UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA

Paulo Miguel Rodrigues Rocha

Comparação entre solução dedicada e operador nativo à linguagem VHDL como estratégias de otimização em hardware para operações aritméticas modulares

Florianópolis
2024

Paulo Miguel Rodrigues Rocha

Comparação entre solução dedicada e operador nativo à linguagem VHDL como estratégias de otimização em hardware para operações aritméticas modulares

Trabalho de Conclusão de Curso do Curso de Graduação em Engenharia Eletrônica do Centro Tecnológico da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Engenharia Eletrônica.
Orientador: Prof. Héctor Pettenghi Roldán, Dr.
Coorientador: Gabriel Bruno Monteiro Fernandes, MSc

Florianópolis
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Rocha, Paulo Miguel Rodrigues
Comparação entre solução dedicada e operador nativo à linguagem VHDL como estratégias de otimização em hardware para operações aritméticas modulares / Paulo Miguel Rodrigues Rocha ; orientador, Héctor Pettenghi Roldán, coorientador, Gabriel Bruno Monteiro Fernandes, 2024.
54 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Eletrônica, Florianópolis, 2024.

Inclui referências.

1. Engenharia Eletrônica. 2. otimização em hardware. 3. operações aritméticas modulares. 4. criptografia de hardware. 5. ASIC. I. Roldán, Héctor Pettenghi. II. Fernandes, Gabriel Bruno Monteiro. III. Universidade Federal de Santa Catarina. Graduação em Engenharia Eletrônica. IV. Título.

Paulo Miguel Rodrigues Rocha

Comparação entre solução dedicada e operador nativo à linguagem VHDL como estratégias de otimização em hardware para operações aritméticas modulares

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Engenharia Eletrônica" e aprovado em sua forma final pelo Curso de Graduação em Engenharia Eletrônica.

Florianópolis, 09 de Julho de 2024.

Documento assinado digitalmente
Daniela Ota Hisayasu Suzuki
Data: 16/07/2024 18:33:41-0300
CPF: ***.397.389-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Daniela Ota Hisayasu Suzuki, Dra.
Coordenadora do Curso

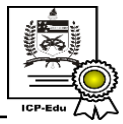
Banca Examinadora

Documento assinado digitalmente
Hector Pettenghi Roldan
Data: 16/07/2024 12:24:57-0300
CPF: ***.054.601-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Héctor Pettenghi Roldán, Dr.
Orientador:
UFSC/CTC/EEL

Documento assinado digitalmente
Cristina Meinhardt
Data: 16/07/2024 18:02:52-0300
CPF: ***.850.380-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Cristina Meinhardt, Dra.
Avaliadora
UFSC/CTC/INE



Documento assinado digitalmente
Raimes Moraes
Data: 15/07/2024 18:46:31-0300
CPF: ***.997.488-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Raimes Moraes, Dr.
Avaliador
UFSC/CTC/EEL

Este trabalho é dedicado aos meus pais, Fernando (in memorian) e Inês, que sempre prezaram pela minha educação, à minha filha Dandara e minha companheira Anícia.

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus pela vida, sabedoria e pela oportunidade a mim concedida.

Agradeço aos meus pais Fernando (in memoriam) e Inês por todo o apoio que prestaram durante minha trajetória e pelas vezes que me deram forças para continuar minha jornada.

Agradeço aos meus irmãos Rilda, Alina, Anabela, Fernando, Nila, sobrinhos e toda família Rocha pelos incentivos que me fizeram ao longos destes anos. Sem vocês nada disto seria possível.

Agradeço à minha filha Dandara Rocha que mesmo pequena me incentiva todos os dias a não desistir, mesmo nos dias mais difíceis.

Agradeço à minha companheira Anícia pelo amor e carinho diário.

Agradeço na figura do meu orientador Héctor Pettenghi Roldán por todo apoio e paciência e ensinamentos durante a execução deste trabalho.

Agradeço ao meu coorientador Gabriel Bruno Monteiro Fernandes pelas correções e direcionamento durante todo o processo.

Por fim, deixo o meu muito obrigado a todos os amigos que fiz no Brasil e que levarei para vida toda.

RESUMO

A pesquisa propôs uma análise comparativa entre estratégias de otimização em hardware para operações aritméticas modulares e o operador nativo da linguagem *VHSIC Hardware Description Language* (VHDL). Os objetivos principais visaram explorar métodos de aprimoramento lógico, com foco na implementação em *Application Specific Integrated Circuit* (ASIC). O estudo abordou a implementação em ASIC do operador modular nativo à linguagem VHDL para operações de soma e multiplicação, promovendo uma comparação direta com soluções otimizadas presentes na literatura. Os resultados mostraram que a solução dedicada superou o operador modular nativo, com ganhos médios de 2,81% usando o módulo no formato $2^n + 1$, 15% no formato $2^n - 1$ e 5,54% no formato $2^n + k$ na soma modular. Na multiplicação modular, os ganhos foram de 17,54% usando o módulo no formato $2^n - 1$, 15,7% no formato $2^n + 1$ e 28,86% no formato $2^n \pm k$. Foi investigada a sensibilidade do operador módulo a diferentes formatos de módulos, incluindo 2^n, $2^n \pm 1$ e $2^n \pm k$. A partir disso, propôs-se otimizações explorando o uso de módulos do formato $2^n \pm 1$ na solução dedicada para o desenvolvimento de uma aplicação no contexto da criptografia.

Palavras-chave: otimização em hardware, operações aritméticas modulares, VHDL, módulos auxiliares, criptografia de hardware.

ABSTRACT

The research proposed a comparative analysis between hardware optimization strategies for modular arithmetic operations and the native operator of the VHSIC Hardware Description Language (VHDL) language. The main objectives aimed at exploring logical improvement methods, focusing on implementation in Application Specific Integrated Circuit (ASIC). The study addressed the ASIC implementation of the VHDL language native modular operator for addition and multiplication operations, promoting a direct comparison with optimized solutions present in the literature. The results indicated that the dedicated solution outperformed the native modular operator, with average gains of $2,81\%$ using the 2^n+1 format, 15% using the 2^n-1 format, and $5,54\%$ using the 2^n+k format in modular addition. In modular multiplication, gains were $17,54\%$ using the 2^n-1 format, $15,7\%$ in the 2^n+1 format, and $28,86\%$ in the $2^n \pm k$ format. The sensitivity of the modulo operator to different module formats was investigated, including 2^n, $2^n \pm 1$ and $2^n \pm k$. Based on this, optimizations were proposed exploring the use of modules in the $2^n \pm 1$ format in the dedicated solution for the development of an application in the context of cryptography.

Keywords: hardware optimization, modular arithmetic operations, VHDL, auxiliary modules, hardware cryptography.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADP	*Area-Delay-Product*
AES	*Advanced Encryption Standard*
ASIC	*Application Specific Integrated Circuit*
CPA	*Carry-Propagate Adder*
CSA	*Carry-Save Adder*
DES	*Data Encryption Standard*
DH	*Diffie-Hellman*
DSA	*Digital Signature Algorithm*
DSP	*Digital Signal Processing*
EAC	*End-Around Carry*
ECC	*Elliptic Curve Cryptography*
ECDH	*Elliptic Curve Diffie-Hellman*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
FIR	*Finite Impulse Response*
FPGA	*Field Programmable Gate Array*
IEAC	*Inverted End-Around Carry*
RNS	*Residue Number System*
ROM	*Read-Only Memory*
RSA	*Rivest Shamir Adleman*
TLS	*Transport Layer Security*
VHDL	*VHSIC Hardware Description Language*

SUMÁRIO

1 INTRODUÇÃO

Historicamente, no campo do projeto de hardware, esforços significativos têm sido feitos de modo a alcançar circuitos aritméticos cada vez mais eficientes. Uma modalidade de aritmética na área da computação que vêm ganhando importância crescente nos últimos anos é a aritmética modular. Este tipo de aritmética tem uma ampla gama de aplicações. Entre elas a implementação de diversos algoritmos de criptografia. Por exemplo, protocolos de troca de chaves como *Diffie-Hellman* (DH) (BOREALE; BUSCEMI, 2003), *Rivest Shamir Adleman* (RSA) (SHIEH *et al.*, 2008) e *Digital Signature Algorithm* (DSA) (MÖLLER; RUPP, 2008) requerem uso eficiente de exponenciação modular. Outra utilização está em filtros digitais (CARDARILLI; NANNARELLI; RE, 2000) que são um dos grandes campos de aplicação de processamento digital de sinal e que recorre a um número significativo de operações aritméticas. Por exemplo o filtro *Finite Impulse Response* (FIR) implementados como operadores dedicados em hardware podem ter um custo reduzido aplicando técnicas elementares de projeto.

Adicionalmente, a necessidade de operar de forma eficiente e utilizar módulos de centenas ou mesmo milhares de bits eleva consideravelmente o nível de complexidade e justifica a necessidade de tais implementações otimizadas. A aritmética modular também nos permite realizar multiplicações mais rapidamente do que algoritmos convencionais. Devido a isto, existem muitas aplicações em que a aritmética modular é realizada em módulos menores, variando de 10 a 32 bits. Uma dessas aplicações é o *Residue Number System* (RNS) (PLANTARD, 2021), que utiliza múltiplas operações modulares menores para substituir operações maiores.

Linguagens de descrição de hardware como VHDL e Verilog geralmente apresentam funções prontas para uso para implementar módulo e multiplicação. No caso da linguagem VHDL o operador "mod ()"pode ser utilizado por qualquer projetista para uma implementação rápida e simplificada de cálculos modulares. Porém, as tecnologias *Field Programmable Gate Array* (FPGA) e ASIC não sintetizam a função "mod ()"de forma otimizada, diminuindo significativamente o desempenho do sistema.

Como alternativa, conjuntos modulares dedicados na forma 2^n, $2^n \pm 1$ e $2^n \pm k$, onde n equivale ao número de bits por canal, têm sido explorados na literatura. Estes conjuntos de módulos realizam implementações modulares otimizadas, reduzindo fortemente o caminho crítico em comparação com os multiplicadores convencionais. Iremos analisar uma possível existência de sensitividade ao tipo de módulo utilizado nesses circuitos, sendo eles os formatos: 2^n, $2^n \pm 1$ e $2^n \pm k$. Os tipos 2^n e $2^n \pm 1$ apresentam um desempenho superior a $2^n \pm k$. Ainda, o atraso de $2^n \pm k$ é diretamente dependente do número de 1's na representação binária de k (PARHAMI, 2009).

1.1 METODOLOGIA

Neste trabalho faremos o uso da linguagem de descrição de Hardware VHDL. Para garantir uma comparação justa, utilizaremos scripts automatizados para gerar sínteses lógicas dos circuitos-alvo, aplicando parâmetros e configurações idênticos a todos os circuitos. O operador nativo "mod ()"será comparado com circuitos gerados automaticamente por uma ferramenta de software desenvolvida em (FERNANDES, 2021) a modo de melhorar uma otimização na implementação. A título de comparação, dividiremos as operações de multiplicação em tipos de módulo. As comparações serão realizadas após as sínteses lógicas serem feitas, considerando a área, o atraso e o produto entre área e atraso dos circuitos, como métricas de avaliação.

1.2 MOTIVAÇÃO

A motivação deste trabalho é explorar e comparar diferentes abordagens para a implementação de operadores modulares, buscando eficiência com o foco especial na operação de multiplicação modular. Os projetistas de hardware geralmente utilizam a implementação do operador mod nativo por ser mais simples. Multiplicadores modulares dedicados utilizam uma estratégia de colocação de compressores semelhante à abordagem de Wallace, alocando o máximo número de compressores por iteração. Será explorado como é feito implementação do operador nativo mod e a solução dedicada. Também será explorada a sensitividade ao tipo de módulo apresentado no estado de arte. Módulos do tipo $2^n \pm k$ possuem desempenho menor que os conjuntos de módulos no formato 2^n e $2^n \pm 1$. Apesar disso, os módulos do tipo $2^n \pm k$ podem ser convertidos módulos auxiliares com melhor desempenho na operação e que será corrigido numa etapa final simples e aritmética. O conceito de módulos auxiliares será um dos temas abordados neste trabalho.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Neste trabalho de conclusão de curso serão explorados métodos de otimização lógica para operações aritméticas modulares, visando alternativas ao operador nativo na linguagem VHDL, com o intuito de implementação em ASIC.

1.3.2 Objetivos Específicos

1. Explorar como é realizada a implementação em ASIC do operador nativo à linguagem VHDL para as operações indicadas, em comparação com soluções específicas de otimização presentes na literatura.

2. Investigar a presença de sensitividade do operador módulo aos diferentes tipos de módulos 2^n, $2^n \pm 1$ e $2^n \pm k$.

3. Buscar otimizações para operações modulares de soma (+), multiplicação (*) e conversão modular explorando módulos auxiliares.

4. Estudar, compreender e propor uma aplicação otimizada voltada à criptografia com a exploração dos módulos auxiliares para operações aritméticas sequenciais.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está dividido em cinco capítulos, organizados da seguinte forma:

O capítulo 1 apresenta a introdução buscando trazer a contextualização, a motivação, a metodologia e os objetivos do trabalho.

O capítulo 2 traz a fundamentação teórica, abordando a teoria do operador modular nativo e da solução dedicada.

O capítulo 3 foca na verificação dos resultados em ASIC do operador nativo mod em VHDL e da solução dedicada. É explicada a proposta de módulos auxiliares visando a sensitividade modular dos módulos $2^n \pm 1$ comparando com o resto dos módulos.

O capítulo 4 aborda os fundamentos da criptografia necessários para o entendimento deste trabalho e apresenta uma proposta de aplicação da sensitividade modular.

O capítulo 5 finaliza o trabalho com as considerações finais e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO DAS OPERAÇÕES MODULARES

A aritmética modular, originada nos estudos pioneiros de Euler e posteriormente aprimorada por Gauss (GAUSS, 1986). A execução de operações modulares envolve a recodificação dos operandos com base nos restos resultantes da divisão pelos seus módulos correspondentes. Um número inteiro X qualquer, em que elementos m_i são denominados módulos, são representados por outro conjunto de X elementos (x_1, x_2,..., x_i), onde X_i é o resto da divisão de X por cada m_i, denotado como:

$$x_i = X \ mod \ m_i = |X|_{m_i}, \quad 0 \leq x_i \leq m_i \tag{2.1}$$

Para exemplificação da Equação 2.1 é mostrado um exemplo de cálculo modular. Assumindo um número inteiro aleatório $X = 23$, calculando 23 $\mod 8$ temos $|23|_8 = 7$. Ou seja, fazendo a divisão de 23 por 8 temos o resto 7. É possível notar uma relação de congruência. Essa relação conecta dois números inteiros, X e Y, quando a diferença entre eles é exatamente divisível pelo módulo m. Em outras palavras, X e Y são considerados congruentes módulo m. Matematicamente expressa-se pela Equação 2.2, onde m é definido como a base ou módulo da operação.

$$X \equiv Y \ mod \ m \tag{2.2}$$

Como exemplo, temos que $29 \equiv 9 \mod 10$ e $-2 \equiv 8 \mod 10$. A partir dessa relação de congruência, novas possibilidades se abrem no campo das operações matemáticas elementares. É possível realizar adição, subtração e multiplicação com números modulares de forma comutativa e associativa. Por exemplo, para a adição, $29+(-2) \equiv 9+8 \mod 10$, resultando em $27 \equiv 7 \mod 10$. Da mesma forma, para a multiplicação, $29 \times (-2) \equiv 9 \times 8 \mod 10$, resultando em $232 \equiv 2 \mod 10$. Essas operações mostram como podemos simplificar cálculos complexos e desenvolver sistemas mais eficientes usando a aritmética modular.

2.1 IMPLEMENTAÇÃO TEÓRICA DAS OPERAÇÕES MODULARES

Nesta subseção serão apresentados e detalhados os principais conceitos necessários para a compreensão e implementação das operações modulares. As operações modulares que serão exploradas são: conversão direta e as operações aritméticas de soma e multiplicação.

2.1.1 Conversão Direta

Neste apartado um número inteiro de $4n$ bits $X = \{x_{(4n-1)}, ..., x_1, x_0\}$, pode ser expressado em notação binária:

$$X = \sum_{i=1}^{4n-1} 2^i x_i = 2^{3n} N_3 + 2^{2n} N_2 + 2^n N_1 + N_0 \tag{2.3}$$

onde tem-se os vetores $N_3 = \{x_{(4n-1)}, \ldots, x_{(3n+1)}, x_{(3n)}\}$, $N_2 = \{x_{(3n-1)}, \ldots, x_{(2n+1)}, x_{(2n)}\}$, $N_1 = \{x_{(2n-1)}, \ldots, x_{(n+1)}, x_{(n)}\}$ e $N_0 = \{x_{(n-1)}, \ldots, x_{(1)}, x_{(0)}\}$.

CASO 2^n

O método mais básico é o conversor que utiliza o módulo 2^n. O valor $|X|_{2^n}$ é o resultado do resto da divisão de X por 2^n, que pode ser obtido através do truncamento do valor de X. Para o cálculo do valor $|X|_{2^n}$, somente os n bits menos significativos são considerados, enquanto o restante dos bits é desprezado. Esse módulo proporciona uma conversão direta sem a necessidade de hardware como apresentado na Figura 2.1a, sendo representada matematicamente por:

$$|X|_{2^n} = |2^{3n}|_{2^n} N_3 + |2^{2n}|_{2^n} N_2 + 2^n N_1 + N_0 = \{x_{(n-1)}, \ldots, x_1, x_0\} \tag{2.4}$$

CASO $2^n - 1$

Devido a que $|2^n|_{2^n-1} = 1$, podemos expressar a Equação 2.3 como:

$$|X|_{2^n-1} = |N_3 + N_2 + N_1 + N_0|_{2^n-1} = |N_3 + |N_2 + N_1 + N_0|_{2^n-1}|_{2^n-1} \tag{2.5}$$

Para implementar o módulo $2^n - 1$, utiliza-se a Equação 2.5. O diagrama de blocos do módulo $2^n - 1$ está ilustrado na Figura 2.2a. Cada *Carry-Save Adder* (CSA) soma três termos, resultando nos vetores $C = c_n, \ldots, c_2, c_1$ e Soma $S = s_{n-1}, \ldots, s_1, s_0$. Supondo que $|2^n c_n|_{2^n-1} = c_n$, podemos realocar o bit c_n na posição 2^0 ou seja, *End-Around Carry* (EAC). Por fim, a soma na última etapa é realizada por um *Carry-Propagate Adder* (CPA), que também utiliza um EAC. Ambas estruturas $CSA + EAC$ e $CSA + IEAC$ para $n = 5$ encontram-se na Figura 2.3 e 2.4 respetivamente.

CASO $2^n + 1$

Devido a que $|2^n|_{2^n+1}$ = -1, podemos expressar a Equação 2.3 como:

$$|X|_{2^n+1} = |N_3 - N_2 + N_1 - N_0|_{2^n+1} = |-N_3 + |N_2 - N_1 + N_0|_{2^n+1}|_{2^n+1} \tag{2.6}$$

Para implementar o canal $m_3 = \{2^n + 1\}$ a partir da Equação 2.6, utiliza-se o diagrama de blocos mostrado na Figura 2.2b. Cada CSA soma novamente três termos, gerando

os barramentos de *carry* $C = c_n, \ldots, c_2, c_1$ e de soma $S = s_{n-1}, \ldots, s_1, s_0$. Considerando que:

$$|2^n c_n|_{2^n+1} = ||2^n|_{2^n+1} c_n|_{2^n+1} = |-c_n|_{2^n+1} = |COR_{level-j} + \overline{c_n}|_{2^n+1}, \tag{2.7}$$

o bit c_n pode ser recolocado na posição 2^0 de forma complementada (ou seja, utilizando a operação de *Inverted End-Around Carry* (IEAC) adicionando um fator de correção $COR_{level-j}$, onde j define o nível do CSA com IEAC associado ao fator de correção. $COR_{level-j}$ pode ser calculado a partir da Equação:

$$|COR_{level-j} + \overline{c_n}|_{2^n+1} = 0 \tag{2.8}$$

quando $c_n = 0$ (ou seja, $\overline{c_n} = 1$). Desta forma, $COR_{level-j} = 2^n$ por nível de CSA-IEAC. Os termos negativos $-N_0$ e $-N_2$ seguem a mesma regra:

$$|-N_i|_{2^n+1} = |COR_{N_i} + \overline{N_i}|_{2^n+1}, \tag{2.9}$$

onde i define qual *array* de N_i está associado ao fator de correção. O fator de correção COR_{N_i} pode ser calculado a partir da equação:

$$|COR_{N_i} + \overline{N_i}|_{2^n+1} = 0 \tag{2.10}$$

quando $N_i = 0$ (ou seja, $\overline{N_i} = 2^n - 1$). Desta forma, $COR_{N_i} = 2$ para cada $-N_i$. O fator de correção final (COR) consiste na soma modular de todas as correções parciais

$$COR = \left| \sum_{j=1,2,3,4} COR_{level-j} + \sum_{i=0,2} COR_{N_i} \right|_{m_3}. \tag{2.11}$$

Ambas estruturas $CSA+EAC$ e $CSA+IEAC$ para $n = 5$ encontram-se na Figura 2.3 e 2.4 respetivamente.

CASO $2^n \pm k$

Um conversor de módulo $2^n \pm k$ transforma um inteiro X, com entradas de m-bits $x_{m-1}, x_{m-2}, ..., x_0$ sendo $m = 4n$, em uma palavra resíduo R de um módulo $2^n + k$ saídas de bits $(r_{a-1}, r_{a-2}, ..., r_0)$, com $a = \log_2((2^n \pm k)$, ou seja, $a = n$ e $a = n + 1$, para módulo $2^n - k$ e $2^n + k$, respectivamente. O valor de entrada X é convertido para

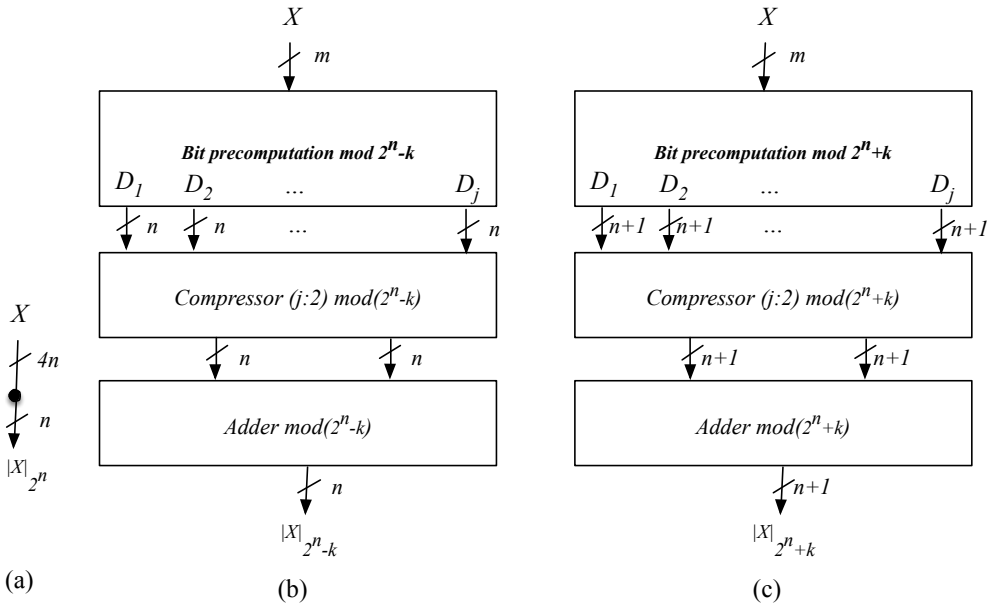
$$|X|_{2^n \pm k} = \sum_{j=0}^{a-1} 2^j \cdot r_j = |\sum_{j=0}^{m-1} 2^j \cdot r_j|_{2^n \pm k} \tag{2.12}$$

Uma técnica que evita o uso de uma operação de divisão para calcular o resto pode ser expresso como na Equação abaixo:

$$|X|_{2^n \pm k} = |\sum_{j=0}^{m-1} |2^j|_{2^n \pm k} \cdot x_j|_{2^n \pm k} \tag{2.13}$$

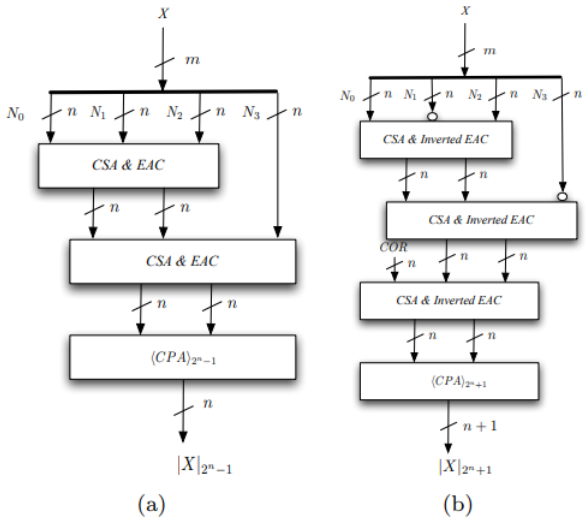
O diagrama de blocos está ilustrado na Figura 2.1b para $2^n - k$ e 2.1c para $2^n + k$. As entradas são pré-computadas e após isso faz-se uma compressão modular e soma.

Figura 2.1 – Conversão Direta dos módulos (a) 2^n, (b) $2^n - k$ e (c) $2^n + k$



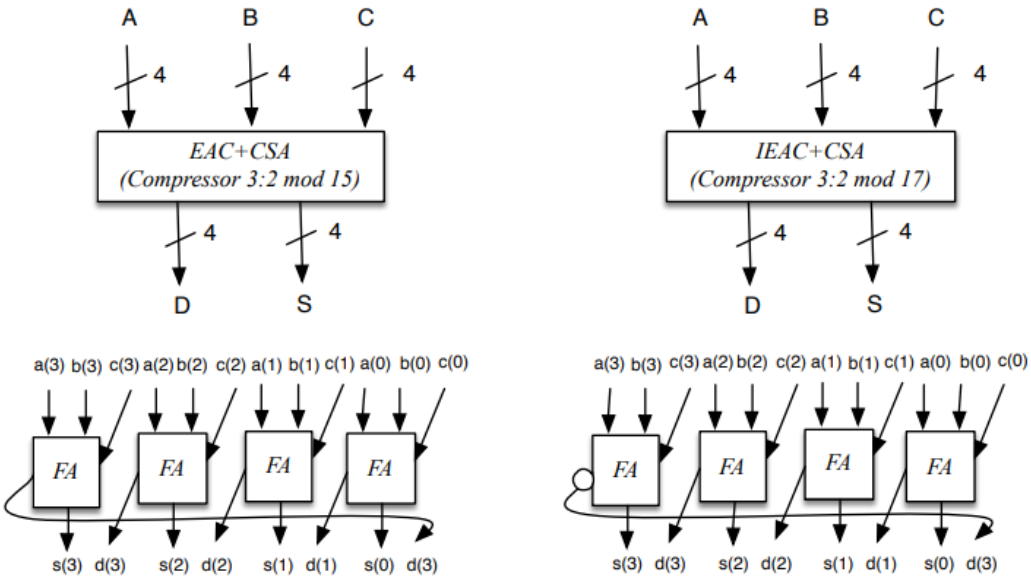
Fonte: Elaborado pelo autor

Figura 2.2 – Conversão Direta dos módulos (a) $2^n - 1$ e (b) $2^n + 1$



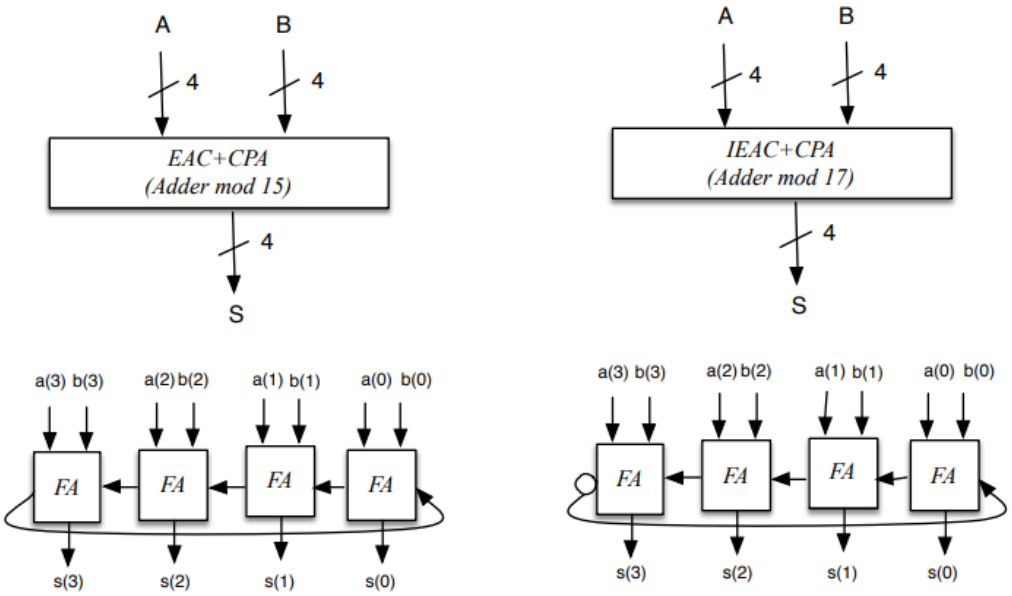
Fonte: Elaborado pelo autor

Figura 2.3 – Estruturas CSA + EAC e CSA + IEAC para módulos 15 e 17



Fonte: Elaborado pelo orientador

Figura 2.4 – Estruturas CPA + EAC e CPA + IEAC para módulos 15 e 17



Fonte: Elaborado pelo orientador

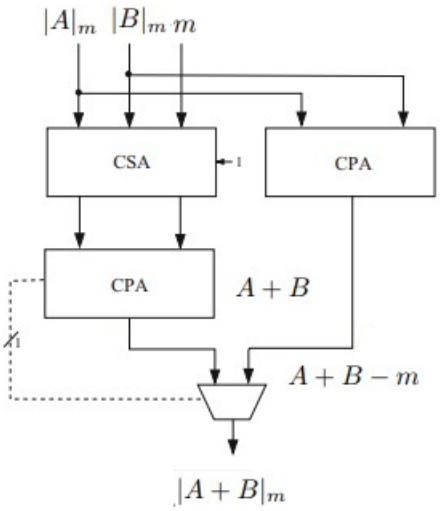
2.1.2 Soma Modular

A operação de soma modular envolve adicionar dois números, denotados como A e B, em que $A,\ B \in [0, m)$ com o módulo $|X|$. Assim é definida como:

$$|X|_m = |A + B|_m \begin{cases} A + B & \text{se } A + B < m \\ A + B - m, & \text{caso contrário} \end{cases} \tag{2.14}$$

A estrutura paralela apresentada na Figura 2.5 processa ambas as operações simultaneamente, utilizando um multiplexador 2:1 para selecionar o resultado correto, controlado pelo sinal da operação $A+B-m$. Para os módulos 2^n, a implementação é direta, bastando um somador binário CPA, ignorando as saídas de carry. No caso do módulo $2^n - 1$, a adição é realizada observando que $|2^n|_{2^n-1} = 2^n - (2^n - 1) = 1$. Assim, a soma de dois operandos de até n bits resulta em $n + 1$ bits, onde o n-ésimo bit, o *carry-out* é realimentado no somador como EAC como mostrado na Figura 2.4. De maneira análoga, $|2^n|_{2^n+1}$ $= 2^n - (2^n + 1) = -1$, e a adição para módulo $2^n + 1$ é realizada com a realimentação invertida, conhecida como IEAC. O EAC e o IEAC são usados para ajustar somas modulares, garantindo resultados dentro do intervalo desejado como mostrado na Figura 2.4. No entanto, a estrutura $CPA + EAC$ pode sofrer com propagação excessiva de *carry* e ajustes repetidos, aumentando a latência. A estrutura $CPA + IEAC$, por sua vez, pode introduzir complexidade adicional e ser sensível a pequenos erros de cálculo. Ambas as combinações têm instabilidades específicas que devem ser consideradas ao implementar operações modulares em hardware e devido a isto, a soma modular passa a ser feita usando a abordagem paralela da Figura 2.5.

Figura 2.5 – Estrutura Paralela Soma Modular



Fonte: Elaborado pelo autor

2.1.3 Multiplicação Modular

CASO 2^n

Em situações em que temos casos de 2^n, a multiplicação modular pode ser realizada truncando os produtos parciais cujo peso seja maior do que 2^n. Isso ocorre porque $|2^n|_{2^n} = 0$ durante o processo de pré-computação. A Figura 2.6 ilustra como ocorre a multiplicação para este módulo. Os pontos pretos correspondem aos produtos parciais da operação.

CASO $2^n - 1$

Para os casos em que temos $2^n - 1$, a multiplicação modular pode ser obtida ao redistribuir as posições dos produtos parciais com peso maior que 2. Isso é possível porque $|2^n + i|_{2^n-1} = 2^i$ para $i \geq n$. Os produtos parciais são rotacionados aplicando método End around Carry (EAC). A Figura 2.6 apresenta um exemplo de cálculo para este módulo $(n = 4)$.

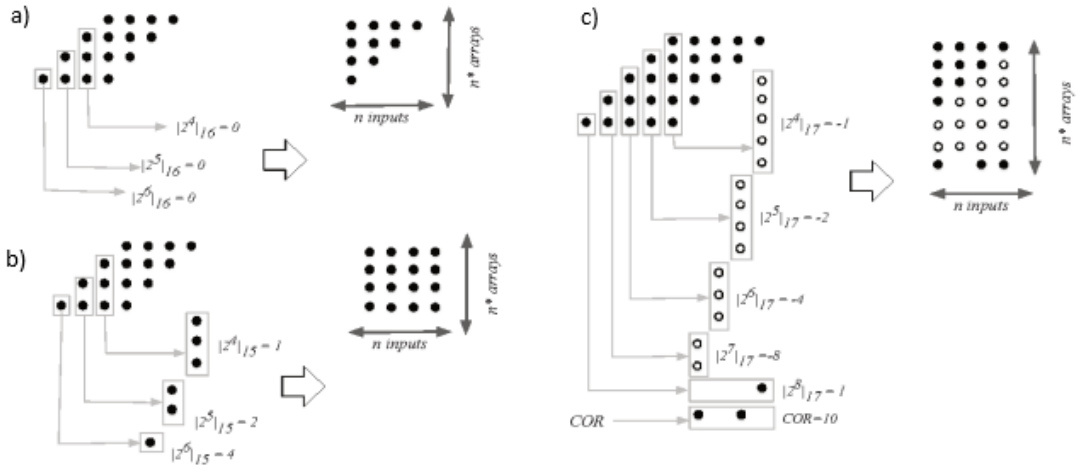
CASO $2^n + 1$

No caso de $2^n + 1$, a multiplicação modular pode ser realizada ao redistribuir as posições dos produtos parciais cujo peso seja maior do que 2^n, levando em consideração que $|2^n + i|_{2^n+1} = -2^i$ para $i \geq n$. Aplica-se Inverted EAC aos produtos parciais e é importante destacar que na operação de Inverted EAC existe um fator de correção devido ao aditivo inverso como mostrado na Figura 2.6 para $(n = 4)$.

CASO $2^n \pm k$

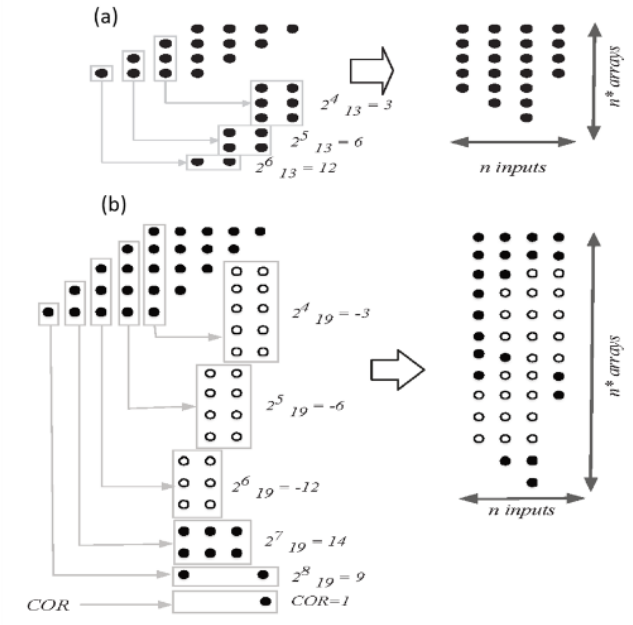
A operação de multiplicação de $2^n \pm k$ é ilustrada com $(k = 3)$ e $(n = 4)$. A Figura 2.7 mostra a pré-computação dos produtos parciais para os módulos 13 $(2^4 + 3)$ e 19 $(2^4 - 3)$. Após a exibição da matriz de $n \times (n*)$ bits, é possível realizar a soma modular de $n*$ vetores de n bits. É importante notar que, na Figura 2.7, o tamanho do vetor foi diminuído para $(n = 4)$ para simplificar a representação. A soma dos vetores e as respetivas contribuições dos *carrys* são somadas e armazenadas em uma memória somente de leitura *Read-Only Memory* (ROM). A soma final $mod(2^4 + 3)$ de três vetores é realizada para obter o vetor resultante da multiplicação modular.

Figura 2.6 – Exemplo de multiplicação para (a) 2^n, (b) $2^n - 1$ e (c) $2^n + 1$ ($n = 4$)



Fonte: Elaborado pelo orientador

Figura 2.7 – Exemplo de multiplicação para (a) $2^n - k$ e (b) $2^n + k$ ($k = 3$)



Fonte: Elaborado pelo autor

2.2 IMPLEMENTAÇÃO EM ASIC DAS OPERAÇÕES MODULARES

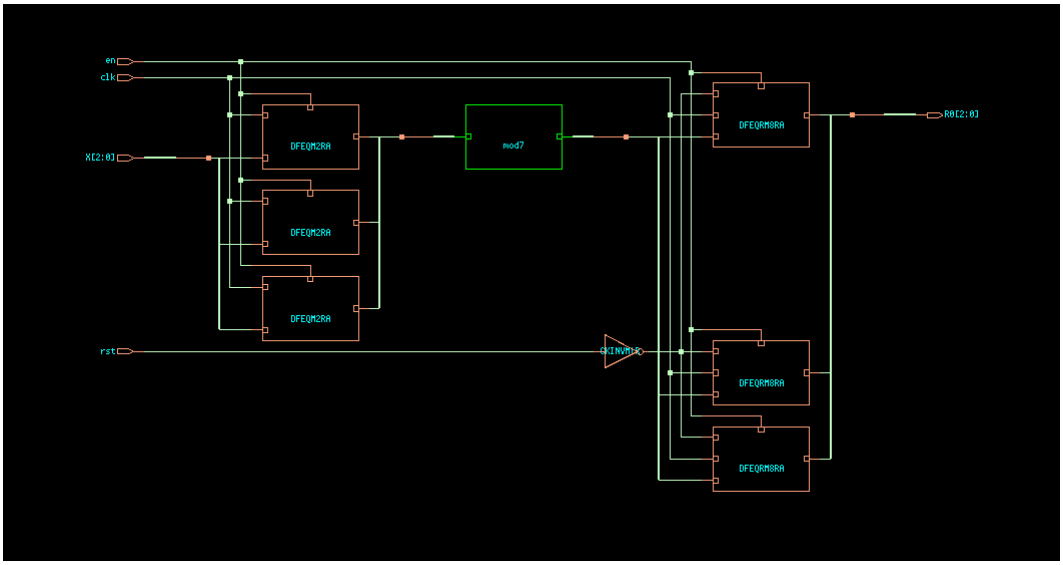
Nas subseções seguintes, serão discutidas duas abordagens para a implementação do operador modulo nativo em VHDL, assim como uma implementação através de uma solução dedicada usando uma ferramenta livre existente. Ambas as abordagens realizam operações de soma, multiplicação e conversão direta como já foi indicado. A operação de soma modular é idêntica nas duas formas de implementação como na Figura 2.5 por serem inestáveis os do tipo $2^n \pm 1$ apresentados nas Figuras 2.3 e 2.4.

2.2.1 Análise da Implementação do Operador MOD por Conversão Direta

O circuito abaixo executa a operação módulo 7 calculando o resto da divisão inteira de uma entrada de quatro bits pelo número 7. Tal circuito foi projetado para ter uma saída de três bits, considerando que a faixa dinâmica de representação do módulo 7 é de 3 bits. A Figura 2.8 demonstra a síntese do operador mod() utilizando a linguagem VHDL e sintetizada pela ferramenta *Genus Synthesis Solution* (versão 16.24-S065-1). O bloco sintetizado do operador módulo está destacado em verde, com as suas respetivas entradas e saídas registradas. A Figura 2.8, por sua vez, exibe os circuitos contidos nesse bloco.

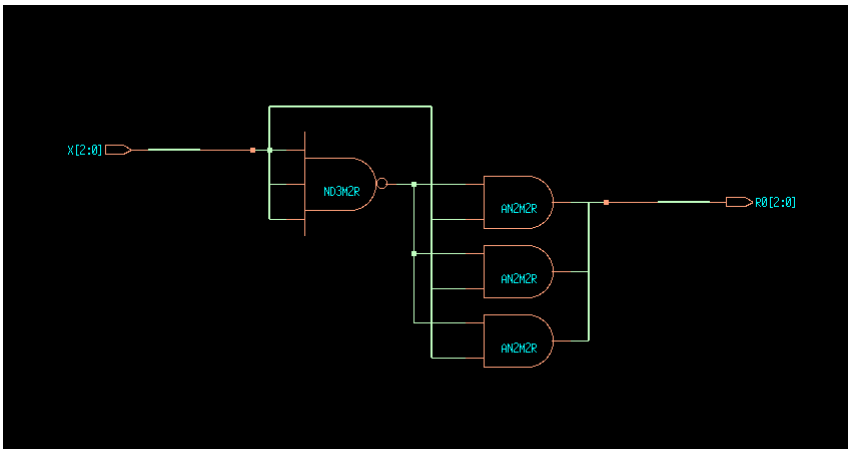
De observar que o circuito gerado possui células padrões do tipo “AO32M2RA”, “MUX2M0RA”, entre outros. A biblioteca utilizada para síntese deste exemplo foi a *UMC-65nm*, entretanto o projetista pode possuir dificuldade em encontrar a descrição detalhada de algumas células em tais bibliotecas comerciais. Em todo o caso, pode-se deduzir a solução do circuito pela tabela da verdade. Basicamente como conclusão é que com este exemplo e com experimentos feitos posteriormente, o que se verificou é que não é possível mapear a implementação de hardware. Não é possível mapear porque basicamente apareceram "caixas pretas". Com isso, vamos ver como se comporta a sensitividade do módulo.

Figura 2.8 – Resultado da síntese lógica do operador mod()



Fonte: Elaborado pelo autor

Figura 2.9 – Resultado da síntese lógica do operador mod()



Fonte: Elaborado pelo autor

2.2.2 Proposta de Operadores Dedicados Baseados pela Proposição de Compressão Iterativa

A solução dedicada explora arquiteturas na forma 2^n, $2^n \pm 1$ e $2^n \pm k$. O objetivo da dissertação foi propor uma solução diferente às arquiteturas existentes no estado da arte as quais são baseadas no uso de memórias. A arquitetura proposta é aplicável para operações aritméticas modulares com foco em *Digital Signal Processing* (DSP) e na implementação de conversores RNS diretos e reversos. Tal solução (FERNANDES, 2021) utiliza uma estratégia de colocação de compressores semelhante à abordagem de árvores de Wallace, alocando o máximo número de compressores por iteração. Os compressores [5:3], [3:2] e [2:2] foram escolhidos como blocos fundamentais do projeto, com ordem de prioridade baseada em seus fatores de compressão. A redução da matriz de informação ocorre até que não haja encaixes disponíveis para cada compressor.

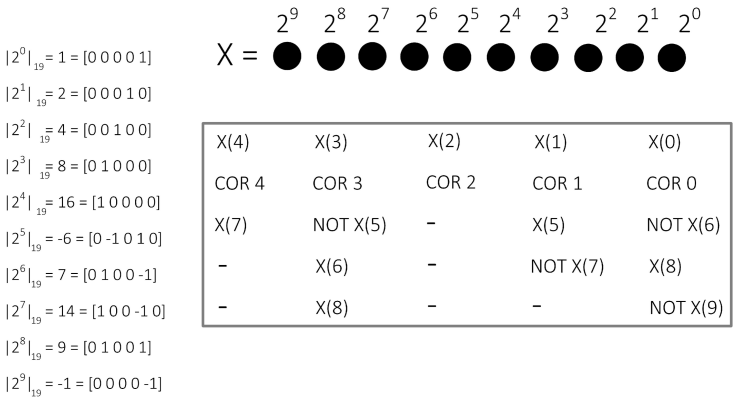
Três soluções de compressão foram geradas uma primeira utilizando apenas compressores [3:2], uma segunda utilizando compressores [3:2] e [2:2] e uma terceira utilizando todo o conjunto de compressores propostos. A solução escolhida foi aquela que apresentou o melhor atraso Os bits reintroduzidos na matriz são representados na forma normal ou recodificada, selecionando-se a opção com menor número de 1's. A recodificação proporciona uma vantagem na compressão, reduzindo o número de elementos reintroduzidos.

Com a criação de uma metodologia para alocação eficiente de compressores foi possível obter resultados satisfatórios para resolução da estrutura com vista a execução de operações modulares. Conseguiu-se observar ganhos no atraso do tempo de computação em relação à solução existente e que foi o ponto de partida da dissertação (PALUDO, 2020). Em busca de explorar métodos de otimização em hardware para operações aritméticas modulares, vamos analisar a existência de sensitividade do operador mod() em VHDL aos diferentes conjuntos de módulo comparando com a solução dedicada.

2.2.2.1 Conversão Direta

A Figura 2.10 ilustra um exemplo de como é feita a pré-computação em um conversor direto com módulo 19. A variável que será convertida para o formato modular tem 10 bits. Os pesos que são maiores que 2^y, onde y é igual a 4 (calculado como $\lceil log_2 19 \rceil$), são realocados para as posições inferiores a y. É possível usar representações tanto positivas quanto negativas, como por exemplo o módulo $|2^5|_{19} = 13$ mas também pode ser $|2^5|_{19} = -6$. Um fator de correção indicado como COR pode ser aplicado para compensar os elementos negativos.

Figura 2.10 – Pré-computação para conversor direto módulo 19



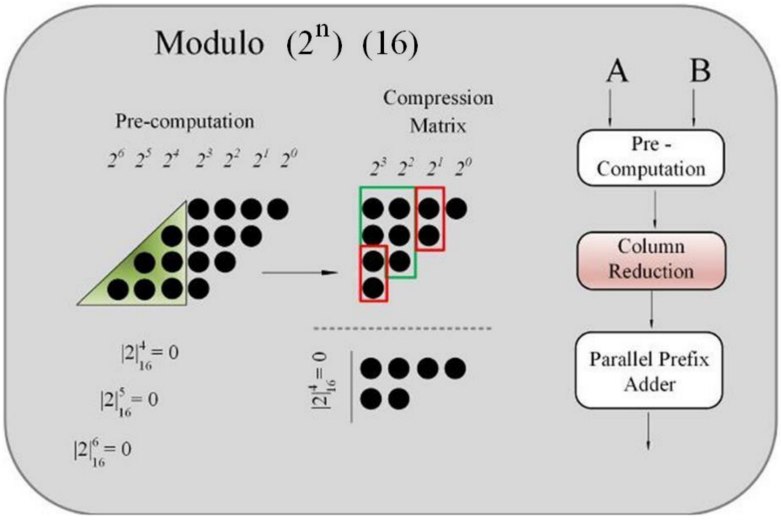
Fonte: Elaborado pelo coorientador

2.2.2.2 Multiplicação Modular

CASO 2^n

Para casos de 2^n, a multiplicação modular pode ser obtida através do truncamento que sai do compressor. Os vetores resultantes são comprimidos através de uma estrutura de árvores, com os sinais de *carry-out* truncados. Na parte final, os dois vetores restantes são somados usando um único somador final, que pode ser um de prefixo paralelo ou um CPA. Como os bits maiores que 2^n são sempre iguais a zero, o sinal de *carry-out* do somador não será usado. A estrutura completa do cálculo pode ser vista na Figura 2.11. O compressor [5:3] está delimitado pela cor verde e os compressores [2:2] pela cor vermelha.

Figura 2.11 – Exemplo de multiplicação modular para o caso 2^n $(n = 4)$



Fonte: Elaborado pelo coorientador

CASO $2^n - 1$

Neste caso, a redução da matriz de informações pode ser realizada usando o método EAC em todos os níveis da árvore de CSA, uma vez que $|2^n + i|_{2^n-1} = 1$. O valor máximo que pode ser obtido na saída do compressor é dado por A + B = 2*($2^n - 1$). No caso de um módulo 31, o maior número que pode ser alcançado após a compressão é 30, levando a um valor máximo de soma de 60. Este valor é o dobro do valor do módulo, portanto, é necessário modularizar a saída. Para isso, usamos um somador módulo $2^n - 1$, que calcula paralelamente a soma $(A + B)$ e $(A + B - m)$, conforme explicado no capítulo sobre somas modulares. No exemplo da Figura 2.12 ($n = 5$) em que os compressores [5:3] estão delimitados pela cor verde e os compressores [3:2] pela cor azul, ambos os vetores estão completos, o que impede a simplificação do hardware da soma final. No entanto, essa é uma situação específica. Em geral, os multiplicadores modulares gerados neste estudo terão somas automaticamente simplificadas quando as oportunidades forem identificadas.

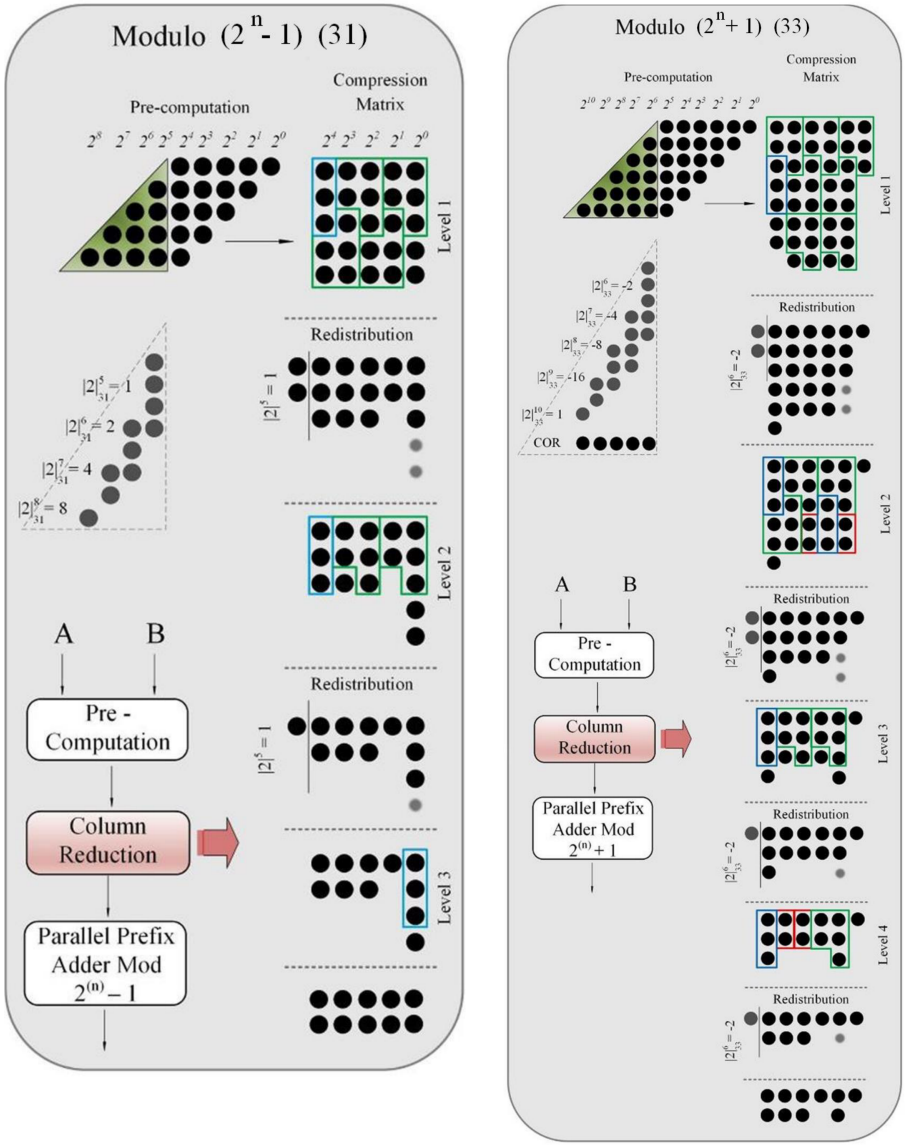
CASO $2^n + 1$

A operação de adicionar vetores na estrutura de compressão pode ser efetuada utilizando o método IEAC, ou seja, os bits de *carry-out* são reintroduzidos de forma parecida ao caso $2^n + 1$, considerando que $|2^n|_{2^n+1}$ resulta em –1, indicando assim que o peso tem um valor negativo. A conclusão do processo de compressão é feita através de um somador de prefixo paralelo no módulo $2^n + 1$. Se houver a oportunidade, o somador final será simplificado. A Figura 2.12 demonstra uma ilustração do caso para ($n = 5$). Os compressores [5:3] estão delimitados pela cor verde, os compressores [3:2] pela cor azul e os compressores [2:2] pela cor vermelha.

CASO $2^n \pm k$

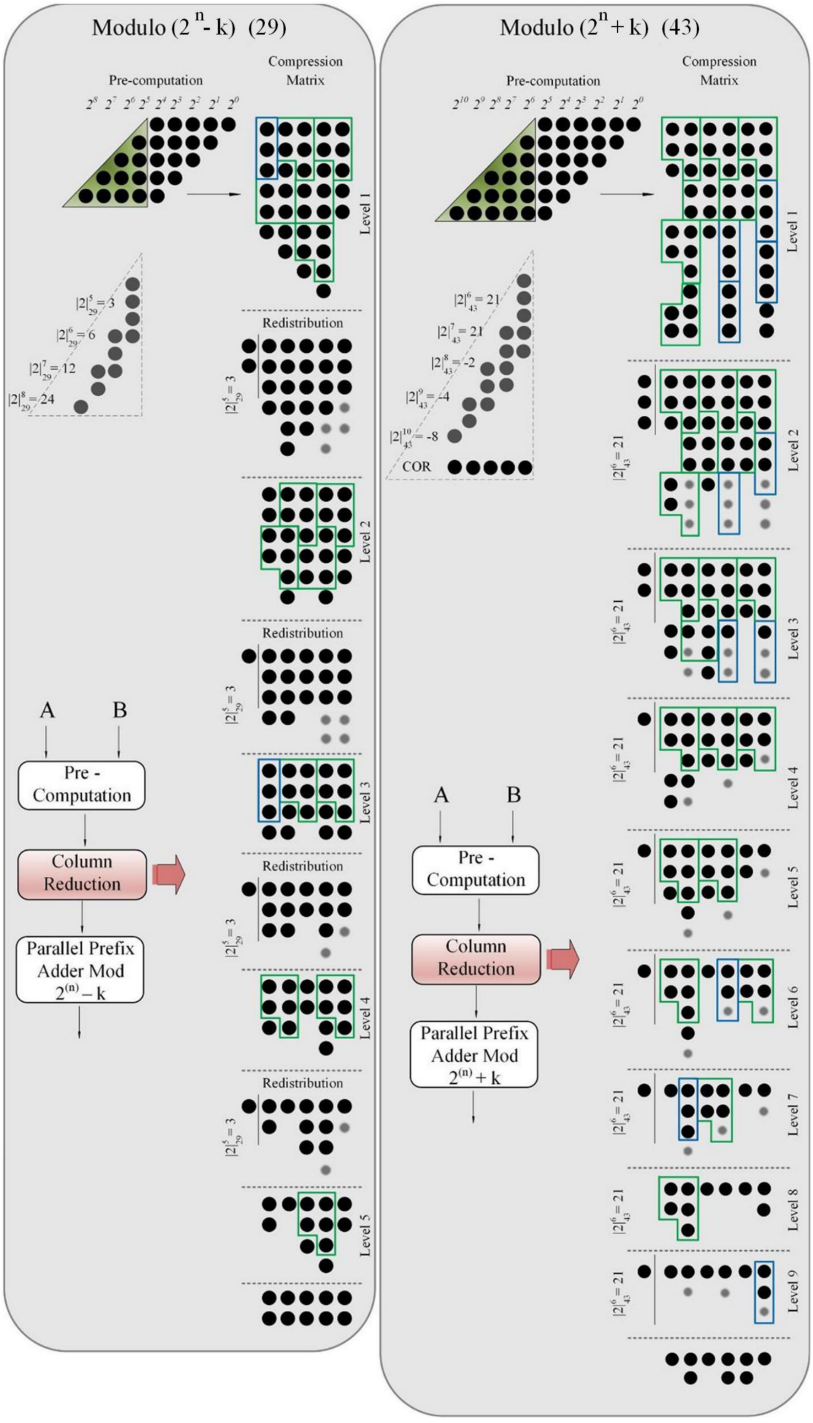
A Figura 2.13 mostra a pré-computação de produtos parciais e a estrutura de hardware $2^5 - 3$ e $2^5 + 11$. Os compressores [5:3] estão delimitados pela cor verde e os compressores [3:2] pela cor azul. O peso 2^6, que pode ser representado como $|23|_{43}$, é igual a 21 ou -22. Dado que ambos os números têm três 1s em suas representações, o 21 positivo é selecionado como um resíduo parcial. Os pesos restantes (2^7, 2^8, 2^9, 2^{10}) são escolhidos como -1, -2, -4, -8, respetivamente, pois têm apenas um bit 1. Após a pré-computação, a compressão é realizada, seguida pela soma final, completando o cálculo. No entanto, neste exemplo, o peso $2^6 = 21$ aumenta a complexidade, pois cada bit que deixa a matriz se transforma em três bits quando reintroduzido em todos os níveis de compressão. Embora seja preferível módulos menores e mais simples na forma de $2^n - 1$ com k contendo um número reduzido de 1's, nem sempre é possível.

Figura 2.12 – Exemplo de multiplicação modular para o caso $2^n - 1$ e $2^n + 1$ ($n = 5$)



Fonte: Elaborado pelo coorientador

Figura 2.13 – Exemplo de multiplicação modular para o caso $2^n - 3$ e $2^n + 11$ ($n = 5$)



Fonte: Elaborado pelo coorientador

3 EXPLORAÇÃO DE PERFORMANCE DAS OPERAÇÕES MODULARES

3.1 ANÁLISE DE PERFORMANCE EM ARITMÉTICA MODULAR

Para fins de comparação e análise de performance, cada arquitetura das operações modulares e da solução dedicada foram descritas em VHDL e sintetizadas. A ferramenta *Genus Synthesis Solution* (versão 16.24-S065-1) foi configurada para aplicar seus recursos de otimização para o determinado processo de síntese. Os resultados estão expostos nas figuras para diferentes conjuntos de módulos. Foram utilizadas as métricas típicas de avaliação de circuitos digitais, atraso (ps), área ($10 \times 10^3 \mu m^2$), bem como o produto da área e do atraso, conhecido como *Area-Delay-Product* (ADP). Serão apresentadas os resultados das sínteses lógicas para as operações de soma e multiplicação do operador mod nativo e da solução dedicada. Atendendo aos resultados irá ser proposta uma aplicação usando módulos auxiliares.

3.1.1 Análise dos Resultados para Soma Modular

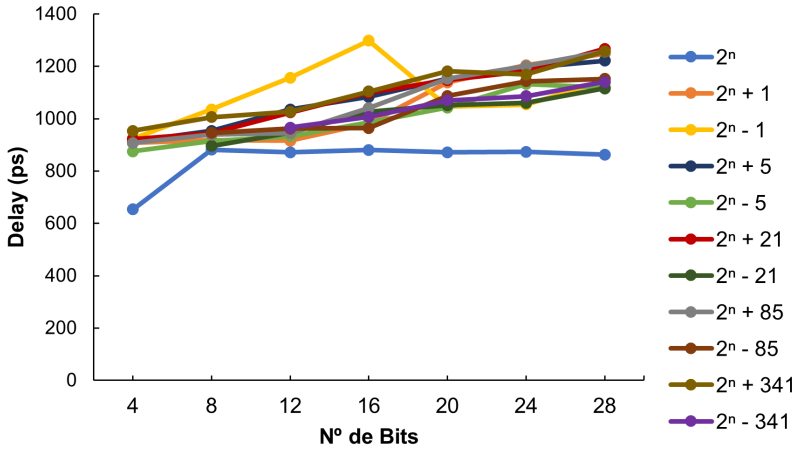
Nesta subseção apresenta-se os resultados de síntese lógica para a soma modular do operador mod() e solução dedicada. Nas Figuras 3.1 e 3.3, temos os resultados de atraso correspondentes a operação de soma modular. É possível observar na Figura 3.2 uma distinta eficiência superior do módulo 2^n em relação aos demais tipos de módulo apresentados para operação modular de soma nativa. Possuí menor atraso em relação aos demais, o que é esperado visto que a operações baseadas em 2^n podem ser simplificadas devido a possibilidade de realizar truncamentos e consequentemente o descarte de bits com pesos superiores ao valor de n.

A eficiência dos demais módulos aglomera-se com proximidade entre os diversos tipos de módulos analisados. Entretanto, existe um aparente aumento de atraso para módulos do tipo $2^n \pm k$ a medida em que cresce o número de 1's binários no valor de k. O gráfico também demonstra um valor de *outliers* para o módulo $2^n - 1$ inesperado para os valores de $n = 8$ até $n = 20$ em que o dito módulo demonstrou atraso superior a todos os demais.

Comparando com os resultados da soma modular utilizando a arquitetura construída pela solução dedicada, observa-se que os módulos $2^n + 1$ e $2^n - 1$ têm valores de atraso similares, enquanto os módulos do tipo $2^n + k$ possuem atraso maior. Isso ocorre, uma vez que os módulos $2^n + k$ geralmente ultrapassam o valor da representação de três vezes o valor do módulo, sendo necessária uma arquitetura de maior complexidade para correção desta faixa mais ampla.

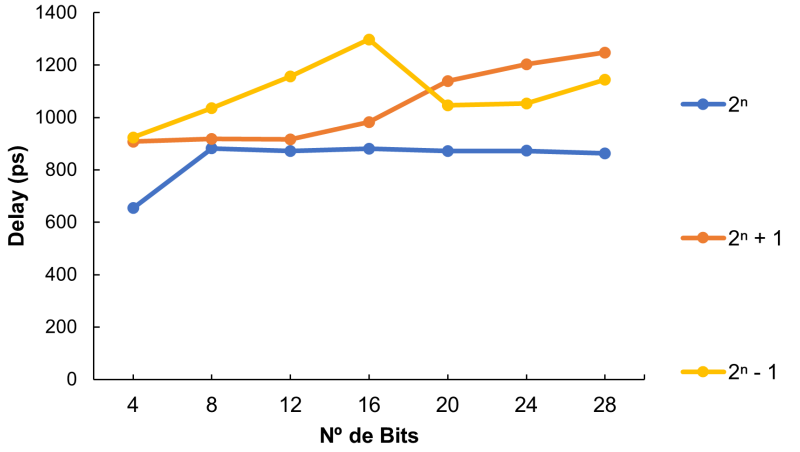
Nesse sentido, o resultado da soma dos demais módulos citados não ultrapassa a contagem de duas vezes o valor do módulo, acarretando assim em uma etapa de correção da soma modular de menor custo em hardware. Ao comparar as abordagens baseadas na

Figura 3.1 – Atraso Soma Modular do Operador MOD



Fonte: Elaborado pelo autor

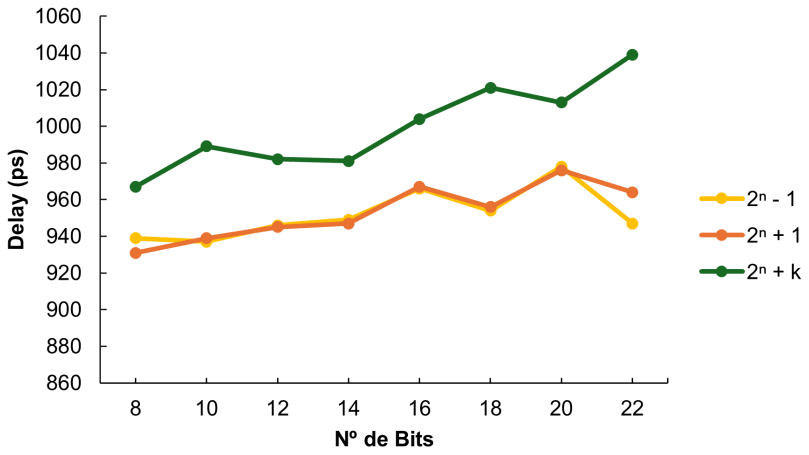
Figura 3.2 – Atraso Soma Modular do Operador MOD para 2^n, $2^n - 1$ e $2^n + 1$



Fonte: Elaborado pelo autor

solução dedicada e no operador módulo nativo é possível observar que os resultados são de certa forma similares. Entretanto a solução dedicada demonstra com valores de atraso menores em comparação com o operador mod nativo, traduzindo em ganhos médios de 2,81% para o módulo $2^n + 1$, 15% para $2^n - 1$ e 5,54% para o módulo $2^n + k$ e mais evidente a medida em que se aumenta o número de bits.

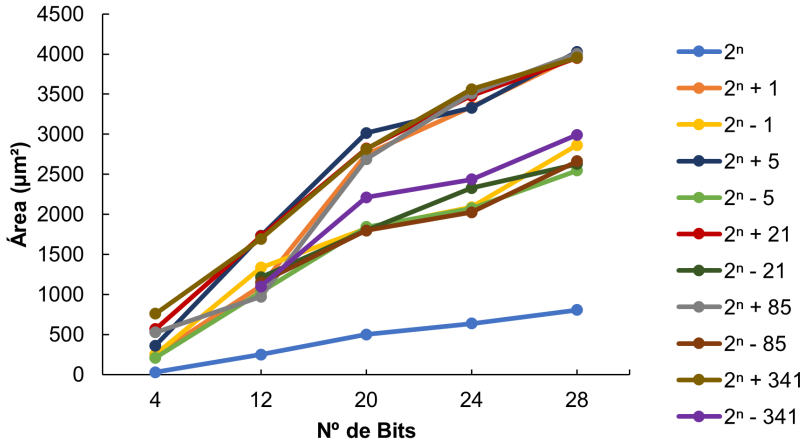
Figura 3.3 – Atraso Soma Modular Solução Dedicada



Fonte: Elaborado pelo autor

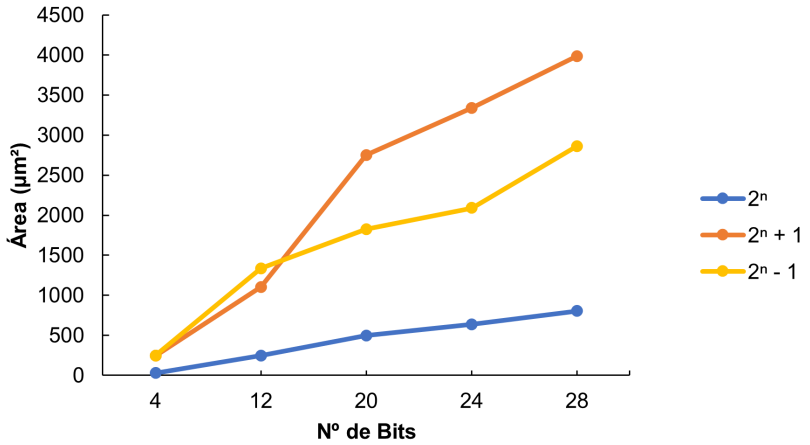
Relativamente à ocupação de área da operação de soma modular do operador nativo, o módulo 2^n apresenta-se melhor como pode-se observar nas Figuras 3.4 e 3.5. Enquanto os demais módulos crescem em área a medida em que se aumenta o número de bits. Existem alguns *outliers* inesperados para o operador modular de soma, mas em geral a área cresce para módulo do tipo $2^n + k$ à medida em que cresce o número de 1's binários no valor de k. Na Figura 3.6 que corresponde ao gráfico da solução dedicada, observa-se menor área ocupada pelo módulo $2^n - 1$, seguido do módulo $2^n + 1$ e por último os módulos $2^n + k$. Comparando as duas formas de implementação é de observar que o operador mod nativo ocupa menor área em relação à solução dedicada com ganhos médios de 26,46% para o módulo $2^n + 1$ e 24,63% para o módulo $2^n + k$. Entretanto a solução dedicada ocupa menor área no módulo $2^n - 1$ com ganho médio de 12,37%.

Figura 3.4 – Área Soma Modular do Operador MOD



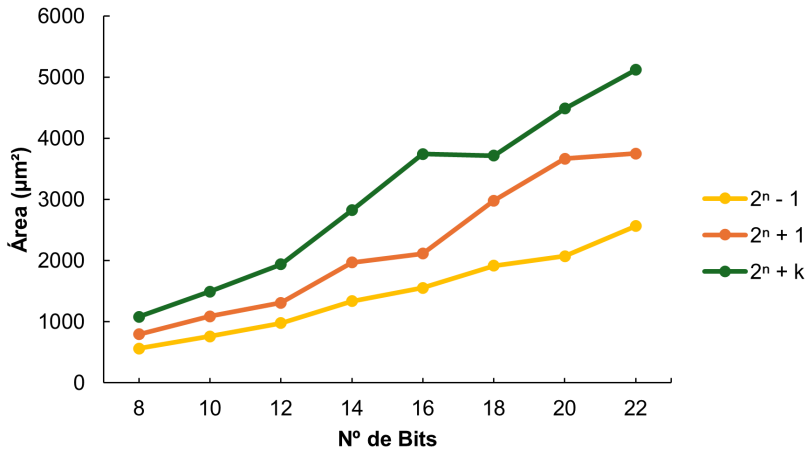
Fonte: Elaborado pelo autor

Figura 3.5 – Área Soma Modular do Operador MOD para 2^n, $2^n - 1$ e $2^n + 1$



Fonte: Elaborado pelo autor

Figura 3.6 – Área Soma Modular Solução Dedicada



Fonte: Elaborado pelo autor

3.1.2 Análise dos Resultados para Multiplicação Modular

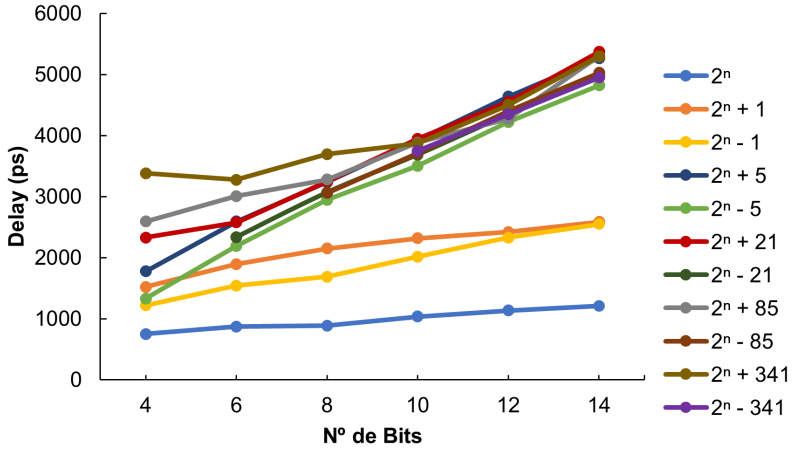
Nesta subseção apresenta-se os resultados de síntese lógica para a multiplicação modular do operador mod e solução dedicada. Analisando os resultados pelas Figuras 3.7 e 3.8, denotamos uma eficiência maior usando a solução dedicada para conjuntos modulares na forma 2^n, $2^n - 1$, $2^n + 1$, $2^n \pm k$. Podemos observar sensitividade para módulos do tipo $2^n + k$ na solução mod (), que a medida em que cresce o número de 1's binários no valor de k até o valor de $n = 10$. Posteriormente os valores se concentram muito próximos uns dos outros.

Sugere-se que o $2^n - k$ não possuí mais sensitividade. As Tabelas 3.1 até 3.5 indicam melhorias em termos de atraso utilizando a solução dedicada. Com o intuito de obter redução de atraso de circuitos, para o conjunto modular $2^n - 1$ conseguiu-se uma média de ganhos de 17,54%, 15,7% para $2^n + 1$ e 28,86% para módulos $2^n \pm k$. Esses valores de ganho são relativos à melhoria obtida com a solução dedicada em comparação com o operador mod nativo. A multiplicação baseada na solução dedicada apresenta maior sensitividade e permite a possibilidade de aplicações dos módulos auxiliares.

Tabela 3.1 – Atraso da multiplicação modular para o módulo 2^n

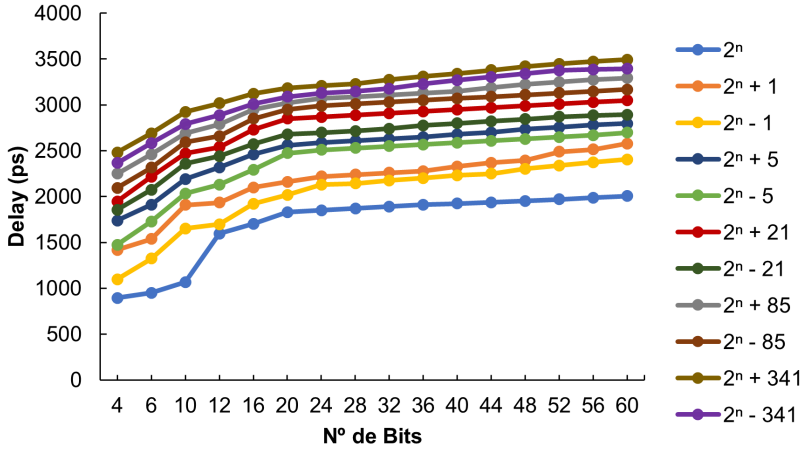
Módulo	16	64	256	1024	4096	16364
Op. mod Nativo (ps)	**752**	**873**	**886**	**1035**	**1135**	**1210**
Solução Dedicada (ps)	894	952	984	1069	1205	1330
Ganho	-18,88%	-9,05%	-11,06%	-3,29%	-6,17%	-9,92%

Figura 3.7 – Atraso Multiplicação Modular do Operador MOD



Fonte: Elaborado pelo autor

Figura 3.8 – Atraso Multiplicação Modular da Solução Dedicada



Fonte: Elaborado pelo autor

Tabela 3.2 – Atraso da multiplicação modular para o módulo $2^n - 1$

Módulo	15	63	255	1023	4095	16363
Op. mod Nativo (ps)	1219	1542	1689	2015	2331	2547
Solução Dedicada (ps)	**1099**	**1397**	**1456**	**1653**	**1697**	**1858**
Ganho	**9,84%**	**9,40%**	**13,79%**	**17,96%**	**27,20%**	**27,05%**

Tabela 3.3 – Atraso da multiplicação modular para o módulo $2^n + 1$

Módulo	17	65	257	1025	4097	16385
Op. mod Nativo (ps)	1520	1894	2150	2317	2423	2584
Solução Dedicada (ps)	**1419**	**1678**	**1764**	**1909**	**1935**	**2055**
Ganho	**6,64%**	**11,40%**	**17,95%**	**17,61%**	**20,14%**	**20,47%**

Tabela 3.4 – Atraso da multiplicação modular para o módulo $2^n - k$

Módulo	13	59	235	1019	4011	16043
Op. mod Nativo (ps)	1358	2191	3069	3506	4397	4951
Solução Dedicada (ps)	**1294**	**1730**	**2199**	**2034**	**2661**	**2957**
Ganho	**4,71%**	**21,04%**	**28,35%**	**41,99%**	**39,48%**	**40,27%**

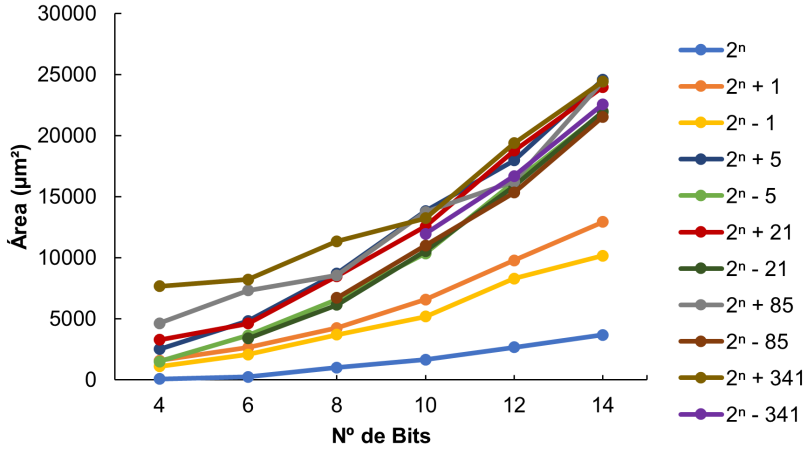
Tabela 3.5 – Atraso da multiplicação modular para o módulo $2^n + k$

Módulo	21	69	277	1045	4181	16725
Op. mod Nativo (ps)	1775	2592	3249	3948	4276	5297
Solução Dedicada (ps)	**1739**	**1911**	**2327**	**2472**	**2790**	**3090**
Ganho	**2,03%**	**26,27%**	**28,38%**	**37,39%**	**34,75%**	**41,67%**

Em relação à área, conforme ilustrado nas Figuras 3.9 e 3.10, as tendências serão semelhantes ao que já foi explicado. Isto é, maior eficiência para módulos do tipo $2^n - 1$ e $2^n + 1$, seguidos pelos módulos $2^n \pm k$ que têm proporcionalidade com o número de 1's no elemento k. Percebe-se um aumento da área conforme o número de bits aumenta conforme apresentado nas Tabelas 3.6 até 3.10.

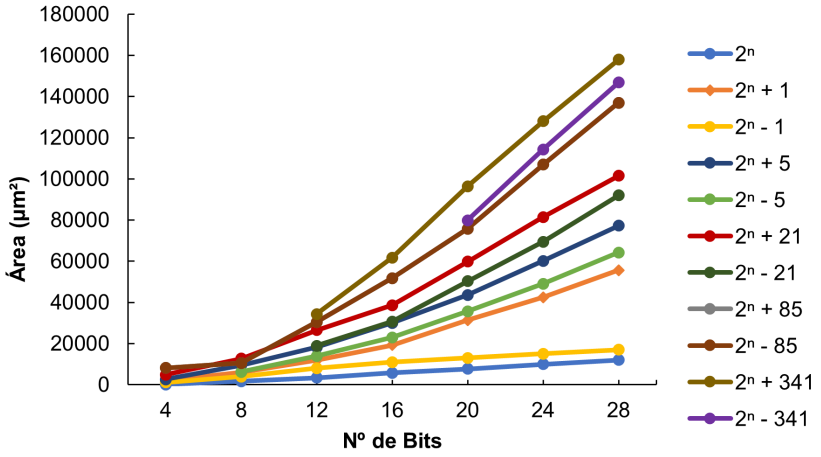
O gráfico do ADP para o operador modular da soma do operador nativo e da solução dedicada estão expostas nas Figuras 3.11 e 3.12. Nota-se, no gráfico do operador mod nativo, podemos observar que o módulo 2^n é mais eficiente em área/delay que os demais. Os restantes módulos aumentam em área/delay à medida que o número de bits aumenta. Embora haja *outliers* inesperados para o operador modular de soma como referido anteriormente, a tendência geral é de crescimento de área para módulos do tipo $2^n \pm k$ à medida que o número de 1's binários em k aumenta. No gráfico do operador modular de multiplicação utilizando a solução dedicada observa-se que os conjuntos modulares 2^n, $2^n - 1$ e $2^n + 1$ apresentam atrasos e áreas menores. Foi explorado o AD^2P e como os resultados da solução dedicada apresentam menor atraso comparado com o operador nativo, então esses resultados vão ser sempre melhores para AD^2P. Confirma-se a sensitividade já vista no gráfico da operação de multiplicação a sensitividade ao tipo de módulo.

Figura 3.9 – Área Multiplicação Modular do Operador MOD



Fonte: Elaborado pelo autor

Figura 3.10 – Área Multiplicação Modular da Solução Dedicada



Fonte: Elaborado pelo autor

Tabela 3.6 – Área da multiplicação modular para o módulo 2^n

Módulo	16	64	256	1024	4096	16364
Op. mod Nativo (μm^2)	**67**	**249**	**993**	**1657**	**2659**	**3668**
Sol. Dedicada (μm^2)	172	430	1295	2173	3399	4636
Ganho	-156,72%	-72,69%	-30,41%	-31,52%	-27,83%	-26,39%

Tabela 3.7 – Área da multiplicação modular para o módulo $2^n - 1$

Módulo	15	63	255	1023	4095	16363
Op. mod Nativo (μm^2)	**1094**	**2059**	**3687**	**5175**	8294	10169
Sol. Dedicada (μm^2)	1189	2172	3955	5780	**8031**	11190
Ganho	-8,68%	-5,49%	-7,27%	-11,69%	**3,17%**	-10,04%

Tabela 3.8 – Área da multiplicação modular para o módulo $2^n + 1$

Módulo	17	65	257	1025	4097	16385
Op. mod Nativo (μm^2)	**1578**	**2643**	**4243**	**6564**	**9782**	**12927**
Sol. Dedicada (μm^2)	2095	3759	6071	8353	11805	16226
Ganho	-32,76%	-42,22%	-43,08%	-27,25%	-20,68%	-25,52%

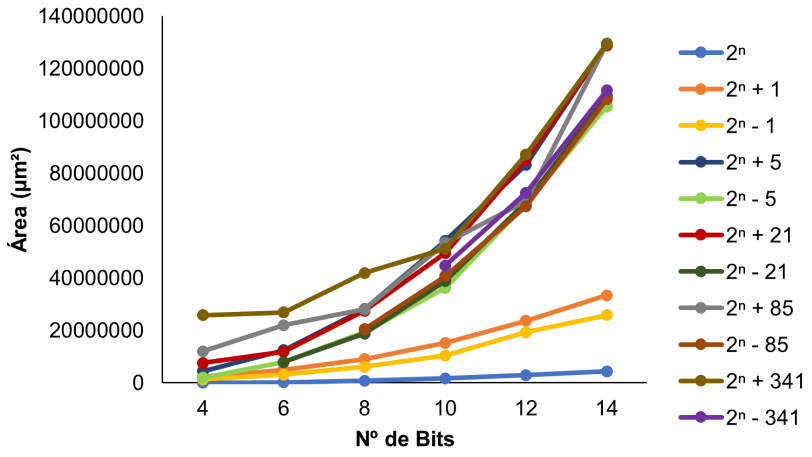
Tabela 3.9 – Área da multiplicação modular para o módulo $2^n - k$

Módulo	13	59	235	1019	4011	16043
Op. mod Nativo (μm^2)	**1429**	**3645**	**6128**	**10357**	**15333**	**24431**
Sol. Dedicada (μm^2)	1688	4013	6181	10796	25479	41260
Ganho	-18,12%	-10,10%	-0,87%	-4,24%	-66,17%	-68,88%

Tabela 3.10 – Área da multiplicação modular para o módulo $2^n + k$

Módulo	21	69	277	1045	4181	16725
Op. mod Nativo (μm^2)	**2522**	**4810**	**8481**	**12568**	**16232**	**24431**
Sol. Dedicada (μm^2)	2567	6421	12688	19637	26530	46551
Ganho	-1,78%	-33,49%	-49,60%	-56,25%	-63,44%	-90,54%

Figura 3.11 – ADP Multiplicação Modular do Operador MOD

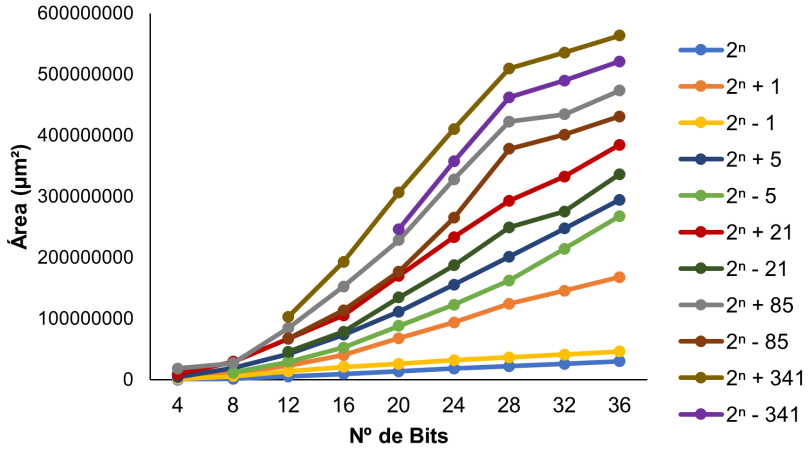


Fonte: Elaborado pelo autor

Tabela 3.11 – ADP da multiplicação modular para o módulo 2^n

Módulo	16	64	256	1024	4096	16364
Mod Nativo ($10^5 \mu m^2$)	**0,50**	**2,17**	**8,80**	**17,14**	**30,18**	**44,37**
Sol. Dedicada ($10^5 \mu m^2$)	1,53	2,51	16,64	23,22	54,21	61,65
Ganho	-207,4%	-15,54%	-89,16%	-35,47%	-79,62%	-38,94%

Figura 3.12 – ADP Multiplicação Modular do Operador Solução Dedicada



Fonte: Elaborado pelo autor

Tabela 3.12 – ADP da multiplicação modular para o módulo $2^n - 1$

Módulo	15	63	255	1023	4095	16363
Mod Nativo ($10^5 \mu m^2$)	13,334	31,753	62,268	104,285	193,338	258,998
Sol. Dedicada ($10^5 \mu m^2$)	**13,067**	**28,757**	**58,098**	**95,543**	**136,286**	**207,910**
Ganho	**2,00%**	**9,43%**	**6,70%**	**8,38%**	**29,51%**	**19,73%**

Tabela 3.13 – ADP da multiplicação modular para o módulo $2^n + 1$

Módulo	17	65	257	1025	4097	16385
Mod Nativo ($10^5 \mu m^2$)	**23,981**	**50,06**	**91,22**	**152,07**	237,02	334,04
Sol. Dedicada ($10^5 \mu m^2$)	29,72	53,34	107,09	159,45	**228,42**	**333,44**
Ganho	-23,96%	-6,54%	-17,40%	-4,85%	**3,62%**	**0,18%**

Tabela 3.14 – ADP da multiplicação modular para o módulo $2^n - k$

Módulo	11	59	235	1019	4011	16043
Mod Nativo ($10^5 \mu m^2$)	20,09	79,86	188,06	363,11	**674,2**	**1116,99**
Sol. Dedicada ($10^5 \mu m^2$)	**10,52**	**69,42**	**135,92**	**219,59**	677,99	1189,11
Ganho	**47,64%**	**13,08%**	**27,73%**	**39,53%**	-0,56%	-6,46%

Tabela 3.15 – ADP da multiplicação modular para o módulo $2^n + k$

Módulo	21	69	277	1045	4181	16725
Mod Nativo ($10^5 \mu m^2$)	44,768	124,669	**275,549**	496,166	**694,088**	**1294,105**
Sol. Dedicada ($10^5 \mu m^2$)	**44,640**	**111,083**	295,249	**485,426**	847,964	1370,934
Ganho	**0,29%**	**10,90%**	-7,15%	**2,16%**	-22,17%	-5,94%

3.2 SENSITIVIDADE MODULAR

De acordo com análises demonstrados anteriormente os tipos de módulo 2^n, $2^n - 1$ e $2^n + 1$ apresentaram um desempenho melhor em termos de atraso comparado com $2^n \pm k$ que ao contrário é sabido que o custo temporal de operações de hardware é inversamente proporcional ao numero de 1's na representação binária. Quando se torna necessário executar uma operação usando um módulo menos eficiente, uma solução prática é substituir por módulos auxiliares que se alinhem com formatos eficientes. Módulos auxiliares, essencialmente, derivam de outros módulos por meio do processo de remodulação com o objetivo de obter um k mais eficaz em termos de atraso. De modo a facilitar esse processo, existe uma equação que permite derivar um módulo auxiliar de qualquer módulo complexo, enunciada como:

$$||X|_{m1}|_{m2} = |X|_{m2}, \quad \text{onde } m_1 = c \cdot m_2 \tag{3.1}$$

sendo X uma operação aritmética, m_1 o módulo original, m_2 o módulo auxiliar e c uma constante inteira. Para ilustrar, consideremos a operação $|a \cdot b| \mod m_2$, onde $a = 1024$, $b = 16$ e $m_2 = 85$. O resultado da operação pode ser reescrito para se ajustar ao formato $2^n + k$:

$$85 \equiv 64 + 21 = 2^6 + 21 \tag{3.2}$$

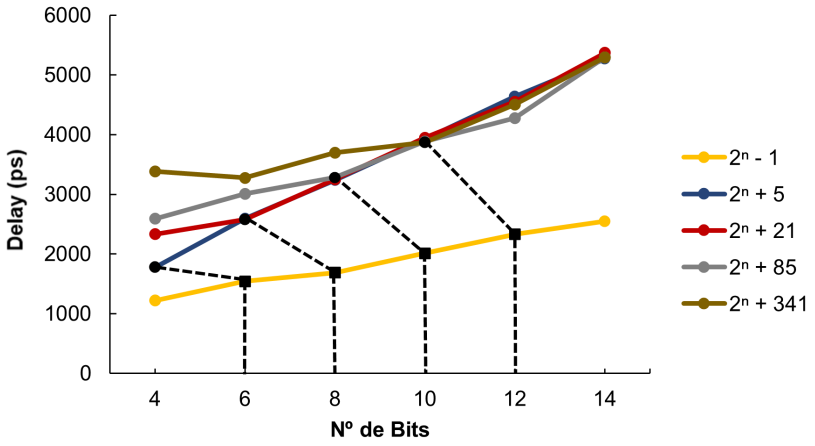
Neste caso, temos $k = 21$, com representação binária $k = 10101_2$, contendo três 1's. Podemos então calcular o módulo auxiliar de 85 usando a propriedade $255 = 3 \cdot 85$ o módulo auxiliar resultante é $2^8 - 1 = 255$, respeitando o formato ideal $2^n - 1$. Concluímos que $255 = 3 \cdot 85$, confirmando ser módulo auxiliar, pois é um múltiplo que atende à condição necessária. É importante destacar que esse processo envolve a conversão inicial do valor de entrada para o módulo auxiliar desejado, seguida das operações aritméticas na base convertida. Somente após a conclusão dessas operações, o resultado é convertido de volta para o módulo original (PARHAMI, 2009).

Para avaliar o impacto de atraso dessa mudança na operação modular de multiplicação utilizando o operador mod nativo e a solução dedicada de hardware, implementamos multiplicadores usando aritmética modular, e analisamos o impacto de atraso nos módulos. A unidade multiplicadora para $2^n \pm k$ utiliza uma memória ROM para armazenar as contribuições dos *carrys* somados. Eles são, menos eficientes em comparação com as unidades $2^n \pm 1$, que operam de forma combinacional sem a necessidade de memória. Para a realização da aplicação de sensitividade foi escolhido um conjunto de módulo específico, com o seu módulo auxiliar para o número de bits n e números de 1´s na representação binária de k. Os módulos escolhidos foram $2^n - 1$ e $2^n + k$ com quatro 1´s em k conforme a tabela abaixo. Os módulos auxiliares estão representados em quadrados no gráfico, enquanto as respetivas informações dos módulos originais estão indicadas pelos círculos.

Tabela 3.16 – Módulos e módulos auxiliares utilizados para teste em hardware

Módulo (m_2)	Valor	1's em k	c	Módulo Auxiliar (m_1)	Valor	1's em k
2^4+5	21	2	3	2^6-1	63	1
2^6+21	85	3	3	2^8-1	255	1
2^8+85	341	4	3	$2^{10}-1$	1023	1
$2^{10}+341$	1365	5	3	$2^{12}-1$	4095	1

Figura 3.13 – Atraso Multiplicação Modular Operador MOD



Fonte: Elaborado pelo autor

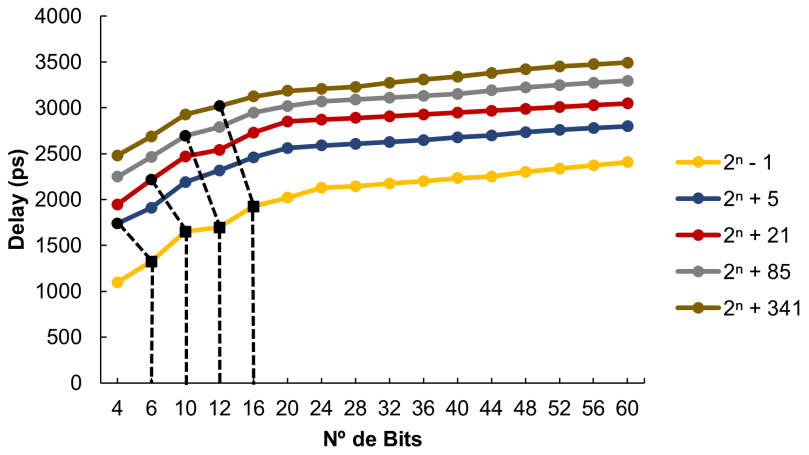
Tabela 3.17 – Resultados de Atraso dos Módulos Auxiliares Operador MOD

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1)	Atraso	Ganho
2^4+5	1775	2^6-1	**1542**	**13,13%**
2^6+21	2578	2^8-1	**1689**	**34,48%**
2^8+85	3279	$2^{10}-1$	**2015**	**38,55%**
$2^{10}+341$	3870	$2^{12}-1$	**2331**	**39,77%**

Ao examinar os resultados alcançados utilizando o operador mod, percebe-se que, os módulos auxiliares no formato 2^n-1, apesar de serem maiores (com maior número n de bit), exibem um menor atraso em comparação com os formatos 2^n+k demonstrando resultado satisfatório com ganhos médios de 31,5%. No caso, com três ou mais 1's, a quantidade de uns na representação binária não impactará mais o atraso, graças à utilização da memória ROM mencionada na arquitetura do multiplicador, a qual elimina tal influência. Portanto, os tipos 2^n-1 exibem um desempenho superior, uma vez que não dependem de componentes de armazenamento, como memórias, que inevitavelmente introduzem atrasos. Apesar disso, é essencial ressaltar a presença de outras implementações que processam a contribuição dos *carrys* de maneira puramente combinacional; nestes casos, a quantidade de 1's terá um impacto no tempo de operação.

Analisando os resultados para a operação modular utilizando a solução dedicada,

Figura 3.14 – Atraso Multiplicação Modular Solução Dedicada



Fonte: Elaborado pelo autor

Tabela 3.18 – Resultados de Atraso dos Módulos Auxiliares Solução Dedicada

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1)	Atraso	Ganho
2^4+5	1739	2^6-1	**1324**	**23,86%**
2^6+21	2216	2^8-1	**1469**	**33,71%**
2^8+85	2580	$2^{10}-1$	**1653**	**35,93%**
$2^{10}+341$	2926	$2^{12}-1$	**1697**	**42%**

destaca-se novamente menor atraso após a correção feita para módulos 2^n-1 com ganhos médios de 33,9%. Assim, demonstra-se que utilizar módulos auxiliares para substituir módulos complexos e de baixo desempenho como 2^n+k, com o intuito de aprimorar o tempo de operação, revela-se uma estratégia eficaz permitindo o uso em algoritmos de exponenciação modular já que envolve operações iterativas modulares.

Todas as análises feitas nesta seção possuem um custo de correção. As Tabelas 3.19 e 3.20 estão expostos os resultados dessas correções. No operador nativo conseguiu-se um ganho de $9,91\%$ para o módulo 2^8+85 e $13,51\%$ para $2^{10}+341$. Na solução dedicada obteve-se um ganho de $8,42\%$ para o módulo $2^{10}+341$. De referir que estes resultados são uma iteração, logo destaca-se esses ganhos porque se fosse realizada várias iterações teríamos um ganho acumulativo.

Tabela 3.19 – Resultados de Correção do Atraso dos Módulos Auxiliares Operador MOD

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1) + Correção	Atraso	Ganho
2^4+5	1775	2^6-1	2456	-38,37%
2^6+21	2578	2^8-1	2623	-1,72%
2^8+85	3279	$2^{10}-1$	**2954**	**9,91%**
$2^{10}+341$	3870	$2^{12}-1$	**3347**	**13,51%**

Tabela 3.20 – Resultados de Correção do Atraso dos Módulos Auxiliares Solução Dedicada

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1) + Correção	Atraso	Ganho
$2^4 + 5$	1739	$2^6 - 1$	2273	-30,70%
$2^6 + 21$	2216	$2^8 - 1$	2428	-9,57%
$2^8 + 85$	2580	$2^{10} - 1$	2621	-1,59%
$2^{10} + 341$	2926	$2^{12} - 1$	**2674**	**8,42%**

4 APLICAÇÃO DA SENSITIVIDADE MODULAR

Neste capítulo será apresentada a fundamentação da criptografia focada no escopo do trabalho. Serão discutidos os princípios da criptografia, algoritmo RSA e os métodos da exponenciação modular. Por fim será exposta uma aplicação de sensitividade modular.

4.1 FUNDAMENTOS DA CRIPTOGRAFIA

Devido à expansão da área de comunicação de dados, proporcionando um fluxo avultado de informações, existe a preocupação com a segurança dos dispositivos eletrónicos. Com isso, sistemas criptográficos têm sido desenvolvidos com o intuito de garantir segurança nos meios de comunicação. A criptografia tem como objetivo transmitir uma mensagem oculta de um transmissor para o recetor de forma segura (BHARDWAJ; KUMAR; BANSAL, 2017).

No campo do estudo de criptografia em sistemas computacionais, surge o desafio equilibrar um bom desempenho da segurança com a rápida velocidade de transmissão de dados com custos reduzidos. As mensagens precisam ser codificadas e decodificadas em tempo real. Na criptografia, a codificação e decodificação de informação são etapas importantes para possibilitar a comunicação entre dois ou mais participantes. Quando essa criptografia é implementada por meio de software, pode haver um comprometimento do tempo de processamento em alguns casos, o que prejudica o processo de comunicação. Uma alternativa para melhorar a eficiência no processo de codificação e decodificação é a utilização de hardware dedicado para a criptografia. Ao executar o algoritmo de criptografia em um processador dedicado, é possível obter uma melhoria significativa no desempenho.

A criptografia é dividida em três ramos: criptografia simétrica que também podemos denominar de chave privada, criptografia assimétrica ou de chave pública e protocolos de criptografia. A criptografia simétrica considera que duas partes conseguem comunicar de forma secreta partilhando a mesma chave privada. O *Data Encryption Standard* (DES) em sua versão triple-DES e o *Advanced Encryption Standard* (AES) são os algoritmos simétricos mais utilizados. No caso da criptografia de chave pública, cada usuário possui a sua chave secreta, usada para decifração partilhando uma chave publica, que é usada na cifração, permitindo o compartilhamento seguro de chaves através de canais de comunicação não seguros, utilizando protocolos de troca de chaves, como o *Diffie-Hellman*. Atualmente, os métodos mais empregados na criptografia de chave pública são o RSA e o *Elliptic Curve Cryptography* (ECC). Por fim, protocolos criptográficos que de maneira ampla, ocupam da implementação de algoritmos criptográficos. Algoritmos simétricos e assimétricos são importantes para a realização de aplicações como a comunicação segura na Internet. O *Transport Layer Security* (TLS), que é utilizado em todos os navegadores web, é um exemplo de um protocolo de criptografia.(PAAR; PELZL, 2009).

4.1.1 Criptografia Assimétrica

A criptografia de chave pública ou assimétrica surgiu em 1976 *Whitfield Diffie* e *Martin Hellman* (DIFFIE; HELLMAN, 1976). Este sistema surgiu como forma de evitar os problemas na criptografia simétrica. A criptografia assimétrica opera com base na ideia de que uma entidade cria duas chaves de criptografia - uma chave pública e uma chave privada. A chave pública é enviada através de um canal de comunicação que pode ser acessado por qualquer pessoa. A outra entidade usa essa chave pública para codificar sua mensagem. A mensagem codificada só pode ser decodificada usando a chave privada, que não é compartilhada através do canal de comunicação.

Essa abordagem resolve o principal desafio da criptografia simétrica, que é compartilhar a chave secreta de forma segura. O esquema de troca de chaves *Diffie-Hellman* é um exemplo desse processo. No entanto, os métodos de criptografia de chave pública usam números muito grandes nos processos de codificação e decodificação, tornando a comunicação mais lenta em comparação com a criptografia simétrica.

4.1.2 Algoritmo de Chave Pública

Na criptografia de chave pública, um par de chaves é gerado de forma que para obter a chave privada a partir da chave pública retoma a resolução de um problema computacional. Há três principais grupos de algoritmos de chave pública que possuem importância prática (LI; HIGGINS; CLEMENT, 2001):

- **Esquemas de Fatoração de Inteiros:** Existem muitos sistemas de chave pública que se baseiam na complexidade de decompor números inteiros grandes. O RSA é o exemplo mais conhecido deste grupo de algoritmos.
- **Esquemas de Logaritmo Discreto:** Há uma série de algoritmos que se baseiam no chamado problema do logaritmo discreto em campos finitos. Alguns dos exemplos mais notáveis são a troca de chaves *Diffie-Hellman*, a criptografia *Elgamal* e o DSA.
- **Esquemas de Curva Elíptica:** Uma extensão do algoritmo de logaritmo discreto são os esquemas de chave pública baseados em curvas elípticas. Os mais utilizados são a troca de chaves *Diffie-Hellman* baseada em *Elliptic Curve Diffie-Hellman* (ECDH) e o DSA baseado em *Elliptic Curve Digital Signature Algorithm* (ECDSA).

4.1.3 Algoritmo RSA

O principal algoritmo de chave assimétrica usado é o RSA, que utiliza enormes números primos para construir o par de chaves. O algoritmo RSA foi publicado em 1977 por *Ron Rivest*, *Adi Shamir* e *Len Adleman* em (RIVEST; SHAMIR; ADLEMAN, 1978). A utilização desse método criptográfico é diversificada, sendo aplicado tanto em sistemas que necessitam de criptografia de pequenos conjuntos de dados (como no caso do transporte

de chaves) quanto na criação de assinaturas digitais. A robustez e segurança deste método estão ancoradas na complexidade e desafio que é a fatoração de grandes números inteiros.

O processo de geração de chaves RSA inicia com a criação de dois números primos grandes e diferentes, denominados p e q. Posteriormente, calcula-se o módulo m, que é o produto de p e q ($m = p \cdot q$). A partir desses números primos, também se calcula a função totiente de Euler, representada por $\phi(m)$ e definida como $(p-1) \cdot (q-1)$. Essa função é utilizada para gerar a chave privada d, que é o inverso multiplicativo de um pequeno número e escolhido como chave pública, no módulo $\phi(m)$, ou seja, $d = e^{-1} \mod \phi(m)$.

Algoritmo 1 Geração de chave para criptografia de chave pública RSA

Dado: Parâmetro de segurança l.
Resultado: Chave pública RSA (m, e) e chave privada d.

1: Escolher dois números primos p e q do mesmo comprimento de bit l;
2: Gera-se um número m e calcula-se $m = pq$ e $\phi(m) = (p-1)(q-1)$;
3: Seleciona-se um número inteiro aleatório e;
4: Calcula-se o número inteiro d satisfazendo $1 < d < \phi(m)$ e $ed \equiv 1 \mod \phi(m)$;
5: Retorna: (m, e, d)

O sistema RSA se baseia na operação de exponenciação modular, que inclui multiplicações modulares como parte dos cálculos totais. A robustez do RSA é determinada pelo tamanho da chave, e para aplicações modernas, o tamanho mínimo da chave é de 1024 bits. Com tamanhos de chave grandes, muitos ciclos de clock são necessários para completar a multiplicação, e isso se torna ainda mais desafiador quando consideramos o tempo total necessário para a operação de exponenciação modular, como a exponenciação modular de Montgomery. No algoritmo RSA, os processos de criptografia e descriptografia são fundamentados na exponenciação modular, conforme resumido nas Equações 4.1 e 4.2.

A segurança matemática do algoritmo RSA depende da dificuldade de fatoração de números inteiros grandes, geralmente de 1024 a 4096 bits.

$$C = |M^e|_m \tag{4.1}$$

$$M = |C^d|_m \tag{4.2}$$

Na Equação 4.1, M representa a mensagem que será criptografada, C é a mensagem após a criptografia, E é o expoente, m é a chave pública, e $||m$ simboliza a operação de módulo m. Neste contexto, C, e e m são informações públicas, transmitidas juntamente com a mensagem criptografada através do canal de comunicação. A chave pública m é obtida ao selecionar dois grandes números primos p e q, de modo que m seja o produto de p e q. Já na descriptografia, ilustrada na Equação 4.2, a mensagem criptografada C e m são públicos e recebidos pelo canal de comunicação, enquanto a mensagem descriptografada M e a chave de descriptografia d são mantidas em privado.

As chaves de criptografia usadas têm 1024 bits, mas podem chegar a 4096 bits. Com a tecnologia atual, é praticamente impossível fatorar números inteiros desse tamanho. A maior dificuldade na implementação do RSA em hardware é criar uma arquitetura rápida que possa realizar a exponenciação modular com números inteiros grandes. A solução para isso é usar algoritmos matemáticos que ajudem em processos aritméticos mais complexos. Assim, a seguir, serão apresentados os métodos de exponenciação modular mais usados nos processos de criptografia e descriptografia do RSA.

4.1.4 Métodos da Exponenciação Modular

A exponenciação modular pode ser dividida em dois métodos, binário e m-ário, em que os métodos binários são divididos em *Left-to-Right* e *Right-to-Left* (MENEZES *et al.*, 1996). No cálculo da exponenciação modular para o algoritmo RSA, a segurança depende do tamanho da chave, e para aplicações atuais, o comprimento mínimo é de 1024 bits. Assim, métodos binários de exponenciação modular são utilizados, onde o expoente E é dado na base 2 binária e interpretado bit a bit. O cálculo é quebrado em várias multiplicações modulares $R_0 \cdot R_1 \mod m$ e quadrados modulares $R_0 \cdot R_0 \mod m$, sendo R_0 e R_1 resultados intermediários da exponenciação modular.

Algoritmo 2 Método Binário *Left-to-Right*

Input: X, m, E = $(e_{k-1}, ..., e_1, e_0)_2$
Output: $X^E \mod m$

1: $R_0 \leftarrow 1, R_1 \leftarrow X$;
2: **for** $i = k - 1$ **to** 0 **do**
3: $R_0 \cdot R_0 \mod m$;
4: **if** $e_0 = 1$ **then**
5: $R_0 \cdot R_1 \mod m$;
6: **end if**;
7: **end for**;
8: **Return:** R_0

Algoritmo 3 Método Binário *Right-to-Left*

Input: $X, m, E = (e_{k-1}, ..., e_1, e_0)_2$
Output: $X^E \mod m$

1: $R_0 \leftarrow 1, R_1 \leftarrow X$
2: **for** $i = 0$ to $k - 1$ **do**
3: **if** $e_i = 1$ **then**
4: $R_0 \cdot R_1 \mod m$
5: **end if**
6: $R_1 \cdot R_1 \mod m$
7: **end for**
8: **Return:** R_0

Os Algoritmos 2 e 3 mostrados, realizam uma operação de quadrado modular a cada vez que interpretam um bit do expoente e_i. Se o bit e_i for 1, uma multiplicação modular é realizada em sequência (da esquerda para a direita) ou em paralelo (da direita para a esquerda) com a operação de quadrado modular. Para o desenvolvimento da aplicação de criptografia utilizou-se o algoritmo *Left-to-Right* para expoentes com muitos bits, pois reduz o armazenamento prévio dos bits de representação binária.

4.2 APLICAÇÃO DA SENSITIVIDADE MODULAR À CRIPTOGRAFIA

Nesta seção pretende-se fazer o uso dos módulos auxliares explicados anteriormente para implementar uma exponenciação mais eficiente e no menor tempo possível. Os conjuntos de módulos propostos apresentaram sensitividade ao tipo de módulo tanto para o operador mod nativo como para a solução dedicada. Na seção 3.2, realizou-se a análise de sensitividade modular para o conjunto de módulos $2^n + k$, comparando-os com módulos auxiliares do tipo $2^n - 1$. Observou-se que os módulos $2^n - 1$ apresentaram um desempenho superior ao utilizar o operador mod nativo como a solução dedicada, conseguindo diminuir o atraso. O que se pretende agora, é aplicar essa metodologia para números de bits mais altos para aplicação criptográfica. Novamente, utilizaremos a Equação 3.1 previamente discutida para a análise na implementação. O objetivo é de aplicar módulos auxiliares de alta eficiência com correção final para um módulo original. Focar-se-á na aplicação, demonstrando como os módulos auxiliares do formato $2^n - 1$ podem ser utilizados para otimizar a exponenciação modular em contextos de alta demanda computacional, como na criptografia.

4.2.1 Aplicação para Criptografia

A segurança do algoritmo RSA depende do número inteiro positivo, que é o múltiplo de dois números primos grandes e precisos. Com isto, surge a problemática de fatorização desses números. O criptossistema RSA é baseado na dificuldade da fatoração de inteiros, pois se um adversário conseguir fatorar o módulo público m contido em uma chave pública RSA em seus fatores primos p e q, ele pode facilmente calcular a chave privada. Portanto, certos criptossistemas de chave pública são projetados de forma que derivar a chave privada da chave pública envolva fatorar um número grande, tornando computacionalmente inviável a realização dessa derivação. A fatorização de números inteiros consiste em dividir um número em um conjunto de números menores (fatores) que, quando multiplicados juntos, formam o número original. Encontrar os fatores de números muito grandes foi um desafio em termos computacionais neste trabalho. Ainda assim, conseguiu-se encontrar números que são fatorizados no formato $3 \times p \times q$. A Tabela 4.1 mostra os módulos auxiliares escolhidos e na Tabela 4.2 estão os módulos auxiliares e seus respetivos módulos originais no formato $2^n \pm k$. Nas tabelas abaixo, a notação na forma k_n indica que k é um número na sua forma decimal e n indica o número de 1's em sua representação binária. Esta notação foi adotada por se estar a lidar com números muito grandes na forma decimal, facilitando a análise.

Tabela 4.1 – Fatoração dos Módulos Auxiliares

Módulos Auxiliares (m_1)	Fatoração
$2^{38}-1$	$3 \times 174763 \times 524287$
$2^{62}-1$	$3 \times 715827883 \times 2147483647$
$2^{122}-1$	$3 \times 2305843009213693951 \times 768614336404564651$

Tabela 4.2 – Módulos e Módulos Auxiliares Utilizados para Teste em Hardware na aplicação Criptográfica

Módulo (m_2)	Valor 1's em k	c	Módulo Auxiliar (m_1)	Valor 1's em k
$2^{36}+k_{18}$	18	3	$2^{38}-1$	1
$2^{60}+k_{30}$	30	3	$2^{62}-1$	1
$2^{120}+k_{60}$	60	3	$2^{122}-1$	1

Foram analisados os módulos utilizados para testes de hardware. Após isso foi feita a análise e comparação dos atrasos dos módulos originais e auxiliares, tanto utilizando o operador mod nativo quanto a solução dedicada. Com base nesses resultados, foca-se em números de bits mais altos, conforme apresentado na tabela 4.2. Utiliza-se os valores de atraso obtidos anteriormente para realizar uma regressão linear, a fim de estimar os valores de atraso para módulos com maior número de bits. Os resultados dos atrasos são expostos nas Tabelas 4.3 para o operador mod nativo e 4.4 para a solução dedicada.

Tabela 4.3 – Resultados Módulos Auxiliares Operador MOD na Aplicação Criptográfica

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1)	Atraso	Ganho
$2^{36}+k_{18}$	13005	$2^{38}-1$	**5757**	**55,73%**
$2^{60}+k_{30}$	21388	$2^{62}-1$	**8957**	**58,12%**
$2^{120}+k_{60}$	42346	$2^{122}-1$	**16957**	**59,96%**

Avaliando o impacto da aplicação de sensitividade para módulos com n de bits extensos utilizando o operador mod, obteve-se ganhos médios de $57,94\%$ para o módulo 2^n-1. Resultados parecidos com o que analisamos na seção 3.2 em que os valores de n eram menores. Para solução dedicada, conseguiu-se ganhos médios de $79,19\%$ utilizando os módulos auxiliares no formato 2^n-1. Este ganho é atribuído à eliminação de 1's no formato de módulo 2^n-k, como relatados neste trabalho. Os módulos 2^n-1 são particularmente eficientes devido à simplificação das operações aritméticas envolvidas. A estrutura simplificada desses módulos permite operações de redução modular mais rápidas, reduzindo o tempo de processamento.

Tabela 4.4 – Resultados Módulos Auxiliares Solução Dedicada na Aplicação Criptográfica

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1)	Atraso	Ganho
$2^{36} + k_{18}$	8057	$2^{38} - 1$	**2154**	**73,27%**
$2^{60} + k_{30}$	12767	$2^{62} - 1$	**2614**	**79,53%**
$2^{120} + k_{60}$	24738	$2^{122} - 1$	**3765**	**84,78%**

Analisando o custo de correção para o operador nativo mod, a Tabela 4.5 mostra que obteve-se ganho em todos os módulos usados para a aplicação. Para o módulo $2^{36}+k_{18}$ conseguiu-se um ganho de $45,11\%$, para o módulo $2^{60} + k_{30}$ um ganho de $49,88\%$ e no módulo $2^{120} + k_{60}$ de $53,54\%$. De reforçar que esses ganhos são apenas para uma iteração.

Tabela 4.5 – Resultados de Correção do Atraso dos Módulos Auxiliares Operador MOD para Aplicação Criptográfica

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1) + Correção	Atraso	Ganho
$2^{36} + k_{18}$	13005	$2^{38} - 1$	**7138**	**45,11%**
$2^{60} + k_{30}$	21388	$2^{62} - 1$	**10719**	**49,88%**
$2^{120} + k_{60}$	42346	$2^{122} - 1$	**19672**	**53,54%**

Da mesma que obteve-se ganho no custo de correção para todos os módulos usando o operador mod, o mesmo pode-se observar na implementação pela solução dedicada. A Tabela 4.6 mostra os resultados. Para o módulo $2^{36}+k_{18}$ conseguiu-se um ganho de $59,67\%$, para o módulo $2^{60} + k_{30}$ um ganho de $70,10\%$ e no módulo $2^{120} + k_{60}$ de $78,81\%$. Estes ganhos são apenas para uma iteração. Fazendo mais iterações os ganhos se acumulariam. De notar ainda que os ganhos melhoram de acordo com o número de bits no expoente.

Tabela 4.6 – Resultados de Correção do Atraso dos Módulos Auxiliares Solução Dedicada para Aplicação Criptográfica

Módulo (m_2)	Atraso	Módulo Auxiliar (m_1) + Correção	Atraso	Ganho
$2^{36} + k_{18}$	8057	$2^{38} - 1$	**3249**	**59,67%**
$2^{60} + k_{30}$	12767	$2^{62} - 1$	**3818**	**70,10%**
$2^{120} + k_{60}$	24738	$2^{122} - 1$	**5242**	**78,81%**

5 CONCLUSÃO

Este trabalho teve como objetivo explorar a implementação em ASIC do operador nativo à linguagem VHDL para as operações indicadas, em comparação com soluções específicas de otimização presentes na literatura. Além disso, buscou-se investigar a presença de sensitividade do operador módulo aos diferentes tipos de módulos 2^n, $2^n \pm 1$ e $2^n \pm k$, bem como buscar otimizações para operações modulares de soma (+), multiplicação (*) e conversão modular explorando módulos auxiliares.

Os resultados obtidos na soma modular demonstraram que, embora as abordagens baseadas na solução dedicada e no operador módulo apresentem resultados similares, a solução dedicada demonstra desempenho superior, com ganhos médios de **2,81%** para o módulo $2^n + 1$, **15%** para $2^n - 1$ e **5,54%** para o módulo $2^n + k$, sendo mais evidente à medida que se aumenta o número de bits.

Na multiplicação modular, com o intuito de obter redução de atraso de circuitos, para o conjunto modular $2^n - 1$ conseguiu-se uma média de ganhos de **17,54%**, **15,7%** para $2^n + 1$ e **28,86%** para módulos $2^n \pm k$. Esses valores de ganho são relativos à melhoria utilizando a solução dedicada. A multiplicação baseada na solução dedicada apresenta maior sensitividade e permite a possibilidade de aplicações de módulos auxiliares.

No que diz respeito aos módulos auxiliares, os resultados alcançados utilizando o operador mod demonstram que, os módulos auxiliares no formato $2^n - 1$, apesar de serem maiores (com maior número n de bit), possuem um menor atraso em comparação com os formatos $2^n + k$, demonstrando resultado satisfatório com ganhos médios de **31,5%**. Analisando os resultados para a operação modular utilizando a solução dedicada, destaca-se novamente menor atraso após a correção feita para módulos $2^n - 1$ com ganhos médios de **33,9%**. Ao aplicar a sensitividade vista na seção 4.2 para módulos com n (número de bits) extensos, os ganhos atingiram a média de **57,94%** utilizando o operador mod e **79,19%** de ganho para a solução dedicada.

Este trabalho contribuiu para a compreensão do operador nativo à linguagem VHDL e das possíveis otimizações para operações modulares. Os resultados obtidos indicam que a solução dedicada pode oferecer desempenho superior em comparação com o operador módulo, especialmente quando se utilizam módulos auxiliares.

5.1 SUGESTÕES PARA TRABALHOS FUTUROS

Como sugestão para trabalhos futuros, sugere-se a possibilidade de analisar os resultados de desempenho de potência das duas abordagens de implementação discutidas: o operador mod nativo e a solução dedicada.

Sugere-se também, expandir a análise para módulos usados neste trabalho com um número maior de bits. Esta expansão permitirá avaliar a escalabilidade das técnicas propostas e identificar possíveis limitações ou vantagens em cenários de alta complexidade.

REFERÊNCIAS

BHARDWAJ, Isha; KUMAR, Ajay; BANSAL, Manu. A review on lightweight cryptography algorithms for data security and authentication in IoTs. **2017 4th International Conference on Signal Processing, Computing and Control (ISPCC)**, p. 504–509, 2017. Disponível em: https://api.semanticscholar.org/CorpusID:28375731.

BOREALE, Michele; BUSCEMI, Maria Grazia. On the symbolic analysis of low-level cryptographic primitives: Modular exponentiation and the Diffie-Hellman protocol. *In:* FOUNDATIONS of Computer Security. [*S.l.*: *s.n.*], 2003. P. 85.

CARDARILLI, Gian Carlo; NANNARELLI, Alberto; RE, Marco. Reducing power dissipation in FIR filters using the residue number system. *In:* 320–323 vol.1. DOI: 10.1109/MWSCAS.2000.951651.

DIFFIE, Whitfield; HELLMAN, Martin E. New Directions in Cryptography. **Democratizing Cryptography**, 1976. Disponível em: https://api.semanticscholar.org/CorpusID:275548.

FERNANDES, G. B. M. **Infraestrutura automática para aritmética computacional baseada em RNS**. 2021. Diss. (Mestrado) – Universidade Federal de Santa Catarina.

GAUSS, Carl Friedrich. **Disquisitiones Arithmeticae**. Edição: William C. Waterhouse. [*S.l.*]: Springer-Verlag, 1986. ISBN 9783540962540. Disponível em: https://books.google.fr/books?id=Y-49PgAACAAJ.

LI, Zhi; HIGGINS, John; CLEMENT, Mark J. Performance of finite field arithmetic in an elliptic curve cryptosystem. **MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**, p. 249–256, 2001. Disponível em: https://api.semanticscholar.org/CorpusID:10984074.

MENEZES, A.J. *et al.* **Handbook of Applied Cryptography**. [*S.l.*]: CRC Press, 1996. (Discrete Mathematics and Its Applications). ISBN 9781439821916. Disponível em: https://books.google.com.br/books?id=nSzoG72E93MC.

MÖLLER, Bodo; RUPP, Andy. Faster multi-exponentiation through caching: accelerating (EC) DSA signature verification. *In:* SPRINGER. INTERNATIONAL Conference on Security and Cryptography for Networks. [*S.l.*: *s.n.*], 2008. P. 39–56.

PAAR, C.; PELZL, J. **Understanding Cryptography: A Textbook for Students and Practitioners**. [*S.l.*]: Springer Berlin Heidelberg, 2009. ISBN 9783642041013. Disponível em: https://books.google.com.br/books?id=f24wFELSzkoC.

PALUDO, R. **Implementações eficientes de conversores reversos e multiplicações por constante usando residue number systems**. 2020. Tese de Doutorado – Universidade Federal de Santa Catarina.

PARHAMI, Behrooz. **Computer Arithmetic: Algorithms and Hardware Designs**. 2nd. USA: Oxford University Press, Inc., 2009. ISBN 0195328485.

PLANTARD, Thomas. Efficient Word Size Modular Arithmetic. **IEEE Transactions on Emerging Topics in Computing**, v. 9, p. 1506–1518, 2021. Disponível em: `https://api.semanticscholar.org/CorpusID:235875462`.

RIVEST, Ronald L.; SHAMIR, Adi; ADLEMAN, Leonard M. A method for obtaining digital signatures and public-key cryptosystems. **Commun. ACM**, v. 21, p. 120–126, 1978. Disponível em: `https://api.semanticscholar.org/CorpusID:2873616`.

SHIEH, Ming-Der *et al.* A new modular exponentiation architecture for efficient design of RSA cryptosystem. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, IEEE, v. 16, n. 9, p. 1151–1161, 2008.