Skalowanie w systemie RNS

Organizacja i architektura komputerów

Łukasz Broll
Kierunek Informatyka, Wydział Elektroniki
Politechnika Wrocławska
Wrocław, Polska
225972@student.pwr.edu.pl

Krzysztof Dombek
Kierunek Informatyka, Wydział Elektroniki
Politechnika Wrocławska
Wrocław, Polska
226093@student.pwr.edu.pl

STRESZCZENIE

Projekt skalowania w systemie RNS opiera się na dokumencie '2^n RNS Scalers for Extended 4-Moduli Sets' autorstwa Leonel Sousa. Na podstawie metody cztero-modułowego setu została wykonana implementacja programu w PARI/GP oraz w Verilogu.

I. WSTĘP

A. Idea skalowania

Skalowanie to dzielenie liczby przez potęgę podstawy. W naszym projekcie opiera się o liczby binarne, czyli dzielenie przez potęgę 2. Algorytm, na podstawie którego został zrealizowany projekt opiera się na resztowym systemie zapisu liczb. W procesie skalowania można wyróżnić następujące punkty: przekonwertowanie dzielnej na system **R**esidue **N**umber **S**ystem (resztowy), osobne przeskalowanie każdej z reszt R_1-R_4 w RNS przez dzielnik, przekonwertowanie wyniku z powrotem na system dziesiętny.

B. Resztowy system zapisu liczb

Zgodnie z teorią jest to system liczbowy służący do reprezentacji liczb całkowitych za pomocą wektora reszt z dzielenia, względem ustalonego wektora, wzajemnie, względnie pierwszych modułów. Chińskie twierdzenie o resztach gwarantuje, że taka reprezentacja jest jednoznaczna dla liczb całkowitych ze zbioru [0,M], gdzie M jest iloczynem wszystkich modułów. Jego główna zaletą jest, to że obliczenia są wykonywane bez przenoszenia bitu nadmiaru oraz równolegle w każdym kanale dla operacji takich jak: dodawanie liczb całkowitych, mnożenie czy dzielenie. Wykorzystywany w kryptografii czy liniowym przetwarzaniu sygnałów, zapewnia lepszą wydajność przy mniejszym zużyciu energii. System stosowany jest również, gdy dane, na których operujemy mają dynamiczny rozmiar.

C. Konwersja

Konwersja odwrotna między RNS a ważonym systemem binarnym, wymaga operacji na sumach częściowych i nie może być indywidualnie wyliczana dla każdego z kanałów RNS. Operacje te mają znaczący wpływ na złożoność arytmetyki RNS, ale tylko w przypadku bardzo złożonych i intensywnych obliczeń. W przeciwieństwie do arytmetyki zmiennoprzecinkowej, arytmetyka stałej liczby całkowitej zazwyczaj wymaga skalowania, aby zapewnić, że obliczone wyniki nie przekroczą dynamicznego zakresu.

D. Metody skalowania

Większość algorytmów RNS korzysta z tablicowania, przez co ich implementacja wymaga pamięci RAM, w celu wspierania różnych zestawów modułów. Niestety nie jest to efektywna metoda, gdyż koszty sprzętu rosną wraz z dynamicznym zasięgiem oraz przepustowość przetwarzania jest ograniczona i maleje wraz z wielkością pamięci. Skalowanie jest istotną operacją arytmetyczną, która jest ciężka do zaprezentowania w systemie RNS. W naszym artykule znajdujemy efektywne rozwiązanie tego problemu, która gwarantuje zakres dla dużych rozmiarów dynamicznych. Zaproponowane zostały różne rozwiązania problemów: 3 modułowe zestawy, rozszerzony 4 moduły zestaw, który rozszerza tradycyjne 3 modułowe zestawy o czwarty element czy narastający 3 modułowy zestaw. Każdy z algorytmów może działać efektywnie dla skalowania RNS, poprzez rozwinięcia zaproponowane w artykule oraz implementacje rozważające dynamiczne zakresy skalowania, uzależnione od potrzeb wykorzystywania danego algorytmu.

E. Definicje zmiennych

Poniżej przedstawiamy krótkie wyjaśnienia poszczególnych zapisów, aby lepiej zrozumieć, co dokładnie zostało zaimplementowane w użytych przez nas wzorach:

- baza $\{m_1, m_2, m_3, \dots, m_N\}$ elementów względnie pierwszych gdzie m_i to moduł, dowolnej liczby całkowitej X.
- R_i – reszta wynikająca z dzielenia X przez moduł m_i, przedstawiana również jako wektor bitowy $r_{i(n-1)}, \dots, r_{i(0)}$.
- Reszty dla określonego modułu $\prod_{i=l}^{n} m_i$ są przedstawione jako $R_{n \to l} (0 < l < n \leq N)$
- S_i określa resztę dla modułu m_i po przeskalowaniu.

II. Środowiska aplikacji

A. PARI/GP

Do obliczeń czysto teoretyczno-matematycznych wykorzystaliśmy pogram Pari/GP – system komputerowy przeznaczony do szybkich obliczeń algebraicznych. Prosty interfejs konsolowy, rozbudowana dokumentacja i intuicyjna obsługa pozwoliły nam w miarę szybko zrozumieć jak program działa oraz jak korzystać z funkcji, które potrzebowaliśmy zaimplementować: tworzenie procedury, pętli, drukowanie wyników i operacje arytmetyczne. Zaimplementowany przez nas kod generował poprawne wyniki, co zostało przetestowane dla kilku różnych przykładów. Aplikacja PARI/GP została przygotowana również pod systemy z rodziny Microsoft Windows i nasz program pisaliśmy na systemie Windows 10. [2]

B. Verilog

Język Veriloga wykorzystuje się do opisu sprzętu w projektowaniu oraz symulacji układów cyfrowych. Największą trudnością dla nas było zrozumienie całej idei języku opisu sprzętu. Do tej pory proste układy implementowaliśmy bezpośrednio na sprzęcie, bez komputerowego przygotowania algorytmu i symulacji. Skrócone kursy uczelniane, przykłady i dokumentacja również w tym przypadku wprowadziły nas w podstawy programowania językiem Verilog, co wystarczyło w naszym symulatorze na zadeklarowanie modułów, sygnałów, pętli, obliczeń arytmetycznych oraz modułu test_bench. Dla potrzeb naszego projektu korzystaliśmy z internetowego środowiska kompilatora firmy Codingground dostępnego pod adresem https://www.tutorialspoint.com/compile_verilog_online.php [3]

III. Implementacja

Implementacja mogła zostać wykonana dla wielu różnych zestawów skalowania RNS, jednak w naszym przypadku ustalony wektor modułów wynosił: { 2^n -1, 2^n, 2^n+1, 2^{n+1}-1}. Chińskie twierdzenie o resztach gwarantuje, że taka reprezentacja jest jednoznaczna dla liczb całkowitych ze zbioru [0,M], gdzie M jest iloczynem wszystkich modułów.
Poniżej wypisaliśmy ostateczne wersje wzorów, z których należy skorzystać w implementacji. Pełne wyprowadzenia każdego wzoru zaprezentowane są dokładnie w dokumencie źródłowym.

$$S_1 = \langle R_1 - \langle R_2 \rangle_{2^n} \rangle_{2^n - 1} \tag{1}$$

$$S_2 = \langle \langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor \rangle_{2^n} + \langle (R_4 - R_{3\leftrightarrow\leftrightarrow 1})\Psi \rangle_{2^{n+1}-1} \rangle_{2^n} \tag{2}$$

$$S_3 = \langle \langle R_2 \rangle_{2^n} - R_3 \rangle_{2^n + 1} \tag{3}$$

$$S_4 = \langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor + (R_4 - R_{3\leftrightarrow\leftrightarrow 1})\Psi(2^{n+1} - 1) \rangle_{2^{n+1}-1} \tag{4}$$

$$\langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor \rangle_{2^n} = \langle \langle \begin{matrix} (2^{n-1} + 2^{2n-1}) * R_1 - 2^n * R_2 + \\ (2^{n-1} + 2^{2n} - 1 - 2^{2n-1}) R_3 \end{matrix} \rangle_{2^{2n}-1} \rangle_{2^n} \tag{5}$$

$$R_{3\leftrightarrow 1} = \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor * 2^n + \langle R_{3\leftrightarrow 1} \rangle_{2^n} \tag{6}$$

$$\Psi = 2^n + 2^{n-2} + \cdots + 2^4 + 2 \tag{7}$$

W tym miejscu warto dodać, iż parametr psi - Ψ zostaje uwzględniany tylko i wyłącznie, jeżeli podane $n \geq 4$. W naszym przykładzie $n = 8$, więc korzystamy z parametru psi, aby wyliczyć składowe reszty S_2 *oraz* S_4.

A. Kod źródłowy implementacji w PARI/GP

Program wykonywany jest w procedurze `scalling(x,n)`, gdzie x to liczba dzielna, a n to wykładnik 2 w dzielnej. Dzięki temu, łatwo można zmienić liczby na których się operuje, a PARI/GP gwarantuje automatyczny dobór rozmiaru zmiennej.
Zmienna `p1` to odpowiednik równania (5), zmienna `p2` równania (6), a pętla w kodzie wylicza psi (7). Pozostałe parametry nazwane są odpowiednio z wykorzystaniem.

```
scalling(x, n) = {
    m1 = 2 ^ n - 1;
    m2 = 2 ^ n;
    m3 = 2 ^ n + 1;
    m4 = 2 ^ (n + 1) - 1;
    M = m1 * m2 * m3 * m4;
    R1 = x % m1;
    R2 = x % m2;
    R3 = x % m3;
    R4 = x % m4;

  p1 = ((((2 ^ (2 * n - 1) + 2 ^ (n - 1)) * R1) - (2
   ^ n * R2) + (2 ^ (n - 1) +2 ^ (2 * n) - 1 -
- 2 ^ (2 * n - 1)) * R3) % (2 ^ (2 * n) - 1)) % 2^n;

 p2 = p1 * 2 ^ n + R2;


    temp = n;
    w = 0;
    for (i = 1, n, i++; w += 2 ^ temp; temp =temp-2;
        if (temp == 2,
            w += 2;
            break;)
    );

    S1 = (R1 - (R2 % 2 ^ n)) % (2 ^ n - 1);
    S3 = (R2 % 2 ^ n - R3) % (2 ^ n + 1);
    S2 = (p1 + (((R4 - p2) * w) % (2 ^ (n + 1) - 1))
* (-1)) % 2 ^ n;
    S4 = (p1 + ((R4 - p2) * w) * (2 ^ (n - 1) - 1))
% (2 ^ (n + 1) - 1);

    print("\nSet reszt z CRT:\nR1:", R1, " R2:", R2,
" R3:", R3, " R4:", R4);
    print("\nParametr S1 = ", S1);
    print("\nParametr S2 = ", S2);
    print("\nParametr S3 = ", S3);
    print("\nParametr S4 = ", S4); }
```

B. Kod źródłowy implementacji w Verilogu

W naszym symulatorze operujemy bezpośrednio na dzielnej i dzielniku, które potrzebne są do wyliczenia poszczególnych modułów, jednak tak naprawdę na wejściu naszego układu podawane są wyliczone wcześniej reszty, na których operujemy, uzyskując ostateczny wynik. Każda z reszt S_1-S_4 rozpisana jest w osobnym module, gdzie zadeklarowane są wykorzystywane porty. Dla każdego portu określamy jaką funkcję pełni – input, output tj. wejścia i wyjścia. Korzystamy również z magistral wewnętrznych, które bezpośrednio związane są z poleceniem „wire". Wywołując nasz program korzystamy z instancji, które wywołują poszczególne moduły. Dla każdego portu przewidzieliśmy zapas, jeżeli chodzi o rozmiar sygnału, który będzie nim wysyłany, aby umożliwić testowanie naprawdę dużych liczb.

```
module s1(
input [7:0] R1,
input [7:0] R2,
input [7:0] n,
output [7:0] S1);

assign S1=(R1-(R2%2**n)+(2**n-1))%(2**n-1);

endmodule

module s3(
input [7:0] R2,
input [8:0] R3,
input [7:0] n,
output [8:0] S3);

assign S3=(R2%2**n-R3)%(2**n+1);

endmodule

module p(
input [7:0] R1,
input [7:0] R2,
input [8:0] R3,
input [7:0] n,
output [15:0] p1,
output [24:0] p2
);

assign p1 = ((((2**(2*n-1)+2**(n-1))*R1)-
(2**n*R2)+(2**(n-1)+2**(2*n)-1-2**(2*n-
1))*R3)%(2**(2*n)-1))%2**n;
assign p2 = p1*2**n+R2;

endmodule

module s2(
input [15:0] p1,
input [24:0] p2,
input [8:0] R4,
input [7:0] n,
output [7:0] S2);

integer i,temp,w;

initial begin

w=0;
temp=n;
begin : block
for(i=1;i<=n; i++ )
begin
    w+=2**temp;
          temp=temp-2;
          if(temp==2)
                    begin
                    w+=2;
                    disable block;
                    end;
    end
end
end

assign S2=(p1+(((R4-p2)*w)%(2**(n+1)-1))*(-1))%2**n;

endmodule

module s4(
input [15:0] p1,
input [24:0] p2,
input [8:0] R4,
input [7:0] n,
output [8:0] S4);

integer i,temp,w;

initial begin
w=0;
temp=n;
begin : block
for(i=1;i<=n; i++ )
begin
    w+=2**temp;
          temp=temp-2;
          if(temp==2)
                    begin
                    w+=2;
                    disable block;
                    end;
    end
end
end
assign S4 = (p1+((R4-p2)*w)*(2**(n-1)-1))%(2**(n+1)-
1);

endmodule

module test();

reg [7:0] r1;
reg [7:0] r2;
reg [8:0] r3;
reg [8:0] r4;

wire [15:0] p1;
wire [24:0] p2;
wire [7:0] ss1;
wire [7:0] ss2;
wire [8:0] ss3;
wire [8:0] ss4;
wire [7:0] n= 8'd8;

s1 S1(
.R1(r1),
.R2(r2),
.n(n),
.S1(ss1)
);

s3 S3(
.R2(r2),
.R3(r3),
.n(n),
.S3(ss3)
);
```

```
p P(
.R1(r1),
.R2(r2),
.R3(r3),
.n(n),
.p1(p1),
.p2(p2)
);

s2 S2(
.p1(p1),
.p2(p2),
.R4(r4),
.n(n),
.S2(ss2)
);

s4 S4(
.p1(p1),
.p2(p2),
.R4(r4),
.n(n),
.S4(ss4)
);

initial
 begin
 r1 = 8'd63;
 r2 = 8'd255;
 r3 = 9'd192;
 r4 = 9'd7;
  $monitor ("S1: %d\nS2: %d\nS3: %d\nS4: %d", ss1,
ss2, ss3, ss4);
 end

endmodule
```

IV. PRZYKŁAD

TABELA 1. PRZYKŁAD SKALOWANIA W RNS LICZBY X = $2^{42}-1$ PRZEZ 2^n DLA SETU 4-MODUŁÓW I DZIELNIKA N=8 ($2^8 = 256$)

Set modułów	255	256	257	511
Oryginalne reszty	3	255	252	63
Odwrotność X	$2^{42}-1 \quad \lfloor (2^{42}-1)/2^8 \rfloor == 2^{34}-1$			
Reszty przeskalowanego wyniku	$S_1 = 252$	$S_2 = 255$	$S_3 = 3$	$S_4 = 127$
$S_1 = \langle R_1 - R_2 \rangle_{2^n}$	$S_1 = \langle 3 - 255 \rangle_{256} = 252$			
$S_3 = \langle R_2 - R_3 \rangle_{2^n}$	$S_3 = \langle 255 - 252 \rangle_{256} = 3$			
$\langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor \rangle_{2^n} = \langle \langle (2^{n-1} + 2^{2n-1}) * R_1 - 2^n * R_2 + (2^{n-1} + 2^{2n} - 1 - 2^{2n-1}) R_3 \rangle_{2^{2n}-1} \rangle_{2^n}$	$\langle \langle (128 + 32768) * 3 - 256 * 255 + (128 + 65535 - 32768) * 252 \rangle_{65535} \rangle_{256} = 3$			
$R_{3\leftrightarrow 1} = \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor * 2^n + \langle R_{3\leftrightarrow 1} \rangle_{2^n}$	$R_{3\leftrightarrow 1} = 3 * 256 + 255 = 1023$			
$\Psi = 2^n + 2^{n-2} + \cdots + 2^4 + 2$	$\Psi = 2^8 + 2^6 + \cdots + 2^4 + 2 = 338$			
$S_2 = \langle \langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor \rangle_{2^n} + \langle (R_4 - R_{3\leftrightarrow\leftrightarrow 1}) \Psi \rangle_{2^{n+1}-1} \rangle_{2^n}$	$S_2 = \langle 3 + \langle (63 - 1023) * 338 \rangle_{511} * (-1) \rangle_{256} = 255$			
$S_4 = \langle \left\lfloor \frac{R_{3\leftrightarrow 1}}{2^n} \right\rfloor + (R_4 - R_{3\leftrightarrow\leftrightarrow 1}) \Psi (2^{n+1} - 1) \rangle_{2^{n+1}-1}$	$S_4 = \langle 3 + \langle (31 - 1023) * 338 \rangle_{511} * 127 \rangle_{256} = 127$			

a.

V. WNIOSKI

Najważniejszym zadaniem projektu było dokładne zrozumienie treści artykułu oraz zaimplementowanie podanych w nim sposobów realizacji określonych algorytmów. Naukowy język oraz fachowość nazewnictwa w języku angielskim sprawiły nam duże problemy w interpretacji treści, a bez dokładnego zrozumienia całej idei skalowania nie bylibyśmy w stanie zrealizować dalszych kroków. Po dokładnym zrozumieniu treści artykułu, zaczęliśmy rozumieć idee skalowania, ćwicząc na swoich przykładach oraz sprawdzając w symulatorach zgodność wyników. Następnie przystąpiliśmy do implementacji kodu w Pari/GP, który wprowadził nas w system operowania na resztach. Realizując kod, który wylicza poprawne wartości, przystąpiliśmy do realizacji zadania w Verilog'u. Brak wcześniejszej styczności z tym środowiskiem przysporzył nam spore problemy, jednak ostatecznie udało nam się zrealizować cel projektu.

ODWOŁANIA

[1] Leonel Sousa, *2n RNS Scalers for Extended 4-Moduli Sets* , IEEE TRANSACTIONS ON COMPUTERS, VOL. 64, NO. 12, DECEMBER 2015

[2] Karim Belabas, *PARI/GP* https://pari.math.u-bordeaux.fr

[3] Verilog, IEEE 1364 https://en.wikipedia.org/wiki/Verilog