



AKADEMIA GÓRNICZO-HUTNICZA

Raport z projektu

# Sprzętowa realizacji funkcji $1/\text{sqr}(x)$

z przedmiotu

## Systemy Dedykowane w Układach Programowalnych

Elektronika i telekomunikacja- Systemy Wbudowane, rok I studiów  
magisterskich

*Agata Chowaniec*

*Iwona Suda*

23.06.2023

## Spis treści

1. Opis algorytmu.....	3
1.1 Porównanie algorytmów dodawania liczb.....	4
1.2 Porównanie algorytmów mnożenia liczb.....	6
1.3 Konwersja liczb .....	6
2. Realizacja sprzętowa.....	8
2.1 Symulacja na microblaze.....	8
2.2 Uruchomienie na fpga .....	10
Bibliografia .....	12

# 1. Opis algorytmu

Sednem działania algorytmu szybkiej odwrotności pierwiastka kwadratowego jest zapis danych w standardzie IEEE 754. Sprawia on, że operację przesunięcia bitowego o jeden w prawo można interpretować jako wartość logarytmu.

Algorytm działa w oparciu o reinterpretację bitów danych wejściowych zmiennoprzecinkowych jako liczby całkowitej. Są one przechowywane w zmiennej  $i$ . Następnie obliczana jest połowa wartości wejściowej i przechowywana do późniejszego wykorzystania. Przesunięcie logiczne w prawo o jeden bit jest wykonywane na liczbie całkowitej  $i$  (bitowo będącego reprezentacją liczbą w standardzie IEEE 754), a wynik przesunięcia jest odejmowany od liczby  $0x5F3759DF$ , która jest zmiennoprzecinkową reprezentacją przybliżenia pierwiastka kwadratowego z 2. Daje to pierwsze przybliżenie odwrotności pierwiastka kwadratowego z danych wejściowych. Następnie przeprowadza się jedną lub dwie iteracje metody Newtona-Raphsona w celu udoskonalenia przybliżenia [1]. Na potrzeby projektu wykonano jedną iterację metody Newtona-Raphsona.

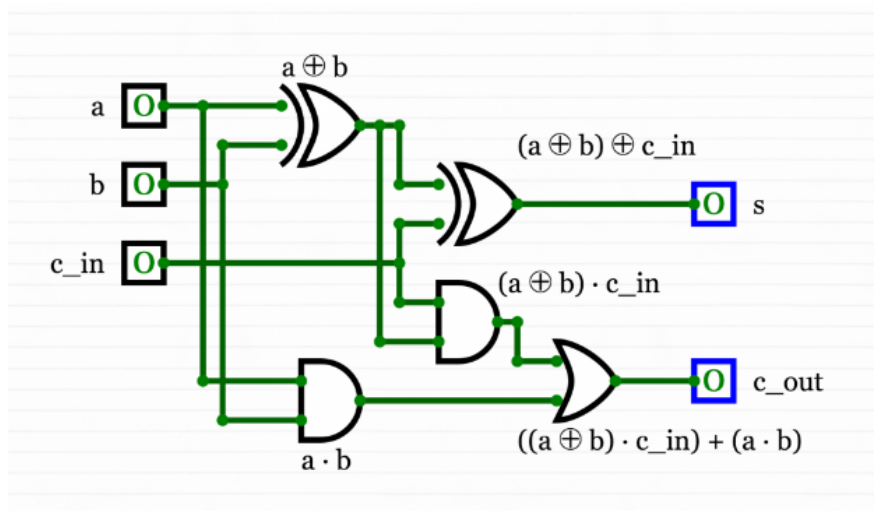
Algorytm opiera się w swoich założeniach na:

- Wykonywaniu operacji dodawania,
- Wykonywaniu operacji mnożenia,
- Wykorzystaniu zapisu fixed point do wykonania operacji logarytmowania.

Z powodu wykonywania wielu operacji mnożenia i dodawania koniecznym było zastosowanie liczb typu fixed point. Spowodowane jest to faktem, iż operacje dodawania dla fixed point są znacznie prostsze logicznie niż operacje na typie floating point.

## 1.1 Porównanie algorytmów dodawania liczb

Wykorzystując bramki XOR, AND oraz OR możemy zbudować układ dodawania dwóch liczb zapisanych w notacji fixed point.



Rysunek 1 Sumator liczb [2]

A i B są to nasze dwie dodawane liczby  $c_{in}$  oznacza (przeniesienie z poprzedniej pozycji) oraz przez  $c_{out}$  generowane przeniesienie (to, które powstanie w wyniku dodawania). Aby teraz ustalić funkcję logiczną sumy i przeniesienia, trzeba rozważyć trzy wartości wejścia i możliwości, dla których bity te przyjmują wartość 1.  $c_{out}$  przyjmuje wartość 1, gdy:

- $C_{in}$  ma wartość 1 oraz jedno z wejść a i b ma wartość 1, lub gdy
- wejścia a i b mają wartość 1 (niezależnie od wejścia  $c_{in}$ ).

Zatem funkcja logiczna opisująca to wyjście przyjmie postać:

$$c_{out} = (c_{in} \cdot (a \oplus b)) + (a \cdot b)$$

Ponownie, aby ustalić funkcję logiczną opisującą bit sumy, musimy przeanalizować wartości, dla których funkcja ta przyjmuje wartość 1.

Tutaj możemy próbować znaleźć pewne wspólne cechy, jak np. to, że gdy  $c_{in}=0$  oraz  $a \oplus b = 1$ , to  $s = 1$ , jednak można zauważyć, że  $s$  przyjmuje wartość 1, gdy na wejściu ma nieparzystą liczbę „jedynek”. Takie zachowanie ma bramka XOR, zatem możemy ustalić następujący wzór funkcji logicznej:

$$s = a \oplus b \oplus c_{in}$$

Dla liczb zapisanych w formacie floating point algorytm dodawania liczb jest znacznie bardziej skomplikowany, podobnie jak sam zapis liczby. Implementacja standardu IEEE 754 na poziomie sprzętowym może być kosztowna i wymagać specjalistycznych układów.

Mantysa sumy (jest sumą (lub różnicą) mantys liczb wyjściowych po sprowadzeniu ich do wspólnej cechy (operacja ta nosi nazwę wyrównania cech liczb zmiennoprzecinkowych). Cecha sumy (lub różnicy) jest równa sumie cech dodawanych (lub odejmowanych) liczb. Po wykonaniu operacji arytmetycznej mantysa wyniku jest sprowadzana do postaci znormalizowanej i zapamiętywana w kodzie liczby zmiennoprzecinkowej.

Fragment realizacji sprzętowej w System Verilog algorytmu powstały w pierwszej iteracji projektu [3]:

```

exponent_diff = exponent1 - exponent2;
mantissa2 = (mantissa2 >> exponent_diff);
{carry, mantissa_temp} = (sign1 ~^ sign2)? mantissa1 + mantissa2 : mantissa1 - mantissa2 ;
exp_adjust = exponent1;
if(carry) begin
    mantissa_temp = mantissa_temp >> 1;
    exp_adjust = exp_adjust + 1'b1;
end
else begin
    while( !mantissa_temp[23] ) begin
        mantissa_temp = mantissa_temp << 1;
        exp_adjust = exp_adjust - 1'b1;
    end
end
sign = sign1;
mantissa = mantissa_temp[22:0];
exponent = exp_adjust;
out = {sign, exponent, mantissa};

```

Wykonanie tego modułu wymaga więcej niż jeden cykl zegara. Z punktu widzenia modułu, który w zamyśle ma być szybki, lepsze są operacje na typie fixed point. Niemniej jednak wymagany jest konwersja liczby na zapis floating point – by wykonać wcześniej wspomnianą operację logarytmowania. Koszty konwersji są jednak mniejsze niż koszty wynikające z korzystania z modułów dla liczb w typie floating point.

Wykonanie obecnej w algorytmie operacji dodawania liczby ujemnej odbywa się w swojej logice jak dodawanie liczb w zapisie dwójkowym. Nie jest to operacja dodawania dwóch liczb zmiennoprzecinkowych.

```
i = 0x5f3759df - ( i >> 1 );
```

Na potrzeby realizacji projektu wykonano moduł oparty o operacje na liczbach zmiennoprzecinkowych, był on jednak zauważalnie wolny. Wynika to z faktu, iż operacja mnożenia/dodawania wymaga wielokrotnych obliczeń.

## 1.2 Porównanie algorytmów mnożenia liczb

Mnożenie liczb zmiennoprzecinkowych (floating point) jest bardziej skomplikowane niż mnożenie liczb całkowitych. Algorytm mnożenia liczb floating point polega na pomnożeniu mantys obu liczb, dodaniu cech i wykonaniu normalizacji wyniku.

Algorytm mnożenia liczb floating point można opisać krokami:

- pomnożenie mantys obu liczb,
- dodanie cech obu liczb,
- wykonanie normalizacji wyniku,
- zaokrąglenie wyniku do odpowiedniej liczby bitów mantysy.

Podsumowując, algorytm mnożenia liczb floating point polega na pomnożeniu mantys obu liczb, dodaniu cech i wykonaniu normalizacji wyniku [4]. Algorytm ten jest bardziej skomplikowany niż mnożenie liczb całkowitych lub binarnych. Z tego powodu wykonywano mnożenia na liczbach fixed point.

Ciekawszym zagadnieniem, z punktu widzenia tego projektu będą szybkie układy mnożące. Z perspektywy projektu napotkano konkretne problemy związane z mnożeniem. Etap syntezy i implementacji modułu nie powodował wypisywania błędów, niemniej środowisko sugerowało, iż operacja mnożenia, rozumiana jako operator  $*$  w System Verilog, dla dużych liczb nie będą wykonywane prawidłowo. Zgodnie ze znalezionymi informacjami układ stałoprzecinkowy DSP odpowiadający z mnożenie nie wykona w stososownych obliczeń.

Powodowało to powstanie dwóch możliwości rozwiązania problemu:

- implementacja mnożenia z wykorzystaniem przesunięć bitowych, oparta o metodę bezpośrednią,
- podzielenie liczby na mniejsze fragmenty mnożone osobno i sumowane – algorytm dziel i zwyciężaj,
- algorytm Bootha [5].

Zdecydowano się na realizację pierwszego z proponowanych rozwiązań, czyli metody bezpośredniej. Uznano, że nadrzędną wartością jest działający moduł, a nie rozwiązanie optymalne, lecz niedziałające.

## 1.3 Konwersja liczb

Aby zamienić liczbę zmiennoprzecinkową na stałoprzecinkową, można ją pomnożyć przez współczynnik skalowania, a następnie zaokrąglić wynik do najbliższej liczby całkowitej. Należy zadbać o to, aby wynik mieścił się w docelowej zmiennej lub rejestrze. W zależności od współczynnika skalowania i wielkości pamięci oraz zakresu liczb wejściowych przeliczenie może nie wiązać się z żadnym zaokrągleniem [6].

Aby zamienić liczbę stałoprzecinkową na zmiennoprzecinkową, można zamienić liczbę całkowitą na zmiennoprzecinkową, a następnie podzielić ją przez współczynnik skalowania.

Istnieje kilka metod i algorytmów konwersji stałoprzecinkowej na zmiennoprzecinkową, w tym wykorzystanie tablic przeglądowych, algorytmów iteracyjnych i przybliżeń wielomianowych. Wybór metody zależy od konkretnych wymagań aplikacji, takich jak pożądana dokładność, szybkość i zasoby sprzętowe.

Prostym podejściem do konwersji między liczbami stałoprzecinkowymi i zmiennoprzecinkowymi jest użycie szybkiego i brudnego algorytmu. Na przykład, aby przekonwertować ze stałego na zmiennoprzecinkowy można obliczyć wartość całkowitą jako liczbę zmiennoprzecinkową pomnożoną przez  $(2^x)$ , a następnie zaokrąglić wynik do najbliższej liczby całkowitej [7].

## 2. Realizacja sprzętowa

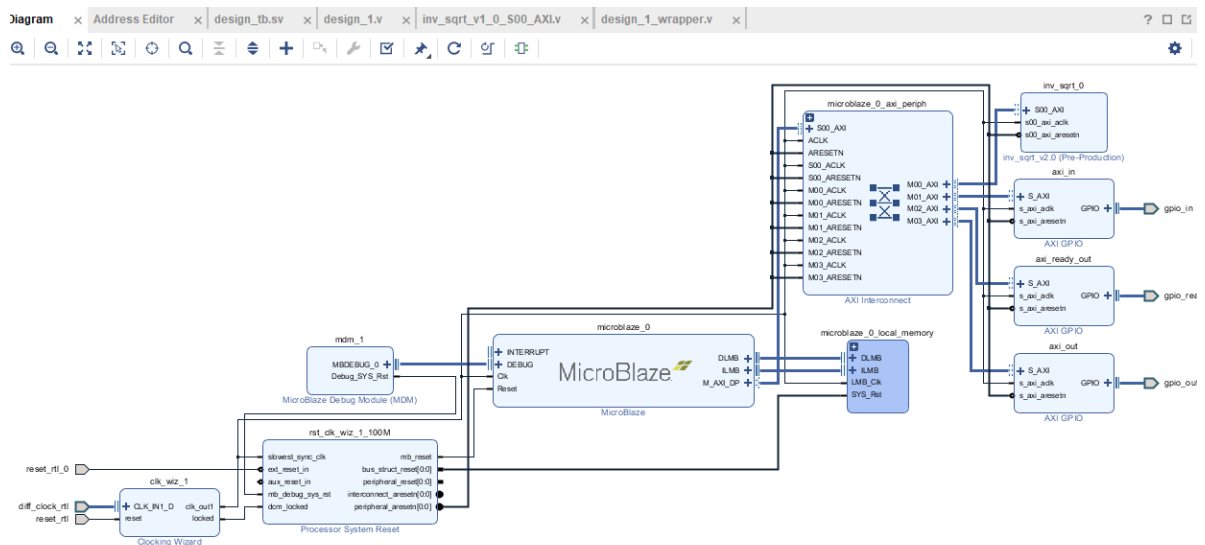
Do zaimplementowania kodu został użyty program Vivado 2018.3 oraz FPGA ZedBoard- Zynq-7000 EPP Development Kit



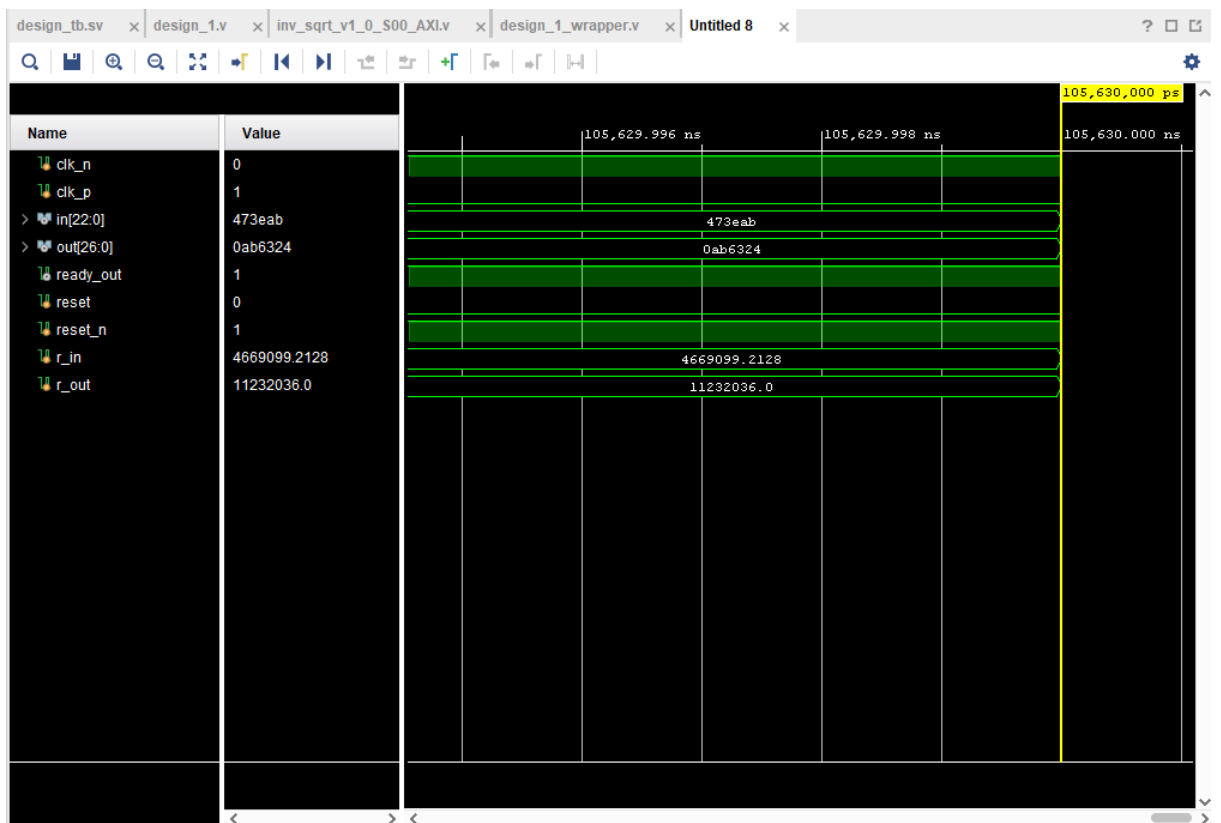
### 2.1 Symulacja na microblaze

Pierwsza iteracji uruchomienia symulacji z wykorzystaniem Microblaze zakończyła się brakiem sukcesu z powodu nieprawidłowego podłączenia sygnału resetu do modułu. Dopiero wyprowadzenie go na GPIO i bezpośrednie ustawienie rejestru pozwoliło na poprawne rozpoczęcie pracy modułu, a także na uzyskanie wyniku. GPIO zostały skonfigurowane zgodnie z długością rejestru wejściowego i wyjściowego. Przeprowadzono symulację oraz sprawdzono wyniki.

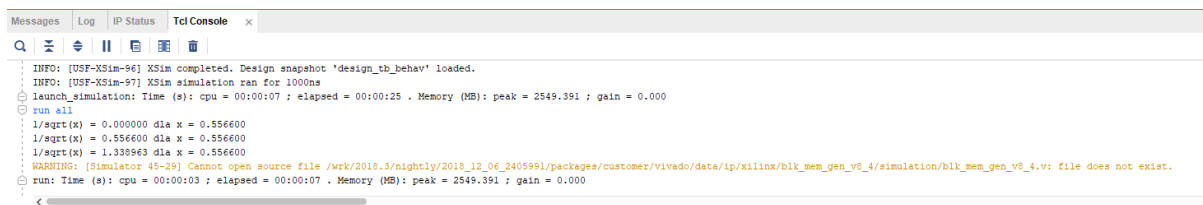




Rysunek 2 Schemat połączenia MicroBlaze z modulem inv\_sqrt



Rysunek 3 Przebieg czasowy symulacji



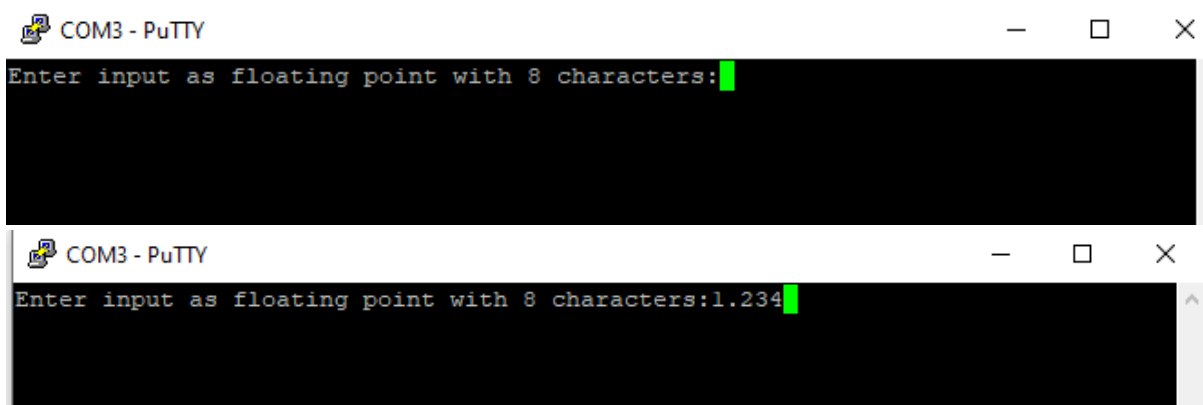
```
Messages | Log | IP Status | Tcl Console x
INFO: [USF-XSim-96] XSim completed. Design snapshot 'design_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:07 ; elapsed = 00:00:25 . Memory (MB): peak = 2549.391 ; gain = 0.000
run all
1/sqrt(x) = 0.000000 dla x = 0.556600
1/sqrt(x) = 0.556600 dla x = 0.556600
1/sqrt(x) = 1.339563 dla x = 0.556600
WARNING: [Simulator 45-29] Cannot open source file /wrk/2018.3/nightly/2018_12_06_2405991/packages/customer/vivado/data/ip/xilinx/blk_mem_gen_v6_4/simulation/blk_mem_gen_v6_4.v: file does not exist.
run: Time (s): cpu = 00:00:03 ; elapsed = 00:00:07 . Memory (MB): peak = 2549.391 ; gain = 0.000
```

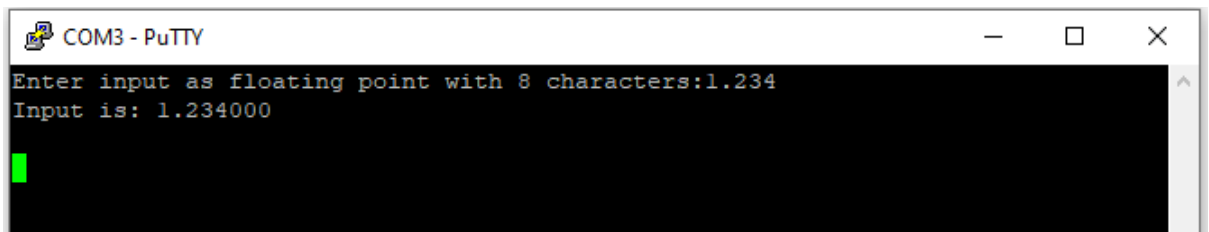
Rysunek 4 Rezultat symulacji

## 2.2 Uruchomienie na fpga

Część dotycząca uruchomienia projektu na układzie fpga zakończyła się niepowodzeniem. Układ po podłączeniu do komputera raportował błędy i wymagał częstego resetowania. Po stworzeniu aplikacji na wbudowany procesor ARM aplikacja zawieszała się na próbie odczytania zawartości rejestru zawierającego flagę gotowości danych wyjściowych – pod zadany adres znajdowało się 32'b00000000000000000000000000000000, choć oczekiwana była wartość równoważna uint32\_t 1. Powodowało to brak przerwania nieskończonej pętli while. Zweryfikowano poprawność wprowadzanych danych, a także doprowadzenie sygnałów. Założono, że moduł inv\_sqrt powinien działać prawidłowo, jako że działał dla symulacji z MicroBlaze.

W obliczu napotkanych problemów zrezygnowano z uruchomienia modułu, jako że rozwiązywanie problemów o nieznanym podłożu wykracza poza zakres tego projektu. Na etapie projektowania układu występowały liczne problemy z oprogramowaniem Vivado, które prawidłowo syntezowało układy o takim samym opisie tylko na urządzeniu jednej ze stron wykonujących projekt.





**Rysunek 5 Wprowadzenie danych wraz z potwierdzeniem aplikacji dotyczącego prawidłowej interpretacji danych wejściowych**

Name	Type	Value
(x)= ret	float	1.234000
(x)= c	char8	'\r'
> buffer	char [64]	"1.234000\000^\2641"\202FpU\021q\026p0\0...
> number	char [9]	"1.234\000\000\000\000"
(x)= i	uint8_t	'\005'
(x)= flag	int	1

**Rysunek 6 Interpretacja danych wejściowych przez program - widok z debuggera**

## 2.3 Załączniki

W ramach projektu przygotowano:

- Moduł w System Verilog z realizacją sprzętową algorytmu odwrotności pierwiastka kwadratowego: inv\_sqrt.sv
- Plik testowy do modułu: inv\_sqrt\_tb.sv
- Moduł AXI: inv\_sqrt\_v2\_0\_S00\_AXI.v
- Plik testowy do MicroBlaze: design\_tb.sv
- Pliki w języku C do MicroBlaze: main.c
- Pliki w języku C do ARM : getResult.c, main.c

# Bibliografia

- [1] **vishal9619**. Fast inverse square root. [Online] <https://www.geeksforgeeks.org/fast-inverse-square-root/>.
- [2] Układy liczące: sumator. [Online] <https://zeszyt.jedlikowski.com/2020-01-05/utk/uklady-liczace-sumator/>.
- [3] **aklim**. FPU-IEEE-754. [Online] <https://github.com/akilm/FPU-IEEE-754/tree/main>.
- [4] **debtanu09**. Verilog implementation of IEEE 754 32 bit floating point multiplier. [Online] <https://github.com/debtanu09/fmultiplier>.
- [5] Szybkie układy mnożące. [Online] <https://docplayer.pl/119233306-Szybkie-uklady-mnozace.html>.
- [6] Fixed-point arithmetic. [Online] [https://en.wikipedia.org/wiki/Fixed-point\\_arithmetic](https://en.wikipedia.org/wiki/Fixed-point_arithmetic).
- [7] Floating-point to Fixed-point. [Online] <https://ee.sharif.edu/~asic/Tutorials/Fixed-Point>.