UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Caio Alonso da Costa

Desenvolvimento de Hardware de Criptografia RSA em Linguagem Verilog

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Engenharia Elétrica.

Área de Concentração: Microeletrônica

Orientador: Dr. Tales C. Primenta
Coorientador: Dr. Otávio A. S. Carpinteiro

Julho de 2014
Itajubá-MG

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Caio Alonso da Costa

Desenvolvimento de Hardware de Criptografia RSA em Linguagem Verilog

Dissertação aprovada por banca examinadora em 7 de julho de 2014, conferindo ao autor o título de **Mestre em Engenharia Elétrica.**

Banca Examinadora:
Prof. Dr. Tales Cleber Pimenta
Prof. Dr. Otávio Augusto Salgado Carpinteiro
Prof. Dr. Robson Luiz Moreno
Prof. Dr. Leonardo Mesquita

Julho de 2014
Itajubá-MG

Ministério da Educação
UNIVERSIDADE FEDERAL DE ITAJUBÁ
Criada pela Lei 10436, de 24 de Abril de 2002

TERMO DE ACEITE
VERSÃO FINAL DA DISSERTAÇÃO

Eu, Prof. Dr. Tales C Pimenta, declaro que aceito o CD com a versão final da dissertação de meu orientado Caio Alonso da Costa, matrícula: 22093 do Programa de Pós-Graduação em Eng. Elétrica e que o mesmo contém as indicações e correções sugeridas pela banca e poderá ser realizada sua homologação.

Em 29 de Julho de 2014.

Assinatura orientador

Este trabalho é dedicado aos meus pais David e Fernanda, ao meu irmão Lucas e a minha namorada Jacqueline que, durante o mestrado, tiveram muita paciência comigo e sempre me apoiaram nos momentos difíceis.

Agradecimentos

A Deus pelas oportunidades em toda a minha vida.

Aos meus pais, David e Fernanda, e ao meu irmão, Lucas, que me apoiaram incondicionalmente durante toda a minha formação.

A minha namorada, Jacqueline, pela paciência e carinho.

Aos professores Tales Cleber Pimenta e Robson Luiz Moreno pelo apoio, amizade, dedicação e ajuda ao longo do desenvolvimento do trabalho.

Ao professor Otávio Augusto Salgado Carpinteiro pela sugestão do tema a ser trabalhado.

Aos amigos de república, faculdade, mestrado e grupo de pesquisa pelo incentivo e colaboração.

Ao CNPq pelo apoio financeiro para desenvolvimento deste trabalho.

Ao pessoal da MINASIC.

A todos que, de forma direta ou indireta, contribuíram para a realização deste trabalho.

“Nossas virtudes e nossas falhas são inseparáveis, como a energia e a matéria. Quando elas se separam, não mais existe o homem."

Nikola Tesla

Resumo

Este trabalho apresenta um modelo e arquitetura de um hardware desenvolvido em linguagem Verilog para exponenciação modular do algoritmo de criptografia assimétrica RSA. Uma breve discussão sobre os tipos de criptografia e a linguagem Verilog é apresentada no início da dissertação. A criptografia RSA e os algoritmos utilizados são apresentados através de pseudocódigos ao longo deste trabalho e implementações em linguagem C e Java são apresentadas nos anexos. A arquitetura desenvolvida foi baseada nos algoritmos apresentados e arquiteturas difundidas nas literaturas com modificações para melhorar o desempenho. Uma arquitetura de 4 bits é apresentada com todos os blocos e interligações comentadas. Um exemplo de cifragem para esta arquitetura com a discussão do caminho dos dados na arquitetura é realizada. Em seguida uma arquitetura para 1024 bits é proposta e exemplificada através de um processo de cifragem e decifragem. O coprocessador RSA foi implementado com um conjunto de células básicas da tecnologia de 0,18μm CMOS IBM7SF. Esta implementação realiza um processo de cifragem ou decifragem de 1024 bits em 8,44 ms e o *throughput* medido na sua máxima frequência de operação é de 121,269 Kbps.

Palavras-chaves: Criptografia, RSA, Multiplicação Modular de Montgomery, Exponenciação Modular, CMOS, ASIC, VLSI.

Abstract

This work presents a model and a hardware architecture for modular exponentiation developed in Verilog language for the RSA asymmetric encryption algorithm. A brief discussion of the types of encryption and Verilog language is presented at the beginning of this work. The RSA encryption and the algorithms used are presented as pseudocodes in this work and implementations in C and Java languages are presented in the Annexes. The architecture was developed based on the algorithms presented and architectures disseminated in the literature with a few modifications to improve performance. A 4-bit architecture is presented with all the blocks and interconnections commented. An example of encryption for this architecture with the discussion of the data path is performed. Then an architecture of 1024 bits is proposed and exemplified through a process of encryption and decryption. The RSA coprocessor is implemented with a set of basic cells from the 0.18μm CMOS IBM7SF technology. This implementation performs an encryption or decryption of 1024 bits in 8.44 ms and the throughput measured at its maximum clock frequency is 121.269 Kbps .

Key-words: Cryptography, RSA, Montgomery Modular Multiplication, Modular Exponentiation, CMOS, ASIC, VLSI.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UNIFEI	Universidade Federal de Itajubá
FPGA	*Field Programmable Gate Arrays*
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
AES	*Advanced Encryption System*
3DES	*Triple Data Encryption Standard*
DES	*Data Encryption Standard*
IDEA	*International Data Encryption Algorithm*
RSA	Rivest Shamir Adleman
HDL	*Hardware Description Language*
RTL	*Register Transfer Level*
mdc	maior divisor comum
RCA	*Ripple Carry Adder*
KSA	*Kogge-Stone Adder*
ASCII	*American Standard Code for Information Interchange*
SPI	*Serial Peripheral Interface*

Sumário

1 Introdução

1.1 Considerações Iniciais

A fascinação de alguns indivíduos em manter informações em sigilo sempre esteve presente em toda a história da humanidade. O primeiro indício do uso de criptografia no mundo foi registrado no Egito, mas de fato, quem foi o primeiro a utilizar esta técnica para enviar mensagens secretas a suas legiões foi o imperador romano Júlio César [1]. Contemporaneamente, em um mundo cada vez mais globalizado onde conhecimento é sinônimo de poder, técnicas para manter informações valiosas de forma segura foram desenvolvidas.

Informações confidenciais, como registros bancários, faturas de cartão de crédito, senhas ou comunicação privada, são (e devem ser) criptografadas - modificadas de tal maneira que devam ser compreensíveis apenas para as pessoas que têm acesso autorizado e indecifráveis para as demais. Proteger dados e sistemas eletrônicos é indispensável nos dias de hoje [2].

Trappe [1] afirma que todos os algoritmos de criptografia são vulneráveis a ataques e falhas. Nenhum código é completamente indecifrável, nem toda a sofisticação matemática pode impedir essa possibilidade, já que um adversário pode ter grande poder computacional e recursos humanos dedicados a tentar decifrar um código. Dessa forma, toda a noção de segurança está diretamente ligada ao poder de computação - um código é tão seguro quanto a quantidade de poder de computação necessário para decifrá-lo [2].

1.2 Motivação e Objetivos

O Grupo de Microeletrônica da Universidade Federal de Itajubá - UNIFEI já domina a área de criptografia em circuitos programáveis, como Arranjos de Portas Programáveis em Campo - FPGA (*Field Programmable Gate Arrays*). Entretanto, os estudos de circuitos criptográficos realizados, até o momento, se limitavam a projetos de criptografia simétrica em FPGA utilizando a Linguagem de Descrição de Hardware de Circuitos Integrados de Altíssima Velocidade - VHDL (*Very High Speed Integrated Circuit Hardware Description Language*).

A dissertação de Campos [3] foi o primeiro trabalho desenvolvido no campo de criptografia simétrica *Advanced Encryption Standard* - AES em hardware utilizando FPGA no grupo de microeletrônica da UNIFEI. Saad [4] foi o segundo pesquisador a desenvolver um hardware para criptografia simétrica AES também utilizando FPGA. Uma das

últimas pesquisas neste mesmo algoritmo, intitulado "Desenvolvimento de hardware configurável de criptografia simétrica utilizando FPGA e linguagem VHDL", foi apresentado por Gomes [5] em 2011. Esta dissertação, que por ora se apresenta, de forma complementar, contribui para o desenvolvimento da área uma vez que sua proposta fundamenta-se na implementação de um circuito integrado para criptografia assimétrica *Rivest Shamir Adleman* - RSA através da Linguagem de Descrição de Hardware Verilog - Verilog HDL (*Verilog Hardware Description Language*).

O método de criptografia assimétrica RSA é amplamente utilizado nos meios de comunicações digitais. Uma das aplicações mais usuais para a criptografia assimétrica RSA é a autenticidade de dados digitais e de assinaturas digitais [6]. As operações de criptografia RSA são tarefas com alto custo computacional pois consiste de operações de exponenciação modular. Dessa forma, uma maneira de reduzir o tempo gasto na operação e liberar o custo de processamento do algoritmo no processador principal de um sistema é através do desenvolvimento de um Circuito Integrado de Aplicação Específica - ASIC (*Application Specific Integrated Circuit*).

O objetivo deste trabalho foi desenvolver um protótipo em ASIC para criptografia assimétrica RSA. Este trabalho introduz o algoritmo de criptografia assimétrica RSA e apresenta uma solução em software implementada em linguagem de programação C e Java. A partir do software, uma solução em forma de circuito foi proposta e desenvolvida utilizando a linguagem Verilog HDL. A partir do código Verilog HDL, o fluxo digital da ferramenta Cadence foi percorrido com finalidade de atingir a última etapa de um projeto de circuito integrado. O *layout* final do circuito integrado pronto para a confecção do protótipo deste trabalho é ilustrado e os resultados são apresentados.

1.3 Organização do Trabalho

Esta dissertação está dividida em cinco capítulos. No Capítulo 1 apresentam-se os principais conceitos sobre criptografia, os tipos de criptografia e a linguagem utilizada para desenvolvimento do circuito integrado proposto.

O algoritmo de criptografia assimétrica RSA é apresentado no Capítulo 2. Nele encontram-se, também, um exemplo do uso de criptografia RSA, uma discussão sobre a segurança do RSA, os algoritmos em pseudocódigo para o processo de criptografia, uma arquitetura de hardware base na qual os estudos foram iniciados e discussões de implementação dos algoritmos em forma de circuito.

No Capítulo 3 apresenta-se a arquitetura proposta para o circuito integrado. Inicialmente discute-se a arquitetura para quatro bits com detalhes dos blocos, interligações entre os blocos e o fluxo de dados. Em seguida apresentam-se as modificações para a arquitetura de 1024 bits.

No Capítulo 4 são feitas as análises dos resultados da implementação deste circuito e uma comparação com os resultados do estado da arte recentemente publicados em congressos e revistas.

Finalmente, no Capítulo 5 conclui-se o estudo desenvolvido. Ainda nesse capítulo são sugeridas ideais para trabalhos futuros.

1.4 Criptografia

Uma mensagem original é conhecida como **texto claro**, **texto pleno**, ou *plain-text*, enquanto a mensagem codificada é chamada de **texto cifrado**, ou *ciphertext*. Para o processo de conversão de um texto claro em um texto cifrado costuma-se utilizar a terminologia **cifragem** (*encripta, codifica, criptografa, cifra*), e para tornar do texto cifrado em texto claro, utiliza-se o termo **decifragem** (*decripta, decodifica, decriptografa, decifra*). Os métodos e algoritmos (modelos) utilizados para a cifragem e decifragem constituem a área de estudo conhecida como **criptografia**. Os conjuntos dos métodos e algoritmos são conhecidos **sistema criptográficos** ou **cifras** [7].

Os estudos específicos que tratam da quebra de sistemas de criptografia denominam-se **análise criptográfica** e os profissionais que as estudam são chamados **criptoanalistas**. As áreas da criptografia e análise criptográfica em conjunto são chamadas de **criptologia** [8].

Em quase a totalidade das literaturas existentes em criptografia, os nomes poéticos atribuídos aos participantes para exemplificar o funcionamento dos algoritmos de forma clara são Alice e Bob. Neste trabalho, os mesmo serão utilizados de forma facilitar o entendimento e preservar o costume.

1.4.1 Tipos de Criptografia

Os algoritmos de criptografia podem ser classificados três diferentes dimensões independentes. A primeira dimensão consiste no tipo de operação usada para transformar um texto claro em texto cifrado. O fundamento principal desta dimensão é que a informação original não seja perdida. A segunda dimensão consiste na quantidade de chaves utilizadas. Para os algoritmos de criptografia simétrica, uma única chave é necessária para fazer o processo de cifragem e decifragem, ou seja, tanto o remetente quanto o destinatário possuem a mesma chave. Entretanto para os algoritmos de criptografia assimétrica, um par de chaves é necessário, em que uma chave é utilizada no processo de cifragem e a outra chave no processo de decifragem. A terceira e última dimensão abordada diz respeito a forma em que os algoritmos que tratam os dados. Um algoritmo de cifra de bloco faz o processo de cifragem e decifragem com blocos de tamanhos fixos, enquanto outros algoritmos tratam os dados em fluxo contínuo [9].

1.4.2 Aplicações de Criptografia

A criptografia, através de cifragem e decifragem de mensagens, é utilizada em problemas reais do mundo que necessitam de segurança de informação. Segundo Trappe [1], existem quatro tipos de problemas que a criptografia pode resolver:

- **Confidencialidade:** Ninguém, além do destinatário e do remetente, conseguem ler a informação enviada de maneira clara, ou seja, o texto claro.
- **Integridade de dados:** As pessoas envolvidas na comunicação desejam ter certeza que a informação enviada não foi alterada. Por exemplo, erros de comunicação podem ocorrer. Alguns algoritmos de criptografia utilizam funções específicas para evitar manipulação indesejada de dados por invasores.
- **Autenticação:** O destinatário deseja ter certeza que somente uma pessoa enviou a mensagem que ele recebeu, ou seja, trata-se de um remetente único. Geralmente a autenticação tem duas vertentes: identificação e informação do dado. A identificação está relacionada aos indivíduos envolvidos na comunicação enquanto a informação do dado está relacionada com as informações da criação do dado, como quem criou os dados, data e hora.
- **Não-repúdio:** O remetente não pode alegar que não enviou a mensagem. Este problema é de particular relevância para transações financeiras que ocorrem em aplicações de comércio eletrônico, onde é importante o consumidor não negar a autorização de uma compra.

Apesar dos conceitos de autenticação e não-repúdio serem bastante semelhantes, existe uma fundamental diferença. Em sistemas de criptografia simétrica, o sistema de autenticação é automático, uma vez que uma pessoa que tem em posse a chave pode cifrar e decifrar o dado. Entretanto, não há uma forma de como provar quem enviou uma mensagem pois a origem da mesma pode ter sido do próprio indivíduo. Dessa forma o não-repúdio é impossível de ser alcançado. Em algoritmos de criptografia assimétrica, tanto a autenticação quanto o não-repúdio são possíveis [1].

1.4.3 Criptografia Simétrica

A criptografia simétrica, também chamada de criptografia convencional ou de chave única, era o único tipo de criptografia em uso antes do desenvolvimento da criptografia assimétrica na década de 1970. Esse continua sendo, de longe, o mais usado dos dois tipos de criptografia. A Figura 1 apresenta um modelo de criptografia simétrica e seus cinco componentes: texto claro, algoritmo de criptografia, chave secreta, texto cifrado e algoritmo de decriptografia [7].

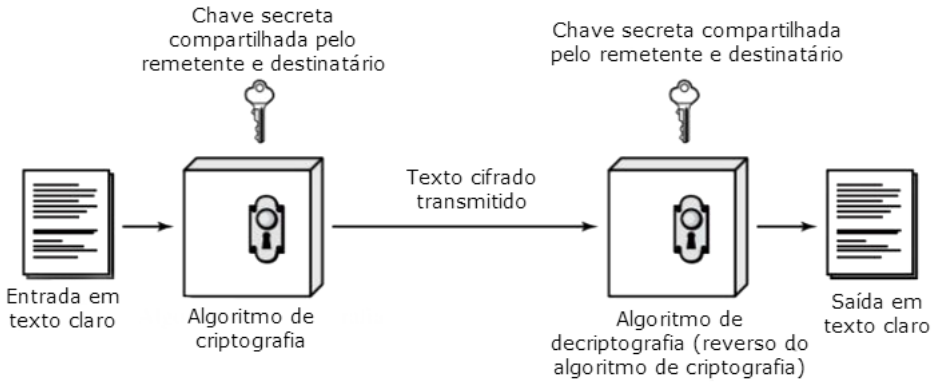

Figura 1: Modelo simplificado criptografia simétrica [7].

O texto claro consiste da mensagem ou dos dados originais, inteligíveis, alimentados no algoritmo como entrada. A chave secreta, que é independente do texto claro e do algoritmo, também é uma entrada do algoritmo. O algoritmo de criptografia realiza diversas substituições e transformações exatas no texto claro que dependem do valor da chave utilizada, produzindo o texto cifrado. Vale ressaltar que para uma mesma mensagem, duas chaves distintas produzirão dois textos cifrados diferentes. O texto cifrado é um conjunto de dados aparentemente aleatório e sem sentido. Em seguida, o texto cifrado é enviado através de um sistema de comunicação que, na maioria dos casos, não consiste de um canal seguro. O algoritmo de decriptografia, ou decifragem, realiza a mesma operação que o algoritmo de criptografia, porém de modo inverso. Para a mesma chave utilizada no processo de cifragem, o algoritmo de decifragem recupera o texto original a partir do texto cifrado [7].

Para a utilização deste modelo de criptografia, deve-se obedecer dois requisitos funcionais. O primeiro requisito é a necessidade da utilização de um algoritmo de criptografia robusto, uma vez que se alguém tem acesso a um ou mais textos cifrados e conheça o algoritmo de criptografia utilizado não seja capaz de decifrar a mensagem original ou descobrir a chave secreta utilizada. O segundo requisito é que ambas as partes envolvidas na comunicação precisam conhecer a chave secreta e devem mantê-la protegida. Se alguém for capaz de descobrir a chave e conhecer o algoritmo de criptografia, toda a comunicação usando essa chave poderá ser lida [7].

Alguns exemplos de algoritmos de criptografia simétrica são: AES, Twofish, Blowfish, 3DES, DES e IDEA[1, 7, 8].

1.4.4 Criptografia Assimétrica

O desenvolvimento da criptografia assimétrica é a maior e talvez a única verdadeira revolução da sua história. Desde o seu início até os tempos modernos, praticamente todos os sistemas criptográficos têm sido baseados nas ferramentas elementares de substituição

e permutação. A criptografia assimétrica veio como uma mudança radical uma vez que é baseada em funções matemáticas [7].

Este tipo de criptografia difere da simétrica em outro aspecto: há duas chaves distintas com correlação matemática, uma para cifragem e outra para decifragem. Uma das motivações principais para o estudo de sistemas de criptografia assimétrica foi a necessidade de resolver dois problemas que a criptografia simétrica não consegue contornar: a distribuição de chaves e a assinatura digital [7].

Há uma grande discussão sobre quem foram os inventores desse tipo de criptossistema, entretanto, muitas literaturas citam os matemáticos Diffie e Hellman como os primeiros a publicarem suas pesquisas nestes estudos [7].

Duas características fundamentais estão presentes no modelo de criptografia assimétrica. A primeira característica diz respeito a segurança. A partir do conhecimento do algoritmo utilizado para a criptografia e de uma das chaves é computacionalmente inviável determinar a outra chave necessária para reverter o processo. A segunda característica é relacionada as chaves para o processo de cifragem e decifragem. Qualquer uma das duas chaves relacionadas pode ser utilizada para a cifragem, com a outra sendo usada para a decifragem. Vale ressaltar que nem todos os algoritmos de criptografia assimétrica possuem essa última característica, entretanto, o algoritmo de estudo deste trabalho, o RSA, apresenta esta peculiaridade [7].

A Figura 2 apresenta um modelo de criptografia assimétrica, que possui seis ingredientes fundamentais: o texto claro, o algoritmo de criptografia, a chave pública, a chave privada, o texto cifrado e o algoritmo de decriptografia.

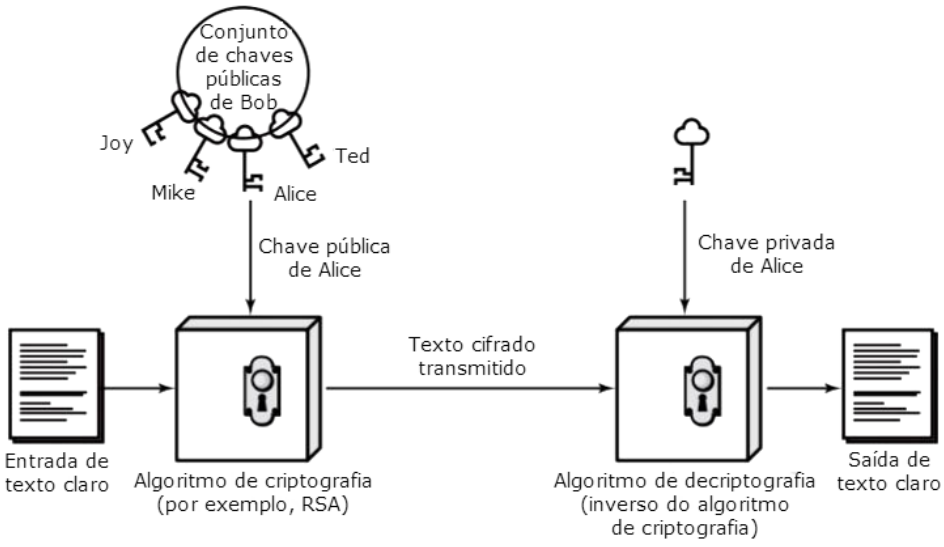

Figura 2: Modelo simplificado criptografia assimétrica [7].

Segundo Stallings [7], as etapas essenciais para utilização desse tipo de criptografia são:

1. Cada usuário gera um par de chaves a ser usado para a cifragem e a decifragem das mensagens.
2. Cada usuário coloca uma das duas chaves em um registro público ou outro arquivo acessível. Essa é denominada chave pública. A outra chave deve permanecer em segredo.
3. Se Bob deseja enviar uma mensagem confidencial para Alice, Bob cifra a mensagem usando a chave pública de Alice.
4. Quando Alice recebe a mensagem, ela decifra usando sua chave privada. Nenhum outro destinatário pode decifrar a mensagem, pois somente Alice conhece a sua chave privada.

Com esta técnica, todos os participantes têm acesso às chaves públicas, as chaves privadas são geradas localmente por cada participante e, portanto, nunca precisam ser distribuídas. Desde que a chave privada de um usuário permaneça protegida e secreta, a comunicação está protegida. A qualquer momento, um sistema pode alterar suas chaves, privada e pública, e fornecer a nova chave pública para substituir sua antiga [7].

As aplicações para os algoritmos de criptografia assimétrica podem ser separadas em três categorias. A primeira categoria é a utilização como criptografia de mensagem com chaves públicas, ou seja, o emissor criptografa sua mensagem utilizando a chave pública do destinatário para garantir que somente o destinatário consiga recuperar o texto claro. A segunda categoria é referente a assinatura digital, conforme ilustrado na Figura 3. Um emissor "assina" uma mensagem com sua chave privada. A assinatura é feita por um algoritmo criptográfico aplicado à mensagem ou a um pequeno bloco de dados da mensagem. Esta técnica é de extrema importância e utilizada para garantir transações financeiras de forma segura na internet.

Finalmente a terceira categoria é a utilização do algoritmo para que os dois lados envolvidos na comunicação estabeleçam uma chave de criptografia simétrica. Sendo assim, através da criptografia assimétrica é definida uma chave de criptografia simétrica para ser utilizada durante a troca de informações [7].

1.4.5 Comparativo entre Criptografia Simétrica e Assimétrica

A Tabela 1 resume alguns aspectos importantes da criptografia simétrica e da criptografia assimétrica. Para diferenciar as duas, será utilizado o termo **chave secreta**

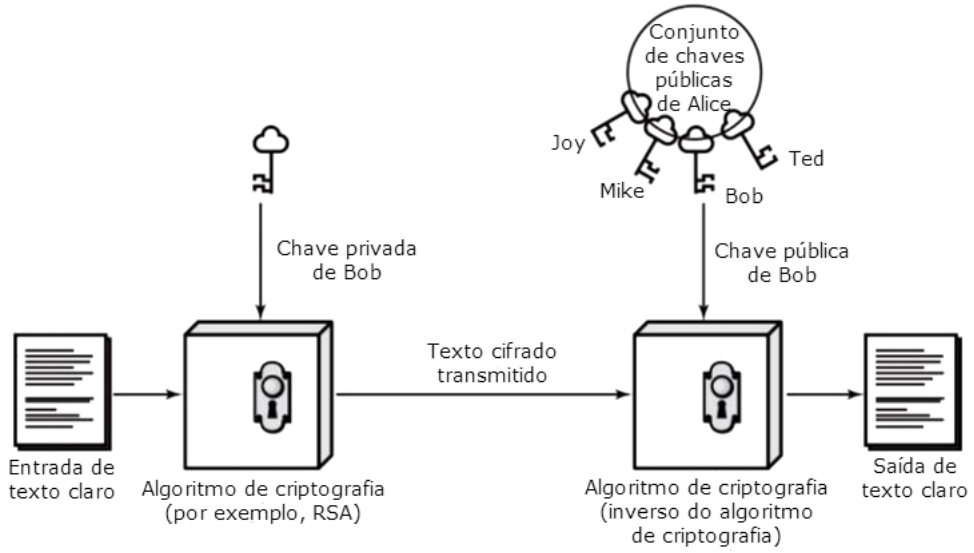

Figura 3: Modelo simplificado criptografia assimétrica - assinatura digital [7].

para referenciar a chave utilizada na criptografia simétrica. As duas chaves utilizadas na criptografia assimétrica serão referenciadas como **chave pública** e **chave privada**.

Tabela 1: Criptografia simétrica e assimétrica [7].

Criptografia simétrica	**Criptografia assimétrica**
Necessário para funcionar: 1. O mesmo algoritmo com a mesma chave é usado para criptografia e decriptografia. 2. O emissor e o receptor precisam compartilhar o algoritmo e a chave.	**Necessário para funcionar:** 1. Um algoritmo é usado para a criptografia e decriptografia com um par de chaves, uma para criptografia e outra para decriptografia. 2. O emissor e o receptor precisam ter uma das chaves do par casado de chaves (não a mesma chave).
Necessário para segurança: 1. A chave precisar permanecer secreta. 2. Deverá ser impossível ou pelo menos impraticável decifrar uma mensagem se nenhuma outra informação estiver disponível. 3. O conhecimento do algoritmo mais amostras do texto cifrado precisam ser insuficientes para determinar a chave.	**Necessário para segurança:** 1. Uma das duas chaves precisa permanecer secreta. 2. Deverá ser impossível ou pelo menos impraticável decifrar uma mensagem se nenhuma outra informação estiver disponível. 3. O conhecimento do algoritmo mais uma das chaves mais amostras do texto cifrado precisam ser insuficientes para determinar a outra chave.

1.5 Descrição de Hardware

1.5.1 Linguagens de Descrição de Hardware - HDLs

Há muito tempo, linguagens de programação como C, C++ e Java são utilizadas para descrever programas de computador que, são instruções sequenciais por natureza. De maneira análoga, os projetistas de circuitos integrados digitais sentiram uma necessidade de desenvolver uma linguagem padrão para descrição de circuitos digitais. Sendo assim, surgiu o conceito de Linguagens de Descrição de Hardware - HDLs (*Hardware Description Languages*). As HDLs permitem que projetistas modelem a concorrência dos processos que existem nos elementos de hardware [10].

As duas linguagens de descrição de hardware que rapidamente se tornaram populares são Verilog HDL e VHDL. As primeiras versões destas linguagens foram muito utilizadas para verificação de lógica e somente com o advento de algoritmos de síntese lógica é que a metodologia de projeto de circuitos digitais sofreu fortes mudanças. A partir destes algoritmos, os circuitos puderam ser descritos através de fluxo de sinais, ou seja transferência de dados entre registradores e operações lógicas efetuadas com estes sinais. Este tipo de descrição de operação de circuitos digitais síncronos recebe o nome de descrição em nível de transferência de registrador - RTL (*Register Transfer Level*). Dessa forma, os projetistas precisam se preocupar com o fluxo de dados entres registradores e como o circuito processa os dados. Os detalhes das portas lógicas e das interconexões do circuito implementado são automaticamente processados pelos algoritmos de síntese lógica [10].

Sendo assim, os projetistas ganharam flexibilidade na descrição de circuitos mais complexos em um nível mais abstrato em termos de funcionalidade e fluxo de dados. Dessa forma, existe um benefício na utilização de HDLs para descrição de circuitos digitais pois os softwares de síntese implementam a funcionalidade em termos de portas e interconexões automaticamente [10].

Segundo Palnitkar [10], as HDLs apresentam algumas vantagens quando comparados a projetos tradicionais baseados em esquemas elétricos:

- Os projetistas podem descrever circuitos digitais em um nível abstrato, ou seja, independente da tecnologia de fabricação.
- Ao descrever circuitos digitais em HDLs, as verificações funcionais do projeto podem ser realizadas em estágios preliminares.
- Desenvolver circuitos digitais em HDLs é equivalente a utilizar linguagens de programação para desenvolvimento de softwares. Utilizar uma linguagem em forma de texto traz como benefícios a facilidade na análise de falhas e erros.

Com o aumento da complexidade dos circuitos digitais e a sofisticação de ferramentas de automação de projetos eletrônicos - EDA (*Electronic design automation*), as HDLs tornaram-se os melhores métodos para descrição de projetos de grandes circuitos digitais [10].

1.5.2 Fluxo Típico de Projeto de Circuito Integrados Digitais

Um fluxograma de projeto para circuitos integrados digitais de alta integração em escala - VLSI (*Very Large Scale Integration*) é mostrado na Figura 4. Os blocos sem sombra mostram o nível da representação do projeto, enquanto os blocos com sombra mostram os processos no fluxo.

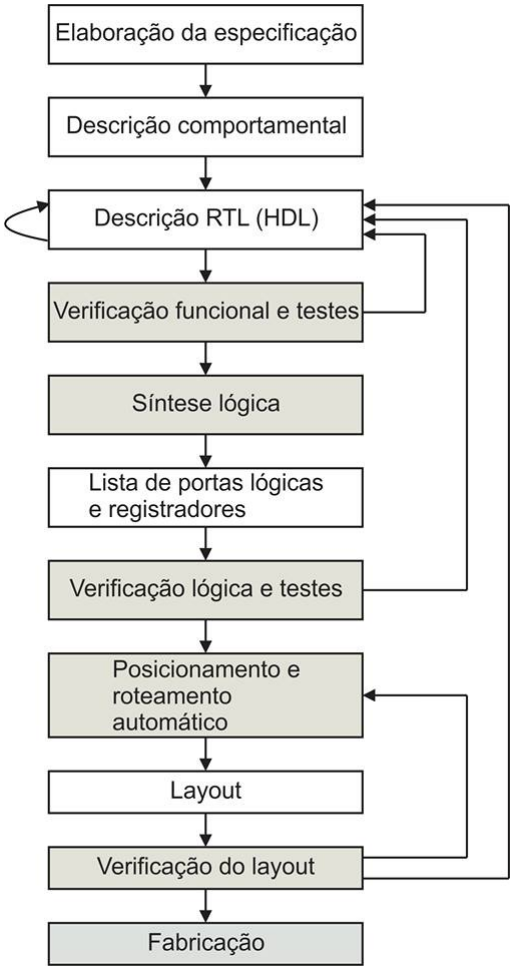

Figura 4: Fluxo de projeto VLSI [10].

O fluxograma apresentado é tipicamente utilizado por projetistas que utilizam HDLs. A primeira parte, em qualquer tipo de projeto de circuito integrado digital, consiste nas elaborações das especificações. Nesta etapa, as funcionalidades, as interfaces, e a arquitetura geral do sistema digital são descritas de forma abstrata, ou seja, os projetistas não precisam pensar em como vão implementar esse circuito. Em seguida, uma descrição comportamental dos circuitos é feita em HDL. A descrição comportamental então é convertida em uma descrição RTL com o auxílio de ferramentas EDA. A partir desta etapa, todo o restante do projeto é feito em ferramentas EDA [10].

Em seguida, inicia-se a etapa de verificação funcional e testes. Neste passo, vetores de testes, mais conhecidos como *testbenches*, são desenvolvidos com finalidade de testar e verificar o comportamento do circuito descrito em HDL. Uma vez finalizado este passo, entra-se na etapa de síntese lógica. A ferramenta de síntese lógica converte a descrição RTL em uma lista de portas lógicas e registradores. Este tipo de arquivo contém informações de conexões entre portas e registradores e é um arquivo de entrada para ferramentas de posicionamento e roteamento automático, responsáveis por gerar o *layout* do circuito integrado. No próximo passo, conhecido como verificação e teste de lógica, os vetores

de testes são mais uma vez utilizados para testar e verificar o circuito produzido na etapa de síntese lógica. Em seguida, entram em ação as ferramentas de posicionamento e roteamento automático para auxiliar no projeto do *layout*. O *layout* então é verificado e o circuito integrado é enviado para a fabricação [10].

Assim, a maioria das atividades dos projetistas de circuitos integrados digitais está concentrada em otimizar a descrição RTL dos circuitos manualmente. Uma vez que estas descrições estão terminadas, as ferramentas de EDA assistem o projetista nos próximos passos. As descrições RTL reduziram o tempo de projeto de anos para alguns meses. Dessa forma, uma maior quantidade de iterações de projetos podem ser realizadas em um espaço de tempo menor. Com o aperfeiçoamento das ferramentas, as atividades de implementação de projetos de circuitos integrados digitais tornaram-se cada vez mais parecidas com linguagens de programação de alto nível. Os projetistas desenvolvem os algoritmos em uma HDL em um nível abstrato e as ferramentas de EDA auxiliam a conversão da descrição comportamental para um circuito integrado pronto para a fabricação [10].

1.5.3 Verilog HDL

O Verilog HDL evolui como uma linguagem de descrição de circuitos padrão. Sua primeira versão aberta é de 1995 com atualizações divulgadas em 2001 e 2005. Em 2005, a linguagem foi normatizada através da norma IEEE 1364-2005. Segundo Palnitkar [10], suas características principais são:

- Verilog HDL é uma linguagem de descrição de hardware de propósito geral que é fácil de se aprender e de usar. Sua sintaxe é similar a linguagem de programação C. Sendo assim, projetistas com experiência em C compreendem Verilog HDL facilmente.
- Verilog HDL permite combinar diferentes níveis de abstração em um mesmo modelo. Dessa forma, um projetista pode definir modelos de hardware em termos de chaves, portas lógicas, RTL ou código comportamental. A mesma linguagem utilizada para descrição também pode ser utilizada para geração de sinais de estímulo para testes.
- A maioria das ferramentas de síntese suportam o Verilog HDL.
- Os fabricantes de circuitos integrados fornecem bibliotecas em Verilog HDL para simulações após síntese lógica.
- A interface processual Verilog - VPI (*Verilog Procedural Interface*) é uma ferramenta poderosa que permite que projetistas escrevam código em linguagem C para interagir com estruturas de dados internas do Verilog HDL.

2 RSA

2.1 O Algoritmo RSA

As publicações de Diffie e Hellman apresentaram uma nova técnica para criptografia, e na realidade, desafiaram os criptologistas a encontrarem um algoritmo que atendesse aos requisitos para o sistema de criptografia assimétrica. Uma das primeiras respostas foi desenvolvida em 1977 por Ron Rivest, Adi Shamir e Len Adleman, no Instituto de Tecnologia de Massachusetts - MIT (*Massachusetts Institute of Technology*), e publicada em 1978. Desde então, o método Rivest-Shamir-Adleman tem reinado soberano como a técnica de uso geral mais aceita e implementada para a criptografia assimétrica. O método do RSA é uma cifra de bloco em que o texto claro e o texto cifrado são números inteiros entre 0 e $n-1$, para algum n. Um tamanho típico para n é 1024 bits, ou seja, n é menor que o número inteiro 2^{1024}. Para o algoritmo RSA, o tamanho chave é definido pelo quantidade de bits necessário para representar o número n, neste caso, tem-se uma chave de 1024 bits. Outros tamanhos de chaves para o RSA são 512, 2048, 4096 e 8192 [7]. A norma da criptografia RSA se encontra atualmente na versão 2.2, descrita no documento intitulado *"PKCS #1 v2.2: RSA Cryptography Standard"*[11].

2.1.1 Descrição do Algoritmo

O método desenvolvido por Rivest, Shamir e Adleman utiliza operações de exponenciação modular. As exponenciações modulares são repetidas multiplicações modulares. O texto claro é sempre cifrado em blocos, com cada bloco contendo um valor binário menor que o número inteiro n, ou seja, o tamanho, em bits, deve ser menor ou igual a $\log_2(n)$. Na prática, o tamanho do bloco é de i bits, onde $2^i < n \leq 2^{i+1}$.

Para gerar uma chave pública e uma respectiva chave privada, dois números primos naturais distintos, denominados p e q, devem ser escolhidos. A multiplicação de p por q resulta em n, que será utilizado como módulo. A quantidade de bits para representação do módulo na base binária representa o tamanho da chave. A função totiente de Euler, $\varphi(n)$, que é utilizada para calcular o número de números naturais menores que n que são primos relativos a n, é descrita pela expressão:

$$\varphi(n) = (p-1) \times (q-1) \tag{2.1}$$

Em seguida, deve-se escolher um número natural e, tal que e seja $1 < e < \varphi(n)$ e que o maior divisor comum de e e $\varphi(n)$ seja igual a 1. Depois, o um número natural d,

que é o multiplicativo inverso de e módulo $\varphi(n)$ é calculado através da expressão:

$$d \times e \equiv 1 \; mod \; (\varphi(n)) \tag{2.2}$$

A técnica mais utilizada para resolver esta expressão é através do algoritmo estendido de Euclides. A chave pública é definida como o par de números naturais (e, n). A chave privada é definida como o par de números naturais (d, n).

A chave pública, como o próprio nome já diz, deve ser de conhecimento público, enquanto a chave privada deve ser guardada para prevenção de ataques. Para exemplificar o processo de cifragem e decifragem, coloca-se no cenário novamente os personagens Alice e Bob. Para este caso, digamos que Bob deseja enviar uma mensagem para Alice.

2.1.2 Cifragem

Para fazer o processo de cifragem, Alice transmite sua chave pública (e, n) para Bob e mantém sua chave privada em segredo. Primeiramente, Bob transforma a mensagem M (texto claro) em um número natural m em blocos de tamanho $0 < m < n$. O próximo passo consiste em calcular a Equação 2.3.

$$c = m^e \; mod \; n \tag{2.3}$$

Esta operação pode ser feita rapidamente usando os métodos de exponenciação por elevação quadrática. Bob transmite c (texto cifrado) para Alice.

2.1.3 Decifragem

Para Alice recuperar a mensagem m (texto claro) de c, ela utiliza o sua chave privada (d, n). Primeiramente, caso a mensagem seja maior que o número n, Alice divide a mensagem em blocos de tamanho $0 < m < n$. Em seguida, para cada bloco, Alice deve calcular a Equação 2.4.

$$m = c^d \; mod \; n \tag{2.4}$$

Dessa forma, o texto claro que Bob enviou para Alice pode ser lido.

2.1.4 Exemplo

Um exemplo obtido de [7] aparece na Figura 5. Para este exemplo, as chaves foram geradas da seguinte forma:

1. Selecionam-se dois números primos distintos, $p = 17$ e $q = 11$.

2. Calcula-se $n = p \times q = 17 \times 11 = 187$.
3. Calcula-se $\varphi(n) = (p-1) \times (q-1) = 16 \times 10 = 160$.
4. Seleciona-se e tal que e seja relativamente primo a $\varphi(n) = 160$ e menor que $\varphi(n)$: escolheu-se $e = 7$.
5. Determina-se d tal que $d \times e \equiv 1 \; mod \; 160$ e $d < 160$. O valor correto é $d = 23$, pois $23 \times 7 \equiv 161 \equiv 1 \times 160 + 1$.

As chaves resultantes são a chave pública $(7, 187)$ e a chave privada $(23, 187)$. O exemplo mostra o uso dessas chaves para uma entrada de texto claro $m = 88$. Para o processo de cifragem, deve-se calcular a Equação 2.5.

$$c = 88^7 \; mod \; 187 \tag{2.5}$$

Explorando algumas propriedades da aritmética modular, pode-se realizar os seguintes passos:

$88^7 \; mod \; 187 = [(88^4 \; mod \; 187) \times (88^2 \; mod \; 187) \times (88^1 \; mod \; 187)] \; mod \; 187$

$88^1 \; mod \; 187 = 88$

$88^2 \; mod \; 187 = 7744 \; mod \; 187 = 77$

$88^4 \; mod \; 187 = 59,969,536 \; mod \; 187 = 132$

$88^7 \; mod \; 187 = (88 \times 77 \times 132) \; mod \; 187 = 894,432 \; mod \; 187 = 11$

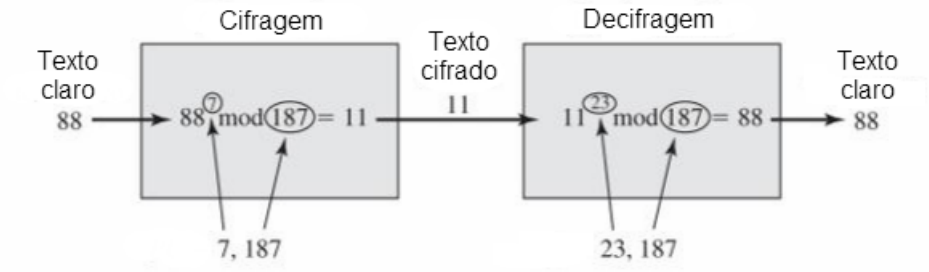

Figura 5: Exemplo de algoritmo RSA [7].

Para a decifragem, deve-se calcular a Equação 2.6.

$$m = 11^{23} \; mod \; 187 \tag{2.6}$$

Novamente, utilizando propriedades da aritmética modular, obtem-se:

$11^{23} \; mod \; 187 = [(11^1 \; mod \; 187) \times (11^2 \; mod \; 187) \times (11^4 \; mod \; 187) \times (11^8 \; mod \; 187) \times$
$(11^8 \; mod \; 187)] \; mod \; 187$

$11^1 \; mod \; 187 = 11$

$11^2 \; mod \; 187 = 121$

$11^4 \; mod \; 187 = 14,641 \; mod \; 187 = 55$

$11^8 \; mod \; 187 = 214,358,881 \; mod \; 187 = 33$

$11^{23} \; mod \; 187 = (11 \times 121 \times 55 \times 33 \times 33) \; mod \; 187 = 79,720,245 \; mod \; 187 = 88$

A multiplicação modular é considerada uma operação aritmética complicada de ser realizada em hardware por possuir inerentemente as operações de multiplicação e divisão. A literatura apresenta alguns algoritmos para realizar estas operações de forma eficiente [1, 7]. Um algoritmo proposto por P. L. Montgomery, que é apresentado neste trabalho, realiza o processo de multiplicação modular através de somas e deslocamentos, facilitando o desenvolvimento de estruturas de software e hardware para cálculo de exponenciações modulares.

2.1.5 A segurança do RSA

Segundo Stallings [7], algumas abordagens possíveis para atacar o algoritmo RSA são:

- **Força bruta:** envolve em tentar todas as chaves privadas possíveis.
- **Ataques matemáticos:** esforços para fatorar o módulo e recuperar os primos utilizados na geração da chave.
- **Ataques de temporização (timing attack):** dependem do tempo de execução do algoritmo de decifragem.
- **Ataques de texto cifrado escolhido:** explora o texto cifrado junto com as propriedades do algoritmo RSA.
- **Ataques de monitoramento de potência (power analysis):** estudo do consumo de energia de um hardware criptográfico para obter informações da chave.

A segurança do RSA reside na dificuldade de fatorar de forma eficiente números inteiros muito grandes. Dessa forma, quanto maior o tamanho da chave, maior a segurança do sistema criptográfico. Em contra partida, quanto maior a chave, mais lento é o sistema. O equilíbrio entre o nível de segurança desejado e a quantidade de informação processada por unidade de tempo é característico para cada tipo de aplicação a ser utilizada no sistema criptográfico.

Quanto aos ataques físicos, novas técnicas de implementação do algoritmo em hardware ainda se encontram em desenvolvimento para evitar que as informações das

chaves e dos resultados intermediários possam revelar dados secretos [7]. Atualmente, o RSA Laboratories recomenda a utilização de chaves para ambientes corporativos com tamanho de 1024 bits [12].

2.2 Modelagem

A multiplicação modular é considerada uma operação aritmética de difícil implementação em hardware por possuir inerente ao seu processo as operações de multiplicação e divisão. Existem duas possibilidades para calcular a multiplicação modular: (1) aplicando o módulo após a multiplicação, ou seja, no produto, ou (2) aplicando o módulo durante a operação de multiplicação. A operação módulo encontra o resto da divisão de um número inteiro por outro. A primeira possibilidade requer um circuito multiplicador de n x n bits com um registro de tamanho $2n$ bits seguido por um circuito divisor de tamanho $2n \times n$ bits. Na segunda possibilidade, a operação de módulo é executada a cada passo da iteração da multiplicação inteira. Dessa forma, a primeira abordagem requer mais circuitos em hardware, enquanto a segunda abordagem requer mais cálculos de somas/subtrações [13].

Os algoritmos de multiplicação modular ordinária para o cálculo de $A \times B \; mod \; M$ utilizam o método de acumulação de dígitos provenientes do produto $A \times b_i$ e intermediariamente acontece a redução modular para manter o resultado menor que M. Estas reduções são realizadas subtraindo o módulo M ou seus múltiplos do resultado intermediário e são dependente dos bits mais significativos do operando [13].

Entretanto, no escopo da aritmética computacional, o algoritmo de Montgomery, apresentado em 1985 por Peter Montgomery, permite que operações de aritmética modular sejam realizadas de forma eficiente para valores de módulo muito grande, tipicamente com várias centenas de bits [13].

2.2.1 Algoritmo de Montgomery

O algoritmo de Montgomery [14] inverte a ordem de tratar os dígitos do multiplicando. O procedimento consiste na utilização dos bits menos significativos do resultado intermediário para realizar uma adição ao invés de uma subtração, e uma operação de deslocamento para a direita ao invés de uma operação de deslocamento para a esquerda a cada iteração [13].

Duas versões parcialmente modificadas do algoritmo de Montgomery são apresentadas por Nedjah[15]. Os parâmetros para os algoritmos são:

$$A = \sum_{i=0}^{k-1} a_i 2^i, \quad B = \sum_{i=0}^{k-1} b_i 2^i, \quad M = \sum_{i=0}^{k-1} m_i 2^i \tag{2.7}$$

$$a_i, b_i, m_i \in \{0, 1\} \tag{2.8}$$

e dado que o módulo M, é, no máximo, um inteiro composto de k bits (isto é, $0 < M < 2^k$), e $A, B < M$, então a quantidade de bits do multiplicador, n, é definida com o valor de $k + 2$ [13].

É necessário ter o multiplicador com a quantidade de bits de $k+2$ para garantir que o resultado intermediário, S, permaneça dentro do intervalo $0 < S < 2^k$. Esta condição permite que o resultado intermediário do algoritmo possa ser utilizado como entrada para a próxima iteração do algoritmo [13]. O Algoritmo 1 ilustra o conceito desenvolvido por Montgomery.

Algorithm 1 Pseudocódigo Algoritmo de Montgomery Modificado 1 - $MonPro1(A, B, M)$ - [15]

```
1: int S = 0;
2: for i = 0 to n − 1 do
3:     S = S + a_i × B;
4:     if (s_0 = 0) then                ▷ (Bit menos significativo de S)
5:         S = S/2;
6:     else
7:         S = (S + M)/2;
8:     end if
9: end for
10: return S;
```

No Algoritmo 1, o cálculo do produto modular é realizado através de acumulação de produto entre os bits e intermediariamente acontece a redução modular. O cálculo se inicia na multiplicação do bit menos significativo a_0 pelo número B e este valor é armazenado na variável S. Uma vez finalizado este produto, dá-se início a etapa de redução modular. Esta etapa consiste na análise do bit menos significativo da variável S, ou seja, s_0. Para o caso $s_0 = 0$, a redução modular é realizada através da divisão da variável S pelo número dois. Em lógica binária o processo de divisão por dois é realizado através de um deslocamento de um bit da variável S para direita. Para o caso $s_0 = 1$, a redução modular é realizada através da soma da variável S com o módulo M e, somente com este novo valor, ocorre a divisão pelo número dois, ou seja, faz-se um deslocamento de um bit para a direita. As próximas iterações realizam os mesmos passos, com os valores atualizados da variável S, até que a variável de iteração i chegue ao seu valor final. Ao final deste processo, a variável S é o resultado da multiplicação modular de Montgomery.

O Algoritmo 2, possui uma pequena modificação, mas realiza a mesma operação.

No Algoritmo 2, o cálculo do bit menos significativo de S, s_0, é armazenado na variável q_i, onde i é a variável de iteração do algoritmo. Como não é necessário realizar

Algorithm 2 Pseudocódigo Algoritmo de Montgomery Modificado 2 - $MonPro2(A, B, M)$ - [15]

1: **int** $S = 0$;
2: **bit** $q = 0$;
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad q_i = (s_0 + a_i \times b_0)\ mod\ 2$;
5: $\quad S = (S + a_i \times B + q_i \times M)/2$;
6: **end for**
7: **return** S;

a multiplicação completa entre a_i e B, ilustrada na linha quatro do Algoritmo 1, para se obter o valor do bit menos significativo da multiplicação, uma forma mais rápida de realizar a mesma operação é dada pela linha três do Algoritmo 2. Para a iteração 0, o valor de s_0 é igual a 0, enquanto para as demais iterações, o valor depende dos cálculos anteriores. A etapa da redução modular do Algoritmo 2 ocorre na linha cinco.

Primeiramente, deve-se adicionar o valor da multiplicação de a_i e B à S, obtendo assim o mesmo valor que a expressão da linha três do Algoritmo 1. Como a redução modular consiste na adição ou não do módulo M, tem-se o bit q_i realizando a máscara da variável M para esta soma no Algoritmo 2, que representa a mesma decisão apresentada entre as linhas quatro a oito do Algoritmo 1. Para o caso do bit $q_i = 0$, a expressão da redução modular no Algoritmo 2 é $S = (S + a_i \times B)/2$, semelhante a linha cinco do Algoritmo 1. Para o caso do bit $q_i = 1$, a expressão da redução modular no Algoritmo 2 é $S = (S + a_i \times B + q_i \times M)/2$, semelhante a linha sete do Algoritmo 1.

A partir da análise realizada em ambos os Algoritmos 1 e 2, percebe-se que a multiplicação modular de Montgomery é reduzida a operações simples de soma e de deslocamento de bits. Entretanto, o resultado da execução desses algoritmos é dado por:

$$MonPro(A, B, M) = A \times B \times r^{-1}\ mod\ M \tag{2.9}$$

Como o fator extra r^{-1} sempre é gerado no cálculo, um pré-cálculo e um pós-cálculo são necessários para produzir o resultado correto para uma multiplicação modular $A \times B\ mod\ M$ [15]. Aqui, r^{-1} é o multiplicativo inverso de $(r\ mod\ M)$, isto é:

$$r^{-1} \times r = 1\ mod\ M \tag{2.10}$$

onde r é dado por:

$$r = 2^n. \tag{2.11}$$

Uma técnica robusta para eliminar o termo indesejado, presente em quase todas

as literaturas [13, 15, 16, 17, 18, 19, 20], pode ser empregada como auxílio para apresentar o resultado correto de uma multiplicação modular. Para se obter os resultados corretos da multiplicação modular de um número inteiro $\mathcal{A}$ por $\mathcal{B}$ módulo M, deve-se realizar um mapeamento de número inteiro $\mathcal{A}$ no domínio ordinário para o seu equivalente m-resíduo A no domínio Montgomery [13]. Esse processo de transformação de um domínio ordinário para o domínio Montgomery é justificado pelas equações apresentados a seguir.

O m-resíduo A, de um inteiro $\mathcal{A} < M$ é definido como:

$$A = \mathcal{A} \times r \ mod \ M. \tag{2.12}$$

O próprio algoritmo de Montgomery pode ser utilizado para converter um inteiro para o seu m-resíduo, ou seja, fazer o mapeamento de um número inteiro no domínio ordinário para o domínio Montgomery, da seguinte forma [13]:

$$MonPro(\mathcal{A}, r^2, M) = \mathcal{A} \times r^2 \times r^{-1} \ mod \ M \tag{2.13}$$

$$MonPro(\mathcal{A}, r^2, M) = \mathcal{A} \times r \ mod \ M \tag{2.14}$$

$$MonPro(\mathcal{A}, r^2, M) = A \tag{2.15}$$

Conforme demonstrado na Equação 2.15, para converter um número inteiro $\mathcal{A}$ no domínio ordinário para seu m-resíduo, A no domínio Montgomery, é necessário calcular $MonPro(\mathcal{A}, r^2, M)$. Dessa forma, o fator r^2 passa a ser representado pela Equação 2.16.

$$r^2 = 2^{2n} \tag{2.16}$$

Entretanto, o valor 2^{2n} está fora da faixa permitida de entrada de parâmetro para o algoritmo de Montgomery, cuja limitação é dada pelo conjunto $(0 < M < 2^{n-2})$. Sendo assim, uma alternativa para esta operação pode ser feita através do cálculo da Equação 2.17 [13].

$$MonPro(\mathcal{A}, 2^{2n} \ mod \ M, M) \tag{2.17}$$

O valor da Equação 2.17 pode ou não ser pré-calculado externamente, isto é, a escolha da implementação de um circuito que realize a operação da Equação 2.17 fica a critério dos parâmetros do projeto. Porém, este valor é constante para um determinado tamanho de bits e o módulo M, e pode ser idealmente armazenado no banco de dados junto com uma chave pública [13].

A partir de outra análise do resultado de ambos os Algoritmos 1 e 2, pode-se verificar, também, que o produto Montgomery de dois m-resíduos, A, B, gera um próprio m-resíduo, S.

$$S = MonPro(A, B, M) \tag{2.18}$$

$$S = A \times B \times r^{-1} \ mod \ M \tag{2.19}$$

Substituindo os m-resíduos, A e B, na Equação 2.19 pelos seus respectivos valores, $(\mathcal{A} \times r \ mod \ M)$ e $(\mathcal{B} \times r \ mod \ M)$, e aplicando propriedades da aritmética modular, tem-se como resultado a Equação 2.22.

$$S = (\mathcal{A} \times r \ mod \ M) \times (\mathcal{B} \times r \ mod \ M) \times (r^{-1} \ mod \ M) \tag{2.20}$$

$$S = \mathcal{A} \times \mathcal{B} \times r \ mod \ M \tag{2.21}$$

$$S = \mathcal{S} \times r \ mod \ M \tag{2.22}$$

Utilizando novamente a aritmética modular e suas propriedades, pode-se manipular a Equação 2.22 de forma a reescrevê-la com o termo isolado $\mathcal{S}$, conforme demonstrado na Equação 2.23.

$$\mathcal{S} = S \times r^{-1} \ mod \ M \tag{2.23}$$

Ao inserir o número inteiro 1 entre o sinal de igual e o termo S na Equação 2.23, recupera-se o formato do resultado dos Algoritmos 1 e 2, conforme representado pela Equação 2.24. Sendo assim, o cálculo final necessário para converter o S de volta para a forma inteira ordinária, $\mathcal{S}$, é representado pela Equação 2.25 [13].

$$\mathcal{S} = 1 \times S \times r^{-1} \ mod \ M \tag{2.24}$$

$$\mathcal{S} = MonPro(1, S, M) \tag{2.25}$$

A pré-condição para que estes algoritmos funcionem é que o módulo, M, seja primo relativo a raiz, r (isto é: $mdc(M, r) = 1$). Este é sempre o caso no sistema criptográfico RSA uma vez que $M = p \times q$, produto de dois primos grandes, e portanto é ímpar. E como r é uma potência de 2, seu valor é sempre par [13].

Devido ao fato de que as operações de $mod\ r$ e $div\ r$ são intrinsecamente mais rápidas em sistemas binários e uma vez que r é uma potência de 2, o método do algoritmo de Montgomery é mais rápido e fácil de calcular que o método normal da multiplicação modular a expressão $\mathcal{A} \times \mathcal{B}\ mod\ M$. Entretanto, as operações adicionais de conversão para o formato m-resíduo e o processo inverso, e o pré-cálculo de $(2^{2n}\ mod\ M)$ incluem novos passos. Assim, não é vantajoso utilizar a técnica de Montgomery quando somente uma multiplicação modular é necessária. A utilização da técnica de Montgomery é mais apropriada quando muitas multiplicações com o mesmo módulo devem ser realizadas, como por exemplo, o algoritmo de exponenciação modular do RSA [13].

De forma a validar os passos do Algoritmo 2, fez-se uma implementação em linguagem de programação C e Java. Verificou-se passo a passo os processos intermediários do algoritmo, bem como o seu resultado final. O software desenvolvido em linguagem C, por limitação das variáveis utilizadas, ficou restrito a operações de multiplicação modular com operandos de no máximo de 16 bits. Para contornar esta limitação, utilizou-se a linguagem Java pois a mesma possui uma classe de representação numérica que comporta números inteiros muito grandes. No software desenvolvido em Java foram realizadas operações com números de até 1024 bits com sucesso. Ambas implementações se encontram nos apêndices.

2.2.2 Exponenciação Modular

A exponenciação modular é realizada através de sucessivas multiplicações modulares. Existem dois algoritmos comuns que podem ser utilizados: o método binário Esquerda-Direita, otimizado em área, e o método binário Direita-Esquerda, otimizado em velocidade. Os métodos são explicados através dos Algoritmos 3 e 4, onde o parâmetro $\mathcal{P}$ é o texto claro, $\mathcal{E}$ é o expoente e M é o módulo. Em ambos os algoritmos a constante $Constm$, que é o resultado da expressão $2^{2n}\ mod\ M$ deve ser previamente calculada e $\mathcal{R}$ representa o resultado.

No Algoritmo 3, a primeira coisa a ser feita é calcular a constante $Constm$, representado pela linha um, para prosseguir com próximas operações. As linhas dois e três do Algoritmo 3 representam o mapeamento dos números inteiros $\mathcal{P}$ do domínio ordinário para a variável P no domínio Montgomery, e do valor inicial unitário da variável R no domínio Montgomery. Em seguida, inicia-se o processo de iteração e, de fato, acontece a exponenciação modular no domínio Montgomery. Vale ressaltar que neste algoritmo, os bits do expoente $\mathcal{E}$ são percorridos da esquerda para a direita, conforme a inicialização da variável de iteração i do algoritmo, na linha quatro.

Em seguida, sempre é realizada a operação de potenciação ao quadrado da linha cinco e a multiplicação da linha sete é condicionada ao bit do expoente da respectiva iteração. A cada passo, a variável i é decrementada e o processo iterativo finaliza quando

Algorithm 3 Pseudocódigo Algoritmo de Exponenciação Modular Esquerda-Direita - $MonExp1(\mathcal{P}, \mathcal{E}, M)$

```
1: int Constm = 2^{2n} mod M;
2: int P = MonPro(Constm, P, M);          ▷ Mapeamento
3: int R = MonPro(Constm, 1, M);          ▷ Mapeamento
4: for i = k − 1 down to 0 do
5:     R = MonPro(R, R, M);               ▷ Potenciação
6:     if (E_i = 1) then
7:         R = MonPro(R, P, M);           ▷ Multiplicação
8:     end if
9: end for
10: R = MonPro(1, R, M);                  ▷ Remapeamento
11: return R;
```

a variável i atinge o valor 0. Após esta etapa, deve-se realizar o remapeamento do resultado R, no domínio Montgomery, para o domínio ordinário, $\mathcal{R}$. Este processo é realizado na linha dez do Algoritmo 3.

É importante ressaltar que neste algoritmo as operações de potenciação ao quadrado e a multiplicação devem ser realizadas sequencialmente. O motivo para isto acontecer se deve ao fato de que o cálculo da linha sete necessita do valor calculado na linha cinco, ou seja, há uma dependência do resultado da variável R para realizar a operação de forma correta. Isso implica que a potenciação ao quadrado e a multiplicação podem ser realizadas em um único multiplicador em hardware, resultando em uma menor área de circuito. O Algoritmo 4, apresentado a seguir, é a segunda técnica a ser discutida neste trabalho. Assim como o Algoritmo 3, este também realiza a operação de exponenciação modular.

Algorithm 4 Pseudocódigo Algoritmo de Exponenciação Modular Direita-Esquerda - $MonExp2(\mathcal{P}, \mathcal{E}, M)$

```
1: int Constm = 2^{2n} mod M;
2: int P = MonPro(Constm, P, M);          ▷ Mapeamento
3: int R = MonPro(Constm, 1, M);          ▷ Mapeamento
4: for i = 0 to k − 1 do
5:     if (E_i = 1) then
6:         R = MonPro(R, P, M);           ▷ Multiplicação
7:     end if
8:     P = MonPro(P, P, M);               ▷ Potenciação
9: end for
10: R = MonPro(1, R, M);                  ▷ Remapeamento
11: return R;
```

Analogamente ao Algoritmo 3, a etapa do mapeamento dos operandos de entrada no domínio ordinário para o domínio Montgomery são idênticos para Algoritmo 4. No Al-

goritmo 4, também são realizadas operações de potenciação ao quadrado e multiplicação, como no Algoritmo 3, mas com algumas diferenças.

A primeira e principal diferença é o sentido em que os bits são percorridos através da variável de iteração i. Enquanto no Algoritmo 3 a variável i era iniciada com o valor $k-1$ e decrementada a cada iteração, no Algoritmo 4 esse processo é invertido, o valor inicial da variável i é com a constante 0 e seu valor é incrementado em uma unidade ao final de cada passo. Devido a este característica o Algoritmo 4 é denominado Direita-Esquerda, contrariamente ao Algoritmo 3, denominado Esquerda-Direita.

A segunda diferença é a ordem das operações de multiplicação e potenciação ao quadrado. Enquanto no Algoritmo 3, primeiro é realizada a operação de potenciação ao quadrado, no Algoritmo 4 primeiro é realizado o teste do bit do expoente $\mathcal{E}$ e feita a operação de multiplicação caso o bit $\mathcal{E}_i$ seja verdadeiro e em seguida, sempre é realizada a operação de potenciação ao quadrado. No término da iteração da variável i, ou seja, no final da execução do algoritmo, deve-se fazer o remapeamento do resultado R no domínio Montgomery para o domínio ordinário, $\mathcal{R}$, representado na linha dez.

Vale ressaltar que no Algoritmo 4 as operações de multiplicação e potenciação ao quadrado são independentes, ou seja, não existe uma dependência da operação de potenciação ser realizada somente após a multiplicação. Sendo assim, estas operações podem ser realizadas paralelamente. Tanto a potenciação ao quadrado quanto a multiplicação só dependem dos resultados anteriores à última iteração da variável i do algoritmo. Dessa forma reduz-se em 50% o número de ciclos de clock necessários para completar a operação de exponenciação modular, quando comparado com o Algoritmo 3. Entretanto, devem existir dois multiplicadores em hardware para alcançar tal velocidade. Sendo assim, o produto velocidade $\times$ área de ambos os algoritmos são bem similares.

O Algoritmo 4 foi escolhido para a implementação da exponenciação modular em hardware por apresentar a característica de paralelismo entre as duas operações de multiplicação modular. Dessa forma, o algoritmo escolhido está alinhando com a proposta do trabalho de desenvolver um circuito criptográfico em hardware com maior *throughput* de dados. Para validar os passos do Algoritmo 4, fez-se uma implementação em linguagem de programação C e Java. Verificou-se passo a passo os processos intermediários do algoritmo, bem como o seu resultado final. O software desenvolvido em linguagem C, por limitação das variáveis utilizadas, ficou restrito a operações de exponenciação modular com operandos de no máximo de 16 bits. Para contornar esta limitação, utilizou-se a linguagem Java pois a mesma possui uma classe de representação numérica que comporta números inteiros muito grandes. No software desenvolvido em Java foram realizadas operações com números de até 1024 bits com sucesso. Ambas implementações se encontram nos apêndices.

2.2.3 Algoritmo de Montgomery Modificado em Lógica Binária

Considerando a expressão $(S+a_i \times B+q_i \times M)$ presente na linha cinco do Algoritmo 2, o resultado da expressão depende dos valores das variáveis a_i e q_i a cada iteração do laço do comando *for*. Os possíveis resultados para esta expressão são apresentados na Tabela 2.

Tabela 2: Resultado da expressão da linha cinco do Algoritmo 2.

a_i	q_i	$S + a_i \times B + q_i \times M$
0	0	S
1	0	$S + B$
0	1	$S + M$
1	1	$S + MB$[1]

[1]MB é o resultado de $M + B$

A partir da Tabela 2, uma versão em lógica binária do algoritmo de Montgomery é apresentado no Algoritmo 5. A linha quatro do Algoritmo 2 se transforma na linha cinco do Algoritmo 5. A operação de $a_i \times b_0$ consiste de uma máscara aplicada no bit b_0 a cada iteração do valor de i. A soma dos bits $a_i \times b_0$ com o bit s_0 é feita através do circuito digital que tem a propriedade de soma, ou seja, através de uma porta ou-exclusivo de duas entradas. Como a expressão completa é, na verdade, $s_0 + a_i \times b_0 \; mod \; 2$, o próprio circuito ou-exclusivo já realiza a operação modular para este caso, não sendo necessário nenhuma lógica adicional.

A linha cinco do Algoritmo 2 se transforma no trecho compreendido entre as linhas seis a 14 do Algoritmo 5. Neste caso, o Algoritmo 5 expande a representação dos operandos em bits para visualização da soma através de circuitos somadores completos de um bit. O comando que percorre todos os bits dos operandos é o *for* da linha seis, através da variável j. O trecho entre as linhas sete a 11 refletem a seleção da parcela a ser somada com o valor anterior do produto modular, enquanto as linhas 12 e 13 representam a soma bit a bit dos operandos e os sinais de propagação de *carry* dos somadores completos de um bit.

Assumindo o Algoritmo 5 como base é possível construir uma arquitetura em forma de matriz, conforme ilustrada na Figura 6. Para esta arquitetura em forma de matriz, a variável i representa as linhas, enquanto a variável j representa as colunas.

Os elementos processadores da arquitetura em formato matricial do multiplicador modular de Montgomery calculam os bits s_j do resíduo S. Isto representa o cálculo da expressão $s_j^{(i+1)} = s_{j+1}^{(i)}$ ***xor*** $x_j^{(i)}$ ***xor*** $carry$ do Algoritmo 5 e o $carry$ de propagação $(s_{j+1}^{(i)}$ ***and*** $x_j^{(i)})$ ***or*** $(s_{j+1}^{(i)}$ ***and*** $carry)$ ***or*** $(x_j^{(i)}$ ***and*** $carry)$ para os elementos vizinhos. A arquitetura do elemento processador básico, isto é, as $células_{i,j}$, onde $1 \leq i \leq n-1$ e $1 \leq j \leq n-1$, é ilustrada na Figura 7. Esta arquitetura implementa as instruções das linhas seis a 14 do Algoritmo 5, que consistem de um somador completo e um multiplexador de

Algorithm 5 Pseudocódigo Algoritmo de Montgomery Modificado em Lógica Binária - $SysMon(A, B, M, MB)$

1: **int** $S = 0$;
2: **bit** x;
3: **bit** $carry = 0$;
4: **for** $i = 0$ to n **do**
5: $q_i = s_0^{(i)}$ ***xor*** $(a_i$ ***and*** $b_0)$
6: **for** $j = 0$ to n **do**
7: **switch** (a_i, q_i)
8: $1, 1 : x_j^{(i)} = mb_j$;
9: $1, 0 : x_j^{(i)} = b_j$;
10: $0, 1 : x_j^{(i)} = m_j$;
11: $0, 0 : x_j^{(i)} = 0$;
12: $s_j^{(i+1)} = s_{j+1}^{(i)}$ ***xor*** $x_j^{(i)}$ ***xor*** $carry$;
13: $carry = (s_{j+1}^{(i)}$ ***and*** $x_j^{(i)})$ ***or*** $(s_{j+1}^{(i)}$ ***and*** $carry)$ ***or*** $(x_j^{(i)}$ ***and*** $carry)$;
14: **end for**
15: **end for**
16: **return** S;

quatro canais de um bit.

A arquitetura do elemento processador da primeira linha e da primeira coluna, isto é, a $célula_{0,0}$, é ilustrada na Figura 8. Além do cálculo das linhas seis a 14 do Algoritmo 5, este elemento processador também realiza o cálculo do valor de q_i na linha cinco. Entretanto, como $s_0^{(0)}$ é 0 (iteração $i = 0$ do algoritmo), o cálculo de q_i fica reduzido a expressão $(a_i$ ***and*** $b_0)$. Além disso, um somador completo de um bit não se faz necessário porque o $carry$ de entrada também é igual a 0, então a expressão $s_1^{(0)}$ ***xor*** x_0^0 ***xor*** $carry$ fica reduzida a x_0^0 e a expressão $(s_1^{(0)}$ ***and*** $x_0^0)$ ***or*** $(s_1^{(0)}$ ***and*** $carry)$ ***or*** $(x_0^0$ ***and*** $carry)$ fica reduzida a 0.

Os elementos processadores restantes da primeira coluna da arquitetura fazem o mesmo processamento que os elementos processadores básicos e, assim como o elemento processador da primeira linha e da primeira coluna, calcula o bit q_i porém para o caso geral, isto é, quando s_0^i não é nulo. Além disso, um somador completo de um bit pode ser substituído por um meio somador de um bit uma vez que o sinal de $carry$ para estes elementos são nulos. Uma ilustração da arquitetura destes elementos processadores é apresentada na Figura 9.

A arquitetura dos elementos processadores da primeira linha, $células_{0,j}$, excluindo a $célula_{0,0}$ é dada pela Figura 10. Como $s_n^{(i)} = 0$, não se justifica a utilização de um somador completo de um bit. Sendo assim, o mesmo é substituído por um meio somador.

O cálculo de $M+B$ é realizado uma única vez no início do processo de multiplicação modular. Esta operação pode ser feita através de uma cadeia de somadores completos

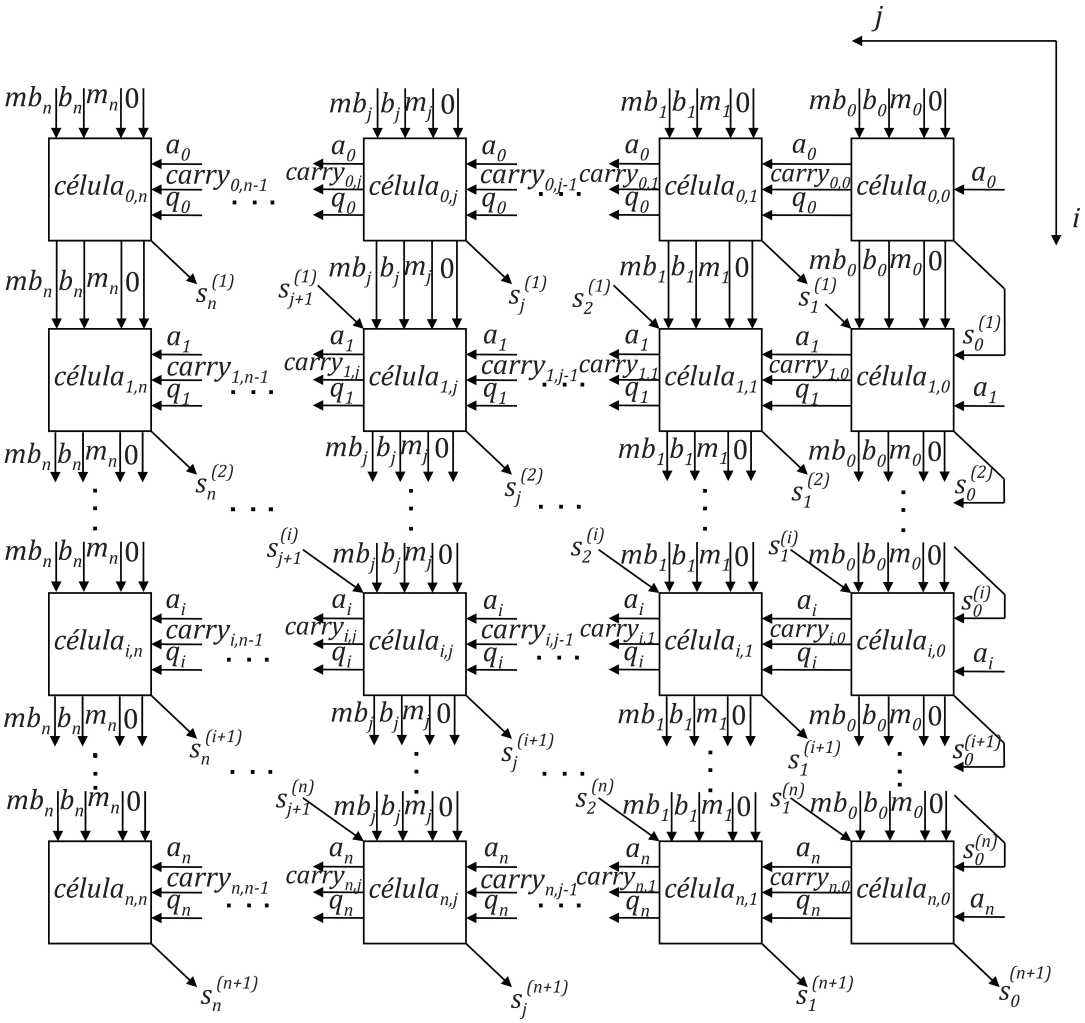

Figura 6: Arquitetura em formato matricial do multiplicador modular de Montgomery [15].

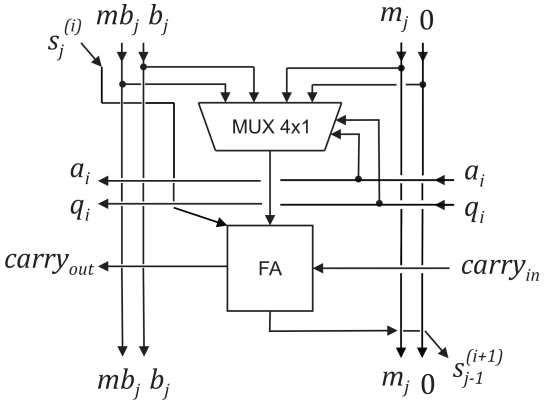

Figura 7: Arquitetura do elemento processador básico [15].

conectados serialmente ou alguma outra arquitetura de somadores mais eficiente [15].

Cada linha da arquitetura é responsável pelo cálculo da expressão $(S + a_i \times B + q_i \times M)$ a cada iteração da variável i do Algoritmo 2. Entretanto, a expressão completa consiste em $(S + a_i \times B + q_i \times M)/2$. Em lógica binária, a divisão pelo número dois

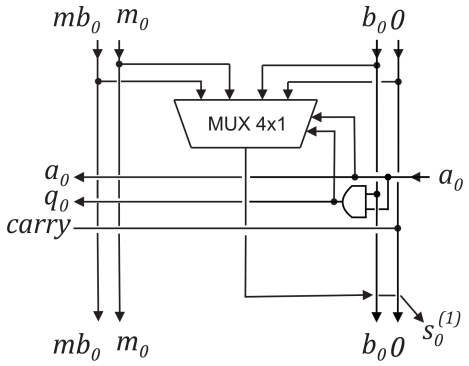

Figura 8: Arquitetura do elemento processador da primeira linha e da primeira coluna - $célula_{0,0}$ [15].

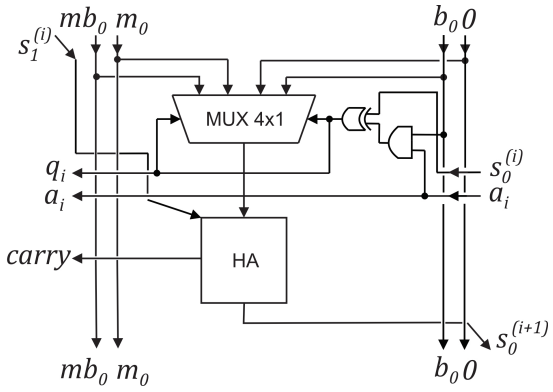

Figura 9: Arquitetura dos elementos processadores da coluna da direita - $células_{j,0}$ [15].

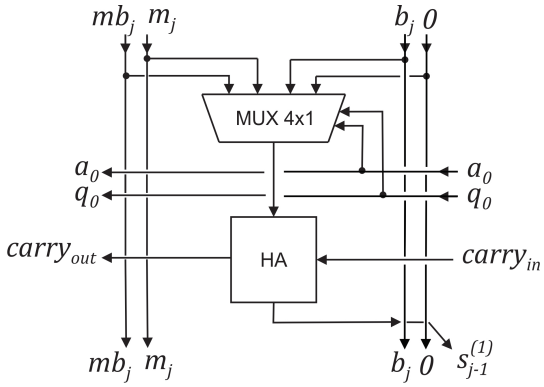

Figura 10: Arquitetura dos elementos processadores da primeira linha - $células_{0,j}$ [15].

representa o deslocamento de um bit para a direita. Sendo assim, o valor do S que é entrada para a linha seguinte da matriz de elementos processadores é deslocado de um bit para a direita. Dessa forma, o cálculo é realizado de acordo com o Algoritmo 2.

Para validar a arquitetura sistólica apresentada na Figura 6, fez-se uma implementação em Verilog HDL para quatro e oito bits. Verificou-se os estágio intermediários dos valores de S, ou seja, os valores da variável S em cada uma das linhas da matriz de elementos processadores, bem como o seu resultado final. Os softwares desenvolvidos também verificaram dos valores intermediários e finais.

Por fim, conclui-se que a arquitetura apresentada realizou as operações com números de até oito bits com sucesso. Como os hardwares desenvolvidos para quatro e oito bits foram realizados através de processos modulares, em que a diferença entre eles consiste somente no aumento da quantidade de bits, pôde-se concluir que a arquitetura proposta por Nedjah [15] realiza a operação de multiplicação modular de Montgomery conforme apresentada pelos Algoritmos 1, 2 e 5.

2.2.4 Discussão da Arquitetura Matricial

A arquitetura apresentada por Nedjah [15] possui como característica principal o cálculo do algoritmo de multiplicação modular de Montgomery através de uma lógica simples e intuitiva. Através da matriz de elementos processadores apresentada, o resultado da multiplicação modular de Montgomery é realizado com multiplexadores, portas lógicas simples de duas entradas e circuitos somadores, ora com somadores completos de um bit ora com meio somadores de um bit.

Para uma arquitetura de n bits, a quantidade necessária de elementos processadores para apresentar o resultado correto da operação é igual a $(n+2) \times (n+2)$, sendo então formada uma matriz de $(n+2)$ linhas por $(n+2)$ colunas. Sendo assim, a quantidade de elementos processadores para arquiteturas de 4, 8, 16, 32, 64, 128, 256, 512 e 1024 bits é apresentada na Tabela 3.

Tabela 3: Relação entre a quantidade de bits e a quantidade de elementos processadores.

Quantidade de bits	**Quantidade de elementos processadores**
4	36
8	100
16	324
32	1156
64	4356
128	16900
256	66564
512	264196
1024	1052676

Analisando a Tabela 3, nota-se que a quantidade de elementos processadores aumenta com ordem quadrática através da expressão $(n+2)^2$. A primeira conclusão a ser elucidada se refere a quantidade elevada de elementos processadores para realizar uma operação. Segundo o RSA Laboratories [12], atualmente recomenda-se a utilização de chaves de tamanho de 1024 bits.

Dessa forma, a implementação da arquitetura em formato matricial do multiplicador modular de Montgomery deveria possuir 1052676 elementos processadores, o que representa um número muito alto para ser implementado em hardware. Além disso, como o algoritmo de exponenciação escolhido para ser implementado necessita da utilização de

dois multiplicadores modulares de Montgomery em paralelo, uma implementação desta natureza possuiria uma quantidade de 2105352 elementos processadores.

Outro ponto merecedor de discussão é a arquitetura utilizada nos circuitos responsáveis pela soma dos operandos. Segundo a arquitetura proposta por Nedjah [15], a operação de soma dos operandos M e B e a soma realizada pelos circuitos somadores dentro dos elementos processadores são implementadas através de uma cadeia de circuitos somadores completos de um bit conectados serialmente - RCA (*Ripple Carry Adder*).

De maneira geral, a arquitetura apresentada por Nedjah [15] apresenta conceitualmente uma forma de se implementar em hardware o algoritmo de multiplicação modular de Montgomery, entretanto não enfatiza a maneira de implementar um hardware mais eficiente em termos de tempo de resposta e tamanho final do circuito. A partir desta arquitetura, este trabalho desenvolveu uma arquitetura para implementar o algoritmo de exponenciação modular apresentando modificações para melhorar o desempenho do circuito em área e em tempo de resposta.

2.2.5 Arquiteturas de Somadores

Somadores são circuitos digitais importantes que são utilizados não somente em unidades lógicas aritméticas, mas também em outras partes de processadores para cálculo de endereços, índices de tabelas e operações similares. Meio somadores e somadores completos de um bit são circuitos versáteis utilizados para construções de diversas arquiteturas de somadores. Um circuito meio somador de um bit recebe dois bits de entrada x e y e produz um bit de soma $s = x \oplus y = \bar{x}y + x\bar{y}$ e um bit de *carry out* $c_{out} = xy$. A Figura 11 ilustra três possibilidades de implementação lógica do circuito [21].

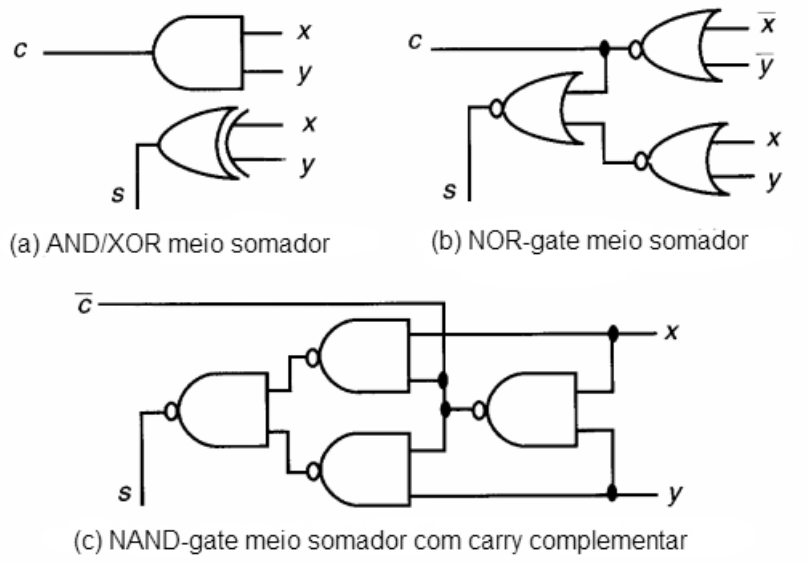

Figura 11: Circuito meio somador de 1 bit [21].

Um circuito somador completo de um bit tem como entrada três bits x, y e c_{in} e produz um bit de soma $s = x \oplus y \oplus c_{in} = xyc_{in} + \bar{x}\bar{y}c_{in} + \bar{x}y\bar{c_{in}} + x\bar{y}\bar{c_{in}}$ e um bit de *carry out* $c_{out} = xy + xc_{in} + yc_{in}$. Este circuito pode ser implementado utilizando dois circuitos meio somadores e uma porta lógica ou de duas entradas conforme ilustrado na Figura 12 [21].

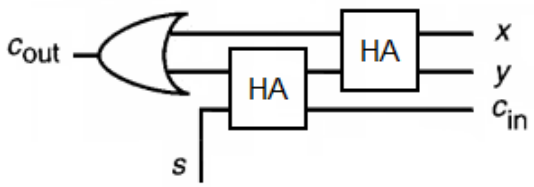

Figura 12: Circuito somador completo de 1 bit [21].

A arquitetura de somadores utilizada por Nedjah [15] no multiplicador modular de Montgomery é a RCA. O circuito desta arquitetura, ilustrada na Figura 13 para quatro bits, é simples, o que permite um tempo de desenvolvimento menor para o projeto do circuito, bem como uma área menor para sua implementação, mas deixa a desejar em desempenho, pois a propagação dos sinais é lenta. O princípio de operação consiste na propagação do sinal de *carry* na cadeia dos somadores completos de um bit até que o último somador completo de um bit processe o sinal de *carry* de saída da soma (bit mais significativo da soma) [21].

Na arquitetura RCA, o caminho crítico é definido a partir da quantidade de bits do somador. Para um somador de n bits, o caminho crítico é definido como o atraso de três portas lógicas (correspondente ao atraso das entradas e do processamento do *carry* de entrada) mais $(n-1) \times 2$, que corresponde às portas lógicas de propagação do *carry*. Exemplificando para um caso onde n é igual a 1024 bits, o atraso total é de $3 + 1023 \times 2$, de 2049 portas lógicas [21].

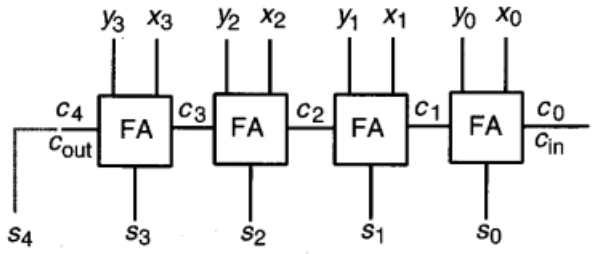

Figura 13: Arquitetura RCA de quatro bits [21].

Uma forma de desenvolver um circuito somador mais rápido é através da inserção de uma lógica de baixa latência para auxílio do cálculo do bit de *carry*. O nome da arquitetura de somadores que dispõem desta técnica é somadores com lógica vai-um antecipado - CLA (*Carry-Lookahead Adder*).

Um exemplo de um somador de 32 bits com arquitetura CLA é descrito a seguir. Como entrada deste somador, tem-se o operando x, o operando y e o sinal de *carry* de entrada c_{in}. O *carry* de entrada do bit 1 do somador é exatamente o *carry* de saída do bit 0 do somador, cuja equação lógica é:

$$c_{in}^1 = x_0 y_0 + (x_0 \oplus y_0) c_{in}$$

De modo análogo, o *carry* de entrada do bit 2 do somador é dado por:

$$c_{in}^2 = x_1 y_1 + (x_1 \oplus y_1) c_{in}^1$$

Substituindo o valor do c_{in}^1 na equação lógica do c_{in}^2, tem-se:

$$c_{in}^2 = x_1 y_1 + (x_1 \oplus y_1)[x_0 y_0 + (x_0 \oplus x_0) c_{in})]$$

Observa-se que estas equações podem ser expandidas até o cálculo do último bit de *carry*. Entretanto, a expansão das equações é de ordem exponencial. Reagrupando a equação lógica do *carry* de forma genérica, obtêm-se:

$$c_{in}^{i+1} = x_i y_i + (x_i \oplus y_i) c_{in}^i$$

A partir da forma genérica é possível observar que os termos $(x_i y_i)$ e $(x_i \oplus y_i)$ se repetem algumas vezes. Estes termos são denominados gerador (g_i) e propagador (p_i).

$$g_i = x_i y_i$$

$$p_i = x_i \oplus y_i$$

Utilizando estas relações, defini-se a equação para c_{in}^i:

$$c_{in}^{i+1} = g_i + p_i c_{in}^i$$

Com base nesta definição, tem-se as seguintes equações lógicas dos *carries* para os bits de 0 a 4:

$$c_{in}^1 = g_0 + (p_0 c_{in})$$

$$c_{in}^2 = g_1 + (p_1 g_0) + (p_1 p_0 c_{in})$$

$$c_{in}^3 = g_2 + (p_2 g_1) + (p_2 p_1 g_0) + (p_2 p_1 p_0 c_{in})$$

$$c_{in}^4 = g_3 + (p_3 g_2) + (p_3 p_2 g_1) + (p_3 p_2 p_1 g_0) + (p_3 p_2 p_1 p_0 c_{in})$$

onde c_{in} é o *carry* de entrada do somador e c_{in}^{4} é o *carry* de saída do somador.

Mesmo essa formulação mais simplificada leva à equações muito grandes e, portanto, a circuitos lógicos grandes e de alto custo de implementação em hardware, mesmo considerando um somador de 32 bits [22]. Uma técnica utilizada para contornar este problema consiste em utilizar múltiplos níveis de lógicas CLA [21].

Uma outra forma de fazer o cálculo da soma de dois operandos é através da arquitetura de antecipação de sinal de *carry* através de computação paralela - PPA (*Parallel-Prefix Adders*). Esta técnica é equivalente à técnica do CLA e a diferença fundamental é a forma como o bloco de geração do *carry* é implementado [23]. A arquitetura genérica PPA foi proposta por Ladner e Fischer [24] em 1980 e acelerou o tempo de resposta de um somador de n bits em tecnologia VLSI.

O projeto de um somador com arquitetura PPA geralmente é dividido em três passos. O primeiro passo consiste em criar os sinais de geração e propagação para todos os bits dos operandos. O segundo passo consiste do cálculo dos *carries* intermediários através do agrupameto lógico dos sinais de geração e propagação. No terceiro e último passo, os bits da soma do circuito somador são calculados com os sinais de propagação e com os bits de *carry* gerados a partir do agrupamento lógico dos sinais de geração e propagação [23]. A Figura 14 ilustra esse processo.

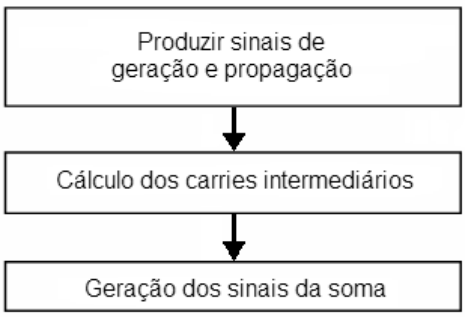

Figura 14: Passos para projeto de um somador de arquitetura PPA [23].

Esta abordagem abre possibilidades para novos arranjos de lógica de cálculo paralelo de *carries* que apresentam uma quantidade enorme de vantagens e desvantagens em termos de profundidade de níveis lógicos do caminho crítico, quantidade de elementos, *fan-out* e quantidade de interconexões. Dessa forma, o projeto de somadores se reduz ao arranjo de lógica dos grupos de geração e propagação - GPGL (*Group Propagate and Generate Logic)* apresentado na Figura 15. A soma é calculada através de uma lógica simples de ou-exclusivo entre o grupo propagação e o agrupameto lógico dos sinais de geração e propagação [25].

O arranjo da lógica de geração dos *carries* intermediários caracteriza o tipo de somador PPA projetado. Dentre os arranjos de lógica mais conhecidos, pode-se citar:

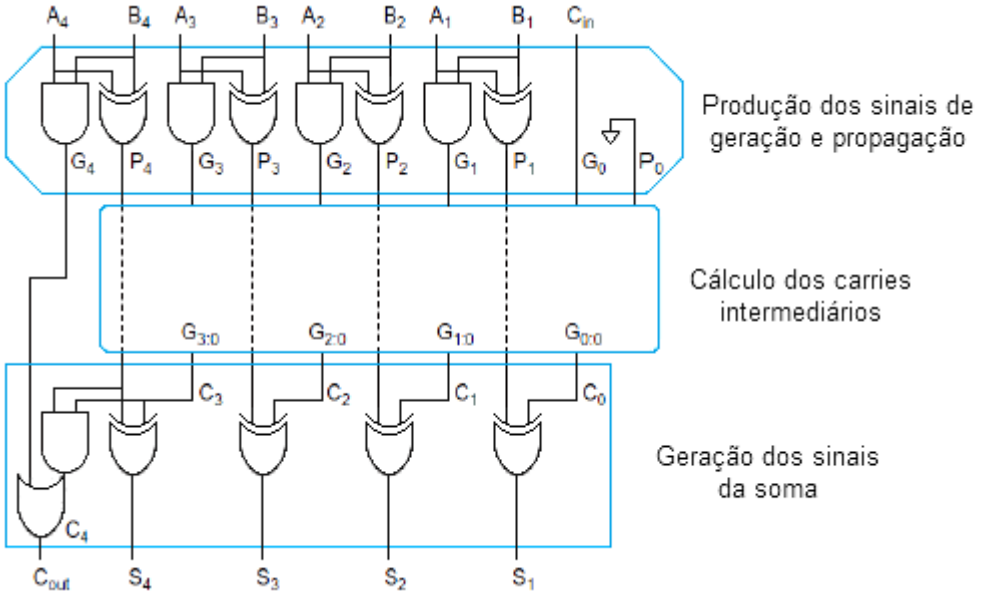

Figura 15: Exemplo esquemático somador quatro bits arquitetura PPA [25].

Kogge-Stone, Brent-Kung e Han-Carlson.

O somador com arranjo de Kogge-Stone - KSA (*Kogge-Stone Adder*) possui as características de menor profundidade de níveis lógicos possível do caminho crítico e o menor *fan-out*, em contrapartida, este também possui uma grande quantidade de elementos e interconexões [23].

O somador com arranjo de Brent-Kung - BKA (*Brent-Kung Adder*) tem como característica principal a menor quantidade possível de elementos, o que aumenta o aproveitamento de área, mas possui uma profundidade de níveis lógicos do caminho crítico que impactam na latência desta estrutura [23].

O somador com arranjo Han-Carlson - HCA (*Han-Carlson Adder*) combina as técnicas dos somadores KSA e BKA e apresenta uma estrutura balanceada entre a profundidade de níveis lógicos do caminho crítico, quantidade de elementos e de interconexões [23].

2.2.6 Somador KSA

A principal característica da arquitetura KSA é a menor profundidade de níveis lógicos possível do caminho crítico do sinal de *carry*. A arquitetura é reconhecida na indústria como a mais rápida e é utilizada nos projetos de circuitos aritméticos de alto desempenho [23]. Esta arquitetura se fundamenta nas técnicas de geração e propagação de *carry* e as utiliza em uma lógica de agrupamento para cálculo *carry*. Um somador KSA é capaz de gerar um sinal de *carry* na ordem de $O(log\ n)$, onde n é o números de bits de entrada do somador.

Entretanto, o custo de possuir a menor profundidade de níveis lógicos no caminho crítico aumenta a quantidade de elementos utilizados para a computação paralela do *carry*, a quantidade de interconexões e as capacitância geradas durante o roteamento entre os elementos que compõem a lógica de agrupamento [25].

A recorrência matemática dos grupos geração e propagação para um conjunto de bits de i a j da arquitetura KSA são apresentadas nas Equações 2.26 e 2.27 [25].

$$G_{i:j} = G_{i:k} + P_{i:k} \times G_{k-1:j} \tag{2.26}$$

$$P_{i:j} = P_{i:k} \times P_{k-1:j} \tag{2.27}$$

onde

$$G_{i:i} \equiv G_i = A_i \times B_i \qquad G_{0:0} = c_{in}$$

$$P_{i:i} \equiv P_i = A_i \oplus B_i \qquad P_{0:0} = 0$$

Os blocos básicos desta arquitetura são células cinzas e células pretas que calculam o grupo de geração e propagação de sinais. A Figura 16 ilustra o esquemático destas células [25].

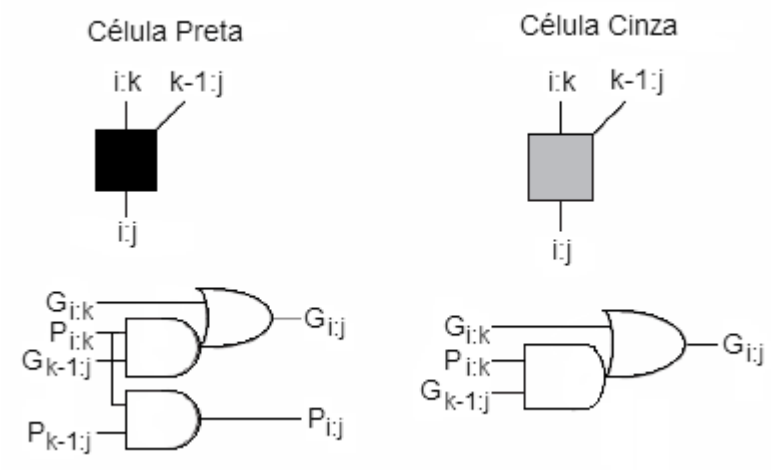

Figura 16: Ilustração e esquemático das células cinza e preta [25].

Uma ilustração da lógica de agrupamento com as células pretas e cinzas do somador KSA pode ser observada na Figura 17. Vale ressaltar que, neste exemplo de 16 bits, a árvore tem profundidade de 4 níveis de lógica de agrupamento de propagação e geração e que o *fan-out* máximo de cada nível da lógica de agrupamento é igual a um.

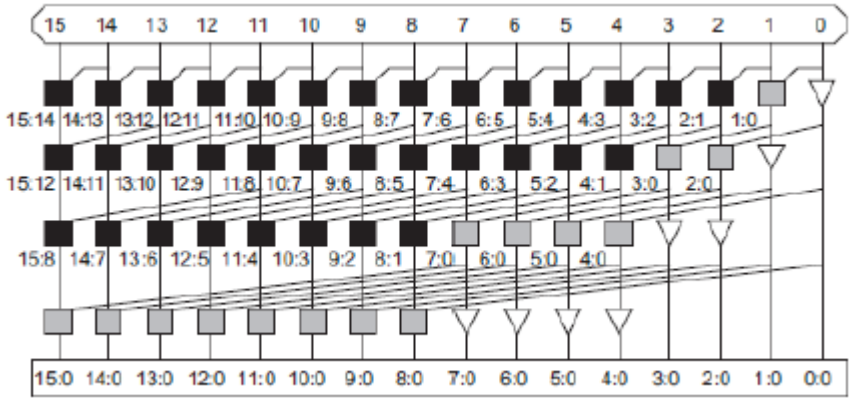

Figura 17: Esquemático da árvore Kogge-Stone de 16 bits [25].

3 Implementação em Hardware

Este capítulo apresenta todos os blocos que fazem parte da estrutura do coprocessador RSA, seus sinais de entrada e de saída, o fluxo interno dos dados e os resultados das simulações para comprovação do funcionamento da arquitetura. A implementação do coprocessador foi feita de maneira modular durante o desenvolvimento. A arquitetura foi inicialmente desenvolvida no software Quartus com tamanho de chave de quatro bits. Uma vez validada a arquitetura através de simulações comportamentais, simulações pós-roteamento e testes no kit de desenvolvimento da Altera, o código para a versão de 1024 bits foi desenvolvido e validado através de simulações comportamentais e pós-síntese no pacote de software Cadence.

3.1 Estrutura hierárquica do coprocessador RSA

A estrutura hierárquica do coprocessador RSA é apresentada na Figura 18 com todos os blocos desenvolvidos. Vale ressaltar que a quantidade de vezes que cada um dos blocos apresentados é utilizado varia de acordo com a quantidade de bits, bem como também os sinais de interligação entre os blocos.

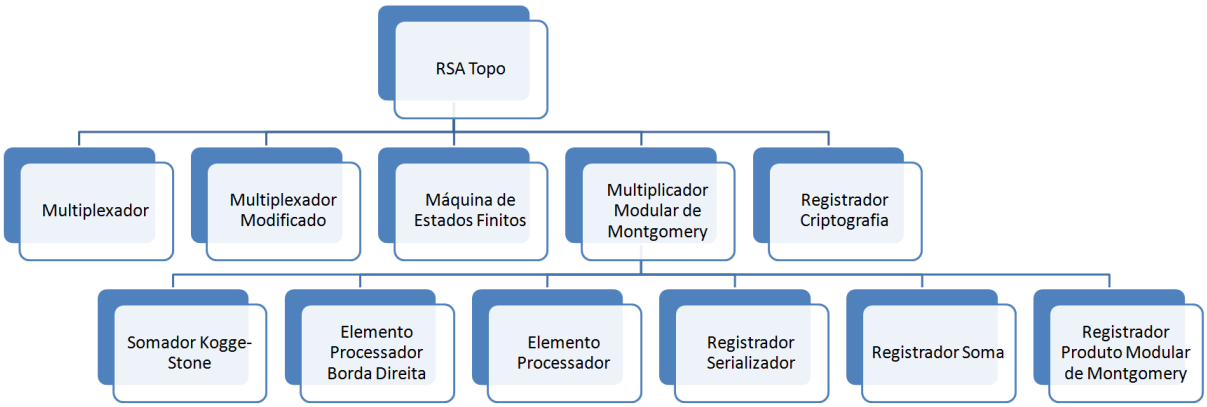

Figura 18: Estrutura hierárquica do coprocessador RSA.

Os detalhes de cada bloco da estrutura são apresentados nas próximas seções através da abordagem *top-down* para um coprocessador de quatro bits. A estrutura para um circuito de 1024 bits será abordada após a explanação de todos os blocos e do fluxo de dados internos. A quantidade de elementos lógicos utilizados, a frequência máxima de operação e outras detalhes são apresentados somente para o circuito de 1024 bits.

3.2 Coprocessador RSA de quatro bits

3.2.1 Bloco RSA Topo

Este bloco conecta todos os sub-blocos desenvolvidos do coprocessador RSA. Trata-se do bloco de hierarquia de nível mais elevado da arquitetura desenvolvida. O código deste bloco contém somente ligações de hierarquia, ou seja, são conexões de sinais sem portas lógicas e registradores.

Os sinais de entrada do bloco RSA Topo são *enable*, *reset*, *clock*, uma palavra P de quatro bits, uma palavra E de quatro bits, uma palavra M de quatro bits e uma palavra *Constm* de quatro bits. Os sinais de saída do bloco RSA Topo são *eoc* e uma palavra C de quatro bits. A Figura 19 ilustra o bloco RSA Topo gerado através do software Quartus.

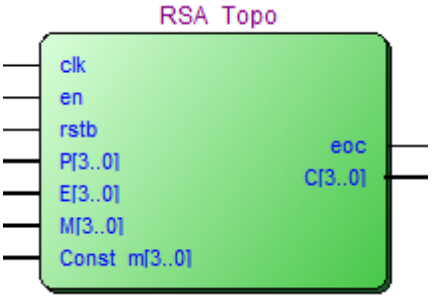

Figura 19: Bloco RSA Topo do coprocessador RSA de quatro bits.

O sinal de *enable* habilita o funcionamento do bloco quando em nível lógico alto. O sinal de *reset* deve ser acionado para colocar os blocos internos no estado inicial antes de realizar uma cifragem ou decifragem. Todos os registradores utilizados no projeto deste coprocessador foram descritos em Verilog como *flip-flops* com *reset* assíncrono e com sensibilidade a uma transição negativa. Desta forma, uma transição negativa do sinal *reset* coloca todos os blocos do coprocessador no estado inicial para a realizar uma cifragem ou decifragem. O sinal de *clock* é a fonte de tempo do circuito. É importante ressaltar neste ponto, que os sinais *enable*, *clock* e *reset* são comuns a todos os sub-blocos. Os sinais P, E e M, todos de tamanho igual a quatro bits, são palavras de entrada referentes a Equação 2.3, reescrita conforme indicado na Equação 3.1. O sinal *Constm* é a constante de Montgomery.

$$C = P^E \ mod \ M \tag{3.1}$$

O sinal de saída C, palavra de tamanho igual a quatro bits, representa o resultado da Equação 3.1. Este dado é a principal saída do circuito, uma vez que trata-se do dado cifrado. O sinal de saída *eoc* foi desenvolvido para ativar um circuito de interrupção e seu estado normal durante o processo do cálculo da Equação 3.1 é nível lógico baixo. Ao

término da cifragem ou decifragem, o sinal *eoc* assume valor lógico alto por exatamente um pulso de *clock*.

O diagrama de blocos internos ao RSA Topo é ilustrado na Figura 20 sendo o mesmo composto por:

- Um bloco Máquina de Estados Finitos;
- Dois blocos Multiplexador;
- Dois blocos Multiplexador Modificado;
- Dois blocos Multiplicador Modular de Montgomery; e
- Um bloco Registrador Criptografia

Os sinais de entrada e saída de cada bloco são apresentados em suas respectivas seções. Para uma implementação de n bits, o número de bits utilizados no algoritmo são $n+2$, ou seja, neste caso, quatro mais dois, totalizando seis bits. Sendo assim, internamente ao bloco RSA Topo, todos os sub-blocos realizam processamento com palavras de tamanho igual a seis bits, em vez de quatro bits[1] .

3.2.2 Bloco Multiplexador

Este bloco é responsável pela seleção dos dados a serem utilizados como entrada do bloco Multiplicador Modular de Montgomery nas etapas de mapeamento e de potenciação do Algoritmo 4. Trata-se de um multiplexador simples de dois canais de seis bits. A chave seletora *sel* seleciona quais dos canais a ou b, serão transferidos para a saída *out*.

As Figuras 21 e 22 ilustram o bloco gerado através do software Quartus e o diagrama RTL do bloco respectivamente.

O bloco Multiplexador é utilizado duas vezes no projeto do coprocessador e está associado a um bloco Multiplicador Modular de Montgomery. Para diferenciar os dois blocos Multiplexador, denominou-se um bloco com a sigla MuxA e outro bloco com a sigla MuxB. A saída do bloco MuxA é conectada na entrada A do bloco Multiplicador Modular de Montgomery, enquanto a saída do bloco MuxB é conectada na entrada B do bloco Multiplicador Modular de Montgomery.

Durante a etapa de mapeamento do Algoritmo 4, o bloco MuxA seleciona o operando $Constm$ e o bloco MuxB seleciona o operando $\mathcal{P}$ como dados de entrada do bloco Multiplicador Modular de Montgomery. Esta operação corresponde à linha dois do Algoritmo 4. Durante a etapa de potenciação do Algoritmo 4, os blocos MuxA e MuxB

[1] Para maiores detalhes ver seção 2.2.1

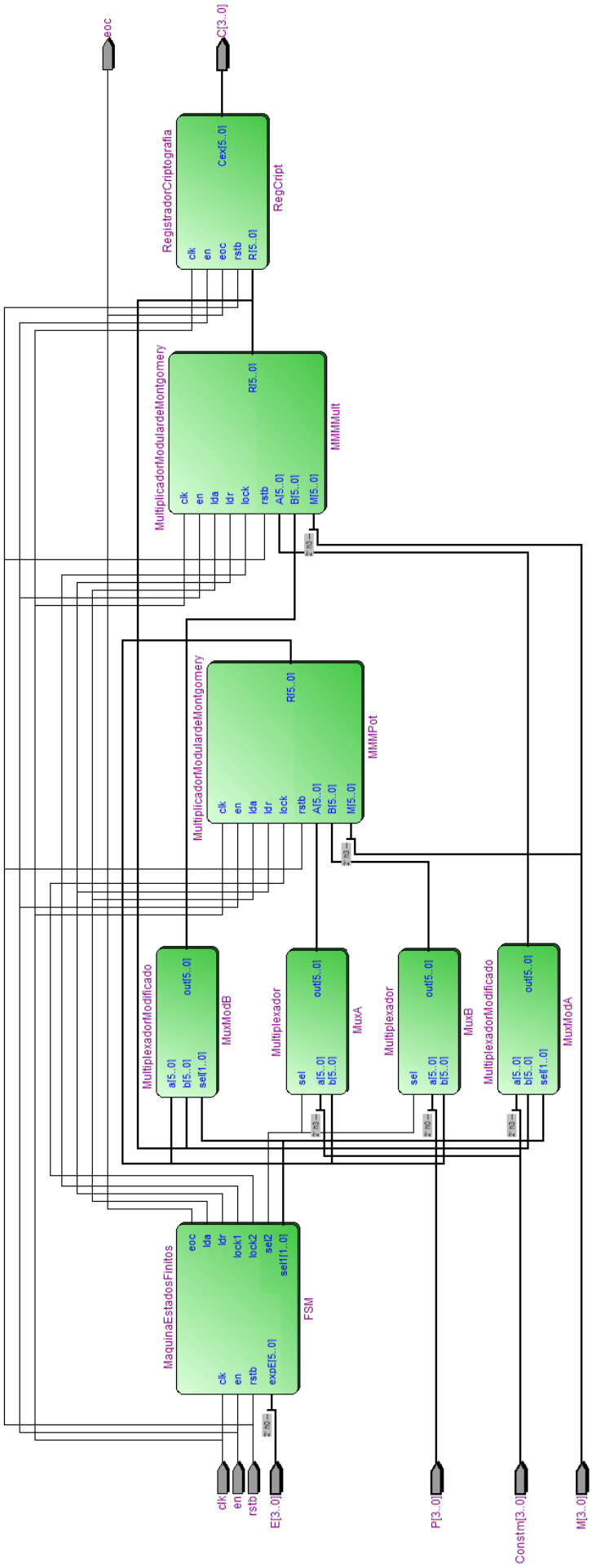

Figura 20: Blocos internos do coprocessador RSA de quatro bits.

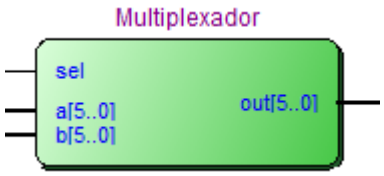

Figura 21: Bloco Multiplexador do coprocessador RSA de quatro bits.

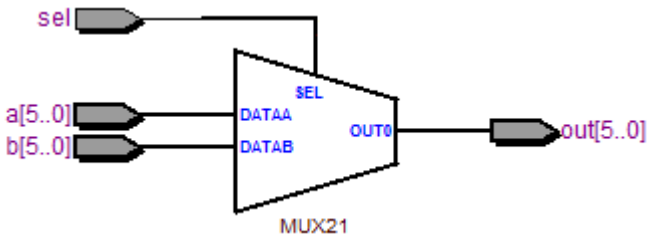

Figura 22: Diagrama RTL do bloco Multiplexador do coprocessador RSA de quatro bits.

selecionam o operando P como dado de entrada do bloco Multiplicador Modular de Montgomery. Esta operação corresponde à linha oito do Algoritmo 4. O código Verilog do bloco Multiplexador se encontra disponível para apreciação.

```
1 module multiplexador (a, b, sel, out);
2   input [5:0] a;
3   input [5:0] b;
4   input sel;
5   output [5:0] out;
6
7   wire [5:0] out;
8   assign out = (sel) ? b : a;
9 endmodule
```

3.2.3 Bloco Multiplexador Modificado

Este bloco é responsável pela seleção dos dados a serem utilizados como entrada do bloco Multiplicador Modular de Montgomery nas etapas de mapeamento, de multiplicação e de remapeamento do Algoritmo 4. Trata-se de um multiplexador de quatro canais de seis bits, dos quais dois canais são sinais constantes. As chaves seletoras, sinal *sel* composto de dois bits, selecionam quais dos canais a, b, c ou d, serão transferidos para a saída *out*. Os canais c e d são internos do multiplexador com valores constantes 0, no formato binário "000000b", e 1, no formato binário "000001b".

As Figuras 23 e 24 ilustram o bloco gerado através do software Quartus e o diagrama RTL do bloco respectivamente.

Observe que apesar de ser um multiplexador de quatro canais de seis bits, somente dois canais de entrada fazem interface com os outros blocos, uma vez que os outros dois canais são as constantes 0 e 1.

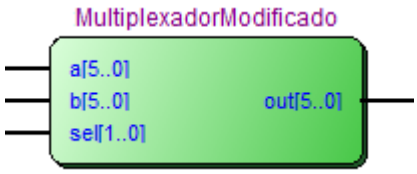

Figura 23: Bloco Multiplexador Modificado do coprocessador RSA de quatro bits.

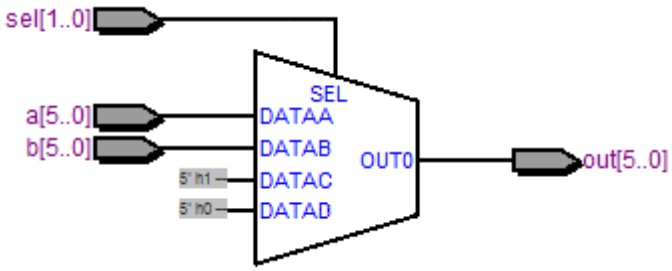

Figura 24: Diagrama RTL do bloco Multiplexador Modificado do coprocessador RSA de quatro bits.

O bloco Multiplexador Modificado é utilizado duas vezes no projeto do coprocessador e está associado a um bloco Multiplicador Modular de Montgomery. Para diferenciar os dois blocos Multiplexador Modificado, denominou-se um bloco com a sigla MuxModA e outro bloco com a sigla MuxModB. A saída do bloco MuxModA é conectada na entrada A do bloco Multiplicador Modular de Montgomery, enquanto a saída do bloco MuxModB é conectada na entrada B do bloco Multiplicador Modular de Montgomery.

Durante a etapa de mapeamento do Algoritmo 4, o bloco MuxModA seleciona o operando *Constm* e o bloco MuxModB seleciona o operando 1 como dados de entrada do bloco Multiplicador Modular de Montgomery. Esta operação corresponde à linha três do Algoritmo 4. Durante a etapa de multiplicação do Algoritmo 4, o bloco MuxModA seleciona o operando R e o bloco MuxModB seleciona o operando P como dados de entrada do bloco Multiplicador Modular de Montgomery. Esta operação corresponde à linha seis do Algoritmo 4. Durante a etapa de remapeamento do Algoritmo 4, o bloco MuxModA seleciona o operando 1 e o bloco MuxModB seleciona o operando R como entradas do bloco Multiplicador Modular de Montgomery. Esta operação corresponde à linha dez do Algoritmo 4.

O código Verilog do bloco Multiplexador Modificado se encontra disponível para apreciação.

```
1 module multiplexador_modificado (a, b, sel, out);
2   input [5:0] a;
3   input [5:0] b;
4   input [1:0] sel;
5   output [5:0] out;
6
```

```
7   reg [5:0] out;
8   supply0 [5:0] zero;
9   supply0 [4:0] one_array;
10  supply1 one_bit;
11  wire [5:0] one = {one_array, one_bit};
12
13  always @(a or b or one or zero or sel) begin
14    case (sel)
15      2'b00 : out = a;
16      2'b01 : out = b;
17      2'b10 : out = one;
18      default out = zero;
19    endcase
20  end
21 endmodule
```

3.2.4 Bloco Registrador Criptografia

Este bloco é responsável pelo armazenamento do dado produzido pelo circuito no final do processo da criptografia, ou seja, no término da execução do Algoritmo 4. Trata-se de um circuito composto por um registrador de seis bits com sinais de habilitação e um multiplexador para seleção do dado a ser armazenado no registrador.

Os sinais de entrada deste bloco são *enable*, *reset*, *clock*, *eoc* e uma palavra R de seis bits. O único sinal de saída deste bloco é uma palavra *Cex* de seis bits.

As Figuras 25 e 26 ilustram o bloco gerado através do software Quartus e o diagrama RTL do bloco respectivamente.

Figura 25: Bloco Registrador Criptografia do coprocessador RSA de quatro bits.

Este bloco é utilizado uma única vez na arquitetura projetada e denominou-se a sigla RegCript como sua referência. A palavra de entrada R é o resultado da execução do Algoritmo 4. O sinal de entrada *eoc*, quando em nível lógico alto, transfere o sinal de entrada R para a saída do multiplexador. O registrador atualiza seu valor com o sinal de entrada R na transição positiva do sinal de *clock*. No caso oposto, quando o sinal *eoc* é nível lógico baixo, o valor do sinal de saída do próprio registrador é transferido para a saída do multiplexador. O registrador atualiza seu valor com seu próprio sinal de saída na transição positiva do sinal de *clock*, ou seja, o registrador mantém seu último valor.

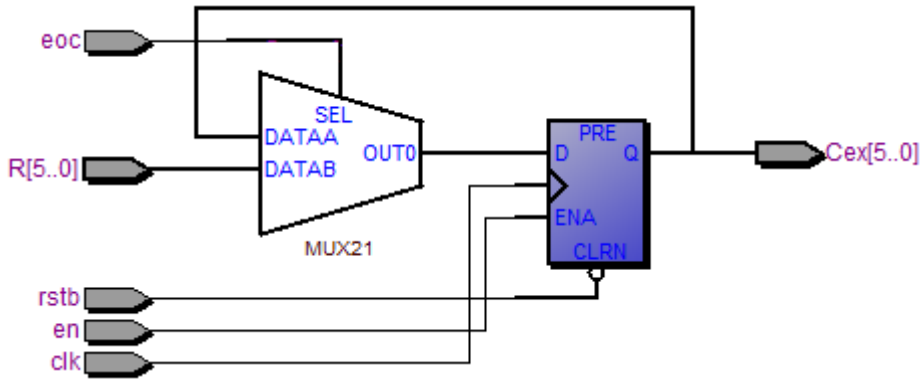

Figura 26: Diagrama RTL do bloco Registrador Criptografia do coprocessador RSA de quatro bits.

O sinal de saída *Cex* é o resultado do processo de cifragem ou decifragem. Apesar do sinal de saída *Cex* ser composto por seis bits, os quatro bits menos significativos são efetivamente o resultado da exponenciação modular da Equação 3.1 para o coprocessador.

O código Verilog do bloco Registrador Criptografia se encontra disponível para apreciação.

```
1  module registrador_criptografia (en, rstb, clk, eoc, R, Cex);
2    input en;
3    input rstb;
4    input clk;
5    input eoc;
6    input [5:0] R;
7    output [5:0] Cex;
8
9    reg [5:0] Cex;
10   always @(negedge(rstb) or posedge(clk)) begin
11     if (!rstb) begin
12       Cex <= {6{1'b0}};
13     end else begin
14       if (en) begin
15         if (eoc) begin
16           Cex <= R;
17         end else begin
18           Cex <= Cex;
19         end
20       end
21     end
22   end
23 endmodule
```

3.2.5 Bloco Máquina de Estados Finitos

Este bloco é responsável pela geração de todos os sinais de controle utilizados nos blocos do coprocessador RSA. Codificado através de uma máquina de estados finitos, este bloco gera os sinais de saída de acordo com o estado em que a máquina se encontra, mantendo o sincronismo entre todos os blocos do coprocessador.

A quantidade de estados da máquina de estados finitos desenvolvida depende do números de bits da arquitetura. A equação genérica para a arquitetura desenvolvida é $(n+4) \times (n+3)$, onde n é a quantidade de bits. Para esta implementação de quatro bits, tem-se um total de 56 estados.

Os sinais de entrada deste bloco são *enable*, *reset*, *clock* e uma palavra *expE* de seis bits. Os sinais de saída deste bloco são *eoc*, *lda*, *ldr*, *lock1*, *lock2*, *sel1* e *sel2*.

A Figura 27 ilustra o bloco gerado através do software Quartus.

Figura 27: Bloco Máquina de Estados Finitos do coprocessador RSA de quatro bits.

Este bloco é utilizado uma única vez na arquitetura projetada e denominou-se a sigla FSM como sua referência. A Tabela 4 apresenta a relação entre os sinais de saída do bloco Máquina de Estados Finitos e os sinais de entrada dos demais blocos do coprocessador no mesmo nível hierárquico.

Tabela 4: Relação entre os sinais de saída do bloco Máquina de Estados Finitos e sinais de entrada dos demais blocos do coprocessador.

Sinal de saída do bloco Máquina de Estados Finitos	**Blocos do coprocessador**
eoc	Registrador Criptografia (RegCript)
lda, *ldr*, *lock1* e *lock2*	Multiplicador Modular de Montgomery
sel1	Multiplexador Modificado (MuxModA e MuxModB)
sel2	Multiplexador (MuxA e MuxB)

O sinal de entrada *ExpE* é o próprio sinal de entrada E do bloco RSA Topo. No Algoritmo 4, os bits do expoente E são percorridos serialmente, do bit menos significativo para o mais significativo, a cada iteração da variável i. A cada passo da variável i, o bit E_i dependendo do seu valor lógico, determina se um dos blocos Multiplicador Modular de Montgomery deve ou não realizar a etapa de multiplicação, operação correspondente

a linha seis do Algoritmo 4. Esta verificação corresponde ao operador *if*, na linha cinco do Algoritmo 4.

O bloco FSM gera os sinais de saída *lock1* e *lock2* a partir do valor lógico dos bits do expoente E_i a cada iteração da variável i. Os sinais de saída *lock1* e *lock2* são responsáveis por travar o bloco Multiplicador Modular de Montgomery quando a operação da etapa de multiplicação, linha seis do Algoritmo 4, não deve ser realizada.

O sinal de saída *eoc* é conectado ao bloco RegCript, o qual o utiliza para capturar o dado cifrado após o término da execução do Algoritmo 4. O sinal de saída *sel1* é utilizado como chave seletora dos blocos MuxModA e MuxModB. O sinal de saída *sel2* é utilizado como chave seletora dos blocos MuxA e MuxB.

Os sinais de saída *lda* e *ldr* são conectados aos dois blocos Multiplicador Modular de Montgomery, cujas funcionalidades serão explicadas na próxima seção. Um fluxograma simplificado do código Verilog desenvolvido se encontra nos Apêndices deste trabalho.

3.2.6 Bloco Multiplicador Modular de Montgomery

Este bloco é responsável pela operação de multiplicação modular de Montgomery. A sequência de comandos para o funcionamento do bloco Multiplicador Modular de Montgomery é gerada a partir do bloco FSM. Assim como no bloco RSA Topo, o código deste bloco também contém somente ligações de hierarquia de seus sub-blocos, ou seja, são conexões de sinais sem utilização de portas lógicas e registradores.

Os sinais de entrada deste bloco são *enable*, *reset*, *clock*, *lda*, *ldr*, *lock*, uma palavra A de seis bits, uma palavra B de seis bits e uma palavra M de seis bits. O único sinal de saída deste bloco é uma palavra R de seis bits.

A Figura 28 ilustra o bloco gerado através do software Quartus.

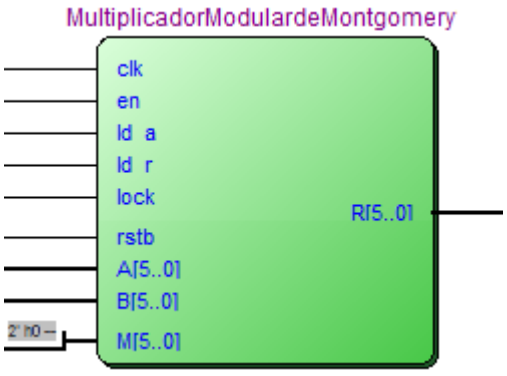

Figura 28: Bloco Multiplicador Modular Montgomery do coprocessador RSA de quatro bits.

Este bloco é utilizado duas vezes na arquitetura projetada. Para diferenciar os dois blocos Multiplicador Modular de Montgomery, denominou-se um bloco com a sigla MMM-

Pot e outro bloco com a sigla MMMMult. A denominação da sigla foi realizada de acordo com a operação que cada bloco executa na etapa de multiplicação (bloco MMMMult) e potenciação (bloco MMMPot) do Algoritmo 4.

Ambos os blocos MMMPot e MMMMult são utilizados durante a etapa de mapeamento do Algoritmo 4. O bloco MMMPot é responsável por executar o cálculo da linha dois do Algoritmo 4, enquanto o bloco MMMMult é responsável pelo cálculo da linha três do Algoritmo 4. Na etapa de multiplicação e potenciação, o bloco MMMMult é responsável pelo cálculo da linha seis do Algoritmo 4, enquanto o bloco MMMPot é responsável pelo cálculo da linha oito do Algoritmo 4. Durante a etapa de remapeamento, linha dez do Algoritmo 4, somente o bloco MMMMult é utilizado para realizar a operação de multiplicação modular de Montgomery.

Os sinais de entrada *lda* e *ldr* são utilizados em alguns sub-blocos para carregar valores pré definidos em registradores utilizados durante os cálculos das multiplicações modular de Montgomery. O sinal de entrada *lock* é utilizado para travar a execução do bloco Multiplicador Modular de Montgomery quando a operação não deve ser executada. O sinal de saída R contém o resultado da Equação 3.2.

$$R = A \times B \times r^{-1} \; mod \; M \tag{3.2}$$

O sinal de saída R dos blocos MMMPot e MMMMult são utilizados como entrada dos próprios blocos a cada iteração da variável i, conforme indicam as linhas seis e oito do Algoritmo 4. Os blocos responsáveis por fazer a conexão do sinal de saída R do bloco MMMPot no sinal de entrada A do bloco MMMPot são os blocos MuxA e MuxB. Os blocos responsáveis por fazer a conexão do sinal de saída R do bloco MMMMult no sinal de entrada A do bloco MMMMult são os blocos MuxModA e MuxModB.

O diagrama dos sub-blocos do bloco Multiplicador Modular de Montgomery para um coprocessador de quatro bits é ilustrado na Figura 29 sendo o mesmo composto por:

- Bloco Somador Kogge-Stone
- Bloco Elemento Processador Borda Direita
- Bloco Elemento Processador
- Bloco Registrador Serializador
- Bloco Registrador Soma
- Bloco Registrador Produto Modular

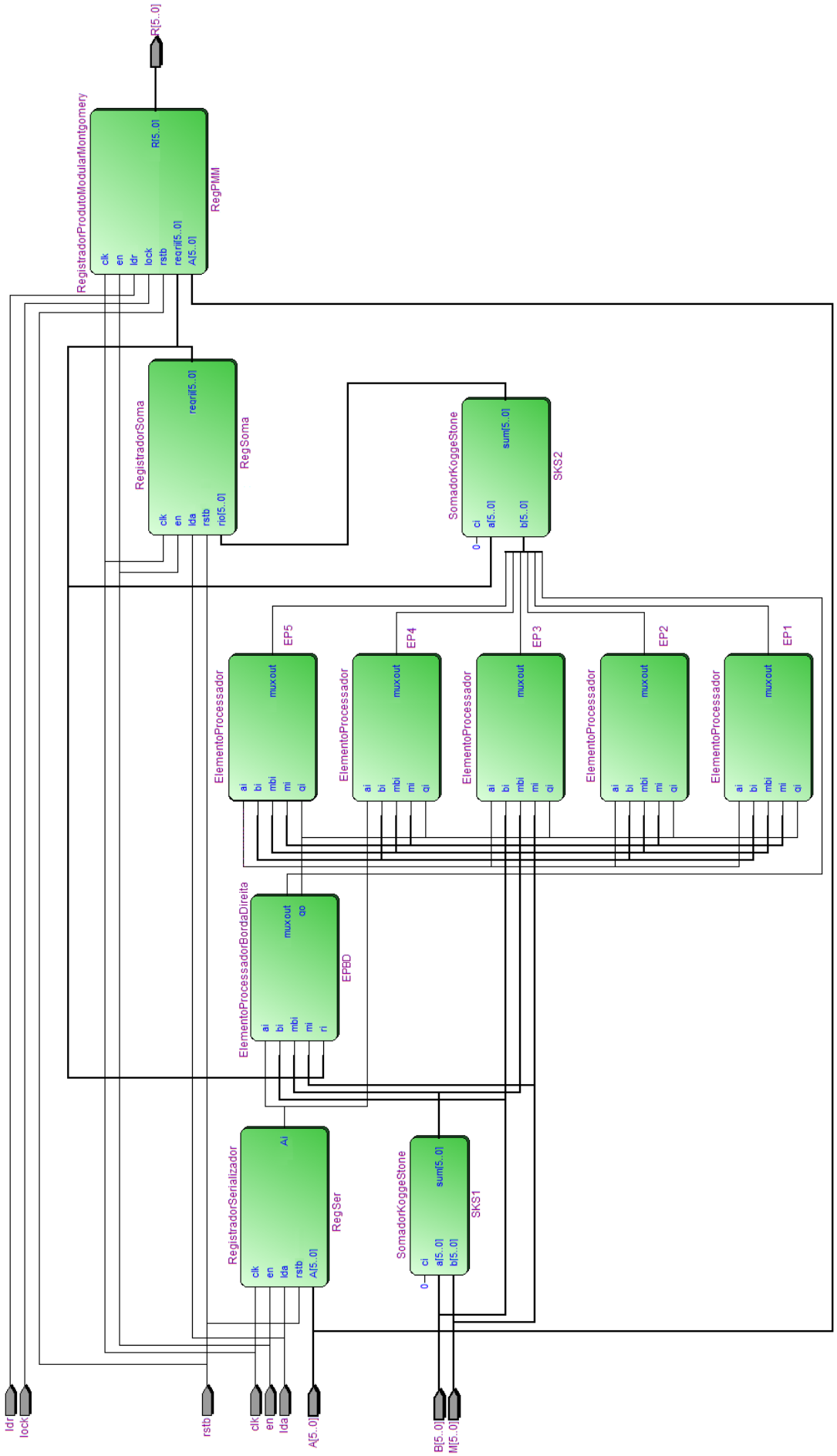

Figura 29: Blocos internos do bloco Multiplicador Modular de Montgomery.

Tabela 5: Quantidade de vezes de utilização de cada sub-bloco em um bloco Multiplicador Modular de Montgomery para coprocessador de quatro bits.

Sub-bloco	**Quantidade de vezes de utilizado**
Somador kogge-Stone	2
Registrador Serializador	1
Registrador Soma	1
Registrador Produto Modular	1
Elemento Processador Borda Direita	1
Elemento Processador	5

A Tabela 5 indica a quantidade de vezes que cada sub-bloco é utilizado dentro de um bloco Multiplicador Modular de Montgomery para o coprocessador de quatro bits.

Para uma arquitetura genérica de n bits, a Tabela 6 indica a quantidade de vezes que cada sub-bloco é utilizado dentro de um bloco Multiplicador Modular de Montgomery.

Tabela 6: Quantidade de vezes de utilização de cada sub-bloco em um bloco Multiplicador Modular de Montgomery - genérico.

Sub-bloco	**Quantidade de vezes de utilizado**
Somador kogge-Stone	2
Registrador Serializador	1
Registrador Soma	1
Registrador Produto Modular	1
Elemento Processador Borda Direita	1
Elemento Processador	$n+1$

A funcionalidade, os sinais de entrada e os sinais de saída de cada sub-bloco serão apresentados e comentadas nas próximas seções.

3.2.7 Bloco Somador Kogge-Stone

Este bloco é responsável pela adição de dois operandos de seis bits. A arquitetura Kogge-Stone, conforme já apresentada, permite realizar a operação de adição em menor tempo do que as arquiteturas convencionais RCA e CLA. O bloco Somador Kogge-Stone tem como característica a ausência de registradores, ou seja, este bloco é composto somente de portas lógicas.

Os sinais de entrada deste bloco são *carry ci*, uma palavra a de seis bits e uma palavra b de seis bits. Os sinais de saída deste bloco são *carry co* e uma palavra *sum* de seis bits. A Figura 30 ilustra o bloco gerado através do software Quartus.

Cada bloco Multiplicador Modular de Montgomery contém dois blocos Somador Kogge-Stone de seis bits. Um dos blocos Somador Kogge-Stone é utilizado para o cálculo da adição dos operandos B e M, que são entradas do bloco Multiplicador Modular de Montgomery e não variam até o final da iteração do Algoritmo 5. A partir deste ponto, este somador será referenciado por SKS1.

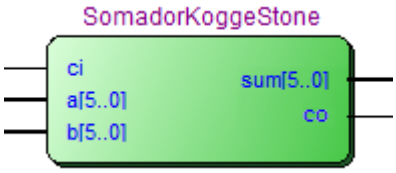

Figura 30: Bloco Somador Kogge-Stone do coprocessador RSA de quatro bits.

O outro bloco Somador Kogge-Stone é utilizado para calcular a adição entre os operandos das etapas intermediárias do algoritmo de multiplicação modular de Montgomery, conforme indicam as linhas 12 e 13 do Algoritmo 5. De forma semelhante, a partir deste ponto, este somador será referenciado por SKS2.

A arquitetura dos somadores Kogge-Stone é somente definida para palavras de entrada de ordem 2^n bits, ou seja, palavras de dois bits, quatro bits, oito bits, etc. Para esta implementação, cujas palavras de entrada são de seis bits, optou-se desenvolver uma arquitetura com n igual a dois e acrescentar dois somadores completos de um bit. Os dois somadores completos de um bit são conectados serialmente à saída do circuito de arquitetura Kogge-Stone, conforme a definição da arquitetura RCA. A Figura 31 ilustra o diagrama RTL do bloco Somador Kogge-Stone de seis bits.

3.2.8 Bloco Elemento Processador Borda Direita

Este bloco é responsável pelo processamento do bit q_i no Algoritmo 5 e pela seleção do bit menos significativo a ser utilizado pelo somador SKS2 dos blocos MMMMult e MMMPot.

O bloco Elemento Processador Borda Direita é composto por um multiplexador de quatro canais com uma chave seletora de dois bits e duas portas lógicas. Os sinais de entrada deste bloco são os bits B_0, A_i, M_0, MB_0 e r_i. Os sinais de saída deste bloco são os bits *muxout* e q_i.

No Algoritmo 5, o bit q_i é processado a partir do valor lógico do bit B_0 e dos bits $S^i_{(0)}$ e A_i, cujos valores dependem do valor da variável de iteração i. Este processamento acontece na linha cinco do Algoritmo 5, segundo a Equação lógica 3.3.

$$q_i = S_0^{(i)} \ \boldsymbol{xor} \ (a_i \ \boldsymbol{and} \ b_0) \tag{3.3}$$

Neste contexto, os sinais A e B são os sinais de entrada do bloco Multiplicador Modular de Montgomery e o sinal S representa o valor intermediário do produto modular de Montgomery.

Além do processamento do bit q_i, este bloco também realiza a seleção do bit menos significativo utilizado na entrada do somador SKS2. A combinação dos bits A_i e q_i

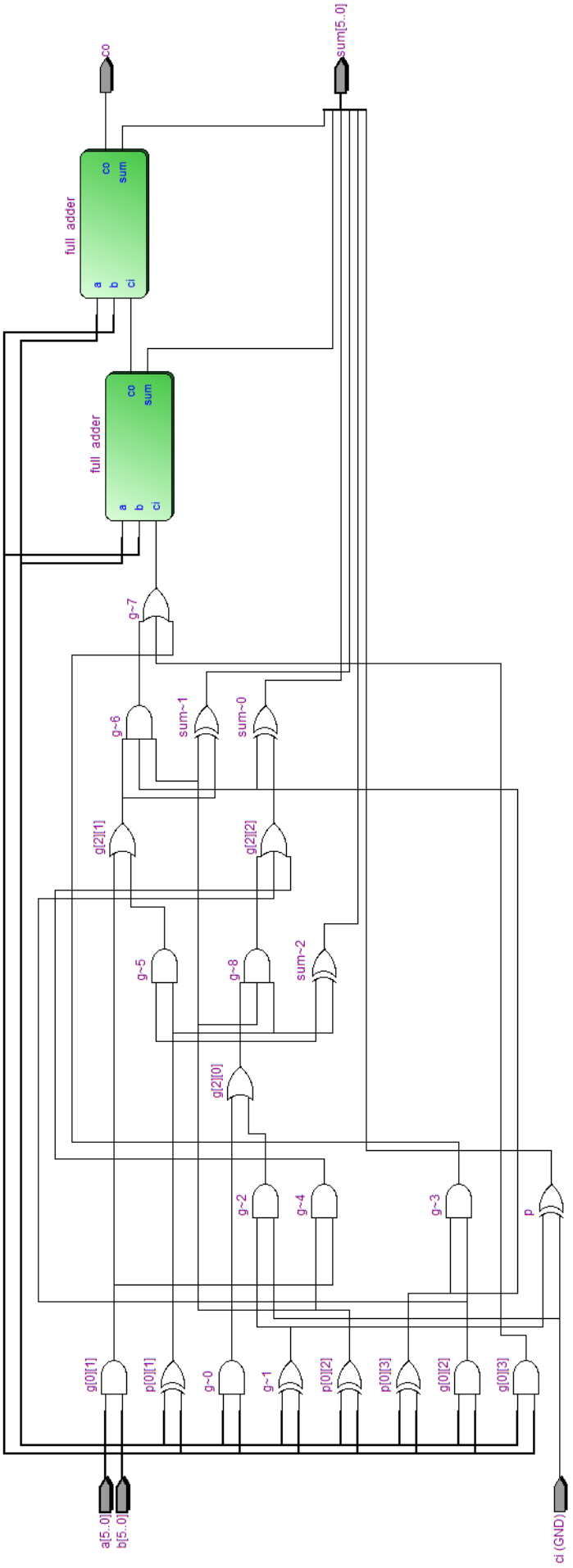

Figura 31: Diagrama RTL do Somador Kogge-Stone do coprocessador RSA de quatro bits.

selecionam quais dos bits serão utilizados no somador SKS2 para a respectiva iteração da variável i. Essa operação corresponde as linhas sete a 11 do Algoritmo 5, quando a variável j é igual a zero. Os possíveis valores que a saída *muxout* pode assumir estão presentes na Tabela 7.

Tabela 7: Resultado da expressão da linha sete a 11 do Algoritmo 5.

a_i	q_i	x_0^i
0	0	0
0	1	m_0
1	0	b_0
1	1	mb_0

As Figuras 32 e 33 ilustram o bloco gerado através do software Quartus e o diagrama RTL do bloco respectivamente.

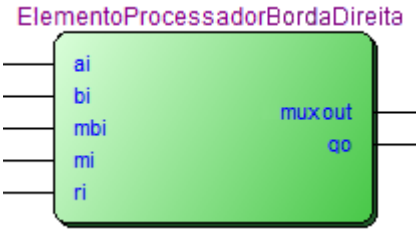

Figura 32: Bloco Elemento Processador Borda Direita do coprocessador RSA de quatro bits.

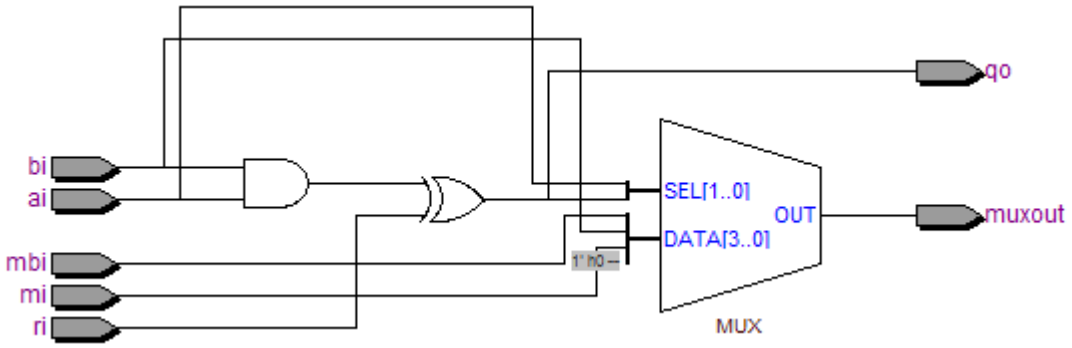

Figura 33: Diagrama RTL do bloco Elemento Processador Borda Direita do coprocessador RSA de quatro bits.

O bloco Elemento Processador Borda Direita é utilizado uma única vez dentro dos blocos MMMMult e MMMPot e denominou-se a sigla EPBD como sua referência. O código Verilog do bloco EPBD se encontra disponível para apreciação. Verifica-se que as linhas 11 e 13 representam o processamento do bit q_i, conforme a Equação lógica 3.3.

As linhas 11, 12 e 15 a 23 implementam um multiplexador de quatro canais, sendo o sinal de entrada A_i e o sinal de saída q_i as chaves seletoras. Esta implementação representa o processamento do bit menos significativo a ser utilizado no somador SKS2.

```
1  module elemento_processador_borda_direita (mi, bi, mbi, ai, ri, qo, muxout)
2    input mi, bi, mbi, ai, ri;
3    output muxout, qo;
4
5    wire [1:0] mux_sel;
6    wire mux_input1;
7    wire qo;
8    reg muxout;
9    assign mux_input1 = ((ai & bi) ^ ri);
10   assign mux_sel = {ai, mux_input1};
11   assign qo = mux_input1;
12   always @(mbi or bi or mi or mux_sel) begin
13     case (mux_sel)
14       2'b11 : muxout = mbi;
15       2'b10 : muxout = bi;
16       2'b01 : muxout = mi;
17       2'b00 : muxout = 1'b0;
18       default muxout = 1'b0;
19     endcase
20   end
21 endmodule
```

3.2.9 Bloco Elemento Processador

Este bloco é responsável pela seleção do j-ésimo bit a ser utilizado na entrada b do somador SKS2 dos blocos MMMMult e MMMPot.

O bloco Elemento Processador é composto por um multiplexador de quatro canais com uma chave seletora de dois bits. Os sinais de entrada deste bloco são os bits B_j, M_j, MB_j, A_i e q_i. O sinal de saída deste bloco é o bit *muxout*.

O bloco Elemento Processador é uma versão mais simples do bloco EPBD. A única função a ser realizada pelo bloco Elemento Processador é selecionar os j-ésimos bits utilizado no somador SKS2 do bloco Multiplicador Modular de Montgomery, uma vez que o sinal de entrada q_i já foi gerado pelo bloco Elemento Processador Borda Direita.

A combinação dos bits A_i e q_i selecionam quais bits serão utilizados no somador SKS2 para a respectiva iteração da variável i. Essa operação corresponde as linhas sete a 11 do Algoritmo 5, quando a variável j é diferente de zero, ou seja, quando $0 < j < (n+2)$, onde neste caso, $0 < j < 6$. Os possíveis valores que a saída *muxout* pode assumir estão presentes na Tabela 8.

As Figuras 34 e 35 ilustram o bloco gerado através do software Quartus e o diagrama RTL do bloco respectivamente.

A quantidade de vezes de utilização do bloco Elemento Processador na arquitetura

Tabela 8: Resultado da expressão da linha sete a 11 do Algoritmo 5.

a_i	q_i	x_j^i
0	0	0
0	1	m_j
1	0	b_j
1	1	mb_j

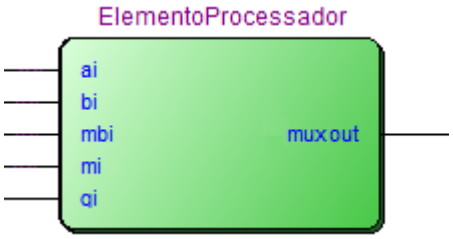

Figura 34: Bloco Elemento Processador do coprocessador RSA de quatro bits.

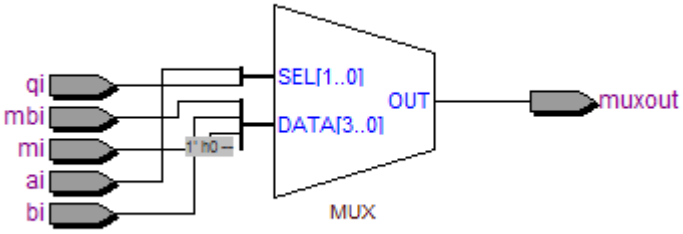

Figura 35: Diagrama RTL do bloco Elemento Processador do coprocessador RSA de quatro bits.

projetada depende da quantidade de bits do coprocessador. A fórmula genérica da quantidade de blocos Elemento Processador por bloco Multiplicador Modular de Montgomery é dada por $n + 1$, onde n é a quantidade de bits do coprocessador. Denominou-se a sigla EPx como sua referência genérica. Neste caso, o bloco Elemento Processador é utilizado cinco vezes dentro de cada bloco Multiplicador Modular de Montgomery e denominou-se as siglas EP1, EP2, EP3, EP4 e EP5 para sua referência.

O código Verilog do bloco EP também se encontra disponível para apreciação. Pode-se verificar que a linha oito e as linhas dez a 18 implementam um multiplexador de quatro canais, sendo os sinais de entrada A_i e q_i as chaves seletoras. Esta implementação representa o processamento dos j-ésimos bits a ser utilizado no somador SKS2.

```
1 module elemento_processador (mj, bj, mbj, ai, qi, muxout);
2   input mj, bj, mbj, ai, qi;
3   output muxout;
4
5   reg muxout;
6   wire [1:0] mux_sel;
7   assign mux_sel = {ai, qi};
```

```
8   always @(mbj or bj or mj or mux_sel) begin
9     case (mux_sel)
10      2'b11 : muxout = mbj;
11      2'b10 : muxout = bj;
12      2'b01 : muxout = mj;
13      2'b00 : muxout = 1'b0;
14      default muxout = 1'b0;
15    endcase
16  end
17 endmodule
```

3.2.10 Bloco Registrador Serializador

Este bloco é responsável por serializar os bits da palavra de entrada A do bloco Multiplicador Modular de Montgomery.

O bloco Registrador Serializador é composto por um registrador de seis bits e um multiplexador com dois canais de seis bits. Os sinais de entrada deste bloco são *enable*, *clock*, *lda*, *reset* e uma palavra A de seis bits. O único sinal de saída deste bloco é o bit *Ai*.

As Figuras 36 e 37 ilustram o bloco gerado através do software Quartus o diagrama RTL respectivamente.

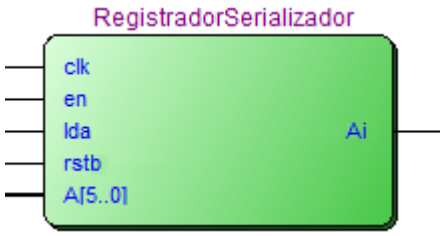

Figura 36: Bloco Registrador Serializador do coprocessador RSA de quatro bits.

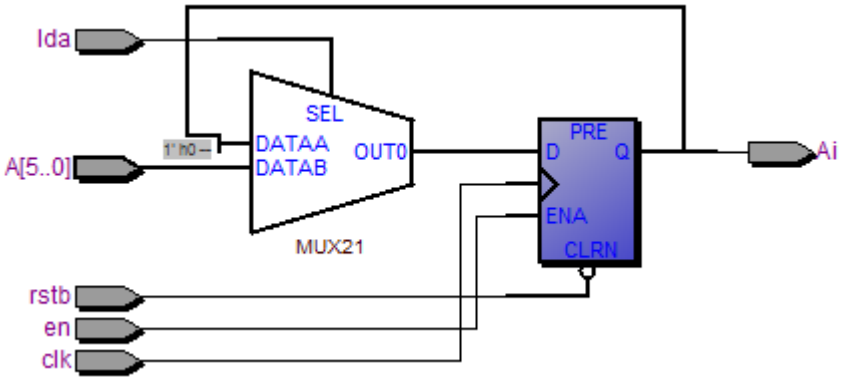

Figura 37: Diagrama RTL do bloco Registrador Serializador do coprocessador RSA de quatro bits.

O bloco Registrador Serializador é utilizado uma vez em cada bloco Multiplicador Modular de Montgomery e denominou-se a sigla RegSer como sua referência.

O sinal de entrada *lda* é a chave seletora do multiplexador. Quando o sinal *lda* está em nível lógico alto, a palavra de entrada A é transferida para a saída do multiplexador e carregada no registrador na transição positiva do sinal de *clock*. Quando o sinal *lda* está em nível lógico baixo, o sinal de saída do registrador com um deslocamento de um bit para a direita é transferido para a saída do multiplexador e carregado no registrador na transição positiva do sinal de *clock*.

O sinal de saída Ai em conjunto com o sinal qi, produzido pelo bloco EPBD, selecionam quais bits serão utilizados nos somadores SKS2 dos blocos MMMMult e MMMPot.

O código Verilog deste bloco se encontra disponível para apreciação.

```
1  module registrador_serializador (en, rstb, clk, lda, A, Ai);
2    input en;
3    input rstb;
4    input clk;
5    input lda;
6    input [5:0] A;
7    output Ai;
8
9    reg [5:0] A_aux;
10   assign Ai = A_aux[0];
11   always @(negedge(rstb) or posedge(clk)) begin
12     if (!rstb) begin
13       A_aux <= {6{1'b0}};
14     end else begin
15       if (en) begin
16         if (lda == 1'b1) begin
17           A_aux <= A;
18         end else begin
19           A_aux <= {A_aux >> 1};
20         end
21       end
22     end
23   end
24 endmodule
```

3.2.11 Bloco Registrador Soma

Este bloco é responsável pelo armazenamento temporário da saída do bloco somador SKS2. A saída do bloco somador SKS2 é conectada a este registrador com uma ressalva importante.

A linha cinco do Algoritmo 2 apresenta a expressão $S + a_i \times B + q_i \times M$. Esta

expressão corresponde a um valor intermediário do resultado da multiplicação modular de Montgomery. Entretanto, após o cálculo da expressão $S + a_i \times B + q_i \times M$, deve-se dividir o resultado pelo número dois.

Em notação binária, a divisão pelo número dois pode ser realizada através de um simples deslocamento de um bit para a direita. Dessa forma, o dado armazenado pelo bloco Registrador Soma é o resultado da soma produzida pelo bloco somador SKS2 deslocado um bit para a direita.

O bloco Registrador Soma é composto por um registrador de seis bits e um multiplexador com dois canais de seis bits. Os sinais de entrada deste bloco são *enable*, *clock*, *lda*, *reset* e uma palavra *rjo* de seis bits. O único sinal de saída deste bloco é uma palavra *regrji* de seis bits.

As Figuras 38 e 39 ilustram o bloco gerado através do software Quartus e o diagrama RTL respectivamente.

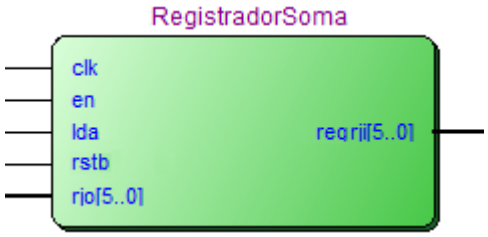

Figura 38: Bloco Registrador Soma do coprocessador RSA de quatro bits.

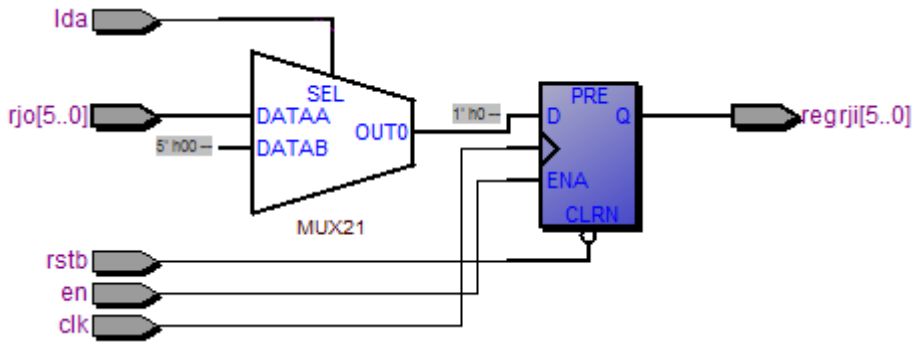

Figura 39: Diagrama RTL do bloco Registrador Soma do coprocessador RSA de quatro bits.

O bloco Registrador Soma é utilizado uma vez em cada bloco Multiplicador Modular de Montgomery e denominou-se a sigla RegSoma como sua referência.

O sinal de entrada *lda* é a chave seletora do multiplexador. Quando o sinal *lda* está em nível lógico alto, uma constante de valor 0 é transferida para a saída do multiplexador e carregada no registrado na transição positiva do sinal de *clock*. Quando o sinal *lda* está em nível lógico baixo, o sinal de entrada *rjo* é transferido para a saída do multiplexador

e carregado no registrador com um bit deslocado para a direita na transição positiva do sinal de *clock*.

O canal do multiplexador com a constante de valor 0 foi projetado pois o valor inicial do registrador bloco Somador deve ser igual a zero, conforme indica a linha um do Algoritmo 2. O código Verilog deste bloco se encontra disponível para apreciação.

```
1  module registrador_soma (en, rstb, clk, lda, rjo, regrji);
2    input en;
3    input rstb;
4    input clk;
5    input lda;
6    input [5:0] rjo;
7    output [5:0] regrji;
8
9    reg [5:0] regrji;
10   always @(negedge(rstb) or posedge(clk)) begin
11     if (!rstb) begin
12       regrji <= {6{1'b0}};
13     end else begin
14       if (en) begin
15         if (lda == 1'b1) begin
16           regrji <= {6{1'b0}};
17         end else begin
18           regrji <= (rjo >> 1);
19         end
20       end
21     end
22   end
23 endmodule
```

3.2.12 Bloco Registrador Produto Modular de Montgomery

Este bloco é responsável pelo armazenamento do resultado do algoritmo de multiplicação modular de Montgomery. O bloco Registrador Produto Modular de Montgomery é composto por um registrador cujos sinais de controle são provenientes do bloco FSM. A principal característica que este bloco apresenta é o controle do resultado da operação de multiplicação modular de Montgomery quando a mesma não deve ser executada.

Os sinais de entrada deste bloco são *enable*, *clock*, *reset*, *ldr*, *lock* , uma palavra *regrji* de seis bits e uma palavra A de seis bits. O único sinal de saída deste bloco é uma palavra R de seis bits.

No Algoritmo 4, o cálculo da linha seis está condicionado a ser executado a partir do valor lógico do bit do expoente em análise a cada iteração da variável i. Quando o bit do expoente é igual a nível baixo, a operação de multiplicação modular de Montgomery

não deve ser executada, o que implica em utilizar um artifício para bloquear a execução da operação. O artifício utilizado para bloquear o bloco MMMMult foi a inserção do sinal de controle *lock*. Nos casos em que a multiplicação modular de Montgomery não deve ser realizada, a palavra de entrada A é copiada diretamente para a saída R. O sinal de entrada que determina esta cópia é o sinal *lock*, produzido pelo bloco FSM nos respectivos estados de análise do bit do expoente.

As Figuras 40 e 41 ilustram o bloco gerado através do software Quartus e o diagrama RTL respectivamente.

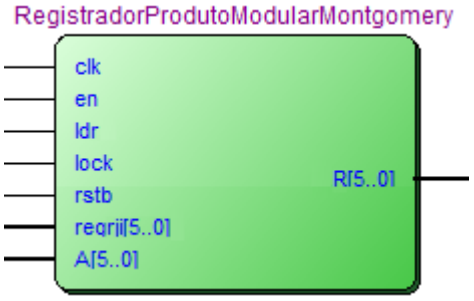

Figura 40: Bloco Registrador Produto Modular do coprocessador RSA de quatro bits.

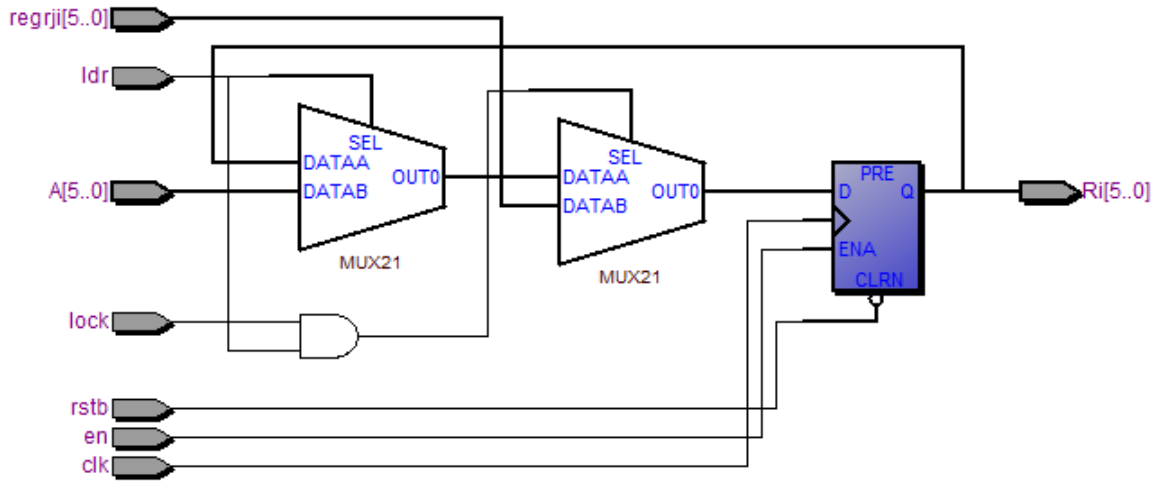

Figura 41: Diagrama RTL do bloco Registrador Produto Modular do coprocessador RSA de quatro bits.

O bloco Registrador Produto Modular de Montgomery é utilizado uma vez em cada bloco Multiplicador Modular de Montgomery e denominou-se a sigla RegPMM como sua referência.

O código Verilog deste bloco também se encontra disponível para apreciação. Assim como nos blocos RegSer e RegSoma. Pode-se observar na linha 26 do código Verilog o teste do valor lógico dos sinais *ldr* e *lock*. Durante a transição positiva do sinal *clock*, quando os sinais *ldr* e *lock* estão em nível lógico alto, o registrador deve ser atualizado, pois trata-se do fim do ciclo de cálculo da multiplicação modular de Montgomery. No caso

do sinal *lock* estiver em nível lógico baixo, o registro faz uma cópia do sinal de entrada A para o sinal de saída, operação programada na linha 30 do código Verilog. Nos demais casos, o sinal de saída R permanece inalterado.

```
1  module registrador_produto_modular_montgomery (en, rstb, clk, lock, ldr,
       regrji, A, R);
2    input en;
3    input rstb;
4    input clk;
5    input lock;
6    input ldr;
7    input [5:0] regrji;
8    input [5:0] A;
9    output [5:0] R;
10
11   reg [5:0] R;
12   always @(negedge(rstb) or posedge(clk)) begin
13     if (!rstb) begin
14       R <= {6{1'b0}};
15     end else begin
16       if (en) begin
17         if ((lock == 1'b1) && (ldr == 1'b1)) begin
18           R <= regrji;
19         end else begin
20           if (ldr == 1'b1) begin
21             R <= A;
22           end else begin
23             R <= R;
24           end
25         end
26       end
27     end
28   end
29 endmodule
```

3.3 Geração da chave para coprocessador de quatro bits

Uma vez apresentados e discutidos os algoritmos envolvidos no processo de multiplicação modular de Montgomery e exponenciação modular, bem como os blocos utilizados no desenvolvimento do coprocessador de quatro bits, esta seção aborda o funcionamento interno do coprocessador, dando ênfase em como os blocos e sub-blocos do coprocessador interagem entre si e como os dados caminham por dentro da arquitetura projetada. Para esta explanação, as chaves com máximo quatro bits foram geradas da seguinte forma:

1. Selecionou-se dois números primos distintos, $p = 3$ e $q = 5$.

2. Calculou-se $n = p \times q = 3 \times 5 = 15$.
3. Calculou-se $\varphi(n) = (p-1) \times (q-1) = 2 \times 4 = 8$.
4. Selecionou-se e tal que e fosse relativamente primo a $\varphi(n) = 8$ e menor que $\varphi(n)$: escolheu-se $e = 7$.
5. Determinou-se d tal que $d \times e \equiv 1 \; mod \; 8$ e $d < 8$. Neste caso, o valor correto é $d = 7$, pois $7 \times 7 \equiv 49 \equiv 6 \times 8 + 1$.

As chaves resultantes são a chave pública (7, 15) e a chave privada (7, 15). É importante ressaltar que para um coprocessador RSA de quatro bits, independentemente da arquitetura projetada, não existe outra opção de módulo possível, uma vez que o módulo 15, produto entre os números primos três e cinco, é o maior número que pode ser representado com quatro bits em notação binária.

Na quarta etapa, a escolha do expoente igual a sete foi proposital para não repetir os primos utilizados para gerar o módulo das chaves. Coincidentemente, para esta escolha, tem-se que a chave para cifragem é igual a chave para a decifragem. Pode-se verificar que este par de chaves não é uma escolha sábia para uso comercial. Entretanto, o par de chaves gerado será utilizado para demonstrar o funcionamento do coprocessador desenvolvido.

Para cifrar o texto pleno P, deve-se utilizar a Equação 3.4, enquanto para decifrar um texto cifrado, deve-se utilizar a Equação 3.5.

$$C = P^7 \; mod \; 15 \tag{3.4}$$

$$P = C^7 \; mod \; 15 \tag{3.5}$$

Para o módulo $M = 15$, tem-se que a quantidade de bits necessárias para o cálculo da exponenciação modular é igual a seis, conforme já relatado nas seções anteriores. Dado um texto pleno $P = 2$ e utilizando-se do software apresentado no Apêndice B, com parâmetros $E = 7$, $M = 15$ e $qtddbits = 6$, pode-se calcular a constante de Montgomery.

A constante de Montgomery depende somente da quantidade de bits utilizados no algoritmo e do valor do módulo. A constante de Montgomery é calculada no software do Apêndice B através das linhas 32 a 34. A arquitetura projetada neste trabalho não apresenta um bloco para cálculo da constante de Montgomery. A constante de Montgomery é um dado de entrada do bloco RSA Topo que deve ser calculada externamente e disponibilizada no coprocessador para o funcionamento correto. Para o exemplo em questão, tem-se que o valor da constante de Montgomery é igual a um.

3.4 Fluxo dos dados coprocessador RSA quatro bits

Com os dados de entrada fixos em $P = 2$, $E = 7$, $M = 15$ e $Constm = 1$, inicia-se o processo de cifragem com a arquitetura projetada. Inicialmente, habilita-se o circuito colocando o sinal de *enable* do bloco RSA Topo em nível lógico alto. Em seguida, um pulso negativo do sinal de *reset* deve ser realizado para colocar em estado inicial todo o coprocessador. Durante o processo de *reset*, todos os registradores do coprocessador são zerados. Na primeira transição de nível alto do sinal de *clock* após o retorno do sinal de *reset* para nível alto, o coprocessador vai para o estado 1.

O bloco FSM, no estado 1, através do sinal *sel2*, seleciona o sinal de entrada $Constm$ para o sinal de saída *out* do bloco MuxA. Ao mesmo tempo, o sinal *sel2* seleciona o sinal de entrada P para o sinal de saída *out* do bloco MuxB. Dessa forma, as entradas A e B do bloco MMMPot são $Constm$ e P, respectivamente, conforme indica a linha dois do Algoritmo 4. O sinal *sel2* será referenciado a partir deste momento por *selMux* para facilitar a identificação dos blocos ao qual o sinal está conectado.

O bloco FSM, ainda no estado 1, através do sinal *sel1*, seleciona o sinal de entrada $Constm$ para o sinal de saída *out* do bloco MuxModA. Ao mesmo tempo, o sinal *sel1* seleciona o sinal de entrada 1 para o sinal de saída *out* do bloco MuxModB. Dessa forma, as entradas A e B do bloco MMMMult são $Constm$ e 1, respectivamente, conforme a indicação da linha três do Algoritmo 4. O sinal *sel1* será referenciado a partir deste momento por *selMuxMod* para facilitar a identificação dos blocos ao qual o sinal está conectado.

O bloco FSM, ainda no estado 1, mantém os sinais de saída *lock1* e *lock2* em nível lógico baixo para não travar as operações de mapeamento do algoritmo do exponenciação modular. Os sinais *lock1* e *lock2* serão referenciados a partir deste momento, respectivamente, por *lockMult* e *lockPot* para facilitar a identificação dos blocos ao qual estes sinais estão conectados.

Ainda no estado 1, o sinal de saída *lda* do bloco FSM é mantido em nível lógico alto para carregar o valor do sinal de entrada A do blocos MMMMult e MMMPot dentro dos respectivos sub-blocos RegSer. Isto é, o sinal $Constm$, cujo valor 1, é carregado em cada bloco RegSer contido nos blocos MMMPot e MMMMult. Ao mesmo tempo, o sinal de saída *lda* limpa o valor dos blocos RegSoma contido nos blocos MMMPot e MMMMult.

Ainda no estado 1, o sinal de saída *ldr* do bloco FSM é mantido em nível lógico baixo pois os blocos RegPMM, que estão contidos dentro dos blocos MMMPot e MMMMult, só devem ser atualizados no final da operação de multiplicação modular de Montgomery. O sinal de saída *eoc* do bloco FSM também é mantido em nível lógico baixo pois o bloco RegCript só deve ser atualizado no final da operação de exponenciação modular.

A segunda transição de nível alto do sinal de *clock* coloca o coprocessador no estado 2. No estado 2, o bloco FSM mantém todos os seus sinais de saída inalterados, com exceção do sinal *lda*, uma vez que os blocos RegSer contido nos blocos MMMPot e MMMMult já foram carregados no estado 1.

Dentro do bloco MMMPot, a operação de mapeamento começa a ser realizada. O bloco RegSer serializa o bit menos significativo do sinal de entrada A, cujo valor é 1, através do sinal de saída Ai. O bloco SKS1 calcula a soma dos sinais de entrada B e M, cujos valores são respectivamente 2 e 15, e disponibiliza o resultado 17 no sinal de saída *sum*.

O bloco EPBD processa o valor do sinal de saída qi, neste caso igual a 0, que em conjunto com o sinal de entrada Ai, seleciona o valor do sinal de saída *muxout* do bloco EPBD, neste caso igual a 0. Os blocos EP processam os sinais de entrada qi e Ai e selecionam os valores dos sinais de saída *muxout*. O conjunto dos sinais de saída *muxout* dos blocos EPBD e EP compõem o sinal de entrada b dos blocos SKS2. O bloco SKS2, por sua vez, calcula a soma dos sinais de entrada b, recém processada pelos blocos EPBD e EP, e dos sinal de entrada a, cujo valor é o sinal de saída do bloco RegSoma.

O sinal de saída *sum* do bloco SKS2 é disposto no sinal de entrada do bloco RegSoma para ser atualizado na próxima transição do sinal de *clock*. O bloco RegPMM não realiza nenhuma operação pois os sinais *lockPot*, e *ldr* permanecem em nível lógico baixo.

Assim como no bloco MMMPot, o bloco MMMMult realiza exatamente os mesmo passos, sendo que os sinais de saída de cada bloco variam de acordo com as entradas A, B e M, cujos valores são respectivamente 1, 0 e 15.

As Tabelas 9 e 10 apresentam os valores dos sinais de saída de cada um dos blocos MMMPot e MMMMult, respectivamente para o estado 2 do coprocessador.

Tabela 9: Sinais de saída dos sub-blocos do bloco MMMPot no estado 2 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	1
Bloco SKS1	sum	17
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	2
Bloco SKS2	rjo	2
Bloco RegSoma	$regrji$	0
Bloco RegPMM	R	0

A terceira transição de nível alto do sinal de *clock* coloca o coprocessador no estado 3. No estado 3, o bloco FSM mantém todos os seus sinais de saída inalterados. Os blocos MMMPot e MMMMult realizam as operações conforme reportadas no estado 3. As Tabelas 11 e 12 apresentam os valores dos sinais de saída de cada um dos blocos MMMPot e MMMMult, respectivamente para o estado 3 do coprocessador.

Tabela 10: Sinais de saída dos sub-blocos do bloco MMMMult no estado 2 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	1
Bloco SKS1	sum	16
Bloco EPBD	qo	1
Bloco EPBD + Blocos EP	$muxout$	16
Bloco SKS2	rjo	16
Bloco RegSoma	$regrji$	0
Bloco RegPMM	R	0

Tabela 11: Sinais de saída dos sub-blocos do bloco MMMPot no estado 3 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	1
Bloco EPBD + Blocos EP	$muxout$	15
Bloco SKS2	rjo	16
Bloco RegSoma	$regrji$	1
Bloco RegPMM	R	0

Tabela 12: Sinais de saída dos sub-blocos do bloco MMMMult no estado 3 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	8
Bloco RegSoma	$regrji$	8
Bloco RegPMM	R	0

A quarta transição de nível alto do sinal de *clock* coloca o coprocessador no estado 4. Assim como no estado 3, o bloco FSM mantém todos os seus sinais de saída inalterados e os blocos MMMPot e MMMMult continuam o processo da operação de multiplicação modular de Montgomery. O coprocessador repete esta sequência por mais quatro vezes, passando pelos estados 4, 5, 6 e 7. As tabelas 13 e 14, 15 e 16, 17 e 18, 19 e 20, apresentam os valores dos sinais de saída de cada um dos blocos MMMPot e MMMMult, respectivamente, para os estados 4, 5, 6 e 7 do coprocessador.

Tabela 13: Sinais de saída dos sub-blocos do bloco MMMPot no estado 4 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	1
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	8
Bloco RegSoma	$regrji$	8
Bloco RegPMM	R	0

Tabela 14: Sinais de saída dos sub-blocos do bloco MMMMult no estado 4 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	4
Bloco RegSoma	$regrji$	4
Bloco RegPMM	R	0

Tabela 15: Sinais de saída dos sub-blocos do bloco MMMPot no estado 5 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	4
Bloco RegSoma	$regrji$	4
Bloco RegPMM	R	0

Tabela 16: Sinais de saída dos sub-blocos do bloco MMMMult no estado 5 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	2
Bloco RegSoma	$regrji$	2
Bloco RegPMM	R	0

Tabela 17: Sinais de saída dos sub-blocos do bloco MMMPot no estado 6 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	2
Bloco RegSoma	$regrji$	2
Bloco RegPMM	R	0

A oitava transição do nível alto do sinal de *clock* coloca o coprocessador no estado 8. No estado 8, o bloco FSM mantém todos os seus sinais de saída inalterados com exceção do sinal ldr, cujo valor é alterado para nível alto. O sinal ldr, quando em nível alto, habilita a escrita do valor do sinal de saída do bloco RegSoma no sinal de saída do bloco RegPMM. Esta etapa corresponde ao final do processo de multiplicação modular de Montgomery para os sinais de entrada dos blocos MMMPot e MMMMult definidas no estado 2. As Tabelas 21 e 22 apresentam os valores dos sinais de saída de cada um dos

Tabela 18: Sinais de saída dos sub-blocos do bloco MMMMult no estado 6 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	1
Bloco EPBD + Blocos EP	$muxout$	15
Bloco SKS2	rjo	16
Bloco RegSoma	$regrji$	1
Bloco RegPMM	R	0

Tabela 19: Sinais de saída dos sub-blocos do bloco MMMPot no estado 7 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	1
Bloco EPBD + Blocos EP	$muxout$	15
Bloco SKS2	rjo	16
Bloco RegSoma	$regrji$	1
Bloco RegPMM	R	0

Tabela 20: Sinais de saída dos sub-blocos do bloco MMMMult no estado 7 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	8
Bloco RegSoma	$regrji$	8
Bloco RegPMM	R	0

blocos MMMPot e MMMMult, respectivamente, para o estado 8.

Além disso, no estado 8, o bloco FSM carrega o expoente E que será utilizado durante a parte iterativa do Algoritmo 4. A quantidade de transições de nível do sinal de *clock* para realizar uma multiplicação modular de Montgomery é exatamente igual a oito para o coprocessador de quatro bits, respeitando a equação genérica $(n+4)$ desenvolvida na elaboração da arquitetura.

Tabela 21: Sinais de saída dos sub-blocos do bloco MMMPot no estado 8 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	17
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	8
Bloco RegSoma	$regrji$	8
Bloco RegPMM	R	0

Tabela 22: Sinais de saída dos sub-blocos do bloco MMMMult no estado 8 do coprocessador

Nome do Bloco	Sinal de saída	Valor
Bloco RegSer	Ai	0
Bloco SKS1	sum	16
Bloco EPBD	qo	0
Bloco EPBD + Blocos EP	$muxout$	0
Bloco SKS2	rjo	4
Bloco RegSoma	$regrji$	4
Bloco RegPMM	R	0

A nona transição de nível alto do sinal de *clock* coloca o coprocessador no estado 9. Neste estado, o bloco FSM altera o sinal de saída *selMux* para reajustar o sinal de entrada do bloco MMMPot conforme a linha oito Algoritmo 4. Isto é, o sinal de saída *selMux* do bloco FSM seleciona o sinal de saída R do bloco MMMPot para o sinal de saída *out* do bloco MuxA, cujo novo valor é igual a 8. Ao mesmo tempo, o sinal de saída *selMux* do bloco FSM seleciona o sinal de saída R para o sinal de saída *out* do bloco MuxB, cujo novo valor é igual a 8. O sinal de saída R do bloco MMMPot será referenciado a partir deste momento por *PotR* para facilitar a identificação do bloco ao qual está conectado. Analogamente, o sinal de saída R do bloco MMMMult será referenciado a partir deste momento por *MultR* para facilitar a identificação do bloco ao qual está conectado.

O bloco FSM, ainda no estado 9, altera também o sinal de saída *selMuxMod* para reajustar o sinal de entrada do bloco MMMMult conforme a linha seis Algoritmo 4. Isto é, o sinal de saída *selMuxMod* seleciona o sinal *MultR* para o sinal de saída *out* do bloco MuxModA, cujo novo valor é igual a 4. Ao mesmo tempo, o sinal de saída *selMuxMod* seleciona o sinal *PotR* para o sinal de saída *out* do bloco MuxModB, cujo novo valor é igual a 8.

A partir do estado 9, o bloco FSM apresenta as mesmas características que as reportas entre os estados 1 a 8 nas próximas oito transições de nível alto do sinal de *clock*, com exceção de que o sinal de *lockMult* contém o valor lógico do bit menos significativo do expoente E. Para este caso, o *lockMult* é igual a 1, o que faz com que o bloco MMMMult não trave o cálculo da linha seis do Algoritmo 4 para a iteração 0.

O estado 17 corresponde ao fim do cálculo da multiplicação modular de Montgomery da iteração 0 do Algoritmo 4. A Tabela 23 apresenta os valores dos sinais *lockMult* e os resultados dos blocos MMMPot e MMMMult para o coprocessador no estado 17, final da iteração 0 do Algoritmo 4.

Ao final da iteração 0, a variável iterativa i do Algoritmo 4 é incrementada e novamente novas multiplicações modulares de Montgomery são realizadas até que a variável i chegue no valor 5. Seguindo a regra de oito ciclos de *clock* para cada multiplicação modular de Montgomery, os estados 25, 33 e 41 do coprocessador correspondem, respectivamente,

Tabela 23: Sinais de saída no estado 17 do coprocessador - final da iteração 0

Nome do Bloco	Sinal de saída	Valor
Bloco FSM	*lockMult*	1
Bloco MMMPot	*PotR*	1
Bloco MMMMult	*MultR*	8

ao final das iterações 1, 2 e 3 do Algoritmo 4.

As Tabelas 24, 25 e 26 apresentam os valores dos sinais *lockMult* e os resultados dos blocos MMMPot e MMMMult, respectivamente, para o coprocessador nos estado 25, 33 e 41.

Tabela 24: Sinais de saída no estado 25 do coprocessador - final da iteração 1

Nome do Bloco	Sinal de saída	Valor
Bloco FSM	*lockMult*	1
Bloco MMMPot	*PotR*	4
Bloco MMMMult	*MultR*	2

Tabela 25: Sinais de saída no estado 33 do coprocessador - final da iteração 2

Nome do Bloco	Sinal de saída	Valor
Bloco FSM	*lockMult*	0
Bloco MMMPot	*PotR*	4
Bloco MMMMult	*MultR*	2

Tabela 26: Sinais de saída no estado 41 do coprocessador - final da iteração 3

Nome do Bloco	Sinal de saída	Valor
Bloco FSM	*lockMult*	0
Bloco MMMPot	*PotR*	4
Bloco MMMMult	*MultR*	2

No exemplo utilizado, as iterações 2 e 3 do Algoritmo 4 não atualizam a saída do bloco MMMMult, uma vez que o sinal *lockMult* do bloco FSM se encontra em nível lógico baixo.

O estado 49 caracteriza o final do processo iterativo do Algoritmo 4. Neste estado, todos os bits do expoente E foram serializados e o cálculo da exponenciação modular no domínio Montgomery foi finalizado. Para voltar o resultado para o domínio ordinário, realiza-se mais uma operação de multiplicação modular de Montgomery conforme a indicação da linha dez do Algoritmo 4. O remapeamento é realizado através do bloco MMMMult.

O bloco FSM, no estado 49, através do sinal *selMuxMod* seleciona o sinal 1 para o sinal de saída *out* do bloco MuxModA. Ao mesmo tempo, o sinal *selMuxMod* seleciona o sinal $MultR$ para o sinal de saída *out* do bloco MuxModB. Dessa forma, as entradas A e B do bloco MMMMult passam a ser 1 e $MultR$, cujos valores são, respectivamente, 1 e

2. O sinal de saída *selMux* permanece inalterado pois a saída *lockPot* é mantida em nível lógico baixo a partir deste estado uma vez que o bloco MMMPot não será mais utilizado na etapa de remapeamento.

A Tabela 27 apresenta o valor dos sinais *lockMult* e os resultados dos blocos MMMPot e MMMMult para o coprocessador no estado 49.

Tabela 27: Sinais de saída no estado 49 do coprocessador - final da iteração 4

Nome do Bloco	Sinal de saída	Valor
Bloco FSM	*lockMult*	1
Bloco MMMPot	*PotR*	1
Bloco MMMMult	*MultR*	8

As próximas oito transições do sinal de entrada *clock* fazem o coprocessador percorrer os estados 49 a 57, que finalizam o remapeamento do cálculo da exponenciação modular. No estado 57, o bloco FSM altera o valor do sinal de saída *eoc* para nível lógico alto.

Na transição 58 de nível alto do sinal de *clock*, o sinal de saída *MultR* do bloco MMMMult é carregado no bloco RegCript. Sendo assim, o sinal de saída *C* do bloco RSA Topo é então atualizado para o valor do sinal de entrada do bloco RegCript, cujo valor no exemplo é igual a 8. Calculando-se manualmente o valor da expressão $2^7 \ mod \ 15$, tem-se que o resultado é igual a 8, conforme apresenta a saída do coprocessador.

A quantidade de transições de nível do sinal de *clock* para realizar um processo de exponenciação modular é exatamente igual a 56 para um coprocessador de quatro bits, respeitando a equação genérica $(n + 4) \times (n + 3)$ desenvolvida na elaboração da arquitetura. Mais duas transições de nível de *clock* são utilizadas para dispor o resultado em um registrador para leitura externa. A Figura 42 ilustra o resultado da simulação do processo de cifragem.

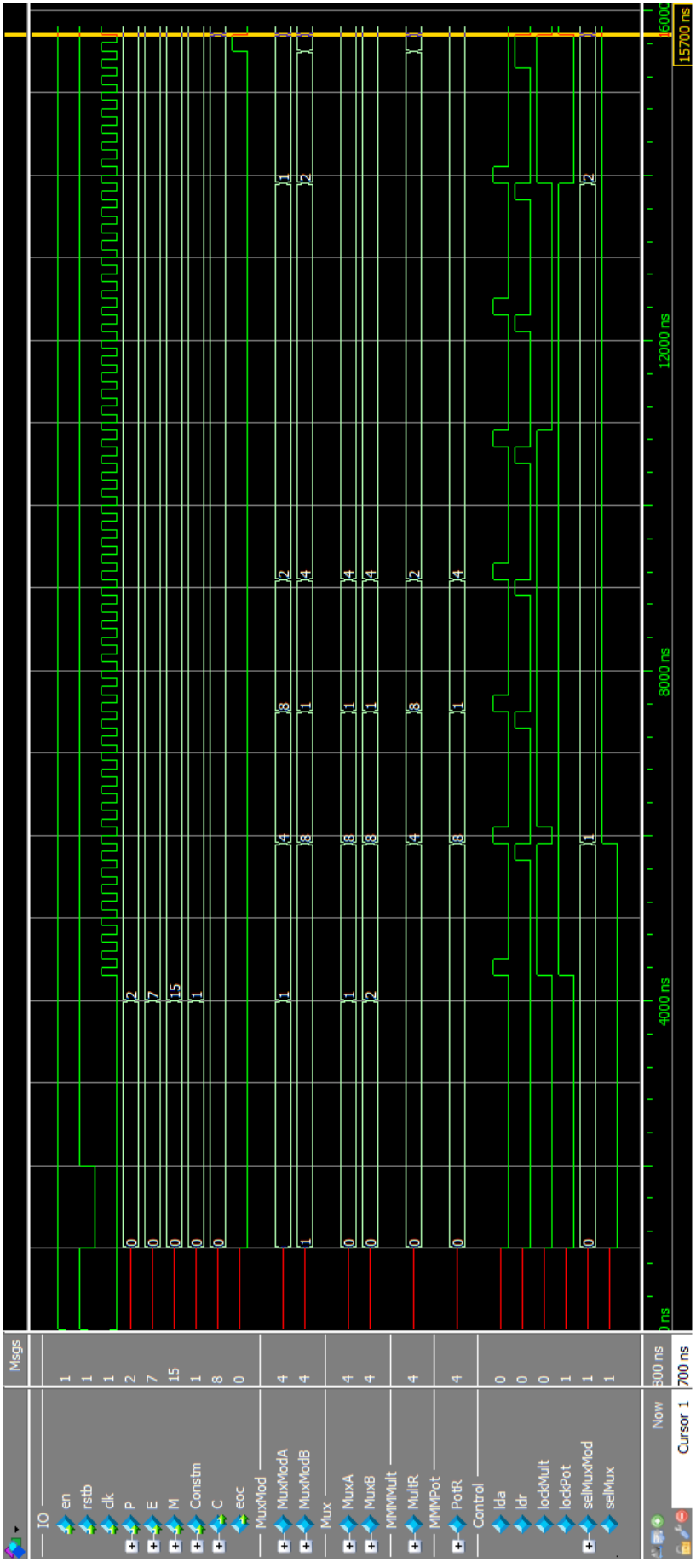

Figura 42: Simulação processo de cifragem para arquitetura de 4 bits.

3.5 Coprocessador RSA de 1024 bits

A estrutura hierárquica do coprocessador de 1024 bits é semelhante à estrutura apresentada na Figura 18 para o coprocessador de quatro bits. Os sinais de entrada e saída de todos blocos permaneceram inalterados, ou seja, sem influenciar a lógica de cada um dos blocos apresentados, salvo o aumento da quantidade de bits.

Vale ressaltar que para uma implementação de n bits, o número de bits utilizados no algoritmo são $n + 2$, ou seja, neste caso, 1024 mais 2, totalizando 1026 bits. Sendo assim, internamente ao bloco RSA Topo, todos os sub-blocos realizam processamento com palavras de tamanho igual a 1026 bits, em vez de 1024 bits [1] .

3.6 Diferenças Coprocessador RSA 1024 bits

A arquitetura apresentada no coprocessador de quatro bits foi desenvolvida com a finalidade de reutilização máxima dos blocos no reprojeto de coprocessadores com diferentes quantidades de bits. Entretanto, alguns blocos para o coprocessador de 1024 bits tiveram que ser reprojetados para se adaptarem à arquitetura de 1024 bits. As diferenças dos blocos são apresentadas nas subseções seguir.

3.6.1 Bloco RSA Topo

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.1 e ilustrado na Figura 19. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada P, E, M e $Const_m$ de quatro bits para 1024 bits. A quantidade de bits do sinal de saída C também aumentou de quatro bits para 1024 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas.

3.6.2 Bloco Multiplexador

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.2 e ilustrado na Figura 21. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada a e b de seis bits para 1026 bits. A quantidade de bits do sinal de saída *out* também aumentou de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador tinha dois canais de seis bits, na arquitetura de 1024 bits o multiplexador passou a ter dois canais de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de seleção entre os canais não foi modificada.

[1] Para maiores detalhes ver seção 2.2.1 e seção 3.2.1

3.6.3 Bloco Multiplexador Modificado

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.3 e ilustrado na Figura 23. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada a e b de seis bits para 1026 bits. A quantidade de bits do sinal de saída *out* também aumentou de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador modificado tinha quatro canais de seis bits, na arquitetura de 1024 bits o multiplexador passou a ter quatro canais de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de seleção entre os canais não foi modificada.

3.6.4 Bloco Registrador Criptografia

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.4 e ilustrado na Figura 25. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits do sinal de entrada R de seis bits para 1026 bits. A quantidade de bits do sinal de saída *Cex* também aumentou de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador e o registrador processavam dados de seis bits, na arquitetura de 1024 bits o multiplexador e o registrador passaram a processar dados de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de seleção do dado a ser registrado não foi modificada.

3.6.5 Bloco Máquina de Estados Finitos

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.5 e ilustrado na Figura 27. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits do sinal de entrada *expE* de seis bits para 1026 bits e o reprojeto da máquina de estados para a nova quantidade de bits. A equação genérica da quantidade de estados necessárias para a arquitetura desenvolvida é $(n + 4) \times (n + 3)$, onde n é a quantidade de bits. O fator $(n + 4)$ corresponde a quantidade de estados e ciclos de *clock* para realizar uma operação de multiplicação modular de Montgomery e o fator $(n + 3)$ corresponde à quantidade de bits do expoente. Enquanto na arquitetura de quatro bits são necessários 56 estados, na arquitetura de 1024 bits são necessários 1059870 estados. Os demais sinais permaneceram com a quantidade de bits inalteradas.

3.6.6 Bloco Multiplicador Modular de Montgomery

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.6 e ilustrado na Figura 28. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada A, B e M de seis bits para 1026 bits. A quantidade de bits do sinal de saída R também aumentou de seis bits para 1026 bits.

A operação lógica da multiplicação modular de Montgomery permaneceu inalterada, salvo o aumento da quantidade de ciclos de *clock* para concluir a operação. A equação genérica da arquitetura para realizar a operação de multiplicação modular de Montgomery é dada por $(n+4)$, onde n é a quantidade de bits. Enquanto para a arquitetura de quatro bits oito ciclos de *clock* eram necessários, para a arquitetura de 1024 bits são necessários 1028 ciclos de *clock*.

A quantidade de utilização dos sub-blocos Somador Kogge-Stone, Registrador Serializador, Registrador Soma, Registrador Produto Modular e Elemento Processador Borda Direita permaneceu com uma utilização na arquitetura. Entretanto, a quantidade de vezes de utilização do sub-bloco Elemento Processador aumentou de cinco para 1025.

3.6.7 Bloco Somador Kogge-Stone

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.7 e ilustrado na Figura 30. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada a e b de seis bits para 1026 bits. A quantidade de bits do sinal de saída *sum* também aumentou de seis bits para 1026 bits. O circuito que realiza o cálculo da soma entre os sinais de entrada foi reprojetado para o nova quantidade de bits. Independentemente da arquitetura de quatro bits ou 1024 bits, cada bloco Multiplicador Modular de Montgomery utiliza dois blocos Somador Kogge-Stone.

3.6.8 Elemento Processador Borda Direita

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.8 e ilustrado na Figura 32. Não foi realizada nenhuma alteração na quantidade de bits dos sinais de entrada e saída deste bloco para a arquitetura de 1024 bits. A lógica interna à este bloco também não foi modificada para a arquitetura de 1024 bits. Independentemente da arquitetura de quatro bits ou 1024 bits, cada bloco Multiplicador Modular de Montgomery utiliza um bloco Elemento Processador Borda Direita.

3.6.9 Elemento Processador

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.9 e ilustrado na Figura 34. Não foi realizada nenhuma alteração na quantidade de bits dos sinais de entrada e saída deste bloco para a arquitetura de 1024 bits. A lógica interna à este bloco também não foi modificada para a arquitetura de 1024 bits. Para a arquitetura de quatro bits, cada bloco Multiplicador Modular de Montgomery utiliza cinco blocos Elemento Processador. Para a arquitetura de 1024 bits cada bloco Multiplicador Modular de Montgomery utiliza 1025 blocos Elemento Processador.

3.6.10 Bloco Registrador Serializador

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.10 e ilustrado na Figura 36. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits do sinal de entrada A de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador e o registrador processavam dados de seis bits, na arquitetura de 1024 bits o multiplexador e o registrador passaram a processar dados de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de serialização do dado não foi modificada.

3.6.11 Bloco Registrador Soma

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.11 e ilustrado na Figura 38. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits do sinal de entrada *rjo* de seis bits para 1026 bits. A quantidade de bits do sinal de saída *regrji* também aumentou de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador e o registrador processavam dados de seis bits, na arquitetura de 1024 bits o multiplexador e o registrador passaram a processar dados de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de seleção do dado a ser registrado não foi modificada.

3.6.12 Bloco Registrador Produto Modular de Montgomery

Este bloco foi apresentado na arquitetura de quatro bits na seção 3.2.12 e ilustrado na Figura 40. Para a arquitetura de 1024 bits, as modificações realizadas foram aumentar a quantidade de bits dos sinais de entrada *regrji* e A de seis bits para 1026 bits. A quantidade de bits do sinal de saída R também aumentou de seis bits para 1026 bits. Enquanto na arquitetura de quatro bits o multiplexador e o registrador processavam dados de seis bits, na arquitetura de 1024 bits o multiplexador e o registrador passaram a processar dados de 1026 bits. Os demais sinais permaneceram com a quantidade de bits inalteradas e a lógica de seleção do dado para o registrador não foi modificada.

3.7 Exemplo coprocessador 1024 bits

Para exemplificar o coprocessador de 1024 bits utilizou-se um dos vetores de testes recomendados pelo RSA Laboratories [26]. Grande parte das literaturas e artigos utilizam a base de vetores de testes para comprovação e validação do processo de cifragem e decifragem do algoritmo RSA. Embora os vetores de testes recomendados pelo RSA Laboratories [26] apresentem somente os valores do módulo e dos expoentes das chaves públicas e privadas, a verificação de qualquer arquitetura é válida, pois independente-

mente do texto pleno utilizado na cifragem, o texto claro precisa ser recuperada a partir do texto cifrado e da chave complementar ao processo.

Neste exemplo apresentado para a arquitetura de 1024 bits, o módulo é definido pelo valor M, o expoente público é definido pelo valor E e o expoente privado é definido pelo valor D. Sendo assim, a chave pública é definida pelo par (E, M), enquanto a chave privada é definida pelo par (D, M). A Tabela 28 apresenta os valores das chaves a serem utilizadas, bem como o valor da constante *Constm* calculada via software. Os valores da Tabela 28 estão representados no formato hexadecimal. É importante ressaltar que o valor do expoente E escolhido é da forma $2^{2^x}+1$, onde quando $x = 4$, tem-se o valor 65537. Este valor é frequentemente utilizado em grande parte das literaturas, artigos e pelo próprio RSA Laboratories por ser o maior número primo da sequência de Fermat.

Tabela 28: Chave pública, chave privada e constante de Montgomery - coprocessador de 1024 bits

Sinal	Valor (Hexadecimal)
M	A8B3B284AF8EB50B387034A860F146C4 919F318763CD6C5598C8AE4811A1E0AB C4C7E0B082D693A5E7FCED675CF46685 12772C0CBC64A742C6C630F533C8CC72 F62AE833C40BF25842E984BB78BDBF97 C0107D55BDB662F5C4E0FAB9845CB514 8EF7392DD3AAFF93AE1E6B667BB3D424 7616D4F5BA10D4CFD226DE88D39F16FB
E	00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000010001
D	53339CFDB79FC8466A655C7316ACA85C 55FD8F6DD898FDAF119517EF4F52E8FD 8E258DF93FEE180FA0E4AB29693CD83B 152A553D4AC4D1812B8B9FA5AF0E7F55 FE7304DF41570926F3311F15C4D65A73 2C483116EE3D3D2D0AF3549AD9BF7CBF B78AD884F84D5BEB04724DC7369B31DE F37D0CF539E9CFCDD3DE653729EAD5D1
Constm	4E740858B51D3CFD90EBF3DE4F09FE88 84AD12F2E9D6EC181BC2E055B36F5A33 7CA9C40E68E710A035A2C44486FCC370 B317753B3E6B0D3C9D34A7ED50FB36D5 C5D6E8164E817BFF80F26281F73E0E1A C9EEDACF6589A06A94901A1B818873BA 2FC0F84EEF5343ABA27BEA0A1F76E5EC FC593499C0FA99615B9B6420694D26FB

Para o texto pleno, utilizou-se a cadeia de caracteres "UNIFEI - MG - Brasil - Mestrado - Desenvolvimento de Hardware de Criptografia RSA em Linguagem Verilog

- Caio Alonso da Costa"como exemplo. Primeiramente a cadeia de caracteres do texto pleno foi convertido para um conjunto de *bytes* seguindo o padrão ASCII. Em seguida, a sequência de *bytes* foi disponibilizado no sinal P de entrada no coprocessador, conforme indicado na Tabela 29. Com os sinais da chave pública (E, M) dispostos nas entradas E e M do coprocessador e a constante *Constm* calculada via software, o processo de cifragem foi iniciado. O coprocessador cifrou mensagem realizando o cálculo da Equação 3.6.

$$C = P^E \ mod \ M \tag{3.6}$$

Após 1055756 ciclos de *clock*, o processo de cifragem encerrou-se e obteve-se como resposta o texto cifrado C apresentado na Tabela 29. O resultado obtido foi comparado com o apresentado pelo software Java e o mesmo valor foi encontrado. O processo de decifragem foi realizado utilizando a chave privada (D, M) e a cadeia de caracteres utilizada como texto pleno foi recuperada. As Figuras 43 e 44 ilustram, respectivamente, os resultados das simulações dos processos de cifragem e decifragem. Sendo assim, a arquitetura proposta do coprocessador de 1024 bits é capaz de cifrar e decifrar dados conforme a descrição padrão do algoritmo de criptografia assimétrica RSA [11]. Para adaptar a arquitetura proposta e prepará-la para o uso, alguns adendos ao coprocessador foram realizados. Os adendos foram projetados para complementar a interface de acesso aos sinais de entrada P, E, M, *Constm* e de saída C, uma vez que todos esses sinais são barramentos de 1024 bits. Além da interface, uma pequena máquina de estados para iniciar o processo de cifragem ou decifragem foi incluído.

Tabela 29: Texto claro e texto cifrado - coprocessador de 1024 bits

Sinal	Valor (Hexadecimal)
P	554E49464549202D204D47202D204272 6173696C202D204D6573747261646F20 2D20446573656E766F6C76696D656E74 6F20646520486172647617265206465 2043726970746F677261666961205253 4120656D204C696E67756167656D2056 6572696C6F67202D204361696F20416C 6F6E736F20646120436F737461
C	55A2AFFF919E421560A273603DB2DF9A 476C3A9253B97A812D4DF991198CE0CF 015B6053E71947D96BF815D0A8EFEAAF C45FAA479E2303CC37B3616AE09A99B0 C3E662CDCDA6C48B49C287F7C2C25712 5433665D388AB57E9A03E8709B931F03 83E1C53FBC3EEBA518594BDD6B24F1EB FC0AEC90836A0A49ACF47B3DEF965D2D

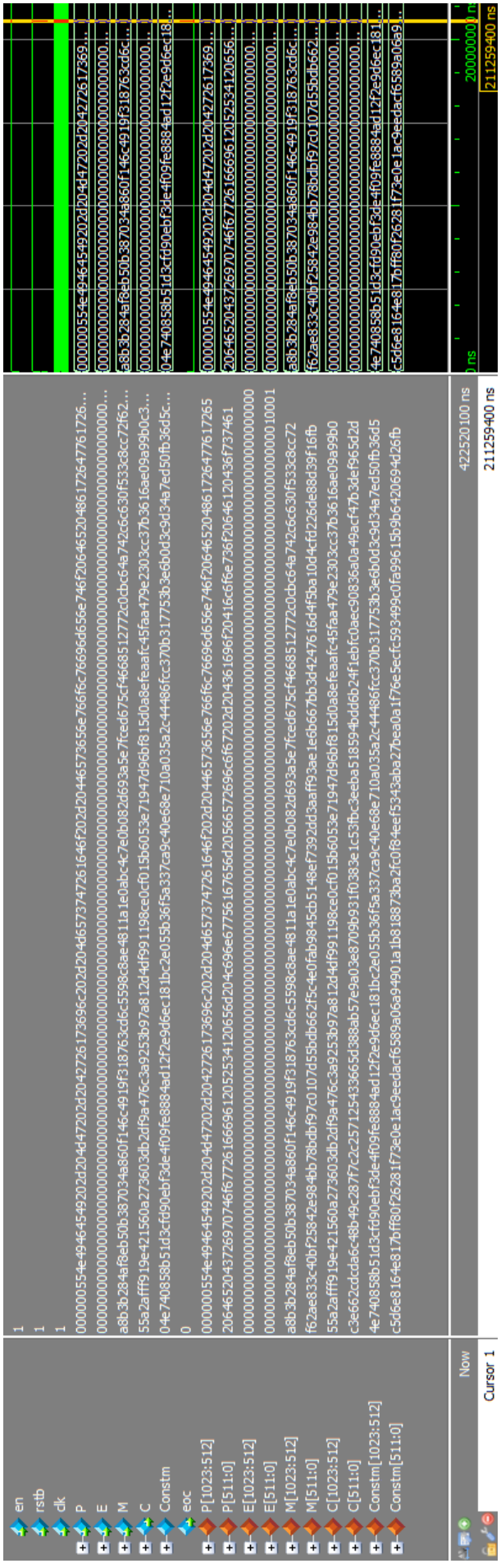

Figura 43: Simulação processo de cifragem para arquitetura de 1024 bits.

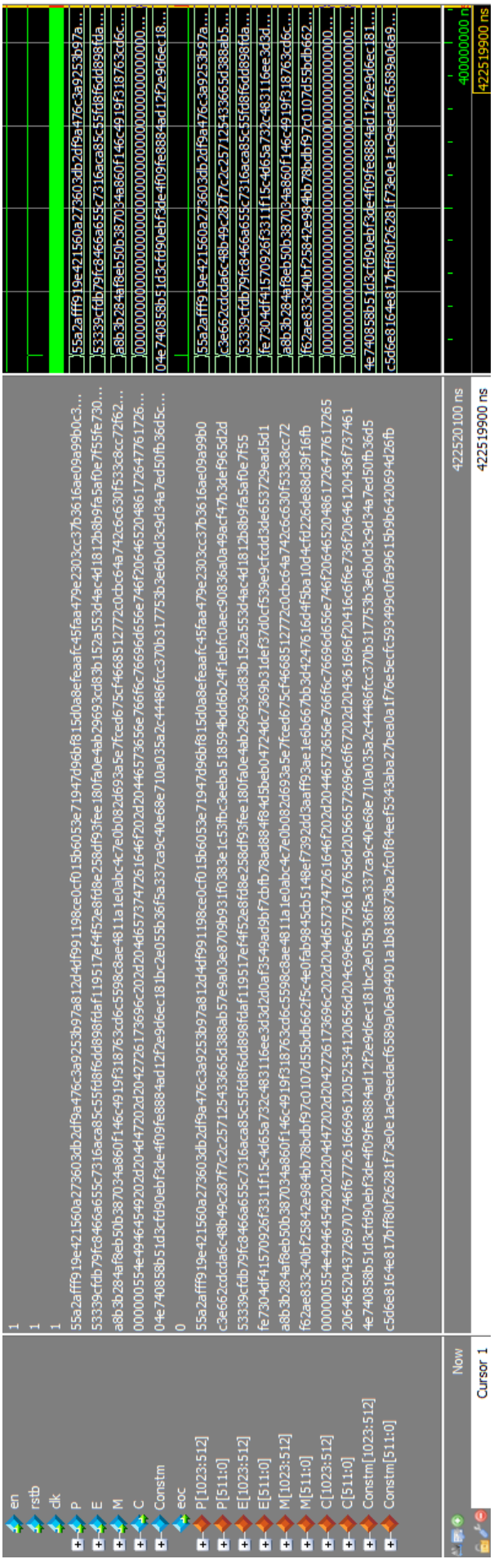

Figura 44: Simulação processo de decifragem para arquitetura de 1024 bits.

3.8 Adendos ao coprocessador de 1024 bits

O coprocessador de 1024 bits apresentado possui uma grande quantidade de pinos de interface. Os sinais de entrada do bloco RSA Topo P, E, M e $Const_m$ e o sinal de saída C são de 1024 bits, totalizando 5120 pinos. Devido a inviabilidade de se fabricar um circuito integrado com a capacidade de suportar a grande quantidade de pinos, houve a necessidade de se projetar uma interface de comunicação para escrita e leitura dos dados dentro do circuito integrado.

A interface serial de comunicação SPI é um padrão desenvolvido da Motorola que opera no modo *full duplex*. Esta interface é utilizada para distancias curtas, onde um dispositivo mestre consegue controlar a comunicação com um ou mais escravos. A interface utiliza de quatro sinais para a comunicação, dos quais dois são para a transmissão dos dados em cada sentido, um sinal de *clock* para sincronismo e um para a seleção do dispositivo para a comunicação. Em ambos os circuitos integrados utilizados na comunicação, um dado de tamanho fixo é serializado a cada pulso de *clock* que o dispositivo mestre enviar, se assemelhando ao comportamento de um *buffer* circular. O projeto da interface SPI para coprocessador de 1024 bit foi desenvolvido para operar no modo escravo.

Além da interface serial, um controlador para manipular os dados enviados e recebidos pela interface também foi desenvolvido. Os valores dos sinais P, E, M e $Constm$ são recebidos via a interface serial e armazenados em bancos de registradores para serem utilizados no coprocessador de 1024 bits. Ao término do processo de cifragem, o sinal C, isto é, o dado cifrado, é enviado para o dispositivo mestre através da interface serial.

Uma máquina de estados foi projetada para controlar a operação de início da cifragem. Através dos comandos enviados via interface serial, é possível iniciar um processo de cifragem ou decifragem. Além dos comandos enviados via interface serial, um pino *start* também pode ser utilizado para iniciar o processo de cifragem. O sinal *eoc* é utilizado para informar que a cifragem foi concluída e o texto cifrado pode ser lido via interface serial.

Dessa forma, o diagrama de blocos final para o coprocessador de 1024 bits é ilustrado na Figura 45. O bloco *rsa* consiste da arquitetura RSA desenvolvida para 1024 bits, o bloco *spi peripheral* realiza o controle da interface serial e armazena os valores dos sinais P, E, M, $Constm$ e C em bancos de registradores e o bloco *rsa logic* controla o início do processo de cifragem e decifragem realizado no bloco *rsa*. A partir desta estrutura final, iniciou-se o desenvolvimento do *layout* do circuito integrado.

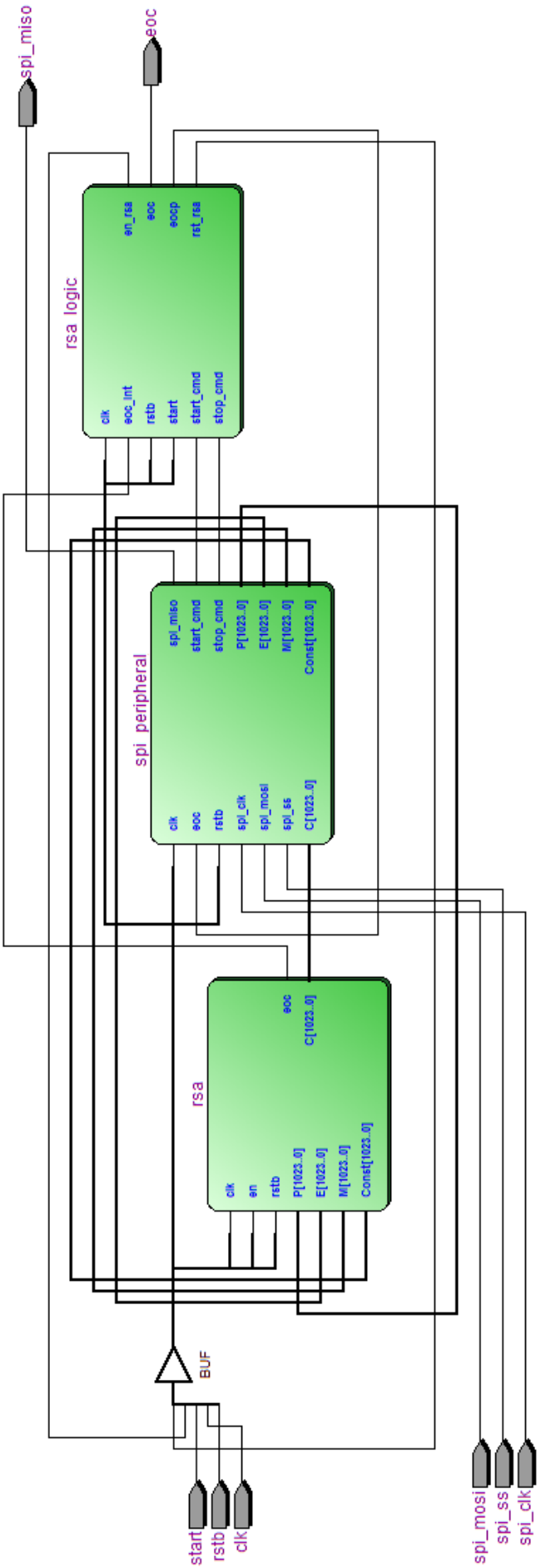

Figura 45: Blocos coprocessador RSA de 1024 bits.

4 Resultados

4.1 Tecnologia

A escolha da tecnologia foi realizada através do convênio da UNIFEI com a *foundry* MOSIS, que fornece *design kits* para estudo e desenvolvimento de circuitos integrados para os alunos de graduação e pós graduação. Dentre as tecnologias oferecidas pela MOSIS, escolheu-se a tecnologia com menor comprimento de canal e que apresentava uma biblioteca de *stardard cells* disponível para o desenvolvimento do *layout* do circuito integrado.

O circuito integrado foi concebido na tecnologia IBM7SF. O *design kit* da tecnologia foi disponibilizado pela MOSIS para instalação dos arquivos de tecnologia nas ferramentas Cadence. A tecnologia IBM7SF utiliza um processo CMOS de 180nm. A tensão de alimentação dos transistores é de 1,8V, com opções de *pads* com alimentação de 2,5V e de 3,3V. Os transistores padrões NFET e PFET apresentam um L_{min} de 0,18μm. O L_{eff} dos transistores NFET é de 0,11μm enquanto o L_{eff} dos transistores PFET é de 0,14μm. As tensões de *threshold* desta tecnologia são de 0,43V para os transistores NFET e -0,38V para os transistores PFET. As correntes de saturação da tecnologia IBM7SF são de 600mA para os transistores NFET e 260mA para os transistores PFET. A camada de óxido de silício da tecnologia IBM7SF tem uma altura de 3,5nm. Ao todo são 6 camadas de metais para roteamento, sendo os metais utilizados cobre e alumínio.

O manual *standard cell* da tecnologia IBM7SF apresenta o conjunto de famílias de células digitais implementadas no processo de 0,18μm. As células possuem uma altura padrão de 6,72μm e comprimento variável. As tensões de alimentação das células são 1,8V com tolerância de $\pm$ 10% e operam entre -55ºC a 100ºC.

As células são dividas em cinco grupos: lógica primitiva, lógica complexa, lógica unária, lógica sequencial e células físicas. As células de lógica primitiva consistem de portas lógicas AND, NAND, OR, NOR, XOR e XNOR de duas a quatro entradas, e do inversor. As células de lógica complexa consistem em agrupamentos das células de lógica primitvas. As células de lógica unária são constituídas de *buffers*, *drivers* de *clock*, comparadores e multiplexadores. As células de lógica sequencial são constituídas de *latches* e *flip flops* do tipo D. As células físicas são células para preenchimento de espaços vazios, terminação de sinais e entradas de portas lógicas não utilizadas.

A biblioteca de *standard cells* possui células com variados tipos de desempenho. Há células com alto desempenho para cargas de baixo valor capacitivo e células com alto desempenho para cargas com alto valor capacitivo. A ferramenta de roteamento

da Cadence seleciona na biblioteca a célula com desempenho mais apropriada para ser utilizada durante o desenvolvimento do *layout*.

4.2 Resultados de desempenho

O desempenho de cada bloco do circuito integrado é compilado na Tabela 30. As informações foram extraídas dos relatórios de síntese física do software Cadence Encounter RTL Compiler utilizando a biblioteca de *standard cells* da tecnologia IBM7SF.

Tabela 30: Resumo síntese física de cada bloco do coprocessador.

Nome do Bloco	**Freq. máx. de operação(MHz)**	**Área** (μm^2)	**Potência** (μW)
RSA Topo	125,031	2723345,050	797941,856
Multiplexador	979,432	46991,078	7720,888
Multiplexador Modificado	979,432	46991,078	7720,888
Máquina de Estados Finitos	381,388	129834,163	17811,701
Registrador Criptografia	607,165	103040,181	15359,796
Multiplicador Modular de Montgomery	149,053	1151253,197	366943,403
Somador Kogge-Stone	239,464	306261,996	129842,915
Elemento Processador Borda Direita	2702,703	94,080	29,810
Elemento Processador	4854,369	90,317	22,358
Registrador Serializador	1841,621	123527,040	15329,987
Registrador Soma	693,962	96958,848	15322,535
Registrador Produto Modular Montgomery	539,665	129236,391	23080,686

O bloco com menor resposta em frequência, excluindo os blocos Multiplicador Modular de Montgomery e RSA Topo que possuem somente conexões de hierarquia, é o bloco Somador Kogge-Stone. Como o bloco Somador Kogge-Stone é utilizado duas vezes dentro de um bloco Multiplicador Modular de Montgomery e são dispostos de forma sequencial com lógica simples, o atraso produzido pelo Somador Kogge-Stone é o principal fator limitante da frequência máxima da arquitetura projetada.

A frequência máxima de operação do *clock* é de 125,03MHz para o bloco RSA Topo. O circuito integrado é capaz de cifrar ou decifrar um bloco de dados de 1024 bits em aproximadamente 8,44ms. A taxa de bits produzidos por segundo desta implementação é de 121,269Kbps. A área total das células do no bloco RSA Topo é de aproximadamente 2,723mm^2. A quantidade de células NAND de duas portas equivalente é de 107k e o consumo de energia estimado é de 797,942mW.

Durante o desenvolvimento do *layout* optou-se pela geometria retangular com a altura metade do valor do comprimento. Ao todo foram utilizadas oito células de *pad* para sinais de topo do circuito integrado, das quais seis são *pads* de entrada e duas são *pads* de saída. As células de *pads* de entrada são: uma célula para o sinal *clock*, uma célula para

o sinal *reset*, uma célula para o sinal *start* e três células para os sinais da interface serial SPI, ss, sclk e mosi. As células de *pads* de saída são: uma célula para o sinal *eoc*, uma célula para o sinal de interface serial SPI miso.

Além das células *pads* dos sinais foram utilizados oito células *pads* para alimentação VCC e oito células *pads* para alimentação GND. As células *pads* dos quatro *corners* foram utilizadas para fechar a geometria retangular e mais 200 células *pads fillers* foram utilizadas para fechar o *pad ring* do circuito integrado.

A Figura 46 ilustra o *layout* final do circuito integrado. O comprimento do circuito integrado é de 4,714 mm e a altura é de 3,034 mm, área de 14,302276 mm^2.

4.3 Análise de desempenho

A Tabela 31 faz uma comparação deste trabalho, denominado Costa, com demais publicações. É importante ressaltar que todas as informações da Tabela 31 são relativas a projetos de criptografia RSA de chave com tamanho igual a 1024 bits que utilizam o mesmo principio da arquitetura de multiplicação modular de Montgomery.

Tabela 31: Comparação de desempenho com outros trabalhos.

Ref.	**Tecnologia**	**Base**	**Fq. máx. (MHz)**	**Tempo exp. mod.(ms)**	***Throughput* (Kbps)**	**Pot. (mW)**	**Escala (kNAND)**
[16]	Stratix III	2	380,66	11,06	92,59	—	—
[27]	CMOS 0,5μm	2	80,00	45,00[1]	22,75[1]	—	—
[17]	CMOS 90nm	2	1087,10	1,93	530,57[1]	—	776
[18]	Virtex-6	17	410,69	11,26	88,78	—	—
[19]	CMOS 90nm	32	471,70	7,27	140,85[1]	—	—
[20]	CMOS 0,18μm	32	140,00	—	6350,00	1600,00	923
Costa	CMOS 0,18μm	2	125,03	8,44	121,27	797,94	107

[1]Estimativa do autor baseado nos trabalhos originais

O trabalho que apresenta o melhor desempenho em tempo encontrado na literatura é o de Wang [17]. A tecnologia e o caminho crítico deste trabalho fazem com que a frequência do clock seja muito elevada. Entretanto, a área para a arquitetura proposta é da ordem de 776k portas lógicas NAND equivalentes, que representa uma área 7,25 vezes maior que a proposta neste trabalho. O trabalho de Zhao [20] apresenta uma arquitetura com *pipeline* que aumenta consideravelmente o *throughput* do projeto. Apesar disso, apresenta uma dissipação de potência de aproximadamente 1,6W, que representa o dobro da potência da arquitetura desenvolvida neste trabalho. Além disso, a área proposta de Zhao [20] é da ordem de 923k portas lógicas NAND equivalentes, que representa uma área 8,62 vezes maior que a proposta neste trabalho. O trabalho de Miyamoto [19] apresenta uma grande quantidade de arquitetura de somadores e algoritmos com diferentes bases para a multiplicação modular de Montgomery. Comparado com este trabalho, a arquitetura proposta em tecnologia CMOS 0,18μm calcula uma operação de exponenciação modu-

lar, com menos ciclos de clock, com quase o mesmo tempo que a arquitetura balanceada apresentada em tecnologia CMOS 90nm.

O trabalho proposto por Renteria [16] apresenta uma solução baseada em uma arquitetura sistólica onde os elementos processadores estão próximos um dos outros. Mesmo que a frequência de clock desta solução seja maior que a apresentada neste trabalho, um maior *throughput* foi alcançado. O trabalho de Song [18] apresenta uma arquitetura com base 17 usando um único bloco DES48E, 201 registradores, 374 *look-up tables* e 36Kbits de memória BRAM. O tempo necessário para calcular uma exponenciação modular nesta arquitetura é de 11,26ms.

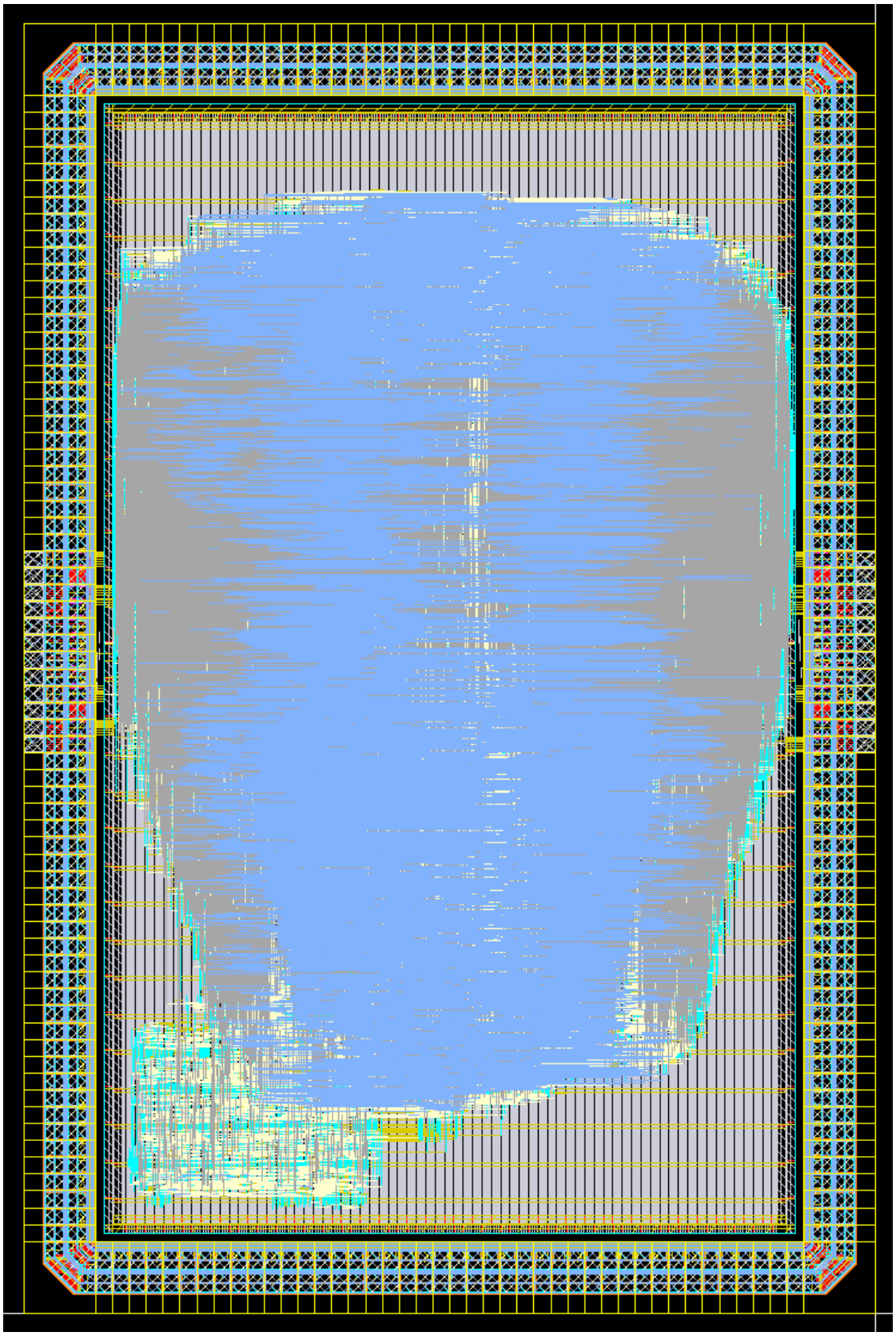

Figura 46: *Layout* coprocessador RSA 1024 bits.

5 Conclusão

Neste trabalho foi desenvolvido um circuito para criptografia assimétrica RSA de 1024 bits na linguagem Verilog e prototipado em ferramenta de projetos de circuitos integrados. Na primeira etapa foi estudado o algoritmo de criptografia RSA e implementações em software do algoritmo. Em seguida, foram desenvolvidas softwares em linguagem C e Java dos algoritmos estudados para realizar a criptografia RSA. O software em linguagem C foi desenvolvido para criptografia de até 16 bits, enquanto o software em linguagem Java foi desenvolvido para criptografia de até 1024 bits. Ambos os softwares foram testados com conjuntos de dados e apresentaram resultados semelhantes para os processos de cifragem e decifragem.

Na segunda etapa, implementações em hardware dos algoritmos foram estudados e reproduzidas, o que permitiu uma compreensão mais detalhada dos algoritmos em software em circuitos de hardware para criptografia. Na terceira etapa foi desenvolvido um protótipo para criptografia de quatro bits, conforme apresentado neste trabalho. A arquitetura proposta foi projetada a partir de modificações dos circuitos estudados para obter uma melhor resposta em frequência. Todos os blocos da arquitetura foram apresentados e suas interconexões apresentadas. Ao final da apresentação da arquitetura de quatro bits, um exemplo foi utilizado para comprovar o funcionamento conforme o algoritmo de criptografia assimétrica RSA. A arquitetura foi testada através de simulações em ferramentas e prototipada em FPGA para validação.

Posteriomente foi desenvolvido a implementação da arquitetura para criptografia de 1024 bits, conforme apresentada neste trabalho. Para contornar a quantidade de pinos necessárias da arquitetura, uma interface serial de comunicação e um controlador de interface foram projetados. A arquitetura para 1024 foi testada através de simulações em ferramentas e não foi prototipada em FPGA. A prototipação em FPGA não foi possível devido a alta quantidade de recursos necessários do projeto. Ao final da apresentação da arquitetura de 1024 bits, uma simulação foi apresentado para comprovar o funcionamento conforme o algoritmo de criptografia assimétrica RSA. Uma vez validada a arquitetura de 1024 bits, o fluxo digital foi percorrido nas ferramentas Cadence para gerar o *layout* final do circuito integrado.

Este trabalho teve como principal objetivo o projeto de um circuito para criptografia assimétrica RSA desenvolvido em linguagem Verilog. Esse desenvolvimento representa um ponto de partida para futuros desenvolvimentos na área de projetos de circuitos integrados de criptografia assimétrica do Grupo de Microeletrônica da UNIFEI. Apesar do tempo limitado para defesa deste trabalho, algumas figuras de mérito foram alcançadas

no projeto e publicadas em congressos internacionais, conforme apresentadas nos anexos desta dissertação.

Como sugestão para trabalhos futuros pode-se considerar a substituição do somador Kogge-Stone por outros tipos de somadores e realizar comparações de resposta em frequência, área e potência entre as arquiteturas. Sugere-se também o estudo de representação binária redundante, como utilizá-la na implementação de somadores de arquitetura *carry save* e os resultados que o circuito integrado projetado apresentaria caso esta representação fosse utilizada.

Sugere-se também a implementação de um circuito para o cálculo da constante de Montgomery, explorar novas formas de implementação do algoritmo de multiplicação modular de Montgomery e do algoritmo de exponenciação modular. Sugere-se também a aumentar a base de operação da multiplicação modular de Montgomery e da exponenciação modular, ou seja, utilizar mais de um bits no processamento do multiplicação modular de Montgomery e na exponenciação modular, com finalidade de atingir uma resposta em frequência melhor do que a apresentada neste trabalho.

Sugere-se também o desenvolvimento de um coprocessador RSA totalmente modular, onde a quantidade de bits é definida somente durante a síntese e os códigos Verilog sejam genéricos. Pode-se ainda prototipar o projeto em outras tecnologias CMOS com menor comprimento de canal dos transistores e desenvolver blocos analógicos para integrar o *layout* do circuito integrado tais como oscilador e *power on reset*.

Referências

1 TRAPPE, W.; WASHINGTON, L. C. *Introduction to Cryptography with Coding Theory*. [S.l.]: Prentice Hall, 2001. Citado 4 vezes nas páginas 15, 18, 19 e 29.

2 C. DRYSDALE R. L., B. K. S. *Matemática discreta para ciências da computação*. São Paulo, Brasil: Pearson, 2008. ISBN 978-85-8143-769-9. Citado na página 15.

3 CAMPOS, A. A. N. *Algoritmo de criptografia AES em hardware, utilizando Dispositivos de Lógica Programável(FPGA) e Linguagem de Descrição de Hardware(VHDL)*. Dissertação (Mestrado) — Universidade Federal de Itajubá - UNIFEI, 2008. Citado na página 15.

4 SAAD, M. W. *Implementação do algoritmo AES em hardware reconfigurável - FPGA*. Dissertação (Mestrado) — Universidade Federal de Itajubá - UNIFEI, 2010. Citado na página 15.

5 GOMES, O. de S. M. *Desenvolvimento de hardware configurável de criptografia simétrica utilizando FPGA e linguagem VHDL*. Dissertação (Mestrado) — Universidade Federal de Itajubá - UNIFEI, 2010. Citado na página 16.

6 RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, ACM, New York, NY, USA, v. 26, n. 1, p. 96–99, jan. 1983. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10-.1145/357980.358017>. Citado na página 16.

7 STALLING, W. *Criptografia e segurança de redes*. São Paulo, Brasil: Pearson Prentice Hall, 2008. ISBN 978-85-7605-119-0. Citado 11 vezes nas páginas 17, 18, 19, 20, 21, 22, 26, 27, 28, 29 e 30.

8 D. PEREIRA F. D., C. R. B. M. E. *Criptografia em Software e Hardware*. [S.l.]: Novatec, 2005. Citado 2 vezes nas páginas 17 e 19.

9 GOMES, O. S. M. *Desenvolvimento de Hardware Configurável de Criptografia Simétrica utilizando FPGA e linguagem VHDL*. Dissertação (Mestrado) — Universidade Federal de Itajubá, 2011. Citado na página 17.

10 PALNITKAR, S. *Verilog hdl: a guide to digital design and synthesis, second edition*. Second. Upper Saddle River, NJ, USA: Prentice Hall Press, 2003. ISBN 0-13-044911-3. Citado 4 vezes nas páginas 22, 23, 24 e 25.

11 RSA-LABORATORIES. *http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf*. 2012. Disponível em: <http://www.emc-.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard-.pdf>. Citado 2 vezes nas páginas 26 e 89.

12 RSA-LABORATORIES. *4.1.2.1 WHAT KEY SIZE SHOULD BE USED?* 2013. Disponível em: <http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/key-size-.htm>. Citado 2 vezes nas páginas 30 e 42.

13 DALY, A.; MARNANE, W. Efficient architectures for implementing montgomery modular multiplication and rsa modular exponentiation on reconfigurable logic. In: *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2002. (FPGA '02), p. 40–49. ISBN 1-58113-452-5. Disponível em: <http://doi.acm.org/10.1145/503048.503055>. Citado 5 vezes nas páginas 30, 31, 33, 34 e 35.

14 MONTGOMERY, P. L. Modular multiplication without trial division. *Mathematics of Computation*, v. 44, n. 170, p. 519–521, 1985. Citado na página 30.

15 NEDJAH, N. et al. Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems. *Computers Digital Techniques, IET*, v. 6, n. 5, p. 290–301, 2012. ISSN 1751-8601. Citado 9 vezes nas páginas 30, 31, 32, 33, 40, 41, 42, 43 e 44.

16 RENTERIA-MEJIA, C.; TRUJILLO-OLAYA, V.; VELASCO-MEDINA, J. Design of an 8192-bit rsa cryptoprocessor based on systolic architecture. In: *Programmable Logic (SPL), 2012 VIII Southern Conference on*. [S.l.: s.n.], 2012. p. 1–6. Citado 3 vezes nas páginas 33, 96 e 97.

17 WANG, Y.; MASKELL, D. L.; LEIWO, J. A unified architecture for a public key cryptographic coprocessor. *J. Syst. Archit.*, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 10, p. 1004–1016, out. 2008. ISSN 1383-7621. Disponível em: <http://dx.doi.org/10.1016/j.sysarc.2008.04.013>. Citado 2 vezes nas páginas 33 e 96.

18 SONG, B.; ITO, Y.; NAKANO, K. Crt-based dsp decryption using montgomery modular multiplication on the fpga. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. [S.l.: s.n.], 2011. p. 532–541. ISSN 1530-2075. Citado 3 vezes nas páginas 33, 96 e 97.

19 MIYAMOTO, A. et al. Systematic design of rsa processors based on high-radix montgomery multipliers. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 19, n. 7, p. 1136–1146, 2011. ISSN 1063-8210. Citado 2 vezes nas páginas 33 e 96.

20 ZHAO, X. et al. A 6.35mbps 1024-bit rsa crypto coprocessor in a 0.18um cmos technology. In: *Very Large Scale Integration, 2006 IFIP International Conference on*. [S.l.: s.n.], 2006. p. 216–221. Citado 2 vezes nas páginas 33 e 96.

21 PARHAMI, B. *Computer arithmetic: algorithms and hardware designs*. Oxford, UK: Oxford University Press, 2000. ISBN 0-19-512583-5. Citado 3 vezes nas páginas 43, 44 e 46.

22 PATTERSON, D. A.; HENNESSY, J. L. *Computer organization & design: the hardware/software interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN 1-55860-281-X. Citado na página 46.

23 NEHRU, K.; SHANMUGAM, A.; VADIVEL, S. Design of 64-bit low power parallel prefix vlsi adder for high speed arithmetic circuits. In: *Computing, Communication and Applications (ICCCA), 2012 International Conference on*. [S.l.: s.n.], 2012. p. 1–4. Citado 2 vezes nas páginas 46 e 47.

24 LADNER, R. E.; FISCHER, M. J. Parallel prefix computation. *J. ACM*, ACM, New York, NY, USA, v. 27, n. 4, p. 831–838, out. 1980. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/322217.322232>. Citado na página 46.

25 MOUDALLAL, Z. et al. A low-power methodology for configurable wide kogge-stone adders. In: *Energy Aware Computing (ICEAC), 2011 International Conference on*. [S.l.: s.n.], 2011. p. 1–5. Citado 4 vezes nas páginas 46, 47, 48 e 49.

26 RSA-LABORATORIES. *Test vectors for RSA PKCS1 v1.5 Encryption*. 2014. Disponível em: <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1-vec.zip>. Citado na página 87.

27 TENCA, A.; KOC, C. A scalable architecture for modular multiplication based on montgomery's algorithm. *Computers, IEEE Transactions on*, v. 52, n. 9, p. 1215–1221, 2003. ISSN 0018-9340. Citado na página 96.

Apêndices

APÊNDICE A – Software em Linguagem C da Multiplicação Modular de Montgomery

Software em linguagem C para realizar a operação de Multiplicação Modular de Montgomery para as entradas A, B e M. Este software em linguagem C possui limitação para variáveis de 16 bits. Esta limitação está associada aos tipos de variáveis utilizados para o desenvolvimento do mesmo.

```
1  /*************************************************************
2  Prototipo  : unsigned short MonPro (unsigned short A, unsigned short B,
       unsigned short M, unsigned short qtdd_bits)
3  Argumentos : unsigned short A - palavra Multiplicando
4               unsigned short B - palavra Multiplicador
5               unsigned short M - palavra Modulo
6               unsigned short qtdd_bits - quantidade de bits das palavras
7  Retorno    : unsigned short R - Produto da Multiplicacao Modular Montgomery
       entre A, B e M
8  Funcao     : Realiza operacao de Multiplicacao Modular Montgomery entre A,
       B e M
9  *************************************************************/
10 unsigned short MonPro (unsigned short A, unsigned short B, unsigned short M
       , unsigned short qtdd_bits)
11 {
12      // Valor do bit 0 do Resultado (LSBit do R)
13      unsigned short ro;
14
15      // Valor do Resultado
16      unsigned short R;
17
18      // Valor do bit 0 do Resultado (LSBit do B)
19      unsigned short bo;
20
21      // Valor do bit 0 do quociente
22      unsigned short qo;
23
24      // Valor do bit i da palavra A
25      unsigned short bit_a;
26
27      // Contador
28      unsigned short i;
```

```
29
30      // Inicializacao do Valor do Resultado
31      R = 0;
32
33      // Loop para realizar a Multiplicacao Modular Montgomery bit a bit
34      for(i=0; i<qtdd_bits; i++)
35      {
36              // Verifica o LSBit do Resultado
37              ro = ((R & 0x0001) ? 0x0001 : 0x0000);
38              // Verifica o LSBit de B
39              bo = ((B & 0x0001) ? 0x0001 : 0x0000);
40              // Verifica o bit de A
41              bit_a = bit_indice(A, i);
42              // Faz o calculo do quociente
43              qo =  ro + bo * bit_a;
44              qo =  (qo & 0x0001) ? 0x0001 : 0x0000;
45              // Faz a conta completa do ciclo de Multiplicacao Modular
                    Montgomery
46              R = (R + bit_a*B + qo*M)/2;
47      }
48
49      // Retorna o Produto Modular Montgomery
50      return R;
51  }
```

APÊNDICE B – Software em Linguagem C da Exponenciação Modular

Software em linguagem C para realizar a operação de Exponenciação Modular para as entradas P, E e M. Este software em linguagem C possui limitação para variáveis de 16 bits. Esta limitação está associada aos tipos de variáveis utilizados para o desenvolvimento do mesmo.

```
1  /*************************************************************
2  Prototipo  : unsigned short MonExp (unsigned short P, unsigned short E,
       unsigned short M, unsigned short qtdd_bits)
3  Argumentos : unsigned short P - palavra Texto Puro
4               unsigned short E - palavra Expoente
5               unsigned short M - palavra Modulo
6               unsigned short qtdd_bits - quantidade de bits das palavras
7  Retorno    : unsigned short R - Resultado da Exponenciacao Modular entre P,
       E e M
8  Funcao     : Realiza operacao de Exponenciacao Modular entre P, E e M
9  *************************************************************/
10
11 unsigned short MonExp (unsigned short P, unsigned short E, unsigned short M
       , unsigned short qtdd_bits)
12 {
13      // Valor da Constante Pre-Calculada
14      unsigned int C;
15
16      // Valor da Constante Pre-Calculada Montgomery
17      unsigned short Cm;
18
19      // Valor do Texto Puro Montgomery
20      unsigned short Pm;
21
22      // Valor do Resultado da Exponenciacao
23      unsigned short R;
24
25      // Valor do bit i da palavra E
26      unsigned short bit_e;
27
28      // Contador de proposito geral
29      unsigned short i;
30
31      // Mapeamento da Constante
32      C = pow(2, (2*qtdd_bits));
```

```
33      C = C % M;
34      Cm = (unsigned short) C;
35
36      // Mapeamento do Texto Pleno
37      Pm = MonPro(Cm, P, M, qtdd_bits);
38
39      // Mapeamento do Resultado
40      R = MonPro(Cm, 1, M, qtdd_bits);
41
42      // Loop para realizar a Exponenciacao Modular Montgomery
43      for(i=0; i<qtdd_bits-1; i++)
44      {
45              bit_e = bit_indice(E, i);
46              if(bit_e == 1)
47              {
48                      R = MonPro(R, Pm, M, qtdd_bits);
49              }
50              Pm = MonPro(Pm, Pm, M, qtdd_bits);
51      }
52
53      // Remapeamento do Resultado
54      R = MonPro(1, R, M, qtdd_bits);
55
56      // Retorna o Produto Modular Montgomery
57      return R;
58  }
```

APÊNDICE C – Software em Linguagem Java da Multiplicação Modular de Montgomery

Software em linguagem Java para realizar a operação de Multiplicação Modular de Montgomery para as entradas A, B e M. Este software em linguagem Java, diferentemente do apresentado em C, não possui limitação de tamanho para as variáveis. Entretanto, os testes foram realizados para variáveis de até 1024 bits.

```
1  import java.math.BigInteger;
2  /*************************************************************
3  Prototipo  : public static BigInteger MonPro(BigInteger A, BigInteger B,
       BigInteger M, BigInteger qtdd_bits)
4  Argumentos : BigInteger A - palavra Multiplicando
5               BigInteger B - palavra Multiplicador
6               BigInteger M - palavra Modulo
7               BigInteger qtdd_bits - quantidade de bits das palavras
8  Retorno    : BigInteger R - Produto da Multiplicacao Modular Montgomery
       entre A, B e M
9  Funcao     : Realiza operacao de Multiplicacao Modular Montgomery entre A,
       B e M
10 *************************************************************/
11 public static BigInteger MonPro(BigInteger A, BigInteger B, BigInteger M,
       BigInteger qtdd_bits) {
12         BigInteger ro;
13         BigInteger R;
14         BigInteger bo;
15         BigInteger qo;
16         Boolean bit_a;
17         BigInteger i;
18
19         R = BigInteger.valueOf(0);
20
21         for(i = BigInteger.valueOf(0); i.longValue() < qtdd_bits.
               longValue(); i = i.add(BigInteger.valueOf(1)))
22         {
23             ro = R.mod(BigInteger.valueOf(2));
24             bo = B.mod(BigInteger.valueOf(2));
25             bit_a = A.testBit(i.intValue());
26             if(bit_a == true) {
27                 qo = (bo.multiply(BigInteger.valueOf(1))).add(ro);
```

```
28                  } else {
29                      qo = (bo.multiply(BigInteger.valueOf(0))).add(ro);
30                  }
31                  qo = qo.mod(BigInteger.valueOf(2));
32                  if(bit_a == true) {
33                      R = ((R.add(BigInteger.valueOf(1).multiply(B))).add(
34                      qo.multiply(M))).divide(BigInteger.valueOf(2));
35                  } else {
36                      R = ((R.add(BigInteger.valueOf(0).multiply(B))).add(
37                      qo.multiply(M))).divide(BigInteger.valueOf(2));
38                  }
39              }
40              return R;
41      }
```

APÊNDICE D – Software em Linguagem Java da Exponenciação Modular

Software em linguagem Java para realizar a operação de Exponenciação Modular para as entradas P, E e M. Este software em linguagem Java, diferentemente do apresentado em C, não possui limitação de tamanho para as variáveis. Entretanto, os testes foram realizados para variáveis de até 1024 bits.

```
1  import java.math.BigInteger;
2  /*************************************************************
3  Prototipo  : public static BigInteger MonExp(BigInteger P, BigInteger E,
       BigInteger M, BigInteger qtdd_bits)
4  Argumentos : BigInteger P - palavra Texto Puro
5               BigInteger E - palavra Expoente
6               BigInteger M - palavra Modulo
7               BigInteger qtdd_bits - quantidade de bits das palavras
8  Retorno    : BigInteger R - Resultado da Exponenciacao Modular entre P, E e
       M
9  Funcao     : Realiza operacao de Exponenciacao Modular entre P, E e M
10 *************************************************************/
11 public static BigInteger MonExp(BigInteger P, BigInteger E, BigInteger M,
       BigInteger qtdd_bits) {
12          BigInteger Cm;
13          BigInteger Pm;
14          BigInteger R;
15          Boolean bit_e;
16          BigInteger i;
17
18          Cm = new BigInteger ("2", 10);
19          Cm = Cm.modPow((BigInteger.valueOf(2)).multiply(qtdd_bits) , M);
20
21          Pm = MonPro(Cm, P, M, qtdd_bits);
22          R = MonPro(Cm, one, M, qtdd_bits);
23
24          for(i = BigInteger.valueOf(0); i.longValue() < (qtdd_bits.
                longValue()-1); i = i.add(one))
25          {
26               bit_e = E.testBit(i.intValue());
27               if(bit_e == true) {
28                    R = MonPro(R, Pm, M, qtdd_bits);
```

```
29                  }
30                  Pm = MonPro(Pm, Pm, M, qtdd_bits);
31              }
32
33              R = MonPro(one, R, M, qtdd_bits);
34
35              return R;
36      }
```

APÊNDICE E – Fluxograma da Máquina de Estados Finitos e Tabela de Valor dos Sinais de Saída

O fluxograma da Máquina de Estados Finitos do coprocessador RSA de 4 bits é ilustrado ilustrado na Figura 47. Para o coprocessador de 4 bits, são ao todos 56 estados necessários para completar um processo de cifragem ou decifrgem. Pode-se verificar no fluxograma que trata-de de um contador simples onde os sinais de saída do bloco Máquina de Estados Finitos são definidos de acordo com o seu presente do contador.

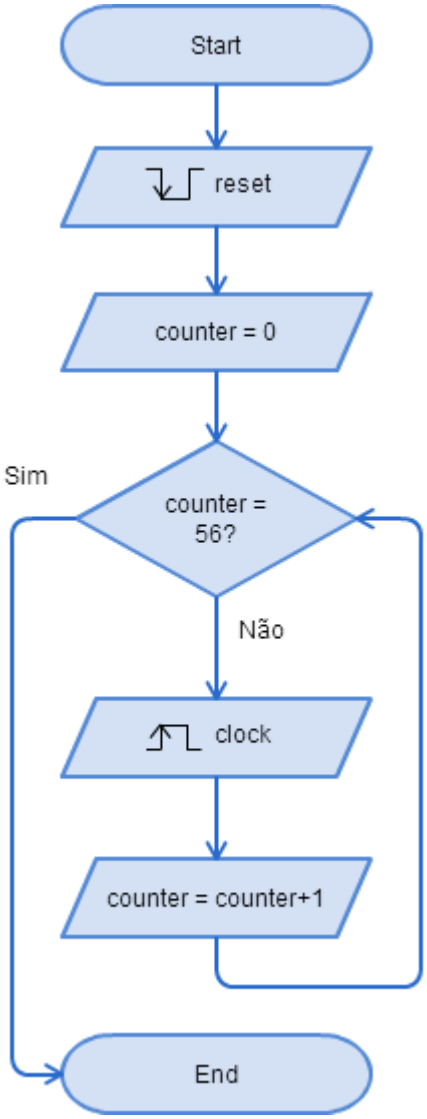

Figura 47: Fluxograma da Máquina de Estados Finitos do coprocessador RSA de quatro bits.

A Tabela 32 apresenta a correspondência do valor dos sinais de saída do bloco Máquina de Estados Finitos em cada um dos estados possíveis.

Como os valores dos sinais de saída do bloco Máquina de Estados Finitos apresentam valores semelhantes para um mesmo conjunto valores do contador, não há necessidade de definir um estado para cada valor do contador. Dessa forma, o código deste bloco pode ser desenvolvido utilizando as faixas de valores do contador, evitando a descrição manual de todos os estados necessários.

Tabela 32: Tabela dos valores dos sinais de saída do bloco Máquina de Estados Finitos.

Counter - Sinal	**lda**	**ldr**	**lock1**	**lock2**	**sel1**	**sel2**	**eoc**
0	1'b1	1'b0	1'b1	1'b1	2'b00	1'b0	1'b0
1 a 6	1'b0	1'b0	1'b1	1'b1	2'b00	1'b0	1'b0
7	1'b0	1'b1	1'b1	1'b1	2'b00	1'b0	1'b0
8, 16, 24, 32, 40	1'b1	1'b0	E[0], E[1], E[2], E[3], E[4]	1'b1	2'b01	1'b1	1'b0
9 a 14, 17 a 22, 25 a 30, 33 a 38, 41 a 46	1'b0	1'b0	E[0], E[1], E[2], E[3], E[4]	1'b1	2'b01	1'b1	1'b0
15, 23, 31, 39, 47	1'b0	1'b1	E[0], E[1], E[2], E[3], E[4]	1'b1	2'b01	1'b1	1'b0
48	1'b1	1'b0	1'b1	1'b0	2'b10	1'b1	1'b0
49 a 54	1'b0	1'b0	1'b1	1'b0	2'b10	1'b1	1'b0
55	1'b0	1'b1	1'b1	1'b0	2'b10	1'b1	1'b0
56	1'b0	1'b1	1'b1	1'b0	2'b10	1'b1	1'b1

Anexos

A 1024 bit RSA Coprocessor in CMOS

Caio A. da Costa, Robson L. Moreno, Otávio S. A. Carpinteiro, Tales C. Pimenta
Department of Microelectronics
Universidade Federal de Itajubá - UNIFEI
Itajubá, MG, Brazil
caioalonso@gmail.com, moreno@unifei.edu.br, otavio.carpinteiro@gmail.com, tales@unifei.edu.br

***Abstract*—This paper presents the architecture and model of a modular exponentiation hardware for RSA public key cryptography algorithm. A radix-2 Montgomery modular multiplication hardware based on a systolic implementation was designed. A kogge-stone adder was designed to reduce the critical path and improve throughput. The data path and dataflow of the Montgomery modular multiplier and the exponentiation hardware is fully exploited. Cadence© Encounter RTL Compiler was used to synthesize the RTL code described in Verilog HDL. The coprocessor was implemented with standard cells library from 0.18µm CMOS IBM 7RF technology. This implementation runs 1024 bit RSA encryption and decryption process in 8.44ms and the throughput of this implementation is 121.269Kbps.**

Keywords—Cryptography, RSA, Montgomery Modular Multiplication, CMOS, ASIC, VLSI.

I. INTRODUCTION

Information security is an important research and development area, and it is focused mainly on military and business applications [1]. In order to protect data, cryptographic algorithms were developed. The public-key RSA cryptosystem is widely used because it can perform public key encryption/decryption [2]. It is used to ensure authenticity of digital data and digital signatures. The security of this scheme relies on the fact that there are no algorithms, in modern computers, that efficiently factorize very large integers. In order to satisfy the ever growing security requirements of high-speed communications, such as personal communications services and wireless local area networks, a dedicated VLSI hardware solution is needed. An integrated circuit can meet the high throughput and high-volume market [3].

The core of RSA is a modular exponentiation, which consists of repetitive modular multiplication. The Montgomery algorithm is the most popular algorithm that achieves high performance in a modular multiplication operation [4]. Many architectures that implements this algorithm in hardware were proposed [1, 3, 5-11]. Solutions presented on FPGAs take the advantage of using dedicated embedded multipliers, digital signal processing and RAM blocks that enhances performance [5, 9]. Others combine Montgomery's algorithm with high radix redundant number system achieving a higher throughput [10, 11]. High radix architectures are optimized for some design parameters, such as size and speed, while the most suitable design point varies depending on the application and the user requirements [10]. A super-pipeline architecture is proposed by Zhao, Wang, Lu and Dai. [11], but the die area and the power consumption are severely affected. Among the Montgomery algorithm implemented in hardware, the radix-2 algorithm proposed by Daly and Marane [5] is primarily implemented with long k-bit adders, increasing fan-out and wire delays. To overcome this deficiency, efficient adder architectures can be designed to reduce the large fan-out and the wire delays for long operands. This work presents a radix-2 Montgomery modular multiplication algorithm, implemented in hardware, using an efficient kogge-stone adder, in order to achieve fast encryption/decryption RSA operation time.

This paper is organized as follows: Section II is a brief theoretical introduction to the RSA cryptosystem and its operation. Section III introduces the Montgomery method of modular multiplication. Section IV describes the radix-2 Montgomery modular multiplier based on systolic array architecture. Section V presents the 1024-bit RSA coprocessor design. Section VI presents the design results and comparison. Finally, some conclusions are drawn in Section VII.

II. THEORETICAL OVERVIEW

The RSA public-key cryptosystem was proposed by R. L. Rivest et al. in 1978 [2]. Encryption and decryption are identical modular exponentiation operations, where the only differences are the inputs. The modular exponentiation is performed by repeated modular multiplications [5].

In order to generate the public and private keys, two large prime numbers P and Q must be chosen. The computation of P times Q results in M, which will be used as the modulus. Its length, usually expressed in bits, is the key length. The Euler Totient Function, $\varphi(M)$, is used to calculate the number of positive integers smaller than M which are relatively prime to M, described by Equation 1.

$$\varphi(M) = (P-1) \times (Q-1) \tag{1}$$

Next, an integer E such that $1 < E < \varphi(M)$ and the greatest common divisor of E and $\varphi(M)$ equals to 1 must be chosen. Then, the integer D, which is the multiplicative inverse of E modulus $\varphi(M)$ is computed by Expression 2.

$$D \times E \equiv 1 \, mod\left(\varphi(M)\right) \tag{2}$$

The public key is the pair of positive integers (E, M). The private key is the pair of positive integers (D, M). To encrypt a message X, the Equation 3 must be used. To decrypt a ciphertext C, the Equation 4 must be used.

$$C = X^E \, mod \, M \tag{3}$$

$$X = C^D \, mod \, M \tag{4}$$

The public key is distributed to anyone, while the private key must be kept in secret in order to prevent attack.

III. MONTGOMERY MODULAR MULTIPLICATION ALGORITHM

The Montgomery modular multiplication algorithm was first introduced by P. L. Montgomery in 1985 [4]. While the ordinary modular multiplication algorithm to compute $A \times B \bmod M$ accumulates the digit-products and interleaves modular reduction to keep the result below M, the Montgomery algorithm uses the least significant bit of the intermediate result to perform addition rather than subtraction and, performs a shift down operation instead a shift up on each iteration [5].

However, the result produced by this algorithm is not exactly a modular multiplication of $A \times B \bmod M$, it produces $A \times B \times r^{-n} \bmod M$. The extra factor of r^{-n} produced must be removed in order to get the right result. The easiest way to do it is to perform another Montgomery modular multiplication by a constant of value $2^{2n} \bmod M$. Since the main objective of this algorithm is to compute exponentiations, it is preferable to pre-multiply the inputs by the constant and then, at the end, multiply the result by 1, to get rid of the 2^{-n} factor. This technique is used by most of the RSA coprocessors designed [1, 5, 6].

A bit wise version of the Montgomery modular multiplication algorithm is presented on Fig. 1 [6]. Assuming the algorithm in Fig. 1 as basis, the switch part of the code (lines 6-10) in Fig. 1 can be implemented in hardware as a 4 channel 1-bit multiplexer, performing the logical combination of a_i and q_i and a full adder can performs the computation of the carry propagation bit and the result of the sum of each bit (line11) [6].

IV. RADIX-2 MONTGOMERY MODULAR MULTIPLIER

The design of the radix-2 Montgomery modular multipliers is based on the systolic hardware architecture for Montgomery modular multiplication presented on [6]. A block diagram form of the implementation is presented on Fig. 2. This block computes the value of the expression $A \times B \times r^{-n} \bmod M$. The inputs of the block are a $(n+2)$-bit operand A, a $(n+2)$-bit operand B, a $(n+2)$-bit operand M, an enable signal, a reset signal and a clock signal, where n is the number of the bits for the design, in this particular case 1024. The other input signals ldR, ldA and $lock$ are control signals. These signals are generated by an external finite state machine that will be explained in the following section.

Adders are important digital circuits that are used not only in arithmetic logic units, but also in other parts of processors to calculate addresses, table indices and similar operations. The simplest form of an n-bit adder is connect full bit adders in a serial circuit, named ripple-carry-adder (RCA). The circuit of such adders is simple, which allows fast design time and a small area; however, the performance is slow. Parallel prefix look-ahead adders perform better than RCA in terms of speed. A kogge-stone adder (KSA), which is the fastest parallel prefix look-ahead adder, generates the carry signal in $O(log\ n)$ time, while a RCA generates it in $O(n)$ time. The KSA maximum fan-out never exceeds 2; however the area to design it is larger and the routing is complex [12, 13]. In order to get a reduced latency time and achieve the best throughput performance of this design, the Circ. 1 and Circ. 5 adders were implemented as KSA.

The KSA1 (Circ. 1) produces the sum of the inputs B and M, called MB on Fig.1 and Fig 2. Similar to the KSA1, the KSA2 (Circ.5) produces the sum of the signals reg_rji and out, called reg_rjo. The sub-circuits Circ. 2 and Circ. 4 illustrated on Fig. 2 are $(n+2)$-bit shift registers. They are composed by a $(n+2)$ D-type flip-flop connected serially. The shiftreg1 (Circ. 2) serializes the $(n+2)$ operand A at every clock cycle. Meanwhile, the shiftreg2 (Circ.4) shifts right 1-bit the input reg_rjo at every clock cycle.

The sub-circuit Circ. 6 illustrated on Fig. 2 is a register that preserves the output of the modular multiplication after the computation is completed. The sub-circuit Circ. 3 illustrated on Fig. 2 is composed by a row of processing elements. They are combinational logic circuits. The processing element PE_0 is illustrated on Fig. 3. It produces an output signal q_o that depends on the value of the least significant bit intermediate result $reg\_rji[0]$ of the modular multiplication, the $B[0]$-bit input and the $A[i]$-bit in the respective clock cycle. This processing element corresponds to the line 4 of Fig. 1.

The output signal q_o of the processing element PE_0 is the input signal q_i to the rest of the processing elements $PE_1, PE_2, PE_3, \ldots, PE_i, \ldots$ and PE_{n+1}. The combination of the input q_i and $A[i]$ selects the appropriated value for the output of these processing elements. This logical combination corresponds to the lines 6-10 of the Fig. 1. The processing elements $PE_1, PE_2, PE_3, \ldots, PE_i, \ldots$ and PE_{n+1} are illustrated on Fig. 4. The dataflow of the radix-2 Montgomery modular multiplier is explained as follows. Once the inputs are ready, the reset signal clears all the data from the three shift registers. Meanwhile, the KSA1 produces the sum of B and M. After data is cleared, the ldA signal loads the shiftreg1 with the operand A and the shiftreg2 remains cleared. As soon as the operand A is loaded, the processing element PE_0 produces the outputs q_o and $out[0]$. The other processing elements PE_1 to PE_{n+1} produces their output signals out. Once the signal out ($out[n+1] \ldots out[0]$), on Circ. 3 (Fig. 2) is ready, the

```
Algorithm SysMon(A, B, M, MB)
{
1:  int R ← 0;
2:  bit carry ← 0, x;
3:  for i = 0 to n
4:      q_i ← r_0^(i) xor (a_i and b_0);
5:      for j = 0 to n
6:          switch (a_i, q_i):
7:              1, 1: x ← mb_j;
8:              1, 0: x ← b_j;
9:              0, 1: x ← m_j;
10:             0, 0: x ← 0;
11:         r_(j)^(i+1) ← r_(j+1)^(i) xor x xor carry;
12:     end for
13:     carry ← (r_(j+1)^(i) and x_j) or (r_(j+1)^(i) and carry)
                                  or (x_j and carry);
14: end for
15: return R;
}
```

Fig. 1. Systolic Montgomery multiplication algorithm [6]

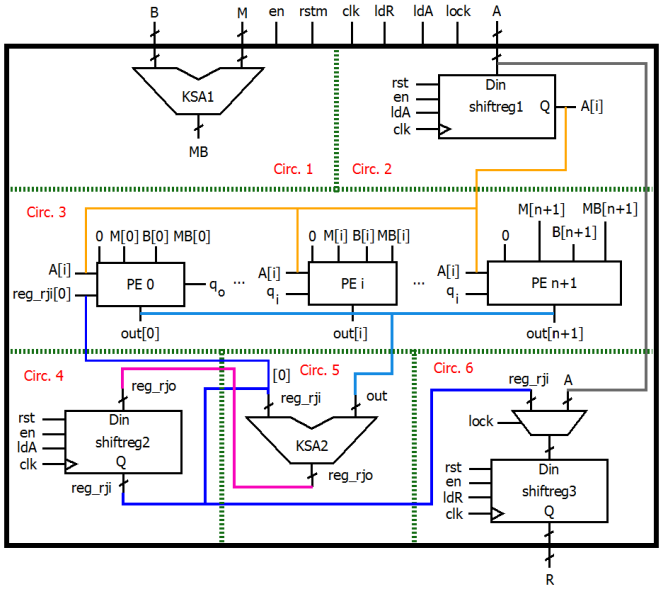

Fig. 2. Radix-2 Montgomery modular multiplier

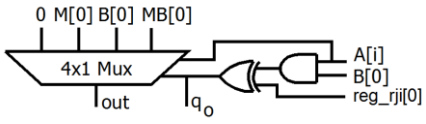

Fig. 3. Processing element PE_0

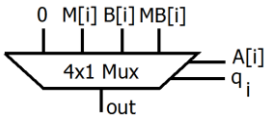

Fig. 4. Processing elements PE_1 to PE_{n+1}

KSA2 performs the sum of signal out and reg_rji. Then, the KSA2 output reg_rjo, is registered at the shiftreg2 shifted right one bit in the following clock cycle. After $(n+2)$ clock cycles, the ldR signal is set and the shiftreg3 load the computation of expression $A \times B \times r^{-n} \bmod M$, depending on the value of the $lock$ signal. The $lock$ signal is used when the Montgomery modular multiplication must not be performed, and the operand A must be copied directly to the result register. The number of clock cycles for this computation is $(n+4)$.

V. 1024-BIT RSA COPROCESSOR

There are two common algorithms to compute the modular exponentiations (3) and (4) that are widely used: the L-R Binary Method, which is area optimized, and the R-L Binary Method, which is speed optimized [5]. On this paper, since the main concern was to achieve a better throughput, the R-L Binary algorithm was implemented. In order to design the 1024-bit RSA coprocessor, two blocks of the radix-2 Montgomery modular multiplier were used. The architecture of this coprocessor is based on the R-L Binary Method exponentiation algorithm presented on [5]. A finite state machine to control the hardware was also designed. The architecture is illustrated on Fig. 5. The MMM1 and MMM2 blocks are the Montgomery modular multipliers. The inputs of

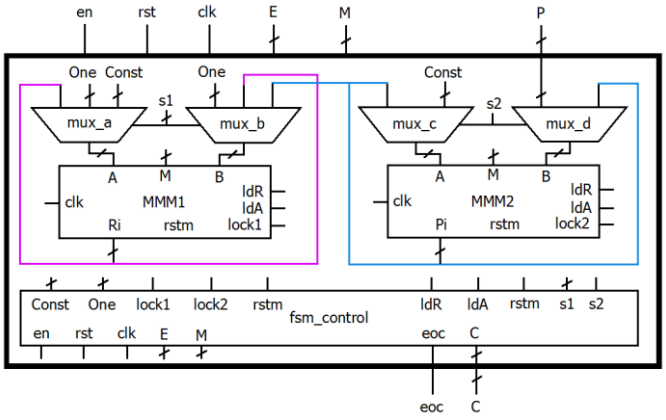

Fig. 5. 1024-bit RSA coprocessor

this block are an enable signal, a reset signal, a clock signal, an 1024-bit plaintext P, an 1024-bit exponent E and an 1024-bit modulus M. The outputs of this block are an 1024-bit chipertext C and an eoc signal to indicates the end of the conversion.

The first step of the modular exponentiation is the Mapping phase, where the input P and the previously computed constant are mapped to the Montgomery domain. The finite state machine, block fsm_control, through the signals $s1$ and $s2$, controls the multiplexers. The signal $s1$ selects the $Const$ $(2^{2n} \bmod M)$ operand from mux_a and the One from mux_b. The signal $s2$ selects the operand $Const$ from mux_c and the operand P from mux_d. The two Montgomery modular multiplications are done in parallel and their results are stored in their internal registers Ri and Pi.

After the Mapping phase, the signal $s1$ selects the operand Ri from mux_a and Pi from mux_b. Meanwhile, the signal $s2$ selects the operands Pi from both mux_c and mux_d. This configuration will persist until the end of the Multiply and Square phase. In this phase, all the bits of the exponent input E bits are serialized and analyzed. The $lock1$ signal from MMM1 is used in the case when the E_i-bit is zero. In this particular case, the Montgomery modular multiplication must not be performed, so the result register Ri receives a copy of the input Ri. Meanwhile, the MMM2 must always Montgomery modular multiply its inputs, which concludes that the signal $lock2$ never "locks" the output of the block.

Once the Multiply and Square phase is done, the finite state machine updates the $s1$ signal in order to prepare the inputs of the MMM1 block to the Re-Mapping phase. The MMM2 block is not used at this phase. The $s1$ signal selects the operand One from mux_a and the operand Ri from mux_b. After the result is computed, the eoc signal indicates that the ciphertext C retains the value of the modular exponentiation $(P^E \bmod M)$. The number of clock cycles of the design is $(n+4) \times (n+3)$, which in the case of the 1024-bit design is equal to 1,055,756.

VI. RESULTS AND COMPARISON

The design was described using Verilog HDL and the synthesis was performed on Cadence© Encounter RTL Compiler using IBM 0.18µm standard cells. The total cell area of the design is 2.723mm^2, and the scale equals 107k NANDs.

TABLE I. PERFORMANCE COMPARISON WITH OTHER PAPER

Paper	Technology	Radix	Clock Frequency (MHz)	Modular exp. time (ms)	Throughput (kbps)	Power (mW)	Scale (k NANDs)
[1]	Stratix III	2	380.66	11.06	92.59	---	---
[7]	CMOS 0.5µm	2	80.00	45.00*	22.75*	---	---
[8]	CMOS 90nm	2	1087.10	1.93	530.57*	---	776
[9]	Virtex-6	17	410.69	11.26	88.78	---	---
[10]	CMOS 90nm	32	471.70	7.27	140.85*	---	11.5
[11]	CMOS 0.18µm	32	140.00	---	6350.00	1600.00	923
This work	CMOS 0.18µm	2	125.03	8.44	121.27	797.94	107

*Author's estimate based on original papers.

The total power consumption is 797.942mW and the maximum operation frequency is 125.03MHz. The design can encrypt/decrypt a message of 1024 bits in 8.44ms with a 125MHz clock. The throughput of this implementation is 121.269Kbps. Table I shows a comparison of other works with our results. It is important to state that all data from Table I is concerned to 1024-bit RSA designs using Montgomery modular multiplication architectures.

The paper that presents the fastest exponentiation time found in the literature is [8]. The technology and the critical path of [8] increase the clock frequency of the design; however the area of this architecture scales 776k NANDs, which is 7.25 times bigger than the proposed in this work. Zhao et. al [11] present a pipelined architecture that increases substantially the performance of the design is presented; however, the design average power dissipation is 1.6W, which is the double of the design presented on this paper. Also the scale of [11] is 923k NANDs, 8.62 times bigger than the design of this paper. Miyamoto [10] presents a large set of adder's architectures and algorithms with different radixes. Compared with this work, the proposed architecture on CMOS 0.18µm calculates a modular exponentiation operation, with less clock cycles, at almost the same time as the balanced architecture of [10] on CMOS 90nm.

Paper [1] presents an architecture based on a systolic array, where processing elements are closed to each other. Even though the clock frequency presented on this work is lower than [1], a higher throughput was achieved. Paper [9] propose a radix-17 architecture using a single DSP48E1 block, 201 slice registers and 374 slices LUTs and one 36Kbits BRAM. The time to compute a modular exponentiation is 11.26ms.

VII. CONCLUSION

This paper presented the design of a 1024-bit RSA coprocessor, which uses a radix-2 Montgomery modular multiplier based on a systolic architecture, implemented in hardware, using an efficient kogge-stone adder, in order to achieve fast modular exponentiation time. This implementation was designed using Verilog HDL and synthesized on Cadence© Encounter RTL Compiler using IBM 7RF 0.18µm standard cells. The design can encrypt/decrypt a message with 1024 bits in 8.44ms with a 125MHz clock and its throughput is 121.269Kbps.This implementation is compared with others architectures and comparison results are presented. The comparison of the results shows that the designed coprocessor presents a good area-throughput trade-off. The reduced power consumption of the proposed architecture also enhances its performance.

ACKNOWLEDGMENT

The authors acknowledge CAPES, CNPq and FAPEMIG for their financial support. The authors also would like to thank Cadence©, MOSIS and IBM.

REFERENCES

[1] Renteria-Mejia, C.P.; Trujillo-Olaya, V.; Velasco-Medina, J., "Design of an 8192-bit RSA cryptoprocessor based on systolic architecture," Programmable Logic (SPL), 2012 VIII Southern Conference on , pp.1-6, 20-23 March 2012.

[2] Rivest, R. L., Shamir, A., Adleman, L.: "A Method for obtaining digital signatures and public-key cryptosystems", *Commun, ACM*, 1978, 21, (2), pp. 120-126.

[3] Yong-Jin Jeong; Burleson, W.P., "VLSI array algorithms and architectures for RSA modular multiplication," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.5, no.2, pp.211-217, June 1997.

[4] P. L.: "Modular Multiplication Without Trial Division", *Math. Comput.*, 1985, 44, pp. 519-521.

[5] Daly, A., Marane, W.: "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*(FPGA '02), ACM, pp. 40-49.

[6] Nedjah, N.; Mourelle, L. M.; Santana, M.; Raposo, S., "Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems," Computers & Digital Techniques, IET , vol.6, no.5, pp.290-301, September 2012.

[7] Tenca, A.F.; Koc, C.K., "A scalable architecture for modular multiplication based on Montgomery's algorithm," Computers, IEEE Transactions on , vol.52, no.9, pp.1215-1221, Sept. 2003.

[8] Wang, Y.; Maskell, D. L.; Leiwo, J., "A unified architecture for a public key cryptographic coprocessor" Journal of System Architectures, 2008.

[9] Bo Song; Ito, Y.; Nakano, K., "CRT-Based DSP Decryption Using Montgomery Modular Multiplication on the FPGA," Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on , vol., no., pp.532-541, 16-20 May 2011.

[10] Miyamoto, A.; Homma, N.; Aoki, T.; Satoh, A., "Systematic Design of RSA Processors Based on High-Radix Montgomery Multipliers," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.19, no.7, pp.1136-1146, July 2011.

[11] Xuemi Zhao; Zhiying Wang; Hongyi Lu; Kui Dai, "A 6.35Mbps 1024-bit RSA crypto coprocessor in a 0.18um CMOS technology,"Very Large Scale Integration, 2006 IFIP International Conference on, pp.216-221, 16-18 Oct. 2006

[12] Parhami B, : C*omputer Arithmetic: Algorithms and Hardware Designs.* Oxford, U.K.: Oxford Univ. Press, 2000.

[13] Moudallal, Z.; Issa, I.; Mansour, M.; Chehab, A.; Kayssi, A., "A low-power methodology for configurable wide kogge-stone adders,"Energy Aware Computing (ICEAC), 2011 International Conference on, pp.1-5, Nov. 30 2011-Dec. 2 2011

Design of a 1024 bit RSA Coprocessor with SPI Slave Interface

Caio A. da Costa, Robson L. Moreno, Otávio S. A. Carpinteiro, Tales C. Pimenta
Department of Microelectronics
Universidade Federal de Itajubá - UNIFEI
Itajubá, MG, Brazil

***Abstract*—This paper presents the architecture and model of a modular exponentiation hardware for RSA public key cryptography algorithm with a SPI slave interface for on-board peripheral communication. A radix 2 Montgomery modular multiplication hardware based on a systolic implementation was designed. A kogge-stone adder is used to reduce the critical path and improve throughput. Cadence© Encounter RTL Compiler was used to synthesize the RTL code described in Verilog HDL. The coprocessor was implemented with standard cells library from 0.18µm CMOS IBM 7RF technology.The SPI maximum SPI transfer rate is 100 Mb/s.This implementation runs 1024 bit RSA encryption and decryption process in 8.44ms and the throughput of this implementation is 121.269Kbps.**

Keywords—Cryptography, RSA, Montgomery Modular Multiplication, CMOS, ASIC, VLSI.

I. INTRODUCTION

Information security is an important research and development area, and it is focused mainly on military and business applications [1]. In order to protect data, cryptographic algorithms were developed. The public-key RSA cryptosystem is widely used because it can perform public key encryption/decryption [2]. It is used to ensure authenticity of digital data and digital signatures. The security of this scheme relies on the fact that there are no algorithms, in modern computers, that efficiently factorize very large integers. In order to satisfy the ever growing security requirements of high-speed communications, such as personal communications services and wireless local area networks, a dedicated VLSI hardware solution is needed. An integrated circuit can meet the high throughput and high-volume market [3].

The core of RSA is a modular exponentiation, which consists of repetitive modular multiplication. The Montgomery algorithm is the most popular algorithm that achieves high performance in a modular multiplication operation [4]. Many architectures that implements this algorithm in hardware were proposed [1, 3, 5-11]. Solutions presented on FPGAs take the advantage of using dedicated embedded multipliers, digital signal processing and RAM blocks that enhances performance [5, 9]. Others combine Montgomery's algorithm with high radix redundant number system achieving a higher throughput [10, 11]. High radix architectures are optimized for some design parameters, such as size and speed, while the most suitable design point varies depending on the application and the user requirements [10].A super-pipeline architecture is proposed by Zhao, Wang, Lu and Dai. [11], but the die area and the power consumption are severely affected. Among the Montgomery algorithm implemented in hardware, the radix-2 algorithm proposed by Daly and Marane [5] is primarily implemented with long *k*-bit adders, increasing fan-out and wire delays. To overcome this deficiency, efficient adder architectures can be designed to reduce the large fan-out and the wire delays for long operands. This work presents a radix-2 Montgomery modular multiplication algorithm, implemented in hardware, using an efficient kogge-stone adder, in order to achieve fast encryption/decryption RSA operationtime.

The SPI and I²C protocol are well suited for communications between integrated circuits on the same board. Even though there is no formal specification for SPI protocol, the main advantage of using this protocol is that the transfer rate depends only on implementation technology; meanwhile the I²C interface is limited to the transfer rate of its standard. Since the main concern is to provide data as fast as possible, a SPI interface meets the requirements of the proposed coprocessor [12, 13].

This paper is organized as follows: Section 2 is a brief theoretical introduction to the RSA cryptosystem and its operation. Section 3 introduces the Montgomery method of modular multiplication. Section 4 describes the radix 2 Montgomery modular multiplier based on systolic array architecture. Section5 presents the 1024 bit RSA coprocessor design. Section 6 presents the design results and comparison. Finally, some conclusions are drawn in Section 7.

II. THEORETICAL OVERVIEW

The RSA public-key cryptosystem was proposed by R.L. Rivest, A. Shamir and L. Adleman in 1978 [2]. Encryption and decryption are identical modular exponentiation operations, where the only differences are the inputs. The modular exponentiation is performed by repeated modular multiplications [5].

In order to generate the public and private keys, two large prime numbers P and Q must be chosen. The computation of P times Q results in M, which will be used as the modulus. Its length, usually expressed in bits, is the key length. The Euler Totient Function, $\varphi(M)$, is used to calculate the number of positive integers smaller than M which are relatively prime to M, described by equation (1).

$$\varphi(M) = (P-1) \times (Q-1) \quad (1)$$

Next, an integer E such that $1 < E < \varphi(M)$ and the greatest common divisor of E and $\varphi(M)$ equals to 1 must be

chosen. Then, the integer D, which is the multiplicative inverse of E modulus $\varphi(M)$ is computed by expression (2).

$$D \times E \equiv 1 \, mod \left(\varphi(M)\right) \quad (2)$$

The public encryption key is the pair of positive integers (E, M). The private decryption key is the pair of positive integers (D, M). To encrypt a message X, the equation (3) must be used. To decrypt a ciphertext C, the equation (4) must be used.

$$C = X^E \, mod \, M \quad (3)$$

$$X = C^D \, mod \, M \quad (4)$$

The public key is distributed to anyone, while the private key must be kept in secret in order to prevent attack. Currently, RSA Lab recommends 1024 bits key size for corporate use[14].

III. MONTGOMERY MODULAR MULTIPLICATION ALGORITHM

The Montgomery modular multiplication algorithm was first introduced by P. L. Montgomery in 1985 [4].While the ordinary modular multiplication algorithm to compute $A \times B \, mod \, M$ accumulates the digit-products and interleaves modular reduction to keep the result below M, the Montgomery algorithm uses the least significant bit of the intermediate result to perform addition rather than subtraction and, performs a shift down operation instead a shift up on each iteration [5].

However, the result produced by this algorithm is not exactly a modular multiplication of $A \times B \, mod \, M$, it produces $A \times B \times r^{-n} \, mod \, M$. The extra factor of r^{-n} produced must be removed in order to get the right result. The easiest way to do it is to perform another Montgomery modular multiplication by a constant of value $2^{2n} \, mod \, M$. Since the main objective of this algorithm is to compute exponentiations, it is preferable to pre-multiply the inputs by the constant and then, at the end, multiply the result by 1, to get rid of the 2^{-n} factor. This technique is used by most of the RSA coprocessors designed [1, 5, 6].

A bit wise version of the Montgomery modular multiplication algorithm is presented on Fig. 1 [6]. Assuming the algorithm in Fig. 1 as basis, the switch part of the code (lines 6-10) in Fig. 1 can be implemented in hardware as a 4 channel 1 bit multiplexer, performing the logical combination of a_i and q_i and a full adder can performs the computation of the carry propagation bit and the result of the sum of each bit (line11) [6].

IV. RADIX 2 MONTGOMERY MODULAR MULTIPLIER

The design of the radix 2 Montgomery modular multipliers is based on the systolic hardware architecture for Montgomery modular multiplication presented on [6]. A block diagram form of the implementation is illustrated on Fig. 2. This block computes the value of the expression $A \times B \times r^{-n} \, mod \, M$. The inputs of the block are a $(n+2)$ bit operand A, a $(n+2)$ bit operand B, a $(n+2)$ bit operand M, an enable signal, a reset signal and a clock signal, where n is the number of the bits for the design, in this particular case 1024. The other input signals ldR, ldA and $lock$ are control signals. These signals are generated by an external finite state machine.

Adders are important digital circuits that are used not only in arithmetic logic units, but also in other parts of processors to calculate addresses, table indices and similar operations. The simplest form of a n-bit adder is connect full bit adders in a serial circuit, named ripple-carry-adder (RCA). The circuit of such adders is simple, which allows fast design time and a small area; however, the performance is slow. Parallel prefix look-ahead adders perform better than RCA in terms of speed. A kogge-stone adder (KSA), which is the fastest parallel prefix look-ahead adder, generates the carry signal in $O(log \, n)$ time, while a RCA generates it in $O(n)$ time.The KSA maximum fan-out never exceeds 2; however the area to design it is larger and the routing is complex [15, 16]. In order to get a reduced latency time and achieve the best throughput performance of this design, the Circ. 1and Circ. 5 adders were implemented as KSA.

The KSA1 (Circ. 1) produces the sum of the inputs B and M, called MB on Fig.1 and Fig 2. Similar to the KSA1, the KSA2 (Circ.5) produces the sum of the signals reg_rji and out, called reg_rjo. The sub-circuits Circ. 2 and Circ. 4 illustrated on Fig. 2 are $(n+2)$ bit shift registers. They are composed by a $(n+2)$ D-type flip-flop connected serially. The shiftreg1 (Circ. 2) serializes the $(n+2)$ operand A at every clock cycle. Meanwhile, the shiftreg2 (Circ.4) shifts right 1 bit the input reg_rjo at every clock cycle.

The sub-circuit Circ. 6 illustrated on Fig. 2 is a register that preserves the output of the modular multiplication after the computation is completed.The sub-circuit Circ. 3 illustrated on Fig. 2 is composed by a row of processing elements. They are combinational logic circuits. The processing element PE_0 is illustrated on Fig. 3. It produces an output signal q_o that depends on the value of the least significant bit intermediate result $reg\_rji[0]$ of the modular multiplication, the $B[0]$ bit input and the $A[i]$ bit in the respective clock cycle. This processing element corresponds to the line 4 of Fig. 1.

The output signal q_o of the processing element PE_0 is the input signal q_i to the rest of the processing elements $PE_1, PE_2, PE_3, \ldots, PE_i, \ldots$ and PE_{n+1}. The combination of the input q_i and $A[i]$ selects the appropriated value for the output of these processing elements. This logical combination corresponds to the lines 6-10 of the Fig. 1. The processing elements $PE_1, PE_2, PE_3, \ldots, PE_i, \ldots$ and PE_{n+1} are illustrated on Fig. 4. The dataflow of the radix-2 Montgomery modular multiplier is explained as follows. Once the inputs are ready, the reset signal clears all the data from the three shift registers. Meanwhile, the KSA1 produces the sum of B and M. After data is cleared, the ldA signal loads the shiftreg1 with the operand A and the shiftreg2 remains cleared. As soon as the operand A is loaded, the processing element PE_0 produces the outputs q_o and $out[0]$. The other processing elements PE_1 to PE_{n+1} produces their output signals out. Once the signal out $(out[n+1] \ldots out[0])$, on Circ. 3 (Fig. 2) is ready, the KSA2 performs the sum of signal out and reg_rji. Then, the KSA2 output reg_rjo, is registered at the shiftreg2 shifted right one bit in the following clock cycle. After $(n+2)$ clock cycles, the ldR signal is set and the shiftreg3 load the computation of expression $A \times B \times r^{-n} \, mod \, M$, depending on the value of the $lock$ signal. The $lock$ signal is used when the

---

Algorithm SysMon(A, B, M, MB)

{

1: *int* $R \leftarrow 0$;

2: *bit* $carry \leftarrow 0, x$;

3: *for* $i = 0$ *to* n

4: $\quad q_i \leftarrow r_0^{(i)}$ ***xor*** $(a_i$ ***and*** $b_0)$;

5: $\quad$ *for* $j = 0$ *to* n

6: $\quad\quad$ *switch* (a_i, q_i):

7: $\quad\quad\quad 1,1: x \leftarrow mb_j$;

8: $\quad\quad\quad 1,0: x \leftarrow b_j$;

9: $\quad\quad\quad 0,1: x \leftarrow m_j$;

10: $\quad\quad\quad 0,0: x \leftarrow 0$;

11: $\quad\quad r_{(j)}^{(i+1)} \leftarrow r_{(j+1)}^{(i)}$ ***xor*** x ***xor*** $carry$;

12: $\quad$ *end for*

13: $\quad carry \leftarrow \left(r_{(j+1)}^{(i)}\right.$ ***and*** $\left.x_j\right)$ ***or*** $\left(r_{(j+1)}^{(i)}\right.$ ***and*** $\left.carry\right)$ ***or*** $\left(x_j\right.$ ***and*** $\left.carry\right)$;

14: *end for*

15: *return* R;

}

Fig. 1. Systolic Montgomery multiplication algorithm [6]

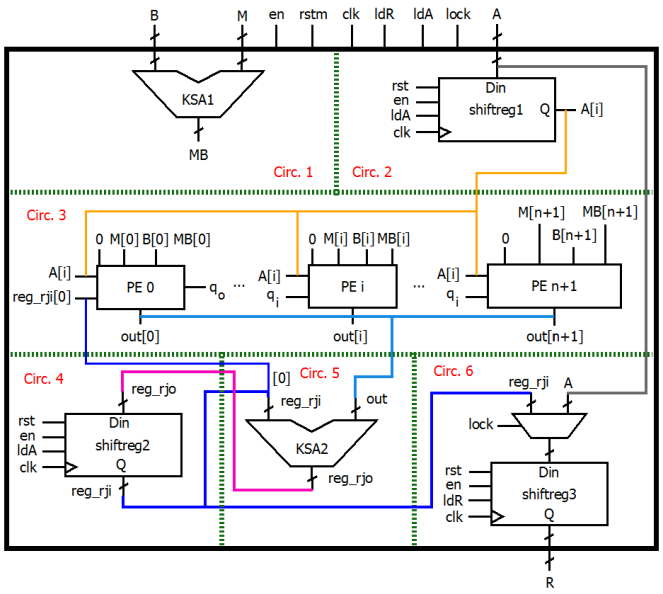

Fig. 2. Radix 2 Montgomery modular multiplier

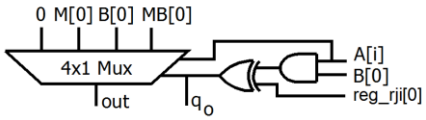

Fig. 3. Processing element PE_0

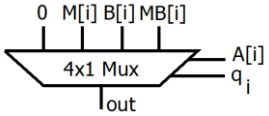

Fig. 4. Processing elements PE_1 to PE_{n+1}

Montgomery modular multiplication must not be performed, and the operand A must be copied directly to the result register. The number of clock cycles for this computation is $(n + 4)$.

V. 1024BIT RSA COPROCESSOR

There are two common algorithms to compute the modular exponentiations (3) and (4) that are widely used:the L-R Binary Method, which is area optimized, and the R-L Binary Method, which is speed optimized [5]. On this paper, since the main concern was to achieve a better throughput, the R-L Binary algorithm was implemented. In order to design the 1024 bit RSA coprocessor, two blocks of the radix-2 Montgomery modular multiplier were used. The architecture of this coprocessor is based on the R-L Binary Method exponentiation algorithm presented on [5].A finite state machine to control the hardware was also designed. The architecture is illustrated on Fig. 5. The MMM1 and MMM2 blocks are the Montgomery modular multipliers. The inputs of this block are an enable signal, a reset signal, a clock signal, an 1024 bit plaintext P, an 1024 bit exponent E and an 1024 bit modulus M. The outputs of this block are an 1024 bit chipertext C and an eoc signal to indicates the end of the conversion.

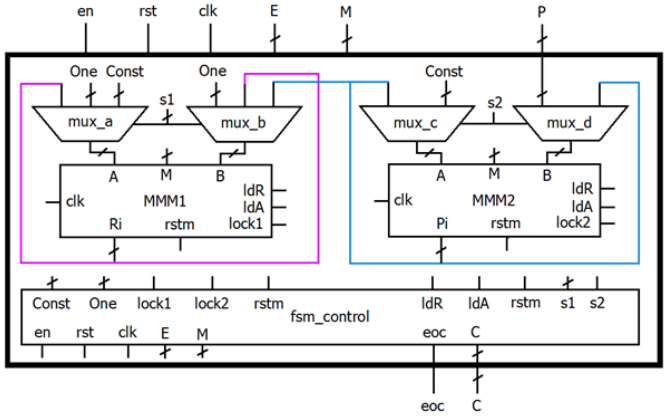

Fig. 5. 1024bit RSA coprocessor

After the result is computed, the eoc signal indicates that the ciphertext C retains the value of the modular exponentiation $(P^E \ mod \ M)$. The number of clock cycles of the design is $(n + 4) \times (n + 3)$, which in the case of the 1024 bit design is equal to 1,055,756. The SPI slave interface, along with its controller, is used to load the bank of register of the operands P, E, M and $Const$ and also to read the result C after the operation is complete. The amount of time to transfer 1024 bits through the SPI bus is 10.24 μs.

VI. RESULTS AND COMPARISON

The design was described using Verilog HDL and the synthesis was performed on Cadence© Encounter RTL Compiler using IBM 0.18µm standard cells. The total cell area of the modular exponentiation is 2.723mm², and the scale equals 107k NANDs. The total power consumption is 797.942mW and the maximum operation frequency is 125.03MHz. The design can encrypt/decrypt a message of 1024 bits in 8.44ms with a 125MHz clock. The throughput of this implementation is 121.269Kbps. Table I shows a comparison of other works with our results. It is important to

TABLE I. PERFORMANCE COMPARISON WITH OTHER PAPER

Paper	Technology	Radix	Clock Frequency (MHz)	Modular exp.time (ms)	Throughput (kbps)	Power (mW)	Scale (k NANDs)
[1]	Stratix III	2	380.66	11.06	92.59	---	---
[7]	CMOS 0.5µm	2	80.00	45.00*	22.75*	---	---
[8]	CMOS 90nm	2	1087.10	1.93	530.57*	---	776
[9]	Virtex-6	17	410.69	11.26	88.78	---	---
[10]	CMOS 90nm	32	471.70	7.27	140.85*	---	11.5
[11]	CMOS 0.18µm	32	140.00	---	6350.00	1600.00	923
This work	CMOS 0.18µm	2	125.03	8.44	121.27	797.94	107

*Author's estimated based on original papers.

state that all data from Table I is concerned to 1024 bit RSA design using Montgomery modular multiplication architectures.

The paper that presents the fastest exponentiation time found in the literature is [8].The technology and the critical path of [8] increase the clock frequency of the design; however the area of this architecture scales 776k NANDs, which is 7.25 times bigger than the proposed in this work. Reference [11] presents a pipeline architecture that increases substantially the performance of the design; however, the design average power dissipation is 1.6W, which is the double of the design presented on this paper. Also the scale of [11] is 923k NANDs, 8.62 times bigger than the design of this paper. Reference [10] presents a large set of adder's architectures and algorithms with different radixes. Compared with this work, the proposed architecture on CMOS 0.18µm calculates a modular exponentiation operation, with less clock cycles, at almost the same time as the balanced architecture of [10] on CMOS 90nm.

Paper [1] presents an architecture based on a systolic array, where processing elements are closed to each other. Even though the clock frequency presented on this work is lower than [1], a higher throughput was achieved. Paper [9] propose a radix-17 architecture using a single DSP48E1 block, 201 slice registers and 374 slices LUTs and one 36Kbits BRAM. The time to compute a modular exponentiation is 11.26ms.

VII. CONCLUSION

This paper presented the design of a 1024 bit RSA coprocessor, which uses a radix-2 Montgomery modular multiplier based on a systolic architecture, implemented in hardware, using an efficient kogge-stone adder, in order to achieve fast modular exponentiation time. This implementation was designed using Verilog HDL and synthesized on Cadence© Encounter RTL Compiler using IBM 7RF 0.18µm standard cells.The design can encrypt/decrypt a message with 1024 bits in 8.44ms with a 125MHz clock and its throughput is 121.269Kbps.This implementation is compared with others architectures and comparison results are presented. The comparison of the results shows that the designed coprocessor presents a good area-throughput trade-off. The reduced power consumption of the proposed architecture also enhances its performance.

ACKNOWLEDGMENT

The authors acknowledge CAPES, CNPq and FAPEMIG for their financial support.

REFERENCES

[1] Renteria-Mejia, C.P.; Trujillo-Olaya, V.; Velasco-Medina, J., "Design of an 8192-bit RSA cryptoprocessor based on systolic architecture," Programmable Logic (SPL), 2012 VIII Southern Conference on , vol., no., pp.1-6, 20-23 March 2012.

[2] Rivest, R. L., Shamir, A., Adleman, L.: "A Method for obtaining digital signatures and public-key cryptosystems", *Commun, ACM*, 1978, 21, (2), pp. 120-126.

[3] Y. Jeong; Burleson, W.P., "VLSI array algorithms and architectures for RSA modular multiplication," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.5, no.2, pp.211-217, June 1997.

[4] P. L. Montgomery: "Modular Multiplication Without Trial Division", *Math. Comput.*, 1985, 44, pp. 519-521.

[5] Daly, A., Marane, W.: "Efficient Architectures for implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic", *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*(FPGA '02), ACM, pp. 40-49.

[6] Nedjah, N.; Mourelle, L. M.; Santana, M.; Raposo, S., "Massively parallel modular exponentiation method and its implementation in software and hardware for high-performance cryptographic systems," Computers & Digital Techniques, IET , vol.6, no.5, pp.290-301, September 2012.

[7] Tenca, A.F.; Koc, C.K., "A scalable architecture for modular multiplication based on Montgomery's algorithm," Computers, IEEE Transactions on , vol.52, no.9, pp.1215-1221, Sept. 2003.

[8] Wang, Y.; Maskell, D. L.; Leiwo, J., "A unified architecture for a public key cryptographic coprocessor" Journal of System Architectures, 2008.

[9] Bo Song; Ito, Y.; Nakano, K., "CRT-Based DSP Decryption Using Montgomery Modular Multiplication on the FPGA," Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on , vol., no., pp.532-541, 16-20 May 2011.

[10] Miyamoto, A.; Homma, N.; Aoki, T.; Satoh, A., "Systematic Design of RSA Processors Based on High-Radix Montgomery Multipliers," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.19, no.7, pp.1136-1146, July 2011.

[11] X. Zhao; Z. Wang; H. Lu; K. Dai, "A 6.35Mbps 1024-bit RSA crypto coprocessor in a 0.18um CMOS technology,"Very Large Scale Integration, 2006 IFIP International Conference on, vol., no., pp.216-221, 16-18 Oct. 2006

[12] Leens, F., "An introduction to I²C and SPI protocols,"Instrumentation & Measurement Magazine, IEEE, vol.12, no.1, pp.8,13, February 2009

[13] Oudjida, A. K.; Berrandjia, M. L.; Tiar, R.; Liacha, A.; Tahraoui, K., "FPGA implementation of I²C SPI protocols: A comparative study,"Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on, vol., no., pp.507,510, 13-16 Dec. 2009.

[14] RSA Labs, available at http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/key-size.htm, 2014.

[15] Parhami B, : "Computer Arithmetic – Algorithms and Hardware Designs" (Oxford University Press, 2000).

[16] Moudallal, Z.; Issa, I.; Mansour, M.; Chehab, A.; Kayssi, A., "A low-power methodology for configurable wide kogge-stone adders,"Energy Aware Computing (ICEAC), 2011 International Conference on, vol., no., pp.1-5, Nov. 30 2011-Dec. 2 2011