UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Júlio César Soares Américo Filho

Análise e implementação de uma arquitetura iterativa com *sub-pipelining* de 3 estágios e *datapath* de 32 bits para um co-processador AES-128

Campinas

2016

UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Júlio César Soares Américo Filho

Análise e implementação de uma arquitetura iterativa com *sub-pipelining* de 3 estágios e *datapath* de 32 bits para um co-processador AES-128

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Telecomunicações e Telemática.

Orientador: Prof. Dr. Luís Geraldo Pedroso Meloni

Este exemplar corresponde à versão final da tese defendida pelo aluno Júlio César Soares Américo Filho, e orientada pelo Prof. Dr. Luís Geraldo Pedroso Meloni

_____

Campinas

2016

Agência(s) de fomento e nº(s) de processo(s): CAPES

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Luciana Pietrosanto Milla - CRB 8/8129

Am35a

Américo Filho, Júlio César Soares, 1987-
Análise e implementação de uma arquitetura iterativa com sub-pipelining de estágios e datapath de 32 bits para um co-processador AES-128 / Júlio Cesar Soares Américo Filho. – Campinas, SP : [s.n.], 2016.

Orientador: Luís Geraldo Pedroso Meloni.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Algoritmos. 2. Hardware - Arquitetura. 3. Computadores canalizados. 4. Criptografia. 5. FPGA (Field Programmable Gate Array). I. Meloni, Luís Geraldo Pedroso,1958-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Analysis and Implementation of an iterative architecture with 3 stages pipeline and 32 bits datapath to an AES-128 co-processor
Palavras-chave em inglês:
Algorithms
Hardware - Architecture
Channeled computers
Encryption
FPGA (Field Programmable Gate Array)
Área de concentração: Telecomunicações e Telemática
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Luís Geraldo Pedroso Meloni [Orientador]
Marco Aurélio Amaral Henriques
Osamu Saotome
Data de defesa: 12-08-2016
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Júlio César Soares Américo Filho **RA:** 161594
Data da Defesa: 12 de Agosto de 2016
Título da Tese:
"Análise e implementação de uma arquitetura iterativa com *sub-pipelining* de 3 estágios e *datapath* de 32 bits para um co-processador AES-128"

Prof. Dr. Luís Geraldo Pedroso Meloni (Presidente, FEEC/UNICAMP)
Prof. Dr. Osamu Saotome (Instituto Tecnológico da Aeronáutica)
Prof. Dr. Marco Aurélio Amaral Henriques(FEEC/UNICAMP)

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no processo de vida acadêmica do aluno.

Resumo

Neste trabalho, propõe-se uma arquitetura de *hardware* para um co-processador capaz de realizar cifração e decifração segundo o padrão AES-128 com suporte aos modos de operação ECB, CBC e CTR. A arquitetura proposta emprega as técnica de *loop rolling* com compartilhamento de recursos (para reduzir a quantidade de lógica necessária) e *sub-pipeling* (para aumentar a frequência de operação do circuito). A largura do *datapath* é 32 bits e o número de estágios do *pipeline* é três. Também documenta-se os resultados do projeto *openAES*. O *openAES* é um projeto *open source* desenvolvido a partir deste trabalho e que disponibiliza um *IP Core* de um co-processador AES compatível com o protocolo AMBA APB. O *IP Core* do projeto *OpenAES* faz uso da arquitetura proposta na primeira parte deste trabalho, adicionando a ela diversas funcionalidades, como suporte a DMA, geração de interrupções e possibilidade de suspensão de mensagens. Como resultados do projeto, são disponibilizados: o RTL, em Verilog, do IP Core, um ambiente de verificação funcional, uma camada de abstração de hardware (HAL), escrita em C, compatível com o padrão ARM CMSIS e um *script* de *timing constraints* no formato SDC. Como forma de validação, o IP foi prototipado em um dispositivo SmartFusion A2F200M3F.

Palavras-chaves: Algoritmo AES, Criptografia, Co-processador, Arquitetura Iterativa, *Sub-pipelining*, *openAES*, IP Core.

Abstract

This work proposes an AES-128 hardware architecture that supports both encryption and decryption for the ECB, CBC and CTR modes. The datapath width is 32 bits and the number of pipeline stages is 3. This work also documents the OpenAES project. The OpenAES is an open source project that provides an IP-Core for an AES co-processor that is compatible with the AMBA APB protocol and is based on the architecture described in the first part of this work. Several features such as DMA capabilites, interruptions generations and suport to message priorization are added to the basic architecture. The project provides: the synthesizable RTL Verilog for the IP Core, a function verification enviroment, a hardware abstraction layer compatible with the CMSIS standard and a SDC timing constraints file. The IP validation was peformed through a SmartFusion A2F200M3F device.

Keywords: AES Algorithm, Cryptography, Co-processor, Iterative architecture, Sub-pipelining, OpenAES, IP Core.

Lista de ilustrações

Lista de tabelas

Lista de Abreviaturas

$AACS$	Advanced Access Content System
AES	Advanded Encryption Standard
$AMBA$	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM	Advanced Risc Machine
$ASIC$	Aplication Specific Integrated Circuit
CBC	Cipher Block Chaining
$CMSIS$	Cortex Microcontroller Software Interface Standard
CSS	Content Scramble System
CTR	Counter Mode
DMA	Direct Memory Access
DPA	Differential Power Analysis
ECB	Electronic CodeBook
EMA	Electromagnetic Analysis
$FPGA$	Field-Programmable Gate Array
GE	Gate Equivalent
HAL	Hardware Abstraction Layer
IoT	Internet of Things
MEF	Máquina de Estados Finitos
$NIST$	National Institute of Standards and Technology
NOP	No Operation
RTL	Register Transfer Level
SCA	Side Channel Attack
SoC	System on a Chip

SPA	Simple Power Analysis
TLS	Transport Layer Security
WPA	Wi-Fi Protected Access
VCD	Value Change Dump

Sumário

1 Introdução

O avanço contínuo da tecnologia de semicondutores possibilitou o desenvolvimento de sistemas digitais altamente integrados, de baixo custo e alto desempenho. Com isso, diversas soluções, antes analógicas, migraram para versões digitais, resultando em ganho de desempenho e redução de custo e novas aplicações, sem quaisquer contraparte analógica, surgiram. Paralelamente, uma infraestrutura global de comunicação de dados digitais foi desenvolvida, o que permitiu a conexão pervasiva de todos esses novos dispositivos.

Este cenário, em que diversos dispositivos heterogêneos trocam dados entre si, e a necessidade de garantir a confidencialidade, a integridade e a autenticidade das informações trafegadas motivaram o uso de protocolos de criptografia, antes restritos a aplicações governamentais e militares, em aplicações comerciais.

Hoje, os protocolos de criptografia são utilizados em uma ampla gama de aplicações e são parte integrante de diversos protocolos de comunicação. Os protocolos TLS (DIERKS; RESCORLA, 2008) e WPA (IEEE; 802.11I, 2004), por exemplo, são protocolos de criptografia utilizados em redes de computadores com e sem fio, respectivamente. Outros usos, como proteção de conteúdo e assinatura digital, agora são comuns. Os protocolos CSS (do inglês *Content Scramble System*) e AACS (do inglês, *Advanced Access Content System*), por exemplo, são utilizados em DVD e Blu-Ray, respectivamente, para evitar cópia não autorizada de conteúdo.

Em seu cerne, os protocolos de criptografia utilizam um ou mais algoritmos para fornecer suas primitivas de segurança. Esses algoritmos de criptografia, em termos gerais, são de dois tipos: simétricos e assimétricos (estritamente falando, as funções *hash* poderiam ser considerados como uma terceira classe de algoritmos (PAAR; PELZL, 2009)). Os algoritmos simétricos são aqueles em que uma mesma chave é utilizada nos processos de encriptação e decriptação. Já os algoritmos assimétricos são aqueles em que, além de uma chave privada, os processos de cifração e decifração utilizam uma chave pública (PAAR; PELZL, 2009).

Dentre os algoritmos simétricos, o AES (FIPS..., 2001) ocupa posição de destaque (MANGARD *et al.*, 2003) e tem sido utilizado em um amplo espectro de aplicações, como nas redes de sensores sem fio ZigBee (ZIGBEE..., 2004) e IEEE 812.15.4 (IEEE..., 2012), nos protocolos de rede IPSec (KENT, 2005), TLS (DIERKS; RESCORLA, 2008), SSH (YLONEN; LONVICK, 2006), WPA (IEEE; 802.11I, 2004), nos padrões sem fio WiFi (IEEE..., 2010) e WiMAX (AHSON; ILYAS, 2008), em controladores de disco de estado sólido (WU *et al.*, 2010), em sistemas de arquivos com suporte a criptografia, como o EFS (MICROSOFT, 2016), em tags RFID (FELDHOFER; WOLKERSTORFER,

2007), dentre outros.

1.1 Motivação

Desde a padronização do AES em 2001 pela agência americana NIST (FIPS..., 2001), uma enorme quantidade de pesquisa tem sido feita com o intuito de derivar implementações eficientes desse algoritmo. Em relação as implementações em *hardware*, esse esforço de pesquisa tem seguido pelo menos quatro direções.

De um lado, há os trabalhos que focam em arquiteturas de alto desempenho para os modos de cifração sem *feedback* (ZHANG; PARHI, 2004; MORIOKA; SATOH, 2002; JäRVINEN *et al.*, 2003; MAYER *et al.*, 2002; VERBAUWHEDE *et al.*, 2003; MOZAFFARI-KERMANI; REYHANI-MASOLEH, 2012; LUTZ *et al.*, 2002; IYER *et al.*, 2006; FAN; HWANG, 2007; KOTTURI *et al.*, 2005). Nesses trabalhos, os compromissos de projeto são explorados com o intuito de se atingir o maior *throughput* possível, independente da área final ocupada pelo *hardware*. Essas arquiteturas exploram ao máximo o paralelismo inerente ao algoritmo AES e fazem um uso agressivo de *pipeline*, o que resulta em taxas de processamento na ordem de Gigabits por segundo. O preço que se paga é o elevado número de recursos lógicos utilizados e o alto consumo de potência.

No outro extremo, há os trabalhos que propõem arquiteturas compactas com foco em baixo consumo e redução de área (FELDHOFER *et al.*, 2005; HSIAO *et al.*, 2006; HAMALAINEN *et al.*, 2006; GAJ, 2003; SATOH *et al.*, 2001; NEDJAH *et al.*, 2006). Essas arquiteturas são adequadas para aplicações em que há escassez de recursos computacionais, como smartcards, tags RFID e nós de redes de sensores sem fio. Nesses trabalhos, os compromissos de projeto são explorados com o intuito de se atingir a menor área possível, mesmo que isso implique em baixas taxas de processamento. Essas arquiteturas fazem um uso massivo de compartilhamento de recursos, o que possibilita o desenvolvimento de soluções com *datapath* de 64, 32 e mesmo 8 bits. Essas modificações de arquitetura, no entanto, têm um impacto negativo no caminho crítico do circuito e no número de ciclos de *clock* necessários para se completar o algoritmo de criptografia.

Outra linha de pesquisa bastante ativa é o estudo de otimizações nas operações básicas do algoritmo AES. O objetivo central é reduzir o custo computacional das principais transformações realizadas pelo algoritmo, como *SBox* e *MixColumns*. Os trabalhos nessa área, como (SATOH *et al.*, 2001; FISCHER *et al.*, 2005; KUO; VERBAUWHEDE, 2001; RUDRA *et al.*, 2001; CANRIGHT, 2005; MENTENS *et al.*, 2005; MORIOKA; SATOH, 2003), exploram a estrutura algébrica dessas operações com o intuito de derivar novas formas de implementação que sejam eficientes em termos de algum parâmetro de projeto, como velocidade de operação, área ocupada ou potência dissipada. Em (SATOH *et al.*, 2001), por exemplo, o autor propõe uma estrutura de *SBox* compacta, já em (MORIOKA;

SATOH, 2003), o autor propõe uma arquitetura de *SBox* com baixo consumo de energia.

Embora o algoritmo AES seja seguro, isto é, não há nenhum método analítico de criptoanálise conhecido cujo desempenho seja superior ao método da força bruta (BOGDANOV *et al.*, 2011), há uma classe de ataques, denominados SCA (do inglês, *Side Channel Attack*), que exploram as fraquezas de implementação com o intuito de quebrar a solução de criptografia. Os principais tipos de SCA são SPA (do inglês, *Simple Power Analysis*), DPA (do inglês, *Differential Power Analysis*) e EMA (do inglês, *Electromagnetic Analysis*) (SHAN *et al.*, 2015). Uma implementação robusta do AES deve ser resistente a esses tipos de ataques. Assim, há um ramo de pesquisa dedicado ao estudo de contramedidas para ataques do tipo SCA. Os trabalhos (SHAN *et al.*, 2015; TIRI *et al.*, 2005; CANRIGHT; BATINA, 2008) são exemplos de pesquisa que abordam esse problema.

Do exposto acima, fica claro que os compromissos de projeto (considerando principalmente *throughput* e área) já estão relativamente bem explorados. Isto é, há uma quantidade considerável de trabalhos que focam em arquiteturas de alto desempenho e em arquiteturas compactas. No entanto, há pouca literatura dedicada ao estudo de arquiteturas moderadas, i.e, arquiteturas de médio *throughput* e média área. O trabalho (WU *et al.*, 2010) é um dos poucos que segue essa linha. Nesse trabalho, o autor propõe uma arquitetura iterativa com *sub-pipeline* de três estágios e um *datapath* de 128 bits para ser usada em controladores de disco de estado sólido.

O autor do presente trabalho desconhece qualquer publicação que explora uma arquitetura iterativa com *sub-pipelining*, como em (WU *et al.*, 2010), mas com um *datapath* de largura inferior a 128 bits. Assim, há um interesse natural, do ponto de vista de pesquisa, em se explorar as peculiaridades, as forças e as fraquezas desse tipo de arquitetura.

A aplicabilidade deste tipo de arquitetura fica clara quando se considera, por exemplo, que foram produzidos, durante o primeiro quadrimestre de 2016, cerca de 1.8 bilhões de chips contendo processadores ARM da família Cortex-M (ARM, a). Esses processadores possuem arquitetura de 32 bits e são apropriados para aplicações que requerem de baixo a médio poder computacional. Portanto, os SoCs (do inglês, *System on a Chip*) contendo processadores ARM Cortex-M que necessitam de coprocessadores de criptografia normalmente utilizam arquiteturas com *datapath* reduzido para que haja um casamento de desempenho e de área entre o processador e o co-processador criptográfico. Veja, como exemplo, (STMICROELECTRONICS, 2015).

Ao mesmo tempo, há um interesse crescente em aplicações de IoT (do inglês, *Internet of Things*) cujo objetivo último é interconectar, por meio da Internet, os mais diversos tipos de dispositivos eletrônicos. Um dos requisitos centrais nesse tipo de aplicação é a necessidade de garantir a confidencialidade, a integridade e a autenticidade dos dados trafegados. Assim, os protocolos de criptografia, como o AES, são peças fundamentais para tornar o IoT possível.

As redes IoT serão caracterizadas pela heterogeneidade dos dispositivos conectados, tanto em relação à sua funcionalidade final, quanto a seu poder computacional. Como consequência direta, haverá a necessidade de arquiteturas de *hardware* que suportem os protocolos de criptografia utilizados e que se adequem aos mais variados requisitos computacionais.

Assim, a escassez de trabalhos de pesquisa que explorem arquiteturas situadas no meio do espaço de projeto aliada a necessidade potencial desse tipo de arquitetura em aplicações de IoT constituem a motivação do presente trabalho.'

1.2 Objetivos

O objetivo dessa dissertação é duplo. Primeiro, propõe-se uma arquitetura de *hardware* para um co-processador capaz de realizar cifração e decifração segundo o padrão AES-128 com suporte aos modos de operação ECB (do inglês, *Electronic Codebook*), CBC (do inglês, *Cipher Block Chaining*) e CTR (do inglês, *Counter Mode*). A arquitetura proposta emprega as técnica de *loop rolling* com compartilhamento de recursos (para reduzir a quantidade de lógica necessária) e *subpipeling* (para aumentar a frequência de operação do circuito). A largura do *datapath* é 32 bits e o número de estágios do *pipeline* é três.

Segundo, documenta-se os resultados do projeto *openAES*. O *openAES* é um projeto *open source* desenvolvido a partir deste trabalho e que disponibiliza um *IP Core* de um co-processador AES compatível com o protocolo AMBA (do inglês, *Advanced Microcontroller Bus Architecture*) APB (do inglês, *Advanced Peripheral Bus*). O *IP Core* do projeto *OpenAES* faz uso da arquitetura proposta na primeira parte deste trabalho, adicionando a ela diversas funcionalidades, como suporte a DMA, geração de interrupções e possibilidade de suspensão de mensagens.

Através do repositório do projeto, acessível em (OPENAES. . . ,), são disponibilizados:

- O RTL, em Verilog, do IP Core;
- Um ambiente de verificação funcional;
- Uma camada de abstração de hardware (HAL), escrita em C, compatível com o padrão ARM CMSIS;
- Um *script* de *timing constraints* no formato SDC

Como forma de validação, o IP foi prototipado em um dispositivo SmartFusion A2F200M3F (MICROSEMI, 2015) e todos os arquivos necessários para embarcá-lo nessa plataforma também são disponibilizados.

1.3 Organização da dissertação

O restante dessa dissertação está organizada como segue:

- Capítulo 2: Neste Capítulo, discute-se as principais opções de arquitetura e micro-arquitetura disponíveis para se explorar o espaço de projeto do algoritmo AES.
- Capítulo 3: Neste Capítulo, é discutido o projeto de uma arquitetura de *hardware* para um co-processador capaz de realizar cifração e decifração segundo o padrão AES-128 com suporte aos modos de operação ECB, CBC e CTR. A arquitetura proposta emprega as técnica de *loop rolling* (com compartilhamento de SBox) e *subpipelining*. A largura do *datapath* é 32 bits e o número de estágios do *pipeline* é 3.
- Capítulo 4: Neste Capítulo, documenta-se os resultados do projeto *openAES*. As principais funcionalidades do *IP Core* são descritas, bem como os passos necessários para transformar essa descrição funcional em circuito digital.
- Considerações Finais: Neste capítulo, faz-se um apanhado geral das principais idéias e contribuições dadas por este trabalho.
- Apêndice A: Neste apêndice, indica-se quais sinais de controle são acionados em cada estado da máquina de estados finitos associada à unidade de controle do co-processador AES-128.
- Apêndice B: Neste apêndice, detalha-se os registros que compõe o mapa de memória do co-processador do projeto OpenAES.

2 Arquiteturas de *Hardware* para o Algoritmo AES

O mapeamento entre uma descrição algorítmica e sua implementação em *hardware* não é único. De fato, há diversas formas igualmente corretas de se implementar dado algoritmo e a decisão de qual delas é apropriada depende fortemente dos requisitos da aplicação final.

Em projeto digital, há basicamente três parâmetros a considerar ao se definir qual arquitetura de *hardware* é mais adequada, a saber: frequência de operação, área ocupada e potência dissipada. Este capítulo revisa, com foco no algoritmo AES, as principais técnicas de arquitetura e micro-arquitetura disponíveis para se explorar os compromissos entre esses parâmetros.

Antes de discutir essas técnicas, no entanto, será feita uma breve descrição do algoritmo AES com o intuito de contextualizar a discussão que segue. Para um tratamento detalhado desse algoritmo bem como das estruturas algébricas nas quais ele se baseia, indicam-se as referências (PAAR; PELZL, 2009; FIPS..., 2001; DAEMEN; RIJMEN, 2002).

2.1 O Algoritmo AES

O algoritmo AES é um cifrador simétrico que opera em blocos de dados de 128 bits e suporta chaves de comprimento 128, 196 e 256 bits. A Figura 1 esquematiza a relação entre a entrada e a saída do algoritmo, enfatizando a largura, em bits, dos dados processados.

Quando operando em modo de cifração, a entrada do algoritmo é comumente denominada *plaintext* e a saída *ciphertext*. Em modo de decifração, os papéis da entrada e saída são invertidos. Internamente, os blocos de 128 bits de dados são organizados em uma matriz 4x4, denominada *State*, onde cada entrada da matriz corresponde a um byte, que é tratado pelo algoritmo como um elemento do corpo de *Galois* de ordem 2^8 ($GF(2^8)$) (FIPS..., 2001).

O AES é um algoritmo iterativo onde o número de iterações *Nr*, denominadas *rounds*, é determinado pelo tamanho da chave de acordo com a Tabela 1. Cada *round*, com exceção do último, é composto por quatro operações, denominadas *layers*, e utiliza uma versão modificada, denominada *round key*, da chave de cifração/decifração. A operação de modificação das chaves durantes os rounds é denominada *Key Expansion*.

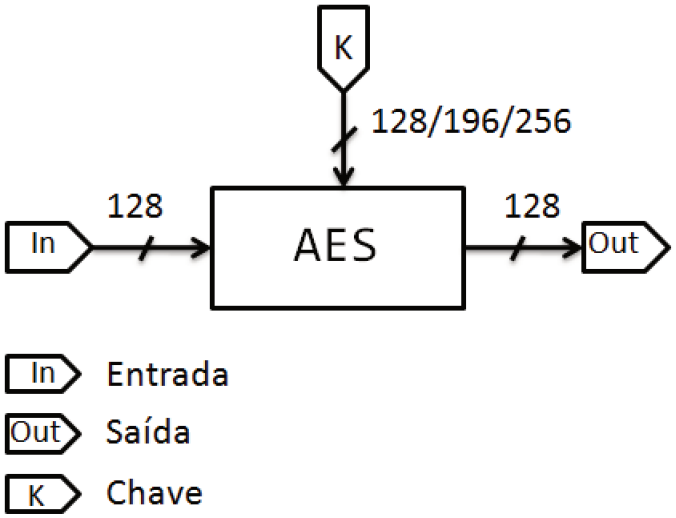


Figura 1 – Diagrama de Blocos do algoritmo AES

Tamanho da Chave	Número de rounds (Nr)
128	10
196	12
256	14

Tabela 1 – Relação entre o tamanho da chave e o número de rounds

O processo de cifração do AES se inicia com a operação *AddRoundKey()*, que realiza uma soma em GF(2^8) entre o *plaintext* e a primeira *round key*. O bloco de dados resultante é processado pelos Nr rounds do algoritmo. Os primeiros $Nr - 1$ *rounds* são compostos pelas operações: *SubBytes()*, *ShiftRows()*, *MixColumns()* e *AddRoundKey()*. O último *round* é semelhante aos anteriores, com a diferença que a operação *MixColumns()* não é realizada. Cada *round* utiliza uma *round key* diferente resultante do processo de *Key Expansion*. Esse processo de cifração do AES é ilustrado na Figura 2.

Para realizar a operação de decifração, todas as etapas da cifração devem ser invertidas, tanto em relação às operações em si quanto a sua ordem de execução. Assim, as *layers SubBytes()*, *ShiftRows()* e *MixColumns()* tornam-se, respectivamente, *InvShiftRows()*, *InvSubByte()* e *InvMixColumns()*. Tanto a cifração quanto a decifração utilizam a mesma *layer AddRoundKey()*. Isso se deve ao fato de que, em GF(2^8), adições e subtrações são realizadas pela mesma operação, e.g, XOR bit-a-bit. Outra diferença em relação à cifração é que as *round keys* devem ser utilizadas na ordem inversa. Assim, a primeira *round key* utilizada na decifração é a última *round key* utilizada na cifração. A Figura 3 ilustra essas diferenças.

Para as implementações que devem suportar tanto cifração quanto decifração, o algoritmo apresentado na Figura 3 é um tanto inadequado devido à diferença na ordem em que as *layers* são executadas. Uma implementação mais adequada para esses casos

é aquela apresentada na Figura 4. Como nem todas as operações comutam, é necessário modificar a etapa de *Key Expansion* para garantir a equivalência entre essas duas versões da decifração AES. Para mais detalhes, consulte (FIPS. . . , 2001).

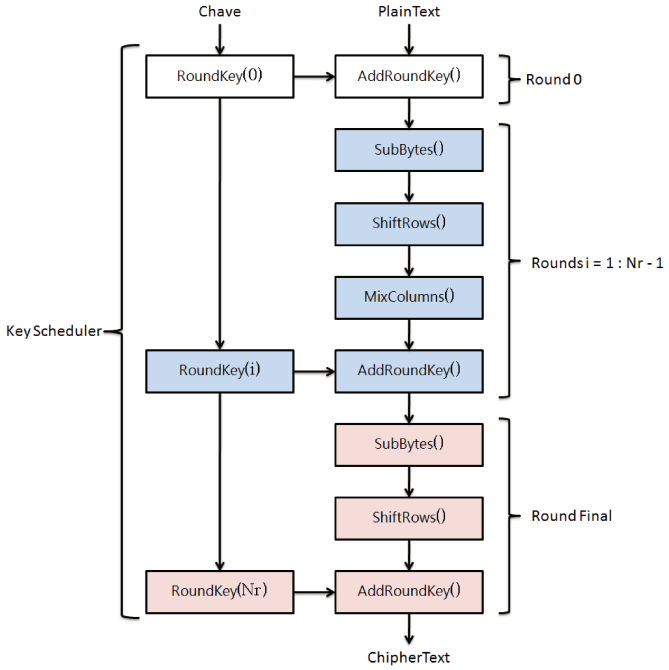


Figura 2 – Processo de Encriptação do algoritmo AES

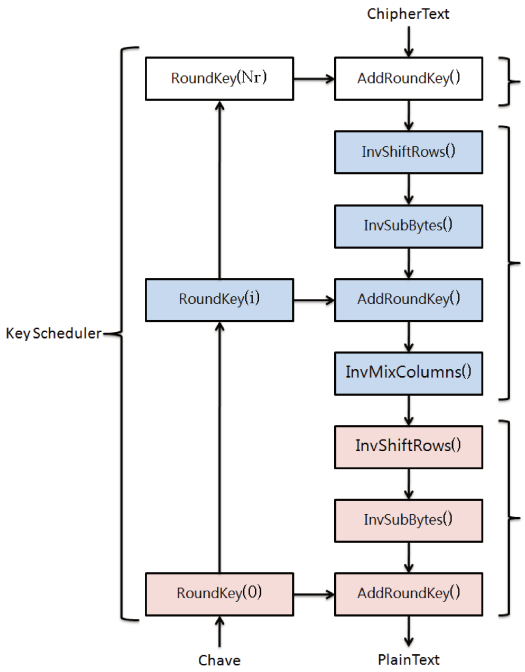


Figura 3 – Processo de Decriptação do algoritmo AES

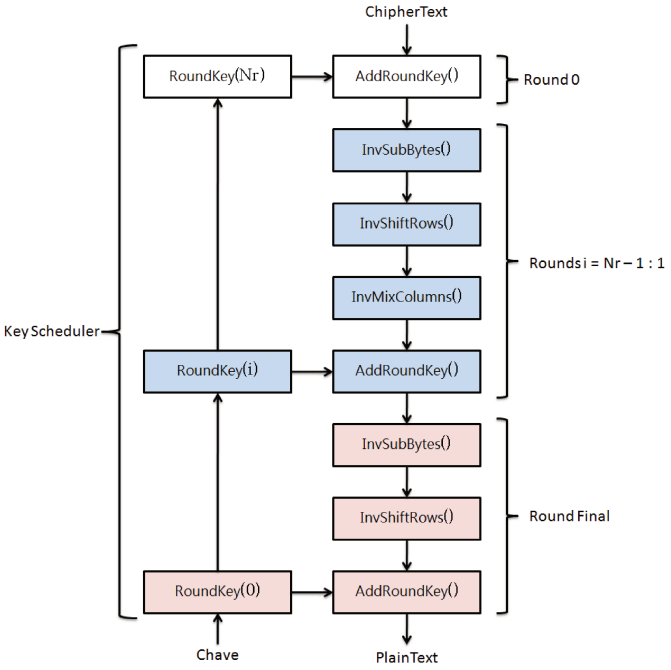


Figura 4 – Versão Alternativa do Processo de Decriptação do algoritmo AES

2.2 Arquiteturas de Hardware

O mapeamento entre uma descrição algorítmica e sua implementação em *hardware* não é único. De fato, há diversas formas igualmente corretas de se implementar dado algoritmo e a decisão de qual delas é apropriada depende fortemente dos requisitos da aplicação final.

Em projeto digital, há três parâmetros importantes a considerar ao se definir qual arquitetura de hardware é mais adequada, são eles: frequência de operação, área ocupada e potência dissipada. Em servidores ATM, por exemplo, o parâmetro preponderante é *throughtput*, já em smart cards é mais importante que a área ocupada pelo *hardware* seja pequena (ZHANG; PARHI, 2002).

Ocorre que esses parâmetros de projeto normalmente são conflitantes, de modo que um aumento em *throughtput* implica também em um aumento de área e de potência dissipada. Da mesmo forma, em geral, uma redução em área é acompanhada por uma diminuição da taxa máxima de processamento que se consegue atingir. Com isso, o espaço de projeto deve ser cuidadosamente explorado para que se consiga uma arquitetura que não só satisfaça os requisitos da aplicação final como o faça a um custo mínimo.

Com relação às implementações em *hardware* do algoritmo AES, há basicamente duas frentes que podem ser exploradas para se derivar uma arquitetura ótima.

Na primeira, as decisões tomadas são em nível de arquitetura, i.e, o espaço de projeto é explorado através de decisões sobre o grau de paralelismo e compartilhamento de recursos e sobre a largura do *datapath* dos rounds do algoritmo.

Na segunda, as decisões são tomadas em nível de micro-arquitetura, i.e, em nível de algoritmo. Nesse caso, o objetivo central é otimizar as principais operações do AES, como *SubBytes()* e *MixColumns()*, segundo algum parâmetro de projeto.

2.3 Técnicas para Otimização de Arquitetura

Com relação à estrutura dos *rounds*, há basicamente duas classes de arquiteturas: as arquiteturas iterativas e as arquiteturas com *loop unrolling*.

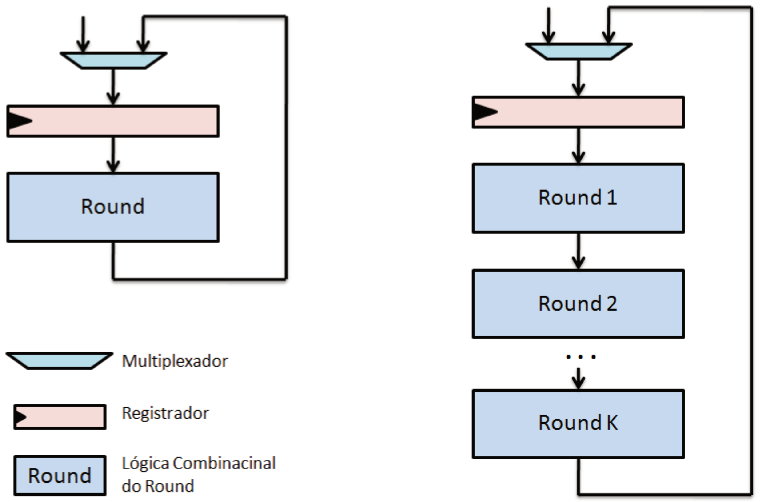
As arquiteturas iterativas utilizam apenas uma unidade de lógica combinacional para implementar todos os N_r *rounds* do algoritmo, assim a sua utilização de área é mínima. Por outro lado, são necessários N_r ciclos de *clock* para completar o algoritmo e a frequência de operação do circuito, que é dada pelo caminho combinacional mais longo, é determinada pelo atraso de toda a lógica combinacional do *round*. Portanto, o *throughput* desse tipo de arquitetura é baixo. Uma representação dessa arquitetura é dada na Figura 5a.

Com o uso de *loop unrolling* é possível reduzir o número de ciclos de *clock* necessários para completar o algoritmo por um fator de K, onde K é o número de *rounds* dispostos em série no *loop* principal, veja Figura 5b (K deve ser um múltiplo de Nr). Por outro lado, a área do circuito aumenta por um fator K e o seu caminho crítico é dado pela cascata de K *rounds*, o que implica em uma redução na frequência de operação do circuito por aproximadamente esse mesmo fator. Como consequência, o *throughput* dessa arquitetura é apenas sensivelmente maior que aquele das arquiteturas iterativas, embora sua área seja K vezes maior. O ganho em *throughput* deve-se à ausência dos tempos de *setup* e de propagação do registro (normalmente, mas não necessariamente, um banco de *flip-flops* do tipo D) durante $K-1$ ciclos de *clock*.

Em ambas as arquiteturas discutidas, o caminho crítico do circuito é longo, o que afeta diretamente a sua frequência máxima de operação e, portanto, o seu *throughput*. Esse caminho crítico pode ser reduzido empregando-se a técnica de *pipelining*.

Com o uso de M estágios de *pipeline*, é possível, em princípio, aumentar a frequência de operação do circuito por um fator M. Na prática, no entanto, há um limite superior para o ganho de desempenho que se consegue com o uso dessa técnica. Os principais limitantes são:

- ***Overhead* de sincronismo**: mesmo que não houvesse lógica alguma entre os estágios de *pipeline*, a frequência máxima de operação seria limitada pelo soma dos tempos de *setup* e de propagação do elemento sequencial utilizado (a rigor, ainda há o atraso devido à interconexão entre esses elementos).
- **Possível presença de elementos irredutíveis**: O princípio básico do *pipelining* é



(a) Arquitetura Iterativa

(b) Arquitetura com *Loop Unrolling*

Figura 5 – Opções de Arquitetura

dividir a lógica combinacional em porções menores e menos lentas e inserir elementos sequenciais entre essas porções de lógica. A frequência de operação do circuito passa a ser definida, então, pela porção de lógica com maior atraso. Caso haja algum elemento não redutível, i.e, que não pode ser dividido em porções menores, este elemento definirá o caminho crítico do circuito e a frequência de operação será dada por seu atraso, independente de possíveis acréscimos de registros de *pipeline* em outros pontos do circuito.

- **Desbalanceamento entre os estágios do *pipeline***: À medida que o número de estágios de *pipeline* aumenta, fica cada vez mais difícil balancear o atraso entre os estágios, i.e, fica difícil dividir a lógica em porções que possuem o mesmo atraso, o que resulta em uma redução no ganho teórico de desempenho.

- **Presença de *feedback***: Em circuitos que apresentam alguma forma de *feedback*, pode não ser possível manter o fluxo contínuo do *pipeline* devido à dependência entre os estágios. Nesses casos, pode ser necessário realizar um *stall* no *pipeline*, o que também reduz o ganho teórico de desempenho.

Como mencionado, é possível aumentar o *throughput* de ambas as arquiteturas da Figura 5 com o uso de *pipelining*. As combinações possíveis são esquematizadas na Figura 6.

Nas arquiteturas iterativas, os registros de *pipeline* são inseridos dentro do *round*, ou seja, a lógica combinacional do *round* é dividida em M estágios e elementos sequenciais são introduzidos entre cada um desses estágios. Com isso, no caso ideal, o caminho crítico do circuito tem seu atraso reduzido por um fator M, o que implica em um aumento na

frequência de operação do circuito por esse mesmo fator. É justamente do aumento da frequência de operação que vem o aumento de *throughput* nesse tipo de arquitetura. Note, no entanto, que o aumento de *throughput* só é significativo caso os estágios de pipeline sejam mantidos operantes, de modo que o uso agressivo de *pipelining* em arquiteturas com elevado grau de compartilhamento de recursos é desaconselhado.

Nas arquiteturas com *loop unrolling* há uma maior flexibilidade sobre onde se colocar os registros de *pipeline*. Quando a inserção é feita apenas entre os *rounds*, diz-se que a arquitetura emprega um *outer-round pipelining* (Figura 6b). Quando a inserção é apenas dentro do *round*, diz-se que a arquitetura emprega um *inner-round pipelining* ou *sub-pipelining* (Figura 6c). Também é possível inserir registros de *pipeline* tanto entre quanto dentro dos *rounds*, nesse caso diz-se que a arquitetura emprega um *full-round pipelining* (Figura 6d).

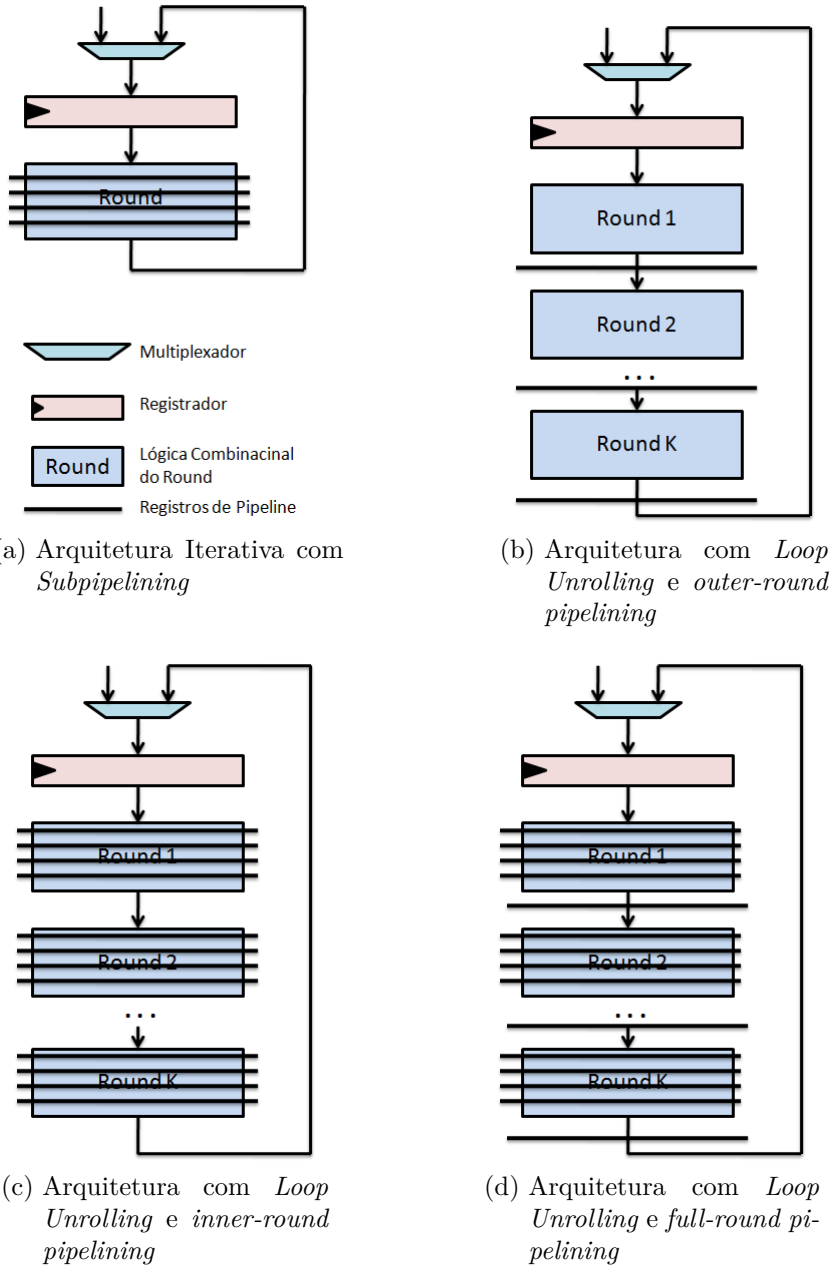
Em suma, o *throughput* médio de uma arquitetura de *hardware* que implementa o algoritmo AES é dado pela Equação 2.1, onde N_b é o número de bits processados, N_c é o número total de ciclos de *clock* necessários para se completar o algoritmo e T_{op} é o período do sinal de *clock*.

$$Throughput = \frac{N_b}{N_c.T_{op}} \tag{2.1}$$

De acordo com a Equação 2.1, caso se queira aumentar o *throughput* do circuito, há essencialmente três parâmetros que podem ser alterados. O parâmetro N_b pode ser aumentado para além de 128 bits utilizando-se múltiplas instâncias paralelas do algoritmo AES. O aumento de área nessa abordagem é proporcional ao número de fluxos paralelos de dados e sua utilização só é possível em modos que não fazem uso de *feedback*. A técnica básica para se alterar o parâmetro N_c é *loop unrolling* e o período de *clock*, T_{op}, pode ser reduzido com o uso de *pipelining*.

As técnicas discutidas até aqui permitem aumentar o *throughput* do circuito, mas isso só é possível com um aumento de área. Caso se queira reduzir a área da implementação final, outras estratégias devem ser utilizadas. A abordagem padrão, em nível de arquitetura, para reduzir área é compartilhar recursos.

Do ponto de vista de redução de área, duas decisões de arquitetura são críticas para garantir uma implementação compacta. Primeiro, deve-se decidir o número de *rounds* que serão reutilizados ou compartilhados. Nas arquiteturas iterativas, por exemplo, implementa-se apenas um *round*, que é reutilizado em todas as rodadas do algoritmo. Obviamente, o grau de compartilhamento e o *throughput* da implementação são parâmetros conflitantes, de modo que uma economia em área normalmente está associada a uma perda de desempenho. Segundo, deve-se definir a largura do *datapath* do *round*. Como o algoritmo AES opera em blocos de dados de 128 bits, a largura natural, por assim dizer,



(a) Arquitetura Iterativa com *Subpipelining*

(b) Arquitetura com *Loop Unrolling* e *outer-round pipelining*

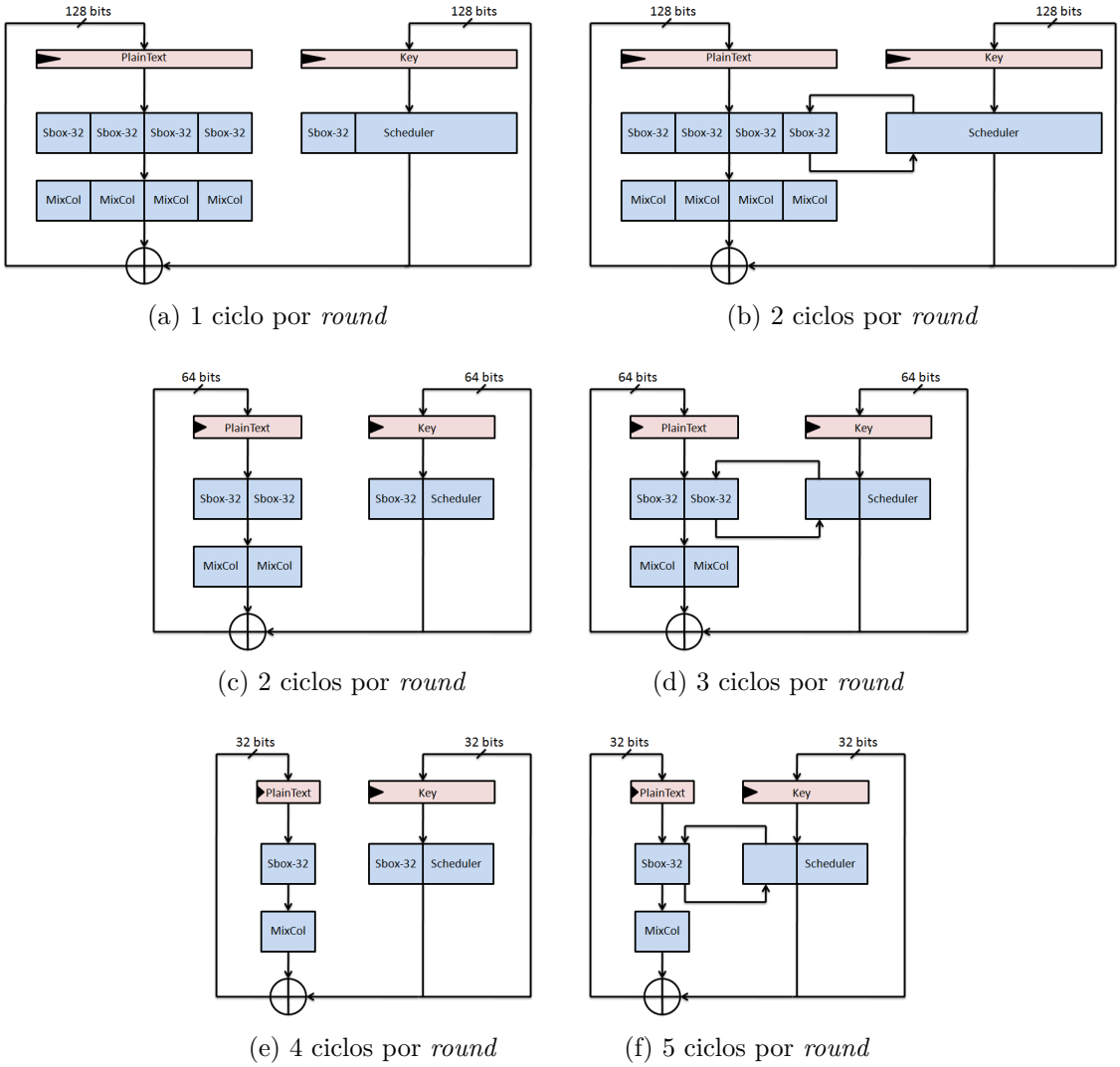
(c) Arquitetura com *Loop Unrolling* e *inner-round pipelining*

(d) Arquitetura com *Loop Unrolling* e *full-round pipelining*

Figura 6 – Opções de Arquitetura com *Pipelining*

do *datapath* é 128 bits. No entanto, para reduzir área, é possível diminuir essa largura para valores tão pequenos quanto 8 bits (FELDHOFER; WOLKERSTORFER, 2007). A *layer SubBytes()* (ou *Sbox*), por exemplo, opera em dados de 8 bits e sua implementação é responsável por grande parte da área total do *round* (SATOH *et al.*, 2001). Assim, ao invés de se utilizar 16 unidades de *Sbox*, pode-se reutilizar esse recurso para derivar um *datapath* de 8 bits que utiliza apenas uma unidade desse bloco. Por outro lado, serão necessários pelo menos 16 ciclos de *clock* para completar o processamento de um *round*. Esse compromisso entre a largura do *datapath* e o número de ciclos de *clock* necessários para completar o *round* é ilustrado na Figura 7 para outros tamanhos de *datapath*. Perceba que a etapa de *Key Expansion* também faz uso de uma *Sbox* de 32 bits, que pode

ou não ser compartilhada. Nos casos em que há compartilhamento de *Sbox*, um ciclo de *clock* adicional é necessário para completar a etapa de *Key Expansion*.



(a) 1 ciclo por *round*

(b) 2 ciclos por *round*

(c) 2 ciclos por *round*

(d) 3 ciclos por *round*

(e) 4 ciclos por *round*

(f) 5 ciclos por *round*

Figura 7 – Relação entre largura do *datapath* e número de ciclos por *round*

2.4 Técnicas para Otimização de Micro-Arquitetura

Cada *round* do algoritmo AES é composto pelas transformações *AddRoundKey()*, *SubBytes()*, *ShiftRows()* e *MixColumns()* no caso de cifração, e pelas respectivas operações inversas no caso de decifração. As técnicas de micro-arquitetura exploram a estrutura algébrica de cada uma dessas transformações com o intuito de otimizá-las segundo algum critério de projeto. Nesta seção serão discutidas algumas alternativas de implementação para cada uma dessas operações.

2.4.1 Alternativas de Implementação para a Operação *AddRoundKey()*

A operação *AddRoundKey()* realiza uma soma em GF(2^8) entre um bloco de dados e uma *round key*. Em GF(2^8), as somas são equivalentes a uma operação XOR bit-a-bit. Assim, a área dessa transformação é proporcional à largura do *datapath* utilizado e o seu atraso é dado pelo tempo de propagação da porta XOR empregada.

Não é possível otimizar esse bloco em termos de micro-arquitetura. De fato, a única forma de otimização é em nível de circuito ou tecnologia. Isto é, só é possível melhorar o desempenho da operação *AddRoundKey()* através do uso de portas XOR mais rápidas, quando o critério é aumentar a frequência de operação, ou menores, quando o critério é redução de área.

A operação *InvAddRoundKey()*, utilizada na decifração, realiza uma subtração em GF(2^8). Nesse estrutura algébrica, no entanto, somas e subtrações são indistinguíveis, de modo que as transformações *AddRoundKey()* e *InvAddRoundKey()* são equivalentes.

2.4.2 Alternativas de Implementação para a Operação *ShiftRows()*

A transformação *ShiftRows()* ciclicamente desloca as linhas da matriz de estado de acordo com a Equação 2.2, onde $S_{l,c}$ representa o elemento da linha l e coluna c dessa matriz.

$$S'_{l,c} = S_{l,((c+l) \ mod \ 4)} \qquad para \ 0 < l < 4 \quad e \quad 0 < c < 4 \tag{2.2}$$

Uma representação gráfica dessa operação é dada na Figura 8, onde se vê que a primeira linha da matriz de estado não sofre alteração, enquanto as demais linhas são deslocadas para esquerda por 1, 2 e 3 posições respectivamente.

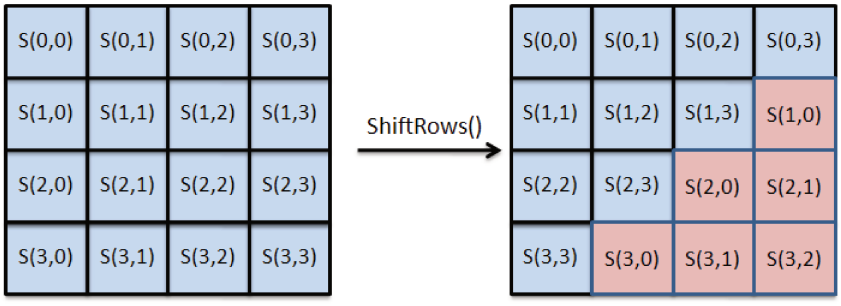


Figura 8 – Operação *ShiftRows*

Do ponto de vista de implementação em *hardware*, essa operação corresponde apenas a um reordenamento de dados e pode ser implementada sem utilizar recurso lógico algum. Portanto, assim como no caso anterior, não há espaço para otimização dessa operação. Ainda assim, a posição em que se decide colocar esse bloco no *datapath* tem influência na arquitetura do *round*.

A operação inversa, *InvShiftRows()*, é dada pela Equação 2.3. Como visto na Figura 9, essa operação mantém a primeira linha da matriz de estado inalterada, enquanto desloca as demais linhas por 1, 2 e 3 posições, respectivamente, para a direita. Essa operação também não requer recursos lógicos para ser implementada.

$$S'_{l,((c+l) \ mod \ 4)} = S_{l,c} \qquad para \ 0 < l < 4 \quad e \quad 0 < c < 4 \tag{2.3}$$

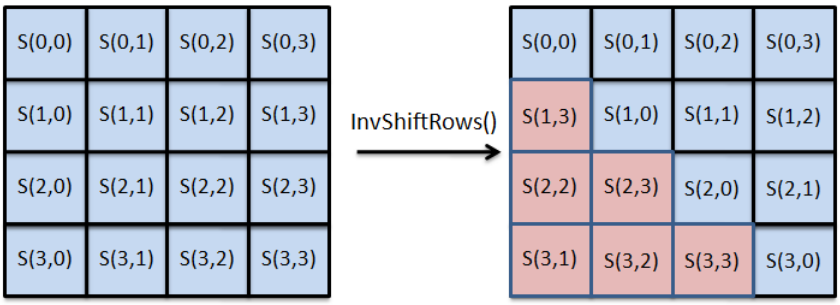


Figura 9 – Operação *InvShiftRows*

2.4.3 Alternativas de Implementação para a Operação *MixColumns()*

As *layers MixColumns()* e *InvMixColumns()* operam sobre as colunas da matriz de estado, encarando-as como polinômios de grau 3 com coeficientes em GF(2^8) em que as multiplicações são realizadas módulo $x^4 + 1$.

A operação *MixColumns()* multiplica cada palavra de 32 bits correspondente a uma coluna da matriz de estado por um polinômio fixo, $a(x)$, dado por:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{2.4}$$

Na Equação 2.4, a notação $\{a_i\}$ representa um polinômio de grau 7 cujos coeficientes são dados pelos bits do número hexadecimal a_i. Por exemplo, o coeficiente $\{03\}$, que é igual a $\{00000011\}$ em binário , corresponde ao polinômio $x + 1$.

Em notação matricial, a operação *MixColumns()* é dada pela Equação 2.5, onde c indexa as colunas da matriz de estado.

$$\begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix} \tag{2.5}$$

A operação inversa, *InvMixColumns()*, é obtida multiplicando-se cada coluna da matriz de estado pelo polinômio inverso, $a^{-1}(x)$, dado por:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{2.6}$$

Em notação matricial, essa operação é dada pela Equação 2.7, onde c indexa as colunas da matriz de estado.

$$\begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix} = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} \tag{2.7}$$

Como a operação *MixColumns()* só faz uso da multiplicação por duas constantes($\{02\}$ e $\{03\}$), não é necessário implementar um multiplicador em GF(2^8) genérico. Na verdade, um multiplicador por $\{02\}$, comumente denominado *xtime()*, é suficiente. De fato, o polinômio $\{03\}$ pode ser escrito como $\{02\} + \{01\}$ e, devido a distributividade da multiplicação em GF(2^8), a multiplicação pela constante $\{03\}$ pode ser escrita como um multiplicação pela constante $\{02\}$ seguida de uma soma.

A multiplicação por $\{02\}$, por sua vez, possui um custo, em termos de recursos lógicos, baixo, sendo necessárias apenas 3 portas XOR de duas entradas para implementá-la. De fato, o produto de um dado $X \in GF(2^8)$ por $\{02\}$ é dado por:

$$\{02\}X \equiv x.\left(\sum_{i=0}^{7} a_i x^i\right) \; mod \; (x^8 + x^4 + x^3 + x + 1), \; onde \; a_i \in GF(2) \tag{2.8}$$

$$\equiv \left(a_7 x^8 + \sum_{i=1}^{6} a_{i-1} x^i\right) \; mod \; (x^8 + x^4 + x^3 + x + 1) \tag{2.9}$$

$$\equiv \left(a_7(x^4 + x^3 + x + 1) + \sum_{i=1}^{6} a_{i-1} x^i\right) \; mod \; (x^8 + x^4 + x^3 + x + 1) \tag{2.10}$$

O polinômio $p(x) = x^8 + x^4 + x^3 + x + 1$ utilizado na Equação 2.10 é o polinômio irredutível definido pelo padrão AES. Em notação binária, o resultado da Equação 2.10 pode ser escrito como:

$$\{02\}X \equiv \left(a_7(x^4 + x^3 + x + 1) + \sum_{i=1}^{6} a_{i-1} x^i\right) \; mod \; (x^8 + x^4 + x^3 + x + 1) \tag{2.11}$$

$$= (0, 0, 0, a_7, a_7, 0, a_7, a_7) + (a_6, a_5, a_4, a_3, a_2, a_1, a_0, 0) \tag{2.12}$$

$$= (a_6, a_5, a_4, a_3 + a_7, a_2 + a_7, a_1, a_0 + a_7, a_7) \tag{2.13}$$

Todas as linhas da matriz da Equação 2.5 possuem os mesmos elementos. Como consequência, a operação *MixColumns()* pode ser implementada repetindo-se quatro vezes o mesmo bloco básico que implementa uma das linhas. A Figura 10 ilustra essa idéia para a primeira linha da matriz. Nessa Figura, também é indicado o caminho crítico do circuito, que é composto por 4 níveis de portas XOR.

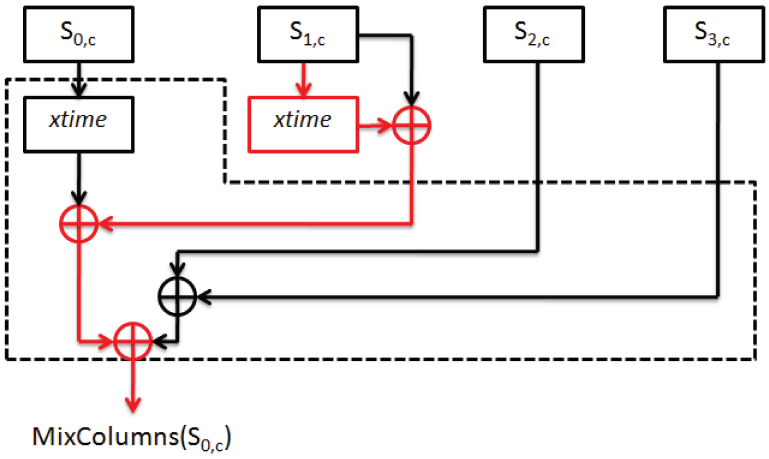


Figura 10 – Computação do primeiro byte da operação *MixColumns()*

O circuito da Figura 10 é uma implementação direta da equação que computa o elemento $S'_{0,c}$. Em (FISCHER *et al.*, 2005), o autor sugere um agrupamento de termos com o intuito de reduzir o número de recursos lógicos necessários para se implementar a operação *MixColumns()*. Utilizando essa idéia, o elemento $S'_{0,c}$ passa a ser computado como:

$$S'_{0,c} = \{02\}S_{0,c} + \{03\}S_{1,c} + \{01\}S_{2,c} + \{01\}S_{3,c} \tag{2.14}$$

$$= (S_{0,c} + S_{1,c} + S_{2,c} + S_{3,c}) + \{02\}(S_{0,c} + S_{1,c}) + S_{0,c} \tag{2.15}$$

Na Equação 2.15, o termo $S_{0,c} + S_{1,c} + S_{2,c} + S_{3,c}$ pode ser reutilizado por todos os quatro bytes da operação *MixColumns()*.

As mesmas considerações feitas até aqui se aplicam à transformação *InvMixColumns()*. Nessa operação, no entanto, os coeficientes são mais complexos, o que demanda uma maior área e resulta em um maior atraso quando comparada à operação *MixColumns()*.

A maior oportunidade de otimização das operações *MixColumns()* e *InvMixColumns()* está nas arquiteturas que devem suportar tanto cifração quanto decifração. Nesses casos, é possível derivar um único *hardware* que suporte ambas as operações. Nesse sentido, há essencialmente duas abordagens reportadas na literatura, que são: o compartilhamento de *xtimes* (ZHANG; PARHI, 2004; LU; TSENG, 2002) e a decomposição da matriz *InvMixColumns()* (FISCHER *et al.*, 2005; SATOH *et al.*, 2001; RODRIGUEZ-HENRIQUEZ *et al.*, 2003).

A idéia básica da técnica de decomposição da matriz *InvMixColumns()* é escrevê-la em função da matriz *MixColumns()*. Com isso, a implementação da operação direta e inversa pode ser feita com o mesmo circuito.

Há dois tipos de decomposição reportadas na literatura (GAJ; CHODOWIEC, 2009). Na decomposição paralela, a operação inversa é escrita como:

$$\begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix} = \Big([C] + [E] \Big) \begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} \tag{2.16}$$

onde $[C]$ é a matriz direta dada pela Equação 2.5 e $[E]$ é matriz dada por:

$$[E] = \begin{pmatrix} 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \\ 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \end{pmatrix} \begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} \tag{2.17}$$

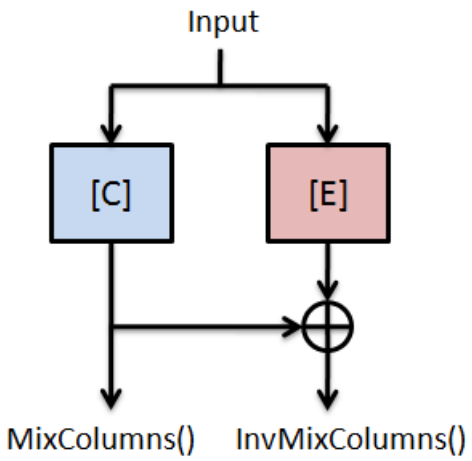


Figura 11 – Decomposição da operação *InvMixColumns()*

Conforme ilustrado na Figura 11, com essa abordagem é possível utilizar um mesmo *hardware* para implementar ambas as direções da operação *MixColumns()*. É importante destacar que o caminho crítico do circuito é dado pela operação inversa devido à maior complexidade dos coeficientes ($\{0c\}$ e $\{08\}$) e à presença de uma camada de portas XOR.

Na decomposição serial, a matriz inversa é escrita como o produto de duas matrizes:

$$\begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix} = \Big([C].[F] \Big) \begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} \tag{2.18}$$

onde $[C]$ é a matriz direta dada pela Equação 2.5 e $[F]$ é matriz dada por:

$$[F] = \begin{pmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{pmatrix} \begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix} \tag{2.19}$$

A matriz [F], quando comparada à matriz [E], possui coeficientes mais simples e é mais esparsa. Como consequência, sua implementação emprega menos recursos lógicos. Por outro lado, o caminho crítico do circuito, nesse caso, é dado pela cascata das lógicas que implementam as matrizes [C] e [F].

Para uma derivação detalhada desses resultados, indica-se (GAJ; CHODOWIEC, 2009) e (FISCHER *et al.*, 2005).

2.4.4 Alternativas de Implementação para a Operação *SubBytes()*

A transformação *SubBytes()* opera sobre cada coluna da matriz de estado, transformando-as segundo uma tabela de substituição denominada *S-Box*. A *S-Box*, por sua vez, opera em *bytes*, de modo que a transformação *SubBytes()* é composta por 16 tabelas *S-Box* operando em paralelo. Do ponto de vista algébrico, essa transformação consiste em uma inversão no campo GF(2^8) seguida de uma transformação afim dada pela Equação 2.20, onde a e b são elementos de GF(2^8) e representam os *bytes* de entrada e saída da *S-Box*, respectivamente:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \tag{2.20}$$

A operação inversa, *InvSubBytes()*, consiste na aplicação da transformação afim inversa àquela da Equação 2.20 seguida de uma inversão multiplicativa em GF(2^8).

Há basicamente três estratégias para implementação da operação *S-Box* e sua inversa. Uma opção é implementar essa operação utilizando apenas *look-up tables*. Nesse caso, são necessários 256×8 bits de memória para implementar uma *S-Box*. Como a *layer SubBytes()* é composta por 16 unidades paralelas de *S-Box*, são necessários $16 \times 256 \times 8$ bits ou $32kbits$ de memória para sua implementação. Caso seja necessário implementar a operação inversa, o dobro de memória deve ser utilizado.

Outra opção é implementar a inversão em GF(2^8) por meio de *look-up tables* e a transformação afim por meio de lógica. Com essa abordagem, é possível reduzir pela metade a quantidade de memória requerida pelas implementações que devem suportar tanto cifração quanto decifração. Isso é possível pois tanto a *S-Box* direta quanto a inversa podem compartilhar a mesma tabela para implementar a inversão em GF(2^8).

Por fim, a *S-Box* pode ser implementada utilizando-se apenas lógica. Uma alternativa é descrever a operação de *S-Box* como uma *look-up table* e delegar a tarefa de simplificação lógica para um sintetizador de circuito. Essa abordagem, no entanto, é pouco efetiva, pois o sintetizador é incapaz de fazer uso da estrutura algébrica da *S-Box*. Uma implementação mais eficiente é possível com o uso de aritmética de sub-campo, como inicialmente proposto por (RIJMEN,) e posteriormente estendido pelos trabalhos (SATOH *et al.*, 2001; CANRIGHT, 2005; ZHANG; PARHI, 2004). A idéia básica dessa técnica consiste em mapear, através de um isomorfismo, os elementos do campo original em um campo composto onde a computação do inverso multiplicativo é menos custosa. Em seguida, o resultado obtido é remapeado, via um isomorfismo inverso, para o campo original. Para mais detalhes sobre essa técnica, sugere-se (SATOH *et al.*, 2001; CANRIGHT, 2005; ZHANG; PARHI, 2004; GAJ; CHODOWIEC, 2009).

A escolha de qual dessas abordagens deve ser utilizada depende fortemente dos requisitos da aplicação e da plataforma final de implementação. Para os casos em que a plataforma final é uma FPGA, por exemplo, uma implementação baseada em *look-up tables* pode ser a melhor escolha, visto que esse tipo de dispositivo já dispõe de blocos de memória embarcados em seu interior, quer eles sejam utilizados ou não. Em ASIC (do inglês, *Application-Specific Integrated Circuit*), por outro lado, há uma maior liberdade sobre a utilização ou não de blocos de memória, de modo que a escolha é fortemente dependente dos requisitos da aplicação. Por exemplo, para os casos em que se deseja um alto *throughput*, a implementação via *look-up tables* pode ser inviável, pois o acesso à memória é irredutível, o que impõe limites para a máxima frequência de operação que se consegue atingir. Nesses casos, uma implementação baseada inteiramente em lógica pode ser mais adequada, uma vez que é possível reduzir o caminho crítico do circuito através da inserção de estágios de *pipeline*.

As técnicas de otimização de arquitetura e microarquitetura discutidas neste capítulo são, de certa forma, ortogonais. A combinação destas técnicas, portanto, gera um grande número de arquiteturas de *hardware* possíveis. Qual dessas implementações é a mais adequada depende em grande parte dos requisitos da aplicação final.

3 Arquitetura Proposta

Neste capítulo, estuda-se as peculiaridades das arquiteturas iterativas com *subpipelining* e *datapath* de 32 bits. Com esse estudo, procura-se, essencialmente, responder às seguintes questões:

- Como o tempo de processamento da arquitetura se comporta quando o número de estágios de *pipeline* aumenta?
- É possível otimizar a etapa de derivação de chave para esse tipo de arquitetura?
- Como as peculiaridades da arquitetura em estudo afetam as decisões de micro-arquitetura?

Ainda neste capítulo, propõe-se uma arquitetura de *hardware* para um co-processador capaz de realizar cifração e decifração segundo o padrão AES-128. Inicialmente, é apresentada uma arquitetura que suporta o modo de cifração sem *feedback* ECB. Em seguida, discute-se como essa arquitetura básica pode ser estendida para suportar os modos de cifração com *feedback* CBC e CTR. A arquitetura proposta emprega as técnicas de *loop rolling* (com compartilhamento de SBox) e *subpipelining*. A largura do *datapath* é 32 bits e o número de estágios do *pipeline* é 3.

3.1 Princípio de Projeto

Ao implementar um dado algoritmo em *hardware*, a corretude funcional é uma condição necessária, mas não suficiente. De fato, um circuito digital, além do requisito funcional, normalmente está sujeito a restrições de implementação como frequência de operação, quantidade de recursos lógicos utilizados e potência consumida. Qual desses fatores é mais importante depende em grande parte da aplicação para a qual o *hardware* se destina. Dessa forma, durante o mapeamento entre uma descrição algorítmica e sua implementação em circuito, o espaço de projeto deve ser cuidadosamente explorado com o intuito de se atingir uma arquitetura que seja eficiente em termos dos parâmetros de projeto preponderantes. Assim, ao se comparar duas implementações do algoritmo AES que operam, respectivamente, a taxas de 1 Gbps e 10 Mbps, por exemplo, não é correto afirmar que a primeira implementação é a mais eficiente sem que se saiba qual parâmetro de projeto é o mais importante.

Na literatura dedicada às implementações em *hardware* do algoritmo AES, há uma enorme quantidade de trabalhos dedicados às arquiteturas cujo parâmetro de projeto mais

importante é *throughput* (ZHANG; PARHI, 2004; MORIOKA; SATOH, 2002; JäRVINEN *et al.*, 2003; MAYER *et al.*, 2002; VERBAUWHEDE *et al.*, 2003; MOZAFFARI-KERMANI; REYHANI-MASOLEH, 2012; LUTZ *et al.*, 2002; IYER *et al.*, 2006; FAN; HWANG, 2007; KOTTURI *et al.*, 2005). Há uma porção igualmente volumosa de trabalhos que otimizam suas arquiteturas para cenários em que a quantidade de recursos lógicos utilizados é o parâmetro principal (FELDHOFER *et al.*, 2005; HSIAO *et al.*, 2006; HAMALAINEN *et al.*, 2006; GAJ, 2003; SATOH *et al.*, 2001; NEDJAH *et al.*, 2006). O meio do espaço de projeto, no entanto, tem sido pouco explorado na literatura, apesar da sua importância em aplicações práticas.

Neste trabalho, propõe-se uma arquitetura cujo princípio de projeto é um compromisso entre área e *throughput*. O objetivo da arquitetura proposta é atingir uma taxa de processamento moderada, utilizando, para isso, uma quantidade moderada de recursos lógicos. O termo "moderado", por si só bastante vago, é utilizado aqui para designar soluções iterativas cujo *datapath* possui 32 ou 64 bits. Nesse tipo de arquitetura, o *throughput* que se consegue é superior ao das arquiteturas compactas (consideradas de baixa taxa), mas é inferior ao das arquiteturas com *loop unrolling e full-round pipelining* (consideradas de alta taxa). Da mesma forma, a área ocupada pelas arquiteturas moderadas é superior a das arquiteturas compactas (consideradas de baixa área), mas inferior a das arquiteturas com *loop unrolling e full-round pipelining* (consideradas de alta área).

Quando o princípio de projeto é máximo *throughput* ou mínima área, os compromissos de arquitetura e micro-arquitetura são claros. No primeiro caso, opta-se por soluções que implicam em altas taxas de processamento independente da área ocupada. No segundo caso, adota-se o critério contrário, i.e, as decisões sempre são tomadas de forma a se obter a menor área ocupada possível. Para as arquiteturas ditas moderadas, a escolha é menos clara, pois deve-se atender a duas restrições normalmente conflitantes. A arquitetura apresentada neste trabalho se enquadra nessa categoria ,como consequência, em alguns pontos foi priorizada taxa e em outros área, como fica claro no restante do texto. Em todo caso, espera-se, qualquer que seja a micro-arquitetura escolhida, que o uso de *sub-pipelining* proporcione ganhos de desempenho significativos em relação às arquiteturas que não o empregam. Esses ganhos, no entanto, vêm acompanhados de um aumento na área final da implementação, devido, principalmente, à inserção de registros de *pipeline* e ao aumento da complexidade da unidade de controle.

3.2 Arquitetura do Co-processador AES-128

Com o intuito de reduzir a área final da implementação, optou-se por uma arquitetura iterativa com *datapath* de 32 bits. Para aumentar a frequência de operação, utilizou-se um *sub-pipelining* de três estágios. O co-processador AES proposto, portanto,

se enquadra na classe de arquiteturas representadas pela Figura 6a.

As razões para a escolha de uma arquitetura de 32 bits são, em parte, aquelas apresentadas por (MORIOKA; SATOH, 2002) e, em parte, a intenção de utilizá-la para implementar co-processadores AES que atuam como periféricos de processadores de 32 bits de baixo a médio poder computacional. A escolha da profundidade do *pipeline*, por sua vez, baseia-se no estudo, feito adiante, do ganho de desempenho que se consegue com o uso de *sub-pipelining* em arquiteturas de 32 bits.

O uso de uma arquitetura iterativa com *sub-pipelining* não é novo e já foi explorado em trabalhos como (WU *et al.*, 2010). Nesse trabalho, o autor também utiliza três estágios de *sub-pipelining*, mas com um *datapath* de 128 bits.

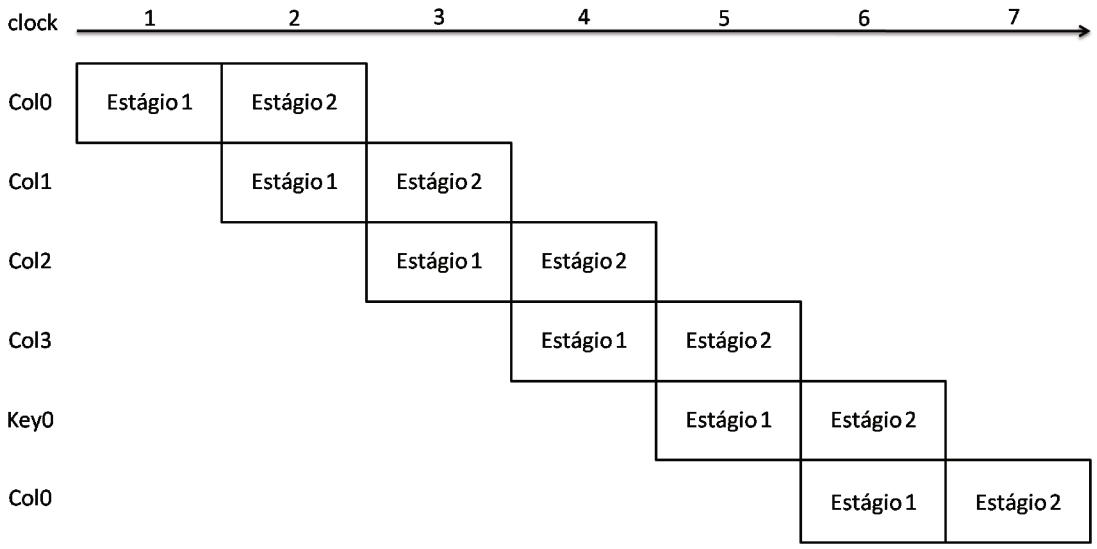
Diferente de (WU *et al.*, 2010), a arquitetura proposta no presente trabalho utiliza um *datapath* de 32 bits, o que impõe certas restrições ao *pipeline* não presentes naquele arquitetura.

De fato, ao se utilizar um *datapath* de 32 bits, são necessários quatro (ou cinco, caso a *SBox* seja compartilhada) rodadas de processamento para se concluir um *round* do algoritmo. Com isso, dependendo do número de estágios, não é possível manter o *pipeline* cheio, o que requer a inserção de ciclos de NOP (do inglês, *No Operation*).

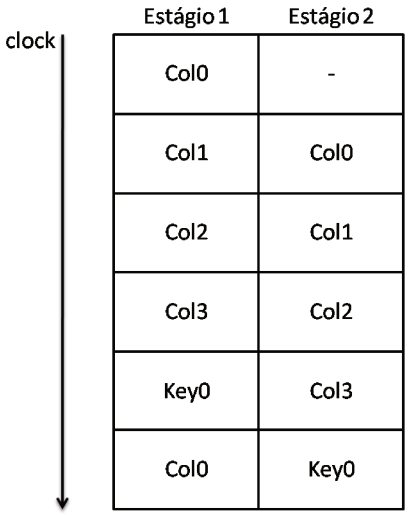
Para entender melhor essa questão, considere um *datapath* de 32 bits com *SBox* compartilhada. Sem o uso de *pipeline*, são necessários cinco ciclos de *clock* para concluir o processamento de um *round*. Desses cinco ciclos, quatro são destinados ao processamento da entrada e um é destinada ao cálculo dos 32 bits mais significativos da *round key*. Assim, no total, são necessários cinquenta e quatro ciclos para concluir o algoritmo (quatro ciclos para o *Round 0* e cinco ciclos por *round* para os outros dez *rounds*).

Ao se utilizar um *pipeline* de dois estágios, são necessários os mesmos cinquenta e quatro ciclos, mas agora a frequência de operação pode ser duplicada (considerando que se consiga dividir a lógica combinacional em duas porções de mesmo atraso e desconsiderando o *overhead* de sincronismo). A Figura 12 ilustra esse caso por meio de dois modos equivalentes de se representar o *pipeline*. A representação da Figura 12a é a mais usual, mas neste trabalho será utilizada a da Figura 12b.

Como visto na Figura 12, para o caso em que há dois estágios, é possível manter o *pipeline* sempre cheio. O mesmo não ocorre quando o número de estágios do *pipeline* é maior que ou igual a três. Nesses casos, há *hazard* de dados causado pela dependência entre os cálculos das colunas. A Figura 13 ilustra essa dependência para o caso em que o número de estágios é 3. Para que o *round* n se inicie, o *round* $n-1$ deve ter sido concluído, no entanto, devido à profundidade do *pipeline*, a computação de *Col0* e *Col3* ocorre no mesmo ciclo o que causa o *hazard*. A solução padrão para esse tipo de problema é utilizar *forwarding*, quando possível, ou inserir um NOP no *pipeline*, quando não for



(a) Representação do *Pipeline* com o eixo do tempo na horizontal



(b) Representação do *Pipeline* com o eixo do tempo na vertical

Figura 12 – Representações do *Pipeline*

possível realizar *forwarding*.

No caso do algoritmo AES, realizar um *forwarding* é inviável, pois implicaria na concatenação da lógica combinacional de dois ou mais estágios do *pipeline*, o que reduz severamente a frequência máxima de operação do circuito. Dessa forma, a melhor solução é inserir um ciclo de NOP, conforme ilustrado na Figura 14. Aagora, quando o processamento de *Col0* se inicia, o *round* anterior já se encerrou, de modo que o *hazard* é resolvido. Por outro lado, há um ciclo de *clock* que não realiza processamento algum, o que implica em uma diminuição no ganho teórico de desempenho do *pipeline*. No total, são necessários sessenta e quatro ciclos de *clock* para concluir o algoritmo AES quando se utiliza um *pipeline* de três estágios. A frequência de operação, no entanto, é triplicada, em teoria, em relação ao caso em que não há *pipeline*.

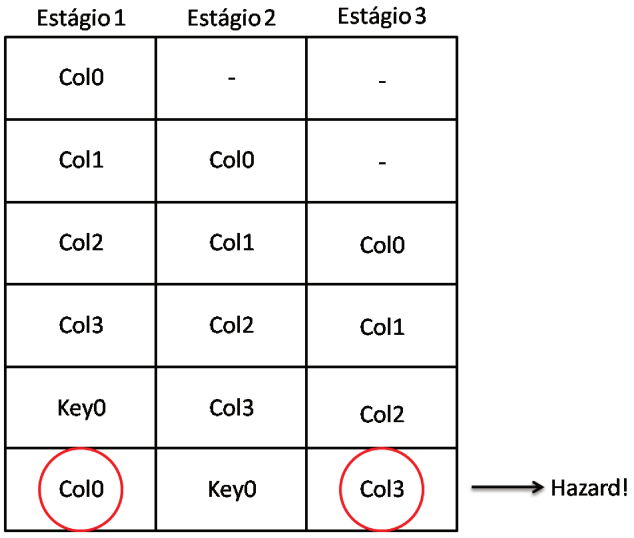


Figura 13 – Ocorrência de *Hazard* em um *Pipeline* de 3 estágios

Estágio 1	Estágio 2	Estágio 3
Col0	-	-
Col1	Col0	-
Col2	Col1	Col0
Col3	Col2	Col1
Key0	Col3	Col2
NOP	Key0	Col3
Col0	NOP	Key0

Figura 14 – Inserção de NOP em um *Pipeline* de 3 estágios

No caso geral, o tempo necessário para se concluir o algoritmo AES para a arquitetura em questão é dado pela Equação 3.1, onde n é o número de estágios do *pipeline* e T é o atraso total da lógica combinacional do *round*.

$$t = \frac{\Big(54 + 10 \times (n-2)\Big)}{n} \times T, \qquad para \quad n > 1 \tag{3.1}$$

De fato, são necessários 4 ciclos de *clock* para se concluir o *round* 0 e $5 + (n-2)$ ciclos para se concluir os demais *rounds* do algoritmo. O termo $n-2$ contabiliza o número de ciclos de NOP que se deve incluir no *pipelining*. Como cada ciclo requer $\frac{T}{n}$ segundos, o tempo total de processamento é aquele dado pela equação 3.1.

A Equação 3.1 assume que é possível dividir a lógica combinacional do *round* em

n porções de mesmo atraso e que quaisquer outros atrasos que não sejam devido à lógica do *round* podem ser desconsiderados. Embora essas hipóteses não se verificam na prática, especialmente para valores de n muito grandes, a Equação 3.1 fornece uma análise de melhor caso que é útil para guiar as escolhas de arquitetura durante as fases iniciais de um projeto.

Na Figura 15, mostra-se como o tempo de processamento da arquitetura varia com o número de estágios do *pipeline*. Nessa figura, o eixo vertical está normalizado em relação ao atraso T. Como se vê, em arquiteturas iterativas de 32 bits, o uso muito agressivo de *pipeline* não compensa, mesmo no caso ideal em que se consegue balancear a lógica entre os estágios. Vale lembrar que, para aumentar o número de estágios do *pipeline*, deve-se inserir registradores, tanto para os dados quanto para os sinais de controle, em número proporcional à largura do *datapath* e ao número de estágios. Esses registradores, por sua vez, aumentam a área e o consumo de potência da implementação final. Portanto, é mais eficiente aumentar o *throughput* por outros meios, como alargando o *datapath*, do que aumentando de forma excessiva o número de estágios do *pipelining*.

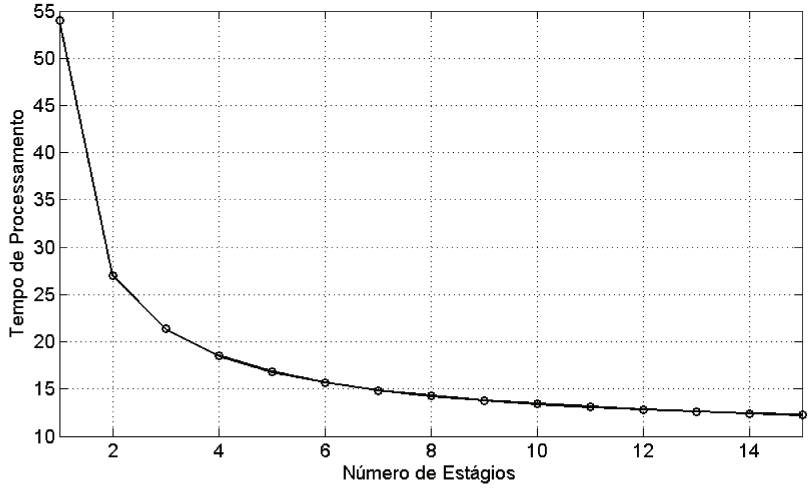


Figura 15 – Tempo de Processamento versus número de estágios do *pipeline*

Na Figura 16, mostra-se a perda de desempenho ocasionada pela inserção dos ciclos de NOP. Na prática, essa perda de desempenho é ainda mais acentuada, em grande parte pela dificuldade de se dividir a lógica do *round* em porções de mesmo atraso. Outras razões são dadas na seção 2.3.

A sensibilidade do *pipeline* ao desbalanceamento dos estágios é dado pela Equação 3.2, onde P representa a variação do período-ideal ($\frac{T}{n}$, onde T é o atraso de toda a lógica combinacional de um *round* do algoritmo) necessária para que o *pipeline* de n estágios seja mais lento que o *pipeline* de $n-1$ estágios, i.e, se o desbalanceamento do *pipeline* de n estágios for superior a $P \times \frac{T}{n}$, não compensa utilizar os n estágios, pois o *pipeline* de $n-1$ estágios é mais rápido. Por exemplo, para $n = 4$, o período ideal é $\frac{T}{4}$. Caso haja um desbalanceamento P maior que ou igual a $0.1532 \times \frac{T}{4}$ (ou 15.32%), é mais

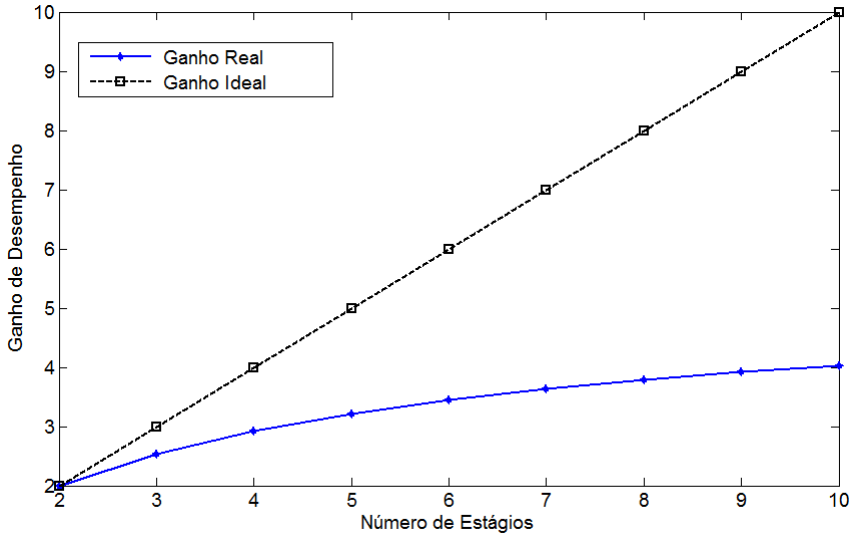


Figura 16 – Efeito da inserção de NOPs no ganho de desempenho proporcionado pelo *pipeline*

vantajoso utilizar um *pipeline* de 3 estágios. A Figura 17 apresenta esse resultado para outros valores de n.

$$P = \frac{34}{n \times (44 + 10 \times n)}, \qquad para \quad n > 2 \tag{3.2}$$

A Equação 3.2 é obtida igualando-se o tempo de processamento de um *pipeline* de n estágios ao tempo de processamento de um *pipeline* de $n+1$ estágios com desbalanceamento dado por $P \times \frac{T}{n+1}$:

$$(34 + 10n) \times \frac{T}{n} = (44 + 10n) \times \frac{T}{n+1} \times (1 + P)$$

Em resumo, o uso de *subpipelining* em arquiteturas iterativas de 32 bits só se justifica para um número pequeno de estágios. De fato, o ganho de desempenho do *pipeline* nesse tipo de arquitetura possui um limite superior devido à inserção de ciclos de NOP. Além disso, quanto maior o número de estágios, mais difícil é balancear os atrasos das lógicas combinacionais que compõe o *pipeline* e menor é a margem de desbalanceamento suportada pela arquitetura.

3.3 Micro-Arquitetura das *Layers* do Co-processador AES-128

Conforme discutido na seção 2.4, há diversas opções de implementação para as *layers* *MixColumns()*, *InvMixColumns()*, *SubBytes()* e *InvSubBytes()*. Da mesma forma, há várias maneiras de se implementar a operação de derivação de chave. A decisão sobre

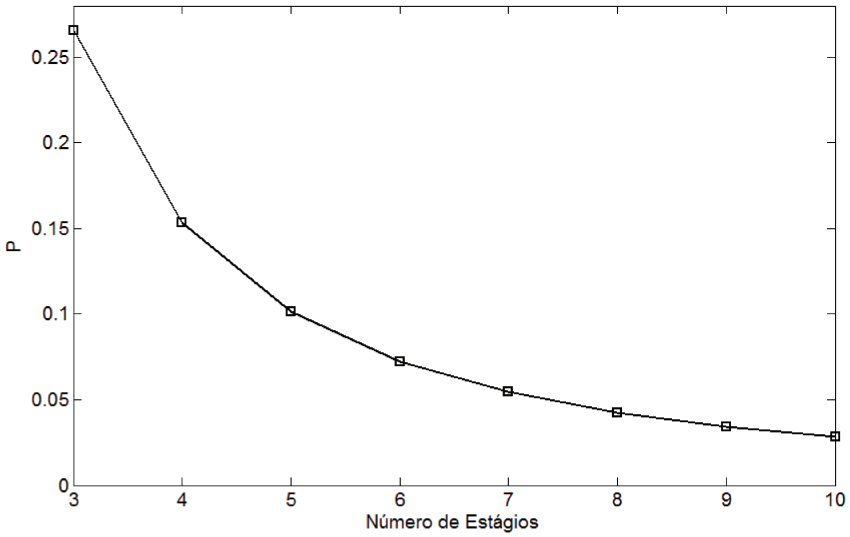


Figura 17 – Sensibilidade da arquitetura ao desbalanceamento dos estágios

qual dessas opções adotar depende dos requisitos do projeto, uma vez que, em geral, há um compromisso entre frequência de operação e área ocupada.

Nesta seção, apresenta-se as opções de micro-arquitetura escolhidas para compor o co-processador AES-128 proposto.

3.3.1 Operação MixColumns()/InvMixColumns()

Para implementar a operação *MixColumns()*, optou-se pela abordagem direta como apresentado na Figura 10. A otimização proposta em (FISCHER *et al.*, 2005) corresponde a um simples agrupamento e compartilhamento de termos, algo que as ferramentas de síntese de circuito lógico atuais fazem com eficiência. Dessa forma, o autor deste trabalho acredita que, em um fluxo de projeto baseado em *standard cells*, não haja diferenças significativas entre as duas abordagens.

A Figura 18 mostra a implementação da operação *MixColumns()* para uma entrada de 32 bits.

Para implementar a operação *InvMixColumns()*, optou-se pela decomposição paralela da matriz de inversão como descrito na seção 2.4.3. A micro-arquitetura resultante é ilustrada na Figura 19. O circuito que implementa a operação *InvMixColumns()* faz uso da saída da operação *MixColumns()*. Como a decomposição escolhida é do tipo paralela, o impacto do compartilhamento de recursos entre as operações direta e inversa no caminho crítico do circuito não é tão alto e se dá por meio de apenas uma camada de portas XOR, como ilustrado na Figura 19. O preço que se paga é um aumento da área ocupado em relação à implementação serial.

Como a arquitetura proposta faz uso de apenas três estágios de *pipeline*, toda a

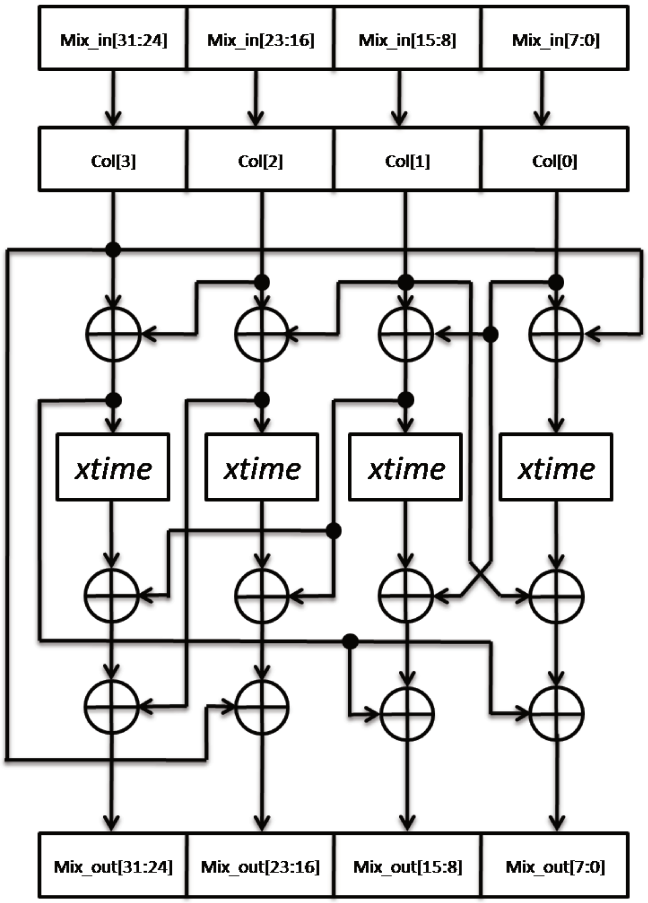


Figura 18 – Micro-arquitetura da operação MixColumns()

lógica combinacional associada à operação *MixColumns()* direta e inversa situa-se em um único estágio, o que torna importante otimizar a implementação dessa etapa para reduzir o seu atraso total, de modo que a escolha apresentada se justifica.

3.3.2 Operação SubBytes()/InvSubBytes()

A *Sbox* direta e inversa é o elemento que possui maior impacto na área e frequência da implementação final (SATOH *et al.*, 2001). Por isso, para a arquitetura em questão, optou-se por separar a *SBox* em dois estágios. A estratégia de implementação utilizada é aquela proposta em (CANRIGHT, 2005). Como visto na Figura 20, os registros de *pipeline* são inseridos de modo a dividir o cálculo do inverso multiplicativo em duas etapas. No total, são necessários 12 registradores para uma *SBox* de 8 bits e 48 registradores para uma *SBox* de 32 bits.

3.3.3 Operação de Derivação de Chave

Cada *round* do algoritmo AES utiliza uma chave diferente resultante do processo de derivação de chave. Para o caso em que o comprimento da chave é 128 bits, esse processo

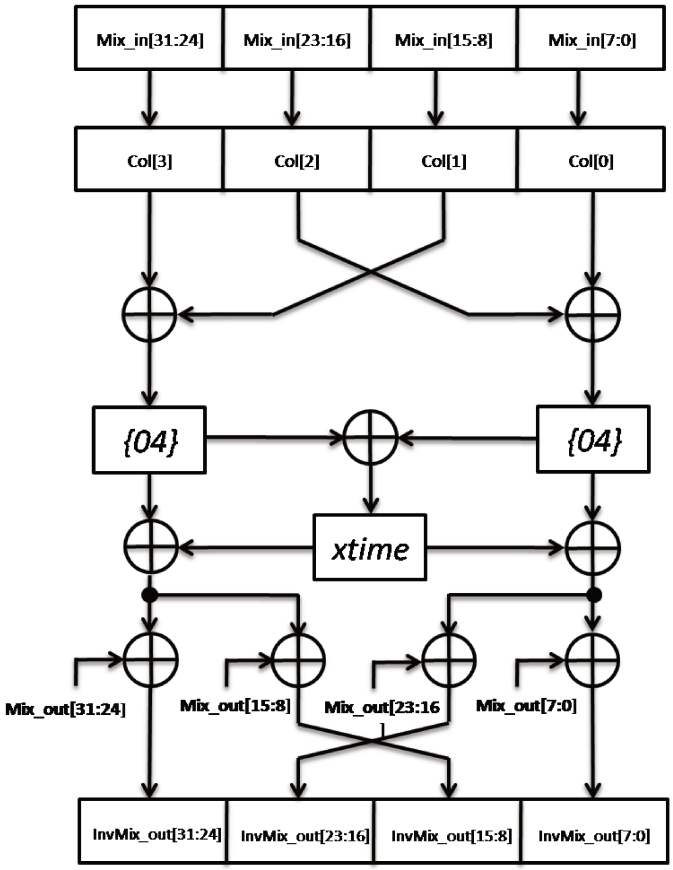


Figura 19 – Micro-arquitetura da operação InvMixColumns()

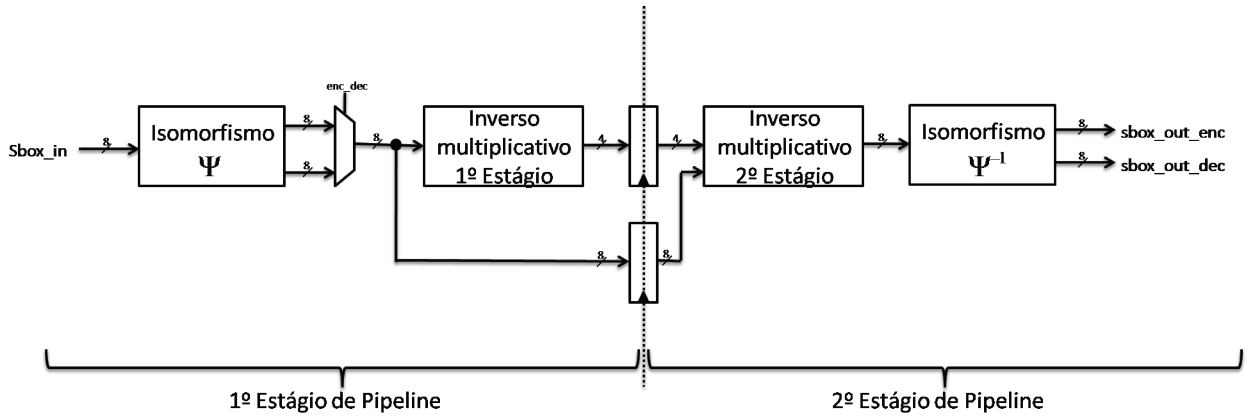


Figura 20 – Micro-arquitetura da operação *SBox*

de derivação é governado pela Equação 3.3, onde $W[i]$ e $W^{'}[i]$ representam palavras de 32 bits de entrada e saída, respectivamente.

$$\begin{cases} W^{'}[0] &= W[0] + g\Big(W[3]\Big) \\ W^{'}[1] &= W[1] + W^{'}[0] \\ W^{'}[2] &= W[2] + W^{'}[1] \\ W^{'}[3] &= W[3] + W^{'}[2] \end{cases} \tag{3.3}$$

A relação entre a palavra $W[i]$ e a chave de entrada é dada pela Equação 3.4

$$W[i] = Key[32 \times (3-i) + 31 : 32 \times (3-i)] \quad para \quad i = 0, 1, 2, 3 \tag{3.4}$$

A Figura 21 mostra como essa operação pode ser implementada. A função $g()$ presente na Equação 3.3 faz uso de uma *SBox* de 32 bits e de um coeficiente RC indexado pelo número do *round* através da relação:

$$RC[i] = x^{i-1} \quad para \quad i = 1, ..., 10 \tag{3.5}$$

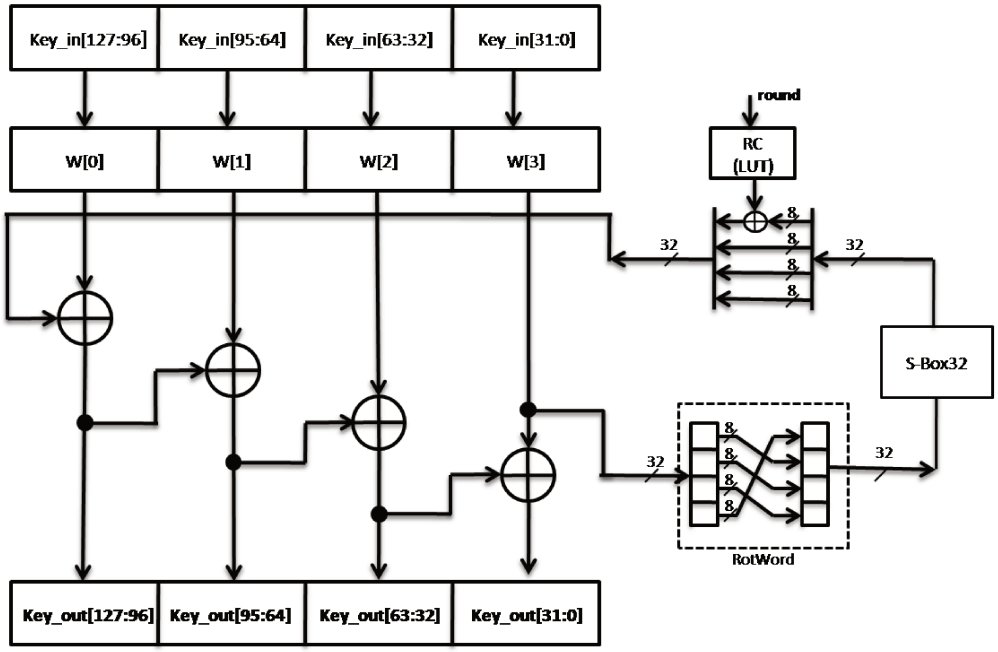


Figura 21 – Implementação da Operação de Derivação de Chave

A operação inversa de derivação de chave é dada pela Equação 3.7 e sua implementação é ilustrada na Figura 22. Os coeficientes $RC^{'}$ utilizados na operação inversa são dados por:

$$RC[i] = x^{10-i} \quad para \quad i = 1, ..., 10 \tag{3.6}$$

$$\begin{cases} W[0] &= W^{'}[0] + g\Big(W[3]\Big) \\ W[1] &= W^{'}[1] + W^{'}[0] \\ W[2] &= W^{'}[2] + W^{'}[1] \\ W[3] &= W^{'}[3] + W^{'}[2] \end{cases} \tag{3.7}$$

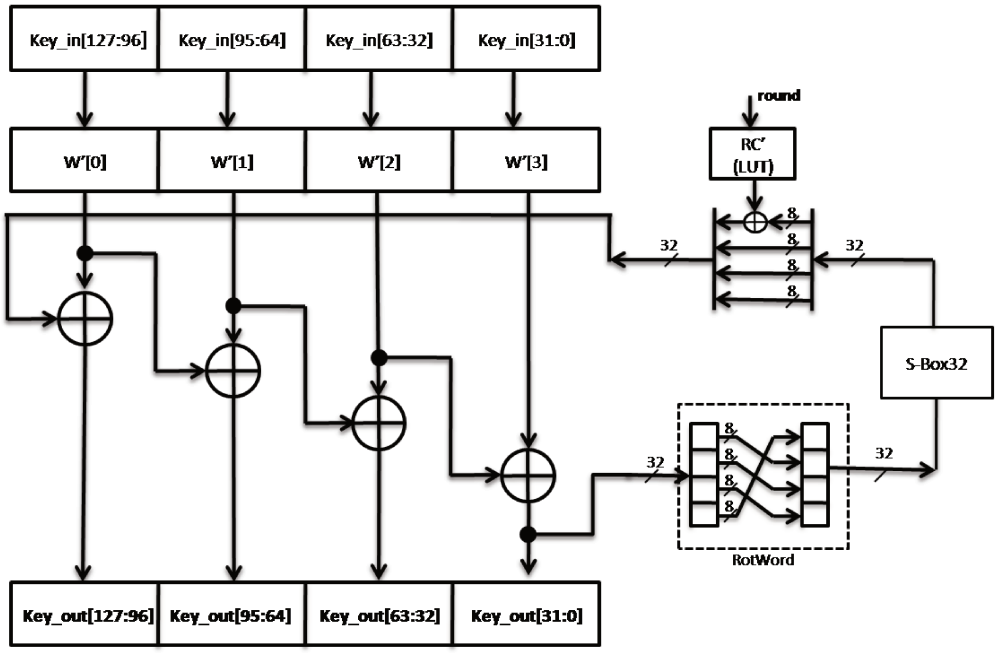


Figura 22 – Implementação da Operação de Derivação de Chave Inversa

As operações de derivação de chave direta e inversa podem ser combinadas, resultando no circuito da Figura 23. Essa é a implementação padrão encontrada na literatura. Veja, por exemplo, (SATOH *et al.*, 2001; ZHANG; PARHI, 2004; FAN; HWANG, 2007).

Essa solução adiciona 4 multiplexadores de 32 bits ao circuito de derivação direta. Como consequência, há um aumento de área ocupada e o caminho crítico do circuito (em destaque na Figura 23) é piorado.

Em arquiteturas iterativas com *datapath* de 32 bits, no entanto, esse circuito pode ser simplificado. Nesse tipo de arquitetura, os dados de entrada são processados uma coluna por vez, de modo que não é absolutamente necessário derivar a chave em um único ciclo. De fato, as chaves podem ser derivadas em paralelo ao processamento das colunas, sendo necessários, portanto, quatro ciclos para a sua conclusão. Caso se deseje compartilhar a *SBox* entre o *datapath* principal e o circuito de derivação de chave, são necessários cinco ciclos, um para o uso da *SBox* e quatro para a computação das chaves. A implementação resultante é mostrada na Figura 24. Como se vê, os multiplexadores de 32 bits já não são necessários. Para entender esse fato, considere que os $W[i]$ são registradores. Para computar $W^{'}[0]$, necessita-se, conforme a Equação 3.3, de $W[0]$ e $W[3]$, ambos já disponíveis nos registradores. Portanto, para o cálculo de $W[0]$, utiliza-se os sinais provenientes da saída dos registradores e não mais os da saída das portas XOR. No segundo ciclo, calcula-se $W^{'}[1]$, que necessita de $W[1]$ e $W^{'}[0]$. Como o valor de $W^{'}[0]$ já foi computado no ciclo anterior e armazenado no registrador, o cálculo de $W^{'}[1]$ também utiliza as saídas dos registradores e não mais as saídas das portas XOR. O mesmo é válido para as outras colunas.

Para que o mesmo circuito seja válido para a derivação da chave de decifração, a computação das colunas deve ser feita na ordem inversa, i.e, a ordem de cálculo deve ser *COl3*, *COL2*, *COL1* e *COL0*.

Uma desvantagem do circuito proposto na Figura 24 é que são necessários 50 ciclos de *clock* para derivar a chave de decifração inicial. Em aplicações nas quais uma mesma chave é utilizada para decriptar muitos blocos de dados, esses ciclos iniciais não são importantes. Se esse não for o caso, pode-se utilizar o circuito mostrado na Figura 25. Nesse caso, são necessários apenas 10 ciclos de *clock* para derivar a chave de decifração inicial. Por outro lado, o caminho crítico e a área do circuito são piorados pela adição de um multiplexador de 32 bits.

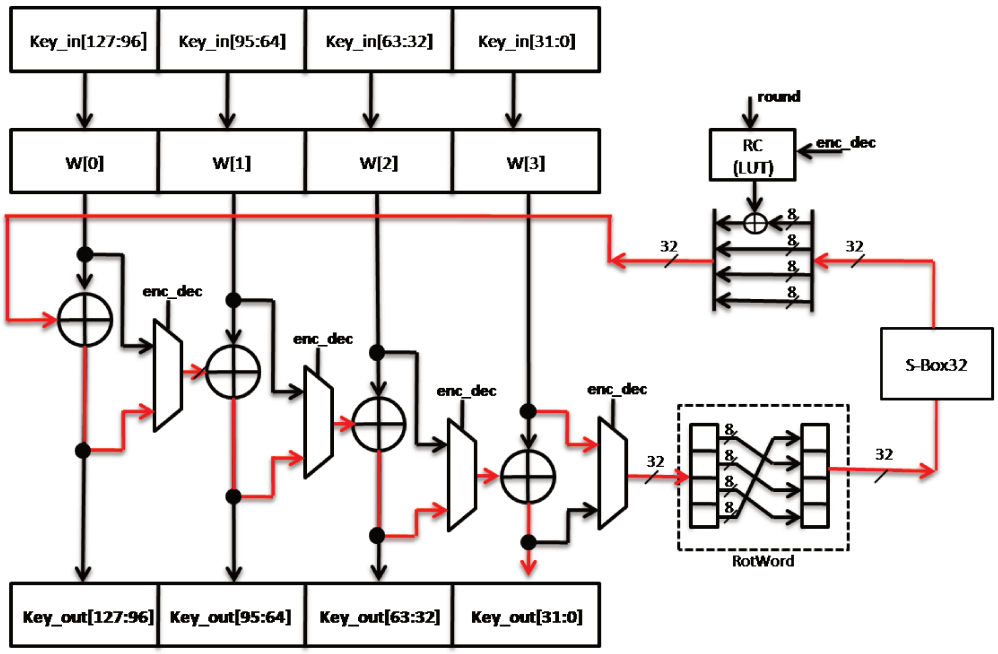


Figura 23 – Implementação da Operação de Derivação de Chave Direta e Inversa

3.4 Projeto Detalhado do Co-processador AES-128 para o modo ECB

Os detalhes de arquitetura e micro-arquitetura discutidos até aqui não são suficientes para especificar uma implementação única de um co-processador AES. Há, ainda, diversos pormenores que precisam ser definidos e que estão diretamente relacionados às funcionalidades que o co-processador deve suportar.

Nesta seção, mostra-se como os circuitos que implementam as principais *layers* do algoritmo AES podem ser combinados para formar o *datapath* do co-processador AES-128. Mostra-se também como esse *datapath* pode ser coordenado por uma unidade de controle

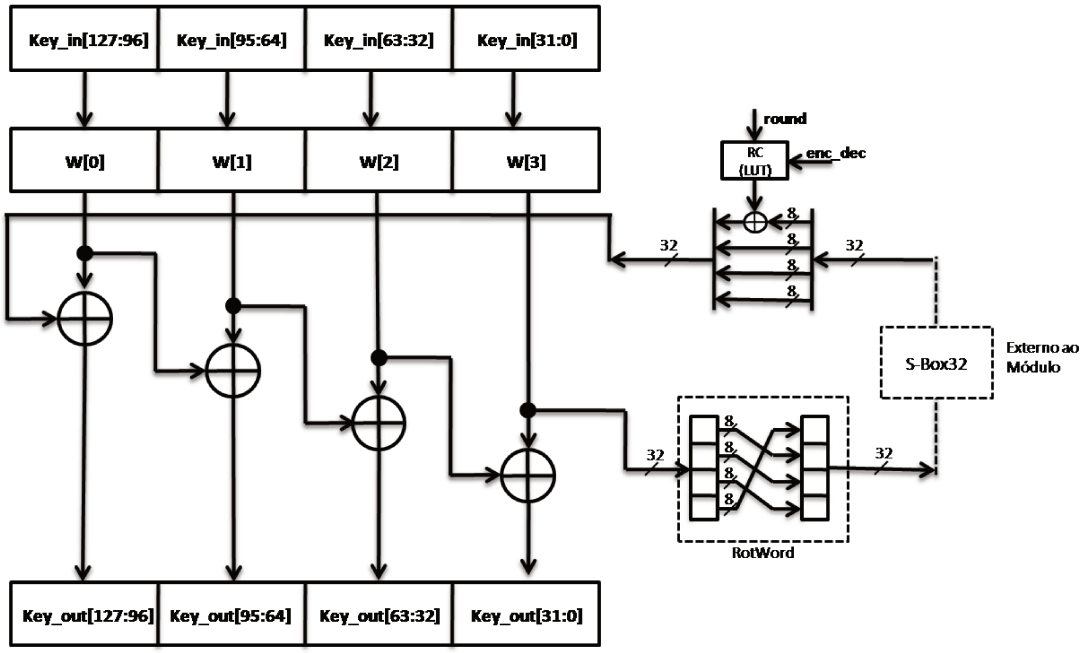


Figura 24 – Implementação da Operação de Derivação de Chave Direta e Inversa utilizando 5 ciclos de *clock*

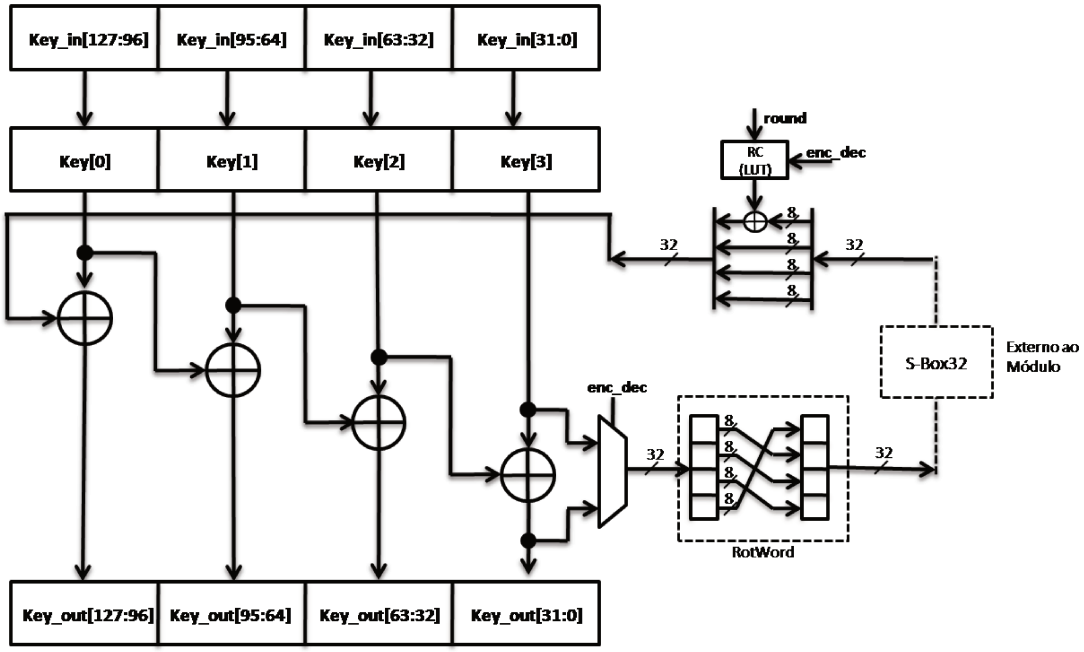


Figura 25 – Implementação da Operação de Derivação de Chave utilizando 5 ciclos de *clock* para a operação direta e 1 ciclo para operação inversa

para implementar as operações de cifração, decifração e algumas funcionalidades básicas segundo o modo ECB.

3.4.1 Funcionalidades do Co-processador AES-128

No projeto do co-processador AES-128 proposto neste trabalho, optou-se por implementar apenas funcionalidades básicas relacionadas ao algoritmo AES. A principal razão é manter esse bloco de *hardware* suficientemente genérico para ser utilizado em uma ampla gama de aplicações. Assim, caso se deseje funcionalidades não previstas no projeto original, pode-se, com relativa facilidade, adicionar circuitos externos e, possivelmente, pequenas alterações ao seu *core* com o intuito de suportar essas novas funções. Por exemplo, é possível desabilitar de forma síncrona o co-processador AES-128 em qualquer ciclo de *clock*. Essa característica, por si só pouco útil, pode, com auxílio de circuito externo, ser utilizada para implementar um mecanismo de interrupção e priorização de mensagens. No Capítulo 4, mostra-se como isso pode ser feito.

Seguindo essa filosofia de projeto, a interface do AES-128 com módulos externos foi mantida genérica, não aderindo a nenhum padrão de barramento específico. Ainda assim, é fácil realizar a conversão de algum padrão de barramento, como o AMBA APB, para o adotado no AES-128.

Os sinais que compõem a interface do AES-128 são descritos na Tabela 2. Os dados a serem processados são internamente armazenados em 128 flip-flops do tipo D, que são agrupados em 4 registradores de 32 bits denominados *Col3*, *Col2*, *Col1* e *Col0*. Esses registradores são escritos e lidos, um por vez, através dos barramentos de entrada e saída, *bus_in* e *col_out*, respectivamente. Como esses barramentos são comuns a todos os registradores, utiliza-se o sinal *addr* para indicar em qual das colunas será realizada a escrita, quando *write_en = 1*, ou a leitura, quando *read_en = 1*.

Da mesma forma, a chave é internamente armazenada em 128 flip-flops do tipo D que são agrupados em 4 registradores de 32 bits denominados *Key3*, *Key2*, *Key1* e *Key0*. Esses registradores também são escritos através do barramento de entrada *bus_in*, mas agora sob o controle dos sinais *key_en*. A leitura dos registros de chave, por sua vez, é feita por meio do barramento de saída *key_out* sob o controle do sinal *key_sel_rd*.

O sinal de controle *op_mode* seleciona um dos quatro modos de operação do co-processador:

- **Encriptação:** Encripta os dados armazenados em *Col* utilizando a chave de cifração armazenada em *Key*.
- **Decriptação:** Decripta os dados armazenados em *Col* utilizando a chave de decifração armazenada em *Key*.
- **Derivação de Chave:** Gera, a partir da chave de cifração, a chave de decifração.

- **Derivação de Chave seguida de Decriptação:** Gera a chave de decifração e a utiliza para decriptar os dados armazenados em *Col*.

Para que o AES-128 inicie sua operação, deve-se manter o sinal *start* em nível lógico alto por pelo menos um ciclo de *clock*. A indicação de que o processamento foi finalizado é dada pelo sinal *end_aes*, que permanece em nível lógico alto por um ciclo de *clock*.

Também é possível desabilitar a operação do AES-128 a qualquer instante. Para isso, basta gerar um pulso no sinal *disable_core*.

Sinal	**Direção**	**Largura (em bits)**	**Descrição**
bus_in	Entrada	32	Barramento de Entrada
key_en	Entrada	4	Sinal *one-hot* que habilita a escrita nos registros de chave
key_sel_rd	Entrada	2	Sinal utilizado para selecionar qual registro de chave será lido
addr	Entrada	2	Sinal utilizado para selecionar qual coluna será lida ou escrita
write_en	Entrada	1	Sinal que habilita a escrita nos registros de colunas
read_en	Entrada	1	Sinal que habilita a leitura dos registros de colunas
op_mode	Entrada	2	Seleciona o modo de operação do AES-128
start	Entrada	1	Sinaliza que a computação do algoritmo pode começar
disable_core	Entrada	1	Encerra qualquer computação realizada pelo AES-128
col_out	Saída	32	Barramento utilizado para leitura das colunas
key_out	Saída	32	Barramento utilizado para leitura das chaves
end_aes	Saída	1	Sinal que indica o término da computação do algortimo

Tabela 2 – Interface do Co-processador AES-128

3.4.2 *Datapath* do Co-processador AES-128

Por conveniência, o *datapath* do co-processador AES-128 será dividido em cinco circuitos. Essa divisão, no entanto, serve apenas para propósitos de exposição, uma vez que todos esses circuitos estão interligados formando um único *hardware*.

3.4.2.1 Circuito de Entrada dos Registradores *Col*

Cada um dos quatro registradores *Col[i]* possui o mesmo circuito de entrada, que especifica sua fonte de dados assim como seus sinais de *enable*. Uma representação, em nível RTL, desse circuito é dada na Figura 26. Os sinais de controle envolvidos são listados na Tabela 3.

Sinal	**Largura (em bits)**	**Descrição**
enc_dec_i	1	Indica se a operação é de cifração ou decriptaçao
col_sel_i	2	Seleciona a fonte dos dados que serão salvos nos registros *Col[i]*
col_en_host_i	4	Sinal de *enable one_hot* gerado pela interface com o usuário
col_en_cnt_unit_i	4	Sinal de *enable one_hot* gerado pela pela unidade de controle
bypass_rk_i	1	Habilita o *bypass* dos registros de *pipeline*

Tabela 3 – Sinais de controle associados ao circuito de entrada dos registros *Col*

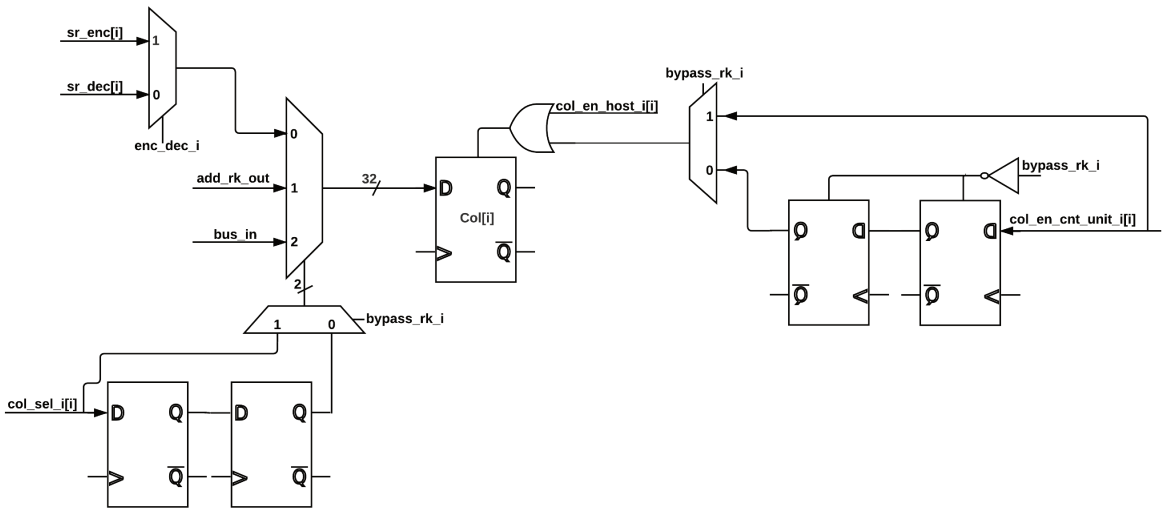


Figura 26 – Circuito de Entrada dos Registradores *Col*

Os sinais *col_en_cnt_unit_i* e *col_en_host_i* não devem ser ativados no mesmo ciclo de *clock*. Como o circuito apresentado assume que essa condição é verdadeira, ela deve ser garantida por alguma lógica externa.

O sinal *col_sel_i* é gerado pela unidade de controle e seleciona a origem da fonte de dados dos registros *Col[i]*. Quando o co-processador não está operando, esse sinal seleciona a entrada *data_in* do multiplexador 3×1 de 32 bits. Nesse caso, qualquer escrita para os registros *Col[i]* é proveniente da interface externa. Quando o AES-128 está realizando alguma operação, o sinal *col_sel_i* seleciona ou a entrada *add_rk_out* ou as entradas *sr_enc* e *sr_dec*.

Como o *datapath* possui três estágios de *pipeline*, é necessário acrescentar dois registros de *pipeline* aos sinais de controle para que eles só atuem no circuito no último estágio. Em certos casos, no entanto, é desejável realizar um *bypass* desses registros para que os sinais de controle atuem no mesmo ciclo de *clock* em que são gerados. No circuito da Figura 26, o *bypass* é controlado pelo sinal *bypass_rk_i*.

Uma situação em que se utiliza *bypass* é durante o *Round 0*. Nesse caso, todo o processamento que é realizado se limita a uma operação XOR entre os registros *Col[i]* e *Key[i]*. Como essa operação pode ser realizada em apenas um ciclo de *clock*, faz-se um *bypass* do *pipeline*. Como resultado, são necessários apenas 4 ciclos para se concluir o *Round 0*. Caso o *bypass* não fosse utilizado, seriam necessários 6 ciclos.

3.4.2.2 Circuito de Entrada dos Registradores *Key*

Para armazenar a chave, são utilizados dois bancos de registradores. Isso foi feito para que o usuário do co-processador não precise reescrever a chave nos registros *Key[i]* a cada rodada do algoritmo. De fato, esses registros são modificados durante a operação do

algoritmo o que acarreta a perda dos valores originais. Adicionando-se um outro conjunto de registradores, no caso *Key_Host[i]*, só é necessário realizar uma nova escrita para os registros de chave caso se deseje processar um bloco de dados com outro valor de chave.

Sempre que a interface do usuário realiza uma operação de escrita para as chaves através do barramento de entrada *bus_in*, os dois bancos de registros são habilitados, i.e, o mesmo valor é armazenado em *Key_Host* e *Key*. Isso é feito para garantir a consistência das leituras dos registros de chave por parte da interface com o usuário. Para economizar *hardware*, o multiplexador utilizado pelo algoritmo AES é reutilizado pela interface externa, de modo que o usuário lê os registros *Key[i]*. Caso os dois bancos de registros não fossem salvos simultaneamente pela interface externa, a única situação que geraria uma leitura inconsistente seria aquele em que o usuário realiza uma escrita nos registros *Key_Host* seguida de uma leitura dos registros *Key*. Certamente, esse tipo de operação não possui qualquer uso prático. Ainda assim, optou-se por habilitar a escrita simultânea nos registros para garantir a consistência das leituras em qualquer situação.

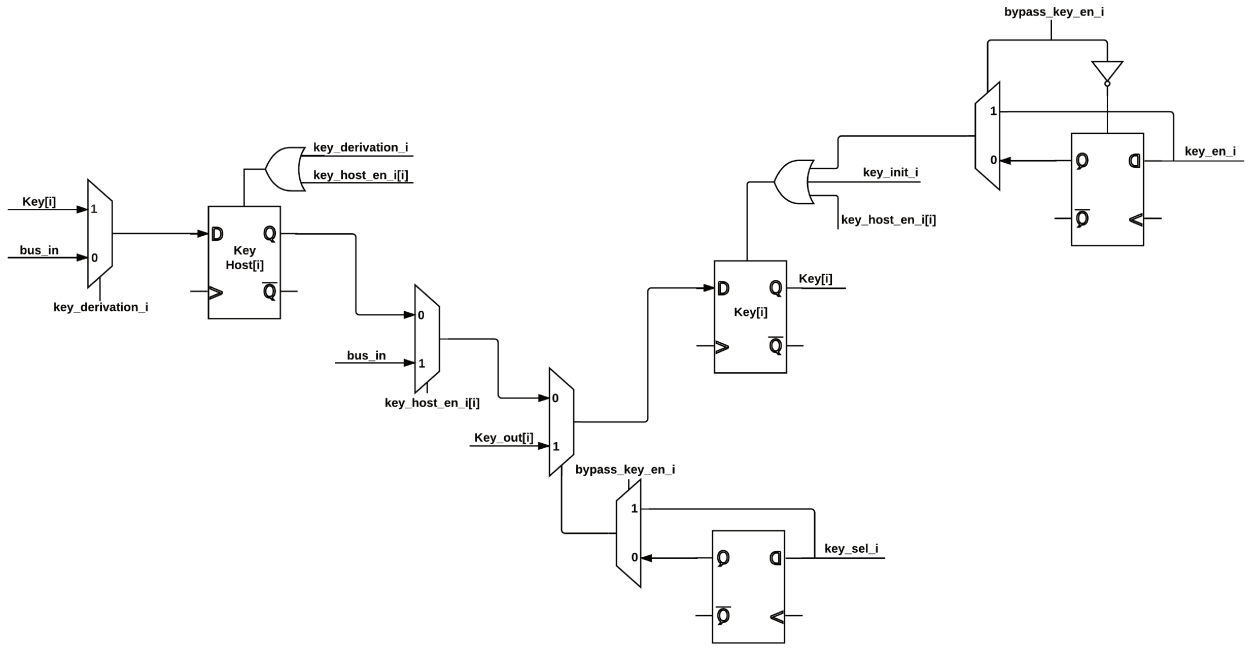


Figura 27 – Circuito de Entrada dos Registradores *Key*

Sinal	Largura (em bits)	Descrição
key_derivation_en_i	1	Indica se a operação de Derivação de Chave foi habilitada
key_host_en_i	4	Sinal de *enable one_hot* gerado pela interface com o usuário
bypass_key_en_i	1	Habilita o *bypass* dos registros de *pipeline*
key_sel_i	1	Seleciona a fonte dos dados que serão salvos nos registros *Key[i]*
key_init_i	1	Habilitado para que a chave armazenada em *Key_host* seja transferida para *Key*
key_en_i	4	Sinal de *enable one_hot* gerado pela pela unidade de controle

Tabela 4 – Sinais de controle associados ao circuito de entrada dos registros *Key*

Sempre que o algoritmo se inicia, gera-se um pulso no sinal *key_init_i* para que o valor armazenado nos registros *Key_Host[i]* seja transferido para os registros *Key[i]*.

Nos modos de operação em que há derivação de chave, o sinal *key_derivation_i* é ativado para que a nova chave computada seja salva nos registros *Key_Host[i]*.

É importante destacar que na Figura 27 só se utiliza um registro de *pipeline* para sincronizar os sinais de controle. Isso ocorre porque o processo de derivação de chave só requer dois ciclos de *clock*. No segundo ciclo a chave gerada é salva nos registro *Key[i]* e no terceiro ciclo elas são utilizadas na *layer AddRoundKey()*.

Durante a operação de derivação de chave, há um *bypass* dos registros de *pipeline* para que os sinais de controle atuem no mesmo ciclo de *clock* em que são gerados. Esse *bypass* é controlado pelo sinal *bypass_key_en_i*.

3.4.2.3 Circuito da *SBox*, *MixColumns* e *AddRoundKey*

O circuito que implementa as *layers SBox()* (direta e inversa), *MixColumns()* (direta e inversa) e *AddRoundKey()* pode ser visto na Figura 28, onde os blocos *SBox* e *MixColumns* encapsulam os circuitos apresentados na seção 3.3. Os sinais de controle utilizados são listados na Tabela 5.

Como a largura do *datapath* é 32 bits, há um multiplexador 5×1 de 32 bits que seleciona a fonte de dados a ser processada no ciclo de *clock* corrente. As alternativas são os quatro registros *Col[i]* e a saída $g()$ do circuito de geração de chave. Esse multiplexador é controlado pelo sinal *sbox_sel_i* gerado pela unidade de controle. Para economizar *hardware*, esse multiplexador é reaproveitado para gerar a saída multiplexada *col_out* que é lida pela interface externa. Nesse caso, a seleção da coluna que será lida é feita pelo sinal *col_sel_host_i*. Os sinais *sbox_sel_i* e *col_sel_host_i* não devem ser ativados no mesmo ciclo de *clock*.

Sinal	**Largura (em bits)**	**Descrição**
key_derivation_en_i	1	Indica se a operação de Derivação de Chave foi habilitada
key_host_en_i	4	Sinal de *enable one_hot* gerado pela interface com o usuário
bypass_key_en_i	1	Habilita o *bypass* dos registros de *pipeline*
last_round_i	1	Sinaliza que o *round* corrente é o último
rk_sel_i	2	Seleciona a fonte de dados da operação *AddRoundKey()*

Tabela 5 – Sinais de controle associados ao circuito da *SBox*, *MixColumns* e *AddRoundKey*

Como discutido no Capítulo 2, para que o mesmo *datapath* seja utilizado tanto para a cifração quanto para a decifração é necessária uma pequena modificação na etapa *AddRoundKey()* da decifração. Na Figura 28, essa modificação é implementada pela inclusão de portas XOR na saída do bloco *SBox* correspondente à operação inversa.

O sinal *rk_sel_i* gerado pela unidade de controle seleciona a entrada da operação *AddRoundKey()*. As opções são *sbox_in* (selecionada durante o *Round 0*), *sbox_pp2* (selecionada durante o último *round*) e *mix_out_enc* (selecionada nos demais casos).

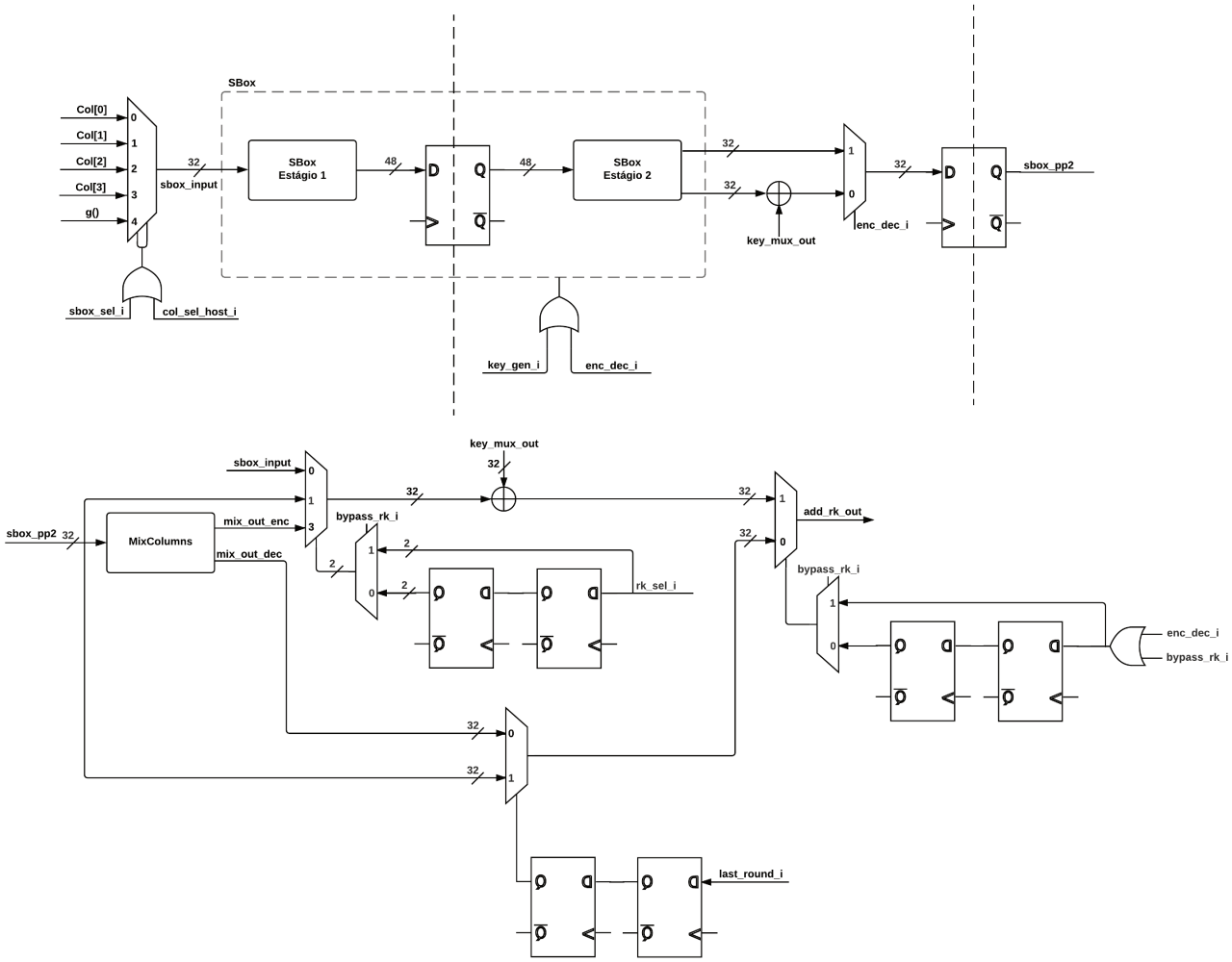


Figura 28 – Circuito da *SBox*, *MixColumns* e *AddRoundKey*

Todos os sinais de controle utilizados passam por dois registros de *pipeline* para que sua atuação só ocorra no último estágio.

3.4.2.4 Circuito de *ShiftRows()*

A operação *ShiftRows()* é feita no último estágio de processamento das colunas. Para a operação de cifração, isso corresponde ao ciclo em que o registro *Col[3]* é processado. Assim, a operação *ShiftRowns()* opera sobre o barramento *{add_rk_out, Col[2], Col[1], Col[0]}*, onde o sinal *add_rk_out* corresponde à saída final do *round*. Durante a decifração, a máquina de estados associada à unidade de controle é percorrida no sentido inverso, de modo que o último estágio de processamento corresponde à atualização do registro *Col[0]*. Nesse caso, a operação *ShiftRowns()* opera sobre o barramento *{Col[3], Col[2], Col[1], add_rk_out}*. A Figura 29 ilustra como essa seleção pode ser implementada.

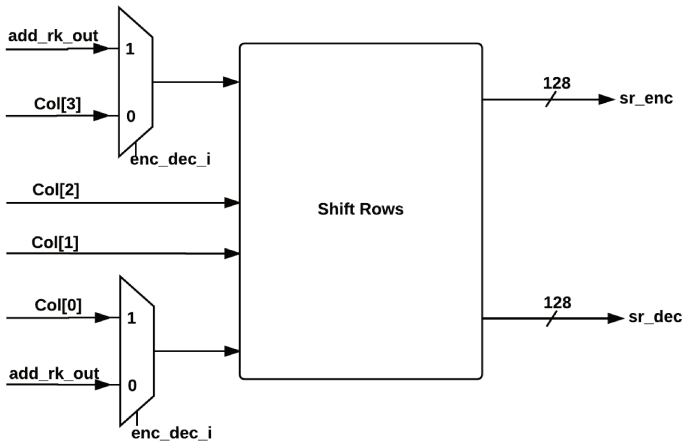


Figura 29 – Circuito de *ShiftRows()*

3.4.2.5 Circuito de Geração de Chave

O circuito de geração de chave é mostrado na Figura 30, onde o bloco *Key Expander* corresponde ao circuito apresentado na seção 3.3.3. O multiplexador 4×1 de 32 bits mostrado nessa Figura é utilizado para selecionar qual dos registros *Key[i]* será utilizado na operação *AddRoundKey()*. Como essa operação normalmente ocorre apenas no último estágio do *pipeline*, é necessário acrescentar dois registros de *pipeline* ao sinal de controle *key_out_sel_i*. Durante o *Round 0* e nos modos em que há derivação de chave, no entanto, deve ocorrer o *bypass* desses registros.

O multiplexador utilizado no circuito em questão é reaproveitado pela interface externa, o que explica a presença do sinal *key_sel_rd_i*.

Como a operação de derivação de chave só requer dois ciclos de *clock*, é suficiente atrasar o sinal *round_i* em um ciclo, o que justifica a presença de apenas um registrador de 4 bits associado a esse sinal.

3.4.3 Unidade de Controle

O papel da unidade de controle é coordenar e sequenciar o *datapath* através da ativação dos sinais de controle adequados. Sua implementação se dá, em geral, por meio de uma máquina de estados finitos (MEF).

A máquina de estados associada ao co-processador AES-128 é ilustrada na Figura 31. Nessa figura, as transições são codificadas por números cujo significado é dado na Tabela 6. A geração dos sinais de controle é sumarizada na Tabela 17 do apêndice A.

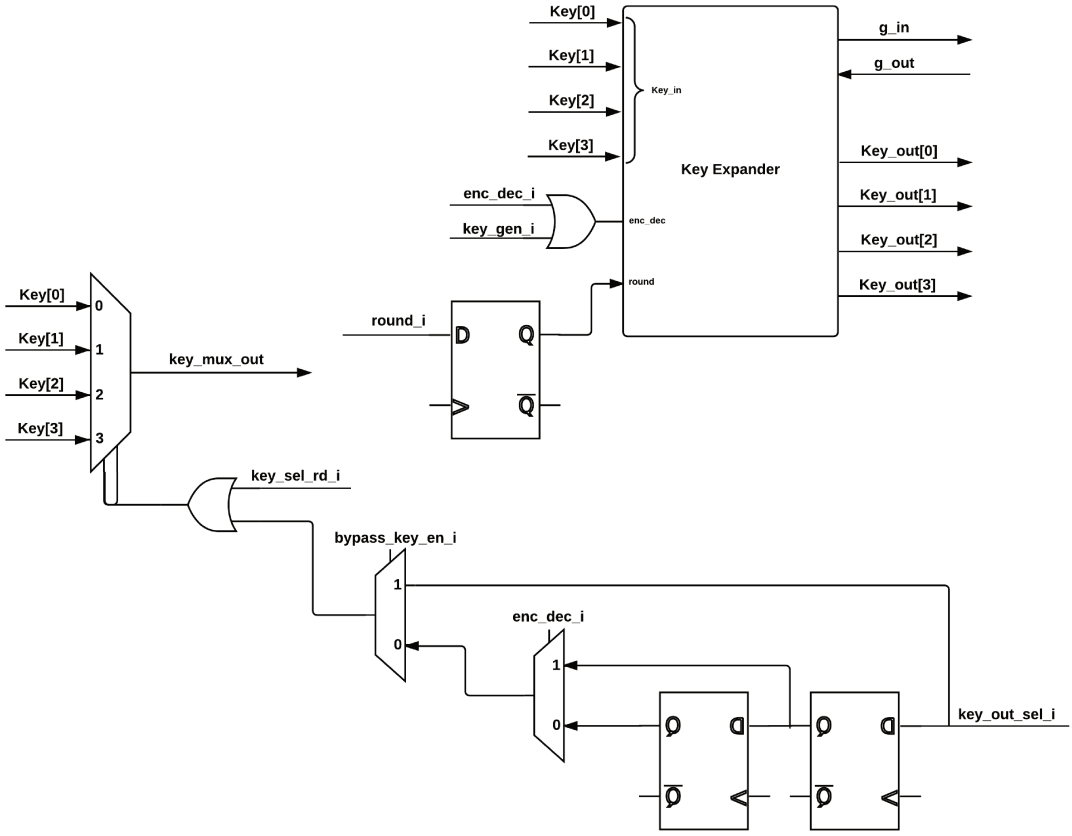


Figura 30 – Circuito de Geração de Chave

Após o *reset*, a máquina de estados vai para o estado *IDLE* onde permanece até que o sinal *start* seja mantido em nível lógico alto por pelo menos um ciclo de *clock*. Enquanto está nesse estado, a MEF permanece em repouso com todos os sinais de controle mantidos desabilitados, de modo que o *datapath* não executa nenhuma ação e os registros *Col[i]* e *Key_Host[i]* estão livres para serem lidos e escritos pela interface externa.

Quando o sinal *start* é ativado, a máquina vai ou para estado *GEN_KEY0*, quando o sinal *op_mode* indica que algum dos modos de geração de chave foi ativado, ou para o estado *ROUND0_COL0*, caso contrário.

Os estados *GEN_KEYx* são responsáveis por coordenar o processo de derivação de chave, que requer 10 ciclos de *clock* para ser concluído. Assim, os estados *GEN_KEYx* devem ser percorridos sequencialmente 10 vezes. Na última iteração, a máquina ou volta para o estado *IDLE* (transição 15), quando a operação selecionada por *op_mode* se limita a derivação de chave, ou vai para o estado *ROUND0_COL3*, iniciando, assim, o processo de decifração.

Os estados *ROUND0_COLx* são utilizados para implementar o *Round 0* do algoritmo AES. Na cifração, esses estados são percorridos na ordem crescente, enquanto na decifração eles são percorridos na ordem inversa.

Os estados *ROUND_COLx* são utilizados para implementar os demais *Rounds* do algoritmo. Durante a cifração, a ordem de execução desses estados é: {*ROUND_KEY0*, *ROUND_COL0*, *ROUND_COL1*, *ROUND_COL2*, *ROUND_COL3*, *NOP*, *ROUND_KEY0*, *ROUND_COL0*, ...}. Note que no primeiro *round* não é necessária a inserção do ciclo de NOP, que só ocorre a partir do segundo *round*. Essa sequência é percorrida 10 vezes para implementar os 10 *rounds* do algoritmo AES. Durante a decifração, a ordem de execução dos estados é invertida: {*ROUND_KEY0*, *ROUND_COL3*, *ROUND_COL2*, *ROUND_COL1*, *ROUND_COL0*, *NOP*, *ROUND_KEY0*, *ROUND_COL3*, ...}. O compartilhamento da *SBox* é controlado pelo estado *ROUND_KEY0*.

Quando o algoritmo finaliza sua operação, o estado *READY* é ativado, para só então a máquina retornar para o estado *IDLE*.

Transição	**Condição**
1	!start
2	start && op_mode == ENCRYPTION
3	start && op_mode == DECRYPTION
4	start && (op_mode == ENCRYPTION || op_mode == DECRYPTION)
5	enc_dec
6	!enc_dec
7	last_round
8	!first_round && !last_round
9	first_round && enc_dec
10	first_round && !enc_dec
11	enc_dec && last_round
12	!enc_dec
13	!last_round && enc_dec
14	last_round && op_mode == KEY_DERIVATION
15	last_round && op_mode != KEY_DERIVATION
16	!last_round

Tabela 6 – Transições da Máquina de Estados

3.5 Extensão para os modos CBC e CTR

Nesta seção, mostra-se como a arquitetura de *hardware* discutida até então pode ser estendida para suportar os modos de operação CBC e CTR. O funcionamento desses modos é ilustrada nas Figuras 32 e 33, onde o bloco $e()$ representa a cifração AES segundo o modo ECB. Para mais detalhes, consulte (PAAR; PELZL, 2009).

Para suportar os modos de operação CBC e CTR, deve-se adicionar o circuito da Figura 34 ao *datapth* do co-processador. Nesse caso, a entrada *bus_in* da Figura 26 deve ser substituída pela saída *data_in* da Figura 34.

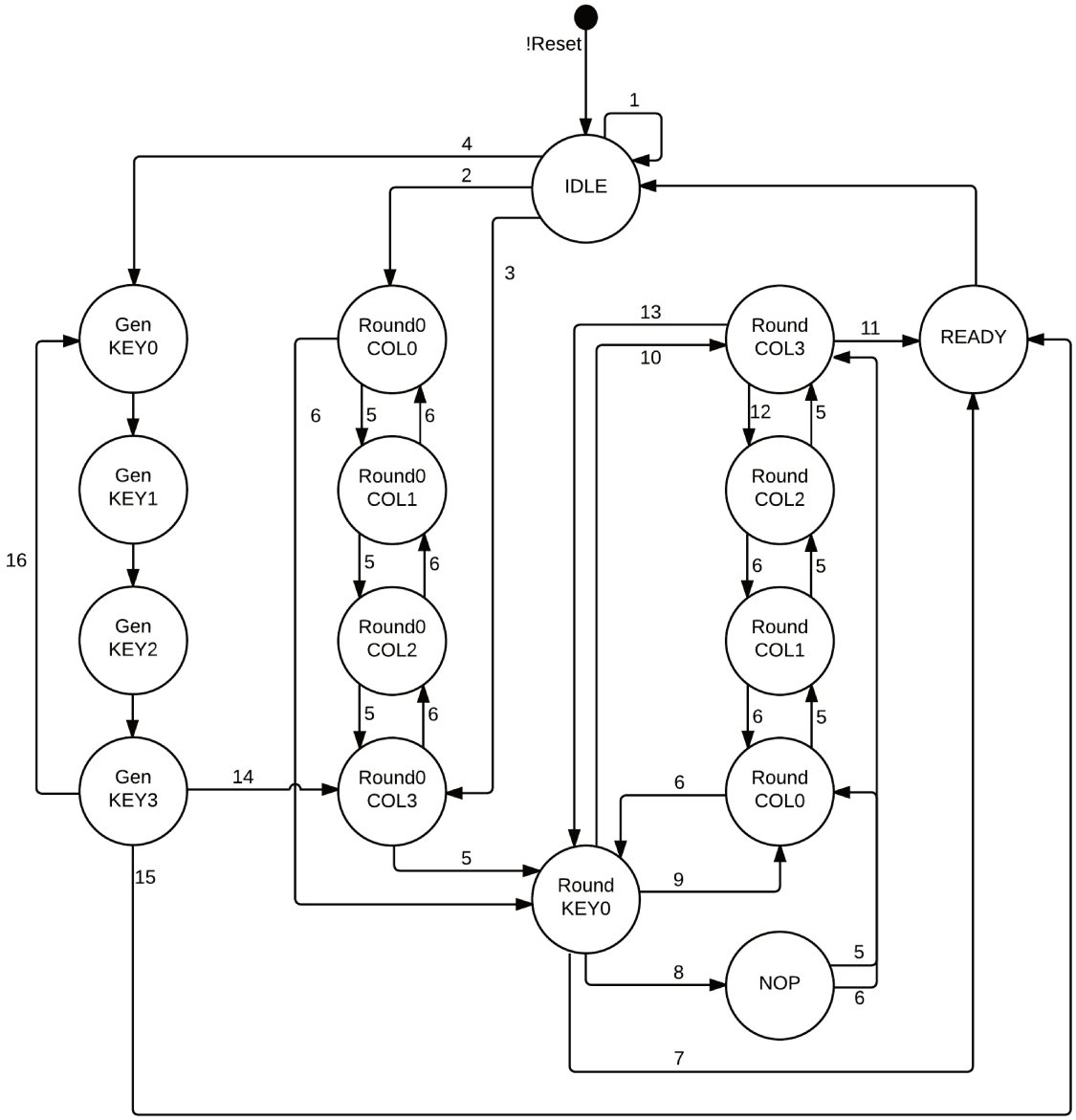


Figura 31 – Máquina de Estados

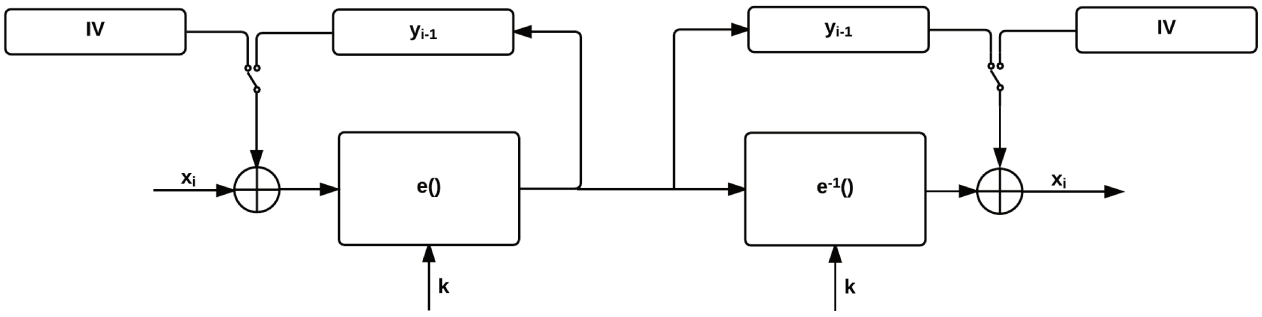


Figura 32 – Encriptação e Decriptação segundo o modo CBC

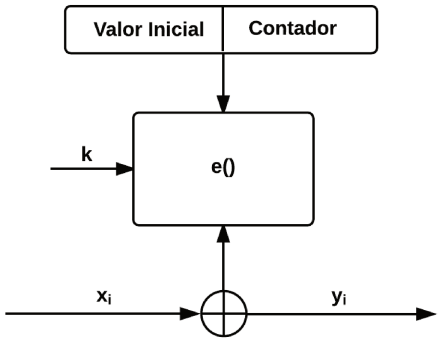


Figura 33 – Encriptação e Decriptação segundo o modo CTR

Também é necessário acrescentar dois bancos de flip-flops do tipo D de 128 bits, aqui denominados *bkp* e *bkp_1*, que são agrupados em quatro registrados de 32 bits, denominados *bkp[i]* e *bkp_1[i]*. O circuito de entrada para esses registradores é mostrado na Figura 35.

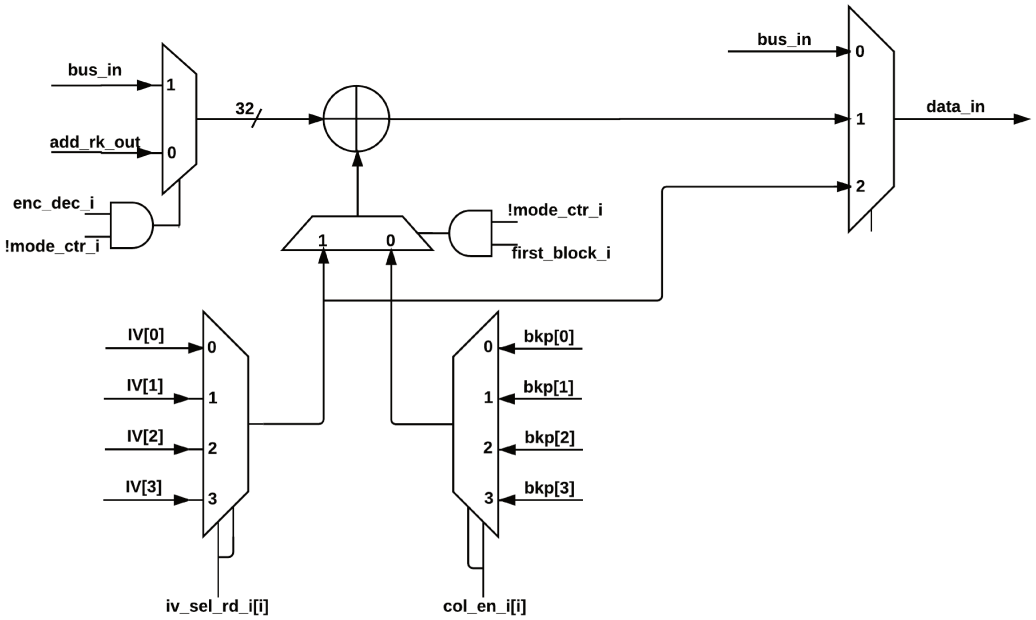


Figura 34 – Circuito Necessário para suportar os modos CBC e CTR

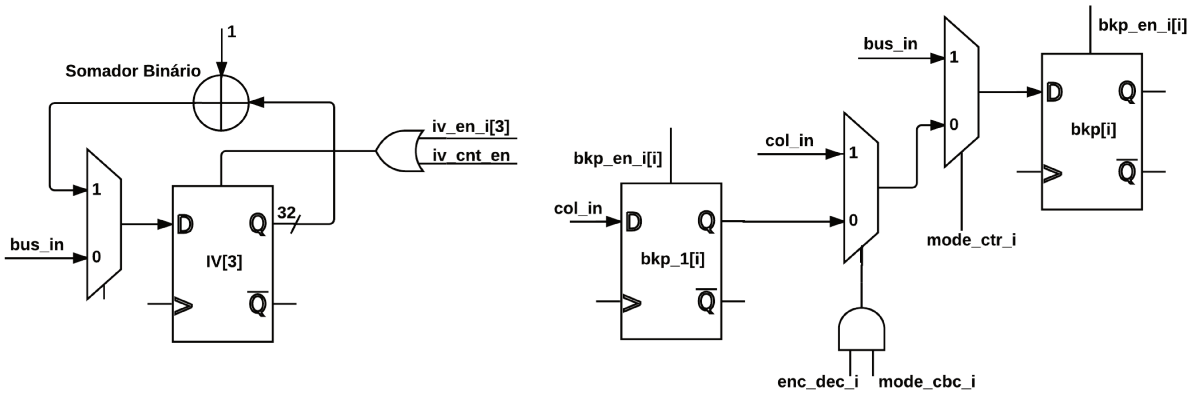


Figura 35 – Circuito de Entrada dos registradores *bkp[i]* e *bkp_1[i]*

As modificações na unidade de controle necessárias para suportar os modos CBC e CTR são indicadas na Tabela 18 do apêndice A.

3.6 Trabalhos Relacionados

Com o intuito de comparar a arquitetura proposta com outras soluções disponíveis na literatura, o *hardware* deste trabalho (sem o suporte aos modos CBC e CTR) foi implementado em um dispositivo SmartFusion A2F200M3 (MICROSEMI, 2015). Os resultados obtidos são sumarizados nas Tabelas 7 e 8, onde a unidade GE (do inglês, *Gate Equivalent*) (KAESLIN, 2014) foi utilizada para facilitar a comparação com arquiteturas implementadas em outras tecnologias. Para compor a Tabela 7, assumiu-se que um *flip-flop* equivale a 6.75 GEs (MORIOKA; SATOH, 2002).

Módulo	**Lógica (GE)**	**Flip-flops**	**Total (GE)**	**%**
Datapath	3222	372	5733	95.72
Unidade de Controle	115	21	256	4.28
Total	3337	393	5989	100

Tabela 7 – Relatório de Área da arquitetura proposta

Parâmetro	
Frequência Máxima	74.51 MHz
Ciclos de *Clock*	64
Throughput	149.02 Mb/s

Tabela 8 – Performance da arquitetura proposta

Em (MORIOKA; SATOH, 2002), o autor propõe uma arquitetura para um co-processador AES-128 com suporte ao modo ECB. Essa é, até o momento, a implementação

baseada em uma arquitetura iterativa com *datapath* de 32 bits mais eficiente reportada na literatura. Sua implementação foi realizada em tecnologia CMOS de 110 nm e os resultados obtidos estão reproduzidos na Tabela 9.

Parâmetro	
GEs	5398
Frequência Máxima	131.24 MHz
Ciclos de *Clock*	54
Throughput	311.09 Mb/s

Tabela 9 – Relatório da Implementação proposta em (MORIOKA; SATOH, 2002)

Como esperado, o número de portas equivalentes (GE) da arquitetura proposta no presente trabalho é maior que aquele reportado em (MORIOKA; SATOH, 2002). Essa diferença se deve basicamente aos flip-flops e à lógica combinacional necessários para suportar o *sub-pipelining*. A diferença, no entanto, é de apenas 11%, o que pode ser considerado um número baixo. De fato, apenas a inserção dos 48 flip-flops necessários para dividir a *SBox* em dois estágios já corresponde a um acréscimo de 6% no número total de portas equivalentes. As principais otimizações que permitem amortizar esse crescimento de área são:

- **Otimização da operação de derivação de chave:** A otimização proposta na seção 3.4.1 permite reduzir o número de portas equivalentes da unidade de geração de chave em 34.7%;
- **Uso de SBox mais eficiente:** Este trabalho utiliza a estrutura de *SBox* proposta em (CANRIGHT, 2005), que é 20% mais eficiente que aquela utilizada em (MORIOKA; SATOH, 2002);
- **Implementação do gerador de coeficientes *Rcon()*:** Diferente da arquitetura sob comparação, no presente trabalho, a unidade *Rcon()* é puramente combinacional, o que permite uma redução em sua área. Por outro lado, o caminho crítico desse circuito é piorado. Essa piora, no entanto, não tem grandes implicações. De fato, mesmo que a unidade *Rcon()* esteja no caminha crítico do circuito de geração de chave, seu atraso é compensado pela otimização proposta na seção 3.4.1;
- **Uso de apenas uma unidade *InvMixColumns()*:** O *datapath* do co-processador proposto faz uso de apenas uma unidade *InvMixColumns()*, enquanto o *datapath* de (MORIOKA; SATOH, 2002) usa duas instâncias dessa unidade;

A comparação em termos de *throughput* é mais difícil de ser feita objetivamente devido à diferença de tecnologia em que as arquiteturas foram implementadas. Enquanto

o presente trabalho foi implementado em uma FPGA de 130 nm (MICROSEMI, 2015), o trabalho sob comparação foi implementado em um ASIC de 110 nm. Tanto a diferença de nó tecnológico quanto a diferença de plataforma alvo têm impactos significativos na frequência máxima com que determinado circuito pode operar. Segundo (KUON; ROSE, 2007), uma implementação em FPGA é, em média, de 3 a 4 vezes mais lenta que uma implementação em ASIC. Os resultados desse estudo, no entanto, baseiam-se em FPGAs e ASICs de 90 nm, de modo que suas conclusões não podem, a rigor, ser diretamente aplicados na comparação aqui realizada. De todo modo, acredita-se que, caso implementadas na mesma tecnologia, a arquitetura proposta neste trabalho apresente desempenho significativamente superior àquela de (MORIOKA; SATOH, 2002).

Supondo, via extrapolação, que a diferença de desempenho devido a implementação em FPGA seja de 3, a arquitetura proposta atingiria um *throughput* de 447.06 Mb/s, que é 1.44 vezes maior que aquele de (MORIOKA; SATOH, 2002). Caso a diferença de desempenho seja de 4, o *throughput* da arquiteura proposta passa a ser 596.24 Mb/s, que é 1.9 vezes maior que aquele de (MORIOKA; SATOH, 2002). Em ambos os casos, o ganho de desempenho é inferior ao previsto pela Equação 3.1, que é cerca de 2.5. As principais razões para essa diferença são:

- A rigor, a Equação 3.1 só pode ser utilizada para comparar o ganho de desempenho devido a inserção de *pipeline* em certa arquitetura. Nas comparações anteriores, no entanto, se está relacionando os ganhos de desempenho teórico e real, mas com o ganho real sendo comparado em relação a duas arquiteturas distintas. É esperado, assim, um desvio do valor teórico, para mais ou para menos, devido à diferença entre os atrasos das lógicas combinacionais que compõe o *round* dessas duas arquiteturas;
- A Equação 3.1 despreza os atrasos das interconexões e supõe que o tempo de *setup* dos flip-flops é zero, enquanto os resultados de síntese incorporam esses valores;
- Apesar do esforço despendido para este fim, é difícil, na prática, manter os estágios do *pipeline* perfeitamente balanceados. Esse desbalanceamento, que é ignorado pela Equação 3.1, impacta de forma negativa o desempenho da implementação final;
- A Equação 3.1 assume que não há qualquer acréscimo de lógica combinacional devido ao *sub-pipelining*. Na prática, pode ser necessário adicionar pequenas porções de lógica ao *datapath* para suportar o *pipeline*.

4 Projeto OpenAES

Neste capítulo, documenta-se os resultados do projeto *openAES*. O *openAES* é um projeto *open source*, fruto deste trabalho, que disponibiliza um *IP Core*, denominado *aes_ip*, de um co-processador AES compatível com o protocolo AMBA APB e que faz uso da arquitetura descrita no Capítulo 3 deste trabalho. A essa arquitetura básica são adicionadas diversas funcionalidades, como suporte a DMA, geração de interrupções e possibilidade de suspensão de mensagens.

Através do repositório do projeto, acessível em (OPENAES...,), são disponibilizados:

- O RTL, em Verilog, do IP Core;
- Um ambiente de verificação funcional;
- Uma camada de abstração de hardware (HAL), escrita em C, compatível com o padrão ARM CMSIS;
- Um *script* de *timing constraints* no formato SDC;

O *IP Core* do projeto *openAES* é funcionalmente compatível com o acelerador AES presente na família de microcontroladores STM32L162xx (STMICROELECTRONICS, 2015).

Para propósito de validação, o IP foi prototipado em um dispositivo SmartFusion A2F200M3F (MICROSEMI, 2015). Todos os arquivos necessários para embarcá-lo nessa plataforma também são disponibilizados através do repositório do projeto.

4.1 Descrição Funcional

O *aes_ip* é um co-processador capaz de encriptar e decriptar blocos de 128 bits de dados segundo o algoritmo AES-128. Os modos de criptografia suportados são: ECB, CBC e CTR. Seus sinais de entrada e saída são descritos na Tabela 10.

Os dados a serem processados são transferidos para o *aes_ip* através de um *buffer* de entrada de 32 bits denominado *AES_DINR*. Os dados processados, por sua vez, são lidos através de um *buffer* de saída de 32 bits denominado *AES_DOUTR*. Como esses *buffers* possuem apenas 32 bits, são necessários quatro acessos para que todos os 128 bits de dados sejam transferidos. O primeiro acesso, seja ele de leitura ou escrita, deve corresponder aos 32 bits mais significativos da palavra de 128 bits. Internamente, o *aes_ip*

Sinal	**Direção**	**Largura (em bits)**	**Descrição**
int_ccf	Saída	1	Interrupção que indica o fim do processamento do *aes_ip*
int_err	Saída	1	Interrupção de Erro
dma_req_wr	Saída	1	*Request* de escrita para o controlador DMA
dma_req_rd	Saída	1	*Request* de leitura para o controlador DMA
PREADY	Saída	1	Sinais AMBA APB (ARM, 2008)
PSLVERR	Saída	1	
PRDATA	Saída	32	
PADDR	Entrada	32	
PWDATA	Entrada	32	
PWRITE	Entrada	1	
PENABLE	Entrada	1	
PSEL	Entrada	32	
PCLK	Entrada	1	
PRESETn	Entrada	1	

Tabela 10 – Interface do Co-processador *aes_ip*

organiza os dados de maneira *little-endian*. Para facilitar a integração com sistemas que utilizem outros formatos de organização de dados, o *IP* possui uma unidade de formatação que é controlada pelos bits *DATATYPE* do registro de configuração *AES_CR*. Os formatos suportados são ilustrados na Figura 36.

O acesso à chave de criptografia é feito através dos quatro registros *AES_KEYRx* cuja organização é *little-endian*, i.e, os 32 bits mais significativos da chave devem ser escritos no registro *AES_KEYR3*. Não há flexibilidade de formatação de dados para esses registros, de modo que os bits *DATATYPE* não os afetam.

O vetor de inicialização utilizado nos modos CBC e CTR é acessado através dos quatro registradores *AES_IVRx*. O contador utilizado no modo CTR corresponde ao registro *AES_IVR0*, que é automaticamente incrementado pelo *hardware* do co-processador.

O *aes_ip* suporta os quatro modos de operação descritos na seção 3.4.1, que são selecionados através dos bits *MODE* do registro de configuração *AES_CR*. O modo de criptografia, por sua vez, é selecionado pelos bits *CHMODE* desse mesmo registro.

O IP também suporta transferência de dados por meio de uma interface DMA. Para esse fim, são disponibilizados dois canais de *request*. Quando o bit *DMAINEN* do registro *AES_CR* é habilitado, o *aes_ip* automaticamente gera quatro *requests* para que o controlador DMA realize as quatro escritas para o registro *AES_DINR*. Quando o bit *DMAOUTEN* do registro *AES_CR* é habilitado, são gerados quatro *requests* para que o controlador DMA realize as leituras do registro *AES_DOUTR*. Ao habilitar ambos esses bits, toda a transferência de dados pode ser feita sem a intervenção da CPU. Uma vez habilitados, os *requests* DMA são gerados até que o *IP* seja desabilitado, o que é feito através do bit *EN* do registro *AES_CR*.

É possível suspender e resumir a operação do *aes_ip* quando os modos CBC e CTR são selecionados. Essa funcionalidade é útil caso se deseje implementar priorização

de mensagens. A Figura 37 ilustra esse mecanismo. É importante destacar que o contexto do co-processador, dado pelos registros *AES_IVRx* e os 128 bits de dados processados, deve ser salvo pelo *software* de aplicação.

É possível detectar o fim da computação realizada pelo *aes_ip* tanto por espera ocupada quanto por interrupção. No primeiro caso, basta que o *software* de aplicação monitore o estado da flag CCF(do inglês *Computation Complete Flag*) do registro de estado *AES_SR*. Essa flag, uma vez ativada, deve ser desativada pelo *software* através de uma escrita para o bit CCFC (do inglês *Computation Complete Flag Clear*) do registro *AES_CR*. No segundo caso, basta que se habilite a geração de interrupção por parte do co-processador, o que é feito através do bit CCFIE (do inglês *CCF Flag Interrupt Enable*).

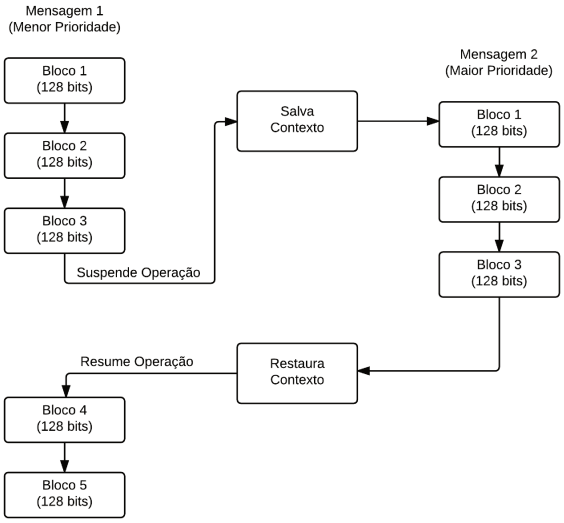


Figura 37 – Ilustração do Mecanismo de Interrupção de Mensagens

O mapa de memória do *aes_ip* e a descrição de cada um de seus registradores encontram-se no apêndice B.

4.2 RTL

O *aes_ip* foi descrito utilizando a linguagem de descrição de *hardware* Verilog. Na pasta *src* do diretório principal do projeto, há duas subpastas, *include* e *rtl*, que contêm os arquivos fontes, em Verilog, do *IP Core*. A Tabela 11 descreve, brevemente, cada um desses arquivos.

4.3 Ambiente de Verificação

Para validar a corretude funcional do RTL do *aes_ip*, foi desenvolvido um ambiente de verificação funcional cuja arquitetura é ilustrada na Figura 38. Esse ambiente

Arquivo	**Pasta**	**Descrição**
control_unit_params.vh	include	Contém os parâmetros utilizados pela unidade de conrole
host_interface.vh	include	Contém os parâmetros utilizados pela interface externa
sbox_functions.vf	include	Contém funções necessárias para a implementação da *SBox*
aes_ip.v	rtl	Contém o módulo principal do *aes_ip*
aes_core.v	rtl	Contém o módulo principal do *AES-128*
host_interface.v	rtl	Contém o circuito que implementa a interface externa do *IP*
data_swap.v	rtl	Contém o circuito que implementa a unidade de formatação de dados
control_unit.v	rtl	Contém a máquina de estados finitos do *aes_ip*
datapath.v	rtl	Contém o datapath do *aes_ip*
key_expander.v	rtl	Contém o circuito que implementa o processo de derivação de chave
mix_columns.v	rtl	Contém o circuito que implementa a *layer MixColumns()* direta e inversa
SBox.v	rtl	Instancia quatro SBox de 8 bits para formar uma SBox de 32 bits
SBox_8.v	rtl	Contém o circuito que implementa a SBox de 8 bits
shift_rows.v	rtl	Contém o circuito que implementa a *layer ShiftRows()* direta e inversa

Tabela 11 – Interface do Co-processador *aes_ip*

de verificação foi escrito nas linguagens Verilog e C, que foram integradas por meio da interface vpi (SUTHERLAND, 1999).

Como modelo funcional de referência, utilizou-se a biblioteca de criptografia do projeto OpenSSl (OPENSSL,).

Como estímulos, foram utilizados tanto casos de teste direcionados, quanto casos de teste aleatórios. Para os casos de teste direcionados, utilizou-se os vetores de teste disponibilizados pela agência americana NIST (THE. . . , 2002). No total, há 816 cenários de testes cujo objetivo é exercitar as principais funcionalidades do *IP Core*.

4.4 Síntese Lógica

O *aes_ip* foi prototipado em um dispositivo SmartFusion A2F200M3F (MICROSEMI, 2015). O primeiro passo desse processo é a síntese lógica cujo propósito é transformar o RTL em um *netlist*. Como ferramenta de síntese, utilizou-se o *software* Synplify Pro (SYNOPSIS,).

É bem conhecido que o estilo de codificação dos estados de uma máquina de estados finitos influencia, durante a síntese, a performance do circuito, tanto em termos de área, quanto de potência e frequência de operação. Durante a codificação do *aes_ip*, no entanto, optou-se por utilizar uma codificação binária sem que se tenha feito qualquer exploração de arquitetura para validar essa escolha. A razão para isso é que as ferramentas de síntese lógica atuais possuem a funcionalidade de, elas mesmas, realizarem essa etapa de exploração de arquitetura de forma automática, bastando, para isso, que sejam configuradas para tal propósito.

Como já mencionado no Capítulo 2 deste trabalho, o desbalanceamento dos estágios de *pipeline* afeta diretamente o ganho de performance que se consegue com a utilização

dessa técnica. Assim, durante o desenvolvimento do RTL do *aes_ip*, foram realizadas diversas pré-sínteses com o intuito de se atingir um *pipeline* o mais balanceado possível. Mesmo com todo esse esforço para se inserir os registros de *pipeline* nos pontos adequados do circuito, é comum que haja uma pequena diferença residual entre os atrasos do estágio. Uma técnica que ajuda a reduzir essa diferença é o *retiming* (PARHI, 1999), também suportado pelas ferramentas de síntese atuais.

A síntese foi realizada na condição de operação *COMWC* (do inglês, *Commercial*, *Worst Case*). Para o dispositivo em questão, a condição de operação comercial é sumarizada na Tabela 12. Para a análise de pior caso, se utiliza a situação em que o dispositivo é mais lento, o que corresponde à maior temperatura e às menores tensões de operação.

	Mínimo	**Típico**	**Máximo**
Temperatura (C)	0	25	85
Tensão do *Core* (V)	1.425	1.5	1.575
Tensão de IO 1	1.4	1.5	1.6
Tensão de IO 2	1.7	1.8	1.9
Tensão de IO 3	2.3	2.5	2.7
Tensão de IO 4	3	3.3	3.6

Tabela 12 – Condição de Operação *COMWC*

Os relatórios finais da síntese do *aes_ip* em um dispositivo SmartFusion A2F200M3 são sumarizados nas Tabelas 13, 14, 15 e 16.

Os resultados de área são apresentados em unidades GE (do inglês, *Gate Equivalent*) para facilitar a comparação com arquiteturas implementadas em outras tecnologias. Como em (MORIOKA; SATOH, 2002), considera-se, neste trabalho, que um *flip-flop* equivale a 6.75 GEs. É importante destacar que os resultados de área aqui reportados podem estar ligeiramente pessimistas. Isso ocorre porque, durante o cálculo do número de GEs do projeto, considerou-se que cada multiplexador de duas entradas e cada porta lógica complexa de dois elementos, como AO (AND-OR) e OA (OR-AND), equivalem a 2 GEs. Por outro lado, para a ferramenta de síntese em FPGA, o custo em termos de área dessas portas lógicas é o mesmo. Como consequência, o sintetizador lógico as utiliza de forma mais ou menos indistinguível (do ponto de vista de área), o que pode implicar em um uso excessivo desse tipo elemento, penalizando, assim, o GE total do projeto.

O consumo de potência reportado na Tabela 16 foi estimado através do *software SmartPower* (MICROSEMI,) utilizando a condição de operação 'Típica' (veja Tabela 12). Para a estimativa da atividade de chaveamento, utilizou-se um arquivo VCD (do inglês, *Value Change Dump*) gerado pelo ambiente de verificação em conjunto com o modo de operação *vectorless analysis* da ferramenta *SmartPower*.

Módulo	**Lógica (GE)**	**Flip-flops**	**Total (GE)**	**%**
Host Interface	309	63	734	6.73
Datapath	3907	884	9874	90.54
Unidade de Controle	163	20	298	2.73
Total	4379	967	10906	100

Tabela 13 – Relatório da Síntese do *aes_ip*

Módulo	**Lógica (GE)**	**Flip-flops**	**Total (GE)**	**%**
SBox	501	48	825	8.35
MixColumns	169	0	169	1.71
Geração de Chave	222	0	222	2.73
Unidade de Swap	192	0	192	1.94
Outros	2823	836	8466	85.27
Total	3907	884	9874	100

Tabela 14 – Relatório da Síntese do *Datapath* do *aes_ip*

Parâmetro	
Frequência Máxima	69.421 MHz
Ciclos de *Clock*	64
Throughput	138.84 Mb/s

Tabela 15 – Performance do *aes_ip*

Módulo	**Lógica (mW)**	**Interconexão (mW)**	**Total (mW)**	**%**
Host Interface	0.307	0.007	0.314	7.27
Datapath	3.709	0.19	3.899	90.34
Unidade de Controle	0.099	0.004	0.103	2.39
Total	4.115	0.201	4.316	100

Tabela 16 – Consumo de Potência do *aes_ip*

4.5 Camada de Abstração de Hardware

O projeto *openAES* também disponibiliza uma camada de abstração de hardware (HAL) cujo intuito é facilitar o desenvolvimento de *software* por parte dos desenvolvedores de aplicações.

A HAL do projeto, escrita em linguagem C, foi desenvolvida para microcontroladores ARM da família Cortex, mas pode ser adaptada para outras famílias de microcontroladores. A HAL desenvolvida segue o padrão ARM CMSIS (ARM, b) e se encontra na pasta *drivers* do repositório do projeto.

4.6 Validação do *IP Core*

Para propósitos de validação, prototipou-se o *IP Core* desenvolvido em um dispositivo SmartFusion A2F200M3F (MICROSEMI, 2015).

Antes de realizar os testes em *hardware*, no entanto, optou-se por validar, por meio de simulação, a integração do *IP Core* com o processador ARM Cortex-M3 presente nos dispositivos SmartFusion. Para isso, desenvolveu-se um *script* de teste que faz uso dos modelos BFM dos periféricos presentes nesses dispositivos. Com esse *script*, por exemplo, é possível validar que os registros do *aes_ip* estão corretamente mapeados na memória do SoC (do inglês, *System on a Chip*).

A aplicação desenvolvida para validar o *aes_ip* em *hardware* foi baseada nos testes descritos na seção 4.3. Esses testes foram modificados para fazer uso do *driver* descrito na seção 4.5. Com isso, foi possível realizar os mesmos testes funcionais utilizados na simulação, mas agora em um ambiente de execução real.

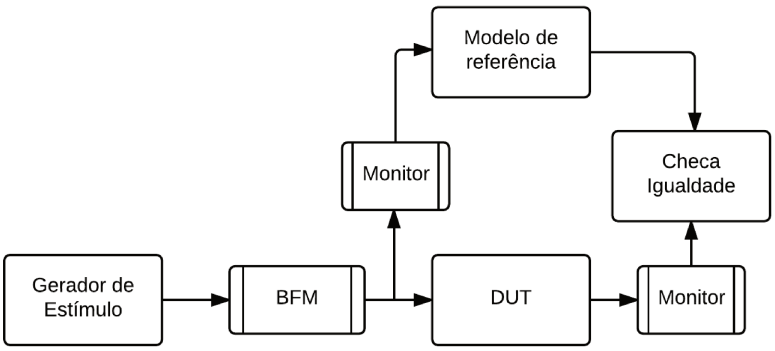


Figura 38 – Arquitetura do Ambiente de Verificação Funcional

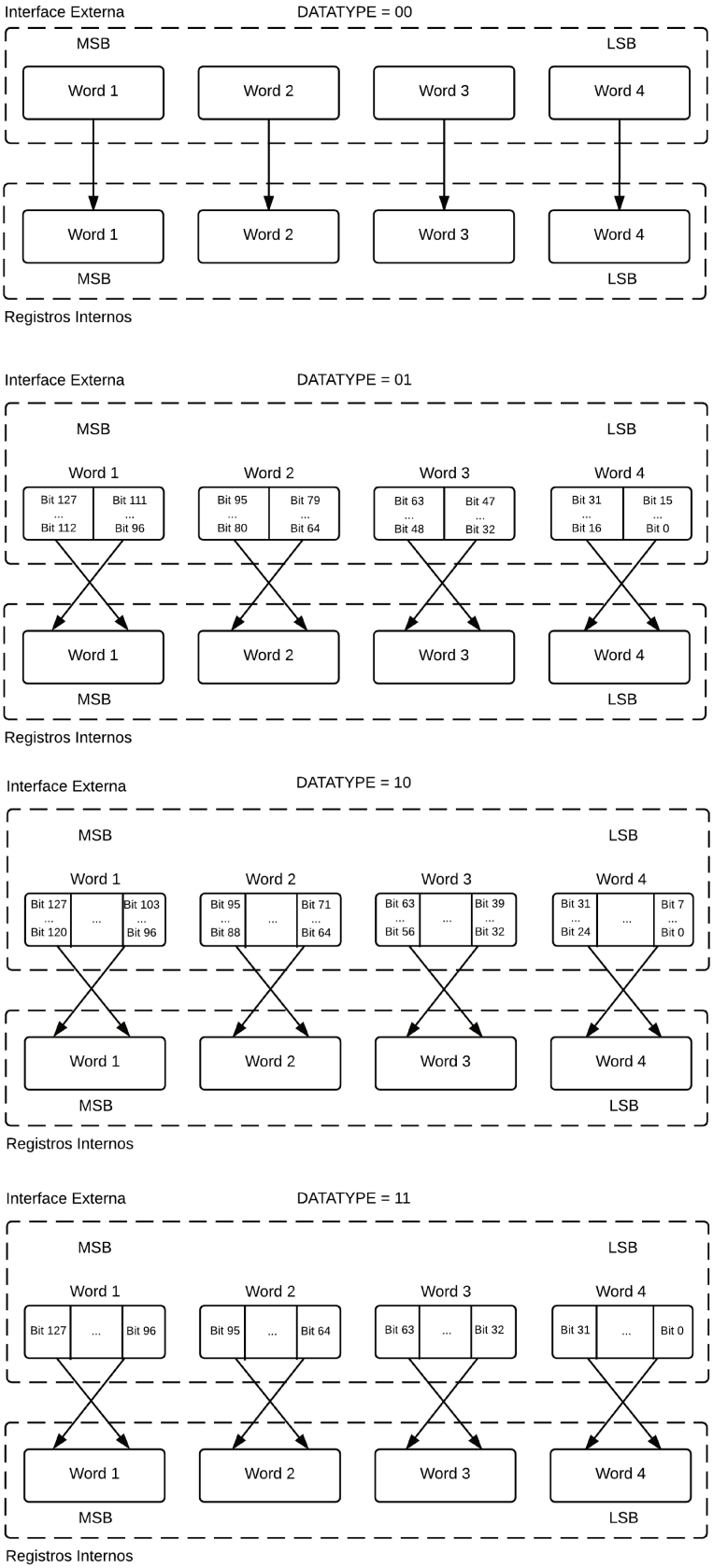


Figura 36 – Formatos de dados suportados

5 Considerações Finais

Grande parte da literatura dedicada ao estudo de implementações em *hardware* do algoritmo AES se concentra ou em arquiteturas de alto *throughput* ou em arquiteturas compactas, havendo, proporcionalmente, poucos trabalhos dedicados a arquiteturas situadas no meio do espaço de projeto. Por outro lado, há uma série de aplicações práticas que requerem soluções de *hardware* cujo projeto seja um compromisso entre taxa de processamento e recursos lógicos utilizados. Assim, a escassez de trabalhos de pesquisa que explorem arquiteturas situadas no meio do espaço de projeto, aliada a necessidade potencial desse tipo de arquitetura em aplicações práticas constituíram a motivação do presente trabalho.

No Capítulo 3, propôs-se uma arquitetura de *hardware* para um co-processador capaz de realizar encriptação e decriptação segundo o padrão AES-128. Inicialmente, foi apresentada uma arquitetura que suporta o modo de encriptação sem *feedback* ECB. Em seguida, discutiu-se como essa arquitetura básica pode ser estendida para suportar os modos de encriptação com *feedback* CBC e CTR. A arquitetura proposta empregou as técnica de *loop rolling* (com compartilhamento de SBox) e *subpipelining*. Nesse capítulo também foi estudado como o ganho teórico de desempenho do *pipeline* se comporta quando o número de estágios de uma arquitetura iterativa com *datapath* de 32 bits cresce. Foi mostrado que esse ganho de desempenho possui, do ponto de vista prático, um limite superior. Também foi estudada a sensibilidade desse tipo de arquitetura ao desbalanceamento entre os estágios do *pipeline*. Dos resultados expostos nesse capítulo, ficou claro que o uso agressivo de *pipeline* em arquiteturas iterativas com *datapath* de 32 bits é desaconselhado. Também mostrou-se que a operação de derivação de chave pode ser otimizada para arquiteturas com *datapath* de 32 bits. e que é possível compartilhar o uso da operação *InvMixColumns()*

No Capítulo 4, documentou-se os resultados do projeto *openAES*. O *openAES* é um projeto *open source* que disponibiliza um *IP Core*, denominado *aes_ip*, de um co-processador AES compatível com o protocolo AMBA APB e que faz uso da arquitetura descrita no Capítulo 3 deste trabalho. A essa arquitetura básica foram adicionadas diversas funcionalidades, como suporte a DMA, geração de interrupções e possibilidade de suspensão de mensagens. Como resultados do projeto, foram disponibilizados: o RTL, em Verilog, do IP Core, um ambiente de verificação funcional, uma camada de abstração de hardware (HAL), escrita em C, compatível com o padrão ARM CMSIS e um *script* de *timing constraints* no formato SDC. Como forma de validação, o IP foi prototipado em um dispositivo SmartFusion A2F200M3F.

5.1 Trabalhos Futuros

O presente trabalho pode ser estendido em diversas direções. Aqui, pontua-se algumas delas:

- Embora prototipado em uma FPGA, o IP Core do projeto openAES foi desenvolvido para ser utilizado em ASIC. Essa é uma das razões pelas quais a *SBox* foi implementada utilizando-se apenas recursos lógicos. Uma continuação natural desse trabalho, portanto, é sintetizar o co-processador desenvolvido segundo o fluxo de projeto de circuitos integrados digitais;

- Da forma como a *SBox* foi implementada neste trabalho, o IP Core desenvolvido é vulnerável a ataques do tipo SCA. Em trabalhos futuros, pretende-se implementar as técnicas de mascaramento de *SBox* descritas em (CANRIGHT; BATINA, 2008) com o intuito de aumentar a robustez da solução proposta;

- No Capítulo 3, foi mostrado que arquiteturas iterativas com *subpipelining* de ordem maior que ou igual a 3 requerem a inserção de ciclos de NOP para evitar a ocorrência de *hazards*. Durante esses ciclos de NOP, o co-processador não realiza processamento algum, o que, claramente, é um desperdício de recursos. Ao mesmo tempo, o processo de derivação da chave de decriptação requer 10 ciclos de *clock* para ser concluído. Como trabalho futuro, pode-se explorar a possibilidade de realizar o processo de derivação de chave durante os ciclos de NOP da encriptação. Com isso, seria possível, em princípio, reduzir a latência da decriptação e minimizar os períodos de inatividade de co-processador.

Referências

AHSON, S.; ILYAS, M. *WiMAX: standards and security*. 1st. ed. [S.l.]: CRC Press, 2008. ISBN 9781420045239. Citado na página 14.

ARM. *ARM HOLDINGS PLC REPORTS RESULTS FOR THE FIRST QUARTER 2016*. Disponível em: <http://www.investegate.co.uk/arm-holdings-plc--arm-/rns/reports-results-for-the-first-quarter-2016/20160420070007 7103V/>. Acesso em: 24/08/2016. Citado na página 16.

ARM. *CMSIS-Cortex Microcontroller Software Interface Standard*. Disponível em: <http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php>. Acesso em: 22/04/2016. Citado na página 68.

ARM. *AMBA 3 APB Protocol Especification*. 2008. Disponível em: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024b/index.html>. Acesso em: 21/04/2016. Citado na página 64.

BOGDANOV, A.; KHOVRATOVICH, D.; RECHBERGER, C. Advances in cryptology – asiacrypt 2011: 17th international conference on the theory and application of cryptology and information security, seoul, south korea, december 4-8, 2011. proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. cap. Biclique Cryptanalysis of the Full AES, p. 344–371. ISBN 978-3-642-25385-0. Disponível em: <http://dx.doi.org/10.1007/978-3-642-25385-0_19>. Citado na página 16.

CANRIGHT, D. Cryptographic hardware and embedded systems – ches 2005: 7th international workshop, edinburgh, uk, august 29 – september 1, 2005. proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. cap. A Very Compact S-Box for AES, p. 441–455. ISBN 978-3-540-31940-5. Disponível em: <http://dx.doi.org/10.1007/11545262_32>. Citado 4 vezes nas páginas 15, 34, 43 e 61.

CANRIGHT, D.; BATINA, L. Applied cryptography and network security: 6th international conference, acns 2008, new york, ny, usa, june 3-6, 2008. proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. cap. A Very Compact "Perfectly Masked" S-Box for AES, p. 446–459. ISBN 978-3-540-68914-0. Disponível em: <http://dx.doi.org/10.1007/978-3-540-68914-0_27>. Citado 2 vezes nas páginas 16 e 72.

DAEMEN, J.; RIJMEN, V. *The Design of Rijndael*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2002. ISBN 3540425802. Citado na página 19.

DIERKS, T.; RESCORLA, E. *The Transport Layer Security (TLS) Protocol Version 1.2*. [S.l.], 2008. Disponível em: <http://www.rfc-editor.org/rfc/rfc5246.txt>. Citado na página 14.

FAN, C.-P.; HWANG, J.-K. Implementations of high throughput sequential and fully pipelined aes processors on fpga. In: *Intelligent Signal Processing and Communication Systems, 2007. ISPACS 2007. International Symposium on*. [S.l.: s.n.], 2007. p. 353–356. Citado 3 vezes nas páginas 15, 36 e 46.

FELDHOFER, M.; WOLKERSTORFER, J. Strong crypto for rfid tags - a comparison of low-power hardware implementations. In: *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. [S.l.: s.n.], 2007. p. 1839–1842. Citado 2 vezes nas páginas 15 e 26.

FELDHOFER, M.; WOLKERSTORFER, J.; RIJMEN, V. Aes implementation on a grain of sand. *Information Security, IEE Proceedings*, v. 152, n. 1, p. 13–20, Oct 2005. ISSN 1747-0722. Citado 2 vezes nas páginas 15 e 36.

FIPS PUB 197, Advanced Encryption Standard (AES). 2001. U.S.Department of Commerce/National Institute of Standards and Technology. Citado 4 vezes nas páginas 14, 15, 19 e 21.

FISCHER, V.; DRUTAROVSKY, M.; CHODOWIEC, P.; GRAMAIN, F. Invmixcolumn decomposition and multilevel resource sharing in aes implementations. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 13, n. 8, p. 989–992, Aug 2005. ISSN 1063-8210. Citado 4 vezes nas páginas 15, 31, 33 e 42.

GAJ, K. Very compact fpga implementation of the aes algorithm. In: *Proceedings of 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES), number 2779 in Lecture Notes in Computer Science*. [S.l.]: Springer-Verlag, 2003. p. 319–333. Citado 2 vezes nas páginas 15 e 36.

GAJ, K.; CHODOWIEC, P. Cryptographic engineering. In: _____. Boston, MA: Springer US, 2009. cap. FPGA and ASIC Implementations of AES, p. 235–294. ISBN 978-0-387-71817-0. Disponível em: <http://dx.doi.org/10.1007/978-0-387-71817-0_10>. Citado 3 vezes nas páginas 32, 33 e 34.

HAMALAINEN, P.; ALHO, T.; HANNIKAINEN, M.; HAMALAINEN, T. Design and implementation of low-area and low-power aes encryption hardware core. In: *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*. [S.l.: s.n.], 2006. p. 577–583. Citado 2 vezes nas páginas 15 e 36.

HSIAO, S.-F.; CHEN, M.-C.; TU, C.-S. Memory-free low-cost designs of advanced encryption standard using common subexpression elimination for subfunctions in transformations. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, v. 53, n. 3, p. 615–626, March 2006. ISSN 1549-8328. Citado 2 vezes nas páginas 15 e 36.

IEEE; 802.11I. Ieee standard for information technology-telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements-part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications: Amendment 6: Medium access control (mac) security enhancements. *IEEE Std 802.11i-2004*, p. 1–190, July 2004. Citado na página 14.

IEEE Draft Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE P802.11-REVma/D89.0*, p. 1–1230, July 2010. Citado na página 14.

IEEE Standard for Local and metropolitan area networks– Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 2: Active Radio Frequency

Identification (RFID) System Physical Layer (PHY). *IEEE Std 802.15.4f-2012 (Amendment to IEEE Std 802.15.4-2011)*, p. 1–72, April 2012. Citado na página 14.

IYER, N.; ANANDMOHAN, P.; POORNAIAH, D.; KULKARNI, V. High throughput, low cost, fully pipelined architecture for aes crypto chip. In: *India Conference, 2006 Annual IEEE*. [S.l.: s.n.], 2006. p. 1–6. Citado 2 vezes nas páginas 15 e 36.

JäRVINEN, K. U.; TOMMISKA, M. T.; SKYTTä, J. O. A fully pipelined memoryless 17.8 gbps aes-128 encryptor. In: *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2003. (FPGA '03), p. 207–215. ISBN 1-58113-651-X. Disponível em: <http://doi.acm.org/10.1145/611817.611848>. Citado 2 vezes nas páginas 15 e 36.

KAESLIN, H. *Top-Down Digital VLSI Design: From Architectures to Gate-Level Circuits and FPGAs, capítulo 1, página 4*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014. ISBN 0128007303, 9780128007303. Citado na página 60.

KENT, S. *IP Encapsulating Security Payload (ESP)*. [S.l.], 2005. Disponível em: <http://www.rfc-editor.org/rfc/rfc4303.txt>. Citado na página 14.

KOTTURI, D.; YOO, S.-M.; BLIZZARD, J. Aes crypto chip utilizing high-speed parallel pipelined architecture. In: *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. [S.l.: s.n.], 2005. p. 4653–4656 Vol. 5. Citado 2 vezes nas páginas 15 e 36.

KUO, H.; VERBAUWHEDE, I. Cryptographic hardware and embedded systems — ches 2001: Third international workshop paris, france, may 14–16, 2001 proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. cap. Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm, p. 51–64. ISBN 978-3-540-44709-2. Disponível em: <http://dx.doi.org/10.1007/3-540-44709-1_6>. Citado na página 15.

KUON, I.; ROSE, J. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 26, n. 2, p. 203–215, Feb 2007. ISSN 0278-0070. Citado na página 62.

LU, C.-C.; TSENG, S.-Y. Integrated design of aes (advanced encryption standard) encrypter and decrypter. In: *Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on*. [S.l.: s.n.], 2002. p. 277–285. ISSN 2160-0511. Citado na página 31.

LUTZ, A. K.; TREICHLER, J.; GüRKAYNAK, F. K.; KAESLIN, H.; BASLER, G.; ERNI, A.; REICHMUTH, S.; ROMMENS, P.; OETIKER, S.; FICHTNER, W. 2gbit/s hardware realizations of rijndael and serpent: A comparative analysis. In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. [S.l.]: Springer, 2002. (Lecture Notes in Computer Science, v. 2523), p. 144–158. Citado 2 vezes nas páginas 15 e 36.

MANGARD, S.; AIGNER, M.; DOMINIKUS, S. A highly regular and scalable aes hardware architecture. *Computers, IEEE Transactions on*, v. 52, n. 4, p. 483–491, April 2003. ISSN 0018-9340. Citado na página 14.

MAYER, U.; OELSNER, C.; KOHLER, T. Evaluation of different rijndael implementations for high end servers. In: *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*. [S.l.: s.n.], 2002. v. 2, p. II–348–II–351 vol.2. Citado 2 vezes nas páginas 15 e 36.

MENTENS, N.; BATINA, L.; PRENEEL, B.; VERBAUWHEDE, I. Topics in cryptology – ct-rsa 2005: The cryptographers' track at the rsa conference 2005, san francisco, ca, usa, february 14-18, 2005. proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. cap. A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box, p. 323–333. ISBN 978-3-540-30574-3. Disponível em: <http://dx.doi.org/10.1007/978-3-540-30574-3_22>. Citado na página 15.

MICROSEMI. *SmartPower v9.1: User's Guide*. Disponível em: <https://www.synopsys.com/tools/implementation/fpgaimplementation/pages/synplify-pro.aspx>. Acesso em: 06/05/2016. Citado na página 67.

MICROSEMI. *SmartFusion Datasheet*. 2015. Disponível em: <http://www.microsemi.com/document-portal/doc_view/130719-smartfusion-customizable-system-on-chip-csoc-datasheet>. Acesso em: 19/02/2016. Citado 6 vezes nas páginas 18, 60, 62, 63, 66 e 69.

MICROSOFT. *Encrypting File System*. 2016. Disponível em: <http://technet.microsoft.com/library/Cc962122>. Acesso em: 17/02/2016. Citado na página 14.

MORIOKA, S.; SATOH, A. A 10 gbps full-aes crypto design with a twisted-bdd s-box architecture. In: *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*. [S.l.: s.n.], 2002. p. 98–103. ISSN 1063-6404. Citado 8 vezes nas páginas 8, 15, 36, 37, 60, 61, 62 e 67.

MORIOKA, S.; SATOH, A. Cryptographic hardware and embedded systems - ches 2002: 4th international workshop redwood shores, ca, usa, august 13–15, 2002 revised papers. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. cap. An Optimized S-Box Circuit Architecture for Low Power AES Design, p. 172–186. ISBN 978-3-540-36400-9. Disponível em: <http://dx.doi.org/10.1007/3-540-36400-5_14>. Citado 2 vezes nas páginas 15 e 16.

MOZAFFARI-KERMANI, M.; REYHANI-MASOLEH, A. Efficient and high-performance parallel hardware architectures for the aes-gcm. *Computers, IEEE Transactions on*, v. 61, n. 8, p. 1165–1178, Aug 2012. ISSN 0018-9340. Citado 2 vezes nas páginas 15 e 36.

NEDJAH, N.; MOURELLE, L.; CARDOSO, M. A compact piplined hardware implementation of the aes-128 cipher. In: *Information Technology: New Generations, 2006. ITNG 2006. Third International Conference on*. [S.l.: s.n.], 2006. p. 216–221. Citado 2 vezes nas páginas 15 e 36.

OPENAES Repositório do projeto. Disponível em: <https://github.com/julioamerico/openAES>. Citado 2 vezes nas páginas 17 e 63.

OPENSSL. *OpenSSl: Cryptography and SSL/TLS Toolkit*. Disponível em: <https://www.openssl.org/>. Acesso em: 21/04/2016. Citado na página 66.

PAAR, C.; PELZL, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 3642041000, 9783642041006. Citado 3 vezes nas páginas 14, 19 e 57.

PARHI, K. K. *VLSI digital signal processing systems : design and implementation*. New York: Wiley, 1999. A Wiley-Interscience publication. ISBN 0-471-24186-5. Disponível em: <http://opac.inria.fr/record=b1097682>. Citado na página 67.

RIJMEN, V. *Efficient Implementation of a Rijndael S-box*. Disponível em: <http://www.researchgate.net/publication/2621085_Efficient_Implementation_of_the_Rijndael_S-box>. Acesso em: 07/03/2016. Citado na página 34.

RODRIGUEZ-HENRIQUEZ, F.; SAQIB, N. A.; DIAZ-PEREZ, A. 4.2 gbit/s single-chip fpga implementation of aes algorithm. *Electronics Letters*, v. 39, n. 15, p. 1115–1116, July 2003. ISSN 0013-5194. Citado na página 31.

RUDRA, A.; DUBEY, P. K.; JUTLA, C. S.; KUMAR, V.; RAO, J. R.; ROHATGI, P. Cryptographic hardware and embedded systems — ches 2001: Third international workshop paris, france, may 14–16, 2001 proceedings. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. cap. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic, p. 171–184. ISBN 978-3-540-44709-2. Disponível em: <http://dx.doi.org/10.1007/3-540-44709-1_16>. Citado na página 15.

SATOH, A.; MORIOKA, S.; TAKANO, K.; MUNETOH, S. A compact rijndael hardware architecture with s-box optimization. In: . [S.l.]: Springer-Verlag, 2001. p. 239–254. Citado 7 vezes nas páginas 15, 26, 31, 34, 36, 43 e 46.

SHAN, W.; FU, X.; XU, Z. A secure reconfigurable crypto ic with countermeasures against spa, dpa, and ema. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 34, n. 7, p. 1201–1205, July 2015. ISSN 0278-0070. Citado na página 16.

STMICROELECTRONICS. *RM0038 Reference Manual*. 2015. Disponível em: <http://www2.st.com/content/ccc/resource/technical/document/reference_manual/cc/f9/93/b2/f0/82/42/57/CD00240193.pdf/files/CD00240193.pdf/jcr:content/translations/en.CD00240193.pdf>. Acesso em: 18/04/2016. Citado 2 vezes nas páginas 16 e 63.

SUTHERLAND, S. (Ed.). *The VERILOG PLI Handbook: A User's Guide and Comprehensive Reference on the VERILOG Programming Language Interface*. 1st. ed. Norwell, MA, USA: Kluwer Academic Publishers, 1999. ISBN 079238489X. Citado na página 66.

SYNOPSIS. *Synplify Pro: Logic Synthesis for FPGA Design*. Disponível em: <https://www.synopsys.com/tools/implementation/fpgaimplementation/pages/synplify-pro.aspx>. Acesso em: 06/05/2016. Citado na página 66.

THE Advanced Encryption Standard Algorith Validation Suite (AESAVS). 2002. U.S.Department of Commerce/National Institute of Standards and Technology. Citado na página 66.

TIRI, K.; HWANG, D.; HODJAT, A.; LAI, B.-C.; YANG, S.; SCHAUMONT, P.; VERBAUWHEDE, I. Aes-based cryptographic and biometric security coprocessor ic in 0.18- mu;m cmos resistant to side-channel power analysis attacks. In: *VLSI Circuits,*

2005. Digest of Technical Papers. 2005 Symposium on. [S.l.: s.n.], 2005. p. 216–219. Citado na página 16.

VERBAUWHEDE, I.; SCHAUMONT, P.; KUO, H. Design and performance testing of a 2.29-gb/s rijndael processor. *Solid-State Circuits, IEEE Journal of*, v. 38, n. 3, p. 569–572, Mar 2003. ISSN 0018-9200. Citado 2 vezes nas páginas 15 e 36.

WU, F.; WANG, L.; WAN, J. A low cost and inner-round pipelined design of ecb-aes-256 crypto engine for solid state disk. In: *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*. [S.l.: s.n.], 2010. p. 485–491. Citado 3 vezes nas páginas 14, 16 e 37.

YLONEN, T.; LONVICK, C. *The Secure Shell (SSH) Protocol Architecture*. [S.l.], 2006. Disponível em: <http://www.rfc-editor.org/rfc/rfc4251.txt>. Citado na página 14.

ZHANG, X.; PARHI, K. Implementation approaches for the advanced encryption standard algorithm. *Circuits and Systems Magazine, IEEE*, v. 2, n. 4, p. 24–46, Fourth 2002. ISSN 1531-636X. Citado na página 22.

ZHANG, X.; PARHI, K. High-speed vlsi architectures for the aes algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 12, n. 9, p. 957–967, Sept 2004. ISSN 1063-8210. Citado 5 vezes nas páginas 15, 31, 34, 36 e 46.

ZIGBEE Specification Version 1.0. [S.l.], 2004. Citado na página 14.

Apêndices

APÊNDICE A – Geração dos sinais de controle da MEF do Co-processador AES-128

Neste apêndice, indica-se quais sinais de controle são acionados em cada estado da MEF associada à unidade de controle do co-processador AES-128.

A.1 Sinais de Controle para o modo ECB

A geração dos sinais de controle associados à máquina de estados finitos descrita na seção 3.4.3 é sumarizada na Tabela 17.

Estado	**Saídas**
IDLE	sbox_sel = COL_0 rk_sel = COL bypass_rk = DISABLE key_out_sel = KEY_0 col_sel = INPUT key_sel = KEY_HOST key_en = KEY_DIS col_en = COL_DIS rd_count_en = DISABLE bypass_key_en = DISABLE key_derivation_en = DISABLE end_comp = DISABLE
ROUND0_COL0	sbox_sel = COL_0 rk_sel = COL, bypass_rk = ENABLE, bypass_key_en = ENABLE, key_out_sel = KEY_0, col_sel = (enc_dec) ? ADD_RK_OUT : SHIFT_ROWS, col_en = (enc_dec) ? EN_COL_0 : COL_ALL

A.2 Sinais de Controle para o modo CBC e CTR

As modificações na unidade de controle necessárias para suportar os modos CBC e CTR são indicadas na Tabela 18, onde só se mostram as diferenças em relação à Tabela 17.

Estado	**Saídas**

ROUND0_COL1	sbox_sel = COL_1
	rk_sel = COL
	bypass_rk = ENABLE
	bypass_key_en = ENABLE
	key_out_sel = KEY_1
	col_sel = ADD_RK_OUT
	col_en = EN_COL_1
	if(!enc_dec)
	  key_sel = KEY_OUT
	  key_en = EN_KEY_1

ROUND0_COL2	sbox_sel = COL_2
	rk_sel = COL
	bypass_rk = ENABLE
	bypass_key_en = ENABLE
	key_out_sel = KEY_2
	col_sel = ADD_RK_OUT
	col_en = EN_COL_2
	if(!enc_dec)
	  key_sel = KEY_OUT
	  key_en = EN_KEY_2

ROUND0_COL3	sbox_sel = COL_3
	rk_sel = COL
	bypass_rk = ENABLE
	bypass_key_en = ENABLE
	key_out_sel = KEY_3
	col_sel = (enc_dec) ? SHIFT_ROWS : ADD_RK_OUT
	col_en = (enc_dec) ? COL_ALL : EN_COL_3
	if(!enc_dec)
	  key_sel = KEY_OUT
	  key_en = EN_KEY_3

ROUND_KEY0	sbox_sel = G_FUNCTION key_sel = KEY_OUT key_en = EN_KEY_0 rd_count_en = ENABLE

Estado	**Saídas**

ROUND_COL0	sbox_sel = COL_0
	rk_sel = (last_round) ? MIXCOL_IN : MIXCOL_OUT
	key_out_sel = KEY_0
	key_sel = KEY_OUT
	if(enc_dec)
	  key_en = EN_KEY_1
	if(!enc_dec)
	  col_sel = (last_round) ? ADD_RK_OUT : SHIFT_ROWS
	else
	  col_sel = ADD_RK_OUT
	if(enc_dec)
	  col_en = EN_COL_0
	else
	  col_en = (last_round) ? EN_COL_0 : COL_ALL

ROUND_COL1	sbox_sel = COL_1
	rk_sel = (last_round) ? MIXCOL_IN : MIXCOL_OUT
	key_out_sel = KEY_1
	key_sel = KEY_OUT
	if(enc_dec)
	  key_en = EN_KEY_2
	else
	  key_en = EN_KEY_1
	col_sel = ADD_RK_OUT
	col_en = EN_COL_1

ROUND_COL2	sbox_sel = COL_2
	rk_sel = (last_round) ? MIXCOL_IN : MIXCOL_OUT
	key_out_sel = KEY_2
	key_sel = KEY_OUT
	if(enc_dec)
	  key_en = EN_KEY_3
	else
	  key_en = EN_KEY_2
	col_sel = ADD_RK_OUT
	col_en = EN_COL_2

Estado	**Saídas**

ROUND_COL3	sbox_sel = COL_3
	rk_sel = (last_round) ? MIXCOL_IN : MIXCOL_OUT
	key_out_sel = KEY_3
	key_sel = KEY_OUT
	if(!enc_dec)
	  key_en = EN_KEY_3
	if(enc_dec)
	  col_sel = (last_round) ? ADD_RK_OUT : SHIFT_ROWS
	else
	  col_sel = ADD_RK_OUT
	if(enc_dec)
	  col_en = (last_round) ? EN_COL_3 : COL_ALL
	else
	  col_en = EN_COL_3

GEN_KEY0	sbox_sel = G_FUNCTION; rd_count_en = ENABLE
GEN_KEY1	key_en = EN_KEY_1 || EN_KEY_0 key_sel = KEY_OUT bypass_key_en = ENABLE
GEN_KEY2	key_en = EN_KEY_2 key_sel = KEY_OUT bypass_key_en = ENABLE
GEN_KEY3	key_en = EN_KEY_3 key_sel = KEY_OUT bypass_key_en = ENABLE
READY	end_comp = ENABLE if(op_mode == KEY_DERIVATION) key_derivation_en = ENABLE

Tabela 17 – Saídas da Máquina de Estados para o modo ECB

Estado	**Saídas**
IDLE	iv_cnt_en = DISABLE iv_cnt_sel = IV_BUS
ROUND_COL0	if((mode_cbc && last_round && !enc_dec) || (mode_ctr && last_round)) col_sel = INPUT
ROUND_COL1	if((mode_cbc && last_round && !enc_dec) || (mode_ctr && last_round)) col_sel = INPUT
ROUND_COL2	if((mode_cbc && last_round && !enc_dec) || (mode_ctr && last_round)) col_sel = INPUT

ROUND_COL3	if((mode_cbc && last_round && !enc_dec) || (mode_ctr && last_round))
	col_sel = INPUT
	if(mode_ctr && last_round)
	iv_cnt_en = ENABLE
	iv_cnt_sel = IV_CNT

Tabela 18 – Modificações nas Saídas da Máquina de Estados necessárias para suportar os modos CBC e CTR

APÊNDICE B – Mapa de Memória do Co-processador do Projeto OpenAES

Neste apêndice, detalha-se os registros que compõe o mapa de memória do co-processador do projeto OpenAES.

B.1 Mapa de Memória

O mapa de memória do *aes_ip* pode ser visto na Figura 39.

Offset	Registro	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0000	AES_CR	Reservado																						DMAOUTIE	DMAINEN	ERRIE	CCFIE	ERRC	CCFC	CHMODE	MODE	DATATYPE	EN
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0004	AES_SR	Reservado																													WRERR	RDERR	CCF
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0008	AES_DINR	AES_DINR[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x000C	AES_DOUTR	AES_DOUTR[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0010	AES_KEY0	AES_KEYR0[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0014	AES_KEY1	AES_KEYR1[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0018	AES_KEY2	AES_KEYR2[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x001C	AES_KEY3	AES_KEYR3[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0020	AES_IVR0	AES_IVR0[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0024	AES_IVR1	AES_IVR1[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0028	AES_IVR2	AES_IVR2[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x002C	AES_IVR3	AES_IVR3[31:0]																															
	Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 39 – Mapa de Memória

B.2 Descrição dos Registros do Mapa de Memória

Nas próximas sub-seções, descreve-se os registros que compõem o mapa de memória do co-processador do projeto.

B.2.1 Registro *AES_CR*

Endereço de offset: 0x00

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reservado															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reservado			DMAOUTEN	DMAINEN	ERRIE	CCFIE	ERRC	CCFC	CHMODE		MODE		DATATYPE		EN
r	r	r	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:13]:** Reservado. Lido como 0
- **Bit 12 - DMAOUTEN:** Habilita a leitura do registro *AES_DOUTR* pela interface DMA. Esse bit não tem efeito algum no modo de operação 2.
  0: DMA desabilitado
  1: DMA habilitado
- **Bit 11 - DMAINEN:** Habilita a escritaa do registro *AES_DINR* pela interface DMA. Esse bit não tem efeito algum no modo de operação 2.
  0: DMA desabilitado
  1: DMA habilitado
- **Bit 10 - ERRIE:** Habilita a geração de interrupção de erro.
  0: Desabilita Interrupção de Erro
  1: Habilita Interrupção de Erro
- **Bit 9 - CCFIE:** Habilita a geração de interrupção CCF.
  0: Desabilita Interrupção CCF
  1: Habilita Interrupção CCF
- **Bit 8 - ERRC:** Zera a *flag* de erro. Uma escrita para esse bit zera as *flags RDERR* e *WRERR*. Esse bit sempre é lido como 0.
- **Bit 7 - CCFC:** Zera a *flag* CCF. Uma escrita para esse bit zera a *flag CCF*. Esse bit sempre é lido como 0.

- **Bit [6:5] - CHMODE:** Seleciona o modo de criptografia utilizado.
  00: ECB (Default)
  01: CBC
  10: CTR
  11: Reservado

- **Bit [4:3] - MODE:** Seleciona o modo de operação utilizado.
  00: Modo 1: Encriptação (Default)
  01: Mode 2: Derivação de Chave
  10: Mode 3: Decriptação
  11: Modo 4: Derivação de chave seguida de Decriptação.

- **Bit [2:1] - DATATYPE:** Seleciona a formatação dos registros de entrada e saída.
  00: Nenhuma alteração
  01: Espelhamento de 16 bits
  10: Espelhamento de 8 bits
  11: Espelhamento bit a bit.

- **Bit 0 - EN:** Inicia a operação do algoritmo AES.
  0: Co-processador desabilitado
  1: Co-processador habilitado

B.2.2 Registro *AES_SR*

Endereço de offset: 0x04

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reservado															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reservado													WRERR	RDERR	CCF
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- **Bit [31:3]:** Reservado. Lido como 0

- **Bit 2 - WRERR:** *Flag* de erro de escrita. Essa *flag* é ativada sempre que se realiza uma escrita para os registros sem que o *aes_ip* esteja desabilitado.
  0: Não houve erro de escrita
  1: Houve erro de escrita.

- **Bit 1 - RDERR:** *Flag* de erro de leitura. Essa *flag* é ativada sempre que se realiza uma leitura dos registros (com exceção do registro *AES_SR*) sem que o *aes_ip* esteja desabilitado.
  0: Não houve erro de leitura
  1: Houve erro de leitura.

- **Bit 0 - CCF:** *Flag* que indica que a operação do *aes_ip* foi concluída.
  0: Computação em curso
  1: Computação concluída

B.2.3 Registro *AES_DINR*

Endereço de offset: 0x08

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DINR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DINR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_DINR:** *Buffer* de entrada por meio do qual os dados a serem processados são transferidos para o *aes_ip*.

B.2.4 Registro *AES_DOUTR*

Endereço de offset: 0x0C

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DOUTR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DOUTR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_DOUTR:** *Buffer* de saída por meio do qual os dados processados pelo *aes_ip* são lidos pela interface externa.

B.2.5 Registro *AES_KEY0*

Endereço de offset: 0x10

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEYR0[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEYR0[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	Rw

- **Bit [31:0] - AES_KEY0:** Esse registro armazena os 32 bits menos significativos da chave de criptografia.

B.2.6 Registro *AES_KEY1*

Endereço de offset: 0x14

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEYR1[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEYR1[15:0]															
Rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	Rw

- **Bit [31:0] - AES_KEY1:** Esse registro armazena os bits [63:32] da chave de criptografia.

B.2.7 Registro *AES_KEY2*

Endereço de offset: 0x18

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEYR2[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEYR2[15:0]															
Rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	Rw

- *Bit [31:0] - AES_KEY2:* Esse registro armazena os bits [95:64] da chave de criptografia.

B.2.8 Registro *AES_KEY3*

Endereço de offset: 0x1C

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
KEYR3[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
KEYR3[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_KEY3:** Esse registro armazena os bits [127:96] da chave de criptografia.

B.2.9 Registro *AES_IVR0*

Endereço de offset: 0x20

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IVR0[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVR0[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_IVR0:** Esse registro armazena os bits [31:0] do vetor de inicialização IV.

B.2.10 Registro *AES_IVR1*

Endereço de offset: 0x24

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IVR1[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_IVR1:** Esse registro armazena os bits [63:32] do vetor de inicialização IV.

B.2.11 Registro *AES_IVR2*

Endereço de offset: 0x28

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IVR2[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVR2[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_IVR2:** Esse registro armazena os bits [95:64] do vetor de inicialização IV.

B.2.12 Registro *AES_IVR3*

Endereço de offset: 0x2C

Valor de Reset: 0x00000000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IVR3[31:16]															
rw	rw	rw	rw	rw	rw	rw	Rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVR3[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **Bit [31:0] - AES_IVR3:** Esse registro armazena os bits [127:96] do vetor de inicialização IV.