Processador de Uso Específico para o Algoritmo SHA-1

Carlos E. B. Santos Júnior [1], **Marcelo A. C. Fernandes**[1]

[1]Departamento de Engenharia de Computação e Automação - DCA
Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 1524 – 59.078-970 – Natal – RN – Brazil

ceduardobsantos@gmail.com, mfernandes@dca.ufrn.br

Abstract. *This work proposes a Application-Specific System Processor (ASSP) hardware for the SHA-1 hash algorithm. The proposed hardware was implemented on a Field Programmable Gate Array (FPGA) Xilinx Virtex 6 xc6vlx240t-1ff1156. The processing time (throughput) and occupied area were analyzed for various parallel instance implementations of the SHA-1 hashing algorithm. The results showed that SHA-1 proposed hardware achieved a throughput about* 0.644 *Gbps for a unique instance and about* 28 *Gbps for* 48 *instances of the SHA-1 processor embedded on a FPGA. Several applications as password recovery, password validation and verification of big data integrity can be executed efficiently and fast with SHA-1 ASSP.*

Resumo. *Este trabalho propõe um hardware de aplicação específica (*Application-Specific System Processor, ASSP*) para o algoritmo* Secure Hash Algorithm *1 (SHA-1). O hardware proposto foi implementado em uma* Field Programmable Gate Array *(FPGA) Xilinx Virtex 6 xc6vlx240t-1ff1156. O tempo de processamento (throughput) e a área ocupada foram analisados para várias implementações em instâncias paralelas do algoritmo de hash. Os resultados mostraram que o hardware proposto para o SHA-1 alcançou um throughput de* $0,644 Gbps$ *para uma única instância e um pouco maior que* $28 Gbps$ *para* 48 *instâncias em um único FPGA. Várias aplicações como, recuperação de senha, 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 SHA1.*

1. Introdução

O *Secure Hash Algorithm* versão um, o SHA-1, é um algoritmo utilizado para verificar a integridade de sequências de dados de comprimento variável a partir de uma operação chamada de função *hash*. Uma função *hash* possui como saída um código de comprimento fixo C dada uma mensagem de comprimento variável K como entrada. Pode-se dizer que a saída da função *hash*, também chamada de código *hash*, é uma assinatura da mensagem de entrada, conhecido na literatura como *fingerprint*. Estas características são usadas em Códigos de Autenticação de Mensagem (MAC), principalmente os *Keyed-Hash Message Authentication Code* (HMAC), os quais fazem o amplo uso do SHA-1 [Kakarountas et al. 2006]. Este que é a versão revisada

do SHA-0, substituto do algoritmo MD5 (*Message Digest 5*) em 1995 pelo NIST (*National Institute of Standards and Technology*), sendo então, o SHA-1 publicado como um padrão de processamento de informações federais (FIPS) de número 180-1 [Stallings 2015].

A função hash SHA-1 além de ter sido selecionada para o Algoritmo de Assinatura Digital (DSA), conforme padronizado pela FIPS 186-4 [NIST 2013], atua na verificação de sequência de dados associados a mensagens de protocolos de comunicação, arquivos e armazenamento de senhas e era usada em certificados digitais antes de ser substituída pelo SHA-2. O SHA-1 gera códigos *hash* de $C = 160$ bits para qualquer tamanho de K. Todavia, com o advento das áreas de Big Data, Internet das coisas (*Internet of Things* - IoT) e outras emergentes faz-se necessário que o código *hash* seja gerado de forma rápida para situações associadas a um grande volume de dados e/ou com um custo energético pequeno no casos de dispositivos dentro de uma rede de sensores no contexto de IoT. Assim, este projeto tem como objetivo uma proposta de implementação em hardware dedicado para o algoritmo SHA-1. O hardware proposto pode ser visto como um processador de aplicação específica também chamado na literatura por *Application-Specific System Processor* (ASSP).

O hardware apresentado foi desenvolvido em um FPGA (*Field-programmable gate array*), 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 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 and Fernandes 2014, da Silva et al. 2016, Noronha and Fernandes 2016, Torquato and Fernandes 2016, Shi et al. 2012]. Uma das vantagens em desenvolver circuitos específicos é a redução do *clock* quando comparado a implementações em sistemas com processadores de uso geral ou GPP (*General Purpose Processor*). O algoritmo SHA-1 em hardware poderá ser utilizado no desenvolvimento do ASIC para aplicações em IoT ou com utilização no próprio FPGA objetivando acelerar o cálculo do código *hash* em várias aplicações como recuperação de senha, validação de senha, verificação de integridade em grandes volumes de dados e outros.

2. Trabalhos Relacionados

O trabalho apresentado em [Jarvinen 2004] faz uso de um FPGA Xilinx Virtex-II XC2V2000-6 para implementar o SHA-1 com *Itarative Looping* (IL). Nesta 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. Estas 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 and Mandal 2013] é realizada uma implementação do SHA-1 em um FPGA Xilinx Virtex 5 Xc5vlx50t com linguagem de descrição de hardware *Verilog*. A implementação realizada com IL é semelhante a apresentada em [Jarvinen 2004], todavia, possui uma taxa de ocupação um pouco maior, em torno de 1.351 LC e *throughput* também um pouco maior 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.

Uma 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 implementação do SHA-1 em outras plataformas de hardware podem ser encontrados em [Marks and Niewiadomska-Szynkiewicz 2014, Al-Kiswany et al. 2009] 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.

A proposta aqui desenvolvida utilizou como alvo o FPGA Virtex 6 xc6vlx240t-11156 e o resultados mostraram um *throughput* de 652Mbps para um único módulo SHA-1. A implementação utilizou 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 uma única FPGA gerando um *throughput* de $28,160$ Gbps.

3. Algoritmo SHA-1

O SHA-1 é um algoritmo de *hash* descrito pela FIPS (*Federal Information Processing Standards Publication*) 180-4 [NIST 2015] e pela RFC 3174 [Network Working Group 2001], o qual opera com mensagens de entrada de comprimento variável. Para cada i-ésima mensagem de entrada, $\mathbf{m}_i$, (de comprimento K_i bits), expressa como

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

o algoritmo SHA-1 gera uma mensagem de saída, $\mathbf{h}_i$, chamada de código *hash*, de tamanho fixo $C = 160$ *bits*, caracterizado como

$$\mathbf{h}_i = \left[\begin{array}{cccc} h_0 & h_1 & \ldots & h_{C-1} \end{array} \right] \text{ onde } h_k \in \{0,1\} \, \forall \, k. \tag{2}$$

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 (ou *Append Padding*). Já a segunda, chamada aqui de $\mathbf{v}_i$, possui T bits e é inserida em uma operação chamada de Inserção do Comprimento (ou *Append Length*). 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,

$$L_i = \frac{Z_i}{M} = \frac{K_i + P_i + T}{512}. \tag{3}$$

O pseudo-código apresentado no Algoritmo 1 apresenta a sequência de etapas necessárias para geração do código *hash* e estas etapas estão descritas em detalhes nas subseções a seguir.

Algoritmo 1 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 para**
16: $\quad \mathbf{h}_i \leftarrow \mathit{AtualizaHash}(\mathbf{H}(n))$
17: **fim para**

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, é também chamada de *Padding*, e

é formada por uma palavra binária de P_i bits no qual o bit mais significativo é 1 e o resto dos bits são 0. A geração da mensagem de preenchimento é realizada pela função *GeraçãoPreenchimento*(K_i) apresentada na linha 2 do Algoritmo 1. O cálculo do valor de P_i pode ser expresso por

$$P_i = \begin{cases} 448 - (K_i \bmod 512) & \text{para } (K_i \bmod 512) < 448 \\ 512 - (K_i \bmod 512) + 448 & \text{para } (K_i \bmod 512) \geq 448 \end{cases}, \quad (4)$$

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

$$\mathbf{p}_i = \left[\begin{array}{cccc} p_0 & p_1 & \dots & p_{P_i-1} \end{array} \right], \quad (5)$$

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

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

$$\mathbf{v}_i = \left[\begin{array}{cccc} v_0 & v_1 & \dots & v_{T-1} \end{array} \right] \text{ onde } v_k \in \{0,1\} \, \forall \, k. \quad (6)$$

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,

$$\mathbf{v}_i = Binary(K, T) \quad (7)$$

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 no padrão *big-endian*.

A norma da FIPS 180-4 [NIST 2015], supõem 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 1). Neste trabalho a mensagem $\mathbf{z}_i$ é caracterizada por um vetor de Z_i bits expresso como

$$\mathbf{z}_i = \left[\begin{array}{cccc} z_0 & z_1 & \dots & z_{Z-1} \end{array} \right] \text{ onde } z_k \in \{0,1\} \, \forall \, k. \quad (8)$$

3.3. Inicialização do Código Hash

A inicialização do código *hash* (ver linha 6 do Algoritmo 1) é padronizada pela FIPS 180-4 [NIST 2015] de acordo com as seguintes expressões:

$$\mathbf{ha} = \left[\begin{array}{ccc} h_0 & \dots & h_{31} \end{array} \right] = Binary(1732584193, 32), \quad (9)$$

$$\mathbf{hb} = \left[\begin{array}{ccc} h_{32} & \dots & h_{63} \end{array} \right] = Binary(4023233417, 32), \quad (10)$$

$$\mathbf{hc} = \left[\begin{array}{ccc} h_{64} & \dots & h_{95} \end{array} \right] = Binary(2562383102, 32), \quad (11)$$

$$\mathbf{hd} = \left[\begin{array}{ccc} h_{96} & \dots & h_{127} \end{array} \right] = Binary(0271733878, 32), \quad (12)$$

e

$$\mathbf{he} = \left[\begin{array}{ccc} h_{128} & \dots & h_{159} \end{array} \right] = Binary(3285377520, 32), \quad (13)$$

onde

$$\mathbf{h}_i = \left[\begin{array}{ccccc} \mathbf{ha} & \mathbf{hb} & \mathbf{hc} & \mathbf{hd} & \mathbf{he} \end{array} \right]. \quad (14)$$

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,

$$\mathbf{z}_i = \left[\begin{array}{cccc} \mathbf{b}_0 & \mathbf{b}_1 & \ldots & \mathbf{b}_{L_i-1} \end{array} \right], \tag{15}$$

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

$$\mathbf{b}_j = \left[\begin{array}{cccc} b_{j,0} & b_{j,1} & \ldots & b_{j,M-1} \end{array} \right] \text{ onde } b_{j,k} \in \{0,1\} \forall\, k. \tag{16}$$

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

$$\mathbf{b}_j = \left[\begin{array}{cccc} \mathbf{u}_j[0] & \mathbf{u}_j[1] & \ldots & \mathbf{u}_j[15] \end{array} \right], \tag{17}$$

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

$$\mathbf{u}_j[k] = \left[\begin{array}{cccc} u_j[k,0] & u_j[k,1] & \ldots & b_j[k,31] \end{array} \right] \tag{18}$$

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

3.5. Inicialização das Variáveis Hash 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 caracterizadas neste trabalho como vetores expressos como

$$\mathbf{X}(n) = \left[\begin{array}{cccc} x_0 & x_1 & \ldots & x_{31} \end{array} \right] \text{ onde } x_k \in \{0,1\} \forall\, k, \tag{19}$$

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

$$\mathbf{H}(n) = \left[\begin{array}{ccccc} \mathbf{A}(n) & \mathbf{B}(n) & \mathbf{C}(n) & \mathbf{D}(n) & \mathbf{E}(n) \end{array} \right]. \tag{20}$$

A inicialização destas variáveis, no instante $n = -1$, (ver linha 10 do Algoritmo 1) 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 $\mathbf{E}(-1) = \mathbf{he}$.

3.6. Cálculo da variável 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 1). Em cada n-ésima iteração é calculada uma variável, $\mathbf{w}(n)$ expressa como

$$\mathbf{w}(n) = \begin{cases} \mathbf{u}_j[n] & \text{para } 0 \leq n \leq 15 \\ \mathbf{sw}[n] & \text{para } 16 \leq n \leq 79 \end{cases}, \tag{21}$$

$$\mathbf{Hash/s} = \begin{cases} 1\ hash \text{ — } 9,932 \cdot 80 \cdot 10^{-9} \\ H\ hash \text{ — } 1s \end{cases} \tag{22}$$

$$\mathbf{Hash/s} = 80,548 \text{ Mhash/s} \tag{23}$$

$$\mathbf{Hash/s} = \begin{cases} 48\ hash \text{ — } 10,909 \cdot 80 \cdot 10^{-9} \\ H\ hash \text{ — } 1s \end{cases} \tag{24}$$

$$\mathbf{Hash/s} = 3,52 \text{ Ghash/s} \tag{25}$$

onde

$$\mathbf{sw}[n] = \text{lr}\left(\mathbf{u}_j[n-3] \oplus \mathbf{u}_j[n-8] \oplus \mathbf{u}_j[n-14] \oplus \mathbf{u}_j[n-16], 1\right) \tag{26}$$

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

$$\text{lr}(\mathbf{r}, \mathbf{s}) = (\mathbf{r} \ll \mathbf{s}) \vee (\mathbf{r} \gg (32 - \mathbf{s})), \tag{27}$$

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

3.7. 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 1), expresso como

$$\mathbf{f}(n) = f(n, \mathbf{B}, \mathbf{C}, \mathbf{D}) = \begin{cases} \alpha(n) \text{ para } n = 0 \ldots 19 \\ \beta(n) \text{ para } n = 20 \ldots 39 \\ \gamma(n) \text{ para } n = 40 \ldots 59 \\ \delta(n) \text{ para } n = 60 \ldots 79 \end{cases}, \tag{28}$$

onde

$$\alpha(n) = (\mathbf{B}(n-1) \wedge \mathbf{C}(n-1)) \vee (\neg\mathbf{B}(n-1) \wedge \mathbf{D}(n-1)), \tag{29}$$

$$\beta(n) = \mathbf{B}(n-1) \oplus \mathbf{C}(n-1) \oplus \mathbf{D}(n-1), \tag{30}$$

$$\gamma(n) = (\mathbf{B}(n-1) \wedge \mathbf{C}(n-1)) \vee (\mathbf{B}(n-1) \wedge \mathbf{D}(n-1)) \vee (C(n-1) \wedge D(n-1)) \tag{31}$$

e

$$\delta(n) = \mathbf{B}(n-1) \oplus \mathbf{C}(n-1) \oplus \mathbf{D}(n-1), \tag{32}$$

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

3.8. 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, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$ após o cálculo de $\mathbf{f}(n)$ (ver linha 14 do Algoritmo 1). A atualização destas variáveis é representada pelas seguintes equações:

$$\mathbf{E}(n) = \mathbf{D}(n-1), \tag{33}$$

$$\mathbf{D}(n) = \mathbf{C}(n-1), \tag{34}$$

$$\mathbf{C}(n) = \text{lr}(\mathbf{B}(n-1), 30), \tag{35}$$

$$\mathbf{B}(n) = \mathbf{A}(n-1) \tag{36}$$

e

$$\mathbf{A}(n) = \mathbf{V}(n) + \mathbf{Z}(n) + \mathrm{lr}(A(n-1), 5), \tag{37}$$

no qual,

$$\mathbf{Z}(n) = \mathbf{W}(n) + \mathbf{E}(n-1) \tag{38}$$

e

$$\mathbf{V}(n) = \mathbf{f}(n) + \mathbf{k}(n). \tag{39}$$

O SHA-1 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

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

3.9. 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:

$$\mathbf{ha} = \mathbf{ha} + \mathbf{A}(79), \tag{41}$$

$$\mathbf{hb} = \mathbf{hb} + \mathbf{B}(79), \tag{42}$$

$$\mathbf{hc} = \mathbf{hc} + \mathbf{B}(79), \tag{43}$$

$$\mathbf{hd} = \mathbf{hd} + \mathbf{D}(79), \tag{44}$$

e

$$\mathbf{he} = \mathbf{he} + \mathbf{E}(79). \tag{45}$$

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

$$N_i = L_i \times 80 \tag{46}$$

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

4. Implementação proposta

A Figura 1 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. 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 1, o controle dos dois *loops* (linhas 7 e 11) e a inicialização das variáveis *hash* ($\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ e $\mathbf{E}(n)$) a cada j-ésimo bloco, $\mathbf{b}_j$, através do sinal $\mathbf{h}0$.

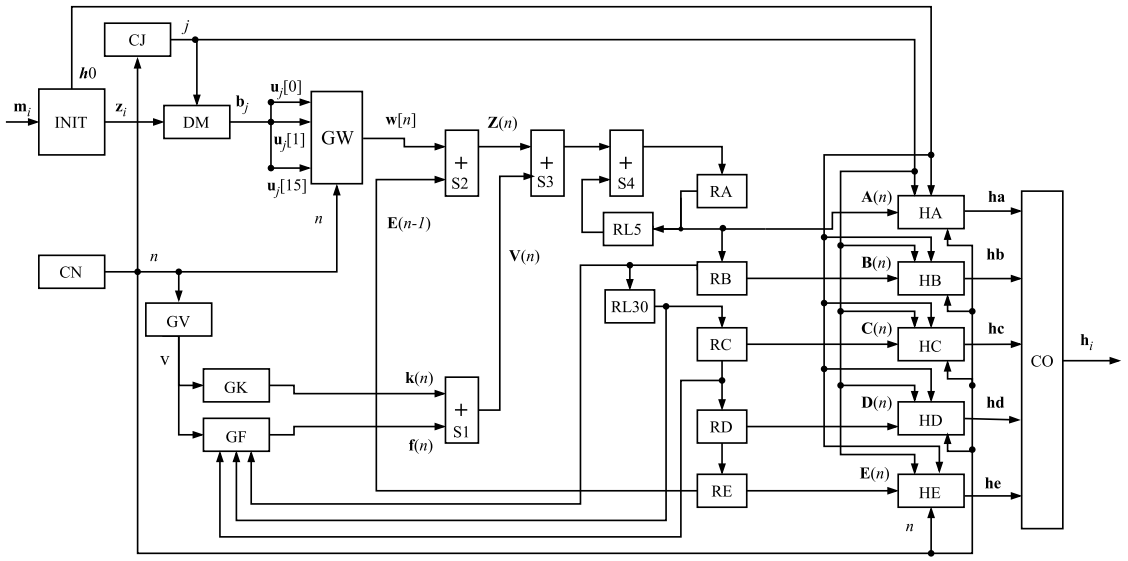

Figura 1. Arquitetura geral da implementação em hardware do SHA-1 proposto.

Os blocos CJ e CN são contadores de $\log_2(L)$ e 7 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 subseção 3.4, o módulo DM 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 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 $\mathbf{u}_j[n]$. Após esta etapa é gerado sinal $\mathbf{w}[n]$ pelo módulo GW (linha 12 do Algoritmo 1) a partir do sinal do contador CN.

Os módulos GF, GK e GW representam as operações expressas pelas Equações 28, 40 e 21, respectivamente. Já o módulos RL5 e RL30 representam operações de leftrotate expressa nas Equações 37 e 35. Observa-se que diferentemente das implementações em processadores sequenciais como GPP, uC (Micro-controladores) e outros, estas equações são executadas em paralelo, acelerando o algoritmo SHA-1. Os detalhes da implementação dos módulos GF, GK, GW, RL5 e RL30 estão detalhados nas subseções a seguir.

4.1. Módulo GF

O módulo GF implementa a função descrita na subseção 3.7 e apresentado na linha 13 do Algoritmo 1. 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 28 e detalhado na Figura 2.

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

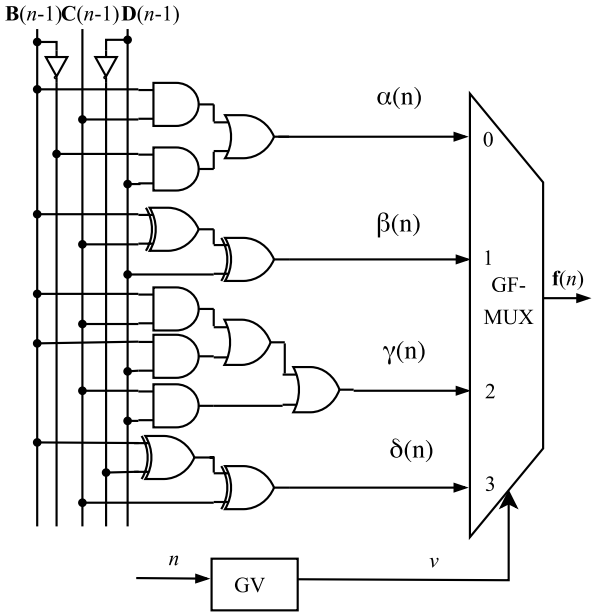

Figura 2. Arquitetura do módulo GF**.**

correspondente a cada intervalo, tendo as seguintes saídas,

$$GV = \begin{cases} 0 & \text{para } n = 0 \ldots 19 \\ 1 & \text{para } n = 20 \ldots 39 \\ 2 & \text{para } n = 40 \ldots 59 \\ 3 & \text{para } n = 60 \ldots 79 \end{cases} . \tag{47}$$

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

4.2. 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 17 da Subseção 3.4, e possui a função de realizar a operação demonstrada pela Equação 21 descrita na subseção 3.6 e linha 12 do Algoritmo 1.

A Figura 3 detalha o módulo que é formado por um multiplexador de 80 entradas, chamado de W-MUX que é selecionado a partir do sinal n. Para os valores de n de 16 a 79 é gerado o sinal $\mathbf{sw}[n]$ expresso pela equação 26 através do módulo SWk onde $k = 16 \ldots 79$, especificado na Figura 4.

Cada k-ésimo módulo SWk é formado por um registrador, chamado aqui de RWk, um módulo de leftrotate (ver Equação 27) 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 26 e o módulo LR1 realiza a função de leftrotate para $\mathbf{s} = 1$. A Figura 5 detalha um módulo genérico associado a implementação do leftrotate com base na Equação 27.

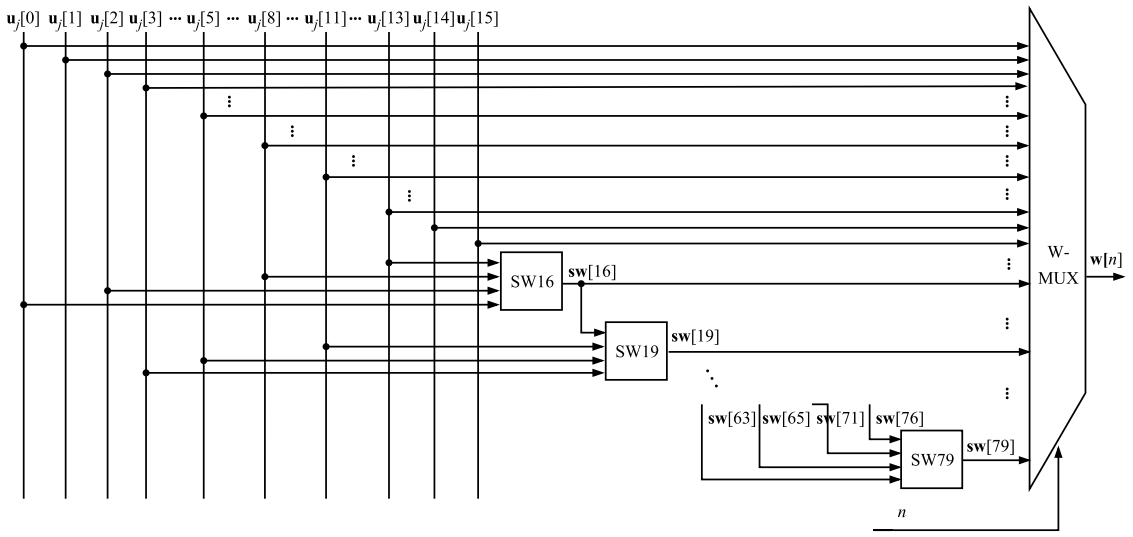

Figura 3. Arquitetura do módulo GW

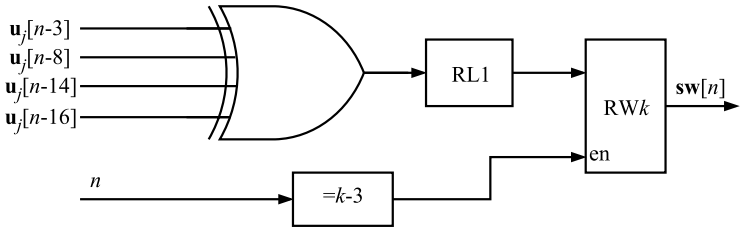

Figura 4. Operações do módulo SWk

Os módulos RL5(s) e RL30 realizam a operação de leftrotate para $\mathbf{s} = 5$ (ver Equação 37) e $\mathbf{s} = 30$ (ver Equação 35), respectivamente. A implementação destes módulos também é caracteriza pela Figura 5.

4.3. 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, 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 38 e 39 e é executado durante a linha 14 do Algoritmo 1. A última etapa de cada n-ésima iteração é a atualização das variáveis *hash*, $\mathbf{A}(n)$, $\mathbf{B}(n)$, $\mathbf{C}(n)$, $\mathbf{D}(n)$ 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, na qual o valor do registrador RC é atualizado por meio da operação $\mathrm{lr}(r, 30)$, expressa em detalhes pela Equação 27. A cada iteração de $\mathbf{n}$ os valores dos registradores deslocam-se entre si atualizando as outras variáveis *hash* de acordo com as Equações 33 a 37. Este procedimento é executado durante a linha 14 do Algoritmo 1.

Ao final das 80 iterações do *loop* em n (linha 11 do Algoritmo 1) as partes que compõem o código *hash*, **ha**, **hb**, **hc**, **hd** e **he** (ver Equação 14), são atualizadas

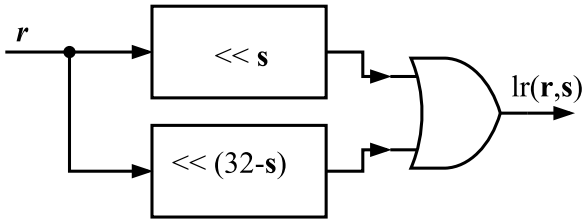

Figura 5. Arquitetura do módulo RLs.

pelos módulos HA, HB, HC, HD e HE, respectivamente. Esta etapa é executada na linha 16 do Algoritmo 1. Finalmente ao final das N_i iterações (ver Equação 46) 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 5 barramentos de 32 bits formados pelos sinais **ha**, **hb**, **hc**, **hd** e **he** e gerar um sinal serial com o código *hash* $\mathbf{h}_i$.

5. Resultados

A Tabela 1 apresenta os resultados obtidos após o processo de síntese do hardware proposto neste trabalho (ver Figura 1). São apresentados resultados relativos a ocupação do hardware no FPGA alvo (Virtex 6 xc6vlx240t-11156) bem como os resultados associados a latência e o *throughput* alcançado após o processo de síntese. Foram gerados resultados para várias implementações em paralelo do algoritmo SHA-1 de acordo com Figura 1 seguindo 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, permitindo acelerar o *throughput* principalmente em casos de recuperação de senha por força bruta, no qual existem um grande número de códigos *hash* a serem gerados.

Tabela 1. 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

A primeira coluna da tabela, chamada de NI, indica o número de implementações paralelas realizadas. A segunda coluna, NR, apresenta o número de registradores utilizados em hardware implementado no FPGA e a terceira coluna, chamada de PR, representa a porcentagem de registradores utilizados em relação ao total disponível no FPGA designado que são de 301.440. Já a quarta e 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 proposto que são de 150.720. Finalmente, a sexta e sétima coluna mostram os resultados, obtidos para várias implementações, de taxa de amostragem, T_s e *throughput*, R_s, respectivamente.

O hardware, desenvolvido de modo paralelo como mostrado na Seção 4, com barramentos de 32 bits para que o tempo de amostragem, T_s, seja correspondente 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 $t_{\text{CLK}} = T_s$. O valores de T_s estão exibidos na sexta coluna da Tabela 1. É possível verificar que não há uma mudança significativa com o aumento de NI, ou seja, para um incremento de 48 vezes de 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 1 e arquitetura apresentada na Figura 1, para cada j-ésimo bloco $\mathbf{b}_j$ de $M = 512$ bits são executadas 80 iterações (ver Equação 46), assim, o *throughput* do hardware proposto pode ser calculado como

$$R_s = \frac{M \times \text{NI}}{80 \times T_s} = \frac{512 \times \text{NI}}{80 \times T_s} = \frac{64 \times \text{NI}}{10 \times T_s}. \tag{48}$$

É importante destacar que valores de *throughput* maiores que 15 Gbps são inéditos 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 no máximo em 20 ms ou um senha também de 6 dígitos alfa numéricos (cada dígito com 62 possibilidades) a partir de um código *hash* no máximo em 17, 4 minutos.

6. Conclusões

Este trabalho apresentou uma proposta de implementação em hardware de algoritmo SHA-1. A estrutura proposta, também chamada de ASSP, foi sintetizada em um FPGA objetivando validar o hardware proposto. Todos os detalhes de implementação da proposta foram apresentados e analisados em termos de área de ocupação e tempo de processamento. Os resultados obtidos são bastante significativos e apontam para novas possibilidades de utilização de algoritmos de *hash* em hardware dedicado para aplicações de tempo real e de grande volume de dados.

Referências

Al-Kiswany, S., Gharaibeh, A., Santos-Neto, E., and Ripeanu, M. (2009). On gpu's viability as a middleware accelerator. *Cluster Computing*, 12(2):123–140.

da Silva, L., Torquato, M., and Fernandes, M. A. C. (2016). Proposta de arquitetura em hardware para fpga da técnica q-learning de aprendizagem por reforço. In *Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016*, Recife, PE.

de Souza, A. and Fernandes, M. (2014). Parallel fixed point implementation of a radial basis function network in an fpga. *Sensors*, 14(10):18223–18243.

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

Jarvinen, K. (2004). Design and implementation of a sha-1 hash module on fpgas.

Kakarountas, A. P., Michail, H., Milidonis, A., Goutis, C. E., and Theodoridis, G. (2006). High-speed fpga implementation of secure hash algorithm for ipsec and vpn applications. *The Journal of Supercomputing*, 37(2):179–195.

Khan, S., ul Abideen, Z., and Paracha, S. S. (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.

Lee, E.-H., Lee, J.-H., Park, I.-H., and Cho, K.-R. (2009). Implementation of high-speed sha-1 architecture. *IEICE Electronics Express*, 6(16):1174–1179.

Marks, M. and Niewiadomska-Szynkiewicz, E. (2014). Hybrid CPU/GPU platform for high performance computing. In *28th European Conference on Modelling and Simulation, ECMS 2014, Brescia, Italy, May 27-30, 2014*, pages 508–514.

Michail, H., Athanasiou, G., Theodoridis, G., and Goutis, C. (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.

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

Michail, H. E., Athanasiou, G. S., Theodoridis, G., Gregoriades, A., and Goutis, C. E. (2016). Design and implementation of totally-self checking sha-1 and sha-256 hash functions' architectures. *Microprocessors and Microsystems*, 45:227 – 240.

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

NIST (2013). Digital signature standard (dss). *FIPS PUB 186-4*.

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

Noronha, D. and Fernandes, M. A. C. (2016). Implementação em fpga de máquina de vetores de suporte (svm) para classificação e regressão. In *Encontro Nacional de Inteligência Artificial e Computacional - ENIAC 2016*, Recife, PE.

Shi, Z., Ma, C., Cote, J., and Wang, B. (2012). Hardware implementation of hash functions. In *Introduction to Hardware Security and Trust*, pages 27–50. Springer.

Stallings, W. (2015). *Criptografia e segurançã de redes*. Pearson Education do Brasil, 6th edition.

Torquato, M. F. and Fernandes, M. A. C. (2016). Proposta de implementação paralela de algoritmo genético em fpga. In *XXI Congresso Brasileiro de Automática (CBA 2016)*, Vitória, ES.

