UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Proposta de implementação dos algoritmos de hash MD5 e SHA-1 em hardware reconfigurável

Carlos Eduardo de Barros Santos Júnior

Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Número de Ordem do PPgEEC: M524
Natal, RN, junho de 2018

Universidade Federaldo Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Santos Júnior, Carlos Eduardo de Barros.
Proposta de implementação dos algoritmos de hash MD5 e SHA-1 em hardware reconfigurável / Carlos Eduardo de Barros Santos Júnior. - 2018.
75f.: il.

Universidade Federal do Rio Grande do Norte, Departamento de Engenharia da Computação e Automação, Programa de Pós-Graduação em Engenharia Elétrica e de Computação, Natal, 2018.
Orientador: Dr. Marcelo Augusto Costa Fernandes

1. Algoritmos de hash - Dissertação. 2. FPGA - Dissertação. 3. Throughput - Dissertação. I. Fernandes, Marcelo Augusto Costa. II. Título.

RN/UF/BCZM CDU 004.021
Elaborado por RAIMUNDO MUNIZ DE OLIVEIRA - CRB-15/429

Universidade Federal do Rio Grande do Norte

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

ATA Nº 524

Aos 20 dia do mês de junho do ano de dois mil e dezoito, foi realizada a 524ª sessão de defesa de dissertação de mestrado do Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN, na qual o mestrando Carlos Eduardo de Barros Santos Júnior apresentou o trabalho que tem como título:Proposta de implementação dos algoritmos de hash MD5 e SHA-1 em hardware reconfigurável. A sessão teve início às 09h00min, tendo a banca examinadora sido constituída pelos seguintes participantes: Marcelo Augusto Costa Fernandes (Dr. UFRN, Presidente), Ivanovitch Medeiros Dantas Da Silva (Dr. UFRN, Examinador Interno ao Programa), Carlos Eduardo Da Silva (Dr. UFRN, Examinador Externo ao Programa), Valentin Obac Roda (Dr. UFRN, Examinador Externo ao Programa) e Adriano Mauro Cansian (Dr. UNESP, Examinador Externo à Instituição). Após a apresentação do trabalho e o exame pela banca, o mestrando foi considerado APROVADO, tendo sido lavrada a presente ata, que vai assinada pelos examinadores e pelo mestrando. A versão final da dissertação deverá ser entregue ao programa, no prazo máximo de 60 dias, contendo as modificações sugeridas pela banca examinadora e constante na folha de correção anexa. Conforme o Artigo 49 da Resolução 197/2013 - CONSEPE, o candidato não terá o título se não cumprir as exigências acima.

Dr. ADRIANO MAURO CANSIAN, UNESP

Examinador Externo à Instituição

Dr. CARLOS EDUARDO DA SILVA, UFRN

Examinador Externo ao Programa

Dr. VALENTIN OBAC RODA, UFRN

Examinador Externo ao Programa

Dr. IVANOVITCH MEDEIROS DANTAS DA SILVA, UFRN

Examinador Interno

Dr. MARCELO AUGUSTO COSTA FERNANDES, UFRN

Presidente

CARLOS EDUARDO DE BARROS SANTOS JÚNIOR

Mestrando

À minha esposa e filho, pela paciência e compreensão durante a realização deste trabalho.

Agradecimentos

Agradeço ao meu Deus, pelo dom da vida e pela constante dose de sabedoria e graça.

Ao meu orientador, prof. Marcelo, pelo continuo ensino e orientações para manter-me focado neste trabalho. Além da credibilidade a mim confiada e no potencial desta pesquisa. Sem o auxílio do professor Marcelo possivelmente teria fadado nesta jornada.

A todos os amigos e companheiros do Grupo de Pesquisa em Sistemas Embarcados e Hardware Reconfigurável (GPSEHR). Em especial ao meu amigo Sérgio Natan pelo apoio, incentivo e recepção inicial. Ao Caio pela parceria na escrita de alguns papers.

Aos demais colegas de pós-graduação, pelas críticas e sugestões.

À minha família pela compreensão e incentivo incondicional durante mais este desafio em minha vida, dividindo um pouco do tempo entre trabalho, estudos e esportes. Principalmente a minha mãe pela dedicação e esforço em mim ensinar e educar, ao meu irmão Jorge Bitú Neto pela grande parceria e confidencialidade, a minha adorável e paciente esposa, Rosy, e ao meu amado, criativo e corajoso filho, Davi.

Aos meus amigos e companheiros de trabalho pela assistência e apoio durante minhas ausências.

Aos meus alunos pelos tantos aprendizados em sala de aula e a minha supervisora pedagógica, Jéssica, pela confiança.

Aos amigos Rodrigo Jorge e Noilson Caio, por acreditarem no potencial das minhas pesquisas e por me motivarem a prosseguir. E aos amigos do grupo InfoSec Brasil por tantos conselhos e ensinamentos.

Resumo

Este trabalho tem como objetivo propor dois hardwares de aplicação específica (*Application Specific System Processor, ASSP*), sendo um para o algoritmo MD5 e o outro para o algoritmo SHA-1, ambos implementados em um *Field Programmable Gate Array* (FPGA) Xilinx Virtex 6 xc6vlx240t-1ff1156. As métricas utilizadas para verificar a eficácia das implementações foram a vazão dos dados (throughput), a área de circuito ocupada, e o consumo de energia. Na qual cada uma foi analisada para várias implementações em instâncias paralelas dos algoritmos. Os resultados mostraram que o hardware proposto para o MD5 alcançou um throughput bem superior aos encontrados em artigos publicados e foi possível implementar 320 instâncias do algoritmo em um único FPGA. Para o algoritmo SHA-1 o throughput e a área ocupada pelos circuitos internos no chip também foram surpreendentes. Várias aplicações como, recuperação de senha (por meio do ataque de força bruta), validação de senha e verificação de integridade de grande volume de dados podem ser executadas de forma eficiente e rápida com um ASSP para o MD5 e para o SHA-1. A métrica do consumo de energia foi avaliada por meio de uma análise comparativa com outras três plataformas de hardware distintas, sendo um micro-processador (μP) de 8 bits, um μP de 32 bits e os hardwares de aplicação específica projetados para cada algoritmo. Os resultados de estimativa de consumo a partir do tempo de processamento (medidos em laboratório) mostram que a utilização dos hardwares dedicados apresentam ganhos significativos de economia de energia.

Palavras-chave: hash, FPGA, MD5, SHA-1, processamento.

Abstract

This work proposes two Application Specific System Processor (ASSP), one to the MD5 algorithm and other to the SHA-1 algorithm implemented on Field Programmable Gate Array (FPGA) Xilinx Virtex 6 xc6vlx240t-1ff1156. The throughput and the occupied area were analyzed for several implementations on parallel instances of those algorithms. The results showed that the hardware proposed for MD5 achieved a better throughput than those found in published articles and it was possible to implement 320 instances of the algorithm in a single FPGA. For the SHA-1 algorithm the throughput and the area occupied by the internal circuits on the chip were also surprising when compared with other papers. Several applications such as password recovery, password validation, and high volume data integrity checking can be performed efficiently and quickly with an ASSP for MD5 and SHA-1. This work also presents a comparative analysis of the energy consumption associated with execution of the MD5 and SHA-1 algorithms for three different hardware platforms, a microprocessor (μP) of 8 bits and 32 bits and the specific application hardware designed for each algorithm. Results of consumption estimation from the processing time (measured in the laboratory) show that the use of dedicated hardware presents significant gains in energy savings.

Keywords: hash, FPGA, MD5, SHA-1, processing.

Sumário

Sumário i

Lista de Figuras iii

Lista de Tabelas v

Lista de Siglas e Abreviaturas v

1 Introdução 1

1.1 Trabalhos Relacionados . . . 2
1.1.1 Algoritmo MD5 . . . 3
1.1.2 Algoritmo SHA-1 . . . 4
1.2 Contribuições . . . 6
1.3 Publicações Relacionadas . . . 6
1.3.1 Artigos Científicos Publicados . . . 6
1.3.2 Palestras técnicas apresentadas em eventos . . . 6
1.4 Estrutura do documento . . . 7

2 Segurança da Informação 9

2.1 Fundamentos de Segurança da Informação . . . 9
2.1.1 Confidencialidade . . . 9
2.1.2 Integridade . . . 10
2.1.3 Disponibilidade . . . 10
2.1.4 Autenticidade . . . 10
2.1.5 Não repúdio . . . 11
2.2 Hashes . . . 11
2.3 Algoritmo MD5 . . . 12
2.3.1 Inserção do Preenchimento . . . 12
2.3.2 Inserção do Comprimento . . . 13
2.3.3 Inicialização do Código Hash . . . 14
2.3.4 Divisão da Mensagem . . . 14
2.3.5 Inicialização das Variáveis Hash $\mathbf{H}(n)$ . . . 14
2.3.6 Cálculo da Função $f(\cdot)$ . . . 15
2.3.7 Atualização das Variáveis Hash . . . 15
2.3.8 Atualização do código *hash* . . . 16
2.4 Algoritmo SHA-1 . . . 17

2.4.1 Inserção do Preenchimento 18
2.4.2 Inserção do Comprimento 18
2.4.3 Inicialização do Código Hash 18
2.4.4 Divisão da Mensagem 19
2.4.5 Inicialização das Variáveis Hash $\mathbf{H}(n)$ 19
2.4.6 Cálculo da variável $\mathbf{w}(n)$ 20
2.4.7 Cálculo da Função $f(\cdot)$ 20
2.4.8 Atualização das Variáveis Hash 20
2.4.9 Atualização do código *hash* 21

3 Computação Reconfigurável 23
3.1 Dispositivos Lógicos Programáveis 23
3.2 FPGA 24
3.2.1 Implementação de circuitos em FPGA 25
3.2.2 Consumo de Energia 28

4 Implementações 29
4.1 Proposta de Implementação do MD5 29
4.1.1 Módulo GF 30
4.1.2 Módulo GG 30
4.1.3 Módulo GQ 31
4.1.4 Módulo LR 32
4.1.5 Processamento do hash $\mathbf{h}_i$ 32
4.2 Proposta de Implementação do SHA-1 34
4.2.1 Módulo GF 36
4.2.2 Módulo GW 36
4.2.3 Processamento do hash $\mathbf{h}_i$ 37

5 Resultados Alcançados 41
5.1 Resultados para a implementação do MD5 41
5.2 Resultados para a implementação do SHA-1 43
5.3 Resultados relacionados ao consumo de potência 44

6 Conclusão 49

Referências bibliográficas 51

Lista de Figuras

3.1 Arquitetura interna do FPGA 25
3.2 Arquitetura interna de um CLB da Intel Cyclone V. 26
3.3 Fluxo de mapeamento do FPGA. Fonte: Próprio autor. 27

4.1 Arquitetura geral da implementação em hardware do MD5 proposto. . . . 30
4.2 Arquitetura do módulo GF. 31
4.3 Arquitetura do módulo GG. 32
4.4 Arquitetura do módulo GQ. 33
4.5 Arquitetura do módulo LR. 34
4.6 Arquitetura do i-ésimo módulo SLR$i(k)$. 35
4.7 Arquitetura geral da implementação em hardware do SHA-1 proposto. . . 36
4.8 Arquitetura do módulo GF. 37
4.9 Arquitetura do módulo GW 38
4.10 Operações do módulo SWk 38
4.11 Arquitetura do módulo RLs. 39

5.1 Grafico dos resultados de potência dinâmica. 47

Lista de Tabelas

1.1 Tabela comparativa entre as implementações do algoritmo MD5 publicadas e a proposta nesta dissertação. 4
1.2 Tabela comparativa entre as implementações do algoritmo SHA-1 publicadas e a proposta nesta dissertação. 6

5.1 Resultados relativos a ocupação, taxa de amostragem e *throughput* para várias implementações paralelas do algoritmo MD5. 42
5.2 Resultados relativos a ocupação, taxa de amostragem e *throughput* para várias implementações paralelas do algoritmo SHA-1. 43
5.3 Resultados relativos ao tempo de execução dos algoritmos MD5 e SHA-1 nas plataformas ATmega 2560 e Intel Galileo Gen 2. 45
5.4 Resultados relativos a ocupação, taxa de amostragem e *throughput* para as implementações em FPGA. 46
5.5 Resultados relativos as potências dinâmicas por cada plataforma de hardware, considerando o *clock* calculado para alcançar o tempo dos μP de 8 e 32-bits. 46

Lista de Siglas e Abreviaturas

R_s *Throughput*

T_s *Tempo Crítico*

μC *Microcontrolador*

μP *Microprocessador*

FLU *Full Loop Unrolling*

FP *Full Pipeline*

FPGA *Field Port Gate Array*

IL *Iterative Loop*

IoT *Internet of Things*

LC *Logic Cell*

MD *Message Digest*

PLU *Pipeline Loop Unrolling*

SHA *Secure Hash Algorithm*

Capítulo 1

Introdução

Com o surgimento de novas tecnologias, o crescimento populacional, a demanda por grande manipulação de dados e o desenvolvimento de sistemas relacionados a Internet das Coisas (IoT, *Internet of Things*) e *Big Data* mudou o modo de operação dos novos projetos, de modo que houve um aumento na priorização de um processamento rápido e eficaz das informações. E ainda, uma preocupação com a capacidade energética e de potência consumida pelos equipamentos que executam tais ações.

Pesquisas relacionadas ao melhoramento no desempenho de sistemas computacionais são antigas, porém vêm sendo aprimoradas a cada ano com o surgimento de novas tecnologias e modelos de processadores de uso geral. Nesta mesma perspectiva o uso de FPGAs (*Field Programmable Gate Arrays*) têm aumentado e, segundo uma pesquisa elaborada pela (TECHNAVIO 2017), há uma forte tendência de crescimento destes dispositivos até 2021.

Com isso, há um aspecto importante a ser considerado nos projetos de grande processamento de dados a segurança da informação, a qual pode ser definida por três significativos pilares: a confiabilidade, a integridade e a disponibilidade (ISO/IEC27002 2005). Sabendo dessa crescente geração de dados é que esta temática se tornou algo primordial, a fim de se proteger uma quantidade diferenciada de dados e tipos de informações as quais estão sendo armazenadas em mídias digitais, como exemplo os servidores de arquivos locais ou em nuvem. Muitos desses dados podem possuir informações sigilosas ou vitais para empresas e pessoas, ao que ocorre com prontuários médicos eletrônicos em ambientes hospitalares.

Um dos fatores motivacionais para o desenvolvimento deste trabalho levou em conta o aumento de dados a serem armazenados e a preocupação dos detentores desses dados em mantê-los minimamente seguros, tendo em vista que por meio dos algoritmos de hash é possível verificar a integridade dessas informações tanto em larga escala quanto em equipamentos de baixa potência, como os utilizados em Internet das Coisas (IoT - *Internet of Things*). Um outro fator é a possibilidade de testes em ambientes computacionais aos quais se armazenam hashes, sobre tudo senhas, como os ataques de força bruta e de dicionário, pois torna-se possível gerar vários hashes em alta velocidade e compará-los com os que se encontram armazenados por meio da implementação em hardware. Além desses, outro motivador foram os vazamentos de credenciais de usuários (login e senha) com códigos hashes dos algoritmos MD5 e SHA-1, tais quais os que ocorreram

em empresas como Yahoo (WIRED 2016), Linkedin (THE REGISTER 2016), MySpace (MOTHERBOARD 2016) e Dropbox (FORBES 2016) (causa pela qual tais algoritmos foram escolhidos). Visto que essas bases possuiam uma grande quantidade de hashes, que consomiriam maior processamento e demorariam mais tempo ao tentar executar os ataques de quebra de senha em um computador com processadores de uso geral do que em uma solução utilizando hardware reconfigurável. Outrosim, este trabalho torna possível atribuir velocidade à verificação da integridade de documentos, arquivos ou informações armazenadas em banco de dados. Também é possível obter um processamento semelhante aos dos hardwares utilizados em equipamentos gerais conectados à internet como os IoT, porém com a característica de consumir menos energia que esses. Tendo em vista, que uma das vantagens em desenvolver circuitos específicos em hardware é a redução do *clock* quando comparado a implementações em sistemas com processadores de uso geral ou GPP (*General Purpose Processor*).

Para solucionar tal situação foi escolhido fazer uso dos FPGAs, os quais podem ser definidos, inicialmente, como uma plataforma de hardware reconfigurável formada por milhares de células lógicas, que após um processo de síntese comporta-se como um hardware específico associado a um dado algoritmo. O FPGA tem sido uma ferramenta indispensável no desenvolvimento de ASSPs (*Application Specific System Processor*) e circuitos integrados dedicados, também chamados de ASIC (*Application-Specific Integrated Circuit*) e ainda como plataforma para aceleração de algoritmos complexos como apresentado em (de Souza & Fernandes 2014, da Silva et al. 2016, Noronha & Fernandes 2016, Torquato & Fernandes 2016, Shi et al. 2012).

Assim, este trabalho teve como objetivo alcançado a implementação os algoritmos de hashes MD5 e SHA-1 em uma plataforma de hardware reconfigurável do tipo FPGA, da fabricante Xilinx e modelo Virtex 6 xc6vlx240t-1ff1156. As métricas utilizadas para analisar o desempenho da solução implementada neste trabalho com as demais publicações encontradas foram o throughput (vazão dos dados), a área ocupada e o consumo de energia. Para isso, foi montado um pseudocódigo de cada algoritmo, a fim de ter um entendimento detalhado do funcionamento desses para que fosse possível a implementação em FPGA utilizando os diagramas de bloco em RTL.

Na seção seguinte observa-se os trabalhos tomados como base para o desenvolvimento e referência do relatado e implementado nesta dissertação.

1.1 Trabalhos Relacionados

Os algoritmos em estudo são conhecidos na comunidade acadêmica e há algumas implementações dos tais em hardware reconfigurável. Nesta seção, dividida em duas subseções (um para cada algoritmo), são abordados os principais *papers* relacionados ao trabalho em questão. O qual teve como métrica as referências e comparativos de *throughput* (a quantidade de bits que passam em um fluxo contínuo do algoritmo a cada segundo no circuito implementado) e o consumo de energia.

1.1.1 Algoritmo MD5

No trabalho apresentado em (Deepakumara et al. 2001), são propostas duas arquiteturas de hardware para o algoritmo MD5 utilizando FPGA (Xilinx Virtex 5 I000FG680-6). Sendo uma chamada de *Full Loop Unrolling* (FLU) e a outra de *Iterative Looping* (IL), as quais obtiveram um *throughput* de 354 Mbps (*clock* de 21 MHz) e 156 Mbps (*clock* de 71,4 MHz), respectivamente. Todavia, a implementação com FLU ocupa em torno de 5 vezes mais espaço que a implementação em IL. O trabalho é um dos pioneiros em concluir a possibilidade real de utilizar FPGAs para acelerar o processamento de algoritmos de hashes.

Outra proposta de implementação em hardware do MD5 é apresentada em (Jarvinen et al. 2005), utilizando também um FPGA (Xilinx Virtex-II XC2V4000-6) para o desenvolvimento do protótipo e a realização dos testes. No trabalho citado, são também propostos dois esquemas de implementação, no qual, o primeiro é chamado de *Iterative* e segue a mesma linha do IL apresentado em (Deepakumara et al. 2001) e o segundo utiliza estágios de pipeline chamado de *Full Pipeline* (FP) para diminuir o caminho crítico do circuito. Para o caso IL foi encontrado um *throughput* de 586 Mbps (*clock* de 75,5 MHz) e para o FP de 725 Mbps (*clock* de 93,4 MHz). Todavia a proposta FP ocupa uma área de circuito de aproximadamente 8 vezes maior que o IL. Diferentemente da proposta apresentada em (Deepakumara et al. 2001), o artigo apresentado em (Jarvinen et al. 2005) propõem também implementações em hardware de vários módulos (*cores*) de MD5 em paralelo objetivando melhorar a performance do algoritmo, para este caso foi atingido um *throughput* de aproximadamente 5,8 Gbps para 10 módulos de MD5 executados em paralelo em um único FPGA.

Já o artigo (He & Xue 2010), propõe uma implementação utilizando o FPGA Altera de modelo Cyclone II EP2C35F672C6. Neste trabalho, semelhante ao apresentado em (Jarvinen et al. 2005), é proposto um esquema com vários módulos do MD5 em hardware trabalhando em paralelo. Para cada módulo MD5 foi alcançado um *throughput* de aproximadamente 362,92 Mbps (*clock* de 48,2 MHz). O trabalho conseguiu sintetizar até 12 módulos em paralelo em um FPGA atingindo 4,355 Gbps de *throughput*. Um ponto interessante neste trabalho é a alocação também de mais de um FPGA para o processamento do MD5, no qual, o artigo apresentado em (He & Xue 2010) demonstra resultados com a utilização de 3 FPGAs, atingindo um *throughput* de $3 \times 4{,}355\,\text{Gbps} \approx 13\,\text{Gbps}$. O artigo (Wang et al. 2010) segue uma proposta semelhante a apresentada no artigo (He & Xue 2010), que também faz várias comparações com os artigos apresentados em (Deepakumara et al. 2001, Jarvinen et al. 2005, Yiakoumis et al. 2005). Foi utilizada um FPGA da Altera modelo Stratix II GX e obtido aproximadamente 800 Mbps (*clock* de 102,7 MHz) de *throughput* para um módulo MD5 com esquema FLU. Neste trabalho os autores conseguiram embarcar até 32 módulos em paralelo atingindo um *throughput* de aproximadamente 32 Gbps.

Trabalhos com implementações do MD5 em outras plataformas de hardware podem ser encontrados em (Morais et al. 2016) , (Al-Kiswany et al. 2009*a*), (Marks & Niewiadomska-Szynkiewicz 2014*a*) e (Kim & Kyong 2012). Em (Morais et al. 2016), o MD5 executado em um GPP Intel i3-M370 2.4 GHz obteve um *throughput* de aproximadamente 148 Mbps. Já em (Al-Kiswany et al. 2009*a*) , (Marks & Niewiadomska-Szynkiewicz

2014*a*) , (Kim & Kyong 2012) comparações entre GPUs e CPUs (GPP - processadores de uso comum) são realizadas. As GPUs do tipo NVIDIA Tesla M2050 com 448 CUDA *cores* e AMD FirePro V7800 com 1440 *stream processors* podem conseguir picos de *throughput* de até 1,5 Gbps. Esses dados e dos demais trabalhos citados nessa subseção podem ser conferidos na Tabela 1.1, a qual mostra um resumo comparativo entre diversas implementações. Ainda na Tabela 1.1, o IL significa a implementação *Iterative Loop*, o FLU a *Full Loop Unrolling*, o FP a *Full Pipeline* e o PLU o *Pipeline Loop Unrolling*. O T_s é o tempo crítico, tempo ao qual um clock leva para ir do início ao fim do circuito, e R_s é o *throughput*.

Tabela 1.1: Tabela comparativa entre as implementações do algoritmo MD5 publicadas e a proposta nesta dissertação.

Referência	Modelo do FPGA	NI	T_s (ns)	R_s (Gbps)
(*Deepakumaraet al*. 2001)	Xilinx Virtex 5	1 (IL)	14,006	0,156
(*Deepakumaraet al*. 2001)	Xilinx Virtex 5	1 (FLU)	47,619	0,354
(*Jarvinenet al*. 2005)	Xilinx Virtex-II	10 (IL)	13,245	5,860
(*Jarvinenet al*. 2005)	Xilinx Virtex-II	4 (FP)	12,392	2,400
(*He& Xue* 2010)	Altera Cyclone II	36 (IL)	20,749	14,903
(*Wanget al*. 2010)	Altera Stratix II GX	32 (PLU)	9,737	32,035

A proposta aqui desenvolvida utilizou como alvo o FPGA Virtex 6 xc6vlx240t-11156 e os resultados mostraram um *throughput* maior que os apresentado na Tabela 1.1 para um único módulo MD5. A implementação utilizou a estratégia *Iterative Looping* que ocupa menos área de circuito quando comparada a outras estratégias (Deepakumara et al. 2001, Jarvinen et al. 2005) e diferentemente dos resultados apresentados na literatura, foi possível sintetizar (no FPGA alvo) até 320 módulos MD5 em uma única FPGA, conforme mostrado no Capítulo 5. Apesar dos modelos utilizados nas referências serem diferentes e mais antigos em relação ao utilizado neste trabalho, o que se deseja comparar é como o circuito foi estruturado dentro do FPGA e ainda em relação ao caminho crítico, o qual influencia no *throughput*. Isso é possível devido ao algoritmo MD5 ser pequeno e ocupar pouco espaço dentro do chip.

1.1.2 Algoritmo SHA-1

Nos trabalhos relacionados ao algoritmo de hash SHA-1, foi encontrado o apresentado em (Jarvinen 2004), que faz uso de um FPGA Xilinx Virtex-II XC2V2000-6 para implementar o SHA-1 por meio do método *Itarative Looping* (IL). Nessa implementação a proposta ocupou em torno de 1.275 *Logic Cells* (LC) operando a um *throughput* de 734 Mbps. Já os trabalhos apresentados em (Michail et al. 2005, Kakarountas et al. 2006) também utilizam um FPGA Xilinx Virtex-II XC2V2000-6 para implementar o SHA-1. Essas propostas apresentam um esquema utilizando *Full Pipeline* (FL) que ocupa em torno de 3.519 LC para um *throughput* de 2,5267 Gbps. Comparando com a proposta

apresentada em (Jarvinen 2004) o *throughput* é 4 vezes maior devido a utilização de 4 módulos do SHA-1 em pipeline, todavia a área de ocupação também é em torno de 4 vezes maior. Em (Lee et al. 2009) também é apresentada um proposta utilizando FL que consegue atingir um *throughput* de 5,9 Gbps.

Em (Iyer & Mandal 2013) foi realizada uma implementação do SHA-1 em um FPGA Xilinx Virtex 5 Xc5vlx50t. A implementação realizada foi com IL, semelhante a apresentada em (Jarvinen 2004). Todavia, possui uma taxa de ocupação um pouco maior, em torno de 1.351 LC, e *throughput* também, em torno de 786 Mbps.

O trabalho apresentado por (Khan et al. 2014) tem o objetivo de trazer uma solução do SHA-1 em FPGA com baixa potência energética para usos em dispositivos desprovidos de alta capacidade de energia e com alto *throughput*, além de um tamanho de área pequeno em relação as implementações comparadas. Para isto, os autores se baseiam nos trabalhos apresentados em (Michail et al. 2005, Kakarountas et al. 2006). Neste trabalho o número de LC é reduzido fazendo uma implementação mais serial, reduzindo também o *throughput*. Uma outra abordagem baseada na implementação descrita em (Michail et al. 2005, Kakarountas et al. 2006) é mostrada em (Michail et al. 2016), no qual, se propõem uma implementação em TSMC 90 nm *Application-Specific Integrated Circuit* (ASIC). Nesta proposta observa-se um *throughput* em torno de 15 Gbps.

Um comparação entre várias plataformas de FPGA Xilinx com implementação do SHA-1 é apresentada em (Michail et al. 2014). A implementação é baseada na proposta apresentada em (Michail et al. 2005, Kakarountas et al. 2006) e observa-se um *throughput* máximo em torno de 14,3 Gbps para um FPGA Xilinx Virtex 7.

Trabalhos com implementações do SHA-1 em outras plataformas de hardware podem ser encontrados em (Marks & Niewiadomska-Szynkiewicz 2014*b*, Al-Kiswany et al. 2009*b*) no qual comparações entre GPUs e CPUs (GPP) são realizadas. As GPUs do tipo NVIDIA Tesla M2050 com 448 CUDA *cores* e AMD FirePro V7800 com 1440 *stream processors* podem conseguir picos de *throughput* de até 1,5 Gbps.

Já a proposta aqui desenvolvida utilizou como hardware alvo o FPGA Virtex 6 xc6vlx2 40t-11156, com uma montagem de circuito diferente das citadas, porém seguindo a estratégia *Iterative Looping* que ocupa menos área de circuito quando comparada a outras estratégias (Michail et al. 2005, Kakarountas et al. 2006) e diferentemente dos resultados apresentados na literatura, foi possível sintetizar até 48 módulos SHA-1 em um único FPGA obtendo resultados ainda não conhecidos na literatura especializada, conforme pode ser visto no Capítulo 5.

Na Tabela 1.2 é possível visualizar uma comparação entre as implementações descritas nessa subseção. A primeira coluna mostra as referências, conforme apresentadas no texto, na segunda coluna estão os dispositivos utilizados nas implementações (modelos de FPGA e um ASIC). Na terceira coluna estão a quantidade de elementos lógicos utilizados (LC) e entre parênteses o tipo de implementação. Na quarta e quinta colunas estão os tempos e os *throughputs*, detalhados na seção 5.2. As colunas que possuem linhas com "–" fazem menção as informações não explícitas nas referências utilizadas.

Tabela 1.2: Tabela comparativa entre as implementações do algoritmo SHA-1 publicadas e a proposta nesta dissertação.

Referência	Modelo do FPGA	LC	T_s (ns)	R_s (Gbps)
(*Jarvinen* 2004)	Xilinx Virtex-II	1.275 (IL)	14,006	0,734
(*Michailet al.* 2005)	Xilinx Virtex-II	3.519 (FL)	–	2,5267
(*Leeet al.* 2009)	Xilinx Virtex-II	2.894 (FL)	–	5,900
(*Iyer& Mandal* 2013)	Xilinx Virtex 5	1.351(IL)	–	0,786
(*Khanet al.* 2014)	Xilinx Virtex-E	– (FL)	29,760	3,441
(*Michailet al.* 2016)	TSMC 90 nm (ASIC)	–	–	14,413

1.2 Contribuições

Esta dissertação tem como contribuição geral a reprodução dos algoritmos de hashes MD5 e SHA-1 em hardware reconfigurável, com obtenção de desempenho superior e um consumo energético mais baixo ao comparar com os processadores de uso geral e das referências citadas nas subseções 1.1.1 e 1.1.2. Utilizando como métrica o *throughtput*, a potência consumida e a quantidade de área utilizada pelos circuitos na implementação de cada algoritmo.

1.3 Publicações Relacionadas

Seguindo a temática deste trabalho três artigos foram publicados em conferências, sendo um nacional e dois regionais. Além de três palestras apresentadas em eventos, sendo dois de cunho regional e um nacional.

1.3.1 Artigos Científicos Publicados

Segue os artigos científicos publicados em Anais de Congresso, simpósio e Workshop:

- Carlos E. B. Santos Júnior e Marcelo A. C. Fernandes. Processador de Uso Específico para o Algoritmo SHA-1. Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg 2017). p.42-55.
- Carlos E. B. Santos Júnior e Marcelo A. C. Fernandes. Implementação do Algoritmo MD5 em FPGA. Workshop de Pesquisa Científica (WPC 2017.1). p.124-129.
- Carlos E. B. Santos Júnior, Sérgio S. Natan, Marcelo A. C. Fernandes. Proposta de Implementação do Algoritmo MD5 em Hardware. Escola Potiguar de Computação e suas Aplicações (EPOCA 2017). p.49-58.

1.3.2 Palestras técnicas apresentadas em eventos

Além dos trabalhos científicos publicados em anais, foram proferidas palestras em eventos conceituados relacionados a hacking e segurança da informação. As três palestras

tiveram como título **Dissecando e aplicando o SHA-1 em hardware reconfigurável**, proferias em:

- Qualitek Security Day. Natal-RN. jun. 2017.
- JampaSec - Conferência de Segurança da Informação. João Pessoa-PB. nov. 2017.
- GTER-44/GTS-30 (Grupo de Trabalho em Segurança de Redes). São Paulo-SP. dez. 2017.

1.4 Estrutura do documento

Esta dissertação de mestrado está dividida em cinco capítulos. No primeiro capítulo encontra-se a introdução, que traz uma visão geral do trabalho, por meio dos trabalhos relacionados, da motivação e contribuições, além dos artigos e palestras que foram resultados deste trabalho. No Capítulo 2 está descrito os principais conceitos relacionados a Segurança da Informação e apresentado em detalhes os algoritmos estudados e implementados neste trabalho, o MD5 e o SHA-1. No Capítulo 3 tem-se uma exposição simples do conceito de Hardware Reconfigurável, com o foco principal no FPGA. Esse capítulo aborda sobre os blocos lógicos, a diferença entre algumas plataformas de hardware e como ocorre o processo de desenvolvimento de uma aplicação em FPGA. Já o capítulo 4 aborda com detalhes cada passo da proposta de implementação dos algoritmos MD5 e SHA-1, com os respectivos módulos criados para melhorar o desempenho e facilitar o desenvolvimento. Após a obtenção dos resultados das implementações foi possível analisá-los e compará-los, os quais estão descritos no Capítulo 5. Isso por meio de uma abordagem baseada no *throughtput* e na área ocupada para cada implementação, além de um comparativo em relação ao consumo de energia. Por fim, finalizado por meio do Capítulo 6 com a conclusão dos fatos, perspectivas relatadas e trabalhos futuros.

Capítulo 2

Segurança da Informação

A segurança da informação tornou-se uma ferramenta primordial no bom andamento de um negócio, seja atrelado aos processos de trabalho, seja relacionado a respostas de auditoria, a preocupação com os dados dos clientes ou ainda com a reputação da empresa. Principalmente em um momento crescente de geração de dados e informações, no qual 90% de todos os dados gerados no mundo foram criados nos últimos dois anos, conforme descrito em (SINTEF 2013). Para este trabalho, define-se o termo informação como a descrição de "fatos e ideia que podem ser representadas em vários formatos de dados"ou ainda, como a "representação da informação por meio de símbolos que podem ser processadas por um computador", conforme a RFC 4949 (Shirey 2007). Assim, este capítulo aborda os conceitos iniciais de segurança da informação, os conceitos de hash e de forma detalhada como funciona os algoritmos MD5 e SHA-1.

2.1 Fundamentos de Segurança da Informação

A NIST (*National Institute of Standards and Technology*) fez uma publicação especial de número 800-12 (NIST 2017) definindo segurança de computadores como a proteção oferecida para um sistema de informação automatizado, a fim de alcançar os objetivos de preservar a integridade, disponibilidade e confidencialidade dos recursos de um sistema de informação. A ISO/IEC (*International Organization for Standardization* e *International Electrotechnical Commission*) (ISO/IEC 27000 2016) corrobora de definição semelhante em relação aos objetivos, porém menciona que também podem ser adicionado outros, tais como autenticidade, responsabilidade, não repúdio e confiabilidade. Os quais são descritos neste capítulo.

2.1.1 Confidencialidade

Na definição de confidencialidade, entende-se que apenas o remetente e o destinatário pretendido de uma mensagem transmitida poderão ser capazes de interpretá-la corretamente, ou seja, que a mensagem seja restrita aos que competem tê-la. A fim de que outros indivíduos não consigam interceptar e obter acesso a algum conteúdo transmitido. Para manter um nível aceitável de confidencialidade as mensagens podem ser cifradas de al-

guma forma, como por exemplo, por meio da criptografia ou esteganografia [1] (Kurose & Ross 2010). A criptografia, conforme (Stallings 2015), pode ser dividida em dois grupos: a criptografia simétrica e a criptografia assimétrica. A primeira faz uso de uma única chave para o processo de criptografia e decriptografia, já a assimétrica precisa de uma chave pública e uma chave privada, sendo que uma irá criptografar e somente a outra poderá descriptografar.

(Stallings 2015), menciona ainda o conceito relacionado a Privacidade dentro do contexto da confidencialidade, pois assegura que os indivíduos possam controlar quais informações possam ser capturadas e armazenadas, além de como, por quem e para quem essas informações são passadas.

2.1.2 Integridade

A (ISO/IEC 27000 2016) define a integridade como a propriedade de exatidão e completude da informação. Assim, oferece um modo a qual não seja possível a modificação ou corrupção dos dados em uma transmissão ou armazenamento por pessoas ou entidades não autorizadas. É possível analisar uma mensagem ou dado, se modificado ou não, por meio da utilização de hash [2].

Um exemplo de aplicação para a propriedade Integridade ocorre quando se gera um hash de determinada mensagem a ser transmitida e ambos são enviados (o hash e a mensagem) ao destinatário. Ao chegar, o destinatário terá como verificar se no percusso a mensagem enviada foi alterada de alguma maneira utilizando uma nova geração do hash e fazendo a comparação. Se o hash gerado no transmissor for igual ao do receptor, indica que a mensagem não teve alteração no percusso.

2.1.3 Disponibilidade

A disponibilidade assegura que os sistemas e serviços estejam operantes sempre que for solicitado por um usuário autorizado. (Lima Filho 2010) menciona que a informação não precisa está a todo o tempo *on-line*, mas sim sempre que for solicitado. E cita o exemplo do RAID (conjunto redundante de discos independentes), que possui redundância da informação, por meio de múltiplas cópias de dados em equipamentos distintos. Outro exemplo citado é o das cópias de segurança ou *backup* em mídias externas.

2.1.4 Autenticidade

A autenticidade é a propriedade que uma entidade é o que afirma ser, está relacionado a propriedade de ser genuíno e capaz de ser verificado e confiável (ISO/IEC 27000 2016). Ou seja, a chance de confirmar que um usuário é quem diz ser e que cada entrada em um sistema qualquer vem de uma fonte confiável (Stallings 2015).

---

[1] processo de embaralhar certa mensagem enviada por meio de imagens, músicas, textos ou vídeos, normalmente imperceptível a olho nu.

[2] código gerado por algoritmos matemáticos de caminho único, ou seja, não há como retornar à mensagem inicial se for feito o processo inverso do algoritmo.

2.1.5 Não repúdio

Está atrelado a capacidade de provar qual é a entidade de origem em uma ocorrência de um evento reivindicado ou ação (ISO/IEC 27000 2016). Ou seja, provê a irretratabilidade, capacidade de comprovação por parte do receptor que um emissor de fato enviou uma mensagem ao receptor, e que quando o receptor a receber consiga comprovar a origem, não há como negar que o emissor enviou a mensagem e nem que a origem (transmissor) não recebeu.

Um exemplo do não-repúdio acontece no envio de uma mensagem de e-mail utilizando certificado digital ou no uso de algum sistema de chave pública, pois uma autoridade certificadora comprovará que aquele é o certificado do remetente ou o uso da chave privada para criptografar a mensagem que será decriptografada por meio da chave pública é exclusiva do remetente, respectivamente.

2.2 Hashes

O código hash ou simplesmente hash ($\mathbf{h}_i$) é um valor de tamanho fixo C referente à saída de uma função hash, a qual tem como entrada uma mensagem $\mathbf{m}_i$ de tamanho variável K. Uma das funções do hash é a verificação da integridade de uma mensagem, pois caso haja a modificação de apenas um bit nesta mensagem o valor do hash calculado será totalmente diferente. Segundo (Stallings 2015), as características para uma função hash é a propriedade de mão única, ou seja, não é possível conhecer a mensagem a partir do código hash; e a propriedade livre de colisão, o qual dois objetos diferentes não podem ter o mesmo código hash.

As funções hashes podem ser usadas em diversas aplicações, uma delas é na autenticação de dados, que segundo (Stallings 2015) é um mecanismo ou serviço usado para verificar a integridade de uma mensagem em transmissão ou um dado estático (armazenado em disco). E caracteriza como resumo de mensagem (ou *message digest*) a saída da função hash quando usada para fornecer autenticação de alguma mensagem. Geralmente, o código de autenticação de mensagem ou MAC (*Message Authentication Code*) é utilizado para troca de mensagens entre duas entidades que compartilham de uma mesma chave criptográfica, para que o hash seja enviado criptografado por meio dessa chave. Outra aplicação é em assinatura digital, semelhante a do MAC, porém o código hash da mensagem é encriptado com a chave privada da entidade de origem e decriptado com a chave pública da origem no destino. Hashes também são usados para arquivar senhas em banco de dados, como forma de protegê-las caso o banco de dados seja exposto ou ainda manter a privacidade dos usuários. Também é usado em ferramentas de detecção de intrusão ou antivírus, pois é calculado o hash do arquivo suspeito e analisado com os hashes dos *malwares* do repositório dessas ferramentas, caso sejam iguais o arquivo é tratado como suspeito de ter um software malicioso.

Este trabalho focou em duas funções hashes, o MD5 (*Message Digest 5*) e o SHA-1 (*Secure Hash Algorithm* 1), padronizados respectivamente pelas RFC (*Request For Comment*) 1321 (Rivest 1992) e 3174 (Network Working Group 2001), descritos detalhadamente nas subseções 2.3 e 2.4

2.3 Algoritmo MD5

O MD5 é um algoritmo de *hash* definido pela RFC 1321 (Rivest 1992) que trabalha com mensagens de entrada de comprimento variável. Para cada i-ésima mensagem de entrada (binária), $\mathbf{m}_i$, (de comprimento K_i bits), expressa como

$$\mathbf{m}_i = \left[\begin{array}{cccc} m_0 & m_1 & \dots & m_{K_i-1} \end{array}\right], \, m_k \in \{0,1\} \forall k, \tag{2.1}$$

o algoritmo MD5 gera uma mensagem de saída, $\mathbf{h}_i$, chamada de código *hash*, de tamanho fixo $C = 128$ bits, caracterizado como

$$\mathbf{h}_i = \left[\begin{array}{cccc} h_0 & h_1 & \dots & h_{C-1} \end{array}\right] \text{ onde } h_k \in \{0,1\} \forall k. \tag{2.2}$$

A i-ésima mensagem de entrada, $\mathbf{m}_i$, de K_i bits é estendida através da inserção de duas palavras binárias. A primeira, chamada aqui de $\mathbf{p}_i$, possui P_i bits e é inserida em uma operação chamada de Inserção do Preenchimento. Já a segunda, chamada aqui de $\mathbf{v}_i$, possui T bits e é inserida em uma operação chamada de Inserção do Comprimento. Assim, o cálculo do código *hash*, $\mathbf{h}_i$, para cada i-ésima mensagem de entrada é realizado em uma mensagem estendida, chamada aqui de z_i, que corresponde a uma concatenação da mensagens $\mathbf{m}_i$, $\mathbf{p}_i$ e $\mathbf{v}_i$, ou seja, $\mathbf{z}_i = [\mathbf{m}_i \, \mathbf{p}_i \, \mathbf{v}_i]$. Cada i-ésima mensagem $\mathbf{z}_i$ possui $Z_i = K_i + P_i + T$ bits que podem ser divididos em L_i blocos de comprimento $M = 512$ bits, ou seja,

$$L_i = \frac{Z_i}{M} = \frac{K_i + P_i + T}{512}. \tag{2.3}$$

O pseudo-código apresentado no Algoritmo 1 mostra a sequência de etapas necessárias para geração do código *hash*, estas foram detalhadas nas subseções seguintes.

2.3.1 Inserção do Preenchimento

Esta etapa (ver linhas 2 e 3 do Algoritmo 1) é feita antes do cálculo do código *hash* e tem como função deixar o comprimento da i-ésima mensagem, $\mathbf{m}_i$, divisível por $M = 512$ após a etapa de inserção do comprimento. A mensagem de preenchimento, $\mathbf{p}_i$, associada a i-ésima mensagem de entrada, também chamada de *Padding*, é formada por uma palavra binária de P_i bits no qual o bit mais significativo é 1 e os demais bits são 0. A geração da mensagem de preenchimento é realizada pela função $\textit{GeraçãoPreenchimento}(K_i)$ apresentada na linha 2 do Algoritmo 1. O cálculo do valor de P_i pode ser expresso por

$$P_i = \begin{cases} 448 - (K_i \bmod 512) \\ 512 - (K_i \bmod 512) + 448 \end{cases} \tag{2.4}$$

sendo $(K_i \bmod 512) < 448$ para a primeira equação e $(K_i \bmod 512) \geq 448$ para a segunda, nos quais a operação $a \bmod b$ retorna o resto inteiro da divisão entre a e b.

Algoritmo 1: MD5 para cada i-ésima mensagem $\mathbf{m}_i$

```
1  z_i ← [m_i]
2  p_i ← GeraçãoPreenchimento(K_i)
3  z_i ← [m_i p_i]
4  v_i ← GeraçãoComprimento(K_i)
5  z_i ← [m_i p_i v_i]
6  h_i ← InicializaçãoHash();
7  para j ← 0 até L_i − 1 faça
8      b_j ← DivisãoMensagem(z_i)
9      n ← −1
10     H(n) ← InicializaVariáveisHash()
11     para n ← 0 até 63 faça
12         f(n) ← CálculoFunçãoF(n, B(n), C(n), D(n))
13         g(n) ← CálculoVariávelG(n)
14         H(n) ← AtualizaVariáveisHash(H(n))
15     fim
16     h_i ← AtualizaHash(H(n))
17 fim
```

2.3.2 Inserção do Comprimento

Nesta etapa (linhas 4 e 5 do Algoritmo 1) é adicionada a mensagem $\mathbf{v}_i$ caracterizada por uma palavra binária de $T = 64$ bits e expressa como

$$\mathbf{v}_i = \begin{bmatrix} v_0 & v_1 & \dots & v_{T-1} \end{bmatrix} \text{ onde } v_k \in \{0,1\} \forall k. \tag{2.5}$$

A geração da mensagem de comprimento é realizada pela função $GeraçãoComprimento(K_i)$ apresentada na linha 4 do Algoritmo 1. A mensagem $\mathbf{v}_i$ armazena o valor do comprimento da i-ésima mensagem de entrada $\mathbf{m}_i$, ou seja,

$$\mathbf{v}_i = Binary(K, T) \tag{2.6}$$

onde $Binary(a, b)$ é uma função que retorna um vetor de tamanho b com a representação binária de um número decimal qualquer a em b bits.

A RFC 1321 (Rivest 1992), supõem que o tamanho, K_i, da maioria das mensagens pode ser representado por 64 bits, ou seja, $K_i < 2^T$. Para os casos em que $K_i \geq 2^T$ são utilizados os 64 bits menos significativos da representação binária de K_i.

Finalmente, ao final da segunda etapa a menagem, $\mathbf{z}_i$, que é uma extensão da i-ésima mensagem original de entrada $\mathbf{m}_i$, é formada (linha 5 do Algoritmo 1).

2.3.3 Inicialização do Código Hash

A inicialização do código *hash* (ver linha 6 do Algoritmo 1) é padronizada na RFC 1321 (Rivest 1992) de acordo com as seguintes expressões:

$$\mathbf{ha} = \begin{bmatrix} h_0 & \dots & h_{31} \end{bmatrix} = Binary(1732584193, 32), \tag{2.7}$$

$$\mathbf{hb} = \begin{bmatrix} h_{32} & \dots & h_{63} \end{bmatrix} = Binary(4023233417, 32), \tag{2.8}$$

$$\mathbf{hc} = \begin{bmatrix} h_{64} & \dots & h_{95} \end{bmatrix} = Binary(2562383102, 32), \tag{2.9}$$

e

$$\mathbf{hd} = \begin{bmatrix} h_{96} & \dots & h_{128} \end{bmatrix} = Binary(0271733878, 32), \tag{2.10}$$

onde

$$\mathbf{h}_i = \begin{bmatrix} \mathbf{ha} & \mathbf{hb} & \mathbf{hc} & \mathbf{hd} \end{bmatrix}. \tag{2.11}$$

2.3.4 Divisão da Mensagem

Nesta etapa, linha 8 do Algoritmo 1, a mensagem $\mathbf{z}_i$ é quebrada em L_i blocos de $M = 512$ bits, ou seja,

$$\mathbf{z}_i = \begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \dots & \mathbf{b}_{L_i - 1} \end{bmatrix}, \tag{2.12}$$

onde cada j-ésimo bloco associado a i-ésima mensagem é expresso como

$$\mathbf{b}_j = \begin{bmatrix} \mathbf{u}_j[0] & \mathbf{u}_j[1] & \dots & \mathbf{u}_j[15] \end{bmatrix}, \tag{2.13}$$

onde $\mathbf{u}_j[k]$ é uma mensagem de 32 bits, ou seja,

$$\mathbf{u}_j[k] = \begin{bmatrix} u_j[k,0] & u_j[k,1] & \dots & u_j[k,31] \end{bmatrix} \tag{2.14}$$

onde $u_j[k,l] \in \{0,1\} \forall l$.

2.3.5 Inicialização das Variáveis Hash $\mathbf{H}(n)$

O núcleo do algoritmo MD5 é formado por quatro variáveis de 32 bits, chamadas de $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$, que atualizam seus valores durante as n iterações do algoritmo. Estas variáveis são caracterizadas neste trabalho como vetores de 32 bits que juntando-as compõem um vetor $\mathbf{H}(n)$ de 128 posições caracterizado como

$$\mathbf{H}(n) = \begin{bmatrix} \mathbf{A}(n) & \mathbf{B}(n) & \mathbf{C}(n) & \mathbf{D}(n) \end{bmatrix}. \tag{2.15}$$

A inicialização destas variáveis, no instante $n = -1$, (ver linha 10 do Algoritmo 1) é padronizada na RFC 1321 (Rivest 1992) com os seguintes valores em decimal:

$$\mathbf{A}(-1) = Binary(1732584193, 32), \tag{2.16}$$

$$\mathbf{B}(-1) = Binary(4023233417, 32), \tag{2.17}$$

$$\mathbf{C}(-1) = Binary(2562383102, 32), \quad (2.18)$$

e

$$\mathbf{D}(-1) = Binary(0271733878, 32). \quad (2.19)$$

2.3.6 Cálculo da Função $f(\cdot)$

Em cada n-ésima iteração de cada j-ésimo bloco, $\mathbf{b}_j(n)$ é calculada uma função não linear, $f(\cdot)$, a partir das informações das variáveis *hash* $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$. A saída da função $f(\cdot)$ é armazenada no vetor $\mathbf{f}(n)$ (linha 13 do Algoritmo 1), expresso como

$$\mathbf{f}(n) = f(n, \mathbf{B}, \mathbf{C}, \mathbf{D}) = \begin{cases} \alpha(n) \text{ para } n = 0 \ldots 15 \\ \beta(n) \text{ para } n = 16 \ldots 31 \\ \gamma(n) \text{ para } n = 32 \ldots 47 \\ \delta(n) \text{ para } n = 48 \ldots 63 \end{cases}, \quad (2.20)$$

onde

$$\alpha(n) = (\mathbf{B}(n-1) \wedge \mathbf{C}(n-1)) \vee (\neg \mathbf{B}(n-1) \wedge \mathbf{D}(n-1)), \quad (2.21)$$

$$\beta(n) = (\mathbf{D}(n-1) \wedge \mathbf{B}(n-1)) \vee (\neg \mathbf{D}(n-1) \wedge \mathbf{C}(n-1)), \quad (2.22)$$

$$\gamma(n) = \mathbf{B}(n-1) \oplus \mathbf{C}(n-1) \oplus \mathbf{D}(n-1), \quad (2.23)$$

e

$$\delta(n) = \mathbf{C}(n-1) \oplus (\mathbf{B}(n-1) \vee \neg \mathbf{D}(n-1)), \quad (2.24)$$

onde $\neg$, $\wedge$, $\vee$ e $\oplus$ são operações: negação, E, OU e OU EXCLUSIVO elemento a elemento (bit a bit), respectivamente.

Juntamente ao cálculo de $\mathbf{f}(n)$ também é calculada uma variável chamada aqui de $g(n)$ que representa uma posição dentro do j-ésimo bloco em processamento (ver linha 13 do Algoritmo 1), segundo (Rivest 1992). O cálculo dessa variável é efetuado em cada n-ésima iteração de cada j-ésimo bloco $\mathbf{b}_j(n)$ e pode ser expresso como

$$g(n) = \begin{cases} n & \text{para } n = 0 \ldots 15 \\ (5 \times n + 1) \bmod 16 & \text{para } n = 16 \ldots 31 \\ (3 \times n + 5) \bmod 16 & \text{para } n = 32 \ldots 47 \\ (7 \times n) \bmod 16 & \text{para } n = 48 \ldots 63 \end{cases}. \quad (2.25)$$

2.3.7 Atualização das Variáveis Hash

Também em cada n-ésima iteração de cada j-ésimo bloco $\mathbf{b}_j(n)$ é atualizado o valor das variáveis, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$ após o cálculo de $\mathbf{f}(n)$ e $g(n)$ (ver linha 14 do Algoritmo 1). A atualização destas variáveis é representada pelas seguintes equações:

$$\mathbf{A}(n) = \mathbf{D}(n-1), \quad (2.26)$$

$$\mathbf{D}(n) = \mathbf{C}(n-1), \quad (2.27)$$

$$\mathbf{C}(n) = \mathbf{B}(n-1) \quad (2.28)$$

e

$$\mathbf{B}(n) = \mathbf{B}(n-1) + \mathbf{lr}(n) \tag{2.29}$$

no qual,

$$\mathbf{lr}(n) = \text{leftrotate}(\mathbf{V}(n) + \mathbf{Z}(n), \mathbf{s}[n]), \tag{2.30}$$

$$\mathbf{V}(n) = \mathbf{A}(n-1) + \mathbf{f}(n) \tag{2.31}$$

e

$$\mathbf{Z}(n) = \mathbf{q}(n) + \mathbf{u}_{i,j}[g(n)]. \tag{2.32}$$

O vetor **s** é um vetor de 64 constantes de 32 bits definidos em (Rivest 1992) como

$$\begin{aligned} \mathbf{s}[0..15] &= [7,12,17,22,7,12,17,22,7,12,17,22,7,12,17,22] \\ \mathbf{s}[16..31] &= [5,9,14,20,5,9,14,20,5,9,14,20,5,9,14,20] \\ \mathbf{s}[32..47] &= [4,11,16,23,4,11,16,23,4,11,16,23,4,11,16,23] \\ \mathbf{s}[48..63] &= [6,10,15,21,6,10,15,21,6,10,15,21,6,10,15,21] \end{aligned} \tag{2.33}$$

$\mathbf{q}(n)$ é um vetor de 32 bits definido como

$$\mathbf{q}(n) = \lfloor 2^{32} \times \mid \sin(n+1) \mid \rfloor \tag{2.34}$$

e

$$\text{leftrotate}(\mathbf{r}, \mathbf{y}) = (\mathbf{r} \ll \mathbf{y}) \vee (\mathbf{r} \gg (32 - \mathbf{y})), \tag{2.35}$$

onde $\ll$ e $\gg$ são operadores de deslocamento bit a bit a esquerda e a direita, respectivamente.

2.3.8 Atualização do código *hash*

Para cada j-ésimo bloco, $\mathbf{b}_j$, o MD5 executa 64 iterações e ao final de cada j-ésimo bloco o código *hash* é atualizado de forma linear seguindo as seguintes expressões:

$$\mathbf{ha} = \mathbf{ha} + \mathbf{A}(63), \tag{2.36}$$

$$\mathbf{hb} = \mathbf{hb} + \mathbf{B}(63), \tag{2.37}$$

$$\mathbf{hc} = \mathbf{hc} + \mathbf{C}(63) \tag{2.38}$$

e

$$\mathbf{hd} = \mathbf{hd} + \mathbf{D}(63). \tag{2.39}$$

Assim para cada i-ésima mensagem, $\mathbf{m}_i$, o valor do código *hash* associado, $\mathbf{h}_i$, é encontrado em

$$N_i = L_i \times 64 \tag{2.40}$$

iterações, onde N_i é definido neste trabalho como o número total de interações para o cálculo do *hash* associado a uma mensagem $\mathbf{m}_i$.

2.4 Algoritmo SHA-1

Já o SHA-1 é descrito pela FIPS (*Federal Information Processing Standards Publication*) 180-4 (NIST 2015) e pela RFC 3174 (Network Working Group 2001). Semelhante ao MD5, o SHA-1 também opera com mensagens de entrada de comprimento variável e para cada i-ésima mensagem de entrada, $\mathbf{m}_i$, (de comprimento K_i bits), conforme a Equação 2.1. Desse mesmo modo, gera uma mensagem de saída, $\mathbf{h}_i$, código *hash* com tamanho fixo em $C = 160$ *bits*, caracterizado da mesma maneira como representado pela Equação 2.2.

Análogo ao MD5, o algortimo SHA-1 opera com uma concatenação de duas palavras binárias em uma i-ésima mensagem de entrada, $\mathbf{m}_i$, de K_i bits, chamada de z_i, a qual o código hash, $\mathbf{h}_i$, será calculado. De igual modo, na operação de Inserção do Preenchimento (ou *Append Padding*) é concatenado P_i bits de $\mathbf{p}_i$. E T bits, por meio de $\mathbf{v}_i$, na operação de Inserção do Comprimento (ou *Append Length*). Assim, z_i, também corresponde a uma concatenação das mensagens $\mathbf{m}_i$, $\mathbf{p}_i$ e $\mathbf{v}_i$, ou seja, $\mathbf{z}_i = [\mathbf{m}_i\,\mathbf{p}_i\,\mathbf{v}_i]$. Como expressado pela Equação 2.3, cada i-ésima mensagem $\mathbf{z}_i$ possui $Z_i = K_i + P_i + T$ bits que podem ser divididos em L_i blocos de comprimento $M = 512$ bits.

No Algoritmo 2 esta apresentado o pseudo-código com a sequência das etapas para geração do código *hash*, conforme o (NIST 2015), as quais estão detalhadas nas subseções seguintes. Neste algoritmo é possível perceber as semelhanças com o MD5, descrito no Algortimo 1, entre as linhas 1 e 6. Apesar que na linha 6 a inicialização possui uma palavra de 32bits a mais, conforme descrito na subseção 2.4.3.

Algoritmo 2: SHA-1 para cada i-ésima mensagem $\mathbf{W}_i$

1. $\mathbf{z}_i \leftarrow [\mathbf{m}_i]$
2. $\mathbf{p}_i \leftarrow \mathit{GeraçãoPreenchimento}(K_i)$
3. $\mathbf{z}_i \leftarrow [\mathbf{m}_i\,\mathbf{p}_i]$
4. $\mathbf{v}_i \leftarrow \mathit{GeraçãoComprimento}(K_i)$
5. $\mathbf{z}_i \leftarrow [\mathbf{m}_i\,\mathbf{p}_i\,\mathbf{v}_i]$
6. $\mathbf{h_i} \leftarrow \mathit{InicializaçãoHash}()$;
7. **para** $j \leftarrow 0$ **até** $L_i - 1$ **faça**
8. $\quad \mathbf{b}_j \leftarrow \mathit{DivisãoMensagem}(\mathbf{z}_i)$
9. $\quad n \leftarrow -1$
10. $\quad \mathbf{H}(n) \leftarrow \mathit{InicializaVariáveisHash}()$
11. $\quad$ **para** $n \leftarrow 0$ **até** 79 **faça**
12. $\quad\quad \mathbf{w}(n) \leftarrow \mathit{CálculoFunçãoW}(n, \mathbf{b}_j)$
13. $\quad\quad \mathbf{f}(n) \leftarrow \mathit{CálculoFunçãoF}(n, \mathbf{B}(n), \mathbf{C}(n), \mathbf{D}(n))$
14. $\quad\quad \mathbf{H}(n) \leftarrow \mathit{AtualizaVariáveisHash}(\mathbf{H}(n))$
15. $\quad$ **fim**
16. $\quad \mathbf{h}_i \leftarrow \mathit{AtualizaHash}(\mathbf{H}(n))$
17. **fim**

2.4.1 Inserção do Preenchimento

Esta etapa (ver linhas 2 e 3 do Algoritmo 2) é feita antes do cálculo do código *hash* e da mesma forma do Algoritmo MD5, descrito na subseção em 2.3.1, tem como função deixar o comprimento da i-ésima mensagem, $\mathbf{m}_i$, após a etapa de inserção do comprimento com um valor divisível por $M = 512$. A mensagem de preenchimento, $\mathbf{p}_i$, associada a i-ésima mensagem de entrada, também chamada de *Padding*, é formada por uma palavra binária de P_i bits no qual o bit mais significativo é 1 e os demais bits são 0. A geração da mensagem de preenchimento é realizada pela função $GeraçãoPreenchimento(K_i)$ apresentada na linha 2 do Algoritmo 2. O cálculo do valor de P_i pode ser expresso por

$$P_i = \begin{cases} 448 - (K_i \bmod 512) & \text{para } (K_i \bmod 512) < 448 \\ 512 - (K_i \bmod 512) + 448 & \text{para } (K_i \bmod 512) \geq 448 \end{cases}, \tag{2.41}$$

onde a operação $a \bmod b$ retorna o resto inteiro da divisão entre a e b. Assim, $\mathbf{p}_i$ pode ser expresso como

$$\mathbf{p}_i = \begin{bmatrix} p_0 & p_1 & \dots & p_{P_i-1} \end{bmatrix}, \tag{2.42}$$

onde, $p_0 = 1$ e $p_i = 0$ para $i = 1 \dots P_i - 1$.

2.4.2 Inserção do Comprimento

Esta etapa adiciona a mensagem $\mathbf{v}_i$, definida por uma palavra binária de $T = 64$ bits, conforme as linhas 4 e 5 do Algoritmo 2 e representada pela Equação 2.5.

O resultado da função $GeraçãoComprimento(K_i)$ apresentada na linha 4 do Algoritmo 2 é a mensagem utilizada nesta etapa, definida pela Equação 2.6. Para este passo a FIPS 180-4 (NIST 2015), supõe que o tamanho, K_i, da maioria das mensagens pode ser representado por 64 bits, ou seja, $K_i < 2^T$.

Finalmente, ao final da segunda etapa a menagem, $\mathbf{z}_i$, que é uma extensão da i-ésima mensagem original de entrada $\mathbf{m}_i$, é formada (linha 5 do Algoritmo 2). Neste trabalho a mensagem $\mathbf{z}_i$ é caracterizada por um vetor de Z_i bits expresso como

$$\mathbf{z}_i = \begin{bmatrix} z_0 & z_1 & \dots & z_{Z-1} \end{bmatrix} \text{ onde } z_k \in \{0,1\} \forall k. \tag{2.43}$$

2.4.3 Inicialização do Código Hash

A inicialização do código *hash* (ver linha 6 do Algoritmo 2) é padronizada pela FIPS 180-4 (NIST 2015) de acordo com as seguintes expressões:

$$\mathbf{ha} = \begin{bmatrix} h_0 & \dots & h_{31} \end{bmatrix} = Binary(1732584193, 32), \tag{2.44}$$

$$\mathbf{hb} = \begin{bmatrix} h_{32} & \dots & h_{63} \end{bmatrix} = Binary(4023233417, 32), \tag{2.45}$$

$$\mathbf{hc} = \begin{bmatrix} h_{64} & \dots & h_{95} \end{bmatrix} = Binary(2562383102, 32), \tag{2.46}$$

$$\mathbf{hd} = \begin{bmatrix} h_{96} & \dots & h_{127} \end{bmatrix} = Binary(0271733878, 32), \tag{2.47}$$

e

$$\mathbf{he} = \left[\begin{array}{ccc} h_{128} & \dots & h_{159} \end{array} \right] = Binary(3285377520, 32), \tag{2.48}$$

onde

$$\mathbf{h}_i = \left[\begin{array}{ccccc} \mathbf{ha} & \mathbf{hb} & \mathbf{hc} & \mathbf{hd} & \mathbf{he} \end{array} \right]. \tag{2.49}$$

2.4.4 Divisão da Mensagem

Nesta etapa, linha 8 do Algoritmo 2, a mensagem $\mathbf{z}_i$ é quebrada em L_i blocos de $M = 512$ bits, ou seja,

$$\mathbf{z}_i = \left[\begin{array}{cccc} \mathbf{b}_0 & \mathbf{b}_1 & \dots & \mathbf{b}_{L_i-1} \end{array} \right], \tag{2.50}$$

onde cada j-ésimo bloco associado a i-ésima mensagem é expresso como

$$\mathbf{b}_j = \left[\begin{array}{cccc} b_{j,0} & b_{j,1} & \dots & b_{j,M-1} \end{array} \right] \text{ onde } b_{j,k} \in \{0,1\} \forall k. \tag{2.51}$$

O j-ésimo bloco, $\mathbf{b}_j$, também pode representado como

$$\mathbf{b}_j = \left[\begin{array}{cccc} \mathbf{u}_j[0] & \mathbf{u}_j[1] & \dots & \mathbf{u}_j[15] \end{array} \right], \tag{2.52}$$

onde $\mathbf{u}_j[k]$ é uma mensagem de 32 bits, ou seja,

$$\mathbf{u}_j[k] = \left[\begin{array}{cccc} u_j[k,0] & u_j[k,1] & \dots & u_j[k,31] \end{array} \right] \tag{2.53}$$

onde $u_j[k,l] \in \{0,1\} \forall l$.

2.4.5 Inicialização das Variáveis Hash $\mathbf{H}(n)$

O algoritmo SHA-1 possui cinco variáveis de 32 bits, chamadas de $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$ que se atualizam durante as iterações do algoritmo. Estas variáveis são especificadas neste trabalho como vetores categorizados como

$$\mathbf{X}(n) = \left[\begin{array}{cccc} x_0 & x_1 & \dots & x_{31} \end{array} \right] \text{ no qual } x_k \in \{0,1\} \forall k, \tag{2.54}$$

onde, $\mathbf{X}(n) \in \{\mathbf{A}(n), \mathbf{B}(n), \mathbf{C}(n), \mathbf{D}(n), \mathbf{E}(n)\}$. A junção destas cinco variáveis compõem um vetor de 160 posições caracterizado como

$$\mathbf{H}(n) = \left[\begin{array}{ccccc} \mathbf{A}(n) & \mathbf{B}(n) & \mathbf{C}(n) & \mathbf{D}(n) & \mathbf{E}(n) \end{array} \right]. \tag{2.55}$$

A inicialização destas variáveis, no instante $n = -1$, (ver linha 10 do Algoritmo 2) de acordo com a FIPS 180-4 (NIST 2015) ocorre com o recebimento dos mesmos valores que iniciam o *hash* $\mathbf{h}_i$, logo $\mathbf{A}(-1) = \mathbf{ha}$, $\mathbf{B}(-1) = \mathbf{hb}$, $\mathbf{C}(-1) = \mathbf{hc}$, $\mathbf{D}(-1) = \mathbf{hd}$ e $\mathbf{E}(-1) = \mathbf{he}$.

2.4.6 Cálculo da variável $\mathbf{w}(n)$

No SHA-1 são necessárias 80 iterações para uma saída válida, $\mathbf{h}_i$, associada a cada i-ésima mensagem (ver linha 11 do Algoritmo 2). Em cada n-ésima iteração é calculada uma variável, $\mathbf{w}(n)$ expressa como

$$\mathbf{w}(n) = \begin{cases} \mathbf{u}_j[n] & \text{para } 0 \leq n \leq 15 \\ \mathbf{sw}[n] & \text{para } 16 \leq n \leq 79 \end{cases}, \tag{2.56}$$

onde

$$\mathbf{sw}[n] = \mathrm{lr}\left(\mathbf{u}_j[n-3] \oplus \mathbf{u}_j[n-8] \oplus \mathbf{u}_j[n-14] \oplus \mathbf{u}_j[n-16], 1\right) \tag{2.57}$$

onde $\oplus$ é a operação ou exclusivo bit a bit e $\mathrm{lr}(\mathbf{r},\mathbf{s})$ representa a função *leftrotate* que é expressa como

$$\mathrm{lr}(\mathbf{r},\mathbf{s}) = (\mathbf{r} \ll \mathbf{s}) \vee (\mathbf{r} \gg (32-\mathbf{s})), \tag{2.58}$$

onde $\vee$, $\ll$ e $\gg$ são os operadores OU e de deslocamento bit a bit a esquerda e a direita, respectivamente.

2.4.7 Cálculo da Função $f(\cdot)$

Em cada n-ésima iteração de cada j-ésimo bloco, $\mathbf{b}_j(n)$ é calculada uma função não linear, $f(\cdot)$, a partir das informações das variáveis *hash* $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$. A saída da função, $f(\cdot)$ é armazenada no vetor $\mathbf{f}(n)$ (linha 13 do Algoritmo 2), expresso como

$$\mathbf{f}(n) = f(n,\mathbf{B},\mathbf{C},\mathbf{D}) = \begin{cases} \alpha(n) \text{ para } n = 0\ldots 19 \\ \beta(n) \text{ para } n = 20\ldots 39 \\ \gamma(n) \text{ para } n = 40\ldots 59 \\ \delta(n) \text{ para } n = 60\ldots 79 \end{cases}, \tag{2.59}$$

onde

$$\alpha(n) = (\mathbf{B}(n-1) \wedge \mathbf{C}(n-1)) \vee (\neg\mathbf{B}(n-1) \wedge \mathbf{D}(n-1)), \tag{2.60}$$

$$\beta(n) = \mathbf{B}(n-1) \oplus \mathbf{C}(n-1) \oplus \mathbf{D}(n-1), \tag{2.61}$$

$$\gamma(n) = (\mathbf{B}(n-1) \wedge \mathbf{C}(n-1)) \vee (\mathbf{B}(n-1) \wedge \mathbf{D}(n-1)) \vee (C(n-1) \wedge D(n-1)) \tag{2.62}$$

e

$$\delta(n) = \mathbf{B}(n-1) \oplus \mathbf{C}(n-1) \oplus \mathbf{D}(n-1), \tag{2.63}$$

onde $\neg$ e $\wedge$ são operações de negação e E bit a bit, respectivamente.

2.4.8 Atualização das Variáveis Hash

O valor das variáveis, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$ é atualizado após o cálculo de $\mathbf{f}(n)$ em cada n-ésima iteração de cada j-ésimo bloco $\mathbf{b}_j(n)$ (ver linha 14 do Algoritmo 2). A atualização dessas variáveis é representada pelas seguintes equações:

$$\mathbf{E}(n) = \mathbf{D}(n-1), \tag{2.64}$$

$$\mathbf{D}(n) = \mathbf{C}(n-1), \tag{2.65}$$

$$\mathbf{C}(n) = \mathrm{lr}(\mathbf{B}(n-1), 30), \tag{2.66}$$

$$\mathbf{B}(n) = \mathbf{A}(n-1) \tag{2.67}$$

e

$$\mathbf{A}(n) = \mathbf{V}(n) + \mathbf{Z}(n) + \mathrm{lr}(A(n-1), 5), \tag{2.68}$$

no qual,

$$\mathbf{Z}(n) = \mathbf{W}(n) + \mathbf{E}(n-1) \tag{2.69}$$

e

$$\mathbf{V}(n) = \mathbf{f}(n) + \mathbf{k}(n). \tag{2.70}$$

O SHA-1 também possui quatro constantes $\mathbf{k}(n)$ de 32 bits, as quais são usadas na n-ésima iteração de cada j-ésimo bloco $\mathbf{b}_j(n)$, conforme especificado por

$$K(n) = \begin{cases} 1518500249 & \text{para } n = 0 \ldots 19 \\ 1859775393 & \text{para } n = 20 \ldots 39 \\ 2400959708 & \text{para } n = 40 \ldots 59 \\ 3395469782 & \text{para } n = 60 \ldots 79 \end{cases}. \tag{2.71}$$

2.4.9 Atualização do código *hash*

Para cada j-ésimo bloco, $\mathbf{b}_j$, o SHA executa 80 iterações e ao final de cada j-ésimo bloco o código *hash* é atualizado de forma linear seguindo as seguintes expressões:

$$\mathbf{ha} = \mathbf{ha} + \mathbf{A}(79), \tag{2.72}$$

$$\mathbf{hb} = \mathbf{hb} + \mathbf{B}(79), \tag{2.73}$$

$$\mathbf{hc} = \mathbf{hc} + \mathbf{C}(79), \tag{2.74}$$

$$\mathbf{hd} = \mathbf{hd} + \mathbf{D}(79), \tag{2.75}$$

e

$$\mathbf{he} = \mathbf{he} + \mathbf{E}(79). \tag{2.76}$$

Assim para cada i-ésima mensagem, $\mathbf{m}_i$, o valor do código *hash* associado, $\mathbf{h}_i$, é encontrado em

$$N_i = L_i \times 80 \tag{2.77}$$

iterações, onde N_i é definido neste trabalho como o número total de interações para o cálculo do *hash* associado a uma mensagem $\mathbf{m}_i$.

Com isso, conclui-se o capítulo com a descrição básica sobre segurança da informação e as principais propriedades envolvendo o assunto. Além do processo detalhado de cada etapa para criação dos algoritmos MD5 e SHA-1, os quais utilizam operações matemáticas com funções não-lineares, operações lógicas e funções. Assim como, a semelhança nas primeiras linhas (1 a 6) dos algoritmos 1 e 2. No próximo capítulo, 3, será possível conhecer sobre plataformas de hardware reconfigurável e sobre a FPGA.

Capítulo 3

Computação Reconfigurável

Neste capítulo é apresentado o conceito de computação reconfigurável, abordando as principais definições e composições. Os FPGAs (*Field Programmable Gate Arrays*) ou Arranjo de Portas Programáveis em Campo estão descritos em maior detalhe por ser a plataforma de estudo para implementação dos algoritmos de hash descritos nesta dissertação.

3.1 Dispositivos Lógicos Programáveis

Com a diminuição dos transistores, o desenvolvimento de circuitos integrados (CI) e o domínio mais aprofundado do silício foi possível a criação de dispositivos menores e a possibilidade desses CIs serem alterados na lógica após a confecção, tornando-o de lógica programável. Esses que precisam do hardware em si e de um software para o desenvolvimento (podendo ter a implementação de uma ou mais funções) são capazes de serem especificados pelo usuário ou pelo fabricante. Diferentemente dos dispositivos de lógica fixa, os quais, algumas vezes, ocupam mais espaço e possuem uma dificuldade maior para efetuar mudanças, caso exista a necessidade de alterar ou mesmo atualizar a lógica de um dispositivo. Tornando a única mudança possível por meio de alterações no hardware da solução, trocando-o por outro. Além disso, um projeto lógico geralmente pode ser implementado mais rápido e com um menor custo com a lógica programável do que com CIs de função fixa (Floyd 2007).

Tal como os ASICs (*Application-Specific Integrated Circuit*) - dispositivos de lógica fixa - chips programados uma única vez para executar uma ou mais funções ou algoritmos. Porém, como mencionado, existe certa inflexibilidade em relação as alterações nessas funções ou algoritmos. Caso seja preciso alterar algo após o término de fabricação do ASIC, será necessário a construção de um novo chip e a troca nos mais diversos dispositivos aos quais fora instalado esse ASIC (Compton & Hauck 2002). Entretanto, para tratar tais situações que existem os dispositivos de lógicas programáveis ou a computação reconfigurável, os quais podem ser divididos em duas categorias principais: PLD (*Programmable Logic Device*), subdividido em SPLD e CPLD; e FPGA (*Field Programmable Gate Array*).

Segundo (Floyd 2007), Dispositivos de Lógica Programável Simples (SPLD), são caracterizados por um arranjo programável de portas AND e um arranjo fixo de portas OR

com saídas programáveis. Podem ainda ser categorizados como PAL (*Programmable Array of Logic)*, lógica que pode ser gravada uma única vez, ou GAL (*Generic Array Logic*), semelhante a um dispositivo PAL, entretanto pode ser reprogramado várias vezes. Já os Dispositivo de Lógica Programável Complexo (CPLD) consistem de múltiplos SPLDs, compostos por blocos de arranjo lógico, do inglês *Logic Array Blocks* (LABs) e um arranjo de interconexões programáveis, do inglês *Programmable Interconnction Array* (PIA). Na próxima subseção (3.2) é abordado com mais detalhes os conceitos de FPGA.

3.2 FPGA

A computação reconfigurável destina-se a preencher o espaço entre o hardware e o software, atingindo desempenhos substancialmente maiores do que implementações de algoritmos ou funções em software e sendo capaz de manter um nível de flexibilidade maior do que as implementações puramente em hardware. Dispositivos reconfiguráveis, como os FPGAs, possuem uma série de elementos computacionais cuja funcionalidade é determinada através de vários bits de configuração programáveis (Compton & Hauck 2002). Os FPGAs possuem elementos internos semelhantes aos LABs do CPLDS, porém com uma lógica menos complexa e são bem mais numerosos (Floyd 2007). Já (ALTERA 2015) trata o LAB (*Logic Array Block*) como a composição de blocos de construção básicos, conhecidos como módulos de lógica adaptativa (ALMs), os quais são possíveis configurar para implementar funções de lógica, funções aritméticas e funções de registro. (Floyd 2007) descreve que o FPGA possui três elementos básicos: os blocos lógicos (do inglês *Logic Blocks* ou CLBs), as interconexões programáveis (*Programmable Interconnects*) e os blocos de entrada/saída (I/O – *in/out – Blocks*), conforme demonstrado pela Figura 3.1. Alguns modelos de FPGAs podem conter certos componentes adicionais, o mais comum de encontrar é o Processador Digital de Sinais ou DSP (*Digital Signal Processor*) que são blocos específicos para efetuar operações de multiplicação, o que pode diminuir a quantidade de utilização de LUTs (*Look up Tables* - tipo de memória programável usada para gerar funções lógicas combinacionais de soma-de-produtos) em uma implementação (Noronha 2017).

Os blocos lógicos são formados por um conjunto de células lógicas (LC ou *Logic Cell*), cada LC contem, normalmente, tabelas de busca ou LUTs de 4 a 8 entradas, somadores completos (*Full Adder*), registradores e multiplexadores (Floyd 2007). Um exemplo dessa construção interna com tais dispositivos pode ser visto na Figura 3.2, a qual representa o Diagrama de bloco de um módulo da arquitetura adaptativa da FPGA Cyclone V da Intel (anteriormente da Altera) (ALTERA 2015).

As matrizes de interconexões programáveis provêm as interconexões de blocos lógicos e as conexões para as entradas e saídas. Já os blocos de entrada/saída (I/O Blocks) permitem a conexão das estruturas de entrada, saída e bidirecional do FPGA (selecionadas individualmente) com interconexões para outros dispositivos, como placas de rede, interfaces USB e JTAG, LEDs, chaves seletoras, entre outros.

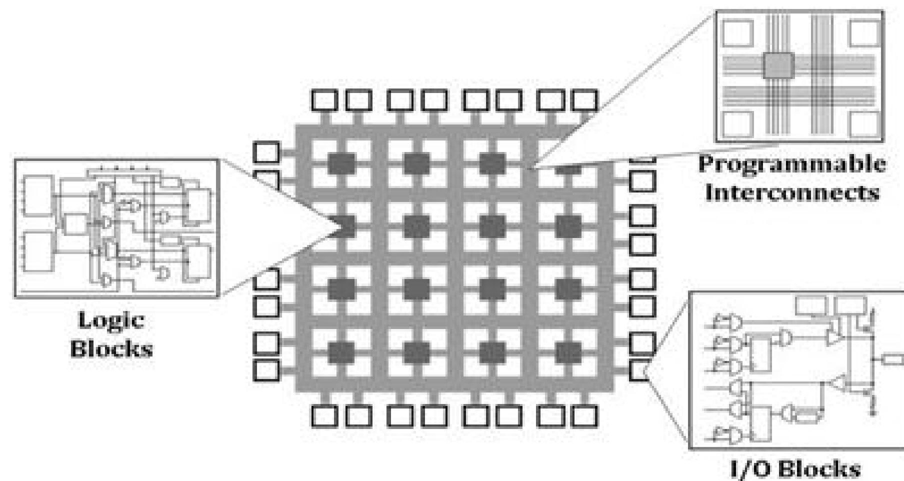

Figura 3.1: Arquitetura interna do FPGA

Fonte: Próprio autor

3.2.1 Implementação de circuitos em FPGA

Para implementar um algoritmo em FPGA é preciso produzir um circuito lógico utilizando os componentes descritos na arquitetura interna do FPGA. Para isso, deve-se seguir o processo de criação de fluxo de dados em formato binário (*bitstream*), conforme Figura 3.3. Por meio desse fluxo é possível fazer as interligações das matrizes de interconexões, CLBs (inclusive internamente com as LUTs, flip-flops, somadores e registradores) e blocos de entrada/saída, por meio das linguagens de descrição de hardware (*Hardware Description Language* - HDL), como VHDL e Verilog. Há também as ferramentas gráficas de projeto, o System Generator e o DSP Builder são exemplos dessas, sendo a primeira pertencente a fabricante Xilinx e a segunda a Intel. Nesses tipos de ferramentas gráficas os blocos básicos são descritos internamente pelo programa utilizando uma HDL. Além do mais, ambos tipos de implementações, tanto a utilizando HDL quanto as ferramentas gráficas podem fazer proveito de componentes de alto nível, incluindo os de propriedade intelectual (*Intellectual Property* - IP Blocks) (Noronha 2017). Para a execução da proposta deste trabalho foi utilizado a ferramenta System Generator, criada para o desenvolvimento rápido de soluções paralelas em FPGA, com suporte a simulação e co-simulação, que pode gerar códigos automáticos para o Simulink [1] e o MATLAB [2], o qual faz a integração com outros componentes em RTL, com IPs e MATLAB, e faz parte do Vivado System Edition Design Suite [3] (XILINX 2018). Importante observar que tanto o System Generator quanto o DSP Builder não são ferramentas de HLS (High Level

[1]desenvolvido pela MathWorks, é um ambiente de programação gráfica para modelagem, simulação e análise de sistemas dinâmicos de múltiplos domínios

[2]um ambiente computacional numérico multi-paradigmático, que possui uma linguagem de programação proprietária desenvolvida pela MathWorks, o qual permite manipulações de matriz, plotagem de funções e dados, e outras funções.

[3]um pacote de software produzido pela Xilinx para síntese e análise de projetos em HDL

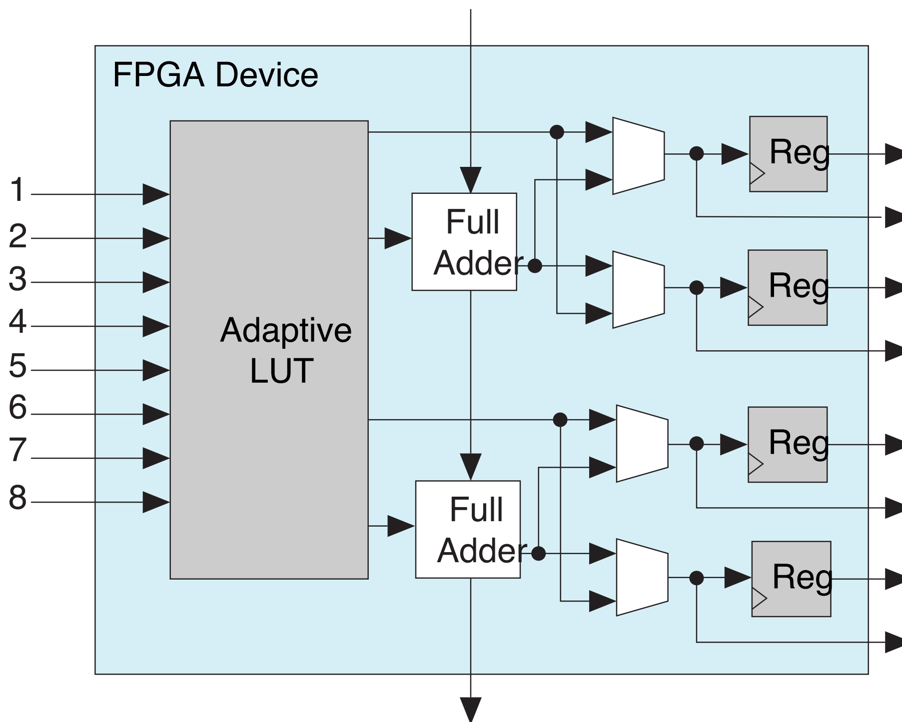

Figura 3.2: Arquitetura interna de um CLB da Intel Cyclone V.

Fonte: Próprio autor

Synthesis), como a MyHDL [4], que converte códigos escritos em Python em códigos HDL e as bibliotecas Python para a PYNQ-Z1 [5], que utiliza o Jupyter Notebook (um ambiente de computação interativo baseado em navegador web) para criação e interação com os *overlays*, bibliotecas para hardware. Também a ferramenta Vivado - High-Level Synthesis (C based), que transforma códigos escritos em C em RTL e sintetiza-os no FPGA da Xilinx. Com essa ferramenta pode-se ainda utilizar as linguagens C++, SystemC ou o OpenCL (*Open Computing Language*).

Como exemplo de utilização e de perspectiva de implementação de circuitos em FPGA que a Intel tem investido em estruturas híbridas de processamento para aumentar o desempenho dos clientes atuais por meio de ARM-FGPA e Xeon (CPU) com FPGA, como pode ser visto por meio das placas DE0-Nano-SoC, a qual utiliza um processador ARM (chamado de HPS - *Hardware Processor System*) e uma FPGA Cyclone V. E a Pilha de Aceleração (*Acceleration Stack*) para processador Intel Xeon com FPGAs que é uma coleção robusta de software, firmware e ferramentas projetadas e distribuídas pela Intel para facilitar o desenvolvimento e implantação de FPGAs na otimização da carga de trabalho em data centers, fornecendo interfaces de hardware (APIs) aprimoradas e simplificadas,

[4]disponível em http://www.myhdl.org

[5]um kit de desenvolvimento de código aberto da Xilinx que facilita o projeto de sistemas embarcados com todos os sistemas programáveis em chips da Xilinx Zynq (APSoCs).

a fim de trazer um desenvolvimento mais rápido. Com isto, a Intel desenvolveu uma única placa (de aceleração programável) com o processador Xeon e o FPGA Arria 10GX (INTEL 2018).

O System Generator, ferramenta utilizada nesta dissertação, não é um HLS, mas proporciona que se desenvolva projetos em RTL permitindo uma maior otimização no desenvolvimento. Por meio dela foi possível desenvolver todo o fluxo de dados para implementação dosos circuitos no FPGA alvo, conforme apresentado na Figura 3.3. No qual traz como primeiro passo a etapa de Síntese, que converte um código comportamental em portas lógicas (da Silva 2016).

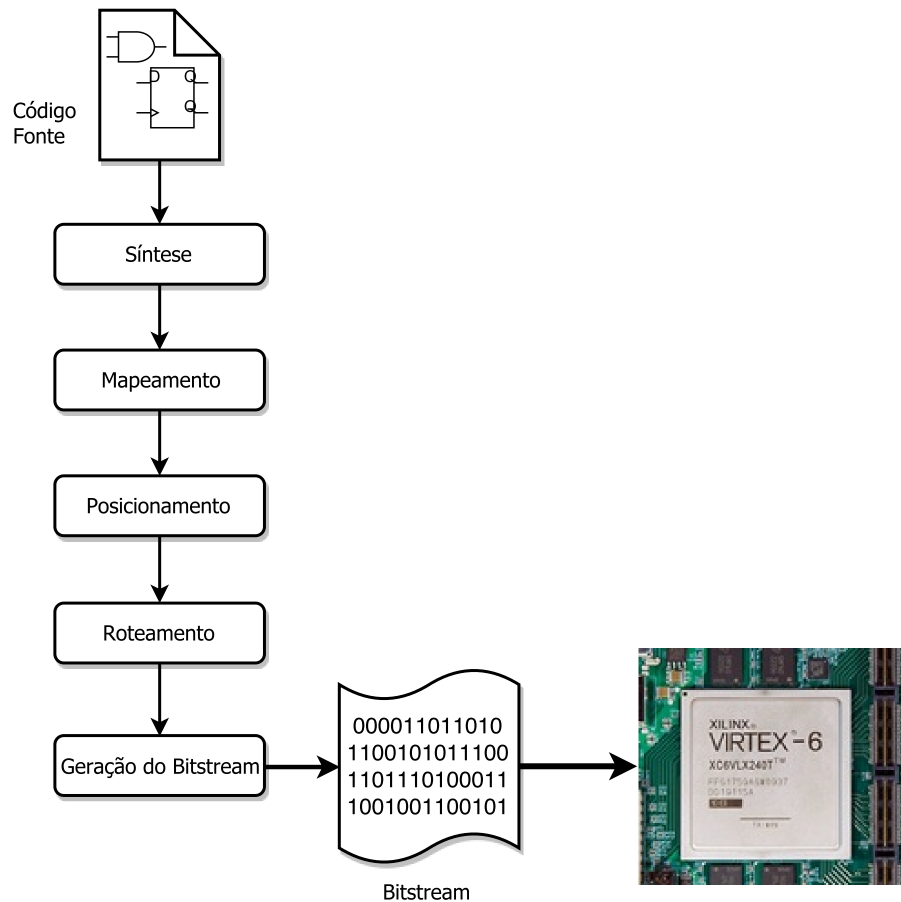

Figura 3.3: Fluxo de mapeamento do FPGA. Fonte: Próprio autor.

Fonte: Próprio autor

A segunda etapa do processo de implementação é o Mapeamento (*mapping*), nessa que ocorre as combinações ou decomposições das operações de computação de acordo com os blocos de hardware específicos do FPGA alvo. Na etapa do Posicionamento, a terceira, os blocos de hardware mapeados são posicionados no FPGA a fim de que os blocos lógicos que possuem possíveis interconexões tornem-se mais próximos um do outro. A etapa seguinte, a do Roteamento (*routing*), as interconexões físicas do FPGA aconte-

cem com os blocos já posicionados na etapa do Posicionamento (Noronha 2017). Por fim, sucede a geração do bitstream, um arquivo binário que define todos os pontos de programação do FPGA, de modo que os blocos lógicos e de roteamento sejam configurados apropriadamente para o FPGA alvo (Torquato 2017). Que neste trabalho é o Xilinx Virtex 6 semelhante ao da Figura 3.3.

3.2.2 Consumo de Energia

O menor consumo de energia e a maior largura de banda são dois requisitos importantes na implementação de aplicações de alta velocidade em hardware reconfigurável. Uma tendência global em vários mercados é o aumento de velocidade nas operações mantendo o mesmo consumo de energia, a qual pode ser utilizada em uma FPGA por meio de uma maior largura de banda com a mesma ou menor potência e custo. Para isso, muitas razões devem ser consideradas ao planejar uma implementação em FPGA, pois é preciso observar fatores específicos que incluem selecionar a tecnologia de processo correta, projetar a arquitetura adequada ao objetivo, aplicar a otimização correta do software e permitir um projeto a nível de sistema simples e eficiente em termos de energia.

Para a (Altera 2012) existem três componentes para o consumo de energia: estático, dinâmico e potência de I/O.

A potência estática é a consumida pelo FPGA quando nenhum sinal está alternando, ou seja, a corrente de fuga, única fonte de dissipação de energia estática. Devido a quantidade ínfima de corrente de fuga, a parcela de potência estática consumida é muito menor que a dinâmica (da Silva 2016).

A potência dinâmica é a energia adicional consumida através da operação do dispositivo causado por carga de sinais e cargas capacitivas carregando e descarregando. As principais variáveis que afetam a potência dinâmica são o carregamento de capacitância, a tensão de alimentação e a frequência do clock. Segundo a (Altera 2012), a potência dinâmica fica menor ao aplicar a lei de Moore, visto que aproveita a diminuição do processo para reduzir a capacitância e a tensão. O desafio torna-se encontrar um limiar entre a área e o clock, pois à medida que a área reduz em cada processo de diminuição, a frequência de clock máximo aumenta. Enquanto a redução de potência diminui para um circuito equivalente entre as extremidades do processo, a capacidade do FPGA duplica e aumenta a frequência de clock máximo. A potência dinâmica foi a utilizada para efetuar as medidas neste trabalho.

A potência de entrada/saída (I/O) inclui o consumo de energia utilizado por blocos de I/O, incluindo os de uso geral e transceptores de série de alta velocidade. Os principais fatores que influenciam o consumo de energia de I/O de uso geral são as resistências de terminação e terminação paralela em chip, capacidade do buffer de saída, taxa de retorno do buffer de saída, fonte de tensão e carga capacitiva (carga e descarga) (Altera 2012).

Capítulo 4

Implementações

Neste capítulo é apresentado como se deu a abordagem de implementação dos algoritmos de hash MD5 e SHA-1 apresentados em detalhes no Capítulo 2, respeitando o fluxo do processo. Cada algoritmo ficou com uma proposta apresentada nas duas seções do capítulo.

4.1 Proposta de Implementação do MD5

A Figura 4.1 apresenta a arquitetura geral da implementação em hardware do MD5 proposto. A estrutura permite a visualização do algoritmo em diagrama de blocos, desenvolvido em *Register Transfer Level* (RTL), no qual, pode-se observar o fluxo de sinais (ou variáveis) entre os componentes de *datapath* e os registradores RA, RB, RC e RD, os quais estão destacados na cor azul. O hardware inicia com a i-ésima mensagem $\mathbf{m}_i$ entrando em um módulo chamado de INIT, o responsável pelas funcionalidades apresentadas entre as linhas 1 e 6 do Algoritmo 1, o controle dos dois *loops* (linhas 7 e 11) e a inicialização das variáveis *hash* ($\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$) a cada j-ésimo bloco, $\mathbf{b}_j$, através do sinal **h0**.

Os blocos CJ e CN (destacados na cor amarelo na Figura 4.1) são contadores de $\log_2(L)$ e de 6 bits, respectivamente. O contador CN é responsável pela iteração do *loop* da linha 11 do Algoritmo 1, gerando o sinal n. Já o contador CJ é incrementado pelo contador CN e controla a iteração do *loop* da linha 7 do Algoritmo 1, através do sinal j. Com base na linha 8 do Algoritmo 1 e capítulo 2 , o módulo DM (na cor laranja da Figura 4.1) faz divisão da i-ésima mensagem $\mathbf{z}_i$ em L blocos de $M = 512$ bits, no qual cada j-ésimo bloco é caracterizado na Figura 4.1 pelo sinal $\mathbf{b}_j$. O sinal $\mathbf{b}_j$, de $M = 512$ bits é, então, dividido igualmente em 16 barramentos de 32 bits, no qual, cada i-ésimo barramento é representado pelo sinal $\mathbf{u}_j[i]$. A seleção de cada i-ésimo barramento é realizada pelo multiplexador M (na cor lilás) a partir do sinal $g(n)$. Esta operação é apresentada no capítulo 2 pelas Equações 2.13 e 2.14. Os módulos GF, GG e GQ (ilustrados na cor verde na Figura 4.1) representam as operações expressas pelas Equações 2.20, 2.25, 2.33 e 2.34. Observa-se que diferentemente das implementações em processadores sequenciais como GPP, microcontroladores e outros, estas equações são executadas em paralelo, acelerando o algoritmo MD5. A implementação do módulo GF é detalhada na próxima subseção. Os detalhes da implementação dos módulos GF, GG, GQ e LR (na cor vermelho da 4.1)

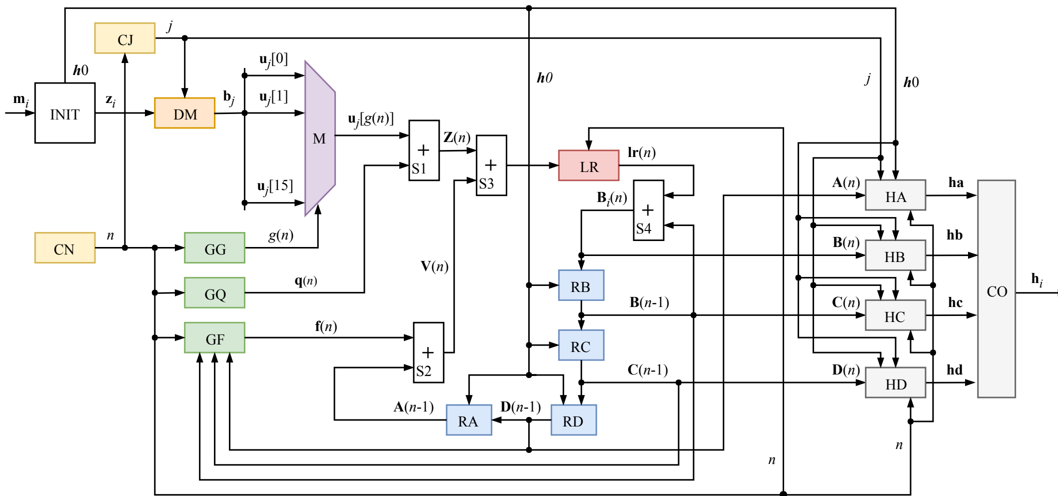

Figura 4.1: Arquitetura geral da implementação em hardware do MD5 proposto.

Fonte: Próprio autor

estão detalhados nas subseções a seguir.

4.1.1 Módulo GF

O módulo GF implementa a função descrita na subseção 2.4.7 apresentado na linha 13 do Algoritmo 1. Este módulo é formado por um multiplexador chamado de GF-MUX que seleciona o tipo da função $f(\cdot)$ a partir do valor de n de acordo com a Equação 2.20.

A escolha da função a ser executada no multiplexador é controlada pelo sinal sGF, que corresponde aos dois bits mais significativos do valor referente ao sinal n o qual possui 6 bits (o sinal n é gerado pelo contador CN).

$$\mathbf{sGF} = \begin{bmatrix} n_5 & n_4 \end{bmatrix} \tag{4.1}$$

onde

$$\mathbf{n} = \begin{bmatrix} n_0 & \ldots & n_5 \end{bmatrix} \text{ onde } n_k \in \{0,1\} \forall k. \tag{4.2}$$

Já o bloco GF-SLC realiza uma operação chamada de *slice* e retira os dois bits mais significativos de n. É importante observar que as operações de *slice* não ocupam espaço em hardware.

4.1.2 Módulo GG

A Figura 4.3 detalha a estrutura de hardware do módulo GG, a implementação da função detalhada na Equação 2.25 e expressa na linha 14 do Algoritmo 1. O bloco GG-SLC1 realiza uma operação semelhante ao bloco GF-SLC retirando os dois bits mais significa-

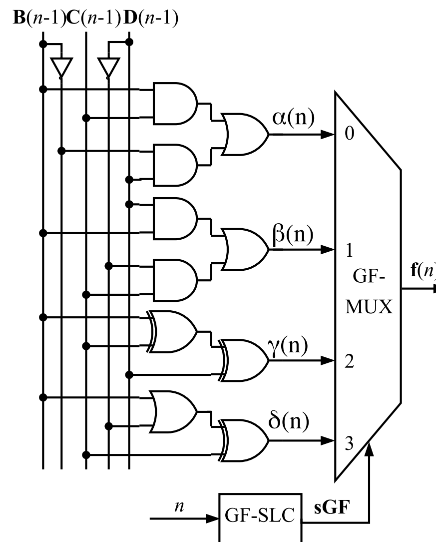

Figura 4.2: Arquitetura do módulo GF.

Fonte: Próprio autor

tivos de n assim,

$$\mathbf{sGG} = \left[\begin{array}{cc} n_5 & n_4 \end{array} \right]. \tag{4.3}$$

Já os blocos GG-SLC2, GG-SLC3, GG-SLC4 e GG-SLC5 realizam a operação de *slice* retirando os 4 bits mais significativos do sinal antes de entrar no multiplexador, GG-MUX. Esta operação deve ser realizada dado que o sinal $g(n)$ é o sinal de entrada do seletor do multiplexador M, apresentado na Figura 4.1. Como as operações de *slice* não ocupam espaço em hardware, elas foram alocadas antes do GG-MUX reduzindo desta forma de 6 para 4 bits os barramentos para construção do multiplexador GG-MUX.

4.1.3 Módulo GQ

Nesta subseção é apresentada a estratégia de implementação em hardware para o módulo GQ, representado na Equação 2.34 e na linha 13 do Algoritmo 1. Como neste caso existe uma função não linear do tipo seno, utilizou-se a estratégia de representá-la por meio de *lookup tables* (LUTs). Essa estratégia é interessante pela redução do *clock* quando comparada a outras estratégias apresentadas na literatura (de Souza & Fernandes 2014). Nos trabalhos apresentados em (Deepakumara et al. 2001, Jarvinen et al. 2005) são utilizadas estratégias semelhantes porém, com blocos de memória diferentemente da implementação apresentada neste trabalho. A utilização de blocos de memória RAM reduz a ocupação de espaço de hardware em circuitos lógicos, porém aumenta a quantidade de clocks necessários para realização do algoritmo. Este módulo pode ser visto na Figura 4.4.

Os valores da função são armazenados em registradores e os blocos GQ-SLC1 e

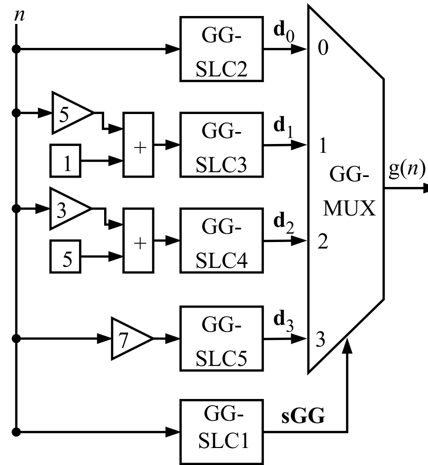

Figura 4.3: Arquitetura do módulo GG.

Fonte: Próprio autor

GQ-SLC2 retiram os bits mais significativo e os 5 menos significativos de n, respectivamente, ou seja,

$$\mathbf{sGQ}_1 = \left[\begin{array}{c} n_5 \end{array} \right] \tag{4.4}$$

e

$$\mathbf{sGQ}_2 = \left[\begin{array}{ccccc} n_4 & n_3 & n_2 & n_1 & n_0 \end{array} \right]. \tag{4.5}$$

4.1.4 Módulo LR

A Figura 4.5 detalha a implementação do módulo LR, que executa a função *leftrotate* descrita pela Equação 2.30 e linha 14 do Algoritmo 1. Como apresentado nas Equações 2.29, 2.33 e 2.35 a operação de *leftrotate* executa vários deslocamentos no sinal $\mathbf{Z}(n) + \mathbf{V}(n)$ cujo valor dos deslocamentos é controlado através do sinal $\mathbf{s}[n]$ (ver Equação 2.33).

4.1.5 Processamento do hash $\mathbf{h}_i$

Como o sinal $\mathbf{s}[n]$ é formado por quatro grupos de vetores de constantes (ver Equação 2.33), é utilizado um multiplexador (SLR-MUX) para selecionar um entre os quatro grupos, onde cada i-ésimo grupo é processado por um módulo chamado SLR$i(k)$ detalhado na Figura 4.6. Cada i-ésimo SLR$i(k)$ trabalha com os valores $\mathbf{s}[i..i+15]$. A seleção de i-ésimo SLR$i(k)$ é controlada pelo bloco LR-SLC2 que através da operação de *slice* captura o dois bits mais significativos de n, ou seja,

$$\mathbf{sLR} = \left[\begin{array}{cc} n_5 & n_4 \end{array} \right]. \tag{4.6}$$

Em cada i-ésimo bloco SLR$i(k)$ são realizadas apenas quatro operações de *leftrotate*

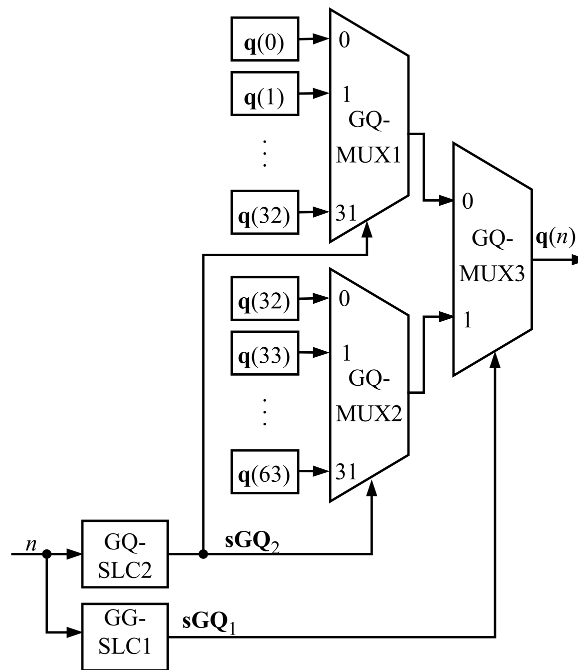

Figura 4.4: Arquitetura do módulo GQ.

Fonte: Próprio autor

como pode-se observar em cada grupo de vetores, $\mathbf{s}[i..i+15]$ (ver Equação 2.33), os valores se repetem em quatro sub-grupos. Assim, pode-se reduzir a implementação em hardware utilizando os dois bits menos significativos em cada i-ésimo grupo. A seleção do valor associado ao deslocamento em cada i-ésimo bloco SLR$i(k)$ é realizada pelo sinal, **sSLR**, através do *slice* LR-SLC1, ou seja,

$$\mathbf{sSLR} = \begin{bmatrix} n_1 & n_0 \end{bmatrix}. \quad (4.7)$$

Após a geração dos sinais $\mathbf{u}_j[g(n)]$, $\mathbf{q}(n)$, $\mathbf{f}(n)$, em cada n-ésima iteração, e do valor $\mathbf{A}(n-1)$, os sinais $\mathbf{Z}(n)$ e $\mathbf{V}(n)$, ambos de 32 bits, são calculados através dos módulos de soma S1 e S2 que são executados em paralelo. Todos os módulos de soma utilizados na implementação são circuitos específicos de 32 bits, o que otimiza o tempo de processamento e o espaço ocupado pelo circuito total. O cálculo dos sinais $\mathbf{Z}(n)$ e $\mathbf{V}(n)$ é expresso pelas Equações 2.31 e 2.32 e é executado durante a linha 14 do Algoritmo 1. A última etapa de cada n-ésima iteração é a atualização das variáveis hash, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$ e $\mathbf{D}(n)$, armazenada nos registradores RA, RB, RC e RD, respectivamente. Para esta operação é calculado o novo valor de $\mathbf{B}(n)$ através dos módulos de soma S3 e S4 do módulo LR. Esse último calcula a operação de leftrotate(r,y) expressa em detalhes pela Equação 2.72. Durante o cálculo do novo valor de $\mathbf{B}(n)$ os valores dos registradores deslocam-se entre si atualizando as outras variáveis *hash* de acordo com as Equações 2.26 à 2.32. Este procedimento é executado durante a linha 14 do Algoritmo 1.

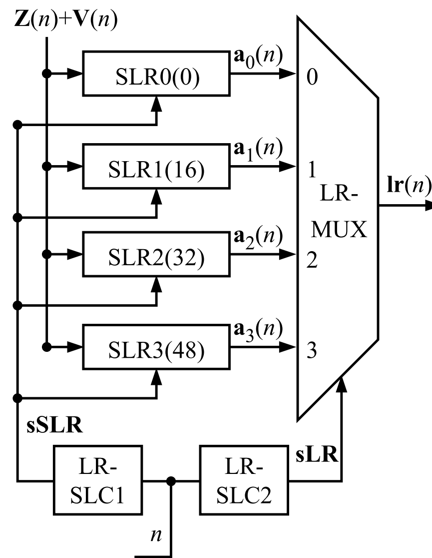

Figura 4.5: Arquitetura do módulo LR.

Fonte: Próprio autor

Ao final das 64 iterações do *loop* em n (linha 11 do Algoritmo 1) as partes que compõem o código *hash*, **ha**, **hb**, **hc** e **hd** (ver Equação 2.49), são atualizadas pelos módulos (registradores) HA, HB, HC e HD (destacados na cor cinza na Figura 4.1), respectivamente. Esta etapa é executada na linha 16 do Algoritmo 1. Por último, ao final das N_i iterações (ver Equação 2.40) tem-se o valor final do código *hash*, $\mathbf{h}_i$, associado a i-ésima mensagem. O módulo CO têm a função de concatenar os 4 barramentos de 32 bits formados pelos sinais **ha**, **hb**, **hc** e **hd** e gerar um sinal serial com o código *hash*, $\mathbf{h}_i$. A estrutura detalhada na Figura 4.1 utiliza uma estratégia chamada de *Iterative Looping* também utilizada nos trabalhos (Deepakumara et al. 2001, Jarvinen et al. 2005, Yiakoumis et al. 2005). Todavia, a implementação proposta aqui neste trabalho segue abordagens diferentes na composição dos módulos, gerando uma proposta mais otimizada com um *throughput* mais elevado (800 Mbps) quando comparado as outras estratégias com IL, conforme descrito na subseção 5.1.

4.2 Proposta de Implementação do SHA-1

A Figura 4.7 apresenta a proposta deste artigo para a arquitetura geral do SHA-1 implementado em hardware. A estrutura permite a visualização do algoritmo em *Register Transfer Level* (RTL), no qual pode-se observar o fluxo de sinais (ou variáveis) entre os componentes de *datapath* e os registradores RA, RB, RC, RD e RE, destacados na cor azul. O hardware inicia com a i-ésima mensagem $\mathbf{m}_i$ entrando em um módulo chamado de INIT que é responsável pelas funcionalidades apresentadas entre linhas 1 e 6 do Algoritmo

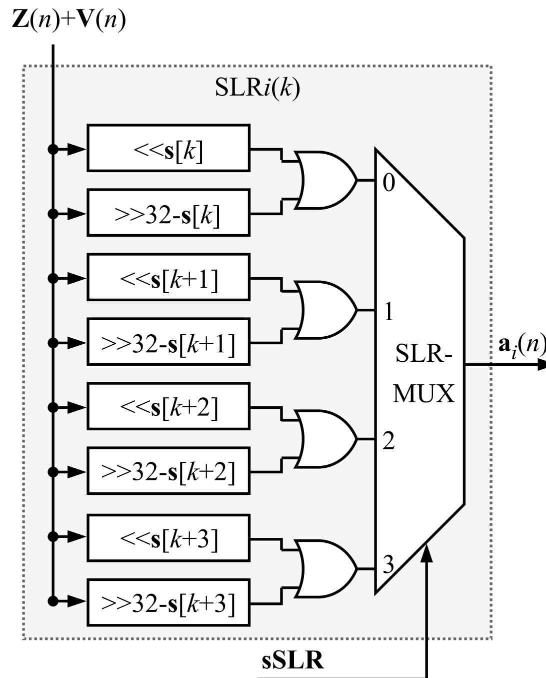

Figura 4.6: Arquitetura do i-ésimo módulo SLR$i(k)$.

Fonte: Próprio autor

2, o controle dos dois *loops* (linhas 7 e 11) e a inicialização das variáveis *hash* ($\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$) a cada j-ésimo bloco, $\mathbf{b}_j$, através do sinal $\mathbf{h}0$.

Os blocos CJ e CN (evidenciados pela cor amarelo) são contadores de $\log_2(L)$ e 7 bits, respectivamente. O contador CN é responsável pela iteração do *loop* da linha 11 do Algoritmo 2, gerando o sinal n. Já o contador CJ é incrementado pelo contador CN e controla a iteração do *loop* da linha 7 do Algoritmo 2, através do sinal j. Com base na linha 8 do Algoritmo 2 e subseção 2.4.4, o módulo DM (na cor laranja) faz a divisão da i-ésima mensagem $\mathbf{z}_i$ em L blocos de $M = 512$ bits, no qual cada j-ésimo bloco é caracterizado na Figura 4.7 pelo sinal $\mathbf{b}_j$. O sinal $\mathbf{b}_j$, de $M = 512$ bits é, então, dividido igualmente em 16 barramentos de 32 bits, no qual, cada i-ésimo barramento é representado pelo sinal $\mathbf{u}_j[n]$. Após esta etapa é gerado sinal $\mathbf{w}[n]$ pelo módulo GW, descrito na linha 12 do Algoritmo 2 e representado na cor lilás, a partir do sinal do contador CN.

Os módulos GF, GK (realçados pela cor verde na Figura 4.7) e GW (na cor lilás) representam as operações expressas pelas Equações 2.59, 2.71 e 2.56, respectivamente. Já o módulos LR5 e LR30 (na cor vermelho) representam operações de *leftrotate* expressa nas Equações 2.68 e 2.66. Observa-se que diferentemente das implementações em processadores sequenciais como GPP, uC (microcontroladores) e outros, estas equações são executadas em paralelo, acelerando o algoritmo SHA-1. Os detalhes da implementação dos módulos GF, GW, RL5 e RL30 estão detalhados nas subseções a seguir.

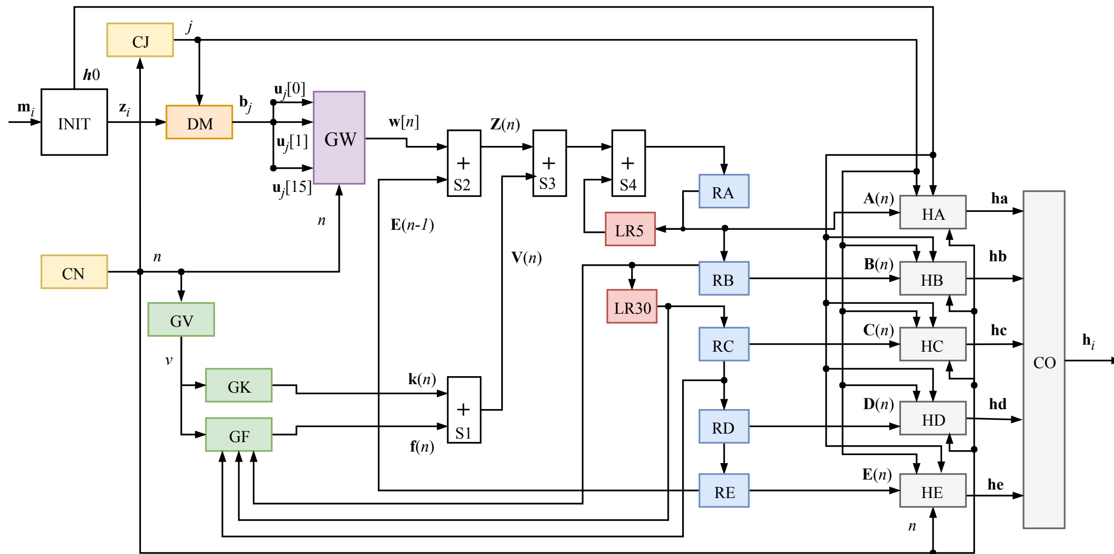

Figura 4.7: Arquitetura geral da implementação em hardware do SHA-1 proposto.

Fonte: Próprio autor

4.2.1 Módulo GF

O módulo GF implementa a função descrita na subseção 2.4.7 e apresentado na linha 13 do Algoritmo 2. Este módulo é formado por um multiplexador chamado de GF-MUX que seleciona o tipo função a partir do valor de n de acordo com a Equação 2.59 e detalhado na Figura 4.8.

A seleção do tipo da função no multiplexador GF-MUX é controlada pelo módulo GV, por meio de uma lógica binária com comparadores e portas lógicas correspondentes a cada intervalo, tendo as seguintes saídas,

$$GV = \begin{cases} 0 & \text{para } n = 0 \dots 19 \\ 1 & \text{para } n = 20 \dots 39 \\ 2 & \text{para } n = 40 \dots 59 \\ 3 & \text{para } n = 60 \dots 79 \end{cases} . \tag{4.8}$$

Cada qual selecionando uma função $\mathbf{f}(n)$ baseado no contador n de 7 bits do módulo CN.

4.2.2 Módulo GW

O módulo GW é composto por 16 mensagens de 32 bits $\mathbf{u}_j[n]$ na entrada, originadas de $\mathbf{b}_j[n]$, conforme a Equação 2.52 da Subseção 2.4.4. E possui a função de realizar a operação demonstrada pela Equação 2.56, descrita na subseção 2.4.6 e linha 12 do Algoritmo 2.

A Figura 4.9 detalha o módulo formado por um multiplexador de 80 entradas, cha-

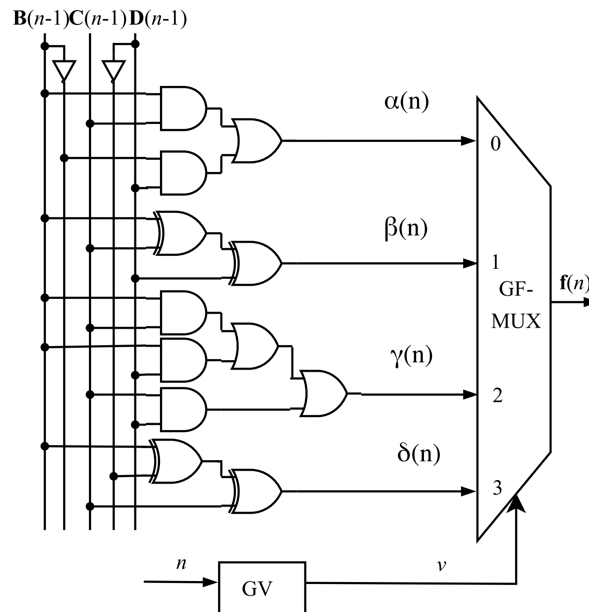

Figura 4.8: Arquitetura do módulo GF.

Fonte: Próprio autor

mado de W-MUX, que possui como seletor das entradas o sinal n de 0..79. Para os valores referentes a n de 16 a 79 é gerado o sinal $\mathbf{sw}[n]$, expresso pela equação 2.57 através do módulo SWk, o qual $k = 16\dots 79$, especificado na Figura 4.10.

Cada k-ésimo módulo SWk é formado por um registrador, chamado aqui de RWk, um módulo de *leftrotate* (ver Equação 2.58) chamado de LR1, uma porta ou exclusivo (XOR) e um comparador. O registrador RWk armazena o valor do sinal $\mathbf{sw}[n]$ através do comparador quando $n = k-3$. A porta XOR realiza a operação descrita na Equação 2.57 e o módulo LR1 realiza a função de *leftrotate* para $\mathbf{s} = 1$. A Figura 4.11 detalha um módulo genérico associado a implementação do *leftrotate* com base na Equação 2.58.

Os módulos LR5(s) e LR30 realizam a operação de leftrotate para $\mathbf{s} = 5$ (ver Equação 2.68) e $\mathbf{s} = 30$ (ver Equação 2.66), respectivamente. A implementação destes módulos também é caracteriza pela Figura 4.11.

4.2.3 Processamento do hash $\mathbf{h}_i$

Após a geração dos sinais $\mathbf{w}(n)$, $\mathbf{k}(n)$, $\mathbf{f}(n)$, em cada n-ésima iteração, e do valor $\mathbf{E}(n-1)$, os sinais $\mathbf{Z}(n)$ e $\mathbf{V}(n)$, ambos de 32 bits, são calculados através dos módulos de soma S1 e S2, executados em paralelo e posteriormente S3 e S4. Todos os módulos de soma utilizados na implementação são circuitos específicos de 32 bits, o que otimiza o tempo de processamento e o espaço ocupado pelo circuito total. O cálculo dos sinais $\mathbf{Z}(n)$ e $\mathbf{V}(n)$ é expresso pelas Equações 2.69 e 2.70 e é executado durante a linha 14 do Algoritmo 2. A última etapa de cada n-ésima iteração é a atualização das variáveis

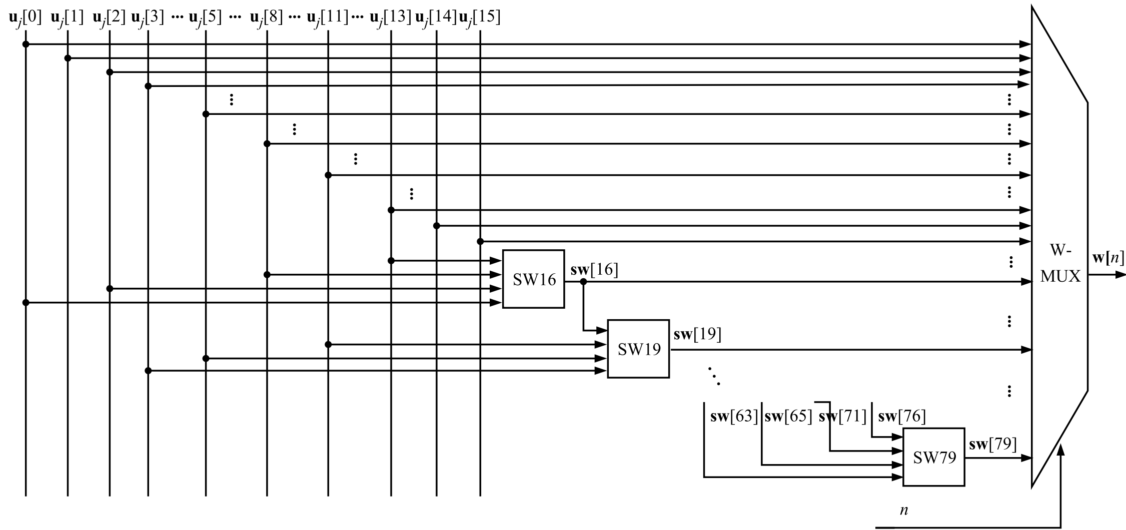

Figura 4.9: Arquitetura do módulo GW

Fonte: Próprio autor

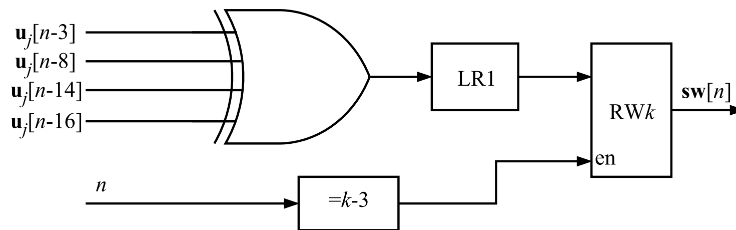

Figura 4.10: Operações do módulo SW*k*

Fonte: Próprio autor

hash, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$, armazenada nos registradores RA, RB, RC, RD e RE, respectivamente. Outra etapa importante no SHA-1 é a descrita na 4.2.2, na qual o valor do registrador RC é atualizado por meio da operação $\text{lr}(r,30)$, expressa em detalhes pela Equação 2.58. A cada iteração de **n** os valores dos registradores deslocam-se entre si atualizando as outras variáveis *hash* de acordo com as Equações 2.64 a 2.68. Este procedimento é executado durante a linha 14 do Algoritmo 2.

Ao final das 80 iterações do *loop* em n (linha 11 do Algoritmo 2) as partes que compõem o código *hash*, **ha**, **hb**, **hc**, **hd** e **he** (ver Equação 2.49), são atualizadas pelos módulos HA, HB, HC, HD e HE, respectivamente. Esta etapa é executada na linha 16 do Algoritmo 2. Assim, tendo o valor final do código *hash*, $\mathbf{h}_i$, associado a i-ésima mensagem. O módulo CO têm a função de concatenar os 5 barramentos de 32 bits formados pelos sinais **ha**, **hb**, **hc**, **hd** e **he** e gerar um sinal serial com o código *hash* $\mathbf{h}_i$. A estrutura detalhada na Figura 4.7 utiliza uma estratégia chamada de *Iterative Looping* também utili-

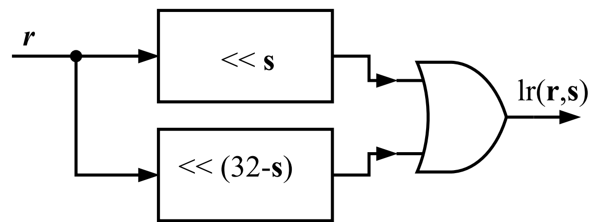

Figura 4.11: Arquitetura do módulo RL*s*.

Fonte: Próprio autor

zada nos trabalhos (Deepakumara et al. 2001, Jarvinen et al. 2005, Yiakoumis et al. 2005) todavia, a implementação proposta aqui neste trabalho segue abordagens diferentes na composição dos módulos gerando uma proposta com um *throughput* de 652 Mbps, conforme apresentados na subseção 5.2.

Capítulo 5

Resultados Alcançados

Neste capítulo estão demonstrados e descritos os resultados alcançados em cada implementação proposta no FPGA alvo dos algoritmos MD5 e SHA-1, um em cada seção. Além de uma seção final com um comparativo de consumo de energia entre dois tipos de microcontroladores, sendo um de 8 bits e o outro de 32 bits, e a implementação proposta em hardware reconfigurável.

5.1 Resultados para a implementação do MD5

A Tabela 5.1 apresenta os resultados obtidos após o processo de síntese do hardware proposto neste trabalho para o algoritmo MD5 (ver Figura 4.1). São apresentados resultados relativos a ocupação do hardware no FPGA alvo (Virtex 6 xc6vlx240t-11156) bem como os resultados associados a área e ao *throughput* alcançados após o processo de síntese. Foram gerados resultados para várias implementações em paralelo do algoritmo MD5 de acordo com a Figura 4.1, seguindo a mesma linha dos trabalhos apresentados em (Jarvinen et al. 2005, He & Xue 2010). A utilização de vários módulos em paralelo permite acelerar o *throughput* principalmente em casos de recuperação de senha por força bruta, nos quais existem um grande número de códigos *hash* a serem gerados.

A primeira coluna da tabela, chamada de NI, indica o número de módulos paralelos implementados, ou seja, Número de Implementações. É importante destacar que o máximo de módulos paralelos era até então de 32 no trabalho apresentado em (Wang et al. 2010). A segunda coluna, chamada de NR apresenta o número de registradores utilizados no FPGA e a terceira coluna, chamada de PR, representa a porcentagem de registradores utilizados em relação ao total disponível no FPGA alvo que são de 301.440. Já a quarta e a quinta coluna, chamadas de NLUT e PLUT, representam a quantidade de células lógicas utilizadas como LUTs (*Lookup Tables*) para construção de circuitos digitais e a porcentagem de LUTs utilizados em relação ao total disponível no FPGA alvo que são de 150.720. Finalmente, a sexta e sétima coluna mostram os resultados, obtidos para várias implementações, da taxa de amostragem, T_s e *throughput*, R_s, respectivamente.

Comparando os resultados de ocupação para NI = 1 (primeira linha da Tabela 5.1) com os trabalhos apresentados em (Deepakumara et al. 2001, Jarvinen et al. 2005, He & Xue 2010) observa que a implementação proposta neste trabalho ocupa uma área de circuito entre 50% e 30% menor.

Tabela 5.1: Resultados relativos a ocupação, taxa de amostragem e *throughput* para várias implementações paralelas do algoritmo MD5.

NI	NR	PR (%)	NLUT	PLUT (%)	T_s (ns)	R_s (Gbps)
1	136	0,05	313	0,21	9,723	0,8228
2	272	0,09	626	0,42	9,941	1,6095
4	544	0,18	1.353	0,90	9,965	3,2112
8	1.088	0,36	2.688	1,78	9,986	6,4090
16	2.176	0,72	5.505	3,65	12,097	10,5811
32	4.352	1,44	10.970	7,28	13,992	18,2962
64	8.704	2,89	22.078	14,65	14,903	34,3555
256	34.816	11,55	88.109	58,46	16,048	127,6171
320	43.520	14,44	109.950	72,95	18,179	140,8218

Assim, apesar do algoritmo ser sequencial o hardware foi desenvolvido de forma paralela, como apresentado na Seção 4. Isso por meio de barramentos de 32 bits, a fim de que o tempo de amostragem, T_s, seja equivalente ao *clock*, isto é, cada n-ésima iteração (ver *Loop* da linha 11 do Algoritmo 1) é realizada em um tempo de pulso de *clock*, chamado aqui de $t_{\text{CLK}} = T_s$. Aos valores de T_s apresentados na sexta coluna observa-se que não existe uma mudança significativa com o aumento de NI, ou seja, para um incremento de 300 vezes de NI houve apenas um incremento de 2 vezes em T_s. Comparando os valores encontrado na literatura para T_s (com NI = 1) observa-se que o único trabalho que encontrou um tempo de *clock* semelhante foi o apresentado em (Wang et al. 2010) que encontrou um valor de $T_s \approx 9,7\,\text{ns}$.

Com base no Algoritmo 1 e arquitetura apresentada na Figura 4.1, para cada j-ésimo bloco $\mathbf{b}_j$ de $M = 512\,\text{bits}$ são executadas 64 iterações (ver Equação 2.40), assim, o *throughput* do hardware proposto pode ser calculado como

$$R_s = \frac{M \times \text{NI}}{64 \times T_s} = \frac{512 \times \text{NI}}{64 \times T_s} = \frac{8 \times \text{NI}}{T_s}. \tag{5.1}$$

É importante destacar que valores de *throughput* maiores que 100 Gbps são inéditos na literatura (NI = 256 e NI = 320). Um *throughput* de 140 Gbps equivale a recuperar uma senha (usando um método de força bruta) de 6 dígitos apenas numéricos em 4 ms ou uma senha de 6 dígitos alfa numéricos (cada dígito com 62 possibilidades) a partir de um código *hash* em torno de 3,5 minutos.

Para validar a implementação proposta neste trabalho e os valores alcançados, utilizou-se uma implementação do MD5 em linguagem de programação *C* e de verificadores online, disponíveis na internet. Desse modo, foram utilizadas mais de cem palavras aleatórias com tamanhos entre 4 e 52 caracteres geradas por meio de um script no *MATLAB*. Cada mensagem dessa se torna uma entrada no programa em C e dentro do FPGA em co-simulação, ao fim do processamento de cada caso foi verificado por meio de uma função que compara strings que todos os resultados estavam iguais. Uma outra validação foi feito

por meio das entradas *abc* (chamada aqui de entrada 1) e *AAAABBBBCCCCDDDDEEEEFFFFMMMMGGGGHHHHIIIIJJJJKKKKLLLLMMMM* (entrada 2). Essas entradas foram adicionadas (uma por vez) estaticamente na entrada do multiplexador *M* (na cor lilás da Figura 4.1). Sendo possível obter os mesmos códigos hashes, $\mathbf{h}_i$, tanto na saída do *CO* (na cor cinza da Figura 4.1) quanto no código C e nos geradores de hash MD5 online [1]. Para a entrada 1 (um) foi possível obter o hash *900150983CD24FB0D6963F7D28E17F72* e para a entrada 2 (dois), o hash *0270A922CD B60A4A5C26F4C937F6FF51*.

5.2 Resultados para a implementação do SHA-1

Para o algoritmo SHA-1, a Tabela 5.2, semelhante a apresentada na seção 5.1, também apresenta os resultados obtidos após o processo de síntese no FPGA proposto nesse trabalho, conforme a Figura 4.7. Os resultados são mostrados segundo a ocupação do FPGA alvo, o Virtex 6 xc6vlx240t-11156, a área utilizada pelo circuito e o *throughput* atingido após o processo de síntese. Também foram gerados resultados para várias implementações em paralelo do algoritmo SHA-1, sendo uma implementação (ou módulo) a que está em acordo com a Figura 4.7, que segue uma linha diferente dos trabalhos apresentados em (Michail et al. 2005, Kakarountas et al. 2006, Khan et al. 2014, Michail et al. 2016), os quais utilizam estruturas seriadas (em *pipeline*). Já a proposta apresentada aqui faz utilização de vários módulos do SHA-1 em paralelo, semelhante a implementada no MD5, a qual permite acelerar o *throughput* principalmente em aplicações que necessitem de alto desempenho.

Tabela 5.2: Resultados relativos a ocupação, taxa de amostragem e *throughput* para várias implementações paralelas do algoritmo SHA-1.

NI	NR	PR (%)	NLUT	PLUT (%)	T_s (ns)	R_s (Gbps)
1	2.154	0,71	2.605	1,72	9,932	0,644
4	8.575	2,84	10,388	6,89	9,961	2,570
8	17.136	5,68	20.662	13,71	9,965	5,138
16	34.255	11,36	43.263	28,70	9,994	10,246
32	68.498	22,72	86.873	57,64	9,994	18,296
48	102.733	34,08	129.902	86,19	10,909	28,160

Análogo aos resultados de MD5, apresentados na seção 5.1, o hardware também foi desenvolvido de modo paralelo como demonstrado no Capítulo 4, por meio de barramentos de 32 bits, a fim de que o tempo de amostragem, T_s, seja correspondente ao *clock*, isto é, cada *n*-ésima iteração (ver *Loop* da linha 11 do Algoritmo 2) é realizada em um tempo de pulso de *clock*, chamado aqui de $t_{\text{CLK}} = T_s$. É possível verificar que não há uma mudança significativa com o aumento de NI, ou seja, para um incremento de 48 vezes de

[1] um exemplo desses é a página https://passwordsgenerator.net/md5-hash-generator/

NI houve apenas um incremento de menos de 1 ns em T_s, o que representa um aumento de quase 32 vezes no *throughput* de geração do *hash*.

Com base no Algoritmo 2 e arquitetura apresentada na Figura 4.7, para cada j-ésimo bloco $\mathbf{b}_j$ de $M = 512\,\text{bits}$ são executadas 80 iterações (ver Equação 2.77), assim, o *throughput* do hardware proposto pode ser calculado como

$$R_s = \frac{M \times \text{NI}}{80 \times T_s} = \frac{512 \times \text{NI}}{80 \times T_s} = \frac{64 \times \text{NI}}{10 \times T_s}. \tag{5.2}$$

É importante destacar que valores de *throughput* maiores que 15 Gbps não foram encontrados na literatura (NI $= 32$ e NI $= 48$). Um *throughput* de $28,16$ Gbps equivale a recuperar uma senha totalmente desconhecida (usando o método de força bruta) de 6 dígitos apenas numéricos em 4 ms ou um senha também de 6 dígitos apenas numéricos no tempo máximo de 20 ms ou uma senha também de 6 dígitos alfa numéricos (cada dígito com 62 possibilidades) a partir de um código *hash* no tempo máximo de $17,4$ minutos.

Os resultados com o algoritmo SHA-1 foram verificados e validados utilizando a ferramenta System Generator do Matlab, seguindo o mesmo modelo do que foi executado no MD5, descritos na subseção 5.1. Foi utilizado uma implementação em linguagem de programação *C* e de verificadores online, disponíveis na Internet. Do mesmo modo, várias entradas foram verificadas, as quais trouxeram as saídas esperadas, conforme pode ser visto no video publicado no YouTube no link https://youtu.be/ZLxNjShODyo. Outro exemplo feito foi feito utilizando a entrada *abc* (entrada 1), a mesma utilizada pelo (NIST 2016), e a *AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM* (entrada 2). Essas entradas foram adicionadas (uma por vez) estaticamente na entrada do multiplexador *GW* (na cor lilás da Figura 4.7). Assim, foi possível obter os mesmos códigos hashes, $\mathbf{h}_i$, tanto na saída do *CO* (na cor cinza da Figura 4.7) quanto no código C e nos geradores de hash SHA-1 online [2]. Para a entrada 1 foi possível obter o hash *A9993E364706816ABA3E25717850C26C9CD0D89D* e para a entrada dois o hash *C49608A52B9E405D4150024B97F6F7F2940F1486*.

5.3 Resultados relacionados ao consumo de potência

Várias aplicações em dispositivos de Internet das Coisas (IoT) têm utilizado algoritmos de hash para verificação da integridade de dados e para manter bases com cadastros de usuários por meio do armazenamento de senhas, alguns desses algoritmos são o MD5 e o SHA-1. Todavia, a execução desses algoritmos pode demandar um elevado processamento e um aumento do consumo de potência dos dispositivos de IoT, os quais, normalmente, possuem uma limitada capacidade de processamento. Esses equipamentos, por muitas vezes, são energizados por meio de baterias e trazer soluções que melhorem o redimento do processamento e autonomia de utilização são benéficas para este nincho de pesquisa e mercado, o qual vem crescendo exponencialmente, segundo (Statista 2016).

Assim, foram executados testes comparativos em ambiente de laboratório utilizando as plataformas de hardware com um microprocessador de 8 bits e outro de 32 bits, além da

[2] um exemplo desses é a página https://passwordsgenerator.net/sha1-hash-generator/

implementação em hardware específico, já descrita no Capítulo 4. Para o caso do μP de 8 bits os resultados foram obtidos a partir de uma implementação em um microcontrolador ATmega2560 integrado a um kit Arduino com 16 MHz de clock alimentado com 5.0V e 500mA. Para o μP de 32 bits os resultados de tempo de processamento foram obtidos com uma implementação embarcada no kit Intel Galileo, utilizando o processador Intel Quark SOC x1000 com 400 MHz de *clock*, alimentado com 12V e $1,25$A. As implementações do SHA-1 e do MD5 foram desenvolvidas na linguagem de programação C, compilados com o GCC versão 4.9.1 na plataforma Intel Galileo, a qual estava embarcada com o sistema operacional *Yocto Standard*, versão 3.8.7 (uma distribuição GNU/Linux produzido para dispositivos embarcados). A Tabela 5.3 apresenta os resultados médio de tempo para quinze execuções associadas aos algoritmos MD5 e SHA-1 para cada plataforma citada.

Tabela 5.3: Resultados relativos ao tempo de execução dos algoritmos MD5 e SHA-1 nas plataformas ATmega 2560 e Intel Galileo Gen 2.

Plataforma	Tempo para o MD5	Tempo para o SHA-1
ATmega2560	$t^{8b}_{MD5} = 2{,}86$ ms	$t^{8b}_{SHA-1} = 3{,}81$ ms
Intel Galileo Gen 2	$t^{32b}_{MD5} = 0{,}05$ ms	$t^{32b}_{SHA-1} = 0{,}05$ ms

Os testes no microcontrolador ATmega 2560 foram realizados seguindo o procedimento de elevar o nível lógico de um pino digital ao iniciar o algoritmo, seja SHA-1 ou MD5, e logo após a execução da função *hash* abaixar o nível lógico, gerando um pulso que possui período igual ao tempo decorrido da execução do código *hash*, $\mathbf{h}_i$, do algoritmo analisado. No caso da plataforma com o microprocessador Intel Quark SoC X1000, foram analisados os tempos de relógio real, ou seja, o tempo exclusivo em que o microprocessador executou as instruções relativas ao algoritmo em questão. Isso por meio do comando *time* da distribuição GNU/Linux utilizada.

A Tabela 5.4 apresenta os resultados obtidos após o processo de síntese do hardware dedicado apresentados no Capítulo 4. As sínteses foram realizadas para duas FPGAs alvos a Artix 7 (modelo xc7a100t-2csg324) e a Spartan 6 (modelo xc6slx9-2csg324). A escolha dessas FPGAs justifica-se pelo baixo custo e pelo pequeno tamanho, semelhantes aos kits Arduino e Galileo experimentados neste trabalho, a fim de tornar a comparação do consumo de energia a principal variável. Para cada FPGA foram realizadas quatro sínteses: uma com o hardware proposto para o MD5 (ver Figura 4.1) com um tempo de amostragem $T_s = \frac{t^{8b}_{MD5}}{64}$ que equivale ao tempo medido do μP de 8 bits no μC ao final de 64 *clocks*; outra com o hardware também do MD5 mas como um tempo equivalente ao do μP de 32 bits, ou seja, $T_s = \frac{t^{32b}_{MD5}}{64}$; a terceira com o hardware proposto para o SHA-1 (ver Figura 4.7) e o tempo de amostragem $T_s = \frac{t^{8b}_{SHA-1}}{80}$ que equivale ao tempo medido do μP de 8 bits no μC ao final de 80 *clocks* e a quarta síntese é equivalente ao μP de 32 bits, ou seja, $T_s = \frac{t^{32b}_{SHA-1}}{80}$. A tabela 5.4 também mostra o número de registradores, NR, e o número de células lógicas, NLUT, ocupadas após a síntese.

Os resultados relativos a ocupação, apresentados na Tabela 5.4, mostram que para ambos os casos, MD5 e SHA-1, as implementações ocuparam uma quantidade muito

Tabela 5.4: Resultados relativos a ocupação, taxa de amostragem e *throughput* para as implementações em FPGA.

FPGA	Algoritmo	NR	NLUT	T_s	*clock*
Artix 7	MD5	136	341	$\frac{t^{8b}_{MD5}}{64} = 44.68\mu s$	22,38 KHz
Artix 7	SHA-1	2.154	2.598	$\frac{t^{8b}_{SHA-1}}{80} = 47.62\mu s$	21,00 KHz
Artix 7	MD5	136	341	$\frac{t^{32b}_{MD5}}{64} = 0{,}78\mu s$	1,28 MHz
Artix 7	SHA-1	2.154	2.598	$\frac{t^{32b}_{SHA-1}}{80} = 0{,}63\mu s$	1,6 MHz
Spartan 6	MD5	136	345	$\frac{t^{8b}_{MD5}}{64} = 44.68\mu s$	22,38 KHz
Spartan 6	SHA-1	2.150	2.600	$\frac{t^{8b}_{SHA-1}}{80} = 47.62\mu s$	21,00 KHz
Spartan 6	MD5	136	345	$\frac{t^{32b}_{MD5}}{64} = 0{,}78\mu s$	1,28 MHz
Spartan 6	SHA-1	2.150	2.600	$\frac{t^{32b}_{SHA-1}}{80} = 0{,}63\mu s$	1,6 MHz

pequena de registradores e células lógicas o que caracteriza um circuito dedicado simples que pode ser embarcado em FPGAs pequenas de baixo custo e tamanho.

A Tabela 5.5 apresenta os resultados relativos ao consumo de potência das implementações apresentadas na Tabelas 5.3 e 5.4. Com base nos resultados apresentados em (ATMEL 2005) o ATmega 2560 possui uma taxa de consumo de $\frac{21\,\text{mA}}{16\,\text{MHz}}$ e uma potência dinâmica de 105mW quando alimentado com uma fonte de 5.0V. Considerando os dados obtidos em (C.C. F. Pereira et al. 2017) a potência dinâmica da plataforma Intel Galileo é de 550mW. A potência dinâmica das FPGAs foram calculadas utilizando a ferramenta *XPower Analyzer* da empresa Xilinx.

Tabela 5.5: Resultados relativos as potências dinâmicas por cada plataforma de hardware, considerando o *clock* calculado para alcançar o tempo dos μP de 8 e 32-bits.

Plataforma	*clock*	MD5	*clock*	SHA-1
ATmega 2560	16 MHz	105 mW	16 MHz	105 mW
Intel Galileo Gen 2	400 MHz	550 mW	400 MHz	550 mW
Artix 7	22,38 KHz	$< 10\mu$W	21,00 KHz	$< 10\mu$W
Artix 7	1,28 MHz	$< 10\mu$W	1,6 MHz	$< 10\mu$W
Spartan 6	22,38 KHz	$< 10\mu$W	21,00 KHz	1 mW
Spartan 6	1,28 MHz	1,0mW	1,6 MHz	2.0mW

A partir dos dados calculados para as FPGAs, observa-se um consumo menor que 10μW, para a implementação de ambos os algoritmos na Artix 7, nos tempos das duas plataformas microprocessadas. Na Spartan 6 percebe-se um consumo similar ao da Artix 7 para o MD5 e um consumo bem superior, mas ainda muito reduzido para o SHA-1, como pode ser observado na Tabela 5.5, a qual também pode ser observada no gráfico demonstrado na Figura 5.1, na qual a visualização das diferenças de consumo de ener-

gia tornam-se bem evidentes. Analisando em contraste com o ATmega2560 as soluções de hardware dedicado possuem ganhos, em termos de economia de potência, de mais de 1000×. Já em relação à Intel Galileo Gen 2, observa-se ganhos de até 500×. Essa análise demonstra que existe vantagens consideráveis em termos de economia de energia ao se utilizar soluções em hardware dedicado o que justifica sua aplicação em dispositivos de IoT, por meio de chips específicos para gerar hashes, o que liberaria o processador para execução de outras atividades em paralelo, não relacionadas a geração de hash. Desso modo, além de diminuir o consumo de energia é possível ter um incremento de performance.

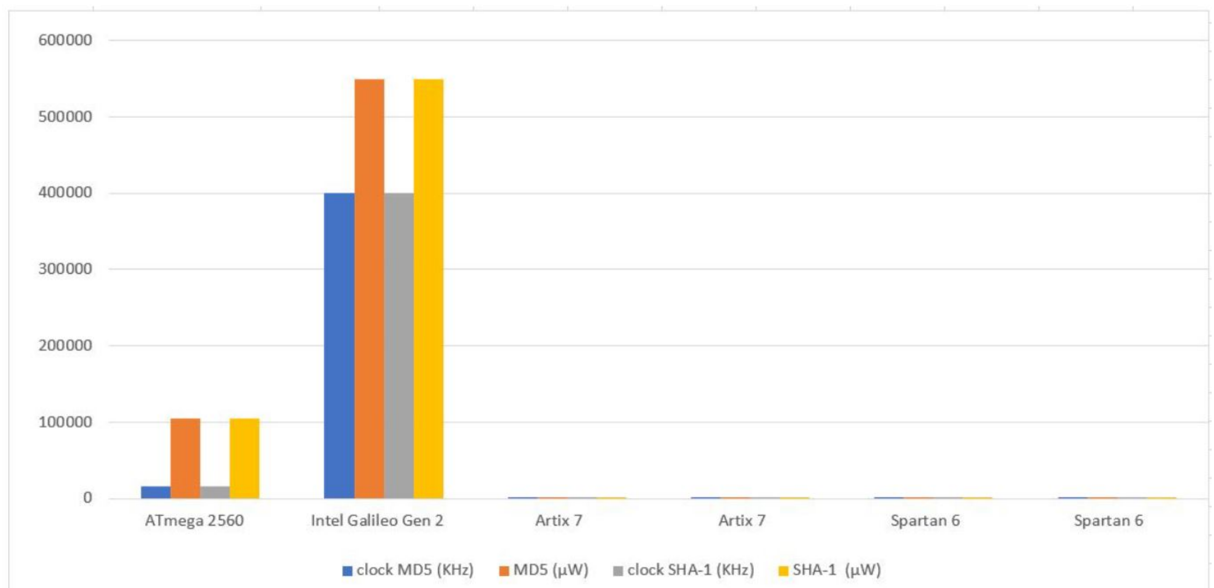

Figura 5.1: Grafico dos resultados de potência dinâmica.

Fonte:Próprio autor

Capítulo 6

Conclusão

Esse trabalho apresentou duas propostas de implementação em hardware para os algoritmos de hash MD5 e SHA-1. Para isso, foram elaborados o capítulo 2, como uma referência para entendimentos dos requisitos básicos de segurança, integridade dos dados e os hashes MD5 e SHA-1 (relatados esmiuçadamente), e o Capítulo 3 com os conceitos de hardware reconfigurável e FPGA.

As estruturas propostas, também chamadas de ASSP, foram sintetizadas em um FPGA (Xilinx Virtex 6 xc6vlx240t-1ff1156) objetivando validar o hardware proposto. Todos os detalhes de implementação descritos no Capítulo 4, foram apresentados e analisados em termos de área de ocupação e *throughput*. Os resultados apresentados no Capítulo 5 são bastante significativos e não foram encontrados resultados semelhantes na literatura, tornando-os inéditos até o momento. Para o MD5 foi alcançado um *throughput* de 140 Gbps utilizando 320 implementações em um único FPGA alvo, já o SHA-1 alcançou um throughput de $28,16$Gbps com 48 implementações. Resultados que apontam para novas possibilidades de utilização de algoritmos de *hash* em hardware dedicado para aplicações de tempo real e de grande volume de dados.

Além desses, foi apresentada uma análise comparativa de diferentes aplicações de hardware para implementações dos algoritmos MD5 e SHA-1. As estruturas comparadas foram um μP de 8 bits associado ao kit ATmega2560, um μP de 32 bits associado ao kit Intel Galileo Gen 2 e um hardware dedicado implementado nos FPGAs da Xilinx, Artix 7 e Spartan 6. Os resultados mostraram ganhos consideráveis de economia de energia em relação ao hardware dedicado quando comparado as outras plataformas testadas. Para o Artix 7 foi possível obter um consumo menor que 10μW, para a implementação de ambos os algoritmos, utilizando os mesmos tempos de processamento das duas plataformas microprocessadas. Com a Spartan 6 o consumo é equivalente ao da Artix 7 para o MD5, porém possui um consumo maior para o SHA-1, entretanto bastante reduzido se comparado as demais plataformas microprocessadas testadas. Com isso, é factível uma economia de potência em mais de $1000\times$ entre a implementação proposta no FPGA em relação ao ATmega2560. Já em relação à Intel Galileo Gen 2, foi atingido ganhos de até $500\times$. As quais demonstraram que existem vantagens relevantes em termos de economia de energia e velocidade ao se utilizar soluções em hardware dedicado, evidenciando aplicações em dispositivos de IoT.

Algumas dificuldades foram encontradas para que tais resultados fossem atingidos, como o total entendimento do algoritmo, o funcionamento bit a bit, pois a implementação

utiliza elementos lógicos, como portas lógicas, que trabalham apenas em níveis lógicos zero e um. Dessa forma, esta compreensão levou certo tempo para ser compreendida. Outra dificuldade encontrada foi em relação a ferramenta utilizada, a qual possui uma limitação da quantidade máxima de entrada para multiplexadores que é de 32 bits. E em ambos algoritmos foi preciso ter um seletor como um multiplexador com mais entradas, 64 para o MD5 e 80 para o SHA-1. Alguns erros foram cometidos durante o processo de desenvolvimento, entre eles o de não compreensão da conversão de caracteres alfanuméricos em ASCII e depois em binário, hexadecimal e decimal da mensagem de entrada do algoritmo, que consumiu um tempo de verificação e adequação correta. O que após análises e correções proporcinou os resultados demonstrados.

Após as implementações efetuadas, experiências e os conhecimentos adquiridos na implementação dos algoritmos MD5 e SHA-1 em FPGA, foi possível perceber a viabilidade e os valores atrelados a essa pesquisa em relação ao desenvolvimento da ciência, quanto a melhoria de padrões no mercado de segurança da informação e dispositivos relacionados à Internet das coisas. Com isso, será dado prosseguimento nos próximos trabalhos com a implementação de outros algoritmos de hash, como o SHA-256 e o SHA-512, os quais fazem parte da família SHA-2, e também o algoritmo mais recente do grupo *Secure Hash Algorithm*, o SHA-3. O que possibilitará novos parâmetros de pesquisa e novas perspectivas de utilização desses algoritmos. (*Lei do Marco Civil da Internet no Brasil* n.d.)

Referências Bibliográficas

Al-Kiswany, Samer, Abdullah Gharaibeh, Elizeu Santos-Neto & Matei Ripeanu (2009*a*), 'On gpu's viability as a middleware accelerator', *Cluster Computing* **12**(2), 123–140.

Al-Kiswany, Samer, Abdullah Gharaibeh, Elizeu Santos-Neto & Matei Ripeanu (2009*b*), 'On gpu's viability as a middleware accelerator', *Cluster Computing* **12**(2), 123–140.

Altera (2012), Reducing power consumption and increasing bandwidth on 28-nm fpgas. **URL:** *https://www.altera.com/en_US/pdfs/literature/wp/wp-01148-stxv-power-consumption.pdf*

ALTERA (2015), Cyclone v device handbook.

ATMEL (2005), *8-bit Microcontroller with 16/32/64KB In-System Programmable Flash*.

C.C. F. Pereira, Geovandro, Renan C. A. Alves, Felipe Da Silva L., Roberto M. Azevedo, Bruno C. Albertini & Cíntia B. Margi (2017), 'Performance evaluation of cryptographic algorithms over iot platforms and operating systems', *Security and Communication Network* **2017**.

Compton, Katherine & Scott Hauck (2002), Reconfigurable computing: A survey of systems and software. acm computing surveys.
URL: *http://www.ee.washington.edu/wp-content/uploads/2016/07/ConfigCompute.pdf*

da Silva, Lucileide, Matheus Torquato & Marcelo A. C. Fernandes (2016), Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço, *em* 'Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016', Recife, PE.

da Silva, Lucileide Medeiros Dantas (2016), Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte - UFRN, Natal, RN.

de Souza, Alisson & Marcelo Fernandes (2014), 'Parallel fixed point implementation of a radial basis function network in an fpga', *Sensors* **14**(10), 18223–18243.

Deepakumara, J., H. M. Heys & R. Venkatesan (2001), Fpga implementation of md5 hash algorithm, *em* 'Canadian Conference on Electrical and Computer Engineering 2001. Conference Proceedings (Cat. No.01TH8555)', Vol. 2, pp. 919–924.

Floyd, Thomas L. (2007), *Sistemas Digitais: fundamentos e aplicações*, 9ª edição, Bookman, Porto Alegre, BRA.

FORBES (2016), 'Why you shouldn't panic about dropbox leaking 68 million passwords', `https://www.forbes.com/sites/thomasbrewster/2016/08/31/dropbox-hacked-but-its-not-that-bad/#b178fec55769`.

He, D. & Z. Xue (2010), Multi-parallel architecture for md5 implementations on fpga with gigabit-level throughput, *em* '2010 International Symposium on Intelligence Information Processing and Trusted Computing', pp. 535–538.

INTEL (2018), Intel fpga acceleration hub, Página na internet, Intel, US.
URL: *https://www.altera.com/solutions/acceleration-hub/overview.html?utm_source=Altera&utm_medium=link&utm_campaign=Homepage&utm_content=Staircase*

ISO/IEC 27000 (2016), 'Information technology — security; techniques — information security; management systems — overview and vocabulary'.

ISO/IEC27002, NBR ABNT (2005), 'Tecnologia da informação — técnicas de segurança — código de prática para controles de segurança da informação'.

Iyer, Nalini C. & Sagarika Mandal (2013), 'Implementation of secure hash algorithm-1 using fpga', *International Journal of Information and Computation Technology* **3**(8), 757–764.

Jarvinen, K., M. Tommiska & J. Skytta (2005), Hardware implementation analysis of the md5 hash algorithm, *em* 'Proceedings of the 38th Annual Hawaii International Conference on System Sciences', pp. 298a–298a.

Jarvinen, Kimmo (2004), 'Design and implementation of a sha-1 hash module on fpgas'.
URL: *http://cwcserv.ucsd.edu/b̃illlin/classes/ECE111/SHA1Javinen.pdf*

Kakarountas, Athanasios P., Haralambos Michail, Athanasios Milidonis, Costas E. Goutis & George Theodoridis (2006), 'High-speed fpga implementation of secure hash algorithm for ipsec and vpn applications', *The Journal of Supercomputing* **37**(2), 179–195.
URL: *http://dx.doi.org/10.1007/s11227-006-5682-5*

Khan, Shahzad, Zain ul Abideen & Shahzad Shahid Paracha (2014), 'An ultra low power and high throughput fpga implementation of sha-1 hash algorithm', *International Journal of Computer Science and Information Security* **12**(8), 80–86.
URL: *http://sites.google.com/site/ijcsis/*

Kim, Keonwoo & Un Sung Kyong (2012), Efficient implementation of md5 algorithm in password recovery of a pdf file, *em* '2012 7th International Conference on Computing and Convergence Technology (ICCCT)', pp. 1080–1083.

Kurose, James F. & Keith W. Ross (2010), *Redes de Computadores e a Internet: Uma abordagem top-down*, trad. 5 ed.ª edição, Addison Wesley, São Paulo.

Lee, Eun-Hee, Je-Hoon Lee, Il-Hwan Park & Kyoung-Rok Cho (2009), 'Implementation of high-speed sha-1 architecture', *IEICE Electronics Express* **6**(16), 1174–1179.

Lei do Marco Civil da Internet no Brasil (n.d.). Disponível em: <http://www.cgi.br/lei-do-marco-civil-da-internet-no-brasil/>. Acessado em 14 jun. 2017.

Lima Filho, Francisco Sales de (2010), Mecanismos de segurança para um sistema cooperativo de armazenamento de arquivos baseado em P2P, Tese de doutorado, Universidade Federal do R io Grande do Norte.

Marks, Michal & Ewa Niewiadomska-Szynkiewicz (2014*a*), Hybrid CPU/GPU platform for high performance computing, *em* '28th European Conference on Modelling and Simulation, ECMS 2014, Brescia, Italy, May 27-30, 2014', pp. 508–514.

Marks, Michal & Ewa Niewiadomska-Szynkiewicz (2014*b*), Hybrid CPU/GPU platform for high performance computing, *em* '28th European Conference on Modelling and Simulation, ECMS 2014, Brescia, Italy, May 27-30, 2014', pp. 508–514.

Michail, H., A. P. Kakarountas, O. Koufopavlou & C. E. Goutis (2005), A low-power and high-throughput implementation of the sha-1 hash function, *em* '2005 IEEE International Symposium on Circuits and Systems', pp. 4086–4089 Vol. 4.

Michail, Harris E., George S. Athanasiou, George Theodoridis, Andreas Gregoriades & Costas E. Goutis (2016), 'Design and implementation of totally-self checking sha-1 and sha-256 hash functions' architectures', *Microprocessors and Microsystems* **45**, 227 – 240.

Michail, H.E., G.S. Athanasiou, G. Theodoridis & C.E. Goutis (2014), 'On the development of high-throughput and area-efficient multi-mode cryptographic hash designs in fpgas', *Integration, the VLSI Journal* **47**(4), 387 – 407.

Morais, D. C., T. W. B. Silva, T. P. Nascimento, E. U. K. Melcher & A. V. Brito (2016), A distributed platform for integration of fpga-based embedded systems, *em* '2016 VI Brazilian Symposium on Computing Systems Engineering (SBESC)', pp. 86–92.

MOTHERBOARD (2016), 'You can now look up your terrible 2006 myspace password'. **URL:** `https://motherboard.vice.com/en_us/article/qkj3n7/myspace-data-breach-427-million-passwords-available-online`

Network Working Group (2001), 'Request for Comments: 3174', http://www.faqs.org/rfcs/rfc3174.html.

NIST (2015), 'Secure Hash Standard (SHS)', http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

NIST (2016), 'Cryptographic standards and guidelines: Secure hashing'.
URL: *"https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values"*

NIST (2017), 'An introduction to computer security: the nist handbook', `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-12.pdf`.

Noronha, Daniel Holanda (2017), Proposta de implementação em fpga de máquina de vetores de suporte (svm) utilizando otimização sequencial mínima (smo), Dissertação de mestrado, Universidade Federal do Rio Grande do Norte - UFRN, Natal, RN.

Noronha, Daniel & Marcelo A. C. Fernandes (2016), Implementação em fpga de máquina de vetores de suporte (svm) para classificação e regressão, *em* 'Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016', Recife, PE.

Rivest, R. (1992), The md5 message-digest algorithm, RFC Editor, United States.

Shi, Zhijie, Chujiao Ma, Jordan Cote & Bing Wang (2012), Hardware implementation of hash functions, *em* 'Introduction to Hardware Security and Trust', Springer, pp. 27–50.

Shirey, Robert W. (2007), 'Internet Security Glossary, Version 2', RFC 4949.
URL: *https://rfc-editor.org/rfc/rfc4949.txt*

SINTEF (2013), 'Big data, for better or worse: 90% of world's data generated over last two years', `www.sciencedaily.com/releases/2013/05/130522085217.htm`.

Stallings, William (2015), *Criptografia e Segurança de Redes: Princípios e Práticas*, 6th[a] edição, Person Education do Brasil, São Paulo, SP, BRA.

Statista (2016), 'Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions)'. Disponível em: <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide//>. Acessado em 14 jun. 2015.

TECHNAVIO (2017), 'Global Field-Programmable Gate Array Market 2017-2021'.
URL: *https://www.technavio.com/report/global-semiconductor-equipment-global-field-programmable-gate-array-market-2017-2021/*

THE REGISTER (2016), Linkedin mass hack reveals ... yup, you're all still crap at passwords, Página na internet, The Register, UK.
URL: *`https://www.theregister.co.uk/2016/05/24/linkedin_password_leak_hack_crack/`*

Torquato, Matheus F. & Marcelo A. C. Fernandes (2016), Proposta de implementação paralela de algoritmo genético em fpga, *em* 'XXI Congresso Brasileiro de Automática (CBA 2016)', Vitória, ES.

Torquato, Matheus Fernandes (2017), Proposta de implementação paralela de algoritmo genético em fpga, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte - UFRN, Natal, RN.

Wang, Yuliang, Qiuxia Zhao, Liehui Jiang & Yi Shao (2010), *Ultra High Throughput Implementations for MD5 Hash Algorithm on FPGA*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 433–441.

WIRED (2016), 'Hack brief: Yahoo breach hits half a billion users'.
URL: *https://www.wired.com/2016/09/hack-brief-yahoo-looks-set-confirm-big-old-data-breach/*

XILINX (2018), 'System generator for dsp'.
URL: *"https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html"*

Yiakoumis, L., M. Papadonikolakis, H. Michail, A. P. Kakarountas & C. E. Goutis (2005), Efficient small-sized implementation of the keyed-hash message authentication code, *em* 'EUROCON 2005 - The International Conference on Computer as a Tool', Vol. 2, pp. 1875–1878.