Universidade Federal de Santa Catarina
CTC - Centro Tecnológico
Programa de Pós-Graduação em Engenharia Elétrica

Fábio de Araújo Bairros

EMPREGO DE TÉCNICAS DE SISTEMAS NUMÉRICOS NA OTIMIZAÇÃO ALGORÍTMICA DA EXPONENCIAÇÃO MODULAR APLICADA À CRIPTOGRAFIA TRADICIONAL - HARDWARE/SOFTWARE

Florianópolis/SC

2024

EMPREGO DE TÉCNICAS DE SISTEMAS NUMÉRICOS NA OTIMIZAÇÃO ALGORÍTMICA DA EXPONENCIAÇÃO MODULAR APLICADA À CRIPTOGRAFIA TRADICIONAL - HARDWARE/SOFTWARE

Fábio de Araújo Bairros

Dissertação de Mestrado submetida ao Programa de Mestrado em Engenharia Elétrica e Eletrônica (PPGEEL), da Universidade Federal de Santa Catarina para a obtenção do título de Mestre em Engenharia Elétrica.
Orientador: Héctor Pettenghi Roldán

Florianópolis/SC

2024

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

de Araújo Bairros, Fábio
EMPREGO DE TÉCNICAS DE SISTEMAS NUMÉRICOS NA OTIMIZAÇÃO ALGORÍTMICA DA EXPONENCIAÇÃO MODULAR APLICADA À CRIPTOGRAFIA TRADICIONAL - HARDWARE/SOFTWARE / Fábio de Araújo Bairros ; orientador, Héctor Pettenghi Roldán, 2024.
83 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia Elétrica, Florianópolis, 2024.

Inclui referências.

1. Engenharia Elétrica. 2. Exponenciação modular. 3. Criptografia. 4. Síntese hardware. 5. Algorítmo. I. Pettenghi Roldán, Héctor. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

Fábio de Araújo Bairros

EMPREGO DE TÉCNICAS DE SISTEMAS NUMÉRICOS NA OTIMIZAÇÃO ALGORÍTMICA DA EXPONENCIAÇÃO MODULAR APLICADA À CRIPTOGRAFIA TRADICIONAL - HARDWARE/SOFTWARE

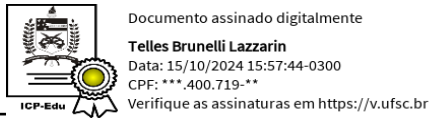
O presente trabalho em nível de mestrado foi avaliado e aprovado, em 30/09/2024 por banca examinadora composta pelos seguintes membros:

Prof. Héctor Pettenghi Roldán, Dr.
Universidade Federal de Santa Catarina

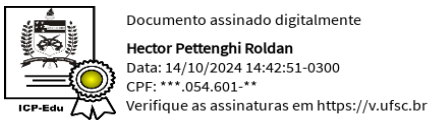
Prof. Eduardo Augusto Bezerra, Dr.
Universidade Federal de Santa Catarina

Prof. Roberto de Matos, Dr.
Instituto Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia Elétrica.



Coordenação do Programa de Pós-Graduação



Prof. Héctor Pettenghi Roldán, Dr.
Orientador

Florianópolis, 2024 .

Agradecimentos

Agradeço a todos que direta e/ou indiretamente me auxiliaram nesta jornada. Bem como as instituições que são constituídas de pessoas impulsionadas por objetivos de aprimoramento contínuo.

Resumo

A presente pesquisa objetivou abranger análises dos tempos de resposta algorítmica da exponenciação modular confrontando módulos que apresentam congruências, aqui denominados de auxiliar e original. Além de atuar na exploração de soluções para aritmética em *hardware* com ações para reduzir o custo e tempo de resposta na multiplicação de variável por variável, emprego dos módulos auxiliares na aritmética modular (multiplicação por constantes e exponenciação modular) e realocação das constantes reversas (vetores) nas operações de conversão entre sistemas numéricos, utilizando um sistema de representação numérica baseado no resíduo, conhecido como RNS (Residual Number System). Para isso, lançou-se mão de ferramentas e técnicas computacionais na medição de tempo, apoiadas nas linguagens computacionais de alto nível como Python, Java e Go Lang. Diante dos resultados obtidos, detentores de relativa robustez a partir de um determinado grau do expoente, conclui-se que há lacunas para exploração desta abordagem e cuja contribuição futura atinge o desenvolvimento de bibliotecas e/ou funções em *software*, bem como arranjos dos núcleos da aritmética de *hardware* em arquiteturas já consagradas.

Palavras-chave: exponenciação modular, congruências, algorítmica, módulos, auxiliar, original, *software*, *hardware*.

Abstract

The present research aimed to cover analyzes of the algorithmic response times of modular exponentiation comparing modules that present congruence, here called auxiliary and original. In addition to working on the exploration of solutions for arithmetic in hardware with actions to reduce the cost and response time in multiplying variable by variable, use of auxiliary modules in modular arithmetic (multiplication by constants and modular exponentiation) and reallocation of reverse constants (vectors) in conversion operations between number systems, using a numerical representation system based on the residue, known as RNS (Residual Number System). To achieve this, computational tools and techniques were used to measure time based on high-level computer languages such as Python, Java and Go Lang. Given the results obtained, which have relative robustness from a certain degree of the exponent, it is concluded that there are gaps in the exploration of this approach and whose future contribution reaches the development of libraries and/or functions, at the software level, as well as arrangements in hardware arithmetic cores within already established architectures.

Keywords: modular exponentiation, congruence, algorithmic, modules, auxiliary, original, software, hardware.

Lista de ilustrações

Figura 2.1 – Arquiterura RNS para os módulos $\{m_1\ ,\ m_2\ ,\ m_3\}$ 6
Figura 2.2 – Multiplicador variável-variável para os módulos 2^n, $2^n \pm 1$ e $2^n \pm k$. . . 6
Figura 2.3 – Circuito básico da conversão direta binário-RNS para os módulos $\{2^n \pm 1\}$ 8
Figura 2.4 – Diagrama em blocos do funcionamento para o somador baseado em CSA & EAC. 9
Figura 2.5 – Diagrama em blocos do funcionamento para o somador baseado em CPA para a operação $|A + B|_{15}$. 9
Figura 2.6 – Diagrama em blocos do funcionamento para o somador baseado em CSA & IEAC (*Inverted End-Around Carry*). 10
Figura 2.7 – Diagrama em blocos do funcionamento para o somador baseado em CPA para a operação $|A + B|_{17}$. 10
Figura 2.8 – Soma modular genérica para operação $|A + B|_m$. 11
Figura 2.9 – Multiplicação modular genérica para operação $|AxB|_m$. 12
Figura 2.10–Multiplicação modular para $m = 2^n$, para $n = 4$. 12
Figura 2.11–Multiplicação modular para $m = 2^n$, para $n = 4$. No formato do diagrama de pontos. 13
Figura 2.12–Multiplicação modular para $m = 2^n - 1$, para $n = 4$. 13
Figura 2.13–Multiplicação modular para $m = 2^n - 1$, para $n = 4$. No formato do diagrama de pontos. 14
Figura 2.14–Multiplicação modular para $m = 2^n + 1$, para $n = 4$. 14
Figura 2.15–Multiplicação modular para $m = 2^n + 1$, para $n = 4$. No formato do diagrama de pontos. 15
Figura 2.16–Diagrama do conversor RNS-binário no formato CRT. Para módulos m_1, m_2 e m_3. 16
Figura 2.17–Diagrama em blocos do conversor RNS-binário usando CRT I. Para os módulos $\{2^n\ ,\ 2^n \pm 1\}$. 17
Figura 2.18–Diagrama em blocos do conversor MRC para os módulos m_1, m_2 e m_3. 19
Figura 3.1 – *Delay* da multiplicação modular para 2^n, $2^n \pm 1$ e $2^n \pm k$. 23
Figura 3.2 – *Delay* por número de operações MAC. 25
Figura 3.3 – Número de operações MAC por ciclos de *clock*. 25
Figura 3.4 – Operação XC em RNS, onde as multiplicativas inversas que estão feitas em (a), ocorrem no CRT e, em (b), na aritmética (proposta). 28
Figura 3.5 – Exponenciação modular comparação de tempos. 32
Figura 4.1 – Ensaio com Python para a operação módulo simples, variando o k. . . 36
Figura 4.2 – Medição de tempos da operação módulo simples para t_{AUX} e t_{ORI}, no formato $2^n - 1$ e com X no formato do ensaio C. Linguagem Python. . 37

Figura 4.3 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI} e t_{COR}. Linguagem Python. 38
Figura 4.4 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI}, t_{COR} e $m_{AUX} = 2^n + 1$. Na linguagem Java. 39
Figura 4.5 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI}, t_{COR} e $m_{AUX} = 2^n + 1$. Na linguagem Go. 40
Figura 4.6 – Operações modulares empregadas e suas respectivas equivalências e correções. 41
Figura 4.7 – Medicação do tempo da função de soma simples usando recursos do paralelismo. 43
Figura 4.8 – Paralelismo: comparação entre os tempos dos módulos m_{ORI}, m_{AUX} e sua fatoração m_{ab}. 43
Figura 5.1 – Exemplo de aplicação da criptografia RSA. 47
Figura 5.2 – Operação módulo simples, exemplos similares ao RSA-100, para 10 repetições. 49
Figura 5.3 – Paralelismo para exponenciação modular em Python. Para t_{ab} sem as correções. 50
Figura 5.4 – Paralelismo para exponenciação modular em Python, variando Y para n com 330 bits e t_{ab} sem as correções. 51
Figura 5.5 – Medições dos tempos t_{ORI}, $t_{COR_{ab}}$ e t_{AUX} com os ns da tabela RSA, para $ny = 200$ (ny_e). 52
Figura A.1 – Diagrama organização das tarefas 60
Figura C.1 – Configuração genérica da CPU na plataforma desktop. 63
Figura C.2 – Relatório da configuração detalhada da CPU na plataforma desktop. . 64
Figura C.3 – Relatório de usuários logados no SO Linux na plataforma desktop. . . . 64
Figura C.4 – Tarefas paralelas no ambiente Ubuntu-Linux. 64
Figura C.5 – Gerenciamento de deamons no ambiente Ubuntu-Linux. 65

Lista de tabelas

Tabela 3.1 – Valores dos módulos para conversão reversa em $M_{5n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+1} \pm 1\}$, $M_{6n} = \{M_{5n},\ 2^{n+2} - 1\}$ e $M_{6n} = \{M_{5n},\ 2^{n+2} + 1\}$. 22

Tabela 3.2 – Valores dos módulos para conversão reversa em $M_{7n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+2} \pm 1\ 2^{n+2} \pm 1\}$, $M_{8n} = \{M_{7n},\ 2^{n+3} - 1\}$ e $M_{8n} = \{M_{7n},\ 2^{n+3} + 1\}$. 22

Tabela 3.3 – Valores dos módulos para conversão reversa em $M_{9n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+2} \pm 1\ 2^{n+3} \pm 1\}$, $M_{10n} = \{M_{9n},\ 2^{n+4} - 1\}$ e $M_{10n} = \{M_{9n},\ 2^{n+4} + 1\}$. 23

Tabela 3.4 – Comparação entre C_i e P_i avaliando área e *delay*, (CONWAY, 2006). Exemplos para os módulos 127, 63 e 31 29

Tabela 3.5 – Distribuição das constantes sem o emprego das combinações 30

Tabela 3.6 – Distribuição das constantes com o emprego das combinações. 30

Tabela 3.7 – Algoritmo de exponenciação rápida aplicado a um exemplo numérico. . 32

Tabela 5.1 – Operação módulo simples com exemplo similar a RSA-100, t_{ORI} *versus* t_{AUX}, para $Y = 2047$. 48

Tabela 5.2 – Ensaios com exemplos similares ao RSA e os respectivos ganhos. Na Linguagem Python para $Y = 2047$. 49

Tabela 5.3 – Expoentes (ny), utilizados para os pontos de inflexão e de extrapolação de ganho. 52

Tabela 5.4 – Valores de t_{ORI} e $t_{COR_{ab}}$ para valores similares de RSA. Utilizando os valores de $ny = 200$ (ny_e). 52

Tabela B.1 – Exemplo do formato de coleta dos dados e o efeito de carga do pipeline 62

Lista de abreviaturas e siglas

RNS: Residual Number System

DSP: Digital System Processing

SO: Sistema Opercional

UMC: United Microelectronics Corporation

VHSIC: Very High Speed Integrated Circuit

VHDL: VHSIC Hardware Description Language

CRT: Chines Remaider Theorem

MRC: Mixed Radix Conversion

RSA: Rivest Shamir and Adleman

DR: Dynamic Range

CSA: Carry Save Adder

EAC: End-Around Carry

IEAC: Inverted End-Around Carry

CPA: Carry Propagate Adder

ASIC: Application-specific Integrated Circuit

FPGA: Field Programmable Gate Array

MAC: Multiply–accumulate operation

ROM: Read-Only Memory

ASCII: American Standard Code for Information Interchange

IA: Inteligência Artificial

Sumário

1 INTRODUÇÃO 1
1.1 Motivação 1
1.2 Objetivo geral 1
1.2.1 Objetivos específicos 2
1.3 Metodologia 2
1.4 Discussões, conclusões e detalhes 3

2 FUNDAMENTAÇÃO TEÓRICA 4
2.1 Motivação 4
2.2 Operação, estrutura modular e DR (*Dynamic Range*) em RNS . . . 4
2.3 Conversão binário-RNS 7
2.4 Aritmética RNS 10
2.5 Conversão RNS-binário 15
2.5.1 Detalhando o CRT e CRT I 15
2.5.2 Detalhando o MRC 18

3 PROPOSTAS EM *HARDWARE* COM USO DE RNS E CRIPTOGRAFIA RSA 20
3.1 Motivação 20
3.2 Aplicação de módulo auxiliar de baixo custo na aritmética RNS . . 20
3.2.1 Módulo auxiliar 20
3.2.2 Realocação de constantes nas conversões RNS-binário 26
3.2.2.1 Aplicação no CRT 26
3.3 Aplicações para o módulo auxiliar de baixo custo nas operações de criptografia 30
3.3.1 Motivação 30
3.3.2 Algoritmo para exponenciação modular rápida 30
3.3.3 Potencializando com módulo auxiliar 32

4 MÓDULO DE MELHOR DESEMPENHO EM APLICAÇÕES DE *SOFTWARE*. 34
4.1 Motivação 34
4.2 Análise de sensitividade modular em linguagens de software 35
4.2.1 Uso da linguagem Python na operação de módulo simples $X \ \% \ m$ 35
4.2.1.1 Ensaio comparativo da operação de módulo simples com módulo $2^n - 1$ 36
4.2.1.2 Ensaio comparativo da operação de módulo simples com módulo $2^n + 1$ 37

4.2.2 Java - operação módulo - (X).mod(m) . . . 38
4.2.3 Go Lang - operação módulo - .mod(X, m) . . . 39
4.2.4 Operação modular com fatoração de termos . . . 40

5 MÓDULO DE MELHOR DESEMPENHO EM APLICAÇÕES CRIPTOGRÁFICAS DE SOFTWARE . . . 45
5.1 Motivação . . . 45
5.2 A criptografia RSA . . . 45
5.3 Resultados experimentais para RSA usando módulo de melhor desempenho . . . 47
5.3.1 Exemplo de RSA usando módulos auxiliares . . . 47
5.3.2 RSA-100 . . . 47
5.3.3 Exemplo de exponenciação modular usando módulos auxiliares decompostos (fatorados) . . . 49

6 DISCUSSÕES, TRABALHOS FUTUROS E CONCLUSÕES . . . 54

REFERÊNCIAS . . . 57

APÊNDICE A – MEDIÇÃO DOS TEMPOS . . . 60
A.1 Coleta das informações . . . 60
A.2 Análise . . . 61
A.3 Configurações . . . 61
A.4 Medições . . . 61

APÊNDICE B – TRATAMENTO ESTATÍSTICO . . . 62

APÊNDICE C – SISTEMA OPERACIONAL E HARDWARE . . . 63

APÊNDICE D – USO DO SHELL SCRIPT . . . 66
D.1 Exemplo de chamada do programa de medição na linguagem Python 66
D.2 Exemplo de chamada do programa de medição na linguagem Java . 66
D.3 Exemplo de chamada do programa de medição na linguagem Go . . 66

APÊNDICE E – COLETAS DE DADOS EM PYTHON . . . 68

APÊNDICE F – COLETAS DE DADOS EM JAVA . . . 69

APÊNDICE G – COLETAS DE DADOS EM GO . . . 70

1 INTRODUÇÃO

Apresentar uma alternativa aos algoritmos e operações tradicionais de criptografia tanto em *hardware* como *software*, referente à exponenciação modular com o emprego de módulos de baixo custo, é o que se propõe o trabalho a seguir. Para tanto foram desenvolvidas ferramentas computacionais de análise dos tempos de execução, aplicando algumas técnicas de otimização dos algoritmos e avaliando estatísticas dos resultados. Os algoritmos foram implementados na linguagem Python (ROSSUM; DRAKE, 2009), estabelecendo-se também comparativos com linguagens mais clássicas como Java (MICROSYSTEMS, 2024) e a relativamente mais recente Go Lang (GOOGLE, 2024). A sequência de estudos se inicia com ensaios realizados em *hardware* que posteriormente motivam as avaliações em *software*, com a utilização das linguagens mencionadas anteriormente. Nos estudos em *hardware* são oferecidas contribuições nas operações de multiplicação por constantes, nas conversões reversas (da representação residual para a binária) e nas operações de exponenciação modular. Nos estudos em *software* as potenciais contribuições estão concentradas nas operações modulares simples e de exponenciação modular, sendo esta última demonstrada praticamente em cálculos criptográficos. Os resultados obtidos dão conta de que existe uma relativa margem de aproveitamento em área e redução no tempo de operações, referentes ao *hardware*, e melhoria no tempo de execução de algoritmos em *software*, e que a mesma aparece em um percentual considerável de casos.

1.1 Motivação

Motivado pelas necessidades de redução de custo relativa à área ocupada (na questão do *hardware*) como ao *delay* (em *hardware*) e tempo de execução (em *software*) o trabalho se propõem a apresentar alternativas às formas atuais de computação envolvendo aritmética modular, seja por operações modulares simples como também nas operações de exponenciação modular. Incluindo também aplicações de cálculos criptográficos empregados na atualidade, promovendo uma contribuição significativa no desempenho das operações internas e abrindo novas perspectivas de estudos futuros.

1.2 Objetivo geral

Propor soluções alternativas que possam atender uma demanda crescente pela otimização das operações aritméticas em *hardware* e *software*. Principalmente com relação às aplicações que trabalham com operações modulares simples e de exponenciação modular.

1.2.1 Objetivos específicos

Para atender de forma ampla os objetivos gerais propõem-se algumas metas intermediárias, sendo descritas a seguir.

- Realizar análises, em *hardware*, para operações de multiplicação modular por constantes e propor soluções alternativas. Tanto em RNS como na aritmética modular;
- Analisar e explorar, em *hardware*, as operações aritméticas modulares simples;
- Empregar o conceito de módulo auxiliar de baixo custo e avaliar seu desempenho nas operações aritméticas modulares em *hardware*;
- Ainda em *hardware*, avaliar e explorar as operações de exponenciação modular, comparando com e sem o uso de módulos auxiliares;
- Em *software*, apresentar o emprego de módulos auxiliares nas funções modulares simples e de exponenciação modular com o uso de linguagens comercialmente empregadas, com aplicação voltada para criptografia digital.

1.3 Metodologia

Para as análises em *hardware* foram utilizadas ferramentas de projeto e síntese de circuitos integrados baseados na tecnologia $65nm$. Ainda em *hardware*, os estudos fazem comparações com o que é empregado como estado da arte da atualidade e avalia criticamente as vantagens do emprego da representação RNS nas operações aritméticas.

A sequência dos estudos e ensaios em *hardware* passam pela avaliação das melhores alternativas em termos da conversão e aritmética modulares, empregando e comparando técnicas já empregadas na atualidade. Em seguida é demonstrada a técnica de melhor aproveitamento da operação de multiplicação por constantes, usando propriedades da aritmética modular e finalizando com avaliações do uso de módulos auxiliares para as operações de exponenciação modular, cuja aplicabilidade prática se dá na atividade criptográfica.

Para a realização dos ensaios e análises, em *software*, foram empregadas técnicas de medição de tempo com o auxílio de comandos ou funções e ferramentas estatísticas. As linguagens de programação utilizadas foram Python, Java e Go (ou Go Lang), com *scripts* desenvolvidos em Shell Script, no sistema operacional Ubuntu Minimal para *desktops*.

Em *software*, as comparações se dão com o emprego de comandos e funções, das linguagens escolhidas, aplicados às operações modulares simples e de exponenciação modular (usadas em criptografia computacional). Estas operações são executadas em

dois módulos que possuem uma correlação matemática e que produzem um resultado compatível (com os devidos ajustes e correções necessários).

Nas linguagens de programação utilizadas foram realizados ensaios que avaliam faixas de valores (explorando sua representação binária) que estão mais alinhadas com as atividades criptográficas da atualidade. Para avaliar um comportamento de tendência, também foram testadas faixas que extrapolam tais valores a fim de se obter uma visão mais ampla do comportamento de determinada linguagem.

Também foram empregadas técnicas para execução dos *scripts* das linguagens que envolvem abordagens como paralelismo e a execução de funções de forma encadeada.

Em paralelismo, os recursos empregados pertencem ao próprio sistema operacional utilizado (Ubuntu-Linux). Já na abordagem encadeada, as aplicações utilizaram chamadas das funções numa única linha, passando como parâmetros novas chamadas de funções intermediárias.

O tratamento estatístico final implantou os conceitos de média, quartis e avaliação de possíveis *outliers* com potencial para mascarar valores mais consistentes das amostras obtidas proporcionando, como resposta final, uma média que representa de maneira mais fiel possível os valores dos tempos de cada execução. Como pode ser visto na seção do apêndice B.

1.4 Discussões, conclusões e detalhes

Na seção de discussões e trabalhos futuros são apresentadas análises críticas do trabalho realizado, apontando algumas limitações tecnológicas de cada abordagem e propondo novos ângulos de visão com novas abordagens na resolução da mesma problemática. Abordagens estas que determinarão uma extensão maior das avaliações nesta temática ou outras co-relacionadas.

Na conclusão do trabalho é apresentado um resumo dos pontos vantajosos obtidos tanto para as aplicações em *hardware* e *software*, bem como sua aplicabilidade nas atividades empregadas na atualidade. Nas aplicações de *hardware* há indicações mais voltadas para sistemas embarcados, já em *software*, as aplicações podem ser implementadas nas diversas plataformas existentes.

Nos apêndices são apresentados, em detalhes, a forma como a coleta, análise, configuração e medição dos tempos foram realizadas, o tratamento estatístico empregado (apêndice B), detalhes específicos do sistema operacional empregado e da linguagem de *scripts* integrada ao sistema operacional. Também são apresentados detalhes do algoritmo básico usado nos testes com a linguagem de programação Python, Java e Go Lang.

2 Fundamentação teórica

Neste capítulo aborda-se o embasamento teórico necessário para o desenvolvimento dos estudos e ensaios realizados no capítulo que trata das propostas e abordagens práticas. Nele encontram-se descritos os fundamentos da operação e estrutura modular, princípios da representação RNS, modelo de processador RNS, conversão binário-RNS, aritmética modular e conversão RNS-binário. Os modelos matemáticos, diagramas e circuitos são baseados em módulos mais genéricos e no formato 2^n, $2^n - 1$ e $2^n + 1$, preparando para as propostas em RNS apresentadas no capítulo 3.

2.1 Motivação

Os esforços para otimização de sistemas computacionais (*hardware* ou *software*), sejam em circuitos embarcados ou em plataformas *desktop*, ou mesmo processamento em nuvem, sempre possuíram como um dos objetivos norteadores as otimizações das sínteses de *hardware* e a melhora no desempenho dos algoritmos. Ambos motivados pela eficiência energética e redução no *delay* das operações. Dentro desta linha de motivação, a presente dissertação se propõem a apresentar os estudos realizados, em *hardware*, com o auxílio da síntese ASIC (*Application-specific Integrated Circuit*) e FPGA (*Field Programmable Gate Array*), demonstrando uma proposta de algoritmos alternativos que usam sistemas numéricos como o RNS. Empregando também abordagens no uso de módulos auxiliares, com o objetivo de minimizar o *delay* das operações aritméticas modulares e também uma abordagem que emprega realocação de coeficientes nas multiplicações modulares por constantes.

Nos últimos tempos, as pesquisas envolvendo o RNS tem se ampliado, principalmente no aproveitamento de sua característica natural que é o paralelismo. Paralelismo este que aumenta o potencial computacional de sistemas aritméticos digitais (SZABO; TANAKA, 1967). Operações como soma, subtração e multiplicação podem ser processadas sem a dependência do sinal de saída *carry*, fato que ocorre no sistema binário de complemento de 2. Dentre alguns campos de atuação estão o DSP (*Digital System Processing*), (CHANG et al., 2015), os filtros digitais e transformadas de Fourrier; e na segurança e confiabilidade digital demandadas pela criptografia.

2.2 Operação, estrutura modular e DR (*Dynamic Range*) em RNS

Para o entendimento mais completo do que vem a ser a representação RNS, o conceito da operação modular e DR são relevantes e estão representados nas equações 2.1

e 2.2, respectivamente.

$$A \equiv \mid \alpha \mid_b, \tag{2.1}$$

onde A é o resíduo do operador α quando aplica-se o módulo b.

As operações modulares podem ser feitas em paralelo e convertidas de volta para o formato binário em processadores que usam numeração residual (RNS). Os módulos envolvidos devem ser co-primos entre si e a multiplicação destes módulos, num sistema RNS, determina a faixa dinâmica da operação em RNS, conforme indicado na equação 2.2.

$$DR = \prod_{i=1}^{n} m_i = m_1 \times m_2 \times ... \times m_n, \tag{2.2}$$

onde DR é a faixa dinâmica e m_is são os módulos envolvidos.

Na figura 2.1 há uma representação da arquitetura RNS (processador RNS) básica dividida em 3 estágios.

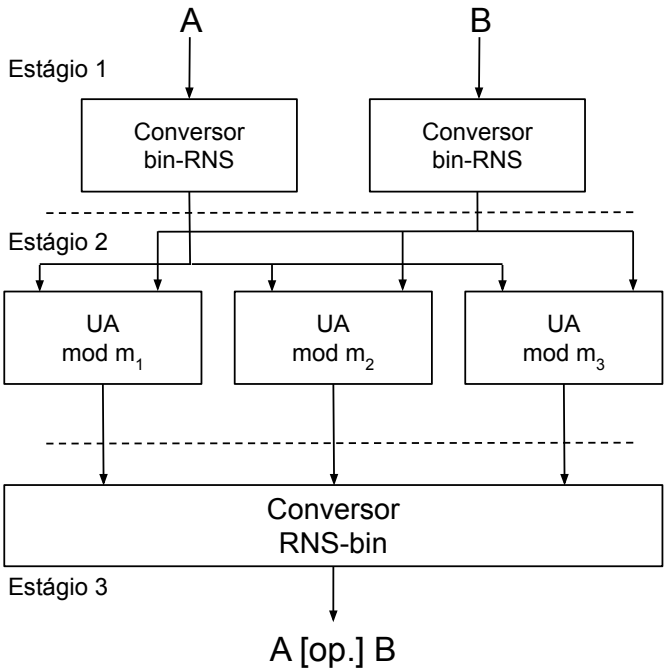
- Estágio 1 representa a conversão direta binário-RNS onde os operadores A e B sofrem suas respectivas conversões para os n módulos (no caso da figura 2.1, são 3 módulos);
- Estágio 2 são as operações aritméticas já no formato RNS. Onde UA significa Unidade Aritmética;
- Estágio 3 indica a conversão reversa para o formato binário e cuja resposta apresenta-se em $A[op.]B$, que pode representar as operações de soma, subtração e multiplicação.

Dentro desta linha de pesquisa e com embasamento em uma série de estudos de análise comparativa da eficiência do uso de RNS, principalmente aplicando um conjunto de módulos determinados por 2^n, $2^n \pm 1$ e $2^n \pm k$, iniciaram-se os testes com análise em aplicações de *hardware* demonstrados no gráfico da figura 2.2.

Na realização desta análise utilizou-se da tecnologia $65nm$ Standarts Cells da UMC (United Microelectronics Corporation) (TSAI; ZHOU, 2006) e a ferramenta utilizada foi a Genus Synthesis Solution (versão 16.24-S065-1). Nesta análise foram realizadas as sínteses dos conjuntos de módulos mencionados anteriormente variando o número de *bits* dos operadores envolvidos, na operação de multiplicação, e observando o *delay* obtido.

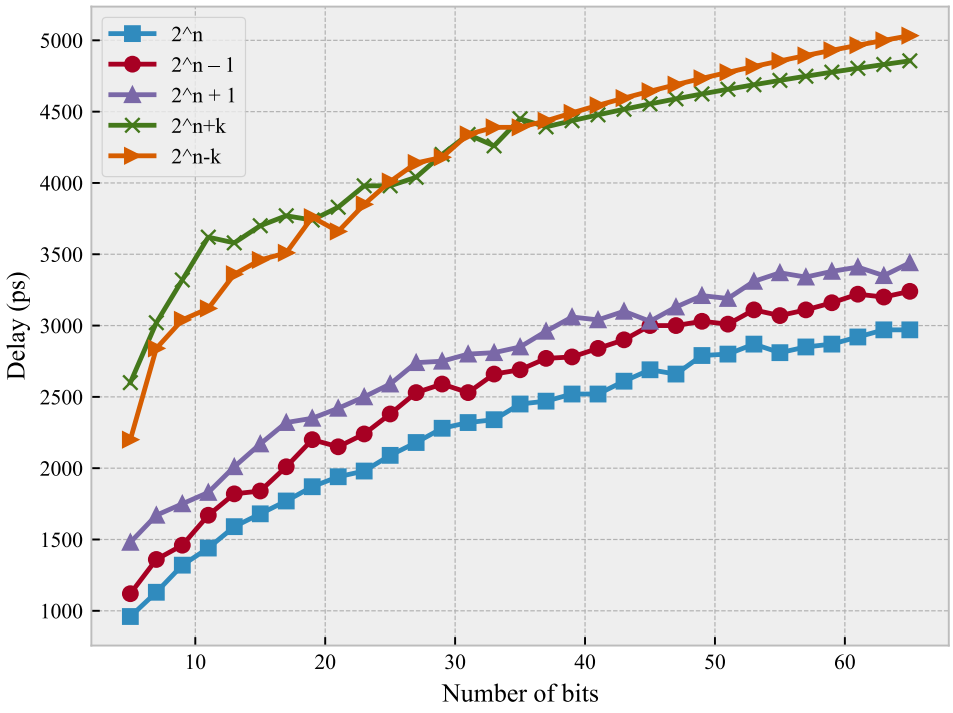
Na figura 2.2 pode-se verificar que existe, na operação aritmética de multiplicação em *hardware* (no caso, variável por variável), um melhor desempenho em relação ao *delay* (menor) da mesma, quando são empregados módulos no formato 2^n e $2^n \pm 1$. Já para os formatos $2^n \pm k$, a operação indica uma forte tendência para apresentar sempre *delays* maiores.

Figura 2.1 – Arquiterura RNS para os módulos $\{m_1\ ,\ m_2\ ,\ m_3\}$



Fonte: Adaptado de (PARHAMI, 2010).

Figura 2.2 – Multiplicador variável-variável para os módulos 2^n, $2^n \pm 1$ e $2^n \pm k$.



Fonte: Adaptado de (PREDIGER et al., 2023)

2.3 Conversão binário-RNS

Uma forma mais simples e intuitiva para obter a conversão do sistema binário para o formato RNS é a realização de sucessivas operações de divisão até se atingir o resto (resíduo) inteiro. No entanto, operações de divisão, em *hardware*, representam um tempo relativamente grande no processamento (MOHAN, 2002). Para uso em aritmética de *hardware* duas são as técnicas principais usadas na conversão de binário para RNS, as baseadas em ROMs (*Read-Only Memories*) e as que usam uma combinação de um determinado conjunto de módulos (MOHAN; MOHAN, 2016). Dentre as que usam combinações modulares estão aquelas com o modelo de *hardware* para operações que elevam na potência n a base 2, no formato $2^n + 1$ *e* $2^n - 1$ como em (BI; JONES, 1988) e no formato $2^n + k$ *e* $2^n - k$ como em (MATUTINO et al., 2011). A equação 2.3 demonstra este processo genérico de conversão.

$$\left|(x_{k-1}...x_1x_0)_2\right|_{m_i} = \left||2^{k-1}x_{k-1}|_{m_i} + ... + |2x_1|_{m_i} + |x_0|_{m_i}\right|_{m_i}, \tag{2.3}$$

onde $(x_{k-1}...x_1x_0)$ é a representação do número no formato binário e m_is os respectivos módulos empregados.

Por exemplo, o número $X = (1010\ 0100)_2 = 164_{10}$ (no seu formato binário e decimal respectivamente), ao ser convertido para o sistema RNS com o conjunto de módulos 3, 5 e 7, produz os resíduos obtidos das equações 2.4, 2.5 e 2.6 respectivamente.

$$X_0 = |X|_3 = \left||2^7|_3 + |2^5|_3 + |2^2|_3\right|_3 = |2 + 2 + 1|_3 = 2, \tag{2.4}$$

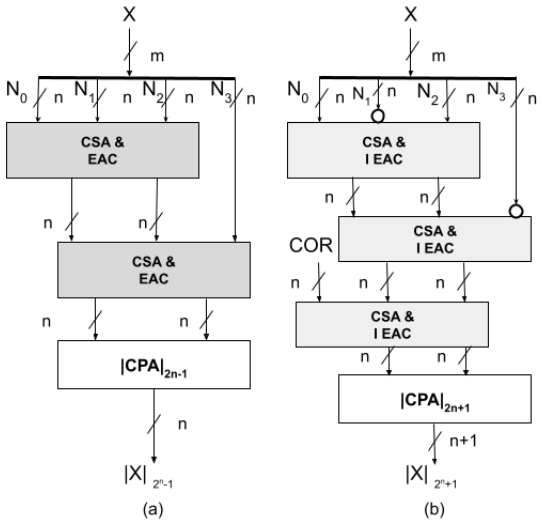
$$X_1 = |X|_5 = \left||2^7|_5 + |2^5|_5 + |2^2|_5\right|_5 = |3 + 2 + 4|_5 = 4, \tag{2.5}$$

$$X_2 = |X|_7 = \left||2^7|_7 + |2^5|_7 + |2^2|_7\right|_7 = |2 + 4 + 4|_7 = 3. \tag{2.6}$$

Para os exemplos de redução modular demonstrados, nessa parte, da-se ênfase na conversão por um vetor de $4n$ *bits*, destacando-se que os estudos se concentrarão no processador RNS (apresentado no capítulo 3) com o conjunto de módulos no formato $\{2^n\ ,\ 2^n - 1\ ,\ 2^n + 1\}$. Na figura 2.3, a entrada X possui um comprimento de $4n$ bits e é segmentada em 4 partes N_i ocupando n *bits* cada. Os n blocos são agrupados passando por algumas etapas como somadores CSA (*Carry Save Adder*) em conjunto com EAC (*End-Around Carry*) definidos em (PIESTRAK, 1995). A etapa final é composta por um CPA (*Carry Propagate Adder*) que produz a saída $|X|_{m_i}$ com n *bits* para o módulo $2^n - 1$ visto na figura 2.3 (a) e $n + 1$ bits para $2^n + 1$, na figura 2.3 (b) (WESTE; HARRIS, 2011). Complementando, os somadores anteriormente mencionados são compostos pelos elementos da soma básica, os FAs (*Full Adders*).

Os elementos N_i (segmentos mencionados anteriormente), da figura 2.3, são formados a partir do tamanho em *bits* definido para X. Cada qual ocupa a fração proporcional

Figura 2.3 – Circuito básico da conversão direta binário-RNS para os módulos $\{2^n \pm 1\}$



Fonte: Adaptado de (MOHAN; MOHAN, 2016).

ao máximo permitido pelos respectivos módulos. Como pode ser visto na representação binária de X e expressa pela equação 2.7.

$$X = \sum_{i=1}^{4n-1} 2^i x_i = 2^{3n}N_3 + 2^{2n}N_2 + 2^n N_1 + N_0 \tag{2.7}$$

2^n: Para as conversões baseadas em canais do tipo 2^n (no caso de $m = 16$ quando $n = 4$) o valor de $|X|_{2^n}$ pode ser obtido do resto da divisão de X por 2^n e de um truncamento no valor de $|X|$, como pode ser demonstrado nas equações 2.8 e 2.9.

$$|X|_{2^n} = |2^{3n}|_{2^n}.N_3 + |2^{2n}|_{2^n}.N_2 + 2^n.N_1 + N_0 \tag{2.8}$$

Como $|2^{3n}|_{2^n}$, $|2^{2n}|_{2^n}$ e $|2^n|_{2^n}$ são iguais a 0 então:

$$|X|_{2^n} = \{x_{(n-1)}, ..., x_1, x_0\} \tag{2.9}$$

$2^n - 1$: A equação 2.10 demonstra a conversão de X para o canal cujo módulo está no formato $2^n - 1$ e o circuito básico é demonstrado na figura 2.3 (a). Como $|2^n|_{2^n-1} = 1$ tem-se que:

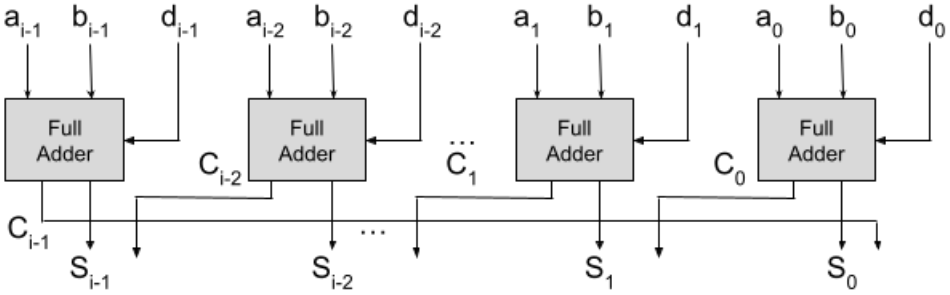
$$|X|_{2^n-1} = \left|N_3 + N_2 + N_1 + N_0\right|_{2^n-1} \tag{2.10}$$

$2^n + 1$: Já para o canal cujo módulo está no formato $2^n + 1$, na figura 2.3 (b), pode-se verificar sua equação geral em 2.11. Como $|2^n|_{2^n+1} = -1$ tem-se que:

$$\begin{aligned} |X|_{2^n+1} &= \left|N_3 - N_2 + N_1 - N_0\right|_{2^n+1} \\ |X|_{2^n+1} &= \left| - N_3 + |N_2 - N_1 + N_0|_{2^n+1}\right|_{2^n+1} \end{aligned} \tag{2.11}$$

A estrutura interna do somador CSA & EAC pode ser vista na figura 2.4 e indica a soma de 3 operadores A, B e D. Por conta deste lay-out (3 entradas e 2 saídas) este tipo de somador também é chamado de compressor 3:2.

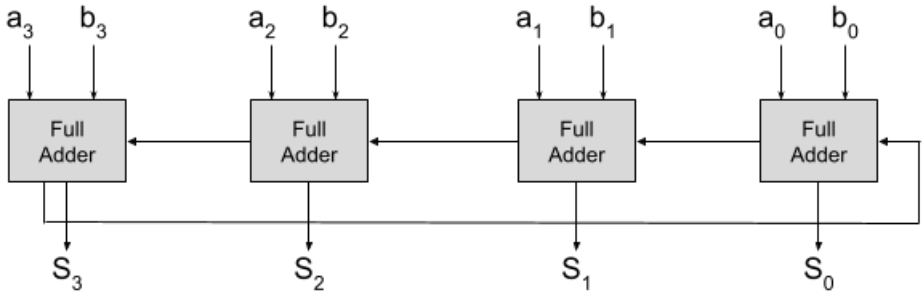
Figura 2.4 – Diagrama em blocos do funcionamento para o somador baseado em CSA & EAC.



Fonte: Adaptado de (PARHAMI, 2010).

A estrutura interna do somador, para módulo 15, pode ser vista na figura 2.5.

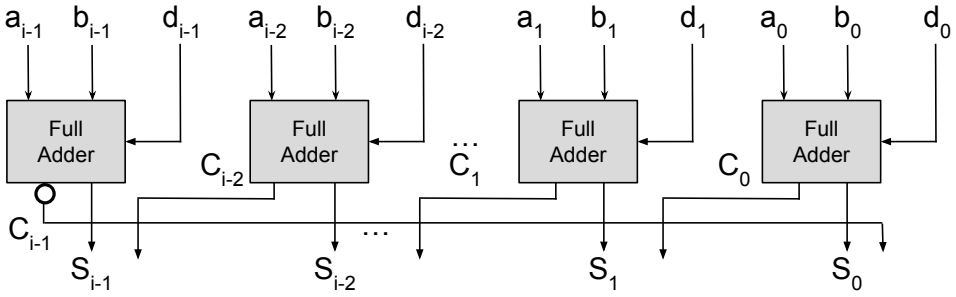
Figura 2.5 – Diagrama em blocos do funcionamento para o somador baseado em CPA para a operação $|A + B|_{15}$.



Fonte: Adaptado de (MOHAN; MOHAN, 2016).

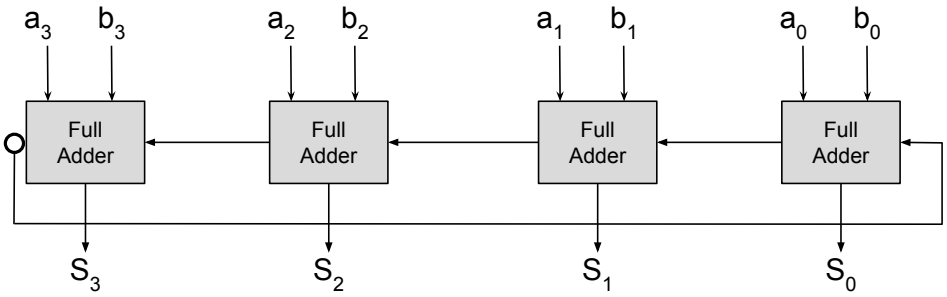
Já para as figuras 2.6 e 2.7 tem-se as operações utilizando módulos nos formato $2^n + 1$. Para o somador da figura 2.6 pode-se verificar que o último *carry out* é negado e passa a ser o dígito menos significativo do resultado da soma, registrado em S_i. Aplicando a um módulo específico, quando $n = 4$ (visto na figura 2.7) sua implementação se dá com CPAs e uma realimentação do último *carry out* para o primeiro *carry in*.

Figura 2.6 – Diagrama em blocos do funcionamento para o somador baseado em CSA & IEAC (*Inverted End-Around Carry*).



Fonte: Adaptado de (PARHAMI, 2010).

Figura 2.7 – Diagrama em blocos do funcionamento para o somador baseado em CPA para a operação $|A + B|_{17}$.



Fonte: Adaptado de (MOHAN; MOHAN, 2016).

As arquiteturas demonstradas nas figuras 2.5 e 2.7 (CPAs) apresentam a característica construtiva que leva a uma instabilidade na apresentação do resultado final. Devido ao enlace feito entre o *carry out* da última célula somadora e o *carry in* da primeira célula que leva a um tempo maior (devido a propagação dos *carries*) de acomodação do resultado da soma final. Em virtude da instabilidade, aplica-se uma nova abordagem que utiliza somadores em paralelo e um multiplexador e que se encontra demonstrada na seção 2.4.

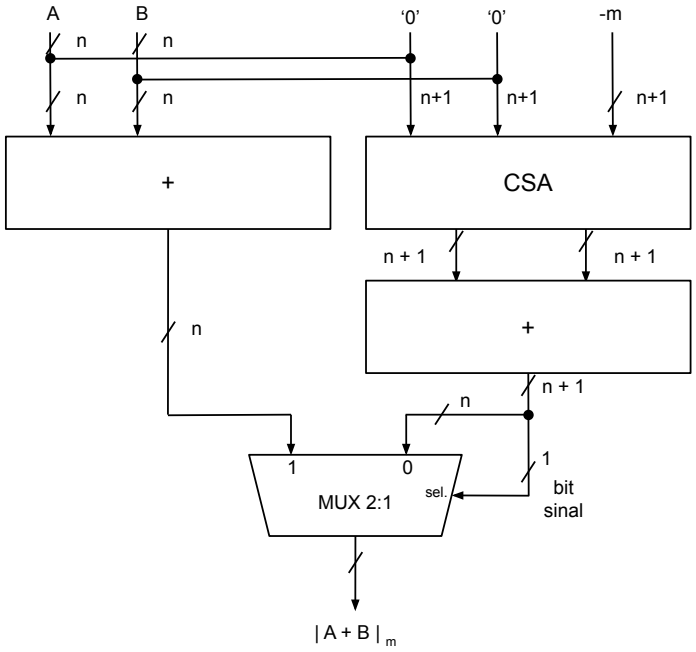
2.4 Aritmética RNS

As operações aritméticas, realizadas utilizando a representação RNS, não são afetadas pelo efeito da propagação do *carry* reduzindo assim o *delay*. *Delay* este que depende principalmente da escolha do conjunto de módulos mais adequado à faixa em que se pretende trabalhar (PARHAMI, 2010). As operações que se beneficiam mais diretamente

da representação RNS são adições/subtrações e multiplicações. Uma das estratégias para se obter um melhor desempenho é escolher os módulos de menor tamanho possível aliada com a maior abrangência da DR.

Adição: Na figura 2.8 tem-se um somador genérico de duas entradas A e B. O primeiro bloco efetua uma soma do tipo CPA entre os operadores e em paralelo uma soma no formato CSA com a entrada m. Posteriormente, os valores entram num multiplexador $2:1$ cuja seleção é feita por um *bit* usado como sinal da operação de soma. A escolha deste formato de somador é motivada pela instabilidade apresentada nas operações com somadores no formato CPA & EAC e CPA & IEAC (demonstrados anteriormente), mesmo estes apresentando um melhor desempenho.

Figura 2.8 – Soma modular genérica para operação $|A + B|_m$.

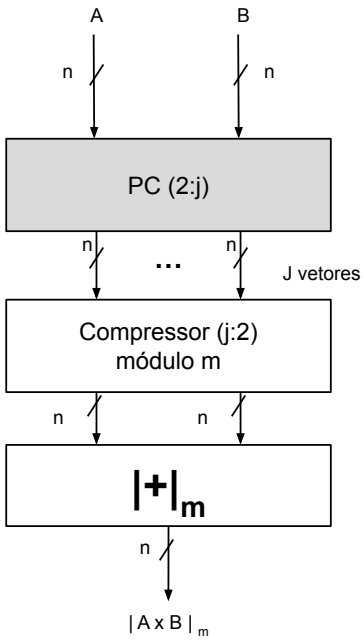


Fonte: Elaborado pelo autor.

Multiplicação: Na figura 2.9 tem-se um multiplicador genérico de duas entradas A e B. Na fase da computação inicial (pré-computação ou PC) o desempenho se relaciona diretamente com a escolha do conjunto de módulos que serão adotados. Já a etapa de compressão costuma ser implementada de forma mais genérica e se ocupa em efetuar a compressão binária utilizando uma soma de vetores. Este conjunto de blocos usa como referência estudos de (DADDA, 1965), (WALLACE, 1964) e (PATRONIK; PIESTRAK, 2017) referente às árvores de compressão basicamente formadas por HAs (*Half Adders*) e FAs.

O funcionamento interno das operações de multiplicação para os módulos 2^n e $2^n \pm 1$ serão descritas a seguir. Admitindo que $n = 4$, nas figuras 2.10 à 2.15 estão apresentados os diagramas em blocos das operações de multiplicação dos módulos mencionados.

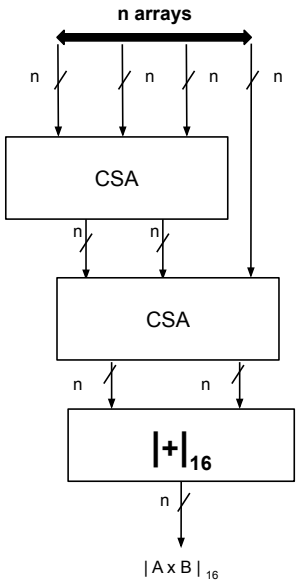
Figura 2.9 – Multiplicação modular genérica para operação $|AxB|_m$.



Fonte: Elaborado pelo autor.

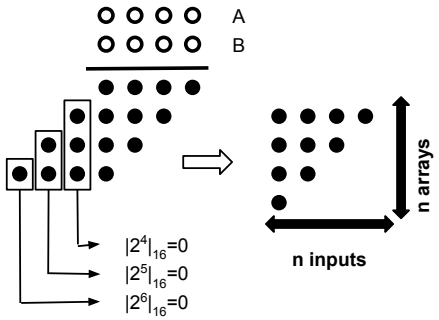
2^n: Nas figuras 2.10 e 2.11 pode-se verificar a representação da multiplicação para módulo 16. Pode-se verificar que no diagrama de pontos, figura 2.11, que as somas parciais são truncadas, pois, $|2^4|_{16}$, $|2^5|_{16}$ e $|2^6|_{16}$ equivalem a 0. O que torna o diagrama da figura 2.10 muito mais simplificado e cujo embasamento está no uso dos compressores (CSAs) e uma soma final que é obtida aplicando-se um único CPA (bloco final).

Figura 2.10 – Multiplicação modular para $m = 2^n$, para $n = 4$.



Fonte: Elaborado pelo autor.

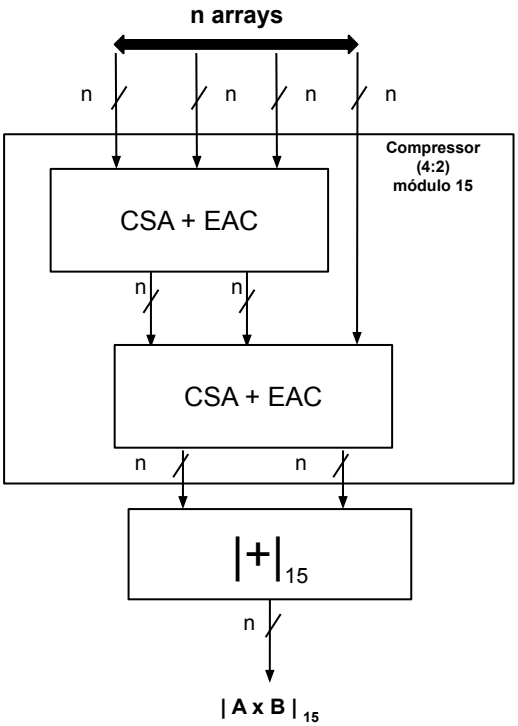
Figura 2.11 – Multiplicação modular para $m = 2^n$, para $n = 4$. No formato do diagrama de pontos.



Fonte: Elaborado pelo autor.

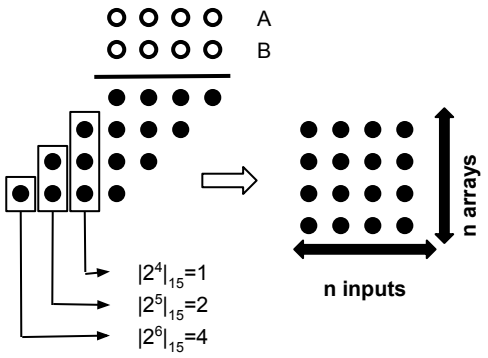
$2^n - 1$: Admitindo que $n = 4$, na figura 2.12 pode-se verificar o diagrama em blocos das operações para uma multiplicação no formato $2^n - 1$ e na figura 2.13 a multiplicação módulo 15 no formato do diagrama de pontos. Nessa última pode-se notar a necessidade de agrupamento dos conjuntos de *bits* de mesma significância (peso) e que se deslocam para comporem as somas parciais, formando uma matriz completa com todos os dígitos. Esta operação de soma e deslocamento é executada pelos blocos CSA e EAC que podem ser vistos na figura 2.12. O resultado é obtido com uma soma módulo $2^n - 1$, representada pelo somador genérico da figura 2.8.

Figura 2.12 – Multiplicação modular para $m = 2^n - 1$, para $n = 4$.



Fonte: Elaborado pelo autor.

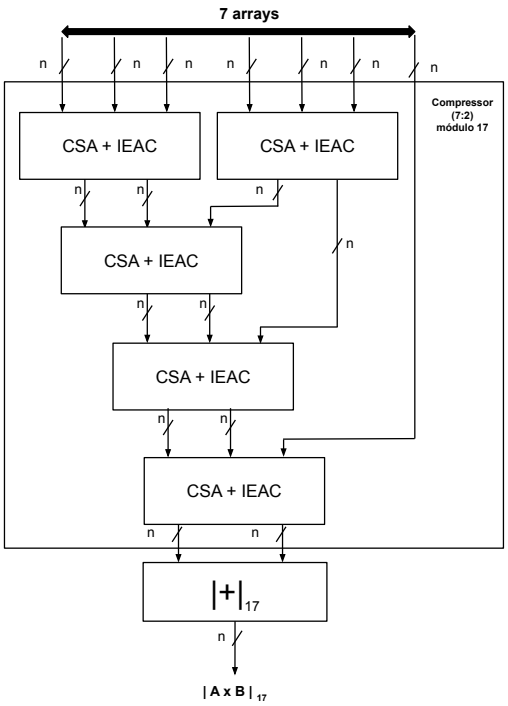
Figura 2.13 – Multiplicação modular para $m = 2^n - 1$, para $n = 4$. No formato do diagrama de pontos.



Fonte: Elaborado pelo autor.

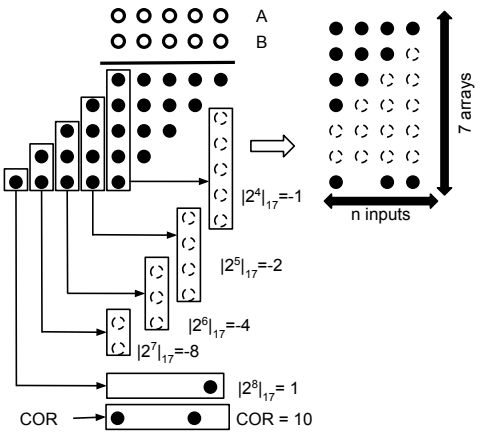
$2^n + 1$: Admitindo que $n = 4$, na figura 2.14 pode-se verificar o diagrama em blocos das operações para uma multiplicação no formato $2^n + 1$ e na figura 2.15 a multiplicação módulo 17 no formato do diagrama de pontos. Nessa última pode-se notar que os grupos excedentes (do truncamento) possuem pesos que representam valores negados (como, por exemplo, $|2^4|_{17} = -1$) e podem ser agrupados formando a matriz final de somas. Por esta razão no diagrama da figura 2.14 pode-se verificar as negações nos blocos formados por CSAs e IEACs. A soma final, em módulo $2^n + 1$, é realizada no último bloco composto pelo somador genérico indicado na figura 2.8.

Figura 2.14 – Multiplicação modular para $m = 2^n + 1$, para $n = 4$.



Fonte: Elaborado pelo autor.

Figura 2.15 – Multiplicação modular para $m = 2^n + 1$, para $n = 4$. No formato do diagrama de pontos.



Fonte: Elaborado pelo autor.

2.5 Conversão RNS-binário

Esta etapa consiste na conversão (ou reversão) da representação RNS para o formato binário que, no *delay* e potência, representa um gargalo no processo. O desempenho deste estágio depende da DR, do tamanho da entrada e da combinação de módulos utilizados. Quanto mais facilitada for esta conversão reversa, melhor será o desempenho do sistema. Basicamente existem duas abordagens principais adotadas para este processo: o CRT (*Chinese Remainder Theorem*), com suas variações, e a MRC (*Mixed Radix Conversion*) (MOHAN; MOHAN, 2016).

2.5.1 Detalhando o CRT e CRT I

O CRT e suas variações representam uma das técnicas de conversão RNS-binário e a sua equação, para o cálculo genérico, encontra-se demonstrada em 2.12.

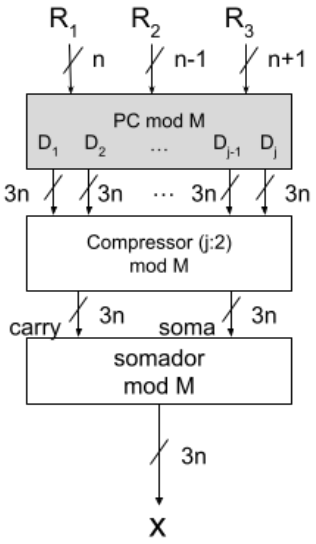
$$X = \left| \sum_{i=1}^{\mathrm{n}} \hat{m}_i |\hat{m}_i{}^{-1}|_{m_i} \; . \; R_i \right|_{DR}, \tag{2.12}$$

onde $DR = \prod_{i=1}^{n} m_i$, $\hat{m}_i = DR/m_i$ e $\left| |\hat{m}_i{}^{-1}|_{m_i} \hat{m}_i \right|_{m_i} = 1$. Os valores de R_i representam os respectivos resíduos de cada canal m_i utilizado, proveniente da aritmética modular e $|\hat{m}_i{}^{-1}|_{m_i}$ são as multiplicativas inversas que são obtidas de acordo com cada conjunto de módulos empregados.

Na figura 2.16 é apresentada uma das implementações da técnica CRT, aplicada para três módulos $\{m_1\ , m_2\ , m_3\}$, no formato $\{2^n\ ,\ 2^n \pm 1\}$, onde PC é uma etapa de pré-computação.

Por exemplo, se $X = 33$, $n = 4$ e os módulos $m_1,\ m_2\ e\ m_3 = \{17,\ 16,\ 15\}$ então os resíduos $R_1,\ R_2\ e\ R_3 = \{16,\ 1,\ 3\}$ são obtidos na conversão binário-RNS.

Figura 2.16 – Diagrama do conversor RNS-binário no formato CRT. Para módulos m_1, m_2 e m_3.



Fonte: Adaptado de (OMONDI; PREMKUMAR, 2007).

Na conversão inversa tem-se:

$$DR = 17 \times \ 16 \times \ 15 = 4080 \tag{2.13}$$

$$\hat{m_1} = \frac{4080}{17} = 240; \hat{m_2} = \frac{4080}{16} = 255; \hat{m_3} = \frac{4080}{15} = 272 \tag{2.14}$$

$$|\hat{m_1}^{-1}|_{17} = 9 \ ; |\hat{m_2}^{-1}|_{16} = 15 \ ; |\hat{m_3}^{-1}|_{15} = 8 \tag{2.15}$$

$$\begin{aligned} X &= \left| |240.16|_{17}.9 + |255.1|_{16}.15 + |272.8|_{15}.3 \right|_{4080} \\ X &= \left| 1920 \ + \ 3825 \ + \ 2448 \right|_{4080} = \left| 8193 \right|_{4080} = 33 \end{aligned} \tag{2.16}$$

Uma nova variação do CRT é o CRT-I (ou novo CRT-I) e que apresenta como premissas as mesmas condições iniciais do CRT, que são: um conjunto de módulos $\{m_1, m_2, ..., m_n\}$ e a representação binária de X pelos seus respectivos resíduos, ou seja, $X = \{R_1, R_2, ..., R_n\}$ (WANG, 2000).

A equação genérica para o cálculo do CRT-I encontra-se demonstrada em 2.17.

$$X = \left| \sum_{i=1}^{\mathrm{n}} |V_i R_i|_{\hat{m_i}} \right|_{\hat{m_i}} m_1 + R_1, \tag{2.17}$$

onde: $V_1 = \frac{(|\hat{m_1}^{-1}|_{m1}.\hat{m_1})-1}{m1}$ e $V_i = \left|\hat{m_i}^{-1}\right|_{m_i}.\frac{\hat{m_i}}{m_1}$ *para* $2 \leq i \leq n$.

Na figura 2.17 é apresentada uma das implementações da técnica CRT I para os módulos $\{m_1 \ , \ m_2 \ , \ m_3\}$ no formato $\{2^n \ , \ 2^n \pm 1\}$ e onde PC é uma etapa de

pré-computação. A equação desta implementação é definida em 2.18.

$$X = \left| V_1.R_1 + V_2.R_2 + V_3.R_3 \right|_{\hat{m}_i} .m_1 + R_1, \tag{2.18}$$

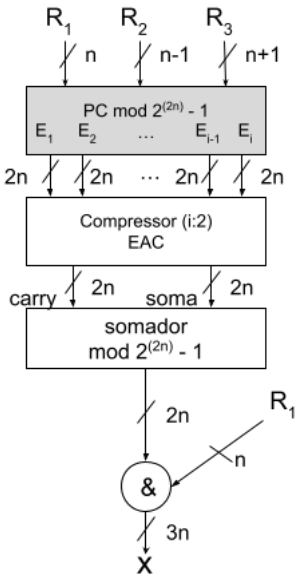
onde $V_1 = \frac{(|\hat{m_1}^{-1}|_{m1}.\hat{m_1})-1}{m1}$, $V_2 = \left|\hat{m_2}^{-1}\right|_{m_2}.\frac{\hat{m_2}}{m_1}$ e $V_3 = \left|\hat{m_3}^{-1}\right|_{m_3}.\frac{\hat{m_3}}{m_1}$.

Para o exemplo numérico de $X = 33$, $n = 4$ e os módulos $m_1,\ m_2\ e\ m_3 = \{16,\ 15,\ 17\}$ então os resíduos $R_1,\ R_2\ e\ R_3 = \{1,\ 3,\ 16\}$ provenientes da conversão binário-RNS em CRT-I e cujo desenvolvimento é demonstrado em 2.19.

$$\begin{aligned} V_1 = \frac{(15.255)-1}{16} = 239\ ;\ V_2 = 8.\frac{272}{16} = 136\ ; V_3 = 9.\frac{240}{16} = 135 \\ X = \left|239.1 + 136.3 + 135.16\right|_{255}.16 + 1 = \left|239 + 408 + 2160\right|_{255}.16 + 1 \\ X = \left|239 + 153 + 120\right|_{255}.16 + 1 = \left|512\right|_{255}.16 + 1 = \left|2\right|_{255}.16 + 1 = 2.16 + 1 = 33 \end{aligned} \tag{2.19}$$

Como $m_1 = 16$ (múltiplo de 2), a multiplicação final torna-se mais facilitada, revelando uma vantagem do CRT I sobre o CRT. A outra facilidade é a concatenação final com R_1, vista na figura 2.17.

Figura 2.17 – Diagrama em blocos do conversor RNS-binário usando CRT I. Para os módulos $\{2^n\ ,\ 2^n \pm 1\}$.



Fonte: Elaborado pelo autor.

2.5.2 Detalhando o MRC

O MRC é uma técnica de conversão RNS-binário realizada de forma sequencial e que implica na realização de subtrações e multiplicações modulares, sua equação geral pode ser vista em 2.20 (OMONDI; PREMKUMAR, 2007).

$$X = z_1 + z_2 m_1 + z_3 m_2 m_1 + ... + z_n m_{n-1} ... m_2 m_1, \tag{2.20}$$

onde z_i deve obedecer à regra $0 \leq z_i \leq (m_{i+1} - 1)$, formando os dígitos Mixed Radix (OMONDI; PREMKUMAR, 2007). Na representação do número X, leva-se em consideração as bases (ou "radices"), indicadas por $\{m_n\ , m_{n-1}\ ...\ m_1\}$ e que são os próprios módulos. Na figura 2.18 tem-se o diagrama em blocos do conversor MRC para os módulos $\{m_1\ , m_2\ , m_3\}$ no formato $\{2^n\ , 2^n \pm 1\}$, onde PC é o processo de pré-computação, R_i são os respectivos resíduos e a sua equação, já resumida, pode ser vista em 2.21.

$$X = z_1 + z_2 m_1 + z_3 m_2 m_1, \tag{2.21}$$

onde z_1, z_2 e z_3 são determinados pelas reduções modulares relacionadas aos respectivos módulos m_1, m_2 e m_3, como pode ser visto na equação 2.22.

$$\begin{aligned} |X|_{m_1} &= z_1 \\ |X|_{m_1} &= R_1 \end{aligned} \tag{2.22}$$

Colocando z_1 para o lado esquerdo da equação 2.21, aplicando a operação $||_{m2}$ em ambos os lados assim como $|m_1^{-1}|_{m_2}$, chega-se a equação 2.23 para obtenção de z_2.

$$z_2 = \left| |m_1^{-1}|_{m_2} (R_2 - z_1) \right|_{m_2} \tag{2.23}$$

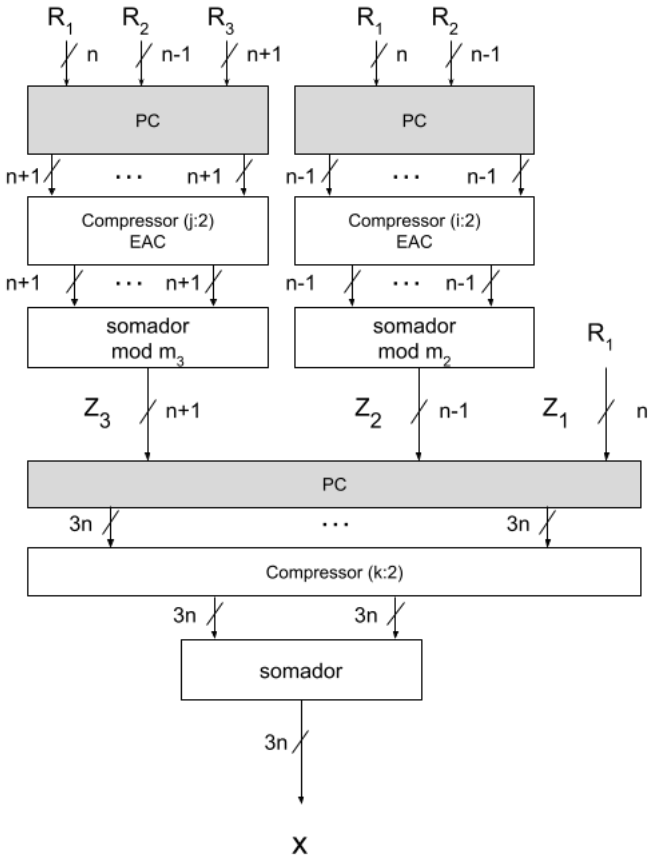
Aplicando o mesmo processo para obter z_3 chega-se na equação 2.24.

$$z_3 = \left| \left| (m_2 . m_1)^{-1} \right|_{m_3} . (R_3 - (z_2 . m_1 + z_1)) \right|_{m_3} \tag{2.24}$$

Como exemplo numérico, se $X = 33$, $n = 4$ e os módulos m_1, m_2 e $m_3 = \{17,\ 16,\ 15\}$, então os resíduos R_1, R_2 e $R_3 = \{16,\ 1,\ 3\}$. Numa conversão binário-RNS em MRC tem-se a sequência do cálculo indicada em 2.25.

$$z_1 = R_1 = 16\ ;\ z_2 = \left| 1.(1 - 16) \right|_{16} = 1\ ;\ z_3 = 0\ e\ X = 16 + 1.17 + 0.16.17 = 33 \tag{2.25}$$

Figura 2.18 – Diagrama em blocos do conversor MRC para os módulos m_1, m_2 e m_3.



Fonte: Adaptado de (OMONDI; PREMKUMAR, 2007).

3 Propostas em *hardware* com uso de RNS e criptografia RSA

Neste capítulo abordam-se as propostas, aplicadas em *hardware*, utilizando a representação RNS e módulos de baixo custo (mencionados no capítulo 2). Nele encontram-se detalhadas a motivação, aplicação dos módulos de baixo custo na aritmética RNS, definição e usos dos módulos auxiliares, estudos de melhorias nas operações de multiplicação por constantes e aplicações em criptografia baseada em RSA, utilizando operações de exponenciação modular.

3.1 Motivação

Os estudos de aplicações apresentados neste capítulo são motivados pelos bons resultados de desempenho das operações modulares com o emprego dos módulos no formato $\{ 2^n , 2^n-1 , 2^n+1\}$. Estes módulos são aqui denominados de baixo custo e os resultados motivadores foram descritos na parte inicial do capítulo da fundamentação teórica (capítulo 2) acompanhada da proposta de um processador baseado em RNS. Outro ponto motivador importante da-se na possibilidade de uso nas atividades de criptografia digital cuja base fundamental está apoiada nos cálculos utilizando a exponenciação modular.

3.2 Aplicação de módulo auxiliar de baixo custo na aritmética RNS

3.2.1 Módulo auxiliar

O conceito de módulo auxiliar é derivado da propriedade cíclica do formato RNS e pode ser representada por equações que obedecem ao formato diofantino (CLAUDIO; ORLANDI; PIAZZA, 1990). Dentre as propriedades das operações modulares existe a que relaciona o módulo original com o seu auxiliar e que está representada na equação 3.1. Esta relação é um dos pontos fundamentais que baseiam os estudos e ensaios que se darão no decorrer deste trabalho.

$$\Big||X|_{m_{AUX}}\Big|_{m_{ORI}} = |X|_{m_{ORI}}, \tag{3.1}$$

onde m_{AUX} = módulo auxiliar, m_{ORI} = módulo original e se relacionam conforme a equação 3.2.

$$m_{AUX} = \gamma.m_{ORI}, \tag{3.2}$$

onde γ é uma constante inteira ímpar maior que 1. Então, por exemplo, partindo de um módulo auxiliar (m_{AUX}) baseado em $2^n - 1$ (módulo de melhor eficiência) e $n = 38$ tem-se, na sua decomposição, os divisores indicados na equação 3.3, (INC., 2024).

$$\begin{gathered}\overbrace{274877906943}^{m_{AUX}} = \overbrace{3}^{\gamma} \cdot \overbrace{91625968981}^{m_{ORI}} \\ m_{AUX} = 2^{38} - 1 = 3.(2^{36} + k)\end{gathered} \tag{3.3}$$

Escolhe-se, então, o menor divisor que seja maior que 1, no caso 3, para ser a constante γ, de forma que a distância numérica entre o original e o módulo auxiliar seja a menor possível, pois distâncias numéricas muito maiores podem não produzir um efeito vantajoso nos tempos de processamento. Obtém-se então a relação entre o módulo original e o auxiliar como $m_{AUX} = 3.m_{ORI}$, onde o módulo original é um valor com baixa eficiência, já que o valor de $k \neq 1$, como demonstrado na figura 2.2.

Partindo desta premissa, foram desenvolvidos alguns estudos com uma série de módulos originais e seus respectivos auxiliares com o objetivo de demonstrar a eficiência destes últimos em relação aos primeiros. Os módulos auxiliares utilizados baseiam-se nos conjuntos $2^n - 1$, $2^n + 1$ e suas derivações de forma a obter os benefícios de menor *delay*.

A escolha de módulos das famílias $2^n - 1$ e $2^n + 1$ se dá por conta dos ensaios realizados no conjunto de DRs com $n = 5 \sim 10$, onde $X.n$ é o número de canais aplicados (PREDIGER et al., 2023). Por exemplo, M_{6n} é o conjunto modular de 6 canais que produz uma $DR = 6n$ *bits*.

O estudo mencionado anteriormente consistiu na análise de extensões do conjunto de módulos $\{2^n,\ 2^n \pm 1\}$, usando formas mais balanceadas das variações $\{2^n \pm 1\}$ e módulos no formato $\{2^n \pm k\}$, que se relacionam de modo que sejam co-primos entre si, ou seja, que não possuem uma relação direta de divisibilidade entre si.

Como pode-se observar, alguns valores indicados na tabela 3.1, como 11, por exemplo (quando $n = 5$), representa o módulo original ($m_{ORI} = 11$), levando a um $m_{AUX} = 3.11 = 33 = 2^5 + 1$ que é mais eficiente que 11, no *delay*. Somando a isso, $m_{AUX} = 33$ oferece um conjunto não coprimo e que será corrigido na operação de conversão reversa.

A tabela 3.1 mostra o resultado do conjunto de módulos proposto para um $DR = 5n,\ 3 \leq\ n \leq\ 13$, para $M_{5n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+1} \pm 1\}$ par $M_{6n} = \{M_{5n},\ 2^{n+2} - 1\}$ e impar $M_{6n} = \{M_{5n},\ 2^{n+2} + 1\}$.

A tabela 3.2 mostra o resultado do conjunto de módulos proposto para um $DR = 7n,\ 4 \leq\ n \leq\ 13$, para $M_{7n} = \{2^n,\ 2^n \pm 1,\ 2^{n+1} \pm 1\ 2^{n+2} \pm 1\}$ par $M_{8n} = \{M_{7n},\ 2^{n+3} - 1\}$ e impar $M_{8n} = \{M_{7n},\ 2^{n+2} + 1\}$.

Como pode-se observar, alguns valores indicados na tabela 3.2, como 43, por exemplo (quando $n = 7$), representa o módulo original (m_{ORI}), levando a um $m_{AUX} =$

Tabela 3.1 – Valores dos módulos para conversão reversa em $M_{5n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+1} \pm 1\}$, $M_{6n} = \{M_{5n},\ 2^{n+2} - 1\}$ e $M_{6n} = \{M_{5n},\ 2^{n+2} + 1\}$.

n	2^n	$2^n - 1$	$2^n + 1$	$2^{n+1} - 1$	$2^{n+1} + 1$	$2^{n+2} - 1$	$2^{n+2} + 1$
3	2^3	$2^3 - 1$	$2^3 + 1$	5	$2^4 + 1$	$2^5 - 1$	–
4	2^4	$2^4 - 1$	$2^4 + 1$	$2^5 - 1$	11	–	13
5	2^5	$2^5 - 1$	11	$2^6 - 1$	$2^6 + 1$	$2^7 - 1$	–
6	2^6	$2^6 - 1$	$2^6 + 1$	$2^7 - 1$	43	–	$2^8 + 1$
7	2^7	$2^7 - 1$	$2^7 + 1$	85	$2^8 + 1$	$2^9 - 1$	–
8	2^8	85	$2^8 + 1$	$2^9 - 1$	$2^9 + 1$	341	–
9	2^9	$2^9 - 1$	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	–
10	2^{10}	341	$2^{10} + 1$	$2^{11} - 1$	$2^{11} + 1$	–	$2^{12} + 1$
11	2^{11}	$2^{11} - 1$	683	$2^{12} - 1$	$2^{12} + 1$	$2^{13} - 1$	–
12	2^{12}	$2^{12} - 1$	$2^{12} + 1$	$2^{13} - 1$	2731	–	3277
13	2^{13}	$2^{13} - 1$	2731	$2^{14} - 1$	$2^{14} + 1$	$2^{15} - 1$	–

Fonte: Adaptado de (PREDIGER et al., 2023).

$3.43 = 129 = 2^7 + 1$ mais eficiente que $m_{ORI} = 43$.

Tabela 3.2 – Valores dos módulos para conversão reversa em $M_{7n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+2} \pm 1\ 2^{n+2} \pm 1\}$, $M_{8n} = \{M_{7n},\ 2^{n+3} - 1\}$ e $M_{8n} = \{M_{7n},\ 2^{n+3} + 1\}$.

n	2^n	$2^n - 1$	$2^n + 1$	$2^{n+1} - 1$	$2^{n+1} + 1$	$2^{n+2} - 1$	$2^{n+2} + 1$	$2^{n+3} - 1$	$2^{n+3} + 1$
3	2^3	$2^3 - 1$	$2^3 + 1$	5	$2^4 + 1$	$2^5 - 1$	11	–	13
4	2^4	5	$2^4 + 1$	$2^5 - 1$	11	$2^6 - 1$	13	$2^7 - 1$	–
5	2^5	$2^5 - 1$	11	$2^6 - 1$	$2^6 + 1$	$2^7 - 1$	43	–	$2^8 + 1$
6	2^6	$2^6 - 1$	13	$2^7 - 1$	43	85	$2^8 + 1$	73	–
7	2^7	$2^7 - 1$	43	85	$2^8 + 1$	$2^9 - 1$	$2^9 + 1$	341	–
8	2^8	17	$2^8 + 1$	$2^9 - 1$	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	–
9	2^9	$2^9 - 1$	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	683	–	$2^{12} + 1$
10	2^{10}	341	$2^{10} + 1$	$2^{11} - 1$	683	819	$2^{12} + 1$	$2^{13} - 1$	–
11	2^{11}	$2^{11} - 1$	683	$2^{12} - 1$	$2^{12} + 1$	$2^{13} - 1$	2731	–	3277
12	2^{12}	819	$2^{12} + 1$	$2^{13} - 1$	2731	5461	$2^{14} + 1$	4861	–
13	2^{13}	$2^{13} - 1$	2731	5461	$2^{14} + 1$	$2^{15} - 1$	$2^{15} + 1$	–	$2^{16} + 1$

Fonte: Adaptado de (PREDIGER et al., 2023).

A tabela 3.3 mostra o resultado do conjunto de módulos proposto para um $DR = 9n,\ 4 \leq\ n \leq\ 13$, para $M_{9n} = \{2^n\ ,\ 2^n \pm 1,\ 2^{n+1} \pm 1,\ 2^{n+2} \pm 1,\ 2^{n+3} \pm 1\}$ par $M_{10n} = \{M_{9n},\ 2^{n+4} - 1\}$ e impar $M_{10n} = \{M_{9n},\ 2^{n+2} + 1\}$.

Pode-se observar que, alguns valores indicados na tabela 3.3, como 341, por exemplo (quando $n = 10$), representa o módulo original (m_{ORI}), levando a um $m_{AUX} = 3.341 = 1023 = 2^{10} - 1$ mais eficiente que $m_{ORI} = 341$.

As tabelas 3.1, 3.2 e 3.3 demonstram os resultados nas variações de n indicadas onde, a partir das observações, foi possível estabelecer uma escalabilidade do processo.

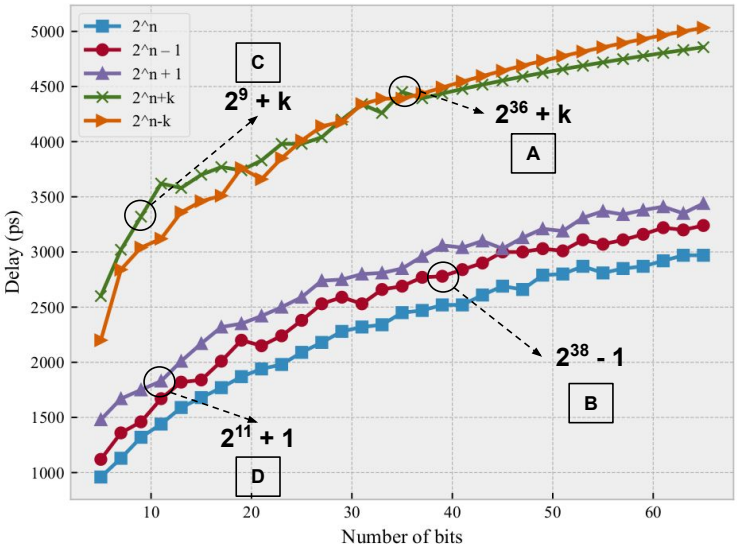
Tabela 3.3 – Valores dos módulos para conversão reversa em $M_{9n} = \{2^{n+k},\ 2^n \pm 1,\ 2^{n+2} \pm 1\ \ 2^{n+3} \pm 1\}$, $M_{10n} = \{M_{9n},\ 2^{n+4} - 1\}$ e $M_{10n} = \{M_{9n},\ 2^{n+4} + 1\}$.

n	2^n	$2^n - 1$	$2^n + 1$	$2^{n+1} - 1$	$2^{n+1} + 1$	$2^{n+2} - 1$	$2^{n+2} + 1$	$2^{n+3} - 1$	$2^{n+3} + 1$	$2^{n+4} - 1$	$2^{n+4} + 1$
4	2^4	5	$2^4 + 1$	$2^5 - 1$	11	$2^6 - 1$	13	$2^7 - 1$	43	–	$2^8 + 1$
5	2^5	$2^5 - 1$	11	$2^6 - 1$	$2^6 + 1$	$2^7 - 1$	43	17	$2^8 + 1$	73	–
6	2^6	7	$2^6 + 1$	$2^7 - 1$	43	17	$2^8 + 1$	73	$2^9 + 1$	341	–
7	2^7	$2^7 - 1$	43	17	$2^8 + 1$	$2^9 - 1$	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	–
8	2^8	17	$2^8 + 1$	$2^9 - 1$	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	683	–	241
9	2^9	73	$2^9 + 1$	341	$2^{10} + 1$	$2^{11} - 1$	683	91	$2^{12} + 1$	$2^{13} - 1$	–
10	2^{10}	341	$2^{10} + 1$	$2^{11} - 1$	683	91	$2^{12} + 1$	$2^{13} - 1$	$2^{13} + 1$	–	3277
11	2^{11}	$2^{11} - 1$	683	2819	$2^{12} + 1$	$2^{13} - 1$	2731	5461	$2^{14} + 1$	4681	–
12	2^{12}	91	$2^{12} + 1$	$2^{13} - 1$	2731	5461	$2^{14} + 1$	4681	$2^{15} + 1$	–	$2^{16} + 1$
13	2^{13}	$2^{13} - 1$	2731	5461	$2^{14} + 1$	$2^{15} - 1$	$2^{15} + 1$	4369	$2^{16} + 1$	$2^{17} - 1$	–

Fonte: Adaptado de (PREDIGER et al., 2023).

Na figura 3.1, retomando a motivação indicada no início deste capítulo, pode-se verificar o resultado do ensaio para indicar o desempenho já abordado, de forma inicial, na seção 2.2. Neste ensaio foram realizadas operações de multiplicação modular com os conjuntos 2^n, $2^n \pm 1$ e $2^n \pm k$. A implementação utilizou a tecnologia de $65nm$ Standard Cell ASIC UMC (INC., 2010) e é extrapolada para o número de *bits* um pouco acima de 60. Da figura pode-se deduzir que canais no formato $2^n \pm k$ não apresentam vantagens, pois resultam em *delays* maiores e que $\Delta(2^n - 1) \leq \Delta(2^n + 1)$. Fazendo com que canais baseados em $2^n - 1$, em *hardware*, sejam mais vantajosos, mesmo com valores de n maiores.

Figura 3.1 – *Delay* da multiplicação modular para 2^n, $2^n \pm 1$ e $2^n \pm k$.



Fonte: Adaptado de (PREDIGER et al., 2023).

Um exemplo, que pode ser extraído da tabela 3.3, é o módulo 683 (linha em que $n = 11$ e coluna $2^n + 1$), que está no formato $2^a + k$ e que apresenta um desempenho pior. Possuindo então, seu equivalente auxiliar no formato $2^n + 1$ com $n = 11$, como sendo $m_{AUX} = 2049$. Na figura 3.1 o exemplo está representado nos pontos C e D, como

os módulos original e auxiliar respectivamente. Observa-se também que há um ganho evidente em termos da redução no *delay*.

Outro exemplo, dado na equação 3.3 e que se encontra em destaque na figura 3.1, B indica um ponto na curva $2^n - 1$ representando o módulo $2^{38} - 1$ e que atua como auxiliar, já o ponto A indica o módulo original $2^{36} + k$. Observamos que há um ganho evidente na redução do *delay*.

Os resultados experimentais para validar o desempenho são provenientes de um estrutura descrita, dentro da família VHSIC (*Very High Speed Integrated Circuit*), em VHDL (VHSIC *Hardware Description Language*) e implementada com a tecnologia $65nm$, empregando bibliotecas Standard Cells ASIC da UMC e usando a ferramenta de síntese Design Vision. Os ensaios demonstram que, ao se utilizar de um relativo número de operações aritméticas de multiplicação, cujo *delay* é demonstrado na equação 3.4 (no caso de $n = 7$ e $DR = 10$), os tempos de resposta das operações se tornam comparativamente menores, justificando seu emprego.

Na equação 3.4, t é o número de operações MAC (*Multiply–accumulate operation*) escolhidas (ou desejadas), $1,77$ é o tempo da conversão direta (**direto**), $5,6$ o tempo para a conversão reversa (**reverso**) e os resultados dos ensaios podem ser vistos na figura 3.2.

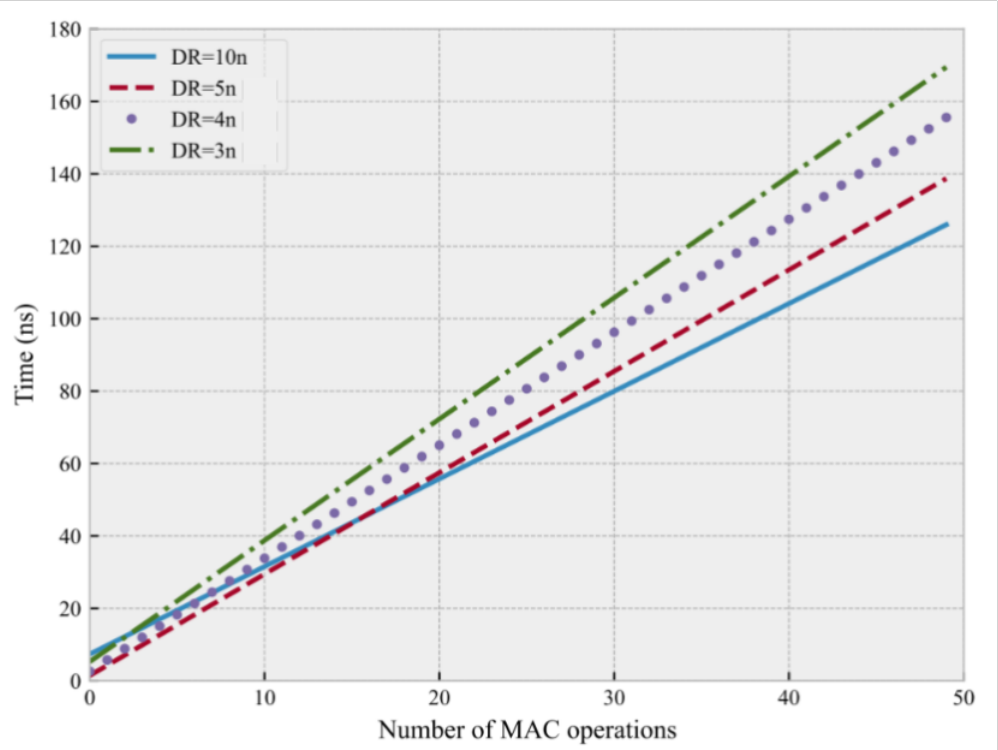
Outra abordagem da análise realizada, demonstrada na figura 3.3, foi a de considerar as operações como uma arquitetura combinacional, utilizando um ciclo de *clock* de $12,1ns$. Essa base tempo foi determinada, pois, representa 5 operações MAC por ciclo de *clock* (descontando tempos de *hold* e *setup* do registrador), o que pode ser observado ainda na equação 3.4 pelo valor constante de $2,42$, ou seja, $12,1 = 5\ .\ 2,42$.

$$delay\ total = \overbrace{1,77}^{direto} + t \times \overbrace{2,42}^{MAC} + \overbrace{5,6}^{reverso} \tag{3.4}$$

Na figura 3.2 pode-se verificar, por exemplo, que $10n$ torna-se preferível do que $5n$ após 16 operações do tipo MAC. Assim como o mesmo $10n$ torna-se preferível do que $4n$ após 7 operações, por exemplo.

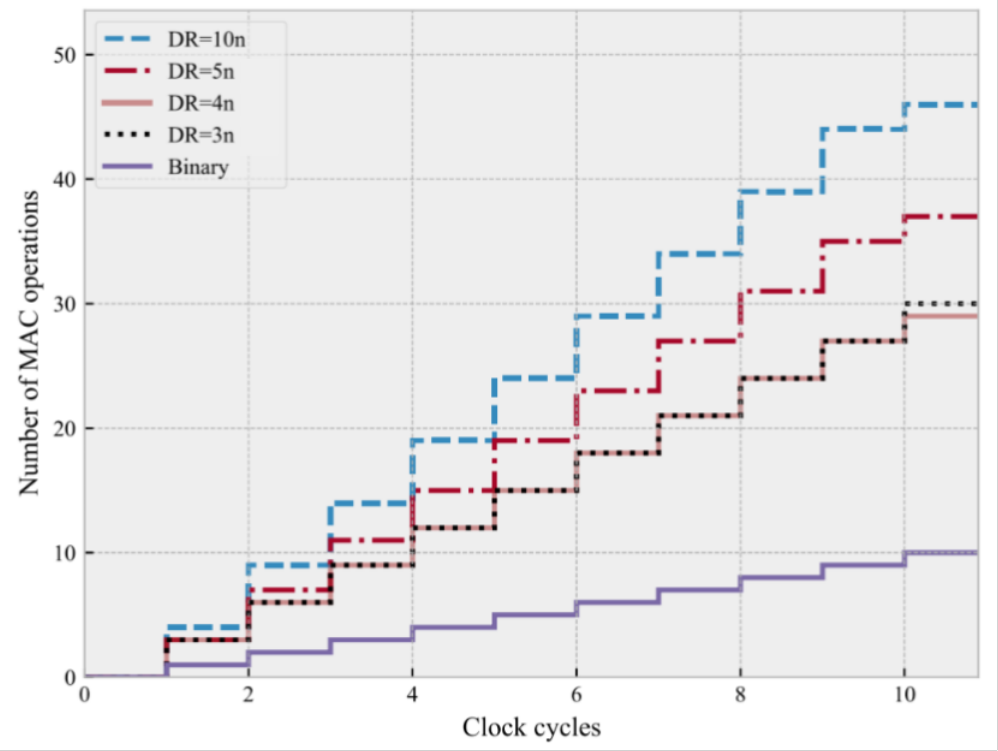
Na figura 3.3 pode-se verificar que a abordagem utilizada pode superar o desempenho de métodos da literatura tradicional (BHARDWAJ; SRIKANTHAN; CLARKE, 1999) e (CAO; CHANG; SRIKANTHAN, 2007) (para uma quantidade maior de operações). Isto pode ser observado na quantidade de operações MAC por ciclo de *clock*, executadas na aritmética binária tradicional (curva *Binary*), menores que as demais abordagens ($3n$, $4n$, $5n$ e $10n$), mesmo considerando as conversões diretas e reversas que são intrínsecas do sistema RNS.

Figura 3.2 – *Delay* por número de operações MAC.



Fonte: Adaptado de (PREDIGER et al., 2023).

Figura 3.3 – Número de operações MAC por ciclos de *clock*.



Fonte: Adaptado de (PREDIGER et al., 2023).

3.2.2 Realocação de constantes nas conversões RNS-binário

Nesta etapa do trabalho procura-se utilizar uma técnica de rearranjo matemático para minimizar os efeitos dos tempos na conversão RNS-binário. Nos estudos realizados obtiveram-se benefícios ao extrair as multiplicativas inversas da conversão no formato CRT e, simultaneamente, a elaboração de equações simples e mais escaláveis. Ao colocar a multiplicativa inversa na aritmética dos cálculos de multiplicações por constantes houve uma ampliação nas soluções de menor área e *delay*.

3.2.2.1 Aplicação no CRT

O *hardware* implementado utilizou o conjunto de 3 módulos escaláveis em n $\{m1,\ m2,\ m3\} = \{2^{n+1}-1,\ 2^n-1,\ 2^{n-1}-1\}$, com n par e, cuja equação do CRT genérica por 3 módulos, pode ser vista na equação 3.5.

$$X = \left| \sum_{i=1}^{3} \hat{m}_i \left| \hat{m}_i^{-1} \right|_{m_i} R_i \right|_{DR}, \tag{3.5}$$

onde R_i com $0 \leq i \leq 3$, indicam os resíduos de m_i respectivamente, para o qual usaremos a representação do *array* como $r_{i(n-1)}$, ..., r_{i0}, posteriormente. Neste trabalho assumimos que as operações aritméticas por canal são operações modulares por constantes, ou pelo menos a operação com melhor desempenho seja a modular por constante, indicado por $R_i = |c_i.X|_{m_i}$. Desta forma podemos incluir a multiplicativa inversa na referida operação modular e pode ser observado no conjunto de equações indicadas em 3.6.

$$\begin{aligned}
\left| \hat{m}_1 \left| \hat{m}_1^{-1} \right|_{m_1} R_1 \right|_{\hat{m}_1} &= \left| (2^{2n-1} - 2^{n+1} + 2^{n-1} + 2^0)(2^n - \sum_{i=1}^{\frac{n}{2}-1} 2^{2i} + 2^0) R_1 \right|_{DR}; \\
\left| \hat{m}_2 \left| \hat{m}_2^{-1} \right|_{m_2} R_2 \right|_{\hat{m}_1} &= \left| (2^{2n} - 2^{n+1} - 2^{n-1} + 2^0)(2^n - 2^2) R_2 \right|_{DR}; \\
\left| \hat{m}_3 \left| \hat{m}_3^{-1} \right|_{m_3} R_3 \right|_{\hat{m}_1} &= \left| (2^{2n+1} - 2^{n+2} - 2^n + 2^0)(\sum_{i=0}^{\frac{n}{2}-1} 2^{2i}) R_3 \right|_{DR}.
\end{aligned} \tag{3.6}$$

Como parte final do estudo, os circuitos com as duas implementações foram sintetizados, comparando entre a multiplicação por constantes com a aritmética modular tradicional (CRT original), como mostrado no diagrama em blocos da figura 3.4 (a), e uma abordagem deslocando as multiplicativas inversas para dentro da aritmética (CRT modificado), visto na figura 3.4 (b). Multiplicativas estas presentes na equação 3.5 e que são deslocadas, formando a equação 3.7.

$$X = \left| \sum_{i=1}^{3} \hat{m}_i S_i \right|_{DR}, \tag{3.7}$$

onde S_i são as operações residuais das multiplicações modulares por constantes, incluindo a multiplicativa inversa, indicada por $S_i = \left|c.|\hat{m}_i^{-1}|_{m_i}.X\right|_{m_i}$. A adição de termos da equação 3.5 para o conjunto de módulos apresentados é demonstrada no conjunto de equações 3.8.

$$\begin{aligned}
|\hat{m}_1 S_1|_{\hat{m}_1} &= \left|(2^{2n-1} - 2^{n+1} + 2^{n-1} + 2^0)S_1\right|_{DR}; \\
|\hat{m}_2 S_2|_{\hat{m}_1} &= \left|(2^{2n} - 2^{n+1} - 2^{n-1} + 2^0)S_2\right|_{DR}; \\
|\hat{m}_3 S_3|_{\hat{m}_1} &= \left|(2^{2n+1} - 2^{n+2} + 2^n + 2^0)S_3\right|_{DR},
\end{aligned} \tag{3.8}$$

onde os seis vetores resultantes comprimidos são escaláveis em n e estão representados na equação 3.9, em conjunto com um sétimo elemento $B_7 = COR$, que é o fator de correção.

$$\begin{aligned}
B_1 =& \overbrace{0\ldots0}^{n-2} * \overbrace{\bar{s}_{1,(n)}\ldots\bar{s}_{1,0}}^{n+1} * \overbrace{s_{1,(n)}\ldots s_{1,0}}^{n+1}; \\
B_2 =& \overbrace{s_{1,(n)}\ldots s_{1,0}}^{n+1} * \overbrace{s_{1,(n-1)}\ldots s_{1,0}}^{n} * \overbrace{0\ldots0}^{n-1}; \\
B_3 =& \overbrace{0\ldots0}^{n-1} * \overbrace{\bar{s}_{2,(n-1)}\ldots\bar{s}_{2,0}}^{n} *0 * \overbrace{s_{2,(n-1)}\ldots s_{2,0}}^{n}; \\
B_4 =& \overbrace{s_{2,(n-1)}\ldots s_{2,0}}^{n} * s_{1,(n)} * \overbrace{\bar{s}_{2,(n-1)}\ldots\bar{s}_{2,0}}^{n} * \overbrace{0\ldots0}^{n-1}; \\
B_5 =& \overbrace{0\ldots0}^{n} *0 * \overbrace{\bar{s}_{3,(n-2)}\ldots\bar{s}_{3,0}}^{n-1} *0 * \overbrace{s_{3,(n-2)}\ldots s_{3,0}}^{n-1}; \\
B_6 =& 0 * \overbrace{s_{3,(n-2)}\ldots s_{1,0}}^{n-1} * \overbrace{\bar{s}_{3,(n-2)}\ldots\bar{s}_{3,0}}^{n-1} * \overbrace{0\ldots0}^{n+1}; \\
B_7 =& COR = \overbrace{1\ldots1}^{n-3} *01 * \overbrace{0\ldots0}^{n-3} *10 * \overbrace{1\ldots1}^{n+1}.
\end{aligned} \tag{3.9}$$

A equação 3.5 pode ser reescrita como indicado na equação 3.10.

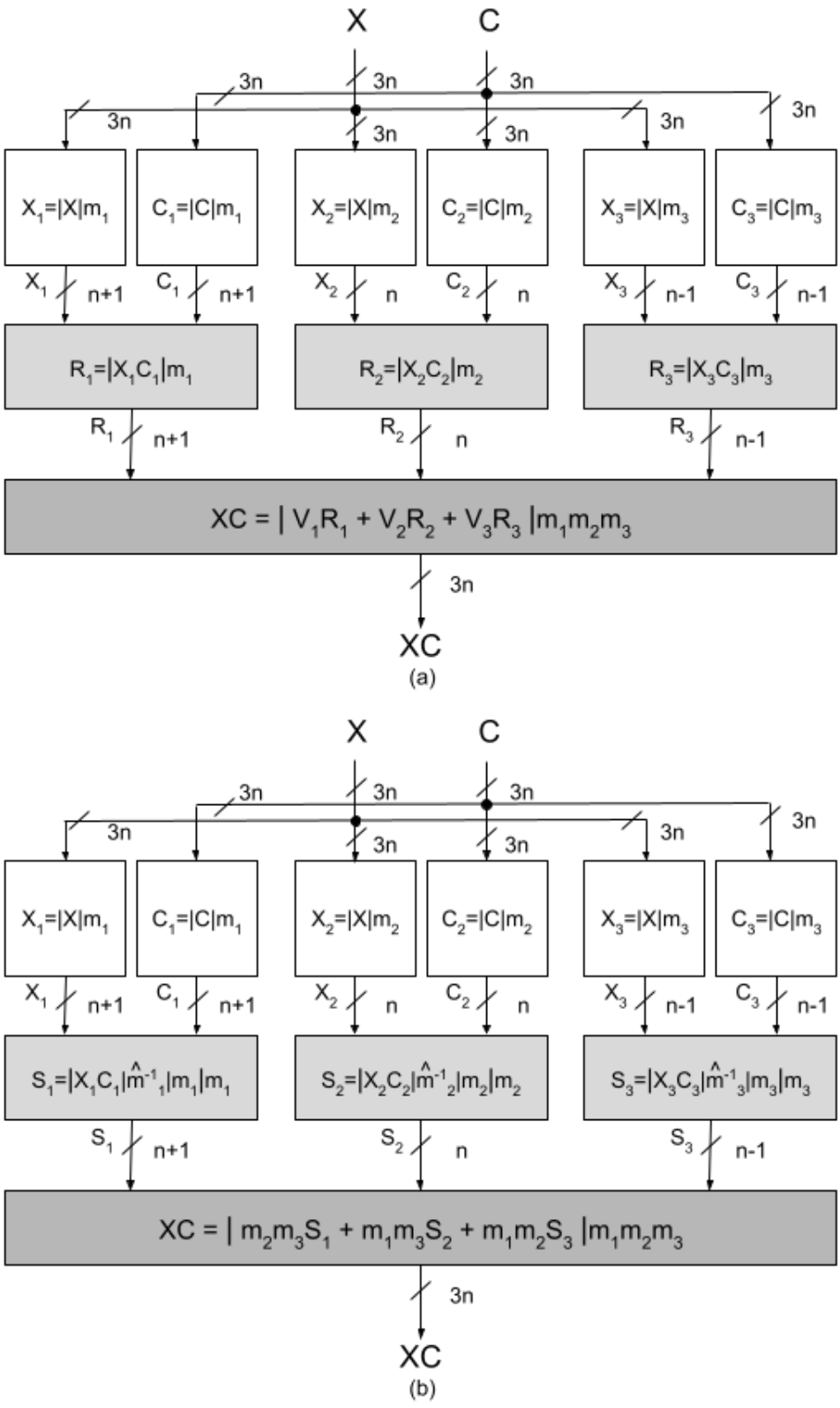
$$X = \left|\sum_{i=1}^{7} B_i\right|_{DR}. \tag{3.10}$$

,

Na conversão de RNS-binário estas multiplicativas entram cada qual no respectivo canal e, no caso de uma operação de multiplicação modular por uma constante, a mesma pode ser agrupada formando a equação 3.11, como exemplo para um canal genérico m_i.

$$S_i = \left|\overbrace{|C_i.X|_{m_i}}^{R_i}.|\hat{m}_i^{-1}|_{m_i}\right|_{m_i} = \left|\overbrace{\left|C_i.|\hat{m}_i^{-1}|_{m_i}\right|_{m_i}}^{P_i}.X\right|_{m_i} \tag{3.11}$$

Figura 3.4 – Operação XC em RNS, onde as multiplicativas inversas que estão feitas em (a), ocorrem no CRT e, em (b), na aritmética (proposta).



Fonte: Elaborado pelo autor.

Na tabela 3.4 observam-se os ganhos obtidos na multiplicação por constantes. Onde m_i é o módulo avaliado, C_i é a constante original e P_i a constante manipulada algebricamente. Isso permite uma operação de multiplicação modular mais eficiente com o uso da multiplicativa inversa.

Para os exemplos numéricos apresentados na tabela 3.4 utilizou-se um $C = 160915$ (constante original). Pode-se observar nesta tabela, que quando $m_1 = 127$ e $C_1 = 6$, tem-se uma área e *delay* iguais a 1 e que estes valores indicam, por exemplo, área e *delay* produzido por uma operação de soma modular. Já quando se aplica a multiplicativa inversa, que para o caso é $|\hat{m}_1^{-1}|_{m_1} = 106$, obtém-se um valor de $P_1 = 1$, que possui área e *delay* iguais a 0. Para o módulo $m_2 = 63$ a constante $C_2 = 13$, com área e *delay* iguais a 2, pode ser reduzida para zero em ambos com a multiplicativa inversa $|\hat{m}_2^{-1}|_{m_2} = 34$. Já para o módulo $m_3 = 31$ a constante beneficiada é a 25, usando a multiplicativa inversa $|\hat{m}_3^{-1}|_{m_3} = 5$.

Tabela 3.4 – Comparação entre C_i e P_i avaliando área e *delay*, (CONWAY, 2006). Exemplos para os módulos 127, 63 e 31

i	m	C_i	Área	*Delay*	P_i	Área	*Delay*
1	127	6	1	1	**1**	**0**	**0**
2	63	13	2	2	**1**	**0**	**0**
3	31	25	2	2	**1**	**0**	**0**

Fonte: Elaborado pelo autor.

Nas tabelas 3.5 e 3.6 tem-se o resultado do estudo que desloca os vetores das multiplicativas inversas para a aritmética modular. Nesse estudo foram testados 127 valores de constantes (de 0 a 126) para os três módulos $\{127,\ 63,\ 31\}$.

Na nomenclatura cunhada por (CONWAY, 2006), $A = 0$ (primeira linha e coluna) equivale a menor área possível representada por uma célula da operação de multiplicação por 2, por exemplo. $D = 0$ (segunda coluna) representa o menor tempo possível de uma determinada operação como a citada anteriormente, que pode ser operacionalizada pelo deslocamento de um bit em direção ao dígito mais significativo (deslocamento à esquerda), sem área ou *delay*.

Como exemplo, o desempenho dos 127 possíveis valores são apresentados nas tabelas 3.5 e 3.6. Para o caso $|C_i X_i|_{m_i}$ (tabela 3.5), há 20 constantes que apresentam *delay* sobre o total, quando se adiciona a multiplicativa inversa no segundo caso (tabela 3.6), $|C_i.|\hat{m}_i^{-1}|_{m_i} X_i|_{m_i}$, o número de constantes,que apresentam a mesma área, se reduz para 9.

O principal ganho desta abordagem é derivado da conversão inversa, que agora é simplificada devido ao deslocamento dos inversos multiplicativos. Neste estudo verificou-se que os benefícios crescem significativamente com o número de bits. Para $n = 20$, há uma redução de 90% na área e de 40% de melhoria no *delay*.

Tabela 3.5 – Distribuição das constantes sem o emprego das combinações

Área/*Delay*	0	1	**2**
0	12	-	-
1	-	8	-
2	-	20	1
3	-	28	14
4	-	-	24
5	-	-	**20**

Fonte: Elaborado pelo autor.

Tabela 3.6 – Distribuição das constantes com o emprego das combinações.

Área/*Delay*	0	1	**2**
0	6	-	-
1	-	11	-
2	-	17	-
3	-	20	28
4	-	-	36
5	-	-	**9**

Fonte: Elaborado pelo autor.

3.3 Aplicações para o módulo auxiliar de baixo custo nas operações de criptografia

3.3.1 Motivação

Uma das aplicações das operações modulares se dá nos processos criptográficos digitais. Neles a operação de exponenciação modular tem seu grande emprego, principalmente, em processos de transmissão de dados pela internet, assinatura digital e armazenamento de informações. Neste contexto, um exemplo de criptografia que mais utiliza esses cálculos é a RSA (Rivest Shamir and Adleman), onde um dos pontos que garante maior segurança nesta técnica criptográfica é o fato de uso de chaves relativamente grandes em comprimento de *bits* e a necessidade computacional de efetuar cálculos com valores de expoentes modulares igualmente grandes, durante o tráfego das informações computadas (STALLINGS, 2014).

3.3.2 Algoritmo para exponenciação modular rápida

Diante da necessidade de algumas aplicações, dentre elas a criptografia, em executar operações de exponenciação modular que são relativamente custosas no processamento, algumas técnicas foram desenvolvidas para minimizar estes custos. Dentre elas há a operação baseada no formato binário do expoente (MA, 2015). A notação apresentada

em 3.12 indica a operação de uma base X elevada à potência Y com a posterior operação com módulo m.

$$\left|X^Y\right|_m \tag{3.12}$$

A medida em que o expoente Y aumenta, maior é o esforço computacional para calcular as sucessivas multiplicações na obtenção da potência X^Y. Com o uso da abordagem de um cálculo mais direto, aplicando o algoritmo indicado a seguir (sequência de operações), as operações se tornam menores e pontuais, minimizando cálculos desnecessários.

1. Expandir o expoente Y no seu formato binário;
2. Executar a operação quadrática da série expandida de Y como x^{2^0} , x^{2^1} , x^{2^2} , ... ;
3. Cada operação quadrática é reduzida por módulo m;
4. Multiplicar os resultados do item 2 com os resultados reduzidos do módulo m. A última operação representa o resultado da operação desejada.

Por exemplo, deseja-se efetuar a operação $|3^{23}|_{29}$. Expandindo o expoente no seu formato binário tem-se:

$$23_{10} = \ [1_{b4}0_{b3}1_{b2}1_{b1}1_{b0}\ ]_2 = 1.2^4 \ + \ 0.2^3 \ + \ 1.2^2 \ + \ 1.2^1 \ + \ 1.2^0 \tag{3.13}$$

Executando a expansão e as multiplicações tem-se os valores da equação 3.14 e tabela 3.7. Cada "1" na representação do exponente significa uma multiplicação e cada "0" significa uma exponenciação.

$$3^{23} \equiv 3 \ . \ 9 \ . \ 23 \ . \ 20 \equiv \mathbf{27} \ . \ 23 \ . \ 20 \equiv \mathbf{12} \ . \ 20 \equiv \mathbf{8} \ |12 \ . \ 20|_{29} \tag{3.14}$$

Tabela 3.7 – Algoritmo de exponenciação rápida aplicado a um exemplo numérico.

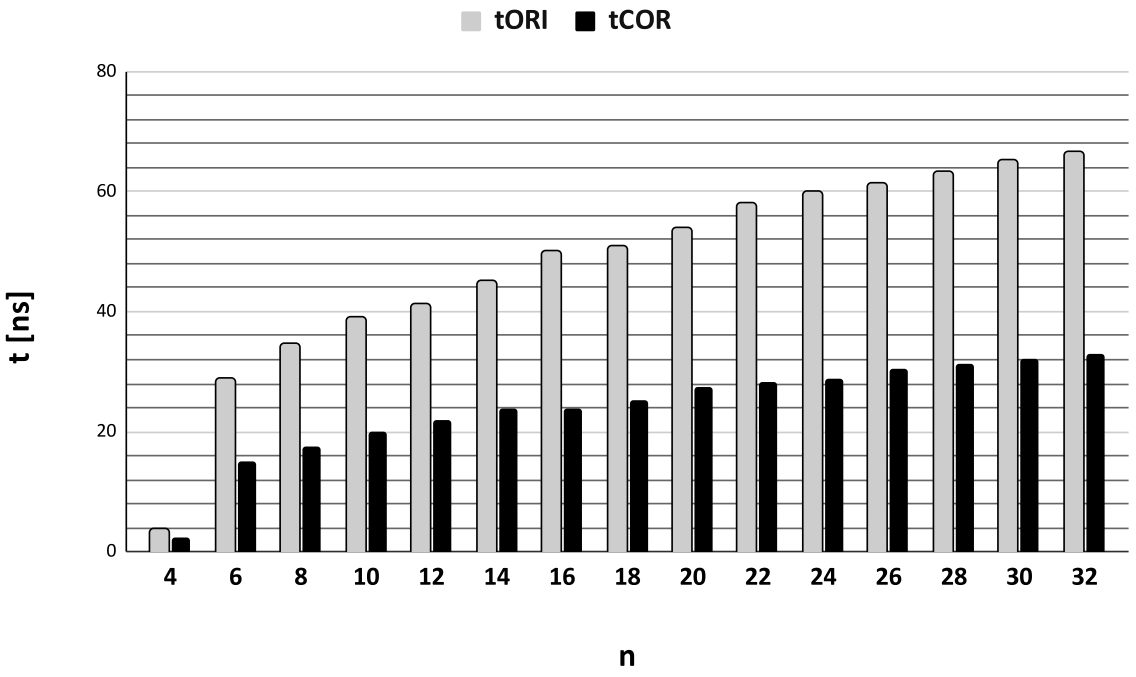
i	$Quadrado \mid \mid_{29}$	$Produto \mid \mid_{29}$
0	$3*$	3
1	$9*$	27
2	$23*$	12
3	$7*$	–
4	$20*$	8

Fonte: Adaptado de (MA, 2015).

3.3.3 Potencializando com módulo auxiliar

Tendo como motivação os estudos realizados com as operações modulares simples, em *hardware*, e seus resultados promissores, buscou-se realizar a análise para as operações de exponenciação modular com o uso dos módulos auxiliares (vistos na seção 3.2.1).

Figura 3.5 – Exponenciação modular comparação de tempos.



Fonte: Elaborado pelo autor.

Na figura 3.5 pode-se observar uma comparação entre o tempo da operação com o módulo original (t_{ORI}) e o tempo da operação do módulo auxiliar já corrigido (t_{COR}), utilizando operadores de 32 *bits* e aplicando uma técnica de compressão. O índice COR indica que a implementação utilizou o módulo auxiliar e depois aplicou a correção necessária para se obter o resultado equivalente (propriedade vista na equação 3.1). Na figura 3.5 verifica-se que a diferença de tempos vai aumentando de maneira gradual indicando uma vantagem no uso dos módulos auxiliares corrigidos. A tecnologia usada é a FPGA MAX10

da Intel®e os detalhes sobre a correção serão vistos em capítulos e seções posteriores, dedicados ao *software*.

O processo de correção é obtido seguindo a equação 3.15, aplicando a propriedade das operações modulares (vista na equação 3.1).

$$\overbrace{\left|X^Y\right|_{m_{ORI}}}^{op.\ original} = \overbrace{\left|\overbrace{\left|X^Y\right|_{m_{AUX}}}^{op.\ auxiliar}\right|_{m_{ORI}}}^{op.\ corrigida} \tag{3.15}$$

Destes ensaios é possível concluir que há uma vantagem no emprego dos módulos auxiliares em relação aos originais, resultando em *delays* cada vez menores (na comparação direta), a medida em que se aumenta o tamanho dos registradores utilizados.

Baseados nos resultados promissores apresentados em *hardware*, principalmente com relação ao uso de módulos auxiliares na exponenciação modular, pretende-se agora elaborar estudos e ensaios, voltados ao *software*, com o uso de linguagens de programação com relativa aceitação no ambiente da tecnologia da informação. O objetivo principal é avaliar a existência de benefícios semelhantes dos encontrados em *hardware*, primeiramente, aplicando a operação modular simples (capítulo 4) e após em exponenciação modular aplicada a criptografia RSA (capitulo 5). A escolha pela avaliação do desempenho dos cálculos criptográficos (com foco em RSA), em *software*, se dá pelo grande volume que este tipo de processamento ocupa nas tarefas cotidianas desde as plataformas *desktop* até operações em grandes servidores de dados e serviços (*mainframes*). Processamentos estes que são os pilares dos serviços de transações comerciais, trocas de informações por correios eletrônicos, segurança em redes privadas de dados, entre outras aplicações que exigem confiabilidade e autenticidade.

4 Módulo de melhor desempenho em aplicações de *software*.

Neste capítulo serão abordados os estudos que avaliam o desempenho da operação modular simples usando as linguagens de programação Python, Java e Go. Os módulos utilizados são os do formato $2^n - 1$ e $2^n + 1$ e os estudos são finalizados aplicando-se conceitos de produtos notáveis, na fatoração dos exponentes, com o emprego dos recursos de paralelismo da arquitetura utilizada.

4.1 Motivação

Buscou-se uma aplicação de operações modulares como aritmética e exponenciação em programação, baseado nos estudos promissores das sínteses de *hardware* do capítulo 3, utilizando-se de algumas linguagens com aceitação no setor tecnológico e aplicando-as a sub-rotinas de programação nas práticas de criptografia tradicional. A técnica RSA foi usada para a aplicação neste caso particular. Parte das linguagens escolhidas para o estudo foram derivadas de sua já consagrada aceitação/utilização (OVERFLOW, 2008), enquanto outras foram adicionadas devido ao seu crescimento de uso, relativamente recente (devmedia, 2019).

Para os estudos e análises a seguir alguns conceitos e técnicas foram empregados para as medições de tempo (apêndice A), estatística dos resultados (apêndice B) e elaboração dos algoritmos (apêndices D, E, F e G) em conjunto com as definições de *pipeline* e paralelismo necessários nas interpretações e execução de alguns algoritmos.

Abaixo seguem mais nomenclaturas padronizadas para o trabalho:

- t_{AUX} = Tempo para o Módulo Auxiliar;
- t_{ORI} = Tempo para o Módulo Original;
- t_{COR} = Tempo para Módulo Auxiliar corrigido = Tempo do Módulo Auxiliar + Tempo da Correção.

O t_{COR} resulta da soma dos tempos das operações com o módulo auxiliar e das operações que efetuam a correção, independente do método adotado.

4.2 Análise de sensitividade modular em linguagens de software

Nesta etapa procura-se descrever os resultados de desempenho obtidos ao aplicar as operações modulares simples e exponenciação modular avaliando o tempo de execução com os módulos no formato $2^n \pm k$ e comparando com os do formato $2^a \pm 1$. Essa avaliação se deu com a implementação de códigos e funções nas linguagens de programação Go Lang, Java e Python. Linguagens estas já consagradas por seus empregos em computação na nuvem e por apresentarem um alto nível de verbosidade.

4.2.1 Uso da linguagem Python na operação de módulo simples $X \ \% \ m$

Para o ensaio demonstrado na figura 4.1 foram realizadas variações de n de 34 até 50, no passo 2 para módulos no formato $2^n + k$, variando também os valores de k conforme a sequência a seguir: $\{ -5, -3, -1, 0, 1, 3, 5 \}$. Apenas como exemplo didático: se $n = 34$, então $m = 2^{34} + k$, de modo que m assume os seguintes valores: $\{(2^{34} - 5), (2^{34} - 3), (2^{34} - 1), 2^{34}, (2^{34} + 1), (2^{34} + 3), (2^{34} + 5)\}$ e na figura 4.1 ilustra-se o ensaio.

Os valores de entrada (operador X) possuem uma relação do dobro de *bits* do módulo m_{AUX} e apresenta uma característica de intercalação entre uns (1s) e zeros (0s) quando a mesma é observada no seu formato binário. Apenas como exemplo, se o módulo possui 34 *bits* então a entrada apresentará 68 *bits* e terá o formato binário de $1010...1010_2$, indicando a intercalação.

A operação realizada encontra-se descrita na equação 4.1, onde R é uma variável que guarda o valor do processamento e o símbolo % é a operação modular simples, realizada na linguagem Python.

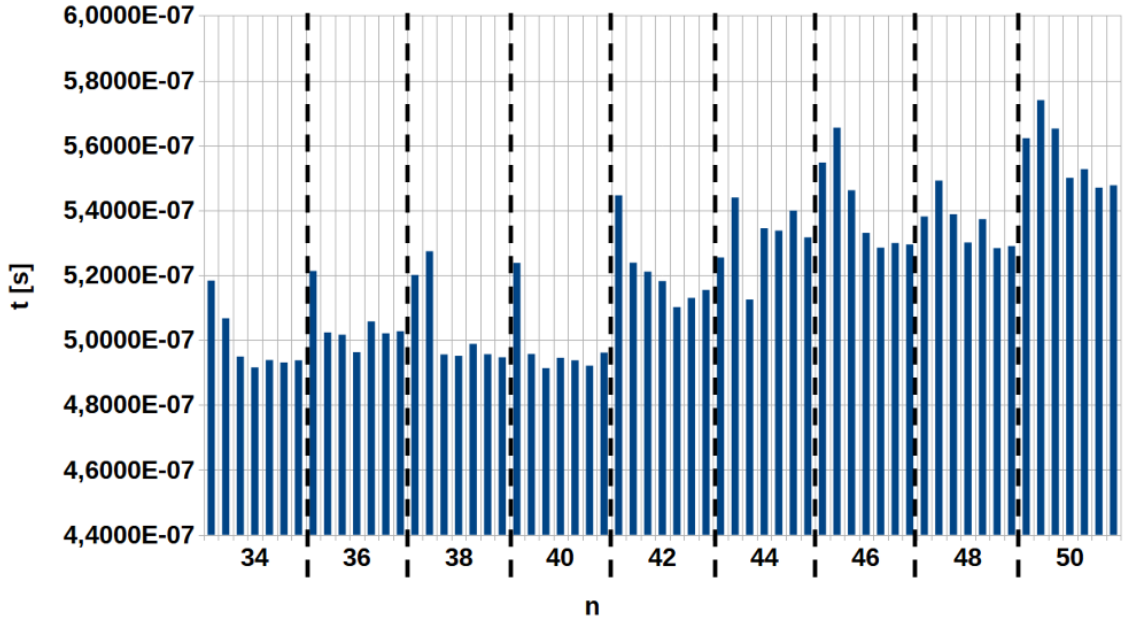
$$R = |X|_{2^n+k} = X \ \% \ (2^n + k) \tag{4.1}$$

O resumo dos ajustes iniciais do ensaio podem ser vistos abaixo:

- $X = 1010...1010_2$;
- X com o comprimento do dobro de *bits* de n;
- módulo utilizado: $2^n + k$;
- $n = 34 \sim 50$ com passo 2;
- $k = \{ -5, -3, -1, 0, 1, 3, 5 \}$.

O resultado do ensaio pode ser visto na figura 4.1, onde as retas verticais tracejadas indicam a separação de cada valor de n.

Figura 4.1 – Ensaio com Python para a operação módulo simples, variando o k.



Fonte: Elaborado pelo autor.

Existe uma variação crescente e gradativa (numa visão geral) nos tempos à medida que se aumenta o valor do módulo. Observa-se uma tendência de subida nos tempos a medida que o módulo aumenta e também é possível verificar que há variações internas indicando que, mesmo com módulos maiores, é possível obter tempos menores, mas sem uma sensitividade explícita de módulos no formato $2^n \pm 1$ em comparação ao $2^n \pm k$.

4.2.1.1 Ensaio comparativo da operação de módulo simples com módulo $2^n - 1$

O resumo dos ajustes iniciais do ensaio, para $m = 2^n - 1$, podem ser vistos abaixo:

- módulo auxiliar (m_{AUX}): $2^n - 1$;
- $m_{AUX} = 3.(m_{ORI})$.
- $n = 102 \sim 200$ com passo 2;
- correção: execução das duas operações módulo simples na mesma linha de comando.

Os valores da entrada X foram explorados até que se pode obter o melhor resultado possível e cujos ensaios encontram-se descritos a seguir.

- X possui a mesma quantidade de 1s que o número de *bits* dos módulos mas deslocado um dígito a esquerda (multiplicado por 2). Neste ensaio o resultado foi satisfatório e está representado na figura 4.2.
  - Como exemplo, se m possuir 4 *bits* então $X = 11110_2$.

Para a obtenção dos resultados indicados na figura 4.2 avaliou-se o tempo de execução das linhas de comando que efetuam a operação modular simples e cuja entrada X obedeceu às regras de formato indicados no início desta seção. As operações/comandos executados encontram-se demonstradas nas equações/operações 4.2, 4.3 e 4.4 onde o operador % é um dos comandos em Python que executa a operação modular simples.

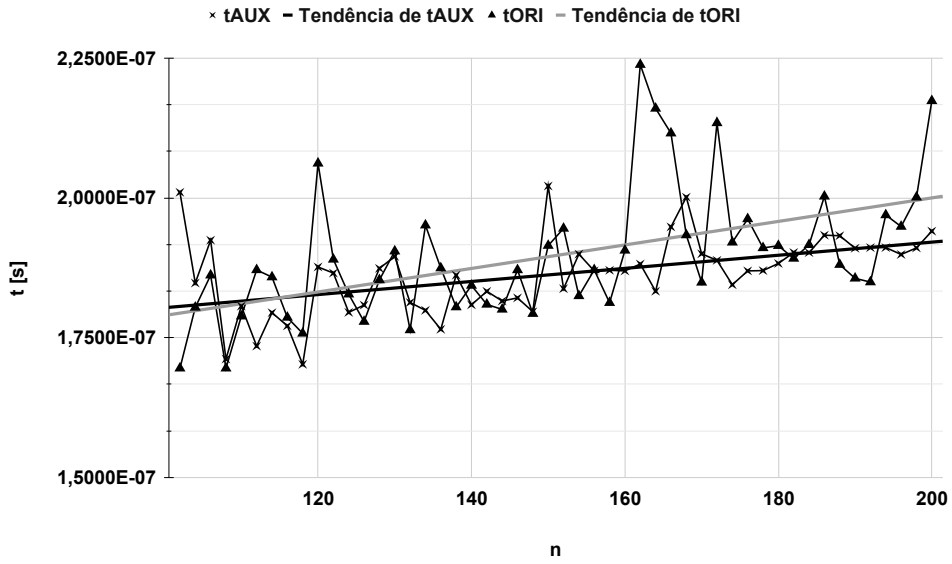
$$R_{AUX} = |X|_{2^n-1} = X \ \% \ m_{AUX} \tag{4.2}$$

$$R_{ORI} = |X|_{2^{n-2}+k} = X \ \% \ m_{ORI} \tag{4.3}$$

$$R_{COR} = \Big||X|_{2^n-1}\Big|_{2^{n-2}+k} = (X \ \% \ m_{AUX}) \ \% \ m_{ORI} \tag{4.4}$$

Na equação 4.2 foi executada a operação modular simples usando o módulo auxiliar, armazenando em R_{AUX}, enquanto em 4.3 foi executada a mesma operação usando o módulo original e guardando em R_{ORI}. Já na equação 4.4 foi processado o cálculo que abrange tanto a equação 4.2 quanto 4.3 (correção completa) sendo armazenado em R_{COR}.

Figura 4.2 – Medição de tempos da operação módulo simples para t_{AUX} e t_{ORI}, no formato $2^n - 1$ e com X no formato do ensaio C. Linguagem Python.



Fonte: Elaborado pelo autor.

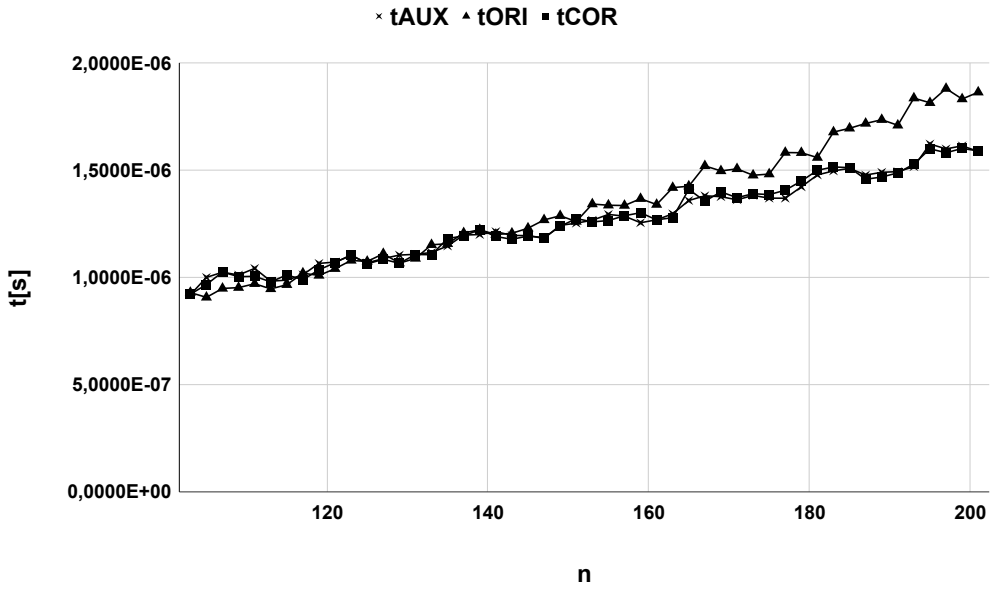
Pode-se observar, na figura 4.2 que exitem aproximações significativas entre t_{AUX} (tempo de execução do módulo auxiliar) e t_{ORI} (para o módulo original), inclusive pontos onde o auxiliar apresentou uma eficiência melhor e as linhas de tendência indicam um melhor desempenho para t_{AUX}. A partir da seção 4.2.4, serão realizados outros ensaios que trarão mais eficiência para este formato de módulo.

4.2.1.2 Ensaio comparativo da operação de módulo simples com módulo $2^n + 1$

O resumo dos ajustes iniciais do ensaio, para $m = 2^n + 1$, podem ser vistos abaixo:

- $X = 1010...1010_2$. Entrada com dígitos intercalados e o dobro de bits do módulo;
- X com o comprimento do dobro de *bits* de n;
- módulo auxiliar (m_{AUX}): $2^n + 1$;
- módulo original (m_{ORI}): $2^{n-2} + k$;
- $\overbrace{2^n + 1}^{m_{AUX}} = 3.\overbrace{2^{n-2} + k}^{m_{ORI}}$
- $n = 103 \sim 200$ com passo 2.

Figura 4.3 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI} e t_{COR}. Linguagem Python.



Fonte: Elaborado pelo autor.

No gráfico da figura 4.3 pode-se verificar que há uma tendência de coincidência entre os tempos do módulo já corrigido t_{COR} e auxiliar t_{AUX}, indicando uma vantagem no uso do módulo auxiliar já corrigido, pois o tempo do original t_{ORI} apresenta um desempenho em desvantagem gradativa.

4.2.2 Java - operação módulo - (X).mod(m)

As preparações para os ensaios usando a linguagem Java seguem de forma idêntica das realizadas com Python, apenas fazendo adequações as características de sua sintaxe. Alguns testes foram realizados utilizando o módulo no formato $2^n - 1$, na mesma faixa que a linguagem Python, e em virtude dos resultados preliminares insatisfatórios, partiu-se diretamente para o uso do nódulo no formato $2^n + 1$.

As operações realizadas, e respectivas sintaxes podem ser vistas em 4.5, 4.6 e 4.7.

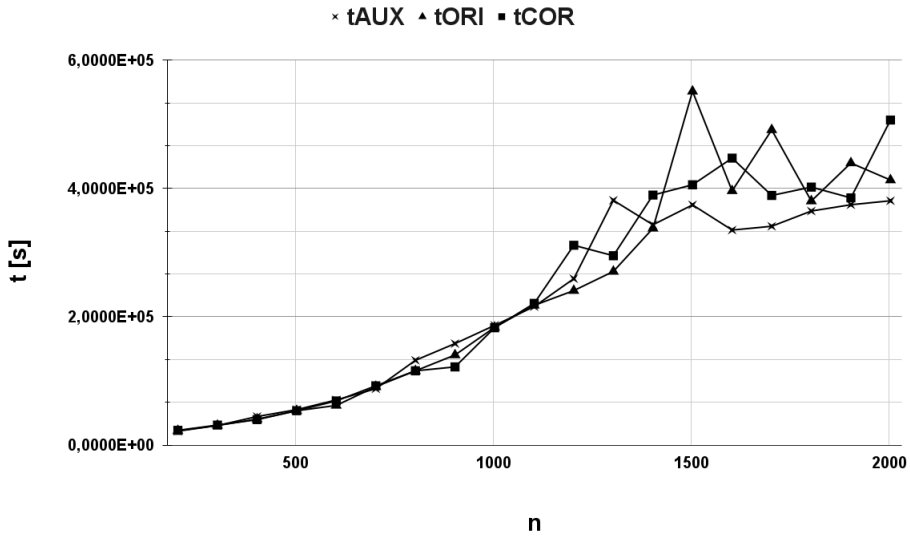
$$R_{AUX} = |X|_{2^n+1} = (X).mod(m_{AUX}) \tag{4.5}$$

$$R_{ORI} = |X|_{2^{n-2}+k} = (X).mod(m_{ORI}) \tag{4.6}$$

$$R_{COR} = \Big||X|_{2^n+1}\Big|_{2^{n-2}+k} = ((X).mod(m_{AUX})).mod(m_{ORI}) \tag{4.7}$$

Os ensaios com o módulo $2^n + 1$ demonstram, numa tendência, valores mais promissores de desempenho. Nos ensaios seguintes procura-se ampliar as faixas de estudo até um valor, de tamanho em *bits* para n, com aplicações reais, tais como criptografia. Os resultados são vistos na figura 4.4 para módulos $2^n + 1$.

Figura 4.4 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI}, t_{COR} e $m_{AUX} = 2^n + 1$. Na linguagem Java.



Fonte: Elaborado pelo autor.

4.2.3 Go Lang - operação módulo - .mod(X, m)

As preparações para os ensaios usando a linguagem Go seguem de forma idêntica das realizadas com Python, apenas fazendo adequações as características de sua sintaxe (operações 4.8, 4.9 e 4.10). Alguns testes foram realizados utilizando o módulo no formato $2^n - 1$, na mesma faixa que a linguagem Python, e em virtude dos resultados iniciais insatisfatórios, partiu-se novamente para o uso do nódulo no formato $2^n + 1$.

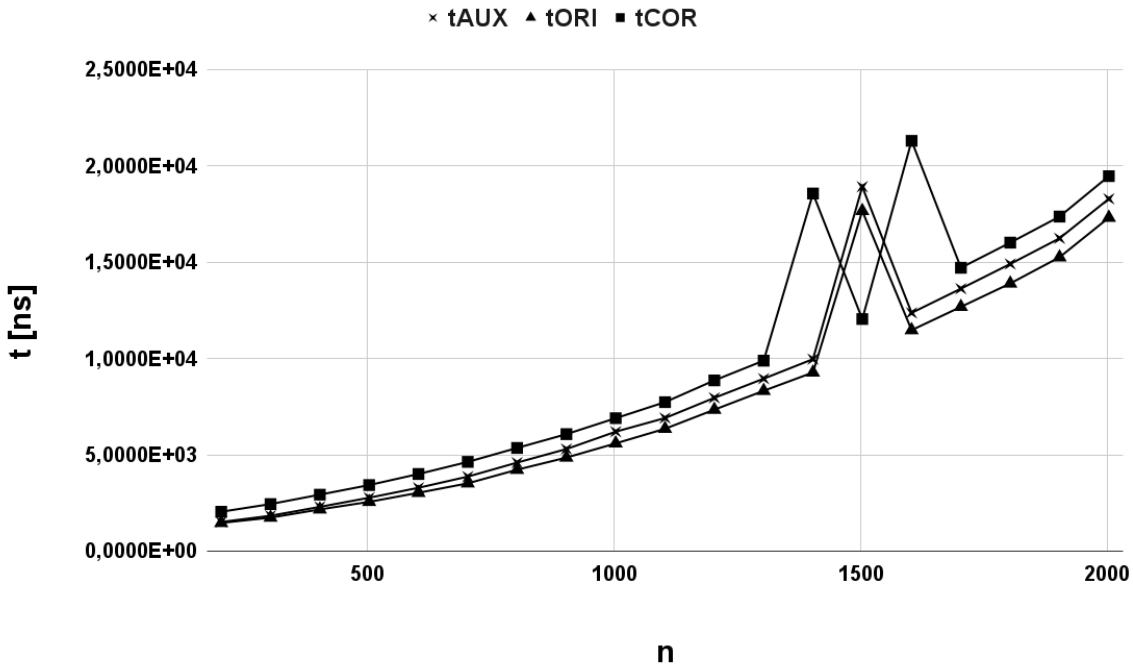
$$R_{AUX} = |X|_{2^n+1} = R.Mod(X, m_{AUX}) \tag{4.8}$$

$$R_{ORI} = |X|_{2^{n-2}+k} = R.Mod(X, m_{ORI}) \tag{4.9}$$

$$R_{COR} = \Big||X|_{2^n+1}\Big|_{2^{n-2}+k} = R.Mod(R.Mod(X, m_{AUX}), m_{ORI}) \tag{4.10}$$

Os resultados são vistos na figura 4.5. Nota-se que há um distanciamento na curva do tempo de correção (t_{COR}), em relação as demais, mas apresentando algumas vantagens em *outliers*.

Figura 4.5 – Medição de tempos da operação módulo simples - t_{AUX}, t_{ORI}, t_{COR} e $m_{AUX} = 2^n + 1$. Na linguagem Go.



Fonte: Elaborado pelo autor.

A sequência de ensaios, nas três diferentes linguagens, apontam para uma vantagem no uso do Python. As demais linguagens como Java e Go apresentam alguma vantagem, numa perspectiva de tendência, para valores de n com faixas acima das aplicadas na atualidade, que são de 2048 *bits*. Ainda avaliando os ensaios em Python, percebe-se que módulos no formato $2^n - 1$ apresentam uma vantagem, quando são exploradas novas possibilidades da entrada X, em relação aos respectivos originais. Na seguinte seção será aplicada, algebricamente, uma técnica que trará melhor desempenho para estes indicadores.

4.2.4 Operação modular com fatoração de termos

Os ensaios a seguir realizam uma abordagem, da operação modular em *software*, aplicando a fatoração no expoente do módulo de maneira a produzir menor impacto no cálculo. O método divide o módulo em dois produtos baseando-se na propriedade do produto da soma pela diferença (propriedade dos produtos notáveis), demonstrada na equação 4.11.

$$\overbrace{2^n - 1}^{m_{AUX}} = \overbrace{(2^{n/2} - 1)}^{m_a} . \overbrace{(2^{n/2} + 1)}^{m_b}, \tag{4.11}$$

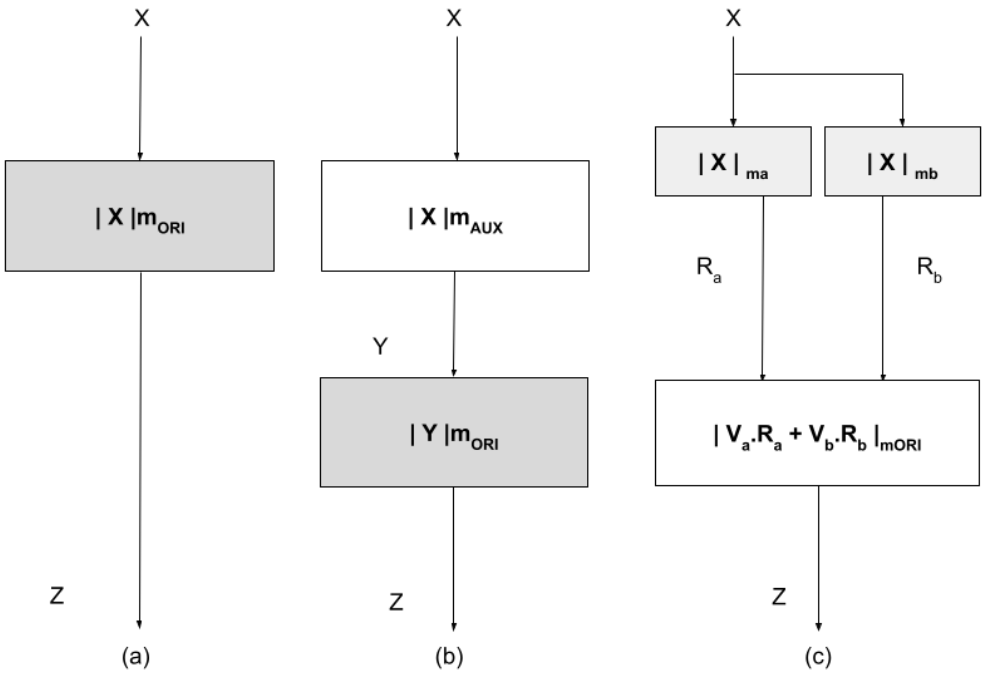
onde $m_a = 2^{n/2} - 1$ e $m_b = 2^{n/2} + 1$.

Para os estudos a seguir algumas nomenclaturas importantes devem ser apresentadas e podem ser vistas abaixo:

- m_a e m_b: módulos fatorados de m_{AUX};
- t_a: tempo de execução utilizando o módulo m_a;
- t_b: tempo de execução utilizando o módulo m_b;
- t_{ab}: tempo de execução utilizando os módulos a e b, de forma paralela, e que representa o caminho crítico (maior *delay*) entre t_a e t_b;

Na figura 4.6 (a) tem-se a operação modular direta com o módulo original m_{ORI}, na figura 4.6 (b) a operação com o módulo auxiliar m_{AUX} e posterior correção $|X|_{m_{ORI}}$, e na figura 4.6 (c) as operações com os módulos fatores de m_{AUX} que são m_a e m_b e uma correção final baseado em CRT.

Figura 4.6 – Operações modulares empregadas e suas respectivas equivalências e correções.



Fonte: Elaborado pelo autor.

Para aplicar a equação 4.11 o exponente de m_{AUX} deve ser par, também é importante explorar as diretivas de paralelismo das linguagens de programação de modo a se obter uma execução mais sincronizada possível, nas operações com os módulos m_a e m_b, como

também evitar que as mesmas disputem recursos de processamento na arquitetura em questão.

Como exemplo numérico, assumindo que $m_{ORI} = 21$, então $m_{AUX} = 3.21 = 63$. Como $m_{AUX} = 2^n - 1 = 2^6 - 1$, então m_a e m_b podem ser escritos como $m_a = 2^{\frac{6}{2}} - 1 = 7$ e $m_b = 2^{\frac{6}{2}} + 1 = 9$. A equação modular final pode ser vista em 4.12.

$$\Big|X\Big|_{m_{ORi}} = \Big|\Big|V_a.R_a + V_b.R_b\Big|_{63}\Big|_{21}, \tag{4.12}$$

onde $R_a = |X|_{m_a}$ e $R_b = |X|_{m_b}$.

Os valores de V_a e V_b podem ser obtidos com o uso de um sistema de equações do primeiro grau, onde se atribui valores para X que deixam os resíduos R_a e R_b de tal modo que podem ser isolados e obtém-se assim, uma resultante. Por exemplo, se $X = 70$ para definir V_b e $X = 72$ para definir V_a então, os valores podem ser obtidos pelo sistema descrito no conjunto de equações indicadas em 4.13, 4.14 e 4.15.

$$\begin{aligned} R_a = |70|_7 = 0\ ,\ R_b = |70|_9 = 7\ &\longrightarrow\ \Big|V_b.7\Big|_{63} = |70|_{63} = 7 \\ R_a = |72|_7 = 2\ ,\ R_b = |72|_9 = 0\ &\longrightarrow\ \Big|V_a.2\Big|_{63} = |72|_{63} = 9 \end{aligned} \tag{4.13}$$

$$V_a = m_b.|\hat{m_a}^{-1}|_{m_a} \tag{4.14}$$

$$V_b = m_a.|\hat{m_b}^{-1}|_{m_b} \tag{4.15}$$

Dando sequência aos ensaios procurou-se aplicar algumas técnicas para acelerar as operações, dentre elas o uso dos recursos de paralelismo que estão presentes em sistemas operacionais da atualidade e podem ser vistos nos testes posteriores.

Na figura 4.7 demonstra-se o resultado do uso da função *xargs* que executa dois ou mais processos (comandos) de forma paralela e cujo processo de distribuição é transparente ao usuário. No exemplo foram executados o *script* `paralelo_s1.sh` que executa a soma de resultado 30 e apresentando um tempo de $0,018s$, logo após o `paralelo_s2.sh` com retorno 300 e tempo de $0,018s$ e por final o *script* `paralelo_01.sh` que possui agrupada às duas funções das somas anteriores e cujo tempo de execução foi de $0,025s$. Agrupamento este que se dá pela distribuição das operações em dois processamentos paralelos.

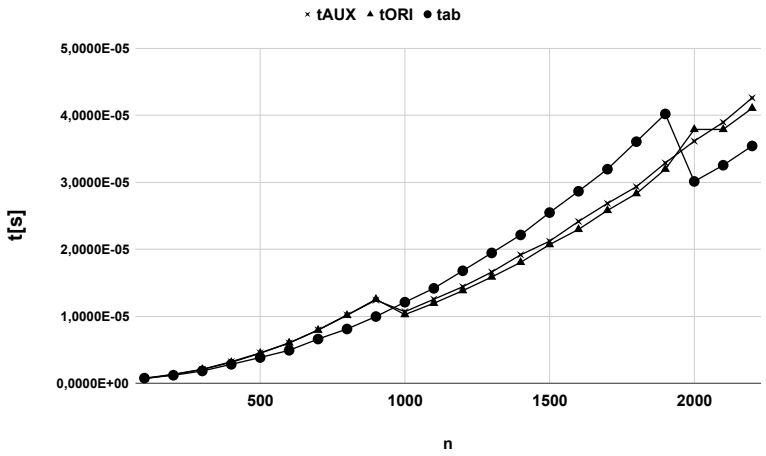
Figura 4.7 – Medicação do tempo da função de soma simples usando recursos do paralelismo.

Fonte: Elaborado pelo autor.

No exemplo utilizado as operações executadas em separado levam $0,018s$ na sua execução. Então, se fossem executadas em sequência, tem-se um tempo de $0,036s$. Já em paralelo à execução leva $0,025s$ o que representa uma redução aproximada de 30%.

No ensaio da figura 4.8, m_{AUX} representa o módulo auxiliar, m_a e m_b representam os módulos fatorados onde o expoente é divido por 2, m_{ORI} é o módulo original e X obedece o formato intercalado da seção 4.2.1.2.

Figura 4.8 – Paralelismo: comparação entre os tempos dos módulos m_{ORI}, m_{AUX} e sua fatoração m_{ab}.



Fonte: Elaborado pelo autor.

Pode-se observar na figura 4.8, que a operação paralela de m_a e m_b apresenta uma relativa vantagem numa faixa inicial do ensaio entre 500 e 900 e acima de 2000 (incluindo). Quando então apresenta-se um distanciamento exponencial (aproximado) em relação a m_{ORI} e m_{AUX}, respectivamente o segundo e terceiro piores tempos. Ainda na figura 4.8, o tempo indicado por t_{ab} representa uma primeira abordagem para o módulo no formato $2^n - 1$ fatorado, sem a devida correção, pois se trata de um ensaio inicial de viabilidade da abordagem. Também é importante destacar a importância de se explorar valores da entrada (X) para se obter um melhor desempenho, como realizado na seção 4.2.1.1. Nesta primeira abordagem é possível perceber que, para valores de $n \geq 2000$, pode-se o observar um ganho significativo e que indica uma tendência promissora de desempenho. Valores de $n \geq 2000$ se justificam, pois, são ordens de grandeza aplicadas em sistemas de criptografia computacional da atualidade e cujos ensaios podem ser vistos no capítulo 5.

5 Módulo de melhor desempenho em aplicações criptográficas de software

5.1 Motivação

Diante da necessidade de otimização nos processos criptográficos computacionais da atualidade, principalmente na economia de tempo e de recursos energéticos, o presente trabalho vem contribuir com alternativas no cálculo da exponenciação modular, sendo uma das etapas importantes no processamento criptográfico.

A criptografia computacional, atualmente, vem sendo utilizada em uma gama crescente de aplicações desde os processos de assinatura digital até os sistemas baseados em cripto ativos (transações financeiras). Ela está presente em praticamente todas as plataformas computacionais atuais como sistemas de grande porte (*mainframes*), computação na nuvem, equipamentos *desktop*, dispositivos pessoais portáteis e também em aplicações mais dedicadas (embarcadas). Estas plataformas executam atividades que exigem um aporte considerável de recursos energéticos naturalmente limitados (e guardadas as devidas proporções), fazendo com que todo e qualquer esforço na direção de um mínimo ganho na otimização trará benefícios à toda a cadeia.

5.2 A criptografia RSA

O RSA surge como proposta de técnica criptográfica baseado no modelo de chave pública para aplicações genéricas e de fácil implementação (STALLINGS, 2010).

Basicamente o algoritmo RSA se constitui de um bloco cifrador onde tanto o texto original como o cifrado são números inteiros na DR do módulo utilizado. As equações 5.1 e 5.2 indicam as relações entre o texto original X (mensagem) e a sua respectiva codificação C.

$$C = \left| \, X^{Y} \, \right|_{m} \tag{5.1}$$

$$X = \left| \, C^{d} \, \right|_{m} = | \, X^{Y} \, |_{m}^{d} = | \, X^{Yd} \, |_{m} \tag{5.2}$$

Ambos, emissor e receptor, conhecem o valor de m, o valor de Y é conhecido pelo emissor e apenas o receptor conhece o valor de d, formando a chave pública $PU = \{Y \, , m\}$ e privada $PR = \{d \, , m\}$. Este tipo de abordagem traz algumas características ao algoritmo (STALLINGS, 2010) e que estão descritas abaixo:

- Valores de Y , d , m. que atendam $X = \left|X^{Yd}\right|_m$, são relativamente fáceis de encontrar;
- $\left|X^{Y}\right|_m$ e $\left|C^{d}\right|_m$ apresentam uma relativa facilidade de cálculo para $X < m$;
- Dependendo da velocidade das transações, a descoberta de d, informando-se Y e m, torna-se inviável.

Observando as relações dadas em 5.2 tem-se que Y e d são multiplicativas inversas $\phi(m)$ (função ϕ de Euler) e utilizando dois números primos p e q tem-se $\phi(pq) = (p-1)(q-1)$, então a relação entre Y e d pode ser expressa pelas equações 5.3 e 5.4.

$$\left|Y.d\right|_{\phi(m)} = 1 \tag{5.3}$$

Equivalendo à:

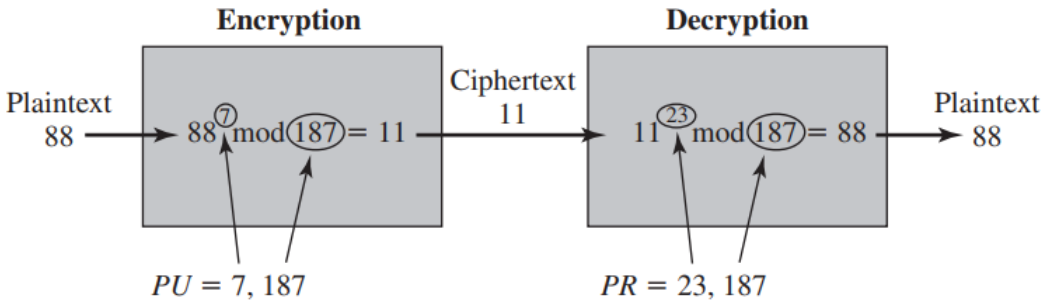
$$|Y^{-1}|_{\phi(m)} = d \tag{5.4}$$

Como exemplo numérico e de definição da sequência básica para as configurações elementares da criptografia RSA, tem-se que:

- p , q são dois números primos | (privados e escolhidos);
  - no exemplo dado, $p = 17$ e $q = 11$
- $m = p.q$ | público e calculado. m é um semi primo (número formado pelo produto de dois primos);
  - para o exemplo $m = p.q = 17.11 = 187$
- Y é tal que $gdc(\phi(m)\ ,\ Y) = 1$ e $1 < Y < \phi(n)$ | público e escolhido;
  - calculando $\phi(m) = (p-1)(q-1) = 16.10 = 160$;
  - escolhe-se $Y = 7$.
- $d \equiv\ |Y^{-1}|_{\phi(m)}$ | privado e calculado
  - $d = 23$, pode ser definido com o uso do algoritmo estendido de Euclides e obedece ao seguinte:
    - $1 \equiv \left|d.Y\right|_{160}$ com $d < 160$;
    - $23.7 = 161 = (1.160) + 1$.

Na figura 5.1 verifica-se a forma operacional do cálculo em diagrama de blocos (STALLINGS, 2010).

Figura 5.1 – Exemplo de aplicação da criptografia RSA.



Fonte: (STALLINGS, 2010).

5.3 Resultados experimentais para RSA usando módulo de melhor desempenho

5.3.1 Exemplo de RSA usando módulos auxiliares

A seguir serão descritos os ensaios realizados usando a exponenciação modular com aplicações em alguns exemplos de criptografia baseado em RSA e os respectivos ganhos entre os módulos originais e auxiliares. Cabe destacar que na criptografia RSA tanto o texto original (X) como o cifrado (C) são inteiros baseados no módulo m.

5.3.2 RSA-100

O RSA-100, cujos resultados aparecem na tabela 5.1, possui 100 dígitos decimais para a mensagem (texto original ou X) o que equivale a 330 *bits* e Y representa o expoente utilizado no processo de cifragem (codificação) para produzir C, conforme indicado na equação 5.6. A quebra (fatoração) nos dois primos iniciais ocorreu em 1991 (DENNY et al., 1994).

$$X = O\ mistério\ é\ agir\ no\ fluxo\ natural. \tag{5.5}$$

$$C = \left| X^Y \right|_m \tag{5.6}$$

Para facilitar o entendimento dos próximos ensaios, retoma-se algumas nomenclaturas apresentadas anteriormente com a inserção de novas definições, vistas abaixo:

- t_{AUX} = Tempo para o Módulo Auxiliar (m_{AUX});

- t_{ORI} = Tempo para o Módulo Original (m_{ORI});
- k_{ORI} = Compõe o módulo original, $m_{ORI} = 2^n + k_{ORI}$;
- t_{COR} = Tempo para Módulo Auxiliar corrigido = Tempo do Módulo Auxiliar + Tempo da Correção.

No ensaio, a mensagem passou pela conversão para o formato ASCII (American Standard Code for Information Interchange) de modo a se aproximar da forma como é utilizada nos processos de cifragem.

$$X_{ASCII} = 79321091051151162331141051113223332971031051143211011132102108117120111321109711611711497108461 \quad (5.7)$$

Tabela 5.1 – Operação módulo simples com exemplo similar a RSA-100, t_{ORI} *versus* t_{AUX}, para $Y = 2047$.

$\mathbf{m_{ORI}}$	$\mathbf{t_{ORI}}[\mu s]$	$\mathbf{m_{AUX}}$	$\mathbf{t_{AUX}}[\mu s]$	**Ganho[%]**
$2^{329} + k_{ORI}$	7,56	$2^{331} + 1$	**6,92**	**8,47**

Fonte: Elaborado pelo autor.

Num ensaio posterior foram realizadas 10 repetições da operação de exponenciação modular no ciclo de medições (visto na figura 5.2). O objetivo é demonstrar que há um ganho crescente à medida que a quantidade das operações vai aumentando, embora o número de repetições esteja muito abaixo de uma aplicação real, é possível estimar sua tendência de comportamento, pois, um número considerável de codificações e decodificações são executadas sequencialmente durante o ciclo de operações dos equipamentos (em sua maioria, ativos ininterruptamente). Por exemplo, um equipamento *desktop* que acessa um número considerável de páginas *web*, ao longo de um dia, terá seu desempenho ampliado, à medida que executa estas operações.

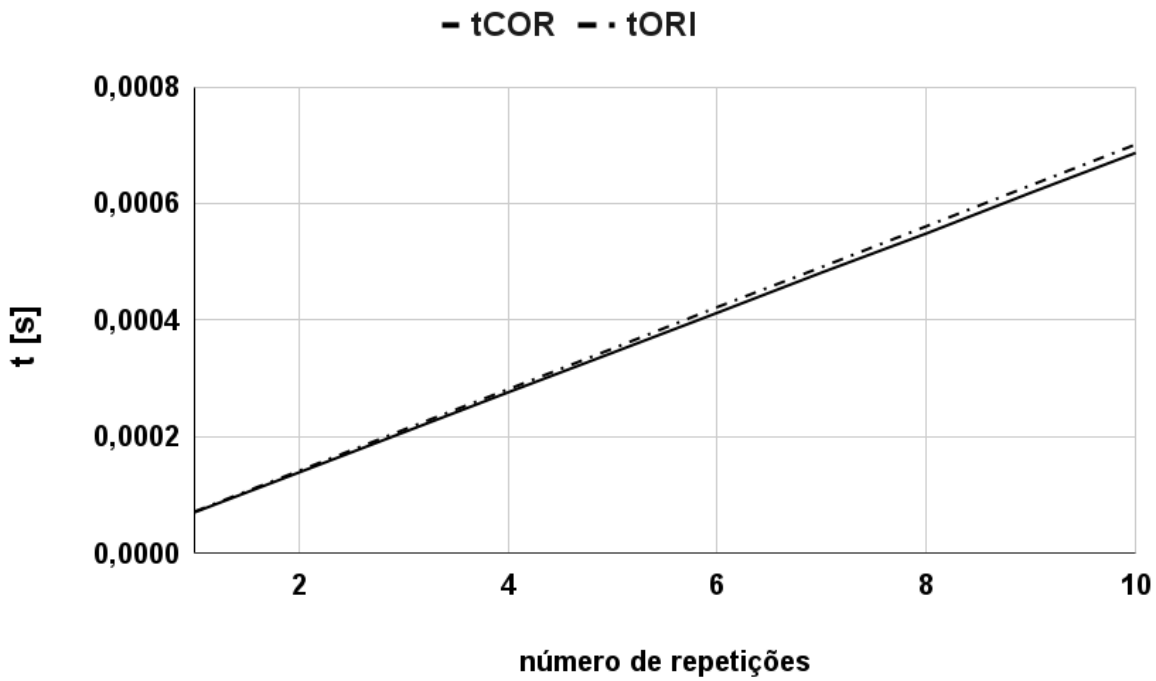
Pode-se observar no gráfico da figura 5.2 que há um distanciamento entre os tempos t_{COR} (execução do módulo auxiliar com a devida correção) e t_{ORI} (execução do módulo original). Indicando uma tendência de melhora de desempenho à medida que as operações são executadas sequencialmente.

A equação 5.8 indica a relação entre os expoentes usados para obter os módulos auxiliar (n_{AUX}) e original (n_{ORI}) e k_{ORI} compõe o módulo original e é constituído pela sequência de "1"s e "0"s intercalados, resultante deste fator de 3 vezes entre m_{AUX} e m_{ORI}.

$$2^{n_{AUX}} - 1 = 3.(2^{n_{ORI}} + k_{ORI}), \quad (5.8)$$

onde n_{AUX} é o exponente utilizado no módulo auxiliar, $n_{ORI} = n_{AUX} - 2$ o exponente usado pelo módulo original.

Figura 5.2 – Operação módulo simples, exemplos similares ao RSA-100, para 10 repetições.



Fonte: Elaborado pelo autor.

O próximo ensaio, demonstrado na tabela 5.2, focou na avaliação de algumas aproximações ao RSA (WIKIPEDIA, 2023) sem explorar o caráter repetitivo, visto anteriormente. Como resultado deste ensaio são demonstrados os respectivos ganhos de cada aproximação RSA para os módulos original e auxiliar.

Tabela 5.2 – Ensaios com exemplos similares ao RSA e os respectivos ganhos. Na Linguagem Python para $Y = 2047$.

RSA	$\mathbf{m_{ORI}}$	$\mathbf{t_{ORI}}[\mu s]$	$\mathbf{m_{AUX}}$	$\mathbf{t_{AUX}}[\mu s]$	**Ganho** [%]
100	$2^{329} + k_{ORI}$	7,56	$2^{331} + 1$	**6,92**	**8,47**
110	$2^{363} + k_{ORI}$	8,07	$2^{365} + 1$	**7,91**	**1,98**
120	$2^{397} + k_{ORI}$	8,77	$2^{399} + 1$	**8,63**	**1,60**
129	$2^{425} + k_{ORI}$	9,54	$2^{427} + 1$	**9,41**	**1,36**
1024	$2^{1023} + k_{ORI}$	22,49	$2^{1025} + 1$	**22,32**	**0,76**

Fonte: Elaborado pelo autor.

5.3.3 Exemplo de exponenciação modular usando módulos auxiliares decompostos (fatorados)

Com o objetivo de aumentar o desempenho do cálculo matemático envolvido nos processos de exponenciação modular, para módulos no formato $2^n - 1$ (com n par), a

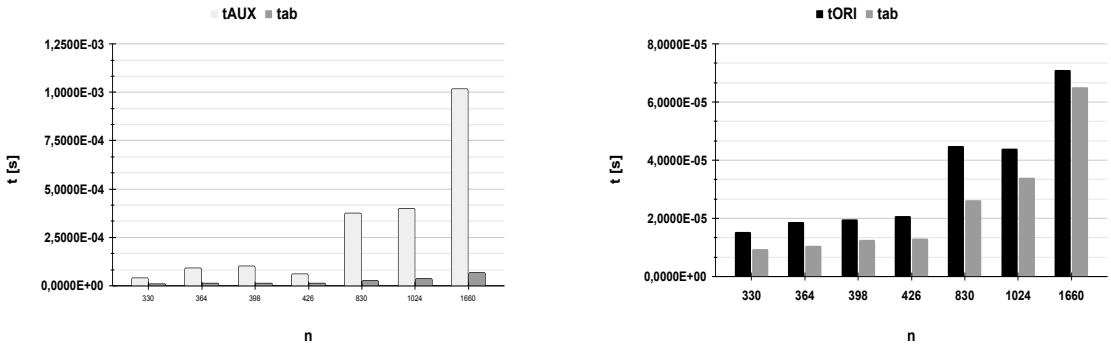
estratégia adotada nos próximos testes empregou o conceito matemático da fatoração definido como produto da soma pela diferença (mencionado em seções anteriores). O uso desta estratégia cria dois canais de operação modular (m_a e m_b) cujos resultados, posteriormente, devem ser convertidos para se obter um resíduo equivalente. A equação 5.9 demonstra a relação entre o módulo original e os novos canais formados pelo processo de fatoração.

$$\left|X^Y\right|_{m_{ORI}} = \left|\overbrace{V_a.R_a + V_b.R_b}^{|X^Y|_{m_{AUX}}}\right|_{m_{ORI}}, \tag{5.9}$$

onde $R_a = \left|X^Y\right|_{m_a}$ e $R_b = \left|X^Y\right|_{m_b}$ são respectivamente os resíduos das operações modulares com m_a e m_b, enquanto V_a e V_b são as constantes multiplicativas necessárias para o processo de operação modular final com m_{ORI}, produzindo, assim, o resultado equivalente à $\left|X^Y\right|_{m_{ORI}}$.

No ensaio demonstrado na figura 5.3 a operação executada foi $pow(X, Y, m)$ onde $Y = 2047$ e valores da tabela 5.2, para m_{AUX} e os respectivos canais modulares m_a e m_b, representados pelo tempo de processamento t_{ab}.

Figura 5.3 – Paralelismo para exponenciação modular em Python. Para t_{ab} sem as correções.



(a) (b)

Pode-se observar na figura 5.3 (a) que existem ganhos significativos, na faixa explorada, entre t_{AUX} e t_{ab} bem como na comparação entre t_{ORI} e t_{ab}, em (b). Este ensaio não considera o cálculo final da correção, mas serve como base referencial, demonstrando os potenciais benefícios da aplicação dos módulos fatorados.

Na sequência, deve-se determinar os valores das constantes multiplicativas V_a e V_b cujas equações estão demonstradas em 5.10 e 5.11. Posteriormente, estas constantes serão aplicadas no cálculo final com o ensaio que potencializa o uso dos módulos auxiliares no formato $2^n - 1$.

$$V_a = 2^{n-1} + 2^{(n-2)/2} \tag{5.10}$$

$$V_b = 2^{n-1} - 2^{(n-2)/2}, \tag{5.11}$$

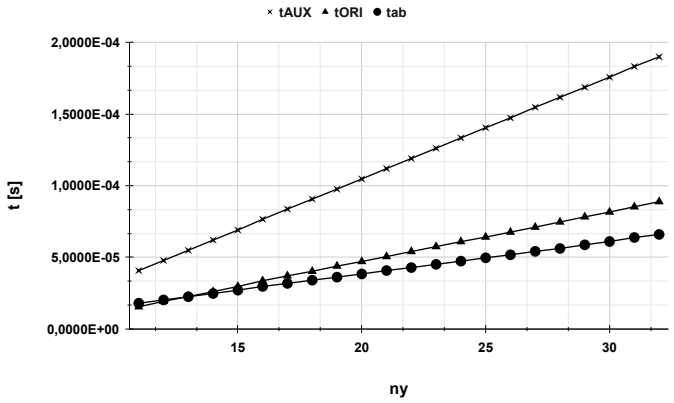
onde n é expoente de m_{AUX}, no formato $2^n - 1$, e sempre par.

No ensaio seguinte procura-se demonstrar a relevância do valor de Y na operação $|X^Y|_m$. Para isso escolheu-se o valor RSA-100 (330 *bits*), da tabela 5.2, e n_y como expoente que compõem o cálculo de Y, conforme descrito na equação 5.12.

$$Y = 2^{n_y} - 1, \tag{5.12}$$

onde n_y é o exponente usado para definir os valores de Y, obedecendo ao formato binário $111...111_2$. Os resultados podem ser observados na figura 5.4.

Figura 5.4 – Paralelismo para exponenciação modular em Python, variando Y para n com 330 bits e t_{ab} sem as correções.



Fonte: Elaborado pelo autor.

Na figura 5.4 pode-se notar que t_{ab} (ainda sem as correções) se torna menor que t_{ORI} quando $ny \geq 14$. Esse ponto é aqui denominado de **ponto de inflexão** representando o momento a partir do qual os valores de Y começam a apresentar relevância no cálculo de $|X^Y|m$ no uso dos expoentes fatorados.

Como sequência do trabalho, determinou-se os pontos de inflexão para alguns exemplos de n com magnitude similar à criptografia RSA e já aplica-se as devidas correções (multiplicativas V_a e V_b). Valores estes que são demonstrados na tabela 5.3, na segunda coluna e definidos como ny_i. Ainda na tabela 5.3, na terceira coluna, são mostrados os valores de ny utilizados com o objetivo de extrapolar e observar um ganho mais robusto, e cuja denominação adotada foi ny_e. Os resultados finais também estão demonstrados na tabela 5.4 e figura 5.5.

Analisando a tabela 5.4, pode-se verificar que houve um significativo aumento nos ganhos entre t_{ORI} e $t_{COR_{ab}}$, tendo este último apresentado valores menores de tempo e por consequência melhor desempenho. Cabe ressaltar que $t_{COR_{ab}}$ representa o caminho crítico

Tabela 5.3 – Expoentes (ny), utilizados para os pontos de inflexão e de extrapolação de ganho.

n	ny_i	ny_e
364	29	200
398	25	200
830	30	200
1024	70	200
1660	120	200

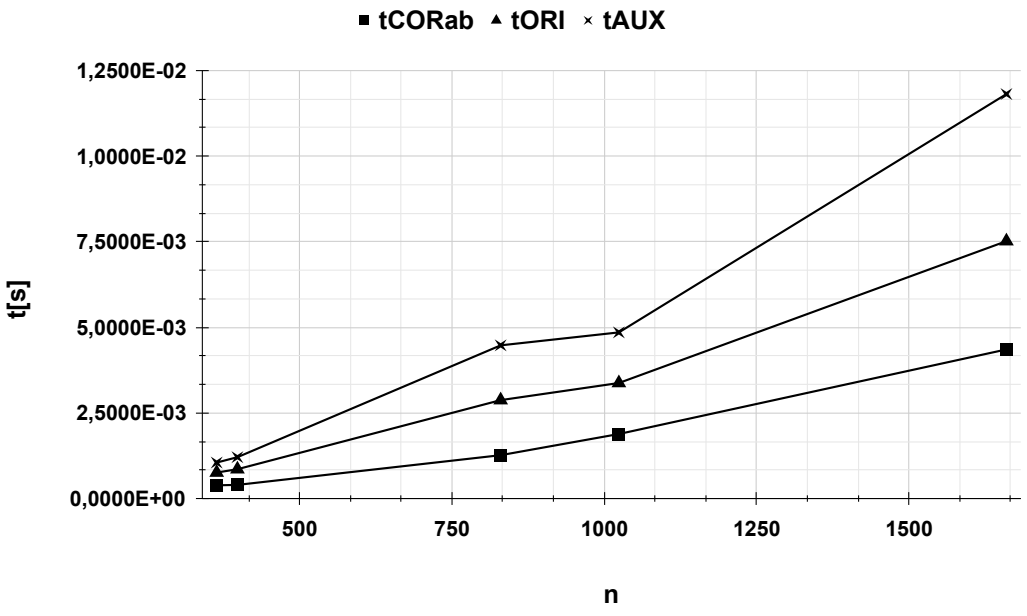
Fonte: Elaborado pelo autor.

Tabela 5.4 – Valores de t_{ORI} e $t_{COR_{ab}}$ para valores similares de RSA. Utilizando os valores de $ny = 200$ (ny_e).

n	$t_{ORI}[\mu s]$	$t_{AUX}[\mu s]$	$t_{COR_{ab}}[\mu s]$	Ganho [%]
364	760,16	1051,8	**387,58**	**49,01**
398	857,58	1208,9	**399,70**	**53,39**
830	2875,0	4474,7	**1267,4**	**55,92**
1024	3375,2	4853,0	**1885,8**	**44,13**
1660	7517,1	11812,0	**4355,5**	**42,06**

entre as operações corrigidas e executadas em paralelo, conforme o conceito já apresentado na seção 4.2.4.

Figura 5.5 – Medições dos tempos t_{ORI}, $t_{COR_{ab}}$ e t_{AUX} com os ns da tabela RSA, para $ny = 200$ (ny_e).



Fonte: Elaborado pelo autor.

No conjunto de resultados observados na figura 5.5 e tabela 5.4, destaca-se a importância da escolha do valor de Y para se obter um melhor desempenho possível. Em igual destaque, o valor de Y também possui relevância para a escolha da melhor chave criptográfica o que dificulta as tentativas de sua quebra, como visto na seção 5.2.

Fazendo um comparativo inicial entre os resultados obtidos nas operações modulares simples e de exponenciação modular pode-se verificar que nesta última há uma vantagem relativamente alta para a operações utilizando os expoentes fracionados. Destas observações pode-se inferir que operações/funções, como operação modular simples e exponenciação modular, são tratados de forma diferenciada dentro de uma linguagem de programação específica.

6 Discussões, trabalhos futuros e conclusões

Com base nas experimentações pode-se determinar a existência de pontos não cobertos pela abordagem dos módulos auxiliares, e neste sentido estudos futuros e mais aprofundados, se fazem necessários a fim de encontrar uma relação que pode beneficiar seu uso, promovendo uma abrangência maior de aplicações. Por outro lado, as famílias dos módulos auxiliares trabalhadas nos estudos revelam um potencial ganho nas melhorias dos tempos de execução. Nesta mesma esteira encontram-se os avanços promovidos pelos estudos em *hardware* que também produzem reduções nos tempos de execução além da área ocupada.

Na metodologia apresentada, é importante destacar que existem também outras formas alternativas para as medições dos tempos de execução que podem ser exploradas, a fim de avaliar seu desempenho em pesquisas futuras, para além das questões de aritmética modular. A exploração destas alternativas trariam maior robustez as diversas linhas de pesquisa reduzindo o engessamento metodológico.

Em virtude da crescente demanda nas atividades criptográficas, o estudo vem contribuir como uma alternativa de processamento que proporciona reduções na potência, sendo um dos gargalos atuais no âmbito computacional. Neste sentido o trabalho aponta, como proposta geral, ações que lançam mão das técnicas matemáticas a fim de tornar as operações mais facilitadas.

O trabalho abre uma série de possibilidades a serem exploradas que proporcionará, além de tudo, o desenvolvimento de ferramentas e/ou bibliotecas deixando todo o processamento transparente aos programadores e reduzindo assim, a curva de aprendizagem da metodologia. Pode-se inclusive iniciar o desenvolvimento de ferramentas a base de IA (Inteligência Artificial) que, devidamente treinadas, trarão benefícios nas avaliações dos caminhos críticos e auxiliando nas definições das melhores estratégias. Outros pontos a serem explorados são o uso das linguagens de baixo nível, mais diretamente ligadas ao *hardware* das arquiteturas, e o uso de linguagens compiladas.

Todas as ações propostas anteriormente são facilitas pela escalabilidade das soluções apresentadas por este trabalho, tanto em *hardware* como *software*, tornando os processos mais robustos para uma adaptação em operações de maior capacidade e que também não impede um aperfeiçoamento contínuo das ferramentas apresentadas.

Diante do conjunto de pesquisas e estudos realizados pode-se constatar que há uma vantagem na aplicação de módulos de baixo custo em operações aritméticas modulares. Tanto em *hardware*, com o uso nas operações modulares simples e exponenciação modular, como em *software* com as mesmas aplicações citadas anteriormente, pelo menos, nas

linguagens de programação utilizadas. Este último apresenta uma relativa pequena margem nos tempos de resposta para operações modulares simples e resultados mais promissores nas operações de exponenciação modular. Visto que cada comando/função são implementadas de formas diferentes na própria linguagem de programação empregada e entre as diversas linguagens existentes.

Nos aspectos relativos aos ensaios em *hardware* houve contribuições com a identificação e uso dos módulos de baixo custo, na reorganização das operações de conversão RNS-binário e nas operações de exponenciação modular. No uso dos módulos de baixo custo se destaca um estudo comparativo entre os formatos $2^n \pm 1$ com os formatos $2^n \pm k$ e que os primeiros apresentaram resultados com melhores desempenhos. Nas operações aritméticas em RNS, principalmente nas multiplicações por constantes, foram obtidos bons resultados na combinação das multiplicativas reversas com os valores de entrada, onde há destaque na redução de 20 para 9 constantes, que apresentavam índice de área 5 e *delay* com índice 2, indicando um melhor desempenho. Ainda na multiplicação por constante, observou-se reduções de 90% na área e de 40% no *delay*. Já em operações de exponenciação modular surgiram resultados que garantem uma redução no *delay* dos processamentos, e onde o destaque do ensaio são curvas com distanciamento gradativamente maiores entre as aplicações com módulo auxiliar comparado ao original, chegando a uma diferença de cerca de 50% em operações de 32 *bits*.

Referente aos estudos realizados em *software*, as contribuições se deram aplicando os módulos de baixo custo nas operações modulares simples, nas operações de exponenciação modular e o emprego das mesmas em criptografia. Todas as análises utilizando as linguagens de programação *Python*, *Java* e *Go*. Nas operações modulares simples destacam-se os bons resultados obtidos diretamente com os módulos no formato $2^n + 1$ e posteriormente com os de formato $2^n - 1$ aplicando-se, neste último, variações nas configurações das entradas bem como a possibilidade de manipulação algébrica com uso de produtos notáveis (produto da soma pela diferença). Detalhando as ações realizadas com os módulos no formato $2^n - 1$ apresentou-se a possibilidade de manipular a entrada X de forma que pudesse apresentar um menor tempo de processamento, principalmente com uso de deslocamentos binários (multiplicação por 2) que pudessem ser reconvertidos de forma também mais simples. Ainda nos módulos de formato $2^n - 1$ o emprego do desmembramento no expoente original (para n par) de forma a constituir dois outros módulos $(2^{n/2} \pm 1)$ e sua posterior recuperação com o uso dos resíduos das operações modulares em conjunto de constantes multiplicativas (obtidas de forma algébrica). Nas operações de exponenciação modular, novamente os módulos no formato $2^n + 1$ apresentaram resultados promissores mais diretamente e os de formato $2^n - 1$ necessitaram de manipulações algébricas para se obter um melhor desempenho. Ainda dentro deste estudo destaca-se também a exploração do expoente Y na operação $|X^Y|_m$ onde foi possível determinar que valores maiores do expoente produziam resultados melhores. Nas operações de criptografia, os estudos se

concentraram na técnica RSA com a aplicação dos módulos no formato $2^n \pm 1$ e testes com o expoente Y, na operação $C = |M^Y|_m$. Onde foi possível obter resultados promissores nos módulos de baixo custo e na exploração do exponente. Em termos quantitativos foi possível o alcance de ganhos acima de 50%, por exemplo, foi observado nos casos onde o valor de n se assemelhava aos de RSA-120 e RSA-250, com 398 e 830 *bits* respectivamente.

Tanto em *hardware* como em *software* as iniciativas trouxeram resultados promissores, de modo geral, na redução do *delay* das operações, servindo como contribuição principal deste trabalho. Além de apontar uma séria de outras sugestões para futuras abordagens, que podem ser exploradas a luz do que foi elaborado até aqui e considerando possíveis novas descobertas, do meio científico acadêmico bem como da sociedade de modo geral.

Referências

BHARDWAJ, M.; SRIKANTHAN, T.; CLARKE, C. T. A reverse converter for the 4-moduli superset {2/sup n/-1, 2/sup n/, 2/sup n/+ 1, 2/sup n+ 1/+ 1}. In: IEEE. *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*. [S.l.], 1999. p. 168–175.

BI, G.; JONES, E. Fast conversion between binary and residue numbers. *Electronics Letters*, IET, v. 24, n. 19, p. 1195–1197, 1988.

CAO, B.; CHANG, C.-H.; SRIKANTHAN, T. A residue-to-binary converter for a new five-moduli set. *IEEE Transactions on Circuits and Systems I: Regular Papers*, IEEE, v. 54, n. 5, p. 1041–1049, 2007.

CHANG, C.-H.; MOLAHOSSEINI, A. S.; ZARANDI, A. A. E.; TAY, T. F. Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. *IEEE circuits and systems magazine*, IEEE, v. 15, n. 4, p. 26–44, 2015.

CLAUDIO, E. D. D.; ORLANDI, G.; PIAZZA, F. Fast rns dsp algorithms implemented with binary arithmetic. In: IEEE. *International Conference on Acoustics, Speech, and Signal Processing*. [S.l.], 1990. p. 1531:1534.

CONWAY, R. Reducing complexity of fixed-coefficient fir filters. Electronics Letters, p. 1185–1186, 2006.

DADDA, L. Some schemes for parallel multipliers. *Alta frequenza*, v. 34, p. 349–356, 1965.

DENNY, T.; DODSON, B.; LENSTRA, A. K.; MANASSE, M. S. On the factorization of rsa-120. In: SPRINGER. *Advances in Cryptology—CRYPTO'93: 13th Annual International Cryptology Conference Santa Barbara, California, USA August 22–26, 1993 Proceedings 13*. [S.l.], 1994. p. 166–174.

devmedia. *Top 10 linguagens de programação mais usadas no mercado*. 2019. Disponível em: <https://www.devmedia.com.br/top-10-linguagens-de-programacao-mais-usadas-no-mercado/39635>. Acessado em: 15 setembro 2023.

GOOGLE, I. *Documentation*. 2024. Disponível em: <https://go.dev/doc/>.

GUEDES, T. A.; MARTINS, A. B. T.; ACORSI, C. R. L.; JANEIRO, V. Estatística descritiva. *Projeto de ensino aprender fazendo estatística*, Universidade Estadual de Maringá Maringá, p. 1–49, 2005.

INC., V. S. T. *UMC high density standards cells library 0.13 μm CMOS process v2.3*. [S.l.], 2010.

INC., W. R. *Mathematica, Version 14.0*. 2024. Disponível em: <https://www.wolfram.com/mathematica>.

MA, D. *A fast way to do modular exponentiation*. 2015. Disponível em: <https://allmathconsidered.wordpress.com/2015/07/10/a-fast-way-to-do-modular-exponentiation/>. Acesso em: 05 maio 2023.

MATUTINO, P. M.; PETTENGHI, H.; CHAVES, R.; SOUSA, L. Multiplierbased binary-to-rns converter modulo {2n±k}. In: *Proc. 26th Conf. DCIS*. [S.l.: s.n.], 2011. p. 125–130.

MICROSYSTEMS, I. S. *OpenJDK Developers' Guide*. 2024. Disponível em: <https://openjdk.org/guide/>.

MOHAN, P. A. *Residue number systems: algorithms and architectures*. [S.l.]: Springer Science & Business Media, 2002. 46 p.

MOHAN, P. A.; MOHAN, P. A. *Residue Number Systems*. [S.l.]: Springer, 2016. 27:35 p.

OMONDI, A. R.; PREMKUMAR, A. B. *Residue number systems: theory and implementation*. [S.l.]: Imperial College Press, 2007. v. 2. 219 p.

OVERFLOW, S. *Stack Overflow*. 2008.

PANTAZI-MYTARELLI, I. The history and use of pipelining computer architecture: Mips pipelining implementation. In: IEEE. *2013 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. [S.l.], 2013. p. 1:7.

PARHAMI, B. *Computer arithmetic*. [S.l.]: OXFORD UNIVERSITY PRESS, 2010. 72:73 p.

PATRONIK, P.; PIESTRAK, S. J. Design of residue generators with cla/compressor trees and multi-bit eac. In: IEEE. *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*. [S.l.], 2017. p. 1–4.

PATTERSON, D. A.; HENNESSY, J. L.; ASHENDEN, P. J. *Computer Organization and Design: The Hardware*. [S.l.]: Elsevier Science Limited, 2014. 272;276 p.

PIESTRAK, S. J. A high-speed realization of a residue to binary number system converter. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, IEEE, v. 42, n. 10, p. 661–663, 1995.

PREDIGER, V.; BAIRROS, F.; SEMAN, L. O.; BEZERRA, E. A.; PETTENGHI, H. Rns processor using moduli sets of the form 2 n±1 2^ n ± 1. *International Journal of Circuit Theory and Applications*, Wiley Online Library, 2023.

ROSSUM, G. V.; DRAKE, F. L. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

SEO, S. *A review and comparison of methods for detecting outliers in univariate data sets*. Tese (Doutorado) — University of Pittsburgh, 2006.

STALLINGS, W. *Cryptography and network security, 5/E*. [S.l.]: Pearson Education, 2010. 280 p.

STALLINGS, W. *Computer security: principles and practice*. [S.l.]: Pearson, 2014.

SZABO, N. S.; TANAKA, R. I. *Residue arithmetic and its applications to computer technology*. [S.l.]: New York: McGraw-Hill, 1967.

TSAI, S.-H. T.; ZHOU, C.-h. Taiwan's united microelectronics corporation (umc). *The Silicon Dragon, High Tech Industry in Taiwan, Northampton MA: Edward Elgar*, p. 95–112, 2006.

WALLACE, C. S. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, IEEE, n. 1, p. 14–17, 1964.

WANG, Y. Residue-to-binary converters based on new chinese remainder theorems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, IEEE, v. 47, n. 3, p. 197–205, 2000.

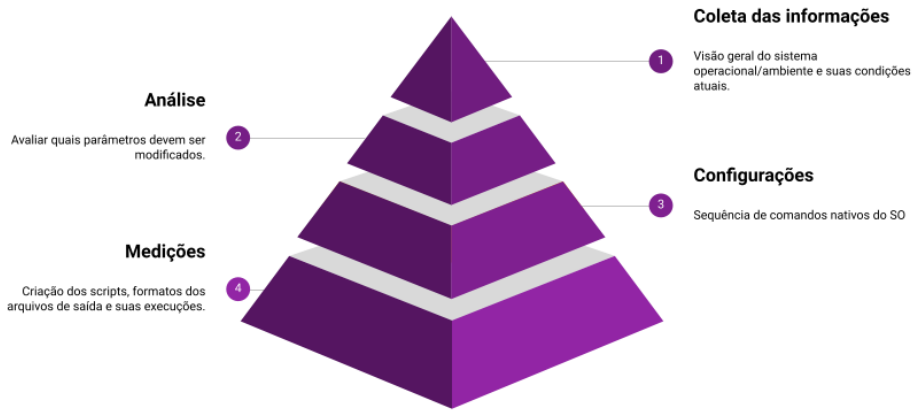
WESTE, N. H.; HARRIS, D. *CMOS VLSI design: a circuits and systems perspective*. [S.l.]: Pearson Education, 2011. 462 p.

WIKIPEDIA. *RSA numbers*. 2023. Disponível em: https://en.wikipedia.org/wiki/RSA numbers.

APÊNDICE A – Medição dos tempos

A sequência de passos para a realização das medições e comparações, incluindo as próprias, ocorreram segundo o diagrama mostrado a seguir. O formato de prisma piramidal, da figura A.1, indica o grau de prioridade de cada etapa.

Figura A.1 – Diagrama organização das tarefas



Fonte: Elaborado pelo autor.

A.1 Coleta das informações

Etapa inicial no processo das medições, consiste no levantamento dos dados dos sistema operacional. Os dados mínimos necessários, independente do sistema operacional adotado, estão elencados abaixo:

- Dados a cerca de como o SO (Sistema Operacional) trata operações que rodam em segundo plano;
- Informações da forma de tratamento e controle de operações agendadas;
- Tratamento pelo SO na gestão de usuários;
- Possibilidade de instalação e disponibilidade de SOs em versões mínimas;
- Existência de scipts para automação de processos.

A.2 Análise

Etapa na qual foram avaliados quais as configurações que devem ser realizadas para as medições dos tempo de maneira mais precisa possível, minimizando ao máximo possível os ruídos produzidos pelo sistema. São configurações que basicamente levam em conta a gestão interna de cada SO no que se refere a processos sendo executados em segundo plano, processos agendados e controle de usuários.

A.3 Configurações

Uma vez definidos quais os parâmetros levados em consideração a etapa seguinte trata de efetuar as configurações, conforme lista abaixo:

- Desligar as conexões com redes de dados externas, como Ethernet. Reduzindo interrupções de hardware que podem mascarar as medições de tempo;
- Desligar as conexões com dispositivos de interface com usuários (exceto o teclado), que igualmente podem gerar interrupções;
- Garantir que apenas um usuário esteja conectado e usando os recursos do terminal/CPU;
- Garantir para que não estejam sendo executadas atividades de segundo plano;
- Garantir para que não sejam programadas atividades agendadas.

A.4 Medições

Realização dos ensaios propriamente ditos. Utilizando protocolos próprios para a nomenclatura dos arquivos produzidos. Em seguida realizando o tratamento estatístico necessário. Para a coleta de dados e registros é necessário obter as seguintes informações a cerca as linguagens que foram utilizadas:

- Elencar os comandos e sintaxes necessários para o uso da linguagem Python;
- Elencar os comandos e sintaxes necessários para o uso da linguagem Java;
- Elencar os comandos e sintaxes necessários para o uso da linguagem Go;

APÊNDICE B – Tratamento estatístico

Conforme mencionado em seções anteriores, as medições de tempo também incluem, nas tomadas inciais de cada série, os efeitos da carga das instruções no pipeline (PATTERSON; HENNESSY; ASHENDEN, 2014). Na tabela abaixo percebe-se que, principalmente na primeira aquisição (indicado em destaque na tabela), há uma diferença de cerca de 3 vezes a maior que as demais leituras. Significando que este tempo adicional, a cada início de cálculo, deve-se a carga das instruções dentro do pipeline (PANTAZI-MYTARELLI, 2013).

Tabela B.1 – Exemplo do formato de coleta dos dados e o efeito de carga do pipeline

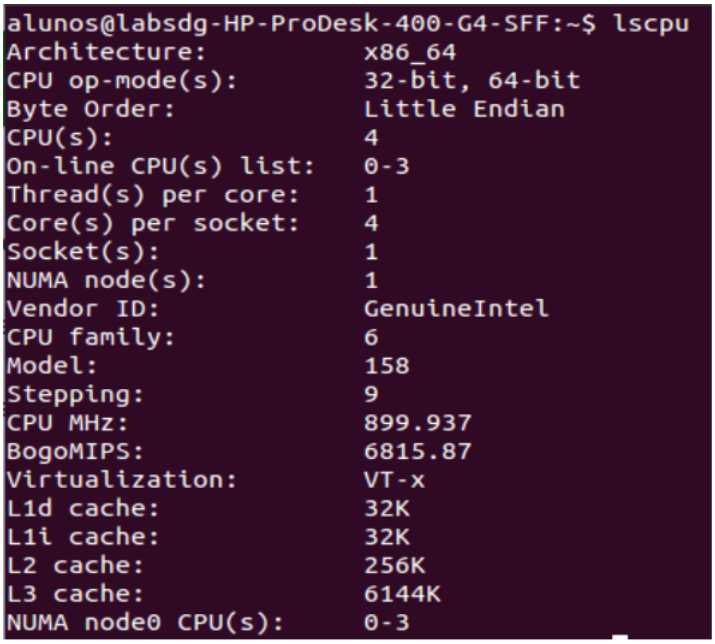
Aquisição	t1 [s]
1	$\mathbf{3{,}325 \cdot 10^{-6}}$
2	$1{,}256 \cdot 10^{-6}$
3	$1{,}062 \cdot 10^{-6}$
4	$1{,}020 \cdot 10^{-6}$
5	$1{,}015 \cdot 10^{-6}$
6	$1{,}013 \cdot 10^{-6}$
7	$1{,}009 \cdot 10^{-6}$
8	$1{,}019 \cdot 10^{-6}$
9	$1{,}008 \cdot 10^{-6}$
10	$1{,}007 \cdot 10^{-6}$
11	$1{,}007 \cdot 10^{-6}$
12	$1{,}009 \cdot 10^{-6}$
13	$1{,}011 \cdot 10^{-6}$
14	$1{,}013 \cdot 10^{-6}$
15	$1{,}013 \cdot 10^{-6}$
16	$1{,}010 \cdot 10^{-6}$
17	$1{,}011 \cdot 10^{-6}$
18	$1{,}014 \cdot 10^{-6}$
19	$1{,}008 \cdot 10^{-6}$
20	$1{,}006 \cdot 10^{-6}$

Para os efeitos das análises do comportamento, em regime permanente, avalia-se como não relevante considerar este tempo inicial na média das leituras obtidas. Neste caso a técnica usada para desconsiderar estas aquisições e eventuais discrepâncias foi defini-las como outliers. O método usado para desconsiderar os outliers foi o de Tukey (SEO, 2006). Após a redefinição do novo conjunto de valores, efetuou-se o cálculo do coeficiente de variação quando obeteve-se o valor de 5%, o que representa uma baixa dispersão dos dados (dados mais homogêneos). Indicando que a média pode ser empregada como medida representativa do conjunto de amostras (GUEDES et al., 2005).

APÊNDICE C – Sistema operacional e hardware

Através de uma série de comandos, fez-se um levantamento das versões de firmware, SO (Ubuntu versão **14.04.6 LTS**), configuração do hardware utilizado. Para a identificação do hardware utiliza-se o comando lscpu e lshw. O resultado, como visto na figura C.1, é um panorama geral da arquitetura de um dos hardwares utilizados nos experimentos.

Figura C.1 – Configuração genérica da CPU na plataforma desktop.



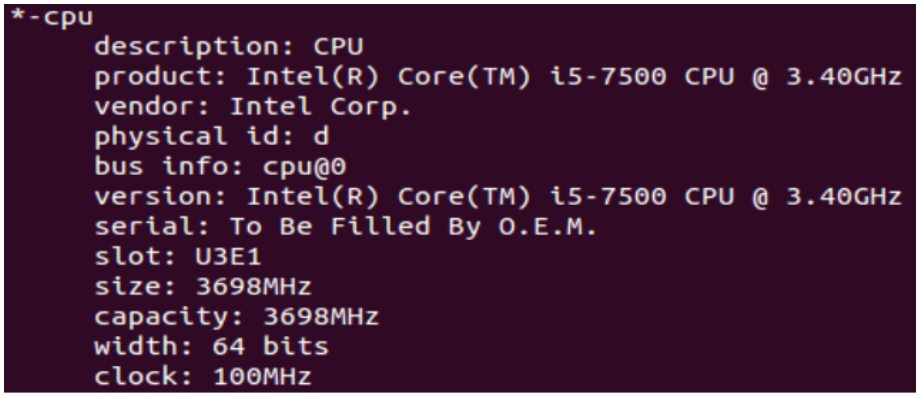
Fonte: Elaborado pelo autor.

No caso da figura C.1, a arquitetura está na sua formação "original", sem as intervenções necessárias para as medições. Na figura C.2, através do comando lshw, tem-se detalhes mais específicos do processador.

Outro ponto analisado se relaciona com o número de usuários conectados na estação de trabalho (equipamento desktop) de onde foram realizadas as medições. Quanto maior o número de usuários, maior será o compartilhamento de recursos de máquina entre eles, o que diminui a performance de execução de um determinado algoritmo. No sistema operacional Linux esta avaliação é feita com o comando who. Na imagem C.3 verifica-se uma imagem do resultado do comando.

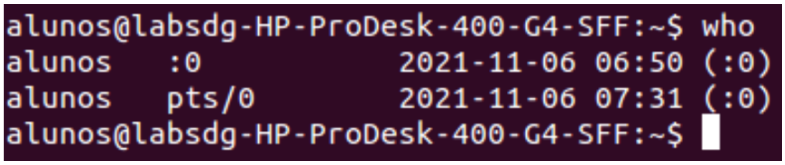
Na imagem C.3 pode-se verificar que há apenas um usuário conectado, no caso alunos.

Figura C.2 – Relatório da configuração detalhada da CPU na plataforma desktop.



Fonte: Elaborado pelo autor.

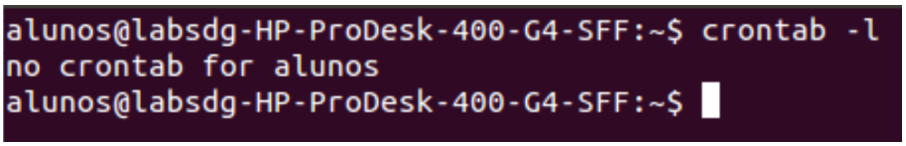
Figura C.3 – Relatório de usuários logados no SO Linux na plataforma desktop.



Fonte: Elaborado pelo autor.

Após verificou-se a existência das tarefas que possivelmente possam estar agendadas (cron jobs). No Linux, uma das maneiras de verificar é através do comando crontab. Na imagem C.4 tem-se o resultado da ação.

Figura C.4 – Tarefas paralelas no ambiente Ubuntu-Linux.

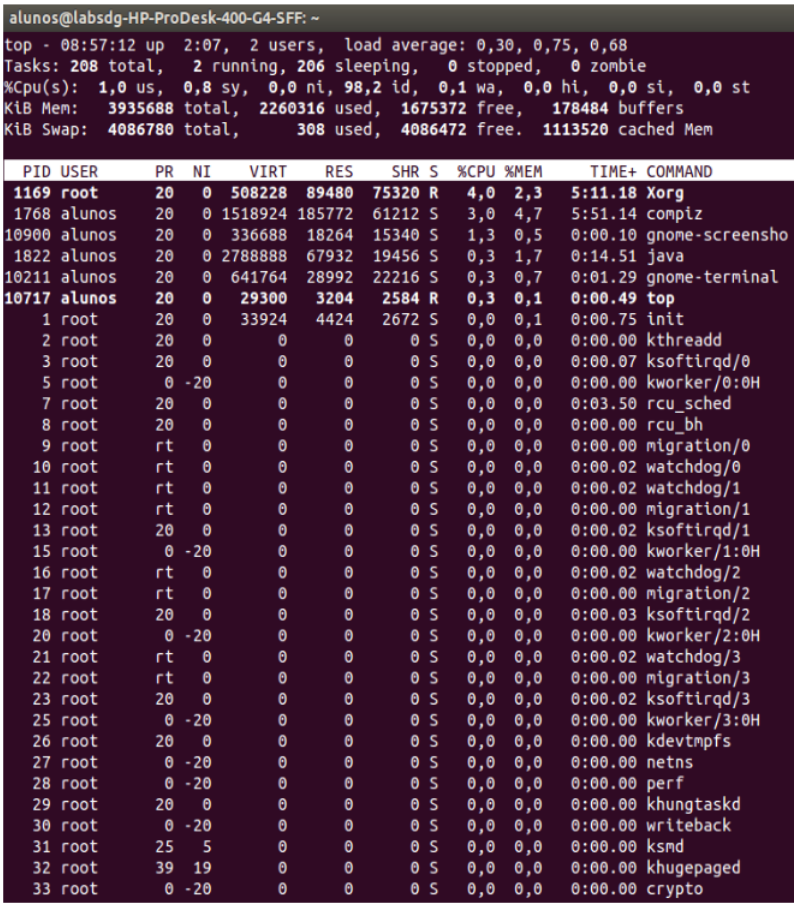


Fonte: Elaborado pelo autor.

No caso específico não há ações agendadas que possam comprometer a execução das medições. Em caso positivo, faz-se necessário que elas sejam suspensas ou extintas antes da coleta dos tempos.

Posteriormente, e na mesma linha da verificação de processos, está o comando top que lista todas as aplicações (deamons) que rodam em paralelo.

Figura C.5 – Gerenciamento de deamons no ambiente Ubuntu-Linux.



Fonte: Elaborado pelo autor.

APÊNDICE D – Uso do Shell Script

Para automatizar os processos de coleta de dados e garantir que os programas sejam chamados de forma externa, com passagem de parâmetros, fez-se necessário o uso da linguagem da família Shell Script, nativa do sistema operacional utilizado. Os códigos a seguir fazem as chamadas dos programas utilizados passando os parâmetros necessários e indicando em que local serão registrados os resultados.

D.1 Exemplo de chamada do programa de medição na linguagem Python

```
#!/bin/bash
for indice in {102..201..2}; do
  python3 me01.py $indice |&tee -a arquivo.txt
done
```

D.2 Exemplo de chamada do programa de medição na linguagem Java

```
#!/bin/bash
for indice in {102..201..2}
do
    java mb01 $indice

done | tee -a arquivo.txt
```

D.3 Exemplo de chamada do programa de medição na linguagem Go

```
#!/bin/bash
for indice in {102..201..2}
do
  ./gmb02 $indice
```

```
6 done | tee -a arquivo.txt
```

APÊNDICE E – Coletas de dados em Python

Os ensaios para medição dos tempo utilizou linguagem Python na versão **3.4.3**. Rotinas, funções e chamadas básicas.

Função da medição de tempo na linguagem Python

Medição do tempo da operação e exponenciação modular

```
#Definicao da funcao de coleta do tempo da operacao
def ft_x2(X, Y, m):
  import time
  i = time.perf_counter()
  A = pow(X, Y, m)
  f = time.perf_counter()
  return (f - i)
```

Chamada da função de medição de tempo

```
from ftx2 import ft_x2
import sys
X = int(sys.argv[1]) #recebe o valor de X
Y = int(sys.argv[2]) #recebe o valor de Y
m = int(sys.argv[2]) #recebe o valor de m

  for i in range(0, 20):
    result = ft_x2(X, Y, m)
    print(str(result).replace(".", ","))
```

APÊNDICE F – Coletas de dados em Java

Os ensaios para medição dos tempo utilizou linguagem Java na versão **1.7.0_201** Open JDK Runtime Environment.

```
public class ModExp02
{
  static int potencia(int a, int b, int m)
  {
    int r = 1;
    a = a % m;
    if(a == 0)
      return 0;
    while(b > 0)
    {
      if((b  & 1) !=0)
        r = (r * a) % m;
      b = b >> 1;
      a = (a  * a) % m;
    }
    return r;
  }
  public static void main(String[] args){
    int valor_a = Integer.valueOf(args[0]);
    int valor_b = Integer.valueOf(args[1]);
    int valor_m = Integer.valueOf(args[2]);
    int result;
    long ti;
    long tt;
    ti = System.nanoTime();

    result = potencia(valor_a, valor_b, valor_m);

    tt = System.nanoTime() - ti;

    System.out.println(tt);
  }
}
```

APÊNDICE G – Coletas de dados em Go

Ensaios para medição dos tempos da operação modular simples com a linguagem Go na versão **Go1.18 Linux/386**.

```
package main
import (
  "fmt"
  "math/big"
  "time"
  "strconv"
  "os"
)
func main() {
  n, errm := strconv.ParseInt(os.Args[1], 10, 64)
  var bn = big.NewInt(n)
  lk := make([]*big.Int, 0)
  lk = append(lk, big.NewInt(-5), big.NewInt(-3), big.NewInt(-1), big.
      NewInt(0),       big.NewInt(1), big.NewInt(3), big.NewInt(5))
  var X = big.NewInt(12)
  var t time.Time
  var t2 time.Time
  var binario string
  var xbin string = ""
  var result = big.NewInt(1)
  var m = big.NewInt(0)
  var m2 = big.NewInt(0)
  var divisor = big.NewInt(3)
  var e = big.NewInt(2)
  var bit bool
  fmt.Println("GO | ", m, errm)
  for k:=4; k <= 4; k++{
    m.Exp(e, bn, nil)
    m.Add(m, lk[k])
    m2.Div(m, divisor)
    binario = fmt.Sprintf("%b", m)
    bit = true
    xbin = ""
    for b:=0; b < (len(binario) * 2); b++{
      if bit == true {
        xbin = xbin + "1"
        bit = false
      } else {
        xbin = xbin + "0"
        bit = true
```

```
        }
    }
    X, ok := X.SetString(xbin, 2)
    fmt.Println("n= ", n, " |k=", lk[k], ok)
    for c:=0; c < 20; c++{
        t = time.Now()
        result = result.Mod((result.Mod(X, m)), m2)     //mCOR
        t2 = time.Now()
        fmt.Println(t2.Nanosecond() - t.Nanosecond())
    }
  }
}
```