FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” - UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIAS DA COMPUTAÇÃO

FERNANDO AUGUSTO GARCIA MUZZI

O PADRÃO DE SEGURANÇA PKCS#11 EM FPGAS: RSA UM ESTUDO DE CASO

Marília
2005

O PADRÃO DE SEGURANÇA PKCS#11 EM FPGA: RSA UM ESTUDO DE CASO

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para obtenção do Título de Mestre em Ciências da Computação. (Área de Concentração: Arquitetura de Computadores).

Orientador
Prof. Dr. Edward David Moreno Ordoñez

MARÍLIA
2005

FERNANDO AUGUSTO GARCIA MUZZI

O PADRÃO DE SEGURANÇA PKCS#11 EM FPGAS: RSA UM ESTUDO DE CASO

Banca examinadora da dissertação apresentada ao programa de mestrado UNIVEM/F.E.E.S.R., para a obtenção do título de mestre em ciência da computação. Na área de concentração: Arquitetura de Computadores.

Resultado : APROVADO

Orientador: Prof. Dr. Edward David Moreno Ordonez

1º Examinador: Prof. Dr. Ildeberto Aparecido Rodello

2º Examinador: Prof. Dr. Adilson Eduardo Guelfi

Marília, 05 de agosto de 2005.

AGRADECIMENTOS

À Deus, por guiar meus passos.

À minha esposa, Cristiana Cândido Amorim Muzzi, pelo apoio, incentivo e motivação.

A minha avó Ana Teixeira Muzzi (in memorian), pelo apoio para lutar pelos meus sonhos.

Ao meu orientador, Prof. Dr. Edward David Moreno Ordonez, pelo apoio, discussões, sugestões, paciência e conselhos, que muito me ajudaram no desenvolvimento deste trabalho.

Aos amigos César Giacomini Penteado, Fábio Dacêncio Pereira e Rodolfo Barros Chiaramonte pela amizade e força nos dias difíceis.

A todos os professores do mestrado da Fundação Eurípides Soares da Rocha de Marilia pelos conhecimentos e experiências que nos passaram.

A CPG – Comissão de Pós Graduação, pelo apoio e por dar condições para que pudéssemos ter um mestrado de alto nível em Marília.

Aos colegas, principalmente os do grupo de arquitetura de sistemas, pela amizade e força nos dias difíceis.

MUZZI, Fernando Augusto Garcia. **O PADRÃO DE SEGURANÇA PKCS#11 EM FPGAS: RSA UM ESTUDO DE CASO.** 2005., 143 f. Dissertação (Mestrado em Ciências da Computação) – UNIVEM Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

Com o aumento da velocidade da banda da rede, bem como o acesso à Internet existe um ganho de velocidade maior do que a evolução das CPUs, por esse motivo há necessidade dos processadores de redes (*também chamados de network processors – NP*). Existem poucos processadores de redes com a implementação de segurança, ou seja criptografia para garantir que um dado seja enviado com segurança pela rede. O PKCS#11 **(***Public Key Cryptography Standards)* – é a especificação que descreve a interface de programação chamada "Cryptoki" utilizada para operações criptográficas em hardwares: HSM (*Hardware Security Module), tokens,* cartões inteligentes. Esse padrão é popular e prevê suporte aos aplicativos e padrões. No entanto é necessário que os processadores de rede ofereçam segurança no processamento e envio do pacote na rede, dessa forma existe a necessidade da implementação de um módulo PKCS#11 para criptografia. Por esse motivo, o objetivo deste trabalho foi analisar o padrão PKCS#11 e implementar em FPGAs uma versão do módulo PKCS#11 com possibilidades futuras de integração a um processador de rede. O padrão PKCS#11 em software usa uma biblioteca desenvolvida em C, chamada cryptoki. Já em hardware não há como chamar essa biblioteca, por isso a especificação do PKCS#11 foi implementado obedecendo a seqüência exatamente como o padrão tem que ser. Usando um FPGA é possível abstrair o funcionamento do padrão e implementá-lo diretamente no hardware. Neste projeto implementou-se a especificação do PKCS#11 por meio de uma máquina de estados finito (FSM), formando o projeto modular e facilmente adaptável a expansões futuras para a comunicação entre máquina e dispositivos. Qualquer algoritmo de criptografia pode ser implementado dentro da especificação PKCS#11, e, neste projeto implementou-se o algoritmo RSA. Além da implementação da FSM foi possível realizar implementação do algoritmo RSA e testes usando o RSA com chaves de 56, 128, 256 e 512 bits.

Palavras-chave: Processador de rede. PKCS#11. Criptografia. FPGA. RSA.

MUZZI, Fernando Augusto Garcia. **O PADRÃO DE SEGURANÇA PKCS#11 EM FPGAS: RSA UM ESTUDO DE CASO.** 2005., 143 f. Dissertação (Mestrado em Ciências da Computação) – UNIVEM Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

With the network's speed increasing, as well as the access to the Internet, there is a greater increase in the velocity than the CPU's evolution, because of that; network processors are needed. There are a few network processors with embedded security system, that is, cryptography to ensure that a data is securely sent on the network. O PKCS#11 **(***Cryptographic Token Interface Standard)* is the specification that describes the programming interface called "Cryptoki" used in hardware cryptographic operations, such as HSM (*Hardware Security Module*), tokens and smart cards. So, it is necessary that the network processors offer security in the processing and sending of a package on the network, for using the cryptography operations from the PKCS#11. For that reason, the aim of this paper was to analyze the pattern PKCS#11 and implement it on FPGAs. The PKCS#11 standard in software uses a developed C library, called criptoki. There is not a way to use this library in hardware, so the PKCS#11 specifications were implemented obeying the exact sequence that the pattern requests. By using a FGPA it is possible to reduce the working part from the pattern and input it directly in the hardware. In this project, the PKCS#11 specifications were designed through a Finite State Machine (FSM), forming the modular project and easily adapting to future expansions for the communication between the machine and the devices. Any cryptography algorithm may be used in the PKCS#11 specification. In this project, the RSA algorithm was the target. In addition to the FSM design, we used inputs and keys using the RSA algorithm and our tests with keys 56, 128, 256 and 512 bits were possible to accomplish. So, we have designed a version of the PKCS#11 with future integration possibilities to a network processor.

Keywords: Network processor. PKCS#11, Cryptography, FPGA, RSA.

MUZZI, Fernando Augusto Garcia
O Padrão de Segurança PKCS#11 em FPGAs: RSA um Estudo de Caso / Fernando Augusto Garcia Muzzi; orientador: Edward David Moreno Ordonez.
Marília, SP: [s.n.], 2005.
143 f.

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

1. Segurança 2. Criptografia 3. FPGA.

CDD: 005.82

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AES - *Advanced Encryption Standard*

ANSI - *American National Standards Institute*

ASIC – *Application Specific Integrated Circuit*

ASN.1– *Abstract Syntax Notation one*

API - *Application Programming Interface*

ASCII - *American Standard Code For Information Interchange*

ASI - *Agere System Interface*

ASIC - *Application Specific Integrated Circuit*

ASIP - *Appplication Specific Instruction Set Processor*

ATM - *Assynchronous Transfer Mode*

CA - *Certificate Authentication*

CISC - *Complex Instruction Set Computing*

COS - *Class of Service*

CP - *Channel Processors*

CPU - *Central Processing Unit*

CSIX - *Common Switch Interface*

DES - *Data Encryption Standard*

DMA - *Acesso Direto a Memória*

DMU - *Data Mover Units*

DRAM - *Dynamic Random Access Memory*

DSL - *Digital Subscriber Line*

EDS - *Eletronic Data System*

EP - *Executive Processor*

FPGAs - *Field Programmable Gate Array*

FPP - *Fast Pattern Processor*

FP - *Fabric Processor*

FSM – *Finite State Machine*

GPP - *General Purpose Processor*

HDLC - *High Level Data Link Control*

HSM - *Hardware Security Module*

I/O - *Input / Outpu*

IOB - *In/Out Block*

IP - *Internet Protocol*

IPSec - *Internet Protocol* Security

IPv4 - *Internet Protocol V4*

IPv6 - *Internet Protocol V6*

IS0 - *lnternational Organization for Standardization*

ISDN - *Integrated Services Digital Network*

LAN - *Local Area Network*

TLU - *Table Lookup Unit*

LUTs - *look-up table* (estrutura interna de um FPGA)

MAC - M*essage Authentication Code*

MD2 - *Message Digest 2*

MD5 - *Message Digest 5*

MIT - *Massachsetts institute of Technology*

NAT - *Network Address Translation*

NPSIM - *Network Processor Simulation*

OSI - *Open System Interconnecting*

PCI - *Peripheral Component Interconnect*

PEM - P*rivacy-Enhanced Mail Protocol*

PKCS - *Public Key Cryptography Standards*

PKI - *Public Key Infraestructure*

PKIX - *Public Key Infrastructure X.500*

PIN - P*ersonal Identification Number*

PROM - *Programmable Read Only Memory*

QMU - *Queue Management Unit*

QoS - *Quality of Service*

R2NP - *Reconfigurable RISC Network Processor*

RCNP - *Reconfigurable CISC Network Processor*

SAN - *Storage Area Network*

RCNP – *Reconfigurable CISC Network Processor*

RISC - *Reduced Instruction Set Computing*

RSA - *Rivest, Shamir e Adleman*

RSP - *Routing Switch Processor*

R2NP – *Reconfigurable RISC Network Processor*

S/MIME- *Secure/Multipurpose Internet Mail Extensions*

SDRAM - *Synchronous Dynamic Random Access Memory*

Sha-1 - *Secure Hash Algorithm*

SME- *Small Medium Enterprise*

SOC -*System on Chip*

SRAM - Static *Random Access Memory*

SSL - *Secure Socket Layer*

TCP - *Transmission Control Protocol*

TOPcore - *Top Optimized Processing Core*

UART - *Universal Asynchronous Receive and Transmit*

ULAs - *Unidades Lógicas e Aritméticas*

USB - *Universal Serial Bus*

USC - *University of Southern Califórnia*

VHDL – *VHSIC Hardware Description Language*

WLL*- Wireless Local Loop*

SUMÁRIO

CAPITULO 1

1. INTRODUÇÃO

1.1 Processadores de Rede e padrão PKCS#11

A Internet está no dia a dia das pessoas, empresas e instituições de ensino. Seu uso tem acarretado um congestionamento dos enlaces de transmissão. Os principais responsáveis são as pontes ("roteadores"), os quais confinam o tráfego entre redes.

Os processadores de rede estão sendo objeto de estudo nas universidades e empresas. Estes processadores surgiram para melhorar a qualidade de serviços (Q*oS: Quality of Services*) [TANEMBAUM 1999].

O tempo de processamento dos pacotes de rede é mais rápido usando os processadores de rede [TANEMBAUM 1999]. É necessário aumentar a capacidade de processamento dos pacotes para que não haja congestionamento e excessiva utilização da largura de banda.

Os processadores geralmente foram baseados em propósito geral e sua arquitetura era parecida com a dos computadores pessoais. Ao invés disso, os processadores de rede foram projetados utilizando modelos de arquiteturas [PATTERSON 1997], sendo o modelo mais utilizado o ASIP (*Application Specific Instruction Set Processor*) e o SoC (*System-on-Chip*) para agregar técnicas de projeto RISC (*Reduced Instruction Set Computing*) e ter um desempenho computacional maior.

Algumas empresas já desenvolveram processadores de redes, entre elas pode-se citar a Lucent e Agere [AGERE 2001], MOTOROLA [MOTOROLA 1999], INTEL [INTEL 2000], EZCHIP [EZCHIP 2002], IBM [IBM 2002].

Os processadores de rede são parte de uma classe emergente de circuitos integrados programáveis baseados na tecnologia SoC, também conhecidos como *Network Processors* (NP).

A expectativa em relação aos processadores de rede, é que os mesmos se tornem o principal componente para redes, da mesma forma que as CPUs são importantes para os PCs.

Os processadores de rede têm fundamental importância para a comunicação de dados, já que a velocidade de transmissão tem aumentado consideravelmente. Assim, o assunto escolhido é importante para a área da computação e contribuirá para o estudo aprofundado dos processadores de rede.

Ainda existe falta de segurança na transmissão de pacotes na rede, por isso é necessário a criação de algum mecanismo de segurança para rede, no caso o processador de rede, um processador com criptografia é fundamental para a transferência segura de pacotes na rede [CROWLEY 2000].

Existem diversas soluções de segurança baseadas em criptografia, porém a maior parte dos estudos se concentram em processadores de rede. Soma-se a isso que há soluções já padronizadas (tais como o *Public Key Cryptography Standards*), porém não são relacionadas com os NPs nem com implementações em hardware.

A segurança da informação é importante para que os pacotes não possam ser lidos por pessoas não autorizadas. Assim, torna-se necessária a criação de mecanismos que possam impedir que o conteúdo do pacote seja divulgado a entidades não autorizadas. Existem padrões e especificações de criptografia que podem ser implementadas em processadores de rede.

1.2 Justificativa e importância do projeto

O grande gargalo nas comunicações tem sido os equipamentos de rede, principalmente os roteadores [CROWLEY 2000]. Eles centralizam o tráfego de pacotes e muitas vezes são os responsáveis pela falta de eficiência da rede. As tecnologias de rede têm evoluído, principalmente com os avanços da internet, e os roteadores precisam aumentar sua capacidade de processamento dos pacotes, para evoluir juntamente com as demais tecnologias e deixar, aos poucos de ser um dos gargalos das comunicações.

Os processadores de rede têm fundamental importância para a comunicação de dados, já que a velocidade de transmissão de dados têm aumentado consideravelmente [CROWLEY 2000]. No entanto é necessário segurança para os processadores de rede, ou seja, a criptografia de pacotes, para garantir a segurança da informação. A

implementação de um algoritmo de criptografia baseado na norma PKCS#11 é importante para segurança dos dados (pacotes que tráfegam na rede), que poderá ser implementado em um FPGA. Com a implementação em hardware das especificações do PKCS#11 que propiciará a criptografia, um processador de rede poderá receber e transmitir pacotes criptografados.

O padrão PKCS#11 é utilizado como interface para invocar operações criptográficas em hardware e é utilizado para prover suporte aos tokens.

Os processadores de rede precisam de um módulo de segurança para o processamento de pacote, para prover maior segurança na rede. Com um módulo de segurança baseado no padrão, o dado poderá ser criptografado no processador de rede e quando o dado chegar no destino será decifrado conforme indicado pelo padrão.

Importante salientar, que o PKCS#11 é um padrão criado para interagir com hardware podendo ser utilizado em rede, baseado em segurança de dados que utilizam como base o algoritmo de criptografia RSA.

Após estudar a literatura existente na comunidade, percebe-se que no Brasil não existem processadores de rede em FPGA e os processadores de rede comerciais que existem, poucos consideram a segurança como fato relevante de projeto.

1.3 Objetivos da Dissertação

Esta dissertação de mestrado tem como objetivo principal o desenvolvimento e análise de um protótipo em FPGA do padrão PKCS#11, permitindo com que pacotes de dados sejam cifrados e decifrados.

O projeto do módulo PKCS#11 foi realizado em VHDL e implementou-se um protótipo em FPGA. Quando o pacote for processado por algum processador, esse pacote será criptografado seguindo a norma PKCS#11 e enviado através da rede para o destino, e quando o pacote chegar no destino o mesmo será decifrado conforme a chave determinada de criptografia seguindo as operações recomendadas pelo padrão. Dessa forma, se o pacote for capturado na rede antes de chegar ao destino, o mesmo não poderá ser lido devido à implementação da criptografia.

1.4 Metodologia

A partir do momento em que foi efetuada a modelagem do módulo PKCS#11 por meio da linguagem de descrição de hardware VHDL, resultados relacionados a viabilidade foram visíveis, como velocidade de transmissão, velocidade de criptografia usando o RSA, temos portanto, um módulo que faz a criptografia dos dados. Com esse módulo é possível fazer simulações e síntese utilizando-se dos ambientes de projeto da Xilinx. O uso de dispositivos programáveis FPGA possibilitou uma análise dos resultados experimentais sobre o funcionamento do módulo que cifra e decifra.

Os testes e validações foram realizados baseados na ferramenta Xilinx para viabilizar o módulo PKCS#11. Depois de se ter o módulo pronto, uma simulação do funcionamento foi realizada para concluir os testes e validar finalmente o módulo PKCS#11 em FPGA que poderá ser usado para criptografia em um processador de rede.

1.5 Organização da Dissertação

Esta dissertação está organizada em 7 capítulos, descritos a seguir:

O capítulo 1 apresenta a introdução, a importância, justificativa e objetivos, assim como a metodologia usada no projeto.

O capítulo 2 apresenta um estudo de processadores de redes, focalizando os tipos de processadores de redes existentes na comunidade, em especial os projetos brasileiros.

O capítulo 3 apresenta os conceitos de criptografia e os princípios do padrão PKCS#11, enfatizando principalmente a definição de segurança e o módulo PKCS#11.

O capítulo 4 apresenta a implementação em VHDL e FPGA do Módulo PKCS#11, enfatizando os resultados obtidos com a implementação.

O capítulo 5 discute algumas otimizações realizadas na implementação do padrão PKCS#11 em FPGAs.

O capítulo 6 apresenta as conclusões, enfatizando os resultados finais do projeto.

O capítulo 7 apresenta as referências bibliográficas.

CAPÍTULO 2
HARDWARE CRIPTOGRÁFICO E DE SEGURANÇA

Neste capítulo se apresentam os conceitos relacionados com processadores de redes (*network processor*), alguns exemplos de NPs comerciais e os processadores de rede comerciais com segurança.

2.1 – NP (*Network Processors) – Processadores de Rede*

O uso da Internet em grande escala e o roteamento de pacotes e o surgimento de novos protocolos e diferentes tipos de redes de dados, têm aumentado o custo para aquisição de novos equipamentos e também o custo de atualização.

Na Figura 2.1 há a comparação entre a velocidade da rede e das CPUs (*Central Processing Unit)* , onde se observa um aumento considerável na velocidade da Internet comparando com a velocidade das CPUs.

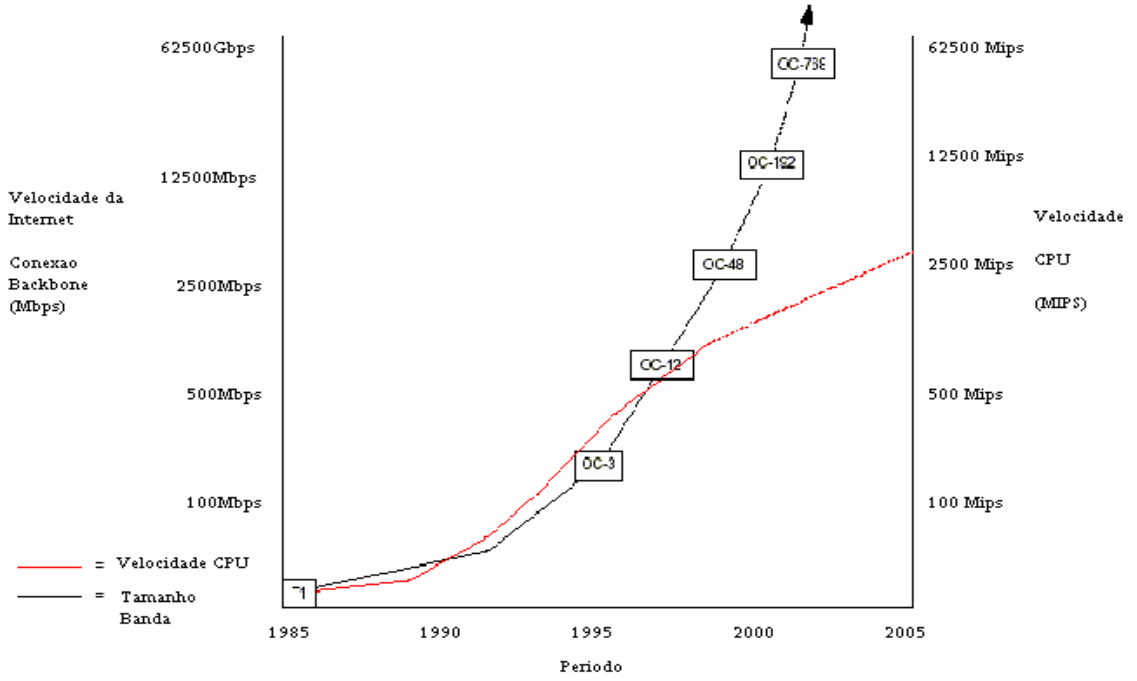

Figura 2.1 Velocidade da Internet x velocidade das CPUs [SHAH 2000]

Processadores de rede são parte de uma classe emergente de circuitos integrados programáveis baseados na tecnologia SOC, classificação essa por possuir vários blocos

como memória, portas e interface de comunicação que executam funções específicas de comunicação de forma mais eficiente que os GPPs (*General Purpose Processor*) [CROWLEY 2000].

Há uma tendência em relação aos processadores de rede, para que os mesmos se tornem o principal componente para redes, da mesma forma que as CPUs são importantes para os PCs. Com o processador de rede será possível aumentar a eficiência e diminuir o gargalo existente na rede hoje [CROWLEY 2000].

Com o avanço da tecnologia, algumas empresas desenvolveram processadores de rede dedicados, capazes de serem mais velozes e com maior desempenho do que processadores de propósito geral. Esses processadores vieram para reduzir o gargalo das comunicações causado muitas vezes pelos roteadores.

As características típicas oferecidas por um NP são: o processamento em tempo real, manipulação de pacotes IP devido à camada de rede no modelo OSI (*Open System Interconnecting)* ser equivalente à camada inter-rede e executar o protocolo IP [TANEMBAUM 1999]. Nessa camada também se encontram os roteadores, outra aplicação dos NPs, que por sua flexibilidade e escalabilidade, permitem a troca de pacotes de tamanhos diferentes em redes com diferentes protocolos e topologias.

2.2 Modelo OSI

Os alvos dos NPs são as camadas de 2 a 7 do modelo OSI e são projetados para aperfeiçoar tarefas específicas de rede, podendo variar das camadas 2 a 4, 2 a 5 e 2 a 7, habilitando o NP a ser usado em um amplo espectro de aplicações de rede, para isto depende do foco inicial do projeto e dos serviços para o qual é desenvolvido.

O modelo OSI foi criado em 1977 pela IS0 (*International Organization for Standardization*) com o objetivo de criar padrões de conectividade para interligar sistemas de computadores locais e remotos. Os aspectos gerais da rede estão divididos em 7 camadas funcionais, facilitando assim a compreensão de questões fundamentais sobre a rede, apesar de tal modelo não ter sido adotado para fins comerciais.

A Tabela 2.1 mostra o modelo ISO/OSI e a atuação dos produtos de comunicação em cada uma das camadas desse modelo. O modelo ISO/OSI tem uma divisão muito clara das camadas de um sistema de comunicação. Este é um grande

auxílio para o entendimento dos diversos protocolos de mercado [TANENBAUM 1999].

Tabela 2.1 Modelo ISO/OSI [TANEMBAUM 1999].

Número	Nome	Função
Camada 7	Aplicação	É representada pelo usuário final no modelo OSI, selecionando serviços a serem fornecidos pelas camadas inferiores, entre eles, o correio eletrônico, transferência de arquivos, etc.
Camada 6	Apresentação	Transfere informações de um software de aplicação da camada de sessão para o sistema operacional. Criptografia, conversão entre caracteres ASCII e EBCDIC, compressão e descompressão de dados são algumas funções acumuladas nesta camada.
Camada 5	Sessão	Reconhece os nós da rede local LAN e configura a tabela de endereçamento entre fonte e destino, isto é, estabelece as sessões, no qual o usuário poderá acessar outras máquinas da rede.
Camada 4	Transporte	Controla a transferência de dados e transmissões. Protocolos de transporte (TCP) são utilizados nesta camada.
Camada 3	Rede	conexão lógica entre dois pontos, cuidando do tráfego e roteamentos dos dados da rede.
Camada 2	Enlace	Acesso lógico ao ambiente físico, transmissão e reconhecimento de erros
Camada 1	Física	Aspectos mecânicos, elétricos e físicos

As camadas a serem implementadas nos processadores de rede dependem especificamente da natureza da aplicação para a qual se deseja criar soluções, como, por exemplo, um roteador, que é um dispositivo que conecta duas LANs diferentes, roteando os pacotes entre elas, este dispositivo é operado nas camadas física, de enlace de dados e de rede.

2.3 Comparação entre Processadores

As tecnologias usadas para o projeto de processadores são: CISC (*Complex Instruction Set Computing*) e RISC (*Reduced Instruction Set Computing*).

Existem diversos fornecedores de soluções de rede que estão voltando os seus projetos para soluções cada vez mais flexíveis e com um custo baixo. Esses desenvolvedores estão combinando a flexibilidade que existe nos processadores de propósito geral e a velocidade dos ASICs como pode ser visto na Figura 2.2.

Essa solução tem atraído a atenção de grandes empresas como Intel, Motorola, IBM entre outras.

Encontra-se no mercado circuitos inteligentes para gerenciamento de rede, de alta flexibilidade e velocidade, proporcionando adequação aos critérios já estabelecidos, evitando todo o processo e custo do desenvolvimento de um ASIC.

Inicialmente utilizavam-se processadores de propósito geral (GPP's: *General-purpose Processor*) [PATTERSON 1997], depois circuitos dedicados (ASIC's: *Application Specific Integrated Circuit*) [KÄSTNER 2003] e por fim processadores específicos (ASIP's) [KÄSTNER 2003] de rede. As gerações de equipamentos e suas principais características podem ser definidas da seguinte forma:

- ***1ª Geração***
  - Processador de Propósito Geral (GPP): Baseado na utilização de softwares para definição de comportamento. Processador flexível para diversas aplicações, porém para aplicações de rede são limitados em funcionalidades e desempenho.
- ***2ª Geração***
  - Circuito de aplicação específica (ASIC): Circuitos projetados para executar determinadas funções específicas. Não utilizam softwares, são bastante rápidos e o grande problema é a flexibilidade, uma vez que o ASIC é projetado apenas para uma aplicação.
- ***3ª Geração***
  - Processadores de Rede: Os Processadores de Rede possuem microarquitetura dedicada para trabalhar em redes de comunicação de dados. O conjunto de instruções também é específico. Por se tratar de processadores, eles são flexíveis e por serem específicos também possuem bom desempenho tal como um ASIC.

Sendo assim, podemos considerar um processador de rede como um processador de aplicação específica (ASIP: *Application Specific Instruction Set Processor*).

Vantagens: União de flexibilidade e desempenho devido, por exemplo, ao projeto dedicado do conjunto de instruções e da arquitetura.

Desvantagens: É mais lento do que um ASIC e menos flexível do que um GPP.

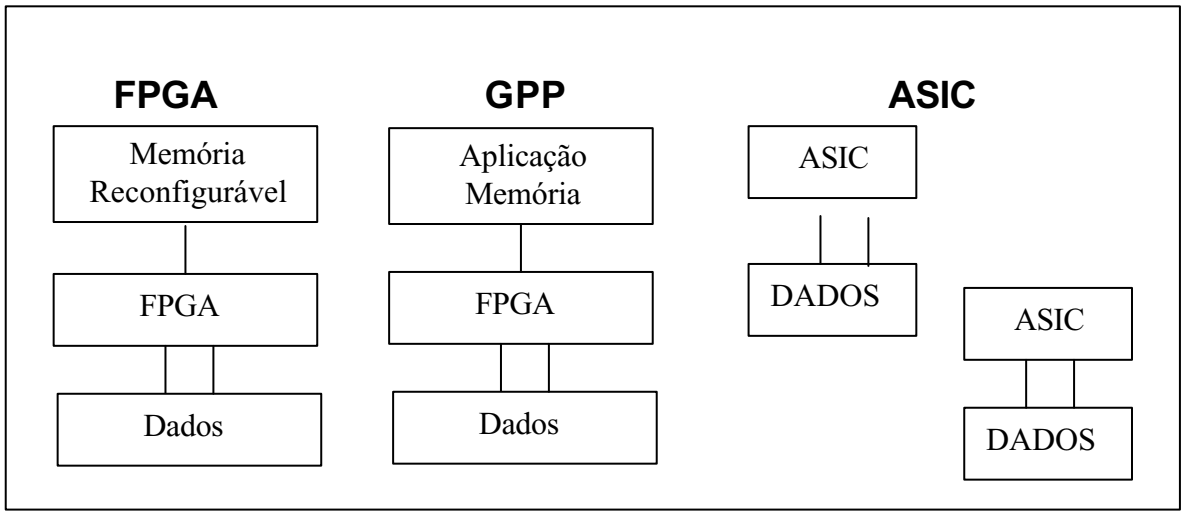

Figura 2.2 Processadores FPGA, GPP e ASIC. [VILLASENOR 1997]

A Tabela 2.2 mostra as possibilidades de aplicações específicas de processamento de pacotes, que podem ser inseridas em modernos e futuros processadores de rede.

Tabela 2.2. Representação de aplicações específicas de processamento de pacotes [CROWLEY 2000]

Aplicação	**Especificação**
Classificação de pacotes e filtragem	Decisões, envio, estatísticas, proteção por *firewall*.
Reenvio de Pacotes IP	Envio de pacotes IPs baseado em informação de roteamento
Network Address Translation (NAT)	Tradução entre roteamento global e pacotes IP privados, mascaramento de IP, servidores web, etc.
Administração de fluxo	Reduzir congestionamento, gargalos, forçar a distribuição da banda (pacotes).

TCP/IP	Descarregar processamento TCP/IP dos webservers para as interfaces da rede
Web Switching	Balanceamento de serviços web e monitoramento do cache do proxy.
Virtual Private Network IP Security (IPSec)	Criptografia (3DES) e autenticação (MD5).
Transcodificação de dados	Conversão de dados de multimídia por demanda de um formato para outro dentro da rede
Supressão de dados duplicados	Reduz transmissão redundante de dados que gera custo alto na transmissão da rede.

2.4 Processadores de Rede Brasileiros

Os processadores de Rede Brasileiros são o RCNP (*Reconfigurable CISC Network Processor)* – Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas [FREITAS 2001] desenvolvido usando tecnologia CISC, e o R2NP (*Reconfigurable RISC Network Processor*) - Processador de Rede RISC Reconfigurável que na verdade é um SoC, pois é composto por vários blocos lógicos.

Os criadores do RCNP e do R2NP usam o NPSIM (*Network Processor Simulator)* para analisar o desempenho das suas propostas.

O NPSIM [MEDEIROS 2002] é um simulador de rede, capaz de realizar testes funcionais dos diversos blocos do processador. Por meio deste simulador é possível escrever e executar diversos algoritmos, escritos em *assembly*, e visualizar a execução e os resultados por meio dos registradores, pilhas e matrizes representados pelos diversos componentes existentes no compilador C++ Builder 5.0, utilizado para construir e compilar o simulador.

Por meio deste simulador também é possível estudar o comportamento de um processador de rede CISC, escrever algoritmos de roteamento e oferecer aos desenvolvedores a possibilidade de escrever compiladores para o processador virtual (simulador) de rede, tal como uma máquina virtual.

2.4.1 RCNP - Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas

No RCNP – (*Reconfigurable CISC Network Processor)* - Processador de Rede com suporte a multi-protocolo e topologias dinâmicas [FREITAS 2001], a sugestão dos autores foi que seria desenvolvido um processador usando tecnologia CISC, com barramento de endereço de 24 bits, e barramento de dados de 8 bits, uma memória de 16 Mb, 8 portas de entrada com *buffers* reconfiguráveis, o que pode resolver o problema de tamanhos diferentes de pacotes, 8 portas de saída que são ligadas às portas de entrada por um seletor de conexão do tipo *Crossbar*, reconfigurável que permitiria topologias diferentes.

Os blocos funcionais que são responsáveis pela caracterização do RCNP como processador de rede, são conhecidos como interfaces de comunicação. Na Figura 2.3 os blocos do conjunto são dedicados para o recebimento e tratamento dos pacotes. Estão presentes neste conjunto as portas de entrada e saída, a chave *crossbar*, os *buffers* de entrada (*temporários e permanentes*) e o registrador de estados (guarda saída livre ou ocupada). A unidade PCI é responsável pela interface com processadores de propósito geral e a unidade dedicada é responsável pela comunicação entre processadores de tipo RCNP.

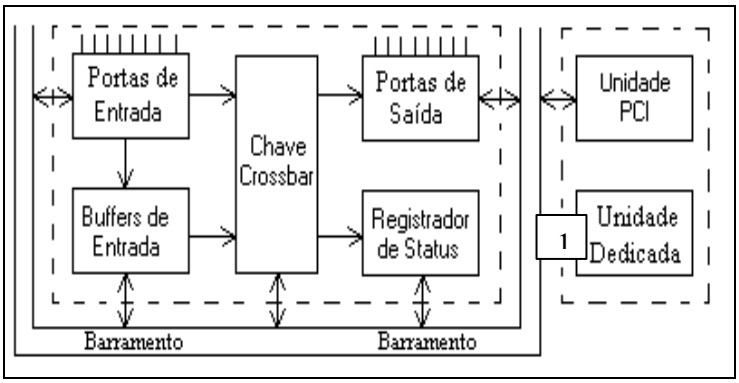

Figura 2.3 – Unidades de comunicação do processador de Rede [FREITAS 2001]

O principal motivo para o desenvolvimento dos processadores de rede é o desempenho [PATTERSON 1997]. A crescente necessidade por velocidade e qualidade

de serviço, principalmente na Internet, tem impulsionado as pesquisas na área. Por meio do NPSIM (*Network Processor Simulator)*, um simulador existente para o RCNP [FREITAS, 2002], foi possível obter via simulações, resultados quantitativos que demonstram a melhoria de processamento quando se tem uma microarquitetura e conjunto de instruções dedicadas.

O padrão de comunicação entre os equipamentos de uma rede é definido pelo protocolo [TANEMBAUM 1999]. Desta forma a troca de informações é estabelecida por meio de programas previamente desenvolvidos que são capazes de interpretar corretamente os dados encontrados em cada posição do pacote.

As topologias são representações físicas da rede, como os nós (*computadores, processadores ou dispositivos*) estão distribuídos.

Por meio do *Assembler* é possível escrever, editar e carregar na memória o programa desenvolvido para ser executado pelo NPSIM. Os autores do RCNP [FREITAS 2002] analisaram três tipos de topologias, anel unidirecional, hipercubo e árvore balanceada.

2.4.2 R2NP - Processador de Rede RISC Reconfigurável

O R2NP – (*Reconfigurable RISC Network Processor*) – Processador de Rede RISC Reconfigurável, é na verdade um SoC, pois é composto por vários blocos lógicos que na maioria das vezes são externos e trabalham em conjunto com os processadores como se pode ver na Figura 2.4. A utilização de *microengines*, e multiplexador no lugar de *buffers* de entrada temporários e estáticos se deve ao fato de cada *microengine* ser um hardware dedicado que tem a função de ler um espaço de memória, onde estão guardados dados referentes ao protocolo, para tomar decisões preliminares de roteamento, redirecionando os pacotes para os *buffers* reconfiguráveis ou para a saída, através da chave *crossbar*.

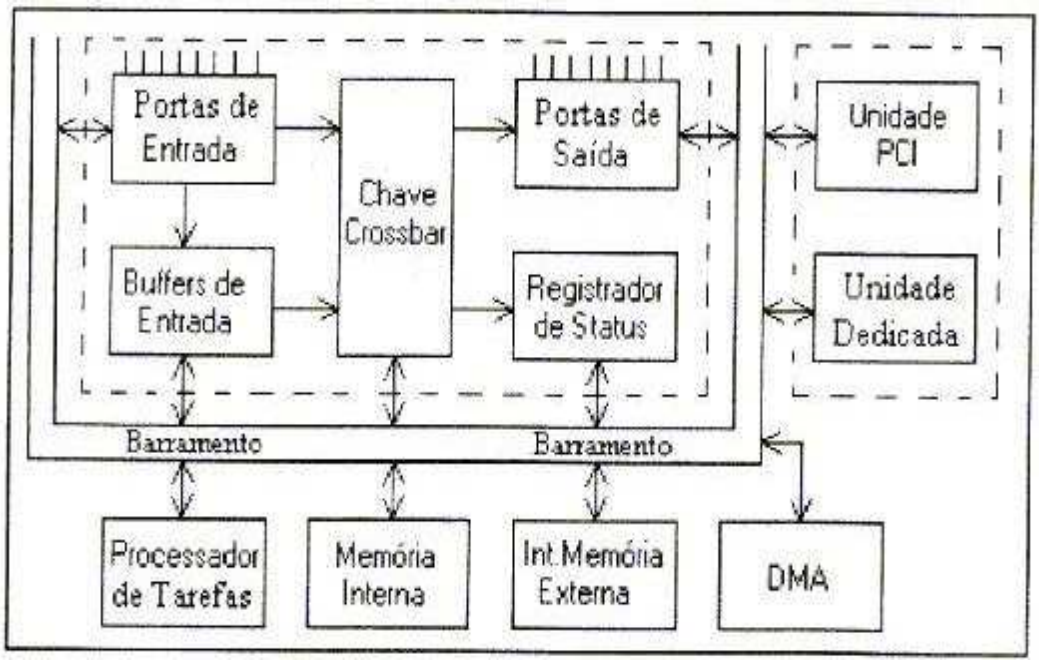

Figura 2.4. Arquitetura do R2NP [FREITAS, 2002]

O R2NP (*Processador de Rede RISC Reconfigurável)* [FREITAS 2002] é uma , versão RISC do RCNP. Este processador aproveita as vantagens da microarquitetura de rede otimizada com as vantagens do modelo RISC.

A Tabela 2.3 mostra que o número de portas de comunicação, buffers reconfiguráveis e seletor de conexão, que dão suporte a topologias dinâmicas, são características dos processadores experimentais que ainda não foram implementados em processadores comerciais. Nesta tabela aparecem informações sobre dois processadores de rede comerciais, o IXP 1200 da Intel e o PC860 da Motorola, que serão melhor descritos na sessão 2.5.

Tabela 2.3. Comparação entre processadores de rede [FREITAS 2002]

	IXP 1200 Intel	**Motorola PC860**	**RCNP**	**R2NP**
Processador	RISC	RISC	CISC	RISC
Barramento de dados	32 bits / 64 bits	8, 16 e 32 bits	8 bits	32 bits

Barramento de endereços	32 bits / 64 bits	32 bits	24 bits	32 bits
Portas seriais de comunicação	1 entrada / 1 saída	6 entradas / 6 saídas	8 entradas / 8 saídas	8 entradas / 8 saídas
Interface de comunicação	SIM	SIM	SIM	SIM
Possui cache de instrução	SIM	SIM	NÃO	NÃO
Possui cache de dados	SIM	SIM	NÃO	NÃO
Possui memória interna	SIM	SIM	SIM	SIM
Possui buffers reconfiguráveis	NÃO	NÃO	SIM	SIM
Suporta várias topologias	NÃO	NÃO	SIM	SIM
Suporta vários protocolos	SIM	SIM	SIM	SIM

2.4.3 NPSoC – Processador de Rede em FPGA.

O Processador proposto NPSoc é baseado no RCNP Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas [FREITAS 2000] [FREITAS 2001] e no R2NP (*Reconfigurable RISC Network Processor*) – Processador de Rede RISC Reconfigurável. Estes processadores propostos não foram implementados ainda, somente um módulo, o do CrossBar, foi simulado na ferramenta Xilinx. No processador, foi definido um conjunto de instruções simples, porém que dá cobertura ao funcionamento do processador, como seus registradores, ULA (Unidade Lógica e Aritmética), a UC (Unidade de Controle), PC (Contador de Programa), RI (Registrador de Instruções) e a maioria das instruções de roteamento, necessárias para a execução de alguns algoritmos propostos para a simulação do funcionamento de algumas redes [PRADO 2004].

Na Figura 2.5, fica claro que o NPSoC é um processador com módulos especiais, INSTRUÇÕES DE ROTEAMENTO, porém com todos os componentes necessários para o funcionamento de qualquer processador, como contador de programa, UC, ULA, registrador de instruções e etc.

Portanto o processador depende de uma Unidade de Controle como todo processador, para realizar suas operações de controle, o que especificamente neste processador, se resolveu fazer utilizando o método de FSM (Finite State Machine), ou seja, máquina de estados finita, onde seus estados são muito bem definidos e respeitam o ciclo de busca, decodificação e execução dos computadores Von Neumann, onde o ciclo se executa simplificadamente desta maneira:

Passo 1: É feita a busca da 1ª instrução;

Passo 2: Soma-se 1 para o contador de programa, ou seja, atualiza o contador;

Passo 3: Decodifica a instrução, através de seus opcodes;

Passo 4: Se essa instrução trouxer dados, serão armazenados em registradores internos;

Passo 5: Executa a instrução, normalmente isso é feito pela ULA;

Passo 6: Registra os resultados em local apropriado: memória, buffer e etc; Retornando ao passo 1, ou seja, irá buscar a próxima instrução, dando continuidade ao ciclo.

A comunicação com esse local apropriado, é feita por meio de um barramento cuja função é levar os dados da ULA para a memória, instruções para o RI, realimentar os registradores da ULA e etc. Para isso foi necessária a implementação de uma memória (uma pilha) de 512 Kbytes, como se pode observar na Figura 2.5.

No NPSoC, além das instruções comuns em qualquer processador, como instruções aritméticas e lógicas, de manipulação e de desvio, temos a implementação de um módulo especial com instruções específicas de roteamento.

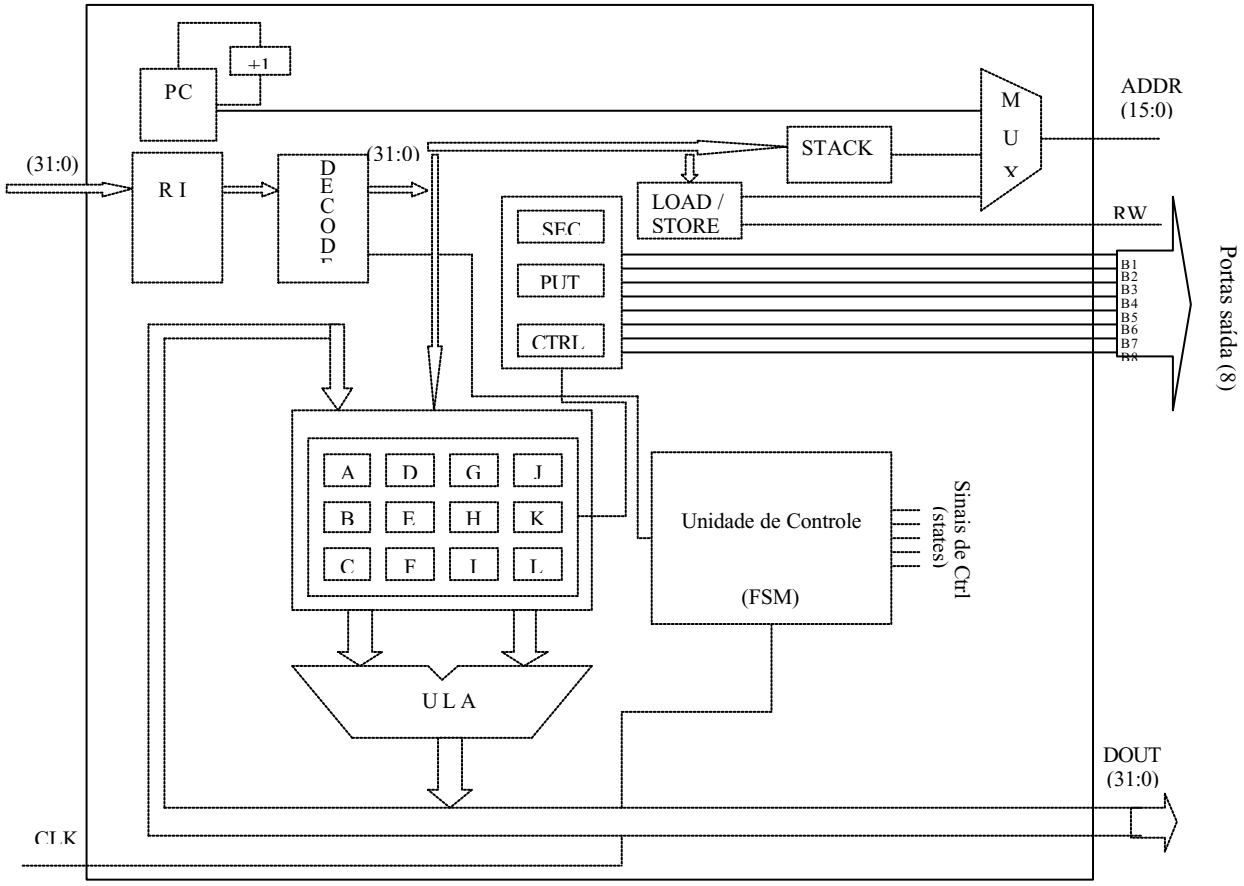

Figura 2.5 - Arquitetura detalhada do NPSoC [PRADO 2004].

2.5 Processadores de Rede Comerciais

2.5.1 A família Motorola (PowerQUICC)

Nesta sessão é apresentado os processadores de redes comerciais existentes e a arquitetura de cada um.

Neste tópico é apresentado o processador de comunicação de dados MPC860 da família PowerQUICC [C-PORT 2002]. As aplicações principais deste processador são: roteadores e *switches* de LAN/WAN, dispositivos de integração de redes, concentradores de rede, *gateways*, DSL/*cable modems* e sistemas de voz sobre IP.

O MPC860 possui um barramento auto-ajustável de 8, 16 e 32 bits. Na Figura 2.6 observa-se que existem três blocos principais, que são descritos com mais detalhes a seguir:

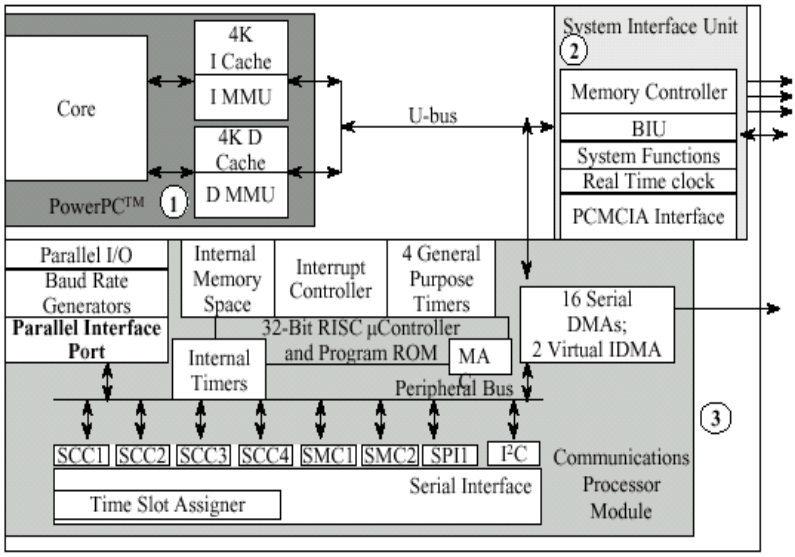

Figura 2.6. – Arquitetura do MPC860 [MOTOROLA 1999]

O bloco 1 chamado de *PowerPC core,* possui o processador principal de 32 bits (*PowerPC*) responsável pela execução dos códigos das camadas mais altas, com o intuito de aumentar o *throughput* (*vazão*). Possui *cache* de instrução e de dados de 4kbytes e duas unidades de gerenciamento de memória, relacionada cada uma com seu respectivo *cache*.

O bloco 2, chamado de *System Interface Unit, é* responsável pela interface do barramento interno com os barramentos externos. Possui uma controladora de memória, um barramento de interface, uma unidade de funções do sistema, o *clock* e uma unidade de interface PCMCIA.

O bloco 3 chamado de *Communications Processor Module* é o módulo de comunicação de dados. Através deste módulo é possível enviar e receber dados de diferentes tipos de dispositivos por meio de quatro canais seriais de comunicação (*SCC*)

e dois de gerenciamento (*SMC*). Os dispositivos conectados podem ser usados de forma individual ou, através destes canais, usar um barramento multiplexador por divisão de tempo.

Este módulo possui um micro-controlador de 32 bits e 16 canais seriais de DMA (*Acesso Direto à Memória),* oito canais para transmissão e oito para recepção. Os canais de DMA são de uso exclusivo do micro-controlador e são usados para obter e mover dados para a memória.

Alguns dos protocolos suportados por este módulo são: *Ethernet*, *Fast Ethernet*, ATM, HDLC, *Appletalk*, UART (*Universal Asynchronous Receive and Transmit*) e ISDN.

A comunicação entre os dois processadores é feita através da memória interna. Através desta memória cada processador pode escrever bits de controle e ler bits de status, possibilitando a requisição e resposta dos dados.

Neste processador, o processamento "rápido" é feito pelo micro-controlador e o processamento "lento" é feito pelo PowerPC.

2.5.2 A família Motorola (C-Port e C5)

Esta família de processadores foi projetada pela C-Port / Motorola [C-PORT 2002] para aplicações de rede mais ligadas à qualidade de serviço (QoS). Portanto, as duas famílias de processadores da Motorola se complementam, um estabelece a comunicação com mais consistência e o outro prioriza os critérios de (QoS), exigência crescente entre as aplicações e serviços de rede.

Os principais blocos do C-5 NP, são mostrados na Figura 2.7, onde é possível observar os blocos CP (*Channel Processors),* EP *(Executive Processor),* FP *(Fabric Processor),* BMU *(Buffers Management Unit),* TLU *(Table Lookup Unit),* QMU *(Queue Management Unit).*

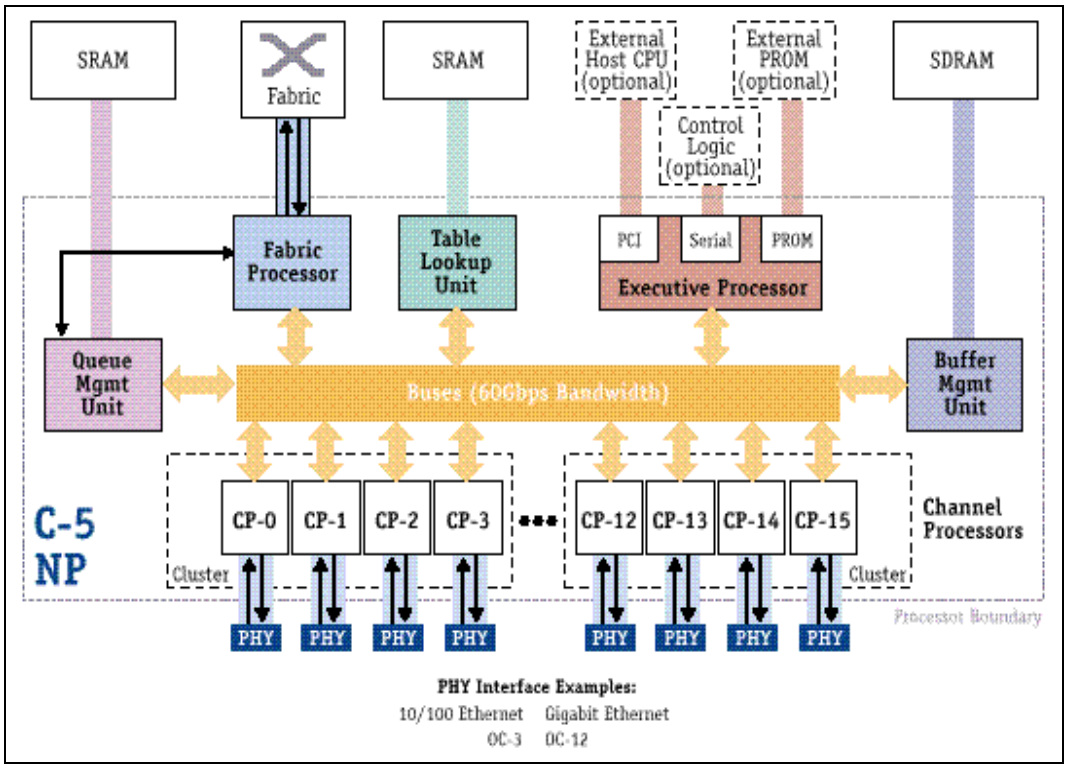

Figura 2.7 – Arquitetura do C-5 NP [C-PORT 2001]

O C-5 NP possui dezesseis processadores de canal que recebem, processam e transmitem os dados. O número de processadores por porta é configurável, dependendo do tipo de interface. Inicialmente um processador é alocado para aplicações onde a largura de banda é média, porém quando há a necessidade de utilização de uma interface de alta velocidade, múltiplos processadores podem ser alocados por porta. Múltiplas portas podem alocar apenas um processador, neste caso é utilizado um multiplexador externo e o tipo de aplicação é para aplicação de banda baixa.

Esta arquitetura suporta protocolos seriais e paralelos: 10Mb *Ethernet* (*RMII*), 100Mb *Ethernet* (*RMII*), 1Gb *Ethernet* (*RMII e TBI*) ,OC-3, OC-12, DS1/DS3, através de multiplexadores externos.

Neste módulo de processador de rede se destaca o EP (*Executive Processor*) que é o ponto principal de processamento. Ele é responsável por gerenciar as interfaces e suas funções são: resetar e iniciar o C-5 NP, carregar o programa que controla os CPs, controla as exceções, gerenciar a interface *host* pela PCI, gerenciar a interface PCI e o barramento e a memória PROM.

O sistema de interface é composto por PCI que é utilizada para se conectar com outros processadores (*host*). SBI: Interface bidirecional de propósito geral, PROM: Memória flash usada para boot. Interface de baixa velocidade. Tamanho máximo de 8Mbytes.

Também se destaca o FP (*Fabric Processor*), que atua como uma interface de rede de alta velocidade. Ele suporta transferência bi-direcional de pacotes, *frames* ou células do C-5 NP para as interfaces de hardware que provêem conectividade com outros processadores de rede ou hardwares de processamento similar.

As Unidades de Gerenciamento de Buffers, conhecidos como BMU - *Buffers Management Unit são* responsáveis pela interface do C-5 NP com a arquitetura de *pipeline* externo, *Single Data Rate Synchronous* DRAM. A memória externa é particionada e usada como *buffers* para receber e transmitir dados entre CPs, FP e EP. Ela está no segundo nível de hierarquia de memória do *Executive Processor*.

A TLU (Unidade Localização de Tabela - *Table Lookup Unit) é* responsável pela busca e a atualização em tabela e serviços associados para os processadores CP, EP e FP. Possui também interface para memória externa.

A unidade QMU (Unidade de Gerenciamento de Fila - *Queue Management Unit), g*erencia o número de aplicações definidas nos descritores de fila. Esta unidade possui um chip interno responsável por guardar os descritores na memória SRAM externa.

2.5.3. O IXP1200 da Intel

Este processador de rede é composto por sete processadores RISC (ver detalhes na Figura 2.8). O primeiro processador é chamado de StrongARM e é responsável pelo processamento mais complexo, tal como construção e manutenção de tabelas e gerenciamento da rede. Os outros seis processadores são chamados de *microengines*. Eles trabalham com multi-processamento e *multi-thread* e são responsáveis pelo processamento e re-direcionamento dos pacotes. Com base nestes dados iniciais, já é

possível definir que, a classe de processamento “rápido” está relacionada com as *microengines* e a classe de processamento “lento” está relacionada com o processador StrongARM.

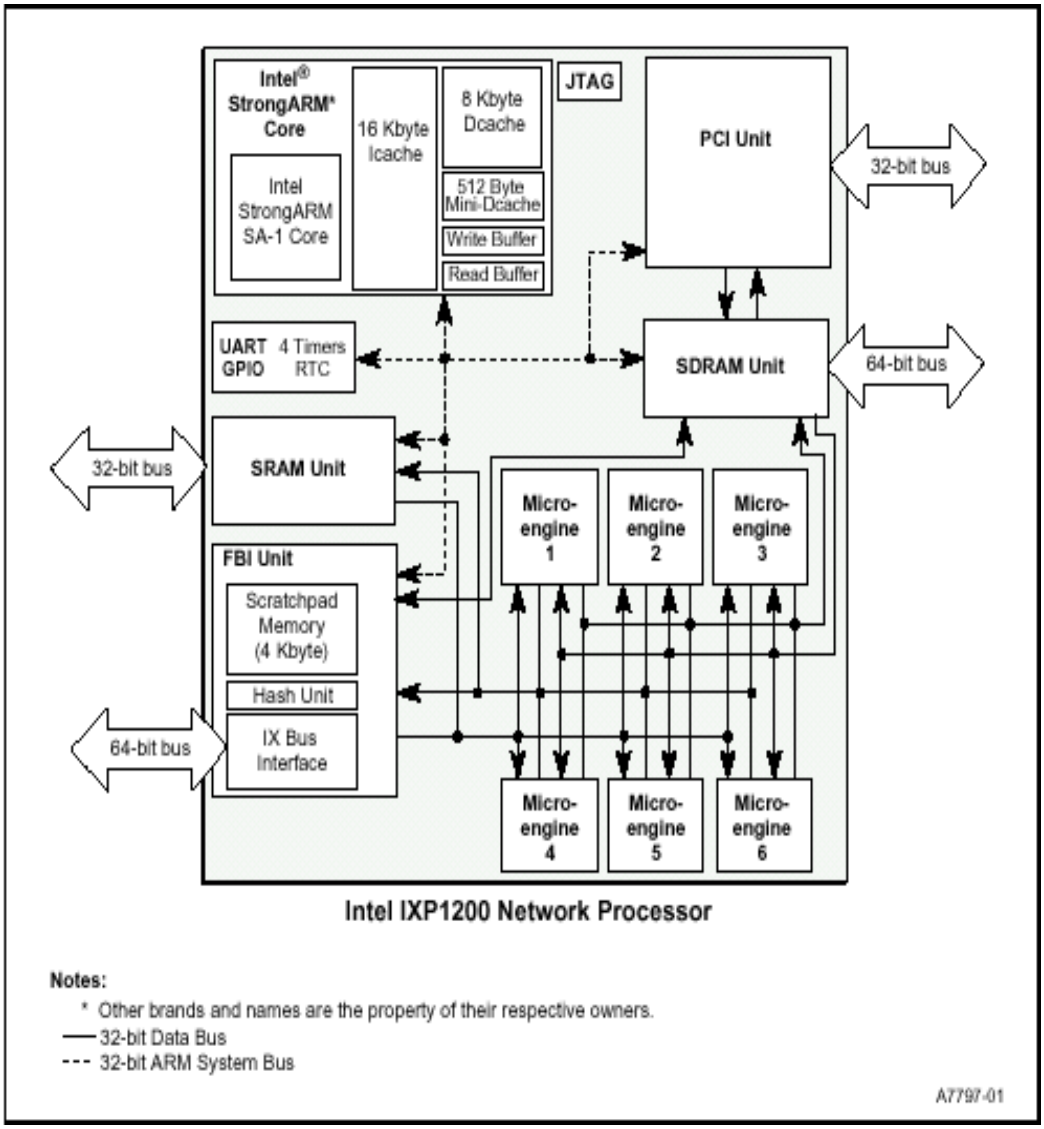

Figura 2.8 – Arquitetura do IXP1200 [INTEL 2000]

A seguir são descritas algumas características básicas de cada bloco lógico do IXP1200:

O Processador *StrongARM é um p*rocessador RISC de 32 bits, com 16kbytes de *cache* de instrução e 8kbytes de *cache* de dados. Possui um *mini-cache* de 512 bytes,

unidade de gerenciamento de memória e acesso às unidades FBI, PCI e memória SDRAM.

As Microengines suportam *multi-thread e* possuem ULAs (*Unidades Lógicas e Aritméticas*) e 128 registradores de propósito geral e 128 de transferência. Possuem *cache* de instrução e de dados e acesso à unidade FBI, aos canais PCI DMA, SRAM e SDRAM.

As Memórias são do tipo SDRAM possui um máximo de endereçamento de 256Mbytes e a SRAM um máximo de 8Mbytes. Existe também uma memória FlashROM de 8Mbytes para boot do processador StrongARM.

A Unidade FBI trabalha na camada MAC sobre o barramento IX e é responsável pelo serviço de periféricos. O barramento IX possui 64 bits em 66 MHz e é responsável pela interface com a ETHERNET, ATM e outros processadores da família IXP.

A Unidade PCI é uma unidade padrão PCI de 32 bits responsável pela interface com outros dispositivos PCI, processadores (CPUs) e dispositivos MAC. Opera em uma freqüência de 66 Mhz.

Este processador IXP1200 possui ainda uma porta serial UART (*Universal Asynchronous Receive and Transmit*), e quatro portas I/O de propósito geral. A família de processadores de rede da Intel ainda é composta pelos seguintes processadores: IXP2850, IXP2800, IXP2400, IXP425, IXP422, IXP421 e IXP420.

2.5.4. O PowerNP NP4GS3 da IBM

Neste tópicos são descritos, os principais blocos do NP4GS3, Processador de Rede IBM [IBM 2002], cuja arquitetura se mostra na Figura 2.9.

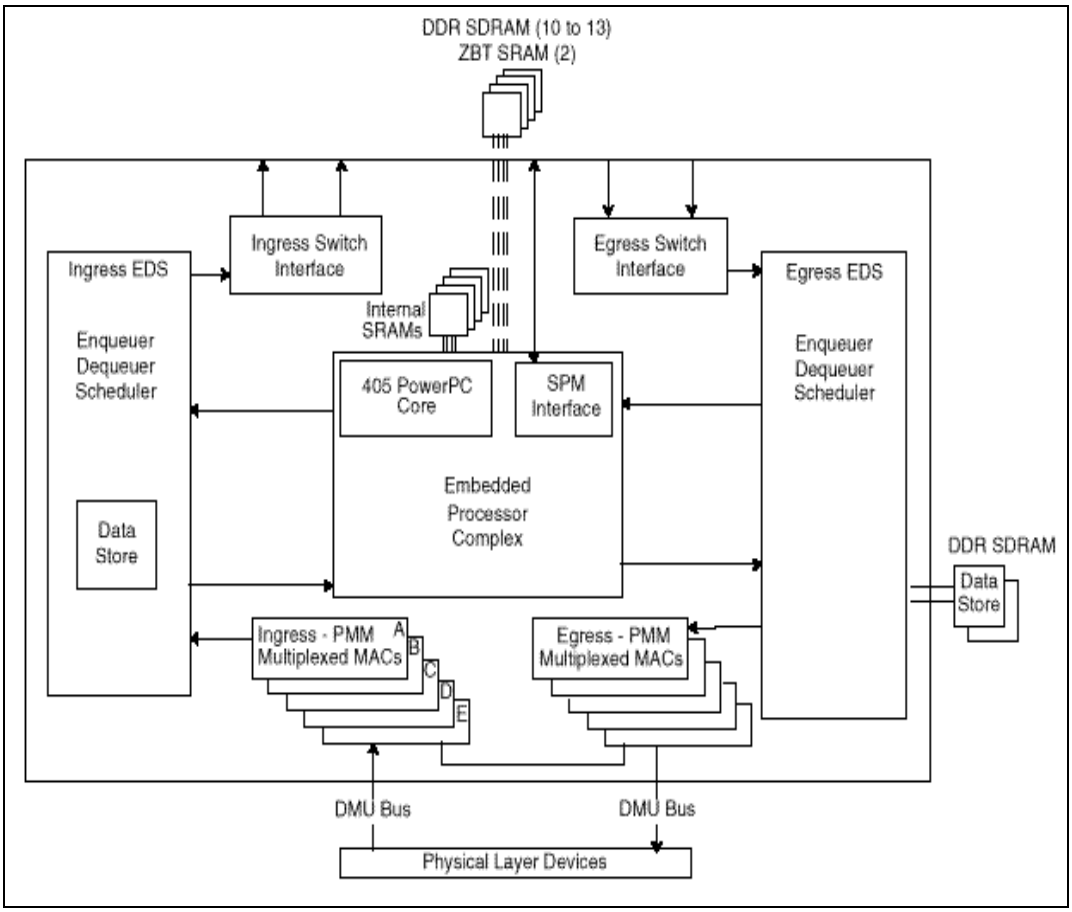

Figura 2.9 – Arquitetura do NP4GS3 [IBM 2002]

O Multiplexador Camada Física provê a interface do processador de rede com portas de comunicação externa. Existem dez unidades de transferência de dados (*Data Mover Units*), sendo que, cinco de entrada e cinco de saída. Quatro pares são usados para interfaces externas. Cada DMU suporta os seguintes tipos de configuração: 10 x 10/100 FDX *Ethernet*, por DMU, 1 x 1GB, por DMU, 4 x OC-3, por DMU, 1 x OC-12, por DMU, 1 x OC-48, por 4 DMUs. O outro par DMU é para comunicação interna ao processador (*Wrap Port*).

O bloco Ingress EDS (I-EDS) é responsável por encaminhar os *frames* recebidos pelo DMU. Suas principais funções são: guardar os *frames* em memória interna RAM, filtrar e decidir algumas alterações no frame, desenfileirar, encaminhar ou descartar o frame.

A interface SI (*Switch Interface*) provê uma célula de dados baseado na interface entre NPs via um *switching fabric* (para três ou mais NPs) ou conexão direta entre dois

NPs. Este bloco está dividido nas seguintes partes: I/E-SDM: Interface lógica entre o I/E-EDS e a célula, I/E-SCI: Transmite e recebe células da interface física. A interface DASL é uma interface física entre: NP e switch fabric, Entrada/saída de 1 NP e entrada/saída de 2 NPs

A interface Egress EDS (E-EDS) apresenta como principais funções: receber *frames* do *Switch Interface,* enfileirar os *frames* e guardar em memória RAM externa, processar os pacotes, desenfileirar e encaminhar os *frames*.

O módulo Shaper Gerencia a largura de banda por frame com base nas portas *Egress* – DMU.

Já o módulo EPC (*Embedded Processor Complex*) determina o que deve ser feito com os *frames* recebidos pelo I-EDS e E-EDS. Contém oito *Dyadic Picoprocessors* e nove co-processadores.

O módulo EPPC *(Embedded PowerPC), é um* PowerPC 405 *engine* e foi desenvolvido para controle de funções de rede tais como: encaminhar e filtrar pacotes analisando campos do protocolo IP, controlar protocolos de roteamento tal como: RIP, OSPF e BGP e, gerenciar, configurar, diagnosticar e suportar agentes SNMP.

O sistema de memórias suporta as memórias interna e externa. Os tipos são: SRAM (interna), Z BT SRAM (externa) e DDR SDRAM (externa)

2.5.5. Lucent/Agere – FPP/ASI/RSP

A solução da Lucent/Agere consiste em três tipos de processadores: FPP (*Fast Pattern Processor*), RSP (*Routing Switch Processor*) e ASI (*Agere System Interface*), conforme se observa na Figura 2.10 [AGERE 2001].

A Figura 2.10 ilustra a arquitetura da solução usando os três processadores. O *pipeline* de dados é realizado através de interface física entre o FPP e o RSP. O processador ASI é utilizado apenas quando há um excesso no gerenciamento.

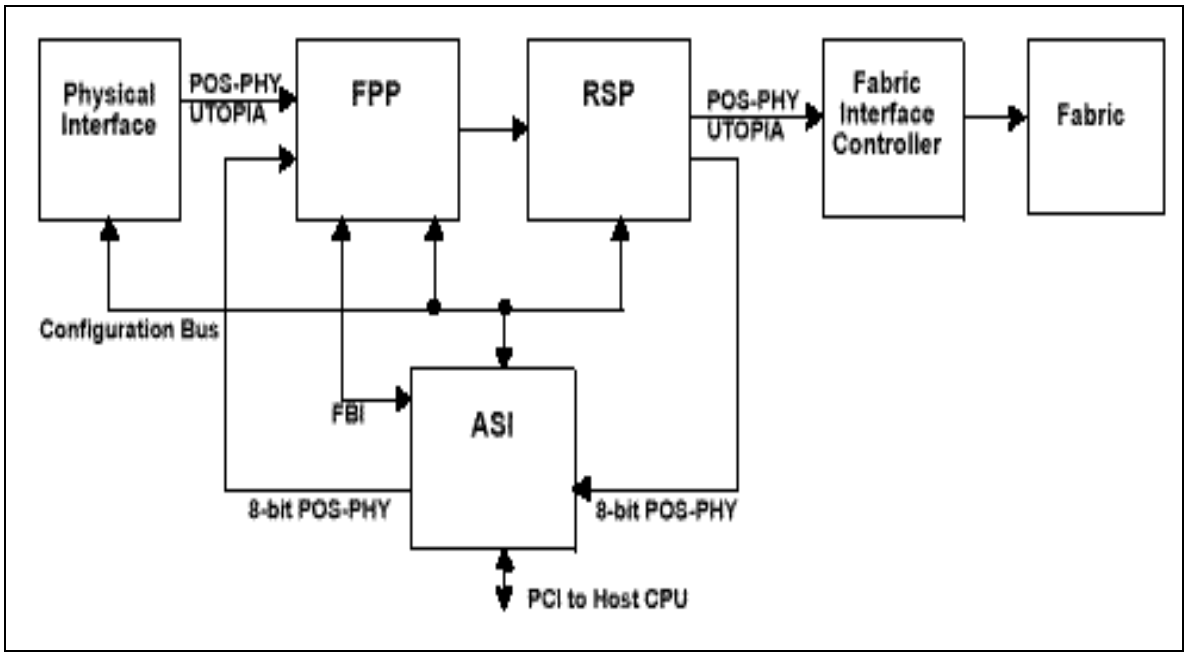

Figura 2.10 – Arquitetura integrada dos três processadores da Lucent/Agere

[LUCENT 1999]

O FPP é um processador com *pipeline* e suporta até 64 *threads*. Cada unidade de pacote de dados (PDU) que chega do barramento UTOPIA é assinalado como uma nova *thread*. O suporte em *hardware* para o chaveamento de contexto habilita o FPP a processar múltiplas PDUs em paralelo, conforme se visualiza na Figura 2.11 [Agere 2001].

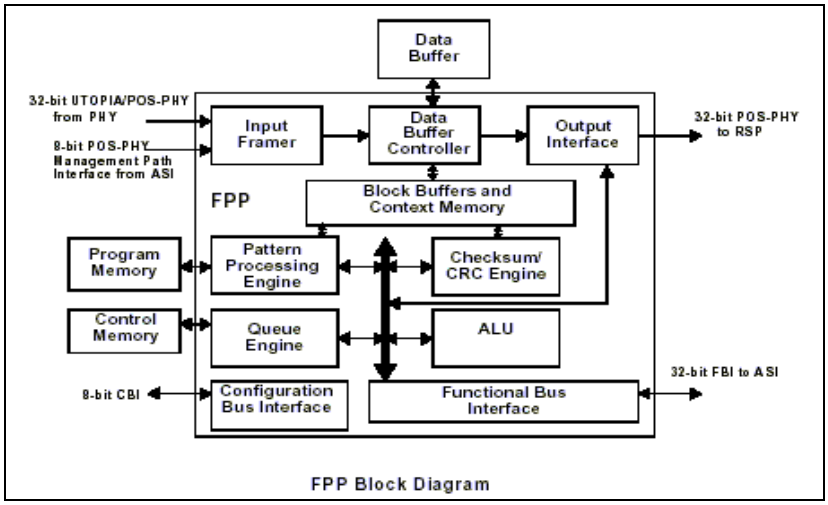

Figura 2.11 – Arquitetura do FPP [AGERE 2001]

As principais aplicações para este processador são: roteadores e *switches, firewalls*, gerenciamento e monitoramento de redes, segmentação ATM e frame *Relay,* processamento de lista e controle de acesso.

O processador RSP recebe a classificação dos dados e as PDUs do FPP e entrega para a *fabric*. As quatro principais funções do RSP são: enfileiramento, gerenciamento de tráfego, gerenciamento de modelo e modificação de pacote, conforme se observa na Figura 2.12 [AGERE 2001].

O RSP recebe os pacotes de instrução do FPP e guarda as PDUs na SDRAM. Baseado nos cálculos de gerenciamento de tráfego, as PDUs podem ser enfileiradas ou descartadas. A PDU pode então ser transmitida ou buscada de memória, modificada e então transmitida.

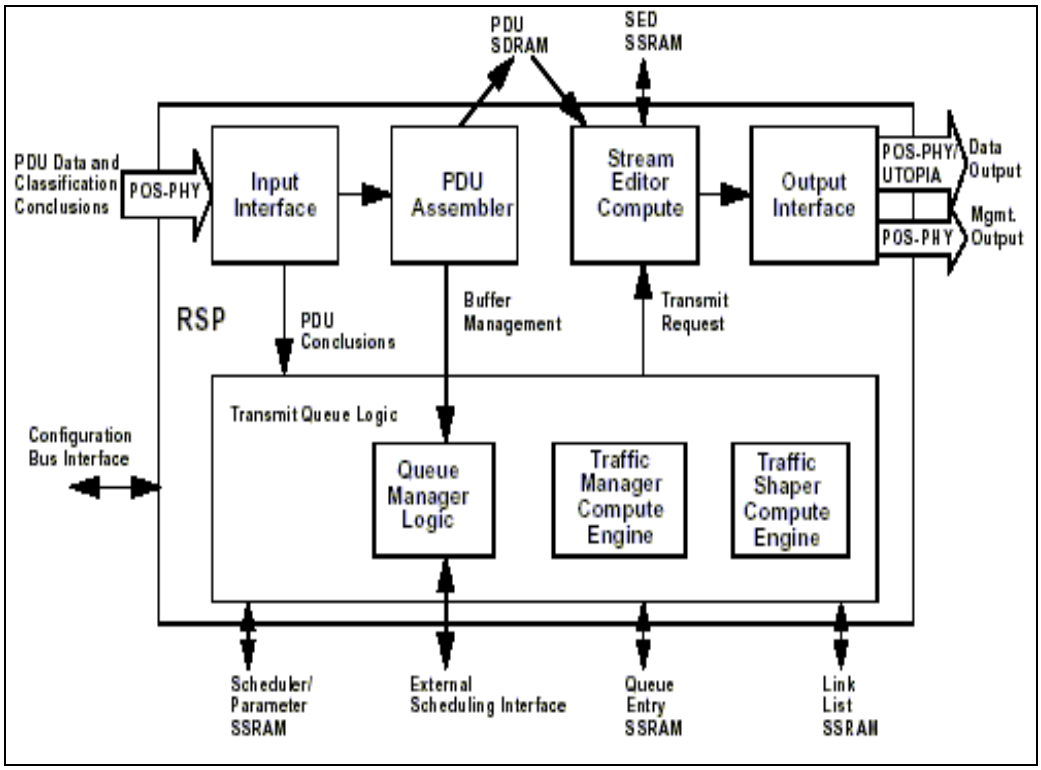

Figura 2.12 – Arquitetura do RSP da Lucent [AGERE 2001]

A interface ASI (*Agere System Interface*) possui como principal função o processamento que requer o caminho "lento". Este processamento é caracterizado por: roteamento, atualizações em tabelas, atualizações em filas, exceções e estatísticas. Esses detalhes são melhor observados na Figura 2.13 [AGERE 2001], onde se observa

também que existe uma interface PCI para gerenciamento externo e uma interface PCI133 SDRAM, para acesso à memória.

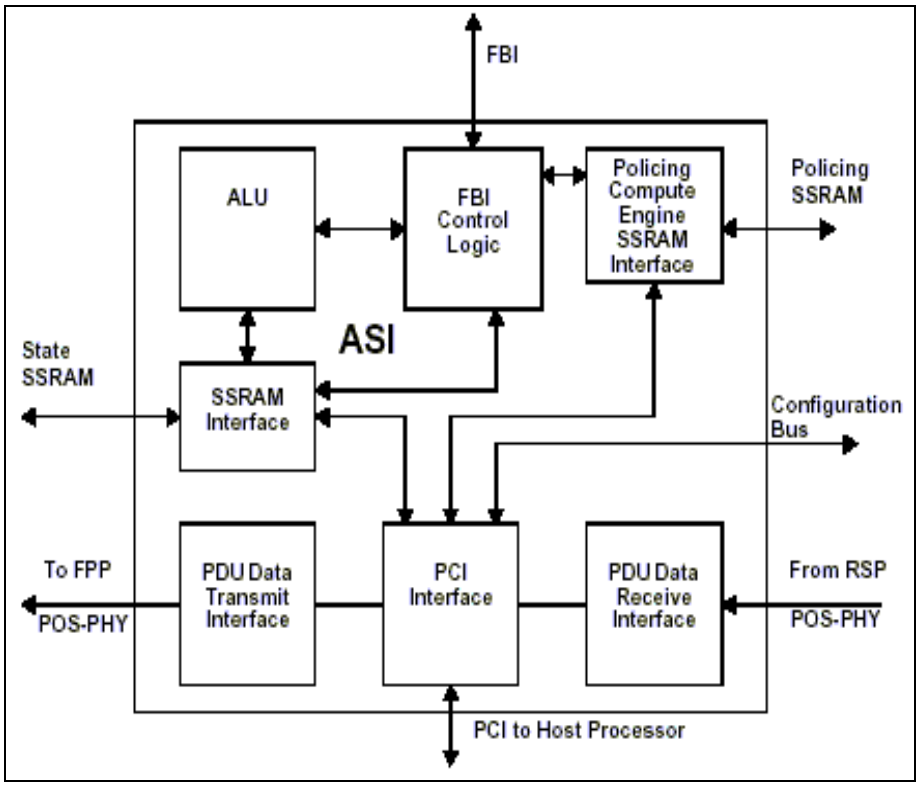

Figura 2.13 – Arquitetura do ASI [AGERE 2001]

2.5.6 O NP – 1 da EZChip

O NP-1 é um Processador de Rede *full-duplex* de 10Gbits camada 7 [EZCHIP 2002]. Este processador incorpora a tecnologia TOPcore (*Top Optimized Processing Core*), que consiste em um *array* integrado de processadores customizados em uma arquitetura superescalar que provê desempenho e flexibilidade de processamento de pacotes.

Suas principais aplicações são: equipamentos de rede, tais como roteadores, *switches* e *gateways, b*alanceamento de carga, analisadores e testadores de rede, f*irewalls* e VPNs (*Virtual Private Network*). A Figura 2.14 mostra as interfaces do NP-1.

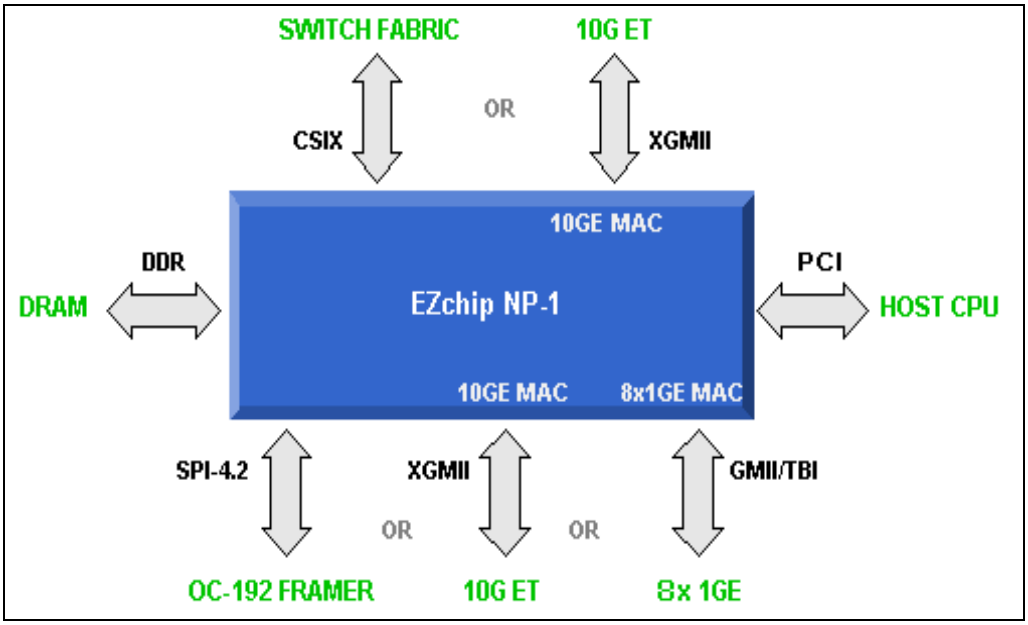

Figura 2.14 – Interfaces do NP-1 [EZCHIP 2002]

O processador NP-1 da EZCHIP contém uma porta OC-192 com interface SPI4.2, Duas portas *Ethernet* 10 Gbits com uma MAC integrada, Oito portas *Ethernet* 1 Gbit com oito MACs integradas de 10/100/1000 Mbps, Uma interface padrão CSIX (*Common Switch Interface*) provê conectividade com um *switch fabric*, Uma interface PCI é usada para interconectar o NP-1 com outro tipo de processador.

A Figura 2.15 ilustra a arquitetura *TOPcore*. A função de cada estágio é a seguinte:

- *TOPparse*: identificar protocolos e extrair cabeçalhos dos pacotes;

- *TOPsearch*: executar *lookups* em diferentes níveis;

- *TOPresolve*: assinala pacotes para enfileirar ou enviar para portas;

- *TOPmodify*: modifica conteúdo dos pacotes;

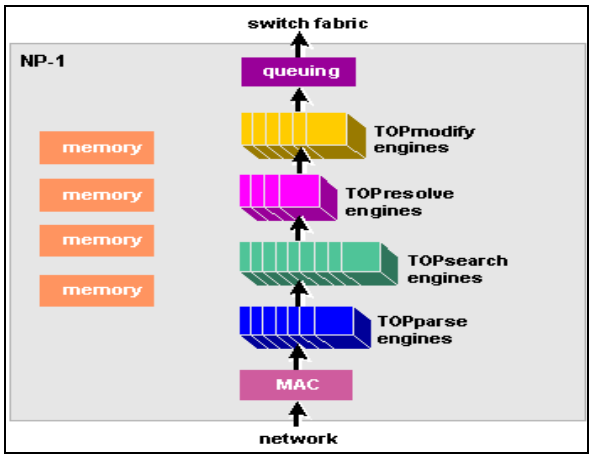

Figura 2.15 – Arquitetura do NP-1 [EZCHIP 2002]

2.5.7. O Prism IQ2000 da família Sitera/Vitesse

Este é um processador de rede que trabalha como um processador padrão [VITESSE 2002], conforme se observa na Figura 2.16. Enquanto o processador padrão faz o trabalho de controle, processamento e gerenciamento de sistemas, o co-processador de rede faz o trabalho de processamento dos pacotes, classificação, *lookups* e análise de QoS (*Quality of Service*) e COS (*Class of Service*). O IQ2000 trabalha na camada 2 e 3 em equipamentos como *switches* e roteadores e suporta pacotes com taxas de 2.5Gbps.

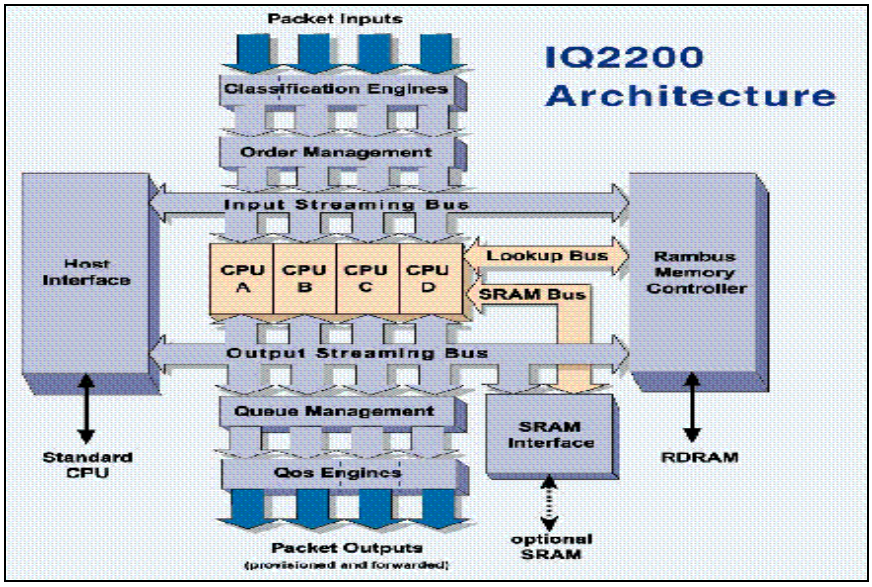

Figura 2.16 – Arquitetura do IQ2000 [VITESSE 2002]

Algumas aplicações do IQ2000 são o roteamento de multi-protocolos complexos, classificação, filtragem, inspeção e criptografia, políticas de QoS, *Multicast,* tradução de endereços de rede e outras operações de processamento de pacotes.

Como se observa na Figura 2.16, o IQ2000 contêm quatro processadores escalares de 200Mhz.

2.5.8. O CS2000 da família Chameleon

O processador de comunicação reconfigurável da Chameleon Systems foi o primeiro da indústria de processadores de rede a utilizar reconfiguração dinâmica como parte do sistema normal de operação [CHAMELEON 2000]. A arquitetura do CS2000 é ilustrada na Figura 2.17.

Algumas características técnicas do CS2000 são: possui um *Fabric* reconfigurável de 32 bits, 84 unidades de caminho de dados de 32 bits, 24 unidades de multiplexadores de 16x24 bits, 3000 MMACS de 16 bits, 24000 MOPS de 16 bits, *Fabric* reconfigurável em 1 ciclo de *clock,* Canais de voz/dados/vídeo por *chip,* Processador ARC integrado de 32 bits, Controlador PCI integrado de 32 bits, Controlador de memória integrado de 64 bits, Controlador de DMA integrado de 16 canais, 160 pinos de entrada e saída programáveis, Largura de banda de 2Gbytes por segundo.

Algumas de suas principais aplicações são: estações base sem fio de 2G e 3G, *Wireless Local Loop* (WLL) [SALEFSKI 2001], e voz sobre IP e alto desempenho DSL (*Digital Subscriber Line).*

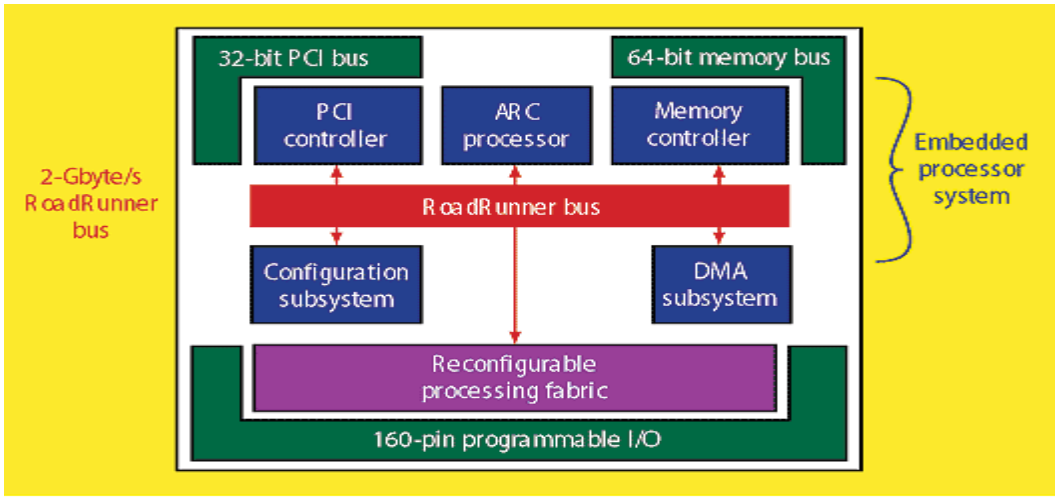

Figura 2.17 – Arquitetura do CS2000 [CHAMELEON 2000]

A unidade reconfigurável conhecida como CS2000 é o *Processing Fabric* [COMPTON 2002] [Martins 2002]. Esta unidade é dividida em submódulos que compõem as unidades básicas de reconfiguração. O CS2112 possui quatro subblocos, e cada um pode ser reconfigurado independentemente. Cada subbloco consiste de três blocos, constituídos de Unidades de Caminho de Dados, Multiplicadores, Memórias Locais e Unidades de Controle.

2.5.9 O Co-processador IBM 4758

O coprocessador criptográfico da IBM é um subsistema seguro e avançado que pode ser instalado em sistemas de usuário para executar o DES e a criptografia de chave pública [SECURITY 2001].

Um coprocessador seguro é um ambiente computacional com um processador de propósito geral com suporte a ataques físicos e ataques lógicos. O dispositivo deve funcionar para os programas que se supõe serem programas de ataque. Um usuário deve poder (remotamente) distinguir entre o dispositivo real e a aplicação, e um inspecionador inteligente. O coprocessador deve manter a segurança mesmo se os adversários realizarem uma análise destrutiva de um ou mais dispositivos.

Muitos usuários costumam trabalhar nos ambientes distribuídos onde é difícil ou impossível fornecer a segurança física completa para processamento. E, em algumas aplicações, o adversário é o usuário da outra extremidade. Portanto, se torna necessário um dispositivo em que se possa confiar mesmo que não possa controlar o ambiente.

O coprocessador IBM está qualificado para detectar e reagir as tentativas de ataques lógico e para executar e processar com segurança, incluindo execução correta de diversos algoritmos comercialmente significativos [IBM 4758].

Suas aplicações podem obter serviços de criptografia por meio do PKCS#11 que caracteriza a sustentação do DES, do triplo-DES (com chaves do triplo-comprimento), do RSA e do DSA, e serviços de hashing Sha-1, MD2 e MD5. Pode-se empregar múltiplos coprocessadores, cada um que opere como unidade independente de PKCS #11. As aplicações são projetadas para suportar o múltiplo gerenciamento de chave, melhorando a vazão e/ou a disponibilidade com um coprocessador adicional.

A arquitetura do co-processador IBM 4758 é ilustrada na Figura 2.18.

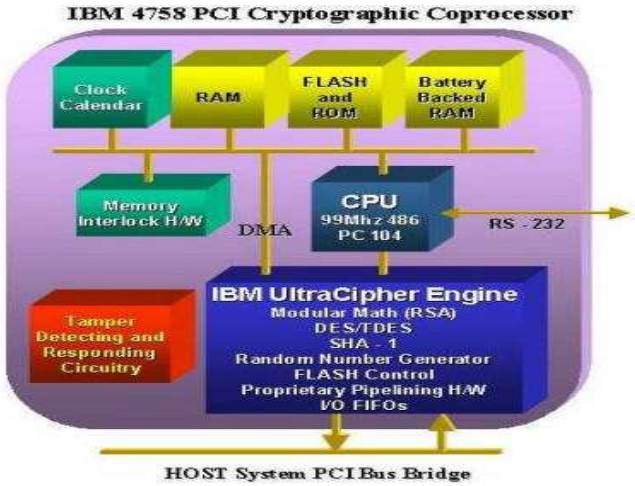

Figura 2.18 – Arquitetura do IBM 4758[IBM 4758]

2.5.10 O Processador de Rede IXP2800

Os projetos precedentes adicionaram a segurança à rede através de um coprocessador ou de um processador de segurança. Os processadores de segurança podem realmente escalar a taxas de dados mais elevadas, mas devem executar muitas

das mesmas funções que o processador de rede faz para conseguir a taxa de dados elevada.

O IXP2800 tem capacidade de processar pacotes da rede em até 10 Gigabits por segundo, com um baixo custo em relação consumo e potência [IXP 2800]. A Figura 2.19 mostra a arquitetura do IXP2800.

O IXP2800 consiste em diversas unidades que são conectadas por meio de um clip. O chip conecta as seguintes unidades que podem diretamente ser usadas para processamento de segurança: microengines, SRAM, DRAM, lookup, PCI, e XScale, através de barramentos. As unidades de criptografia são adicionadas ao chip e são acessíveis pelos microengines.

Microengines é usado processamento, como ESP com processamento para o tráfego de IPSec e modificar a informação do estado da segurança. Os microengines são projetados para processar pacotes e para colocar os dados na memória.

O IXP2800 possui três memórias DRAM, fornecendo bastante capacidade para assegurar milhões de associações de segurança e bastante vazão a taxas de 10 Gigabit por segundo usando IPSec.

O hashing para lookups pode ser usado encontrar a informação requerida da associação da segurança para um pacote de dado.

Pci usado para operações de chave pública para conduzir pelo barramento as operações de criptografia

O IXP2800 inclui um processador de XScale que executa processamento de propósito geral. O processador de XScale pode ser usado para a manipulação de exceção para o pacote que processa, ou para os protocolos da instalação de sessão, tais como a chave da Internet. O IXP2800 permite diversos benefícios para processamento de segurança. Estes benefícios incluem a flexibilidade da execução dos protocolos, da otimização dos protocolos para determinadas aplicações.

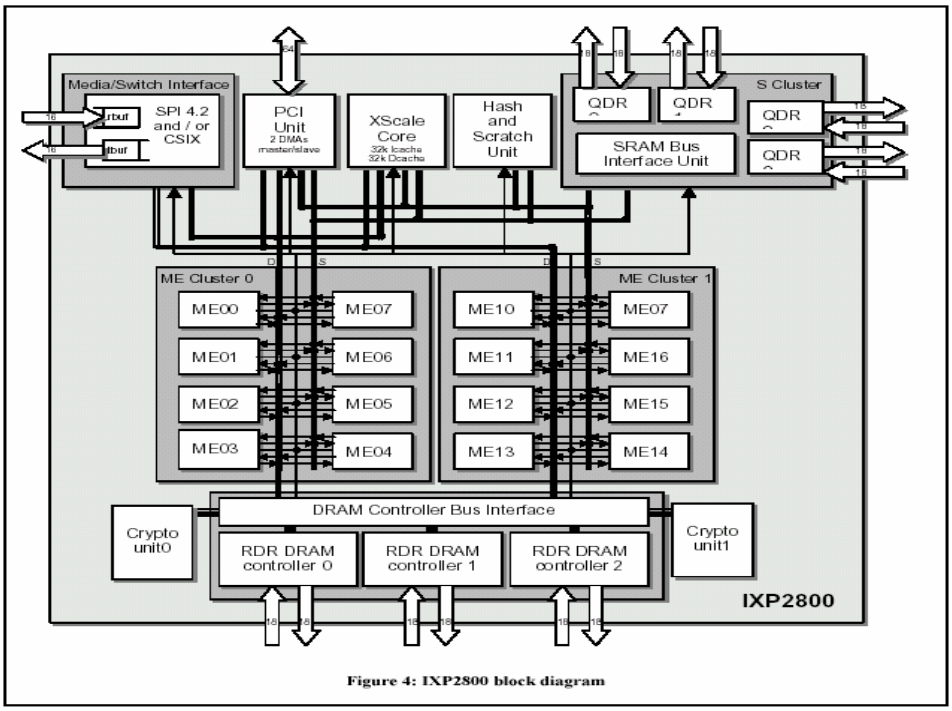

Figura 2.19 – Arquitetura do IXP2800 [IXP 2800]

2.6 ARQUITETURAS DOS SISTEMAS DE SEGURANÇA

Há três maneiras preliminares que permitem adicionar funções de segurança a um equipamento de rede.

Primeiro é o método de usar um coprocessador acoplado a um processador de rede ou um processador geral. Este método torna-se menos prático porque o pacote deve atravessar recursos compartilhados tais como barramentos ou memória de dados.

O segundo método deve adicionar um processador da segurança com um processador de rede, sendo possível taxas de dados elevadas. O processador de segurança deve executar muita das mesmas funções obrigatórias do processador de rede, tais como a remontagem do pacote; assim o trabalho deve ser repetido e a área deve ser duplicada.

O terceiro método deve adicionar a funcionalidade da segurança no mesmo processador de rede, assim adicionando a funcionalidade da segurança no processador de rede ao manter a taxa e ao minimizar a área do processador.

Os processadores atuais com segurança requerem que o pacote este disponível inteiro na memória antes de ser processado, sendo assim o pacote estará inteiro na memória antes de ser transmitido.

2.6.1 Arquitetura Look-aside

Como se pode observar na Figura 2.20, um processador de rede e um processador da segurança na configuração look-aside. Os dados do pacote tipicamente devem atravessar a memória quatro vezes, ao contrário de duas vezes, e o barramento entre o processador de rede e o processador de segurança deve ser capaz de fazer ao menos duas vezes a taxa desejada.

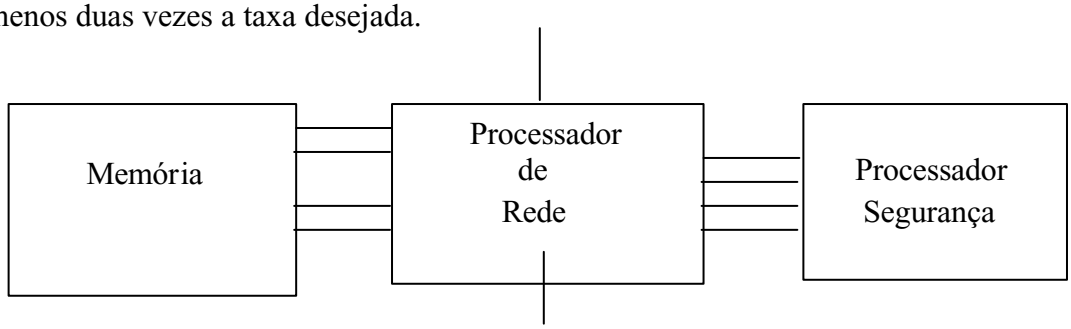

Figura 2.20 - Arquitetura look-aside

2.6.2 Arquitetura *Flow-through*

A arquitetura *Flow-through* resolve os problemas de desempenho da arquitetura look-aside, mas requer que o processador de segurança seja capaz de fazer muitas das funções que o processador de rede requer. Algumas destas tarefas incluem a remontagem dos pacotes, protocolo que processa, e manipulação de exceção.

É desejável poder usar a mesma arquitetura subjacente para múltiplas aplicações. Isto seria possível se a funcionalidade de segurança fosse adicionada ao processador de rede.

Na Figura 2.21 pode-se observar a divisão da arquitetura ***Flow-through.*** Algumas aplicações requerem que o processador de rede esteja colocado antes do processador de segurança, outros requerem que o processador da segurança esteja colocado antes do processador de rede, e alguns requerem um processador de rede antes

e depois do processador da segurança. Por o exemplo, uma aplicação do proxy do SSL (*Secure Socket Layer)* requer a terminação de uma conexão do TCP, do SSL que processam, e então do estabelecimento de uma nova conexão do TCP [INTEL 2002].

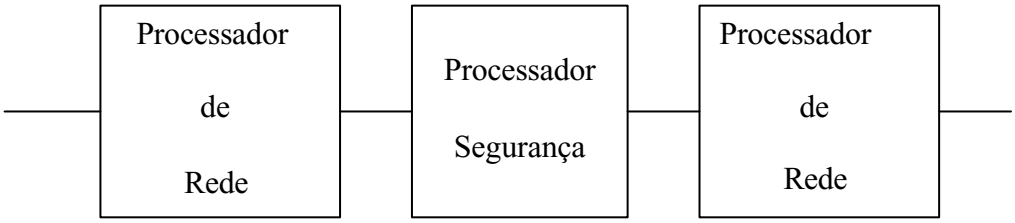

Figura 2.21 - *Arquitetura* Flow-through

Geralmente a segurança é implementada através de algoritmos de criptografia que incluem a manipulação de dados na totalidade dos pacotes, mantendo a confiabilidade e a integridade.

A arquitetura Flow-through requer do processador de segurança fazer funções similares às funções do processador de rede, que além do processamento devem também trabalhar com os diferentes protocolos. Isto pode ser feito com um processador dedicado ou talvez um ou muitos processadores do protocolo integrados no processador de segurança.

No IXP2800 foi adicionada a funcionalidade e a flexibilidade extensivas do processador de rede e adicionaram-se os algoritmos de criptografia necessários.

O IXP2800 consiste em diversas unidades, as quais são conectadas por barramento. A conexão das unidades podem diretamente ser usadas para o processamento de segurança: *microengines*, SRAM, DRAM, mistura do lookup, PCI, e Xscale. As unidades de criptografia são adicionadas e são acessíveis pelos *microengines* [ADILETTA 2002].

Microengines são usados para fazer o processamento, como o ESP que processa o tráfego de IPSec. O processamento inclui construir novos encabeçamentos, o campo de um encabeçamento a outro, e modificar a informação do estado da segurança. Os microengines são projetados para processamento do pacote. Por exemplo, durante processamento acontecem verificações ESP em pacotes entrantes, ele é necessário para

ler o estado do número de seqüência, e para escrever para trás os dados à memória [INTEL 2002]

O IXP2800 fornece três barramentos independentes da memória DRAM, com capacidade de assegurar milhões de associações da segurança e de bastante *throughput* alcançando, no IPSec taxas de até 10Gigabit/segundo.

O hashing para localização pode ser usado para encontrar a informação requerida da associação da segurança para um pacote dado. Embora outros métodos possam ser aplicados, combinar uma unidade dedicada a localização com o uso de SRAM externo rende um mecanismo efetivo em custo para conduzir muitas localizações requeridas.

Um aspecto importante do processamento de segurança deve encontrar associações da segurança a fim de aplicar os algoritmos de criptografia apropriados para um fluxo dado. Ao estabelecer a associação da segurança, algumas operações, tais como computações de chaves públicas, são requeridas. Um coprocessador que seja conectado ao barramento do PCI pode conduzir estas operações.

O IXP2800 permite diversos benefícios da característica para processamento de segurança. Estes benefícios incluem a flexibilidade da execução dos protocolos, da otimização dos protocolos para determinadas aplicações, e da execução de aplicações diferentes usando o mesmo processador.

A funcionalidade da segurança é projetada para permitir a sustentação para muitos protocolos, tais como IPSec, SSL/TLS, ATM, e os protocolos futuros que usam algoritmo 3DES, AES, e Sha-1.

A unidade de criptografia é compreendida de diversos algoritmos que na junção fornecem confidencialidade e a integridade dos dados.

A funcionalidade adicionada ao processador de segurança suporta cifrar os dados usando o (DES), 3DES, e os algoritmos para cifrar avançado (AES) junto com o algoritmo de hashing (Sha-1).

A Figura 2.22 mostra o trajeto de dados na unidade de criptografia, que consiste em dois núcleos 3DES, em um núcleo de AES, e em dois núcleos de SHA 1. É possível

processar os dados através dos núcleos de SHA 1 qualquer um antes ou depois das cifras processarem os dados.

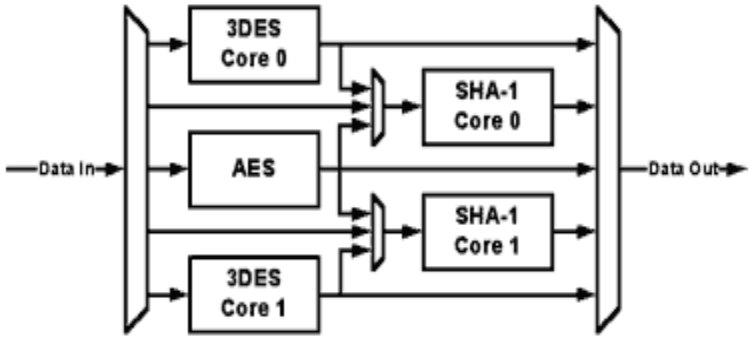

Figura 2.22 - Vista geral da unidade de criptografia [INTEL 2002]

Um benefício grande de adicionar funções da segurança ao IXP2800 é a economias no consumo de energia. Se os processadores atuais de segurança fossem escalados a 10 Gigabits por segundo de taxas, requereriam entre 13 e 30 watts. Os projetistas do processador tiveram a atenção para a redução do consumo de energia, o mais importante é a falta de dispositivos de I/O o que contribui para a funcionalidade do processador, o IXP2800 com a unidade de criptografia reduz o consumo de energia por um fator 10X sobre seus concorrentes. Este é um benefício tremendo para poder funcionar em muitos equipamentos de *networking.*

2.7 Considerações finais do Capítulo

Este capítulo apresentou os conceitos básicos sobre processadores de rede e discutiu a arquitetura dos princípais NP que estão hoje no mercado.

Existem diversos processadores de rede comerciais de diversos fabricantes.

Na Tabela 2.4 pode-se observar um resumo das princípais características de alguns processadores de redes comerciais, analisados e pesquisados neste projeto. Importante salientar que há poucos processadores de rede com suporte à segurança.

Tabela 2.4. Processadores de Redes Comerciais

Processadores de Rede Comerciais	**Características mais Importantes**	**Segurança**
Motorola – PowerQUICC Family	Aplicações em roteadores e *switches* de LAN/WAN, dispositivos de integração de redes, concentradores de rede, *gateways*, DSL/*cable modems* e sistemas de voz sobre IP.	Não
Motorola/C-Port – C5 Family	Prioriza os critérios de QoS (qualidade de Serviço)	Não
Intel – IXP1200	Sete processadores RISC, responsável pelo processamento mais complexo, tal como construção e manutenção de tabelas e gerenciamento da rede.	Não
IBM – PowerNP NP4GS3	Encaminhar e filtrar pacotes analisando campos do protocolo IP, Controlar protocolos de roteamento tal como: RIP, OSPF e BGP, Gerenciar, configurar, diagnosticar e suportar agentes SNMP.	Não
Lucent / Agere – FPP/ASI/RSP	Aplicações para este processador são: Roteadores e *switches, Firewalls*, gerenciamento e monitoramento de redes, Segmentação ATM e Frame *Relay,* Processamento de lista e controle de acesso.	Não
EZCHIP – NP – 1	Aplicações em roteadores, *switches* e *gateways,* Balanceamento de carga, Analisadores e testadores de rede, *Firewalls* e VPNs.	Não
Sitera/Vitesse – Prism IQ2000 Family	Processamento dos pacotes, classificação, *lookups* e análise de QoS (*Quality of Service*) e Cós (*Class of Service*). O IQ2000 trabalha na camada 2 e 3 em equipamentos como *switches* e roteadores e suporta pacotes com taxas de 2.5Gbps.	Não
Chameleon CS2000 Family	Aplicações em estações base sem fio de 2G e 3G, *Wireless* Voz sobre IP e alto desempenho DSL. processador reconfigurável.	Não
Intel – IXP2800	Módulo de segurança no qual realiza criptografia e flexibilidade da execução dos protocolos, otimização dos protocolos para determinadas aplicações, e execução de aplicações diferentes.	Sim

IBM – 4758	O co-processador criptográfico da IBM é um subsistema seguro e avançado que pode ser instalado em sistemas de usuário para executar o DES e a criptografia de chave pública.	Sim

Existem também processadores de rede com módulo de segurança que realizam criptografia, entre eles podemos citar o IXP2800 da Intel e o coprocessador IBM – 4758.

O processador de rede IXP2800 é projetado para gateways, pontos de acesso wireless e roteadores e switches SME, com suporte a criptografia.

Não existem no Brasil processadores de rede em FPGA com um módulo de segurança usando o padrão PKCS#11. Assim, implementação de um módulo PKCS#11 em um processador de rede comercial é um grande avanço para a segurança da informação. No próximo capítulo será mostrado os conceitos de segurança usando o padrão PKCS#11.

CAPÍTULO 3

3. CONCEITOS DE SEGURANÇA E O PADRÃO PKCS#11

Este capítulo apresenta conceitos gerais de criptografia e detalhes sobre o padrão PKCS#11.

3.1 Conceitos Gerais de Criptografia

Criptografia é a arte ou ciência de escrever em cifra, de forma a permitir normalmente que apenas um destinatário a decifre e compreenda. Quase sempre decifrar requer uma chave, uma informação secreta disponível ao destinatário.

No âmbito da tecnologia de informação, a criptografia é importante para que se possa garantir a segurança em todo o ambiente computacional que necessite de sigilo em relação às informações.

Desde o tempo dos egípcios a criptografia já estava sendo usada no sistema de escrita hieroglífica. Historicamente, a confidencialidade na comunicação entre pessoas podia ser garantida pela realização de encontros reservados.

Para a comunicação à distância, seria necessário enviar mensagens através de intermediários, e para manter a confidencialidade foram desenvolvidos códigos e cifras para esconder o conteúdo das mensagens dos bisbilhoteiros que as interceptassem, como dos próprios intermediários.

A criptografia é a ciência de desenvolver cifras, principalmente nas áreas diplomáticas e militares dos governos.

As telecomunicações aumentaram a rapidez e confiabilidade da comunicação remota. No século 20, o uso da criptografia foi automatizado, para tornar mais rápida e eficaz sua aplicação.

Atualmente a criptografia é usada como uma técnica de transformação de dados, segundo um código (ou algoritmo), para que eles se tornem ilegíveis, a não ser para quem possui a chave do código.

Existem dois tipos principais de criptografia: a simétrica e a assimétrica. Na criptografia simétrica, o remetente e o destinatário usam a mesma chave.

A criptografia assimétrica utiliza uma chave (pública) para cifrar e outra

(privada), para decifrar. Pode-se dizer que, ao invés de compartilhar uma chave secreta, utiliza-se duas chaves as quais são relacionadas matematicamente. Uma das chaves é aberta para que todos possam usá-la (chave pública) e a outra é mantida em sigilo (chave privada).

Dessa forma, uma mensagem criptografada com uma chave pública somente poderá ser decifrada com a chave privada correspondente do destinatário.

A criptografia assimétrica é usada com maior freqüência na Internet, pois é mais viável tecnicamente, pois não qual rota o pacote seguirá na internet, sabendo previamente onde serão enviados os dados. Se fosse usada a criptografia simétrica, podería-se ter grandes problemas, pois para distribuir a chave para todos os usuários autorizados teria-se um problema de atraso de tempo e possibilitaria também que a chave chegasse a pessoas não autorizadas.

A criptografia nos computadores não é usada somente para misturar e desembaralhar informações. O seu uso é para garantir segurança nos meios de transmissão e armazenamento, e também é muito usado para codificar dados e mensagens antes de serem enviados, para que mesmo que sejam interceptados, dificilmente poderão ser decodificados.

Uma ferramenta chave para garantir a privacidade é a autentificação. Diariamente usa-se autentificação, por exemplo, quando se assina um cheque. Quando usa-se o meio eletrônico de comunicação também se faz uso de autentificação. A assinatura digital garante a auditoria do documento. Assim pode-se usá-lo para controlar acessos a discos rígidos compartilhados ou controlar canais de TV pagas por tempo de uso. Com certas ferramentas básicas e com o uso de assinaturas digitais e criptografia, pode-se elaborar esquemas e protocolos que permitem o uso de *dinheiro eletrônico* [STALLINGS 1996].

O algoritmo simétrico de chave única, mais difundido é o DES (*Data Encryption Standard*). Esse algoritmo foi desenvolvido pela IBM e adotado como um padrão nos Estados Unidos desde 1977 [STALLINGS 1996]. O algoritmo DES trabalha codificando blocos de 64 bits, usando uma chave de 56 bits mais 8 bits de paridade. Para quebrar o DES pela força bruta, isso é, tentar todas as combinações

possíveis de chave, como é uma chave de 56 bits tem-se um valor total resultante de valor obtido pela operação de (2 elevado a 56 chaves possíveis).

O algoritmo de chave pública, isso é, para criptografia assimétrica mais difundido é o RSA (significa o nome dos autores: Rivest, Shamir e Adleman). A segurança do algoritmo se baseia na intratabilidade da fatoração de produtos de dois números primos [STALLINGS 1996] [TERADA 2000].

3.2 Padrões de Segurança

Existem diversos padrões de segurança baseado em criptografia, tais como aqueles descritos na Tabela 3.1.

Tabela 3.1 - Padrões de Segurança

Padrão	**Descrição**
ANSI X9	Utilizado nas industrias.
S/MIME	Troca de mensagens na Internet com sigilo, autenticação e integridade.
SSL	Utilizado pelo HTTPS e fornece sigilo e autenticação em conexões na WEB.
PKIX	Os certificados criados para prover a segurança necessária ao padrão X.500.
PKCS	Padrão de Chave Pública de Criptografia.

O ANSI X9 é um Padrão Nacional Americano de segurança.

O S/MIME (Secure Multipurpose Internet Mail Extensions) consiste em um esforço de um consórcio de empresas, liderado pela RSADSI e pela Microsoft, para adicionar segurança a mensagens eletrônicas no formato MIME. Apesar do S/MIME e PGP serem ambos padrões Internet, o S/MIME deverá se estabelecer no mercado corporativo, enquanto o PGP no mundo do mail pessoal. O PGP foi inventado por Phil Zimmermman em 1991, é um programa criptográfico famoso e bastante difundido na Internet, destinado a criptografia de e-mail pessoal [BERNERS 1989].

S/MIME é um padrão para a segurança de mensagens. O S/MIME pressupõe um PKI para assinar mensagens digitalmente e criptografar mensagens e anexos sem a

necessidade de compartilhar uma chave secreta. Como o e-mail é uma das aplicações mais utilizadas na Internet, o grupo de trabalho do S/MIME lidera a implementação das especificações do PKI, utilizando os padrões PKIX quando possível e complementando com outros padrões quando necessário.

O SSL é um padrão importante para a segurança no acesso a servidores Web. Estes protocolos também são usados para prover segurança em ambientes cliente/servidor e em outras aplicações não Web. Ambos dependem de um PKI para emissão de certificados para clientes e servidores [BERNERS 1989].

PKI (*Public Key Infrastructures*), ou Infra-estrutura de chaves públicas, é um ambiente para prover aos negócios eletrônicos condições de viabilidade a fim de que tenham os mesmos resultados daqueles conferidos aos contratos fora da rede. Tal recurso viabiliza a autenticação oficial e a integridade do documento, assim como sua elaboração e a confidencialidade nas operações e na assinatura digital, garantindo o valor jurídico e precavendo os envolvidos nas negociações da recusa do que foi firmado anteriormente [ADAMS 1999].

O PKCS é o Padrão de Chave Pública de Criptografia, foi escolhido por ser uma especificação de criptografia que pode ser usado para criptografia em hardware, por esse motivo apresentamos mais detalhes na próxima secão.

3.3 O padrão PKCS

O PKCS Padrão de criptografia de Chave Pública (*Public Key Cryptography Standards*) é uma série de especificações produzidas pelos Laboratórios RSA em cooperação com desenvolvedores de sistemas de segurança de várias partes do mundo, que visa acelerar, por meio da padronização, a utilização e o desenvolvimento de algoritmos de chave pública [RSA 03].

O padrão PKCS#11 surgiu em 1991, como resultado de encontros de um pequeno grupo de precursores no uso da tecnologia de chave pública e desde então tem se tornado referência até mesmo para padrões já estabelecidos, como ANSI X9, PKIX, SET, S/MIME e SSL. Atualmente seu desenvolvimento ocorre basicamente através de lista de discussões e *workshops* ocasionais.

A assinatura digital é o ato capaz de verificar e garantir a origem e a integridade de um documento reproduzido em meio digital, similar à estrutura de autenticação de cartório (muito utilizada no sistema legal brasileiro).

Tendo sido realizado pela empresa possuidora da patente do RSA, o sistema de chave pública descrito nesses padrões é basicamente o próprio algoritmo RSA.

Os objetivos globais do PKCS são manter compatibilidade com Internet PEM (*Privacy-Enhanced Mail Protocol*), estender o Internet PEM para lidar com qualquer tipo de dados e tratar um número maior de atividades e servir como proposta para ser parte dos padrões OSI [RSA 03].

A empresa *RSA Data Security*, formada pelos inventores das técnicas RSA de criptografia de chave pública, tem um papel importante na criptografia moderna. Nomeadamente, a sua divisão RSA *Laboratories* mantém uma série de padrões (*standards*) denominados de *Public Key Cryptography Standards* (*PKCS*) muito importantes na implementação e utilização de PKIs (*Public-Key Infrastructure*).

Os PKCS visam preencher o vazio que existe nas normas internacionais relativamente a formatos para transferência de dados que permitam a compatibilidade/interoperabilidade entre aplicações que utilizem criptografia de chave pública.

Atualmente existem doze padrões deste tipo: PKCS#1, #3, #5, #6, #7, #8, #9, #10, #11, #12, #13 e #15 [RSA 02]. Os objetivos da RSA na publicação destes padrões são: [RSA 02].

- Manter a compatibilidade com os padrões existentes, nomeadamente com PEM (*Privacy-Enhanced Mail Protocol*).
- Ir além dos padrões existentes, para permitir uma melhor e mais completa integração entre aplicações, normalizando a troca segura de qualquer tipo de dados.
- Produzir um padrão que possa ser incluído numa futura versão dos padrões OSI (*Open Systems Interconnection*).

Os PKCS descrevem a sintaxe de mensagens de uma forma abstrata, utilizando o ASN.1, e não restringem a sua codificação.

Como pode ser visto na Tabela 3.2, a especificação do PKCS responsável pela padronização da criptografia e verificação de assinatura e pela decriptografia e geração de assinatura utilizando o criptossistema RSA é constituída de 12 tipos de documentos gerados pelo PKCS entre elas o PKCS#11.

Tabela 3.2 - Temas tratados pelas especificações PKCS.

Número	**Tema**
1	• PKCS #1 – Representação das chaves Kpri e Kpub e como cifrar e assinar usando sistemas criptográficos RSA
3	• PKCS #3 – Padrão de Normalização de chave Diffie-Hellman
5	• PKCS #5 – Como cifrar com chaves secretas derivadas de um password
6	• PKCS #7 – Sintâxe de mensagens cifradas contendo assinaturas digitais
7	• PKCS #8 – Formato da informação de uma chave privada
8	• PKCS #9 – Tipos de atributos e sua utilização nas normas PKCS
9	• PKCS #10 – Requisição de certificados
10	• PKCS #11 – Define API de criptografia (Criptoki)
11	• PKCS #12 – Formato portável para armazenamento ou transporte (exportação e importação de certificados)
13	• PKCS #13 – Como cifrar e assinar com criptografia de curva elíptica
14	• PKCS #14 – Padrão para geração de números pseudo-random
15	• PKCS #15 – está ainda em desenvolvimento. Tem em vista propor uma norma para armazenamento de credenciais em "*token-based devices*" (*incluindo smart cards*)

3.3.1. O algoritmo RSA

O RSA é um sistema de criptografia de chave assimétrica ou criptografia de chave pública que foi inventado por volta de 1977 pelos professores do MIT

(*Massachusetts institute of Technology*) Ronald Rivest, Adi Shamir e o professor Leonard Adleman da USC (*University of Southern Califórnia*) (RSA 01).

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

Quanto maior os números primos utilizados para a criação da chave, maior é a segurança proporcionada por esse algoritmo. Hoje em dia os números primos que são utilizados, combinados formam uma chave de 1024 bits. Em algumas aplicações como, por exemplo bancárias que exigem o máximo de segurança, a chave chega a ser de 2048 bits para autoridades certificadoras [CHIARAMONTE 2003].

Com o passar do tempo, a tendência é que o comprimento da chave aumente cada vez mais. Esse fenômeno acontece, em grande parte, pelo avanço nos sistemas computacionais que acompanham o surgimento de computadores que são capazes de fatorar chaves cada vez maiores em um tempo muito baixo [CHIARAMONTE 2003].

O algoritmo RSA usado para a geração da chave pública e privada usadas para cifrar e decifrar as mensagens são simples. Observe-os a seguir [CHIARAMONTE 2003].

- Escolhe-se dois números primos grandes (p e q);
- Gera-se um número “n” através da multiplicação dos números escolhidos anteriormente (n = p . q);
- Escolhe-se um número “d”, tal “d” é menor que “n” e “d” é relativamente primo à (p-1).(q-1);
- Escolhe-se um número “e” tal que (ed-1) seja divisível por (p-1).(q-1). Para realizar esse cálculo é necessário o algoritmo de Euclides estendido.
- Os valores “e” e “d” são de expoentes públicos e privados, respectivamente. O par (n,e) é a chave pública e o par (n,d) é a chave privada. Os valores “p” e “q” devem ser mantidos em segredo ou destruídos.

Para cifrar uma mensagem com esse algoritmo é realizado o seguinte cálculo : $C = T^e \bmod n$, onde C é a mensagem cifrada, T é o texto original, e o n são dados a partir da chave pública (n,e).

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

3.3.2. Especificações do PKCS

A seguir alguns detalhes sobre cada uma das especificações do padrão PKCS [RSA 06].

O PKCS#1 Padrão Criptografia RSA (*RSA Encryption Standard***)** normaliza a utilização do algoritmo RSA nas seguintes aplicações:

- Representação de chave privada (Kpri) e chave pública (Kpub).
- Assinaturas Digitais: A informação a assinar é inicialmente reduzida a um valor de *hash* utilizando um algoritmo de *message digest* (*MD5*). O resultado é então cifrado com a chave privada RSA.
- Envelopes Digitais : A informação a proteger é cifrada com a chave de sessão utilizando um algoritmo simétrico (*DES*). Posteriormente, a chave de sessão é cifrada com a chave pública RSA.
- O formato de uma mensagem contendo uma assinatura digital ou um envelope digital PKCS#1 está definido no PKCS#7.
- Esta norma também inclui uma sintaxe para chaves RSA que é compatível com a norma X.509 [MYERS 1999]. Assim, utilizando esta norma é possível interligar aplicações PKI baseadas no RSA [RSA 03].

O PKCS#3 Padrão de Normalização de chave Diffie-Hellman (*Di_E-Hellman Key Agreement Standard*) normaliza a utilização do protocolo de acordo de chaves Diffie-Hellman no estabelecimento de chaves secretas (de sessão).

Este protocolo permite a dois usuários adotarem uma chave secreta, sobre um canal inseguro, sem trocarem informação que permita a um intruso obter essa mesma chave.

O PKCS#5 Senha baseada no padrão de Criptografia (*Password-Based Encryption Standard*) : Descreve um método para cifrar um arranjo de bytes utilizando uma chave secreta calculada a partir de um *password (Password-Based Encryption* ou PBE).

Destina-se à proteção de chaves privadas em situações que exijam a sua transferência. Isto pode ser necessário, por exemplo, quando as chaves são geradas pela CA (*Certificate Authentication*), e não pelo utilizador; ou quando o utilizador necessita transferir a chave para outra máquina. A cifragem utilizada baseia-se no algoritmo simétrico DES, existem outros esquemas de cifragem mais robustos.

O PKCS#6 Extensão de Certificação Padrão (*Extended-Certificate Syntax Standard*) estende a definição de certificados X.509 permitindo a associação de outros atributos à entidade titular do certificado.

O PKCS#7 Sintaxe para mensagens Criptográficas (*Cryptographic Message Syntax Standard*) define uma sintaxe para mensagens criptográficas, nomeadamente assinaturas digitais e envelopes digitais.

Esta sintaxe admite recursividade, isto é, pode haver uma assinatura digital de um envelope digital. No caso das assinaturas digitais, permite a associação de atributos de natureza arbitrária aos dados assinados.

O PKCS#8 Sintaxe para Informações relativas a chaves privadas (*Private-Key Information Syntax Standard*) define uma sintaxe para informação relativa à chaves privadas: o valor da chave, o algoritmo correspondente e um conjunto de atributos associados.

Define também uma sintaxe para chaves cifradas recorrendo às técnicas PBE definidas no PKCS#5.

A norma **PKCS#9** lista alguns dos atributos que podem ser associados a uma chave privada, conforme explicado na norma PKCS#6.

O PKCS#9 Seleção do tipo de atributo (*Selected Attribute Types)* lista diversos atributos que podem ser incluídos num certificado X.509.

Um exemplo de um atributo definido nestas normas é o endereço de e-mail do titular. Este atributo é bastante utilizado.

O PKCS#10 Sintaxe para pedidos de certificação (*Certification Request Syntax Standard*) define uma sintâxe para pedidos de certificação. Um pedido de certificação inclui:

- Os atributos de identificação do futuro titular do certificado;

- Outros atributos o endereço da entidade que faz a requisição para que lhe seja enviado o certificado.
- A chave pública a incluir no certificado;
- Uma assinatura digital do pedido que simultaneamente demonstra o conhecimento da chave privada e assegura a integridade da mensagem.

Pretende-se que um pedido deste tipo forneça à CA (*Certificate Authentication*) toda a informação necessária para gerar o certificado. Note-se, no entanto, que existem outros aspectos a se ter em conta no processo de certificação, nomeadamente a prova de identidade que tem de ser fornecida pelo titular do certificado.

O PKCS #11 Interface padrão de criptografia para Token (*Cryptographic Token Interface Standard*) descreve a interface de programação chamada “Cryptoki" utilizada para operações criptográficas em hardwares: tokens, smart cards. Esta norma é popular e se utiliza para prover o suporte aos tokens.

O PKCS#12 Sintâxe para transferência de informação de identificação pessoal *(Personal Information Exchange Syntax*) descreve uma sintaxe para a transferência de informação de identificação pessoal, incluindo chaves privadas, certificados, chaves secretas e extensões.

Este padrão define o formato para armazenamento e transporte de chaves privadas, certificados, entre outros. Suporta a transferência de informação pessoal em diferentes condições de manutenção da privacidade e integridade.

O grau de segurança mais elevado prevê a utilização de assinaturas digitais e cifras assimétricas para proteção da informação. Isto implica a utilização de certificados e pares de chaves associados às plataformas de origem e destino, entre as quais se transfere a informação. Um nível de segurança intermédio prevê a utilização de PBE para proteção dos segredos.

Esta norma é uma extensão do PKCS#8 para transferência de informação de identificação pessoal.

?

O PKCS #13 Padrão de Curva eliptica de criptografia (*Elliptic Curve Cryptography Standard)* padronização de algoritmos de criptografia baseados em

curvas elípticas incluindo o formato, a geração de validação de chaves, assinaturas digitais e etc.

Finalmente, **O PKCS #15 -** Informação do Formato padrão de Token *(Cryptographic Token Information Format Standard) é* o padrão que define o uso da tecnologia de criptografia baseada em tokens. O PKCS#1 versão 2.1 [RSA 06] estabelece algumas normas na base de cálculos do RSA já vista anteriormente, o método de cifragem e decifração, assinaturas digitais, e padrões para gerenciamento e armazenamento de chaves.

Neste trabalho de dissertação de mestrado, o interesse principal é no PKCS#11 pois é um padrão de criptografia para token e hardware, por esse motivo apresentamos mais detalhes na próxima sessão.

3.4 Especificação do PKCS#11

O algoritmo do PKCS#11 da RSA foi implementado em linguagem C [RSA 06].

A maioria das chaves utilizadas nos dias de hoje tem 1024 bits de comprimento [RSA 06] e quando a implementação é em hardware, usa-se a norma PKCS#11, que é o mais utilizado em interfaces API (*Application Programming Interface)* para módulo criptográfico. Este padrão especifica uma interface (API), chamado Cryptoki, para dispositivos que fazem segurança usando criptografia e executam funções de criptografia usando uma chave que é chamada de *crypto*-chave.

O padrão usa o ANSI (Padrão de Nacional americano para linguagens de programação – C) linguagem de programação de C e são providas funções tipicamente usando cabeçalho de linguagem C usando uma biblioteca chamada Cryptoki.

Cryptoki é planejado para dispositivos de criptografia associados com um único usuário, assim são omitidas algumas características que poderiam ser incluídas em uma interface de propósito geral. Por exemplo, Cryptoki não tem usuários múltiplos e distintos.

No CD em anexo a dissertação pode-se verificar as especificações do PKCS#11 v2.11, o qual está dividido em 4 arquivos; conforme se visualiza na Tabela 3.3.

Tabela 3.3 – Arquivos especificações PKCS#11

Nome arquivo	**Descrição**
cryptoki.h	Inclui diretivas para construir bibliotecas para aplicações para Win32.
pkcs11.h	Definição de macros para conversão e empacotamento.
pkcs11f.h	Contém funções e protótipos
pkcs11t.h	Definição de macros e as convenções de estrutura

Ver detalhes da implementação em C que mostra os códigos fontes da biblioteca PKCS#11 (Cryptoki.h, pkcs11.h, pkcs11f.h, pkcs11t.h), que aparecem no CD que acompanha a dissertação.

3.5 Funcionamento do padrão PKCS#11

A Figura 4.1 mostra a abertura de sessões conforme o login de usuário no Sistema Operacional. Quando o usuário faz a autenticação no sistema através de um login e senha, é aberta uma sessão, na qual o usuário tem permissões de leitura e escrita e, finalmente, quando o usuário faz um logout, ou seja sai do login, a sessão é fechada.

Pode-se verificar que há 3 tipos de funções quando a sessão é aberta: operações de leitura e escrita (R/W) do sistema operacional, R/W públicas e R/W do usuário. Quando é efetuado o logout, a sessão é fechada e os dispositivos são removidos.

Também é possível verificar que no padrão PKCS#11 existe um dispositivo chamado de slot que pode ser usado para transmitir um dado. A seguir tem-se o significado de alguns itens necessários para que o padrão PKCS#11 possa funcionar

corretamente.

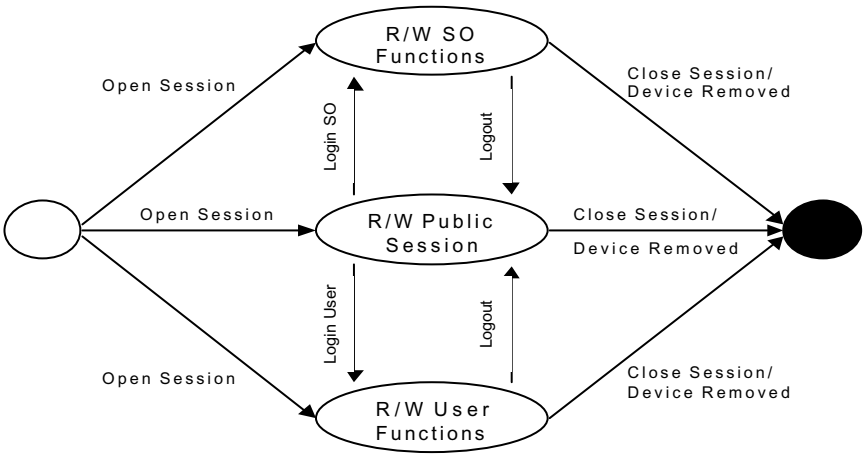

Figura 4.1 – Estados Sessão Leitura/Escrita [RSA 01]

- Cryptoki provê uma interface a um ou mais dispositivos de criptografia que são ativos no sistema por vários " slots ". Cada slot corresponde a um leitor físico ou a uma interface de dispositivo que pode conter um token.
- Um token está tipicamente presente em um slot.
- A visão lógica de Cryptoki de um token é um dispositivo que armazena objetos e pode executar funções de criptografia.

Se um usuário normal foi autenticado ao TOKEN, então a aplicação tem acesso de leitura e escrita.

A Figura 4.2 mostra duas aplicações acessando a biblioteca Cryptoki do padrão PKCS#11 do RSA, logo após é acessado um dispositivo que contém slots e esses slots por sua vez possuem um token.

É realizada a chamada à biblioteca Cryptoki e logo após, a chamada de um slot para comunicação entre computadores. O slot por sua vez faz a chamada ao token e nesse momento é aberta uma sessão entre as duas aplicações em dois computadores diferentes. O token está implícito em um slot, o slot pode ser uma porta serial, paralela, USB ou uma placa de rede.

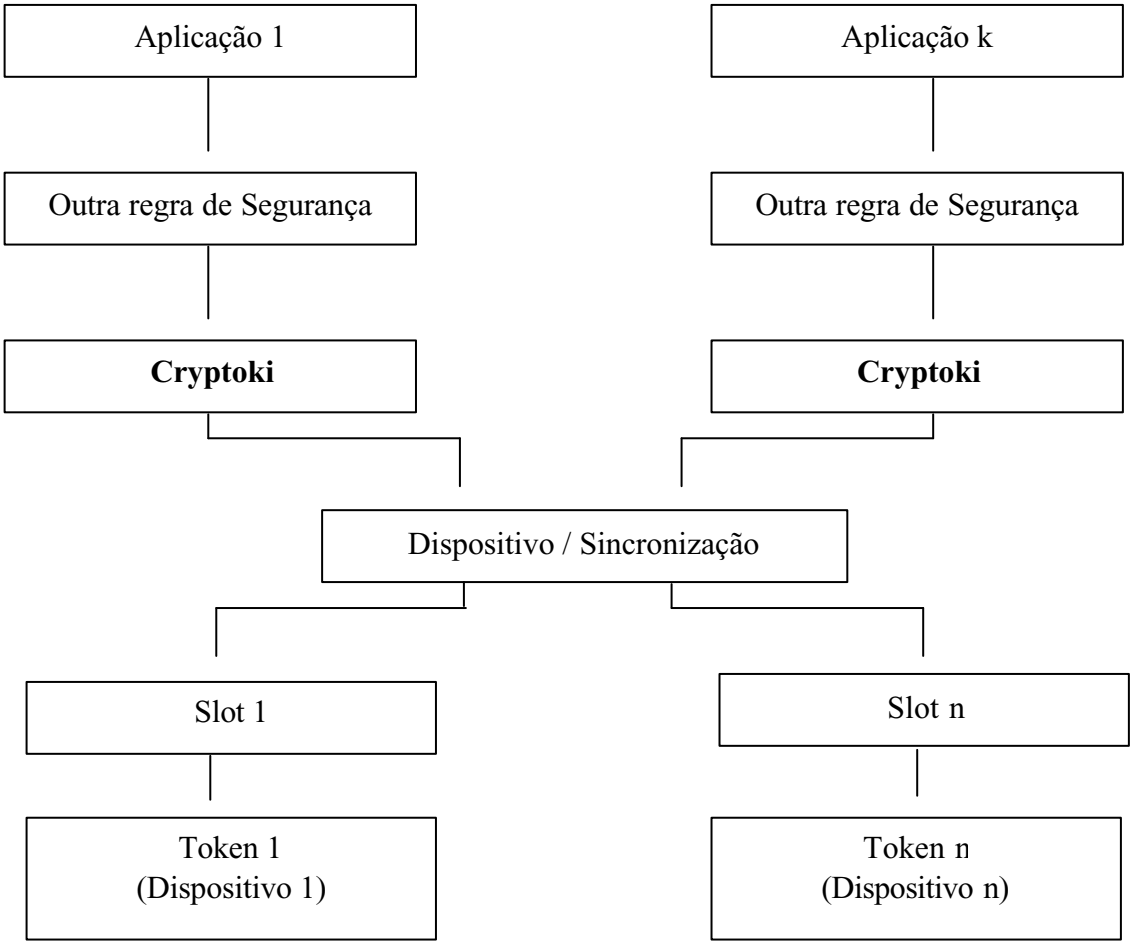

Figura 4.2 – Modelo da biblioteca Cryptoki [RSA 01]

A Figura 4.3 mostra a seqüência de processos conforme o padrão de criptografia PKCS#11. Uma mensagem de 8 bits, por exemplo, irá passar por todos os itens do padrão PKCS#11, a começar pelo "C_Initialize" que inicia a biblioteca cryptoki utilizada pelo padrão. Como o código implementado está em VHDL não é possível iniciar a biblioteca Cryptoki, porque está em software.

A mensagem de 8 bits que está no PC A irá para o PC B, obedecendo o padrão PKCS#11, passando desde C_Initialize até o item que efetua a criptografia da mensagem de 8 bits ou seja quando a mensagem chegar ao PC B estará criptografada no padrão PKCS#11.

Depois que "C_initialize" é carregado, a próxima etapa é o "C_Getslotlist" função pela qual se irá obter a lista de informações dos Slots. Nessa etapa todos os slots do computador serão listados (desde o slot serial, paralelo, USB, placa de rede, assim

como todos slots existentes no computador). Após obter a lista de slots, a função "C_Gettokeninfo" obtém informações sobre um token e "C_Inittoken" inicializa um token. Após inicializar um token, a função "C_InitPIN" inicializa um usuário normal de acordo com o usuário que efetuou o login no sistema.

A função "C_Opensession" abre uma conexão entre uma aplicação e um token. Após a abertura da sessão, a função "C_Login" irá fazer um login a um token e estará com a função de carregar a mensagem ao seu destino, mas antes é necessário iniciar as operações de criptografia que serão executadas pela função "C_EncryptInit". A função "C_Encrypt" irá cifrar a mensagem e após a cifragem a mensagem será enviada para um outro micro (por exemplo, a máquina B).

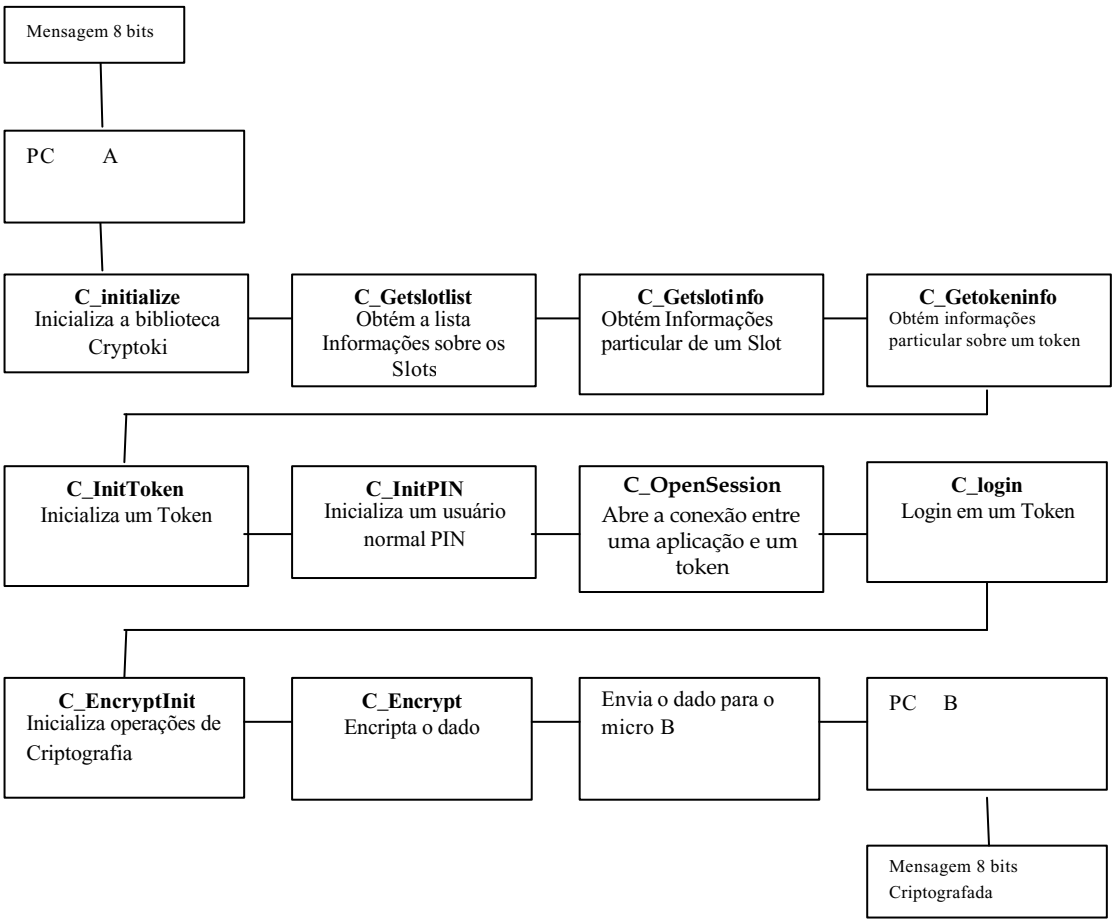

Figura 4.3 – Seqüência de operações e criptografia no padrão PKCS#11

A Figura 4.4 mostra a seqüência de processos conforme o padrão de decriptografia PKCS#11. Uma mensagem de 8 bits, por exemplo, irá passar por todos os itens do padrão PKCS#11, para que o algoritmo possa estar de acordo com o padrão PKCS#11 é necessário que a mensagem passe por todos os itens do padrão, a começar pelo "C_Initialize" que inicia a biblioteca cryptoki utilizada pelo padrão. Como o código implementado está em VHDL não é possível iniciar a biblioteca Cryptoki.

A mensagem de 8 bits que está no PC B irá para o PC A, obedecendo o padrão PKCS#11, passando desde C_Initialize até o item que efetua a criptografia da mensagem de 8 bits ou seja quando a mensagem chegar ao PC A estará decriptografada no padrão PKCS#11.

Depois que "C_initialize" é carregado, a próxima etapa é o "C_Getslotlist" função pela qual se irá obter a lista de informações dos Slots. Nessa etapa todos os slots do computador serão listados (desde o slot serial, paralelo, USB, placa de rede, assim como todos slots existentes no computador). Após obter a lista de slots, a função "C_Gettokeninfo" obtém informações sobre um token e "C_Inittoken" inicializa um token. Após inicializar um token, a função "C_InitPIN" inicializa um usuário normal de acordo com o usuário que efetuou o login no sistema. A função "C_Opensession" abre uma conexão entre uma aplicação e um token. Após a abertura da sessão, a função "C_Login" irá fazer um login a um token e estará com a função de carregar a mensagem ao seu destino, mas antes é necessário iniciar as operações de decriptografia que serão executadas pela função "C_DecryptInit". A função "C_Decrypt" irá decifrar a mensagem e após a decifragem a mensagem será enviada para um outro micro (por exemplo, a máquina A).

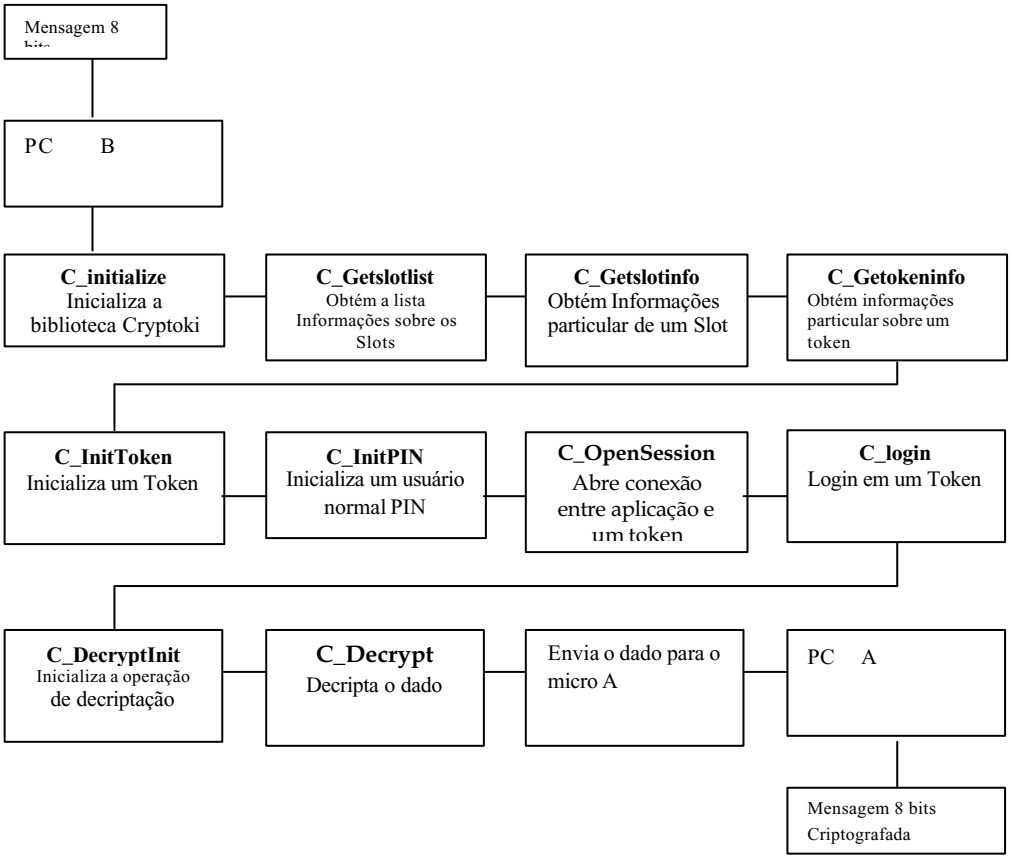

Figura 4.4 – Seqüência de operações e decriptografia no padrão PKCS#11

3.6 Importância do PKCS#11 em hardware

O padrão PKCS#11 é utilizado para operações criptográficas em hardware (tokens, smart cards e etc) para prover suporte aos tokens.

Conforme visto no capítulo 2, os processadores de rede precisam de um módulo de segurança para o processamento de pacotes, para prover maior segurança na rede e com o módulo de segurança baseado no PKCS#11, o dado poderá ser criptografado no processador de rede e quando o dado chegar no destino será decriptografado usando o padrão.

Nesta dissertação será utilizado o PKCS#11 porque é um padrão criado para hardware podendo ser utilizado em rede, baseado em segurança que utilizam como base os algoritmos de criptografia RSA.

O PKCS#11 é baseado no padrão que fornece recomendações para a execução de criptografia baseada em chave pública e o algoritmo utilizado é o RSA.

Devido a seu pioneirismo e simplicidade do algoritmo, tornaram-se padrão de fato em PKIs .

Processar criptografia do algoritmo inclui a manipulação de dados que necessitaria ser feita na totalidade dos pacotes, tais como manter a confidencialidade e a integridade.

No capítulo 2, foi possível observar que existem processadores de rede com segurança implementada, por exemplo o IXP2800 da Intel, que tem algoritmos de criptografia necessários para manter segurança.

A segurança da informação se torna mais importante no mundo de hoje, e é necessário que o equipamento de *networking* permita funções de criptografia. O processador de rede pode conter características que permitam a adição da funcionalidade da segurança com vantagens significativas.

Com o aumento crescente dos negócios e dos indivíduos em redes de computadores, há um aumento correspondente na importância da segurança da informação. Este aumento requer que algumas funcionalidades de segurança, tal como confidencialidade e a integridade dos dados, sejam incorporados em processadores de redes.

O grupo de arquitetura de Sistemas Computacionais do UNIVEM (www.fundanet.br/ppgcc) está projetando um processador de rede e prototipação em FPGAs. Esse processador conterá primitivas de segurança, motivo pelo qual o presente trabalho colaborará com o padrão PKCS#11, fazendo com que esse processador possa em uma rede se comunicar conforme o padrão.

A implementação de um módulo PKCS#11 usando Hardware, VHDL e FPGA apresenta as seguintes vantagens [MORENO 2005]:

a) Maior velocidade: o tempo gasto para criptografar utilizando a implementação em hardware do algoritmo RSA seguindo o PKCS#11 é mais rápido do que a respectiva implementação em Software;

b) Maior confiabilidade do sistema, item chave para desenvolvimento de aplicação de tempo real;

c) Em sistemas seqüenciais, o detalhamento da lógica de controle é realizado pelas ferramentas de automação do projeto, o que evita a trabalhosa e limitada aplicação das técnicas manuais tradicionais;

d) O objetivo do projeto fica mais claro que na representação por esquemáticos, nos quais a implementação se sobrepõe à intenção do projeto;

e) O volume de documentação diminui, já que um código bem comentado em VHDL substitui com vantagens o esquemático e a descrição funcional do sistema;

f) O projeto ganha portabilidade, já que pode ser compilado em qualquer ferramenta e para qualquer tecnologia. É comum, na indústria, o uso de FPGAs e CPLDs para produções iniciais ou de menores escalas em projetos que posteriormente possam ser implementados em ASICs. Todas as implementações podem usar o mesmo código VHDL.

A implementação do PKCS#11 em FPGA apresenta as seguintes vantagens [MORENO 2003]:

a) Possibilidade de desenvolvimento conjunto de hardware e software, sem interdependência, de modo a aumentar a velocidade com que o produto final chega à linha de produção;

b) Maior confiabilidade do sistema, item chave para desenvolvimento de tempo real ;

c) Maior velocidade de chegada do produto ao mercado consumidor, pela detecção antecipada de problemas quanto ao hardware do sistema.

3.7 Considerações finais do Capítulo

Este capítulo apresentou os conceitos básicos de segurança e o padrão PKCS#11.

Existem 12 especificações para o padrão PKCS.

O funcionamento das especificações do padrão PKCS#11 é possível usando uma interface chamada “Criptoki”, usando software.

É possível visualizar a seqüência de operações e criptografia no padrão PKCS#11.

É importante o PKCS#11 em hardware para prover segurança aos dispositivos.

4. O PADRÃO PKCS#11 EM HARDWARE

O padrão PKCS#11 é utilizado para operações criptográficas em hardware. O PKCS#11 é baseado no padrão que fornece recomendações para a execução de criptografia baseada em chave pública e o algoritmo é o RSA. Este capítulo apresenta uma versão em hardware do PKCS#11.

4.1 Descrição da Máquina de Estados Finitos padrão PKCS#11

A Figura 4.5 mostra os estados da máquina de estado finito que implementa em hardware o padrão PKCS#11, bem com a transição de um estado para outro. Em nossa implementação, os estados variam do estado 0 até o estado 6, ou seja, a máquina de estado finito RSA padrão PKCS#11 é constituída de 7 estados que estão relacionados entre si. Em cada momento dependendo de um evento seja baseado no clock ou no estado da máquina, um estado estará em funcionamento e passará para o estado seguinte assim que o estado anterior for completado e a respectiva condição for satisfeita.

No projeto, a máquina de estado finito padrão PKCS#11 tem início no estado 0 (zero). Quando a máquina está em estado 0 (zero), ou seja, a variável estado recebe o valor 0 (zero) e a variável "CH" recebe o valor 0 (zero). Assim, no estado 0 a máquina é inicializada.

No estado 1 são atribuídos os valores dos tokens conforme escolha do usuário, ou seja, a variável VAR_CH irá receber o valor do token escolhido, e a variável ESTADO recebe DOIS. Assim o estado da máquina passará para o próximo estado que corresponde a 2. No estado 2 a informação do token é enviada serialmente pela porta serial do computador.

No estado 3, o dado do token é enviado para a o módulo de criptografia que implementa o algoritmo RSA, é quando o dado é liberado para criptografia e a variável S_P_CRIP recebe o valor 1 (um) indicando que acabou a criptografia. A variável estado recebe o valor 4 indicando para passar para o estado seguinte.

No estado 4, o dado criptografado é enviado para a porta serial do computador ou seja, o dado que foi criptografado no estado 3 agora é enviado pela porta serial no

estado 4. A variável “Lib_est_quatro” receberá o valor 1 (um) indicando que passou pelo estado quatro e o valor D_OUT receberá o valor de S_D_CRIP que corresponde ao dado criptografado. Quando “Lib_est_quatro” recebe o valor 1 (um) indica que já enviou o dado pela porta serial do micro.

No estado 5, o dado libera a função de decriptografia RSA e se a variável S_P_DECRIP for igual a 1 (um) indica que foi realizada a decriptografia e a variável estado recebe o valor 6 o que indicando que a máquina irá para o próximo estado que corresponde ao estado 6.

No estado 6, o dado decriptografado será transmitido pela porta serial do micro, D_OUT receberá o valor de S_D_DECRIP que corresponde ao dado decriptografado pelo módulo RSA. Quando isso ocorre, “Lib_Est_SEIS” recebe o valor 1 (um) indicando que o dado decriptografado foi enviado para o outro computador pela porta serial. No estado 5 e 6 pode-se visualizar que é realizado a decifragem, mesmo aos após a realização de cifragem, isso ocorre para validar os testes, porque foi implementado para realizar teste em um único FPGA, podendo em projetos futuros usar dois FPGA uma para cifrar e outro para decifrar.

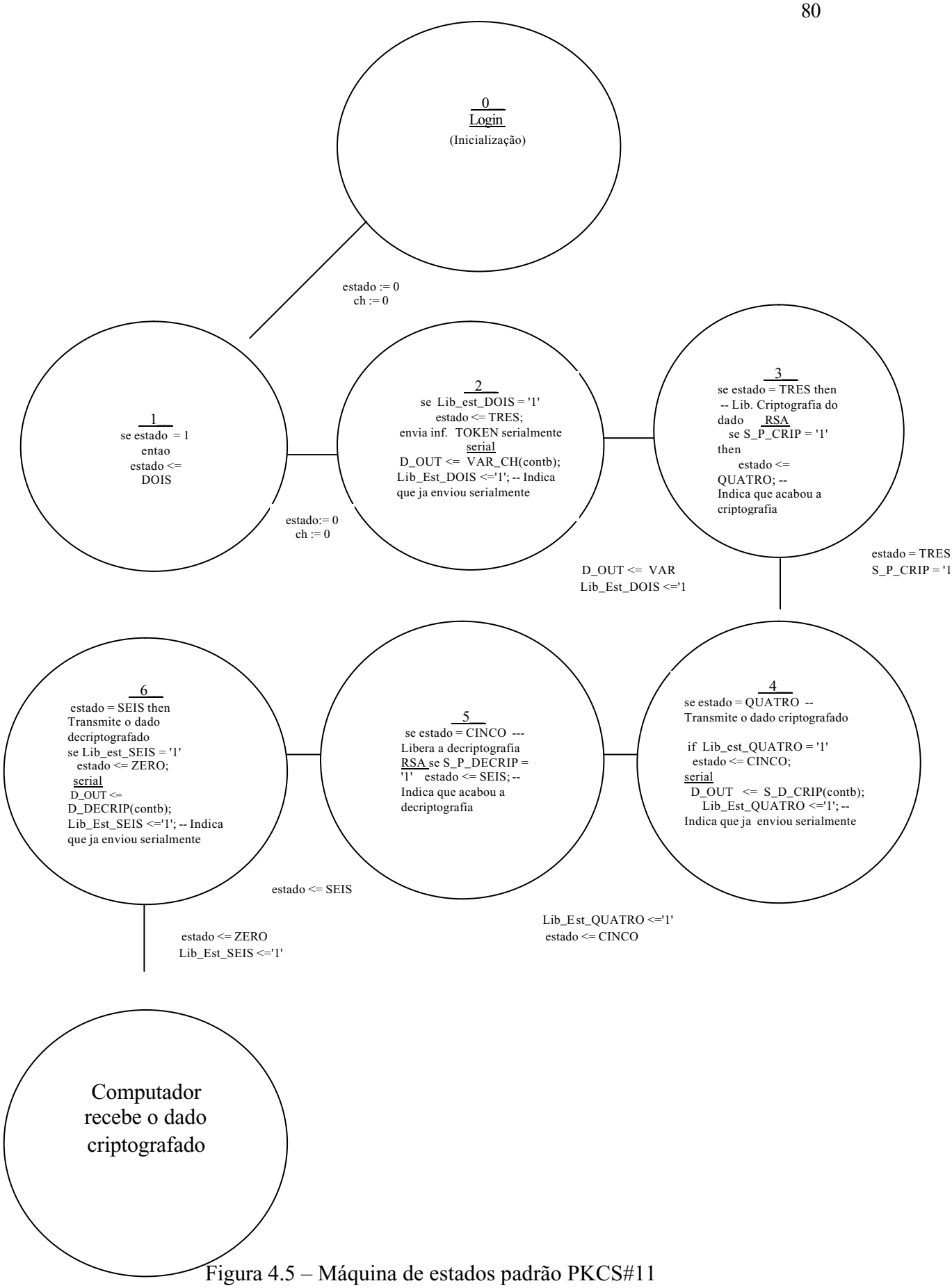

Figura 4.5 – Máquina de estados padrão PKCS#11

4.2 - Descrição detalhada do código em VHDL da máquina de estados finito no padrão PKCS#11 implementada em hardware

A Tabela 4.1 mostra os std_logic usados como entrada e saída definindo a finalidade do uso de cada um deles.

Tabela 4.1 – Portas da máquina serial

Nome std_logic	Descrição
CH	O CH é um vetor de entrada usado para armazenamento do número referente ao token.
CLK	Clock
RESET	O RESET é usado para zerar a Máquina de estados
D_OUT	O D_OUT é usado para atribuir o valor de saída do dado criptografado para envio pela porta serial (slot de saída)

A Figura 4.6 mostra a arquitetura do módulo serial da máquina de estados no padrão PKCS#11.

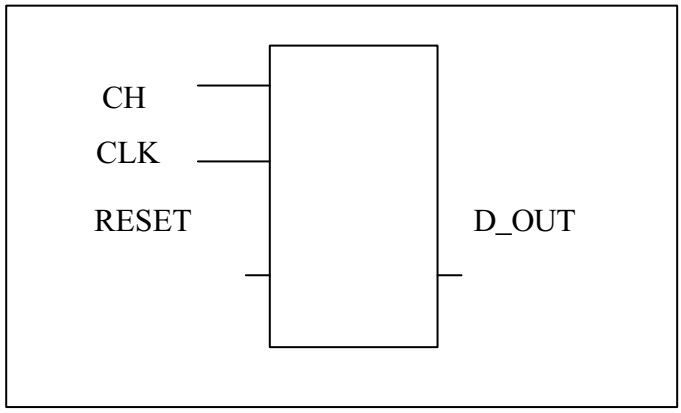

Figura 4.6 - Arquitetura do módulo da máquina serial

As entradas e saídas são:

? CH é um vetor de entrada usado para armazenamento do token, no qual é formado por dois bytes ou seja 16 bits, usados no início da execução da máquina de estado de acordo com a escolha do usuário. Na implementação esses bytes poderão ser AA, AB, BC, CD, DE ou EF.

- CLK é o clock definido para a máquina de estado.
- RESET é usado para dar inicio à máquina de estados. Quando o reset estiver com o valor “zero” a máquina está parada; quando receber o valor 1, a máquina começa a funcionar.
- D_OUT é usado para atribuir o valor de saída do dado criptografado que após a escolha do token e a passagem do dado original pelo módulo de criptografia RSA o dado será atribuído a D_OUT. Nesse momento o dado criptografado estará contido em D_OUT.

A Tabela 4.2 mostra as portas de saída criptografada segundo o algoritmo RSA. o módulo de criptografia RSA tem portas de saída onde cada uma tem uma finalidade específica:

- No D_CRIP é atribuído o dado criptografado.
- No D_DECRIP é atribuído o dado decriptografado. Essas duas portas de saída são importantes pois nelas se atribuem os valores criptografado e decriptografado.
- No P_CRIP é atribuído o valor 1 (um) quando o módulo RSA faz a criptografia, dessa forma é possível visualizar na simulação exatamente quando foi realizada a criptografia. Enquando não for realizada a criptografia, o valor padrão de P_CRIP é 0 (zero).
- No P_DECRIP é atribuído o valor 1 (um) quando o módulo RSA faz a decriptografia, dessa forma é possível visualizar na simulação exatamente quando foi realizada a decriptografia. Enquando não for realizada a decriptografia, o valor padrão de P_DECRIP é 0 (zero) .

Tabela 4.2 – Portas de saída criptografada do RSA

Nome std_logic	**Descrição**
D_CRIP	No D_CRIP é atribuído o dado criptografado
D_DECRIP	No D_DECRIP é atribuído o dado decriptografado
P_CRIP	No P_CRIP é atribuído o valor “1” quando é realizado a criptografia pelo módulo RSA
P_DECRIP	No D_DECRIP é atribuído o valor “1” quando é realizado a decriptografia pelo módulo RSA

A Tabela 4.3 mostra as constantes que são utilizadas para a respectiva definição do token, ou seja cada constante irá receber um valor conforme o valor de CH que pode ser 0,1,2,3,4 ou 5 conforme a escolha do usuário.

A escolha sempre será de dois bytes, porque foi definido o tamanho do vetor para receber dois bytes, o quais serão criptografados pelo módulo RSA, a escolha do token é definida por CH. Neste projeto foi escolhido dois bytes para teste, que corresponde a CH, mas poderia ser escolhido outros dois bytes diferentes de CH. Um token escolhido pelo usuário faz com que seja atribuído dois bytes para criptografia no módulo RSA.

Por exemplo, o valor de CH igual a 1 indica que o usuário seleciona uma operação que fará uma concatenação entre as contantes token0 e token1 e atribuído à variável VAR_CH que receberá os bytes correspodentes à AB.

Tabela 4.3 – Definição das configurações para o TOKEN

Nome	**Descrição**	**Bits**	**Byte**
TOKEN0	Vetor de 8 bits para atribuir valor do token	01100101	A
TOKEN1	Vetor de 8 bits para atribuir valor do token	01100110	B
TOKEN2	Vetor de 8 bits para atribuir valor do token	01100111	C
TOKEN3	Vetor de 8 bits para atribuir valor do token	01101000	D
TOKEN4	Vetor de 8 bits para atribuir valor do token	01101001	E
TOKEN5	Vetor de 8 bits para atribuir valor do token	01101010	F

A Tabela 4.4 mostra as constantes que são utilizadas para fazer a criptografia e decriptografia, baseando-se na formula do RSA que é $C = M^e \bmod n$ usado para criptografia e $M = C^d \bmod n$ usado para decriptografia.

O cálculo do RSA é realizado usando os vetores conforme pode-se visualizar na Tabela 4.4. O vetor E é o expoente usado no cálculo para a realização da criptografia RSA. O vetor N é utilizado para o cálculo e realização da criptografia RSA. Usando o valor N é possível calcular o MOD, sendo que N é o valor atribuído para calcular o MOD.

O vetor D é usado para a realização da decriptografia em RSA, ou seja D é o expoente utilizado para a decriptografia.

A coluna de bits da Tabela 4.4 mostra os bits correspondentes aos valores atribuídos para cada vetor no cálculo da criptografia RSA e também os valores em hexadecimal e decimal.

Como se pode observar na coluna da Tabela 4.4 os valores já estão definidos neste projeto, onde está o valor em decimal, esse valor poderá ser alterado a qualquer momento, definindo assim os valores das chaves para o calculo da criptografia usando o módulo RSA.

Tabela 4.4 - Constantes para o RSA

Nome	**Descrição**	**Bits**	**Valor em Hexadecimal**	**Valor em Decimal**
E	Vetor de 17 bits utilizado para atribuição do valor do Expoente para ser usado na criptografia do RSA	01011100101011101	0B95D	47453
N	Vetor de 24 bits utilizado para atribuição do valor de n ou seja o n será usado para calculo do mod usado na criptografia do RSA	001100011110001000111111	31E23F	3269183
D	Vetor de 17 bits usado como expoente na decriptação.	01110101010101101	0EAAD	60077

A Tabela 4.5 mostra os vetores e sinais usados na máquina de estado finito do RSA padrão PKCS#11.

Tabela 4.5 - Vetores e Sinais utilizados na Máquina de estados finito

Vetor	Descrição	
Estado	Vetor de 3 bits utilizado para definição do estado da máquina de estados finito	O vetor estado define o estado da máquina cada estado que a máquina no momento de sua execução é atribuído o numero do estado para o vetor estado, sendo possível vizualizar na simulação o valor de estado da máquina e o estado que a máquina se encontra.
VAR_CH	Vetor de 16 bits usado para guardar informações do Token	O vetor VAR_CH é o vetor que receberá o valor do token, ou seja cada token é de um byte, serão atribuído dois bytes para o token conforme a escolha do usuário

		em CH conforme visto na Tabela 2.18, VAR_CH receberá dois bytes por esse motivo VAR_CH é um vetor de 16 bits.
S_P_CRIP	Sinal Libera a criptografia segundo um determinado algoritmo, em nosso caso foi escolhido o RSA.	S_P_CRIP receberá o valor 1 (um) que indicará que o módulo RSA poderá realizar a criptografia.
S_P_DECRIP	Sinal Libera a decriptografia segundo um determinado algoritmo, em nosso caso foi escolhido o RSA.	S_P_DECRIP receberá o valor 1 (um) que indicará que o módulo RSA poderá realizar a decriptografia.
S_D_CRIP	Vetor de 24 bits utilizado como registrador interno do dado criptografado	S_D_CRIP receberá o dado criptografado que funcionará como um registrado interno contendo o dado criptografado.
S_D_DECRIP	Vetor de 16 bits utilizado como registrador interno do dado decriptografado	S_D_DECRIP receberá o dado decriptografado que funcionará como um registrado interno contendo o dado decriptografado.
Lib_est_DOIS	Sinal que indica que passou pelo estado 2	Lib_est_DOIS receberá o valor 1 (um) no que indica que a máquina passou pelo estado 2 da máquina.
Lib_est_QUATRO	Sinal que indica que passou pelo estado 4	Lib_est_QUATRO receberá o valor 1 (um) no que indica que a máquina passou pelo estado 4 da máquina.
Lib_est_SEIS	Sinal que indica que passou pelo estado 6	Lib_est_SEIS receberá o valor 1 (um) no que indica que a máquina passou pelo estado 6 da máquina.

A Tabela 4.6 mostra sinais usados na transmissão serial do dado criptografado..

Tabela 4.6 - Transmissão Serial

Nomes	**Descrição**
S_CLK	Clock da transmissão Serial
Conta	Sinal usado entre o valor 0 até 444444, para sincronizar com a freqüência do FPGA.
Contb	Sinal usado na transmissão serial como contador, se estado = 2 e menor que 15 envia o dado criptografado para transmissão serial

A Tabela 4.7 mostra as variáveis e vetores usados na criptografia usando o algoritmo RSA, o módulo do RSA contém esses vetores e variáveis.

Tabela 4.7 - Encriptação RSA

Nomes	**Descrição**	**Função**
Temp	Vetor de 32 bits	O vetor temp é usado como temporário para o módulo RSA.
Flag	Vetor de 3 bits	Flag é usado para como vetor de 3 bits no qual é responsável por controlar quando é realizado a criptografia.
A	Vetor de 32 bits	A é um vetor usado para a realização da criptografia.
B	Vetor de 32 bits	B é um vetor usados para a realização da criptografia.
R	Vetor de 32 bits	R é um vetor usado para a realização da criptografia.
I	Variável tipo Integer	I é um contador
J	Variável tipo Integer	J é um contador

4.3 Resultados Máquina de estados Finito no padrão PKCS#11

Nesta seção pode-se visualizar os resultados obtidos através da Máquina de estados finito no padrão PKCS#11.

A Figura 4.7 mostra os resultados de simulação obtidos da máquina de estados que implementa todas as operações indicadas no padrão PKCS#11 para realizar uma conexão segura. Nesta Figura pode-se observar que há seis estados. Em cada estado a máquina executa as funções próprias, conforme indicado na figura e testados. Pode-se notar que no estado 1 é mostrado que VAR_CH está no valor 6768, que corresponde ao dado que será criptografado, no estado 2 é criptografado o dado usando o RSA e enviado via serial usando o D_OUT, no S_D_CRIP tem-se o dado criptografado que corresponde a 1A68FE que está em hexadecimal. Os valores atribuídos representam um exemplo neste projeto.

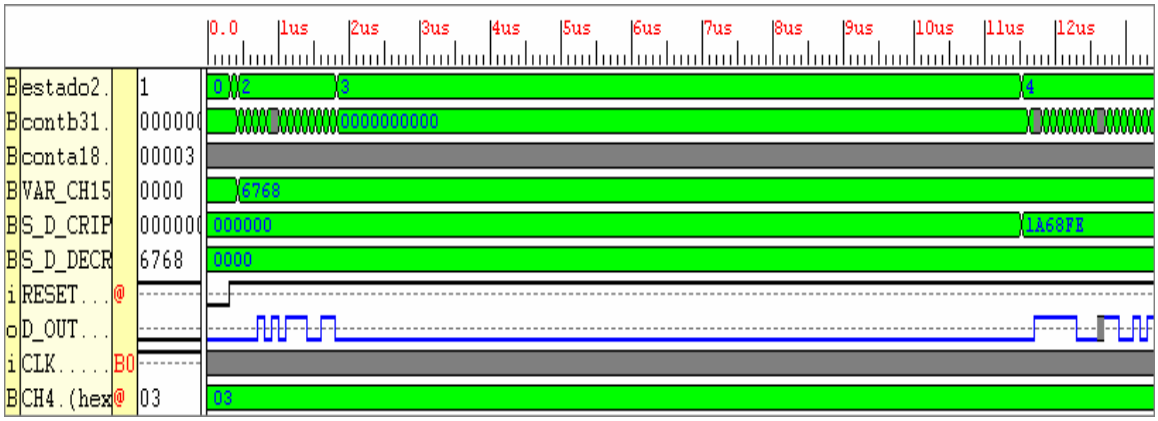

Figura 4.7 – Simulação da Máquina de estados Finito

A variável “conta” corresponde a um contador que irá varia de 1 até 83333 e que sincroniza o clock (CLK) da máquina principal com o clock (s_clk) da saída do dado criptografado pela porta serial, denominado slot pelo padrão PKCS#11 e em nosso projeto é o slot default. Quando o valor do contador conta for igual a 166666, a variável conta recebe novamente o valor 0, para sincronizar com 50 Megahertz do FPGA que é a frequência de funcionando usado no teste.

O reset é usado para zerar a máquina quando o reset está com o valor 0. Quando recebe o valor 1, dá inicio ao funcionamento da máquina de estados finito.

O vetor CH, de tamanho 4, corresponde ao Token, onde pode-se determinar qual será o valor do token a ser usado. Quando se escolhe o 3, por exemplo corresponde ao valor 67 e 68 que na tabela ASCII corresponde aos bytes C e D, esse valores são exemplos de utilização no teste, podendo ser usado qualquer outro valor.

O sinal S_D_DECRIP recebe o valor decriptografado quando o estado da máquina for igual a 5.

O contb é um contador que é usado para indicar que o dado foi enviando pela serial. Quando o valor da variável “contb” for igual a 15, o número 15 é apenas um contador, e quando chegar a esse número a variável contb é zerada, e Lib_Est_DOIS recebe o valor 1 indicando que o dado criptografado foi enviado serialmente. Quando for igual ao valor 23 indica que o dado decriptografado foi enviado pela serial, e o sinal “Lib_est_seis” recebe o valor 1, o número 23 é apenas um contador, e quando “contb” chegar a esse número a variável “Lib_est_seis” receberá o valor 1, indicando que o dado já foi enviado serialmente.

4.4 - O módulo RSA_CRIP

A Figura 4.8 mostra a arquitetura da máquina de estados finito que implementa o padrão PKCS#11. A máquina de estados consiste em diversos módulos, cada um com uma função determinada, a começar pelo token que tem a função de "pegar" um token escolhido pelo usuário. Esse token terá saída em s_token onde aparecerão informações do token.

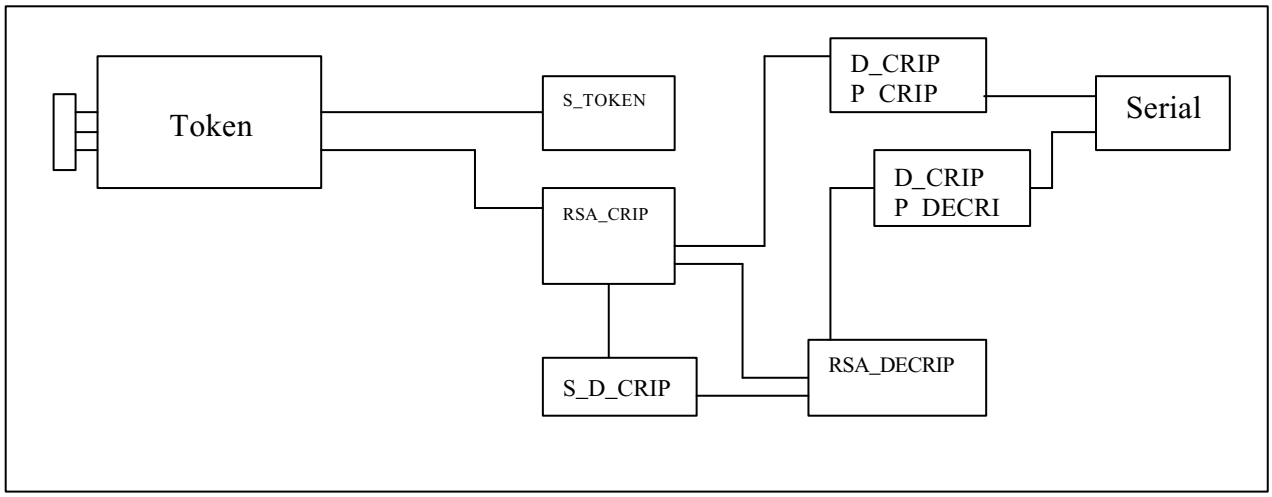

Figura 4.8 – Arquitetura da Máquina de estados Finito RSA no padrão PKCS#11

O módulo RSA_CRIP tem a função de realizar a criptografia baseado no algoritmo RSA, criptografia essa com chaves de tamanho 24 bits usando uma mensagem, inicialmente de 16 bits. Após realizar a criptografia da mensagem, S_D_CRIP recebe o valor "1", que representa que a mensagem foi criptografada. D_CRIP é o dado criptografado e P_CRIP é um sinal indicando que a mensagem já foi criptografada. Após D_CRIP receber o dado criptografado é realizado o mesmo processo e é enviado pela porta serial do computador.

Após realizar a criptografia pelo módulo RSA_CRIP é enviado o dado criptografado para RSA_DECRIP para realizar o processo inverso ou seja pegar o dado criptografado e decriptografar voltando assim ao dado original. Após voltar o dado original é enviado para D_CRIP o dado e P_CRIP recebe o sinal que o dado foi decriptado. Após ser decriptado e estar novamente na forma original é enviado para a porta serial do micro.

A máquina de estado funciona de acordo com o padrão PKCS#11, criptografa e decriptografa e envia o dado pela porta serial do micro, para uma outra máquina conectada no sistema.

Dessa forma é possível transmitir um dado criptografado de um computador para outro, usando somente mecanismos implementados em hardware.

A máquina de estados Finito (*FSM*) que implementa o algoritmo RSA e segue o padrão PKCS#11, consiste em sete estados, sendo o estado inicial 0 e o último estado 6.

A Tabela 4.8 mostra a máquina de estados finito que faz a criptografia RSA usando o padrão PKCS#11. A FSM tem vários estados necessários para o funcionamento da máquina.

Cada estado faz uma função, ou seja cada um deles é responsável pela execução de uma parte da máquina de estados. A principal vantagem da FSM RSA no padrão PKCS#11 é que todo os processos são executados partindo de um dado original, que é criptografado e enviado por um slot, que neste projeto é a porta serial do computador. Importante observar, que a tabela também mostra a relação da função de cada estado com a funcionalidade no padrão PKCS#11.

Tabela 4.8 – Estados da FSM do algoritmo RSA no padrão PKCS#11

Nº do estado	**Descrição do estado**	**Padrão PKCS#11**
0	Inicializa Máquina	Inicializa
1	Leitura do Token	Obtém a lista Informações sobre os Slots. Obtém Informações particular de um Slot. Obtém informações particular sobre um token
2	Envia as Informações do Token	Inicializa um Token. Inicializa um usuário normal PIN. Abre a conexão entre uma aplicação e um token. Login em um Token
3	Libera a Criptografia do dado	Inicializa operações de Criptografia. Encripta o dado
4	Transmite o dado criptografado	Envia o dado Criptografado
5	Libera a decriptografia	Inicializa operações de Decriptografia. Decripta o dado
6	Transmite o dado decriptografado	Envia o dado Decriptografado

O estado 0 da máquina é o estado que inicializa a máquina e corresponde à inicialização dentro do padrão PKCS#11.

O estado 1 faz a leitura do token, que está contido no slot que pode ser porta serial, paralela, USB ou placa de rede. Em nossa implementação o slot padrão (*default*) é o serial, pois foi determinado que o envio do dado criptografado para outro computador e usando a porta serial. Assim a leitura do token é feita dentro da máquina de estados no estado 1, que corresponde ao padrão PKCS#11, pois são obtidas informações sobre um slot e informações sobre um determinado token.

No estado 2 é inicializado o token quando é determinado o valor de CH. (escolhido por nós como teste), mas pode ser qualquer valor. Por exemplo se escolher o número 1, o token será AB que será o dado que será criptografado e decriptografado pela máquina. Nesse momento é determinado o valor do token simulando assim a obtenção de informações de um token de um slot. Assim, o usuário tem uma sessão aberta de acordo com o login ao token. O CH é um vetor de entrada usado para armazenamento do número referente ao token.

O estado 3 inicializa as operações de criptografia usando o módulo que implementa o algoritmo RSA, que encripta o dado.

O estado 4 é responsável por enviar o dado criptografado pela porta serial para outro computador, a transmissão serial é feita bit a bit sendo necessário um tratamento para o envio desse bit pela porta serial. O D_OUT é usado para atribuir o valor de saída do dado criptografado para envio pela porta serial (slot de saída).

O estado 5 inicializa as operações de decriptografia. No nosso caso, usamos novamente o algoritmo RSA, esta operação é realizada pelo módulo RSA que decifra o dado.

O estado 6 é responsável por enviar o dado que foi decriptografado pela porta serial.

Como se pode observar, todos os estados permitem a verificação dentro do padrão PKCS#11, onde cada estado corresponde ao padrão PKCS#11, que foi criado para promover suporte ao token e mecanismos de criptografia.

Quando há um padrão de criptografia é possível que cada vez mais tecnologias sejam adotadas e usadas obedencendo esse padrão.

4.5 Resultados da FSM no padrão PKCS#11

A Tabela 4.9 mostra a porcentagem total de ocupação da máquina de estados finito no FPGA e também a quantidade total de Flip Flops usados pela máquina de estados finito no FPGA.

O Flip Flop é um circuito digital básico que armazena um bit de informação. A saída de um Flip Flop só muda de estado durante a transição de um sinal do clock [MORENO 2003].

Pode-se verificar a quantidade de LUTs (*lookup table*) que são tabelas verdade utilizadas no FPGA, pode-se verificar que com o aumento do número de bits o número de LUTs utilizado pela máquina de estados é maior.

Pode-se verificar o total de IOBs (*Input/Output block*) que são blocos de entrada e saída utilizado na periferia dos FPGAs, são responsáveis pela interface com o ambiente [MORENO 2003]. Interessante observar que a máquina de estados finito usa sempre a mesma quantidade de IOBs, mesmo em FPGAs de marcas e modelos diferentes.

Pode-se verificar que a Virtex II Pro é a FPGA que tem a maior frequência de operação (82.642 Mhz), em segundo lugar aparece a Spartam 3 com 72.001 MHz, em terceiro a Virtex E com 47.492 MHz e por último a Spartam 2E com 41.714 MHz.

Pode-se verificar o tempo da máquina de estados em nanosegundos. O maior tempo utlizado é a Spartam 2E com 23.973 ns, em segundo lugar a VirtexE com 21.056 ns e em terceiro lugar a Spartam 3 com 13.889 ns e por último a Virtex II PRO com 23.200 ns.

Esse tempos são resultado apurado pela ferramenta de síntese do Xilinx.

A baixa porcentagem de Slices da Virtex II Pro representa que o FPGA é de tamanho maior sendo que o percentual utilizado no FPGA usando a FSM é de apenas 1%. Usando a Spartan2E no mesmo projeto, representa 35% da ocupação da FSM no FPGA.

Tabela 4.9 – Resultados da implementação da máquina de estados finito padrão PKCS#11

FPGAs	Slices	Flip Flop	LUTS	IOBs	Máximo Periodo ns	Frequência MHz
xc2x200e - (Spartan2E)	1054 (35%)	518 (11%)	1539 (32%)	44 (32%)	23.973	41.714
xc3s5000-5fg1156(Spartan 3)	1040 (2%)	516 (0%)	1506 (2%)	44 (6%)	13.889	72.001
xcv3200e-8fg1156 (Virtex E)	1052 (2%)	518 (0%)	1532 (2%)	44 (5%)	21.056	47.492
xc2vp100-6ff1704 (Virtex II - Pro)	1030 (1%)	516 (0%)	1523 (2%)	44 (3%)	12.100	82.642

4.6 Resultados Criptografia usando RSA

4.6.1 – Implementação do RSA em software (C)

Nesta seção será apresentam os resultados obtidos do algoritmo RSA implementado em liguagem C.

Foi utilizado para análise dos resultados um computador Pentium IV 1.6 GHZ com 128 de memória RAM e sistema operacional Windows 2000 e o DEV-C++.

A Figura 4.9 mostra o tempo necessário para a geração de chaves usando linguagem C, que foi compilado usando o DEV-C++.

A chave é gerada pelo módulo RSA usando o algoritmo de Euclides [CHIARAMONTE 2003], esse algoritmo gera chaves de criptografia padrão RSA. Assim essas chaves são geradas usando números primos.

Pode-se verificar no gráfico da figura que o tempo é bem maior no código com 56 bits que o de 24 bits. Enquando o de 24 bits leva 1 segundo para gerar a chave, o de 56 bits leva 180 segundos, isso ocorre porque o algoritmo de Euclides usa probabilidade para achar os números primos, o que sugere que um outro algoritmo, mais especializado, seja usado para gerar chaves, em especial, quando elas possuem um tamanho maior acima de 24 bits.

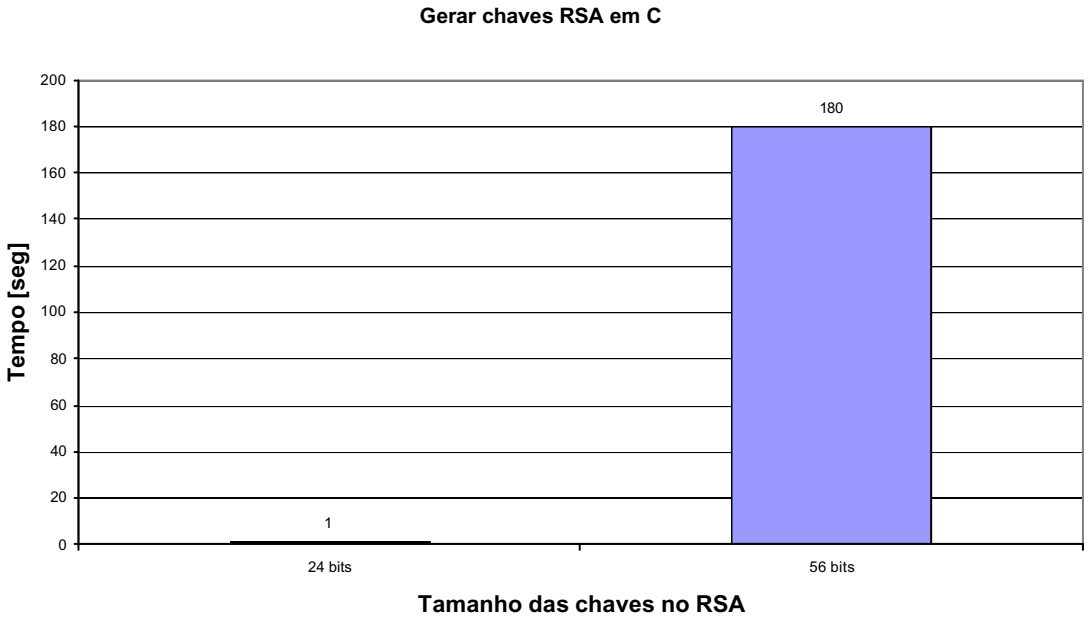

Figura 4.9 – Tempo para geração de chaves no RSA em linguagem C

A Figura 4.10 mostra o tempo em segundos para cifrar arquivo de 1,0 MB e 1,5 MB usando o algoritmo RSA implementado em linguagem C. Conforme pode-se observar no gráfico, para criptografar o arquivo de 1,0 MB levou 16 segundos utilizado o módulo RSA de 24 bits, já utilizando o módulo RSA de 56 bits levou 18 segundos.

Já o arquivo de 1,5 MB usando o módulo de 24 bits levou 24 segundos e usando o módulo de 56 bits levou 35 segundos. Isso implica em um aumento de 45,8 % quando se aumenta o tamanho da chave do RSA de 24 para 56 bits.

A figura 4.10 mostra o tempo em segundos para decifrar o mesmo arquivo de 1,0 MB e 1,5 MB usando RSA. Como esperado, se observa que aumentando o tamanho da mensagem a ser cifrada aumenta-se o tempo necessário para cifrá-la. Similar acontece com o aumento do tamanho da chave, em especial, quando se trabalha com mensagem grandes (acima de 1.5 Mbytes). Igual acontece com o processo de decifrar, como se observa na Figura 4.10.

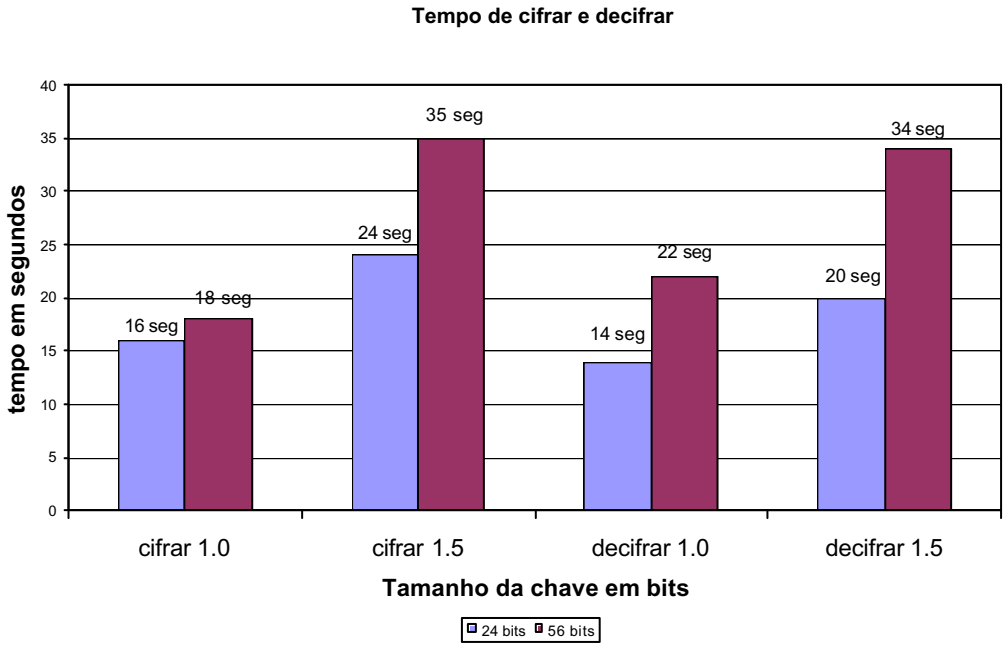

Figura 4.10 – Tempo para cifrar e decifrar arquivos usando RSA em C

Ver detalhes da implementação em C no apêndice 1 que mostra os códigos para o algoritmo RSA-56. A outra implementação (RSA-24) aparecem no CD que acompanha a dissertação.

4.6.2 Implementação do RSA em hardware (FPGAs)

Esta seção apresenta os resultados obtidos usando a criptografia RSA em hardware. Foi utilizado para análise dos resultados um computador Pentium IV 1.6 GHZ com 128 de memória RAM e sistema operacional Windows 2000 e o Xilinx 6.2i.

Os gráficos a seguir mostram informações quanto aos módulos de criptografia baseado no RSA, com diferentes quantidades de bits para a realização do processo de criptografia.

O módulo de sintâxe retorna dados que são muito importantes para o levantamento de resultados, desde a ocupação do código RSA no FPGA até o número de IOBs, Flip Flop, LUTs e o máximo de freqüência em Megahertz no FPGA.

A Figura 4.11 apresenta o máximo de freqüência obtido no módulos do RSA com quantidade de bits diferentes usando diferentes FPGAs. Foram utilizados três FPGAs para a análise, a saber : Virtex II pro, Virtex E e Spartan 3.

Observando a Figura 4.11 é possível perceber que usando diferentes FPGAs o resultado obtido com a velocidade do algoritmo RSA em hardware é diferente usando um FPGA Virtex II Pro é possível obter uma velocidade maior se comparado com o FPGA Virtex E e Spartan 3.

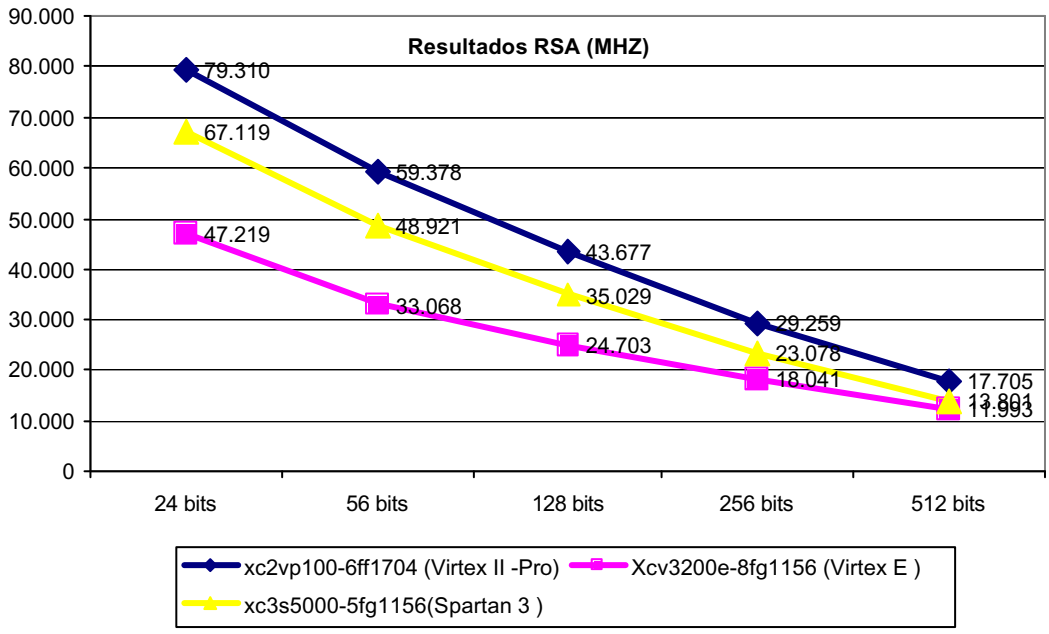

Figura 4.11 – Velocidade do algoritmo RSA em hardware

A Figura 4.12 apresenta o impacto que o tamanho em bits das chaves do algoritmo RSA tem influência na velocidade. Como se esperava aumentando o tamanho (em bits) do algoritmo RSA, a velocidade de cifrar e decifrar diminui.

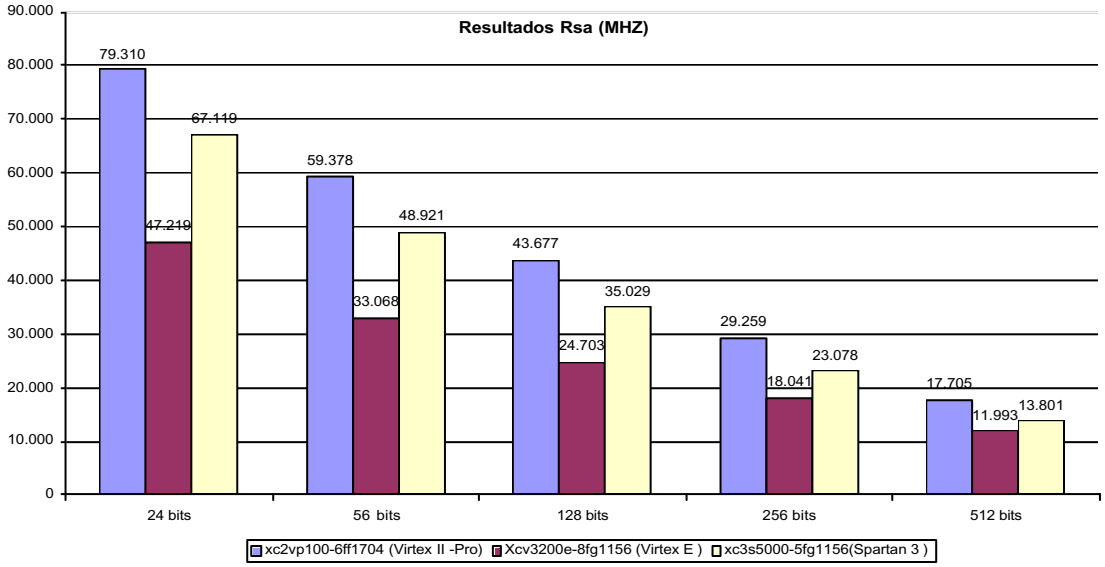

Figura 4.12 – Impacto do tamanho das chaves em (Bits) na velocidade do algoritmo RSA

As Figuras 4.13 até 4.18 apresentam o número de Flip Flop, IOBs, Slices e LUTs utilizados no FPGA.

Nessa Figura é possível observar que a quantidade de Flip Flop aumenta conforme a quantidade de bits usados para criptografar e decriptografar pelo algoritmo RSA, que está em função do tamanho dos bits das chaves.

A Figura 4.16 mostra o número de IOBs em diferentes FPGAs e diferentes números de bits. Podemos verificar que com o aumento do número de bits a ocupação no FPGA é maior ou seja utiliza mais espaço no FPGA. LUTs (*lookup table*) são tabelas utilizadas no FPGA, novamente pode-se verificar que com o aumento do número de bits o número de LUTs utilizado pelo módulo RSA é maior. Similarmente acontece com a porcentagem de ocupação (slices) que aparece nas figuras.

Conforme se esperava, pode-se verificar que com o aumento do número de bits no tamanho da chave do algoritmo RSA a ocupação no FPGA é maior ou seja utiliza mais espaço no FPGA.

A Figura 4.18 mostra o tempo do RSA em nanosegundos, esse tempo é dado pela ferramenta utilizada e representa o tempo utilizado pelo módulo RSA, podemos verificar então que com o aumento do número de bits, há um aumento de período

máximo, diminuindo assim a frequência de operação da implementação em hardware (em FPGAs).

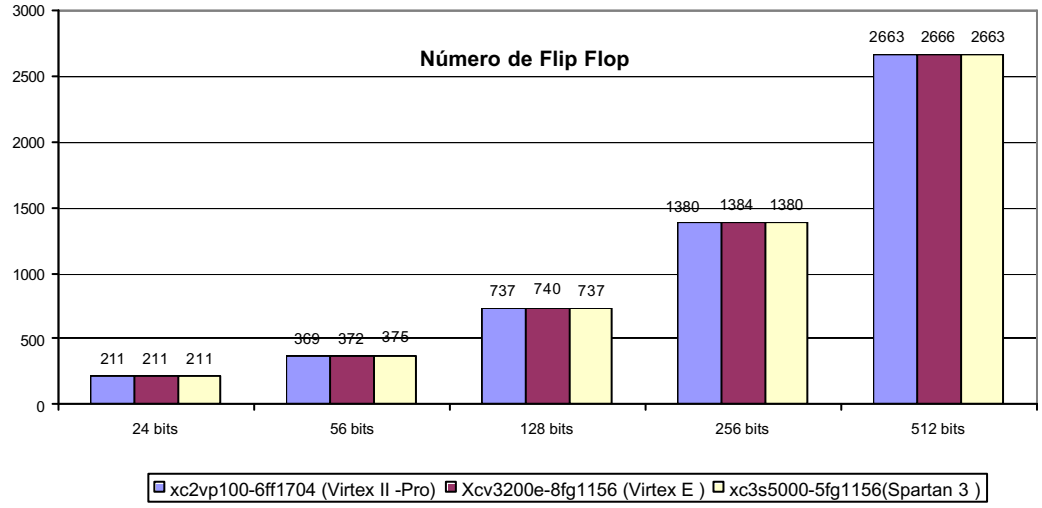

Figura 4.13 – Números de Flip Flop utilizados pelo RSA

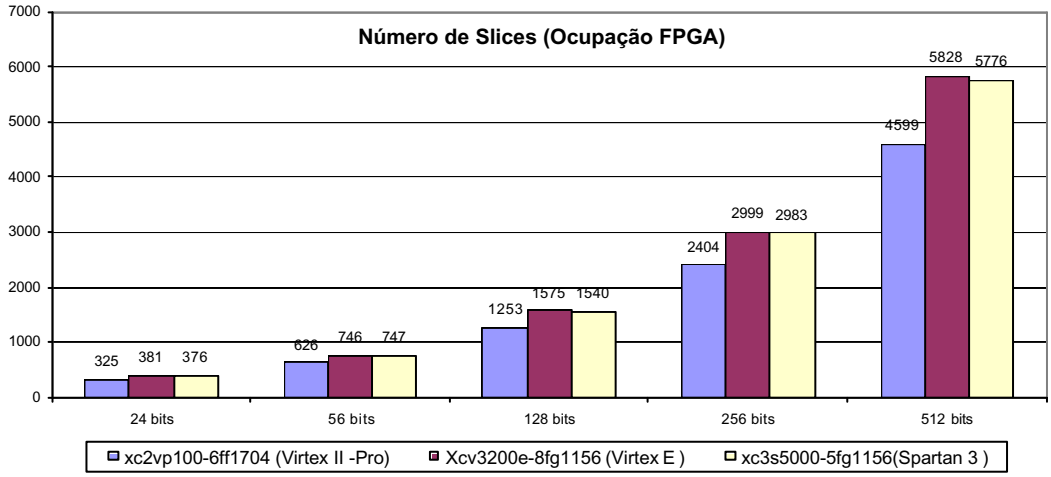

Figura 4.14 – Números de Slices utilizados pelo RSA

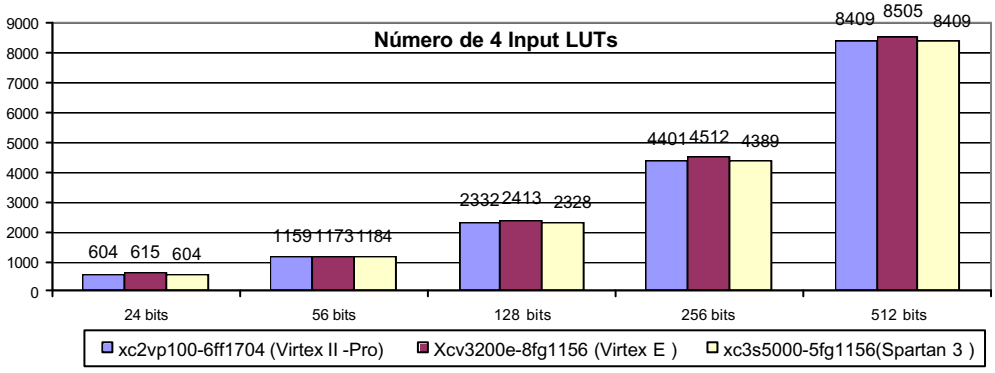

Figura 4.15 – Número de 4 Luts – 4 entradas utilizados pelo RSA

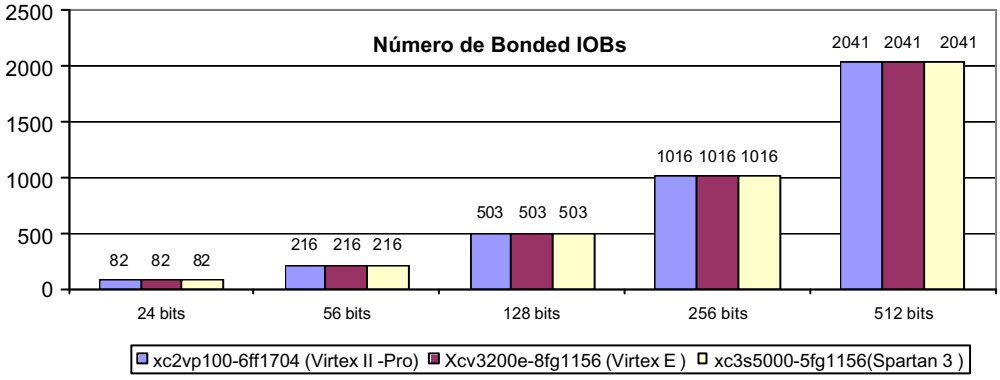

Figura 4.16 - Números de IOBs utilizados pelo RSA

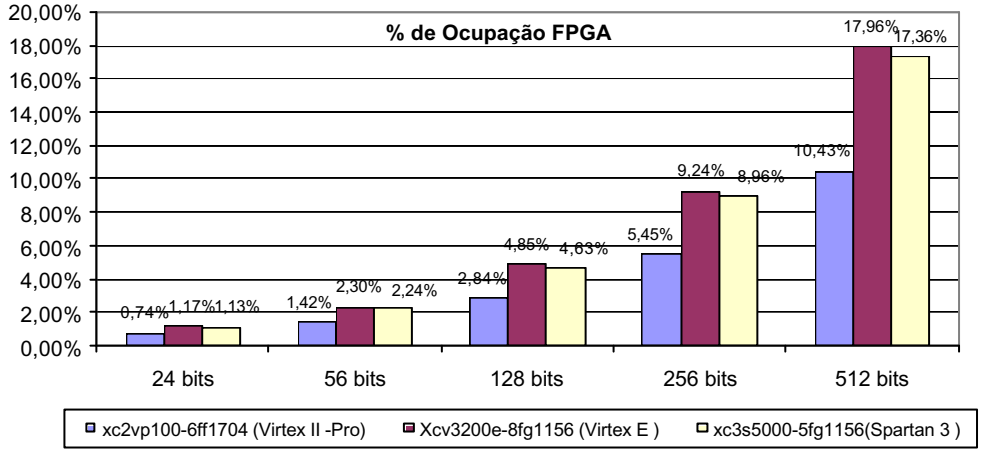

Figura 4.17 – Impacto do tamanho (em bits) do RSA em FPGAs

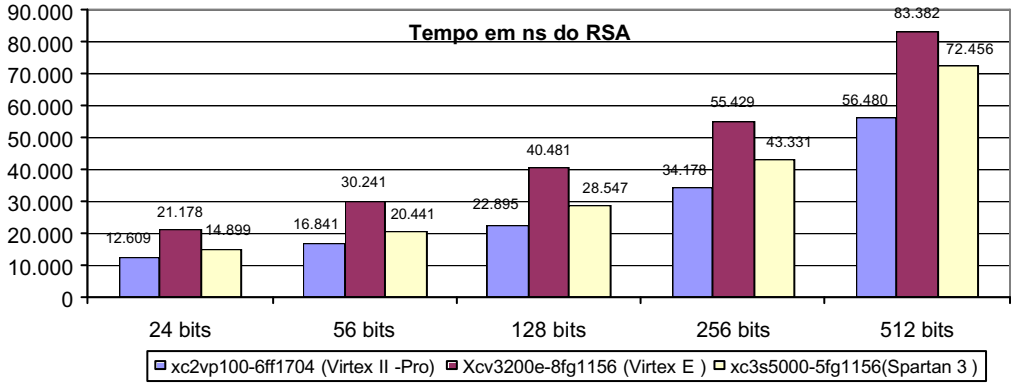

Figura 4.18 - Tempo em nanosegundos do processo de cifrar no RSA

A Tabela 4.10 mostra o percentual de ocupação (SLICE) de cada um dos FPGAs utilizado. Como se esperava, aumentando o tamanho (em bits) da chave no do algoritmo RSA, o percentual de ocupação aumenta. Interessante perceber que a ocupação é pequena máximo de (18%), o que permite pensar que é possível colocar mais funcionalidades no FPGA, executando o algoritmo RSA, por exemplo o padrão PKCS#11.

Tabela 4.10 – Tamanho total permitido de ocupação no FPGA

FPGA	24 bits	56 bits	128 bits	256 bits	512 bits	Total Ocupação
xc2vp100-6ff1704 (Virtex II -Pro)	0,74%	1,42%	2,84%	5,45%	10,43%	44096
Xcv3200e-8fg1156 (Virtex E)	1,17%	2,30%	4,85%	9,24%	17,96%	32448
xc3s5000-5fg1156(Spartan 3)	1,13%	2,24%	4,63%	8,96%	17,36%	33280

A Tabela 4.11 mostra o total máximo permitido de 4 input LUTs e o total de ocupação e o total de IOBs nos FPGAs. Esse são os totais máximos que se pode utilizar desses FPGAs. A quantidade total máxima permitida de entradas, total de ocupação e total de IOBs.

Tabela 4.11 – Número Total de LUTs – 4 Input, Ocupação e IOBs

FPGA	LUTs	Total Ocupação	Total IOBs
Xc2vp100-6ff1704 (Virtex II-Pro)	88192	44096	1040
Xcv3200e-8fg1156 (Virtex E)	64896	32448	808
Xc3s5000-5fg1156(Spartan 3)	66560	33280	784

Ver detalhes da implementação em VHDL no apêndice 2 que mostra os códigos para o algoritmo RSA-512. As outras implementações (RSA-24, RSA-56, RSA-128 e RSA-256) aparecem no CD que acompanha a dissertação.

4.7 Considerações finais do capítulo

No projeto da máquina de estado finito RSA no padrão PKCS#11 versão 6.2, foi utilizado o FPGA Spartan2E e a ferramenta de síntese XST do Xilinx 6.2, foi utilizado um computador Pentium IV, com 256 de memória RAM no qual recebe o dado original e cifrado e decifrado usando o módulo de criptografia RSA, a máquina de estado esta no padrão PKCS#11.

5. OTIMIZAÇÕES DA IMPLEMENTAÇÃO EM HARDWARE DO PKCS#11

5.1 Otimizações

5.1.1 Considerações sobre cada código

O código da máquina de estados versão 3.1 foi implementado com 7 estados usando o Xilinx 3.1 e simulado usando a ferramenta de síntese chamada FPGA Express.

A versão 6.1 corresponde ao código semelhante ao código denominado versão 6.2 com a diferença de algumas adequações que foram realizadas devido às diferenças entre ferramentas de síntese. O mesmo código implementado usando o Xilinx 3.1 e simulado, poderia ser usado no Xilinx 6.2, mas devido às diferença entre o FPGA Express e o XST, o mesmo código precisa ter algumas mudanças.

Existem 3 versões do código, o primeiro implementado e simulado usando o Xilinx 3.1 e a ferramenta de síntese FPGA Express, o segundo denominado versão 6.1, código implementado no Xilinx 3.1 usando o FPGA Express e o terceiro código denominado versão 6.2 implementado em hardware usando a ferramenta de síntese XST e usando o FPGA Spartan2E.

A nossa versão 6.2 no qual denominamos, tem 10 estados e foi implementada usando o Xilinx 6.2 e a ferramenta de síntese XST. É importante observar que existem diferenças nas ferramentas de síntese. Entre essas diferença podemos citar a atribuição de valores na criação de vetores, na versão 3.1 não é necessário atribuir valor na criação do vetor, já na versão 6.2 é necessário atribuir um valor na criação do vetor, mesmo que seja zero.

A versão 6.2 da FSM tem 10 estados e a primeira versão denominada versão 3.1 com somente 7 estados, essa diferença entre versões é devido à nossa versão 6.2 ter sido implementada em hardware, incluindo estados para montar o byte a ser transmitido pela serial, sendo necessário colocar o um bit antes do byte a ser enviando denominado start bit, no início antes dos 8 bits e o stop bit no final do byte a ser enviado, no dado criptografado e decriptografado e o dado original para enviar pela serial. Na versão 6.2 foi implementado também o cleartosend para envio do dado pela serial, para limpar os registradores e também a transmissão serial foi implementada para enviar o dado em

uma freqüência de 50 MHZ, quando se trabalha com uma freqüencia é necessário sincronizar o envio do byte com a freqüencia do FPGA.

5.1.2 Máquina de estados Finito – versão 6.2

A Figura 4.19 mostra os resultados obtidos da máquina de estados que implementa todas as operações indicadas no padrão PKCS#11 para realizar uma conexão segura. Essa versão foi implementada para funcionar no Xilinx 6.2, foi implementada com 10 estados, sendo o inicial 0 e o último estado 9.

Pode-se verificar na Figura 4.19 os estados da FSM e a saída de s_out pela porta serial, S_D_CRIP corresponde ao dado cifrado e S_D_DECRIP corresponde ao dado decifrado e clk corresponde ao clock.

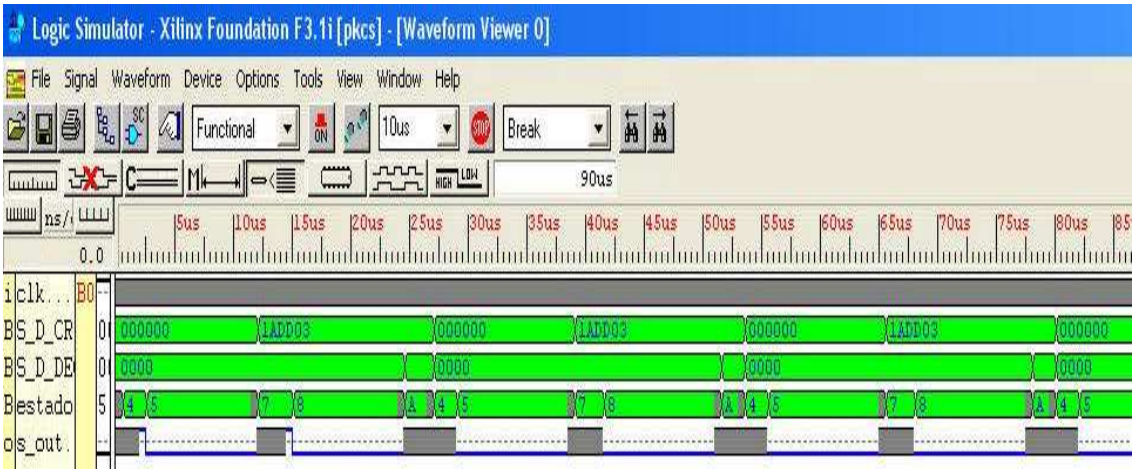

Figura 5.1 - Simulação da máquina de estados Finito – versão 6.2

A Tabela 4.12 mostra a descrição dos estados na máquina de estados finito que faz a criptografia RSA usando o padrão PKCS#11. A FSM tem vários estados necessários para o funcionamento da máquina.

Cada estado faz uma parte, ou seja cada um deles é responsável pela execução de uma parte da máquina de estados. A principal vantagem da FSM RSA versão 6.2 no padrão PKCS#11 é que todos os processos são executados partindo de um dado original, que é criptografado e enviado por um slot, que neste projeto é a porta serial do computador.

Tabela 5.1 – Estados da Máquina de estados Finito RSA padrão PKCS#11 versão 6.2

Nº do estado	Descrição do estado
0	Inicializa Máquina
1	Atríbue valores ao token
2	Monta informações do TOKEN
3	Envia informações do TOKEN serialmente
4	Libera a Criptografia do dado
5	Monta o dado criptografado
6	Transmite o dado criptografado
7	Libera a decriptografia
8	Monta o dado decriptografado
9	Transmite o dado decriptografado

5.1.3 Descrição da Máquina de estados Finito padrão PKCS#11 versão 6.2

A Figura 4.20 mostra os estados da máquina de estado finito versão 6.2 que implementa em hardware o padrão PKCS#11, bem com a transição de um estado para outro. Os estados variam do estado 0 até o estado 9 ou seja a máquina de estado finito versão 6.2 padrão PKCS#11 é constituída de 10 estados que estão relacionados entre si. Em cada momento, dependendo de um evento seja baseado no clock ou no estado da máquina, um estado estará em funcionamento e passará para o estado seguinte assim que o estado anterior for completado e a respectiva condição for satisfeita. Em nosso projeto, a máquina de estado finito padrão PKCS#11 tem inicio no estado 0 (zero). Quando a máquina está em estado 0 (zero) as variáveis “B1”, “B2”, “envch1”, “envch2” , “envenc1”, “envenc2” , “envenc3” , “envdec1”, “envdec2” recebem o valor 0 (zero) e a variável “microcont” recebe o valor 1 que será somado. Assim, no estado 0 a máquina é inicializada.

No estado 1 são atribuídos os valores dos tokens conforme escolha do usuário, ou seja a variáveis B1 e B2 irão receber o valor do token escolhido, e a variável ESTADO recebe DOIS, valor no qual o estado da máquina passará para o próximo estado que corresponde a 2.

No estado 2 é montada informação para o Token, a variável envch1 recebe o valor “01” que representa o start bit e o valor de B1 que corresponde ao token escolhido representando o primeiro byte e também o “10” que representa o stop bit. Sendo

representado da seguinte forma envch1 <= "01" & not(B1) & "10" , da mesma forma que envch2 recebe o start bit e o B2 e stop bit representando da seguinte forma envch2 <= "01" & not(B2) & "10" logo após a variável VAR receber o valor de B1 e B2 sendo que as duas variáveis são concatenadas para VAR receber o valor. A variável ESTADO recebe TRES, valor no qual o estado da máquina passará para o próximo estado que corresponde a 3.

No estado 3 são enviadas as informações do token, se a variável “libtres” for igual a 1 (um), indica que passou pelo estado 3. A variável estado recebe o valor 4 indicando para passar para o estado seguinte.

No estado 4, é liberada a criptografia do dado, se a variável S_P_CRIP for igual ao valor 1 (um) indica que acabou a criptografia, e a variável estado recebe o valor 5 indicando para passar para o estado seguinte.

No estado 5, é montado o dado para criptografia, a variável envenc1 recebe o valor “01” que representa o start bit e o valor de S_D_CRIP(7 downto 0) que corresponte ao dado criptografado escolhido representando o primeiro byte e também o “10” que representa o stop bit. Sendo representado da seguinte forma envenc1 <= "01" & not(S_D_CRIP(7 downto 0)) & "10", da mesma forma que envec2 recebe o start bit e o S_D_CRIP(15 downto 8) e stop bit representando da seguinte forma envenc2 <= "01" & not(S_D_CRIP(15 downto 8)) & "10"; e envenc3 <= "01" & not(S_D_CRIP(23 downto 16)) & "10". A variável ESTADO recebe SEIS, valor no qual o estado da máquina passará para o próximo estado que corresponde a 6. O estado 6 transmite o dado criptografado, se o valor da variável LIBSEIS for igual a “1”. A variável estado recebe o valor SETE indicando para passar para o estado seguinte.

O estado 7 libera a decriptografia, se o valor da variável S_P_DECRIP for igual a “1”. A variável estado recebe o valor OITO indicando para passar para o estado seguinte.

O estado 8 monta o dado decriptografado, a variável envdec1 recebe "01" & not(S_D_DECRIP(15 downto 8)) & "10" e envdec2 recebe "01" & not(S_D_DECRIP(7 downto 0)) & "10". A variável estado recebe o valor NOVE indicando para passar para o estado seguinte. No estado 9 a variável estado recebe o valor ZERO retornando ao estado 0 da FSM.

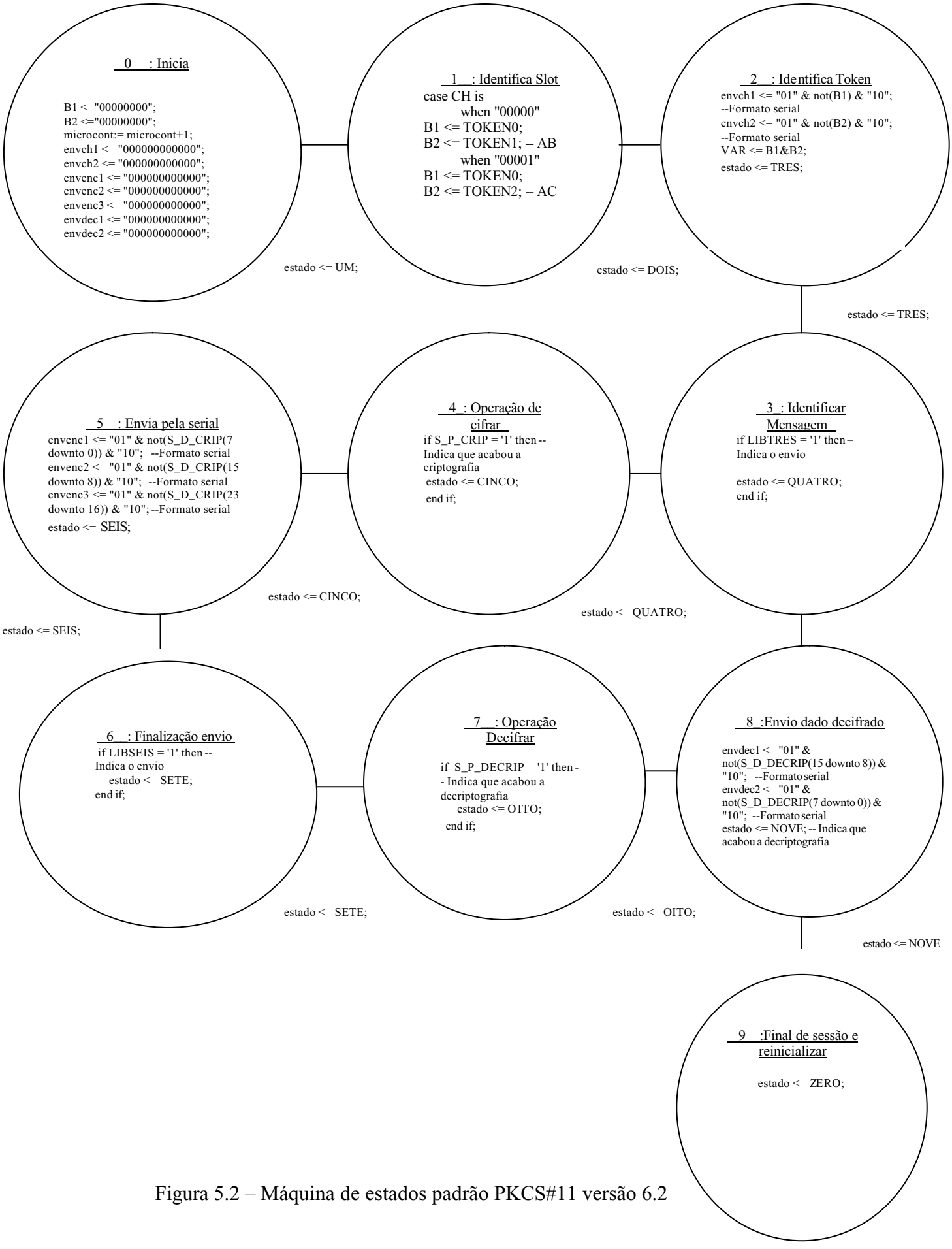

Figura 5.2 – Máquina de estados padrão PKCS#11 versão 6.2

Na Figura 4.21 pode-se verificar através de um osciloscópio o envio dos bits pela porta serial, a linha superior corresponde aos bits enviados para outro computador, que são os dados originais e o respectivo dado cifrado e decifrado. Na linha inferior pode-se verificar o sinal do “cleartosend”, que é responsável por zerar os registrados para o envio do próximo quadro de bits.

Figura 5.3 - Sinal digital da transmissão serial da FSM versão 6.2

Na Figura 4.22 pode-se vizualizar o osciloscópio e o FPGA transmitindo dados para o outro computador através da porta serial. O FPGA usado no projeto foi uma Spartan2E.

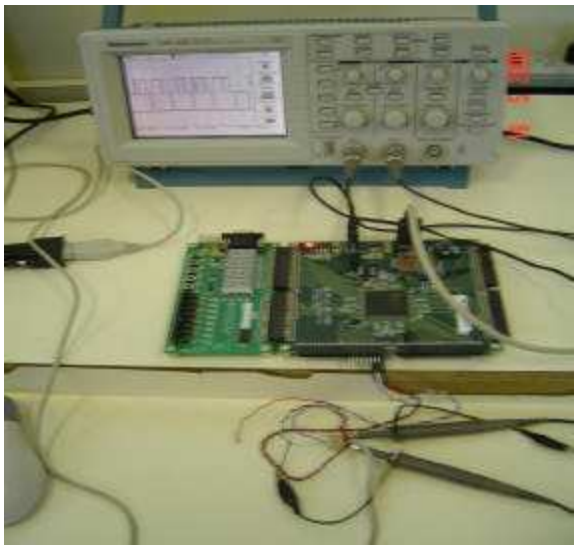

Figura 5.4 - Osciloscópio e FPGA usados no projeto da FSM versão 6.2

Na Figura 4.23 pode-se vizualizar o computador utilizado no projeto que recebe o dado do FPGA pela porta serial do computador. Primeiro é enviado o dado original escolhido, por exemplo dois bytes que corresponde às letras " DE" logo após é enviado o dado criptografado e depois o dado decriptografado.

Figura 5.5 - Computador, Osciloscópio e FPGA usados no projeto da FSM versão 6.2

Na Figura 4.24 pode-se vizualizar o dado enviado pelo FPGA para a porta serial deste computador, no qual corresponde aos bytes " DE", sendo o dado original escolhido pelo usuário e logo após o dado cifrado.

Após enviar o dado cifrado é enviado o dado decifrado que corresponde ao mesmo dado original ou seja "DE" .

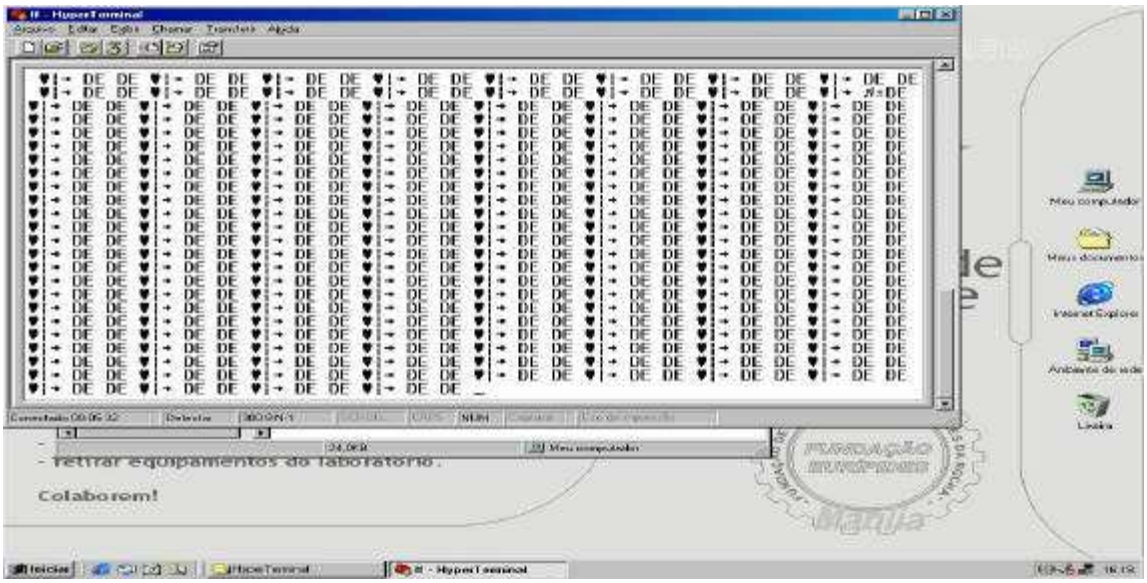

Figura 5.6 - Tela do computador que recebe o dado pela porta serial.

Na Figura 4.25 pode-se vizualizar a ocupação do FSM versão 6.2 no FPGA usando o Xilinx 6.2.

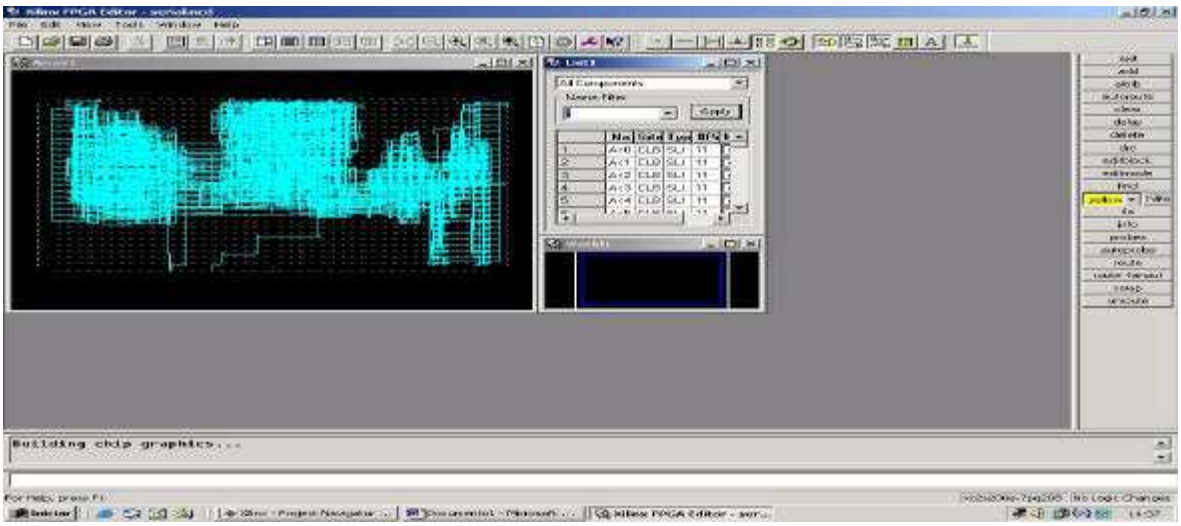

Figura 5.7 - Ocupação da FSM versão 6.2 no FPGA

A Tabela 4.13 mostra a porcentagem total de ocupação da máquina de estados finito RSA padrão PKCS#11 versão 6.2 no FPGA, corresponde a 44%.

Pode-se verificar o total de Flip Flop e a porcentagem de Flip Flop, a Spartan2E tem um percentual maior, 14% de Flip Flop devido ao FPGA ser de tamanho menor. O mesmo ocorre com o percentual de LUTs que chega a 41% na Spartan2E.

Pode-se verificar que a Virtex II Pro é a FPGA que tem a maior frequência de operação (82.642 MHz), em segundo lugar aparece a Spartam 3 com 71.831 Mhz, em terceiro a Virtex E com 47.492 Mhz e por último a Spartam 2E com 41.714 Mhz.

Pode-se verificar o tempo da máquina de estados em nanosegundos. O maior tempo utlizado é a Spartam 2E com 23.973 ns, em segundo lugar a VirtexE com 21.056 ns e em terceiro lugar a Spartam 3 com 13.922 ns e por último a Virtex II PRO com 12.100 ns.

A baixa porcentagem de Slices da Virtex II Pro representa que o FPGA é de tamanho maior sendo que o percentual utilizado no FPGA usando a FSM é de apenas 2%. Usando a Spartan2E no mesmo projeto, representa 44% da ocupação da FSM no FPGA.

Tabela 5.2 – Resultados da implementação da máquina de estados finito padrão PKCS#11 versão 6.2

FPGAs	Slices	Flip Flop	LUTS	IOBs	Máximo Período NS	Frequência MHZ
xc2x200e - (Spartan2E)	1054 (44%)	662 (14%)	1932 (41%)	3 (2%)	23.973	41.714
xc3s5000-5fg1156(Spartan 3)	1040 (3%)	660 (0%)	1902 (2%)	3 (0%)	13.922	71.831
xcv3200e-8fg1156 (Virtex E)	1052 (3%)	662 (1%)	1929 (2%)	3 (0%)	21.056	47.492
xc2vp100-6ff1704 (Virtex II-Pro)	1030 (2%)	660 (0%)	1915 (2%)	3 (0%)	12.100	82.642

Ver detalhes da implementação em VHDL no apêndice 3 que mostra os códigos para o algoritmo FSM 6.2. A outra implementação (FSM 3.1, FSM 6.1) aparecem no CD que acompanha a dissertação.

5.2 Comunicação entre Dois FPGAs com o Padrão PKCS#11

Na Figura 4.26 pode-se vizualizar a comunicação entre dois FPGAs cada um com a FSM versão 6.2. O transmissor A envia o dado cifrado para o receptor B que irá realizar a decifragem. O mesmo ocorre no inverso, o transmissor B enviar o dado cifrado para o receptor A que irá decifrar o dado.

5.8 – Comunicação entre FPGAs das FSM versão 6.2 padrão PKCS#11

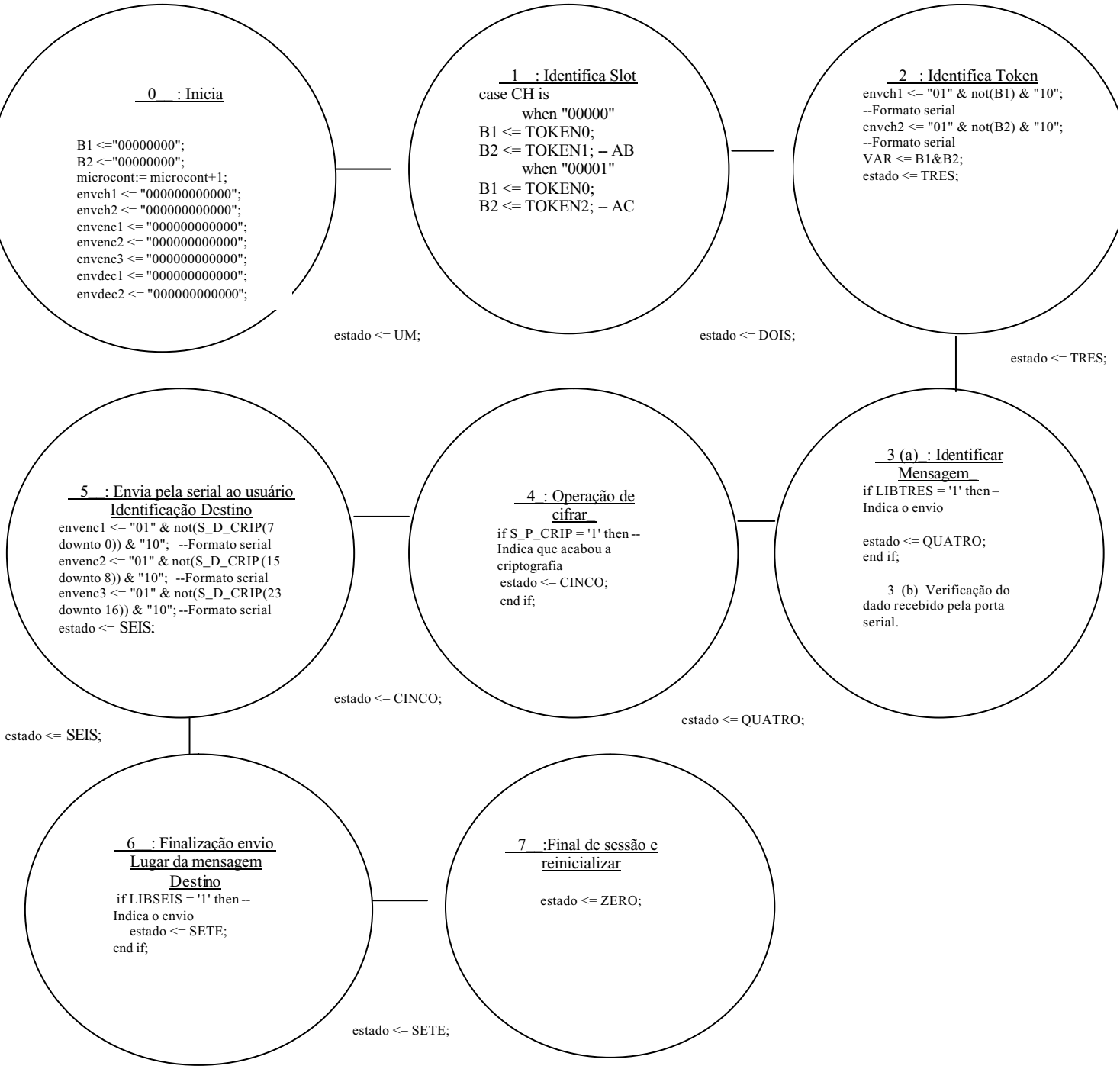

Figura 5.9 – Máquina de estados padrão PKCS#11 versão 6.2 para comunicação com outro FPGA enviando um dado cifrado.

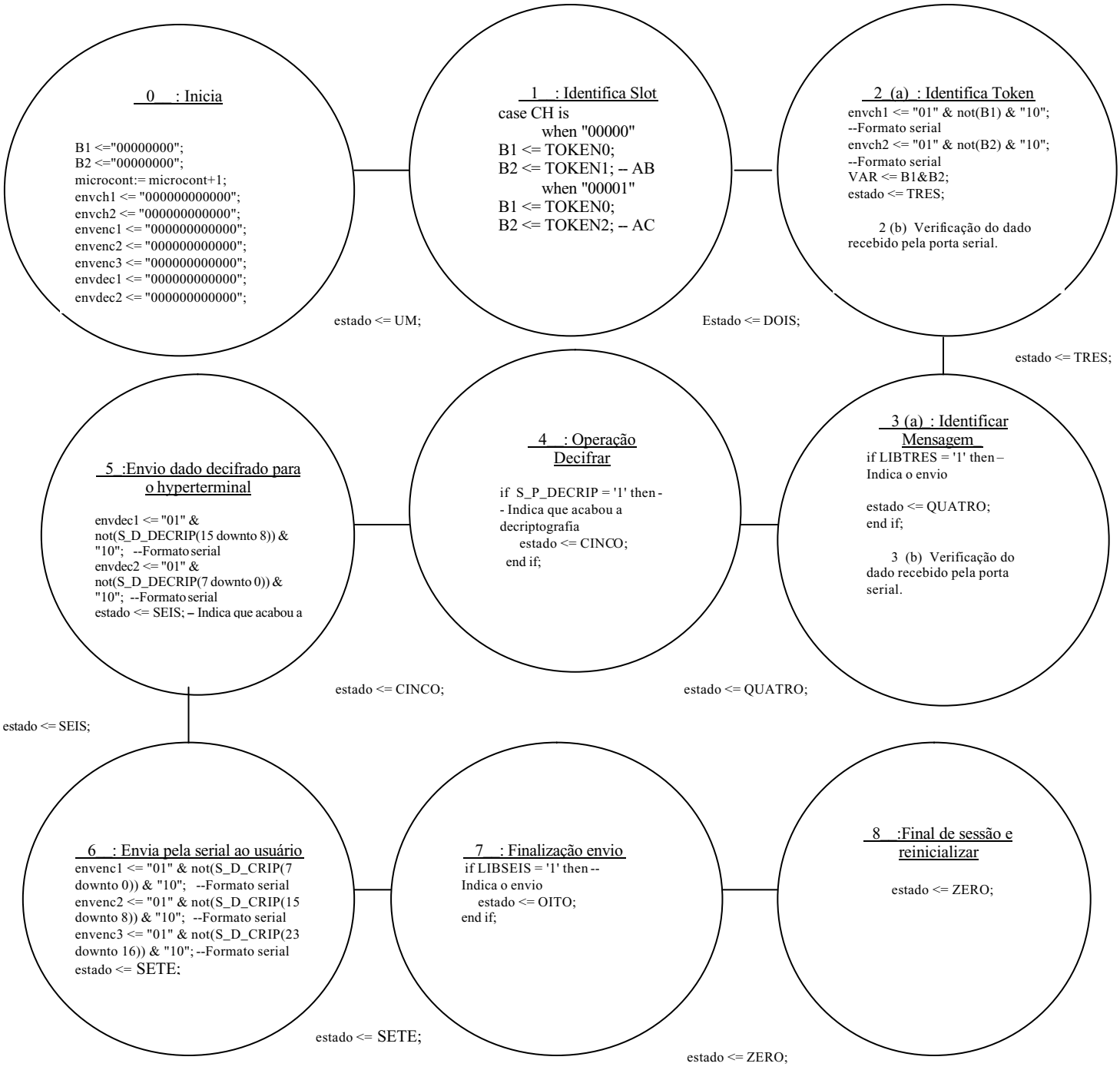

Figura 5.10 – Máquina de estados padrão PKCS#11 versão 6.2 para comunicação com outro FPGA recebendo um dado cifrado e decifrando-o.

A Figura 4.27 mostra os estados da máquina de estado finito que irá cifrar o dado para enviar para outro FPGA, a FSM poderá ser transmissor e receptor, no momento que for transmistir um dado cifrado será transmissor e quando for receber o

dado cifrado de outro FPGA será receptor. A Figura 4.28 mostra os estados da FSM e irá realizar a decifragem no momento que receber um dado cifrado da outro máquina através do FPGA, que será enviando pela serial para um computador que receberá o dado decifrado e mostrará no Hyperterminal.

5.3 Considerações finais do capítulo

No projeto da máquina de estado finito RSA no padrão PKCS#11 versão 6.2, foi utilizado o FPGA Spartan2E e a ferramenta de síntese XST do Xilinx 6.2, foi utilizado um computador Pentium IV, com 256 de memória RAM no qual recebe o dado original e cifrado e decifrado usando o módulo de criptografia RSA, a máquina de estado esta no padrão PKCS#11. A implementação deste projeto contribui para a transmissão de dados com segurança, usando o módulo RSA e o padrão de criptografia de chave pública PKCS#11 é possível usar um outro módulo de criptografia não precisa ser somente o RSA, o importante é que a máquina de estados finito está no padrão PKCS#11, que é o padrão para hardware. Um FPGA que transmite um dado para outro computador e efetua criptografia e decriptografia é importante para a comunicação entre computadores.

6. CONCLUSÕES

Existem diversos processadores de rede para uso comercial, existem vários fabricantes e modelos diferentes que são específicos para trabalho em rede, a fim de melhorar o QoS (*Qualidade de serviços*), para melhorar o tráfego e a transição de dados evitando assim gargalos, principalmente em roteadores e switches que acabam sendo os principais componentes em uma rede. Os processadores de rede podem ser um SOC (*System on Chip*) sendo possível ser implementado em um chip, bem como implementado em um roteador ou switch.

Os processadores de rede foram criados para justamente diminuir o gargalo do tráfego da rede e tratar os pacotes, os roteadores e switch ativos por exemplo, são aqueles que verificam o cabeçalho do pacote e processam o mesmo analisando esse cabeçalho. Com o aumento do tráfego e a Internet cada vez mais presente é necessário a criação e utilização de processadores de rede.

Existem diversos processadores de rede comerciais, mas poucos com implementação de segurança usando criptografia no processamento de pacotes. Por isso a necessidade de criação de um protótipo com módulo de segurança para processadores de redes.

Não existem processadores de rede no Brasil implementados em VHDL e FPGA.

A criação de um protótipo com módulo PKCS#11 para segurança em processadores de rede é importante, já que o PKCS#11 é um padrão de segurança criado para criptografia baseada em token, podendo ser usado em software como em hardware.

Este trabalho teve como objetivo pesquisar as tecnologias e tipos de processadores de rede, tipos de segurança e criptografia baseados no PKCS#11, bem como a implementação de segurança para processadores de rede com a criação de um módulo de criptografia do padrão PKCS#11, implementado em FPGAs.

A implementação do protótipo com o padrão PKCS#11 foi feita em VHDL e prototipado em hardware usando FPGA, seguindo uma implementação baseada em máquinas de estado finito. Além da análise de resultados obtidos usando criptografia em

hardware, foi possível o envio de um dado criptografado usando o padrão PKCS#11 para outro computador usando uma comunicação serial.

Foram criadas três versões da máquina de estados finito - FSM, a versão 3.1, 6.1 e 6.2. A versão 3.1 foi implementada na ferramenta Xilinx versão 3.1 e simulada usando a ferramenta de síntese FPGA Express. A versão 6.1 foi implementada no Xilinx 6.2 e finalmente a versão 6.2 foi também implementada no Xilinx 6.2.

A máquina de estado implementada em hardware, chamada de FSM 6.2 usa o algoritmo de criptografia RSA, que também foi implementado em FPGAs. É bom lembrar que pode ser usado qualquer outro algoritmo de criptografia, não precisa ser específico o RSA. O importante neste projeto foi mostrar que o padrão de criptografia PKCS#11 pode ser implementado em hardware e não somente em software, enfatizando nos recursos utilizados em um determinado FPGA.

Está implícito no padrão uma série de especificações para estabelecer uma comunicação segura entre slots, comunicações baseadas em tokens, como por exemplo o login de um usuário no token e a abertura de uma sessão, enquanto a sessão estiver aberta a comunicação entre dois slots está se estabelecendo, quando termina a sessão a mesma é fechada.

Obteve-se resultados significativos para demonstrar o padrão PKCS#11 usando criptografia RSA se comunicando com o slot e com outro computador. Usando hardware (FPGAs) foi possível enviar um dado original, um cifrado e um decifrado; o que torna possível maior segurança para a comunicação entre computadores e dispositivos.

Foi realizada a implementação em linguagem C e VHDL do algoritmo RSA e foi possível verificar a diferença entre a implementação em software usando linguagem C e em hardware (VHDL e FPGAs), e como esperado, em hardware houve um desempenho superior.

O algoritmo RSA em C foi implementado usando 24 bits e 56 bits, não foi possível aumentar o número de bits devido às variáveis em C não aceitarem tamanho maior que 32 bits, dessa forma não foi possível implementar o RSA em C com 128 bits, 256 bits e 512 bits e 1024 bits.

Na implementação usando VHDL foi possível implementar o RSA com 56 bits, 128 bits, 256 bits e 512 bits. Não foi possível implementar 1024 bits devido a que em nossa implementação usou-se um vetor de 1024 bits, e a ferramenta Xilinx 6.2 tem restrições para vetores deste tamanho.

Para implementação em software usando o padrão PKCS#11 é possível usar uma biblioteca em C chamada cryptoki, já em hardware não foi possível usar essa biblioteca sendo necessário abstrair o padrão PKCS#11 para a realização da implementação em hardware usando FPGAs.

Foi usado o FPGA Spartan2E, no qual foi possível a realização de teste enviando um dado criptografado para outro computador usando uma comunicação serial. Nós usamos o software chamado hyperterminal do windows para verificar o dado que computador receptor recebeu. É possível usar um código maior já que a ocupação do FPGA não foi total, sendo possível implementações do RSA usando chaves maiores neste FPGA.

A segurança usando criptografia aumenta a segurança e um módulo que cifra e decifra usando um padrão poderá ser implementado em um processador de rede.

Trabalhos Futuros

É possível desenvolver trabalhos futuros como por exemplo adequar a implementação do padrão PKCS#11 a uma comunicação usando o slot USB com um FPGA, toda comunicação baseada em token é possível implementar um algoritmo em hardware para comunicação usando criptografia. Outra possibilidade de trabalho futuro é um servidor responsável pela criptografia de tudo que trafega em uma LAN, por exemplo usando um FPGA no servidor e o padrão PKCS#11, outra possibilidade é o uso de Smart Card para enviar e receber dados de forma segura usando o padrão. É possível implementar e inserir no padrão PKCS#11 outros algoritmos de criptografia como por exemplo o DES, AES e o RC6.

7. BIBLIOGRAFIA

[ADAMS 1999] Adams, C. e Lloyd, S. Understanding Public-Key Infrastructure: Concepts, Standards and Deployment Considerations. Macmillan Technical Publishing, 1999.

[ADILETTA 2002] M. Adiletta, et. al, geração seguinte de processadores de rede de Intel IXP *jornal da tecnologia de Intel* , edição 3 do Vol. 6, agosto 2002.

[AGERE 2001] Agere System, Fast Pattern Processor (FPP) Product Brief, April 2001, http://www.agere.com

[BERNERS 1989] *Information Management: A Proposal* by Tim Berners-Lee, CERN, March 1989, May 1990 . Em http://www.w3.org/History/1989/proposal.html em Fev/1998.

[BUYA 1999] Buya, R., High Performance Cluster Computing, Volume 1, Prentice Hall, 1999.

[CHAMELEON 2000] Chameleon Systems, "CS2000 Reconfigurable Communications Processor", Family Product Brief, 2000.

[CHIARAMONTE 2003] CHIARAMONTE, R. B., Implementação e Teste em Hardware e Software de Sistemas Criptográficos. Trabalho de Conclusão de Curso. UNIVEM - Centro Universitário Eurípides de Marília - Marília, 2003.

[CISCO 2001] Cisco Systems White Paper, "The Evolution of high-end Router Architectures-Basic Scalability and Performance Considerations for Evaluating Large-Scale Router Designs", 2001, http://www.cisco.com

[C-PORT 2002] C-Port, C5e Network Processor Product Brief, January 2002, http://www.motorola.com

[CROWLEY 2000] CROWLEY, P., et al, "Characterizing processor architectures for programmable network interfaces", Em Procedings of International Conference on Supercomputing, Santa Fé, 2000.

[COMPTON 2002] Compton, K. e Hauchk, S. "Reconfigurable Computing : A Survey of Systems and Software", ACM Computing Surveys, Vol. 34, Nº.2, June 2002, pp.171-210.

[EZCHIP 2002] EZChip Network Processors, http://www.ezchip.com

[FREITAS 2001] H. C. Freitas, C. A. P. S. Martins, "Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas", II Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2001, Pirenópolis - GO, pp.31-38.

[FREITAS 2000] H. C. Freitas, C. A. P. S. Martins, "Projeto de Processador com Microarquitetura Dedicada para Roteamento em Sistemas de Comunicação de Dados", I Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2000, São Pedro - SP, pp.63, (Iniciação Científica).

[FREITAS 2002] H. C. Freitas, C. A. P. S. Martins, "Simulation Tool of Network Processor for Learning Activities". Frontiers in Education Conference (FIE 2002), Boston, USA, November 2002.

[GLESNER 1998] M. Glesner, A. Kirschbaum, "State-of-the-Art in Rapid Prototyping", XI Brazilian Symposium on Integrated Circuit Design, SBCCI'98, Búzios, Rio de Janeiro, 1998, pp.60-65.

[IBM 2002] PowerNP NP4GS3 Databook, http://www.ibm.com

[IBM 4758] Coprocessor 4758 IBM
http://www-3.ibm.com/security/cryptocards/

[INTEL 2001] Intel WAN/LAN Access Switch Example Design for the Intel IXP 1200 Network Processor, May, 2001, http://www.intel.com

[INTEL 2000] Intel, "IXP 1200 - Network Processor", Datasheet, May 2000, http://www.intel.com

[INTEL 2002] Packet SONET excedente: Conseguindo o pacote de 10 Gigabit/sec que processa com um IXP2800 *jornal da tecnologia de Intel* , edição 3 do Vol. 6, agosto 2002.
http://developer.intel.com/technology/itj/2002/volume06issue03

[KÄSTNER 2003] Kästner, D., "Classification of Microprocessors", Departamento de Ciência da Computação, Universität des Saarlandes, Alemanha, 2003.

[LEXRA] Lexra, NetVortex Network Communications System Multiprocessor NPU, http://www.lexra.com

[LUCENT 1999] Lucent Technologies, Building for Next Generation Network Processors, September 1999.

[LUCENT-2 1999] Lucent Technologies, The Challenge for Next Generation Network Processors, September 10, 1999.

[MARTINS 2001] FREITAS, Henrique C. de; MARTINS, Carlos Augusto P. S., "Processador de Rede com Suporte a Multi-protocolo e Topologias Dinâmicas", II Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2001, Pirenópolis - GO, pp.31-38 *(Artigo Completo)*.

[MARTINS 2000] FREITAS, Henrique C. de; MARTINS, Carlos Augusto P. S., "Projeto de Processador com Microarquitetura Dedicada para Roteamento em Sistemas de Comunicação de Dados", I Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2000, São Pedro - SP, pp.63 *(Iniciação Científica)*.

[MARTINS-2 2000] FREITAS, Henrique C. de; MARTINS, Carlos Augusto P. S., "Processador Dedicado para Roteamento em Sistemas de Comunicação de Dados", I Congresso de Lógica Aplicada à Tecnologia – LAPTEC'2000, São Paulo - SP, pp.717-721 *(Iniciação Científica)*.

[MEDEIROS 2002] T. H. Medeiros, C. A. P. S. Martins, "Reconf_KMT, Uma Ferramenta Reconfigurável para a Simulação de Microprocessadores", artigo

submetido ao Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2002

[MENEZES 1996] A MENEZES, P van Oorschot and S. Vanstone, CRC Press, 1996 - Cap. 1, 3, 8, 9, 11 e 13.

[MMC] MMC Networks, "EPIF-105, EPIF-200, GPIF-207, XPIF-300, Packet Processors", http://www.mmcnet.com

[MORENO 2003] MORENO, E. et al. *Projeto, Desempenho e Aplicações de Sistemas Digitais em Circuitos Programáveis (FPGAs).* Bless, 2003.

[MORENO 2005] MORENO, David Edward., PEREIRA, Fábio Dacêncio., CHIARAMONTE, Rodolfo Barros., Criptografia em Software e Hardware. Novatec, 2005.

[MOTOROLA 1999] Motorola Corporation, MPC860 "Computer Organization and Design: The Hardware/Software Interface", Morgan Kaufmann Publisher.

[MYERS 1999] Myers, M. et al 1999, "X.509 Internet Public Key Infrastructure – Online Certificate Status Protocol - OCSP".

[MYRINET] Myrinet Overview, http://www.myri.com/myrinet/overview/index.html

[NP4GS3] IBM PowerNP NP4GS3 Databook, http://www.ibm.com

[PATTERSON 1997] PATTERSON, D. A., J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface", Morgan Kaufmann Publisher, 1997.

[PRADO 2004] PRADO, Ricardo P., NPSOC – ARQUITETURA E PROTÓTIPO DE UM NOVO PROCESSADOR DE REDE, Dissertação de Mestrado, Ciência da Computação, Univem, Centro Universitário Eurípides Soares da Rocha de Marília, p. 92, 2004.

[PUC] Projeto de Processador de Rede com Suporte a Multi-protocolo e Topologias *Dinâmicas* : http://www.inf.pucminas.br/projetos/pad-r/

[RSA 01] RSA Labs. Public Key Criptography Standards (PKCS). *Version 2.1 - 2002, disponível em http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html.*

[RSA 02] RSA, http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/index.html

[RSA 03] RSA Labs. Public Key Criptography Standards (PKCS). *Version 2.1 - 2002, disponível em http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html.*

[RSA 04] RSA Labs. Factorization of RSA-155. *http://www.rsasecurity.com/rsalabs/challenges/factoring/rsa155.html*, 1999.

[RSA 05] RSA Labs. Has the RSA algorithm been compromised as a result of Bernstein's Paper? *http://www.rsasecurity.com/rsalabs/technotes/bernstein.html*, 2002.

[RSA 06] RSA Labs. Public Key Criptography Standards (PKCS). *Version* 2.20 – *2004, disponível em* ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20fd.doc

[R2NP 2002] H. C. Freitas, C. A. P. S. Martins, "R2NP – Processador de Rede RISC Reconfigurável", artigo submetido ao Workshop de Sistemas Computacionais de Alto Desempenho, WSCAD'2002.

[SHAH 2000] Shah, N. "Understanding Network Processors".

[STALLINGS 1996] STALLINGS, William, Cryptography and Network Security – Principles and Practice – Second Edition.

[SECURITY 2001] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart, Building the IBM 4758 secure Coprocessor, revista Computer, outubro 2001.

[SITERA] Sitera IQ2000, Network Processor Product Brief, http://www.sitera.com

[TANEMBAUM 1999] Tanembaum, A. S. "Redes de Computadores", Editora Campus, 3º Edição, Editora Campus,1999.

[TENNENHOUSE 1997] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, G. J. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, Volume 35, Nº 1, pp.80-86, 1997.

[TERADA 2000] TERADA, R.; Segurança de Dados Criptografia em Redes de Computadores., Ed. Edgard Blücher, 1ª Edição, 2000.

[TRI 01] TRISCEND Corporation, http://www.triscend.com

[WOLF 2000] T. Wolf and J. Turner, "Design Issues for High Performance Active Routers", International Zurich Seminar on Broadband Communications, Zurich, Switzerland, February 2000, pp. 199-205.

[VILLASENOR 1997] VILLASENOR, J., W. H. Mangione, "Configurable Computing", Scientific American, Junho de 1997. Disponível em http://xputers.informatik.unikl.de/reconfigurable_ computing/villasenor/0697villasenor.html

[XILINX 1998] XILINX Development Systems, "Synthesis and Simulation Design Guide – Designing FPGAs with HDL", 1998.

ANEXO 1

C_inicialize **Inicializa a biblioteca Cryptoki**	CK_RV CK_ENTRY C_Initialize(CK_VOID_PTR pReserved); **C_Initialize** initializes the Cryptoki library. **C_Initialize** should be the first Cryptoki call made by an application, except for calls to **C_GetFunctionList**. What this function actually does is implementation-dependent: for example, it may cause Cryptoki to initialize its internal memory buffers, or any other resources it requires; or it may perform no action. The *pReserved* parameter is reserved for future versions; for this version, it should be set to NULL_PTR. If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize** should (eventually) be succeeded by a single call to **C_Finalize**. Return values: none other than the "universal" return values.

C_GetSlotList
 	
Obtêm a lista Informações sobre os Slots
	CK_RV CK_ENTRY C_GetSlotList(
	CK_BBOOL tokenPresent,
	CK_SLOT_ID_PTR pSlotList,
	CK_ULONG_PTR pulCount
	);
	C_GetSlotList is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list obtained includes only those slots with a token present (TRUE), or all slots (FALSE); *pulCount* points to the location that receives the number of slots.
	There are two ways for an application to call **C_GetSlotList**: 1. If *pSlotList* is NULL_PTR, then all that **C_GetSlotList** does is return (in **pulCount*) the number of slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on entry to **C_GetSlotList** has no meaning in this case, and the call returns the value CKR_OK.
	2. If *pSlotList* is not NULL_PTR, then **pulCount* must contain the size (in terms of **CK_SLOT_ID** elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots, then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetSlotList** returns the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number of slots.
	Because **C_GetSlotList** does not allocate any space of its own, an application will often call **C_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots with a token present, then the number of such slots can change between when the application asks for how many such slots there are, and when the application asks for the slots themselves). However, this is by no means required.
	Return values: CKR_BUFFER_TOO_SMALL.
	Example: CK_ULONG ulSlotCount, ulSlotWithTokenCount; CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList; CK_RV rv; /* Get list of all slots */
	rv = C_GetSlotList(FALSE, NULL_PTR, &ulSlotCount);
	if (rv == CKR_OK) {
	pSlotList =
	(CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
	rv = C_GetSlotList(FALSE, pSlotList, &ulSlotCount);
	if (rv == CKR_OK) {
	/* Now use that list of all slots */
	}
	free(pSlotList);
	}
	/* Get list of all slots with a token present */
	pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
	ulSlotWithTokenCount = 0;
	while (1) {
	rv = C_GetSlotList(
	TRUE, pSlotWithTokenList, ulSlotWithTokenCount);
	if (rv != CKR_BUFFER_TOO_SMALL)
	break;
	pSlotWithTokenList = realloc(
	pSlotWithTokenList,
	ulSlotWithTokenList*sizeof(CK_SLOT_ID));
	}
	if (rv == CKR_OK) {
	/* Now use that list of all slots with a token present */
	}
	`free(pSlotWithTokenList);`

C_GetSlotInfo Obtem Informações particular de um Slot	CK_RV CK_ENTRY C_GetSlotInfo(CK_SLOT_ID slotID, CK_SLOT_INFO_PTR pInfo); **C_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo* points to the location that receives the slot information. Return values: CKR_DEVICE_ERROR, CKR_SLOT_ID_INVALID. Example: see **C_GetTokenInfo.**

C_GetTokenInfo
 	
Obtém informações particular sobre um token
	CK_RV CK_ENTRY C_GetTokenInfo(
	CK_SLOT_ID slotID,
	CK_TOKEN_INFO_PTR pInfo
	);
	
	C_GetTokenInfo obtains information about a particular token in the system. *slotID* is the ID of the token's slot; *pInfo* points to the location that receives the token information. Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED.
	
	Example:
	CK_ULONG ulCount;
	CK_SLOT_ID_PTR pSlotList;
	CK_SLOT_INFO slotInfo;
	CK_TOKEN_INFO tokenInfo;
	CK_RV rv;
	rv = C_GetSlotList(FALSE, NULL_PTR, &ulCount);
	if ((rv == CKR_OK) && (ulCount > 0)) {
	pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
	rv = C_GetSlotList(FALSE, pSlotList, &ulCount);
	assert(rv == CKR_OK);
	/* Get slot information for first slot */
	rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
	assert(rv == CKR_OK);
	/* Get token information for first slot */
	rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
	if (rv == CKR_TOKEN_NOT_PRESENT) {
	}
	free(pSlotList);
	}

C_InitToken
 	
Inicializa um Token
	CK_RV CK_ENTRY C_InitToken(
	CK_SLOT_ID slotID,
	CK_CHAR_PTR pPin,
	CK_ULONG ulPinLen,
	CK_CHAR_PTR pLabel
	);
	C_InitToken initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN; *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (must be padded with blank characters). When a token is initialized, all objects that can be destroyed are destroyed (*i.e.*, all except for "indestructible" objects such as keys built into the token). Also, access by the normal user is disabled until the SO sets the normal user's PIN. Depending on the token, some "default" objects may be created, and attributes of some objects may be set to default values.
	If the token has a "protected authentication path", as indicated by the CKR_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin* parameter to **C_InitToken** should be NULL_PTR. During the execution of **C_InitToken**, the SO's PIN will be entered through the protected authentication path. If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_InitToken** can be used to initialize the token. A token cannot be initialized if Cryptoki detects that an application has an open session with it; when a call to **C_InitToken** is made under such circumstances, the call fails with error CKR_SESSION_EXISTS. It may happen that some other application *does* have an open session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other applications using the token. If this is the case, then what happens as a result of the **C_InitToken** call is undefined.
	Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INCORRECT, CKR_SESSION_EXISTS, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_TOKEN_WRITE_PROTECTED.
	Example:
	CK_SLOT_ID slotID;
	CK_CHAR pin[] = {"MyPIN"};
	CK_CHAR label[32];
	CK_RV rv;
	memset(label, ' ', sizeof(label)); memcpy(label, "My first token", sizeof("My first token"));
	rv = C_InitToken(slotID, pin, sizeof(pin), label); if (rv == CKR_OK) {
	}

C_InitPIN
 	
Inicializa um usuário normal PIN
	CK_RV CK_ENTRY C_InitPIN(
	CK_SESSION_HANDLE hSession,
	CK_CHAR_PTR pPin,
	CK_ULONG ulPinLen
	};
	C_InitPIN initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal user's PIN; *ulPinLen* is the length in bytes of the PIN.
	C_InitPIN can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any other state fails with error CKR_USER_NOT_LOGGED_IN. If the token has a "protected authentication path", as indicated by the CKR_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. To initialize the normal user's PIN on a token with such a protected authentication path, the *pPin* parameter to **C_InitPIN** should be NULL_PTR. During the execution of **C_InitPIN**, the SO will enter the new PIN through the protected authentication path. If the token has a protected authentication path other than a PINpad, then it is token-dependent whether or not **C_InitPIN** can be used to initialize the normal user's token access.
	Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY, CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.
	Example:
	CK_SESSION_HANDLE hSession;
	CK_CHAR newPin[]= {"NewPIN"};
	CK_RV rv;
	rv = C_InitPIN(hSession, newPin, sizeof(newPin));
	if (rv == CKR_OK) {
	}

C_OpenSession **Abre a conexão entre uma aplicação e um token**	CK_RV CK_ENTRY C_OpenSession(CK_SLOT_ID slotID, CK_FLAGS flags, CK_VOID_PTR pApplication, CK_NOTIFY Notify, CK_SESSION_HANDLE_PTR phSession); **C_OpenSession** has two distinct functions: it can set up an application callback so that na application will be notified when a token is inserted into a particular slot, or it can open a session between an application and a token in a particular slot. *slotID* is the slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to the notification callback; *Notify* is the address of the notification callback function (see Section 9.17); *phSession* points to the location that receives the handle for the new session. To set up a token insertion callback (instead of actually opening a session), the **CKF_INSERTION_CALLBACK** bit in the *flags* parameter should be set. As a result of setting up this callback, when a token is inserted into the specified slot, the application-supplied callback *Notify* will be called with parameters (0, CKN_TOKEN_INSERTION, pApplication) . If a token is already present when **C_OpenSession** is called, then *Notify* will be called immediately (conceivably even before **C_OpenSession** returns). When **C_OpenSession** is called to set up a token insertion callback, the return code is either CKR_INSERTION_CALLBACK_NOT_SUPPORTED (if the token doesn't support insertion callbacks) or CKR_OK (if the token does support insertion callbacks). When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or more bit flags defined in the **CK_SESSION_INFO** data type. For example, if no bits are set in the *flags* parameter, then **C_OpenSession** attempts to open a shared, read-only session, with certain cryptographic functions being performed in parallel with the application. Any or all of the **CKF_EXCLUSIVE_SESSION**, **CKF_RW_SESSION**, and **CKF_SERIAL_SESSION** bits can be set in the *flags* parameter to modify the type of session requested. If an exclusive session is requested (by setting the **CKF_EXCLUSIVE_SESSION** bit), but is not available (because there is already a session open), **C_OpenSession** returns CKR_SESSION_EXISTS. If a parallel session is requested (by not setting the **CKR_SERIAL_SESSION** bit), but is not supported on this token, then **C_OpenSession** returns CKR_PARALLEL_NOT_SUPPORTED. These two error returns have equal priorities. In a parallel session, cryptographic functions may return control to the application before completing (the return value CKR_FUNCTION_PARALLEL indicates that this condition applies). The application may then call **CKN_SURRENDER** application callback, even if that particular function is actually executing in serial with the application. There may be a limit on the number of concurrent sessions with the token, which may depend on whether the session is "read-only" or "read/write". An attempt to open a session which does not succeed because there are too many existing sessions of some type should return CKR_SESSION_COUNT. If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only readonly sessions may be opened with it. If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code CKR_SESSION_READ_WRITE_SO_EXISTS (see Section 5.5.8). The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not wish to support callbacks, it should pass a value of NULL_PTR as the *Notify* parameter. See Section 9.17 for more information about application callbacks. Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_INSERTION_CALLBACK_NOT_SUPPORTED, CKR_SESSION_COUNT, CKR_SESSION_EXISTS, CKR_SESSION_EXCLUSIVE_EXISTS, CKR_SESSION_PARALLEL_NOT_SUPPORTED, CKR_SESSION_READ_WRITE_SO_EXISTS, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_TOKEN_WRITE_PROTECTED.

C_Login Login em um Token	CK_RV CK_ENTRY C_Login(CK_SESSION_HANDLE hSession, CK_USER_TY PE userType, CK_CHAR_PTR pPin, CK_ULONG ulPinLen); **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to the user's PIN; *ulPinLen* is the length of the PIN. Depending on the user type, if the call succeeds, each of the application's sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User Functions" state. If the token has a "protected authentication path", as indicated by the **CKR_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means that there is some way for a user to be authenticated to the token without having the application send a PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the token itself, or on the slot device. Or the user might not even use a PIN—authentication could be achieved by some fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin* parameter to **C_Login** should be NULL_PTR. When **C_Login** returns, whatever authentication method supported by the token will have been performed; a return value of CKR_OK means that the user was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was denied access. If there are any active cryptographic or object finding operations in a session, and then **C_Login** is successfully executed, it may or may not be the case that those operations are still active. Therefore, before logging in, any active operations should be finished. If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the SO into a session (see Section 5.5.8). An attempt to do this will result in the error code CKR_SESSION_READ_ONLY_EXISTS. Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_PIN_INCORRECT, CKR_SESSION_READ_ONLY_EXISTS, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TYPE_INVALID.
C_EncryptInit Inicializa operações de Criptografia	CK_RV CK_ENTRY C_EncryptInit(CK_SESSION_HANDLE hSession, CK_MECHANISM_PTR pMechanism, CK_OBJECT_HANDLE hKey); **C_EncryptInit** initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key. The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports encryption, must be TRUE. After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** *to actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application must call **C_EncryptInit** again. Return values: CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OPERATION_ACTIVE, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.
C_Encrypt Encripta o dado	CK_RV CK_ENTRY C_Encrypt(CK_SESSION_HANDLE hSession, CK_BYTE_PTR pData, CK_ULONG ulDataLen, CK_BYTE_PTR pE ncryptedData, CK_ULONG_PTR pulEncryptedDataLen); **C_Encrypt** encrypts single-part data. *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted data. **C_Encrypt** uses the convention described in Section 9.2 on producing output. The encryption operation must have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the ciphertext. For some encryption mechanisms, the input plaintext data has certain length constraints (either because the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input data must consist of an integral number of blocks). If these constraints are not satisfied, then **C_Encrypt** will fail with return code CKR_DATA_LEN_RANGE. The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to the same location. **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** and **C_EncryptFinal**. Return values: CKR_BUFFER_TOO_SMALL, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_PARALLEL, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

APÊNDICE 1

Código RSA 56 Bits em C

```
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <conio.h>

#define MM 2147483647      //251
#define AA 48271           //
#define QQ 44488           //
#define RR 3399                    //

int X=251;

int primo(unsigned int n); // algoritmo por fatoração //
int escolha(int min, int max);
int calc_e(unsigned int phi_n, unsigned int d);
void criachave(int *a, int *b, int *c);
int modexp(unsigned long m, unsigned long e,
unsigned int n);
void cifrar(unsigned int e, unsigned int n, unsigned
char mens[], unsigned char c[]);
void decifrar(unsigned int d, unsigned int n, unsigned
char mens[], unsigned char c[]);

void cifrarc(unsigned int e, unsigned int n, unsigned
char mens[], unsigned char c[]);
void decifrarc(unsigned int d, unsigned int n,
unsigned char mens[], unsigned char c[]);

void decifra_arq(unsigned int d, unsigned int n)
{

        FILE * novo;
        FILE * cifrado;
        char n_nome[128];
        char c_nome[128];
        char aux[3],c[4];
        int problema_arquivo = 0;
        unsigned int aux1=0;
        long inicio,fim;
        int i;

        aux[2] = '\0';
        c[3] = '\0';

        printf("\nNome do arquivo cifrado: ");
        scanf("%s", &c_nome);

        printf("\nNome do arquivo destino: ");
        scanf("%s", &n_nome);

        cifrado = fopen(c_nome,"rb");
        novo = fopen(n_nome,"wb");

        if (novo == NULL){
                printf("nao foi possivel abrir o
arquivo para escrita");
                problema_arquivo=1;          }
        if(cifrado==NULL)            {
                printf("nao foi possivel abrir o
arquivo para leitura");
                problema_arquivo=1;          }


        if (!problema_arquivo)
        {
                inicio = clock();
                while (!feof(cifrado))
                {
                        i =
fread(&c,sizeof(char),3,cifrado);
                        if (i > 0)
                        {

        decifrarc(d,n,aux,c);
                                //novo
/*                              aux1 = 0;
                                aux1 = c[0]
<< 8;
                                aux1 += c[1];
                                aux1 =
modexp(aux1,d,n);
                                aux[0] = aux1
>> 8;
                                aux[1] = aux1
& 0x00ff; */
                                //novo

        fwrite(&aux,sizeof(char),2,novo);
                                }
                        }
                fim = clock();
                printf("Tempo: %d", fim-inicio);
                fclose(novo);
                fclose(cifrado);
        }
}

void cifra_arq(unsigned int e, unsigned int n)
{

        FILE * original;
        FILE * cifrado;
        char o_nome[128];
        char c_nome[128];
        char aux[3],c[4];
        int problema_arquivo = 0;
        long inicio,fim;
        unsigned int aux1=0;
```

```
	int i;

	aux[2] = '\0';
	c[3] = '\0';

	printf("\nNome do arquivo original: ");
	scanf("%s", &o_nome);

	printf("\nNome do arquivo destino: ");
	scanf("%s", &c_nome);

	original = fopen(o_nome,"rb");
	cifrado = fopen(c_nome,"wb");

	if (original == NULL)		{
		printf("nao foi possivel abrir o
arquivo para leitura");
		problema_arquivo=1;		}
	if(cifrado==NULL)		{
		printf("nao foi possivel abrir o
arquivo para escrita");
		problema_arquivo=1;		}


	if (!problema_arquivo)
	{
		inicio = clock();
		while (!feof(original))
		{
			i =
fread(&aux,sizeof(char),2,original);
			if (i > 0)
			{

	cifrarc(e,n,aux,c);
					//novo
/*				aux1 = 0;
				aux1 = aux[0]
<< 8;
				aux1 +=
aux[1];
				aux1 =
modexp(aux1,e,n);
				c[0] = aux1
>> 8;
				c[1] = aux1 &
0x00ff;*/
					//novo

	fwrite(&c,sizeof(char),3,cifrado);
			}
		}
		fim = clock();
		printf("Tempo: %d", fim-inicio);
		fclose(original);
		fclose(cifrado);
	}
}

int main()
{
	int opc;
	unsigned char mens[50];
	unsigned char c[50];
	unsigned int e,d,n;
	int i;

	while (opc != 6)
	{
		printf("\n1 - Criar chaves");
		printf("\n2 - Cifrar");
		printf("\n3 - Decifrar");
		printf("\n4 - Cifrar arquivo");
		printf("\n5 - Decifrar arquivo");
		printf("\n6 - Sair");
		printf("\nOpcao: ");
		scanf("%d", &opc);
		switch (opc)
		{
		case 1:
			{

	/*printf("\nphi_n: ");
				scanf("%d",
&phi_n);
				printf("\nd: ");
				scanf("%d",
&d);

	euclidesext(phi_n,d);*/
				e = 1;
				d = 0;
				while (e > d)

	criachave(&e,&d,&n);
				printf("\nd
%u", d);
				printf("\ne
%u", e);
				printf("\nn
%u", n);
				break;
			}
		case 2:
			{

	printf("Mensagem para cifrar: ");
//				scanf("%s",
&mens);
				scanf("%s",
&mens);

	cifrar(e,n,mens,c);


	printf("\nCifrado: ");
				i =0;
				while (c[i] !=
'\0')
				{

	printf(" %x ", c[i]);
					i++;
				}
				printf("\n");

				break;
			}
```

```
                case 3:
                        {

        decifrar(d,n,mens,c);

        printf("Decifrado: %s", mens);
                                break;
                        }
                case 4:
                        {
                                cifra_arq(e,n);
                                break;
                        }
                case 5:
                        {

        decifra_arq(d,n);
                                break;
                        }
                case 6:
                        {
        exit(0);
                                break;
                        }
                default:
                        printf("Opcao invalida");
                }
        }
}

int primo(unsigned int n) // verifica se é primo
// algoritmo por fatoração //
{
        int i;
        i = (n / 2) + 1;
        while(n % i != 0)
                i--;
        if (i==1)
                return 1; // é primo
        else
                return 0; // não é primo
}

int escolha(int min, int max)  //escolhe um numero
primo aleatório
{
        int aux=0;
        while(aux<min)
        {
                X=AA*(X%QQ)-
RR*(long)(X/QQ);  // Calcula o proximo numero
aleatorio
                if (X<0) X+=MM;             //
(Knuth, 1997)
                aux = X % max;
                if (aux>min)
                        while (!primo(aux))   //
verificar se é primo
                                aux++;
        }
        return aux;
}
```

```
int calc_e(unsigned int phi_n, unsigned int d)
// algoritmo de Euclides Estendido//
{
        int u[3];
        int v[3];
        int t[3];
        int q;
        u[0] = 1;
        u[1] = 0;
        u[2] = phi_n;
        v[0] = 0;
        v[1] = 1;
        v[2] = d;
        while (v[2] != 0)
        {
                q = (u[2] / v[2]);
                t[0] = u[0] - (v[0] * q);
                t[1] = u[1] - (v[1] * q);
                t[2] = u[2] - (v[2] * q);
                u[0] = v[0];
                u[1] = v[1];
                u[2] = v[2];
                v[0] = t[0];
                v[1] = t[1];
                v[2] = t[2];
        }
        return u[1];
}

void criachave(int *a, int *b, int *c)
{
        unsigned int long p,q,d,e,n;
        p = escolha(500000000,1000000000);  // os
valores escolhidos devem estar entre
                // 256 e 4000 para gerar chaves de
24 bits
        q = p;
        while (p == q)
                q = escolha(9999999,99999999);
        n = p * q;
        d = escolha(9999999,99999999);
        e = calc_e(((p-1)*(q-1)),d);
        *a = e;
        *b = d;
        *c = n;
}

int BLAKLEY(int a, int b, int n) //multiplicação
modular - Otimizado
// entradas: a,b,n
// saida: R = (a * b) mod n
{
        unsigned int aux;
        int R;
        R = 0;
        for (aux = 0x80000000; aux > 0; aux >>= 1)
        {
                R = (R << 1 );
                if((a & aux) != 0) R += b;
                if (R > n) R -= n;
                if (R > n) R -= n;
        }
        return R;
}
```

```
int BLAKLEY1(int a, int b, int n) //multiplicação
modular - Original
// entradas: a,b,n
// saida: R = (a * b) mod n
{
	int i,aux=1;
	int a_[56];
	int R;

	for (i=0;i<56;i++)
	{
		a_[i] = (!((a & aux) == 0));
		aux+=aux;
	}
	R = 0;
	for (i=0;i<56;i++)
	{
		R = 2 * R + (a_[56-1-i] * b);
		R = R % n;
	}
	return R;
}

int modexp(unsigned long m, unsigned long e,
unsigned int n)
/*
algoritmo binário para exponenciação
m = base
e = expoente
n = modulo
t = nº de bits
*/
{
	unsigned long temp=1;
	int j,t;
	int b[56];
	int aux=1;
	int achou;
//	unsigned long mult;

	for (j=0;j<56;j++)
	{
		b[j] = !((e &aux) == 0);
		aux += aux;
	}

	j = 55;
	achou = 0;
	while ((!achou) && (j >= 0))
	{
		if (b[j] == 1)
		{
			t = j+1;
			achou = 1;
		}
		j--;
	}

	for (j=t;j>=0;j--)
	{
		temp = BLAKLEY(temp,temp,n);
		if (b[j] == 1)
			temp =
BLAKLEY(temp,m,n);
	}
	return temp;
}


void cifrar(unsigned int e, unsigned int n, unsigned
char mens[], unsigned char c[])
{
	//C = M^e mod n
	//c = modexp(m,e,n)
	int i=0,j=0;
	unsigned int aux;
	int acabou=0;
	while (!acabou)
	{
		aux = 0;
		if (mens[i] != '\0')
			aux = mens[i] << 8;
		else
			acabou++;
		i++;
		if (mens[i] != '\0' && !acabou)
			aux += mens[i];
		else
			acabou++;
//		printf(" %x ", aux);
		aux = modexp(aux,e,n);

		c[j] = aux >> 16; j++;
		c[j] = ((aux >> 8) & 0x00ff); j++;
		c[j] = aux & 0x00ff; j++;
		i++;
	}
	c[j] = '\0';
}

void decifrar(unsigned int d, unsigned int n, unsigned
char mens[], unsigned char c[])
{
	//M = C^d mod n
	//m = modexp(c,d,n)
	int i=0,j=0;
	unsigned int aux;
	int acabou=0;
	while (!acabou)
	{
		aux = 0;
		if (c[i] != '\0')
			aux = c[i] << 16;
		else
			acabou++;
		i++;
		if (c[i] != '\0' && !acabou)
			aux += c[i] << 8;
		else
			acabou++;
		i++;
		if (c[i] != '\0' && !acabou)
			aux += c[i];
		else
			acabou++;
```

```
                aux = modexp(aux,d,n);

                mens[j] = aux >> 8; j++;
                mens[j] = aux & 0x00ff; j++;
                i++;
        }
        mens[j] = '\0';
}

void cifrarc(unsigned int e, unsigned int n, unsigned
char mens[], unsigned char c[])
{
        //C = M^e mod n
        //c = modexp(m,e,n)
        int i=0,j=0;
        unsigned int aux;
        aux = 0;
        if (mens[i] != '\0')
                aux = mens[i] << 8;
        i++;
        if (mens[i] != '\0')
                aux += mens[i];

        aux = modexp(aux,e,n);

        c[j] = aux >> 16; j++;
        c[j] = ((aux >> 8) & 0x00ff); j++;
        c[j] = aux & 0x00ff; j++;
        i++;
        c[j] = '\0';
}

void decifrarc(unsigned int d, unsigned int n,
unsigned char mens[], unsigned char c[])
{
        //M = C^d mod n
        //m = modexp(c,d,n)
        int i=0,j=0;
        unsigned int aux;
        aux = 0;
        if (c[i] != '\0')
                aux = c[i] << 16;
        i++;
        if (c[i] != '\0')
                aux += c[i] << 8;
        i++;
        if (c[i] != '\0')
                aux += c[i];

        aux = modexp(aux,d,n);

        mens[j] = aux >> 8; j++;
        mens[j] = aux & 0x00ff; j++;
        i++;
        mens[j] = '\0';
}
```

APÊNDICE 2

O Código RSA 512 Bits em VHDL utilizado para cifrar mensagem é apresentado a seguir:

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

ENTITY d_RSA is
	port(
		C : in std_logic_vector(511 downto 0);
		D : in std_logic_vector(510 downto 0);
		n : in std_logic_vector(511 downto 0);
		CLK : in std_logic;
		M : out std_logic_vector(504 downto 0);
		PRONTO : out std_logic
	);
end d_RSA;

architecture arc of d_RSA is
begin
	process(clk)
	variable temp: std_logic_vector(518 downto 0);
	variable flag: std_logic_vector(2 downto 0);
	variable A,B,R : std_logic_vector(518 downto 0);
	variable i,j : integer;
	begin
		if clk'event and clk = '1' then
			if flag = "000" then
				j := 17; --"010001"; --17
			temp :=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000001"; --1


				flag := "001";
			elsif flag = "001" then
				if j >=0 then
					A := temp;
					B := temp;
```

```
                    R :=
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000"; --1

                                        i := 0; -- "000000";
                                        flag := "010";
                                else
                                        PRONTO <= '1';    --Atribuição dos valores aos
                                        M <= temp(504 downto 0);    --sinais de saída.
                                end if;
                        elsif flag = "011" then
                                temp := R;
                                if D(j) = '1' then
                                        A := temp;
                                        B(511 downto 0 ) := C;
                                        B(518 downto 512) := "0000000";
                    R :=
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000"; --1

                                        i := 0; --"000000";
                                        flag := "100";
                                else
                                        j := j - 1;
                                        flag := "001";
                                end if;
                        elsif flag = "101" then
                                temp := R;
                                j := j - 1;
                                flag := "001";
                        elsif flag = "010" or flag = "100" then -- BLAKLEY
                                if i < 512 then
                                        R := SHL(R,"1");
                                        if A(511-i) = '1' then R := R + B; end if;
                                        if R > n then R := R - n; end if;
                                        if R > n then R := R - n; end if;
                                        i := i + 1;
                                else
                                        flag := flag + 1;
                                end if;
                        end if;
                end if;
```

```
        end process;
end arc;
```

O código do RSA 512 bits utilizado para decifrar mensagem é apresentado a seguir:

Mod_Exp.vhd

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

ENTITY mod_exp is
        port(
                M : in std_logic_vector(504 downto 0);
                E : in std_logic_vector(510 downto 0);
                n : in std_logic_vector(511 downto 0);
                CLK : in std_logic;
                C : out std_logic_vector(504 downto 0);
                PRONTO : out std_logic
        );
end mod_exp;

architecture arc of mod_exp is
begin
        process(clk)
        variable temp: std_logic_vector(518 downto 0);
        variable flag: std_logic_vector(2 downto 0);
        variable A,B,R : std_logic_vector(518 downto 0);
        variable i,j : integer;
        begin
                if clk'event and clk = '1' then
                        if flag = "000" then
                                j := 17; --"010001"; --17
                temp :=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000001"; --1

                                flag := "001";
                        elsif flag = "001" then
                                if j >=0 then
                                        A := temp;
```

```
                    B := temp;
            R :=
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000"; --1

                    i := 0; -- "000000";
                    flag := "010";
                else
                    PRONTO <= '1';    --Atribuição dos valores aos
                    C <= temp(504 downto 0);    --sinais de saída.
                end if;
            elsif flag = "011" then
                temp := R;
                if E(j) = '1' then
                    A := temp;
                    B(504 downto 0 ) := M;
                    B(518 downto 505) := "0000000000000000";
            R :=
"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000"; --1

                    i := 0; --"000000";
                    flag := "100";
                else
                    j := j - 1;
                    flag := "001";
                end if;
            elsif flag = "101" then
                temp := R;
                j := j - 1;
                flag := "001";
            elsif flag = "010" or flag = "100" then -- BLAKLEY
                if i < 518 then
                    R := SHL(R,"1");
                    if A(517-i) = '1' then R := R + B; end if;
                    if R > n then R := R - n; end if;
                    if R > n then R := R - n; end if;
                    i := i + 1;
                else
                    flag := flag + 1;
                end if;
```

```
                                end if;
                        end if;
                end process;
end arc;
```

APÊNDICE 3

Máquina de estados Finito Rsa no padrão PKCS#11 versão 6.2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity serial is
   port (
      clk: in STD_LOGIC;
      s_led: out STD_LOGIC;
      s_out: out STD_LOGIC;
      freq: out STD_LOGIC;
            CTS: out STD_LOGIC     );
           --CH: out std_logic_vector(4 downto 0));
end serial;

architecture Behavioral of serial is

-- Definindo configurações para o TOKEN
constant TOKEN0 : std_logic_vector(7 downto 0):="01000001"; -- A
constant TOKEN1 : std_logic_vector(7 downto 0):="01000010"; -- B
constant TOKEN2 : std_logic_vector(7 downto 0):="01000011"; -- C
constant TOKEN3 : std_logic_vector(7 downto 0):="01000100"; -- D
constant TOKEN4 : std_logic_vector(7 downto 0):="01000101"; -- E
constant TOKEN5 : std_logic_vector(7 downto 0):="01000110"; -- F
constant ESP : std_logic_vector(7 downto 0):="00100000"; -- 13 quebra de linha

-- Estados da maquina de estados;
constant ZERO      : std_logic_vector(3 downto 0):="0001";
constant UM        : std_logic_vector(3 downto 0):="0010";
constant DOIS      : std_logic_vector(3 downto 0):="0011";
constant TRES      : std_logic_vector(3 downto 0):="0100";
constant QUATRO : std_logic_vector(3 downto 0):="0101";
constant CINCO : std_logic_vector(3 downto 0):="0110";
constant SEIS : std_logic_vector(3 downto 0):="0111";
constant SETE      : std_logic_vector(3 downto 0):="1000";
constant OITO      : std_logic_vector(3 downto 0):="1001";
constant NOVE      : std_logic_vector(3 downto 0):="1010";
constant CH : std_logic_vector (4 downto 0):="00100";
```

```
-- Constantes para o RSA

constant E : std_logic_vector(16 downto 0):="01011100101011101";                    -- 0B95D
constant n : std_logic_vector(23 downto 0):="001100011110001000111111"; -- 31E23F
constant D : std_logic_vector(16 downto 0):="01110101010101101";                    -- 0EAAD
signal estado          : std_logic_vector(3 downto 0):="0001";
signal B1 : std_logic_vector(7 downto 0); -- Guarda info do TOKEN
signal B2 : std_logic_vector(7 downto 0); -- Guarda info do TOKEN
signal VAR : std_logic_vector(15 downto 0); -- Guarda info do TOKEN

signal S_P_CRIP : std_logic:='0'; -- Libera a criptografia do RSA
signal S_P_DECRIP : std_logic:='0'; -- Libera a decriptorafia do RSA

signal S_D_CRIP : std_logic_vector(23 downto 0); -- Registrador interno do dado criptografado
signal S_D_DECRIP :std_logic_vector(15 downto 0); -- Registrador  interno do dado decriptografado

signal LibTRES, LibSEIS, LibNOVE: std_logic:='0';

-- Transmissao serial
signal S_CLK : std_logic;
signal conta: integer:=0;
--signal contb , carac: integer:=0;
signal envch1 :std_logic_vector(11 downto 0);
signal envch2 :std_logic_vector(11 downto 0);
signal envenc1 :std_logic_vector(11 downto 0);
signal envenc2 :std_logic_vector(11 downto 0);
signal envenc3 :std_logic_vector(11 downto 0);
signal envdec1 :std_logic_vector(11 downto 0);
signal envdec2 :std_logic_vector(11 downto 0);
signal envesp :std_logic_vector(11 downto 0);
signal cont:integer:=0;  -- cont pisca led
signal clk_serial:std_logic;     -- clk 300 bps
signal enviar:std_logic_vector(11 downto 0);

begin
-- LED (1 segundo)
        process (clk)
        begin
          if clk'event and clk = '1' then
                cont <= cont + 1;
                if cont < 25000000 then
                        s_led <= '0';
                else
                        s_led <= '1';
                end if;
```

```
                    if cont = 50000000 then
                            cont <= 0;
                    end if;
              end if;
            end process;

envesp <= "01" & not(ESP) & "10"; -- Espaco simples no formato serial
   process(CLK)
   variable microcont: integer:=0;
   begin
   if CLK'event and CLK='1' then
    case estado is
    when ZERO=>
          B1 <="00000000";
                  B2 <="00000000";
              microcont:= microcont+1;
              envch1 <= "000000000000";
              envch2 <= "000000000000";
              envenc1 <= "000000000000";
              envenc2 <= "000000000000";
              envenc3 <= "000000000000";
              envdec1 <= "000000000000";
              envdec2 <= "000000000000";
              if microcont = 10 then
           estado <= UM;
       end if;
    when UM =>
          microcont := 0;
    case CH is
      when "00000"          => B1 <= TOKEN0;
                               B2 <= TOKEN1; -- AB

      when "00001"          => B1 <= TOKEN0;
                               B2 <= TOKEN2; -- AC

                    when "00010"      => B1 <= TOKEN1;
                                         B2 <= TOKEN2; -- BC

      when "00011"          => B1 <= TOKEN2;
                               B2 <= TOKEN3; -- BD

      when "00100"          => B1 <= TOKEN3;
                               B2 <= TOKEN4; -- CD

      when "00101"          => B1 <= TOKEN4;
```

```
                    B2 <= TOKEN5; -- EF

    when others  => estado <= ZERO;
            end case;
        estado <= DOIS;

when DOIS => -- monta informações do TOKEN
                envch1 <= "01" & not(B1) & "10"; --Formato serial
                envch2 <= "01" & not(B2) & "10"; --Formato serial
                VAR <= B1&B2;
                estado <= TRES;

        when TRES =>  -- envia informações do TOKEN serialmente
                if LIBTRES = '1' then -- Indica o envio
                    estado <= QUATRO;
                end if;

when QUATRO => -- Libera a Criptografia do dado
        if S_P_CRIP = '1' then -- Indica que acabou a criptografia
          estado <= CINCO;
        end if;
when CINCO => -- Monta o dado criptografado
        envenc1 <= "01" & not(S_D_CRIP(7 downto 0)) & "10";   --Formato serial
                envenc2 <= "01" & not(S_D_CRIP(15 downto 8)) & "10";  --Formato serial
                envenc3 <= "01" & not(S_D_CRIP(23 downto 16)) & "10"; --Formato serial
                estado <= SEIS;

        when SEIS => -- Transmite o dado criptografado
                if LIBSEIS = '1' then -- Indica o envio
                        estado <= SETE;
                end if;

when SETE => -- Libera a decriptografia
        if S_P_DECRIP = '1' then -- Indica que acabou a decriptografia
          estado <= OITO;
        end if;

        when OITO => -- Monta o dado decriptografado
                envdec1 <= "01" & not(S_D_DECRIP(15 downto 8)) & "10";   --Formato serial
                        envdec2 <= "01" & not(S_D_DECRIP(7 downto 0)) & "10";  --Formato serial
                estado <= NOVE; -- Indica que acabou a decriptografia

when NOVE => -- Transmite o dado decriptografado
    if LIBNOVE = '1' then
                estado <= ZERO;
```

```
                end if;
            when others=> estado <= ZERO;
    end case;
    end if;
end process;
--- Processo Encripta RSA
        process(clk)
        variable temp: std_logic_vector(31 downto 0);
        variable flag: std_logic_vector(2 downto 0);
        variable A,B,R : std_logic_vector(31 downto 0);
        variable i,j : integer;
        begin
        if clk'event and clk = '1' then
                --if estado = ZERO then
                --   S_D_CRIP <= "000000000000000000000000";
                --   S_P_CRIP <= '0';
                --   temp :="00000000000000000000000000000001";
                --   flag :="000";
                --   A :="00000000000000000000000000000000";
                --   B :="00000000000000000000000000000000";
                --   R :="00000000000000000000000000000000";
                --   i:=0;
                --   j:=17;
                --end if;
                if estado = QUATRO then
                        if flag = "000" then
                                j := 17; --"010001"; --17
                                temp := "00000000000000000000000000000001"; --1
                                flag := "001";
                        elsif flag = "001" then
                                if j >=0 then
                                        A := temp;
                                        B := temp;
                                        R := "00000000000000000000000000000000";
                                        i := 0; -- "000000";
                                        flag := "010";
                                else
                                        S_D_CRIP <= temp(23 downto 0); -- Registrador interno do dado
criptografado
                                        S_P_CRIP <= '1';

                                end if;
                        elsif flag = "011" then
                                temp := R;
                                if E(j) = '1' then
                                        A := temp;
```

```
                                        B(15 downto 0 ) := VAR;
                                        B(31 downto 16) := "0000000000000000";
                                        R := "00000000000000000000000000000000";
                                        i := 0; --"000000";
                                        flag := "100";
                                else
                                        j := j - 1;
                                        flag := "001";
                                end if;
                        elsif flag = "101" then
                                temp := R;
                                j := j - 1;
                                flag := "001";
                        elsif flag = "010" or flag = "100" then -- BLAKLEY
                                if i < 32 then
                                        R := SHL(R,"1");
                                        if A(31-i) = '1' then R := R + B; end if;
                                        if R > n then R := R - n; end if;
                                        if R > n then R := R - n; end if;
                                        i := i + 1;
                                else
                                        flag := flag + 1;
                                end if;
                        end if;
                end if; -- lib_rsa
        end if;
end process;

-- Processo DECRIPTA RSA

        process(clk)
        variable temp: std_logic_vector(31 downto 0);
        variable flag: std_logic_vector(2 downto 0);
        variable A,B,R : std_logic_vector(31 downto 0);
        variable i,j : integer;
        begin
                if clk'event and clk = '1' then
                -- if estado = ZERO then
                --    S_D_DECRIP <= "0000000000000000";
                --    S_P_DECRIP <= '0';
                --    temp :="00000000000000000000000000000000";
                -- flag :="000";
                -- A :="00000000000000000000000000000000";
                -- B :="00000000000000000000000000000000";
                -- R :="00000000000000000000000000000000";
```

```
--   i:=0;
--   j:=0;
--   end if;

  if estado = SETE then
        if flag = "000" then
              j := 17; --"010001"; --17
              temp := "00000000000000000000000000000001"; --1
              flag := "001";
        elsif flag = "001" then
              if j >=0 then
                    A := temp;
                    B := temp;
                    R := "00000000000000000000000000000000";
                    i := 0; -- "000000";
                    flag := "010";
              else
                    S_D_DECRIP <= temp(15 downto 0); -- Registrador de armazenamento
                    S_P_DECRIP <= '1';
              end if;
        elsif flag = "011" then
              temp := R;
              if D(j) = '1' then
                    A := temp;
                    B(23 downto 0 ) := S_D_CRIP; -- Dado Criptografado
                    B(31 downto 24) := "00000000";
                    R := "00000000000000000000000000000000";
                    i := 0; --"000000";
                    flag := "100";
              else
                    j := j - 1;
                    flag := "001";
              end if;
        elsif flag = "101" then
              temp := R;
              j := j - 1;
              flag := "001";
        elsif flag = "010" or flag = "100" then -- BLAKLEY
              if i < 32 then
                    R := SHL(R,"1");
                    if A(31-i) = '1' then R := R + B; end if;
                    if R > n then R := R - n; end if;
                    if R > n then R := R - n; end if;
                    i := i + 1;
              else
```

```
                                        flag := flag + 1;
                                    end if;
                                end if;
                            end if; -- end S_P_CRIP
                        end if;
                end process;

-- Transmissao serial           300bps
process(CLK)
begin
 if CLK'event and CLK='1' then
    conta<=conta+1;
    if conta < 83333 then
    -- if conta < 4 then
      S_CLK <='0';
    else
      S_CLK <='1';
    end if;
    if conta = 166666 then
    --if conta = 8 then
      conta <= 0;
    end if;
  end if;
end process;
process(s_clk)
            variable contb, carac : integer:=0;
            begin
             if s_clk'event and s_clk='1' then
               case estado is
               when ZERO =>
                                CTS<='0';
                                LIBTRES<='0';
                                LIBSEIS<='0';
                                LIBNOVE<='0';
                                contb:=0;
                                carac:=0;

               when TRES => -- Envia o token serialmente

                                if contb = 0 then CTS<='1'; else CTS<='0'; end if;
                                contb := contb+1;

                                if contb > 0 and contb < 13 then
                                        if carac = 0 then
                                                s_out <= envch1(contb-1);
```

```
                end if;
                if carac = 1 then
                        s_out <= envch2(contb-1);
                end if;
                if carac = 2 then
                        s_out <= envesp(contb-1);
                end if;
        end if;
        if contb = 13 then
                contb := 0;
                carac := carac + 1;
        end if;
        if carac = 3 then
          contb := 0;
          carac := 0;
          LIBTRES<='1';
        end if;

when SEIS => -- Envia o dado Criptografado

if contb = 0 then CTS<='1'; else CTS<='0'; end if;
        contb :=contb+1;

        if contb > 0 and contb < 13 then
                if carac = 0 then
                        s_out <= envenc1(contb-1);
                end if;
                if carac = 1 then
                        s_out <= envenc2(contb-1);
                end if;
                if carac = 2 then
                        s_out <= envenc3(contb-1);
                end if;
                if carac = 3 then
                        s_out <= envesp(contb-1);
                end if;

        end if;
        if contb = 13 then
                contb := 0;
                carac := carac + 1;
        end if;
        if carac = 4 then
          carac := 0;
          LIBSEIS<='1';
```

```
                end if;
            when NOVE =>-- Envia o dado decriptografado
            if contb = 0 then CTS<='1'; else CTS<='0'; end if;
                contb :=contb+1;
                if contb > 0 and contb < 13 then
                    if carac = 0 then
                        s_out <= envdec1(contb-1);
                    end if;
                    if carac = 1 then
                        s_out <= envdec2(contb-1);
                    end if;
                    if carac = 2 then
                        s_out <= envesp(contb-1);
                    end if;

                end if;
                if contb = 13 then
                    contb := 0;
                    carac := carac + 1;
                end if;
                if carac = 3 then
                  carac := 0;
                  LIBNOVE<='1';
                end if;
            when others => CTS<='0';
            end case;
        end if;
end process;
end Behavioral;
```