Dep. de Eng. de Computação e Automação Industrial
Faculdade de Engenharia Elétrica e Computação,
Universidade Estadual de Campinas.

CO-PROCESSADOR PARA ALGORITMOS DE CRIPTOGRAFIA ASSIMÉTRICA

Autor: Maurício Araújo Dias

Orientador: Prof. Dr. José Raimundo de Oliveira

Banca:
Prof. Dr. José Raimundo de Oliveira – Presidente
Profa. Dra. Alice M. H. Tokarnia
Prof. Dr. Marco Aurélio Amaral Henriques
Prof. Dr. Ricardo Dahab

Dissertação apresentada como requisito parcial à obtenção do grau de **Mestre em Engenharia Elétrica**.

Campinas, fevereiro de 2002

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

D543c

Dias, Maurício Araújo
Co-processador para algoritmos de criptografia assimétrica / Maurício Araújo Dias.--Campinas, SP: [s.n.], 2002.

Orientador: José Raimundo de Oliveira.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Criptografia. 2. Criptografia de dados (Computação). 3. Curvas elípticas. 4. Hardware. 5. Circuitos integrados. 6.VHDL (Linguagem descritiva de hardware). I. Oliveira, José Raimundo de. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Resumo

Este trabalho tem como objetivo o desenvolvimento de um co-processador para algoritmos de criptografia assimétrica. Trata-se de um co-processador que pode servir de base para a implementação de algoritmos de criptografia assimétrica, não apenas de um dispositivo dedicado a um único algoritmo criptográfico. Para tanto, ele dispõe de uma biblioteca de módulos de circuitos que implementam rotinas básicas úteis a vários desses algoritmos. A implementação é feita em um dispositivo do tipo FPGA. Para testar o funcionamento do co-processador foi escolhido o algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas. Os testes práticos do co-processador apóiam-se no uso de curvas elípticas distintas e de diferentes pontos pertencentes a cada uma dessas mesmas curvas.

Abstract

This work has as main objective the development of a co-processor for asymmetric cryptography algorithms. It is a co-processor that can serve for the implementation of asymmetric cryptography algorithms. It isn't a device dedicated to only a cryptographic algorithm. So, it uses a library of *hardware* modules that implement basic routines useful to several of these algorithms. The implementation is made in a FPGA device. In order to test the operation of this co-processor, we choose the asymmetric cryptography algorithm based on the elliptic curve discrete logarithm problem. The practical tests of the co-processor are based on the use of distinct elliptical curves and different points over these same curves.

Índice

Glossário

ACC (*Accumulator*) – registrador acumulador.

ALU (*Arithmetic and Logic Unit*) – unidade lógico-aritmético.

ASIC (*Application-Specific Integrated Circuit*) – um *chip* que pode ser projetado para implementar uma função digital.

AU (*Arithmetic Unit*) – unidade aritmética.

AUC (*Arithmetic Unit Controler*) – controlador da unidade aritmética.

BSR (*Bit Status Register*) – registrador do estado do bit.

Carry look ahead – Técnica digital utilizada para acelerar as operações de soma pela previsão da propagação ou geração de "vai-um" entre os dois operandos.

CHES *(Cryptographic Hardware and Embedded Systems)* – *workshop* que apresenta o estado da arte em sistemas criptográficos implementados em *hardware*.

CPU (*Central Processing Unit*) – unidade central de processamento.

DSP (*Digital Signal Processor*) – processador digital de sinais.

ECC (*Elliptic Curve Cryptography*) – criptografia baseada em curvas elípticas.

ECP *(Elliptic Curve Processor)* – processador para curvas elípticas.

EEPROM (*Eletrically Erasable Programmable Read-Only Memory*) – memória somente para leitura, que podia ser desgravada e regravada eletricamente muitas vezes.

EPROM (*Erasable Programmable Read-Only Memory*) – memória somente para leitura, que podia ser desgravada através da exposição do *chip* a raios ultra-violeta. Em seguida, ela podia ser regravada.

FPGA (*Field Programmable Gate Array)* – é um tipo de dispositivo programável, isto é, uma classe de *chips* com propósito geral que podem ser configurados por uma variedade de aplicações.

HDL *(Hardware Description Language)* – linguagem de descrição de *hardware*.

LSI (*Large Scale Integration*) – integração em larga escala.

Mainframe – computador de grande porte

MC (*Main Controler*) – controlador principal.

PC (*Personal Computer*) – computador pessoal.

PLD (*Programmable Logic Device*) – arquitetura base para a implementação de circuitos diversos, através da programação lógica criada pelo próprio usuário final.

PROM (*Programmable Read-Only Memory*) – memória somente para leitura, que é programável pelo usuário final somente uma única vez.

PSW (*Program Status Word*) – palavra de estado do programa.

RAM (*Randomic Access Memory*) – memória de acesso randômico.

ROM (*Read-Only Memory*) – memória somente para leitura.

RSA (Rivest-Shamir-Adleman) – um algoritmo de criptografia assimétrica.

Smart-card – Dispositivo no formato de um cartão de crédito que têm embutido um circuito de processamento e de armazenamento. Candidato a substituir os cartões magnéticos de créditos.

TEMP (*Temporary*) – registrador temporário.

Verilog – Linguagem de descrição de hardware

VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) – uma linguagem de descrição de *hardware*.

VLSI (Very *Large Scale Integration*) – integração em escala muito larga.

Capítulo 1
Introdução

1.1 M1 – Um co-processador para algoritmos de criptografia assimétrica.

O presente trabalho descreve o projeto e desenvolvimento de um co-processador para algoritmos de criptografia assimétrica. A este projeto foi dado o nome de **Co-processador M1**. Não se trata de um dispositivo dedicado a um único algoritmo criptográfico. Pelo contrário, ele propõe uma arquitetura básica e uma biblioteca de módulos de circuitos, descritos em sua maior parte em VHDL (*Very High Speed Integrated Circuit Hardware Description Language*), que implementam rotinas básicas úteis a vários desses algoritmos.

O algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas (vide [6], [7], [10], [11] e [12]), também conhecido como algoritmo de ECC (*Elliptic Curve Cryptography*), foi escolhido para demonstrar as operações que podem ser realizadas pelo co-processador M1. As simulações do co-processador M1 são baseadas no uso de curvas elípticas distintas e de diferentes pontos dispostos sobre essas mesmas curvas.

Ele foi compilado de modo a dar prioridade para uma menor área em detrimento a uma maior velocidade. A implementação do co-processador M1 é feita em um dispositivo do tipo FPGA (*Field Programmable Gate Array)* (vide [20]), da família APEX20K da Altera (vide [2], [3], [4] e [5]). O co-processador M1 foi inteiramente compilado, sintetizado e programado utilizando-se o software Quartus II (vide [1]), também da Altera.

1.2 Considerações iniciais

Atualmente, milhões de computadores encontram-se interligados através de redes de computadores. Por esses meios, circulam inúmeros tipos de informações: desde simples mensagens de correio eletrônico entre namorados, arquivos de jogos e de músicas até informações altamente confidenciais, como números de cartões de crédito, transações bancárias, declarações de imposto de renda, projetos secretos de empresas, etc. As informações que apresentam caráter sigiloso demandam a presença de mecanismos capazes

de prover segurança contra espionagem, adulteração ou mesmo desvio das mesmas. Segundo Willian Stallings, em [17]: "Segurança em um ambiente de rede depende da aplicação prática de um conjunto de princípios fundamentais de segurança. As ferramentas básicas de segurança em redes procuram garantir a proteção do conteúdo das informações durante as transmissões, assegurar a autenticidade das partes envolvidas em uma interação e impedir que intrusos utilizem as redes para tentar danificar sistemas". Tais ferramentas empregam os chamados algoritmos de criptografia. Esses algoritmos procuram dificultar a ação mal-intencionada de pessoas que visam ter acesso às mensagens sigilosas. Entretanto, de tempos em tempos, uma dessas ações é bem sucedida, o que coloca em dúvida a eficiência do algoritmo. Por esse motivo, esses mecanismos de segurança em redes de computadores sofrem constantes aprimoramentos.

De uma forma geral, a transmissão segura de informações por meio de uma rede de computadores depende da codificação das mesmas através do emprego de representantes de uma classe de algoritmos de criptografia, conhecidos como algoritmos de criptografia simétrica (vide [17]). Tais algoritmos empregam uma chave secreta, a qual é um código binário compartilhado entre o transmissor e o receptor da informação codificada. Este código é o único segredo capaz de ser usado para decodificar tal informação e torná-la novamente inteligível. Entretanto, para que essa chave possa ser compartilhada entre as partes envolvidas na comunicação, de maneira realmente secreta, é necessário o emprego de representantes de uma segunda classe de algoritmos de criptografia, chamados de algoritmos de criptografia assimétrica. Os algoritmos pertencentes a essas duas classes são apresentados em detalhes no apêndice A.

Para alcançarem êxito em suas funções, os algoritmos de criptografia assimétrica necessitam empregar chaves com muitos bits. O RSA (Rivest-Shamir-Adleman) (vide [17]), por exemplo, emprega chaves com pelo menos 1024 bits. Já o algoritmo baseado em ECC, quando usa uma chave de 160 bits, alcança o mesmo grau de segurança do RSA, quando este último usa uma chave de 1024 bits. Embora ambos empreguem chaves com muitos bits, existe uma diferença significativa entre o número de bits das chaves do RSA e do algoritmo baseado em ECC, para um mesmo nível de segurança. É justamente esse menor número de bits que está chamando a atenção dos pesquisadores para ECC.

O grande número de bits das chaves usadas por esses algoritmos faz com que a implementação deles em *software* gere sistemas de segurança relativamente lentos, quando comparados aos mesmos implementados diretamente por *hardware*. Isso ocorre porque esses programas são geralmente executados sobre um processador de propósito geral (vide [16]) de 64 bits, o que significa que eles manipulam as chaves em fatias de 64. Como os processadores não conseguem lidar com uma chave inteira de uma só vez, eles são obrigados a realizar uma mesma operação aritmética ou lógica diversas vezes sobre diferentes porções da mesma chave, a fim de completar toda a operação. Implementações em *hardware*, por outro lado, permitem trabalhar com o tamanho integral das chaves, agilizando o processo de execução do algoritmo.

Atualmente as implementações em *hardware* são feitas, em sua grande maioria, de duas formas: através de desenvolvimento de circuitos dedicados, em geral, nos chamados ASICs (*Application Specific Integrated Circuits*, vide [9]), ou através dos circuitos FPGA.

As implementações em ASICs, por sua vez, embora apresentem um desempenho superior às implementações em *software*, por trabalharem com mais bits, empregam sempre um único algoritmo de criptografia. Se for necessário fazer uma atualização de um mesmo algoritmo, por exemplo, para que ele possa trabalhar com uma chave maior, o que é necessário de tempos em tempos, ou mesmo se for desejado usar um outro algoritmo de criptografia, um novo ASIC precisa ser adquirido, pois não é possível alterar seu conteúdo. Neste sentido, as implementações em FPGAs são mais flexíveis, uma vez que é possível programá-las novamente com a atualização de um mesmo algoritmo ou para substituir um algoritmo por outro.

Segundo Christof Paar [14], "Dispositivos re-configuráveis, como as FPGAs, apresentam grandes oportunidades para a implementação de sistemas criptográficos. Em particular, a rápida substituição de algoritmos criptográficos em *hardware* é possível em FPGAs, enquanto é muito difícil nos tradicionais ASICs".

Grande parte dos algoritmos de criptografia assimétrica é empregada em sistemas de segurança principalmente através do uso de rotinas desenvolvidas em linguagens de alto nível, como a linguagem C. Contudo, os pesquisadores têm começado a voltar a sua

atenção às implementações realizadas por meio de circuitos integrados de aplicação específica, do tipo ASICs e às potencialidades das FPGAs, que estão sendo cada vez mais usadas para receber as implementações desses algoritmos (vide [14]).

Hoje, é possível encontrar, sobretudo através da Internet, diversos artigos descrevendo implementações relacionadas direta ou indiretamente com este trabalho de pesquisa. Entretanto, trabalhos envolvendo *hardware* para criptografia são tão recentes, que o primeiro *workshop* sobre o assunto, o *Cryptographic Hardware and Embedded Systems* (CHES) (vide [14]), foi realizado pela primeira vez em 1999, nos Estados Unidos. O CHES apresenta o estado da arte em sistemas criptográficos implementados em *hardware*.

Esse *workshop* tem como objetivo apresentar todos os aspectos envolvendo criptografia e segurança em sistemas embutidos (vide [9]) e em hardware em geral. Através dele, tanto a comunidade científica como a indústria pode mostrar os resultados de suas mais recentes pesquisas. Um interesse maior é dado para *software* de alta performance e para novas maneiras de realizar implementações eficientes em *hardware*, especialmente para sistemas criptográficos voltados para uso em *smart cards* (vide [8]), microprocessadores, DSPs (vide [11]), etc.

Essa tendência de pesquisa é comentada por Christof Paar em [14]: "Historicamente, as redes de computadores consistiram principalmente de computadores tradicionais, isto é, servidores ou *mainframes* junto com clientes do tipo PC. Entretanto, é largamente esperado que a próxima geração de dispositivos para redes também consistirá de vários tipos de aplicações embutidas. Tais aplicações podem incluir telefones sem fio, carros com acesso à Internet, dispositivos domésticos, sensores de infra-estrutura, dispositivos médicos, etc".

1.3 Exemplos de trabalhos relacionados.

A seguir são apresentados, de forma resumida, quatro trabalhos que têm alguma relação com este. Desses, o primeiro foi muito importante para este trabalho, pela proximidade e facilidade de interação. Trata-se do trabalho desenvolvido no DCA-FEEC-UNICAMP pelo grupo de pesquisa REGRA_C, dirigido pelo professor Dr. Marco Aurélio Amaral Henriques (vide [11]). O segundo trabalho apresenta um exemplo de aplicação de algoritmos de criptografia em *smart-cards* (vide [8]). Os dois últimos são exemplos

similares a este trabalho e foram publicados em anais do *workshop Cryptographic Hardware and Embedded Systems* (CHES) (vide [13] e [19]).

1.3.1 Sistema criptográfico baseado em curvas elípticas em DSP

Macedo e os membros do grupo REGRA_C, em 2001, apresentaram um trabalho (vide [11]), no qual é possível encontrar a descrição de uma implementação de sistema criptográfico baseado em curvas elípticas para ambientes computacionalmente restritos.

A arquitetura base para a implementação desse sistema criptográfico foi o processador digital de sinais (DSP (*Digital Signal Processor*)) da família TMS320C54x da Texas Instruments (vide [11]). Mais especificamente, o dispositivo DSP usado na implementação do código e na obtenção dos resultados foi o TMS320VC5402, que possui 32Kbytes de RAM, 8Kbytes de ROM e trabalha a uma freqüência de 100MHz. A escolha desse dispositivo foi motivada pelo aumento de aplicações que usam microprocessadores DSP. Essas aplicações variam desde equipamentos de áudio digital até radares militares sendo que os DSPs estão tendo um uso crescente em produtos de comunicação sem fio, sobretudo telefones celulares.

Aplicações voltadas para os DSPs da Texas podem ser desenvolvidas com o auxílio do Code Composer Studio (vide [11]) da Texas Instruments. Este ambiente suporta todo o ciclo de desenvolvimento de programas em sistema embutidos: codificação, depuração e análise em tempo real. Para a implementação em questão, a codificação foi feita em linguagem C.

O código foi dividido em camadas e foram implementadas as funções das camadas 1, 2 e 3, representadas na figura 1.1 (fonte: [11]).

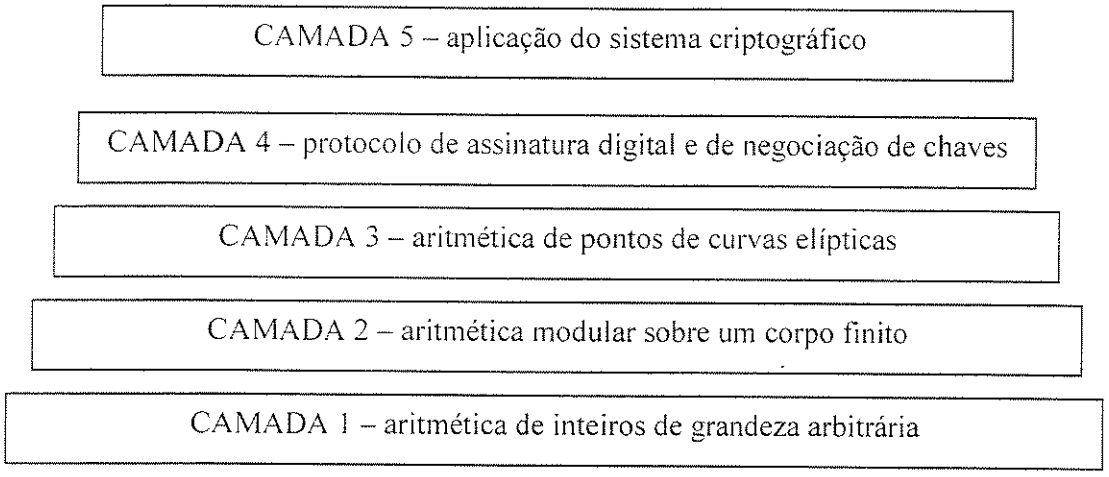

Figura 1.1: Implementação de software de criptografia em camadas.

1.3.2 Sistemas criptográficos baseados em curvas elípticas para *smart cards*

O *Certicom Research Group*, em 2000, apresentou um trabalho (vide [8]), no qual é possível encontrar um relato a respeito de sistemas criptográficos baseados em curvas elípticas para *smart cards*, os quais são dispositivos pequenos, portáteis e que possuem capacidade de armazenamento e processamento de dados. Eles são empregados em diversas áreas, como por exemplo, comércio eletrônico e identificação pessoal, dentre tantas outras aplicações. Muitos desses usos requerem serviços de criptografia, como por exemplo, assinaturas digitais e trocas de chaves. Um *smart card* é um ambiente propício para a implementação de sistemas criptográficos porque ele contém várias características de segurança, as quais melhoram a proteção das informações sigilosas, tanto durante o armazenamento, quanto durante o processamento das mesmas.

Ao mesmo tempo, algumas das características dos *smart cards* fazem a implementação de algoritmos de criptografia assimétrica sobre esses dispositivos tornar-se um desafio. A pouca memória disponível e a baixa capacidade de processamento são apenas dois exemplos de fatores limitantes para a implementação de sistemas criptográficos nesse ambiente.

Os *smart cards* oferecidos atualmente no mercado possuem uns poucos Kbytes de memórias RAM, ROM e EEPROM, além de uma CPU de 8 bits trabalhando a uma freqüência relativamente baixa (pouco superior a 3 MHz). Qualquer incremento de memória e CPU provoca um aumento significativo nos preços dos *smart cards*. O problema é que, para serem largamente utilizados, eles precisam ser baratos. Sistemas criptográficos baseados em curvas elípticas, por ocuparem menos memória que os outros sistemas, são bons candidatos para serem implementados nesses ambientes restritos.

1.3.3 Co-processador criptográfico baseado em curvas elípticas sobre $F(2^m)$

Torii e outros pesquisadores, em 2000, apresentaram um trabalho (vide [19]), no qual é possível encontrar a descrição da implementação LSI de um co-processador criptográfico baseado em curvas elípticas sobre Galois Field $GF(2^m)$ em uma FPGA e em um ASIC, assim como os resultados de suas implementações e testes. Ele realiza a multiplicação escalar para curvas pseudo-randômicas (vide [12]) e curvas de Koblitz (vide [12]), independentemente do tamanho do corpo finito.

A tecnologia usada como base para a primeira implementação foi uma FPGA da Altera, a EPF10K250AGC599-2 (vide [5]). O co-processador opera a 3 MHz e era composto por um multiplicador de 82 bits x 4 bits. Para uma multiplicação escalar envolvendo chaves de 163 bits, ele obtém o resultado em 80ms para curvas pseudo-randômicas e em 45ms para uma curva de Koblitz.

A segunda implementação usou como arquitetura base um ASIC da FUJITSU [19] de 0.25μm. O co-processador operou a 66MHz, usando um multiplicador de 288 bits x 8 bits, implementado sobre 165.000 portas lógicas. Para uma multiplicação escalar envolvendo chaves de 163 bits, ele obteve o resultado em 1,1ms para curvas pseudo-randômicas e em 0,65ms para uma curva de Koblitz. Para chaves de 571 bits, os resultados foram obtidos em 22ms e 13ms respectivamente para as mesmas curvas.

O co-processador foi descrito através da HDL *(Hardware Description Language)* Verilog (vide [9]). O diagrama básico do co-processador é mostrado na figura 1.2 (fonte: [19]). Ele é composto de quatro blocos funcionais:

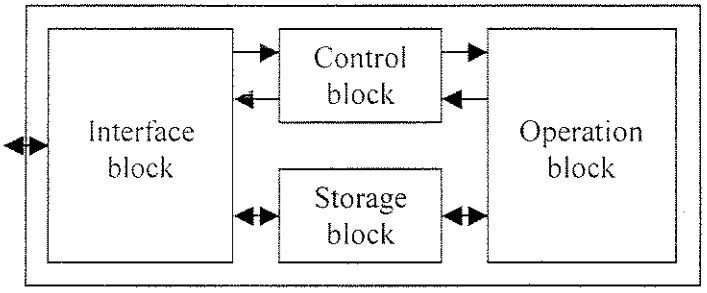

Figura 1.2: Diagrama básico do co-processador de ECC.

1 – *Interface block*, responsável pelo controle da comunicação entre o servidor e o co-processador;

2 – *Operation block*, que contém o multiplicador citado anteriormente;

3 - *Storage block*, o qual memoriza as entradas, as saídas e os valores intermediários;

4 – *Control block*, responsável por controlar o bloco de operação para que este realize a multiplicação escalar.

1.3.4 Processador de alta performance para sistemas criptográficos baseados em curvas elípticas sobre GF(2^m)

Orlando e outros pesquisadores, em 2000, apresentaram um trabalho (vide [13]), no qual é possível encontrar a descrição de um processador de alta performance para sistemas criptográficos baseados em curvas elípticas sobre GF(2^m), implementado em uma FPGA.

A tecnologia usada como base para a implementação do co-processador foi uma FPGA da Xilinx, a XCV400E-8-BG432 (Virtex E). O projeto foi codificado em VHDL (vide [15]) e sintetizado através das ferramentas FPGA Express 3.0 da Synopsis e Design Manager M2.1i da Xilinx (vide [13]).

A estrutura interna do processador chamado de ECP (*Elliptic Curve Processor*), é mostrada na figura 1.3 (fonte: [13]).

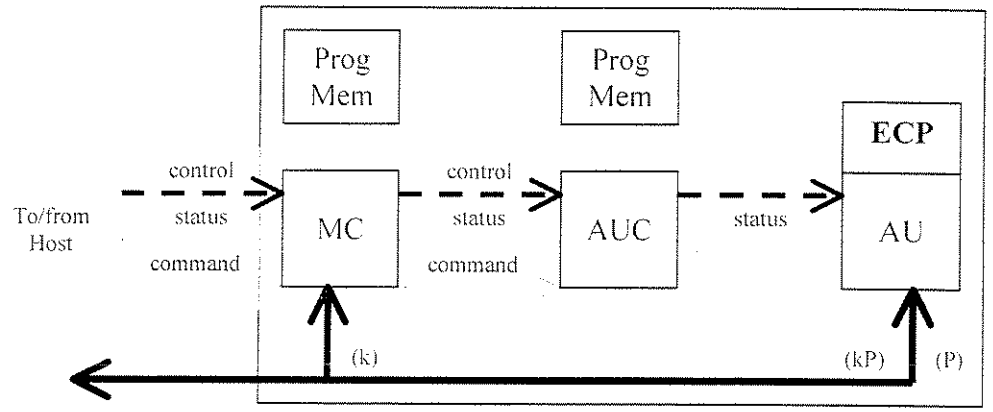

Figura 1.3: Arquitetura de um processador para curvas elípticas.

Ele possui três componentes principais: o MC (*Main Controler*), o AUC (*Arithmetic Unit Controler*) e o AU (*Arithmetic Unit*).

O MC controla a realização de operação kP, onde k é um inteiro e P um ponto na forma (Xp, Yp), pertencente a uma curva elíptica. Ele controla a execução do *Double_and_Add* e da multiplicação Montgomery (vide [11]). O MC também comanda a comunicação com o servidor do sistema. Como o próprio nome já diz, a AUC controla a AU, comandando as operações de adição de pontos, duplicação de ponto e conversão de coordenadas. A AU realiza adições, multiplicações, divisões e inversões para $F2^m$, sob controle da AUC.

A implementação usou: um MC de 16 bits com uma memória de 256 palavras de programa; uma AUC de 24 bits com uma memória de 512 palavras de programa; 128 registradores de 167 bits cada; uma interface de entrada e saída de 32 bits para comunicação com o servidor do sistema. O multiplicador foi implementado com os tamanhos 4, 8 e 16 para seus dígitos, não simultaneamente, para avaliar a escalabilidade do ECP.

O processador suporta mudança de algoritmos e curvas elípticas para atualização do sistema criptográfico, a fim de conservar a segurança desse mesmo sistema onde ele está

sendo usado. Trabalhando a uma freqüência de 76,7MHz e com um multiplicador de 16 dígitos, o processador consegue obter o resultado de uma multiplicação de pontos em 0,21ms.

1.4 Organização deste Trabalho

Seguem neste trabalho sete capítulos e cinco apêndices. Nos capítulos é apresentado o projeto desde os seus aspectos mais globais até os seus detalhes de construção. Os capítulos ainda discutem os resultados obtidos a partir da simulação do projeto e comparam estes resultados com aqueles descritos por outros trabalhos similares. Os apêndices apresentam informações importantes relacionadas a este trabalho. A seguir é feito um breve resumo de cada um dos capítulos e apêndices deste trabalho.

O Capítulo 2 apresenta algumas considerações preliminares do projeto, procurando analisar as questões de custo, limitações e aplicação do co-processador M1.

O Capítulo 3 descreve a estrutura interna do co-processador M1, revelando, em detalhes, como cada um dos seus componentes foi projetado e como eles são integrados de modo a garantir o funcionamento adequado do co-processador M1.

O Capítulo 4 descreve as características do dispositivo usado para a implementação do co-processador M1: uma FPGA, da família APEX20K, da Altera. Outras famílias de dispositivos FPGAs da Altera, que poderiam ser usadas na implementação do projeto, também são apresentadas nesse mesmo capítulo.

O Capítulo 5 revela como o algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas, foi implementado no co-processador M1 com intuito de demonstrar as operações que podem ser ali realizadas.

O Capítulo 6 apresenta os resultados obtidos por simulação, assim como comparações do co-processador M1 com outras implementações em *hardware*.

O Capítulo 7 apresenta as considerações finais a respeito do trabalho, incluindo comentários, conclusão e sugestões para trabalhos futuros.

Após o Capítulo 7 apresentam-se as referências bibliográficas.

O apêndice A faz uma breve introdução à segurança em redes de computadores, criptografias simétrica e assimétrica, autenticação, assinaturas digitais, curvas elípticas e o problema do logaritmo discreto sobre curvas elípticas.

O apêndice B apresenta uma breve introdução às características comuns às FPGAs existentes atualmente no mercado, incluindo comentários a respeito de seus elementos internos.

O apêndice C apresenta uma lista contendo todos os sinais de controle usados pelo co-processador M1, incluindo o número e a função de cada um deles.

O apêndice D permite conhecer as principais características da linguagem de descrição de *hardware*, VHDL, usada no projeto do co-processador M1.

O apêndice E permite conhecer as principais características da ferramenta de engenharia auxiliada por computador, o Quartus II, usada no projeto do co-processador M1.

Os apêndices A e B representam a base fundamental deste trabalho e, ao mesmo tempo, constituem um importante material complementar para o entendimento mais aprofundado dos assuntos relacionados com este trabalho.

Capítulo 2
O Co-processador Criptográfico M1

Um co-processador é um dispositivo que tem como finalidade auxiliar o processador principal a realizar funções complementares. O tipo mais conhecido é o co-processador numérico ou matemático que trata operações numéricas em ponto flutuante. Além deste, existem outros tipos de co-processadores com funções bastante específicas projetados para realizarem operações com maior precisão ou velocidade do que o processador principal. Pode-se citar como exemplos, co-processadores gráficos ou de gerência de entrada e saída.

Neste capítulo, é apresentado o projeto de um co-processador voltado ao tratamento de algoritmos criptográficos. A este co-processador foi dado o nome de M1.

2.1 Ponto de Partida do Trabalho

O desenvolvimento do co-processador M1 iniciou-se com a observação e a análise das características de muitos programas e algoritmos de criptografia assimétrica. Nesta análise, a necessidade de se trabalhar com uma grande quantidade de bits dentro dos algoritmos chamou a atenção. Isto obriga os programadores a desenvolver programas cujas funções manipulam enormes vetores numéricos. Essas funções apresentam encadeamentos de vários comandos IF, o que torna a execução do programa muito demorada. Uma das maneiras de se aumentar o desempenho desses algoritmos é construir um *hardware* composto por unidades capazes de trabalhar com longos vetores de bits. Em outras palavras, a existência de um dispositivo dotado de registradores de 512 bits, por exemplo, e uma unidade lógico-aritmética capaz de realizar de uma só vez operações aritméticas com o conteúdo desses registradores, eliminaria a necessidade do uso de grandes quantidades de laços dentro dos programas de criptografia assimétrica.

2.2 M1: Um Co-processador Voltado para Algoritmos de Criptografia Assimétrica

O dispositivo apresentado neste capítulo é um co-processador criptográfico. Pretende-se com a proposta deste dispositivo criar uma arquitetura básica e uma série de módulos

descritos em VHDL que poderão ser combinados para a implementação de algoritmos de criptografia assimétrica. Não se trata de um dispositivo dedicado a um único algoritmo criptográfico. O co-processador M1 representa uma espécie de "esqueleto" para a implementação de algoritmos de criptografia assimétrica em *hardware*, possuindo a estrutura básica para suportar a implementação desses algoritmos. É importante deixar claro que cada implementação do M1 trabalha com apenas um algoritmo. Assim sendo, o chamado "esqueleto" facilita muito o trabalho do projetista, durante o desenvolvimento do *hardware*.

O co-processador M1 apresenta uma estrutura tal que poderá ser facilmente adaptada a diferentes algoritmos de criptografia assimétrica, em função de todos eles possuírem características similares entre si. Uma dessas semelhanças é que esses algoritmos costumam ser baseados em funções matemáticas do tipo mão-única (*one way functions*) (vide [17]) - as quais funcionam como uma armadilha-, isto é, aplicadas em uma "direção", elas podem servir para criptografar dados, porém, é impossível usá-las na "direção contrária", para tentar decriptografar os mesmos dados. Essas diferentes funções podem até ser resolvidas matematicamente de formas distintas, mas, sendo todas elas funções matemáticas, podem usar uma mesma unidade lógico-aritmética para serem resolvidas, bastando, talvez, fazer algumas pequenas modificações no *hardware*. Por ter sido desenvolvido sobre uma FPGA, o co-processador M1 é bastante flexível para aceitar alterações.

2.3 Características do Projeto do Co-processador M1

O co-processador M1 foi desenvolvido parte em VHDL, parte em esquemático. A porção em VHDL compreende toda a micro-memória, incluindo o algoritmo de criptografia assimétrica. A porção em esquemático compreende todo o restante do co-processador M1, isto é, registradores, contadores, unidade lógico-aritmética, barramentos, etc.

A maior complexidade do nosso projeto está na rotina do algoritmo criptográfico, escrita em VHDL. Na verdade, a micro-memória por completa é bastante complexa e está sujeita a possíveis alterações (caso uma substituição de algoritmos seja desejada). Dessa forma ela foi desenvolvida inteiramente em VHDL, para facilitar o trabalho do projetista. Além de o projetista ter que entender a função matemática apresentada pelo algoritmo, e

como ela será empregada na criptografia, ele ainda tem que se preocupar em como transformar tudo isso em *hardware*. Descrever a rotina de criptografia em VHDL é seguramente mais simples do que projetar através de esquemático. Por isso, esse componente do co-processador M1 foi escrito nessa linguagem.

O restante do co-processador M1, isto é, registradores, contadores, unidade lógico-aritmética, barramentos, foi desenvolvido em esquemático, porque é provável que esta porção do hardware não sofra muitas alterações, quando se desejar substituir um algoritmo de criptografia por outro. Esses componentes devem ser mantidos dessa forma, porque o projeto em esquemático produz melhores resultados nas FPGAs em termos de área e desempenho, quando comparados com projetos em VHDL.

2.4 Custos do Projeto em Relação a ASICs

Implementações em ASICs costumam apresentar baixos custos porque, neste caso, um projeto é feito uma única vez e é gravado em série – linha de produção-, nos chips. Porém, como o co-processador M1 deve poder trabalhar com vários algoritmos de criptografia assimétrica, seria necessário um ASIC para cada tipo de algoritmo, para se ter a mesma funcionalidade. É possível pensar ainda que cada um desses algoritmos, dependendo da maneira como o co-processador M1 foi projetado, teria diferentes padrões de tamanhos de chaves, cada um implementado em um *chip* distinto. Dessa forma, seriam necessários muitos *chips* para implementar o projeto como um todo, elevando seu custo.

No caso deste trabalho, apenas um único chip é suficiente para dar suporte aos algoritmos de criptografia assimétrica. Os algoritmos serão desenvolvidos uma única vez e poderão ser implementados sobre a FPGA apropriada. Quando houver necessidade de trocar um algoritmo, por exemplo, por outro mais seguro, basta apenas adquirir o novo algoritmo e re-programar a FPGA. Não é preciso comprar um novo chip toda vez que se desejar efetuar a troca de um algoritmo. Isso facilita muito o trabalho do projetista, durante o desenvolvimento do *hardware*.

O co-processador M1 pode ser mudado de 128 para 160, 512 bits ou qualquer outra quantidade de bits facilmente, pois basta alterar os valores de algumas constantes do projeto. Para trabalhar com essas grandes porções de bits, é necessário empregar os

dispositivos EP20K600E, EP20K1000E e EP20K1500E. Para fazer a compilação desses dispositivos foi usado um processador Intel Pentium 4, de 1,5GHz de freqüência, com 1024Mbytes de memória RAM e 30Gbytes de HD. A tabela 2.1 (fonte: [1]) mostra o tamanho mínimo exigido para a memória RAM e para o *swap* no HD necessários para simular projetos de *hardware* sobre dispositivos APEX da Altera.

Tabela 2-1 Tamanho mínimo de RAM e de "swap" para simular projetos sobre o APEX.

Dispositivos APEX	Tamanho mínimo exigido para a memória RAM	Tamanho mínimo exigido para Swap no HD
EP20K60E, EP20K100/100E, EP20K160E, EP20K200/200E	256 MB	256 MB
EP20K300E, EP20K400/400E, EP20K600E	512 MB	512 MB
EP20K1000E	1024 MB	1024 MB
EP20K1500E	1024 MB	1024 MB

2.5 Aplicação do Co-processador M1

Apresenta-se aqui o exemplo de uso do M1 como um co-processador para um microcomputador compatível com IBM-PC com um barramento PCI. O M1 está instalado numa placa adaptadora com uma unidade de memória de dupla porta, compartilhada pelo barramento PCI e pelo M1. Considere ainda que o M1 deste exemplo está configurado para operar inteiros de 192 bits e para isto utiliza 20 registradores de 192 bits. O barramento PCI em sua versão 2.1 permite a transferência de palavras de 64 bits a uma velocidade de 66MHz (vide[21]).

A figura 2.1 ilustra o exemplo acima. Nela destaca-se a memória de dupla porta MDP que pode ser acessada pelo barramento PCI e pelo M1. O barramento PCI enxerga a MDP como composta por 60 palavras de 64 bits (480 *bytes*). O M1 enxerga a mesma MDP como organizada em 20 palavras de 192 bits.

O processamento criptográfico começa com o processador principal do barramento PCI escrevendo na MDP dados em unidades de 3 palavras de 64 bits, usando para isto 3 ciclos de acesso. O M1 acessa cada uma destas unidades como uma única palavra de 192 bits através de um único ciclo. Após tratar os dados, o M1 coloca os resultados de suas operações de volta em MDP, para que sejam acessados pelo barramento.

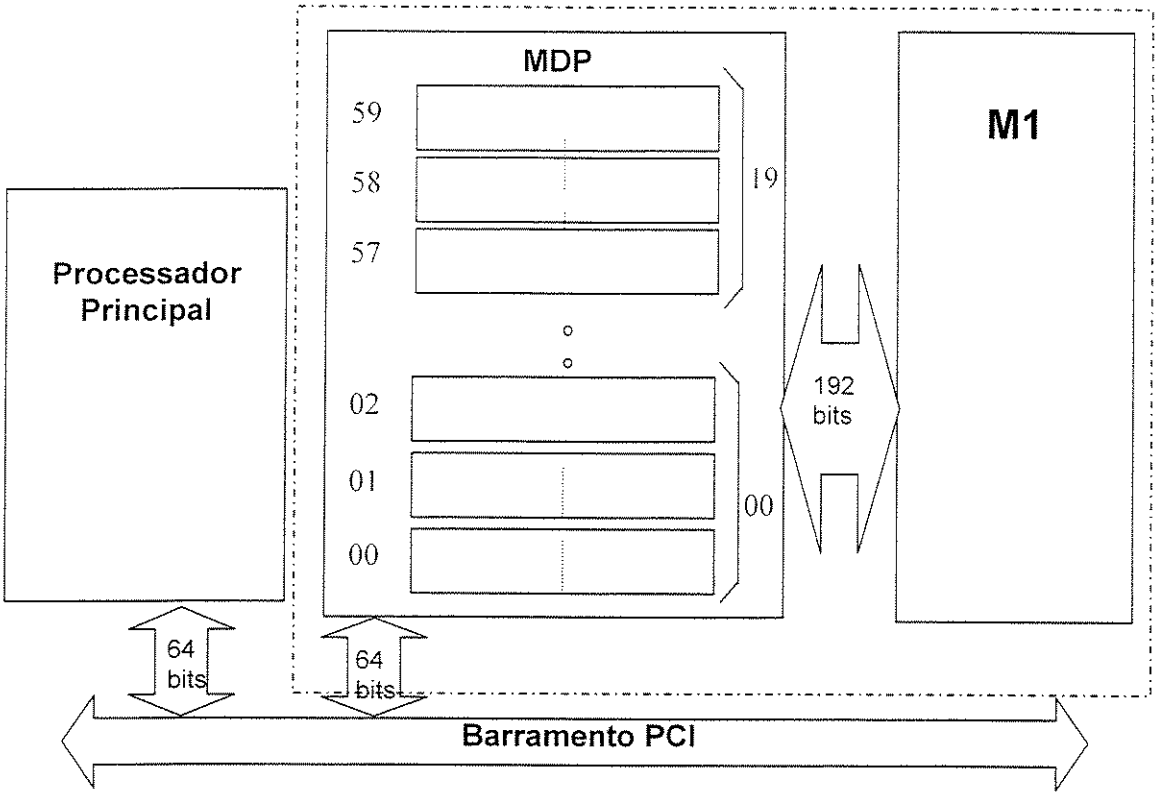

Figura 2.1: Exemplo de esquema da comunicação entre o M1 e uma CPU através de uma memória compartilhada.

2.6 Comentários

O desenvolvimento do co-processador M1 é resumido nesse capítulo. Aqui foram considerados aspectos gerais de desenvolvimento e esboçado um possível circuito de aplicação num barramento PCI.

Nos demais capítulos a seguir, é possível aprofundar na descrição deste projeto.

Capítulo 3
A Estrutura Interna do M1

Este capítulo descreve a estrutura interna do co-processador M1, revelando, em detalhes, como cada um dos seus componentes foi projetado e como eles são integrados de modo a garantir o funcionamento adequado do co-processador M1. Para agilizar e facilitar a descrição, este capítulo foi ilustrado com o exemplo de uma implementação do co-processador M1 com somente 8 bits.

3.1 Visão Geral da Estrutura Interna do Co-processador M1

Conforme pode ser visto na figura 3.1, o co-processador M1 é composto, basicamente, por uma unidade de controle, uma unidade lógico-aritmética, vários registradores auxiliares de uso geral, um registrador acumulador (ACC) e um registrador temporário (TEMP), além de algumas outras unidades menores, como contadores e PSW.

O número de elementos que compõem o co-processador M1 pode variar de implementação para implementação, dependendo das exigências impostas pelo algoritmo de criptografia que estiver sendo usado.

A implementação feita para este trabalho exigiu que o co-processador M1 fosse dotado de uma unidade de controle, uma unidade lógico-aritmética, 18 registradores, sendo 16 deles, registradores de uso geral, um registrador acumulador e um registrador temporário. Também foram necessários dois contadores e uma PSW.

Embora praticamente todos os elementos do co-processador M1 sejam aptos a manipular longos vetores de bits de uma só vez, o co-processador M1 descrito neste capítulo trabalha com apenas 8 bits, apenas por uma questão de facilitar a representação das figuras mostradas mais à diante. Em outras palavras, todos os registradores do co-processador M1, assim como a unidade lógico-aritmética, serão mostrados com tamanho igual a 8 bits.

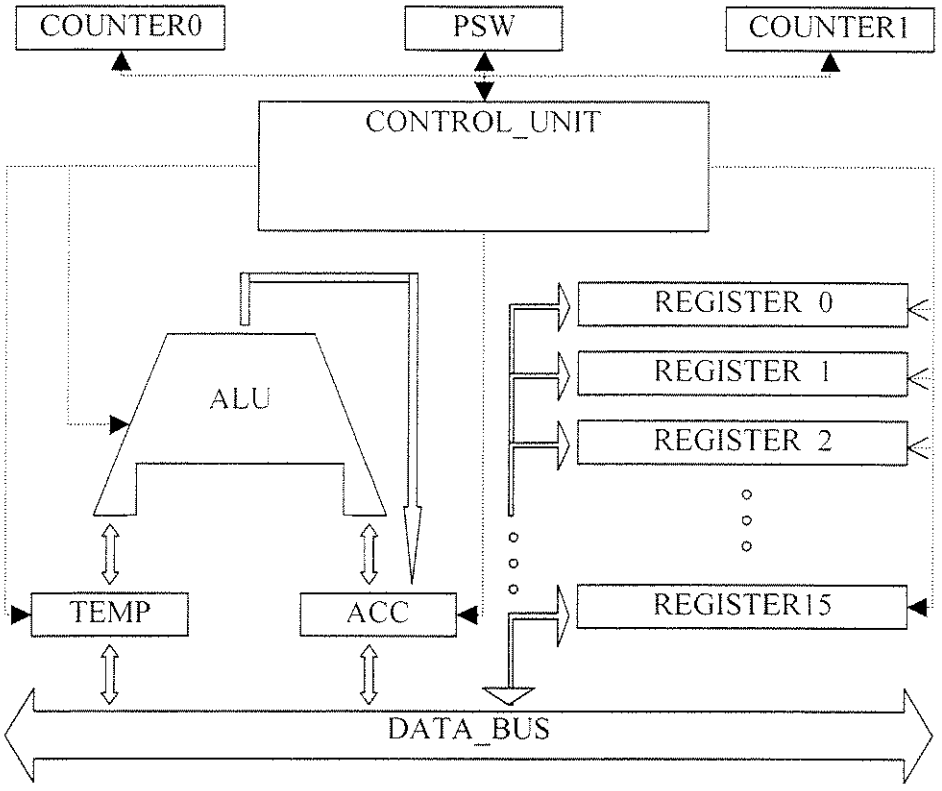

Figura 3.1: Visão geral da estrutura interna do co-processador M1.

Na verdade, outras implementações do M1 podem vir a apresentar elementos com tamanhos variando de 2 bits até n bits. O valor de "n" é limitado pela capacidade da FPGA que é usada para implementar o circuito do M1.

Um buffer (a MDP), que não faz parte da estrutura interna do co-processador M1 e por isso não aparece na figura 3.1, alimenta o barramento de dados (DATA_BUS) com as informações que serão processadas pelo M1. Todos os registradores têm acesso ao DATA_BUS. A unidade lógico-aritmética também tem acesso ao DATA_BUS, mas de forma indireta, por intermédio do ACC e do TEMP. Sendo assim, este barramento representa um dos principais meios de comunicação entre os elementos que compõem o M1. Por ele, circulam todas as informações que são processadas pelo co-processador M1 durante a execução do algoritmo de criptografia.

Outro meio de comunicação importante, presente dentro do M1, é o barramento de controle, o qual permite que a unidade de controle mantenha comunicação com os demais

elementos do M1. As linhas tracejadas da figura 3.1 representam sinais de controle que chegam e saem da unidade de controle. Eles são responsáveis por comandar todos os elementos presentes no M1. Esse trabalho integrado entre os elementos é que permite que o M1 execute suas tarefas com sucesso. Esses elementos foram desenvolvidos através de esquemático e serão apresentados em detalhes nas próximas seções deste mesmo capítulo.

Além desses elementos, o M1 dispõe de uma biblioteca de módulos de circuitos que implementam rotinas básicas úteis a vários algoritmos de criptografia assimétrica, como por exemplo, o módulo capaz de realizar multiplicação de números da ordem de algumas centenas de bits. Outros módulos, que também podem manipular longos vetores de bits de uma só vez, são capazes de realizar operações como divisão, multiplicação, entre outras.

3.2 Unidade de Controle

A unidade de controle é responsável pelo controle de todas as atividades do co-processador M1. Ela comanda os passos necessários para a busca de dados no buffer, assim como a execução do algoritmo e a colocação do resultado final de volta no buffer.

A figura 3.2 mostra a representação do circuito da unidade de controle. Todas as entradas, com exceção de INSTRUCTION e NEXT_MICROINSTRUCTION, representam sinais provenientes da PSW. Esses sinais impõem condições à execução do algoritmo de criptografia, determinando a seqüência do fluxo de execução do mesmo. Essa seqüência aparece em INSTRUCTION e em NEXT_MICROINSTRUCTION, na forma de endereços de 16 bits. Os sinais de controle gerados pela unidade de controle são colocados na saída MICROINSTRUCTION. Esta unidade gera até duzentos sinais de controle diferentes. O apêndice C mostra a lista contendo todos os sinais de controle usados nesta implementação.

Na verdade, a unidade de controle é descrita em VHDL, como pode ser visto na figura 3.3. Através dela, é possível notar as declarações de todas as entradas comentadas anteriormente, assim como a saída MICROINSTRUCTION, para os duzentos sinais de controle. As demais linhas representam uma pequena parte do algoritmo (vide capítulo 5), o qual emprega os módulos de circuitos da biblioteca, comentados na seção anterior.

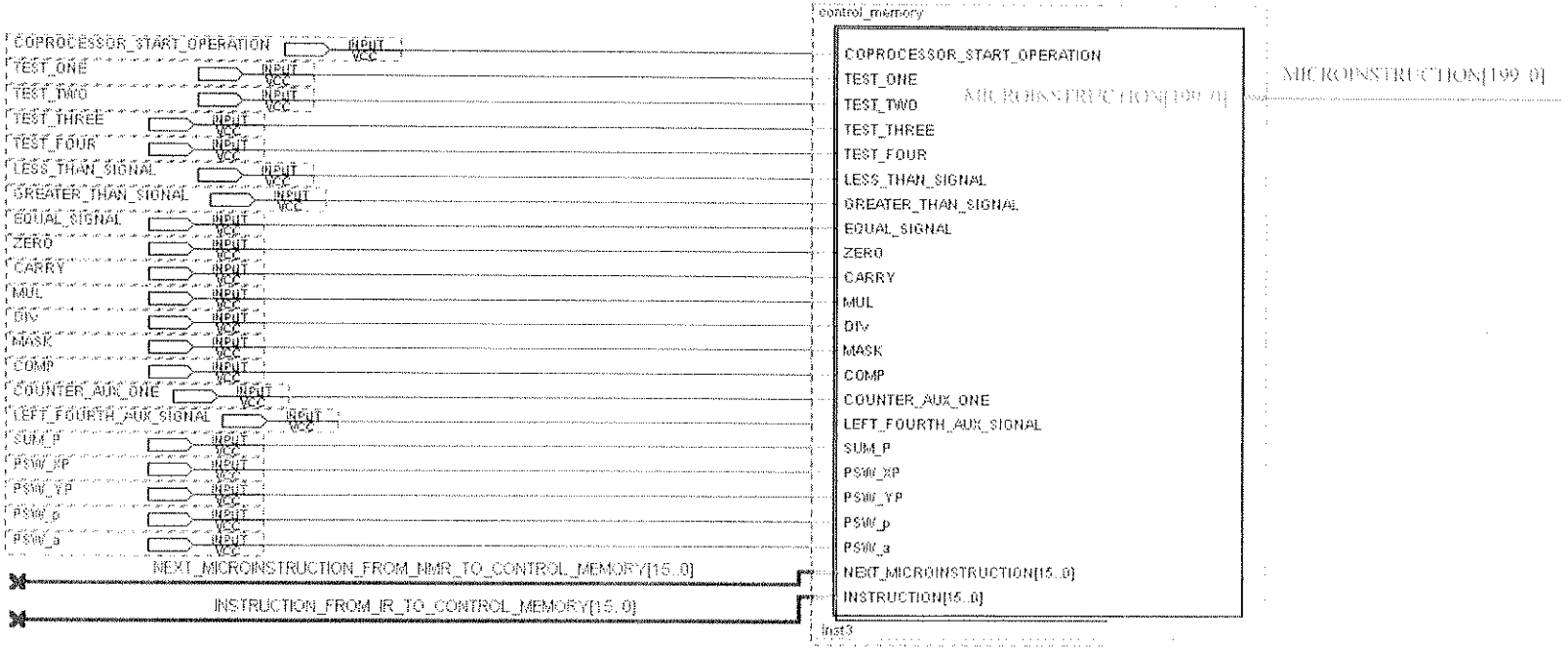

Figura 3.2: Visão geral do circuito da unidade de controle.

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

entity CONTROL_MEMORY is
    port(   COPROCESSOR_START_OPERATION, TEST_ONE    : in  std_logic;
            TEST_TWO, TEST_THREE, TEST_FOUR          : in  std_logic;
            LESS_THAN_SIGNAL, GREATER_THAN_SIGNAL    : in  std_logic;
            EQUAL_SIGNAL, ZERO, CARRY, MUL, DIV      : in  std_logic;
            MASK, COMP, COUNTER_AUX_ONE              : in  std_logic;
            LEFT_FOURTH_AUX_SIGNAL, SUM_P            : in  std_logic;
            PSW_XP, PSW_YP, PSW_p, PSW_a             : in  std_logic;
            NEXT_MICROINSTRUCTION, INSTRUCTION       : in  std_logic_vector(  15 downto 0 );
            MICROINSTRUCTION                         : out std_logic_vector( 199 downto 0 )
        );
end CONTROL_MEMORY;

architecture CONTROL_MEMORYBehave of CONTROL_MEMORY is
begin

-------------------------------
    MICROINSTRUCTION <=
-------------------------------

  11111111111111111111111111111111111111111111 000000
  99999999999999999999999777777777766666666666 000000
  98765432109876543210987654321098765543210 543210


OPERATION ( Q = RP )

 "10000000000000010000000000000000000000000 000000" when (NEXT_MICROINSTRUCTION = X"0000")                                                                  else
 "00000000000000010000000000000000000000001 000000" when (NEXT_MICROINSTRUCTION = X"0001")                                                                  else
 "00000000000000000000000000000000000000000 000000" when (NEXT_MICROINSTRUCTION = X"0002") and ( COPROCESSOR_START_OPERATION = '0' ) else
```

Figura 3.3: Descrição da unidade de controle em VHDL.

3.3 PSW (Program Status Word)

O circuito PSW (*Program Status Word*) é composto por vários registradores BSR (Bit Status Register) de 1bit, independentes uns dos outros. Dentro da PSW, cada BSR armazena o estado de um determinado elemento do co-processador M1 ou registra a condição de uma dada operação, durante a execução do algoritmo de criptografia. Por exemplo, a PSW possui um BSR para registrar se a contagem realizada pelo contador COUNTER0 foi reinicializada ou não; o mesmo acontece para o contador COUNTER1.

O circuito apresentado pela figura 3.4 é o BSR, responsável pelo armazenamento de um único bit. Ele é composto, quase que exclusivamente, por um flip-flop JK. O sinal de entrada CONTROL_SIGNAL_TO_BSR é responsável por permitir o armazenamento de um novo valor no registrador. CLEAR_BSR é um sinal de entrada que muda o estado do registrador para zero. O sinal de saída SIGNAL_FROM_BSR_TO_CU tem como função informar a unidade de controle a respeito do valor armazenado pelo registrador BSR.

A realimentação da entrada com o sinal da saída serve para que o registrador BSR conserve indefinidamente o seu valor, mesmo que haja alterações de valores no sinal de entrada. Em outras palavras, se o registrador estiver armazenando o valor 0 e receber um sinal com valor 1 em sua entrada, ele passará a registrar o valor 1. O valor armazenado no registrador não vai mudar, independentemente da variação de valores do sinal de entrada, graças a essa realimentação. O registrador somente voltará a armazenar o valor 0, se receber um sinal de *clear*.

A PSW possui outros BSRs importantes, a saber: um BSR para registrar o *carry-out* de uma operação de adição; um BSR para registrar se o resultado de uma comparação resultou em igualdade; um BSR para registrar se o resultado de uma comparação resultou em um número maior que o outro; um BSR para registrar se o resultado de uma comparação resultou em um número menor que o outro.

Em suma, a PSW registra o estado da execução do algoritmo de criptografia, para um dado momento.

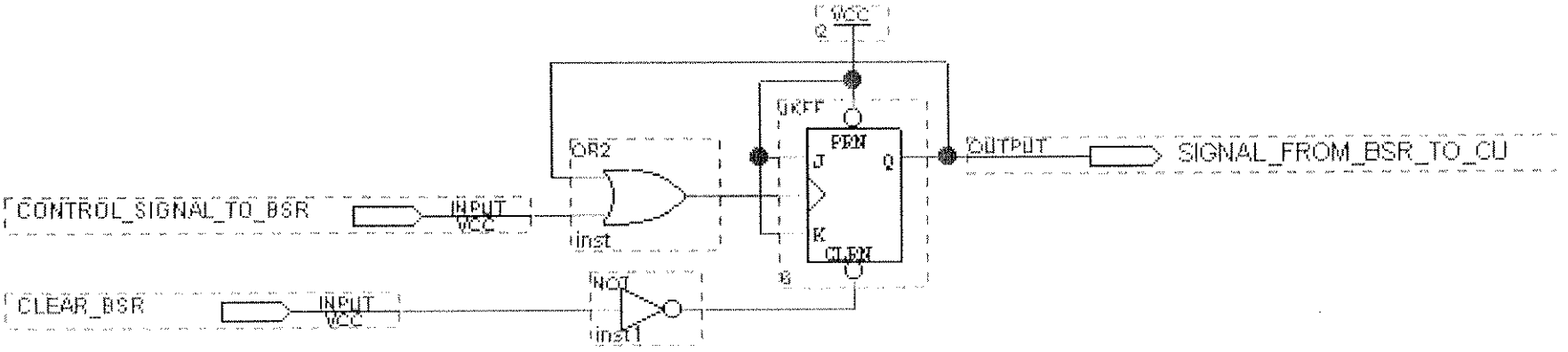

Figura 3.4: Estrutura interna da PSW (registrador BSR).

3.4 Registradores

Cada um dos dezesseis registradores auxiliares de uso geral do co-processador M1 possui 8 bits de tamanho. A maioria funciona como registrador auxiliar, usado pelo co-processador M1 para guardar dados. Todos eles podem ser usados para realizar deslocamentos à direita ou à esquerda.

A estrutura interna de um desses registradores é apresentada em detalhes pela figura 3.5. Através dela é possível notar que o registrador é dotado de oito flip-flops JK e possui diversas entradas para sinais de controle e para dados.

Para que um novo valor possa ser armazenado no registrador, basta alimentar DATA_INPUT-OUTPUT_SHIFT_REGISTER[7..0], através do DATA_BUS, com o valor desejado. Quando DATA_INPUT_IN_SHIFT_REGISTER_CONTROL mudar seu valor de 0 para 1, o registrador armazenará o novo valor.

Se for desejado saber o valor presente no registrador, em um dado instante de tempo, basta mudar o valor de DATA_INPUT-OUTPUT_SHIFT_REGISTER_CONTROL de 0 para 1 e o conteúdo do registrador será apresentado no DATA_BUS, por meio do DATA_INPUT-OUTPUT_SHIFT_REGISTER[7..0].

As entradas DATA_STORED_IN_SHIFT-REGISTER_LEFT_CONTROL e DATA_STORED_IN_SHIFT-REGISTER_RIGHT_CONTROL recebem os sinais de controle, respectivamente, para as operações de deslocamento à esquerda e de deslocamento à direita. Quando ocorre um deslocamento à esquerda, o bit 8 (bit excedente) do registrador é colocado na saída DATA_STORED_IN_SHIFT-REGISTER_LEFT e o novo bit 0 é introduzido no registrador através da entrada DATA_STORED_IN_SHIFT-REGISTER0. Quando ocorre um deslocamento à direita, o bit excedente do registrador é colocado na saída DATA_STORED_IN_SHIFT-REGISTER_RIGHT e o novo bit é introduzido no registrador através da entrada DATA_STORED_IN_SHIFT-REGISTER9.

As operações de deslocamento à esquerda e de deslocamento à direita são importantes, dentre outras coisas, para a realização das operações aritméticas de multiplicação e de divisão.

O sinal CLEAR_SHIFT_REGISTER obriga o registrador a armazenar o valor 0.

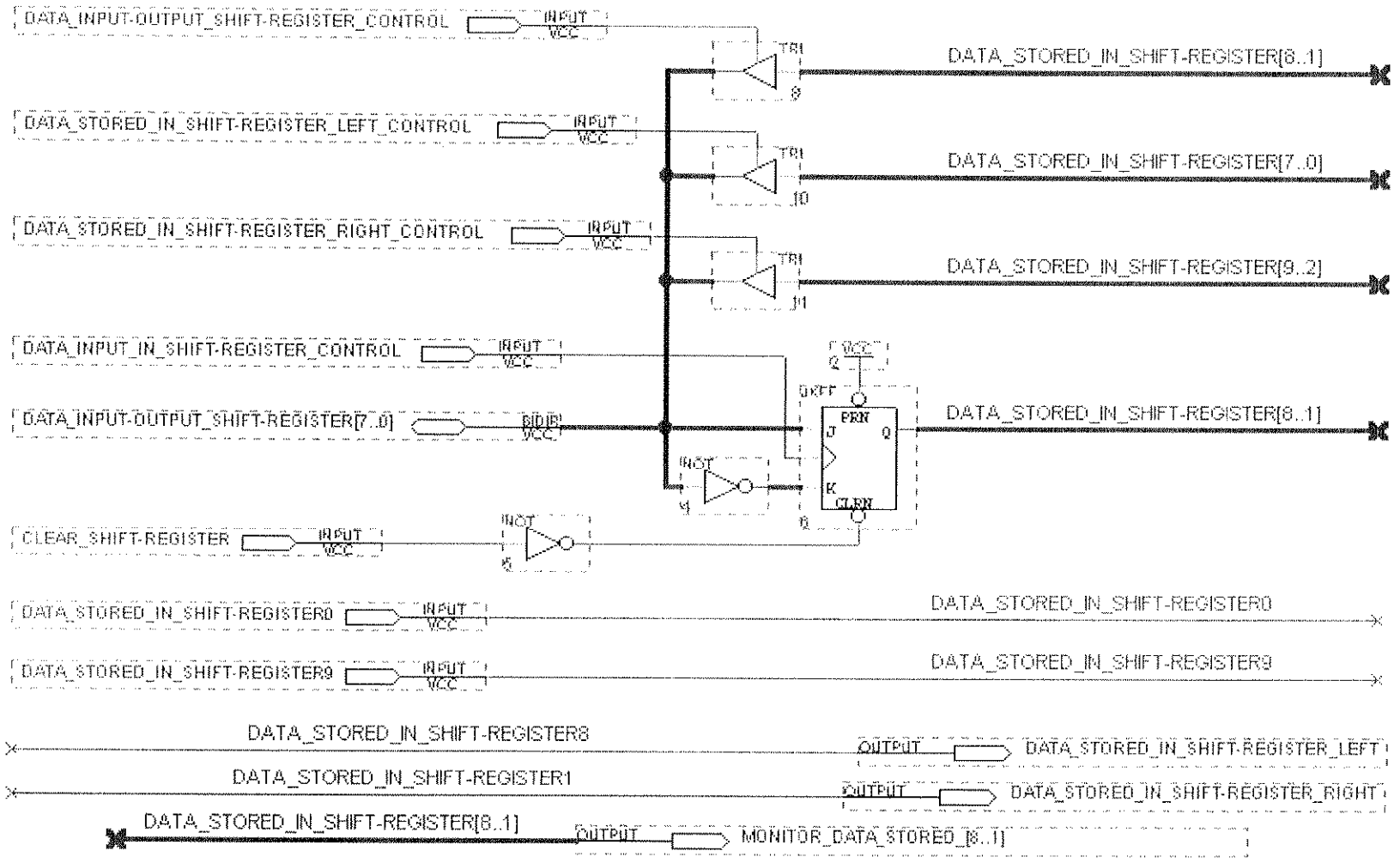

Figura 3.5: Estrutura interna de um registrador.

3.5 Unidade Lógico-Aritmética

A unidade lógico-aritmética (ALU) é capaz de realizar operações aritméticas (soma, subtração, multiplicação e divisão) e operações lógicas (AND, OR, NOT e XOR), além de comparações (maior que, menor que e igual a) tanto para números inteiros como para polinômios.

Enquanto unidades lógico-aritméticas tradicionalmente realizam operações em dados de 16, 32 ou 64 bits, a unidade lógico-aritmética do co-processador M1 é capaz de realizar qualquer uma das operações anteriores em dados com até algumas centenas de bits, de uma só vez. Essa característica elimina a necessidade de se fazer programas nos quais grandes inteiros são tratados como vetores de inteiros menores, operados separadamente uns dos outros com o auxílio de vários laços aninhados.

A figura 3.6a revela que as entradas DATA_FROM_ACCUMULATOR_TO_ALU e DATA_FROM_TEMPORARY_TO_ALU recebem as informações que serão processadas pela ALU, respectivamente, do ACC e do TEMP. O resultado das operações realizadas pela ALU é colocado na saída DATA_FROM_ALU_TO_ACCUMULATOR, para ser posteriormente armazenado pelo ACC. A mesma figura 3.6a mostra o conjunto de portas lógicas responsáveis por realizarem as operações lógicas AND, OR, NOT e XOR. Essas operações são ativadas, respectivamente, pelos sinais de controle AND_OPERATION_CONTROL, OR_OPERATION_CONTROL e XOR_OPERATION_CONTROL, para as duas últimas operações. Os circuitos necessários para as operações de soma e subtração foram omitidos por não serem necessários para este projeto, por isso não aparecem na figura 3.6a.

Um outro circuito que também não aparece, mas poderia estar presente internamente na unidade lógico-aritmética, em implementações de outros algoritmos, e, por isso, merece ser comentado, é o circuito *carry look ahead* 512 bits. Tal circuito aumenta em muito a velocidade das operações aritméticas, uma vez que o carry não precisa se propagar através de 512 somadores de 1bit. Na verdade o *carry* se propagará por apenas 16 unidades de *carry look ahead* de 32 bits cada uma, possibilitando maior velocidade nas operações aritméticas. O último *carry* é propagado através da saída carry_output.

A unidade lógico-aritmética possui um circuito chamado compare, mostrado na figura 3.6b. Esse circuito é responsável por realizar a operação de comparação entre números inteiros. A figura 3.7 mostra a descrição desse circuito em VHDL. A figura 3.7 mostra que suas entradas DATA_FROM_ACCUMULATOR_TO_ALU e DATA_FROM_TEMPORARY_TO_ALU são alimentadas, respectivamente, com o valor do inteiro armazenado em TEMP e com o valor do inteiro armazenado em ACC. Se o resultado de uma comparação resulta em igualdade, a saída EQUAL_SIGNAL assume o valor 1; se o valor do inteiro vindo do ACC é maior que o valor do inteiro vindo do TEMP, a saída GREATER_THAN_SIGNAL assume o valor 1; se o valor do inteiro vindo do ACC é menor que o valor do inteiro vindo do TEMP, a saída LESS_THAN_SIGNAL assume o valor 1.

A mesma unidade lógico-aritmética possui um circuito chamado polydegree, mostrado na figura 3.6b. Esse circuito é responsável por realizar a operação de comparação entre polinômios. A figura 3.8 mostra a estrutura interna desse circuito em detalhes. Ele realiza sua função sempre que o sinal START assume um valor igual a 1. O circuito polydegree é composto por três outros circuitos: dois circuitos chamados degree e um circuito chamado polycomp.

O degree é responsável por calcular o grau de um polinômio. Dentro de polydegree, existem dois circuitos degree porque um deles serve para calcular o grau do polinômio armazenado em ACC, enquanto o outro serve para calcular o grau do polinômio armazenado em TEMP. Esses dois polinômios entram em polydegree, respectivamente, através das entradas DATA_FROM_ACC e DATA_FROM_TEMP, cada uma com 8 bits.

O polycomp é responsável por comparar os graus de dois polinômios. A figura 3.8 mostra que ele compara os graus gerados na saída dos dois circuitos degree, comentados anteriormente. Em outras palavras, suas entradas poly1 e poly2 são alimentadas, respectivamente, com o grau do polinômio armazenado em TEMP e com o grau do polinômio armazenado em ACC. Se o resultado de uma comparação resulta em igualdade, a saída EQUAL assume o valor 1; se o grau do polinômio vindo do ACC é maior que o grau do polinômio vindo do TEMP, a saída GREATER_THAN assume o valor 1; se o grau do polinômio vindo do ACC é menor que o grau do polinômio vindo do TEMP, a saída LESS_THAN assume o valor 1.

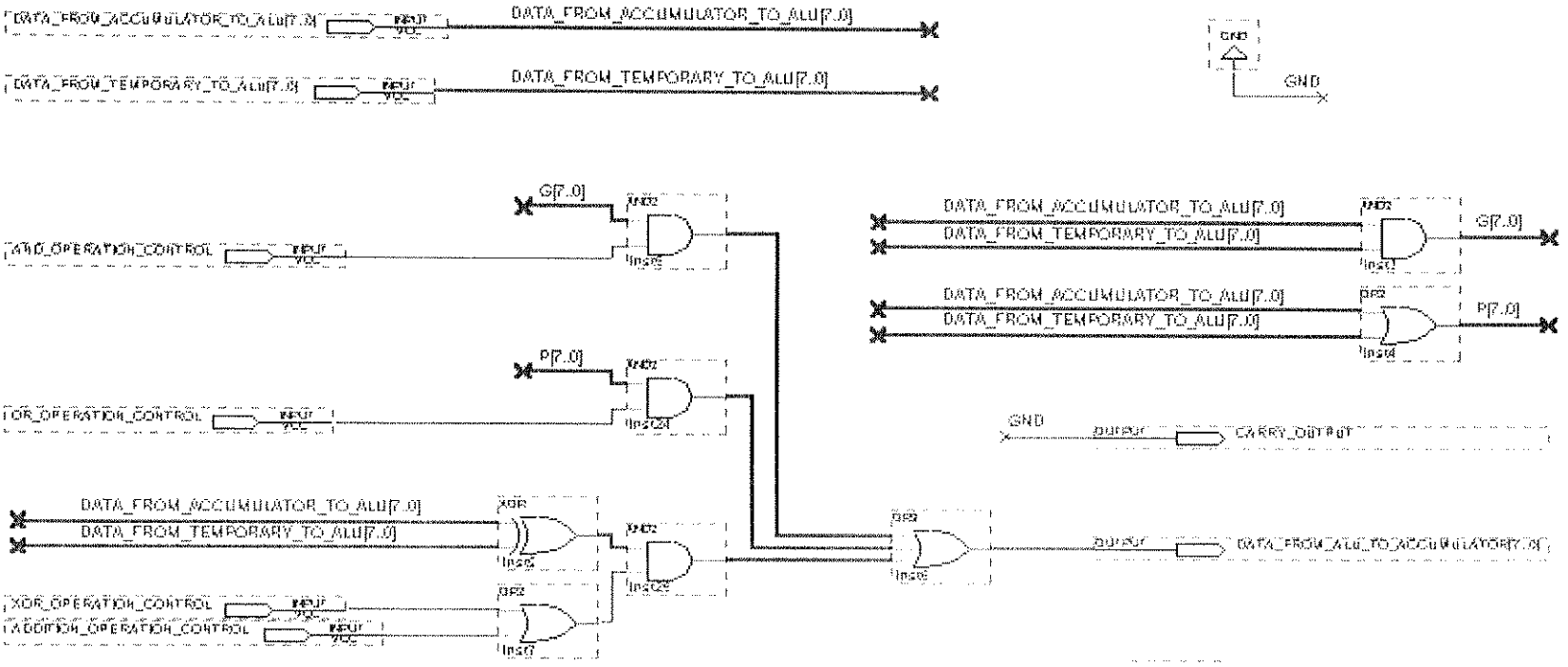

Figura 3.6a: Estrutura interna da ALU.

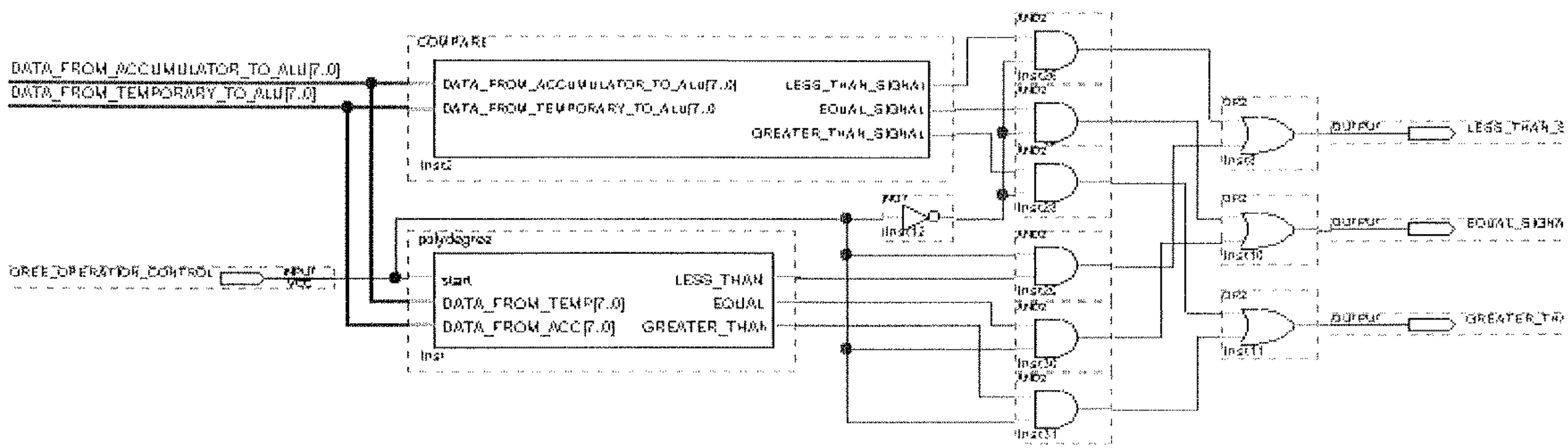

Figura 3.6b: Estrutura interna da ALU.

```
ENTITY compare IS

    PORT(DATA_FROM_ACCUMULATOR_TO_ALU, DATA_FROM_TEMPORARY_TO_ALU : IN  bit_vector(7 DOWNTO 0);
         LESS_THAN_SIGNAL, EQUAL_SIGNAL, GREATER_THAN_SIGNAL : OUT bit
        );

END compare;

ARCHITECTURE compare_bhv OF compare IS
BEGIN

LESS_THAN_SIGNAL <= '1' WHEN DATA_FROM_ACCUMULATOR_TO_ALU<DATA_FROM_TEMPORARY_TO_ALU ELSE '0';
EQUAL_SIGNAL     <= '1' WHEN DATA_FROM_ACCUMULATOR_TO_ALU=DATA_FROM_TEMPORARY_TO_ALU ELSE '0';
GREATER_THAN_SIGNAL<='1' WHEN DATA_FROM_ACCUMULATOR_TO_ALU>DATA_FROM_TEMPORARY_TO_ALU ELSE '0';

END compare_bhv;
```

Figura 3.7: Descrição do circuito compare em VHDL.

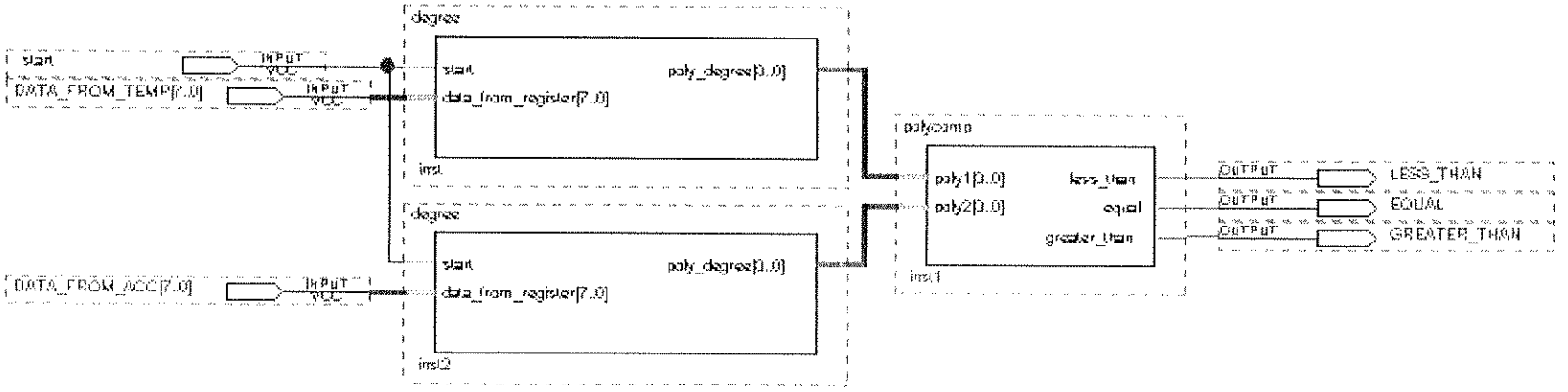

Figura 3.8: Estrutura interna do polydegree.

3.6 Contadores

Contadores são circuitos que possibilitam ao co-processador M1 saber o número de vezes que uma certa operação está sendo realizada, durante a execução do algoritmo de criptografia.

A figura 3.9 mostra a estrutura interna de um contador. O contador da figura 3.9 possui três flip-flops JK e uma série de portas lógicas que estão relacionadas às operações de controle do contador.

O contador pode ser programado com um valor inicial. Para tanto, basta alimentar INITIAL_VALUE_TO_COUNTER_FROM_BUS[2..0], através do DATA_BUS, com o valor desejado. Quando INITIAL_VALUE_TO_COUNTER_FROM_BUS_CONTROL mudar seu valor de 0 para 1, o contador será programado com o novo valor.

Se for desejado saber o valor presente no contador, em um determinado instante de tempo, basta mudar o valor de DATA_STORED_IN_COUNTER_OUTPUT_CONTROL de 0 para 1 e o conteúdo do contador será apresentado no DATA_BUS, por meio do INITIAL_VALUE_TO_COUNTER_FROM_BUS[2..0]. Para que o contador inicie sua contagem em zero, basta submetê-lo a um sinal *clear*, através de CLEAR_COUNTER_CONTROL. A direção da contagem, crescente ou decrescente, é determinada pelo sinal COUNTER_INCREMENTE-DECREMENTE_CONTROL. A entrada RIPPLE_INPUT0 informa ao contador o valor (0 ou 1) de seu bit menos significativo.RIPPLE[2..0] e RIPPLE[3..1] estão diretamente relacionados à capacidade de contagem do contador.

A saída ZERO tem seu valor alterado para 1, toda vez que o contador reinicia sua contagem em zero. Quando isso acontece, esse valor 1 é armazenado na PSW, para ser usado pela unidade de controle.

Por possuir apenas três flip-flops, o contador da figura 3.9 pode contar apenas de 0 até 7 ou de 7 até 0, mas nada mais do que isso, o que na verdade é suficiente para o projeto de 8 bits apresentado neste capítulo.

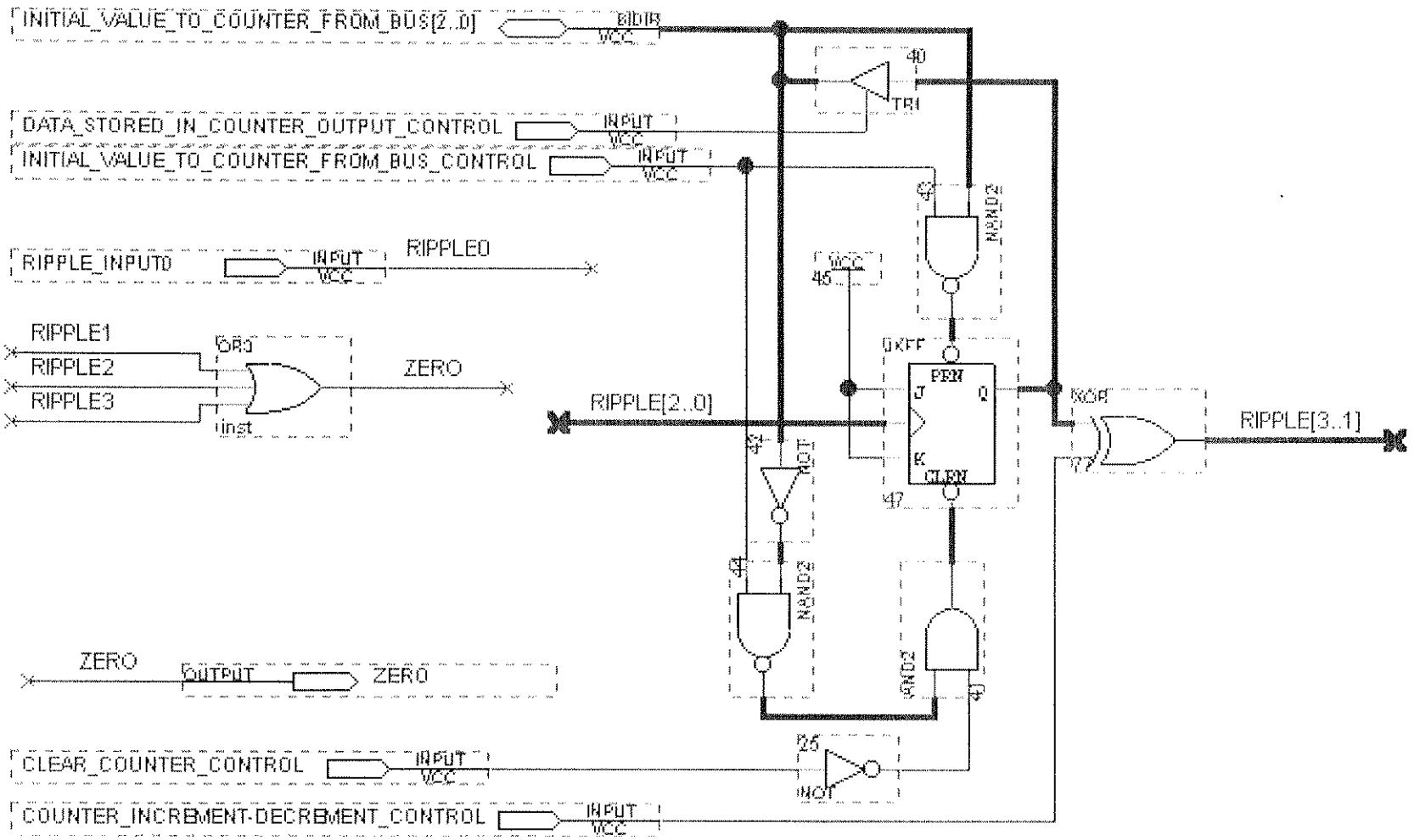

Figura 3.9: Estrutura interna de um contador.

3.7 Registrador Acumulador

O ACC é um registrador de 8 bits que armazena um dos dados de entrada da unidade lógico-aritmética. O resultado de uma operação é sempre armazenado de volta no ACC. Esse registrador recebe o nome de acumulador porque ele pode acumular os resultados de operações sucessivas realizadas pela unidade lógico-aritmética. Ele pode ser usado também para realizar deslocamentos à esquerda. A estrutura interna desse registrador é apresentada em detalhes na figura 3.10. O registrador é dotado de oito flip-flops JK e possui diversas entradas para sinais de controle e para dados.

Para que um novo valor possa ser armazenado no registrador, basta alimentar DATA_FROM-TO_ACC_TO-FROM_BUS[7..0], através do DATA_BUS, com o valor desejado. Quando DATA_INPUT_IN_ACC_CONTROL mudar seu valor de 0 para 1, o registrador armazenará o novo valor.

Se for desejado saber o valor presente no registrador, em um dado instante de tempo, basta mudar o valor de DATA_FROM-TO_ACC_TO-FROM_BUS_CONTROL de 0 para 1 e o conteúdo do registrador será apresentado no DATA_BUS, por meio do DATA_FROM-TO_ACC_TO-FROM_BUS [7..0].

A entrada DATA_STORED_IN_ACC_INPUT_CONTROL recebe o sinal de controle para a operação de deslocamento à esquerda. Quando ocorre um deslocamento à esquerda, o bit 8 (bit excedente) do registrador é desprezado e o novo bit 0 é introduzido no registrador através da entrada DATA_STORED_IN_ACC.

Através de DATA_FROM_ACC_TO_ALU, o conteúdo do ACC é passado adiante para a ALU. A entrada DATA_INVERTED_FROM_ACC_TO_ALU_CONTROL permite que o valor armazenado em ACC chegue invertido à ALU. Isso é importante para a operação aritmética de subtração (para realizar esta e outras operações, a ALU precisa, também, de dados provenientes do TEMP, o qual é mostrado na figura 3.11 e descrito na próxima seção).

O ACC recebe o resultado da operação proveniente da ALU através da entrada DATA_FROM_ALU_TO_ACC[7..0], que é controlada pelo sinal de controle DATA_FROM_ALU_TO_ACC_CONTROL.

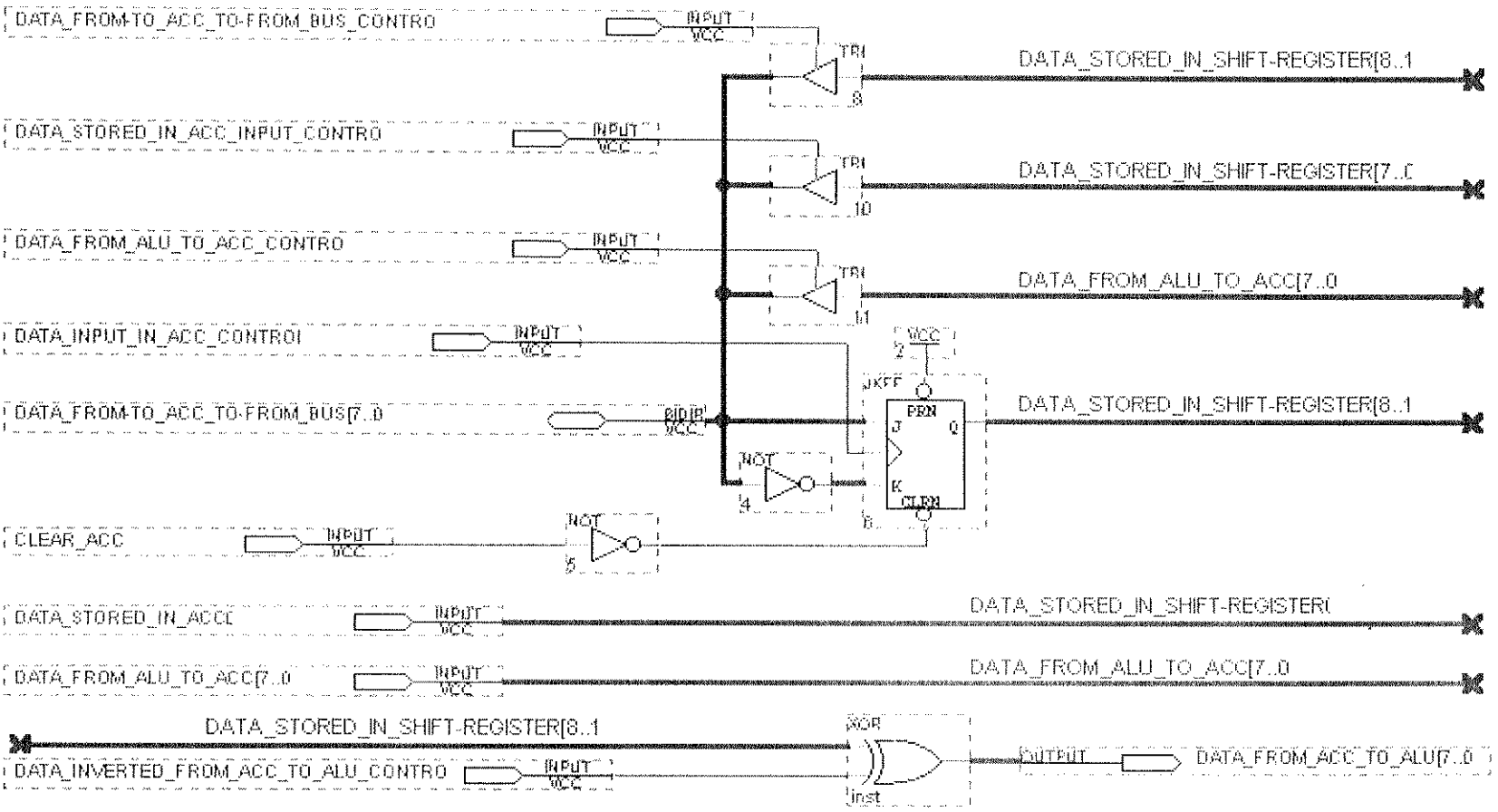

Figura 3.10: Estrutura interna do ACC.

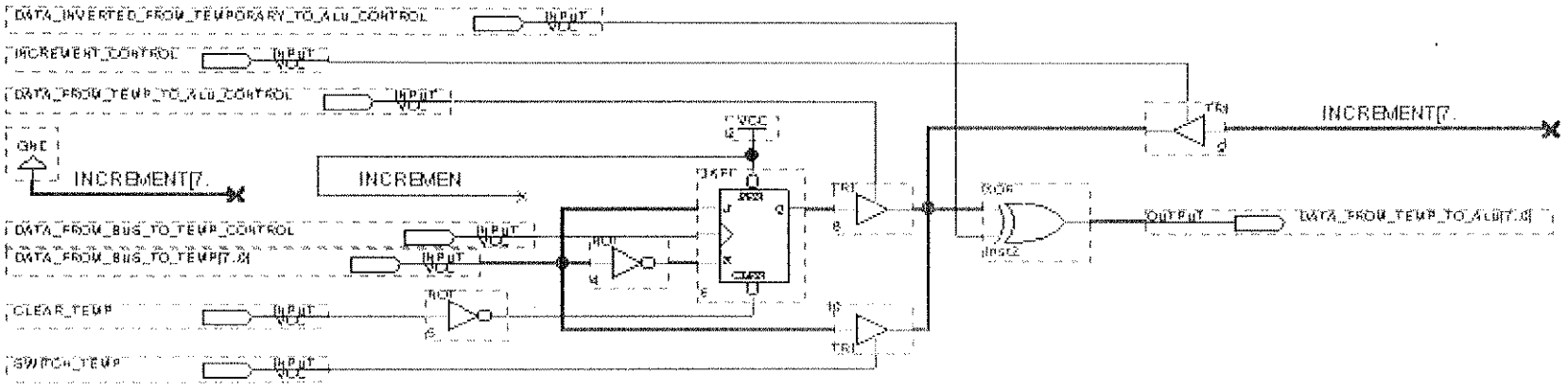

Figura 3.11: Estrutura interna do TEMP.

3.8 Registrador Temporário

O TEMP é um registrador de 8 bits que armazena um dos dados de entrada da unidade lógico-aritmética.

A estrutura interna desse registrador é apresentada em detalhes na figura 3.11. Através dela é possível notar que o registrador é dotado de oito flip-flops JK e possui diversas entradas para sinais de controle e para dados.

Para que um novo valor possa ser armazenado no registrador, basta alimentar DATA_FROM_BUS_TO_TEMP[7..0], através do DATA_BUS, com o valor desejado. Quando DATA_FROM_BUS_TO_TEMP_CONTROL mudar seu valor de 0 para 1, o registrador armazenará o novo valor.

O sinal CLEAR_SHIFT_REGISTER obriga o registrador a armazenar o valor 0.

Através de DATA_FROM_TEMP_TO_ALU, o conteúdo do TEMP é passado para a ALU. A entrada DATA_INVERTED_FROM_TEMPORARY_TO_ALU_CONTROL permite que o valor armazenado no TEMP chegue invertido à ALU.

A entrada SWITCH_TEMP permite que um valor presente no DATA_BUS seja passado diretamente à ALU, sem ser armazenado no TEMP.

INCREMENT_CONTROL gera o valor 1 em DATA_FROM_TEMP_TO_ALU. Isso é importante para a operação de complemento.

3.9 Comentários

A estrutura interna do co-processador M1, revelando em detalhes como cada um dos seus componentes foi projetado e como eles são integrados de modo a garantir o funcionamento adequado do co-processador M1, foi descrita neste capítulo.

Para agilizar e facilitar a descrição, este capítulo foi ilustrado com o exemplo de uma implementação do co-processador M1 com somente 8 bits. Para converter a implementação de 8 bits em uma outra de, por exemplo, 128 bits, é necessário substituir "7" por "127" em cada uma das entradas, saídas e sinais mostrados anteriormente neste capítulo. Por exemplo, a entrada DATA_FROM_BUS_TO_TEMP[7..0] seria alterada para

DATA_FROM_BUS_TO_TEMP**[127..0]**. Já para os circuitos contadores, que aqui contam números de 0 a 7, devem ter a sua entrada INITIAL_VALUE_TO_COUNTER_FROM_BUS**[2..0]** mudada para INITIAL_VALUE_TO_COUNTER_FROM_BUS**[6..0]**, a fim de passarem a contar números de 0 a 127. No capítulo 6 estas mudanças são mais profundamente discutidas.

Além desta flexibilidade de ampliar a largura dos dados, outros circuitos podem ser facilmente adicionados a esta estrutura, se a implementação de um novo algoritmo exigir. Essa facilidade só é possível graças à implementação do projeto sobre um dispositivo do tipo FPGA. Esse será o assunto abordado pelo próximo capítulo.

Capítulo 4
Dispositivo Usado para a Implementação do Co-processador M1: FPGA

Este capítulo descreve as características do dispositivo usado para a implementação do co-processador M1: uma FPGA, da família APEX20K, da Altera. Outras famílias de dispositivos FPGAs da Altera, que poderiam ser usadas na implementação do projeto, também são apresentadas neste mesmo capítulo.

4.1 A Família APEX 20K

Devido às exigências de alta densidade de elementos lógicos programáveis, o dispositivo escolhido como arquitetura base para implementação do co-processador M1 pertence à família APEX20K. Segundo a Altera, em [3], dispositivos de outras famílias também suportariam o projeto, mas para que o desempenho e eficiência fossem similares à implementação em um dispositivo APEX20K, seria necessário integrar um conjunto de dispositivos pertencentes a várias famílias diferentes como FLEX10K, FLEX6000 e MAX 7000, e implementar o projeto sobre todos eles ao mesmo tempo.

Muitos projetistas costumam usar uma combinação de dispositivos FLEX10K, FLEX6000 e MAX 7000 de forma integrada sobre a mesma placa, a fim de tirar melhor proveito das características particulares de cada família de dispositivos, visto que para cada família existirá sempre um tipo de aplicação em particular que terá uma implementação mais eficiente sobre seus dispositivos.

Entretanto, uma arquitetura que combine os benefícios proporcionados por essas três famílias permite integrar um sistema complexo em um único dispositivo. Isso acaba com a necessidade do emprego de múltiplos dispositivos integrados, salva espaço na placa de circuito impresso e facilita a implementação do sistema complexo.

A família APEX20K combina e incrementa as vantagens e características das famílias FLEX10K, FLEX6000 e MAX 7000 em um único dispositivo. A figura 4.1 (fonte: [4]) permite visualizar mais claramente essa integração

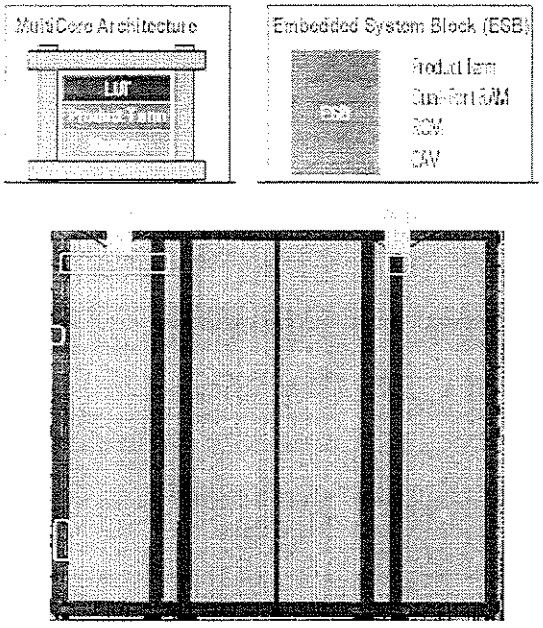

Figura 4.1: Estrutura responsável pelo desempenho da família APEX20K.

Conforme pode ser observado na figura 4.1, a arquitetura batizada de MultCore pela Altera combina três tipos de estruturas PLDs distintas: as LUTs (*Look-Up Tables*), comumente encontradas em membros das famílias FLEX10K e FLEX6000; os *product-term blocks*, normalmente presentes em dispositivos da família MAX7000; e os *enhanced embedded memory blocks*, que podem ser achados em integrantes da família FLEX10KE. Cada pequeno conjunto formado por essas três estruturas recebe o nome de LE (*Logic Element*). O grupo composto por dez desses elementos lógicos é batizado pela sigla LAB (*Logic Array Block*). Em um nível hierárquico superior, dezesseis LABs e um ESB (*Embedded Sistem Block*) compõem a estrutura conhecida por MegaLab. Essa estrutura é a maior responsável pelo desempenho da família APEX20K.

4.2 Outras famílias de dispositivos FPGAs da Altera

Para a implementação do projeto, encontrava-se à disposição um conjunto diversificado de famílias de dispositivos. Essas famílias serão resumidamente descritas a seguir:

APEX 20K – essa família é marcadamente caracterizada por apresentar o conjunto de dispositivos com a mais elevada densidade de elementos lógicos programáveis. Um único representante dessa família pode ser suficiente para integrar o projeto de um sistema inteiro. Cada dispositivo apresenta uma arquitetura, a qual integra *look-up table logic*, *product-term logic* e memórias (RAM, ROM e CAM – *Content Addressable Memory*). O trabalho integrado dessas estruturas torna mais fácil e eficiente a implementação de funções bastante complexas, como as Megafunctions, além de permitir alta performance das mesmas.

ACEX – os dispositivos pertencentes a essa família são *look-up table based*. Apresentam custo relativamente baixo, alta performance e uma densidade média de elementos lógico programáveis. Suportam uma comunicação de 66MHz via *slot* PCI de 64 bits.

FLEX10K – foi a primeira família de dispositivos a apresentar uma *embedded array*. Os EABs (*Embedded Array Blocks*) implementam funções relativas a RAM, ROM e FIFO. Muito usada para implementar aplicações de tempo real, que exigem ao mesmo tempo alta performance e média densidade de elementos lógico-programáveis.

FLEX 6000 e FLEX 8000 – compreendem dispositivos rápidos e eficientes, com áreas bastante reduzidas.

MAX 7000 e MAX 3000 – enquanto os dispositivos pertencentes à família MAX 7000 caracterizam-se pela alta velocidade, os dispositivos da família MAX 3000 são conhecidos pelo seu baixo custo, quando comparados com dispositivos de outras famílias.

A Figura 4.2 (fonte: [5]) apresenta um gráfico de comparação entre as características das famílias de FPGAs e a Tabela 4.2 (fonte: [5]) mostra a relação entre as diferentes famílias de dispositivos.

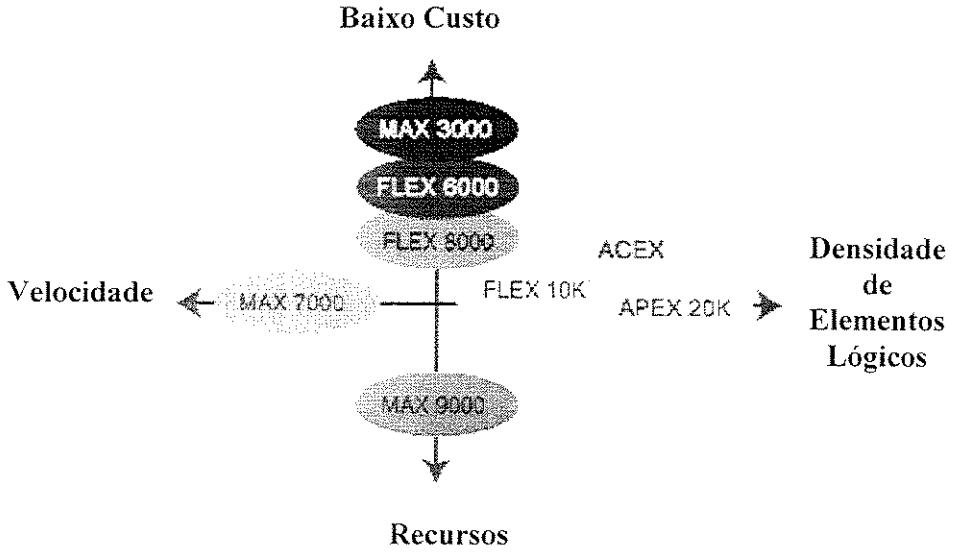

Figura 4.2: Gráfico de comparação entre as características das famílias de FPGAs.

A figura 4.2 revela que a família MAX 7000 é composta pelos dispositivos mais rápidos da Altera. Já os dispositivos pertencentes à família MAX 9000 são os que apresentam o maior número de recursos. As famílias MAX 3000, FLEX 6000 e FLEX 8000 possuem dispositivos de baixo custo. Finalmente, as maiores densidades ficam por conta dos dispositivos pertencentes às famílias FLEX 10K, ACEX e, sobretudo, APEX 20K. É possível notar, também através da figura 4.2, que, quanto mais denso, mais lento será o dispositivo e vice-versa; quanto mais recursos oferecer, mais caro será o dispositivo e vice-versa.

Assim, através de uma análise da figura 4.2, é possível deduzir que o dispositivo usado para a implementação do co-processador M1, uma FPGA pertencente à família APEX20K, é um dispositivo que apresenta alta densidade, baixa velocidade, custo médio e uma quantidade razoável de recursos.

Já a tabela 4.2 apresenta as quantidades de pinos e portas lógicas de cada família, assim como suas voltagens. Ela também revela que a FPGA APEX 20K, usada para implementar o co-processador M1, apresenta a maior quantidade de portas lógicas e o maior número de pinos dentre os dispositivos da Altera.

Tabela 4-1 - Relação quantificada entre diferentes tipos de dispositivos.

Famílias	Qtde. de Pinos	Qtde. de Portas Lógicas	Voltagens
APEX 20K	252 to 808	65,024 to 1,520,640	2.5 V and 1.8 V
ACEX	130 to 333	10,000 to 150,000	2.5 V and 1.8 V
FLEX 10K	59 to 470	10,000 to 250,000	2.5 V, 3.3 V and 5.0 V
FLEX 6000	71 to 218	10,000 to 24,000	3.3 V and 5.0 V
FLEX 8000	68 to 208	2,500 to 16,000	3.3 V and 5.0 V
MAX 9000	59 to 216	6,000 to 12,000	5.0 V
MAX 7000	36 to 212	600 to 10,000	2.5 V, 3.3 V and 5.0 V
MAX 3000	34 to 158	600 to 5,000	2.5 V and 3.3 V
Classic	22 to 64	300 to 900	5.0 V

4.3 Comentários

As características do dispositivo usado para a implementação do co-processador M1: uma FPGA, da família APEX20K, da Altera, são descritas neste capítulo. Outras famílias de dispositivos FPGAs da Altera, que poderiam ser usadas na implementação do projeto, também são apresentadas neste mesmo capítulo. No capítulo seguinte, a implementação de um algoritmo de criptografia assimétrica no co-processador M1 será descrita.

Capítulo 5
Implementação de um Algoritmo de Criptografia Assimétrica no Co-processador M1

Este capítulo revela como o algoritmo de criptografia assimétrica, para solução do problema do logaritmo discreto sobre curvas elípticas, foi implementado no co-processador M1 com intuito de mostrar as operações que podem ser realizadas pelo co-processador M1.

5.1 O algoritmo de ECC

Para esse algoritmo em especial, o co-processador M1 realiza a operação Q = kP, que é a multiplicação de um inteiro k por um ponto P, pertencente a uma determinada curva elíptica. A fim de realizar essa operação, o *buffer* que serve o co-processador M1 deve ser alimentado com os dados k, Xp, Yp, p e a, que representam, respectivamente, um inteiro, as coordenadas do ponto P na curva, um polinômio redutor e, finalmente, o valor de a, necessário para formar a curva. Esses dados podem ser informados, por exemplo, por um *software* de segurança, sendo executado pelo processador hospedeiro de uma máquina. Assim que o primeiro desses dados, o k, é armazenado no *buffer*, o co-processador M1 já inicia suas atividades, isto é, não é necessário esperar que todos os dados sejam colocados no *buffer* para começar a execução do algoritmo. Essa característica contribui para melhorar um pouco mais o desempenho do co-processador M1.

Segundo Macedo, em [11], o cálculo do produto de um número escalar k pelo ponto P de uma curva elíptica constitui a operação essencial do algoritmo de ECC. Pode-se realizar essa operação somando-se o ponto P k vezes. Porém, esse método é lento para ser usado em ECC. O algoritmo empregado para se executar essa função é o *Double_and_Add* (vide [6]), que faz a redução no número de somas necessárias para o cálculo do produto. Esse algoritmo baseia-se na adição e na duplicação do ponto, conforme o estado dos bits no número k. Dessa forma, as duas operações necessárias para o cálculo de kP são a adição de dois pontos e a duplicação de um ponto.

Por exemplo, para Q = 10P, o valor de k, em binário, é 00001010. O algoritmo analisa esse número a partir do bit mais à esquerda até o bit mais à direita. A idéia é encontrar a primeira ocorrência do valor 1, neste caso para o bit número 3. A partir desse instante, o algoritmo irá considerar os valores dos bits seguintes ao bit número 3 para executar sua tarefa (isto é, 010): quando o algoritmo detectar a ocorrência de um bit com valor 0, ele fará a duplicação do ponto P; quando o algoritmo detectar a ocorrência de um bit com valor 1, ele fará a duplicação do resultado anterior e somará com P. Ou seja, para 010, o algoritmo irá considerar: ((2P)2+P)2, que é equivalente a 10P. Como pode ser notado, ao invés de somar P a ele mesmo por dez vezes, o emprego do *Double_and_Add* para este exemplo permitiria obter o mesmo resultado com apenas quatro operações.

Outro exemplo seria, para Q = 17P, o valor de k, em binário, é 00010001. O algoritmo analisa esse número a partir do bit mais à esquerda até o bit mais à direita. A idéia é encontrar a primeira ocorrência do valor 1, neste caso para o bit número 4. A partir desse instante, o algoritmo irá considerar os valores dos bits seguintes ao bit número 4 para executar sua tarefa (isto é, 0001): conforme comentado anteriormente, quando o algoritmo detectar a ocorrência de um bit com valor 0, ele fará a duplicação do ponto P; quando o algoritmo detectar a ocorrência de um bit com valor 1, ele fará a duplicação do resultado anterior e somará com P. Ou seja, para 0001, o algoritmo irá considerar: (((2P)2)2)2+P, que é equivalente a 17P. Como pode ser notado, ao invés de somar P a ele mesmo por dezessete vezes, o emprego do *Double_and_Add* para este exemplo permitiria obter o mesmo resultado com apenas cinco operações. Os dois exemplos aparecem resumidos na figura 5.1.

A adição de dois pontos distintos pode ser compreendida considerando-se uma curva elíptica de parâmetros a, b e p com $p = 2^m$, para a qual a soma entre seus pontos $P(X_P, Y_P)$ e $R(X_R, Y_R)$, resulta em um ponto $Q(X_Q, Y_Q)$, cujas coordenadas são expressas por (fonte: [7]):

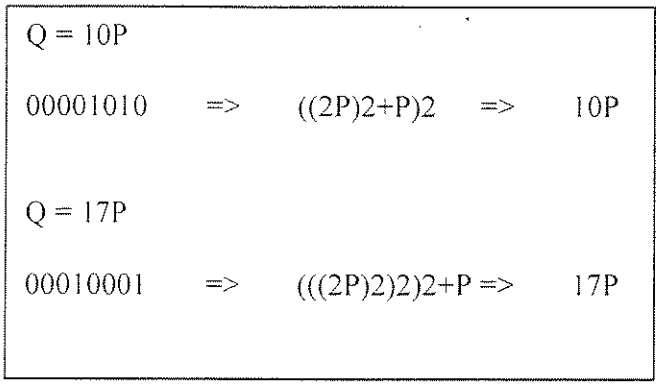

Figura 5-0 -: Dois exemplos do emprego do método *Double an Add* para o calculo de Q=kP.

$$s = ((y_P - y_R) / (x_P + x_R)) \bmod p \quad (5.1)$$

$$x_Q = (s^2 + s + x_P + x_R + a) \bmod p \quad (5.2)$$

$$y_Q = (s(x_P + x_R) + x_R + y_P) \bmod p \quad (5.3)$$

Para a duplicação do ponto P, dado o ponto $P(X_P, Y_P)$ e o ponto $Q(X_Q, Y_Q) = 2P$, as coordenadas de Q são (fonte: [7]):

$$s = ((x_P + y_P) / x_P) \bmod p \quad (5.4)$$

$$x_Q = (s^2 + s + a) \bmod p \quad (5.5)$$

$$y_Q = (x_P + (s + 1) * x_Q) \bmod p \quad (5.6)$$

O co-processador M1 encerra sua operação retornando as coordenadas do ponto Q para o aplicativo, através do mesmo *buffer*.

Quando se trabalha com aritmética modular, faz-se necessário o emprego da operação conhecida por inverso multiplicativo.

Considere a operação (fonte: [7]):

$$x = (a/b) \bmod p, \quad (5.7)$$

para a qual a, b, x e p são inteiros pertencentes ao corpo finito.

A divisão de a por b é definida como o produto de a por b^{-1}. Assim, essa divisão só existe quando b^{-1} existe.

Para ter certeza absoluta que o resultado da operação anterior gerará um inteiro como resultado, é necessário, inicialmente, calcular o inverso multiplicativo de b, isto é, achar b^{-1}, a partir de b e p, e depois multiplicá-lo por a.

$$x = (a/b) \bmod p \Rightarrow x = (a.b^{-1}) \bmod p \qquad (5.8) \quad \text{(fonte: [7])}$$

Em um sistema criptográfico baseado nas curvas elípticas, as velocidades dos cálculos do inverso multiplicativo, das multiplicações e das divisões estão diretamente relacionadas ao desempenho desse sistema. As operações (adição, duplicação e multiplicação por um escalar k), realizadas com pontos pertencentes a uma determinada curva elíptica sobre um corpo finito, resultam em um ponto, que também pertence a essa mesma curva elíptica.

Os sistemas criptográficos baseados nas curvas elípticas se apóiam no conceito do problema do logaritmo discreto sobre curvas elípticas para empregarem um método de troca de chaves chamado de Diffie-Hellman (vide [17]).

O algoritmo, escrito em VHDL, ocupa a maior parte dos elementos lógicos presentes em um dispositivo FPGA escolhido para implementar o co-processador M1.

O método empregado para descrever este algoritmo em VHDL foi o *Dataflow*. Outros métodos, que podem ser empregados em descrições de algoritmos através de VHDL, são comentados no apêndice D.

5.2 Comentários

Este capítulo revelou como o algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas, foi implementado no co-processador M1 com intuito de demonstrar as operações que podem ser realizadas pelo co-processador M1.

Os próximos capítulos apresentam resultados obtidos a partir de simulações do co-processador M1, usando esse algoritmo de criptografia. Algumas comparações com outros trabalhos são comentadas.

Capítulo 6
Resultados

Este capítulo apresenta os resultados finais do trabalho, em notação hexadecimal, obtidos a partir das compilações e das simulações do co-processador M1. A ferramenta usada para realizar a simulação foi o Quartus II, da Altera (vide[apêndice E]).

6.1 Resultados da Simulação

Todos os resultados apresentados neste capítulo são frutos das diversas simulações realizadas para a implementação de 8 bits do co-processador M1. Optou-se por relatar apenas a simulação da versão de 8 bits, porque os seus resultados demonstram o adequado funcionamento do co-processador M1. Além disso, essas simulações são relativamente rápidas e confortáveis, quando comparadas às simulações para as versões de 64 e 128 bits. Enquanto o simulador levava três minutos para gerar cada resultado para a implementação de 8 bits, para as outras duas implementações, eram necessários, respectivamente, 32 e 50 minutos para cada resultado, sem contar o tempo de compilação, que será descrito na próxima seção.

Os resultados apresentados nesta seção referem-se ao cálculo Q = kP, discutidos no capítulo 5. Sete valores diferentes de k foram multiplicados por três diferentes pontos escolhidos sobre duas curvas elípticas distintas. Os dados usados para a simulação do co-processador M1 foram fornecidos por [11]. Os valores finais da simulação do co-processador M1 foram comparados aos valores obtidos em testes de uma implementação em *software*, realizada em um trabalho descrito por [11]. O co-processador M1 obteve sucesso para todos os valores.

Para a primeira curva elíptica, definida por a = 7 e b = F, as coordenadas dos pontos escolhidos foram: (6,4), (A_H,E_H) e (2,5). A tabela 6.1 (fonte: [11]) mostra os resultados da multiplicação de cada um dos sete valores do escalar k pelos pontos escolhidos sobre a curva, considerando a=7, b=F_H e p=13. Todos os resultados representam coordenadas de pontos pertencentes a essa mesma curva elíptica.

Tabela 6-1 - Multiplicação por um escalar k para pontos da curva com a=7, b=F_H e p=13.

Multiplicação por um escalar k para pontos da curva com a=7, b=F e p=13						
P	2P	3P	4P	5P	6P	7P
(6,4)	($3,C_H$)	($3,F_H$)	(6,2)	O	(6,4)	($3,C_H$)
(A_H,E_H)	(2,5)	(9,2)	($3,F_H$)	(8,2)	(4,8)	(B_H,F_H)
(2,5)	($3,F_H$)	(4,8)	(6,4)	($0,C_H$)	(6,2)	($4,C_H$)

A tabela 6.2 mostra a quantidade de tempo, em milisegundos, que o co-processador M1 leva para calcular cada um dos pontos (resultados) da tabela 6.1.

Tabela 6-2 - Tempo (em ms) necessário para realizar a multiplicação por um escalar k para pontos da curva com a=7, b=F_H e p=13.

Tempo (em ms) necessário para realizar a multiplicação por um escalar k para pontos da curva com a=7, b=F_H e p=13						
P	2P	3P	4P	5P	6P	7P
(6,4)	0,542	1,171	1,061	1,089	1,687	2,199
(A_H,E_H)	0,772	1,299	1,230	1,584	1,765	2,175
(2,5)	0,484	0,840	1,001	1,584	1,417	2,050

Para a segunda curva elíptica, definida por a = 2 e b = 6, as coordenadas dos pontos escolhidos foram: (E_H,2), (8,A_H) e (7,A_H). A tabela 6.3 (fonte: [11]) mostra os resultados da multiplicação de cada um dos sete valores do escalar k pelos pontos escolhidos sobre a curva, considerando a=2, b=6 e p=13_H. Todos os resultados representam coordenadas de pontos pertencentes a essa mesma curva elíptica.

Tabela 6-3 - Multiplicação por um escalar k para pontos da curva com a=2, b=6 e p=13_H.

Multiplicação por um escalar k para pontos da curva com a=2, b=6 e p=13_H						
P	2P	3P	4P	5P	6P	7P
(E_H,2)	(6,8)	(9,9)	(0,7)	(9,0)	(6,E_H)	(E_H,C_H)
(8,A_H)	(5,E_H)	(9,0)	(1,E_H)	(F_H,8)	(6,8)	(D_H,A_H)
(7,A_H)	(7,D_H)	O	(7,A_H)	(7,D_H)	O	(7,A_H)

A tabela 6.4 mostra a quantidade de tempo, em milisegundos, que o co-processador M1 leva para calcular cada um dos pontos (resultados) da tabela 6.3.

Tabela 6-4 - Tempo (em ms) necessário para realizar a multiplicação por um escalar k para pontos da curva com a=2, b=6 e p=13_H.

Tempo (em ms) necessário para realizar a multiplicação por um escalar k para pontos da curva com a=2, b=6 e p=13_H						
P	2P	3P	4P	5P	6P	7P
(E_H,2)	0,484	1,113	1,007	1,483	1,614	2,185
(8,A_H)	0,652	1,175	1,177	1,739	1,635	2,095
(7,A_H)	0,494	0,524	0,968	1,436	0,524	0,550

Os resultados das simulações apresentados nas tabelas 6.1, 6.2, 6.3 e 6.4 demonstram que o co-processador M1 funciona adequadamente. Comparações feitas entre essas tabelas e resultados obtidos através de outros trabalhos, comentados a seguir, mostram que o desempenho do co-processador M1 é inferior ao desempenho de um circuito dedicado a um único algoritmo criptográfico. Isso pode ser explicado porque os parâmetros de compilação do projeto do co-processador M1 foram escolhidos de forma a priorizar uma menor área de circuito, em detrimento a uma maior velocidade. Além disso, a FPGA escolhida para implementar o co-processador M1 pertence à família APEX 20K, que compreende os dispositivos mais lentos da Altera, como pode ser visto através da figura 4.2, exibida no capítulo 4. E ainda, somado a tudo isso, circuitos descritos em VHDL costumam apresentar desempenho menor do que circuitos projetados através de esquemático. Como uma parte significativa do co-processador M1 foi descrita através de VHDL, já era de se esperar que isso também contribuísse para uma queda no desempenho do co-processador M1.

Para comprovar essa diferença de desempenho, paralelamente ao projeto do co-processador M1, iniciou-se o desenvolvimento de um circuito de 8 bits, dedicado exclusivamente ao algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas. O circuito apresenta-se desenvolvido quase inteiramente através de esquemático. Suas simulações também são realizadas com o auxílio do Quartus II.

Para ambas as implementações, as operações que consomem mais tempo de processamento são: o inverso multiplicativo, a divisão e a multiplicação. Esta última, só para se ter uma idéia, é realizada em 14ns (nanosegundos) pelo circuito dedicado, enquanto

leva 60us (microsegundos) para ser efetuada pelo co-processador M1. Esse resultado obtido com o circuito dedicado supera até mesmo o desempenho dos trabalhos descritos em [13] e [19].

Um outro trabalho usado para comprovar essa diferença de desempenho é descrito em [8]. Trata-se de um relato a respeito de sistemas criptográficos baseados em curvas elípticas para *smart cards*. Os *smart cards* oferecidos atualmente no mercado possuem uns poucos Kbytes de memórias RAM, ROM e EEPROM, além de uma CPU de 8 bits trabalhando a uma freqüência relativamente baixa (pouco superior a 3MHz). Em outras palavras, essa implementação do algoritmo de ECC para *smart cards* trabalha com um *clock* pouco superior a 300ns (nanosegundos), enquanto o co-processador M1 trabalha com um *clock* de 700ns (nanosegundos).

Levando em consideração a implementação de 128 bits e não mais a de 8 bits, a área ocupada pelo M1 pode ser comparada com a área ocupada pelo processador descrito em [13].

Em [13] é possível encontrar a descrição de um processador de alta performance para sistemas criptográficos baseados em curvas elípticas sobre $F2^m$, implementado em uma FPGA. A arquitetura usada como base para a implementação do co-processador M1 foi uma FPGA da Xilinx, a XCV400E-8-BG432 (Virtex E). O projeto foi codificado em VHDL (vide [15]) e sintetizado através das ferramentas FPGA Express 3.0 da Synopsis e Design Manager M2.1i da Xilinx (vide [13]).

Pode-se dizer que a diferença entre as áreas ocupadas pelos dois projetos é, de certa forma, pequena, pois embora a FPGA XCV400E-8-BG432 apresente um número de elementos lógicos um pouco inferior ao da FPGA EP20K600EBC652-1, usada para implementar o co-processador M1 de 128 bits, a complexidade do projeto do co-processador M1 e bem superior à complexidade do projeto descrito em [13]. Sendo assim, poder-se-ia esperar que o co-processador M1 ocupasse uma área bem maior do que a área do outro projeto, entretanto isso não aconteceu.

6.2 Resultados das Compilações

As compilações foram feitas através do *software* Quartus II, executado sobre um processador Intel Pentium 4, de 1,5GHz de freqüência, com 1024Mbytes de memória RAM e 30Gbytes de HD.

A tabela 6.5 mostra os resultados das compilações do co-processador M1, para as suas versões de 8, 64 e 128 bits. Ela permite constatar que a compilação do co-processador M1 de 8 bits é relativamente rápida, se for levado em consideração que as compilações dos co-processadores M1 de 64 e 128 bits consomem um tempo superior a 24h.

Realizando-se uma análise sobre essa mesma tabela, é possível comprovar que a porção do co-processador M1 descrita em VHDL ocupa uma grande parte do total da área destinada ao dispositivo. Para verificar isso, é preciso, inicialmente, saber que a descrição feita em VHDL ocupou sempre a mesma área para as três diferentes implementações e que o co-processador M1 de 128 bits é composto pelos mesmos circuitos presentes no co-processador M1 de 64 bits, com a diferença que os circuitos daquela implementação apresentam o dobro do tamanho desta. Apesar de toda essa diferença de tamanho, segundo a tabela 6.5, o co-processador M1 de 128 bits usa apenas 4930 elementos lógicos (vide [20]) a mais que o co-processador M1 de 64 bits, ou seja, a cada 64 bits adicionados no projeto do co-processador M1, registra-se um incremento de 4930 elementos lógicos para a área da implementação. Isso significa que, em todas as implementações, 9608 elementos lógicos são dedicados apenas para a porção descrita em VHDL, o que representa mais de 90% da área da versão de 8 bits, 66% da área da versão de 64 bits e, aproximadamente, 50% da área da versão de 128 bits.

Tabela 6-5 - Resultados das Compilações.

Resultados das Compilações				
Tamanho do Co-processador	Nome do Dispositivo	Total de Elementos Lógicos	Total de Pinos	Tempo de Compilação (dias:horas:minutos)
8 bits	EP20K400EBC652-1	10622 / 16640 (63%)	164 / 488 (33%)	00:02:10
64 bits	EP20K400EBC652-1	14538 / 16640 (87%)	216 / 488 (44%)	01:17:51
128 bits	EP20K600EBC652-1	19468 / 24320 (80%)	408 / 488 (83%)	02:02:30

A tabela 6.5 também permite notar que as implementações de 8 e 64 bits do co-processador M1 ocupam, respectivamente, 63% e 87% da área total do dispositivo EP20K400EBC652-1, que possui, ao todo, 16640 elementos lógicos. Já a implementação de 128 bits ocupa 80% da área total do dispositivo EP20K600EBC652-1, que possui, ao todo, 24320 elementos lógicos.

Com base nesses dados, é possível deduzir que a maior implementação do co-processador M1 teria 512 bits e ocuparia 49048 elementos lógicos da FPGA EP20K1500EBC652-1X, a qual possui um total de 51840. Isso representa 95% da sua área total. Essa é a FPGA que apresenta maior área dentre os dispositivos da Altera.

6.3 Comentários

Os resultados finais do trabalho, obtidos a partir da compilação e da simulação do co-processador M1, com o auxílio da ferramenta Quartus II da Altera, foram apresentados neste capítulo.

Capítulo 7
Considerações Finais

7.1 Comentários

O desenvolvimento do M1 para algoritmos de criptografia assimétrica (vide [17]) foi coberto por este trabalho. Todos os capítulos anteriores contribuíram para descrever o projeto do M1. O M1 dispõe de uma biblioteca de módulos de circuitos que implementam rotinas básicas úteis a algoritmos de criptografia assimétrica, fazendo com que ele não seja apenas um dispositivo dedicado a um único algoritmo criptográfico. Com o objetivo de demonstrar as operações que podem ser realizadas pelo M1, escolheu-se empregar o algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas (vide [6], [7], [10], [11] e [12]). Foram apresentados os resultados provenientes da simulação do projeto. Os testes práticos do co-processador M1 foram baseados no uso de curvas elípticas distintas e de diferentes pontos dispostos sobre essas mesmas curvas. Comentários foram feitos com base na comparação dos resultados obtidos através desses testes com resultados de testes provenientes de implementações tanto em *software* como em *hardware* realizadas em outros trabalhos. O M1 foi inteiramente compilado, sintetizado e programado utilizando-se o *software* Quartus II (vide [1]), também da Altera.

7.2 Conclusão

Pode-se concluir que este trabalho teve um resultado bastante positivo, tanto em termos de realização de pesquisa, como em termos de desenvolvimento de projetos. Portanto, este trabalho representa uma fonte complementar de pesquisa para aqueles que visam aprofundar-se no estudo da criptografia, sobretudo para aqueles que têm a intenção de trabalhar com projetos de *hardware*, voltados a ambientes onde se faz necessário empregar sistemas de garantia de segurança para as suas informações.

Ao final deste trabalho, conclui-se que, não só foi possível implementar o co-processador M1 para algoritmos de criptografia assimétrica, como também os resultados

das simulações comprovam que o co-processador M1 funciona de forma adequada. Como era esperado desde o início do projeto, o desempenho do co-processador M1 é inferior ao desempenho de um circuito dedicado a um único algoritmo criptográfico, devido a três fatores principais:

- os parâmetros de compilação do projeto do co-processador foram escolhidos de forma a priorizar uma menor área de circuito, em detrimento a uma maior velocidade;
- a FPGA escolhida para implementar o co-processador M1 pertence à família APEX 20K, que compreende os dispositivos mais lentos da Altera, como pode ser visto através da figura 4.2, exibida no capítulo 4;
- circuitos descritos em VHDL no ambiente Quartus costumam apresentar desempenho menor do que circuitos projetados através de esquemático.

Ainda com relação à questão de desempenho, os resultados obtidos através deste trabalho permitem concluir que uma maior velocidade de processamento pode ser alcançada através de circuitos dedicados exclusivamente a um único algoritmo de criptografia. Portanto, este trabalho também demonstra que, se a alta velocidade representar uma questão crucial em um determinado projeto, este deve ser dedicado exclusivamente a um único algoritmo de criptografia. Deverá também ser descrito, preferencialmente, em esquemático e implementado sobre um ASIC, visto que esse dispositivo costuma apresentar melhor desempenho e custo inferior, quando comparado às FPGAs.

Com relação às questões envolvendo área de um projeto, pode-se concluir que é possível implementar um co-processador criptográfico para algoritmos de criptografia assimétrica de até 512 bits em uma única FPGA da Altera, conforme foi explicado no capítulo 6.

7.3 Sugestões para Trabalhos Futuros

- Dar continuidade ao trabalho do co-processador M1: realizar a sua síntese sobre o chip FPGA da ALTERA usado para a simulação;
- Usar para a implementação uma FPGA de outro fabricante, a fim de fazer comparações entre as duas implementações;

- Utilizar outras ferramentas para sintetizar o circuito;
- Desenvolver um co-processador criptográfico de alta performance para algoritmos de criptografia assimétrica;

Referências Bibliográficas

[1] Altera Corporation, Quartus, Programmable Logic Development System & Software, Data Sheet, ver. 1.01, 1999.

[2] Altera Corporation, APEX 20K, Programmable Logic Device Family, Data Sheet, ver. 4.1, 2001.

[3] Altera Corporation, APEX 20K, Device Family Architecture, 2001.

[4] Altera Corporation, APEX: A Revolutionary Embedded Architecture, 2001.

[5] Altera Corporation, Device Selector Guide, 2001.

[6] Blake, Ian; Seroussi, Gadiel; Nigel, Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.

[7] Certicom Research, SEC 1: *Elliptic Curve Cryptography*, Standards for Efficient Cryptography, 2000, Version 1.0.

[8] Certicom Research, *The Elliptic Curve Processor for Smart Cards*, the seventh in a series of ECC white papers, 2000.

[9] Gajski, Daniel D.; Vahid, Frank; Narayan, Sanjiv; Gong, Jie, *Specification and Design of Embedded Systems*, 1st ed., Prentice-Hall, 1994.

[10] Institute of Electrical and Electronics Engineers (IEEE), Inc. P1363 – *Standard Specification for Public Key Cryptography*, 2000.

[11] Macedo, Robson G., Relatório Técnico: Algoritmos de Criptografia Baseados em Curvas Elípticas: Implementação em Ambientes Computacionais Restritos, FEEC/UNICAMP, 2001.

[12] Menezes, Alfred J., *Elliptic Curve Public Key Cryptosystems*, Kluer Academic Publishers, 1993.

[13] Orlando, Gerardo; Paar, Christof, A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m), Workshop on Cryptographic Hardware and Embedded Systems (CHES), 2000.

[14] Paar, Christof, Security in Future Embedded Ad-hoc Networks, *WPI Cryptography and Data Security Seminar*, 2001.

[15] Perry, Douglas, *VHDL*, 3rd ed., McGraw-Hill, 1999.

[16] Stallings, Willian, *Computer Organization and Architecture*, 4th ed., Prentice-Hall, 1996.

[17] Stallings, Willian, *Network and Internetwork Security*, 2nd ed., Prentice-Hall, 1998.

[18] Tanenbaum, Andrew S., *Computer Networks*, 3rd ed., Prentice-Hall, 1996.

[19] Torii, Naoya; Okada, Souichi; Itoh, Kouichi; Takenaka, Masahiko, Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on an FPGA, Fujitsu Laboratories Ltd., 2000.

[20] Virtual Computer Corporation, Field Programmable Gate Arrays (FPGAs) – An Enabling Technology, 2000.

[21] Zelenovsky, Ricardo e Mendonça, Alexandre. *PC: Um guia prático de Hardware e interfaceamento*. 2a. edição MZ Editora, 1999

Apêndice A
Segurança em Redes de Computadores

Este apêndice faz uma breve explicação sobre segurança em redes de computadores, criptografias simétrica e assimétrica, autenticação, assinaturas digitais, curvas elípticas e o problema do logaritmo discreto sobre curvas elípticas.

A.1 A Criptografia

Uma das principais características humanas é a capacidade de comunicação. Desde os primórdios, ela tem se mostrado fundamental para o desenvolvimento social e intelectual do homem, pois ela é o meio pelo qual conhecimentos e experiências são compartilhados entre indivíduos, entre agrupamentos humanos distintos e, dentro destes, entre as diferentes gerações.

Entretanto, nem toda comunicação tem caráter público, isto é, muitas vezes é desejável que uma determinada informação seja compartilhada apenas entre uns poucos indivíduos. Não raros são também os casos em que as informações são tão particulares a um indivíduo, que o mesmo evita qualquer meio de comunicá-las. Essas formas de sigilo passam a ser uma necessidade para o homem, quando a informação de que ele tem posse pode prejudicá-lo de alguma forma, se chegar ao conhecimento de determinados indivíduos mal intencionados.

Infelizmente, nem toda característica humana é tão admirável quanto à capacidade de comunicação. Ao contrário, determinados comportamentos do homem podem até mesmo vir a comprometer a troca e a guarda de informações. Um caso típico é aquele em que alguém destrói ou altera as informações de outra pessoa, pelo simples prazer de prejudicá-la. Um outro caso se dá quando uma pessoa acessa informações de outra com o objetivo de espiar sua vida particular. Outro caso é aquele em que um indivíduo nega ter passado ou

recebido determinada informação. Vale a pena comentar ainda um quarto caso muito comum, no qual informações são roubadas, visando benefício próprio.

Para evitar esses tipos de problemas, durante toda a história, o homem tem inventado diferentes métodos de assegurar as informações, seja para seu armazenamento ou para sua comunicação. Um dos mais notáveis e também um dos mais antigos desses métodos é a criptografia. Ela consiste em transformar a informação privada em um código, o qual somente poderá ser decifrado por quem detiver o método de decodificação para aquele código. É sabido que esse artifício tem sido usado para manter informações sigilosas desde os tempos do Império Romano.

Muito tempo passou desde aquela época, mas a importância da criptografia é tamanha, que ela continua sendo muito utilizada ainda hoje. Isso é fácil de entender, uma vez que as características, necessidades e defeitos humanos, citados anteriormente, não se perderam com o passar do tempo. A diferença mais notável atualmente é que existe um maciço e sempre crescente uso de computadores, tanto para o armazenamento quanto para a comunicação das informações. A criptografia precisou apenas ser adaptada a esse ambiente digital.

Entretanto, essa adaptação não tem se revelado uma tarefa tão simples quanto possa parecer à primeira vista. Primeiro, porque as mudanças tecnológicas no mundo da computação são freqüentes. Segundo, porque as técnicas de ataque usadas para superar os métodos que asseguram informações têm se tornado cada vez mais sofisticadas e eficientes. Dessa forma, embora a idéia básica da criptografia tenha permanecido a mesma através dos tempos – a codificação de dados – para garantir a segurança de informações nesse ambiente mutante, a criptografia está ganhando crescente diversificação e sofisticação, quanto aos métodos empregados para a codificação de dados. Além disso, ela tem sido usada em associação com outras ferramentas de segurança direta ou indiretamente relacionadas a ela, como, por exemplo, as assinaturas digitais, entre outras.

A.2 Garantias de Segurança

Como conseqüência da evolução do uso dessas ferramentas, a segurança, que é o conjunto de práticas e ferramentas usado para prover segurança às informações em computadores e redes, passou a oferecer uma série mais ampla de garantias para assegurar essas informações. Dentre elas é possível citar:

Confidentiality - garantia de que o conteúdo, a fonte, o destino, a freqüência e o tamanho de mensagens permaneçam confidenciais durante uma transmissão.

Authentication - garante uma identificação confiável da origem e do destino das mensagens: se o conteúdo da mensagem afirma que ela veio de uma fonte X, então a ferramenta deve assegurar que ela realmente tenha vindo de X; deve ser garantido que as partes envolvidas em uma conexão são realmente quem afirmam ser; deve impedir que uma terceira entidade seja intermediadora da comunicação entre outras duas, quando aquela visa interceptar as mensagens destas, com propósito de efetuar qualquer tipo de ataque.

Integrity - deve garantir que uma mensagem seja recebida pelo receptor, da exata forma como foi anteriormente enviada pelo transmissor, isto é, sem duplicação, inserção, modificação, reordenação, retransmissão ou destruição dos dados.

Nonrepudiation - impede que tanto o emissor quanto o receptor neguem a existência de uma mensagem transmitida.

Access control - limita e controla o acesso a sistemas servidores e aplicações via *links* de comunicação.

Availability - impede ataques que degradam a eficiência de um sistema ou rede de computadores.

A.3 Tipos de Ataques

Essa variedade de garantias visa impedir, prevenir, detectar e suportar diversos tipos de ataques contra a segurança das informações, sendo os mais comuns:

- *Interruption* (*active*[1] *attack on availability*) – destruição de *hardware*; corte de cabo; inibir uso de facilidade de comunicação (*denial of service*); etc.

- *Interception* (*passive*[2] *attack on confidentiality*) – observação do conteúdo (*release of message contents*), da fonte, do destino, da freqüência e do tamanho de mensagens em uma linha telefônica ou de *e-mail* (*traffic analysis*); etc.

- *Modification* (*active attack on integrity*) – alterar mensagens; mudar valores em bases de dados; alterar programas para que executem de outra forma; fazer com que outros violem protocolos por modificação ou introduzindo informações incorretas; etc.

- *Fabrication* (*active attack on authenticity*) – inserção de mensagens; retransmissão de mensagem (*replay*); adição de arquivos ou programas para comprometer outras pessoas; um usuário tenta passar-se por outro (*mascarade*); afirmar ter recebido de outro usuário informação criada por si mesmo; afirmar ter enviado informação que não enviou; etc.

- *Invasion* (*active attack on availability and on access control*) – virus, *hackers* e *worms*; etc.

A figura A.1 (fonte: [17]), apresentada mais à frente, ilustra os diferentes tipos de ataques comentados anteriormente.

A.4 Ferramentas de Segurança

Atualmente, existem numerosas ferramentas que oferecem as garantias necessárias contra os ataques à segurança de informações em computadores e redes. As mais conhecidas serão brevemente descritas a seguir.

- DES (*Data Encryption Standard*) – trata-se de um algoritmo de criptografia simétrica largamente utilizado, principalmente para garantir sigilo.

[1] Ataques ativos são fáceis de detectar porque há alteração de dados, mas são difíceis de prevenir.
[2] Ataques passivos são difíceis de detectar porque não há alteração de dados, mas são fáceis de prevenir.

- RSA (Rivest-Shamir-Adleman) – trata-se de um algoritmo de criptografia assimétrica também muito utilizado, principalmente para garantir sigilo.
- ECC (*Elliptic Curve Cryptosystem*) – assim como o RSA, é um algoritmo de criptografia assimétrica, empregado para garantir sigilo.
- RIJNDAEL – assim como o DES, é um algoritmo de criptografia simétrica, empregado para garantir sigilo.
- IDEA (*International Data Encryption Algorithm*) – foi proposto como um padrão internacional de criptografia.
- SKIPJACK – Esse algoritmo foi desenvolvido pelo NSA (*National Security Agency*) para garantir segurança em comunicações de voz e dados e, ao mesmo tempo, prover acesso a agências autorizadas, para estas fazerem escuta telefônica.
- LUC – é um algoritmo de criptografia assimétrica tão forte e eficiente quanto o RSA.
- MD5 (*Message Digest Algorithm*) – esta importante *Hash Function* é talvez o algoritmo mais popular para prover autenticação e assinaturas digitais em uma variedade de aplicações.
- SHA (*Secure Hash Algorithm*) – é similar ao MD5 e foi determinado como padrão federal pelo NIST (*U.S. National Institute of Standards and Tecnology*).
- Kerberos – é um protocolo para autenticação baseado em criptografia simétrica, que tem recebido largo suporte e é usado em vários sistemas.
- X.509 – especifica um algoritmo para autenticação e define uma *certificate facility*. Esta última habilita usuários a obterem certificados de chaves públicas, de forma que uma comunidade de usuários possa ter garantia da autenticidade dessas chaves públicas.
- DSS (*Digital Signature Standard*) – é um padrão federal de assinatura digital proposto pelo NIST.
- PGP (*Pretty Good Privacy*) – trata-se de uma ferramenta largamente utilizada para garantir *authenticity* e *confidentiality* em correio eletrônico. É independente de qualquer organização ou autoridade.

- PEM (*Privacy Enhanced Mail*) – também se trata de uma ferramenta largamente utilizada para garantir *authenticity* e *confidentiality* em correio eletrônico. Foi desenvolvida especificamente para ser um padrão a ser usado na Internet.

A figura A.2 (fonte: [17]), apresentada mais à frente, ilustra a relação entre os algoritmos e as aplicações comentadas anteriormente.

A.5 Comunicação Segura

Para se fazer uma comunicação segura em uma rede de computadores, deve-se seguir quatro passos básicos:

1. Escolher um algoritmo que ofereça garantias de segurança;
2. Gerar a informação secreta que será usada com o algoritmo;
3. Utilizar métodos para a distribuição e o compartilhamento da informação secreta;
4. Usar um protocolo de comunicação adequado para estabelecer a comunicação entre as partes envolvidas, as quais fazem uso do algoritmo de segurança previamente escolhido.

Para aumentar ainda mais a segurança, é importante que em cada uma das partes envolvidas haja sistemas utilizando procedimentos de *log-in* e *screening logic*. O primeiro é baseado em senhas de proteção que são designadas para permitir acesso ao sistema apenas por parte de usuários autorizados. O segundo é usado para detectar e rejeitar invasores, vírus e outros ataques similares.

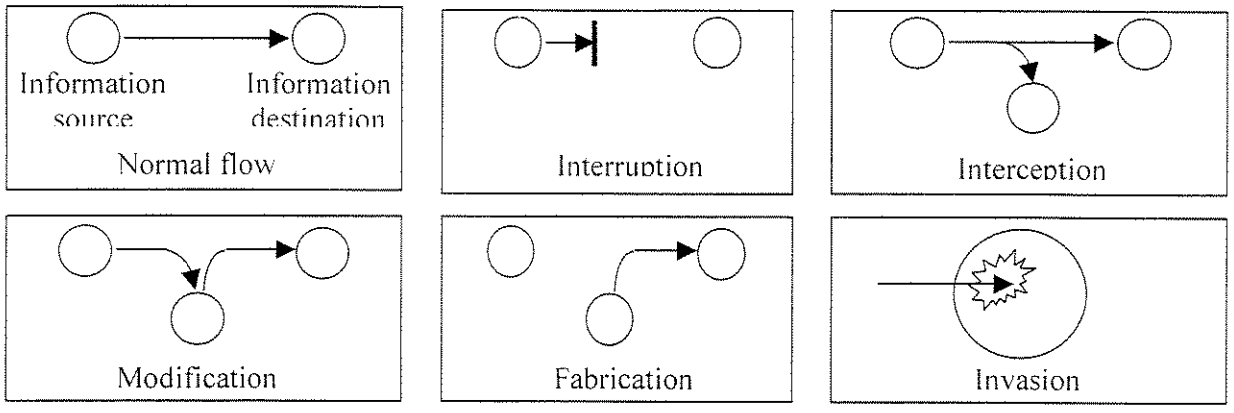

Figura A.1: Tipos de ataques.

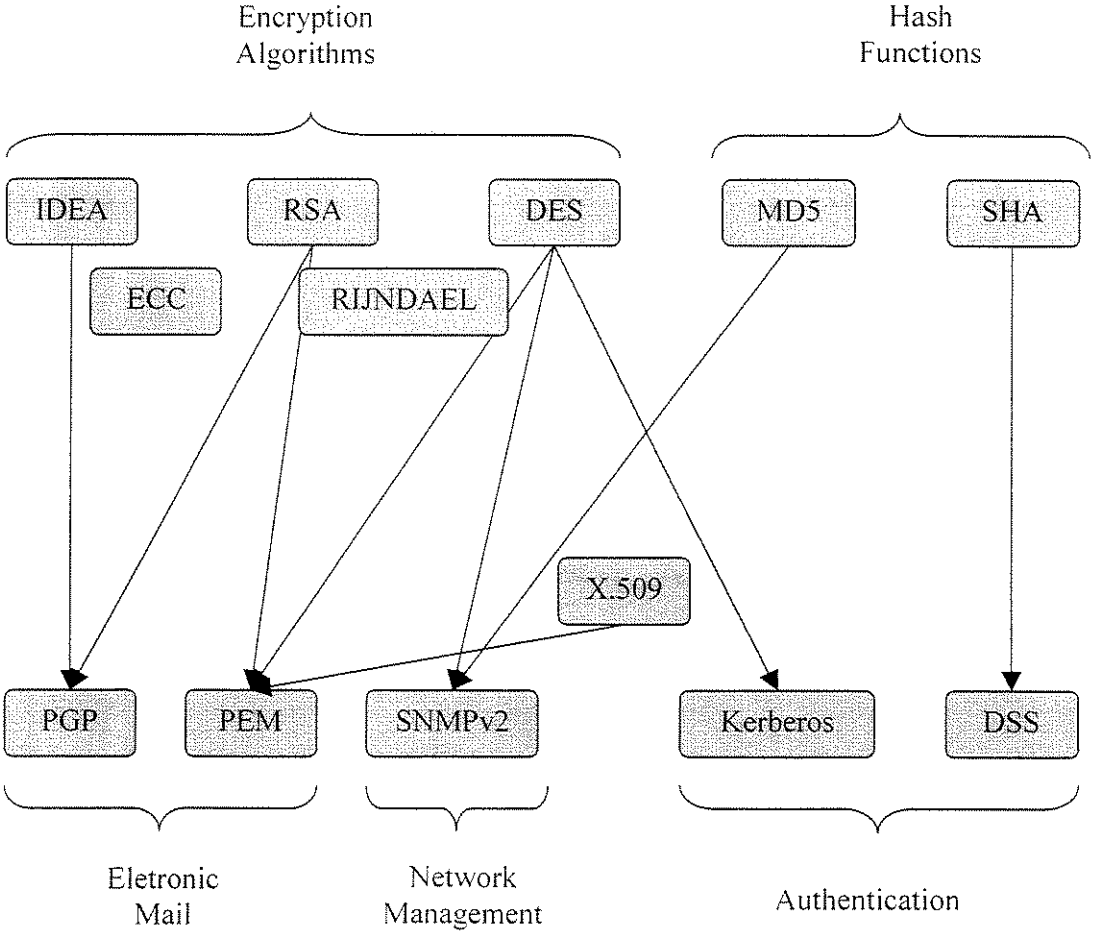

Figura A.2: Ferramentas de segurança.

A.6 Apresentação de um modelo de sistema criptográfico

Esta seção tem como objetivo apresentar um modelo de sistema criptográfico, o qual pode ser empregado para garantir a segurança de informações que trafegam por uma rede de computadores. Esse modelo será descrito através de vários passos. Mais adiante, neste mesmo apêndice, cada passo será associado a um estudo teórico mais profundo, visando uma compreensão mais abrangente do modelo. Criptografias assimétrica e simétrica, autenticação e assinaturas digitais serão discutidas em detalhes nas próximas seções.

O modelo de um sistema criptográfico (fonte: [17]) pode apresentar os seguintes passos:

1°. A => KDC: $E_{KU_{auth}} [ID_A \| ID_B]$;

2°. KDC => A: $E_{KR_{auth}} [ID_B \| E_{KU_B}]$;

3°. A => B: $E_{KU_B} [N_A \| ID_A]$;

4°. B => KDC: $E_{KU_{auth}} [ID_B \| ID_A \| N_A]$;

5°. KDC => B: $E_{KR_{auth}} [ID_A \| E_{KU_A}] \| E_{KU_B} \{ E_{KR_{auth}} [N_A \| ID_A \| ID_B] \}$;

6°. B => A: $E_{KU_A} \{ E_{KR_{auth}} [N_A \| ID_A \| ID_B] \| N_B \}$;

7°. A => B: $E_{K_S} [N_B]$;

8°. A => B: $E_{K_S} [M_1 \| N_A] \| H(M_1)$;

9°. B => A: $E_{K_S} [M_2 \| N_B] \| H(M_2)$.

A e B representam dois usuários[1] que necessitam trocar informações sigilosas, um com o outro, dentro de uma rede. O KDC (*Key Distribution Center*) representa uma entidade idônea intermediadora na comunicação entre A e B. Ele concentra em si um banco de dados contendo todas as chaves públicas dos usuários da rede, o qual será compartilhado entre eles mesmos. Isso evita que haja múltiplas cópias desse banco de dados na rede, isto

[1] Neste apêndice, a palavra usuário estará representando um aplicativo, uma pessoa, um dispositivo eletrônico, ou qualquer outro elemento que deseje estabelecer uma conexão segura através de uma rede.

é, uma para cada usuário. A rede precisa garantir que toda comunicação iniciada entre A e B siga os passos do modelo.

1°. Passo - A deseja iniciar sua troca de informações com B. Antes, porém, A precisa enviar uma mensagem para o KDC. A mensagem enviada por A ao KDC deve conter os números identificadores tanto da origem como do destino das mensagens que trafegarão na rede, isto é, o ID_A e o ID_B. Cada usuário em uma rede deve possuir seu ID particular e exclusivo, o qual será conhecido por todos os usuários dessa mesma rede. Resumidamente, o 1° passo pode ser expresso da seguinte forma: A => KDC: $E_{KU_{auth}}$ [ID_A || ID_B]. Note que a mensagem foi criptografada usando a chave pública $E_{KU_{auth}}$. O KDC é a única entidade capaz de decriptografar uma mensagem criptografada pela chave pública $E_{KU_{auth}}$. Por outro lado, qualquer usuário da rede pode criptografar uma mensagem fazendo uso da chave pública $E_{KU_{auth}}$ e envia-la ao KDC.

2°. Passo - O KDC envia uma mensagem de resposta para A, contendo o número de identificação de B, assim como a chave pública de B (E_{KU_B}). Resumidamente, o 2° passo pode ser expresso da seguinte forma: KDC => A: $E_{KR_{auth}}$ [ID_B || E_{KU_B}]. Note que a mensagem foi criptografada usando a chave privada $E_{KR_{auth}}$. O KDC é a única entidade capaz de criptografar uma mensagem usando a chave privada $E_{KR_{auth}}$. Por outro lado, qualquer usuário da rede pode decriptografar tal mensagem fazendo uso da chave pública $E_{KU_{auth}}$.

3°. Passo - A usa a chave pública de B (E_{KU_B}), recebida no passo anterior, para criptografar uma mensagem, a qual será enviada diretamente para B. Essa chave garante que B seja o único usuário da rede capaz de decriptografar a mensagem. Tal mensagem visa informar que A pretende estabelecer uma conexão lógica com B. Para tanto, ela deve conter o ID_A, necessário para que B possa identificar a origem da mensagem. Também deve estar presente o N_A, conhecido por nonce, que nada mais é que um número gerado aleatoriamente pelo próprio A. Toda vez que uma nova conexão é iniciada em uma rede, um novo par diferente de nonces, neste caso N_A e N_B, é gerado e colocado dentro das mensagens. Isso garante que as mensagens sejam reconhecidas como pertencentes àquela determinada conexão. Se uma mensagem contém um nonce não pertencente a aquele par,

isso significa que ela não pertence à conexão atual e por isso deve ser ignorada. Resumidamente, o 3° passo pode ser expresso da seguinte forma: A => B: E_{KU_B} [N_A || ID_A].

4°. Passo - B aceita trocar informações com A e, por isso, pretende enviar uma mensagem para A, a fim de confirmar a conexão. Antes, porém, B precisa enviar uma mensagem para o KDC. A mensagem enviada por B ao KDC deve conter os números identificadores tanto da origem como do destino das mensagens que trafegarão na rede, isto é, o ID_A e o ID_B. Além disso, ela deve conter também o N_A, que serve como um rótulo para identificar essa conexão. Resumidamente, o 4° passo pode ser expresso da seguinte forma: B => KDC: $E_{KU_{auth}}$ [ID_B || ID_A || N_A].

5°. Passo - O KDC envia uma mensagem de resposta para B, contendo os números de identificação de A e B (ID_A e ID_B), o nonce de A (N_A) e a chave pública de A (E_{KU_A}). Como o KDC é a única entidade capaz de criptografar uma mensagem usando a chave privada $E_{KR_{auth}}$, a informação $E_{KR_{auth}}$ [N_A || ID_A || ID_B] só pode ter sido gerada pelo próprio KDC. Isso garante a autenticidade da mensagem. A mesma informação está criptografada pela chave pública de B (E_{KU_B}). Essa chave garante que B seja o único usuário da rede capaz de decriptografar a informação. Isso impede que um outro usuário tente usar essa informação para estabelecer uma conexão fraudulenta com A. Resumidamente, o 5° passo pode ser expresso da seguinte forma: KDC => B: $E_{KR_{auth}}$ [ID_A || E_{KU_A}] || E_{KU_B} { $E_{KR_{auth}}$ [N_A || ID_A || ID_B] }. Note novamente que a mensagem foi criptografada usando a chave privada $E_{KR_{auth}}$. Como já foi dito anteriormente, o KDC é a única entidade capaz de criptografar uma mensagem usando a chave privada $E_{KR_{auth}}$. Por outro lado, qualquer usuário da rede pode decriptografar tal mensagem fazendo uso da chave pública $E_{KU_{auth}}$.

6°. Passo - B usa a chave pública de A (E_{KU_A}), recebida no passo anterior, para criptografar uma mensagem, a qual será enviada diretamente para A. Essa chave garante que A seja o único usuário da rede capaz de decriptografar a mensagem. Tal mensagem contém a informação $E_{KR_{auth}}$ [N_A || ID_A || ID_B] (igual ao passo anterior), além do nonce de B (N_B), o qual nada mais é que um número gerado aleatoriamente pelo próprio B e que garante que a mensagem seja reconhecida como pertencente àquela determinada conexão.

N_A e N_B existirão apenas enquanto durar a conexão lógica entre A e B. Após esse período, eles perderão sua utilidade. Resumidamente, o 6° passo pode ser expresso da seguinte forma: B => A: E_{KU_A} { $E_{KR_{auth}}$ [N_A || ID_A || ID_B] || N_B }.

7°. Passo – A e B calculam a chave secreta E_{K_S}, como será explicado nas seções subseqüentes. A usa a chave secreta E_{K_S} para criptografar uma mensagem, a qual será enviada diretamente para B. Essa chave garante que A e B sejam os únicos usuários da rede capazes de decriptografar a mensagem. Tal mensagem contém o nonce de B (N_B), o qual garante que a mensagem seja reconhecida como pertencente àquela determinada conexão. Resumidamente, o 7° passo pode ser expresso da seguinte forma: A => B: E_{K_S} [N_B].

8°. e 9°. Passos - estes passos resumem o restante da comunicação entre A e B. Daqui para diante, até o final da conexão, todos os passos apresentarão as mesmas características presentes nos passos 8° e 9°. A e B trocarão entre si as informações sigilosas (M). Justamente para que essas informações possam manter seu sigilo durante sua transmissão pela rede, é que se emprega um modelo de sistema criptográfico como esse que está sendo apresentado. Dessa forma, agora, toda mensagem poderá ser expressa da seguinte forma: A => B: E_{K_S} [M_1 || N_A] || $H(M_1)$ ou B => A: E_{K_S} [M_2 || N_B] || $H(M_2)$. Além de M, toda mensagem deverá conter um nonce, N_A ou N_B, o qual garantirá que a mensagem seja reconhecida como pertencente àquela determinada conexão. E, por fim, toda mensagem deverá apresentar a *Hash Function* H(M) (vide [17]), que pode ser usada para fazer uma assinatura digital para cada uma das mensagens transmitidas durante a conexão. A presença de H(M) permite detectar qualquer alteração feita sobre a mensagem original, conseguindo, com isso, impedir modificações fraudulentas das informações. Note que cada mensagem será criptografada usando a chave secreta E_{K_S}. Essa chave garante que A e B sejam os únicos usuários da rede capazes de criptografar e decriptografar as próximas mensagens.

A.7 Criptografia Simétrica

A criptografia simétrica é usada onde se necessita codificar a informação de forma rápida. A figura A.3 (fonte: [17]) apresenta um exemplo simplificado do uso da criptografia simétrica.

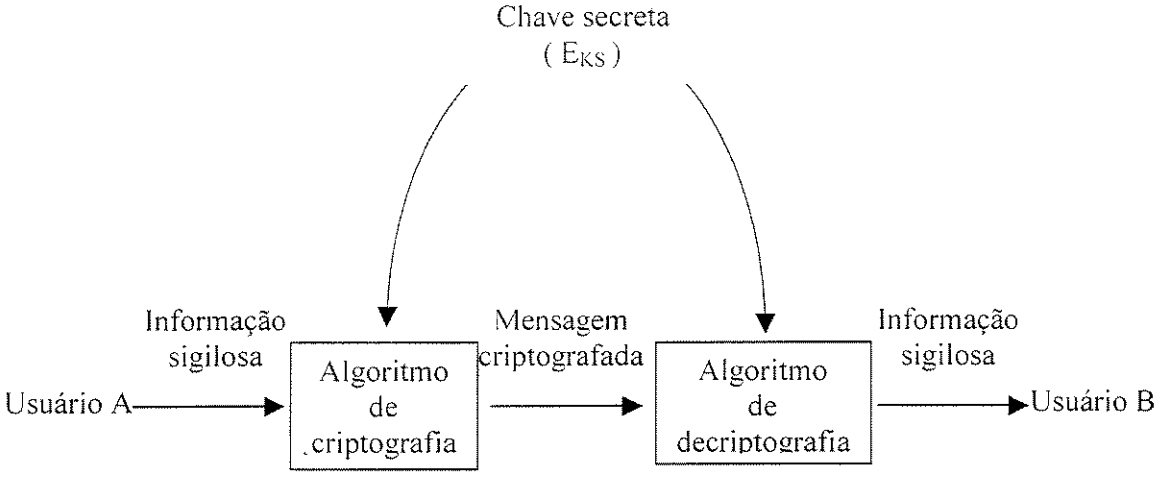

Figura A.3: Exemplo simplificado do uso da criptografia simétrica.

A informação sigilosa original é transformada em uma mensagem ininteligível, que é a mensagem criptografada. Para fazer a criptografia, emprega-se um algoritmo e uma chave.

O algoritmo baseia-se em técnicas de transposição (troca das posições das letras de uma mensagem) e substituição (troca de cada letra da mensagem, por outra letra diferente da original). Por exemplo, uma mensagem contendo apenas a palavra *rato*, após sofrer transposição poderia virar *troa* e após sofrer substituição poderia virar *kpiz*.

A chave secreta é um código independente da informação sigilosa e é usada para controlar o algoritmo. O algoritmo de criptografia gerará diferentes mensagens em função da chave secreta que estiver sendo utilizada naquele momento. Para cada chave distinta

usada com o algoritmo, haverá uma nova mensagem criptografada sendo produzida, mesmo que a informação sigilosa seja sempre a mesma.

Toda mensagem criptografada é transmitida a partir da origem, trafegando pela rede até chegar a um destino certo, onde é recebida. Após a recepção, a mensagem criptografada pode ser convertida para sua forma original, isto é, para a informação sigilosa inteligível. Isso só é possível graças ao algoritmo de decriptografia, o qual faz uso da mesma chave secreta usada para a criptografia da informação. Um exemplo de sistema criptográfico que apresenta essas características é o DES.

A.8 Criptografia Assimétrica

Por usar chaves distintas para cifrar e decifrar, a criptografia assimétrica possui uma série de vantagens sobre a criptografia simétrica. Entretanto, por ser baseada em funções matemáticas, possui baixo desempenho e acaba sendo usada apenas para cifrar pequenos volumes de informação, como chaves de algoritmos de criptografia simétrica, por exemplo.

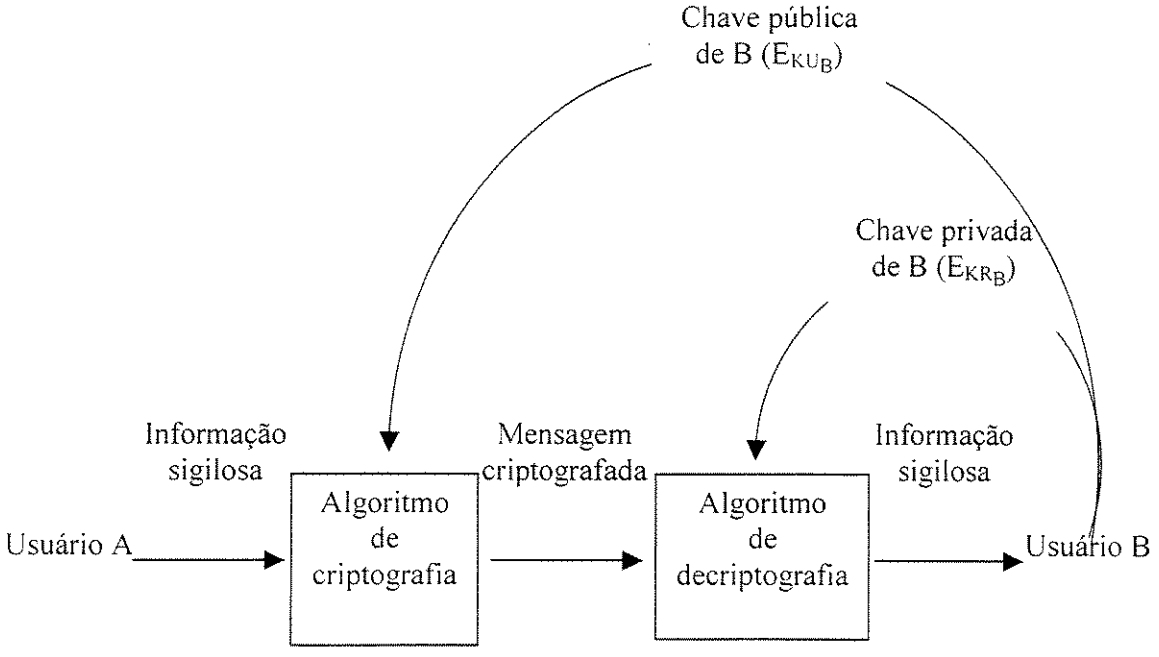

Figura A.4: Exemplo simplificado do uso da criptografia assimétrica.

A figura A.4 (fonte: [17]) apresenta um exemplo simplificado do uso da criptografia assimétrica.

Cada usuário da rede mantém um par de chaves para serem usadas respectivamente na criptografia e na decriptografia das informações sigilosas que serão transmitidas. Cada usuário publica uma de suas chaves através de um registro ou arquivo públicos (KDC). Esta é a chave pública (E_{KU}). A outra, que não deve ser publicada em hipótese alguma, é conhecida por chave privada (E_{KR}). Se um usuário A deseja enviar uma informação sigilosa para B, ele criptografa essa informação através de um algoritmo de criptografia, o qual faz uso da chave pública de B para gerar uma mensagem criptografada ininteligível e, em seguida, transmite essa mensagem pela rede com destino ao usuário B.

Quando B recebe essa mensagem, ele a decriptografa através de um algoritmo de decriptografia, o qual faz uso da chave privada de B (E_{KR_B}) para converter a mensagem criptografada novamente para a informação sigilosa inteligível. Nenhum outro usuário é capaz de decriptografar a mensagem, pois somente B detém essa chave privada.

Os algoritmos envolvidos nesse tipo de criptografia baseiam-se em funções e problemas matemáticos complexos, como por exemplo, as curvas elípticas e o problema do logaritmo discreto sobre curvas elípticas. Tal complexidade aumenta o tempo de execução dos algoritmos, o que penaliza as transmissões de informações *on-line*. Isso acaba limitando o uso dos algoritmos de criptografia assimétrica. Por isso, eles costumam ser empregados somente para realizarem a distribuição de chaves secretas (E_{KS}), apenas no início das conexões lógicas entre usuários de uma rede.

A.9 Autenticação

Qualquer mecanismo empregado para garantir autenticação de mensagem ou para gerar assinaturas digitais pode possuir fundamentalmente dois níveis.

No nível inferior, deve existir alguma função (H(M)) usada para produzir um valor que pode ser utilizado para autenticar a mensagem. O objetivo dessa função nada mais é do que gerar um número, o qual depende exclusivamente de cada uma das letras da mensagem.

Basta a mudança de apenas uma letra da mensagem para que a função gere um número diferente daquele previamente registrado junto à mensagem. A usuário que recebe a mensagem reconhece sua autenticidade apenas recalculando a função e comparando o novo resultado com aquele já presente na mensagem (H(M)). Se os valores forem iguais, a mensagem é autêntica, isto é, não sofreu nenhuma modificação desde o momento em que foi transmitida pelo usuário de origem. Graças a essa característica, é possível detectar qualquer alteração feita sobre a mensagem original, assim, sendo possível impedir modificações fraudulentas das informações presentes na mensagem.

Esse nível inferior aparece inserido em um nível superior, maior e mais abrangente que o primeiro. Aquele primeiro modelo de sistema criptográfico apresentado anteriormente neste apêndice pode ser considerado como um exemplo desse nível superior. Isso porque, de uma forma geral, o modelo comporta-se como um protocolo de autenticação, o qual permite ao receptor das mensagens verificar vários quesitos fundamentais para garantir a autenticidade das mensagens. Por exemplo, o modelo possibilita que os usuários envolvidos na conexão lógica possam identificar, com muita segurança, a origem e o destino de uma determinada mensagem, assim como se ela realmente pertence ou não àquela conexão e, ainda, se a mensagem sofreu ou não algum tipo de alteração de conteúdo, enquanto trafegava por algum ponto da rede.

A.10 Assinaturas Digitais

A autenticação de mensagem costuma proteger dois usuários envolvidos na troca de informações sigilosas contra a intromissão de um terceiro usuário. Entretanto, ela não protege os dois usuários, um contra o outro. É possível haver muitas formas de disputas entre os dois usuários, como, por exemplo, um deles negar ter recebido ou mesmo enviado uma determinada mensagem.

Em situações em que não se pode confiar totalmente no outro usuário envolvido na conexão lógica, é necessário empregar algum recurso adicional aos mecanismos de autenticação de mensagens. Esse recurso pode ser a assinatura digital, que, em termos de

funcionalidade, assemelha-se, por exemplo, a assinatura feita à mão em um cheque ou contrato.

Uma assinatura digital deve possuir algumas características importantes. Ela deve permitir verificar o autor, a data e a hora da assinatura. É necessário (e indicado) poder autenticar uma mensagem primeiro, para só então, em seguida, assiná-la. Deve haver a arbitragem de uma terceira parte para resolver a disputa. A assinatura deve ser um valor como o H(M), explicado anteriormente. Deve ser fácil produzir, reconhecer e verificar uma assinatura. É preciso que a assinatura permaneça imune e inalterada frente a qualquer tipo de tentativa de fraude. Finalmente, os usuários de origem e de destino de uma mensagem devem armazenar H(M).

Um exemplo de função que é bastante usada para garantir autenticidade e gerar assinaturas digitais, e que possui as características apresentadas anteriormente, é a Hash Function, empregada no primeiro modelo de sistema criptográfico apresentado anteriormente neste apêndice.

A.11 Curvas Elípticas

O apêndice A apresentou, até aqui, conceitos gerais a respeito de segurança e criptografia em redes de computadores. A partir desta seção, o apêndice A dará atenção a uma ferramenta específica, usada para prover segurança em redes de computadores: o algoritmo de criptografia assimétrica, baseado no problema do logaritmo discreto sobre curvas elípticas, que será chamado de algoritmo de ECC. Para entender melhor como esse algoritmo trabalha, é importante, primeiramente, saber o que é uma curva elíptica.

Segundo [7], uma curva elíptica pode ser definida como um conjunto de pontos na forma (X, Y) que satisfazem a equação: $y^2 = x^3 + ax + b.$ (A.1)

Os valores de a e b definem a curva, isto é, cada novo conjunto de valores a e b será responsável por criar uma curva elíptica diferente. A figura A.5 (fonte: [7]) mostra um exemplo de curva elíptica, para a = 1 e b = 0.

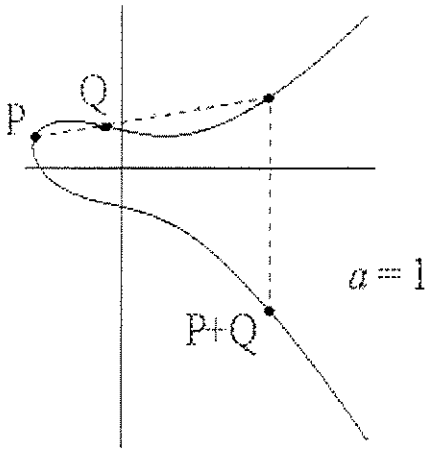

Figura A.5: Um exemplo de curva elíptica sobre números reais com adição de pontos.

Toda curva elíptica apresenta uma característica importante para seu uso em sistemas criptográficos: a formação de um grupo adição, o qual nada mais é do que um grupo de números satisfazendo as condições abaixo citadas (fonte: [7]), para os pontos P, Q e R dessa mesma curva elíptica:

1- Para os pontos $P(X_P, Y_P)$, $Q(X_Q, Y_Q)$ e $R(X_R, Y_R)$: $P + (Q + R) = (P + Q) + R$; (A.2)

2- Para um ponto O, representando o infinito: $P + O = P$; (A.3)

3- Para cada ponto P somado ao seu inverso: $P + (-P) = O$. (A.4)

Sobre essa curva elíptica, temos $P + Q = R$. Essa operação, como pode ser observado através da figura A.5, pode ser definida geometricamente, em se tratando de uma curva elíptica sobre números reais. Porém, os cálculos sobre números reais são lentos e imprecisos, devido aos erros de arredondamento nos computadores.

Sistemas criptográficos exigem uma aritmética rápida e precisa, que, ao contrário dos números reais, pode ser obtida por meio do uso de curvas elípticas sobre corpos finitos. Um corpo finito sobre um grupo adição representa um conjunto finito de elementos, no qual as três condições anteriormente citadas são satisfeitas.

Para sistemas de criptografia, baseados nas curvas elípticas, os corpos finitos podem ser de dois tipos: Fp e $F2^m$. Para os corpos finitos Fp ou corpos finitos primos, p representa

um número primo chamado de primo redutor. Para os corpos finitos $F2^m$ ou corpos finitos de característica 2, temos $p = 2^m$, para qualquer $m \geq 1$.

As relações existentes para curvas elípticas sobre números reais são mantidas para curvas sobre corpos finitos. Assim, uma curva elíptica sobre um corpo finito Fp pode ser definida como um conjunto de pontos na forma (X, Y) que satisfazem a equação (fonte: [7]):

$$(y^2) \bmod p = (x^3 + ax + b) \bmod p, \qquad (A.5)$$

onde m mod p retorna o resto da divisão inteira de m por p.

A figura A.6 (fonte: [7]) apresenta um exemplo de curva elíptica sobre um corpo finito F_{23}, definida por a = 1 e b = 0.

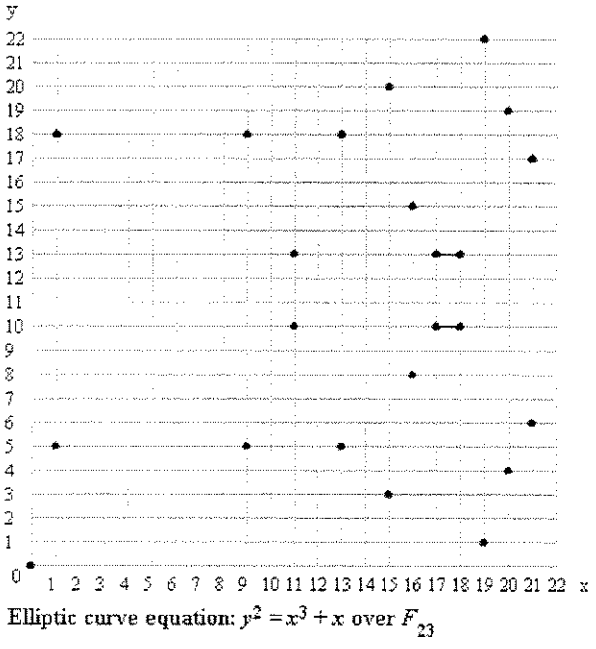

Figura A.6: Uma curva elíptica sobre um corpo finito F_{23}.

Computadores podem realizar a aritmética para corpos finitos $F2^m$ com maior eficiência do que para Fp, uma vez que $F2^m$ opera com *strings* de bits. Por isso, nosso trabalho considera apenas a implementação do algoritmo de ECC para $F2^m$.

Elementos de um corpo finito $F2^m$, na verdade, são *strings* de tamanho m bits. As regras da aritmética para $F2^m$ podem ser definidas tanto para a representação em base normal, como para a representação polinomial.

Uma curva elíptica sobre $F2^m$ pode ser definida como um conjunto de pontos na forma (X, Y) que satisfazem a equação (fonte: [7]):

$$y^2 + xy = x^3 + ax^2 + b, \qquad (A.6)$$

para b diferente de 0.

Para a representação polinomial, a equação anterior sofre uma pequena modificação (fonte: [7]):

$$(y^2 + xy) \bmod p = (x^3 + ax^2 + b) \bmod p, \qquad (A.7)$$

para b diferente de 0, $m \geq 1$ e $p = 2^m$.

A figura A.7 (fonte: [7]) apresenta um exemplo de curva elíptica sobre um corpo finito $F2^4$, definida por a = 3 e b = 1.

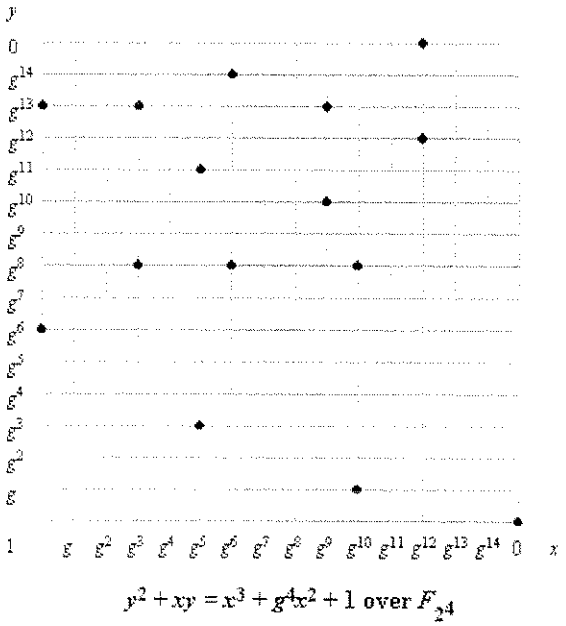

Figura A.7: Uma curva elíptica sobre um corpo finito $F2^4$.

Aqui, os elementos do corpo de uma curva elíptica são representados como polinômios. Assim, a redução na aritmética dessa representação é feita com a divisão do resultado por um polinômio irredutível (aquele que não é divisível por qualquer polinômio de grau menor).

A.12 Sistemas Criptográficos Baseados em Curvas Elípticas

As curvas elípticas são estudadas na matemática desde o século XVII. Entretanto, os primeiros registros do seu emprego em sistemas criptográficos datam de 1985. Os precursores desse tipo de uso para as curvas elípticas foram Neal Koblitz, na Universidade de Washington e, de forma independente, Victor Miller, na IBM de Yorktown Heights. De lá para cá, as pesquisas e o desenvolvimento de sistemas de segurança baseados em criptografia com curvas elípticas têm se intensificado cada vez mais. Hoje, eles representam os menores e mais rápidos sistemas de criptografia assimétrica. O sucesso dos sistemas criptográficos baseados nas curvas elípticas ocorre porque eles conseguem oferecer a

mesma segurança contra ataques proporcionada por outros algoritmos, como o RSA. A grande vantagem aqui é que, enquanto o RSA trabalha com chaves de 1024 bits, o algoritmo de ECC consegue o mesmo resultado com chaves de 160 bits, um número de bits significativamente menor.

A tabela A.1 (fonte: [12]) compara os tamanhos de chaves necessários para se garantir o mesmo nível de segurança através dos sistemas criptográficos ECC e RSA. Ela considera a quantidade de anos necessários para descobrir a chave usada com os algoritmos, utilizando-se um equipamento MIPS.

Tabela A1 - Tamanhos de chaves que garantem a mesma segurança para ECC e RSA.

Anos para descobrir a chave	RSA	ECC	RSA/ECC
	Tamanho da chave	Tamanho da chave	Tam.chave/tam.chave
10^4	512	106	5:1
10^8	768	132	6:1
10^{11}	1024	160	7:1
10^{20}	2048	210	10:1
10^{78}	2100	600	35:1

Segundo o grupo *Certicom Research,* em [8], "Desde a invenção da criptografia assimétrica, inúmeros sistemas criptográficos foram propostos. Cada um desses sistemas baseia sua segurança em um problema matemático de difícil solução. Nenhum foi provado intratável (impossível de resolver de uma maneira eficiente). Entretanto, acredita-se que sejam "intratáveis", porque anos de estudos intensivos, conduzidos por matemáticos e por cientistas da computação, falharam na tentativa de alcançar algoritmos eficientes para resolver esses problemas, de modo que, na prática, eles continuam intratáveis com a tecnologia computacional atual. Quanto maior for o tempo necessário para se desvendar uma chave, usando para isso o melhor algoritmo conhecido para um dado problema, mais seguro será o sistema criptográfico baseado nesse problema".

As curvas elípticas garantem o maior nível de segurança por bit dentre os algoritmos de criptografia assimétrica, devido à dificuldade para se achar a solução do problema sobre o qual elas são baseadas – o problema do logaritmo discreto sobre curvas elípticas

(ECDLP). É justamente a grande dificuldade em resolver esse problema que garante segurança através de uma chave relativamente pequena. Algoritmos, como o RSA, precisam de chaves maiores, porque são baseados na fatoração de inteiros.

Sistemas que usam criptografia assimétrica são tradicionalmente baseados em fatoração de inteiros e em logaritmos discretos sobre corpos finitos. Para o primeiro problema, existe um inteiro M, com tamanho de m bits, que é fatorável em dois números primos p e q. Em outras palavras, M = pq. O problema aqui é encontrar p e q. O segundo problema envolve corpos finitos (corpos de Galois GF( )), contendo um número primo de elementos ou 2^m elementos. Dado um elemento y pertencente a um desses corpos, o problema aqui está em encontrar um inteiro k, tal que $y = a^k$, onde a é um elemento primitivo fixo, pertencente ao corpo (y e k são representados por m bits). Os dois problemas descritos anteriormente são computacionalmente difíceis de serem resolvidos. Entretanto, há algoritmos sub-exponenciais para ambos os problemas.

Dado um ponto Q = kP, e assumindo que P e os parâmetros da curva são conhecidos, o problema do logaritmo discreto sobre curvas elípticas resume-se a encontrar o valor de k, o qual deve ter m bits de tamanho. A curva tem somente a estrutura aditiva, em oposição aos corpos que possuem tanto a aditiva como a multiplicativa. Isso é uma das razões pelas quais o problema do logaritmo discreto é mais difícil de ser resolvido quando aplicado sobre curvas elípticas do que quando aplicado sobre os outros tipos de corpos.

Como é possível notar, é muito difícil determinar k, dados Q e P, entretanto, é relativamente fácil calcular Q, dados k e P. Segundo Macedo, em [11], o cálculo do produto de um número escalar k pelo ponto P de uma curva elíptica constitui a operação essencial em ECC. Pode-se realizar essa operação somando-se o ponto P k vezes. Porém, esse método é lento para ser usado em ECC. O algoritmo empregado para se executar essa função é o *Double_and_Add* (vide [6]), que faz a redução no número de somas necessárias para o cálculo do produto. Esse algoritmo baseia-se na adição e na duplicação do ponto, conforme o estado dos bits no número k. Dessa forma, as duas operações necessárias para o cálculo de kP são a adição de dois pontos e a duplicação de um ponto, descritas anteriormente.

Os sistemas criptográficos baseados nas curvas elípticas se apóiam no problema do logaritmo discreto sobre curvas elípticas para empregarem um método de troca de chaves chamado de Diffie-Hellman (vide [17]). O exemplo a seguir ilustra como se dá a troca de chaves nesses sistemas criptográficos. Mais especificamente, o sistema criptográfico do exemplo é, na verdade, o mesmo modelo apresentado anteriormente neste mesmo apêndice. Só que agora, ele mostra em que momento o algoritmo baseado nas curvas elípticas e o método Diffie-Hellman são empregados:

1. A e B publicamente escolhem um corpo finito $F2^m$ e uma curva elíptica E, definida sobre esse corpo;
2. A e B publicamente escolhem um ponto fixo P, pertencente à curva elíptica E;
3. A gera um inteiro E_{KR_A}, aleatoriamente, e o mantém em segredo, pois ele representa sua chave privada. B faz o mesmo, ou seja, gera e mantém em sigilo E_{KR_B}, (sua chave privada);
4. A calcula $E_{KU_A} = E_{KR_A} P$, que pertence à curva E. B calcula $E_{KU_B} = E_{KR_B} P$, que pertence à curva E;
5. A e B, respectivamente, publicam E_{KU_A} e E_{KU_B} no KDC, para uma comunicação entre os próprios A e B em um instante futuro.

Supondo, agora, que A e B queiram estabelecer uma conexão e trocar informações entre si de forma segura.

6. A => KDC: $E_{KU_{auth}} [ID_A \| ID_B]$;
7. KDC => A: $E_{KR_{auth}} [ID_B \| E_{KU_B}]$;
8. A => B: $E_{KU_B} [N_A \| ID_A]$;
9. B => KDC: $E_{KU_{auth}} [ID_B \| ID_A \| N_A]$;
10. KDC => B: $E_{KR_{auth}} [ID_A \| E_{KU_A}] \| E_{KU_B} \{ E_{KR_{auth}} [N_A \| ID_A \| ID_B] \}$;
11. B => A: $E_{KU_A} \{ E_{KR_{auth}} [N_A \| ID_A \| ID_B] \| N_B \}$;
12. A utiliza E_{KU_B} e sua chave privada E_{KR_A} para calcular $E_{K_S} = E_{KR_A} E_{KU_B}$ => $E_{K_S} = E_{KR_A} (E_{KR_B} P)$ => $E_{K_S} = E_{KR_A} E_{KR_B} P$. Do outro lado da conexão, B utiliza E_{KU_A} e sua chave privada E_{KR_B} para calcular $E_{K_S} = E_{KR_B} E_{KU_A}$ => $E_{K_S} = E_{KR_B} (E_{KR_A} P)$ => $E_{K_S} = E_{KR_B} E_{KR_A} P$.

13. A => B: E_{K_S} [N_B];

14. A => B: E_{K_S} [M || N_A || H(M)];

15. B => A: E_{K_S} [M || N_B || H(M)].

Repare que ambos chegaram a um mesmo ponto E_{K_S}, pertencente à curva elíptica E, que pode ter parte de seus bits usados como chave secreta em um algoritmo de criptografia simétrica.

Segundo [12], uma vantagem oferecida pelo uso das curvas elípticas em sistemas criptográficos é que, mesmo quando todos os usuários estão trabalhando com um mesmo corpo finito, cada par de usuários pode selecionar uma diferente curva pertencente aquele corpo. Como conseqüência, o mesmo *hardware* pode ser usado para realizar aritmética relativa aquele corpo para todos os usuários, mas a curva pode ser trocada periodicamente, a fim de incrementar a segurança do sistema.

A.13 Comentários

Segurança em redes de computadores, criptografias simétrica e assimétrica, autenticação, assinaturas digitais, curvas elípticas e o problema do logaritmo discreto sobre curvas elípticas foram os assuntos abordados por este apêndice.

Apêndice B
FPGA

Este apêndice faz uma breve explicação sobre as características comuns às FPGAs existentes atualmente no mercado, incluindo comentários a respeito de seus elementos internos.

B.1 As FPGAs

Segundo a *Virtual Computer Corporation*, em [20], FPGA (*Field Programmable Gate Array*) é um tipo de dispositivo programável, isto é, uma classe de *chips* com propósito geral que podem ser configurados por uma variedade de aplicações (vide [apêndice E]). A memória PROM (*Programmable Read-Only Memory*) foi o primeiro representante dessa classe de *chips* a ser largamente utilizada. Embora pudesse ser gravada pelo usuário final, este somente poderia realizar essa tarefa uma única vez. A evolução dessa memória resultou na EPROM (*Erasable Programmable Read-Only Memory*), que podia ser desgravada através da exposição do *chip* a raios ultravioleta. Em seguida, ela podia ser regravada. Posteriormente, foi desenvolvida uma memória batizada com o nome EEPROM (*Eletrically Erasable Programmable Read-Only Memory*), que podia ser desgravada e regravada eletricamente muitas vezes. Mais tarde, essa classe de *chips* evoluiu para os chamados PLDs (*Programmable Logic Devices*), que foram desenvolvidos com o intuito de servirem de arquitetura base para a implementação de circuitos diversos, através da programação lógica criada pelo próprio usuário final.

Os PLDs possuem uma matriz de portas lógicas AND, conectada a uma matriz de portas lógicas OR. Uma sub-classe de PLDs, chamada de PAL (*Programmable Array Logic*), compreende os *chips* mais usados atualmente. A PAL caracteriza-se por apresentar um plano programável de portas lógicas AND, seguido por um plano fixo de portas lógicas OR.

As FPGAs, cujos elementos interconectados foram projetados para serem programados diretamente pelo usuário final, foram primeiramente apresentadas pela Xilinx, em 1985. Atualmente, o mercado disponibiliza quatro categorias principais de FPGAs: matriz simétrica, baseada em linha, PLD hierárquica e mar de portas lógicas, as quais diferenciam-se pela forma como são programadas e por suas interconexões, como pode ser observado na figura B.1 (fonte: [20]).

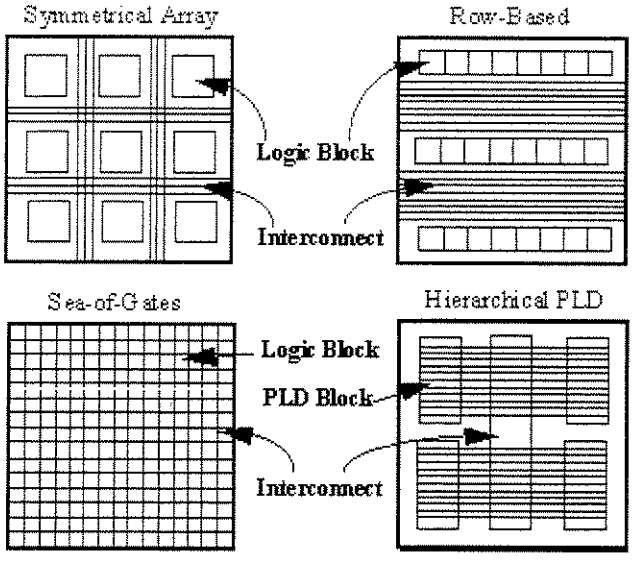

Figura B.1: As categorias de FPGAs.

A figura B.1 permite perceber que as quatro categorias de FPGAs possuem basicamente os mesmos elementos. A diferença entre elas fica por conta da maneira como esses elementos são interconectados.

Paralelamente, as FPGAs comercializadas são fabricadas utilizando-se uma das seguintes tecnologias: static RAM cells, anti-fusi, EPROM transistors e EEPROM transistors.

Static RAM – as conexões programáveis são compostas por transistores, portas lógicas ou multiplexadores, controlados por células SRAM. Pode ser rapidamente reprogramada, mas exige um número grande de componentes internos.

Anti-fuse – é comercializada em alta impedância, para posteriormente ser programada em baixa impedância ou *fused state*. Embora seja mais barata que as FPGAs static RAM, limitam-se por serem programadas uma única vez.

EPROM e EEPROM – já foram apresentadas no início desse apêndice.

As principais características dessas FPGAs são apresentadas na tabela B.1 (fonte: [20]):

Tabela B1 - Características das FPGAs comerciais.

Tecnologia	Volátil	Reprogramável	Área do Chip	R (ohm)	C (ff)
Static RAM	Sim	in-circuit	Grande	1 -2 K	10 - 20 ff
PLICE Anti-Fuse	Não	Não	anti-fuse ----- Pequena prog. trans.--- Grande	300 - 500	3 - 5 ff
ViaLink Anti-Fuse	Não	Não	anti-fuse ----- Pequena prog. trans.----Grande	50 - 60	3 - 5 ff
EPROM	Não	out of circuit	Pequena	2 - 4 k	10 -20 ff
EEPROM	Não	out of circuit	2x EPROM	2 - 4 k	10 -20 ff

Algumas FPGAs disponíveis comercialmente são apresentadas na tabela B.2 (fonte: [20]):

Tabela B2 - Algumas FPGAs comerciais

Fabricante	Arquitetura	Tipo de Bloco Lógico	Tecnologia Programável
Actel	Baseada em linha	Multiplexadores	anti-fuse
Altera	PLD hierárquica	Blocos de PLDs	EPROM
QuickLogic	Matriz simétrica	Multiplexadores	anti-fuse
Xilinx	Matriz simétrica	Look-up Table	Static RAM

As tabelas B.1 e B.2 revelam que diferentes fabricantes têm optado por distintas arquiteturas de FPGAs, cada uma contendo características bastante particulares em termos de área, resistência, capacitância, volatilidade, tipo de bloco lógico, tecnologia programável e capacidade de reprogramação.

B.1.1 FPGA Static RAM

Essa FPGA apresenta vantagens em termos de custo inicial, tempo para desenvolvimento de projetos e presença de erros, quando comparada às técnicas convencionais de gravação

de *chips* através de máscaras, como no caso dos ASICs. Sua programação é feita carregando-se dados de configuração para suas células internas de memória, via comunicação serial ou paralela, a partir de um PC ou estação de trabalho. Não há restrições quanto ao tipo de circuito que será gravado nela ou quanto à quantidade de vezes que ela pode ser desgravada e regravada. Atualmente, até mesmo a velocidade e o tamanho dos circuitos projetados para as FPGAs estão deixando de ser fatores limitantes para seu uso.

Os principais elementos configuráveis dessas FPGAs são os CLBs (*Configurable Logic Blocks*), os IOBs (*Input/Output Blocks*) e os recursos de interconexão. Todos eles são apresentados pela figura B.2 (fonte: [20]).

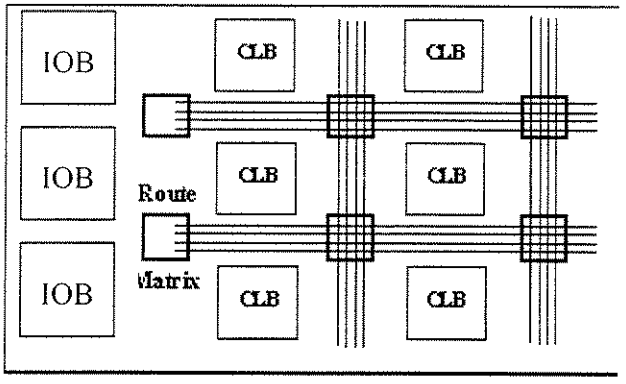

Figura B.2: Principais elementos configuráveis das FPGA Static RAM.

Os CLBs provêem os elementos funcionais para construir a lógica do usuário. Os IOBs provêem as interfaces entre os pinos do *chip* e as linhas de sinais internas. Os recursos de interconexão programáveis provêem os caminhos para a apropriada interconexão entre as entradas e saídas dos CLBs e dos IOBs. A configuração personalizada do *chip* é estabelecida pela programação interna das células da memória estática, que determina as funções lógicas e as conexões internas implementadas na FPGA.

O CLB implementa a maior parte da lógica de um projeto na FPGA, graças à flexibilidade e simetria de sua arquitetura. Cada CLB contém dois flip-flops e dois geradores de função, cada um com quatro entradas, como pode ser visto através da figura B.3 (fonte: [20]).

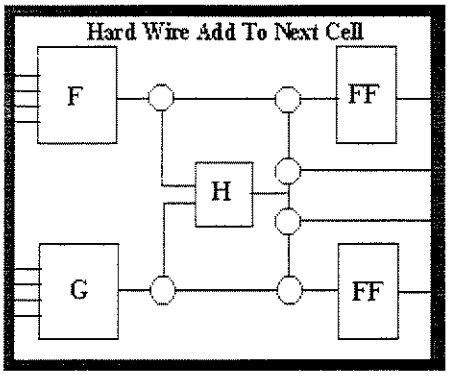

Figura B.3: Estrutura interna dos CLBs.

Uma descrição mais detalhada a respeito da estrutura interna das FPGAs pode ser encontrada em [20].

B.2 Comentários

As características comuns às FPGAs existentes atualmente no mercado, incluindo comentários a respeito de seus elementos internos foram os assuntos abordados por este apêndice.

Apêndice C
Sinais de Controle do Co-processador M1

Este apêndice apresenta uma lista contendo todos os sinais de controle usados pelo co-processador M1, incluindo o número e a função de cada um deles.

C.1 Sinais de Controle

```
CONTROL_SIGNAL000 = SWITCH_TEMP
CONTROL_SIGNAL001 = INCREMENT_CONTROL
CONTROL_SIGNAL002 = DATA_FROM_TEMP_TO_ALU_CONTROL
CONTROL_SIGNAL003 = DATA_FROM_BUS_TO_TEMP_CONTROL
CONTROL_SIGNAL004 = AND_OPERATION_CONTROL
CONTROL_SIGNAL005 = OR_OPERATION_CONTROL
CONTROL_SIGNAL006 = XOR_OPERATION_CONTROL
CONTROL_SIGNAL007 = ADDITION_OPERATION_CONTROL
CONTROL_SIGNAL008 = SET_THIRD_AUXILIARY
CONTROL_SIGNAL009 = ADDRESS_FROM_PC_TO_MAR_CONTROL
CONTROL_SIGNAL010 = INITIAL_VALUE_TO_COUNTER_FROM_BUS_CONTROL
CONTROL_SIGNAL011 = DATA_STORED_IN_COUNTER_OUTPUT_CONTROL
CONTROL_SIGNAL012 = RIPPLE_INPUT0
CONTROL_SIGNAL013 = CLEAR_COUNTER_CONTROL
CONTROL_SIGNAL014 = COUNTER_INCREMENT-DECREMENT_CONTROL
CONTROL_SIGNAL015 = DATA_FROM-TO_FIRST_AUX_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL016 = DATA_STORED_IN_FIRST_AUX_INPUT_CONTROL
CONTROL_SIGNAL017 = DATA_INPUT_IN_FIRST_AUX_CONTROL
CONTROL_SIGNAL018 = CLEAR_FIRST_AUX
CONTROL_SIGNAL019 = DATA_STORED_IN_FIRST_AUX0
CONTROL_SIGNAL020 = DATA_STORED_IN_THIRD_AUX_LEFT_CONTROL
CONTROL_SIGNAL021 = DATA_STORED_IN_THIRD_AUX_RIGHT_CONTROL
CONTROL_SIGNAL022 = DATA_STORED_IN_THIRD_AUX0
CONTROL_SIGNAL023 = DATA_STORED_IN_THIRD_AUX9
CONTROL_SIGNAL024 = CLEAR_SECOND_AUX
CONTROL_SIGNAL025 = DATA_STORED_IN_SECOND_AUX0
CONTROL_SIGNAL026 = DATA_STORED_IN_SECOND_AUX9
CONTROL_SIGNAL027 = DATA_FROM-TO_SECOND_AUX_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL028 = DATA_STORED_IN_SECOND_AUX_LEFT_CONTROL
CONTROL_SIGNAL029 = DATA_STORED_IN_SECOND_AUX_RIGHT_CONTROL
CONTROL_SIGNAL030 = DATA_INPUT_IN_SECOND_AUX_CONTROL
CONTROL_SIGNAL031 = DATA_INVERTED_FROM_ACC_TO_ALU_CONTROL
CONTROL_SIGNAL032 = DATA_FROM-TO_ACC_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL033 = CLEAR_THIRD_AUXILIARY
CONTROL_SIGNAL034 = DATA_STORED_IN_ACC_INPUT_CONTROL
CONTROL_SIGNAL035 = DATA_STORED_IN_ACC0
CONTROL_SIGNAL036 = DATA_INPUT_IN_ACC_CONTROL
```

```
CONTROL_SIGNAL037 = CLEAR_ACC
CONTROL_SIGNAL038 = DATA_FROM_ALU_TO_ACC_CONTROL
CONTROL_SIGNAL039 = ADDRESS_FROM_PC_COUNTER_TO_PC_REGISTER_CONTROL
CONTROL_SIGNAL040 = JUMP_FROM_BUS_TO_PC_JUMP_REGISTER_CONTROL
CONTROL_SIGNAL041 = PC_JUMP_ADDER_CONTROL
CONTROL_SIGNAL042 = DATA_STORED_IN_PC_COUNTER_OUTPUT_CONTROL
CONTROL_SIGNAL043 = INITIAL_VALUE_TO_PC_COUNTER_FROM_BUS_CONTROL
CONTROL_SIGNAL044 = PC_RIPPLE_INPUT
CONTROL_SIGNAL045 = CLEAR_PC_COUNTER_CONTROL
CONTROL_SIGNAL046 = PC_COUNTER_INCREMENT-DECREMENT_CONTROL
CONTROL_SIGNAL047 = DATA_INPUT_IN_THIRD_AUXILIARY_CONTROL
CONTROL_SIGNAL048 = DATA_INPUT-OUTPUT_THIRD_AUXILIARY_CONTROL
CONTROL_SIGNAL049 = COPROCESSOR_END_OPERATION
CONTROL_SIGNAL050 = CLEAR_FOURTH_AUX
CONTROL_SIGNAL051 = DATA_STORED_IN_FOURTH_AUX0
CONTROL_SIGNAL052 = DATA_STORED_IN_FOURTH_AUX9
CONTROL_SIGNAL053 = DATA_FROM-TO_FOURTH_AUX_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL054 = DATA_STORED_IN_FOURTH_AUX_LEFT_CONTROL
CONTROL_SIGNAL055 = DATA_STORED_IN_FOURTH_AUX_RIGHT_CONTROL
CONTROL_SIGNAL056 = DATA_INPUT_IN_FOURTH_AUX_CONTROL
CONTROL_SIGNAL057 = MEMORY ACCESS
CONTROL_SIGNAL058 = CLEAR_FIFTH_AUX
CONTROL_SIGNAL059 = DATA_STORED_IN_FIFTH_AUX0
CONTROL_SIGNAL060 = DATA_STORED_IN_FIFTH_AUX9
CONTROL_SIGNAL061 = DATA_FROM-TO_FIFTH_AUX_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL062 = DATA_STORED_IN_FIFTH_AUX_LEFT_CONTROL
CONTROL_SIGNAL063 = DATA_STORED_IN_FIFTH_AUX_RIGHT_CONTROL
CONTROL_SIGNAL064 = DATA_INPUT_IN_FIFTH_AUX_CONTROL
CONTROL_SIGNAL065 = READ/WRITE_MEMORY
CONTROL_SIGNAL066 = ADDRESS_BUFFER2
CONTROL_SIGNAL067 = ADDRESS_BUFFER1
CONTROL_SIGNAL068 = ADDRESS_BUFFER0
CONTROL_SIGNAL069 = INITIAL_VALUE_TO_COUNTER_AUX_ONE_FROM_BUS_CONTROL
CONTROL_SIGNAL070 = DATA_STORED_IN_COUNTER_AUX_ONE_OUTPUT_CONTROL
CONTROL_SIGNAL071 = RIPPLE_INPUT0
CONTROL_SIGNAL072 = CLEAR_COUNTER_AUX_ONE_CONTROL
CONTROL_SIGNAL073 = COUNTER_AUX_ONE_INCREMENT-DECREMENT_CONTROL
CONTROL_SIGNAL074 = CLEAR_SIXTH_AUX
CONTROL_SIGNAL075 = DATA_STORED_IN_SIXTH_AUX0
CONTROL_SIGNAL076 = DATA_STORED_IN_SIXTH_AUX9
CONTROL_SIGNAL077 = DATA_FROM-TO_SIXTH_AUX_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL078 = DATA_STORED_IN_SIXTH_AUX_LEFT_CONTROL
CONTROL_SIGNAL079 = DATA_STORED_IN_SIXTH_AUX_RIGHT_CONTROL
CONTROL_SIGNAL080 = DATA_INPUT_IN_SIXTH_AUX_CONTROL
CONTROL_SIGNAL081 = SIGNAL_TEST_ONE_FROM_CU_TO_PSW
CONTROL_SIGNAL082 = SIGNAL_TEST_TWO_FROM_CU_TO_PSW
CONTROL_SIGNAL083 = SIGNAL_TEST_THREE_FROM_CU_TO_PSW
CONTROL_SIGNAL084 = SIGNAL_TEST_FOUR_FROM_CU_TO_PSW
CONTROL_SIGNAL085 = SIGNAL_COMP_FROM_LESS_THAN_TO_PSW
CONTROL_SIGNAL086 = DATA_TEMP_INVERTED
CONTROL_SIGNAL087 = CLEAR_TEMP
CONTROL_SIGNAL088 = ALU_MASK_CONTROL
CONTROL_SIGNAL089 = CLEAR_PK
CONTROL_SIGNAL090 = DATA_STORED_IN_PK0
CONTROL_SIGNAL091 = DATA_STORED_IN_PK9
CONTROL_SIGNAL092 = DATA_FROM-TO_PK_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL093 = DATA_STORED_IN_PK_LEFT_CONTROL
```

```
CONTROL_SIGNAL094 = DATA_STORED_IN_PK_RIGHT_CONTROL
CONTROL_SIGNAL095 = DATA_INPUT_IN_PK_CONTROL
CONTROL_SIGNAL096 = CLEAR_PK_ONE
CONTROL_SIGNAL097 = DATA_STORED_IN_PK_ONE0
CONTROL_SIGNAL098 = DATA_STORED_IN_PK_ONE9
CONTROL_SIGNAL099 = DATA_FROM-TO_PK_ONE_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL100 = DATA_STORED_IN_PK_ONE_LEFT_CONTROL
CONTROL_SIGNAL101 = DATA_STORED_IN_PK_ONE_RIGHT_CONTROL
CONTROL_SIGNAL102 = DATA_INPUT_IN_PK_ONE_CONTROL
CONTROL_SIGNAL103 = CLEAR_PK_TWO
CONTROL_SIGNAL104 = DATA_STORED_IN_PK_TWO0
CONTROL_SIGNAL105 = DATA_STORED_IN_PK_TWO9
CONTROL_SIGNAL106 = DATA_FROM-TO_PK_TWO_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL107 = DATA_STORED_IN_PK_TWO_LEFT_CONTROL
CONTROL_SIGNAL108 = DATA_STORED_IN_PK_TWO_RIGHT_CONTROL
CONTROL_SIGNAL109 = DATA_INPUT_IN_PK_TWO_CONTROL
CONTROL_SIGNAL110 = CLEAR_RK
CONTROL_SIGNAL111 = DATA_STORED_IN_RK0
CONTROL_SIGNAL112 = DATA_STORED_IN_RK9
CONTROL_SIGNAL113 = DATA_FROM-TO_RK_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL114 = DATA_STORED_IN_RK_LEFT_CONTROL
CONTROL_SIGNAL115 = DATA_STORED_IN_RK_RIGHT_CONTROL
CONTROL_SIGNAL116 = DATA_INPUT_IN_RK_CONTROL
CONTROL_SIGNAL117 = CLEAR_RK_ONE
CONTROL_SIGNAL118 = DATA_STORED_IN_RK_ONE0
CONTROL_SIGNAL119 = DATA_STORED_IN_RK_ONE9
CONTROL_SIGNAL120 = DATA_FROM-TO_RK_ONE_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL121 = DATA_STORED_IN_RK_ONE_LEFT_CONTROL
CONTROL_SIGNAL122 = DATA_STORED_IN_RK_ONE_RIGHT_CONTROL
CONTROL_SIGNAL123 = DATA_INPUT_IN_RK_ONE_CONTROL
CONTROL_SIGNAL124 = CLEAR_RK_TWO
CONTROL_SIGNAL125 = DATA_STORED_IN_RK_TWO0
CONTROL_SIGNAL126 = DATA_STORED_IN_RK_TWO9
CONTROL_SIGNAL127 = DATA_FROM-TO_RK_TWO_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL128 = DATA_STORED_IN_RK_TWO_LEFT_CONTROL
CONTROL_SIGNAL129 = DATA_STORED_IN_RK_TWO_RIGHT_CONTROL
CONTROL_SIGNAL130 = DATA_INPUT_IN_RK_TWO_CONTROL
CONTROL_SIGNAL131 = CLEAR_QK
CONTROL_SIGNAL132 = DATA_STORED_IN_QK0
CONTROL_SIGNAL133 = DATA_STORED_IN_QK9
CONTROL_SIGNAL134 = DATA_FROM-TO_QK_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL135 = DATA_STORED_IN_QK_LEFT_CONTROL
CONTROL_SIGNAL136 = DATA_STORED_IN_QK_RIGHT_CONTROL
CONTROL_SIGNAL137 = DATA_INPUT_IN_QK_CONTROL
CONTROL_SIGNAL138 = CLEAR_QK_ONE
CONTROL_SIGNAL139 = DATA_STORED_IN_QK_ONE0
CONTROL_SIGNAL140 = DATA_STORED_IN_QK_ONE9
CONTROL_SIGNAL141 = DATA_FROM-TO_QK_ONE_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL142 = DATA_STORED_IN_QK_ONE_LEFT_CONTROL
CONTROL_SIGNAL143 = DATA_STORED_IN_QK_ONE_RIGHT_CONTROL
CONTROL_SIGNAL144 = DATA_INPUT_IN_QK_ONE_CONTROL
CONTROL_SIGNAL145 = CLEAR_QK_TWO
CONTROL_SIGNAL146 = DATA_STORED_IN_QK_TWO0
CONTROL_SIGNAL147 = DATA_STORED_IN_QK_TWO9
CONTROL_SIGNAL148 = DATA_FROM-TO_QK_TWO_TO-FROM_BUS_CONTROL
CONTROL_SIGNAL149 = DATA_STORED_IN_QK_TWO_LEFT_CONTROL
CONTROL_SIGNAL150 = DATA_STORED_IN_QK_TWO_RIGHT_CONTROL
```

```
CONTROL_SIGNAL151 = DATA_INPUT_IN_QK_TWO_CONTROL
CONTROL_SIGNAL152 = CLEAR_TEST_ONE
CONTROL_SIGNAL153 = CLEAR_TEST_TWO
CONTROL_SIGNAL154 = CLEAR_TEST_THREE
CONTROL_SIGNAL155 = CLEAR_TEST_FOUR
CONTROL_SIGNAL156 = CLEAR_COMP
CONTROL_SIGNAL157 = CLEAR_COPROCESSOR_END_OPERATION
CONTROL_SIGNAL158 = DATA_STORED_IN_FIRST_AUX_RIGHT_CONTROL
CONTROL_SIGNAL159 = DATA_STORED_IN_FIRST_AUX9
CONTROL_SIGNAL160 = CLEAR_PSW
CONTROL_SIGNAL161 = CLEAR_PSW_LEFT_FOURTH_AUX
CONTROL_SIGNAL162 = MUL_ENABLE
CONTROL_SIGNAL163 = CLEAR_MASK
CONTROL_SIGNAL164 = MASK_ENABLE
CONTROL_SIGNAL[182..165] = FREE
```

Os dezoito últimos sinais do co-processador M1, isto é, do CONTROL_SIGNAL165 até o CONTROL_SIGNAL182, estão reservados para uso futuro.

C.2 Comentários

Uma lista contendo todos os sinais de controle usados pelo co-processador M1, incluindo o número e a função de cada um deles foi apresentada neste apêndice.

Apêndice D
VHDL

Este apêndice permite conhecer as principais características da linguagem de descrição de *hardware*, usada no projeto do co-processador M1: o VHDL.

D.1 A Linguagem VHDL

Segundo Douglas Perry [15], "VHDL é uma linguagem padronizada para a indústria, usada para descrever *hardware* de um nível abstrato para um nível concreto [...]. Ela está sendo rapidamente abraçada como um meio de comunicação universal de projetos".

A grande dificuldade dos projetistas de *hardware* era ter que produzir circuitos integrados complexos, tendo em mãos apenas ferramentas baseadas no nível de portas lógicas. Circuitos VLSI representavam um desafio extremamente difícil para eles. A necessidade da criação de ferramentas que permitissem uma descrição de *hardware* facilitada e eficiente era evidente.

A linguagem de descrição de *hardware* VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) foi desenvolvida pelo DoD (Departamento de Defesa dos Estados Unidos), durante as décadas de setenta e oitenta, com o objetivo de ser usada como uma ferramenta auxiliar para produzir a nova geração de circuitos integrados, levando em consideração as limitações tecnológicas de cada etapa do projeto e manufatura desses circuitos.

Seu uso inicial limitou-se à documentação e simulação de circuitos, mas logo foi integrada a ferramentas de síntese para descrever *hardware*.

Uma de suas maiores vantagens era que através dela os projetistas de *hardware* poderiam descrever e sintetizar circuitos complexos com maior facilidade, quando

comparada aos métodos tradicionais de *design*. Uma outra grande vantagem era que ela poderia ser usada como um padrão de linguagem de descrição de *hardware*, possibilitando o compartilhamento e reutilização de projetos entre engenheiros de forma padronizada.

Foi essa última característica que levou o IEEE a propor uma versão aprimorada do VHDL como padrão de linguagem de descrição de *hardware* em dezembro de 1987 – o IEEE1076.

Segundo Gajski, em [9], no VHDL, a "entidade" é o nível mais básico de abstração para o *hardware*. Partes de um projeto, que podem ser bem definidas com base nas funções específicas que exercem e em suas entradas e saídas particulares, são identificadas e representadas por essas entidades. As atividades realizadas por essas entidades podem ser descritas através de estruturas, linhas de comandos, fluxos de dados, ou por meio da combinação desses elementos.

A linguagem suporta hierarquia estrutural através de instancias de componentes e blocos hierárquicos. A seguir, será apresentada a especificação em VHDL de uma entidade representada por um contador decimal de 1 digito, utilizando-se a descrição estrutural (fonte: [9]).

```
entity Counter_E is
    port (clk: in bit; cnt: out integer);
end Counter_E;
architecture Counter_struct of Counter_E is
    component Reg_E
            port (d: in integer; clk: in bit; o: out integer; clear: in
bit);
    end component;
    component Add_E
            port (a, b: in integer; o: out integer;);
    end component;
    component Cmp_E
```

```
            port (i0, i1: in integer; o: out bit;);
    end component;
    .....
signal one: integer := 1;
signal nine: integer := 9;
signal cnt_in, cnt_out, add_out: integer;
signal clear: bit;
    .....
begin
    Conreg: Reg_E
            port map (cnt_in, clk, cnt_out, clear);
    Adder: Add_E
            port map (cnt_out, one, add_out);
    Comparator: Cmp_E
            port map (nine, cnt_in, clear);
    .....
    cnt <= cnt_out;
end Counter_struct;
```

Clk e cnt representam os dois *ports* externos do contador. Este tem seu valor incrementado cada vez que o sinal clk muda de 0 para 1. Sendo um contador decimal de apenas 1 dígito, seus valores variam de 0 a 9. Se o contador estiver indicando o valor 9 em sua saída cnt e, o clk muda de 0 para 1, o contador voltará a registrar 0, reiniciando sua contagem progressiva. A declaração *architecture* compreende não só o nome dos componentes do contador (Reg_E, add_E e comp_E), como também os seus respectivos *ports*. As instâncias desses componentes (Conreg, Adder e Comparator) aparecem no corpo de *architecture*. As associações entre os *ports* dos componentes especificam a interligação entre os mesmos. Por exemplo, o mapeamento do sinal cnt_out para os *ports* o e a de Conreg e de Adder, respectivamente, especifica que a saída de Conreg está ligada a uma entrada de Adder.

Outra maneira de especificar o contador decimal de 1 dígito através de VHDL seria usando a descrição comportamental, como é mostrado abaixo (fonte: [9]):

```
entity Counter_E is
    port (clk: in bit; cnt: out integer);
end Counter_E;
architecture Counter_beh of Counter_E is
begin
    process
            variable convar: integer := 0;
    begin
            cnt <= convar;
            wait until (clk = '1') and not(clk'stable);
            if (convar = 9) then
                    convar := 0;
            else
                    convar := convar + 1;
            end if;
    end process;
end Counter_beh;
```

O código apresenta um processo composto por uma seqüência de comandos, como em uma linguagem de programação do tipo ADA. VHDL oferece também uma variedade de tipos de dados, como por exemplo: *integer, real, enumeration, physical, array, record* e *pointer*.

Hierarquia comportamental é suportada pela linguagem, mas em apenas dois níveis: o dos processos e o das *procedures*. No primeiro nível, a especificação pode ser feita através

de processos com execução concorrente (concorrência em nível de tarefa). Já no segundo nível, a especificação é dada por meio da seqüência de *procedures* que compõem um processo.

Um aspecto interessante dentro da comunicação entre processos em VHDL pode ocorrer através do uso de sinais compartilhados entre esses processos. Todos eles podem emiti-los e recebê-los, imitando um modelo de memória compartilhada.

A sincronização em VHDL pode ser alcançada por meio do comando wait ou por listas sensíveis de processos.

No primeiro caso, um comando wait pode suspender um processo até que a ocorrência de uma condição ou de um evento (mudança do valor de um sinal específico) seja detectada. Com o comando wait on x,y until (start = '1'), por exemplo, a execução de um processo só continuaria após x ou y detectarem um evento ou start assumir o valor 1.

No segundo caso, um processo somente retoma sua execução, quando ocorre um evento sobre pelo menos um dos sinais que compõem sua lista sensível.

O exemplo a seguir mostra um processo com sua lista sensível. Enquanto não houver mudança nos valores de um dos sinais, start e x, o processo P permanecerá suspenso. Esses recursos permitem aos projetistas sincronizar a execução entre os processos que compartilham esses sinais.

```
P: process (start, x)
  begin
  .....
  end process;
```

(fonte: [9])

Tempo, em VHDL, é especificado pela cláusula after, como em a $\leq$ 2 after 20ns, comando que somente atribuirá o valor 2 ao sinal a, após passados 20ns. Um *timeout* também é possível em VHDL, basta seguir o exemplo: wait on start for 100ns. Esse

comando define que o processo estará suspenso até que ocorra uma mudança no sinal start. Mesmo não ocorrendo qualquer evento sobre start, o processo voltará à execução após 100ns. A hora do sistema pode ser acessada em VHDL usando-se a expressão now.

Uma terceira forma de especificar o contador, por meio de VHDL, usa o estilo de descrição de *hardware* conhecido por *dataflow*, segundo pode ser visto a seguir (fonte: [9]):

```
entity Counter_E is
    port (clk: in bit; cnt: out integer);
end Counter_E;
architecture Counter_dflow of Counter_E is
    signal consig: integer := 0;
begin
    block ((clk = '1') and not(clk'stable))
    begin
            consig <= guarded 0 when (consig = 9)
                    else consig + 1;
    end block;
    cnt <= consig;
end Counter_dflow;
```

A possibilidade de atribuir valores a sinais de forma concorrente em VHDL, permite aos projetistas expressar comportamentos do tipo *dataflow*, como no caso da terceira forma em que o contador foi especificado.

Além das variáveis, velhas conhecidas das linguagens de alto nível, VHDL apresenta também a idéia de sinais. Estes apresentam concorrência em nível de comandos, como pode ser observado a seguir (fonte: [9]):

```
a<=b;
b<=a;
```

```
wait on a;
```

O valor de b é apresentado em a e o valor de a é apresentado em b simultaneamente, pois esses comandos são executados em paralelo.

D.2 Comentários

As principais características da linguagem de descrição de *hardware* VHDL, usada no projeto do co-processador M1, foram apresentadas neste apêndice.

Apêndice E
Quartus II

Este apêndice permite conhecer as principais características da ferramenta de engenharia auxiliada por computador, usada no projeto do co-processador M1: o Quartus II.

E.1 O Quartus II

Segundo Douglas Perry, em [15], "Hoje, quando um projetista desenvolve um novo componente de *hardware*, este é provavelmente projetado através de um *software* de Engenharia Auxiliada por Computador (CAE)".

A ferramenta CAE Quartus II foi escolhida para o projeto do co-processador M1, porque ela é da mesma empresa fornecedora da FPGA do co-processador M1, a Altera.

Quartus II é a quarta geração de ferramentas para desenvolvimento de projetos de *hardware* produzidas pela Altera. Seu principal objetivo é auxiliar engenheiros durante o desenvolvimento de circuitos VLSI sobre PLDs (*Programmable Logic Devices*).

Abaixo, segue o resumo de suas principais características, conforme pode ser encontrado em [1].

Workgroup Computing – gerenciamento de arquivos e controle de revisão de projetos globais permitem que vários engenheiros trabalhem ao mesmo tempo em um mesmo projeto.

Incremental Recompilation – o compilador age somente sobre a parte modificada do projeto, diminuindo o tempo total de compilação.

Advanced Synthesis – provê um mapeamento otimizado do projeto para um PLD.

Multiple Platforms – é compatível com os sistemas operacionais Windows e UNIX.

Full Integration – permite integração com outras ferramentas de auxílio a projetos de *hardware*.

Modular Tools – diferentes modos de entrada, compilação, verificação e programação de dispositivos tornam possível a personalização do ambiente de desenvolvimento do projeto.

Hardware Description Languages (HDLs) – várias linguagens de descrição de *hardware*, como o VHDL, AHDL e Verilog, são suportadas pelo *software*.

Internet-Enabled Support – soluções e atualizações do *software* podem ser conseguidas diretamente da Altera via Internet.Também é possível submeter serviços para aplicações da Altera disponíveis na Internet e monitorar seu progresso on-line.

MegaCore Functions – compreende uma biblioteca de funções diversas, escritas em alguma HDL. Está disponível para ser usada pelos projetistas, a fim de simplificar e diminuir o tempo do projeto. Representam dispositivos simples que podem ser introduzidos em projetos de circuitos mais complexos.

Integrated Logic Analysis Functionality – permite reduzir significativamente o tempo de verificação.

O processo de desenvolvimento de um projeto, utilizando o *software* Quartus II, é composto por quatro fases: entrada, compilação, verificação e programação do dispositivo.

E.1.1 Entrada

A entrada refere-se a quais são os diferentes componentes de um circuito e como são interconectados para formá-lo.

Através do *software* Quartus II, pode-se criar uma hierarquia de componentes integrados. Isso permite que erros sejam localizados e destacados no próprio projeto durante a compilação, a simulação e a análise de tempo. O projetista pode sair de um componente e entrar em outro, dentro do mesmo projeto, independentemente se o componente foi projetado através de esquemático ou HDL, assim também como do número de níveis hierárquicos do projeto.

Os componentes que integram um circuito podem ser adquiridos a partir de bibliotecas de funções disponibilizadas pelo Quartus II, como as Megafunctions, as LPMs, entre outras, ou criadas pelo próprio engenheiro, por meio de esquemático ou alguma HDL.

As LPM functions representam uma biblioteca de funções padronizada para a indústria. Disponibilizam componentes dos tipos: memória RAM, contadores, somadores, multiplexadores, etc., que podem ser integrados para formarem dispositivos mais complexos.

As Megafunctions oferecem componentes mais elaborados e especializados, projetados para PCI (*Peripheral Component Interconnect*), interfaces de barramento, processamento digital de sinais e outras aplicações, como a de comunicação.

As funções de ambas as bibliotecas podem ser instanciadas e integradas dentro dos projetos e simuladas com o *software* Quartus II, a fim de produzir circuitos integrados mais complexos. São mais de 300 funções oferecidas pelo *software*, incluindo as bibliotecas acima citadas.

O editor de blocos do Quartus II permite criar projetos gráficos. Através dele o projetista pode criar e editar um projeto contendo combinações de diagramas de blocos, Megafunctions e símbolos elementares. Símbolos podem ser criados e modificados para ganharem uma aparência mais personalizada.

Os blocos funcionais podem ser criados nos primeiros estágios do desenvolvimento do projeto, a fim de representarem níveis mais elementares de blocos. Esses blocos podem ser interconectados por meio de linhas que conectam sinais comuns entre os blocos.

O editor de textos da Quartus II suporta a entrada e a edição de projetos descritos em VHDL, Verilog e AHDL. Essas HDLs permitem a implementação de máquinas de estado, tabelas verdade, lógica condicional, equações booleanas e operações como adição, subtração e comparação. Esse editor também permite manipular funções LPM descritas em algumas HDLs.

Outras duas aplicações do Quartus II, que contribuem para auxiliar o desenvolvimento de um projeto nessa faze inicial, são o Floorplan Editor e o Navegador do projeto.

Através do Floorplan Editor, células lógicas e pinos de entrada e saída podem ser programados, graças à imagem gráfica, em alto nível ou detalhada, de cada dispositivo usado na compilação. O projetista pode visualizar e modificar as células lógicas e pinos programados antes e depois da fase de compilação.

O Floorplan Editor provê um código de cores pré-estabelecido que permite identificar no dispositivo as posições relativas às células lógicas usadas ou não, às informações do tipo fan-in e fan-out e às características específicas da arquitetura. Qualquer um desses elementos pode ser re-arranjado sobre o Flooplan, bastando para isso arrastá-los com o mouse e depositá-los sobre a posição desejada.

O Navegador do projeto oferece um sistema de gerenciamento global de arquivos, o qual possibilita que um mesmo projeto seja acessado simultaneamente por vários computadores ligados a uma rede, sem interromper o ciclo do projeto.

Embora os acessos ao projeto possam ser simultâneos, todos eles são cuidadosamente controlados pelo *software* de modo a evitar que dois ou mais projetistas tentem modificar uma mesma parte do projeto concorrentemente.

Outro importante recurso disponibilizado pelo Navegador é a visualização através dos diversos níveis hierárquicos do projeto. Por meio dela, o projetista pode navegar facilmente entre os diferentes níveis hierárquicos, acessando e editando as diferentes entidades do projeto.

E.1.2 Compilação

A compilação de um projeto por meio do compilador Quartus II gera um conjunto de arquivos usados para a programação de dispositivos, simulação e análise de tempo. Além da compilação total do projeto, o *software* Quartus II também permite a compilação parcial com a finalidade de processar apenas as porções recém-modificadas do projeto. Essa

característica faz com que o tempo de compilação seja reduzido consideravelmente, uma vez que a parte não modificada não precisa ser re-compilada.

E.1.2.1 Síntese Lógica e Fitting

Uma das principais características do módulo sintetizador lógico do compilador Quartus II é a presença de algoritmos que reduzem e eliminam lógicas redundantes e não usadas presentes no projeto. Isso garante que os recursos lógicos presentes no dispositivo usado na compilação sejam utilizados da maneira mais eficiente possível.

Uma outra propriedade do compilador Quartus II é a capacidade de analisar o projeto, a fim de dividi-lo por suas funções e atribui-las a *look-up tables*, *product terms* ou *embedded memory logic blocks* presentes na arquitetura APEX.

O projeto sintetizado é automática e eficientemente implementado em um ou mais dispositivos, graças à aplicação de regras heurísticas pelo módulo Fitter do compilador Quartus II. Ele também se encarrega de reportar informações sobre a implementação do projeto, assim como sobre as partes dos dispositivos que não foram utilizadas.

E.1.2.2 Timing-driven Compilation

As opções de tempo que o usuário pode especificar para funções lógicas ou para o projeto todo através do compilador Quartus II são (fonte: [1]):

tPD – *propagation delays*;

tCO – *clock-to-output delays*;

tSU – *setup times*;

fmax – *internal clock frequency e system clock frequency*.

Os detalhes a respeito de como as opções de tempo são implementadas no projeto aparecem na janela de mensagens.

E.1.2.3 Formatos de simulação padronizados para a indústria

O compilador Quartus II cria arquivos netlist que podem ser utilizados em uma variedade de ambientes de simulação. Esses arquivos contêm informações funcionais e de tempo oriundas da síntese do projeto, as quais podem ser usadas com ferramentas de verificação de projetos padronizados para simulações dos tipos device-level ou board-level.

As intefaces disponíveis são: Verilog Netlist e VHDL Netlist.

Para cada interface, o compilador gera as versões 1.0 ou 2.1 do Standar Delay Output Format File (.sdo), que incluem informações de tempo e informações funcionais em arquivos separados.

E.1.2.4 Geração do arquivo contendo a programação

Os arquivos contendo a programação e a configuração do projeto são gerados pelo compilador e pelo programador Quartus II (fonte: [1]):

Programmer Object File (.pol);
SRAM Object File (.sof)
Hexadecimal (Inter-format) File (.hex)
Raw Binary File (.rbf)

E.1.3 Design Verificaton

No *software* Quartus II a verificação é feita por simulação e análise de tempo.

E.1.3.1 Simulação

Em um projeto, um ou mais dispositivos podem ser modelados de forma controlada e flexível pelo simulador Quartus II, que realiza simulações funcionais e de tempo com base em informações resultantes da compilação desse mesmo projeto.

Geralmente, os circuitos realizam pelo menos algum tipo de processamento sobre os sinais que alimentam suas entradas e direcionam os resultados para suas saídas também na forma de sinais. Dessa forma, resumidamente, a análise das saídas de um dispositivo permite interpretar funcional e temporalmente o conjunto de transformações sofrido pelos sinais de entrada. A principal tarefa do simulador Quartus II é tornar essa interpretação viável, uma vez que ele imita todo o trabalho do circuito, dede o recebimento do conjunto de sinais de entrada, passando por sua transformação, até disponibilizar os sinais de saída que carregam os resultados do processamento.

Há duas formas de se alimentar o simulador com os sinais de entrada. Uma delas é preencher os campos de uma tabela com números que irão representar os valores assumidos pelos sinais a serem processados. A outra, é utilizar o Waveform Editor para adicionar os valores dos sinais graficamente em forma de ondas. Assim como as entradas, os sinais resultantes da simulação podem ser expressos de forma gráfica (ondas) ou por textos.

O simulador permite ao projetista realizar uma série de tarefas, dentre elas: monitorar a presença de *glitches*; verificar o valor de registradores; detectar a presença de violação de tempo; alterar o estado de flip-flops; definir o conteúdo inicial de memórias RAM e ROM. Através da janela de mensagens e do Waveform Editor, o simulador pode revelar o tempo exato em que ocorre alguma falha de projeto detectada na simulação, ou ainda, pode mostrar a quantidade de tempo necessário para que o circuito atinja seu objetivo com sucesso.

Erros lógicos podem ser fácil e rapidamente detectados pelo engenheiro, uma vez que o simulador Quartus II suporta simulação funcional para testar operações lógicas do projeto, após ele ter sido sintetizado. O Waveform Editor mostra os resultados da simulação funcional e provê fácil acesso a todos os nós do projeto, incluindo as funções combinatórias.

Múltiplos dispositivos da Altera trabalhando conjuntamente, podem ser simulados ao mesmo tempo.

E.1.3.2 Análise de tempo

É a função do analisador de tempo Quartus II (fonte: [1]):

- Calcular, ponto a ponto, a matriz de atrasos de um dispositivo;
- Determinar a freqüência de *clock* necessária para os pinos do dispositivo;
- Calcular a freqüência máxima de *clock* para o circuito;
- Realizar análises de vários *clocks* simultaneamente;
- Realizar análises de tempo do tipo *multi-cycle*;
- Determinar os atrasos de propagação mais curtos e mais longos dentro do projeto;
- Identificar caminhos críticos no arquivo fonte do projeto e no Floorplan Editor.

E.1.4 Programação do dispositivo

Através do programador Quartus II é possível programar, configurar, verificar, examinar, realizar *blank-check* e testar funcionalmente os dispositivos da Altera. Para realizar essas tarefas é necessário que a placa com o dispositivo (ou dispositivos) Altera esteja conectada a uma porta (serial, paralela ou USB) de um PC via um cabo especial fornecido pela própria Altera.

Para que o *software* Quartus II seja executado adequadamente, um PC deve trabalhar com uma velocidade de pelo menos 400Mhz, ter uma memória RAM de 256MB até mais de 1 GB (dependendo do dispositivo selecionado na compilação), ter ao menos 4GB de espaço livre no disco rígido e possuir uma saída paralela, serial ou USB.

E.2 Comentários

As principais características da ferramenta de engenharia auxiliada por computador Quartus II, usada no projeto do co-processador M1, foram apresentadas neste apêndice.