Politechnika Wrocławska

Opracowanie publikacji Danila Gorodocky'ego i Tiziano Villa'y - sumatory modularne

Łukasz Wdowiak 264026
Damian Jabłoński 264025

Prowadzący: dr inż. Piotr Patronik
Grupa: K03-47a, Pon 15:15-16:55 TN

Wydział Informatyki i Telekomunikacji
Informatyka Techniczna
IV semestr

Spis treści

1 Cele projektu

Celem projektu jest analiza oraz implementacja poszczególnych algorytmów związanych z arytmetyką modularną zaprezentowanych w artykule naukowym *Efficient Hardware Operations for the Residue Number System by Boolean Minimization* autorstwa Dana Gorodecky i Tomera Villi. W ramach projektu zostaną zaimplementowane oraz wytłumaczone algorytmy:

- Modulo dla dowolnych X i P - bit po bicie
- Algorytm mnożenia modułowego

2 Założenia projektu

Nasz projekt powinien skupiać się na implementacji oraz analizie algorytmów związanych z arytmetyką modularną. W ramach projektu powinny zostać zaimplementowane wcześniej wymienione algorytmy. Algorytmy implementowane będą w języku Python.

3 Algorytmy

Autorzy artykułu zaproponowali kilka algorytmów związanych z arytmetyką modularną. W ramach projektu skupiliśmy się na kilku z nich.

3.1 Modulo dla dowolnych X i P - bit po bicie

Algorytm ten pozwala na $X(\mathrm{mod}P)$ z dowolnych liczb. Jego głowna cechą jest to, że redukujemy liczbę X bit po bicie, aż dojdziemy do reszty z dzielenia.

3.1.1 Opis algorytmu

W artykule autorzy zaproponowali sposób obliczania oparty na następującej reprezentacji:

$$X = P \cdot Q + R = \tag{1}$$

$$= P \cdot 2^{\delta} \cdot q_{\delta} + P \cdot 2^{\delta-1} \cdot q_{\delta-1} + \ldots + P \cdot 2^{0} \cdot q_0 + R \tag{2}$$

$X(\mathrm{mod}P) = R$, gdzie $X = (x_{\psi}, x_{\psi-1}, \ldots, x_1)$ i δ jest określona nierównością $P \cdot 2^{\delta} < 2^{\psi} - 1 \leq P \cdot 2^{\delta+1}$.

Na przykład, $X = (x_{10}, x_9, \ldots, x_1)$ i $P = 21$, przy $\delta = 5$. Wynosi:

$$\begin{aligned} X = 21 \cdot Q + R = \\ &= 21 \cdot 2^5 \cdot q_5 + 21 \cdot 2^4 \cdot q_4 + 21 \cdot 2^3 \cdot q_3 + \\ &+ 21 \cdot 2^2 \cdot q_2 + 21 \cdot 2^1 \cdot q_1 + 21 \cdot 2^0 \cdot q_0 + R. \end{aligned}$$

Każdy kolejny iloczyn częściowy jest wejściem kolejnego bloku obliczeniowego. R jest wynikiem szóstego bloku oraz resztą z dzielenia przez 21. Jeśli chcemy policzyć $X(\mathrm{mod}P) = R$, gdzie $X = 888$, a $P = 21$, to:

$$\begin{aligned} &X_5 \geq 21 \cdot 2^5,\ 888 \geq 672,\ X_4 = 888 - 21 \cdot 2^5 = 216; \\ &X_4 < 21 \cdot 2^4,\ 216 < 336,\ X_3 = 216; \\ &X_3 \geq 21 \cdot 2^3,\ 216 \geq 168,\ X_2 = 216 - 21 \cdot 2^3 = 48; \\ &X_2 < 21 \cdot 2^2,\ 48 < 84,\ X_1 = 48; \\ &X_1 \geq 21 \cdot 2^1,\ 48 \geq 42,\ X_0 = 48 - 21 \cdot 2^1 = 6; \\ &X_0 < 21 \cdot 2^0,\ 6 < 21,\ R = 6. \end{aligned}$$

W pierwszym kroku porównujemy X z $21 \cdot 2^5$. Następnie odejmujemy $21 \cdot 2^5$ od X i otrzymujemy $X_4 = 216$. W kolejnym kroku porównujemy X_4 z $21 \cdot 2^4$ i otrzymujemy $X_3 = 216$. Następnie odejmujemy $21 \cdot 2^3$ od X_3 i otrzymujemy $X_3 = 48$. W kolejnym kroku porównujemy X_2 z $21 \cdot 2^2$ i otrzymujemy $X_1 = 48$. Następnie odejmujemy $21 \cdot 2^1$ od X_1 i otrzymujemy $X_0 = 6$. W kolejnym kroku porównujemy X_0 z $21 \cdot 2^0$ i otrzymujemy $R = 6$.

Jak widać powyżej jest to prosta operacja odejmowania i porównywania, która jest wykonywana w pętli.

3.1.2 Implementacja algorytmu

Algorytm został zaimplementowany za pomocą trzech funkcji.

- **calc_length** - oblicza długość binarną liczby ‘number‘ poprzez przesuwanie jej bitów w prawo i zliczanie ilości przejść, zwracając ostateczną długość.
- **get_delta** - funkcja obliczająca δ na podstawie parametrów l i P, sprawdzając warunek związanym z potęgami dwójki.
- **mod_bit_by_bit** - funkcja obliczająca resztę z dzielenia - w pętli obliczane są kolejne wartości X. Jeżeli $X_i \geq P \cdot 2^i$, to $X_{i-1} = X_i - P \cdot 2^i$, w przeciwnym wypadku $X_{i-1} = X_i$. Pętla kończy się, gdy wartość δ wynosi 0 ,a funkcja zwraca R jako reszte z dzielenia.

3.2 Algorytm mnożenia modułowego

Algorytm ten pozwala na mnożenie liczb w systemie resztowym. Jego główną cechą jest to, że dzielimy liczby na subwektory, a następnie mnożymy je w sposób opisany poniżej.

3.2.1 Opis algorytmu

Autorzy artykułu zaproponowali algorytm mnożenia modułowego, który pozwala policzyć $A \cdot B = R(\bmod P)$, gdzie $A = (A_\delta, A_{\delta-1}, \ldots, A_1), B = (B_\delta, B_{\delta-1}, \ldots, B_1)$ A_δ oraz B_δ oznaczają najbardziej znaczące bity liczb A i B, a δ jest długością słow binarnych z których się składaja. Np. $A = 13$ i $B = 14$ to $A = (1,1,0,1)$ i $B = (1,1,1,0)$, a $\delta = 2$ to $A_2 = (1,1)$ oraz $B_2 = (1,1)$. Staramy się dzielić liczby wejściowe na dwu, trzy lub czterobitowe subwektory. Odpowiednie pary subwektorów mnożymy używając poniższego wzoru:

$$R = \sum_{i=1}^{\delta} \sum_{j=1}^{\delta} A_i \cdot B_j \cdot \left(2^{m-(i+j-2)-3} (\mathrm{mod} P)\right) = S\_\mathrm{temp}$$

S_{temp} nie może przekraczać $2^{3 \cdot \delta + 2}$

Przykład wykorzystania algorytmu:
Wybieramy dwie 6-bitowe liczby $A = 45$ and $B = 15$ oraz $P = 47$. Dzielimy je na 3-bitowe subwectory. Oznacz to, że $\delta = 2$
$A_1 = (1,0,1) = 5$ $B_1 = (1,1,1) = 7$ $A_2 = (1,0,1) = 7$ $B_2 = (1) = 1$
$A \cdot B = S(\text{mod}47)$

$S_{temp} = A_1 \cdot B_1(\text{mod}47) + A_1 \cdot B_2 \cdot 2^3(\text{mod}47) + A_2 \cdot B_1 \cdot 2^3(\text{mod}47) +$
$A_2 \cdot B_2 \cdot 2^6(\text{ mod } 47) = 5 \cdot 7(\text{ mod } 47) + 5 \cdot 1 \cdot 2^3(\text{ mod } 47) + 5 \cdot 7 \cdot 2^3(\text{ mod } 47) + 5 \cdot 1 \cdot$
$2^6(\text{mod}47) = 35(\text{mod}47) + 40(\text{mod}47) + 45(\text{mod}47) + 38(\text{mod}47) = 158$

$158 >= 2^{3 \cdot 2+2} = 128$, co oznacza, że musimy wykonać kolejna iteracje aby zmniejszyć S_{temp}.

$S_{temp} = 158 = (1,0,0,1,1,1,1,0)$

$S_{temp1} = (1,1,0)$ $S_{temp2} = (0,1,1)$ $S_{temp3} = (1,0)$

$S_{temp} = 6 + 3 \cdot 2^3(\text{mod}47) + 2 \cdot 2^6(\text{mod}47) = 6 + 24 + 34 = 64.$

$64 <= 128$, więc
$S(\text{mod}47) = 64(\text{mod}47) = 17$

3.2.2 Implementacja algorytmu

Algorytm został zaimplementowany za pomoc¡ pięciu funkcji.

- **bit_counter** - oblicza długość binarną liczby 'X' poprzez przesuwanie jej bitów w prawo i zliczanie ilości przejść, zwracając ostateczną długość.
- **create_binary_subvector_alg_gorodecky** - tworzy podwektory dla danej liczby number. Przyjmuje trzy argumenty: number - liczba do podziału na podwektory, nrOfBits - liczba bitów w każdym podwektorze, nrOfSubvectors - liczba podwektorów do utworzenia. Tworzy listę podwektorów i przypisuje bity liczby number do odpowiednich podwektorów, zaczynając od najmniej znaczącego bitu.
- **binary_to_dec** - konwertuje podwektor reprezentowany w postaci binarnej na liczbę dziesiętną.

- **modulo_multiplication** - funkcja oblicza $A \cdot B(\mathrm{mod}P)$, gdzie A i B są liczbami dziesiętnymi, a P jest liczbą modulo. Jeśli finalResult przekracza wartość P, wartość jest odpowiednio pomniejszana o 'P'.

4 Minimalizacja logiczna w operacjach modułowych

Minimalizacja logiczna w operacjach modułowych odnosi się do procesu uproszczenia i optymalizacji logicznej struktury modułów w systemach cyfrowych. Do optymalizacji wykorzystaliśmy program ABC oraz przykładowy kod od Ana Petkovska z Politechniki Federalnej w Lozannie.

4.1 Minimalizacja programu w ABC

Przeprowadzona minimalizacja:

```
damian@damian-laptop:~/ABC/abc/examples$ cat rca2.v
    module rca2 (a0, b0, a1, b1, s0, s1, s2);
    input a0, b0, a1, b1;
    output s0, s1, s2;
    wire c0;
    assign s0 = a0 ^ b0 ;
    assign c0 = a0 & b0 ;
    assign s1 = a1 ^ b1 ^ c0;
    assign s2 = (a1 & b1) | (c0 & (a1 ^ b1));
    endmodule
    damian@damian-laptop:~/ABC/abc/examples$ ../abc

UC Berkeley, ABC 1.01 (compiled Jun 20 2023 21:43:32)
abc 01> rca2.v
abc 02> read rca2.v
abc 03> strash
abc 04> print_stats
rca2                          : i/o =    4/    3  lat =
                                  0  and =     13  lev =  4
abc 04>
```

- "i/o = 4/3": Wskazuje liczbę portów wejścia/wyjścia (i/o) modułu.
- łat = 0": Reprezentuje opóźnienie modułu, które odnosi się do liczby cykli zegara lub kroków czasowych wymaganych do wygenerowania przez moduł prawidłowych danych wyjściowych.
- ąnd = 13": Wskazuje liczbę bramek AND używanych w module.

- łev = 4": Reprezentuje głębokość logiczną lub poziom modułu, który jest maksymalną liczbą opóźnień bramek wzdłuż dowolnej ścieżki od wejść do wyjść.

4.2 Zmiana kodu autorów z obliczania modulo 47 na modulo 41

Udało nam się zmienić kod autorów artykułu z obliczania modulo 47 na modulo 41 oraz z liczby 100 bitowej na liczbę 25 bitową. Przetestowaliśmy to w kompilatorze online Veriloga.

4.3 Próba minimalizacji kodu autorów artykułu

Podjeliśmy próbe zminimalizowania obliczania modulo 41 liczby 25 bitowej. Niestety nie udało nam się tego dokonać. Ze względu na wyskakujący błąd. Problemem był błąd syntaxu, mimo że kod w kompilatorze online Veriloga działał poprawnie.

```
damian@damian-laptop:~/ABC/abc/examples$ ../abc
UC Berkeley, ABC 1.01 (compiled Jun 20 2023 21:43:32)
abc 01> read x_25_mod_41.v
x_25_mod_41.v (line 3): Cannot find closing bracket in this
                                  line.
abc: src/base/abc/abcObj.c:590: Abc_NtkFindOrCreateNet:
                                Assertion 'Abc_NtkIsNetlist
                                (pNtk)' failed.
Aborted
```

5 Wnioski

Artykuł naukowy skupiał się na kilku sposobach rozwiązania problemu obliczenia modulo z dużej liczby. Opisane przez nas algorytmy są tylko częścią z nich. Pierwszy algorytm stosował podejście bit po bice, a drugi dzielił liczby na subwektory. Podczas implementacji algorytmów w języku Python napotkaliśmy na kilka problemów, które zgrabnie rozwikłaliśmy. Algorytmy najpierw zostały przetestowane w sposób empiryczny tzn. na kartce papieru. Wyniki były tożsame z tymi, które obliczyły nasze pythonowe implementacje.

Dodatkowo przetestowaliśmy program ABC, którego możemy wykorzystać do minimalizacji naszych układów pod różnymi postaciami. Do wyboru mamy wiele form, jednak nie daliśmy rady przetestować każdej ze względu na ich złożoność. Minimalizator logiczny ABC zawiera podobne zasady i

techniki jak ESPRESSO. Celem jest zmniejszenie liczby bramek logicznych i optymalizacja projektu pod kątem takich czynników, jak obszar, zużycie energii i wydajność. Chociaż ABC zapewnia możliwości minimalizacji logiki, takie jak ESPRESSO, jest to bardziej wszechstronne narzędzie, które zawiera dodatkowe funkcje do syntezy, weryfikacji i testowania obwodów cyfrowych. ABC oferuje szerszy zakres funkcji poza minimalizacją logiki, dzięki czemu jest wszechstronnym narzędziem do cyfrowego projektowania i optymalizacji.

Literatura

[1] D. Gorodecky and T. Villa, "Efficient Hardware Operations for the Residue Number System by Boolean Minimization", Advanced Boolean Techniques, Minsk, Belarus, January, 2020, p. 237-258

[2] https://en.wikipedia.org/wiki/Residue_number_system

[3] https://www.tutorialspoint.com/compile_verilog_online.php

[4] https://pl.wikipedia.org/wiki/Verilog

[5] https://people.eecs.berkeley.edu/~alanmi/abc/

[6] https://www.dropbox.com/s/qrl9svlf0ylxy8p/ABC_GettingStarted.pdf

6 Kod źródłowy

6.1 bit_by_bit.py

```
import math


def calc_length(number):
length = 0
while number != 0:
number >>= 1
length += 123


def mod_bit_by_bit(X, P, delta):
while delta >= 0:
if X >= P * math.pow(2, delta):
X -= P * math.pow(2, delta)
print("X" + str(delta) + " : ", X)
delta -= 1
return X


def bit_by_bit(X, P):
l = calc_length(X)
delta = get_delta(l, P)
R = mod_bit_by_bit(X, P, delta)
print("Ilosc bitow: ", l)
print("Ile iteracji: ", delta)
print("Reszta: ", R)


if __name__ == "__main__":
X = 888
P = 21
bit_by_bit(X, P)
```

6.2 modular_multiplication.py

```
import math


def bit_counter(number):
if number == 0:
return 1

count = 0
while number > 0:

count += 1
number >>= 1
return count


def create_binary_subvector_alg_gorodecky(number, nrOfBits,
                                          nrOfSubvectors):

subvectors = []
for _ in range(nrOfSubvectors):
subvector = [0] * nrOfBits
subvectors.append(subvector)

for i in range(nrOfSubvectors):
for j in range(nrOfBits - 1, -1, -1):
subvectors[i][j] = number % 2
number //= 2
return subvectors


def binary_to_dec(subvector):

dec = 0
for i in range(len(subvector)):
dec += subvector[i] * int(math.pow(2, len(subvector) - i -
                                   1))
return dec


def modulo_multiplication(A, B, P):

bitsOfbiggerNumber = bit_counter(max(A, B))
nrOfBits = 3  # r
nrOfSubvectors = math.ceil(bitsOfbiggerNumber / nrOfBits)
                                   # delta
```

```
    subvectorsA = create_binary_subvector_alg_gorodecky(
    A, nrOfBits, nrOfSubvectors)

    subvectorsB = create_binary_subvector_alg_gorodecky(
    B, nrOfBits, nrOfSubvectors)

    result = 0

    for i in range(len(subvectorsA)):
    for j in range(len(subvectorsB)):

    A_iter = binary_to_dec(subvectorsA[i])
    B_iter = binary_to_dec(subvectorsB[j])

    S_temp = A_iter * B_iter * int(math.pow(2,
    ((i + 1) + (j + 1) - 2) * nrOfBits)) % P

    result += S_temp

    range_ = int(math.pow(2, 3 * nrOfSubvectors +
                                    nrOfSubvectors))

    rangeBitCount = bit_counter(range_)
    resultSubvectorCount = math.ceil(rangeBitCount / nrOfBits)
    resultSubvectors = create_binary_subvector_alg_gorodecky(
                                    result, nrOfBits,
    resultSubvectorCount)

    power = 0
    finalResult = 0
    for i in range(nrOfSubvectors + 1):
    ans_temp = binary_to_dec(
    resultSubvectors[i]) * int(math.pow(2, power)) % P
    power += nrOfBits
    finalResult += ans_temp
    if finalResult > P:
    finalResult -= P
    print(finalResult)


    A = 35
    B = 77
    P = 53

    modulo_multiplication(A, B, P)
```

6.3 Obliczanie modulo 41 liczby 25 bitowej - verilog

```
module main(X, S);
input [24:0] X;
output [5:0] S;
wire [14:0] S_temp_1;
wire [11:0] S_temp_2;
wire [10:0] S_temp_3;
wire [9:0] S_temp_4;
wire [8:0] S_temp_5;
wire [7:0] S_temp_6;
wire [6:0] S_temp_7;
reg [5:0] S_temp;

assign S_temp_1 = X[5:0] + X[11:6] * 6'b101001 + X[17:12] *
                                    6'b100011 + X[23:18] * 7'
                                    b1001001;
assign S_temp_2 = S_temp_1[5:0] + S_temp_1[13:6] * 6'
                                    b101001 + S_temp_1[14:14] *
                                    6'b100011;
assign S_temp_3 = S_temp_2[5:0] + S_temp_2[11:6] * 6'
                                    b101001;
assign S_temp_4 = S_temp_3[5:0] + S_temp_3[10:6] * 6'
                                    b101001;
assign S_temp_5 = S_temp_4[5:0] + S_temp_4[9:6] * 6'b101001
                                    ;
assign S_temp_6 = S_temp_5[5:0] + S_temp_5[8:6] * 6'b101001
                                    ;
assign S_temp_7 = S_temp_6[5:0] + S_temp_6[7:6] * 6'b101001
                                    ;


always @(S_temp_7) begin
if (S_temp_7 >= 6'b101001)
S_temp <= S_temp_7 - 6'b101001;
else
S_temp <= S_temp_7;
end

assign S = S_temp;
endmodule

module testbench;
reg [24:0] X;
wire [5:0] S;
main dut(.X(X), .S(S));

initial begin
X = 42;
#10;
```

```
    $display("S = %d", S);
    $finish;
    end
    endmodule
```

6.4 Kod minimalizowany w ABC

```
    module rca2 (a0, b0, a1, b1, s0, s1, s2);
    //--------------Input Ports Declarations--------------------
                                        ---------
    input a0, b0, a1, b1;
    //--------------Output Ports Declarations-------------------
                                        ----------
    output s0, s1, s2;
    //--------------Wires---------------------------------------
                                        --------
    wire c0;
    //--------------Logic---------------------------------------
                                        --------
    assign s0 = a0 ^ b0 ;
    assign c0 = a0 & b0 ;
    assign s1 = a1 ^ b1 ^ c0;
    assign s2 = (a1 & b1) | (c0 & (a1 ^ b1));
    endmodule
```