

Architektury systemów komputerowych

Lista zadań nr 4

Na zajęcia 25 marca 2021

W zadaniach 5 – 9 można używać wyłącznie poniższych instrukcji, których semantykę wyjaśniono na stronie [x86 and amd64 instruction reference](#)¹. Wartości tymczasowe można przechowywać w rejestrach %r8 ... %r11.

- transferu danych: mov cbw/cwde/cdq/cwd/cdq/cqo movzx movsx,
- arytmetycznych: add adc sub sbb imul mul idiv div idiv inc dec neg cmp,
- logicznych: and or xor not sar sarx shr shrx shl shlx ror rol test,
- 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.
- Instrukcja goto jest zabroniona. Należy używać instrukcji sterowania if, for, while i switch.
- Jeśli to ma sens pętle while należy przetłumaczyć do pętli for.

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. Zaimplementuj w języku C poniższy algorytm wygładzania wykładniczego (ang. *exponential smoothing*) szeregu $\{x_i\}$ wartości fizycznych (np. temperatury otoczenia) na 32-bitowym procesorze, który nie implementuje przetwarzania liczb zmiennopozycyjnych².

$$s_0 = x_0$$
$$s_i = \alpha \cdot x_i + (1 - \alpha) \cdot s_{i-1}, \quad i > 0$$

Wartości x_i i s_i są przechowywane jako **liczby stałoprzecinkowe** (ang. *fixed point*) w formacie $Q_{10.6}$, tj. z dziesięcioma bitami na część całkowitą i sześcioma na część ułamkową. Dla dokładności stałą $\alpha \in (0, 1)$ zapisano w formacie $Q_{0.16}$. Podaj fragment kodu obliczający α typu `uint16_t` z liczby typu `float` w trakcie kompilacji oraz s_i typu `int16_t` w trakcie uruchomienia programu.

¹<http://www.felixcloutier.com/x86/>

²Dobrym przykładem są popularne mikrokontrolery ARM Cortex-M4 na płytkach rozwojowych **Nucleo**.

Zadanie 4. 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**³ 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 asemblerze. Postaraj się, aby była ona jak najbardziej zwięzła.

Zadanie 5. Zaimplementuj w asemblerze 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».

Zadanie 6. Zaimplementuj w asemblerze 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.

Wskazówka! Użyj instrukcji «adc». Rozwiązanie wzorcowe składa się z 3 instrukcji bez «ret».

Zadanie 7. Zaimplementuj w asemblerze x86-64 funkcję liczącą wyrażenie «x * y». Argumenty i wynik funkcji są 128-bitowymi liczbami całkowitymi bez znaku. Argumenty i wynik są przypisane do tych samych rejestrów co w poprzednim zadaniu. 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 asembler.

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

Zadanie 8. Zaimplementuj poniższą funkcję w asemblerze x86-64. Wartości «x» i «y» typu «uint64_t» są przekazywane przez rejestry %rdi i %rsi, a wynik zwracany w rejestrze %rax. Najpierw rozwiąż zadanie używając instrukcji skoku warunkowego. Potem przepisuj je używając instrukcji «sbb».

$$addu(x, y) = \begin{cases} \text{ULONG_MAX} & \text{dla } x + y \geq \text{ULONG_MAX} \\ x + y & \text{w p.p.} \end{cases}$$

Wskazówka! Rozwiązanie wzorcowe składa się z 3 instrukcji bez «ret».

Zadanie 9. Zaimplementuj funkcję zdefiniowaną poniżej w asemblerze x86-64. Taka procedura w języku C miałaby sygnaturę «long cmp(uint64_t x, uint64_t y)».

$$cmp(x, y) = \begin{cases} -1 & \text{gdy } x < y \\ 1 & \text{gdy } x > y \\ 0 & \text{gdy } x = y \end{cases}$$

Wskazówka: Rozwiązanie wzorcowe ma cztery wiersze (bez ret). Użyj instrukcji adc, sbb i neg.

³<https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>