



M 2014

IMPLEMENTAÇÃO EM VERILOG DO ALGORITMO DE CIFRA AES-CTR PARA APLICAÇÕES HDMI 2.0

HUGO MIGUEL TEIXEIRA FERNANDES

DISSERTAÇÃO DE MESTRADO APRESENTADA

À FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO EM
ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

A Dissertação intitulada

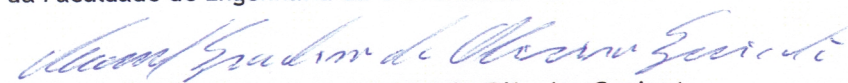
“Implementação em Verilog do Algoritmo de Cifra AES-CTR para Aplicações
HDMI 2.0”

foi aprovada em provas realizadas em 14-07-2014

o júri



Presidente Professor Doutor António José Duarte Araújo
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

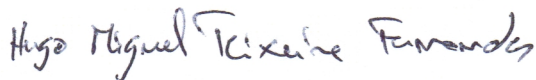


Professor Doutor Manuel Gradim de Oliveira Gericota
Professor Adjunto do Departamento de Engenharia Eletrotécnica do Instituto
Superior de Engenharia do Porto



Professor Doutor João Paulo de Castro Canas Ferreira
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.



Autor - Hugo Miguel Teixeira Fernandes

Faculdade de Engenharia da Universidade do Porto



Implementação em Verilog do algoritmo de cifra AES-CTR para aplicações HDMI 2.0

Hugo Miguel Teixeira Fernandes

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador FEUP: Prof. Doutor João Canas Ferreira

Orientador Synopsys: Eng.º Rui Rainho Almeida

31 de Julho de 2014

Resumo

Com uma base instalada superior a dois mil milhões de dispositivos, a tecnologia HDMI tornou-se a interface multimédia com maior adoção para aplicações de entretenimento caseiro e multimédia móvel. A recente atualização da especificação HDMI para a versão 2.0 trouxe a possibilidade de transporte de resoluções mais elevadas, tais como Ultra HD (4K x 2K a 60Hz). De forma a proteger os conteúdos *premium* contra cópia não autorizada, a interface HDMI permite a inclusão de tecnologia *high-bandwidth Digital Content Protection System* (HDCP), a qual recorre ao algoritmo de cifra *Advanced encryption standard* em modo contador (AES-CTR).

A inclusão da cifra AES-CTR em aplicações HDMI requer um objectivo de desempenho muito elevado, uma vez que a cifra é aplicada no caminho de dados áudio/vídeo de 24 bits, com uma cadência de até 600MHz.

Esta dissertação apresenta um estudo sobre a cifra AES e a sua implementação no protocolo HDCP 2.2 para aplicações HDMI 2.0. Foi realizado um estudo de várias implementações possíveis, sendo analisado o desempenho e a área ocupada.

Foram também implementadas e desenvolvidas várias arquiteturas AES para utilização em modo contador no protocolo HDCP 2.2. Sendo realizado o fluxo de projeto de *Front-End* desde a especificação até a *netlist gate-level*, assim como a validação do mesmo, desenvolvendo o plano de verificação e ambiente de teste.

Abstract

With over two billion devices, HDMI technology has become the largest multimedia interface adopted in home entertainment, multimedia and mobile applications. The latest update of the HDMI specification to version 2.0 increased the capability to carry higher video resolutions, such as Ultra HD (4K x 2K at 60Hz). In order to protect premium content from unauthorized copying, HDMI technology allows the inclusion of high-bandwidth Digital Content Protection System (HDCP), which uses the AES-CTR encryption algorithm.

The inclusion of the cipher AES-CTR in HDMI applications entails a very high performance objective, since the cipher is applied to the audio path data / video 24-bit, with a cadence of up to 600MHz.

This dissertation presents a study on the AES cipher and its implementation in the HDCP 2.2 protocol for HDMI 2.0 applications. A study of several possible implementations was performed with the objective of analyzing performance and area.

The Front End flow was held from specification to the gate-level netlist. A verification plan and test-bench was created to validate all implementations.

Agradecimentos

Gostaria de agradecer,

Em especial aos Meus Pais, Mário e Júlia Fernandes, e a minha irmã Marta, por acreditarem sempre em mim, espero de alguma forma poder retribuir e compensar todo o carinho, apoio e dedicação que, constantemente me ofereceram, sem eles nada disto seria possível. E a eles, dedico todo este trabalho.

Aos meus orientadores, Professor Doutor João Canas Ferreira e ao meu orientador da Synopsys Eng. Rui Rainho Almeida, por toda a paciência e compreensão. Agradeço a oportunidade e o privilégio que tive em frequentar este Mestrado que muito contribuiu para o enriquecimento da minha formação académica, assim como a oportunidade de ter realizado esta dissertação na Synopsys onde aprendi, desenvolvi os meus conhecimentos e cresci a nível profissional e pessoal.

Agradeço a toda a equipa de HDMI da Synopsys que me acolheu e me proporcionou todas as condições necessárias para a elaboração deste trabalho, por todos os conhecimentos que me proporcionaram, paciência, amizade e dedicação, e em especial ao Eng. Luís Laranjeira que fez com que tudo isto fosse possível.

Agradeço a todos os meus amigos que de uma forma directa ou indirecta sempre estiveram comigo e me apoiaram. Obrigada pela vossa amizade, companheirismo e ajuda, factores muito importantes na realização desta dissertação e que me permitiram que cada dia fosse encarado com particular motivação.

“Believe you can and you’re halfway there ”

Theodore Roosevelt

Conteúdo

1	Introdução	1
1.1	Enquadramento geral	1
1.2	Sobre a Synopsys	2
1.3	Objetivos	2
1.4	Estrutura do documento	3
2	Enquadramento técnico e Estado da Arte	5
2.1	High Digital Content Protection (HDCP)	5
2.2	Advanced Encryption Standard (AES)	6
2.3	Operações Aritméticas Básicas	8
2.4	Estrutura Interna do AES	8
2.4.1	SubBytes	8
2.4.2	ShiftRows	9
2.4.3	MixColumns	9
2.4.4	AddRoundKey	10
2.4.5	Key Expansion	10
2.5	Decifração	11
2.5.1	Inverse MixColumns	11
2.5.2	Inverse ShiftRows	12
2.5.3	Inverse SubBytes	12
2.5.4	Key Expansion	13
2.6	Modos de Operação	13
2.6.1	Electronic codebook (ECB)	13
2.6.2	Cipher Block Chaining (CBC)	13
2.6.3	Cipher Feedback Mode (CFB)	14
2.6.4	Output Feedback Mode (OFB)	15
2.6.5	Counter Mode (CTR)	16
2.7	Implementações AES	17
3	Arquiteturas Desenvolvidas	25
3.1	Introdução	25
3.2	Interface e Requisitos do Projeto	25
3.3	Interface de Alto Nível	26
3.4	Plano de Verificação	26
3.5	AES4box	27
3.6	AES8box	32
3.7	AES16box	35
3.8	AES16box2	38

3.9	Ambiente de teste	38
4	Fluxo de Projeto	45
4.1	Fluxo de Projeto Front-End	46
4.2	Especificação	46
4.3	Arquitetura Alto Nível	46
4.4	Desenvolvimento do código	46
4.5	Linting RTL	46
4.6	Simulação/Verificação	47
4.7	Síntese Lógica	47
4.8	Verificação de Equivalência	52
4.9	Verificação de Scan	52
4.10	Análise Temporal Estática	53
5	Resultados	55
5.1	Resultados obtidos em FPGA	55
5.2	Resultados obtidos em ASIC	56
6	Conclusão	59
6.1	Conclusões	59
6.2	Trabalho Futuro	59
A	Relatórios da Analise Estática temporal	61
B	Bancada de Teste	65
C	Resultados dos Padrões de teste	73
D	Caminho crítico da Síntese para FPGA	81
	Referências	85

Lista de Figuras

2.1	Estrutura da cifra AES-CTR no protocolo HDCP[1]	6
2.2	Estrutura geral do algoritmo AES[2]	7
2.3	Transformação Affine	9
2.4	Transformação ShiftRows[3]	9
2.5	Transformação MixColumns[2]	10
2.6	Algoritmo de expansão da chave[1]	12
2.7	Transformação inversa <i>affine</i>	13
2.8	Cifração e Decifração no modo CBC	14
2.9	Cifração e Decifração no modo CFB	15
2.10	Cifração e Decifração no modo OFB	16
2.11	Cifração e Decifração no modo CTR	17
2.12	Implementações S-box realizadas em [2]	18
2.13	Área vs Taxa de transferência com <i>Key Expansion online</i> [2]	19
2.14	Área vs Taxa de transferência com <i>Key Expansion offline</i> [2]	19
2.15	Arquiteturas implementadas em [3]	20
2.16	Resultados obtidos por [4]	21
2.17	Arquitetura realizada em [5]	21
2.18	Arquitetura da implementação AES em [6]	22
2.19	Resultado obtidos em [6]	23
3.1	Arquitetura AES4box Top	28
3.2	Ronda da Arquitetura AES4box	28
3.3	Topo da Arquitetura AES8box	33
3.4	Ronda da Arquitetura AES8box	36
3.5	Topo da Arquitetura AES16box	36
3.6	topo da Arquitetura AES16box2	39
3.7	Ronda da Arquitetura AES16box e AES16box2	39
3.8	Formas de onda da verificação funcional	43
3.9	Simulação com auto verificação	43
4.1	Fluxo de projeto <i>Front-End</i> [7]	45
4.2	Resultados de coverage obtido da Arquitetura AES16box2	48

Lista de Tabelas

3.1	Sinais do bloco de Alto nível	26
3.2	Plano de Verificação do design efetuado	27
5.1	Resultados Obtidos das Várias Arquiteturas	55
5.2	Resultados obtidos após Place-and-Route	56
5.3	Diferenças das várias Arquiteturas	56
5.4	Resultados Obtidos das Várias Arquiteturas	57
5.5	Comparação com a literatura	57

Abreviaturas e Símbolos

AES	Advanced encryption standard
AES-CTR	Advanced encryption standard in counter mode operation
AESAVS	AES Algorithm Validation Suite
ASIC	Aplication Specific Integrated Circuit
AV	Áudio e Vídeo
BD	Blu-ray
CEC	Consumer Electronics Control
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CTR	Counter Mode
DVD	Digital Versatile Disc
DUT	Device Under Test
DVI	Digital Visual Interface
ECB	Electronic Codebook
FPGA	Field Programmable Gate Array
GF	Campos finitos de Galois
HD	High Definition
HDCP	High-Bandwith Digital Content Protection
HDMI	High-Definition Multimedia Interface
IV	Input Vector
k_s	Chave De Sessão
LUT	Look-Up-Table
NIST	National Institute of Standards and Technology
OFB	Output Feedback
RTL	Register Transfer Logic
TMDS	Transition-Minimized Differential Signaling
Soc	System-On-A-Chip
Ultra HD	Ultra High Definition

Capítulo 1

Introdução

Esta dissertação foi proposta pela SYNOPSYS Portugal e é realizada no âmbito do Mestrado Integrado em Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto.

1.1 Enquadramento geral

Os conteúdos audiovisuais estão cada vez mais a ser distribuídos aos consumidores em formato digital, através da Internet, redes de satélites, reprodutores *DVD(digital versatile disc)* ou *BD (blu-ray)*. Esta vasta disponibilidade de conteúdo digital fez com que os seus fornecedores se preocupassem cada vez mais com a cópia não autorizada e a sua distribuição. Como resultado os fabricantes de conteúdos, fornecedores e os fabricantes de dispositivos eletrónicos implementaram uma variedade de técnicas de proteção, que protegem o acesso ao conteúdo multimédia distribuído através de diferentes meios de comunicação. O protocolo *HDCP (High-bandwidth Digital Content Protection)* é uma parte desta cadeia de proteção. Este protege o último estágio do processo de distribuição, a transmissão do conteúdo entre um dispositivo transmissor e receptor, como por exemplo, um leitor de *DVD* e um televisor. Tendo sido desenvolvido pela Intel, o sistema foi feito para parar de transmitir conteúdo em dispositivos não autorizados ou que foram modificados para copiar o conteúdo *HDCP*. Antes de enviar, o dispositivo transmissor verifica se o recetor está autorizado a receber o conteúdo. Se tal acontecer, o transmissor cifra o conteúdo para prevenir a cópia deste enquanto é transmitido. Os interfaces que obedecem a norma *HDCP* são o *HDMI(High-Definition Multimedia Interface)*, *DVI* e *DisplayPort*. Com uma base instalada superior a dois mil milhões de dispositivos de consumo, sendo de longe a interface com maior adoção para aplicações "home entertainment" e "mobile multimedia", o interface *HDMI* tornou-se o standard substituindo o *DVI*. O desenvolvimento do *HDMI 1.0* foi iniciado em 2002 com o objetivo de criar um conector AV que fosse retro compatível com o *DVI*. Na altura o standard *DVI* estava a ser usado em televisores *HD*. O *HDMI 1.0* foi desenhado para melhorar o *DVI* ao usar um conector mais pequeno com suporte para áudio e suporte melhorado para "YCbCr" e funções de controlo dos dispositivos (*CEC*) [8]. Desde então a interface *HDMI* tem evoluído, estando neste momento na especificação

2.0. A cada nova especificação foram sendo acrescentadas funcionalidades e aumentada a largura de banda. Quando foi lançada a especificação HDMI 1.0 esta suportava uma taxa de transferência *TMDS* máximo de 4.95 Gbits/s tendo evoluído para 18Gbit/s na especificação HDMI 2.0. A especificação HDMI 2.0 trouxe a possibilidade de transporte de resoluções mais elevadas, tais como *Ultra HD (Ultra High Definition)* (4k x 2k a 60Hz). O aumento da taxa de transferência trouxe a necessidade da implementação em hardware de um algoritmo de cifra AES-CTR capaz de suportar uma taxa de transferência de 18Gbit/s. Esta dissertação realizada em conjunto com a SYNOPSYS tem como objetivo a implementação de um algoritmo de cifra AES-CTR capaz de obter a taxa de transferência acima referida, assim como uma otimização a nível de área e número de gates de modo a se conseguir reduzir o tamanho do die e consequentes custos de produção.[1]

1.2 Sobre a Synopsys

A SYNOPSYS é a empresa líder de mercado em propriedade intelectual de semicondutores (IP) e automação de desenho eletrónico (EDA). Por mais de 25 anos, a SYNOPSYS tem sido o motor na aceleração da inovação eletrónica com engenheiros em todo o mundo a usarem a sua tecnologia para projetar e criar com sucesso milhares de milhões de chips e sistemas eletrónicos de que as pessoas dependem no dia a dia. Foi fundada em 1986 pelo Dr. Aart de Geus e uma equipa de engenheiros do Centro de Micro-eletrónica da General Electrics na Carolina do Norte. A empresa foi pioneira na aplicação comercial de síntese lógica que já foi adotada por todas as grandes empresas de design de semicondutores do mundo. Esta tecnologia proporcionou um salto exponencial de produtividade no projeto de desenho de circuitos integrados (IC), permitindo aos engenheiros especificar a funcionalidade dos chips a um nível mais alto de abstração. Sem essa tecnologia, os projetos complexos de hoje não seriam possíveis. A empresa abastece o mercado de eletrónica global com software, propriedade intelectual(IP), serviços utilizados no design de circuitos integrados, verificação e fabrico. Tudo isto é conseguido fornecendo um portfólio integrado de implementação, verificação, fabrico, e *field-programmable gate array* (FPGA) para abordar os desafios na produção de circuitos integrados (IC) [7].

1.3 Objetivos

A inclusão da cifra AES-CTR em aplicações HDMI 2.0 acarreta um objetivo de desempenho muito elevado, uma vez que a cifra é aplicada no caminho de dados áudio/vídeo de 24 bits, com uma cadência de até 600MHz. Pretende-se por isso desenvolver uma implementação que permita síntese para ASIC em tecnologia 40nm, cumprindo os objetivos de desempenho mencionados. Esta dissertação tem como objetivos o desenvolvimento, implementação e verificação do algoritmo de cifra *Advanced Encryption Standard* (AES) em modo de operação contador (CTR). A implementação será realizada em linguagem de descrição de hardware Verilog e terá de ser sintetizável em tecnologia TSMC 40nm, com um objetivo de desempenho de 600 MHz. Esta também

terá de ser sintetizada em FPGA Xilinx Virtex 5, com um objetivo de desempenho de 300 MHz. A implementação deverá ser capaz de um processamento de 24 bits por ciclo de relógio.

1.4 Estrutura do documento

Este trabalho está organizado em 6 capítulos, incluindo este capítulo de Introdução.

No capítulo 1 é feita uma introdução do trabalho, onde é descrita o enquadramento e a motivação para a sua elaboração.

No capítulo 2 "Enquadramento técnico e Estado da Arte", é feito um estudo do algoritmo de cifra AES, sobre o modo como é feita a cifração e decifração do mesmo assim como os seus vários modos de operação e o seu enquadramento em aplicações HDMI 2.0. É apresentado também um estudo sobre implementações do algoritmo.

No capítulo 3 "Arquiteturas Desenvolvidas", são apresentadas as arquiteturas do algoritmo AES em modo CTR desenvolvidas durante este trabalho, onde são apresentados os sinais de interface, diagrama de blocos e é explicado o funcionamento de cada uma delas.

No capítulo 4, "Fluxo do projeto", são apresentadas as várias etapas realizadas no desenvolvimento deste trabalho, explicada a sua importância, o que representa cada fase, assim como as ferramentas utilizadas em cada uma delas.

No capítulo 5, são apresentados os resultados obtidos, tais como resultados de síntese para ASIC e FPGA.

No capítulo 6, são apresentadas as conclusões e o trabalho futuro.

Capítulo 2

Enquadramento técnico e Estado da Arte

Neste capítulo é feita uma introdução ao protocolo *HDCP* e mais concretamente a cifração do conteúdo, o algoritmo usado e o seu funcionamento interno.

2.1 High Digital Content Protection (HDCP)

Com a introdução e adoção de *HDTV*(*High-Definition Television*) e a sua capacidade de suportar resoluções de vídeo muito elevadas, alguns segmentos da indústria aumentaram as suas preocupações em relação aos riscos de pirataria, principalmente devido a cópia não autorizada de filmes em alta resolução e a consequente perda de rendimentos que daí advém, fez com que se tomassem medidas para a sua prevenção. Apesar de já existirem métodos de proteção contra cópia não autorizada do meio físico, por exemplo de *DVD* e *BD*. Tornou-se necessário proteger também a transmissão do seu conteúdo durante a reprodução de modo a não poder ser copiado durante essa fase. O *HDCP* foi desenvolvido para proteger a transmissão de conteúdo audiovisual entre um transmissor e recetor, o *HDCP* recorre ao algoritmo de cifra *AES-CTR* para a cifração e decifração do conteúdo transmitido. O processo decorre da seguinte forma, após a fase de autenticação entre o transmissor e o recetor, é iniciada uma sessão de troca de chave, onde o transmissor gera uma chave de sessão de 128 bits e um vetor de entrada pseudo aleatório *IV* de 64 bits, encripta esses valores e os envia para o recetor que os recupera. Após esta troca, o transmissor espera pelo menos 200ms até ativar a encriptação e dar início a transmissão de conteúdo encriptado. A encriptação é feita usando o algoritmo de cifra *AES* no modo *counter* que será explicado ao pormenor nas secções seguintes. Na figura 2.1 é apresentada a estrutura da cifra *HDCP*, em que a chave de cifração corresponde a um *XOR* entre k_s , que corresponde a chave de sessão e lc_{128} que é uma constante global de 128 bits partilhada por todos os dispositivos *HDCP*. O bloco de entrada de 128 bits a ser cifrado corresponde a uma concatenação entre o vetor de entrada de 64 bits *IV* e o bloco *inputCtr* também de 64 bits. Por sua vez, *inputCtr* corresponde a uma concatenação de *FrameNumber* com *DataNumber*, $inputCtr = FrameNumber || DataNumber$. *FrameNumber* é um

valor de 38 bits que indica o número de tramas cifradas desde o início da cifração no protocolo *HDCP*. O valor de *FrameNumber* é incrementado por 1 a cada trama transmitida. *DataNumber* é um valor de 26 bits que é incrementado a cada geração de um bloco *keystream* de 128 bits. O valor do *inputCtr* é inicializado a zero quando a encriptação é ativada pela primeira vez após o início de sessão. *keystream* corresponde ao bloco de saída da cifra, em que é realizado um *XOR* com *InputData* que correspondem aos dados a serem cifrados e temos como saída *Encrypted Output* que corresponde aos dados cifrados[1].

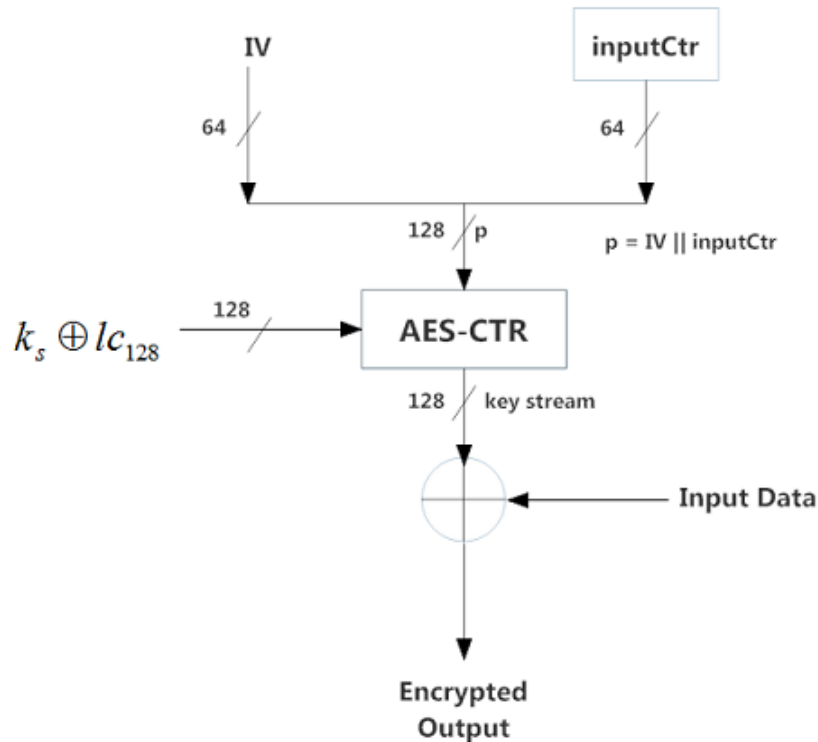


Figura 2.1: Estrutura da cifra AES-CTR no protocolo HDCP[1]

2.2 Advanced Encryption Standard (AES)

Em 1997 o *National Institute of Standards and Technology*(NIST) abriu um concurso para um novo padrão de encriptação Avançada(AES).

Os algoritmos candidatos tinham de preencher os seguintes requisitos:

- Cifra de bloco com tamanho de 128 bits;
- Suportar três tamanhos de chave diferentes (128, 192 e 256) bits;
- Segurança relativamente aos outros algoritmos submetidos;
- Eficiência em hardware e software.

Em 2000 o *NIST* anunciou a escolha do bloco *Rijndael* como a cifra AES, tornando-se o padrão em 26 de Maio de 2002. Esta cifra é obrigatória em muitos padrões da indústria tais como *HDPC*, *TLS*, padrão de cifra *wifi IEEE 802.11i*, entre outros. A cifra AES, é uma cifra de bloco simétrica em que o comprimento do bloco de dados é de 128 bits e o comprimento da chave de cifração pode ser 128, 192 ou 256 bits. O AES opera numa matriz de bytes 4x4 denominada de matriz de estado e a maioria dos cálculos do AES são feitos em operações aritméticas básicas em campos finitos de Galois(2^8).

A encriptação é feita em rondas. O número de rondas é definido pelo comprimento da chave e pode ser 10, 12 ou 14 conforme a chave seja de 128, 192 e 256 bits. Cada ronda com exceção da última, é composta por quatro transformações: SubBytes, ShiftRows, MixColumns e AddRoundKey as quais serão explicadas na secção seguinte. Na matriz de estado os bytes são orientados por coluna, isto é, os primeiros quatro bytes do bloco de entrada são dispostos na primeira coluna, os quatro bytes seguintes na segunda coluna e assim consecutivamente. [9][10][11]

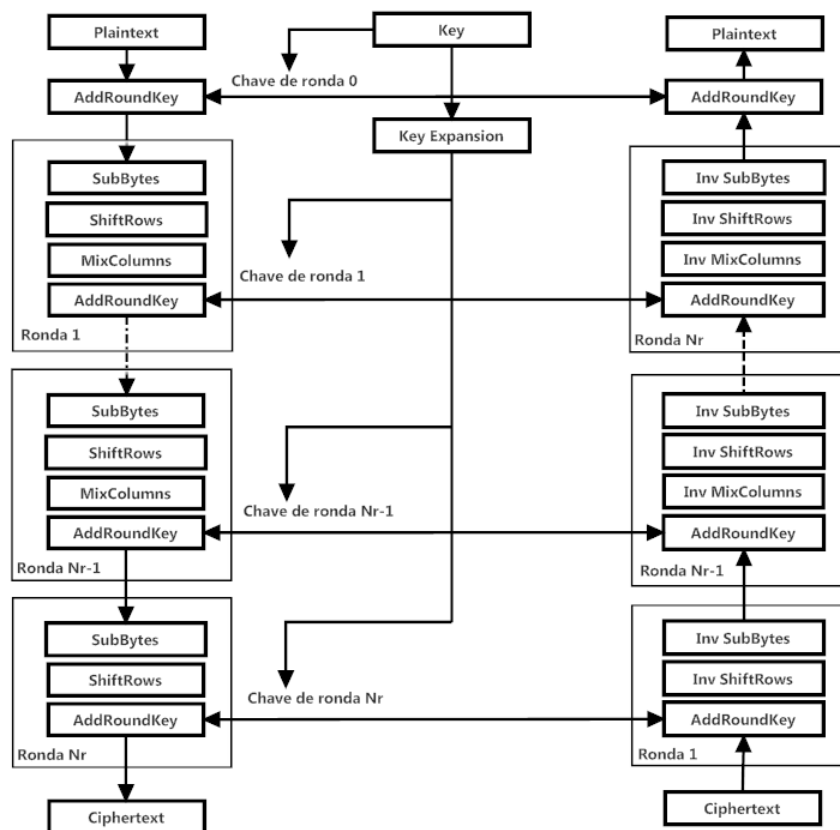


Figura 2.2: Estrutura geral do algoritmo AES[2]

A estrutura geral do algoritmo AES, é apresentada na figura 2.2. Na figura pode-se ver que o "plaintext", dados a serem cifrados com o tamanho de 16 bytes, são colocados na matriz de estado 4x4, a qual é feita uma transformação inicial chamada de "key whitening" que consiste numa transformação AddRoundKey com a chave original, após a qual se dá o início das rounds

que são todas exatamente iguais com a exceção da última em que a transformação MixColumns não é feita, dando depois origem ao "*Ciphertext*" que corresponde aos dados cifrados.

2.3 Operações Aritméticas Básicas

No algoritmo AES, todos os bytes são interpretados como sendo elementos de um campo finito que podem ser representados por uma descrição polinomial, como a expressa pela equação

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$$

A adição de dois elementos em campos finitos é conseguida, adicionando os coeficientes das respectivas potências através de uma operação XOR modulo 2 o que corresponde a um *XOR bitwise*. A adição e a subtração de polinómios produzem o mesmo resultado.

No caso da multiplicação, esta é dada pelo resto da divisão do produto de dois polinómios por um polinómio irreduzível, ou, seja, um polinómio de grau 8 que é divisível por 1 e por ele próprio.

No algoritmo AES, o polinómio irreduzível é o da expressão seguinte:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

A redução modular por $m(x)$ assegura que o resultado será sempre um polinómio binário de grau inferior a oito, e que pode ser representado por um byte.

2.4 Estrutura Interna do AES

Nesta secção é analisada a estrutura interna do AES.

2.4.1 SubBytes

A transformação SubBytes, consiste numa transformação não linear em que cada byte da matriz de estado é substituído por outro através de uma tabela de referência à qual se dá o nome de *S-box*. A tabela de substituição ou S-box é invertível e é construída pela composição de duas operações, a multiplicativa inversa em $GF(2^8)$ e a transformação *Affine* em $GF(2)$, esta última dada por:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

para $0 \leq i < 8$, onde b_i é o i^{th} bit do byte e c_i é o i^{th} bit do byte c com o valor hexadecimal 0x63. Esta transformação pode ser representada no formato de matriz, como mostra a figura 2.3.

A tabela S-box pode ser vista como uma tabela 16 x 16 com 1 byte de entrada e 1 byte de saída. A substituição é feita da seguinte forma, os quatro bits mais significativos do byte a ser substituído representam a posição da linha e os quatro menos significativos a posição na coluna. Estes valores servem de índices da tabela para seleccionar um byte da mesma.

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \pmod{2}$$

Figura 2.3: Transformação Affine

2.4.2 ShiftRows

A transformação ShiftRows, consiste numa permutação simples, onde a primeira linha se mantém inalterada, a segunda linha da matriz é rodada à esquerda por um byte, a terceira linha é rodada à esquerda de dois bytes e a quarta linha de três bytes. O objetivo desta é de aumentar a propriedade de difusão. A transformação ShiftRows é ilustrada na figura 2.4.

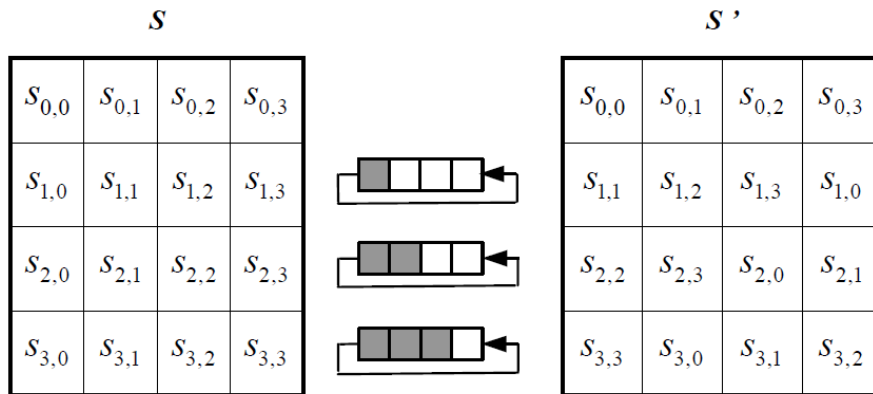


Figura 2.4: Transformação ShiftRows[3]

2.4.3 MixColumns

A transformação MixColumns consiste numa substituição que faz uso da aritmética sobre $GF(2^8)$ e opera em cada coluna da matriz de estado individualmente. Cada coluna de 4 bytes é considerada como um vetor e é multiplicada por uma matriz 4x4. A matriz contém valores constantes. A figura 2.5 apresenta a transformação MixColumns.

Cada elemento da matriz de saída é a soma dos produtos dos elementos de uma linha e uma coluna. A transformação de uma única coluna da matriz de estado pode ser expressa pelas equações

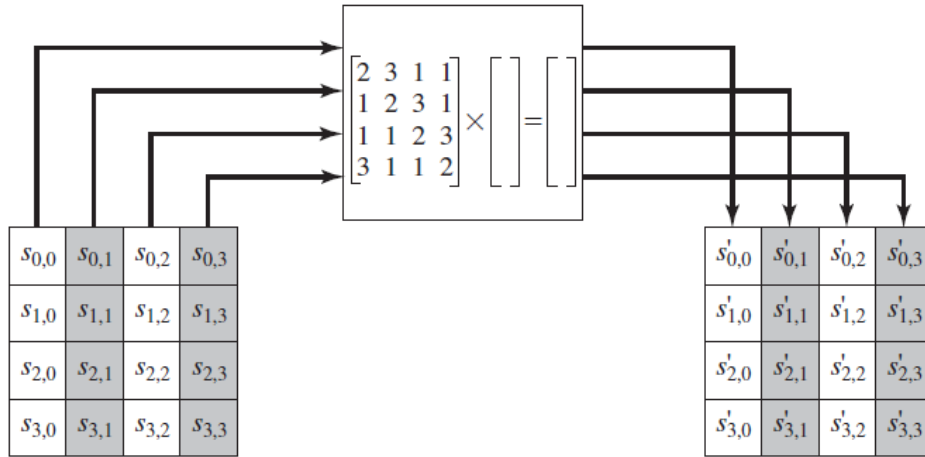


Figura 2.5: Transformação MixColumns[2]

2.1:

$$\begin{aligned}
 s'_{0,j} &= (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\
 s'_{1,j} &= s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j} \\
 s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j}) \\
 s'_{3,j} &= (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j})
 \end{aligned} \tag{2.1}$$

Como se pode observar pelas equações 2.1, cada byte de entrada influencia quatro bytes de saída, esta operação é o maior elemento difusor da cifra *AES*. A combinação de ShiftRows e MixColumns torna possível que, ao fim de apenas três rondas, cada byte da matriz dependa de todos os 16 bytes do *plaintext*.

2.4.4 AddRoundKey

Esta transformação consiste apenas na aplicação de uma operação lógica XOR bit a bit do bloco da matriz de estado com os 128 bits da chave expandida correspondente a essa *round*.

2.4.5 Key Expansion

O algoritmo de expansão de chave tem como entrada a chave sendo calculadas 10 subchaves, que são depois guardadas, a chave e as subchaves, num array de 44 *words*. A chave e as subchaves são posteriormente usadas em cada ronda na transformação AddRoundKey. As subchaves são calculadas como mostra a figura 2.6, em que a *word* mais a esquerda de cada chave de ronda $W[4i]$, é calculada da seguinte forma, sendo $i = 1, \dots, 10$:

$$W[4i] = W[4(i-1)] + g(W[4i-1])$$

onde $g()$ é uma função não linear com quatro bytes de entrada e saída. As restantes três *words* de cada subchave são calculadas como:

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j]$$

onde $i = 1, \dots, 10$ e $j = 1, 2, 3$. A função $g()$ como mostra a figura 2.6, faz uma rotação de um byte para a esquerda da word de entrada, executa uma substituição SubBytes nos quatro bytes de entrada e adiciona uma constante *round coefficient*(RC) ao byte mais a esquerda na função $g()$. O *round coefficient* varia de ronda para ronda de acordo com:

$$RC[1] = x^0 = (00000001)_2$$

$$RC[2] = x^1 = (00000010)_2$$

$$RC[3] = x^2 = (00000100)_2$$

$$\vdots$$

$$RC[10] = x^9 = (00110110)_2$$

A finalidade da função $g()$ é a de adicionar não linearidade e remover a simetria das chaves.

2.5 Decifração

No processo de decifração todas as transformações do AES têm de ser invertidas. A SubBytes transforma-se na transformação Inverse SubBytes, a Shiftrows em Inverse Shiftrows e a Mixcolumns em Inverse Mixcolumns. Na decifração, a ordem das transformações é invertida em relação ao processo de cifração, onde a última ronda da cifração, torna-se na primeira ronda da decifração e as transformações de cada ronda também são executadas na ordem inversa. Como a última ronda de encriptação não executa a transformação Mixcolumns, a primeira ronda da decifração, também não executa a inversa da transformação Mixcolumns. A operação Addroundkey mantém-se igual, isto porque a inversa da operação XOR é ela própria.

2.5.1 Inverse MixColumns

Na decifração, de maneira a reverter a transformação Mixcolumns, deve-se aplicar a sua inversa. Na transformação Inverse MixColumns cada coluna da matriz de estado é multiplicada por uma matriz 4x4, tal como a transformação Mixcolumns, mas os valores fixos desta matriz correspondem aos valores inversos dos utilizados em Mixcolumns. A multiplicação e adição dos coeficientes é feita igualmente em $GF(2^8)$.

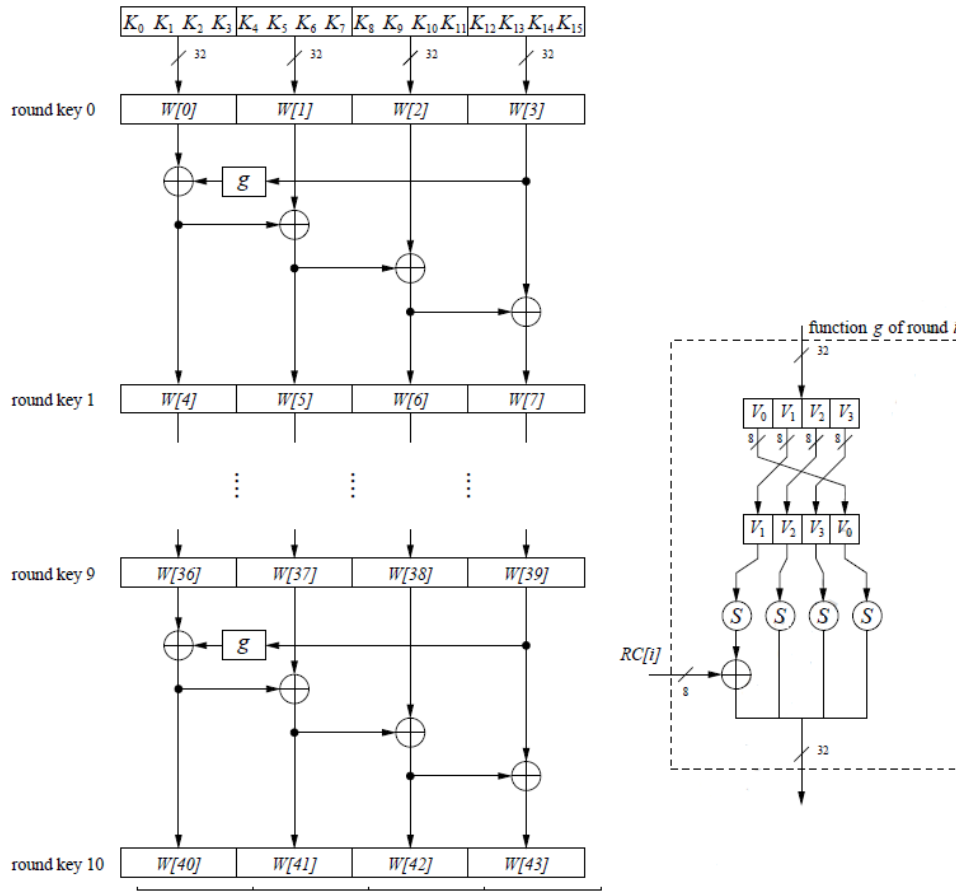


Figura 2.6: Algoritmo de expansão da chave[1]

2.5.2 Inverse ShiftRows

De maneira a reverter a operação ShiftRows, temos de rodar a matriz na direção oposta à transformação ShiftRows. A primeira linha não é alterada, a segunda linha sofre a rotação de um byte a direita, a terceira linha, de dois bytes na mesma direção e a última linha de três bytes também para a direita.

2.5.3 Inverse SubBytes

Para o cálculo da S-Box inversa, é necessário primeiro calcular a inversa da transformação *affine*, apresentada na figura 2.7. Para isso cada byte B_i é considerado um elemento de $GF(2^8)$. A transformação inversa *affine* em cada byte é definida por:

onde $(b_7, \dots, b_0)$ é o vetor de representação bitwise de $B_i(x)$, e $(b'_7, \dots, b'_0)$ o resultado da transformação *affine* inversa. Após esta transformação, tem de se inverter a inversa de Galois Field. Isto é conseguido aplicando a inversa de Galois Field, como se vê na equação seguinte:

$$A_i = (B'_i)^{-1}$$

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} \equiv \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \pmod{2},$$

Figura 2.7: Transformação inversa *affine*

2.5.4 Key Expansion

A transformação Key Expansion, não sofre qualquer alteração, sendo igual à usada para a encriptação. Como a primeira ronda da operação de decifração precisa da última chave de ronda usada na operação de cifração, a segunda ronda precisa da penúltima, ou seja, as chaves são usadas em ordem invertida. Torna-se necessário que a transformação Key Expansion seja computada a priori, sendo as chaves de ronda guardadas e utilizada a respetiva chave na sua ronda correspondente.

2.6 Modos de Operação

2.6.1 Electronic codebook (ECB)

Este é o método de encriptação em que, para uma determinada chave, a encriptação de um bloco de dados fixo tem sempre como resultado o mesmo valor encriptado. Este é o método de encriptação mais simples, a mensagem é separada em blocos, sendo cada bloco encriptado separadamente.

A cifração e decifração são definidas como, sendo $j = 1 \dots n$:

$$C_j = CIPH_k(P_j)$$

$$P_j = CIPH_k^{-1}(C_j)$$

2.6.2 Cipher Block Chaining (CBC)

O modo de operação Cipher Block Chaining(CBC), é um modo de confidencialidade onde o processo de encriptação do bloco a ser cifrado, depende sempre do bloco anterior, à exceção do primeiro bloco, onde é usado um vetor de inicialização que é adicionado ao bloco a ser cifrado antes de se dar início à cifração. Nos blocos seguintes a serem cifrados é sempre adicionado com o

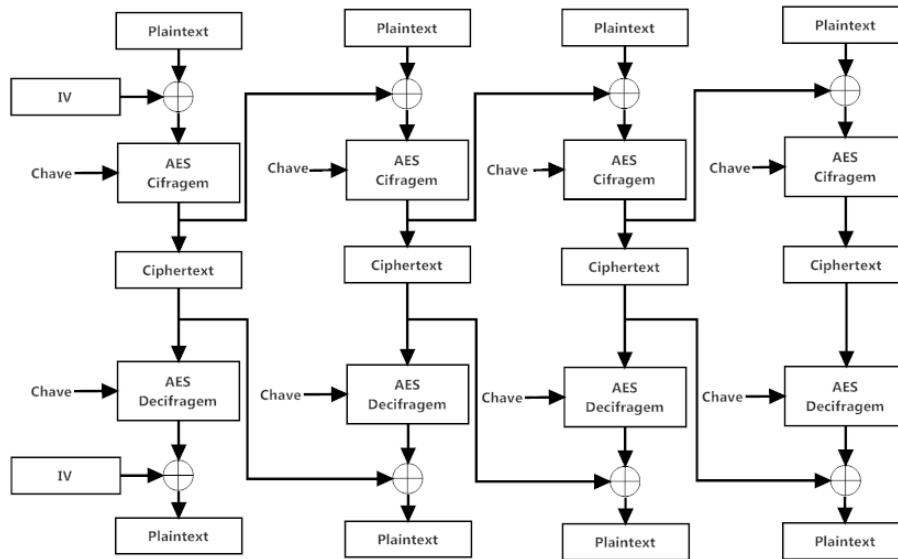


Figura 2.8: Cifração e Decifração no modo CBC

bloco cifrado anterior, através de um XOR, sendo portanto um processo encadeado. Este processo pode ser visto na figura 2.8, onde é apresentado o processo de cifração e decifração usando este modo de operação. A cifração e decifração são definidas como, sendo j , o número do bloco, para $j = 2 \dots n$ e IV o vetor de inicialização:

$$C_1 = CIPH_k(P_1 \oplus IV)$$

$$C_j = CIPH_k(P_j \oplus C_{j-1})$$

$$P_1 = CIPH_k^{-1}(C_1) \oplus IV$$

$$P_j = CIPH_k^{-1}(C_j) \oplus C_{j-1}$$

Este é o modo de operação mais usado, devido a ser um modo de operação sequencial, em que o bloco seguinte depende do bloco cifrado anterior. Este modo de operação não pode ser paralelizado.

2.6.3 Cipher Feedback Mode (CFB)

Este é um modo de cifração em avanço, onde o algoritmo é aplicado a um bloco de entrada e é feita a cifração do mesmo antes de termos os dados a serem cifrados. O modo de operação CFB possibilita a cifração de blocos de dados de tamanho inferior a 128bits, podendo mesmo cifrar um bit de cada vez. Este usa um parâmetro inteiro, chamado de "s", que define o tamanho do bloco a ser cifrado, estando compreendido entre $1 < s < 128$ bits. O segmento do bloco cifrado de tamanho "s", é usado como "feedback" e adicionado ao bloco de entrada seguinte. Normalmente o tamanho

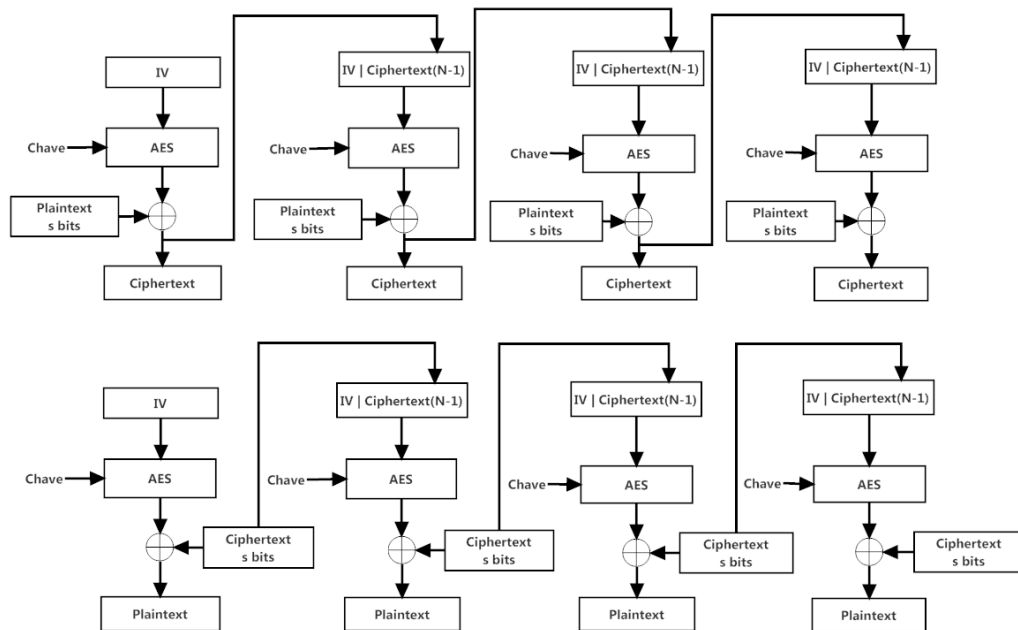


Figura 2.9: Cifração e Decifração no modo CFB

deste parâmetro costuma dar o nome ao modo de operação como por exemplo, 1-bit CFB mode, 8-bit CFB mode, 64-bit CFB mode, ou 128-bit CFB mode.

O primeiro bloco de entrada é o vetor de entrada, IV, sendo aplicada a operação de cifração, produzindo o primeiro bloco de saída. Em seguida é adicionado a este o segmento de dados "s" o qual é cifrado, através da operação XOR *bitwise*, com os s bits mais significativos do bloco de saída, obtendo o segmento de tamanho "s" de dados cifrados. Os restantes bits do bloco de saída da cifra são descartados.

Os restantes blocos de entrada são formados da seguinte forma, os 128-s bits menos significativos do IV são concatenados com os s bits do primeiro segmento de texto cifrado para formar o segundo bloco de entrada. Este processo é repetido com os sucessivos blocos de entrada. Em geral, cada bloco de entrada sucessivo é encriptado para produzir um bloco de saída. Os "s" bits mais significativos de cada bloco de saída são usados para a operação XOR com o segmento correspondente do bloco a ser cifrado para formar o segmento do texto cifrado, sendo realimentado para o próximo bloco de entrada, como descrito na figura 2.9, onde é mostrada a cifração e decifração usando este modo. Cada IV tem de ser único para a mesma chave de cifra

O processo de decifração é idêntico ao de cifração sendo que, a única diferença é que a operação *bitwise* XOR é feita com o segmento de texto cifrado, obtendo dessa forma o segmento de texto original

2.6.4 Output Feedback Mode (OFB)

Neste modo de operação a cifra AES também é feita em avanço, sendo depois o bloco de dados cifrado através da operação *bitwise* XOR, com o bloco de saída obtido da cifra. No primeiro bloco

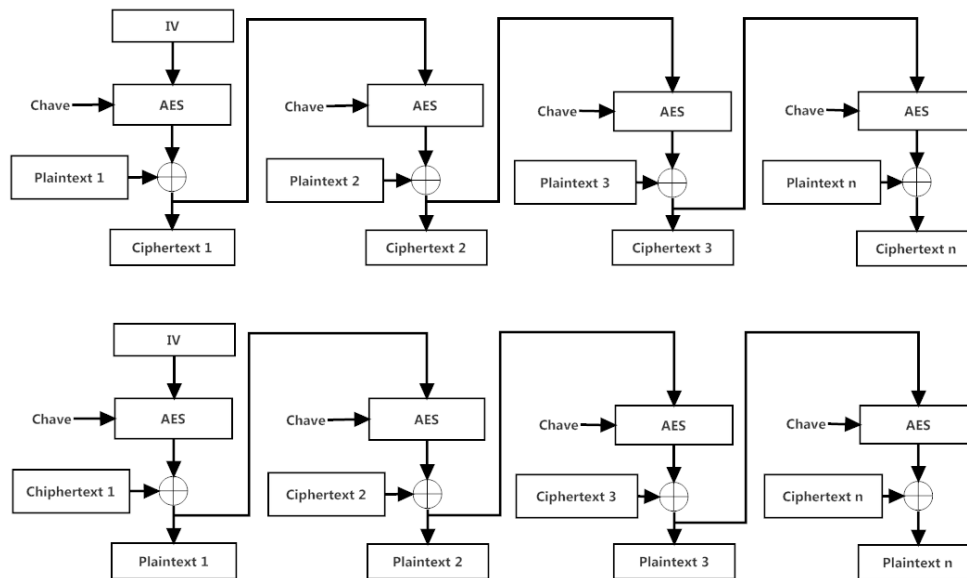


Figura 2.10: Cifração e Decifração no modo OFB

de entrada é usado um IV. Este tem de ser único para cada execução com uma dada chave. Nos restantes blocos, é usado o bloco de saída anterior como bloco de entrada, sendo a cifração feita da mesma forma que no primeiro bloco.

A decifração é feita exatamente da mesma forma. A operação de cifração e decifração são apresentadas na figura 2.10.

2.6.5 Counter Mode (CTR)

Este é o modo de funcionamento AES que foi implementado, sendo este o modo usado no protocolo HDCP. O modo de operação contador é caracterizado pela computação do bloco de cifração a priori utilizando blocos de entrada, chamados de contadores, o resultado da cifração é depois adicionada ao bloco de dados que queremos cifrar através de uma operação *XOR bitwise*, obtendo dessa forma o nosso bloco cifrado, como demonstra a figura 2.11

A sequência de contadores deve ter a propriedade de serem diferentes para cada bloco, e esta condição tem de ser verificada para todas as mensagens que são encriptadas com a mesma chave. Ou seja, todos os contadores têm de ter valor diferente.

Para o último bloco, que pode ser um bloco parcial de n bits, os n bits mais significativos do último bloco de saída são usados para a operação de XOR, sendo os restantes bits do bloco de saída descartados.

Tanto na cifração como na decifração em modo contador, o bloco pode ser executado em paralelo. Similarmente, o texto que corresponde a um bloco cifrado em particular pode ser recuperado independentemente de qualquer outro bloco se o correspondente IV puder ser determinado. A cifra pode ser aplicada aos contadores em avanço, antes que os dados a serem cifrados ou decifrados estejam disponíveis.

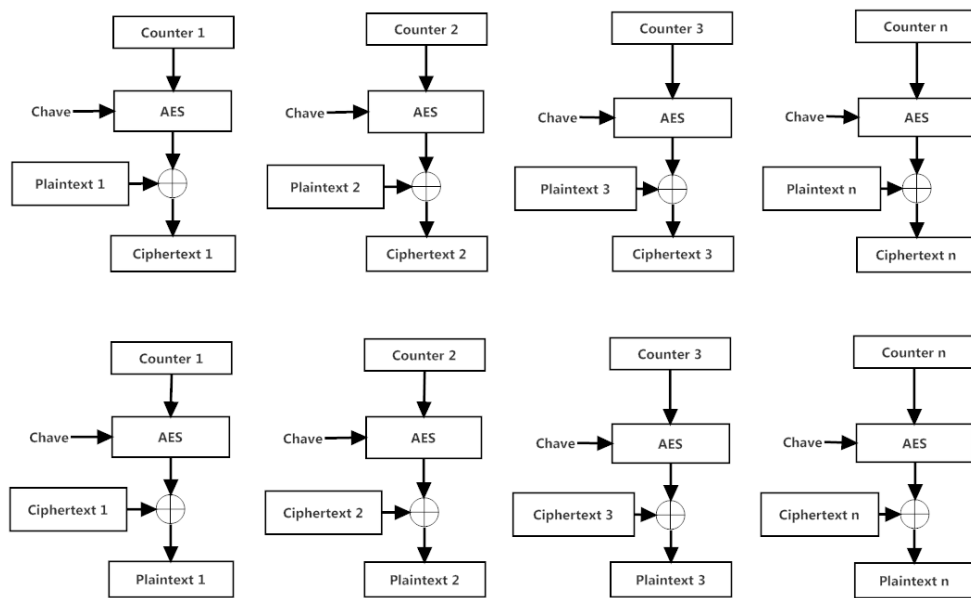


Figura 2.11: Cifração e Decifração no modo CTR

2.7 Implementações AES

Tipicamente a implementação das S-box para uma alta taxa de transferência é feita mediante a utilização de LUTs onde o valor de substituição se encontra pré-computado, permitindo desta forma uma taxa de transferência elevada ao permitir que esta transformada possa ser executada em apenas um ciclo de relógio recorrendo a paralelização. Mas esta implementação consome muita memória ao ser necessária a replicação da LUTs para cada byte e a replicação também de cada ronda, sendo necessárias 160 LUTs, 16 por byte mais 10 rounds, o que torna a sua implementação em ASIC demasiado custosa em termos de área consumida. Para se conseguir um baixo consumo de área, tem de se executar o cálculo das S-box durante a execução, mas o cálculo durante a execução das S-box usando campos finitos de Galois($GF(2^8)$) é complexa, reduzindo significativamente a taxa de transferência da cifra. Um dos inventores da cifra AES, sugeriu em [12] em vez do uso de $GF(2^8)$ para a computação da S-box, a decomposição e respectivo cálculo em $GF(2^4)$.

Cada elemento de $GF(2^8)$ pode ser escrito como um polinómio de primeiro grau com coeficientes de $GF(2^4)$.

O valor de entrada é mapeado em dois elementos $GF(2^4)$. A seguir é calculada a multiplicativa inversa usando $GF(2^4)$, sendo depois inversamente mapeados para $GF(2^8)$. Esta implementação apresenta duas vantagens, a redução da complexidade dos cálculos executados, assim como a possibilidade de *sub-pipelining* podendo dessa forma fazer-se o cálculo da S-box durante a execução, mantendo uma taxa de transferência próxima dos valores alcançados pela implementação através de LUT.

Embora a aplicação do cálculo da S-box durante a execução sobre $GF(2^4)$ reduzir bastante a área utilizada, este também reduz significativamente a taxa de transmissão, devido a esta imple-

mentação sofrer de um caminho crítico longo. Para ultrapassar esta desvantagem é usado *sub-pipelining*. Dessa forma consegue-se atingir uma taxa de transferência próxima da conseguida através de *LUT* [13].

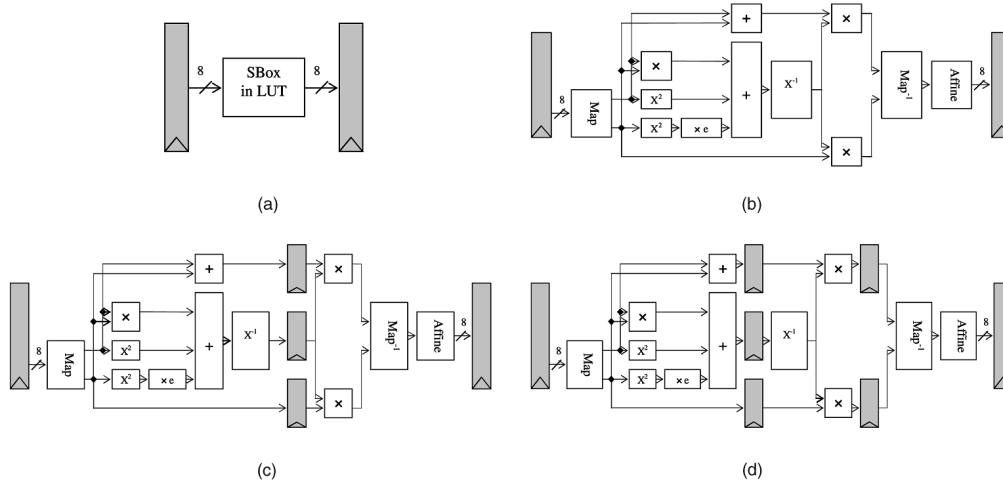


Figura 2.12: Implementações S-box realizadas em [2]

Em [2] foram realizadas várias implementações da cifra AES, explorando o *trade-off* entre a área e taxa de transferência para implementação ASIC em tecnologia CMOS de $0.18\mu\text{m}$, tendo alcançado entre 30 a 70 Gbits/s dependendo da implementação realizada. Foram utilizadas diferentes estratégias de implementação, tais como o uso de LUTs para o cálculo das S-box ou executando o cálculo das S-box durante a execução em $\text{GF}(2^4)$ utilizando diferentes estágios de *sub-pipelining*, assim como o cálculo das subchaves pela transformação key expansion *offline* e *online*. As várias implementações realizadas da transformação SubBytes são apresentadas na figura 2.12, onde a) corresponde a implementação usando LUT, b) implementação sem *sub-pipelining*, c) dois estágios de *sub-pipelining* e d) três estágios de *sub-pipelining*. Realizaram a comparação das várias implementações em relação a área utilizada e taxa de transferência máxima conseguida.

Nas figuras 2.13 e 2.14, são apresentados os resultados da comparação das várias implementações em relação à área utilizada e taxa de transferência máxima conseguida usando o cálculo das subchaves *online* e *offline*.

Chegaram à conclusão de que, usando 3 estágios de *pipelining* dentro da transformação SubBytes conseguem diminuir a área em 32%. Concluíram também que como a chave de encriptação se mantém constante durante uma sessão, ou seja, as subchaves de cada round são constantes durante uma sessão inteira, ao efetuar o cálculo das subchaves para cada round no modo *offline* é conseguida dessa forma uma redução da área utilizada em 28%. Com a combinação desses dois métodos de implementação foi possível uma redução de 48% em relação a uma implementação com a mesma taxa de transferência utilizando *LUT* efetuando o cálculo das subchaves *online*.

Em [14] realizou-se uma implementação em que todas as rondas foram instanciadas e com utilização *sub-pipelining*, isto é, com registos dentro de cada ronda, dividindo dessa forma o tamanho do caminho crítico, obtendo dessa forma uma maior frequência de relógio. Cada ronda tem

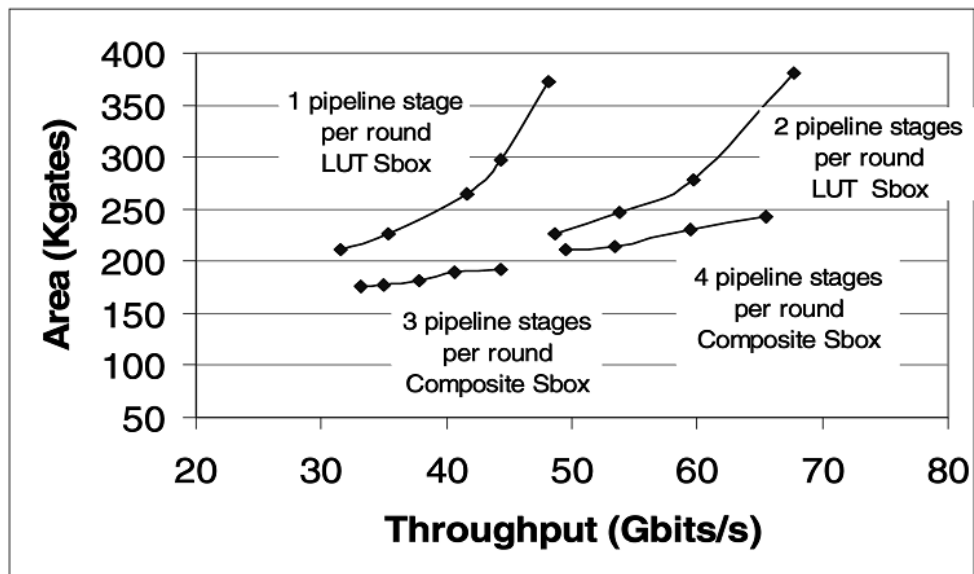


Figura 2.13: Área vs Taxa de transferência com *Key Expansion online*[2]

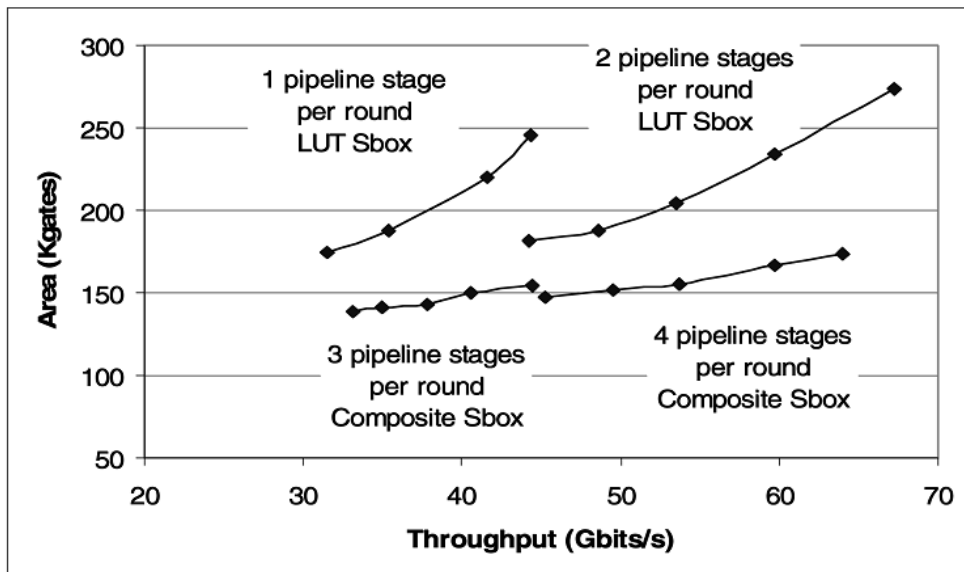


Figura 2.14: Área vs Taxa de transferência com *Key Expansion offline*[2]

4 estágios, 3 estágios para a computação das S-box sobre $GF(2^{128})$ e um estágio para o resto das transformações. Desta forma foi conseguido um débito de 54 Gbps para uma área equivalente de 272 kgates e o valor de débito de 38.44 Gbps com uma área equivalente de 262 kgates.

Na implementação em [3], foi realizada a implementação em ASIC de 5 arquiteturas com as S-box implementadas em $GF(2^2)$, duas arquiteturas com um caminho de dados de 32 bits, uma com 5 ciclos de relógio por rodada e outra com 4. Duas arquiteturas com caminho de dados de 64 bits, com 3 e 4 ciclos de relógio por rodada respectivamente e uma implementação com caminho de

dados de 128 bits com 1 ciclo de relógio por ronda 2.15.

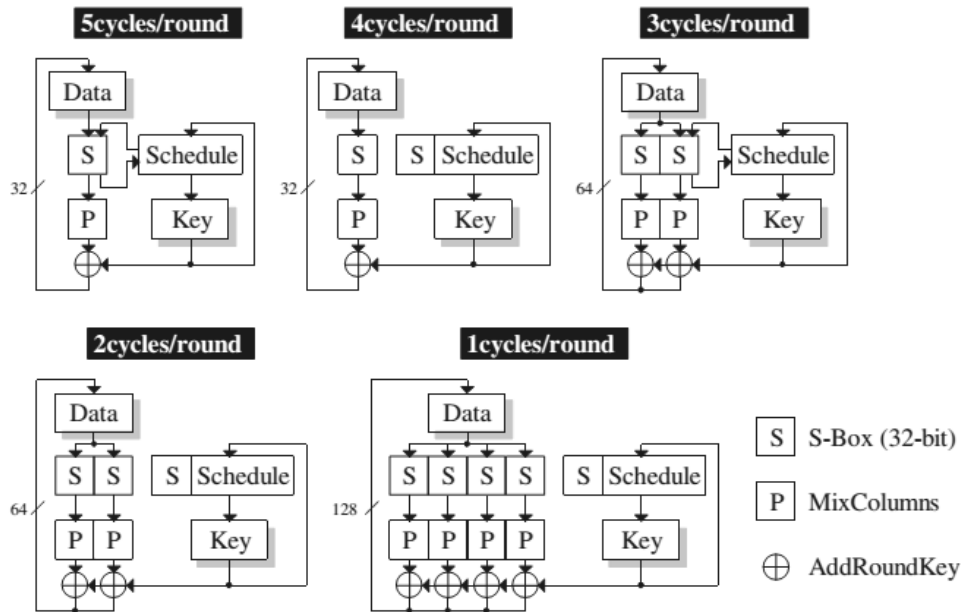


Figura 2.15: Arquiteturas implementadas em [3]

Na implementação com 5 ciclos de relógio por ronda, são instanciadas 4 S-box, capazes de calcular simultaneamente 32 bits, as S-box são partilhadas com o cálculo da chave de ronda através de um *MUX* 5x1, sendo utilizadas as S-box no primeiro ciclo de relógio para o cálculo da chave de ronda. Enquanto a chave de ronda é calculada é realizada a operação ShiftRows, tendo invertido a ordem de execução com a transformação SubBytes. Os 4 ciclos de relógio restantes são usados no cálculo da transformação SubBytes e Mixcolumns. Na implementação com 4 ciclos de relógio, a única diferença é a instanciação de mais 4 S-box para a geração das chaves de ronda, sendo esta feita em paralelo e demorando desta forma menos um ciclo de relógio a completar uma ronda. As outras arquiteturas são variações destas, apenas com o uso de caminhos de dados com diferentes dimensões e consequentemente, um maior número de S-box instanciadas e menor número de ciclos de relógio para a execução de uma ronda. Com este método de otimização para as S-box, foi conseguido $\frac{1}{4}$ da área ocupada em comparação com o uso de LUTs e uma performance 20% melhor em comparação com a implementação das S-box em $GF(2^4)$.

A implementação com um maior débito conseguido foi a implementação com o caminho de dados de 128 bits e 1 ciclo relógio por ronda, em que obtiveram um débito de 2.6 Gbps para uma área equivalente de 21.3 Kgates e uma frequência de 224 MHz em tecnologia CMOS de 110nm.

[4] [15] propõem ainda a decomposição de $GF(2^4)$ em $GF(2^2)$ e as operações $GF(2^2)$ em $GF(2)$, podendo desta forma toda a implementação S-box ser feita usando simplesmente bit *XOR* para adição e bit *AND* para a multiplicação. A inversão em $GF(2)$ do valor 1 é 1 e a inversão de 0 não existe. Por isso toda a inversão em $GF(2^8)$ pode ser decomposta em circuitos lógicos

usando unicamente gates *XOR* e gates *AND*. O principal objetivo deste artigo é obter uma grande taxa de transferência, tendo para isso recorrido a uma implementação *full pipelined*. Foi feita uma implementação com *pipeline* balanceado, em que foi calculado o atraso de cada operação básica, e foram aplicados os registros de *pipelining* de modo a balancear o tempo entre cada registro de *pipeline*. A implementação foi realizada usando uma FPGA Virtex 5. Os resultados são apresentados na figura 2.16.

Pipeline stages	Throughput (Gbps)	Area (slice)	BRAM (%)	Latency (ns)
1	20	8,800	0	6.25
2	27	9,086	0	4.67
4	48	10,561	0	2.777
8	67	13,649	0	1.905

Figura 2.16: Resultados obtidos por [4]

Em [5] é realizada uma implementação do algoritmo AES modo contador em tecnologia CMOS 90nm. A arquitetura desenvolvida consiste em duas rondas instanciadas, com 8 estágios de *pipelining*, 6 dos quais no cálculo das S-box, sendo estas realizadas em $GF(2^2)$ figura 2.17. Com esta implementação foi atingida uma frequência de 769MHz, com um débito de 19.6Gbps e uma área equivalente de 46.3 Kbytes

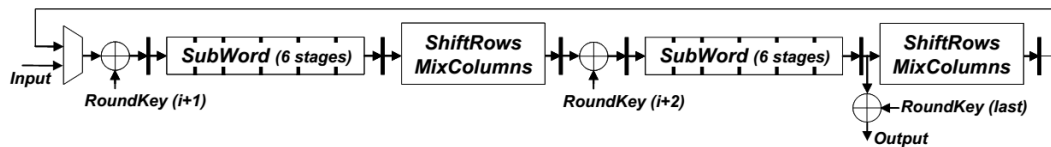


Figura 2.17: Arquitetura realizada em [5]

Em [6] foi feita uma implementação em FPGA Xilinx Virtex 5 para especificações de cinema digital. A implementação das S-box foi realizada usando lógica combinacional, segundo o autor esta implementação tem a vantagem de se obter uma grande redução de área, ocupando cada S-box 32 LUTs em comparação com as implementações usando os blocos de memória RAM da FPGA. A substituição é feita em um ciclo de relógio, obtendo dessa forma uma baixa latência do circuito em comparação com as S-box implementadas em $GF(2^8)$. As implementações para especificações de cinema digital cifram grande quantidade de dados usando a mesma chave mestra, por esta razão as chaves de ronda são pré-calculadas e guardadas em registros, permitindo dessa forma reduzir a área. As S-box usadas para a cifração do modulo e para o cálculo das chaves de ronda são as mesmas, sendo usado um *MUX* para alternar entre ambas. Na figura 2.18 é apresentada a arquitetura realizada neste artigo.

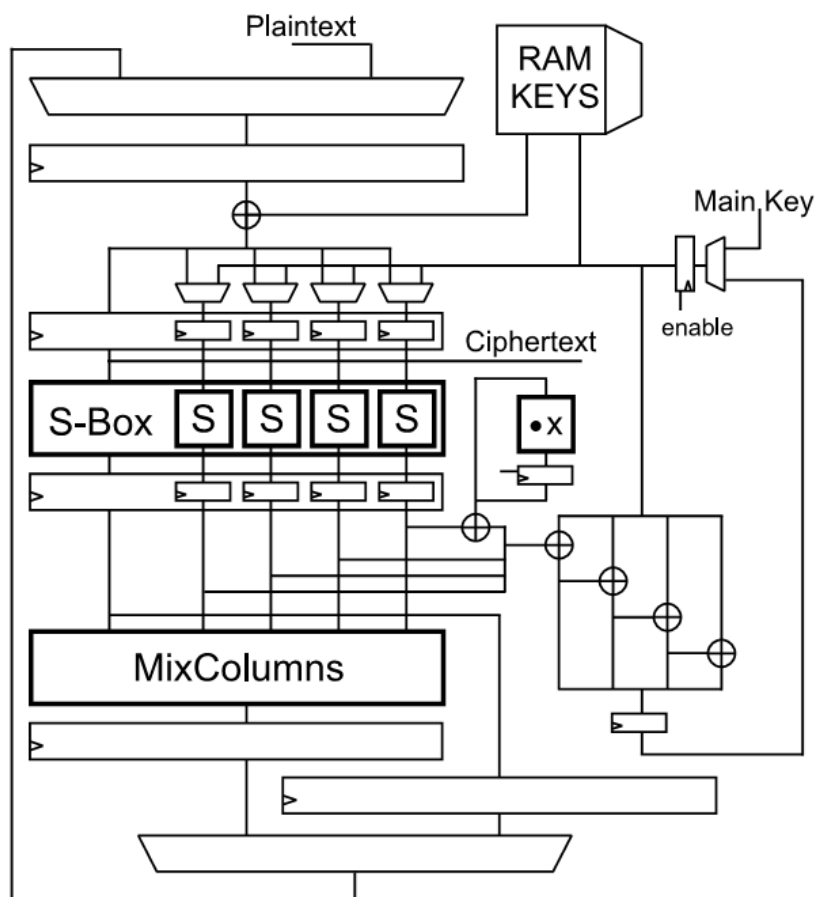


Figura 2.18: Arquitetura da implementação AES em [6]

Como podemos ver pela imagem, em cada ronda é feito *sub-pipelining*, ou seja, cada ronda está dividida por registros. Neste caso, cada ronda demora quatro ciclos de relógio para ser completada. Os resultados obtidos por esta implementação são apresentados na figura 2.19, onde é apresentado o resultado obtido, com diferentes FPGAs. Na FPGA Virtex 5 da Xilinx, cada *slice* contém 4 LUTs, enquanto que nas outras plataformas, apenas contém 2 LUTs. Para poder fazer uma comparação justa, os valores entre parêntesis são o dobro dos obtidos, permitindo uma melhor comparação com os valores das outras arquiteturas. O autor também realizou a implementação do algoritmo de decifração usando as transformações inversas, apresentando esses resultados a seguir aos obtidos para a cifração, separados por uma barra.

Device	Slices	BRAM	Freq. (MHz)	Thr. (Gbps)	Thr. / Area (Mbps/slice)
Virtex-5	400 / 550 (800 / 1100)	0	350	4.1	10.2 / 7.4
Virtex-4*	700 / 1220	8	250	2.9	4.1 / 2.3 *
Spartan-3	1800 / 2150	0	150	1.7	0.9 / 0.8

Figura 2.19: Resultado obtidos em [6]

Capítulo 3

Arquiteturas Desenvolvidas

3.1 Introdução

Neste capítulo são apresentados os passos dados e as decisões tomadas na implementação e desenvolvimento das arquiteturas do algoritmo de cifra AES-CTR. Como foi visto nos objetivos, esta arquitetura é aplicada no caminho de dados áudio/vídeo que tem uma cadência de 24 bits por ciclo de relógio, pretendendo-se obter um débito de 14.4Gbps em ASIC.

Pela especificação, a cifra AES-CTR recebe blocos de 128 bits encriptando-os com uma chave também ela de 128 bits. Como o bloco de cifragem é aplicado no caminho de dados de áudio/vídeo com uma cadência de 24 bits por ciclo de relógio, isto significa que o nosso bloco tem de ser capaz de efetuar uma cifra a cada 5 ciclos de relógio. Para o conseguir, foram desenvolvidas arquiteturas *pipelined*, através da divisão do circuito combinacional em andares com a introdução de registos síncronos com o sinal de relógio. Desta forma é possível libertar os estágios para a receção de novos dados.

Durante a realização deste trabalho, foram desenvolvidas quatro Arquiteturas diferentes, e foi analisado o seu desempenho, assim como a área consumida e o consumo de energia.

3.2 Interface e Requisitos do Projeto

O primeiro passo para o levantamento dos requisitos e definição do interface, foi o estudo do algoritmo de cifra AES, as transformações efetuadas pelo mesmo e o seu modo de funcionamento em modo contador. Pelo documento de especificação [9], o algoritmo AES cifra apenas blocos de dados de 128 bits, utilizando uma chave também ela de 128 bits. Como o caminho de dados onde se insere o bloco, como já foi visto, tem uma cadência de 24bits por ciclo de relógio, temos como requisito que a implementação seja capaz de cifrar um bloco de dados a cada 5 ciclos de relógio, perfazendo dessa forma os 24bits por ciclo de relógio.

Sinais	Tamanho	Entrada/Saída	Descrição
iclk	1	Entrada	Sinal de Relógio
irst_n	1	Entrada	Sinal de Reset negado
i_en	1	Entrada	Habilita/desabilita o bloco
istart	1	Entrada	Inicia a encriptação, carrega os valores de itext e ikey
itext	128	Entrada	Dados de entrada a serem cifrados
ikey	128	Entrada	Chave de cifragem
odone	1	Saída	Encriptação completa, dados em otext validos
otext	128	Saída	Dados cifrados
oerror	1	Saída	Sinal de erro

Tabela 3.1: Sinais do bloco de Alto nível

3.3 Interface de Alto Nível

A partir dos requisitos e especificações do sistema foi definido o interface do nosso bloco tabela 3.1. Como podemos ver pela tabela, o sinal *iclk*, é o sinal de relógio da arquitetura, o qual efetua a sincronização dos sinais internos e do projeto com o restante bloco HDCP. O sinal de *reset*, o qual reinicia o nosso circuito para o estado inicial. O sinal *i_en*, serve para habilitar ou desabilitar o bloco. A necessidade deste sinal prende-se com o facto do protocolo HDCP, permitir a transmissão de conteúdo não cifrado, sendo efetuado um *bypass* ao bloco AES. O sinal *itext*, é um sinal de entrada que recebe o bloco de 128 bits (contador) a ser cifrado. O sinal *ikey*, sinal de entrada, recebe a chave de 128 bits a ser usada durante a cifragem. O sinal *otext* é um sinal de saída de 128bits que corresponde ao bloco cifrado. O sinal *istart*, é um sinal de entrada de 1 bit, que quando ativo, significa que os dados presentes em *itext* e *ikey* são válidos e dá início à sua cifragem. O sinal *odone*, é um sinal de saída que, quando ativo, significa que a cifra está pronta e os dados presentes em *otext* são válidos. O sinal de erro *oerror*, é um sinal de saída de 1bit, o qual quando ativo informa que ocorreu um erro no *stream* de dados, monitorizando o sinal de entrada *istart* e o sinal de saída *otext*, verificando que não é recebido nenhuma ativação do sinal *istart* antes dos 5 ciclos de relógio, o que provocaria uma cifragem errada dos dados.

3.4 Plano de Verificação

A verificação é um processo usado para demonstrar o funcionamento correto de um projeto, este é conhecido por verificação lógica ou simulação. O propósito da verificação é o de assegurar que o projeto cumpre a função pretendida.

A partir da especificação e dos requisitos do projeto, foi elaborado um plano de verificação, a partir do qual foi posteriormente criado o ambiente de teste. O plano de verificação permite perceber se a Arquitetura desenvolvida apresenta o comportamento esperado.

Na tabela 3.2 podemos ver as características identificadas e sobre as quais foi desenvolvido o ambiente de teste. Durante o teste temos de verificar que são recebidos os blocos a serem cifrados a cada 5 ciclos de relógio. Devido a especificação *HDCP*, na qual o bloco AES está

Características a Verificar	Critério de Verificação
Entrada de dados a cada 5 ciclos de relógio	Verificar o sinal <i>istart</i>
<i>i_en</i> = 1, o design tem de estar habilitado	Verificar o funcionamento normal do bloco, verificar os valores das saídas
<i>i_en</i> = 0, design tem de estar desabilitado	Verificar que todas as saídas estão a zero.
Sinal de Reset activo	Verificar que todas as saídas estão a zero.
Verificar a funcionalidade do design	Quando sinal <i>odone</i> =1, Comparar o valor de saída <i>otext</i> com o valor esperado.
Saída de dados a cada 5 ciclos de relógio	verificar o sinal <i>odone</i>
Detecção de erro	entrada de dados fora dos requisitos, verificar <i>oerror</i> .

Tabela 3.2: Plano de Verificação do design efetuado

num caminho de dados de 24 bits/ciclo de relógio, para executar a cifragem a cada 5 ciclos de relógio necessitamos de cifrar 120 bits, sendo esta feita com os 120 bits mais significativos do bloco de saída do AES, descartando os 8 bits menos significativos. Se a cifra fosse obtida a cada 6 ciclos de relógio, necessitaríamos de cifrar 144 bits, o que não é possível, pois o AES cifra blocos de tamanho fixo de 128 bits, para testar esta característica é verificado o sinal *istart*. Para testar a funcionalidade do bloco, são aplicados à entrada *itext* e *key* valores conhecidos de dados e de chave respetivamente e é comparado o resultado obtido com o valor esperado. Para proceder a esta verificação, após a aplicação dos vetores de teste, quando o sinal *odone* ficar ativo, ou seja, a cifragem esteja feita, é verificado o resultado. Outra característica a verificar é o correto funcionamento do sinal de *enable*: quando este sinal estiver ativo, o bloco está habilitado e como tal, deve ter um funcionamento normal. Quando desativo, os registos devem ser zerados. A verificação é feita analisando se os sinais de saída estão zerados enquanto o sinal estiver desativo. O mesmo teste é aplicado ao sinal de *reset*. Do mesmo modo que é necessário garantir que os dados de entrada são recebidos a cada 5 ciclos, é necessário também garantir que os blocos de saída cifrados são obtidos com a mesma cadência. Para testar é verificado se o sinal *odone* fica ativo a cada 5 ciclos de relógio. Para o teste do sinal de erro, são induzidos erros no projeto, tais como ativação do sinal *istart*, com um intervalo superior a 5 ciclos de relógio, sendo depois verificado se o sinal de erro ficou ativo.

3.5 AES4box

A arquitetura AES4box é uma arquitetura com *loop* desdobrado, em que todas as 10 rondas são instanciadas. O cálculo das chaves de ronda é feito durante a execução figura 3.1.

Nesta implementação, todas as transformações, incluindo o cálculo da chave da ronda são feitas dentro do módulo ronda, este recebe como entrada a matriz de estado de 128 bits com o valor intermédio da cifra, a chave de ronda atual, também ela de 128 bits, a constante de ronda *rcon* de 8 bits, o sinal de *istart* de 1 bit e como saída, o sinal *odone* de 1bit, a matriz de estado intermédia da ronda seguinte, assim como a chave de ronda seguinte, ambos de 128 bits. O sinal de

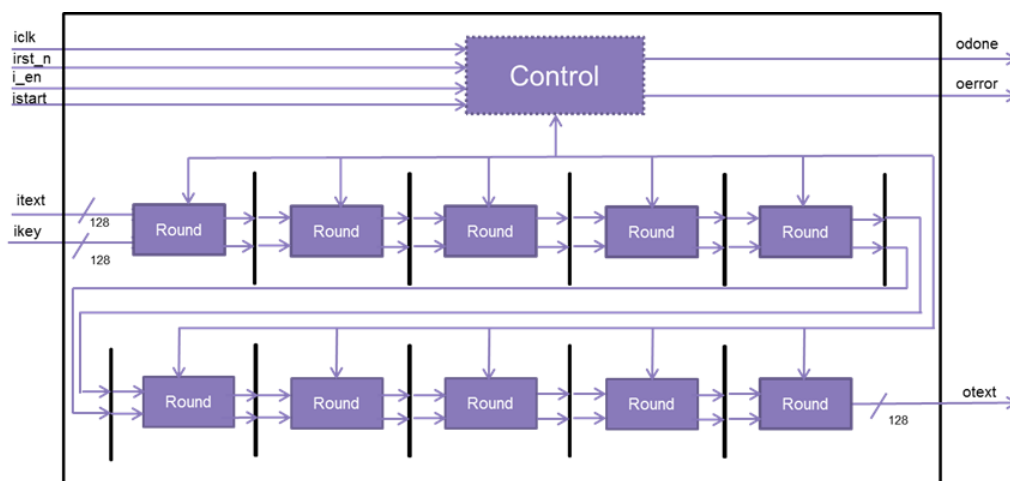


Figura 3.1: Arquitetura AES4box Top

istart e *odone*, correspondem ao sinal que dá início à execução do bloco e de que a transformação está pronta e os dados à saída do bloco são válidos, respetivamente.

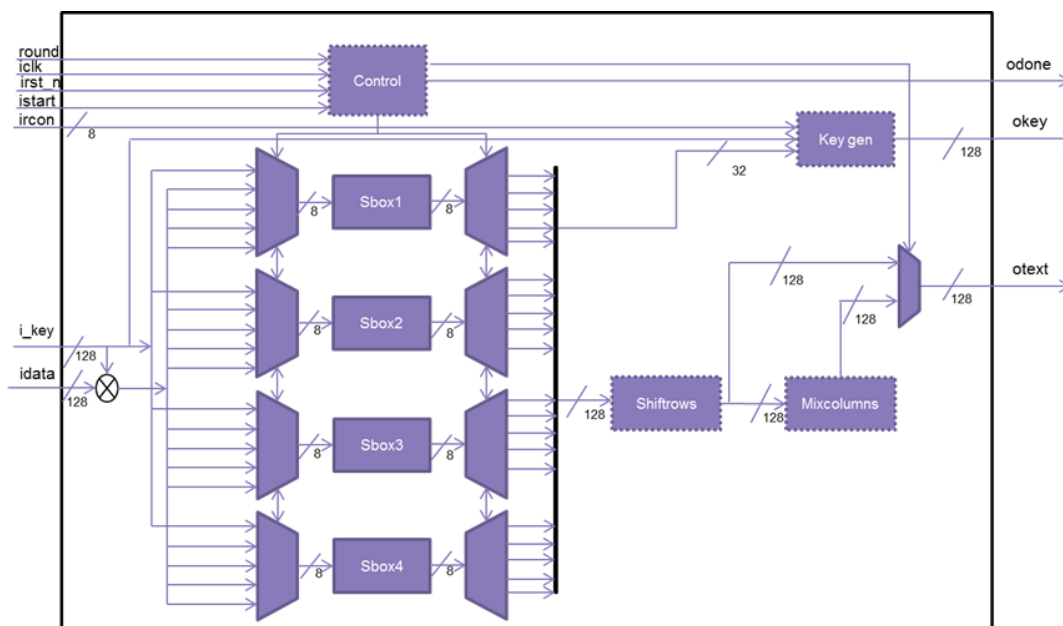


Figura 3.2: Ronda da Arquitetura AES4box

Cada ronda figura 3.2, demora 5 ciclos de relógio para fazer as transformações, após os quais, o valor da cifra intermédia resultante é guardada em registos e aplicada à entrada da ronda seguinte. Na transformação SubBytes é feita uma substituição S-box orientada ao byte, sendo necessário a realização de 16 transformações. Como as S-box consomem bastante área, a ideia desta arquitetura foi a de reutilizar as S-box de maneira a otimizar a arquitetura e minimizar a área utilizada, instanciando 4 S-box em vez das 16 e reutiliza-las utilizando um multiplexador e um

desmultiplexador respectivamente na entrada e saída das mesmas. O cálculo da chave de ronda também necessita de realizar uma transformação SubBytes em 4 bytes na realização da função $g()$. Desta forma, as S-box instanciadas são usadas durante 4 ciclos de relógio para a transformação dos dados e um ciclo de relógio para a transformação necessária para o cálculo da chave de ronda. A implementação do multiplexador é apresentada a seguir, onde a variável $wdata[0]$ é a entrada para uma das S-box, $rcount$ é um registo que controla a posição do multiplexador e do desmultiplexador, sendo este incrementado a cada ciclo de relógio. O seu valor inicial é zero, sendo que quando $rcount$ tem o valor de zero, é feito o *assign* de um byte da *word* mais significativa da chave de ronda e quando $rcount$ assume outros valores $wdata[0]$ assume o valor de um dos bytes que constituem a matriz de dados.

```
assign wdata[0] = (rcount == 3'd0) ? wword[3][31:24] :
                (rcount == 3'd1) ? wdataux[0] :
                (rcount == 3'd2) ? wdataux[4] :
                (rcount == 3'd3) ? wdataux[8] : wdataux[12];
```

A S-box, recebe como entrada a variável $wdata[0]$, e tem como saída $wbox[0]$, sendo o valor $wbox[0]$ a entrada para o desmultiplexador

```
AES_SBox u1_AES_SBox(
    .idata    (wdata[0]),
    .odata    (wbox[0])
);
```

A implementação da S-box foi realizada usando lógica combinacional através da substituição direta do valor corresponde por *look-up-table*. Em baixo são apresentados alguns valores de substituição.

```
module AES_SBox(
input      [7:0] idata ,    //SBox input byte
output reg [7:0] odata     //SBox output
);
```

```
always @(*)
```

```
begin
```

```
    case (idata)                //Look Up Table
        8'h00: odata = 8'h63;
        8'h01: odata = 8'h7c;
        8'h02: odata = 8'h77;
        8'h03: odata = 8'h7b;
        8'h04: odata = 8'hf2;
        8'h05: odata = 8'h6b;
        8'h06: odata = 8'h6f;
        8'h07: odata = 8'hc5;
```

```

8'h08: odata = 8'h30;
8'h09: odata = 8'h01;
8'h0a: odata = 8'h67;
8'h0b: odata = 8'h2b;
8'h0c: odata = 8'hfe;
8'h0d: odata = 8'hd7;
8'h0e: odata = 8'hab;

```

O demultiplexador 1x4 implementado é apresentado em baixo, sendo este síncrono com o sinal de relógio.

```

case (rcount)
  3'd0: begin
    rbyte[0] <= wbox[0];
    rbyte[1] <= wbox[1];
    rbyte[2] <= wbox[2];
    rbyte[3] <= wbox[3];
  end
  3'd1: begin
    rstate[0] <= wbox[0];
    rstate[1] <= wbox[1];
    rstate[2] <= wbox[2];
    rstate[3] <= wbox[3];
  end
  3'd2: begin
    rstate[4] <= wbox[0];
    rstate[5] <= wbox[1];
    rstate[6] <= wbox[2];
    rstate[7] <= wbox[3];
  end
  3'd3: begin
    rstate[8] <= wbox[0];
    rstate[9] <= wbox[1];
    rstate[10] <= wbox[2];
    rstate[11] <= wbox[3];
  end

  default: ;
endcase

```

Durante o primeiro ciclo de relógio são calculadas as transformações S-box usadas no cálculo da chave, sendo o valor resultante guardado durante a flanco ascendente do ciclo de relógio

seguinte e ao mesmo tempo incrementado o registo *rcount*. Durante o segundo ciclo é feita a transformação S-box nos primeiros 4 bytes de dados a serem cifrados e em paralelo é feito o cálculo da chave seguinte. No flanco ascendente do 5º ciclo de relógio são guardadas as transformações correspondentes aos penúltimos 4 bytes, incrementado *rcount* para o valor 4 e é ativada a *flag odone*, sinalizando que a cifragem está realizada. Durante o 5º ciclo de relógio são realizadas as últimas substituições S-box, não sendo estas guardadas em registos, permitindo dessa forma concluir as transformações de ronda neste ciclo de relógio. Isso é conseguido porque as transformações ShiftRows e Mixcolumns são implementadas usando lógica combinacional, sendo apenas os dados finais da ronda capturados a saída do módulo ronda, no topo durante o flanco ascendente do 5º ciclo de relógio e alimentados para a ronda seguinte.

Um excerto da operação Shiftrows é apresentado em baixo. Esta foi realizada através de uma reordenação dos bytes, de modo a obter o resultado correspondente a esta operação, onde a variável *rstate*, são os registos onde foram sendo guardadas as substituições feitas na etapa anterior e onde se pode ver também, que os últimos 4 bytes foram atribuídos diretamente.

```
assign wstate_aux[0] = rstate[0];
assign wstate_aux[1] = rstate[5];
assign wstate_aux[2] = rstate[10];
assign wstate_aux[3] = wbox[3];
assign wstate_aux[4] = rstate[4];
assign wstate_aux[5] = rstate[9];
assign wstate_aux[6] = wbox[2];
```

Após a transformação ShiftRows é realizada a operação Mixcolumns. O código em baixo mostra como foi executada a multiplicação por dois de cada byte em campos finitos de galois (2^8). É testado o bit mais significativo para verificar se vai ocorrer *overflow*. Se este estiver a um, a multiplicação é feita com uma rotação à esquerda seguida de uma operação *XOR bitwise* com o polinómio redutor, que no algoritmo de cifra AES é usado o valor hexadecimal 1B. Caso não esteja a 1, apenas é feita a rotação à esquerda.

```
assign wstate_mul[i] = (wstate_aux[i][7]) ?
((wstate_aux[i]<<1) ^ 8'h1b) :(wstate_aux[i]<<1);
```

Em baixo é apresentado uma parte do código RTL para o cálculo da transformação MixColumns do primeiro byte da primeira coluna da matriz, sendo este o byte 0. Esta operação é composta pela multiplicação dele próprio, mais a soma da multiplicação por 3 do byte 1, com a soma do byte 2 e do byte 3. A multiplicação por três foi conseguida através de um *XOR bitwise* do próprio valor com a variável auxiliar *wstate_mul[i]*, que contém o valor da multiplicação por 2.

$$S'_{0,j} = (2 \cdot S_{0,j}) \oplus (3 \cdot S_{1,j}) \oplus S_{2,j} \oplus S_{3,j}$$

```
assign wstate[0] = wstate_mul[0] ^ (wstate_mul[1] ^ wstate_aux[1])
^ wstate_aux[2] ^ wstate_aux[3];
```

Durante o primeiro ciclo de relógio, é feita a adição da chave de ronda com os dados intermédios, através de um *XOR bitwise* e é realizada a primeira substituição S-box dos 4 bytes para o cálculo da chave de ronda, sendo o resultado guardado em registos, após o qual o bloco de controlo altera a posição do *MUX* e *DEMUX*

A implementação do cálculo da chave de ronda é apresentado a seguir, onde o cálculo da *word* mais significativa da nova chave de ronda é feita pela soma da *word* mais significativa da chave de ronda anterior, com a *word* menos significativa, após sofrer a transformação *g()*.

A transformação *g()* consiste na rotação circular de um byte da *word*, seguida da transformação *SubBytes* aplicada aos 4 bytes que constituem essa *word*. Sendo depois adicionada a constante de ronda *rcon* ao byte mais significativo. Na implementação desta função, foi trocada a ordem entre a transformação *SubBytes* e a rotação, sendo a substituição dos 4 bytes feita em primeiro lugar. As variáveis *rbyte[0]..[3]*, contêm o resultado da substituição S-Box, em seguida como se pode ver na atribuição feita a *wword[4]*, sendo este o valor da *word* mais significativa da nova chave. Os bytes foram reordenados, adicionado a constante de ronda *rcon* ao byte mais significativo, sendo depois concatenados e adicionados a *word* mais significativa anterior da chave de ronda anterior. O cálculo das *words* menos significativas, começa por ordem descendente, onde a *word* a ser calculada é resultado da soma da *word* mais significativa calculada imediatamente antes desta com a *word* da chave de ronda anterior correspondente à mesma posição.

```
// calculo da word mais significativa da key
assign wword[4] = ikey[127:96] ^
    {(rbyte[1] ^ ircon), rbyte[2], rbyte[3], rbyte[0]};

// calculo das words menos significativas da key
generate
for (i=1; i<=3; i=i+1) begin: word
    assign wword[4+i] = wword[4+i-1] ^ wword[i];
end
endgenerate
```

O bloco ronda, também recebe um sinal de entrada que controla um *MUX* 2x1, para quando for a última ronda não ser executada a transformação *Mixcolumns*.

São usadas 4 S-box por ronda, num total de 40 S-box. O circuito apresenta uma latência inicial de 50 ciclos de relógio.

3.6 AES8box

Na Arquitetura AES8box são instanciadas cinco rondas, as quais são usadas duas vezes cada através da realimentação da saída de cada ronda. Após um dado bloco de dados ter usado uma ronda instanciada duas vezes, passa para a ronda seguinte, recebendo a ronda um novo bloco

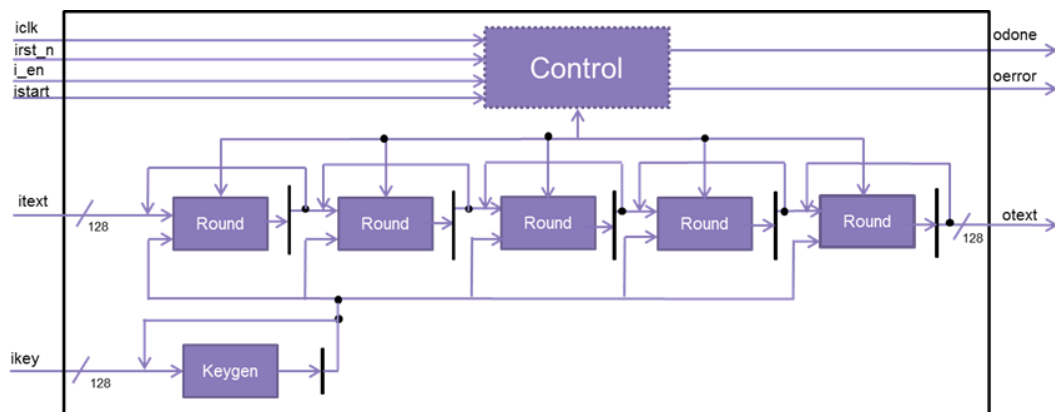


Figura 3.3: Topo da Arquitetura AES8box

de dados e assim sucessivamente, tendo cada bloco de dados de repetir cada ronda duas vezes, perfazendo dessa forma as dez rondas da especificação do algoritmo AES para uma chave de cifra de 128 bits figura 3.3. O cálculo das chaves de ronda é feito à parte, no módulo instanciado no topo Keygen. Cada transformação de ronda é executada em 2 ciclos de relógio, sendo cada ronda utilizada pelo mesmo bloco de dados durante 4 ciclos de relógio, tendo um 1 ciclo de relógio inativo ate receber novo bloco de dados. Como se pode ver na figura 3.4 a instancia ronda é constituída por 8 S-box instanciadas, utilizando também um *MUX* e *DEMUX* na entrada e saída de cada S-box, sendo a cada ciclo de relógio feita a transformação de 8 bytes, demorando 2 ciclos de relógio a fazer os 16 bytes. Depois são executadas ainda durante o segundo ciclo de relógio as transformações ShiftRows e MixColumns, tal como na arquitetura anterior.

Em baixo podemos ver a instanciação de dois dos cinco blocos de ronda, assim como a instanciação do bloco de calculo das chaves de ronda.

```
AES_keygen u1_AES_keygen(
    .iclk      ( iclk ),
    .rst_n     ( rst_n ),
    .ircon     ( rrcon ),
    .ivalid    ( rvkey ),
    .ikey      ( rkin ),
    .ovalid    ( wkvalid ),
    .okey      ( wokey )
);

// Round 1
    AES_round u1_AES_round(
        .iclk      ( iclk ),
        .rst_n     ( rst_n ),
```

```

        .ivalid    ( rvalidin [0] ),
        .ilround   ( ROUND ),
        .idata     ( rstate [0] ),
        .ovalid    ( wstvalidout [0] ),
        .odata     ( wauxi [0] )
    );
//Round 2
    AES_round u2_AES_round(
        .iclk      ( iclk ),
        .irst_n    ( irst_n ),
        .ivalid    ( rvalidin [1] ),
        .ilround   ( ROUND ),
        .idata     ( rstate [1] ),
        .ovalid    ( wstvalidout [1] ),
        .odata     ( wauxi [1] )
    );

```

O cálculo das chaves de ronda é realizado apenas uma única vez durante o cálculo do primeiro bloco de cifra, sendo as chaves guardadas em registos à medida que são calculadas. Esses registos são usados no cálculo dos blocos seguintes. Estas são calculadas em paralelo com a execução do cálculo do primeiro bloco cifrado. Desta forma é evitada a latência inicial causada, caso as chaves fossem pré-calculadas antes do início da cifragem. Como cada ronda nesta implementação demora 2 ciclos de relógio a ser executada, para o cálculo da chave de ronda em paralelo com a execução da cifragem, necessitamos uma nova chave de ronda com a mesma cadência. Por isso foram apenas instanciadas 2 S-box, as quais são reutilizadas uma vez para o cálculo da função $g()$, permitindo desta forma economizar 2 S-box. A cifragem dos blocos seguintes obtêm as chaves de ronda dos registos onde elas foram guardadas.

Em baixo é apresentada uma parte da implementação da reutilização das rondas, onde é mostrada a utilização da primeira ronda instanciada. A ronda, tem como sinais de entrada além do sinal de relógio do sistema e de *reset*, os dados $rstate[0]$, sob os quais vão ser realizadas as transformações. O sinal $rvalidin[0]$, dá o sinal de início da ronda, o sinal *ROUND*, controla o multiplexador 2x1 que realiza o *bypass* à transformação Mixcolumns caso este sinal esteja a 1, indicando que é a última ronda do algoritmo AES e a transformação Mixcolumns não é executada. O sinal $wstvalidout[0]$, indica que a ronda está concluída e os dados de saída da ronda $wauxi[0]$ são válidos. Quando é recebido o sinal *istart*, entrada de um novo bloco de dados a ser cifrado, a este é adicionada a chave de cifragem original, sendo o resultado aplicado à primeira ronda. Quando o sinal $wstvalidout[0]$ é ativado a ronda está pronta. A variável $raux[0]$ é a variável auxiliar que controla se é a primeira vez que a ronda foi executada. Caso este sinal de 1 bit esteja a 1. O resultado da ronda depois de adicionada a chave de ronda correspondente, é aplicado à instância da ronda seguinte $rstate[1]$ e atribuído à variável $raux[0]$ o valor de zero e assim sucessivamente.

```

if(istart) begin
    rstate[0]    <= itext^ikey;
    rvalidin[0] <= 1'b1;
end
else begin
    if((wstvalidout[0]) && (!raux[0])) begin
        rstate[0]    <= wauxi[0] ^ rkey[1];
        raux[0]      <= 1'b1;
        rvalidin[0] <= 1'b1;
    end
    end
    if((wstvalidout[0]) && (raux[0])) begin
        rstate[1]    <= wauxi[0] ^ rkey[2];
        raux[0]      <= 1'b0;
        rvalidin[1] <= 1'b1;
    end
    if((wstvalidout[1]) && (!raux[1])) begin
        rstate[1]    <= wauxi[1] ^ rkey[3];
        raux[1]      <= 1'b1;
        rvalidin[1] <= 1'b1;
    end
    if((wstvalidout[1]) && (raux[1])) begin
        rstate[2]    <= wauxi[1] ^ rkey[4];
        raux[1]      <= 1'b0;
        rvalidin[2] <= 1'b1;
    end
end

```

3.7 AES16box

Na figura 3.5 é apresentado o diagrama de blocos da arquitetura AES16box. Nesta arquitetura são instanciados dois blocos de ronda, cada ronda é repetida cinco vezes pelo mesmo bloco de dados a ser cifrado, antes de passar para o bloco de ronda seguinte. Na entrada do bloco ronda, encontra-se um multiplexador 2x1, em que a cada cinco ciclos de relógio é selecionada a entrada de novo bloco e nos outros quatro ciclos de relógio é selecionada a entrada da realimentação do bloco, perfazendo as cinco rondas. O mesmo acontece para o segundo bloco de ronda, contém um multiplexador 2x1 a entrada, selecionando uma novo bloco de dados a cada 5 ciclos de relógio e durante esse intervalo o bloco é realimentado quatro vezes, perfazendo as cinco rondas restantes das dez rondas da especificação, após as quais o bloco de dados se encontra cifrado e é ativado o sinal odone que indica que a cifragem está concluída. O módulo de cálculo das chaves de ronda implementado nesta arquitetura, executa o cálculo de uma chave de ronda por ciclo de relógio,

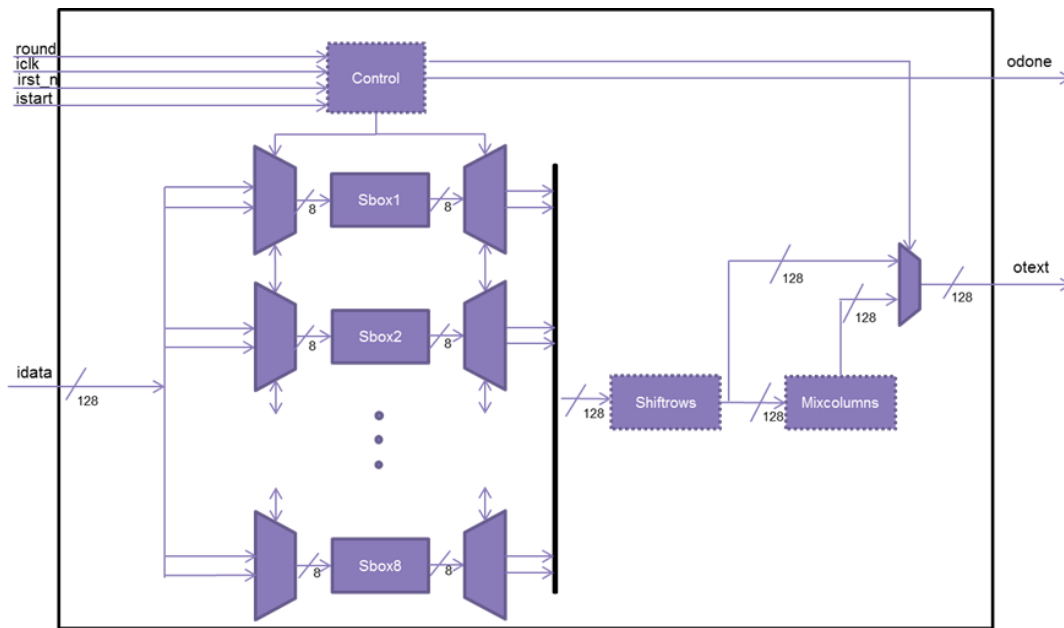


Figura 3.4: Ronda da Arquitetura AES8box

tendo sido instanciado as 4 S-box necessárias para a função $g()$. Esta implementação usa apenas lógica combinacional. As chaves de ronda são calculadas durante a cifragem do primeiro bloco de dados e guardadas em registros. Desta forma são usadas menos 4 S-box. Nesta arquitetura são usadas um total de 36 S-box.

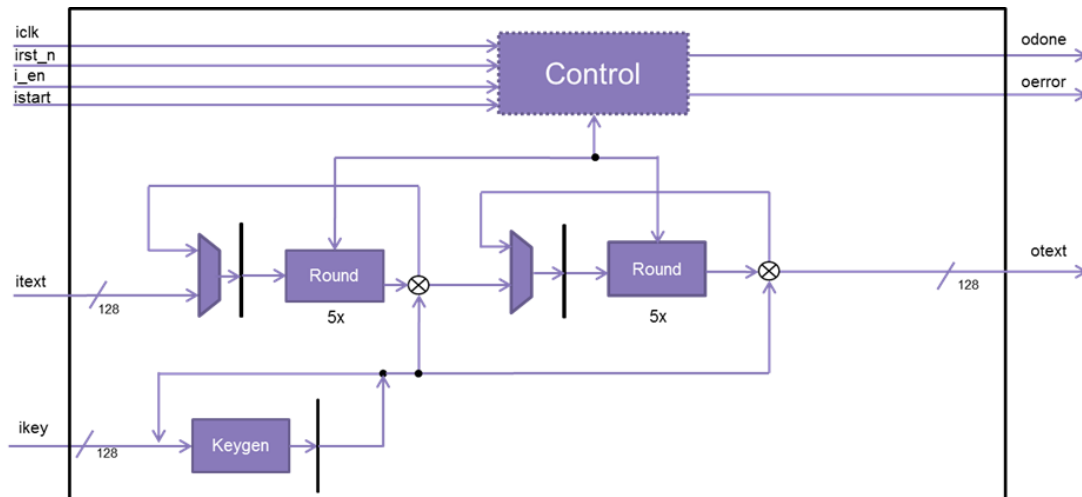


Figura 3.5: Topo da Arquitetura AES16box

Em baixo é apresentado a instanciação dos blocos. O bloco de cálculo das chaves AES_keygen é praticamente igual ao da arquitetura anterior, a única diferença é a instanciação de 4 S-box,

permitindo o cálculo de uma chave de ronda por ciclo de relógio, sendo este bloco puramente combinacional.

```
AES_keygen u1_AES_keygen(
    .ircon    (rrcon),
    .ikey     (rkin),
    .okey     (wokey)
);

//Round 1
    AES_round u1_AES_round(
        .ilround (ROUND),
        .idata   (rstate[0]),
        .odata   (wauxi[0])
    );

//Round 2
    AES_round u2_AES_round(
        .ilround (rlround),
        .idata   (rstate[1]),
        .odata   (wauxi[1])
    );
```

Em baixo é apresentado um pedaço da implementação do topo da arquitetura AES16box, o registo *rcount1*, controla o multiplexador de entrada das rondas instanciadas, sendo este incrementado a cada ciclo de relógio. *wauxi[i]* é o valor de saída de cada ronda e *rkey[i]* a chave de ronda correspondente. O sinal *rvalidin[1]* fica ativo ao fim de 5 ciclos de relógio, ativando dessa forma o segundo módulo de ronda, isto porque durante os primeiros 5 ciclos desde o início da cifragem, apenas é executada a primeira ronda. A cada ciclo de relógio, é atualizado o valor de entrada.

```
case(rcount1)
    3'd0: begin
        rstate[0] <= wauxi[0] ^ rkey[1];
        if(rvalidin[1])
            rstate[1] <= wauxi[1] ^ rkey[6];
    end
    3'd1: begin
        rstate[0] <= wauxi[0] ^ rkey[2];
        if(rvalidin[1])
            rstate[1] <= wauxi[1] ^ rkey[7];
    end
end
```

```

3'd2:  begin
    rstate[0] <= wauxi[0] ^ rkey[3];
    if(rvalidin[1])
        rstate[1] <= wauxi[1] ^ rkey[8];
end
3'd3:  begin
    rstate[0] <= wauxi[0]^rkey[4];
    if(rvalidin[1]) begin
        rlround    <= LROUND;
        rstate[1] <= wauxi[1] ^ rkey[9];
    end
end
endcase

```

O bloco de ronda desta arquitetura (figura 3.7), é um bloco puramente combinacional, demorando a sua execução um ciclo de relógio. São instanciadas 16 S-box, fazendo desta forma a transformação SubBytes de uma só vez ao contrario das arquiteturas anteriores. O resto do bloco é igual às outras arquiteturas, onde os 16bytes após a transformação SubBytes executam a transformação ShiftRows e Mixcolumns de acordo com a especificação. De igual modo, este também tem um multiplexador 2x1 para fazer *bypass* da transformação Mixcolumns, a qual não é executada na última ronda do algoritmo AES.

3.8 AES16box2

Esta Arquitetura é muito similar à anterior como se pode ver na figura 3.6. A única diferença está no modo como são geradas as chaves. Neste caso, são instanciados dois módulos de geração de chaves, sendo as chaves todas calculadas durante a execução, não sendo guardadas em registos. O módulo ronda e keygen são exatamente iguais à implementação anterior.

Cada instância keygen, é responsável pela geração das 5 chaves de ronda usadas em cada módulo ronda. Esta implementação usa um total de 40 S-box, 4 S-box adicionais do segundo módulo de geração de chave, mas menos 10 registos de 128 bits, porque não é necessário guardar as chaves de ronda. O bloco ronda não sofre qualquer alteração em relação à arquitetura anterior. Demorando igualmente 1 ciclo de relógio por ronda e 10 ciclos de relógio de latência do bloco de cifra.

3.9 Ambiente de teste

O correto funcionamento de um circuito digital é uma grande consideração no projeto de sistemas digitais. Dados os custos extremamente altos do fabrico de microchips, as consequências de falhas no projeto que passem despercebidas durante o desenvolvimento, até a fase de produção, são muito custosas.

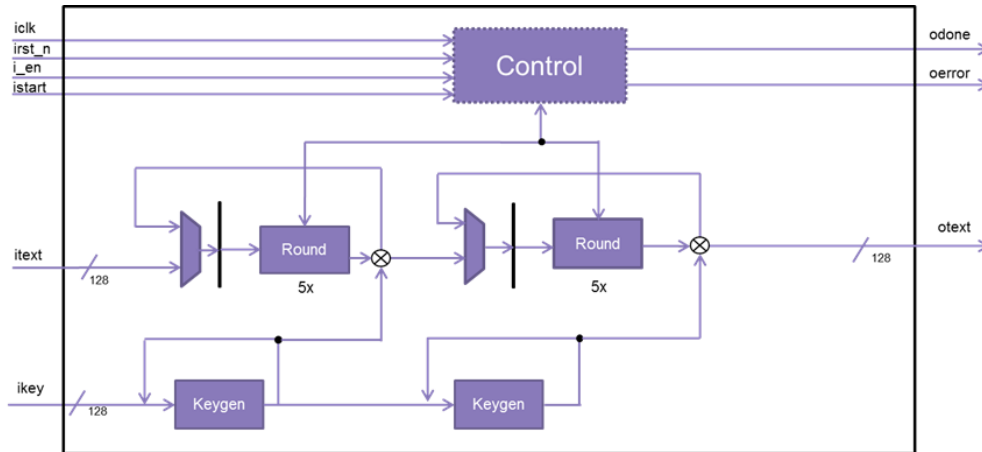


Figura 3.6: topo da Arquitetura AES16box2

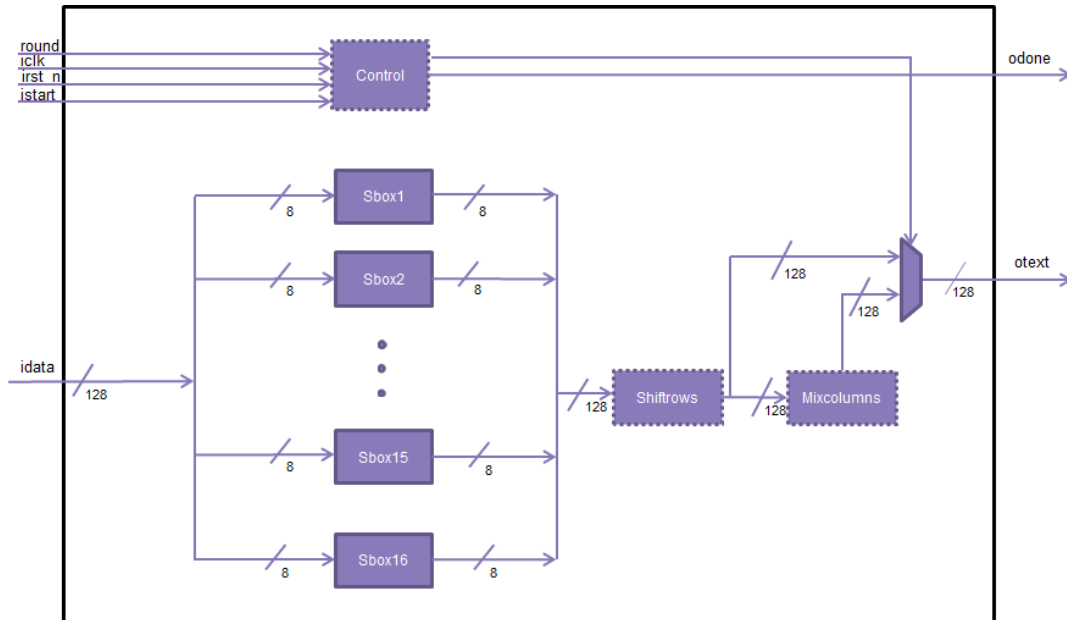


Figura 3.7: Ronda da Arquitetura AES16box e AES16box2

O ambiente de teste foi desenvolvida em Verilog a partir do plano de verificação, reproduzindo esses mesmos testes na simulação.

Para a verificação funcional foram aplicadas as entrada do nosso projeto a ser testado(DUT), os vetores de teste presentes no documento AESAVS. Este documento especifica os processos envolvidos na validação de implementações do algoritmo AES. Fornecendo um conjunto de vetores de teste (chave e dados) e correspondentes resultados a ser aplicados a implementações AES para estas obterem o certificado de conformidade [16]. Estes vetores de teste (chave e dados), assim como o resultado da sua cifragem foram guardados em arquivos sendo estes carregados usando o comando,

```
$readmemh("memout.data",tb_mem_otext);
$readmemh("memkey.data",tb_mem_key);
$readmemh("meminput.data",tb_mem_itext);
```

Após a instanciação do projeto e da inicialização das variáveis para as condições iniciais é inicializada a simulação.

Para a aplicação dos vetores de teste no nosso DUT, foi criada uma *task*. Em primeiro é apresentada a chamada da mesma durante a simulação,

```
#100

repeat(135) begin
    sample_data(input_zero,tb_mem_itext[tb_i],1'b1);
    tb_i = tb_i + 1;
end
$display("*****\n");
```

No final da execução da *task*, é incrementado o *array* que contém os blocos de dados e é executada a *task* novamente, durante 135 vezes, correspondendo a um dos testes presentes no AESAVS. A *task*, *sample_data*, apresentada a seguir recebe como entrada a chave de sessão, o bloco de dados a ser cifrado e o sinal *istart*, os dados de entrada são então aplicados no nosso DUT sincronizados como flanco ascendente do sinal de relógio, como o DUT recebe dados de entrada a cada 5 ciclos de relógio, durante quatro ciclos de relógio, o sinal de *istart* fica a 0.

```
task sample_data (input [127:0] ikey, input [127:0] itext, input istart);
begin
    @(posedge tb_iclk) begin
        $display("load_ikey_itext_start -> Load ikey=%h,
        itext=%h, istart=%d @ %0d",ikey,itext,istart,$time);
        tb_ikey    <= ikey;
        tb_itext    <= itext;
        tb_istart   <= istart;
    end
    repeat (4)@(posedge tb_iclk)
        tb_istart <= 1'b0;
end
endtask
```

O ambiente de teste implementado auto verifica os dados recebidos do modulo AES, este verifica a cada ciclo de relógio se o sinal *odone* é ativado, dentro de um bloco *always* síncrono com o sinal de relógio.

```

if(tb_odone) begin //check output values
    check_values (tb_otext,tb_mem_otext[tb_o],tb_error_tmp);
    tb_odone_flag  <= 1'b1;
    tb_error_total <= tb_error_total + tb_error_tmp;
    tb_o <= tb_o +1;
end

```

Se o sinal *tb_odone* estiver ativo é chamada a task *check_values*,

```

task check_values (input [127:0] iread_text, input [127:0] iexpected_text,
output error);
begin
    if (iread_text != iexpected_text) begin
        $display("ERROR!! - check_otext -> Read otext=%h != Expected
otext=%h @ %0d",iread_text,iexpected_text,$time);
        error = 1;
    end
    else begin
        $display("SUCESS!! - check_otext -> Read otext=%h == Expected
otext=%h @ %0d",iread_text,iexpected_text,$time);
        error = 0;
    end
end
endtask

```

Esta recebe como entrada o valor cifrado obtido do DUT e compara-o com o valor esperado presente no array *tb_mem_otext[tb_o]*. Caso os valores não forem iguais, é incrementado um contador de erro e apresentada uma mensagem de erro. Caso os valores sejam iguais, é apresentada uma mensagem de que a cifragem foi bem sucedida.

Para testar o sinal de *reset*, quando é aplicado o sinal de *reset* é chamada a task *check_reset* e após um ciclo de relógio são verificadas se as saídas do DUT, foram zeradas, sendo aplicado o mesmo teste quando o DUT esta desabilitado.

```

task check_values (input [127:0] iread_text,
input [127:0] iexpected_text, output error);
begin
    if (iread_text != iexpected_text) begin
        $display("ERROR!! - check_otext -> Read otext=%h !=
Expected otext=%h @ %0d",iread_text,iexpected_text,$time);
        error = 1;
    end
    else begin

```

```

    $display("SUCESS!! - check_otext -> Read otext=%h ==
    Expected otext=%h @ %0d", iread_text, iexpected_text, $time);
    error = 0;
end
end
endtask

```

Para o teste do sinal de erro, são aplicados dados ao nosso DUT fora dos 5 ciclos de relógio. A variável *tb_count_istart*, conta o numero de ciclos de relógio entre ativações sucessivas do sinal *istart*. Caso o sinal seja ativado fora desse intervalo, é verificado o sinal de erro com a *task check_error*. A figura 3.9 apresenta a simulação de auto verificação. O ambiente de teste completo é apresentado no anexo B.

```

if(tb_istart) begin
    if(tb_count_istart != 3'd4 && tb_iflag) begin
        $display(" ERROR-data input not sampled at
        5 clock cycles %0d", $time);
        tb_error_istart = tb_error_istart + 1;
        #(1)
        check_error;
    end
    tb_count_istart <= 0;
    tb_iflag <= 1;
end

if(!tb_istart && tb_iflag)
    tb_count_istart <= tb_count_istart + 1;
end

```

Na figura 3.8, pode-se ver parte da simulação realizada pelo ambiente de teste. Nesta podemos ver a entrada e saída de dados com uma cadência de 5 ciclos de relógio, assim como a latência do circuito, neste caso de 10 ciclos de relógio, porque esta simulação é da arquitetura AES16box2. Também se pode ver que após o sinal de *reset* ir a zero, as saídas do projeto são zeradas.

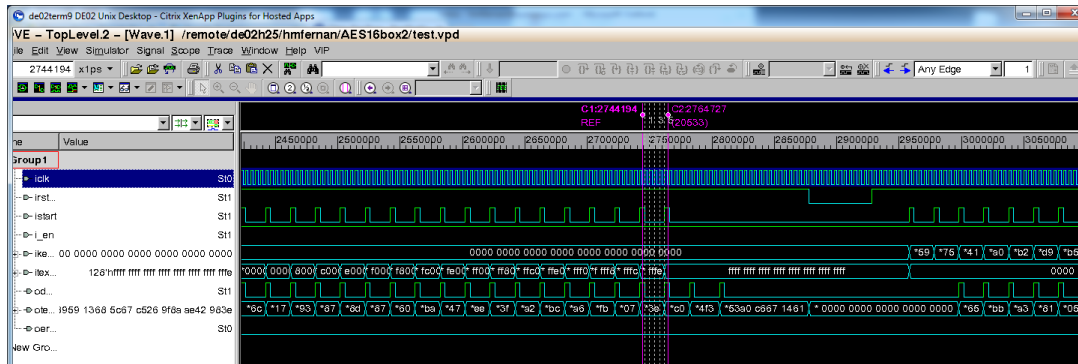


Figura 3.8: Formas de onda da verificação funcional

```

hmferrnan@de02dwm141:/remote/de02h25/hmferrnan/AES4box
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2602
SUCCESS!! - check_otext -> Read_otext=123c1f4af313ad8c2ce648b2e71fb6e1 == Expected_otext=123c1f4af313ad8c2ce648b2e71fb6e1 @ 2606
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2622
SUCCESS!! - check_otext -> Read_otext=1ffc626d30203dcdb0019fb80f726cf4 == Expected_otext=1ffc626d30203dcdb0019fb80f726cf4 @ 2626
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2642
SUCCESS!! - check_otext -> Read_otext=76da1fbc3a50728c50fd2e621b5ad885 == Expected_otext=76da1fbc3a50728c50fd2e621b5ad885 @ 2646
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2662
SUCCESS!! - check_otext -> Read_otext=082eb8bc35f442fb52668e16a591d1d6 == Expected_otext=082eb8bc35f442fb52668e16a591d1d6 @ 2666
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2682
SUCCESS!! - check_otext -> Read_otext=e656f9ecf5fe27ec3e4a73d00c282fb3 == Expected_otext=e656f9ecf5fe27ec3e4a73d00c282fb3 @ 2686
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2702
SUCCESS!! - check_otext -> Read_otext=2ca8209d63274cd9a29bb74bcd77683a == Expected_otext=2ca8209d63274cd9a29bb74bcd77683a @ 2706
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2722
SUCCESS!! - check_otext -> Read_otext=79bf5dce14bb7dd73a8e3611de7ce026 == Expected_otext=79bf5dce14bb7dd73a8e3611de7ce026 @ 2726
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2742
SUCCESS!! - check_otext -> Read_otext=3c849939a5d29399f344c4a0eca8a576 == Expected_otext=3c849939a5d29399f344c4a0eca8a576 @ 2746
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2762
SUCCESS!! - check_otext -> Read_otext=ed3e0a94e59bec98835da7a4f07ca2 == Expected_otext=ed3e0a94e59bec98835da7a4f07ca2 @ 2766
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2782
SUCCESS!! - check_otext -> Read_otext=63919ed4ce10196438b6ad0949ca795 == Expected_otext=63919ed4ce10196438b6ad0949ca795 @ 2786
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2802
SUCCESS!! - check_otext -> Read_otext=7678f3a833f19fea95f3c6029c2bc610 == Expected_otext=7678f3a833f19fea95f3c6029c2bc610 @ 2806
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2822
SUCCESS!! - check_otext -> Read_otext=3aa426831067d36b92be7c5f81c13c56 == Expected_otext=3aa426831067d36b92be7c5f81c13c56 @ 2826
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2842
SUCCESS!! - check_otext -> Read_otext=9272e2d2cdd11050998c845077a30ea0 == Expected_otext=9272e2d2cdd11050998c845077a30ea0 @ 2846
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2862
SUCCESS!! - check_otext -> Read_otext=088c4b53f5ec0ff814c19adae7f6246c == Expected_otext=088c4b53f5ec0ff814c19adae7f6246c @ 2866
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2882
SUCCESS!! - check_otext -> Read_otext=4010a5e401fd0a0354ddbcc0d012b17 == Expected_otext=4010a5e401fd0a0354ddbcc0d012b17 @ 2886
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2902
SUCCESS!! - check_otext -> Read_otext=a87a385736c0a6189bd6589bd8445a93 == Expected_otext=a87a385736c0a6189bd6589bd8445a93 @ 2906
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2922
SUCCESS!! - check_otext -> Read_otext=545f2b83d9616dccc6f0fa9830e3ca287 == Expected_otext=545f2b83d9616dccc6f0fa9830e3ca287 @ 2926
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2942
SUCCESS!! - check_otext -> Read_otext=4b706f79240635294037644f4680d == Expected_otext=4b706f79240635294037644f4680d @ 2946
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2962
SUCCESS!! - check_otext -> Read_otext=b7972b3941c4b90afa7b264dffa7387 == Expected_otext=b7972b3941c4b90afa7b264dffa7387 @ 2966
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 2982
SUCCESS!! - check_otext -> Read_otext=6f45732cf10881546f0fd23896d2bb60 == Expected_otext=6f45732cf10881546f0fd23896d2bb60 @ 2986
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 3002
SUCCESS!! - check_otext -> Read_otext=2e3579ca15af27f64b3c955a5bfc30ba == Expected_otext=2e3579ca15af27f64b3c955a5bfc30ba @ 3006
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 3022
SUCCESS!! - check_otext -> Read_otext=34a2c5a91ae2aec99b7d1b5fa6780447 == Expected_otext=34a2c5a91ae2aec99b7d1b5fa6780447 @ 3026
load_ikey_itey_start -> Load_ikey=00000000000000000000000000000000, itext=ffffffffffffffffffffffff000000, istart=1 @ 3042
SUCCESS!! - check_otext -> Read_otext=a4d6616bd04f87335b0e53351227a9ee == Expected_otext=a4d6616bd04f87335b0e53351227a9ee @ 3046
SUCCESS!! - check_otext -> Read_otext=7f692b03945867d16179a8cfc83ea3f == Expected_otext=7f692b03945867d16179a8cfc83ea3f @ 3066
SUCCESS - test ended correctly
SUCCESS - input data sampled correctly
SUCCESS - output data sampled correctly
$finish called from file "sim/tb_AES_top.u", line 179.
$finish at simulation time 4058000
V C S Simulation Report
Time: 4058000 ps
CPU Time: 0.739 seconds; Data structure size: 0.1Mb

```

Figura 3.9: Simulação com auto verificação

Capítulo 4

Fluxo de Projeto

Durante o desenvolvimento de um circuito digital, este passa por várias etapas até termos o produto final, o seu desenvolvimento pode ser dividido em três partes.

O projeto *Front-end*, que corresponde as tarefas necessárias para ir da especificação até a *netlist gate-level*. O projeto de *Back-end*, que corresponde as tarefas necessárias para ir da *netlist gate-level* até ao *layout* físico. A verificação onde é verificado se o nosso projeto está de acordo com a especificação e se se comporta da maneira esperada. Neste trabalho foi realizado o fluxo de projeto do *Front-end*, assim como a verificação do projeto. A seguir são apresentados as várias etapas deste fluxo, assim como as ferramentas usadas em cada etapa.

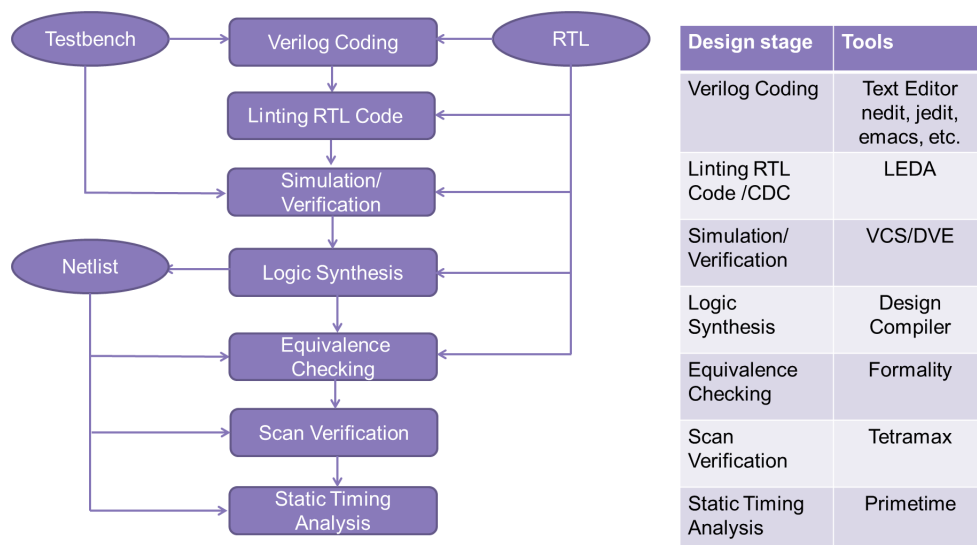


Figura 4.1: Fluxo de projeto *Front-End* [7]

4.1 Fluxo de Projeto Front-End

O fluxo de projeto *Front-End* é composto pela especificação, Código RTL, seguido das ferramentas de *linting*, simulação, síntese, verificação formal, análise de tempo estática e geração de padrões de varrimento.

4.2 Especificação

As especificações do projeto estão normalmente presentes num documento, descrevendo um conjunto de funcionalidades que a solução final tem de fornecer e um conjunto de restrições que têm de ser satisfeitas. Durante esta fase é feito o levantamento dos requisitos.

4.3 Arquitetura Alto Nível

Após os levantamentos dos requisitos da especificação foi realizado uma arquitetura de alto nível, em que foi desenvolvido um modelo funcional da especificação não sintetizável. Através de uma aproximação hierárquica, o projeto foi subdividido em blocos, neste caso, nas várias transformações que fazem parte do algoritmo AES. Após os blocos apresentarem o comportamento funcional esperado, são integrados no topo e é testada a funcionalidade do projeto como um todo.

4.4 Desenvolvimento do código

Durante esta fase é desenvolvido o nosso modelo RTL, assim como o nosso ambiente de teste. A partir do modelo de projeto funcional, é procedida a fase de projeto RTL. Durante esta fase, a descrição da arquitetura é refinada, entrando em conta com os componentes funcionais que queremos implementar, elementos de memória, desenvolvimento do sistema de relógio, objetivos de desempenho, área e energia. Também durante esta fase é desenvolvido o nosso ambiente de teste, através do qual testamos o nosso modelo RTL, aplicando estímulos nas entradas e verificando as saídas, de modo a verificar se o nosso modelo RTL cumpre os requisitos da especificação. Esta fase de desenvolvimento é iterativa, sendo que após a simulação e posterior verificação do projeto, vai se corrigindo as possíveis falhas tanto no modelo RTL como no ambiente de teste. Para verificarmos a qualidade do mesmo, durante a simulação faz-se a verificação de cobertura, que é uma métrica da qualidade do nosso ambiente de teste. Esta diz-nos, por exemplo, se o ambiente de teste criado estimula todos os sinais, e se todo o código RTL é executado durante a simulação.

4.5 Linting RTL

Nesta fase é verificada a qualidade do nosso código RTL. Para o fazer é utilizada a ferramenta LEDA. Esta verifica se o código cumpre um conjunto de regras, tais como, normas de programação como o Verilog 2001, IEEE std 1800-2005 que corresponde a norma System Verilog e outras

regras de projeto, podendo estas serem definidas pelo utilizador. A ferramenta é carregada com o nosso código RTL e verifica se este cumpre as regras, avisando qual a regra que não foi cumprida e onde. Nesta fase a ferramenta analisa o projeto do início até ao fim por erros que podem causar problemas durante a execução do fluxo. Por exemplo, esta ferramenta pode verificar se o código é sintetizável, se foram inferidas latches no projeto, se todas as linhas de código são executáveis, verificar se não existem atribuições simultâneas à mesma variável, entre outras [17].

4.6 Simulação/Verificação

Nesta fase é quando o nosso modelo RTL é verificado, aplicando os testes e estímulos criados no ambiente de teste, para esse efeito é usada a ferramenta VCS. Esta tem como entrada os ficheiros RTL e o ambiente de teste criado para testar o modelo. Este faz a simulação do circuito, após a qual é criado um ficheiro com a informação de transição de cada sinal do projeto. Para a visualização desse ficheiro é usada a ferramenta DVE, que nos permite fazer a visualização das formas de onda geradas pelo VCS, permitindo dessa forma analisar o comportamento do circuito e fazer a sua depuração. A fase de verificação e simulação consiste em obtermos um nível de confiança de que o circuito irá funcionar corretamente. A motivação adjacente à verificação é de remover todos os possíveis erros do projeto antes de prosseguirmos para a fabricação, onde a deteção de uma falha nessa fase é muito cara. Sempre que são encontrados erros funcionais, o modelo RTL precisa de ser modificado de modo a corrigi-los e refletir o comportamento correto. A qualidade dos testes de verificação é normalmente avaliada em termos de cobertura, que corresponde a uma medida da percentagem do projeto que foi verificado. A verificação funcional pode fornecer apenas uma cobertura parcial devido a sua aproximação. O objectivo é então maximizar a cobertura do projeto a ser testado. Várias medidas são usadas, como *line coverage*, que conta o número de linhas da descrição RTL foram ativadas durante a simulação. *Condition coverage* dá-nos a percentagem das configurações possíveis do projeto que foram estimuladas, do mesmo modo o *toggle coverage* dá-nos a percentagem dos sinais que foram ativados [18].

Na figura 4.2 podemos ver o resultados detalhados da cobertura para a Arquitetura AES16box2, em que se pode ver um score de quase 100% , tirando a instância *keygen*, mas isso é explicado pelo facto de a chave ter valor fixo durante uma sessão inteira no modo CTR e dessa forma não foram utilizadas todos os valores possíveis das S-box usadas para o cálculo das chaves.

4.7 Síntese Lógica

Depois de se verificar que o nosso projeto cumpre os requisitos da especificação e se comporta corretamente, pode se considerar o código RTL fechado, estando o nosso projeto pronto para seguir para a próxima fase, a síntese. Neste passo o projeto é sintetizado para uma dada tecnologia. As entradas para se proceder à síntese são o código RTL, a biblioteca da tecnologia e as restrições de projeto. O resultado geral desta fase é a geração de um modelo detalhado do circuito, o qual

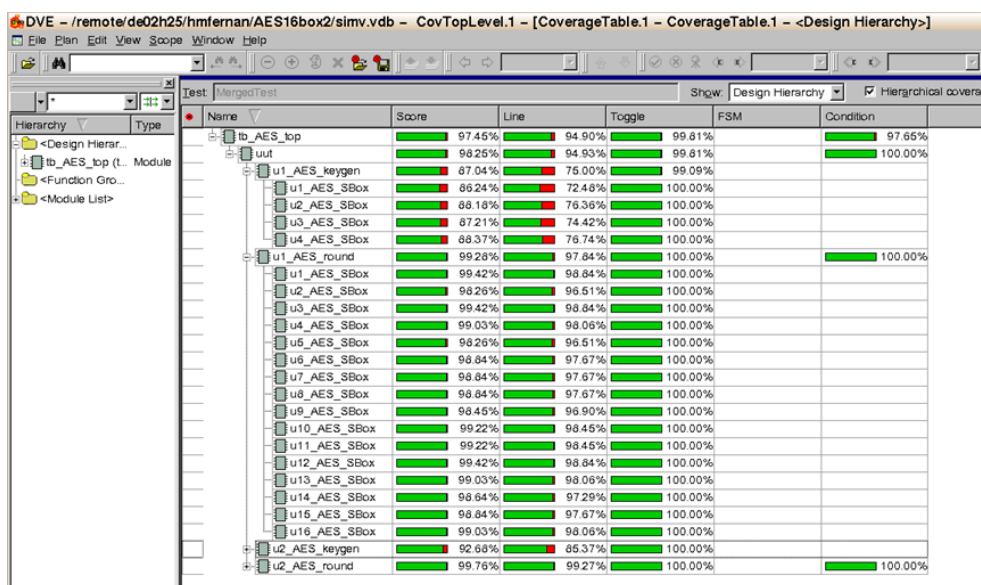


Figura 4.2: Resultados de coverage obtido da Arquitetura AES16box2

é otimizado baseado nas restrições definidas pelo projetista. Um projeto pode ser otimizado para um baixo consumo de energia, área, débito e testabilidade.

A síntese é um processo automático. Esta envolve múltiplas iterações de tentativa erro até que se consiga convergir para uma solução que satisfaça as restrições. Durante esta fase do projeto é também feita a inserção da cadeia de varrimento usando a ferramenta DFT. A inserção da cadeia de varrimento permite que o nosso projeto seja testado posteriormente durante o processo de manufatura, para verificar se houve falhas durante esse processo [19].

O processo de síntese de um circuito é um processo iterativo e começa pela definição das restrições de projeto e a definição da biblioteca da tecnologia usada.

A biblioteca da tecnologia contém a informação que a ferramenta de síntese necessita para gerar a *netlist gate-level* para um projeto, baseado no comportamento lógico e nas suas restrições.

As bibliotecas contêm a função lógica das células, área, atrasos, restrições de *fanout* e tempos de *hold* e *setup*.

As restrições de projeto definem as metas que se querem atingir. Estas podem consistir em restrições de área e temporais, geralmente derivadas das especificações de projeto.

A ferramenta de síntese usa estas restrições e tenta otimizar o projeto de modo a conseguir cumpri-las. Para se proceder à síntese, foram primeiramente definidas as restrições do projeto. Estas são apresentadas a seguir,

```
set period 1.66
set half_period [expr 0.5*$period]

set wave_rise 0.0
set wave_fall [expr 0.5*$period]
```

```

set waveform [list $wave_rise $wave_fall]

set max_period_input [expr 0.6*$half_period]
set min_period_input 0.0

set max_period_output [expr 0.4*$half_period]
set min_period_output 0.0

create_clock [get_ports {iclk}] -name "iclk" -period $period
-waveform [list $wave_rise $wave_fall]

set_input_delay $max_period_input -max -clock iclk [get_ports {istart}]
set_input_delay $min_period_input -min -clock iclk [get_ports {istart}]

set_input_delay $max_period_input -max -clock iclk [get_ports {i_en}]
set_input_delay $min_period_input -min -clock iclk [get_ports {i_en}]

set_input_delay $max_period_input -max -clock iclk [get_ports {ikey}]
set_input_delay $min_period_input -min -clock iclk [get_ports {ikey}]

set_input_delay $max_period_input -max -clock iclk [get_ports {itext}]
set_input_delay $min_period_input -min -clock iclk [get_ports {itext}]

set_output_delay $max_period_output -max -clock iclk [get_ports {odone}]
set_output_delay $min_period_output -min -clock iclk [get_ports {odone}]

set_output_delay $max_period_output -max -clock iclk [get_ports {otext}]
set_output_delay $min_period_output -min -clock iclk [get_ports {otext}]

set_output_delay $max_period_output -max -clock iclk [get_ports {oerror}]
set_output_delay $min_period_output -min -clock iclk [get_ports {oerror}]

set_load 1 [all_outputs]

```

O comando **create_clock** define o período e a forma de onda de um determinado relógio, a opção **-period** define o período de relógio e a opção **-waveform** controla o *duty cycle* do mesmo, tendo sido definido um período de 1.66ns, equivalente a uma frequência de 600MHz, com um *duty cycle* de 50%.

A opção **get_ports** especifica qual o porto associado à restrição que está a ser definida. O comando **set_input_delay** especifica o tempo de chegada de um sinal em relação ao relógio do sistema. Este especifica o tempo que um sinal demora a estar disponível na entrada do pino após o flanco de relógio. Este é usado nos pinos de entrada, para especificar o tempo que um sinal fica estável depois do flanco de relógio. Este representa o atraso do caminho combinacional de um registo externo até às entradas do nosso projeto. A opção **-max** e **-min** especifica se é o valor máximo e mínimo respetivamente, **-clock** especifica em relação a que sinal de relógio são aplicados os atrasos.

O comando **set_output_delay** especifica o tempo antes do flanco do sinal de relógio onde o sinal é necessário. Este representa o atraso do caminho combinacional para um registo fora do projeto, mais o

tempo de *setup* ou o tempo de *hold* caso se trate do atraso máximo ou mínimo.

Para definir a cadeia de varrimento, é preciso definir as suas restrições, tais como o período e forma de onda do relógio de varrimento, definir modos de teste, especificar portas de teste, além de identificar e marcar as células que não se quer verificar. As definições usadas na cadeia de varrimento são apresentadas em baixo,

```
create_clock [get_ports {iscanclk}] -name "iscanclk" -period $period
-waveform [list $wave_rise $wave_fall]

set_dft_signal -view spec -type ScanDataOut -port oscanout
set_dft_signal -view spec -type ScanDataIn -port iscanin
set_dft_signal -view spec -type ScanEnable -active state 1 -port iscanen
set_dft_signal -view spec -type TestMode -active_state 1 -port iscanmode

set_dft_signal -view existing_dft -type ScanClock -timing {45 55} -port iscanclk
set_dft_signal -view existing_dft -type Reset -port iscanrst_n -active_state 0
set_dft_signal -view existing_dft -type Constant -active_state 1 -port iscanmode

set_input_delay $max_period_input -max -clock iscanclk [get_ports {iscanmode}]
set_input_delay $min_period_input -min -clock iscanclk [get_ports {iscanmode}]

set_input_delay $max_period_input -max -clock iscanclk [get_ports {iscanin}]
set_input_delay $min_period_input -min -clock iscanclk [get_ports {iscanin}]

set_output_delay $max_period_output -max -clock iscanclk [get_ports {oscanout}]
set_output_delay $min_period_output -min -clock iscanclk [get_ports {oscanout}]

set_input_delay $max_period_input -max -clock iscanclk [get_ports {iscanen}]
set_input_delay $min_period_input -min -clock iscanclk [get_ports {iscanen}]

set_scan_configuration -chain_count 1
set_scan_path chain0 -view spec -complete false -scan_master_clock iscanclk \
-scan_data_in iscanin -scan_data_out oscanout
```

O relógio de varrimento foi definido com um *duty cycle* de 50% e um período de 40ns, frequência de 25MHz. O comando usado para a especificação dos portos de scan,

```
set_dft_signal
```

as opções que foram usadas com este comando são introduzidas em seguida:

- -view
 - spec, especifica que o sinal ainda não foi definido.
 - existing_dft, especifica que o sinal já foi definido como sinal de varrimento.

- -port, diz ao compilador qual o porto que está a ser definido.
- -type, define a sua função do sinal.
- -timing, define o tempo de subida e descida do relógio de varrimento.
- -active_state define o estado ativo do sinal.

O comando

```
set_scan_configuration
```

permite nos especificar a cadeia de varrimento do nosso projeto, a opção - **chain_count**, especifica o número de cadeias de varrimento que queremos inserir. Neste trabalho foi inserida apenas uma.

```
set_scan_path
```

serve para definir uma cadeia de varrimento, -**view spec**. Indica que a cadeia não existe e terá de ser inserida, -**complete false**, indica ao compilador que a cadeia de varrimento não está completa e que este pode adicionar mais células de varrimento durante a sua inserção para a balancear. -**scan_master_clock** define o sinal de relógio a ser usado durante a configuração de teste, -**scan_data_in** e -**scan_data_out** especificam o porto de entrada e saída da cadeia de varrimento.

Após estarem definidas as restrições do projeto, sinais e cadeia de varrimento, pode se dar início à síntese.

No primeiro passo é carregada a biblioteca da tecnologia, e o código RTL e as restrições do projeto. Em seguida é verificado o projeto por erros de consistência,

```
check_design -summary
```

a opção -**summary**, apresenta um sumário com todos os *warnings* encontrados. Caso seja detectado algum erro a síntese é interrompida. Após esta verificação e não tendo encontrado erros, o processo de síntese é iniciado,

```
compile_ultra -scan -gate_clock -area_high_effort_script
```

O comando **compile_ultra** executa a síntese em alto esforço para se obter melhores resultados.

A opção -**scan** é usada para permitir a inserção de cadeia de varrimento, substituindo os elementos sequenciais normais por elementos de varrimento.

A opção -**gate_clock** são inseridas automaticamente *clock_gates*. Estas permitem reduzir o consumo dinâmico do circuito, ao desativar partes do mesmo que não estejam a ser usadas. -**area_high_effort_script** corre um *script* preparado para melhorar a área do projeto. Este script aplica uma estratégia de compilação que pode ativar ou desativar diferentes opções de otimização conforme o objetivo.

Em seguida são carregadas as restrições da cadeia de varrimento e verificadas usando o comando

```
dft_drc
```

Este verifica a implementação da cadeia de varrimento especificada com o comando **set_scan_configuration** por violações antes da sua inserção, caso não tiver erros é feita a inserção da cadeia de varrimento.

```
insert_dft
```

Depois da inserção da cadeia de varrimento o projeto é novamente sintetizado, usando o modo incremental. Este modo é apenas usado após a primeira compilação. Neste modo a ferramenta não faz o mapeamento, apenas trabalha a nível da *gate*, para melhorar o comportamento temporal

```
compile_ultra -scan -incremental
```

Em seguida é realizada uma otimização de área gate a gate, sem degradar os tempos, nem os consumos.

```
optimize_netlist -area
```

Após a otimização de área, o processo de síntese está terminado, sendo criados uma série de relatórios, assim como a descrição do projeto ao nível da gate para a tecnologia escolhida. O relatório mais importante é o temporal, que mostra a folga dos vários caminhos críticos, e se foi possível cumprir as restrições temporais definidas, não havendo violações. Caso não tenha conseguido atingir o desempenho temporal desejado, teremos que alterar o código RTL ou modificar as restrições temporais e correr este passo de novo.

4.8 Verificação de Equivalência

A verificação de equivalência é um método para verificar se a síntese foi feita corretamente sem recorrer a qualquer simulação, através da comparação do código RTL com a descrição do projeto ao nível da gate obtido na síntese. Esta verificação é feita através da comparação comportamental ponto a ponto entre ambos os designs. Para a realização da verificação de equivalência foi usada ferramenta Formality. Durante a síntese, o Design Compiler, guarda informação de setup para a verificação de equivalência. Cada vez que o Design Compiler faz uma alteração, essa informação é em guardada sob a forma de *automated setup file* (.svf). Este ficheiro ajuda o Formality a perceber e a processar as mudanças de projeto que foram efetuadas durante a síntese. O Formality usa esta informação para ajustar o seu processo de verificação [20].

4.9 Verificação de Scan

Como foi visto anteriormente, a cadeia de varrimento serve para verificarmos se o nosso circuito não apresenta defeitos de fabrico. Como tal, a cadeia de varrimento, caso seja bem inserida, tem de ser capaz de cobrir todo o projeto permitindo controlar e observar todos os nós do circuito. Um nó é controlável se for possível fazer com que ele tome um valor específico aplicando dados nas entradas do módulo e é observável se for possível prever a sua resposta e propagá-la para as saídas, observando dessa forma a resposta. Nesta fase são gerados os padrões de teste que permitem a sensibilização e propagação das falhas estruturais do circuito. Os defeitos de fabrico podem não ser detetados pelo teste funcional, mas causar um comportamento incorreto, ou encurtar o tempo de vida do circuito integrado. A geração destes padrões de teste é feita usando a ferramenta TetraMax, sendo a geração feita automaticamente, esta tenta maximizar a cobertura do teste, usando o menor número de vetores de teste possível. Quando existe um defeito de fabrico, este provoca uma alteração no comportamento lógico do circuito. Esta ferramenta deteta principalmente dois tipo de falhas. As falhas *stuck-at*, em que um nó permanece preso a um determinado valor, tal como um curto-circuito à alimentação em que o nó irá permanecer sempre a 1, independentemente do estímulo que lhe é aplicado. Este também deteta as falhas *transition delay fault*, onde são detetadas transições lentas de 0-1 ou 1-0. Um defeito deste tipo significa que para a máxima frequência de operação

do dispositivo, não será produzido o resultado correto. Para o detetar, é lançada uma transição em um flanco do relógio e é capturado o efeito da transição no outro flanco. O intervalo de tempo entre os dois flancos testa o dispositivo para o seu correto comportamento [21]. Os resultados e cobertura dos padrões de varrimento gerado pelo TetraMax são apresentados no anexo C.

4.10 Análise Temporal Estática

A análise temporal estática, é uma parte essencial do projeto. Esta valida exhaustivamente o desempenho temporal ao verificar todos os caminhos em busca de violações temporais, sem usar simulação lógica ou vectores de teste. Nesta etapa foi usada a ferramenta PrimeTime. Para efetuar a verificação, esta divide o projeto em conjuntos de caminhos temporais. O cálculo dos atrasos de propagação dos sinais é feito através da soma dos atrasos nas ligações e células. Esta ferramenta permite-nos detectar violações temporais, tais como violações de tempo de *setup* e *hold*, assim como o *skew* e os caminhos lentos que nos limitam a frequência do nosso sistema. A análise temporal feita pelo PrimeTime é diferente da realizada pelo Design Compiler. Uma das diferenças está nos atrasos dependentes da carga. O Design Compiler, na geração de um relatório temporal originado a partir do início de um caminho segmentado, transfere a porção de carga dependente do atraso da *gate* de origem para o atraso da *net* de saída. Este método pode fornecer melhores resultados de síntese, mas não é o mais apropriado para análise temporal estática. O PrimeTime não adiciona o atraso dependente da carga para a *net*. A medição dos tempos de transição também é diferente, o PrimeTime propaga os tempos de transição [22].

Capítulo 5

Resultados

Este capítulo descreve os resultados obtidos após o desenvolvimento e síntese das Arquiteturas desenvolvidas

5.1 Resultados obtidos em FPGA

Após a síntese do projeto para FPGA Xilinx Virtex 5 é possível verificar na tabela 5.1 a informação sobre a utilização do sistema, onde se pode ver o número de registos usados, o número de LUTs utilizadas, entre outras características assim como a frequência máxima do sistema.

Arquiteturas	AES4box	AES8box	AES16box	AES16box2
FlipFlops	4460(2%)	2889(1%)	1818(0%)	671(0%)
RAM/ROM 256x1	320	336	288	320
Total LUTs	5046(2%)	3640(1%)	2846(1%)	2607(1%)
Frequência Máxima	301.5MHz	345.8MHz	299.7MHz	301.6MHz

Tabela 5.1: Resultados Obtidos das Várias Arquiteturas

Apesar de o caminho crítico na arquitetura AES16box e AES16box2 ser o mesmo, que corresponde a uma ronda completa, estes são sintetizados de maneira diferente, daí a pequena diferença de frequência máxima entre ambos, sendo o atraso total do caminho critico em AES16box2 dado por,

$$\text{Total Path delay} = 1.531 \text{ (logic)} + 1.793 \text{ (route)} = 3.324\text{ns}$$

e o atraso do caminho critico em AES16box dado por,

$$\text{Total Path delay} = 1.493 \text{ (logic)} + 1.844 \text{ (route)} = 3.337\text{ns}$$

Isto é devido ao facto de a implementação AES16box usar mais LUTs que a implementação AES16box2, tendo por isso um maior atraso nas *nets*. Os caminhos criticos das arquiteturas são apresentadas no anexo D

Em comparação com a arquitetura implementada em [4], em que as S-box foram implementadas usando GF(2), com vários estágios de pipelining, estes conseguiram um débito de 20Gbps em FPGA Xilinx Virtex 5, sem recorrer a *sub-pipelining*, enquanto que as implementações realizadas neste trabalho, tinham como

target, um débito de 7.2Gbps, metade do objetivo a alcançar em ASIC. Em relação ao consumo de área, eles utilizaram 8800 slices, contendo cada slice 4 LUTs, perfazendo um total de 35200 LUTs, enquanto que a arquitetura AES16box2 apenas utilizou 2607 LUTs, aproximadamente 13x menos. Na implementação [6], foi conseguida uma implementação em FPGA Xilinx Virtex 5 com um débito de 4.1 Gbps e uma frequência máxima de 350Mhz, ocupando uma área de 400 slices, ou seja 1600 LUTs. Obtendo menos 1000 LUTs em termos de área, mas com um débito de aproximadamente metade.

Arquiteturas	AES4box	AES8box	AES16box	AES16box2
Frequência Máxima	231MHz	254MHz	223MHz	224MHz

Tabela 5.2: Resultados obtidos após Place-and-Route

Também foi realizado o *place-and-route*, onde foi simulado a integração, do projeto através da criação de um novo topo onde o nosso projeto foi instanciado, criando uma cadeia de registos entre o projeto e os pinos de entrada e saída, eliminando depois os atrasos entre os pinos e a cadeia de registos, visto que o propósito deste projeto é a integração noutra mais complexo sem ligações físicas com o exterior. Obtendo desta forma um resultado mais fidedigno. Os Resultados do podem ser vistos na tabela 5.2, estando estes proporcionais aos valores de frequência obtidos na síntese.

5.2 Resultados obtidos em ASIC

Na tabela 5.3 são apresentadas as diferenças entre as várias arquiteturas desenvolvidas e os resultados obtidos da síntese para ASIC das várias arquiteturas é apresentado na tabela 5.4. Estes resultados foram obtidos para uma frequência de operação de 600MHz e utilizando a ferramenta Design Compiler versão 1-2013-SP2.

Arquiteturas	AES4box	AES8box	AES16box	AES16box2
Pré-cálculo das chaves	Não	Sim	Sim	Não
Nº rondas replicadas	10	5	2	2
Duração de cada ronda (Ciclos de relógio)	5	2	1	1
Nº S-box por ronda	4	8	16	16
Latência inicial (ciclos de relógio)	50	30	10	10
# S-box	40	42	36	40

Tabela 5.3: Diferenças das várias Arquiteturas

Da tabela 5.4 podemos concluir que a melhor implementação das quatro realizadas foi a AES16box2, conseguindo uma área equivalente mais baixa e também um consumo mais baixo, embora este último valor não seja muito realista, pois trata-se de uma análise estática e não toma em conta as transições. Como podemos ver, pela comparação entre as arquiteturas AES16box e AES16box2, apesar de a primeira ter menos 4 S-box instanciadas, a sua implementação é 20% pior em termos de área, visto que ao fazer a pré-computação das chaves são necessários 10 registos de 128 bits, um para cada chave de ronda, o que faz triplicar a Área não combinacional da arquitetura AES16box. Também se pode observar, que a instanciação

de menos 4 S-box, não fez grande diferença em relação a área combinacional consumida. A arquitetura AES4box mostrou-se desadequada para esta especificação, porque apesar de ter o mesmo número de S-box instanciadas que a implementação AES16box2, o facto de ter todas as rondas instanciadas cria um grande aumento de área consumida devido a replicação das rondas e também devido ao número de registos usados, visto que torna necessário fazer *pipelining* no final de cada ronda, assim como dentro de cada ronda para a reutilização das S-box. Este tipo de implementação é mais aconselhada para se obterem valores maiores de débito.

As arquiteturas AES4box e AES8box, apesar de terem resultados bastante inferiores em termos de área em relação a AES16box e AES16box2, serão capazes de conseguir um débito bastante superior, mas a custa de uma latência inicial muito superior, podendo ser úteis em futuras especificações com objetivos de desempenho mais elevados.

Arquiteturas	AES4box	AES8box	AES16box	AES16box2
Cell Area				
Combinacional	35214.9	24828.8	26246.5	26368.2
Não combinacional	23315.1	15464.4	9980.5	3785
Buf/Inv	3511.4	2955.9	3511	3312.4
Total	58530	40293	35385	30153.9
Gates Equivalentes(K)	82.9	57.1	50.1	42.7
Diferença de Area(%)	+194	+133	+117	-
Static Power Estimations				
Total (mW)	5.96	4.08	5.08	2.60

Tabela 5.4: Resultados Obtidos das Várias Arquiteturas

Na tabela 5.5 são comparadas as implementações referidas na literatura com a nossa, embora uma comparação completamente justa não seja possível de fazer, visto que as implementações usam diferentes tecnologias, assim como diferentes objetivos.

Arquiteturas	S-box	Tecnologia (nm)	Frequência (MHz)	Débito (Gbps)	Área (Kgates)	Kbits/gate
AES16box2	LUT combinational	40	600	15.36	42.7	359
[2]	LUT	180		33	175	188
	GF			34	145	234
[5]	GF	90	769	19.67	46.3	424
[14]	GF	130	429	54.94	272.3	201
[4]	GF	110	224	2.6	21.3	122

Tabela 5.5: Comparação com a literatura

Normalmente na literatura é usado o valor *kbits/gate* como termo de comparação entre diferentes arquiteturas, para medir a eficiência da cada uma delas, na tabela 5.5, podemos ver que a nossa implementação é a segunda mais eficiente, com um débito de 359Kbits/gate.

Em comparação com [5], na qual foi atingida uma frequência de 769MHz, com um débito de 19.6Gbps e uma área equivalente de 46.3Kgates em tecnologia CMOS 90nm. Sendo esta a implementação mais

próxima da nossa, estes conseguiram um débito 28% superior, com uma área 8% superior em relação a arquitetura AES16box2. Para tal recorreram à implementação das S-box em $GF(2^2)$, com 6 estágios de *pipelining*, num total de 8 estágios de *pipelining* por ronda. Apesar de terem conseguido um débito superior, a arquitetura desenvolvida, apresenta uma desvantagem ao ser necessário ter um grande banco de registos à entrada para alimentar o algoritmo, pois este recebe 16 blocos de dados durante os primeiro 16 ciclos de relógio e depois tem de esperar 144 ciclos de relógio para que a cifragem desses 16 blocos esteja pronta e para dar início a cifragem de mais 16 blocos de dados. Para se conseguir um débito semelhante ou mesmo superior na arquitetura AES16box2, bastaria adicionar um estágio de pipeline à saída das S-box e antes das transformações ShiftRows Mixcolumns e Addroundkey, com um aumento de apenas 2 registos de 128 bits.

Capítulo 6

Conclusão

6.1 Conclusões

Nesta dissertação foram desenvolvidas, implementadas e verificadas quatro soluções do algoritmo AES-CTR para aplicações HDMI 2.0. Estas tendo sido sintetizadas para a tecnologia de 40nm e para FPGA Xilinx Virtex 5 LX330. Durante a implementação foi utilizado o fluxo de projeto *front-end* da Synopsys, onde foram executados todos os passos necessários no desenvolvimento de um projeto desde a especificação até a síntese.

A parte mais crítica da implementação AES são as S-box. É o processo que mais recursos consome e portanto, onde é mais concentrada a investigação numa tentativa de conseguir reduzir o seu peso. As implementações das S-box mais comuns são as implementações usando LUT. Como já foi visto anteriormente, consistem em tabelas com os valores de substituição pré-computados. Estas podem ser implementadas em lógica sequencial ou lógica combinacional. A implementação em lógica sequencial é na qual é conseguido um maior débito, mas em contrapartida esta consome muita área, visto que é necessário guardar 256 bytes por cada S-box. A implementação com lógica combinacional, como não recorre a elementos de memória, esta consome muito menos área, mas também, consequentemente tem um menor débito. Outro método de implementação das S-box, é fazendo o seu cálculo durante a execução, onde é feito o cálculo da inversa em GF. Esta implementação torna-se mais complexa, mas recorrendo a *sub-pipelining* é conseguido um débito próximo da implementação LUT com lógica sequencial. Esta implementação permite uma redução na ordem dos 35% em comparação com a implementação usando lógica sequencial.

Neste projeto foi usada a implementação das LUT pré-computadas usando lógica combinacional, visto que com ela conseguiu-se executar cada ronda AES num único ciclo de relógio, alcançando os objetivos de desempenho requeridos, permitindo dessa forma obter um projeto compacto, com uma área bastante otimizada e com menor latência inicial.

Foram cumpridos todos os objetivos tendo sido obtido um projeto completamente funcional, cumprindo as especificações funcionais e os requisitos de desempenho.

6.2 Trabalho Futuro

Nesta secção final é apresentada uma possível expansão do trabalho. Este poderia ser a realização das S-box calculando o seu valor durante a execução recorrendo a vários estágios de *pipelining* e a campos finitos de galois, comparando os resultados com os que foram obtidos neste trabalho. Outra possível expansão seria

a realização do fluxo de *back-end*, até o projeto estar pronto para ser produzido. Sendo o fluxo *back-end* composto pelos passos pós-síntese tais como:

- *Place and route*, onde é convertida a *netlist gate-level* produzida durante a síntese num projeto físico, feita a colocação das células e a síntese da árvore de relógio.
- Extração de parasitas, onde é criado o modelo RC do circuito para simulações futuras, análises temporais e de análise de consumo.
- Verificação estática temporal sobre o projeto físico.
- Verificação pós *layout*.

Desta forma seria possível obter resultados mais precisos de funcionamento e desempenho das arquiteturas realizadas.

Anexo A

Relatórios da Analise Estática temporal

Neste anexo são apresentados os resultados temporais obtidos com a ferramenta PrimeTime

```
Report : timing
-path_type full
-delay_type min_max
-input_pins
-nets
-slack_lesser_than 0.000
-max_paths 1
-sort_by slack
Design : AES_top
Version: H-2013.06
Date   : Wed May 28 01:17:37 2014
```

No paths with slack less than 0.000.

No paths with slack less than 0.000.

1

```
Report : global_timing
Design : AES_top
Version: H-2013.06
Date   : Wed May 28 01:17:37 2014
```

No setup violations found.

No hold violations found.

1

```

*****
Report : clock
Design : AES_topp
Version: H-2013.06
Date   : Wed May 28 01:17:37 2014
*****

```

Attributes:

```

p - Propagated clock
G - Generated  clock
I - Inactive   clock

```

Clock	Period	Waveform	Attrs	Sources
iclk	1.660	{0 0.83}		{iclk}
iscanclk	40.000	{0 20}		{iscanclk}

1

```
pt_shell> report_timing
```

```

*****
Report : timing
-path_type full
-delay_type max
-max_paths 1
-sort_by slack
Design : AES_topp
Version: H-2013.06
Date   : Wed May 28 01:18:49 2014
*****

```

```

Startpoint: u1_AES_Top/rstate_reg_0__9_
             (rising edge-triggered flip-flop clocked by iclk)
Endpoint:   u1_AES_Top/rstate_reg_1__69_
             (rising edge-triggered flip-flop clocked by iclk)
Path Group: iclk
Path Type:  max

```

Point	Incr	Path
-------	------	------

clock iclk (rise edge)	0.000	0.000
clock network delay (ideal)	0.000	0.000
u1_AES_Top/rstate_reg_0__9_/CP (SDFCNQD4BWP)	0.000	0.000 r
u1_AES_Top/rstate_reg_0__9_/Q (SDFCNQD4BWP)	0.192	0.192 f
u1_AES_Top/U2131/ZN (CKND6BWP)	0.031	0.223 r
u1_AES_Top/U1716/ZN (CKND12BWP)	0.034	0.257 f
u1_AES_Top/u1_AES_round/idata[9] (AES_round_0)	0.000	0.257 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/idata[1] (AES_SBox_18)	0.000	0.257 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/U100/ZN (INV6BWP)	0.039	0.296 r
u1_AES_Top/u1_AES_round/u15_AES_SBox/U290/ZN (NR2XD1BWP)	0.040	0.336 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/U36/ZN (CKND0BWP)	0.057	0.394 r
u1_AES_Top/u1_AES_round/u15_AES_SBox/U68/ZN (CKND2D0BWP)	0.071	0.464 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/U343/ZN (AOI22D0BWP)	0.091	0.555 r
u1_AES_Top/u1_AES_round/u15_AES_SBox/U92/ZN (CKND2D0BWP)	0.072	0.627 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/U90/ZN (NR2D0BWP)	0.069	0.696 r
u1_AES_Top/u1_AES_round/u15_AES_SBox/U300/ZN (CKND2D0BWP)	0.068	0.764 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/U276/ZN (AOI21D1BWP)	0.058	0.822 r
u1_AES_Top/u1_AES_round/u15_AES_SBox/U275/ZN (CKND2D1BWP)	0.087	0.910 f
u1_AES_Top/u1_AES_round/u15_AES_SBox/odata[5] (AES_SBox_18)	0.000	0.910 f
u1_AES_Top/u1_AES_round/U201/ZN (XNR2D1BWP)	0.155	1.065 r
u1_AES_Top/u1_AES_round/U21/Z (XOR3D2BWP)	0.242	1.306 f
u1_AES_Top/u1_AES_round/odata[69] (AES_round_0)	0.000	1.306 f
u1_AES_Top/U450/Z (XOR2D1BWP)	0.127	1.433 r
u1_AES_Top/U2584/ZN (NR2XD1BWP)	0.041	1.474 f
u1_AES_Top/U86/ZN (CKND2BWP)	0.027	1.500 r
u1_AES_Top/U2248/ZN (CKND2D2BWP)	0.032	1.533 f
u1_AES_Top/rstate_reg_1__69_/D (SDFCNQD4BWP)	0.000	1.533 f
data arrival time		1.533
clock iclk (rise edge)	1.660	1.660
clock network delay (ideal)	0.000	1.660
u1_AES_Top/rstate_reg_1__69_/CP (SDFCNQD4BWP)		1.660 r

library setup time	-0.127	1.533
data required time		1.533

data required time		1.533
data arrival time		-1.533

slack (MET)		0.000

Anexo B

Bancada de Teste

Neste anexo é apresentado o código da bancada de teste criada, sendo esta igual para todas as arquiteturas.

```
`timescale 1ns / 1ps

module tb_AES_top;

// Inputs
reg          tb_iclk   ;
reg          tb_first_n ;
reg          tb_istart ;
reg [127:0]  tb_ikey   ;
reg [127:0]  tb_itext  ;
reg          tb_i_en   ;

// Outputs
wire         tb_odone   ;
wire [127:0] tb_otext   ;
wire         tb_oerror  ;

reg [31:0]   tb_error_total      ;
reg [31:0]   tb_error_odone      ;
reg [31:0]   tb_error_istart     ;
reg [31:0]   tb_count_odone      ;
reg [31:0]   tb_count_istart     ;
reg          tb_error_tmp        ;
reg          tb_odone_flag       ;
reg          tb_iflag            ;
reg          tb_oflag            ;
reg [32:0]   tb_i               ;
reg [32:0]   tb_o               ;

endmodule
```

```

reg [127:0]  tb_mem_otext[0:283]  ;
reg [127:0]  tb_mem_itext [0:135] ;
reg [127:0]  tb_mem_key[0:148]   ;
parameter    input_zero = 128'h00000000000000000000000000000000;

// Instantiate the Unit Under Test (UUT)
AES_topp uut (
    .iclk      (tb_iclk)      ,
    .irst_n    (tb_irst_n)    ,
    .istart    (tb_istart)    ,
    .i_en      (tb_i_en)      ,
    .ikey      (tb_ikey)      ,
    .itext     (tb_itext)     ,
    .odone     (tb_odone)     ,
    .otext     (tb_otext)     ,
    .oerror    (tb_oerror)    ,
    .iscanin   (1'b0)        ,
    .iscanen   (1'b0)        ,
    .iscanmode (1'b0)        ,
    .iscanclk  (1'b0)        ,
    .iscanrst_n (1'b1)       ,
    .oscanout   ()

);

`ifdef DUMP_DEPTH
    initial
    begin
        `ifdef VCS
            $vcdplusfile("test.vpd");
            `ifdef DUMP_MEM
                $vcdplusmemon;
            `endif
            $vcdplusglitchon;                //Saves zero time glitches
            $vcdplusautoflushon;             //automatic flushes
            $vcdpluson(`DUMP_DEPTH,tb_AES_topp);
        `else
            //any other simulator
            $dumpfile("test.vcd");
            $dumpvars(`DUMP_DEPTH,tb_AES_topp);
            $dumpon;
        `endif
    end
`endif

```

```

//input test sequence
initial begin
    tb_iclk          = 0          ;
    tb_i             = 0          ;
    tb_o             = 0          ;
    tb_istart        = 0          ;
    tb_i_en          = 1          ;
    tb_first_n       = 1          ;
    tb_itext         = input_zero ;
    tb_ikey          = input_zero ;
    tb_error_total   = 0          ;
    tb_error_odone   = 0          ;
    tb_error_istart  = 0          ;
    tb_count_odone   = 0          ;
    tb_count_istart  = 0          ;
    tb_error_tmp     = 0          ;
    tb_odone_flag    = 0          ;
    tb_iflag         = 0          ;
    tb_oflag         = 0          ;

    $readmemh("memout.data",tb_mem_otext);
    $readmemh("memkey.data",tb_mem_key);
    $readmemh("meminput.data",tb_mem_itext);

#20

    reset_dut(100);

#10

    //AESAVS- AES Algorithm Validation Suite

#100
//Plaintext variation test as stated in AESAVS
    repeat(135) begin
        sample_data(input_zero,tb_mem_itext[tb_i],1'b1);
        tb_i = tb_i + 1;
    end
    $display("*****\n");

#100

```

```

    reset_dut(50);

#100
//key variation test as stated in AESAVS
    repeat(149) begin
        sample_data(tb_mem_key[tb_i],input_zero,1'b1);
        tb_i = tb_i + 1;
    end
    $display("*****\n");

#100
    disable_dut(30);

//istart signal activation outside 5 clock cycle specification interval
    $display("ERROR INSERTION");
    @(posedge tb_iclk)
        tb_istart = 1'b1;
    repeat(3) @(posedge tb_iclk)
        tb_istart = 1'b0;
    @(posedge tb_iclk)
        tb_istart = 1'b1;
    @(posedge tb_iclk)
        tb_istart = 1'b0;
#100

#1000;

    if (tb_error_total == 0) begin
        $display("SUCCESS - test ended correctly");
    end
    else begin
        $display("FAILURE - test ended with %d ERRORS",tb_error_total);
    end

    if (tb_error_istart == 0) begin
        $display("SUCCESS - input data sampled correctly");
    end
    else begin
        $display("FAILURE - input data sampled badly %d times",tb_error_istart);
    end

    if (tb_error_odone == 0) begin
        $display("SUCCESS - output data sampled correctly");
    end

```

```

        end
        else begin
            $display("FAILURE - output data sampled badly %d times",tb_error_odone);
        end

        $finish;
    end

    //Generate Clock
    always #0.8 tb_iclk = ~tb_iclk;

    //Check outputs
    always @ (posedge tb_iclk or negedge tb_first_n)
    begin
        if(!tb_first_n || !tb_i_en) begin
            tb_iflag          <= 1'b0;
            tb_oflag          <= 1'b0;
            tb_i              <= 32'd0;
        end

        if(tb_odone) begin //check output values
            check_values (tb_otext,tb_mem_otext[tb_o],tb_error_tmp);
            tb_odone_flag <= 1'b1;
            tb_error_total <= tb_error_total + tb_error_tmp;
            tb_o <= tb_o +1;
        end

        if(tb_odone) begin
            if((tb_count_odone != 3'd4) && tb_oflag) begin
                // tb_oflag to ignore the first activation,
                // for not missfire the error signal
                $display(" ERROR - data output not sampled at 5 \
                clock cycles %0d", $time);
                tb_error_odone <= tb_error_odone + 1;
                check_error; // check if the error signal is flagged
            end
            tb_count_odone <= 0;
            tb_oflag <= 1;
        end

        // counter for the number of clock cycles that takes
        //place a new activation of odone flag
        if(!tb_odone && tb_oflag)
            tb_count_odone <= tb_count_odone + 1;

        if(tb_istart) begin

```

```

//same case as above, but for input
if(tb_count_istart != 3'd4 && tb_iflag) begin
    $display(" ERROR - data input not sampled at 5 clock cycles %0d", $time);
    tb_error_istart = tb_error_istart + 1;
    #(1)
    check_error;
end
tb_count_istart <= 0;
tb_iflag <= 1;
end

if(!tb_istart && tb_iflag)
    tb_count_istart <= tb_count_istart + 1;
end

//Manage reset (input delay sets duration of reset)
task reset_dut (input [63:0] delay);
begin
    $display("reset_dut -> STAR DUT Async Reset @ %0d", $time);
    tb_irst_n <= 1'b0;
    #(4)
    //check registers one cycle clock after reset signal
    // is asserted
    check_reset;
    #(delay-4);
    tb_irst_n <= 1'b1;
    $display("reset_dut -> END DUT Async Reset @ %0d", $time);
end
endtask

task disable_dut (input [63:0] delay);
begin
    $display("disable_dut -> STAR DUT Sinc Disable @ %0d", $time);
    tb_i_en <= 1'b0;
    #(10)
    //check registers one cycle clock after reset i_en
    //as been deasserted
    check_reset;
    #(delay-10);
    tb_i_en <= 1'b1;
    $display("disable_dut -> END DUT Sync Disable @ %0d", $time);
end
endtask

//Load values to DUT (no delay assumed)

```

```

task sample_data (input [127:0] ikey, input [127:0] itext, input istart);
begin
    @(posedge tb_iclk) begin
        $display("load_ikey_itext_start -> Load ikey=%h, itext=%h,
            istart=%d @ %0d",ikey,itext,istart,$time);
        tb_ikey    <= ikey;
        tb_itext    <= itext;
        tb_istart   <= istart;
    end
    repeat (4)@(posedge tb_iclk)
        tb_istart <= 1'b0;
    end
endtask

//Compare values
task check_values (input [127:0] iread_text, \
input [127:0] iexpected_text, output error);
begin
    if (iread_text != iexpected_text) begin
        $display("ERROR!! - check_otext -> Read otext=%h != Expected
            otext=%h @ %0d",iread_text,iexpected_text,$time);
        error = 1;
    end
    else begin
        $display("SUCESS!! - check_otext -> Read otext=%h == Expected
            otext=%h @ %0d",iread_text,iexpected_text,$time);
        error = 0;
    end
end
endtask

task check_reset;
begin
    if((!tb_otext) && (!tb_oerror) && (!tb_odone)) begin
        $display("SUCESS!!! - Register reset done correctly @ %0d", $time);
    end
    else begin
        $display("FAIL!! - Register reset gone wrong @ %0d", $time);
    end
end
endtask

task check_error;
begin
    if (tb_oerror) begin

```

```
        $display("SUCESS!!! - Error Signal displayed correctly @ %0d",$time);
    end
    else begin
        $display("FAIL!!! - Error Signal not displayed correctly @ %0d",$time);
    end
end
endtask

endmodule
```

Anexo C

Resultados dos Padrões de teste

Neste anexo são apresentados os resultados obtidos com a ferramenta TetraMax, onde é apresentada a cobertura do teste, falhas detetadas, não detetadas e não testadas.

Arquitetura AES16box2

Uncollapsed Transition Fault Summary Report

fault class	code	#faults
Detected	DT	162694
detected_by_simulation	DS	(160010)
detected_by_implication	DI	(2684)
Possibly detected	PT	0
Undetectable	UD	27
undetectable-tied	UT	(4)
undetectable-redundant	UR	(23)
ATPG untestable	AU	6064
atpg_untestable-not_detected	AN	(6064)
Not detected	ND	427
not-controlled	NC	(74)
not-observed	NO	(353)
total faults		169212
test coverage		96.16%

Inactive Fault Summary Report

fault model	total faults	test coverage
Stuck	173296	3.12%

Pattern Summary Report

#internal patterns	466
--------------------	-----

#basic_scan patterns	1
#fast_sequential patterns	465
# 3-cycle patterns	465
# 1-load patterns	465

Uncollapsed Stuck Fault Summary Report

fault class	code	#faults
Detected	DT	172557
detected_by_simulation	DS	(167156)
detected_by_implication	DI	(5401)
Possibly detected	PT	0
Undetectable	UD	36
undetectable-tied	UT	(2)
undetectable-redundant	UR	(34)
ATPG untestable	AU	703
atpg_untestable-not_detected	AN	(703)
Not detected	ND	0
total faults		173296
test coverage		99.59%

Inactive Fault Summary Report

fault model	total faults	test coverage
Transition	169212	96.16%

Pattern Summary Report

#internal patterns	22
#basic_scan patterns	22

Arquitetura AES16box

Uncollapsed Transition Fault Summary Report

fault class	code	#faults
Detected	DT	167992
detected_by_simulation	DS	(160668)
detected_by_implication	DI	(7324)
Possibly detected	PT	0

Undetectable	UD	10
undetectable-redundant	UR	(10)
ATPG untestable	AU	4332
atpg_untestable-not_detected	AN	(4332)
Not detected	ND	3118
not-controlled	NC	(6)
not-observed	NO	(3112)

```

-----
total faults          175452
test coverage        95.75%
-----

```

Inactive Fault Summary Report

fault model	total faults	test coverage
-----	-----	-----
Stuck	186518	7.88%

Pattern Summary Report

#internal patterns	465
#basic_scan patterns	1
#fast_sequential patterns	464
# 3-cycle patterns	464
# 1-load patterns	464

Uncollapsed Stuck Fault Summary Report

Uncollapsed Stuck Fault Summary Report

fault class	code	#faults
-----	----	-----
Detected	DT	186490
detected_by_simulation	DS	(171787)
detected_by_implication	DI	(14703)
Possibly detected	PT	0
Undetectable	UD	17
undetectable-redundant	UR	(17)
ATPG untestable	AU	11
atpg_untestable-not_detected	AN	(11)
Not detected	ND	0

```

-----
total faults          186518
test coverage        99.99%
-----

```

Inactive Fault Summary Report

fault model	total faults	test coverage
Transition	175452	95.75%

Pattern Summary Report

#internal patterns	45
#basic_scan patterns	45

Arquitetura AES8box

Uncollapsed Transition Fault Summary Report

fault class	code	#faults
Detected	DT	187633
detected_by_simulation	DS	(176065)
detected_by_implication	DI	(11568)
Possibly detected	PT	0
Undetectable	UD	10
undetectable-tied	UT	(4)
undetectable-redundant	UR	(6)
ATPG untestable	AU	5610
atpg_untestable-not_detected	AN	(5610)
Not detected	ND	7159
not-observed	NO	(7159)
total faults		200412
test coverage		93.63%

Inactive Fault Summary Report

fault model	total faults	test coverage
Stuck	218026	10.70%

Pattern Summary Report

#internal patterns	323
#basic_scan patterns	1
#fast_sequential patterns	322
# 3-cycle patterns	322
# 1-load patterns	322

Uncollapsed Stuck Fault Summary Report

fault class	code	#faults
Detected	DT	217356
detected_by_simulation	DS	(194037)
detected_by_implication	DI	(23319)
Possibly detected	PT	0
Undetectable	UD	10
undetectable-tied	UT	(2)
undetectable-redundant	UR	(8)
ATPG untestable	AU	523
atpg_untestable-not_detected	AN	(523)
Not detected	ND	137
not-observed	NO	(137)
total faults		218026
test coverage		99.70%

Inactive Fault Summary Report

fault model	total faults	test coverage
Transition	200412	93.63%

Pattern Summary Report

#internal patterns	36
#basic_scan patterns	36

Arquitetura AES4box

Uncollapsed Transition Fault Summary Report

fault class	code	#faults
Detected	DT	227884
detected_by_simulation	DS	(210452)
detected_by_implication	DI	(17432)
Possibly detected	PT	0
Undetectable	UD	6
undetectable-tied	UT	(6)
ATPG untestable	AU	32249
atpg_untestable-not_detected	AN	(32249)
Not detected	ND	125
not-controlled	NC	(125)

```

-----
total faults                260264
test coverage               87.56%
-----

```

Inactive Fault Summary Report

```

-----
fault model      total faults  test coverage
-----
Stuck            286914       12.29%
-----

```

Pattern Summary Report

```

-----
#internal patterns                473
  #basic_scan patterns             1
  #fast_sequential patterns       472
    # 3-cycle patterns            472
    # 1-load patterns             472
-----

```

Uncollapsed Stuck Fault Summary Report

```

-----
fault class          code  #faults
-----
Detected             DT    284253
  detected_by_simulation DS  (249004)
  detected_by_implication DI (35249)
Possibly detected    PT      0
Undetectable         UD    2535
  undetectable-tied   UT     (3)
  undetectable-redundant UR  (2532)
ATPG untestable      AU     124
  atpg_untestable-not_detected AN (124)
Not detected         ND      2
  not-observed       NO     (2)
-----
total faults                286914
test coverage             99.96%
-----

```

Inactive Fault Summary Report

```

-----
fault model      total faults  test coverage
-----
Transition       260264       87.56%
-----

```

Pattern Summary Report

#internal patterns	38
#basic_scan patterns	38

Anexo D

Caminho crítico da Síntese para FPGA

Neste anexo são apresentados os caminhos críticos obtidos na síntese para FPGA com a ferramenta Synplify.

Arquitetura AES16box2

Path information for path number 1:

Requested Period: 3.330
- Setup time: 0.001
= Required time: 3.329

- Propagation time: 3.323
= Slack (critical) : +0.006

Number of logic level(s): 4
Starting point: rstate_1_[24] / Q
Ending point: rstate_1_[24] / D
The start point is clocked by iclk [rising] on pin C
The end point is clocked by iclk [rising] on pin C

Instance / Net Name	Type	Pin Name	Pin Dir	Delay	Arrival Time	No. of Fan Out (s)
rstate_1_[24]	FDCE	Q	Out	0.375	0.375	-
rstate_1_[24]	Net	-	-	0.531	-	8
u2_AES_round.u13_AES_SBox.odata_14_7	ROM256X1	A4	In	-	0.906	-
u2_AES_round.u13_AES_SBox.odata_14_7	ROM256X1	O	Out	0.539	1.444	-
wstate_aux\[12\[7]	Net	-	-	0.477	-	12
u2_AES_round.u2_AES_SBox.odata_14_7_RNIGHOA	LUT2	I0	In	-	1.921	-
u2_AES_round.u2_AES_SBox.odata_14_7_RNIGHOA	LUT2	O	Out	0.172	2.093	-
N_469_i	Net	-	-	0.421	-	3
u2_AES_round.u12_AES_SBox.odata_10_0_RNI7K5P2	LUT6	I5	In	-	2.514	-
u2_AES_round.u12_AES_SBox.odata_10_0_RNI7K5P2	LUT6	O	Out	0.246	2.760	-
N_553	Net	-	-	0.392	-	2
u2_AES_round.u13_AES_SBox.odata_10_0_RNISQK26	LUT6_L	I5	In	-	3.152	-
u2_AES_round.u13_AES_SBox.odata_10_0_RNISQK26	LUT6_L	LO	Out	0.172	3.324	-
odata_10_0_RNISQK26	Net	-	-	0.000	-	1
rstate_1_[24]	FDCE	D	In	-	3.324	-

=====
Total path delay (propagation time + setup) of 3.324 is 1.531(44.2%) logic and 1.793(55.8%) route.
Path delay compensated for clock skew. Clock skew is added to clock-to-out value, and is subtracted from setup time value

Arquitetura AES16box

Path information for path number 1:

Requested Period: 3.330
- Setup time: -0.080
= Required time: 3.410

- Propagation time: 3.417
= Slack (critical) : -0.007

```

Number of logic level(s):          4
Starting point:                    svbl_69.rstate_0_[64] / Q
Ending point:                      svbl_69.rstate_0_[43] / D
The start point is clocked by      iclk [rising] on pin C
The end point is clocked by        iclk [rising] on pin C

```

Instance / Net Name	Type	Pin Name	Pin Dir	Delay	Arrival Time	No. of Fan Out (s)
svbl_69.rstate_0_[64]	FDCE	Q	Out	0.375	0.375	-
rstate_0_[64]	Net	-	-	0.531	-	8
u1_AES_round.u8_AES_SBox.svbl_68.odata_14_7	ROM256X1	A4	In	-	0.906	-
u1_AES_round.u8_AES_SBox.svbl_68.odata_14_7	ROM256X1	O	Out	0.539	1.444	-
wstate_aux\[11\][7]	Net	-	-	0.723	-	28
svbl_69.rstate_1__15_iv_83_x4_0_a2_0_x	LUT5	I4	In	-	2.167	-
svbl_69.rstate_1__15_iv_83_x4_0_a2_0_x	LUT5	O	Out	0.402	2.569	-
rstate_1__15_iv_83_x4_0_a2_0_x	Net	-	-	0.384	-	2
svbl_69.rstate_0__10_m1[43]	LUT5	I4	In	-	2.953	-
svbl_69.rstate_0__10_m1[43]	LUT5	O	Out	0.086	3.039	-
svbl_69.rstate_0__10_m1[43]	Net	-	-	0.206	-	1
svbl_69.rstate_0__10[43]	LUT6_L	I4	In	-	3.245	-
svbl_69.rstate_0__10[43]	LUT6_L	LO	Out	0.172	3.417	-
svbl_69.rstate_0__10[43]	Net	-	-	0.000	-	1
svbl_69.rstate_0_[43]	FDCE	D	In	-	3.417	-

=====
Total path delay (propagation time + setup) of 3.337 is 1.493(44.8%) logic and 1.844(55.2%) route.

Path delay compensated for clock skew. Clock skew is added to clock-to-out value, and is subtracted from setup time value

Arquitetura AES8box

Path information for path number 1:

```

Requested Period:          3.330
- Setup time:              0.001
= Required time:           3.329

- Propagation time:        2.891
= Slack (critical) :       0.438

```

```

Number of logic level(s):    3
Starting point:              u1_AES_keygen.svbl_68.rflag2 / Q
Ending point:                svbl_99.rkin[28] / D
The start point is clocked by iclk [rising] on pin C
The end point is clocked by  iclk [rising] on pin C

```

Instance / Net Name	Type	Pin Name	Pin Dir	Delay	Arrival Time	No. of Fan Out (s)
u1_AES_keygen.svbl_68.rflag2	FDC	Q	Out	0.375	0.375	-
wkvalid	Net	-	-	0.756	-	162
u5_AES_round.m85	LUT6	I5	In	-	1.131	-
u5_AES_round.m85	LUT6	O	Out	0.434	1.566	-
rkey_1__5[92]	Net	-	-	0.586	-	13
u5_AES_round.m183	LUT4	I3	In	-	2.151	-
u5_AES_round.m183	LUT4	O	Out	0.086	2.237	-
rkey_1__5[28]	Net	-	-	0.567	-	11
u5_AES_round.m2579	LUT4_L	I3	In	-	2.805	-
u5_AES_round.m2579	LUT4_L	LO	Out	0.086	2.891	-
N_6593_mux	Net	-	-	0.000	-	1
svbl_99.rkin[28]	FDCE	D	In	-	2.891	-

=====
Total path delay (propagation time + setup) of 2.892 is 0.982(34.0%) logic and 1.909(66.0%) route.

Path delay compensated for clock skew. Clock skew is added to clock-to-out value, and is subtracted from setup time value

Arquitetura AES4box

Path information for path number 1:

```

Requested Period:          3.330
- Setup time:              0.001
= Required time:           3.329

- Propagation time:        3.315
= Slack (critical) :       0.014

```


Referências

- [1] DIGITAL CONTENT PROTECTION. Digital content protection for new home theater networking scenarios, November 2008. URL: <http://www.digital-cp.com> [último acesso em 2014-12-12].
- [2] Ingrid Verbauwhede Alireza Hodjat. Area-throughput trade-offs for fully pipelined 30 to 70 Gbits/s AES processors. *IEEE TRANSACTIONS ON COMPUTERS*, VOL. 55, NO.4, 2006.
- [3] Akashi Satoh, Sumio Morioka, Kohji Takano, e Seiji Munetoh. A compact rijndael hardware architecture with S-box optimization. Em Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, volume 2248 de *Lecture Notes in Computer Science*, páginas 239–254. Springer Berlin Heidelberg, 2001. URL: http://dx.doi.org/10.1007/3-540-45682-1_15.
- [4] Karthick Ramu Chethan Ananth. Fully pipelined implementations of AES with speeds exceeding 20 Gbits/s with S-boxes implemented using logic only. *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
- [5] P. Maistri e R. Leveugle. 10-gigabit throughput and low area for a hardware implementation of the advanced encryption standard. Em *Digital System Design (DSD), 2011 14th Euromicro Conference on*, páginas 266–269, Aug 2011. doi:10.1109/DSD.2011.37.
- [6] Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, e Gaël Rouvroy. *Implementation of the AES-128 on Virtex-5 FPGAs*, volume 5023 de *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008. URL: http://dx.doi.org/10.1007/978-3-540-68164-9_2.
- [7] SYNOPSYS. Corporate backgrounder, 2014. URL: <http://www.synopsys.com/Company/AboutSynopsys/Pages/CompanyProfile.aspx> [último acesso em 2014-01-12].
- [8] IDC1. Hdmi, the digital display link, December 2006. URL: [http://www.hdmi.org/pdf/whitepaper/SilicaonImageHDMIWhitePaperv73\(2\).pdf](http://www.hdmi.org/pdf/whitepaper/SilicaonImageHDMIWhitePaperv73(2).pdf) [último acesso em 2014-01-29].
- [9] FIPS PUBS. Advanced encryption standard (AES). Relatório técnico, Federal Information Preprocessing Standards Publications, November 2001.
- [10] Morris Dworkin. Recommendation for block cipher modes of operation. Relatório técnico, National Institute of Standards and Technology, 2001.
- [11] C. Paar e J. Pelzl. The advanced encryption standard(AES). Em *Understanding-Cryptography*, páginas 87–117. Springer-Verlag, 2010.

- [12] Vincent Rijmen. Efficient implementation of the rijndael S-box. *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.
- [13] Karim M. Abdellatif, Roselyne Chotin-Avot, e Habib Mehrez. The effect of S-box design on pipelined AES using FPGAs. apresentado em *Colloque GDR SoC-SiP 2012*, Maio 2012.
- [14] Akashi Satoh, Takeshi Sugawara, e Takafumi Aoki. High-speed pipelined hardware architecture for galois counter mode. Em *Proceedings of the 10th International Conference on Information Security, ISC'07*, páginas 118–129, Berlin, Heidelberg, 2007. Springer-Verlag. URL: <http://dl.acm.org/citation.cfm?id=2396231.2396242>.
- [15] Alireza Hodjat e Ingrid Verbauwhede. A 21.54 Gbits/s fully pipelined AES processor on FPGA. *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.
- [16] Lawrence E. Bassham III. The Advanced encryption standard algorithm validation suite (AESAVS). Relatório técnico, Federal Information Preprocessing Standards Publications, November 2002.
- [17] *Leda® User Guide, Version I-2014.03*.
- [18] *VCS®/VCSiTM User Guide, Version I-2014.03-2*.
- [19] *Design Compiler® User Guide, Version I-2013.12-SP4*.
- [20] *Formality® User Guide, I-2013.12-SP2*.
- [21] *TetraMAX® ATPG User Guide, Version I-2013.12-SP4*.
- [22] *PrimeTime® User Guide, Version I-2014.03-2*.