Architektury systemów komputerowych

Lista zadań nr 4

Na zajęcia 19–20 marca 2018

UWAGA! W składniach `AT&T` i `Intel` występują rozbieżności w nazwach mnemoników – dla rozróżnienia będziemy odpowiednio zapisywać je małymi i dużymi literami.

W zadaniach 4 – 6 można używać wyłącznie instrukcji z rozdziałów 5.1.2, 5.1.4 i 5.1.5 dokumentacji Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture[1] oraz `lea`, `mov`, `movz` (`MOVZX`), `movs` (`MOVSX`) i `cto` (`CDQ` i `CQO`). Wartości tymczasowe można przechowywać w rejestrach `%r8` ... `%r11`. Semantykę instrukcji można znaleźć na stronie http://www.felixcloutier.com/x86.

Przy tłumaczeniu kodu w asemblerze `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:

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

Zadanie 2. 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. `shrq  $4,%rax`
4. `incq  16(%rax)`
5. `decq  %rcx`
6. `imulq 8(%rax)`
7. `leaq  7(%rcx,%rcx,8),%rdx`
8. `leaq  0xA(,%rdx,4),%rdx`

Zadanie 3. Zaimplementuj w asemblerze `x86-64` funkcję 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`.

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

Zadanie 4. 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. Należy użyć instrukcji set! Jak uprościłby się kod, gdyby można było użyć instrukcji adc?

Zadanie 5. 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 $n = n_{127\ldots64} \cdot 2^{64} + n_{63\ldots0}$, 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 6. Zaimplementuj poniższą funkcję w asemblerze x86-64, przy czym wartości «x» i «y» typu «uint64_t» są przekazywane przez rejestry %rdi i %rsi, a wynik zwracany w rejestrze %rax. Jak uprościłby się kod, gdyby można było użyć instrukcji cmov?

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

UWAGA! Według AT&T versus Intel Syntax[2] do ładowania 64-bitowych stałych do rejestru służy instrukcja movabsq.

Zadanie 7. 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 8. Zapisz w języku C funkcję o sygnaturze «int puzzle(long x, unsigned n)», której kod w asemblerze podano niżej. Przedstaw jednym zdaniem co ta procedura robi.

```
1  puzzle: testl %esi, %esi
2          je    .L4
3          xorl  %edx, %edx
4          xorl  %eax, %eax
5  .L3:    movl  %edi, %ecx
6          andl  $1, %ecx
7          addl  %ecx, %eax
8          sarq  %rdi
9          incl  %edx
10         cmpl  %edx, %esi
11         jne   .L3
12         ret
13 .L4:    movl  %esi, %eax
14         ret
```

UWAGA! Wszystkie instrukcje operujące na dolnej połowie 64-bitowego rejestru czyszczą jego górną połowę. Można o tym przeczytać w §3.4.1.1 dokumentacji z nagłówka listy!

[2] https://sourceware.org/binutils/docs/as/i386_002dVariations.html
[3] https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf