Desenvolvimento de uma Cifra de Fluxo Baseada em Autômatos Celulares

Flávio du Pin Calmon, Felipe Miranda Costa e Anderson C. A. Nascimento

***Resumo*— Este artigo apresenta o projeto e teste de uma cifra de fluxo baseada em Autômatos Celulares sobre $GF(q)$. O funcionamento da cifra é descrito, junto com o projeto para implementação em FPGA. Além disso, resultados de testes estatísticos são elencados. A cifra pode ser utilizada em cenários que requerem um grande fluxo de dados com pequeno atraso, como aplicações de vídeo e redes de alto desempenho. No melhor conhecimento dos autores, a implementação e análise de uma cifra de fluxo deste tipo nunca foi apresentada na literatura.**

***Palavras-Chave*— Cifra de Fluxo, Autômatos Celulares, Criptografia, FPGA.**

***Abstract*— This paper presents the design and test of a stream cipher based on Cellular Automata over $GF(q)$. The cipher operation is described, along with design guidelines for FPGA implementation. Furthermore, results of statistical tests are shown. This cipher can be used in cenarios that require large data transfer with small delay, such as video applicatios and high-performance networks. To the best of the authors' knowledge, such analysis and implementation are unprecedented in the literature.**

***Keywords*— Stream Cipher, Cellular Automata, Cryptography, FPGA.**

I. Introdução

Cifras de fluxo são uma importante classe de algoritmos criptográficos, tendo aplicações extremamente relevantes em redes de alto desempenho [1]. Seu funcionamento se baseia no ciframento de bits individuais de uma mensagem, usando uma transformação criptográfica que pode ou não variar com o tempo. Uma forma simples de realizar o ciframento de uma mensagem é conhecida como *one-time pad*, que consiste na operação de "ou-exclusivo" entre uma seqüência aleatória, definida como *keystream*, com um texto em claro, i.e., uma seqüencia de bits a ser cifrado. O *one-time pad* é incondicionalmente seguro contra um ataque utilizando apenas texto cifrado [2]. No entanto, o *one-time pad* requer que sua chave seja tão longa quanto a mensagem em claro, dificultando a distribuição e o manejamento de chaves. Esta dificuldade motivou o desenvolvimento de cifras de fluxo onde o *keystream* é gerado de forma pseudo-aleatória, i.e., utilizando um algoritmo determinístico que recebe como parâmetro de entrada uma chave secreta menor. O objetivo deste procedimento é criar um *keystream* que pareça aleatório para um adversário limitado computacionalmente.

Flávio du Pin Calmon está com o Wireless Technology Lab. (WissTek), Universidade Estadual de Campinas, Campinas, SP, Brasil. Felipe Miranda Costa está com o Ochiai Lab, Electrical and Computer Engineering Department, Yokohama National University, Yokohama, Japão. Anderson C. A. Nascimento está com o Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, Brasil. E-mails: fcalmon@wisstek.org, felipemir@gmail.com, andclay@ene.unb.br

A criação de cifras de fluxo baseadas em números pseudo-aleatórios é, no entanto, um processo difícil, tendo em vista que, com o aumento do poder computacional disponível, os esquemas de segurança estão sujeitos a ataques cada vez mais poderosos. Além disso, não existe uma referência para uma cifra de fluxo de alta qualidade, com a maior parte das cifras desenvolvidas sendo proprietárias ou com deficiências conhecidas [3]. Surge a necessidade, assim, de usar estruturas alternativas e inovadoras para o desenvolvimento de cifras de fluxo e, conseqüentemente, geradores de números pseudo-aleatórios.

Neste artigo, apresentamos o projeto de uma cifra de fluxo que busca atender a essas necessidades. A cifra analisada neste trabalho pertence a uma família de cifras introduzida em [4]. Sua elaboração é baseada em técnicas apresentadas recentemente na literatura, que nos permite implementar geradores de *keystream* seguros e eficientes. Essas técnicas incluem o emprego de PCAs (autômatos celulares programáveis) sobre um *corpo de Galois* $GF(q)$ e o princípio do mapeamento não linear variável no tempo. Até o momento no qual esse artigo foi escrito, não havia nenhuma outra proposta de gerador de *keystream* baseado em $GF(q)$, $q > 2$. Assim, o projeto apresentado aqui busca aliar novas formas de ciframento às técnicas de projeto digital que fornecem o desempenho suficiente para as aplicações modernas.

O restante do artigo está dividido da seguinte forma: a seção II apresenta alguns conceitos sobre Autômatos Celulares em $GF(q)$. Em seguida, a seção III descreve o funcionamento da cifra. A seção IV apresenta o algoritmo desenvolvido para determinar as regras de transição do PCA utilizado pela cifra. A seção V descreve o projeto em hardware da cifra de fluxo, visando à implementação em FPGA. Os resultados de testes estatísticos aplicados na seqüência pseudo-aleatória gerada pela cifra estão elencados em VI. Finalmente, na seção VII são apresentadas as considerações finais do trabalho.

II. Autômatos Celulares em $GF(q)$

Definem-se autômatos celulares (CA) como sistemas dinâmicos discretos, cujo comportamento é completamente determinado a partir de relações locais [5]. Cada componente individual do sistema é definido como uma *célula*, podendo assumir uma quantidade finita de estados. A transição entre os estados em uma célula depende do estado de seus vizinhos e, possivelmente, dela mesma, acontecendo em passos discretos de tempo.

Pode-se classificar um CA em relação aos valores que suas células podem assumir. Neste trabalho, considera-se apenas

autômatos celulares cujos estados são elementos de $GF(q)$, onde q é o número de estados assumidos pela célula. Consideramos, também apenas autômatos celulares unidimensionais, onde as células estão dispostas em uma "linha", e cada uma possui uma vizinhança de três células (ela mesma e seus vizinhos imediatos). Uma análise mais profunda das diferentes topologias que um autômato celular pode assumir é apresentado em [6].

Considere um autômato celular unidimensional com N células. Denota-se por $s_i[t]$ o estado da célula i, $i = 1, \ldots, N$, no instante t e por f_i uma função determinística que define o próximo estado que a célula deverá assumir, chamada de *regra* ou *função de transição*. Nesse caso, temos, por definição:

$$s_i[t+1] = f_i(s_{i-1}[t], s_i[t], s_{i+1}[t]) \ . \tag{1}$$

Quando as regras de transição aplicadas a cada célula de um autômato celular são distintas, ele é classificado como *híbrido*. Caso as funções de transição sejam lineares, o autômato celular é *linear*. É possível classificar, ainda, um CA como *programável* (PCA). Nesse caso, a regra pode variar com a evolução no tempo.

O autômato celular considerado ao longo do texto é híbrido, linear e programável, com todas as operações sendo realizadas sobre $GF(q)$. Portanto, ele pode ser analisado utilizando propriedades de máquinas de estados finitos lineares (LFSM). Como o autômato celular é linear, pode-se representar o próximo estado de uma célula como uma combinação linear. Portanto, (1) se torna [7]:

$$s_i[t+1] = c_i s_{i-1}[t] + d_i s_i[t] + b_i s_{i+1}[t] \tag{2}$$

Assim, é possível representar a evolução de um CA pelo produto entre uma matriz de transição e o vetor $s[t]$, onde a matriz possui a forma (considerando um fronteira nula):

$$A_0 = \begin{pmatrix} d_1 & b_1 & 0 & \cdots & 0 & 0 \\ c_2 & d_2 & b_2 & \ddots & 0 & 0 \\ 0 & c_3 & d_3 & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ddots & d_{n-1} & b_{n-1} \\ 0 & 0 & 0 & \cdots & c_n & d_n \end{pmatrix} \tag{3}$$

O autômato celular considerado terá comprimento máximo caso a seqüência de estados $s(0), s(1), s(2)...s(0)$ incluir todos os estados não nulos para qualquer $s(0)$. Por ser um LFSM sobre $GF(q)$, isso ocorre se, e somente se, a matriz de transição possuir um polinômio característico primitivo [7], onde o polinômio característico é dado por:

$$p(x) = |xI - A_0| \, . \tag{4}$$

III. Funcionamento da Cifra

A cifra desenvolvida é um *one-time pad* baseado em um gerador de números pseudo-aleatórios. O gerador é uma máquina de estados finitos que retorna uma seqüência de elementos de $GF(2^l)$. Os principais elementos desse gerador são:

1) Autômato celular em $GF(q)$ programável linear (q primo) com L células e polinômio característico primitivo;
2) RAM com 2^l células cada uma delas com um elemento de $GF(2^l)$, $2^l < q$;
3) ROM com N diferentes regras de transição para PCA em $GF(q)$, cada uma delas correspondente a um polinômio característico primitivo, $1 \leq N \leq 2^l$;
4) Circuito lógico simples para controle.

A chave secreta determina o estado inicial do PCA e o estado inicial da RAM. O comprimento do corpo q é restrito a ser primo, uma vez que um CA que respeite as limitações da matriz de transição citadas acima, e que funcione sobre um corpo cujo comprimento não é primo, é redutível [7].

Para facilitar a melhor compreensão desta seção, adotam-se as seguintes notações: (i) PCA_i representa o conteúdo da i-ésima célula do PCA que é um elemento de $GF(q)$, $i = 1, ..., L$, (ii) RAM(a_i) é o conteúdo do endereço a_i da RAM, $i = 1, ..., 2^l$ e (iii) SWAP(RAM(ads_i),RAM(ads_j)) denota a operação de mudança dos conteúdos da RAM localizados nos endereços ads_i e ads_j.

O gerador funciona sincronamente com o *clock* através dos passos (*rounds*), cada um deles inclui L ciclos de *clock*, de acordo com a descrição contida na seção abaixo. Lembramos que a cifragem ocorre com a realização da operação "ou-exclusivo" entre um elemento do *keystream* e o texto em claro.

A. O passo do gerador

Cada passo do gerador consiste em duas etapas :

1) Transição do PCA do estado presente para o seu próximo estado
   a Seleciona-se uma regra de transição para o PCA como o conteúdo da ROM no endereço especificado por (RAM(ads) $\mod N$), considerando o estado atual da RAM. ads é um parâmetro fixado, $0 \leq ads \leq 2^l - 1$. Na implementação, utilizou-se $ads = 0$;
   b Atualiza-se o estado atual do PCA empregando a regra de transição selecionada.
2) Geração dos símbolos do *keystream* pelo seguinte procedimento. Para $i = 1, 2, ..., L$:
   a Altera os conteúdos da RAM de acordo com o seguinte procedimento:
   Se ($PCA_i \mod 2^l$) = PCA_i, rotacione os elementos presentes na RAM por PCA_i posições,
   - Para $j = 0, 1, ...C - 1$:
   SWAP(RAM(ads_j),RAM(ads_k)),
   $k = 2^l - 1 - j$, onde cada ads_j e ads_k são o j-ésimo e k-ésimo elementos da RAM e C é um certo parâmetro da cifra;
   b Se $PCA_{(i+L/2) \mod L} = (PCA_{(i+L/2) \mod L}) \mod 2^l$, gera-se um elemento do *keystream* como o conteúdo da RAM no endereço $ADS = PCA_{(i+L/2) \mod L}$ e prossiga. Caso contrário, prossiga sem gerar nenhum elemento do *keystream*.

B. Considerações sobre parâmetros da cifra

Os diferentes parâmetros L, q, l, N, ads e C especificam cifras distintas da família de cifras propostas em [4]. Neste mesmo trabalho, sugere-se que as seguintes condições sejam satisfeitas, $l \geq 8$, $L \geq 30$, $N \geq L$, $c > l$, $2^l/q \leq 0,75$. No projeto desenvolvido, buscou-se escolher os valores dos parâmetros que facilitassem a implementação futura em hardware. Os valores escolhidos para os parâmetros foram: $l = 8$, $L = 32$, $N = 64$ e $C = 32$. O parâmetro q tinha que atender ao requisito $2^l/q \leq 0,75 \Rightarrow q >= 2^8/0.75 \Rightarrow q \geq 341,33$. Definiu-se, então $q = 347$, o primeiro primo que satisfaz essa desigualdade.

Na implementação aqui proposta, utilizou-se uma chave secreta de 256 bits. Para inicializar o PCA, a chave foi dividida em 32 partes de 8 bits e, com o valor contido em cada uma dessas partes, inicializou-se cada uma das células do PCA. Garantiu-se, dessa forma, a inicialização de todas as células do PCA. Em relação ao estado inicial da RAM, foi utilizado o *peso de Hamming* e rotacionada a RAM esta quantidade de vezes. Para uma implementação com fins comerciais, recomenda-se que um esquema mais completo seja elaborado, utilizando-se, além da chave secreta, um *vetor de inicialização* que varie de instantes em instantes, evitando que duas mensagens em claro sejam cifradas com um mesmo *keystream*.

C. Buffer de saída

Uma das fraquezas da cifra proposta é a taxa não constante na qual os elementos do *keystream* são gerados. Essa irregularidade está intimamente relacionada ao estado em que a cifra se encontra, revelando informações que podem ser utilizadas em eventuais ataques. Para contornar esse problema, sugere-se que seja adicionado à estrutura um buffer de saída, dimensionado de forma que ele esteja cheio a maior parte do tempo.

IV. Determinando as regras de transição

Sugere-se em [4] que a matriz de transição do CA com n células, apresentada em (3) tenha as seguintes propriedades:

1) $b_i = 1$, $1 \leq i \leq n-1$,
2) $c_i = -1$, $2 \leq i \leq n$, onde -1 representa o inverso aditivo do elemento identidade da operação de multiplicação em $GF(q)$,
3) $d_i \in \{0,1\}$, $1 \leq i \leq n$,
4) A quantidade de coeficientes d_i iguais a 1 deve ser minimizado.

Essas quatro condições simplificam a implementação do autômato celular, visto que o cálculo do próximo estado requer apenas operações de soma e subtração. As três primeiras condições para os coeficientes impõem ainda que q seja primo [7]. Na implementação, utilizou-se regras de evolução onde a diagonal de (3) continha um *peso de Hamming* menor ou igual a 5.

O problema de determinar as regras de transição se reduz, portanto, a encontrar os valores da diagonal da matriz A_0. Na seção II, foi discutido que, para o CA considerado ter período máximo, sua respectiva matriz de transição deve possuir um polinômio característico primitivo. Como um período grande é uma característica extremamente desejável para as seqüências geradas por uma cifra de fluxo, buscou-se matrizes A_0 de forma que seu polinômio característico fosse primitivo.

Algoritmo 1: Rotina criada para encontrar as regras de transição.

Entrada: Valor do primo q de $GF(q)$, peso máximo da diagonal p_{max}, número de matrizes que serão analisadas N.

Saída: Imprime os valores das diagonais D que geram matrizes da forma A_0 com polinômio característico primitivo.

Dados: $P(x)$ é um polinômio sobre $\mathbb{Z}_q[x]$ e A é uma matriz da forma apresentada em (3).

```
1  início
2     Inicialização() ;
3     para i de 1 a N faça
4        A ⟵ PróximaMatriz(p_max);
5        D ⟵ Diagonal(A) ;   // Diagonal(A)
         retorna a diagonal da matriz.
6        P(x) ⟵ |xI − A| ;          // Equação 4
7        se ChecarPrimitividade(P(x)) = 1 então
8           imprimir D;
9        fim
10    fim
11 fim
```

Nota-se que, por A_0 depender apenas dos valores de sua diagonal, cujo peso deve ser minimizado, o espaço de matrizes que podem ser utilizadas é reduzido. Tendo isso em vista, uma forma para encontrar as regras de transição é realizar uma busca por todo o espaço de matrizes que satisfazem as condições desejadas. Esse método, em uma percepção inicial, pode parecer precipitado e ingênuo. No entanto, como é necessário um número pequeno de regras, a aplicação de um algoritmo não probabilístico é, intuitivamente, um caminho mais seguro do que depender de uma busca probabilística.

O Algoritmo 1 apresenta o pseudocódigo simplificado do algoritmo para busca de regras de transição. Ele é composto por duas rotinas principais, descritas a seguir.

1) Verificando se um polinômio é primitivo: O algoritmo desenvolvido foi fundamentado em [1]. Considerando os parâmetros utilizados na cifra, verifica-se que o algoritmo para determinar se um polinômio é primitivo (ou a primitividade do polinômio) enfrenta três obstáculos: (i) a determinar se um polinômio é irredutível, (ii) a operação módulo envolvendo um monômio de grau muito elevado e (iii) a fatoração de um número grande. As duas primeiras operações, para as dimensões consideradas na cifra, não são suportadas por linguagens de programação tradicionais sem o uso de bibliotecas especiais. Portanto, as operações modulares entre polinômios e a verificação de irredutibilidade foram abordados usando a biblioteca NTL para C++ [8].

2) Inicialização do algoritmo: A inicialização dos parâmetros do algoritmo requer a fatoração de um número

com mais de 80 dígitos ($q^L - 1$). Para isso, foi utilizado o pacote GMP (*GNU Multi Precision Arithmetic Library*), uma biblioteca gratuita para aritmética de precisão arbitrária [9], junto com o ECM 6.0.1 [10], uma biblioteca que implementa fatoração usando o método da curva elíptica baseada no GMP.

O método da curva elíptica é amplamente utilizado para fatorar números grandes. Contudo, ele falha para os fatores primos de menor dimensão ($< 10^4$). Portanto, foi desenvolvida uma rotina que verifica quais dos primeiros 1200 números primos são fatores de ($q^L - 1$), usando os métodos encontrados no GMP. Depois de determinados os fatores de menor dimensão, aplica-se o método da curva elíptica para obter os fatores restantes. Foi possível fatorar números da ordem de 10^{81} em poucos segundos.

V. A ESTRUTURA EM HARDWARE

O projeto da cifra em hardware se concentrou em tentar paralelizar ao máximo a operação dos módulos, visando à síntese em FPGA. O objetivo desta seção é apresentar as estruturas de forma genérica, independente de qualquer linguagem de descrição de hardware (HDL) específica. Acredita-se, assim, que um projeto para uma HDL pode ser facilmente derivado a partir das estruturas apresentadas.

A apresentação da estrutura será dividida em quatro partes: (i) a RAM, (ii) o PCA, (iii) o buffer de saída e (iv) a ROM.

A. Uma RAM rápida

O módulo da cifra referido como a RAM, descrita na seção III, apresenta algumas barreiras para a criação do seu modelo em hardware. O nome RAM vem do fato que, por armazenar um vetor de elementos que podem ser acessados a qualquer instante, essa estrutura pode ser facilmente implementada usando uma memória RAM. No entanto, o uso de uma memória de acesso seqüencial se contrapõe à tentativa de paralelizar as funções dos módulos. De fato, a operação de "*swap*" iria necessitar de, pelo menos, um ciclo de *clock* para cada acesso à posição da RAM. Considerando que para os parâmetros utilizados são necessários 32 *swaps* para cada ciclo de operação da cifra, isso geraria uma freqüência de geração do *keystream* demasiadamente pequena. Conseqüentemente, a estrutura da RAM deve ser implementada diretamente usando lógica digital. No caso do FPGA, isso significaria que o módulo seria mapeado nos elementos lógicos disponíveis no chip.

Para conciliar a complexidade requerida pelas duas operações da RAM (*swap* e rotação) com a velocidade buscada no projeto em hardware, criou-se uma nova estrutura, batizada de FORAM (*Fast Operational RAM*). A idéia subjacente a este módulo é a observação de que tanto a operação de rotação quanto a de *swap* podem ser realizadas apenas com os endereços das posições de memória, e não com os elementos armazenados em si. Isso decorre diretamente do fato das operações realizadas na RAM resultarem apenas em permutações dos elementos de 0 a $2^l - 1$.

A rotação pode ser realizada somando (módulo 2^l) o valor da rotação com um ponteiro de endereço. O *swap*, conforme utilizado no algoritmo de ciframento, envolve trocar o elemento em uma posição ads, $ads < C$, com a posição $2^l - 1 - ads$. Considerando que o endereço ads possui representação binária $b_l b_{l-1} \ldots b_0$, o elemento que estava em ads passa a ser armazenado em $\overline{b}_l \overline{b}_{l-1} \ldots \overline{b}_0$ após o *swap*. Em outras palavras, utilizando apenas os ponteiros de endereço, a troca entre os registradores executado na RAM pode ser feita através de uma comparação e, em seguida, a inversão dos bits da representação binária do endereço. Por exemplo, suponha que seja desejado fazer a troca entre os valores do registrador na posição 3 com a posição $2^l - 4$. Considerando $l = 8$ o elemento que está no endereço 3 (00000011_b) passaria a ser armazenado na posição 252 (11111100_b).

A *FORAM* é formada por 2^l módulos básicos, compostos por um registrador de l bits e uma lógica adicional. Esses módulos, assim como na concepção original da RAM, armazenam os números de 0 a $2^l - 1$. No entanto, esses valores representam um endereço, e não um elemento propriamente dito. O valor armazenado no módulo de posição 0, por exemplo, indica o endereço onde o elemento 0 estaria armazenado caso a RAM fosse implementada da maneira tradicional, i.e., como um vetor de 2^l elementos onde as operações de permutação seriam realizadas.

Cada registrador da *FORAM* possui uma entrada de 8 bits, 2 bits de controle, 1 bit para habilitar a escrita e apenas uma saída, como ilustrado na Figura 1. O sinal de controle pode indicar uma entre as quatro operações listadas a seguir. Todas as operações são síncronas com a entrada de relógio (*CLK*). As funções executadas por cada registrador da FORAM são:

1) *Armazenar valor no registrador* – Utilizada quando a cifra é iniciada, essa operação carrega o valor encontrado na entrada *DATA_IN* para o registrador interno de l bits. A escrita é permitida apenas quando o sinal *EN* estiver habilitado
2) *Rotacionar* – Essa instrução soma o valor na entrada *DATA* com o valor do registrador, armazenando o resultado no próprio registrador.
3) *Swap* – Cada registrador realiza a comparação entre o número armazenado no seu registrador (val) e C. Caso $val < C$ ou $val > 2^l - 1 - C$, os bits armazenados são invertidos. É importante destacar que, caso C seja uma potência de 2, a comparação pode ser consideravelmente simplificada. A comparação pode ser realizada de maneira assíncrona, não dependendo da entrada de dados.
4) *Retornar elemento* – Quando se coloca um endereço na entrada da FORAM, o valor armazenado naquela posição deve ser retornado. No módulo, isso é feito através de uma comparação entre a palavra binária no registrador e o valor na entrada, lembrando que um registrador armazena um endereço, e não propriamente um dado. Se ambos os valores são iguais, o *flag* na saída é colocado em 1. Caso contrário, o valor do *flag* é 0.

A estrutura final da FORAM, apresentada na Figura 1, é composta por 2^l módulos (registradores) idênticos, com as entradas de dados, relógio e bits de controle sendo diretamente conectadas às entradas do FORAM (*DATA_IN*, *CLK* e *CONTROL*, respectivamente). Os sinais *flag* são ligados a um

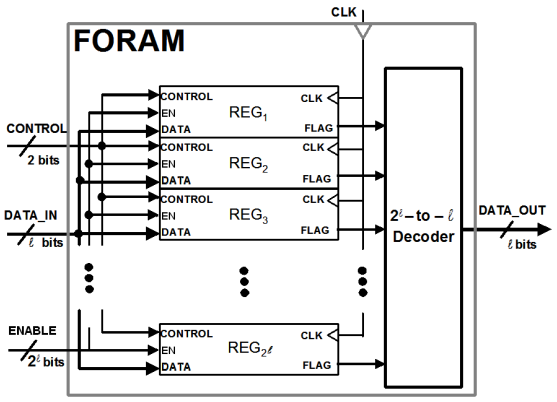

Fig. 1. Estrutura do FORAM (*Fast Operational RAM*), usada para paralelizar as operações realizadas na RAM.

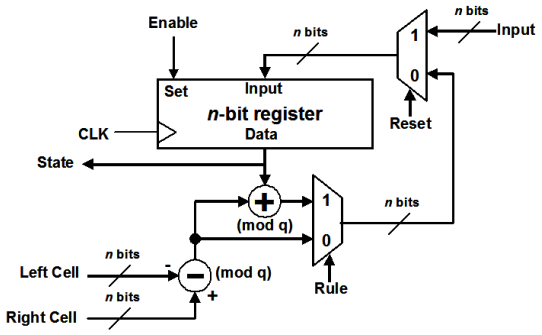

Fig. 2. Esquemático da célula.

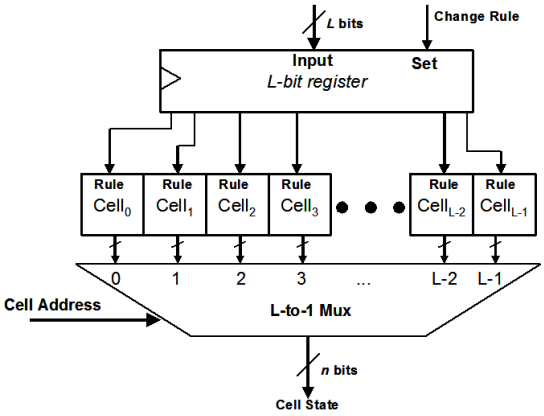

Fig. 3. Esquemático ilustrando a estrutura do PCA.

decodificador, cuja saída é indicada por *DATA_OUT*.

A entrada para habilitar a escrita (*ENABLE*) foi representada utilizando 2^l bits, mas poderia ser reduzido para l bits ao inserir um decodificador apropriado. No entanto, como essa entrada é utilizada apenas na inicialização da cifra, poderia-se ajustar cada registrador para iniciar com um valor determinado e ajustar a posição inicial da FORAM utilizando comandos de rotação e *swap*.

Para retornar um elemento armazenado em uma determinada posição, o endereço é colocado na entrada *DATA_IN*, e os bits de controle são ajustados para o valor apropriado. Os módulos REG_i, $0 \leq i < 2^l$, realizam a comparação com os números em seus registradores e, considerando que a FORAM foi inicializada corretamente, apenas um módulo terá sua saída em nível lógico alto. O decodificador retorna na saída *DATA_OUT* a posição do módulo com o *flag* ativado, que, por sua vez, é o valor armazenado no endereço indicado.

Foi possível, assim, criar uma estrutura para a RAM que realiza as operações necessárias de rotação e troca de elementos entre registradores em apenas um ciclo de clock cada. Claramente, o ganho no desempenho vem em detrimento do número de recursos requisitados para a implementação. É necessária, conseqüentemente, uma complexidade consideravelmente maior do que seria encontrada caso fosse utilizada uma memória comum. No entanto, o aumento dramático na velocidade justifica esse custo.

B. Estrutura do PCA

O autômato celular é composto por um conjunto de células básicas interconectadas entre si. O esquemático da célula é apresentada na Figura 2.

A célula básica é composta por um registrador de n bits, onde $n = \lceil \log_2(q) \rceil$, e lógica de controle adicional. Como o CA é programável, ela possui uma entrada para a regra (*Rule*), que representa um bit correspondente ao valor d_i na diagonal da matriz de transição. Além disso, são utilizadas entradas para os estados das células vizinhas e um sinal de *reset* para carregar no registrador um número pré-determinado. A saída da célula é seu estado atual, i.e., o valor armazenado no registrador.

É importante notar que a soma e a subtração são realizadas módulo q. Dependendo da implementação de ambas as operações, pode ser necessário ou não fornecer o valor de q externamente para a célula. Encontra-se na literatura formas simples e eficientes de realização de aritmética modular em hardware, possibilitando que a soma e a subtração sejam calculadas localmente em cada célula [11].

A estrutura final do autômato celular é formada concatenando L células. Um registrador temporário de L bits deve ser criado para armazenar o valor da regra de transição (no caso a diagonal da matriz em (3)). Para selecionar a saída de uma célula específica, sugere-se o uso de um multiplexador. Um esquemático simplificado da estrutura final com apenas alguns sinais é apresentado na Figura 3.

C. O buffer de saída

O buffer de saída, usado para normalizar a taxa de geração do *keystream*, é uma estrutura amplamente desenvolvida em hardware, não sendo necessária uma descrição detalhada de sua operação. Existe uma vasta literatura sobre a implementação de buffers, inclusive em linguagens de descrição de hardware [12].

Para o uso na cifra, não é preciso fazer um tratamento caso ocorra um *overflow*, visto que alguns elementos do *keystream* podem ser descartados. No entanto, é fundamental que haja uma sinalização de controle para indicar quando o buffer está vazio, evitando possíveis erros no processo de ciframento. Obviamente, a taxa de geração do *keystream* deve

ser maior que a de dados do texto em claro. Estratégias para seu dimensionamento podem ser encontradas em [13].

D. A ROM

O módulo que armazena os valores das regras de transição, i.e., as diagonais da matriz em (3) é chamado de ROM. Como as regras são parâmetros fixos da cifra, sugere-se que esse módulo seja implementado diretamente em uma memória. Essa abordagem é viável, especialmente considerando que as placas de desenvolvimento para FPGAs geralmente disponibilizam uma RAM *off-chip* de alguns megabytes.

VI. Testes e resultados

O principal estudo realizado em cifras de fluxo é a análise das propriedades estatísticas da seqüência pseudo-aleatória que ela gera. O *keystream* deve ter um comportamento semelhante a de uma seqüência verdadeiramente aleatória, ou a cifra estará sujeita a ataques que exploram as características estatísticas dos dados cifrados [1]. Nesse sentido, foi utilizada a bateria de testes estatísticos *NIST 800-22* [14].

O pacote de testes estatísticos do NIST é formado por um conjunto de 15 testes que foram desenvolvidos para testar a aleatoriedade de uma seqüência binária arbitrariamente grande. Essas seqüências podem ser produzidas por geradores de números aleatórios ou pseudo-aleatórios criptográficos em software ou em hardware. Os testes buscam uma variedade de tipos diferentes de desvios da aleatoriedade ideal que podem existir em uma seqüência.

Cada teste retorna um valor que é uma função dos dados de entrada. Esse valor é utilizado para calcular um parâmetro *P*, referido como *P-value*, que indica a probabilidade com que um gerador de números aleatórios ideal teria produzido uma seqüência "menos aleatória" que a seqüência testada. Um valor *P* igual a 1 indica que a seqüência, para aquele teste, possui uma aleatoriedade perfeita. Por outro lado, um valor *P* igual a 0 indica que a seqüência é completamente não-aleatória. Um nível de significância α pode ser escolhido para analisar os resultados dos testes. Se o valor *P* é maior ou igual a α, considera-se que a seqüência aparentemente é aleatória.

Realizou-se os testes para uma seqüência de 1Gb, utilizando o software fornecido em [15], considerando $\alpha = 0.01$, seguindo a metodologia sugerida em [14]. A cifra foi inicializada com uma chave de 32 caracteres '1' (00110001_b) em ASCII. Escolheu-se esta chave por ela inserir um elevado grau de redundância no estado inicial da cifra. Os resultados obtidos estão presentes na Tabela I, onde "A" e "R" indicam aprovação e reprovação em um determinado teste, respectivamente. Os testes indicaram que a cifra, pelos padrões do NIST, é um gerador de números pseudo-aleatórios de alta qualidade.

TABELA I
RESULTADOS DO TESTE DO NIST.

Nome do Teste	Blocos aprovados (%)	A/R
Frequency (Monobit)	0.987	A
Frequency Test within a Block	0.993	A
Cumulative Sums (Cusums)	0.9875	A
Runs Test	0.987	A
Longest-Run-of-Ones in a Block	0.989	A
Binary Matrix Rank	0.993	A
Discrete Fourier Transform	0.992	A
Non-overlapping Temp. Matching	0.9901	A
Overlapping Template Matching	0.99	A
"Universal Statistical" Test	0.991	A
Approximate Entropy Test	0.987	A
Random Excursions Test	0.9911	A
Random Excursions Variant Test	0.9939	A
Serial Test	0.9915	A
Linear Complexity Test	0.98	A

VII. Conclusões

Foi apresentado o projeto de uma cifra de fluxo baseada em autômatos celulares sobre $GF(q)$. A implementação em FPGA permite que seja criada uma cifra eficiente e portável, com aplicação direta na segurança de diversos tipos de sistemas de comunicação que exigem altas taxas de transmissão. Além disso, através de um conjunto de testes amplamente utilizado, verificou-se que a cifra apresenta excelentes propriedades estatísticas.

Referências

[1] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*. Boca Raton,EUA: CRC Press, Inc., 1996, disponível on-line: *http://www.cacr.math.uwaterloo.ca/hac/*.

[2] C. E. Shannon, "Communication theory of secrecy systems," *Bell System Technical Journal*, vol. 28, 1949.

[3] A. Shamir, "Stream ciphers: Dead or alive?" in *ASIACRYPT*, 2004, p. 78.

[4] M. Mihaljevic and H. Imai, "A family of fast keystream generators based on programmable linear cellular automata over GF(q) and time-variant table," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, no. 1, 1999.

[5] T. Toffoli and N. Margolus, *Cellular Automata Machines*. Cambrige, EUA: MIT Press, 1987.

[6] J. Kari, "Theory of cellular automata: a survey," *Theoretical Computer Science*, vol. 334, no. 1-3, 1995.

[7] K. Cattel and J. C. Muzio, "Analysis of one-dimensional linear hybrid cellular automata over GF(q)," *IEEE Transactions on Computers*, vol. 45, no. 7, 1996.

[8] V. Shoup, *NTL: A library for doing number theory*, disponível on-line: *http://www.shoup.net/ntl*.

[9] *The GNU Multiple Precision Arithmetic Library*, 4th ed., Maio 2006, disponível on-line: *http://www.swox.com/gmp/gmp-man-4.2.1.pdf*.

[10] "GMP ECM 6.0.1," Abril 2005, disponível on-line: *http://www.komite.net/laurent/soft/ecm/ecm-6.0.1.html*.

[11] J.-L. Beuchat, "A vhdl library for integer and modular arithmetic," Ecole Normale Sup´erieure de Lyon, Tech. Rep., 2004, disponível on-line: *http://perso.ens-lyon.fr/jean-luc.beuchat/ArithLib/*.

[12] K. Skahill, *VHDL for programmable logic*, 1st ed. Menlo Park, EUA: Addison Wesley Longman, 1996.

[13] I. Kessler and H. Krawczyk, "Minimum buffer length and clock rate for the shrinking generator cryptosystem," IBM T.J. Watson Research Center, Research Report RC 19938 (88322), 1995.

[14] "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, Special Publication 800-22, 2001, disponível on-line: *http://csrc.nist.gov/rng/SP800-22b.pdf*.

[15] "NIST statistical test suite (version 1.8)," Março 2005, disponível on-line: *http://csrc.nist.gov/rng/*.