

Rozszerzenie Programowe RNS Procesora x86

Mnożenie Montgomeryego

Jan Jakub Jurec, Student, PWR, Filip Toruń, Student, PWR

Streszczenie—Kryptografia jest jedną z najszybciej rozwijających się i najbardziej kluczowych dziedzin informatyki. Skuteczne zaszyfrowanie danych pozwala na bezpieczne ich przechowywanie oraz wymianę choćby w systemach bankowych czy prywatnych rozmowach. Do najbezpieczniejszych systemów kryptograficznych należą kryptosystemy asymetryczne, wśród których bardzo popularne są algorytmy RSA i Diffiego-Hellmana. Oba oparte są na operacjach modularnej na wielkich liczbach, która jest niezwykle kosztowna. Ten artykuł pokazuje drogę, jaką przeszli autorzy, by zbliżyć się do zrozumienia działania mnożenia Montgomeryego - algorytmu, który znacznie zmniejsza koszt obliczeń w arytmetyce modulo.

Słowa kluczowe—System Resztowy, Chińskie Twierdzenie o Resztach, Systemy Liczbowe, Arytmetyka Komputerowa, Organizacja i Architektura Komputerów, Piotr Patronik Projekt, Mnożenie Montgomeryego, Kryptografia, Optymalizacja

1 WSTĘP

Aby zrozumieć działanie mnożenia Montgomeryego należy najpierw poznać własności systemu resztowego i zrozumieć, jak przebiegają w nim operacje. W tym celu autorzy zaimplementowali konwersję w przód i w tył, używając chińskiego twierdzenia o resztach. Później napisali również dodawanie, odejmowanie, mnożenie oraz dzielenie przez potęgę liczby 2 w systemie resztowym dla liczby 32-bitowej w systemie modułów 7, 15, 31, 127, 8192. Rezultat tych zmagani, które zajęły większą część czasu projektu udostępniają autorzy w Dodatku A. Po zapoznaniu się z arytmetyką modularną autorzy przystąpili do implementacji algorytmu mnożenia Montgomeryego w samodeskryptywnym języku skryptowym Python. Wynikowy kod udostępniają w Dodatku B. Własności mnożenia w systemie resztowym zauważone przez Piotra Montgomeryego pozwalają na drastyczne przyspieszenie mnożenia a tym samym podnoszenia do potęgi w tymże systemie. Dodatkowo, odpowiednio dobierając parametry systemu modulo, można jeszcze bardziej usprawnić obliczenia wykorzystując natychmiastowość dzielenia oraz uzyskiwania reszty z dzielenia przez potęgę liczby 2 w komputerach ogólnego zastosowania.

2 MNOŻENIE MONTGOMEREGO

Warty ponownego podkreślenia jest fakt, że produktem mnożenia Montgomeryego jest liczba w systemie modulo n . Wszystkie operacje przedstawione w tym rozdziale odbywają się właśnie w takim systemie. Należy porzucić intuicję wyniesioną z arytmetyki liczb wymiernych. Algorytm Montgomeryego działa dla dowolnie wielkich liczb naturalnych. Operuje on bowiem na słowach liczby. Dla przykładu: liczba 256-bitowa może składać się z 8 słów 32-bitowych albo 4 64-bitowych. Wynikiem mnożenia Montgomeryego jest produkt Montgomeryego o następującym zapisie:

$$MonPro(a, b) = abr^{-1} \bmod n$$

Zdaniem autorów przed objaśnieniem wprowadzonych symboli warto wyjaśnić, że r^{-1} nie oznacza odwrotności liczby w sensie wymiernym a raczej liczbę, która po przemnożeniu

przez r da resztę równą 1 po podzieleniu przez n . Jest to odwrotność multiplikatywna. W ścisłym zapisie matematycznym:

$$rr^{-1} = 1 \pmod{n}$$

Wprowadzona liczba n oznacza podstawę systemu resztowego a r arbitralnie jest dobranym dodatkowym parametrem. Ważne, żeby a , b , r i n spełniały poniższe dodatkowe założenia:

$$\begin{aligned} a, b &< n \\ nwd(n, r) &= 1 \\ ab &< r \end{aligned}$$

Aby dodatkowo usprawnić działanie algorytmu założono, że:

$$\begin{aligned} r &= 2^k \\ 2^{k-1} &\leq n < 2^k \end{aligned}$$

Spełnienie pierwszego równania zapewni szybkość wykonania operacji z użyciem r , drugiego natomiast spowoduje, że możliwe będzie używanie względnie dużych a i b . Przed wprowadzeniem nowych symboli zostanie wreszcie przedstawiony algorytm mnożenia Montgomeryego w trzech krokach.

$$\begin{aligned} t &= ab \\ u &= \frac{t + (tn' \bmod r)n}{r} \\ u \geq n &? r = u - n : r = u \end{aligned}$$

Jasnym jest, że obliczenie t w kroku pierwszym przyspieszy późniejsze obliczenia. Krok drugi natomiast praktycznie oblicza produkt z r^{-1} w $(\bmod n)$. Ostatni krok tak naprawdę wykonuje działanie oblicza już ostatnią resztę z dzielenia przez n poprzez odjęcie n od ewentualnie za dużego wyniku. Nowym wprowadzonym do działań symbolem jest n' . Jest to liczba spełniająca równanie:

$$rr^{-1} - nn' = 1 \pmod{n}$$

a więc:

$$n' = \frac{1}{r-n} \pmod{n}$$

Oznacza to tyle, że n' jest odwrotnością multiplikatywną liczby $r-n$ w (\pmod{n}) .

Przedstawione implementacje mnożenia Montgomeryego korzystają z powyższych kroków. Metoda Oddzielnego Skanowania Operandów (en. SOS - Separated Operand Scanning) wszystkie kroki wykonuje kaskadowo, po sobie. Jednak metoda Zgrubnie Zintegrowanego Skanowania Operandów (en. CIOS - Coarsely Integrated Operand Scanning) łączy pierwsze dwa kroki, dzięki czemu zajmuje mniej miejsca w pamięci oraz wymaga wykonania mniejszej ilości instrukcji. W żadnej implementacji nie jest użyte r^{-1} . Jest ono jednak potrzebne do sprawdzenia poprawności wyniku na przykład w programie gp.

3 IMPLEMENTACJA

3.1 Metoda Separated Operand Scanning

Pierwszą metodą analizowaną przez autorów była metoda Separated Operand Scanning. Postępuje ona zgodnie z przedstawionym powyżej schematem i dzieli się na 3 kroki. Pierwszy krok wygląda następująco:

```
for i=0 to s-1
  C := 0
  for j=0 to s-1
    (C,S) := t[i+j] + a[j]*b[i] + C
    t[i+j] := S
  t[i+s] := C
```

Przy czym dodać trzeba, że tablica t musi być wyzerowana, będzie mieć ona długość $2s$ słów.

Krok drugi:

```
for i=0 to s-1
  C := 0
  m := t[i]*n'[0] mod W
  for j=0 to s-1
    (C,S) := t[i+j] + m*n[j] + C
    t[i+j] := S
  ADD (t[i+s], C)
for j=0 to s
  u[j] := t[j+s]
```

Tutaj produktem jest tablica u o długości $s+1$ słów. Warty nadmienienia jest, że wykorzystany tu został $n'[0] = n' \pmod{2^w}$ - najmniej znaczący bit odwrotności multiplikatywnej $z r-n$. Funkcja ADD propaguje zadane przeniesienie na bardziej znaczące słowa.

Ostatni krok powtarza się w obu omawianych metodach. I jest to zwykłe porównanie u z n i zwrócenie $u-n$ gdy u jest większe, a w przeciwnym wypadku zwrócenie po prostu u .

```
B := 0
for i=0 to s-1
  (B,D) := u[i] - n[i] - B
```

```
t[i] := D
(B,D) := u[s] - B
t[s] := D
if B = 0 then return t[0], t[1], ..., t[s-1]
else return u[0], u[1], ..., u[s-1]
```

Ten krok zwraca już wynik.

3.2 Metoda Coarsely Integrated Operand Scanning

Druga metoda integruje krok pierwszy oraz drugi, dzięki czemu zmniejszone jest zapotrzebowanie na pamięć. Integracja ta polega na wykorzystaniu zewnętrznej pętli do obu operacji - mnożenia oraz redukcji. Jest to możliwe ponieważ $m[i]$ zależy jedynie od $t[i]$.

```
for i = 0 to s - 1
  C := 0
  for j = 0 to s - 1
    (C,S) := t[j] + a[j]b[i] + C
    t[j] := S
  (C,S) := t[s] + C
  t[s] := S
  t[s + 1] := C
  C := 0
  m := t[0]n'[0] mod W
  for j = 0 to s - 1
    (C,S) := t[j] + mn[j] + C
    t[j] := S
  (C,S) := t[s] + C
  t[s] := S
  t[s + 1] := t[s + 1] + C
  for j = 0 to s
    t[j] := t[j + 1]
```

Znowu tablica t musi być wyzerowana na starcie. Tutaj można już zastosować krok 3 na tablicy t by otrzymać wynik.

4 PODSUMOWANIE

Autorzy są zdania, że zrealizowali cel projektu. Poprawnie zaimplementowali dwa algorytmy mnożenia Montgomeryego (w tym jeden bardzo wydajny czasowo i pamięciowo) dla liczb składających się z dowolnej ilości słów dowolnej wielkości. Zrozumiawszy sztuczki matematyczne użyte w przykładowych implementacjach dokonanych przez Koca i Acara rozsądnym czasie zdołaliby zaimplementować kolejne algorytmy.

Dobrym pomysłem było podzielenie projektu na dwie części. Łagodne wprowadzenie do systemów resztowych zrealizowane poprzez implementację prostych algorytmów dało autorom podstawowe obycie i intuicję podczas analizowania algorytmów przedstawionych przez ww. badaczy.

Jest pocieszającym fakt, że dzięki pokazanym algorytmom można znacznie przyspieszyć mnożenie a przez to podnoszenie do potęgi w systemach modulo nawet na niededykowanym do tego sprzęcie elektronicznym - na przykład komputerach osobistych. Dzięki naukowcom takim jak Piotr Montgomery kryptografia i bezpieczeństwo danych nie są domeną wybranych, których stać na specjalistyczny hardware a stają się dostępne dla wszystkich. Bez wątpienia w konsekwencji zwiększa to spokój wewnętrzny i komfort ogółu z informatyzowanego świata.

Dodatek A

Podstawowe operacje RNS w ASM AT&T

```

shl $29, %rdx
mov %rdx, %rbx

.data
EXIT_SUCCESS=0
SYSEXIT=60

value_rns:
# 123456 100 0110 01110 0001100 0001001000000
.quad 0x8ce18240
value_pos:
.quad 123456

# 2^3 - 1 (for relative primarity)
# 2^4 - 1
# 2^5 - 1
# 2^7 - 1
# 2^13 - 1
m1: .quad 7
m2: .quad 15
m3: .quad 31
m4: .quad 127
m5: .quad 8192

# 7 * 15 * 31 * 127 * 8192
M: .quad 3386449920

# 3386449920 / 7
# 3386449920 / 15
# 3386449920 / 31
# 3386449920 / 127
# 3386449920 / 8192
M1: .quad 483778560
M2: .quad 225763328
M3: .quad 109240320
M4: .quad 26664960
M5: .quad 413385

# multiplicative inversions
y1: .quad 6
y2: .quad 2
y3: .quad 7
y4: .quad 54
y5: .quad 2937

# Convert positional system number
# to RNS number
# RAX(rns) = ARG(pos)
.macro rns pos_num
push %rbx
push %rcx
push %rbx
push %r9

mov \pos_num, %r9
xor %rbx, %rbx

mov %r9, %rax
xor %rdx, %rdx
mov $7, %rcx
div %rcx

mov %r9, %rax
mul %r11
mov y1, %rbx
mul %rbx

# Convert RNS number to positional
# system number
# RAX(pos) = ARG(rns)
.macro drns rns_num
push %r11
push %r8
push %r9
push %rbx
push %rdx
push %rcx

mov \rns_num, %r9

mov %r9, %rax
shr $29, %rax
and $7, %rax
mov %rax, %r11

```

```

mov %rax, %r8
mov %r9, %rax
shr $25, %rax
and $15, %rax
mov %rax, %r11

mov M2, %rax
mul %r11
mov y2, %rbx
mul %rbx
add %rax, %r8

mov %r9, %rax
shr $20, %rax
and $31, %rax
mov %rax, %r11

mov M3, %rax
mul %r11
mov y3, %rbx
mul %rbx
add %rax, %r8

mov %r9, %rax
shr $13, %rax
and $127, %rax
mov %rax, %r11

mov M4, %rax
mul %r11
mov y4, %rbx
mul %rbx
add %rax, %r8

mov %r9, %rax

and $8191, %rax
mov %rax, %r11

mov M5, %rax
mul %r11
mov y5, %rbx
mul %rbx
add %rax, %r8

mov M, %rbx
xor %rdx, %rdx
mov %r8, %rax
div %rbx
mov %rdx, %rax

pop %rcx
pop %rdx
pop %rbx
pop %r9

pop %r8
pop %r11
.endm

# Add two RNS numbers.
# One in RAX, other as ARG.
# RAX = RAX + ARG
.macro addrns rns_num
push %rbx
push %rcx
push %rdx
push %r9
push %r10
push %r11
push %r12

mov \rns_num, %r12
mov %rax, %r9
mov %r12, %rax
xor %r11, %r11
shr $29, %rax
and $7, %rax
mov %rax, %r10
mov %r9, %rax
shr $29, %rax
and $7, %rax
add %r10, %rax
mov $7, %rbx
xor %rdx, %rdx
div %rbx
shl $29, %rdx
or %rdx, %r11

mov %r12, %rax
shr $25, %rax
and $15, %rax
mov %rax, %r10
mov %r9, %rax
shr $25, %rax
and $15, %rax
add %r10, %rax
mov $15, %rbx
xor %rdx, %rdx
div %rbx
shl $25, %rdx
or %rdx, %r11

mov %r12, %rax
shr $20, %rax
and $31, %rax
mov %rax, %r10
mov %r9, %rax
shr $20, %rax
and $31, %rax
add %r10, %rax
mov $31, %rbx
xor %rdx, %rdx
div %rbx
shl $20, %rdx
or %rdx, %r11

```

```

    mov %r12, %rax
    shr $13, %rax
    and $127, %rax
    mov %rax, %r10
    mov %r9, %rax
    shr $13, %rax
    and $127, %rax
    add %r10, %rax
    mov $127, %rbx
    xor %rdx, %rdx
    div %rbx
    shl $13, %rdx
    or %rdx, %r11

    mov %r12, %rax
    and $8191, %rax
    mov %rax, %r10
    mov %r9, %rax
    and $8191, %rax
    add %r10, %rax
    mov $8192, %rbx
    xor %rdx, %rdx
    div %rbx
    or %rdx, %r11

    mov %r11, %rax

    pop %r12
    pop %r11
    pop %r10
    pop %r9
    pop %rdx
    pop %rcx
    pop %rbx
.endm

# Multiple two RNS numbers.
# One in RAX, other as ARG.
# RAX = RAX * ARG
.macro mulrns rns_num
    push %rbx
    push %rcx
    push %rdx
    push %r9
    push %r10
    push %r11
    push %r12

    mov \rns_num, %r12
    mov %rax, %r9
    mov %r12, %rax
    xor %r11, %r11
    shr $29, %rax
    and $7, %rax
    mov %rax, %r10
    mov %r9, %rax
    shr $29, %rax
    and $7, %rax
    mul %r10
    mov $7, %rbx
    xor %rdx, %rdx

    div %rbx
    shl $29, %rdx
    or %rdx, %r11

    mov %r12, %rax
    and $8191, %rax
    mov %rax, %r10
    mov %r9, %rax
    and $8191, %rax
    mul %r10
    mov $8192, %rbx
    xor %rdx, %rdx
    div %rbx
    or %rdx, %r11

    mov %r11, %rax

    pop %r12
    pop %r11

```

```

    pop %r10
    pop %r9
    pop %rdx
    pop %rcx
    pop %rbx
.endm

.macro shr_rns_step
    push %rbx
    push %rcx
    push %rdx
    push %r9
    push %r10
    push %r11

    xor %rdx, %rdx
    xor %r11, %r11

    mov %rax, %r9
    mov %rax, %r8

    shr $29, %rax
    and $7, %rax
    mov %rax, %r10
    shr %r10
    and $1, %rax
    shl $2, %rax
    or %r10, %rax
    shl $29, %rax
    or %rax, %r11

    mov %r9, %rax

    shr $25, %rax
    and $15, %rax
    mov %rax, %r10
    shr %r10
    and $1, %rax
    shl $3, %rax
    or %r10, %rax
    shl $25, %rax
    or %rax, %r11

    mov %r9, %rax

    shr $20, %rax
    and $31, %rax
    mov %rax, %r10
    shr %r10
    and $1, %rax
    shl $4, %rax
    or %r10, %rax
    shl $20, %rax
    or %rax, %r11

    mov %r9, %rax

    shr $13, %rax
    and $127, %rax
    mov %rax, %r10
    shr %r10
    and $1, %rax
    shl $6, %rax
    or %r10, %rax
    shl $13, %rax
    or %rax, %r11

    mov %r11, %rax

    pop %r11
    pop %r10
    pop %r9
    pop %rdx
    pop %rcx
    pop %rbx
.endm

.macro shr_rns positions
    push %rsi
    mov \positions, %rsi

filip_tribute:
    cmp $0, %rsi
    jle exit_shr_rns

    shr_rns_step
    dec %rsi
    jmp filip_tribute

exit_shr_rns:
    pop %rsi
.endm

# Compare two RNS numbers.
# One in RAX, other as ARG.
# If RAX bigger -> RAX = 1,
# If RAX smaller -> RAX = -1,
# If equal -> RAX = 0
.macro cmprns rns_num
    push %rbx

    drns %rax
    mov %rax, %rbx
    mov \rns_num, %rax
    drns %rax
    cmp %rax, %rbx
    jl arg_greater
    je both_equal

rax_greater:
    mov $1, %rax
    jmp leave_cmprns

```

```

arg_greater:
    mov $-1, %rax
    jmp leave_cmprns

both_equal:
    xor %rax, %rax

leave_cmprns:
    pop %rbx
.endm

.text
.global main
main:
    movq %rsp, %rbp

rns_check:
    rns $128

    shr_rns $2

    drns %rax

exit:
    movq $SYSEXIT, %rax
    movq $EXIT_SUCCESS, %rdi
    syscall

Dodatek B
Mnożenie Montgomeryego SOS i CIOS w Pythonie

#return array of base-system words
def radix(x, base):
    digits = []

    while x:
        digits.append(int(x % base))
        x /= int(base)

    digits.reverse()
    return digits

# ax + by = gcd(a, b)
def egcd(a, b):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while b != 0:
        q, a, b = a // b, b, a % b
        x0, x1 = x1, x0 - q * x1
        y0, y1 = y1, y0 - q * y1
    return a, x0, y0

# Multiplicative inverse 1/d mod n
def mulinv(d, n):
    g, x, _ = egcd(d, n)
    if g == 1:
        return x % n

# Propagate carry
def ADD(t, i, C, W):
    for j in range(i, len(t)):
        if C:
            d_w = t[j] + C
            t[j] = d_w % W
            C = d_w // W
        else:
            return t

#Mon. Mul. Separate Operand Scanning
def MonProSOS(a, b, s, w, n):
    k = s * w
    W = 2 ** w
    r = 2 ** k
    n_p = mulinv(r-n, r)
    n0 = n_p % W
    aT = radix(a, W)[-s:][::-1]
    bT = radix(b, W)[-s:][::-1]
    nT = radix(n, W)[-s:][::-1]

    # Step 1
    t = [0]*(2*s)
    for i in range(s):
        C = 0
        for j in range(s):
            d_w = t[i+j] + aT[j] * bT[i] + C
            C, S = d_w // W, d_w % W
            t[i+j] = S
        t[i+s] = C

    # Step 2
    t = t + [0]
    C2 = 0
    for i in range(s):
        C = 0
        m = (t[i] * n0) % W
        for j in range(s):
            d_w = t[i+j] + m * nT[j] + C
            C, S = d_w // W, d_w % W
            t[i+j] = S

        t = ADD(t, i+s, C, W)
    u = t[s:]

    # Step 3
    B = 0
    for i in range(s):
        d_w = u[i] - nT[i] - B
        B, D = d_w // W, d_w % W
        t[i] = D
    d_w = u[s] - B
    B, D = d_w // W, d_w % W
    t[s] = D
    if B:
        cT = t[:s]
    else:
        cT = u[:s]
    c = 0
    for i, w in enumerate(cT):
        c += w*W**i
    return c

```

```

def MonProCIOS(a, b, s, w, n):
    k = s * w
    W = 2 ** w
    r = 2 ** k
    n_p = mulinv(r-n, r)
    n0 = n_p % W
    aT = radix(a, W)[-s:][: -1]
    bT = radix(b, W)[-s:][: -1]
    nT = radix(n, W)[-s:][: -1]

    t = [0]*(s+2)
    # Steps 1 and 2
    for i in range(s):
        C = 0
        for j in range(s):
            d_w = t[j] + aT[j]*bT[i] + C
            C, S = d_w // W, d_w % W
            t[j] = S
        d_w = t[s] + C
        C, S = d_w // W, d_w % W
        t[s] = S
        t[s + 1] = C
        C = 0
        m = (t[0] * n0) % W
        for j in range(s):
            d_w = t[j] + m * nT[j] + C
            C, S = d_w // W, d_w % W
            t[j] = S
        d_w = t[s] + C
        C, S = d_w // W, d_w % W
        t[s] = S
        t[s + 1] = t[s + 1] + C
        for j in range(s+1):
            t[j] = t[j+1]

    u = t

    # Step 3
    B = 0
    for i in range(s):
        d_w = u[i] - nT[i] - B
        B, D = d_w // W, d_w % W
        t[i] = D
    d_w = u[s] - B
    B, D = d_w // W, d_w % W
    t[s] = D
    if B:
        cT = t[:s]
    else:
        cT = u[:s]

    # Compute value of result
    c = 0
    for i, w in enumerate(cT):
        c += w * W ** i
    return c

```

```

if __name__ == '__main__':
    a = 100
    b = 240
    s = 4
    w = 4
    n = 33533
    c = MonProSOS(a, b, s, w, n)
    c = MonProCIOS(a, b, s, w, n)

```

Podziękowanie

Jan Jurec pragnie podziękować swojej narzeczonej Katarzynie za nieustanne wsparcie w uprawianiu nauki, pomoc w ciężkich chwilach i pyszne obiady, które dostarczają mu energii i motywacji do zgłębiania tajników organizacji i arytmetyki komputerów. Filip Toruń natomiast dziękuje Dr Żołnierkowi, za to, że zaszczepił w nim pasję poznawania i niechęć do prostych rozwiązań. Obaj zaś dziękują Doktorowi Piotrowi Patronikowi, który dzielił się swoją wiedzą i skutecznie motywował do udanego przeprowadzenia projektu.

Literatura

- [1] C. K. Koc and T. Acar, Analyzing and Comparing Montgomery Multiplication Algorithms, IEEE Micro, 16(3):26-33, June 1996.



Jan Jakub Jurec jest po raz trzeci studentem trzeciego roku Politechniki Wrocławskiej wydziału Elektroniki kierunku Informatyka. Marzeniem autora jest zdanie kursu Organizacja i Architektura Komputerów za trzecim podejściem, jako że kieruje się zasadą "do trzech razy sztuka"! E-mail: jurec@protonmail.com



Filip Toruń jest studentem Politechniki Wrocławskiej wydziału Elektroniki kierunku Informatyka. Jego zainteresowania to technologie mobilne, obliczenia binarne, atlasy geograficzne. E-mail: 209428@student.pwr.edu.pl