Wydajne algorytmy obliczania modulo

Implementacja wybranych algorytmów modulo na podstawie pracy "Efficient Hardware Operations for the Residue Number System by Boolean Minimization"

Autorzy sprawozdania: Michał Dziedziak 263901, Joachim Szewior 263943
Imię i Nazwisko prowadzącego kurs: Dr inż. Piotr Patronik
Dzień i godzina zajęć: Poniedziałek, 17:05 - 18:45 TN

Spis treści

Spis rysunków

1 Wstęp

1.1 Tematyka Efficient Hardware Operations for the Residue Number System by Boolean Minimization

Praca badawcza skupia się na optymalizacji sprzętowych operacji dzięki Systemowi Liczb Resztkowych (RNS) poprzez minimalizację funkcji logicznych. Przykładem zastosowania RNS są systemy przeciwlotnicze, obliczenia neuronowe, przetwarzanie sygnałów oraz kryptografia. Idea RNS pochodzi ze starożytnych Chin, jednak popularność zyskała w 19 wieku dzięki Gaussowi. Autorzy pracy [1] proponują wykorzystanie metody boolowskiej do minimalizacji funkcji logicznych, pozwala znacznie zwiększyć znacznie wydajność sprzętu oraz zwiększyć niezawodność i odporność na zakłócenia. Przedstawione eksperymentalne wyniki potwierdzają skuteczność tych technik, co ma duże znaczenie dla przyszłego rozwoju sprzętu opartego na RNS.

1.2 Cel projektu

Celem projektu było zbadanie i zaimplementowanie wybranych rozwiązań proponowanych w powyższej pracy. Zaimplementowane zostały:

- Algorytm modulo dla dowolnych dwóch liczb w pythonie.
- Algorytm modulo w Verilogu dla dowolnej liczby modulo 129.
- Mnożenie modulo dwóch liczb w pythonie.

2 Algorytm modulo

2.1 Opis algorytmu

Algorytm składa się z dwóch kroków:

1. Dzielimy X na k podwektorów, w którym każdy podwektor ma długość δ (równą długości binarnej P), gdzie:

$$\delta = \lceil \log P \rceil$$

2. Powstałe podwektory łączymy zgodnie z równaniem

$$X(modP) = \sum_{i=1}^{k} X_i * (2^{\delta(i-1)}(modP))$$

gdzie,

- P-wartość modulo
- δ - długość podwektora
- X_i - i'ty podwektor

2.2 Implementacja w Pythonie

2.2.1 Założenia i cele

Celem napisania algorytmu modulo w języku python było zbadanie ilości iteracji występujących podczas trwania algorytmu. Pozwala to zbadać maksymalną wartość dla dowolnego x oraz p, co ułatwiło napisane tego algorytmu w Verilogu. Dodatkowo umożliwia to zrobienie wykresów, potrzebnych do analizy algorytmu.

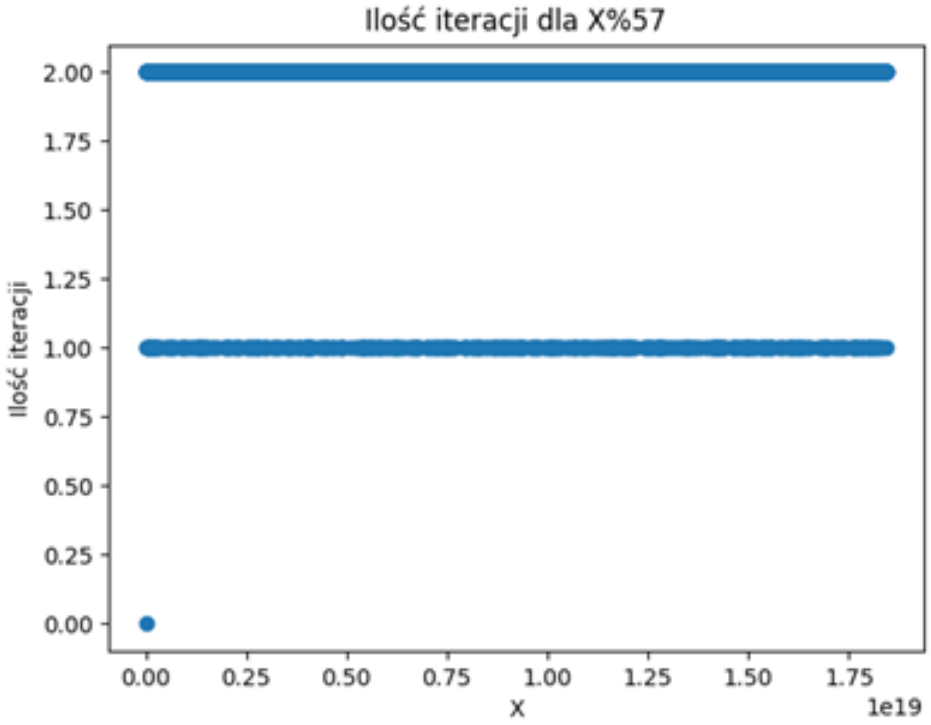
2.2.2 Badanie zależności dla ilości powtórzeń pętli

Wykresy dla stałych P

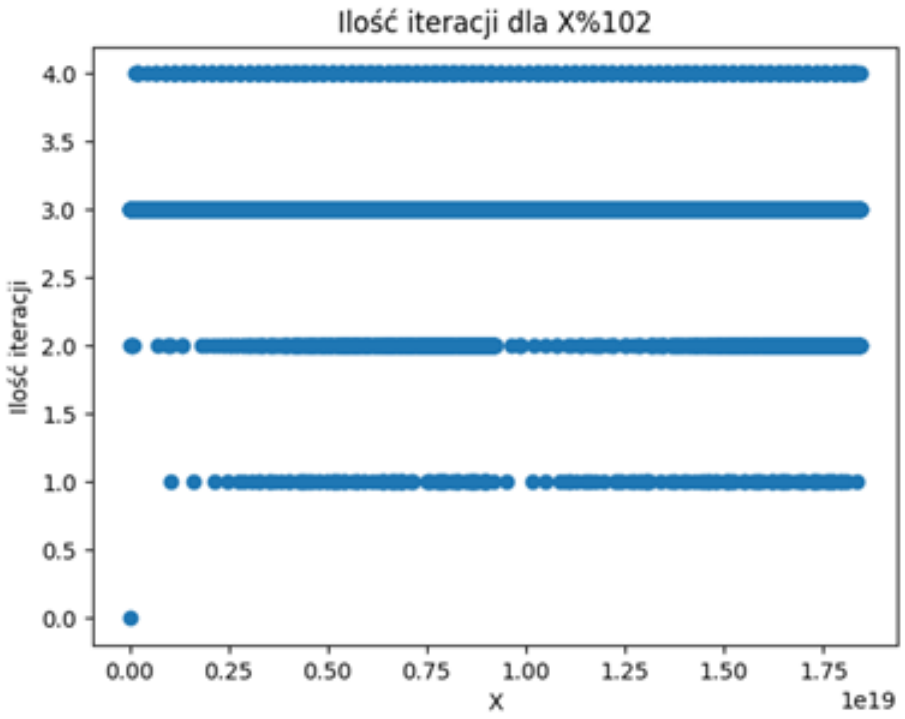
W celu głębszego zbadania ilości iteracji pętli w zależności od X oraz P postanowiliśmy przetestować nasz algorytm dla dowolnych liczb z wybranych zakresów:

- X z zakresu od 0 do $2^{64} - 1$. Czyli od 1 do 64 bitów
- P z zakresu od 2 do $2^{32} - 1$. Czyli od 2 do 32 bitów

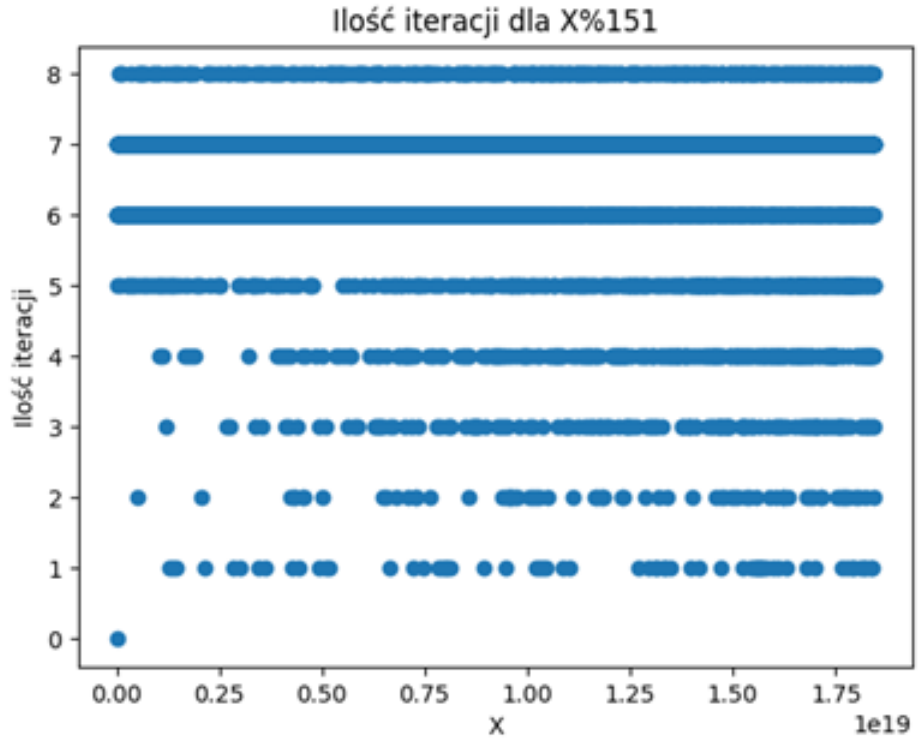
Następnie nasze wyniki zamieniliśmy w wykresy. Wybrane wykresy poniżej.



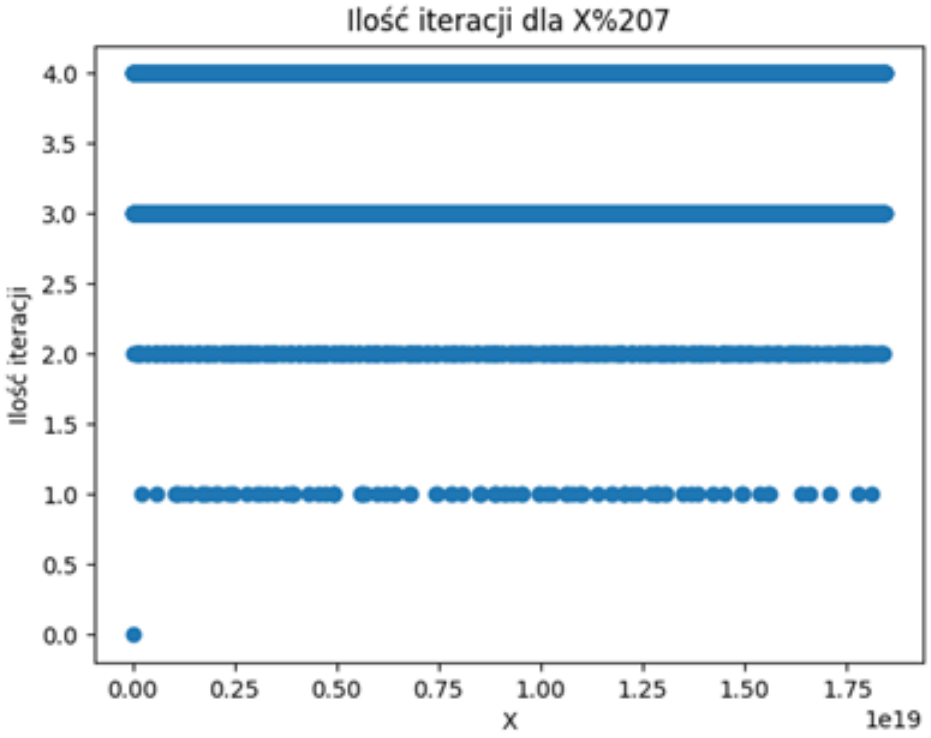
Rysunek 1: Wykres ilości iteracji od wartości X dla P = 57 (długość binarna P: 6)



Rysunek 2: Wykres ilości iteracji od wartości X dla P = 102 (długość binarna P: 7)



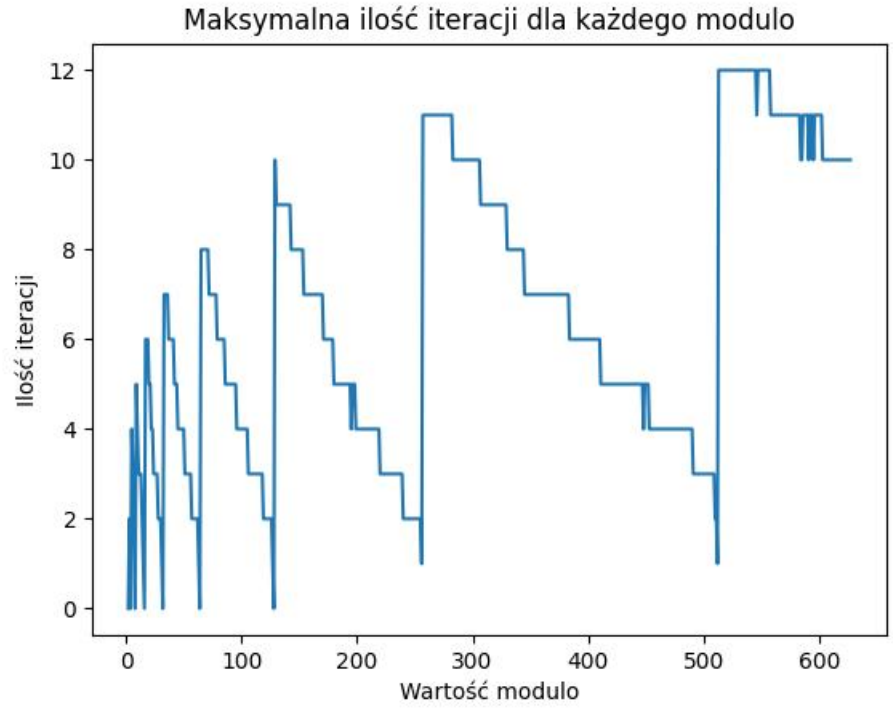
Rysunek 3: Wykres ilości iteracji od wartości X dla P = 151 (długość binarna P: 8)



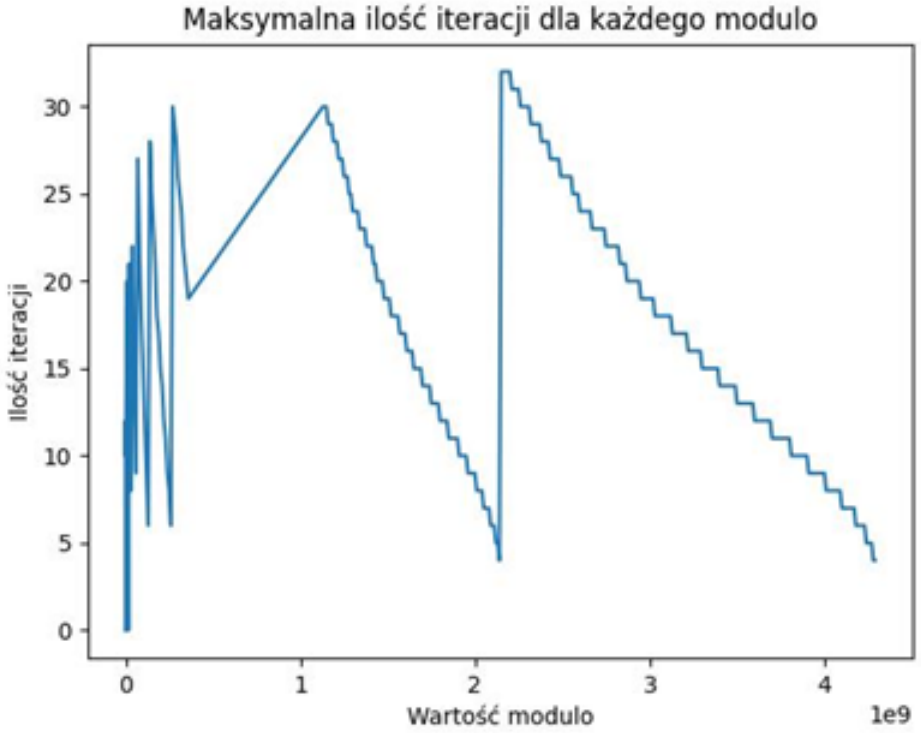
Rysunek 4: Wykres ilości iteracji od wartości X dla P = 207 (długośc binarna P: 8)

Wykresy zależności maksymalnej ilości powtórzeń pętli od P

W trakcie badań zauważyliśmy, że nie istnieje bezpośrednie powiązanie pomiędzy ilością iteracji pętli a wartością X dla stałej wartości modulo. Dlatego też postanowiliśmy zbadać maksymalną ilość iteracji dla danej wartości modulo dla dowolnego X. Doprowadziło nas to do zauważenia pewnej zależności. Maksymalna ilość iteracji w zależności od wartości P co określony interwał dynamicznie przebija wcześniejsze maksimum, następnie schodkowo spada, aby za chwilę osiągnąć nowe maksimum. (zakładamy, że jest to związany z szerokością bitową wartości modulo).



Rysunek 5: Wykres przedstawiający maksymalną, wyznaczoną ilość iteracji pętli dla każdego P z zakresu od 0 do 628 (długośc binarna P: od 1 do 10)



Rysunek 6: Wykres przedstawiający maksymalną, wyznaczoną ilość iteracji pętli dla każdego P z zakresu od 0 do $2^{32} - 1$ (długośc binarna P: od 1 do 32)

Wnioski

Analizując wykresy, doszliśmy do wniosku, że ilość iteracji pętli zależy od rozmiaru zmiennej p. Na pokazanych wykresach widać, że ilość iteracji pętli skokowo maleje wraz ze wzrostem wartości modulo, po to aby raz na pewien okres skoczyć w górę (zakładamy, że jest to związane z długością bitową wartości modulo). Dodatkowo wykresy potwierdziły nasze założenia, że algorytm modulo jest szybki dla dużych wartości liczbowych.

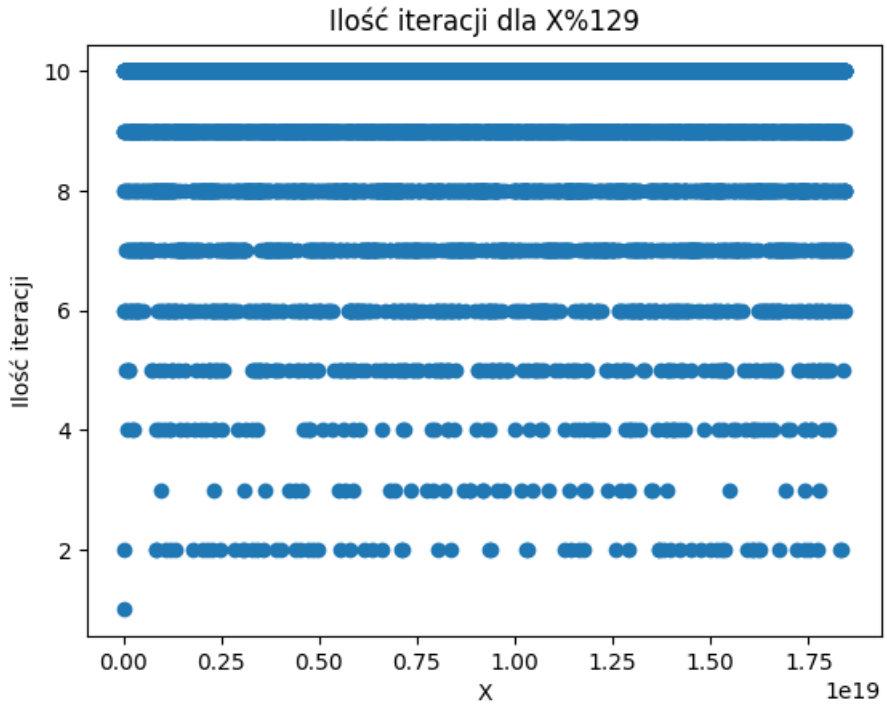
2.3 Implementacja w Verilogu dla mod129

2.3.1 Założenia

Dla wybranego P równego 129 zaimplementowaliśmy algorytm modulo w języku opisu sprzętu, Verilog.

2.3.2 Wyznaczona ilość iteracji dla P = 129

Dla P równego 129 wyznaczyliśmy maksymalnie 10 powtórzeń pętli. Mając na uwadze to, że nasz algorytm oblicza pierwsze S_temp poza pętlą to łączna, potrzebna ilość S_temp jest równa 11. Nasz kod w Verilogu korzysta z możliwości automatycznego powielania (generowania) układu zadaną ilość razy, więc dla wybranego przez nas modulo parametr ten ustawiony jest na 11.



Rysunek 7: Wykres ilości iteracji od wartości X dla P = 129

3 Algorytm mnożenia modularnego

3.1 Opis algorytmu

Algorytm poczatkowo dzieli mnożną i mnożnik na 4, 3 lub 2 bitowe wektory. Następnie kolejne wektory są mnożone ze sobą i stałą zależną od P. Wynik algorytmu to suma mnożeń podwektorów.

Przedstawiony algorytm można zapisać wzorem:

$$R = \sum_{i=1}^{\delta} \sum_{j=1}^{\delta} A_i * B_i * (2^{m*(i+j-2)*l}(modP)) = S_temp$$

gdzie:

- δ - ilość podwektorów
- A_i i B_j - podwektor A i B
- l - długość binarna podwektorów A i B

4 Wnioski

Podczas naszych badań zapoznaliśmy się z operacjami modularnymi, w szczególności operacją modulo oraz mnożeniem modularnym. Przeprowadziliśmy implementację tych operacji, co pozwoliło nam na lepsze zrozumienie ich działania i zastosowań.

Operacja modulo, oznaczana symbolem "%", oblicza resztę z dzielenia jednej liczby przez drugą. W naszym przypadku algorytm dzieli liczby na podwektory co przyśpiesza działanie algorytmu modulo. Jest to bardzo pożądana cecha programów w obecnych czasach.

Mnożenie modularne polega na obliczeniu iloczynu dwóch liczb, a następnie zastosowaniu operacji modulo do wyniku. Było to nasze pierwsze spotkanie z takim podejściem, jednakże udało nam się poprawnie zaimplementować algorytm.

Podsumowując, nasze badania skupiły się na poznaniu, przetestowaniu oraz zbadaniu operacji modularnych, takich jak operacja modulo i mnożenie modularne.

Ostatecznie możemy powiedzieć, że praca została pomyślnie zakończona.

Literatura

[1] Danila Gorodecky and Tiziano Villa, *Efficient Hardware Operations for the Residue Number System by Boolean Minimization*, January 2020.

5 Kod programów

5.1 Kod algorytmu obliczania modulo i mnożenia modularnego w Pythonie

Projekt w Pythonie składa się z kilku klas.

5.1.1 main.py

```
import math
import os
import time
from ModuloAlgorithm import modulo_computation_algorithm
from ModularProductAlgorithm import calculate_modular_product
```

```python
# Function to save results in a file
def save_in_file(x, p, counter):
    directory_path = "Data/"
    f = open(directory_path + f'mod_{p}.csv', "a")
    f.write(f'{x};{p};{counter}\n')
    f.close()


# Function to save results in a file with a specific name
def save_in_file_with_name(x, p, counter, file_name):
    f = open(file_name, "a")
    f.write(f'{x};{p};{counter}\n')
    f.close()


# Function to perform calculations for a range of values (will save results in files)
def perform_calculations(x_max, x_min, x_step, p_max, p_min, p_step):
    max_iterations = -1
    directory_path = "Data/"
    max_file_path = "current_max.csv"
    if os.path.exists(directory_path + max_file_path):
        f = open(directory_path + max_file_path, "r")
        content = f.read()
        lines = content.split("\n")
        lines.pop()
        val_str = lines.pop().split(";")[2]
        max_iterations = int(val_str)

    for p in range(p_min, p_max + 1, p_step):
        if os.path.exists(directory_path + f'mod_{p}.csv'):
            os.remove(directory_path + f'mod_{p}.csv')
        start = time.time()
        for x in range(x_min, x_max + 1, x_step):
            if x % (x_step * 100) == 0:
                print(f'p = {p} | x = {x} | perc = {(x*100/x_max):0.15f}%')
            x_mod_p, counter = modulo_computation_algorithm(x, p)
            save_in_file(x, p, counter)
            if counter > max_iterations:
                max_iterations = counter
                save_in_file_with_name(x,p,counter,directory_path + max_file_path)
            if x_mod_p != x % p:
                print("ERROR!")
        end = time.time()
        minutes = int(math.floor((end-start)/60))
        seconds = int(end - start - minutes*60)
        print(f'Done for mod{p} in {minutes} min {seconds} sec\n')


# Entry point of the program
if __name__ == '__main__':
    a = 45
    b = 15
    p = 47
    res = calculate_modular_product(a, b, p)
    print(f"{a}*{b}(mod{p}) = {a * b % p}")
    print(f"algh res = {res}")
```

```python
x = 2721979
p = 129
res, iterations = modulo_computation_algorithm(x, p)
print("\nRESULT:")
print(f"{x}(mod{p}) = {x%p}")
print(f"Algorithm result = {res} | iterations amount = {iterations}")

print("BEGINS...")
p_min = 2
p_step = 16_777_217
p_32 = 4_294_967_295
x_min = 0
x_step = 1_000_000_000_003
x_64 = bin_to_tens([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
perform_calculations(x_64, x_min, x_step, 255, 255, 1)
print('END')
```

5.1.2 NumbersManipulations.py

```python
import math
import numpy as np


# Function to convert a decimal number to a binary array
def tens_to_bin(num):
    bits_arr = [int(i) for i in list('{0:0b}'.format(num))]
    return bits_arr


# Function to convert a binary array to a decimal number
def bin_to_tens(bits_arr):
    res = 0
    base = 0
    for bit in reversed(bits_arr):
        res = res + bit * pow(2, base)
        base = base + 1
    return int(res)


# Function to split a bit array into subvectors
def split_bit_array_into_subvectors(bits_array, subvectors_amount):
    result_array = np.array_split(bits_array, subvectors_amount)
    return result_array


# Function to extend a binary array with zeros (zeros are added to beginning)
def extend_bin_with_zeros(bits_arr, zeros_amount):
    zeros_to_add = []
    for i in range(zeros_amount):
        zeros_to_add.append(0)
    zeros_to_add.extend(bits_arr)
    return zeros_to_add
```

5.1.3 ModuloAlgorithm.py

```
# funkcja do obliczenia s_temp
def calculate_s(bits_arr, p, r, k):
    s = 0
    i = 1
    for i in range(1, k + 1):
        sub_X = bits_arr.pop() # zabranie podwektora z listy
        sub_x = int(bin_to_tens(sub_X)) # zamiana na system dziesiętny
        cons = int(pow(2, r * (i - 1))) % p # obliczenie stałej zależnej od P
        s = s + sub_x * cons # dodanie do sumy
    return int(s) # zwrócenie sumy


# Funkcja do szybkiego liczenia modulo dużych liczb
def modulo_computation_algorithm(x, p):
    X = tens_to_bin(x)  # zamiana x na system binarny
    r = int(math.ceil(math.log2(p)))  # długość p w bitach
    k = int(math.ceil(len(X) / r))  # liczbę podwektorów x

    # jeżeli długość binarna X jest mniejsza niż k*r to rozszerzamy X zerami
    if len(X) < k * r:
        X = extend_bin_with_zeros(X, k * r - len(X))

    X = split_bit_array_into_subvectors(X, k) # Dzielimy X na podwektory (nadpisujemy zmienną X)

    # obliczamy pierwszy s_temp
    s1 = calculate_s(X, p, r, k)

    s_temp = s1

    # pomocnicza zmienna do liczenia ilości iteracji pętli
    loop_counter = 1

    while s_temp >= 2 * p:
        loop_counter = loop_counter + 1

        # zamieniamy obecny (zapisany dziesiętnie) s_temp na zapis binarny
        S_temp = tens_to_bin(s_temp)

        # obliczamy długość S_temp
        n_temp = len(S_temp)

        # obliczamy liczbę wektorów
        k_temp = int(math.ceil(n_temp / r))

        # uzupełniamy S_temp zerami jeśli jest za malo
        if len(S_temp) < k_temp * r:
            S_temp = extend_bin_with_zeros(S_temp, k_temp * r - len(S_temp))

        # dzielimy na podwektory
        S_temp = split_bit_array_into_subvectors(S_temp, k_temp)

        # ponownie obliczamy s
        s_temp = calculate_s(bits_arr=S_temp, p=p, r=r, k=k_temp)

    # jeżeli modulo jest mniejsze równe s_temp to zwraca s_temp - p oraz ilość iteracji
    if p <= s_temp:
```

```
            return s_temp - p, loop_counter
        # w przeciwnym przypadku zwracamy s_temp oraz ilość iteracji
        else:
            return s_temp, loop_counter
```

5.1.4 ModularProductAlgorithm.py

```
    def calculate_modular_product(a: int, b: int, p: int):
        #zamieniamy wejściowe liczby na zapis binarny
        a_bin = tens_to_bin(a)
        b_bin = tens_to_bin(b)

        #ustawiamy, aby miały te same długości
        if len(a_bin) < len(b_bin):
            a_bin = extend_bin_with_zeros(a_bin, len(b_bin) - len(a_bin))
        elif len(b_bin) < len(a_bin):
            b_bin = extend_bin_with_zeros(b_bin, len(a_bin) - len(b_bin))


        #deklarujemy rozmiar podwektorów i ich ilość na -1
        bits_per_subvector = -1
        subvectors_amount = -1

        #ustawiamy ilość bitów na podwektor i ich ilość
        #dopóki długość binarna zmiennej a_bin nie dzieli się przez 4,3,2 dodajemy zera na początek
        while bits_per_subvector < 0:
            if len(a_bin) % 4 == 0:
                bits_per_subvector = 4
                subvectors_amount = len(a_bin)/bits_per_subvector
            elif len(a_bin) % 3 == 0:
                bits_per_subvector = 3
                subvectors_amount = len(a_bin)/bits_per_subvector
            elif len(a_bin) % 2 == 0:
                bits_per_subvector = 2
                subvectors_amount = len(a_bin)/bits_per_subvector
            else:
                a_bin = extend_bin_with_zeros(a_bin, 1)
                b_bin = extend_bin_with_zeros(b_bin, 1)

        #dzielimy liczbę a i b na podwektory
        a_subvectors = split_bit_array_into_subvectors(a_bin, subvectors_amount)
        b_subvectors = split_bit_array_into_subvectors(b_bin, subvectors_amount)

        #zamieniamy kolejność podwektorów w listach (najbardziej znaczące bity na koniec)
        a_subvectors.reverse()
        b_subvectors.reverse()

        result = 0
        a_index = 0

        #dla każdego pojedyńczego podwektora a
        for a_sub in a_subvectors:

            #zwiększamy indeks a i zerujemy indeks b
            a_index = a_index + 1
            b_index = 0
```

```
        # dla każdego pojedyńczego podwektora b
        for b_sub in b_subvectors:

            #zwiększamy indeks b
            b_index = b_index + 1
            #obliczamy do której potęgi trzeba podnieść
            pow_to_raise = (a_index + b_index - 2) * bits_per_subvector

            cons = int(pow(2, pow_to_raise))
            s_tmp = bin_to_tens(a_sub) * bin_to_tens(b_sub) * cons
            s_tmp = s_tmp % p
            result = result + s_tmp

    #zwracamy obliczone zmienne (skorygowane o algorytm modulo)
    result, iterations = modulo(result, p)
    return result
```

5.2 Kod algorytmu obliczania modulo dla P = 129 w Verilogu

```
// Moduł do obliczania S_temp
module sub_S_calculation
(
    input [64:1] S_in,
    output [64:1] S_out
);

    // Mnożenie kolejnych sum cząstkowych ze stałymi.
    assign S_out =
        S_in[8:1  ] * 1 +       // 2^0 = 1
        S_in[16:9 ] * 127 +     //2^[8]mod129 = 127 (0111 1111)
        S_in[24:17] * 4 +       //2^16(mod129) = 4
        S_in[32:25] * 121 +     //2^24(mod129) = 121
        S_in[40:33] * 16 +      //2^32(mod129) = 16
        S_in[48:41] * 97 +      //2^40(mod129) = 97
        S_in[56:49] * 64 +      //2^48(mod129) = 64
        S_in[64:57] * 1;        //2^56(mod129) = 1

endmodule //sub_S_calculation

// Moduł do obliczania Xmod129 przedstawionym algorytmem
module x_modulo_129 #(parameter N = 8)
(
input [64:1] X,
output [8:1] S
);

    wire [0 : N] [64:1] S_tmp; // Zmienna do przechowywania S_temp
    reg [64:1] Sfin_tmp; //Zmienna do weryfikacji czy ostateczny S_temp jest < P

    genvar i;

    // Przypisanie wartości X jako pierwszy S_temp
    assign S_tmp[0] = X;

    //Obliczanie kolejnych S_temp
    generate
        for (i = 0; i < N; i = i + 1) begin
```

```
            sub_S_calculation s_sub(S_tmp[i], S_tmp[i+1]);
        end
    endgenerate

    // Korekta dla ostatniego S_temp
    always @(S_tmp[N-1]) begin
        if (S_tmp[N-1] >= 129)
            Sfin_tmp <= S_tmp[N-1] - 129;
        else
            Sfin_tmp <= S_tmp[N-1];
    end

    // Przypisanie wyniku
    assign S = Sfin_tmp;

    // Wypisanie wszystkich S_temp w konsoli (w calach demonstracji)
    integer j;
    always @(Sfin_tmp) begin
        for (j = 0; j < N; j = j + 1) begin
            $display("S_tmp[%2d] = %d", j + 1, S_tmp[j]);
        end
        $display("S_tmp_fin = %d", Sfin_tmp);
    end
endmodule // x_modulo_129

module Main;
    reg [64:1] x;
    wire [8:1] s;

    //Stworzenie modułu
    x_modulo_129 #(.N(11)) mod(.X(x), .S(s));


    initial begin
    // Przypisanie wartości do x
    x = 5132051709291; // mod val = 84

    // Odczekanie chwili (aby symulacja się zaktualizowała)
    #10;

    // Wyświetl wartości
    $display("\n\n====== RESULTS ======");
    $display("Result (s): %d", s);

    // Zakończ symulacje
    $finish;
    end
endmodule //main
```