Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aceleração da Implementação do Algoritmo de Criptografia AES-128 em um Processador MIPS

Antonio Martino Neto

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Msc. Marcos Fagundes Caetano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Neto, Antonio Martino.

Aceleração da Implementação do Algoritmo de Criptografia AES-128 em um Processador MIPS / Antonio Martino Neto. Brasília : UnB, 2016.
99 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. FPGA, 2. MIPS, 3. AES, 4. corpo finito

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Aceleração da Implementação do Algoritmo de Criptografia AES-128 em um Processador MIPS

Antonio Martino Neto

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Msc. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Bacharelado em Ciência da Computação

Brasília, 26 de fevereiro de 2016

Dedicatória

A Alexandre "kure" Silva Dantas (*in memoriam*).

Agradecimentos

Agradeço ao professor Marcus Lamar pelas aulas de arquitetura de computadores, pela orientação e pela paciência; ao professor Diego Aranha pelas aulas de criptografia; aos amigos, colegas, professores e familiares, pelo apoio que me deram para chegar até aqui.

Resumo

Uma forma de melhorar o desempenho de aplicações é utilizar soluções em *hardware*, que podem variar desde instruções dedicadas em um processador até chips inteiros dedicados exclusivamente a uma tarefa. Neste trabalho, implementamos em Verilog um processador *pipeline* baseado na arquitetura MIPS; adicionamos ao processador uma nova instrução de multiplicação em um corpo finito, um cálculo recorrente no algoritmo AES-128. Com o processador sintetizado em FPGA, compilamos e executamos uma implementação do algoritmo AES-128, criando duas versões do código: uma sem modificações e uma modificada para utilizar a nova instrução; finalmente, comparamos o desempenho das duas versões. Nossos resultados mostram que a nova instrução reduz o tempo de execução de forma expressiva.

Palavras-chave: FPGA, MIPS, AES, corpo finito

Abstract

One way to improve performance of applications is to use harware solutions, which can vary between dedicated instructions in a processor and whole chips dedicated exclusively to one task. In this work, we implement in Verilog a pipeline processor based on the MIPS architecture; we add a new instruction for multiplication in a finite field, a recurring calculation in the AES-128 algorithm. With the processor synthesized in an FPGA, we compile and execute an implementation of the AES-128 algorithm, creating two versions of the code: one without modifications and one modified to use the new instruction; finally, we compare the performance of both versions. Our results show that the new instruction reduces execution time in an expressive way.

Keywords: FPGA, MIPS, AES, finite field

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um computador moderno, seja um smartphone ou um servidor, pode ser dividido em duas partes complementares: hardware e software. A primeira parte corresponde aos componentes físicos, desde transistores, capacitores e resistores até processadores, memórias e dispositivos de armazenamento. A segunda parte engloba dados e programas, de uma simples instrução para adicionar dois números até um sistema operacional completo. A interação entre as duas partes é um ponto fundamental para o correto funcionamento de um computador, sendo objeto de estudo da área de organização e arquitetura de computadores.

A organização de um processador corresponde à disposição lógica do circuito; uma mesma tarefa pode ser igualmente executada (isto é, produzir o mesmo resultado) por circuitos diferentes, que tentem minimizar a ocupação de espaço em chip, o uso de energia ou o tempo de execução, por exemplo. A arquitetura define o conjunto de instruções que o processador deve ser capaz de executar, bem como algumas regras relacionadas ao uso das instruções.

MIPS é uma família de arquiteturas de conjunto de instruções (*instruction set architecture*, ISA) para computadores de conjunto de instruções reduzido (*reduced instruction set computer*, RISC) que começou a ser desenvolvida na década de 1980 por John Hennessy e seus estudantes, resultando, em 1984, na criação da empresa MIPS Computer Systems. A empresa foi adquirida pela Silicon Graphics e em 1998 deu origem à MIPS Technologies, focada em propriedade intelectual para dispositivos embarcados [19]. A arquitetura passou por várias revisões, culminando nas atuais MIPS32 e MIPS64 (para implementações de 32 e 64 bits, respectivamente) [11, 15].

Um arranjo de portas programável em campo (*field-programmable gate array*, FPGA) é um circuito integrado que pode ser reconfigurado por um consumidor final ou desenvolvedor após sua fabricação — por isso, programável em campo. Um FPGA é composto de vários blocos lógicos programáveis relativamente simples que, em conjunto, resultam em circuitos capazes de realizar funções complexas, como processamento digital de sinais. O circuito desejado é desenvolvido utilizando uma linguagem de descrição de hardware (*hardware description language*, HDL), compilado e carregado em um FPGA; este processo de implementação é denominado síntese. Verilog e VHDL são as duas linguagens de descrição de hardware mais utilizadas em desenvolvimento de circuitos eletrônicos, sendo ambas capazes de descrever circuitos complexos [6]. A natureza reprogramável de um FPGA permite desenvolver e testar circuitos digitais a um custo inferior em comparação

a circuitos integrados de aplicação específica (*application-specific integrated circuit*, ASIC), mais adequados à produção em larga escala. Circuitos FPGA para desenvolvimento geralmente são disponibilizados em placas contendo componentes auxiliares, como memórias, codificadores, decodificadores, botões, interruptores, *displays* de LCD e/ou 7 segmentos e interfaces de entrada e saída como VGA e PS/2, facilitando a etapa de desenvolvimento.

A criptografia, de forma simplificada, abrange o estudo e a prática de técnicas para que duas entidades possam se comunicar, através de um canal inseguro, de forma que uma terceira entidade não possa saber o que está sendo comunicado [18]. Para cumprir este objetivo, a mensagem deve ser cifrada antes do envio e decifrada após o recebimento, utilizando uma chave que apenas o remetente e o destinatário conheçam. Este modelo de comunicação, onde a mensagem é cifrada e decifrada com a mesma chave, informada previamente através de um canal seguro, é conhecido como criptografia simétrica.

A cifra Rijndael foi escolhida pelo NIST (*National Institute of Standards and Technology*), em um processo aberto, para se tornar o novo padrão de criptografia do governo norte-americano, nomeado AES (*Advanced Encryption Standard*), e substituir o antigo padrão DES (*Data Encryption Standard* [18]). A cifra opera com blocos de 128 bits e chaves de 128, 192 ou 256 bits.

1.1 Motivação

A arquitetura MIPS possui significativa importância comercial e histórica, sendo objeto de estudo em cursos de graduação na área de computação [16]. Uma implementação em Verilog com síntese em FPGA permite uma abordagem prática do assunto, servindo de referência para estudos e outras implementações.

O criptossistema AES é peça fundamental da segurança de dados atualmente, sendo utilizado em uma variedade de aplicações e tendo, inclusive, instruções dedicadas em processadores recentes [10]. A cifra é exemplo de um algoritmo de uso real e complexidade razoável que permite demonstrar o correto funcionamento do processador.

Um processador sintetizável em FPGA pode ser adaptado para a execução de tarefas específicas, como a cifra AES-128, abrindo espaço para o estudo de circuitos reconfiguráveis como solução para melhorar, em determinados contextos, o desempenho de processadores de uso geral.

1.2 Objetivos

O objetivo geral deste trabalho é implementar um processador *pipeline* de 32 bits em Verilog, baseado no processador desenvolvido para a disciplina de Organização e Arquitetura de Computadores, sintetizá-lo em um FPGA, compilar uma implementação do algoritmo AES-128 e executar o programa resultante no processador sintetizado. Os objetivos específicos são:

- Corrigir erros presentes na implementação do processador, otimizar o código existente e acrescentar novas funcionalidades;
- Descrever os principais módulos, as modificações e adições feitas ao processador;

- Descrever o processo de compilação, montagem e execução em FPGA de uma implementação em C da cifra AES-128;
- Introduzir uma solução em hardware para melhorar o desempenho da cifra AES-128 nesta implementação;
- Avaliar o desempenho da execução da cifra AES-128 no processador implementado.

É importante ressaltar que, apesar de ter a arquitetura MIPS como referência, este projeto não é afiliado, de nenhuma forma, à Imagination Technologies, tendo fins meramente acadêmicos.

1.3 Estrutura de documento

No Capítulo 2, apresentamos uma revisão teórica de conceitos básicos de processadores, focando em um processador *pipeline* de cinco estágios; também apresentamos conceitos de criptografia, o funcionamento da cifra AES de 128 bits e seus fundamementos algébricos. No Capítulo 3, descrevemos as modificações feitas no processador, detalhando o funcionamento dos novos módulos implementados, e o processo para compilar, montar e executar uma implementação da cifra AES-128 no processador, ressaltando os ajustes necessários. No Capítulo 4, avaliamos os resultados das modificações feitas no processador e da execução da cifra AES-128. No Capítulo 5, apresentamos as conclusões do projeto e sugerimos oportunidades para possíveis trabalhos futuros.

Capítulo 2

Revisão teórica

Patterson e Hennessy citam cinco componentes básicos de um computador: entrada, saída, memória, caminho de dados e controle; os dois últimos, às vezes, são combinados e chamados de processador [16]. A entrada e a saída são os meios de comunicação de um computador com o mundo exterior; como exemplo, podemos citar mouse e teclado como dispositivos de entrada e monitor e alto-falantes como dispositivos de saída. A memória é responsável por armazenar os dados, seja de forma temporária, como um módulo RAM (*random access memory*), ou a longo prazo, como um HDD (*hard disk drive*). Por último, temos o processador, incumbido de manipular os dados para produzir o resultado desejado.

2.1 Tipos de processadores

Com uma arquitetura de conjunto de instruções definida, é possível desenvolver um processador de diferentes formas. Três tipos básicos de processadores são apresentados: de ciclo único, multiciclo e *pipeline*.

2.1.1 Processador de ciclo único e multiciclo

Uma abordagem simples ao implementar um processador é dedicar um ciclo de clock para a execução de cada instrução. Para que todas as instruções sejam executadas corretamente, o período de clock é determinado pela instrução mais lenta, geralmente a de leitura da memória. A desvantagem desta implementação logo se torna aparente: uma instrução que é executada rapidamente passará o restante do período de clock sem fazer nada, diminuindo a eficiência do processador.

Para evitar que instruções rápidas fiquem esperando o fim do ciclo de clock, podemos dividir a execução de cada instrução em etapas menores, de acordo com as partes do caminho de dados utilizadas. Patterson e Hennessy apresentam uma divisão em cinco etapas:

1. busca da instrução;
2. decodificação da instrução e busca dos registradores;
3. execução, cálculo do endereço ou conclusão do desvio;
4. acesso à memória ou conclusão de instrução do tipo R;

5. conclusão da leitura de memória.

Assim, tipos diferentes de instruções necessitam uma quantidade diferente de ciclos para serem executadas: uma instrução do tipo R requer 4 etapas; uma instrução de acesso à memória, 5 etapas; uma instrução de desvio, 3 etapas. Cada etapa requer um ciclo de clock, mas como elas são apenas uma parte do caminho de dados, o período de clock necessário é menor em relação ao processador de ciclo único. Para controlar as etapas a serem executadas, é necessário hardware adicional; uma forma de realizar este controle é através de uma máquina de estados.

Esta implementação multiciclo faz uma melhor divisão do tempo de acordo com a necessidade de cada instrução, evitando que o processador fique ocioso enquanto aguarda o fim do ciclo de clock. Entretanto, apenas uma etapa é executada por vez, o que ainda indica uma certa ineficiência nesta abordagem.

2.1.2 Processador *pipeline* de cinco estágios

Pipeline é uma técnica de paralelismo que divide instruções em etapas menores, executando diferentes etapas de diferentes instruções simultaneamente. Desta forma, cada etapa pode ser realizada em um intervalo de tempo menor, assim como em uma implementação multiciclo, além de aumentar a vazão de instruções executadas, isto é, a quantidade de instruções executadas em um determinado intervalo de tempo.

Esta técnica, no entanto, apresenta algumas limitações. A latência de cada instrução, isto é, o tempo total necessário para executá-la, não diminui; pelo contrário, a latência das instruções pode aumentar por causa da separação em diferentes estágios. Além disso, a execução de diferentes instruções simultaneamente pode levar a situações em que um operando necessário para uma instrução não tenha sido calculado ainda por uma instrução anterior. Estes casos devem ser tratados adequadamente para que os benefícios de uma implementação *pipeline* seja vantajosa.

O número de etapas nas quais um processador *pipeline* é dividido depende da implementação e da arquitetura. O processador ARM7TDMI-S, por exemplo, implementa a arquitetura ARM7 e possui um pipeline de 3 estágios [5]. O processador Cortex-A8 possui um pipeline de 13 estágios. Arquiteturas RISC clássicas, como a MIPS, dividem o *pipeline* em cinco estágios [12]. Mais estágios são comuns em arquiteturas CISC (*complex instruction set computer*).

Um processador pode ser dividido em cinco estágios de *pipeline*:

1. **IF** (*instruction fetch*): a instrução é lida da memória de instruções a partir da posição definida pelo registrador PC (*program counter*).
2. **ID** (*instruction decode*): a instrução define os sinais da unidade de controle e seleciona os operandos do banco de registradores.
3. **EX** (*execute*): com os sinais e os operandos definidos, a unidade lógica e aritmetica realiza o cálculo necessário à instrução.
4. **MEM** (*memory*): estágio de acesso à memória de dados para leitura ou escrita, se necessário.

5. **WB** (*write back*): estágio de escrita do resultado da instrução no banco de registradores. Se a leitura no banco de registradores é feita na borda positiva do clock, a escrita é feita na borda negativa para que ambas ocorram em um único ciclo de clock.

Entre dois estágios há um conjunto de registradores para armazenar os dados que serão utilizados pelo estágio seguinte. Cada conjunto é identificado pelos estágios que os limitam. O conjunto de registradores entre os dois primeiros estágios, por exemplo, é identificado como IF/ID.

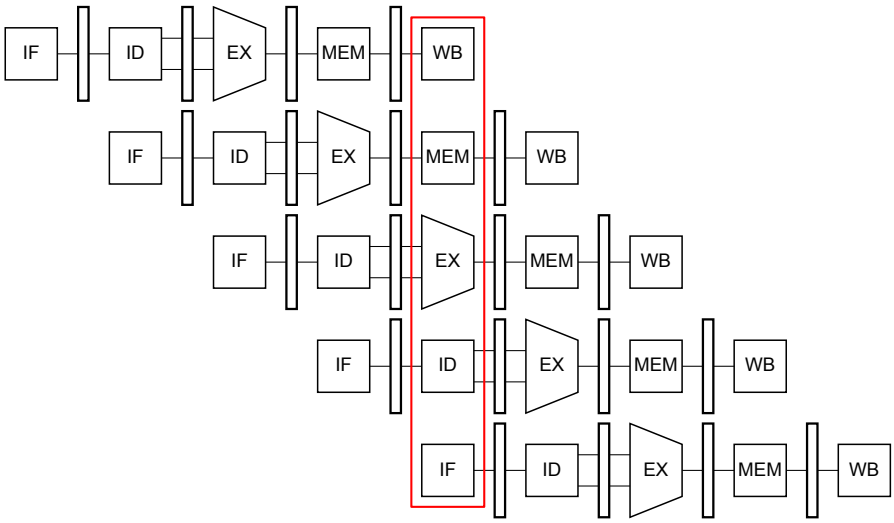

Figura 2.1: Representação de instruções em um processador *pipeline* [16]

A figura 2.1 apresenta uma forma de visualizar a execução de instruções em um processador *pipeline*. Na figura, cada linha representa os estágios do *pipeline* para uma dada instrução. A região em destaque corresponde a um instante da execução no qual a primeira instrução, no topo, está no estágio WB; a segunda instrução está no estágio MEM, e assim por diante.

2.1.3 Cuidados com *pipeline*

A técnica de *pipeline* introduz alguns problemas referentes a duas instruções serem executadas simultaneamente, sendo que uma delas afeta a outra. A seguir, apresentamos os diferentes tipos de problemas que podem ocorrer.

Risco de dados

Como o processador começa a executar uma instrução antes de a execução das quatro instruções anteriores terminar, há a possibilidade de a instrução utilizar um dado que não terminou de ser computado, resultando em uma execução incorreta do código. Este problema é denominado risco de dados (*data hazard*) e pode ser resolvido através de diferentes métodos, dependendo de como ocorrem.

Quando um dado já foi devidamente computado mas não foi escrito no banco de registradores, isto é, a instrução correspondente não chegou ao estágio WB, é possível

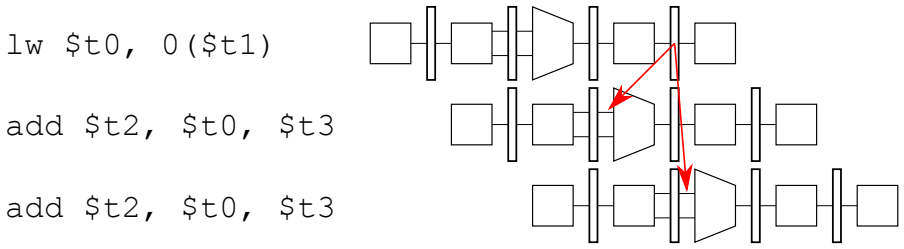

Figura 2.2: *Load delay slot* [16]

fazer um encaminhamento (*forwarding*) deste dado para o estágio de outra instrução que o necessite. Desta forma, a execução correta das instruções é garantida. O encaminhamento é feito dos registradores EX/MEM ou MEM/WB para as entradas da ULA e controlado por uma unidade de encaminhamento (*forward unit*).

Em alguns casos, o encaminhamento não evita o risco de dados. É necessário, portanto, esperar uma instrução avançar no *pipeline* antes de continuar a próxima instrução. Para isso, os estágios anteriores são pausados, desabilitando a escrita dos registradores entre os estágios.

Uma instrução `load` só obtém os dados da memória após o estágio MEM. Se a instrução seguinte precisar destes dados, será necessário esperar um ciclo de clock, isto é, ocasionar um *stall* para que os dados estejam disponíveis. A posição localizada após uma instrução `load` é chamada de *load delay slot*. Nesta posição, é preferível ter uma instrução que não dependa da instrução `load` anterior. Um montador deve levar este problema em consideração para tornar o código mais eficiente.

A Figura 2.2 mostra um exemplo do problema. Se a instrução `add` vier imediatamente após a instrução `lw`, os dados não podem ser encaminhados porque ainda não foram lidos da memória. Com um intervalo entre as duas instruções, os dados podem ser encaminhados do banco de registradores MEM/WB para uma das entradas da ALU.

Risco de controle

Riscos de controle ou riscos de desvio ocorrem quando um desvio é tomado: as instruções das posições seguintes ao desvio já estão no *pipeline* e só devem ser executadas se o desvio não for tomado. Portanto, se o desvio for tomado, é necessário descartar (*flush*) estas instruções. Este descarte afeta o desempenho do processador, que, por alguns ciclos de clock, não executará nenhuma instrução. Para compensar este problema, algumas técnicas podem ser adotadas.

Para minimizar a quantidade de instruções descartadas, a decisão de desvio é feita no segundo estágio do *pipeline*, já que o cálculo para decisão de desvios geralmente é simples. Para uma instrução `bne`, por exemplo, uma operação XOR entre os dois registradores seguida de uma operação OR entre todos os *bits* do resultado é o suficiente para decidir se o desvio será tomado ou não. No entanto, esta medida requer *hardware* adicional no segundo estágio e cria novos riscos de dados, pois os dados que poderiam ser encaminhados para o terceiro estágio deverão ser encaminhados para o segundo estágio.

Quando uma instrução de desvio atinge o estágio de execução, a instrução seguinte já começou a ser executada. Ao invés de descartar esta instrução, a arquitetura MIPS define que a instrução seguinte a um desvio deve sempre ser executada, seja o desvio tomado ou

não. Esta posição no código é chamada de *branch delay slot*. O montador deve reordenar as instruções na tentativa de ocupar o *branch delay slot* com uma instrução prévia, sem alterar o funcionamento do programa. Caso isto não seja possível, ele deve inserir uma instrução `nop`.

2.1.4 Coprocessadores

A arquitetura MIPS prevê a utilização de até quatro unidades auxiliares, chamadas de coprocessadores, para realizar funções de controle e estender a arquitetura. Cada coprocessador é identificado por um número de 0 a 3. Os coprocessadores 0 e 1 têm uso definido e os coprocessadores 2 e 3 podem ser utilizados para extensões da arquitetura para aplicações específicas.

Coprocessador 0: tratamento de exceções e interrupções

O coprocessador 0 é responsável pelo controle do processador, incluindo tarefas como:

- **Configuração da CPU**: o hardware de um processador MIPS geralmente é bem flexível, sendo possível definir algumas características da CPU, como a ordem dos *bytes*. Um ou mais registradores internos fornecem controle e visibilidade destas opções.
- **Controle de exceções e interrupções**: algumas instruções especiais e os registradores do coprocessador 0 definem o que acontece após interrupções e exceções e como elas serão tratadas.
- **Controle da unidade de gerenciamento de memória**: controla o endereçamento virtual, *translation look-aside buffer*, entre outros.
- **Outras tarefas**: outras funcinalidades, como temporizadores e detecção de erro de paridade também são tratados pelo coprocessador 0.

Coprocessador 1: unidade de ponto flutuante

Na arquitetura MIPS, operações em ponto flutuante são realizadas por uma unidade à parte, o coprocessador 1. A arquitetura segue o padrão IEEE 754 para representação de números em ponto flutuante. Uma unidade de ponto flutuante pode ser implementada de diferentes maneiras, com *tradeoffs* entre latência, vazão e área em *chip* [13].

O padrão IEEE 754 foi criado para corrigir os diversos problemas e divergências entre diferentes representações de números racionais. O padrão define exatamente qual resultado será obtido por um pequeno conjunto de operações básicas, mesmo em casos bem específicos, garantindo que programadores obtenham resultados idênticos a partir de entradas idênticas, independentemente da máquina que estejam usando. Esta abordagem requer a maior precisão possível para cada formato de dado.

Algumas das definições do padrão IEEE 754 são:

- **Arredondamento e precisão dos resultados**: mesmo os resultados mais simples podem não ser representáveis em uma fração finita. O padrão IEEE 754 permite ao usuário escolher entre quatro opções: arredondamento para cima, arredondamento para baixo, arredondamento para zero, arredondamento para o mais próximo.

- **Resultados excepcionais**: um cálculo pode produzir os seguintes resultados:
  - Inválido: raiz quadrada de -1;
  - Infinito: divisão explícita ou implícita por zero;
  - *Overflow*: resultado grande demais para ser representado;
  - *Underflow*: resultado pequeno demais para ser representado;
  - Impreciso: não pode ser perfeitamente representado e deve ser arredondado.
- **O que fazer em caso de exceção**: para cada caso de exceção, o usuário pode interromper a computação e sinalizar isto ao programa, dependendo do SO e da linguagem de programação, ou definir qual valor deve ser produzido. *Overflows* e divisão por zero geram infinito com distinção de sinal, operações invalidas geram NaN (*not a number*), etc.

2.2 O algoritmo AES-128

Antes de descrever o algoritmo em si, esta seção abordará alguns tópicos relacionados ao AES para fornecer uma melhor visão sobre o criptossistema e o contexto que o cerca.

2.2.1 Histórico

O padrão DES (*Data Encryption Standard*), adotado em 1977, não resistiu ao tempo: em 1998, o grupo Electronic Frontier Foundation construiu um computador, nomeado DES Cracker, que conseguiu encontrar uma chave da cifra DES em 56 horas; em 1999, junto com a rede de computadores distributed.net, conseguiram encontrar uma chave em 22 horas e 15 minutos. Uma das principais críticas ao DES é o curto espaço de chaves, de apenas 2^{56} bits. Além disso, ataques de criptoanálise linear e diferencial, ainda que inviáveis na prática, motivaram a busca por criptossistemas mais seguros [18].

Em 1997, o Instituto Nacional de Padrões e Tecnologia (*National Institute of Standards and Technology*, NIST) iniciou um processo de seleção para definir o próximo padrão de criptografia, a ser chamado de *Advanced Encryption Standard* (Padrão de Cifração Avançada). Em 1998, foram enviados 21 criptossistemas, dos quais 15 foram aceitos como candidatos. Em 1999, 5 dos candidatos foram escolhidos como finalistas: MARS, RC6, Rijndael, Serpent e Twofish. Finalmente, em 2000, o criptossistema Rijndael foi escolhido como o vencedor, sendo adotado como padrão no ano seguinte [7].

2.2.2 Princípios de criptografia

Primeiramente, apresentamos uma definição de criptossistema, relacionando texto claro (*plaintext*), texto cifrado (*ciphertext*) e chave (*key*).

Definição. *Um criptossistema é uma tupla* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$*, onde as seguintes condições são satisfeitas:*

1. $\mathcal{P}$ *é um conjunto finito de textos claros;*
2. $\mathcal{C}$ *é um conjunto finito de textos cifrados;*

3. *$\mathcal{K}$, o espaço de chaves, é um conjunto finito de possíveis chaves;*
4. *Para cada $K \in \mathcal{K}$, existe uma regra de cifração $e_K \in \mathcal{E}$ e uma regra correspondente de decifração $d_K \in \mathcal{D}$. $e_K : \mathcal{P} \to \mathcal{C}$ e $d_K : \mathcal{C} \to \mathcal{P}$ são funções tais que $d_K(e_K(x)) = x$ para todo texto claro $x \in \mathcal{P}$.*

Um princípio importante para a criptografia, denominado princípio de Kerckhoffs, dita que um criptossistema deve ser seguro mesmo que todos os detalhes do criptossistema, exceto a chave, sejam de conhecimento público [18]. Em outras palavras, o segredo de um criptossistema deve estar somente na chave, e não no sistema.

A criptografia pode ser dividida em duas áreas: simétrica e assimétrica. Na criptografia simétrica, a chave utilizada para cifrar um texto claro é a mesma utilizada para decifrá-lo. Na criptografia assimétrica, também chamada de criptografia de chave pública, são utilizadas duas chaves, uma pública e uma privada. Como o nome sugere, a chave pública fica disponível abertamente, e pode ser utilizada para cifrar dados. No entanto, apenas a chave privada pode decifrar os dados cifrados com a chave pública correspondente. Este mecanismo permite a comunicação segura entre duas pessoas através de um meio de comunicação inseguro, isto é, com possíveis observadores.

2.2.3 Fundamentos algébricos

O criptossistema AES foi construído com base em conceitos de corpos finitos. A compreensão de tais conceitos não é necessária para implementar o algoritmo, mas é importante para entender as decisões por trás do projeto da cifra.

Grupo abeliano

Começamos com a definição de um grupo abeliano para, em seguida, definir novos conceitos baseados neste.

Definição 1. *Um grupo abeliano $(G, +)$ consiste em um conjunto G e uma operação, simbolizada por $+$, definida em seus elementos:*

$$+ : G \times G \to G : (a, b) \mapsto a + b$$

Para ser considerado um grupo abeliano, a operação $+$ deve satisfazer as seguintes propriedades:

1. *Fechamento:* $\forall\ a, b \in G : a + b \in G$
2. *Associatividade:* $\forall\ a, b, c \in G : (a + b) + c = a + (b + c)$
3. *Comutatividade:* $\forall\ a, b \in G : a + b = b + a$
4. *Elemento neutro:* $\exists\ 0 \in G, \forall\ a \in G : a + 0 = a$
5. *Elemento inverso:* $\forall\ a \in G, \exists\ b \in G : a + b = 0$

O exemplo mais comum de um grupo abeliano é $(\mathbb{Z}, +)$, o conjunto dos inteiros com a operação de adição. Quaisquer dois números inteiros adicionados sempre resultarão em um número inteiro.

Anel

Um anel é um grupo abeliano com mais uma operação e que satisfaz mais duas propriedades.

Definição 2. *Um anel $(R, +, \cdot)$ consiste em um conjunto R e duas operações, aqui simbolizadas por $+$ e $\cdot$. Para ser considerado um anel, as operações $+$ e $\cdot$ devem satisfazer as seguintes propriedades:*

1. *A estrutura $(R, +)$ é um grupo abeliano.*
2. *A operação $\cdot$ possui fechamento e associatividade em R. Há um elemento neutro para $\cdot$ em R.*
3. *As operações $+$ e $\cdot$ estão relacionadas pela distributividade:*

$$\forall\ a, b, c \in R : (a + b) \cdot c = (a \cdot c) + (b \cdot c)$$

Se a operação $\cdot$ é comutativa, o anel $(R, +, \cdot)$ é chamado de anel comutativo. O conjunto dos inteiros com as operações de adição e multiplicação, $(\mathbb{Z}, +, \cdot)$, é um exemplo de anel comutativo.

Corpo

Um corpo é um anel que satisfaz as propriedades comutativas e de elemento inverso.

Definição 3. *A estrutura $(F, +, \cdot)$ é um corpo se as seguintes propriedades são satisfeitas:*

1. *$(F, +, \cdot)$ é um anel comutativo.*
2. *Para todo elemento de F, há um elemento inverso em F em relação à operação $\cdot$, exceto para 0, o elemento neutro de $(F, +, \cdot)$.*

Um corpo finito ou corpo de Galois[1] é um corpo com um número finito de elementos. A quantidade de elementos no conjunto é chamada de ordem do corpo. Um corpo com ordem m existe se e somente se m é uma potência de um número primo, isto é, $m = p^n$ para algum inteiro n e p primo. p é chamado de característico do corpo finito.

Corpos da mesma ordem são isomórficos, isto é, apresentam a mesma estrutura algébrica, diferindo apenas na representação dos elementos. Ou seja, para cada potência de um número primo, há um único corpo finito, denotado por $\mathrm{GF}(p^n)$.

Um polinômio sobre um corpo F é uma expressão na forma $b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + \ldots + b_2x^2 + b_1x + b_0$, onde x é o indeterminado do polinômio e $b_i \in F$ são os coeficientes.

O grau de um polinômio é igual a d se $b_j = 0, \forall j > d$, e d é o menor número com esta propriedade. O conjunto de polinômios sobre um corpo F é denotado por $F[x]$. O conjunto de polinômios sobre um corpo F que têm grau menor que d é denotado por $F[x]|_d$.

O criptossistema AES utiliza o corpo $\mathrm{GF}(2^8)$, associando um byte $b_7b_6b_5b_4b_3b_2b_1b_0$ ao polinômio $b(x) = b_7x^7 + b_6x^6 + \ldots + b_1x^1 + b_0x^0$.

Exemplo 1. O polinômio $x^7 + x^4 + x^3 + x + 1$ corresponde à cadeia de bits `10011011` ou, em hexadecimal, `9C`.

[1] Em inglês, *Galois field*, por isso a notação GF.

Adição em GF(2^8)

A soma de polinômios consiste em somar os coeficientes com o mesmo grau de x, e a soma dos coeficientes ocorre no corpo F:

$$c(x) = a(x) + b(x) \Leftrightarrow c_i = a_i + b_i, 0 \leq i < n. \tag{2.1}$$

O elemento neutro da adição é o polinômio com todos os coeficientes iguais a 0. O elemento inverso de um polinômio pode ser encontrado trocando cada coeficiente pelo seu inverso em F. O grau de $c(x)$ é no máximo o maior grau entre $a(x)$ e $b(x)$, portanto a adição é fechada. A adição em GF(2^8) é facilmente implementada através da operação XOR.

Exemplo 2. A soma entre os polinômios representados por `C9` e `1D` é o polinômio representado por `D4`, pois

$$\begin{aligned}&(x^7 + x^6 + x^3 + 1) \oplus (x^4 + x^3 + x^2 + 1) = \\ &x^7 + x^6 + x^4 + (1 \oplus 1)x^3 + x^2 + (1 \oplus 1) = \\ &x^7 + x^6 + x^4 + x^2.\end{aligned}$$

Multiplicação em GF(2^8)

A multiplicação de polinômios é associativa, comutativa e distributiva em relação à adição de polinômios. O polinômio de grau 0 e coeficiente $x^0 = 1$ é elemento neutro. Para que a multiplicação seja fechada em $F[x]|_d$, selecionamos um polinômio $m(x)$ de grau d, chamado de polinômio de redução. Assim, definimos a multiplicação entre dois polinômios $a(x)$ e $b(x)$ como o produto algébrico dos polinômios módulo $m(x)$:

$$c(x) = a(x) \cdot b(x) \Leftrightarrow c(x) \equiv a(x) \times b(x) \pmod{m(x)}. \tag{2.2}$$

Assim, a estrutura $< F[x]|_d, +, \cdot >$ é um anel comutativo.

Definição 4. *Um polinômio $d(x)$ é dito irredutível sobre o campo $GF(p)$ se e somente se não existem dois polinômios $a(x)$ e $b(x)$ com coeficientes em $GF(p)$ tais que $d(x) = a(x) \times b(x)$, tendo $a(x)$ e $b(x)$ graus maiores que 0.*

Seja $a(x)$ o polinômio do qual queremos encontrar o inverso. O algoritmo de Euclides estendido pode ser utilizado para encontrar dois polinômios $b(x)$ e $c(x)$ tais que

$$a(x) \times b(x) + m(x) \times c(x) = \gcd(a(x), m(x)), \tag{2.3}$$

onde $\gcd(a(x), m(x))$ denota o maior divisor comum de $a(x)$ e $m(x)$. Como $m(x)$ é irredutível, temos que $\gcd(a(x), m(x)) = 1$. Aplicando a redução modular, temos

$$a(x) \times b(x) \equiv 1 \mod m(x). \tag{2.4}$$

Assim, $b(x)$ é o inverso de $a(x)$ para a operação $\cdot$.

Exemplo 3. A multiplicação $(x^7 + x^5 + x^4 + x^2 + x) \cdot (x^6 + x^4 + x + 1)$ é igual a

Para facilitar a compreensão, vamos efetuar a multiplicação por cada termo do segundo polinômio separadamente:

$$
\begin{aligned}
(x^7 + x^5 + x^4 + x^2 + x) \times x^6 &= x^{13} + x^{11} + x^{10} + x^8 + x^7 \\
(x^7 + x^5 + x^4 + x^2 + x) \times x^4 &= x^{11} + x^9 + x^8 + x^6 + x^5 \\
(x^7 + x^5 + x^4 + x^2 + x) \times x &= x^8 + x^6 + x^5 + x^3 + x^2 \\
(x^7 + x^5 + x^4 + x^2 + x) \times 1 &= x^7 + x^5 + x^4 + x^2 + x
\end{aligned}
$$

Agora, realizamos a soma dos polinômios; como a operação é feita em GF(2), dois termos iguais se cancelam, como evidenciado na soma

$$
\begin{array}{llllllllll}
x^{13} + & \cancel{x^{11}} + & x^{10} + & & \cancel{x^8} + & \cancel{x^7} + & & & & \\
& \cancel{x^{11}} + & & x^9 + & \cancel{x^8} + & & \cancel{x^6} + & \cancel{x^5} + & & \\
& & & & x^8 + & & \cancel{x^6} + & \cancel{x^5} + & x^3 + & \cancel{x^2} + \\
& & & & & \cancel{x^7} + & & x^5 + x^4 + & & \cancel{x^2} + x = \\
\hline
x^{13} + & & x^{10} + x^9 + x^8 + & & & & & x^5 + x^4 + x^3 + & & x
\end{array}
$$

Assim, o polinômio resultante é $x^{13}+x^{10}+x^9+x^8+x^5+x^4+x^3+x$. Por fim, devemos realizar a redução módulo $x^8+x^4+x^3+x+1$. Uma maneira simples de fazer esta redução é utilizar o algoritmo de divisão longa. Ressaltando que $(x^8 + x^4 + x^3 + x + 1) \times x^5 = x^{13} + x^9 + x^8 + x^6 + x^5$ e $(x^8 + x^4 + x^3 + x + 1) \times x^2 = x^{10} + x^6 + x^5 + x^3 + x^2$, temos

$$
\begin{array}{llllllll}
x^{13} + x^{10} + x^9 + x^8 + & & x^5 + x^4 + x^3 + & & x+ \\
x^{13} + & x^9 + x^8 + x^6 + x^5 + & & & \\
& x^{10} + & x^6 + x^5 + & x^3 + x^2 + & \\
\hline
& & x^5 + x^4 + & x^2 + x &
\end{array}
$$

2.2.4 Visão geral do algoritmo

O criptossistema AES define um tamanho de bloco de 128 bits e três tamanhos de chave: 128, 192 e 256 bits. A cifra AES é iterada e o número de rodadas N_R depende do tamanho da chave: $N_R = 10$ para chaves de 128 bits; $N_R = 12$ para chaves de 192 bits e $N_R = 14$ para chaves de 256 bits.

A cifra pode ser descrita em alto nível através do pseudocódigo apresentado no algoritmo 1. x representa o bloco de entrada, em texto claro, e y representa o bloco de saída, cifrado. *State* é a matriz utilizada pelo algoritmo para representar o bloco sobre o qual as operações são efetuadas. O bloco é tratado como uma matriz no formato

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix},$$

onde B_0 representa o primeiro byte do bloco, B_2 representa o segundo byte e assim por diante.

Algoritmo 1 Cifrar o bloco x com AES-128

```
1: State ← x
2: AddRoundKey(0, State)
3: for i = 1 → N_R − 1 do
4:     SubBytes(State)
5:     ShiftRows(State)
6:     MixColumns(State)
7:     AddRoundKey(i, State)
8: end for
9: SubBytes(State)
10: ShiftRows(State)
11: AddRoundKey(N_R, State)
12: y ← State
```

2.2.5 Etapas

`AddRoundKey`

A etapa `AddRoundKey` realiza uma operação XOR entre a matriz *State* e a chave da rodada. São necessárias $N_R + 1$ chaves de rodada: a chave original, adicionada no início da cifra, e N_R chaves derivadas da original, adicionadas a cada rodada.

A expansão de chaves gera um total de $N_B(N_R + 1)$ words, onde N_B representa o número de words no bloco da cifra. A chave expandida é representada por um vetor de words $w[i]$, com $0 < i < N_B(N_R + 1)$. Assim, a chave da primeira rodada corresponde às words $w[0]$, $w[1]$, $w[2]$ e $w[3]$; a chave da segunda rodada corresponde às words $w[4]$, $w[5]$, $w[6]$ e $w[7]$, e assim por diante. O algoritmo 2 realiza a expansão da chave; N_K representa o número de words da chave, podendo ser 4, 6 ou 8.

Para $N_K = 4$, a primeira etapa do algoritmo é copiar a chave original para as posições $w[0]$ a $w[3]$. A última linha é copiada para um vetor *temp*. Se a posição do vetor atual for múltipla de N_K, são realizadas três operações: `RotWord`, que permuta os bytes do vetor da forma $\begin{bmatrix} B_0 & B_1 & B_2 & B_3 \end{bmatrix} \rightarrow \begin{bmatrix} B_3 & B_0 & B_1 & B_2 \end{bmatrix}$; `SubWord`, que substitui cada byte por outro de acordo com a tabela 2.1; por fim, uma operação XOR com uma word de um vetor de constantes *Rcon*, onde a posição $Rcon[i]$ contém os bytes $\begin{bmatrix} x^{i-1}, \texttt{00}, \texttt{00}, \texttt{00} \end{bmatrix}$, com $x = \texttt{02}$ e x^{i-1} denotando potenciação em GF(2^8).

Algoritmo 2 Expandir a chave $key[4N_K]$

```
1: i ← 0
2: while i < N_K do
3:     w[i] ← word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
4:     i ← i + 1
5: end while
6: i ← N_K
7: while i < N_B * (N_R + 1) do
8:     temp ← w[i − 1]
9:     if i mod N_K = 0 then
10:        temp ← SubWord(RotWord(temp)) xor Rcon[i/N_K]
11:    else if N_K > 6 ∧ i mod N_K = 4 then
12:        temp ← SubWord(temp)
13:    end if
14:    w[i] ← w[i − N_K] ⊕ temp
15:    i ← i + 1
16: end while
```

`SubBytes`

A etapa `SubBytes` realiza a substituição de cada byte da matriz S por outro byte, de acordo com a tabela 2.1. O byte `53`, por exemplo, é substituído pelo byte `ED`. A tabela de substituição é chamada de *S-box*.

Os valores da tabela não são aleatórios; eles correspondem a uma sequência de operações que podem ser definidas algebricamente. Primeiramente, o byte $a_7a_6a_5a_4a_3a_2a_1a_0$ é mapeado ao elemento $\sum_{i=0}^{7} a_i x^i$ do corpo finito $\mathrm{GF}(2^8) = \mathrm{GF}(2)[x]/(x^8+x^4+x^3+x+1)$. Pelo mesmo mapeamento anterior, associamos o inverso multiplicativo do elemento ao byte $b_7b_6b_5b_4b_3b_2b_1b_0$. Este byte é transformado através do mapeamento afim

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{2.5}$$

Esta transformação linear também pode ser representada pela expressão

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i, \tag{2.6}$$

onde b_i é o i-ésimo bit do byte e c_i é o i-ésimo bit do byte $\mathtt{63}_{(16)} = \mathtt{01100011}_{(2)}$ [17].

O objetivo destas operações é tornar a cifra resistente a ataques de criptoanálise linear e diferencial. Como as operações são fixas, normalmente utiliza-se uma tabela *lookup* com os valores pré-calculados para melhorar o desempenho da cifra. O processo de decifração consiste em realizar todas as operações na ordem inversa.

Tabela 2.1: Substituição de bytes da S-box. [7]

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

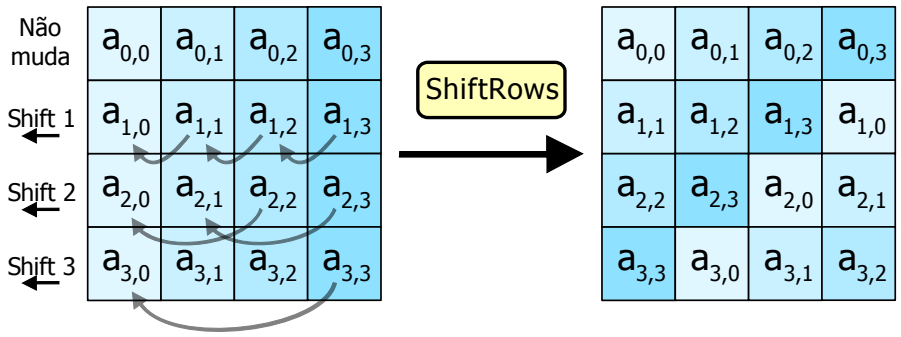

Figura 2.3: Demonstração da etapa `ShiftRows` [2]

`ShiftRows`

A etapa `ShiftRows` realiza uma simples permutação entre os bytes da matriz *State*. A primeira linha não é modificada; a segunda linha é deslocada 1 byte à esquerda; a terceira linha é deslocada 2 bytes à esquerda; a quarta linha é deslocada 3 bytes à esquerda. Os bytes que extrapolam o limite esquerdo da matriz são realocados à direita. A figura 2.3 mostra o deslocamento de cada linha e a matriz resultante.

`MixColumns`

A etapa `MixColumns` transforma, separadamente, cada uma das colunas da matriz *State*. As colunas são tratadas como polinômios em $\mathrm{GF}(2^8)$ e multiplicadas módulo x^4+1 por um polinômio fixo $a(x) = 3x^3 + 1x^2 + 1x + 2$. Esta operação pode ser representada

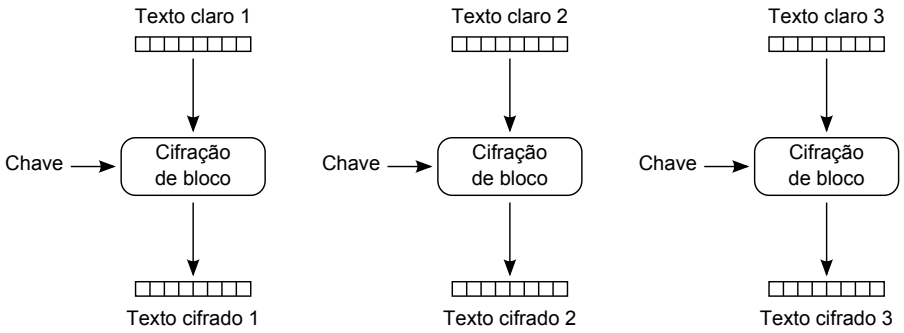

Figura 2.4: Modo de operação ECB [8]

como a multiplicação de matrizes

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \tag{2.7}$$

e, assim, os bytes da coluna podem ser definidos por

$$\begin{aligned} s'_0 &= (02 \cdot s_0) \oplus (03 \cdot s_1) \oplus s_2 \oplus s_3 \\ s'_1 &= s_0 \oplus (02 \cdot s_1) \oplus (03 \cdot s_2) \oplus s_3 \\ s'_2 &= s_0 \oplus s_1 \oplus (02 \cdot s_2) \oplus (03 \cdot s_3) \\ s'_3 &= (03 \cdot s_0) \oplus s_1 \oplus s_2 \oplus (02 \cdot s_3), \end{aligned}$$

onde $\cdot$ denota multiplicação em $GF(2^8)$.

2.2.6 Modos de operação

Cifras de bloco, como o AES, operam sobre um grupo de bits de tamanho fixo, chamado de bloco. Um modo de operação define como aplicar uma cifra repetidamente em uma quantidade de dados maior que o bloco. Cinco modos de operação comuns são ECB (*Electronic Codebook*), CBC (*Cipher Block Chaining*), CFB (*Cipher Feedback*), OFB (*Output Feedback*) e CTR (*Counter*) [8]. Naturalmente, a decifração ocorre pelo processo inverso. Cada modo de operação possui vantagens e desvantagens, e a escolha ideal depende da aplicação.

ECB (*Electronic Codebook*)

O modo de operação ECB é o mais trivial: a cifra é aplicada individualmente a cada bloco. A figura 2.4 ilustra o funcionamento deste modo de operação.

Como a aplicação em cada bloco não é afetada por outros blocos, este modo de operação pode ser facilmente paralelizado. No entanto, blocos idênticos resultarão no mesmo texto cifrado, preservando os padrões do texto claro. Por causa disso, o EBC não é recomendado para uso em protocolos criptográficos.

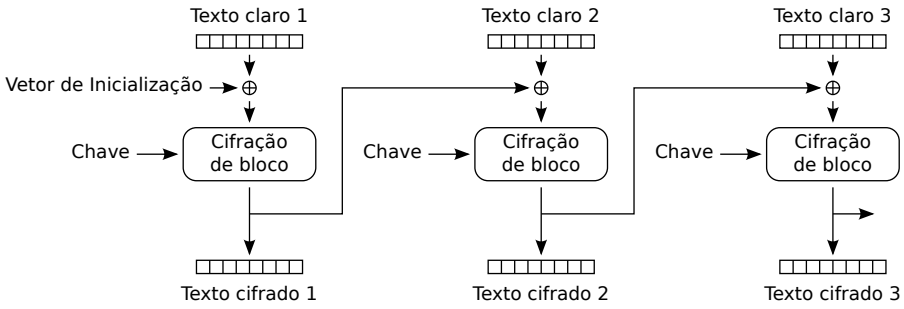

Figura 2.5: Modo de operação CBC [8]

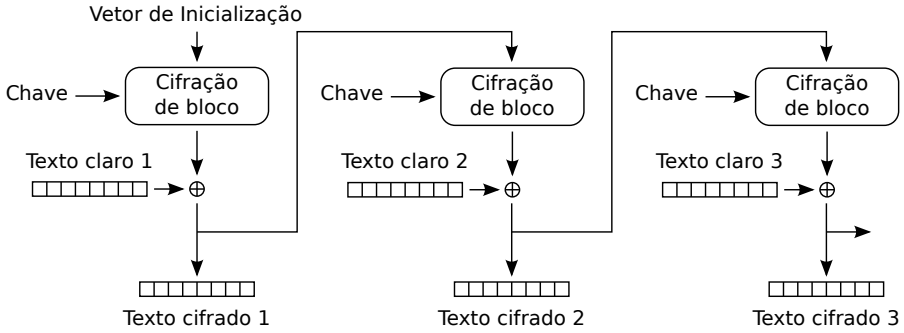

Figura 2.6: Modo de operação CFB [8]

CBC (*Cipher Block Chaining*)

Um procedimento simples para evitar o problema que ocorre no modo de operação ECB é realizar uma operação XOR entre o texto claro de um bloco e o texto cifrado de um bloco anterior, como mostra a figura 2.5. No caso do primeiro bloco, é utilizado um vetor de inicialização, isto é, uma sequência de bits aleatórios. Este vetor é necessário para decifrar o primeiro bloco, mas não para os demais; assim, para decifrar o texto por completo sem conhecer o vetor de inicialização, é possível inserir um bloco descartável antes do texto a ser cifrado.

A cifração de um bloco depende da cifração do bloco anterior, portanto este modo de operação não é paralelizável. Entretanto, a decifração é paralelizável, pois a operação XOR é realizada após a decifração de cada bloco.

CFB (*Cipher Feedback*)

Mudando algumas operações e operandos de lugar, temos o modo de operação CFB, como ilustra a figura 2.6. Uma das propriedades deste modo de operação é que se uma parte do texto cifrado for perdida (por falhas na transmissão, por exemplo), apenas uma parte do texto claro será perdida.

Repare que, para obter o texto claro 2, precisamos realizar uma operação XOR entre o texto cifrado 2 e a cifração do texto cifrado 1; ou seja, para decifrar a mensagem, não precisamos utilizar o algoritmo de decifração, apenas o de cifração.

Com a apresentação de conceitos básicos relacionados a um processador *pipeline* e à cifra AES-128, discutiremos o processo de implementação no capítulo seguinte.

Capítulo 3

Implementação

Neste capítulo, apresentamos os detalhes de implementação do processador, relatando mudanças e acréscimos feitos ao projeto, e o processo de compilação, montagem e execução do algoritmo AES-128 no processador desenvolvido.

3.1 Processador

O projeto utiliza o *software* Quartus II 13.0.1 Build 232 Web Edition e a placa de desenvolvimento Altera DE2-70, mostrado na figura 3.1, com um circuito FPGA Cyclone II, modelo EP2C70F896C6N.

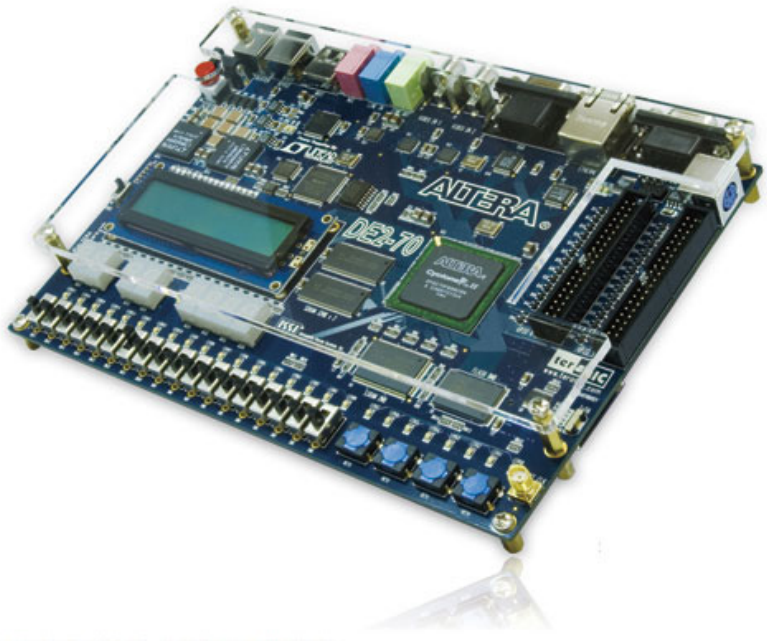

Figura 3.1: Placa de desenvolvimento Altera DE2-70 [1]

O subconjunto de instruções implementadas é apresentado na Tabela 3.1 e segue o padrão MIPS [15]. Instruções novas ou que foram modificadas de alguma forma em relação ao projeto inicial estão marcadas com uma estrela (⋆). Este subconjunto foi escolhido

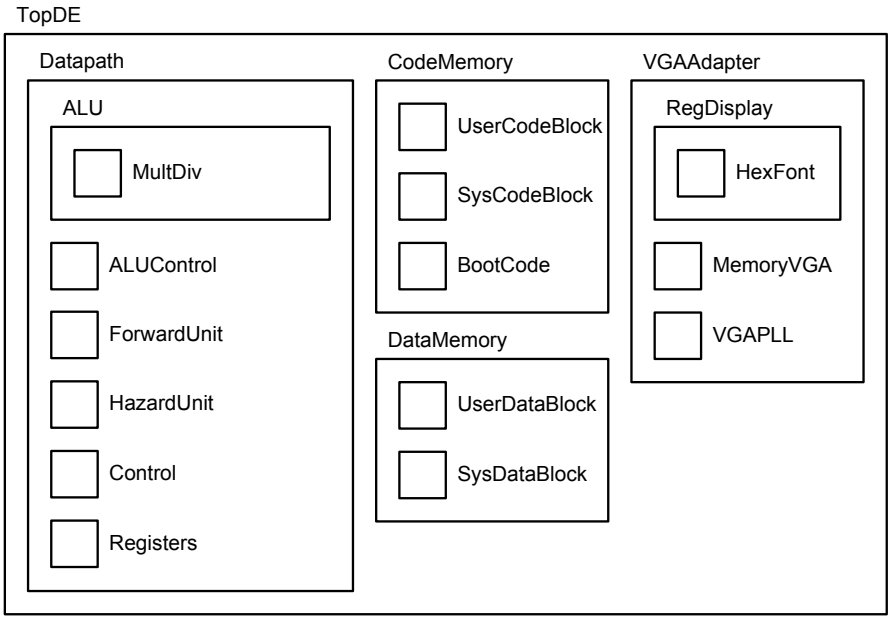

Figura 3.2: Hierarquia dos módulos do processador

buscando um equilíbrio entre a flexibilidade para o programador e a complexidade de implementação do processador.

A figura 3.2 apresenta a hierarquia dos principais módulos do processador. Módulos pequenos, como circuitos combinacionais para exibir nibbles em displays de 7 segmentos, e módulos não utilizados neste projeto, como interface de teclado e mouse, não estão representados na figura.

3.1.1 Descrição dos módulos

Nesta seção, apresentaremos uma breve descrição dos principais módulos e das modificações realizadas em relação à versão inicial do projeto.

O módulo `TopDE` é definido como entidade de topo e inclui os outros módulos do projeto, implementando a interface entre eles e as entradas e saídas da placa de desenvolvimento.

Datapath

O módulo `Datapath` implementa o processador propriamente dito, isto é, o caminho de dados, instanciando os módulos necessários e implementando registradores entre estágios e diversos circuitos auxiliares, como multiplexadores.

No estágio ID, os sinais `ExtSigImm` (extensão de sinal do imediato), `ExtZeroImm` (extensão com zero do imediato) e `ConcatZeroImm` (concatenação com zero do imediato) eram calculados no estágio ID e armazenados no registrador ID/EX para serem utilizados no estágio EX. Este cálculo foi adiado para o estágio EX sem afetar significativamente a temporização do estágio, pois operações de concatenação e extensão utilizam circuitos muito simples.

Alguns recursos redundantes e ineficientes estavam presentes no caminho de dados e foram modificados ou eliminados. Os registradores ID/EX foram reduzidos de 245 para 143; os registradores EX/MEM, de 122 para 110; os registradores MEM/WB, de 120 para 38. No total, 196 registradores foram eliminados.

Os módulos de memória eram instanciados dentro do caminho de dados. Para melhorar a organização do processador, eles foram movidos para o módulo `TopDE`.

Foi adicionado um parâmetro que permite definir, em tempo de compilação, se a instrução no *branch delay slot* será executada; desta forma, um código desenvolvido para uma implementação uniciclo ou multiciclo, nos quais o conceito de *branch delay slot* não existe, pode ser executado nesta implementação *pipeline* corretamente.

Control

O módulo de controle é responsável por definir, a partir da instrução atual, os sinais que controlarão os demais componentes do processador, tais como leitura/escrita em memória, registrador de destino e operação da ALU.

ALU

A unidade lógico-aritmética efetua cálculos de adição, subtração, igualdade, deslocamento, multiplicação e divisão de inteiros. O circuito da ALU é puramente combinacional.

A operação de deslocamento aritmético, isto é, com extensão de sinal, estava implementada incorretamente. Um pequeno módulo implementando corretamente a operação foi criado e testado.

Dentro da unidade lógico-aritmética foi implementado o módulo de multiplicação e divisão de inteiros. Como são operações um pouco mais complexas, uma implementação em circuito puramente combinacional resultaria em um período de clock muito grande. Para evitar isto, a implementação é feita em um circuito sequencial e o cálculo é concluído após 32 ciclos de clock (1 ciclo para cada bit dos operandos). Os algoritmos implementados neste módulo são apresentados por Patterson e Hennessy [16].

Hazard Unit

A unidade de risco detecta os riscos de dados e de controle e gera os sinais para bloquear ou realizar um *flush* em alguns estágios do processador.

Forward Unit

A unidade de encaminhamento verifica se há riscos de dados que podem ser resolvidos por encaminhamento e controla os multiplexadores que determinam as entradas da ALU. O circuito é combinacional.

Registers

Este módulo implementa o banco de registradores, composto de 32 registradores com 32 bits cada, e o circuito necessário para selecionar três registradores para leitura e um para escrita, sendo uma das leituras para visualização (ver seção 3.1.3). O registrador de número zero (`$zero`) não pode ser escrito, apenas lido.

Como pode ser visto na tabela 4.1, esta implementação requer muitos elementos lógicos; a escrita é realizada na borda de descida do clock e deve ser concluída até a próxima borda de subida. Como consequência, este módulo é um ponto crítico para a temporização do processador.

Outra forma de implementar o banco de registradores é utilizar um módulo de memória com duas portas de leitura e uma de escrita ou dois módulos de memória com uma porta de escrita e uma de leitura; esta abordagem é utilizada pelo processador Plasma.

Como a memória embarcada no chip FPGA (*on-chip memory*) já é utilizada como memória de dados e de código, não utilizamos esta solução; ademais, a visualização de registradores, como implementada agora, não seria possível.

3.1.2 Melhorias

Nesta seção, descrevemos as mudanças feitas em módulos e recursos já presentes na versão inicial do projeto.

Mapeamento da memória

A memória utilizada pelo processador foi remapeada para ficar mais próxima das configurações adotadas pela arquitetura MIPS [19]. O novo mapeamento é apresentado na tabela 3.2.

Linearização da memória de vídeo

Na versão inicial, os endereços da memória de vídeo seguiam o formato `0x00XXXYYY`: os 12 bits menos significativos correspondiam à coordenada y do pixel, e os 12 bits seguintes correspondiam à coordenada x. Como a memória de vídeo possui uma resolução de 320 por 240 pixels, os endereços cujas coordenadas estavam no intervalo $0 \leq x < 320$ e $240 \leq y < 4096$ eram ignorados.

O endereçamento original facilitava o cálculo do endereço de um pixel específico sem utilizar multiplicação. Em vez disso, bastava utilizar um deslocamento lógico à esquerda para posicionar os 12 bits da coordenada x e adicionar

Na versão atual, a memória está linearizada: o endereço de cada pixel segue a equação $A = B + (y \times W) + x$, onde A é o endereço, B é o endereço inicial da memória de vídeo e W é a resolução horizontal da saída de vídeo. Este endereçamento permite que a memória seja facilmente representada, em uma linguagem de programação, por uma matriz bidimensional.

Apesar de utilizar uma operação de multiplicação, é possível realizar este cálculo mais rapidamente. Como $W = 320$, temos que

$$\begin{aligned} y \times 320 &= y \times (256 + 64) \\ &= (y \times 256) + (y \times 64) \\ &= (y \times 2^8) + (y \times 2^6). \end{aligned}$$

Multiplicações por potências de 2 podem ser substituídas por deslocamento lógico à esquerda. Assim, esta multiplicação pode ser substituída, por exemplo, pelas instruções

```
sll $t0, $t0, 6
add $t1, $t1, $t0
sll $t0, $t0, 2
add $t1, $t1, $t0
```

onde `$t0` representa y e `$t1` representa o endereço base da memória de vídeo.

Outro problema no endereçamento da memória de vídeo na versão inicial é o endereçamento a byte: apenas as instruções `lw` e `sw` estavam implementadas, sendo utilizadas sem alinhamento, e escreviam e liam apenas um byte. Com a implementação das instruções `load` e `store` para halfwords e bytes, o acesso à memória de vídeo ocorre de forma mais coerente: uma instrução `sb` modifica um pixel, uma instrução `sw` modifica dois pixels e uma instrução `sw` modifica quatro pixels. A leitura segue a mesma lógica, respeitando a extensão de sinal de acordo com a instrução. Esta mudança permite desenhar pixels na tela muito mais rapidamente, especialmente para padrões repetitivos, como preencher a tela com uma única cor.

Profundidade de cores

Cada byte da memória de vídeo representa um pixel, no formato RGB332. Como o DAC da placa de desenvolvimento recebe 10 bits de entrada por canal, os bits precisam ser expandidos; esta expansão é feita através da repetição periódica dos bits. Nos canais R e G, os bits $b_2b_1b_0$ são expandidos para $b_2b_1b_0b_2b_1b_0b_2b_1b_0b_2$. No canal B, os bits b_1b_0 são expandidos para $b_1b_0b_1b_0b_1b_0b_1b_0b_1b_0$. Na versão inicial, a expansão era feita apenas por concatenação com zeros, resultando em cores mais escuras do que o esperado, principalmente para cores mais claras.

3.1.3 Novas funcionalidades

Visualizador de registradores

Para auxiliar na análise da execução de um programa, o processador implementado permite visualizar o conteúdo dos registradores de uso geral. Na versão inicial, era necessário selecionar manualmente o registrador através de 5 interruptores para que os dados no registrador fossem exibidos, em base hexadecimal, nos *displays* de 7 segmentos. Este método é limitante e ineficiente, pois permite visualizar apenas um registrador por vez e requer uma manipulação frequente dos interruptores, consumindo tempo e aumentando as chances de equívocos.

Com o objetivo de facilitar a visualização dos registradores, foi desenvolvido um novo módulo que exibe na saída de vídeo os dados de todos os 32 registradores de uso geral. Esta visualização pode ser ativada e desativada através de um único interruptor, mesmo com o processador em funcionamento. O mesmo seletor presente na versão inicial é utilizado, mas a seleção dos registradores é feita automaticamente através do novo módulo.

A interface do módulo é ilustrada na figura 3.3, sendo constituída dos seguintes sinais:

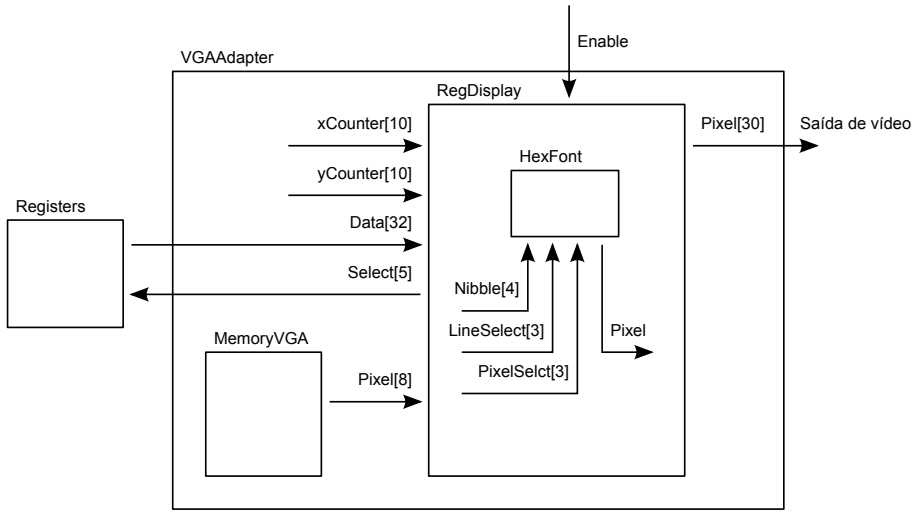

Figura 3.3: Interface do módulo `RegDisplay`

- `xCounter`: contador da posição horizontal do sinal de vídeo.
- `yCounter`: contador da posição vertical do sinal de vídeo.
- `iPixel`: valor do pixel na memória de vídeo, no formato RGB332.
- `Data`: valor do registrador selecionado por `Select`.
- `Enable`: ativa a visualização de registradores na saída de vídeo.
- `oPixel`: valor do pixel a ser exibido na saída de vídeo, já expandido para 10 bits por canal.
- `Select`: seleciona o registrador em função de `xCounter` e `yCounter`.

O módulo de vídeo implementado gera uma resolução de 640 por 480 pixels, mas a memória de vídeo representa uma resolução de 320 por 240 pixels. Esta diferença é resolvida ao ignorar o bit menos significativo dos contadores `xCounter` e `yCounter`, de forma que cada pixel e cada linha sejam lidos duas vezes da memória.

Representamos a tela dividida em blocos de 8×8 pixels, resultando em uma matriz de 40×30 blocos. Esta representação simplifica a lógica do hardware para exibir os dados dos registradores, pois basta ignorar três bits de cada contador para identificar o bloco da posição atual.

1. Regiões inalteradas: os pixels são exibidos sem modificações.
2. Regiões sombreadas: os pixels são escurecidos.
3. Regiões com caracteres: os caracteres são desenhados em branco com o fundo escurecido.

As coordenadas `xCounter` e `yCounter` do sinal de vídeo definem o registrador a ser selecionado e o nibble a ser exibido. O módulo `HexFont` recebe o nibble e os contadores como entrada; com estas informações, ele determina se o pixel corresponde ao caractere ou

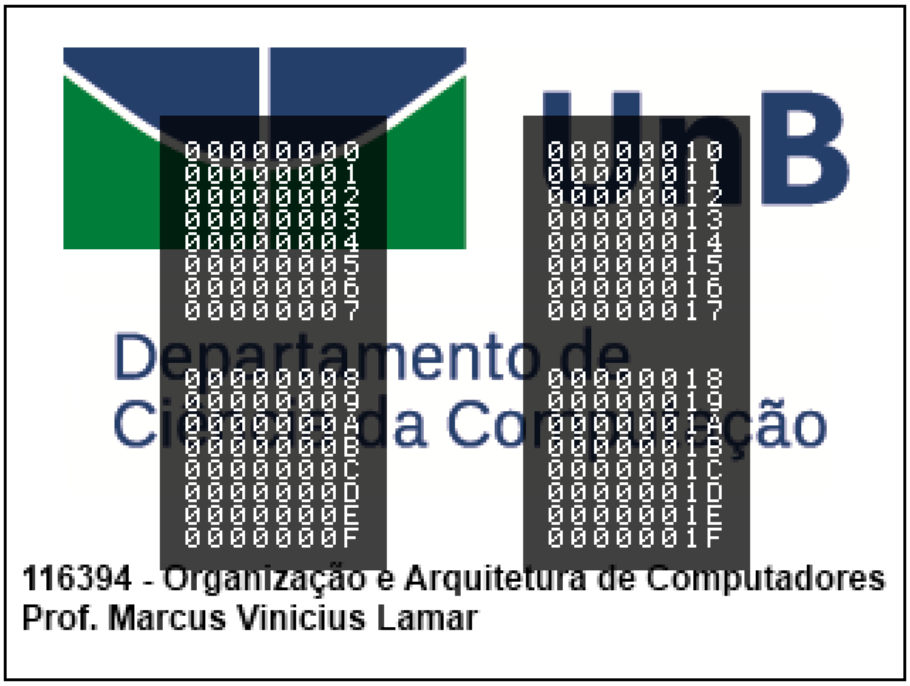

Figura 3.4: Saída de vídeo com o módulo `RegDisplay` ativado

ao fundo e envia um bit para o módulo `RegDisplay` substituir o pixel original por um pixel branco ou apenas escurecê-lo. Para escurecer os pixels, os dois bits mais significativos de cada canal são zerados.

O mapa de bits dos caracteres está dentro do módulo e ocupa um total de 1024 bits. Esta implementação focou na simplicidade do código para melhor entendimento; uma implementação que otimize o uso de recursos certamente é possível.

A figura 3.4 mostra o resultado final. Nela, cada registrador contém o valor correspondente ao número do registrador. `$t8`, por exemplo, contém o valor $18_{(16)} = 24_{10}$.

Módulo de multiplicação e divisão

Foi implementado um módulo de multiplicação e divisão de inteiros baseado nos algoritmos apresentados por Patterson e Hennessy [16]. Optando pela simplicidade, a implementação utiliza algoritmos iterativos e requer um ciclo de *clock* por *bit*, totalizando 32 ciclos por operação. O módulo utiliza o mecanismo de *interlock*: caso uma instrução que utilize o módulo seja executada antes que o cálculo anterior tenha sido finalizado, um sinal é ativado e pausa o resto do processador até que o cálculo termine. Neste cenário, é recomendado inserir, de forma similar ao *load delay slot*, instruções que não causem *interlock*, a fim de minimizar o tempo de pausa do processador.

Este módulo pode ser otimizado significativamente. No entanto, por não ser foco deste projeto, a implementação concentrou-se em disponibilizar as operações de multiplicação e divisão, sem se preocupar com o desempenho.

Módulo *gfmult*

Este módulo implementa a nova instrução `gfmult`, que realiza a operação de multiplicação módulo $x^8 + x^4 + x^3 + x + 1$ (representado em binário por `100011011`) entre dois polinômios, definida na seção 2.2.3. Esta instrução é utilizada na etapa `MixColumns`, descrita na seção 2.2.5. A instrução tem três operandos, mas apenas os 8 bits menos significativos de cada operando, representando polinômios em GF(2^8), são utilizados. Definimos, arbitrariamente, que `gfmult` é uma operação do tipo R com o campo opcode igual a `3F`.

O cálculo é dividido em duas etapas: multiplicação e redução. Na multiplicação, para cada bit b_n do multiplicador, com $0 \leq n \leq 7$, o multiplicando é expandido para 15 bits e deslocado n bits à esquerda, seguido de uma operação AND entre b_n e cada bit do multiplicando. Como resultado, temos 8 cadeias de 15 bits cada, que são adicionadas com a operação XOR. Para reduzir a latência do módulo, as operações XOR estão organizadas em uma estrutura de árvore binária.

O resultado das operações da primeira etapa possui 15 bits, correspondentes a um polinômio de grau até 14; denotaremos os bits do resultado como r_n, com $0 \leq n \leq 14$. A redução é realizada subtraindo, através da operação XOR, múltiplos do polinômio redutor (`100011011`) até que os 7 bits mais significativos sejam iguais a zero. Para o bit r_{14}, deslocamos o polinômio redutor à esquerda em 6 posições e realizamos uma operação AND entre r_{14} e cada bit do polinômio redutor; desta forma, a subtração só terá efeito se $r_{14} =$ `1`. Para os três bits seguintes, o processo segue o mesmo padrão, mudando apenas o bit r_n e o número de deslocamentos à esquerda.

Para os três bits restantes, é necessário um pouco mais de cuidado. O bit r_{10} pode ser anulado da mesma forma descrita anteriormente, mas, se $r_{14} =$ `1`, o bit já será anulado pelo bit 4 do polinômio redutor. Portanto, para o bit r_{10}, devemos subtrair um múltiplo do polinômio redutor apenas se um dos dois bits, r_{14} ou r_{10}, for igual a `1`. Seguindo o mesmo raciocínio, encontramos as condições para os bits r_9 e r_8.

3.2 AES-128

Procuramos por implementações da cifra AES-128 adequadas para o processador e optamos pelo Tiny AES128 [3], uma implementação em C, de código aberto e com foco em portabilidade e simplicidade.

Como a cifra AES manipula bytes diretamente, a ordem dos bytes (*endianness*) não interfere no algoritmo. No entanto, é importante ter em mente a ordem dos bytes de entrada e saída da cifra. O MARS utiliza a ordenação *little-endian*, portanto o texto claro `6B C1 BE E2 2E 40 9F 96 E9 3D 7E 11 73 93 17 2A` é representado na memória como as sequências de words `E2 BE C1 6B`, `96 9F 40 2E`, `11 7E 3D E9` e `2A 17 93 73`.

3.2.1 Compilação

Para compilar o Tiny AES128 tendo a arquitetura MIPS como alvo, utilizamos o compilador `mips-sde-elf-gcc` (versão 4.8.1), baseado no GCC e fornecido pela Mentor Graphics.

A compilação não apresenta dificuldades: o sinalizador `-S` instrui o compilador a compilar o código-fonte sem realizar a montagem; assim, o arquivo de saída estará em linguagem *assembly*. O comando `mips-sde-elf-gcc aes.c -S -o aes.s` produz o arquivo em *assembly* que será utilizado pelo MARS.

3.2.2 Ajustes

O código em *assembly* produzido pelo compilador não é diretamente compatível com o MARS, sendo necessário realizar alguns ajustes. Para automatizar e agilizar esta etapa, alguns dos ajustes necessários podem ser feitos através do uso de expressões regulares. Com o programa `rxrepl` [14] e um arquivo de lote (*batch*) com as substituições adequadas, o processo de compilação e ajuste para montagem é realizado em apenas um comando.

Instruções

O compilador produz instruções nos formatos `sltu $a, $b, c`, `slt $a, $b, c` e `j $a`, onde `$a` e `$b` são registradores de uso geral e `c` é um valor imediato. Pelos operandos, é possível inferir que as instruções produzidas são, na verdade, `sltiu`, `slti` e `jr`, respectivamente. O MARS não faz esta inferência, sendo necessário alterar o código.

A instrução `bgez` (*Branch on greater than or equal to zero*) não está implementada no processador. Em vez de implementar a instrução, podemos substituí-la pelo par de instruções `slt` e `beq`, respectivamente.

A instrução `seb` (*Sign-extend byte*) não está implementada e não é reconhecida pelo MARS, mas pode ser facilmente substituída pelo par de instruções `sll` e `sra`, respectivamente. Uma forma simples de fazer esta substituição é definir `seb` como uma macro para as duas instruções citadas.

Diretivas

O código produzido pelo compilador inclui diretivas que instruem o montador a montar o código da forma adequada [4]. Muitas das diretivas não são reconhecidas pelo MARS, mas podem ser descartadas sem prejuízo. Diretivas para fins de debug, por exemplo, não são relevantes para o processador, pois o código é executado no processador sem a supervisão de outro programa. A lista a seguir apresenta as diretivas presentes no código que não são reconhecidas pelo MARS:

- `.file`: informa o nome do arquivo que foi compilado, para fins de debug.
- `.section`: instrui o montador a montar o código em uma seção com o nome informado.
- `.previous`: troca a seção/subseção atual pela seção/subseção anterior.
- `.nan`: indica qual codificação de NaN (*not a number*) é usada nos arquivos em assembly, podendo ser a codificação original (`legacy`) ou a do padrão IEEE 754-2008 (`2008`).
- `.gnu_attibute`: define atributos GNU do objeto, como ABI (*application binary interface*) da FPU.

- `.local`: marca os símbolos listados como locais, de forma que não sejam externamente visíveis.
- `.comm`: declara um símbolo comum que pode ser fundido a outro símbolo comum com o mesmo nome em outro arquivo.
- `.rdata`: indica o início do segmento de dados apenas para leitura (*read-only data*).
- `.type`: define o tipo de um símbolo.
- `.size`: define o tamanho de um símbolo.
- `.ent`: marca o início de uma função.
- `.frame`: define o formato da pilha através dos argumentos: registrador do quadro (geralmente `$fp`), deslocamento, registrador de retorno (geralmente `$ra`).
- `.mask`: indica os registradores que devem ser salvos na pilha. Cada registrador é representado por um bit, utilizado como máscara pelo montador para salvar os registradores correspondentes.
- `.fmask`: assim como a diretiva `.mask`, indica os registradores de ponto flutuante que devem ser salvos na pilha.
- `.end`: marca o fim de uma função.
- `.ident`: insere *tags* em arquivos objeto.

As seguintes diretivas presentes no código são reconhecidas pelo MARS:

- `.align`: alinha o endereço a um múltiplo do valor especificado.
- `.byte`: insere um byte específico naquela posição.
- `.text`: instrui o montador a montar o código no segmento de texto.
- `.set`: ativa/desativa opções do montador, como a expansão de macros (`macro` e `nomacro`) e a reordenação de instruções (`reorder` e `noreorder`). Esta diretiva, apesar de reconhecida, é ignorada pelo MARS.
- `.globl`: torna um símbolo visível globalmente.
- `.data`: indica o início do segmento de dados.

O compilador insere valores constantes, como a tabela S-box, no segmento `.rdata`. Como esta diretiva não é reconhecida pelo MARS, trocamos a diretiva por `.data`.

Instrução `gfmult`

O MARS não oferece suporte a instruções customizadas. Para inserir a nova instrução `gfmult` no código, inserimos uma instrução `nop` no código em assembly e, após exportar para o formato MIF, identificamos a instrução `nop` inserida e a substituímos pelo código hexadecimal correspondente à instrução `gfmult` com os operandos certos.

3.2.3 Execução

Com o código montando e executando corretamente no MARS, utilizamos a ferramenta *MIF Exporter*, desenvolvida para a disciplina de Organização e Arquitetura de Computadores, para exportar o código no formato de inicialização de memória do Quartus II.

O código é carregado na memória do processador através da ferramenta *In-system Memory Content Editor* do programa Quartus II. Com o código escrito na memória, ativamos o clock do processador através dos botões na placa de desenvolvimento.

Com o processador e o algoritmo em funcionamento, podemos obter dados referentes à implementação e à execução. No próximo capítulo, faremos a apresentação e análise dos dados obtidos.

Tabela 3.1: Instruções implementadas

Mnemônico	Instrução	Tipo	Opcode	Funct
`add`	Add Word	R	`00`	`20`
`addi`	Add Immediate Word	I	`08`	—
`addiu`	Add Immediate Unsigned Word	I	`09`	—
`addu`	Add Unsigned Word	R	`00`	`21`
`and`	And	R	`00`	`24`
`andi`	And Immediate	I	`0C`	—
`beq`	Branch on Equal	I	`04`	—
`bne`	Branch on Not Equal	I	`05`	—
`div`	Divide Word⋆	R	`00`	`1A`
`divu`	Divide Word Unsigned⋆	R	`00`	`1B`
`gfmult`	Galois Field Multiply⋆	R	`00`	`3F`
`j`	Jump	J	`02`	—
`jal`	Jump and Link	J	`03`	—
`jr`	Jump Register	R	`00`	`08`
`lb`	Load Byte⋆	I	`20`	—
`lbu`	Load Byte Unsigned⋆	I	`24`	—
`lh`	Load Halfword⋆	I	`21`	—
`lhu`	Load Halfword Unsigned⋆	I	`25`	—
`lui`	Load Upper Intermediate	I	`0F`	—
`lw`	Load Word	I	`23`	—
`mfhi`	Move From HI Register⋆	R	`00`	`10`
`mflo`	Move From LO Register⋆	R	`00`	`12`
`mthi`	Move To HI Register⋆	R	`00`	`11`
`mtlo`	Move To LO Register⋆	R	`00`	`13`
`mult`	Multiply Word⋆	R	`00`	`18`
`multu`	Multiply Word Unsigned⋆	R	`00`	`19`
`nop`	No Operation	R	`00`	`00`
`nor`	Not Or	R	`00`	`27`
`or`	Or	R	`00`	`25`
`ori`	Or Immediate	I	`0D`	—
`sb`	Store Byte⋆	I	`28`	—
`sh`	Store Halfword⋆	I	`29`	—
`sll`	Shift Word Left Logical	R	`00`	`00`
`slt`	Set on Less Than	R	`00`	`2A`
`slti`	Set on Less Than Immediate	I	`0A`	—
`sltiu`	Set on Less Than Immediate Unsigned	I	`0B`	—
`sltu`	Set on Less Than Unsigned	R	`00`	`2B`
`sra`	Shift Word Right Arithmetic⋆	R	`00`	`03`
`srl`	Shift Word Right Logical	R	`00`	`02`
`sub`	Subtract Word	R	`00`	`22`
`subu`	Subtract Unsigned Word	R	`00`	`23`
`sw`	Store Word	I	`2B`	—
`syscall`	System Call	R	`00`	`0C`
`xor`	Exclusive Or	R	`00`	`26`
`xori`	Exclusive Or Immediate	I	`0E`	—

Tabela 3.2: Mapeamento da memória

Endereço	Uso
00000000 a 000000FF	Código de *boot*
00400000 a 00403FFF	Segmento `.text`
10010000 a 10011FFF	Segmento `.data`
10012000 a 10091FFF	Memória SRAM
10091FFC	Pilha
80000000 a 80001FFF	Segmento `.ktext`
90000000 a 90000FFF	Segmento `.kdata`
FF000000	`MMIO_START_ADDRESS`
FF000000 a FF012BFF	Memória de vídeo
FFFF0000	`AUDIO_INL_ADDRESS`
FFFF0004	`AUDIO_INR_ADDRESS`
FFFF0008	`AUDIO_OUTL_ADDRESS`
FFFF000C	`AUDIO_OUTR_ADDRESS`
FFFF0010	`AUDIO_CTRL1_ADDRESS`
FFFF0014	`AUDIO_CTRL2_ADDRESS`
FFFF0100	`KEYBOARD_BUFFER0_ADDRESS`
FFFF0104	`KEYBOARD_BUFFER1_ADDRESS`
FFFF0110	`KEYBOARD_MOUSE_ADDRESS`
FFFF0114	`BUFFERMOUSE_ADDRESS`
FFFF0130 a FFFF014F	LCD
FFFF0150	LCD Clear

Capítulo 4

Resultados

Este capítulo apresenta os critérios utilizados para avaliar o processador e a cifra, bem como uma análise dos resultados obtidos.

4.1 Processador

As novas funcionalidades implementadas no processador, como a visualização de registradores, representam melhorias subjetivas no projeto. Para avaliar o processador de forma mais objetiva, analisamos a frequência de clock e a quantidade de elementos lógicos utilizados, fazendo um contraste em relação à versão inicial.

4.1.1 Temporização

Utilizando a ferramenta TimeQuest Timing Analyzer, incluída no Quartus II, é possível analisar a temporização do circuito implementado em vários aspectos. Aqui, utilizamos a macro *Report Top Failing Paths* para identificar os caminhos do circuito que não conseguem cumprir as restrições impostas; os caminhos são listados a partir dos piores casos. Para realizar esta análise, fizemos os seguintes passos dentro do TimeQuest:

1. Na janela *Tasks*, clicar duas vezes em *Create Timing Netlist*;
2. No menu *Constraints*, selecionar *Create Clock...* e criar um clock com o alvo `CLK`;
3. Na janela *Tasks*, clicar duas vezes em *Update Timing Netlist*;
4. Na janela *Tasks*, clicar duas vezes na macro *Report Top Failing Paths*.

Observamos que uma restrição de 25 MHz (período de 40 ns) para o clock do processador (`CLK`) é cumprida pela implementação atual. Uma restrição de 50 MHz (período de 20 ns) para o clock, cogitada inicialmente, não pode ser cumprida por vários caminhos do circuito; mais especificamente, os piores casos são do caminho entre o banco de registradores e a escrita dos registradores IF/ID. Os registradores são escritos na borda de descida do clock e a saída deles define o sinal `wID_Equal` que, por sua vez, afeta o sinal `wIFID_Flush`, que, por fim, é necessário para definir a escrita dos registradores IF/ID. De fato, há apenas meio ciclo de clock para que o sinal percorra um caminho composto por uma quantidade significativa de multiplexadores; a não ser que o processador sofra uma

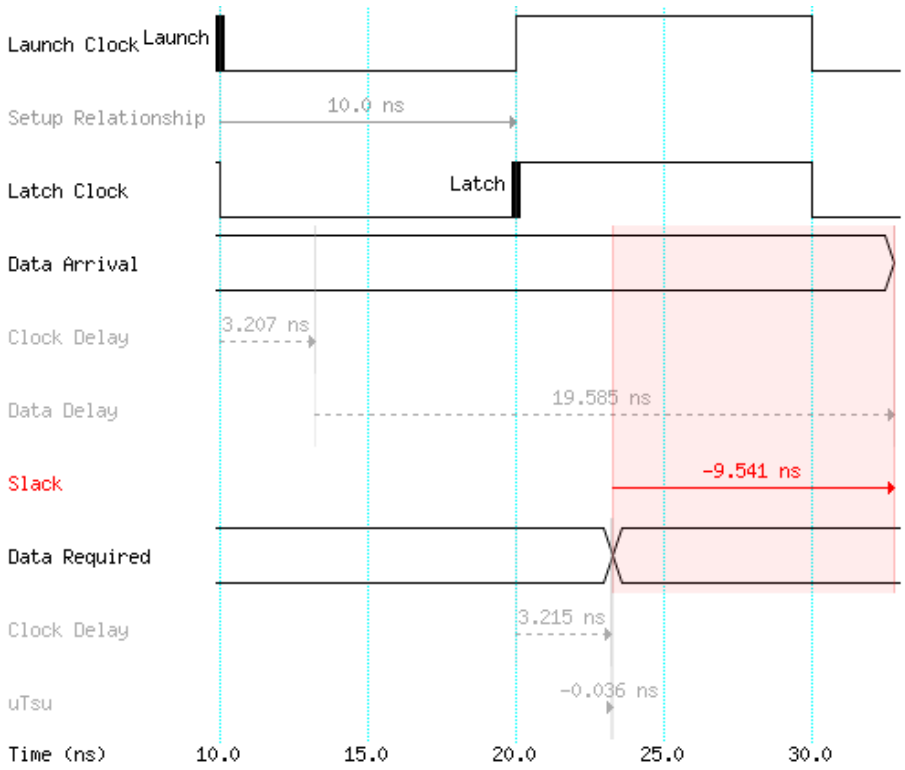

Figura 4.1: *Slack* do pior caminho para uma restrição de 50 MHz

mudança substancial, esta parte do projeto deve ser modificada para que seja possível utilizar frequências maiores de clock. A figura 4.1 mostra o diagrama de tempo apresentado pelo TimeQueste para o*slack* do caminho de pior caso.

4.1.2 Uso de recursos do FPGA

A tabela 4.1 apresenta dados referentes ao uso de elementos lógicos e de registradores do FPGA pelos principais módulos do projeto. Os dados foram obtidos pelo programa Quartus II, definindo cada módulo como entidade de topo e compilando-o. A tabela compara os dados da versão inicial, que serviu de base para este projeto, com a versão resultante das modificações descritas no capítulo 3.

É importante ressaltar que os dados apresentados não devem ser interpretados de forma absoluta. A versão atual do módulo `MultDiv`, por exemplo, utiliza 144 registradores, enquanto o módulo `ALU`, que instancia o `MultDiv`, utiliza 142 registradores. Algumas diferenças podem ocorrer por causa dos algoritmos de planejamento e ajuste de circuitos, mas a comparação entre módulos semelhantes, especialmente nos casos em que os valores são discrepantes, indica otimizações no uso de recursos.

Parte da redução na quantidade de elementos lógicos e registradores utilizados pelo módulo `Datapath` corresponde à retirada dos módulos `CodeMemory` e `DataMemory`, que agora são instanciados no módulo `TopDE`. Otimizações e eliminação de códigos redundantes e ineficientes também contribuíram para esta diminuição.

O módulo `ALU` teve um crescimento no uso de registradores por instanciar o módulo `MultDiv`, inexistente na versão inicial. Outro módulo instanciado dentro da `ALU` é o de

Tabela 4.1: Uso de recursos do FPGA pelos módulos

	Elementos lógicos		Registradores	
	Versão inicial	Versão atual	Versão inicial	Versão atual
`Datapath`	7885	6125	1953	1746
`ALU`	3830	1938	64	142
`MultDiv`	–	740	–	144
`Registers`	2110	2814	992	992
`Control`	54	50	0	0
`ForwardUnit`	28	28	0	0
`HazardUnit`	23	23	0	0
`VGAAdapter`	405	412	46	38
`RegDisplay`	–	180	–	0
`CodeMemory`	483	425	226	226
`DataMemory`	448	448	225	225
`gfmult`	–	56	–	0
`TopDE`	10256	9292	3077	3159

deslocamento aritmético, que possui um circuito bem simples e não utiliza registradores. Ainda assim, o uso de elementos lógicos diminuiu.

O aumento na quantidade de elementos lógicos utilizados pelo módulo `Registers` advém do circuito extra para selecionar um registrador e exibir seu conteúdo através do módulo `RegDisplay`. O módulo original possuía três destes seletores: dois para o processador selecionar os operandos e um para o usuário selecionar, através dos interruptores da placa de desenvolvimento, o registrador a ser exibido nos displays de 7 segmentos. O primeiro seletor para exibição de dados de registradores não foi eliminado por ser útil em aplicações que não utilizam a saída de vídeo. Uma possível solução para eliminar tal redundância seria utilizar um único seletor para os dois casos, que poderiam ser alternados por um único interruptor.

Os módulos `Control`, `ForwardUnit` e `HazardUnit` não sofreram mudanças significativas. Como implementam circuitos combinacionais, não utilizam registradores.

Algumas otimizações foram feitas no módulo `CodeMemory`, especialmente em relação à memória de *boot*. O módulo `DataMemory` não teve mudanças significativas.

O módulo `TopDE` é a entidade de topo e inclui todos os outros módulos. Considerando que novas funcionalidades foram implementadas e nenhuma foi retirada, a diferença de 964 elementos lógicos (9,40%) e o uso de apenas 82 registradores a mais (2,66%) indicam uma otimização significativa no uso de recursos do FPGA.

Outro resultado decorrente das modificações no processador é o número de alertas (*warnings*), que foi reduzido consideravelmente. A compilação da versão inicial produz 502 alertas, que foram reduzidos a 193 na versão atual.

4.2 AES-128

Para testar o correto funcionamento da cifra, utilizamos os vetores de exemplo do modo de operação ECB disponíveis em [8]. Após as operações de cifração e decifração,

os textos cifrados e textos decifrados são carregados nos registradores `$s0` a `$s7` para serem comparados com os valores esperados. Em todos os casos testados, a comparação foi bem-sucedida.

As estatísticas de execução no simulador MARS apresentadas na tabela 4.2 foram obtidas através da ferramenta *Instruction Counter* incluída no programa.

Tabela 4.2: Estatísticas de execução (MARS)

	Procedimento	Instruções			
		Tipo R	Tipo I	Tipo J	Total
Sem *delayed branch*	`main`	7	60	1	68
	`AES128_ECB_ENCRYPT`	11331	22844	554	34729
	`AES128_ECB_DECRYPT`	129879	79138	6170	215187
	Total	141217	102042	6725	249984
Com *delayed branch*	`main`	8	60	1	69
	`AES128_ECB_ENCRYPT`	12624	22844	554	36022
	`AES128_ECB_DECRYPT`	136788	79138	6170	222096
	Total	149420	102042	6725	258187

O código produzido pelo compilador não possui instruções úteis nos *branch delay slots*, contendo apenas a operação `nop`. Ao coletar dados de execução do código com e sem atraso no desvio, ativando e desativando a opção no MARS, podemos avaliar o impacto de *branch delay slots* no desempenho do código: 8203 ciclos de clock, ou 3,28% do total de instruções. É uma quantia significativa, que evidencia o impacto de desvios em implementações *pipeline*. A instrução `nop` é considerada do tipo R por corresponder à instrução `sll $0,$0,0`, cujo opcode é `00000000`; por isso, os dados das instruções dos tipos I e J não são afetados.

Após verificar tais dados, torna-se evidente que o procedimento `AES128_ECB_DECRYPT` requer a execução de muito mais instruções que o procedimento `AES128_ECB_ENCRYPT`. Uma breve análise do código-fonte ajuda a esclarecer o motivo: a função `MixColumns` chama quatro vezes a função `xtime`, enquanto a função inversa, `InvMixColumns`, utiliza 16 vezes a macro `Multiply` que, por vez, chama a função `xtime` 10 vezes, resultando em um total de 160 chamadas. Assim, este trecho de código se tornou um ótimo candidato para ser otimizado em hardware.

A tabela 4.3 apresenta os dados de execução no processador sintetizado em FPGA. Para obter tais dados, foram criados dois contadores, que são incrementados a cada ciclo de clock do processador e a cada ciclo com o sinal de *hazard* ativado. Estes contadores ficam acessíveis para leitura através de dois endereços da memória de entrada e saída.

Primeiramente, observamos que os dados coincidem com os apresentados na tabela 4.2: subtraindo o número de *stalls* dos ciclos de clock, os procedimentos `AES128_ECB_ENCRYPT` e `AES128_ECB_DECRYPT` executam a mesma quantidade de instruções que no simulador MARS. A pequena diferença quanto ao procedimento `main` se dá pelo uso da memória de *boot* no processador sintetizado, que apenas executa um desvio para o início do segmento `.text`, e pelo atraso do *pipeline* até a leitura do registrador mapeado na memória.

De posse destas informações, podemos estimar o desempenho desta implementação. Desconsiderando o *overhead* para calcular os argumentos do procedimento, são necessários 41082 ciclos de clock a uma frequência de 25 MHz para cifrar 128 bits. Isto corresponde

Tabela 4.3: Estatísticas de execução (FPGA)

	Procedimento	Ciclos de clock	*Stalls*
Sem `gfmult`	`main`	74	0
	`AES128_ECB_ENCRYPT`	41082	5060
	`AES128_ECB_DECRYPT`	240548	18452
	Total	281704	23512
Com `gfmult`	`main`	74	0
	`AES128_ECB_ENCRYPT`	38778	4772
	`AES128_ECB_DECRYPT`	45050	4998
	Total	83902	9760

a uma taxa de cifração de 77893 bits por segundo, ou 9,508 KiB/s. Seguindo o mesmo raciocínio, chegamos a uma taxa de decifração de 1,623 KiB/s. Utilizando a instrução `gfmult`, a taxa de cifração sobe para 10,073 KiB/s e a de decifração alcança a marca de 8,671 KiB/s.

Capítulo 5

Conclusão

Este trabalho apresentou uma proposta de aceleração da implementação do algoritmo AES-128 pela incorporação de uma nova instrução na arquitetura MIPS. Inicialmente, estudos sobre o estado da arte em criptografia, arquitetura de computadores e síntese de sistemas digitais em FPGA foram apresentados. A seguir, o processador MIPS com estrutura *pipeline* de 5 estágios foi apresentado e implementado em um componente FPGA. O algoritmo AES-128 foi estudado e, a partir de uma codificação padrão em C, foi compilado e modificado de forma a ser executável no processador MIPS implementado. Observou-se que grande parte do consumo de tempo na execução do algoritmo se deve à operação de multiplicação em GF(2^8). Deste modo, propomos a inclusão da instrução `gfmult` em sua arquitetura. Esta instrução implementa a multiplicação em GF(2^8) diretamente em hardware, demandando um único ciclo de clock. Os resultados obtidos indicam que a implementação MIPS convencional, com processador em uma frequência de clock de 25 MHz, atinge uma taxa de cifração de 9,508 KiB/s e uma taxa de decifração de 1,623 KiB/s. Com a incorporação da nova instrução, obteve se um ganho de 5,94% na cifração e 434,26% na decifração. O aumento em utilização de hardware foi de 43 elementos lógicos de uma FPGA da fabricante Altera, correspondendo a um aumento de 0,5%.

5.1 Trabalhos futuros

Aqui, apresentamos possíveis direções para as quais este trabalho pode ser expandido.

5.1.1 Processador

O processador desenvolvido neste projeto, ainda que capaz de executar códigos de razoável complexidade, carece de alguns recursos. Podemos citar, por exemplo, a ausência dos coprocessadores 0 (tratamento de exceções e interrupções) e 1 (unidade de ponto flutuante). A placa de desenvolvimento DE2-70 oferece muitos recursos que o projeto, em seu estado atual, não utiliza, como memórias DRAM, memória Flash e interface para cartões SD.

O módulo de multiplicação e divisão foi implementado com o objetivo de disponibilizar as instruções ao processador, sem focar em otimização. Uma implementação que utilize menos ciclos de clock para concluir os cálculos, como os algoritmos apresentados por Patterson e Hennessy [16], pode ser implementada sem muitas dificuldades; múltiplas

implementações podem ser incluídas no projeto e deixadas como opção em tempo de compilação.

5.1.2 Desempenho

O desempenho apresentado pelo algoritmo neste processador pode ser melhorado em três pontos básicos:

- Otimização do processador: sofisticando a implementação do processador em Verilog com o objetivo de possibilitar a operação em frequências maiores, o desempenho da cifração/decifração também será beneficiado. No entanto, a taxa de cifração por clock não é afetada, e a frequência limite de operação do FPGA, juntamente com a complexidade de um processador *pipeline*, deixam pouco espaço para otimizações neste aspecto.
- Otimização do código: a implementação em C utilizada no projeto foca em portabilidade e simplicidade. Uma implementação que priorize o desempenho e recorra a otimizações por parte do compilador pode resultar em ganhos expressivos. A sub-rotina `xtime`, por exemplo, possui 26 instruções na implementação atual e é chamada 16 vezes durante a cifração e 160 vezes durante a decifração. Como a sub-rotina recebe 1 byte de argumento e retorna 1 byte, ela poderia ser substituída por uma tabela *lookup* de 256 bytes, podendo ser consultada em uma única instrução.
- Otimização em hardware: utilizando hardware específico para a cifra AES, um processador moderno pode chegar a taxas de cifração de 1,5 ciclos por byte em uma frequência de operação de 2,67 GHz [10], enquanto uma implementação em hardware de todo o algoritmo pode chegar a taxas de 25 Gb/s [9]. Dependendo do modo de operação utilizado, é possível paralelizar a cifração e/ou a decifração, levando a taxas ainda maiores. A otimização pode variar desde instruções para operações específicas até um coprocessador dedicado à função.

Referências

[1] Altera DE2-70 Board. http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=226. Acessado em 02/03/2015. vii, 20

[2] File:AES-ShiftRows.svg. http://en.wikipedia.org/wiki/File:AES-ShiftRows.svg. Acessado em 02/03/2015. vii, 16

[3] Tiny AES128. https://github.com/kokke/tiny-AES128-C. Acessado em 02/03/2015. 27

[4] Using as. https://sourceware.org/binutils/docs/as/. Acessado em 02/03/2015. 28

[5] ARM Limited. *ARM7TDMI-S Technical Reference Manual*, 2000. Disponível em http://infocenter.arm.com/help/topic/com.arm.doc.ddi0084f/DDI0084.pdf. 5

[6] Peter J. Ashenden. *Digital Design: An Embedded Systems Approach Using Verilog*. Morgan Kaufmann, Burlington, MA, 2007. 1

[7] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002. viii, 9, 16

[8] Morris Dworkin. NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, 2001. Disponível em http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. vii, 17, 18, 35

[9] Tim Good and Mohammed Benaissa. AES on FPGA from the fastest to the smallest. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 427–440. Springer, 2005. 39

[10] Shay Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. Technical report, Intel Mobility Group, 2010. Disponível em https://software.intel.com/sites/default/files/m/d/4/1/d/8/AES_WP_Rev_03_Final_2010_01_26.pdf. 2, 39

[11] Joseph Heinrich. *MIPS R4000 User's Manual*. Prentice Hall, Englewood Cliffs, NJ, 1993. 1

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Burlington, MA, 2011. 5

[13] Jian Liang, Russell Tessier, and Oskar Mencer. Floating point unit generation and evaluation for FPGAs. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 185–194. IEEE, 2003. 8

[14] Ross MacGregor. Regular Expression Search and Replace Command Line Tool. https://sites.google.com/site/regexreplace/. Acessado em 02/03/2015. 28

[15] MIPS Technologies. *MIPS32 Architecture for Programmers*, 2013. Disponível em http://www.imgtec.com/mips/mips32-architecture.asp. 1, 20

[16] David A. Patterson and John L. Hennessy. *Organização e Projeto de Computadores*. Campus, São Paulo, 2005. vii, 2, 4, 6, 7, 22, 26, 38

[17] NIST FIPS Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001. Disponível em http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. 15

[18] Douglas R. Stinson. *Cryptography: theory and practice*. CRC Press, 2005. 2, 9, 10

[19] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann, Burlington, MA, 2006. 1, 23