Architektury systemów komputerowych 2017

Lista zadań nr 4

Na zajęcia 22 i 23 marca 2017

W zadaniach 7 – 9 można używać wyłącznie instrukcji z rozdziałów 5.1.2, 5.1.4 i 5.1.5 książki Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture[1] oraz `mov`, `movzb`, `movsb`, `lea` i `ctoq`. Wartości tymczasowe można przechowywać w rejestrach `%r8` ... `%r11`.

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
0x104	0xAB	%rcx	1
0x108	0x13	%rdx	3
0x10C	0x11		

Oblicz wartość poniższych operandów:

1. `%rax`
2. `0x104`
3. `$0x108`
4. `(%rax)`
5. `4(%rax)`
6. `9(%rax,%rdx)`
7. `260(%rcx,%rdx)`
8. `0xFC(,%rcx,4)`
9. `(%rax,%rdx,4)`

Zadanie 2. Dla każdej z poniższych instrukcji wyznacz odpowiedni sufiks (tj. `b`, `w`, `l` lub `q`) na podstawie rozmiarów operandów:

1. `mov %eax,(%rsp)`
2. `mov (%rax),%dx`
3. `mov $0xFF, %bl`
4. `mov (%rsp,%rdx,4),%dl`
5. `mov (%rdx),%rax`
6. `mov %dx,(%rax)`

Zadanie 3. Rejestry `%rax` i `%rcx` przechowują odpowiednio wartości `x` i `y`. Podaj wyrażenie, które będzie opisywać zawartość rejestru `%rdx` po wykonaniu każdej z poniższych instrukcji:

1. `leaq 6(%rax),%rdx`
2. `leaq (%rax,%rcx),%rdx`
3. `leaq (%rax,%rcx,4),%rdx`
4. `leaq 7(%rax,%rax,8),%rdx`
5. `leaq 0xA(,%rcx,4),%rdx`
6. `leaq 9(%rax,%rcx,2),%rdx`

Zadanie 4. Każdą z poniższych instrukcji wykonujemy w stanie maszyny opisanym tabelką z zadania 1. Wskaż miejsce, w którym zostanie umieszczony wynik działania instrukcji, oraz obliczoną wartość.

1. `addq  %rcx,(%rax)`
2. `subq  16(%rax),%rdx`
3. `imulq $16,(%rax,%rdx,8)`
4. `incq  16(%rax)`
5. `decq  %rcx`
6. `subq  %rdx,%rax`

[1] `https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf`

Zadanie 5. Funkcję o sygnaturze «`long decode2(long x, long y, long z)`» poddano deasemblacji, która dała następujący kod:

```
1 decode2:
2     subq  %rdx,%rsi
3     imulq %rsi,%rdi
4     movq  %rsi,%rax
5     salq  $63,%rax
6     sarq  $63,%rax
7     xorq  %rdi,%rax
8     ret
```

Zgodnie z System V Application Binary Interface[2] dla architektury `x86-64`, argumenty `x`, `y` i `z` są przekazywane odpowiednio przez rejestry `%rdi`, `%rsi` i `%rdx`, a wynik zwracany w rejestrze `%rax`. Napisz funkcję w języku `C`, która będzie liczyła dokładnie to samo co powyższy kod w asemblerze.

Zadanie 6. Zaimplementuj w asemblerze `x86-64` funkcję konwertującą liczbę typu `long` między formatem *little-endian* i *big-endian*. Argument funkcji jest przekazany w rejestrze `%rdi`, a wynik zwracany w rejestrze `%rax`. Należy użyć instrukcji cyklicznego przesunięcia bitowego `ror` lub `rol`.

Podaj wyrażenie w języku `C`, które kompilator może przetłumaczyć do instrukcji `ror` lub `rol`.

Zadanie 7. 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), argument `y` przez `%rdx` i `%rcx`, a wynik jest zwracany w rejestrach `%rax` i `%rdx`.

Jak uprościłby się kod, gdyby można było użyć instrukcji `set` lub `addc`?

Zadanie 8. 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 zwrócić 128-bitowy wynik. Wiedząc, że $n = n_{127\ldots64} \cdot 2^{64} + n_{63\ldots0}$, zaprezentuj metodę obliczenia iloczynu, a dopiero potem przetłumacz algorytm na asembler. Postaraj się opracować metodę, która używa co najwyżej trzech instrukcji mnożenia.

Zadanie 9. Zaimplementuj poniższą funkcję w asemblerze `x86-64`, przy czym wartości `x` i `y` są przekazywane przez rejestry `%rdi` i `%rsi`, a wynik zwracany w rejestrze `%rax`.

$$adds(x, y) = \begin{cases} \texttt{MIN\_INT} & \text{dla } x + y \leq \texttt{MIN\_INT} \\ \texttt{MAX\_INT} & \text{dla } x + y \geq \texttt{MAX\_INT} \\ x + y & \text{w p.p.} \end{cases}$$

Wiemy już, że wyrażenie «`b ? x : y`» można w pewnych warunkach przetłumaczyć do «`b * x + !b * y`», a przy odrobinie sprytu można pozbyć się mnożenia i używać instrukcji `and`.

Jak uprościłby się kod, gdyby można było użyć instrukcji `cmov`?

[2] `http://www.x86-64.org/documentation/abi.pdf`