

# Architektury systemów komputerowych

## Lista zadań nr 4

Na zajęcia od 24 do 26 marca 2025

W zadaniach 4 – 7 można używać wyłącznie poniższych instrukcji, których semantykę wyjaśniono na stronie [x86 and amd64 instruction reference](#)<sup>1</sup>. Wartości tymczasowe można przechowywać w rejestrach %r8 ... %r11.

- transferu danych: mov cbw/cwde/cdq/cwd/cdq/cqo movzx movsx,
- arytmetycznych: add sub imul mul idiv div inc dec neg,
- logicznych: and or xor not sar shr shl ror rol,
- innych: lea ret.

Przy tłumaczeniu kodu w assemblerze x86-64 do języka C należy trzymać się następujących wytycznych:

- Używaj złożonych wyrażeń minimalizując liczbę zmiennych tymczasowych.
- Nazwy wprowadzonych zmiennych muszą opisywać ich zastosowanie, np. result zamiast rax.

**UWAGA!** Nie wolno korzystać z kompilatora celem podejrzenia wygenerowanego kodu!

**Zadanie 1.** Poniżej podano wartości typu long leżące pod wskazanymi adresami i w rejestrach:

Adres	Wartość	Rejestr	Wartość
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	1
0x110	0x13	%rdx	3
0x118	0x11		

Oblicz wartość poniższych **operandów źródłowych** operacji movq:

- |            |                  |                     |
|------------|------------------|---------------------|
| 1. %rax    | 4. (%rax)        | 7. 0xFC(,%rcx,4)    |
| 2. 0x110   | 5. 8(%rax)       | 8. (%rax,%rdx,8)    |
| 3. \$0x108 | 6. 21(%rax,%rdx) | 9. 265(%rcx,%rdx,2) |

**Zadanie 2.** Każdą z poniższych instrukcji wykonujemy w stanie maszyny opisanym tabelką z poprzedniego zadania. Wskaż miejsce, w którym zostanie umieszczony wynik działania instrukcji, oraz obliczoną wartość.

- |                        |                              |
|------------------------|------------------------------|
| 1. addq %rcx, (%rax)   | 5. decq %rcx                 |
| 2. subq 16(%rax), %rdx | 6. imulq 8(%rax)             |
| 3. shrq \$4, %rax      | 7. leaq 7(%rcx,%rcx,8), %rdx |
| 4. incq 16(%rax)       | 8. leaq 0xA(,%rdx,4), %rdx   |

**Zadanie 3.** W wyniku deasemblacji procedury long decode(long x, long y) otrzymano kod:

```
1 decode: leaq (%rdi,%rsi), %rax
2         xorq %rax, %rdi
3         xorq %rax, %rsi
4         movq %rdi, %rax
5         andq %rsi, %rax
6         shrq $63, %rax
7         ret
```

Zgodnie z [SYSTEM V ABI](#)<sup>2</sup> dla architektury x86-64, argumenty x i y są przekazywane odpowiednio przez rejestry %rdi i %rsi, a wynik zwracany w rejestrze %rax. Napisz funkcję w języku C, która będzie liczyła dokładnie to samo co powyższy kod w assemblerze. Postaraj się, aby była ona jak najbardziej zwięzła.

<sup>1</sup><http://www.felixcloutier.com/x86/>

<sup>2</sup><https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf>

**Zadanie 4.** Zaimplementuj w assemblerze x86-64 procedurę konwertującą liczbę typu `uint32_t` między formatem *little-endian* i *big-endian*. Argument funkcji jest przekazany w rejestrze `%edi`, a wynik zwracany w rejestrze `%eax`. Należy użyć instrukcji cyklicznego przesunięcia bitowego `ror` lub `rol`.

Podaj wyrażenie w języku C, które kompilator optymalizujący przetłumaczy do instrukcji `ror` lub `rol` (i sprawdź czy rzeczywiście tak się stanie).

**Zadanie 5.** Zaimplementuj w assemblerze x86-64 funkcję liczącą wyrażenie  $x * y$ . Argumenty i wynik funkcji są 128-bitowymi liczbami całkowitymi ze znakiem i nie mieszczą się w rejestrach maszynowych. Zatem  $x$  jest przekazywany przez rejestry `%rdi` (starsze 64 bity) i `%rsi` (młodsze 64 bity), analogicznie argument  $y$  jest przekazywany przez `%rdx` i `%rcx`, a wynik jest zwracany w rejestrach `%rdx` i `%rax`.

Instrukcja `mul` wykonuje co najwyżej mnożenie dwóch 64-bitowych liczb i zwraca 128-bitowy wynik. Wiedząc, że  $n = n_{127...64} \cdot 2^{64} + n_{63...0}$ , zaprezentuj metodę obliczenia iloczynu, a dopiero potem przetłumacz algorytm na assembler.

**UWAGA!** Zapoznaj się z dokumentacją instrukcji `mul` ze względu na niejawne użycie rejestrów `%rax` i `%rdx`.

**Zadanie 6.** W rejestrze `%rax` przechowujemy osiem drukowalnych znaków w kodzie ASCII, tj. każdy bajt ma wartość od `0x20` do `0x7f`. Podaj kod w assemblerze x86-64, który małą liczbą instrukcji<sup>3</sup> przepisze w rejestrze `%rax` wszystkie małe litery na duże litery, a pozostałe znaki zostawi niezmienione.

**Źródło:** Zadanie 87 z [1, 7.1.3] jest prostszym wariantem tego zadania. Twój kod powinien zamieniać tylko znaki z przedziału od `0x61` do `0x7A`.

**Zadanie 7.** Liczba w formacie BCD (ang. *binary coded decimal*) jest reprezentowana przez liczbę binarną, której kolejne półbajty (ang. *nibble*) kodują cyfry dziesiętne od 0 do 9. Przykładowo, liczba szesnastkowa `0x2970` oznacza po prostu liczbę 2970. Napisz w assemblerze x86-64 ciało funkcji `bcd_add`, która dodaje dwie 16-cyfrowe liczby przekazane w rejestrach `%rdi` i `%rsi`, a wynik zwróci w rejestrze `%rax`. Nie można używać instrukcji mnożenia i dzielenia.

**Źródło:** Zadanie 100 z [1, 7.1.3].

## Literatura

- [1] „*Art of Computer Programming, Volume 4A, The: Combinatorial Algorithms, Part 1*”  
Donald E. Knuth; Addison-Wesley Professional; 1st edition (January 12, 2011)

---

<sup>3</sup>Twój kod powinien korygować wszystkie znaki równocześnie.