AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

SYSTEMY DEDYKOWANE W UKŁADACH PROGRAMOWALNYCH

Algorytm *Fast inverse square root*

Autorzy: *Łukasz Orzeł; Jakub Świebocki*

Kierunek studiów: *Mikroelektronika w technice i medycynie*

Kraków, 2023

Spis treści

Wstęp

W niniejszym raporcie przedstawiamy wyniki projektu, którego celem było zaimplementowanie algorytmu Fast Inverse Square Root na układzie FPGA (Field-Programmable Gate Array). Jako, że algorytm Fast Inverse Square Root, ze względu na szybkość działania, był szeroko stosowany w dziedzinie grafiki komputerowej oraz obliczeń naukowych, prezentujemy również przykładowe aplikacje, które korzystają z obliczeń naszej implementacji.

Pierwszym zastosowaniem algorytmu Fast Inverse Square Root było zaimplementowanie w grze komputerowej Quake III Arena. Algorytm ten wykorzystano w celu przyspieszania obliczeń związanych z oświetleniem i renderingiem graficznym. Stąd też nazwa tego algorytmu to Quake Fast Inverse Square Root algorithm. Działa on na podstawie magii bitowej (bitwise magic) oraz manipulacji liczbami zmiennoprzecinkowymi w reprezentacji binarnej.

1. Informacje projektowe

1.1. Struktura projektu

Struktura projektu Fast_inverse_square_root na GitHub:

- BehavioralModel - zawiera plik z kodem w C oraz plik .exe, aby można było uruchamiać gotowy skrypt.
- FPGA_files - zawiera plik IP (ip_repo) oraz wszystkie pliki wykorzystane do potokowego modelu syntezowalnego (src, sim, constr_1)
- PythonApp - zawiera przykładowe aplikacje w Pythonie, które przedstawiają praktyczne działanie algorytmu FISR
- Report_files - zaweira plik raportu, zdjęcia wykorzystane w raporcie, filmy przedstawiające działanie skrytpów
- SDK_files - projekt możliwy do uruchomienia na SDK, w celu zaprogramowania procesora.
- scripts - inne skrypty, które były pomocne przy testowaniu, tworzeniu kodu.

1.2. Wykorzystywane zasoby

Projekt realizowany w Vivado 2018.2, SDK, Python 3.9, Windows. Wykorzystany sprzęt - płytka rozwojowa Zedboard ZYNQ-7000.

2. Teoretyczny wstęp algorytmu

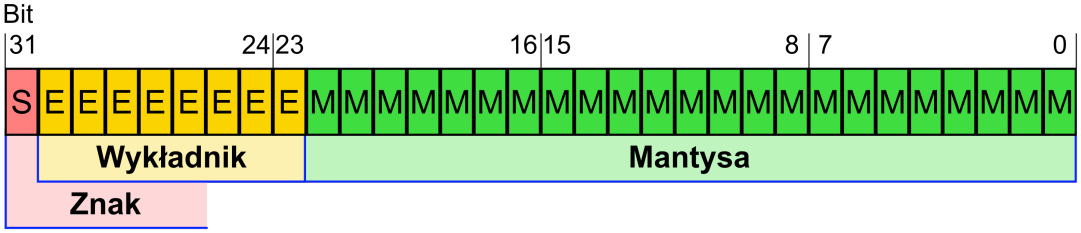
2.1. Standard IEEE754

Algorytm działa w oparciu o zastosowanie standardu IEE754 formalnie dzielący zmienną na 3 części - znak, wykładnik i mantysa.

- **Znak** - Gdy jest równy 1, wówczas liczba będzie ujemna. Gdy jest 0, liczba jest dodatnia
- **Wykładnik** - 8 bitów kodujących wykładnik 2. Tutaj dodatkowa cecha to odejmowanie od wykładnika 127 (BIAS) co daje zakres $\langle -127, 128 \rangle$ dając do dyspozycji zapisanie liczb bardzo dużych oraz bardzo małych
- **Mantysa** - 24 bity, gdzie 23 bity są zawsze używane a pierwszy bit jest pomijany gdyż jego wartość jest zawsze ustalona na 1. Taka operacja jest zastosowana ponieważ w założeniu jest że liczba ta będzie miała reprezentację typu 1.xxx ... xxx.

Liczba zapisana w takim formacie (Rys. 2.1) dzisiaj nazywa się pojedynczej precyzji. Miejsce bitowe na nią przeznaczone jest bardzo dobrze wykorzystane, jednak niesie ze sobą pewne absurdy:

- **Dwa rodzaje zer** - zero dodanie i zero ujemne
- **Dwa rodzaje nieskończoności** - nieskończoność dodatnia i nieskończoność ujemna
- **Not a Number** (NaN)
- Znacząco niższa precyzja zapisu, gdy wykładnik jest równy zero (**liczby bardzo małe**)



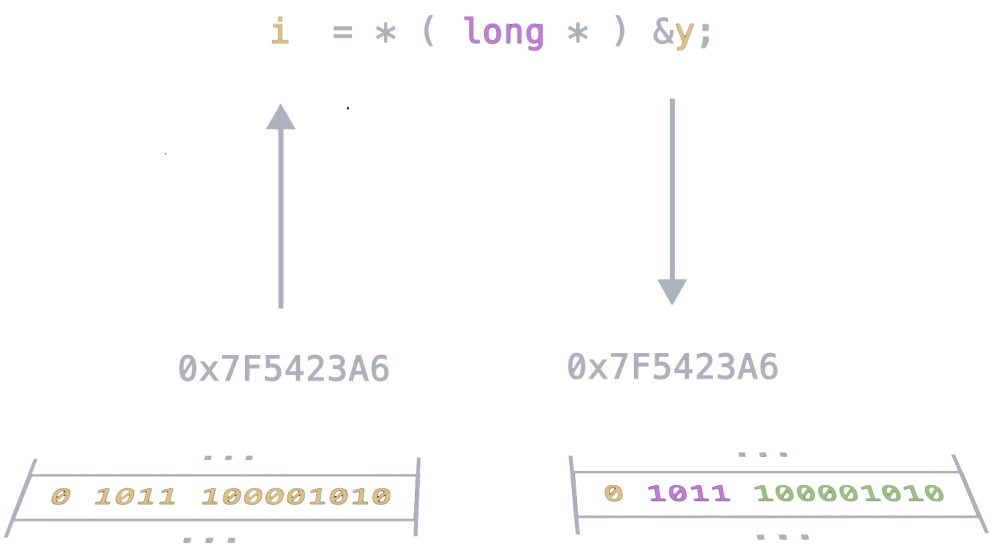
Rysunek 2.1: Standard IEEE754

2.2. Opis algorytmu

Podstawowy schemat działania algorytmu Fast Inverse Square Root można przedstawić w kilku krokach:

1. Wykorzystanie reprezentacji liczby zmiennoprzecinkowej i zapisanie jej ciąg bitów w zmiennej całkowitej (Rys. 2.2). Tutaj jest wykorzystywany adres, gdzie przechowywana jest zmienna i kopiowana jest zawartość do drugiej zmiennej. Dzięki temu możemy pracować na znanej nam liczbie jak na danym ciągu bitów.

$$iDestination = *(long*)\ \&fSource$$



Rysunek 2.2: Przeniesienie wartości standardu IEEE754 do zmiennej całkowitej (Nemean (2020))

2. Zastosowanie magicznej liczby 0x5f3759df czyli pewnych operacji matematycznych i bitowych na tej reprezentacji, które mają na celu przybliżone obliczenie wartości odwrotności pierwiastka kwadratowego. Wykorzystywany jest właśnie standard IEEE754.

   Wprowadzając założenia pracy na logarytmach, obliczenie algorytmu staje się możliwe.

$$log(IEE754) = \frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127 \quad \begin{array}{l} M - mantysa \\ E - Wykladnik \\ \mu - stala \\ -127 - bias \end{array} \tag{2.1}$$

$$\frac{1}{\sqrt{y}} \ -> \ log\left(\frac{1}{\sqrt{y}}\right)$$

$$log\left(\frac{1}{\sqrt{y}}\right) \ -> \ log\left(y^{-\frac{1}{2}}\right) \ -> \ -\frac{1}{2}log\,(y)$$

Zakładając że rozwiązanie nasze to $log(\Gamma)$:

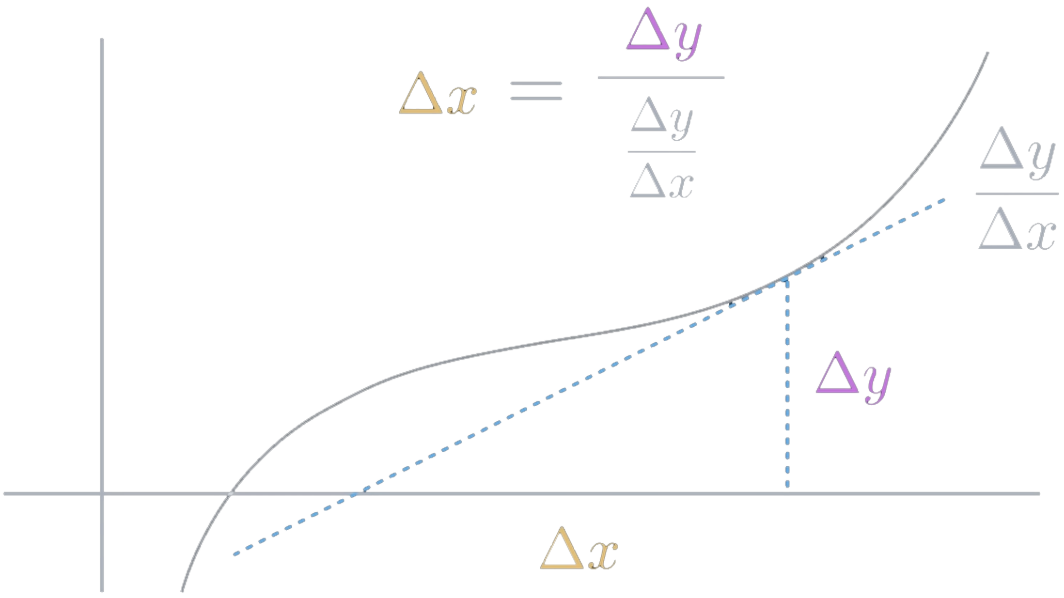
$$log(\Gamma) = -\frac{1}{2} log(y)$$

$$\frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127 = -\frac{1}{2}\left(\frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127\right)$$

$$\begin{aligned}(M_\Gamma + 2^{23} * E_\Gamma) &= \frac{3}{2}2^{23}(127 - \mu) - \frac{1}{2}(M_y + 2^{23} * E_y) \\ &= 0x5F3759DF - (iDestination >> 1)\end{aligned}$$

W ostatecznym rozrachunku stosowaliśmy logarytmy, które się znoszą, dzięki zastosowaniu ich po obu stronach równania. Natomiast dzielenie przez 2 jest wykonywane operacją sprzętową, czyli przesuwania bitów w daną stronę.

3. Ostateczne dostosowanie wyniku, aby uzyskać większą precyzję, jest realizowane za pomocą przybliżenia Newton-Raphson'a (Rys. 2.3). W tej metodzie skupiamy się na otrzymanym wyniki x_0 i dokonujemy jego interpolacji. Im więcej razy zostanie powtórzona interpolacja, tym mniejszy potencjalnie jest błąd wyniku w porównaniu do wartości rzeczywistej.



Rysunek 2.3: Przybliżenie Newton-Raphson (Nemean (2020))

W kolejnych sekcjach raportu szczegółowo omówimy proces implementacji algorytmu Fast Inverse Square Root na układzie FPGA oraz przedstawimy wyniki naszych badań i analizę efektywności implementacji.

3. Model behawioralny algorytmu

3.1. Implementacja modelu w języku C

Wykorzystano algorytm Quake opisany we wstępie raportu. Stworzono w tym celu funkcję *Q_rsqrt()*. Na wejściu otrzymuje dane typu *float*, a następnie implementuje opisany wcześniej algorytm. Fragment kodu dotyczący tej funkcji przedstawiono poniżej

```
int Q_rsqrt( float number ){
    long i;
    unsigned long r;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;
    i  = 0x5f3759df - ( i >> 1 );
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );
    r  = * ( long * ) &y;
    return r;
}
```

Listing 3.1: Model behawioralny algorytmu Fast Inverse Square Root

3.2. Testbench modelu behawioralnego w języku C

Testbench przyjmuje dane wprowadzane z konsoli przez użytkownika w postaci *float*. Następnie użytkownik dostaje informację zwrotną o wartości wyliczoną przez algortym Fast Inverse Square Root i przez wbudowaną funkcję *sqrt()* z biblioteki *math.h* oraz różnicę między tymi wynikami.

Poniżej umieszczono kod testujący - testbench:

```
int main(){
    while(1){
        float input;
        printf("Enter a value: ");
        scanf("%f", &input);

        if(input <= 0.0){
            printf("This value is not allowed\n");
        }
        else{
            float math_out  = 1.0 / sqrt(input);
            float fisr_out = ieee754_to_float(Q_rsqrt(input));
            float abs_val = fabs(fisr_out - math_out);
            printf("-------------------------------------\n");
            printf("Entered Value -> %.3f\n", input);
            printf("Algorithm Output -> %.8f\n", fisr_out);
            printf("Math.h Function -> %.8f\n", math_out);
            printf("Difference |FISR-MATH| -> %.8f\n",abs_val);
            printf("-------------------------------------\n");
        }
    }
}
```

Listing 3.2: Testbench modelu behawioralnego

W tym kodzie znajduje się również funkcja odpowiedzialna za konwersję liczby wyjściowej z algorytmu na wartość *float*. Jej działanie jest zgodne z opisem przedstawionym na Rys. 2.2. Poniżej umieszczono kod funkcji *ieee754_to_float()*

```
float ieee754_to_float(unsigned int value){
    return *((float*)&value);
}
```

Listing 3.3: Konwersja z IEEE754 do float

3.3. Prezentacja wyników

Poniżej przedstawiono działanie kodu. Otrzymywane wyniki potwierdzają poprawne funkcjonowanie algorytmu Quake. Można przyjąć, że aktualnie dokładniejsze wyniki posiadają funkcje wbudowane, jak np. sqrt(), jednakże algorytm Fast Inverse Square Root również działa z dużą precyzją oferując szybkość przetwarzania. Precyzję tą można zwiększać poprzez stosowanie powtórnego przybliżenia (Rys. 3.1a) Newtona-Raphsona, jednak zwiększa to czas wykonywania funkcji.

Kod jest zabezpieczony przed wprowadzaniem danych ujemnych oraz 0 (Rys. 3.1b).

```
            1x Newton-Raphson
Enter a value: 1
-----------------------------------------
Entered Value -> 1.000
Algorithm Output -> 0.99830717
Math.h Function -> 1.00000000
Difference |FISR - MATH| -> 0.00169283
-----------------------------------------

            2x Newton-Raphson
Enter a value: 1
-----------------------------------------
Entered Value -> 1.000
Algorithm Output -> 0.99999565
Math.h Function -> 1.00000000
Difference |FISR - MATH| -> 0.00000435
-----------------------------------------

            3x Newton-Raphson
Enter a value: 1
-----------------------------------------
Entered Value -> 1.000
Algorithm Output -> 0.99999994
Math.h Function -> 1.00000000
Difference |FISR - MATH| -> 0.00000006
```

(a) Prezentacja zwiększania precyzji

```
Enter a value: 0
This value is not allowed
Enter a value: -0.01
This value is not allowed
Enter a value: 1
-----------------------------------------
Entered Value -> 1.000
Algorithm Output -> 0.99830717
Math.h Function -> 1.00000000
Difference |FISR - MATH| -> 0.00169283
-----------------------------------------
Enter a value: 0.5
-----------------------------------------
Entered Value -> 0.500
Algorithm Output -> 1.41386008
Math.h Function -> 1.41421354
Difference |FISR - MATH| -> 0.00035346
-----------------------------------------
Enter a value: 5
-----------------------------------------
Entered Value -> 5.000
Algorithm Output -> 0.44714102
Math.h Function -> 0.44721359
Difference |FISR - MATH| -> 0.00007257
-----------------------------------------
Enter a value: 0.001
-----------------------------------------
Entered Value -> 0.001
Algorithm Output -> 31.58506393
Math.h Function -> 31.62277603
Difference |FISR - MATH| -> 0.03771210
-----------------------------------------
Enter a value: 1000
-----------------------------------------
Entered Value -> 1000.000
Algorithm Output -> 0.03156984
Math.h Function -> 0.03162277
Difference |FISR - MATH| -> 0.00005293
-----------------------------------------
```

(b) Wyniki dla poszczególnych danych

Rysunek 3.1: Prezentacja działania kodu

4. Potokowy model syntezowalnego algorytmu

4.1. Układ przeliczenia wstępnego

Układ ten ma za zadanie wstępne oszacowanie wartości $\frac{1}{\sqrt{x}}$. Do tego celu jest wykorzystywana stała, nazwaną MAGIC oznaczająca wartość *0x5F3759DF*. Dodatkowo dzielenie przez 2 jest wykonywane przez operację przesunięcia bitowego. W tym module również jest obliczana wartość połowy danej wejściowej, potrzeba w dalszych obliczeniach algorytmu.

```
1 Half_DataIN_nxt = {1'b0, DataIn[30:23] - 8'b0000_0001, DataIn
     [22:0]};
2 DataOut_nxt = MAGIC - (DataIn >> 1);
```

Listing 4.1: Układ przeliczenia wstępnego - fragment kodu

4.2. Układ mnożący

Ta część algorytmu jest odpowiedzialna za przemnożenie dwóch wartości w standardzie IEEE754. Pierwsza część kodu dodaje wykładniki obu liczb i odejmowana jest wartość 127 (bias umożliwiający zapisywanie liczb zmiennoprzecinkowych). W następnej kolejności mnożone są ze sobą mantysy z uwzględnieniem jedynki na najstarszym bicie. Jedynki te nie występują bezpośrednio w samej liczbie, ponieważ mantysy są zapisywane w formacie 1.xxxxxx. Oznacza to, że najbardziej znacząca cyfra zawsze jest jedynką i nie musi być bezpośrednio zapisywana. Ostatni etap tego układu jest zapisanie liczby nadal do formatu 32 bitowego. Aby to zrobić wyniki mnożenia mantysy jest zapisywany do 48 bitowego rejestru a w dalszej kolejności jest odpowiednio normalizowane w zależności od tego na którym miejscu jest najstarsza jedynka.

```
1 E_Square_nxt = Number_1[30:23] + Number_2[30:23] - 127;
2 M_Square_nxt = ( {1'b1, Number_1[22:0]} * {1'b1, Number_2[22:0
     ]} );
3
4 Product_nxt = {Sign, E_Square + M_Square[47], ( M_Square[47] ?
     M_Square[46:24] : M_Square[45:23] )};
```

Listing 4.2: Układ mnożący - fragment kodu

4.3. Układ odejmujący

W układzie tym, odejmowanie zachodzi poprzez odejmowanie pierwszej liczby od drugiej przesuniętą o różnicę między wykładnikami obu liczb. W tej części jedynka jest również dodawana na najstarsze miejsce mantysy ze względu na działanie standardu. Po wstępnej operacji odejmowania konieczna jest normalizacja liczby poprzez przesunięcie wszystkich bitów w lewą stronę, tak aby na dwóch najstarszych bitach mantysy były znaki 01xxx ... xxx. Na końcu po dokonanej normalizacji liczba jest składana w jedną 32 bitowa wartość.

```
Sub_mantissa_nxt = ( {1'b1, OneAndHalf[22:0]} ) - ( {1'b1, NumB
   [22:0]} >> (OneAndHalf[30:23] - NumB[30:23]) );

if(Sub_mantissa[23] == 1) begin
    M_Norm_nxt = Sub_mantissa << 1;
    E_Norm_nxt = E_max - 0;
    end

NumOut_nxt = {Sign, E_Norm, M_Norm[23:1]};
```

Listing 4.3: Układ odejmujący - fragment kodu

4.4. Implementacja kodu

Wszystkie bloki posiadają doprowadzenie do CLK oraz RST. Dane wejściowe w pierwszej kolejności wchodzą na moduł wstępnego obliczenia żądanej wartości a następnie do modułu korekcyjnego Newton-Raphson. Taka hierarchia pozwala na łatwe zwiększanie precyzji, poprzez proste dokładanie kolejnego takiego bloku w szeregu.

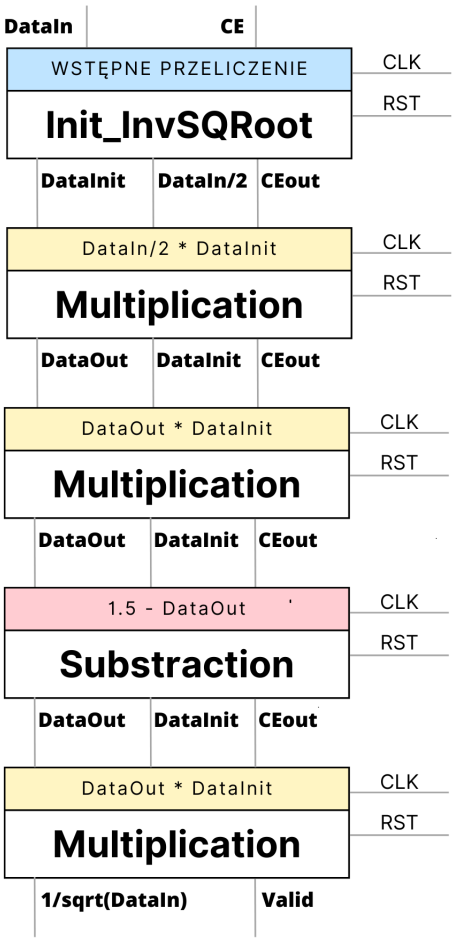
Algorytm ten zawiera dodatkowo sygnały CE oraz VALID. Sygnał VALID mówi o tym czy dana na wyjściu jest potencjalnie prawidłową, biorąc pod uwagę ilość cykli konieczną do przejścia. Sygnał CE może wyłączać CLK na określony czas, co powoduje zatrzymanie pracy algorytmu. Zatrzymanie to nie następuję od razu, ale propaguję się równo z pracą potoku.

```
//Init_InvSQRoot
Init_InvSQRoot Init_InvSQRoot(
    .clk(clk),
    .rst(rst),
    .ce(ce),
    .DataIn(DataIn),
    .DataOut(InitData),
    .Half_DataIN(Half_DataIN),
    .ce_out(ce_1)
    );
```

```
11
12 //NewtonApprox
13 NewtonApprox NewtonApprox_1(
14     .clk(clk),
15     .rst(rst),
16     .ce(ce_1),
17     .Data_in1(InitData),
18     .Data_in2(Half_DataIN),
19
20     .Data_out(Data_result),
21     .Valid(Valid)
22     );
23 //Assings
```

Listing 4.4: Implementacja algorytmu - fragment kodu

4.5. Schemat blokowy



Rysunek 4.1: Schemat blokowy algorytmu

4.6. Testbench

Układ testujący posiada wprowadzane dane do obliczeń, wyniki pochodzące od algorytmu oraz poprawne wyniki. Początkowo dane są wprowadzane do algorytmu co takt zegara a następnie zapasywane do pliku. Gdy zapis zostanie ukończony pomyślnie rozpoczyna się druga faza testowania, która polega na porównaniu wyników algorytmu oraz wprowadzonych rzeczywistych wyniki do 10 miejsca po przecinku. Jeśli porównywane dane różnią się na 6 miejscu po przecinku to zwracana jest informacja, dla jakiej wartości wejściowej różnica jest znacząca (ustalone przez użytkownika).

```
task compare_data();
    begin
        $readmemb("InputData.mem", inputs_data);
        $readmemh("OutputData.mem", verilog_outputs);
        $readmemh("Output_C_data.mem", c_outputs);
        $display("Start comparing...");
        for (i=0; i<Samples; i=i+1) begin
            input_data = inputs_data[i];
            c_output = c_outputs[i];
            verilog_output = verilog_outputs[i];
            div = (verilog_output > c_output) ? verilog_output
  - c_output: c_output - verilog_output;
            if(div > 3) begin
                $display("%d. Different outputs for %h, C
  output: %h, Verilog output: %h, Difference: %h", i+1,
  input_data, c_output, verilog_output, div);
            end
            #10;
        end
        $display("Comparison done...");
    end
endtask
////////////////////////////////////////
```

Listing 4.5: Testbench - porównanie danych

```
initial begin
    stop = 0;
    file_handle = $fopen("OutputData.mem", "wb");
    $display("OutputData has been opened!");
    $readmemb("InputData.mem", memory);
    $display("Reading InputData...");
    for (i=0; i<Samples; i=i+1) begin
        if(i >= 400 && i <= 410) ce = 1'b0;
        else ce = 1'b1;
        DataIn = memory[i];
        #10;
        end
    $display("Reading completed!");
    #100; $fclose(file_handle);
    $display("OutputData has been closed!");
    stop = 1;
    compare_data();
    $stop;
    end
```

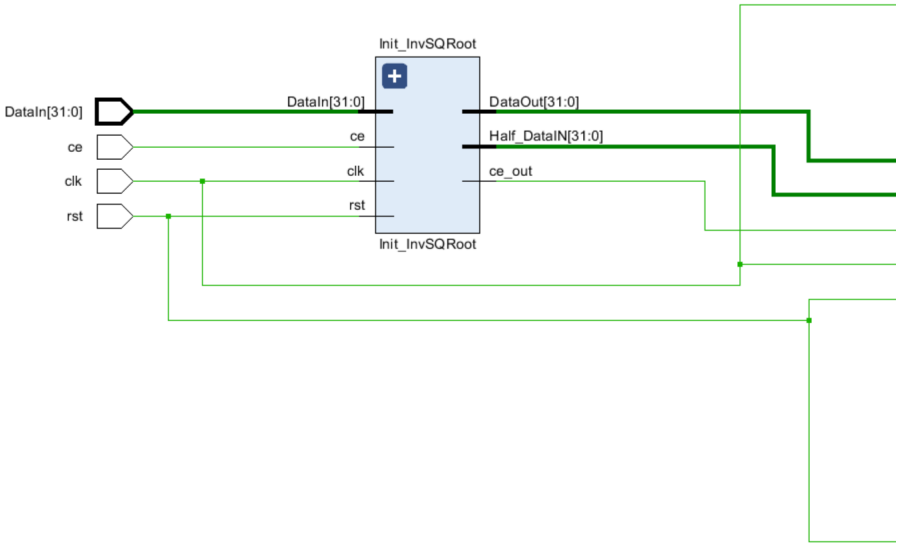
Listing 4.6: Testbench - przeliczenie danych wejściowych

4.7. Opis struktury potokowej

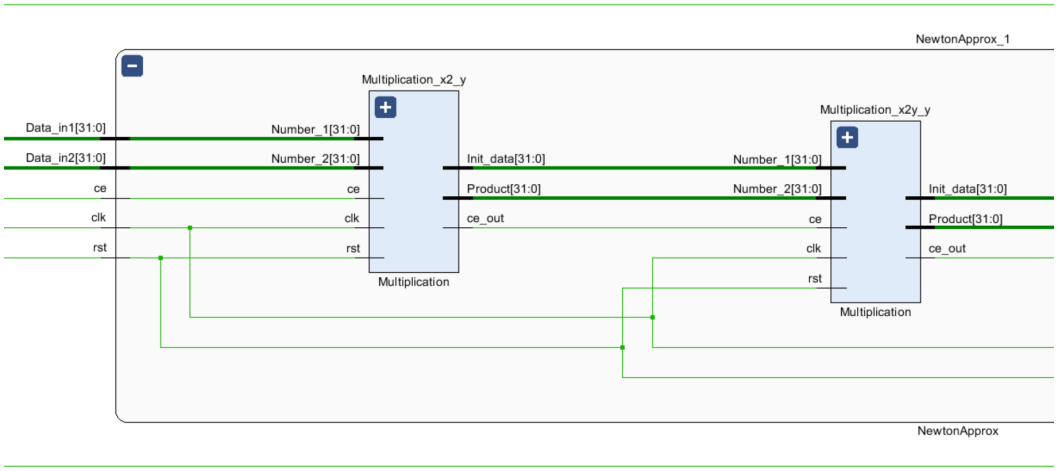
Układ został zaprojektowany w taki sposób, aby nowe dane wejściowe były pobierane do obliczeń co takt zegara. Pozwala to na bardzo szybką pracę modułu, pomijając warunki startowe algorytmu. Warunki startowe programu to 10 cykli zegara konieczne do przetworzenia danych przez wszystkie moduły w szeregu.

- Najdłuższą częścią algorytmu jest moduł odejmowania, na którym odkładane są 3 cykle zegarowe, ze względu na czasochłonną normalizację.
- Moduły mnożenia do pełnego przetworzenia danej wykorzystują 2 cykle
- Wstępne przeliczenie wartości żądanej trwa 1 cykl zegarowy.

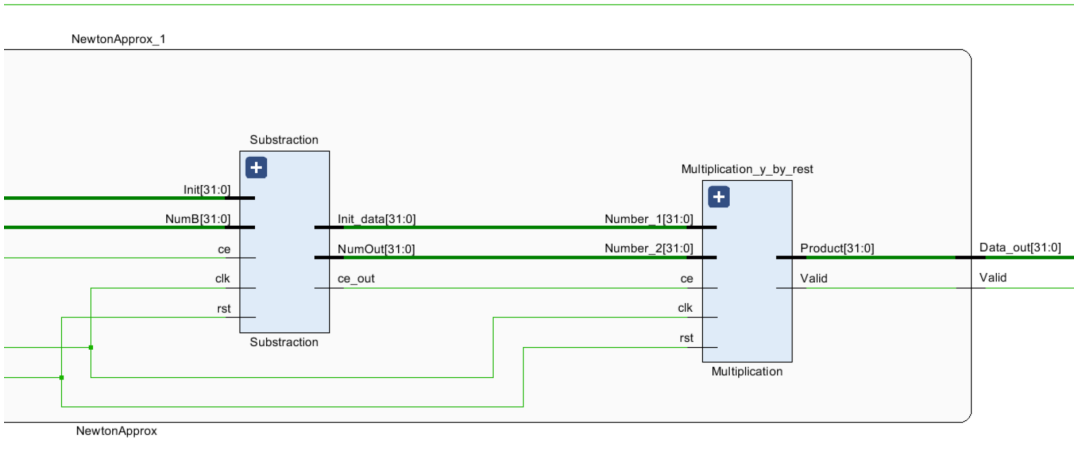
W działanie układu został zagnieżdżony sygnał CE, definiujący pracę **CLK** w modułach. Jest on propagowany zgodnie z działaniem potoku między modułami. Dzięki takiej implementacji w momencie gdy sygnał **CE** zostanie wyłączony, dane znajdujące się w potokowym przetwarzaniu zostaną obliczone. Po dokończeniu obliczeń sygnał **Valid** dla kolejnych *zablokowanych* już danych zostanie ustawiony na 0.



Rysunek 4.2: RTL część 1



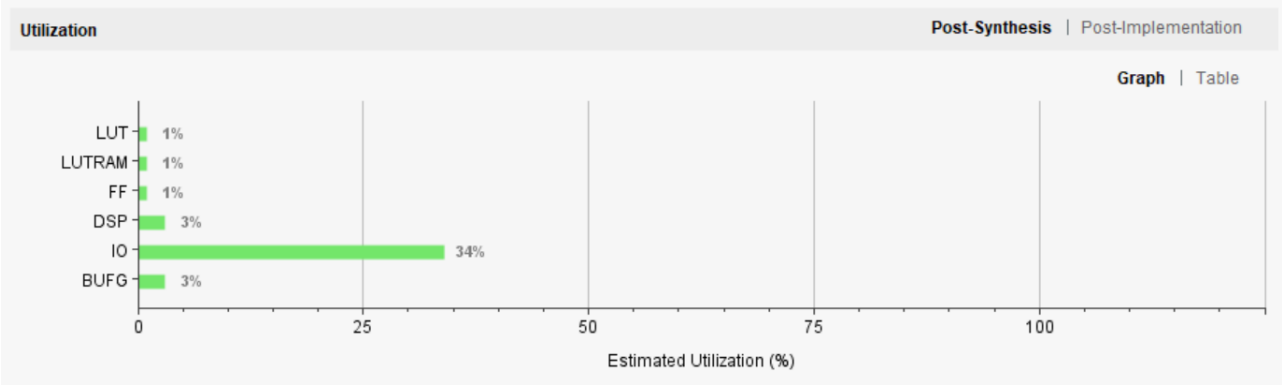
Rysunek 4.3: RTL część 2



Rysunek 4.4: RTL część 3

4.8. Prezentacja wyników syntezy

Algorytm nie wymaga stosowania dużej liczby modułów. Największą część zajmują DSP oraz BUFG (Rys. 4.5). Część dotycząca IO wskazuje na wysoką ilość wykorzystania, ale to z powodu braku danych w pliku constraints. Na cały algorytm używanych jest 6 bloków DSP 48 bitowych (Rys. 4.6). Są one wykorzystywane w operacji mnożenia 24 bitowych mantys dwóch liczb wejściowych, co jest konieczne dla poprawnego zaokrąglenia pośredniego iloczynu 48 bitowego.

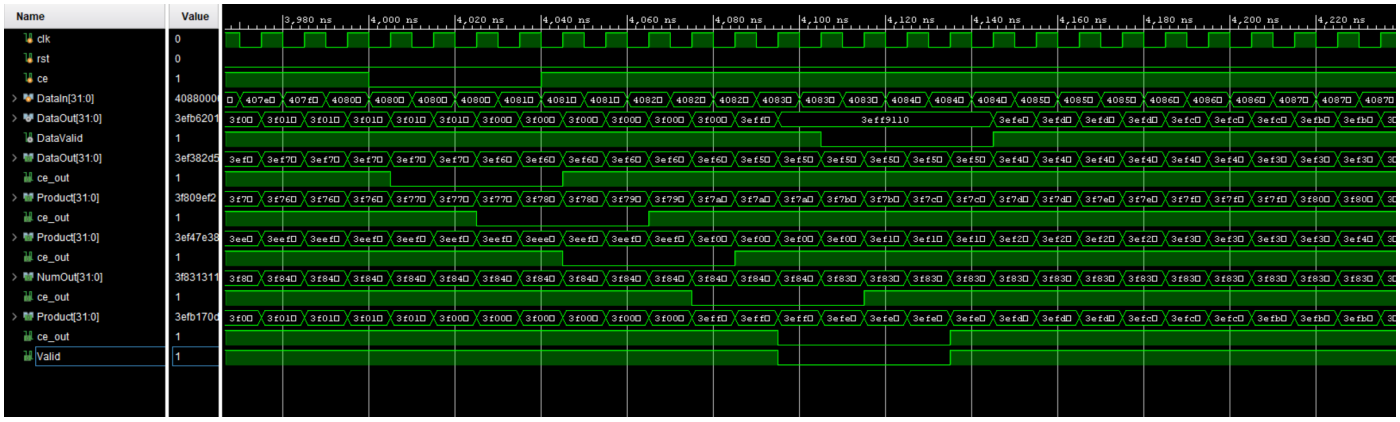


Rysunek 4.5: Zużyte zasoby FPGA

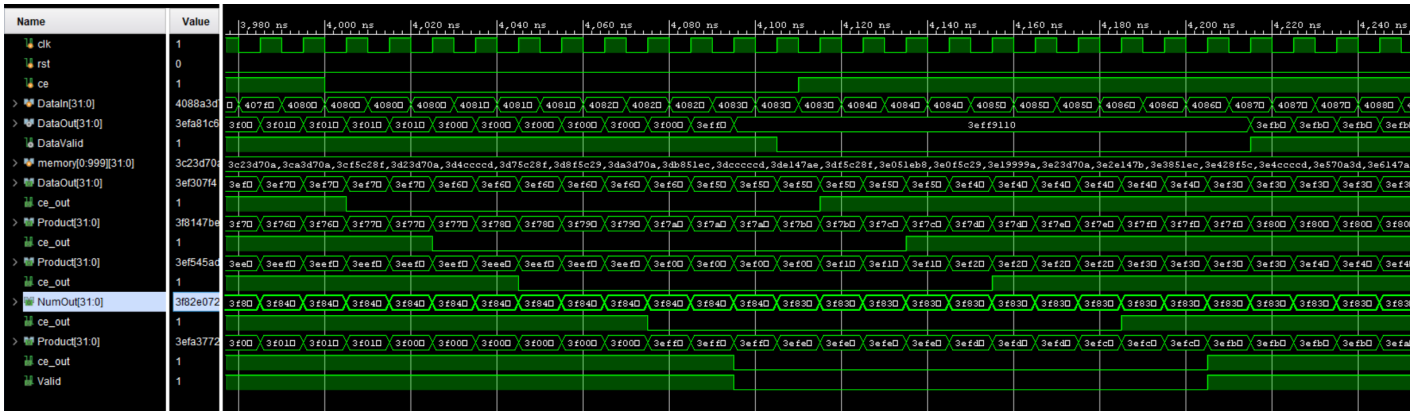
Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	DSPs (220)	Bonded IOB (200)	BUFGCTRL (32)
InvertSQRoot	382	353	1	6	68	1
Init_InvSQRoot (Init_InvSQRoot)	33	63	0	0	0	0
NewtonApprox_1 (NewtonApprox)	349	258	1	6	0	0
Multiplication_x2_y (Multiplication)	30	67	0	2	0	0
Multiplication_x2y_y (Multiplication_0)	107	39	0	2	0	0
Multiplication_y_by_rest (Multiplication_1)	36	40	0	2	0	0
Substraction (Substraction)	176	112	1	0	0	0

Rysunek 4.6: Użyte moduły FPGA

Podczas symulowania algorytmu były sprawdzane liczne przypadki potwierdzające poprawną pracę modułu. Główna część funkcjonalną stanowi sygnał CE (Clock Enable), który określa czas pracy modułów. W jednej z symulacji (Rys. 4.7) ustalono wyłączenie sygnału na 4 cykli zegara. Obserwuje się propagowanie sygnału CE oraz po pewnym czasie częściowa pracę algorytmu na końcu a także na początku modułu z wyłączoną częścią w środku. Natomiast w drugiej symulacji (Rys. 4.8) ustalono wyłączenie sygnału na 11 cykli zegara pokazując szeregowe zatrzymanie pracy algorytmu, jednocześnie zachowanie ciągłości danych wyjściowych i dokończenie obliczeń dla danych znajdujących się jeszcze w trakcie przetwarzania potokowego.



Rysunek 4.7: Wybrane wyniki symulacji

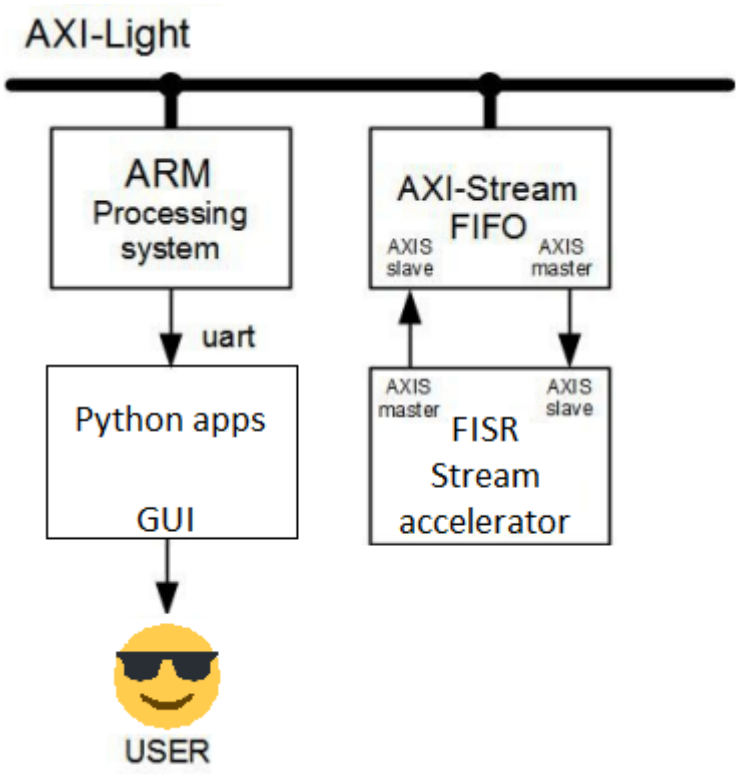


Rysunek 4.8: Wybrane wyniki symulacji

5. AXI-stream

W tej sekcji przedstawimy sprzętową implementację systemu procesora z wykorzystaniem potokowego akceleratora Fast Inverse Square Root.

System ma strukturę zgodną z przedstawioną na Rys. 5.1. Do połączenia akceleratora FISR z podsystemem procesora ARM użyto AXI-Stream FIFO. AXI-Stream wykorzystuje dwa buforowane FIFO: jedno do wysyłania danych i drugie do odbierania. Procesor ARM jest przystosowany do komunikacji z aplikacją na PC za pomocą lower level UART. Dla użytkownika przygotowano GUI, dzięki któremu może w sposób nieco bardziej praktyczny odczuć działanie algorytmu.



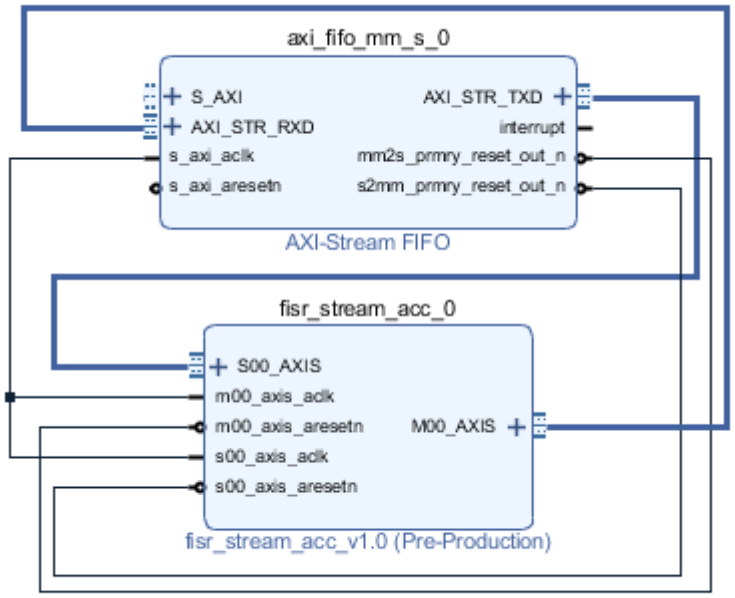
Rysunek 5.1: Implementacja sprzętowa (SDUP (2023))

5.1. IP repo

Przygotowano IP repo dla akceleratora FISR (*fisr_stream_acc_v1_0*). I zestawiono go z komórką *axi_fifo_mm_s_0* Mapowanie połączeń między portami instancji FISR stream a AXI-Stream FIFO przedstawione jest w Tab. 5.1, natomiast na Rys. 5.2 przedstawiono odzwierciedlenie tego połączenia.

Tabela 5.1: Mapowanie portów między AXI-Stream FIFO oraz fisr_stream_acc_v1_0

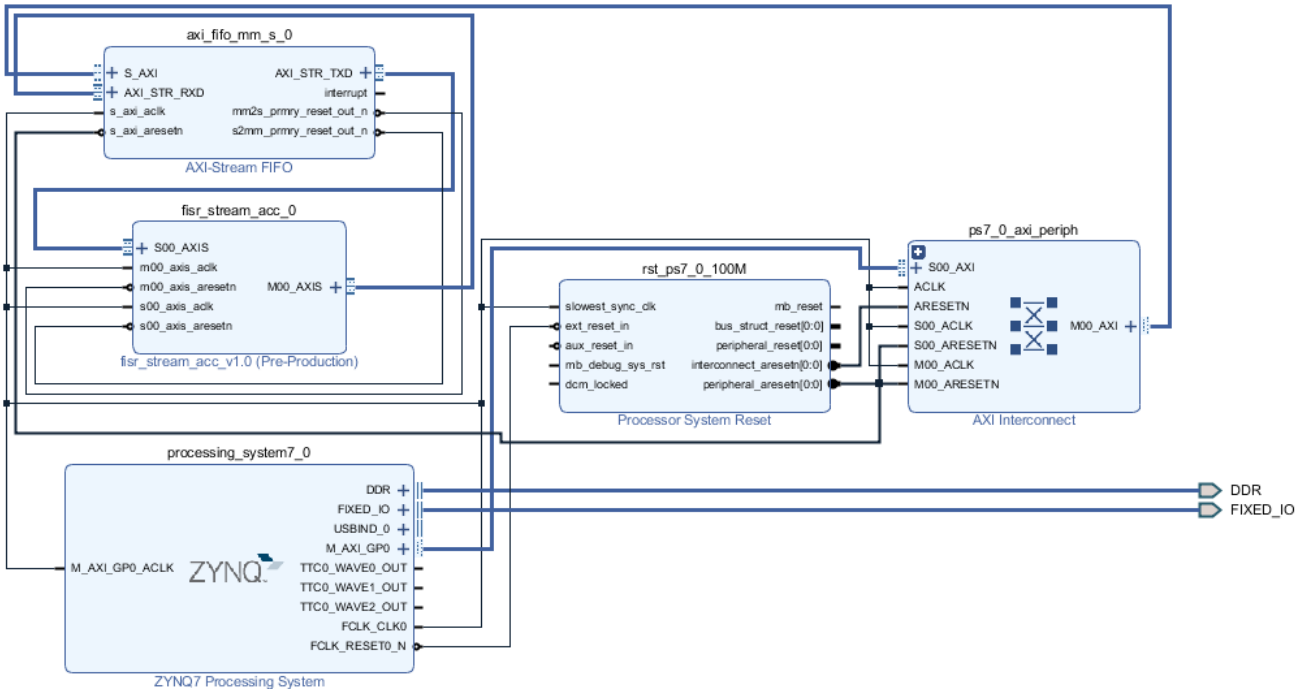
AXI-Stream FIFO		**FISR Stream acc**	
Nazwa portu	**Opis**	**Nazwa Portu**	**Opis**
AXI_STR_RXD	AXI-Stream odbiornik, SLAVE	M00_AXIS	AXI-Stream master
AXI_STR_TXD	AXI-Stream nadajnik, MASTER	S00_AXIS	AXI-Stream slave
s_axi_aclk	Sygnał zegarowy	m00_axis_aclk	zegar mastera
		s00_axis_aclk	zegar slave'a
m2mm_prmry_reset_out_n	Rst OUT do transmitera AXI-S	m00_axis_areset	Rst IN mastera AXI-S
s2mm_prmry_reset_out_n	Rst OUT do odbiornika AXI-S	s00_axis_areset	Rst IN slave'a AXI-S



Rysunek 5.2: Połączenie między AXI-Stream FIFO oraz fisr_stream_acc_v1_0

5.2. Design Wrapper

Pełne zestawienie bloków przedstawione zostało na Rys. 5.3. Wcześniej przygotowane połączenie AXI-Stream zostało dołączone do AXI-Light i skonfigurowane z procesorem ZYNQ.



Rysunek 5.3: Kompletne zestawienie wykorzystanych bloków

6. Uruchomienie na sprzęcie

6.1. Zaprogramowanie FPGA

Zaprogramowanie FPGA wykonuje się za pośrednictwem środowiska SDK używając złącza JTAG

6.2. PC <-> Zedboard - UART

Do komunikacji między komputerem a płytką rozwojową Zedboard wykorzystano interfejs UART. W środowisku SDK zaimplementowano niższy poziom obsługi UART (Kizheppatt (2020)) , który zapewnia większą niezawodność i umożliwia bardziej szczegółową kontrolę nad komunikacją niż UART w postaci terminala tekstowego.

```
1  inputData = malloc(sizeof(u8)*VectSize);
2  outputData8 = malloc(sizeof(u8)*VectSize);
3  myUartConfig = XUartPs_LookupConfig(XPAR_PS7_UART_1_DEVICE_ID
    );
4  status = XUartPs_CfgInitialize(&myUart, myUartConfig,
    myUartConfig->BaseAddress);
```

Listing 6.1: Inicjalizacja interfejsu UART

```
1  while(totalReceivedBytes < dataSize){
2    receivedBytes = XUartPs_Recv(&myUart,(u8*)&inputData[
    totalReceivedBytes],100);
3    totalReceivedBytes += receivedBytes;}
```

Listing 6.2: Odbieranie danych za pośrednictwem UART

```
1  while(totalTransmittedBytes < dataSize){
2    transmittedBytes = XUartPs_Send(&myUart, (u8*)&outputData8[
    totalTransmittedBytes], 4);
3    totalTransmittedBytes += transmittedBytes;
4    usleep(10);}
```

Listing 6.3: Wysyłanie danych za pośrednictwem UART

7. Zastosowanie algorytmu

Algorytm Fast Inverse Square Root (FISR) znajdował zastosowanie przede wszystkim w programowaniu graficznym i tworzeniu gier, gdzie optymalizuje obliczenia związane z oświetleniem, przekształceniami 3D oraz symulacjami fizycznymi. Dzięki przybliżonej wartości odwrotnego pierwiastka kwadratowego, FISR znacznie przyspiesza te obliczenia, redukując czas potrzebny na ich wykonanie.

Kolejne zastosowania tego algorytmu dotyczą również dziedzin naukowych i inżynieryjnych, gdzie istotne są szybkie obliczenia numeryczne. Może być wykorzystywany w przetwarzaniu sygnałów, analizie danych, symulacjach numerycznych. Fast Inverse Square Root zwiększa wydajność obliczeniową w tych dziedzinach.

Należy jednak pamiętać, że algorytm Fast Inverse Square Root jest przybliżony i może wprowadzać pewne błędy obliczeniowe. Dlatego stosuje się go tam, gdzie nie jest wymagana doskonała precyzja, a ważniejsza jest szybkość obliczeń. Należy jednak pamiętać, że zgodnie z opisem, który został umieszczony w sekcji 2.2 (oraz na Rys. 3.1a), można zwiększać dokładność algorytmu poprzez wykonywanie kolejnych korekcji Newtona-Raphsona. Wspomniany błąd obliczeniowy będzie przedstawiony w aplikacji w sekcji 7.2.

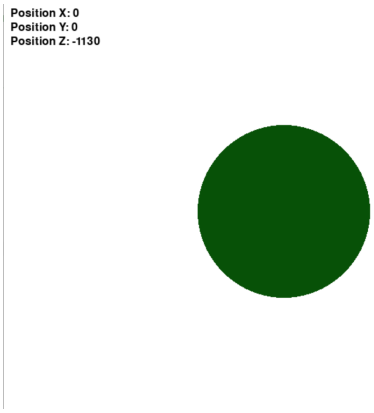
Dla dwóch pierwszych aplikacji wstawiono filmy prezentujące ich działanie - na repozytorium *Report_files/videos*.

7.1. Aplikacja FigureMoveIn3D

Jednym z zastosowań algorytmu Fast Inverse Square Root jest obliczanie odległości w przypadku przetwarzania obrazów, na przykład w grach lub przy dostosowywaniu oświetlenia obiektów. Dlatego funkcjonalność tej aplikacji obejmuje poruszanie się obiektu w 3D (Rys. 7.1).

Frontend: Możemy poruszać się obiektem w płaszczyźnie X i Z za pomocą strzałek, a do wykonania ruchu w płaszczyźnie Y (podskok) używamy spacji. Kolor obiektu staje się ciemniejszy wraz ze zmianą odległości od ekranu.

Backend: Aplikacja komunikuje się poprzez port szeregowy z procesorem ZYNQ, gdzie dane są przetwarzane przez FPGA i przesyłane z powrotem do aplikacji uruchomionej na komputerze.



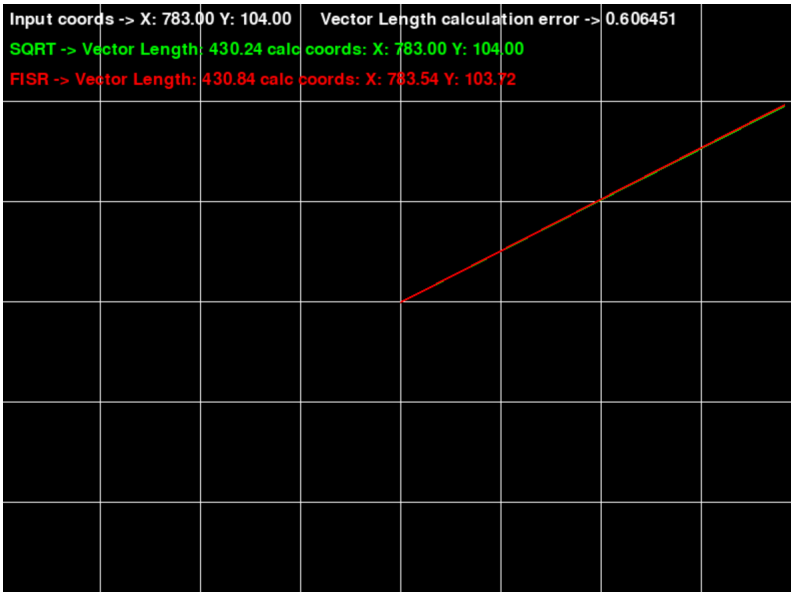
Rysunek 7.1: Zrzut ekranu z aplikacji FigureMoveIn3D

7.2. Aplikacja VectorMoving

Jest to aplikacja z interfejsem graficznym (GUI), która wykorzystuje funkcjonalność algorytmu FISR i w sposób praktyczny porównuje z wynikami funkcji *sqrt()* (Rys. 7.2).

Frontend: Użytkownik może kliknąć na konkretny punkt za pomocą myszy. Po kliknięciu zmieniają się wyświetlane wektory. Jeden z tych wektorów jest generowany przez algorytm FISR, a drugi jest obliczany za pomocą funkcji *sqrt()*. Użytkownik na bieżaco może obserwować punkt w który nacisną, długości wektorów oraz punktu nacisku wyliczone przez program.

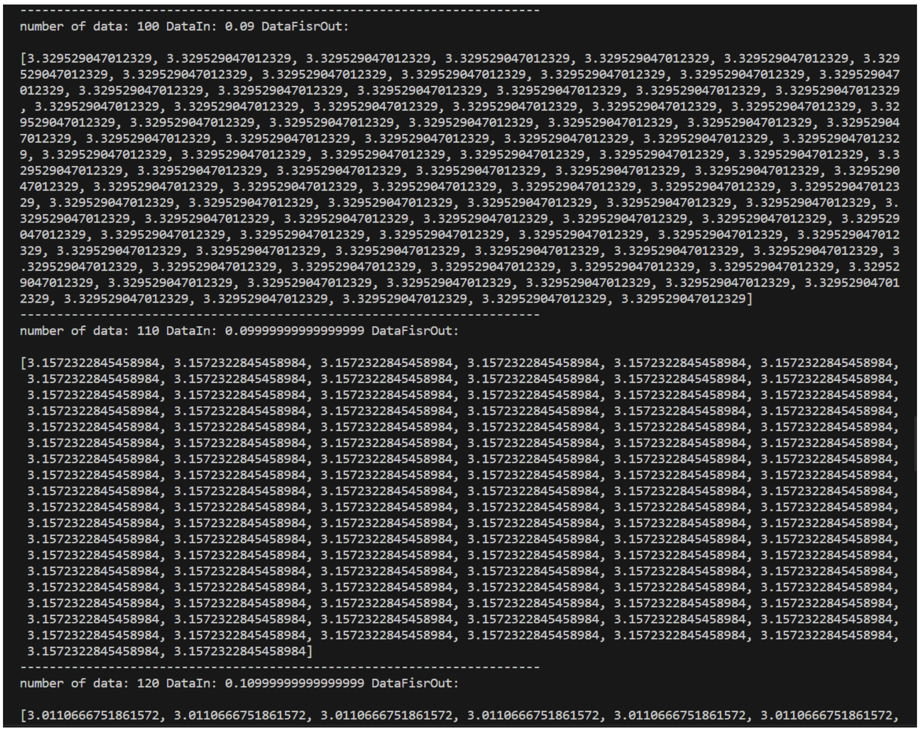
Backend: Na podstawie punktu wybranego przez kliknięcie myszki wyliczana jest długość wektora z dwóch algorytmów, wyznaczany jest również kąt wektora. Na podstawie tych danych wyznaczany jest punkt nacisku i rysowany wektor. Dzięki takiej funkcjonalności można zaobserwować błąd jaki wynika z sposobu działania algorytmu Fast Inverse Square Root.



Rysunek 7.2: Zrzut ekranu z aplikacji VectorMoved

7.3. Aplikacja CommunicationTest

Prosty skrypt sprawdzający poprawność działania komunikacji UART oraz przetwarzania danych przez FPGA. Z aplikacji (Rys. 7.3) wysyłane jest wiele danych do przetworzenia poprzez UART, a następnie odbierane są przeliczone wartości.



Rysunek 7.3: Zrzut ekranu z aplikacji UART_communication

Spis tabel

Spis rysunków

Kody programów

Bibliografia

Kizheppatt, V. (2020). *Data Transfer between PC and ZedBoard through UART Interface*. [Online video] https://www.youtube.com/watch?v=lzQ9hJ-wevg&t=1065s. (data dostępu: 12 czerwca 2023).

Nemean (2020). *Data Transfer between PC and ZedBoard through UART Interface*. [Online video] https://www.youtube.com/watch?v=p8u_k2LIZyo&t=65s. (data dostępu: 12 czerwca 2023).

SDUP (2023). Instrukcje do przedmiotu sdup. *AGH*.