9, de forma a obter as coordenadas projetivas X_1/Z_1 e X_2/Z_2, representadas por `x1`, `z1`, `x2`, e `z2`. Por fim, a conversão de coordenadas projetivas para coordenadas afins é realizada em *software* através da função `Mxy`.

```
// Leitura de x1
x1[0] = IORD(CP_KP_LD_CORE_BASE, 0);
...
x1[5] = IORD(CP_KP_LD_CORE_BASE, 5);
// Leitura de z1
z1[0] = IORD(CP_KP_LD_CORE_BASE, 6);
...
z1[5] = IORD(CP_KP_LD_CORE_BASE, 11);
// Leitura de x2
x2[0] = IORD(CP_KP_LD_CORE_BASE, 12);
...
x2[5] = IORD(CP_KP_LD_CORE_BASE, 17);
// Leitura de z2
z2[0] = IORD(CP_KP_LD_CORE_BASE, 18);
..
z2[5] = IORD(CP_KP_LD_CORE_BASE, 23);
```

Figura 5.9: Chamadas de Leitura ao Periférico "Núcleo de kP"

kP Completo

Da mesma maneira que o Núcleo de kP, o módulo kP Completo recebe o inteiro k, o parâmetro b da curva, e as coordenadas x e y do ponto P. São assim necessárias 24 chamadas ao periférico, que são realizadas de maneira idêntica ao módulo "Núcleo de kP" (Figura 5.8). Como a função de conversão de coordenadas projetivas para afins (`Mxy`) faz parte do módulo kP Completo, esse retorna o valor das coordenadas x e y referente ao resultado da multiplicação de ponto. Sendo assim, como mostrado na Figura 5.10, obtêm-se o resultado através de 12 leituras do periférico.

```
// Leitura de x
x[0] = IORD(CP_KP_LD_FULL_BASE, 0);
...
x[5] = IORD(CP_KP_LD_FULL_BASE, 5);
// Leitura de y
y[0] = IORD(CP_KP_LD_FULL_BASE, 6);
...
y[5] = IORD(CP_KP_LD_FULL_BASE, 11);
```

Figura 5.10: Chamadas de Leitura ao Periférico "kP Completo"

5.2 Análise dos Módulos *Stand-Alone*

Avaliamos nessa seção os módulos *stand-alone* em termos de desempenho e área, quando os mesmos são implementados pensando em seus usos posteriores como periféricos. Consideramos aqui somente os módulos multiplicadores e inversores em corpos finitos, dado que seria difícil comparar os desempenhos dos módulos das operações divisíveis, uma vez que essas operam em blocos de 32 bits e não nos 163 bits dos argumentos. Dessa forma, postergamos essas análises para a Seção 5.3, ou seja, quando são usadas em um sistema real na forma de instruções especializadas.

Assim como fizemos no caso das instruções especializadas na avaliação de desempenho dos módulos *stand-alone*, não consideramos os custos de passagem de parâmetros e de leitura/escrita de dados dos periféricos. Tal abordagem será tratada na Seção 5.3, onde todos os *overheads* são considerados em execuções de aplicações reais. Fizemos isso propositalmente para poder comparar os métodos de medida de desempenho (*stand-alone* e *end-to-end*), e posteriormente mostrar que é necessário ter cuidado com as medidas de desempenho, pois elas podem não se concretizar em aplicações reais.

5.2.1 Multiplicação

As áreas de implementação e os desempenhos de cada multiplicador são mostrados na Tabela 5.1. Sem explorar paralelismo nem desenrolar laços dos algoritmos, nota-se que o método NIST Modif. possui o pior desempenho que os multiplicadores López e NIST. Esses dois últimos executam uma multiplicação, respectivamente, em 650,88ns e 603,91ns. O método López resulta ainda na menor área (862 ALUTs), e na maior freqüência de operação (269,91MHz).

Permitindo o uso de mais área, experimentamos com paralelismo e com desenrolamento de laços dos algoritmos para implementar os multiplicadores. Nesse caso, como observado na Tabela 5.1, a situação favorece o método NIST Modif. 27p, que passa a ocupar a menor área de implementação (7213 ALUTs) quando comparado com o NIST 27p e com o López 27u. Esse método, que calcula 27 bits do resultado por ciclo de *clock*, também é o mais rápido no cálculo da multiplicação e possui a maior freqüência de operação, realizando uma operação em 57,05ns quando operando a 122,70MHz. Como resultado, tem-se o módulo NIST Modif. 27p como nosso multiplicador mais eficiente.

Considerando a área de implementação, nota-se que o multiplicador López 27u é 9 vezes maior que sua versão básica, e executa pouco mais de 10 vezes mais rápido que aquela. Com essa mesma análise, observa-se que o multiplicador NIST Modif. 27p oferece melhores ganhos de desempenho por área, dado que ele é aproximadamente 4,5 vezes maior que sua versão básica, porém executa mais de 11 vezes mais rápido.

Comparando-se os multiplicadores implementados com as referências [18] e [40], concluímos que nossas implementações são mais eficientes em termos de desempenho. Infelizmente, não foi possível comparar com as áreas desses trabalhos relacionados.

Método de Multiplicação	Área (ALUTs)	Desempenho (ns)	Freqüência (MHz)
NIST	1117	605,88	269,03
NIST 6p	3744	216,67	129,23
NIST 9p	4476	153,10	124,10
NIST 18p	7677	91,06	109,82
NIST 27p	9533	68,75	101,82
López	862	603,91	269,91
López 6u	2942	137,65	203,42
López 9u	3542	113,53	167,36
López 18u	5698	72,41	138,10
López 27u	7935	57,18	122,43
NIST Modif.	1582	636,84	255,95
NIST Modif. 6p	2661	192,14	145,73
NIST Modif. 9p	3682	136,94	138,75
NIST Modif. 18p	5221	79,57	125,68
NIST Modif. 27p	7213	57,05	122,70
Gura et al. [18] †	-	900	66,4
Trujillo et al. [40] ‡	-	673	82,43

† Usando bases polinomiais e uma FPGA Xilinx Virtex XCV2000E-FG680-7

‡ Usando GNB e uma FPGA Altera EP2A70B724C7

Tabela 5.1: Periféricos: Módulos *Stand-Alone* dos Multiplicadores

5.2.2 Inversão

Similarmente aos multiplicadores, iniciamos a análise dos inversores por suas versões básicas, isto é, sem explorar paralelismo nem desenrolar laços dos algoritmos. Observando-se a Tabela 5.2, nota-se que o inversor baseado na multiplicação López tem o pior desempenho (9,31μs) e a maior área de implementação (2531 ALUTs). Já o inversor usando o método NIST provê simultaneamente, entre as versões básicas dos inversores, a menor área de implementação (2314 ALUTs), o melhor desempenho (8,92μs), e a melhor freqüência de operação (165,59 MHz) .

Utilizando paralelismo, temos os módulos NIST 27p e NIST Modif. 27p, ambos calculando 27 bits do resultado por ciclo de *clock*. Comparando esses dois módulos, nota-se

que o NIST Modif. 27p é a opção mais vantajosa em termos de área (8626 ALUTs), desempenho (0,63μs) e freqüência de operação (116,16MHz).

Já o López 18u, utilizando a uma freqüência de operação de 109,61MHz, computa uma inversão em 0,67μs. Esse resultado é bastante semelhante ao do NIST Modif. 27p, porém a área de implementação do López 18u é consideravelmente maior (10652 ALUTs) que a ocupada pelo NIST Modif. 27p (8626 ALUTs).

Método de Multiplicação	Área (ALUTs)	Desempenho (μs)	Freqüência (MHz)
NIST	2314	8,92	165,59
NIST 9p	5909	1,52	119,16
NIST 27p	11468	0,76	95,82
López	2531	9,31	158,60
López 9u	6341	1,35	133,76
López 27u	10652	0,67	109,61
NIST Modif.	2248	9,25	159,69
NIST Modif. 9p	4555	1,44	125,42
NIST Modif. 27p	8626	0,63	116,16
Gura et al. [18] †	-	14,13	66,4
Trujillo et al. [40] ‡	-	10	82,43

† Usando bases polinomiais e uma FPGA Xilinx Virtex XCV2000E-FG680-7

‡ Usando GNB e uma FPGA Altera EP2A70B724C7

Tabela 5.2: Periféricos: Módulos *Stand-Alone* dos Inversores

Considerando os ganhos de desempenho por área, observa-se que o inversor López 27u é mais de 4 vezes maior que sua versão básica, e executa quase 14 vezes mais rápido que aquela. Da mesma maneira, nota-se que o inversor NIST Modif. 27p é quase 4 vezes maior que sua versão básica, porém executa mais de 14 vezes mais rápido. Dessa maneira, o inversor NIST Modif. 27p é o nosso inversor mais rápido, e o que oferece melhores ganhos de desempenho por área.

Nossos inversores são mais eficientes em termos de desempenho do que as implementações em [18] e [40]. Novamente, não foi possível comparar com as áreas desses trabalhos.

5.2.3 Multiplicação de Pontos (kP)

Como os módulos *stand-alone* do "Núcleo de kP" não realizam a operação de multiplicação de pontos por completo, não os abordaremos aqui, porém na seção 5.3 quando são utilizados como periféricos do NIOS2. Assim, discutimos nessa seção somente a implementação dos módulos *stand-alone* de "kP Completo". Para isso tomaremos como referência a Tabela 5.3.

kP Completo usando	Área (ALUTs)	Desempenho (μs)	Freqüência (MHz)
1 NIST	8197	1991,12	75,95
1 NIST 27p	16391	114,28	59,53
1 López	8206	1885,80	85,83
1 López 27u	15931	115,27	63,00
1 NIST Modif.	7813	1937,95	78,19
1 NIST Modif. 27p	13947	107,18	83,10
2 NIST	10198	1010,28	80,67
2 NIST 27p	27252	64,02	67,31
2 López	11982	1052,01	77,47
2 López 27u	25750	60,75	60,05
2 NIST Modif.	9576	1056,10	77,17
2 NIST Modif. 27p	22678	54,63	78,88
3 NIST	11200	725,39	75,95
3 NIST 27p	33357	50,61	59,53
3 López	11923	710,51	77,54
3 López 27u	29234	39,12	68,40
3 NIST Modif.	10142	704,60	78,19
3 NIST Modif. 27p	28044	36,30	83,01
Shu et al. [38] †	-	48	68,90
Ansari et al. [43] ‡	-	41,67	100,00

† Usando bases polinomiais e uma FPGA Xilinx Virtex XC2000E

‡ Usando bases polinomiais e uma FPGA Xilinx Virtex II XC2V2000

Tabela 5.3: Periféricos: Módulos *Stand-Alone* de "kP Completo"

Observemos inicialmente o módulo 1 NIST Modif., que é o que possui a menor área de implementação (7813 ALUTs), e o segundo pior tempo de cálculo de kP (1937,95μs). Quando utilizamos o mesmo multiplicador finito, mas com nível 27 de paralelismo (1 NIST Modif. 27p), obtivemos um multiplicador de ponto ocupando somente 1,79 vezes mais área (13947 ALUTs) que sua versão básica, mas 18 vezes mais veloz.

Dentre os multiplicadores de ponto que utilizam apenas um multiplicador finito, 1 NIST Modif. 27p é o módulo mais rápido, uma vez que calcula kP em 107,18μs.

Analisado as implementações com dois multiplicadores finitos, tanto em suas versões básicas quanto paralelizadas/desenroladas, temos na grande maioria dos casos o método NIST Modif., oferecendo simultaneamente a menor área e um desempenho satisfatório.

De forma a explorar ao máximo a possibilidade de paralelismo do algoritmo de López-Dahab, implementamos módulos voltados a desempenho, os quais utilizam três multiplicadores finitos. Dentre as versões que fazem uso de multiplicadores finitos sem paralelismo e desenrolamento de laços, observa-se novamente que o 3 NIST Modif. possui simultaneamente menor área de implementação (10142 ALUTs) e melhor desempenho (704,60μs).

Por fim, analisamos as implementações mais eficientes que usam três multiplicadores finitos, que por sua vez exploram internamente paralelismo/desenrolamento de laços. Nesse caso temos o 3 NIST Modif. 27p como nosso multiplicador de ponto mais eficiente em termos de área (28044 ALUTs) e desempenho, sendo que kP pode ser calculado em apenas 36,30μs.

Se compararmos os módulos 1 NIST Modif. 27p com o 3 NIST Modif. 27p, temos que o segundo ocupa aproximadamente o dobro da área, mas é quase 3 vezes mais eficiente que o primeiro. Os resultados são mais surpreendentes se compararmos a versão mais elaborada (3 NIST Modif. 27p) com a mais simples (1 NIST Modif.). Nesse caso, o módulo 3 NIST Modif. 27p, ocupando 3,6 mais área que o 1 NIST Modif., realiza o cômputo de kP aproximadamente 53 vezes mais rápido. Através da Tabela 5.3, concluise que no caso dos módulos *stand-alone*, o método NIST Modif. é mais apropriado para a multiplicação de ponto, tanto em área quanto em desempenho, do que o original NIST e o de López.

Ressaltamos por fim, que os trabalhos [38] e [43], também baseados no algoritmo de López-Dahab, são reportados (até o momento) na literatura como as implementações mais eficientes em tempo, pois calculam kP, em 48μs e 41,67μs, respectivamente. Entretanto, nosso trabalho apresenta o 3 NIST Modif. 27p como uma implementação mais rápida para o cálculo da multiplicação de ponto.

Como conseqüência, respondemos à questão em aberto apresentada na Introdução, que era a de não se saber se bases normais Gaussianas poderiam ser mais eficientes que as bases polinomiais para implementações em *hardware*. Como ambos os trabalhos [38] e [43] utilizam bases polinomiais, e corpos de tamanho $\mathbb{F}_{2^{163}}$, a resposta para tal pergunta é: Sim. Mostramos que nesse caso **GNBs são mais rápidas em** *hardware* **que bases polinomiais**.

5.3 Análise dos Sistemas com Periféricos

Nessa seção consideramos as operações em corpos finitos sobre $\mathbb{F}_{2^{163}}$ e a multiplicação de pontos implementadas como periféricos de um processador NIOS2. Novamente temos, devido ao aumento da área do sistema (NIOS2 mais periférico), uma menor freqüência de operação quando comparando com os módulos *stand-alone*. Para manter a consistência com as operações em corpos finitos implementadas como instruções especializadas, fixamos em 120MHz a freqüência de operação do processador quando utilizando periféricos.

Como veremos adiante, os módulos multiplicadores de pontos ocupam grandes áreas de implementação, que diretamente influenciam na freqüência de operação de todo o sistema devido à complexidade do roteamento interno à FPGA. Por outro lado, queremos determinar qual o multiplicador de pontos mais eficiente em termos de desempenho e área, independente da freqüência de operação do sistema. De qualquer forma, o que importa é o tempo de cômputo de kP, e assim, adotamos uma aproximação menos rigorosa (somente) para os multiplicadores de pontos, na qual permitimos o sistema operar em freqüências abaixo de 120MHz.

Adotamos a implementação de apenas um periférico por processador, para podermos determinar precisamente as características para cada uma das operações aritméticas, como por exemplo *speedup* e área do sistema. Utilizamos a plataforma de trabalho para a avaliação dos periféricos de maneira idêntica ao caso das instruções especializadas. Mais precisamente, para determinar os *speedups*, medimos os desempenhos das implementações *software* e *hardware* executando programas escritos na linguagem de programação C. Utilizamos, novamente, programas de teste altamente otimizados e compilados com o compilador GCC (versão 3.4.1) portado para o processador NIOS2. Em todos os testes, adotamos o nível de otimização `-O2`, que produziu, na maioria dos casos, os programas mais rápidos.

Em nossas medições de desempenho utilizamos uma abordagem *end-to-end*, não interessando se uma determinada operação sob teste foi implementada em *software* ou em *hardware*. Isso significa que todos os *overheads* são considerados: carga dos operandos da memória, comunicação com o módulo através de várias chamadas aos periféricos, escrita dos resultados na memória, etc. Como resultado, podemos comparar, de forma mais justa, implementações *software* e *hardware*, e assim, ter uma idéia mais real de quais serão os *speedups* de tais implementações quando utilizadas em um criptosistema real.

No caso de periféricos, utilizamos abordagens *hardware/software* apenas para a multiplicação de pontos, onde o núcleo de kP é realizado em *hardware*, e finalizado em *software* com a execução da função `Mxy`. Salientamos que kP também foi implementado inteiramente em *hardware*, permitindo comparar seu desempenho nas implementações *hardware/software*, e assim, determinar, qual a opção mais apropriada para sistemas que exigem desempenho e para sistemas com restrições de área.

5.3.1 Operações Elementares

Os desempenhos das operações elementares em *software* e *hardware* são mostrados na Tabela 5.4. Observa-se que não obtivemos nenhum *speedup* quando implementamos as operações de adição, quadrado, raiz quadrada, e *trace* em *hardware*. Ou seja, não é vantajoso implementar essas operações como periféricos.

Operação	*Software* (μs)	*Hardware* (μs)	*Speedup*
Adição	0,44	0,94	0,47
Quadrado	0,59	0,68	0,88
Raiz Quadrada	0,52	0,68	0,76
Trace	0,43	0,43	1,00
Solução de E.Q.	2,53	0,68	3,70
Função β	27,25	0,69	39,40

Tabela 5.4: Periféricos: Desempenho e *Speedups* das Operações Elementares

Operação	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)
Adição	3553	1,32
Quadrado	3349	2,62
Raiz Quadrada	3786	2,00
Trace	3432	2,91
Solução de E.Q.	3582	10,32
Função β	4018	98,05

Tabela 5.5: Periféricos: Áreas de Sistema e *Speedups*/Área das Operações Elementares

Nota-se que o módulo que calcula a solução de equação quadrática resulta em um bom *speedup*, executando 3,68 vezes mais rápido que sua versão em *software*. Essa operação opera bit a bit, e para implementá-la em *software* deve-se fazer uso das operações de deslocamento, *ands*, *ors*, e de tabelas. Dessa maneira, tem-se dificuldade de se utilizar o conjunto de instruções original do NIOS2 para se obter desempenho satisfatório em *software*.

Dentre as operações elementares, a função β produz o melhor *speedup* (39,40), fato explicado pela maneira eficiente com que as permutações podem ser implementadas em *hardware*, consistindo basicamente de um conjunto de fios e uma função *xor*. Sua implementação em *software* é relativamente lenta, uma vez que é restrita a uma enorme seqüência de deslocamentos, *and*s, *or*s e *xor*s.

Pensando em ambientes com restrição de área, tomamos novamente a razão *speedup*/área como um fator de decisão para o que deve ser mantido em *software* e o que deve ser movido para *hardware*. As áreas de implementação dos sistemas composto pelo NIOS2 mais periférico, bem como as razões *speedup*/área do sistema, são mostrados na Tabela 5.5.

Como as operações de adição, quadrado, raiz quadrada e *trace* oferecem vantagens de implementação como periféricos, não discutimos suas áreas de implementação nessa seção. Embora a operação de solução de equação quadrática ofereça um alto *speedup*, ela possui uma razoável razão *speedup*/área de sistema ($10{,}32\times10^{-4}$).

Analisando a função β notamos que, embora essa ocupe uma área maior que as outras operações elementares, ela provê uma boa razão ($98{,}05\times10^{-4}$), resultante de seu bom *speedup*.

5.3.2 Multiplicação

Dentre todas as operações implementadas como periféricos, apenas a função *beta* resultou em um bom *speedup*. Felizmente, essas operações mais simples não são tão importantes para o cálculo de kP quanto a multiplicação em corpos finitos.

Assim como instruções especializadas, implementamos multiplicações como periféricos usando os métodos NIST, López e NIST modificado. Devido à importância da multiplicação em corpos finitos para o cálculo de kP, utilizamos paralelismo e desenrolamento de laços dos algoritmos de forma a obter maior desempenho. Diferentemente das instruções especializadas, os periféricos podem operar abaixo de 120MHz dado que seus *clocks* são independentes do *clock* do processador.

De forma a analisar os *speedups* de nossas implementações, compararemos nossos multiplicadores em *hardware* com sua respectiva implementação em *software*. Consideramos novamente em nossas análises o *speedup* normalizado tomando como base o método mais rápido para implementações em *software* que é o DHHLLM. Assim, temos a indicação de qual implementação em *hardware* provê o melhor *speedup* em face da melhor multiplicação em *software*, garantindo assim que esses *speedups* não são provenientes de um algoritmo com desempenho deficiente em *software*, mas sim de uma boa implementação em *hardware*.

Como mostrado na Tabela 5.6, o método DHHLLM realiza uma multiplicação em 156,33μs quando executando em um processador NIOS2 a 120MHz.

Analisamos inicialmente uma abordagem *hardware/software* de forma a economizar área de implementação. Mais precisamente, fizemos essa análise para os métodos López e NIST modificado, dado que ambos utilizam a função β. Assim, apenas a função β é executada em *hardware*, sendo o restante do algoritmo executado em *software*. Como resultado, os métodos López e NIST Modif. realizaram uma multiplicação em, respectivamente, 200,04μs e 232,88μs, resultando em *speedups* acima de 10. Entretanto, esses tempos de execução quando comparados com DHHLLM, podem ser traduzidos para *speedups* normalizados de, respectivamente, 0,78 e 0,67. Ou seja, ao invés de se gastar área para implementar a função β em *hardware*, é mais vantajoso utilizar o conjunto de instruções originais do NIOS2 e o método de DHHLLM em *software*.

Multiplicador	*Software* (μs)	*Hardware* (μs)	Freq. Periférico (MHz)	*Speedup*	*Speedup* Normalizado
DHHLLM	156,33	-	-	-	-
López *	2309,77	200,04	120	11,55	0,78
NIST Modif. *	4233,08	232,88	120	18,18	0,67
NIST	7222,03	2,92	150	2476,13	53,60
NIST 6p	7222,03	1,18	120	6103,13	132,11
NIST 9p	7222,03	1,11	120	6516,12	141,05
NIST 18p	7222,03	2,08	100	3466,58	75,04
NIST 27p	7222,03	2,28	75	3162,93	68,46
López	2309,77	2,90	150	796,47	53,91
López 6u	2309,77	1,18	120	1951,92	132,11
López 9u	2309,77	1,11	120	2084,00	141,05
López 18u	2309,77	2,06	100	1122,15	75,95
López 27u	2309,77	2,03	100	1140,63	77,20
NIST Modif.	4233,08	2,91	150	1455,50	53,75
NIST Modif. 6p	4233,08	1,18	120	3577,25	132,11
NIST Modif. 9p	4233,08	1,11	120	3819,32	141,05
NIST Modif. 18p	4233,08	2,07	100	2048,26	75,64
NIST Modif. 27p	4233,08	2,02	100	2099,05	77,52

* Multiplicador implementado utilizando somente a função β em *hardware*

Tabela 5.6: Periféricos: Desempenho e *Speedups* dos Multiplicadores

Como vimos até o momento, a função β, quando utilizada em abordagens *hardware/software* (em diversos níveis), não apresentou ganhos de desempenho que fossem maiores que uma implementação em *software* utilizando o método DHHLLM. Sendo assim, de agora em diante, não examinaremos mais o uso dessa função nas implementações das multiplicações, inversões e multiplicações de pontos.

Partindo para a implementação dos multiplicadores inteiramente em *hardware*, observamos que os *speedups* aumentam significativamente, sendo todos eles superiores a 790. O maior *speedup* (6516,12) é obtido pelo multiplicador NIST 9p devido ao baixo desempenho de seu algoritmo em *software*.

Observa-se pela Tabela 5.6 que os multiplicadores, para um mesmo nível de paralelismo ou desenrolamento de laços, possuem desempenhos bastante semelhantes. Os sistemas com os periféricos NIST 9p, López 9u, e NIST Modif. 9p são os mais rápidos, sendo que computam uma multiplicação em apenas 1,11μs, sendo assim, mais de 140 vezes mais rápidos que o DHHLLM. Entretanto, como López 9u ocupa menor área, concluímos que esse multiplicador é uma melhor alternativa para sistemas voltados a desempenho.

Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
López *	4018	28,74	1,94
NIST Modif. *	4018	45,24	1,67
NIST	4546	5446,82	117,90
NIST 6p	6973	8752,51	189,45
NIST 9p	8469	7694,08	166,54
NIST 18p	10585	3274,99	70,89
NIST 27p	12347	2561,70	55,45
López	4297	1853,55	125,45
López 6u	6085	3207,75	217,10
López 9u	6845	3044,56	206,06
López 18u	8897	1261,27	85,36
López 27u	11374	1002,84	67,87
NIST Modif.	4783	3043,07	112,38
NIST Modif. 6p	5943	6019,26	222,29
NIST Modif. 9p	7385	5171,72	190,99
NIST Modif. 18p	8324	2460,67	90,87
NIST Modif. 27p	10002	2098,63	77,50

* Multiplicador implementado utilizando somente a função β em *hardware*

Tabela 5.7: Periféricos: Áreas de Sistema e *Speedups*/Área dos Multiplicadores

Quando desenrolamos o laço do método de López 27 vezes (módulo López 27u), e quando paralelizamos o cálculo de 27 bits do resultado por ciclo de *clock* (módulos NIST 27p e NIST Modif. 27p), tivemos de reduzir a freqüência de operação do periférico para respectivamente, 100, 75 e 100 MHz. Vale a pena relembrar que mantivemos o processador a 120MHz. Como conseqüência, o tempo de cálculo da multiplicação subiu para mais de 2μs, não sendo portanto, vantajoso utilizar esses módulos uma vez que

seus *speedups* normalizados ficaram em torno de 70. Para efeito de comparação, os *speedups* normalizados das implementações mais simples (NIST, López, e NIST Modif.), que ocupam menos área, ficaram pouco acima de 50.

Pensando em ambientes restritos, temos que a melhor área é obtida quando se utiliza o multiplicador López, ou seja, 4297 ALUTs. Porém, pensando na relação de compromisso entre o ganho de desempenho e a área de implementação, observa-se que a melhor razão *speedup*/área do sistema é obtida quando utilizamos o multiplicador NIST 6p, como mostrado na Tabela 5.7. Essa razão ($8752{,}51\times10^{-4}$) é quase 3 vezes maior que a razão do multiplicador López 6u ($3207{,}75\times10^{-4}$), e aproximadamente 1,5 vezes maior que o multiplicador NIST Modif. 6p.

Entretanto, analisando o *speedup* normalizado/área do sistema, vemos que o multiplicador NIST Modif. 6p resulta em uma razão de $222{,}29\times10^{-4}$. Os sistemas com os multiplicadores López 6u e NIST 6p, possuem respectivamente razões $217{,}10\times10^{-4}$ e $189{,}45\times10^{-4}$, de onde concluímos que a multiplicação NIST Modif. 6p é mais adequada para sistemas com restrições de área.

5.3.3 Inversão

Realizamos nessa seção a avaliação dos módulos de inversão em corpos finitos utilizados como periféricos de um processador NIOS2 executando a 120MHz. Diferentemente da seção anterior, os periféricos inversores se comportaram de tal maneira que todos eles possuem freqüência de operação de 75MHz.

Inversão usando Multiplicador	*Software* (μs)	*Hardware* (μs)	*Speedup*	*Speedup* Normalizado
DHHLLM	1456,43	-	-	-
NIST	65018,64	21,24	3060,90	68,56
NIST 9p	65018,64	3,94	16495,22	369,50
NIST 27p	65018,64	2,50	26007,46	582,57
López	20009,13	21,23	942,71	68,62
López 9u	20009,13	3,93	5097,87	371,07
López 27u	20009,13	2,49	8030,42	584,52
NIST Modif.	38135,97	21,23	1796,75	68,62
NIST Modif. 9p	38135,97	3,93	9716,17	371,07
NIST Modif. 27p	38135,97	2,50	15254,39	582,57

Tabela 5.8: Periféricos: Desempenho e *Speedups* dos Inversores

Como mostrado na Tabela 5.8, a inversão pode ser computada em 1456,43μs com uma implementação inteiramente em *software* usando o eficiente multiplicador DHHLLM. Assim como no caso dos multiplicadores, os módulos inversores possuem desempenho muito parecidos.

Considerando os multiplicadores em suas versões básicas, implementados como periféricos do NIOS2, tem-se o maior *speedup* para a inversão (26007,46) quando utilizando o multiplicador NIST 27p. Porém, esse valor é resultado de uma implementação deficiente em *software* desse método (65,32ms), pois seu *speedup* normalizado (582,57) não o aponta como o método mais rápido a ser utilizado para inversão.

O inversor utilizando o multiplicador López 27u resulta em um *speedup* normalizado (584,52) um pouco maior que os obtidos com o NIST 27p e NIST Modif. 27p. Dessa maneira, López 27u apresenta-se como método multiplicador mais eficiente para se utilizar na inversão, que pode ser computada em 2,49μs.

Inversão usando Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
NIST	5930	5161,72	115,62
NIST 9p	9594	17193,26	385,13
NIST 27p	14393	18069,52	404,76
López	6396	1473,91	107,28
López 9u	8245	6182,98	450,05
López 27u	13038	6159,24	448,32
NIST Modif.	5675	3166,07	120,91
NIST Modif. 9p	8160	11907,07	454,74
NIST Modif. 27p	11790	12938,41	494,12

Tabela 5.9: Periféricos: Áreas de Sistema e *Speedups*/Área dos Inversores

Analisando a Tabela 5.9, nota-se que a inversão com melhor razão speedup/área do sistema é que utiliza o multiplicador NIST 27p. Sua razão (18069,52$\times 10^{-4}$) é aproximadamente 3 vezes maior que o sistema utilizando López 27u (6159,24$\times 10^{-4}$). Entretanto, analisando a razão *speedup* normalizado/área do sistema, o sistema com maior valor (494,12$\times 10^{-4}$) é o que utiliza o inversor baseado no multiplicador NIST Modif. 27p. No caso da inversão, o método NIST Modif. 27p mostra-se como uma boa opção para sistemas com restrições de área.

5.3.4 Multiplicação de Ponto (kP)

Nessa seção analisamos os módulos "Núcleo de kP" e "kP Completo" implementados como periférico do processador NIOS2. Devido ao seu algoritmo ser consideravelmente mais complexo que os das outras operações apresentadas até o momento, temos uma maior área de implementação para kP. Soma-se a isso uma maior complexidade de roteamento interno à FPGA, que por conseqüência, implica em uma redução inevitável de suas freqüências de operações.

Como nessa abordagem de periféricos o processador e o periférico em si operam com sinais de *clock* distintos, permitimos a utilização de freqüências de operações diferentes de 120MHz. Mais precisamente, eles podem operar em uma das seguintes freqüências 120, 100, 75, 66,67, 60 e 50MHz.

Acreditamos que esse tipo de flexibilização nos traria resultados mais interessantes do que simplesmente descartar sistemas executando abaixo de 120MHz. Assim, pudemos experimentar com uma grande diversidade de sistemas em nossa plataforma de trabalho, e assim determinar qual provê melhor resultados quando utilizado em uma aplicação real.

Núcleo de kP

Recordamos que quando utilizamos o módulo "Núcleo de kP", parte do cômputo de kP (função `Mxy`) tem de ser realizado em *software*. Analisado a Tabela 5.10 nota-se que, para um mesmo método de multiplicação finita, obtêm-se desempenhos bastante semelhantes, não importando quantos multiplicadores finitos (1, 2 ou 3) foram utilizados, ou se foi utilizado paralelismo/desenrolamento de laços.

Na realidade, quando utilizamos paralelismo/desenrolamento de laços, tivemos áreas de implementações maiores, como mostrado na Tabela 5.11. Como conseqüência, tivemos de reduzir as freqüências de operação do processador e dos periféricos, resultando em sistema mais lentos. Nesse caso, é inviável a utilização de tais módulos, como podemos observar pelos *speedups* normalizados que ficaram em sua maioria em torno de 1.

Tomemos como exemplo o periférico 3 López que obteve o melhor *speedups* normalizado (3,67), o melhor desempenho (41,95ms), e uma área de implementação (9908 ALUTs) razoável. Esse periférico executa a 75MHz, tendo o processador operado a 120MHz. Nota-se que esses resultados são melhores que a versão explorando desenrolamento de laços, 3 López 27u, uma vez que nessa última temos o processador e o periférico operando, respectivamente, a 100MHz e 66,67MHz, prejudicando bastante o desempenho como um todo. Além disso, 3 López 27u utiliza 3 vezes mais área (30334 ALUTs) que 3 López (9908 ALUTs), inviabilizando sua utilização.

Núcleo de kP usando Multiplicador	*Software* (ms)	Freq. NIOS2 (MHz)	*Hardware* (ms)	Freq. Periférico (MHz)	*Speedup*	*Speedup* Normalizado
DHHLLM	154,02	120	-	-	-	-
1 NIST	8409,08	120	139,48	75	60,29	1,10
1 NIST 27p	10090,90	100	164,95	75	61,17	0,93
1 López	2521,22	120	43,35	75	58,16	3,55
1 López 27u	3025,47	100	49,52	75	61,09	3,11
1 NIST Modif.	4929,42	120	82,79	75	59,54	1,86
1 NIST Modif. 27p	4929,42	120	80,80	75	61,01	1,91
2 NIST	8409,08	120	138,40	75	60,76	1,11
2 NIST 27p	10090,90	100	164,91	66,67	61,19	0,93
2 López	2521,22	120	42,30	75	59,61	3,64
2 López 27u	3025,47	100	49,49	66,67	61,14	3,11
2 NIST Modif.	5915,30	100	97,83	75	60,46	1,57
2 NIST Modif. 27p	5915,30	100	96,84	75	61,09	1,59
3 NIST	10090,90	100	165,56	75	60,95	0,93
3 NIST 27p	10090,90	100	164,99	66,67	61,16	0,93
3 López	2521,22	120	41,95	75	60,10	3,67
3 López 27u	3025,47	100	48,70	66,67	62,12	3,16
3 NIST Modif.	4929,42	120	81,39	75	60,53	1,89
3 NIST Modif. 27p	5915,30	100	96,85	75	61,08	1,59

Tabela 5.10: Periféricos: Desempenho e *Speedups* do "Núcleo de kP"

Partindo para a análise de qual periférico seria mais apropriado para sistemas com restrições de área, e descartando alguns módulos pelos seus insignificativos *speedups* normalizados, nos resulta basicamente o método de López, com 1, 2, e 3 multiplicadores finitos. Dentre esses, nota-se que a melhor razão *speedup* normalizado/área do sistema ($4{,}13\times10^{-4}$) é obtida com 1 López, sendo esse, o periférico mais adequado para sistemas com restrições de área.

Núcleo de kP usando Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
1 NIST	9521	63,32	1,16
1 NIST 27p	17890	34,19	0,52
1 López	8598	67,65	4,13
1 López 27u	16631	36,73	1,87
1 NIST Modif.	8446	70,49	2,20
1 NIST Modif. 27p	14163	43,08	1,35
2 NIST	9936	61,15	1,12
2 NIST 27p	25499	24,00	0,37
2 López	9618	61,97	3,79
2 López 27u	23224	26,33	1,34
2 NIST Modif.	9804	61,67	1,61
2 NIST Modif. 27p	21253	28,74	0,75
3 NIST	11544	52,80	0,81
3 NIST 27p	35576	17,19	0,26
3 López	9908	60,66	3,71
3 López 27u	30334	20,48	1,04
3 NIST Modif.	10694	56,63	1,77
3 NIST Modif. 27p	28624	21,34	0,56

Tabela 5.11: Periféricos: Áreas de Sistema e *Speedups*/Área do "Núcleo de kP"

kP Completo

Já no caso de "kP Completo" todo o algoritmo da multiplicação de ponto foi implementado em um módulo de *hardware*, e então, utilizado como um periférico do NIOS2. Analisado a Tabela 5.12 nota-se que agora, diferentemente do "Núcleo de kP", os desempenhos possuem relações diretas com o número de multiplicadores finitos (1, 2 ou 3) utilizados, e também com o uso de paralelismo/desenrolamento de laços.

Nota-se que, para os periféricos com as implementações mais simples dos multiplicadores finitos (sem paralelismo nem desenrolamento de laços), os desempenhos se mantêm bastante semelhantes. No caso dos 1 NIST, 1 López, e 1 NIST Modif., uma multiplicação de ponto pode ser calculada em 2,154ms, resultando em um *speedup* normalizado de 71,51. Já no caso dos 2 NIST e 2 López, kP é calculado em 1,089ms, enquanto 2 NIST Modif. calcula kP em 1,225ms. A diferença de desempenho desse último se deve ao fato do periférico operar a 66,67MHz, enquanto os anteriores operam a 75MHz. Por fim, 3 NIST, 3 López, e 3 NIST Modif. calculam uma multiplicação de ponto em torno de 0,738ms.

kP Completo usando Multiplicador	*Software* (ms)	Freq. NIOS2 (MHz)	*Hardware* (ms)	Freq. Periférico (MHz)	*Speedup*	*Speedup* Normalizado
DHHLLM	154,02	120	-	-	-	-
1 NIST	8409,08	120	2,154	75	3904,06	71,51
1 NIST 27p	8409,08	120	0,131	66,67	63987,90	1171,98
1 López	2521,22	120	2,154	75	1170,47	71,50
1 López 27u	4033,96	75	0,131	60	30721,65	1172,96
1 NIST Modif.	4929,42	120	2,154	75	2288,71	71,51
1 NIST Modif. 27p	4929,42	120	0,117	75	42149,78	1316,96
2 NIST	8409,08	120	1,089	75	7720,42	141,40
2 NIST 27p	16818,15	60	0,093	66,67	180420,02	1652,26
2 López	2521,22	120	1,089	75	2315,33	141,44
2 López 27u	3025,47	100	0,071	60	42415,09	2159,23
2 NIST Modif.	5915,30	100	1,225	66,67	4830,08	125,76
2 NIST Modif. 27p	5915,30	100	0,072	66,67	81635,39	2125,56
3 NIST	8409,08	120	0,738	75	11393,77	208,68
3 NIST 27p	15136,34	66,67	0,073	50	208834,68	2124,97
3 López	2521,22	120	0,739	75	3411,52	208,40
3 López 27u	4033,96	75	0,056	60	72622,85	2772,77
3 NIST Modif.	4929,42	120	0,739	75	6670,46	208,42
3 NIST Modif. 27p	5915,30	100	0,053	66,67	111315,40	2898,34

Tabela 5.12: Periféricos: Desempenho e *Speedups* de "kP Completo"

Observamos a partir desses dados, certa linearidade entre o número de multiplicadores finitos, e o desempenho: Usando dois multiplicadores, reduz-se o tempo de cômputo de kP pela metade; usando três a um terço. Infelizmente, não conseguiríamos maiores ganhos de desempenho se utilizássemos mais de três multiplicadores finitos para a implementação de "kP Completo". Três é o limite de nível de paralelismo que o algoritmo permite, ou seja, se adicionássemos mais um multiplicador, o desempenho do sistema permaneceria o mesmo, porém estaríamos utilizando maior área de implementação.

Sendo assim, exploramos a única opção restante: utilizar maior paralelismo e desenrolamento de laços internamente aos multiplicadores finitos. Nesse caso, através da Tabela 5.12 notamos que os *speedups* aumentam bastante, sendo o maior deles (208834,68) obtido com o periférico 3 NIST 27p. Entretanto, isso é resultado de sua implementação deficiente em *software*.

Analisado os *speedups* normalizados, nota-se que o maior ganho real de desempenho se dá pelo uso de 3 NIST Modif. 27p, cujo *speedup* normalizado é de 2898,34. Esse, na verdade, é o periférico que computa kP mais rapidamente, ou seja, em apenas 53μs.

Comparando o resultado desse periférico com seu módulo *stand-alone*, o qual computa kP em 36,30μs, observamos nesse caso uma diferença de 46% entre as duas medições. Vale a pena recordar que para "kP Completo" são necessárias menos chamadas de leitura ao periférico (leitura das coordenadas x e y) do que eram necessárias no "Núcleo de kP" (leitura de X_1,Z_1,X_2, e Z_2). Mais, no caso de "kP Completo", nenhuma parte do algoritmo é executada em *software*, o que favorece tal ganho de desempenho. Dessa forma, temos que 3 NIST Modif. 27p é o periférico recomendado para aplicações que requerem desempenho com primeira prioridade.

kP Completo usando Multiplicador	Área do Sistema (ALUTs)	*Speedup* / Área do Sistema ($\times 10^{-4}$)	*Speedup* Normalizado / Área do Sistema ($\times 10^{-4}$)
1 NIST	12389	3151,23	57,72
1 NIST 27p	19919	32124,05	588,37
1 López	11519	1016,12	62,07
1 López 27u	20903	14697,24	561,15
1 NIST Modif.	12245	1869,09	58,40
1 NIST Modif. 27p	17763	23728,98	741,40
2 NIST	15057	5127,43	93,91
2 NIST 27p	30360	59426,88	544,22
2 López	16473	1405,53	85,86
2 López 27u	30570	13874,74	706,32
2 NIST Modif.	13878	3480,38	90,62
2 NIST Modif. 27p	25546	31956,23	832,05
3 NIST	16829	6770,32	124,00
3 NIST 27p	37614	55520,47	564,94
3 López	14343	2378,52	145,30
3 López 27u	36255	20031,13	764,80
3 NIST Modif.	14866	4487,06	140,20
3 NIST Modif. 27p	31928	34864,51	907,78

Tabela 5.13: Periféricos: Áreas de Sistema e *Speedups*/Área de "kP Completo"

O sistema com maior razão *speedup*/área do sistema (59426,88$\times 10^{-4}$) é o 2 NIST 27p como conseqüência de seu alto *speedup*, o qual, por sua vez é resultante de sua deficiente implementação em *software*. Pensando em sistemas com restrições de área, observa-se que o sistema com melhor razão *speedup* normalizado/área do sistema (907,78$\times 10^{-4}$) é o 3 NIST Modif. 27p, sendo esse implementado em 31928 ALUTs. Dessa maneira, esse periférico se apresenta também como melhor opção para ambientes restritos.

5.3.5 Resumo da Análise das Operações

Essa seção resume as discussões sobre as implementações dos módulos de *hardware* como periféricos do processador NIOS2. A Tabela 5.14 mostra os tempos gastos nas execuções das operações, as áreas de implementação dos sistemas. No caso das operações que utilizam multiplicações em corpos finitos, o método de multiplicação é informado.

Observando essa tabela, conclui-se que para as operações de multiplicação, inversão, e kP com seu núcleo em *hardware*, o método de López, em suas mais variadas formas, é o mais apropriado para sistemas voltados a velocidade. Além disso, para essas mesmas operações, conclui-se que o método NIST Modif., na maioria dos casos, é o mais indicado para sistemas com restrições de área. Entretanto, consideremos a multiplicação de pontos inteiramente em *hardware*, implementada como periférico sob o nome de (kP Completo). Nesse caso temos o periférico 3 NIST Modif. 27p como o mais recomendado tanto para sistemas voltados a desempenho como para ambientes restritos em área.

Operação		Sistemas	
		Velocidade	Restritos
Adição	Desempenho	0,94μs	
	Área	3553 ALUTs	
Quadrado	Desempenho	0,68μs	
	Área	3349 ALUTs	
Raiz Quadrada	Desempenho	0,68μs	
	Área	3786 ALUTs	
Trace	Desempenho	0,43μs	
	Área	3432 ALUTs	
Solução de E.Q.	Desempenho	0,68μs	
	Área	3582 ALUTs	
Função β	Desempenho	0,69μs	
	Área	4018 ALUTs	
Multiplicação	Método	López 9u	NIST Modif. 6p
	Desempenho	1,11μs	1,18μs
	Área	6845 ALUTs	5943 ALUTs
Inversão	Método	López 27u	NIST Modif. 27p
	Desempenho	2,49μs	2,50μs
	Área	13038 ALUTs	11790 ALUTs
Núcleo de kP	Método	3 López	1 López
	Desempenho	41,95ms	43,35ms
	Área	9908 ALUTs	8598 ALUTs
kP Completo	Método	3 NIST Modif. 27p	
	Desempenho	53μs	
	Área	31928 ALUTs	

Tabela 5.14: Periféricos: Resumo das Análises das Operações

Capítulo 6

Comparações entre Instruções Especializadas e Periféricos

Nesse capítulo comparamos instruções especializadas e periféricos e apresentamos as vantagens e desvantagens de cada uma das abordagens em termos de número de chamadas, armazenamento interno, área de implementação do módulo de *hardware*, e ganho de desempenho do sistema. Para realizar as comparações das operações elementares, tomamos como base as Tabelas 4.3, 4.4, 5.4, e 5.5. Já para os multiplicadores nos baseamos nas Tabelas 4.5, 4.6, 5.6, e 5.7. Para os inversores nos baseamos nas Tabelas 5.8 e 5.9, e por fim, para os multiplicadores de ponto, nas Tabelas 5.12 e 5.13.

As Figuras 6.1, 6.2, 6.3, 6.4, 6.5, e 6.6 mostram apenas os gráficos dos melhores resultados obtidos para a multiplicação, inversão e multiplicação de pontos, não importando se a operação foi implementada como instrução especializada ou como periférico.

6.1 Adição

Adotando-se a abordagem de instrução especializada, a adição pode ser realizada através de seis chamadas ao módulo de *hardware*. Em cada chamada de instrução especializada envia-se ao módulo dois argumentos de 32 bits, e retorna-se um bloco de 32 bits correspondendo ao resultado parcial da soma. Dessa maneira, bastam seis chamadas à instrução para realizar uma soma de 163 bits. Já utilizando periféricos, são necessárias dezoito iterações com o módulo de *hardware* para realizar a soma de 163 bits. Isso acontece devido ao fato do periférico possuir apenas uma entrada de dados de 32 bits, enquanto a instrução especializada possui duas entradas de 32 bits. Além disso, embora ambos possuam apenas uma saída de resultado de 32 bits, tem-se maior eficiência nas chamadas das instruções especializadas, dado que em cada chamada já são retornados 32 bits do resultado. No caso dos periféricos além das duas escritas de parâmetros, tem-se de fazer

mais uma chamada para leitura dos resultados.

A abordagem de periféricos também é pouco vantajosa no tocante à necessidade de armazenamento interno. Enquanto as instruções especializadas não necessitam de armazenamento interno, os periféricos obrigatoriamente necessitam de um registrador interno de 32 bits para armazenar temporariamente argumentos e resultados.

Entretanto, essa diferença de armazenamento interno não influencia na área final do sistema dado que o sistema especializado para adição, utilizando instrução especializada, possui área de implementação de 3596 ALUTs, enquanto o baseado em periféricos ocupou 3553 ALUTs. Sendo assim, como essa característica se repete na maioria das operações elementares que possuem módulos bastante simples, temos um indicativo que o NIOS2 utiliza menor lógica para integrar um periférico do que seria necessário para integrar uma instrução especializada.

A adição implementada como instrução especializada pode computar uma operação em 0,44μs, enquanto sua implementação como periférico necessita de 0,94μs. Esse resultado é reflexo da interface do periférico que possui apenas uma entrada de 32 bits, contra duas entradas de 32 bits das instruções especializadas, sendo assim necessário um maior número de chamadas aos periféricos para executar a adição. Observamos queda de desempenho em todos os módulos das operações elementares devido a esse maior número de chamadas.

Devido a essas razões, concluímos que seria mais apropriado utilizar a abordagem de instrução especializada para a adição. Entretanto, como essa operação é uma operação lógica *xor* já encontrada no conjunto de instruções do NIOS2, concluímos que essa operação não traz nenhuma vantagem quando inserida em um NIOS2 especializado para criptografia. Salvo raras exceções, pode-se expandir essas conclusões para processadores de propósito geral normalmente encontrados no mercado.

6.2 Quadrado

Implementar o quadrado como instrução especializada, faz-se uso da grande vantagem de se ter duas entradas de dados de 32 bits, quando comparado com a única entrada de dados de 32 bits presente nos periféricos. Outra vantagem é a de se obter 32 bits de resultado a cada chamada de instrução especializada. Como conseqüência, consegue-se realizar a operação de quadrado nos 163 bits do argumento com apenas seis chamadas de instruções especializadas, e ainda, usar apenas 3 bits de armazenamento interno ao módulo. Já na abordagem de periféricos, são necessárias doze chamadas para realizar a mesma operação, e também, utilizar 35 bits de armazenamento interno ao módulo.

Assim como a adição, o sistema especializado para o quadrado também usa uma área menor de implementação quando implementado como periférico, ou seja, 3349 ALUTs, contra as 3593 ALUTs necessárias para a implementação utilizando instrução especia-

lizada. O sistema com a abordagem de instrução especializada calcula o quadrado em 0,51μs, ou seja, é mais rápida que os 0,68μs do sistema usando periféricos. O uso do sistema utilizando periféricos é descartado pois esse é mais lento que a implementação em *software* (0,59μs) utilizando o conjunto de instruções original do NIOS2.

Como resultado, a abordagem de instrução especializada mostra-se mais apropriada para a implementação da operação de quadrado.

6.3 Raiz Quadrada

Da mesma maneira que a adição e o quadrado, consegue-se realizar a operação de raiz quadrada com apenas seis chamadas de instruções especializadas, utilizando para isso, 32 bits de armazenamento interno ao módulo de *hardware*. Na abordagem de periféricos são necessárias doze chamadas, e 64 bits de armazenamento, ou seja, dobra-se o número de chamadas e de bits para armazenar dados temporários.

De maneira semelhante ao quadrado, o sistema usando o periférico de raiz quadrada possui menor área (3786 ALUTs) que o utilizando instrução especializada (3851 ALUTs). Porém o sistema com instrução especializada computa a raiz quadrada em 0,50μs, ou seja, mais rápido que os 0,68μs do sistema utilizando periférico. O sistema utilizando periféricos é descartado por ser mais lento que a melhor implementação em *software* (0,52μs) utilizando o conjunto de instruções original do NIOS2.

Novamente, para a implementação da raiz quadrada tem-se também instruções especializadas como abordagem mais apropriada.

6.4 *Trace*

Diferentemente das operações anteriores, o *trace* implementado como periférico necessita do mesmo número de registradores que quando implementado como instrução especializada. Em ambas as abordagens, faz-se necessário apenas um bit de armazenamento interno. Entretanto, pela interface de periféricos contar com apenas uma entrada de 32 bits, faz-se necessário seis iterações com o módulo de *hardware* para computar o *trace* de um elemento de 163 bits. Dado que a instrução especializada possui duas entradas de 32 bits, ela utiliza apenas três chamadas às instruções especializadas.

O *trace* é outro exemplo de operação cujo sistema ocupa menor área quando utilizando periféricos. São necessárias nessa abordagem 3432 ALUTs, enquanto o sistema com instrução especializada necessita de 3834 ALUTs. No caso do *trace* temos o sistema utilizando instrução especializada executando o cálculo do *trace* (0,28μs) 1,5 vezes mais rápido que o utilizando periféricos (0,43μs), sendo ainda esse último tempo idêntico à

implementação em *software*, também 0,43μs.

Assim, para a operação de *trace* a abordagem de instrução especializada mostra-se mais apropriada para sua implementação em *hardware*.

6.5 Solução de Equação Quadrática

Enquanto a implementação da solução de equação quadrática como instrução especializada exige armazenamento interno de apenas um bit, e são necessárias apenas seis chamadas ao módulo de *hardware*. Já sua implementação como periférico exige 32 bits de armazenamento interno, e doze iterações com o módulo.

Para a solução de equação quadrática temos que seu sistema utilizando instrução especializada ocupa 3497 ALUTs e computa uma operação em 0,53μs, enquanto o utilizando periférico utiliza 3582 ALUTs e realiza a operação em 0,68 μs. Dessa maneira, o sistema utilizando instrução especializada é mais eficiente tanto em termos de área quanto desempenho.

Como resultado, na implementação em *hardware* da solução de equação quadrática é mais apropriada a abordagem de instruções especializadas.

6.6 Função β

A função β faz parte da classe de operações não-divisíveis, pois precisamos de todos os 163 bits do argumento para realizar a permutação. Dessa forma, tanto na abordagem de instruções especializadas quanto periféricos, são necessários dois registradores de 163 bits para armazenar o argumento e o resultado. Da mesma maneira que as outras operações elementares, a função β necessita de mais chamadas ao periférico para escrever o argumento, do que as que necessárias quando utilizamos instruções especializadas.

O sistema de cômputo da função β utilizando instruções especializadas ocupa área maior (4482 ALUTs) que o utilizando periféricos (4018 ALUTs). Utilizando a abordagem de instrução especializada a operação é executada em um tempo menor (0,51μs) que se utilizando periférico (0,69μs). Como fator de decisão, analisamos a razão *speedup*/área do sistema. Essa razão para o sistema com instrução especializada ($119{,}60\times10^{-4}$) é mais alto que o do sistema com periférico ($98{,}05\times10^{-4}$).

Por fim, concluímos que a função β é mais apropriada para implementações em *hardware* utilizando instruções especializadas.

6.7 Multiplicação

A multiplicação em corpos finitos também é considerada uma operação não-divisível dado que é bastante difícil partir sua execução em blocos de 32 bits. Dessa forma, em ambos os casos de instrução especializada e periférico, necessita-se de dois registradores de 163 bits para armazenamento dos argumentos da multiplicação, e para armazenar o resultado da operação, mais um registrador de 163 bits .

Devido ao módulo de instrução especializada possuir duas entradas de dados de 32 bits, consegue-se escrever os argumentos utilizando seis chamadas à instrução. Como a última chamada de escrita já retorna 32 bits do resultado, são necessárias mais cinco chamadas de leitura para obter o restante do resultado. Assim, tem-se o total de onze chamadas ao módulo multiplicador, quando esse é utilizado como instrução especializada. A interface de periféricos possui apenas uma entrada de dados de 32 bits fazendo com que sejam necessárias doze chamadas para escrita de argumentos. Como a última chamada não retorna valores, mais seis chamadas ao periférico são necessárias para se obter o resultado. Dessa forma, tem-se o total de dezoito chamadas para realizar uma multiplicação.

Pelas análises das instruções especializadas e periféricos, percebemos que a operação de multiplicação está no limiar entre as duas abordagens. Dessa forma, faz-se necessário analisar os módulos em termos de área e desempenho com bastante atenção, de forma a determinar a abordagem mais apropriada para a multiplicação.

No caso de periféricos, diferentemente das instruções especializadas, as freqüências de operação dos periféricos não dependem da freqüência de operação do processador. Assim sendo, pode-se tomar proveito dessa característica para manter os periféricos executando a uma freqüência mais alta. Entretanto, tomando com base as Tabelas 4.5 e 5.6, pode-se notar que os desempenhos dos periféricos das multiplicações NIST (2,04μs), López (2,03μs), e NIST Modif. (2,03μs) executando a 150MHz, são piores que os desempenhos das instruções especializadas de multiplicação NIST (2,92μs), López (2,90μs), e NIST Modif. (2,91μs) executando a 120MHz. Dessa maneira, observa-se que a entrada de 32 bits dos periféricos prejudicam bastante o desempenho, mesmo se executando em freqüências mais altas.

Mais, quando se tenta aumentar o desempenho com periféricos com mais paralelismo/desenrolamento de laços, os sistemas passam a rodar em freqüências menores, inviabilizando tal otimização. Observando a Figura 6.1, nota-se então que o nível máximo de paralelismo/desenrolamento de laços, sem prejudicar o ciclo de *clock* do processador, é de 18, obtido com a instrução especializada López 18u, sendo esse o multiplicador mais apropriado para aplicações almejando desempenho.

Continuando com a análise de desempenho entre módulos utilizando o mesmo nível de paralelismo e desenrolamento de laços, as instruções especializadas ainda se mostram mais

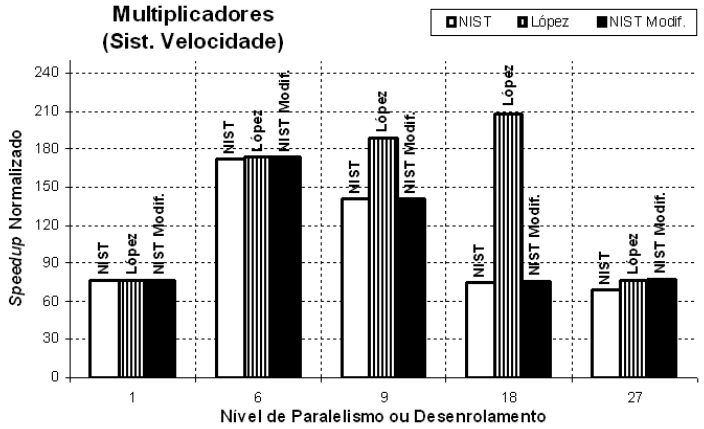


Figura 6.1: Gráfico de Análise dos Multiplicadores para Sistemas de Velocidade

eficientes que os periféricos. Tomando como exemplo o caso de instrução especializada López 18u que é a que possui o maior *speedup* normalizado (208,43) entre os multiplicadores. Já o mesmo módulo multiplicador implementado como periférico, possui *speedup* normalizado de apenas 75,95. Os periféricos NIST 9p, López 9u, e NIST Modif. 9p são os multiplicadores com maior *speedup* normalizados (141,05). Esse resultado é menor do que todos os obtidos com os módulos implementados como instruções especializadas que fizeram uso de paralelismo e desenrolamento de laços.

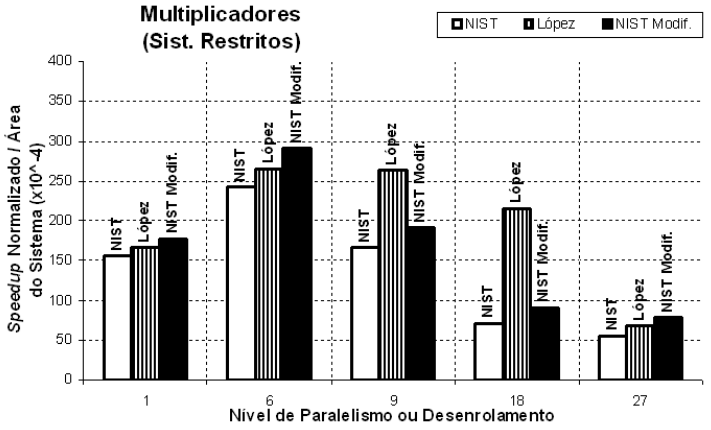


Figura 6.2: Gráfico de Análise dos Multiplicadores para Sistemas Restritos

Partindo para a análise das áreas de implementação do sistema, tomando como referência as Tabelas 4.6 e 5.7, nota-se que as áreas, para um mesmo nível de paralelismo/desenrolamento de laços, têm-se que os periféricos ocupam ligeiramente menos área. Entretanto, atentando para os periféricos, com mesmo nível de paralelismo/desenrolamento

de laços, nota-se que todos sem exceção possuem a razão *speedup* normalizado/área do sistema menor que as instruções especializadas. Como mostrado graficamente na Figura 6.2, a instrução especializada com maior razão *speedup* normalizado/área do sistema é a NIST Modif. 6p, sendo essa a mais apropriada para sistemas com restrição de área.

Por fim, levando em consideração essa discussão, concluímos que a abordagem mais apropriada para se implementar os multiplicadores é a de instruções especializadas.

6.8 Inversão

Na implementação das operações aritméticas em corpos finitos utilizando a abordagem de instruções especializadas assumimos a freqüência mínima de operação de 120MHz. Em outra palavras, sistemas com freqüência de operação abaixo de 120MHz não foram considerados em nossos experimentos. Fizemos isso para poder ter um ponto de referência de quais operações eram mais apropriadas para serem utilizadas como instrução especializadas (sem alterar a freqüência de operação do processador), e quais deveriam ser movidas para periféricos.

Infelizmente, nenhum sistema com módulos inversores implementados como instrução especializada atingiu freqüência de operação maior que 120MHz. Como resultado, implementamos essa operação como periférico, podendo assim, utilizar freqüências menores de operações para os inversores, e ainda assim manter o NIOS2 a 120MHz.

Através das análises dos módulos inversores apresentados nos Capítulos 4 e 5, tivemos indícios bastante claros que devemos utilizar a abordagem de periféricos para implementar a inversão em *hardware*.

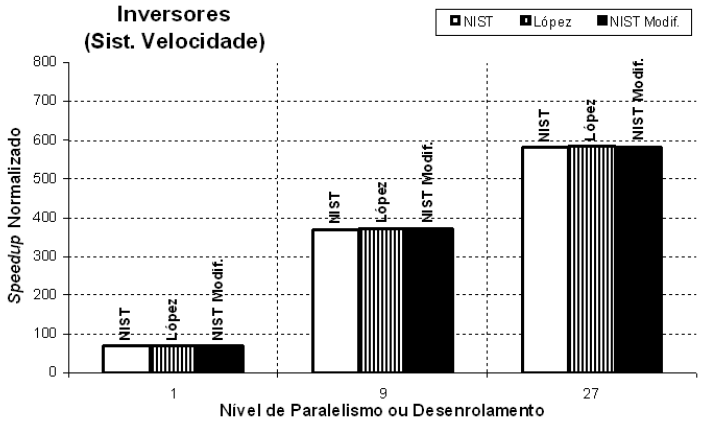


Figura 6.3: Gráfico de Análise dos Inversores para Sistemas de Velocidade

Observando a Figura 6.3 observa-se que os *speedups* normalizados são muito parecidos, ou seja, nenhum método de multiplicação se sobressaiu quando utilizado na inversão. Na verdade, o inversor usando o multiplicador López 27u apresenta um *speedup* normalizado (584,52) ligeiramente superior aos dos demais. Como conseqüência, recomendamos esse módulo para sistemas voltados a velocidade.

Já para sistemas com restrições de área, como mostrado graficamente na Figura 6.4, recomenda-se o inversor utilizando o multiplicador NIST Modif. 27p por possuir uma melhor razão *speedup* normalizado/área do sistema.

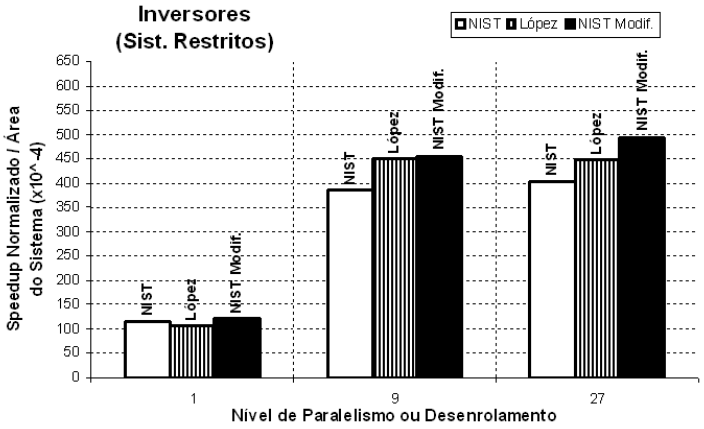


Figura 6.4: Gráfico de Análise dos Inversores para Sistemas Restritos

6.9 Multiplicação de Pontos (kP)

O algoritmo de multiplicação de pontos quando implementado em *hardware*, tanto como "Núcleo de kP" quanto "kP Completo", resulta em circuitos bastante custosos em área, que acabam por reduzir suas freqüências de operação. De forma semelhante à inversão, não foi possível implementar os módulos de cômputo da multiplicação como instrução especializada.

Sendo assim, essa é outra operação a ser utilizada como periférico do NIOS2, que na verdade, pode ser tratado como um co-processador criptográfico para curvas elípticas.

A Figura 6.5 mostra graficamente que o melhor multiplicador de pontos para sistemas voltados a desempenho é obtido utilizando-se 3 multiplicadores NIST Modif. 27p operando em paralelo. Isso porque essa é a implementação que apresenta o maior *speedup* normalizado.

Além disso, como mostra o gráfico da Figura 6.6, o 3 NIST Modif. 27p também apresenta a melhor relação *speedup* normalizado/área do sistema, tornando-o também apropriado para sistemas com restrições de área.

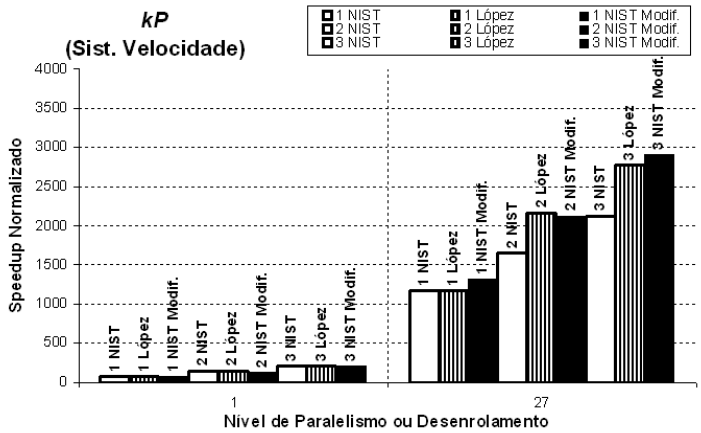


Figura 6.5: Gráfico de Análise dos Multiplicadores de Pontos para Sistemas de Velocidade

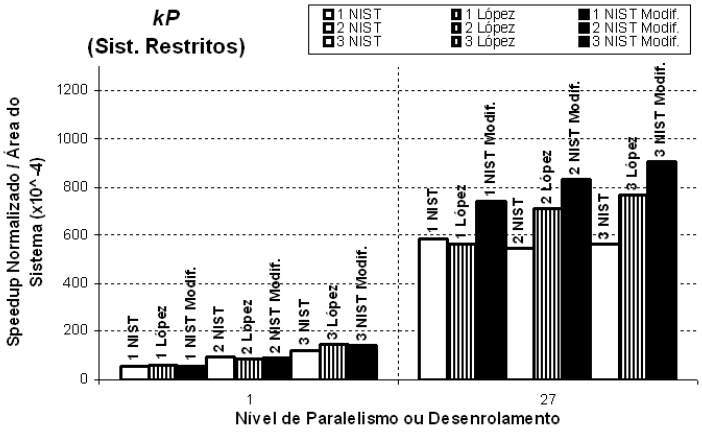


Figura 6.6: Gráfico de Análise dos Multiplicadores de Pontos para Sistemas Restritos

6.10 Resumo das Comparações

Nessa seção sintetizamos as abordagens de implementações para cada uma das implementações, como mostrado na Tabela 6.1. Por se tratar de uma operação lógica *xor*, a adição não é uma operação propícia de ser implementada em *hardware*, dado que o conjunto de instruções do NIOS2 já possui tal operação lógica. As operações de quadrado, raiz quadrada, *trace*, solução de equação quadrática, função β resultam em melhores desempenhos quando implementadas em *hardware* como instruções especializadas.

A multiplicação, mesmo provendo resultados satisfatórios como periféricos, ainda é mais bem implementada em *hardware* como instruções especializadas. Isso vale tanto para sistemas voltados a desempenho quanto a sistemas restritos em área. Já as operações de inversão e multiplicação de pontos (kP), por conta das complexidades de seus algoritmos, são casos típicos de serem implementadas em *hardware* como periféricos. Novamente, essa conclusão é válida para sistemas voltados a desempenho e para sistemas restritos em área.

Operação		Sistemas		Tipo de Implementação
		Velocidade	Restritos	
Adição	Desempenho	0,44μs		SW
	Área	3596 ALUTs		Conj. Instruções NIOS2
Quadrado	Desempenho	0,51μs		HW
	Área	3593 ALUTs		Instrução Especializada
Raiz Quadrada	Desempenho	0,50μs		HW
	Área	3851 ALUTs		Instrução Especializada
Trace	Desempenho	0,28μs		HW
	Área	3834 ALUTs		Instrução Especializada
Solução de E.Q.	Desempenho	0,53μs		HW
	Área	3497 ALUTs		Instrução Especializada
Função β	Desempenho	0,51μs		HW
	Área	4482 ALUTs		Instrução Especializada
Multiplicação	Método	López 18u	NIST Modif. 6p	HW
	Desempenho	0,75μs	0,90μs	Instrução Especializada
	Área	9649 ALUTs	5978 ALUTs	
Inversão	Método	López 27u	NIST Modif. 27p	HW
	Desempenho	2,49μs	2,50μs	Periférico
	Área	13038 ALUTs	11790 ALUTs	
kP	Método	3 NIST Modif. 27p		HW
	Desempenho	53μs		Periférico
	Área	31928 ALUTs		

Tabela 6.1: Resumo das Comparações entre Instruções Especializadas e Periféricos

Capítulo 7

Conclusões

Estudamos nesse trabalho a especialização de arquiteturas para criptografia em curvas elípticas, utilizando instruções especializadas e periféricos. Mais precisamente, exploramos a aritmética em corpos finitos sobre $\mathbb{F}_{2^{163}}$ e a multiplicação de pontos (kP). Além disso, analisamos os desempenhos dos módulos *stand-alone*.

Tomando como base o algoritmo de López-Dahab, e utilizando três multiplicadores baseados no método modificado do NIST (3 NIST Modif. 27p), implementamos um módulo de *hardware* capaz de realizar uma multiplicação de pontos em apenas 36,30μs. Na literatura especializada, encontra-se como o menor tempo para cálculo de kP o trabalho de Ansari et al. [43], onde kP é calculado em 41,67μs. Dessa maneira, não só mostramos nesse trabalho que bases normais Gaussianas são bastante apropriadas para implementações em *hardware*, mas também, até onde é de nosso conhecimento, que ela resulta no mais eficiente multiplicador de pontos da literatura.

Estudamos também as operações utilizadas em ECC através de análises *end-to-end*, visando a aplicações reais, e que propiciam melhor precisão nos ganhos reais de desempenhos resultantes de implementações em *hardware*. Através dessas análises, determinamos que as operações de quadrado, raiz quadrada, *trace*, solução de equação quadrática, função β, e a multiplicação em corpos finitos resultam em melhor desempenho quando implementadas como instruções especializadas em um processador NIOS2. As operações mais complexas, como inversão e multiplicação de pontos, propiciam excelentes resultados quando implementadas como periféricos do NIOS2.

Assumimos agora, nosso multiplicador em corpos finitos mais eficiente como uma instrução especializada. Adotando uma abordagem HW/SW, onde somente a multiplicação López 18u é computada em *hardware* como instrução especializada, a multiplicação de pontos pode ser computada em 1,44μs. Esse resultado é 126,37 vezes mais rápido que a melhor implementação em *software* no NIOS2, a qual utiliza o método DHHLLM para a multiplicação finita. Se fossemos utilizar essa mesma abordagem HW/SW em *smart-cards*

e pequenos sensores, seria mais interessante adotar o multiplicador NIST Modif. 6p. Com esse multiplicador, kP é calculado 1,62ms, e a sua área de implementação o torna mais apropriado para sistemas com restrições de área.

Partindo agora para a implementação de todo o multiplicador de pontos em *hardware* como um periférico do NIOS2, e utilizando 3 multiplicadores finitos baseados no método modificado do NIST (3 NIST Modif. 27p), temos kP sendo computado em apenas 53μs. Esse resultado é aproximadamente 2900 vezes mais rápido que a melhor implementação de kP em *software* no NIOS2. Mostramos nesse trabalho que essa implementação é simultaneamente apropriada para sistemas voltados a desempenho e para sistemas com restrições de área.

Comparando esse último resultado(53μs) com o desempenho do módulo *stand-alone* (36,30μs), observa-se uma grande diferença nos tempos das duas medições, que chega a ser de 46%. Mesmo tendo inúmeros trabalhos utilizando essa medições *stand-alone*, concluímos ser muito mais interessante nos basearmos em medições *end-to-end* de forma a ter maior precisão dos ganhos reais de desempenho resultantes de uma implementação em *hardware*.

Capítulo 8

Trabalhos Futuros

Nesse trabalho experimentamos com a aritmética em corpos finitos e a multiplicação de pontos sobre $\mathbb{F}_{2^{163}}$ utilizando bases normais Gaussianas. Embora, na atualidade, a utilização de $\mathbb{F}_{2^{163}}$ 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, como por exemplo $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$, e $\mathbb{F}_{2^{571}}$.

De forma a ter uma comparação mais precisa de trabalhos futuros com este trabalho, sugerimos fortemente que seja utilizado o mesmo processador NIOS2 versão 5.0, as mesmas versões de ferramentas QuartusII (versão 5.0) e NIOSIDE (versão 5.0), e o mesmo kit de desenvolvimento NIOS2 da Altera [36, 42]. Seria interessante seguir nossa metodologia de maneira a se poder comparar os casos de instruções especializadas e periféricos em diversos tamanhos de corpos binários.

A pesquisa realizada nesse trabalho tomou como base implementações em bases polinomiais de terceiros, que utilizaram plataformas de trabalhos e sistemas de medições de desempenho diferentes dos nossos. Para se ter maior precisão dos reais benefícios das bases normais Gaussianas, seria interessante realizar as mesmas implementações (instruções especializadas e periféricos), incluindo os corpos supracitados, e bases polinomiais.

Observamos, através de nossos experimentos sobre $\mathbb{F}_{2^{163}}$, que é bastante difícil expandir uma determinada implementação da aritmética finita em bases normais Gaussianas para um corpo de maior tamanho. Assim sendo, esses estudos complementares de corpos de maior tamanho usando GNBs e bases polinomiais, poderiam indicar qual tipo de base provê melhor reaproveitamento de módulos de *hardware* de forma a se poder ter um sistema criptográfico configurável em tempo de execução. Isso, evidentemente, sem se ter a necessidade de adotar uma implementação para cada tamanho de corpo.

Pode-se notar que consideramos esse trabalho apenas as avaliações de *speedup* e área do sistema, bem como suas variações, *speedup* normalizado, *speedup*/área do sistema, *speedup* normalizado/área do sistema. Consideramos muito interessante complementar o trabalho com análises de consumo de potência de cada um dos módulos, bem como estudar a resistência a *side-channel attacks* de cada um dos sistemas implementados.

Por fim, mesmo de posse de altos *speedups* para os módulos de multiplicação de corpo, poderiam ser adotados maior paralelismo e desenrolamento de laços internamente aos multiplicadores em corpos finitos. De forma a aumentar ainda mais o desempenho do periférico, seria bom utilizar uma estratégia de *pipeline* entre as escritas, execuções e leituras de resultados do periférico. Em nossas implementações atuais, tem-se três etapas distintas. Primeiro, escreve-se os argumentos no periférico, que em seguida dispara a execução da operação. Essa última chamada fica bloqueada até o término da execução. Por fim, os dados são lidos do periférico. No caso de se adicionar 3 registradores de 163-bits ao módulo, poder-se-ia, durante uma execução em *hardware*, ler os resultados da execução anterior e também escrever os dados da execução seguinte. Essa estratégia faria com que se obtivesse desempenhos similares aos obtidos quando avaliamos os módulos *stand-alone*. Isso porque o interfaceamento com o módulo ocorreria em paralelo a uma execução, e assim, poder-se-ia ter uma execução em seguida da outra, sem nenhum atraso.

Apêndice A

Exemplos de Estratégias de Chamadas aos Módulos de *Hardware*

A.1 Operações Divisíveis

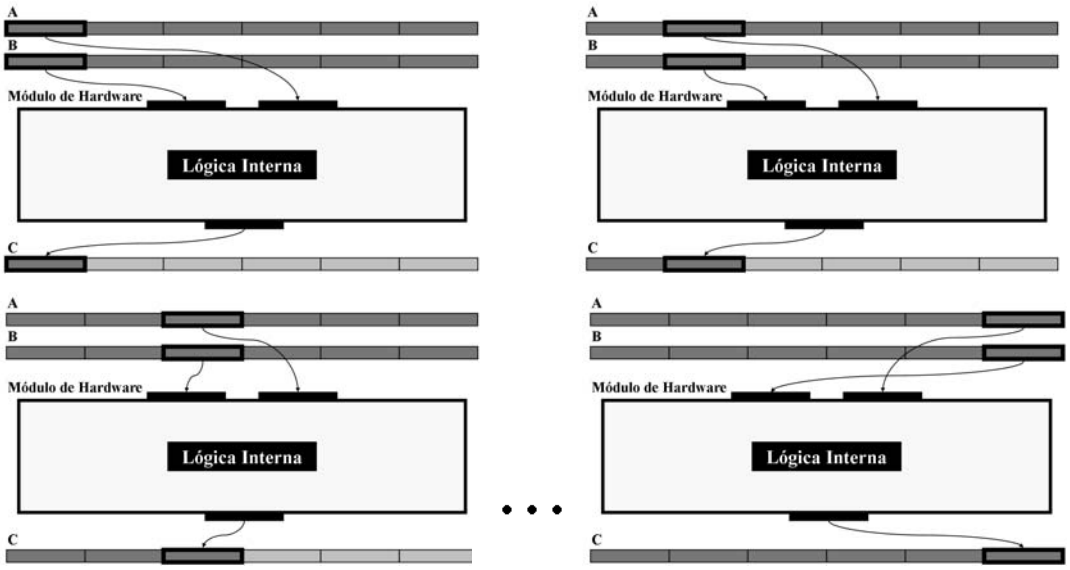


Figura A.1: Chamadas às Instruções Especializadas

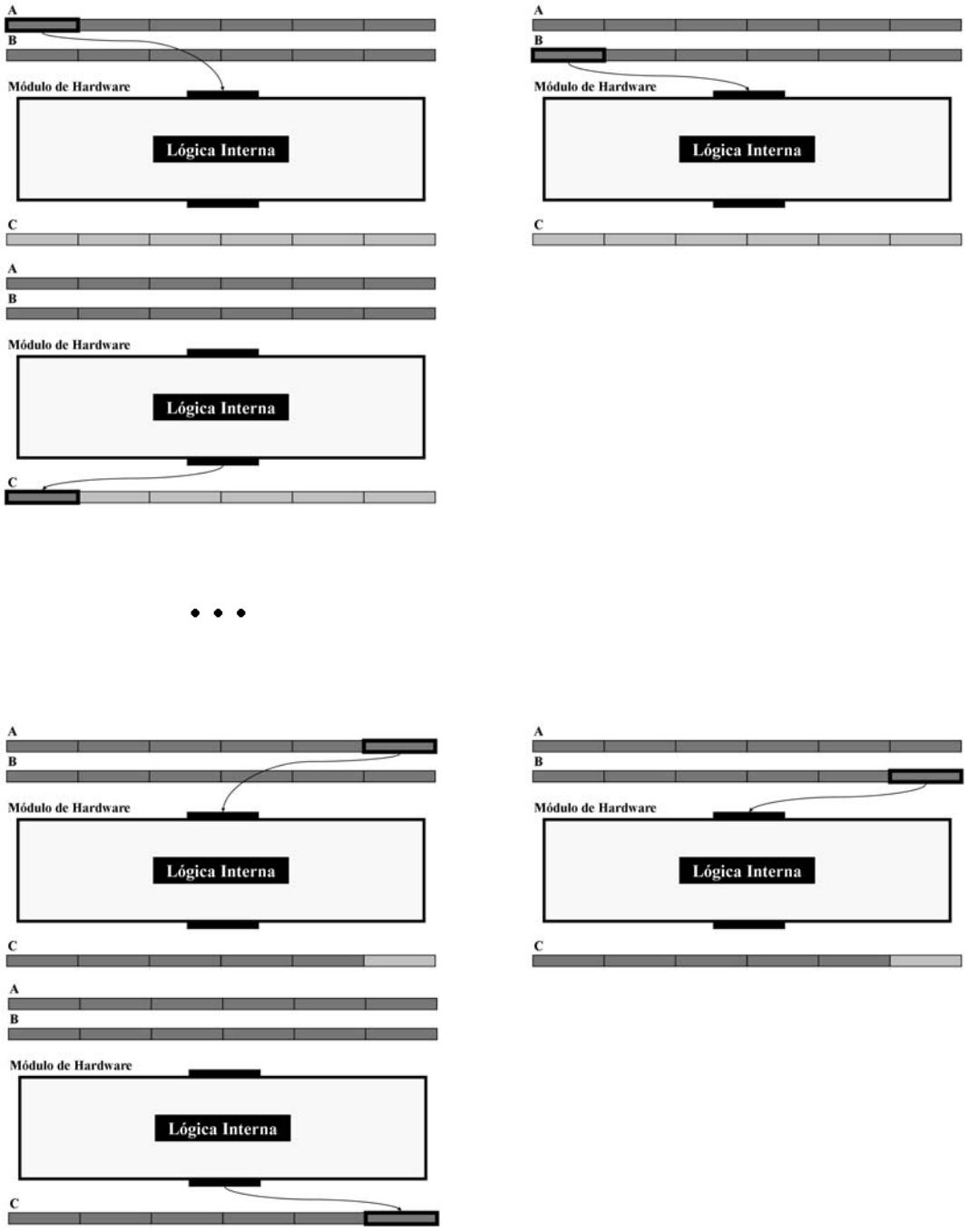


Figura A.2: Chamadas aos Periféricos

A.2 Operações Não-Divisíveis

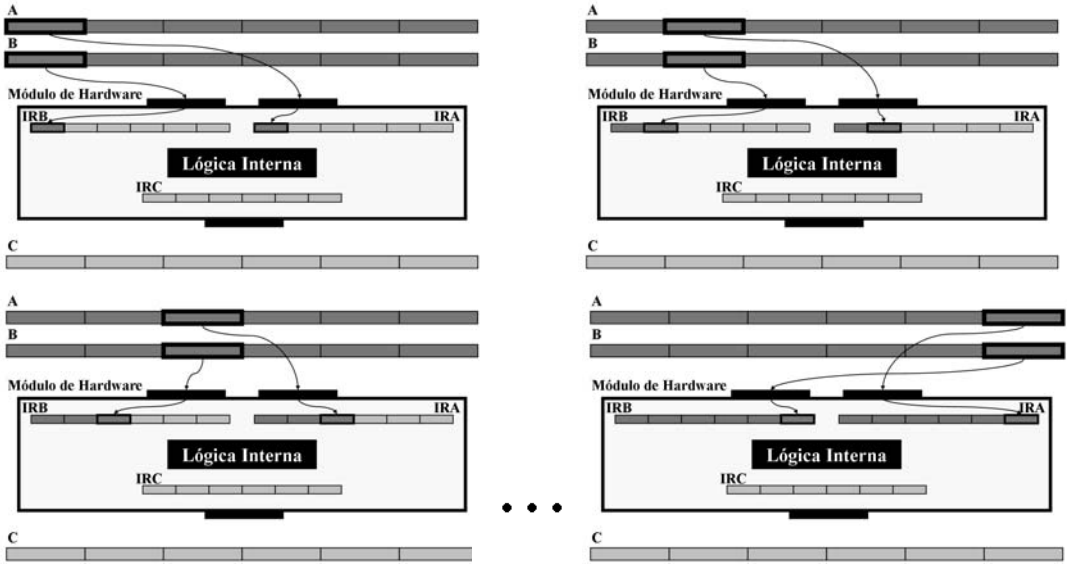


Figura A.3: Escrita de Argumentos nos Registradores das Instruções Especializadas

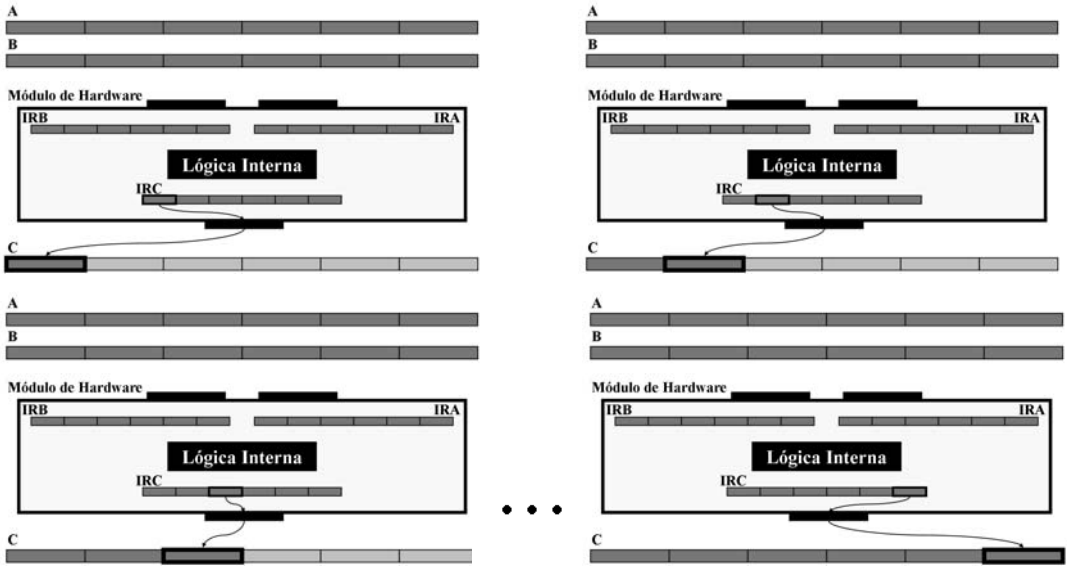


Figura A.4: Leitura de Resultados dos Registradores das Instruções Especializadas

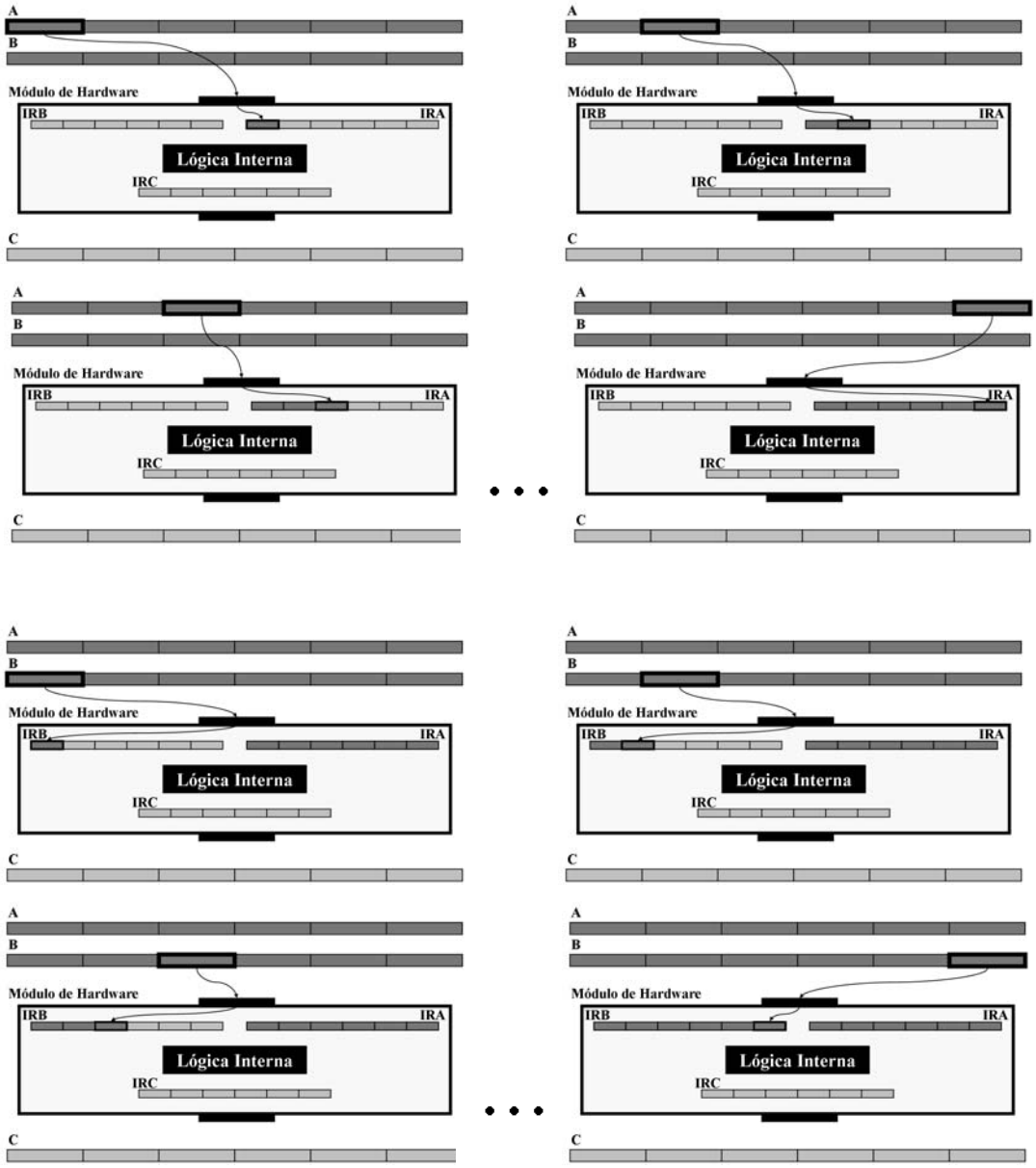


Figura A.5: Escrita de Argumentos nos Registradores dos Periféricos

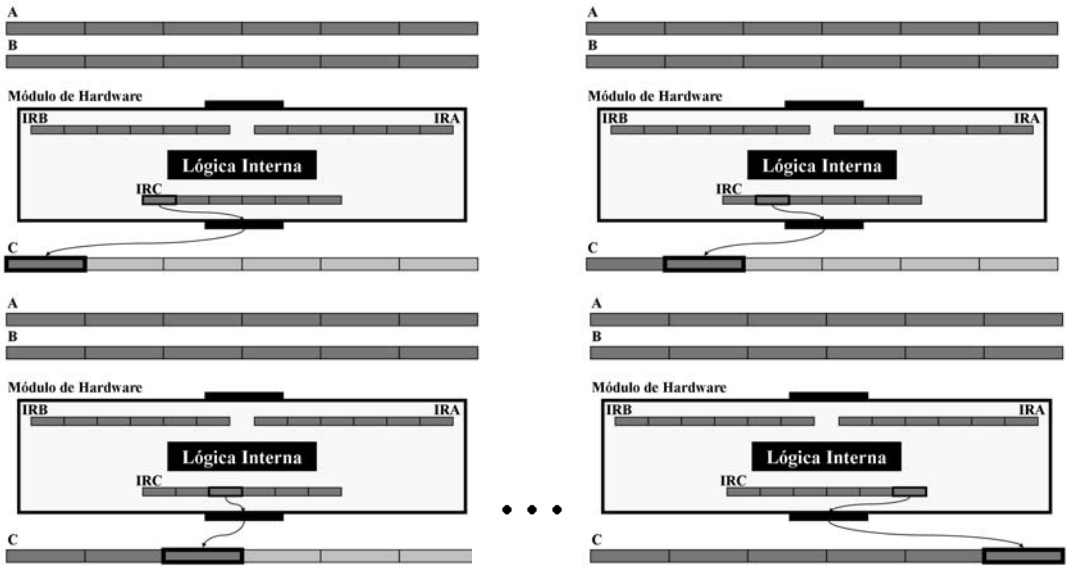


Figura A.6: Leitura de Resultados dos Registradores dos Periféricos

Referências Bibliográficas

[1] W. Diffie and M.E. Hellman, "New Directions in Cryptography," In IEEE Transactions on Information Theory, vol. IT-22, no. 6, pp. 644-654, 1976.

[2] V.S. Miller, "Use of Elliptic Curves in Cryptography," In Proc. Crypto'85, pp. 417-426, 1986.

[3] N. Koblitz, "Elliptic Curve Cryptosystems," Math. Computation, vol. 48, pp. 203-209, 1987.

[4] T. Itoh and S. Tsujii, "A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) using Normal Bases," In Information and Computation, vol. 78, no. 3, pp. 171-177, 1988.

[5] D.W. Ash, I.F. Blake, and S.A. Vanstone, "Low Complexity Normal Bases," Discrete Applied Math., vol. 25, pp. 191-210, 1989.

[6] A. Menezes, Elliptic Curve Public Key Cryptosystems: Kluwer Academic Publishers, 1993.

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

[8] B. Schneier, Applied Cryptography - Protocols, Algorithms and Source Code in C: John Wiley & Sons, Inc., 2nd edition, 1996.

[9] M. Rosing, Implementing Elliptic Curve Cryptography: Manning Publications, 1998.

[10] J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over GF(2^m) without pre-computation," In Proc. of the I Workshop on Cryptographic Hardware and Embedded Systems (CHES), LNCS 1717, pages 316-327, January 1999.

[11] IEEE Std. 1363-2000, IEEE Standard Specification for Public-Key Cryptography: IEEE, January 2000.

[12] J. López and R. Dahab, "High-speed software multiplication in F(2^m)," In INDOCRYPT 2000, LNCS 1977, pp. 203-212, 2000.

[13] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor," In Proc. of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 68-76, 2000.

[14] A. Reyhani-Masoleh and M.A. Hasan. "Fast Normal Basis Multiplication Using General Purpose Processors," In Selected Areas in Cryptography (SAC 2001), LNCS 2259, pp. 230-244, 2001.

[15] P. Ning and Y. Yin, "Efficient Software Implementation for Finite Field Multiplication in Normal Basis," In Information and Communications Security 2001. LNCS 2229, pp. 177-189, 2001.

[16] J. Wolkerstorfer and W. Bauer, "A PCI-Card for Accelerating Elliptic Curve Cryptography," Proceedings of Austrochip 2002, 2002.

[17] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel, "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^m)," In CHES 2002, LNCS 2523, pp. 381-399, 2002.

[18] N. Gura, S. Shantz, H. Eberle, D. Finchelstein, S. Gupta, V. Gupta, and D. Stebila, "An End-to-End Systems Approach to Elliptic Curve Cryptography," In CHES 2002, LNCS 2523, pp. 349-365, 2002.

[19] P. Leong and I. Leung, "A Microcoded Elliptic Curve Processor using FPGA Technology," In IEEE Transactions on VLSI, vol. 10, no. 5, pp. 550-559, 2002.

[20] T. Kerins, E.M. Popovici, W.P. Marnane, and P. Fitzpatrick, "Fully Parameterizable Elliptic Curve Cryptography Processor over GF(2^m)," In FPL 2002, LNCS2438, pp. 750-759, 2002.

[21] Altera Corporation, Avalon Bus Specification - Reference Manual: Altera Corporation, July 2003.

[22] A. Reyhani-Masoleh and M.A. Hasan, "Fast Normal Basis Multiplication Using General Purpose Processors," In IEEE Transactions on Computers, vol. 52, no. 11, pp. 1379-1390, November 2003.

[23] F.R. Henríquez, N.A. Saqib, and A.D. Pérez, "Ultra Fast Parallel Implementation of Elliptic Curve Point Multiplication over GF(2^m)," Preprint submitted to Elsevier Science, October 20, 2003.

[24] L.C. Washington, Elliptic Curves: Number Theory and Cryptography: Chapman & Hall/CRC, 2003.

[25] W. Mao, Modern Cryptography: Theory and Practice: Pentice Hall, 1st edition, July 2003.

[26] Altera Corporation, Nios II Processor Reference Handbook: Altera Corporation, 1st edition, May 2004.

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

[28] J. Großschaedl and E. Savas, "Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m)," In CHES 2004, LNCS 3156, pp. 133-147, 2004.

[29] J. López, "A note on Multiplication over GF(2^m) using Gaussian Normal Bases," Manuscript, 2004.

[30] J. Lutz and A. Hasan, "High Performance FPGA based Elliptic Curve Cryptographic Co-Processor," In Proc. of the International Conference on Information Technology: Coding and Computing (ITCC 2004), vol. 2, pp. 486-492, 2004.

[31] N. Telle, W. Luk, and R.C.C. Cheung, "Customizing Hardware Designs for Elliptic Curve Cryptography," In SAMOS 2004, LNCS 3133, pp. 274-283, 2004.

[32] S. Tillich and J. Großschaedl, "A Simple Architectural Enhancement for Fast and Flexible Elliptic Curve Cryptography over Binary Finite Fields GF(2^m)," In Proceedings of 9th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC 2004), pp. 282-295, 2004.

[33] Altera Corporation, Avalon Interface Specification: Altera Corporation, April 2005.

[34] Altera Corporation, Quartus II Version 5.0 Handbook: Altera Corporation, May 2005.

[35] M. Juliato, G. Araujo, J. López, R. Dahab, "A Custom Instruction Approach for Hardware and Software Implementations of Finite Field Arithmetic over $\mathbb{F}_{2^{163}}$ using Gaussian Normal Bases," In Proc. of the IEEE 2005 International Conference on Field Programmable Technology (FPT'05), December 2005.

[36] Altera Corporation, Nios Development Board Reference Manual, Stratix II Edition: Altera Corporation, July 2005.

[37] C. Pühringer, and J. Wolkerstorfer, "High Speed Elliptic Curve Cryptography Processor for GF(p)," In Proc. Austrochip 2005 Mikroelektronik Tagung, pp. 153-160, 2005

[38] C. Shu, K. Gaj, T. El-Ghazawi, "Low Latency Elliptic Curve Cryptography Accelerators for NIST Curves on Binary Fields," Proc. IEEE 2005 Conference on Field Programmable Technology, FPT'05, Singapore, December 2005.

[39] R.C.C. Cheung, W. Luk, and P.Y.K. Cheung, "Reconfigurable Elliptic Curve Cryptosystems on a Chip," In Proc. of IEEE Design Automation and Test in Europe (DATE 2005), vol. 1, pp. 24-29, 2005.

[40] V. Trujillo, J. Velasco, and J. López, "Design of an Elliptic Curve Cryptoprocessor over GF(2^{163})," In XI Iberchip, March 2005.

[41] Altera Corporation, Stratix II Device Handbook: Altera Corporation, April 2006.

[42] Altera Corporation, Nios II Development Kit Getting Started User Guide: Altera Corporation, May 2006.

[43] B. Ansari, and A. Hasan, "High Performance Architecture of Elliptic Curve Scalar Multiplication," Technical Report CACR 2006-01, University of Waterloo, 2006.

[44] D.R. Stinson, Cryptography Theory and Practice: Chapman & Hall/CRC, 3rd edition, 2006.

[45] M. Juliato, G. Araujo, J. López, R. Dahab, "A Custom Instruction Approach for Hardware and Software Implementations of Finite Field Arithmetic over $\mathbb{F}_{2^{163}}$ using Gaussian Normal Bases," (Extended version), In the Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology, 2006. (To appear)

[46] R. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren, Handbook of Elliptic and Hyperelliptic Curve Cryptography: Chapman & Hall/CRC, 2006.

[47] R. Dahab, D. Hankerson, F. Hu, M. Long, J. López, and Alfred Menezes, "Software Multiplication using Gaussian Normal Bases," In IEEE Trans. Computers, 2006. (To appear)