

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji

**Algorytm Barrel Wheel Transform**

Systemy Dedykowane w układach programowalnych

*Autorzy:* Paweł Frączkiewicz, Piotr Duszkiewicz

*Kierunek:* Elektronika

Kraków, 2023

1. Wstęp

Celem projektu było zaimplementowanie algorytmu Barrel Wheel Transform na platformie FPGA (ang. Field Programmable Gate Array). Algorytm ten jest często stosowany w kompresji danych i analizie sekwencji tekstowych. Transformata polega na konwersji wejściowego wyrazu i składa się z trzech kroków: rotacji, sortowaniu oraz ekstrakcji końcowych liter tworząc nowy wyraz. BWT wykorzystywana jest między innymi w kompresji, ponieważ ułatwia skompresowanie ciągu powtarzających się znaków redukując rozmiar danych. Dodatkowo wiele algorytmów kompresji, takich jak bzip2, wykorzystują transformatę BWT do przygotowania danych przed zastosowaniem innych technik kompresji. BWT tworzy sekwencje znaków o dużej skupialności, które są bardziej podatne na skuteczną kompresję. Transformatę BWT można również zastosować w procesie szyfrowania danych. Poprzez odpowiednie przekształcenie tekstu za pomocą BWT i zastosowanie algorytmów szyfrowania, można uzyskać zaszyfrowane dane, które są trudniejsze do odczytania bez odpowiedniego klucza.

Urządzeniem, na którym dokonano implementacji algorytmu jest płyta ewaluacyjna Zedboard Zynq-7000. Posiada ona dwurdzeniowy procesor ARM Cortex A9, 512MB pamięci DDR3 oraz 256MB pamięci flash QSPI. Płyta ewaluacyjną może zostać zastosowana do przetwarzania wideo, obliczeń rekonfigurowalnych czy przyspieszania oprogramowania.

Projekt transformaty wykonano w środowisku Vivado w wersji 2018.3 oraz Visual Code Studio w celu utworzenia aplikacji prezentującej logikę działania BWT w języku Python w wersji 3.11.

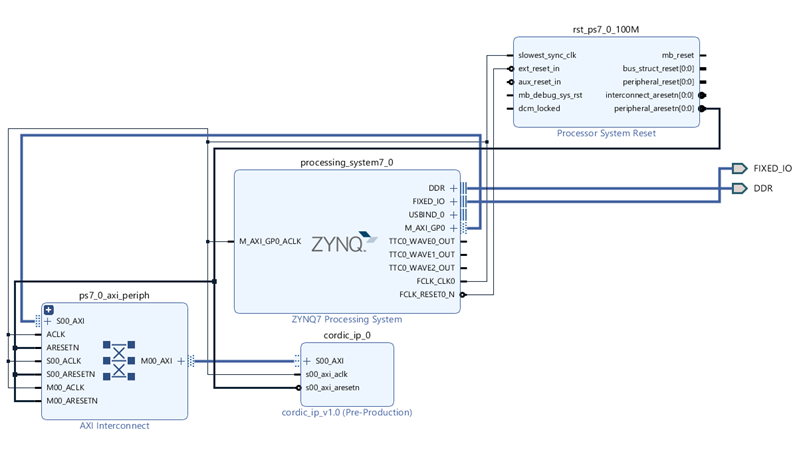
2. Struktura projektu

Projekt znajduje się na repozytorium github:

https://github.com/fraczpa22/SDUP\_BWT\_transformata

Struktura repozytorium wygląda następująco:

* BWT – zawiera dwa modele behawioralne implementujące transformatę. Pierwszy model wykorzystuje strukturę sekwencyjnej maszyny stanów symulując działanie klasycznego mikroprocesora, gdzie wszystkie operacje następują jedna po drugiej. Natomiast drugi model wykorzystuje strukturę potokową (ang. pipeline) prezentując zalety wykorzystania układów FPGA jako akceleratorów w wielu dziedzinach.
* BWT\_cordic – zawiera diagram blokowy integrujący płytę deweloperską Zedboard z utworzonym przez nas akceleratorem (cordic\_ip\_0). W celu połączeniu wyżej wymienionych bloków, wykorzystana została magistrala AXI.



* Ip\_repo – zawiera plik ip, stosowany w diagramie blokowym powyżej. W swojej strukturze zapisany został akcelerator transformaty BWT
* BWT.py – plik prezentujący działanie transformaty w języku Python

3. Opis działania algorytmu

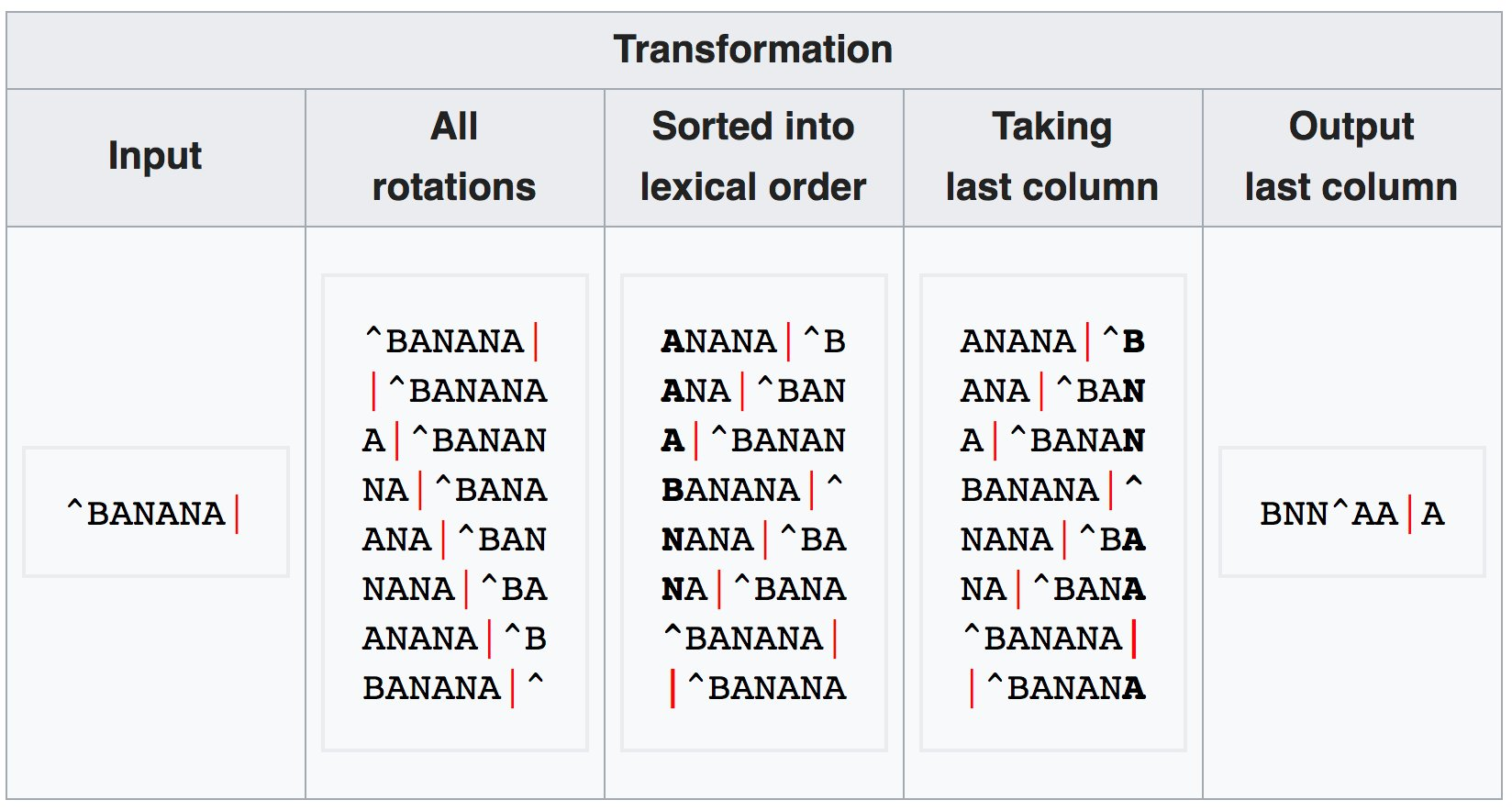
Tranformata BWT składa się z trzech poniższych kroków:

a) rotacja - należy utworzyć wszystkie cykliczne rotacje danego wyrazu wejściowego

b) sortowanie - następnym krokiem jest posortowanie rotacji leksykograficznie

c) ekstrakcja - polega na wydobyciu ostatniej litery ze wszystkich wyrazów znajdujących się w utworzonej kolumnie

Poniższy rysunek obrazuję zastosowanie transformaty BWT na przykładowym wyrazie:



4. Implementacja algorytmu w środowisku Python

W tym podrozdziale zostanie przybliżone działanie algorytmu opracowane w środowisku Python.

Kod został napisany w taki sposób, aby wymusić jak najmniejsze modyfikacje podczas przenoszenia do środowiska Vivado. Wiąże się to również z tym, że nie wykorzystujemy żadnych zewnętrznych bibliotek.

Opis kodu:

1.Na początku użytkownik zostaje poproszony o wprowadzenie wyrazu, który ma zostać poddany transformacji BWT.

2.Następnie, dla podanego wyrazu, generowane są wszystkie możliwe rotacje tego wyrazu. Każda rotacja to przesunięcie znaków wyrazu o jedno miejsce w lewo.

3.Wygenerowane rotacje są przechowywane w liście new\_word.

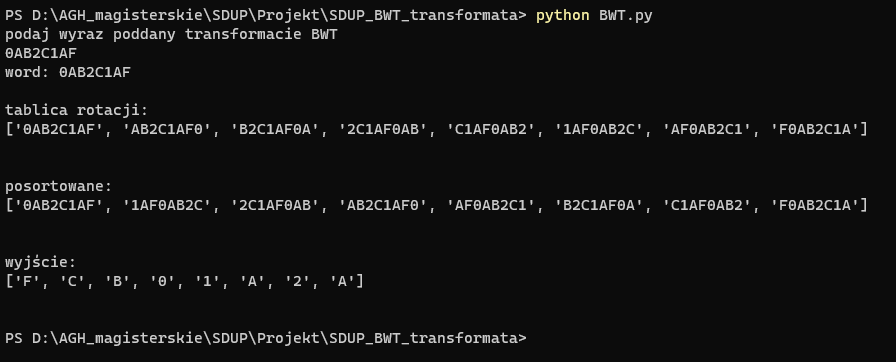
4.Kolejnym krokiem jest sortowanie rotacji w liście new\_word. Sortowanie odbywa się zgodnie z regułami algorytmu BWT. Jeśli dwa wyrazy mają ten sam pierwszy znak, porównuje się kolejne znaki aż do znalezienia pierwszego różniącego się znaku. Jeśli pierwszy znak w pierwszej rotacji jest większy od pierwszego znaku w drugiej rotacji, następuje zamiana miejscami tych rotacji. Jeśli pierwszy znak w pierwszej rotacji jest mniejszy od pierwszego znaku w drugiej rotacji, to nie ma zamiany miejscami.

5.Po posortowaniu, lista new\_word zawiera rotacje w odpowiedniej kolejności.

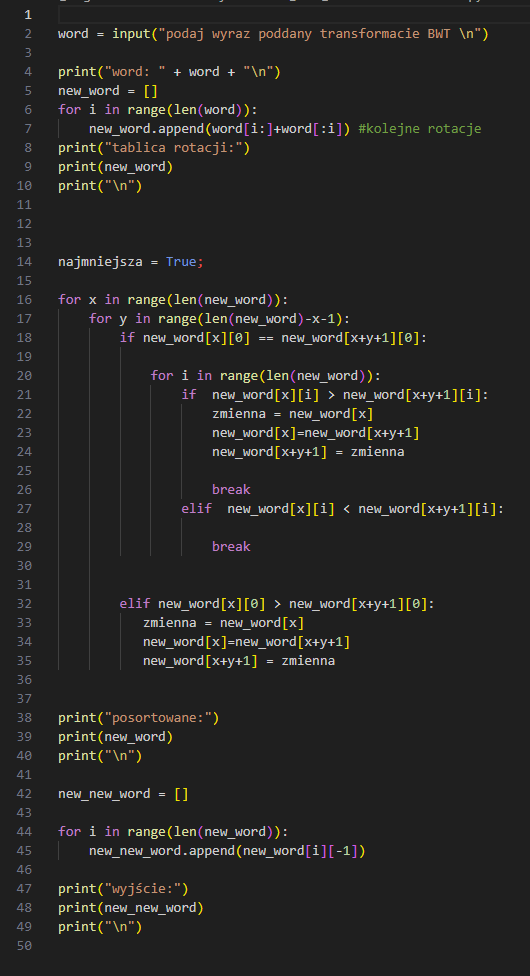
6.Następnie, tworzona jest lista new\_new\_word, która przechowuje ostatnie znaki poszczególnych rotacji (znaki ostatniej kolumny w macierzy rotacji).

7.Na koniec, lista new\_new\_word jest wyświetlana jako wynik transformacji BWT.

Na poniższym zrzucie widoczny jest działanie przedstawionego kodu:



Opisany kod znajduje się poniżej:



5. Podstawowa wersja algorytmu i jego symulacje

Kolejnym etapem projektu było zaimplementowanie algorytmu w środowisku Vivado. Pierwsza wersja algorytmu miała proste działanie - tj. nie korzysta z pipeline. Poniżej przedstawiono opis działania poszczególnych stanów modułu “BWT\_transform” w którym to zaimplementowany został algorytm BWT.

Opis działania kodu:

Moduł BWT\_transform przyjmuje sygnały wejściowe clk, rst, start oraz dane wejściowe data\_in.

Parametr dl\_wyraz określa szerokość wyrazu (w liczbie bitów) i jest bezpośrednio zależny od szerokości data\_in.

Parametr szer\_litery określa szerokość pojedynczej litery w wyrazie (w liczbie bitów).

Parametr ilosc\_liter oblicza ilość liter w wyrazie na podstawie dl\_wyraz i szer\_litery.

Sygnał wyjściowy data\_out przechowuje dane wyjściowe po transformacji BWT.

Sygnał wyjściowy done jest flagą oznaczającą zakończenie obliczeń.

Zmienna lokalna buffor jest tablicą, na której odbywają się wszystkie operacje rotacji i sortowania.

Zmienna lokalna data\_var przechowuje wartość bazową podczas wymiany indeksów podczas sortowania.

Zmienna lokalna dana\_wyj przechowuje ostatnie bity, tworząc tym samym wyraz wyjściowy.

Moduł zawiera parametry S1, S2, ..., S8, które określają kolejne stany automatu.

Zmienne i, x, y i z są licznikami, które kontrolują przebieg procesu transformacji.

Parametry wejściowe:

clk - Sygnał zegara.

rst - Sygnał resetu.

start - Sygnał rozpoczęcia obliczeń.

data\_in - Dane wejściowe, reprezentujące wyraz poddawany transformacji BWT.

Zmienne i sygnały wyjściowe:

data\_out - Dane wyjściowe po transformacji BWT.

done - Flaga oznaczająca zakończenie obliczeń.

Opis działania kodu:

Po wystąpieniu zbocza narastającego sygnału zegara (posedge clk), w zależności od stanu rst (resetu), następuje przypisanie wartości początkowych dla zmiennych.

Wykorzystując konstrukcję case, następuje przejście przez kolejne stany automatu.

Stan S1 oczekuje na aktywację sygnału start. Po jego aktywacji, przechodzi do stanu S2.

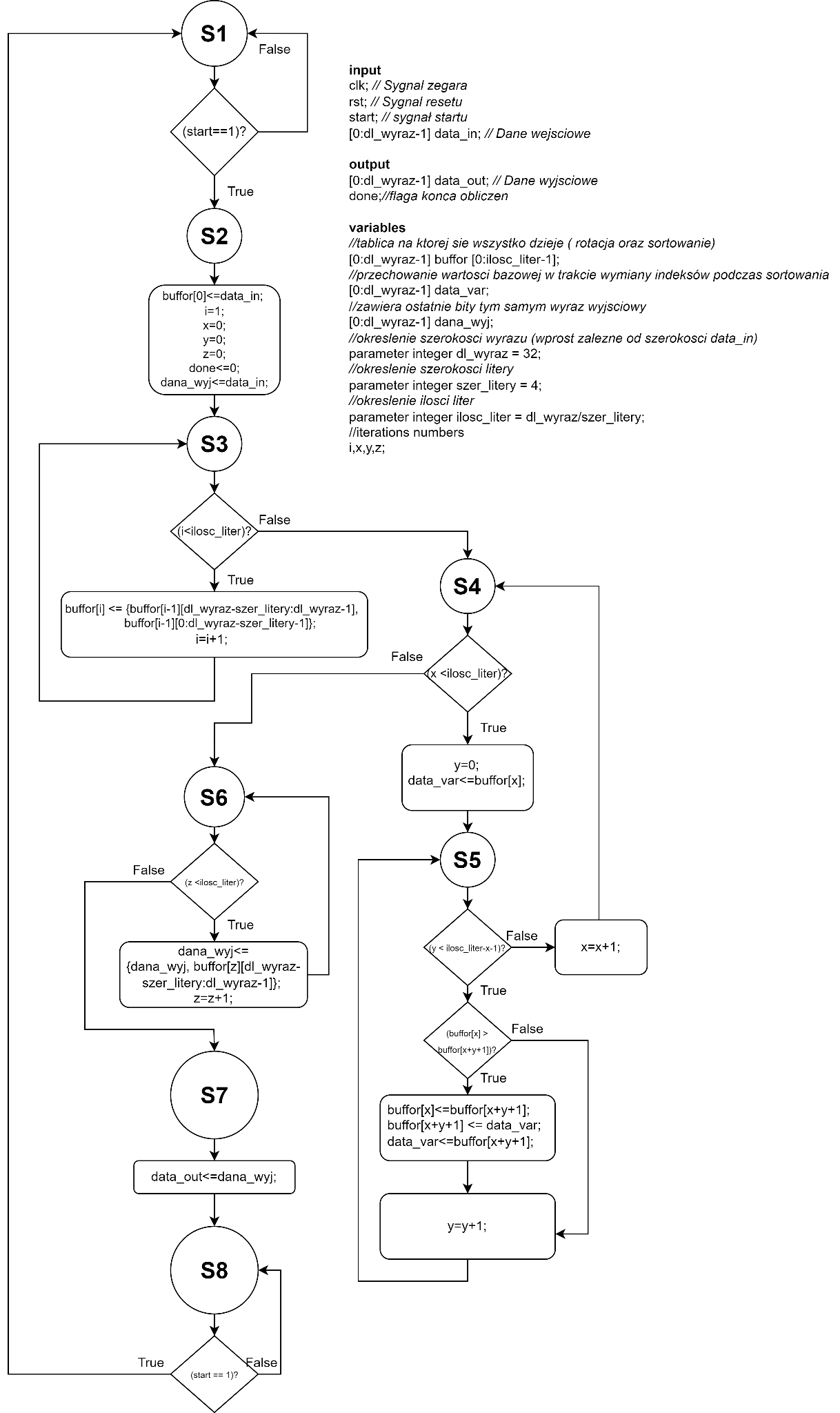
Stan S2 inicjalizuje zmienną buffor[0] wartością data\_in i ustala wartości początkowe dla zmiennych licznikowych.

Następnie następuje przejście przez kolejne stany S3, S4, S5 i S6, w których wykonuje się generowanie rotacji i sortowanie w tablicy buffor.

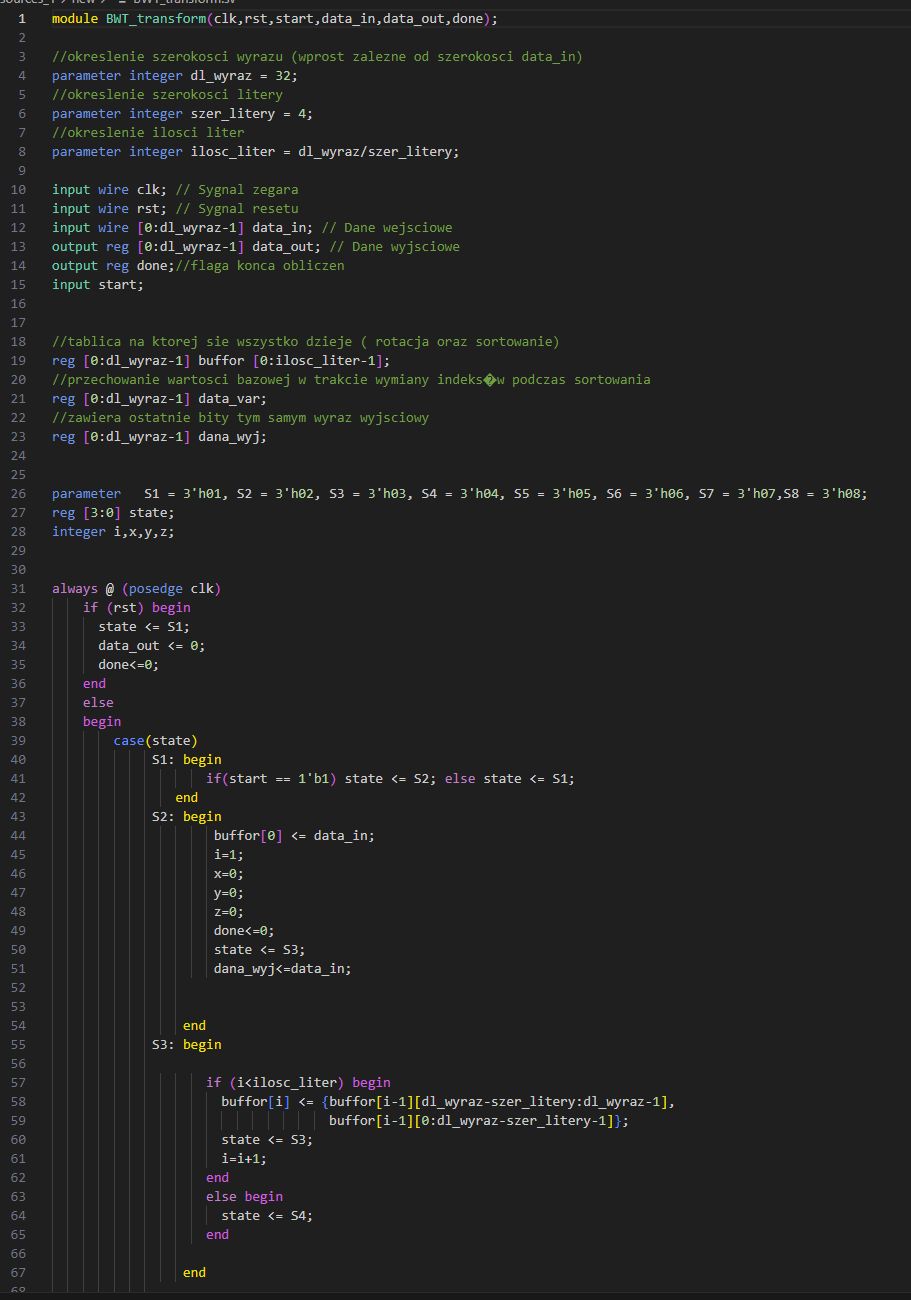
W stanie S7 ustawiana jest flaga done, przypisywane są dane wyjściowe do data\_out i przechodzimy do stanu S8.

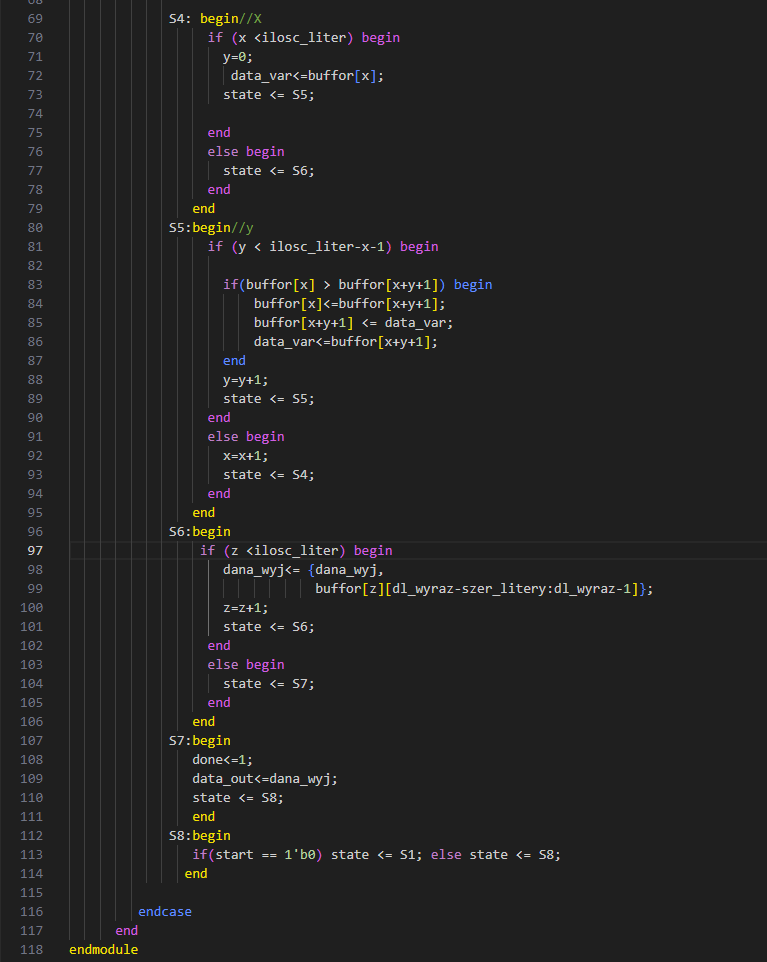
W stanie S8 oczekujemy na wyłączenie sygnału start. Po wyłączeniu, przechodzimy ponownie do stanu S1.

Graficzne przedstawienie działania algorytmu:

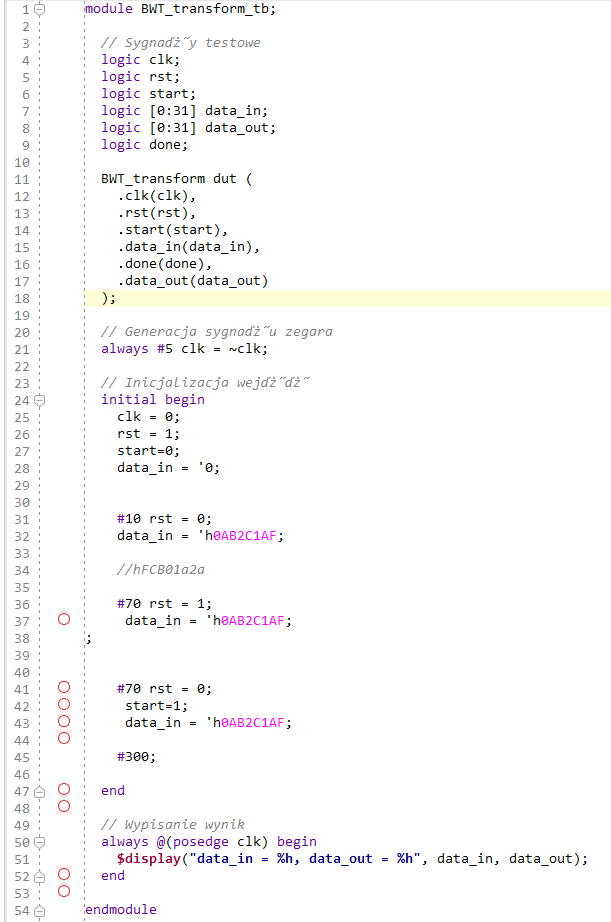


Opisany kod znajduje się poniżej:

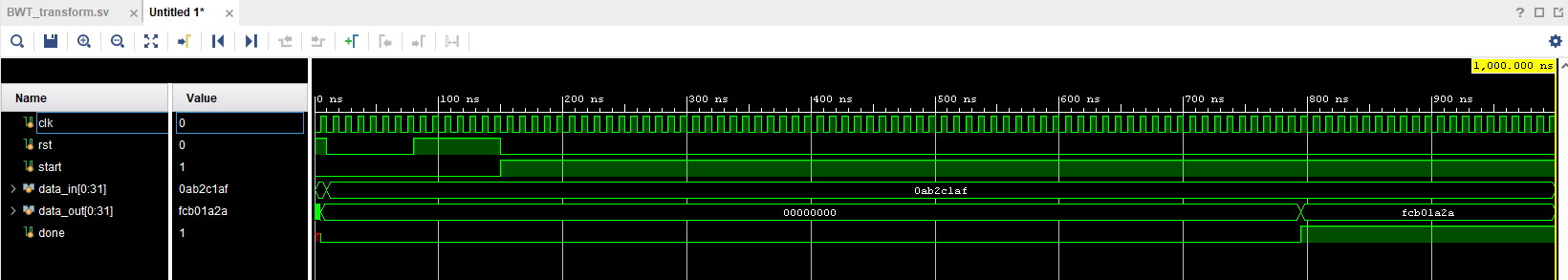




Kolejnym etapem było stworzenie testbench’u w celu weryfikacji poprawnoscia działania modułu. Dane wejściowe symulacji zostały podane takie same jak w przypadku implementacji w środowisku Python w celu możliwie łatwego zweryfikowania i zestawienia wyników.



Na poniższym zrzucie widać przebieg symulacji oraz otrzymaną wartość wyjściową. Jak można zaobserwować wynik wyjściowy pokrywa się z wynikiem uzyskanym w przypadku implementacji w środowisku Python.



6. Dodanie potoku do algorytmu.

W celu zrelizowania potoku utworzone zostały dodatkowe dwa moduły “BWT\_last\_letter “ oraz “BWT\_step”.

Opis działania modułu “BWT\_step”::

1. Moduł BWT\_step przyjmuje sygnały wejściowe ‘clk’, ‘star’t oraz dane wejściowe ‘wejscie’.
2. Parametr ‘moved’ określa liczbę przesunięć, czyli ile liter jest przesuwanych w jednym kroku transformacji BWT.
3. Parametr ‘dl\_wyraz’ określa szerokość wyrazu (w liczbie bitów) i jest bezpośrednio zależny od szerokości ‘wejscie’.
4. Parametr ‘szer\_litery’ określa szerokość pojedynczej litery w wyrazie (w liczbie bitów).
5. Sygnał wyjściowy ‘wyjscie’ przechowuje dane wyjściowe po wykonaniu jednego kroku transformacji BWT.
6. W bloku ‘always @(posedge clk)’ następuje reakcja na zbocze narastające sygnału zegara.
7. Jeśli sygnał ‘start’ jest aktywny (‘start == 1'b1’), wykonuje się przesunięcie liter w wyrazie zgodnie z wartością parametru ’moved’.
8. Do sygnału ‘wyjscie’ przypisywane są przesunięte wartości z wyrazu wejscie.
9. W przeciwnym razie, gdy sygnał start jest nieaktywny, sygnał ‘wyjscie’ jest wyzerowany (wyjscie <= 0).

Opis działania modułu “BWT\_last\_letter”:

1. Moduł “BWT\_last\_letter” przyjmuje sygnały wejściowe clk, start oraz dane wejściowe ‘wejscie’.
2. Parametr ‘moved’ jest bezużyteczny w tym module, ponieważ określa liczbę przesunięć, które nie są stosowane w ostatnim kroku transformacji.
3. Parametr ‘dl\_wyraz’ określa szerokość wyrazu (w liczbie bitów) i jest bezpośrednio zależny od szerokości ‘wejscie’.
4. Parametr ‘szer\_litery’ określa szerokość pojedynczej litery w wyrazie (w liczbie bitów).
5. Sygnał wyjściowy ‘wyjscie’ przechowuje dane wyjściowe po wykonaniu ostatniego kroku transformacji BWT.
6. W bloku ’always @(posedge clk)’ następuje reakcja na zbocze narastające sygnału zegara.
7. Jeśli sygnał start jest aktywny (‘start == 1'b1’), wartość sygnału ‘wejscie’ przypisywana jest bezpośrednio do sygnału ‘wyjscie’.
8. Do sygnału ‘wyjscie’ przypisywane są tylko ostatnie ‘szer\_litery’ bitów z wyrazu ‘wejscie’.
9. W przeciwnym razie, gdy sygnał start jest nieaktywny, sygnał wyjscie jest wyzerowany (wyjscie <= 0).

Moduł `BWT\_transform\_with\_Pipeline` jest implementacją transformacji BWT (Burrows-Wheeler Transform) w języku Verilog z zastosowaniem pipelinowania dla lepszej wydajności. Poniżej przedstawiam opis działania kodu oraz parametry wejściowe i wyjściowe:

Parametry wejściowe:

- `clk` - sygnał zegara

- `rst` - sygnał resetu

- `start` - sygnał rozpoczęcia transformacji BWT

- `data\_in` - dane wejściowe

Parametry wyjściowe:

- `data\_out` - dane wyjściowe po wykonaniu transformacji BWT

- `done` - flaga końca obliczeń

Zmienne:

- `state` - zmienna reprezentująca aktualny stan procesu transformacji BWT.

- `x` - zmienna pomocnicza używana w pętli do iteracji po literach wyrazu.

- `y` - zmienna pomocnicza używana w pętli do iteracji po literach wyrazu.

- `zacznij` - zmienna sterująca, wskazująca, czy należy rozpocząć przepisywanie wartości z `buffor\_rotacji` do `buffor` (w przypadku pierwszej iteracji).

- `scal` - zmienna pomocnicza używana do skalowania indeksów podczas przypisywania danych wyjściowych.

Parametry:

- `dl\_wyraz` - parametr określający szerokość wyrazu (liczba bitów) oraz liczność tablic `data\_in`, `data\_out` i `buffor`.

- `szer\_litery` - parametr określający szerokość pojedynczej litery w wyrazie (liczba bitów).

- `ilosc\_liter` - parametr określający liczbę liter w wyrazie, czyli ilość przesunięć, które zostaną wykonane w ramach transformacji BWT.

- `S1`, `S2`, `S3`, `S4`, `S5`, `S6` - parametry reprezentujące stany procesu transformacji BWT.

Parametry te umożliwiają konfigurację i dostosowanie działania modułu `BWT\_transform\_with\_Pipeline` do konkretnych wymagań aplikacji. Przykładowo, można zmienić szerokość wyrazu `dl\_wyraz` na inną wartość, taką jak 64 lub 128 bitów, dostosować szerokość pojedynczej litery `szer\_litery` do określonego formatu danych wejściowych oraz zmienić liczbę liter `ilosc\_liter` w zależności od oczekiwanej wydajności i ilości przesunięć, które chcemy wykonać podczas transformacji BWT.

Przez modyfikację tych zmiennych i parametrów można dostosować moduł `BWT\_transform\_with\_Pipeline` do różnych aplikacji, umożliwiając wykonywanie transformacji BWT na różnych typach danych wejściowych.

Opis działania kodu:

1. Moduł `BWT\_transform\_with\_Pipeline` przyjmuje sygnały wejściowe `clk`, `rst`, `start` oraz dane wejściowe `data\_in`.

2. Parametr `dl\_wyraz` określa szerokość wyrazu (w liczbie bitów) i jest bezpośrednio zależny od szerokości `data\_in`.

3. Parametr `szer\_litery` określa szerokość pojedynczej litery w wyrazie (w liczbie bitów).

4. Parametr `ilosc\_liter` określa ilość liter, czyli ilość przesunięć, które zostaną wykonane w ramach transformacji BWT.

5. Sygnał wyjściowy `data\_out` przechowuje dane wyjściowe po wykonaniu transformacji BWT.

6. Sygnał `done` jest flagą końca obliczeń.

7. Sygnał `start` inicjuje proces transformacji BWT.

8. Tablica `buffor` jest używana do przechowywania danych, na których odbywa się rotacja i sortowanie.

9. Tablica `buffor\_rotacji` przechowuje wynik rotacji z poprzedniego kroku transformacji BWT.

10. Sygnał `data\_var` przechowuje wartość bazową podczas wymiany indeksów podczas sortowania.

11. W bloku `generate` tworzone są instancje modułów `BWT\_step` i `BWT\_last\_letter` w celu zastosowania pipelinowania w transformacji BWT.

12. W bloku `always @(posedge clk)` następuje reakcja na zbocze narastające sygnału zegara.

13. Gdy sygnał `rst` (reset) jest aktywny, inicjalizowane są początkowe wartości zmiennych, a stan `state` ustawiany jest na `S1`.

14. Gdy sygnał `start` jest aktywny (`start == 1'b1`), następuje rozpoczęcie transformacji BWT.

15. W stanie `S2` następuje przypisanie danych wejściowych do tablicy `buffor[0]`, ustawienie zmiennych pomocniczych i flag, oraz przejście do stanu `S3`.

16. W stanach `S3` i `S4` następuje sortowanie danych w tablicy `buffor` za pomocą algorytmu Bubble Sort.

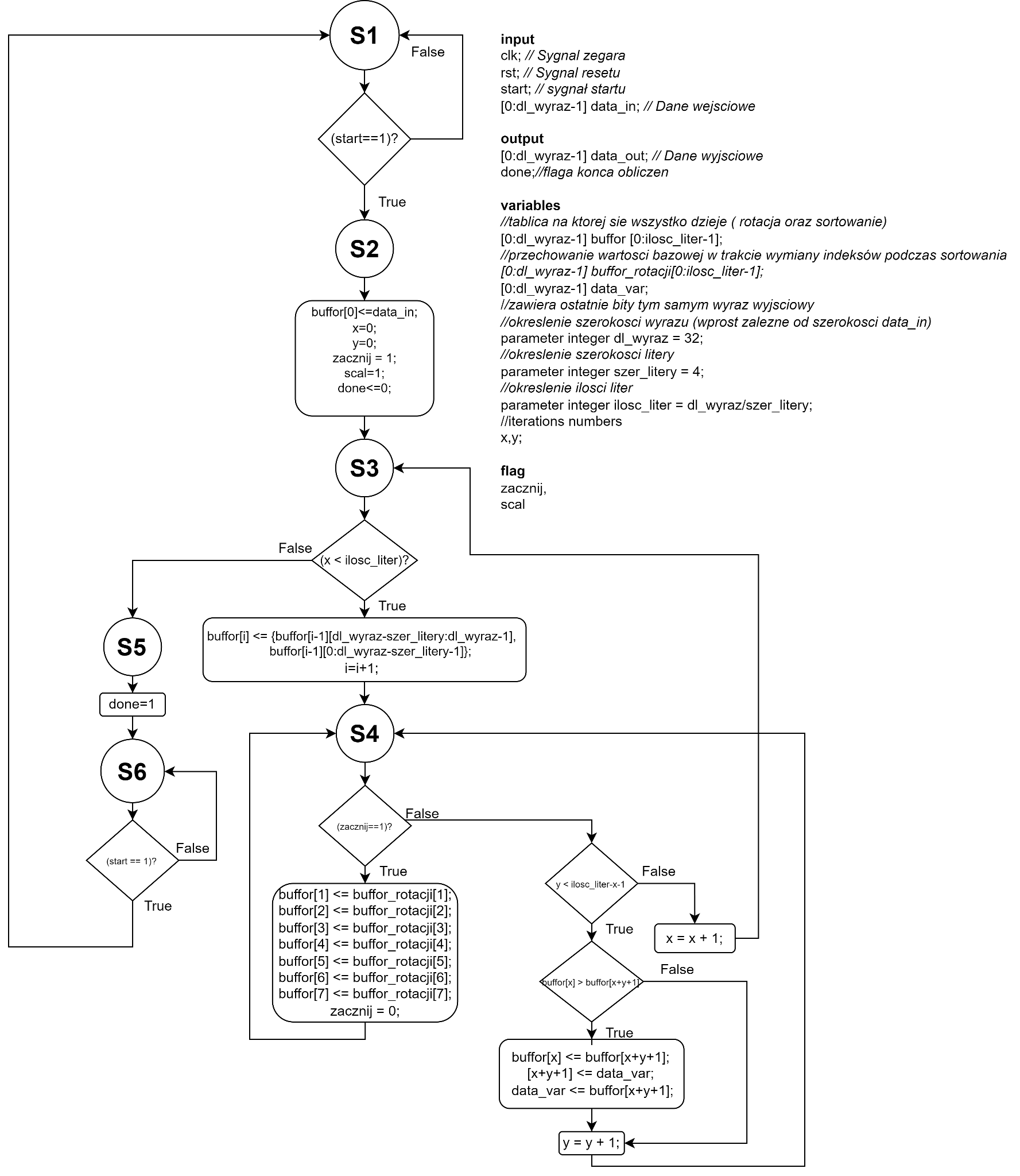
17. Po zakończeniu sortowania, w stanie `S5` ustawiana jest flaga `done` na wartość `1`, a dane wyjściowe są przechowywane w tablicy `data\_out`.

18. W stanie `S6` następuje sprawdzenie, czy sygnał `start

` jest aktywny. Jeśli nie, następuje powrót do stanu `S1`.

19. Proces transformacji BWT jest powtarzany, dopóki sygnał `start` jest aktywny.

Graficzne przedstawienie działania algorytmu:



Opis stanów:

S1: Stan początkowy. Oczekuje na sygnał start. Jeśli otrzyma sygnał start, przechodzi do stanu S2; w przeciwnym razie pozostaje w stanie S1.

S2: Inicjalizacja bufora zerowego, iteratorów i zmiennych pomocniczych. Ustawienie flag scal i zacznij. Przechodzi do stanu S3.

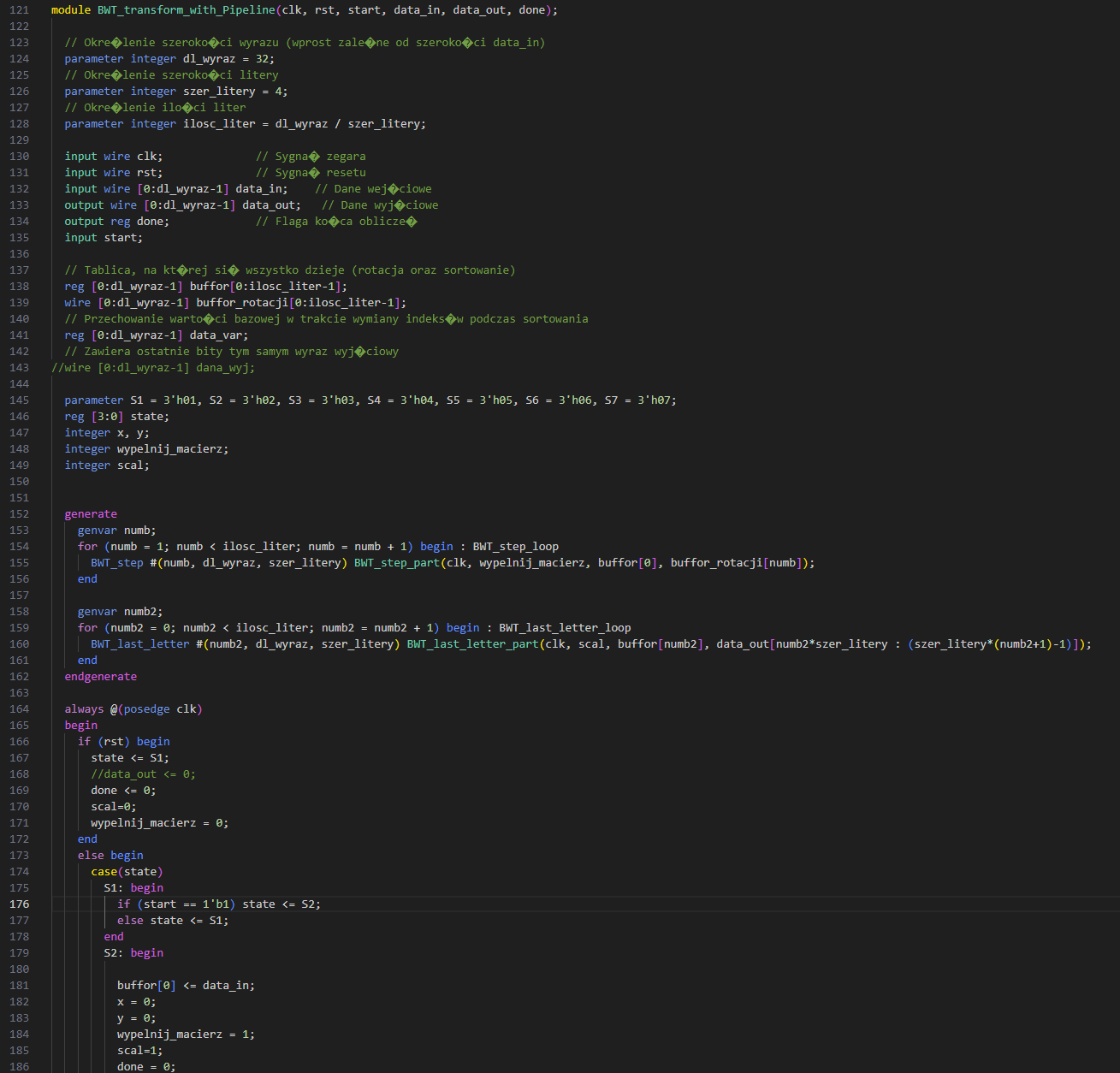
S3: Wykonywanie rotacji dla kolejnych liter w buforze. Przechodzi do stanu S4, gdy wszystkie rotacje są zakończone.

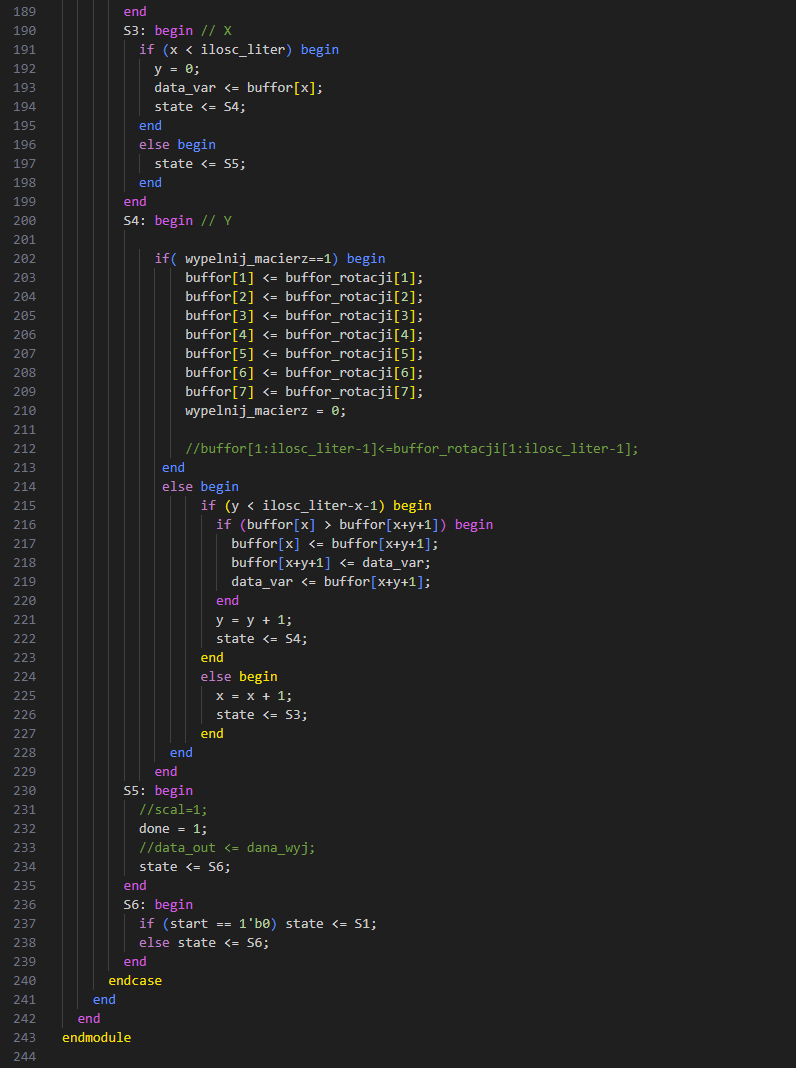
S4: Sortowanie bufora. Jeśli zmienna zacznij jest równa 1, przypisuje wartości bufor\_rotacji do odpowiednich buforów. W przeciwnym razie porównuje i wymienia elementy bufora. Przechodzi do stanu S3, aby kontynuować sortowanie.

S5: Końcowy stan sortowania. Ustawia flagę "done" na 1. Przechodzi do stanu S6.

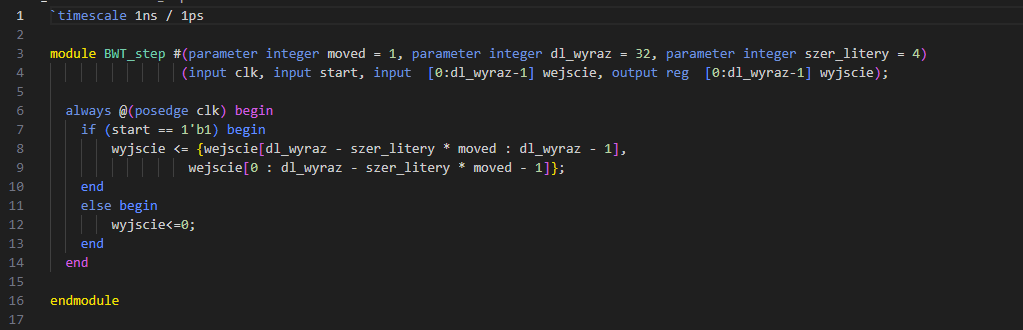
S6: Oczekiwanie na sygnał "start" równy 0. Jeśli otrzyma sygnał "start" równy 0, przechodzi do stanu S1; w przeciwnym razie pozostaje w stanie S6.

Opisany kod znajduje się poniżej:

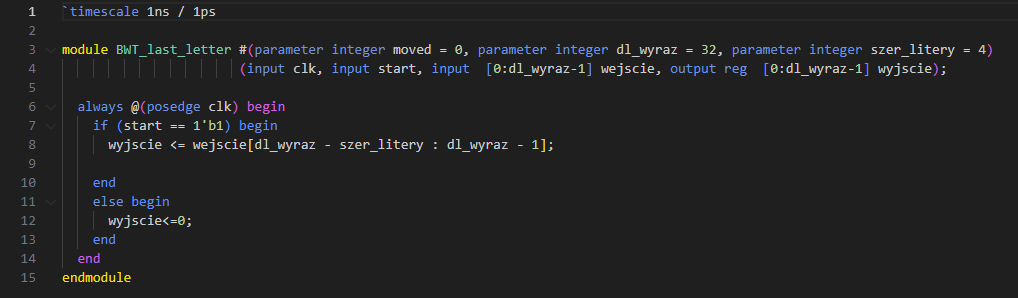




moduł do wszystkich rotacji



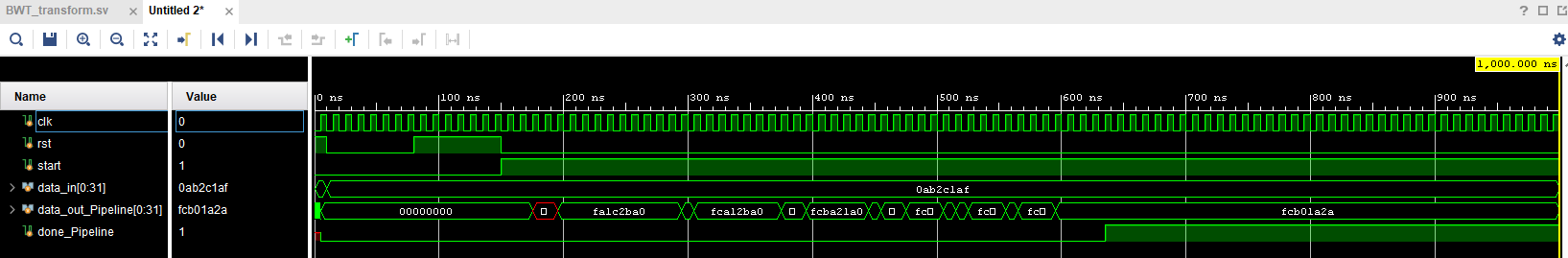
moduł do zebrania ostatniej litery w słowach



Poniżej przedstawiono strukturę projektu

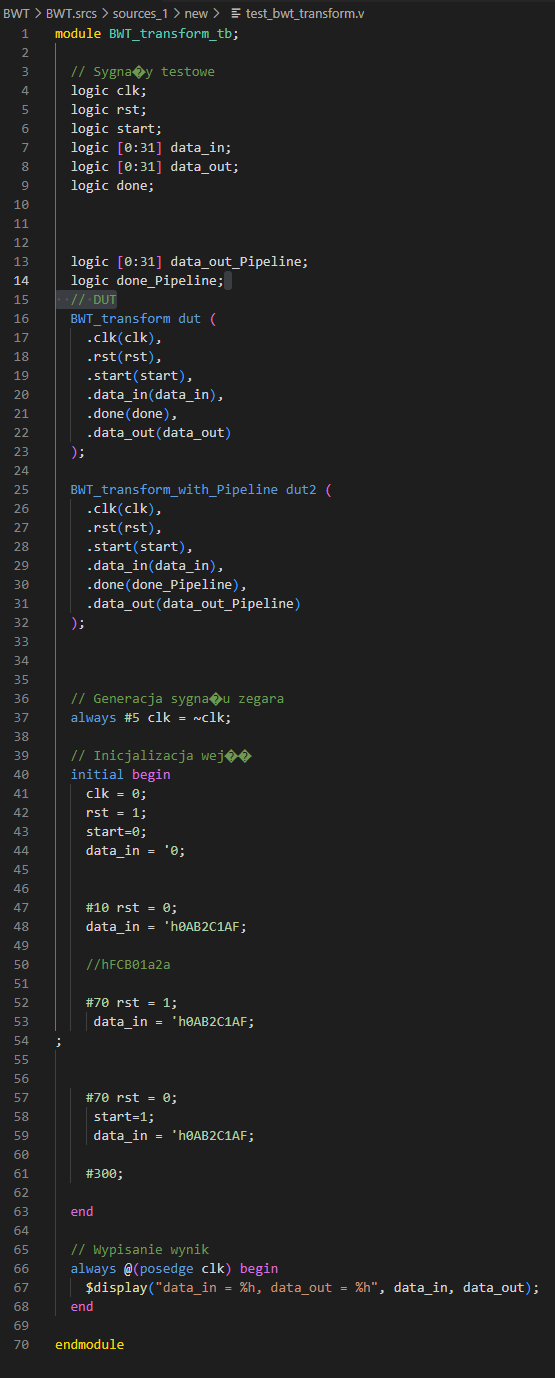


Na poniższym zrzucie widać przebieg symulacji oraz otrzymaną wartość wyjściową. Jak można zaobserwować wynik wyjściowy pokrywa się z wynikiem uzyskanym w przypadku implementacji w środowisku Python.

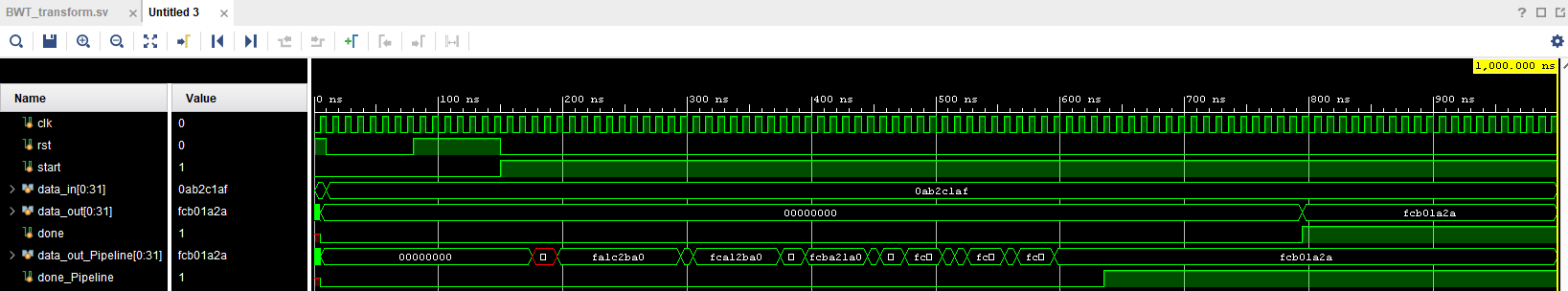


7. Zestawienie symulacji z wykorzystaniem potoku i bez.

W celu łatwego zestawienia symulacji zmodyfikowany został testbench tak aby obie wersje algorytmu działały równolegle.



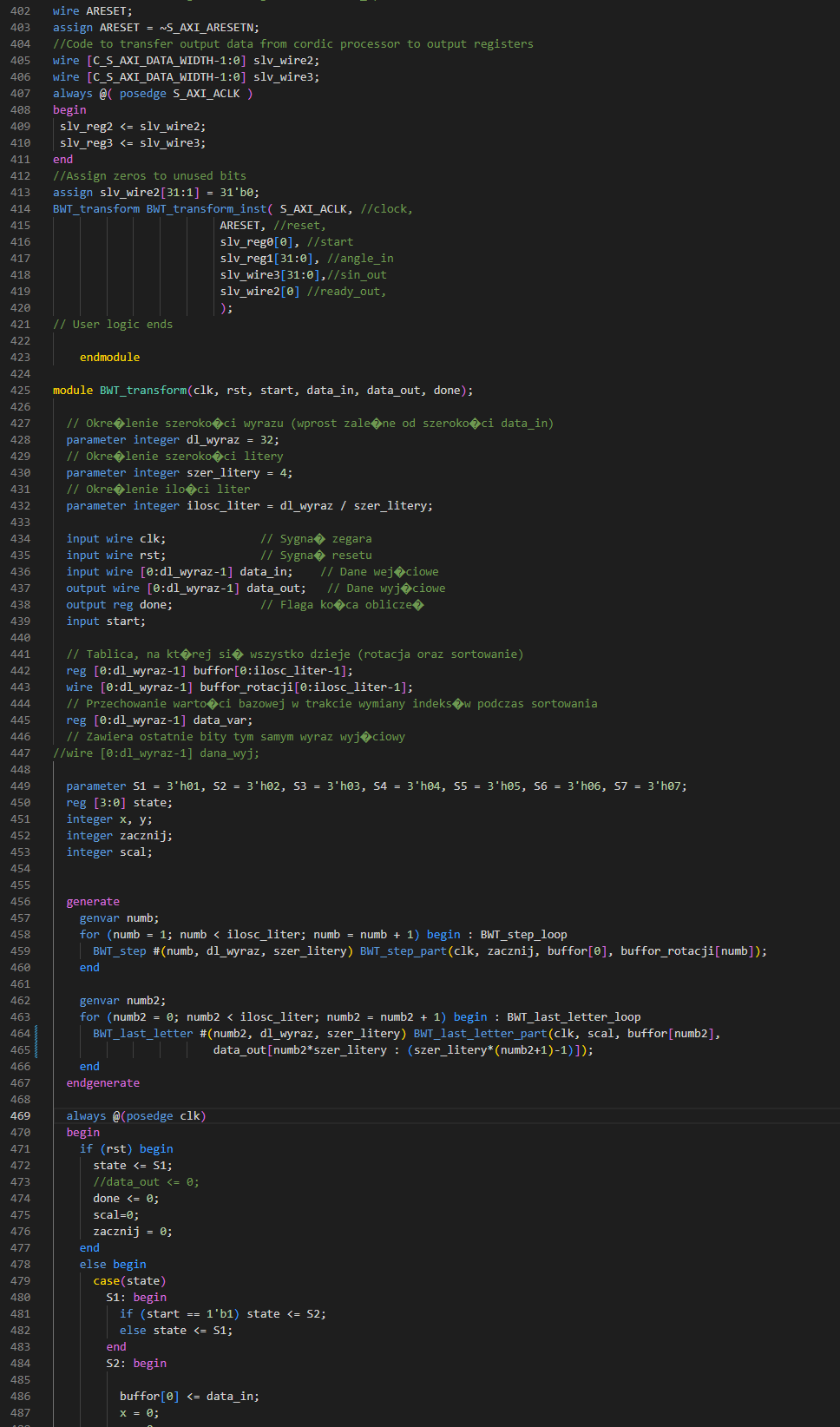
Na poniższych przebiegać można porównać czasy wykonania algorytmów/

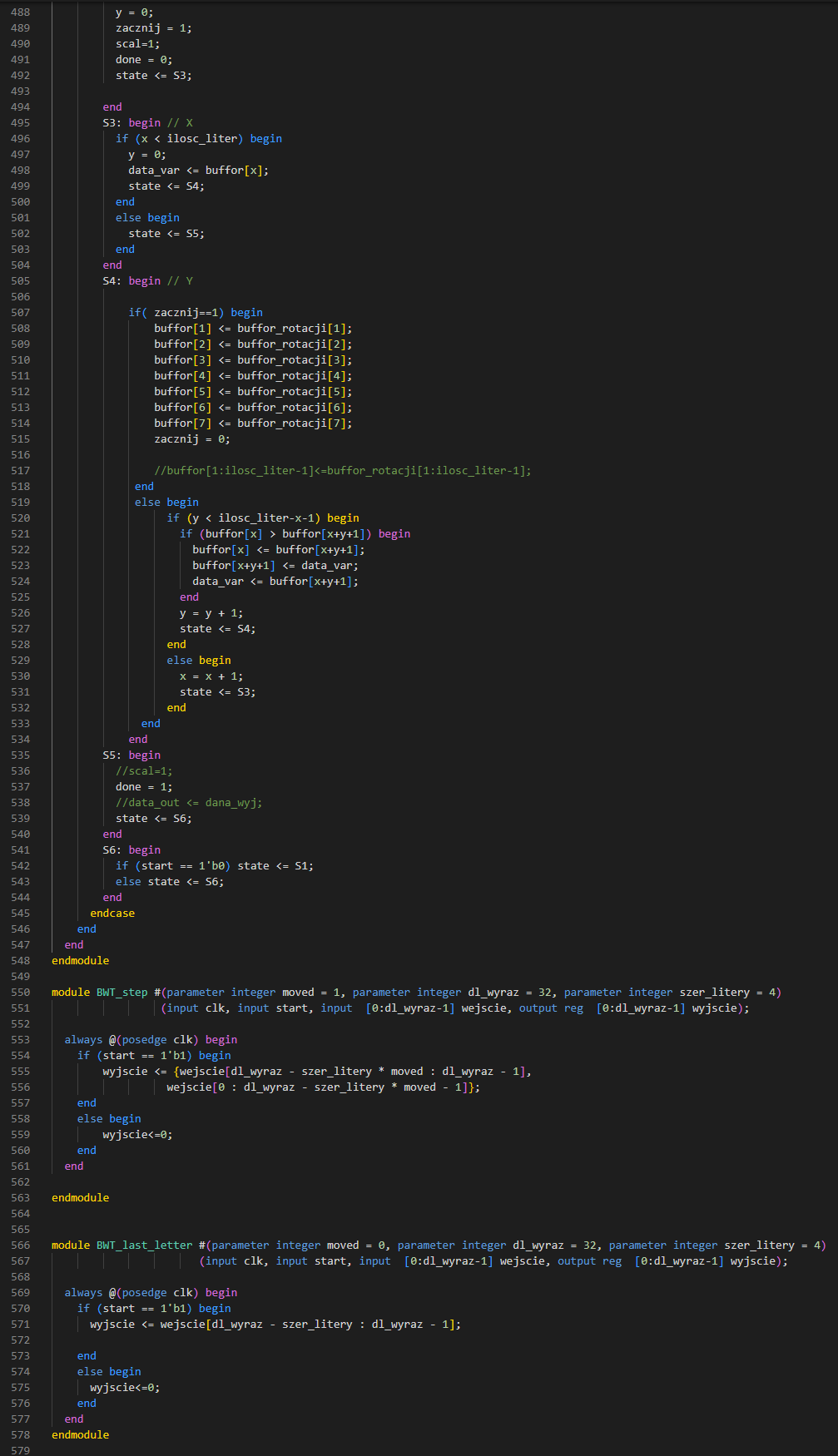


**W tym przypadku zastosowanie potoku pozwoliło skrócić czas symulacji o 25%!**

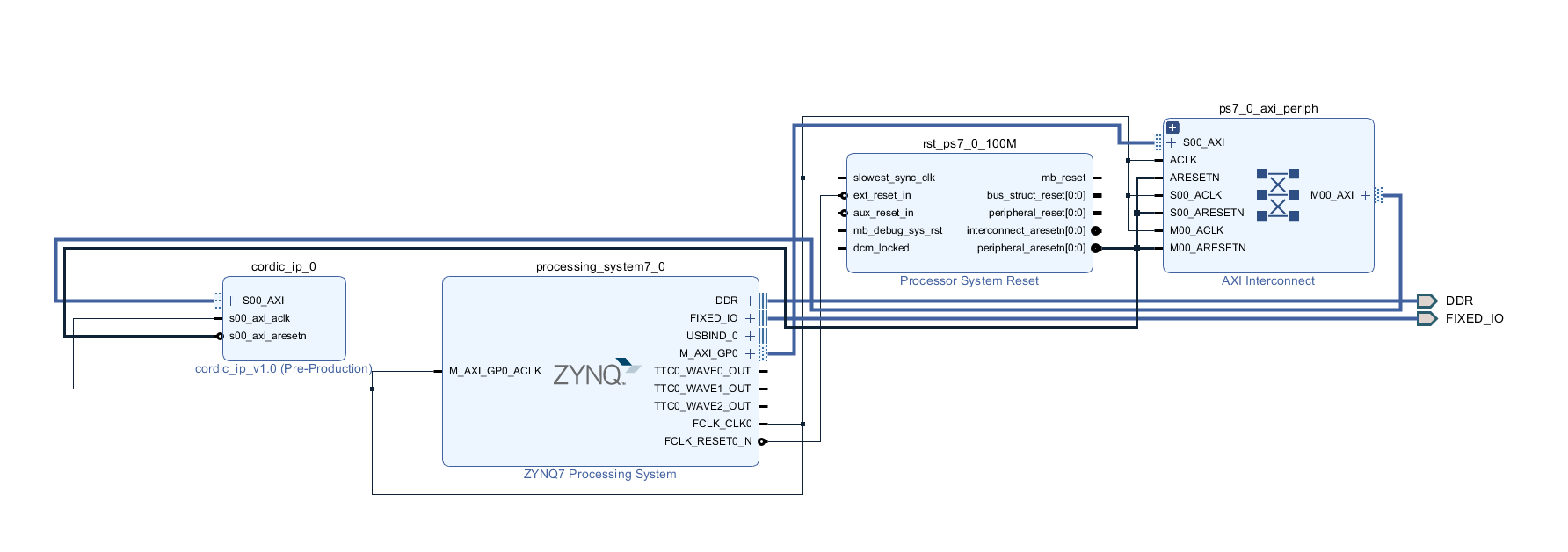
8. Przygotowanie modułu Ip

kod cordic Ip\_v1\_0\_S00\_AXI:

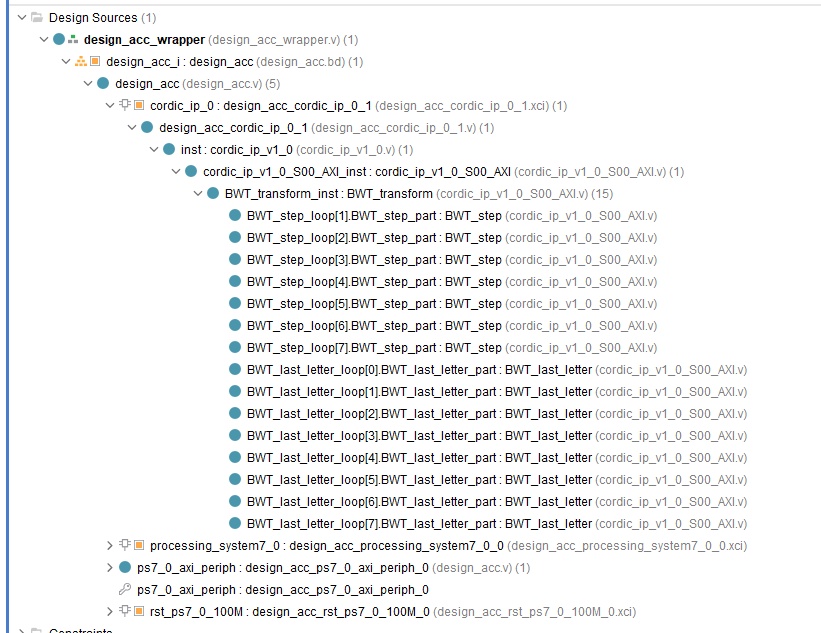




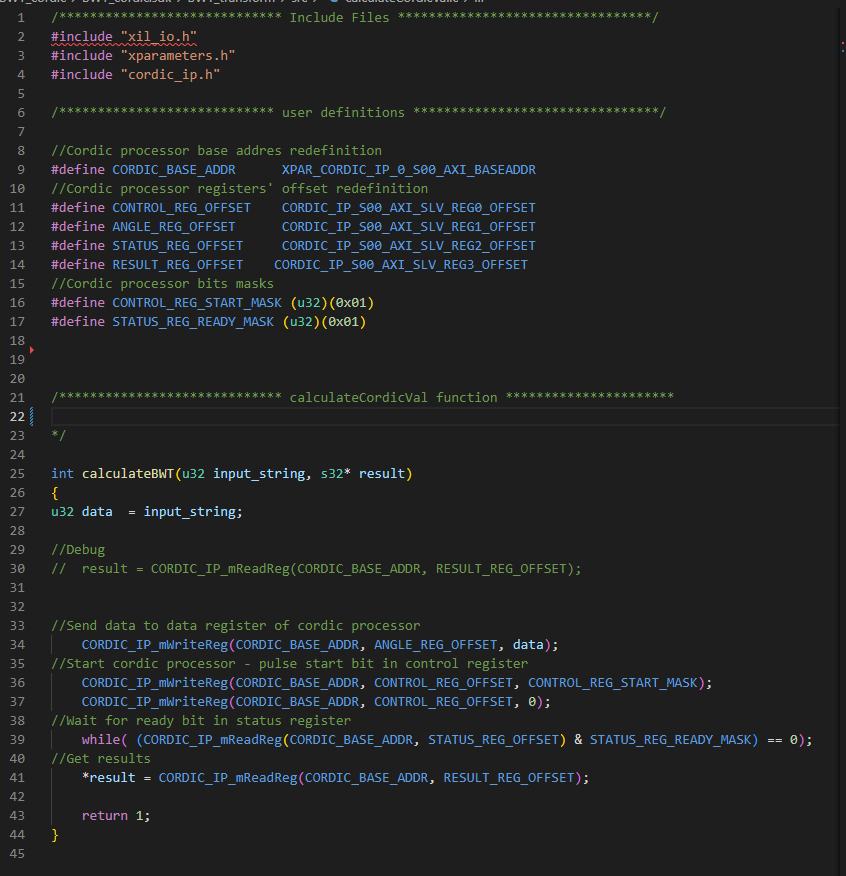
8. Przygotowanie diagramu blokowego Microblaze



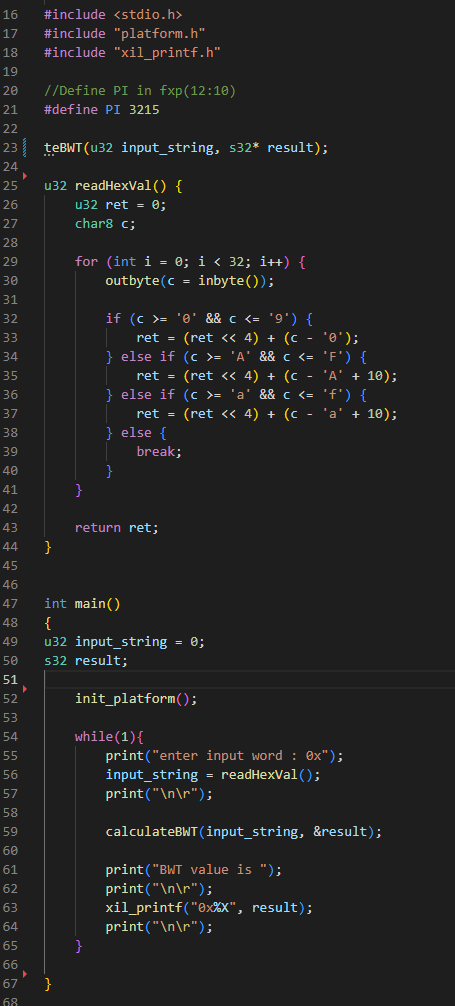
Struktura projektu:



8. Wyeksportowanie platformy sprzętowej do SDK

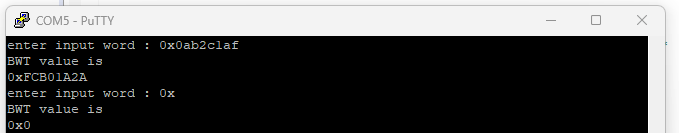


Kod “main.c”



8. Uruchomienie programu oraz zestawienie z pozostałymi wynikami

Uruchomienie algorytm na platformie Zedboard :



Poniższe ilustracje zestawiają wyniki otrzymane w poszczególnych etapach projektu:

