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

Algoritmo de Criptografia AES em Hardware, Utilizando Dispositivo de Lógica Programável (FPGA) e Linguagem de Descrição de Hardware (VHDL).

Alessandro Augusto Nunes Campos

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos necessários para a obtenção de título de Mestre em Ciências em Engenharia Elétrica.

JUNHO de 2008

Itajubá - MG

A minha esposa Daniela, filho João Matheus, aos meus pais, Aloísio e Maria Roselisse, e irmãos, Benedito, Carlo e Fausto.

“Os pequenos atos que se executam são melhores que todos aqueles grandes que se planejam”.

George C. Marshall

Agradecimentos

Inicialmente a Deus, que no meu caminhar sempre me orientou e me conduziu principalmente nas mais difíceis decisões.

Seguido de um agradecimento especial a minha esposa e ao meu filho que nestas últimas etapas souberam suportar meu humor e minha ausência além de me ajudar com grande incentivo

Sem esquecer todos os meus familiares, especialmente os meus pais e irmãos, por seu incondicional incentivo e aposta na minha conquista.

Uma referência ao professor e orientador, Dr. Robson Luiz Moreno, pela disponibilidade e entusiasmo, pela confiança, amizade e pela ajuda sem medida na realização desse trabalho.

Lembrando também dos amigos e colegas de trabalho do Instituto Nacional de Pesquisas Espaciais - INPE e ao professor Dr. Eurico Rodrigues de Paula, pelo apoio, incentivo e constante preocupação com o andamento deste trabalho.

Por fim aos amigos, pelo apoio e pelos momentos de descontração, além de importantes conselhos.

Meus mais sinceros agradecimentos.

Resumo

A comunicação, das mais variadas formas e meios, sempre foi uma constante na existência humana, sendo considerada, por muitos especialistas, como uma necessidade básica do homem para a sua sobrevivência e existência. Com a evolução da comunicação, da forma falada para a escrita e atualmente para os sistemas de telecomunicação e informática, o homem sentiu a necessidade e a importância do sigilo para a realização de determinadas operações de envio e recebimento de informação. Sendo assim, o processo de se comunicar ou enviar informações consideradas sigilosas, passou a ter especial atenção por parte da humanidade. Visto que existem relatos de cifragem de informação escrita desde a época dos faraós e dos imperadores romanos, nos dias de hoje, com o advento das novas tecnologias, a criptografia se tornou ferramenta básica para transferência mais segura de informação eletrônica.

Este trabalho apresenta o algoritmo de criptografia AES (Advanced Encryption Standard), denominado algoritmo de criptografia simétrica, que foi homologado e certificado pelo NIST (National Institute of Standards and Technology) órgão governamental dos Estados Unidos da América, como novo padrão de cifragem de informação, utilizado em operações de comércio, tecnologia da informação e segurança para dados eletrônicos. Este trabalho foi desenvolvido utilizando linguagem de descrição de hardware (VHDL) e aplicado em dispositivo de lógica programável (FPGA) com o intuito de se obter um sistema de grande velocidade e capacidade de processamento de dados. Será apresentado o seu funcionamento, além de implementações de alguns blocos funcionais do algoritmo, com os resultados obtidos em um dispositivo do fabricante ALTERA ® .

Abstract

The communication, of the most varied forms and ways, was always constant in human existence, as thought by many specialists like a basic necessity for the survival and existence of man. With the evolution of the communication form, the spoken to the written way and at present, for the telecommunication systems and computers, man felt the necessity and the importance of secrecy, for the realization of determined operations of sending and receiving information. Being so the process of communicating or sending respected secret information started to have special attention by humanity. Seeing that encoded information reports exist from the time of the Pharaohs and the Roman emperors, nowadays, with the advent of the new technologies, the cryptography became a basic tool for transferring of electronic information safer.

This work presents the algorithm of cryptography AES (Advanced Encryption Standard), called algorithm of symmetrical cryptography, which was ratified and certified by the NIST (National Institute of Standards and Technology), government organ of the United States of America, like new standard of information encoding to be used in operations of commerce, technology of the information and security for electronic data. This work was developed using hardware description language (VHDL) and applied in device of programmable logic (FPGA) with the intention of obtaining a system of great speed and capacity of processing. His functioning will be presented, besides implementations of some functional blocks of the algorithm, with the obtained results in a device of the manufacturer ALTERA ®.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

AES	*Advanced Encryption Standard*
IBM	*Information Business Machine*
NBS	*National Bureau of Standards*
NSA	*National Security Agency*
DES	*Digital Encryption Standard*
RSA	*Rivest-Shamir-Adleman*
ACM	*Association for Computing Machinery*
FBI	*Fedral Bureau if Investigation*
RC3,4,5,6	*Ron's Cipher*
PGP	*Pretty Good Privacy*
NIST	*National Institute of Standards and Technology*
SSL	*Secure Socket Layer*
DH	*Diffie, Hellman*
IDEA	*International Data Encryption Algorithm*
MD4,5	*Message Digest*
SHA	*Secure Hash Algorithm*
RAM	*Random access memory*
GF	*Galois Field*
XOR	*Exclusive OR*
ASICs	*Application Specific Integrated Circuits*
FPGAs	*Field Programmable Gate Arrays*
CAD	*Computer Aid Design*

HPC	*High Performance Computing*
CLB	*Configurable Logics Blocks*
LUT	*Look Up Table*
SRAM	*Static RAM*
LAB	*Logic Array Block*
PLD	*Programmable logic device*
EPROM	*Erasable Programmable Read-Only Memory*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
ROM	*Read-Only Memory*
PCB	*Printed Circuit Board*
HDL	*Hardware Description Language*
VHDL	*Very High Speed Integrated Circuit HDL*
RTL	*Register Transfer Level*
IEEE	*Institute of Electrical and Electronics Engineers, Inc*

Capítulo 1

Introdução

Nos dias atuais, dificilmente alguém se depara com a enorme quantidade de avanços tecnológicos que o homem desenvolveu e consegue entender por completo o seu impacto na vida cotidiana. Talvez isto ocorra por falta de informação ou talvez por abstração da tecnologia, mais uma coisa é certa, ela está ai e nos auxilia diariamente em nossas tarefas das mais variadas, ainda que não a percebamos.

A capacidade da inventividade humana nunca parou de se desenvolver, e com isto temos atualmente maravilhas tecnológicas improváveis e impensáveis há pouco mais de 100 anos. A tecnologia mãe desses avanços é, sem dúvida, considerada por muitos, como sendo a descoberta da eletricidade e do magnetismo. Dito isto, podemos delimitar a história tecnológica da humanidade em sendo antes e depois destas descobertas. Com este conhecimento inicial, o homem deu seqüência ao desenvolvimento e pesquisa destas áreas de estudos sendo que, através de mais descobertas, surgiram como uma das principais variantes posteriores, as comunicações por fios e ondas de rádio [01][02].

O surgimento desta comunicação, agora denominada “Telecomunicação”, provocou um salto considerável no desenvolvimento de novas tecnologias e na sua própria, causada principalmente pelo incremento de velocidade no envio e recebimento de novas informações, agora disponíveis a humanidade. Visto isto, e com o olhar no agora, o mundo

vislumbra um outro incremento da sua capacidade de desenvolvimento tecnológico pois, mais recentemente com a criação e desenvolvimento dos computadores e da Internet, podemos realizar outro salto tecnológico, como já visto no passado.

Agora podemos iniciar realmente a discussão de que propõe este trabalho. Sua proposta não poderia deixar de englobar estes últimos avanços da humanidade, no que diz respeito ao envio e recebimento de informação por computadores através de redes de telecomunicações como a internet. Conforme será apresentado a seguir, este trabalho irá tratar do uso mais seguro de envio e recebimento de dados usando-se uma técnica consagrada a milhares de anos para tal fim denominado criptografia, que será explicada, detalhada e com algumas de suas funções aplicadas em um dispositivo especialmente desenvolvido para este fim.

1.1. Origem e Justificativa

A busca por sistemas de criptografia mais seguros e rápidos tem sido uma constante no desenvolvimento tecnológico atual, principalmente nos dispositivos que fazem uso de computadores e internet nos dias atuais. Assim a busca por um sistema com estas características sempre está evoluindo como será visto no capitulo dois referente ao histórico da criptografia neste trabalho. É facilmente comprovado hoje, pelos meios de comunicação e mídias diversas, que o investimento em técnicas seguras e rápidas para envio de informação eletrônica é crescente, sendo o sigilo o fator mais importante em determinadas áreas de transferência de dados como, por exemplo, a financeira, principalmente o comércio eletrônico, de segurança e defesa, no que diz respeito a soberania e interesses estratégicos de alguns paises, além de diversas outras [02][03].

Sendo assim, a origem e motivação deste trabalho surgiram com o interesse em colaborar e tentar ajudar a resolver as questões relacionadas a estes problemas tecnológicos que insistem em desafiar os pesquisadores a cada dia. Como cada sistema de segurança criado diariamente provoca por um lado o interesse de alguns outros em “decifrá-lo” e “quebrá-lo”, a corrida por uma solução plausível atual e mais segura está longe de acabar. Assim o uso de técnicas modernas de implementação de algoritmos de criptografia homologados, consagrados e indicados por órgãos internacionais, aliados a inovações em linguagens de

programação e dispositivos de lógica programável são uma opção muito próxima do ideal para os dias de hoje. Assim consideramos que esta necessidade é a justificativa principal do nosso trabalho.

1.2. Objetivos

Nos tempos atuais, temos no mercado diversas soluções comerciais que hoje usam recursos específicos de hardware e software, para o provimento de dispositivos de criptografia para os mais variados fins. Assim sendo, apresenta-se neste trabalho, alguns métodos de criptografia e discuti-se o funcionamento específico da nossa opção. Como dito, a criptografia atual possui diversas técnicas e implementações e cada uma possui uma aplicação mais indicada, sendo que após uma vasta pesquisa, podemos identificar que, em uma determinada área de mercado há oportunidades de desenvolver inovações com pouco investimento e bons resultados.

O objetivo principal dessa dissertação é a apresentação e implementação parcial de blocos funcionais de um algoritmo de criptografia [1][2][3](já homologado e consagrado pelo uso) em dispositivos de baixo custo, capaz de prover um bom nível de segurança para transação de grande quantidade de dados por uma rede de comunicação sem que seja necessário utilizar soluções proprietárias de alto custo.

Este trabalho tem em vista também, com a implementação deste algoritmo em circuito integrado, a redução de trabalho de processamento e consumo de recursos computacionais compartilhados para a execução de tarefas de cifragem e decifragem de dados. Com isto, pretende-se tirar vantagens dos recursos de uma estrutura especifica de criptografia, propiciando atingir o objetivo de uso em sistemas de telecomunicação de alto tráfego de dados.

Com a demanda crescente por sistemas de criptografia para uso em sistemas de comunicação para uso na internet, a proposta de desenvolvimento vislumbra que no futuro este sistema possa ser implementado em um dispositivo de lógica programável de baixo custo

e implantado em uma simples placa de comunicações de rede “Ethernet” ou em sistemas mais robustos como roteadores e “switches”.

1.3. Estrutura do Trabalho

O trabalho está organizado em sete capítulos, sendo um de introdução, um de conclusão e os demais de desenvolvimento.

O Capítulo 2 descreve um estudo detalhado sobre a criptografia, abordando desde a definição e métodos desta área do conhecimento até uma abordagem sobre as técnicas que mais são utilizados nos dias de hoje. Neste capítulo apresenta-se curiosidades das antigas formas de cifragem e também o poder que a computação atribuiu aos sistemas mais modernos usados nas transações de telecomunicações atuais.

O Capítulo 3 apresenta o algoritmo de criptografia AES (*Advanced Encryption Standard*), que foi o escolhido nesta proposta como o que melhor pode cumprir as exigências de segurança e desempenho definidas por este trabalho. Neste capítulo poderá ser encontrada uma descrição completa de funcionamento do algoritmo, abordando o histórico do algoritmo e uma descrição completa dos seus blocos funcionais de cifragem, decifragem e geração de chaves de criptografia.

O Capítulo 4 mostra uma descrição sobre dispositivos de lógica programável. Poderá ser visto como é composto e funciona um dispositivo de lógica programável, suas partes e detalhes construtivos, além dos recursos de programação junto com as possibilidades de aplicação atualmente.

O Capítulo 5 faz uma apresentação sobre linguagem de descrição de hardware. Será possível entender como a linguagem de descrição de hardware funciona e instancias os seus componentes além de sua importância atual na portabilidade e flexibilidade de desenvolvimento de projetos para várias plataformas.

O Capítulo 6 apresenta o trabalho propriamente dito. Com a caracterização e organização do funcionamento do algoritmo de criptografia, o detalhamento de algumas funções de criptografia utilizadas e o fluxo de funcionamento destas mesmas (modelagem). Desta forma poderá se verificar a funcionalidade do projeto, dentro de tudo o que foi proposto, além da validação e os resultados desta dissertação.

O Capítulo 7 apresenta as conclusões e as sugestões de trabalhos futuros que poderão ser implementados, principalmente com versões completas do algoritmo de criptografia, além de apontar possíveis melhorias que poderiam ser feitas no algoritmo de criptografia para um possível incremento no desempenho do sistema.

Também são encontrados neste trabalho de dissertação alguns anexos que tem por intenção contribuir com informações complementares, a fim de auxiliar a compreensão do trabalho. Para finalizar este capítulo vamos a seguir apresentar uma descrição mais profunda sobre criptografia.

Capítulo 2

Criptografia

A intenção principal deste capítulo será realizar uma apresentação sobre criptografia englobando todos os aspectos necessários à compreensão e avaliação de funcionamento de técnicas antigas e atuais existentes.

Pensando em melhorar o entendimento, serão utilizados no transcorrer deste trabalho diversos termos formais cuja intenção é atribuir certa atividade ou função dentro do universo da criptografia sendo que, o processo que converte informações sigilosas em algo sem sentido, costuma utilizar os termos (*encripta, codifica, criptografa, cifra*), e para tornar as informações sigilosas cifradas em legíveis novamente, utiliza-se os termos (*decripta, decodifica, decriptografa, decifra*). Vale como informação que, existem estudos específicos que tratam da quebra de sistemas de criptografia denominados como *análise criptográfica* e os profissionais que as estudam são denominados *criptoanalistas* [02][03].

Varias pessoas acreditam que a criptografia foi desenvolvida para uso militar durante a Primeira e a Segunda Grande Guerra, mas o que se sabe é que a criptografia existe há milhares de anos. Há evidências de que ela já se encontrava presente no sistema de escrita hieroglífica dos egípcios com a intenção, dentre varias outras, de ajudar a preservar os tesouros e riquezas dos faraós e em documentos de generais gregos e romanos que eventualmente eram enviados a frente de batalha com a intenção de impedir, em caso de

interceptação, a sua interpretação. Porém somente no início dos anos 70 que a criptografia surgiu como área de investigação acadêmica de conhecimento generalizado e hoje com o uso da computação pode ser demonstrado todo o seu potencial [03].

2.1. Definição e Características

A criptografia é formada a partir dos termos gregos *kryptos* (escondido, oculto) e *graphé* (grafia, escrita) e é a ciência que torna possível a comunicação mais segura entre dois agentes, sobre um canal aberto, convertendo informação legível em algo sem sentido, com a capacidade de ser recuperada ao estado original com o uso de processos inversos ou não.

Para Terada [01] a Criptografia pode ser descrita como:

> "a ciência que estuda a transformação de dados de maneira a torná-los incompreensíveis sem o conhecimento apropriado para a sua tradução, tornando os conteúdos secretos, evitando riscos internos e externos que venham a ocorrer durante o trajeto dos dados enviados, que são convertidos em um código que só poderão ser traduzidos por quem possuir a "chave" secreta, enquanto que a Criptoanálise executa o processo inverso, sendo a ciência que estuda a decifração, tornando o código compreensível."

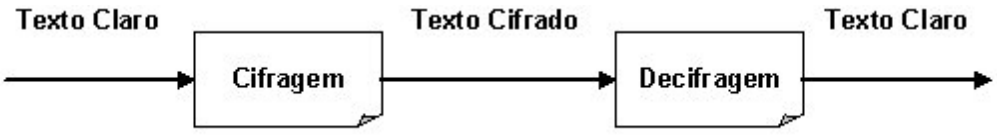


Figura 2.1 - Esquema geral para cifragem e decifragem de um texto

Conforme esquema apresentado pela figura 2.1, temos o fluxo de cifragem e decifragem de um texto. Existem basicamente dois meios de se criptografar informações, um usando métodos de cifras e outro com o uso de códigos. As cifras protegem as informações com o uso de substituição ou transposição de caracteres na mensagem original, sendo que apenas as pessoas que conhecem os processos de cifragem podem ter acesso às informações

ali contidas. Porém caso consiga-se descobrir a cifra o algoritmo todo é exposto.

Já os códigos, fazem uso de uma chave de criptografia que é introduzida no algoritmo, sendo assim mais seguros no aspecto onde, mesmo que se conheça o algoritmo de cifragem, a informação ainda se manterá segura, sendo que ainda é necessário que se conheça a chave de criptografia para se decifrar a informação. Visto isto, os algoritmos de códigos são mais utilizados na atualidade e o desenvolvimento de códigos está mais direcionado nos dias de hoje a procura de um algoritmo que possa gerar chaves mais seguras e difíceis de se descobrir e não mais no código ou cifras propriamente ditas.

2.2. Histórico

Conforme dito anteriormente, a criptografia não é um recurso de segurança tão novo assim, visto que já estava presente desde os tempos dos faraós e romanos. Existem relatos que os romanos utilizavam códigos secretos para enviar e receber informações nas batalhas. Com o acontecimento das grandes guerras mundiais e a invenção do computador a criptografia se desenvolveu muito rápido introduzindo assim uma nova modalidade de estudos de estratégia, utilizando cada vês mais algoritmos com matemática ainda mais complexa e por conseqüência potencialmente mais seguros [03].

O primeiro exemplo de documento de escrita cifrada tem como origem aproximada 1900 a.C., quando um escriba do faraó egípcio Khnumhotep II realizou a substituição de algumas palavras e trechos de texto em um documento que informava o caminho aos tesouros, fazendo assim, caso o documento fosse roubado, com que o ladrão ficasse perdido nas catacumbas da pirâmide e assim morresse de fome [03].

Batizada com o nome do imperador romano que a desenvolveu e utilizou, a cifra de substituição de César datada de 50 a.C. é um exemplo clássico das importantes e imprescindíveis características que a criptografia possui para fins militares e de defesa. Para compor seu texto César alterava letras desviando-as em três posições do alfabeto para que assim o texto se tornasse indecifrável, ou seja, ‘A’ se tornava ‘D’, ‘B’ se tornava ‘E’, ‘C’ se tornava ‘F’ e assim consequentemente além de reforçar seu método substituindo letras latinas

por gregas. O código de César é o único que ainda é utilizado até os dias de hoje com algumas alterações.

Tabela 2.1 - Histórico de evolução da criptografia.

Ano	Descrição
1943	Máquina Colossus, projetada para quebrar códigos.
1969	James Ellis desenvolve um sistema de chaves públicas e privadas separadas
1976	Diffie-Hellman, algoritmo baseado no problema do logaritmo discreto é criado e a IBM apresenta a cifra Lúcifer ao NBS (*National Bureau of Standards*) que após algumas modificações a NSA (*National Security Agency*) adota a cifra como padrão (FIPS 46-3, 1999) conhecido hoje como DES (*Data Encryption Standard*).
1977	Ronald L. Rivest, Adi Shamir e Leonard M. Adleman discutem como criar um algoritmo de chave pública prático e a partir disto desenvolvem um sistema baseado na dificuldade de fatoração de números primos grandes como técnica de criptografia, que futuramente seria denominado RSA, as iniciais de seus autores.
1978	O algoritmo RSA é publicado na *Association for Computing Machinery*
1991	Phil Zimmermann torna pública sua primeira versão do PGP (*Pretty Good Privacy*) em resposta as exigências do FBI (*Federal Bureau of Investigation*) de acessar qualquer texto claro da comunicação entre usuários de uma rede de comunicação digital. Por ser freeware tornou-se rapidamente padrão mundial.
1994	Ronald L. Rivest, autor dos algoritmos RC3 e RC4, publica a proposta do algoritmo RC5 na internet, o qual usa rotação dependente de dados como sua operação não linear e é parametrizado de forma que o usuário possa variar o tamanho da chave, do bloco de cifragem e o número de estágios usados no processo de cifragem das informações.
1997	O PGP 5.0 é amplamente distribuído para uso não comercial. O código DES de 56 bits é quebrado por uma rede de 14.000 computadores
1998	O código DES é quebrado em 56 horas por pesquisadores do vale do silício
1999	O código DES é quebrado em 22 horas e 15 minutos, mediante a união da Eletronic Frontier Foundation e a Distributed.Net, que reuniam 100.000 computadores pessoais ao DES Cracker pela internet
2000	O NIST (*National Institute of Standards and Technology*), antigo NBS, anunciou um novo padrão de uma chave secreta de cifragem escolhidos entre 15 candidatos através de concurso. Este novo padrão foi criado para substituir o algoritmo DES cujo tamanho das chaves tornou-se insuficiente para conter ataques de força bruta. O algoritmo Rijndael, cujo nome é abreviação do nome dos autores Rijmen e Daemen, foi escolhido para ser o futuro AES (*Advanced Encryption Standard*) (FIPS 197, 2001).
2000-Atual	Muitos professores, estudantes e profissionais de computação em universidades e centros de pesquisa se motivaram e iniciaram pesquisas sobre novas formas de se implementar algoritmos e soluções de segurança, surgindo assim uma nova onda voltada a realizar otimizações e desenvolvimento destas primeiras implementações onde a principal tendência é a implementação em hardware.

Considerado para todos os efeitos, que todo método que utilize cifras baseados na substituição cíclica do alfabeto é denominado como código de César. Para avaliar a sua importância e simplicidade esta cifra foi utilizada pelos oficiais sulistas na guerra de sucessão americana e pelo exército russo em 1915 [02][03].

Com a chegada das comunicações sem fio em meados de 1901, apesar da vantagem de uma comunicação de longa distância se tornar imensamente mais barata do ponto de vista de infra-estrutura, o sistema aberto iniciava um novo desafio para criptografia e seus desenvolvedores, sendo que agora a questão era como trafegar informações sigilosas em canais abertos com um razoável nível de segurança.

Assim em 1921, Edward Hugh Hebern fundou uma companhia produtora de máquinas de cifragem eletromecânicas [02] baseadas em rotores que giravam a cada caractere cifrado, sendo muito utilizada para comunicações em ambientes de canal aberto. Após isto, entre 1933 e 1945, Arthur Scherbius aperfeiçoou a máquina enigma até se tornar a ferramenta criptográfica mais importante da Alemanha nazista. Este sistema foi quebrado pelo matemático polonês Marian Rejewski e foi um dos maiores fatores que auxiliaram os Aliados na vitória sobre as tropas do Eixo na segunda guerra mundial [02][03].

2.3. Métodos de Criptografia

Como já dito, os principais métodos de criptografia que foram utilizados pela humanidade são sem duvida os que usam codificação (chaves) ou cifragem (substituição e transposição). Já os mais utilizados atualmente são os de codificação, que apresentam maior segurança velocidade e capacidade de gerenciamento de chaves. Uma outra classificação dos algoritmos de cifragem pode ser realizada se considerarmos a forma com que as informações serão tratadas, ou seja, os dados podem ser criptografados em blocos de tamanhos determinados ou de forma contínua denominada criptografia de fluxo. Assim, temos aplicações ideais para cada tipo de trabalho de cifragem onde cada algoritmo deve ser determinado não só pelas características de segurança mais também pela aplicação como visto na tabela 2.2 [03].

Tabela 2.2 - Algoritmos por aplicação

Aplicação	Cifragem recomendada	Comentários
Banco de Dados	Bloco	A interoperabilidade de um outro software não é relevante, mas é necessário reutilizar as chaves.
E-mail	AES	Se ganha interoperabilidade em todos os pacotes de e-mail utilizando o AES padrão.
SSL	RC4	A velocidade é extremamente importante, cada conexão pode ter uma nova chave. Assim a maioria dos navegadores e servidores possuem RC4
Criptografia de Arquivos	Bloco	A interoperabilidade não é relevante, porém cada arquivo pode ser cifrado com a mesma chave.

Como o principal algoritmo de substituição e transposição já foi descrito com suficiente abrangência anteriormente (Cifra de César), vamos nos ater aos algoritmos de codificação que fazem uso de chaves de criptografia. Os algoritmos mais usados atualmente que trabalham com chaves para criptografar mensagens podem, dentro de um subtipo de algoritmos, ser classificados de duas formas básicas: os que trabalham com uma única chave privativa para cifrar e decifrar a mensagem, chamados de algoritmos de chave simétrica e os que utilizam uma chave pública para cifrar e outra privativa para decifrar a informação, chamados por sua vez de algoritmos de chave assimétrica, conforme a seguir apresentado na figura 2.2.

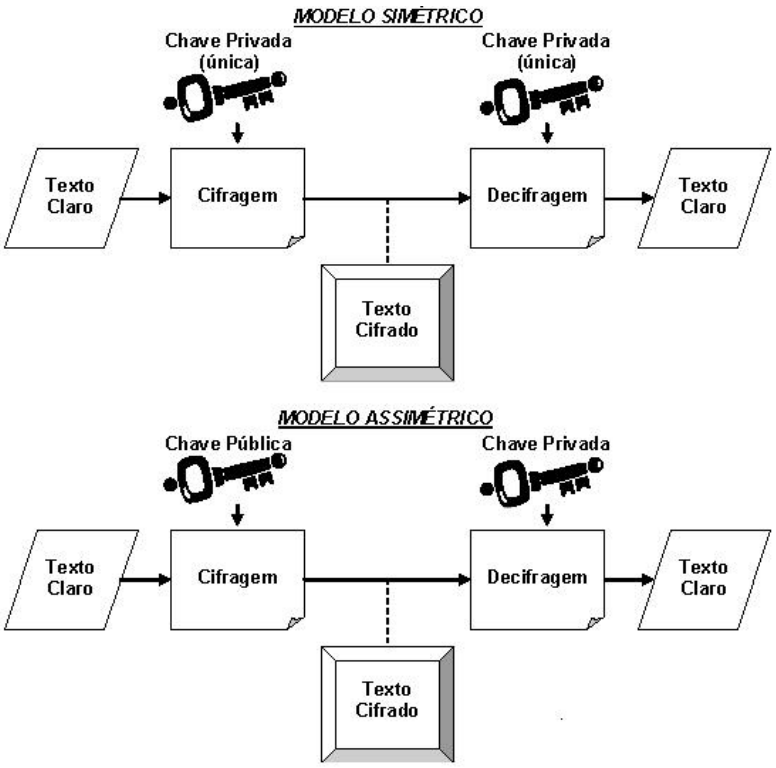


Figura 2.2 - Modelos simétrico e assimétrico de criptografia.

Assim sendo, podemos já identificar vantagens e desvantagens destes métodos que, conforme já informado, possuem recursos interessantes para determinadas aplicações. Na tabela 2.3 podemos visualizar características já um tanto quanto óbvias, com relação a alguns parâmetros em cada um destes algoritmos, sendo entre eles, a sua velocidade, manipulação de chaves e recursos extras [03].

Tabela 2.3 - Comparação entre algoritmos de chave simétrica e assimétrica.

Criptografia simétrica	**Criptografia assimétrica**
Rápida	Lenta
Gerência e distribuição de chaves complexa	Gerência e distribuição de chaves simples
Sem recursos como assinatura digital	Com recursos como assinatura digital

O algoritmo de chave simétrica é classificado como rápido devido a sua particularidade de usar apenas uma única chave de tamanho pré-determinado, enquanto comparado a um de chave assimétrica que utilizará duas chaves de mesmo tamanho tornando assim o processo mais lento, quando existir a necessidade de troca contínua de chaves. Já com relação ao gerenciamento e distribuição de chaves, como o algoritmo assimétrico possui o recurso de chave pública para cifrar informação, este recurso é muito utilizado para o envio, troca e gerenciamento seguro de suas chaves, enquanto que no simétrico isto não é possível, sendo necessárias outras formas para troca e gerenciamento de chaves [03].

Os recursos extras como a assinatura digital desenvolvida e utilizada hoje, nada mais é do que o uso invertido do algoritmo de chave assimétrica. Seu funcionamento utiliza a chave privativa de quem assina para cifrar a informação e a pública para decifrar, comprovando assim que a informação tem origem confiável uma vez que, pelo funcionamento do algoritmo, apenas quem poderia ter codificado aquela informação seria o portador da chave privativa.

2.4. Uso Atual

Com a intenção de fornecer uma informação mais global, vamos a seguir listar e apresentar alguns dos algoritmos simétricos mais usados e importantes desta área de conhecimento, tentando seguir certa ordem cronológica.

Tabela 2.4 - Características dos algoritmos simétricos mais conhecidos [03].

Algoritmo	**Tipo**	**Tamanho da chave**	**Tamanho do bloco**
DES	Bloco	56 bits	64 bits
Triplo DES	Bloco	56 bits	64 bits
IDEA	Bloco	128 bits	64 bits
Blowfish	Bloco	32 - 448 bits	64 bits
RC4	Fluxo	0 - 256 bits	-----
RC5	Bloco	0 - 2040 bits	32-64-128 bits
CAST-128	Bloco	40 - 128 bits	64 bits
AES	Bloco	128-192-256 bits	128-192-256 bits
MARS [04]	Bloco	Variável	128 bits
RC6 [05]	Bloco	Variável	128 bits
Serpent [06]	Bloco	Variável	128 bits
Twofish [07]	Bloco	128-192-256 bits	128 bits

No caso dos algoritmos assimétricos, o mais citado e utilizado é sem dúvida o RSA onde, através de uma chave pública e outra privativa se realiza o processo de cifragem e decifragem da informação. Devido à ineficiência e baixa velocidade dos algoritmos assimétricos, os métodos de assinatura digital utilizados na prática não cifram os documentos propriamente ditos, mas uma súmula destes, obtidas pelo processamento dos documentos ou informações, por meio de uma função denominada “função de hashing” [03].

A função de hashing gera uma saída independente da informação original de tamanho fixo (dependendo do algoritmo de 128 - 256 bits), não importando o tamanho da entrada, assim pode-se ter um outro dado separado da informação com o qual se pode utilizar para certificar a integridade e originalidade da informação. No uso de funções de hashing para assinatura digital, a verificação da informação é feita gerando-se o código de *hash* logo após a informação ser recebida, sendo assim, pode-se comparar este código com o código inicial gerado na origem e em caso positivo confirmar a autenticidade e integridade da informação.

As características mínimas que uma função de hash deve ter são: ser simples (eficiente e rápido) de se computar o hash de dada mensagem; impraticável de se determinar a entrada a partir de seu hash; impraticável de se determinar uma outra entrada, que resulte no mesmo hash. Apresenta-se na tabela 2.5 alguns dos algoritmos de hashing mais conhecidos da comunidade de segurança.

Tabela 2.5 - Características de algoritmos de hashing [03].

Algoritmo de hash	**Tamanho do hash**	**Kbytes/s**
Abrreast Bavies-Meyer (c/IDEA)	128 bits	22
Davies-Meyer (c/DES)	64 bits	9
GOST-Hash	256 bits	11
NAVAL (5 passos)	Variável	118
MD4 – Message Digest 4	128 bits	236
MD5 – Message Digest 5	128 bits	174
N-NASH (15 rounds)	128 bits	24
RIPE-MD	128 bits	182
SHA – Secure Hash Algorithm	160 bits	75
SENEFRU (8 passos)	128 bits	23

Conforme [03], alguns destes algoritmos sugerem aplicações de origem governamental como militar e defesa além de outros de uso comercial privativo, porém podemos citar como mais conhecidos pela comunidade acadêmica e comercial os SHA, MD4 e MD5, sendo muito comum o seu uso em aplicações internet como em servidores e navegadores.

Capítulo 3

Algoritmo AES (Rijndael)

No dia 2 de Janeiro de 1997, o órgão oficial norte-americano NIST – National Institute of Standards in Technology (Instituto Nacional de Normas em Tecnologia) anunciou formalmente um plano para definir um algoritmo como o novo padrão para criptografia, convidando qualquer pessoa a desenvolver um algoritmo. Definido que o novo padrão seria conhecido como AES, foi aberta assim uma espécie de "concurso", com a condição que o vencedor não teria quaisquer direitos quanto à propriedade intelectual do algoritmo selecionado. Os critérios para avaliação dos algoritmos que viessem a concorrer seriam: segurança (sem nenhuma fraqueza algorítmica), desempenho (rápido em várias plataformas) e tamanho (não poderia ocupar muito espaço nem utilizar muita memória) [03].

Várias pessoas físicas e jurídicas desenvolveram seus algoritmos, e no dia 20 de agosto de 1998, o NIST selecionou 15 candidatos. Vários dos 15 algoritmos originais não duraram muito tempo, pois em alguns foram descobertas fraquezas e em outros simplesmente eram muito grandes ou muito lentos, reduzindo a lista em apenas 5 algoritmos até agosto de 1999, sendo eles o MARS, RC6, Rijndael, Serpent e Twofish. No ano seguinte, estes algoritmos foram testados para decidir qual deles seria o vencedor. Finalmente, no dia 2 de outubro de 2000 [03], o NIST anunciou como grande vencedor o algoritmo Rijndael [08] [09], desenvolvido por dois pesquisadores belgas, Vincent Rijmen e Joan Daemen.

O Rijndael é um algoritmo de criptografia de bloco simétrico (mesma chave de cifragem e decifragem), com tamanho (quantidade de bits de trabalho) de bloco e de chave variáveis, podendo ser especificados independentemente para 128, 192 ou 256 bits [08], possui facilidade de implementação, propiciando uso em *Smart Cards* (cartões magnéticos utilizados em operações bancárias ou de compra eletrônica) e outros equipamentos que utilizam pouca memória RAM, além disso, utiliza poucos ciclos de processamento [03]. O código desse algoritmo é bem enxuto e não depende de nenhum outro tipo de componente criptográfico, como gerador de números randômicos. Esse aspecto faz com que sua utilização apresente um nível de segurança superior [09].

No AES o resultado das diversas operações intermediárias realizadas é colocado em um lugar denominado “matriz de estado” (Fig. 3.1). Uma matriz de estado pode ser definida por uma matriz retangular de bytes, onde esta matriz possui quatro linhas por um número *Nb* (número de blocos) de colunas que é igual ao tamanho do bloco (128, 192, 256) dividido por 32 [09].

A cifra da chave também pode ser definida como uma matriz retangular de bits também com quatro linhas por um número *Nk* de colunas igual ao tamanho da chave (128, 192, 256), dividido por 32, conforme exemplificado na figura 3.1 abaixo [08] [09].

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figura 3.1 - Matriz de estado e matriz de chave para *Nb* = 6 e *Nk* = 4 [08].

Em alguns casos precisa-se demonstrar estas matrizes como vetores de quatro bytes, cujo comprimento se refere ao tamanho da linha da matriz retangular. Como a quantidade de colunas da matriz é obtida dividindo-se o tamanho do bloco por 32, as colunas da matriz somente podem assumir o tamanho de 4, 6 ou 8.

Quando é necessário especificar cada byte dentro do vetor usamos as letras *a, b, c* e *d* como índice. Tanto a entrada como a saída do processo de criptografia AES é tratada como um vetor de 8-bits (1 byte) numerados de tamanho igual a *4*Nb - 1* (estes

blocos podem ser de 16, 24 ou 32 bytes). A cifragem da chave também é tratada da mesma forma, com a diferença de que o vetor é numerado de *4*Nk-1*. O algoritmo transforma os dados através do número de rodadas, usando quatro "funções" diferentes, sendo elas: *ByteSub, ShiftRow, MixColumn* e *AddRoundKey*. A última rodada é um pouco diferente, pois ela apresenta apenas as funções: *ByteSub, ShiftRow* e *AddRoundKey*. Logo descreveremos as quatro transformações que ocorrem a cada rodada do algoritmo [09]. A seguir explicaremos com mais detalhes o algoritmo e suas funções.

3.1. Especificação do Algoritmo

Para o algoritmo AES final aprovado, definiu-se que o tamanho do bloco de entrada (input), saída (output) e da matriz de estado interna deve ser de 128 bits (16 bytes). Isto é representado por *Nb* = 4, que reflete o número de palavras de 32 bits (números de colunas) na matriz de estado. Já o tamanho da chave de criptografia 'K' pode ser de 128, 192, ou 256 bits (16, 24 e 32 bytes). O tamanho da chave é representado por *Nk*= 4, 6, ou 8, que reflete o número de palavras de 32 bits (número de colunas) na chave de cifragem.

O número de rodadas (quantidade de interações) que serão utilizados no algoritmo, para a cifragem ou decifragem, depende do tamanho da chave a ser utilizada, sendo este representado por *Nr*, onde *Nr* = 10 quando *Nk = 4*, *Nr*= 12 quando *Nk* = 6, e *Nr* = 14 quando *Nk* = 8, sendo *Nk* o número de palavras de 32 bits (ou numero de colunas da matriz de chave). Para a implementação, podemos definir o tamanho da chave, o tamanho do bloco e o número de rodadas conforme visto na figura. 3.2 a seguir assim, pode-se modificar facilmente a implementação com o intuito de ajustar o tamanho dos elementos lógicos para quaisquer mudanças, alterações e ajustes na norma atual [09].

	TAMANHO DA CHAVE *(Nk words)*	TAMANHO DO BLOCO *(Nb words)*	NUMERO DE RODADAS *(Nr)*
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figura 3.2 - Tabela de combinações [08].

Tanto na cifragem como na decifragem, o algoritmo AES trabalha com rodadas (número de iterações), que são formadas por diferentes transformações orientadas por bytes:

1)Substituição de byte usando a tabela de substituição (S-box);

2)Deslocamento de linhas da matriz de estado por valores diferentes;

3)Mistura dos dados com cada coluna da matriz de estados;

4)Adicionar a chave de rodada na matriz de estado [09].

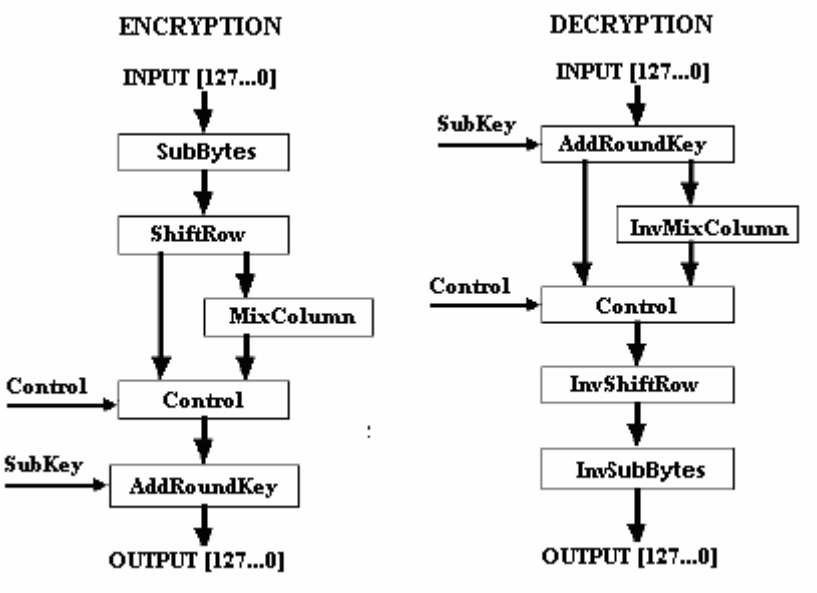


Figura 3.3 - Diagrama detalhado da função de processamento principal [08] [09].

3.1.1. Especificação *Galois Field* (GF 2^8)

Todos os bytes no algoritmo AES são interpretados como sendo elementos de um campo finito que podem ser representados por uma descrição polinomial, do tipo $b_7x^7+ b_6x^6+ b_5x^5+ b_4x^4+ b_3x^3+ b_2x^2+ b_1x+ b_0$, ou seja, o byte 57H {binário 01010111} pode ser representado pelo polinômio x6 + x4 + x2 + x + 1. Desta forma se torna possível toda uma descrição matemática de funções como adição, multiplicação e outras tratando-se agora o byte como polinômio e efetuando tais operações, como operações polinomiais.

A representação polinomial de *Galois Field* possui uma representação mais fácil e permite uma representação numérica quando ao realizar operações polinomiais o resultado produz um outro polinômio de ordem maior do que os operandos originais. Desta maneira a teoria aplicada (*GF* 2^8) na dedução matemática usa do recurso de um polinômio irredutível (ou módulo), “x4 + 1”, com o qual se pode manter o resultado final com o mesmo índice dos operandos originais, reduzindo o resultado a partir deste polinômio [08] [09].

3.2. O Cifrador

No início do processo de cifragem, a entrada é copiada para a matriz de estado. Depois da rodada inicial de adição da chave, a matriz de estado é transformada pela implementação da função de rodada 10, 12, ou 14 vezes (dependendo do tamanho da chave), com a rodada final sendo diferente das anteriores. Após rodada final, a matriz de estado é copiada para a saída e assim termina-se o processo. A função de rodada é parametrizada usando o escalonamento de chaves que consiste em uma matriz unidimensional de palavras de quatro bytes derivadas da chave original, produzido pela função de expansão de chaves.

O cifrador é descrito em pseudocódigo na figura 3.4. As transformações individuais - SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() - processam a matriz de estado como está descrito a seguir. Na figura 3.3 a matriz *w[]* contém o vetor de escalonamento das chaves que será usado na cifragem, todas as rodadas *Nr* são idênticas com exceção da rodada final que não inclui a transformação MixColumns() [08] [09].

```
01 Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
02 begin
03     byte  state[4,Nb]

05     state = in

07     AddRoundKey(state, w[0, Nb-1])

09     for round = 1 step 1 to Nr-1
           SubBytes(state)
           ShiftRows(state)
           MixColumns(state)
           AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
14     end for

16     SubBytes(state)
       ShiftRows(state)
18     AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

20     out = state
21 end
```

Figura 3.4 - Pseudocódigo para o cifrador [08] [09].

Na figura 3.4, no pseudocódigo do cifrador, temos na linha 01 a declaração da chamada da função, onde estão declarados os parâmetros (tamanho da chave de criptografia, do bloco de entrada e saída dos dados) que devem ser passados para a função a fim de que se obtenha o resultado correto. Já entre as linhas 02 e 05 temos o início função e a declaração de algumas variáveis que serão utilizadas, e na linha 07 a primeira interação da função onde a

chave inicial é adicionada (XOR) com a matriz de estado. Entre as linhas 09 e 14 temos as execução das rodadas propriamente ditas, com a chamada de cada sub-função, e nas as linhas 16, 17 e 18, a rodada final que difere das rodadas básicas anteriores por não executar a função MixColumns(). Assim na linha 20 obtemos o resultado de retorno da função e na linha 21 a sua finalização.

3.2.1. Transformação SubBytes()

A transformação *SubByte* é uma substituição não linear que opera em cada byte da matriz de estado independentemente. A função que implementa a tabela de substituição (S_box*)* usada nesta transformação é inversível e é construída pela composição de duas transformações. Executando uma multiplicação inversa em *Galois Field* (2^8) [10][11], que é a base matemática deste algoritmo, temos a função polinomial convertida em representação matricial da seguinte forma na figura 3.5:

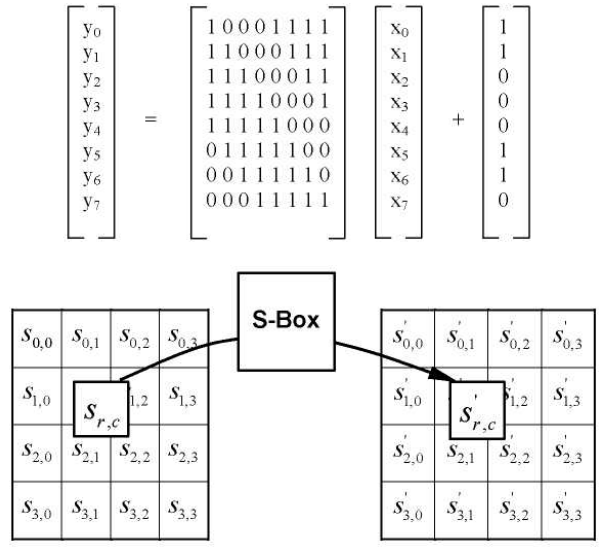
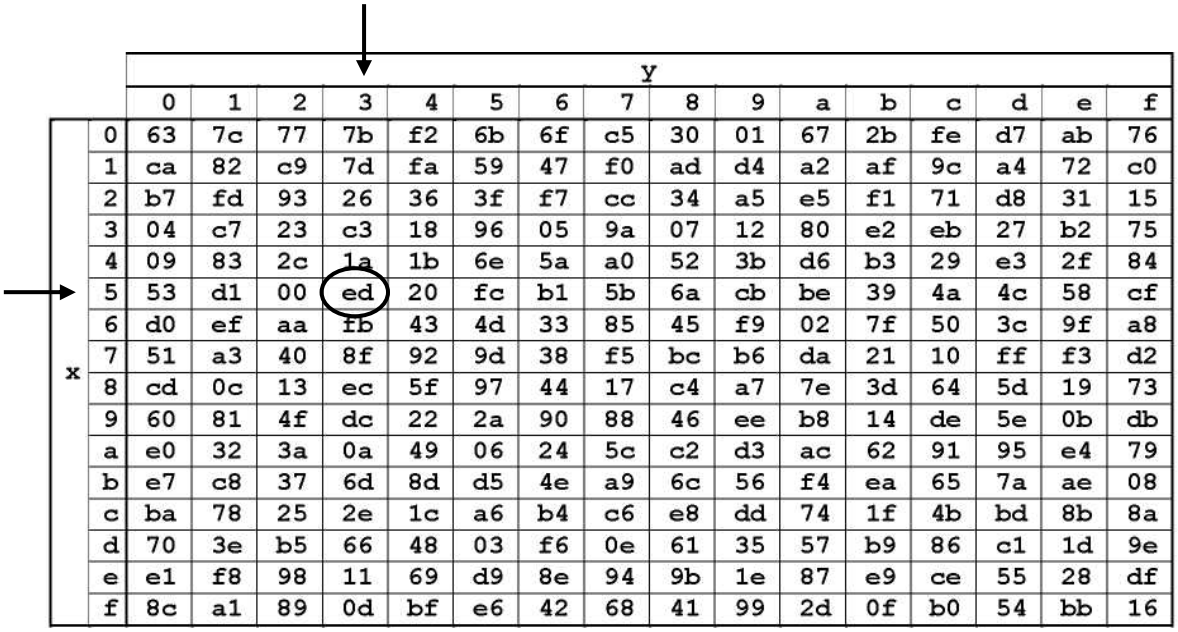


Figura 3.5 - SubBytes() aplica a *S_box* para cada byte da matriz de estado[08] [09].

Ao invés de aplicarmos a operação matemática para cada byte toda vez que efetuarmos a chamada da função, podemos simplesmente utilizar uma tabela pré-preenchida [08][09] e utilizá-la como *S_box* (tabela de substituição) na transformação SubBytes(). A seguir na figura 3.6 é apresentada em formato hexadecimal a tabela pronta da função *S_box*. Com exemplo, se $s_{1,1}$ = {53}, o valor de substituição será determinado pela interseção da linha ‘5’ com a coluna ‘3’ na figura 3.6. Isto resultará em $s'_{1,1}$ = {ed}.



x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 3.6 - S_box: substituição de valores pelo byte xy (hexadecimal) [08] [09].

3.2.2. Transformação ShiftRows()

Nesta transformação, as linhas da matriz de estado são alteradas (deslocadas para a direita) ciclicamente sendo C1, C2 e C3 o deslocamento que se deve aplicar nas linhas da matriz de estado onde a primeira linha não é alterada, a segunda linha é deslocada em "*C1"* bytes, conforme apresentado na figura 3.7, a terceira linha é deslocada em "*C2"* bytes e a quarta linha é deslocada em "*C3"* bytes. Estes parâmetros *C1*, *C2* e *C3* dependem do número de bytes (*Nb)* da coluna [09].

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Figura 3.7 - Parâmetros para a quantidade de bytes deslocados no ShiftRow

A figura 3.8 mostra o efeito da transformação ShiftRow em uma matriz de estado, onde se pode ver como o deslocamento cíclico das colunas afetará os bytes na matriz de estado.

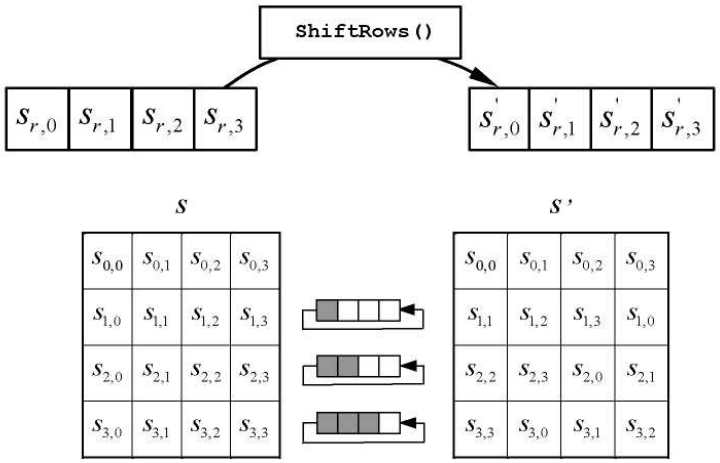


Figura 3.8 - Transformação ShiftRow() em uma matriz de estado.

3.2.3. Transformação MixColumns()

Para se realizar esta função, as colunas de uma matriz de estado são consideradas como polinômios do tipo *Galois Field* (2^8) [10] [11], denotadas por a(x) e são multiplicadas por um polinômio reduzido em módulo por "x^4+1". Segundo a norma [09] o polinômio para multiplicação (a(x)) pode ser obtido por:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

O resultado da multiplicação pode ser denotado por a(x) e a multiplicação pode ser exemplificada pela expressão vista abaixo:

$$s'(x) = a(x) \otimes s(x):$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \textbf{PARA } 0 \le c < Nb.$$

Como resultado desta multiplicação matricial, os quarto bytes na coluna são modificados como a seguir:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

Assim teremos, após a multiplicação, uma alteração nas colunas da matriz de estado, como visto na figura 3.9, esta função opera coluna por coluna da matriz de estado.

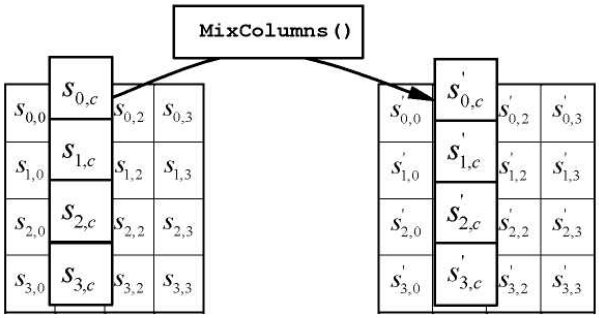


Figura 3.9 - MixColumns() opera a matriz de estado coluna por coluna.

3.2.4. Transformação AddRoundKey()

Esta transformação consiste somente em aplicar uma operação lógica “ou exclusivo” (XOR) na matriz de estado, usando a matriz proveniente da função de escalonamento de chaves. A ação de transformação é ilustrada na figura 3.10 onde o índice “*l”* que identifica a coluna na matriz de estado que será utilizada, é igual ao produto do índice da rodada atual, pelo número de bytes da matriz (*l=round * Nb).* Sendo assim pode-se verificar que esta função realiza um XOR coluna por coluna entre a matriz de estado atual (determinada pela rodada) e a chave de rodada obtida pela função de expansão de chaves.

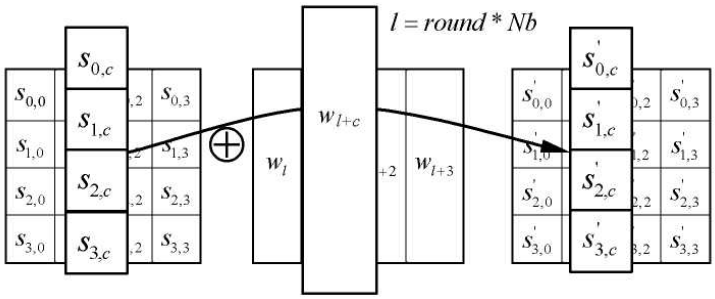


Figura 3.10 - Transformação AddRoundKey.

3.3. Geração da Chave

As inserções de chave no meio do algoritmo são feitas através da função *RoundKey* que são provenientes do processamento da chave. Este processamento da chave é tratado por uma função de escalonamento de chave que possui dois componentes: Expansão da Chave e Seleção da *RoundKey*. O número total de *RoundKeys* necessário é igual ao tamanho do bloco, multiplicado pelo número de rodadas mais um dividido por “*Nb*8”*. Por

exemplo: um bloco de 128 bits, 10 rodadas e *Nb* = 4 necessita de 1408 bits ou 44 *RoundKeys* de 32 bits cada. As *RoundKeys* são selecionadas da *ExpandedKey (vetor principal de chaves)* de acordo o índice de rodada [08].

3.3.1. Expansão da Chave

A *ExpandedKey* é um vetor linear de palavras de 4-bytes e é denotado por *W[Nb*(Nr+1)]*, onde *Nr* é o numero de rodadas e *Nb* o de colunas da matriz de estado. As primeiras *Nk* (número de colunas da matriz de chaves) palavras possuem a chave inicial, enquanto que as outras, são definidas pela função de expansão de chaves. A função que calcula a *ExpandedKey* depende de *Nk*, existindo uma versão para *Nk* < 6 (para o vetor de chaves ser de 128 bits) e outra para *Nk* > 6 (para o vetor de chaves ser 192 ou 256 bits) [08].

```
01 KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
02 begin
     word  temp
     i = 0
05   while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
08   end while
     i = Nk
10   while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
           temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
           temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
19   end while
20 end
```

Figura 3.11 - Pseudocódigo de expansão da chave [08] [09].

Na figura 3.11 encontra-se o pseudocódigo da função de expansão de chaves onde, na linha 01 é realizada a chamada da função e os parâmetros necessários que devem ser fornecidos para a sua execução. Nas linhas 02,03 e 04 tem-se a inicialização da função, onde se observa a declaração e inicialização de variáveis. Já nas linhas 05, 06, 07 e 08 ocorre a primeira interação da função onde é iniciado o vetor de expansão com o valor da chave original, introduzida no algoritmo. Na linha 09 tem-se a inicialização do contador de interações das rodadas que deverão ser necessárias para a expansão das chaves e entre as linhas 10 e 19 a expansão propriamente dita onde, de acordo com o índice do contador a sub-chave (palavra de 32 bits) sofrerá a ação de determinadas operações para a sua determinação. E por fim, na linha 20, o algoritmo é encerrado.

3.4. Decifrador

O cifrador pode ser invertido e implementado de modo a decifrar a informação gerada anteriormente pelo algoritmo AES. As transformações individuais usadas no decifrador - InvShiftRows(), InvSubBytes(), InvMixColumns(), e AddRoundKey() - são processadas na matriz de estados como apresentado pelo pseudocódigo na figura 3.12. Na decifragem, a matriz produzida pela função deverá ser utilizada de ordem inversa.

```
01 InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
02 begin
     byte  state[4,Nb]
     state = in
05   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
06   for round = Nr-1 step -1 downto 1
        InvShiftRows(state)
        InvSubBytes(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
        InvMixColumns(state)
11   end for
12   InvShiftRows(state)
     InvSubBytes(state)
     AddRoundKey(state, w[0, Nb-1])
15   out = state
16 end
```

Figura 3.12 - Pseudocódigo para o decifrador [08] [09].

Na figura 3.12, tem-se na linha 01 a chamada da função e os parâmetros declarados que devem ser passados para a função. Já entre as linhas 02 e 04 tem-se o início da função e a declaração de variáveis necessárias. Na linha 05 ocorre a primeira interação da função onde é adicionada a chave inicial (XOR) com a matriz de estado. Entre as linhas 06 e 11, tem-se as execução das rodadas propriamente ditas, com a chamada de cada sub-função até que o laço FOR termine. Já nas linhas 12, 13 e 14 ocorre a rodada final que difere das rodadas anteriores por não executar a função InvMixColumns(). Na linha 15 obtem-se o resultado de retorno da função e na linha 16 a sua finalização.

3.4.1. Transformação InvShiftRows()

A função InvShiftRows(), ilustrada na figura 3.13, é o inverso da transformação ShiftRows(). Os bytes nas ultimas três linhas da matriz de estado são ciclicamente deslocados para a esquerda por diferentes números de bytes (offsets). A primeira linha, r = 0, não é deslocada. As três consecutivas são deslocados por *Nb-z(r,Nb)* bytes, onde *z* depende do número da linha.

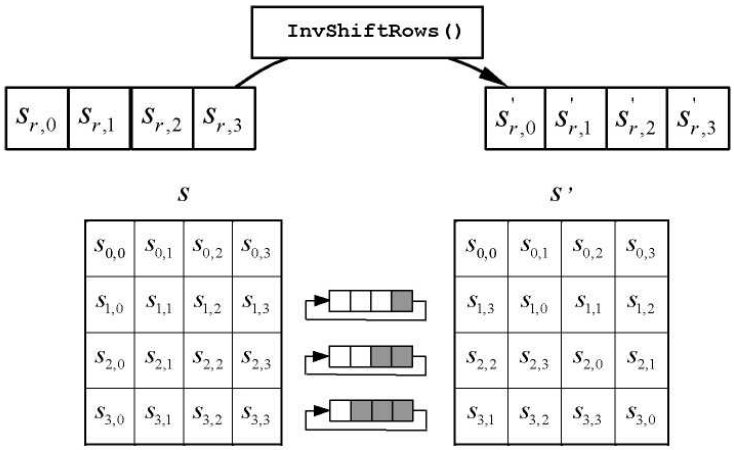


Figura 3.13 - InvShiftRows() deslocamento cíclico de linhas.

3.4.2. Transformação InvSubBytes()

InvSubBytes() é o inverso da transformação de substituição de bytes, onde a S-box inversa é aplicada para cada byte da matriz de estados.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 3.14 - S_box Inversa: substituição de valores pelo byte xy (hexadecimal) [08].

3.4.3. Transformação InvMixColumns()

A função InvMixColumns() é o inverso da MixColumns(), e trabalha na matriz de estados coluna por coluna, tratando cada coluna como um polinômio de quatro termos. As colunas são consideradas como polinômios do tipo *Galois Field* (2^8) [10][11], denotadas por

$a^{-1}(x)$ onde são multiplicadas por um polinômio e reduzidas em módulo por "x^4+1". Na norma [09] o polinômio para multiplicação ($a^{-1}(x)$) pode ser obtido por:

$$a^{-1}(x) = \{0b\}x3 + \{0d\}x2 + \{09\}x + \{0e\}$$

Podemos ver na forma matricial a multiplicação de matrizes como apresentado a seguir:

$$s'(x) = a^{-1}(x) \otimes s(x):$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \textbf{PARA } 0 \le c < Nb.$$

Como resultado desta multiplicação, os quarto bytes na coluna podem ser modificados como a seguir:

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

Assim, após a multiplicação, uma alteração nas colunas da matriz de estado, como visto na figura 3.15, esta função opera coluna por coluna da matriz de estado.

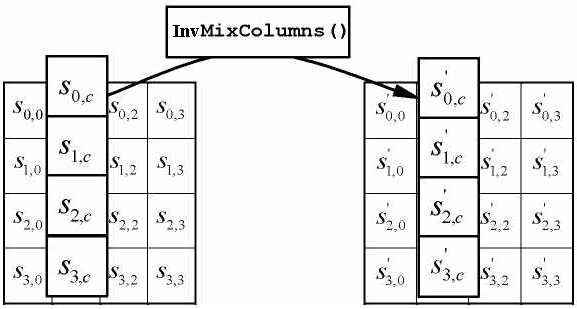


Figura 3.15 - InvMixColumns() opera a matriz de estado coluna por coluna.

3.4.4. Inverso da Transformação AddRoundKey()

A função AddRoundKey(), que está descrita anteriormente, pode ser invertida desde que se aplique uma operação "ou exclusivo" (XOR) com o vetor de chave que foi produzido pela função de escalonamento de chaves.

No caso do processo de decifragem a ordem de utilização do vetor de chaves deve também ser invertida para que assim o processo funcione corretamente [10] [11].

Capítulo 4

Dispositivo de Lógica Programável (FPGA)

Atualmente, uma técnica de desenvolvimento de circuitos integrados, tem oferecido uma opção de implementação de hardware que é muito mais flexível do que os já consagrados *Circuitos Integrados* de *Aplicações Específicas* (ASICs - Application Specific Integrated Circuits). Esta técnica utiliza projeto com linguagem de descrição de hardware e *Dispositivos de Lógica Programável* (os mais utilizados: FPGAs - Field Programmable Gate Arrays) [12].

A tecnologia FPGA foi escolhida para implementar o algoritmo de criptografia AES em hardware devido a varias razões, entre elas a substituição de uma função do algoritmo criptográfico por outra (revisão de versões), que é trivial em software, mas não está disponível em um projeto de hardware convencional (ASIC). Assim, as soluções que permitem revisões de hardware com se fossem software, podem oferecer uma grande melhoria de desempenho quanto à velocidade, segurança, e custo de desenvolvimento.

A agilidade refere-se ao fato de que o mesmo FPGA pode ser reprogramado em tempo de execução para suportar diferentes algoritmos. Outro fator chave que favorece o uso de FPGA é a possibilidade de abstração do funcionamento elétrico dos circuitos que, como poderá ser visto, permite implementar em hardware um algoritmo de criptografia (abstração matemática), sem se conhecer a fundo o funcionamento de transistores. O menor

tempo usado no desenvolvimento do projeto, a escalabilidade de segurança e os parâmetros de arquiteturas variáveis também são fatores importantes para definição por esta tecnologia [13].

4.1. FPGAs - Visão Geral

Um dispositivo de lógica programável FPGA (Field Programmable Gate Array) é um pedaço de silício que contém uma matriz de blocos lógicos configuráveis (*CLBs)*, *flip-flops* e interconexões programáveis entre os blocos lógicos. Diferentemente de um *Circuito Integrado de Aplicação Específica* (ASIC - Application Specific Integrated Circuit), que pode executar uma função única e direcionada, um FPGA pode ser reprogramado para executar uma função diferente após alguns microssegundos [12].

Os FPGAs foram introduzidos como uma alternativa aos circuitos integrados customizados, para a implementação de sistemas em um único chip e para fornecer flexibilidade de reprogramação para o usuário, resultando na melhoria de densidade quanto comparado a componentes discretos (aproximadamente 10 vezes). Outra vantagem do FPGA sobre o *CustomICs* é que, com o auxilio de ferramentas *CAD (Computer Aid Design)*, os circuitos passaram a ser implementados em um intervalo de tempo menor (sem processo físico de *layout*, sem produção de mascara, e sem fabricação de circuito integrado específico)[12]. Na figura 4.1 apresentamos parâmetros de comparação entre as tecnologias de projeto de circuitos integrados sendo *NRE: Non-recurring Engineering Costs* e *TTM: Time To Market.*

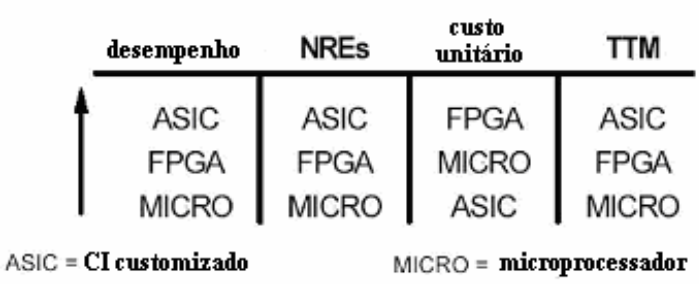


Figura 4.1 - Análise comparativa [12] [13]

Antes que seja programado, um circuito integrado não pode se comunicar com os dispositivos que o cercam, isto é tanto ruim quanto bom, mais vai lhe permitir muita flexibilidade na utilização diária, aumentando o seu poder e complexidade de programação. A

capacidade de reprogramação do FPGA, o levou a ser largamente utilizado por projetistas de hardware para circuitos de protótipos. Alguns anos atrás os FPGAs começaram a conter bastantes recursos, tornando-os muito interessantes para a comunidade e as empresas que desenvolvem sistemas de computação de alta desempenho (HPC - High Performance Computing).

Recentemente os vendedores de hardware de HPC começaram a oferecer soluções que incorporam FPGAs em sistemas HPC [12], onde pode-se utiliza-los como co-processadores ou outros sistemas auxiliares de desempenho, acelerando *key kernels* (núcleos de sistemas operacionais) dentro de certas aplicações. Como um FPGA é formado por uma matriz de CLBs, esta matriz pode ser simplesmente estendida para um processo de fabricação de 65nm ou mais novo, evitando a necessidade de re-engenharia intensa da arquitetura a cada nova geração de um chip, como no caso dos *ASICs* [13].

A onda recente no interesse de FPGAs da comunidade HPC, vem no momento em que os microprocessadores convencionais estão lutando para acompanhar a *lei de Moore*. Esta redução de ganho de desempenho em microprocessadores, junto com o aumento do custo para manter exigências mínimas de desempenho, levou a um aumento do interesse em qualquer tecnologia que possa oferecer uma alternativa mais rentável e viável. O uso de FPGAs em sistemas HPC pode prover três vantagens distintas sobre os grupos de computação convencional. Inicialmente, FPGAs consomem menos energia do que muitos microprocessadores convencionais; segundo, usando FPGAs como aceleradores auxiliares dos sistemas de computação, pode-se aumentar significativamente a densidade de *computação*; terceiro, FPGAs podem fornecer um aumento significativo no uso e desempenho de certas aplicações [12].

Internamente, os FPGAs consistem de interconexões de chaves programáveis eletricamente, conforme a figura 4.2, que os diferenciam dos circuitos integrados customizados que são fabricados com interconexões fixas de metal entre os blocos lógicos. O FPGA permite um modo de configurar as interconexões entre os blocos lógicos e a função de cada bloco lógico [11]. O FPGA pode ser configurado de forma a prover desde funcionalidade de um transistor até a complexidade de um microprocessador. Isto pode ser implementado de diferentes formas, com lógica combinacional ou lógica seqüencial: 1)Par de Transistores; 2)Portas combinacionais *NAND* ou *XOR*; 3) Tabelas verdade de *n-input*; e 4)Multiplexadores.

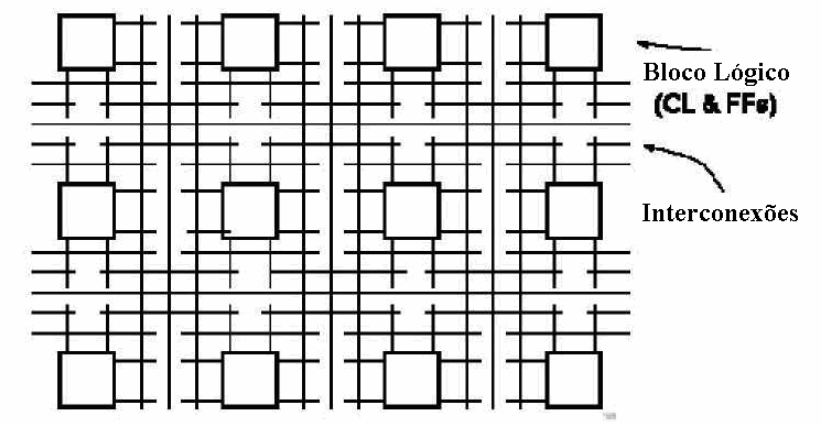


Figura 4.2 - Versão simplificada da arquitetura interna de um FPGA [13].

A densidade dos blocos lógicos usados num FPGA depende do projeto, do comprimento e do número de segmentos (trilhas) usados para o roteamento dos circuitos. O número de segmentos para interconexão tipicamente é uma troca entre a densidade de blocos lógicos usados e o total de área usada para o roteamento.

4.2. O Bloco Lógico

O bloco lógico em um FPGA pode ser implementado de modo que se pode abstrair no número de entradas e saídas, a complexidade de funções lógicas que ele pode implementar e o número total de transistores que ele consome. A seguir uma descrição de como funciona os blocos lógicos de alguns fabricantes

FPGA Crosspoint: composto de dois tipos de blocos lógicos, onde um é o par de transistores. Os blocos lógicos são executados em linhas paralelas como mostrado na figura 4.3 a baixo:

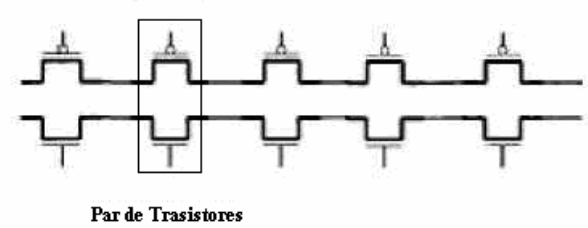


Figura 4.3 - Par de transistores em um FPGAs *Cross-point* [13].

O segundo tipo de blocos lógicos é a lógica de memória de acesso aleatório (RAM) que pode ser usada para implementar uma memória de acesso aleatório.

FPGA *Plessey*: aqui o bloco de construção básica é uma porta *NAND* de duas entradas, como na figura 4.4, que conecta outro elemento para implementar a função desejada.

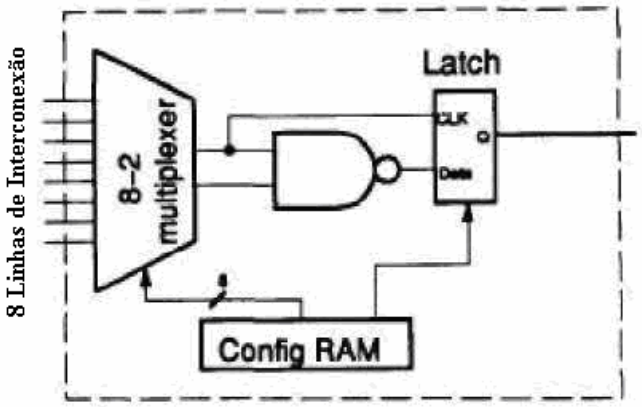


Figura 4.4 - Bloco lógico para FPGAs *Plessey* [13].

FPGA Actel: Se as entradas de um multiplexador forem conectadas a uma constante ou a um sinal, isto pode ser usado para implementar funções lógicas diferentes com na figura 4.5. Por exemplo, um multiplexador de 2 entradas com entradas *a*, *b* e *select*, poderá implementar a função ac+bc. Se b=0 então ele implementará ac, e se a=0 ele implementará bc.

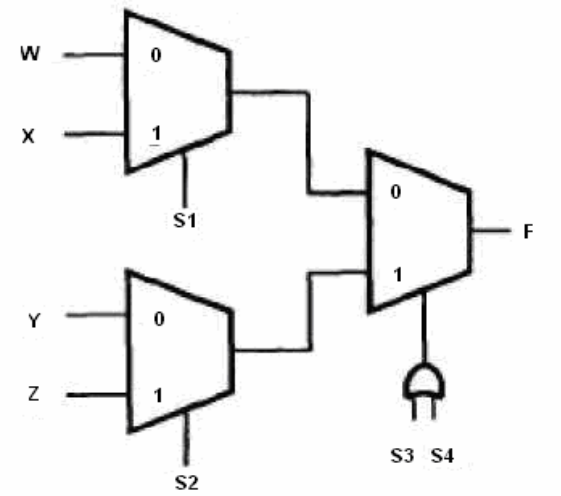


Figura 4.5 - Bloco lógico para FPGAs *Actel* [13].

Tipicamente um bloco lógico Actel é formado de um número múltiplo de multiplexadores e portas lógicas.

FPGA Xilinx: No bloco lógico Xilinx a *Look up table* (LUT) é usada para implementar diversas funcionalidades. As linhas de entrada habilitam esta *LUT* e a sua saída fornece o resultado da função lógica implementada. A *LUT* é implementada usando uma SRAM, e uma função lógica de k-entradas pode ser implementada usando $2^{\wedge k}$ de tamanho de uma SRAM. O número de funções possíveis para a LUT de entrada k é de $2^{\wedge 2k}$. A vantagem de tal arquitetura é que ele suporta a implementação de muitas funções lógicas, contudo a desvantagem é o número excepcionalmente grande de células de memória necessárias para implementar um bloco lógico no caso do número de entradas ser grande como na figura 4.6.

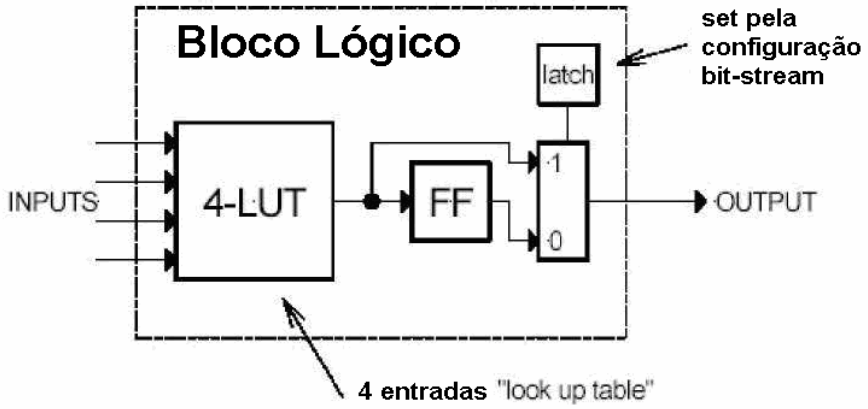


Figura 4.6 - Bloco lógico para FPGAs *Xilinx* [13].

FPGA Altera: O bloco lógico do FPGA Altera desenvolveu-se antes do aparecimento dos atuais *PLDs*. Ele é composto de uma entrada bem larga (até 100 entradas) de portas *AND* que são alimentadas por portas *OR* de 3 a 8 entradas. Se o transistor de porta flutuante estiver baseado em uma chave programável, é habilitada qualquer trilha (roteamento) vertical que passe perto de uma porta *AND* e possa ser usado como entrada da mesma. A vantagem de uma implementação baseado numa quantidade grande de portas de entrada *AND* consiste que, poucos blocos lógicos podem implementar uma funcionalidade completa e assim se reduz da área necessária para implementar interconexões do FPGA. Por outro lado, temos a desvantagem no uso de baixas densidades de blocos lógicos em um projeto que necessite de menos lógica de entrada.

Outra desvantagem é o uso de dispositivos de *pull-up* (portas *AND*) que consomem energia estática. Para melhorar isto, os fabricantes fornecem blocos lógicos de baixo consumo de energia que provocam o custo maior no atraso do dispositivo. Tais blocos lógicos podem ser usados em caminhos não-críticos [12] [13].

4.3. Negociação - Tamanho vs Desempenho

O tamanho dos blocos lógicos passa por um grande impasse, quando se avalia entre a densidade de blocos lógicos e a otimização de área utilizada no FPGA, visto que isto é imprescindível para o desempenho dos dispositivos. Um grande bloco lógico implementa mais lógica e assim, um menor número de blocos lógicos e necessário para implementar uma funcionalidade no FPGA. Por outro lado, um grande bloco lógico consumirá mais espaço. Portanto, o ótimo tamanho do bloco lógico é aquele que usa menos números de blocos lógicos na implementação de uma funcionalidade, consumindo o menor espaço possível.

A área lógica ativa é geralmente menor do que a área lógica total devido à presença das conexões programáveis. A área lógica total é a soma de área lógica ativa e da área consumida por conexões programáveis. A área roteável em um FPGA é tipicamente mais do que a área ativa. Ela é 70 a 90 por cento da área total em um FPGA [13].

4.4. Técnicas de Roteamento de FPGAs

A arquitetura de roteamento compreende chaves programáveis e trilhas. O roteamento fornece uma conexão entre os blocos de entrada-saída e os blocos lógicos, e entre um bloco lógico e outro. O tipo de arquitetura de roteamento decide a área consumida pelo roteamento e a densidade de blocos lógicos. Técnicas de roteamento usadas em um FPGA basicamente decidem a quantidade de área a ser usada pelas trilhas e as chaves programáveis comparando-as com a área consumida por blocos lógicos.

Uma trilha pode ser descrita como dois pontos finais de uma interconexão sem chaves programáveis entre eles. Uma seqüência de uma ou mais trilhas em um FPGA pode ser chamada de pista. Tipicamente um FPGA tem blocos lógicos, interconexões e blocos de entrada/saída. Os blocos de entrada/saída estão na periferia dos blocos lógicos e interconexões. Os blocos de conexão são unidos aos blocos lógicos e, dependendo da exigência do projeto, um bloco lógico é conectado a outro [12][13].

• **<u>Arquitetura de roteamento Xilinx</u>:** No roteamento da Xilinx, as conexões são feitas do bloco lógico ao canal de interconexão por um bloco de conexão. Como a tecnologia SRAM é usada para implementar uma *LUT*, as conexões locais são grandes.

O bloco lógico é rodeado por blocos de conexão. Estes unem os pinos dos blocos lógicos com a trilha de roteamento. Como na Figura 4.7, os transistores são usados para implementar conexões de pinos de saída enquanto, pinos multiplexadores de entrada guardam o número de células SRAM requeridas.

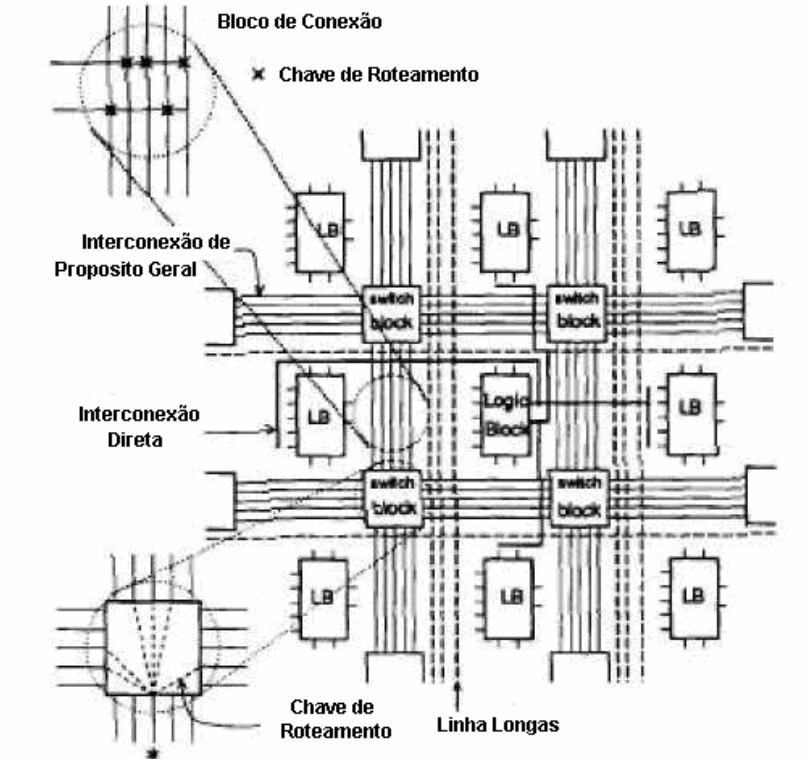


Figura 4.7 - Arquitetura de roteamento da *Xilinx*[13].

• **<u>Arquitetura de roteamento Actel:</u>** O projeto da Actel tem mais segmentos de trilhas na direção horizontal do que na direção vertical. Os pinos de entrada conectam todas as pistas do canal que estão no mesmo lado que o pino. O pino de saída estende-se através de dois canais sobre o bloco lógico e dois canais abaixo dele.

O pino de saída pode ser unido aos quatro canais que ele cruza. Os blocos de chaveamento são distribuídos pelos canais horizontais. que todas as pistas verticais podem fazer conexão com cada pista horizontal incidente como visto na figura 4.8 [13].

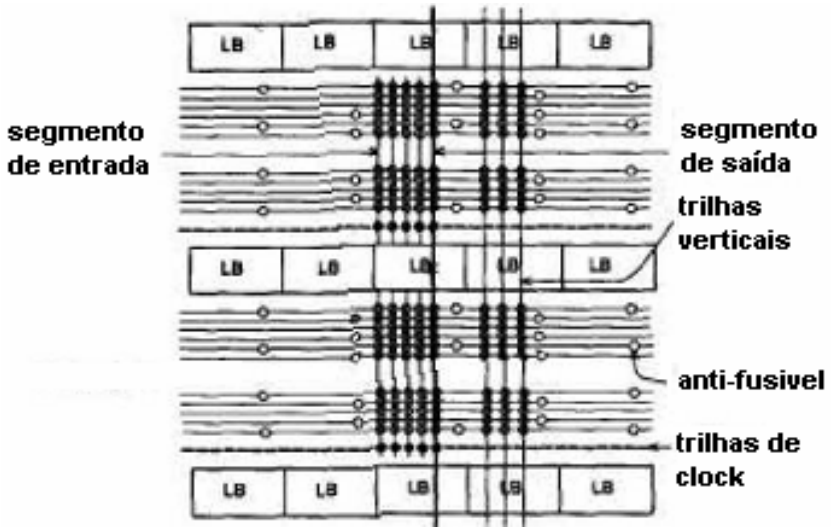


Figura 4.8 - Arquitetura de roteamento da *Actel* .

• **Arquitetura de roteamento Altera:** A arquitetura de roteamento da Altera tem dois níveis de hierarquia. No primeiro nível da hierarquia, 16 ou 32 dos blocos lógicos são agrupados em um Bloco de Matriz Lógica (*LAB*), sendo a estrutura do *LAB* muito semelhante a um PLD. As conexões são formadas usando-se EPROM como transistores de porta flutuante.

O canal é o conjunto de trilhas que percorrem verticalmente o FPGA. As pistas são usadas para as seguintes conexões: conexões de saídas dos blocos lógicos do *LAB*; conexões de expansão; conexões de saídas de blocos lógicos; conexões de conectores de I/O.

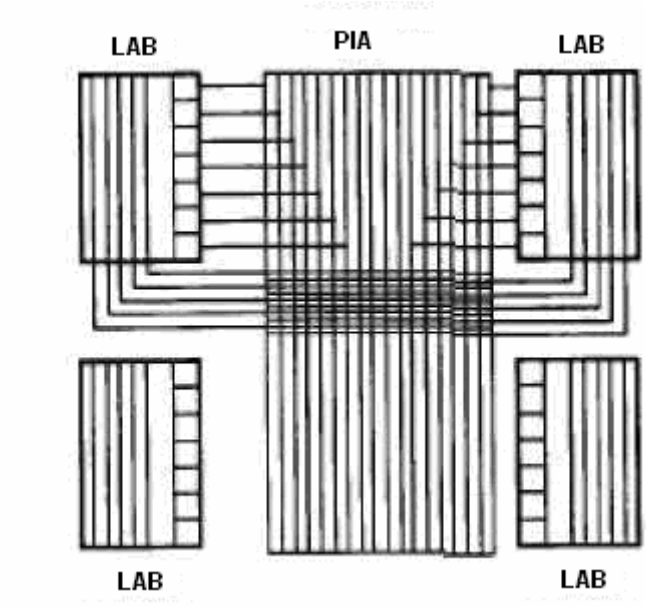


Figura 4.9 - Arquitetura de roteamento global da *Altera*. (MAX 5000) [13]

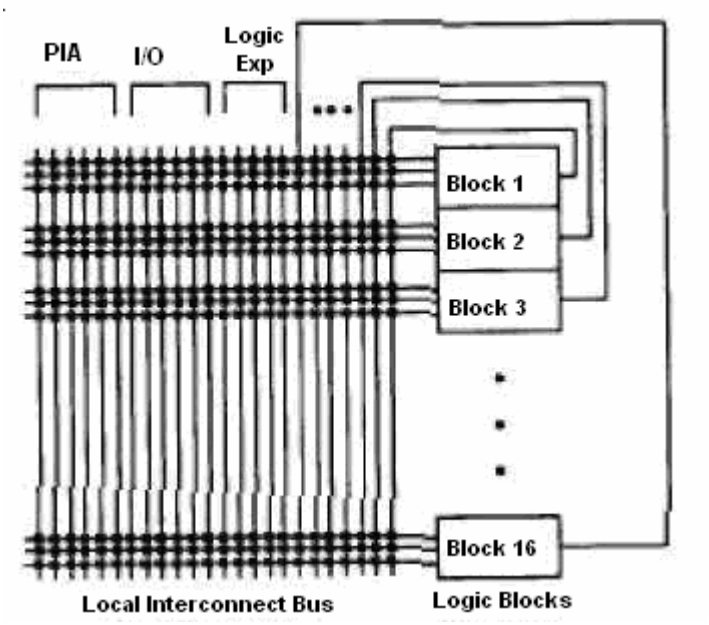


Figura 4.10 - Arquitetura de roteamento local da *Altera*. (MAX 5000) [13]

4.5. Classificação Estrutural de FPGAs

O arranjo dos blocos é específico para cada fabricante que o desenvolve segundo suas opções de mercado e comercialização. Com base no arranjo interno, os FPGAs podem ser dividido em três classes:

- **Matrizes Simétricas**

Esta arquitetura compõe-se de elementos lógicos (chamados CLBs) arranjados em linhas e colunas de uma matriz, juntamente com um *layout* de interconexão entre eles. Esta matriz simétrica é rodeada de blocos de entrada/saída que são conectados ao mundo exterior como na figura 4.11.

Cada CLB é formado de uma *LUT* (*Lookup Table*) de n-entrada e um par de flip-flops programáveis. Interconexões fornecem o caminho de roteamento, e a direta interligação entre elementos lógicos adjacentes produz menor atraso do que se comparados com as interconexões de propósitos geral [13].

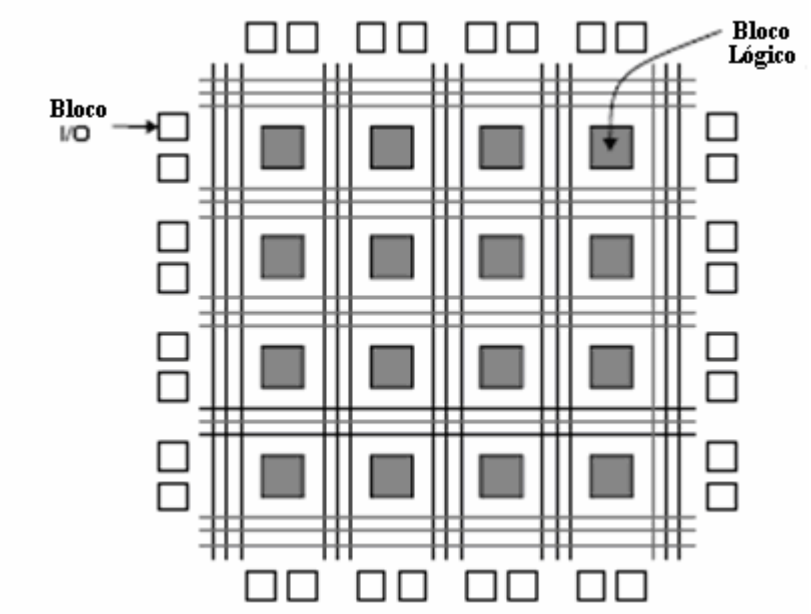


Figura 4.11 - FPGAs, de matrizes simétricas.

- **Arquitetura Baseada em Linhas**

A arquitetura baseada em linhas é formada de linhas alternadas de módulos lógicos programáveis que interligam as trilhas. Os blocos de entrada/saída são localizados na periferia das linhas. Uma linha pode ser unida a linhas adjacentes via interconexão vertical. Os módulos lógicos podem ser implementados com diversas combinações. Os módulos combinatórios só contêm elementos combinacionais, já módulos de seqüenciais contêm ambos, elementos combinacionais junto com flip-flops. Estes módulos seqüenciais podem implementar funções combinatórias e seqüenciais complexas como visto na figura 4.12.

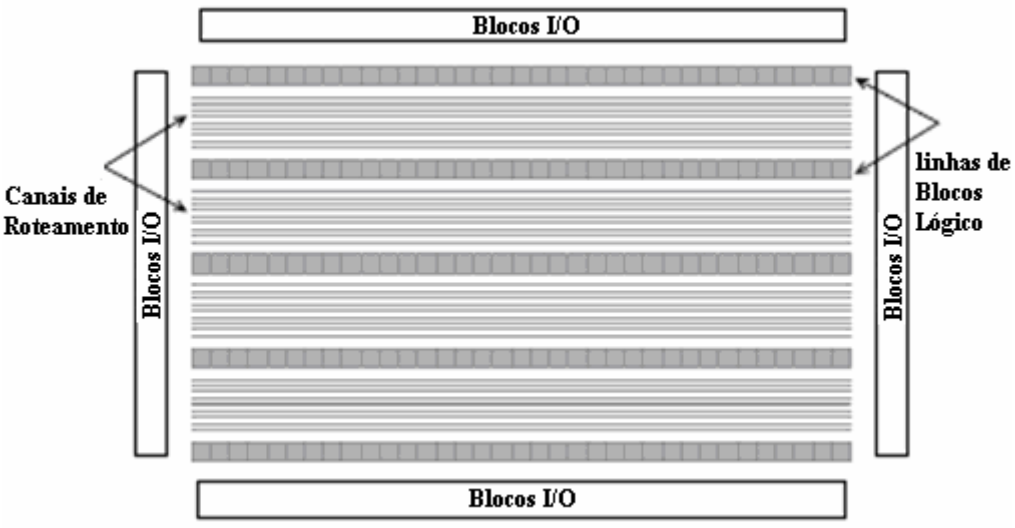


Figura 4.12 - FPGAs, arquitetura baseada em linhas [13].

• **PLDs Hierárquicos**

Esta arquitetura é projetada da maneira hierárquica com o nível superior contendo apenas blocos lógicos e interconexões. Cada bloco lógico contém um número de módulos lógicos, e cada módulo lógico tem elementos combinatórios, como também elementos funcionais seqüenciais. Cada um desses elementos funcionais é controlado por uma memória pré-programada. Na figura 4.13 [13], vimos que a comunicação entre blocos lógicos é realizada por meio de uma matriz de interconexão programável.

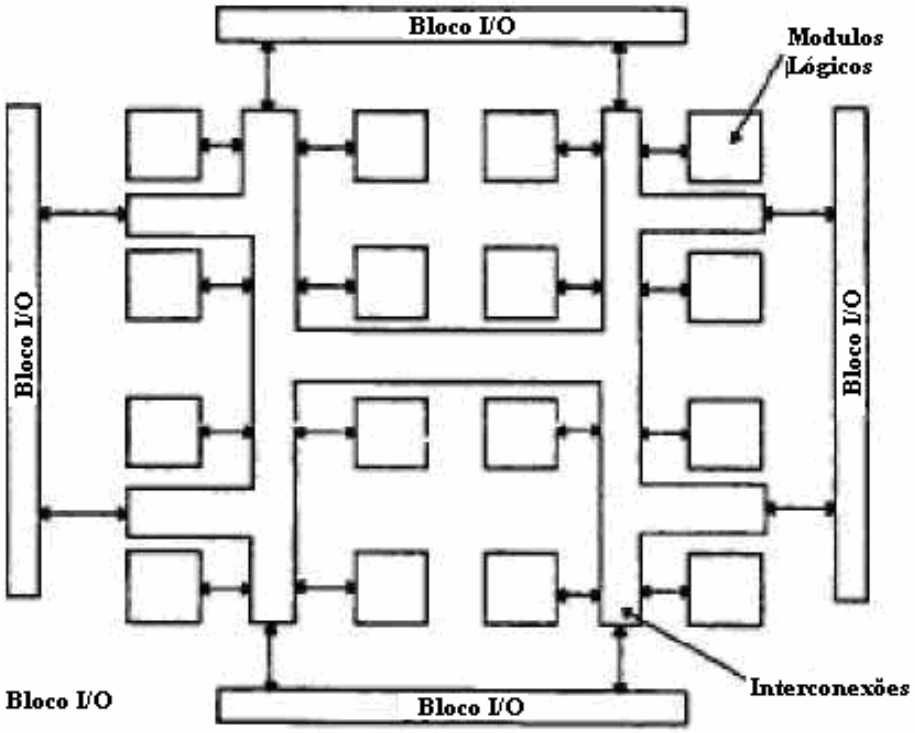


Figura 4.13 - FPGAs, PLD hierárquico.

4.6. Metodologias de Programação

Os comutadores eletricamente programáveis são usados para programar um FPGA. A desempenho de um FPGA, em termos de área e densidade lógica é uma função das propriedades desses comutadores. As propriedades desses comutadores programáveis é que fazem a diferença com relação à resistência, capacitância parasita, volatilidade dos dados, reprogramação, tamanho, etc. As técnicas empregadas são:

• **Tecnologia de Programação SRAM**

As células estáticas *RAM* são usadas para controlar portas de passagem ou multiplexadores. Para usar uma porta de passagem como uma chave fechada, o valor 1 (um) é armazenado na célula *SRAM*. A figura 4.14 apresenta o uso de *SRAM*:

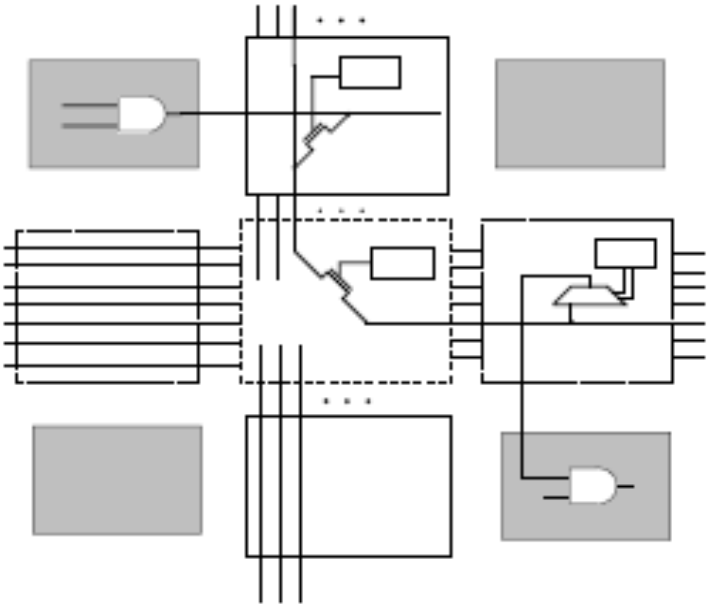


Figura 4.14 - FPGAs, programação SRAM [13].

Para usar uma *SRAM* como multiplexador, o estado dos valores de controle armazenados na *SRAM* decide quais as entradas do multiplexador deverão ser unidas à saída. A vantagem é que a *SRAM* provê rápida reprogramação e tecnologia de fabricação. A desvantagem é o espaço consumido pelos transistores para implementar a célula de memória.

• **Programação de Porta Flutuante**

A tecnologia, encontrada nos EPROM's que são apagadas com luz ultravioleta e nos EEPROMs apagáveis eletricamente, é usada em alguns FPGAs do fabricante Altera com na figura 4.15. A chave programável é um transistor que pode ser desabilitado. Aqui novamente, a vantagem é a reprogramação, mas temos uma outra, nenhuma fonte de memória permanente externa (ROM), é a necessária para programar o dispositivo na inicialização.

A desvantagem é o alto consumo de energia devido a estática do resistor de *pull up* e alta *ON-resistance* do transistor EPROM.

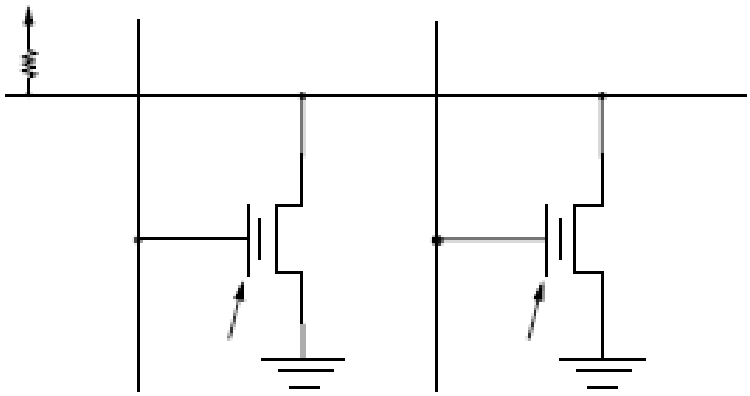


Figura 4.15 - FPGAs, programação de porta flutuante [13].

• **Tecnologia Anti-fusível (Antifuse programming methodology)**

Um anti-fusível é um dispositivo de dois terminais com estado não programado, que fornece uma resistência muito alta entre os seus terminais. Para criar uma conexão de baixa resistência entre os dois terminais, uma alta voltagem é aplicada através dos terminais para queimar o anti-fusível. Um bit extra do circuito é necessário para programar um anti-fusível. A tecnologia de anti-fusível é usada por FPGA'S da Actel, QuickLogic e Crosspoint. A vantagem do anti-fusível é o tamanho relativamente pequeno e a redução de área, que é anulada pela área consumida pelo circuito extra necessário para programar o FPGA. Outra vantagem é a baixa resistência em série e a baixa capacitância parasita [13].

4.7. FPGA Fluxo de Projeto

Uma das vantagens mais importantes no projeto baseado em FPGA é que o usuário pode projetá-lo usando ferramentas *CAD,* fornecidas pelas companhias de automação de projeto. Geralmente o fluxo de projeto de um FPGA inclui os seguintes passos [13]:

• **Projeto de Sistemas**

Nesta etapa o projetista tem que decidir, que parte da sua funcionalidade tem de ser implementada em FPGA e como integrar aquela funcionalidade ao resto do sistema.

• **Integração de I/O com o resto do sistema**

Os fluxos de entrada/saída do FPGA estão integrados a Placa de Circuito

Impresso (PCB), onde é importante o projeto da *PCB* antes do processo de projeto. Os fabricantes de FPGA fornecem soluções extras de software para o processo de projeto de I/O.

- **Descrição de Projeto**

Projetistas descrevem as funcionalidades do projeto com editores esquemáticos ou usando uma das várias linguagens de descrição de hardware (HDL) como Verilog e VHDL

- **Síntese**

Uma vez que o projeto foi definido são usadas ferramentas para implementar o projeto em um determinado FPGA. A síntese inclui otimizações genéricas e otimizações de energia, seguidas por otimizações de locação de elementos lógicos e roteamento. A implementação inclui, dependendo das ferramentas utilizadas, a possibilidade de particionamento do projeto em mais de um dispositivo. A saída da fase de implementação do projeto é o arquivo de *bit-stream*.

- **Verificação de Projeto**

O arquivo de *bit-stream* é alimentado para um simulador que verifica a funcionalidade do projeto e informa erros no comportamento do projeto. Ferramentas de *timing* são usadas para determinar a freqüência máxima de funcionamento do projeto e por fim o projeto é carregando para o dispositivo de FPGA final onde são feitos testes de trabalho no ambiente real.

Capítulo 5

Linguagem Descrição de Hardware (VHDL)

Além das grandes vantagens já vistas anteriormente com o uso de FPGAs em projetos de hardware, normalmente existe a necessidade de migração do projeto para outras plataformas, como os ASICs, ou mesmo FPGAs de outros fabricantes (portabilidade), além da possibilidade de mudança na tecnologia de fabricação dos chips (0,90nm -> 0,65nm). Assim, é necessário que se encontre uma solução integrada de forma que, o desenvolvimento de uma ferramenta de projeto resolva estas questões. Esta necessidade pode ser suprida com o uso de linguagens de descrição de hardware HDL (*Hardware Description Language*). Devendo, com características principais, serem padronizadas internacionalmente e reconhecidas pelos fabricantes e grupos de desenvolvimento.

Um sistema digital pode ser descrito com diferentes níveis de abstração e pontos vista [14]. Um HDL deve modelar com fidelidade, exatidão e descrever um circuito com visões comportamentais e/ou estruturais definidas pela abstração necessária. Como as HDLs são modeladas depois do hardware, a sua semântica e uso são muito diferentes daquelas linguagens de programação tradicionais. Existem diversas linguagens de descrição de hardware disponíveis no mercado, porém devido a sua origem e funcionalidade, sem duvida a VHDL (*VHSIC Hardware Description Language*) é a mais conhecida. O uso de VHDL permite que o projeto tenha portabilidade, fácil compreensão (documentação de projeto) e um ‘time to marketing’ menor para projetos de hardware.

5.1. VHDL - Visão Geral

O VHDL e Verilog são as duas das mais utilizadas HDLs. Embora a sintaxe e "a aparência" das duas línguas sejam muito diferentes, as suas capacidades e os seus alcances são bastante semelhantes. Ambos são padrões industriais e são suportados pela maior parte das ferramentas de *softwares* disponíveis atualmente [15]. O desenvolvimento do VHDL foi alavancado inicialmente pelo Departamento de Defesa dos Estados Unidos como um padrão de documentação de *hardware* no início dos anos 1980 e logo foi transferido ao IEEE (*Institute of Electrical and Electronic Engineers*). O IEEE ratificou o VHDL em 1987, onde se fez referência a esta linguagem como VHDL-87. Sua evolução é apresentada na figura 5.1

Figura 5.1 - Evolução do padrão

Cada padrão IEEE é revisto após alguns anos, além de quando ocorrer fatos novos que justifiquem, pode-se fazer uma reavaliação extraordinária. O IEEE revisou o padrão VHDL em 1993, denominado como VHDL-93. Outras pequenas modificações foram fixadas em 2001, denominado agora o padrão como VHDL-2001, que por sinal não possui nenhuma diferença significativa com relação ao VHDL-93. Como aviso, um sufixo é às vezes acrescentado ao padrão IEEE para indicar o ano que o padrão foi lançado. Depois do lançamento inicial, várias extensões foram desenvolvidas para facilitar vários projetos e exigências de modelagem. Essas extensões são documentadas em vários padrões IEEE [14] [15]:

0 padrão IEEE 1076.1-1999, VHDL Analog and Mixed Signal Extensions (VHDLAMS): define a extensão de modelagem de sinais.

0 IEEE standard 1076.2-1996, VHDL Mathematical Packages: define funções matemática extras para números complexo e reais

0 IEEE standard 1076.3- 1997, Synthesis Packages: define operações aritméticas sobre uma coleção de bits.

0 IEEE standard 1076.6-1999, VHDL Register Transfer Level (RTL) Synthesis: define um subconjunto que é util para a síntese.

0 IEEE standard 1029.1-1998, VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES): define como usar VHDL para troca de informação em ambientes de simulação.

5.2. Conceitos Básicos de VHDL

A fim de se compreender a estrutura de programação VHDL, deve-se inicialmente estudar a forma com a qual esta linguagem define os diferentes processos e componentes que fazem parte de um sistema. Todo componente VHDL é definido como uma entidade (*entity*), o que nada mais é do que uma representação formal de um componente ou de um processo. Em descrições mais simples uma entidade pode ser o próprio projeto, mas em implementações de grandes sistemas, o projeto é composto por diversas entidades distintas.

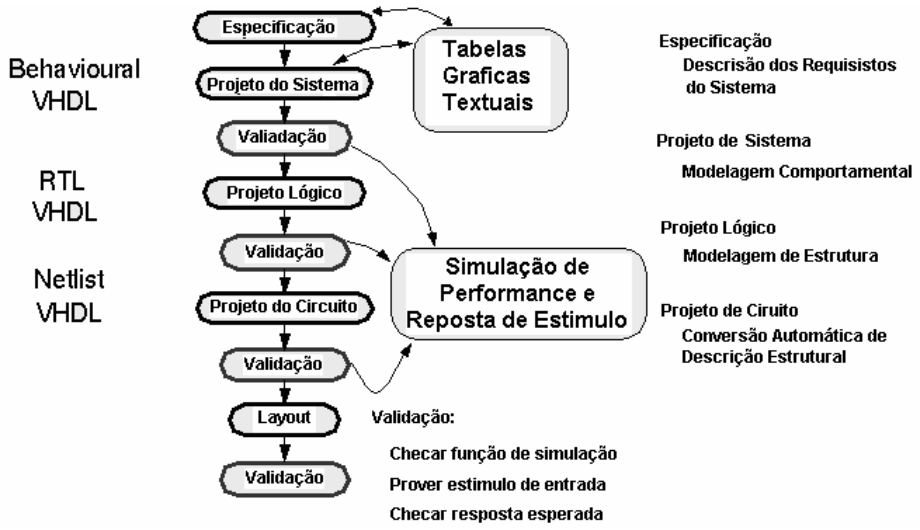


Figura 5.2 - VHDL: Fluxo ASIC [14]

Como apresentado anteriormente na figura 5.2, é possivel ver as principais etapas envolvidas no processo de desenvolvimento de um chip (*ASIC*), onde poderemos estabelecer um ponto ótimo para o uso do VHDL. Conforme o fluxo, temos atividades desde a especificação do projeto, onde se trabalha com um documento de texto simples (linguagem corrente) e alguns recursos gráficos como esquemas e fluxogramas, passando pelo projeto propriamente dito (sistema, lógico e circuito) até a parte de projeto físico e validação. Diferentemente de um fluxo de projeto FPGA, o fluxo ASIC necessita de outras etapas para a sua correta finalização.

Independente da plataforma de projeto escolhida a linguagem VHDL sempre é utilizada nas etapas de desenvolvimento do projeto de sistema e lógico, além de ser a ideal para a parte de teste e verificação dos mesmos. Desta forma podemos utilizar não só no fluxo ASCI, nas áreas de desenvolvimento de projeto comportamental, RTL e *netlist*, como também no fluxo de projeto em FPGA em etapas semelhantes e também em teste e verificação.

O modelo de definição de uma entidade em VHDL segue uma estrutura bem específica, composta por duas partes, que serão apresentadas a seguir [16]:

-Declaração de entidade;
-Corpo de arquitetura

5.2.1. Declaração de Entidade

Declaração de uma entidade é a definição de uma estrutura funcional, que pode ser um componente completo ou só parte de um sistema com a identificação apropriada, de modo a permitir que a mesma seja utilizada posteriormente. Normalmente o nome da entidade é declarado após a palavra-chave *entity* e repetido após a palavra-chave *end entity*. Dentro da declaração da entidade é definida a interface do componente de forma similar a definição de pinos de um componente [16].

O instanciamento da entidade é feito através da declaração de portas (*ports*) de interface da *entity* que, além de um nome próprio, deve conter o modo que indica de que tipo é o sinal (de entrada (*in*), saída (*out*) ou bidirecional (*inout*)). O tipo do sinal é que determina

o tipo de informação que irá trafegar pela porta. Inicialmente, são apresentados na declaração da *entity* os nomes dos sinais e a seguir, separados por dois pontos ':', o modo e o tipo dos mesmos. Na definição das portas, podem-se definir sinais que comportem bits, tipos complexos como bytes (8bits) e até tipos compostos como arrays de variáveis, conforme necessidade do projeto. Na figura 5.3 temos um exemplo de uma declaração de entidade bem simples [17].

```
entity modelo is
port (   a, b : in bit;
         c : out bit);
end entity modelo;
```

Figura 5.3 - VHDL: declaração de entidade.

Denominada *modelo,* esta entidade possui duas portas de entrada do tipo bit, declaradas como a e b e uma porta de saída também do tipo bit, declarada como c. Normalmente as portas de entrada de uma entidade são declaradas antes das de saída. Cada declaração de interface é seguido por ponto e vírgula ';', exceto a última. Também é necessário colocar-se ponto e vírgula no final da definição de porta.

Bit é um tipo pré-definido em VHDL, que pode assumir valores '0' e '1'. Este tipo é usado para representar dois níveis lógicos. Na prática, para representar valores digitais reais são utilizadas bibliotecas definidas pela IEEE ou por um vendedor de componentes. É comum se encontrar os tipos STD_LOGIC ou VL_BIT, que são representações de circuito bem mais próximas da realidade que os simples valores '0' e '1'. É possível, em VHDL, a definição de valores iniciais para portas, como na figura 5.4, durante a definição da entidade [17].

```
entity modelo is
port (   a, b : in bit := '1';
         c : out bit);
end entity modelo;
```

Figura 5.4 - VHDL: declaração de entidade com atribuição inicial.

5.2.2. Corpo de Arquitetura

O corpo de arquitetura de uma entidade define o seu funcionamento interno, a partir de uma série de instruções de operação em um ou mais processos. Um processo é como uma unidade básica descritiva de um comportamento. Um processo é executado normalmente em resposta a mudança de valores de sinais, e usa os valores correntes destes sinais e variáveis para determinar os novos valores de saída. È importante saber que mais de um processo pode ser executado simultaneamente, provocando assim processos concorrentes, sem esquecer que são permitidas estruturas de *loop* como If-Them-Else; Do-While e outras com a intenção de se flexibilizar mais os recursos da linguagem de programação. Em VHDL existem dois conceitos de modelamento da descrição funcional de uma entidade [17] [18]. São eles:

- Modelo comportamental
- Modelo estrutural

• **Modelo Comportamental**

Pode-se montar um corpo de arquitetura comportamental descrevendo sua função a partir sentenças de processo, que são conjuntos de ações a serem executadas. Os tipos de ações que podem ser realizadas incluem expressões de avaliação, atribuição de valores a variáveis, execução condicional, expressões repetitivas e chamadas de subprogramas [18].

A declaração do comportamento de uma entidade é feito através da programação de seus algoritmos com base nos diversos comandos e estruturas da linguagem. A seguir na figura 5.5 tem-se um exemplo de uma declaração de arquitetura para a entidade *latch*.

```
architecture opera of latch is     -- Declaração de arquitetura
begin
        q <= r nor nq;             -- Saídas q e nq recebem resultado de
        nq <= s nor q;             -- operação NOR com as entradas r e s
end opera;
```

Figura 5.5 - VHDL: declaração de arquitetura.

A primeira linha da declaração indica que esta é a definição de uma nova estrutura denominada *opera*, que pertence à entidade denominada *latch*. Assim esta arquitetura descreve a operação da entidade latch. As linhas entre as diretivas de começo (begin) e fim (end) descrevem a operação do latch.

Assim como em outras linguagens de programação, variáveis internas são definidas também em VHDL, com o detalhe de que estas podem somente ser utilizadas dentro de seus respectivos processos, procedimentos ou funções. Para troca de dados entre processos devem-se utilizar sinais ao invés de variáveis [17] [18]. Outro exemplo possível de um corpo de arquitetura comportamental é apresentado na figura 5.6 para a entidade *reg4*, registrador de 4 bits:

```
architecture comportamento of reg4 is
begin
carga: process (clock)
variable d0_temp, d1_temp, d2_temp, d3temp : bit;
        begin
                if clk = '1' then           -- Carrega as entradas no registrador
                        if en = '1' then    -- temporário
                                d0_temp := d0;
                                d1_temp := d1;
                                d2_temp := d2;
                                d3_temp := d3;
                        end if;
                end if;                     -- Descarrega o registrador na saída
                q0 <= d0_temp after 5ns;
                q1 <= d1_temp after 5ns;
                q2 <= d2_temp after 5ns;
                q3 <= d3_temp after 5ns;
        end process carga;
end comportamento;
```

Figura 5.6 - VHDL: arquitetura comportamental.

Neste corpo de arquitetura, a parte após a diretiva *begin* inclui a descrição de como o registrador se comporta. Inicia com a definição do nome do processo, chamado carga, e termina com a diretiva *end process*. Na primeira parte da descrição é testada a condição de que ambos os sinais *en* e *clk* sejam iguais a '1'. Se eles são, as sentenças entre as diretivas *then* e *end if* são executadas, atualizando as variáveis do processo com os valores dos sinais de entrada. Após a estrutura *if* os quatro sinais de saída são atualizados com um atraso de 5ns. O processo *carga* é sensível ao sinal *clock*, o que é indicado entre parênteses na declaração do mesmo. Quando uma mudança no sinal *clock* ocorre, o processo é novamente executado [17][18].

- **Modelo Estrutural**

Existem várias formas pela qual um modelo estrutural pode ser expresso [17][18]. Uma forma comum é o esquemático de circuito. Símbolos gráficos são usados para representar subsistemas que são conectados usando linhas representando fios. Esta forma gráfica é geralmente uma das preferidas pelos projetistas, entretanto a mesma forma estrutural pode ser representada textualmente na forma de uma lista de conexões. Uma descrição estrutural de um sistema é expressa, portanto em termos de subsistemas interconectados por sinais. O mapeamento de portas especifica que portas da entidade são conectadas para quais sinais do sistema que se está montando. Por exemplo, tomando-se como base a entidade *flip-flop* tipo D, temos internamente a representação da entidade *d_latch* cuja declaração encontra-se na figura 5.7:

```
entity d_latch is
port (  d, clk : in bit;
        q : out bit);
endentity d_latch;
```

Figura 5.7 - VHDL: declaração da entidade flip-flop.

Pode-se utilizar esta estrutura para se compor sistemas mais complexos, como registradores de um número qualquer de bits. Para ilustrar isto na figura 5.8 é apresentada uma descrição de um registrador de 2 bits, chamado *reg_2bits*, construído a partir do flip_flop *d_latch.*

```
entity reg_2bits is
port ( d0, d1, clk : in bit;            -- Declaração de conexões externas
       q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
begin:
bit0: entity work.d_latch               -- Chamada da estrutura de um lacth
      port map ( d0, clk, q0)           -- Interligações
bit1: entity work.d_latch               -- Chamada da estrutura de um lacth
      port map  (d1, clk, q1)           -- Interligações
end architecture estrutura;
```

Figura 5.8 - VHDL: descrição de um registrador de 2 bits.

Note que neste caso, na própria indicação dos componentes que constituem o sistema *reg_2bits* já é feito o mapeamento de pinos desejados, ou seja, *d0* é associado ao pino *d* do primeiro flip-flop, assim como o pino *clk* ao sinal de porta de mesmo nome e *q0* ao bit *q*, conforme a estruturação *bit0*. A estrutura *bit1* é descrita de forma análoga somente que uma nova entidade *d_latch* (uma cópia deste componente) será criada para a realização de suas ligações. Entretanto, apesar da forma apresentada ser funcional, muitas vezes na prática se costuma descrever formalmente o mapeamento de portas desejados. Isso torna o esquema de ligações implementado mais visual e intuitivo. O mapeamento então se caracteriza pela indicação dos sinais de origem/destino para cada conexão como figura 5.9 [17][18].

```
entity reg_2bits is
port (    d0, d1, clk : in bit;
q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
begin:
bit0: entity work.d_latch               -- Chamada da estrutura de um lacth
     port map (d => d0, clk => clk, q => q0)
bit1: entity work.d_latch               -- Chamada da estrutura de um lacth
    port map (d => d1, clk => clk, q => q1)
end architecture estrutura;
```

Figura 5.9 - VHDL:descrição de um registrador de 2 bits de forma mais clara.

Este formato traz por si só uma maior flexibilidade do projeto na manipulação de sinais. Pode-se assim trabalhar com arranjos mais flexíveis como quando se utilizam variáveis de tipo composto (*bit_vector, etc...*). Na descrição da figura 5.10, por exemplo:

```
entity registrador_4bits is
port (    d : in bit_vector (0 to 3);    -- Definição da entidade que será
          clk : in bit;                  -- utilizada posteriormente
          q : out bit_vector (0 to 3);   -- (este é o registrador de 4 bits)
end entity registrador_4bits;
...
entity registrador_8bits is
port (    din : in bit_vector (0 to 7);  -- Definição dos pinos de conexão
          clk : in bit;                  -- externa da implementação
          dout : out bit_vector (0 to 7)); -- do registrador de  bits
end entity registrador_8bits;            -- Inicio da implementação
architecture estrutura of registrador_8bits is
begin:                                   -- Inicio da implementação
estrutura: entity work.registrador_4bits  -- para um novo reg de 8 bits
port map ( d (0 to 3) => din (0 to 3), -- Instaciamento do reg de 4 bits
               clk => clk,               -- para os 4 bits menos significativos
               q (0 to 3) => dout (0 to 3))
port map ( d (0 to 3) => din (4 to 7), -- Instaciamento do reg de 4 bits
               clk => clk,               -- para os 4 bits mais significativos
               q (0 to 3) => dout (4 to 7))
end architecture estrutura;
```

Figura 5.10 - VHDL:descrição de um registrador de 8 bits.

Pode-se montar um registrador de 8 bits (*registrador_8bits*) a partir de registradores de 4 bits (*registrador_4bits*) operando com sinais tipo *bit_vector*, o que simplifica a indicação das ligações. Muitas vezes, como em alguns esquemáticos de circuitos feitos com componentes discretos, desejam-se conectar sinais a valores fixos ou até mesmo desligá-los do circuito final. Isto é também possível em VHDL pela adequada atribuição de valores no mapeamento de conexões [17] [18]. A figura 5.11 tem apresentação deste recurso.

```
entity FF_D is                          -- Definição de um flip-flop
port (   d, clk, rs, pr : in bit;       -- será usado no reg de 2 bits
         q, nq : out bit);
end entity FF_D;
...
entity reg_2bits is
port (    d0, d1, clk : in bit;         -- Declaração do reg de 2 bits
          q0, q1 : out bit);
end entity reg_2bits;
architecture estrutura of reg_2bits is
begin:
bit0: entity work.FF_D                  -- Uso do flip-flop para o 1.o
        port map ( d => d0, clk => clk,  -- bit do registrador
                   rs => '1', pr => '1',
                    q => q0, nq => open)
bit1: entity work.FF_D(behavioral)      -- Uso do flip-flop para o 2.o
        port map ( d => d1, clk => clk,  -- bit do registrador
                   rs => '1', pr => '1',
                   q => q1, nq => open)
end architecture estrutura;
```

Figura 5.11 - VHDL:descrição de um registrador de 2 bits instanciando sinais.

5.2.3. Interligação de Modelos com Sinais

Modelos não precisam ser puramente estruturais ou comportamentais. Frequentemente é útil especificar um modelo com algumas partes compostas de instâncias de componentes e outras descritas usando processos. Utilizam-se sinais no sentido de interligar instâncias de componentes e processos. Um sinal pode ser associado a porta de uma instância de componente e pode ser assinalado para ser lido ou escrito em um processo [17][18].

Pode-se escrever um projeto, com um modelo híbrido que inclua ambas as abordagens. Estas instruções são coletivamente chamadas de sentenças concorrentes, uma vez que todos seus processos são executados concorrentemente, quando o modelo é simulado.

Uma demonstração disto pode ser observada na figura 5.12. Este modelo descreve um multiplicador que consiste de uma parte de tratamento de dados e uma seção de controle.

A parte de tratamento de dados é descrita estruturalmente, usando um número de instâncias de componentes. A seção de controle pode ser descrita comportamentalmente usando um processo que liga os sinais de controle com a parte de tratamento de dados.

```
entity multiplicador is
port ( clk, reset, multiplicando, multiplicador: in integer;
        produto: out integer);
end entity multiplicador;
architecture prj_completo of multiplicador is
signal produto_parcial, produto_final: integer;
signal controle_aritmet, result_en, mult_bit, mult_load: bit;
begin
        unid_aritmet: entity work.shift_adder(behavior)
        port map (somador => multiplicador, augend => produto_final,
                sum => produto_parcial,
                add_control => controle_ aritmet);
        resultado: entity work.reg(behavior)
        port map ( d => produto_parcial, q => produto_final,
                en => result_en, reset => reset);
        multiplicador_rs: entity work.shift_reg(behavior)
        port map ( d => multiplicador, q => mult_bit,
                load => mult_load, clk => clk);
                produto <= produto_final;
        secao_controle : process
                ...
                wait on clk, reset;
        end process secao_controle;
end architecture prj_completo;
```

Figura 5.12 - VHDL:descrição estrutural e comportamental [17][18].

Capítulo 6

Caracterização, Modelagem e Resultados.

Conforme apresentado no capítulo 3, o algoritmo de criptografia AES pode ser implementado tanto em hardware como em software. O grande trunfo da versatilidade de implementação deste algoritmo esta no fato de se poder subdividi-lo em partes menores e trabalhar com uma implementação modular. Assim podemos trabalhar com a otimização de cada parte, agora denominada de bloco funcional, e realizar testes de desempenho e funcionamento individualizados.

Sem dúvida, para a implementação deste algoritmo em hardware, o recurso de desmembramento do projeto em partes menores é muito importante para uma otimização ainda maior do funcionamento do mesmo, onde se pode avaliar não só o resultado funcional de cada bloco, mais também se pode melhorar diversas outras características de operação do projeto. Como exemplo, podemos citar como otimizações importantes, o consumo de área ocupada pela função, a análise de temporização do dispositivo e as características com relação a implementação de determinadas operações, utilizando-se o máximo de recurso do componente de FPGA, além de diversos outros [12][13][14].

Outra grande vantagem deste desenvolvimento fracionado juntamente com uso de linguagem de descrição de hardware (VHDL), é a capacidade que o projetista tem de testar se a implementação esta produzindo realmente o elemento lógico desejado, através de síntese

RTL. Com este recurso é possível implementar funções lógicas que utilizam recursos específicos do dispositivo de lógica programável (FPGA), como por exemplo, áreas reservadas para implementação de memórias devem ser utilizadas para se implementar memórias e áreas reservadas para implementação de lógica devem-se implementar funções lógicas, entre outras. Assim, podemos tirar proveito máximo dos FPGAs e direcionar o desenvolvimento do projeto para o melhor resultado de desempenho e consumo de recursos.

No próximo item, apresentaremos a descrição e caracterização funcional do algoritmo AES, além de detalhes e modelagem de alguns de seus blocos funcionais que foram desenvolvidos em VHDL e implementados em um dispositivo FPGA da ALTERA ®.

6.1. Caracterização Funcional

A fim de se realizar uma caracterização funcional do algoritmo AES, realizamos como visto na figura 6.1, uma caracterização do algoritmo em forma de fluxo de funcionamento, onde definiu-se a forma e a ordem que as funções de operação devem ocorrer para o funcionamento correto do projeto. Posteriormente será apresentada uma discussão detalhada de cada unidade funcional do algoritmo.

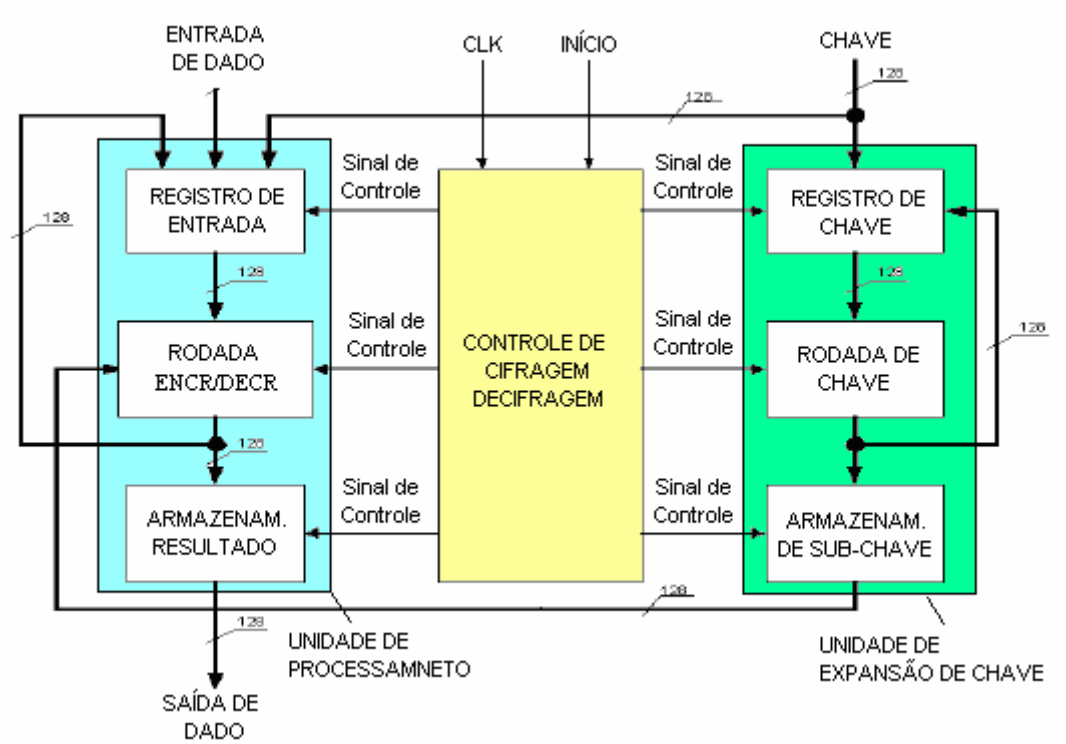


Figura 6.1 - Diagrama geral de funcionamento

Assim os blocos básicos de trabalho do algoritmo estão definidos, sendo estes, a unidade de expansão de chaves, e unidade de processamento propriamente dito que realiza a função de criptografia . Tanto a unidade de processamento quanto a unidade de expansão de chaves, podem por si só serem subdivididos em subfunções menores. Isto nos auxiliará no desenvolvimento do projeto, ajuste de desempenho, além de facilitar a compreensão de funcionamento do bloco funcional.

Antes de iniciarmos um detalhamento maior de funcionamento das funções vale aqui apresentar os parâmetros utilizados neste trabalho. Será utilizado um bloco de cifragem de 128 bits (visando desempenho não usaremos outros tamanhos de chaves), conforme definidos pela norma FIPS-197 [08], e também uma chave de processamento de 128 bits, devido a limitações de recursos (pinos, elementos lógicos e memória) do dispositivo de lógica programável (FPGA), que dispunha-se no início deste trabalho. Assim, conforme determinado pela norma [08], o numero de rodadas de expansão de chaves e de execução do algoritmo será uma inicial mais dez subseqüentes.

Observa-se na figura 6.2, com mais detalhes, como funciona o bloco funcional de expansão de chaves. Com isto pode-se realizar um estudo mais detalhado do seu funcionamento e desenvolver um modo de implementá-lo em *VHDL* com rapidez, segurança e garantia de operação correta, conforme exigido e definido pelo algoritmo original [08].

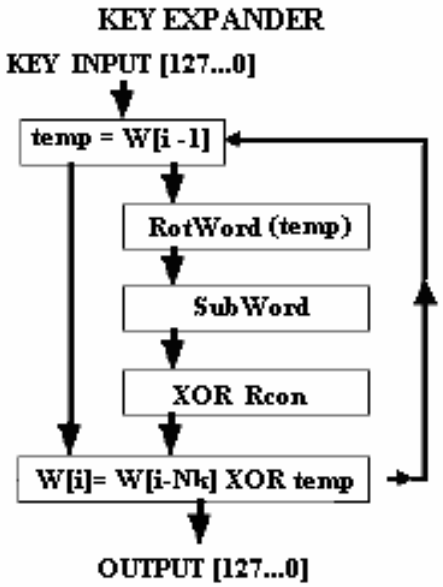


Figura 6.2 - Diagrama detalhado da função de expansão de chaves [08] [09].

Na figura 6.2, pode-se verificar que a função de expansão de chaves trabalha da seguinte forma: inicialmente a chave original é dada como entrada da função e armazenada em um registrador ou memória. Após isto, a matriz de estado da chave (w[i]) é instanciada em uma variável *temp* que recebe coluna por coluna a matriz de chaves e assim processa a expansão.

Esta expansão é realizada de forma que, caso o índice da coluna que esteja sendo processada no momento, seja múltiplo de quatro (no caso deste trabalho, uma chave de 128 bits), conforme definido pelo algoritmo original no capítulo 3 [08], esta palavra de 32 bits sofrerá a ação das funções RotWord, SubWord, XOR com Rcon e no final uma operação XOR com uma outra palavra de 32 bits cujo índice será quatro posições anteriores a atual. Caso não seja múltiplo, então sofrerão apenas a ação da operação XOR com outra palavra de 32 bits quatro posições anteriores.

Desta forma processamos a expansão de chaves onde, no caso deste nosso trabalho, produzirá um vetor com quarenta e quatro palavras de 32 bits que quando agrupadas quatro a quatro formarão a chave de processamento de cada um das onze rodadas, sendo uma inicial com a chave original e mais dez com as chaves expandidas pela função [08].

Agora na figura 6.3, apresenta-se a função de processamento principal do algoritmo que também pode ser desmembrada em um conjunto de subfunções que podem auxiliar também muito no seu desenvolvimento. Como descrito anteriormente, onde o bloco de processamento principal faz uso que algumas funções do bloco funcional de expansão de chaves, como por exemplo, a função de substituição de bytes (sub_byte), e também com alguns ajustes, pode-se utilizar a função de rotação de palavras (rot_word) apresentada na expansão de chaves para se rotacionar a matriz de estados da unidade de processamento principal , implementando assim a função *ShifRow* [08].

O funcionamento do processo de cifragem e decifragem funciona de forma parecida, onde inicialmente tem-se a entrada do dado em um registrador ou modulo de memória, que posteriormente será submetido a diversas ações pelo algoritmo, com o interesse que se realizar a encriptação ou a decriptação da informação conforme necessidade do processo.

Na cifragem, a informação (bloco de 128 bits), passa por uma operação de substituição byte a byte da sua matriz de estados (ByteSub), e posteriormente sofre um rotação pré-definida nas linha da matriz de estado conforme o algoritmo de criptografia (ShiftRow). Após, esta matriz é alterada por uma operação de redução polinomial (MixColumns) já definida anteriormente no capitulo 3 [08], e por fim se realiza uma operação XOR com a chave de rodada (AddRoundKey) originada da função de expansão de chaves. Vale resaltar que na última interação (rodada), a operação de redução polinomial não é executada.

Já no processo de decifragem, a informação (bloco de 128 bits) simplesmente realiza a operação inversa do processo de cifragem. Ou seja, inicialmente se adiciona (XOR) a matriz de estado com a chave de rodada, depois realizamos a redução polinomial, que não deve executada na primeira rodada, e depois as funções inversas respectivas do processo de cifragem, sendo estas, realizadas na ordem inversa, como na figura 6.3. Com isto, realiza-se tanto o processo de cifragem como decifragem dentro da unidade de processamento, lembrando que antes de qualquer operação devemos definir se vamos realizar operações de cifragem ou decifragem de dados, para posteriormente executarmos as operações propriamente ditas [08].

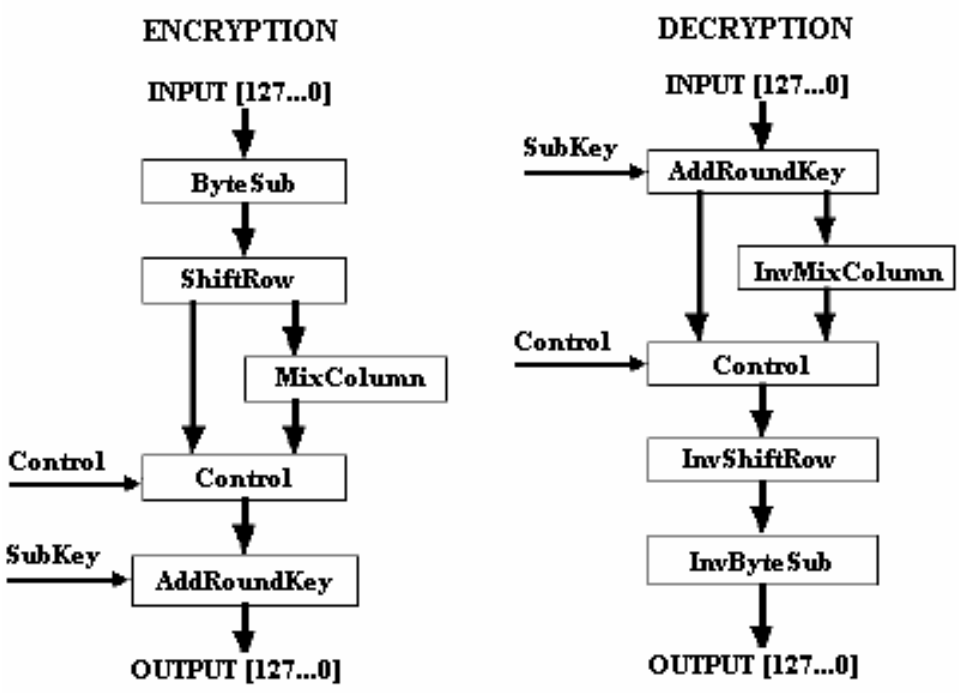


Figura 6.3 - Diagrama detalhado da função de processamento principal [08] [09].

6.2. Parâmetros do Trabalho

Antes de apresentarmos a modelagem e os resultados é importante mostrar o ambiente em que foi desenvolvido este trabalho, juntamente com os parâmetros de implementação. Entre as opções de projeto, linguagem de implementação e componentes, já se pode afirmar que os resultados variam de uma opção para outra. Uma boa forma de resolver esta questão é a uniformalização de alguns parâmetros, entre eles principalmente, a família de FPGAs utilizadas e o componente específico que se irá trabalhar, pois assim podemos garantir que as medidas de desempenho serão realizadas sobre a mesma plataforma.

Definimos, conforme disponibilidade de recursos, qual família e dispositivo deverá ser desenvolvido o trabalho. Sendo assim, estavam disponíveis somente os componentes do fabricante ALTERA ®, da família ACEX1K, o mesmo foi utilizado nos processos de desenvolvimento e implementação do trabalho. Na tabela 6.1 apresentamos um resumo descritivo de alguns parâmetros e recursos disponíveis no componente (FPGA) utilizado neste trabalho.

Tabela 6.1 - Recursos disponíveis da família de FPGAs ACEX1K [19].

ACEX™ 1K Device Features	
Feature	**EP1K50**
Typical gates	50,000
Maximum system gates	199,000
Logic elements (LEs)	2,880
EABs	10
Total RAM bits	40,960
Maximum user I/O pins	249

E quais os parâmetros de desempenho deverão ser monitorados para comparar e avaliar a implementação? Sem dúvida, para comprovar a eficiência da implementação o principal item que se deve avaliar é o tempo de propagação do sinal elétrico dentro do FPGA a partir da lógica utilizada, ou seja, o tempo crítico que se necessita para se obter o valor estável de dado na saída a partir da entrada. Assim, tem-se que medir o tempo de propagação em cada função e depois tentar realizar uma extrapolação matemática teórica para um sistema completo.

Além do fator tempo, também é importante medir e apresentar a quantidade de recursos consumidos no FPGA (elementos lógicos, área e memória consumida, além do custo de cada componente), pois assim se pode realizar uma avaliação de custo - beneficio do trabalho, projetando futuramente estes parâmetros em outras famílias de FPGAs deste e de outros fabricantes.

6.3. Modelagem, Funcionamento e Resultados.

De acordo com a importância funcional e a possibilidade de incremento de desempenho em uma futura implementação completa do algoritmo, optou-se por algumas funções para implementarmos em linguagem VHDL e testar o funcionamento e operação em um dispositivo de lógica programável FPGA. Com esta opção, realizou-se a modelagem das funções seguindo não só a melhor forma de descrição VHDL, mas adequando sempre os códigos as características do componente escolhido, no caso ALTERA ® família ACEX1K.

6.3.1. Função ByteSub

A primeira função escolhida a ser implementada, foi a de substituição de bytes (ByteSub), que realiza a troca do byte de entrada por um outro predeterminado como descrito no capítulo 3 [08], que pode ser implementado com o uso de uma tabela fixa, sendo assim modelado como uma memória de valores fixos (ROM). Para a implementação desta função, utilizou-se uma descrição VHDL [18] que cria uma memória do tipo *ROM* com capacidade de 256 endereços de 8 bits, consumindo assim 2048 bits de memória no componente. Como esta função é utilizada na função de expansão da chave (normal ou inversa) e também tanto na cifragem como na decifragem de dados, o seu estado inverso, denominado de função inversa de substituição de bytes (InvByteSub), também foi implementado da mesma forma. Assim o consumo total de células de memória é de 4048 bits por chamada da função.

Para esclarecer melhor o funcionamento da função pode-se ver na figura 6.4 o código VHDL desta função onde detalhamos apenas as partes referentes a definição de arquitetura e o seu funcionamento:

```
30 ARCHITECTURE rtl OF byte_sub_ROM IS
31     TYPE ram_t IS ARRAY (0 TO 255) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
32     SIGNAL memory: ram_t := (X"63",X"7C",X"77",X"7B",X"F2",X"6B",X"6F",X"C5",
33                              X"30",X"01",X"67",X"2B",X"FE",X"D7",X"AB",X"76",
34                              X"CA",X"82",X"C9",X"7D",X"FA",X"59",X"47",X"F0",
35                              X"AD",X"D4",X"A2",X"AF",X"9C",X"A4",X"72",X"C0",
36                              X"B7",X"FD",X"93",X"26",X"36",X"3F",X"F7",X"CC",
37                              X"34",X"A5",X"E5",X"F1",X"71",X"D8",X"31",X"15",
38                              X"04",X"C7",X"23",X"C3",X"18",X"96",X"05",X"9A",
39                              X"07",X"12",X"80",X"E2",X"EB",X"27",X"B2",X"75",
40                              X"09",X"83",X"2C",X"1A",X"1B",X"6E",X"5A",X"A0",
41                              X"52",X"3B",X"D6",X"B3",X"29",X"E3",X"2F",X"84",
42                              X"53",X"D1",X"00",X"ED",X"20",X"FC",X"B1",X"5B",
43                              X"6A",X"CB",X"BE",X"39",X"4A",X"4C",X"58",X"CF",
44                              X"D0",X"EF",X"AA",X"FB",X"43",X"4D",X"33",X"85",
45                              X"45",X"F9",X"02",X"7F",X"50",X"3C",X"9F",X"A8",
46                              X"51",X"A3",X"40",X"8F",X"92",X"9D",X"38",X"F5",
47                              X"BC",X"B6",X"DA",X"21",X"10",X"FF",X"F3",X"D2",
48                              X"CD",X"0C",X"13",X"EC",X"5F",X"97",X"44",X"17",
49                              X"C4",X"A7",X"7E",X"3D",X"64",X"5D",X"19",X"73",
50                              X"60",X"81",X"4F",X"DC",X"22",X"2A",X"90",X"88",
51                              X"46",X"EE",X"B8",X"14",X"DE",X"5E",X"0B",X"DB",
52                              X"E0",X"32",X"3A",X"0A",X"49",X"06",X"24",X"5C",
53                              X"C2",X"D3",X"AC",X"62",X"91",X"95",X"E4",X"79",
54                              X"E7",X"C8",X"37",X"6D",X"8D",X"D5",X"4E",X"A9",
55                              X"6C",X"56",X"F4",X"EA",X"65",X"7A",X"AE",X"08",
56                              X"BA",X"78",X"25",X"2E",X"1C",X"A6",X"B4",X"C6",
57                              X"E8",X"DD",X"74",X"1F",X"4B",X"BD",X"8B",X"8A",
58                              X"70",X"3E",X"B5",X"66",X"48",X"03",X"F6",X"0E",
59                              X"61",X"35",X"57",X"B9",X"86",X"C1",X"1D",X"9E",
60                              X"E1",X"F8",X"98",X"11",X"69",X"D9",X"8E",X"94",
61                              X"9B",X"1E",X"87",X"E9",X"CE",X"55",X"28",X"DF",
62                              X"8C",X"A1",X"89",X"0D",X"BF",X"E6",X"42",X"68",
63                              X"41",X"99",X"2D",X"0F",X"B0",X"54",X"BB",X"16");
64     SIGNAL iaddr: integer range 0 to 255;
65
66 BEGIN
67
68     iaddr <= suv2int(a);
69
70     PROCESS (clk,iaddr,memory)
71     BEGIN
72         IF clk'EVENT AND clk = '1' THEN
73             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
74                 do <= memory(iaddr);
75             ELSE
76                 memory(iaddr) <= di;    --  processo de escrita na RAM
77             END IF;
78         END IF;
79     END PROCESS;
80 END rtl;
```

Figura 6.4 - Arquitetura da função *ByteSub.*

Como apresentado na figura 6.4, pode-se ver que na linha 30, é utilizada uma arquitetura denominada *'rtl'* a partir de um uma entidade denominada *'byte_sub_ROM'*, e a seguir entre as linhas 31 e 64, declara-se os sinais de controle e constantes que serão necessárias a implementação do algoritmo. Particularmente na linha 31 é definido um tipo

(constante), que será utilizado depois para instanciar as células de memórias de 256 posições por 1 byte de largura, agora entre as linhas 32 a 63 inicializamos a memória propriamente dita e já damos carga na mesma, fazendo-a trabalhar como uma *RAM* pré-carregada, e na linha 64 definimos o vetor que será usado no endereçamento da memória. Após isto, tem-se na linha 66 a inicialização de funcionamento da arquitetura, que neste caso implementa uma estrutura comportamental como já explicado no capítulo 5 [17][18], assim na linha 68, converte-se o parâmetro que é passado para a função (endereçamento) do formato binário para o decimal, sendo este mais adequado para se referenciar a posição de memória.

Já entre as linhas 70 e 79, tem-se a definição do processo principal de funcionamento da memória. Pode-se ver que, quando se verifica uma alteração ou evento em *'clk'* que representa a porta de entrada de relógio, inicializa-se, dependendo do estado de *'wr_rd_en'*, a leitura do conteúdo do endereço de memória pra a porta de saída *'do'*, ou a escrita do conteúdo da porta de entrada *'di'* para o endereço de memória. Assim, na linha 80, finaliza-se a descrição desta função, lembrando que, como a intenção é utilizar esta função como uma memória apenas de leitura (ROM), então o valor da porta de entrada *'wr_rd_en'* deverá estar conectado sempre em *'GND'*, permitindo assim trabalhar apenas como acesso de leitura na mesa, desprezando-se qualquer conteúdo em *'di'*.

Com o interesse de se esclarecer ainda mais o funcionamento da função, pode-se observar na figura 6.5, como deveria ser representada de forma gráfica a mesma.

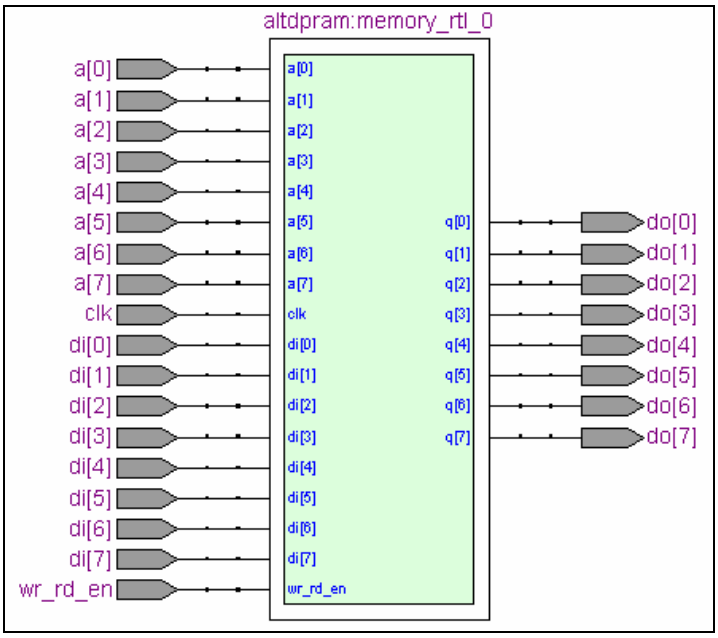


Figura 6.5 - Função *ByteSub*, representação gráfica.

Assim como visto na figura 6.5, verifica-se a existência de todos as portas e entrada/saída, necessárias para o funcionamento da função, como os vetores a[i], di[i], do[i] e os sinais *clk* e *wr_rd_en*, que no caso de uma implementação utilizando-se de interface gráfica pode ser usada como modelo.

Conforme descrição do algoritmo de criptografia [08], tem-se na figura 6.6 a síntese RTL desta função realizada pelo software QUARTUS II ® da ALTERA ®. Como poderá ser verificado, está síntese mostra a implementação de uma estrutura de memória síncrona com um vetor de 8 bits de endereçamento e outros dois de 8 bits para a entrada de dados e leitura dos mesmos (aplicação da função). A descrição será implementada apenas com a função de leitura, obrigando o funcionamento da função como uma *ROM*.

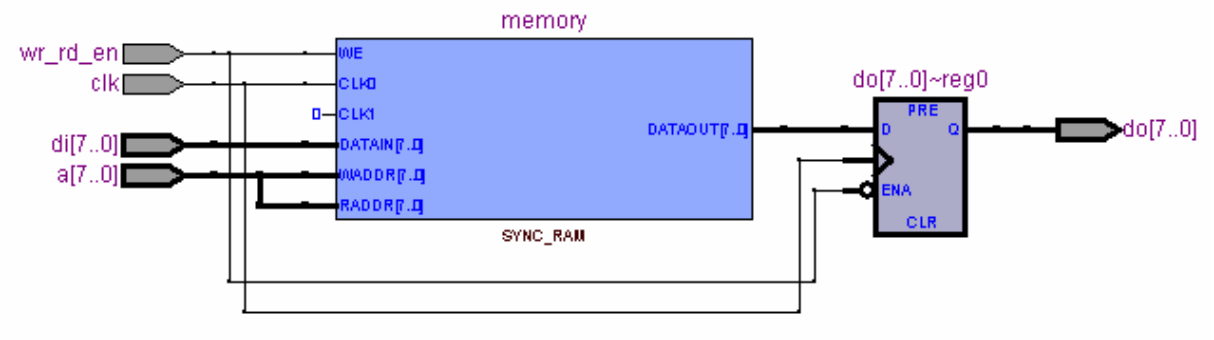


Figura 6.6 - Descrição RTL da função *ByteSub* e *InvByteSub*.

Agora são apresentados e discutidos os resultados obtidos por simulação da função no software de desenvolvimento. Na figura 6.7, observa-se o gráfico de forma de onda que representa os resultados de simulação obtidos pelo software QUARTUS II ®.

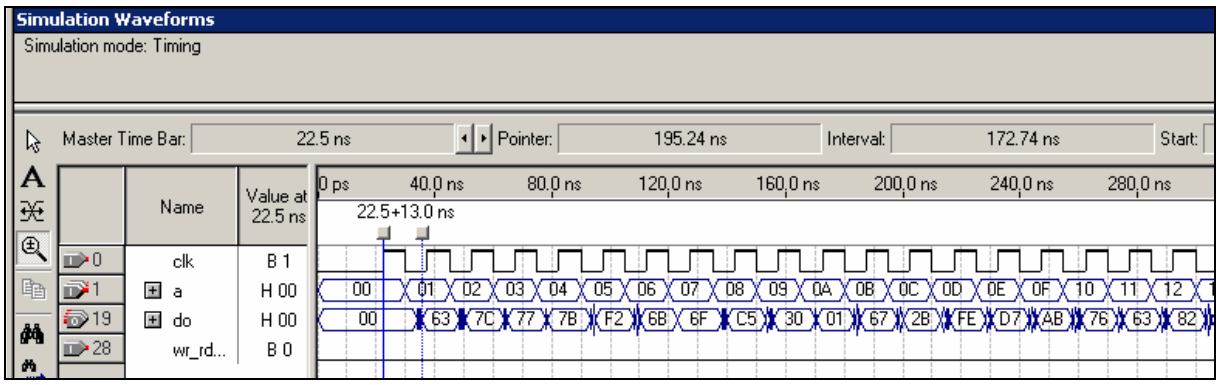


Figura 6.7 - Função *ByteSub*, gráfico de forma de onda.

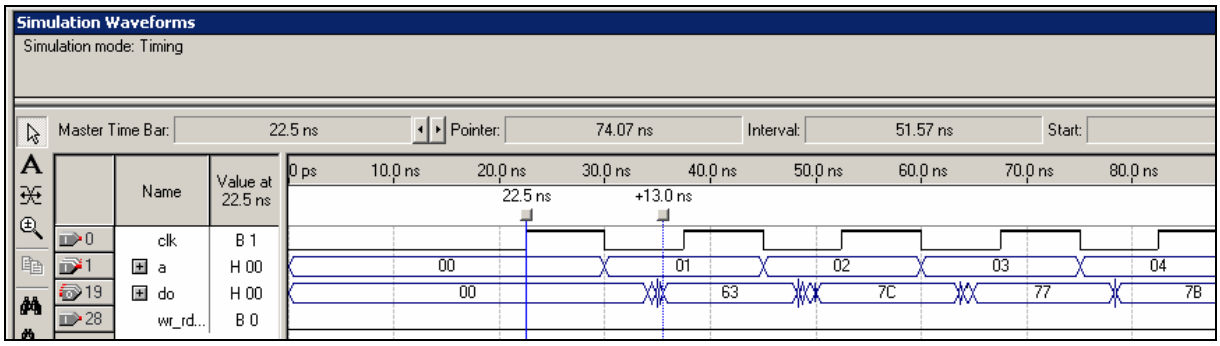


Figura 6.8 - Função *ByteSub*, gráfico de forma de onda (detalhes).

Já na figura 6.8, pode-se ver em detalhes o resultado da função, onde se aplicou um sinal de relógio com ciclos de 15ns, iniciando-se no instante 15ns, e também com o vetor de endereçamento *'a'*, sendo incrementado de uma unidade a partir de 15ns com um ciclo de também 15ns. Conforme visto o resultado desta simulação, com a porta *'wr_rd_en'* em *'0'*, apresenta corretamente o esperado pela função, uma vez que os valores colhidos na porta de saída *'do'* correspondem exatamente ao esperado pela descrição do algoritmo de criptografia [08]. Pode-se ver também uma marca aos 22,5ns e outra 13ns após, demonstrando o tempo de atraso máximo da função.

O resultado completo pode ser visto na figura 6.9 abaixo, onde é demonstrado os parâmetros de desempenho obtidos por esta implementação.

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Total logic elements	0
2	Total combinational functions	0
3	-- Total 4-input functions	0
4	-- Total 3-input functions	0
5	-- Total 2-input functions	0
6	-- Total 1-input functions	0
7	-- Total 0-input functions	0
8	Total registers	0
9	I/O pins	26
10	Total memory bits	2048

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time
1	Worst-case tsu	N/A	None	11.500 ns
2	Worst-case tco	N/A	None	13.000 ns
3	Worst-case th	N/A	None	0.300 ns
4	Clock Setup: 'clk'	N/A	None	149.25 MHz (period = 6.700 ns)
5	Total number of failed paths			

Figura 6.9 - Função ByteSub, resultados da implementação

Como apresentado acima, pode ser observdo que a avaliação de recursos consumidos pela implementação no dispositivo de lógica programável (FPGA), apresentou apenas o consumo de 2048 bits de espaço de memória e o uso de 26 pinos de entrada e saída, não utilizando nenhum elemento lógico, funções combinacionais ou mesmo registradores.

Já no desempenho de analise de tempo de propagação, pode-se verificar na mesma figura 6.9 que o sumário de análise apresenta os valores de temporização de pior caso variando entre 0.3ns a 13ns. Vimos que: o tempo mínimo que o endereço (porta *'a'*) da memória deve estar disponível na entrada (t_{SU}) deve ser de 11,5ns; o tempo máximo que a função demora para retornar o dado válido (t_{CO}) na saída é de 13ns; o tempo mínimo que o dado (porta *'a'*) deve estar disponível na entrada depois de um evento de relógio (t_H) para que a função consiga processar a informação é de 0,3ns; a freqüência de relógio máxima que pode ser aplicada sem violar a organização interna (t_{SU}) e manter exigências de tempo (t_H) para esta implementação é 149,25Mhz.

O mesmo processo foi aplicado para a função *InvByteSub*, obtendo os mesmos resultados, lembrando que o conteúdo de carga da memória ROM é diferente e implementa o que esta definido na descrição do algoritmo no capítulo 3 [08], conforme a figura 3.13.

O código VHDL completo desta função, além dos pacotes de funções e bibliotecas complementares, necessários para o funcionamento da mesma, está disponível para análise no apêndice 'A' deste trabalho.

6.3.2. Função Rcon

Como a função anterior (ByteSub) modelava uma memória, aproveitamos os mesmos arranjos para modelarmos a função de constante de rodada (Rcon), necessária para o uso no bloco funcional de expansão de chaves. Esta função implementa uma constante que deve ser usada numa operação OU exclusivo (XOR) durante a geração das chaves de rodada. Como cada rodada utiliza uma constante diferente para a operação lógica, o meio mais simples de modelagem desta função é o instanciamento de uma unidade de memória capaz de armazenar as constantes para cada rodada e identificá-las (endereçá-las) para o correto uso em sua rodada predeterminada.

Segunda a especificação feita pelo algoritmo de criptografia, estas constantes *Rcon* devem ser implementadas de acordo com o numero de rodadas exigidas pelo bloco de expansão de chaves. Pode-se ter 10, 12 ou 14 constantes de acordo com o número de rodadas necessárias para a expansão da chave, uma vez que as chaves podem ter o tamanho de 128,

192 e 256 bits. Lembrando que, como já informado anteriormente, na nossa implementação optamos pela chave com tamanho fixo de 128 bits, como meio de simplificar o processo, sendo assim necessária apenas a expansão e armazenamento de 10 constantes fixas de rodada.

Como na implementação de *ByteSub*, a função *Rcon* também necessita do instanciamento de uma memória para implementa-la de uma forma mais coerente e rápida, além de também utilizar as células de memórias já disponíveis, para este tipo de implementação no FPGA. Assim, na modelagem desta função foi utilizado um elemento de memória com a capacidade de 10 endereços de 8 bits, consumindo assim 80 bits de memória. Lembrando que como é necessária a mesma quantidade de memória para as constantes de rodada do processo inverso de expansão de chaves, tem-se que o consumo total de memória destas funções será de 160 bits.

Na figura 6.10 temos o código VHDL desta função, aonde observa-se apenas as partes referentes a definição de arquitetura e o seu funcionamento.

```
31 ARCHITECTURE rtl OF r_con_ROM IS
32     TYPE ram_t IS ARRAY (0 TO 9) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
33     SIGNAL memory: ram_t := (X"01",X"02",X"04",X"08",X"10",X"20",X"40",X"80",
34                              X"1B",X"36");
35     SIGNAL iaddr: integer range 0 to 9;
36
37 BEGIN
38 iaddr <= suv2int(a);
39     PROCESS (clk)
40     BEGIN
41         IF clk'EVENT AND clk = '1' THEN
42             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
43                 do <= memory(iaddr);
44             ELSE
45                 memory(iaddr) <= di;    --  processo de escrita na RAM
46             END IF;
47         END IF;
48     END PROCESS;
49 END rtl;
```

Figura 6.10 - Arquitetura da função *Rcon.*

Na figura 6.10, observa-se que na linha 31 é instancianda uma arquitetura denominada *'rtl'* a partir de um uma entidade denominada *'r_con_ROM'*, e a seguir entre as linhas 32 e 35, declarando as constantes e os sinais de controle utilizados na implementação. Na linha 32, definiu-se um tipo, que será utilizado depois para instanciar as células de memórias de 10 posições por 1 byte de largura, já nas linhas 33 e 34 foi inicializado a memória e já damos carga na mesma, fazendo-a trabalhar como uma *RAM* pré-carregada, e na

linha 35 tem-se o vetor que será usado no endereçamento. Após isto tem-se na linha 37 a inicialização da arquitetura, que neste caso implementa uma estrutura comportamental como visto na função *ByteSub*, com isto, na linha 38 foi convertido o parâmetro *'a'* que é passado para a função (endereçamento) do formato binário para o decimal, sendo assim mais adequado para se referenciar as posições da memória.

Entre as linhas 39 a 48, tem-se a definição do processo principal da memória. Vê-se que, quando ocorre uma subida de nível lógico em *'clk'*, que é a porta de entrada de relógio, inicializa-se a leitura do conteúdo do endereço de memória pra a porta de saída *'do'*, dependendo do estado de *'wr_rd_en'*, ou a escrita do conteúdo da porta de entrada *'di'* para o endereço de memória. Na linha 49 finalizamos a descrição da função. Vale lembrar que, como a intenção é utilizar a função como uma memória apenas de leitura (ROM), então a porta de entrada *'wr_rd_en'* será conectada sempre em '*GND'*, trabalhando assim apenas como acesso de leitura na mesa, como na implementação anterior *ByteSub*.

Na figura 6.11, com a intenção de se esclarecer ainda mais o funcionamento e o instanciamento, observa-se como deveria ser representada da função lógica de forma gráfica.

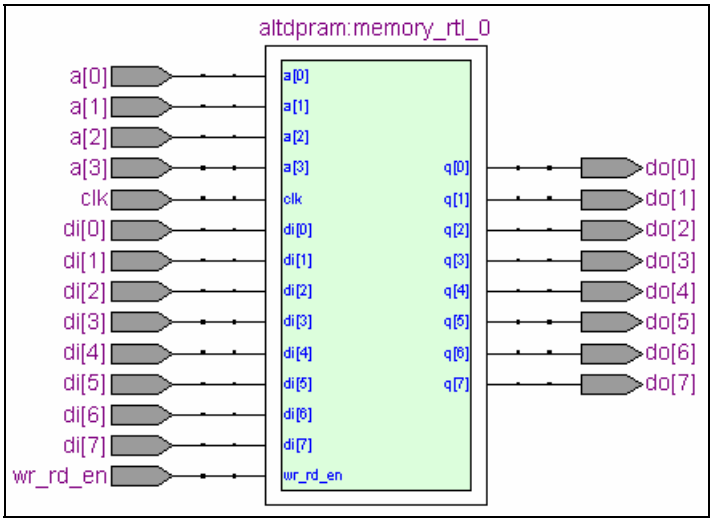


Figura 6.11 - Função *Rcon*, representação gráfica.

Pode-se verificar na figura 6.11, a existência dos sinais *clk* e *wr_rd_en* e todas as portas e entrada/saída, necessárias para o funcionamento da função, como os vetores a[i], di[i], do[i], que em uma implementação usando interface gráfica pode ser usada como modelo da mesma forma como apresentado na função anterior *ByteSub*, guardando as respectivas diferenças com relação ao sinal de endereçamento, que nesta caso.agora, endereça apenas 10

posições de memória.

É apresentado na figura 6.12, a síntese RTL desta função, realizada pelo software QUARTUS II ® da ALTERA ®. Pode-se ver que, de acordo com o código VHDL, realmente implementa-se a estrutura de uma memória síncrona com um vetor de quatro bits de endereçamento (programando apenas dez posições), e outros dois outros de oito bits, sendo um para a entrada e outro para leitura de dados. Lembrando que a carga da função será desabilitada para o correto funcionamento do sistema como uma *ROM*.

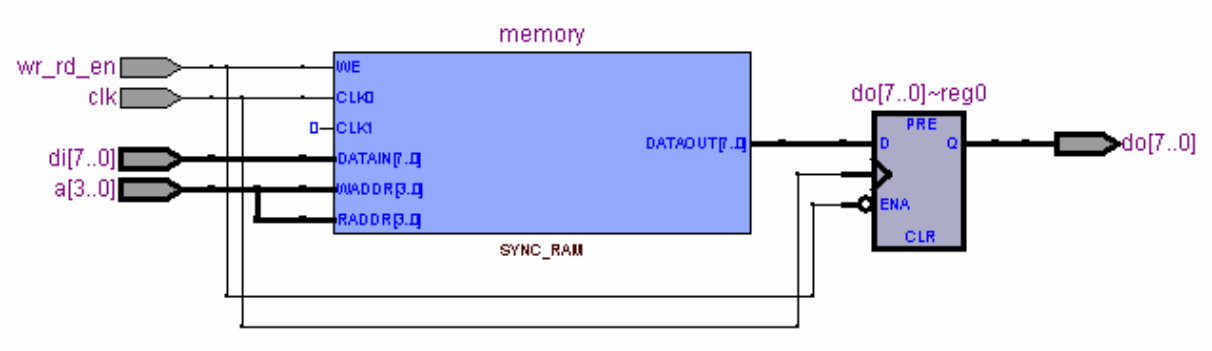


Figura 6.12 - Descrição RTL da função *Rcon.*

Apresenta-se agora os resultados obtidos por simulação da função no software de desenvolvimento. A figura 6.13, ilustra o gráfico de forma de onda que apresenta os resultados de simulação obtidos pelo software QUARTUS II ®.

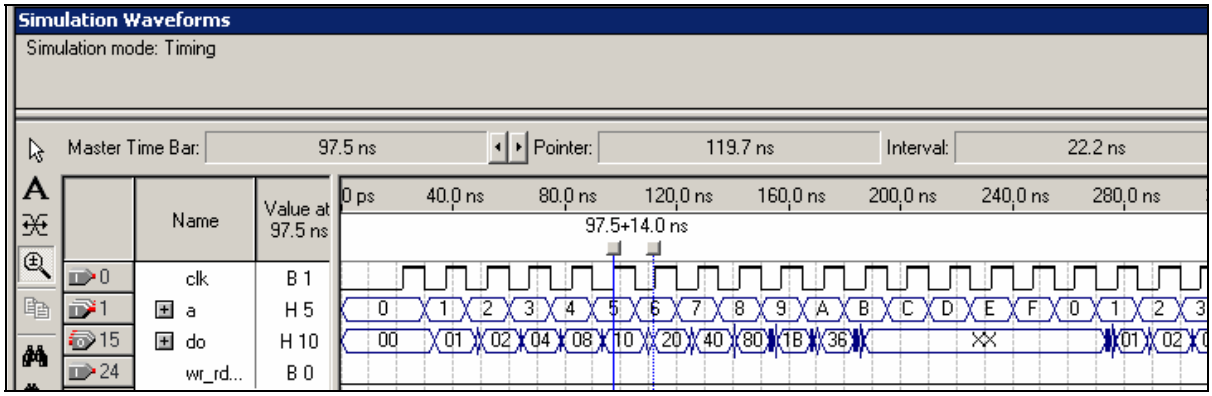


Figura 6.13 - Função *Rcon*, gráfico de forma de onda.

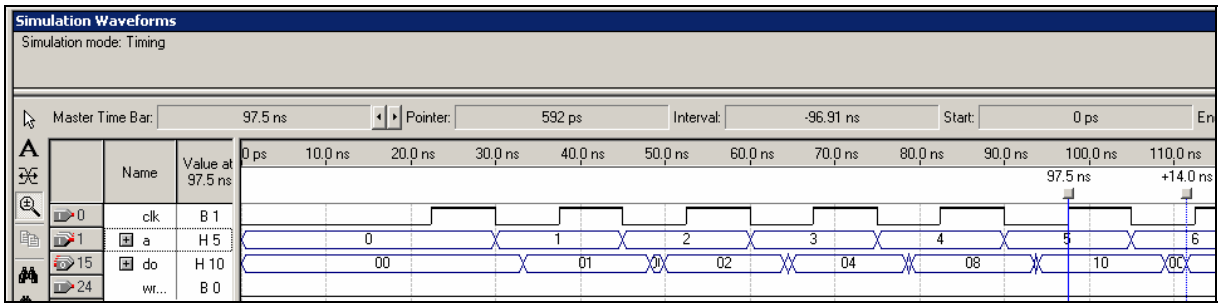


Figura 6.14 - Função *Rcon*, gráfico de forma de onda (detalhes).

Na figura 6.14, apresenta-se o resultado da simulação da função, aplicou-se um estímulo de relógio com ciclos de 15ns como na simulação anterior *ByteSub*, iniciando no tempo de 15ns, com o vetor de endereço *'a'*, incrementado de um a partir de 15ns a cada ciclo de 15ns. Visto isto, o resultado desta simulação apresenta o esperado pela função, com a porta *'wr_rd_en'* em *'0'*, uma vez que os valores na porta *'do'* correspondem de forma correta ao esperado conforme a descrição do algoritmo de criptografia [08]. Verifica-se também uma marca aos 97,5ns e outra 14ns após, mostrando o tempo de atraso máximo da função. Abaixo na figura 6.15, apresentamos os parâmetros de desempenho obtidos por esta implementação no software de desenvolvimento QUARTUS II ® da ALTERA ®.

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Total logic elements	0
2	Total combinational functions	0
3	-- Total 4-input functions	0
4	-- Total 3-input functions	0
5	-- Total 2-input functions	0
6	-- Total 1-input functions	0
7	-- Total 0-input functions	0
8	Total registers	0
9	I/O pins	22
10	Total memory bits	80

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time
1	Worst-case tsu	N/A	None	8.100 ns
2	Worst-case tco	N/A	None	14.000 ns
3	Worst-case th	N/A	None	0.300 ns
4	Clock Setup: 'clk'	N/A	None	149.25 MHz (period = 6.700 ns)
5	Total number of failed paths			

Figura 6.15 - Função *Rcon*, resultados da implementação

Acima na figura 6.15, observa-se que os recursos consumidos pela implementação apresentaram apenas o consumo de 80 bits de espaço de memória e o uso de 22 pinos de entrada e saída no dispositivo de lógica programável (FPGA), não utilizando nenhum elemento lógico, funções combinacionais ou mesmo registradores.

O desempenho de analise de tempo de propagação, também pode ser verificado na figura 6.15 onde tem-se o sumário de análise dos valores de temporização de pior caso,

variando entre 0.3ns a 14ns. Tem-se que: o tempo mínimo que o endereço da memória (porta *'a'*) deve estar disponível na entrada (t_{SU}) deve ser de 8,1ns; o tempo máximo que a função demora para retornar o dado válido (t_{CO}) na saída é de 14ns; o tempo mínimo que o dado (porta *'a'*) deve estar disponível na entrada depois de um evento de relógio (t_H) para que a função consiga processar a informação é de 0,3ns; a freqüência máxima a ser aplicada sem violar a organização (t_{SU}) e manter exigências (t_H) para esta implementação é 149,25Mhz.

O mesmo processo foi aplicado para a função *InvRcon*, obtendo os mesmos resultados. O conteúdo de carga da memória ROM é diferente da *Rcon* para a *InvRcon* e implementa o que esta definido na descrição do algoritmo no capítulo 3 [08]. O código VHDL completo desta função, além dos pacotes de funções e bibliotecas complementares, necessários para o funcionamento da mesma, está disponível no apêndice 'A' deste trabalho.

6.3.3. Função RotWord

A forma mais adequada de modelagem desta função com certeza é a criação de registradores de 32 bits que devem ser utilizados como entrada, e após a chamada da função lógica, esta realiza o deslocamento cíclico de um byte neste registrador conforme o que se encontra definido pelo algoritmo no capitulo 3 [08], retornando assim o dado correto.

```
30 architecture texte of rot_word is
31 begin
32 -- Rotação de words
33     text: process(clk,reset,in_data)
34     begin
35
36         if(reset = '1') then                    -- reset processo
37             out_data <= (others =>(others => '0'));
38
39         else                                    -- inicializa processo
40             if clk'event and clk='1' then       -- borda subida
41                 out_data(0) <= in_data(1);
42                 out_data(1) <= in_data(2);
43                 out_data(2) <= in_data(3);
44                 out_data(3) <= in_data(0);
45             end if;
46         end if;
47     end process;
48 end texte;
```

Figura 6.16 - Arquitetura da função *RotWord*.

Na figura 6.16 apresenta-se o código parcial em VHDL desta função, onde pode-se ver a declaração de arquitetura e a descrição comportamental de funcionamento da

função. Na linha 30, tem-se instanciado a arquitetura denominada *'texte'* a partir de um uma entidade denominada *'rot_word'*. A seguir na linha 31, tem-se a inicialização da arquitetura, que implementa uma estrutura comportamental de um registrador de 32 bits, como será visto no resultado de síntese RTL. Na linha 33 e 34 inicializa-se o processo da função e entre as linhas 35 e 46, executa-se a função propriamente dita, onde um laço *'IF'*, controlado pelo valor do pino *'reset'* e pelos eventos de relógio (clock), determina se o conteúdo da palavra de 32 bits na porta de entrada *'in_data'* será enviado para a porta de saída *'out_data'* de forma rotacionada, ou se a saída receberá o nível lógico *'0'*, conforme o estado do pino *'reset'*.

Agora tem-se que, a melhor descrição funcional desta operação de rotação é a que se encontra detalhada na figura 6.17, onde pode ser observado a descrição RTL desta função após a síntese no software de desenvolvimento. Lembrando que, como esta função não utiliza recursos de armazenamento de memória, agora utilizaremos elementos lógicos para sua implementação, conforme será visto nos resultados de desempenho.

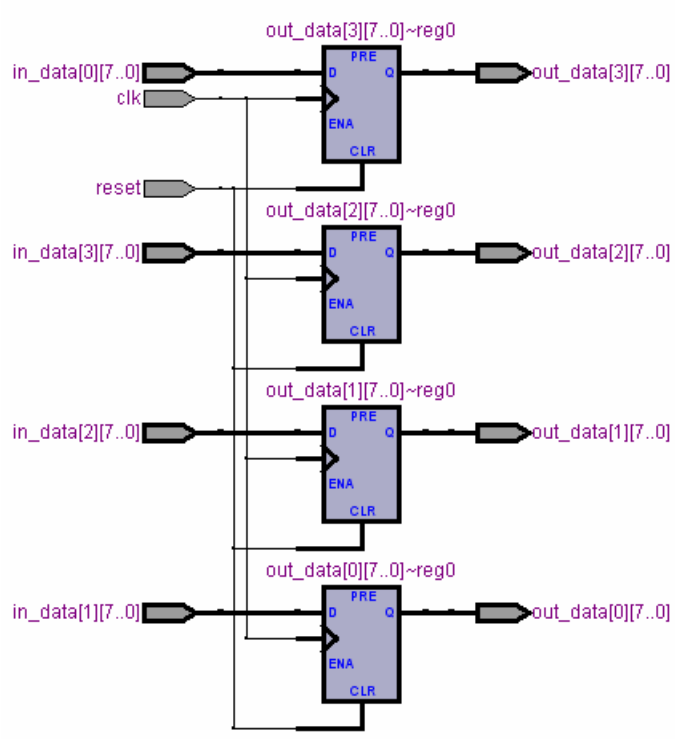


Figura 6.17 - Descrição RTL da função *RotWord.*

Já que agora esta função começa a trabalhar com o uso de lógica combinacional, é importante salientar que as funções apresentadas até este momento, buscam trabalhar com a implementação VHDL de forma comportamental, e no momento da implementação dos blocos principais, esta será realizada de forma estrutural.

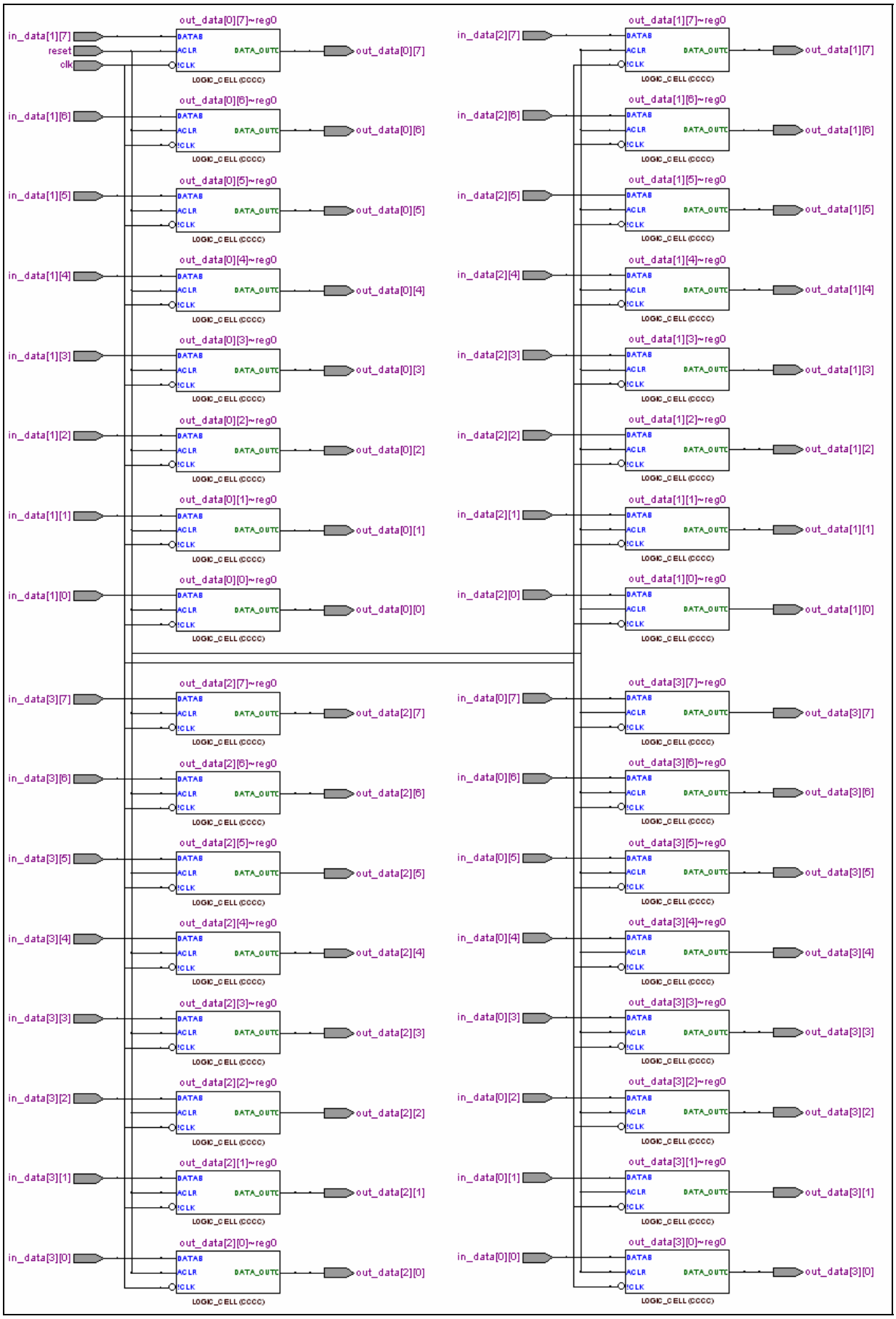


Figura 6.18 - Função *RotWord*, representação gráfica.

Na figura 6.18, com a intenção de se esclarecer ainda mais o funcionamento e o instanciamento da função lógica, pode-se observar como deve ser representada de forma gráfica esta função. Pode-se visualizar os sinais *reset* e *clk* e todas as portas e entrada/saída, necessárias para o funcionamento da função, como os vetores *in_data[i][i]* e *out_data[i][i]*, que, em uma implementação usando projeto gráfico, pode ser usada como modelo.

Com visto, temos a apresentação e instanciamento de cada bit do registrador de entrada em uma célula lógica, como um armazenamento temporário, sem utilizar modelagem de memória. Sendo assim, após o inicio do processo, o conteúdo destas células é enviado de forma síncrona para a saída realizando uma rotação nos bits.

Serão apresentados os resultados obtidos no software de desenvolvimento, da simulação da função. Na figura 6.19, observa-se o gráfico de forma de onda que representa os resultados de simulação do software QUARTUS II ®.

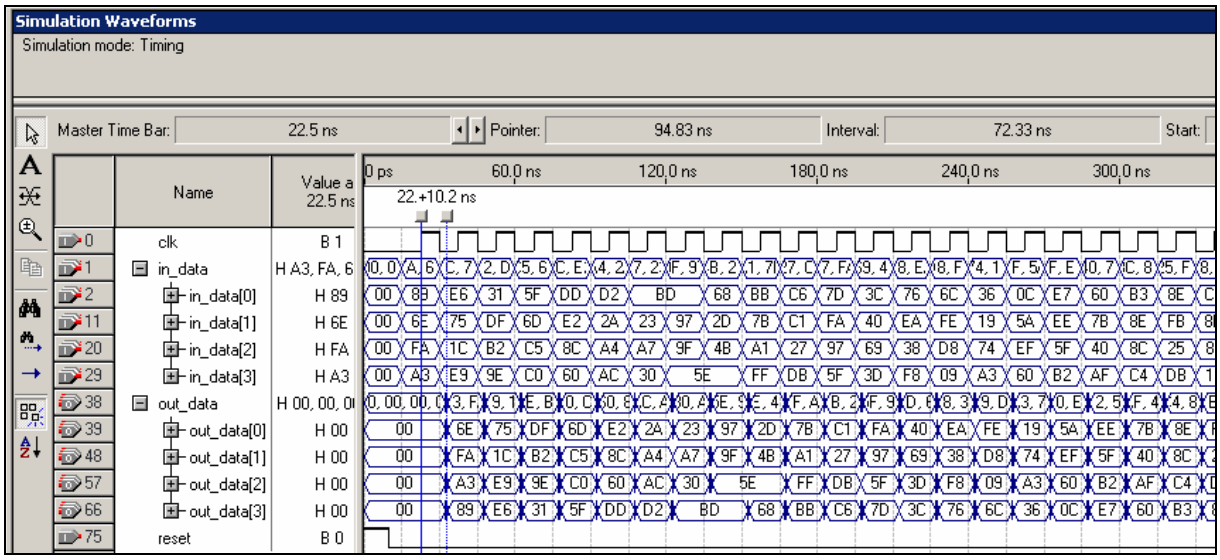


Figura 6.19 - Função *RotWord*, gráfico de forma de onda.

Pode-se observar na figura 6.20, que o correto funcionamento da implementação, corresponde ao esperado no algoritmo de criptografia [08]. É possível ver que quando se aplica um determinado valor na entrada (*in_data*) e após um determinado tempo (10,2ns) depois do evento de relógio, a saída (*out_data*) assume o resultado esperado, rotacionando de um byte o dado de entrada composto por uma *word* de 32 bits. Como exemplo basta observar o primeiro dado de entrada, “A3FA6E89”, que após o evento de relógio é transformado (rotacionado) para “89A3FA6E”.

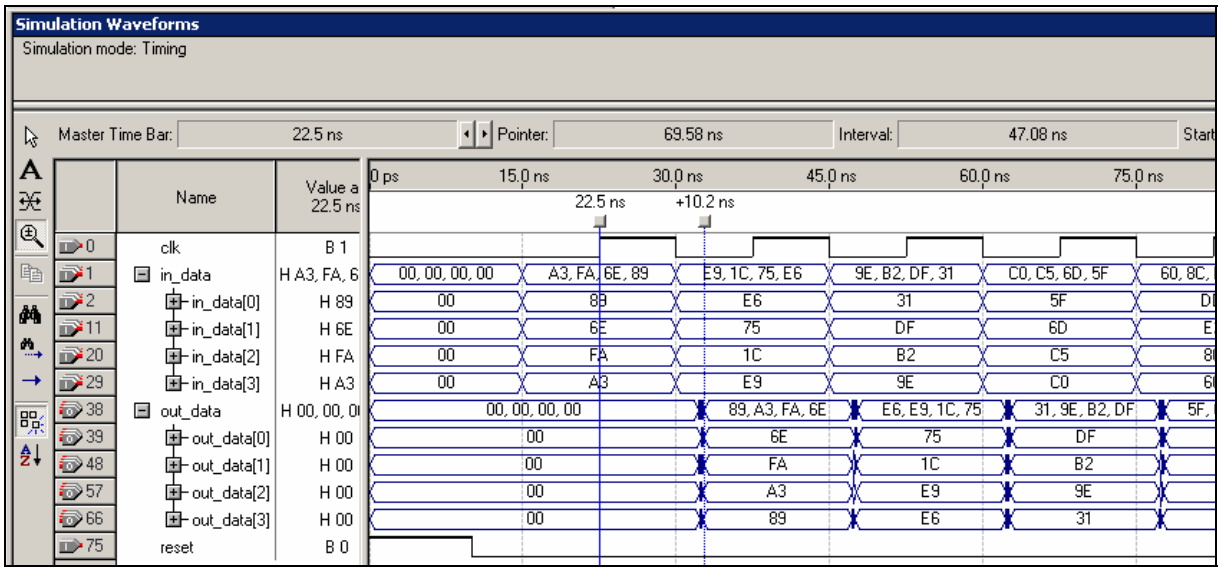


Figura 6.20 - Função *RotWord*, gráfico de forma de onda (detalhes).

Como visto na figura 6.20 podemos ver em detalhes o resultado da simulação da função, na qual foi aplicado um sinal de relógio com ciclos de 15ns, iniciando-se no instante 15ns, e que após o sinal *'reset'* mudar do nível lógico *'1'* para *'0'*, se deu inicio a execução da função. Durante o processo de execução, quando se aplica um evento de relógio, o conteúdo da porta de entrada onde se encontra o vetor *'in_data'* que foi instanciado com valores aleatórios a cada 15ns, é envidado para a porta de saída no vetor *'out_data'* com o valor modificado (rotacionado) conforme o resultado da simulação apresentada, correspondendo assim exatamente ao esperado pela descrição do algoritmo de criptografia [08]. Pode-se ver também uma marca aos 22,5ns e outra 10,2ns após, demonstrando o tempo de atraso máximo da função.

Na figura 6.21 pode ser o resultado completo, onde são apresentados os parâmetros de desempenho desta implementação.

Analysis & Synthesis Resource Usage Summary

	Resource	Usage
1	Total logic elements	32
2	Total combinational functions	0
3	-- Total 4-input functions	0
4	-- Total 3-input functions	0
5	-- Total 2-input functions	0
6	-- Total 1-input functions	0
7	-- Total 0-input functions	0
8	Total registers	32
9	I/O pins	66
10	Total memory bits	0

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time
1	Worst-case tsu	N/A	None	5.000 ns
2	Worst-case tco	N/A	None	10.200 ns
3	Worst-case th	N/A	None	-0.600 ns
4	Total number of failed paths			

Figura 6.21 - Função *RotWord*, resultados da implementação

Como apresentado na figura 6.21, viu-se que a avaliação de recursos consumidos pela implementação no dispositivo de lógica programável (FPGA), não apresentou consumo de espaço sintetizado como memória, porém usou 66 pinos de entrada e saída, 32 elementos lógicos e 32 registradores, não necessitando de funções combinacionais de nenhum tipo.

Agora no desempenho de analise de tempo de propagação, pode-se verificar na mesma figura 6.21, que o sumário apresenta os valores de temporização de pior caso variando entre -0,6ns a 10,2ns. Tem-se que: o tempo mínimo que o dado, *'in_data'* deve ficar disponível na entrada (t_{SU}) deve ser de 5,0ns; o tempo máximo que a função demora para retornar o dado válido (t_{CO}) na saída é de 10,2ns; o tempo mínimo que o dado *'in_data'* deve estar disponível na entrada depois de um evento de relógio (t_H) para que a função consiga processar a informação é de -0,6ns.

Neste caso em particular temos o tempo negativo (t_H) que pode ser explicado da seguinte forma: como o tempo que o dado tem que estar disponível na saída é negativo, significa que antes do dado ser passado para o próximo registrador, ele já foi transferido da entrada "D" do flip-flop posterior para a sua respectiva saída "Q", não necessitando esperar tempo algum para que a função seja executada corretamente. Este fato pode ocorrer devido a atrasos na temporização de *clock* das funções, uma vez que o atraso causado (*clock skew*) pode induzir a propagação do sinal de forma errada. Deve-se saber que se a entrada de sinal de *clock* ocorrer do ultimo registrador para o primeiro, então o atraso no *clock* pode ser uma vantagem no tempo de propagação do sinal, auxiliando muito os tempos de espera (t_H).

Como esta implementação não utilizada de sincronismo de dados de endereçamento, leitura e escrita como as memórias, não temos apresentação de dados de freqüência de operação. O código VHDL completo está disponível para análise no apêndice 'A' deste trabalho.

Capítulo 7

Conclusões e Trabalhos Futuros

Nesta dissertação vimos uma implementação parcial de blocos funcionais do algoritmo de criptografia AES (Rijndael) em um dispositivo de lógica programável (*FPGA*) do fabricante *Altera* utilizando linguagem de descrição de hardware VHDL. Com isto e após a apresentação dos resultados obtidos, poderemos agora realizar algumas ponderações.

Inicialmente, após apresentarmos os resultados de desempenho da implementação e os valores de temporização como visto no capitulo 6, vamos realizar uma extrapolação matemática e assim conseguir parâmetros teóricos de uma implementação geral da expansão de chaves, não avaliando os módulos de cifragem ou decifragem. Conforme visto no capítulo 3, o bloco funcional de expansão de chaves utiliza algumas sub-funções, entre elas a *byte_sub*, a r_con e a rot_word, assim com os valores de temporização apresentados, poderemos realizar uma aproximação para avaliar o tempo total desta implementação.

7.1. Conclusões de Desempenho

Apresentaremos agora uma avaliação do desempenho de temporização das funções, analisando os valores apresentados no capítulo anterior.

Conforme visto e discutido no capítulo 6 nos resultados de síntese e temporização da função *byte_sub*, necessitamos de um tempo mínimo de 11,5ns de espera para que o valor de entrada esteja estável (t_{SU}) e 13,0ns para que o dado esteja disponível na saída da função (t_{CO}), sendo que este valor se refere ao tempo mínimo necessário para se processar apenas a entrada de um byte. Assim, sabemos que o tempo de espera para que o dado esteja estabilizado na entrada pode ser sobreposto ao tempo de processamento e teremos no momento inicial o valor de 11,5 + 13 = 24,5ns para execução da função e a partir da segunda entrada e nos demais teremos apenas 13ns.

Já para a função *r_con*, temos o tempo de 14ns para que o dada esteja estável na saída (t_{CO}) e 8,1ns de espera para que o dado fique estável na entrada para processamento (t_{SU}). Realizando a mesma aproximação da função *byte_sub*, teremos para efeito de calculo, com uma sobreposição do tempo de processamento sobre o tempo de disponibilidade do dado no momento inicial o valor de 8,1 + 14 = 22,1ns, e da mesma forma serão necessários para as demais entradas apenas 14ns para a correta resposta da função.

Agora para a função *rot_word*, temos o tempo mínimo na entrada de 10,2ns (t_{CO}) e mais 5ns para estabilidade do dado de entrada (t_{SU}), porém de nada vale um tempo menor se teremos que implementar a função pelo tempo crítico da função de pior desempenho. Porém da mesma forma teríamos um tempo sobreposto de 10,2 + 5 = 15,2ns necessário para o momento inicial e apenas 5ns nos demais consecutivos. Lembrando que agora esta função processa entradas com tamanho de 32 bits e não mais 8 bits como nas funções byte_sub e r_con.

Para efeito de extrapolação matemática concluímos que um ciclo de clock de 15ns será o suficiente para o correto funcionamento das funções, uma vez que este valor e superior ao pior caso das funções e assim será utilizado na nossa extrapolação.

Conforme o pseudocódigo da função de expansão de chaves apresentada e discutido no capítulo 3 (figura 3.10), onde podemos entender o fluxo do processo e a quantidade de chamadas das funções, é possível verificar quantas vezes cada uma é necessária para processar a expansão e fornecer as sub-chaves de criptografia para a entrada na unidade de processamento. Assim conforme o visto na figura 3.10 teremos os seguintes resultados:

Temos que a função byte_sub é chamada 10 vezes no processo de expansão de chaves (apenas quando o índice do vetor de trabalho w[i] é múltiplo de quatro) e trabalha com um vetor de quatro bytes, apesar de processar apenas um byte por vez, poderemos instanciá-la lado a lado quatro vezes. Desta forma temos o seguinte tempo gasto: 10*15ns, que resultará em um tempo total estimado de 0,15us para o processamento das chaves de criptografia.

Já para a função *r_con*, que também é instanciada 10 vezes, temos que para o processamento das constantes de rodada: 10*15ns, com 0,15us para processamento desta função. Porém lembrando que esta função deve ser executada de forma paralela com a função anterior byte_sub, não necessitamos contabilizá-la novamente para um calculo final de desempenho.

Agora para a função *rot_word*, temos que, como a função é chamada dez vezes também durante a expansão de chaves teremos: 10*15ns, com 0,15us de tempo consumido. Agora, neste caso em particular, conforme visto no algoritmo de expansão, esta função deverá ser executada antes das funções de byte_sub, assim contabilizando o seu tempo gasto para o tempo total gasto.

Para finalizar o estudo de tempo consumido para se realizar a função de expansão de chaves, necessitamos conhecer ainda o tempo necessário para se realizar a operação *ou exclusivo* (XOR) usada neste bloco funcional. Sendo assim, realizamos um teste individual com a chamada da função XOR, entre duas words de 32 bits, e obtivemos valores de temporização inferiores aos tempos críticos necessários das funções, sendo assim utilizamos o valor de 15ns como tempo crítico para esta operação também, padronizando assim mais uma vez a nossa extrapolação matemática.

Como dentro da função de expansão de chaves necessitaremos fazer a operação XOR uma vez inicial depois mais dez vezes entre o retorno das funções *r_con* e a *temp*, mais quarenta e quatro vezes da *temp* com a *w[i-Nk]*, então termos 55 chamadas desta função, totalizando: 55*15ns, igual a 0,825us. Desta forma, como na figura 7.1 a seguir apresentamos uma descrição gráfica estrutural de como seria realizada a chamada da função no momento em que o índice do vetor de processamento w[*i*], fosse múltiplo de quatro. [08].

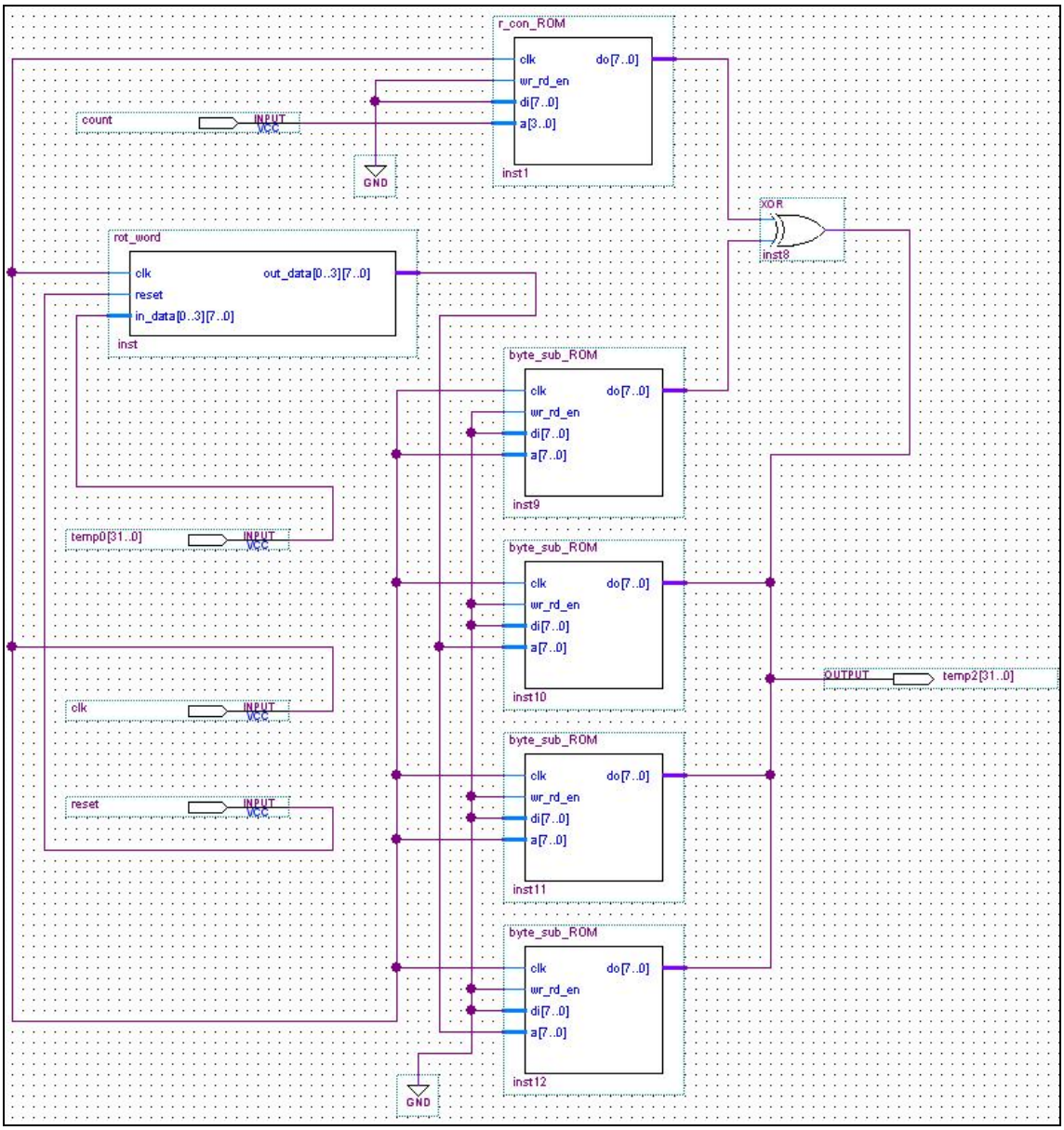


Figura 7.1 - Representação gráfica, função principal de expansão de chaves.

A seguir na tabela 7.1 apresentamos um resumo da extrapolação dos cálculos para uma idéia teórica do tempo necessário usado na função de expansão de chaves.

Tabela 7.1 - Resultados de desempenho expansão de chaves.

Função	*Número de Chamadas*	*Tempo Critico*	*Total*	*Total acumulado*
byte_sub	10	15	0,15us	-------
r_con	10	15	0,15us	0,15us
rot_word	10	15	0,15us	0,30us
xor	55	15	0,825us	0,825us
		TOTAL	***1,425us***	***1,125us***

Considerando que para a chamada da função de expansão de chaves, os dados de entrada (chave com 128 bits) já devem estar disponíveis nos registradores de entrada (assim não computado como tempo gasto), teremos o seguinte desempenho (apenas processamento de funções):

Throughput = 128 / 1,125 ≈ 108,5 Mb/s

Esta mesma extrapolação matemática pode ser realiza para a unidade de processamento, porém como não implementamos todas as funções para a implementação deste bloco funcional (*ShifTransf e MixColumns*), na será possível dimensionar o desempenho teórico.

Por fim, como conclusão deste trabalho, apresentamos a seguir na figura 7.2, um esquemático completo da implementação, demonstrando os blocos funcionais do algoritmo que foram efetivamente desenvolvidos neste trabalho, e os que faltam ser implementados para uma completa implementação do sistema. Lembrando que é importante possuir em mãos uma plataforma de desenvolvimento mais completa, como por exemplo, um FPGA com maior número de pinos de I/O para que seja possível implementar a carga dos dados de entrada (informação e chave) de uma só vez de forma paralela.

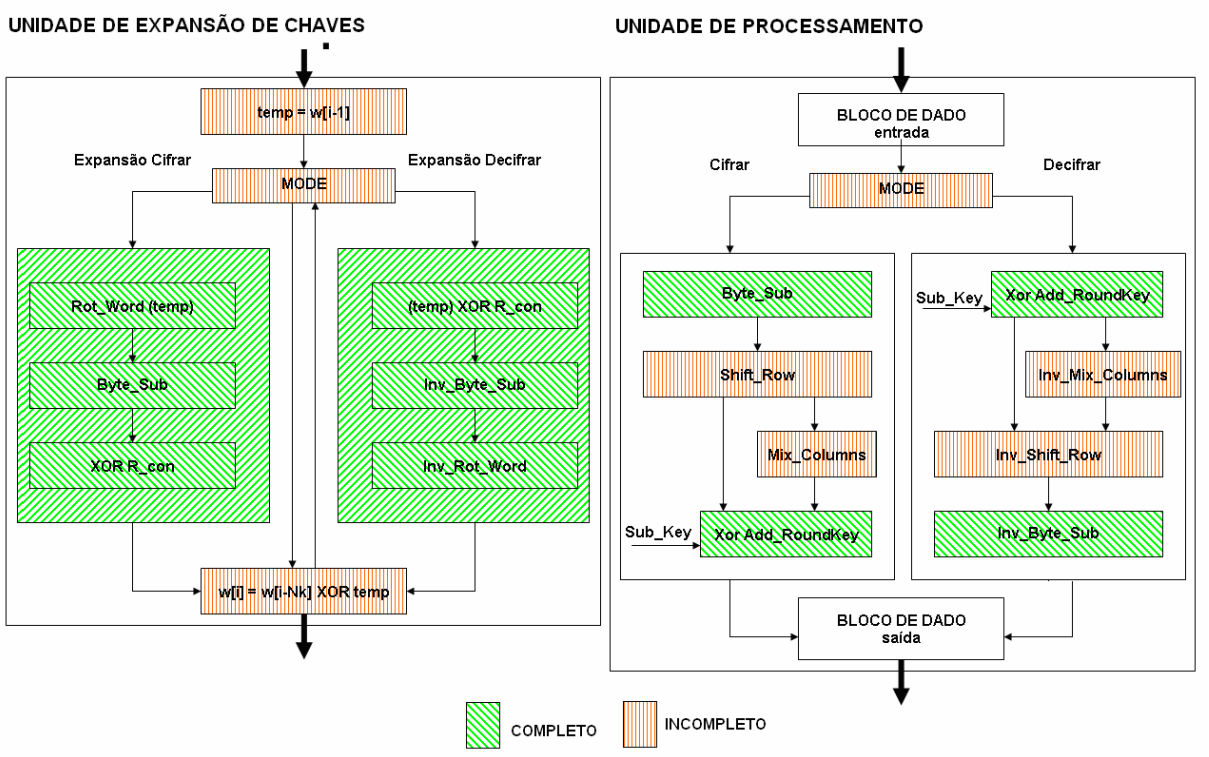


Figura 7.2 - Desenvolvimento do trabalho.

7.2. Sugestões e Trabalhos Futuros

Como a implementação deste trabalho não levou em consideração o maior uso de processamento paralelo, além de técnicas de *pipelined*, uma alternativa para outras implementações seria a utilização destas técnicas. Desta forma, poderão ser atingidas velocidades ainda maiores na implementação do algoritmo, além de otimizar os recursos lógicos no FPGA e também a migração para famílias de dispositivos mais novos e rápidos, e que possuem mais recursos, pois desta forma conseguiríamos atingir parâmetros de desempenho suficientemente rápidos para utilização deste projeto em sistemas de comunicação de alta desempenho.

Não poderíamos deixar de mencionar melhorias que poderiam ser implementadas no algoritmo de criptografia, assim sugerimos algumas modificações na forma com que se trata o processamento das chaves de criptografia. Conforme apresentado neste trabalho, a expansão de chaves deve ser realizada primeiro, para depois ser utilizada a sub-chave na respectiva rodada de criptografia, assim seria conveniente a criação de uma função de expansão de chaves que se tornasse independente de passos anteriores. Conforme visto neste trabalho [08], com todos os recursos de processamento paralelo e *pipelided*, a matemática atual do algoritmo não permite que se realize o processamento da sub-chave com a matriz de estado (byte a byte), uma vez que a geração desta sub-chave de rodada não esteja completamente pronta.

Como a geração de cada palavra de 32 bits que forma uma nova coluna na matriz de sub-chaves, depende de outra palavra quatro posições anteriores no vetor de expansão de chaves, não é permitido apagar ou perder este dado (palavra 32 bits) antes que este seja utilizado na expansão de sub-chaves seguintes e por conseqüência utilizá-lo imediatamente no processo de criptografia. Assim, não é possível realizar a produção byte a byte da chave e a sua utilização direta na matriz de estado sendo, no mínimo, necessária a produção de uma palavra de 32 bits antes de utilizá-la no processamento principal.

A pretensão a partir de agora é a busca por um trabalho complementar, onde poderemos identificar outras necessidades de mercado e trabalhá-las em uma futura tese de doutorado. Sendo assim, acreditamos que a o aperfeiçoamento das técnicas de implementação

deste algoritmo em linguagem de descrição de hardware (VHDL), e o maior conhecimento sobre os dispositivos de lógica programável (FPGAs), nos servirão de base para o próximo passo. Além da procura por outros algoritmos de criptografia e análise de suas características mais detalhadas, poderemos citar como futuras pretensões, a criação e desenvolvimento de um novo cripto-processador.

Acredita-se que a nova tendência de soluções de segurança seja a implementação em hardware de sistemas de criptografia, e temos que trabalhar ainda mais para desenvolver esta tecnologia, com a intenção de abastecer a demanda por soluções baratas, de alto desempenho e com satisfatório nível de segurança que virão.

Referências Bibliográficas

[01] TERADA, Routo. ***Segurança de Dados: Criptografia em Redes de Computador***. Edgard Blucher, 2000.

[02] TKOTZ, V.: ***Criptografia Numaboa***. Acesso em Dez 2007, disponível em: http://www.numaboa.com/content/section/11/57.

[03] MORENO E. D.; PEREIRA F. D.; CHIARAMONTE R. B.; ***Criptografia em Software e hardware***. Novatec. 2005.

[04] IBM Corporation, ***MARS – a candidate cipher for AES***. 1999.

[05] RIVEST R. L.; ROBSHAW M. J. B.; SIDNEY R.; YIN Y. L. ***The RC6 Block Cipher***. RSA Laboratories, 1998.

[06] ANDERSON, R.; BIHAM, E.; KNUDSEN, L. ***Serpent: A Proposal for the Advanced Encryption Standard***. 1999.

[07] SCHNEIER, B.; KELSEY, J.; WHITING, D.; WAGNER, D.; HALL, C.; FERGUSON, N.. ***Twofish: A 128-Bit Block Cipher.*** 1998.

[08] NIST ***Advanced Encryption Standard (AES)***. Página oficial do AES. 2001, disponível em: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[09] DAEMEM, J.; RIJMEN, V. ***AES Proposal: Rijndael***. 1999.

[10] MOREIRA, J. C.; FARRELL P. G., ***Essentials of Error-Control Coding.*** John Wiley & Sons, Ltd. 2006

[11] MORELOS-ZARAGOZA R. H., ***The Art of Error Correcting Coding.*** John Wiley & Sons, Ltd. 2006

[12] WAIN R.; BUSH I.; GUEST M.; DEEGAN M.; KOZIN I.; KITCHEN C.; ***An overview of FPGAs and FPGA programming.*** 2006

[13] BROWN S.; ROSE J.; ***Architecture of FPGAs and CPLDs: A Tutorial***. University of Toronto. 1996

[14] HEINKEL U.; ***VHDL Tutorial.*** Universität Erlangen-Nürnberg. 2000 disponível em: http://www.vhdl-online.de/~vhdl

[15] CHU, PONG P., ***RTL Hardware Design Using VHDL***. Wiley-Interscience. 2006

[16] PEDRONI, V. A., ***Circuit design with VHDL.*** MIT Press. 2004

[17] LIPSETT, R.; SCHAEFER C. F.; USSERY C., ***VHDL: Hardware Description and Design.*** Kluwer Academic Publishers. Ninth Printing 1992

[18] RUSHTON, A., ***VHDL for Logic Synthesis.*** McGraw-Hill Book Company. 1995.

[19] ALTERA CORPORATION., ***ACEX 1K Programmable Logic Device Family (Data Sheet).*** ALTERA Corporation. Maio 2003, versão 3.4.

Apêndice A

Códigos VHDL da Implementação

A seguir serão apresentados os códigos VHDL da implementação deste trabalho, juntamente com as respectivas identificações dos blocos funcionais. Em todos os códigos é possível identificar os parâmetros de comentários do programa que visa aumentar a documentação e compreensão dos mesmos.

```
1  --*****************************************************************************
2  -- Projeto     : AES128                                                       *
3  --                                                                           *
4  -- Nome do Bloco : aes_pkg.vhd                                               *
5  --                                                                           *
6  -- Autor       : Alessandro Campos                                           *
7  --                                                                           *
8  -- Email       : acampos@unifei.edu.br                                       *
9  --                                                                           *
10 -- Descrição   : Este pacote contém a declaração do tipo de word de 32 bits*
11 --               função coversão de byte para inteiro                       *
12 --                                                                           *
13 -- Para maires detalhe consultar documento FIPS-197                         *
14 --                                                                           *
15 --*****************************************************************************
16
17 LIBRARY ieee;
18 USE ieee.std_logic_1164.ALL;
19
20 PACKAGE aes_pkg IS
21     TYPE word_32_bits IS ARRAY (0 TO 3) OF std_logic_vector(7 DOWNTO 0);
22     FUNCTION suv2int (a: STD_LOGIC_VECTOR) RETURN NATURAL;
23 END aes_pkg;
24
25   ---------------------------------------------------------------------------
26
27 PACKAGE BODY aes_pkg IS
28
29 FUNCTION suv2int (l: STD_LOGIC_VECTOR) RETURN NATURAL IS
30 VARIABLE result: NATURAL := 0;
31 BEGIN
32     FOR t1 IN l'RANGE LOOP
33         result := result * 2;
34         IF l(t1) = '1' THEN
35             result := result + 1;
36         END IF;
37     END LOOP;
38 RETURN result;
39 END suv2int;
40
41 END PACKAGE BODY aes_pkg;
```

```
--*******************************************************************************
-- Projeto          : AES128                                                    *
--                                                                              *
-- Nome do Bloco    : byte_sub_ROM.vhd                                          *
--                                                                              *
-- Autor            : Alessandro Campos                                         *
--                                                                              *
-- Email            : acampos@unifei.edu.br                                     *
--                                                                              *
-- Descrição        : Este programa implementa e componetiza a função byte_sub  *
--                                                                              *
-- Para maiores detalhe consultar documento FIPS-197                            *
--                                                                              *
--*******************************************************************************

LIBRARY ieee;
    USE ieee.std_logic_1164.ALL;
    USE work.aes_pkg.ALL;

-------------------------------------------------------------------------

ENTITY byte_sub_ROM IS
    PORT(
        clk     : IN  STD_LOGIC;
        wr_rd_en: IN  STD_LOGIC;
        di      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        a       : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        do      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END byte_sub_ROM;

-------------------------------------------------------------------------

ARCHITECTURE rtl OF byte_sub_ROM IS
    TYPE ram_t IS ARRAY (0 TO 255) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL memory: ram_t := (X"63",X"7C",X"77",X"7B",X"F2",X"6B",X"6F",X"C5",
                             X"30",X"01",X"67",X"2B",X"FE",X"D7",X"AB",X"76",
                             X"CA",X"82",X"C9",X"7D",X"FA",X"59",X"47",X"F0",
                             X"AD",X"D4",X"A2",X"AF",X"9C",X"A4",X"72",X"C0",
                             X"B7",X"FD",X"93",X"26",X"36",X"3F",X"F7",X"CC",
                             X"34",X"A5",X"E5",X"F1",X"71",X"D8",X"31",X"15",
                             X"04",X"C7",X"23",X"C3",X"18",X"96",X"05",X"9A",
                             X"07",X"12",X"80",X"E2",X"EB",X"27",X"B2",X"75",
                             X"09",X"83",X"2C",X"1A",X"1B",X"6E",X"5A",X"A0",
                             X"52",X"3B",X"D6",X"B3",X"29",X"E3",X"2F",X"84",
                             X"53",X"D1",X"00",X"ED",X"20",X"FC",X"B1",X"5B",
                             X"6A",X"CB",X"BE",X"39",X"4A",X"4C",X"58",X"CF",
                             X"D0",X"EF",X"AA",X"FB",X"43",X"4D",X"33",X"85",
                             X"45",X"F9",X"02",X"7F",X"50",X"3C",X"9F",X"A8",
                             X"51",X"A3",X"40",X"8F",X"92",X"9D",X"38",X"F5",
                             X"BC",X"B6",X"DA",X"21",X"10",X"FF",X"F3",X"D2",
                             X"CD",X"0C",X"13",X"EC",X"5F",X"97",X"44",X"17",
                             X"C4",X"A7",X"7E",X"3D",X"64",X"5D",X"19",X"73",
                             X"60",X"81",X"4F",X"DC",X"22",X"2A",X"90",X"88",
                             X"46",X"EE",X"B8",X"14",X"DE",X"5E",X"0B",X"DB",
                             X"E0",X"32",X"3A",X"0A",X"49",X"06",X"24",X"5C",
                             X"C2",X"D3",X"AC",X"62",X"91",X"95",X"E4",X"79",
                             X"E7",X"C8",X"37",X"6D",X"8D",X"D5",X"4E",X"A9",
                             X"6C",X"56",X"F4",X"EA",X"65",X"7A",X"AE",X"08",
                             X"BA",X"78",X"25",X"2E",X"1C",X"A6",X"B4",X"C6",
                             X"E8",X"DD",X"74",X"1F",X"4B",X"BD",X"8B",X"8A",
                             X"70",X"3E",X"B5",X"66",X"48",X"03",X"F6",X"0E",
                             X"61",X"35",X"57",X"B9",X"86",X"C1",X"1D",X"9E",
                             X"E1",X"F8",X"98",X"11",X"69",X"D9",X"8E",X"94",
                             X"9B",X"1E",X"87",X"E9",X"CE",X"55",X"28",X"DF",
                             X"8C",X"A1",X"89",X"0D",X"BF",X"E6",X"42",X"68",
                             X"41",X"99",X"2D",X"0F",X"B0",X"54",X"BB",X"16");
    SIGNAL iaddr: integer range 0 to 255;
```

```
69
70 BEGIN
71
72     iaddr <= suv2int(a);
73
74     PROCESS (clk,iaddr,memory)
75     BEGIN
76         IF clk'EVENT AND clk = '1' THEN
77             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
78                 do <= memory(iaddr);
79             ELSE
80                 memory(iaddr) <= di;    --  processo de escrita na RAM
81             END IF;
82         END IF;
83     END PROCESS;
84 END rtl;
```

```
1  --*****************************************************************************
2  -- Projeto          : AES128                                                  *
3  --                                                                           *
4  -- Nome do Bloco    : inv_byte_sub_ROM.vhd                                    *
5  --                                                                           *
6  -- Autor            : Alessandro Campos                                       *
7  --                                                                           *
8  -- Email            : acampos@unifei.edu.br                                   *
9  --                                                                           *
10 -- Descrição        : Este programa implementa e compon. a função inv_byte_sub *
11 --                                                                           *
12 -- Para maiores detalhe consultar documento FIPS-197                         *
13 --                                                                           *
14 --*****************************************************************************
15
16 LIBRARY ieee;
17     USE ieee.std_logic_1164.ALL;
18     USE work.aes_pkg.ALL;
19
20 ---------------------------------------------------------------------------
21
22 ENTITY inv_byte_sub_ROM IS
23     PORT(
24         clk     : IN  STD_LOGIC;
25         wr_rd_en: IN  STD_LOGIC;
26         di      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
27         a       : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
28         do      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
29         );
30 END inv_byte_sub_ROM;
31
32 ---------------------------------------------------------------------------
33
34 ARCHITECTURE rtl OF inv_byte_sub_ROM IS
35     TYPE ram_t IS ARRAY (0 TO 255) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
36     SIGNAL memory: ram_t := (X"52",X"09",X"6a",X"d5",X"30",X"36",X"a5",X"38",
37                              X"bf",X"40",X"a3",X"9e",X"81",X"f3",X"d7",X"fb",
38                              X"7c",X"e3",X"39",X"82",X"9b",X"2f",X"ff",X"87",
39                              X"34",X"8e",X"43",X"44",X"c4",X"de",X"e9",X"cb",
40                              X"54",X"7b",X"94",X"32",X"a6",X"c2",X"23",X"3d",
41                              X"ee",X"4c",X"95",X"0b",X"42",X"fa",X"c3",X"4e",
42                              X"08",X"2e",X"a1",X"66",X"28",X"d9",X"24",X"b2",
43                              X"76",X"5b",X"a2",X"49",X"6d",X"8b",X"d1",X"25",
44                              X"72",X"f8",X"f6",X"64",X"86",X"68",X"98",X"16",
45                              X"d4",X"a4",X"5c",X"cc",X"5d",X"65",X"b6",X"92",
46                              X"6c",X"70",X"48",X"50",X"fd",X"ed",X"b9",X"da",
47                              X"5e",X"15",X"46",X"57",X"a7",X"8d",X"9d",X"84",
48                              X"90",X"d8",X"ab",X"00",X"8c",X"bc",X"d3",X"0a",
49                              X"f7",X"e4",X"58",X"05",X"b8",X"b3",X"45",X"06",
50                              X"d0",X"2c",X"1e",X"8f",X"ca",X"3f",X"0f",X"02",
51                              X"c1",X"af",X"bd",X"03",X"01",X"13",X"8a",X"6b",
52                              X"3a",X"91",X"11",X"41",X"4f",X"67",X"dc",X"ea",
53                              X"97",X"f2",X"cf",X"ce",X"f0",X"b4",X"e6",X"73",
54                              X"96",X"ac",X"74",X"22",X"e7",X"ad",X"35",X"85",
55                              X"e2",X"f9",X"37",X"e8",X"1c",X"75",X"df",X"6e",
56                              X"47",X"f1",X"1a",X"71",X"1d",X"29",X"c5",X"89",
57                              X"6f",X"b7",X"62",X"0e",X"aa",X"18",X"be",X"1b",
58                              X"fc",X"56",X"3e",X"4b",X"c6",X"d2",X"79",X"20",
59                              X"9a",X"db",X"c0",X"fe",X"78",X"cd",X"5a",X"f4",
60                              X"1f",X"dd",X"a8",X"33",X"88",X"07",X"c7",X"31",
61                              X"b1",X"12",X"10",X"59",X"27",X"80",X"ec",X"5f",
62                              X"60",X"51",X"7f",X"a9",X"19",X"b5",X"4a",X"0d",
63                              X"2d",X"e5",X"7a",X"9f",X"93",X"c9",X"9c",X"ef",
64                              X"a0",X"e0",X"3b",X"4d",X"ae",X"2a",X"f5",X"b0",
65                              X"c8",X"eb",X"bb",X"3c",X"83",X"53",X"99",X"61",
66                              X"17",X"2b",X"04",X"7e",X"ba",X"77",X"d6",X"26",
67                              X"e1",X"69",X"14",X"63",X"55",X"21",X"0c",X"7d");
68     SIGNAL iaddr: integer range 0 to 255;
```

```
69
70 BEGIN
71
72     iaddr <= suv2int(a);
73
74     PROCESS (clk,iaddr,memory)
75     BEGIN
76         IF clk'EVENT AND clk = '1' THEN
77             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
78                 do <= memory(iaddr);
79             ELSE
80                 memory(iaddr) <= di;    --  processo de escrita na RAM
81             END IF;
82         END IF;
83     END PROCESS;
84 END rtl;
```

```
1  --*****************************************************************************
2  -- Projeto          : AES128                                                  *
3  --                                                                           *
4  -- Nome do Bloco    : r_con_ROM.vhd                                           *
5  --                                                                           *
6  -- Autor            : Alessandro Campos                                       *
7  --                                                                           *
8  -- Email            : acampos@unifei.edu.br                                   *
9  --                                                                           *
10 -- Descrição        : Este programa implementa e componetiza a funação r_con_ROM*
11 --                                                                           *
12 -- Para maiores detalhe consultar documento FIPS-197                         *
13 --                                                                           *
14 --*****************************************************************************
15
16 LIBRARY ieee;
17     USE ieee.std_logic_1164.ALL;
18     USE work.aes_pkg.ALL;
19
20 ---------------------------------------------------------------------------
21
22 ENTITY r_con_ROM IS
23     PORT(
24         clk     : IN  STD_LOGIC;
25         wr_rd_en: IN  STD_LOGIC;
26         di      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
27         a       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
28         do      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
29         );
30 END r_con_ROM;
31
32 ---------------------------------------------------------------------------
33
34 ARCHITECTURE rtl OF r_con_ROM IS
35     TYPE ram_t IS ARRAY (0 TO 9) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
36     SIGNAL memory: ram_t := (X"01",X"02",X"04",X"08",X"10",X"20",X"40",X"80",
37                              X"1B",X"36");
38     SIGNAL iaddr: integer range 0 to 9;
39
40 BEGIN
41 iaddr <= suv2int(a);
42     PROCESS (clk)
43     BEGIN
44         IF clk'EVENT AND clk = '1' THEN
45             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
46                 do <= memory(iaddr);
47             ELSE
48                 memory(iaddr) <= di;    --  processo de escrita na RAM
49             END IF;
50         END IF;
51     END PROCESS;
52 END rtl;
```

```
1  --*****************************************************************************
2  -- Projeto          : AES128                                                  *
3  --                                                                           *
4  -- Nome do Bloco    : inv_r_con_ROM.vhd                                       *
5  --                                                                           *
6  -- Autor            : Alessandro Campos                                       *
7  --                                                                           *
8  -- Email            : acampos@unifei.edu.br                                   *
9  --                                                                           *
10 -- Descrição        : Este programa implementa e compon. a funação inv_r_con_ROM*
11 --                                                                           *
12 -- Para maiores detalhe consultar documento FIPS-197                         *
13 --                                                                           *
14 --*****************************************************************************
15
16 LIBRARY ieee;
17     USE ieee.std_logic_1164.ALL;
18     USE work.aes_pkg.ALL;
19
20 ---------------------------------------------------------------------------
21
22 ENTITY inv_r_con_ROM IS
23     PORT(
24         clk     : IN  STD_LOGIC;
25         wr_rd_en: IN  STD_LOGIC;
26         di      : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
27         a       : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
28         do      : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
29         );
30 END inv_r_con_ROM;
31
32 ---------------------------------------------------------------------------
33
34 ARCHITECTURE rtl OF inv_r_con_ROM IS
35     TYPE ram_t IS ARRAY (0 TO 9) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
36     SIGNAL memory: ram_t := (X"36",X"1B",X"80",X"40",X"20",X"10",X"08",X"04",
37                              X"02",X"01");
38     SIGNAL iaddr: integer range 0 to 9;
39
40 BEGIN
41 iaddr <= suv2int(a);
42     PROCESS (clk)
43     BEGIN
44         IF clk'EVENT AND clk = '1' THEN
45             IF wr_rd_en = '0' THEN      --  processo de leitura da RAM
46                 do <= memory(iaddr);
47             ELSE
48                 memory(iaddr) <= di;    --  processo de escrita na RAM
49             END IF;
50         END IF;
51     END PROCESS;
52 END rtl;
```

```
1  --*******************************************************************************
2  -- Projeto          : AES128                                                    *
3  --                                                                             *
4  -- Nome do Bloco    : rot_word.vhd                                              *
5  --                                                                             *
6  -- Autor            : Alessandro Campos                                         *
7  --                                                                             *
8  -- Email            : acampos@unifei.edu.br                                     *
9  --                                                                             *
10 -- Descrição        : Este programa implementa e componetiza a funação rot_word *
11 --                                                                             *
12 -- Para maiores detalhe consultar documento FIPS-197                           *
13 --                                                                             *
14 --*******************************************************************************
15
16 library ieee;
17     use ieee.std_logic_1164.all;
18     use ieee.std_logic_unsigned.all;
19     use work.aes_pkg.all;
20
21 ---------------------------------------------------------------------------
22
23 entity rot_word is
24 port(
25       clk       : in std_logic;
26       reset     : in std_logic;
27       in_data   : in word_32_bits;              -- Fornecendo entrada
28       out_data  : out word_32_bits              -- Fornecendo saída
29      );
30 end rot_word;
31
32 ---------------------------------------------------------------------------
33
34 architecture texte of rot_word is
35 begin
36 -- Rotação de words
37     text: process(clk,reset,in_data)
38     begin
39
40         if(reset = '1') then                    -- reset processo
41             out_data <= (others =>(others => '0'));
42
43         else                                    -- inicializa processo
44             if clk'event and clk='1' then       -- borda subida
45                 out_data(0) <= in_data(1);
46                 out_data(1) <= in_data(2);
47                 out_data(2) <= in_data(3);
48                 out_data(3) <= in_data(0);
49             end if;
50         end if;
51     end process;
52 end texte;
```