Laboratorium 1

Pętle, podstawowe operacje logiczne i arytmetyczne

Program 1

- wczyta ze standardowego wejścia podany przez użytkownika łańcuch znaków
- dla każdego znaku sprawdzi, czy jest on wielką/małą literą/innym znakiem i w zależności od tego wykona odpowiednią akcję na tym znaku
  - do kodów ASCII cyfr doda stałą wartość 5,
  - do kodów ASCII wielkich liter doda stałą wartość 3
  - pozostałe znaki pozostawi bez zmian
- wypisze zmieniony łańcuch znaków na ekran

Program 2

- wczyta ze standardowego wejścia podany przez użytkownika łańcuch znaków
- wprowadzony ciąg potraktuje jako reprezentację w systemie siódemkowym
- wypisze na standardowe wyjście reprezentację liczby w systemie dziesiętnym

Laboratorium 2

Utrwalenie umiejętności tworzenia prostych konstrukcji programowych

Program 1

- wczyta z pliku łańcuch znaków dowolnej długości
  - ciąg nie musi zawierać parzystej liczby znaków
  - w pliku może się znajdować do 1024 znaków ASCII
- wczytany ciąg znaków potraktuje jako reprezentację szesnastkową, wpisze do bufora w postaci liczbowej
  - zamiana z ASCII na ciąg heksadecymalny
- zamieni liczbę na reprezentację ósemkową
  - korzystając z własności baz skojarzonych
  - używając operacji logicznych
  - zamiana na ASCII
- zapisze wynik do innego pliku

Program 2

- wczyta ze standardowego wejścia dwa łańcuchy znaków i potraktuje je jako reprezentacje w systemie szesnastkowym
  - użytkownik może podać do 1024 znaków ASCII (dla każdej liczby)
- wpisze ciągi do buforów w postaci liczbowej
  - zamiana z ASCII na ciągi heksadecymalne
- doda liczby do siebie używając rejestrów 64 – bitowych oraz wbudowanej propagacji przeniesień (flaga CF)
- wynik zamieni z powrotem na ASCII i wypisze na standardowe wyjście

FAQ

Jak wpisać liczbę do pamięci, by możliwe było np. dodawanie z użyciem flagi CF?

Jeśli wczytamy ze standardowego wejścia następujący ciąg znaków ASCII:

> 9123456789ABCDEF123456789ABCDEF

to będzie to równoznaczne zadeklarowaniu w sekcji `.data` poniższego łańcucha znaków:

```
string: .ascii „9123456789ABCDEF123456789ABCDEF”
```

czyli pod adresem `string` znajduje się wartość bajtu ‘`9`’ = 0x39.

Jeśli odkodujemy szesnastkową wartość znaków (odejmiemy wartość ‘`0`’ dla cyfr lub ‘`A`’ dla liter), to każdy z tych znaków będzie kodowany na 4 bitach. Aby w przyszłości można było skopiować

8 bajtów danych do rejestru 64b – by np. wykonać optymalne dodawanie wielkich liczb – musimy skleić ze sobą kolejne pary znaków (poczynając od najmłodszych pozycji wpisanej liczby). Zaczniemy więc od końca napisu (‘E’ i ‘F’) i po zamianie na wartości rzeczywiste (`00001110` i `00001111`), skleimy je ze sobą (`11101111`), a następnie otrzymany bajt `0xef` wpiszemy pod adres innego bufora (tu `number`) – bez przesunięcia. Tzn., aby w przyszłości można było wykonać taką operację:

```
movq number(, %rsi, 8), %rax
```

i zobaczyć w debugerze wartość (dla `$rsi = 0`):

```
$rax = 0xf1 23 45 67 89 ab cd ef
```

to zawartość pamięci pod adresem `number` musi wyglądać następująco (adres `number` wskazuje na bajt `0xef`):

```
0xef cd ab 89 67 45 23 f1 de bc 9a 78 56 34 12 09
```

Warto tu poczytać o konwencji *little endian*.

W jaki sposób można wtedy dokonać konwersji wykorzystując własność baz skojarzonych?

Jedną z metod jest grupowanie bajtów po 3 (3B = 24b, 3 | 24), sklejanie ich w jednym rejestrze i wyłuskiwanie kolejnych trójek bitów ($2^3 = 8$). Dla powyższego przykładu pierwszą trójką (zaczynamy od najmłodszych bitów!!!) bajtów będzie `0x ef cd ab`. Wykonując operacje:

#zakładając, że wartość 0xefcdab początkowo znajdowała się w rejestrze %rax

```
movb %al, %bl

andb $0x7, %bl

shrq $3, %rax
```

najpierw tworzymy kopię interesującego fragmentu rejestru `%rax`, następnie wyłuskujemy 3 najmłodsze bity (które są wartością cyfry w systemie ósemkowym – wystarczy zamienić na ASCII), a na końcu przesuwamy bitowo wartość z rejestru `%rax` o 3 bity w prawo. Dzięki ostatniej operacji możemy znów wrócić do pierwszej instrukcji i wyłuskiwać kolejne trójki bitów. Wiemy, że pętla musi się wykonać 8 razy (24/3).

W jaki sposób wyłuskać po 3 bajty?

Można trzykrotnie skopiować po jednym bajcie z pamięci, jedną z tych wartości przesunąć bitowo w lewo o 8 pozycji, jedną o 16 pozycji w lewo, a następnie skleić wszystkie trzy operacją logiczną `or`.

Laboratorium 3

Funkcje

Program 1

- wygeneruje liczby pierwsze według algorytmu sita Eratostenesa
  - zakres 64 bitów – konieczność używania rejestrów o rozmiarze 8B (rezerwujemy mniej pamięci, tak dużych liczb generować nie będziemy…)
- zapisze wygenerowane liczby pierwsze do pamięci
- wypisze wygenerowane liczby pierwsze na ekran
- za pierwsze 2 podpunkty jest odpowiedzialna 1 funkcja

Program 2

- zgodnie z algorytmem Euklidesa obliczy największy wspólny dzielnik dwóch liczb
  - funkcja rekurencyjna
  - argumenty zapisane w pamięci
  - zakres 64 bitów
  - na 2 sposoby – raz argumenty i wynik przekazywane przez rejestry, raz przez stos
- wypisze wynik na standardowe wyjście

Laboratorium 4

Łączenie różnych języków programowania w jednym projekcie

Program 1

- napisany w języku asemblera
  - wywoła funkcje napisane w języku C umieszczone w osobnym pliku – przyjmujące argumenty stało- i zmiennoprzecinkowe oraz zwracające jakiś wynik
  - wywoła funkcje biblioteczne (np. scanf, printf)

Program 2

- napisany w języku C
  - wstawka w języku Asemblera wykorzystująca zmienne i stałe zadeklarowane w języku C

Program 3

- napisany w języku C
  - wywoła funkcje napisane w języku Asemblera, umieszczone w osobnym pliku – przyjmujące argumenty stało- i zmiennoprzecinkowe oraz zwracające jakiś wynik

Laboratorium 5

Jednostka zmiennoprzecinkowa (FPU)

Program 1

- napisany w języku C wywoła następujące funkcje napisane w języku asemblera:
  - funkcja pozwalająca na sprawdzenie precyzji obliczeń
  - funkcja pozwalająca na ustawienie wybranej przez użytkownika precyzji obliczeń
- powinien zawierać przystępny interfejs użytkownika, który w pętli będzie pytał użytkownika o żądaną akcję (sprawdzenie/ustawienie precyzji obliczeń)

Program 2

- napisany w dowolnym języku programowania
- wywoła funkcję napisaną w języku asemblera obliczającą aproksymację funkcji sinus
  - należy wykorzystać rozwinięcie szeregu Taylora
  - funkcja powinna przyjmować następujące argumenty: kąt w radianach oraz liczba kroków (wyrazów ciągu)
- wypisze wynik na standardowe wyjście

Laboratorium 6

Rozszerzenia wektorowe (MMX/SSE)

Program 1

Należy ze strony http://www.zak.ict.pwr.wroc.pl/materials/architektura/laboratorium/MMX/ ściągnąć przykładowy program, w którym zaimplementowano podstawowe operacje na bitmapie. Następnie:

- napisać w języku C funkcję, która dokona prostego przetworzenia bitmapy, np. utworzy negatyw
- napisać w języku asemblera funkcję, która z użyciem rozszerzeń wektorowych zrealizuje tę samą operację
- za pomocą *rdtsc* zmierzyć czas w pierwszym i w drugim przypadku, porównać