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 ... 3
1. Model behawioralny algorytmu ... 6
1.1. Implementacja modelu w języku C ... 6
1.2. Testbench modelu behawioralnego w języku C ... 7
1.3. Prezentacja wyników ... 8
2. Model syntezowalny potokowy algorytmu ... 10
2.1. Układ przeliczenia wstępnego ... 10
2.2. Układ mnożący ... 10
2.3. Układ odejmujący ... 10
2.4. Implementacja kodu ... 11
2.5. Schemat blokowy ... 12
2.6. Testbench ... 13
3. AXI ... 15
3.1. Wrapper ... 15
4. Uruchomienie na sprzęcie ... 16
4.1. Test układu przez konsolę ... 16
5. Zastosowanie ... 17
5.1. Struktura blokowa ... 17
5.2. Komunikacja między PC, a Zedboard ... 17
5.3. Dostosowanie kodu na płytce Zedboard ... 17
5.4. Implementacja kodu z interakcją użytkownika ... 17
5.4.1. FigureMoveIn3D ... 17
5.4.2. VectorMoving ... 17
Spis tabel ... 18
Spis rysunków ... 19

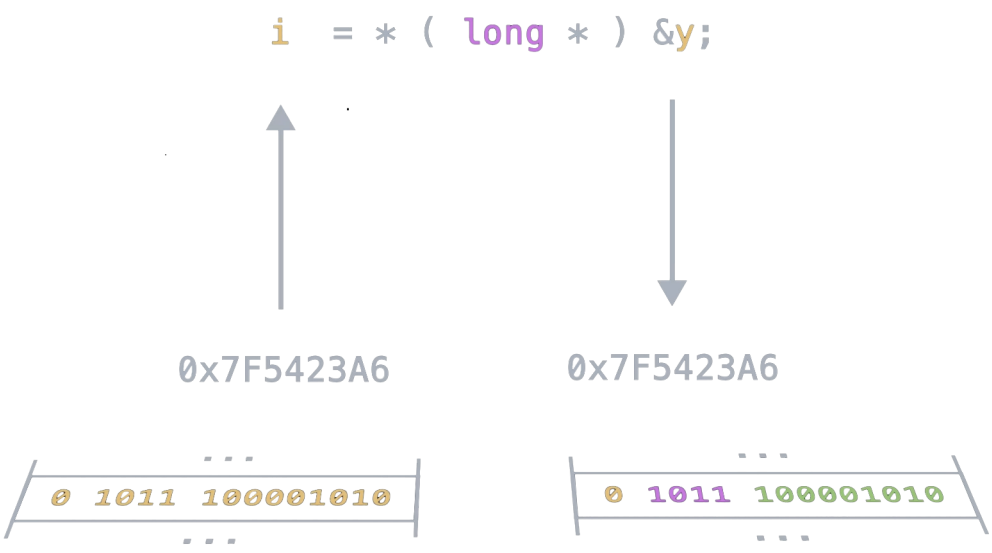
Wstęp

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 znalazł zastosowanie w przyspieszaniu 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. Podstawowy schemat działania algorytmu Fast Inverse Square Root można przedstawić w kilku krokach:

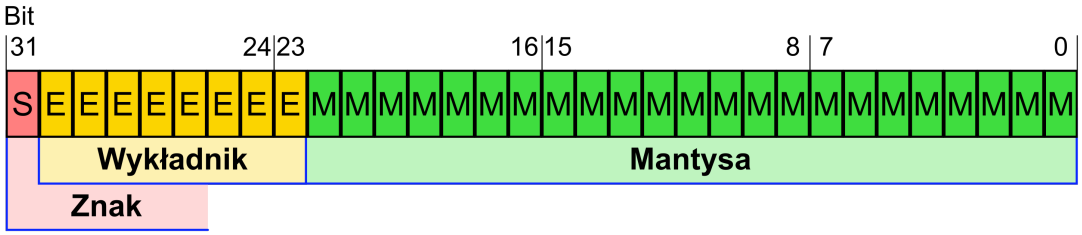
1. Wykorzystanie reprezentacji liczby zmiennoprzecinkowej i zapisanie jej ciąg bitów w zmiennej całkowitej (Rys. 1). Tutaj jest wykorzystywany adres, gdzie przechowywana jest zmienna i kopiowana jest zawartość do drugiej zmiennej. Dzięki temu możemy pracować na znanej nam liczbie jak na danym ciągu bitów.

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



Rysunek 1: Przeniesienie wartości standardu IEEE754 do zmiennej całkowitej

2. Zastosowanie magicznej liczby 0x5f3759df czyli pewnych operacji matematycznych i bitowych na tej reprezentacji (Rys. 2, które mają na celu przybliżone obliczenie wartości odwrotności pierwiastka kwadratowego. Wykorzystywany jest standard IEEE754, formalnie dzielący zmienną na 3 części - znak, eksponente i mantyse.



Rysunek 2: Standard IEEE754

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

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

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

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

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

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

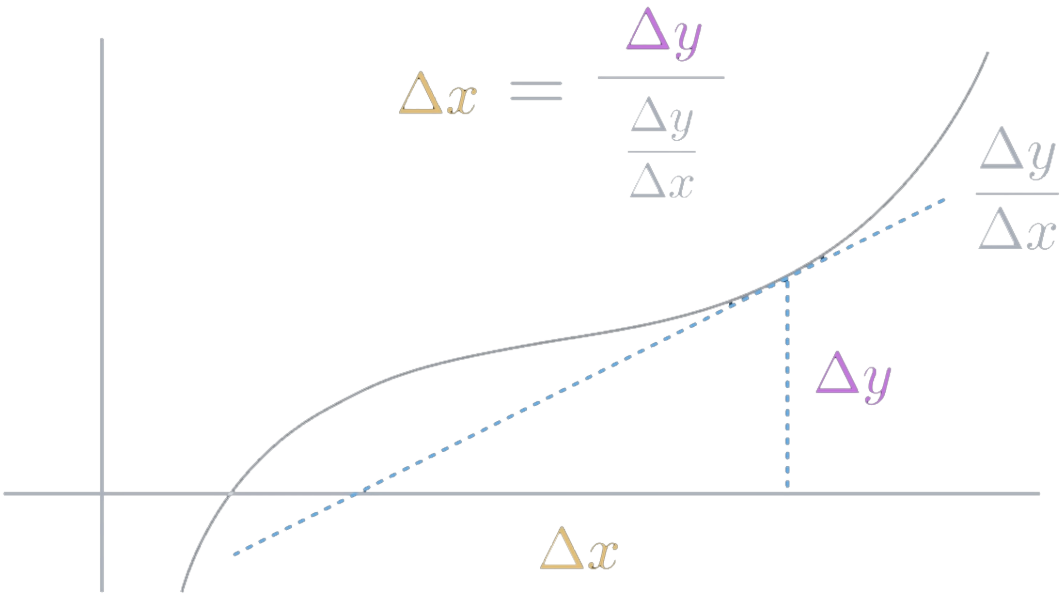
$$\frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127 = -\frac{1}{2}\left(\frac{1}{2^{23}}(M + 2^{23} * E) + \mu - 127\right)$$

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

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 (Rys. 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 3: Przybliżenie Newton-Raphson

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.

1. Model behawioralny algorytmu

1.1. 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;
}
```

1.2. 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");
        }
    }
}
```

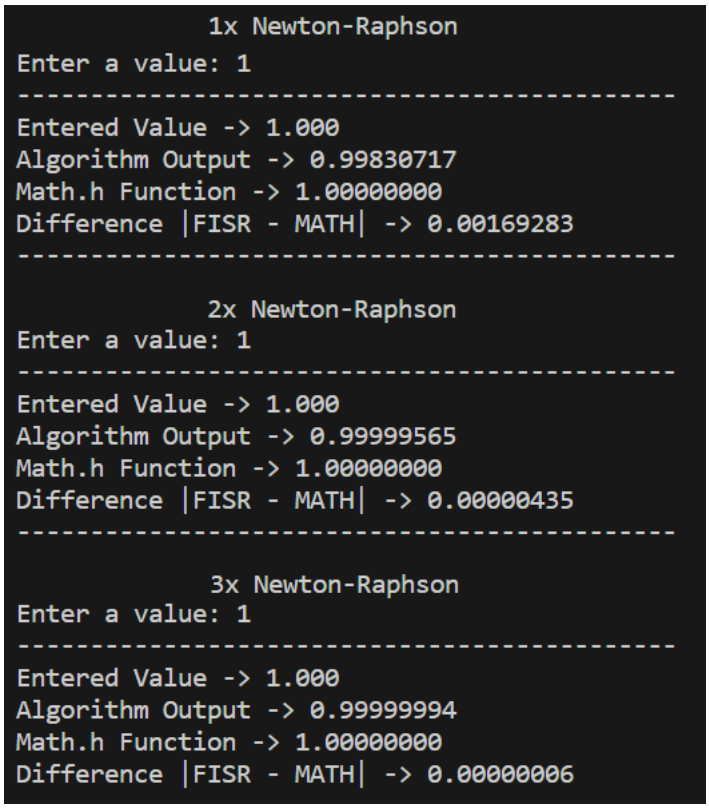
W tym kodzie znajduje się również funkcja odpowiedzialna za konwersję liczby wyjściowej z algorytmu na wartość *float*. Jej działanie jest zgodne z opisem przedstawionym na Rys. 1. Poniżej umieszczono kod funkcji *ieee754_to_float()*

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

1.3. 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. 1.1) Newtona-Raphsona, jednak zwiększa to czas wykonywania funkcji.

Kod jest zabezpieczony przed wprowadzaniem danych ujemnych oraz 0 (Rys. 1.2).



Rysunek 1.1: Prezentacja działania kodu

```
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
-----------------------------------------
```

Rysunek 1.2: Prezentacja działania kodu

2. Model syntezowalny potokowy algorytmu

2.1. Układ przeliczenia wstępnego

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

2.2. Układ mnożący

```
E_Square_nxt = Number_1[30:23] + Number_2[30:23] - 127;
M_Square_nxt = ( {1'b1, Number_1[22:0]} * {1'b1, Number_2[22:0
   ]} );

Product_nxt = {Sign, E_Square + M_Square[47], ( M_Square[47] ?
   M_Square[46:24] : M_Square[45:23] )};
```

2.3. Układ odejmujący

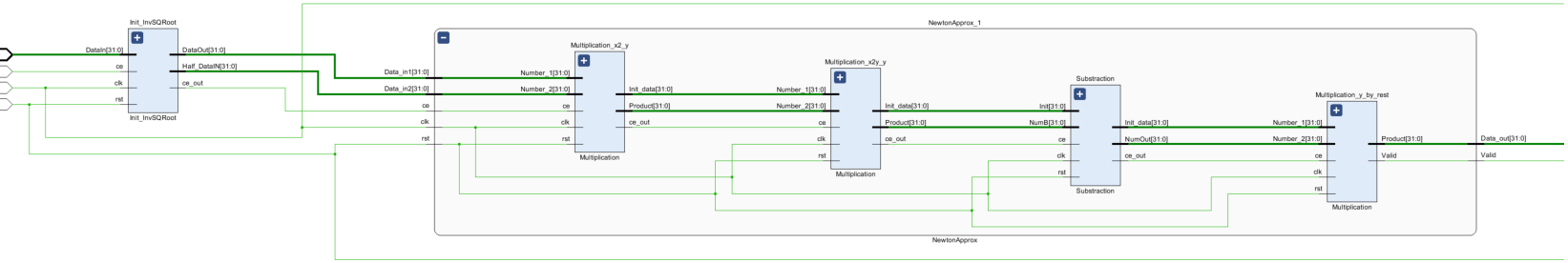
```
Sub_mantissa_nxt = ( {1'b1, OneAndHalf[22:0]} ) - ( {1'b1, NumB
   [22:0]} >> (OneAndHalf[30:23] - NumB[30:23]) );

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]};
```

2.4. Implementacja kodu

2.5. Schemat blokowy



Rysunek 2.1: RTL

2.6. Testbench

```
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
//////////////////////////////////////
```

```
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
//////////////////////////////////////////
```

3. AXI

3.1. Wrapper

4. Uruchomienie na sprzęcie

4.1. Test układu przez konsolę

5. Zastosowanie

5.1. Struktura blokowa

5.2. Komunikacja między PC, a Zedboard

5.3. Dostosowanie kodu na płytce Zedboard

5.4. Implementacja kodu z interakcją użytkownika

5.4.1. FigureMoveIn3D

Jednym z zastosowań algorytmu Fast Inverse Square Root jest obliczanie odległości przy przetwarzaniu obrazów np. w grach, czy w przypadku dobierania oświetlenia dla obiektów. Dlatego funkcjonalności aplikacji to przemieszczanie w 3D obiektu.

Strzałkami poruszamy się w płaszczyźnie X oraz Z, natomiast za pomocą spacji wykonujemy ruch w płaszczyźnie Y (podskok). Wraz ze zmianą odległości od ekranu kolor obiektu staje się ciemniejszy.

5.4.2. VectorMoving

Spis tabel

Spis rysunków

1 Przeniesienie wartości standardu IEEE754 do zmiennej całkowitej 3
2 Standard IEEE754 4
3 Przybliżenie Newton-Raphson 5

1.1 Prezentacja działania kodu 8
1.2 Prezentacja działania kodu 9

2.1 RTL 12