AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

INSTYTUT ELEKTRONIKI

Systemy Dedykowane w Układach Programowalnych

Szybka Transformacja Fouriera - FFT

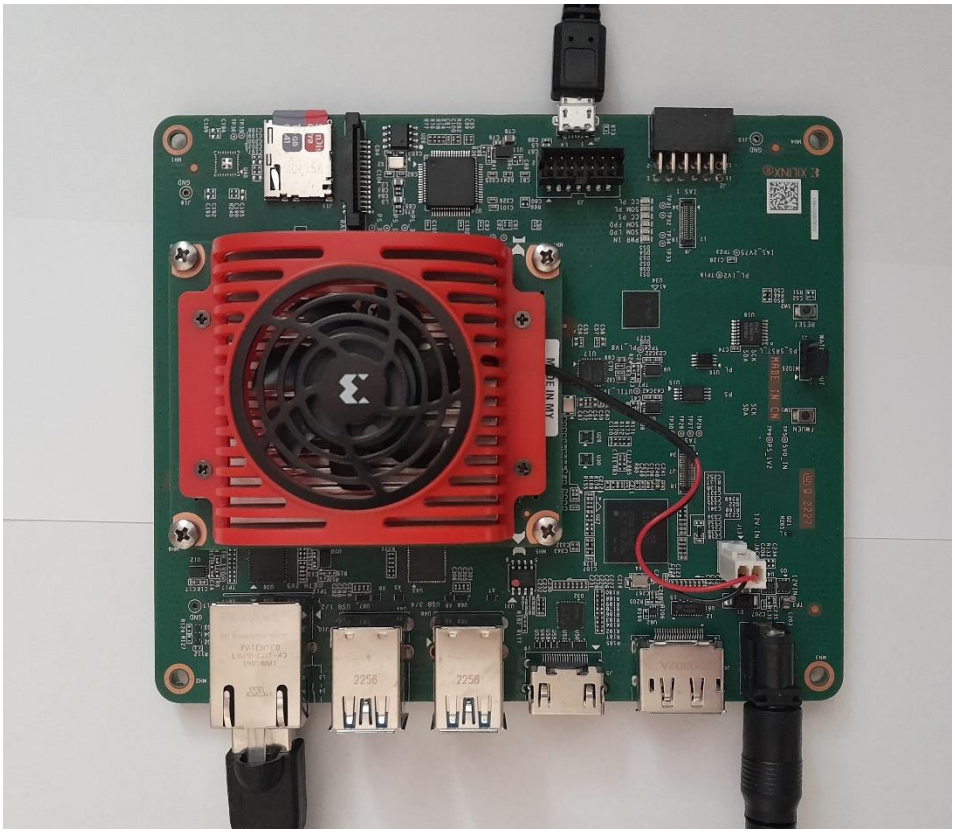
Autorzy: *Maciej Zieliński, Piotr Wilkoń*
Kierunek studiów: *Elektronika i Telekomunikacja*

Kraków, 2024

Spis treści

1. Wstęp

Celem niniejszego projektu było zaprojektowanie oraz zaimplementowanie algorytmu szybkiej transformacji Fouriera (FFT – *Fast Fourier Transform*) w układzie FPGA oraz opracowanie zewnętrznego oprogramowania umożliwiającego praktyczne wykorzystanie koprocesora i demonstrację poprawności jego działania. Do realizacji projektu wykorzystano Vivado 2022.1, Pycharm, Python. 3.9, platforma PYNQ oraz płytkę Kria KV260 (Rys.1).



Rys. 1: Płytka rozwojowa Kria KV260

Transformacja Fouriera jest jedną z podstawowych transformacji matematycznych. Używana jest ona szczególnie często w technice, gdzie służy do analizy oraz syntezy sygnałów dźwiękowych, telekomunikacyjnych lub obrazów.

Wykonanie projektu rozpoczęto od teoretycznej analizy zagadnienia. Początkowo algorytm zaimplementowano w języku Matlab oraz Python, a następnie przeniesiono go do języka Verilog, gdzie wykonano pierwsze symulacje działania. W pierwszej wersji algorytm zrealizowany był całkowicie sekwencyjnie, tj. dane były przetwarzane po kolei przez każdy krok algorytmu. Następnie program został podzielony na funkcjonalne moduły, co znacznie zwiększyło elastyczność i umożliwiło stworzenie kolejnej wersji projektu. Wersja ta zakładała równoległe wykonywanie kolejnych (opisanych szczegółowo w późniejszych sekcjach dokumentu) kroków, co skutkowało istotnym zwiększeniem wydajności algorytmu. Ze względu na blokową naturę FFT ułatwiona była także dalsza rozbudowa o swoistą potokowość, która umożliwiła dodatkowe zwiększenie szybkości.

2. Opis teoretyczny algorytmu

2.1. Transformacja Fouriera

Transformacja Fouriera (FT – *Fourier Transform*), w ujęciu technicznym, jest operacją pozwalającą na przekształcenie dowolnych danych w funkcji czasu na odpowiadające dane w funkcji częstotliwości, tj. na obliczenie *widma* danego sygnału wejściowego. Zarówno dane wejściowe, jak i wyjściowe, są liczbami zespolonymi, dzięki czemu możliwa jest reprezentacja zależności fazowych w sygnale. Sama transformacja Fouriera operuje na danych ciągłych, stąd jest nierealizowalna w oprogramowaniu. Istnieje natomiast jej odpowiednik operujący na policzalnym zbiorze danych, zwany dyskretną transformacją Fouriera (DFT – *Discrete Fourier Transform*). Algorytm DFT jest niemalże identyczny, jak ten dla FT, z tym że operuje na dyskretnych danych oraz współczynnikach. Jego zaletą jest możliwość obliczenia transformaty dla zbioru danych o dowolnej wielkości, niemniej będąc algorytmem wymagającym czasowo: jego złożoność obliczeniowa jest proporcjonalna do kwadratu ilości danych ($O(n^2)$), co wiąże się z ogromnymi czasami obliczeń dla dużych zbiorów, niejednokrotnie zbyt dużymi do praktycznego zastosowania.

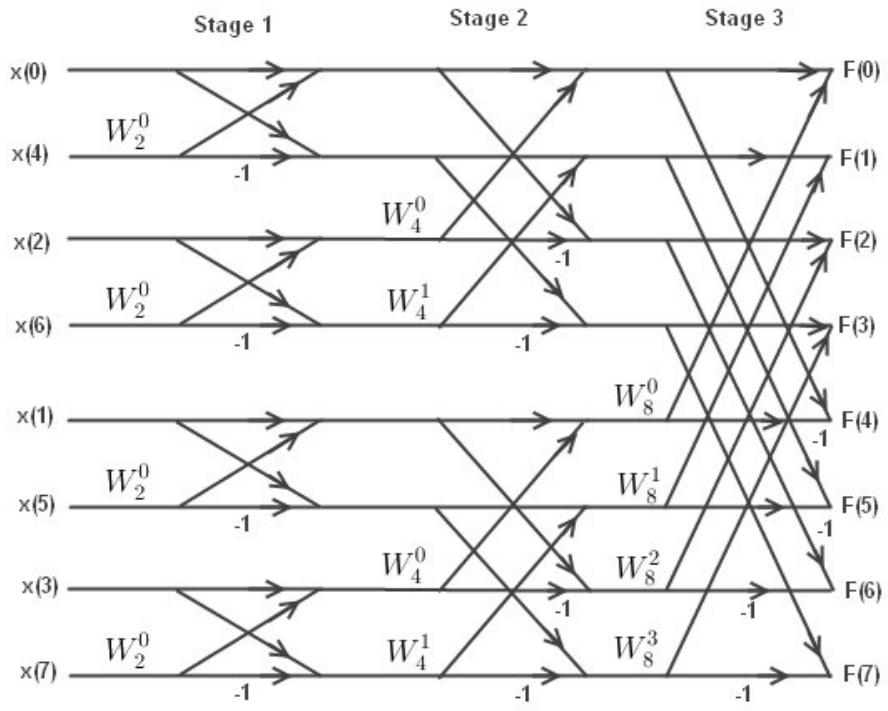
2.2. Szybka transformacja Fouriera

Przy pewnych założeniach co do ilości danych problem dużej złożoności obliczeniowej jest jednak rozwiązywalny. Algorytm Cooleya-Tukeya polega na rozbiciu DFT o długości N na N_1 transformacji o długości N_2. Algorytm ten można zastosować rekurencyjnie, ostatecznie dochodząc do trywialnego DFT o długości 1. Szczególnym przypadkiem tego algorytmu jest rekursywny podział DFT o długości N na dwa DFT o długości $N/2$. Takie wyrażenie algorytmu nazywane jest lematem Danielsona-Lanczosa. W ogólności, na mocy lematu Danielsona-Lanczosa, DFT można obliczyć jako sumę dwóch DFT po elementach parzystych oraz nieparzystych wraz z odpowiednim współczynnikiem:

$$X_n = \sum_{k=0}^{\frac{N}{2}-1} x_{2k} e^{\frac{-j2k\pi n}{\frac{N}{2}}} + W_N^n \sum_{k=0}^{\frac{N}{2}-1} x_{2k+1} e^{\frac{-j2k\pi n}{\frac{N}{2}}},$$

gdzie X_n oraz x_n to, odpowiednio, elementy wektora wyjściowego i wejściowego, a $W_N^n = e^{\frac{-j2\pi n}{N}}$ jest współczynnikiem zwanym z ang. *twiddle factor*. Jeśli założymy, że dane wejściowe mają długość równą $N = 2^r$, a r jest liczbą naturalną większą od zera, to takie DFT może zostać łatwo obliczone poprzez rekurencyjny podział i przeplot mniejszych transformacji.

Algorytm Cooleya-Tukeya, dla N będącego potęgą liczby 2, można zaprezentować za pomocą tzw. *diagramu motylkowego*. Diagram taki, dla $N=8$, przedstawiono na Rys. 2.



Rys. 2: Diagram motylkowy dla FFT o długości 8 [1]

Trzeba zauważyć, że indeksy elementów wejściowych są w nietypowej kolejności. Kolejność tę uzyskuje się przez odwrócenie bitów indeksów. Odwrócenie bitów indeksów wejściowych nazywane jest *decymacją w czasie* (DIT – *decimation-in-time*) i jest najczęściej spotykanym rozwiązaniem, choć możliwa jest również *decymacja w częstotliwości* (DIF – *decimation-in-frequency*). Operacja odwrócenia bitów jest konieczna dla zbudowania diagramu motylkowego i efektywnego obliczenia DFT. Analizując podany wcześniej wzór dla kolejnych kroków algorytmu można zauważyć, że do obliczenia każdego kolejnego kroku wykorzystywane są jedynie wyniki poprzednich kroków oraz współczynniki *twiddle factor* i nie jest konieczne obliczenie wartości eksponencjalnych czynników widocznych we wzorze. Ciekawą właściwością opisywanego algorytmu jest możliwość obliczeń *w miejscu* (ang. *in place,* łac. *in situ*), tj. użycie pamięci niezależne od długości danych, a także znacznie zmniejszona złożoność obliczeniowa ($O(N*log_2(N))$). Dzięki temu algorytm ten jest jednym z przykładów szybkiej transformacji Fouriera (FFT – *Fast Fourier Transform*).

Na diagramie można zauważyć, że w każdym etapie (*stage*) wykorzystywana jest taka sama liczba motylków (przecinających się linii) równa *N/2*, a liczba etapów jest zawsze równa $log_2(N)$. Ponadto wszystkie motylki w danym etapie korzystają z innych danych i nie wpływają na siebie wzajemnie. Tym samym algorytm ten dobrze nadaje się do implementacji w układach FPGA, po pierwsze umożliwiając równoległe wykonywanie obliczeń przez wszystkie motylki, a po drugie naturalnie dzieląc algorytm na kroki, umożliwiając zastosowanie struktury typu *pipeline*.

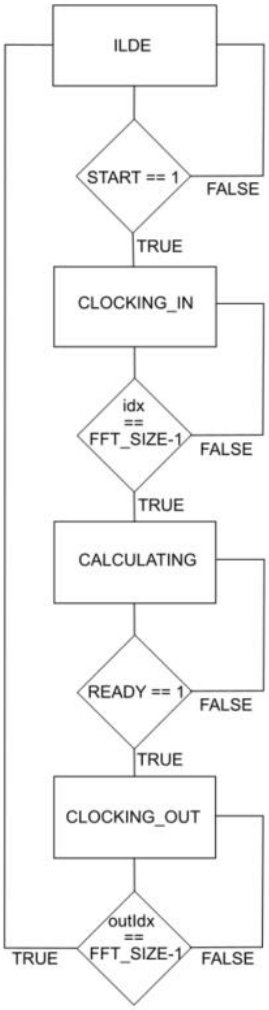
3. Model behawioralny

Początkowo sporządzono model behawioralny FFT w językach wysokiego poziomu: Python oraz Matlab. Pierwszy z nich, napisany w języku Python, został zaimplementowany rekurencyjnie (co wynika z natury algorytmu Cooleya-Tukeya) i służył następnie do weryfikacji poprawności działania implementacji sprzętowej. Algorytm w języku Matlab został zaimplementowany sekwencyjnie i posłużył jako wzór do implementacji w języku Verilog.

4. Projekt Verilog

Pierwsza wersja algorytmu zakładała sekwencyjne wejście danych, a następnie wykonywanie pojedynczo każdej z operacji motylka. Algorytm poruszał się zgodnie z maszyną stanów przedstawioną na Rys. 3. Algorytm rozdzielony został na kilka modułów:

- *fft.v* - moduł główny, w który znajduje się maszyna stanów oraz generowanie współczynników ang. twiddles wykorzystywanych do obliczeń.
- *reverse.v* - moduł odpowiedzialny za odwrócenie bitów indeksów danych wejściowych, aby uzyskać dane wyjściowe w odpowiedniej kolejności
- *butterfly.v* - moduł wykonujący obliczenia dla pojedynczego "motylka"
- *cplxmul.v* - moduł odpowiedzialny za mnożenie liczb zespolonych wykorzystywanie w module butterfly.v.



Rys. 3: Maszyna stanów

Opis maszyny stanów:
- *ILDE* – oczekiwanie na zewnętrzny sygnał startu
- *CLOCKING_IN* – stan odpowiedzialny za pobieranie danych wejściowych i ustawianie ich w odpowiedniej kolejności w buforze. Dane pobierane są z wejścia modułu z każdym taktem zegara, aż do momentu pobrania odpowiedniej liczby danych określonych przez parametr *FFT_SIZE*.
- *CALCULATING* – stan, w którym obliczane jest FFT z danych wejściowych.
- *CLOCKING_OUT* – stan odpowiedzialny za wysyłanie danych do wyjścia. Kolejne dane wysyłane są wraz z każdym cyklem zegara.

Ważnym elementem utworzonego algorytmu jest jego responsywność, która umożliwia dostosować rozmiar obliczanego FFT poprzez ustawienie parametru *FFT_SIZE*.

Kolejna wersja algorytmu zakłada zwiększenie szybkości poprzedniej wersji. Z racji, że w każdym kolejnym stopniu FFT wykonywana jest ta sama ilość operacji motylkowych, zdecydowano się je zrównoleglić. W tym celu w module *fft.v* tworzone jest kilka instancji modułu *butterfly.v.* Ilość tworzonych motylków zależy od rozmiaru FFT, a dokładnie od $\log_2$(Rozmiar FFT). Wykorzystane moduły oraz maszyna stanów zaczerpnięta została z pierwszej wersji algorytm

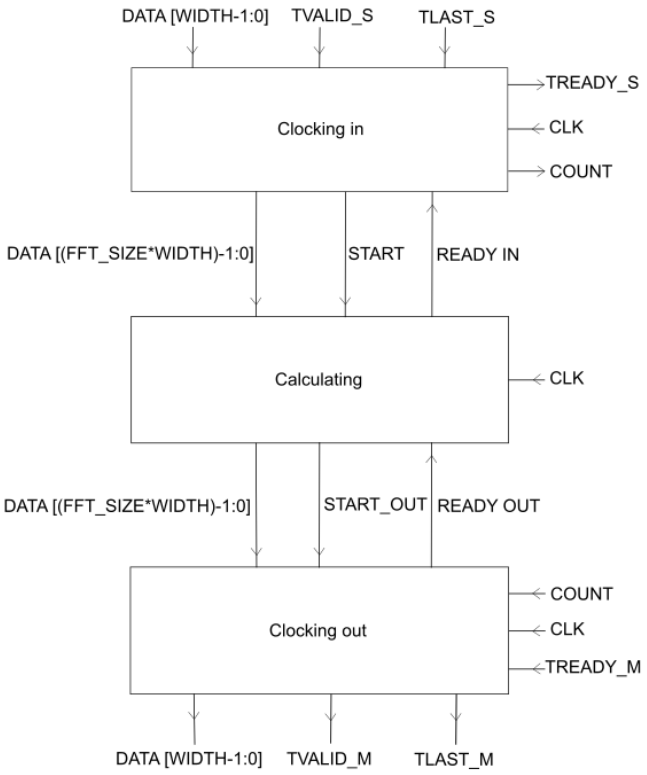
Trzecią wersją był algorytm, który pomijał maszynę stanów oraz zakładał, że wszystkie dane podawane są jednocześnie na wejście modułu. Następnie wykonywał on operacje odwracania bitów indeksów danych wejściowych. Kolejnym krokiem było stworzenie $\log_2$(Rozmiar_FFT) stopni FFT, które wykonywane były jeden po drugim. W każdym stopniu tworzona była odpowiednia ilość instancji motylków, a następnie przydzielono każdej z instancji odpowiednie dane wejściowe. Po wykonaniu każdego stopnia otrzymane dane przekazywane był do kolejnych stopni. Ostatni stopień przekazywał dane na wyjście modułu. Algorytm również napisany został w sposób responsywny dla różnej wartości FFT.

Algorytm rozdzielony został na kilka modułów:
- *fft.v* - moduł główny, w którym generowana jest odpowiednia ilość stopni FFT.
- *stageN.v* – moduł reprezentujący pojedynczy stopień FFT, generuje odpowiednią ilość motylków
- *reverse.v* - moduł odpowiedzialny za odwrócenie bitów indeksów danych wejściowych.
- *butterfly.v* - moduł wykonujący obliczenia dla pojedynczego “motylka”.
- *cplxmul.v* - moduł odpowiedzialny za mnożenie liczb zespolonych.
- *twiddle.v* - moduł generujący odpowiednią ilość współczynników dla danego stopnia FFT.

Każda z wersji algorytmu przechodziła poprawnie tworzone symulacje oraz syntezę. W algorytmach wykorzystywane są liczby 24-bitowe, gdzie 14 bitów to część całkowita natomiast 10 bitów przeznaczono na cześć ułamkową.

Ostateczna, czwarta wersja projektu to zmodyfikowany algorytm z wersji 2 rozwinięty o działanie potokowe. Zbudowaną maszynę stanu rozdzielono na 3 osobne moduły: clocking in, calculating, clocking out. Pozwalały one odpowiednio pobierać dane, obliczać FFT oraz wysłać dane wynikowe. Główną cechą jest to, że każdy z modułów pracuje niezależnie, więc możliwe jest wykonywanie równocześnie odpowiednich etapów dla 3 kolejnych zestawów danych. Z racji, że układ obliczający trwa najwięcej cykli zegara, więc to on wymusza czas trwania pozostałych układów. Układ pobierający dane wejściowe zapisuje dane z każdym cyklem zegara, aż do momentu zebrania 8 próbek (rozmiar FFT). Następnie oczekuje na sygnał gotowości do pobrania nowych danych przez układ obliczający. Po takim sygnale przekazuje zebrane dane i zaczyna ponownie pobierać kolejny zestaw próbek. Moduł obliczający po odbiorze zestawu nowych danych oblicza FFT, a po zakończeniu obliczeń przekazuje wynik do układu wysyłającego. Ostatni moduł po otrzymaniu wyniku z poprzedniego bloku, wystawia z każdym cyklem zegara pojedynczo każdą z liczb

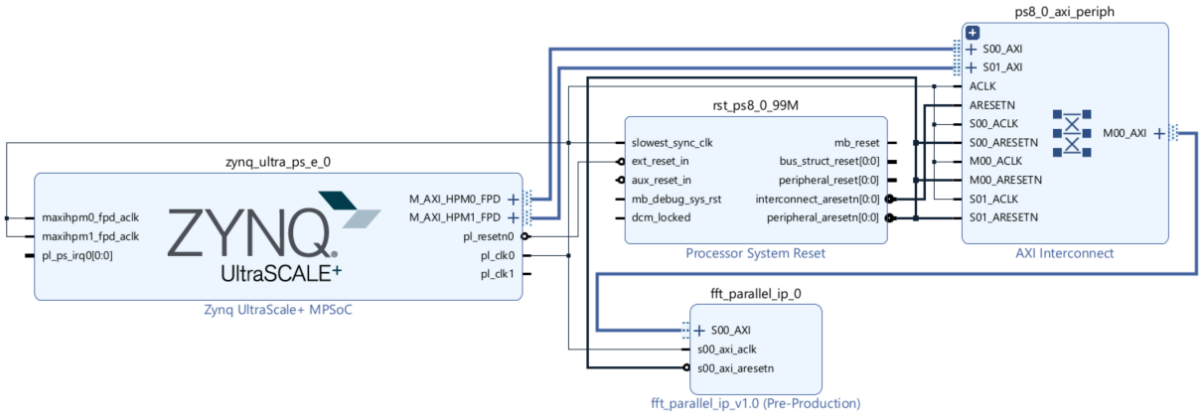
wynikowych na wyjście całego układu FFT. Po wystawieniu wszystkich liczb (8) ponownie oczekuje na dane z układu obliczającego. Na Rys. 4 przedstawiono schemat blokowy działania całego modułu FFT wraz ze wszystkimi sygnałami wejściowym i wyjściowymi dla magistrali AXI Stream. Projekt skonfigurowano dla FFT o rozmiarze 8 oraz danych 16 bitowych (format 10:6).



Rys. 4: Schemat blokowy (pipeline FFT).

5. PYNQ

Ostatnim etapem projektu było przetestowanie stworzonych algorytmów ze sprzętem. W tym celu stworzone algorytmy zamknięto w osobne układy IP-Core, a następnie połączono z procesorem przy użyci magistrali AXI-Lite. Przykładowy diagram blokowy dla jednego z algorytmów przedstawiono na Rys. 5.



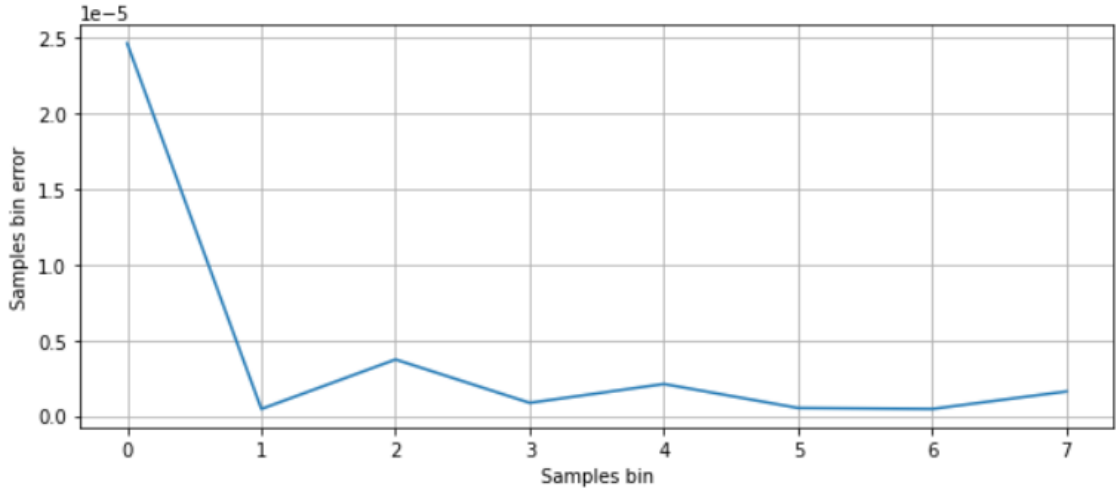
Rys. 5: Schemat blokowy układu dla drugiej wersji algorytmu.

Do komunikacji z układem wykorzystano platformę PYNQ firmy Xilinx (AMD), która umożliwia proste pisanie kodu do komunikacji z układami ZYNQ. Pozwala programować urządzenie przez notatnik Jupyter przy użyciu języka Python. Platforma umożliwia możliwość reprogramowania układu poprzez dynamiczne wczytywanie stworzonych bitstreamów.

W projekcie wykorzystano magistralę AXI-Lite, która jest typu memory mapped, więc komunikujemy się przy użyciu adresów rejestrów przesuniętych o pewien offset od adresu bazowego.

Mimo responsywności stworzonych algorytmów, aby wygenerować bitstream należało wcześniej zdecydować się na rozmiar FFT. Dla wszystkich 3 algorytmów zdecydowano się ustawić rozmiar FFT na 8. W algorytmach wykorzystujących maszynę stanów, dane wprowadzane są sekwencyjnie jedna po drugiej, więc przy tworzeniu IP Cora wykorzystano 2 rejestry danych wejściowych (część rzeczywista i urojona), 2 rejestry danych wyjściowych oraz rejestry sygnalizacyjne. Natomiast dla trzeciej wersji projektu, która wymagała wprowadzenia jednocześnie wszystkich danych, wykorzystano 16 rejestrów danych wejściowych (8 liczb część rzeczywista oraz 8 liczb część urojona) oraz 16 rejestrów danych wyjściowych i rejestry sygnalizacyjne. Dla algorytmów 1 i 2 istnieje możliwość wygenerowania kolejnego bitstreamu z ustawionym większym rozmiarem FFT. Dla algorytmu 3 uzyskanie wersji dla większego rozmiaru FFT jest bardziej skomplikowane i zmusza do wykorzystania większej ilości rejestrów.

W notatniku Jupyter zaimplementowano stworzoną wcześniej funkcję wzorcową w Pythonie, a następnie utworzony został generator losowych próbek. W dalszej części notatnika tworzono overlay dla każdej wersji algorytmu oraz obliczano FFT dla tych samych danych wejściowych i porównywano z wynikiem funkcji wzorcowej. Błędy obliczeniowe dla każdej z próbek w danej implementacji algorytmu przedstawiane są na wykresie. Przykładowy wykres znajduje się na Rys. 6.



Rys. 6: Wykres błędu dla każdej z obliczonych próbek.

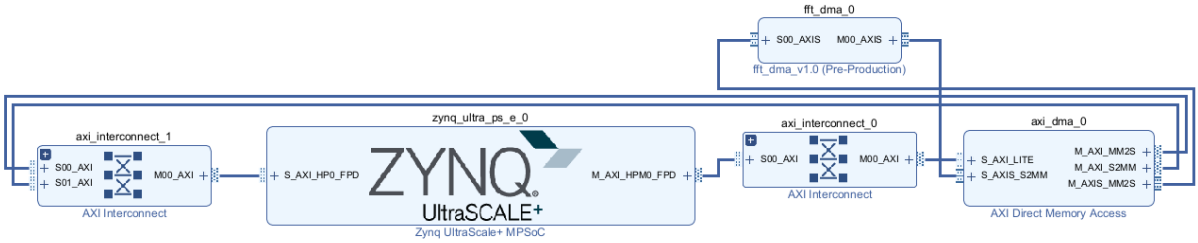
Z wykresu widać, że implementacja na układach FPGA jest dość dokładna względem funkcji wzorcowej. Dodatkowo obliczana została średnia kwadratowa błędu RMSE (ang. Root Mean Square Error) wyrażana wzorem:

$$RMSE = \sqrt{\frac{(X_1 - Z_1)^2 + \cdots + (X_N - Z_N)^2}{N}}$$

gdzie:
- X – obliczona wartość próbki przez układ FPGA,
- Z – obliczona wartość próbki przez funkcje wzorcową,
- N – rozmiar FFT.

W ramach porównania trzech algorytmów sprawdzono czas wykonania algorytmów. Jednak wynik nie był miarodajny, ponieważ czas obliczeń był dużo krótszy niż czas sprawdzania rejestru *ready*, toteż zrezygnowano z tego pomiaru.

Dla ostatniej wersji projektu wykorzystującej potokowość wykorzystano magistralę AXI Stream oraz układ DMA. Schemat blokowy przedstawiono na Rys. 7.

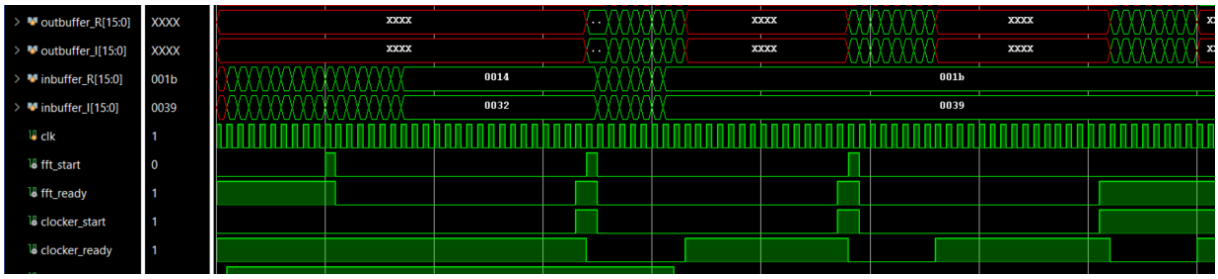


Rys. 7: Schemat blokowy potokowe FFT.

Z racji, że dane nie były pobierane z DMA z każdym cyklem zegara należało odpowiednio ustawiać sygnały tvalid i tready na magistrali AXIS. W celu poprawnej implementacji należało bardzo dokładnie przeanalizować dokumentację producenta w celu zrozumienia zasady działania wszystkich sygnałów sterujących tvalid, tready i tlast. Złe ustawianie/kasowanie wymienionych sygnałów spowodowało wiele komplikacji przy uruchomianiu projektu na sprzęcie, pomimo że wyniki symulacji w testbenchu były poprawne.

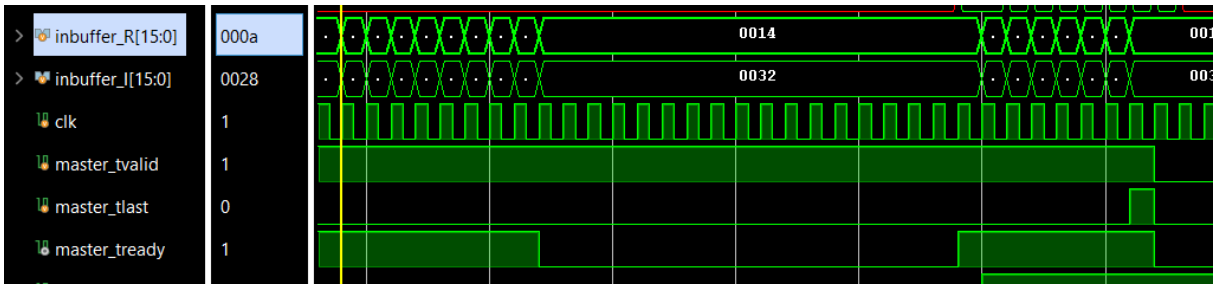
Sygnał tvalid wystawia nadawca i informuje odbiorcę, że dana jest gotowa do odbioru. Sygnał tready wystawia odbiorca i informuje, że jest gotów do odbioru danych. Sygnał tlast ustawiony jest w stan wysoki tylko podczas wysłania ostatniej danej i informuje odbiorcę o końcu transmisji. Przesył danych odbywa się tylko na zboczu narastającym zegara w momencie, gdy sygnał tvalid i tready jest w stanie wysokim.

Na Rys. 8 przedstawiono przebieg pokazujący potokowe działanie układu. Na przebiegu widać, że po odebraniu pierwszych 8 danych wejściowych ustawiany jest sygnał *fft_start* i jeśli sygnał *fft_ready* również jest w stanie wysokim moduł odbierający dane przekazuje je do układu obliczającego kasując przy tym sygnał *fft_start* i *fft_ready*. Następnie pobiera kolejne próbki i oczekuje na sygnał *fft_ready* oznaczający, że FFT zostało obliczone i można przekazać dane. W momencie, gdy układ obliczający skończy pracę przekazuje sygnał *fft_ready* jako sygnał *clocker_start* oznaczający, że moduł wysyłający wynik może pobrać dane do wysyłki. Po skończonej wysyłce układ clocking out wystawia sygnał *clocker_ready* oznaczający gotowość do pobrania nowych danych do wysyłki. Z przebiegu widać działanie potokowe układu. W momencie wysłania wyniku dla pierwszego zestawu danych pobierane są dane dla trzeciego zestawu, a drugi zestaw jest w tym czasie obliczany.



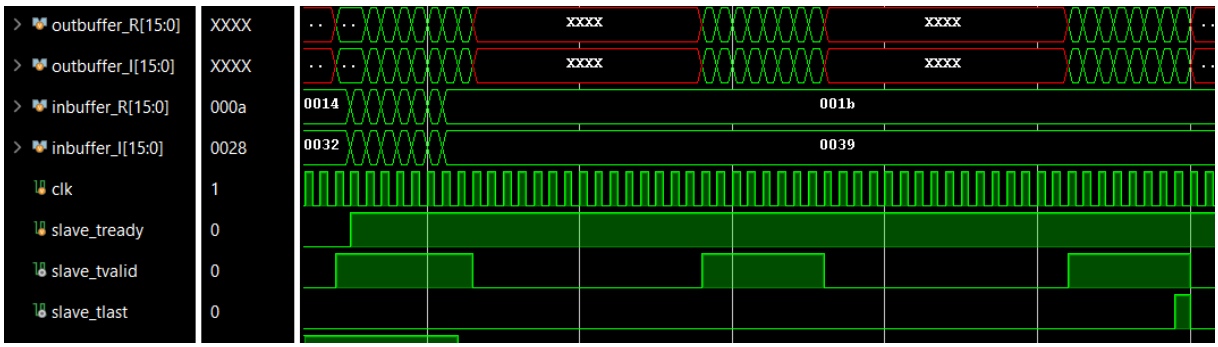
Rys. 8: Przebieg obrazujący potokowe działanie układu.

Na Rys. 9 przedstawiono sposób symulacji sygnałów sterujących AXIS przy odbiorze danych przez układ FFT. Sygnał *master_tvalid* wystawiany przez DMA sygnalizuje, że ma gotowe dane do wysłania, więc ustawiony jest do momentu wysłania ostatniej liczby znajdującej się w buforze. Następnie sterujemy sygnałem *master_tready* w module FFT, aby pobierać tylko 8 liczb, a później zatrzymać transmisję do momentu, kiedy zebrane dane przekazane zostaną do układu obliczającego. Po wysłaniu ostatniej liczby DMA wystawia sygnał *master_tlast*.



Rys. 9: Symulacja sygnałów sterujących (odbieranie danych)

Sytuację odwrotną, czyli wysyłanie danych z modułu FFT do DMA przedstawiono na Rys. 10. Tym razem sterujemy sygnałem *slave_tvalid*, który oznacza, że układ FFT ma dane do wysłania. Sygnał *slave_tready* dla układu DMA ustawiony jest cały czas po pierwszym wykryciu sygnału *slave_tvalid*. Podczas wysyłania ostatniej liczb dla ostatniego zestawu danych, układ FFT ustawia w stan wysoki sygnał *slave_tlast* sygnalizujący, że wysłano już wszystkie dane. Warto wspomnieć, że układ FFT wie, kiedy może wysłać sygnał *slave_tlast*, ponieważ podczas odbierania danych zapamiętywana jest liczba odebranych danych wejściowych, więc analogicznie tyle samo danych jest wysyłanych do DMA.



Rys. 10: Symulacja sygnałów sterujących (wysyłanie wyniku)

Stworzony układ został pozytywnie przetestowany na rzeczywistym sprzęcie. W tym celu w notatniku stworzono funkcję generującą zestaw losowych danych wejściowych. Liczba generowanych zestawów określona jest przez parametr *N_packs*, który można modyfikować. Umożliwia to wysłanie do układu FFT kilku paczek danych w celu wykorzystania zaimplementowanej potokowości. Wygenerowane dane trafiają do wcześniej alokowanego buforu

wejściowego w odpowiednim formacie. Bufor składa się z liczb 32-bitowych. 16 starszych bitów każdej z liczb reprezentuje część urojoną próbki, natomiast 16 bitów młodszych reprezentuje część rzeczywistą. Przypisanie danych przedstawiono na Rys. 11.

```
in_buffer = allocate(shape=(N_packs*N_pipeline,), dtype=np.uint32)
out_buffer = allocate(shape=(N_packs*N_pipeline,), dtype=np.uint32)
array1_uint32 = data_real_pipeline.astype(np.uint32)
array2_uint32 = data_imag_pipeline.astype(np.uint32)

for i in range(N_packs*N_pipeline):
    in_buffer[i] =(array2_uint32[i] << 16) | array1_uint32[i]
print(in_buffer)
```

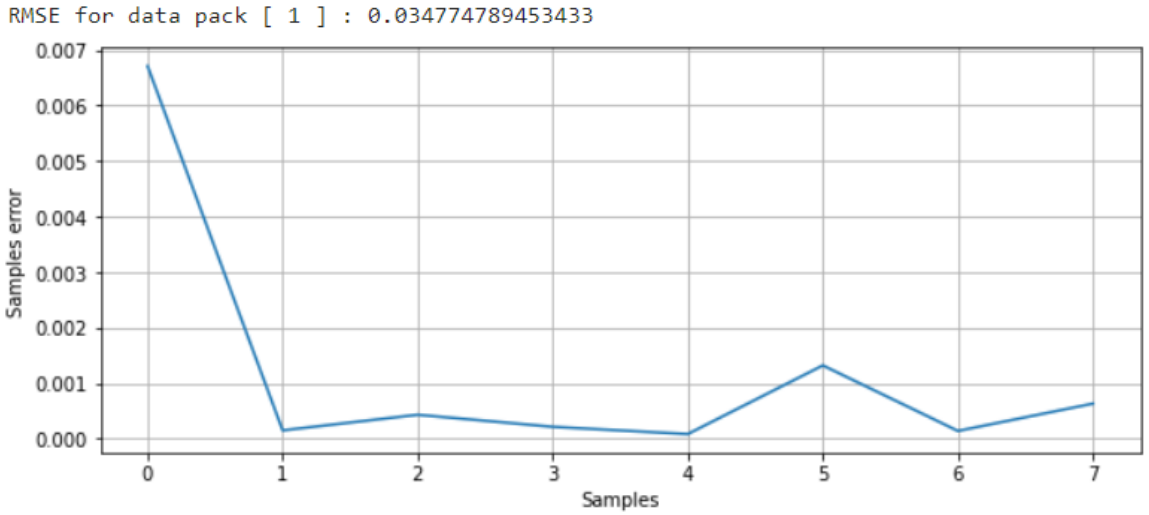
Rys. 11: Przypisanie danych do bufora wejściowego.

Następnie gotowy bufor wejściowy wysyłany jest do DMA, a później wynik odbierany w buforze wyjściowym (Rys. 12).

```
fft_pipe.sendchannel.transfer(in_buffer)
fft_pipe.recvchannel.transfer(out_buffer)
```

Rys. 12: Komunikacja z DMA.

Końcowym etapem jest rozdzielenie wyniku na kilka zestawów w zależności od wcześniej ustawionego parametru *N_packs* i porównanie z wynikami funkcji wzorcowej. Wynik przedstawiono w podobny sposób, jak dla poprzednich algorytmów (Rys. 13).



Rys. 13: Przedstawienie wyniku.

Dla każdego z zestawów generowany jest osobny wykres oraz obliczany RMSE. W przypadku tego algorytmu można zauważyć wzrost błędu między próbkami niż w przypadku pozostałych wersji algorytmów. Spowodowane jest do zmniejszeniem liczby bitów reprezentujących cześć ułamkową z 10bitów do 6 bitów. Zdecydowano się na redukcję, aby dopasować rozmiar wysyłanych danych do DMA.

6. Bibliografia

[1] A DFT and FFT TUTORIAL, AlwaysLearn.com, https://www.alwayslearn.com/DFT%20and%20FFT%20Tutorial/DFTandFFT_FFT_Butterfly_8_Input.html

Fast Fourier Transform, Wikipedia, https://en.wikipedia.org/wiki/Fast_Fourier_transform

Github implementacja, https://github.com/MaciejZielinski407530/sdup_fft