Implementação em verilog de criptografia AES-128 com protocolo de comunicação I²C

Andre Luiz E. Araujo
andrediaraujo@gmail.com

Daniella Vicentini Azevedo dos Santos
daniella.vicentini@yahoo.com.br

Guilherme Henrique Duarte Mendes
gui_mendes10@hotmail.com

Janaína G. M. Oliveira
janaina.gmoliveira@gmail.com

Lucas Manoel Leite de Souza
lmanoelso@gmail.com

Maria Teresa Rocha Carvalho
mariateresarochacarvalho@gmail.com

Matheus Henrique Martins Paiva
mtpaivamrhenrique@gmail.com

Sérgio Henrique Azevedo dos Santos
shasantos2005@gmail.com

Felipe Gustavo de Freitas Rocha
felipef.rocha@inatel.br

***Resumo*—Este artigo descreve a implementação em verilog de um encriptador e decriptador AES-128, onde a comunicação de entrada é feita através do protocolo I²C. A palavra a ser criptografada/decriptografada, a chave desta operação e o tipo de operação a ser feita (criptografia ou decriptografia) são recebidos de forma serial e então passados a um circuito de controle que sincroniza todas as operações necessárias. A saída é disponibilizada de forma paralela. O circuito foi desenvolvido de forma que cada bloco da criptografia é independente e assíncrono, facilitando o processo de escrita e teste do código. Todos os módulos desenvolvidos foram testados individualmente e em conjunto, validando sua operação.**

***Keywords*—I²C, AES128, Verilog, FPGA**

I. INTRODUÇÃO

I²C ou *Inter-Integrated Circuit* (Circuito Inter-Integrado) é um protocolo de comunicação serial do tipo mestre-escravo desenvolvido pela Philip Semicondutores [1]. Tem baixa velocidade e é usado para conectar dispositivos como microcontroladores e processadores, à dispositivos alvo tais como Assistentes Digitais Pessoais (PDA), DVD's, ADCs, DACs, LCDs, memórias, entre outros. Como não é afetado por fatores externos, não tem perda de dados, aumentando sua eficiência [1, 2]. Em geral qualquer número de escravos (receptores) ou mestres (transmissores) é permitido. Estes dispositivos se comunicam através de um protocolo com endereço, dados e bits de controle.

O I²C é usado para comunicação intra barramento, ou seja, associa dispositivos que pertencem ao mesmo sistema. Usa 2 linhas bidirecionais de sinal, que são chamadas de SDA (*Serial Data*) e SCL (*Serial Clock*) [1,3]. Cada dispositivo conectado tem um endereço que o identifica, o que faz com que o protocolo não precise de seleção do dispositivo escravo ou qualquer outro tipo de lógica [1].

Durante a comunicação, um mestre é capaz de ler e escrever no barramento. Qualquer dispositivo conectado fica o escutando permanentemente, aguardando um sinal que indica o início de uma transmissão [3].

A criptografia é um componente da segurança de informação, que usa técnicas matemáticas para garantir a privacidade e autenticidade dos dados, além de manter a integridade dos mesmos e verificar sua origem [4,5].

O algoritmo de Rijndael ou AES (*Advanced Encryption Standard*) é um modelo de criptografia adotado como padrão pelo *National Institute of Standards and Technology* (NIST) em 2001 e descrito na FIPS-197[6]. Este modelo opera com chave simétrica de tamanho variável entre 128, 192 ou 256 bits. A palavra de entrada tem tamanho fixo de 128 bits. A cifragem é feita através de rodadas, executando substituições a nível de byte e permutações de blocos de bytes, até que se obtenha uma cifra na saída [4]. A quantidade de rodadas a ser executada depende do tamanho da chave, podendo ser 10 rodadas para 128 bits, 12 rodadas para 192 bits e 14 rodadas para 256 bits.

Neste trabalho foi desenvolvido um módulo escravo do protocolo I²C que recebe os dados necessários para criptografar/decriptografar uma mensagem. Foi implementada a criptografia AES com chave de 128 bits e 10 rodadas de processamento.

II. FUNDAMENTAÇÃO TEÓRICA

No algoritmo AES, os bytes são interpretados como elementos de campo finito, permitindo uma descrição matemática na forma de polinômios de todas as operações executadas durante a criptografia. Internamente, os bytes de entrada são organizados por colunas na forma de uma matriz, chamada de matriz estado, para que as operações possam ser executadas. Tanto a palavra e a chave de entrada quanto a cifra de saída têm 128 bits e podem ser representadas através dessa matriz.

Na criptografia e na decriptografia o algoritmo executa rodadas compostas por 4 transformações a nível de byte: *SubByte* ou substituição de bytes, *ShiftRows* ou deslocamento de linhas, *MixColumns* ou embaralhamento das colunas e *AddRoundKey* ou soma da chave ao estado.

A Figura 1 mostra o diagrama de blocos do fluxo da criptografia e decriptografia AES-128.

A criptografia inicia com a adição da chave à palavra de entrada. A matriz de estado gerada é então inserida em um *loop* de 10 rodadas e transformada em cada uma delas através das 4 operações citadas. A rodada final é levemente diferente das 9 primeiras rodadas pois a operação *MixColumns* não é necessária. Ao final das 10 rodadas a cifra é então disponibilizada na saída representando a palavra criptografada com a chave de entrada. Cada rodada utiliza uma subchave diferente previamente gerada através da expansão de chave (*ExpansionKey*).

Na decriptografia, a 10ª subchave gerada é adicionada à cifra de entrada e então 10 rodadas de operação se iniciam.

As funções *InvShiftRows*, *InvSubBytes* e *InvMixColumns* são inversas às usadas na criptografia. A função *AddRoundKey* é a mesma, o que a difere da criptografia é a ordem em que as subchaves geradas são usadas, que é inversa. A rodada de número 10 difere das 9 primeiras já que a função *InvMixColumns* não é necessária [6].

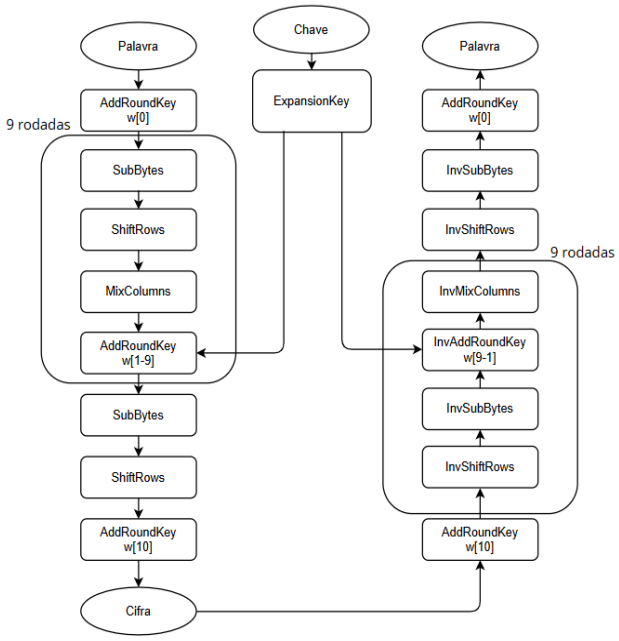

Fig. 1. Fluxo da criptografia e decriptografia

Cada uma das operações é detalhada a seguir.

A. *SubBytes e InvSubBytes*

É a substituição não linear e independente dos bytes de entrada por valores previamente calculados e mantidos em uma tabela de substituição *S-box* ou *InvS-box*.

Cada byte do estado é localizado na *S-box* (ou *InvS-box*) onde os 4 bits mais significativos estão na linha e os 4 bits menos significativos estão na coluna. O valor encontrado substitui o valor do byte na matriz de estado.

B. *ShiftRows e InvShiftRows*

São responsáveis pela difusão dos dados na matriz de estado. Na *ShiftRows* é feito um deslocamento para a esquerda e na *InvShiftRows*, para a direita. A linha 0 permanece inalterada enquanto as linhas 1, 2 e 3 são deslocadas de 1, 2 ou 3 posições.

C. *MixColumns e InvMixColumns*

São responsáveis por misturar os bytes de cada coluna da matriz de estado promovendo difusão dos dados cifrados e dificultando ataques ao texto. Cada byte de saída depende de todos os bytes da coluna de entrada.

Esta função opera coluna a coluna do estado, tratando cada coluna como um polinômio e o multiplicando por um outro polinômio reversível de valor fixo. A função *InvMixColumns* transforma cada coluna através da multiplicação por um polinômio inverso ao usado na *MixColumns*. As operações são realizadas de acordo com um campo finito ou Campo de Galois.

D. *AddRoundKey*

Nesta função, a cada rodada uma subchave diferente é adicionada ao estado através de um *XOR*. As subchaves são geradas através da função *ExpansionKey*. Na criptografia as chaves são usadas na ordem crescente e na decriptografia na ordem decrescente.

E. *ExpansionKey*

É responsável por gerar um conjunto de subchaves para as rodadas a partir da chave original da entrada. Ao total são geradas 44 subchaves necessárias para as 10 rodadas.

A *ExpansionKey* utiliza uma lógica iterativa onde para cada palavra w[i], se $i < 4$, a subchave é igual a chave original. Se $i \geq 4$, a nova subchave será gerada a partir de w[i-1] e w[i-4], com as seguintes possibilidades:

- Se o resto da divisão de i por 4 é igual a 0, aplica-se a rotação dos 4 bytes da palavra para a esquerda (*RotWord*), a substituição byte a byte (*SubWord* com a *S-box*) e *XOR* com a constante da rodada *Rcon*, para por fim realizar a operação *XOR* novamente entre o valor obtido após as três operações citadas com w[i-4].
- Para as demais situações, $w[i] = w[i-1] \oplus w[i-4]$.

A *Rcon* é uma constante já fornecida pela FIPS-197.

Neste trabalho foram implementadas, na linguagem Verilog, tanto a criptografia quanto a decriptografia com chaves de 128 bits. Os módulos foram testados e validados segundo dados fornecidos na FIPS-197.

III. ARQUITETURA PROPOSTA

A arquitetura desenvolvida para este projeto, composta por 4 grandes blocos, pode ser vista na Figura 2.

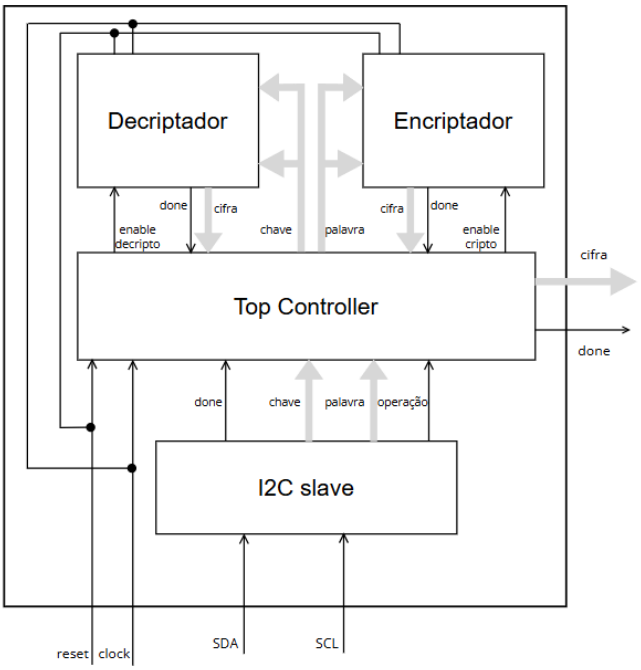

Fig. 2. Arquitetura proposta

A. *Top Controller*

O *TopController* é o bloco responsável por coordenar a execução de todo o circuito. Ele recebe os dados do I²C, os

envia para o encriptador ou decriptador e ativa um destes blocos para operar, além de disponibilizar a saída do circuito. É formado por uma máquina de estados (FSM).

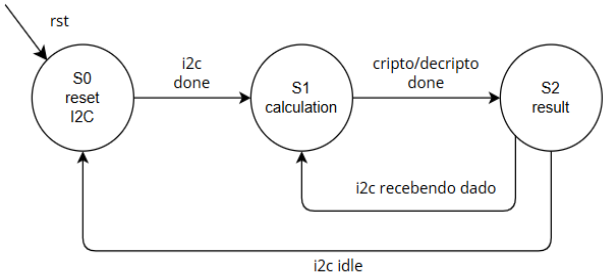

Fig. 3. Máquina de estados - TopController

A execução inicia no estado S0 com um sinal de *reset*, onde as *flags* que habilitam tanto a criptografia como a decriptografia são colocadas em 0, assim como a *flag done* que mostra que a operação foi finalizada. Assim que o módulo I²C envia um sinal indicando que todos os dados foram recebidos, a FSM vai para o estado S1 e a operação pode começar. Se a operação recebida for 0, a criptografia é habilitada e se for 1, a decriptografia é habilitada. A máquina permanece no estado S1 até que ou a criptografia ou a decriptografia envie uma *flag* indicando o fim da operação. A FSM passa então ao estado S2 onde a *flag* de *done* do circuito é habilitada e a saída é disponibilizada. Caso o sinal de *start* do I²C seja 0, o que indica que algum dado está sendo recebido na entrada, a máquina vai para o estado S0, caso contrário, ou seja, novos dados já foram recebidos, a máquina vai para o estado S1.

Com esta máquina de estados pode-se garantir a execução sincronizada de todos os blocos que compõem o circuito, onde nenhum dado é perdido e somente quando os valores estiverem estáveis a saída é disponibilizada. A maneira que a máquina foi desenvolvida também garante que várias entradas podem ser processadas sequencialmente sem a necessidade de um *reset*.

B. *I²C Slave*

O módulo slave I²C recebe 33 bytes de dados. Isso inclui uma palavra de 16 bytes, uma chave de 16 bytes e 1 byte de indicador de operação (criptografia ou decriptografia). A implementação lida apenas com operações de escrita, onde os dados recebidos são salvos e disponibilizados para o módulo *TopController*, para que seja executada a criptografia ou decriptografia.

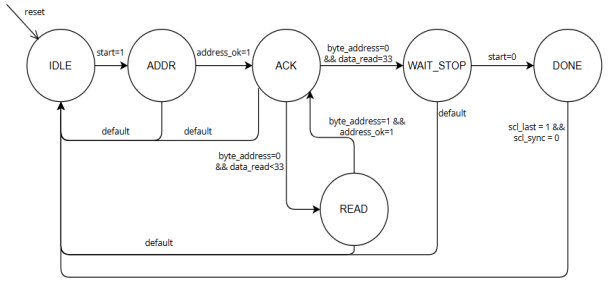

Fig. 4. Máquina de estados - I²C

O controle é feito através de uma máquina de estados (Figura 4), onde no estado *IDLE* uma condição de *start* é aguardada. Assim que recebida, a FSM vai para o estado *ADDR*, onde o byte de endereço é recebido. Em seguida, passa ao estado *ACK*, onde um sinal de reconhecimento é enviado. No estado *READ* os bytes de dados são lidos. Então é aguardada uma condição de *stop* no estado *WAIT_STOP*. Finalmente, quando todos os dados foram recebidos, o estado *DONE* é iniciado e em seguida transicionado de volta para *IDLE*.

O endereço definido para este dispositivo escravo foi 6Ah.

C. *Encriptador e Decriptador*

O bloco encriptador é responsável por criptografar a palavra recebida usando a chave, também recebida. É formado por uma máquina de estados (*ControllerCriptografia*) que controla a execução e os blocos responsáveis por cada etapa da operação (Figura 5).

Assim que o circuito recebe um *reset*, a FSM vai para o estado S0, onde os registradores são zerados. Quando um sinal de *start* do *TopController* é recebido, o estado S1 inicia a execução da expansão da chave de entrada. No próximo ciclo de *clock*, a FSM vai para o estado S2, onde as chaves calculadas no estado anterior são divididas em 11 registradores de 128 bits para serem usadas em cada rodada. Também, a primeira execução do *AddRoundKey* é feita entre a palavra da entrada e a subchave gerada para a rodada 0. A máquina vai então para o estado S3 executar a operação de *SubBytes*, então para o estado S4, onde a operação *ShiftRows* é executada. O contador é então checado: se for 10, ou seja, é a última rodada a ser executada, a FSM vai para o estado S6. Caso contrário, vai para o estado S5 executar o *MixColumns*. No estado S6 a subchave correspondente à rodada é usada no *AddRoundKey* e em seguida, se esta não for a rodada 10, a FSM vai para o estado S3 iniciar uma nova rodada. Caso esta seja a última rodada, o estado S7 é executado, a saída é disponibilizada e uma *flag* indicando que a criptografia terminou é habilitada. A máquina volta ao estado S0 e fica esperando um novo sinal de *start* para iniciar um novo processamento.

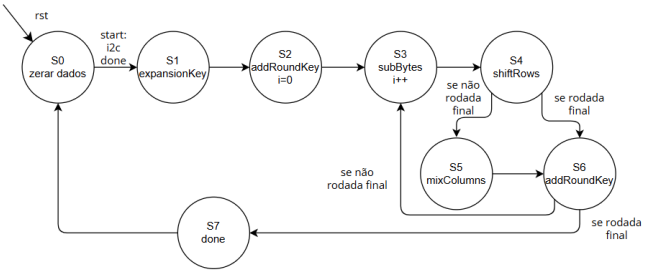

Fig. 5. Máquina de estados - Controle Criptografia

No desenvolvimento deste controlador, foi necessária atenção à passagem entre os estados e à alocação da saída de um bloco na entrada de outro. Como o processamento é serial, ou seja, a cada ciclo de *clock* uma função é executada, os valores de saída são válidos somente no estado em que a função foi executada. Este processamento serial permite uma menor área de circuito ao custo de uma maior complexidade no funcionamento e maior tempo de execução. Como a criptografia AES já tem um tempo alto e não é muito utilizada em projetos de alta velocidade, optou-se por garantir uma área menor para o circuito.

O bloco decriptador também é formado por uma máquina de estados (*ControllerDecriptografia*) que controla a execução dos blocos das operações (Figura 6).

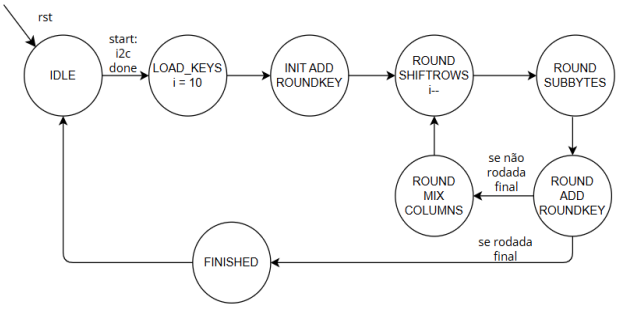

Fig. 6. Máquina de estados - Controle Decriptografia

O estado *IDLE* aguarda o sinal de *done* do I²C, indicando que o processamento pode iniciar e que os dados estão estáveis. No estado *LOAD_KEYS* os dados gerados na *ExpansionKey* são guardados e então a rodada inicial *AddRoundKey* com a última chave gerada é executada (estado *INIT_ADDROUNDKEY*). Iniciam-se então as rodadas intermediárias nos estados *ROUND_SHIFTROWS*, *ROUND_SUBBYTES*, *ROUND_ADDROUNDKEY* e *ROUND_MIXCOLUMNS*. A cada rodada finalizada o contador é decrementado até chegar ao valor 0 (na última *AddRoundKey*). Assim é possível voltar exatamente ao estado original.

A seguir, a implementação de cada bloco é detalhada.

1) SubBytes e InvSubBytes

Substituição dos bytes de entrada pelos valores da *S-box* e da *InvS-box* através de pesquisa em um módulo com os valores possíveis. Estes dois módulos são implementados como *case* com todos os 256 valores possíveis para os bytes.

2) ShiftRows e InvShiftRows

Ambos os módulos operam sobre um vetor de entrada de 128 bits. A saída consiste no vetor re-ordenado conforme o padrão: manter a linha 0 inalterada, deslocar a linha 1 em 1 posição, deslocar a linha 2 em 2 posições e deslocar a linha 3 em 3 posições. No caso da *ShiftRows* o deslocamento é feito para a esquerda e da *InvShiftRows*, é feito para a direita.

3) MixColumns e InvMixColumns

Nas multiplicações de matrizes, a adição é a operação *XOR* realizada bit a bit. A multiplicação por 2 (*xtime*) pode ser interpretada por um deslocamento à esquerda, onde se o bit mais significativo do resultado for 1, deve-se realizar um *XOR* com 0x1b para garantir que o resultado permaneça correto em 1 byte. A multiplicação por 3 (0x03) é feita com multiplicação por 2 seguida de uma soma (*XOR*), ou seja: *xtime*(x)⊕ x.

4) AddRoundKey

O módulo aplica a operação *XOR* diretamente entre a matriz de estados de entrada e a subchave da rodada. A diferença no uso desta função na criptografia e decriptografia está na ordem de uso das subchaves: enquanto a primeira usa as subchaves na ordem crescente, a segunda as usa na ordem decrescente.

5) ExpansionKey

O módulo gera 44 palavras de 32 bits formando as 11 subchaves que serão usadas ao longo do algoritmo. A saída é um vetor de 1408 bits com as subchaves sequenciais.

Inicialmente, as palavras **w[0] a w[3]** são diretamente extraídas da chave original de 128 bits. As palavras subsequentes **w[4] até w[43]** são geradas iterativamente utilizando operações padrão definidas na especificação do AES:

Quando o índice i é múltiplo de 4, aplica-se:

RotWord(): rotação cíclica à esquerda dos bytes da palavra.
SubWord(): substituição de cada byte pela saída da S-box.
XOR com a constante **Rcon[i/4]**.

Nos demais casos (quando i não é múltiplo de 4), a palavra é gerada com uma simples operação:

$$\mathbf{w[i] = w[i-1] \oplus w[i-4]}$$

Ao final, as 44 palavras formam as 11 subchaves de 128 bits, e o vetor de 1408 bits é disponibilizado na saída, juntamente com uma flag de conclusão.

III. RESULTADOS E SIMULAÇÕES

Após a implementação de cada bloco, testes foram feitos através de arquivos de *testbench*.

A. Testbench I²C

O teste simula uma comunicação I²C, onde a *testbench* atua como mestre. O módulo deve receber os 33 bytes corretamente, respondendo com um *ACK* a cada byte enviado. A linha SDA é bidirecional: se estiver em alta impedância, o escravo pode enviar o *ACK*. Caso contrário, o mestre define o valor da linha.

No teste foi verificado que o módulo escravo recebe todos os 264 bits enviados corretamente, respondendo com *ACKs* nos momentos apropriados, seguindo o protocolo I²C.

B. Testbench S-box e InvS-box

Foi utilizado o comando *for* para gerar os 256 valores possíveis (00 a ff). Os resultados obtidos estavam conforme esperado.

C. Testbench SubBytes e InvSubBytes

Foram realizados testes com 12 valores, sendo as rodadas 1 a 10 seguindo vetores disponibilizados na FIPS-197, a rodada 11 com 128 bits em 0 e a rodada 12 com 128 bits em 1.

A simulação inicial do *SubBytes* teve a saída em alta impedância, indicando erro pois o módulo *S-box* não havia sido integrado ao projeto no *ModelSim*. Após a correção, os testes ocorreram da maneira esperada.

Para o *InvSubBytes*, na simulação inicial os valores recebidos foram diferentes dos esperados. Isto ocorreu pois este módulo estava instanciando a *InvS-box* com entrada de 128 bits ao invés de 8 bits. Também, os valores usados na *testbench* eram da Equivalente Inversa. Após as correções, os resultados gerados foram iguais aos esperados.

D. Testbench ShiftRows e InvShiftRows

Foi utilizado o *for* para geração de vetores aleatórios com *$random* e *task* para criação dos modelos de referência, garantindo um valor esperado para comparação dos resultados.

Foram executados 10 ciclos de testes distintos para cada módulo, cobrindo ampla variedade de padrões de bits, com todos os resultados conforme o esperado.

E. *Testbench MixColumns e InvMixColumns*

Foram realizados testes com 12 valores de entrada, para *MixColumns* e *InvMixColumns*, sendo: rodadas 0 a 9 seguindo vetores disponibilizados na FIPS-197. Rodada 10 com bits em 0 e rodada 11 com bits em 1.

As saídas foram analisadas e comparadas a cada rodada com os valores esperados, garantindo a funcionalidade do módulo da maneira esperada.

F. *Testbench AddRoundKey*

Foram realizados testes com 5 valores distintos, sendo: rodadas 0 a 2 seguindo vetores extraídos diretamente da FIPS-197, rodada 3 com todos os bits do vetor em 0 e rodada 4 com todos os bits do vetor em 1.

A saída foi analisada e comparada, a cada rodada, com os valores esperados de acordo com a especificação, garantindo a funcionalidade correta do módulo.

G. *Testbench ExpansionKey*

O módulo foi testado utilizando como base uma chave descrita na FIPS-197, uma chave com todos os bits zerados, uma chave com todos os bits em 1, uma chave com padrão incremental e uma chave do caso 0 com erro proposital na etapa 5, para validação de robustez.

A cada iteração, a *testbench* compara bit a bit a saída gerada com o vetor de referência. O teste foi considerado bem-sucedido pois os casos de 0 a 3 passaram integralmente, e o caso 4 falhou de maneira controlada, conforme esperado.

Durante a simulação foi verificado um problema na leitura dos dados da *S-box*, sendo necessária uma alteração do código. Com a correção, o módulo não apresentou problemas na execução.

H. *Testbench ControllerCriptogtrafia e ControllerDecriptografia*

O *testbench* foi desenvolvido com 5 valores provenientes de fontes confiáveis: testes 0 e 1 com vetores oficiais da FIPS-197, teste 2 com palavra zerada e chave não zerada, teste 3 com palavras com todos os bits 1 e teste 4 de valor genérico. Todos os testes puderam ser validados através do site *https://testprotect.com*. Os resultados estavam corretos para todos os casos.

Além da validação dos resultados finais, pode-se também monitorar em tempo real os estados da máquina de estados (FSM), imprimindo os resultados intermediários de cada fase do algoritmo AES (como *AddRoundKey*, *SubBytes*, *ShiftRows* e *MixColumns*), facilitando a depuração por meio de formas de onda e conferência textual.

Os testes de decriptografia são análogos aos testes de criptografia, porém os valores de saída da criptografia são usados como entrada na decriptografia. As chaves são as mesmas. Ao final dos testes, é esperado que o valor de saída seja igual à entrada da criptografia correspondente. Neste projeto todos os resultados estavam corretos.

Assim como na criptografia, pode-se analisar também os estados e etapas intermediárias do processamento.

I. *Testbench Módulo Completo*

Este teste tem como objetivo enviar um conjunto de dados pela linha SDA do protocolo I²C e verificar se a integração entre os módulos está correta e fornece o resultado esperado.

A *testbench* funciona como o mestre do I²C e inicia a comunicação com o módulo aqui desenvolvido. Ela envia através da linha SDA um sinal de *start* e em seguida o endereço do dispositivo. Envia então os 128 bits da palavra e os 128 bits da chave. Após cada byte de dados enviado, o controle da linha SDA é alternado e se aguarda o bit de *ACK* do escravo. A transmissão finaliza com um sinal de *stop*.

Dos 5 testes feitos, 4 são baseados nos vetores da FIPS-197. O 5º teste utiliza dados arbitrários.

Todos os testes foram bem sucedidos. Foram necessários 6201 ciclos de clock a partir do sinal de start para que a saída estivesse pronta.

IV. CONCLUSÃO

A implementação de um módulo de criptografia e decriptografia com comunicação de entrada via protocolo I²C foi executada obtendo-se os resultados esperados. Foi possível verificar toda a comunicação serial e o processamento correto da criptografia e em seguida usar a saída gerada para testar a decriptografia.

Todos os módulos foram testados separadamente, facilitando o teste do circuito completo. A divisão do circuito em blocos pequenos e independentes facilitou o desenvolvimento e os testes. Cada bloco e seu inverso foram desenvolvidos pelo mesmo *designer*, facilitando e agilizando a implementação.

Trabalhos futuros podem ser feitos a fim de implementar a saída seguindo o protocolo I²C, diminuir a área usada e o tempo de processamento.

REFERÊNCIAS

[1] Abhinav Boddupalli, “Design and Implementation of I2C bus protocol on FPGA using verilog for EEPROM”, Proceedings of IEEEFORUM International Conference, 01st October, 2017, Pune, India

[2] "A Basic Guide to I2C" (SBAA565), Texas Instruments, {Link: TI.com https://www.ti.com/lit/pdf/sbaa565}

[3] F. Leens, "An introduction to I2C and SPI protocols," in IEEE Instrumentation & Measurement Magazine, vol. 12, no. 1, pp. 8-13, February 2009, doi: 10.1109/MIM.2009.4762946. keywords: {Master-slave;Clocks;Protocols;Instruments;Frequency;Topology;Codecs;Voltage;Sampling methods}

[4] Yadav, S., Girdhar, G., & Yadav, S. (2024). AES 128 Bit Optimization: High-Speed and Area-Efficient through Loop Unrolling. International Journal of Scientific Research in Engineering and Management, 8(5), 1-5.

[5] SAAD, M. W. (2010). Implementação do algoritmo AES em hardware reconfigurável - FPGA.

[6] National Institute of Standards and Technology (2001) Advanced Encryption Standard (AES). (Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publication (FIPS) NIST FIPS 197-upd1, updated May 9, 2023. https://doi.org/10.6028/NIST.FIPS.197-upd1