Politechnika Wrocławska

Wydział Elektroniki
Kierunek Informatyka

Architektura komputerów: Mnożenie liczb stałoprzecinkowych wymiernych w procesorze RNS

Konrad Stręk 248900, czw. TP 17:05
Maksim Birszel 241353, czw. TN 18:55

Prowadzący:
Dr inż. Piotr Patronik

Wrocław 2020

Spis treści

Rozdział 1

Wprowadzenie

1.1 Cele projektu

- Analiza i implementacja mnożenia liczb wymiernych w systemie RNS
- Implementacja schematu mnożenia w zadanym procesorze RNS według istniejącego schematu

1.2 Wstęp

Mnożenie modularne jest operacją, która pozwala rozbić tradycyjne mnożenie na mniejsze operacje. Ze względu na możliwość wykonywania tych operacji równolegle mnożenie modularne jest dobrą alternatywą, gdy zależy nam na wydajności. Przykładami zastosowań dla takiego mnożenia są algorytmy kryptografii.

Mnożenie takie może być także zastosowane dla reprezentacji stałoprzecinkowych, dla których arytmetyka może być wydajniejsza od zmiennoprzecinkowej lub po prostu może być bardziej odpowiednia dla danego problemu.

W naszej pracy zajmiemy się algorytmem mnożenia stałoprzecinkowego opartego o patent Olsena[1].

Rozdział 2

Podstawy matematyczne

2.1 Wstęp

System resztowy (RNS od ang. residue number system) – system liczbowy służący do reprezentacji liczb całkowitych wektorem reszt z dzielenia względem ustalonego wektora wzajemnie względnie pierwszych modułów.

Na liczbach reprezentowanych w systemie resztowym może być efektywnie przeprowadzonych wiele operacji arytmetycznych. Wykonuje się je, przeprowadzając niezależnie na odpowiednich resztach „zwykłe" operacje, a następnie operację modulo odpowiedniego modułu.[7,5].

Ogromną zaletą operacji w systemach resztowym jest wyeliminowanie propagacji przeniesiem, co znacznie przyspiesza obliczenia.

W ramach implementowanego algorytmu konieczne jest zapoznanie się z poniższymi pojęciami.

2.2 Odwrotność multiplikatywna

Odwrotność multiplikatwna(z ang. multiplicative inverse) lub odwrotność modularna liczby całkowitej a to taka liczba x, dla której iloczyn ax przystaje do 1 modulo m, co możemy symbolicznie zapisać jako:

$$ax \cong 1 \ (mod \ m)$$

Aby obliczyć odwrotność modularną można zastosować podejście naiwne, czyli dla kolejnych liczb sprawdzamy czy iloczyn ax przystaje do 1 modulo m, ale można także skorzystać z rozszerzonego algorytmu Euklidesa, który jest wydajniejszym rozwiązaniem.

2.3 Chińskie twierdzenie o resztach

Chińskie twierdzenie o resztach mówi o tym, że znając znając reszty z dzielenia danej liczby a przez liczby z bazy $M = \{m_1, m_2, ..., m_n\}$ jesteśmy w stanie jednoznacznie wyznaczyć liczbę a, pod warunkiem, że liczby z bazy M są względnie pierwsze. Oznacza to, że liczbę a możemy przedstawić jako wektor $< a_1, a_2, ..., a_n >$, gdzie:

$$a_n = a \ (mod \ m_n)$$

Maksymalna liczba wartości jakie możemy wyznaczyć w ten sposób jest równa M_{max}, gdzie:

$$M_{max} = m_1 \cdot m_2 \cdot ... \cdot m_n$$

2.3.1 Konwersja z systemu resztowego na pozycyjny

Aby dokonać konwersji z systemu resztowego wykorzystywane jest równanie [6]:

$$a = \left(\sum_{i=1}^{n} \hat{m}_i \cdot (\hat{m}_i^{-1} \ mod \ m_i) \cdot a_i \right) \ mod \ M_{max}$$

gdzie:

$$\hat{m}_i = \prod_{j=1}^{n} m_j \text{ dla } j \neq i$$

2.4 System z mieszanymi podstawami (MRS)

MRS jest systemem wagowym. Liczba X z zakresu określonego przez bazę RNS, ma reprezentację w MRS w postaci:

$$X = a_n \prod_{i=1}^{n-1} m_i + ... + a_3 m_1 m_2 + a_2 m_1 + a_1 = a_n P_n + ... + a_3 P_3 + a_2 P_2 + a_1 P_1$$

gdzie $a_1, ..., a_n$ są cyframi MRS.

Reprezentacja liczby w MRS określona jest jako $(a_n, a_{n-1}, ..., a_1)$ gdzie a_n jest cyfrą MRS stojącą przy najbardziej znaczącej wadze. Każda liczba z zakresu określonego przez RNS ma dokładnie jedną reprezentację w MRS. System dziesiętny jest szczególnym przypadkiem MRS, w którym wszystkie $m_i = 10$.

2.5 Wnioski

Zrozumienie podstaw matematycznych w systemie resztowym jest podstawą do wykonania w nim jakichkolwiek operacji arytmetycznych.

Z odpowiednią znajomością wymienionych terminów, jesteśmy w stanie wykonać niektóre z nich wydajniej niż za pomocą tradycyjnych sposobów. Jedną z takich operacji jest mnożenie liczb stałoprzecinkowych z użyciem algorytmu konwersji do MRN.

Rozdział 3

Implementacja

3.1 Wstęp

W początkowej fazie, algorytm został zaimplementowany w Pari/GP, na potrzeby zrozumienia jego działania i określenia najważniejszych jego elementów.

Następnie wykorzystany został specjalny Assembler, którego składnia jest opisana w dokumencie[3].

Głównym, a zarazem najważniejszym elementem programu jest część zawierająca konwersję na system MRN. Całość została zaimplementowana na podstawie algorytmu przedstawionego w źródle[1].

3.2 Sposób implementacji

3.2.1 Pari/GP

Implementując mnożenie stałoprzecinkowe w Pari wzorowaliśmy się głównie na schematach opisujących algorytm mnożenia oraz konwersji z RNS do MRN zamieszczonych poniżej[1]:

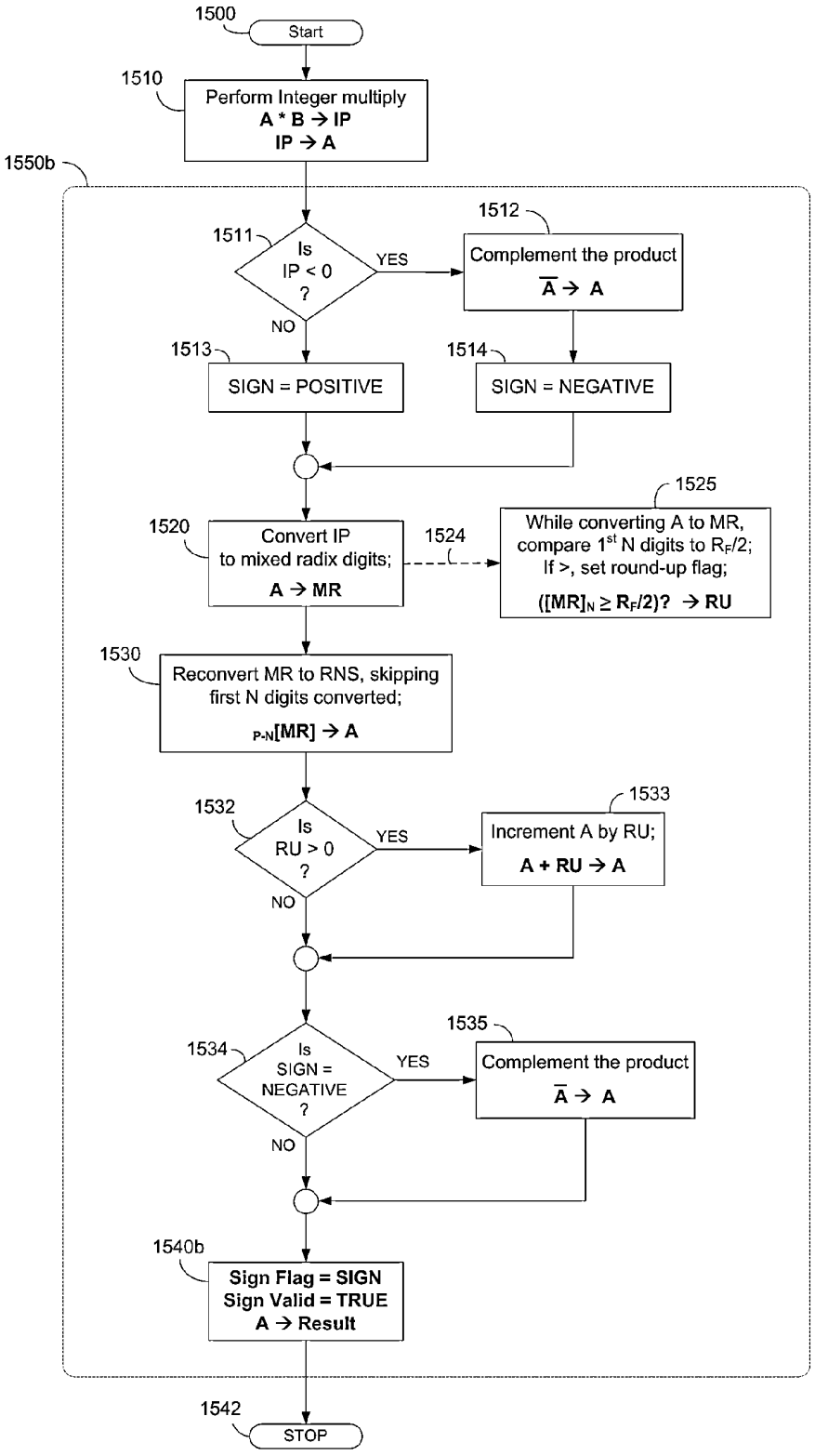
1500
Start
1510
Perform Integer multiply
A * B → IP
IP → A
1550b
1511
Is
IP < 0
?
YES
NO
1512
Complement the product
$\overline{A}$ → A
1513
SIGN = POSITIVE
1514
SIGN = NEGATIVE
1520
Convert IP
to mixed radix digits;
A → MR
1524
1525
While converting A to MR,
compare 1st N digits to $R_F/2$;
If >, set round-up flag;
($[MR]_N \geq R_F/2$)? → RU
1530
Reconvert MR to RNS, skipping
first N digits converted;
$_{P-N}$[MR] → A
1532
Is
RU > 0
?
YES
NO
1533
Increment A by RU;
A + RU → A
1534
Is
SIGN =
NEGATIVE
?
YES
NO
1535
Complement the product
$\overline{A}$ → A
1540b
Sign Flag = SIGN
Sign Valid = TRUE
A → Result
1542
STOP

Figure 15B

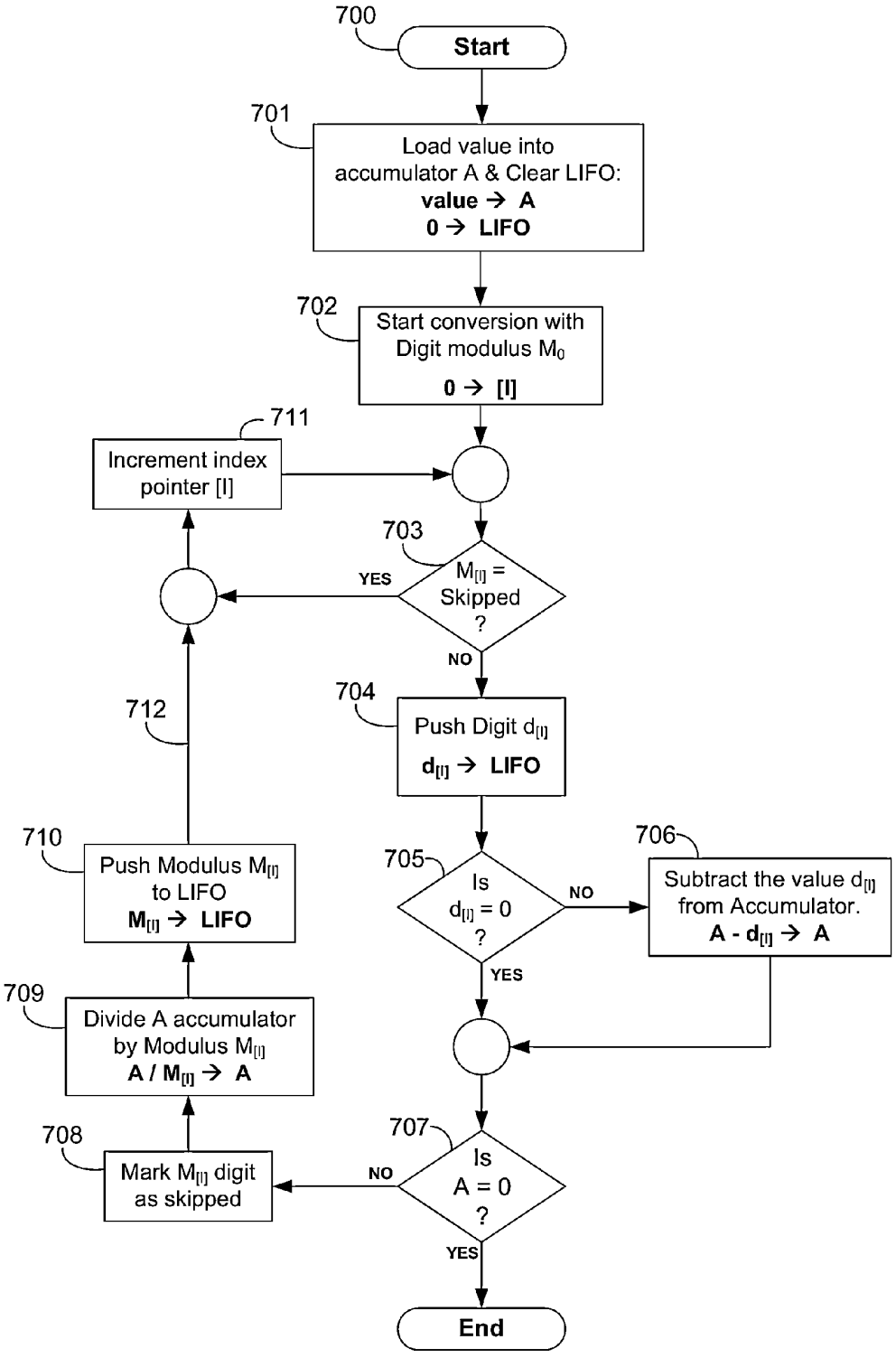
700
Start
701
Load value into accumulator A & Clear LIFO:
value → A
0 → LIFO
702
Start conversion with Digit modulus M0
0 → [I]
711
Increment index pointer [I]
703
M[I] = Skipped ?
YES
NO
704
Push Digit d[I]
d[I] → LIFO
712
710
Push Modulus M[I] to LIFO
M[I] → LIFO
705
Is d[I] = 0 ?
NO
706
Subtract the value d[I] from Accumulator.
A - d[I] → A
YES
709
Divide A accumulator by Modulus M[I]
A / M[I] → A
708
Mark M[I] digit as skipped
707
Is A = 0 ?
NO
YES
End

Figure 7A

W związku z tym zaimplementowaliśmy operację sprawdzania znaku, konwersji z obcięciem(z ang. truncaton) oraz zaokrąglania w górę.

Znak liczby

Określanie znaku polega na porównaniu liczby z połową zakresu dla danej bazy modułów. Porównywanie polega na stopniowej konwersji obydwu argumentów do MRN i porównywaniu kolejnych wartości MRN, zaczynając od najniższej wagi.

Gdy już raz określimy znak argumentów, jesteśmy w stanie wnioskować znak wyniku mnożenia na podstawie określonych wcześniej flag znaku.

Sam znak nie wpływa znacząco na algorytm mnożenia. Ważne jest tylko, aby przed i po konwersji dopełnić liczbę, jeśli jest ujemna.

Konwersja do MRN i konwersja odwrotna

Głównym zadaniem była implementacja konwersji do MRN, która jest kluczowa dla sposobu mnożenia przedstawionego w patencie[1]. Widać to na poniższym równaniu:

$$(Y_1 * Y_2)/(R_F * R_F) = ((Y_1 * Y_2)/R_F)/R_F \tag{3.1}$$

Musimy dokonać dzielenia przez R_F, aby znormalizować wynik. Konwersja do MRN pozwala nam to zrobić bardzo wydajnie, ponieważ będąc w systemie wagowym, gdy dokonujemy konwersji odwrotnej, po prostu ignorujemy moduły odpowiadające za część ułamkową.

Wzorując się na patencie Olsena zaimplementowaliśmy konwersję stosując LIFO(stos), który przy konwersji odwrotnej pozwala na dostęp do elementów w odwrotnej kolejności. Śledząc liczbę elementów na stosie jesteśmy także w stanie łatwo pominąć ostatnie N elementów.

Dokonując konwersji na MRN dokonujemy równocześnie porównania z połową zakresu ułamka, aby zdecydować czy ostateczny wynik należy zaokrąglić w górę.

3.2.2 Assembler RNS

Opis składni

Wykorzystany przez nas Assembler opiera się na kilku podstawowych instrukcjach, które wykonują operacje równolegle na wszystkich kanałach:

- `f_residue` - wykonuje konwersję na system RNS
- `f_mul_m` - wykonuje mnożenie modulo
- `f_add_m` - wykonuje dodawanie modulo

Przy implementacji założyliśmy, że mamy dostępne 3 akumulatory. W kodzie są to odpowiednio *acc*, *acc*2, *digits*.

Implementacja

W tym przypadku z powodu ogarniczonej liczby rozkazów zaimplementowaliśmy mnożenie bez sprawdzania znaku i zaokrąglania w górę. Mnożenie stałoprzecinkowe zostało zaimplementowane dla bazy 5-modułowej. Przy założeniu, że w tym 2 moduły odpowiadają za część ułamkową.

Jako, że tym razem nie był dostępny stos, kolejne elementy są odseparowywane z akumulatora.

3.3 Wnioski

Wybierając język Pari/GP w początkowej implementacji algorytmu, byliśmy w stanie zapoznać się z pełnym jego działaniem. Następnie używając assemblera mogliśmy zrozumieć w jaki sposób takie mnożenie może być wykonane w przykładowym procesorze, który używa rozkazów wykonujących operacje równolegle na wielu kanałach.

Rozdział 4

Wnioski

Mnożenie modularne jest operacją, która pozwala uzyskać poprawę wydajności w stosunku do tradycyjnego mnożenia. Jego głównym atutem jest możliwość wykonania mnożenia w mniejszych porcjach, które mogą być wykonywane równolegle.

Niemniej jednak niektóre operacje w systemie resztowym są trudniejsze do wykonania niż w systemie pozycyjnym. W związku z tym potrzebne są wydajne metody, które pozwolą rywalizować systemowi resztowemu z systemem pozycyjnym. Przykładem może być operacja określenia znaku, która jest kosztowna w systemie resztowym, ale możemy ograniczyć jej wykorzystywanie przez przechowywanie informacji o znaku liczby i dedukcję znaku wyniku danej operacji. Wiele z tych problemów może zostać rozwiązane przez konwersję z RNS do MRN, która jest stosunkowo prostą operacją. Takie rozwiązanie często jest widoczne w patencie[1].

Operacja mnożenia nie jest ograniczona jedynie do liczb całkowitych. Możemy z powodzeniem wykonywać mnożenie na reprezentacjach stałoprzecinkowych. Aby wykonywać je efektywnie możemy wykorzystać konwersję do MRN i konwersję odwrotną, które pozwalają na wydajne znormalizowanie wyniku opisane w równaniu 3.1

Bibliografia

[1] E. B. Olsen, *Residue number arithmetic logic unit*. July 14 2015, US Patent 9,081,608.

[2] E. B. Olsen, *System and method for improved fractional binary to fractional residue converter and multipler*. July 18 2017, US Patent 9,712,185.

[3] Piotr Patronik, Stanisław J. Piestrak, *Hardware/Software Approach to Designing Low-Power RNS-Enhanced Arithmetic Units*. May 2017, ieee on circuits and systems-i: regular papers, vol. 64, no. 5.

[4] Daniel Anderson, *Design and implementation of an instruction set architecture and an instruction execution unit for the rez9 coprocessor system*. 2011, Bachelor of Science in Computer Engineering University of Nevada Las Vegas.

[5] mgr inż. Jakub Piotr Olszyna, *Analiza i projektowanie układów kryptograficznych przeznaczonych do sieci czujnikowych*. Warszawa 2013, Politechnika Warszawska Wydział Elektroniki i Technik Informacyjnych.

[6] *wikipedia.org/wiki/Systemresztowy*

[7] *michalp.net/open/AK/resztowe11.pdf*

[8] *https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/*