Projekt SYCYF

-

Zespół nr. 2

-

Karolina Ciupak

Natalia Klucznik

Jan Burs

Jakub Dreszer

Ryszard Ptak

Oskar Zając

Politechnika Warszawska

24 stycznia 2021

Historia zmian

Wersja	Data	Autor	Opis zmian
1.0	18.10.2020	OZ	Pierwsza wersja raportu Etapu I
1.1	19.10.2020	NK, JB, OZ	Poprawki do Etapu I
2.0	10.11.2020	KC, JB, NK, OZ, JD, RP	Pierwsza wersja raportu Etapu II
2.1	01.12.2020	KC	Poprawki do etapu II
3.0	27.11.2020	NK	Wstęp do Etapu III
3.1	28.11.2020	CK,OZ	Poprawa Etapu I oraz II
3.2	29.11.2020	NK, OZ, JB, JD	Dalsza część Etapu III
4.0	02.01.2021	JB	Wstęp do Etapu IV
4.1	01.01.2021	NK	Encoder
4.2	02.01.2021	OZ, CK, JD, JB, RP	Decoder
4.3	02.01.2021	KN, CK, OZ, JD, JB	Poprawki do etapu III i IV
5.0	24.01.2021	KN, CK, OZ, JD, JB	Pierwsza wersja raportu Etapu V

Spis treści

1. Organizacja pracy (Etap I)

W celu ustalenia organizacji pracy przy projekcie z Systemów Cyfrowych spotkaliśmy się na naszym kanale w MS Teams. Ponieważ grupa nie posiada naturalnego lidera, postanowiliśmy podzielić się zarządzaniem danymi etapami na zmianę zgodnie z poniższą tabelą:

Dodatkowo, każdy etap poza liderem będzie miał wyznaczoną osobę zatwierdzającą go. Dzięki temu nigdy nie dojdzie do sytuacji w której błąd jednej osoby przyczyni się do obniżenia jakości produktu efektu danego etapu.

W ramach projektu będziemy korzystać hybrydowo z metody Design Thinking (Double Diamond) oraz metodyki

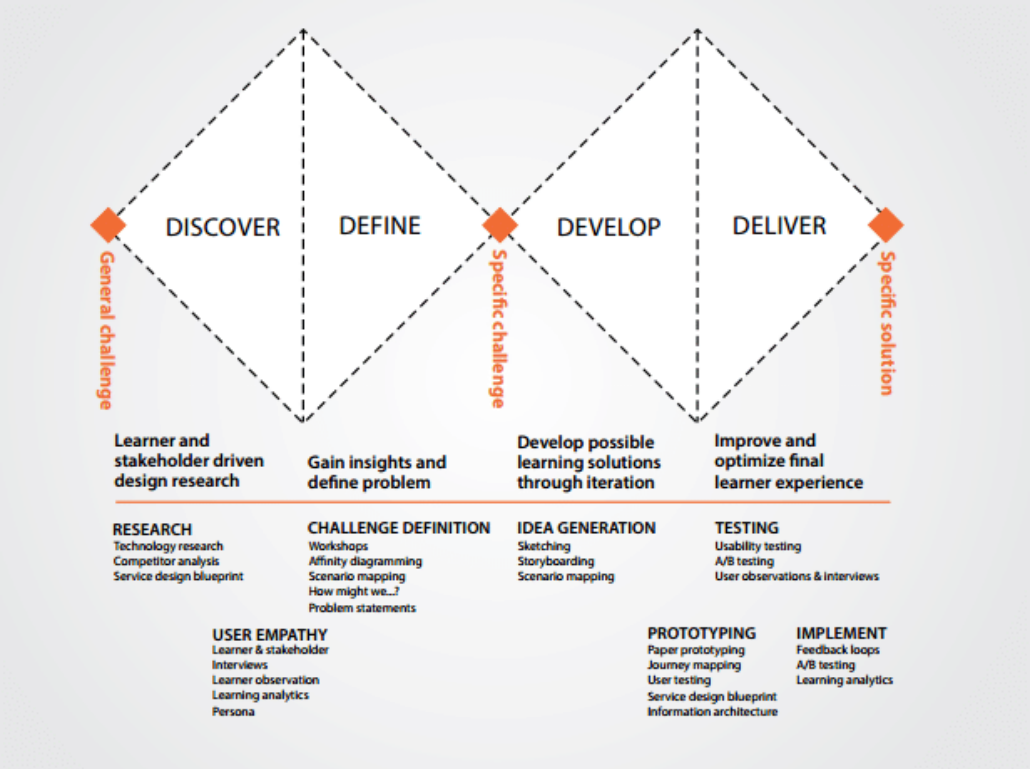
Tabela 2. Zarządzanie poszczególnymi etapami

Etap	Lider
I	Oskar Zając
II	Karolina Ciupak
III	Natalia Klucznik
IV	Jan Burs
V	Jakub Dreszer

zwinnej (Agile). Połączenie to wydaje się rozsądne, ponieważ równocześnie korzystamy z narzędzia do określania problemów, które musimy rozwiązać oraz ze sposobu na wykonywanie pracy w tzw. Sprintach. Na początku każdego sprintu zwołane zostanie spotkanie na którym zostanie wyznaczona osoba zatwierdzająca dany etap oraz dojdzie do podziału pracy między członków grupy. Gdy po zakończeniu etapu okaże się, że któraś z prac zajęła więcej czasu niż na początku szacowano, osoba która wykonała większą pracę dostanie mniej zadań w kolejnym sprincie.

1.1. Design Thinking

W naszej realizacji projektu będziemy korzystać tylko z lewego "diamentu". Początkowo będziemy zbierać informacje na temat danego problemu, następnie będziemy tworzyć grupowe burze mózgów(rozszerzająca się część diamentu), których produktem będą mapy myśli "na brudno". Z tych roboczych map będziemy definiować zadania (zwężenie diamentu) na najbliższe tygodnie oraz dzielić się pracą i opcjonalnie tworzyć 2-3 osobowe podzespoły.



Rys. 1. Double Diamond [7]

1.2. Zarządzanie projektem

1.2.1. Metody

Zdecydowaliśmy się zastosować w naszym projekcie metodykę zwinną (Agile). Wybraliśmy ją, ponieważ jest ona stosowana w wielu topowych firmach oraz część grupy miała już okazję z niej korzystać. Pasuje ona również do realizacji uczelnianego projektu: grupa spotyka się, ustala zadania do wykonania na najbliższy tydzień, dzieli się nimi i po tygodniu zbiera się aby omówić postępy. Agile to metodyka bardzo często stosowana w tworzeniu oprogramowania. Jak nazwa wskazuje, jej największą zaletą jest zwinność w reagowaniu na zmienność wyzwań w trakcie tworzenia oprogramowania.

1.2.2. Narzędzia

Projekt realizowany jest zdalnie, zatem będziemy korzystać w wielu narzędzi komunikacyjnych oraz organizacyjnych. Zdecydowaliśmy się na te, które już znamy:

— Overleaf / TeXstudio - praca nad sprawozdaniami. Overleaf pozwoli nam na równoległą pracę nad jednym sprawozdaniem, TeXstudio jest wygodniejsze do pracy indywidualnej którą można potem przekleić do projetu w Overleaf
— Trello - nasza tablica kanban. Będziemy systematycznie uzupełniać ją o nowe zadania oraz śledzić pracę nad aktualnymi. Będzie też pomocna przy ocenie aktywności poszczególnych człownków zespołu po zakończeniu proejktu.
— MS Teams - cotygodniowe spotkania, przechowywanie plików
— Github/Gitlab - praca nad softwarem. Będą one konieczne od Etapu III. Gitlab to narzędzie które znamy już dobrze z przedmiotu Podstawy Programowania. Dzięki proejktowi poznamy aspekt grupowej pracy nad jednym kodem.
— Messenger - ustalanie terminów spotkań, spontaniczna komunikacja

Gdy rozpocznie się etap III i będziemy dokładniej znali nasze zadania, lista narzędzi rozrzeczy się o te bardziej techniczne (np. Quartus, ModelSim)

2. Informacje podstawowe

W tym rozdziale zadaniem grupy było szukanie w różnych dostępnych źrółach informacji na temat naszego projektu. Zagadnienia, które nas najbardziej zainteresowały, to:
— kod splotu
— kodowanie korekcyjne
— cykliczny kod nadmiarowy (CRC)
— kod Hamminga
— kontrola parzystości
Powyższe zagadnienia zostały opisane w podroździałach poniżej.

2.1. Kod splotowy

Kod splotowy jest to rodzaj kodu korekcyjnego, który nie wymaga dzielenia strumieni informacji na bloki i zwykle jest określany przez trzy parametry:

$$(n, k, m)$$

gdzie:
n – liczba bitów wyjściowych,
k – liczba bitów wejściowych,
m - długość rejestru pamięci.

Jego ideą jest generowanie n-bitowego ciągu wyjściowego w zależności od k-bitowego ciągu wejściowego oraz zawartości komórek pamięci, a jego działanie przypomina operację splotu. Dla tego typu kodów, dekoder realizujący algorytm Viterbiego, stanowi dekoder o maksymalnej wiarygodności ML (maximum likelihood), a ponadto ma duże możliwości korekcyjne kodów splotowych oraz stosunkowo prosty algorytm dekodowania. Metoda ta polega na analizowaniu wszystkich ścieżek i porównaniu ich z sekwencją odebraną. Patrzymy przy

tym na liczbę pozycji, na których się różnią i wybieramy oczywiście ścieżkę najbardziej podobną do odebranej sekwencji. Dekodowanie to jest stosowane w systemach, gdzie błędy występują stosunkowo rzadko – małe jest prawdopodobieństwo przekłamania bitów oraz błędy występują zupełnie przypadkowo, a prawdopodobieństwo błędu podwójnego jest o wiele mniejsze od prawdopodobieństwa wystąpienia błędu pojedynczego.

Źródła: [14] [15]

2.2. Kodowanie korekcyjne

Kodowanie korekcyjne jest to technika dodawania nadmiarowości do transmitowanych pakietów(będących ciągiem zer i jedynek o określonej długości) poprzez dodanie do nich dodatkowych bitów. Umożliwia całkowicie lub częściowo wykryć i skorygować błędy powstałe w wyniku różnych zakłóceń. Zalety kodowania korekcyjnego:

— nie trzeba wykorzystywać kanału zwrotnego, aby poinformować nadawcę o błędzie i konieczności ponownego przesłania bloków informacji

— umożliwia detekcję i korekcję błędów, kiedy retransmisja jest niemożliwa(ze względu na zbyt wysokie koszty lub ograniczenia czasowe)

Wady:

— stosując kodowanie korekcyjne możemy utracić część potrzebnych informacji w przypadku, gdy nie wykryjemy wszystkich błędów, co nie wystąpi kiedy poinformujemy nadawcę o zakłóceniu odebranych pakietów i poprosimy o retransmisję (to tylko moje spostrzeżenie nie wiem czy poprawne)

W naszym zadaniu projektowym możemy rozważyć dodanie pewnej ilości nadmiarowych bitów do transmitowanych 256-bitowych bloków w celu wykrycia i korekcji ewentualnych błędów spowodowanych silnymi zakłóceniami.

Źródła: [16] [15] [9]

2.3. Cykliczny kod nadmiarowy

CRC (Cyclic Redundancy Check) to kod stosowany do wykrywania błędów pojawiających się podczas przesyłania bądź magazynowania danych w postaci binarnej. Ma określoną z góry długość i jest wygenerowany z ciągu danych wejściowych, a następnie dodawany do oryginalnej wiadomości. Nadawca (lub np. proces zapisujący dane na dysk) oblicza CRC dla wiadomości, którą chce wysłać i dodaje je do wysyłanych danych. Odbiorca (lub np. proces odczytujący dane z dysku) również oblicza sumę kontrolną danych, które otrzymał i porównuje ją z wartością CRC wysłaną przez nadawcę. Jeśli wyniki się różnią, doszło do przekłamania informacji. Algorytm obliczania n-bitowego kodu CRC: 1. Do ciągu danych dodaje się n wyzerowanych bitów, 2. W linii poniżej zapisuje się n+1 bitowy dzielnik CRC, 3. Jeżeli nad najstarszą pozycją dzielnika jest wartość 0, to przesuwa się dzielnik w prawo o jedną pozycję, aż do napotkania 1, 4. Wykonuje się operację XOR pomiędzy bitami dzielnika i odpowiednimi bitami ciągu danych, uwzględniając dopisane n bitów, 5. Wynik zapisuje się w nowej linii – poniżej, 6. Jeżeli liczba bitów danych jest większa lub równa 4, przechodzi się do kroku 2, 7. N najmłodszych bitów stanowi szukane CRC, czyli cykliczny kod nadmiarowy. Kod CRC jest szeroko wykorzystywany w transmisji cyfrowej i wydaje się przystępnym narzędziem, z którego będziemy mogli skorzystać w naszym projekcie.

Źródła: [3]

2.4. Kod Hamminga

Kod Hamminga pozwala na detekcję błędu przekłamania jednego, dwóch a niekiedy trzech bitów, jednak możliwa jest tylko korekcja błędu pojedynczego bitu. W skrócie : Jeśli w wiadomości jest więcej bitów korygujących i jeśli te bity dla różnej kombinacji bitów przekłamanych dają różne rezultaty, wtedy możemy zidentyfikować nieprawidłowe bity według następującego algorytmu:Przyjmijmy, że bity parzystości znajdują się na pozycjach będących potęgami 2. Wszystkie pozycje będące potęgami 2 (1, 2, 4, 8, 16,...) są bitami parzystości. Wszystkie pozycje niebędące potęgami 2 (3, 5, 6, 7, 9, 10,...) to bity informacyjne. Każdy bit parzystości wskazuje parzystość pewnej grupy bitów w słowie, a jego pozycja określa, które bity ma sprawdzać, a które opuszczać. Bit na pozycji n opuszcza n-1 bitów sprawdza n następnie opuszcza n i sprawdza n... Jeśli liczba jedynek w sprawdzonych bitach jest parzysta wtedy na pozycji n umieszczamy 0 jeśli nieparzysta to 1. Jeżeli więc dla tej samej wiadomości n-bitowej otrzymamy inne bity parzystości możemy stwierdzić, że mamy gdzieś przekłamanie. Jednak każdy bit jest sprawdzany unikalną kombinacją bitów parzystości co daje nam możliwość zlokalizowania i skorygowania przekłamanego bitu. Wystarczy, że zsumujemy ze sobą pozycję bitów parzystości, na których znaleźliśmy różnicę a otrzymamy pozycję bitu przekłamanego.

Jednak kod Hamminga nie jest odpowiednim kodem korekcyjnym, ponieważ, jak zostało powiedziane wyżej, jest on pomocny tylko przy korekcji błędów pojedynczych, a nie przy błędach seryjnych (burst error). Jednak można to zmienić, stosując odpowiedni preprocesing, np metodę przeplatania (interleaving).

Przeplatanie to narzędzie używane do ulepszania istniejących kodów korekcji błędów, tak, aby można je było wykorzystać również do korekcji błędów seryjnych. Większość kodów korekcji błędów ma na celu korygowanie błędów pojedynczych, spowodowanych przez szum. Burst error to błędy występujące w sekwencji lub w grupach. Są one spowodowane defektami nośników pamięci lub przerwami w sygnałach komunikacyjnych z powodu czynników zewnętrznych, takich jak wyładowania atmosferyczne itp. Przeplot modyfikuje kod korekcji lub wykonuje pewne przetwarzanie danych po ich zakodowaniu. Podczas przeplatania symbole wiadomości są rozmieszczane w wielu blokach kodu przez element przeplatający przed wysłaniem przez kanały sieciowe. Z tego powodu długie sekwencje szumów są rozłożone na wiele bloków. Gdy dekoder przestawia bloki, błędy pojawiają się jako niezależne błędy losowe lub błędy serii o krótkich długościach. Dekoder jest w stanie poprawić błędy za pomocą algorytmu korekcji błędów.

Źródła: [5] [13] [10] [12]

2.5. Kontrola parzystości

Kontrola parzystości jest metodą detekcji błędów w przesłanej wiadomości. Polega ona na dodaniu do informacji bitu kontrolnego(bitu parzystości) którego wartość sygnalizuje, czy liczba jedynek w informacji jest parzysta
(bit kontrolny = 0) czy nieparzysta(bit kontrolny = 1).

Zaletami tej metody są:
— Bardzo niskie zwiększenie rozmiaru przesyłanej wiadomości - w wiadomości 256 bitowej jest to mniej niż 0,4 %
— Szybkość działania - aby ustalić czy liczba jedynek jest parzysta wystarczy wszystkie bity połączyć bramką XOR która zwróci 1 jeśli suma jedynek jest nieparzysta i 0 w przeciwnym przypadku.

Minusy tej metody:
— Wykrywa do 1 błędu - w przypadku parzystej liczby błędów metoda nie zasygnalizuje ich.
— W przypadku błędu na bicie parzystości oraz nieparzystej liczbie błędów w informacji nie wykryje błędu

W związku z powyższym można ocenić, że ta metoda jest skuteczna tylko jeśli wystąpienie pojedynczego błędu jest anomalią a ich większa ilość jest pomijalnym przypadkiem. W żadnym wypadku nie nadaje się do zastosowania w naszym projekcie. Błędy typu burst errors nie są pojedyncze, zatem wystąpienie takowego kłóci się z ideą błędu wykrywalnego kontrolą parzystości.

Źródła: [4] [17]

2.6. Autorska metoda

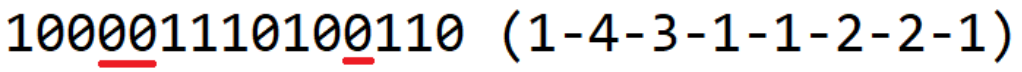
W czasie burzy mózgów przyszła nam do głowy następująca metoda: utworzyć listę przechowującą długości wystąpień łańcuchów tego samego bitu (100111 wygeneruje 1-2-3). Pozwoli to równocześnie wykryć błąd (różne listy długości dla sygnału wysłanego i odebranego) i miejsce/obszar wystąpienia błędu. Możemy zatem - najlepiej trzykrotnie - przesłać do nadawcy informację o tym, które bity wymagają (również najlepiej trzykrotnego) ponownego przesłania.

2.6.1. Prosty przykład

Jeśli wysłany sygnał 10001011 (1-3-1-1-2) po naniesieniu błędu na bity o indeksach 2-5 zamieni się w np. 10111011 (1-1-3-1-2). Łatwo zauważyć, że dla odebranego sygnału mamy inny niż wyjściowo wygenerowany szereg. Po analizie widać, że błąd jest na bitach 2-5 (pierwszy ciąg (1) jest identyczny, potem mamy różnice 3–>1, 1–>3), po czym reszta jest identyczna.

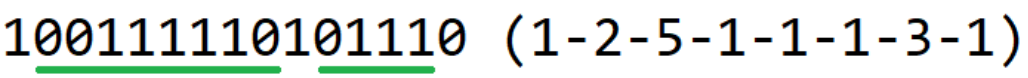
2.6.2. Większy przykład

100001110100110 (1-4-3-1-1-2-2-1) zamieni się w 100111110101110 (1-2-5-1-1-1-3-1).



Rys. 2. Wysłany sygnał i bity na których wystąpi błąd

Pierwszy bit OK, potem błędy (4–>2, 3–>5), potem 2 wyrazy OK, następnie błędy (2–>1, 2–>3) i ostatni bit OK. Widać zatem, że błędy są na bitach 1-8 (pierwszy burst) oraz 11-14. Bity te należy wysłać ponownie. Nie wszystkie wśród tych bitów były zakłamane ale są to przedziały na których wystąpiły burst error.



Rys. 3. Odebrany sygnał i bity do retransmisji

2.7. Złożoność

Główną wadą tego algorytmu jest to, że dodatkowe informacje do przesłania mają bardzo różną wielkość. Przypadkiem najbardziej optymistycznym jest sygnał złożony z samych 1 lub 0, jego wygenerowany sygnał kontrolny to (256) - jedna liczba którą można zapisać na 10 bitach. Przypadek najgorszy to sygnał 1010...1010, który wygeneruje (1-1- ... -1-1) co zajmuje 256 bitów. Jest do podwojenie sygnału do przesłania, równocześnie musimy pamiętać o przesłaniu indeksów błędów i retransmisji.

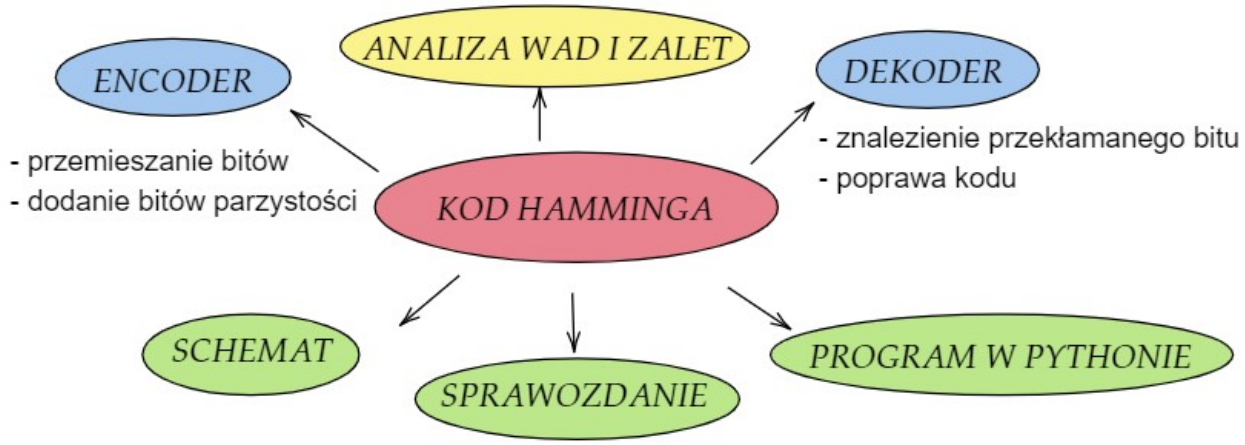
2.8. Podsumowanie naszej metody

Metoda ta wydaje się bardzo prosta do zaimplementowania w językach wysokopoziomowych typu Java, Python, lecz w przypadku vHDL jest to znacznie trudniejsze ze względu na zastosowanie kolekcji listy w tej metodzie. Dodatkowo poziom obciążenia łącza jest względnie wysoki oraz konieczna jest możliwość sygnału zwrotnego. Istnienie metod znacznie bardziej skomplikowanych matematycznie i ich stosowanie w praktyce przez dzisiejsze technologie implikuje, że są one znacznie bardziej wydajne. Nasza metoda zatem wydaje się być ciekawostką, uproszczonym modelem - tym samym czym dla współczesnych algorytmów sortowania jest sortowanie bąbelkowe.

3. Rozwijanie projektu

3.1. Wybranie sposobu realizacji zadania

Wcześniej zaproponowana przez nas metoda wiązała się ze zbyt dużym nadmiarem wysyłanych bitów, co przy niskiej szybkości transmisji danych sprawiało, że nie było to optymalne rozwiązanie problemu. Po ponownej analizie kodów korekcyjnych i uwzględnieniu sugestii z uwag do raportu etapu II zdecydowaliśmy się na metodę, która bazuje na kodzie Hamminga opisanego już wcześniej, mimo wszystko nadal brakowało nam infomacji, dlatego szukaliśmy dalej. [1] [2] [6]



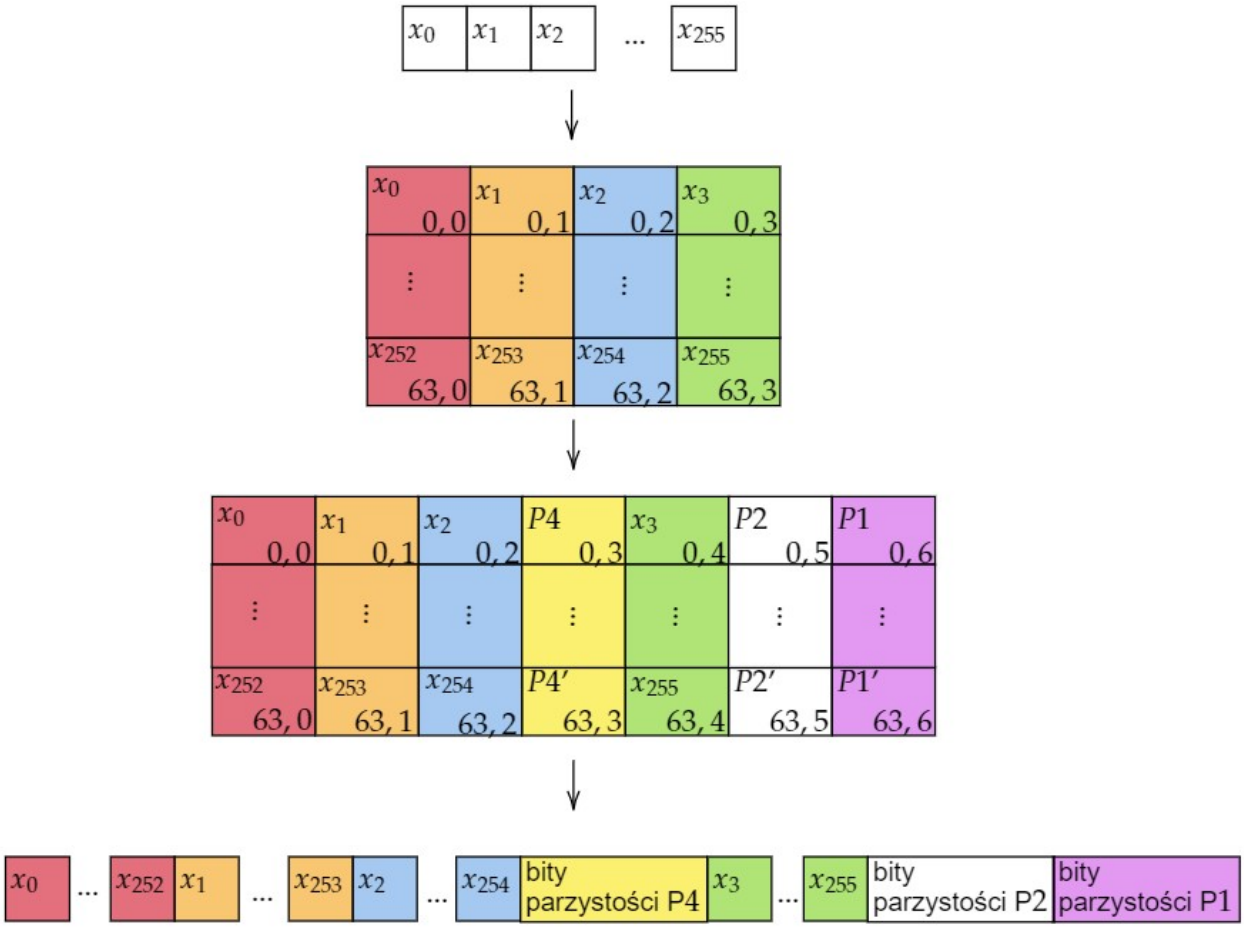
Rys. 4. Schemat burzy mózgów przed przystąpieniem do opisu metdory.

3.2. Koncepcja rozwiązania

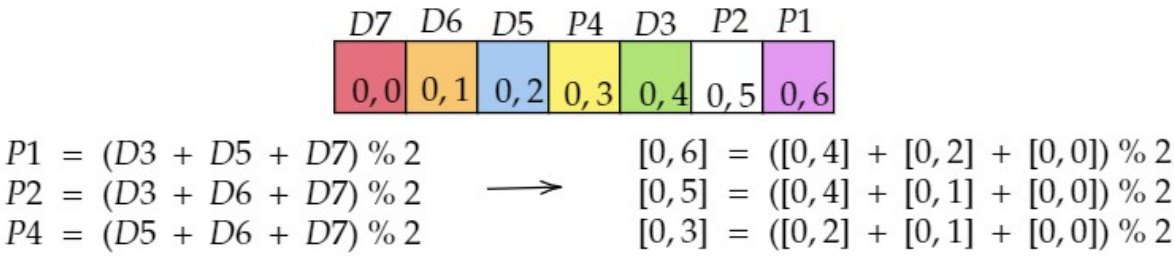
3.2.1. Encoder

Najpierw ze wstępnego sygnału tworzymy macierz (w finalnej realizacji nie będziemy jej tworzyć, jest ona nam tylko potrzebna do przedstawienia koncepcji i modelu referencyjnego) 64 wierszy pakietów 4−bitowych (rys 4.), które od tej pory będziemy nazywać bitami informacyjnymi i oznaczać jako D− data bit. Do tak przygotowanych wierszy dodajemy bity parzystości (rys 5.), które będziemy oznaczać jako P − parity bit, zgodnie z koncepcją, że znajdują się one na pozycjach będących potęgami dwójki (rys 6.).

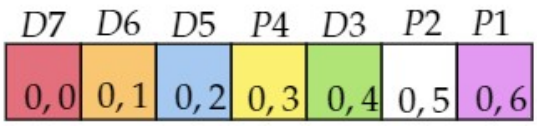
$[x, y]$ – *indeks elementu macierzy*



Rys. 5. Schemat encodera.

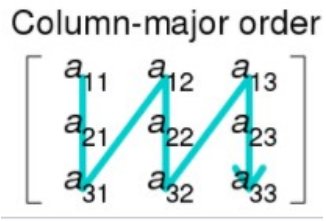


Rys. 6. Metoda obliczania bitów parzystości na przykładzie pierwszego wiersza.



Rys. 7. Ułożenie bitów w pakiecie na przykładzie pierwszego wiersza.

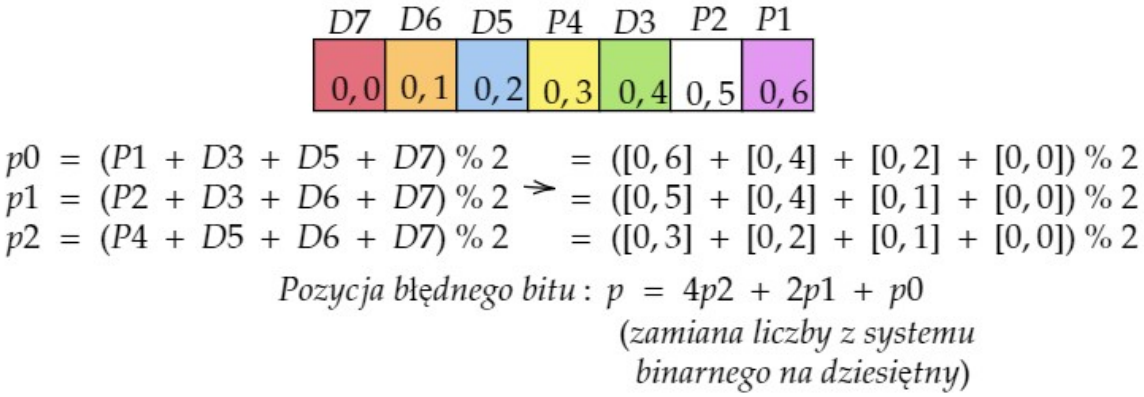
Kod Hamminga służy do korekcji pojedyńczych błędów, by zaimplementować go do naszego projektu przed wysłaniem sygnału z bitami parzystości musimy przemieszać go tak, by pojedyncze przekłamane bity z burst errors znajdowały się jak najdalej od siebie – stosujemy do tego metodę Column-major order(rys 7.) i wysyłamy sygnał (rys 4.).



Rys. 8. Przedstawienie metody column-major order [11].

3.2.2. Decoder

Z otrzymanego sygnału z błedami typu burst error, tworzymy macierz analogiczną do tej z encodera i na podstawie sposobu przedstawionego poniżej (rys 8.) dla każdego wiersza szukamy miejsca występowania błędu i jeśli istnieje to go poprawiamy.



Rys. 9. Wyznaczenie pozycji, na której znajduje się błąd na przykładzie pierwszego wiersza.

Wartość p	Pozycja błędu
0	None – brak błędu
1	P1 - [0,6]
2	P2 - [0,5]
3	D3 - [0,4]
4	P4 - [0,3]
5	D5 - [0,2]
6	D6 - [0,1]
7	D7 - [0,0]

Tabela 3. Wykrywanie pozycji błędu na przykładzie pierwszego wiersza.

Usuwamy bity parzystości, zamieniamy macierz w odpowiedni ciąg bitów i otrzymujemy finalny, poprawiony sygnał.

3.3. Model referencyjny.

file signal.txt nazwa pliku ze startowym sygnałem
filev2 coded_signal.txt nazwa pliku z sygnałem po przeróbkach i z błędami typu burst errors
filev3 decoded_signal.txt nazwa pliku z usuniętymi błędami

3.3.1. Generator

Ta prosta funkcja generuje nam przykładowy ciąg złożony z wyrazów {0,1}

```
1
2       def generator(file):
3           f = open(file, 'w')
4           for i in range(256):
5               f.write(str(random.randint(0, 1)))
6           f.close()
```

3.3.2. Encoder

W pierwszej części naszego kodera program otwiera plik (wiersz 2), który otrzymał od generatora i tworzy z niego listę zmiennych typu integer. Następnie tworzy listę 4 bitowych słów z kolejnych bitów z listy - lista list (wiersze 9-13). Na ich podstawie oraz w oparciu o kodowanie Hamminga program wyznacza bity kontrolne, które są umieszczane na odpowiednich miejscach, tworząc tym samym listę słów 7 bitowych.(wiersze 15-29). Ostatnim zadaniem kodera jest stworzenie z listy słów ciągu znaków (wiersze 31-35), który zostanie wysłany w formie pliku do dekodera (właśnie tu mogą pojawić się błędy). Program porządkuje bity w taki sposób jaki ukazuje to rysunek nr.5. Tak uporządkowaną listę program zamienia na tekst, który umieszczany jest w wysyłanym pliku (wiersze 37-40).

```
1  def hamming_encoder(file, filev2):
2      with open(file, "r") as f:
3          data = f.readline()
4      words = []
5      word_start = -4
6      word_end = 0
7      data = [int(b) for b in data]
8
9      for i in range(0, 64):
10         word_start += 4
11         word_end += 4
12         word = data[word_start:word_end]
13         words.append(word)
14
15     for word in words:
16         if (word[0] + word[1] + word[3]) % 2 == 0:
17             word.append(0)
18         else:
19             word.append(1)
20
21         if (word[0] + word[2] + word[3]) % 2 == 0:
22             word.append(0)
23         else:
24             word.append(1)
25
26         if (word[0] + word[1] + word[2]) % 2 == 0:
27             word.insert(3, 0)
28         else:
29             word.insert(3, 1)
30
31     coded_signal = []
```

```
32
33      for n in range(0, 7):
34          for word in words:
35              coded_signal.append(word[n])
36
37      with open(filev2, "w") as f:
38          for bit in coded_signal:
39              bit = str(bit)
40              f.write(bit)
```

3.3.3. Zaszumiacz

Funkcja ta nanosi błędy typu burst na przesyłany sygnał złożony z bitów informacyjnych i bitów parzystości. W modelu referencyjnym zakładamy wystąpienie jednego takiego błędu o długości 1-4. Miejsce wystąpienie błędu jest losowe. Fragment kodu nanoszący błędy:

```
1       for i in range(noise_length):
2
3           if signal_str[noise_start + i] == '1':
4               signal_str[noise_start + i] = '0'
5
6           elif signal_str[noise_start + i] == '0':
7               signal_str[noise_start + i] = '1'
```

3.3.4. Dekoder

Zadaniem dekodera jest ponowne utworzenie tym razem 7 bitowych słów, analogicznych do wierszy z tabeli z rys.5 (linie 1-10). Do każdego słowa zostaną utworzone zmienne tymczasowe: p4, p2, p1.(13-15). Jeżeli za pomocą konkatenacji utworzymy z nich wektor 3-bitowy a następnie zamienimy go na liczbę dziesiętną (16) to dostaniemy indeks (liczony od 1) bitu na którym wystąpił błąd (lub 0 dla braku błędów). Znając położenie błędu za pomocą szeregu instrukcji warunkowych if dokonujemy zamiany wartości danego bitu na przeciwną(18-40). Na koniec wystarczy "obciąć" bity kontrolne(41-43) oraz zapisać kolejne słowa do pliku wyjściowego(45-53).

```
1       def hamming_decoder(filev2, filev3):
2       with open(filev2, "r") as f:
3           data = f.readline()
4
5       words = []
6       for j in range(64):
7           word = []
8           for i in range(0, 448, 64):
9               word.append(int(data[i + j]))
10          words.append(word)
11
12      for word in words:
13          p4 = (word[0] + word[1] + word[2] + word[3]) % 2
14          p2 = (word[0] + word[1] + word[4] + word[5]) % 2
15          p1 = (word[0] + word[2] + word[4] + word[6]) % 2
16          wrong_bit = 4 * p4 + 2 * p2 + 1 * p1
17
18          if wrong_bit == 3:
19              if word[4] == 0:
20                  word[4] = 1
21              else:
22                  word[4] = 0
23
24          if wrong_bit == 5:
25              if word[2] == 0:
```

```
26                  word[2] = 1
27              else:
28                  word[2] = 0
29
30          if wrong_bit == 6:
31              if word[1] == 0:
32                  word[1] = 1
33              else:
34                  word[1] = 0
35
36          if wrong_bit == 7:
37              if word[0] == 0:
38                  word[0] = 1
39              else:
40                  word[0] = 0
41          del word[6]
42          del word[5]
43          del word[3]
44
45      coded_signal = []
46
47      for word in words:
48          for bit in word:
49              coded_signal.append(bit)
50
51      with open(filev3, "w") as f:
52          for bit in coded_signal:
53              f.write(str(bit))
```

3.3.5. Funkcja compare

Ta bardzo prosta funkcja ma za zadanie porównać plik z wyjściowym sygnałem do pliku otrzymanego po zdekodowaniu przesłanego sygnału. Porównuje on ich długości oraz kolumna po kolumnie szuka różnic w ich treści. W przypadku znalezienia błędu drukuje numer błędnej kolumny.

```
1
2
3  def compare(file, filev2):
4      f = open(file, 'r')
5      data_a = f.readline()
6      f.close()
7
8      g = open(filev2, 'r')
9      data_b = g.readline()
10     g.close()
11
12     if len(data_a) is not len(data_b):
13         print("R  ne D Ćugo Żci plik w")
14         return
15     for index, data in enumerate(data_a):
16         if data != data_b[index]:
17             print(f"Error in {index} column")
```

3.3.6. Funkcja główna

W funkcji main kolejno tworzymy przykładowy sygnał (2), kodujemy go(3), nanosimy szum(4), dekodujemy(5) i na koniec kontrolnie porównujemy orzymany na koniec sygnał z sygnałem przykładowym(6).

```
if __name__ == '__main__':
    generator("signal.txt")
    hamming_encoder("signal.txt", "coded_signal.txt")
    noise_generator("coded_signal.txt")
    hamming_decoder("coded_signal.txt", "decoded_signal.txt")
    compare("decoded_signal.txt", "signal.txt")
```

3.4. Narzędzia, które zostały użyte.

— Python – do realizacji modelu referencyjnego
— Mathcha.io – do stworzenia rysunków

4. Implementacja

4.1. Zadania do wykonania

W tym etapie nasz zespół skupił się na implementacji modułów encodera i decodera w języku Verilog HDL oraz sprawdzeniu ich poprawności. Pomocne przy tym zadaniu okazały modele referencyjne stworzone w etapie III oraz testbenche wykonane na potrzebę każdego z modułów.

4.2. Encoder

Stworzony przez nas moduł encodera przetwarza 4-bitową paczkę zgodnie z rys.6.
Kod Verilog [8]:

```
module encoder(
    input [3:0] data_in,
    output [6:0] data_out
    );

    wire p0,p1,p2;

    assign p0 = data_in[0] ^ data_in[1] ^ data_in[3];
    assign p1 = data_in[0] ^ data_in[2] ^ data_in[3];
    assign p2 = data_in[1] ^ data_in[2] ^ data_in[3];

    assign data_out = {data_in[3],data_in[2],data_in[1], p2, data_in[0],p1, p0};
endmodule
```

Test bench:

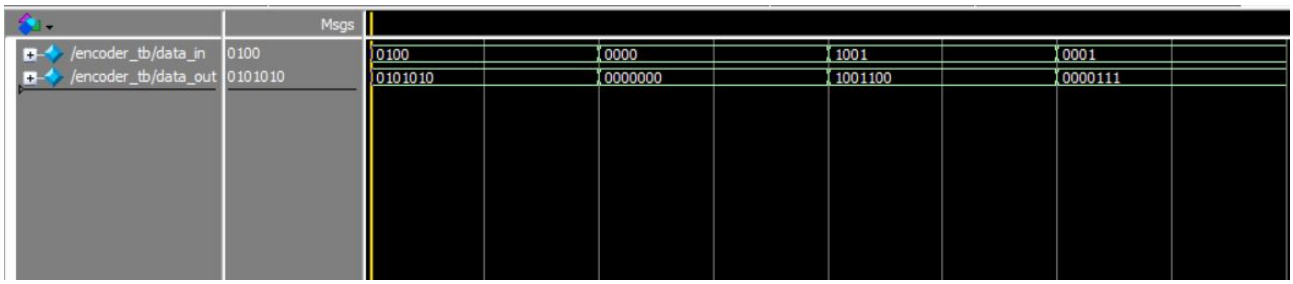
```
`timescale 1ns / 1ps
module encoder_tb();

reg [3:0] data_in;
wire [6:0] data_out;

encoder DUT(data_in, data_out);

initial
    begin
        data_in = 4'b100;
        #10;
        data_in = 4'b0;
        #10;
        data_in = 4'b1001;
```

```
16          #10;
17          data_in = 4'h1101;
18          #10;
19      end
20  endmodule
```

Przetestowaliśmy cztery pakiety:

Testowany pakiet (data_in)	Poprawny pakiet wyjściowy (data_out)
0100	0101010
0000	0000000
1001	1001100
1101	0000111

Poprawny pakiet wyjściowy został obliczony na podstawie algorytmu z rys. 6.
Symulacja Modelsim:



Rys. 10. Przebieg sygnałów - test bench

Wyniki symulacji zgadzają się z tymi z tabelki powyżej, co świadczy o poprawności wykonania modułu encodera.

4.3. Decoder

Kod decodera prezentuje się następująco:

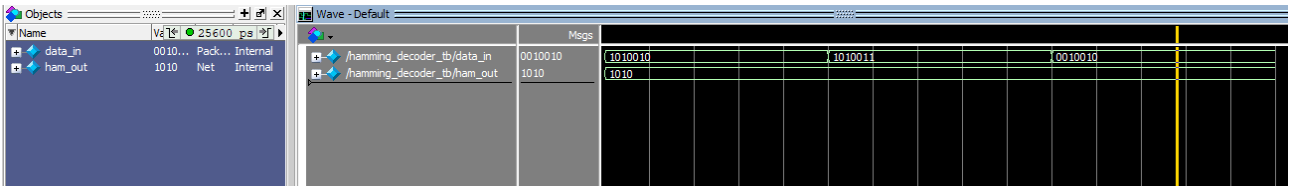
```
1  module hamming_decoder(
2      input [6:0] data_in,
3      output [3:0] ham_out
4      );
5
6      reg p0,p1,p2, p_new;
7          reg [3:0] p;
8
9          reg [6:0] out;
10
11         always@(*) begin
12                 p0 = data_in[0] ^ data_in[2] ^ data_in[4] ^ data_in[6];
13                 p1 = data_in[1] ^ data_in[2] ^ data_in[5] ^ data_in[6];
14                 p2 = data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6];
15
16                 p = (4*p2) + (2*p1) + (p0);
17
18                 out = data_in;
19
20                 if (p != 0) begin
```

```
21                          p_new = ~ out[p-1];
22                          out[p-1] = p_new;
23                  end
24
25
26          end
27          assign ham_out = {out[6],out[5],out[4],out[2]};
28  endmodule
```

Wejściami do decodera są "paczki" siedmiu bitów. W wierszach 12-17 znajduje się kod obliczający indeks bitu, na którym znajduje się błąd, zgodnie z działaniem kodu Hamminga. W wierszach 21-24 znajduje się instrukcja "if", która dokonuje zmiany błędnego bitu na poprawny. Następnie decoder wysyła właściwy kod - czyli poprawiony za pomocą kodu Hamminga oraz po usunięciu bitów parzystości.

Wykonaliśmy również testbench decodera oraz jego symulację w programie modelsim. Kod wraz z przebiegiem czasowymi widoczne są poniżej:

```
1  `timescale 1ns / 1ps
2  module hamming_decoder_tb();
3
4  reg [6:0] data_in;
5  wire [3:0] ham_out;
6
7  Hamming DUT(data_in,ham_out);
8
9  initial
10     begin
11
12                         //poprawny
13                         data_in = 7'b1010010;
14                         #10;
15                         //przeklamany kontrolny (ostatni)
16                         data_in = 7'b1010011;
17                         #10;
18
19                         //przeklamany informacyjny      (pierwszy)
20                         data_in = 7'b0010010;
21                         #10;
22
23     end
24 endmodule
```



Rys. 11. Symulacja testbench'a w modelsimie

Wyniki symulacji są zgodne z naszymi przewidywaniami oraz potwierdzają poprawność działania decodera, ponieważ testbench został skonstruowany w taki sposób aby druga i trzecia wartość "wchodząca" do dekodera była przekłamaniem pierwszej na jednym bicie (druga wartość bit kontrolny, trzecia bit informacyjny). Na wyjściu natomiast widzimy cały czas jedną wartość.

4.4. Użyte narzędzia

— Quartus – do realizacji modułów encodera, decodera i testbenchów
— ModelSim – do symulacji przebiegów czasowych

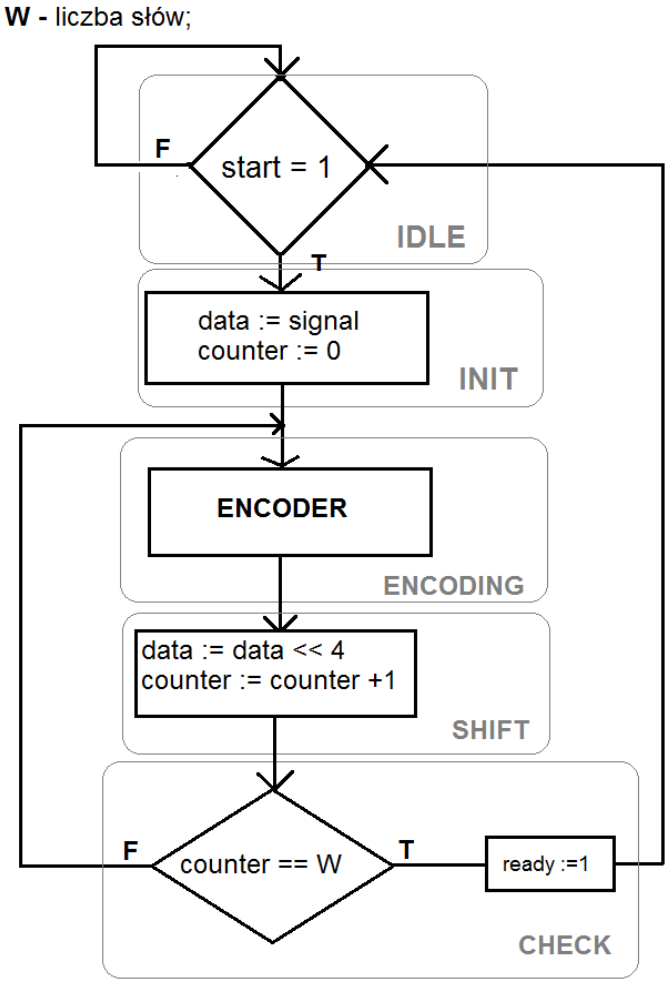
5. Uruchomienie

5.1. Zadania do wykonania

W tym etapie nasz zespół miał za zadanie uruchomić zaprojektowany system i sprawdzić jego działanie. Aby to zrobić, musieliśmy uzupełnić system o to, czego nie udało nam się zaimplementować w etapie IV, czyli o mechanizm który przemiesza zakodowane bity w sposób widoczny na rys. 5. (część nadawcza) i z powrotem przemiesza, aby umożliwić zdekodowanie (część odbiorcza). Ustaliliśmy, że podzielimy nasz system na dwa osobne moduły: transmitter i receiver. Ich implementacja znajduje się w kolejnych podrozdziałach.

5.2. Transmitter

Stworzony przez nas moduł transmitter (dalej nazywany nadajnikiem) dzieli wejściowy ciąg bitów na słowa 4-bitowe, koduje je przy użyciu zaimplementowanego w poprzednim etapie encodera i na wyjściu podaje ciąg przemieszany w sposób widoczny na rys. 5. Udało się to dzięki zastosowaniu, algorytmicznego układu sekwencyjnego ze zintegrowaną ścieżką przepływu danych ASMD. Z powodu ograniczonej ilości pinów I/O nie mogliśmy sobie pozwolić na podanie na wejściu 256 bitów, dlatego transmitowaną paczkę będziemy dzielić na mniejsze, których długość zależy od parametru. Algorytm, kod w verilogu, raport z kompilacji i maksymalna częstotliwość pracy są widoczne poniżej.



Rys. 12. Algorytm działania nadajnika

```
1  module transmitter
2  #(
3          parameter N = 128, //Wielkosc paczki wejsciowej
4          parameter W = N/4 //Ilosc slow po podziale
5  )
6  (
7          input rst,
8          input clk,
9          input ena,
10         input start,
11         input [N-1:0] signal,
12         output reg [(W*7)-1:0] coded_signal,
```

```
13          output reg rdy
14  );
15          localparam [2:0] idle = 3'h0,
16                           init = 3'h1,
17                           encoding = 3'h2,
18                           shift = 3'h3,
19                           check = 3'h4;
20
21          reg [2:0] state_reg, state_next;
22          reg rdy_next;
23
24          reg [(W*7)-1:0] coded_signal_next;
25          reg [N-1:0] data_reg, data_next;
26          reg [6:0] counter_reg, counter_next;
27
28
29          //State register
30          always@(posedge clk or posedge rst) begin
31                  if (rst)
32                  begin
33                          state_reg <= idle;
34                          rdy <= 1'b0;
35                  end
36                  else if (ena)
37                  begin
38                          state_reg <= state_next;
39                          rdy <= rdy_next;
40                  end
41          end
42
43
44          //Registers
45          always@(posedge clk or posedge rst) begin
46                  if (rst)
47                  begin
48                          data_reg <= 0;
49                          counter_reg <= 0;
50                          coded_signal <= 0;
51                  end
52                  else if (ena)
53                  begin
54                          data_reg <= data_next;
55                          counter_reg <= counter_next;
56                          coded_signal <= coded_signal_next;
57                  end
58          end
59
60
61          //Encoder module
62          wire[3:0] uncoded_word;
63          wire[6:0] coded_word;
64          assign uncoded_word = data_reg[N-1:N-4];
65          encoder encoder_inst(
66          .data_in (uncoded_word),
67          .data_out (coded_word)
68          );
```

```
69
70
71          // Next state logic
72          always@(*)
73                  case(state_reg)
74                          idle : begin
75                                  if (start) state_next = init;
76                                  else state_next = idle;
77                          end
78
79                          init : state_next = encoding;
80
81                          encoding : state_next = shift;
82
83                          shift : state_next = check;
84
85                          check : if (counter_reg == W)
86                                                  state_next = idle;
87                                          else
88                                                  state_next = encoding;
89
90                          default: state_next = init;
91                  endcase
92
93
94          //Microoperation logic
95          always@(*) begin
96                  data_next = data_reg;
97                  counter_next = counter_reg;
98                  coded_signal_next = coded_signal;
99                  rdy_next = 1'b0;
100
101                 case(state_reg)
102                         init : begin
103                                 data_next = signal;
104                                 counter_next = 0;
105                         end
106
107                         encoding : begin
108                                 coded_signal_next[(W*7) - 1 - counter_reg] = coded_word[6]
109                                 coded_signal_next[(W*6) - 1 - counter_reg] = coded_word[5]
110                                 coded_signal_next[(W*5) - 1 - counter_reg] = coded_word[4]
111                                 coded_signal_next[(W*4) - 1 - counter_reg] = coded_word[3]
112                                 coded_signal_next[(W*3) - 1 - counter_reg] = coded_word[2]
113                                 coded_signal_next[(W*2) - 1 - counter_reg] = coded_word[1]
114                                 coded_signal_next[(W*1) - 1 - counter_reg] = coded_word[0]
115                         end
116
117                         shift : begin
118                                 counter_next = counter_reg + 1;
119                                 data_next = data_reg << 4;
120                         end
121
122                         check : if (counter_reg == W) rdy_next = 1;
123
124                         default : ;
```

```
125                     endcase
126                 end
127 endmodule
```

Flow Status	Successful - Sun Jan 24 20:34:34 2021
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	transmitter
Top-level Entity Name	transmitter
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	1,096 / 114,480 (< 1 %)
Total registers	365
Total pins	357 / 529 (67 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

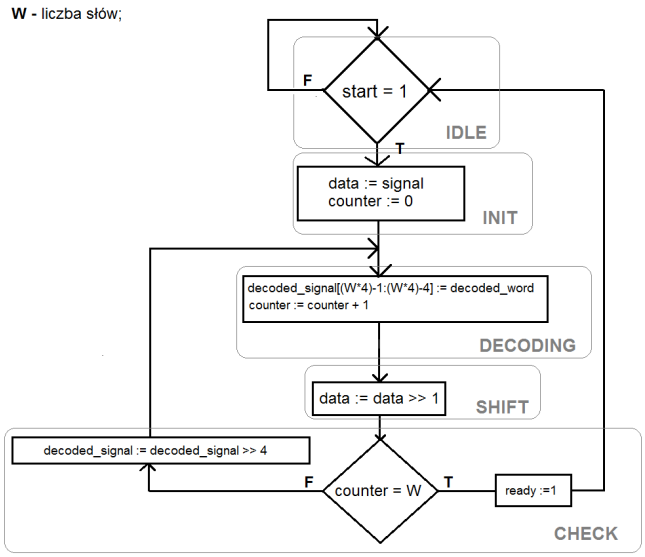
Rys. 13. Raport z kompilacji nadajnika

	Fmax	Restricted Fmax	Clock Name	Note
1	246.24 MHz	246.24 MHz	clk	

Rys. 14. Maksymalna częstotliwość pracy nadajnika

5.3. Receiver

Stworzony przez nas moduł receiver (dalej nazywany odbiornikiem) przyjmuje na wejściu zakodowany i przemieszany ciąg a na wyjściu podaje odkodowaną oryginalną wiadomość. Podobnie jak nadajnik został stworzony przy użyciu ASMD i wykorzystuje napisany wcześniej dekoder. Algorytm, kod w verilogu, raport z kompilacji i maksymalna częstotliwość pracy są widoczne poniżej.



Rys. 15. Algorytm działania odbiornika

```
1  module receiver
2  #(
3          parameter N = 224, //Wielkosc strumienia wejsciowego
4          parameter W = N/7 //Ilosc slow po podziale
5  )
6  (
7          input rst,
8          input clk,
9          input ena,
10         input start,
11         input [N-1:0] signal,
12         output reg [(W*4)-1:0] decoded_signal,
13         output reg rdy
14 );
15
16         localparam [2:0] idle = 3'h0,
17                                                 init = 3'h1,
18                                                 decoding = 3'h2,
19                                                 shift = 3'h3,
20                                                 check = 3'h4;
21
22         reg [2:0] state_reg, state_next;
23         reg rdy_next;
24
25         reg [(W*4)-1:0] decoded_signal_next;
26         reg [N-1:0] data_reg, data_next;
27         reg [6:0] counter_reg, counter_next;
28
29
30         //Stage register
31         always@(posedge clk or posedge rst) begin
32                 if (rst)
33                 begin
34                         state_reg <= idle;
35                         rdy <= 0;
36                 end
37                 else if (ena)
38                 begin
39                         state_reg <= state_next;
40                         rdy <= rdy_next;
41                 end
42         end
43
44
45         //Register
46         always@(posedge clk or posedge rst) begin
47                 if(rst) begin
48                         data_reg <= 0;
49                         counter_reg <= 0;
50                         decoded_signal <= 0;
51                 end
52
53                 else if (ena)
54                 begin
55                         counter_reg <= counter_next;
56                         data_reg <= data_next;
```

```
57                      decoded_signal <= decoded_signal_next;
58              end
59      end
60
61
62      //Decoder module
63      wire [6:0] coded_word;
64      wire [3:0] decoded_word;
65      assign coded_word = {data_reg[N-W], data_reg[N-(2*W)], data_reg[N-(3*W)],
66                                                              data_reg[N-(4*W)], data_reg
67      decoder decoder_inst(
68      .data_in (coded_word),
69      .data_out (decoded_word)
70      );
71
72
73      // Next state logic
74      always@(*)
75              case(state_reg)
76                      idle : begin
77                              if (start) state_next = init;
78                              else state_next = idle;
79                      end
80
81                      init : state_next = decoding;
82
83                      decoding : state_next = shift;
84
85                      shift : state_next = check;
86
87                      check : if (counter_next == W)
88                                                      state_next = idle;
89                                      else
90                                                      state_next = decoding;
91
92                      default: state_next = idle;
93              endcase
94
95
96      //Microoperation logic
97      always@(*) begin
98              data_next = data_reg;
99              counter_next = counter_reg;
100             decoded_signal_next = decoded_signal;
101             rdy_next = 1'b0;
102
103             case(state_reg)
104                     init : begin
105                             data_next = signal;
106                             counter_next = 0;
107                     end
108
109                     decoding : begin
110                             decoded_signal_next[(W*4)-1:(W*4)-4] = decoded_word;
111                             counter_next = counter_reg + 1;
112                     end
```

```
113
114                             shift : begin
115                                     data_next = data_reg >> 1;
116                             end
117
118                             check : if (counter_next == W) rdy_next = 1;
119                                                 else   decoded_signal_next = decoded_signal >> 4;
120
121
122                             default : ;
123                     endcase
124             end
125 endmodule
```

Flow Status	Successful - Sun Jan 24 20:48:32 2021
Quartus Prime Version	19.1.0 Build 670 09/22/2019 SJ Lite Edition
Revision Name	receiver
Top-level Entity Name	receiver
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	395 / 114,480 (< 1 %)
Total registers	365
Total pins	357 / 529 (67 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Rys. 16. Raport z kompilacji odbiornika

	Fmax	Restricted Fmax	Clock Name	Note
1	288.35 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Rys. 17. Maksymalna częstotliwość pracy odbornika

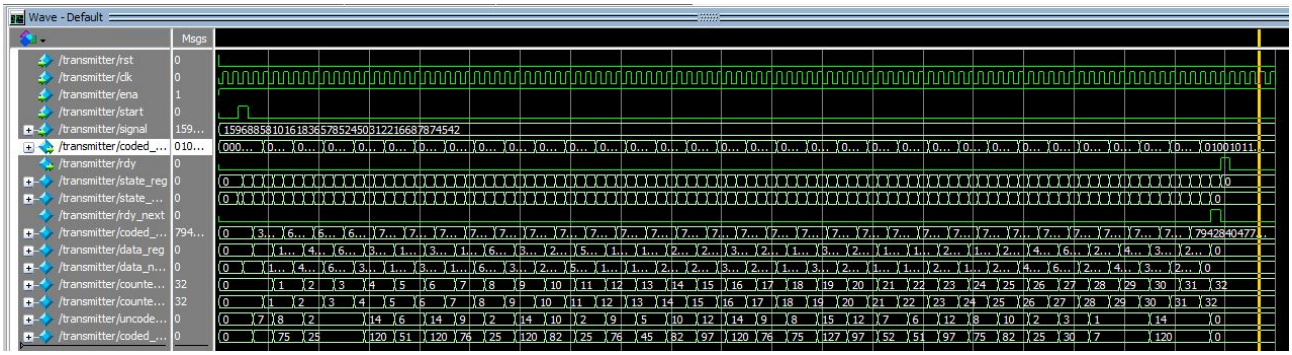
5.4. Część główna

Po pomyślnym zaimplementowaniu nadajnika i odbiornika mogliśmy w końcu przejść do uruchomienia naszego systemu. Za pomocą modeli referencyjnych opisanych w III rozdziale wygenerowaliśmy losowy ciąg 128 bitów i zakodowaliśmy go. Przy pomocy funkcji compare będziemy sprawdzać czy na wyjściu nadajnika i odbiornika znajdują się właściwe ciągi.

5.4.1. Test nadajnika

Działanie nadajnika sprawdziliśmy w ModelSimie przy użyciu poniższego skryptu testującego:

```
1 restart -nowave -force
2 add wave -radix unsigned *
3 force clk 0 0, 1 10 -r 20
4 force rst 1 0, 0 1
5 force ena 1 0
6 force start 0 0, 1 40, 0 60
7 force signal //tutaj 128 bitow z generatora
8 run 2.1 ns
```



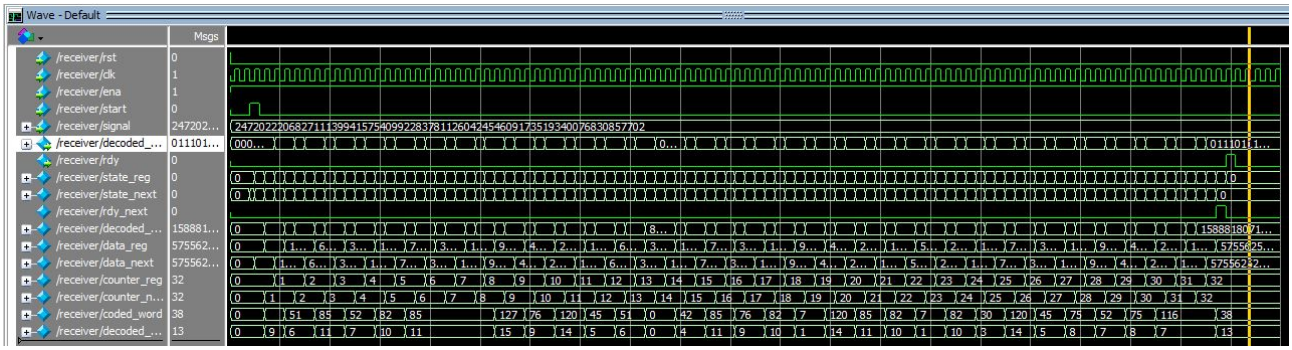
Rys. 18. Symulacja działania nadajnika

Otrzymany na wyjściu (coded_signal) ciąg skopiowaliśmy i porównaliśmy z zakodowanym w Pythonie. Okazało się, że są równe, zatem nadajnik działa poprawnie.

5.4.2. Test dbiornika

Działanie odbiornika sprawdziliśmy w ModelSimie przy użyciu poniższego skryptu testującego:

```
1 restart -nowave -force
2 add wave -radix unsigned *
3 force clk 0 0, 1 10 -r 20
4 force rst 1 0, 0 1
5 force ena 1 0
6 force start 0 0, 1 40, 0 60
7 force signal //tutaj 224 bity z encodera
8 run 2.1 ns
```



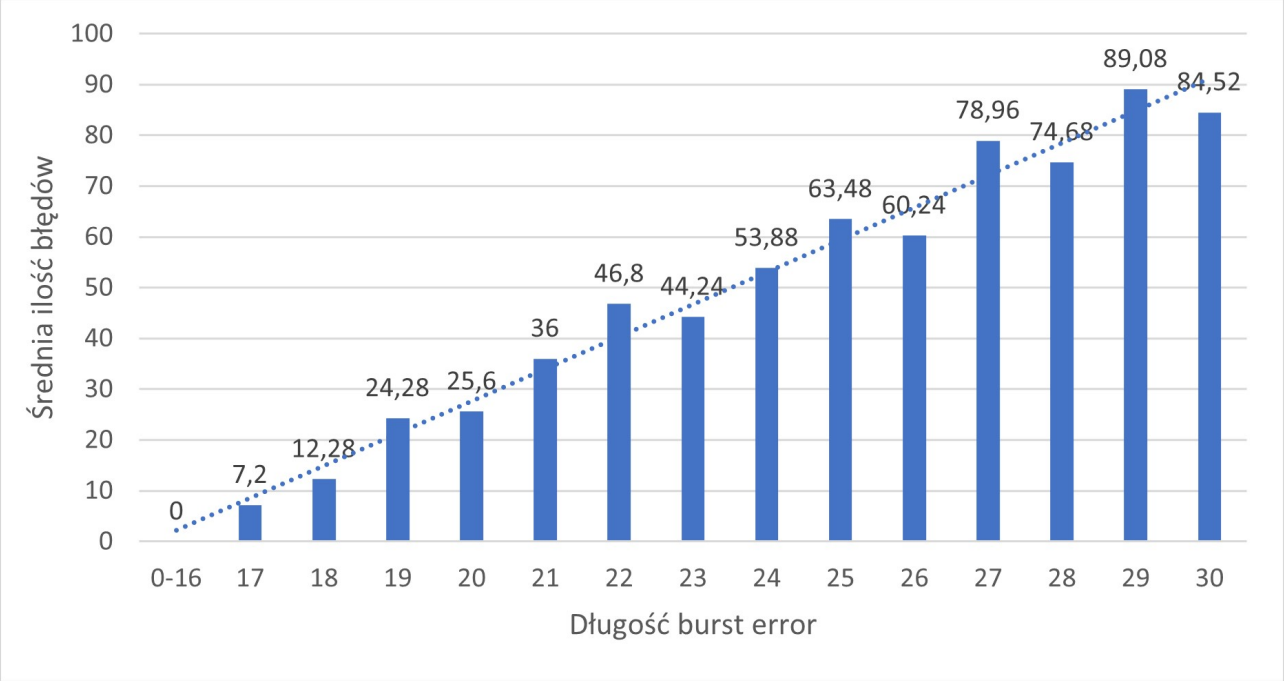
Rys. 19. Symulacja działania odbiornika

Otrzymany na wyjściu (decoded_signal) ciąg skopiowaliśmy i porównaliśmy z pierwotnym wygenerowanym w Pythonie. Okazało się, że są równe zatem odbiornik również działa poprawnie. Zanim przeszliśmy do poważnych testów z nanoszeniem błędów zmieniliśmy w skrypcie testującym ręcznie kilka losowych bitów. Na wyjściu ponownie otrzymaliśmy poprawny ciąg, co na tym etapie pozwala nam stwierdzić, że nasz system zabezpiecza transmisję przed pojedynczymi błędami.

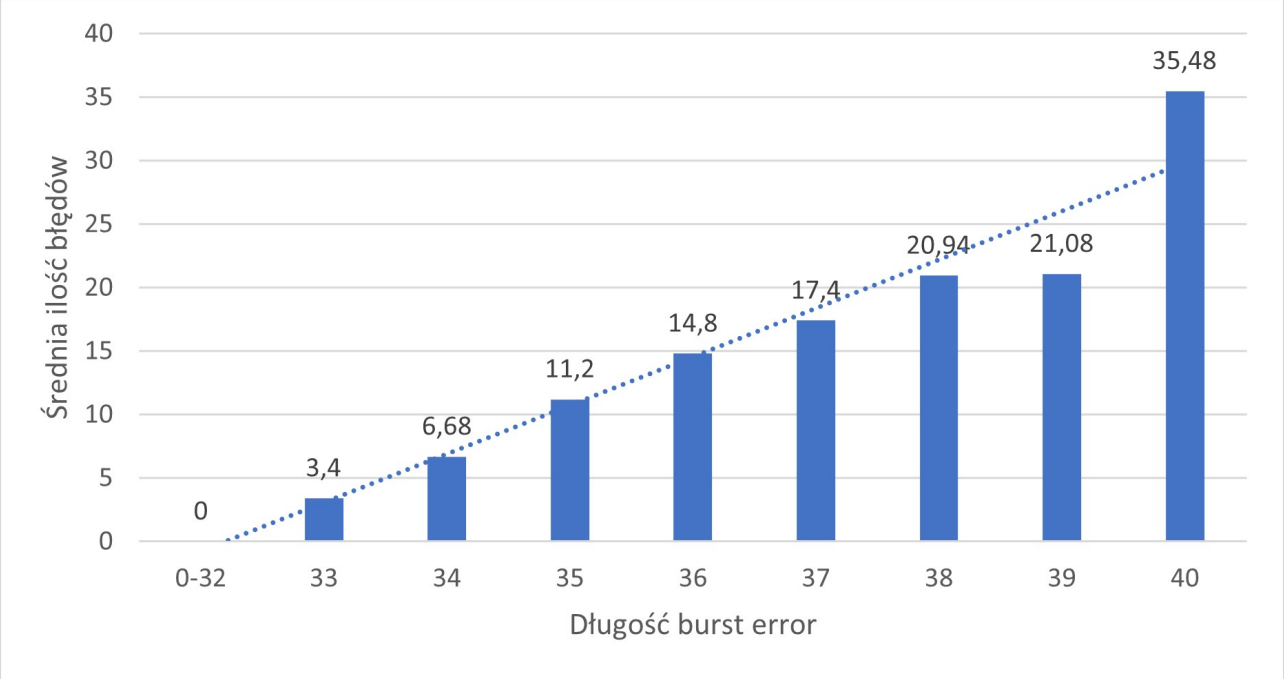
5.4.3. Testy zabezpieczenia transmisji

Przeprowadziliśmy serie symulacji i obliczyliśmy średnią ilość występujących błędów względem długości burst error i odległości między nimi, gdy 256-bitowy ciąg zostanie podzielony na moduły o wielkości 64 lub 128 bitów.

a) Dla pojedynczego burst error.

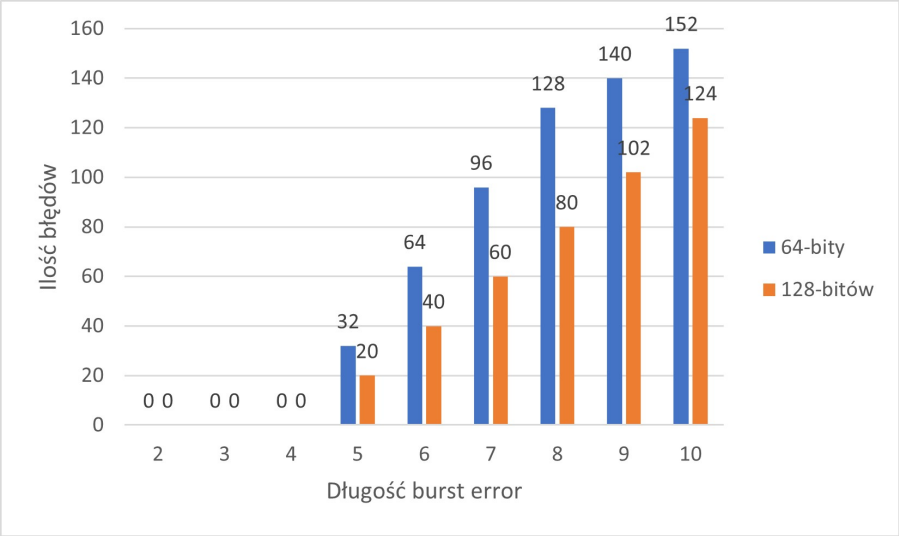


Rys. 20. Wykres przy podziale transmitowanego ciągu bitów na moduły o wielkości 64 bitów

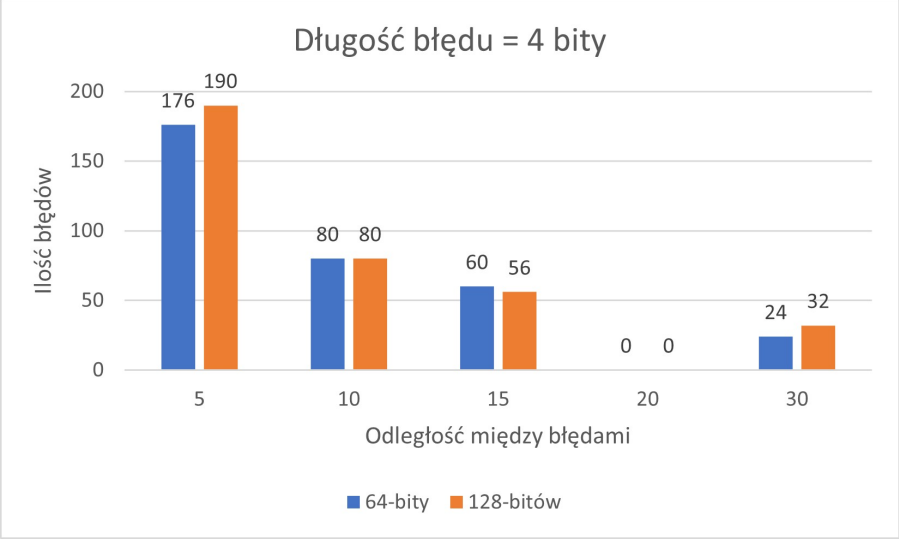


Rys. 21. Wykres przy podziale transmitowanego ciągu bitów na moduły o wielkości 128 bitów

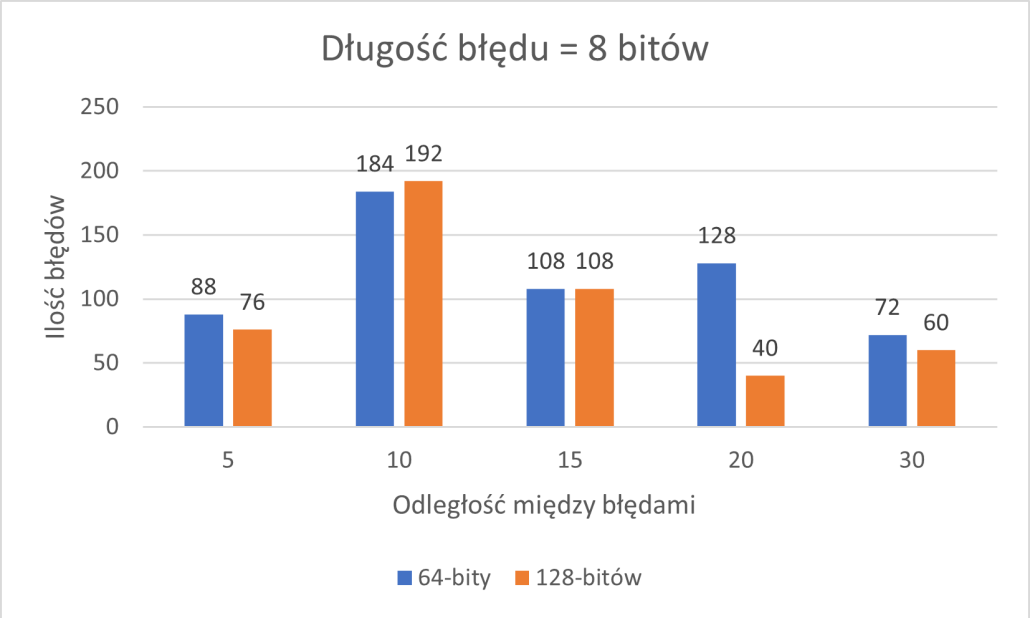
b) Dla błędów burst error o określonej długości i odległości między sobą.



Rys. 22. Wykres dla odległości 20 bitów między kolejnymi błędami



Rys. 23. Wykres dla długości błędu równej 4 bity



Rys. 24. Wykres dla długości błędu równej 8 bitów

6. Podsumowanie

Przez cały okres realizacji projektu, mimo występujących trudności sukcesywnie zbliżaliśmy się do celu. Na przełomie etapu IV i V murem nie do przeskoczenia wydawało się zaimplementowanie interlievera, jednak finalnie udało nam się dzięki zastosowaniu ASMD. Działanie naszego systemu uznajemy za zadowalające i żałujemy, że z powodu sytuacji epidemiologicznej nie dane nam było go przetestować na fizycznych urządzeniach na wydziale. Wszyscy są zgodni co do tego, że projekt był rozwijający. Znacznie rozwinął nasze umiejętności posługiwania się językiem opisu sprzętu Verilog, pracy w grupie i myślenia projektowego oraz zainteresował nas zagadnieniem systemów cyfrowych.

Literatura

[1] N. Academy. Hamming Code — Error Correction. https://www.youtube.com/watch?v=wbH2VxzmoZk&feature=share&fbclid=IwAR3eCeYSzd7PlIMz5kSjbf1LTtZRjoyvLkmH9bNhgYENoYDMTaKWuf2G39M.

[2] N. Academy. Hamming Code — Error detection. https://www.youtube.com/watch?v=1A_NcXxdoCc&feature=share&fbclid=IwAR3sP2BRvYRRsqkps_dOwN_-GBluKHO3EA2WzgAgPMw9eM10IKQHhuuogOU.

[3] CDRinfo. Jak działa CRC, type = url - https://www.cdrinfo.pl/slownik/crc-136, note = https://www.cdrinfo.pl/slownik/crc-136.

[4] S. elekrtoniki. Do czego służy bit parzystości. https://www.serwis-elektroniki.com.pl/do-czego-sluzy-bit-parzystosci-ang-parity-bit/.

[5] M. Gębala. Detekcja i korekcja błędów. Kody Hamminga, 2010. https://zagorski.im.pwr.wroc.pl/courses/kiki_2010/wyklad11.pdf.

[6] T. P. I. L. Mr. Arnab Chakraborty. Error Detection and Correction with Hamming Code. https://www.youtube.com/watch?v=tcEa2dW3Ajg&t=289s.

[7] A. Tribe. Double Diamond, 2016. https://www.researchgate.net/figure/LX-Double-Diamond-by-Academic-Tribe-2016_fig1_313914239.

[8] VLSICoding. Verilog Code for (7,4) Systematic Hamming Encoder. https://vlsicoding.blogspot.com/2017/06/verilog-code-for-74-systematic-hamming-encoder.html?fbclid=IwAR3x957DNoBMdxVh8dkwK5ZjrhOz2wm3y_jyA9QuR5o1HAmDOVCqfNmmAeY.

[9] B. wiedzy. Kodowanie korekcyjne. https://www.bazawiedzy24.pl/download/ecc.pdf.

[10] Wikipedia. Burst error. https://pl.wikipedia.org/wiki/Kod_Hamminga.

[11] Wikipedia. Column-major-order. https://en.wikipedia.org/wiki/Burst_error-correcting_code#Interleaved_codes.

[12] Wikipedia. Interleaving. https://www.tutorialspoint.com/what-is-interleaving.

[13] Wikipedia. Kod Hamminga.
[14] Wikipedia. Kod Splotowy. https://pl.wikipedia.org/wiki/Kod_splotowy#Dekodowanie_kod%C3%B3w_splotowych.
[15] Wikipedia. Kodowanie kanałowe. https://pl.wikipedia.org/wiki/Kodowanie_korekcyjne.
[16] Wikipedia. Kodowanie korekcyjne. https://pl.wikipedia.org/wiki/Kodowanie_korekcyjne.
[17] Wikipedia. Kontrola parzystości. https://pl.wikipedia.org/wiki/Kontrola_parzysto%C5%9Bci.