UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

MATHEUS MITSUO DE ALMEIDA KOTAKI

Gerador de números aleatórios baseado no k −mapa logístico

São Carlos

2021

MATHEUS MITSUO DE ALMEIDA KOTAKI

Gerador de números aleatórios baseado no k −mapa logístico

Versão Corrigida

Dissertação apresentada à Escola de Engenharia de São Carlos como requisito para obtenção do título de Mestre em Ciências, Programa de Pós-Graduação em Engenharia Elétrica.

Área de Concentração: Processamento de Sinais e Instrumentação

Orientador: Prof. Dr. Maximiliam Luppe

São Carlos

2021

AUTORIZO A REPRODUÇÃO E DIVULGAÇÃO TOTAL OU PARCIAL DESTE TRABALHO, POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Kotaki, Matheus Mitsuo de Almeida
K87g Gerador de números aleatórios baseado no k-mapa logístico / Matheus Mitsuo de Almeida Kotaki; orientador Maximiliam Luppe. São Carlos, 2021.

Dissertação (Mestrado) - Programa de Pós-Graduação em Engenharia Elétrica e Área de Concentração em Processamento de Sinais e Instrumentação -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2021.

1. PRNG. 2. Gerador de números aleatórios. 3. Mapa logístico. 4. Testes do NIST. 5. FPGA. 6. Criptografia de dados. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE JULGAMENTO

Candidato: Engenheiro **MATHEUS MITSUO DE ALMEIDA KOTAKI.**

Título da dissertação: "Gerador de números aleatórios baseado no k-mapa logístico".

Data da defesa: 14/04/2021.

Comissão Julgadora	**Resultado**
Prof. Dr. **Maximiliam Luppe** **(Orientador)** (Escola de Engenharia de São Carlos – EESC/USP)	Aprovado
Prof. Dr. **Breno Ortega Fernandez** (Centro Universitário de Lins/UNILINS)	Aprovado
Prof. Dr. **Paulo Matias** (Universidade Federal de São Carlos/UFSCar)	Aprovado

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica:
Prof. Dr. **João Bosco Augusto London Junior**

Presidente da Comissão de Pós-Graduação:
Prof. Titular **Murilo Araujo Romero**

Aos meus pais Arnaldo e Gláucia

AGRADECIMENTOS

Agradeço primeiramente a Deus pelas oportunidades a mim dadas, e conquistas alcançadas.

Aos meus pais, familiares e amigos, pelo constante apoio e por sempre acreditarem no meu potencial.

Ao amigo e orientador Prof. Dr. Maximiliam, pelas orientações e suporte durante todo o processo.

A todos os professores que contribuíram para a minha trajetória acadêmica desde a minha graduação no Centro Universitário de Lins até o atual momento na Escola de Engenharia de São Carlos.

“Não apenas na pesquisa, mas também no mundo cotidiano da política e da economia, estaríamos todos em melhor situação se mais pessoas percebessem que sistemas não lineares simples não possuem necessariamente propriedades dinâmicas simples ”

Robert May, difusor do mapa logístico [1]

RESUMO

KOTAKI, M. M. A. **Gerador de números aleatórios baseado no k-mapa logístico**. 2021. 98p. Dissertação (Mestrado em Ciências) – Programa de Pós-Graduação em Engenharia Elétrica, Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2021.

O crescente uso de dispositivos de baixo custo e baixa segurança ligados à internet, associados ao conceito de Internet das coisas, tem aumentado a preocupação quanto à possibilidade de invasão de computadores conectados à internet por meio das brechas e vulnerabilidades apresentadas por estes dispositivos. A criptografia de dados é uma forma de aumentar a segurança e a confiabilidade dos dados que trafegam pela internet. As chaves criptográficas são geradas, direta ou indiretamente, por geradores de números aleatórios (RNG). Além desta aplicação, esses geradores também são usados em simulações, recreação e programação. Para avaliar o desempenho de um RNG, usualmente, aplica-se uma bateria de testes compostos por cálculos estatísticos nos bits gerados, como os testes propostos pelo *National Institute of Technology* (NIST). Há duas classes de RNG, o gerador de números aleatórios verdadeiros (TRNG), baseados em fenômenos físicos, como ruído térmico, e o gerador de números pseudoaleatórios (PRNG), baseados em sistemas determinísticos, como o mapa logístico. PRNGs baseados em mapa logístico podem gerar sequências com propriedades estatísticas fracas, entretanto, o k-mapa logístico melhora essas propriedades ao adicionar mais uma etapa à geração de valores do mapa logístico original. Esse mapa descarta k dígitos decimais mais significativos de uma órbita gerada a partir da equação do mapa logístico tradicional. Esse trabalho desenvolveu um PRNG, na Cyclone V SoC FPGA, baseado no conceito do k-mapa logístico utilizando representação em ponto-fixo não sinalizada com 32 bits e $0 \leq k \leq 10$. A partir da síntese do PRNG projetado concluiu-se que possui um *throughput* maior e uso de *hardware* semelhante quando comparado a outros trabalhos. Através de um sistema gerenciado por uma máquina de estados finitos e pelo HPS (*Hard Processor System*) da Cyclone V SoC FPGA, foi possível gerar e armazenar sequências para cada valor de k, e posteriormente testá-las através do conjunto de testes do NIST. Verificou-se que todas as sequências com $k \geq 6$ foram aprovadas em todos os testes. Portanto, o PRNG desenvolvido, com um alto valor de k, consegue gerar sequências com boas propriedades aleatórias.

Palavras-chave: PRNG. Geradores de números aleatórios. Mapa logístico. Testes do NIST. FPGA.

ABSTRACT

KOTAKI, M. M. A. **Random number generator based on k-logistic map.** 2021. 98p. Dissertation (Master's in Science) – Graduate Program in Electrical Engineering, São Carlos School of Engineering, University of São Paulo, São Carlos, 2021.

The increasing use of low cost and low security devices connected to the internet, associated with the IoT (Internet of Things) concept, has raised concerns about the possibility of invasion of computers connected to the Internet through the bottlenecks and vulnerabilities presented by these devices. Data encryption is a way to increase the security and reliability of data that travels over the internet. Random number generators (RNG) generate cryptographic keys, directly or indirectly. In addition to this application, these generators are also used in simulations, recreation and programming. To evaluate the performance of a RNG, a set of tests composed of statistical calculations is usually applied to the generated bits, as the tests proposed by the National Institute of Technology (NIST). There are two classes of RNG, the true random number generator (TRNG), based on physical phenomena, such as thermal noise, and the pseudo random number generator (PRNG), based on deterministic systems, such as the logistic map. PRNGs based on logistic map can generate sequences with weak statistical properties; however, the k-logistic map improves these properties, by adding another step to the original logistic map generation of values. This map discards the most significant k decimal digits of an underlying orbit generated from the traditional logistic map equation. Thus, this map is a great option to implement a PRNG. This work developed a PRNG, in Cyclone V SoC FPGA, based on the concept of the k-logistic map using the 32-bit unsigned fixed-point representation and $0 \leq k \leq 10$. The results from the synthesis of the designed PRNG showed that it has a higher throughput and similar use of hardware when compared to other works. Through a system managed by a finite state machine and the HPS (Hard Processor System) of the Cyclone V SoC FPGA, it was possible to generate and store sequences for each value of k, and then test them using the NIST test suite. All sequences generated with $k \geq 6$ passed all tests. Therefore, the developed PRNG, with a high value of k, can generate sequences with good random properties.

Keywords: PRNG. Random number generators. Logistic map. NIST test suit. FPGA.

LISTA DE FIGURAS

Figura 1 – Geração de bit aleatório com um *latch* RS...28

Figura 2 – LUT *latch*. 29

Figura 3 – Etapa de pós-processamento do TRNG proposto por Hata e Ichikawa [20]. 29

Figura 4 – Conceito do TRNG baseado em TERO. 30

Figura 5 – Condições de colapso do RO e formas de onda. 31

Figura 6 - TRNG baseado em RO com *host processor*. 32

Figura 7 – Formas de onda da operação do TRNG. 33

Figura 8 - Registrador de deslocamento com realimentação linear LFSR. 34

Figura 9 – Diagrama de fase bidimensional do mapa logístico para $3{,}6 \leq \mu \leq 4$..................35

Figura 10 – Gráficos da dinâmica do mapa logístico para diferentes valores de μ. Em (a-f) é apresentado o digrama espaço-tempo e diagrama *cobweb*. Em (g) é apresentado o diagrama de bifurcação. Em (h) é apresentado a curva do expoente de Lyapunov. Em (i) é apresentado uma ampliação em uma região do diagrama de bifurcação. 37

Figura 11 - Representação de um dado em ponto-fixo, com 3 bits na parte inteira e 5 bits na parte fracionária. 38

Figura 12 – Diagrama de bifurcação do mapa logístico para barramentos de tamanho p. 39

Figura 13 – Curvas de distribuição de frequência do mapa logístico para diferentes valores de μ. 40

Figura 14 – Diagrama espaço-tempo do k-mapa logístico para diferentes valores de μ, correspondente às órbitas $k0$, $k1$, $k2$, $k3$ e $k4$...41

Figura 15 – Diagrama *cobweb* do k-mapa logístico para diferentes valores de μ, correspondente às órbitas $k0$, $k1$, $k2$ e $k3$. 42

Figura 16 – Diagrama de bifurcação do k-mapa logístico correspondente às órbitas $k0$, $k1$, $k2$ e $k3$. 43

Figura 17 – Expoente de Lyapunov do k-mapa logístico correspondente às órbitas $k0$, $k1$, $k2$ e $k3$. 44

Figura 18 – Curvas de distribuição de frequências do k-mapa logístico correspondente às órbitas $k0$, $k1$, $k2$ e $k3$, para $\mu = 4$. 44

Figura 19 – Diagrama de Poincaré do k-mapa logístico para $3{,}6 \leq \mu \leq 4$ e $0 \leq k \leq 2$. 45

Figura 20 – Fluxograma genérico dos testes do NIST. 46

Figura 21 – Estrutura geral de uma FPGA. 47

Figura 22 – Modelo da SoC FPGA da Altera. 48

Figura 23 – Placa de desenvolvimento Terasic DE10-Nano. 49

Figura 24 – Estrutura básica do sistema. 50

Figura 25 – PRNG proposto. 51

Figura 26 – Unidade Aritmética. 52

Figura 27 – Unidade de Descarte de Dígitos. (a) UDD com k estágios. (b) Dentro de um estágio k. 54

Figura 28 – Sistema de geração e armazenamento de valores aleatórios. 56

Figura 29 – Máquina de estados finitos do componente FSM. 57

Figura 30 - Curvas de distribuição de frequência de sequências geradas pelo PRNG proposto. 62

Figura 31 – Diagrama de Poincaré do PRNG proposto para $0 \leq k \leq 5$. 63

Figura 32 - Diagrama de Poincaré do PRNG proposto para $6 \leq k \leq 10$. 64

LISTA DE TABELAS

Tabela 1 – Relação estado x componentes. 57

Tabela 2 – Resultado da síntese do sistema de geração e armazenamento. 59

Tabela 3 – Resultado da síntese do circuito do PRNG proposto..60

Tabela 4 – Comparação do uso de *hardware* em trabalhos diferentes....................................61

Tabela 5 – Comparação de frequências máximas e *throughputs* de trabalhos diferentes. 61

Tabela 6 – Resultados do conjunto de testes do NIST. 65

LISTA DE ABREVIATURAS E SIGLAS

µSD	Micro SD
ALM	Adaptative Logic Module
ALU	Arithmetic Logic Unit
ARM	Advanced RISC Machine
ASIC	Application-specific Integrated Circuit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
HP	Host Processor
HPS	Hard Processor System
I/O	Input/Output
IoT	Internet of Things
LED	Light Emitting Diode
LFSR	Linear-Feedback Shift Register
LUT	Look-Up Table
MOSFET	Metal-Oxide-Semiconductor Field Effect Transistor
NIST	National Institute of Standards and Technology
PLD	Programmable Logic Device
PLL	Phase Locked Loop
PRNG	Pseudorandom Number Generator
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RNG	Random Number Generator

RO	Ring Oscillator
SD	Secure Digital
TERO	Transition Effect Ring Oscillator
TRNG	True Random Number Generator
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

SUMÁRIO

1 INTRODUÇÃO 23

1.1 Objetivo da pesquisa 24

1.2 Organização do documento 24

2 GERADORES DE NÚMEROS ALEATÓRIOS 27

2.1 TRNG 27

2.2 PRNG 33

2.2.1 Mapa logístico 35

2.2.2 k-mapa logístico 40

2.3 Conjunto de testes do NIST 45

3 MATERIAIS E MÉTODOS 47

3.1 Arquitetura FPGA 47

3.2 Estrutura básica do sistema 49

4 IMPLEMENTAÇÃO EM *HARDWARE* 51

4.1 Implementação do PRNG proposto 51

4.2 Sistema de geração e armazenamento de valores aleatórios 55

5 RESULTADOS E DISCUSSÕES 59

6 CONCLUSÃO 67

REFERÊNCIAS 69

APÊNDICES 73

APÊNDICE A – Publicação gerada durante o período de mestrado 75

APÊNDICE B – Descrição do PRNG baseado no ***k***-mapa logístico (VHDL) 77

APÊNDICE C – Descrição do componente FSM (VHDL) 83

APÊNDICE D – Descrição da conexão entre os componentes: FSM, RAM, CONTADOR e PRNG (VHDL) 85

APÊNDICE E – Descrição do sistema da figura 28 (Verilog) 89

APÊNDICE F – Código em linguagem C do programa da figura 28 97

1 INTRODUÇÃO

O forte crescimento de aplicações baseadas em Internet das Coisas (IoT) [2]-[4] tem levantado novas preocupações, quanto à segurança das informações que trafegam pela rede, devido à grande quantidade de dispositivos com recursos limitados de *hardware* e, consequentemente, de *software*, conectados a esta. Embora várias ameaças desafiem a segurança da IoT, a principal fonte de desconfiança começa pela segurança do *hardware* [5]–[7]. A criptografia de dados permite aumentar a segurança e a confiabilidade dos dados que trafegam pela internet. As chaves criptográficas, utilizadas nesse processo, são geradas, direta ou indiretamente, por geradores de números aleatórios (*Random Number Generator* – RNG), que além de aplicações em sistemas de segurança também são usados em simulações, tomadas de decisões e recreação. Os RNGs se dividem em duas classes: *True Random Number Generator* (TRNG) e *Pseudorandom Number Generator* (PRNG) [8].

O TRNG utiliza como fonte de entropia, fenômenos físicos, como ruído térmico e metaestabilidade. É imprevisível e aperiódico. Geralmente, os bits gerados por essas fontes estão sujeitos a defeitos estatísticos, como *bias*, ou seja, uma probabilidade maior de gerar zeros ou uns. Logo, para melhorar suas características estatísticas utiliza-se um estágio de pós-processamento, gerando valores verdadeiramente aleatórios [9]. Muitos TRNGs são baseados em RO (osciladores em anel), cujo funcionamento é susceptível a variações de processos de fabricação, temperatura, alimentação. Assim, visando manter a segurança do gerador enquanto em funcionamento, alguns trabalhos abordam a implementação de um sistema de supervisão junto ao RNG, como em [10] e [11].

O PRNG utiliza métodos determinísticos, são mais rápidos e de mais fácil implementação do que o TRNG [9].

Um método de implementação de um PRNG se dá pelo uso de sistemas dinâmicos caóticos não lineares, como o mapa logístico, que sob determinadas faixas de operação se torna caótico e apresenta alta susceptibilidade a pequenas perturbações dos parâmetros iniciais [13]. Há pesquisas como em [12], [13] e [14] que atestam o uso de mapas logísticos para geração de números aleatórios. Entretanto deve-se ter cautela na escolha de seus parâmetros, uma vez que podem levar a gerar números que convirjam a um valor fixo, ou, a gerar uma sequência de valores que se repete, agindo de maneira periódica. Os números gerados por esse tipo de PRNG podem possuir propriedades estatísticas fracas, contudo, são melhoradas pela utilização do k-

mapa logístico proposto em [12]. Tal mapa descarta k dígitos mais significativos à direita do ponto decimal do valor gerado pelo mapa logístico original e mantém os demais. Apesar dessa variação do mapa logístico apresentar um bom funcionamento, nenhuma implementação em *hardware* foi feita.

Ambos os RNGs podem ser implementados em dispositivos lógicos como ASIC (*Application-specific Integrated Circuit*) e FPGA (*Field Programmable Gate Array*), contanto que gerem sequências suficientemente aleatórias. O principal requisito para um RNG é que sua saída possua boas propriedades estatísticas, ou seja, possua valores uniformemente distribuídos e imprevisíveis (um valor não possui relação com os anteriores). Para verificar esses requisitos utilizam-se testes estatísticos, como o NIST SP 800-22 e DIEHARD [9].

1.1 Objetivo da pesquisa

Esse trabalho tem como objetivo apresentar a implementação em *hardware* de um PRNG baseado no conceito do k-mapa logístico, utilizando a representação em ponto-fixo não sinalizada de tal forma que se tenha uma baixa utilização de *hardware*, alta frequência de geração e possua sequências geradas aprovadas pelo conjunto de testes do NIST [8].

O bom desempenho de uma FPGA ao ser utilizada para realizar operações em tempo real e pesados processamentos de dados, além de sua flexibilidade no desenvolvimento de projetos de *hardware*, fez com que essa arquitetura tenha sido escolhida para alcançar tal objetivo.

1.2 Organização do documento

Este documento está organizado em seis capítulos. A partir deste, o segundo capítulo traz uma revisão bibliográfica abordando os geradores de números aleatórios, TRNGs e PRNGs, detalhando principalmente o k-mapa logístico e suas propriedades, e uma breve introdução aos testes do NIST.

No terceiro capítulo são abordados os materiais e métodos, descrevendo a arquitetura FGPA e a estrutura básica do sistema.

Em seguida expõe-se no capítulo quatro a implementação em *hardware* do sistema apresentado no capítulo 3, especificando seus principais componentes e funcionamento.

O capítulo cinco apresenta os resultados obtidos a partir da síntese do PRNG desenvolvido, e de suas sequências geradas, tais como uso de *hardware*, frequência máxima de operação, *throughput*, diagramas de distribuição em frequência e resultados dos testes do NIST. E os compara com trabalhos existentes.

O capítulo seis se restringe às conclusões do trabalho desenvolvido.

Por fim, seguem as referências bibliográficas e os apêndices associados.

2 GERADORES DE NÚMEROS ALEATÓRIOS

Segundo Knuth [15], de certo modo, não existe número aleatório, isto é, não podemos concluir se o número 3 é aleatório. Em vez disso, fala-se de uma sequência de números aleatórios cujas propriedades estatísticas são distribuição uniforme, ou seja, ao referir a uma sequência binária, a probabilidade de se obter o valor 1 ou 0 é de 50%, e independência, isto é, cada número foi obtido por chance, não tendo relação com números anteriores a ele.

Valores obtidos de forma aleatória possuem diversas aplicações, por exemplo: simulação, como quando é utilizado para simular fenômenos naturais; programação de computadores, onde os valores aleatórios são uma boa fonte de dados para teste de algoritmos; tomar decisões, quando se é necessário tomar decisões imparciais; recreação, como roletas e dados em casinos; e criptografia, para segurança de dados [12].

Os RNGs podem ser divididos em duas classes, o TRNG, ou dito gerador de números aleatórios verdadeiros, e o PRNG, gerador de números pseudoaleatórios. E ambas as classes podem ter suas sequências testadas por conjuntos de testes, como por exemplo o NIST SP 800-22.

2.1 TRNG

Um gerador de números aleatórios verdadeiros, ou, TRNG, se baseia em fenômenos físicos, como ruído térmico, e possui tanto implementações analógicas quanto digitais para a geração dos valores. Os bits gerados a partir da fonte de entropia são chamados de *raw bits* e não precisam ser perfeitamente aleatórios, uma vez que passam por uma etapa de pós-processamento que melhora as propriedades aleatórias da sequência gerada [11]. Alguns algoritmos usados como módulo de pós-processamento são: corretor XOR e até mesmo PRNGs, como os baseados em mapa logístico [17].

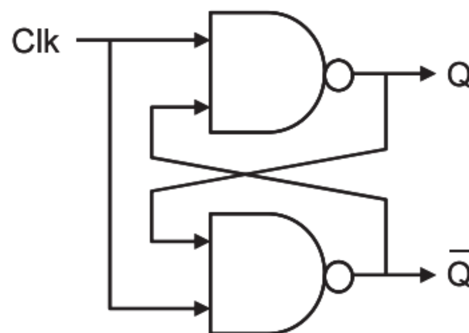
O método mais convencional de uma implementação analógica é amplificar e quantificar o ruído de dispositivos, como ruído em SiN MOSFETs [18], que exibem maior robustez contra a temperatura em comparação com outros mecanismos baseados na amplificação térmica do ruído (por exemplo, resistências) [19].

Implementações digitais de TRNGs utilizam arquiteturas FPGA ou ASIC e métodos baseados em metaestabilidade que podem exibir maior entropia, isto é, desordem, na saída [20], e *jitter*, ou seja, uma variação pequena de um evento em torno do seu valor ideal, em PLL ou osciladores em anel.

Quando as restrições de configuração e tempo de espera de um *latch* são violadas ele apresenta mau funcionamento e a sua saída oscila entre valores lógicos, nesse caso, o *latch* entra em um estado metaestável [20]. Explorando esse fenômeno, é possível criar um TRNG para geração de números aleatórios, já que a saída do *latch* se torna imprevisível.

Hata e Ichikawa [20] desenvolveram uma implementação desse tipo de TRNG, em FPGA, usando *latches* RS que entram em estado metaestável quando as entradas R e S são ativadas simultaneamente. Na figura 1 é apresentado um *latch* RS, cujas entradas estão ligadas a um sinal CLK. Quando $CLK = 0$, o *latch* está estável e as saídas Q e $\bar{Q}$ valerão 1, e quando $CLK = 1$, entrará em um estado metaestável na borda de subida do clk e, eventualmente, se tornará estável em $Q = 1$ e $\bar{Q} = 0$, ou $Q = 0$ e $\bar{Q} = 1$, resultando em uma saída aleatória.

Figura 1 – Geração de bit aleatório com um *latch* RS.

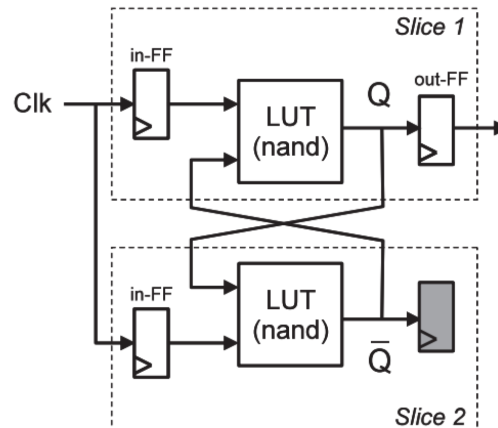


Fonte: [20].

O princípio de um TRNG baseado em metaestabilidade pode parecer simples; porém não é fácil atingir uma alta qualidade de aleatoriedade, porque qualquer tipo de desequilíbrio, por exemplo *clock skew*, ou seja, o mesmo sinal de relógio chegando em componentes diferentes em tempos diferentes, pode levar a um *bias* da saída, isto é, um viés na saída, havendo probabilidade maior de gerar zeros ou uns.

Baseado no *latch* da figura 1, Hata e Ichikawa [20] implementaram, em FPGA, *latches* compostos por duas LUTs (*look-up table*), que chamaram de LUT *latch*es. Na figura 2 é apresentado um LUT *latch*.

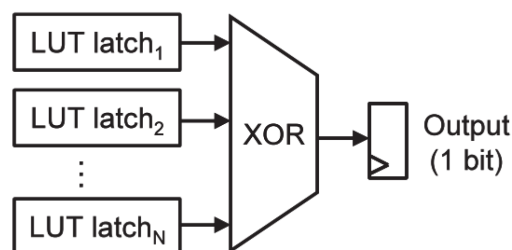
Figura 2 – LUT *latch*.



Fonte: [20].

Como comentado anteriormente, após os dados serem gerados pelo TRNG é necessário utilizar uma etapa de pós-processamento para tornar os bits verdadeiramente aleatórios. Para o projeto de Hata e Ichikawa [20], é realizado a operação XOR entre todos os bits gerados pelos LUT *latches*, como apresentado na figura 3.

Figura 3 – Etapa de pós-processamento do TRNG proposto por Hata e Ichikawa [20].

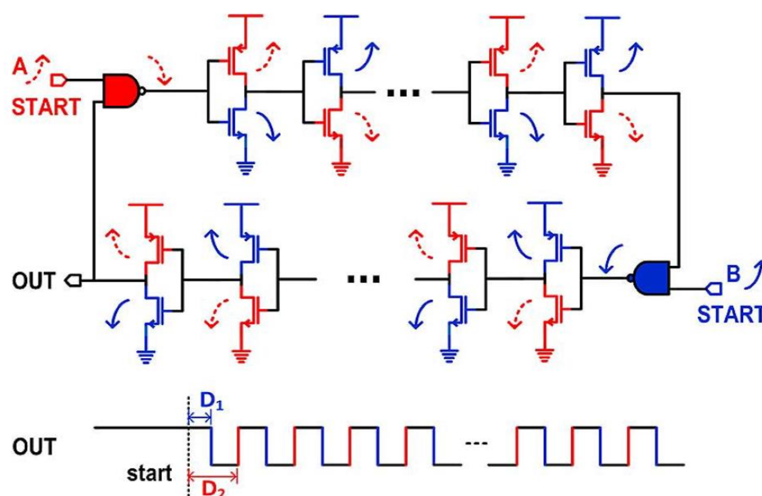


Fonte: [20].

Outra implementação digital é a utilização de *jitter* em osciladores em anel (RO) como fonte de entropia. Alguns métodos de geração de números aleatórios baseados em *jitter* em RO [21]: medem a diferença de frequência de dois osciladores em paralelo, ou, o número de ciclos que o RO leva para colapsar [10].

O oscilador em anel, chamado *Transition Effect Ring Oscillator* (TERO), usado por Yang et al. [10] é apresentado na figura 4 contendo um número par de estágios inversores. Simultaneamente é injetado dois sinais no RO, A e B, através de portas NAND. Por possuir um número par de estágios inversores cada sinal atinge a saída sempre com a mesma borda, subida ou descida. Como observa-se na figura 4, o sinal OUT tem uma borda de descida quando B atinge a saída do RO e uma borda de subida quando A atinge a saída, portanto, OUT oscila. D1 e D2 correspondem respectivamente aos tempos, a partir de quando os sinais B e A foram injetados até que a primeira borda de descida (sinal B) e subida (sinal A) atinjam a saída do oscilador.

Figura 4 – Conceito do TRNG baseado em TERO.

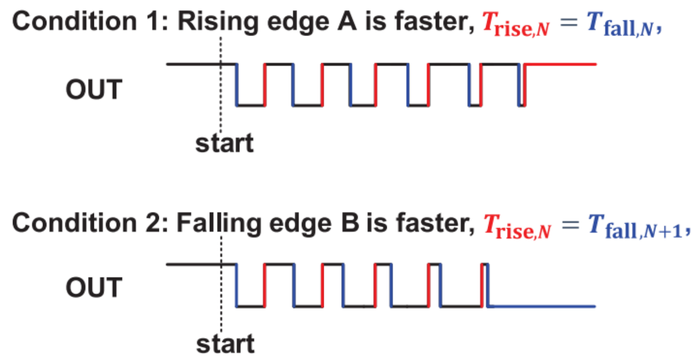


Fonte: [10].

Como os sinais A e B percorrem o RO de forma independente, seus tempos de propagação são diferentes. O tempo para A chegar a OUT é chamado de $T_{rise,N}$ e o tempo para B chegar a OUT é chamado de $T_{fall,N}$. Em ambas as variáveis aparece a letra N, ela indica em qual iteração se calculou o tempo.

Com o passar das iterações N, os valores de $T_{rise,N}$ e $T_{fall,N}$ se alterarão, chegando a uma iteração em que $T_{rise,N} = T_{fall,N}$ ou $T_{rise,N} = T_{fall,N+1}$, colapsando o RO, ou seja, a saída OUT não irá mais oscilar. Tal comportamento é observado na figura 5. O valor que OUT terá quando o RO colapsar depende de qual borda é mais rápida. Se A for mais rápida que B, OUT terá nível alto. Se B for mais rápida que A, OUT terá nível baixo.

Figura 5 – Condições de colapso do RO e formas de onda.



Fonte: [10].

Yang et al. [10] usaram desse RO para criar um TRNG no qual o valor aleatório é gerado a partir do número de ciclos que o RO leva para colapsar. Para gerar os bits aleatórios faz-se um pós-processamento dos dados, no qual extrai-se um determinado número de bits menos significativos para se obter uma maior variação dos valores.

Enquanto o *jitter* aleatório nos osciladores em anel é uma excelente fonte de entropia física para o TRNG, eles são suscetíveis a variações no processo de fabricação, tensão de alimentação e temperatura de operação. Estes RNGs podem sofrer ataques de tensão, temperatura, causando a diminuição da entropia do sistema, comprometendo o seu correto funcionamento. Um tipo de ataque foi apresentado em [22] através da injeção de frequências na fonte de alimentação de um RNG cuja fonte de entropia vem da comparação de *jitter* entre ROs.

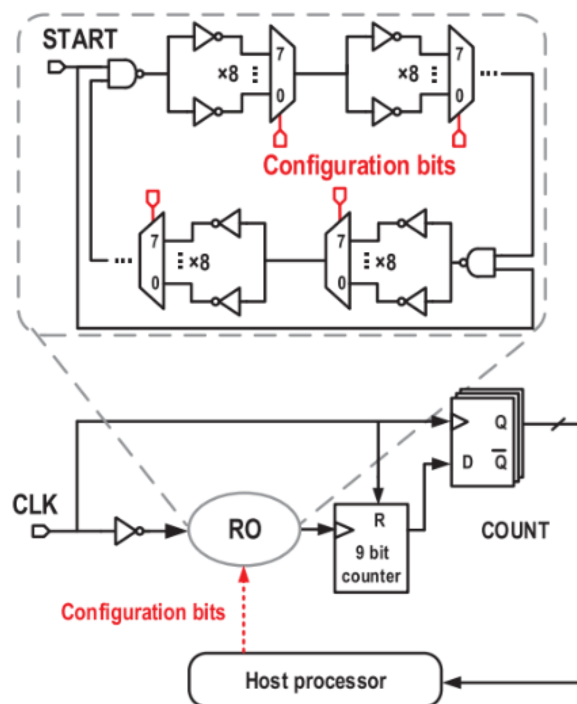
Se nenhuma compensação for feita para combater esses problemas, pode resultar em geração de entropia altamente variável. Tais RNGs necessitam manter os níveis de entropia do sistema altos para garantir a geração de números aleatórios, e isto é feito por meio de circuitos

supervisores. Para tanto, Yang et al. [10] implementaram junto ao TRNG baseado em RO, um sistema de supervisão que chamou de *host processor* (HP), como apresentado na figura 6.

No RO Yang et al. [10] utilizaram oito portas inversoras e um multiplexador em cada estágio inversor, tendo no total 32 estágios inversores. O propósito do multiplexador é selecionar qual dos oito inversores estará atuando. Logo, o TRNG possui $8^{32} = 7{,}8 \times 10^{28}$ possíveis configurações.

O RO inicia-se na borda de descida do sinal CLK, uma vez que na entrada do RO há uma porta inversora. O elemento na sequência do RO é um contador que contará quantas oscilações aconteceram até o RO colapsar. Ao ocorrer a borda de subida do CLK, o valor do contador, ligado à entrada do registrador, é enviado para o *host processor*, e por fim o contador é reinicializado. Esse funcionamento está descrito na figura 7.

Figura 6 - TRNG baseado em RO com *host processor*.

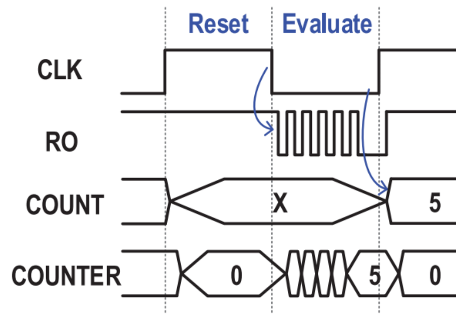


Fonte: [10].

O *host processor* realiza um pós-processamento dos bits recebidos, monitora a entropia dos bits processados, por meio do valor médio da quantidade de ciclos que o RO leva para colapsar, e alimenta um novo conjunto de bits de configuração, em cada multiplexador, para

cada estágio no oscilador em anel, se a entropia medida não for suficiente. Portanto ele consegue identificar quando o TRNG está sendo atacado e reconfigurar, automaticamente, os caminhos físicos do RO, restaurando a média de ciclos e a aleatoriedade desejada.

Figura 7 – Formas de onda da operação do TRNG.



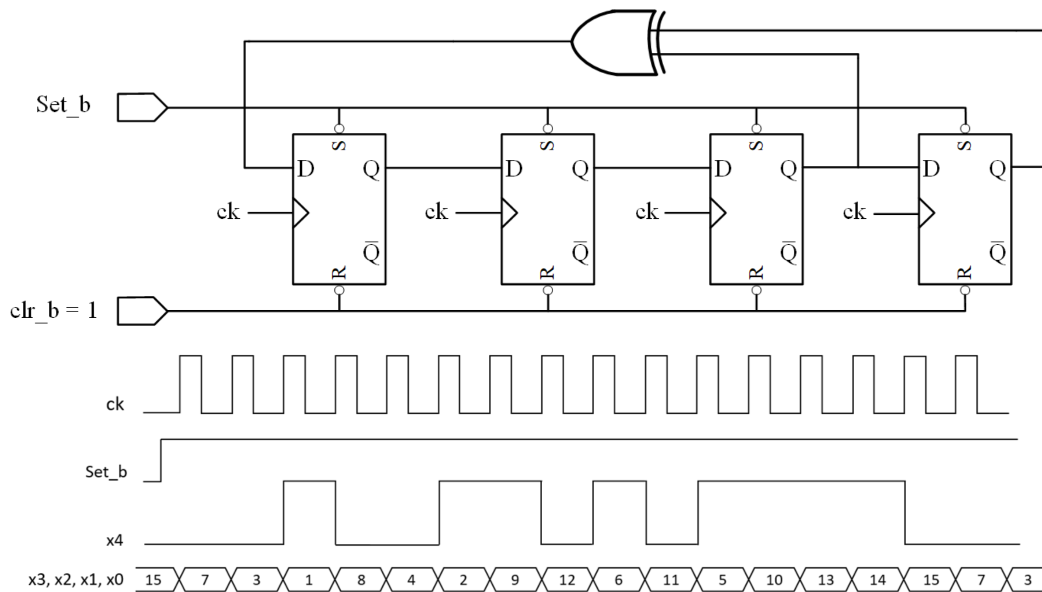
Fonte: [10].

2.2 PRNG

Além dos TRNGs, outra fonte de aleatoriedade muito usada são os geradores de números pseudoaleatórios (PRNG) [6]. Este tipo de gerador utiliza métodos determinísticos e necessita de um valor inicial (semente), a partir do qual são geradas sequências com propriedades estatísticas próximas às esperadas de sequências verdadeiramente aleatórias [12], por isso, se diz que são gerados números pseudoaleatórios. Em relação ao TRNG o PRNG é mais rápido e de mais fácil implementação, entretanto, as sequências geradas pelo PRNG precisam ter longos períodos, já que esse tipo de gerador é computacionalmente periódico [12].

O PRNG mais conhecido é o *Linear Feedback Shift Register* (LFSR), usualmente construído com *flip-flops* tipo D e portas OU-Exclusivo (XOR), que provê maior eficiência e meios econômicos para testar a aleatoriedade [23]. O LFSR apresentado na figura 8 possui quatro estágios, logo, gera 15 padrões pseudoaleatórios. Para iniciar uma sequência de valores, deve ser fornecida um padrão inicial, diferente de “0000”.

Figura 8 - Registrador de deslocamento com realimentação linear LFSR.



Fonte: Adaptada de [24].

Outro tipo de PRNG é baseado em sistemas caóticos. Os sistemas caóticos são sistemas altamente suscetíveis a pequenas perturbações dos parâmetros iniciais [13], que levam a resultados imprevisíveis e aparentemente aleatórios. Sistemas dinâmicos não lineares como mapas iterativos, tal qual, mapa logístico e mapa da tenda, são usados como base para geradores caóticos [12].

Para um sistema dinâmico não linear $f: S \rightarrow S$, tem-se as seguintes notações:

- S é o espaço de fase e contém todos os possíveis estados do sistema;
- n é a quantidade de iterações de evolução do sistema;
- f é a lei de evolução, através dela é possível obter estados futuros a partir dos estado anteriores de forma determinística;
- x_0 é a condição inicial do sistema tal que, $x_0 \in S$;
- $\mathcal{O}(x_0) = \{x_0, x_1, \dots, x_n\}$ é a órbita, sequência de pontos espaço-tempo, gerada por f a partir de x_0;

A taxa de divergência com que duas órbitas de condições iniciais muito próximas se afastam exponencialmente em S após n iterações é medida pelo expoente de Lyapunov, $\lambda(x_0)$, que para sistemas caóticos $\lambda(x_0) > 0$ [12]. Isto é, quanto maior o expoente de Lyapunov maior é sensibilidade a variações de x_0. Tal expoente, para sistemas dinâmicos discretos, como o

mapa logístico, é obtido através da equação (1), adaptada de [12], em que n é a quantidade de iterações calculadas em $f(x_i)$ e $f'(x_i)$ é a primeira derivada da função que representa o sistema.

$$\lambda(x_0) = \frac{1}{n}\sum_{i=0}^{n-1} ln|f'(x_i)| \qquad (1)$$

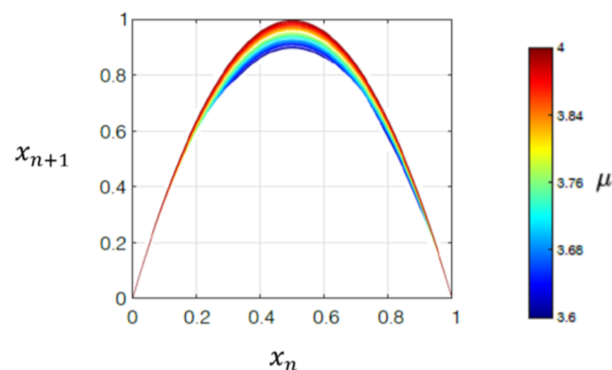
2.2.1 Mapa logístico

O mapa logístico é um modelo matemático bem simples, geralmente usado para descrever o crescimento de populações biológicas [25]. Ele é representado pela equação (2) [12], e seu espaço de fase é definido em $f: [0,1] \rightarrow [0,1]$, para $\mu \in [0,4]$.

$$x_{n+1} = f(x_n, \mu) = \mu x_n(1 - x_n) \qquad (2)$$

Através do diagrama de fases, ou diagrama de Poincaré, apresentado na figura 9, visualiza-se a relação entre x_n e x_{n+1} para diferentes valores de μ, com um valor máximo de x_{n+1} para $x_n = 0{,}5$ e $\mu = 4$. A característica de uma parábola invertida se dá devido ao mapa logístico corresponder à uma equação quadrática [12].

Figura 9 – Diagrama de fase bidimensional do mapa logístico para $3{,}6 \leq \mu \leq 4$.



Fonte: Adaptada de [12].

Outras ferramentas de visualização são apresentadas em [12], elas são, o diagrama espaço-tempo, diagrama de *cobweb* e diagrama de bifurcação.

A primeira apresenta a evolução no espaço de fase do mapa logístico de uma condição inicial x_0.

A segunda é um gráfico, construído no plano x_n, x_{n+1}, que auxilia a visualização da estabilidade ou instabilidade de um sistema dinâmico. A sua construção se inicia traçando uma linha vertical, a partir de uma condição inicial x_0, até encontrar $f(x_0, \mu)$, em seguida, traça-se uma linha horizontal até a função identidade $f(x) = x$ concluindo uma iteração, e assim sucessivamente. Dessa maneira pode-se obter uma órbita iniciada em x_0.

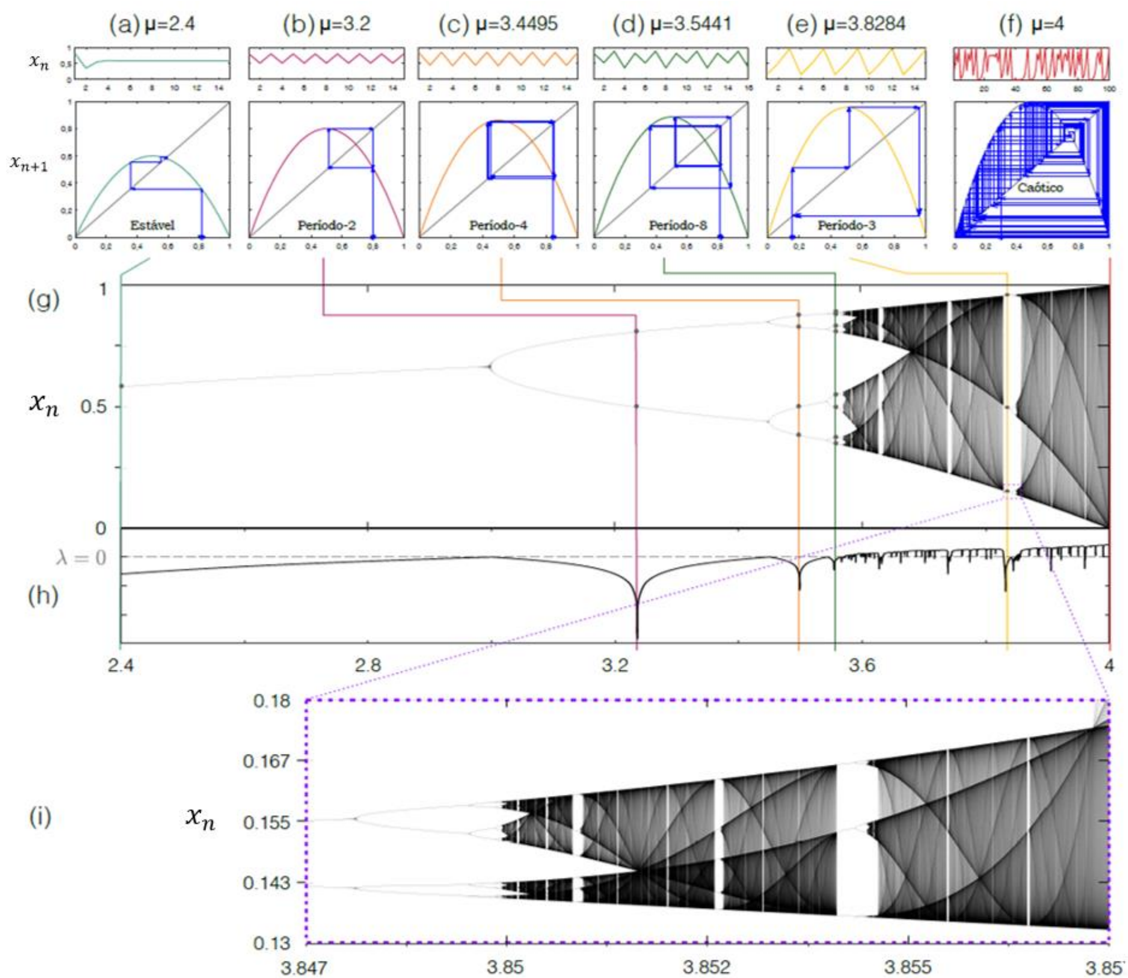
Por último, o diagrama de bifurcação permite analisar o comportamento das órbitas em relação aos valores de x_n quando μ variar.

Na figura 10 são apresentadas as ferramentas de visualização, descritas anteriormente, para valores diferentes de μ. Na figura 10a - 10f observa-se o diagrama espaço-tempo e o diagrama de *cobweb* para diferentes valores de μ. As setas em azul são usadas para representar a órbita a partir de uma condição inicial x_0. Observa-se que na figura 10a é apresentada uma órbita assintoticamente estável que após poucas iterações atinge um ponto fixo. Na figura 10b-e são apresentadas órbitas periódicas, ou seja, repete-se um determinado conjunto de valores. Na figura 10f é mostrado um comportamento caótico no sistema.

Na figura 10g é apresentado o diagrama de bifurcação para $2{,}4 \leq \mu \leq 4$. Nessa figura não é mostrado a região em que $0 \leq \mu < 2{,}4$; para $0 \leq \mu \leq 1$ as órbitas convergem para zero e na região $1 < \mu < 2{,}4$ as órbitas convergem, assim como em $2{,}4 \leq \mu < 3$, para $\frac{\mu-1}{\mu}$. A partir de $\mu > 3$ surgem bifurcações. Observa-se que a sequência das bifurcações leva à criação de órbitas que vão se bifurcando até entrar em regime caótico, aproximadamente perto de $\mu = 3{,}57$. Interessante observar que no meio do caos há rápidas interrupções dele, nas quais há órbitas periódicas, como em $3{,}82 \leq \mu \leq 3{,}86$, conforme observa-se na figura 10e. Pode-se comprovar essas interrupções do caos observando a figura 10h, na qual é mostrado os valores do expoente de Lyapunov para diferentes μ, nota-se nela que no meio dos valores positivos, na região caótica, há valores negativos ou nulos, ou seja, interrupções do caos; já em $\mu = 4$ o sistema atinge o máximo regime caótico com o maior valor do expoente de Lyapunov. Assim, para esse trabalho foi escolhido $\mu = 4$.

Na figura 10i observa-se uma propriedade de um sistema caótico, a auto-similaridade, ao ampliar uma parte da região caótica no diagrama de bifurcação observa-se o padrão do mapa logístico, e se for feito mais uma ampliação será encontrado, novamente, o padrão do mapa logístico e assim sucessivamente.

Figura 10 – Gráficos da dinâmica do mapa logístico para diferentes valores de μ. Em (a-f) é apresentado o digrama espaço-tempo e diagrama *cobweb*. Em (g) é apresentado o diagrama de bifurcação. Em (h) é apresentado a curva do expoente de Lyapunov. Em (i) é apresentado uma ampliação em uma região do diagrama de bifurcação.



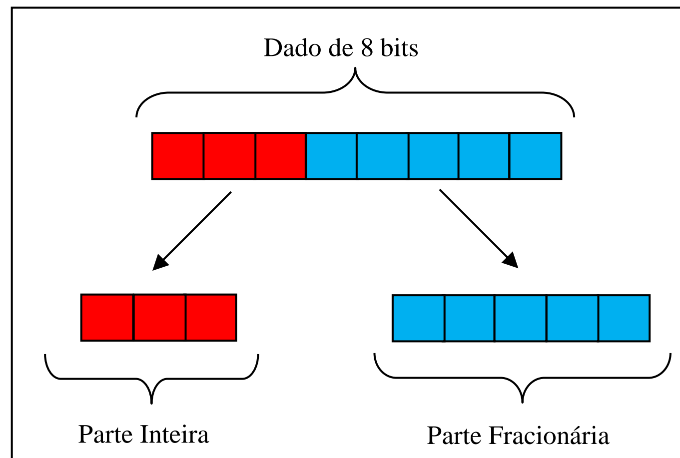
Fonte: Adaptada de [12].

Além dos valores de μ, também deve ser levado em conta, na implementação, a forma de representação dos dados.

Um tipo de representação de valores decimais em binário é a representação em ponto-fixo não sinalizada, na qual possui bits representando a parte inteira e bits representando a parte

fracionária, como apresentado na figura 11. Por exemplo, se o número em binário 10110 estiver seguindo a representação em ponto-fixo de 5 bits, sendo a parte inteira e a parte fracionária representada, respectivamente, por 3 e 2 bits, o seu valor na base decimal é de $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} = 5{,}5$.

Figura 11 - Representação de um dado em ponto-fixo, com 3 bits na parte inteira e 5 bits na parte fracionária.



Fonte: Elaborada pelo autor.

Conforme Sayed et al. [14], o número de bits fracionários implica na caoticidade do sistema, quanto mais bits tiver o barramento, mais caótico fica o sistema.

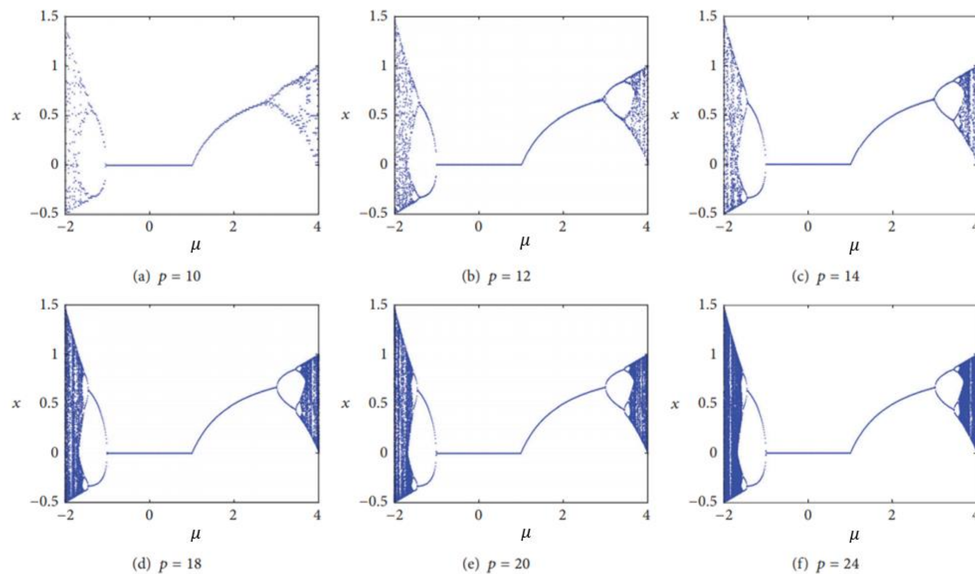
Observa-se na figura 12 os diagramas de bifurcação do mapa logístico, sendo que em cada um deles utilizou-se a representação em ponto-fixo, em que p representa tamanhos diferentes da representação. Nesses diagramas foram apresentados valores negativos de μ, apresentando também a presença de uma região caótica.

A grande vantagem de se utilizar representação em ponto-fixo é que seus cálculos aritméticos são mais rápidos, em *hardware*, do que uma implementação, equivalente, de ponto flutuante, além de que a maioria dos *hardwares* de unidade lógica e aritmética (ALU), como a FPGA, são baseados em ponto-fixo [14].

Tanto Sayed et al. [14], quanto Dabal e Pelka [13], apresentam em seus trabalhos conclusões que testificam que um mapa caótico, mais especificamente o mapa logístico, pode ser usado como um RNG, nesses casos, sendo classificado como PRNG, uma vez que aplicaram testes estatísticos, como os propostos pelo NIST, que possibilitam verificar a aleatoriedade dos

dados obtidos, e obtiveram aprovações neles. Entretanto há pesquisadores que possuem certas restrições quanto a algumas propriedades do mapa logístico, quando utilizado para aplicações criptográficas, como a distribuição de probabilidade não uniforme e a dependência de μ, podendo levar a periodicidade [12].

Figura 12 – Diagrama de bifurcação do mapa logístico para barramentos de tamanho p.



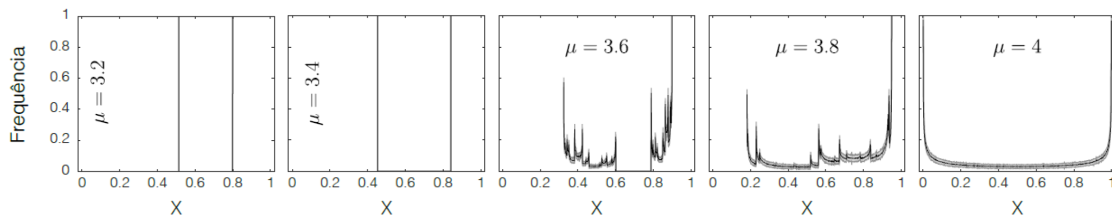
Fonte: Adaptada de [14].

Na figura 13 são apresentadas curvas de distribuição de frequência para cinco valores de μ. Nota-se que quando $\mu = 4$, há uma frequência maior nas bordas $0 \leq x \leq 0{,}1$ e $0{,}9 \leq x \leq 1$ do que na região central, logo não se tem uma distribuição uniforme o que é uma das propriedades desejadas para uma sequência aleatória.

Se o mapa logístico for implementado como um PRNG sem nenhuma modificação ou estágio de pós-processamento, os valores gerados não terão propriedades aleatórias suficientes e sua utilização poderá comprometer sistemas de segurança.

Para melhorar as propriedades estatísticas das sequências geradas por um PRNG baseado em mapa logístico, Machicao [12] propõe uma variação do mapa logístico, chamado de k-mapa logístico.

Figura 13 – Curvas de distribuição de frequência do mapa logístico para diferentes valores de μ.



Fonte: Adaptada de [12].

2.2.2 k-mapa logístico

O k-mapa logístico é uma função que obtém órbitas geradas a partir de uma órbita original do mapa logístico descrito pela equação (2), mantendo $0 \leq \mu \leq 4$ e o espaço de fase no intervalo [0,1]. Ele é descrito pela equação (3), em que $\lfloor\ \rfloor$ é a função $floor$, ou seja, $x_{n+1}10^k$ é arredondado para baixo.

$$x_{n+1}^{k} = x_{n+1}10^{k} - \lfloor x_{n+1}10^{k} \rfloor \tag{3}$$

Na equação (3) x_{n+1}^{k} é o valor formado ao descartar k dígitos mais significativos à direita do ponto decimal do ponto da órbita original x_{n+1} e preservar os demais. Evidentemente, k deve ser menor que a precisão computacional, caso contrário x_{n+1}^{k} será nulo.

Por exemplo, seja $x_{n+1} = 0{,}3695287$, logo $x_{n+1}^{1} = 0{,}695287$ e $x_{n+1}^{3} = 0{,}5287$.

No trabalho de Machicao [12] é realizado uma análise do comportamento dinâmico do k-mapa logístico, como descrito abaixo.

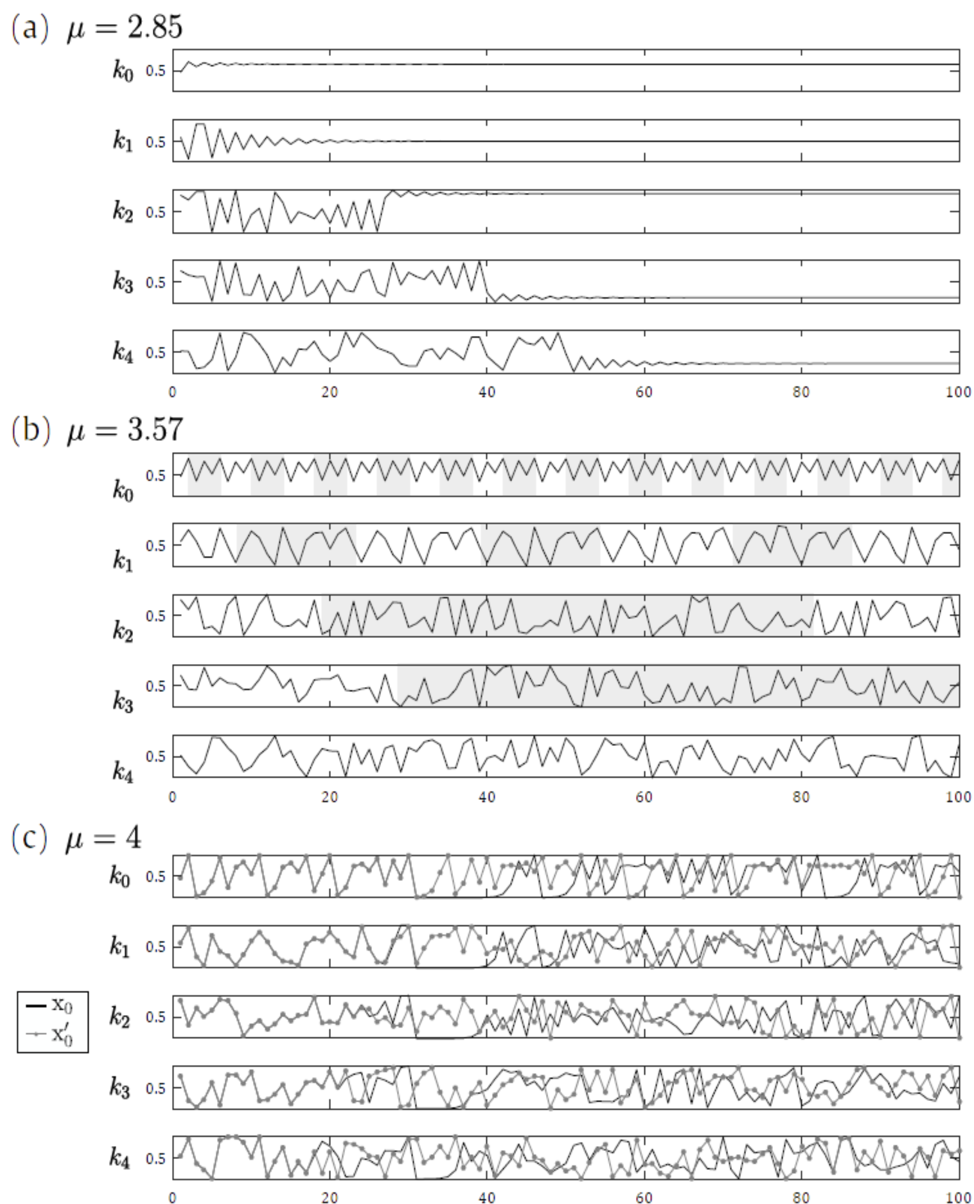
Na figura 14 é apresentado o diagrama espaço-tempo do k-mapa logístico para valores iniciais x_0 e x_0' próximos e μ diferentes durante as primeiras 100 iterações. Importante observar na figura 14 que k_0 implica $k = 0$, e este representa o mapa logístico original.

Na figura 15 observa-se o diagrama de *cobweb* para o k-mapa logístico para diferentes valores de μ.

Nota-se que em 14a tem-se órbitas assintoticamente estáveis, cujo número de iterações necessárias para convergência, para um ponto fixo, aumenta conforme k aumenta, assim como

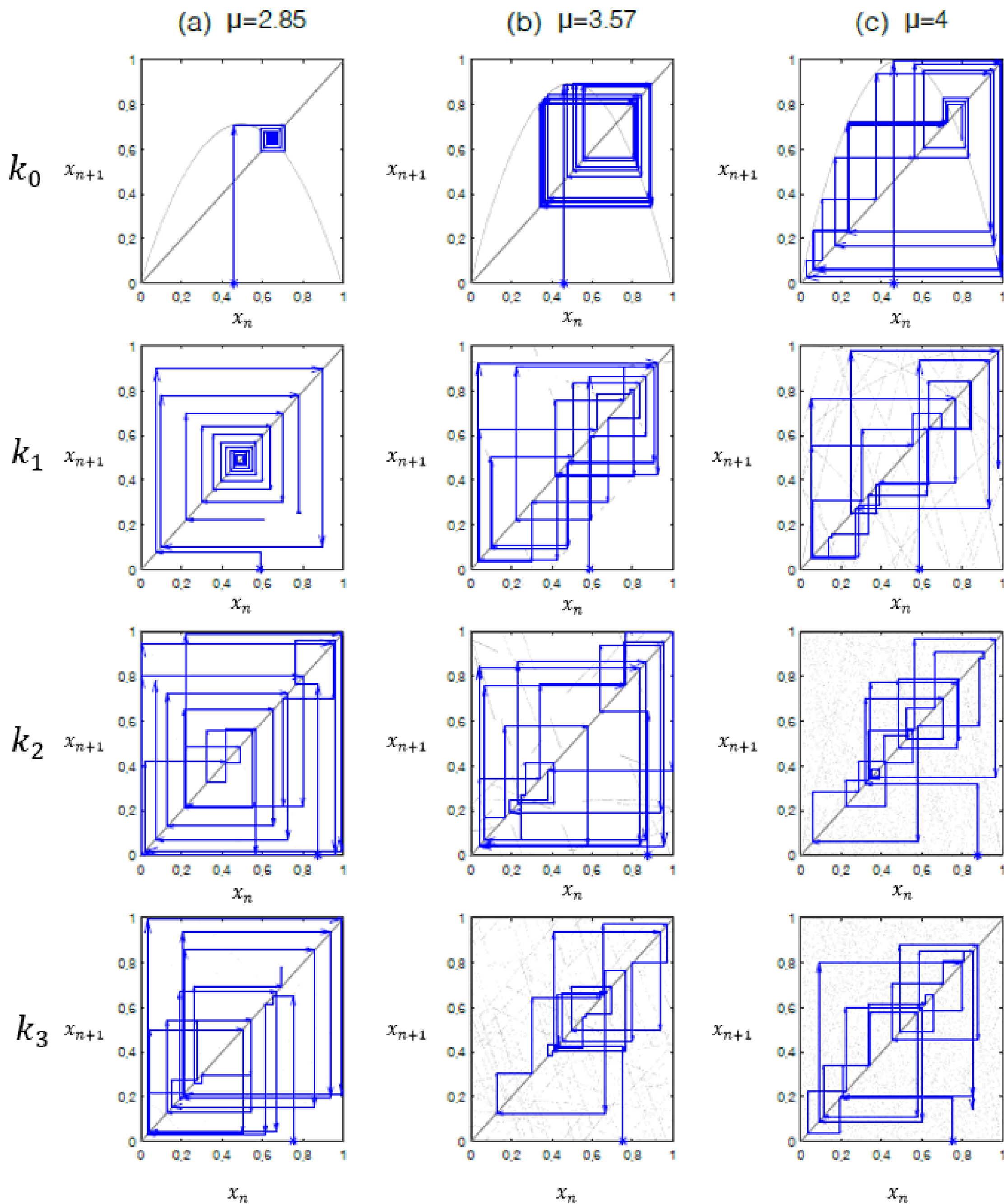
constata-se na figura 15a. Na figura 14b nota-se a preservação da periodicidade e que com o aumento de k aumenta-se o tamanho da janela de periodicidade (em cinza), da mesma forma nota-se em 15b que o número de períodos aumentam conforme k aumenta. Na figura 14c é apresentado duas órbitas com condições iniciais x_0 e x_0', e observa-se que com o aumento de k o número de iterações necessárias para que as curvas divirjam diminui. Ou seja, a sensibilidade do sistema às condições iniciais é aumentada conforme k aumenta. Na figura 15c verifica-se que as trajetórias das órbitas parecem mais aleatórias conforme k aumenta.

Figura 14 – Diagrama espaço-tempo do k-mapa logístico para diferentes valores de μ, correspondente às órbitas k_0, k_1, k_2, k_3 e k_4.



Fonte: [12].

Figura 15 – Diagrama *cobweb* do k-mapa logístico para diferentes valores de μ, correspondente às órbitas k_0, k_1, k_2 e k_3.

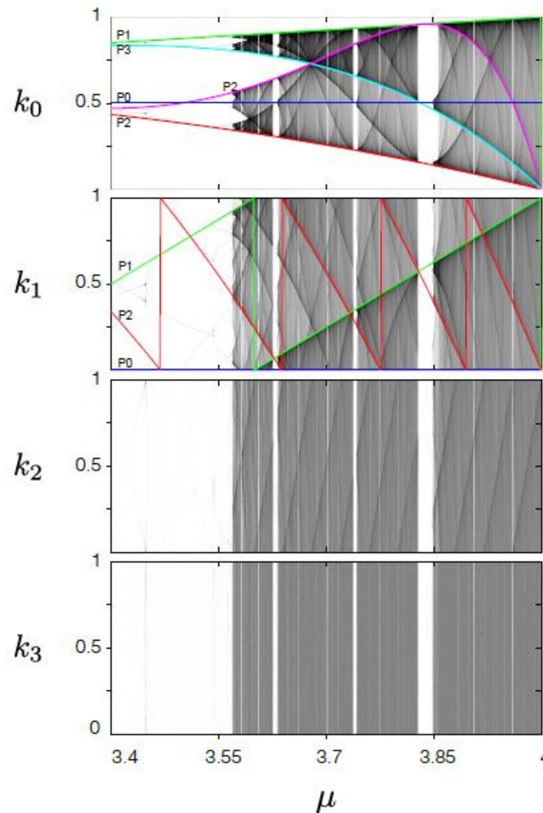


Fonte: Adaptada de [12].

Na figura 16 é apresentado diagramas de bifurcação do k-mapa logístico, para quatro valores de k incluindo o mapa logístico original. Comparando o diagrama de bifurcação original com os demais, verifica-se que as interrupções súbitas do caos permanecem e nos mesmos intervalos. Assim, como já concluído anteriormente, o k-mapa logístico preserva a

periodicidade. Nota-se também que os padrões ficam mais preenchidos a medida que k aumenta, isso devido ao aumento dos períodos, assim como visto nas figuras 14 e 15.

Figura 16 – Diagrama de bifurcação do k-mapa logístico correspondente às órbitas k_0, k_1, k_2 e k_3.

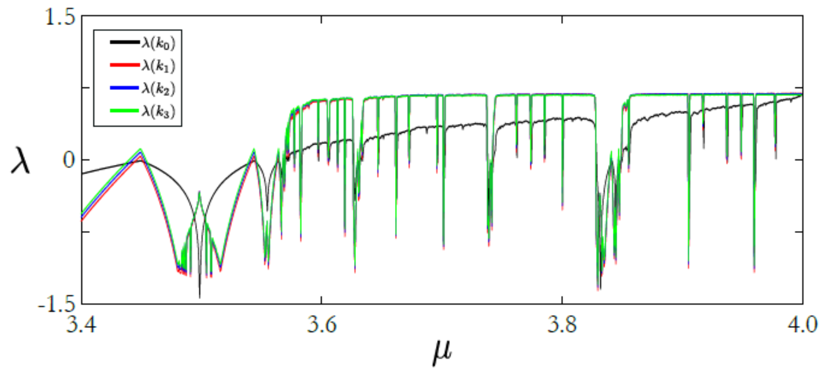


Fonte: Adaptada de [12].

A partir da análise do diagrama espaço-tempo, do diagrama *cobweb* e do diagrama de bifurcação, Machicao [12] concluiu que a dinâmica global do mapa logístico original é preservada no k-mapa logístico, já que as regiões estáveis, periódicas e caóticas são preservadas, e que o expoente de Lyapunov aumenta no k-mapa logístico, como observa-se na figura 17. Entretanto as regiões onde ele originalmente é negativo, ele permanece negativo, já que no k-mapa logístico as regiões estáveis e periódicas são preservadas, como concluído previamente.

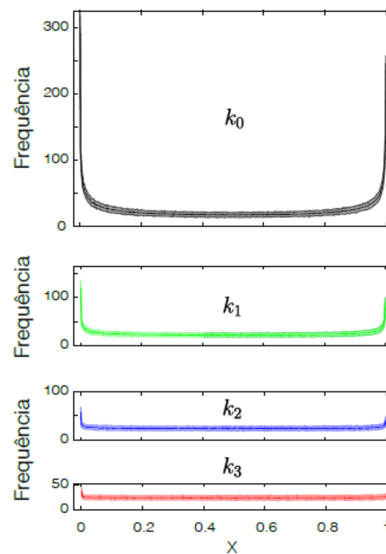
Além de melhorar o expoente de Lyapunov, o k-mapa logístico melhora a distribuição de frequência, como verifica-se na figura 18. Observa-se que a medida que k aumenta, os valores x ficam mais bem distribuídos, se aproximando cada vez mais a uma distribuição mais uniforme.

Figura 17 – Expoente de Lyapunov do k-mapa logístico correspondente às órbitas k_0, k_1, k_2 e k_3.



Fonte: [12].

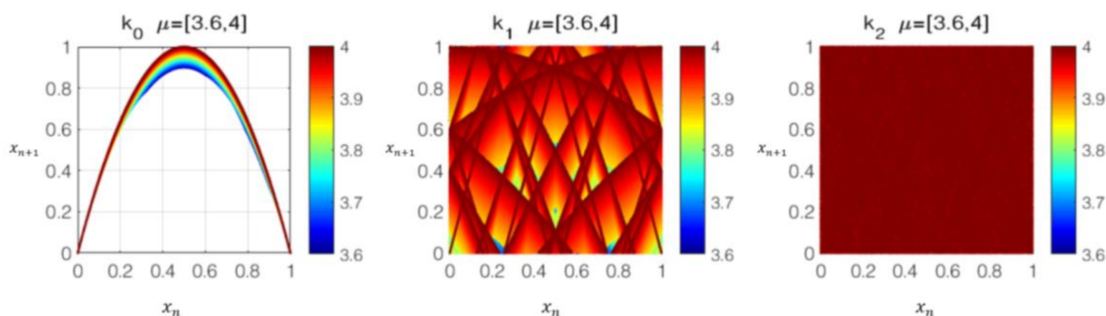
Figura 18 – Curvas de distribuição de frequências do k-mapa logístico correspondente às órbitas k_0, k_1, k_2 e k_3, para $\mu = 4$.



Fonte: Adaptada de [12].

Na figura 19 é apresentado a transformação do diagrama de Poincaré do mapa logístico original, parábola invertida, para o diagrama do k-mapa logístico. Quando $k = 1$, o diagrama apresenta padrões zigue-zague, e quando $k = 2$, os padrões desaparecem, se tornando visualmente aleatórios, logo, fica mais difícil encontrar os parâmetros do mapa, já que se torna cada vez mais não-invertível.

Figura 19 – Diagrama de Poincaré do k-mapa logístico para $3{,}6 \leq \mu \leq 4$ e $0 \leq k \leq 2$.



Fonte: Adaptada de [12].

Assim conclui-se que a utilização do k-mapa logístico melhora as propriedades estatísticas das sequências geradas de forma que suas propriedades se aproximem ao máximo das esperadas em uma sequência verdadeiramente aleatória, isto é, uma distribuição uniforme e imprevisibilidade.

2.3 Conjunto de testes do NIST

A qualidade da aleatoriedade dos dados gerados por um RNG é analisada por testes estatísticos [16]. Um conjunto de testes foi formulado pelo NIST, especificamente para aplicações criptográficas, tanto para TRNGs quanto para PRNGs [8].

Conforme Rukhin et al. [8], o *NIST Test Suit* possui 15 testes estatísticos individuais focados em diferentes tipos de não aleatoriedade que podem existir em uma sequência. São eles:

a) *Frequency* (*Monobit*) *Test;*
b) *Frequency Test within a Block;*
c) *Runs Test;*
d) *Test for the Longest Run of Ones in a Block;*
e) *Binary Matrix Rank Test;*
f) *Discrete Fourier Transform (Spectral) Test;*
g) *Non-overlapping Template Matching Test;*
h) *Overlapping Template Matching Test;*
i) *Maurer's "Universal Statistical" Test;*

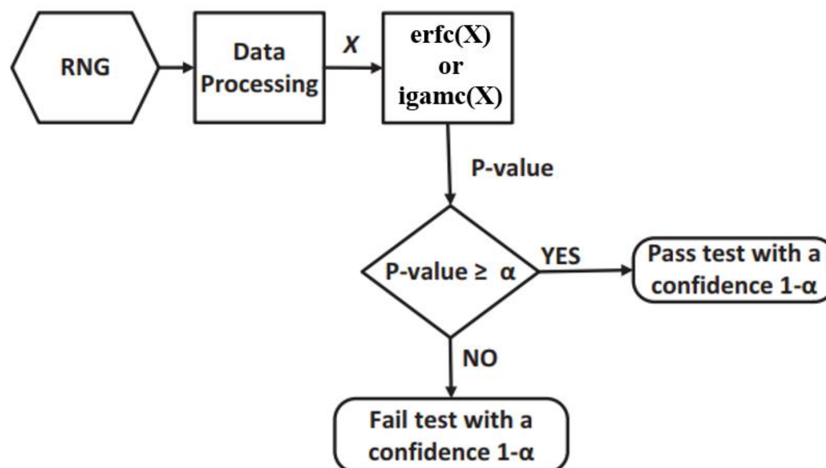
j) *Linear Complexity Test;*
k) *Serial Test;*
l) *Approximate Entropy Test;*
m) *Cumulative Sums (Cusum) Test;*
n) *Random Excursions Test;*
o) *Random Excursions Variant Test.*

Cada teste, a partir do processamento de uma sequência de bits, gera uma variável X que, através da utilização da função *erfc* (*Complimentary Error Function*), equação (4), ou da função *igamc* (*Upper Incomplete Gamma Function*), equação (5), gera um *P-value* que é comparado com um valor pré-definido α, também chamado de valor crítico, que é fixado na faixa de $0.001 \leq \alpha \leq 0.01$. Se o *P-value* for maior ou igual a α a sequência é aleatória com uma confiança de 1-α. A figura 20, mostra um fluxograma genérico dos testes do NIST.

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_{z}^{\infty} e^{-u^2} du \tag{4}$$

$$\text{igamc}(a, x) = \frac{1}{\Gamma(a)} \int_{x}^{\infty} e^{-t} t^{a-1} dt \qquad , onde\ \Gamma(a) = \int_{0}^{\infty} t^{a-1} e^{-t} dt \tag{5}$$

Figura 20 – Fluxograma genérico dos testes do NIST.



Fonte: Adaptada de [26].

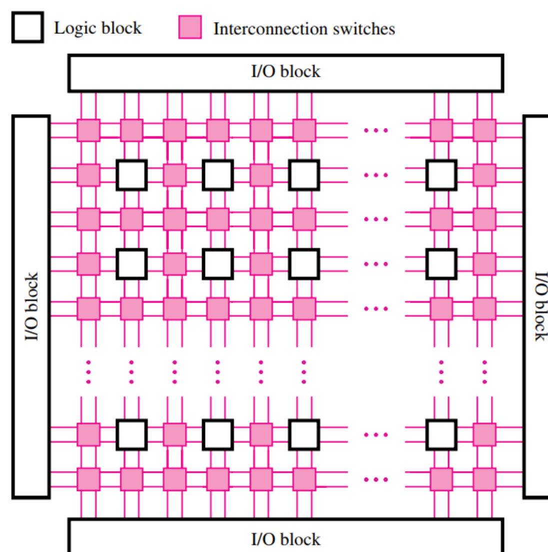
3 MATERIAIS E MÉTODOS

Este capítulo traz uma breve descrição da arquitetura escolhida para esse projeto, a arquitetura FPGA, os materiais e de que modo eles foram utilizados para o desenvolvimento da pesquisa.

3.1 Arquitetura FPGA

Segundo [27], uma FPGA é um dispositivo lógico programável (PLD) composto por três partes principais: blocos lógicos, blocos de I/O (entrada e saída), e interconexões programáveis, como apresentado na figura 21. Os blocos lógicos são utilizados para implementar pequenas funções lógicas, mas a utilização de vários possibilita implementações complexas. É através das interconexões programáveis que um bloco lógico se conecta ao outro de diversas maneiras, assim como se conectam aos blocos de I/O, que possibilitam a entrada e saída de sinais da FPGA.

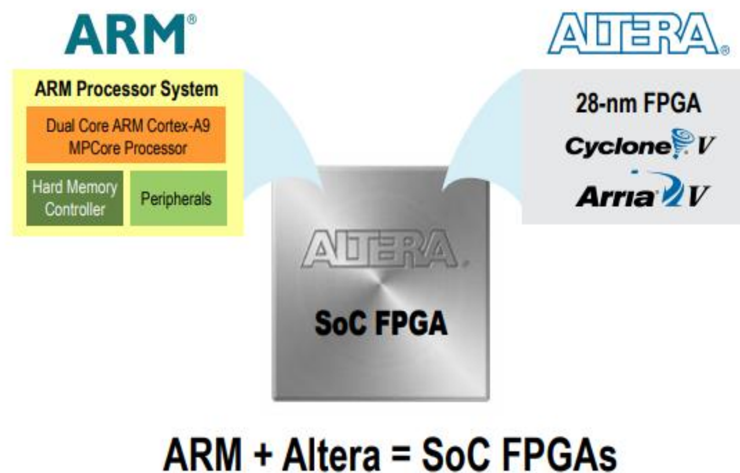
Figura 21 – Estrutura geral de uma FPGA.



Fonte: [27].

A maioria dos sistemas embarcados atuais têm como núcleo processadores e FPGAs. Com a junção dessas duas arquiteturas é possível criar plataformas computacionais embarcadas extremamente poderosas [28]. Assim, há uma linha de FPGAs, chamada de SoC (System on Chip) FPGAs que integra processadores e FPGAs em um único dispositivo, como mostrado na figura 22.

Figura 22 – Modelo da SoC FPGA da Altera.



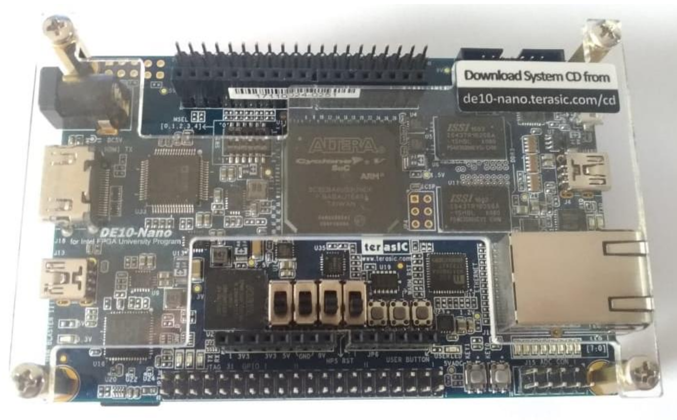
Fonte: [28].

Para esta pesquisa optou-se pela utilização da placa de desenvolvimento Terasic DE10-Nano, a qual possui a Cyclone V SoC FPGA, figura 23.

A Cyclone V SoC FPGA é dividida em duas partes, a primeira contém a FPGA Cyclone V, e a segunda um processador ARM Cortex-A9, tal parte é chamada de Hard Processor System (HPS). Na placa, a parte da FPGA, além de conter a FPGA em si, inclui também GPIOs, botões, LEDs, entre outros componentes. O HPS, além do processador, contém acelerômetro, conexão Ethernet RJ45, soquete para cartão micro SD (µSD), entre outros.

Para a implementação de um determinado sistema, na FPGA, por meio de ferramentas computacionais, como o Quartus Prime, é necessário descrevê-lo em uma das linguagens de descrição de *hardware* (HDLs). Segundo [29], as duas HDLs mais utilizadas são Verilog e VHDL (VHSIC *Hardware Description Language*). Ainda no Quartus Prime é possível configurar a comunicação entre a FPGA e o HPS.

Figura 23 – Placa de desenvolvimento Terasic DE10-Nano.



Fonte: Elaborada pelo autor.

Nesse trabalho, para se utilizar o HPS, optou-se pelo sistema operacional Linux com uma distribuição específica para dispositivos embarcados. Mais especificamente, Angstrom GNU/Linux v2014.12 (*core edition*), cuja versão do kernel é Linux 4.5.0-00185-g3bb556b. É através de um programa executado no Linux que se é possível acessar e enviar sinais ao circuito descrito no lado FPGA da Cyclone V SoC.

Outros *softwares* foram usados como: ModelSim, para simulações de projetos feitos no Quartus Prime, Putty, para comunicação serial entre a placa DE10-Nano e o computador, e FileZilla, para transferências de arquivos entre a placa de desenvolvimento e o computador.

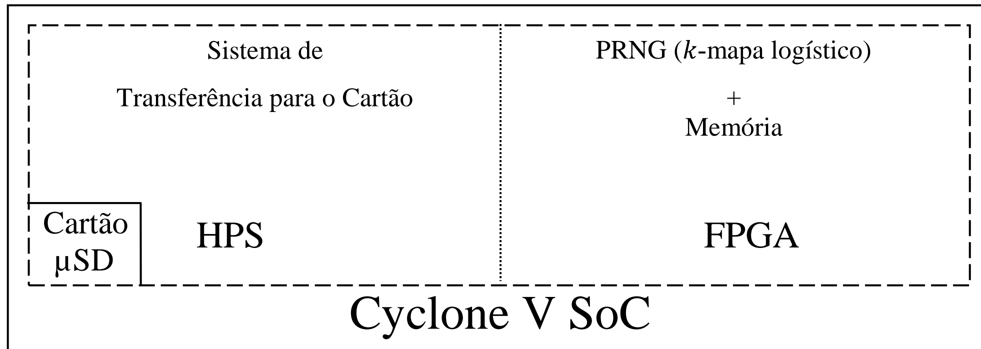
3.2 Estrutura básica do sistema

O sistema desenvolvido é composto de um PRNG baseado no k-mapa logístico, proposto em [12], juntamente a uma memória e um sistema de transferência dos valores gerados para o cartão µSD, como apresentado na figura 24.

O funcionamento do sistema é composto por 2 etapas que são melhor detalhadas no capítulo 4. A primeira etapa é a geração de uma sequência de valores pelo PRNG e o armazenamento da mesma em uma memória, na parte FPGA da Cyclone V SoC. A segunda etapa é a transferência dessa sequência para o HPS e o armazenamento dela no cartão µSD, possibilitando que ela seja transferida ao computador e analisada através do conjunto de testes

do NIST. Através desse funcionamento é possível validar a implementação em *hardware* do k-mapa logístico.

Figura 24 – Estrutura básica do sistema.



Fonte: Elaborada pelo autor.

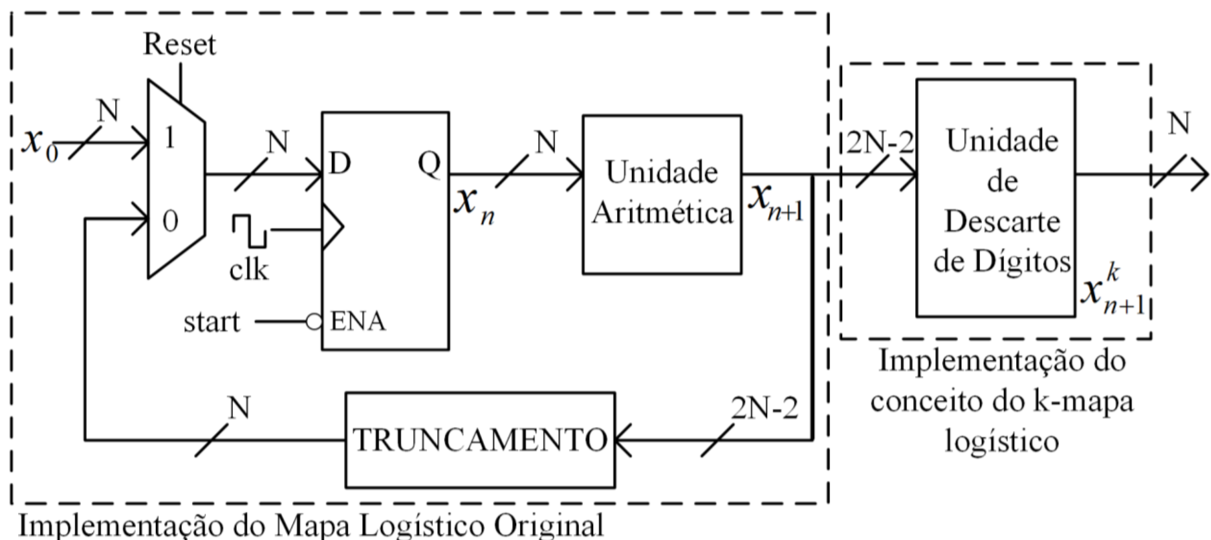
4 IMPLEMENTAÇÃO EM *HARDWARE*

O presente capítulo traz um detalhamento maior sobre a elaboração e o funcionamento do sistema.

4.1 Implementação do PRNG proposto

O PRNG proposto baseado no conceito do k-mapa logístico apresenta dois estágios principais, como apresentado na figura 25. O primeiro estágio é a implementação do mapa logístico original, apresentado na equação (2), enquanto, o segundo, chamado de Unidade de Descarte de Dígitos (UDD) é a implementação do conceito do k-mapa logístico. A letra N representa o tamanho do barramento, que para essa pesquisa escolheu-se N = 32 bits, utilizando a representação em ponto-fixo não sinalizada, com os 32 bits representando a parte fracionária.

Figura 25 – PRNG proposto.

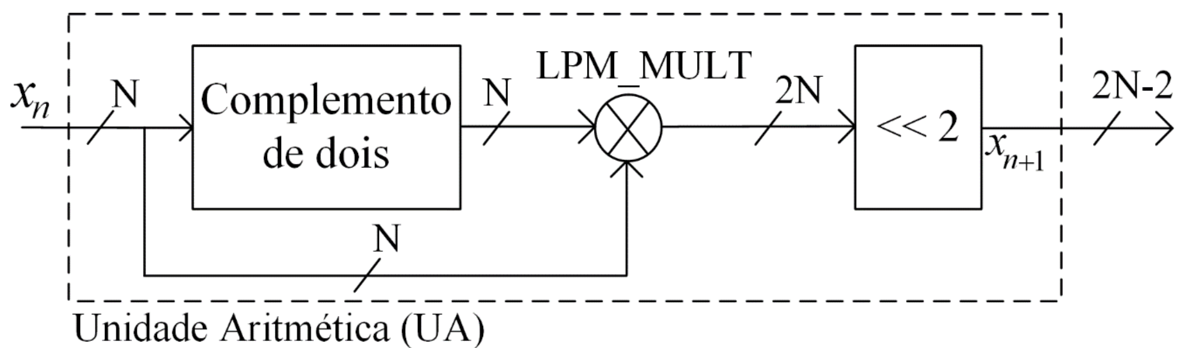


Fonte: Elaborada pelo autor.

No circuito do mapa logístico original, verifica-se a presença de um multiplexador que é responsável por alimentar o mapa com um valor inicial. Quando o sinal *reset* tiver nível alto a entrada com o sinal x_0 é selecionada, possibilitando assim a reinicialização do mapa. Se o

sinal $reset$ tiver nível baixo a outra entrada do multiplexador é escolhida, logo, o mapa é realimentado com o seu último valor gerado. A presença do registrador se faz necessária para que a cada pulso do relógio clk seja gerado um novo valor. É importante observar também, no registrador, a presença da entrada ENA, cuja função é habilitar a geração de valores quando o sinal $start$ tiver nível baixo. Outro importante componente é Unidade Aritmética (UA), figura 26, responsável por realizar as operações presentes na equação (2), com $\mu = 4$.

Figura 26 – Unidade Aritmética.



Fonte: Adaptada de [30].

Sabe-se que o complemento de dois de um número binário x_n de N bits, $x_n^{"}$, é:

$$x_n^{"} = 2^N - x_n \quad (6)$$

Sendo que, 2^N é o maior valor representável de x_n mais *ulp* (*unit in the least position*) [31]. Como, nesse trabalho utiliza-se a representação em ponto-fixo não sinalizada com todos os bits representando somente a parte fracionária, isto é, $x_n = (0, b_1 b_2 b_3 \ldots\ b_n)_2$:

$$2^N = (0{,}1111 \ldots 1111)_2 + ulp = 1_2 \quad (7)$$

Logo, levando a equação (7) em (6):

$$x_n^{"} = 1_2 - x_n \quad (8)$$

Portanto, buscando usar o mínimo de recursos de *hardware* possível, a UA efetua a operação denotada na equação (9). A subtração $1 - x_n$ é calculada por x_n'', a multiplicação $x_n(x_n'')$ é feita utilizando o módulo da Altera LPM_MULT, e a multiplicação entre o valor resultante de LPM_MULT e 4 é feita aplicando um deslocamento a esquerda de 2 bits em $x_n(x_n'')$.

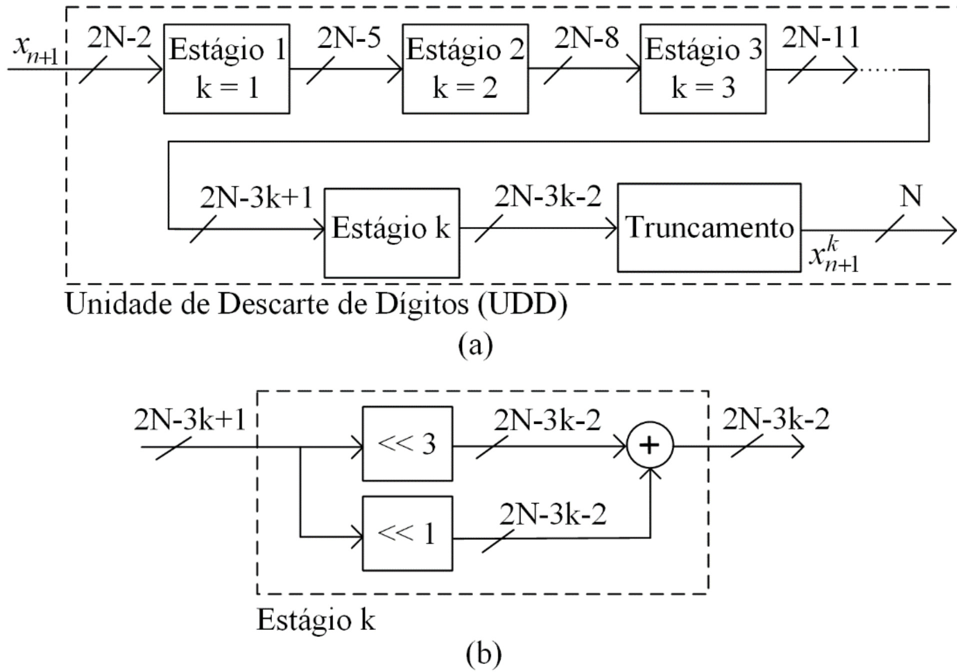
$$4x_n(1 - x_n) = [x_n(x_n'')] \ll 2 \tag{9}$$

Importante observar que o tamanho do barramento muda após a multiplicação em LPM_MULT, já que quando multiplicado dois valores com N bits, tem-se como resultado um valor com 2N bits. De forma semelhante, o barramento sofre mais uma mudança após o deslocamento de 2 bits, já que essa operação adiciona dois zeros aos bits menos significativos, podendo eles serem descartados, resultando, na saída da UA, um barramento de 2N-2 bits.

Após gerado, o valor resultante da UA, x_{n+1}, com 2N-2 bits, segue dois caminhos. Em um ele segue para a UDD e no outro ele realimenta o mapa para gerar um novo valor. Porém, o tamanho do barramento para realimentar o mapa é de N bits, assim, foi adicionado uma etapa intermediária onde ocorre um truncamento reduzindo x_{n+1} de 2N-2 bits para N bits.

A fim de aplicar o conceito do k-mapa logístico, apresentado na equação (3), a UDD, apresentada na figura 27, possui k estágios. Cada estágio, figura 27b, é responsável por descartar um dígito decimal mais significativo de x_{n+1}. Isto é, se $k = 2$, logo, a UDD tem de descartar dois dígitos decimais do valor gerado pelo mapa logístico, assim, ela terá dois estágios.

Figura 27 – Unidade de Descarte de Dígitos. (a) UDD com k estágios. (b) Dentro de um estágio k.



Fonte: Adaptada de [30].

Da mesma forma que na equação (3), cada estágio multiplica o valor em sua entrada por dez e a parte inteira da representação em ponto-fixo é descartada. Buscando usar o mínimo de recursos de *hardware* possível, especialmente multiplicadores embarcados, a multiplicação por dez foi implementada por duas operações de deslocamento e uma soma, como apresentado na equação (10). Essa implementação pode ser feita uma vez que a equação (11) é verdadeira.

$$10y = y \ll 3 + y \ll 1 \tag{10}$$

$$10y = 2^3 y + 2^1 y \tag{11}$$

Na saída de cada estágio o tamanho do barramento é reduzido em três bits, já que a operação de deslocamento adiciona zeros nos bits menos significativos que podem ser descartados. Assim, pode-se calcular o valor máximo de k conforme a equação (12), onde $\lfloor\ \rfloor$ é a função $floor$.

$$k_{max} = \lfloor (N-2)/3 \rfloor \tag{12}$$

Após efetuados todos os k estágios, o tamanho do barramento tem de ser ajustado para N bits, portanto um estágio de truncamento é utilizado, como observa-se na figura 27a. Como para essa pesquisa escolheu-se $N = 32$ bits, logo $k_{max} = 10$.

4.2 Sistema de geração e armazenamento de valores aleatórios

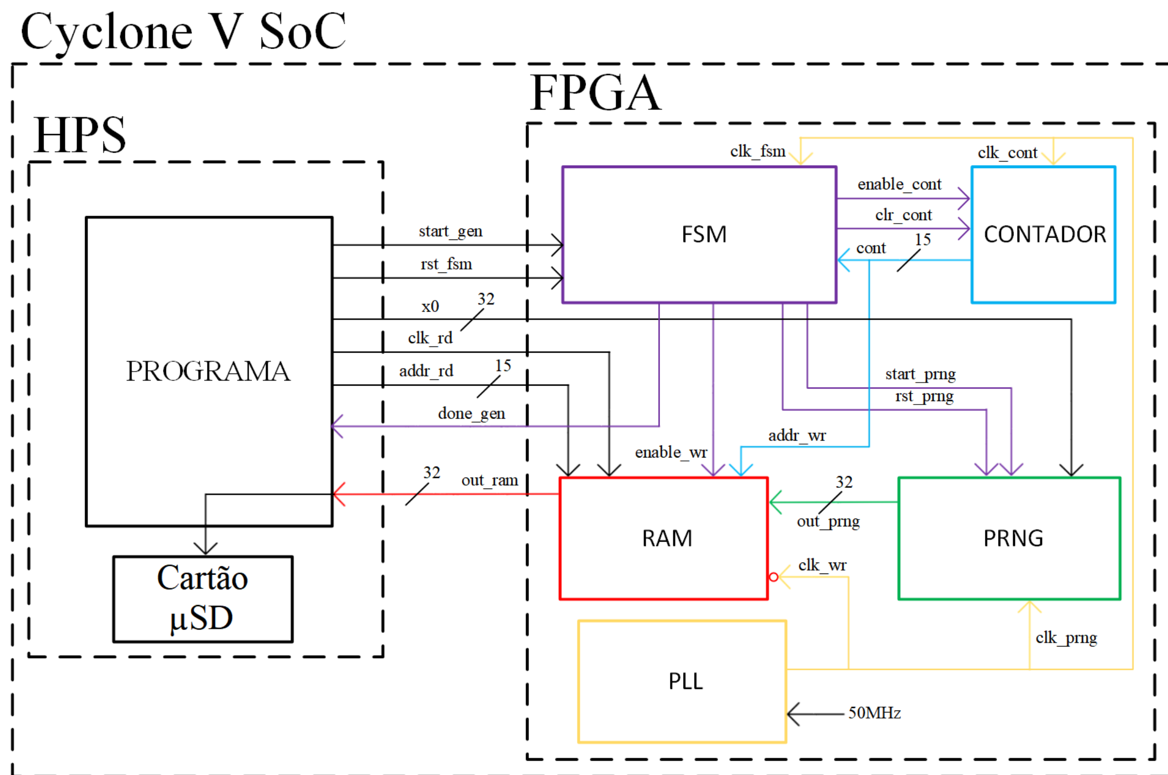
Como dito previamente, a Cyclone V SoC é composta por duas partes, FPGA e HPS. Nessa pesquisa elaborou-se, na FPGA, um sistema responsável por gerenciar a reinicialização do PRNG proposto e armazenar os valores gerados dentro de uma memória. Já no HPS, um sistema que lê os valores armazenados na memória, na FPGA, e os transfere para um arquivo no cartão µSD.

A figura 28 apresenta os principais componentes e sinais do sistema. Dessa figura, observa-se na FPGA os seguintes componentes: PLL (*Phase Locked Loop*), FSM (*Finite-State Machine*), CONTADOR, PRNG e RAM (*Random Access Memory*). O PLL é responsável por gerar um relógio que alimenta os componentes FSM, CONTADOR, RAM e PRNG, a partir do relógio de 50 MHz fornecido na placa DE10-Nano. Nesse projeto, optou-se por gerar um relógio de 5 MHz. Importante observar que os sinais *clk_wr*, *clk_prng*, *clk_cont* e *clk_fsm*, estão ligados ao mesmo sinal de saída do componente PLL.

O componente PRNG foi implementado conforme indicado na figura 25. Como foi escolhido N=32, verifica-se que os barramentos de entrada do valor inicial x0 (originado do HPS) e de saída *out_prng* possuem 32 bits. Como os bits gerados serão analisados pelo conjunto de testes do NIST, foi escolhido gerar e armazenar uma sequência com 32768 valores, resultando em 2^{20} bits, já que o mínimo recomendado pelo NIST SP 800-22 [8] é de 10^6 bits.

Para armazenar esses valores é necessária uma memória que consiga alocar todos eles, e essa é a função do componente RAM. O componente é composto pelo módulo da Altera RAM: 2-PORT, uma RAM com relógios e endereços separados para escrita e leitura. Como serão armazenados 32768 valores, de 32 bits cada, os barramentos de endereço possuem 15 bits cada. O uso dessa componente de memória se dá uma vez que ao conectar a saída do PRNG direto no HPS, ele não consegue acompanhar uma alta velocidade de geração do PRNG, ficando assim, a frequência de geração limitada pelo desempenho do HPS.

Figura 28 – Sistema de geração e armazenamento de valores aleatórios.



Fonte: Elaborada pelo autor.

O componente CONTADOR é composto pelo módulo da Altera LPM_COUNTER, como ele precisa contar 32768 valores, ele é um contador de 15 bits. Além de contar quantos valores foram gerados ele também funciona como fonte do endereço de escrita do componente RAM. Isto é, à medida que ocorre uma borda de subida do relógio *clk_cont*, um valor é gerado pelo PRNG, o contador é incrementado e na borda de descida o valor gerado é armazenado na RAM no endereço fornecido pelo CONTADOR, *addr_wr*.

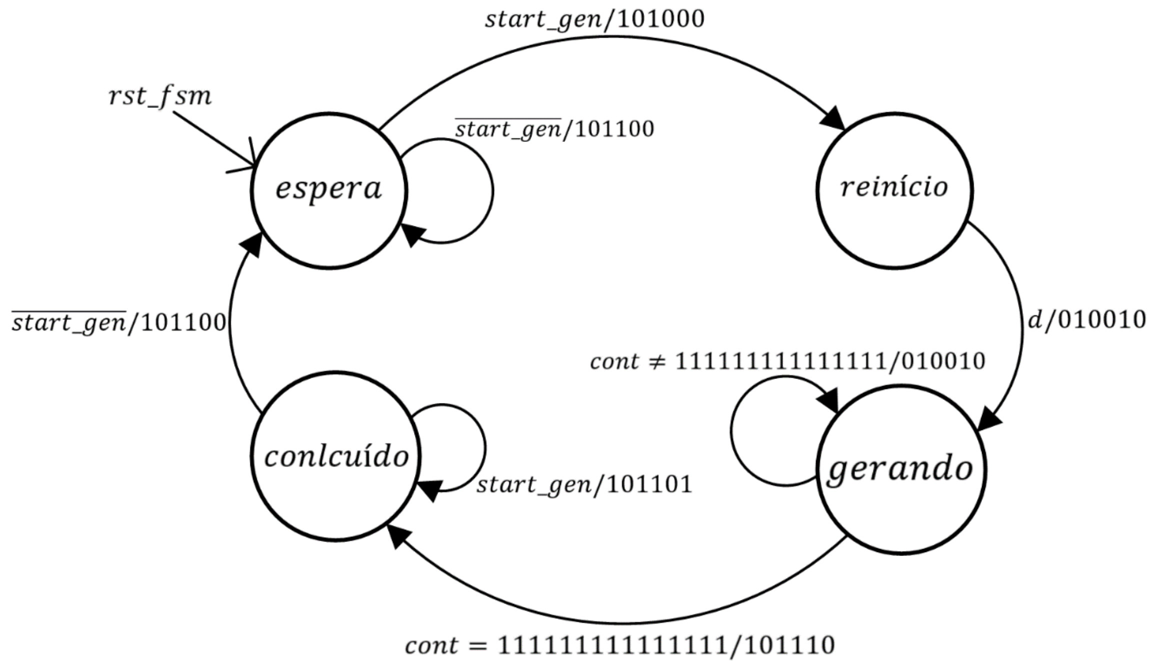
Foi implementado uma máquina de estados finitos, no componente (FSM), figura 29, que juntamente ao HPS gerencia o processo de reinicialização do PRNG, armazenamento da sequência gerada na RAM, leitura da RAM e armazenamento no cartão µSD. Observa-se que na figura 29 a entrada ‘d’ significa que para mudar do estado “reinício” para “gerando” os valores das entradas não importam.

A tabela 1 apresenta como se encontram os componentes CONTADOR, PRNG e RAM em cada um dos estados da FSM mostrada na figura 29.

Figura 29 – Máquina de estados finitos do componente FSM.

Entradas: *start_gen*; *cont*

Saídas: *rst_prng*; *enable_cont*; *clr_cont*; *start_prng*; *enable_wr*; *done_gen*;



Fonte: Elaborada pelo autor.

Tabela 1 – Relação estado x componentes.

Estado	***Status* do CONTADOR**	***Status do PRNG***	***Status* da RAM**
espera	desativado e zerado	desativado	escrita desativada
reinício	desativado e zerado	reiniciando com valor inicial x0	escrita desativada
gerando	ativado	gerando	escrita habilitada
concluído	desativado e zerado	desativado	escrita desativada

Fonte: Elaborada pelo autor.

A máquina de estados ao ser reinicializada, através do sinal *rst_fsm*, gerado pelo HPS, encontra-se no estado “espera” nesse estado o PRNG, o CONTADOR e a RAM estão desativados, ou seja, nada ocorre.

Quando o HPS coloca o sinal *start_gen* em nível alto a FSM vai para o estado “reinício”, nesse estado o PRNG é reiniciado com o valor inicial fornecido pelo HPS, x0, e no próximo pulso de relógio a FSM vai para o estado “gerando”, independentemente dos valores das entradas, uma vez que é necessário somente um pulso de relógio para o mapa logístico reiniciar. Nesse estado, na mesma frequência em que se geram os valores, eles também são armazenados na RAM.

A FSM fica no estado “gerando” até que os 32768 valores tenham sido gerados e guardados na RAM, isso é possível pois uma das entradas da FSM é a saída do CONTADOR, isto é, quando o sinal *cont* for “111111111111111” significa que a sequência pretendida foi gerada e armazenada e a FSM pode ir para o próximo estado, “concluído”.

No estado “concluído” os componentes CONTADOR, PRNG e RAM apresentam comportamento semelhante ao de quando estão no estado “espera”, entretanto, no estado “concluído”, o sinal *done_gen* fica em nível alto, indicando ao HPS que ele pode ler os 32768 valores armazenados na RAM. Sendo assim, o HPS gera um relógio para a leitura da RAM, *clk_rd* = 260kHz. A cada pulso de relógio realiza-se uma leitura, armazena-se o valor lido em um arquivo no cartão µSD, e altera-se o endereço de leitura enviado para a RAM, *addr_rd*, até que toda a memória tenha sido mapeada. Terminado esse processo o HPS coloca o sinal *start_gen* em nível baixo, possibilitando com que a FSM mude do estado “concluído” para “espera”, e nele permanece até que o sinal *start_gen* volte a nível alto.

5 RESULTADOS E DISCUSSÕES

Como esse trabalho adotou N=32 bits, tem-se onze possíveis implementações do mapa logístico, incluindo o mapa logístico tradicional, visto que $k_{max} = 10$. Utilizando o *software* Quartus Prime essas onze variações foram implementadas.

A tabela 2 apresenta o uso de *hardware* obtido a partir da síntese do sistema da figura 28, para cada variação. Enquanto, a tabela 3 apresenta o uso de *hardware* apenas do PRNG da figura 25 para cada valor de k.

Tabela 2 – Resultado da síntese do sistema de geração e armazenamento.

k	***Adaptative Logic Module* (ALM)**	**Registradores**	**Multiplicadores Embarcados 18x18-bits**	**Blocos de Bits de Memória**
0	3182/41910(8%)	4619	4/224(1,8%)	1051392/5662720(19%)
1	3205/41910(8%)	4628	4/224 (1,8%)	1051392/5662720(19%)
2	3243/41910(8%)	4593	4/224 (1,8%)	1051392/5662720(19%)
3	3260/41910(8%)	4621	4/224 (1,8%)	1051392/5662720(19%)
4	3286/41910(8%)	4595	4/224 (1,8%)	1051392/5662720(19%)
5	3303/41910(8%)	4582	4/224 (1,8%)	1051392/5662720(19%)
6	3331/41910(8%)	4614	4/224 (1,8%)	1051392/5662720(19%)
7	3346/41910(8%)	4621	4/224 (1,8%)	1051392/5662720(19%)
8	3357/41910(8%)	4614	4/224 (1,8%)	1051392/5662720(19%)
9	3373/41910(8%)	4599	4/224 (1,8%)	1051392/5662720(19%)
10	3388/41910(8%)	4599	4/224 (1,8%)	1051392/5662720(19%)

Fonte: Elaborada pelo autor.

Tabela 3 – Resultado da síntese do circuito do PRNG proposto.

k	*Adaptative Logic Module* **(ALM)**	**Registradores**	**Multiplicadores Embarcados 18x18-bits**
0	63/41910(<1%)	32	4/224(1,8%)
1	86/41910(<1%)	32	4/224 (1,8%)
2	115/41910(<1%)	32	4/224 (1,8%)
3	141/41910(<1%)	32	4/224 (1,8%)
4	165/41910(<1%)	32	4/224 (1,8%)
5	186/41910(<1%)	32	4/224 (1,8%)
6	205/41910(<1%)	32	4/224 (1,8%)
7	223/41910(<1%)	32	4/224 (1,8%)
8	239/41910(<1%)	32	4/224 (1,8%)
9	252/41910(<1%)	32	4/224 (1,8%)
10	264/41910(<1%)	32	4/224 (1,8%)

Fonte: Elaborada pelo autor.

A implementação do mapa logístico, com barramento de 32 bits, presente em [13] utiliza 17 *slices*, 66 *look-up tables* (LUT), 32 registradores e 4 multiplicadores 18x25-bits, isto quando sintetizado no dispositivo Virtex 5 FX 30T FPGA (Xilinx). Enquanto que em [14], a síntese no dispositivo XC5VLX50T, de um mapa logístico com um barramento de 27 bits, utiliza 253 LUT, 27 registradores e 4 multiplicadores 18x25-bits. A tabela 4 compara o uso de *hardware* em [13] e [14] com o uso médio desse trabalho (considerando todas as variações do mapa logístico). Assim, verifica-se que o PRNG projetado possui uso de recursos similar, para tamanhos de barramento próximos.

Utilizando o *software* Quartus Prime foi realizada uma estimativa da frequência máxima de operação do PRNG proposto na figura 25. Considerando todas as variações do mapa logístico possíveis a partir do conceito do k-mapa logístico, para um barramento de 32 bits,

tem-se uma frequência máxima média de $\bar{f}_{max} = 155{,}90\ MHz$ e desvio padrão de $\sigma_{fmax} = 6{,}55\ MHz$.

Tabela 4 – Comparação do uso de *hardware* em trabalhos diferentes.

Referência	Uso de *hardware*	Registradores	Multiplicadores Embarcados
Dabal e Pelka [13]	66 (LUT) [1]	32	4 (18x25-bits)
Sayed et al. [14]	253 (LUT)[1]	27	4 (18x25-bits)
Trabalho atual	177 (ALM) [2]	32	4 (18x18-bits)

Fonte: Elaborada pelo autor.

A tabela 5 compara as frequências máximas e os *throughputs* apresentados em [13] e [14] com os estimados nesse trabalho. Observa-se que o PRNG proposto, nesse trabalho, possui frequência e *throughput* maior do que nos demais.

Tabela 5 – Comparação de frequências máximas e *throughputs* de trabalhos diferentes.

Referência	$f_{max}\ [MHz]$	*throughput* [Mbit/s]
Dabal e Pelka [13]	76,10	2435,2
Sayed et al. [14]	61,812	1668,9
Trabalho atual	155,90	4988,8

Fonte: Elaborada pelo autor.

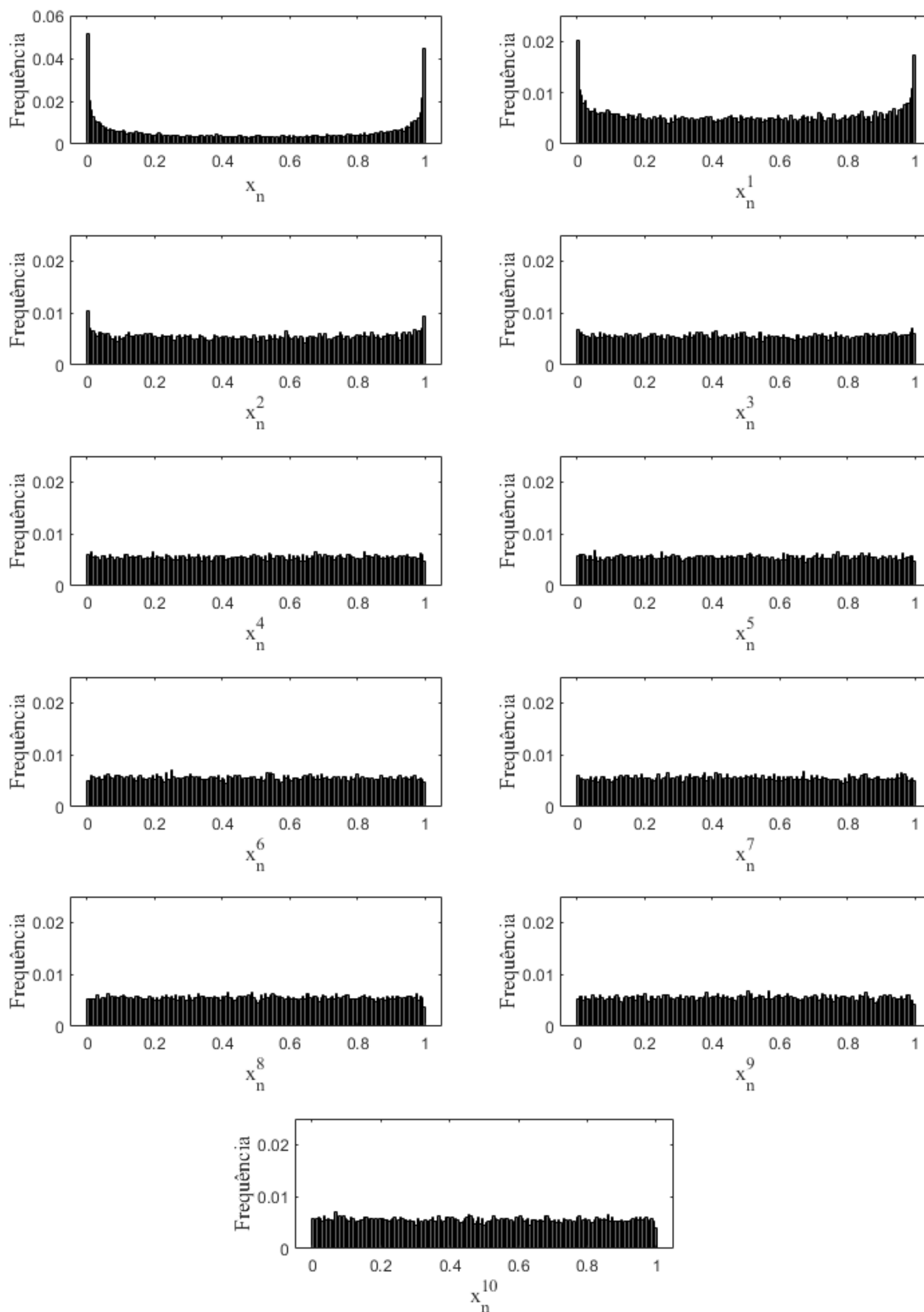
A partir do sistema apresentado na figura 28, foram geradas sequência de 32768 valores, com 32 bits cada, totalizando 2^{20} bits. Cada sequência foi gerada a partir do mesmo valor inicial $x_0 = (3CD11396)_{16}$, porém para diferentes valores de k, com $0 \leq k \leq 10$.

[1] LUT de 6 entradas.

[2] Cada ALM contém uma LUT de 8 entradas.

A figura 30 apresenta as curvas de distribuição de frequência das sequências geradas. Ao comparar as curvas da figura 30 com as da figura 18, prova-se que o PRNG implementado tem comportamento similar ao k-mapa logístico proposto em [12]. Ou seja, a medida que k aumenta a curva de distribuição fica mais uniforme, uma das propriedades desejadas para uma sequência aleatória.

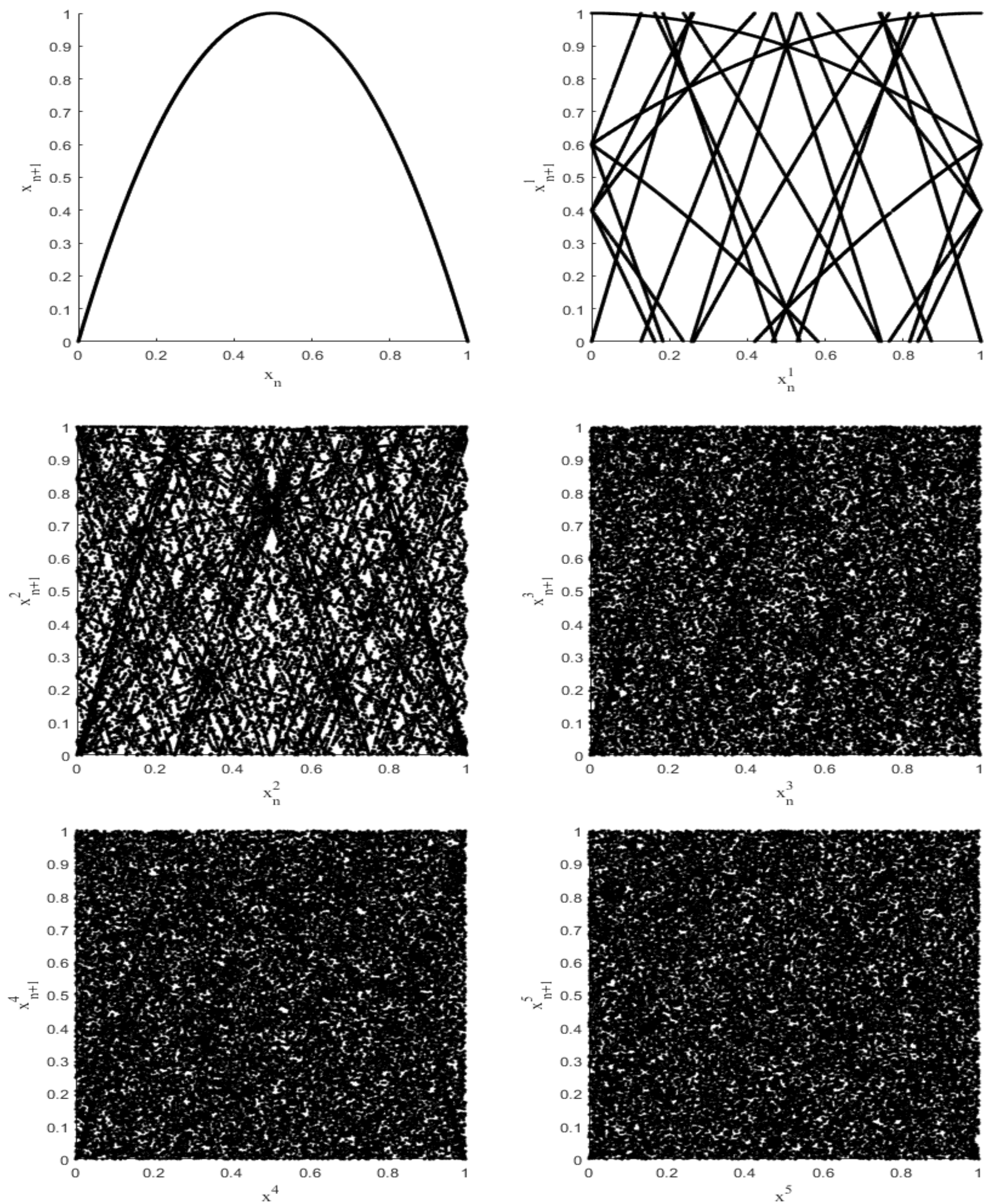
Figura 30 - Curvas de distribuição de frequência de sequências geradas pelo PRNG proposto.



Fonte: Elaborada pelo autor.

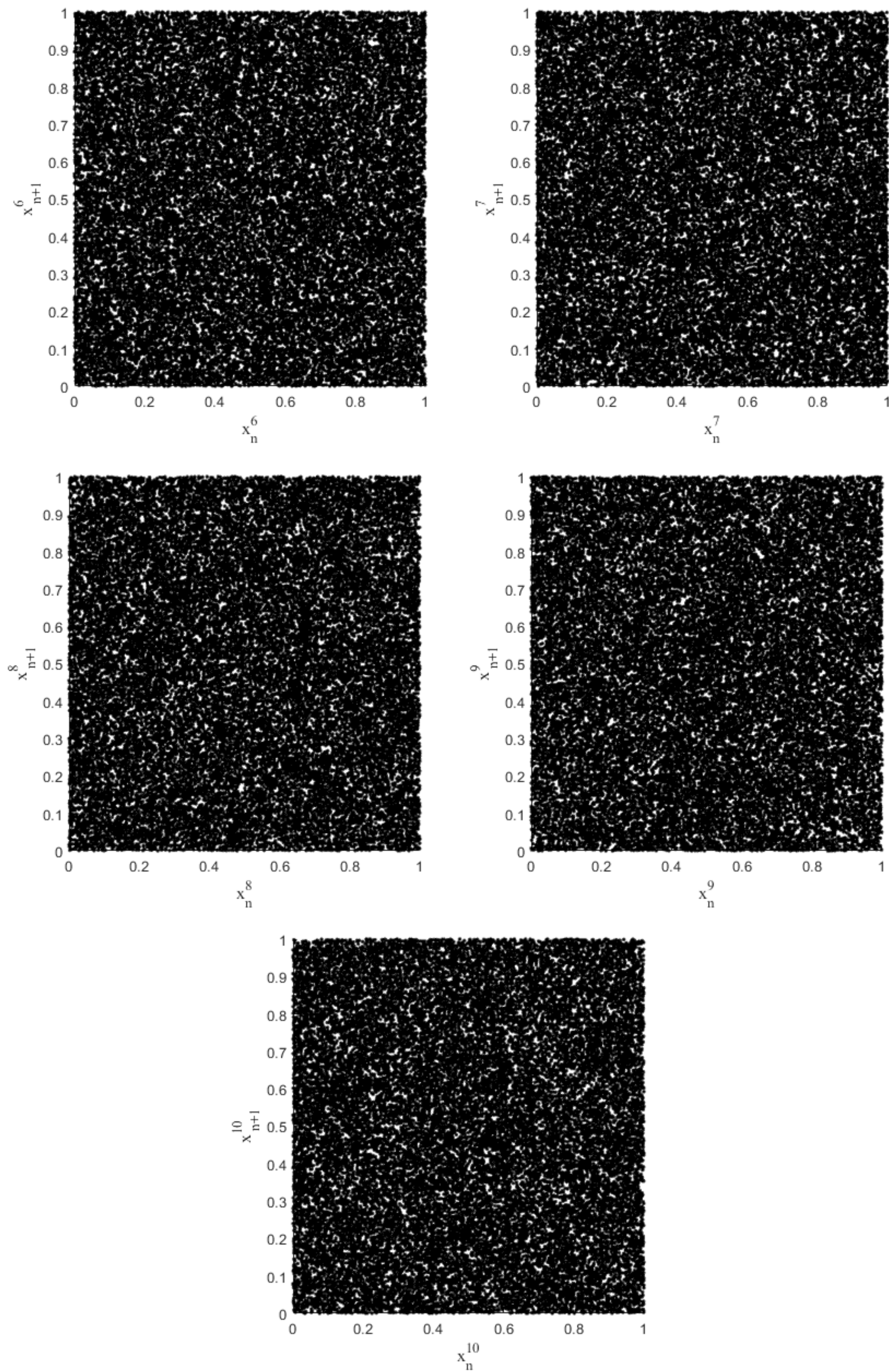
A partir das sequências geradas, também foi construído os diagramas de Poincaré apresentados nas figuras 31 e 32. Ao comparar esses diagramas com os mostrados na figura 19 verifica-se também a semelhança entre eles, como esperado.

Figura 31 – Diagrama de Poincaré do PRNG proposto para $0 \leq k \leq 5$.



Fonte: Elaborada pelo autor.

Figura 32 - Diagrama de Poincaré do PRNG proposto para $6 \leq k \leq 10$.



Fonte: Elaborada pelo autor.

Analisando as sequências geradas através do conjunto de testes do NIST apresentado em [8] tem-se os resultados mostrados na tabela 6. Os testes foram aplicados com um valor crítico de $\alpha = 0{,}01$.

Tabela 6 – Resultados do conjunto de testes do NIST.

k	Aprovação
0	9/15 (60%)
1	9/15 (60%)
2	10/15 (67%)
3	15/15 (100%)
4	15/15 (100%)
5	13/15 (87%)
6	15/15 (100%)
7	15/15 (100%)
8	15/15 (100%)
9	15/15 (100%)
10	15/15 (100%)

Fonte: Elaborada pelo autor.

Os resultados na tabela 6 mostram que quando aplicadas as sequências geradas pelo mapa logístico original e pelo k-mapa logístico, para $k = 1$, $k = 2$ e $k = 5$, falhas parciais são observadas. Ainda assim, para $k \geq 6$ as sequências passaram com sucesso.

Em [14], o tamanho mínimo do barramento para que a sequência gerada pelo mapa logístico implementado fosse aprovada em todos os testes do NIST é de 45 bits. Portanto, com os resultados na tabela 6, verifica-se que o PRNG proposto, baseado no k-mapa logístico, permite uma implementação para barramentos menores.

6 CONCLUSÃO

Como descrito anteriormente, o objetivo dessa pesquisa foi a implementação, em FPGA, de um PRNG baseado no conceito do k-mapa logístico apresentado em [12] utilizando representação em ponto-fixo não sinalizada. Para tanto, desenvolveu-se o circuito apresentado na figura 25, com um barramento de N = 32 bits, sendo $k_{max} = 10$. Tal circuito é composto por duas etapas, uma que realiza a operação do mapa logístico original, descrito pela equação (2) e outra que aplica o conceito do k-mapa logístico apresentado na equação (3).

A partir da síntese desse PRNG proposto obteve-se que o uso de *hardware* foi semelhante a trabalhos já existentes em [13] e [14], como mostrado na tabela 4. Já o *throughput* estimado para esse PRNG é maior que os presentes em [13] e [14], como observa-se na tabela 5.

A fim de atestar a correta implementação do conceito do k-mapa logístico e analisar sequências geradas através do conjunto de testes do NIST, foi desenvolvido o sistema mostrado na figura 28. Esse sistema utiliza do HPS e uma máquina de estados finitos, projetada na FPGA, para gerenciar o funcionamento do PRNG, gerando e armazenando em um arquivo, no cartão µSD, 32768 valores, com 32 bits cada, totalizando 2^{20} bits.

A partir das sequências geradas, utilizando cada implementação com $0 \leq k \leq 10$, foi possível construir as curvas de distribuição de frequência , figura 30, assim como os diagramas de Poincaré, figura 31 e 32. E ao comparar esses gráficos com os das figuras 18 e 19, foi possível constatar que o conceito do k-mapa logístico foi realmente aplicado.

Aplicando as sequências geradas ao conjunto de testes do NIST foram obtidos ótimos resultados para $k \geq 6$, como apresentado na tabela 6.

Conclui-se então que a implementação do circuito proposto, na figura 25, com um valor alto de k consegue gerar sequências com boas propriedades aleatórias, logo, é uma ótima abordagem para um PRNG. Importante ressaltar que para sua utilização em aplicações criptográficas são necessários mais estudos e testes.

Para trabalhos futuros sugere-se o estudo do comportamento desse PRNG quando atacado, analisando a robustez do gerador. Outro ponto de grande interesse é a implementação de um sistema supervisor que verifique as propriedades aleatórias em tempo real da sequência gerada, podendo, se necessário, reinicializar o gerador com um novo valor inicial x_0.

REFERÊNCIAS[3]

1 MAY, R. M. Simple Mathematical Models With Very Complicated Dynamics. **Nature**, vol. 261, pp. 459-467, 2011. doi: 10.1038/261459a0.

2 EVANS, D. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. **Cisco Internet Business Solutions Group (IBSG)**, 2011.

3 MARENA, T. Hardware security in the IoT. **Embedded Computing Design**, 2015.

4 SADEGHI, A-R.; WACHSMANN, C.; WAIDNER, M. Security and Privacy Challenges in Industrial Internet of Things. **IEEE Design Automation Conference**, New York, pp. 54:1–54:6, 2015. doi: 10.1145/2744769.2747942.

5 ROSTAMI, M.; KOUSHANFAR, F.; KARRI, R. A Primer on Hardware Security: Models, Methods, and Metrics. **Proceedings of IEEE**, vol. 102, n. 8, pp. 1283–1295, 2014. doi: 10.1109/JPROC.2014.2335155.

6 O'FLYNN, C.; CHEN, Z. D. ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research. **Constructive Side-Channel Analysis and Secure Design**, E. Prouff, Ed. Springer International Publishing, pp. 243–260, 2014.

7 KUMARAGE, H.; KHALIL, I.; ALABDULATIF, A.; TARI, Z.; YI, X. Secure Data Analytics for Cloud-Integrated Internet of Things Applications **IEEE Cloud Computing**, vol. 3, n. 2, pp. 46-56, 2016. doi: 10.1109/MCC.2016.30.

8 RUKHIN, A.; SOTO, J.; NECHVATAL, J.; SMID, M.; BARKER, E.; LEIGH, S.; LEVENSON, M.; VANGEL, M.; BANKS, D.; HECKERT, A.; DRAY, J.; VO, S. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. **NIST Special Publication 800-22**, National Institute of Standards and Technology, 2010.

[3] De acordo com a Associação Brasileira de Normas Técnicas (ABNT NBR 6023).

9 FISCHER, V.; DRUTAROVSKY, M. True Random Number Generation on FPGA. **Training School on Trustworthy Manufacturing and Utilization of Secure Devices**, Lisboa, Portugal, 2014.

10 YANG, K.; BLAAUW, D.; SYLVESTER, D. An All-Digital Edge Racing True Random Number Generator Robust Against PVT Variations. **IEEE Journal of Solid-State Circuits**, vol. 51, n. 4, pp. 1022–1031, 2016. doi: 10.1109/JSSC.2016.2519383.

11 YANG, B.; ROZIC, V; MENTENS, N.; DEHAENE, W.; VERBAUWHEDE, I. TOTAL: TRNG on-the-fly testing for attack detection using Lightweight hardware. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**, pp 127-132, 2016.

12 MACHICAO, J. **Padrões e pseudo-aleatoriedade usando sistemas complexos**. 2017. Tese (Doutorado em Ciências) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 2017.

13 DABAL, P.; PELKA, R. A Chaos-Based Pseudo-Random Bit Generator Implemented in FPGA Device. **IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems**, 2011. doi: 10.1109/DDECS.2011.5783069.

14 SAYED, W. S.; RADWAN, A. G.; REZK, A. A.; FAHMY, A. H. Finite Precision Logistic Map between Computational Efficiency and Accuracy with Encryption Applications. **Complexity**, v. 2017, 21 p., 2017. doi: 10.1155/2017/8692046.

15 KNUTH, D. E. **The Art of Computer Programming**. 3 ed. Addison-Wesley, 1997. v. 2: Seminumerical Algorithms.

16 VELJKOVIC, F.; ROZIC, V.; VERBAUWHEDE, I. Low-Cost Implementations of On-the-Fly Tests for Random Number Generators. **Design, Automation & Test in Europe Conference & Exhibition (DATE)**, pp 959-964, 2012. doi: 10.1109/DATE.2012.6176635.

17 AVAROĞLU, E.; TUNCER T.; ÖZER, A. B.; ERGEN B.; TÜRK, M. A novel chaos-based post-processing for TRNG. **Nonlinear Dynamics**, v. 81, pp 189-199, 2015. doi: 10.1007/s11071-015-1981-9.

18 MATSUMOTO, M.; YASUDA, S.; OHBA, R.; IKEGAMI, K.; TANAMOTO, T.; FUJITA, S. 1200 μm^2 Physical Random-Number Generators Based on SiN MOSFET for Secure Smart-Card Application. **IEEE International Solid-State Circuits Conference (ISSCC) – Digest of Technical Papers**, San Francisco, pp. 414–624, 2008. doi: 10.1109/ISSCC.2008.4523233.

19 PETRIE, C. S.; CONNELLY, J. A. A noise-based IC random number generator for applications in cryptography. **IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications**, vol. 47, no. 5, pp. 615–621, 2000. doi: 10.1109/81.847868.

20 HATA, H.; ICHIKAWA, S. FPGA Implementation of Metastability-Based True Random Number Generator. **IEICE Transactions on Information and Systems**, vol. E95, no. 2, pp. 426-436, 2012. doi: 10.1587/transinf.E95.D.426.

21 CARREIRA, L. B. **Gerador de números aleatórios digital, reconfigurável, de baixa latência com detecção e correção de viés de saída**. 2019. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2019.

22 MARKETTOS, A. T.; MOORE, S. W. The Frequency Injection Attack on Ring-Oscillator-Based True Number Generators. **Cryptographic Hardware and Embedded Systems – CHES**, Lausanne, 2009. doi: 10.1007/978-3-642-04138-9_23.

23 SALUJA, K. K. **Linear Feedback Shift Registers Theory and Applications**. Department of Electrical and Computer Engineering, University of Wisconsin-Madison, 1987.

24 D'AMORE, R. **VHDL Descrição e Síntese de Circuitos Digitais**. 2. Ed., Rio de Janeiro: LTC, 2012.

25 PATIDAR, V.; SUD, K. K.; PAREEK, N. K. A Pseudo Random Bit Generator Based on Chaotic Logistic Map and its Statistical Testing. **Informatica**, v. 33, pp. 441-452, 2009.

26 SURESH, V. B.; ANTONIOLI, D.; BURLESON, W. P. On-chip Lightweight Implementation of Reduced NIST Randomness Test Suite. **IEEE International Symposium**

on Hardware-Oriented Security and Trust (HOST), Austin, 2013. doi: 10.1109/HST.2013.6581572.

27 BROWN, S.; VRANESIC, Z. **Fundamentals of Digital Logic with Verilog Design**. McGraw-Hill, 2002.

28 ALTERA. **AB1-01225: What is an SoC FPGA?.** Altera, 2014. Architecture brief.

29 HARRIS, D. M.; HARRIS, S. L. **Digital Design and Computer Architecture**. Morgan Kaufmann Publishers, 2007.

30 KOTAKI, M. M. A.; LUPPE, M. FPGA Implementation of a Pseudorandom Number Generator Based on k-Logistic Map. **IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)**, San Jose, pp.1-4, 2020. doi: 10.1109/LASCAS45839.2020.9068999.

31 NEPOMUCENO, E. G.; LIMA, A. M.; Arias-García, J.; PERC, M., REPNIK, R. Minimal Digital Chaotic System. **Chaos, Solitons & Fractals**, v. 120, pp. 62-66, 2019. doi: 10.1016/j.chaos.2019.01.019.

APÊNDICES

APÊNDICE A – Publicação gerada durante o período de mestrado

KOTAKI, M. M. A.; LUPPE, M. FPGA Implementation of a Pseudorandom Number Generator Based on k-Logistic Map. **IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)**, San Jose, pp.1-4, 2020. doi: 10.1109/LASCAS45839.2020.9068999.

APÊNDICE B – Descrição do PRNG baseado no *k*-mapa logístico (VHDL)

```
--=================================================================
-- Descricao do PRNG baseado no k-mapa logistico
-- Matheus Mitsuo de A. Kotaki, São Carlos-SP
-- EESC - USP - 2021
-- Link GitHub: https://github.com/kot4ki/kot4ki-c5_k_map_log.git
--=================================================================

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity maplog is

    generic (
        size : natural := 32
    );

    port(
        -- Entradas
        x0    : in std_logic_vector (size-1 DOWNTO 0);
        rst   : in std_logic;
        clk   : in std_logic;
        start : in std_logic;

        --Saidas
        xt_out : out std_logic_vector (size-1 DOWNTO 0)
    );

end maplog;

architecture archmap of maplog is

--Componente Complemento de dois
component twocomp

    generic(
        size: natural
    );

    port(
        n            : in std_logic_vector  (size-1 DOWNTO 0);
        result       : out std_logic_vector (size-1 DOWNTO 0)
    );

end component;

--Componente Multiplicacao
component mult

    port
    (
        dataa            : in  std_logic_vector (size-1   DOWNTO 0);
        datab            : in  std_logic_vector (size-1   DOWNTO 0);
        result           : out std_logic_vector (2*size-1 DOWNTO 0)
    );

end component;
```

```
component soma1
    generic(
        P   : natural
    );

    port(
        dataa           : in std_logic_vector  (P DOWNTO 0);
        datab           : in std_logic_vector  (P DOWNTO 0);
        result          : out std_logic_vector (P DOWNTO 0)
    );
end component;

--Signals
    signal xt     : std_logic_vector (size-1    DOWNTO 0); -- valor de
entrada da equacao
    signal xt1    : std_logic_vector (size-1    DOWNTO 0); -- valor de
realimentacao
    signal d      : std_logic_vector (size-1    DOWNTO 0); -- entrada do
flip-flop
    signal eq1    : std_logic_vector (size-1    DOWNTO 0); -- resultado do
complemento de dois
    signal eq2    : std_logic_vector (2*size-1  DOWNTO 0); -- resultado da
multiplicacao
    signal s1     : std_logic_vector (2*size-6  DOWNTO 0);
    signal s2     : std_logic_vector (2*size-6  DOWNTO 0);
    signal s3     : std_logic_vector (2*size-9  DOWNTO 0);
    signal s4     : std_logic_vector (2*size-9  DOWNTO 0);
    signal s5     : std_logic_vector (2*size-12 DOWNTO 0);
    signal s6     : std_logic_vector (2*size-12 DOWNTO 0);
    signal s7     : std_logic_vector (2*size-15 DOWNTO 0);
    signal s8     : std_logic_vector (2*size-15 DOWNTO 0);
    signal s9     : std_logic_vector (2*size-18 DOWNTO 0);
    signal s10    : std_logic_vector (2*size-18 DOWNTO 0);
    signal s11    : std_logic_vector (2*size-21 DOWNTO 0);
    signal s12    : std_logic_vector (2*size-21 DOWNTO 0);
    signal s13    : std_logic_vector (2*size-24 DOWNTO 0);
    signal s14    : std_logic_vector (2*size-24 DOWNTO 0);
    signal s15    : std_logic_vector (2*size-27 DOWNTO 0);
    signal s16    : std_logic_vector (2*size-27 DOWNTO 0);
    signal s17    : std_logic_vector (2*size-30 DOWNTO 0);
    signal s18    : std_logic_vector (2*size-30 DOWNTO 0);
    signal s19    : std_logic_vector (2*size-33 DOWNTO 0);
    signal s20    : std_logic_vector (2*size-33 DOWNTO 0);
    signal soma01 : std_logic_vector (2*size-6  DOWNTO 0);
    signal soma02 : std_logic_vector (2*size-9  DOWNTO 0);
    signal soma03 : std_logic_vector (2*size-12 DOWNTO 0);
    signal soma04 : std_logic_vector (2*size-15 DOWNTO 0);
    signal soma05 : std_logic_vector (2*size-18 DOWNTO 0);
    signal soma06 : std_logic_vector (2*size-21 DOWNTO 0);
    signal soma07 : std_logic_vector (2*size-24 DOWNTO 0);
    signal soma08 : std_logic_vector (2*size-27 DOWNTO 0);
    signal soma09 : std_logic_vector (2*size-30 DOWNTO 0);
    signal soma10 : std_logic_vector (2*size-33 DOWNTO 0);

begin

    -- Multiplexador
    process (rst,xt1)
    begin
        if (rst = '1') then
```

```
            d <= x0; -- se reset for ativado xt recebe o valor inicial x0
        else
            d <= xt1; -- se nao for ativado o reset, xt passa a valer xt1
        end if;
    end process;

    -- Registrador
    process (clk)
    begin
        if(clk'event and clk = '1') then
            if start ='0' then
                xt<=d;
            end if;
        end if;
    end process;

    -- Unidade aritmetica

    -- 1-xt
    sub: twocomp
        generic map(size)
        port map (
                    n      => xt,
                    result => eq1
                    );

    -- multiplicacao por xt

    multplic: mult
        port map (
                    dataa  => eq1,
                    datab  => xt,
                    result => eq2
                    );

    -- mult. por mi (realimentacao original)
    xt1 <= eq2(2*size-3 DOWNTO size-2);

    --multiplicacao por 4 feita em s1 e s2


    --mult. por 10 (k=1):

    s1 <= eq2(2*size-6 DOWNTO 0);
    s2 <= eq2(2*size-4 DOWNTO 2);

    somaa: soma1
        generic map(2*size-6)
        port map (
                    dataa  => s1,
                    datab  => s2,
                    result => soma01
        );



--mult. por 100 (k=2)

    s3 <= soma01(2*size-9 DOWNTO 0);
    s4 <= soma01(2*size-7 DOWNTO 2);
```

```
    somaB: soma1
        generic map(2*size-9)
        port map (
                    dataa  => s3,
                    datab  => s4,
                    result => soma02
        );


--mult. por 1000 (k=3)

    s5 <= soma02(2*size-12 DOWNTO 0);
    s6 <= soma02(2*size-10 DOWNTO 2);

    somaC: soma1
        generic map(2*size-12)
        port map (
                    dataa  => s5,
                    datab  => s6,
                    result => soma03
        );



--mult por 10000 (k=4)
    s7 <= soma03(2*size-15 DOWNTO 0);
    s8 <= soma03(2*size-13 DOWNTO 2);

    somaD: soma1
        generic map(2*size-15)
        port map (
                    dataa  => s7,
                    datab  => s8,
                    result => soma04
        );


--mult. por 100000 (k=5)

    s9  <= soma04(2*size-18 DOWNTO 0);
    s10 <= soma04(2*size-16 DOWNTO 2);

    somaE: soma1
        generic map(2*size-18)
        port map (
                    dataa  => s9,
                    datab  => s10,
                    result => soma05
        );

-- mult. por 1000000 (k=6)

    s11 <= soma05(2*size-21 DOWNTO 0);
    s12 <= soma05(2*size-19 DOWNTO 2);

    somaF: soma1
        generic map(2*size-21)
        port map (
                    dataa  => s11,
                    datab  => s12,
                    result => soma06
```

```
        );

--mult por 10^7 (k=7)

    s13 <= soma06(2*size-24 DOWNTO 0);
    s14 <= soma06(2*size-22 DOWNTO 2);

    somaG: soma1
        generic map(2*size-24)
        port map (
                    dataa  => s13,
                    datab  => s14,
                    result => soma07
        );

--mult por 10^8 (k=8)

    s15 <= soma07(2*size-27 DOWNTO 0);
    s16 <= soma07(2*size-25 DOWNTO 2);

    somaH: soma1
        generic map(2*size-27)
        port map (
                    dataa  => s15,
                    datab  => s16,
                    result => soma08
        );

--mult por 10^9 (k=9)

    s17 <= soma08(2*size-30 DOWNTO 0);
    s18 <= soma08(2*size-28 DOWNTO 2);

    somaI: soma1
        generic map(2*size-30)
        port map (
                    dataa  => s17,
                    datab  => s18,
                    result => soma09
        );

--mult por 10^10 (k=10)

    s19 <= soma09(2*size-33 DOWNTO 0);
    s20 <= soma09(2*size-31 DOWNTO 2);

    somaJ: soma1
        generic map(2*size-33)
        port map (
                    dataa  => s19,
                    datab  => s20,
                    result => soma10
        );


    -- k=0 (mapa original)
    --xt_out <= xt1;

    --k=1
    --xt_out <= soma01(2*size-6 DOWNTO size-5);
```

```
    --k=2
    --xt_out <= soma02(2*size-9 DOWNTO size-8);

    --k=3
    --xt_out <= soma03(2*size-12 DOWNTO size-11);

    --k=4
    --xt_out <= soma04(2*size-15 DOWNTO size-14);

    --k=5
    --xt_out <= soma05(2*size-18 DOWNTO size-17);

    --k=6
    --xt_out <= soma06(2*size-21 DOWNTO size-20);

    --k=7
    --xt_out <= soma07(2*size-24 DOWNTO size-23);

    --k=8
    --xt_out <= soma08(2*size-27 DOWNTO size-26);

    --k=9
    --xt_out <= soma09(2*size-30 DOWNTO size-29);

    --k=10
    xt_out <= soma10(2*size-33 DOWNTO size-32);

end archmap;
```

APÊNDICE C – Descrição do componente FSM (VHDL)

```
--==================================================================
-- Descricao da maquina de estados finitos (FSM)
-- Matheus Mitsuo de A. Kotaki, São Carlos-SP
-- EESC - USP - 2021
-- Link GitHub: https://github.com/kot4ki/kot4ki-c5_k_map_log.git
--==================================================================

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FSM is
port(
    clk_fsm, rst_fsm          : in std_logic;
    start_g                   : in std_logic;
    count_q                   : in std_logic_vector(14 downto 0);

    reset_prng, enable_count, clear_counter, start_prng, wr_enable,
done_gen: out std_logic
    );
end FSM;

architecture arch of FSM is

    type state_type is (std_by, reset, generating, done);
    signal current_state, next_state: state_type;

begin

    process(clk_fsm, rst_fsm)
    begin
        if (rst_fsm = '1') then -- se resetado vai para o estado = std_by
            current_state <= std_by;
        elsif (clk_fsm'event and clk_fsm = '1') then -- senão, atualiza o
estado
            current_state <= next_state;
        else
            null;
        end if;
    end process;

    process(current_state, start_g, count_q)
    begin

        case current_state is
            when std_by => --se estado = std_by (espera)

                if start_g = '1' then
                    next_state      <= reset;
                    reset_prng      <= '1';
                    enable_count    <= '0';
                    clear_counter   <= '1';
                    start_prng      <= '0';
                    wr_enable       <= '0';
                    done_gen        <= '0';
                else
                    next_state      <= std_by;
```

```
                    reset_prng      <= '1';
                    enable_count    <= '0';
                    clear_counter   <= '1';
                    start_prng      <= '1';
                    wr_enable       <= '0';
                    done_gen        <= '0';
                end if;

            when reset =>  --se estado = reset (reinicio)
                    next_state       <= generating;
                    reset_prng       <= '0';
                    enable_count     <= '1';
                    clear_counter    <= '0';
                    start_prng       <= '0';
                    wr_enable        <= '1';
                    done_gen         <= '0';

            when generating =>  --se estado = generating (gerando)
                if count_q = "111111111111111" then
                    next_state     <= done;
                    reset_prng     <= '1';
                    enable_count   <= '0';
                    clear_counter  <= '1';
                    start_prng     <= '1';
                    wr_enable      <= '1';
                    done_gen       <= '0';
                else
                    next_state     <= generating;
                    reset_prng     <= '0';
                    enable_count   <= '1';
                    clear_counter  <= '0';
                    start_prng     <= '0';
                    wr_enable      <= '1';
                    done_gen       <= '0';
                end if;

            when done =>  --se estado = done (concluido)

                if start_g = '0' then
                    next_state      <= std_by;
                    reset_prng      <= '1';
                    enable_count    <= '0';
                    clear_counter   <= '1';
                    start_prng      <= '1';
                    wr_enable       <= '0';
                    done_gen        <= '0';
                else
                    next_state     <= done;
                    reset_prng     <= '1';
                    enable_count   <= '0';
                    clear_counter  <= '1';
                    start_prng     <= '1';
                    wr_enable      <= '0';
                    done_gen       <= '1';
                end if;

        end case;
    end process;
end arch;
```

APÊNDICE D – Descrição da conexão entre os componentes: FSM, RAM, CONTADOR e PRNG (VHDL)

```
--=================================================================
-- Descricao do PRNG baseado no k-mapa logistico
-- Matheus Mitsuo de A. Kotaki, São Carlos-SP
-- EESC - USP - 2021
-- Link GitHub: https://github.com/kot4ki/kot4ki-c5_k_map_log.git
--=================================================================

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity prng is

    generic (
        size : natural := 32
    );

    port(
        --Entradas
        clk_prng : in std_logic;
        clk_read : in std_logic;
        x0_prng  : in std_logic_vector (size-1 DOWNTO 0);
        start_gen: in std_logic;
        read_addr: in std_logic_vector (14 DOWNTO 0);
        reset_fsm: in std_logic;

        --Saidas
        xt_out_prng: out std_logic_vector (size-1 DOWNTO 0);
        LED_start   : out std_logic;
        generated   : out std_logic

    );

end prng;

architecture archprng of prng is


component maplog

    generic (
        size : natural
    );

    port(
        -- Entradas
        x0   : in std_logic_vector (size-1 DOWNTO 0);
        rst  : in std_logic;
        clk  : in std_logic;
        start: in std_logic;

        --Saidas
        xt_out : out std_logic_vector (size-1 DOWNTO 0)
    );
```

```
end component;


component ram_module IS
    PORT
    (
        data        : in std_logic_vector (31 DOWNTO 0);
        rdaddress   : in std_logic_vector (14 DOWNTO 0);
        rdclock     : in std_logic ;
        wraddress     : in std_logic_vector (14 DOWNTO 0);
        wrclock       : in std_logic  := '1';
        wren          : in std_logic  := '0';
        q             : out std_logic_vector (31 DOWNTO 0)
    );
end component;

component counter0 is
    PORT
        (
            clock       : in std_logic ;
            cnt_en      : in std_logic ;
            sclr        : in std_logic ;
            q           : out std_logic_vector (14 DOWNTO 0)
        );
end component;

component FSM is

    port(
        clk_fsm, rst_fsm: in std_logic;
        start_g         : in std_logic;
        count_q         : in std_logic_vector(14 DOWNTO 0);

        reset_prng, enable_count, clear_counter, start_prng, wr_enable,
done_gen: out std_logic
        );

end component;

signal xt_out_map  :std_logic_vector (size-1 DOWNTO 0) := X"00000000";
signal clk_ram     :std_logic;
signal we_ram      :std_logic := '0';
signal rst_prng    :std_logic := '1';
signal strt_prng   :std_logic := '1';
signal clr_counter :std_logic := '0';
signal en_count    :std_logic := '0';
signal count       :std_logic_vector (14 DOWNTO 0) := "000000000000000";

begin


    map_log: maplog             -- PRNG baseado no k-mapa logistico
        generic map(size)
        port map(
            x0     =>x0_prng,
            rst    =>rst_prng,
            clk    =>clk_prng,
            start  =>strt_prng, --start_prng = 0 gera novo valor
            xt_out =>xt_out_map
```

```
        );


    ram0:   ram_module  --RAM para armazenar 32768 valores de 32 bits cada
    port map
    (
        data         => xt_out_map,
        rdaddress    => read_addr,
        rdclock      => clk_read,
        wraddress    => count,
        wrclock      => clk_ram,
        wren         => we_ram,
        q            => xt_out_prng
    );


    count1: counter0 --contador de 15 bits
    port map(
        clock    =>clk_prng,
        cnt_en   => en_count,
        sclr     => clr_counter,
        q        => count

    );

    FSM0: FSM  --maquina de estados finitos
    port map(
    clk_fsm         =>clk_prng,
    rst_fsm         =>reset_fsm,
    start_g         =>start_gen,
    count_q         =>count,
    reset_prng      =>rst_prng,
    enable_count    =>en_count,
    clear_counter   =>clr_counter,
    start_prng      =>strt_prng,
    wr_enable       => we_ram,
    done_gen        => generated

    );


    clk_ram   <= not clk_prng; --armazena na RAM na borda de descida do
relógio do PRNG
    LED_start <= not strt_prng;



end archprng;
```

APÊNDICE E – Descrição do sistema da figura 28 (Verilog)

```
//===================================================================
// Descricao do sistema de geracao e armazenamento de
// 2^20 bits aleatorios
// Matheus Mitsuo de A. Kotaki, São Carlos-SP
// EESC - USP - 2021
// Link GitHub: https://github.com/kot4ki/kot4ki-c5_k_map_log.git
//===================================================================

module DE10_NANO_SoC_GHRD(

    //////////// CLOCK //////////
    input                   FPGA_CLK1_50,
    input                   FPGA_CLK2_50,
    input                   FPGA_CLK3_50,

    //////////// HDMI //////////
    inout                   HDMI_I2C_SCL,
    inout                   HDMI_I2C_SDA,
    inout                   HDMI_I2S,
    inout                   HDMI_LRCLK,
    inout                   HDMI_MCLK,
    inout                   HDMI_SCLK,
    output                  HDMI_TX_CLK,
    output       [23: 0]    HDMI_TX_D,
    output                  HDMI_TX_DE,
    output                  HDMI_TX_HS,
    input                   HDMI_TX_INT,
    output                  HDMI_TX_VS,

    //////////// HPS //////////
    inout                   HPS_CONV_USB_N,
    output       [14: 0]    HPS_DDR3_ADDR,
    output       [ 2: 0]    HPS_DDR3_BA,
    output                  HPS_DDR3_CAS_N,
    output                  HPS_DDR3_CK_N,
    output                  HPS_DDR3_CK_P,
    output                  HPS_DDR3_CKE,
    output                  HPS_DDR3_CS_N,
    output       [ 3: 0]    HPS_DDR3_DM,
    inout        [31: 0]    HPS_DDR3_DQ,
    inout        [ 3: 0]    HPS_DDR3_DQS_N,
    inout        [ 3: 0]    HPS_DDR3_DQS_P,
    output                  HPS_DDR3_ODT,
    output                  HPS_DDR3_RAS_N,
    output                  HPS_DDR3_RESET_N,
    input                   HPS_DDR3_RZQ,
    output                  HPS_DDR3_WE_N,
    output                  HPS_ENET_GTX_CLK,
    inout                   HPS_ENET_INT_N,
    output                  HPS_ENET_MDC,
    inout                   HPS_ENET_MDIO,
    input                   HPS_ENET_RX_CLK,
    input        [ 3: 0]    HPS_ENET_RX_DATA,
    input                   HPS_ENET_RX_DV,
    output       [ 3: 0]    HPS_ENET_TX_DATA,
    output                  HPS_ENET_TX_EN,
    inout                   HPS_GSENSOR_INT,
```

```
    inout                   HPS_I2C0_SCLK,
    inout                   HPS_I2C0_SDAT,
    inout                   HPS_I2C1_SCLK,
    inout                   HPS_I2C1_SDAT,
    inout                   HPS_KEY,
    inout                   HPS_LED,
    inout                   HPS_LTC_GPIO,
    output                  HPS_SD_CLK,
    inout                   HPS_SD_CMD,
    inout       [ 3: 0]     HPS_SD_DATA,
    output                  HPS_SPIM_CLK,
    input                   HPS_SPIM_MISO,
    output                  HPS_SPIM_MOSI,
    inout                   HPS_SPIM_SS,
    input                   HPS_UART_RX,
    output                  HPS_UART_TX,
    input                   HPS_USB_CLKOUT,
    inout       [ 7: 0]     HPS_USB_DATA,
    input                   HPS_USB_DIR,
    input                   HPS_USB_NXT,
    output                  HPS_USB_STP,

    //////////// KEY //////////
    input       [ 1: 0]     KEY,

    //////////// LED //////////
    output      [ 7: 0]     LED,

    //////////// SW //////////
    input       [ 3: 0]     SW,

     output clk_pin,
     output start_pin,
     output reset_pin,
     output done_pin

);


//=======================================================
//  REG/WIRE declarations
//=======================================================
wire hps_fpga_reset_n;
wire     [1: 0]     fpga_debounced_buttons;
wire     [6: 0]     fpga_led_internal;
wire     [2: 0]     hps_reset_req;
wire                hps_cold_reset;
wire                hps_warm_reset;
wire                hps_debug_reset;
wire     [27: 0]    stm_hw_events;
wire                fpga_clk_50;


wire prng_clk;
wire rd_clk;
wire [1:0] start_reset;
wire [31:0] x0;
wire [31:0] xt_out;
wire [14:0] raddress;
```

```
wire done_gen;


// connection of internal logics
assign fpga_clk_50 = FPGA_CLK1_50;
assign stm_hw_events = {{15{1'b0}}, SW, fpga_led_internal,
fpga_debounced_buttons};

assign clk_pin = rd_clk;
assign start_pin = start_reset[1];
assign reset_pin = start_reset[0];



//=======================================================
//  Structural coding
//=======================================================
soc_system u0(
               //Clock&Reset
               .clk_clk(FPGA_CLK1_50),
//                            clk.clk
               .reset_reset_n(hps_fpga_reset_n),
//                          reset.reset_n
               //HPS ddr3
               .memory_mem_a(HPS_DDR3_ADDR),
//                         memory.mem_a
               .memory_mem_ba(HPS_DDR3_BA),
//                                .mem_ba
               .memory_mem_ck(HPS_DDR3_CK_P),
//                                .mem_ck
               .memory_mem_ck_n(HPS_DDR3_CK_N),
//                                .mem_ck_n
               .memory_mem_cke(HPS_DDR3_CKE),
//                                .mem_cke
               .memory_mem_cs_n(HPS_DDR3_CS_N),
//                                .mem_cs_n
               .memory_mem_ras_n(HPS_DDR3_RAS_N),
//                                .mem_ras_n
               .memory_mem_cas_n(HPS_DDR3_CAS_N),
//                                .mem_cas_n
               .memory_mem_we_n(HPS_DDR3_WE_N),
//                                .mem_we_n
               .memory_mem_reset_n(HPS_DDR3_RESET_N),
//                                .mem_reset_n
               .memory_mem_dq(HPS_DDR3_DQ),
//                                .mem_dq
               .memory_mem_dqs(HPS_DDR3_DQS_P),
//                                .mem_dqs
               .memory_mem_dqs_n(HPS_DDR3_DQS_N),
//                                .mem_dqs_n
               .memory_mem_odt(HPS_DDR3_ODT),
//                                .mem_odt
               .memory_mem_dm(HPS_DDR3_DM),
//                                .mem_dm
               .memory_oct_rzqin(HPS_DDR3_RZQ),
//                                .oct_rzqin
               //HPS ethernet
               .hps_0_hps_io_hps_io_emac1_inst_TX_CLK(HPS_ENET_GTX_CLK),
//    hps_0_hps_io.hps_io_emac1
```

```
                .hps_0_hps_io_hps_io_emac1_inst_TXD0(HPS_ENET_TX_DATA[0]),
//       .hps_io_emac1_inst_TXD0
                .hps_0_hps_io_hps_io_emac1_inst_TXD1(HPS_ENET_TX_DATA[1]),
//       .hps_io_emac1_inst_TXD1
                .hps_0_hps_io_hps_io_emac1_inst_TXD2(HPS_ENET_TX_DATA[2]),
//     .hps_io_emac1_inst_TXD2
                .hps_0_hps_io_hps_io_emac1_inst_TXD3(HPS_ENET_TX_DATA[3]),
//     .hps_io_emac1_inst_TXD3
                .hps_0_hps_io_hps_io_emac1_inst_RXD0(HPS_ENET_RX_DATA[0]),
//     .hps_io_emac1_inst_RXD0
                .hps_0_hps_io_hps_io_emac1_inst_MDIO(HPS_ENET_MDIO),
//     .hps_io_emac1_inst_MDIO
                .hps_0_hps_io_hps_io_emac1_inst_MDC(HPS_ENET_MDC),
//        .hps_io_emac1_inst_MDC
                .hps_0_hps_io_hps_io_emac1_inst_RX_CTL(HPS_ENET_RX_DV),
//     .hps_io_emac1_inst_RX_CTL
                .hps_0_hps_io_hps_io_emac1_inst_TX_CTL(HPS_ENET_TX_EN),
//      .hps_io_emac1_inst_TX_CTL
                .hps_0_hps_io_hps_io_emac1_inst_RX_CLK(HPS_ENET_RX_CLK),
//      .hps_io_emac1_inst_RX_CLK
                .hps_0_hps_io_hps_io_emac1_inst_RXD1(HPS_ENET_RX_DATA[1]),
//     .hps_io_emac1_inst_RXD1
                .hps_0_hps_io_hps_io_emac1_inst_RXD2(HPS_ENET_RX_DATA[2]),
//         .hps_io_emac1_inst_RXD2
                .hps_0_hps_io_hps_io_emac1_inst_RXD3(HPS_ENET_RX_DATA[3]),
//           .hps_io_emac1_inst_RXD3
                //HPS SD card
                .hps_0_hps_io_hps_io_sdio_inst_CMD(HPS_SD_CMD),
//                                .hps_io_sdio_inst_CMD
                .hps_0_hps_io_hps_io_sdio_inst_D0(HPS_SD_DATA[0]),
//                                .hps_io_sdio_inst_D0
                .hps_0_hps_io_hps_io_sdio_inst_D1(HPS_SD_DATA[1]),
//                                .hps_io_sdio_inst_D1
                .hps_0_hps_io_hps_io_sdio_inst_CLK(HPS_SD_CLK),
//                                .hps_io_sdio_inst_CLK
                .hps_0_hps_io_hps_io_sdio_inst_D2(HPS_SD_DATA[2]),
//                                .hps_io_sdio_inst_D2
                .hps_0_hps_io_hps_io_sdio_inst_D3(HPS_SD_DATA[3]),
//                                .hps_io_sdio_inst_D3
                //HPS USB
                .hps_0_hps_io_hps_io_usb1_inst_D0(HPS_USB_DATA[0]),
//                                .hps_io_usb1_inst_D0
                .hps_0_hps_io_hps_io_usb1_inst_D1(HPS_USB_DATA[1]),
//                                .hps_io_usb1_inst_D1
                .hps_0_hps_io_hps_io_usb1_inst_D2(HPS_USB_DATA[2]),
//                                .hps_io_usb1_inst_D2
                .hps_0_hps_io_hps_io_usb1_inst_D3(HPS_USB_DATA[3]),
//                                .hps_io_usb1_inst_D3
                .hps_0_hps_io_hps_io_usb1_inst_D4(HPS_USB_DATA[4]),
//                                .hps_io_usb1_inst_D4
                .hps_0_hps_io_hps_io_usb1_inst_D5(HPS_USB_DATA[5]),
//                                .hps_io_usb1_inst_D5
                .hps_0_hps_io_hps_io_usb1_inst_D6(HPS_USB_DATA[6]),
//                                .hps_io_usb1_inst_D6
                .hps_0_hps_io_hps_io_usb1_inst_D7(HPS_USB_DATA[7]),
//                                .hps_io_usb1_inst_D7
                .hps_0_hps_io_hps_io_usb1_inst_CLK(HPS_USB_CLKOUT),
//                                .hps_io_usb1_inst_CLK
                .hps_0_hps_io_hps_io_usb1_inst_STP(HPS_USB_STP),
//                                .hps_io_usb1_inst_STP
```

```
                .hps_0_hps_io_hps_io_usb1_inst_DIR(HPS_USB_DIR),
//                                .hps_io_usb1_inst_DIR
                .hps_0_hps_io_hps_io_usb1_inst_NXT(HPS_USB_NXT),
//                               .hps_io_usb1_inst_NXT
                //HPS SPI
                .hps_0_hps_io_hps_io_spim1_inst_CLK(HPS_SPIM_CLK),
//                                .hps_io_spim1_inst_CLK
                .hps_0_hps_io_hps_io_spim1_inst_MOSI(HPS_SPIM_MOSI),   //
.hps_io_spim1_inst_MOSI
                .hps_0_hps_io_hps_io_spim1_inst_MISO(HPS_SPIM_MISO),     //
.hps_io_spim1_inst_MISO
                .hps_0_hps_io_hps_io_spim1_inst_SS0(HPS_SPIM_SS),
//                                .hps_io_spim1_inst_SS0
                //HPS UART
                .hps_0_hps_io_hps_io_uart0_inst_RX(HPS_UART_RX),
//                                .hps_io_uart0_inst_RX
                .hps_0_hps_io_hps_io_uart0_inst_TX(HPS_UART_TX),
//                                .hps_io_uart0_inst_TX
                //HPS I2C1
                .hps_0_hps_io_hps_io_i2c0_inst_SDA(HPS_I2C0_SDAT),
//                                .hps_io_i2c0_inst_SDA
                .hps_0_hps_io_hps_io_i2c0_inst_SCL(HPS_I2C0_SCLK),
//                                .hps_io_i2c0_inst_SCL
                //HPS I2C2
                .hps_0_hps_io_hps_io_i2c1_inst_SDA(HPS_I2C1_SDAT),
//                                .hps_io_i2c1_inst_SDA
                .hps_0_hps_io_hps_io_i2c1_inst_SCL(HPS_I2C1_SCLK),
//                                .hps_io_i2c1_inst_SCL
                //GPIO
                .hps_0_hps_io_hps_io_gpio_inst_GPIO09(HPS_CONV_USB_N),=
// .hps_io_gpio_inst_GPIO09
                .hps_0_hps_io_hps_io_gpio_inst_GPIO35(HPS_ENET_INT_N),
//     .hps_io_gpio_inst_GPIO35
                .hps_0_hps_io_hps_io_gpio_inst_GPIO40(HPS_LTC_GPIO),
//     .hps_io_gpio_inst_GPIO40
                .hps_0_hps_io_hps_io_gpio_inst_GPIO53(HPS_LED),
//                                .hps_io_gpio_inst_GPIO53
                .hps_0_hps_io_hps_io_gpio_inst_GPIO54(HPS_KEY),
//                                .hps_io_gpio_inst_GPIO54
                .hps_0_hps_io_hps_io_gpio_inst_GPIO61(HPS_GSENSOR_INT),
//           .hps_io_gpio_inst_GPIO61
                //FPGA Partion
                .led_pio_external_connection_export(fpga_led_internal),
//    led_pio_external_connection.export
                .dipsw_pio_external_connection_export(SW),
//  dipsw_pio_external_connection.export

.button_pio_external_connection_export(fpga_debounced_buttons)
//  button_pio_external
                .hps_0_h2f_reset_reset_n(hps_fpga_reset_n),
//                  hps_0_h2f_reset.reset_n
                .hps_0_f2h_cold_reset_req_reset_n(~hps_cold_reset),
//       hps_0_f2h_cold_reset_req.reset_n
                .hps_0_f2h_debug_reset_req_reset_n(~hps_debug_reset),
//      hps_0_f2h_debug_reset_req.reset_n
                .hps_0_f2h_stm_hw_events_stm_hwevents(stm_hw_events),
//hps_0_f2h_stm_hw_events
                .hps_0_f2h_warm_reset_req_reset_n(~hps_warm_reset),       //
hps_0_f2h_warm_reset_req.reset_n
///////////////////////////////////////////////////////////////
    .seed_external_connection_export(x0),                  // valor inicial
```

```
    .read_clk_external_connection_export(rd_clk),  // relogio de leitura da
RAM
    .prng_out_external_connection_export(xt_out),     // valor de saida da
RAM
    .fsm_reset_external_connection_export(start_reset[0]),    // reinicia a
FSM
    .prng_start_external_connection_export(start_reset[1]),  // inicia o
funcionamento do sistema
    .rec_done_external_connection_export(done_pin), // fim do armazenamento
no cartao SD
    .r_addr_external_connection_export(raddress),  // endereco de leitura
da RAM
    .gen_done_external_connection_export(done_gen)      // sinal que indica
o fim da geracao

           );


    pll0 pll_prng (    //PLL

                .refclk(fpga_clk_50),
                .rst(1'b0),
                .outclk_0(prng_clk) //5 MHz

    );

    prng prng0 (  //FSM + CONTADOR + PRNG + RAM

        .clk_prng(prng_clk),
        .clk_read(rd_clk),
        .x0_prng(x0),
        .start_gen(start_reset[1]),
        .read_addr(raddress),
        .reset_fsm(start_reset[0]),
        .xt_out_prng(xt_out),
        .LED_start(LED[7]),
        .generated(done_gen)

    );

    assign LED[5] = done_gen;    // LED indicando o fim da geracao dos
valores


// Debounce logic to clean out glitches within 1ms
debounce debounce_inst(
             .clk(fpga_clk_50),
             .reset_n(hps_fpga_reset_n),
             .data_in(KEY),
             .data_out(fpga_debounced_buttons)
         );
defparam debounce_inst.WIDTH = 2;
defparam debounce_inst.POLARITY = "LOW";
defparam debounce_inst.TIMEOUT = 50000;          // at 50Mhz this is a
debounce time of 1ms
defparam debounce_inst.TIMEOUT_WIDTH = 16;     // ceil(log2(TIMEOUT))

// Source/Probe megawizard instance
hps_reset hps_reset_inst(
              .source_clk(fpga_clk_50),
              .source(hps_reset_req)
```

```
          );

altera_edge_detector pulse_cold_reset(
                         .clk(fpga_clk_50),
                         .rst_n(hps_fpga_reset_n),
                         .signal_in(hps_reset_req[0]),
                         .pulse_out(hps_cold_reset)
                    );
defparam pulse_cold_reset.PULSE_EXT = 6;
defparam pulse_cold_reset.EDGE_TYPE = 1;
defparam pulse_cold_reset.IGNORE_RST_WHILE_BUSY = 1;

altera_edge_detector pulse_warm_reset(
                         .clk(fpga_clk_50),
                         .rst_n(hps_fpga_reset_n),
                         .signal_in(hps_reset_req[1]),
                         .pulse_out(hps_warm_reset)
                    );
defparam pulse_warm_reset.PULSE_EXT = 2;
defparam pulse_warm_reset.EDGE_TYPE = 1;
defparam pulse_warm_reset.IGNORE_RST_WHILE_BUSY = 1;

altera_edge_detector pulse_debug_reset(
                         .clk(fpga_clk_50),
                         .rst_n(hps_fpga_reset_n),
                         .signal_in(hps_reset_req[2]),
                         .pulse_out(hps_debug_reset)
                    );
defparam pulse_debug_reset.PULSE_EXT = 32;
defparam pulse_debug_reset.EDGE_TYPE = 1;
defparam pulse_debug_reset.IGNORE_RST_WHILE_BUSY = 1;


endmodule
```

APÊNDICE F – Código em linguagem C do programa da figura 28

```
//============================================================
// Codigo em C do programa para execucao no HPS para geracao
// e armazenamento de 2^20 bits aleatorios
// Matheus Mitsuo de A. Kotaki, São Carlos-SP
// EESC - USP - 2021
// Link GitHub: https://github.com/kot4ki/kot4ki-c5_k_map_log.git
//=================================================================

#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <error.h>
#include <stdint.h>
#include <sys/mman.h>
#include "hps_0.h"

#define HPS_TO_FPGA_LW_BASE 0xFF200000
#define HPS_TO_FPGA_LW_SPAN 0x0020000

int main() {
    FILE *fp;
    void * lw_bridge_map = 0;
    int devmem_fd = 0;
    int result = 0;

    //Entradas
    uint32_t *clk_read = 0;
    uint32_t *done_generation = 0;
    uint32_t *xt_out = 0;
    uint32_t *fsm_rst = 0;


    // Saidas
    uint32_t *start = 0;
    uint32_t *x0 = 0;
    uint32_t *done = 0;
    uint32_t *read_addr = 0;


    devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
    if(devmem_fd < 0) {
        perror("devmem open");
        exit(EXIT_FAILURE);
    }


lw_bridge_map = (uint32_t*)mmap(NULL, HPS_TO_FPGA_LW_SPAN,
PROT_READ|PROT_WRITE,
         MAP_SHARED, devmem_fd, HPS_TO_FPGA_LW_BASE);
    if(lw_bridge_map == MAP_FAILED) {
        perror("devmem mmap");
        close(devmem_fd);
        exit(EXIT_FAILURE);
    }
```

```
    // Mapeamento dos PIOS
     clk_read = (uint32_t*)(lw_bridge_map + READ_CLK_BASE);
     done_generation = (uint32_t*)(lw_bridge_map + GEN_DONE_BASE);
     fsm_rst = (uint32_t*)(lw_bridge_map + FSM_RESET_BASE);
     start = (uint32_t*)(lw_bridge_map + PRNG_START_BASE);
     x0 = (uint32_t*)(lw_bridge_map + SEED_BASE);
     xt_out = (uint32_t*)(lw_bridge_map + PRNG_OUT_BASE);
     done = (uint32_t*)(lw_bridge_map + REC_DONE_BASE);
     read_addr = (uint32_t*)(lw_bridge_map + R_ADDR_BASE);


    *(done_generation+0x3) = 0x1;       //limpa o registrador do detector
de borda

    *fsm_rst = 0;
    *start = 0x0;
    *done = 0x0;
    *clk_read = 0x0;
    *read_addr = 0b000000000000000;
    *x0 = 0x3CD11396;

    *start = 0x1; //inicia a geracao
    while (!(*(done_generation+0x3)>0x0)){}      //espera a borda de subida
    *(done_generation+0x3) = 0x1;       //limpa o registrador do detector
de borda

    fp = freopen("file.txt", "w+", stdout);     //abre o arquivo para
armazenar os valores
    *clk_read = 0x1;


    while (!(*read_addr==0b111111111111111)){     //32768 vezes
        *clk_read = 0x0;
        *read_addr = *read_addr + 0b1;      //proximo endereco da RAM
        *clk_read = 0x1;
        printf("%x\r\n",*xt_out);        // armazena no cartao SD
        if (*read_addr==0b111111111111111){
            *clk_read = 0x0;
            *clk_read = 0x1;
            printf("%x\r\n",*xt_out);         // armazena no cartao SD
        }
    }
    fclose(fp);

    *done = 0x1;             //indica o fim da gravacao no cartao SD
    *start = 0x0;            // volta ao estado espera

    //limpa o mapeamento
    result = munmap(lw_bridge_map, HPS_TO_FPGA_LW_SPAN);
    if(result < 0) {
        perror("devmem munmap");
        close(devmem_fd);
        exit(EXIT_FAILURE);
    }

    close(devmem_fd);
    exit(EXIT_SUCCESS);

    return( 0 );
}
```