

Projekt Systemy Cyfrowe

Zespół nr 7

Szymon Boniuk

Adam Witczak

Piotr Świtaj

Jakub Celiński

Bartosz Skrzypczak

Politechnika Warszawska, Wydział EiT

7 czerwca 2025

Historia zmian

Wersja	Data	Autor	Opis zmian
1.0	10.03.2025	SB, AW, PŚ, JC, BS	Pierwsza wersja raportu Etapu 1
1.1	10.03.2025	SB, AW, PŚ, JC, BS	Dodano informacje, czym jest Design Thinking
1.2	10.03.2025	SB, AW, PŚ, JC, BS	Ustalono sposób zarządzania projektem
1.3	10.03.2025	SB, AW, PŚ, JC, BS	Ustalono zbiór narzędzi niezbędnych do wykonania projektu
1.4	11.03.2025	SB, AW, PŚ, JC, BS	Wstępne spotkanie projektowe
1.5	20.03.2025	SB, AW, PŚ, JC, BS	Wprowadzenie korekt na postawie uwag koordynatora projektu
1.6	01.04.2025	SB, AW, PŚ, JC, BS	Wstępna realizacja etapu II projektu
1.7	02.04.2025	SB, AW, PŚ, JC, BS	Poprawki w realizacji etapu II projektu
1.8	22.04.2025	SB, AW, PŚ, JC, BS	Realizacja etapu III zadania projektowego
1.9	17.05.2025	SB, AW, PŚ, JC, BS	Realizacja etapu IV zadania projektowego
1.10	07.06.2025	SB, AW, PŚ, JC, BS	Realizacja etapu V zadania projektowego

Spis treści

Historia zmian	1
1. Wstęp	3
1.1. Cel projektu	3
1.2. Etapy realizacji	3
2. Organizacja prac	3
2.1. Design Thinking	3
2.2. Zarządzanie projektem	4
2.2.1. Narzędzia	5
3. Informacje podstawowe	5
3.1. Istniejące rozwiązania	5
3.2. Podstawy przetwarzania sygnałów	6
3.2.1. Jak działa transformata Fouriera?	6
3.2.2. DFT i FFT	6
3.2.3. Różnice między transformatą Fouriera, a FFT	7
4. Koncepcja	7
4.1. Mapa myśli	7
4.2. Schemat blokowy rozwiązania	8
4.3. Opis schematu blokowego	8
4.4. Referencyjny kod w języku Python	9
4.5. Kod sprawdzający w MATLAB	9
4.6. Testowanie kodu referencyjnego	11
4.6.1. Wynik działania kodu referencyjnego napisanego w języku Python	11
4.6.2. Wynik analizy sygnału w analizatorze widma online	11
4.6.3. Wynik działania kodu sprawdzającego MATLAB	12
4.6.4. Podsumowanie testów	13
5. Implementacja	13
5.1. Finalna realizacja systemu w języku Python	13
5.2. Testowanie finalnego kodu w języku Python	17
5.2.1. Wynik działania finalnego kodu w języku Python	17
5.2.2. Wynik działania kodu testującego Matlab	17
5.2.3. Wynik działania analizatora widma online	18
5.3. Implementacja zadania w języku Verilog	18
5.3.1. Założenia projektowe - FPGA	18
5.3.2. Schemat postępowania projektowego	18
5.3.3. Omówienie realizacji poszczególnych modułów	19
5.3.4. Symulacja projektu w programie Modelsim oraz jej wyniki	29
5.3.5. Analiza wyników uzyskanych z FFT w Verilog	33
5.3.6. Porównanie widm wygenerowanych przy pomocy kodu referencyjnego z pliku .wav oraz widma po obliczeniu FFT przy pomocy Verilog	35
5.3.7. Zasoby niezbędne do działania projektu	35
6. Uruchomienie	37
7. Podsumowanie	38
7.1. Wnioski	38
Literatura	39

1. Wstęp

1.1. Cel projektu

Celem projektu jest zaprojektowanie i realizacja systemu sprzętowo-programowego, który pozwoli w jak najkrótszym czasie i jak najdokładniej przebadać próbki sygnału odebranego z lazika marsjańskiego w celu wykrycia zakresu fal, na których odbywa się potencjalna transmisja.

1.2. Etapy realizacji

Niniejszy raport będzie w sposób przyrostowy dokumentował realizację projektu związanego z przedmiotem Systemy Cyfrowe w semestrze 25L. Poszczególne etapy realizacji projektu obejmują:

- I Etap wstępny – stworzenie zespołu i organizacja warsztatu pracy,
- II Etap zdobywania informacji – analiza literatury, istniejących metod, zebranie wiedzy teoretycznej związanej z tematem projektu,
- III Etap opracowania koncepcji – szukanie rozwiązań, najlepiej sprawdzi się proces burzy mózgów (mapy myśli), opracowanie koncepcji rozwiązania na podstawie zdobytej wiedzy, opracowanie prostego modelu referencyjnego (Python, MATLAB/GNU Octave, itp) i danych do testowania
- IV Etap implementacji – na tym etapie rozwijamy i rozbudowujemy koncepcje projektowe docelowego systemu, modelujemy elementy systemu w HDL, weryfikujemy funkcjonalnie, integrujemy i oceniamy prototypy,
- V Etap uruchomienia – wdrożenie projektu, uruchomienie na docelowej platformie, przetestowanie według wcześniej opracowanych scenariuszy testowych.

2. Organizacja prac

Prace nad projektem będą przebiegały według metody projektowej "Design Thinking" opisanej w rozdziale 2.1.

Zostało zorganizowane spotkanie członków grupy mające na celu dokładne zapoznanie z założeniami oraz wstępny podział obowiązków. Zostały rozdzielone również role w zespole.

Każdy z uczestników projektu wyjściowo powinien wykonać około 20 % pracy w trakcie każdego z etapów.

2.1. Design Thinking

Design Thinking [1] [2] to iteracyjna metoda rozwiązywania problemów, która koncentruje się na użytkowniku i jego potrzebach. Jest to proces kreatywnego myślenia, który łączy analityczne podejście z intuicją projektową, umożliwiając generowanie innowacyjnych rozwiązań.

Głównymi etapami Design Thinking są:

1. Empatyzacja (Empathize)

Polega na dogłębny zrozumieniu potrzeb użytkowników. Wymaga prowadzenia badań, obserwacji i rozmów z użytkownikami, aby poznać ich oczekiwania, problemy i motywacje. Wykorzystuje techniki takie jak wywiady, mapy empatii czy shadowing (obserwacja użytkownika w naturalnym środowisku).

2. Definiowanie problemu (Define)

Po zebraniu informacji z etapu empatyzacji, formułuje się konkretną definicję problemu. Umożliwia to precyjne określenie wyzwania, na które należy znaleźć rozwiązanie. W tym etapie pomocne są techniki, takie jak analiza przyczynowo-skutkowa czy budowanie person użytkowników.

3. Generowanie pomysłów (Ideate)

Proces twórczy, w którym uczestnicy generują jak najwięcej możliwych rozwiązań. Ważne jest myślenie nieszablonowe i odrzucenie ograniczeń. Techniki wykorzystywane na tym etapie to m.in. burza mózgów, SCAMPER (modyfikowanie istniejących pomysłów), mapowanie myśli czy metoda 6 kapeluszy myślowych de Bono.

4. Prototypowanie (Prototype)

Polega na tworzeniu uproszczonych wersji rozwiązań, które można szybko testować i modyfikować. Prototypy mogą mieć różne formy, np. rysunki, makiety, modele 3D czy wersje testowe oprogramowania. Celem jest szybkie wykrycie ewentualnych błędów i wprowadzenie poprawek na wczesnym etapie.

5. Testowanie (Test)

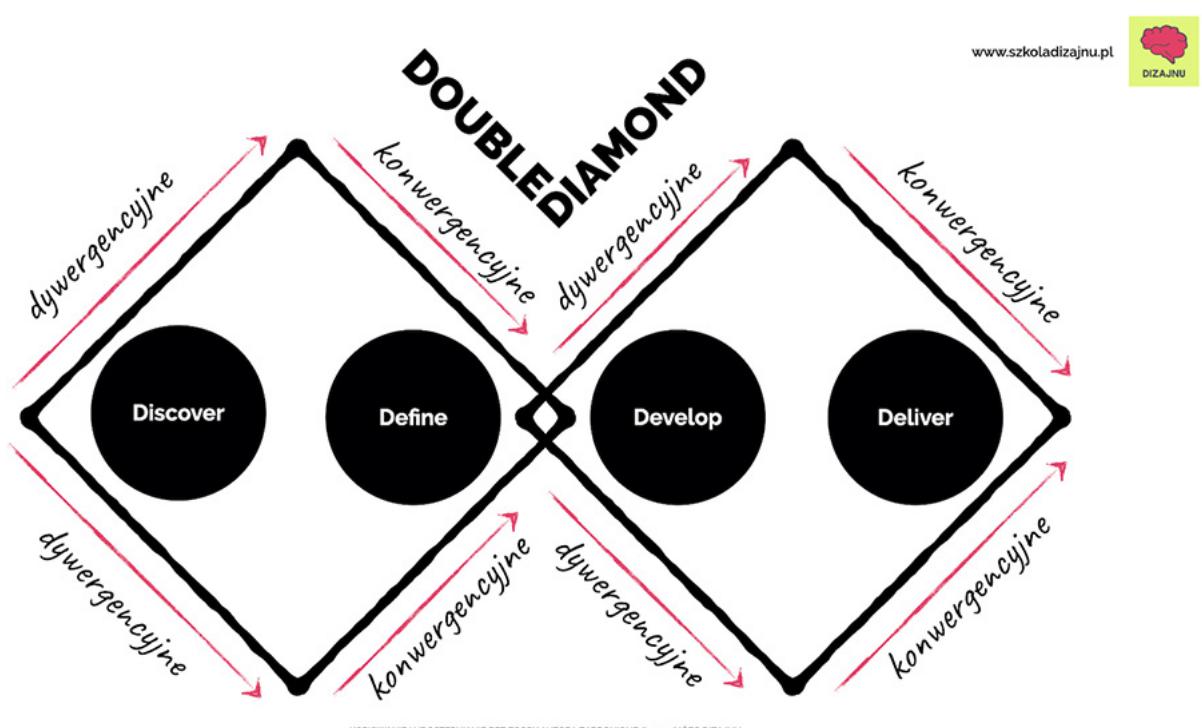
Sprawdzenie, jak użytkownicy reagują na stworzone rozwiązania. Umożliwia iteracyjne dopracowanie produktu na podstawie rzeczywistych opinii użytkowników. Testowanie może obejmować wywiady, obserwacje bądź analizę danych użytkowników.

2.2. Zarządzanie projektem

Istnieje wiele technik zarządzania projektem [3]. Oto kilka z nich:

- 1. "PRINCE2" - metoda zarządzania projektami skupiająca się na produktach. Podczas jej realizacji wykorzystuje się doświadczenie specjalistów, którym następnie przypisuje się konkretne obowiązki. Cały proces koncentruje się na tworzonym produkcie, co sprawia, że jest to metoda przejrzysta, stosunkowo prosta i ułatwiająca pracę zespołową. Wśród jej zalet wymienia się dokładność i konieczność tworzenia sumiennej, ustandaryzowanej dokumentacji,
- 2. Klasyczna - "Waterfall",
- 3. "SCRUM" - wspiera zespół w ustrukturyzowanej pracy za pomocą zestawu wartości i konkretnych zasad, ostatecznie doprowadzając do kreatywnego tworzenia jakościowych wyrobów,
- 4. "LEAN" - zakłada dostarczanie produktów w jak najszybszy i najprostszy sposób poprzez położenie nacisku na działania, które dodają wartość do procesu, eliminując marnotrawstwo,
- 5. **Design Thinking - Double Diamond**,

W celu realizacji projektu została wybrana dokładnie metoda "Double Diamond" [4].



Rys. 1. Design Thinking - Double Diamond, grafika przedstawiająca schemat działania metody [4]

Proces działania tej metody wygląda następująco:

- **Discover** - pierwszy etap - odkrywanie potrzeb klienta bądź użytkowników, założenie projektowych, zbieranie informacji z wielu źródeł,
- **Define** - drugi etap - analiza zebranych danych, definiowanie proponowanego sposobu rozwiązania problemu, precyzowanie rozwiązania,
- **Develop** - trzeci etap - generowanie pomysłów na rozwiązanie problemu,
- **Deliver** - czwarty etap - wybór jednego z pomysłów i jego realizacja, dostarczanie prototypu i testy. Ostatecznie - dostarczenie końcowego rozwiązania.

2.2.1. Narzędzia

- * Lista narzędzi niezbędnych w projekcie może ulec zmianie wraz realizacją kolejnych etapów
- **GitHub** - zamieszczanie, wersjonowanie, aktualizacja kodu źródłowego programu realizowanego w ramach projektu,
- **TeXstudio** - generowanie raportu z poszczególnych etapów projektowych,
- **Google Scholar** - prace naukowe, artykuły pomocne w realizacji zadania
- **Środowisko programistyczne Pycharm** (język Python),
- **Discord** - platforma posłuży do komunikacji oraz wymiany informacji z członkami zespołu,
- **Matlab** - obliczenia numeryczne i symboliczne
- **Quartus Prime Lite** - projektowanie w języku Verilog
- **ModelSim** - testowanie, symulacja gotowego kodu źródłowego przed uruchomieniem na płytce FPGA

3. Informacje podstawowe

Obecnie przetwarzanie i analiza sygnałów radiowych są kluczowe w działaniu wielu technologii:

- **GSM** - w sieciach 5G stosuje się "massive MIMO" (Multiple Input Multiple Output), wymagające zaawansowanego przetwarzania sygnałów w czasie rzeczywistym,
- **Radiolokacja** - radary lotnicze, wojskowe, meteorologiczne
- **Radioastronomia i badania kosmosu** - radioteleskopy i sondy kosmiczne

We wszystkich wspomnianych technologiach wymagana jest również korekcja błędów, która nie może być zrealizowana bez analizy sygnału. Szczególnie uwidacznia się to w technologii GSM. Jeśli urządzenie oddala się od stacji nadawczej, jakość przetwarzanego sygnału spada, zatem urządzenie zwiększa moc nadawania lub łączy się z inną anteną nadawczą. Taka operacja wymaga analizy i przetworzenia sygnału oraz danych odebranych.

Celem tego projektu również jest przetworzenie pewnego sygnału.

3.1. Istniejące rozwiązania

Istnieje wiele różnych metod i programów służących do przetwarzania sygnałów:

- **SDR** (Software-Defined Radio) [5] - programy i urządzenia tego typu pozwalają na przetwarzanie odebranych sygnałów, w szerokim zakresie częstotliwości, w czasie rzeczywistym z wykorzystaniem wielu różnych technik modulacji.
Przykładowy program pozwalający na obsługę urządzeń typu SDR:
 - **HDSDR** [6] - Program umożliwia zarówno analizę sygnału pochodzącego z odbiornika w czasie rzeczywistym, ale również nagranych uprzednio w formacie .wav,
- **GNU Radio** [7] - Program ma funkcjonalność podobną do HDSDR, jednak proces przetwarzania można zdefiniować samodzielnie przy pomocy bloków. Program jest szeroko stosowany przez hobbytów, środowiska akademickie i komercyjne, wspierając zarówno badania nad komunikacją bezprzewodową, jak i rzeczywiste systemy radiowe.
- **Audacity** [8] - Program pozwalający na zaawansowaną analizę i modyfikację ścieżek dźwiękowych. Umożliwia analizę przy pomocy spektrogramów oraz nakładanie filtrów.
- **Paczka narzędzi "Signal Processing Toolbox" do programu MATLAB** [9] - Dostarcza funkcje oraz aplikacje do zarządzania, analizy, wstępnego przetwarzania i ekstrakcji cech z sygnałów próbkowanych równomiernie i nierównomiernie. Toolbox obejmuje narzędzia do projektowania i analizy filtrów, zmiany częstotliwości próbkowania, wygładzania, usuwania trendów oraz estymacji widma mocy. Można skorzystać z aplikacji Analizator Sygnałów do wizualizacji i przetwarzania sygnałów jednocześnie w dziedzinie czasu, częstotliwości oraz czasu-częstotliwości. Toolbox pozwala również na projektowanie własnych filtrów.

- **Baza przykładów Transformat Fouriera (GitHub)** [10] - zawiera przykładowe implementacje transformat Fouriera (w szczególności FFT) napisanych w języku Python
- **Biblioteki dla języka Python wspomagające przetwarzanie sygnałów:**
 - **SciPy** [11] - SciPy to otwarte oprogramowanie do zastosowań matematycznych, naukowych i inżynierskich. Zawiera moduły do statystyki, optymalizacji, całkowania numerycznego, algebry liniowej, transformat Fouriera, przetwarzania sygnałów i obrazów, rozwiązywania równań różniczkowych zwyczajnych (ODE), oraz wiele innych funkcjonalności.
 - **NumPy** [12] - biblioteka Pythona służąca do pracy z tablicami. Oferuje również funkcje do obliczeń w zakresie: algebry liniowej, transformat Fouriera, operacji na macierzach.
 - **Matplotlib** [13] - popularna biblioteka do wizualizacji danych w Pythonie. Często wykorzystywana do tworzenia: statycznych wykresów, interaktywnych wizualizacji, animowanych prezentacji danych. Dzięki Matplotlib można w kilku linijkach kodu generować: wykresy liniowe i słupkowe, histogramy, diagramy punktowe (scatter plots). Jest przydatna do wizualizacji wyniku działania programu, który wyświetla końcowe wyniki działań programów (np. po skończonej analizie sygnału w postaci wykresu mocy i zakresu częstotliwości).

3.2. Podstawy przetwarzania sygnałów

Podstawą wszelkiego przetwarzania, analizy sygnałów jest transformata Fouriera [14] [15] oraz różne jej modyfikacje.

3.2.1. Jak działa transformata Fouriera?

Transformata Fouriera [14] rozkłada przebieg na sumę sinusoid o różnych częstotliwościach. Oznacza to, że transformata Fouriera reprezentuje te same informacje, co oryginalny kształt fali, tylko w dziedzinie częstotliwości (w przeciwnieństwie do dziedziny czasu).

Poniższe równanie przedstawia ciągłą transformatę Fouriera i pokazuje, w jaki sposób dowolny sygnał czasu ciągłego może być reprezentowany jako suma (całka) sinusoid o wszystkich możliwych częstotliwościach:

$$S(f) = \int_{-\infty}^{\infty} s(t) \cdot e^{-j2\pi ft} dt$$

Gdzie:

- $S(f)$ reprezentuje transformatę Fouriera sygnału $s(t)$,
- e jest podstawą logarytmu naturalnego ($e \approx 2,71828$),
- j jest jednostką urojoną (spełniającą $j^2 = -1$).

Jednak ta transformata w czystej postaci nie była wystarczająco wydajna, dlatego powstały algorytmy usprawniające jej działanie:

3.2.2. DFT i FFT

DFT (Dyskretna Transformata Fouriera) [14] to narzędzie matematyczne służące do przekształcania dyskretnych, okresowych sygnałów z ich reprezentacji w dziedzinie czasu na reprezentację w dziedzinie częstotliwości. Pozwala to na analizę i manipulację w dziedzinie częstotliwości przed potencjalnym przekształceniem z powrotem do dziedziny czasu za pomocą odwrotnej DFT.

Dyskretna Transformata Fouriera wyraża się wzorem:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j(\frac{2\pi}{N})kn}$$

Gdzie:

- $X[k]$ jest k -tym elementem reprezentacji w dziedzinie częstotliwości,
- $x[n]$ jest n -tym elementem sygnału w dziedzinie czasu,
- e jest podstawą logarytmu naturalnego (w przybliżeniu równą 2,71828),
- j jest jednostką urojoną (spełniającą $j^2 = -1$).

Zauważono, że istnieje dalsza możliwość znacznego usprawnienia algorytmu liczenia DFT. Wynikiem tego spostrzeżenia była **FFT (Szybka Transformata Fouriera)** [14]. Została opracowana przez **Jamesa W. Cooleya i Johna W. Tukeya** w 1965 roku. Ich wersja algorytmu znacznie zmniejszyła złożoność obliczeniową przetwarzania dużych zbiorów danych.

Dla przykładu [16] obliczenie DFT dla 1000 próbek wymaga 1 048 576 mnożeń i 1 047 552 dodawań. Dla porównania, w przypadku FFT, jeśli liczba próbek jest wielokrotnością liczby 2 (a więc 64, 128, 256, 512 czy 1024), to obliczenia można znacznie uprościć. W algorytmie FFT obliczanie wyniku dla 1024 próbek wymaga tylko 5 120 mnożeń i 10 240 dodawań.

FFT jest algorytmem używanym do obliczania dyskretnej transformaty Fouriera (DFT) i jej odwrotności. DFT jest przekształceniem stosowanym w przetwarzaniu sygnałów, w celu przekształcenia dyskretnego sygnału w jego reprezentację w dziedzinie częstotliwości. FFT przyspiesza proces obliczania DFT, umożliwiając jego wykorzystanie w aplikacjach czasu rzeczywistego i dla dużych zbiorów danych.

3.2.3. Różnice między transformatą Fouriera, a FFT

Różnica między szybką transformatą Fouriera (FFT) a transformatą Fouriera (FT) [14] leży w sposobie obliczeń i efektywności. FFT to specjalna, wydajniejsza metoda obliczania dyskretnej transformaty Fouriera (DFT), która jest z kolei dyskretnym odpowiednikiem ciągłej transformaty Fouriera (FT).

FT przekształca sygnał ciągły w dziedzinie czasu na sygnał ciągły w dziedzinie częstotliwości. W praktyce jednak często mamy do czynienia z sygnałami próbkowanymi (dyskretnymi), dlatego używa się DFT, która przekształca ciąg próbek sygnału w ciąg równie licznych próbek jego widma częstotliwościowego.

FFT to zoptymalizowana wersja DFT, która redukuje złożoność obliczeniową. Dzięki temu FFT jest znacznie szybsza w obliczeniach niż bezpośrednie zastosowanie DFT, szczególnie dla dużych zestawów danych. Optymalizacje te opierają się na dekompozycji DFT na mniejsze DFT o rozmiarach będących potęgami liczb 2.

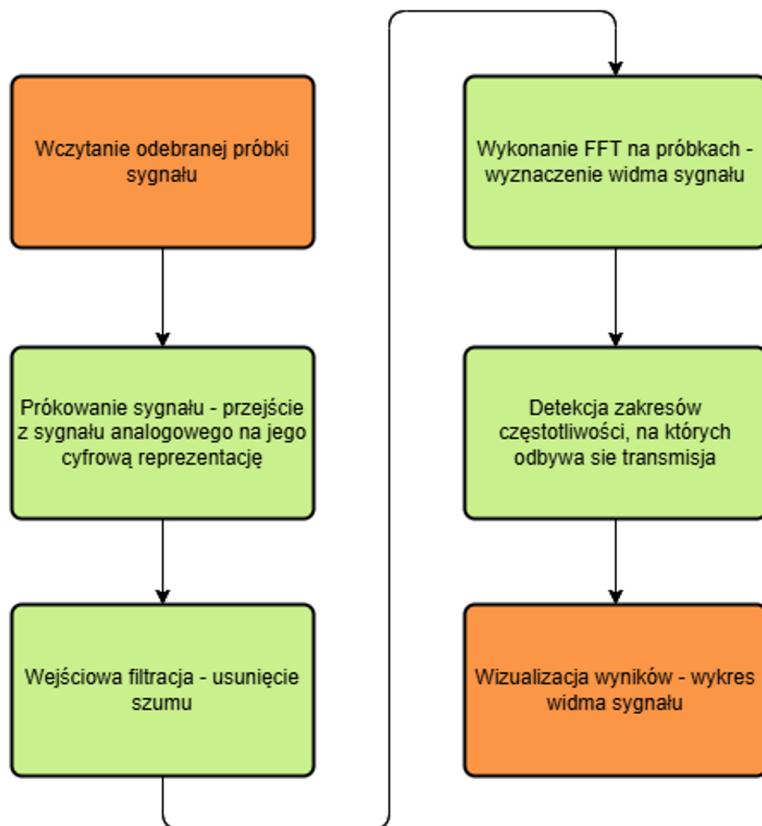
4. Koncepcja

Po przeanalizowaniu założeń oraz wymagań projektowych, gotowych rozwiązań oraz dostępnych bibliotek pozwalających przetwarzać sygnały i wykonywać operacje na nich, wykładała się pewna koncepcja rozwiązania.

4.1. Mapa myśli

- **Cel projektu:** Szybka i dokładna analiza sygnału odebranego z łazika marsjańskiego w celu wykrycia zakresu częstotliwości transmisji
- **Metody analizy:**
 - Szybka Transformata Fouriera (FFT) - wybrana ze względu na wysoką wydajność działania oraz niskie zużycie zasobów
 - Filtracja sygnału - usuwanie ewentualnych szumów, które mogłyby utrudniać, bądź uniemożliwić poprawną analizę sygnału, także usuwanie składowych stałych (normalizacja)
 - Wizualizacja wyników - w postaci wykresu widma sygnału
- **Narzędzia:**
 - Język programowania Python oraz biblioteki pozwalające na analizę sygnału oraz wizualizację wyników (NumPy, SciPy, Matplotlib) - model referencyjny, pierwsza wersja rozwiązania
 - MATLAB - weryfikacja działania programu napisanego w języku Python
 - FPGA (Verilog) - docelowa implementacja sprzętowa projektu
- **Testowanie:**
 - Losowy sygnał wygenerowany przez kod referencyjny
 - Analizator widma online [17]
 - W późniejszym etapie sygnały, które zostały dostarczone wraz z zadaniem projektowym

4.2. Schemat blokowy rozwiązania



Rys. 2. Schemat blokowy rozwiązania

4.3. Opis schematu blokowego

- I **Wczytanie odebranej próbki sygnału** - do zadania zostały załączone próbki sygnałów, które należy przeanalizować. Z czego wynika, że kod musi umożliwiać wprowadzenie danych, które będzie analizował (w tym przypadku plik dźwiękowy w formacie .wav)
- II **Próbkowanie sygnału** - umożliwia przejście z sygnału analogowego na cyfrową jego reprezentację. Pozwoli to na wykonywanie wszystkich operacji na sygnale (w zasadzie jego próbkach) prowadzących do końcowego rozwiązania - wykrycia zakresu częstotliwości transmisji
- III **Wejściowa filtracja** - pozwoli na usunięcie szumu oraz składowych stałych z sygnału co umożliwia przetwarzanie próbek oraz sprawi, że widmo będzie reprezentować sygnał przetworzony w sposób czytelniejszy
- IV **Wykonanie FFT** - (Szybkiej Transformaty Fouriera) - główna operacja na sygnale, przekształca sygnał z dziedziny czasu w dziedzinę częstotliwości. Po jej wykonaniu możliwa będzie reprezentacja sygnału przy pomocy jego widma
- V **Detekcja zakresów częstotliwości** - główne założenie zadania. Po wykonaniu FFT, próbki można bardzo łatwo zaprezentować na wykresie, gdzie widoczne będą maksima amplitudy sygnału oraz częstotliwość, na której występują
- VI **Wizualizacja wyników** - wyniki zostaną przedstawione jako wykres, gdzie na osi poziomej naniesione będą wartości częstotliwości, natomiast na pionowej - amplitudy

4.4. Referencyjny kod w języku Python

Program generuje losowy sygnał testowy, zapisuje go do pliku .wav, następnie analizuje sygnał i wyświetla widmo.

```
1 import numpy as np
2 from scipy.fft import fft
3 from scipy.io import wavfile
4 import matplotlib.pyplot as plt
5
6 # Parametry sygnalu
7 fs = 5000 # częstotliwość próbkowania [Hz]
8 T = 1.0    # czas trwania [s]
9 N = 1024   # liczba punktów FFT
10
11 # Generowanie sygnalu testowego (2 składowe + szum)
12 t = np.linspace(0, T, int(fs*T), endpoint=False)
13 signal = (1.0 * np.sin(2*np.pi*300*t) + \
14 (0.5 * np.sin(2*np.pi*800*t)) + \
15 (0.2 * np.random.randn(len(t))))
16
17 # Normalizacja sygnalu do zakresu [-1, 1]
18 signal_normalized = signal / np.max(np.abs(signal))
19
20 # Zapis do pliku WAV (16-bit)
21 wavfile.write('D:/test_signal.wav', fs, (signal_normalized * 32767).astype(np.int16))
22
23 # Obliczenie FFT
24 yf = fft(signal[:N])
25 xf = np.linspace(0, fs/2, N//2)
26
27 # Wizualizacja
28 plt.figure(figsize=(10,4))
29 plt.plot(xf, 2/N * np.abs(yf[:N//2]))
30 plt.title('Widmo sygnalu testowego')
31 plt.xlabel('Częstotliwość [Hz]')
32 plt.ylabel('Amplituda')
33 plt.grid()
34 plt.show()
35
36 print("Sygnał został zapisany do pliku 'test_signal.wav'")
```

Listing 1: Referencyjny kod w języku Python

4.5. Kod sprawdzający w MATLAB

Dodatkowo, w oparciu o stronę dokumentacji [18] [19], został napisany kod w MATLAB (używający paczki narzędzi Signal Processing Toolbox [9]), który również jest w stanie pobrać plik .wav, spróbować, przefiltrować i policzyć Szybką Transformatę Fouriera. Wyniki również prezentuje w postaci wykresu.

```

1      % Analiza widma sygnalu z pliku WAV
2 clear all;
3 close all;
4 clc;
5      % 1. Wczytanie pliku WAV
6 [filename, pathname] = uigetfile('.wav', 'Wybierz plik WAV do analizy');
7 if isequal(filename, 0)
8 disp('Anulowano wybor pliku');
9 return;
10 end
11 [y, fs] = audioread(fullfile(pathname, filename));
12      % 2. Preprocessing sygnalu
13      % Konwersja na mono jesli stereo
14 if size(y, 2) > 1
15 y = mean(y, 2);
16 disp('Sygnal stereo przekonwertowany na mono');
17 end
18      % Parametry analizy
19 N = length(y);           % Dlugosc sygnalu
20 t = (0:N-1)/fs;         % Os czasu
21 f = (0:N-1)*(fs/N);    % Os cestotliwosci
22 f = f(1:floor(N/2));   % Tylko cestotliwosci dodatnie
23      % 3. Obliczenie transformaty Fouriera
24 Y = fft(y);
25 P2 = abs(Y/N);          % Spektrum dwustronne
26 P1 = P2(1:floor(N/2));  % Spektrum jednostronne
27 P1(2:end-1) = 2*P1(2:end-1); % Skalowanie amplitudy
28      % 4. Wykrywanie glownych skladowych cestotliwosciowych
29 [peaks, locs] = findpeaks(P1, 'MinPeakHeight', 0.1*max(P1));
30 freq_peaks = f(locs);
31      % 5. Wizualizacja wynikow
32 figure('Name', 'Analiza widma sygnalu', 'NumberTitle', 'off');
33      % Wykres sygnalu w dziedzinie czasu
34 subplot(2,1,1);
35 plot(t, y);
36 title('Sygnal w dziedzinie czasu');
37 xlabel('Czas [s]');
38 ylabel('Amplituda');
39 xlim([0 t(end)]);
40 grid on;
41      % Wykres widma cestotliwosciowego
42 subplot(2,1,2);
43 plot(f, P1);
44 title('Widmo cestotliwosciowe (FFT)');
45 xlabel('Cestotliwosc [Hz]');
46 ylabel('|Amplituda|');
47 hold on;
48 plot(f(locs), peaks, 'ro'); % Zaznaczenie pikow
49 for i = 1:length(peaks)
50 text(f(locs(i)), peaks(i), sprintf(' %.1f Hz', freq_peaks(i)));
51 end
52 hold off;
53 grid on;
54 xlim([0 fs/2]);

```

Listing 2: Kod sprawdzający MATLAB - 1

```

1 % Wyswietlenie informacji o sygnale
2 fprintf('\nAnaliza pliku: %s\n', filename);
3 fprintf('Czestotliwosc probkowania: %d Hz\n', fs);
4 fprintf('Czas trwania: %.2f s\n', N/fs);
5 fprintf('Wykryte glowne skladowe czestotliwosciowe:\n');
6 disp(freq_peaks);

```

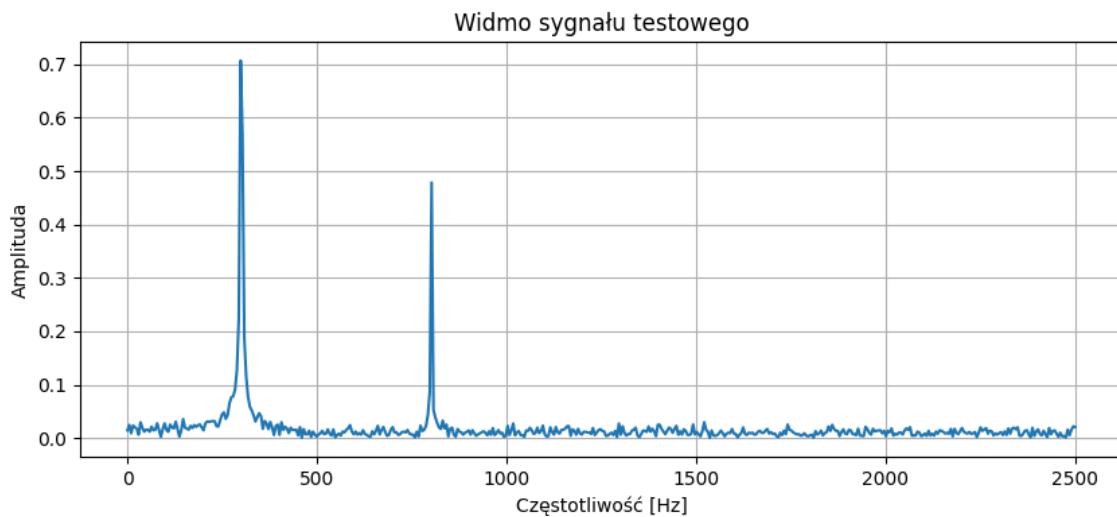
Listing 3: Kod sprawdzający MATLAB - 2

4.6. Testowanie kodu referencyjnego

4.6.1. Wynik działania kodu referencyjnego napisanego w języku Python

Kod sam wygeneruje losowy sygnał o 2 składowych, który następnie zapisze do pliku .wav. Pozwoli to na użycie tego samego sygnału w innym narzędziu analizującym widmo (w tym przypadku [17]) w celu zweryfikowania poprawności działania kodu.

Wynik działania kodu referencyjnego:



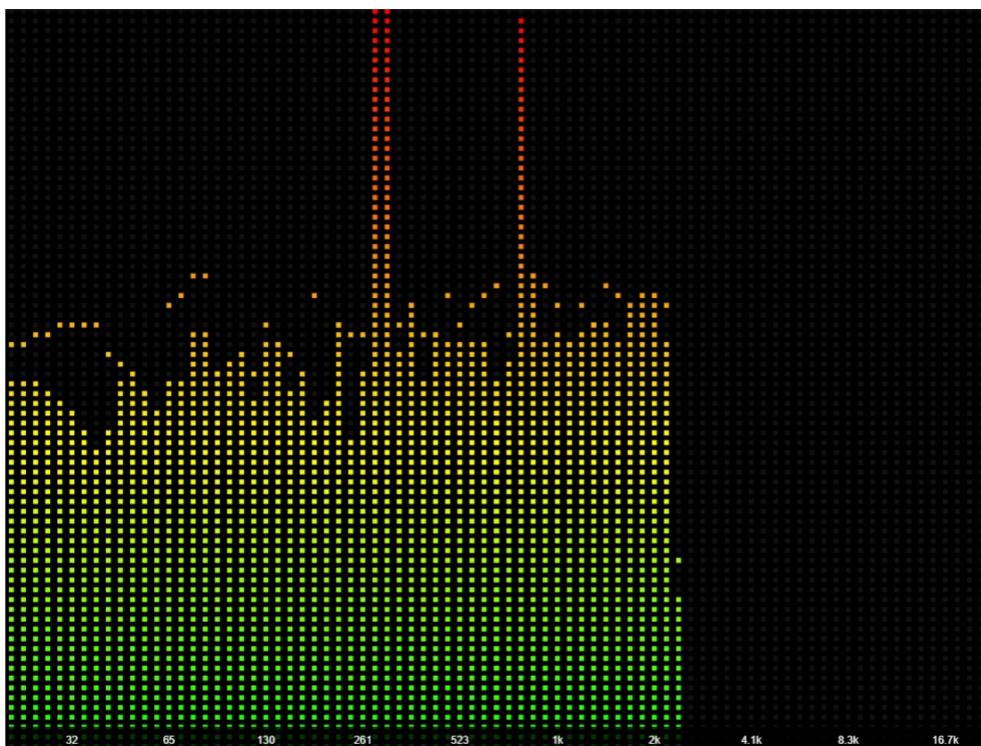
Rys. 3. Wynik działania kodu referencyjnego

Wykres prezentuje widmo wygenerowanego sygnału, który został sprobkowany, znormalizowany oraz policzona została Szybka Transformata Fouriera, a wynik zaprezentowany w postaci wykresu.

4.6.2. Wynik analizy sygnału w analizatorze widma online

W celu weryfikacji poprawności działania kodu skorzystano z analizatora widma dostępnego online [17]. Sygnał zapisany przez kod referencyjny został zimportowany do programu i przeanalizowany.

Wynik jest następujący:

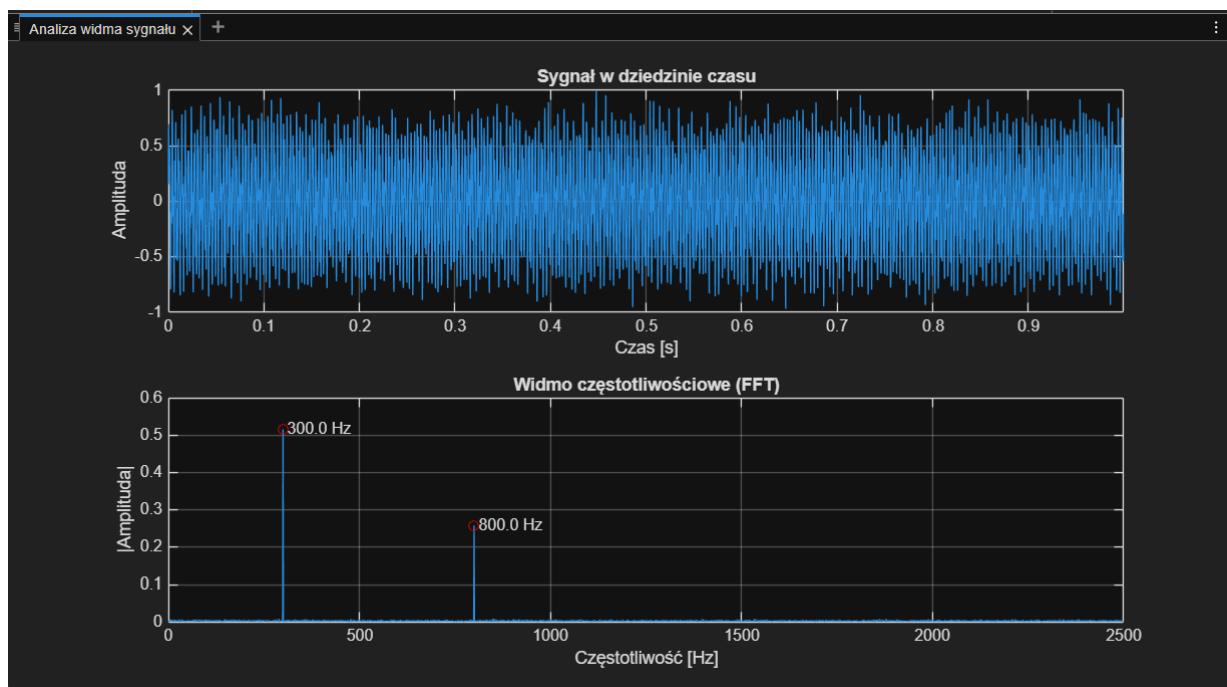


Rys. 4. Wynik działania analizatora widma online

4.6.3. Wynik działania kodu sprawdzającego MATLAB

Kolejną metodą weryfikacji będzie kod napisany w MATLAB, który również jest w stanie przeanalizować sygnał i przedstawić jego widmo w postaci wykresu.

Wynik jego działania jest następujący:



Rys. 5. Wynik działania kodu MATLAB

4.6.4. Podsumowanie testów

Zarówno na wykresie wygenerowanym przez kod referencyjny, program online jak i kod w MATLAB można zaobserwować, że maksimum amplitudy sygnału występuje w okolicach 300 Hz oraz 800 Hz. Dowodzi to poprawności napisanego kodu (tego w języku Python oraz tego w MATLAB) oraz jego zdolności do poprawnego przetworzenia sygnału wejściowego - FFT została policzona poprawnie, wykres widma również został utworzony w sposób poprawny. Rozwiązania te zostaną wykorzystane na etapie finalnej implementacji rozwiązania.

5. Implementacja

5.1. Finalna realizacja systemu w języku Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import wave
4 import struct
5 import sys
6 import os
7
8
9 def read_wav_file(filename):
10
11     """Odczytuje plik .wav i zwraca sygnał oraz częstotliwość próbkowania."""
12     with wave.open(filename, 'rb') as wav_file:
13         n_channels = wav_file.getnchannels()
14         sample_width = wav_file.getsampwidth()
15         framerate = wav_file.getframerate()
16         n_frames = wav_file.getnframes()
17
18         frames = wav_file.readframes(n_frames)
19         if sample_width == 1:
20             fmt = f"{n_frames * n_channels}B" # unsigned char
21         elif sample_width == 2:
22             fmt = f"{n_frames * n_channels}h" # short
23         else:
24             raise ValueError("Nieobsługiwana szerokość próbki")
25
26         signal = np.array(struct.unpack(fmt, frames))
27
28         if n_channels == 2:
29             signal = signal.reshape(-1, 2)
30             signal = signal.mean(axis=1) # Konwersja stereo na mono
31
32     return signal, framerate
```

Listing 4: Finalna realizacja systemu w języku Python - 1

```

1 def fft(x):
2
3     """Ręcznie zaimplementowana szybka transformata Fouriera (FFT)."""
4     N = len(x)
5     if N <= 1:
6         return x
7     even = fft(x[0::2])
8     odd = fft(x[1::2])
9     T = [np.exp(-2j * np.pi * k / N) * odd[k] for k in range(N // 2)]
10    return [even[k] + T[k] for k in range(N // 2)] + [even[k] - T[k] for k in range(N // 2)]
11
12 def analyze_frequencies(signal, sample_rate):
13
14     """Analizuje sygnał i znajduje znaczące piki częstotliwościowe."""
15     n = len(signal)
16
17     # Usuwanie składowej stałej (DC offset)
18     signal = signal - np.mean(signal)
19
20     # Zastosowanie okna Hanninga
21     window = np.hanning(n)
22     signal_windowed = signal * window
23
24     # Obliczenie FFT
25     fft_result = np.abs(fft(signal_windowed))
26
27     # Obliczenie częstotliwości (pomijamy częstotliwość ujemną)
28     freqs = np.fft.fftfreq(n, d=1 / sample_rate)[:n // 2]
29     fft_magnitude = fft_result[:n // 2]
30
31     # Normalizacja (do wartości maksymalnej)
32     if np.max(fft_magnitude) > 0:
33         fft_magnitude = fft_magnitude / np.max(fft_magnitude)
34
35     # Znajdź piki (powyżej pewnego progu)
36     threshold = 0.1 # Próg dla znaczących pików
37     peaks = np.where(fft_magnitude > threshold)[0]
38
39     # Pogrupuj bliskie piki
40     peak_groups = []
41     if len(peaks) > 0:
42         current_group = [peaks[0]]
43         for peak in peaks[1:]:
44             if peak - current_group[-1] < 2: # Grupuj bliskie częstotliwości
45                 current_group.append(peak)
46             else:
47                 peak_groups.append(current_group)
48                 current_group = [peak]
49                 peak_groups.append(current_group)

```

Listing 5: Finalna realizacja systemu w języku Python - 2

```

1 # Wybierz najwyższy pik z każdej grupy
2 significant_peaks = []
3 for group in peak_groups:
4     max_peak_idx = group[np.argmax(fft_magnitude[group])]
5     significant_peaks.append(max_peak_idx)
6
7 peak_freqs = freqs[significant_peaks]
8 peak_mags = fft_magnitude[significant_peaks]
9
10 # Filtruj piki powyżej 20 Hz (aby pominać resztki DC i niskie zakłócenia)
11 mask = peak_freqs > 20
12 peak_freqs = peak_freqs[mask]
13 peak_mags = peak_mags[mask]
14
15 return freqs, fft_magnitude, peak_freqs, peak_mags
16
17 def plot_spectrum(freqs, spectrum, peak_freqs, peak_mags, filename):
18
19     """Rysuje widmo sygnału z oznaczonymi pikami."""
20     plt.figure(figsize=(12, 6))
21     plt.plot(freqs, spectrum, label="Widmo amplitudowe")
22
23     # Zaznacz tylko istotne piki
24     if len(peak_freqs) > 0:
25         plt.scatter(peak_freqs, peak_mags, color='red', label="Znaczące piki")
26         for freq, mag in zip(peak_freqs, peak_mags):
27             plt.text(freq, mag + 0.05, f"{freq:.2f} Hz", ha='center', fontsize=9)
28
29     plt.title(f"Analiza częstotliwościowa sygnału: {os.path.basename(filename)}")
30     plt.xlabel("Częstotliwość (Hz)")
31     plt.ylabel("Amplituda (znormalizowana)")
32     plt.grid()
33     plt.legend()
34     plt.xlim(0, max(freqs)) # Pomijamy częstotliwości ujemne
35     plt.show()
36
37
38 def main():
39     # Sprawdź argumenty linii poleceń
40     if len(sys.argv) != 2:
41         print("Użycie: python analizator_fft.py <plik.wav>")
42         print("Przykład: python analizator_fft.py test.wav")
43         sys.exit(1)
44
45     filename = sys.argv[1]
46
47     # Sprawdź czy plik istnieje
48     if not os.path.isfile(filename):
49         print(f"Błąd: Plik '{filename}' nie istnieje!")
50         sys.exit(1)

```

Listing 6: Finalna realizacja systemu w języku Python - 3

```

1 # Sprawdź rozszerzenie pliku
2 if not filename.lower().endswith('.wav'):
3     print("Uwaga: Plik nie ma rozszerzenia .wav, ale spróbuję go wczytać...")
4
5 # Wczytaj sygnał z pliku .wav
6 try:
7     signal, sample_rate = read_wav_file(filename)
8 except Exception as e:
9     print(f"Błąd podczas wczytywania pliku: {e}")
10    sys.exit(1)
11
12 # Ogranicz do pierwszych N próbek (dla wydajności)
13 max_samples = 4096 # Można zwiększyć dla lepszej rozdzielczości
14 if len(signal) > max_samples:
15     signal = signal[:max_samples]
16
17 # Analiza częstotliwości
18 freqs, spectrum, peak_freqs, peak_mags = analyze_frequencies(signal, sample_rate)
19
20 # Wizualizacja
21 plot_spectrum(freqs, spectrum, peak_freqs, peak_mags, filename)
22
23 # Wyświetl znalezione piki
24 print("\nZnalezione znaczące piki częstotliwościowe (>20 Hz):")
25 if len(peak_freqs) > 0:
26     for freq, mag in zip(peak_freqs, peak_mags):
27         print(f"- {freq:.2f} Hz (względna amplituda: {mag:.2f})")
28 else:
29     print("Nie znaleziono istotnych pików powyżej progu.")
30
31 if __name__ == "__main__":
32     main()

```

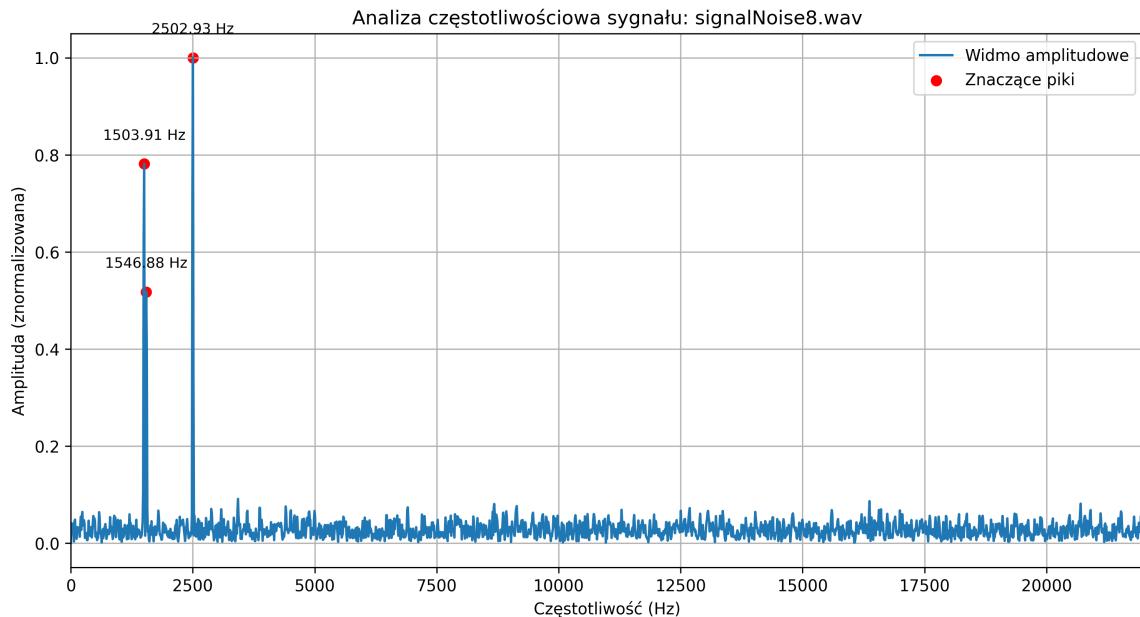
Listing 7: Finalna realizacja systemu w języku Python - 4

Powyższy kod został napisany w oparciu o program z rozdziału 4.4, teraz jednak, zamiast generować i analizować wygenerowany sygnał, należy podać nazwę sygnału do analizy jako argument przy uruchamianiu kodu z poziomu wiersza poleceń. Kod wykona czynności niezbędne do przetworzenia sygnału (odczyt z pliku, usunięcie składowej stałej), obliczy szybką Transformatę Fouriera, a następnie poda wynik w postaci widma sygnału z oznaczonymi istotnymi "pikami" na osi częstotliwości wraz z ich dokładną lokalizacją.

5.2. Testowanie finalnego kodu w języku Python

Testy zostały przeprowadzone przy użyciu sygnału o nazwie *SignalNoise8.wav*, który został dostarczony wraz z zadaniem projektowym. Sygnał ten został zaimportowany do wszystkich metod analizy opisanych w rozdziale 4.6.

5.2.1. Wynik działania finalnego kodu w języku Python



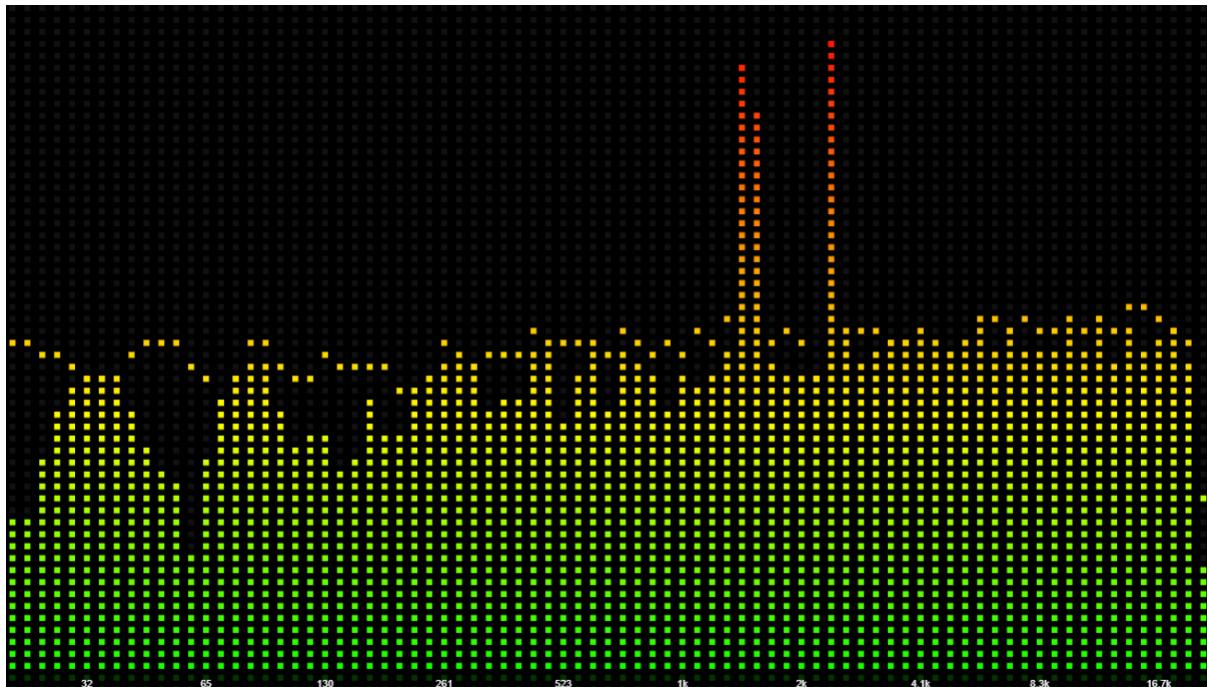
Rys. 6. Wynik działania finalnego kodu w języku Python

5.2.2. Wynik działania kodu testującego Matlab



Rys. 7. Wynik działania kodu testującego MATLAB

5.2.3. Wynik działania analizatora widma online



Rys. 8. Wynik działania analizatora widma online dla signalNoise8.wav

Wszystkie metody testowe zwróciły takie same, poprawne odpowiedzi. Można więc wnioskować, że kod działa poprawnie i kolejnym etapem jest przejście do implementacji zadania w Verilog.

5.3. Implementacja zadania w języku Verilog

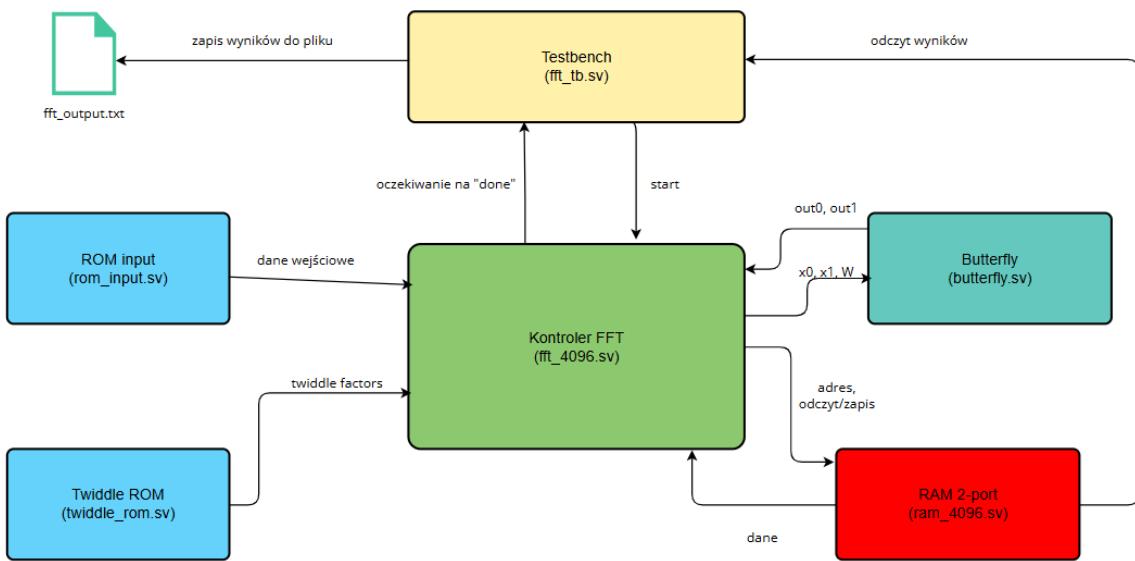
W celu zaimplementowania zadania projektowego na platformie FPGA został wykorzystany język Verilog oraz programy: Intel Quartus Prime Lite (kompilacja) oraz Modelsim (symulacja i otrzymanie pliku .txt z wynikami FFT)

5.3.1. Założenia projektowe - FPGA

- Implementacja bez wykorzystania IP Core
- Źródłem danych jest plik signalSample.txt z gotowymi próbками (dostarczony wraz z zadaniem projektowym)
- Wynik działania projektu, po symulacji, ma być zapisany do pliku, a następnie przy pomocy kodu w języku Python ma zostać wygenerowane widmo sygnału, z którego pochodzą wejściowe próbki (signalSample.txt)

5.3.2. Schemat postępowania projektowego

Całość zadania (pod względem funkcjonalnym) została rozłożona na czynniki pierwsze, następnie wykłarowano niezbędne bloki projektu oraz relacje między nimi.



Rys. 9. Schemat przepływu danych wewnątrz projektu

Opis poszczególnych bloków:

- **rom_input.sv** - przechowuje próbki danych (pochodzące z pliku signalSample.txt, zapisane na stałe) w formacie stałoprzecinkowym - 16 bitów dla części rzeczywistej oraz 16 bitów dla części urojonej. Z tej pamięci dane ładowane są do RAM-u w kolejności z odwróceniem bitów.
- **fft_4096.sv** - główny moduł kontrolera FFT. Odpowiada za:
 - sekwencyjne ładowanie danych do RAM,
 - iterację po etapach FFT (etapy motyli)
 - zarządzanie adresami i współczynnikami *twiddle*,
 - zapis wyników obliczeń z powrotem do RAM
- **butterfly.sv** - wykonuje obliczenia dla dwóch punktów FFT z użyciem odpowiedniego współczynnika twiddle. Otrzymuje dane x_0 , x_1 , W i zwraca $x_0 + t$ oraz $x_0 - t$ po pomnożeniu zespolonym,
- **ram_4096.sv** - dwuportowa pamięć RAM, w której przechowywane są dane wejściowe i pośrednie wyniki FFT. Każdy port posiada:
 - adres, sygnał zapisu, wejścia i wyjścia *real* i *imag*,
 - dostęp jednoczesny przez port A i B
- **twiddle_rom.sv** - ROM zawierający współczynniki $e^{-j2\pi k/N}$ dla wszystkich etapów FFT (preobliczone, na stałe zapisane wartości sinusa i cosinusa w formacie Q15)
- **fft_tb.sv** - testbench odpowiedzialny za:
 - wygenerowanie zegara i resetu,
 - aktywowanie sygnału *start*
 - oczekiwanie na *done*
 - odczyt danych z RAM po obliczeniu FFT i zapis ich do pliku

5.3.3. Omówienie realizacji poszczególnych modułów

W tym rozdziale zostaną dokładniej przybliżone metody realizacji poszczególnych bloków funkcjonalnych projektu. Zostaną również przedstawione najważniejsze fragmenty kodu.

- **rom_input.sv** - pamięć ROM zawierająca 4096 próbek sygnału wejściowego. W pliku dostarczonym z zadaniem projektowym w pliku znajduje się 4000 próbek - został on uzupełniony zerami do wartości, która jest konkretną potęgą liczby 2. Dane zawarte w module są 32-bitowe (16 bitów część rzeczywista i 16 bitów dla części urojonej, która zawsze jest równa 0). ROM adresowany jest 12-bitowo (4096 elementów). W celu utworzenia takiej postaci danych w ROM wykorzystano kod w języku Python

```

1 # Otwarcie pliku i odczyt próbek
2 with open('signalSample.txt', 'r') as file:
3     samples = [float(line.strip()) for line in file.readlines()]
4
5 # Sprawdzenie liczby próbek
6 if len(samples) != 4096:
7     print(f"Uwaga: Liczba próbek ({len(samples)}) nie wynosi 4096!")
8
9 # Generowanie danych wyjściowych
10 with open('output_samples.txt', 'w') as f:
11     for i, sample in enumerate(samples):
12         scaled_value = int(sample * 32768)
13         # Formatowanie ujemnych wartości (używając wartości bezwzględnej i znaku minus)
14         sign = '-' if scaled_value < 0 else ''
15         abs_value = abs(scaled_value)
16         line = f"12'd{i:<4} : begin data_real = {sign}16'${abs_value}; data_imag = -16'${abs_value}; end\n"
17         f.write(line)
18
19 print("Przetwarzanie zakończone. Wynik zapisano w pliku 'output_samples.txt'")

```

Listing 8: Kod przetwarzający próbki na format Q1.15

```

1 module rom_input (
2     input [11:0] addr,
3     output reg signed [15:0] data_real,
4     output reg signed [15:0] data_imag
5 );
6     always @(*) begin
7         case (addr)
8             12'd0 : begin data_real = 16'sd3584; data_imag = 16'sd0; end
9             12'd1 : begin data_real = 16'sd5632; data_imag = 16'sd0; end
10            12'd2 : begin data_real = 16'sd5120; data_imag = 16'sd0; end
11            12'd3 : begin data_real = 16'sd10752; data_imag = 16'sd0; end
12            12'd4 : begin data_real = 16'sd9472; data_imag = 16'sd0; end
13            12'd5 : begin data_real = 16'sd12032; data_imag = 16'sd0; end
14            12'd6 : begin data_real = 16'sd12288; data_imag = 16'sd0; end
15            12'd7 : begin data_real = 16'sd11264; data_imag = 16'sd0; end
16            12'd8 : begin data_real = 16'sd7424; data_imag = 16'sd0; end
17             \\\pozostałe próbki
18             \\\...
19            12'd4092 : begin data_real = 16'sd0; data_imag = 16'sd0; end
20            12'd4093 : begin data_real = 16'sd0; data_imag = 16'sd0; end
21            12'd4094 : begin data_real = 16'sd0; data_imag = 16'sd0; end
22            12'd4095 : begin data_real = 16'sd0; data_imag = 16'sd0; end
23            default: begin data_real = 16'sd0; data_imag = 16'sd0; end
24         endcase
25     end
26 endmodule

```

Listing 9: Fragment kodu modułu rom_input.sv

- **fft_4096.sv** - główny kontroler FFT realizujący pełne przetwarzanie 4096-punktowe w użyciu RAM, ROM i butterfly. Zaimplementowany jest FSM, który zarządza etapami *IDLE*, *LOAD*, *FFT_READ*, *FFT_CALC*, *FFT_WRITE*, *DONE*. Obsługuje 12 etapów ($\log_2(4096) = 12$). Dodatkowo moduł ten współpracuje i integruje pozostałe (*rom_input*, *ram_4096*, *twiddle_rom*, *butterfly*). Tutaj również tworzone są instancje

większości modułów.

Główne sygnały:

- *addr_a/b, we_a/b, write_real_a/b, read_real_a/b*: interfejs RAM
- *rom_addr, tw_addr*: adresy ROM
- *stage, group_count, k_count*: sterowanie etapami FFT

```
1 module fft_4096 (
2     input      clk,
3     input      reset,
4     input      start,
5     output reg   done
6 );
7     // Parametry FFT
8     localparam N = 4096;
9     localparam LOG2N = 12;
10
11    // Sygnały pamięci RAM (port A i B)
12    reg we_a, we_b;
13    reg [LOG2N-1:0] addr_a, addr_b;
14        reg [LOG2N-1:0] addr_i1, addr_i2;
15    reg signed [15:0] din_real_a, din_imag_a;
16    wire signed [15:0] dout_real_a, dout_imag_a;
17    reg signed [15:0] din_real_b, din_imag_b;
18    wire signed [15:0] dout_real_b, dout_imag_b;
19
20    // Sygnały ROM-ów
21    reg [LOG2N-1:0] rom_addr;
22    wire signed [15:0] in_real, in_imag; // wyjście ROM z danymi wejściowymi
23    wire signed [15:0] tw_real, tw_imag; // wyjście ROM z współczynnikami twiddle
24    reg [LOG2N-1:0] tw_addr;
```

Listing 10: Definicje parametrów i sygnałów

```
1 // Instancje pamięci: ROM (dane wejściowe), ROM (twiddle) oraz dwuportowy RAM
2 rom_input rom_inst(.addr(rom_addr), .data_real(in_real), .data_imag(in_imag));
3 twiddle_rom tw_inst(.addr(tw_addr), .twiddle_real(tw_real), .twiddle_imag(tw_imag));
4 ram_4096 ram_inst (
5     .clk(clk),
6     .we_a(we_a),
7     .addr_a(addr_a),
8     .din_real_a(din_real_a),
9     .din_imag_a(din_imag_a),
10    .dout_real_a(dout_real_a),
11    .dout_imag_a(dout_imag_a),
12    .we_b(we_b),
13    .addr_b(addr_b),
14    .din_real_b(din_real_b),
15    .din_imag_b(din_imag_b),
16    .dout_real_b(dout_real_b),
17    .dout_imag_b(dout_imag_b)
18 );
```

Listing 11: Przykład tworzenia instancji

```

1 // Sterowanie FSM
2 always @(posedge clk or posedge reset) begin
3   if (reset) begin
4     // Reset asynchroniczny
5     state      <= IDLE;
6     done       <= 1'b0;
7     we_a       <= 1'b0;
8     we_b       <= 1'b0;
9     load_count <= 0;
10    stage      <= 0;
11    group_count <= 0;
12    k_count    <= 0;
13    addr_a     <= 0;
14    addr_b     <= 0;
15    din_real_a <= 0;
16    din_imag_a <= 0;
17    din_real_b <= 0;
18    din_imag_b <= 0;
19    rom_addr   <= 0;
20    tw_addr    <= 0;
21  end else begin

```

Listing 12: Sterowanie FSM

```

1 LOAD: begin
2   // Ładuj dane wejściowe z ROM do RAM (z odwróceniem bitów adresu)
3   addr_a     <= bit_reverse12(load_count);
4   din_real_a <= in_real;
5   din_imag_a <= in_imag;
6   we_a       <= 1'b1;
7   rom_addr   <= load_count + 1;
8   if (load_count < N-1) begin
9     load_count <= load_count + 1;
10    state <= LOAD;
11  end else begin
12    // Ostatni zapis
13    we_a       <= 1'b0;
14    load_count <= load_count + 1;
15    stage      <= 0;
16    group_count <= 0;
17    k_count    <= 0;
18    state      <= FFT_READ;
19  end
20 end

```

Listing 13: Stany FSM - 1 LOAD

```

1 FFT_READ: begin
2     // Przygotuj odczyt dwóch elementów dla motyla
3     we_a    <= 1'b0;
4     we_b    <= 1'b0;
5     // Oblicz adresy elementów wewnątrz grupy
6     addr_i1 = base + k_count;
7     addr_i2 = base + half + k_count;
8     tw_addr <= k_count * step;
9     addr_a <= addr_i1;
10    addr_b <= addr_i2;
11    state   <= FFT_WAIT;
12
13
14 //-----
15 FFT_WAIT: begin
16     // Czekaj cykl na dane z RAM
17     state <= FFT_CALC1;
18

```

Listing 14: Stany FSM - 2 READ, 3 WAIT

```

1 FFT_CALC1: begin
2     // Odczyt danych (przechowaj w rejestrach motyla)
3     bf_xr <= dout_real_a;
4     bf_xi <= dout_imag_a;
5     bf_yr <= dout_real_b;
6     bf_yi <= dout_imag_b;
7     bf_wr <= tw_real;
8     bf_wi <= tw_imag;
9     state <= FFT_CALC2;
10
11
12 //-----
13 FFT_CALC2: begin
14     // Wykonaj motyla (z użyciem modułu butterfly)
15     out0_real <= bf_out0_r >>> 1;
16     out0_imag <= bf_out0_i >>> 1;
17     out1_real <= bf_out1_r >>> 1;
18     out1_imag <= bf_out1_i >>> 1;
19     state <= FFT_WRITE;
20
21

```

Listing 15: Stany FSM - 4 CALC1, 5 CALC2

```

1 FFT_WRITE: begin
2     // Zapisz wyniki motyla z powrotem do RAM
3     we_a      <= 1'b1;
4     we_b      <= 1'b1;
5     addr_a    <= addr_i1;
6     addr_b    <= addr_i2;
7     din_real_a <= out0_real;
8     din_imag_a <= out0_imag;
9     din_real_b <= out1_real;
10    din_imag_b <= out1_imag;
11    if (k_count < (half - 1)) begin
12        // Następny motyl w grupie
13        k_count <= k_count + 1;
14        state   <= FFT_READ;
15    end else begin
16        // Koniec grupy
17        k_count <= 0;
18        if (group_count < ((N >> (stage+1)) - 1)) begin
19            group_count <= group_count + 1;
20            state   <= FFT_READ;
21        end else begin
22            // Koniec etapu
23            group_count <= 0;
24            if (stage < (LOG2N - 1)) begin
25                stage <= stage + 1;
26                state <= FFT_READ;
27            end else begin
28                // Koniec FFT
29                we_a <= 1'b0;
30                we_b <= 1'b0;
31                done <= 1'b1;
32                state <= DONE_STATE;
33            end
34        end
35    end
36
37    //-----
38    DONE_STATE: begin
39        // FFT zakończone
40        done <= 1'b1;
41    end
42
```

Listing 16: Stany FSM - 6 WRITE, 7 DONE

- **butterfly.sv** - moduł realizuje podstawową operację FFT - obliczanie tzw. motyla, czyli transformacji dwóch punktów danych przy pomocy współczynnika *twiddle*.

Wejścia:

- xr, xi - dane wejściowe punktu A (real i imag)
- yr, yi - dane wejściowe punktu B (real i imag)
- wr, wi - współczynnik twiddle

Wyjścia:

- $out0_r, out0_i$ - wynik $A + B^*W$
- $out1_r, out1_i$ - wynik $A - B^*W$

```

1 module butterfly (
2     input signed [15:0] xr, xi,      // x0: część rzeczywista i urojona
3     input signed [15:0] yr, yi,      // x1: część rzeczywista i urojona
4     input signed [15:0] wr, wi,      // współczynnik twiddle: część rzeczywista i urojona
5     output signed [15:0] out0_r, out0_i, // wynik x0 + t
6     output signed [15:0] out1_r, out1_i // wynik x0 - t
7 );
8 // Mnożenia pośrednie (32-bitowe)
9 wire signed [31:0] mult0 = yr * wr;
10 wire signed [31:0] mult1 = yi * wi;
11 wire signed [31:0] mult2 = yr * wi;
12 wire signed [31:0] mult3 = yi * wr;
13 // Obliczenie  $t = y * W$ 
14 wire signed [15:0] t_real = (mult0 - mult1) >>> 15;
15 wire signed [15:0] t_imag = (mult2 + mult3) >>> 15;
16 // Wyniki motyla:  $x0 + t$  oraz  $x0 - t$ 
17 assign out0_r = xr + t_real;
18 assign out0_i = xi + t_imag;
19 assign out1_r = xr - t_real;
20 assign out1_i = xi - t_imag;
21 endmodule

```

Listing 17: Kod pliku butterfly.sv

— **ram_4096.sv** - dwuportowa pamięć RAM z niezależnymi portami A i B. Posiada 4096 komórek dla części rzeczywistej i urojonej, obsługuje zapis/odczyt synchroniczny na zboczu *clk*, inicjalizacja zerami.

```
1 (* ramstyle = "no_rw_check, M10K" *)
2 module ram_4096 (
3     input clk,
4
5     // Port A
6     input          we_a,
7     input [11:0]    addr_a,
8     input signed [15:0] din_real_a, din_imag_a,
9     output reg signed [15:0] dout_real_a, dout_imag_a,
10
11    // Port B
12    input          we_b,
13    input [11:0]    addr_b,
14    input signed [15:0] din_real_b, din_imag_b,
15    output reg signed [15:0] dout_real_b, dout_imag_b
16 );
17     // Wspólna pamięć 32-bitowa (łączymy real i imag w jednym słowie)
18     reg signed [31:0] mem[0:4095];
19     integer i;
20     initial begin
21         for (i = 0; i < 4096; i = i + 1) begin
22             mem[i] = 32'sd0;
23         end
24     end
25
26     always @ (posedge clk) begin
27         // Port A: zapis
28         if (we_a) begin
29             mem[addr_a] <= {din_real_a, din_imag_a};
30         end
31         // Port B: zapis
32         if (we_b) begin
33             mem[addr_b] <= {din_real_b, din_imag_b};
34         end
35         // Port A: odczyt (po zapisie lub bez)
36         dout_real_a <= mem[addr_a][31:16];
37         dout_imag_a <= mem[addr_a][15:0];
38         // Port B: odczyt
39         dout_real_b <= mem[addr_b][31:16];
40         dout_imag_b <= mem[addr_b][15:0];
41     end
42 endmodule
43
```

Listing 18: Kod z pliku ram_4096.sv

- **twiddle_rom.sv** - ROM zawierający współczynniki twiddle $e^{-j2\pi k/N}$. Zapisane w nim wartości zostały obliczone przy pomocy kodu w języku Python i zapisane w postaci Q15. Dane są 32-bitowe (16 bitów real + 16 bitów imag). Dla FFT 4096 potrzebne są 2048 unikalnych współczynników.

```

1 import numpy as np
2
3 def generate_twiddle_factors(N, filename="twiddle_rom.txt"):
4     with open(filename, "w") as f:
5         for k in range(N // 2):
6             angle = -2 * np.pi * k / N
7             re = np.cos(angle)
8             im = np.sin(angle)
9
10            # Q15 format (skalowanie na zakres -32768 .. 32767)
11            re_q15 = int(np.round(re * (2**15 - 1)))
12            im_q15 = int(np.round(im * (2**15 - 1)))
13
14            # Zabezpieczenie przed overflow (zachowaj -32768 jako granicę)
15            re_q15 = np.clip(re_q15, -32768, 32767)
16            im_q15 = np.clip(im_q15, -32768, 32767)
17
18            # Format do Veriloga: np. -16'sd12345, 16'sd6789
19            line = f"{'-' if re_q15 < 0 else ''}16'sd{abs(re_q15)},"
20            {'-' if im_q15 < 0 else ''}16'sd{abs(im_q15)}\n"
21            f.write(line)
22
23    print(f"Współczynniki zapisane do {filename}")
24
25 generate_twiddle_factors(4096)
26

```

Listing 19: Kod w języku Python, który pozwolił na obliczenie współczynników twiddle

```

1 module twiddle_rom (
2     input [11:0] addr,
3     output reg signed [15:0] twiddle_real,
4     output reg signed [15:0] twiddle_imag
5 );
6     always @(*) begin
7         case (addr)
8             12'd0 : begin twiddle_real = 16'sd32767; twiddle_imag = 16'sd0; end
9             12'd1 : begin twiddle_real = 16'sd32767; twiddle_imag = -(16'sd51); end
10            12'd2 : begin twiddle_real = 16'sd32767; twiddle_imag = -(16'sd101); end
11            12'd3 : begin twiddle_real = 16'sd32767; twiddle_imag = -(16'sd151); end
12            12'd4 : begin twiddle_real = 16'sd32767; twiddle_imag = -(16'sd202); end
13            \pozostałe współczynniki
14            \...
15            12'd2046: begin twiddle_real = -(16'sd32767); twiddle_imag = -(16'sd101); end
16            12'd2047: begin twiddle_real = -(16'sd32767); twiddle_imag = -(16'sd51); end
17            default: begin twiddle_real = 16'sd32767; twiddle_imag = 16'sd0; end
18        endcase
19    end
20 endmodule

```

Listing 20: Fragment kodu z pliku twiddle_rom.sv

- **fft_tb.sv** - Testbench symulujący działanie całego systemu FFT. Inicjalizuje *clk*, *reset*, *start*, czeka na *done* z FFT, odczytuje dane z RAM (port A lub B) po zakończeniu FFT, zapisuje wyniki do pliku tekstowego *fft_output.txt* w formacie: < *real* >< *imag* >.

```

1  `timescale 1ns/1ps
2  module fft_tb;
3      reg clk;
4      reg reset;
5      reg start;
6      wire done;
7
8      reg signed [31:0] fft_val;
9      reg signed [15:0] fft_real, fft_imag;
10     integer i, outfile;
11
12 // Instancja modułu FFT 4096
13 fft_4096 fft_uut (
14     .clk(clk),
15     .reset(reset),
16     .start(start),
17     .done(done)
18 );
19
20 // Generowanie zegara
21 always #5 clk = ~clk;
22
23 initial begin
24     // Reset i start
25     clk = 0;
26     reset = 1;
27     start = 0;
28     #20;
29     reset = 0;
30     #10;
31     start = 1;
32     #10;
33     start = 0;
34
35     // Czekaj na sygnał done
36     @(posedge done);
37     $display("FFT computation completed. Reading results...");
```

38

```

39     // Otwórz plik wynikowy
40     outfile = $fopen("fft_output.txt", "w");
41     if (!outfile) begin
42         $display("Nie mogę otworzyć pliku wyjściowego!");
43         $finish;
44     end
```

Listing 21: Kod z pliku fft_tb.sv - 1

```

1 // Odczytaj kolejne próbki z pamięci RAM i zapisz do pliku
2 for (i = 0; i < 4096; i = i + 1) begin
3     fft_val = fft_uut.ram_inst.mem[i];           // dostęp do wewnętrznej pamięci RAM
4     fft_real = fft_val[31:16];
5     fft_imag = fft_val[15:0];
6     $fwrite(outfile, "%d %d\n", fft_real, fft_imag);
7 end
8
9 $fclose(outfile);
10 $display("Results written to fft_output.txt");
11 $finish;
12 end
13 endmodule

```

Listing 22: Kod z pliku fft_tb.sv - 2

5.3.4. Symulacja projektu w programie Modelsim oraz jej wyniki

Po ukończeniu komplikacji w programie Quartus Prime Lite można uruchomić symulację w programie Modelsim. Przygotowany został plik *fft_simulation.do*, który automatycznie dodaje wszystkie niezbędne sygnały w celu analizy, diagnostyki projektu.

```

# FFT Simulation DO File - Full Signal Waveform View
restart -force
vlib work
vlog -sv fft_tb.sv
vsim -voptargs=+acc fft_tb

# Dodaj wszystkie sygnały z FFT
add wave -divider "Top-Level Signals"
add wave -dec /fft_tb/clk
add wave -dec /fft_tb/reset
add wave -dec /fft_tb/start
add wave -dec /fft_tb/done

add wave -divider "FFT Internal Signals"
add wave -dec /fft_tb/fft_uut/state
add wave -dec /fft_tb/fft_uut/addr_a
add wave -dec /fft_tb/fft_uut/addr_b
add wave -dec /fft_tb/fft_uut/rom_addr
add wave -dec /fft_tb/fft_uut/tw_addr
add wave -dec /fft_tb/fft_uut/we_a
add wave -dec /fft_tb/fft_uut/we_b

add wave -divider "RAM Interface A"
add wave -dec /fft_tb/fft_uut/din_real_a
add wave -dec /fft_tb/fft_uut/din_imag_a
add wave -dec /fft_tb/fft_uut/dout_real_a
add wave -dec /fft_tb/fft_uut/dout_imag_a

```

Listing 23: Plik fft_simulation.do - 1

```

add wave -divider "RAM Interface B"
add wave -dec /fft_tb/fft_uut/din_real_b
add wave -dec /fft_tb/fft_uut/din_imag_b
add wave -dec /fft_tb/fft_uut/dout_real_b
add wave -dec /fft_tb/fft_uut/dout_imag_b

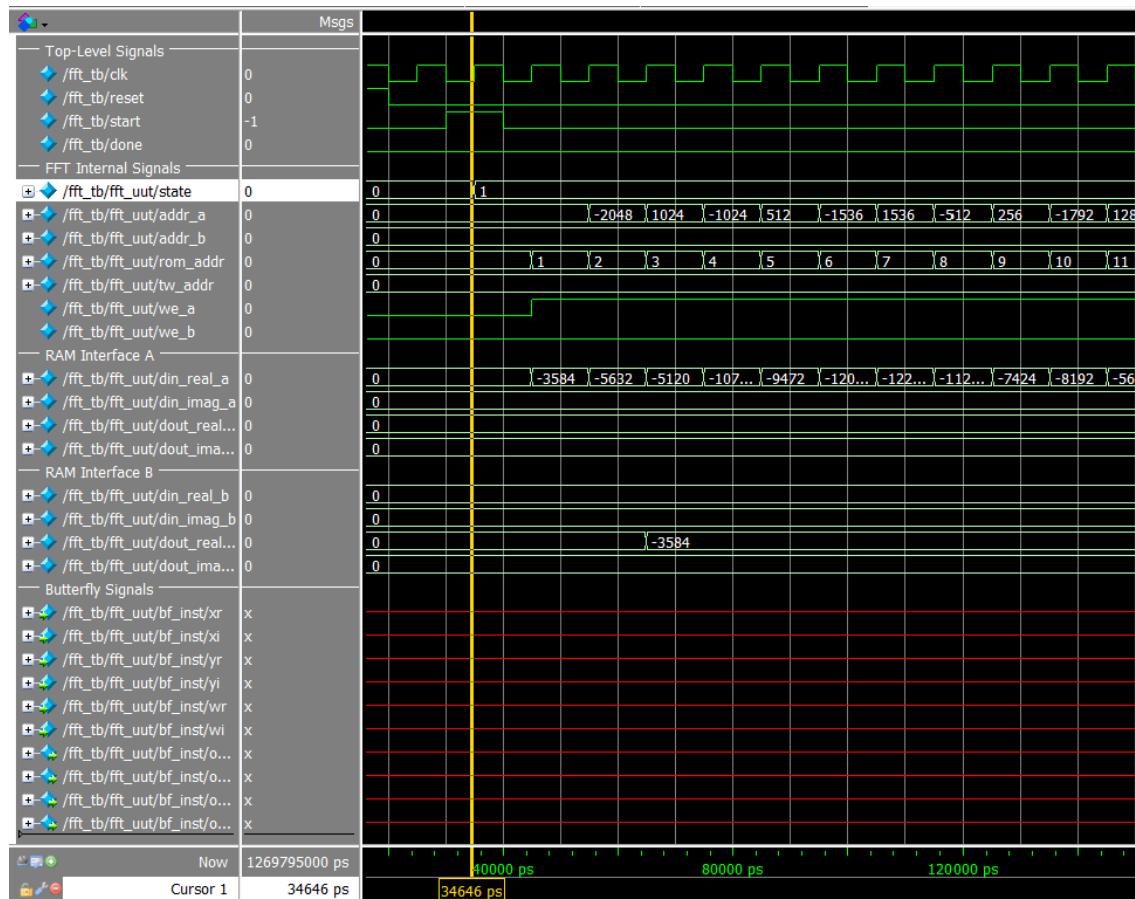
add wave -divider "Butterfly Signals"
add wave -dec /fft_tb/fft_uut/bf_inst/xr
add wave -dec /fft_tb/fft_uut/bf_inst/xi
add wave -dec /fft_tb/fft_uut/bf_inst/yr
add wave -dec /fft_tb/fft_uut/bf_inst/yi
add wave -dec /fft_tb/fft_uut/bf_inst/wr
add wave -dec /fft_tb/fft_uut/bf_inst/wi
add wave -dec /fft_tb/fft_uut/bf_inst/out0_r
add wave -dec /fft_tb/fft_uut/bf_inst/out0_i
add wave -dec /fft_tb/fft_uut/bf_inst/out1_r
add wave -dec /fft_tb/fft_uut/bf_inst/out1_i

run -all

```

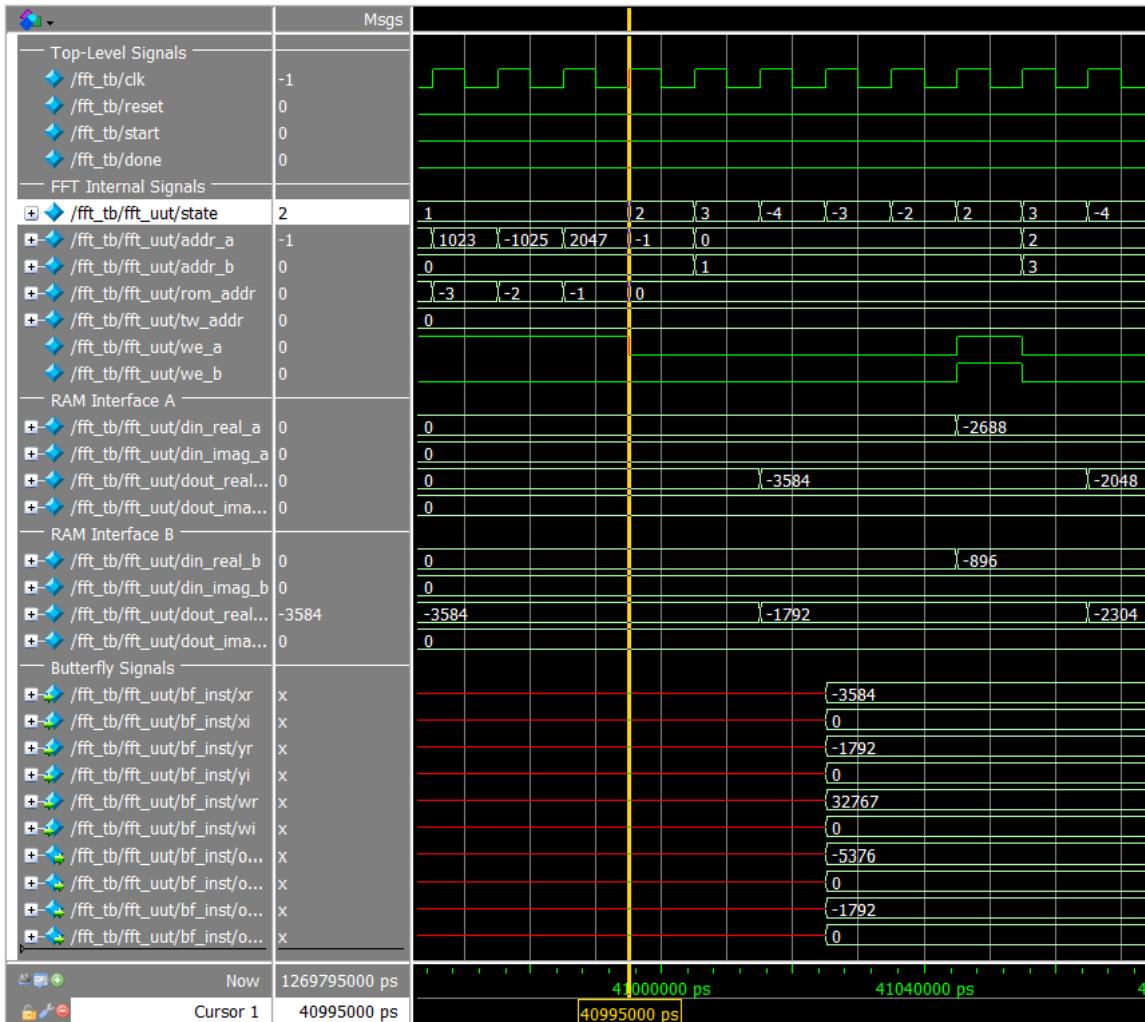
Listing 24: Plik fft_simulation.do - 2

Jako wynik działania pliku .do (po zakończeniu symulacji) zostanie wyświetlane okno z przebiegami czasowymi sygnałów. Można zaobserwować zmiany stanów FSM - po uruchomieniu symulacji zostaje wysłany sygnał *reset*. Następnie *start* - FSM przechodzi do stanu pierwszego, czyli *LOAD*:



Rys. 10. Pierwszy stan FSM - LOAD

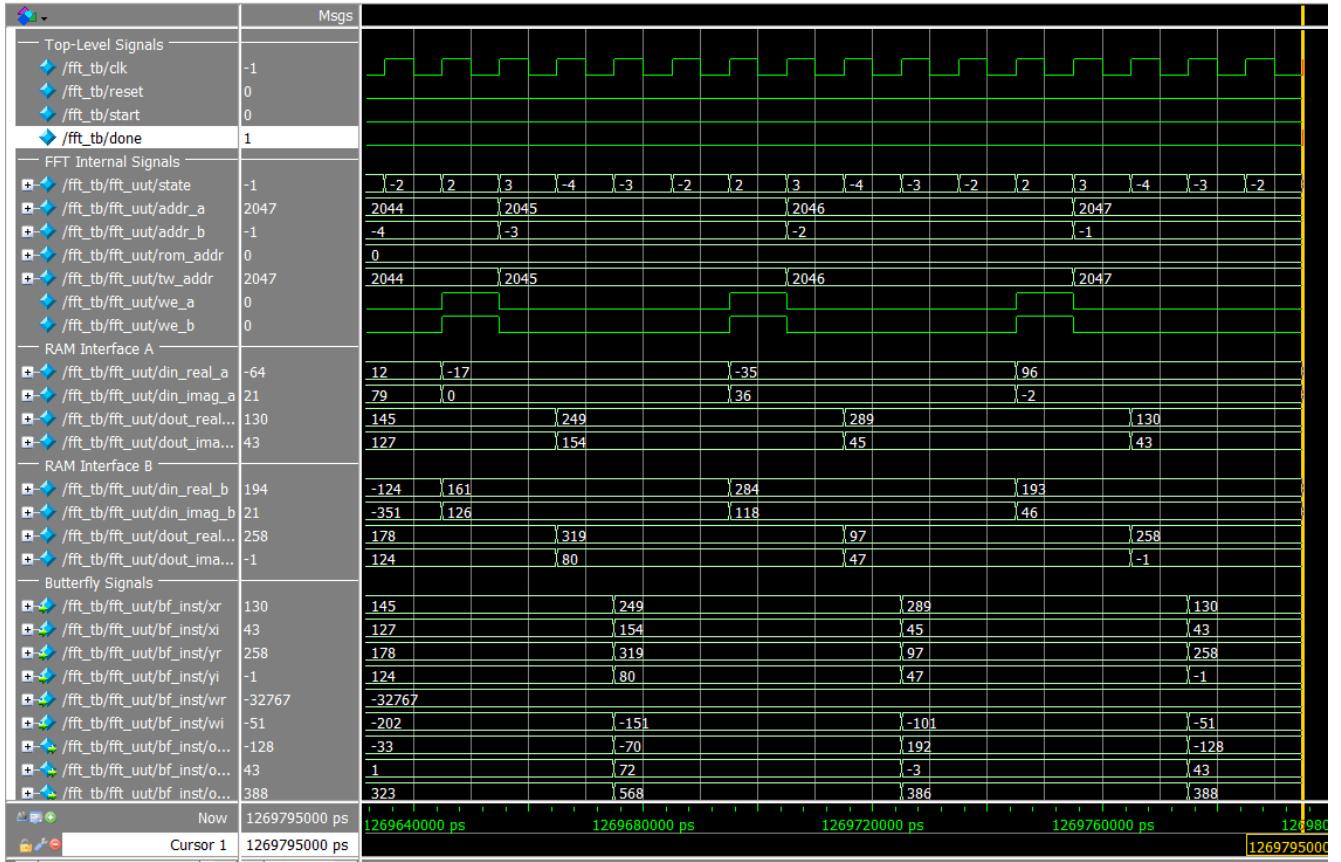
W tym stanie dane (próbki) pobierane są z pamięci ROM i zapisywane do RAM dla późniejszych obliczeń. Ten stan trwa do momentu aż nie zostaną przeiterowane wszystkie zadeklarowane komórki ROM:



Rys. 11. Następne stany FSM - obliczenia właściwe

Następnie rozpoczynają się obliczenia właściwe. Dane pobierane są z pamięci RAM, wykonywane są na nich operacje przy użyciu niezbędnych modułów i przetworzone dane wracają z powrotem do RAM.

Gdy przetworzone zostaną wszystkie dane przepisane z pamięci ROM do RAM, FSM wystawia stan wysoki na sygnale *done* - następuje zrzut przetworzonych danych do pliku - zajmuje się tym testbench.



Rys. 12. Działanie zakończone

Ponadto w lokalizacji, w której uruchamiana była symulacja, zostanie zapisany plik *fft_output.txt* z wynikami działania FFT:

```

1      5      0
2     -41     4
3      32    -96
4     -14    -23
5     -85    -59
6      24    -19
7     -23     41
8     -13     50
9     111    -46
10     81    -15
11    -54    -35
12      2    -30
13     ...    ...

```

Listing 25: Wynik działania FFT

Takich wyników będzie 4096 (tyle ile próbek) i są one postaci *<czesc_rzeczywista > <czesc_urojona >*.

5.3.5. Analiza wyników uzyskanych z FFT w Verilog

W celu wygenerowania widma z przetworzonych próbek sygnału napisany został kod w języku Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.signal import find_peaks
4
5 # Wczytanie danych z pliku
6 data = np.loadtxt('fft_output.txt')
7
8 # Podział na części rzeczywiste i urojone
9 real_part = data[:, 0]
10 imag_part = data[:, 1]
11
12 # Obliczenie amplitud
13 amplitudes = np.sqrt(real_part**2 + imag_part**2)
14
15 # Normalizacja amplitud
16 amplitudes = amplitudes / len(amplitudes)
17
18 # Generowanie osi częstotliwości
19 N = len(amplitudes)
20 fs = 44100
21 frequencies = np.fft.fftfreq(N, 1/fs)
22
23 # Wybór tylko częstotliwości dodatnich (prawa połowa)
24 positive_freq = frequencies[:N//2]
25 positive_amp = amplitudes[:N//2]
26
27 # Wykrywanie pików
28 peaks, _ = find_peaks(positive_amp, height=0.1*np.max(positive_amp),
29                         distance=3)
30
31 # Znajdowanie 5 najwyższych pików
32 if len(peaks) > 0:
33     top_peaks_indices = np.argsort(positive_amp[peaks])[-5:][::-1]
34     top_peaks = peaks[top_peaks_indices]
35
36 # Rysowanie widma
37 plt.figure(figsize=(12, 6))
38 plt.plot(positive_freq, positive_amp, label='Widmo')
39 plt.scatter(positive_freq[peaks], positive_amp[peaks],
40             color='red', label='Wszystkie piki')
```

Listing 26: Kod generujący widmo z wyników FFT w Verilog - 1

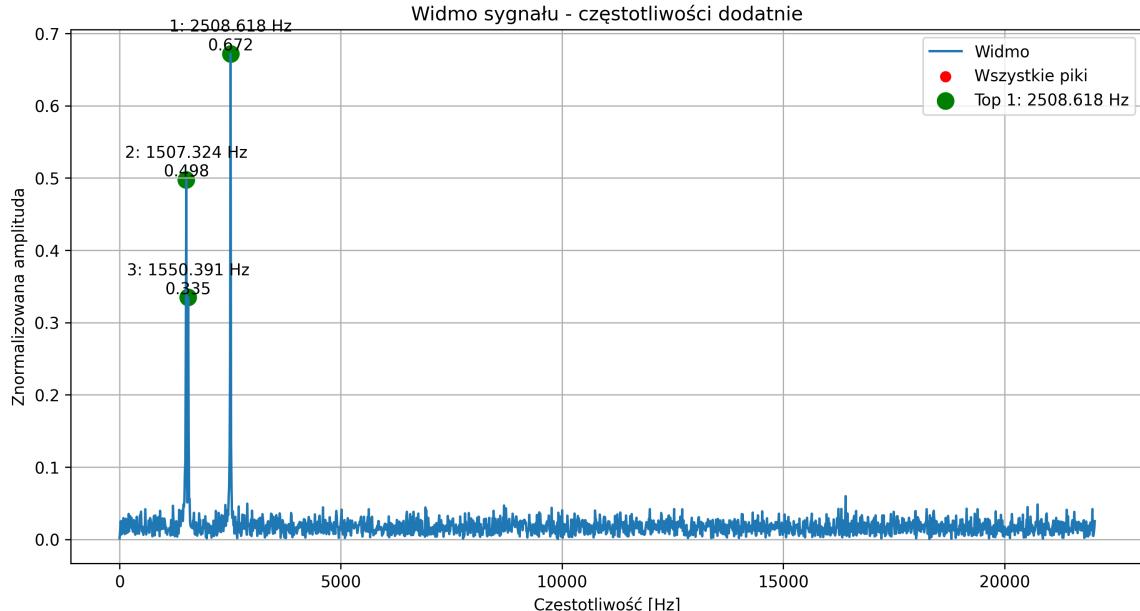
```

1 # Zaznaczanie i etykietowanie najwyższych pików
2 if len(peaks) > 0:
3     for i, peak in enumerate(top_peaks):
4         plt.scatter(positive_freq[peak], positive_amp[peak] ,
5                     color='green', s=100,
6                     label=f'Top {i+1}: {positive_freq[peak]:.3f} '
7                           f'Hz' if i == 0 else None)
8         plt.text(positive_freq[peak], positive_amp[peak] ,
9                  f'{i+1}: {positive_freq[peak]:.3f} '
10                 f'Hz\n{positive_amp[peak]:.3f}' ,
11                 ha='center', va='bottom')
12
13 plt.title('Widmo sygnału - częstotliwości dodatnie')
14 plt.xlabel('Częstotliwość [Hz]')
15 plt.ylabel('Znormalizowana amplituda')
16 plt.grid(True)
17 plt.legend()
18
19 # Wyświetlanie informacji o pikach
20 if len(peaks) > 0:
21     print("Znalezione piki na częstotliwościach:")
22     for i, peak in enumerate(top_peaks):
23         print(f"Top {i+1}: {positive_freq[peak]:.6f} Hz, "
24               f"amplituda: {positive_amp[peak]:.6f}")
25
26 plt.show()

```

Listing 27: Kod generujący widmo z wyników FFT w Verilog - 2

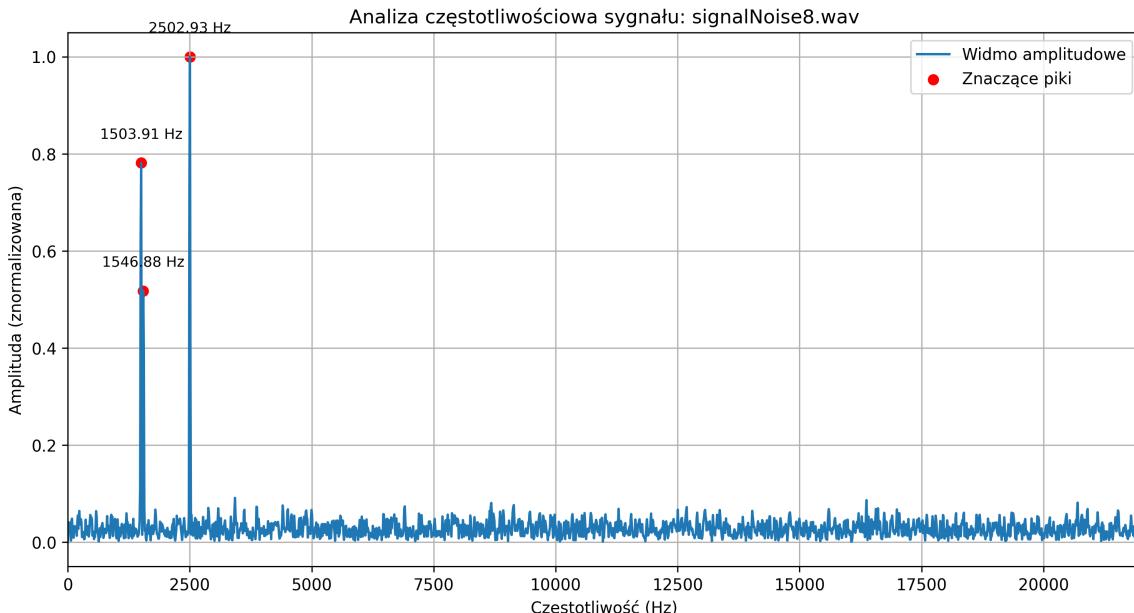
Wynikiem działania powyższego kodu jest obraz widma:



Rys. 13. Widmo z wyników FFT

5.3.6. Porównanie widm wygenerowanych przy pomocy kodu referencyjnego z pliku .wav oraz widma po obliczeniu FFT przy pomocy Verilog

Dla przypomnienia - tak wyglądało widmo sygnału signalNoise8.wav wygenerowane w rozdziale 5.2.1

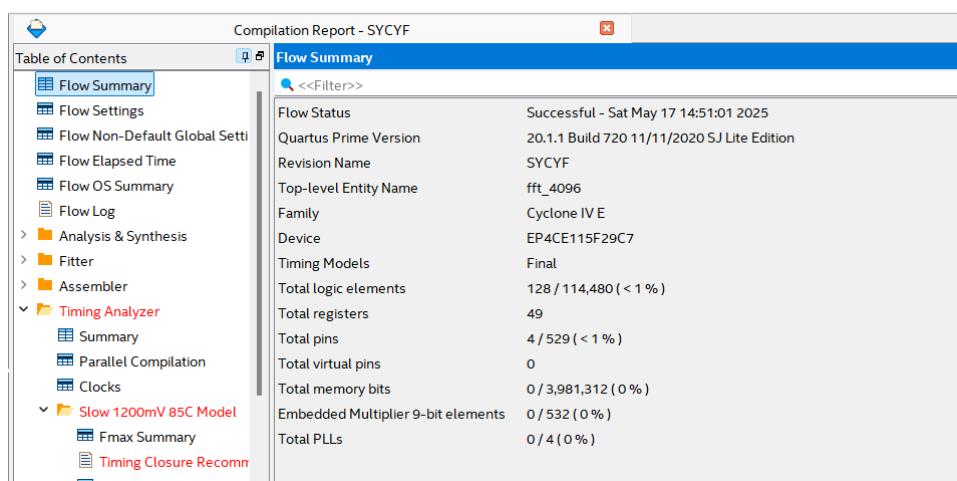


Rys. 14. Widmo signalNoise8.wav - Python

Widmo utworzone z wyników FFT przeprowadzonej w kodzie Verilog na próbkach z pliku signalSample.txt praktycznie idealnie odwzorowuje widmo sygnału signalNoise8.wav. Pojawiły się tylko nieznaczne rozbieżności rzędu kilku herców, ponieważ w implementacji FFT w języku Verilog zastosowano reprezentację stałoprzecinkową (Q1.15). Język Python potrafi wykonać bez przeszkód obliczenia na liczbach zmiennoprzecinkowych. Ponadto tabela wartości współczynników twiddle ($e^{-j2\pi k/N}$) z Verilog została zapisana z pewnym przybliżeniem (normalnie występują tam wartości niewymierne).

5.3.7. Zasoby niezbędne do działania projektu

Poniższe wyniki zostały odczytane z raportu po komplikacji projektu i poszczególnych modułów.



Rys. 15. Zajęte zasoby - cały projekt

Flow Status	Successful - Sat Jun 07 18:20:07 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 Patches 1.07std SJ Lite Edition
Revision Name	SYCYF
Top-level Entity Name	butterfly
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	126 / 114,480 (< 1 %)
Total registers	0
Total pins	160 / 529 (30 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	8 / 532 (2 %)
Total PLLs	0 / 4 (0 %)

Rys. 16. Zajęte zasoby - butterfly

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sat Jun 07 18:18:48 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 Patches 1.07std SJ Lite Edition
Revision Name	SYCYF
Top-level Entity Name	fft_4096
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	128 / 114,480 (< 1 %)
Total registers	49
Total pins	4 / 529 (< 1 %)
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. 17. Zajęte zasoby - fft

Flow Summary	
<<Filter>>	
Flow Status	Flow Failed - Sat Jun 07 18:56:11 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 Patches 1.07std SJ Lite Edition
Revision Name	SYCYF
Top-level Entity Name	ram_4096
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	314,097 / 114,480 (274 %)
Total registers	131136
Total pins	155 / 529 (29 %)
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. 18. Zajęte zasoby - ram

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sat Jun 07 18:21:55 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 Patches 1.07std SJ Lite Edition
Revision Name	SYCYF
Top-level Entity Name	rom_input
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,268 / 114,480 (3 %)
Total registers	0
Total pins	44 / 529 (8 %)
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. 19. Zajęte zasoby - rom

Projekt, według kompilatora, zajmie zaledwie 128 bloków i 49 rejestrów co jest wynikiem bardzo dobrym. Stanowi to poniżej 1% wszystkich dostępnych zasobów, pomimo że przeprowadzane obliczenia są dosyć skomplikowane i angażujące kilka odrębnych modułów.

Compilation Report - SYCYF				
Slow 1200mV 85C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	136.48 MHz	136.48 MHz	clk	

Table of Contents

- Parallel Compilation
- Clocks
- Slow 1200mV 85C Model
 - Fmax Summary
 - Timing Closure Recom
 - Setup Summary
 - Hold Summary
 - Recovery Summary
 - Removal Summary
 - Minimum Pulse Width S
 - Worst-Case Timing Path
 - Metastability Summary
- Slow 1200mV OC Model
- Fast 1200mV OC Model

Rys. 20. Maksymalna częstotliwość zegara - raport

Biorąc pod uwagę, że wbudowany zegar na płytce FPGA dostępnej w laboratorium taktuje z częstotliwością 50 MHz, prędkość działania kodu realizującego FFT jest zadowalająca - kod bez problemu działałby na pełnej prędkości zegara bez konieczności jego spowolniania (stosowania clock-divider'a) podczas fizycznej implementacji na płytce FPGA. Jednak, w celu udowodnienia poprawności fizycznej implementacji na płytce FPGA, zegar może zostać spowolniony w celu "sztucznego" spowolnienia prędkości działania projektu.

6. Uruchomienie

Wyniki uruchomienia projektu zostały zaprezentowane w test_bench'u.

7. Podsumowanie

Projekt zakończył się pełnym sukcesem, realizując założony cel stworzenia systemu do analizy widma sygnału z łazika marsjańskiego. Wykorzystując metodę Design Thinking - Double Diamond, zespół przeszedł przez wszystkie etapy od analizy problemu po finalną implementację.

Kluczowe osiągnięcia:

- **Implementacja referencyjna w Pythonie** - stworzono funkcjonalny kod z własną implementacją FFT, wizualizacją wyników i obsługą plików .wav
- **Pełna implementacja sprzętowa w Verilog** - zrealizowano 4096-punktową FFT z wykorzystaniem reprezentacji stałoprzecinkowej Q1.15
- **Wysoką dokładność** - wyniki implementacji Verilog praktycznie idealnie odwzorowują kod referencyjny

Weryfikacja i testowanie: Przeprowadzono kompleksowe testy z wykorzystaniem MATLAB-a, analizatora widma online oraz symulacji ModelSim, potwierdzając poprawność działania na wszystkich etapach. System został przetestowany na rzeczywistych próbkach sygnału signalNoise8.wav.

7.1. Wnioski

Metodologiczne:

- Etapowy proces rozwoju od prototypu programowego do implementacji sprzętowej pozwolił na wczesne wykrycie i eliminację błędów projektowych
- Weryfikacja wieloetapowa (Python → MATLAB → Verilog) znaczaco zwiększyła pewność poprawności rozwiązania

Techniczne:

- Implementacja FFT bez wykorzystania IP Core jest możliwa i efektywna zasobowo, co daje pełną kontrolę nad algorytmem
- Reprezentacja stałoprzecinkowa Q1.15 zapewnia wystarczającą dokładność dla większości zastosowań praktycznych przy znacznej oszczędności zasobów
- Architektura z maszyną stanów (FSM) i dwuportową pamięcią RAM umożliwia efektywne przetwarzanie pipeline'owe

Praktyczne:

- System jest skalowalny - można łatwo dostosować liczbę punktów FFT poprzez zmianę parametrów
- Wysoka częstotliwość pracy pozwala na przetwarzanie sygnałów w czasie rzeczywistym

Edukacyjne:

- Projekt pokazał praktyczne zastosowanie teorii transformaty Fouriera w systemach cyfrowych
- Demonstrował różnice między implementacją programową a sprzętową oraz ich wpływ na dokładność obliczeń
- Podkreślił znaczenie systematycznego testowania i weryfikacji w projektowaniu systemów cyfrowych

Literatura

- [1] Wikipedia. Design Thinking – Wikipedia, 2025. https://en.wikipedia.org/wiki/Design_thinking, Dostęp zdalny 20.03.2025.
- [2] Dr Sylwia Wrona. Design Thinking, 2025. https://www.parp.gov.pl/attachments/article/69720/CRMSP_Design_Thinking_Sylwia_Wrona.pdf, Dostęp zdalny 20.03.2025.
- [3] EY Polska. Metodyki zarządzania projektami w pigułce, 2023. https://www.ey.com/pl_pl/insights/consulting/metodyki-zarzadzania-projektami-w-pigulce/, Dostęp zdalny 01.04.2025.
- [4] Marcel Mitraszewski. Czym jest double diamond, design thinking i service design?, 2025. <https://szkoladizajnu.pl/blog/projektowanie/double-diamond-design-thinking-service-design/>, Dostęp zdalny 20.03.2025.
- [5] Software-Defined Radio, 2024. https://en.wikipedia.org/wiki/Software-defined_radio, Dostęp zdalny 01.04.2025.
- [6] Program HDSDR. <https://www.hdsdr.de/>, Dostęp zdalny 01.04.2025.
- [7] Program GNU Radio. https://wiki.gnuradio.org/index.php?title=Main_Page, Dostęp zdalny 01.04.2025.
- [8] Program Audacity. <https://manual.audacityteam.org/#>, Dostęp zdalny 01.04.2025.
- [9] Signal Processing Toolbox - Matlab. <https://www.mathworks.com/products/signal.html>, Dostęp zdalny 01.04.2025.
- [10] balzer82. Przykłady implementacji transformat Fouriera. <https://github.com/balzer82/FFT-Python/>, Dostęp zdalny 02.04.2025.
- [11] Biblioteka SciPy. <https://docs.scipy.org/doc/scipy/>, Dostęp zdalny 01.04.2025.
- [12] Biblioteka NumPy. <https://numpy.org/doc/stable/>, Dostęp zdalny 01.04.2025.
- [13] Biblioteka Matplotlib. <https://matplotlib.org/stable/users/index.html>, Dostęp zdalny 01.04.2025.
- [14] Svantec Akademia. Transformata Fouriera, FFT, DFT. <https://svantek.com/pl/akademia/fft-transformata-fouriera/>, Dostęp zdalny 01.04.2025.
- [15] R. W. Schafer A. V. Oppenheim. Discrete-Time Signal Processing, 1998. https://research.iaun.ac.ir/pd/naghsh/pdfs/UploadFile_2230.pdf, Dostęp zdalny 01.04.2025.
- [16] Prof zw dr. hab. inż. Sławomir Tumański. Dyskretna Transformata Fouriera. <http://www.tumanski.pl/lab3.pdf>, Dostęp zdalny 02.04.2025.
- [17] Christopher Chang. Analizator widma online. <https://www.checkhearing.org/audiospectrum.php>, Dostęp zdalny 22.04.2025.
- [18] Basic Spectral Analysis. <https://www.mathworks.com/help/matlab/math/basic-spectral-analysis.html>, Dostęp zdalny 22.04.2025.
- [19] How to plot frequency spectrum of a signal in matlab? <https://www.mathworks.com/matlabcentral/answers/251061-how-to-plot-frequency-spectrum-of-a-signal-in-matlab>, Dostęp zdalny 22.04.2025.