FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" – UNIVEM
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

MARÇAL LUIZ BISSOLI

IMPACTO DA MULTIPLICAÇÃO E EXPONENCIAÇÃO MODULAR EM *HARDWARE* NO ALGORITMO RSA

MARÍLIA
2007

MARÇAL LUIZ BISSOLI

IMPACTO DA MULTIPLICAÇÃO E EXPONENCIAÇÃO MODULAR EM *HARDWARE* NO ALGORITMO RSA

Dissertação apresentada ao Centro Universitário Eurípides de Marília – UNIVEM, mantido pela Fundação de Ensino Eurípides da Rocha, para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas).

Orientador:
Prof. Dr. Edward David Moreno Ordonez

MARÍLIA
2007

BISSOLI, Marçal Luiz.

Impacto da Multiplicação e Exponenciação Modular em Hardware no Algoritmo RSA / Marçal Luiz Bissoli; orientador: Prof. Dr. Edward David Moreno Ordonez. Marília, SP: [s.n.], 2007.
105 f.

Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha.

CDD: 004.2

MARÇAL LUIZ BISSOLI

IMPACTO DA MULTIPLICAÇÃO E EXPONENCIAÇÃO MODULAR EM *HARDWARE* NO ALGORITMO RSA

Banca examinadora da dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília – UNIVEM, para obtenção do Título de Mestre em Ciência da Computação (Área de Concentração: Arquitetura de Sistemas).

Resultado: _____

ORIENTADOR: Prof. Dr. Edward David Moreno Ordonez

1º EXAMINADOR: Profa. Dra. Kalinka Regina L. J. Castelo Branco (UNIVEM)

2º EXAMINADOR: Prof. Dr. Humberto Feresoli (UNESP – Bauru)

Marília, _____ de _____ de 2007.

AGRADECIMENTOS

A Deus.

À minha esposa pelo apoio e paciência.

Ao meu orientador professor Edward David Moreno Ordonez pela sua imensa competência e dedicação.

Ao professor Fábio Dascêncio Pereira, fundamental e onipresente nos meus momentos de luta com o VHDL.

Ao professor Rodolfo Barros Chiaramonte e ao Lúcio Felippe de Mello Neto especialmente pela ajuda na minha lida com a linguagem C e ao professor Elvis Fusco pelo imenso apoio no meu começo nessa dura empreitada que é o mestrado.

BISSOLI, Marçal Luiz. **Impacto da Multiplicação e Exponenciação Modular em *Hardware* no Algoritmo RSA**. 2007. 105 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMO

Com o uso intensivo da Internet para operações de transmissão de dados, em especial aplicações bancárias e comerciais, percebe-se a necessidade da segurança nas comunicações de dados sigilosos. Considerando que tanto as invasões quanto as comunicações usando sistemas computacionais estão sendo cada vez mais intensas, somadas ao intermitente avanço do poder computacional, torna o incremento de medidas de segurança um objetivo a ser perseguido continuamente. Os meios de segurança atuais empregam primordialmente a criptografia. Dentre os algoritmos criptográficos existentes atualmente, os mais seguros e os mais utilizados são os algoritmos de chave pública, sendo o RSA um dos mais conhecidos e utilizados. O RSA utiliza enormes números primos (1200 *bits* ou mais), sendo o controle e o processamento desses dados uma tarefa muito lenta, em especial, para chaves muito grandes. Assim, para acelerar o processo de criptografia necessita-se de algoritmos computacionais específicos para esse fim. Com essa finalidade, esta dissertação apresenta o estudo, implementação e análise do impacto que o algoritmo Multiplicativo de Montgomery, associado ao algoritmo multiplicativo de Gutub, tem no algoritmo RSA. Foram realizadas implementações em linguagens C e VHDL e também foram feitos protótipos em *hardware* usando FPGAs. Os testes em FPGAs permitiram obter dados para chaves de até 64 *bits*, uma vez que a ocupação das placas (das FPGAs) foi muito maior do que se previa ao se iniciar este trabalho, porém, a velocidade de execução é bem mais rápida do que a respectiva implementação em VHDL sem o uso do algoritmo de Montgomery.

Palavras-Chave: Algoritmos Criptográficos, Algoritmo RSA, Algoritmo de Montgomery, Algoritmo de Gutub, *Hardware* e FPGAs.

BISSOLI, Marçal Luiz. **Impacto da Multiplicação e Exponenciação Modular em *Hardware* no Algoritmo RSA**. 2007. 105 f. Dissertação (Mestrado em Ciência da Computação) - Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

With the Internet intensive use for data-communication operations, especially banking and commercial applications, one can perceive the security need in data-communications of data that demand secrecy. The communication invasions in using computational systems being more and more intense, added by the intermittent computational power's advance, makes the increment of security measures an objective to be pursued The current security ways use primordially cryptography. Amongst the currently existing cryptographic algorithms, the safest and most used are the public keys algorithms, being de RSA one of the most known and used. The RSA uses enormous prime numbers (1200 *bits* or more), being the the control and processing a very slow work, in special, for very great keys. Thus, to speed up the cryptography process it is needed specific computational algorithms. With this objective, this thesis, presents the study, implementation and analysis of the impact that the Montgomery's multiplicative Algorithm associated with the Gutub's multiplicative algorithm have in RSA algorithm. In This work, we have implemented those algorithms in C and VHDL languages and also prototypes of then were made in Hardware with FPGAs. The experiences with FPGAs allow us to get data for keys until 64 *bits*, since the platforms' occupation was greater then one could suppose, but, the execution's velocity is too fast that the respective implementation without the Montgomery algorithm's use.

Keywords: Cryptographyc Algorithm, Algoritmo RSA, Algoritmo de Montgomery, Algoritmo de Gutub, *Hardware* e FPGAs.

LISTA DE ILUSTRAÇÕES

Figura 1 - Segurança no envio de mensagens cifradas. 20
Figura 2 – Exemplo 1 de utilização do Algoritmo de Euclides 25
Figura 3 – Exemplo 2 do Algoritmo de Euclides 27
Figura 4 – Algoritmo de Gutub 34
Figura 5 – Multiplicação Modular utilizando o Algoritmo De Gutub (GUTUB,2000) 35
Figura 6 – Multiplicação Modular utilizando Montgomery. 38
Figura 7 – Cálculo do MDC(32,27) com o Algoritmo de Euclides. 38
Figura 8 – Algoritmo de Multiplicação Modular no Domínio de Montgomery 40
Figura 9 – Algoritmo de Koç para exponenciação modular 43
Figura 10 – Cálculo de 23^5 com o Método Exponencial de Koç 44
Figura 11 – Algoritmo de Multiplicação 1 de Daly. 45
Figura 12 – Algoritmo de Multiplicação 2 de Daly. 47
Figura 13 – Algoritmo de Multiplicação 3 de Daly. 49
Figura 14 – Algoritmo de Exponenciação 1 de Daly. 51
Figura 15 - Algoritmo de Exponenciação 2 de Daly 53
Figura 16 – Função em Linguagem C na Implementação de Criptografia com o RSA 57
Figura 17 – Programa de Cálculo do número R^{-1} 58
Figura 18 – Função ME(a,e,m) 59
Figura 19 – Primeira Implementação de Criptografia com RSA utilizando Montgomery 60
Figura 20 – Comparativo de Velocidade de Criptografia. 61
Figura 21 – Função de Criptografia do RSA - Montgomery/Gutub 62
Figura 22 – Comparativo de Velocidades de criptografia. 63
Figura 23 – Operações realizadas em cada operação com o algoritmo de Brakley 64
Figura 24 – Comparação de Tempos de Criptografia para Chaves de mesmo tamanho com Módulos de Tamanhos Diferente. 65
Figura 25 – Algoritmo Radix 2 de Montgomery (AMANOR, 2005) 68
Figura 26 – Trecho de Programa em VHDL que efetua a Multiplicação de Montgomery 69
Figura 27 – Método binário de Quadratura e Multiplicação. 70
Figura 28 - Máquina de Estados de Gutub/ Quadratura e Multiplicação. 71

Figura 29 - Arquitetura da Implementação do algoritmo Gutub em VHDL. 73
Figura 30 - Máquina de Estados – programa com o algoritmo Rápido de Montgomery..... 75
Figura 31 - Arquitetura da Implementação do algoritmo Fast Montgomery em VHDL..... 76
Figura 32 - Simulação de criptografia para chaves de 16 bits..... 80
Figura 33 – Comparação de desempenho em hardware e software..... 81
Figura 34 - Esquema do Gerador de Números Aleatórios. 90
Figura 35 - Algoritmo de Euclides Estendido..... 91

LISTA DE TABELAS

Tabela 1. Cadastro de Chaves Criptográficas. *19*
Tabela 2. Valores dos Caracteres para Exemplo do RSA. *32*
Tabela 3. Comparativo de desempenho entre Criptografia realizada utilizando os algoritmos de Montgomery, de Montgomery/Gutub e Brakley *62*
Tabela 4. Tempos de Criptografia – Algoritmo de Gutub. *79*
Tabela 5. Tempos de Criptografia – Algoritmo de Montgomery. *79*
Tabela 6. Comparativo de tempos de criptografia *82*

LISTA DE SIGLAS

VHDL - *Very High Speed Integrated Circuit Hardware Description Language*

ASICs - *Application Specific Integrated Circuit*

VLIW - *Very Long Instruction Word*

SUMÁRIO

CAPÍTULO 1. INTRODUÇÃO 15

Justificativa 21

Objetivo 21

Organização da Dissertação 22

CAPÍTULO 2. O ALGORITMO ASSIMÉTRICO RSA 24

2.1 Algoritmo Estendido de Euclides 24

2.2 Introdução ao RSA 28

2.3 Exemplos de Funcionamento 30

2.3.1 Exemplo Simples 30

2.3.2 Exemplo Mais Elaborado 30

CAPÍTULO 3. ALGORITMOS MULTIPLICATIVOS ESPECIAIS 34

3 .1 Algoritmo de Multiplicação Modular de Gutub 34

3.2 Algoritmo de Montgomery 36

3.2.1 Método de Gutub para Multiplicação Modular de Montgomery ... 39

3.2.2 Método de Koç para Exponenciação Modular Utilizando a Multiplicação Modular de Montgomery 42

3.3 Algoritmos de Multiplicação por Alan Daly 44

3.3.1 Algoritmo de Multiplicação 1: MM1(X',Y') 45

3.3.2 Algoritmo de Multiplicação 2: MM2(X',Y') 47

3.3.3 Algoritmo de Multiplicação 3: MM3(X',Y') 48

3.4 Algoritmos de Exponenciação por Alan Daly 50

3.4.1 Algoritmo de Exponenciação 1: ME1(A,e) 51

3.4.2 Algoritmo de Exponenciação 2: ME2(A,e) 53

3.5 Considerações Finais 54

CAPÍTULO 4. IMPLEMENTAÇÃO DE ALGORITMOS EM LINGUAGEM DE PROGRAMAÇÃO C 56

4.1 Considerações Iniciais 56

4.2 Implementações com Operações Simples 57

4.3 Programa de Cálculo do Número "R^{-1}" 57
4.4 Implementações utilizando o Algoritmo de Montgomery 59
4.4.1 Primeira Implementação de Criptografia RSA Utilizando Montgomery 59
4.4.2 Segunda Implementação de Criptografia com RSA em Linguagem C 61
4.5 Considerações Finais 63
4.5.1 Chaves de mesmo tamanho com diferentes dimensões para "e" e "n" do algoritmo RSA). 65
CAPÍTULO 5. IMPLEMENTAÇÕES De ALGORITMOS CRIPTOGRÁFICOS EM VHDL 67
5.1 Implementação em VHDL utilizando o Algoritmo de Montgomery para Criptografia com o Algoritmo RSA 67
5.2 Implementação de Criptografia com o Algoritmo RSA Utilizando o Método de Gutub Associado ao Método de Quadratura e Multiplicação. 69
5.3 Arquitetura da Implementação com o Método de Gutub associado ao Método de quadratura e Multiplicação 72
5.4 Implementação de Criptografia com o Algoritmo RSA Utilizando o Método Rápido de Multiplicação Modular de Montgomery. 74
5.6.1 Comparação entre os Desempenhos em Linguagem C e em VHDL da Criptografia que Utiliza o Algoritmo de Montgomery 79
5.7 Comparação entre os Algoritmos de Brakley e de Montgomery para que Utiliza o Algoritmo Criptografia Criptográfico RSA em FPGAs. 82
CAPÍTULO 6. CONCLUSÕES FINAIS 83
6.1 Sugestões para Trabalhos Futuros 85
APÊNDICE A – Algoritmo gerador de números aleatórios 90
APÊNDICE B – Algoritmo de Euclides Estendido. 91
APÊNDICE C – Criptografia RSA Utilizando o Algoritmo de Brakley 92
APÊNDICE D – Almost Montgomery Inverse 96
APÊNDICE E – Criptografia RSA com o Método de Gutub Associado ao Método de Quadratura e Multiplicação em Linguagem VHDL 97

APÊNDICE F – Criptografia RSA Utilizando o Método Fast de Montgomery 99

APÊNDICE G – Somador 101

APÊNDICE H – Criptografia RSA Utilizando o Algoritmo Faster de Montgomery 103

CAPÍTULO 1. INTRODUÇÃO

Desde a idade antiga, no Egito e através de toda a história, a confidencialidade na transmissão de informações sempre teve um papel de grande importância. Nos últimos anos, com a proliferação dos meios de comunicação, especialmente o telex, o telefone e as redes de computadores, a segurança na transmissão/recepção de dados tornou-se um problema maior ainda, especialmente em razão da transmissão de dados bancários e do comércio eletrônico.

Diversos autores, como (KHALDOON, 2002) e (DALY, 2003) previam para os primeiros anos deste século, uma grande expansão da demanda por telefones celulares, computadores sem fio, TV por assinatura, cinema digital e, ampla proliferação das redes de computadores. Por essas razões e em favor dos esquemas de proteção a cópias de vídeo e áudio e dos milhões de operações comerciais e bancárias que se realizarão a cada segundo e demais operações que exijam sigilo, a segurança e rapidez nas informações será de capital importância. Por essa razão a cifragem e decifragem de mensagens que já são o objeto de muitos estudos serão motivos de grandes investimentos ainda.

Um dos principais desafios dos estudos de criptografia em sistemas computacionais refere-se à conciliação de um bom desempenho (segurança e alta velocidade de transmissão de dados) e baixo custo. As mensagens devem ser cifradas, enviadas e decifradas em tempo real.

Segundo (KHALDOON, 2002) existem três possíveis soluções para melhorar o desempenho da computação criptográfica: *Software*, *hardware* utilizando *Application-*

Specific Integrated Circuit (ASICs) e *Hardware* utilizando *Field-Programmable Gate Arrays* (FPGAs). A solução via *Software* é a mais barata e a mais flexível, entretanto, por utilizar recursos do computador (memória e tempo de processamento), é a mais lenta. A solução através de ASICs é a mais segura, em razão do encapsulamento do código, mais rápida, por ter um projeto dedicado, mas é inflexível (muito difícil de mudar o código), muito cara e necessita de um maior tempo de desenvolvimento. A solução utilizando FPGAs é flexível, rápida, relativamente barata, segura e necessita de um menor tempo para desenvolvimento de projetos do que as ASICs.

Existem algoritmos criptográficos de chave assimétrica, nos quais são utilizadas duas chaves, uma pública, que não precisa necessariamente ser secreta e uma chave privada, conhecida apenas pelo receptor da mensagem e que serve para decifrar a mesma e, algoritmos de chave simétrica que se baseia na utilização de uma única chave secreta, por ambos os interlocutores da troca de mensagens.

Segundo (MCTAGGART, 2001), a criptografia simétrica embora seja simples quanto à utilização, apresenta alguns problemas como:

- Para haver comunicação de forma segura uma rede de "n" usuários necessitaria de "n^2" chaves, quantidade que pode dificultar o gerenciamento das chaves;
- Existe o problema da comunicação e armazenamento das chaves de forma segura, entre as partes, o que nem sempre é fácil de ser garantido;
- Não pode garantir a identidade de quem enviou ou recebeu a mensagem, trazendo problemas quanto à autenticidade de quem enviou a mensagem e ao repúdio de quem a recebeu.

Uma forma de evitar os problemas da criptografia simétrica consiste em utilizar a criptografia assimétrica ou de chave pública, que está baseada no conceito de par de chaves, a saber, uma chave pública e uma chave privada. Uma das chaves é utilizada para cifrar uma mensagem e a outra para decifrá-la. As mensagens cifradas com uma das chaves do par só podem ser decifradas com a outra chave correspondente. A chave privada deve ser mantida em segredo enquanto a pública deve ser divulgada. Um importante problema da chave assimétrica é a sua lentidão em relação à chave simétrica (MORENO *et al.*, 2005).

Segundo (MCTAGGART, 2001), o principal algoritmo de chave assimétrica usado é o RSA (conhecido pelas iniciais de seus três criadores: *Rivest*, *Shamir* e *Adleman*), que utiliza enormes números primos para construir o par de chaves.

A vantagem do RSA consiste na facilidade para multiplicar dois números primos muito grandes, mas é muito difícil fatorá-los para obtê-los a partir do produto. Quebrar a chave privada a partir da chave pública envolve fatorar um número muito grande. Se o número for suficientemente grande e bem escolhido, será muito difícil conseguir realizar a fatoração em uma quantidade de tempo razoavelmente pequena (MCTAGGART, 2001).

Para tornar o processo de cifragem/decifragem mais rápido, utilizando o algoritmo RSA, ainda será necessário o desenvolvimento de algoritmos de exponenciação mais adequados do que os algoritmos atualmente existentes. Para tornar o processo criptográfico mais seguro é necessário aumentar o tamanho dos números. Isto tudo envolve a exponenciação modular, que nada mais é do que um processo de repetição da multiplicação modular. Assim, para tornar o processo ainda mais rápido é necessário que se empregue o método de exponenciação modular, que será explanado

neste trabalho, enfatizando na sua implementação em *Hardware* (FPGAs) o respectivo desempenho quando se usa o algoritmo RSA.

Um benefício importante da criptografia com chave pública é a assinatura digital, que torna possível garantir a autenticidade de quem envia a mensagem, bem como da integridade do seu conteúdo. Quando alguém deseja enviar uma mensagem e garantir sua integridade e autenticidade cifra a mesma com sua chave privada e a envia, em um processo denominado assinatura digital. Cada destinatário que receber a mensagem deverá decifrá-la, ou seja, verificar a validade da assinatura digital, utilizando para isso a chave pública do emitente da mensagem. Como a chave pública do emitente da mensagem apenas decifra (e garante a validade) mensagens cifradas com sua chave privada, fica assim garantida a autenticidade, integridade e não repudio da mensagem, pois se outra pessoa assiná-la ao invés do emitente da mensagem, o sistema de verificação não irá reconhecer a assinatura digital do emitente como sendo válida (RIVEST, 1978).

É importante perceber que a assinatura digital, como descrita no exemplo anterior, não garante a confidencialidade da mensagem. Qualquer um poderá acessá-la e verificá-la, mesmo um intruso, apenas utilizando a chave pública do emitente. Para obter confidencialidade com assinatura digital, basta combinar os dois métodos. O emitente primeiro assina a mensagem, utilizando sua chave privada. Em seguida, ele cifra a mensagem novamente, juntamente com sua assinatura, utilizando a chave pública do receptor da mensagem. Este, ao recebê-la, deve, primeiramente, decifrá-la com sua chave privada, o que garante sua privacidade. Em seguida, "decifrá-la" novamente, ou seja, verificar sua assinatura utilizando a chave pública do emitente, garantindo assim sua autenticidade (RIVEST, 1978).

O código *Hash* de uma mensagem é um resumo numérico que se faz da mensagem toda. Funciona como os dígitos verificadores dos números de contas-correntes dos bancos. Se uma mensagem que tem um determinado código *Hash* tiver um único *bit* alterado, produzirá um código *Hash* completamente diferente.

Conforme o exemplo da Tabela 1 e da Figura 1, Ari deseja enviar uma mensagem a Bia. Primeiramente ele cria o código *Hash* da mensagem. Isso garante a integridade da mesma; se a mensagem for alterada, o código *Hash* gerado por Bia, após ela recebê-la, será completamente diferente. Ele, então, cifra o código *Hash* com sua chave secreta (da, mad). Isso garante que a assinatura é sua (só poderá ser decifrada com sua chave pública). Em seguida, ele anexa o código *Hash* à mensagem e a cifra com a chave pública de Bia (eb, mb). Esse processo garante a confidencialidade da mensagem; somente Bia poderá decifrar a mensagem com sua chave privada. Agora a mensagem é enviada para Bia. Se Cid interceptar a mensagem ela não poderá ser decifrada, pois isso só será possível com o uso da chave privada de Bia (porque foi cifrada com sua chave pública).

Tabela 1. Cadastro de Chaves Criptográficas.

Usuário	Chave pública	Chave privada
Ari	(e_a, m_a)	(d_a, m_{ad})
Bia	(e_b, m_b)	(d_b, m_{bd})
Cid	(c_c, m_c)	(d_c, m_{cd})
outro	.....	.....

Segundo (MORENO *et al.*, 2005) os algoritmos simétricos mais conhecidos são o DES, Triple DES (de duas e três chaves), IDEA, Blowfish, RC5, CAST 128, RC2, RC4, Rijndael (AES), MARS, RC6, Serpent e Twofish.

Conforme (MCTAGGART, 2001) e (MORENO *et al.*, 2005), os principais algoritmos assimétricos e de *Hash* são: *Diffie-Hellman*, RSA, *Abreast Bavies-Meyer*,

Gost-Hash, Naval de 3, 4 e 5 passos, MD4, MD5 N-*Hash* (12 e 15 *Rounds*), RIPE-MD, RIPE – MD – 160, SHA e SNEFRU (4 e 8 passos).

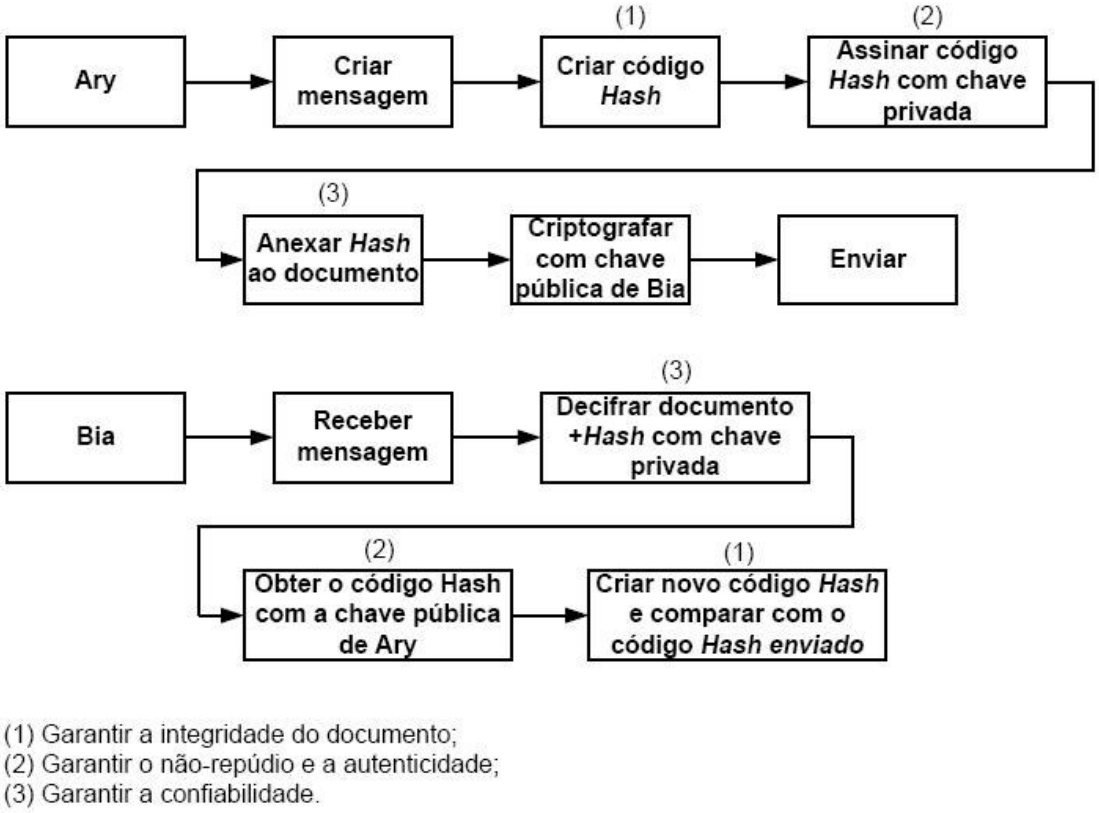

Figura 1 - Segurança no envio de mensagens cifradas.

Neste trabalho é examinado o desempenho do algoritmo criptográfico RSA para aplicação em criptografia computacional fazendo uso de chaves de diversos tamanhos utilizando algoritmos exponenciais, que foram implementados em linguagens C e VHDL (sendo prototipados em FPGAs) com o intuito de se efetuar comparações relativas à velocidade de cifragem/decifragem de mensagens, entre os algoritmos utilizados.

Justificativa

Segundo (MCTAGGART, 2001) os laboratórios RSA (empresa Americana pertencente aos criadores do algoritmo RSA) mencionam um estudo acerca da segurança conforme o tamanho de chaves baseados em técnicas de fatoração disponíveis em 1995. Pelo estudo, uma chave de 512 *bits* pode ser fatorada por menos de U$ 1 milhão em oito meses de trabalho. De fato, uma chave de 512 *bits* conhecida como RSA-155 foi fatorada em sete meses no ano de 1999, como parte do desafio regular de segurança dos laboratórios RSA.

A confecção de chaves com números muito grandes (128 a 2048 *bits* atualmente) é um processo computacional muito demorado. Daí a motivação para estudar algoritmos multiplicativos modulares, especialmente aqueles que utilizam o método de *Montgomery*, pois são os mais usados com o algoritmo RSA (KHALDOON, 2002), que trocam o tipo de operações aritméticas a serem realizadas (transforma divisões em subtrações, que são realizadas mais rapidamente pelo computador), produzindo significativa melhora no tempo de processamento do processo criptográfico.

Objetivo

O objetivo deste trabalho reside em examinar diversos algoritmos computacionais multiplicativos e exponenciais, especialmente aqueles que utilizam o método de multiplicação de *Montgomery* e sua aplicação em criptografia com o

algoritmo criptográfico RSA. Estes algoritmos são implementados em linguagem de programação C e em VHDL para se fazer comparações de simulação em *hardware* com trabalhos realizados por (CHIARAMONTE, 2006) e (MUZZI, 2005) que utilizaram o algoritmo de Brakley (MORENO *et al.*, 2005) para as operações de potenciação exigidas pelo algoritmo RSA.

Organização da Dissertação

Esta dissertação está composta de cinco capítulos, a saber:

No capítulo 1 apresenta-se a introdução da proposta de trabalho; é também feita uma introdução a respeito de conceitos de criptografia, destacando a importância do projeto (justificativa), os objetivos da dissertação e a organização da mesma.

No capítulo 2 é apresentado o funcionamento do algoritmo simétrico RSA, detalhes de sua implementação em linguagem C, exemplos da implementação e do desempenho em termos de velocidade dessas implementações.

No capítulo 3 é apresentado um suporte matemático para os algoritmos multiplicativos e exponenciais aplicados ao método de *Montgomery* que são discutidos neste estudo, assim como o método de *Montgomery* para multiplicação modular e também alguns outros algoritmos multiplicativos e exponenciais que utilizam a multiplicação modular.

No capítulo 4 é descrito o projeto de implementação de criptografia com o algoritmo de Montgomery aplicado ao RSA, em linguagem C e se analisa o impacto de diferentes algoritmos baseados na multiplicação de Montgomery na criptografia, quando se utiliza o algoritmo RSA.

No capítulo 5 são feitas três implementações do algoritmo de multiplicação modular de Montgomery usando a linguagem de descrição de *hardware* (VHDL), e se observou o impacto na velocidade de criptografia. Além disso, são examinados os aspectos arquiteturais das respectivas implementações.

No capítulo 6 são destacadas as conclusões e sugerem-se idéias para trabalhos futuros.

CAPÍTULO 2. O ALGORITMO ASSIMÉTRICO RSA

Este capítulo apresenta inicialmente, como suporte matemático, exemplos do funcionamento do algoritmo estendido de Euclides necessário para a compreensão do algoritmo RSA, sem, entretanto, apresentar o citado algoritmo. Apresenta ainda o funcionamento do algoritmo assimétrico RSA, bem como dá detalhes da sua implementação em *Software* (Linguagem C) e *hardware* (FPGAs).

2.1 Algoritmo Estendido de Euclides

O algoritmo de Euclides é uma maneira de se determinar o Máximo Divisor Comum de dois números. O algoritmo estendido de Euclides fornece uma combinação linear desses números.

Inicialmente se apresenta um exemplo de como se obtém o máximo divisor comum de dois números e em seguida obtém-se uma combinação linear destes dois números.

Observação: De maneira simplificada, podemos dizer que um conjunto de números forma uma combinação linear de um outro número quando, submetidos a operações de multiplicação ou divisão ou adição ou subtração, são iguais a esse número. Por exemplo, 14 é uma combinação linear de 2,3,4 e 5, porque 14 = 4.5 – 2.3.

As letras “MDC (x,y)” representam o Máximo Divisor Comum dos números x e y.

x mod m (lê-se x módulo m) representa o resto da divisão inteira de x por m. Por exemplo: 20 mod 6 = 2 (2 é o resto da divisão de 20 por 6).

Observação: De ora em diante, nas operações matemáticas de multiplicação estará sendo utilizado o"." para indicar essas multiplicações.

O algoritmo RSA é fundamentado no algoritmo estendido de Euclides, que fornece o inverso multiplicativo modular, portanto a existência e unicidade desse inverso multiplicativo modular são muito importantes. (DATA & RAKESNAKE, 2002) demonstram o Teorema da Existência do Inverso Multiplicativo Modular: "Os elementos de **Z** (**Z** é o conjunto dos números inteiros) que possuem inversos multiplicativos módulo **m** são aqueles relativamente primos a **m**, isto é, a congruência R.x= 1 mod m (**x** é o inverso multiplicativo de **R** módulo **m**) tem uma única solução módulo **m** se MDC(R,m) = 1. Se o inverso multiplicativo existe, ele pode ser determinado utilizando o algoritmo estendido de Euclides"

Exemplo:

Determinar $x = 32^{-1} \bmod 29$.

R = 32 e m = 29. Determinaremos R^{-1} e adicionalmente um fator **m',** obtido simultaneamente com R^{-1}, que será importante no decorrer desta dissertação.

Utilizando o algoritmo de Euclides para o MDC(32,29), tem-se (Figura 2)

	1	9	1	3
32	29	3	2	1
	3	2	1	0

Figura 2 – Exemplo 1 de utilização do Algoritmo de Euclides

Onde os números da primeira linha da Figura 2 representam os quocientes da divisão de 32 por 29, de 29 por 3, de 3 por 2 e de 2 por 1 e a terceira linha apresenta os

respectivos restos dessas divisões (que serão os próximos divisores). Este exemplo da utilização do algoritmo de Euclides mostra que o MDC(32,29) = 1, ou seja, 32 e 29 são primos entre si (números que têm MDC = 1).

O algoritmo estendido de Euclides prova que existem **R** e $\mathbf{R^{-1}}$, **m** e **m'** tais que

$R.\ R^{-1} - m.m' = MDC(R,m)$.

Observa-se que R. R^{-1} - m.m' é uma combinação linear do MDC(R,m)

Verifica-se que: $MDC(32,29) = 1 = 3-2$ (considerando R = 32 e m = 29)

$2 = 29 - 9.3 \Rightarrow 1 = 3 - (29 - 9.3) = 3 - 29 + 9.3 = 10.3 - 29 \Rightarrow$
$1 = 10.3 - 29$
$3 = 32 - 29 \Rightarrow 1 = 10.(32-29) - 29 = 10.32 - 10.29 - 29 =$
$10.32 - 11.29 \Rightarrow$
$32.10 - 29.11 = 1$
$R.\ R^{-1} - m.m' = 1 \Rightarrow$ R^{-1} mod m = 10 e m´= 11

Observe-se também que todos os termos destas igualdades são iguais a 1.

Exemplo:

3–(29-9.3) = 1

10 é o inverso multiplicativo de 32 mod 29, isto é:
$32.32^{-1} \bmod 29 = 1$ ou $32.10 \bmod 29 = 1$.

Prova:

$32.10 \bmod 29 = 320 \bmod 29 = 11+1/29$. Portanto, $32.10 \bmod 29 = 1$.

Observemos adicionalmente que:

$29.11 \bmod 32 = -1$.

Prova de que $R.R^{-1} \bmod m = 1$

Sabe-se que R. R^{-1}-m.m' = 1 $\Rightarrow$ ($R.R^{-1}$ - mm') mod m = 1 mod m $\Rightarrow$

$R.R^{-1} \bmod m - m.m' \bmod m = 1 \bmod m$

Como mm' mod m = 0 (porque mm' é divisível por m), então $R .R^{-1} \bmod m=1$

É também importante se notar que m' mod R = $- m^{-1}$

Prova:

$R R^{-1} - mm' = 1 \Rightarrow (R .R^{-1} - m.m') \bmod R = 1 \bmod R \Rightarrow$
$R .R^{-1} \bmod R - m.m' \bmod R = 1 \bmod R.$
Como $R R^{-1} \bmod R = 0$ (porque $R R^{-1}$ é divisível por R), então
$mm' \bmod R = 1 \Rightarrow -m' \bmod R = 1/m \Rightarrow m' \bmod R = - m^{-1}$

Pode ocorrer também uma situação em que R^{-1} é negativo, como no exemplo seguinte: Calcular R^{-1} e m' tal que R = 32 e m = 25.

		5	1	1	1
32	25	7	4	3	1
7		3	1	0	

Figura 3 – Exemplo 2 do Algoritmo de Euclides

Segundo a Figura 3, se observa que:

$7 = 32 - 25$
$4 = 25 - 3.7$
$3 = 7 - 4$
$1 = 4 - 3 = 4 - (7 - 4) = 4 - 7 + 4 = 2.4 - 7 =$
$2.(25 - 3.7) - 7 = 2.25 - 6.7 - 7 =$
$2.25 - 7.7 = 2.25 - 7.(32 - 25) = 2.25 - 7.32 + 7.25 =$
$32.(-7) + 9.25.$

Como pode ser observado R^{-1} = -7. Para transformá-lo num número positivo da mesma classe basta que se some 25 (o módulo m) a -7. Assim, R^{-1} = -7 + 25 = 18.

Verifique-se que 32.18 mod 25 = 576 mod 25 = 23 + 1/25 = 1 mod 25.

Para determinar m e m' tais que $R.R^{-1}$ - m.m' = 1 basta verificar que 32.18 = 576 dividido por 25 dá quociente 23 e resto 1, ou seja, $576=25.23+1$, logo

32.18 = 25.23+1.

Transpondo 25.23 para o primeiro membro obtém-se 32.18 – 25.23 = 1, donde se conclui que $R^{-1} = 18$ e m' = 23.

Concluídos os conceitos relativos ao algoritmo estendido de Euclides, fica agora mais fácil estudar o algoritmo RSA.

2.2 Introdução ao RSA

Segundo Mc Taggart (MC TAGGART, 2001), em 1978, três pesquisadores Ron *Rivest,* Adi *Shamir* e Leonard *Adleman*, desenvolveram um método prático para criptografia assimétrica baseado nas idéias sugeridas num artigo publicado por Diffie-Hellman em 1977 (DIFFIE-HELLMAN, 1977), descrito como método de chave assimétrica.

O sistema consiste em gerar uma chave pública, utilizada para cifrar os dados e uma chave privada, utilizada para decifrar os dados através de enormes números primos, o que dificulta a obtenção de uma chave a partir da outra.

Como em outros sistemas, o Algoritmo RSA utiliza enormes números primos para construir o par de chaves. Cada par compartilha o produto de dois números primos, o módulo e um expoente específico (MC TAGGART, 2001).

A segurança proporcionada por esse algoritmo, logicamente, aumenta à medida que aumentam o valor (tamanho em *bits*) dos números primos. Os números primos utilizados atualmente têm geralmente 512 *bits* de comprimento, mas em algumas aplicações, especialmente aquelas que exigem maior segurança podem chegar a 2048 *bits*.

A expectativa é que, com o aumento do poder computacional, será possível fatorar esses enormes números primos em um menor tempo, o que provocará a confecção de chaves com números maiores ainda (MC TAGGART, 2001).

Os algoritmos utilizados para as chaves pública e privada são definidos segundo Rivest (RIVEST, 1978) da seguinte maneira:

1. Escolhe-se dois números primos com pelo menos 64 bits (**p** e **q**); (existe um *Software* que realiza o cálculo desses números em (MORENO *et al.*, 2005).
2. Gera-se um número **n** através da multiplicação dos números escolhidos anteriormente (**n = p . q**);
3. Escolhe-se um número **d**, tal que **d** é menor que **n** e **d** é relativamente primo à **(p-1).(q-1)**;
4. Escolhe-se um número **e** tal que **ed – 1 seja divisível por (p-1).(q-1)**. Isto significa que **ed – x.(p-1).(q-1) = 1** (x é um número inteiro). Para realizar esse cálculo é necessário o algoritmo Estendido de Euclides (AMANOR, 2005).

Os valores **e** e **d** são chamados de expoentes público e privado, respectivamente. O par **(n,e)** é a chave pública e o par **(n,d)** é a chave privada. Os valores **p** e **q** devem ser sempre mantidos em segredo ou destruídos.

Para cifrar uma mensagem com esse algoritmo é realizado o seguinte cálculo:

$C = T^e \bmod n$, onde **C** é a mensagem cifrada, **T** é o texto original, **e** e **n** são dados a partir da chave pública **(n,e)**.

A única chave que pode decifrar a mensagem **C** é a chave privada **(n,d)** através do cálculo de: $T = C^d \bmod n$.

2.3 Exemplos de Funcionamento

Nesta seção serão apresentados exemplos de como funciona o algoritmo RSA, bem como serão feitas algumas comparações entre programas em linguagem de programação C que executam o algoritmo criptográfico RSA.

2.3.1 Exemplo Simples

Cifrar a letra H e decifrar o valor obtido, obtendo novamente a letra H. Supondo que o valor numérico de H seja 8 (oitava letra do alfabeto). Escolhe-se **p =3 e q = 5** (números primos). Determina-se **n = p.q => n = 3.5 => n = 15.** Escolhe-se **d = 11** (d<n e primo com **(3-1).(5-1) = 8).**

Com o algoritmo estendido de Euclides determina-se **3.11 - 4.8 = 1**

e.d – x.(p–1).(q–1) = 1 [d = 11 e (p–1).(q–1) = 10] => **x** = 4 e **e** = 3

Criptografando H = 8 (T =8)

$C = T^{e} \bmod m \Rightarrow C = 8^{3} \bmod 15 = 512 \bmod 15 = 2$

$T = C^{d} \bmod m \Rightarrow T = 2^{11} \bmod 15 = 2048 \bmod 15 = 8 = H$

2.3.2 Exemplo Mais Elaborado

O exemplo seguinte cifra e decifra a mensagem *"ITS ALL GREEK TO ME"*, (RIVEST, 1978) utilizando chaves assimétricas. Este exemplo foi utilizado também por Moreno (MORENO *et al.*, 2005)

Etapa 1: Satisfazendo a criação de chaves descrita anteriormente, são selecionados dois números primos aleatórios **p = 53**, **q = 61**. No exemplo utilizado por (MORENO *et al.*, 2005) foram escolhidos números pequenos para que possam ser representados facilmente.

Etapa 2: n = p . q, portanto n = 53 . 61 = 3233;

Etapa 3: escolher um número **d** tal que **d** seja menor que **n** e relativamente primo a **(p-1).(q-1).** Para tanto basta escolher um número primo aleatório maior que **p** e **q**. Para esse exemplo foi escolhido **d = 193**.

Etapa 4: escolhe-se um número **e** tal que **(ed-1)** seja divisível por **(p-1).(q-1)**, para realizar tal cálculo é utilizado o algoritmo de Euclides Estendido. Fazendo o uso desse algoritmo calcula-se que **e = 97**.

Portanto, neste exemplo, os valores importantes do algoritmo RSA são:

p = 53, q = 61, n = 53 . 61 = 3233, d = 193 e e = 97.

No artigo (*RIVEST* et al, 1978) as chaves utilizadas foram:

p = 47, q = 59, n = 2773, d = 157, e = 17.

Para realizar a criptografia foi adotada uma tabela

Tabela 2) para representar numericamente as letras maiúsculas do alfabeto. Quando se implementa na prática, a tabela utilizada é a tabela ASCII, que não será utilizada nesse exemplo para simplificar os cálculos.

A frase "*ITS ALL GREEK TO ME*" é representada numericamente como:

09 20 19 00 01 12 12 00 07 18 05 05 11 00 20 15 00 13 05 00

Tabela 2. Valores dos Caracteres para Exemplo do RSA.

	A	B	C	D	E	F	G	H	I	J	K	L
00	01	02	03	04	05	06	07	08	09	10	11	12
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25	26

Como **n = 3233**, a mensagem pode ser criptografada em blocos de duas letras, pois o valor máximo que pode aparecer na frase original é o equivalente à seqüência **"ZZ" = 2626** que é menor que **n (3233)**. Se utilizássemos blocos de três letras, poderíamos obter números tais como **"ABC" =10203>3233** Portanto, agrupando a mensagem em grupos de duas letras obtém-se:

I T S A L L G R E E K T O M E

0920 1900 0112 1200 0718 0505 1100 2015 0013 0500

Para cifrar cada bloco é realizado o cálculo: $\mathbf{C = T^e \bmod n,}$ onde **C** é a mensagem cifrada e **T** é a mensagem original.

Para T = 0920, temos: $C = 0920^{97} \bmod 3233 = 2546$;

Para T = 1900, temos: $C = 1900^{97} \bmod 3233 = 1728$;

Para T = 0112, temos: $C = 0112^{97} \bmod 3233 = 0514$;

Para T = 1200, temos: $C = 1200^{97} \bmod 3233 = 0210$;

Para T = 0718, temos: $C = 0718^{97} \bmod 3233 = 2304$;

Para T = 0505, temos: $C = 0505^{97} \bmod 3233 = 0153$;

Para T = 1100, temos: $C = 1100^{97} \bmod 3233 = 2922$;

Para T = 2015, temos: $C = 2015^{97} \bmod 3233 = 2068$;

Para T = 0013, temos: $C = 0013^{97} \bmod 3233 = 1477$;

Para T = 0500, temos: $C = 0500^{97} \bmod 3233 = 2726$.

Assim tem-se a mensagem cifrada:

2546 1728 0514 0210 2304 0153 2922 2068 1477 2426

Para decifrar cada bloco é realizado o cálculo: $\mathbf{T = C^d \bmod n,}$ onde **T** é a mensagem original e **C** é a mensagem cifrada.

Para C = 2546, temos: $T = 2546^{193} \bmod 3233 = 0920$;

Para C = 1728, temos: $T = 1728^{193} \bmod 3233 = 1900$;

Para C = 0514, temos: $T = 0514^{193} \bmod 3233 = 0112$;

Para C = 0210, temos: $T = 0210^{193} \bmod 3233 = 1200$;

Para C = 2304, temos: $T = 2304^{193} \bmod 3233 = 0718$;

Para C = 0153, temos: $T = 0153^{193} \bmod 3233 = 0505$;

Para C = 2922, temos: $T = 2922^{193} \bmod 3233 = 1100$;

Para C = 2068, temos: $T = 2068^{193} \bmod 3233 = 2015$;

Para C = 1477, temos: $T = 1477^{193} \bmod 3233 = 0013$;

Para C = 2726, temos: $T = 2726^{193} \bmod 3233 = 0500$.

Assim obtém-se a mensagem decifrada:

0920 1900 0112 1200 0718 0505 1100 2015 0013 0500,

que retorna à mensagem original.

Este capítulo teve o intuito especial de mostrar uma visão clara e didática da criptografia com o algoritmo matemático RSA. No próximo capítulo serão apresentados os algoritmos multiplicativos e exponenciais que serão as bases das implementações que posteriormente serão realizadas nos capítulo 5 e 6.

CAPÍTULO 3. ALGORITMOS MULTIPLICATIVOS ESPECIAIS

Os algoritmos matemáticos estudados neste capítulo terão utilidade na criptografia com o RSA. Tem-se o intuito de fazer comparações para a escolha dos mais adequados em termos de velocidade de criptografia com o RSA. Exemplos da forma como esses algoritmos efetuam operações computacionais serão apresentados.

3 .1 Algoritmo de Multiplicação Modular de Gutub

Adnan Gutub (GUTUB, 2000) propôs um algoritmo de multiplicação modular ao qual denominou "*The Modified Modulo Multiplication Algorithm*" que é apresentado na Figura 4, seguido de um exemplo numérico.

Algoritmo: The Modified Modulo Multiplication Algorithm

```
Entradas: X, Y e m (X,Y<m)
Saída: P = X.Y mod m
k é o número de bits em X (multiplicador)
i = k-1
X_i é o i-ésimo bit de X.
P = 0;
P = 2.P;
Se P >= m então P:= P - m;
Se X_i = 1 então
 {
  P:= P + Y;
  Se P >= m então P:= P - m
 };
i = i - 1;
if i<0 então GoTo 2;
Retorno P;
```

Figura 4 – Algoritmo de Gutub

A Figura 5, descreve o produto modular de 23.20 mod 27 (**X** = 23, **Y**= 20 e **m** = 27) utilizando o algoritmo computacional proposto por (GUTUB, 2000).

i	P:=2P	$P \geq m \Rightarrow$ P:=P-m Senão, P:=P	X_i	$X_i = 1 \Rightarrow$ P:=P+Y Senão, P:=P	$P \geq m \Rightarrow$ P:=P-m Senão, P:= P
4	0	0	1	0+20 = 20	20
3	40	40-27=13	0	13	13
2	26	26	1	26+ 20 = 46	46-27 = 19
1	38	38-27=11	1	11+20 = 31	31-27 = 4
0	8	8	1	8+20 = 28	28-27=1

Figura 5 – Multiplicação Modular utilizando o Algoritmo De Gutub (GUTUB,2000)

É indiferente se o multiplicando (Y = 20) e o módulo (m = 27) são números na base 2 ou 10 (ou qualquer outra). Entretanto, é fundamental que o multiplicador (X = 23)seja expresso na base 2. Nesse caso, o multiplicador, 23, é escrito $10111_{(2)}$.

X = 10111, portanto $X_0 = 1, X_1 = 1, X_2 = 1, X_3 = 0$ e $X_4 = 1$. Observe ainda que quando a soma P = P + X_i.Y é ímpar, soma-se **m** a **P** (que produz um número da mesma classe) e o transforma em número par (já que **m** é ímpar também) o que possibilita a sua divisão por 2 (ou $10_{(2)}$).

O algoritmo de Gutub é apropriado a operações de multiplicação modular, pois efetua multiplicações por dois, seguidas de uma subtração sempre que o valor do produto (em cada iteração) ultrapassar o valor do módulo. Este algoritmo, juntamente com o algoritmo de Montgomery, receberá especial estudo neste trabalho, porque se supunha até então que, por efetuar multiplicações muito rapidamente, ele tornaria o trabalho de criptografia mais rápido do que quando se utiliza o algoritmo de Montgomery.

3.2 Algoritmo de Montgomery

A multiplicação e a exponenciação modulares são operações aritméticas realizadas demoradamente pelos computadores em virtude de que elas exigem a multiplicação e a divisão, que são operações demoradas. O que se almeja é um processo que acelere a multiplicação modular, obtendo-se simultaneamente um grande impacto nas operações de radiciação e exponenciação modulares, que são processos repetitivos da multiplicação modular.

Segundo Khaldoon (KHALDOON, 2002), Todorov (TODOROV, 2000) e Koç (KOÇ, 1996), o algoritmo de multiplicação de *Montgomery* é um método muito engenhoso para se calcular a multiplicação modular. Ele substitui a divisão por um deslocamento e adição modular, se necessária, que são muito mais rápidas para o computador. Além disso, o algoritmo é bem adequado para implementação em *hardware(FPGA/ASIC).* Segundo (KHALDOON, 2002), o algoritmo é inadequado quando se pretende simplesmente realizar uma multiplicação modular, entretanto, quando a proposta é realizar um número muito grande de multiplicações esse processo de multiplicação modular é extremamente adequado.

Escreve-se em seguida o Método multiplicativo de *Montgomery* (MM). Para tanto, são introduzidas as seguintes definições e notações:

m é o módulo da multiplicação modular.

X é o multiplicador da multiplicação modular.

X_i é o *bit* de X que ocupa a posição i.

Y é o multiplicando da multiplicação modular.

Z é o produto modular (Z = X.Y mod m)

n é o número de *bits* do multiplicador

R é uma constante igual a 2^n.

R^{-1} é o inverso multiplicativo de R mod m ($R.R^{-1} \bmod m = 1$)

Z é o resultado (no conjunto dos Inteiros) da multiplicação de X por Y mod m

Z' é o resultado da multiplicação modular de Montgomery de X' por Y' no domínio de Montgomery

A multiplicação modular de *Montgomery* calcula $MM(X,Y) = XYR^{-1} \bmod(m)$, onde m é um inteiro no intervalo $2^{n-1} \leq m < 2^n$ tal que MDC(m,R) = 1 (que ocorre sempre que m for ímpar já que $R = 2^n$ é par). O algoritmo transforma um inteiro no intervalo [0, m-1] em outro inteiro do mesmo intervalo o qual será denominado m-resto do inteiro referido anteriormente e será pertencente ao conjunto que denominaremos Domínio de Montgomery conforme pode ser visto na Figura 6.

A seguir, calcula-se a multiplicação modular Z de dois inteiros X e Y utilizando o método de *Montgomery,* conforme figura 6.

As imagens de X e Y, X' e Y', no domínio de Montgomery, respectivamente, são obtidas por:

$X' = MM(X,R^2) = X.R^2.R.^{-1} \bmod m = X.R \bmod m$ (Figura 6)
$Y' = MM(Y,R^2) = YR^2.\ R.^{-1} \bmod m = Y.R \bmod m$ (Figura 6)

1. A imagem Z' de Z é calculada efetuando-se a multiplicação de *Montgomery* nas imagens X' e Y'

   $Z' = MM(X',Y') = MM(X.R \bmod m, YR \bmod m) = X.Y.R \bmod m$
   (Figura 6)

2. A multiplicação modular Z é obtida por MM(Z',1) assim: Z = MM(Z',1) = $X.Y.\ R.R^{-1} \bmod m = X.Y \bmod m = Z$ (Figura 6)

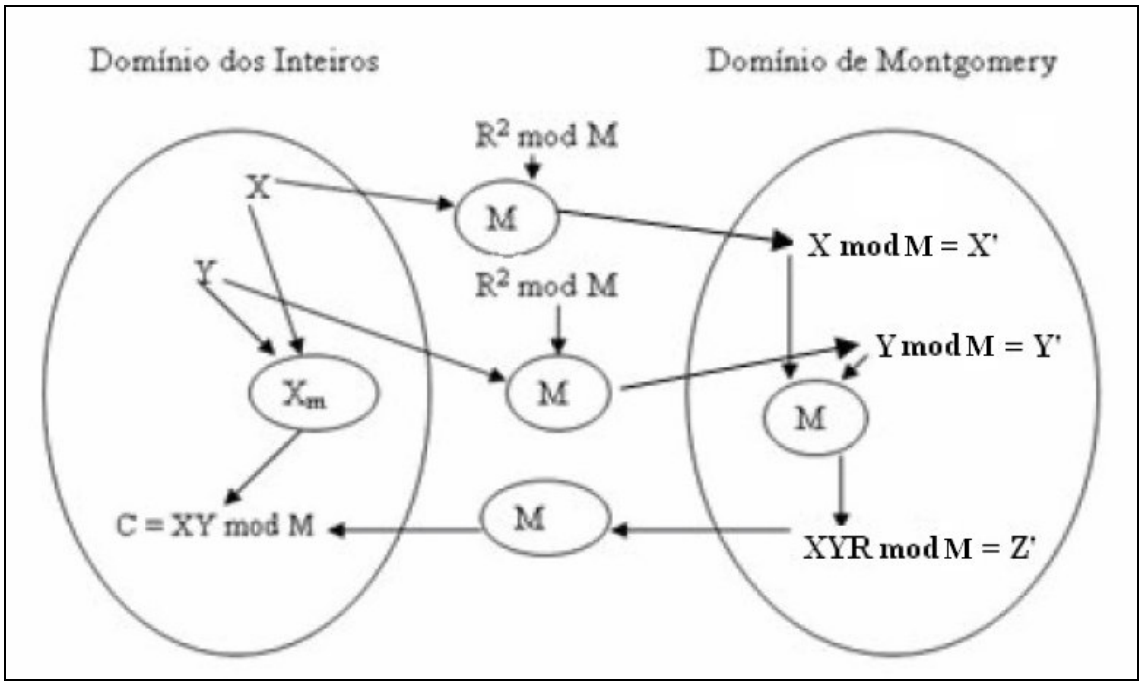

Figura 6 – Multiplicação Modular utilizando Montgomery.

Exemplo: Considera-se os seguintes valores:

Sejam X = 23, Y = 20 e m = 27 .

Observe-se que: `X.Y mod 27 = 23.20 mod 27 = 460mod 27 = 17 + 1/27 = 1`

Note-se que o número de *bits* de 27 (o módulo do exemplo) é 5, portanto n=5 e $R = 2^5 = 32$

Calcula-se $R^{-1} \bmod 27$ utilizando-se o algoritmo estendido de Euclides para o cálculo do máximo divisor comum entre 32 e 27 conforme Figura 7.

	1	5	2	1
32	27	5	2	2
5	2	1	0	

Figura 7 – Cálculo do MDC(32,27) com o Algoritmo de Euclides.

Utilizando-se o algoritmo estendido de Euclides:

$1 = 5 - 2.2 = 5 - 2(27 - 5.5) = 5 - 2.27 + 10.5 = 11.5 - 2.27 = 11.(32 - 27) - 2.27 = 11.32 - 11.27 - 2.27 = 11.32 - 13.27 = 32.11 - 27.13 = R.R^{-1} - m.m'.$

Então temos que $R^{-1} \bmod m = 11$ $(32^{-1} \bmod m)$ e m' = 13.

A seguir se calcula os valores de X', Y', Z'(no domínio de *Montgomery*) e Z(no domínio dos inteiros):

$X' = MM(23,32^2) = 23.32^2.32^{-1} \bmod 27 = 23.32 \bmod 27 = 736 \bmod 27 = 7$

$Y' = MM(20,32^2) = 20.32^2.32^{-1} \bmod 27 = 20.32 \bmod 27 = 640 \bmod 27 = 19$

$Z' = MM(7,19) = 7.19.32^{-1} \bmod 27 = 7.19.11 \bmod 27 = 1463 \bmod 27 = 5$

$Z = MM(Z',1) = Z'.1.R^{-1} = 5.1.32^{-1} \bmod 27 = 5.1.11 \bmod 27 = 55 \bmod 27 = 1$

A importância deste algoritmo se resume no fato de que são substituídas as operações de divisão (que são operações realizadas demoradamente pelos computadores), por multiplicações e subtrações. No caso específico de multiplicações e subtrações na base 2, as operações de multiplicação são realizadas por simples deslocamentos de bits à esquerda.

3.2.1 Método de Gutub para Multiplicação Modular de Montgomery

- A apresentação do método (Figura 8) é feita em paralelo com a apresentação do mesmo modelo numérico apresentado anteriormente, conforme (GUTUB, 2000).

- Passos de inicialização:
- Escolher R>m tal que $R = 2^k$ onde k é o número de *bits* de m, de forma que R e m sejam primos entre si.
- Computar R^{-1} e m´ tais que R. R^{-1} mod m = 1 e m.m' mod R = -1 (que ocorre quando se determina R. R^{-1} - m.m'=1 pelo algoritmo estendido de Euclides)
- Transformar X e Y em X' e Y' utilizando a multiplicação modular de *Montgomery*:

  X'= MM(X,R²)=X.R mod m e Y' = MM(Y,R²)= Y.R mod m

  Determinados X' e Y', utiliza-se então o algoritmo e Euclides para a determinação de Z'

```
Passos do cálculo de Z',
1. P = X'.Y'
2. U = P + m*(P*m'mod R)
3. S=U/R
4. Se S<m então Z' =  S
5. Senão Z' = S - m
```

Figura 8 – Algoritmo de Multiplicação Modular no Domínio de Montgomery

Após todas as iterações, fica determinado o valor de Z'.

Passo final : Transformar Z' para a representação normal:

$Z = Z'.R^{-1} \bmod m = MM(Z',1)$

Exemplo numérico: Considerando-se os mesmos valores utilizados no exemplo anterior (X = 23, Y = 20 e m = 27), tem-se .

Inicialização

1) Escolhe –se $R = 2^5$ e m = 27 (R>m e R e m primos entre si).

2) No exemplo resolvido anteriormente já se havia calculado:

$32.11 - 27.13 = 1$, logo, $R^{-1} \bmod 27 = 11$ e m´ = 13

3) $X' = MM(X, R^2) = MM(23, 32^2) = 23.32^2.32^{-1} \bmod 27 = 23.32 \bmod 27 = 7.$

$Y' = MM(Y, R^2) = MM(20, 32^2) = 20.32^2.32^{-1} \bmod 27 = 20.32 \bmod 27 = 19.$

Utilização do Algoritmo - Cálculo de Z'

$P = X'.Y' \Rightarrow P = 19.7 = 133$

$U = P + m.(P.m' \bmod R) \Rightarrow U = 133 + 27.(133.13 \bmod 32)$
$133 + 27.1 = 160$

$S = U/R \Rightarrow S = 160/32 \Rightarrow S = 5$
$S < m \Rightarrow Z' = 5$

Passo Final: Cálculo de Z

$Z = MM(Z', 1) = Z'.1.R^{-1} \bmod m = 5.1.32^{-1} \bmod 27 = 5.1.11 \bmod 27 = 55 \bmod 27 = 1.$

Segundo (GUTUB, 2000), a diminuição de tempo quando se utiliza este algoritmo não é muito significante quando os valores utilizados são pequenos, entretanto, para números grandes (superiores a 64 *bits*), efetuar multiplicações por 2, que representam apenas deslocamentos do número binário à esquerda e em seguida subtrações, provoca uma diminuição de tempo significativa em comparação com as operações de multiplicação e divisão (a operação módulo **m** dá como resultado o resto da divisão de um número inteiro por **m**), que são efetuadas de forma muito mais lenta pelo computador.

3.2.2 Método de Koç para Exponenciação Modular Utilizando a Multiplicação Modular de Montgomery

Inicialmente mostra-se com um exemplo como se calcula a exponenciação no domínio de *Montgomery*, seguindo-se o método proposto em (SAVAS & KOÇ, 2000).

Exemplo: Calcular $23^5 \bmod 27$.

- No domínio dos Inteiros $23^5 \bmod 27 = 6436343 = 238383 + 2/27$. Portanto, $23^5 \bmod 27 = 2$.

- No domínio de *Montgomery*, a exponenciação só pode ser realizada por lei de recorrência, isto é, para se calcular $(X')^n$ é necessário conhecer o valor de $(X')^{n-1}$.

De outra forma, pode-se dizer que:

$(X' \bmod m) = (...(((X'.X' \bmod m).X' \bmod m).X' \bmod m \ldots ... \ (X' \bmod m).$

As operações de multiplicação de *Montgomery* utilizarão as mesmas condições já expostas anteriormente, isto é, X = 23, Y = 20, m = 27 e R = 32.

Com o algoritmo estendido de Euclides calcula-se $R^{-1} = 11$.

Processo:

1 – Transforma-se o número 23 na sua imagem no domínio de *Montgomery*:

$X' = MM(X, R^2) = MM(23, 32^2) = 23.32^2.11 \bmod 27 = 23.32 \bmod 27 = 7$

2 – Efetua-se passo a passo as multiplicações de X' por si mesmo.

$(X')^2 = MM(7,7) = 7.7.11.\bmod 27 = 26$

$(X')^3$ = MM(7,26) = 7.26.11 mod 27 = 4
$(X')^4$ = MM(7,4) = 7.4.11 mod 27 = 11
$(X')^5$ = MM(7, 11) = 7.11.11 mod 27 = 10

3 – Para se obter $(X)^5$ no conjunto dos inteiros basta efetuar a multiplicação de *Montgomery* entre $(X')^5$ e 1.

$(X)^5$ mod 27 = MM($(X')^5$, 1) = MM(10,1)= 10.1.11 mod 27 = 110 mod 27 = 2

Observa-se que o resultado obtido é o mesmo que o anterior quando se calculou 23^5 mod 27 pelo método direto.

Na Figura 9, é apresentado o algoritmo proposto por (KOÇ, 1996) para exponenciação modular

Seja "j" o número de *bits* do expoente "e", e_i, o i-ésimo *bit* de "e", "a", a base da potência que se deseja obter e m o módulo da multiplicação modular.

Define-se:

Função ME(a,e,m)

```
Passo 1: a' = MM(a,R²) = a.R mod m
Passo 2: X' = MM(R²,1) = R mod m
Passo 3: Para i variando de j-1 até 0
                X':= MM(X'.X')
                    Se  ej = 1 então
                          X':= MM(X',a')
Passo 4: Retorno X:= MM(X',1).
```

Figura 9 – Algoritmo de Koç para exponenciação modular

Exemplo: Considerando os mesmos valores utilizados em cálculos anteriores, agora aplicando o algoritmo de Koç, calcular 23^5 mod 27 (também calculado anteriormente): **a** = 23, **e** = 5 = 101_2, **R** = 32 e $\mathbf{R}^{-1} = 11$.

```
Passo 1: a' = MM(a,R²) = a.R mod m = 23.32 mod 27 = 7
Passo 2: X' = 1.R mod m = 1.32 mod 27 = 5
Passo 3: para i variando de 2 até 0
      i = 2 ⇒  X':= MM(X',X')= MM(5,5)= 5.5.11 mod 27 = 5
           e2= 1 ⇒X':= MM(X',a')= MM(5,7)= 5.7.11 mod 27 = 7
      i = 1 ⇒ X':= MM(X',X')= MM(7,7)= 7.7.11 mod 27 = 26
           e1= 0
      i = 0 ⇒ X':= MM(X',X')= MM(26,26)= 26.26.11 mod 27 = 11
           e0=1 ⇒ X':= MM(X',a')= MM(11,7)= 11.7.11 mod 27 = 10
Passo 4: X:= MM(X',1) = MM(10,1) = 10.1.11 mod 27 = 2
```

Figura 10 – Cálculo de 23^5 com o Método Exponencial de Koç

Como pode ser verificado na Figura 10, obteve-se $23^5 \bmod 27 = 2$ efetuando-se a potenciação no conjunto dos inteiros ou no domínio de Montgomery bem como utilizando o algoritmo proposto por (SAVAS & KOÇ, 2000).

3.3 Algoritmos de Multiplicação por Alan Daly

Nesta sessão são apresentados cinco algoritmos propostos por Allan Daly (DALY, 2002), os três primeiros multiplicativos e os demais exponenciais (que utilizam a multiplicação como base das operações).

Esses algoritmos mantêm as mesmas características definidas anteriormente, isto é, $R = 2^k$, m e R são primos entre si e X e Y são menores do que m. Além disso, para ficar assegurado que Z' permanecerá limitado, isto é, $0 \leq Z' < m$, deverá se definir n = k+2, onde k é o número de *bits* de R.

Cada uma das funções que são apresentadas computará um produto da forma $Z' = MM(X', Y') = X'.Y'.R^{-1} \bmod m$, onde X' e Y'(imagens de X e Y) estão no

domínio de *Montgomery* e devem ser pré-calculados. Assim, X' = MM (X, R^2) = X.R mod m e analogamente, Y' = YRmod m.

Entretanto, o valor R^2 não pertence ao intervalo permitido para os valores de entrada da função MM ($0<m<2^{n-2}$), então devem ser computados:

$X' = MM(X, 2^{2n} \bmod m) = X.\ (2^{2n} \bmod m).2^{-n} \bmod m = (X.2^{n} \bmod m) \bmod m$

e analogamente, **Y'=** $MM(Y, 2^{2n} \bmod m) = (Y.\ 2^{n} \bmod m) \bmod m$.

3.3.1 Algoritmo de Multiplicação 1: MM1(X',Y')

Sendo X e Y dois números inteiros e X' e Y' suas respectivas imagens no domínio de Montgomery ; sendo m o módulo da multiplicação modular, tem-se na Figura 11, o algoritmo de multiplicação 1 de Daly, cujo retorno S = Z' = MM(X',Y').

```
{
  M = (m+1)/2 ;
  S-1 = 0;
  for i = 0 to n do
      qi := (Si-1) mod 2;          // (LSB of Si-1)
      Si := Si-1 / 2 + qi .M + X'i.Y'
  end for
  Return Sn
}
```

Figura 11 – Algoritmo de Multiplicação 1 de Daly.

Exemplo numérico:

Nos cinco exemplos numéricos seguintes serão utilizados X = 16, Y = 12, R = 2^{5+2} = 128 e m = 27(5 *bits*) => k = 5.

Observe-se que 16.12 mod 27 = 3, pois utilizando os três algoritmos multiplicativos de Daly, obter-se-á como resultado o mesmo valor (3).

1) Passos de inicialização (servem para os três algoritmos de multiplicação).

- Calcula-se R = 128 mod 27 e obtém-se R = 20.
- Utilizando o algoritmo estendido de Euclides e considerando R = 20, determina-se R^{-1} mod 27 = 23.
- Transformando X em X' e Y em Y' no domínio de *Montgomery*:

X′ = MM(16,20²) = 16.20 mod 27 = 23 (10111_2) (coincidência com R^{-1}) e
Y′ = MM(12,20²) = 12.20 mod 27 = 24.

2) Utilizando o algoritmo 1 de Daly

M = (m+1)/2 ⇒M = (27+1)/2 ⇒ M = 14
S_{-1} = 0, X′ = 00010111.
Para i = 0 até 7

- i = 0 ⇒ q_0= 0 mod 2 = 0;
  X'_0=1 ⇒ S_0 = 0/2 + 0.14 + 1.24 = 24
- i = 1 ⇒ q_1 = 24 mod 2 = 0
  X'_1=1 ⇒ S_1 = 24/2 + 0.14 + 1.24 = 36
- i = 2 ⇒ q_2= 36 mod 2 = 0
  X'_2 = 1 ⇒ S_2 = 36/2 + 0.14 + 1.24 = 42
- i = 3 ⇒ q_3= 42 mod 2 = 0
  X'_3 = 0 ⇒ S_3 = 42/2 + 0.14 + 0.24 = 21
- i = 4 ⇒ q_4 = 21 mod 2 = 1
  X'_4 = 0 ⇒ S_4 = 21/2 + 1.14 + 1.24 = 48
- i = 5 ⇒ q_5 = 48 mod 2 = 0
  X'_5 = 0 ⇒ S_5 = 48/2 + 0.14 + 0.24 = 24
- i = 6 ⇒ q_6 = 24 mod 2 = 0
  X'_6 = 0 ⇒ S_6 = 24/2 + 0.14 + 0.24 = 12
- i = 7 ⇒ q_7 = 12 mod 2 = 0
  X'_7 = 0 ⇒ S_7 = 12/2 + 0.14 + 0.24 = 6

Z′ = S ⇒ Z′ = 6.

3) Passo final: Retorna-se ao domínio dos inteiros:

$Z = MM(Z',1) = Z'.1.R^{-1} \bmod m \Rightarrow Z = 6.1.23 \bmod 27 = 3$

3.3.2 Algoritmo de Multiplicação 2: MM2(X',Y')

Sendo X e Y dois números inteiros e X' e Y' suas respectivas imagens no domínio de Montgomery; sendo m o módulo da multiplicação modular, tem-se na Figura 12, o algoritmo de multiplicação 2 de Daly, cujo retorno S é igual a MM(X',Y').

```
{
    S_-1 = 0;
  for i = 0 to n-1  do
      q_i := (S_i-1 + X'_i) mod 2;          // (LSB of S_i-1)
       S_i := (S_i-1 + q_i .m + X'_i.Y') / 2
  end for
  Return S_n-1
}
```

Figura 12 – Algoritmo de Multiplicação 2 de Daly.

Exemplo numérico:

1) Inicialização: transformar X e Y, respectivamente em X' e Y' no domínio de *Montgomery* conforme descrito na introdução (item 3.2), sendo m o módulo e $R = 2^n$.

2) Utilizando o algoritmo 2 de Daly (Figura 12)

$S_{-1} = 0, \quad X' = 00010111.$

Para i = 0 até 7

- $i = 0 \Rightarrow q_0 = (0 + 1.24) \bmod 2 = 0;$
  $X'_0 = 1 \Rightarrow S_0 = (0 + 0.27 + 1.24)/2 = 12$

- $i = 1 \Rightarrow q_1 = (12 + 1.24) \bmod 2 = 0$

$X'_1=1 \Rightarrow S_1 = (12 + 0.27 + 1.24)/2 = 18$

- $i = 2 \Rightarrow q_2 = (18+1.24) \bmod 2 = 0$
  $X'_2 = 1 \Rightarrow S_2 = (18+ 0.27 + 1.24)/2 = 21$

- $i = 3 \Rightarrow q_3 = (21 + 0.24) \bmod 2 = 1$
  $X'_3 = 0 \Rightarrow S_3 = (21 + 1.27 + 0.24)/2 = 24$

- $i = 4 \Rightarrow q_4 = (24 + 1.24) \bmod 2 = 0$
  $X'_4 = 0 \Rightarrow S_4 = (24 + 0.27 + 1.24)/2 = 24$

- $i = 5 \Rightarrow q_5 = (24 + 0.24) \bmod 2 = 0$
  $X'_5 = 0 \Rightarrow S_5 = (24 + 0.27 + 0.24)/2 = 12$

- $i = 6 \Rightarrow q_6 = (12 + 0.24) \bmod 2 = 0$
  $X'_6 = 0 \Rightarrow S_6 = (12 + 0.27 + 0.24)/2 = 6$

Portanto, Z' = 6

3) Passo final: Como no algoritmo 1 de Daly, retorna-se ao domínio dos inteiros e obtém-se para Z o valor 3.

3.3.3 Algoritmo de Multiplicação 3: MM3(X',Y')

Sendo X e Y dois números inteiros e X' e Y' suas respectivas imagens no domínio de Montgomery ; sendo m o módulo da multiplicação modular, tem-se na Figura 13, o algoritmo de multiplicação 3 de Daly, cujo retorno S = Z' = MM(X',Y').

```
{
    S_-1 = 0;
    Y:= 2.Y
  for i = 0 to n  do
```

```
      q_i := (S_{i-1}) mod 2;          // (LSB of S_{i-1})
      S_i := (S_{i-1} + q_i .m + X'_i.Y') / 2
  end for
  Return S_n
}
```

Figura 13 – Algoritmo de Multiplicação 3 de Daly.

Exemplo numérico:

1) Inicialização: Transformar X e Y, respectivamente em X' e Y' no domínio de *Montgomery* conforme descrito na introdução, sendo m o módulo e $R = 2^n$.

2) Utilizando o algoritmo 3 de Daly (Figura 13)

$S_{-1} = 0$, $X' = 00010111$.
$Y = 2.24 = 48$

Para i = 0 até 7

- $i = 0 \Rightarrow q_0 = 0 \bmod 2 = 0$;
  $X'_0 = 1 \Rightarrow S_0 = (0 + 0.27 + 1.48)/2 = 24$

- $i = 1 \Rightarrow q_1 = 24 \bmod 2 = 0$
  $X'_1 = 1 \Rightarrow S_1 = (24 + 0.27 + 1.48)/2 = 36$

- $i = 2 \Rightarrow q_2 = 36 \bmod 2 = 0$
  $X'_2 = 1 \Rightarrow S_2 = (36 + 0.27 + 1.48)/2 = 42$

- $i = 3 \Rightarrow q_3 = 42 \bmod 2 = 0$
  $X'_3 = 0 \Rightarrow S_3 = (42 + 0.27 + 0.48) = 21$

- $i = 4 \Rightarrow q_4 = 21 \bmod 2 = 1$
  $X'_4 = 0 \Rightarrow S_4 = (21 + 1.27 + 1.48)/2 = 48$

- $i = 5 \Rightarrow q_5 = 48 \bmod 2 = 0$

$X'_5 = 0 \Rightarrow S_5 = (48 + 0.27 + 0.48)/2 = 24$

- $i = 6 \Rightarrow q_6 = 24 \bmod 2 = 0$

  $X'_6 = 0 \Rightarrow S_6 = (24 + 0.27 + 0.48) = 12$

- $i = 7 \Rightarrow q_7 = 12 \bmod 2 = 0$

  $X'_7 = 0 \Rightarrow S_7 = (12 + 0.27 + 0.48) = 6$

$Z' = S \Rightarrow Z'$

3) Passo final: Retorna-se ao domínio dos inteiros:

$Z = MM(Z',1) = Z'.1.R^{-1} \bmod m \Rightarrow Z = 6.1.23 \bmod 27 = 3$

Com base nestes algoritmos, (DALY,2002) propôs dois algoritmos exponenciais que serão examinados no item 3.4.1.

3.4 Algoritmos de Exponenciação por Alan Daly

A exponenciação modular é realizada repetindo-se multiplicações modulares. Os dois algoritmos são: O método binário da esquerda para a direita e o método binário da direita para a esquerda. Nestes algoritmos são considerados os valores A no conjunto dos inteiros , "e", o expoente, m o módulo, C uma constante igual a $2^{2n} \bmod m$, A' a imagem de A no domínio de *Montgomery*, k o número de *bits* de "e", "e_i", cada *bit* de "e", R = $2^{2n} \bmod m$ e R^{-1} obtido com o algoritmo estendido de Euclides conforme já discutido anteriormente.

3.4.1 Algoritmo de Exponenciação 1: ME1(A,e)

Sendo **A** um número no conjunto dos Inteiros e **C** e **A'** as imagem de **1** (um) e **A,** respectivamente**,** no domínio de Montgomery, obtém-se a exponenciação $\mathbf{A^n}$, utilizando-se o Algoritmo de exponenciação 1 de Daly (Figura 14), como se segue:

Inicialização:

1. Obtém-se: $C := 2^{2n} \bmod m$

2. Obtém-se: A' : = MM(A,C), equivalente a MM(X,R²), determinando a imagem de A

3. Obtém-se: R:= MM(C,1) (equivalente a R² mod m)

```
for i = k-1 downto 0
       X':= MM(X',X')
        If  e_i = 1 then
            X':= MM(R,X')
         End if
   End for
  X:= MM(X',1)  //  Retornando X' ao conjunto dos inteiros
   Return X
}
```

Figura 14 – Algoritmo de Exponenciação 1 de Daly.

Apresenta-se em seguida um exemplo numérico.

Exemplo numérico:

Determinar $17^5 \bmod 27$ (A = 17 , e = 5)

Observação: $17^5 \bmod 27 = 1419857 \bmod 27 = 8$

1) Calcula-se $C := 2^{2n} \bmod m$

$C = 2^{2.7} \bmod 27 = 2^{14} \bmod 27 = 16384 \bmod 27 = 22$

2) Calcula-se A'

$A' = MM(C,A) = C.A.R^{-1} \bmod m$ ($R^{-1} = 23$ calculado no algoritmo 1 de Daly)

$A' = MM(22,17) = 22.17.23 \bmod 27 = 16$

3) Calcula-se $X' = R = MM(C,1) = MM(22,1) = 22.1.23 \bmod 27 = 20$

Para $i = 2$ até 0 ($e = 5 = 101_2 \Rightarrow k = 3$)

- $i = 2 \Rightarrow X' = MM(X',X') = MM(20,20) = 20.20.23 \bmod 27 = 20$ (não é mera coincidência de valores. Na verdade equivale a $17^0 = 1$)

  $e^2 = 1 \Rightarrow X' = MM(A',X') = A'.X'.R^{-1} \bmod m = 16.20.23 \bmod 27 = 16$ (não é mera coincidência de valores. Na verdade equivale a $17^1 = 17$).

- $i = 1 \Rightarrow X' = MM(X',X') = MM(16,16) = 16.16.23 \bmod 27 = 2$
  $e_1 = 0$

- $i = 0 \Rightarrow X' = MM(X',X') = 2.2.23 \bmod 27 = 11$
  $e_0 = 1 \Rightarrow X' = MM(A',X') = 16.11.23 \bmod 27 = 25$

Passo final: retorna- se X' para o domínio dos inteiros fazendo-se:

$X = MM(X',1) = X'.1.R^{-1} \bmod m = 25.1.23 \bmod 27 = 8$

3.4.2 Algoritmo de Exponenciação 2: ME2(A,e)

Sendo **A** um número no conjunto dos Inteiros e **C** e **A'** as imagem de **1** (um) e **A**, respectivamente, no domínio de Montgomery, obtém-se a exponenciação $\mathbf{A^n}$, utilizando-se o Algoritmo de exponenciação 2 (Figura 15) de Daly como se segue:

Inicialização:

1) Obtém-se C:= 2^{2n} mod m

2) Obtém-se A': = MM(A,C) //equivalente a MM(X,R²),

determinando a imagem de A

3) Obtém-se R:= MM(C,1) (equivalente a R² mod m)

```
{
  for i = k-1 downto 0
          If e^i = 1 then
             X':= MM(A',X')
          End if
          A':= MM(A',A')
   End for
  X:= MM(X',1)  //  Retornando X' ao conjunto dos inteiros
   Return X
}
```

Figura 15 - Algoritmo de Exponenciação 2 de Daly.

Apresenta-se em seguida um exemplo numérico.

Exemplo numérico.

Determinar 17^5 mod 27 (A = 17, e = 5)

1) Calcula-se C:= 22n mod m

C = $2^{2.7}$ mod 27 = 2^{14} mod 27 = 16384 mod 27 = 22

2) Calcula-se A'

$A' = MM(C,A) = C.A.R^{-1} \bmod m$ ($R^{-1} = 23$ calculado no algoritmo 1 de Daly)

$A' = MM(22,17) = 22.17.23 \bmod 27 = 16$

3) Calcula-se $X' = R = MM(C,1) = MM(22,1) = 22.1.23 \bmod 27 = 20$

Para $i = 0$ até 2 ($e = 5 = 101_2 \Rightarrow k = 3$)

- $e_0 = 1 \Rightarrow X' = MM(A',X') = MM(16,20) = 16.20.23 \bmod 27 = 1$
  $A' = MM(A',A') = MM(16,16) = 16.16.23 \bmod 27 = 2$

- $e_1 = 0 \Rightarrow A' = MM(A',A') = MM(2,2) = 2.2.23 \bmod 27 = 11$

- $e_2 = 1 \Rightarrow X' = MM(A',X') = MM(11,16) = 11.16.23 \bmod 27 = 25$
  $A' = MM(A',A') = MM(11,11) = 11.11.23 \bmod 27 = 2$

Passo final: retorna- se X' para o domínio dos inteiros fazendo-se:

$X = MM(X',1) = X'.1.R^{-1} \bmod m = 25.1.23 \bmod 27 = 8$

3.5 Considerações Finais

Neste capítulo a intenção foi, mais do que mostrar os algoritmos de multiplicativos e exponenciais, apresentar exemplos numéricos de fácil execução que permitem o fácil entendimento desses algoritmos.

Segundo (DALY, 2002), no algoritmo ME1(A,e), as operações de quadratura e multiplicação devem ser executadas seqüencialmente e, conseqüentemente, as 2n multiplicações devem ser executadas em série. Isso significa que ambas as operações, quadrado e multiplicação, podem ser executadas em um único multiplicador, desta forma economizando área em um *hardware(*FPGA*)*. No algoritmo ME2(A,e), as

operações de quadrado e multiplicação são independentes e podem ser executadas em paralelo. Desta forma, 50 % dos clocks são requeridos para completar a exponenciação. Portanto, para se conseguir velocidade, em virtude da arquitetura proposta por Daly, são necessários dois multiplicadores.

CAPÍTULO 4. IMPLEMENTAÇÃO DE ALGORITMOS EM LINGUAGEM DE PROGRAMAÇÃO C

Nesta seção são apresentadas implementações de programas em linguagem de programação C que executam o algoritmo criptográfico RSA e também uma implementação que calcula o valor R^{-1} do algoritmo estendido de Euclides (R. R^{-1} – m.m' = 1), utilizada na multiplicação modular de Montgomery.

4.1 Considerações Iniciais

Todas as pessoas que utilizam a Internet e realizam operações comerciais ou bancárias sabem que as facilidades para realizar operações destes tipos trazem junto consigo um perigo eminente: o perigo dos *rackers*; eles realmente "invadem" os computadores e obtêm informações valiosas dos usuários da rede mundial de computadores, se utilizam dessas informações e trazem imensos prejuízos às pessoas físicas e jurídicas. Este fato por si só representa uma fortíssima razão para que se obtenha maior segurança na comunicação e manutenção de dados. Por essa razão, a criptografia tende a desempenhar em muito pouco tempo um papel muito mais importante daquele que possui agora. A melhoria na segurança passa pela obtenção de algoritmos computacionais e criptográficos proporcionalmente mais eficientes à medida que se aumenta o poder computacional.

Em seguida serão feitas diversas implementações em linguagem C, relacionadas com a criptografia ou que auxiliam nos processos criptográficos em estudo.

4.2 Implementações com Operações Simples

O primeiro passo dentro dos trabalhos desta implementação foi o de se construir um algoritmo que realizasse as operações de potenciação exigidas pelo RSA com as rudimentares operações de multiplicação seguidas (em cada interação) de uma subtração (Figura 16), quando o valor do produto superasse o valor do módulo.

```
int eleva(unsigned long int a,unsigned long int b,unsigned long
int m) //efetua a elevado a b mod m
    {
                unsigned long int i, c;
                c = 1;
               for(i=0;i<b;i++)
              {
                 c=c*a;
                 while (c>m)
                    c=c-m;
              }
          return(c);
    }
```

Figura 16 – Função em Linguagem C na Implementação de Criptografia com o RSA

Essa implementação em linguagem de programação C revelou-se demorada em relação ao parâmetro estabelecido (utilização do método de *Brakley*, (MORENO *et al.*, 2005)), para chaves minúsculas (até 16 *bits*) e totalmente ineficiente para chaves de 24 *bits* (77 horas para criptografar um arquivo de 0,5 MB).

4.3 Programa de Cálculo do Número "R^{-1}"

A Figura 17 apresenta um programa importante para utilização do Algoritmo de Montgomery em linguagem C, implementado neste trabalho. Este programa calcula

o valor R^{-1} na expressão $R.R^{-1} – m.m' = 1$ em menos de 15s, para números até com 32 *bits*.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
 unsigned long int k,n;
 unsigned long int
Inverso(R,m)
 {
float z;
     unsigned long int i;
    unsigned long int
r,u,v,s;
    u=m;
    v=R;
    r=0;
    s=1;
    k=0;
    while(v>0)
    {
       if(u%2 == 0)
       {
          u = u/2;
          s = 2*s;
       }
       else if( v%2 == 0)
       {
          v = v/2;
          r = 2*r;
       }
       else if(u>v)
       {
          u = (u-v)/2;
          r = r+s;
          s = 2*s;
       }
       else if(v>=u)
       {
          v = (v-u)/2;
         s = s + r;
          r = 2*r;
         }
         k = k+1;
      }
       if(r>=m)
       {
          r = r - m;
       }
       r = m-r;
      for(i=0;i<k;i++)
      {
       if(r%2 == 0)
           r = r/2;
       else
           r = (r+m)/2;
         }
       return r;
 }
unsigned long int main()
{
unsigned long int R, m,x;
printf(" \nDigite o valor de
R:  ");
scanf("%d", &R);
printf(" Digite o valor de
m:");
scanf("%d", &m);
while(m%2 == 0)
{
   printf(" INVALIDO - Digite
para m um numero impar");
   scanf("%d",&m);
}

x = Inverso(R,m);
printf("inverso = %d ", x);
printf("\n1'= %d ", R-m);
system("pause");
}
```

Figura 17 – Programa de Cálculo do número R^{-1}

A principal dificuldade na implementação ocorreu especialmente em razão das limitações computacionais (excessiva lentidão) para se obter o número R^{-1} com valores

superiores a 32 *bits*. Para se construir uma chave de 64 *bits*(com **m** = 32 *bits* e **e** = 32 *bits*) é necessário que o tamanho do número R seja de 33 *bits*(exigência do algoritmo de *Montgomery*). O trabalho de implementação se baseou nos algoritmos de Kalisky, Almost *Montgomery* Inverse e Real Modular Inverse, (KALISKY, 1995) descritos e modificados em (DALY, 2003), (DORMALE, 2004) e (SAVAS & KOÇ, 2000), exposto (Apêndice D) conforme em (SAVAS & KOÇ, 2000).

4.4 Implementações utilizando o Algoritmo de Montgomery

Utilizou-se para estas implementações, o algoritmo proposto por (SAVAS & KOÇ, 2000), descrito da seguinte maneira:

Sejam "**j**" o número de *bits* do expoente "**e**", $\mathbf{e}_i$, o i-ésimo *bit* de "**e**", "**a**", a base da potência que se deseja obter e "**m"** o módulo da multiplicação modular. Define-se Função Exponenciação Modular de Montgomery, à função apresentada na Figura 18.

```
Passo 1: a' = MM(a,R²) = a.R mod m
Passo 2: X' = MM(R²,1) = R.1. mod m
Passo 3 : Para i variando de j-1 até 0
              X' := MM(X'.X')
              Se e_i = 1 então X':= MM(X',a')
Passo 4: Retorno X:= MM(X',1).
```

Figura 18 – Função ME(a,e,m)

4.4.1 Primeira Implementação de Criptografia RSA Utilizando Montgomery

A Figura 19 apresenta a primeira implementação em C de criptografia com o RSA utilizando o Algoritmo de Montgomery, exibido na Figura 18.

```
unsigned long int eleva(int a , int b,unsigned long int m){
  int i, c;
  c = 1;
  for(i=0;i<b;i++)
  {
   c=c*a;
   while (c>m)
   c=c-m;
   }
   return(c);
 }
void criptografando(char*origem, char *destino,int y,int n){
  FILE *arquivo1, arquivo2;
  unsigned long int num,j=0;
  char ch;
  time_t start,end;
  long unsigned t;
  float x,TempoMedio,Tempo=0;
  arquivo1=fopen(origem,"r");
  if (!arquivo1)
  {
     printf ("\nErro na abertura do arquivo.");
     getchar( );
     exit (1);
   }
  else
     printf("\nArquivo aberto com sucesso -> %s",origem)
arquivo2=fopen(destino,"w");
     if (!arquivo2){
  printf ("\nErro na abertura do arquivo.");
  getchar( );
  exit (1);
}
while (!feof(arquivo1)){
             ch=getc(arquivo1);
num=eleva(ch,y,n); //
putw(num,arquivo2);
}
fclose(arquivo1);
fclose(arquivo2);
j++;
}
```

Figura 19 – Primeira Implementação de Criptografia com RSA utilizando Montgomery

Nesta implementação, utilizou-se simplesmente o conceito de se transformar a base **a** da potência para o domínio de *Montgomery*, evidentemente após o pré-calculo do módulo **m**, do elemento **R** que transforma um número inteiro em outro inteiro no

domínio de *Montgomery* e do valor $\mathbf{R^{-1}}$ necessário ao produto modular no domínio de *Montgomery* bem como da função de potenciação (Figura 18).

O resultado da implementação mostrou um desempenho, em termos de tempo de criptografia, quase dez vezes superior aos tempos obtidos quando se utilizou o algoritmo de *Brakley (*Apêndice *C)*, utilizado por (MORENO *et al.*, 2005) e (MUZZI, 2005), que pode ser observado na Figura 20. Pode-se concluir também que os tempos de criptografia (de cada um dos algoritmos) aumentam em proporção um pouco menor ao aumento do número de *bits* utilizado.

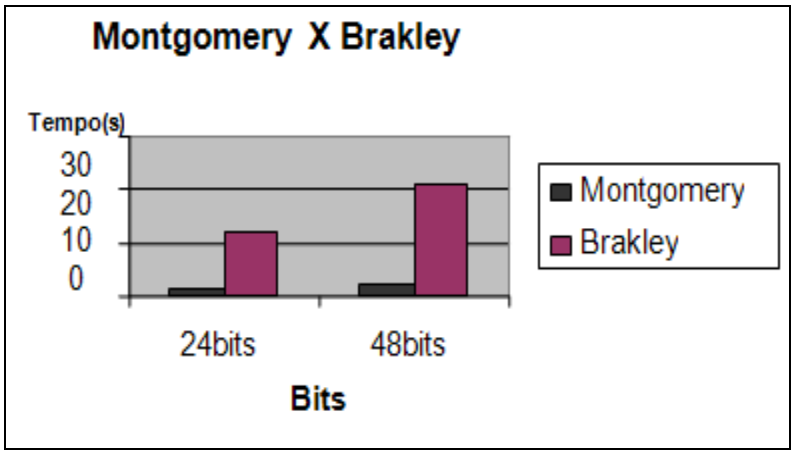

Figura 20 – Comparativo de Velocidade de Criptografia.

4.4.2 Segunda Implementação de Criptografia com RSA em Linguagem C

Nessa segunda implementação utilizou-se o algoritmo otimizador de multiplicações de Gutub (GUTUB, 2000), implementado na Figura 21, para obter uma melhora no desempenho do programa descrito na seção 3.2.1.

```
void criptografando (char
*origem, char
*destino,unsigned long int
e,unsigned long int
n,unsigned long int
m,unsigned long int
Y,unsigned long int
RInv,unsigned long  int R)
{
FILE *arquivo1, *arquivo2;
int num,i,aux, j=0 ;
char ch, echr[MAX];
time_t start,end;
unsigned int Alinha,Z;
float x, TempoMedio=0;
itoa(e,echr,2);
n = strlen(echr);
origem
while(j<3)
{
arquivo1=fopen(origem,"r");
if (!arquivo1)
   {
    printf ("\nErro.Fim");
          getchar( );
          exit (1);
   }
else printf("\nArquivo
aberto -> %s",origem);
arquivo2=fopen(destino,"w");
if (!arquivo2)
   {
    printf ("\nErro.Fim.");
    getchar( );
    exit (1);
   }
else printf("\nArquivo
criado.  -> %s",destino);
while (!feof(arquivo1))
{
  ch=getc(arquivo1);
  Alinha = Gutub(ch,R,m);
  for(i=0;i<n;i++)
   {
    aux = echr[i] - 48;
    Z = Gutub(Y,Y,m);
    Y = Gutub(Z,RInv,m);
 if(aux==1)
 {
  Z=Gutub(Y,Alinha,m);
  Y = Gutub(Z,RInv,m)
 }
 num=Gutub(Y,RInv,m);
 putw(num,arquivo2
}
j++;
fclose(arquivo1);
fclose(arquivo2);
}
}
```

Figura 21 – Função de Criptografia do RSA - Montgomery/Gutub

Os tempos de criptografia obtidos com os programas em linguagem C, seções 4.4.1 e 4.4.2, serão comparados na Tabela 3 e Figura 22.

Tabela 3. Comparativo de desempenho entre Criptografia realizada utilizando os algoritmos de Montgomery, de Montgomery/Gutub e Brakley

	Tamanho de chaves							
Algoritmo	8 bits	16 bits	24 bits	32 bits	40 bits	48 bits	56 bits	64 bits
Montgomery	0,69s	0,95s	1,44s	1,79s	2,11s	2,54s	2,99s	3,13s
Mont + Gutub	1,38s	1,89s	2,84s	32,16s	48,19s	71,62s	101s	143,99s
Brakley	5,44s	7,98s	13,07s	16,24s	19,42s	23,9s	28,33s	29,62s

Tempos em segundos – Arquivos de 1MB

Os resultados obtidos com as implementações são discutidos na seção 4.5.

4.5 Considerações Finais

Como se depreende da Tabela 3 e da Figura 22, inicialmente (chaves de 8 *bits*) o método criptográfico que utiliza o algoritmo de Montgomery tem tempos de criptografia em torno de 8 vezes menor do que o algoritmo que utiliza o algoritmo de Brakley e 2 vezes menor do que o método que utiliza Montgomery associado ao algoritmo de Gutub.

O aumento excessivo nos tempos de criptografia do método Montgomery-Gutub provavelmente se deve ao fato de que o método multiplicativo de Gutub é rápido para multiplicação de dois números inteiros, mas o algoritmo de Montgomery exige multiplicação de três números, o que provoca a cada iteração, duas aplicações do método de Gutub.

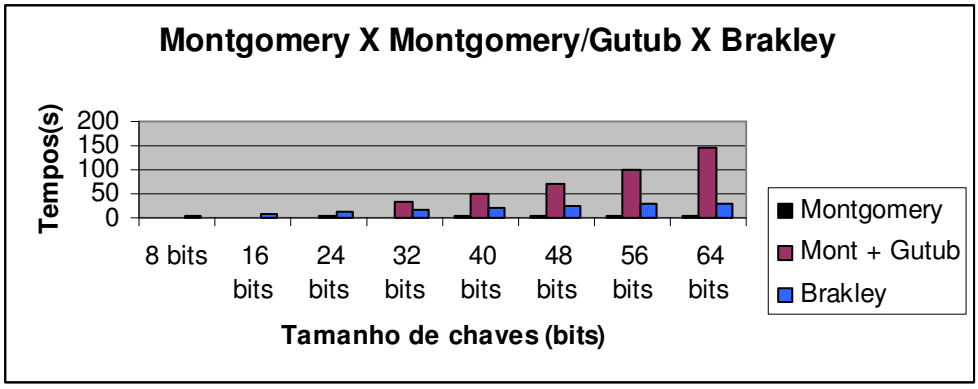

Figura 22 – Comparativo de Velocidades de criptografia.

Conforme pode se depreender da Figura 22, o aumento do tamanho das chaves não provoca aumentos significativos dos tempos de criptografia quando se utiliza o algoritmo de Montgomery, mas provoca aumentos significativos nos programas que utilizam Brakley e Gutub associado ao algoritmo de Montgomery. Provavelmente isso se deva ao fato de que o algoritmo de Montgomery exija a multiplicação de três variáveis em cada iteração e o algoritmo de Gutub é desenhado para efetuar multiplicações de duas variáveis. Já, quando se utiliza o algoritmo de Brakley, se faz, por exemplo, para chaves de 24 *bits*, 32 operações como aquelas da Figura 23 e em cada uma delas se faça uma divisão modular, isso tudo numa única iteração.

```
R = 2 * R + (a.[32-1-i] * b);
R = R%n;
```

Figura 23 – Operações realizadas em cada operação com o algoritmo de Brakley

Entretanto, ao se utilizar o algoritmo de Montgomery faz-se um número de iterações (j iterações onde j é o valor do expoente **e** do RSA) menor do que a chave, já que o tamanho da chave é dado por (**e + m**), onde **e** é o expoente e **m** o módulo. Além disso, **X'** e **a**' são números pequenos, porque em cada iteração é subtraído o valor do módulo **m** do valor **X'.**

Devido a esse desempenho em *Software*, espera-se que em *Hardware*, para chaves de pelo menos 64 bits, se tenha excelente desempenho em termos de velocidade de criptografia, que será descrito no próximo capítulo.

4.5.1 Chaves de mesmo tamanho com diferentes dimensões para “e” e “n” do algoritmo RSA).

Cabe aqui complementar o trabalho realizado por (MORENO *et al.*, 2005), já que se verificou neste trabalho, utilizando sua implementação, que para chaves de mesma dimensão, quando se utiliza diferentes combinações do número de *bits* para o expoente “**e”** e do módulo ”**n"**, obtêm-se diferentes tempos de execução. Por exemplo, para um arquivo de 1 MB e chaves de 48 *bits*, conseguiu-se os seguintes tempos de criptografia, descritos em seguida e na Figura 24.

1. e= 16 *bits*, n = 32 *bits* → tempo = 17,625 s
2. e = 24 *bits*, n = 24 *bits* → tempo = 23,161 s
3. e = 32 *bits*, n = 16 *bits* → tempo = 20,875 s

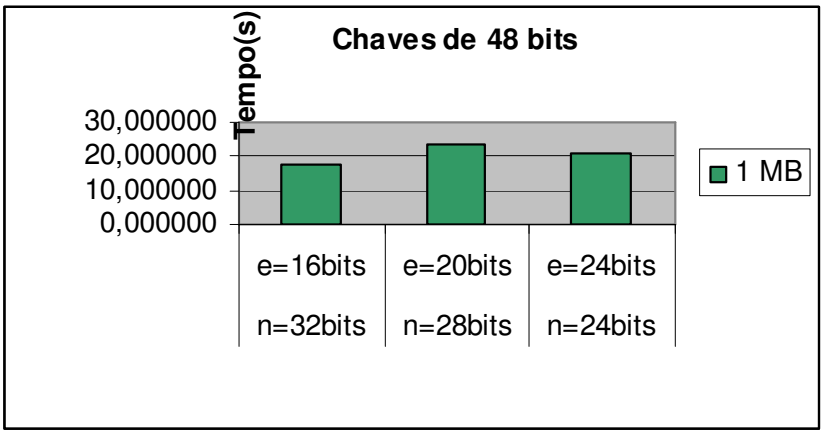

Figura 24 – Comparação de Tempos de Criptografia para Chaves de mesmo tamanho com Módulos de Tamanhos Diferente.

Pode ser observado que, em cada chave, o aumento do número de *bits* de “**e**” (e conseqüente diminuição do número de *bits* de “**n”**) não provoca linearidade nem mesmo qualquer tipo de proporcionalidade na velocidade de criptografia. Não foi

possível analisar neste trabalho as razões pelas quais esse fato ocorre, tendo se limitado exclusivamente à detecção do fato.

Neste capítulo devem-se observar especialmente três fatos, a saber:

1. É necessária uma investigação mais ampla para que se concluam as razões pelas quais, quando se utilizam chaves de mesmo tamanho com tamanhos diferentes para "**e**" e "**n**", se obtenha tempos de criptografia com as aparentes disparidades que pode ser verificada na Figura 24.
2. Embora não se tenha feito neste trabalho comparações com o método utilizado por (MORENO *et al.*, 2005) para computação do número R^{-1}, inverso multiplicativo de R módulo m, o programa em linguagem C desenvolvido neste trabalho para cálculo do referido R^{-1} (Figura 24) é milhares de vezes mais rápido do que aquele. Acredita-se que, utilizando bibliotecas numéricas especiais da linguagem C, seja possível determinar o inverso multiplicativo módulo "**m"** com tamanho maior do que números de 32 *bits*, que seria de extrema utilidade para autores de novos trabalhos de criptografia que utilizem o algoritmo de Montgomery.
3. A implementação em linguagem C da criptografia com o algoritmo matemático RSA que utiliza o algoritmo computacional de Montgomery é muito mais eficiente, em termos de tempo de cifragem, que aquela que utiliza o algoritmo de Brakley.

No próximo capítulo serão examinadas implementações em VHDL realizadas neste trabalho.

CAPÍTULO 5. IMPLEMENTAÇÕES DE ALGORITMOS CRIPTOGRÁFICOS EM VHDL

São apresentadas nesta seção três implementações em VHDL que cifram caracteres com o algoritmo criptográfico RSA, utilizando os métodos multiplicativos de Montgomery e de Gutub. São apresentados ainda os algoritmos que produziram essas implementações bem como comentários e conclusões acerca dessas implementações.

5.1 Implementação em VHDL utilizando o Algoritmo de Montgomery para Criptografia com o Algoritmo RSA

A Figura 25 mostra o algoritmo Radix-2 utilizado para efetuar as exponenciações no domínio de *Montgomery*, (PRINCE, 2002), (GAUBATZ, 2002) e (AMANOR, 2005), conforme exposto em (AMANOR, 2005). Sejam X = (x_{m-1} , ..., x_1 , $x_{o)}$ e Y, dois operandos multiplicativos no domínio de *Montgomery (* Figura 6*)*, isto é, foram obtidos através do produto X = A.R mod m e Y = B.R mod m, A e B, sendo dois números inteiros e R, relativamente primo com n (o máximo divisor comum de m e R é 1), uma potência de dois com o mesmo número de *bits* de n (R>m).

```
So = 0;
Para i variando de 0 até n-1
  Se (Si + Xi.Y) é par
  {
  Se Si+1 :=  (Si + Xi.Y)/2  é par, então
    Si+1 :=  (Si + Xi.Y)/2;
  Senão
    Si+1 :=  (Si + Xi.Y + m)/2;
  }
Se Sm ≥ M então
  Sm:= Sm - m
```

Figura 25 – Algoritmo Radix 2 de Montgomery (AMANOR, 2005)

Segundo (AMANOR, 2005), este algoritmo é adequado para implementação em *Hardware* porque é composto de operações simples (multiplicação de número inteiro por número binário, *bit* a *bit*) e deslocamento simples à direita (nas divisões por dois). O teste da condição de paridade também é bem simples de implementar, pois basta checar o *bit* menos significativo das somas parciais S para decidir se o módulo será somado à soma parcial S (no caso de não paridade). Observe-se (no exemplo abaixo) que a adição do módulo não altera a congruência módulo m.

Por exemplo, 20 mod 7 = 6 (20 = 2.7 + 6). Verifique-se que 20 + 7 = 27 não altera a congruência módulo 7, pois 27 mod 7 = 6 (27 = 3.7 + 6).

Este algoritmo computa $S = X.Y.2^{-n} \bmod m$. A idéia de *Montgomery* (AMANOR, 2005) foi manter os tamanhos dos resultados intermediários menores do que n + 1 *bits*. Isto é atingido ao se intercalar a computação das somas parciais (Si ou $Si+_1$ e adição de novos produtos parciais (Xi.Y), com a divisão por 2. Cada uma das divisões reduz o tamanho do número para n *bits*. A Figura 26 contém um trecho de programa em VHDL que efetua a multiplicação de Montgomery:

```
for j in 0 to 'n' loop
    if X(j) = '0' then
      P:=SHR((P+Y),1);
    else
      P:= SHR((P+Y+M),1);
    end if;
  end loop;

if P>=M then

P:=P-M;
Sendo P, inicialmente um número igual a zero.
```

Figura 26 – Trecho de Programa em VHDL que efetua a Multiplicação de Montgomery

O programa em VHDL utilizando o código da forma como acaba de ser exposta ainda é muito lento. Existem algoritmos otimizados de Montgomery, como o (*Fast Montgomery Algorithm*) e o algoritmo de Gutub que serão expostos nas próximas seções.

5.2 Implementação de Criptografia com o Algoritmo RSA Utilizando o Método de Gutub Associado ao Método de Quadratura e Multiplicação.

Noções de potenciação podem ser obtidas ao se utilizar um algoritmo ingênuo. Para se obter uma nova potência, calcula-se cada uma das potências, a partir da potência 2, multiplicando-se o resultado pelo primeiro multiplicador. Por exemplo, para se calcular T^{19}, pode-se utilizar a potenciação da seguinte maneira:

$T \rightarrow T^2 \rightarrow T^3 \rightarrow T^4 \rightarrow$.....-$\rightarrow T^{19}$.

Seriam necessárias 18 multiplicações. Uma forma de se minimizar a quantidade de multiplicações a serem feitas é calcular:

$T.T = T^2$; $T^2.T^2 = T^4$; $T^4 . T^4 = T^8$; $T^8 . T = T^9$; $T^9 . T^9 = T^{18}$; $T.T^{18} = T^{19}$.

$T \rightarrow T^2 \rightarrow T^4 \rightarrow T^8 \rightarrow T^9 \rightarrow T^{18} \rightarrow T^{19}$.

Neste último caso, foram necessárias 6 multiplicações. Segundo (KOÇ, 1994) o algoritmo da Figura 27 permite calcular cada potência com o número mínimo de operações de multiplicação modular.

```
Entradas T,e, m.
Saída: Te mod m
Se ek-1  = 1, então
  C:= T,
Senão
  C:= 1;
Para i variando de k-2 até zero,
  2a . C:= C.C mod m
  2b . Se ei = 1 então
    C:= C.M mod m
Retorno: C
```

Figura 27 – Método binário de Quadratura e Multiplicação.

A segunda implementação (programa completo no Apêndice E) utiliza o método de (GUTUB, 2000) (Figura 4) para efetuar as multiplicações e o algoritmo denominado Método de Quadratura e Multiplicação (Figura 27) que remonta à antiguidade (KOÇ, 1994) e é utilizado para se efetuar um número mínimo de operações para se obter a potência desejada.

A Figura 28 mostra uma máquina de estados para criptografia com o RSA utilizando o algoritmo de Gutub associado ao método de Quadratura e Multiplicação

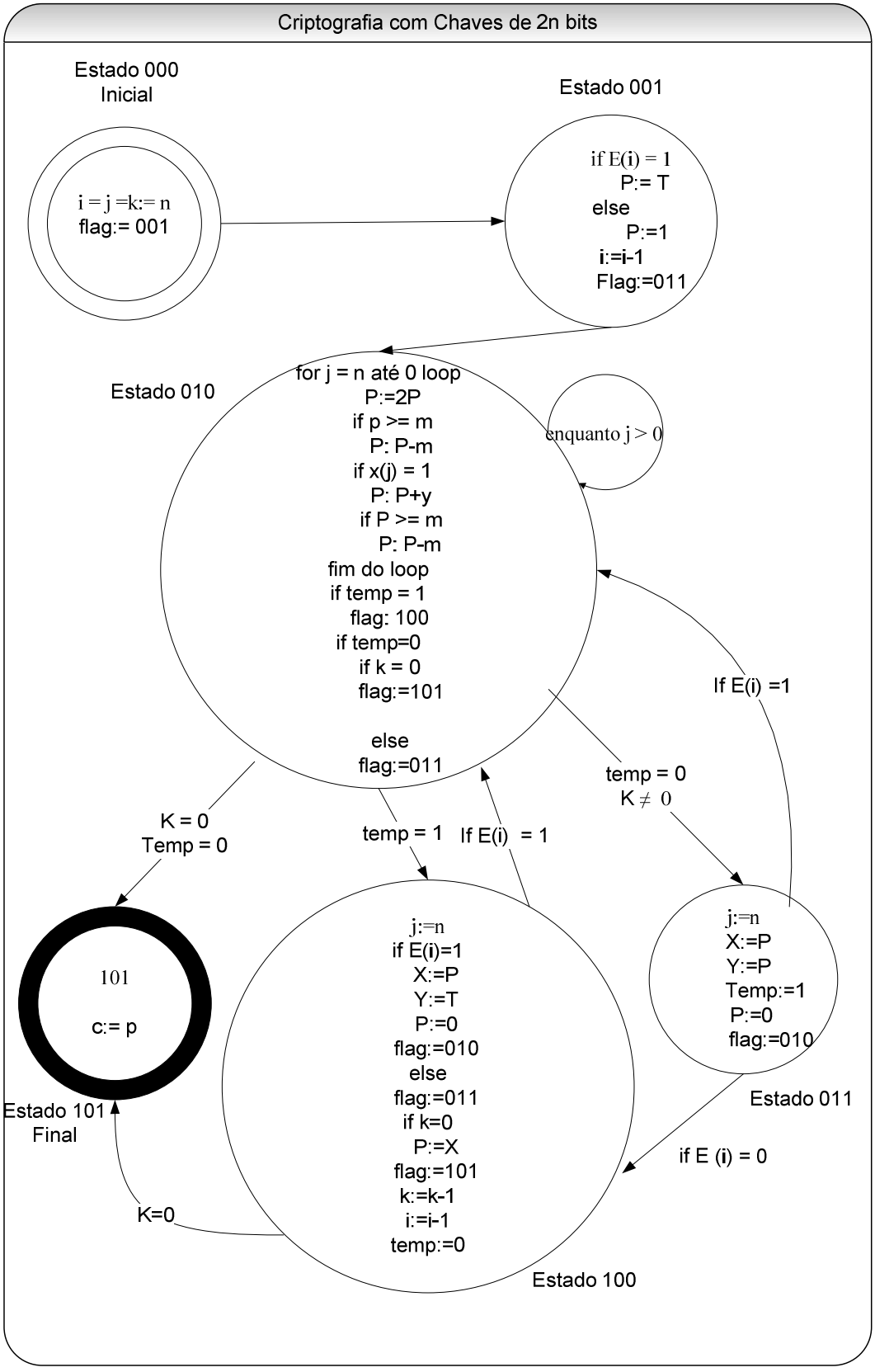

Figura 28 - Máquina de Estados de Gutub/ Quadratura e Multiplicação.

Com relação à Figura 28, deve ser observado que:

No estado 000 são feitas as inicializações

No estado 001, decide-se, de acordo com o método de Quadratura e multiplicação, se o bit mais significativo da base da potência é 1.

No estado 010 é aplicado o método de (Gutub, 2000) para multiplicação modular.

Nos estados 011 e 100 utiliza-se o algoritmo de quadratura e multiplicação que comanda, após ser verificado se o bit que vai iniciar nova iteração é zero ou um, a "chamada" do estado 010.

No estado 101 atribui-se o valor do caractere criptografado e encerra-se programa.

5.3 Arquitetura da Implementação com o Método de Gutub associado ao Método de quadratura e Multiplicação

A Figura 29 mostra uma macro-Arquitetura da Implementação em VHDL de criptografia com o algoritmo matemático RSA utilizando o algoritmo computacional de Gutub (Figura 4) associado ao Método de Quadratura e Multiplicação (Figura 27).

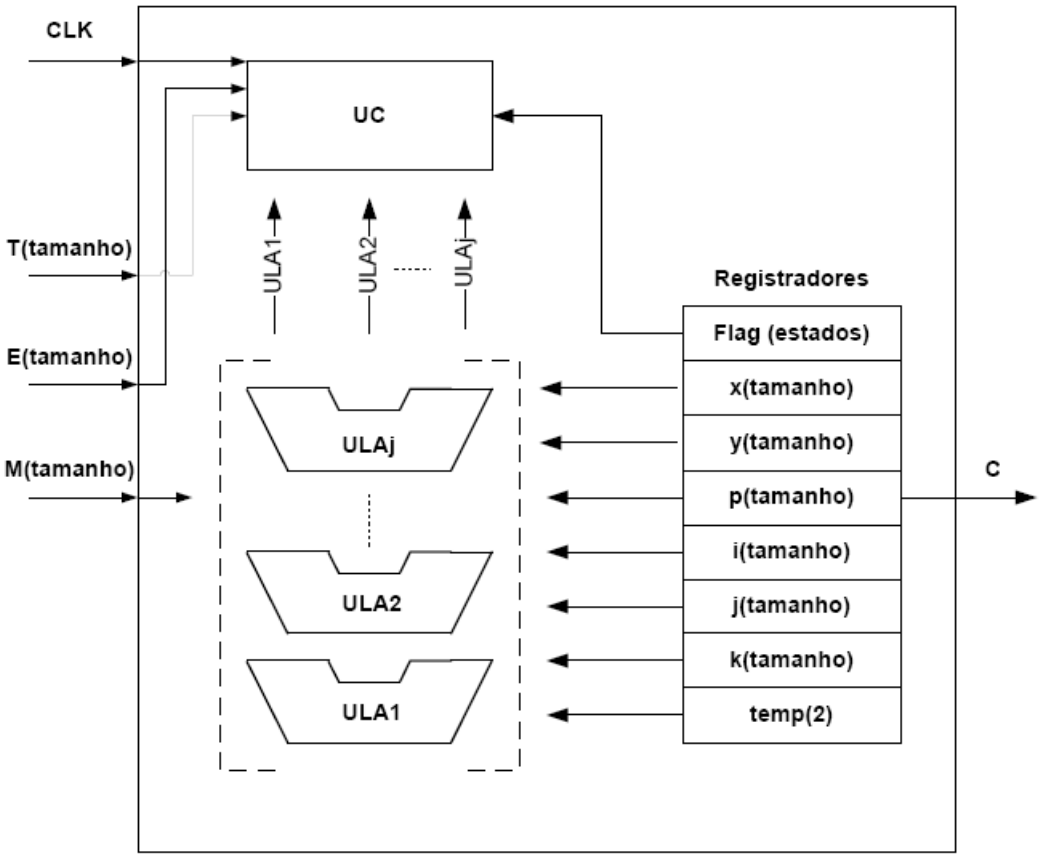

Figura 29 - Arquitetura da Implementação do algoritmo Gutub em VHDL.

A seguir uma representação dos elementos da Arquitetura mostrada na Figura 30.

T: Porta de entrada que recebe o valor **T** a ser criptografado.

E: Porta de entrada que recebe o valor **e** do RSA.

M: Porta de entrada que recebe o módulo **m** do RSA.

Clk: Clock

C: Porta de saída que recebe o dado criptografado.

Flag: Variável que controla a passagem de um estado para outro.

P: Variável que recebe o valor criptografado temporário em cada iteração.

X: Variável que recebe o valor temporário de **P** antes de iniciar cada iteração.

Y: Variável que recebe o valor temporário de **P** antes de iniciar cada interação ou o valor da entrada **T** antes de iniciar nova interação.

i,k: variáveis que decrementam o valor binário de **E**, *bit* a *bit*.

j: Variável que controla o *loop* em cada interação.

Temp: variável que desvia condicionalmente o estado 2 para os estados 3, 4 ou 5.

Observação 1: Tamanho é a metade do tamanho das chaves, ou seja, por exemplo, numa chave de 32 *bits* tamanho é igual a 16.

Observação 2: O processamento ocorre no estado 2, onde são efetuadas as operações de multiplicação por 2 e subseqüente subtração do módulo para manter os valores envolvidos menores do que o módulo.

5.4 Implementação de Criptografia com o Algoritmo RSA Utilizando o Método Rápido de Multiplicação Modular de Montgomery.

Esta terceira implementação (Apêndice F) utiliza o método Rápido de Montgomery para efetuar as multiplicações Modulares.

A Figura 30 mostra uma máquina de estados para criptografia com o RSA utilizando o algoritmo Rápido de Montgomery.

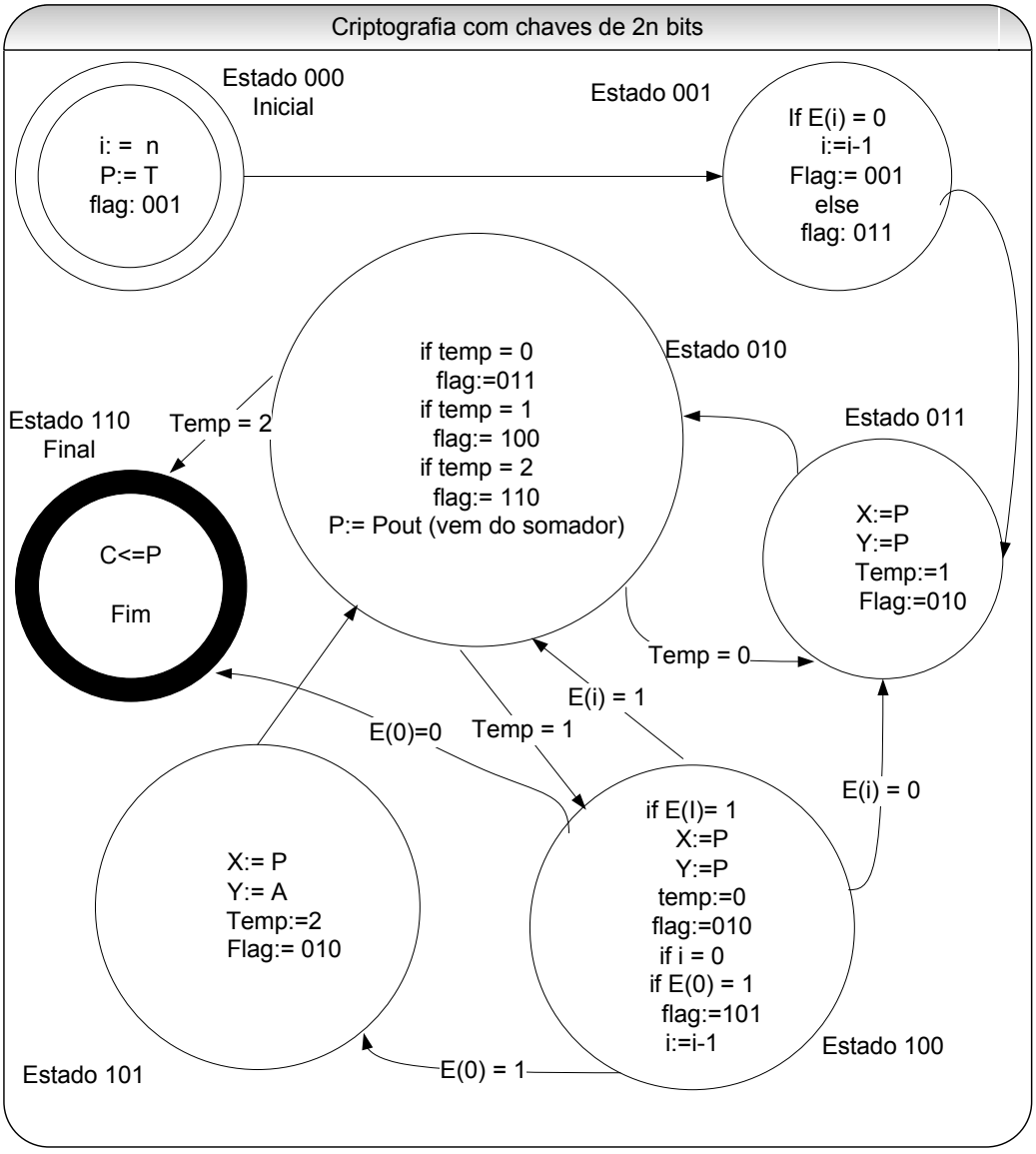

Figura 30 - Máquina de Estados – programa com o algoritmo Rápido de Montgomery.

Com relação à Figura 30, devem ser feitas as seguintes considerações:

No estado 000 são feitas as inicializações.

O estado 001 chama o estado 011 decrementa **i** (contador dos *bits* de E) até localizar o primeiro *bit* significativo de E.

No estado 010 recebe o valor do somador e decide se chama os estados 011, 100 ou 110, de conformidade com o algoritmo de Montgomery.

Os estados 011, 100 ou 110, atribuem os valores às X e Y, de acordo com o algoritmo de Montgomery.

O Estado 110 finaliza o programa e atribui à Porta de Saída **C** o valor criptografado pelo RSA.

5.5 Arquitetura de Implementação com o Método de Montgomery

A Figura 31 mostra uma macro-Arquitetura da Implementação em VHDL de criptografia (Apêndice F) com o algoritmo matemático RSA utilizando o algoritmo Rápido de Montgomery (Figura 30).

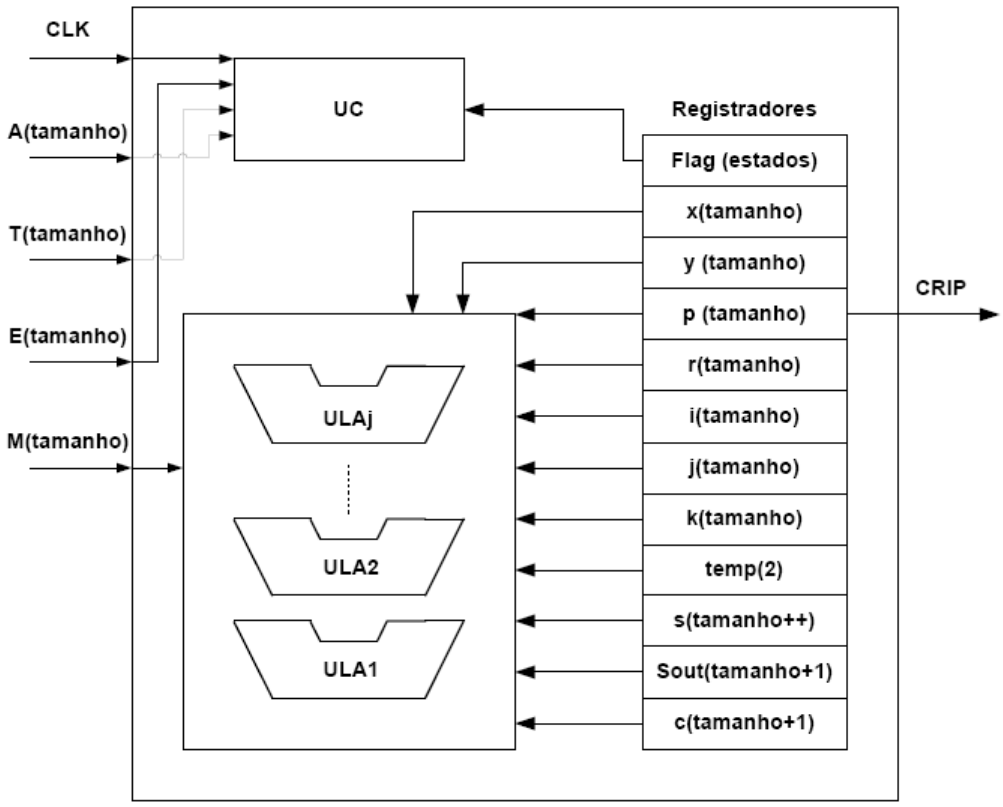

Figura 31 - Arquitetura da Implementação do algoritmo Fast Montgomery em VHDL.

Na Figura 31 se vê a Arquitetura da Implementação do algoritmo Montgomery, que foi descrita em VHDL(Apêndice F), e nela se observam os seguintes detalhes:

A: Porta de entrada – Recebe a imagem, no domínio de Montgomery , do número a ser criptografado. A = a.R mod m, onde a é o número a ser criptografado.

T: Porta de entrada: Recebe a imagem de 1 no domínio de Montgomery (T = 1.R mod m)

E: Porta de Entrada: Recebe o valor do expoente "**e"** do RSA.

M: Porta de entrada – Recebe o valor do módulo **m**.

Clk: clock

Crip: Porta de saída – Vai receber o valor criptografado no domínio de Montgomery. O valor criptografado no domínio dos inteiros será dado por C = Crip.R mod m.

P: Variável que recebe o valor temporário do dado criptografado e na última interação receberá o valor criptografado que será atribuído à variável Crip.

X e Y: Variáveis que receberão os valores envolvidos na criptografia que estão no *loop* que realiza as operações de criptografia.,

S: Soma de Xj.Y em cada interação – Somador

C: juntamente com S forma a arquitetura do Carry-Save-Adder.

R: Variável que armazenará o valor de Y+M durante o processamento de dados.

Sout: Armazena o valor temporário de S durante o processamento.

J,k: Variáveis que controlam o decremento de cada *bit* de E.

Temp: Variável que controla as passagens do estado 3 para o estado 4 ou estado 5.

Obs: Tamanho é a metade do tamanho das chaves, ou seja, por exemplo, numa chave de 32 *bits*, tamanho é igual a 16.

Observação: O conjunto ULA1, ULA2, ..., ULAj é um componente em VHDL(módulo independente) e provocado pelo comando **FOR**.

Os desempenhos entre as implementações em VHDL utilizando os Algoritmos de Gutub/Método de quadratura e Multiplicação e os Algoritmo de Gutub serão examinados no item 0

5.6 Comparativo Entre os Desempenhos dos Algoritmos Implementados

Utilizando-se uma placa FPGA XC4VFX12 modelo 12FF668 em computadores Pentium 4 com 512 MB de memória RAM obteve-se com a ferramenta XST da versão 8.2 do *Xilinx* os desempenhos listados nas Tabela 4 e Tabela 5. É interessante acrescentar que essa versão mais moderna do *Xilinx* recomenda que se utilize computadores com pelo menos 2GB de memória RAM. Provavelmente por essa razão não se conseguiu criptografar com chaves superiores a 128 *bits* (esperava-se obter-se criptografia com chaves de pelo menos 512 *bits*).

Tabela 4. Tempos de Criptografia – Algoritmo de Gutub.

	Slices	Flip-Flops	Luts(4)	*TMP(ns).	Freq.Max
16bits	386 = 7%	52 = 0%	727 = 6%	55.324	18.378
32 bits	1342= 24%	157 =1%	2582 = 23%	118.252	8.530
64 bits	4992 =91%	335 = 3%	9743 = 89%	272.347	3.688
128bits	19322= 353%	792 = 7%	38020= 347%	709.647	1.412

*Tempo Máximo de Propagação

Tabela 5. Tempos de Criptografia – Algoritmo de Montgomery.

	Slices (5472)	Flip-Flop (10944)	Luts(4) (10944)	TPM*(ns).	Freq.Maxima
16 *bits*	249= 4%	70 = 0%	481 = 4%	9.359	106.211
32 bits	801 =14%	178 =1%	1557 =14%	18.944	52.943
64 bits	2979 = 54%	415 =3%	5823 =53%	44.334	22.915
128bits	11225 =205%	912 = 8%	22082 201%	84.194	12.039

*Tempo Máximo de Propagação

Dos resultados das Tabela 4 e Tabela 5 conclui-se que dos algoritmos utilizados, embora o algoritmo de Gutub tenha sido otimizado pela sua associação com o método de quadratura e multiplicação, ainda assim mostrou uma maior ocupação de área, assim como um desempenho em tempos de velocidade de criptografia muito inferior ao programa que utilizou o algoritmo de Montgomery.

O desempenho inferior do programa que utiliza o algoritmo de Gutub se deve ao fato de que ele é adequado para efetuar multiplicações de dois números enquanto que o algoritmo de Montgomery exige a multiplicação de três números.

5.6.1 Comparação entre os Desempenhos em Linguagem C e em VHDL da Criptografia que Utiliza o Algoritmo de Montgomery

Examinando a simulação de criptografia utilizando o algoritmo Rápido de Montgomery (Figura 32) fornecida pelo simulador do *Xilinx* 3.1, com placa Spartan 2,

verifica-se que são necessários 50 ciclos para se criptografar um caractere com chave de 16 *bits* **N =** 50. O tempo máximo total de cifragem se obtém por **T = N* tempo máximo de propagação.**

Assim, o total de ciclos seria dado no pior caso, para e = 255 $(FF)_{Hex}$ (nenhum zero binário) T = 50 * 9,359 ns(Tabela 5) = 467,95 ns

Para que se determine a velocidade de criptografia em MB/s basta fazer:

```
16 bits ----------------467,95 ns
1000000bits----------          X
```

X = 29.246.875 ns = 0,029.246.875 s

```
1 MB ------- 0,029.246.875  s
  X-----------1s
```

X = 34,19 MB/s → O programa criptografa 34,19 MB em um segundo.

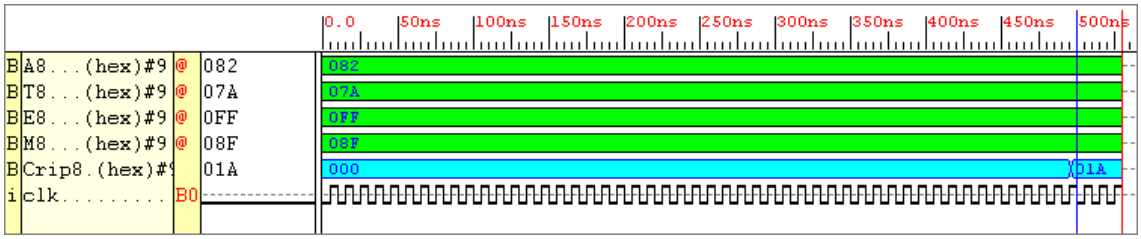

Figura 32 - Simulação de criptografia para chaves de 16 bits.

De forma análoga, concluiu-se que para chaves de 32 e 64 bits, o programa criptografa, respectivamente, com velocidades de 17,23 MB/s e 7,44 MB/s.

Verifica-se também que o programa para criptografia com o RSA utilizando o algoritmo de Montgomery em linguagem C criptografa 1 MB em 0,95 s para chaves de 16 *bits*, em 1,79 s para chaves de 32 *bits* e em 3, 13 s para chaves de 64 *bits* (Tabela 3).

Isso leva a velocidades de criptografia de 1,052 MB/s, 0, 558 MB/s e 0, 319 MB/s, respectivamente.

Na Figura 33 é apresentado um comparativo de desempenho entre as implementações em *Hardware* e em *software* para criptografia com o RSA utilizando o algoritmo de Montgomery

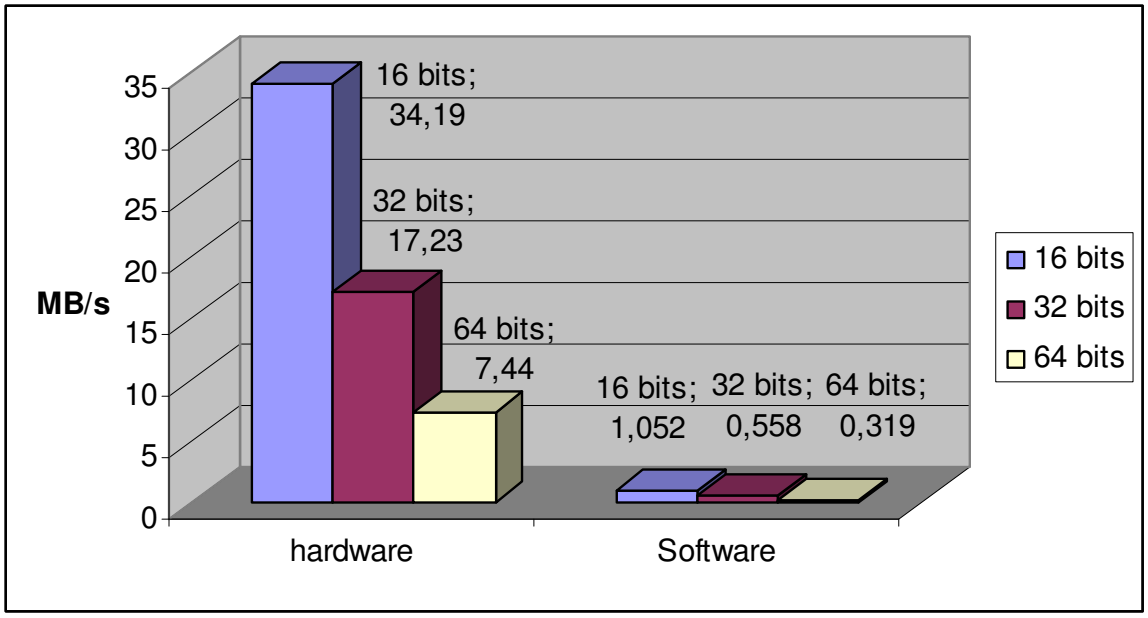

Figura 33 – Comparação de desempenho em hardware e software.

Embora, teoricamente uma implementação em *hardware* seja sempre mais rápida do que a mesma implementação em *software,* cabe neste trabalho, quantificar as diferenças de velocidade de criptografia. Observa-se na Figura 33 que o programa em VHDL criptografa aproximadamente 32,5 vezes mais rápido com chaves de 16 *bits* (34,19(MB/s)/1,052 (MB/s)), 30 vezes mais rápido (17,23 (MB/s)/0,558(MB/s)) com chaves de 32 *bits* e 23 vezes mais rápido com chaves de 64 *bits* (7,44 (MB/s)/0,319 (MB/s), do que a respectiva implementação em linguagem C.

5.7 Comparação entre os Algoritmos de Brakley e de Montgomery para que Utiliza o Algoritmo Criptografia Criptográfico RSA em FPGAs.

Verificou-se para chaves de 16 *bits* que, utilizando o algoritmo de Brakley para o RSA (MUZZI, 2005) com a versão 3.1 do Xilinx e placa Spartan 2, necessita-se de 527 ciclos para um tempo de demora máxima no circuito de 4,914 ns e que para arquivos de 32 MB o número de ciclos sobe para 1826 ciclos. Naquele caso (chaves de 16 bits) o tempo de criptografia é dado por T = 527*4,914ns = 2.589ns e neste caso, por T = 1.826*7,309 = 13.346 ns. Utilizando-se a mesma versão do *Xilinx* e a mesma placa o método de Montgomery proposto neste trabalho utiliza 93 ciclos com tempo de demora máximo no circuito, de 3,354 ns, para chaves de 16 *bits* que fornece um tempo de criptografia T = 322 ns, em torno de 8 vezes menor que o utilizado com Brakley (Tabela 6). O número de ciclos para chaves de 32 *bits* quando se utiliza o algoritmo de Montgomery é 185 e o tempo de demora máxima no circuito é 5,032 ns, o que fornece um tempo máximo de criptografia T = 931 ns. Neste caso, o programa com o algoritmo de Montgomery criptografa 14 vezes mais rápido (Tabela 6).

Tabela 6. Comparativo de tempos de criptografia .

	16 bits	32 bits
Montgomery	322 ns	931 ns
Brakley	2.589ns	13.346 ns

CAPÍTULO 6. CONCLUSÕES FINAIS

Este trabalho objetivou o estudo de algoritmos computacionais que pudessem acelerar os tempos de criptografia com o algoritmo RSA, em especial o algoritmo de Montgomery. Estudaram-se diversos algoritmos multiplicativos e exponenciais propostos por (KOÇ, 1994) e (DALY, 2003), com exemplos práticos ilustrativos. Estudou-se também, alternativamente, o método de Gutub para multiplicação modular binária, também aplicado ao RSA. Estes algoritmos foram implementados em linguagem de programação C e em linguagem de descrição de *Hardware* VHDL, com objetivo de compará-los com implementações em *Software* e em *Hardware* implementadas por (MUZZI, 2005) que utilizava o algoritmo multiplicativo de Brakley para cifragem de mensagens.

Havia inicialmente intenção de se criptografar com o algoritmo criptográfico RSA, utilizando o algoritmo computacional de Montgomery, com chaves de pelo menos 512 *bits* para se poder fazer uma comparação com o trabalho de (MUZZI, 2005), que utilizou o algoritmo de Brakley. Entretanto, isso não foi possível, em razão de que o trabalho de (MUZZI, 2005) tinha preocupação principalmente quanto à utilização da menor área possível da placa FPGA, sem maiores considerações com os tempos de criptografia, enquanto que a meta deste trabalho tem foco especial na velocidade de criptografia e então ocorreu problemas quanto à ocupação da placa. Em verdade, o que se obteve foi um tempo de criptografia muito superior aos tempos obtidos com o algoritmo de Brakley, utilizado por (MUZZI, 2005), testado neste trabalho, com

resultados na Figura 20 e mesmo bastante superior ao programa desenvolvido em VHDL que utiliza o algoritmo de Gutub associado ao método de quadratura e multiplicação, que otimiza o algoritmo de Gutub.

Contudo, foi possível se verificar que, como se suspeitava, o algoritmo RSA tem desempenho em torno de 10 vezes superior em termos de velocidade de criptografia, quando se utiliza o algoritmo de Montgomery, em vez do algoritmo multiplicativo de Brakley, tanto em *software* quanto em *Hardware* (seções 0 e 0).

Criptografar com chaves muito grandes é uma dificuldade que ainda não se conseguiu superar, pelo menos a baixo custo, utilizando-se de computadores comuns.

Segundo (HANS, 2004) pode-se criptografar até 2048 *bits* utilizando-se de uma Máquina de Dupla Emissão (*Dual-Issue Machine*) de dois processadores, um aritmético e o outro, controlador, que consiste numa arquitetura VLIW (*Very Long Instruction Word*) que executa um grande conjunto de operações concorrentemente.

Literaturas como (AMANOR, 2005), (BAILEY *et al.*, 1999), (Daly, 2003), (DATA & RAKESNAKE, 2002), (DORMALE, 2004), (GUTUB, 2000) e muitas outras escrevem acerca das dificuldades para se criptografar com chaves de tamanho muito grande e utilizam o algoritmo de Montgomery para melhorar os tempos de criptografia com o RSA. Os exemplos dos dois parágrafos acima, além de reiterar a existência das dificuldades que existem para se obter segurança com a utilização de chaves assimétricas para criptografia, propõem novas arquiteturas para resolver o problema.

O objetivo deste trabalho residiu em examinar diversos algoritmos computacionais multiplicativos e exponenciais, especialmente aqueles que utilizam o método de multiplicação de *Montgomery* e sua aplicação em criptografia com o algoritmo criptográfico RSA. Isso foi realizado e embora não se tenha criptografado

com chaves superiores as a 64 bits, isso se deveu simplesmente a limitações com o Hardware utilizado (computadores com 512 MB de memória RAM). Como resultado adicional fica neste trabalho um estudo bem detalhado da utilização dos algoritmos de Euclides, Montgomery, Gutub e RSA.

Enfim, a criptografia segura com chaves assimétricas permanece ainda no estado de tricotomia constante dentro da computação, ou as soluções são rápidas e caras ou são lentas e baratas ou descobre-se um algoritmo que torne essas soluções ágeis e não dispendiosas. A criptografia aguarda.

6.1 Sugestões para Trabalhos Futuros

A seguir uma lista de idéias visando dar continuidade ao trabalho aqui realizado:

Criar e propor novas arquiteturas que consigam ter uma boa utilização de recursos das FPGAs, diminuindo a ocupação das placas sem diminuir a velocidade (minimizando o tempo de propagação no circuito), para o algoritmo de Montgomery.

Pesquisar um algoritmo computacional que melhore a velocidade do algoritmo de Montgomery.

Tentar diminuir o tempo de propagação no circuito para o algoritmo de Gutub/Método de quadratura e multiplicação, já que ele utiliza menos ciclos para realizar a criptografia.

Associar o algoritmo de multiplicação e quadratura ao algoritmo de Montgomery, da mesma forma como foi associado o algoritmo de Gutub ao algoritmo de Montgomery neste trabalho.

Realizar otimizações nas descrições em linguagem VHDL realizadas, visando obter mais velocidade e menos recursos espaciais (físicos) das FPGAs.

Tentar criar e propor arquiteturas especiais que permitam implementar esses algoritmos em FPGAs para um número maior de chaves. Uma dessas arquiteturas poderia ser a VLIW.

REFERÊNCIAS BIBLIOGRÁFICAS

(AMANOR, 2005) Amanor, D.N., Dissertação de Mestrado, The University of Applied Sciences Offenburg, Germany, Supervisores da dissertação: Prof. Dr. Angelika Erhardt e Prof. Dr. Christof Paar.

(BAILEY *et al.*, 1999) Bailey DV, Cammack W, Guajardo J, Paar C; Cryptography in Modern Communication Systems; Texas Instruments DSPS FEST, Houston, TX, Agosto, 1999.

(CHIARAMONTE, 2006) Chiaramonte R. B.; SICO: Um Sistema de Comunicação de Dados com Suporte Dinâmico à Segurança; Dissertação de Mestrado, UNIVEM, Marília, 2006.

(DALY, 2003) Daly Alan, Marname L., Popovici E., Fast Modular Inversion in The Montgomery Domain on Reconfigurable Logic – ISSC, julho de 2003.

(DALY, 2002) Daly A.; Marname W; Efficient Architectures for implementing Montgomery Modular mulltiplication and RSA Modular Exponentiation on Reconfigurable Logic; ACM, 2002.

(DATA & RATTLESNAKE, 2002) Data, S.A.V.A.; Rattlesnake, J.H.A.; RSA Encryption Algorithm in a Nutshell; Neworder Bol St, 2002.

(DIFFIE-HELLMAN, 1977) W. Diffie and M.E. Hellman, Exhaustive cryptanalysis of the NBS Data Encryption Standard; Computer 10, 1977.

(DORMALE, 2004) Dormale, G.M., Bulens, P., Quisquater, J-J., An Improved Montgomery Modular Inversion Targeted for Efficient Implementation on FPGA; International Conference on Field-Programmable Technology, 2004.

(GAUBATZ, 2002) Gaubatz, Gunnar - A Thesis Submitted to the Faculty of the Worcester Polytechnic Institute, grau de mestre, orientadores: Dr. Berk Sunar e Dr. Fred J. Looft, abril de 2002.

(GUTUB, 2000) A. Gutub. Modulo Multiplication *Hardware* Design; Oregon State University, EGE 575, 2000.

(HANS, 2004) EBERLE H., GURA N., SHANTZ S.C, GUPTA V., RARICK L, SUNDARAN S, A Public-Key Cryptographic Process for RSA and ECC, 15º Conferencia Internacional IEEE para Sistemas de Aplicações Específicas, Arquiteturas e Processadores, 2004.

(KHALDOON, 2002) Khaldoon M., Prototyping Of Scalable Montgomery Multiplier Using Field Programmable Gate Arrays (FPGAs); M.S. Thesis, Department of Electrical & Computer Engineering, Oregon State University, July 23, 2002.

(KALISKY, 1995) Kalisy Jr, B.S., The Montgomery Inverse and its Applications; IEEE, 44(8), pg.1064-1065, Agosto de 1995.

(KNUTH, 1997) Knuth, E. The Art of Computer Programming - Semi numerical Algorithms; Addison-Wesley, 1997.

(KOÇ, 1996) Koç Ç. K.; Acar T; Kalisky Jr, B.S.. – Analyzing and Comparing Montgomery Multiplication Algorithms; IEEE Micro, v.16 n.3, p.26-33, June 1996

(KOÇ, 1994) Koç, Ç, K.; Montgomery Reduction With Even Modulus; IEEE Proceedings: Computers and Digital Techniques – Setembro de 1994

(MUZZI, 2005) Muzzi, F. A. G., O Padrão de Segurança PKCS#11 em FPGA's – RSA um Estudo de Caso; Dissertação de Mestrado, UNIVEM, Marília, 2006.

(MCTAGGART, 2001) Mctaggart M., Introduction to cryptography, Part 3: Asymmetric cryptography, 01.03.2001, página da IBM, Internet, abril de 2006

(MORENO *et al.*, 2005) E.D.Moreno; Pereira F.D.; Chiaramonte R.B.; Criptografia em *Software* e *hardware*; Livro pela Editora Novatec, pg. 288 – 2005

(PRINCE, 2002) Prince, B.J., Scalable Montgomery Multiplication Algorithm; Oregon State University, Department of Electrical & Computer Engineering, 2002.

(RIVEST, 1978) Rivest, R.; Shamir, A.; Adleman, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems; Communications of ACM, 1978.

(SAVAS & KOÇ, 2000) Savas E,; Koç, Ç.K. The Montgomery Modular Inverse – Revisited; IEEE, Special issue on computer arithmetic, Vol.49, nº 7, julho de 2000, pg 763-766.

(SAVAS & KOÇ, 2000) S. C. Shantz S. C. Shantz, "From Euclid's GCD to Montgomery multiplication to the great divide," Tech. Rep. TR-2001-95, Sun Microsystems Laboratories, Santa Clara, Calif, USA, June 2001.

(TODOROV & TENCA, 2000) Todorov G.; Tenca A.F.; ASIC Design, Implementations and Analysis of A Scalable High-Radix Montgomery Multiplier – M.S.Thesis - Dezembro – 2000

APÊNDICE A – Algoritmo gerador de números aleatórios.

Este gerador de números aleatórios (MORENO *et al.*, 2005) é utilizado para gerar os números p, q e d, citados. Para a criação das chaves é necessário um gerador de números aleatórios para a escolha dos números p, q e d descritos nas etapas 1 e 3. O gerador de números aleatórios utilizado na implementação foi inspirado no apresentado no livro de Knuth (KNUTH, 1997), seu esquema é apresentado na Figura 34.

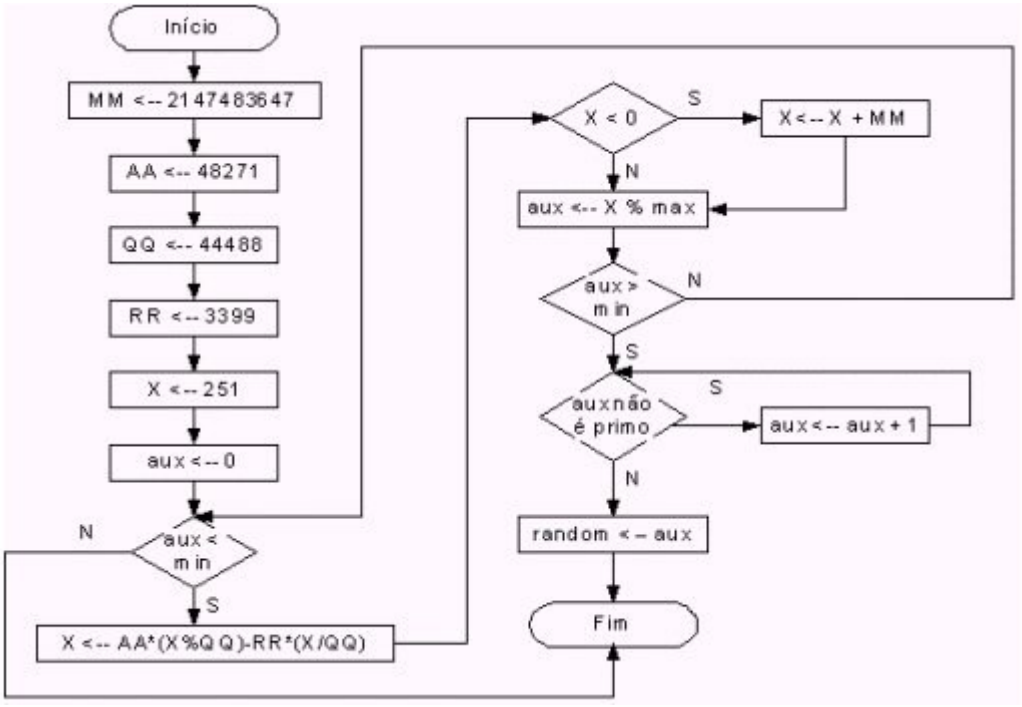

Figura 34 - Esquema do Gerador de Números Aleatórios.

Os valores atribuídos à MM, AA, QQ e RR são definidos para que os números gerados não se repitam facilmente, o valor X é o valor a partir do qual será calculado o próximo número.

APÊNDICE B – Algoritmo de Euclides Estendido.

Outro algoritmo necessário é o algoritmo de Euclides Estendido que é utilizado para o calculo de **e** na etapa 4, seu esquema é apresentado na Figura 35.

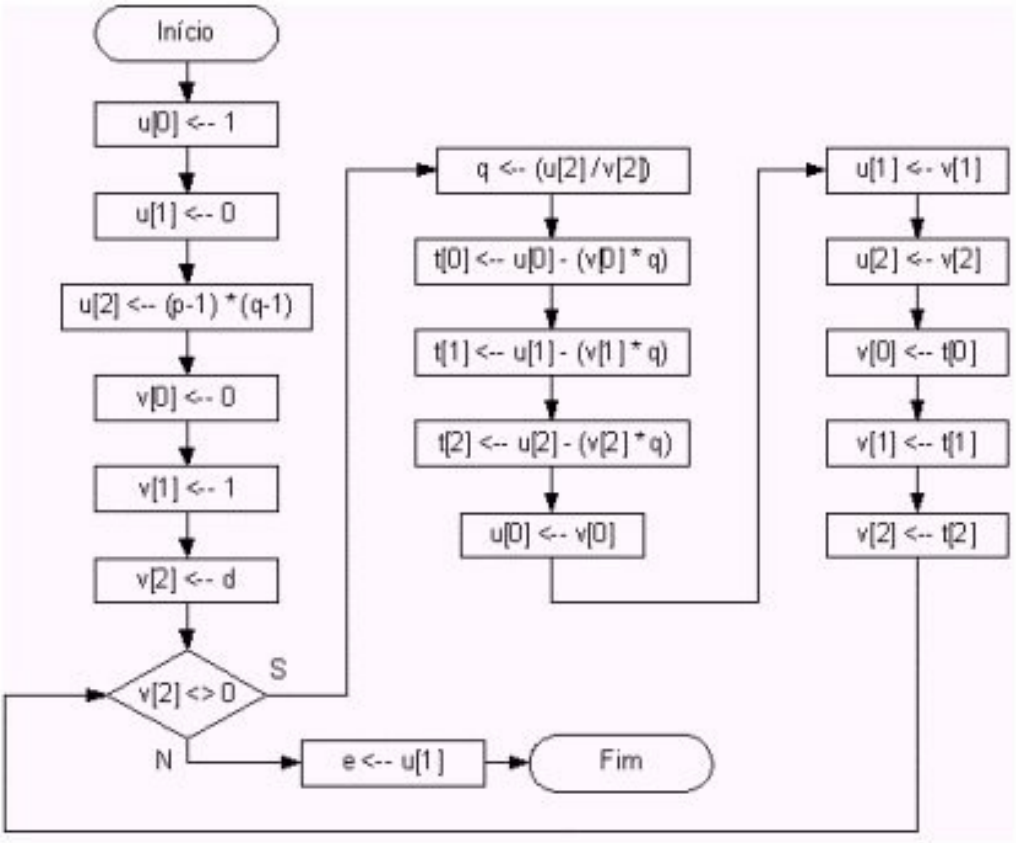

Figura 35 - Algoritmo de Euclides Estendido.

APÊNDICE C – Criptografia RSA Utilizando o Algoritmo de Brakley

Programa extraído de (MORENO,2005)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

ENTITY mod_exp is
generic (tamanho: integer := 24);
	port(
		M : in std_logic_vector(tamanho-1 downto 0);
		E : in std_logic_vector(tamanho downto 0);
		n : in std_logic_vector(tamanho+7 downto 0);
		CLK : in std_logic;
		C : out std_logic_vector(tamanho+7 downto 0);
		PRONTO : out std_logic
	);
end mod_exp;
architecture arc of mod_exp is
begin
	process(clk)
	variable temp: std_logic_vector(2*tamanho-1 downto 0);
	variable flag: std_logic_vector(2 downto 0);
	variable A,B,R : std_logic_vector(2*tamanho-1 downto 0);
	variable i,j : integer;
	begin
		if clk'event and clk = '1' then
			if flag = "000" then
				j := 17; --"010001"; --17
				temp :=
"000000000000000000000000000000000000000000000001"; --2*tamanho
				flag := "001";
			elsif flag = "001" then
				if j >=0 then
					A := temp;
					B := temp;
					R := (others=>'0');
					i := 0; -- "000000";
					flag := "010";
				else
					PRONTO <= '1';  --
Atribuição dos valores aos
					C <= temp(tamanho+7
downto 0); --sinais de saída.
				end if;
			elsif flag = "011" then
				temp := R;
```

```
                                if E(j) = '1' then
                                        A := temp;
                                        B(tamanho-1 downto 0 ) :=
M;
                                        B(2*tamanho-1 downto
tamanho) := "000000000000000000000000";-- tamanho zeros
                                        R := (others=>'0');
                                        i := 0; --"000000";
                                        flag := "100";
                                else
                                        j := j - 1;
                                        flag := "001";
                                end if;
                        elsif flag = "101" then
                                temp := R;
                                j := j - 1;
                                flag := "001";
                        elsif flag = "010" or flag = "100" then -
- BLAKLEY
                                if i < 2*tamanho then
                                        R := SHL(R,"1");
                                        if A(2*tamanho-1-i) = '1'
then R := R + B; end if;
                                        if R > n then R := R - n;
end if;
                                        if R > n then R := R - n;
end if;
                                        i := i + 1;
                                else
                                        flag := flag + 1;
                                end if;
                        end if;
                end if;
        end process;
end arc;
```

Somador utilizado no Algoritmo Rápido de Montgomery

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

ENTITY c_s_adder1 IS
generic (tamanho: integer := 4);-- chaves de 2*tamanho bits
PORT(
        x: IN std_logic_vector(tamanho DOWNTO 0);
        y: IN std_logic_vector(tamanho DOWNTO 0);
        M: IN std_logic_vector(tamanho DOWNTO 0);
```

```
        CC,CC1,CC2,CC3: out std_logic_vector(tamanho+2 DOWNTO
0);
        PP2: out std_logic_vector(tamanho+1 DOWNTO 0);
        P: out std_logic_vector(tamanho DOWNTO 0)
        );
END c_s_adder1;

-- INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF c_s_adder1 IS


BEGIN
  process(x,y,M)
        variable est: std_logic;
   variable P1: std_logic_vector(tamanho DOWNTO 0);
        variable  Y_AUX,M_AUX,P2: std_logic_vector(tamanho+1
DOWNTO 0);
  variable S, C,Sout,Cout: std_logic_vector(tamanho+2 downto 0);

  begin

        S:=(others=>'0');
        C:=(others=>'0');

       for j in 0 to tamanho loop


                  if X(j) = '1' then
                      Y_AUX(tamanho downto 0):=Y;
                  else
                     Y_AUX:=(others=>'0');
                  end if;

                  Sout := '0' &((S(tamanho+1 downto 0) XOR
C(tamanho+1 downto 0)) XOR  Y_AUX);
            Cout:= (((S(tamanho+1 downto 0) AND C(tamanho+1
downto 0)) OR (S(tamanho+1 downto 0) AND Y_AUX)) OR (C(tamanho+1
downto 0) AND  Y_AUX)) &'0';
             CC<=Sout;
             CC1<=Cout;
                  if (Sout(0)='1') then
                     M_AUX( tamanho downto 0):=M;
                  else
                     M_AUX:=(others=>'0');
                  end if;

                  S :='0'& ((Sout(tamanho+1 downto 0) XOR
Cout(tamanho+1 downto 0)) XOR  M_AUX);
            C := (((Sout(tamanho+1 downto 0) AND Cout(tamanho+1
downto 0)) OR (Sout(tamanho+1 downto 0) AND M_AUX)) OR
(Cout(tamanho+1 downto 0) AND M_AUX))&'0';

            CC2<=S;
```

```
            CC3<=C;
                    S:=SHR(S,"1");
                  C:=SHR(C,"1");
                 end loop;
                  P2 :=  S(tamanho+1 downto 0)+C(tamanho+1
downto 0);

               if (P2>=M) then
                  P2 := P2-M;
               end if;

              P<=P2(tamanho downto 0);
  end process;

END behavioral;
```

APÊNDICE D – Almost Montgomery Inverse

Algoritmo 1 : Almost *Montgomery* Inverse
AlmMonInv (a, M)
Dados : a ∈ [1;M – 1] and M
Resultado: r e k onde $r = a^{-1}2^k \pmod M$
e $m \leq k \leq 2m$
1. u := M, v := a, r := 0, s := 1
2. k := 0
3. Enquanto (v > 0)
4. se u é par então u := u/2, s := 2s
5. senão se v é par então v := v/2, r := 2r
6. senão se u>v então u:=(u – v)/2,r :=r + s, s := 2s
7. senão se v≥u então v:=(v – u)/2,s :=s + r,r := 2r
8. k := k + 1
9. se r≥ M então r := r –M
retorno r := M – r
retorno k
ModInv (r, M, k)
Input : r, M e k do AlmMonInv

Algoritmo 2 de Montgomery – Real Modular Inverse

Resultado : x, onde $x = a^{-1} \pmod M$
Para i = 1 até k faça
se r é par então r := r=2
senão r := (r +M)/2
retorno x := r

APÊNDICE E – Criptografia RSA com o Método de Gutub Associado ao Método de Quadratura e Multiplicação em Linguagem VHDL

```
use library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity CripGutKoc is
generic (tamanho: integer:=5);--chaves de 2*tamanho bits
  port(
           T:in std_logic_vector(tamanho downto 0);--Base da
potencia
           E:in std_logic_vector(tamanho downto 0);--expoente do
RSA
           M:in std_logic_vector(tamanho downto 0);--Modulo n da
potencia
           clk: in std_logic;
           C:out std_logic_vector(tamanho downto 0)

      );
           end CripGutKoc;
          architecture arc of CripGutKoc  is

          begin

           process(clk)
VARIABLE flag: std_logic_vector(2 downto 0);
VARIABLE P,X,Y,Z: std_logic_vector(tamanho downto 0);

VARIABLE i : integer range 0 to tamanho;
VARIABLE j : integer range 0 to tamanho;
VARIABLE temp : integer range 0 to 7;
VARIABLE k: integer range 0 to tamanho;
begin
  if(clk'event and clk='1') then
                          if flag = "000" then-- atribui os
valores iniciais
    i:= tamanho;
    j:= tamanho;
    k:= tamanho;
  flag:= "001";
  elsif flag = "001" then                                    if
E(i) = '1' then                                  P:=T;
else
  P:="000001"; -- tamanho + 1                                   end
if;
                                  i:= i-1;
flag:= "011";                  elsif flag = "010" then --
multiplicação modular de Gutub
```

```
for j in tamanho downto 0 loop
  P:= SHL(P,"1");
if P>=M then
P:= P-M;
  end if;                                    if  X(j) = '1' then
  P:=P+Y;
  if P>=M then
    P:=P-M;
  end if;
end if;
                        end loop;
if temp = 1 then
  flag:="100";
                                     elsif temp = 0 then
                                       if k = 0 then
                                         flag:= "101";
                                       else
                                       flag:= "011";
                                      end if;
end if;

elsif flag = "011" then -- faz o estado 001 executar
potenciação-
  j:=tamanho;
  X:=P;
  Y:=P;
  temp:=1;
  P:=(others=>'0');
  flag:="010";                                    elsif flag = "100"
then
j:=tamanho;
  if E(i) = '1' then
  X:=P;
    Y:=T;
    flag:= "010";
    P:=(others=>'0');
  else
    flag:="011";
  end if;
  if k = 0 then
    P:=X;
                                       flag:= "101";
                                    end if;
  k:=k-1;
                                    i:=i-1;
                                    temp:=0;
                                   elsif flag = "101" then
                                    C<= P; --fim
                                    end if;
                         end if;
                    end process;
end arc;
```

APÊNDICE F – Criptografia RSA Utilizando o Método Fast de Montgomery

```
library ieee; // Programa do componente no Apêndice G
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity Montgomery is
generic (tamanho: integer := 4);-- chaves de 2*tamanho bits
  port(
           A:in std_logic_vector(tamanho downto 0);
           T:in std_logic_vector(tamanho downto 0);
           E:in std_logic_vector(tamanho downto 0);
           M:in std_logic_vector(tamanho downto 0);
           clk: in std_logic;
           C:out std_logic_vector(tamanho downto 0)

      );
           end Montgomery;

   architecture arc of Montgomery  is

  COMPONENT c_s_adder1
 PORT(   x: IN std_logic_vector(tamanho DOWNTO 0);
        y: IN std_logic_vector(tamanho DOWNTO 0);
        M: IN std_logic_vector(tamanho DOWNTO 0);
        P: out std_logic_vector(tamanho DOWNTO 0)
        );
   END COMPONENT;

    signal X,Y,POUT: std_logic_vector(tamanho downto 0);

     begin

     S1:c_s_adder1 port map(X,Y,M,POUT);

           process(clk)
               VARIABLE flag: std_logic_vector(2 downto 0);
                VARIABLE P: std_logic_vector(tamanho DOWNTO 0);
             VARIABLE i : integer range 0 to tamanho+1;
             VARIABLE temp : integer range 0 to 2;


           begin
                 if(clk'event and clk='1') then

                          if flag = "000" then
                                   i:= tamanho;
```

```
                    P:= T;
                  flag:= "001";
                elsif flag = "001" then
                  if E(i) = '0' then
                    i:= i-1;
                    flag:= "001";
                  else
                    flag:= "011";
                  end if;
                elsif flag = "010" then
                          if temp=0 then
                           flag:="011";
                          elsif temp=1 then
                             flag:="100";
                         elsif temp=2 then
                                  flag:="110";
                          end if;
                          P:=POUT;
                    elsif flag = "011" then
              X<=P;
              Y<=P;
              temp:=1;
              flag:= "010";
                    elsif flag = "100" then
                if E(i) = '1' then
                  X<=P;
                  Y<=A;
                  temp:=0;
                  flag:= "010";
                            else
                              flag:="011";
                            end if;
                            if i= 0 then
                              if E(0) = '1' then
                                flag:= "101";
                              else
                                flag:= "110";
                              end if;
                            end if;
                            i:= i-1;
                  elsif flag = "101" then
                  X<= P;
                  Y<= A;
                  temp:= 2;
                  flag:= "010";
                  elsif flag = "110" then
                          C<=P;
                  end if;
          end if;
      end process;
end arc;
```

APÊNDICE G – Somador

Componente do programa do Apêndice F

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- EXTERNAL PORTS
ENTITY c_s_adder1 IS
PORT(   x: IN std_logic_vector(4 DOWNTO 0);
        y: IN std_logic_vector(4 DOWNTO 0);
        M: IN std_logic_vector(4 DOWNTO 0);
        P: out std_logic_vector(4 DOWNTO 0)
        );
END c_s_adder1;

-- INTERNAL BEHAVIOR
ARCHITECTURE behavioral OF c_s_adder1 IS


BEGIN
  process(x,y,M)


        variable  Y_AUX,M_AUX,P1: std_logic_vector(4 DOWNTO 0);
  variable S, C,Sout,Cout: std_logic_vector(5 downto 0);

  begin
        S:=(others=>'0');
        C:=(others=>'0');

       for j in 0 to 4 loop


                   if X(j) = '1' then
                       Y_AUX:=Y;
                   else
                      Y_AUX:=(others=>'0');
                   end if;

                   Sout := '0'&((S(4 downto 0) XOR C(4 downto 0))
XOR Y_AUX);
```

```
            Cout:= (((S(4 downto 0) AND C(4 downto 0)) OR (S(4
downto 0) AND Y_AUX)) OR (C(4 downto 0)AND Y_AUX))&'0';



                  if (Sout(0)='1') then
                     M_AUX:=M;
                  else
                     M_AUX:=(others=>'0');
                  end if;

                  S := '0' & ((Sout(4 downto 0) XOR Cout(4
downto 0)) XOR M_AUX);
            C := (((Sout(4 downto 0) AND Cout(4 downto 0)) OR
(Sout(4 downto 0) AND M_AUX)) OR (Cout(4 downto 0) AND M_AUX))&
'0';

                  S:=SHR(S,"1");
                  C:=SHR(C,"1");


              end loop;

               P1 :=  S(4 downto 0)+C(4 downto 0) ;
               if (P1>=M) then
                  P1 := P1-M;
               end if;

              P<=P1(4 downto 0);


  end process;

END behavioral;
```

APÊNDICE H – Criptografia RSA Utilizando o Algoritmo Faster de Montgomery

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

-- Potenciacao modular no dominio de Montgomery C = T**E mod M
-- T = Imagem da base base da potencia no dominio de Montgomery
-- A = Imagem de 1
-- E = expoente de potencia
-- M = Modulo da multiplicação modular
-- Crip-Texto criptografado no dominio de Montgomery - P/se
obter C basta fazer C=Crip*R**(-1) mod M
--Obs. A chave (E,M) terá sempre o mesmo numero de bits para 'E'
e 'M'.
--Obs. 'E' e 'M' terão sempre 'tamanho' bits
entity Mtgmr1 is
generic (tamanho: integer := 5);-- chave de 2 tamanho bits --
'E' tem 5 bits e 'M' tem 5 bits
  port(
           A:in std_logic_vector(tamanho downto 0);
           T:in std_logic_vector(tamanho downto 0);
           E:in std_logic_vector(tamanho downto 0);
           M:in std_logic_vector(tamanho downto 0);
           clk: in std_logic;
           Crip:out std_logic_vector(tamanho downto 0)

      );
           end Mtgmr1;

   architecture arc of Mtgmr1  is


     begin


           process(clk)
               VARIABLE flag: std_logic_vector(2 downto 0);
                VARIABLE I,P,R,Y,X: std_logic_vector(tamanho
DOWNTO 0);
             VARIABLE j,k : integer range 0 to tamanho; -- em
função dos tamanhos de E e de  X
             VARIABLE temp : integer range 0 to 1;
             VARIABLE C,S, Sout: std_logic_vector((tamanho+1)
DOWNTO 0);
```

```
            begin
                  if(clk'event and clk='1') then

                              if flag = "000" then

                                    k:=tamanho;

                                    P:= T;

                                  flag:= "001";
                                elsif flag = "001" then
                                  if E(k) = '0' then
                                    k:= k-1;
                                    flag:= "001";
                                  else
                                    flag:= "100";
                                  end if;
                                elsif flag = "010" then


                                                  if ((S(0) =  C(0))  AND
(X(j) = '0'))then
                                                      I:=(others=>'0');
                                                  elsif ((S(0) =  C(0))
AND (X(j) = '1'))then
                                                    if Y(0) = '0'then
                                                      I:=Y;
                                                    else
                                                      I:= R;
                                                    end if;

                                                elsif ((S(0) /=  C(0))
AND (X(j) = '0'))then
                                                    I:=M;
                                                  elsif ((S(0) /=  C(0))
AND (X(j) = '1'))then
                                                    if Y(0) = '0'then
                                                      I:=R;
                                                    else
                                                      I:= Y;
                                                    end if;
                                                  end if;
                                                Sout := '0' & ((S(tamanho
downto 0) XOR C(tamanho downto 0)) XOR I);
                                                   C := (((S(tamanho
downto 0) AND C(tamanho downto 0)) OR (S(tamanho downto 0) AND
I)) OR (C(tamanho downto 0) AND I))& '0' ;
                                              S:= SHR(Sout,"1");
                                                  C:= SHR(C,"1");
                                                  j:= j+1;
                                                  if j<tamanho then
                                                    flag:= "010";
                                                  end if;
```

```
                                        flag:= "011";
                                        elsif flag = "011" then
                                       P:= S(tamanho downto
0)+C(tamanho downto 0);
                                        if P>=M then
                                           P:= P-M;
                                      end if;

                                        if temp=0 then
                                        flag:= "100";
                                        elsif temp=1 then
                                        flag:="101";
                                        end if;

                                 elsif flag = "100" then
                                 --j:=0;
                                 S:=(others=>'0');
                                 C:=(others=>'0');
                           X:=P;
                           Y:=P;
                           R:= Y+M;
                           temp:=1;
                           flag:= "010";
                                 elsif flag = "101" then
                             if E(k) = '1' then
                               --j:=0;
                               S:=(others=>'0');
                                   C:=(others=>'0');
                               X:=P;
                               Y:=A;
                               R:= Y+M;
                               temp:=0;
                               flag:= "010";
                                         else
                                           flag:="100";
                                         end if;
                                         if k=0 then
                                           flag:="110";
                                         end if;
                                         k:=k-1;
                                 elsif flag = "110" then
                                 Crip<=P;
                                 end if;
                        end if;
                  end process;
            end arc;
```