Prowadzący	dr inż. Dominik Żelazny
Grupa	
Termin zajęć	Czwartek 17:05 TN

Kalkulator wykonujący obliczenia w systemie resztowym.

13 czerwca 2021

Spis treści

1 Temat

1.1 Treść

Kalkulator w systemie resztowym z ograniczeniem do liczb long long w języku asemblera. Implementacja przekształcenia na RNS, dodawania, odejmowania, mnożenia oraz ponownej konwersji na system dziesiętny.

1.2 Założenia

- Kalkulator pobiera od użytkownika liczby, konwertuje je na system RNS, dokonuje w nim obliczeń, a wynik po konwersji odwrotnej podaje w systemie dziesiętnym.
- Działania: dodawanie, odejmowanie, mnożenie.
- Kalkulator obsługuje liczby ujemne.
- Wykorzystaliśmy język asemblera w architekturze 64-bitowej i składni AT&T w połączeniu z językiem C. Główne funkcje jak znajdowanie zakresu dynamicznego N, konwersje oraz działania są napisane w assemblerze, natomiast szkielet programu w języku C.

2 Podstawy teoretyczne oraz wybrane metody

2.1 RNS

W systemie resztowym liczba naturalna a jest reprezentowana jako reszty a_i *modulo* m_i gdzie m_i są pairwise coprime, czyli w zbiorze N każda para liczb musi być względnie pierwsza, a ich produkt jest równy N, czyli $N = m_1 * m_2 * ... * m_i$. Produkt N określa zakres w jakim może znajdować się liczba a tj. $0 \leqslant a \leqslant N$.

2.2 Konwersja INT do RNS

Pierwszym krokiem konwersji było znalezienie zakresu dynamicznego N oraz m_i składające się na niego. Strategię jaką podjęliśmy w celu znalezienia zakresu polegała na wymnażaniu kolejnych liczb pierwszych do momentu aż ich iloczyn będzie większy od liczby, którą chcemy zapisać w systemie RNS. Niestety to podejście okazało się mało efektywne. Tak otrzymany iloczyn często był nawet pięciokrotnie większy od liczby zamienianej. Powodowało to problemy z przekroczeniem zakresu unsigned long long, w którym był przechowywany iloczyn.

Usprawnieniem pomysłu okazało się „spowolnienie" wzrostu iloczynu. Zamiast kolejnej liczby pierwszej, algorytm sprawdza czy istnieje wcześniejsza liczba pierwsza, która podniesiona do potęgi jest mniejsza od kolejnej. Dla przykładu: w ciągu $2, 3, 5$ zamiast brania 7 jako kolejnej wartości, brana była 2^2 i ciąg wyglądał wtedy następująco: $2^2, 3, 5$. Kolejnym krokiem optymalizacyjnym była próba usunięcia liczb nadmiarowych. Zastosowaliśmy tutaj podobną strategię jak w przypadku zapełniania zbioru. W przypadku gdy produkt N był większy od liczby zamienianej usuwaliśmy potęgi kolejnych liczb, począwszy od najmniejszej. W przypadku gdy liczba była w pierwszej potędze zostawała ona usuwana ze zbioru.

Ostatnim krokiem było podzielenie liczby przez kolejne wartości m_i oraz zapisanie otrzymanych reszt z dzielenia w tablicy.

2.3 Konwersja RNS do INT

Wybraną przez nas metodą konwersji powrotnej jest wyprowadzenie wagi pozycji dla RNS na podstawie Chińskiego twierdzenia o resztach (CRT).

Na przykładzie:
Rozważając konwersję $y = (3|2|4|2)_{RNS}$ z RNS(8|7|5|3) na system dzisiętny.
Na podstawie właściwości RNS możemy pisać:

$$(3|2|4|2)_{RNS} = (3|0|0|0)_{RNS} + (0|2|0|0)_{RNS}$$
$$+(0|0|4|0)_{RNS} + (0|0|0|2)_{RNS} = 3(1|0|0|0)_{RNS}$$
$$+2(0|1|0|0)_{RNS} + 4(0|0|1|0)_{RNS} + 2(0|0|0|1)_{RNS}$$

Zatem znając wartości następujących czterech stałych (wagi pozycji RNS) pozwoliłoby nam przekonwertować dowolną liczbę z RNS(8|7|5|3) na dziesiętną przy użyciu czterech mnożeń i trzech dodawań.

$$(1|0|0|0)_{RNS} = 105$$
$$(0|1|0|0)_{RNS} = 120$$
$$(0|0|1|0)_{RNS} = 336$$
$$(0|0|0|1)_{RNS} = 280$$

Tak więc znajdujemy:
$(3|2|4|2)_{RNS} = \{(3 * 105) + (2 * 120) + (4 * 336) + (2 * 280)\}_{840} = 779$
Pozostaje tylko pokazać, w jaki sposób wyprowadzono poprzednie wagi. Jak na przykład to zrobiono dla $w3 = (1|0|0|0)_{RNS} = 105$? Aby określić wartość $w3$, zauważamy, że jest ona podzielna przez 3, 5 i 7, ponieważ jej ostatnie trzy reszty to 0. Stąd $w3$ musi być wielokrotnością 105. Następnie musimy wybrać odpowiednią wielokrotność 105 tak, że jego reszta z dzielenia przez 8 wynosi 1.

2.4 Liczby ujemne

Ze względu na równość:

$$-x \; mod \; m_i = (N - x) \; mod \; m_i$$

wszystkie dostępne w zakresie dynamicznym wartości mogą być użyte do reprezentowania liczb np. dla 840: od 0 do 839, od -420 do +419 lub dowolny inny przedział 840 kolejnych liczb całkowitych. W efekcie liczby ujemne są reprezentowane za pomocą dopełniacza ze stałą dopełnienia N.
Biorąc pod uwagę reprezentację x w RNS, reprezentację $-x$ można znaleźć dopełniając każdą z cyfr x_i w odniesieniu do jej modułów m_i.

2.5 Działania

Skoro każda liczba jest zdefiniowana przez dwa zbiory to działania wyglądają następująco:
Mamy dwie liczby $a = \{a_1, a_2, ..., a_i\}$ i $b = \{b_1, b_2, ..., b_i\}$, oraz $m = \{m_1, m_2, ..., m_i\}$, to

$$c_i = a_i \; ? \; b_i \; mod \; (m_i)$$

gdzie:
? - znak dodawania +, odejmowania -, mnożenia *
c_i - wynik działania zapisywany do c.
W dodawaniu, jeśli

$$c \geqslant N$$

to

$$c \leftarrow c - N$$

W odejmowaniu, jeśli

$$c < 0$$

to

$$c \leftarrow c + N$$

W mnożeniu

$$c = c \; mod \; (m_i)$$

3 Implementacja

Rozwijanie programu polegało na pisaniu poszczególnych funkcji w języku C, testowaniu i późniejszym przekodowaniu na assemblera.
Przykładem takiej implementacji jest funkcja obliczająca produkt, czyli zakres dynamiczny liczby. Pierwowzór w C:

```
long long number;
unsigned long long produkt = 1L;

int i = 0, done = 0;
for(i; i < 18; i++) {
    if(produkt >= number)
        break;
    done = 0;
    for(int j = 0; j < i; j++){
        long long cur = primeNumber[j];
        while(cur <= primeNumber[i]){
            cur *= primeNumber[j];
        }
        cur /= primeNumber[j];
        if (cur != primeNumber[j] && cur != N[j]){
            produkt /= N[j];
            N[j] = cur;
            produkt *= N[j];
            i--;
            done = 1;
            if(produkt >= number){
                break;
            }
        }
    }

    if(done == 0){
        N[i] = primeNumber[i];
        produkt *= primeNumber[i];
        if(produkt >= number){
            break;
        }
    }
}

for(int k = 0 ; k < i; k++) {
    if(N[k] != primeNumber[k]){
        produkt /= primeNumber[k];
        if(produkt < number){
            produkt *= primeNumber[k];
```

```
                break;
            }
            N[k] /= primeNumber[k];
            k--;
            continue;
        }
        produkt /= N[k];
        if(produkt >= number){
            N[k] = 0;
            for(int s = k; s < 19; s++){
                N[s] = N[s+ 1];
            }
            continue;
        }
        produkt *= N[k];
    }
```

Oraz docelowa funkcja w assemblerze:

```
.text
.global produkt

# VARIABLES:
# r13 - liczba zamieniana
# r14 - tablica N
# r15 - tablica z liczbami pierwszymi
# r8  - petla pierwsza   [i]
# r9  - boolean
# r10 - petla wewnetrzna [j]
# r11 - produkt
# rbx - buffor

produkt:
    push %rbp
    push %rbx
    push %r12
    push %r13
    push %r14
    push %r15
    mov %rsp, %rbp
    # zapisuje argumenty
    movq %rdi, %r13
    movq %rsi, %r14
    movq %rdx, %r15

    movq $1, %r11   # produkt = 1
    xor %r8, %r8    # i = 0
    petla_zew:                          # petla for(i; i < 18; i++)
        cmpq %r13, %r11                 # if (produkt >= number)
        jae koniec
        xor %r9, %r9                    # done = 0
        xor %r10, %r10                  # j = 0
        petla_wew:
            cmp %r8, %r10               # j < i
            jae else
            movq (%r15, %r10, 8), %rbx  # cur = primeNumber[j];
            movq %rbx, %rax             # cur to rax
            while_pow:                  # while(cur <= primeNumber[i])
```

```
                mul %rbx                #   cur *= primeNumber[j]
                cmpq %rax, (%r15, %r8, 8)
                ja while_pow
            xor %rdx, %rdx
            div %rbx                    # cur /= primeNumber[j];
            inc %r10
            cmpq %rax, %rbx             # cur (rax) != primeNumber[j] (rbx)
            je petla_wew
            dec %r10
            clc
            movq (%r14, %r10, 8), %rcx
            inc %r10
            cmpq %rax, %rcx             # if (cur != N[j])
            je petla_wew
            dec %r10
            xor %rdx, %rdx
            movq %rax, %rbx             # w rax jest cur^ wiec przerzucam go do rbx
            movq %r11, %rax             # produkt do rax
            movq (%r14, %r10, 8), %r12  # wczytuje N[j] do r12
            div %r12                    # produkt /= N[j]
            movq %rbx, (%r14, %r10, 8)  # N[j] = cur
            mul %rbx                    # produkt *= N[j] (cur)
            movq %rax, %r11             # zapisuje wymnozony produkt w r11
            dec %r8
            mov $1, %r9                 # done = 1
            inc %r10                    # j++
            cmp %r13, %r11              # produkt >= number
            jae koniec
            jmp petla_wew
        else:
        inc %r8
        # _if(done == 0)
        cmp $1, %r9
        je petla_zew
        dec %r8
        movq (%r15, %r8, 8), %rbx       # primeNumber[i]
        movq %rbx, (%r14, %r8, 8)       # N[i] = primeNumber[i]
        movq %r11, %rax                 # produkt do rax
        mul %rbx                        # produkt *= primeNumber[i]
        movq %rax, %r11                 # zapisanie wymnozonego produktu w r11
        inc %r8                         # i++
        cmpq %r13, %r11                 # produkt >= number
        jae koniec
        jmp petla_zew
koniec:

# VARIABLES:
# r8  - wartosc 'i' z petli powyzej
# r9  - licznik petli
# r11 - produkt
xor %r9, %r9
petla_wy:                       # for(int k = 0 ; k < i; k++)
    cmp %r8, %r9                # k < i
    jae finally
    movq (%r14, %r9, 8), %rbx   # wczytuje N[k]
    movq (%r15, %r9, 8), %rcx   # wczytuje primeNumber[k]
    cmpq %rbx, %rcx             # if(N[k] != primeNumber[k])
```

```
    je entire                    # jesli nie jest to spelnione to probujemy usunac
                                   cala liczbe a nie tylko potege
    movq %r11, %rax              # produkt do rax
    div %rcx                     # produkt /= primeNumber[k]
    cmpq %r13, %rax              # if(produkt < number)
    jb multiply                  # jesli tak to przywroc produkt i zakoncz
    movq %rax, %r11              # zapisz produkt
    movq (%r14, %r9, 8), %rax    # wczytuje N[k]
    div %rcx                     # N[k] /= primeNumber[k]
    movq %rax, (%r14, %r9, 8)    # zapisuje N[k]
    jmp petla_wy                 # i kolejny obieg

    multiply:
        mul %rcx
        movq %rax, %r11
        jmp finally

    entire:
        movq %r11, %rax              # laduje produkt do rax
        movq (%r14, %r9, 8), %rbx    # wczytuje N[k]
        div %rbx                     # produkt /= N[k]
        inc %r9
        cmpq %r13, %rax              # if (produkt >= number)
        jb petla_wy
        dec %r9
        movq %rax, %r11              # zapisuje produkt
        movq $0, (%r14, %r9, 8)      # N[k] = 0
        movq %r9, %r10
        inc %r9 # zwiekszam licznik petla_wy
        smol:                        # petla for(int s = k; s < 19; s++)
            cmp $19, %r10            # s < 19
            jae petla_wy
            inc %r10
            movq (%r14, %r10, 8), %rbx # wczytuje N[s + 1]
            dec %r10
            movq %rbx, (%r14, %r10, 8) # N[s] = N[s+ 1]
            inc %r10
            jmp smol

finally:
movq %r11, %rax

pop %r15
pop %r14
pop %r13
pop %r12
pop %rbx
pop %rbp
ret
```

4 Użyte narzędzia

- System operacyjny 64-bitowy Linux dystrybucji Ubuntu na maszynie wirtualnej,
- Edytor Visual Studio Code,
- LaTeX,
- System kontroli wersji Git oraz serwis Github.

5 Wnioski

Wykonanie zadania projektowego jakim jest napisanie kalkulatora liczącego w systemie resztowym okazało się być problematyczne ze względu na słabą dostępność klarownych informacji w dostępnych nam źródłach oraz samą złożoność RNS, a także używanie do tego języka asemblera. Wiele pojęć w systemie resztowym jest określanych jako trudne do implementacji. Dlatego działania, które udało nam się zaimplementować to dodawanie, odejmowanie oraz mnożenie. Zostały one poprzedzone implementacją obliczania zakresu dynamicznego liczb oraz konwersji między systemami. Ostatnim krokiem było uwzględnienie w obliczeniach liczb ujemnych. Zapisanie powyższych operacji bezpośrednio w assemblerze było trudne do wykonania, dlatego najpierw tworzyliśmy prototypy funkcji w języku C, by potem przepisać je w składni assemblerowej.

6 Źródła

- Behrooz Parhami: COMPUTER ARITHMETIC Algorithms and Hardware Designs, 2000
- https://pl.linkfang.org/wiki/System_resztowy
- Janusz Biernat: AK1-5-18-Systemy resztowe
- http://neo.dmcs.p.lodz.pl/csII/ca2_ResidueNS.pdf