POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI

Instytut Telekomunikacji Multimedialnej

Praca dyplomowa inżynierska

Cyfrowy generator liczb losowych

Mateusz Madej, 140138

Promotor

dr inż. Jakub Nikonowicz

Poznań, 2022

Spis treści

Streszczenie 2
1. Wstęp 3
2. Liczby rzeczywiście losowe w cyberbezpieczeństwie 5
3. Konstrukcje generatorów liczb losowych 8
3.1. Wybrane metody pozyskiwania entropii 8
3.1.1 Teoria informacji i entropia 8
3.1.2 Generatory oparte o różne rodzaje szumów 10
3.1.3 Generatory bazujące na chaosie 12
3.1.4 Generatory implementowane na płytkach programowalnych 15
3.1.5 Generatory oparte o jitter 15
3.1.6 Generatory wykorzystujące pętlę synchronizacji fazy 16
3.1.7 Generatory bazujące na oscylatorach pierścieniowych 17
3.1.8 Generatory bazujące na metastabilności 17
3.1.9 Generatory oparte o interakcję człowiek-komputer 18
3.2. Postprocessing 19
3.2.1 Cel postprocessingu 19
3.2.2 Exclusive OR 19
3.2.3 Korekta Von Neumanna 20
3.2.4 Funkcje haszujące 20
3.2.5 Funkcja liniowa 20
3.2.6 Rejestr przesuwny 21
3.2.7 Inne podejście do postprocessingu 22
3.3 Analiza porównawcza 22
4. Implementacja generatora 25
4.1. Platforma sprzętowa 25
4.2. Akwizycja danych 36
4.3 Testy statystyczne 37
Podsumowanie 44
Bibliografia 46
Załączniki 49

Streszczenie

Niniejsza praca stanowi przegląd znanych rozwiązań dla generacji liczb rzeczywiście losowych. Podjęto się w niej uzasadnienia potrzeby stosowania liczb losowych dla zastosowań cyberbezpieczeństwa i podano przykłady miejsc ich wykorzystania. Następnie przytoczono artykuły naukowe traktujące o tematyce generacji liczb rzeczywiście losowych. Przegląd dostępnych rozwiązań podzielono na sekcję dotyczącą wybranych metod pozyskiwania entropii oraz sekcję stanowiącą przegląd dostępnych metod postprocessingu i zakończono krótką analizą porównawczą przedstawionych propozycji. Kolejnym etapem pracy była implementacja w FPGA generatora liczb rzeczywiście losowych o zakładanych właściwościach. Ciąg bitów losowych pobranych z generatora, poddano testom NIST, a także wyznaczono empiryczny rozkład wartości i metryki SAA. W podsumowaniu dokonano krótkiego komentarza obszaru zastosowań generatora.

1. Wstęp

Podejście do teorii prawdopodobieństwa zmieniało się na przestrzeni wieków. Wczesne noty dotyczące pojęcia szansy można znaleźć w wielu antycznych kulturach. Zjawisko losowości było używane przez wczesne cywilizacje. Przykładem może być powszechne użycie kości, które były wykorzystywane nie tylko do rozrywki, lecz często w cywilizacjach takich jak egipska czy chińska posługiwano się nimi do podejmowania decyzji. Pomimo to przez wieki natura losowości nie była obiektem teoretycznych spekulacji ludzi nauki i aż do średniowiecza nie było naukowego określenia losowości [1]. Jak wspomniano w [1] pierwsze znaczenie losowości możemy znaleźć w Hiszpańskim słowniku, gdzie słowo „losowy" jest zdefiniowane jako „Niepewny. Taki, który zależy od szczęścia i szansy", słowo „szansa" zdefiniowane jest jako przypuszczalna przyczyna zjawiska losowego. Część filozofów wiązała szansę z przyczynowością, na przykład grecki filozof Leukippos twierdził, że nic nie dzieje się losowo, wszystko ma swoją przyczynę i dzieje się z konieczności. Znany grecki filozof Arystoteles był zdania, że szansa jest wynikiem zbiegu okoliczności kilku niezależnych zdarzeń, których wynik interakcji jest ciężki do przewidzenia [1],[2]. Pod koniec XVIII wieku badania dotyczące zjawisk losowych wyszły poza świat gier, stały się częścią nauk przyrodniczych i społecznych. Przykładem wykorzystania zjawiska losowości w naukach przyrodniczych może być ustalanie krwi dziecka. Prawdopodobieństwo wystąpienia określonego rodzaju krwi u dziecka nie jest równe i zależne jest od krwi rodziców [1]. Na początku XX wieku ówczesne podejście to zaczęło się zmieniać. Wyróżniono zjawiska, które mogą być wyjaśnione za pomocą teorii prawdopodobieństwa i takie dla których rachunek prawdopodobieństwa nie ma zastosowania. Według [1] nawet jeżeli dla jakiegoś losowego zjawiska poprawnie zdefiniujemy zasady nim rządzące, nie znaczy to, że niepoprawny stanie się jego opis za pomocą teorii prawdopodobieństwa. Różni matematycy XX wieku próbowali dokonać formalizacji matematycznej teorii prawdopodobieństwa. Dzisiaj ta koncepcja podobnie jak pojęcie szansy i chaosu jest obecna w świadomości ludzi [1].

Zjawisko losowości i teoria prawdopodobieństwa są powszechnie używane w różnych dziedzinach życia. Poszczególne zastosowania cechują się innymi wymaganiami i stosują inne podejścia do pozyskiwania losowości. Początkowo, jak zostało to już wspomniane, losowość była używana w różnego rodzaju grach. Generacja losowości w grach dokonywana jest poprzez rzut kostką, tasowanie kart czy kręcenie kołem ruletki. Zastosowanie liczb losowych dotyka również tematów naukowych. Wiele eksperymentów w fizyce opiera się na analizie statystycznej uzyskanych wyników. Czasami w wyniku eksperymentu lub przeprowadzonego badania otrzymujemy zestaw danych, które następnie są analizowane. Specyficzne przypadki takie jak

zbieranie danych rentgenowskich mogą wymagać odróżnienia rzeczywistego sygnału podchodzącego z badanego źródła od szumu. Taka analiza statystyczna jest możliwa dzięki generacji liczb losowych. Kolejnym obszarem zastosowań losowości są wszelkiego rodzaju symulacje komputerowe zjawisk występujących w rzeczywistości. Wiele znanych zjawisk zdeterminowanych jest przez inne losowe zjawiska takie jak na przykład szumy. Symulacje komputerowe starając się możliwie rzetelnie te zjawiska odwzorować muszą być zaopatrzone w narzędzia generujące losowość [3].

Generowaniem liczb losowych zajmują się tak zwane generatory liczb losowych, które można podzielić na rzeczywiście losowe i pseudolosowe. Druga grupa generatorów charakteryzuje się tym, że kolejne powstające sekwencje są deterministyczne, to znaczy kolejne wartości powstają na podstawie poprzednich według określonego wzoru. Często generacja taka jest poprzedzona przez inicjalizację generatora pewną wartością początkową [4]. Generatory liczb rzeczywiście losowych nie posiadają jednego określonego wzoru, a ich losowość pochodzi ze zjawisk, których wyniku w poszczególnym momencie czasu nie jesteśmy w stanie przewidzieć, ponieważ mają na niego wpływ różne czynniki zewnętrze o charakterze losowym. Zależnie od zastosowań używane jest inne podejście do generacji liczb losowych. Liczby pseudolosowe są dla przykładu często używane w symulacjach statystycznych, na przykład metoda Monte Carlo pozwala na przybliżenie wartości liczby π dzięki generowaniu liczb losowych. Nie jest istotne, czy kolejno wygenerowane liczby będą rzeczywiście losowe, jednak generowane liczby powinny podlegać rozkładowi równomiernemu [5]. Istnieją takie obszary, gdzie liczby pseudolosowe mogą stanowić potencjalne zagrożenie ze względu na swoją przewidywalność. Mając na uwadze trudność w przewidzeniu kolejnych generowanych sekwencji generatory liczb rzeczywiście losowych posiadają szerokie zastosowanie w cyberbezpieczeństwie [3]. Poznanie lub przewidzenie generowanych liczb byłoby dużym zagrożeniem dla funkcjonowania systemu, w którym generator został wdrożony.

Ze względu na szerokie zastosowanie generatorów liczb rzeczywiście losowych za cel pracy obrano przegląd wybranych źródeł entropii i metod postprocessingu przedstawianych w artykułach naukowych i ich analizę porównawczą. Kolejnym celem jest implementacja wybranego generatora zgodnie z zasadami przedstawionymi w artykule biorąc pod uwagę zarówno źródło entropii, metodę jej pozyskiwania, jak i różne rodzaje postprocessingu opisywane w źródłach. Celem jest stworzenie generatora liczb rzeczywiście losowych z wykorzystaniem FPGA, który będzie spełniał pewne założenia pozwalające obiektywnie stwierdzić, że generowany przez niego ciąg liczb posiada dobre właściwości statystyczne.

2. Liczby rzeczywiście losowe w cyberbezpieczeństwie

Jednym z najważniejszych pojęć w szeroko rozumianym cyberbezpieczeństwie jest kryptografia, która zajmuje się zabezpieczaniem informacji. Operacja zabezpieczania informacji ma na celu przekształcenie jej w taki sposób, aby tylko docelowy adresat mógł poznać jej prawdziwe znaczenie. W dodatku operacja ta może zapewniać integralność, autentyczność oraz niezaprzeczalność informacji. Dzisiejsza skala wymiany informacji jest ogromna, zatem rośnie zapotrzebowanie na coraz bezpieczniejsze i szybsze sposoby zabezpieczania informacji. Rosnąca moc obliczeniowa komputerów również zwiększa wymagania dla stosowanych sposobów utajniania informacji. Jak się okazuje, zwykłe zastosowanie znanego obu stronom klucza do zaszyfrowania tekstu jawnego nie jest już wystarczające.

System kryptograficzny składa się z mniejszych bloków zwanych prymitywami kryptograficznymi. Systemy kryptograficzne cechują się pewną niezawodnością, a ich twórcy budując je z poszczególnych prymitywów powinni mieć pewność, że elementy te spełniają określone wymagania. W przypadku, gdy jeden z podstawowych składników danego systemu okaże się wadliwy, wówczas cały system należy uważać za wadliwy. Twórcy prymitywów kryptograficznych dają innym twórcom narzędzia do tworzenia większych systemów. Do ich wytworzenia sami muszą jednak korzystać z pewnych mniejszych bloków i narzędzi, takich jak teoria prawdopodobieństwa i liczby losowe.

Liczby losowe posiadają szerokie zastosowanie w kryptografii i są dziś krytycznym elementem wielu systemów. Niezależnie od tego w jakim systemie i w którym miejscu tego systemu wprowadzimy ciąg liczb losowych, musi on spełniać następujące wymagania [6]:

- Wystąpienie każdej z liczb losowych powinno być tak samo prawdopodobne, a zatem rozkład prawdopodobieństwa powinien mieć postać rozkładu równomiernego. Spełnienie tego wymagania utrudnia ataki skupiające się na najczęściej występujących wartościach. Jako przykład można podać rozkład normalny. Przy założeniu, że posiadamy generator liczb losowych generujący wartości podlegające rozkładowi normalnemu, złośliwy użytkownik podczas zgadywania jaka jest następna liczba losowa wygenerowana przez nasz system podając wartość oczekiwaną zwiększyłby swoje prawdopodobieństwo na odgadnięcie kolejnej generowanej liczby, ponieważ jak wiemy rozkład normalny osiąga swoje maksimum w wartości oczekiwanej. Nie oznacza to jednak, że liczby losowe podlegające rozkładowi normalnemu nie

są stosowane, jednak z punktu widzenia systemu kryptograficznego kluczowe są wartości o takim samym prawdopodobieństwie wystąpienia.

- Każda z liczb losowych powinna być nieprzewidywalna. Wymaganie to narzuca odporność na ustalenie kolejnych wartości na podstawie poprzednich. Generatory liczb rzeczywiście losowych z definicji bazują na zjawiskach, których stanów nie da się przewidzieć, zatem są doskonałym źródłem losowości dla systemów zapewniających cyberbezpieczeństwo.

Prymitywy kryptograficzne jako wektory inicjalizacyjne mogą używać liczb losowych. Wektory te mogą mieć postać na przykład pojedynczej liczby, czyli tak zwanej nonce (*z ang. Number used once*). Jest to liczba generowana losowo lub pseudolosowo, która następnie używana jest do uwierzytelniania użytkownika za pomocą odpowiedniego protokołu autentykacji. Może być traktowana jako tak zwane wyzwanie dla nadawcy, który następnie używając tej liczby dostarcza swój unikalny identyfikator ukrywając go przy użyciu liczby podanej przez odbiorcę. Dzięki temu odbiorca może stwierdzić, czy nadana wiadomość rzeczywiście pochodzi od danego nadawcy. W cyberbezpieczeństwie mówimy często o kluczach, wspominane są takie pojęcia jak klucz publiczny czy prywatny. Kombinacja kluczy prywatnych z mechanizmami uwierzytelniania zapewnia bezpieczną komunikację.

W celu uwierzytelniania klienta, jeżeli myślimy o architekturze klient-serwer często stosuje się identyfikatory sesji. Są to pewne losowe klucze przypisywane do danego klienta. Na tej podstawie przyznawane są prawa do poszczególnych zasobów oraz funkcji. W celu wygenerowania takiego identyfikatora haszuje się na przykład pewną wygenerowaną losowo liczbę. Źródło używane do generowania losowości identyfikatora sesji powinno posiadać wysoką entropię. Jak twierdzi OWASP Foundation, czyli Open Web Application Security Project, ID sesji w przypadku uwierzytelniania użytkownika na stronie internetowej powinno posiadać przynajmniej 64 bity entropii [7]. Jak wykazano w [7] zakładając klucz o długości 128 bitów, którego entropia wynosi 64 bity oraz 100000 poprawnych sesji, zgadując 10000 kombinacji na sekundę atakujący zgadnie poprawny identyfikator sesji najszybciej w 292 lata.

Algorytmy szyfrowania są kolejnym przykładem prymitywów kryptograficznych. Można rozróżnić algorytmy symetryczne oraz niesymetryczne. Algorytmy symetryczne cechują się tym, że klucz wykorzystywany do szyfrowania i deszyfrowania tekstu jawnego jest taki sam. Algorytmy te są wydajne oraz proste w zrozumieniu i implementacji. Algorytmy symetryczne można podzielić również na szyfry blokowe i strumieniowe.

Szyfry blokowe operują na blokach bitów, które posiadają pewną ustaloną długość. Oznacza to, że taki algorytm jako wejście może przyjąć np. określoną liczbę bitów pewnego tekstu jawnego, a na wyjściu wygenerować bity będące zaszyfrowaną wersją wiadomości. Zazwyczaj wejście i wyjście takiego szyfru posiadają taki sam rozmiar. Przykładem takiego szyfru jest między innymi DES (*z ang. Data Encryption Standard*), Triple DES, AES (*z ang. Advanced Encryption Standard*), oraz wiele innych. DES używa pewnych losowych bitów wejściowych do generacji kolejnych kluczy. W przypadku szyfrów blokowych bardzo ważna jest długość klucza. Znając klucz jesteśmy w stanie zarówno zaszyfrować jak i odszyfrować wiadomość, co jest zresztą definicją klucza symetrycznego. Data Encryption Standard jest uważany za niebezpieczny [8] z powodu krótkiego klucza wynoszącego 56 bitów.

W przypadku szyfrów strumieniowych, które są drugim podtypem algorytmów symetrycznych szyfrujemy każdy bit informacji. Najczęściej odbywa się to poprzez wykonanie operacji XOR na bicie klucza i wiadomości. Mowa jest tu o pewnym strumieniu kluczy. Strumień ten najczęściej zostanie wygenerowany na podstawie innego klucza, który powinien być losowy. Rozwiązania polegające na generacji całego strumienia na podstawie liczb rzeczywiście losowych ze względów wydajnościowych nie są często stosowane.

Algorytmy asymetryczne w przeciwieństwie do algorytmów symetrycznych bazują na zestawie kluczy. Posiadamy klucz prywatny, służący najczęściej do odszyfrowywania wiadomości oraz klucz publiczny, który często służy do szyfrowania. Jeżeli wiadomość zostanie zaszyfrowana przez jeden z kluczy, to odszyfrowanie będzie możliwe tylko przez drugi klucz z zestawu. Zapobiega to niebezpieczeństwu, jakie generują algorytmy symetryczne, polegającemu na przesyłaniu klucza.

Jednym z zagrożeń dla systemów teleinformatycznych oraz informatycznych są tak zwane ataki typu bocznokanałowego (*ang. side-channel*). Polegają one na pozyskiwaniu informacji z takich źródeł jak wydawany przez system dźwięk, czasy odpowiedzi czy szum elektromagnetyczny. Są to informacje związane z aktywnością urządzenia, na którym zaimplementowany jest atakowany system. Ataki te są zazwyczaj skierowane na wydobycie małej ilości informacji pozwalających na rozpracowanie systemu. Mogą to być na przykład klucze, hasła, czy też krytyczne podatności systemu. Obecnie znane jest kilka rozwiązań mających na celu ochronę przed atakami tego typu, a często też próbę opóźnienia skuteczności takich działań. Podejścia są różne, jednak część z nich przewiduje wprowadzenie pewnej losowości w systemie. Może to się odbywać na zasadzie wprowadzania losowości w opóźnieniach wysyłanych

odpowiedzi, co jest pomocne w przypadku obrony przez atakami korzystającymi z informacji o czasie odpowiedzi systemu. Ta metoda jednak nie zapewnia pełnej skuteczności, zazwyczaj pomaga wydłużyć atak, ponieważ atakujący musi uśrednić uzyskane wyniki [9]. Warto wspomnieć, że wartość dodana informacji wydedukowanej na podstawie opóźnień opiera się na podstawie bardzo małego czasu, często są to mikrosekundy. Biorąc pod uwagę inne opóźnienia na przykład w globalnej sieci złożonej z wielu urządzeń sieciowych, uzyskanie takiej dokładności wiąże się z pewną estymacją, co wydłuża cały proces. Losowość może być dodana do samej wiadomości, co również jest stosowane na przykład w algorytmie Rivesta Shamira Adlemana (*RSA*). Proces taki jest odwracalny, umożliwiają to proste operacje matematyczne. Dzięki temu wszystkie operacje są wykonywane tak naprawdę na pewnym losowym ciągu, który można przekształcić w pierwotną wiadomość.

3. Konstrukcje generatorów liczb losowych

3.1. Wybrane metody pozyskiwania entropii

3.1.1 Teoria informacji i entropia

Generatory liczb losowych powinny działać samodzielnie, a potrzeba generacji kolejnych sekwencji losowych bitów może nastąpić w dowolnym momencie. Sprawia to, że generatory liczb losowych muszą być w stanie uzyskać losowość z otaczającego ich świata. Stosowane implementacje wykorzystują fizyczne oraz niefizyczne źródła losowości [6]. Można rozróżnić źródła na te czysto cyfrowe oraz analogowe. Do fizycznych źródeł należą między innymi różne rodzaje szumów takie jak termalny, śrutowy, rozpad promieniotwórczy, fluktuacje energii próżni, mikrofalowe promieniowanie tła, oraz wiele innych. Z drugiej strony mamy cyfrowe źródła, do których można zaliczyć metastabilność przerzutników, jitter zegara, a nawet zależności czasowe wynikające z interakcji człowieka z komputerem [6][10].

W kontekście generatorów liczb losowych mówimy o entropii, a konkretnie o jej źródłach oraz o sposobach jej pozyskiwania. Generatory liczb losowych czerpią z entropii pożytek przekształcając panujący we wszechświecie chaos w liczby losowe. Wykorzystywane są różne źródła entropii, które posiadają swoje wady i zalety, oraz lepiej nadają się do różnych zastosowań.

Sygnały pochodzące ze źródła entropii powinny być pozyskane i przekształcone w strumień bitów. Sposób wykonania tego zadania będzie różny w zależności od źródła entropii. W przypadku sygnałów analogowych pochodzących z analogowych źródeł entropii do pozyskania z nich losowości wykorzystywane są często konwertery analogowo-cyfrowe, liczniki czy konwertery napięcia na częstotliwość. Pozyskiwanie entropii nie jest zadaniem oczywistym i wiąże się z różnymi wyzwaniami. Prostym przykładem może być tak zwane nadpróbkowanie,

to znaczy przypadek, gdy próbkowanie jakiejś wartości odbywa się więcej niż raz ze względu na zbyt dużą częstotliwość próbkowania. W przypadku cyfrowych źródeł entropii również musimy w jakiś sposób tę entropię pozyskać. Przykładem może być interakcja człowieka z komputerem, gdzie aby uzyskać wymagane dane trzeba najpierw doprowadzić do interakcji z komputerem, która musi trwać jakiś czas.

Mówiąc o entropii mamy na myśli pewną miarę losowości układu. Entropia może być definiowana jako nieprzewidywalność stanu określonego zdarzenia elementarnego w określonym momencie czasu. Możemy się spotkać z różnymi definicjami entropii w zależności od dziedziny w jakiej się poruszamy.

Teoria informacji określa entropię pewnej zmiennej losowej jako miarę informacji, lub niepewności przypadającej na konkretną wiadomość. Twórcą teorii informacji jest Claude Shannon, amerykański matematyk i inżynier. W artykule "A Mathematical Theory of Communication" [11] opublikowanym w Bell System Technical Journal w roku 1948 przedstawił on koncepcję entropii informacji. Jeżeli jakieś zdarzenie elementarne A ma prawdopodobieństwo wystąpienia równe jeden, to informacja mówiąca "zdarzenie A nastąpi" niesie zerową wartość informacji. Gdyby prawdopodobieństwo wystąpienia tego samego zdarzenia, lub innego zdarzenia było bliskie zeru, to informacja ta zyskuje wówczas znacznie większą wartość. Jest to podstawowa idea teorii informacji. Podejście to mogłoby sugerować zapis matematyczny wartości informacji przedstawiony w równaniu **3.1**, gdzie P(A) jest prawdopodobieństwem zdarzenia elementarnego A, I(A) jest miarą wartości informacji.

$$I(A) = \frac{1}{P(A)} \tag{3.1}$$

Przy takim podejściu dla P(A) równego jeden I(A) jest również warte jeden, co nie jest zgodne z ideą tej miary. Stosuje się tutaj więc logarytm jak pokazano w równaniu **3.2**.

$$I(A) = \log_2\left(\frac{1}{P(A)}\right) \tag{3.2}$$

Shannon zdefiniował entropię dyskretnej zmiennej losowej jako wartość oczekiwaną wartości treści informacji, co można wyrazić równaniem **3.3** na funkcję gęstości prawdopodobieństwa.

$$H(x) = -\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{P}(\mathrm{x_i}) \cdot \log_2\big(\mathrm{P}(\mathrm{x_i})\big) \tag{3.3}$$

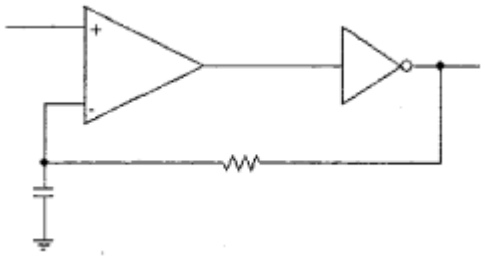
X reprezentuje pewną zmienną losową, której potencjalne wartości należą do zbioru $\{x_1, x_2, \ldots x_n\}$, a $P(x_i)$ prawdopodobieństwem wystąpienia wartości x_i [11].

3.1.2 Generatory oparte o różne rodzaje szumów

Szum termiczny jest często stosowany jako źródło entropii. Występuje on w każdym oporniku. Fakt ten został wykorzystany w różnych implementacjach generatorów liczb rzeczywiście losowych. W artykule "A Truly Random Number Generator Based on Thermal Noise " [12] autorstwa Huang Zhun oraz Chen Hongyi można przeczytać, że szum termiczny jako źródło entropii jest idealny wydajnościowo, czyli umożliwia stworzenie generatora, który będzie bardzo wydajny i równocześnie łatwy w implementacji. W artykule [12] wykorzystany jest efekt losowego falowania napięcia pomiędzy dwoma terminalami opornika. Efekt ten związany jest z rezystancją i temperaturą, pozostaje niezależny od natężenia prądu. Zmienne napięcie opornika jest również nazywane szumem Johnsona. Ten typ szumu został odkryty i po raz pierwszy zmierzony przez urodzonego w Szwecji inżyniera i fizyka Johna Bertranda Johnsona w roku 1926 w Bell Labs. Swoje odkrycia opisał dla Harrego Nyquista, a ten był w stanie te wyniki zinterpretować i opisać. W 1928 Nyquist udowodnił, że gęstość widmowa mocy danego szumu jest określona przez równanie **3.4**,

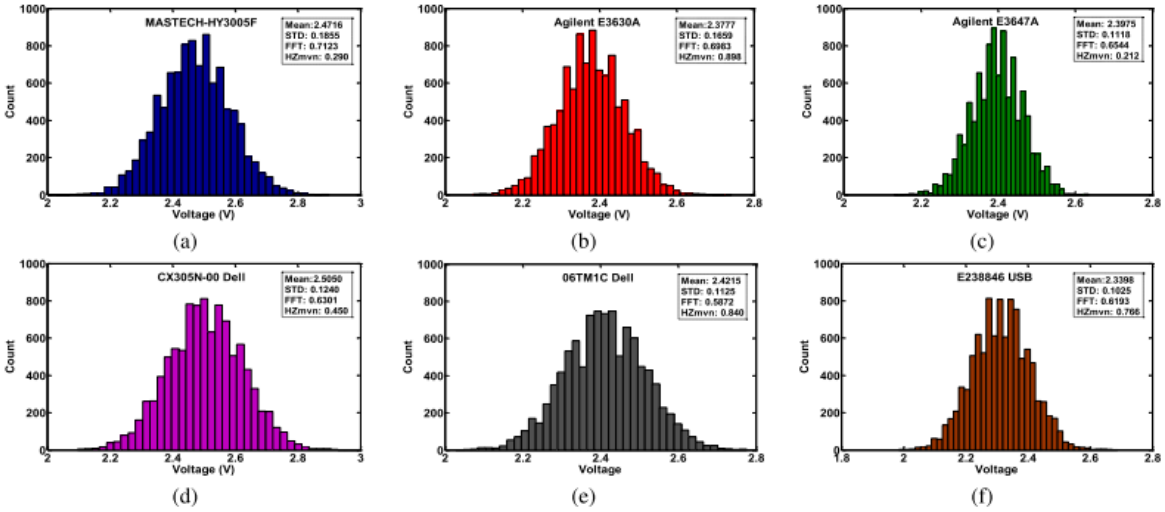
$$S(f)=4kRT \tag{3.4}$$

W równaniu **3.4** k jest stałą Boltzmana, R jest rezystancją, a T temperaturą w Kelvinach. Gęstość widmowa mocy wynika z temperatury, dlatego źródłem entropii jest właśnie szum termiczny. Przedstawiona w artykule [12] koncepcja zakłada późniejsze wzmocnienie szumu, co stawia kolejne wyzwania. Wyjście wzmacniacza zawiera zarówno szum źródła entropii, jak i szum samego wzmacniacza, co może mieć negatywny wpływ na jakość produkowanych liczb losowych. Autorzy przedstawiają w artykule propozycję rozwiązania tego problemu. Sygnał następnie trafia na komparator, który porównuje szum z sygnałem odniesienia produkując "1" lub "0" w zależności od tego, czy sygnał szumu jest większy czy mniejszy od tego sygnału. Schemat układu formującego losowy strumień bitów użyty w artykule przedstawia rysunek **3.1**

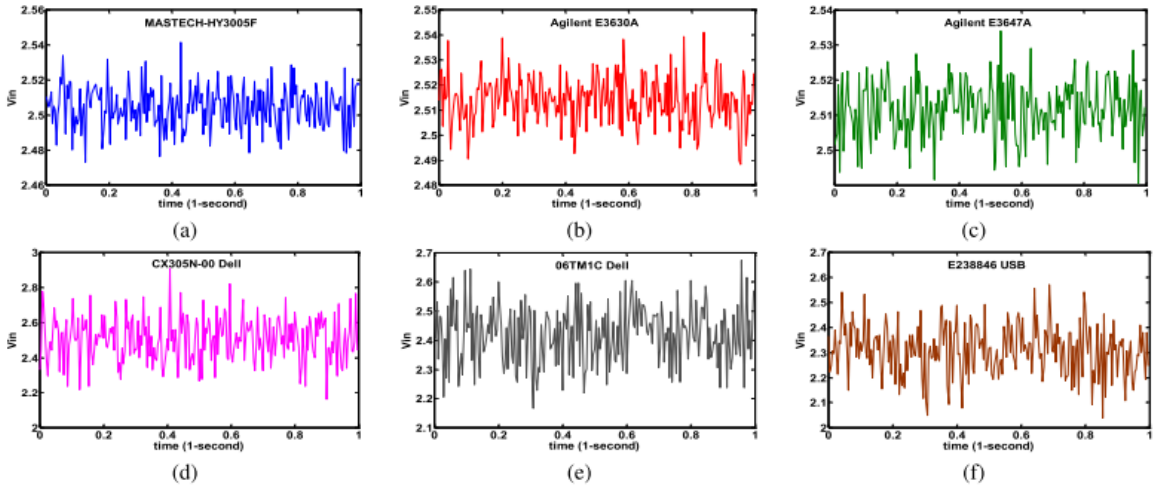


Rysunek 3.1 Schemat układu formującego losowy strumień bitów [12]

Jako źródło entropii można również wykorzystać szum zasilacza. Jak pokazali autorzy artykułu “DVFT: A Lightweight Solution for Power-Supply Noise-Based TRNG Using Dynamic Voltage Feedback Tuning System” [13] napięcie generowane przez zasilacz nie jest idealne, to znaczy nie w każdym momencie na wyjściu zasilacza uzyskujemy napięcie deklarowane przez producenta. W artykule przedstawiono histogram wartości napięcia kilku zasilaczy oraz te wartości w dziedzinie czasu.



Rysunek 3.2 Rozkład napięcia wyjściowego zasilaczy [13]



Rysunek 3.3 Wartości napięcia wyjściowego zasilaczy w dziedzinie czasu [13]

Jak pokazano na wykresach przedstawionych na rysunku **3.2** zasilacze poddane badaniu wykazały rozkład napięcia wyjściowego przypominający rozkład normalny. Wykresy na rysunku **3.3** przedstawiają wartości napięć zasilaczy w dziedzinie czasu. W zależności od zasilacza odstrojenie od idealnego napięcia różniło się. Przeprowadzona została analiza uzyskanych

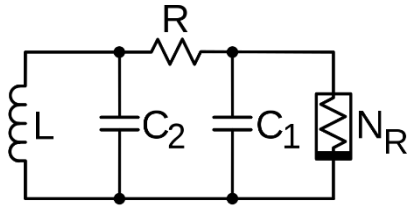
wyników. W jej wyniku autorzy doszli do wniosku, że źródła zasilania demonstrują standardowy proces Gaussowski, oraz stwierdzili, że nadają się jako źródło losowości. Obliczone wartości uzyskanych rozkładów, takie jak między innymi wartość średnia czy odchylenie standardowe są według autorów akceptowalne. W celu zamiany uzyskanych sygnałów analogowych na strumień bitów wykorzystane mogą zostać właściwości rozkładu normalnego. W tym rozkładzie połowa wartości przypada powyżej średniej, a połowa poniżej średniej, co może zostać wykorzystane jako logika zero-jedynkowa. Jeden może oznaczać wartość większą niż średnia, a zero mniejszą niż średnia.

3.1.3 Generatory bazujące na chaosie

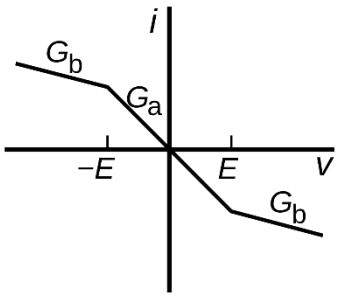
Kolejną ważną grupą generatorów liczb rzeczywiście losowych są generatory bazujące na chaosie. Generatory te jako źródło entropii używają systemów deterministycznych. Systemy chaotyczne są bardzo podatne na warunki początkowe, nawet najmniejsza zmiana może skutkować znaczącą różnicą w wyniku [14].

Można rozróżnić dwie główne struktury generatorów liczb rzeczywiście losowych opartych na chaosie. Pierwszą grupą są generatory bazujące na chaotycznych systemach czasu ciągłego. Znane są różne chaotyczne systemy czasu ciągłego takie jak system Lorena czy obwód Chua. Kolejną grupą są generatory oparte o chaotyczne systemy czasu deterministycznego. Do systemów tych możemy zaliczyć mapy logiczne (logistic mapping), mapowanie namiotowe (tent mapping), czy mapowanie Bernoulliego.

Obwód Chuy jest elektronicznym obwodem, który wykazuje zachowanie chaotyczne. Wynaleziony w roku 1983 przez Leona Ong Chua, amerykańskiego inżyniera elektronika, obwód produkuje przebieg oscylacyjny, który się nigdy nie powtarza. Obwód Chuy przedstawiono na rysunku **3.4**.



Rysunek 3.4 Obwód Chuy [15]



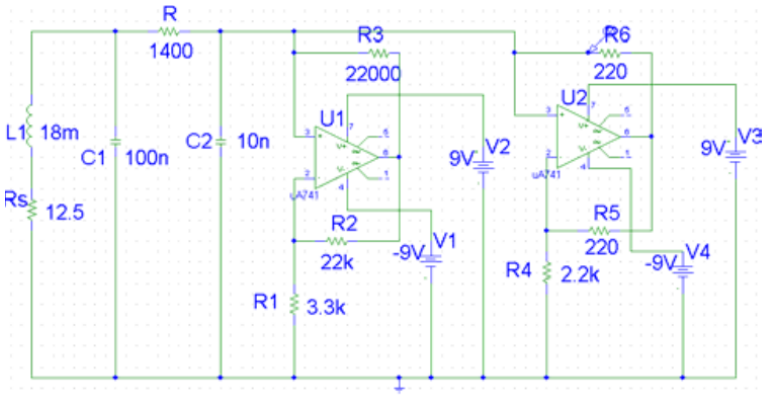
Rysunek 3.5 Charakterystyka prądowo-napięciowa diody Chuy [16]

N_R reprezentuje tutaj tak zwaną diodę Chuy, której charakterystyka prądowo-napięciowa została przedstawiona na rysunku **3.5**. Jest to tak naprawdę pewien nieliniowy rezystor.

Według autorów artykułu "A True Random Binary Sequence Generator Based On Chaotic Circuit" [14], aby można było mówić o chaotycznym zachowaniu obwodu powinien on zawierać:

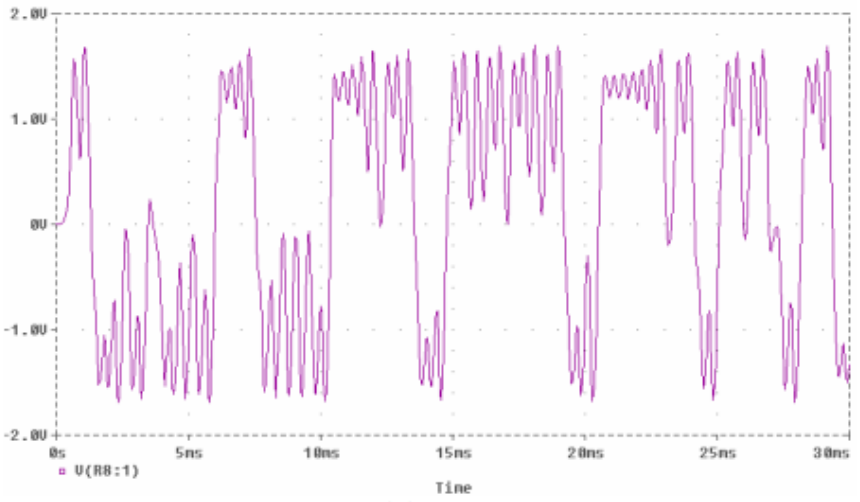
- przynajmniej jeden element nieliniowy
- przynajmniej jeden aktywny rezystor
- przynajmniej trzy elementy magazynowania energii

Obwód Chua spełnia te warunki. W artykule [14] właściwości obwodu Chuy zostały ustalone przez symulację PSpice, dla której zostały użyte parametry praktycznej implementacji obwodu, która została przedstawiona przez autorów tak jak na rysunku **3.6**.

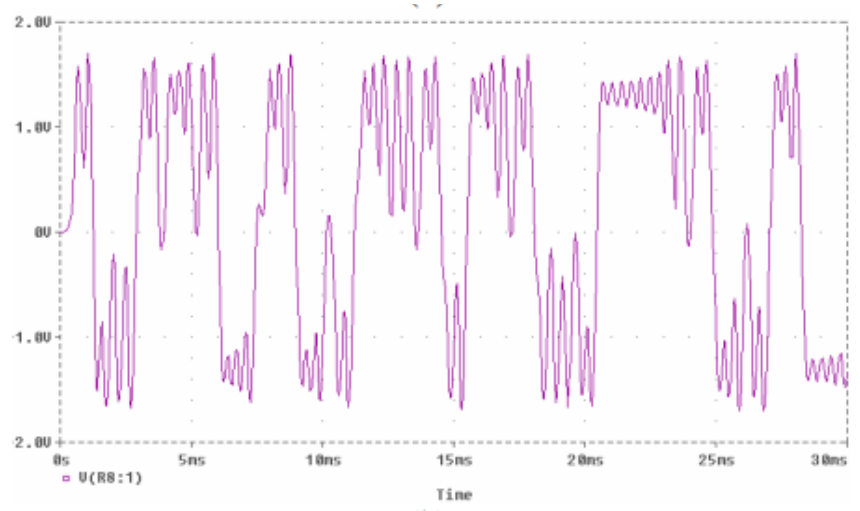


Rysunek 3.6 Implementacja schematu obwodu Chuy w PSpice [14]

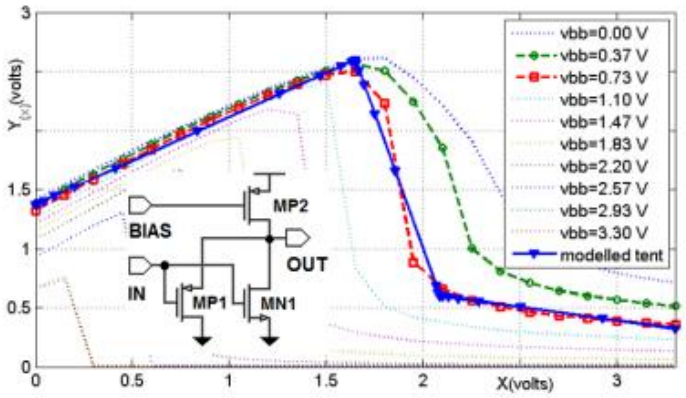
Przeprowadzono symulacje tego obwodu w dziedzinie czasu dla dwóch różnych rezystancji R. Rysunek **3.7** przedstawia wynikowe napięcie dla R=1400Ω, a rysunek **3.8** wynikowe napięcie dla R=1401Ω. Jak można zauważyć drobna zmiana rezystancji przełożyła się na zupełnie inną charakterystykę wyjściową. Zostało tu również wspomniane, że drobna zmiana temperatury może wpływać na niewielkie zmiany w rezystancji opornika, co powoduje, że nawet mała zmiana temperatury może spowodować zupełnie inną sekwencję bitów.



Rysunek 3.7 Wynikowe napięcie obwodu Chuy dla R=1400Ω [14]



Rysunek 3.8 Wynikowe napięcie obwodu Chuy dla R=1401Ω [14]



Rysunek 3.9 Nieliniowa charakterystyka mapy namiotowej wraz ze schematem [17]

Tak uzyskany sygnał można następnie przepuścić przez komparatory porównujące go z sygnałami odniesienia. W artykule zaproponowano ustanowienie dwóch komparatorów z dwoma różnymi

progami, które produkują bity, kiedy sygnał przechodzi przez te progi. W wyniku tego uzyskujemy dwa sygnały, które reprezentują chaotyczne sygnałowe przejścia przez odpowiednio ustawione progi.

Wykorzystanie chaotycznego oscylatora czasu dyskretnego w generatorze liczb losowych zostało zaproponowane w artykule “Discrete Chaos - Based Random Number Generator” [17]. Oscylator jest tutaj złożony z nieliniowej mapy, bufora i generatora zegarowego. Pierwszy elementem oscylatora jest mapa namiotowa (*z ang. Tent map*). Składa się ona z trzech tranzystorów, a jej nieliniową charakterystykę wraz ze schematem autorzy zaprezentowali na rysunku **3.9**.

3.1.4 Generatory implementowane na płytkach programowalnych

Generatory liczb rzeczywiście losowych są często implementowane z użyciem programowalnych urządzeń logicznych takich jak FPGA, czy ASIC. Urządzenia takie z zasady zawsze powinny być w dobrze zdefiniowanych stanach, co utrudnia wydobywanie z nich entropii, ponieważ do generacji liczb losowych potrzebujemy losowe zjawiska na których stany nie możemy mieć wpływu, oraz których nie możemy przewidzieć. Jako źródła entropii używane są w tym przypadku zjawiska takie jak jitter zegara, metastabilność, chaos czy sygnały analogowe takie jak szum termiczny [6].

3.1.5 Generatory oparte o jitter

Jitter zegara jest pewnym odstrojeniem zbocza sygnału zegarowego. To znaczy, zbocze zegara nie pojawia się w założonym momencie czasu, a w innym, co jest spowodowane różnymi szumami wpływającymi na działanie urządzeń elektronicznych. Jitter nie jest zjawiskiem pożądanym w urządzeniach logicznych, ponieważ wpływa on negatywnie na systemy komunikujące się z dużą częstotliwością i szybkością. W przypadku generatorów liczb rzeczywiście losowych jitter spowodowany losowymi szumami jest pożądany, ponieważ możemy z niego wydobyć entropię. Zanim wygenerujemy losową sekwencję bitową z jittera powinniśmy znać jego właściwości statystyczne. Większość generatorów używa niezależnych szumów *(ang. independent noise*) jako źródło losowości.

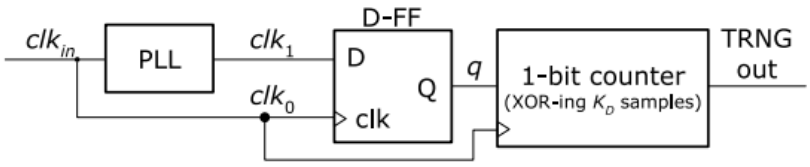
Generowanie liczb losowych z użyciem jittera jako źródła entropii wymaga cyfryzacji pozyskanej entropii. Najczęściej jitter zegara jest próbkowany, co może się odbyć przy użyciu przerzutnika, na którego wejście podłączany jest sygnał zawierający jitter, a na wejście zegarowe dostarczany jest pewien zegarowy sygnał referencyjny. Do produkcji losowych bitów potrzebujemy takiego sygnału zegarowego, który będzie w stanie próbkować jitter z dużą dokładnością. Produkowany jitter może być bardzo mały, co stawia duże wymagania dla sygnału

zegara, który spełnia rolę sygnału odniesienia. Jitter jest nie do uniknięcia, dlatego sygnał odniesienia również zawiera jitter, powoduje to dodatkowe utrudnienia.

3.1.6 Generatory wykorzystujące pętlę synchronizacji fazy

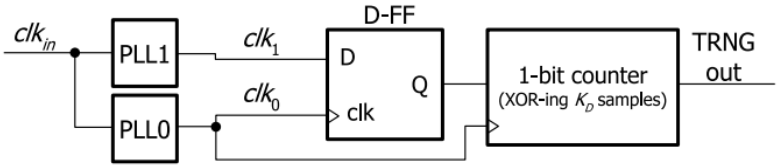
Przedstawiona w artykule „Modern random number generator design – Case study on a secured PLL-based TRNG" [18] metoda zakłada użycie pętli synchronizacji fazy jako źródła entropii i koherentnej metody próbkowania do generacji losowych bitów. Przy próbkowaniu koherentnym, jak zaznaczają autorzy zarówno sygnał próbkujący jak i próbkowany są okresowe ze znanym stosunkiem częstotliwości. W implementacji zastosowano binarne sygnały zegarowe. Pętla synchronizacji fazy gwarantuje nam, że zachodzi związek pomiędzy częstotliwością sygnału próbkowanego i próbkującego.

Koncepcja generatora liczb losowych opartego na pętli synchronizacji fazy została przez autorów [18] przedstawiona jak na rysunku **3.10**.



Rysunek 3.10 Schemat generatora liczb losowych opartego na pętli synchronizacji fazy [18]

Przetestowano tu różne wariancje konfiguracji architektur generatorów opartych o pętle synchronizacji fazy. Zostało tu również wspomniane, że możliwe jest użycie dwóch pętli synchronizacji fazy. Równoległa konfiguracja takiego generatora jest przedstawiona na rysunku **3.11**

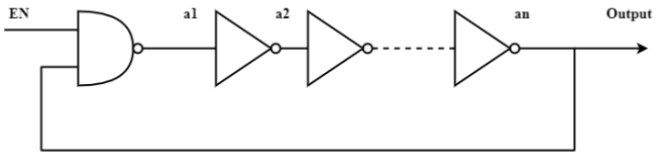


Rysunek 3.11 Schemat generatora liczb losowych opartego na dwóch pętlach synchronizacji fazy [18]

Rozwiązanie to stosuje trochę inne podejście konwersji jitteru na losowe bity. Mamy tu do czynienia z akumulacją jittera do momentu, gdy jego wielkość jest większa niż okres próbkowanego sygnału. Zapewnia nam to pętla synchronizacji fazy, która zawiera dzielniki częstotliwości. Jeżeli odpowiednio podzielimy naszą częstotliwość clk_{in}, to możemy clk_1 próbkować z mniejszą częstotliwością, co pozwoli na zakumulowanie się jittera fazy.

3.1.7 Generatory bazujące na oscylatorach pierścieniowych

Inne podejście zakłada użycia wielu oscylatorów pierścieniowych niezależnych od siebie. Przykład oscylatora pierścieniowego przedstawiono na rysunku **3.12**. Propozycja takiego podejścia została przedstawiona w artykule „A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks” [19]. Jak zaznaczyli autorzy, prosty cyfrowy oscylator może zostać zbudowany poprzez połączenie nieparzystej liczby inwerterów w pierścień. Sygnał produkowany przez taki oscylator w przypadku idealnym powinien mieć charakter przebiegu prostokątnego, a jego okres powinien być zdeterminowany przez liczbę inwerterów. Wyjściowy sygnał nie jest idealny, a jego okres jest różny w różnych momentach czasu. Powstaje nam więc jitter, który może zostać wykorzystany jako źródło entropii. W artykule zaznaczono, że praktycznym podejściem do tego zagadnienia jest próbkowanie wyjścia jednego oscylatora wyjściem drugiego oscylatora. Powoduje to jednak pewne problemy, ponieważ dokładne dopasowanie okresu dwóch oscylatorów nie jest łatwe. Z powodu niedoskonałości oba sygnały mogą się względem siebie przesuwać.



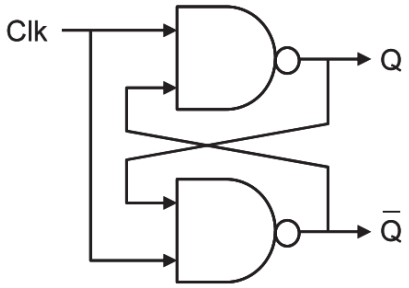
Rysunek 3.12 Przykład oscylatora pierścieniowego [20]

Użycie wielu wolno działających oscylatorów pierścieniowych zostało opisane w artykule „Chaotic Ring Oscillator Based True Random Number Generator Implementations in FPGA” [20]. Autorzy zakładają zatrzaskiwanie wyjścia każdego z oscylatorów przez przerzutnik D w takt pewnego sygnału zegarowego. Wyjścia wszystkich przerzutników są następnie przepuszczane przez drzewo XOR, którego wynikiem jest pojedynczy bit. Bit ten jest następnie zatrzaskiwany przez przerzutnik typu D, który taktowany jest tym samym sygnałem zegarowym co przerzutniki D na wyjściu każdego z oscylatorów.

3.1.8 Generatory bazujące na metastabilności

O metastabilności systemu możemy powiedzieć, jeżeli jest on w stanie przez nieokreślony okres czasu pozostać w stanie, w którym teoretycznie nie powinien się znaleźć. Nawet najmniejsze odchylenia od położenia równowagi mogą spowodować, że nastąpi przejście do stanu stabilnego, dlatego aby pozostać w stanie metastabilnym potrzebna jest pełna równowaga. Jako przykład często podaje się rzut monetą, dla którego wynikiem powinno być wylądowanie monety orłem lub

reszką do góry. Istnieje jednak bardzo mała szansa, że moneta wyląduje na swoim grzbiecie. Do zachowania takiego stanu potrzebna będzie duża równowaga, a w przypadku jej zachwiania moneta wróci do stanu stabilnego, czyli wyląduje na jednej z dwóch swoich stron. Przykład użycia metastabilności do stworzenia generatora liczb rzeczywiście losowych zaproponowano w artykule „FPGA Implementation of Metastability-Based True Random Number-Generator" [21]. Przedstawiono tu między innymi ogólną koncepcję działania generatora opartego na metastabilności przedstawiając ekstrakcję entropii bazując na przerzutniku typu RS. Przerzutnik ten został przedstawiony na rysunku **3.13**. Dla sygnału CLK równego zero na wyjściu mamy stan stabilny, to znaczy (Q,Q') = (1,1). W przypadku, gdy sygnał CLK będzie równy jeden mamy stan nieustalony. Przerzutnik wchodzi w stan metastabilny, z którego ostatecznie przejdzie do jednego ze stanów stabilnych, gdy (Q,Q') = (1,0) lub (Q,Q') = (0,1). Przejście następuje w wyniku odchylenia od stanu równowagi, odchylenie to może być spowodowane na przykład, jak podają autorzy, przez szum termiczny. Czas przejścia podlega rozkładowi prawdopodobieństwa, dzięki temu wyjście jest losowe.



Rysunek 3.13 Przerzutnik typu RS [21]

3.1.9 Generatory oparte o interakcję człowiek-komputer

Kolejną grupą generatorów są te oparte na interakcji użytkownika z komputerem. Losowość można tu pozyskać z różnych źródeł takich jak ruchy myszką [10], sygnał zbierany przez mikrofon, obraz z kamery internetowej, czy nawet opóźnienia związane z interakcją z klawiaturą.

Generator oparty o zegar systemowy i pewne źródło wizji został zaproponowany i opisany w artykule „Non-Recurring Improved Random Number Generator- a new step to improve cryptographic algorithms" [22]. W zaproponowanym podejściu najpierw pozyskujemy ziarno z zegara systemowego, a następnie na podstawie odczytanej wartości wybieramy pewien piksel ze zdjęcia wykonanego przez kamerę systemową. Dzięki temu wygenerowane ziarno jest oparte o dwa losowe źródła. Oczywiście stan samego zegara w idealnym przypadku jesteśmy w stanie

przewidzieć, jednak zegary nie odmierzają czasu idealnie i zachodzą w nich pewne losowe zjawiska. W celu zapewnienia losowości nie wystarczy skupienie się na poziomie sekund, może to zapewnić zbyt małą losowość, ponieważ w tym przedziale czasu większość zegarów jest dokładna. Autorzy nie określają czym dla nich jest wartość piksela, jednak może to być jeden z kanałów RGB lub ich odpowiednia kombinacja.

3.2. Postprocessing

3.2.1 Cel postprocessingu

Celem postprocessingu jest zapewnienie losowości przez poprawienie właściwości statystycznych, których źródłem niedoskonałości może być samo źródło entropii oraz proces jej pozyskiwania. Operacja taka powinna zapewnić na wyjściu generatora nieskorelowane, równomiernie rozłożone bity. Postprocessing może polepszyć ilość entropii przypadającej na jeden bit równocześnie zmniejszając szybkość generowanych bitów.

3.2.2 Exclusive OR

Jak pisze Robert B Davies w „Exclusive OR (XOR) and hardware random number generators" [23] operacja XOR jest często używana w celu zmniejszenia skłonności (*z ang. bias*) bitu ku jednemu ze stanów. Ta skłonność bitu jest określona jako odchylenie średniej wartość generowanych bitów od wartości $\frac{1}{2}$. Autor wspomina, że sekwencja losowych bitów $X_1, X_2, X_3 \ldots$ jest statystycznie niezależna, jeżeli nie jesteśmy w stanie przewidzieć wartości dowolnie wybranego bitu X_i na podstawie żadnego innego dowolnie wybranego bitu, ani żaden inny bit nie dostarcza nam żadnych użytecznych informacji odnośnie bitu X_i. Korelacja bitów musi być równa zero. Jak napisano w [23], jeżeli X i Y są niezależnymi losowymi bitami, których wartość oczekiwana X jest równa μ, a wartość oczekiwana Y jest równa ν, to dla wartości oczekiwanej operacji XOR tych bitów następuje równość opisana równaniem **3.5**. Autor zaznacza, że jeżeli μ i ν są bliskie $\frac{1}{2}$, to wartość oczekiwana $X \otimes Y$ jest również bardzo bliska $\frac{1}{2}$. Dla przykładu podaje przypadek, gdy $\mu = \nu = 0.6$, wówczas $E(X \otimes Y) = 0.48$, daje nam to wartość znacznie bliższą do założonej, czyli do 0.5. Operacja XOR jest bardzo prosta, do jej przeprowadzenia wymagana jest jedna bramka XOR. Tablica prawdy operacji XOR dla dwóch wartości wejściowych została przedstawiona w tabeli **3.1**.

$$\mathrm{E}(\mathrm{X} \otimes \mathrm{Y}) = \mu + \nu - 2\mu\nu = \frac{1}{2} - 2(\mu - \frac{1}{2})(\nu - \frac{1}{2})$$

Równanie 3.5 [23]

Tabela 3.1. Tablica prawdy dla bramki logicznej XOR

Input		Output
X	Y	X⊗Y
0	0	0
0	1	1
1	0	1
1	1	0

3.2.3 Korekta Von Neumanna

Często stosowaną metodą postprocessingu jest korekcja Von Neumanna. Została ona zaproponowana w [24]. Praktyczne użycie tej metody można znaleźć w artykule „Chaotic Ring Oscillator Based True Random Number Generator Implementations in FPGA" [20]. Zasada działania jest prosta, wejście jest odrzucane, jeżeli dwa kolejne bity są takie same, to znaczy, jeżeli występuje „11" lub „00". W przypadku wejścia równego „10' wyjście jest równe „1", a dla wejścia „01" wyjście jest równe „0". W [24] została zastosowana analogia do rzutu monetą. Jak pisze autor jesteśmy w stanie uzyskać szansę 50-50 nawet w rzucie źle wyważoną monetą, jeżeli rzucimy nią dwa razy. W przypadku dwóch takich samych wyników, to znaczy orzeł-orzeł lub reszka-reszka odrzucamy wynik, natomiast w przypadku orzeł-reszka lub reszka-orzeł możemy zaakceptować wynik jako pojedynczą wartość, czyli jako orzeł lub reszka. Jak zaznaczono, mimo że znacznie poprawiliśmy nasz wynik i w przybliżeniu uzyskaliśmy równe szanse na uzyskanie obu wartości, w wyniku tej operacji jesteśmy w stanie uzyskać co najwyżej 25% wydajności zwykłego rzucania monetą, to znaczy zamiast 8 wyników będziemy mieć maksymalnie 2.

3.2.4 Funkcje haszujące

Hasz kryptograficzny jest deterministycznym algorytmem, który na wejście przyjmuje pewien blok danych, a na wyjściu zwraca łańcuch o pewnej określonej wielkości. Przykładem funkcji haszujących może być rodzina funkcji skrótu taka jak SHA. Dla przykładu SHA-0 i SHA-1 na podstawie 2^{64} bitów są w stanie wygenerować 160 bitów. Jak zaznaczono w [25], jeżeli wejściowe dane funkcji haszującej posiadają wysoką entropię, to dane wyjściowe będą miały rozkład bliski równomiernemu. Używanie funkcji skrótu jest powszechnie stosowaną metoda postprocessingu.

3.2.5 Funkcja liniowa

W artykule [26] została przedstawiona metoda, która jak zostało napisane, dostarcza dobrą kompresję liniową dla losowej generacji liczb w oparciu o dobre kody korekcji błędów. Przykład takiej funkcji autorzy artykułu [26] przedstawili jak na równaniu **3.6**.

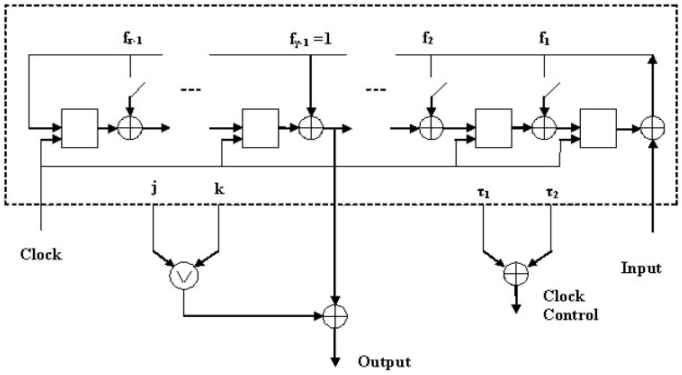
$$L(X, Y) = X \oplus (X \ll 1) \oplus (X \ll 2) \oplus (X \ll 4) \oplus Y$$

Równanie 3.6 [26]

Funkcja jako wejście przyjmuje 16 niezależnych losowych bitów, a w wyniku jej działania uzyskujemy 8 bitów. Jak zostało to wspomniane w artykule, jeżeli każdy bit sekwencji wejściowej posiadał skłonność (*z ang. bias*) ku jednemu ze stanów równą e, ta skłonność każdego skompresowanego bitu będzie wynosić $2^4 * e^5$, co w przypadku, gdy zakłamanie bitu wynosiło 0.05 daje nam $2^4 * 0.05^5 = 0.000005$.

3.2.6 Rejestr przesuwny

Jako metodę postprocessingu można również zastosować korekcję opartą o rejestr przesuwny (*ang. linear feedback shift register, LFSR*). W przypadku LFSR taktowanych regularnym sygnałem taktowania jesteśmy w stanie przewidzieć następną sekwencję. Jak zaznaczono w [27] możemy uniknąć liniowej przewidywalności, jeżeli LFSR będzie taktowany nieregularnie, czyli gdy niektóre bity wyjściowe są pomijane w takt pewnego sygnału zegarowego. W artykule przedstawiono podstawową strukturę LFSR w konfiguracji Galois używaną do postprocessingu przedstawioną na rysunku **3.14**.



Rysunek 3.14 Podstawowa struktura LFSR w konfiguracji Galois [27]

Przedstawiony rejestr posiada pewną liczbę przerzutników połączonych szeregowo. Każdy przerzutnik taktowany jest synchronicznie przez jeden zegar. Wyjście każdego przerzutnika używane jest jako wejście kolejnego po przepuszczeniu przez bramkę XOR wraz z sygnałem pętli zwrotnej. Sygnał pętli zwrotnej jest wyjściem ostatniego przerzutnika przepuszczonym przez bramkę XOR wraz z wejściem oznaczonym na rysunku jako Input. Jak wspominają autorzy, aby zapewnić dobre właściwości statystyczne i pełny okres rejestru, to znaczy, żeby liczba jego stanów była równa 2^r-1, gdzie r jest liczbą przerzutników, to wielomian opisujący nasz rejestr powinien

być pierwotny. W [27] przeprowadzone testy DIE-HARD wykazały, że dla opisywanego przez autorów generatora r = 64 było wystarczające do ich zdania.

3.2.7 Inne podejście do postprocessingu

Autorzy artykułu [22] stosują nietypową metodę postprocessingu zakładającą generację kolejnych wartości na podstawie poprzednich oraz ziarna. Metoda ta posiada również zabezpieczenie przed powtarzalnością zakładające generację nowego ziarna w przypadku wykrycia powtarzającej się sekwencji. Autorzy zaznaczają, że proces ten nie tylko polepsza właściwości statystyczne zapewniając bardziej równomierny rozkład wartości, ale też polepsza wydajność generatora. Mamy tu do czynienia z bardzo dobrym źródłem szumu jakim jest zegar systemowy. Drugie źródło jakie jest tutaj wykorzystywane to wartość losowo wybranego punktu obrazu. Mimo bardzo dobrych właściwości statystycznych entropii samego zegara kolejne generowane serie bazują na wartości pikseli oraz poprzednich wartościach i nowo generowanych liczbach pierwszych. Generowane wartości są sprawdzane, aby upewnić się, że wartości występują równomiernie. Spowodowane to jest faktem, że generowane na tym etapie wartości są pseudolosowe. Wartość aktualnego stanu zależy od stanu poprzedniego i pewnej wartości piksela ze zdjęcia, których liczba jest ograniczona. Może się tak zdążyć, że wartości zaczną się powtarzać. Dobrą analogią może być tutaj rejestr przesuwny z liniowym sprzężeniem zwrotnym, który wykazuje pozorną losowość mając tak naprawdę ograniczoną liczbę stanów, gdzie na podstawie znajomości stanu aktualnego jesteśmy w stanie ustalić stan następny.

3.3 Analiza porównawcza

Porównując różne generatory liczb rzeczywiście losowych można brać pod uwagę kilka czynników. Oczywiste jest tutaj uwzględnienie właściwości statystycznych generowanych bitów losowych. Ocena właściwości statystycznych odbywa się poprzez przeprowadzenie odpowiednich testów statystycznych. W artykułach zajmujących się generatorami liczb rzeczywiście losowych najczęściej można trafić na testy NIST (National Institute of Standards and Technology) lub DIE-HARD.

Testy DIE-HARD są zestawem testów statystycznych, które mają na celu zmierzenie jakości generowanych liczb losowych. Ich autorem jest George Marsaglia, amerykański matematyk i informatyk. Do zestawu wchodzą między innymi takie testy jak „Monkey test", „Parking lot test", czy „Birthday test". Dla większości testów wynikiem jest pewna wartość pvalue, która dla niezależnych liczb losowych powinna mieć rozkład równomierny.

W publikacji NIST „A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" [28] można przeczytać, że liczba możliwych testów

statystycznych jest nieskończona. Każdy z testów testuje występowanie pewnego wzoru, który gdy zostanie wykryty będzie sugerował, że testowana sekwencja nie jest losowa. Jak zaznaczają autorzy testy powinny być tak tworzone, aby stawiały pewną hipotezę. Następnie nasze bity losowe są testowane, a jako wynik testu powinniśmy dostać potwierdzenie lub zaprzeczenie hipotezy.

W idealnym przypadku źródło entropii dostarcza nam bity losowe o jakości pozwalającej na zdanie testów statystycznych bez potrzeby przeprowadzania postprocessingu. Przykład wyników testu NIST przeprowadzony dla generatora opartego na wolno działających oscylatorach pierścieniowych zaprezentowany w artykule [20] przedstawiono w tabeli **3.2**. Jak można zauważyć w przypadku oscylatorów pierścieniowych z 24 inwerterami generator był w stanie zdać testy NIST nawet bez zastosowania postprocessingu.

Pomimo deklarowanych w artykułach właściwości statystycznych różnych generatorów rzeczywiste wyniki uzyskane po zaimplementowaniu takiego generatora mogą się różnić. Dobrym przykładem może być generator zaproponowany w [13]. Jak przedstawili to autorzy, entropia źródła, którym w tym przypadku jest zasilacz, może się różnić w zależności od zastosowanego zasilacza. Nawet w przypadku zastosowania zasilaczy o teoretycznie takim samym napięciu wyjściowym faktyczny rozkład napięć będzie różny.

Tabela 3.2. Wyniki testów NIST dla generatora opartego o oscylatory pierścieniowe o *n* inwerterach [20]

NIST Test	Obtained p-values for different n				p-values after PPU for different n			
	4	16	24	32	4	16	24	32
Approximate Entropy	0.401391	.490767	0.399408	0.453072	0.566372	0.647145	0.255131	0.466209
Block Frequency	0.896434	.810602	0.970762	0.246160	0.464026	0.236336	0.568513	0.995166
Cumulative Sums(Forward)	0.700777	.754256	0.207048	0.533949	0.482311	0.338189	0.612840	0.331430
Cumulative Sums(Reverse)	0.665229	.608321	0.087391	0.494314	0.358432	0.321779	0.854630	0.532261
FFT	0.024559	.0495553	0.581909	0.189433	0.538664	0.720427	0.225775	0.659591
Frequency	0.969688	.830547	0.163922	0.295561	0.435391	0.188174	0.791780	0.322174
Linear Complexity	0.077109	.225641	0.666134	0.097590	0.262372	0.368617	0.492545	0.487640
Longest run	0.195949	.203929	0.745495	0.751795	0.377307	0.665577	0.212544	0.336088
Non Overlapping Template	0.998080	.950600	0.973253	0.980711	0.979166	0.986618	0.972078	0.982214
Overlapping Template	Failure	.955265	0.341905	0.476116	0.088771	0.560200	0.514623	0.894241
Random Excursions	Failure	Failure	0.922159	Failure	0.916284	Failure	0.984183	0.900329
Random Excursions Variant	Failure	Failure	0.871639	Failure	0.961817	Failure	0.957319	1
Rank	0.313460	.163319	0.775720	0.677041	0.531234	0.241791	0.709478	0.553668
Runs	0.113193	.267019	0.527427	0.243172	0.365673	0.702262	0.189497	0.455051
Serial	0.725812	.365497	0.658976	0.602901	0.971023	0.500264	0.224153	0.703402
Universal	0.466080	.265656	0.740585	0.219211	0.813215	0.812727	0.887714	0.545650

Jednym z kryteriów wyboru sposobu implementacji generatora liczb rzeczywiście losowych może być jego przeznaczanie, a dokładnie miejsce implementacji. Składa się to na kolejne warunki stawiane danemu generatorowi. Może to być między innymi ilość pobieranej energii, zajmowane miejsce, wymagane zasoby, produkowana energia cieplna, czy

koszty. Autorzy [27] dopuszczają się krytyki generatorów opartych o oscylatory pierścieniowe twierdząc między innymi, że są powolne. Dalej autorzy prowadzą rozważania na temat generatorów opartych o metastabilność jako przykład podając [29]. Twierdzą, że generatory oparte o metastabilność są również powolne, ponieważ zjawiska metastabilne są rzadkie. Pomimo krytyki różnych rozwiązań generatorów opartych o oscylatory pierścieniowe można znaleźć wiele artykułów, które je opisują. Implementacja takiego generatora na FPGA nie jest trudna i może odbyć się bez problemu na różnym sprzęcie dostarczonym od różnych producentów. W [12] możemy przeczytać, że zaproponowane rozwiązanie może być zastosowane między innymi w chipach kart inteligentnych jako rdzeń, co może być zaletą generatora, jeżeli jego zastosowanie pokrywa się z naszymi potrzebami.

Implementując generator trzeba również wziąć pod uwagę jego wydajność. W zależności od źródła entropii, sprzętu na którym implementujemy, a także od metody postprocessingu możemy uzyskać różną przepustowość bitową. Przepustowość ta podawana jest w bitach na sekundę. Jako przykład mogą posłużyć wyniki przedstawione w artykule [18], które zostały również przedstawione w tabeli **3.3**. Autorzy przepustowość zaprezentowali w kb/s i została ona przedstawiona dla różnych parametrów generatora. Największą przepustowość uzyskano dla K_M=244, K_D=9, clk_0=16, clk_1=433.8 i wyniosła ona 1778 kb/s.

Tabela 3.3. Parametry PLL-TRNG dla sześciu różnych przetestowanych konfiguracji [18]

K_M	K_D	Clk0 [MHz]	Clk1 [MHz]	R [kb/s]	Δ [ps]
1015	279	99.2	360.1	355	9.9
185	217	99.2	84.6	457	54.5
185	93	99.2	197.3	1067	54.5
595	93	99.2	634.7	1067	16.9
244	9	16	433.8	1778	256.1
214	205	248	258.9	1210	18.8

Dobór odpowiedniego sposobu postprocessingu jest kluczowym elementem projektowania generatora. Jak zostało to już wspomniane, w idealnym przypadku postprocessing nie jest potrzebny. Efektem postprocessingu będzie zmniejszenie liczby bitów na wyjściu, co zmniejsza przepustowość generatora. Skrajnym przykładem mogą być funkcje skrótu SHA-0 lub SHA-1, które w wyniku swojego działania z maksymalnie 2^{64} bitów wytwarzają 160-bitowy skrót. Tak radykalne zmniejszenie przepustowości bitów nie jest powszechnie praktykowane, chociaż same

funkcje skrótu są używane. Jak stwierdzono w [25], jeżeli dane wejściowe funkcji skrótu posiadają dużą entropię, to wyjście funkcji będzie miało rozkład bliski równomiernemu. W przypadku tej metody postprocessingu, jak podają autorzy [26], trudno jest określić jak bardzo statystyka naszych danych wyjściowych polepszyła się w porównaniu do statystyki danych wejściowych. Kolejnym wspomnianym minusem tego rodzaju operacji są ograniczenia sprzętowe, implementacja tego rodzaju przetwarzania końcowego może być mało efektywna z uwagi na fakt, iż funkcje skrótu są funkcjami nieliniowymi. W przypadku metody postprocessingu Von Neumanna w jej standardowej formie na wyjściu uzyskujemy maksymalnie 25% ilości informacji z wejścia. Dla przetwarzania końcowego z wykorzystaniem pojedynczej bramki XOR przepustowość wynosi 50% przepustowości wejściowej, co jest logiczne z uwagi na fakt, że z dwóch wejść otrzymujemy jedno wyjście. Różne źródła, między innymi [27] podają, że zaletą metody Von Neumanna względem bramki XOR jest wyjściowe odstrojenie od wartości średniej bitów $\frac{1}{2}$.

4. Implementacja generatora

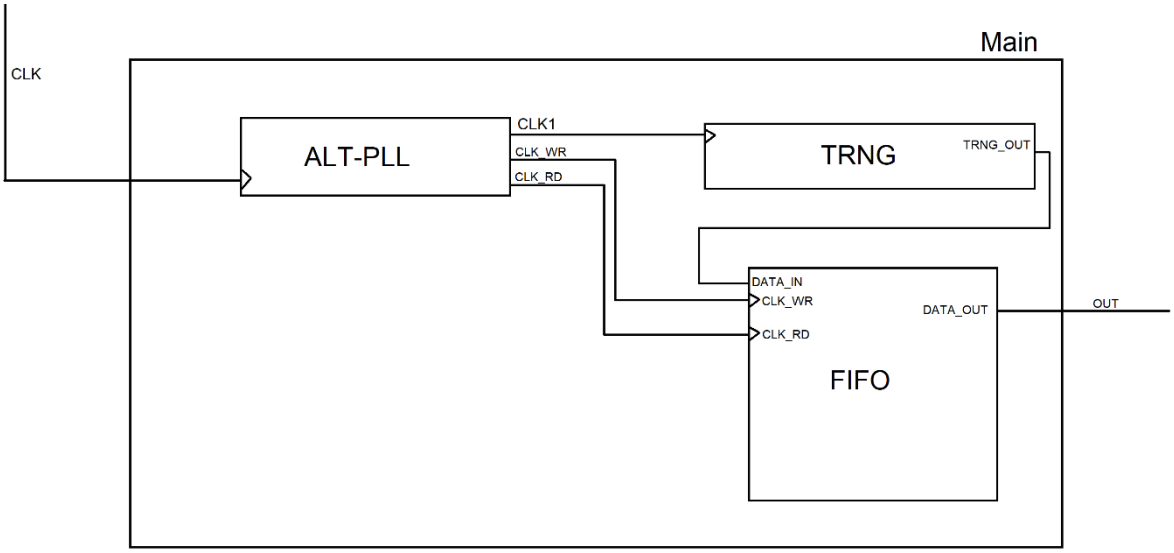
4.1. Platforma sprzętowa

Generator liczb rzeczywiście losowych został zaimplementowany na bezpośrednio programowalnej macierzy bramek (*ang. field-programmable gate array, FPGA*). Wykorzystano płytkę Terasic DE10-Lite MAX 10 10M50DAF484C7G. Do implementacji został użyty program Quartus Prime w wersji 20.1.1 Build 720 11/11/2020 SJ Lite Edition. Program napisano w języku Verilog.

Bezpośrednio programowalne macierze bramek zapisywane w skrócie „FPGA”, jak wspomniano na stronie xilinx.com [30] są to urządzenie półprzewodnikowe budowane na podstawie macierzy konfigurowalnych bloków logicznych (*ang. configurable logic block, CLB*), pomiędzy którymi połączenia są programowalne. FPGA może zostać przeprogramowane w celu zmiany funkcjonalności lub sposobu aplikacji. Bezpośrednio programowalne macierze bramek posiadają wiele zastosowań zarówno w bezpieczeństwie, jak i innych dziedzinach takich jak medycyna, komunikacja bezprzewodowa, czy elektronika konsumencka.

Początkowa implementacja zakładała podzielenie projektu na moduły takie jak moduł główny, moduł generatora liczb losowych, moduł bufora pamięci będący kolejką typu FIFO oraz moduł pętli synchronizacji fazy. Wyjście generatora powinno być odpowiednio próbkowane i przechowywane w kolejce FIFO, która następnie po zapełnieniu na żądanie udostępnia przechowywane bity. Zastosowano tu dostępną w programie Quartus Prime implementację kolejki

FIFO, którą można znaleźć w zakładce „IP Catalog”. Parametry kolejki są przestrajalne. Na potrzeby projektu zastosowano 1-bitowe wejście, oraz 1-bitowe wyjście. Ilość słów, która jest pojemnością kolejki została ustawiona na maksymalną dostępną, ich ilość wyniosła 131072. Kolejka FIFO potrzebowała dwa osobne zegary, jeden służący do kolekcji danych, drugi służący do jej opróżniania. Wynika to z faktu, że w kolejnym etapie dane muszą zostać przesłane przez kartę pomiarową o ograniczonej maksymalnej częstotliwości próbkowania, który posiada znacznie mniejszą przepustowość, niż większość implementacji generatorów liczb losowych. Platforma, na której odbywała się implementacja posiada trzy źródła zegarowe [31]. Pierwsze źródło zegarowe o częstotliwości 10MHz dla przetwornika analogowo-cyfrowego (z ang. Analog-To-Digital Converter), zapisywany w skrócie ADC, drugie i trzecie źródło to zegary o częstotliwości 50MHz. Koncepcyjny schemat takiego układu znajduje się na rysunku **4.1**. Zostały tu pominięte sygnały informujące o zapełnieniu kolejki oraz o pustym buforze.

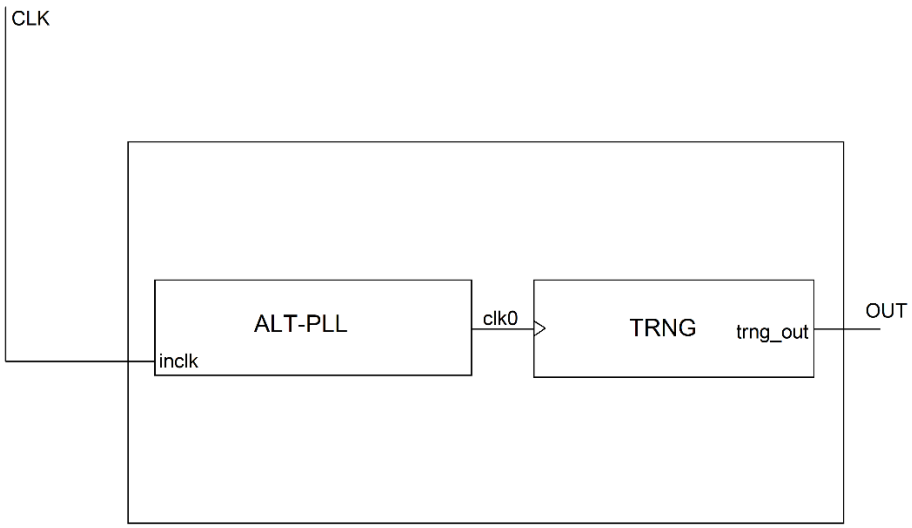


Rysunek 4.1 Schemat układu generatora z kolejką FIFO

W celu weryfikacji poprawności tak skonstruowanej konfiguracji w miejsce generatora liczb losowych został zaimplementowany rejestr przesuwny *(ang. linear feedback shift register, LFSR)* oparty na 8 przerzutnikach. Początkowo ustawienia FIFO zmieniono tak, aby na wejściu i wyjściu znajdowało się 8 bitów. Wyjście każdego z przerzutników było próbkowane podczas każdego zbocza narastającego zegara i zapisywane do kolejki. Po zapełnieniu kolejki dane były zczytywane, a bity wyświetlane na dostępnych na platformie diodach LED. Częstotliwość pobierania danych została znacznie zmniejszona, co wymagało przepuszczenia sygnału taktowania przez dwa moduły dokonujące syntezy częstotliwości, ponieważ pojedynczy moduł posiada ograniczenia co do maksymalnego dzielnika częstotliwości. Ograniczenie częstotliwości

było wymagane, aby możliwa była wzrokowa ocena poprawności wyświetlanych sekwencji na diodach LED. Na każdej diodzie LED był wyświetlany inny bit odpowiadający wyjściu jednego z przerzutników. Po stwierdzeniu poprawności danej implementacji konfiguracja kolejki została zmieniona tak, aby na wejściu i wyjściu były dane 1-bitowe. Próbkowane było wyjście tylko jednego z przerzutników symulując bit losowy. Diody LED zostały podzielone na dwa segmenty po 4, jeden segment odpowiadał za zliczanie ilości zer, a drugi za zliczanie ilości jedynek. Miało to na celu późniejsze zweryfikowanie występowania entropii w generatorze liczb losowych. W tym przypadku nie było potrzebne aż tak radykalne zmniejszenie częstotliwości opróżniania kolejki, dlatego zastosowano tylko jedną pętlę synchronizacji fazy. Tak przygotowana konfiguracja umożliwiała implementację generatora liczb losowych w miejsce LFSR.

Mając na uwadze ograniczenia sprzętowe konieczna była modyfikacja schematu. Było to związane z implementacją kolejki FIFO, której jednym z sygnałów wejściowych jest flaga informująca kolejkę o tym, że powinna wysłać dane na wyjście w momencie kolejnego zbocza narastającego zegara. Rozpoczęcie pobierania danych z kolejki powinno nastąpić, gdy platforma akwizycji rozpocznie pobieranie danych, a pamięć kolejki zostanie zapełniona. Pierwotne rozwiązanie zakładało cykliczne zapełnianie i opróżnianie kolejki. Podczas zapełniania kolejki, której wielkość wynosiła 131072, możliwa byłaby sytuacja, podczas której platforma akwizycji próbkowałaby wyjście kolejki w momencie, gdy jest ona zapełniana. Metoda postprocessingu musiałaby uwzględniać ten fakt. Zaimplementowana kolejka FIFO, po wyżej wymienionym wykorzystaniu w weryfikacji pracy generatora, została usunięta ze schematu. Nowy układ generatora nie posiada kolejki FIFO, a wyjście generatora jest równocześnie wyjściem całego układu. Rozwiązanie to powoduje, że nie każdy bit wyjściowy generatora jest uwzględniany, co dodatkowo może zmniejszyć ewentualną korelację pomiędzy sąsiednimi bitami. Zmodyfikowany moduł znajduje się w załączniku [Załącznik 1], a jego schemat na rysunku **4.2**.



Rysunek 4.2 Schemat układu generatora bez kolejki FIFO

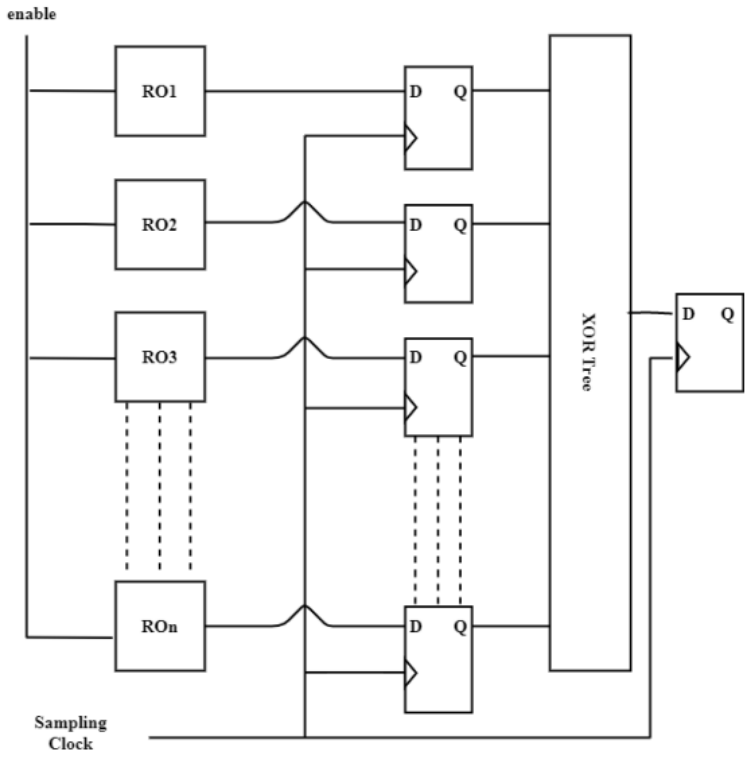
Na potrzeby akwizycji danych moduł posiada jedno bitowe wyjście oraz wejście zegarowe. Wejście zegarowe znajduje się na pinie *PIN_P11* i jest to źródło zegarowe o taktowaniu 50MHz. Sygnał zegarowy został następnie podłączony do pętli synchronizacji fazy, która ma za zadanie zmniejszenie częstotliwości taktowania generatora. Pozwala to na zakumulowanie się różnicy faz pomiędzy sygnałami propagowanymi w oscylatorach polepszając właściwości statystyczne generatora [6]. Deklaracja modułu głównego znajduje się w kodzie źródłowym **4.1**, można tam znaleźć wejściowy sygnał zegarowy *CLK*, oraz jedno bitowe wyjście reprezentujące bit losowy *OUT*.

```
module Main(input CLK, output [0:0] OUT);
```

Kod źródłowy 4.1

W literaturze można znaleźć wiele przykładów implementacji generatorów liczb losowych na płytkach FPGA. Mając na uwadze dobre wyniki opublikowanych testów statystycznych, skalowalność generatora i uniwersalność, w pracy podjęto się implementacji generatora opartego o oscylatory pierścieniowe, który został przedstawiony w [20]. Zaprezentowana w [20] propozycja zawiera oscylatory pierścieniowe zbudowane z pojedynczej bramki NAND i kolejno połączonych szeregowo inwerterów. Źródłem entropii są różne opóźnienia sygnałów propagowanych w oscylatorach związane ze zjawiskami losowymi takimi jak np. szum termiczny. W tym celu zaimplementowano moduł reprezentujący oscylator pierścieniowy przedstawiony w [Załącznik 2]. Następnie w module TRNG [Załącznik 3] zostały umieszone instancje modułu oscylatora, których wyjścia zostały połączone drzewem XOR. Wyjście drzewa XOR stanowi równocześnie wyjście

modułu TRNG dla wariantu bez postprocessingu. Zaproponowany w artykule schemat generatora przedstawia rysunek **4.3**.



Rysunek 4.3 Schemat generatora liczb losowych [20]

Jak można zauważyć na przedstawionym schemacie generator składa się z oscylatorów pierścieniowych, których wyjście jest zapamiętywane przez przerzutniki typu D taktowane przez wspólny sygnał zegarowy. Wyjście przerzutników stanowi wejście drzewa XOR, którego wyjście zapamiętywane jest przez przerzutnik typu D. Przerzutnik ten taktowany jest przez ten sam sygnał zegarowy co przerzutniki na wyjściu oscylatorów pierścieniowych.

Moduł oscylatora pierścieniowego w zastosowanej implementacji [Załącznik 2] zawiera wbudowany przerzutnik D, a na wejście przyjmuje sygnał zegarowy, który powoduje zapamiętanie wyjścia oscylatora podczas zbocza narastającego. Deklarację modułu RO przedstawiono w kodzie źródłowym **4.2**. Autorzy [20] sugerują, że dla 24 inwerterów generator zdał testy statystyczne bez konieczności stosowania postprocessingu dlatego w implementacji oscylatora zastosowano 24 inwertery. Oprócz sygnału taktowania można tu również zauważyć sygnał wejściowy *EN*, oraz rejestr wyjściowy *OUT*. Sygnał wejściowy *EN* stanowi wejście dla bramki NAND, wynika to bezpośrednio z propozycji oscylatora pierścieniowego, która została przedstawiona w [20]. Rejestr wyjściowy stanowi reprezentację przerzutnika typu D na wyjściu oscylatora, jest to jedno-bitowy sygnał. Deklarację tablicy

reprezentującej sygnały przesyłane pomiędzy inwerterami można zobaczyć w kodzie źródłowym **4.3**. Wielkość wektora określana jest przez parametr *delay*, którego deklarację możemy zobaczyć w kodzie źródłowym **4.2**. Kawałek kodu imitujący zachowanie bramki NAND przedstawiono w kodzie źródłowym **4.4**.

```
module RO #(parameter delay = 24)(

   (* preserve *)output reg [0:0] OUT,

   (* keep *)input wire EN/*synthesis keep*/,

   input wire CLK

);
```

Kod źródłowy 4.2

```
wire [delay : 0] nn;
```

Kod źródłowy 4.3

```
assign nn[6] = !(nn[delay] && EN);
```

Kod źródłowy 4.4

Przyjmuje ona na wejście jeden z sygnałów wektora *nn*, oraz sygnał *EN*. W programie Quartus Prime znajdują się gotowe implementacje modułów bramek NAND różnych wielkości, jednak autorzy artykułu wspomnieli, że w ich przypadku potrzebne było zasymulowanie działania bramki NAND, ponieważ stosowanie wbudowanych bramek powodowało, że opóźnienia były za małe, przez co wyjścia oscylatorów nie wprowadzały entropii. Zastosowano pętlę for w celu stworzenia określonej przez parametr delay ilości inwerterów. Wycinek kodu przedstawiający wspomnianą pętlę for tworzącą określoną ilość inwerterów został przedstawiony w kodzie źródłowym **4.5**. Można tu zauważyć deklarację *genvar c*, jest to zmienna używana do generacji pętli *for*. Przechowuje ona dodatnie liczby całkowite. W porównaniu do innych zmiennych języka Verilog jej wartości mogą być modyfikowane podczas kompilacji i elaboracji. Jak wspomniano na stronie hdlworks.com [32], *genvar* musi zostać zadeklarowane w module, w którym jest używane, jednak deklaracja ta może być zarówno wewnątrz, jak i na zewnątrz bloku *generate*. Blok *generate* pozwala powielać instancje modułów oraz operacje, a także tworzyć instancje i operacje

warunkowe, to znaczy w zależności od pewnego warunku można zadeklarować różne moduły i wykonać różne operacje.

```
genvar c;
generate
    for (c = 1; c < delay + 1; c = c + 1)
    begin: inverters
        (* keep *) assign nn[c] = !nn[c-1];
    end
endgenerate
```

Kod źródłowy 4.5

W tym przypadku wykorzystano pętlę for, która reprezentuje powielaną logikę inwertera. Użyta została tu zmienna *c* typu *genvar*. Konstrukcją pętli nie różni się znacznie od większości języków programowania, podzielona jest na kilka najważniejszych bloków. Pierwszy blok mówi o wartości początkowej zmiennej wykorzystanej w pętli, w tym przypadku wartość początkowa *c* została ustawiona na 1. Kolejny definiuje warunek, który musi być spełniony, aby pętla się wykonywała, jeżeli warunek nie jest spełniony, to następuje wyjście z pętli. Wykorzystana pętla posiada warunek *c < delay + 1*, co oznacza że będzie się ona wykonywać dopóki prawdą jest, że *c* jest mniejsze od *delay+1*. Trzeci blok reprezentuje instrukcję wykonywaną podczas każdego przejścia pętli. W zastosowanym podejściu wartość zmiennej *c* jest zwiększana za każdym razem o 1, co odpowiada kolejnym sygnałom reprezentowanym przez wektor *nn*. Ciało pętli zaczyna się od *begin: inverters* i kończy słowem kluczowym *end*. Słowo inverters reprezentuje tu nazwę pętli i nie jest z góry ustalone przez język Verilog. W pętli wykonywana jest prosta czynność przypisująca kolejnemu sygnałowi *nn* wartość poprzedniego sygnału *nn*, co reprezentuje inwerter. Przerzutnik wymaga sygnału zegarowego do prawidłowego działania, w języku Verilog reprezentacja logiki wykonywanej podczas opadającego lub narastającego zbocza zegarowego odbywa się w bloku *always@*. Przykład bloku wykorzystany w [Załącznik 3] znajduje się w kodzie źródłowym **4.6**. W nawiasach został podany warunek wykonania bloku always, w tym przypadku blok zostanie wykonany za każdym razem, gdy sygnał *CLK* będzie zboczem dodatnim. Ciało bloku ponownie zamknięte jest pomiędzy słowami kluczowymi *begin* i *end*. Do rejestru *OUT* przypisywane jest wyjście jednego z inwerterów określone przez wektor *nn* o konkretnym adresie, w tym przypadku jest to adres równy parametrowi *delay*.

Moduł reprezentujący generator liczb losowych bez zastosowania postprocessingu zawarty jest w [Załącznik 3]. Deklaracja modułu przedstawiona jest w kodzie źródłowym **4.7**.

```
always@(posedge CLK)
begin
      (* preserve *) OUT <= nn[delay];
end
```

Kod źródłowy 4.6

```
module rotrng(input CLK, input EN, output reg [0:0] OUT);
```

Kod źródłowy 4.7

Sygnałami wejściowymi są tu sygnał zegarowy *CLK*, oraz sygnał *EN*. Sygnał *EN* służy jako wejście każdej instancji reprezentującej oscylator pierścieniowy. Znajduje się tu również jedno bitowy rejestr *OUT*, który jest rejestrem oznaczonym jako wyjściowy. Moduł generatora liczb losowych składa się z wielu instancji oscylatora pierścieniowego. Deklarację określonej ilości instancji przedstawia kod źródłowy **4.8**.

```
genvar c;
generate
   for (c = 0; c < 1000; c = c + 1)
   begin: ros
       (* preserve *)RO ro_impl (
      .CLK(CLK),
      .EN(EN),
      .OUT(nn[c])); /*synthesis keep*/
   end
endgenerate
```

Kod źródłowy 4.8

Zaimplementowane rozwiązanie posiada 1000 wolno działających oscylatorów. Podobnie jak w przypadku modułu oscylatora pierścieniowego została tu wykorzystana pętla *for*. Podczas każdego wykonania pętli do kolejnych wartości wektora *nn*, którego deklaracja przedstawiona jest w kodzie źródłowym **4.9**. są przypisywane wartości wyjściowe poszczególnych implementacji modułu oscylatora pierścieniowego. W efekcie otrzymujemy sygnały reprezentujące wyjście

każdego z oscylatorów, które następnie mogą być użyte w module generatora. Według [20] wyjścia każdego z przerzutników zawierającego sygnał pojedynczego oscylatora są następnie przepuszczane przed drzewo XOR. Zostało to zaimplementowane w bloku *always* przedstawionym w kodzie źródłowym **4.10**.

```
wire [1000-1 : 0] nn;
```

Kod źródłowy 4.9

```
always@(posedge CLK) begin
   OUT <= ^nn;
end
```

Kod źródłowy 4.10

Drzewo XOR ma jedno bitowe wyjście, które jest przypisywane do rejestru *OUT* podczas każdego zbocza narastającego sygnału zegarowego *CLK*. Rejestr *OUT* jest rejestrem wyjściowym i reprezentuje pojedynczy losowy bit generatora.

Kolejnym krokiem była implementacja metody postprocessingu Von Neumanna [24], która ma na celu zwiększenie losowości bitów. Moduł generatora [Załącznik 3] został w tym celu zmodyfikowany i wzbogacony o dodatkowe rejestry. Zmieniony moduł znajduje się w [Załącznik 4]. Postprocessing odbywał się w bloku *always@*, wycinek kodu zawierający zmodyfikowany względem modułu bez postprocessingu blok *always@* znajduje się w kodzie źródłowym **4.11**.

```
always@(posedge CLK) begin
    skip <= !skip;
    memory3 <= memory2;
    memory2 <= memory1;
    memory1 <= ^nn;
    if(skip == 0 && (memory2 != memory3)) begin
        memory4 <= memory2;
    end
     OUT <= memory4;
end
```

Kod źródłowy 4.11

Rejestr *memory4* przyjmuje wartość rejestru *memory2* w momencie, gdy wartości w rejestrach *memory1* i *memory2* są różne, oraz gdy wartość rejestru *skip* jest zerem. Konieczność wprowadzenia rejestru *skip* wynika z faktu, iż metoda Von Neumanna [24] porównuje ze sobą dwa kolejne bity losowe, tak więc porównanie dwa razy tego samego bitu jest z tą metodą sprzeczne.

Metoda postprocessingu zakładająca wykorzystanie bramki logicznej Exclusive-OR [23] została zaimplementowana poprzez modyfikację modułu generatora podobnie jak miało to miejsce w metodzie Von Neumanna, modyfikacji uległ blok *always@*. Zmodyfikowany moduł znajduje się w [Załącznik 5], a nowy blok *always@* w kodzie źródłowym **4.12**.

```
always@(posedge CLK) begin
    skip <= !skip;
    memory3 <= memory2;
    memory2 <= memory1;
    memory1 <= ^nn;
    if(skip == 0) begin
        memory4 <= memory2 ^ memory3;
    end
    OUT <= memory4;
end
```

Kod źródłowy 4.12

Rejestr skip tak samo jak w przypadku metody Von Neumanna jest tu zastosowany, aby porównywane były ze sobą dwa kolejne bity. Tym razem do rejestru *memory4* przypisywany jest wynik operacji XOR na wartości rejestrów *memory2* i *memory3* w momencie, gdy wartość przechowywana w rejestrze *skip* jest równa zero.

Implementacja generatora liczb losowych na bezpośrednio konfigurowalnej macierzy bramek wiąże się z pewnymi wyzwaniami. W procesie analizy i syntezy implementacja jest optymalizowana. Powtarzalna logika jest usuwana lub znacznie upraszczana. W przypadku generatora liczb losowych opartego o entropię oscylatora pierścieniowego ma to szczególne znaczenie. Oscylator taki składa się z kolejno połączonych inwerterów, co z punktu widzenia urządzenia logicznego jest niepotrzebne. Nieparzystą liczbę inwerterów połączonych szeregowo można zastąpić pojedynczym inwerterem, ponieważ w idealnym przypadku wynik operacji na pojedynczym bicie przez pojedynczy inwerter jest taki sam jak na dowolnej nieparzystej ilości

inwerterów. W przypadku parzystej liczby inwerterów można taki układ zastąpić zwykłym połączeniem, ponieważ na wyjściu otrzymujemy to samo co na wejściu. Program Quartus Prime Lite posiada narzędzie pozwalające na podgląd układu po kompilacji, dzięki czemu w początkowej fazie projektu stwierdzono, że układ w procesie analizy i syntezy został uproszczony. Początkowe podejście do rozwiązania tego problemu zakładało użycie bufora *lcell*. Jak zostało to napisane na stronie intel.com [33] bufor *lcell* alokuje jedną komórkę pamięci dla projektu. Nie jest on usuwany z projektu podczas syntezy logicznej. W [33] wspomniano, że nie powinno się tego bufora używać w celu tworzenia opóźnień czy sygnałów asynchronicznych, ponieważ opóźnienie może być zmienne w zależności od temperatury, czy zasilacza. Przykładową deklarację bufora *lcell* przedstawiono w kodzie źródłowym **4.13**. Można tu zauważyć sygnał wejściowy *.in,* oraz sygnał wyjściowy *.out*. Rozwiązanie to nie znalazło się jednak w ostatecznym projekcie. Jest to spowodowane niezgodnością z artykułem [20], ponieważ jego autorzy nie przewidzieli występowania dodatkowych elementów opóźniających. W celu możliwie rzetelnego odzwierciedlenia implementacji przedstawionej w artykule zaistniała potrzeba zmiany podejścia do zapobiegania optymalizacji. Kolejnym minusem tej koncepcji było to, że komórka pamięci musiała znajdować się w każdym module, przez co niezbędne stało się tworzenie modułów wszystkich bramek logicznych z użyciem *lcell*. Na stronie intel.com [34] można znaleźć informację o wspieranych przez program Quartus Prime atrybutach i dyrektywach dotyczących syntezy Verilog HDL. Kluczowe okazały się atrybuty *keep* oraz *preserve*, dzięki którym możliwa stała się rezygnacja z bufora *lcell*. Każdy atrybut zapisywany jest w nawiasie pomiędzy gwiazdkami, przykład znajduje się w kodzie źródłowym **4.14**. Atrybut może zostać również zapisany w komentarzu, jednak w tym przypadku musi zostać poprzedzony słowem kluczowym *synthesis*. Przykład znajduje się w kodzie źródłowym **4.15**.

```
lcell lc1 (.in(in), .out(out));
```

Kod źródłowy 4.13

```
(* preserve *) reg my_reg;
```

Kod źródłowy 4.14 [34]

```
reg my_reg /* synthesis preserve */;
```

Kod źródłowy 4.15 [34]

Atrybut *preserve* powoduje, że podczas analizy i syntezy rejestr nie jest optymalizowany, gdy usuwane są nadmiarowe rejestry. Kolejny atrybut *keep* pozwala na wskazanie połączeń, które nie powinny być optymalizowanie podobnie jak ma to miejsce w przypadku rejestrów. Dzięki zastosowaniu tych dwóch atrybutów możliwa była synteza implementacji do postaci przedstawionej w artykule.

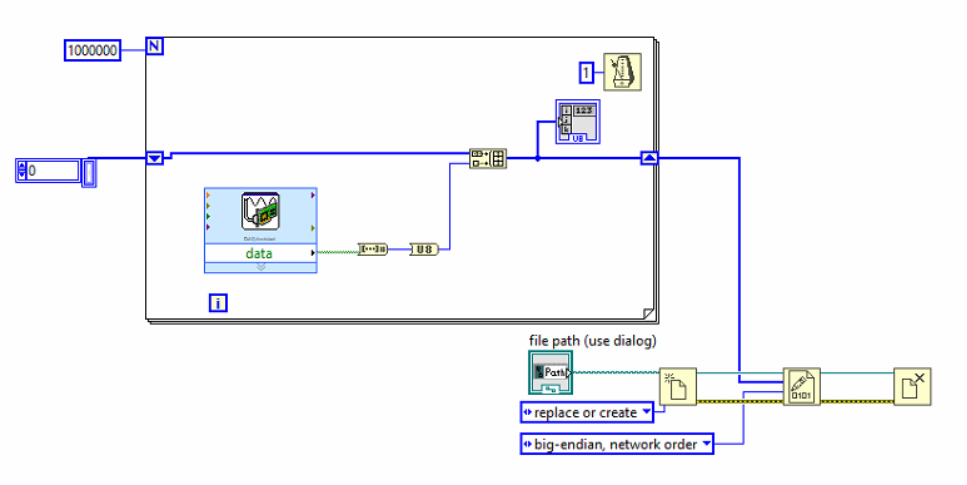
Realizowany projekt generatora taktowany jest źródłem zegarowym o częstotliwości 50MHz [31]. Sygnał zegarowy jest następnie przepuszczany przez pętlę synchronizacji fazy w celu zmniejszenia prędkości taktowania, co pozwala na polepszenie właściwości statystycznych generatora. Zaimplementowane rozwiązanie wykorzystuje 10-krotne spowolnienie taktowania, w praktyce oznacza to przepustowość generatora rzędu 5MHz dla wariantu bez postprocessingu. Zastosowanie postprocessingu wiąże się ze zmniejszeniem przepustowości. Dla techniki Von Neumanna maksymalna przepustowość jaką można uzyskać wynosi 25% przepustowości oryginalnej [24]. Biorąc pod uwagę informacje przedstawione w [24], dla wariantu generatora z postprocessingiem Von Neumanna jesteśmy w stanie uzyskać maksymalną przepustowość rzędu 1.25MHz. Postprocessing z wykorzystaniem bramki XOR bierze pod uwagę dwa kolejne bity sekwencji losowej, powoduje to zmniejszenie przepustowości wyjściowej do 50% przepustowości oryginalnej, czyli w praktyce dla implementowanego rozwiązania przepustowość generatora dla wariantu z postprocessingiem Exclusive-OR wyniesie 2.5MHz.

4.2. Akwizycja danych

Akwizycja danych odbywała się przy pomocy płytki NI USB-6002. Specyfikacja płytki została przedstawiona w [35]. Maksymalna prędkość próbkowania danych wynosi 50kS/s. Jedno z wejść platformy akwizycji zostało połączone z wyjściem generatora liczb losowych. W tym przypadku zastosowano połączenie pojedynczym przewodem. Został on podłączony do wyjścia generatora, którego bity były wystawiane na pinie *PIN_V10*. Drugi koniec przewodu został podłączony do wejścia platformy akwizycji.

Implementacja programu pobierającego dane odbyła się przy użyciu środowiska programistycznego LabVIEW. Za stworzenie oprogramowania odpowiada firma NI zwana dawniej National Instruments [36]. Jest to środowisko pozwalające na programowanie graficzne w języku G [36]. Zaimplementowany w LabVIEW program pozwala na pobieranie próbek z wyjścia generatora za pomocą platformy do akwizycji, jego schemat przedstawiono na rysunku **4.4**. Platforma próbkuje jedno bitowe wyjście generatora liczb losowych. Na schemacie można zauważyć elementy konwertujące pobrany z generatora bit do postaci 8 bitowej liczby bez znaku.

W przypadku, gdy wyjście generatora jest równe 1, będzie to odpowiadało liczbie zapisanej w systemie binarnym jako *00000001*, a w przypadku gdy wyjście generatora jest równe zero uzyskamy *00000000*. Bity są następnie zapisywane w pliku z rozszerzeniem *bin*.

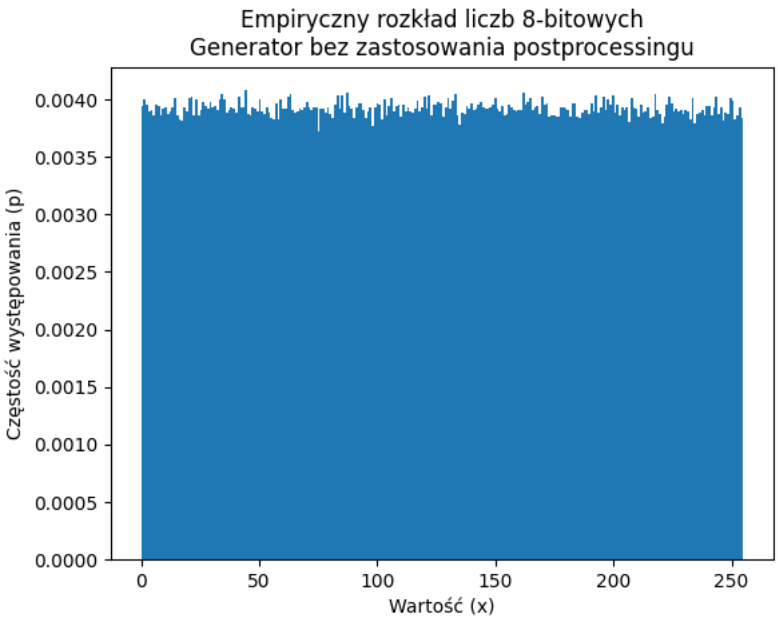


Rysunek 4.4 Schemat programu akwizycji danych w programie LabVIEW

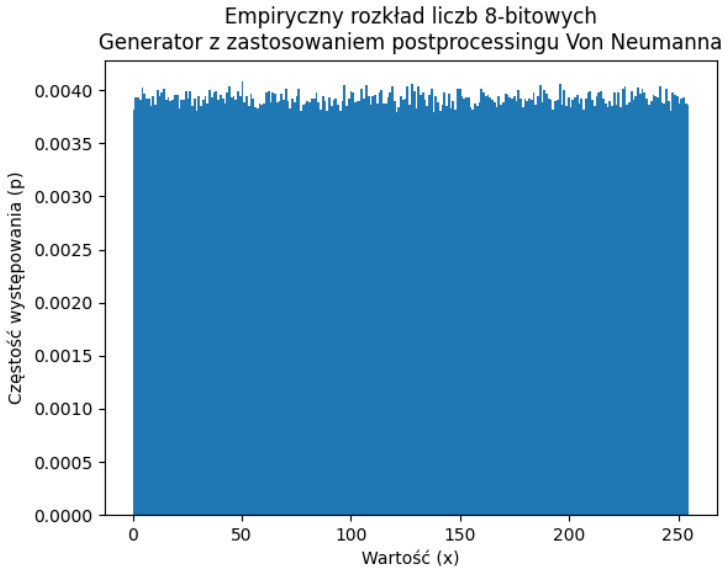
Jedno bitowe wyjście generatora próbkowane jest w odstępie 1ms, a następnie zapisywane do pliku. Generator działa bez przerwy wystawiając próbki z większą częstotliwością niż są one pobierane. Dane są przesyłane z platformy akwizycji do komputera za pomocą kabla USB. Przesyłanie danych odbywa się przy wykorzystaniu uniwersalnego asynchronicznego nadajnika-odbiornika (ang. *universal asynchronous receiver-transmitter, UART*) [37]. Dzięki układowi scalonemu jesteśmy w stanie przesłać dane przez port szeregowy. Jest to uniwersalny standard wspierany przez wiele urządzeń.

4.3 Testy statystyczne

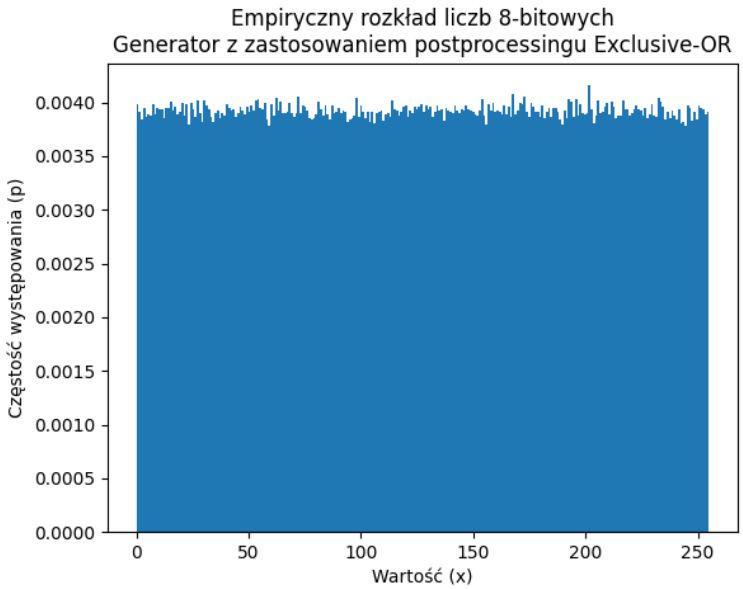
Histogramy wygenerowanych bitów losowych wyrysowano dla wartości 8 bitowych, co w efekcie oznacza liczby z zakresu od 0 do 255. Każdy histogram opracowany jest na podstawie 1 000 000 losowych liczb 8 bitowych, czyli 8 000 000 bitów i przedstawiono je na rysunkach **4.5**, **4.6**, **4.7** odpowiednio dla wariantu bez postprocessingu, z zastosowaniem postprocessingu Von Neumanna i dla postprocessingu Exclusive-OR.



Rysunek 4.5 Empiryczny rozkład liczb 8-bitowych dla wariantu generatora bez postprocessingu



Rysunek 4.6 Empiryczny rozkład liczb 8-bitowych dla wariantu generatora z postprocessingiem Von Neumanna



Rysunek 4.7 Empiryczny rozkład liczb 8-bitowych dla wariantu generatora z postprocessingiem Exclusive-OR

Jak można zauważyć przedstawione na rysunkach histogramy przypominają rozkład równomierny.

Kolejnym krokiem oceny właściwości statystycznych zaimplementowanego generatora było przeprowadzenie testów zdefiniowanych przez National Institute of Standards and Technology (NIST) [28]. Oprogramowanie NIST Statistical Test Suite wraz z dokumentacją, zawierające unormowane testy statystyczne znajduje się w [42]. Podczas oceny statystycznej wykorzystano oprogramowanie w wersji 2.1.2. Do przeprowadzenia testów z generatora zebrano 20 000 000 bitów i zapisano je w postaci pliku binarnego. Testy przeprowadzono dla 20 sekwencji po 1 000 000 bitów każda. Domyśle ustawienia testów nie zostały zmienione i są zgodne z ustawieniami testów dla danego programu w wersji 2.1.2. Format wejściowy pliku został ustawiony na binarny. Oznacza to, że każdy bajt w pliku danych zawiera 8 bitów danych.

Analizie za pomocą testów NIST zostały poddane trzy przypadki, którymi są generator bez postprocessingu, generator wykorzystujący metodę postprocessingu Von Naumanna oraz generator oparty o metodę postprocessingu Exclusive-OR. W każdym z tych przypadków ustawienia testów pozostały niezmienione w stosunku do ustawień domyślnych programu. NIST Statistical Test Suite zapisuje szczegółowe wyniki każdego z testów oraz plik zbiorczy zawierający raport ze wszystkich przeprowadzonych testów. Wygenerowane raporty można znaleźć w załącznikach [Załącznik 6], [Załącznik 7], [Załącznik 8], które zawierają wyniki testów przeprowadzonych odpowiednio na generatorze bez zastosowania metody postprocessingu, generatorze opartym o metodę postprocessingu Von Naumanna i generatorze opartym o metodę

postprocessingu Exclusive-OR. Wybrane wyniki zawierające stosunek testów zdanych do niezdanych zawarto w tabeli 4.1

Tabela 4.1 Wyniki testów NIST dla 1000 oscylatorów po 24 inwertery każdy.

Test	Bez postprocessingu	Von Neumann	Exclusive-OR
Frequency	19/20	20/20	20/20
BlockFrequency	20/20	20/20	20/20
CumulativeSums	40/40	40/40	40/40
Runs	20/20	20/20	20/20
LongestRun	20/20	19/20	20/20
Rank	20/20	20/20	20/20
FFT	20/20	20/20	20/20
NonOverlappingTemplate	2931/2960	2927/2960	2933/2960
OverlappingTemplate	19/20	20/20	20/20
Universal	20/20	20/20	20/20
ApproximateEntropy	20/20	20/20	19/20
Serial	39/40	40/40	40/40
LinearComplexity	19/20	20/20	20/20

Testy przeprowadzone dla 20 sekwencji uznawane są za zdane w przypadku zdania przynajmniej 18 podtestów, co oznacza próg zdawalności na poziomie 90%. Zaimplementowany generator dla 1000 oscylatorów pierścieniowych po 24 inwertery każdy był w stanie zdać testy nawet bez zastosowania postprocessingu. Warianty generatora zawierające metody postprocessingu Von Neumanna i Exclusive-OR również zdały wszystkie testy. Po zastosowaniu postprocessingu udało się poprawić wyniki niektórych z nich. Są jednak takie, których wyniki są gorsze po zastosowaniu metody postprocessingu. Jako przykład może posłużyć ApproximateEntropy test, wariant generatora bez postprocessingu przeszedł wszystkie 20 podtestów, jednak w przypadku zastosowania metody Exclusive-OR tylko 19 z 20 podtestów zostało zdanych. Wariant generatora z postprocessingiem Exclusive-OR test przeszedł, jednak z wynikiem gorszym niż wariant bez postprocessingu. Gorszy wynik w pojedynczym teście nie musi oznaczać, że zastosowanie danej metody postprocessingu pogarsza właściwości statystyczne względem wariantu bez postprocessingu, a nawet jak wspomniano w [39], statystycznie możliwe jest niezdanie jakiegoś

testu w poprawnie działającym generatorze. Jak twierdzą autorzy [39] prawdopodobieństwo zaistnienia takiej sytuacji zwiększa się wraz z liczbą podtestów.

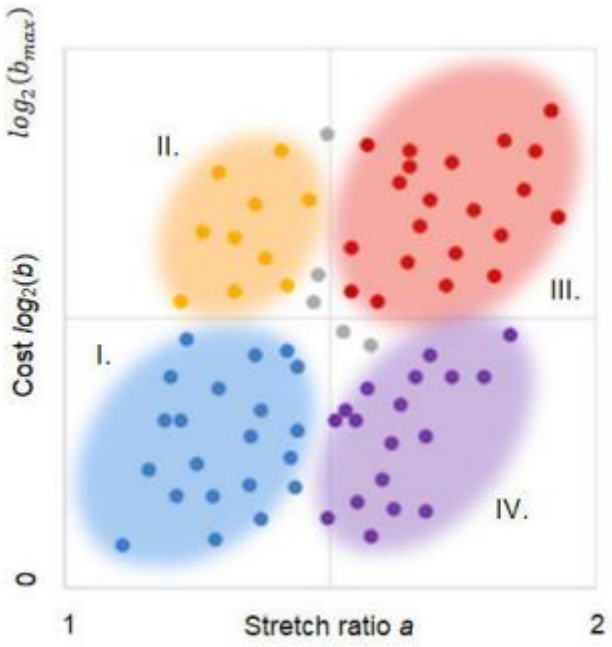
Ocena właściwości statystycznych generowanych ciągów liczb losowych została dokonana również przy zastosowaniu algorytmu dopasowania sekwencji (*ang. Sequence Alignment Algorithm, SAA*) [40]. Jak podają autorzy [40] algorytm dokonuje wzajemnego dopasowania dwóch losowych sekwencji poprzez sukcesywne szukanie wspólnych elementów i poprzez wprowadzanie przerw w sekwencji dla elementów, które nie zostały dopasowane w funkcji kosztu wykładniczego. Wynikiem analizy są dwie metryki, koszt i współczynnik rozciągnięcia. Współczynnik rozciągnięcia definiowany jest jako współczynnik długości sekwencji po wprowadzeniu przerw do długości sekwencji pierwotnej. Dodatkowo w [40] wspomniano, że w wyniku działania algorytmu sekwencje ulegają dynamicznemu zwiększaniu, co może doprowadzić do wytworzenia nadmiarowych elementów w jednej z sekwencji, które nie będą miały dopasowania w drugiej sekwencji. Równanie **4.1** prezentuje przedstawione przez autorów [40] równanie, gdzie a_i jest miarą współczynnika rozciągnięcia, L_i' reprezentuje długość sekwencji po wprowadzeniu przerw, L_i jest długością początkową sekwencji, a L_t określone jest jako liczba elementów nadmiarowych, które nie posiadają dopasowania w drugiej sekwencji.

$$a_i = (L_i' - L_t)/(L_i/L_t) \tag{4.1}$$

Miara kosztu rozciągnięcia przedstawiona jest przez autorów jak na równaniu **4.2**. Zgodnie z artykułem j reprezentuje indeks kolejnej serii przerw, a G_i^j jest równe liczbie przerw w serii numer j.

$$b_i = \sum_i 2^{G_i^j - 1}/(L_i/L_t) \tag{4.2}$$

Uzyskane za pomocą równań **4.1** i **4.2** wyniki autorzy reprezentują na dwuwymiarowym wykresie, którego osią poziomą jest miara współczynnika rozciągnięcia, a osią pionową miara kosztu w skali logarytmicznej. Następnie dokonują oni podziału wykresu na cztery znaczeniowe podgrupy, co zaprezentowano na rysunku **4.8**.



Rysunek 4.8 Wykres przedstawiający podział wykresu na cztery znaczeniowe podgrupy [40]

Cztery grupy znaczeniowe zostały w [40] opisane następująco:

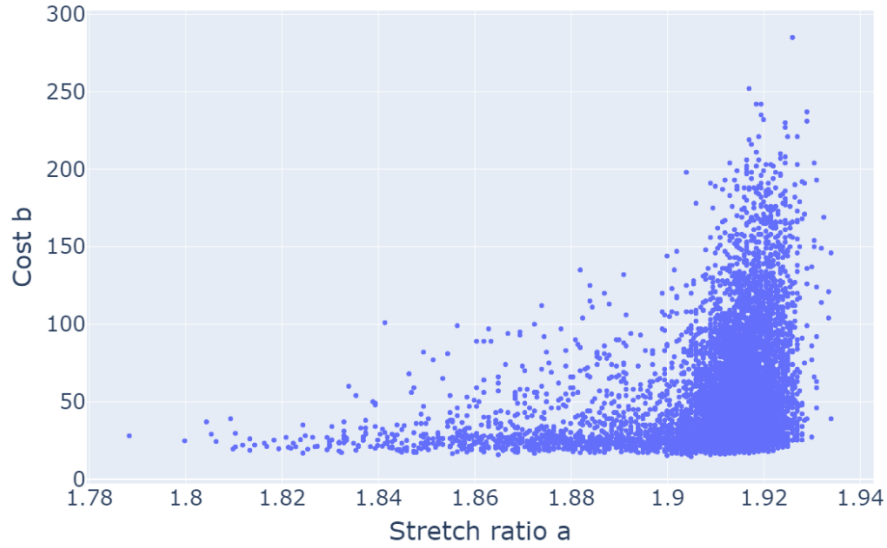
I. Niski poziom rozciągnięcia wskazuje na duże podobieństwo sekwencji. Niski koszt może wskazywać na pojedyncze i rozsiane przerwy, a zatem rzadkie niedopasowania.

II. Podobnie jak w przypadku I mówimy o niskim współczynniku rozciągnięcia, który oznacza wysokie podobieństwo. Wysoki koszt oznacza dużo luk pod rząd.

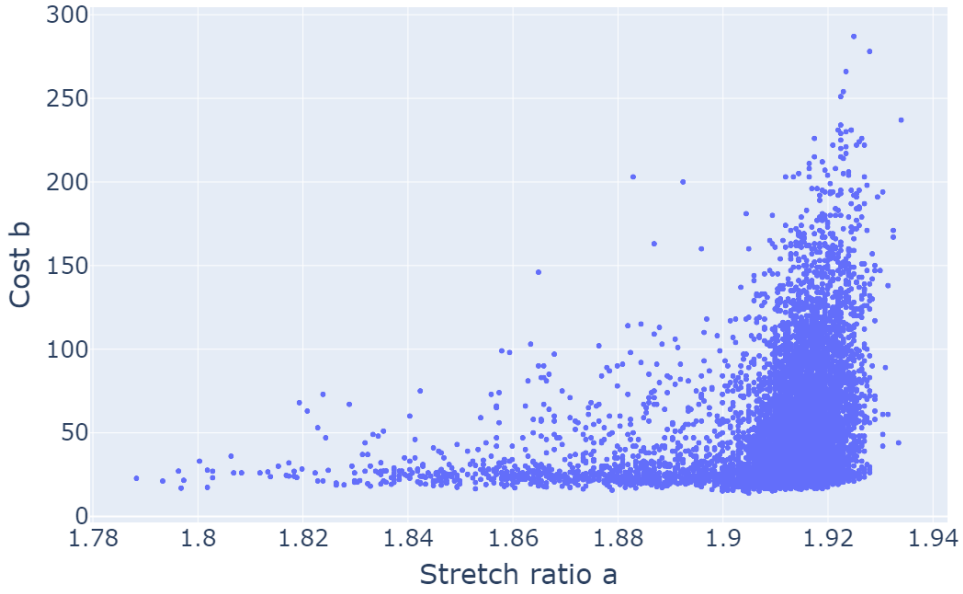
III. Wysoki współczynnik rozciągnięcia i duży koszt są rezultatem długich serii wprowadzanych luk przeplatanych rzadkimi dopasowaniami. Oznacza to brak podobieństwa pomiędzy sekwencjami

IV. Wysoki poziom rozciągnięcia podczas utrzymywania niskiego kosztu oznacza częste pojedyncze luki. Stanowi to przesłankę, że może występować statystyczna zależność ciągów

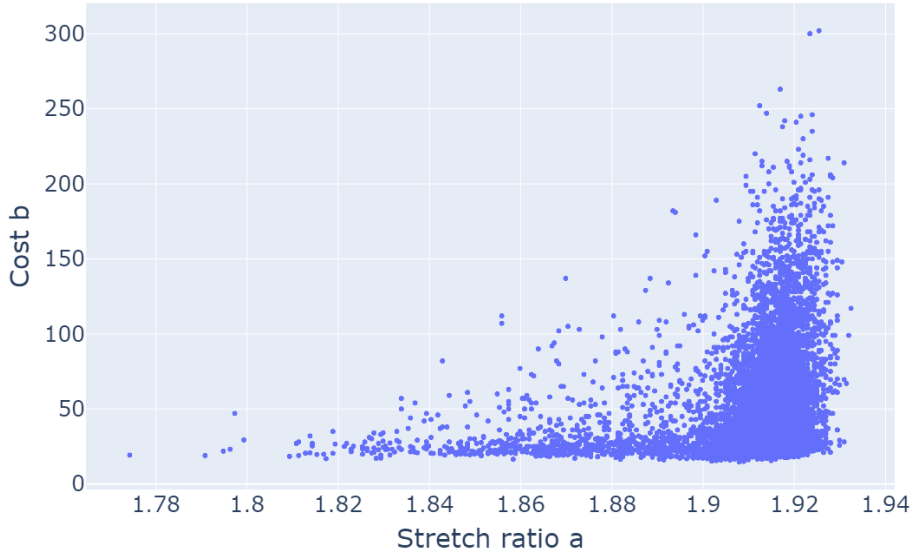
Metryki SAA zostały wyznaczone za pomocą platformy ORANGUTAN Politechniki Poznańskiej [41]. Jako dane wejściowe platforma przyjmuje pliki binarne. Wielkość danych wejściowych to 1 000 000 bitów. Analizy dokonano dla trzech wariantów generatora, wyniki przedstawiono na rysunkach **4.9**, **4.10**, **4.11** odpowiednio dla generatora bez postprocessingu, generatora z postprocessingiem Von-Neumanna i generatora z postprocessingiem Exclusive-OR.



Rysunek 4.9 Uzyskane wyniki metryk SAA dla generatora bez postprocessingu



Rysunek 4.10 Uzyskane wyniki metryk SAA dla generatora z postprocessingiem Von-Neumanna



Rysunek 4.11 Uzyskane wyniki metryk SAA dla generatora z postprocessingiem Exclusive-OR

Uzyskane wykresy są podobne dla wszystkich trzech przypadków. Większość wyników znajduje się w III i IV ćwiartce, co oznacza, że posiadają wysoki współczynnik rozciągnięcia. Luki tworzone są podczas niedopasowania sekwencji, tak więc duży współczynnik rozciągnięcia potwierdza małe podobieństwo sekwencji.

Podsumowanie

Generowanie liczb rzeczywiście losowych stanowi bardzo ważny element cyberbezpieczeństwa. Posiadają one zastosowanie w wielu znanych algorytmach i pomagają zapewnić poufność, integralność i niezaprzeczalność informacji. Losowe sekwencje wykorzystywane na potrzeby cyberbezpieczeństwa powinny się cechować równomiernym prawdopodobieństwem wystąpienia, a wartość kolejnych generowanych liczb powinna być nieprzewidywalna. Powstało wiele artykułów dotyczących generowania liczb rzeczywiście losowych. Przedstawiono w pracy artykuły opisujące implementacje generatorów opartych o różne źródła entropii takie jak szumy [12],[13], chaos [14],[17], generatory oparte na oscylatorach pierścieniowych [19],[20], metastabilność [21], czy interakcję człowiek-komputer [22]. Generatory te posiadają różne zastosowania i mogą być implementowane za pomocą różnych narzędzi. Poszczególne metody postprocessingu takie jak postprocessing za pomocą bramki XOR [23], metoda Von Neumanna [24], funkcje haszujące [25], funkcja liniowa [26], czy wykorzystanie rejestru przesuwnego [27] pozwalają na poprawienie właściwości statystycznych generowanych ciągów losowych zwiększając równocześnie jakość generatora.

Negatywnym skutkiem postprocessingu jest zmniejszenie przepustowości, czyli prędkości generowania bitów.

Implementację generatora przeprowadzono według artykułu [20]. Autorzy [20] przedstawiają dwa podejścia, jedno oparte jest o oscylatory pierścieniowe, drugie podejście zakłada wprowadzenie chaosu do oscylatora pierścieniowego. Rozwiązanie oparte o oscylatory pierścieniowe zostało zaprogramowane w FPGA i napisane w języku opisu sprzętu Verilog. Według autorów artykułu wariant generatora dla 24 inwerterów zdał wszystkie testy bez zastosowania postprocessingu, dlatego podczas implementacji stosowano się do tych parametrów. Ilość oscylatorów pierścieniowych została ustawiona na 1000. Po prawidłowym zaimplementowaniu przedstawionego w [20] generatora dokonano implementacji postprocessingu Von Neumanna i postprocessingu z wykorzystaniem bramki XOR. Wszystkie trzy warianty, czyli generator bez postprocessingu, generator z postprocessingiem Von Neumanna i generator z wykorzystaniem bramki XOR zostały następnie poddane testom NIST. Zostały dla nich również wyrysowane empiryczne rozkłady wygenerowanych liczb losowych i wyznaczone metryki SAA. Przeprowadzone testy NIST wykazały, że wszystkie trzy warianty generatora zdały testy. Wyrysowane rozkłady empiryczne wygenerowanych liczb losowych przypominają rozkład równomierny, a wyznaczone metryki SAA sugerują, że nie mamy do czynienia z wysokim podobieństwem pomiędzy kolejnymi generowanymi sekwencjami. Generator ten dla wariantu bez postprocessingu jest w stanie zapewnić przepustowość rzędu 5MHz równocześnie zachowując dobrą jakość statystyczną generowanych bitów, co podkreśla przeprowadzona analiza statystyczna. Rozwiązanie to raczej nie znajdzie zastosowania w różnego rodzaju symulacjach zjawisk losowych, ponieważ te zazwyczaj wykorzystują głównie generatory pseudolosowe [3]. Projekty implementowane w FPGA cechują się tym, że to samo rozwiązanie może być zaimplementowane w innym miejscu na innej platformie. Ponad to jak twierdzą autorzy [42] generatory implementowane za pomocą FPGA są układami cyfrowymi, a ponieważ systemy kryptograficzne są w większości konstrukcjami cyfrowymi, to stawiane są wymagania mówiące o tym, że generatory liczb rzeczywiście losowych powinny być zbudowane z wykorzystaniem układów cyfrowych. Według autorów [20] generatory zaimplementowane z wykorzystaniem FPGA można zintegrować z produktami Internetu Rzeczy. Można z tego wywnioskować, że zrealizowany generator może zostać zastosowany jako element systemu kryptograficznego stosowanego na potrzeby Internetu Rzeczy.

Bibliografia

[1] C. Batanero, „Understanding randomness. Challenges for research and teaching”, *Ninth Congress of European Research in Mathematics Education*, Prague, 2015

[2] H. S Harris, „Leukippos and Demokritos”, *The Reign of the Whirlwind*, 1999

[3]Wikipedia, „Applications of randomness” [Online], Available: https://en.wikipedia.org/wiki/Applications_of_randomness

[4] Khan Academy, "Pseudorandom number generators", 2016.

[5]Wikipedia, „Metoda Monte Carlo” [Online], Available: https://pl.wikipedia.org/wiki/Metoda_Monte_Carlo

[6] O. Petura. “True random number generators for cryptography: Design, securing and evaluation. Micro and nanotechnologies/Microelectronics”, Université de Lyon, 2019.

[7] G. Heartsfield, “Insufficient Session-ID Length”, https://owasp.org/www-community/vulnerabilities/Insufficient_Session-ID_Length

[8] B. Schneier: Kryptografia dla praktyków: protokoły, algorytmy i programy źródłowe w języku C. Warszawa: Wydawnictwa Naukowo-Techniczne, 2002, s. 341–382

[9] H. Bar-El, “Introduction to Side Channel Attacks” http://gauss.ececs.uc.edu/Courses/C653/lectures/SideC/intro.pdf

[10] Hu, Yue & Liao, Xiaofeng & Wong, Kwok-wo & Zhou, Qing. (2009). A true random number generator based on mouse movement and chaotic cryptography. Chaos, Solitons & Fractals. 40. 2286-2293. 10.1016/j.chaos.2007.10.022.

[11] C. E. Shannon, "A mathematical theory of communication," in The Bell System Technical Journal, vol. 27, no. 3, pp. 379-423, July 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

[12] H. Zhun, Ch. Hongyi, "A truly random number generator based on thermal noise," ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549), 2001, pp. 862-864, doi: 10.1109/ICASIC.2001.982700.

[13] F. Tehranipoor, P. Wortman, N. Karimian, W. Yan and J. A. Chandy, "DVFT: A Lightweight Solution for Power-Supply Noise-Based TRNG Using Dynamic Voltage Feedback Tuning System," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 6, pp. 1084-1097, June 2018, doi: 10.1109/TVLSI.2018.2804258.

[14] M. Blaszczyk and R. A. Guinee, "A true random binary sequence generator based on chaotic circuit," IET Irish Signals and Systems Conference (ISSC 2008), 2008, pp. 294-299, doi: 10.1049/cp:20080678.

[15] Wikipedia, “Chua's circuit” [Online], Available: https://en.wikipedia.org/wiki/Chua%27s_circuit

[16] Wikipedia, “Chua's diode” [Online], Available: https://en.wikipedia.org/wiki/Chua%27s_diode

[17] J. A. Aguilar Angulo, E. Kussener, H. Barthelemy and B. Duval, "Discrete chaos - based Random Number Generator," 2014 IEEE Faible Tension Faible Consommation, 2014, pp. 1-4, doi: 10.1109/FTFC.2014.6828610.

[18] Fischer, V., Bernard, F., & Bochard, N. (2019). Modern random number generator design – Case study on a secured PLL-based TRNG. it - Information Technology, 61, 3 - 13.

[19] B. Sunar, W. J. Martin and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks," in IEEE Transactions on Computers, vol. 56, no. 1, pp. 109-119, Jan. 2007, doi: 10.1109/TC.2007.250627.

[20] A. Zacharias, C. G. Gisha and B. A. Jose, "Chaotic Ring Oscillator Based True Random Number Generator Implementations in FPGA," 2020 24th International Symposium on VLSI Design and Test (VDAT), 2020, pp. 1-6, doi: 10.1109/VDAT50263.2020.9190505.

[21] Hata, Hisashi & Ichikawa, Shuichi. (2012). FPGA Implementation of Metastability-Based True Random Number Generator. IEICE Transactions. 95-D. 426-436. 10.1587/transinf.E95.D.426.

[22] Thomas, Antu & Varghese, Paul. (2018). Non-Recurring Improved Random Number Generator- a new step to improve cryptographic algorithms. EAI Endorsed Transactions on Energy Web. 5. 154813. 10.4108/eai.12-6-2018.154813.

[23] R. B. Davies, Exclusive OR (XOR) and hardware random number generators [Online], Available: http://www.robertnz.net/pdf/xor2.pdf

[24] J. Von Neumann, Various Techniques Used in Connection with Random Digits, Applied Mathematics Series 12, National Bureau of Standards, Washington, DC, 1951, pp. 36-38.

[25] S. Halevi. Cryptographic Hash Functions and their many applications. In: USENIX Security Symposium. (2009) http://people.csail.mit.edu/shaih/pubs/Crypto graphic-Hash-Functions.ppt

[26] Kwok Siew-Hwee. Ee Yen-Ling, Chew Guanhan, Zheng Kanghong, Khoo Khoongming, Tan Chik How. (2011). “A Comparison of Post-Processing Techniques for Biased Random Number Generators” 175-190. 10.1007/978-3-642-21040-2_12.

[27] J. D. J. Golic, "New Methods for Digital Generation and Postprocessing of Random Data," in IEEE Transactions on Computers, vol. 55, no. 10, pp. 1217-1229, Oct. 2006, doi: 10.1109/TC.2006.164.

[28] NIST, “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications” [Online], Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf

[29] M. Epstein, L. Hars, R. Krasinski, M. Rosner, and H. Zheng, “Design and Implementation of a True Random Number Generator Based on Digital Circuits Artifacts,” Proc. Conf. Cryptographic Hardware and Embedded Systems (CHES ’03), pp. 152-165, 2003

[30] Xilinx, “What is FPGA?” [Online], Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[31] Terasic, “DE10-Lite User Manual” [Online], Available: https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-2912030810549-de10-lite-user-manual.pdf

[32] Hdlworks, “Genvar” [Online], Available: https://www.hdlworks.com/hdl_corner/verilog_ref/items/Genvar.htm

[33]Intel, “LCELL Primitive” [Online], Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/prim/prim_file_lcell.htm

[34] Intel, “Verilog HDL Synthesis Attributes and Directives” [Online], Available: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir.htm

[35] NI, “NI USB-6002 Specification” [Online], Available: https://www.ni.com/pdf/manuals/374371a.pdf

[36] Wikipedia, “National Instruments” [Online], Available: https://pl.wikipedia.org/wiki/National_Instruments

[37] M. Kościelnicka, „Wykład 5: Komunikacja ze światem zewnętrznym - UART” [Online], Available: https://www.mimuw.edu.pl/~mwk/pul/05_uart/index.html

[38] NIST, „NIST SP 800-22: Download Documentation and Software” [Online], Available: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software

[39] M. Sýs, Z. Riha, V. Matyas, K. Marton, A Suciu, (2015). On the interpretation of results from the NIST statistical test suite. 18. 18-32.

[40] J. Nikonowicz, Ł. Matuszewski, P. Kubczak, „Sequence Alignment Algorithm for Statistical Similarity Assessment”, IEEE Access 9 (2021): 102153-102160.

[41] Politechnika Poznańska [Online], Available: http://orangutan.put.poznan.pl/smetric/upload

[42] M. Jessa , Ł. Matuszewski, M. Jaworski, „Losowość generatora TRNG zaimplementowanego w FPGA”, *Pomiary Automatyka Kontrola*. 2011, R. 57, nr 8s. 880–882.

Załączniki

[Załącznik 1] Moduł Main bez FIFO

```
module Main(input CLK, output [0:0] OUT);
	wire clk_trng;


	(*keep = "true"*)rotrng rotrng1(
		.CLK(clk_trng),
		.EN(1),
		.OUT(rotrng_out)
	);

	pll pll0 (
		.inclk0(CLK),
		.c0(clk_trng)
	);


	assign OUT = rotrng_out;
endmodule
```

[Załącznik 2] Moduł RO

```
module RO #(parameter delay = 24)(
(* preserve *)output reg [0:0] OUT,
                                (* keep *)input wire EN/*synthesis keep*/,
                                input wire CLK,
                                input wire RST
   );


        (* keep *)wire [delay : 0] nn;

        (* preserve *)(* keep *)assign nn[0] = !(nn[delay] && EN);/*synthesis keep*/

        genvar c;
        generate
                for (c = 1; c < delay + 1; c = c + 1)
                begin: inverters
                        (* keep *) assign nn[c] = !nn[c-1];
                end
        endgenerate


always@(posedge CLK)
begin
        (* preserve *)OUT <= nn[delay];/*synthesis keep*/
end


endmodule
```

[Załącznik 3] Moduł TRNG bez postprocessingu

```
module rotrng(input CLK, input EN, output reg [0:0] OUT);

	wire [1000-1 : 0] nn;

	genvar c;
	generate
		for (c = 0; c < 1000; c = c + 1)
		begin: ros
			(* preserve *)RO ro_impl (.CLK(CLK), .EN(EN),
.OUT(nn[c]));/*synthesis keep*/
		end
	endgenerate


	always@(posedge CLK) begin
		OUT <= ^nn;
	end
endmodule
```

[Załącznik 4] Moduł TRNG z postprocessingiem Von-Neumanna

```
module rotrng(input CLK, input EN, output reg [0:0] OUT);

	wire [1000-1 : 0] nn;
	reg skip = 0;
	reg memory1;
	reg memory2;
	reg memory3;
	reg memory4;

	genvar c;
	generate
		for (c = 0; c < 1000; c = c + 1)
		begin: ros
			(* preserve *)RO ro_impl (.CLK(CLK), .EN(EN),
.OUT(nn[c]));/*synthesis keep*/
		end
	endgenerate


		always@(posedge CLK) begin

			skip <= !skip;

			memory3 <= memory2;
			memory2 <= memory1;
			memory1 <= ^nn;

			if(skip == 0 && (memory2 != memory3)) begin
				memory4 <= memory2;
			end


			OUT <= memory4;

		end
endmodule
```

[Załącznik 5] Moduł TRNG z postprocessingiem Exclusive-OR

```
module rotrng(input CLK, input EN, output reg [0:0] OUT);

	wire [1000-1 : 0] nn;
	reg skip = 0;
	reg memory1;
	reg memory2;
	reg memory3;
	reg memory4;

	genvar c;
	generate
		for (c = 0; c < 1000; c = c + 1)
		begin: ros
			(* preserve *)RO ro_impl (.CLK(CLK), .EN(EN),
.OUT(nn[c]));/*synthesis keep*/
		end
	endgenerate


	 always@(posedge CLK) begin

		skip <= !skip;

		memory3 <= memory2;
		memory2 <= memory1;
		memory1 <= ^nn;

		if(skip == 0) begin
			memory4 <= memory2 ^ memory3;
		end


		OUT <= memory4;

	    end
endmodule
```