



**AGH**

**AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W  
KRAKOWIE**

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,  
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

**KATEDRA METROLOGII I ELEKTRONIKI**

**Projekt dyplomowy**

*Projekt warstwy kodującej protokołu Interlaken w układzie  
FPGA.*

*Design of a physical coding sublayer for Interlaken protocol  
for FPGAs.*

Autor:

*Jan Wołowiec*

Kierunek studiów:

*Mikroelektronika w Technice i Medycynie*

Opiekun pracy:

*dr inż. Piotr Otfinowski*

Kraków, 2021

*Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór; artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.*

*Serdecznie dziękuję rodzinie i promotorowi za wsparcie okazane w trakcie pisanie tej pracy.*



# Spis treści

<b>1. Wprowadzenie</b>	7
1.1. Cele pracy	7
1.2. Zawartość pracy	7
<b>2. Część teoretyczna</b>	9
2.1. Protokół Interlaken	9
2.1.1. Format słów	10
2.1.2. Meta Frame	11
2.1.3. Niwelacja offsetu DC	11
2.2. Wykorzystane bloki logiczne	12
2.2.1. IP Core	12
2.2.2. Transceiver	12
2.2.3. SerDes	12
2.2.4. Scrambler	13
2.2.5. Gearbox	15
<b>3. Implementacja układu</b>	17
3.1. Architektura i wykorzystane narzędzia	17
3.2. Projekt warstwy kodującej	18
3.2.1. Hierarchia modułu głównego	19
3.2.2. Podmoduł Core	21
3.2.3. Podmoduł TX	25
3.2.4. Podmoduł RX	30
<b>4. Weryfikacja układu</b>	37
4.1. Bloki VIO	38
4.2. Bloki ILA	39
4.3. Frame Generator	39

4.4. Frame Checker.....	40
4.5. Stream Manipulator.....	41
4.6. Przeprowadzone testy .....	41
4.6.1. Zmiana offsetu w bloku Stream Manipulator .....	42
4.6.2. Zmiana maski w bloku Stream Manipulator.....	42
<b>5. Podsumowanie i wnioski.....</b>	<b>43</b>
<b>A. Diagramy .....</b>	<b>45</b>
<b>Bibliografia .....</b>	<b>49</b>

# 1. Wprowadzenie

Trudno wyobrazić sobie dzisiejszy świat bez wykorzystania protokołów pozwalających osiągać multi-gigabitowe prędkości transmisji. Jednym z pierwszych zastosowań jakie przychodzi na myśl jest szybki dostęp do internetu, gdzie korzystając z protokołu Ethernet miliony użytkowników na świecie mogą pobierać duże ilości danych, umożliwiając na przykład oglądanie filmów w wysokiej jakości używając mediów strumieniowych. Transmisja szeregową pozwoliła również na zwielokrotnienie przepustowości w przypadku przemysłowych zastosowań. Jednym z protokołów wykorzystujących tę technologię jest protokół Interlaken, zoptymalizowany pod względem uzyskania dużych prędkości transmisji oraz ograniczenia błędów transmisyjnych, dodatkowo dzięki możliwości korzystania z wielu kanałów pozwalający na dostosowywanie go do własnych wymagań.

## 1.1. Cele pracy

Celem poniższej pracy jest projekt warstwy kodującej protokołu Interlaken, współpracującej z szybkimi układami serializującymi (SerDes). Zaprojektowany układ ma za zadanie nawiązywać połączenie, utrzymywać je oraz udostępniać sygnały diagnostyczne. Projekt powinien być syntezywalny, pozwalając na wykorzystanie go na różnych architekturach układów FPGA, a także w dedykowanych układach scalonych.

## 1.2. Zawartość pracy

Rozdział 1 stanowi wstęp wprowadzający w tematykę pracy, cel i zakres wykonywanych prac.

Rozdział 2 opisuje wykorzystane technologie i protokoły oraz wyjaśnia pojęcia wykorzystywane w późniejszych rozdziałach.

W rozdziale 3 zamieszczony został projekt warstwy kodującej protokołu Interlaken wykonanej w ramach tej pracy. Część ta opisuje budowę wykonanego modułu z podziałem na hierarchię

podmodułów, zaczynając od ogólnej architektury rozwiązania, kolejno opisując logikę rdzenia, a następnie części nadawczej i odbiorczej stworzonego projektu. Następnie zostały opisane poszczególne bloki funkcjonalne zaczynając względem kolejności przepływu danych.

W rozdziale 4 opisany został sposób testowania wykonanego projektu oraz zostały omówione bloki wykorzystane w tym celu.

W rozdziale 5 znajduje się podsumowanie wykonanych prac i jak przebiegały oraz refleksja nad możliwościami rozwoju oraz poprawy projektu.

Oprócz wyżej wymienionych rozdziałów został również dodany załącznik A, zawierający schemat RTL stworzonego IP Core z rdzeniem transceivera oraz diagram blokowy wykonany w środowisku Vivado ukazujący faktyczny widok połączeń wykorzystanych modułów RTL i IP Core.

Na sam koniec została załączona bibliografia z wykazem źródeł wykorzystanych w trakcie pisania tej pracy oraz wykaz skrótów użytych pojęć. W nawiasach kwadratowych obecnych przy zapożyczonych treściach znajdują się odwołania do pozycji w bibliografii.



## 2. Część teoretyczna

Rozdział ten opisuje wykorzystane technologie i protokoły oraz wyjaśnia pojęcia wykorzystywane w późniejszych rozdziałach.

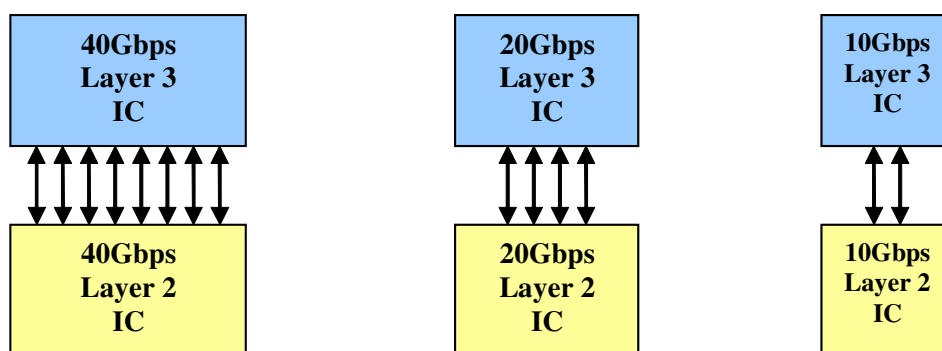
### 2.1. Protokół Interlaken

Protokół Interlaken został stworzony, aby wykorzystać nowoczesne technologie komunikacji szeregowej. Interfejs ten zapewnia duże prędkości transmisji oraz wszechstronność, z możliwością skalowania poprzez konfigurowalną liczbę linii transmisyjnych oraz zwiększoną odpornością na błędy transmisji.[1]

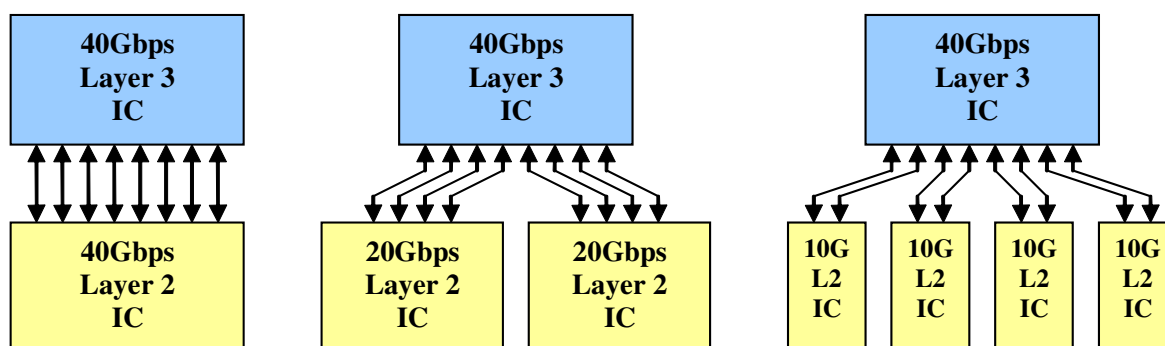
Bazowe funkcjonalności[2]:

- Protokół ten natywnie wspiera do 256 kanałów, lecz może to być rozszerzone do 64K kanałów.
- Wykorzystanie ramek *Meta Frame* o konfigurowalnej długości, pozwalających na synchronizację wielu linii, synchronizację scramblera, kompensację zegara oraz sprawdzenie stanu linii.
- Niezależność protokołu od ilości wykorzystanych linii oraz prędkości transmisji.
- Niezależna od kanału funkcjonalność *Flow Control*.
- Kodowanie 64B/67B oraz wykorzystanie scramblera synchronicznego celem uniknięcia powielania błędów.
- Skalowanie przepustowości wraz z ilością linii transmisyjnych.

Na rysunkach 2.1 i 2.2 zostały przedstawione aspekty skalowalności oraz elastyczności protokołu Interlaken w różnych zastosowaniach.



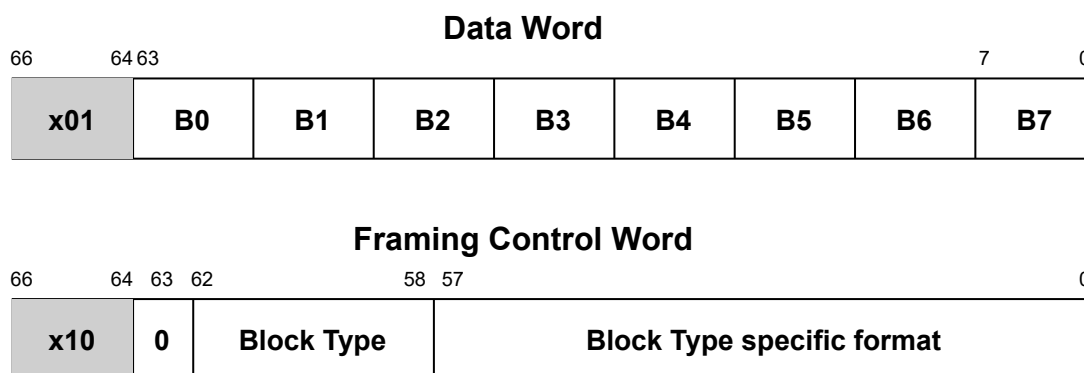
Rys. 2.1. Skalowanie przepustowości wraz ze wzrostem ilości linii[1]



Rys. 2.2. Elastyczność wykorzystania protokołu[1]

### 2.1.1. Format słów

W przypadku warstwy kodującej protokołu Interlaken możemy rozróżnić 2 rodzaje słów, jak zostało to przedstawione na rysunku 2.3. Słowa te są rozróżniane za pomocą bitów 64 i 65, w przypadku *Data Word* są one równe 01, natomiast dla *Framing Control Word* są ustawione jako 10.



Rys. 2.3. Format słów

*Data Word* są wykorzystywane do transmisji danych i składają się z 8 bajtów. W przypadku *Framing Control Word* zależnie od zawartości pola *Block Type* mogą być transmitowane 4 rodzaje słów[2]:

**Synchronization** – Block Type: 011110 - Oznacza początek ramki.

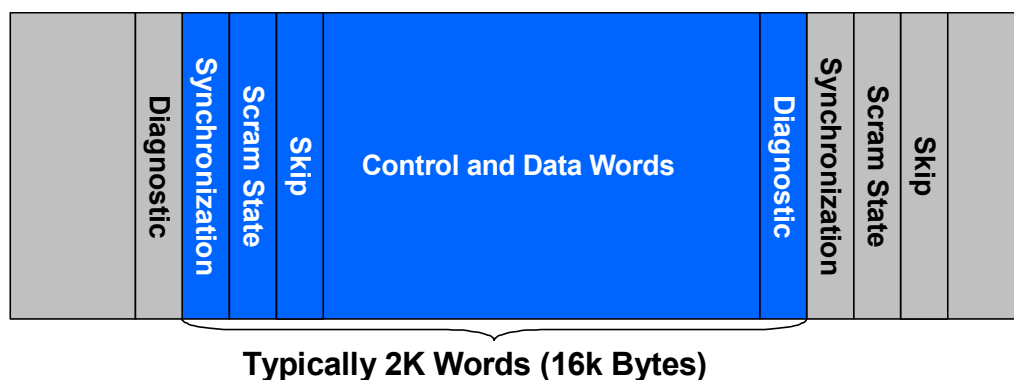
**Scrambler State** – Block Type: 001010 - Używane do synchronizacji scramblera.

**Skip** – Block Type: 000111 - Używane do kompensacji zegara.

**Diagnostic** – Block Type: 011001 - Służy do weryfikacji otrzymanych danych pod względem błędów oraz opcjonalnie może posiadać status.

### 2.1.2. Meta Frame

W czasie transmisji przesyłane dane są enkapsulowane za pomocą ramki Meta Frame składającej się ze słów przedstawionych na rysunku 2.4. Ramki takie przesyłane są okresowo i składają się z równej ilości słów, będącej konfigurowalnym parametrem nazywanym Meta Frame Length. Parametr ten reprezentuje sumę wszystkich transmitowanych *Framing Control Word* wraz z payloadem. Ramka Meta Frame jest ortogonalna względem transmisji danych i może nastąpić w dowolnym momencie transmisji.[2]



Rys. 2.4. Format ramki Meta Frame[1]

### 2.1.3. Niwelacja offsetu DC

Jednym z wyzwań w szybkich szeregowych interfejsach jest zachowanie balansu DC, czyli jednakowej ilości transmitowanych symboli 0 oraz 1, aby odbiornik był w stanie poprawnie zdekodować strumień danych. Niektóre kody takie jak 8B/10B zachowują balans DC na krótkim przedziale, wynoszącym kilka symboli. Jednak inne kodowania jak 64B/66B zachowują balans

DC jedynie statystycznie, wykorzystując scrambling danych. W przypadku takich kodów brak zachowania balansu może powodować błędy transmisyjne w bardziej wymagających przypadkach.

Z racji że protokół Interlaken został zaprojektowany do wykorzystania w wielu różnych aplikacjach, dodatkowy bit oznaczający inwersję danych (stąd kodowanie 64B/67B) pozwala utrzymać lepszą kontrolę nad balansem DC, dając możliwość na wykorzystywanie go również w tych bardziej wymagających przypadkach.[1]

## 2.2. Wykorzystane bloki logiczne

### 2.2.1. IP Core

IP Core z angielskiego (Intellectual Property), są to prekonfigurowalne funkcje logiczne, które pozwalają na szybkie wykorzystanie istniejących już i przetestowanych funkcjonalności. Często są one dostarczane przez producenta FPGA, co sprawia że mogą być zoptymalizowane pod wykorzystywany układ. IP Core można podzielić na 2 kategorie [3]:

**Soft Cores** – najczęściej są to synteżowalne bloki napisane w językach opisu sprzętu takich jak VHDL i Verilog.

**Hard Cores** – nie mogą być modyfikowane z racji, że jest to często niskopoziomowy opis sprzętu przygotowany dla danego układu.

### 2.2.2. Transceiver

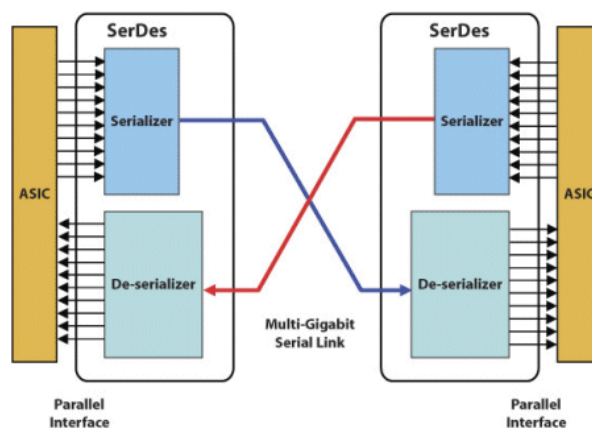
Transceivery są to układy będące kombinacją nadajnika i odbiornika w jednym bloku za pośrednictwem medium transmisyjnego. Nazwa *transceiver* pochodzi od połączenia słów *transmitter* i *receiver*. Transmisja jest najczęściej nawiązywana za pośrednictwem fal radiowych, połączenia przewodowego lub światłowodu.[4]

W przypadku układów takich jak *Kintex-7* wykorzystywane są multi gigabitowe transceivery, w skrócie MGT. Są to bloki SerDes operujące na prędkościach powyżej 1 Gigabita/s.[5]

### 2.2.3. SerDes

W magistralach szeregowych urządzenia zwane SerDes są wykorzystywane do transmisji i odbierania danych poprzez połączenie szeregowe. SerDesy są to w praktyce szeregowe transceivery, które konwertują równoległe dane w strumień danych szeregowych po stronie nadajnika oraz konwertują szeregowe dane z powrotem do postaci równoległej po stronie odbiornika. Z

powodu braku obecności osobnego sygnału zegarowego przestają ograniczać prędkość transmisji parametry takie jak *Time Skew* oraz *Setup/Hold Time*. W wyniku tego transmisja szeregową może operować na dużo większych prędkościach przesyłu danych. Wadą takiego rozwiązania jest konieczność użycia bardziej skomplikowanych układów w odbiorniku celem odzyskania sygnału zegarowego.[6]



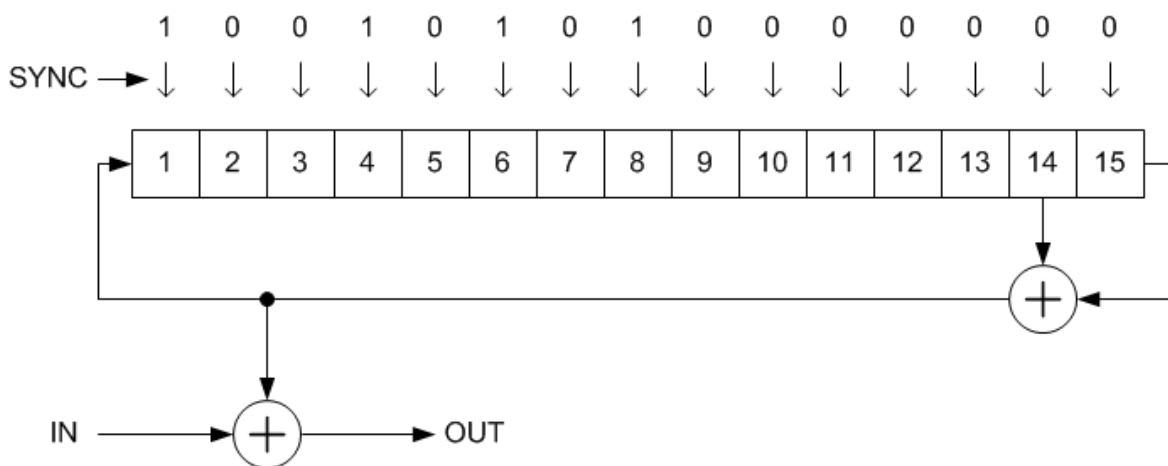
Rys. 2.5. Transmisja z wykorzystaniem SerDes[6]

## 2.2.4. Scrambler

Proces scrambling jest wykorzystywany, aby zapewnić odpowiednią ilość zmian wysyłanych symboli, aby odbiornik mógł odzyskać sygnał zegarowy. W nowoczesnych systemach cyfrowych często zdarza się, że są transmitowane długie ciągi zer i jedynek, co jest zjawiskiem niepożądanym, ponieważ utrudnia ono odzyskanie danych przez odbiornik. W celu uniknięcia tego problemu stosuje się scrambling, które pozwala wprowadzić losowość do nadawanych danych.[7]

### 2.2.4.1. Addytywny (synchroniczny) scrambler

Jak zostało przedstawione na rysunku 2.6, ten rodzaj scramblera najczęściej wykorzystuje LFSR celem generowania pseudo-losowej sekwencji bitów (PRBS). Wartości generowane przez LFSR są zależne od tego które pozycje bitów wpływają na następny stan układu, realizując dany wielomian. Tak uzyskiwany stan wewnętrzny jest używany celem przekształcenia danych wejściowych za pomocą dodawania modulo 2. Zaletą takiego rozwiązania jest to, że przekształcane dane wejściowe nie wpływają na stan wewnętrzny scramblera, co pozwala uniknąć multiplikowania błędów. Do wad natomiast można zaliczyć konieczność synchronizacji stanu wewnętrznego takich układów w nadajniku i odbiorniku, najczęściej jest to wykonywane za pomocą okresowo przesyłanych słów synchronizujących.

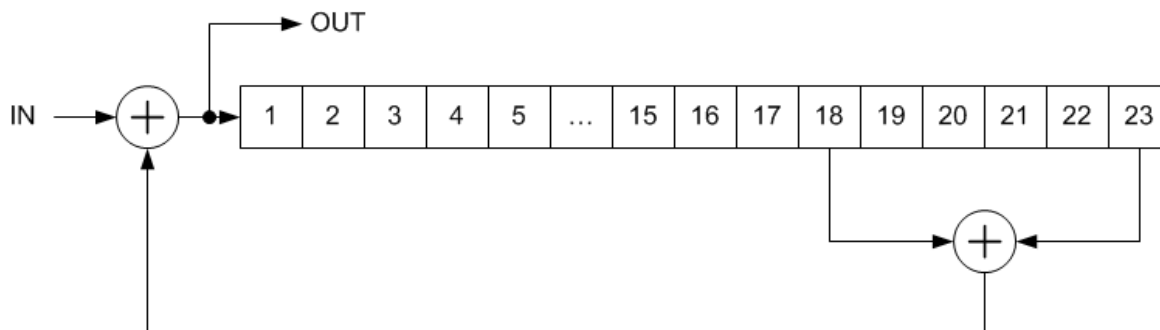


Rys. 2.6. Przykład scramblera addytywnego[8]

W przypadku protokołu Interlaken wykorzystywane są niezależne scramblery addytywne na każdej linii, odbiornik używa okresowo przesyłanego stanu scramblera, celem synchronizacji descramblera i później descramblowania odbieranych słów.[1]

#### 2.2.4.2. Multiplikatywny (samosynchronizujący się) scrambler

Scrambler multiplikatywny zgodnie ze swoją nazwą wykonuje mnożenie danych wejściowych z funkcją przejścia scramblera. Zaletą takiego rozwiązania jest brak konieczności synchronizacji takich bloków w nadajniku i odbiorniku, z tego też powodu są one określane jako samosynchronizujące. Niestety wpływ danych wejściowych na stan wewnętrzny takiego bloku wprowadza możliwość multiplikacji błędów z racji istniejącego sprzężenia zwrotnego jak jest to widoczne na rysunku 2.7. Wystąpienie jedno-bitowego błędu na wejściu spowoduje  $x$  błędów na wyjściu, gdzie  $x$  jest ilością bitów używanych w sprzężeniu zwrotnym, nazywanych *feedback taps*.

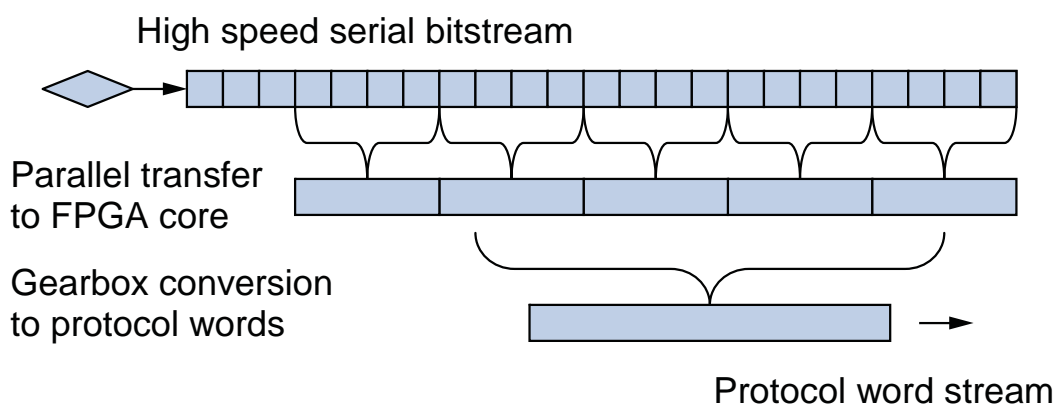


Rys. 2.7. Przykład scramblera multiplikatywnego[8]

### 2.2.5. Gearbox

Celem wykorzystywania gearboxów jest zaadaptowanie transmisji szeregowej do innej szerokości słowa. W przypadku układu *Kintex-7* obsługiwane przez transceiver szerokości słów to 20, 40 lub 80 bitów. Z kolei protokoły takie jak Ethernet lub Interlaken wymagają szerokości słowa 66 lub 67 bitów.

Jednym podejściem do rozwiązania tego problemu może być wykorzystanie kolejki FIFO z dwoma zegarami, których częstotliwości powinny odzwierciedlać stosunek szerokości słów. Rozwiązanie ma to jednak pewne wady, główną z nich są ograniczone zasoby zegarowe w FPGA, co może wymagać stosowania droższych układów. Drugim problemem może być skomplikowanie układu przez dodawanie kolejnych domen zegarowych. Dodatkowo kolejki FIFO dostarczane przez w tym wypadku Xilinx, najczęściej obsługują szerokości słów będących jedynie potęgami dwójki lub wielokrotnościami dziesiątek nie pozwalając na konfigurację konwersji z 67 bitów na tę obsługiwaną przez transceiver. Lepszym rozwiązaniem jest używanie sygnału *Valid* do kontroli przepływu danych wraz z buforem, który precyzyjnie konwertuje te dane wykorzystując jeden sygnał zegarowy i takie rozwiązanie zostało wykorzystane w tym projekcie.[9]



Rys. 2.8. Odbieranie danych szeregowych[9]





## 3. Implementacja układu

Rozdział ten został poświęcony na zaprezentowanie szczegółów zaimplementowanego rozwiązania, opisuje on budowę wykonanego modułu z podziałem na hierarchię podmodułów, zaczynając od ogólnej architektury rozwiązania, kolejno opisując logikę rdzenia, a następnie części nadawczej i odbiorczej stworzonego projektu. Następnie zostały opisane poszczególne bloki funkcjonalne zaczynając względem kolejności przepływu danych. Ostatnią częścią tego rozdziału jest omówienie testowania wykonanego projektu i komunikacji z nim.

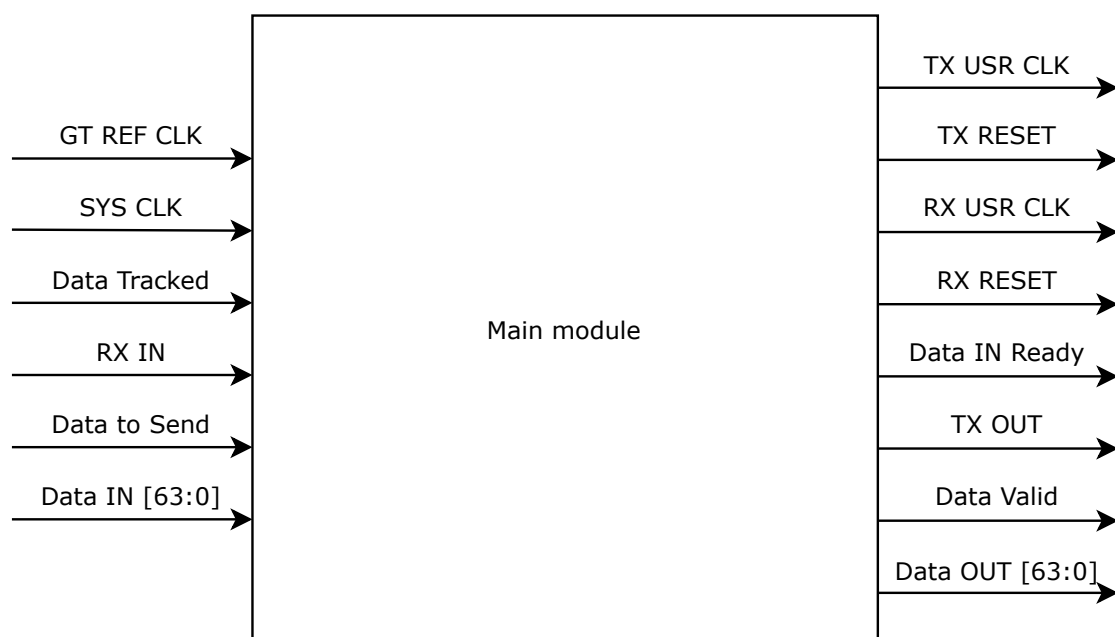
### 3.1. Architektura i wykorzystane narzędzia

Architektura zaimplementowanej warstwy została podzielona na podmoduły i bloki w celu ułatwienia późniejszej implementacji. Wynika to z faktu, że narzędzie wykorzystywane do projektowania i syntezy *Xilinx Vivado*, pozwala na tworzenie schematów blokowych(Block Design) i graficzne łączenie stworzonych modułów RTL oraz IP Core. Funkcjonalność ta umożliwia uniknięcie zarządzania dużą ilością połączeń oraz ułatwia przedstawienie w graficzny sposób implementowanej funkcjonalności. Oczywiście niesie to za sobą minusy takie, jak ograniczenie w swobodzie tworzenia kodu lub w dużym stopniu utrudnienie współpracy z narzędziami takimi jak *Git*. Wynika to z powodu automatycznego generowania plików w przypadku korzystania ze schematów blokowych w tym narzędziu.

W załączniku A został załączony diagram stworzony z pomocą tego narzędzia, niestety z powodu automatycznego generowania połączeń przez program, jego czytelność może być w pewnym stopniu ograniczona.

## 3.2. Projekt warstwy kodującej

Będąc tematem tej pracy warstwa kodująca protokołu Interlaken, została umieszczona w bloku nazwanym modulem głównym. Jako pierwszy krok warto zwrócić uwagę na to jakie połączenia będą potrzebne w celu nawiązania komunikacji oraz jakie jest ich zastosowanie jak przedstawiono na rysunku 3.1 oraz w tabeli 3.1.



**Rys. 3.1.** Wyprowadzenia modułu głównego

Przedstawiony w wcześniej wspomnianej tabeli opis wyprowadzeń zawiera jedynie wymagane minimum. Dodatkowe dane diagnostyczne były zbierane za pomocą bloków ILA opisanych w rozdziale 4.2.

Tabela 3.1. Opis wyprowadzeń modułu głównego

Nazwa pinu	Opis
GT REF CLK	Zegar referencyjny transceivera <b>125MHz</b>
SYS CLK	Zegar systemowy <b>100MHz</b>
Data Tracked	Sygnał wejściowy maszyny stanów transceivera określający czy odbierane dane są poprawne
RX IN	Wejście szeregowe transceivera
Data to Send	Wejście określające czy dane na wejściu <b>Data IN</b> powinny zostać wysłane
Data IN	64-bitowe dane do wysłania
TX USR CLK	Zegar części nadajnika transceivera
TX RESET	Reset części nadajnika transceivera
RX USR CLK	Zegar części odbiornika transceivera
RX RESET	Reset części odbiornika transceivera
Data IN Ready	Wyjście sygnalizujące że kolejne dane mogą zostać podane na wejście <b>Data IN</b>
TX OUT	Wyjście szeregowe transceivera
Data Valid	Wyjście określające czy dane na wyjściu <b>Data OUT</b> powinny być odbierane
Data OUT	64-bitowe odebrane dane

### 3.2.1. Hierarchia modułu głównego

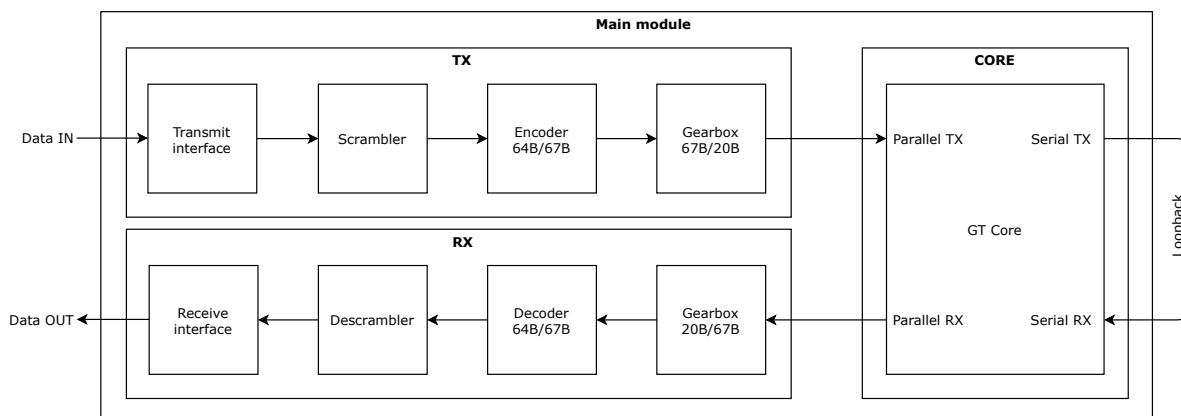
Analizując budowę modułu głównego ze względu na funkcjonalność możemy wyodrębnić 3 podmoduły jak przedstawia to rysunek 3.2:

**TX** – podmoduł formatujący dane przed przesłaniem ich do transceivera

**RX** – podmoduł formatujący dane po odebraniu ich od transceivera

**CORE** – podmoduł zawierający IP Core z transceiverem oraz blok VIO obsługujący sygnały diagnostyczne

Diagram przedstawiony na rysunku 3.2 prezentuje ogólne połączenie utworzonych bloków, których szczegóły implementacji będą podane w kolejnych podrozdziałach.



**Rys. 3.2.** Przepływ danych w module głównym

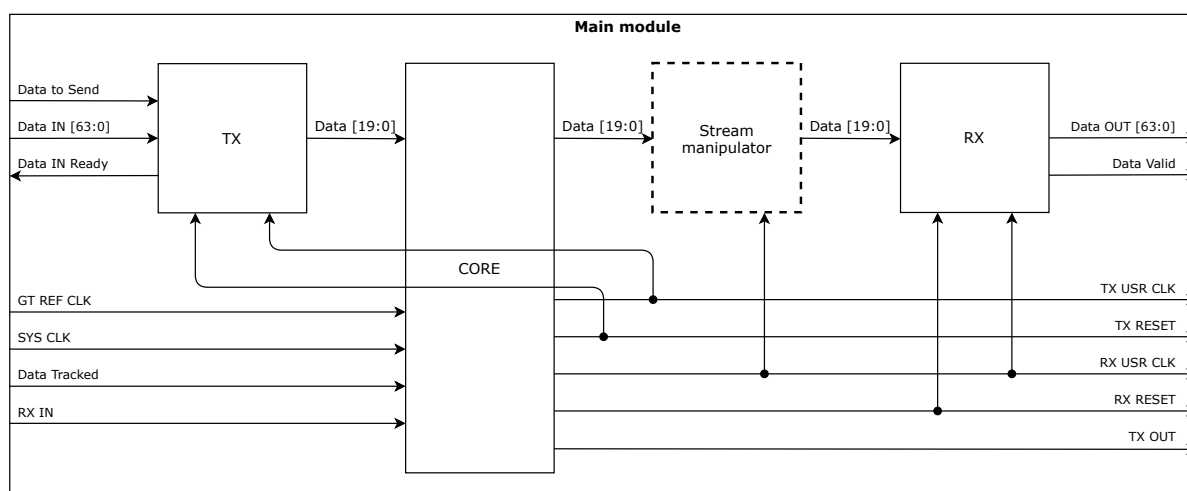
**Transmit/Receive Interface** – odpowiadają za kontrolę przepływu danych

**Scrambler/Descrambler** – odpowiadają za uzyskanie odpowiedniej ilości przejść w sygnale za pomocą scamblowania danych, aby umożliwić odzyskanie sygnału zegarowego przez odbiornik

**Encoder/Decoder** – odpowiadają za wykrywanie 67-bitowych słów spośród otrzymywanych danych i formatowanie ich zgodnie z kodowaniem 64B/67B

**Gearbox** – odpowiadają za zaadaptowanie transmisji szeregowej do innej szerokości słowa

**Transceiver core** – odpowiada za transmisję i odbieranie danych

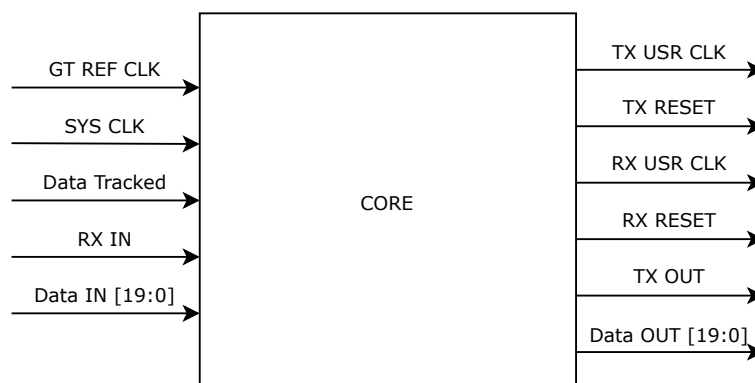


**Rys. 3.3.** Szczegóły połączeń wewnętrznej hierarchii

Z racji że rysunek 3.2 przedstawia jedynie ogólnie zarys jak wyglądają wewnętrzne połączenia między poszczególnymi podmodułami, powstał przedstawiony na rysunku 3.3 diagram,

ukazujący szczegółowo połączenia pomiędzy podmodułami. Dodatkowo został przedstawiony na nim blok “Stream manipulator” opisany w rozdziale 4.5, wykorzystywany do testowania wpływu wprowadzonych błędów danych na działanie modułu.

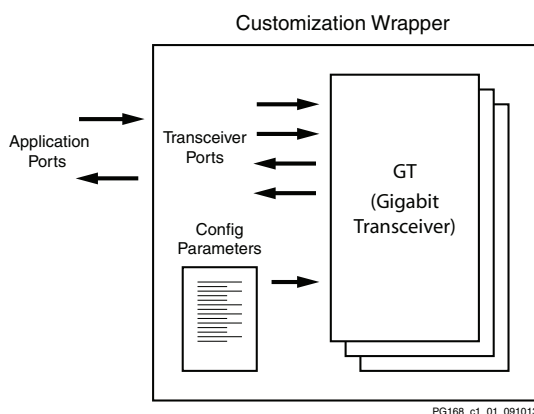
### 3.2.2. Podmoduł Core



Rys. 3.4. Wyprowadzenia podmodułu Core

Jak można zauważyć na rysunku 3.3 wiele połączeń tego podmodułu jest współdzielone z modułem głównym i opisującą go tabelą 3.1. Główną różnicą są 20-bitowe wejście Data IN oraz wyjście Data OUT, wynikające z wewnętrznej szerokości słowa ustawionej w transceiverze. Wewnątrz podmodułu znajduje się instancja stworzonego IP Core’a o nazwie *GT Core*. Dodatkowo znalazł się tam blok VIO pozwalający na wykonanie soft resetu.

#### 3.2.2.1. GT Core



Rys. 3.5. Konfigurator “7 Series FPGAs Transceivers Wizard”[10]

Z powodu ograniczeń środowiska Vivado bezpośrednie osadzenie wygenerowanego rdzenia transceivera w Block Designie nie jest możliwe, wymusiło to stworzenie własnego IP Core'a nazywanego *GT Core*. Stanowi on wrapper dla później opisanego rdzenia transceivera, pozwalając na używanie go w Block Designie. Mimo wprowadzonego utrudnienia rozwiązanie to ma również plusy, jak na przykład ograniczenie ilości wyprowadzeń zapewniając jedynie te konieczne. W załączniku został dodany rysunek A.1 z widokiem RTL modułu.

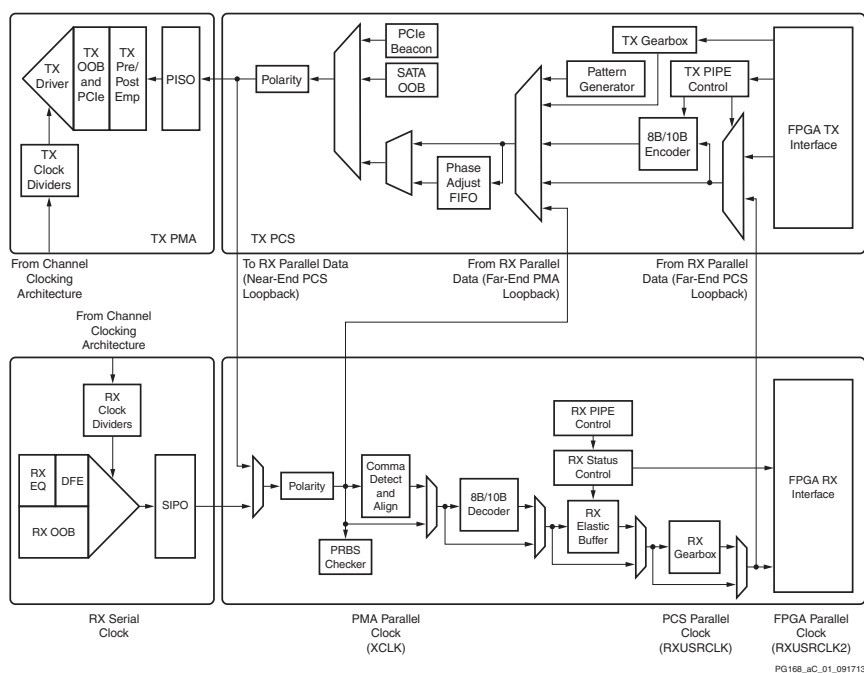
W celu stworzenia instancji multi-gigabitowego transceivera obecnego na płycie *Digilent Genesys 2* należało wykorzystać udostępniony przez producenta konfiguratora *7 Series FPGAs Transceivers Wizard*. Pierwszym z kroków było wybranie jednej z 16 dostępnych linii obsługiwanych przez transceiver, z racji że do przeprowadzenia testów miały zostać wykorzystane obecne na płycie porty Mini DisplayPort wymagane było posłużenie się schematem ideowym[11] w celu wybrania odpowiedniej pary linii. W przypadku tego projektu została wybrana para oznaczona jako *GTX\_X0Y12* będąca pierwszą z 4 par wykorzystywanych przez interfejs DisplayPort. Kolejnym krokiem wybraniu odpowiedniej pary jest konfiguracja odpowiednich parametrów. W przypadku tego projektu zostały one wybrane jak jest to widoczne w tabeli 3.2.

**Tabela 3.2.** Skonfigurowane parametry transceivera

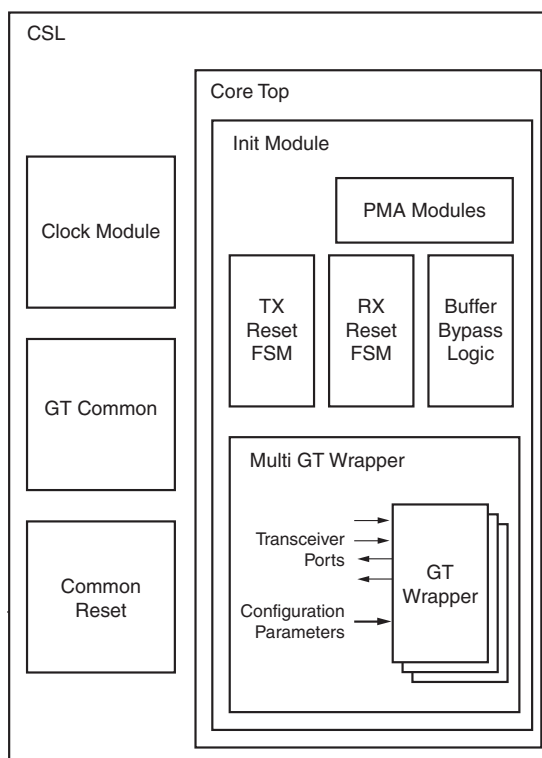
Parametr	Wartość
Line Rate(Gbps)	1.25
System clock(MHz)	100
Reference clock(MHz)	125
TX USR CLK(MHz)	62.5
RX USR CLK(MHz)	62.5
Encoding	None
Decoding	None
Internal Data width	20
External Data width	20

Wybrane zegary systemowy i referencyjny transceivera zostały ustawione na domyślne wartości. Mimo maksymalnej możliwej przepustowości na poziomie  $10.3125\text{Gbps}$ , została ona ustawiona na poziomie  $1.25\text{Gbps}$ , celem zapewnienia stabilności działania w razie niskiej jakości zastosowanego kabla DisplayPort. Kolejno z racji zastosowanych gearbox'ów szerokość słowa została ustawiona na 20. Dodatkowo zostały wyłączone wszystkie obecne w module

funkcjonalności enkodowania i dekodowania, jako że celem tej pracy było własnoręczne stworzenie takich modułów.



Rys. 3.6. Struktura wewnętrzna kanału transceivera[10]



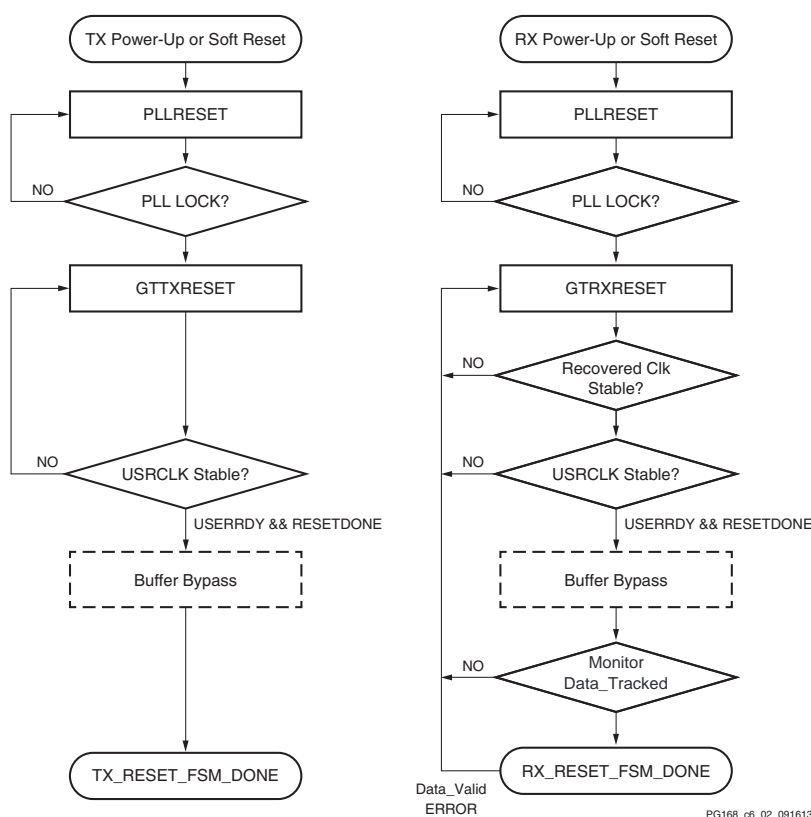
Rys. 3.7. Struktura generowanego rdzenia transceivera[10]

Na rysunku 3.6 została dodatkowo przedstawiona wewnętrzna struktura kanału transceivera wraz z obecnymi funkcjonalnościami. Jak można zauważyć w standardowym wykorzystaniu takiego bloku można użyć zaimplementowane już gearboxy oraz dekodery/enkodery, bez dodatkowego nakładu pracy z naszej strony.

Finalnie po przeprowadzonej jak przedstawiono powyżej konfiguracji i kliknięciu “OK” generowany jest rdzeń transceivera, którego struktura została przedstawiona na rysunku 3.7.

Jedną z części na którą warto zwrócić uwagę są maszyny stanów sterujące inicjalizacją części odbiorczej i nadawczej transceivera. W celu uniknięcia kolizji oznaczeń na rysunku 3.8 sygnał *Data\_Valid* został zmieniony na *Data\_Tracked*.

W przypadku części TX praktycznie jak szybko jak zegary się ustabilizują możliwe jest rozpoczęcie nadawania. Część RX natomiast wymaga kilku dodatkowych kroków jak widać na wcześniej wspomnianym diagramie, należy do nich odzyskanie zegara z otrzymanych danych oraz monitorowanie linii *Data\_Tracked* obecnej na rysunku 3.4. Linia ta powinna być sterowana z zewnątrz w zależności od tego czy odbierane dane są poprawne. Jeżeli przy inicjalizacji lub w trakcie działania linia ta będzie w stanie nieaktywnym przez dłużej niż  $100\mu s$  oraz port *DONT\_RESET\_ON\_DATA\_ERROR* będzie ustawiony na 0, transceiver zostanie zresetowany.

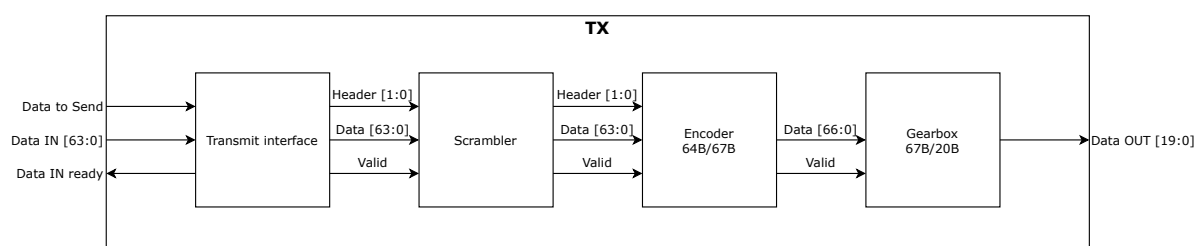


Rys. 3.8. Generowana maszyna stanów[10]



### 3.2.3. Podmoduł TX

Jak widać na diagramie podmodułu TX przedstawionym na rysunku 3.9, podmoduł ten jest odpowiedzialny za kontrolę przepływu danych jak i ich formatowanie zgodnie z kodowaniem 64B/67B oraz przesyłanie odpowiednich pakietów *Control Words* jak przedstawiono w rozdziale 2.1.2. Cały podmoduł kontrolowany jest jednym sygnałem zegarowym, dane wejściowe pobierane są w 64 bitowych pakietach, zgodnie z sygnałem *Data IN ready* sygnalizującym że kolejny pakiet może zostać podany na wejście, jednocześnie na każdy cykl zegara dane są opróżniane po 20 bitów do wyjścia *Data OUT*. Dodatkowo w zależności od sygnału *Data to Send*, jeśli jest on w stanie aktywnym zostaną wysłane dane z wejścia *Data IN*, w innym wypadku będą transmitowane *Idle Words*.



Rys. 3.9. Diagram podmodułu TX

#### 3.2.3.1. Transmit Interface

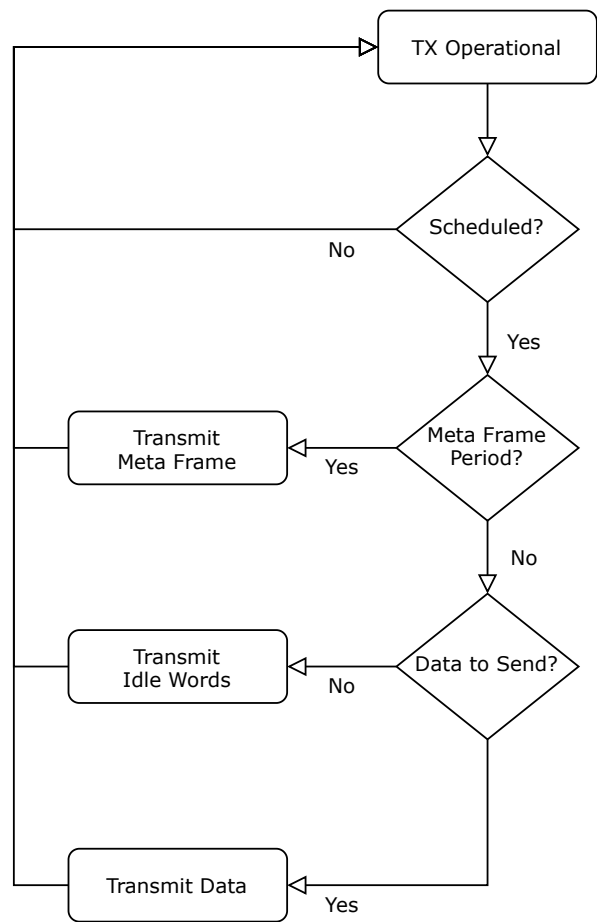
Główną funkcją bloku *Transmit Interface* jest przede wszystkim kontrola przepływu danych, jak również decydowanie kiedy powinny być transmitowane *Control Words*, a kiedy *Data Words* zgodnie z budową ramki *Meta Frame*.

Z powodu zastosowania jednego sygnału zegarowego w podmodule TX oraz ustawieniu szerokości danych w transceiverze na 20 bitów, zgodnie z tabelą 3.2, wymagało to specyficznego harmonogramu przesyłania danych. Aby nie nastąpiło przepełnienie bufora gearboxu dane muszą być transmitowane dokładne 20 razy w ciągu 67 cykli, zostało to uzyskane za pomocą wewnętrznego sygnału *scheduled* predefiniowanego jako:

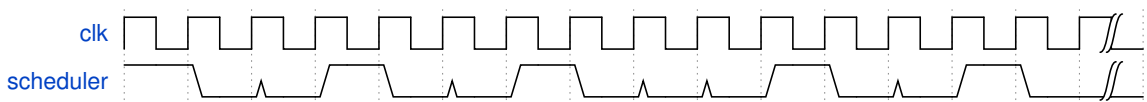
```
67'b10010010001001001000100100100010010010001001001001000100100100100010010010001001000
```

Sygnał ten powtarzany jest okresowo, na każdym cyklu zegara podając kolejny bit z 67 bitowego ciągu, jak to częściowo przedstawiono na rysunku 3.11. W celu koordynacji działania pozostałych bloków sygnał ten jest podawany z odpowiednim opóźnieniem na wyjście *Valid*.

Dodatkowo z powodu konieczności okresowego transmitowania *Control Words* został dodany licznik przesłanych ramek o maksymalnej wartości *Meta Frame Length*, która została

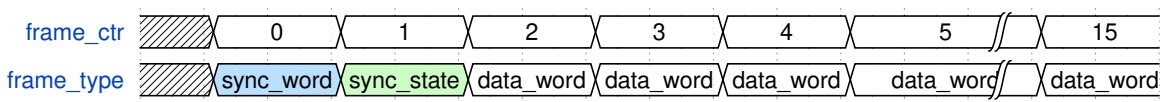


Rys. 3.10. Maszyna stanów bloku Transmit Interface



Rys. 3.11. Harmonogram transmisji danych

ustawiona w tym przypadku na 16. Z racji że ten projekt implementuje jedynie warstwę kodującą, wymagane jest jedynie dodatkowe przesyłanie ramek pozwalających na synchronizację scramblera, jak jest to przedstawione na rysunku 3.12.



Rys. 3.12. Kolejność słów w Meta Frame

Wszystkie predefiniowane słowa używane przez ten podmoduł zostały przedstawione w tabeli 3.3. Warto zwrócić uwagę, że w przypadku słowa *sync\_state* pole w którym powinien

znajdować się stan scramblera jest wypełnione zerami, zostanie ono wypełnione właściwą wartością dopiero po dotarciu do kolejnych bloków. Dodatkowo w przypadku gdy nie chcemy transmitować danych, została zaimplementowana funkcjonalność by za pomocą wejścia *Data to Send* móc kontrolować czy powinny być wysyłane słowa danych z wejścia *Data IN*, czy puste słowa - *Idle Words*. Ponownie z racji implementacji jedynie warstwy kodującej zostało ono predefiniowane jak jest to widoczne w wcześniej wspomnianej tabeli.

**Tabela 3.3.** Predefiniowane słowa w bloku Transmit Interface

Słowo	Wartość
sync_word	0x78f678f678f678f6
sync_state	0x2800000000000000
idle_word	0xaaaaaaaaaaaaaaaa

### 3.2.3.2. Scrambler

Po wyjaśnieniu zagadnienia scramblingowania danych w rozdziale 2.2.4, możemy teraz omówić jego praktyczną implementację. W listingu 3.1 został przedstawiony kod służący do scramblingowania danych o szerokości równej *DATA\_WIDTH*, wykorzystujący wielomian  $x^{58} + x^{39} + 1$ . W przypadku protokołu interlaken jedynie 64 dolne bity są scramblingowane, 3 kolejne bity synchronizujące dodawane przez blok *Encoder 64B/67B* są przesyłane bez zmian.

```
always @(scrambler, DATA_IN)
begin
    poly = scrambler;
    for (i=0; i<=(DATA_WIDTH-1); i=i+1)
    begin
        xorBit = DATA_IN[i] ^ poly[38] ^ poly[57];
        poly = {poly[56:0], xorBit};
        tempData[i] = xorBit;
    end
end
end
```

**Listing 3.1.** Logika scramblingującą

W trakcie działania tego bloku jego głównym zadaniem jest ustawienia scramblingowanych danych z rejestru *tempData* na wyjście oraz aktualizacja stanu wewnątrzengo w rejestrze *scrambler* nową wartością *poly*. Jedynym wyjątkiem jest gdy zostaną wykryte słowa *sync\_word* lub *sync\_state*, nadawane przez blok *Transmit Interface*. W przypadku słowa *sync\_word* jest ono przekazywane bez zmian na wyjście, aby mogło zostać ono wykryte przez descrambler, stan wewnętrzny scramblera przy transmisji tego słowa nie jest aktualizowany. W przypadku słowa

*sync\_state* scrambler kopiuje w miejsce 58 dolnych bitów swój stan wewnętrzny jak przedstawiono w tabeli 3.4 i przekazuje na wyjście również bez aktualizacji stanu wewnętrznego.

**Tabela 3.4.** Słowa synchronizujące stan scramblera

	[65:64]	[63:58]	[57:0]
<i>sync_word</i>	b10	b011110	h0F678F678F678F6
<i>sync_state</i>	b10	b001010	Scrambler State

### 3.2.3.3. Encoder 64B/67B

W przypadku bloku enkodera 64B/67B jego zadaniem jest dodawanie bitów synchronizujących do transmitowanych słów, z racji specyfiki tego kodowania musi on jednak dodatkowo kontrolować dysproporcję w ilości transmitowanych zer i jedynek, celem uniknięcia wprowadzania offsetu DC. W listingu 3.2 przedstawiono sposób obliczania tej dysproporcji.

```
always @*
begin
    temp = 0;
    for (i=0; i<=(64-1); i=i+1)
    begin
        temp = temp + DATA_IN[i];
    end
    word_disparity = (temp<<1) - 64;
end
```

**Listing 3.2.** Obliczanie dysproporcji ilość zer i jedynek w słowie.

Zmienna *word\_disparity* została zdefiniowana jako rejestr ze znakiem, dysproporcja danego słowa obliczana jest jako  $(2 \cdot ones) - 64$ . Tak obliczona dysproporcja słowa jest porównywana z ogólną dysproporcją i w zależności od ich znaków różni się działanie algorytmu:

**taki sam znak** – bit 66 jest ustawiony na 1, bity [63:0] są odwrócone, ogólna dysproporcja jest aktualizowana jako  $disparity = disparity - word\_disparity + 1$

**różny znak** – bit 66 jest ustawiony na 0, bity [63:0] są nieodwrócone, ogólna dysproporcja jest aktualizowana jako  $disparity = disparity + word\_disparity - 1$

Do ogólnej dysproporcji wliczany jest również bit 66, co wpływa na dodanie lub odjęcie jedynki od niej podczas aktualizacji jej wartości. W tabeli 3.5 przedstawiono wszystkie możliwe stany bitów synchronizujących.

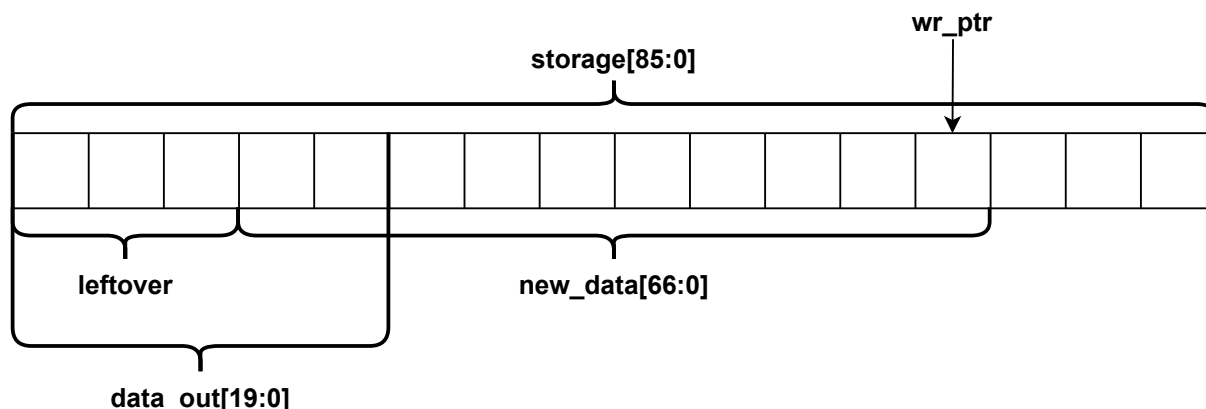
Tabela 3.5. Bity synchronizujące

Bity [66:64]	Interpretacja
001	Data Word, bity [63:0] nieodwrócone
010	Control Word, bity [63:0] nieodwrócone
101	Data Word, bity [63:0] odwrócone
110	Control Word, bity [63:0] odwrócone
Pozostałe	Nielegalny stan

### 3.2.3.4. Gearbox 67B/20B

W przypadku gearboxa 67B/20B została wykorzystana implementacja pochodząca z książki [9], gdzie również dostępny jest cały kod źródłowy. Cel stosowania gearboxów został opisany w rozdziale 2.2.5.

Aby pozwolić na poprawne działanie tego układu konieczne było transmitowanie 67 bitowych słów określonym tempem, co zostało uzyskane implementując specjalny harmonogram jak to zostało opisane w rozdziale 3.2.3.1, transmitując słowa 20 razy w ciągu 67 cykli zegara.



Rys. 3.13. Buforowanie transmitowanych danych

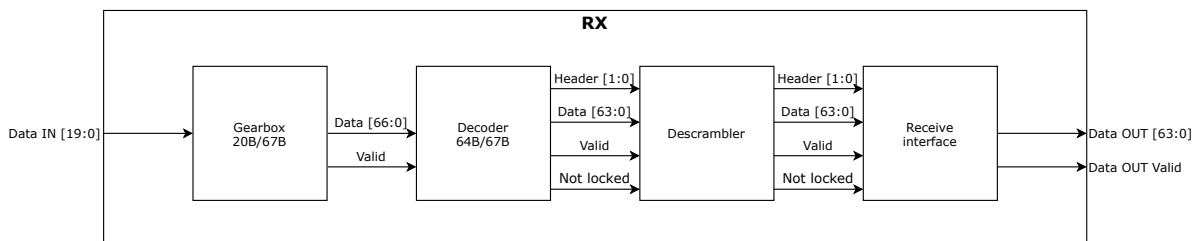
Rozmiar bufora wewnętrznego *storage* dobrany jest tak, aby pozwolić na obsłużenie najgorszego przypadku, tj. pozostałych 19 bitów + 67 przychodzących, co daje 86 bitów. Na rysunku 3.13 przedstawiono jak wygląda wpisywanie nowych danych do bufora.

Blok ten w każdym cyklu zegara przesuwa rejestr *storage* o 20 bitów w lewo, co powoduje podanie na wyjście kolejnego 20 bitowego słowa. W momencie gdy wejście *Valid* jest w stanie aktywny, po przesunięciu rejestru następuje dodatkowe wpisanie nowego 67 bitowego słowa danych zaczynając od bitu na który wskazuje zmienna *wr\_ptr*. Wtedy też jest zapisywana nowa

wartość tej zmiennej jako  $wr\_ptr = (wr\_ptr + 13) \bmod 20$ , co pozwala na wyrównanie nowej danej zaraz za bitami, które pozostały w buforze.

### 3.2.4. Podmoduł RX

W przypadku podmodułu RX, którego diagram przedstawiono na rysunku 3.14, podobnie jak podmoduł TX opisany w rozdziale 3.2.3 jest kontrolowany jednym sygnałem zegarowym. W każdym cyklu zegara pobierane są 20 bitowe pakiety pochodzące z transceivera, w momencie gdy zbierze się odpowiednia ilość bitów, pakiet jest przetwarzany przez kolejne bloki podmodułu. Na wyjście *Data OUT* przekazywane są 64 bitowe pakiety danych, sygnał *Data OUT Valid* sygnalizujący że dane mogą zostać odczytane.



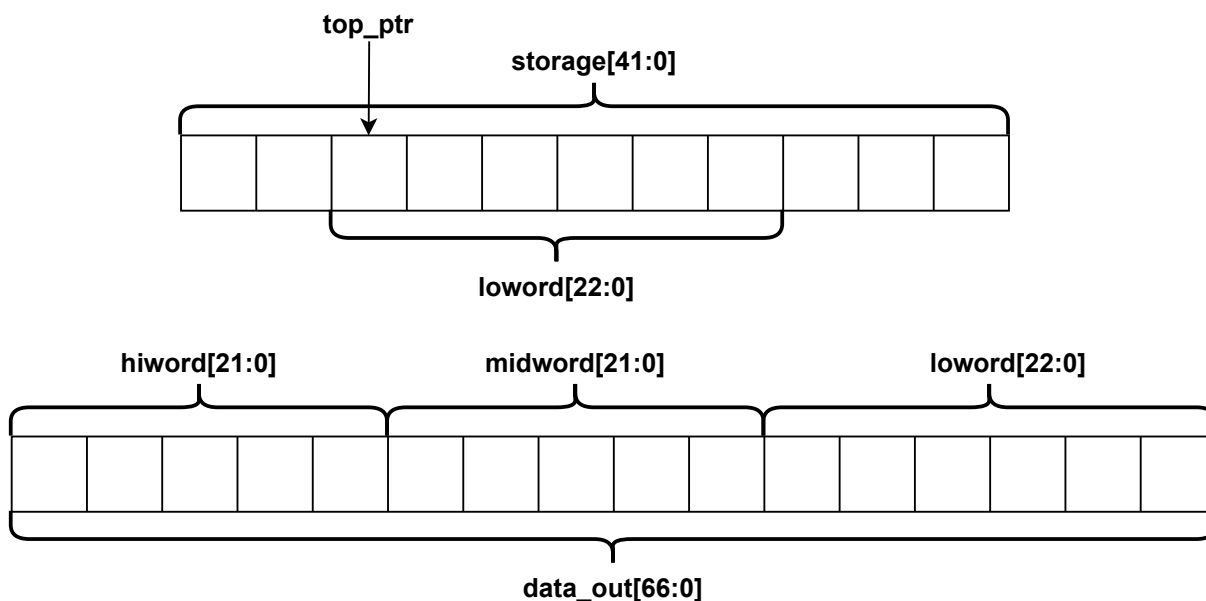
Rys. 3.14. Diagram podmodułu RX

#### 3.2.4.1. Gearbox 20B/67B

Gearbox części odbiorczej podobnie jak w części nadawczej została wykorzystana implementacja pochodząca z książki [9]. W przeciwieństwie jednak do części nadawczej ten blok nie wymaga zewnętrznych bloków sterujących, ponieważ sam generuje sygnał *Valid*. Na rysunku 3.15 przedstawiono wewnętrzne przetwarzanie danych tego bloku.

Posiadające 67 bitów słowo wyjściowe zostało podzielone na 3 części kolejno: 22, 22 i 23 bitowe. Części te kolejno zapisywane są korzystając z bufora *storage*, którego rozmiar został ustawiony na wystarczający dla obsłużenia najgorszego przypadku, czyli gdy posiadamy 22 bity w buforze przy potrzebnych 23, a otrzymujemy na wejściu kolejne 20 bitów, co daje nam 42 bity.

W trakcie operacji dodawania danych do bufora jest on przesuwany o 20 bitów w lewo, a w to miejsce jest wstawiane nowe 20 bitowe słowo. Zmienna *top\_ptr* przechowuje w sobie górną granicę danych które nie zostały jeszcze odczytane i jest odpowiednio modyfikowana w przypadku odczytu jak i zapisu:



Rys. 3.15. Buforowanie odebranych danych

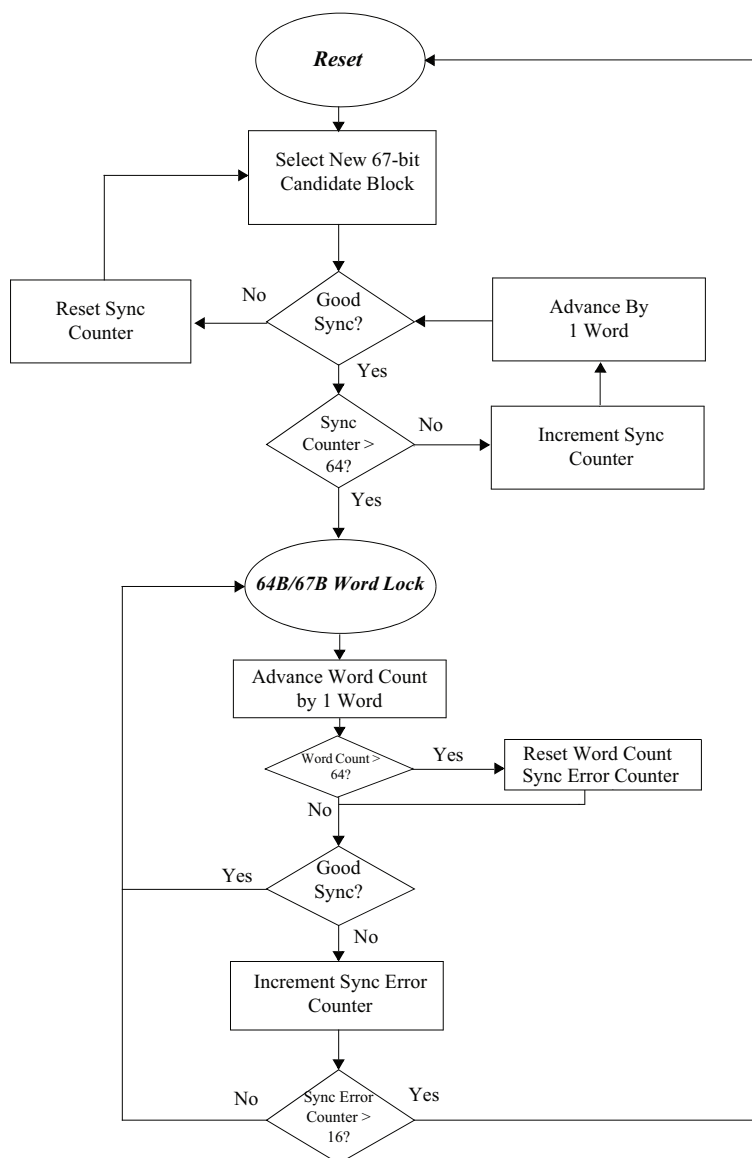
**jeśli w buforze jest wystarczająca ilość danych** – `top_ptr` jest zmniejszany o 2 lub 3 w zależności od tego która część 67 bitowego słowa jest aktualnie zapisywana.

**jeśli w buforze nie ma wystarczającej ilości danych** – `top_ptr` jest zwiększany o 20, ale żadna z części 67 bitowego słowa nie jest uzupełniana.

Blok ten posiada wewnętrzną maszynę stanów, która decyduje która część słowa wyjściowego ma być zapisana. W momencie gdy słowo wyjściowe jest kompletne, sygnał *Valid* jest ustawiany w stan aktywny i słowo może zostać przetworzone przez kolejne bloki.

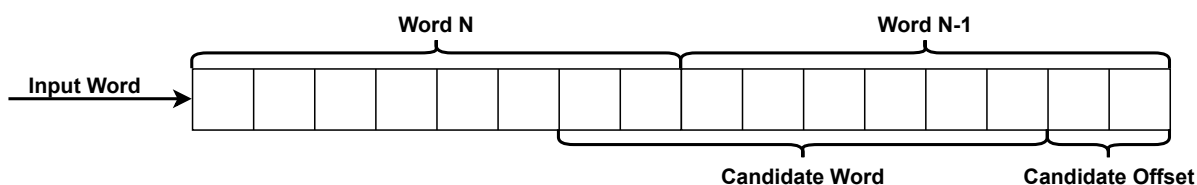
### 3.2.4.2. Decoder 64B/67B

Blok dekodera implementuje maszynę stanów przedstawioną na rysunku 3.16 zgodnie z dokumentacją protokołu Interlaken[2]. Jak można było zauważyć przy okazji bloku kodera, w tabeli 3.5 jedyne legalne stany bitów 64 i 65 to 01 oraz 10, bity te pozwalają dekoderoowi na identyfikację prawidłowych 67 bitowych słów. Dekoder sprawdza czy wybrany kandydat posiada prawidłowe stany tych bitów i po wykryciu 64 poprawnych przejść maszyna stanów wchodzi w stan *Word Lock* ustawiając wyjście *Not locked* w stan nieaktywny i pozwalając na synchronizację kolejnych bloków podmodułu RX. W przypadku gdy 64 poprawne przejścia nie zostaną wykryte kolejny kandydat zostanie wybrany i nastąpi ponowne sprawdzanie stanu bitów 64 i 65, wyjście *Not locked* w stanie aktywnym.



Rys. 3.16. Maszyna stanów dekodera 64B/67B[2]

Wybieranie kandydata zostało zaimplementowane jak przedstawiono na rysunku 3.17. Bufor przesuwny przechowuje ostatnio odebrane i aktualne 67 bitowe słowa. Zgodnie z kandydatem wybranym przez maszynę stanów odpowiednie słowo jest odczytywane i sprawdzany jest 66 bit, aby sprawdzić, czy bity [63:0] muszą być odwrócone.

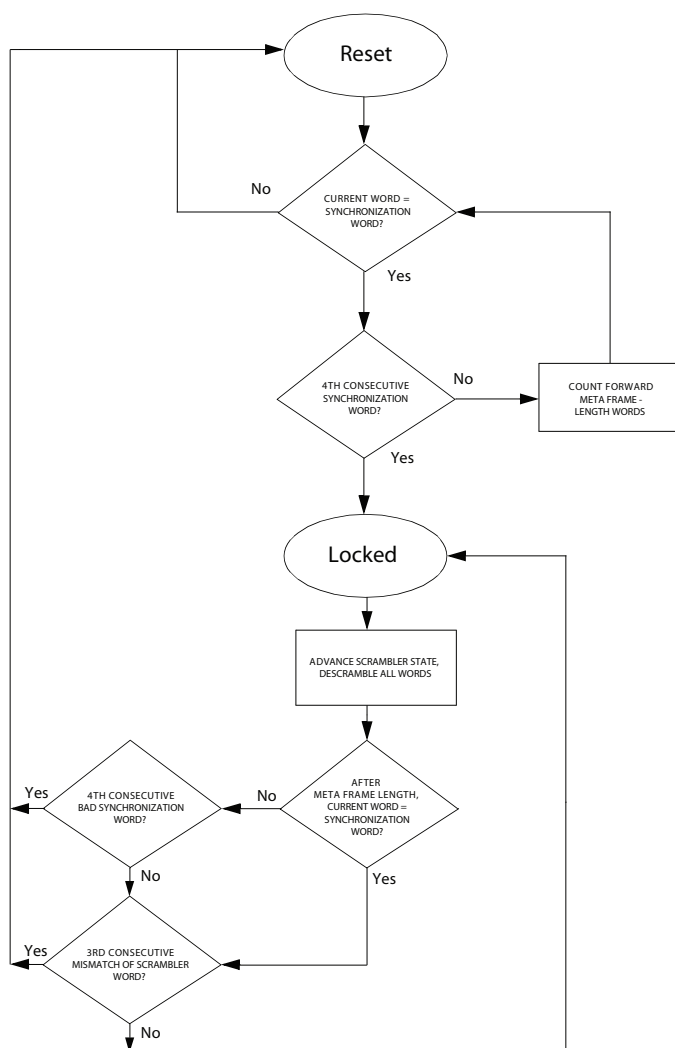


Rys. 3.17. Wybór kandydata



### 3.2.4.3. Descrambler

Implementacja descramblera, podobnie jak w przypadku dekodera implementuje maszynę stanów przedstawioną na rysunku 3.18 zgodnie z dokumentacją protokołu Interlaken[2]. Blok ten wykorzystuje słowa synchronizujące przedstawione w tabeli 3.4 oraz tę samą logikę scramblującą co przedstawioną w listingu 3.1.



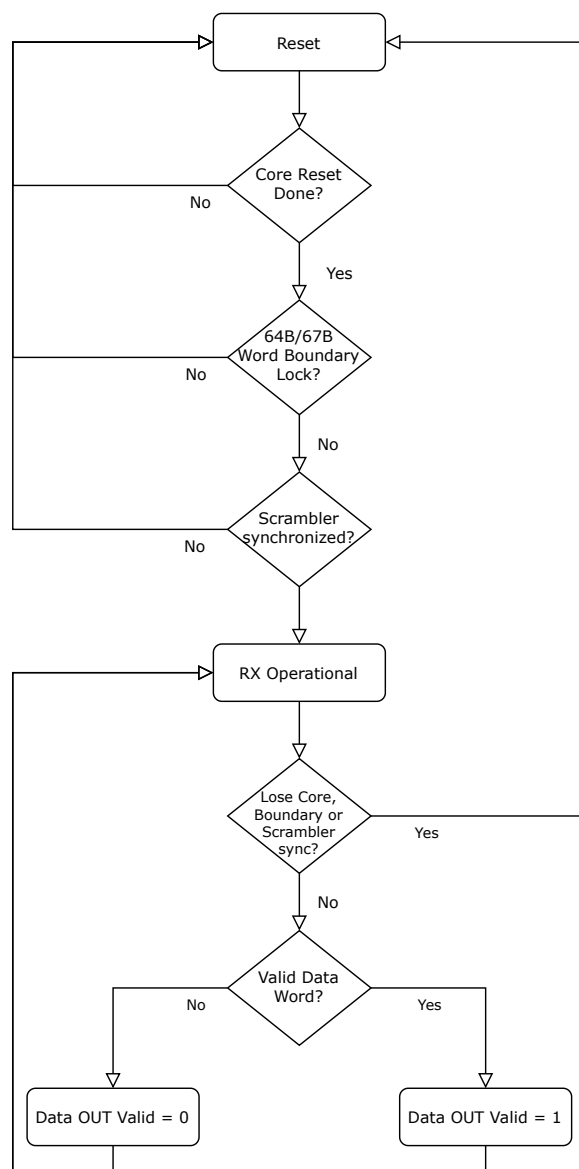
Rys. 3.18. Maszyna stanów descramblera[2]

Maszyna stanów na początku weryfikuje, czy słowo synchronizujące jest otrzymywane z okresem *Meta Frame Length*. Po wykryciu 4 prawidłowych słów synchronizujących maszyna wchodzi w stan *Locked* i ustawia wyjście *Not locked* w stan nieaktywny, które w stanie po resecie jest ustawione w stan aktywny. W trakcie działania, maszyna stanów sprawdza czy słowa synchronizacyjne są otrzymywane okresowo, oraz czy stan wewnętrzny wielomianu logiki scramblującej jest zgodny z tym otrzymywanym za pomocą słów synchronizujących. Podobnie jak w przypadku scramblera, wszystkie słowa są descramblowane z wyłączeniem słów

*sync\_word* oraz *sync\_state*, po otrzymywaniu których nie zmienia się stan wewnętrznego wielomianu descramblera.

#### 3.2.4.4. Receive Interface

W przypadku bloku Receive Interface jego celem jest zapewnienie, aby na wyjście *Data OUT* podmodułu RX były udostępnione jedynie oczekiwane dane wraz z sygnałem *Data OUT Valid* wskazującym, że dana może być odczytana.



Rys. 3.19. Diagram bloku Receive Interface

Sprawdzenie warunku czy bloki takie Dekoder 64B/67B, Descrambler oraz sam rdzeń transceivera zostały zresetowane i są w stanie pozwalającym na poprawne dekodowanie danych, zostało zrealizowane poprzez szeregowe resetowanie tych bloków. Powoduje to, że jeśli choć

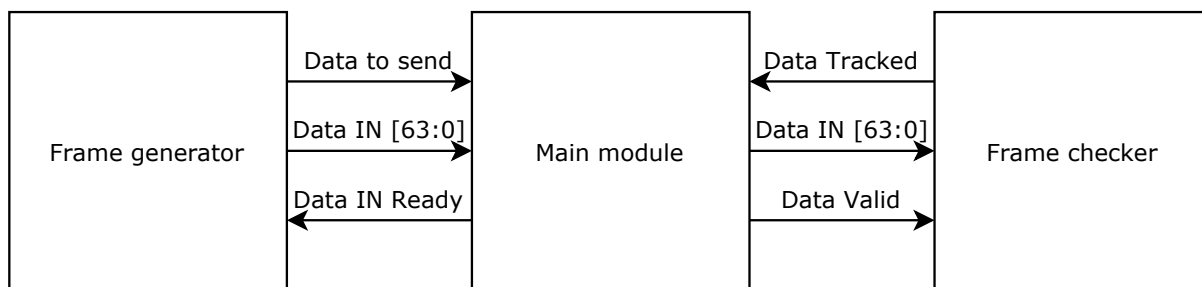
jeden z bloków znajdzie się w niepoprawnym stanie na przykład na skutek błędu transmisji, pozostałe bloki zostaną zresetowane do czasu ponownego wejścia w poprawny stan danego bloku.

Wyjście *Data OUT Valid* jest sterowane zgodnie z sygnałem *Valid* pochodzącym od transceivera jak jest to widoczne na rysunku 3.14. Dodatkowym warunkiem, aby wyjście to zostało ustawione w stan aktywny, jest otrzymanie słowa danych *Data Word*, słowa *Control Words* nie są transmitowane na wyjście.



## 4. Weryfikacja układu

W celu przetestowania działania transmisji należało dodać bloki generujące dane jak i weryfikujące je. Jak przedstawiono na rysunku 4.1 zostały one nazwane *Frame generator* oraz *Frame checker*. Wykorzystanie takiej architektury pozwala na łatwą weryfikację poprawności działania układu poza symulacją.



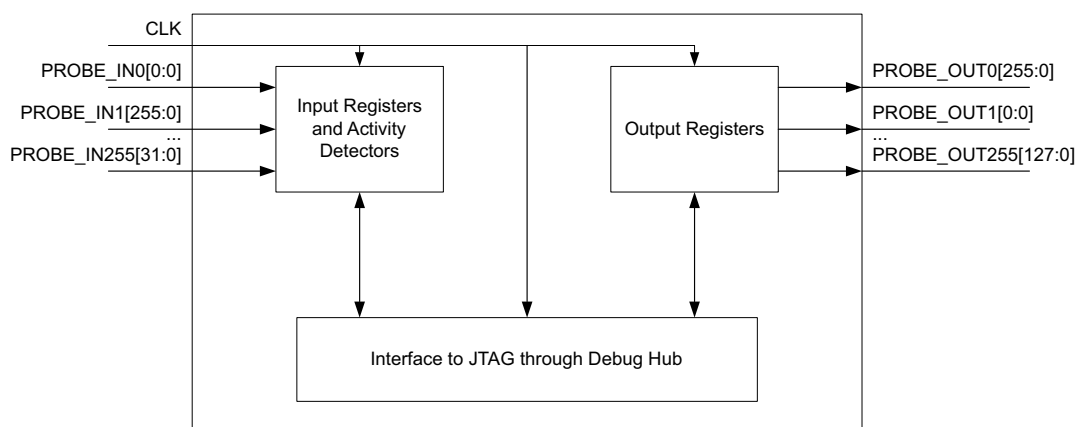
Rys. 4.1. Współpraca modułu z blokami weryfikującymi

Sposób prowadzenia połączeń po stronie równoległej nadajnika został przedstawiony na rysunku 4.1, w celu uproszczenia diagramu pominięte zostały linie *TX USB CLK* i *TX RESET* dla bloku *Frame generator* jak i odpowiadających im sygnałów w przypadku bloku *Frame checker*.

Po stronie szeregowej natomiast wyprowadzone są jedynie pojedyncze linie *TX IN* oraz *RX IN*. W przypadku zastosowanej płytki *Digilent Genesys 2* zostały przypisane do obecnych na płycie złącz mini DisplayPort. Każdy z tych portów posiada po 4 linie różnicowe, lecz w przypadku tego projektu transmisja odbywa się wyłącznie po jednej linii. W przypadku chęci zastosowania większej ilości linii należałoby dodać kolejne moduły obsługujące niezależne linie transmisyjne oraz dodać logikę zarządzającą wraz z układem do formatowania odebranych danych.














## 4.1. Bloki VIO

IP Core *Virtual Input/Output (VIO)*, w skrócie VIO, są blokami służącymi do sterowania jak i monitorowania wewnętrznych sygnałów FPGA w czasie rzeczywistym. Dodatkową funkcjonalnością są detektory aktywności pozwalające wykryć szybkie zmiany na wejściach. Ilość wejść i wyjść bloku jest konfigurowalna, tak samo jak szerokość portu w bitach.



Rys. 4.2. Diagram blokowy VIO[12]

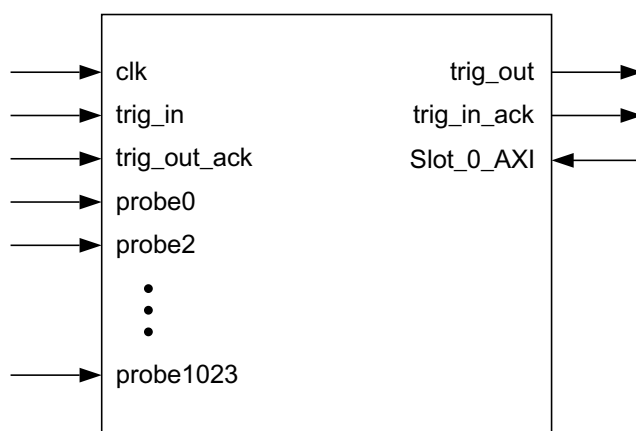
Na rysunku 4.3 przedstawiono wygląd interfejsu z przykładowymi sygnałami podpiętymi do różnych bloków VIO. Aby uzyskać dostęp do tego interfejsu, należy wcześniej nawiązać połączenia *JTAG*, za pośrednictwem *Hardware Manager* w programie *Vivado*.

>  CANDIDATE[6:0]	[U] 42		Input	hw_vio_2
 TX_FSM_RESET_DONE	[B] 1		Input	hw_vio_1
 RX_FSM_RESET_DONE	[B] 1		Input	hw_vio_1
>  ERROR_COUNT_OUT[7:0]	[H] 00		Input	hw_vio_5
 TRACK_DATA_OUT	[B] 1		Input	hw_vio_5
 PASSTHROUGH_ENCODER	[B] 0	▼	Output	hw_vio_4
 PASSTHROUGH_DECODER	[B] 0	▼	Output	hw_vio_2
 PASSTHROUGH_SCRAMBLER	[B] 0	▼	Output	hw_vio_4
 PASSTHROUGH_DESCRAMBLER	[B] 0	▼	Output	hw_vio_2
 SOFT_RESET	[B] 0	▼	Output	hw_vio_1
 OVERRIDE_DATA_VALID	[B] 1	▼	Output	hw_vio_1
>  OFFSET[5:0]	[U] 0	▼	Output	hw_vio_3
>  MASK[19:0]	[H] 0_0000	▼	Output	hw_vio_3

Rys. 4.3. Interfejs VIO

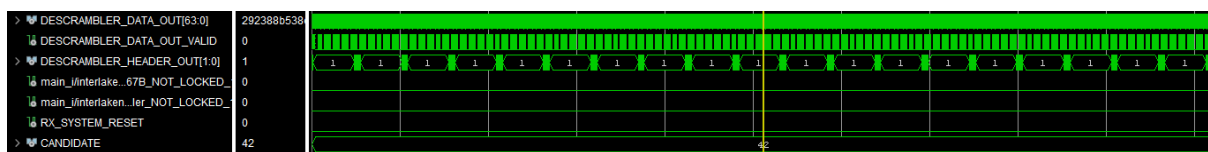
## 4.2. Bloki ILA

W przypadku IP Core o nazwie *Integrated Logic Analyzer (ILA)*, w skrócie ILA, możemy jedynie korzystać z wejść celem zbierania danych, w przeciwieństwie do bloków VIO, które pozwalały również na kontrolę wyjść. Kolejną znaczącą różnicą w porównaniu do VIO, jest konieczność wykorzystania pamięci BRAM celem przechowywania skonfigurowanej ilości próbek. Jak sama nazwa wskazuje, blok ten działa jak analizator stanów logicznych, pozwalając na konfigurację triggera wyzwalającego zapis, a następnie pozwalając na wyświetlenie zapisanych przebiegów w programie Vivado po odczycie za pośrednictwem interfejsu *JTAG*.



Rys. 4.4. Rdzeń ILA[13]

Na rysunku 4.5 przedstawiono przykładowe przebiegi zapisane za pośrednictwem tego bloku i wyświetlone w programie Vivado.

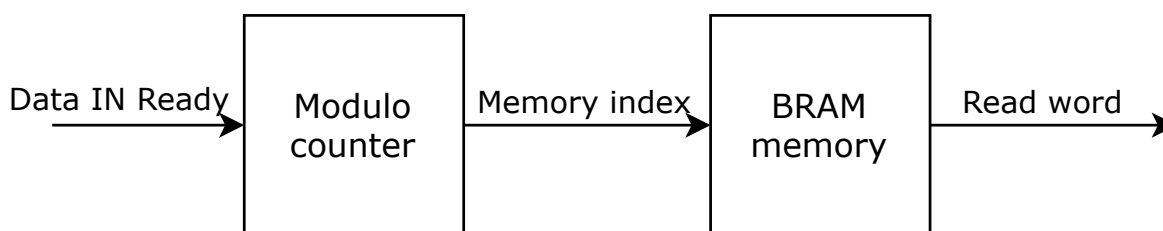


Rys. 4.5. Interfejs ILA

## 4.3. Frame Generator

Generator ramek został zaimplementowany jako prosty blok z predefiniowaną pamięć BRAM, oraz licznik modulo inkrementowany na zboczu zegara jeśli sygnał wejściowy *Data IN Ready* jest w stanie aktywnym. Maksymalna wartość licznika jest dobierana tak, aby mógł on pokryć cały zakres pamięci. Na podstawie wartości licznika, blok zwraca odpowiednią wartość

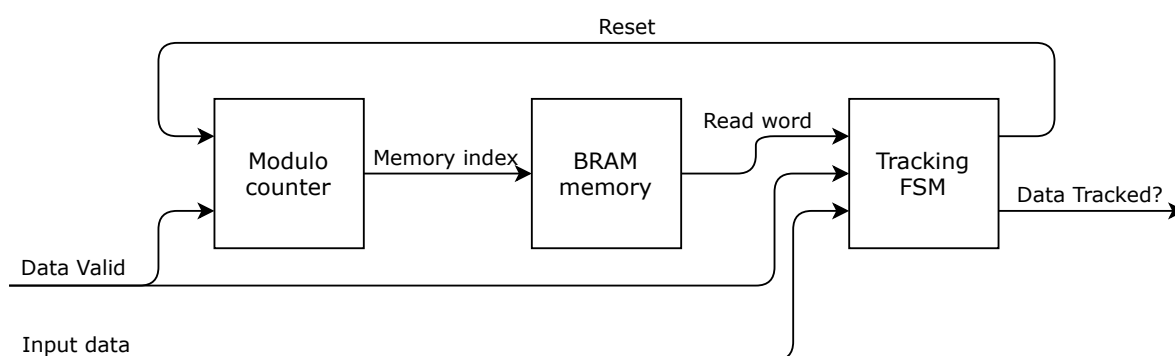
z pamięci na wyjście. Wyjście *Data to send* widoczne na rysunku 4.1 jest cały czas ustawione w stan aktywny.



Rys. 4.6. Odczyt danych w bloku Frame Generator

## 4.4. Frame Checker

W przypadku weryfikacji odbieranych ramek, została wykorzystana zmodyfikowana maszyna stanów generowana w przykładowym projekcie wykorzystującym transceiver. Zasada jej działania została zachowana, lecz musiała zostać przystosowana do obsługi sygnału *Data Valid* w taki sposób, by dane wejściowe były weryfikowane jedynie jeśli sygnał *Data Valid* jest w stanie aktywnym. Dodatkowo musiał zostać zmieniony sposób odbierania danych, jako że pierwotnie blok ten nie przewidywał on wykorzystania bloków obecnych w podmodule RX, opisanym w rozdziale 3.2.4.



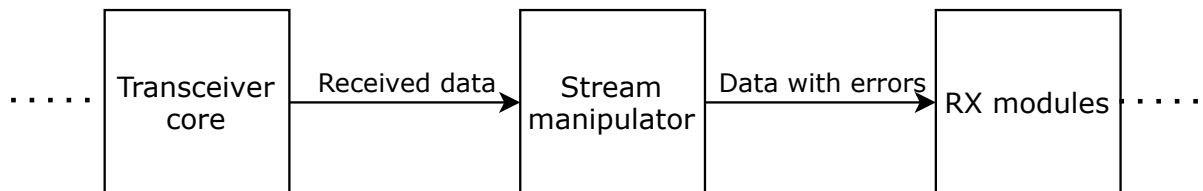
Rys. 4.7. Weryfikacja danych w bloku Frame Checker

Wygenerowana maszyna stanów rozpoczyna swoje działanie szukając wcześniej zdefiniowanego 64 bitowego słowa startowego. Po wykryciu go, rozpoczyna się porównywanie otrzymanych słów z tymi odczytywanymi z pamięci. Po rozpoczęciu porównywania, jeśli zostanie wykryty błąd licznik błędów zostanie zinkrementowany, a maszyna stanów wróci do stanu szukania słowa startowego.[10]



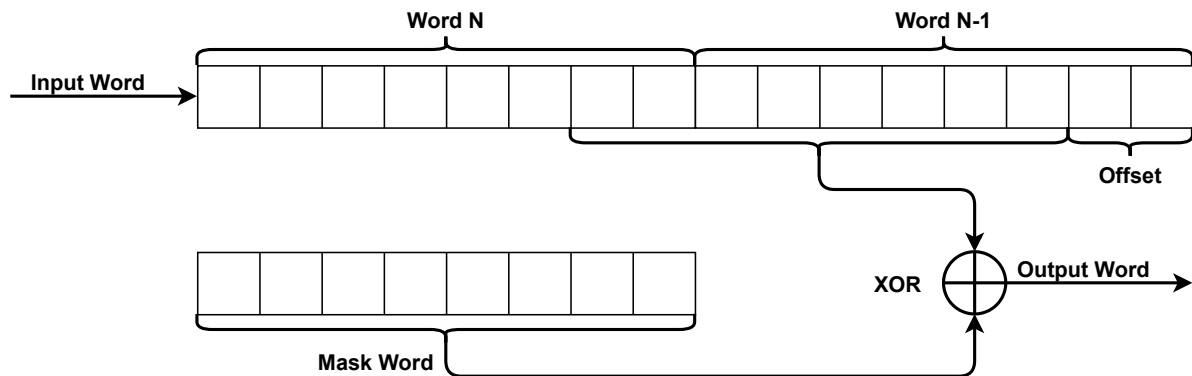
## 4.5. Stream Manipulator

Blok ten został stworzony w celu dodania możliwości wprowadzenia błędów transmisji i został on umieszczony pomiędzy modulem transceivera a resztą układu, jak jest to widoczne na rysunkach 3.3 oraz 4.8.



Rys. 4.8. Wykorzystanie bloku Stream Manipulator

Główną funkcjonalnością tego bloku jest możliwość zmiany przesunięcia danych w otrzymywanych pakietach, w celu sprawdzenia czy zostanie wykryty błąd oraz czy układ będzie mógł ponownie prawidłowo wykryć 67 bitowe słowo. Dodatkowo została dodana opcja negacji odbieranych bitów zgodnie z podaną maską jak widoczne jest to na rysunku 4.8.



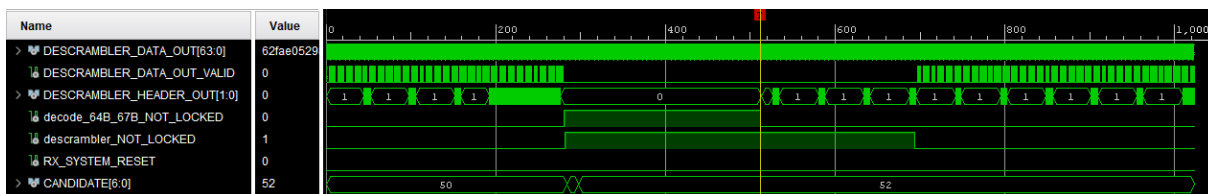
Rys. 4.9. Sposób działania bloku Stream Manipulator

## 4.6. Przeprowadzone testy

Po dokonaniu syntezy i zaprogramowaniu układu, transmisja kablem DisplayPort jest poprawna i nie występują błędy. W ramach testu został wykorzystany blok *Stream Manipulator* opisany w rozdziale 4.5.

### 4.6.1. Zmiana offsetu w bloku Stream Manipulator

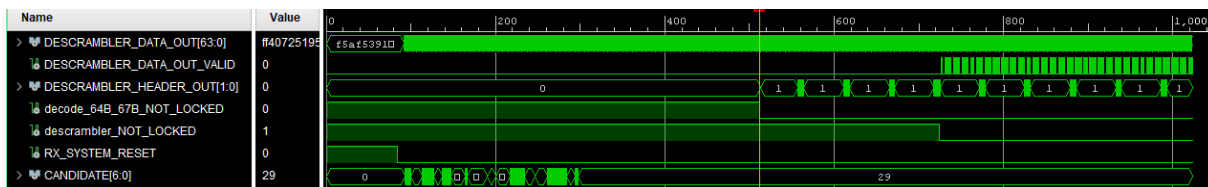
Jak jest to widoczne na rysunku 4.10 zmiana ta spowodowała błędy w transmisji, co można zaobserwować poprzez zmianę do tej pory okresowych zmian sygnału *Descrambler header out*, spowodowało to zresetowanie bloków descramblera i dekodera oraz wybranie nowego kandydata przez blok dekodera. Po wykryciu 64 poprawnych przejść, sygnał *Not locked* dekodera przeszedł w stan nieaktywny i descrambler rozpoczął synchronizację. Gdy zostały wykryte 4 słowa synchronizacyjne, descrambler również ustawił sygnał *Not locked* w stan nieaktywny i transmisja została wznowiona.



Rys. 4.10. Efekt zmiany offsetu

### 4.6.2. Zmiana maski w bloku Stream Manipulator

Z racji że zmiana maski powoduje całkowite utracenie komunikacji na rysunku 4.11 został uchwycony moment, gdy zmiana ta została cofnięta, a transmisja przywrócona. Z powodu długotrwałego zerwania transmisji wewnętrzna maszyna stanów transceivera przedstawiona wcześniej na rysunku 3.8 spowodowała reset części RX transceivera, co można zaobserwować przez sygnał *RX System Reset* w stanie aktywnym przez pewien czas. Po zakończeniu resetu, blok dekodera rozpoczął próbę wybrania właściwego kandydata, co można zaobserwować dzięki zmianom sygnału *Candidate*. Po tym kroku reszta przebiega dokładnie jak w przypadku zmiany offsetu i transmisja zostaje nawiązana ponownie.



Rys. 4.11. Efekt zmiany maski

## 5. Podsumowanie i wnioski

Zaprezentowany projekt warstwy kodującej protokołu Interlaken działa zgodnie z postawionymi mu celami, transmisja jest nawiązywana poprawnie, a w razie wystąpienia błędów transmisji wznawiana. Projekt jest syntezywalny i był uruchomiony na płycie *Digilent Genesys 2*.

Spośród elementów zaimplementowanych w ramach tego projektu zdecydowanie najwięcej trudu przyniosło zastosowanie gearboxów, gdyż dokumentacja protokołu nie precyzowała jak powinien być zaimplementowany taki blok. Finalnie z powodu trudności w samodzielnym zaprojektowaniu takiego bloku została wykorzystana implementacja pochodząca z książki [9]. Mimo korzystania z gotowego rozwiązania nadal konieczne było zintegrowanie tych bloków z pozostałymi częściami projektu, poprzez dodanie sygnałów *Valid* koordynujących działanie pozostałych bloków.

Drugim zadaniem które spowodowało pewną dozę problemów, było uruchomienie samego bloku transceivera. Bardzo pomocna była opcja generowania przykładowego projektu, lecz mimo to uruchomienie takiego projektu na sprzęcie sprawiało trudności. Dopiero po skonfigurowaniu projektu z najbardziej podstawowymi opcjami udało się uzyskać działający projekt. W początkowym etapie szerokość danych w transceiverze była ustawiona na 80 bitów, kiedy jeszcze nie były zaimplementowane gearboxy, aby najpierw uzyskać najprostszy działający projekt. W późniejszych etapach kolejne funkcjonalności były kolejno dodawane i po prowadzeniu gearboxów szerokość danych transceivera została zmieniona na 20 bitów.

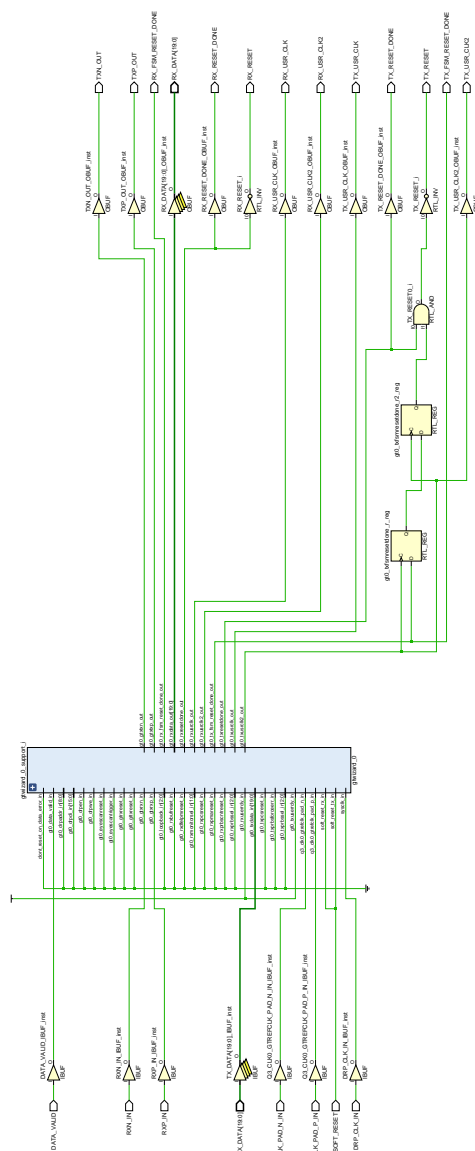
W ramach możliwości rozwijania projektu jednym z punktów może być zbadanie działania funkcjonalności bitflip obecnej w bloku gearbox 20B/67B z książki [9]. Na ten moment wybieranie słowa jest realizowane poprzez manipulacje rejestrem przesuwным przechowującym 2 słowa, lecz rozwiązanie to z pewnością jest mniej optymalne.

Drugim punktem może być optymalizacja funkcjonalności zliczania ilości jedynek w słowie obecnej w bloku enkodera. Jak jest to wspomniane w sekcji dotyczącej “Bit Population Count” w książce [9] wykorzystywanie pętli for do takich obliczeń może powodować powstawanie struktur trudnych do zinterpretowania przez narzędzia do syntezy.

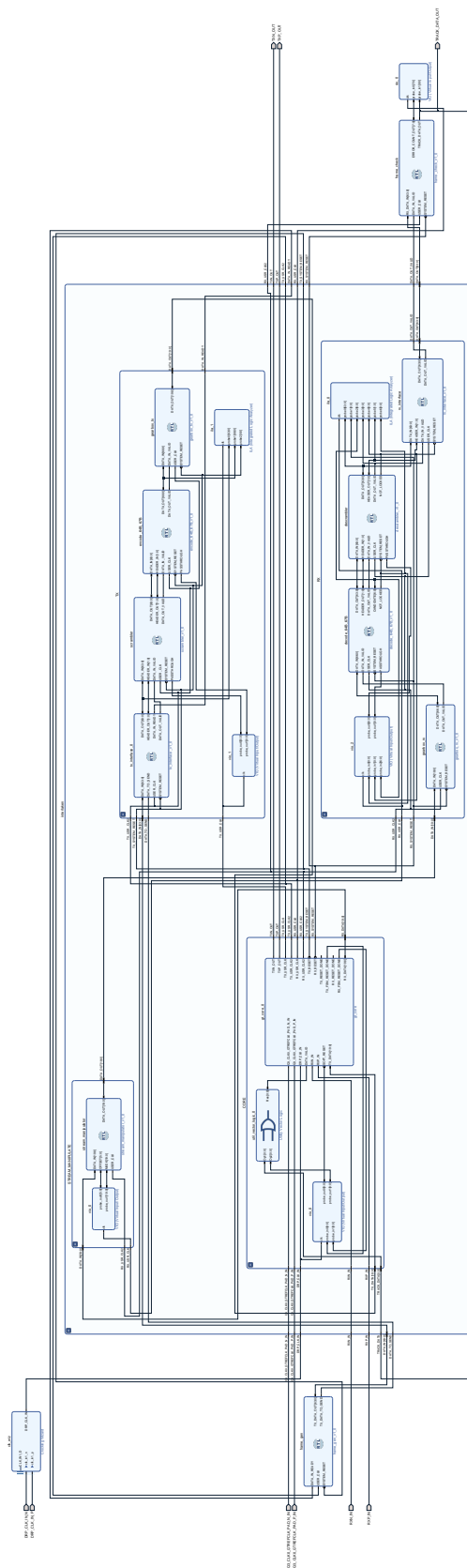
Finalnie można również dokonać rozszerzenia funkcjonalności tego projektu do obsługi wielu linii jednocześnie, lecz wymaga to dodania bloków zarządzających takimi liniami.

Podsumowując stworzony projekt może być świetną bazą do rozwijania projektów wykorzystujących transmisję szeregową, jak zostało to zaprezentowane w tym projekcie na przykładzie protokołu Interlaken.

### A. Diagramy



**Rys. A.1.** Schemat RTL “gt\_core”



Rys. A.2. Widok Block Design projektu w środowisku Vivado







## Bibliografia

- [1] *Interlaken Technology: New-Generation Packet Interconnect Protocol White Paper*. v1.0. Cortina Systems, Cisco Systems i Silicon Logic Engineering Inc. 2007.
- [2] *Interlaken Protocol Definition, A Joint Specification of Cortina Systems and Cisco Systems*. Revision 1.2. Cortina Systems Inc. i Cisco Systems. 2008.
- [3] *Semiconductor intellectual property core*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Semiconductor\\_intellectual\\_property\\_core](https://en.wikipedia.org/wiki/Semiconductor_intellectual_property_core).
- [4] *Transceiver*. Wikipedia. URL: <https://en.wikipedia.org/wiki/Transceiver>.
- [5] *Multi-gigabit transceiver*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Multi-gigabit\\_transceiver](https://en.wikipedia.org/wiki/Multi-gigabit_transceiver).
- [6] Jerry C. Chen. *Multi-Gigabit SerDes: The Cornerstone of High Speed Serial Interconnects*. Design-Reuse. URL: <https://www.design-reuse.com/articles/10541/multi-gigabit-serdes-the-cornerstone-of-high-speed-serial-interconnects.html>.
- [7] *Digital Encoding Technique Of Scrambling Computer Science Essay*. UKEssays. URL: <https://www.ukessays.com/essays/computer-science/digital-encoding-technique-of-scrambling-computer-science-essay.php>.
- [8] *Scrambler*. Wikipedia. URL: <https://en.wikipedia.org/wiki/Scrambler>.
- [9] *Advanced Synthesis Cookbook*. 11.0. Altera. 2011.
- [10] *7 Series FPGAs Transceivers Wizard - Product Guide*. v3.6. Xilinx. 2016.
- [11] *Digilent Genesys 2 Schematic*. Rev H.1. Digilent. 2015. URL: [https://reference.digilentinc.com/\\_media/genesys2/genesys2\\_public\\_sch.pdf](https://reference.digilentinc.com/_media/genesys2/genesys2_public_sch.pdf).
- [12] *LogiCORE IP Virtual Input/Output - Product Guide*. v2.0. Xilinx. 2013.
- [13] *Integrated Logic Analyzer - Product Guide*. v6.1. Xilinx. 2016.