Lista 4

tags: ASK

Zadanie 1

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		

1. ***%rax*** => 0x100
2. ***0x110*** => 0x13
3. ***&0x108*** => 0x108
4. ***(%rax)*** => 0xFF traktujemy wartość w `rax` jako adres i wybieramy wartość pod tym adresem
5. ***8(%rax)*** => 0xAB dereferujemy zinkrementowany adres
6. ***21(%rax,%rdx)*** => 0x11 (%rax + %rdx + 0x15) = (0x118)
7. ***0xFC(,%rcx,4)*** => 0xFF (4 * %rcx + 0xFC) = (0x04 + 0xFC) = (0x100)
8. ***(%rax,%rdx,8)*** => 0x11 (%rax + 8 * %rdx) = (%rax + 0x18) = (0x118)
9. ***265(%rcx,%rdx,2)*** => 0x13 (%rcx + 2 * %rdx + 265) = (%rcx + 0x10F) = (0x110)

Zadanie 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.

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

1. ***addq %rcx,(%rax)***
   - miejsce: 0x100
   - wartość: 0x100
2. ***subq 16(%rax),%rdx*** 16(%rax) -> (0x100 + 0x10) = (0x110) = 0x13
   - miejsce: %rdx
   - wartość: 0x03 - 0x13 = -16
3. ***shrq $4,%rax*** *shift logical right*: 0x100 >> 4
   - miejsce: %rax
   - wartość: 0x10
4. ***incq 16(%rax)*** *increment* 16(%rax) -> (0x100 + 0x010) = (0x110) = 0x13
   - miejsce: 0x110
   - wartość: 0x13 + 0x01 = 0x14
5. ***decq %rcx*** *decrement*
   - miejsce: %rcx
   - wartość: 0
6. ***imulq 8(%rax)*** unarne mnożenie *RDX:RAX ← RAX ∗ r/m64* (%rax + 8) -> (0x100 + 0x008) = (0x108) = 0xAB
   - miejsce: RDX:RAX

- wartość: 0x100 * 0xAB = 0xAB00

7. ***leaq 7(%rcx,%rcx,8),%rdx*** 7(%rcx,%rcx,8) -> %rcx + 8 * %rcx + 7 = %rcx + 15 = 0x10
   - miejsce: %rdx
   - wartość: 0x10
8. ***leaq 0xA(,%rdx,4), %rdx*** 0xA(,%rdx,4) -> 4 * 3 + 0xA = 12 + 0xA = 0x16
   - miejsce: %rdx
   - wartość: 0x16

Zadanie 4

W wyniku deasemblacji procedury «`long decode(long x, long y)`» otrzymano kod:

```
decode: leaq (%rdi,%rsi), %rax
        xorq %rax, %rdi
        xorq %rax, %rsi
        movq %rdi, %rax
        andq %rsi, %rax
        shrq $63, %rax
        ret
```

`leaq (%rdi,%rsi), %rax` -> %rax = x+y //sumuje x i y wynik zapisuje w RAX `xorq %rax, %rdi` -> %rdi = x ^ (x+y) `xorq %rax, %rsi` -> %rsi = y ^ (x+y) `movq %rdi, %rax` -> %rax = x ^ (x+y) `andq %rsi, %rax` -> %rax = [(x ^ (x+y)) & (y ^ (x+y))] `shrq $63, %rax` ->%rax = [(x ^ (x+y)) & (y ^ (x+y))] >> 63 //logical right shift zostawia zera na najstarszych bitach

```
long decode(long x, long y) {
    return (unsigned long)(((x + y) ^ y) & ((x + y) ^ x)) >> 63;
}
```

Zadanie 5

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`».

%di to młodsze 16 bitów z %edi

```
//ABCD
eswap:  //ABDC
        rorw $8, %di

        //DCAB
        rorl $16, %edi

        //DCBA
        rorw $8, %di

        movl %edi, $eax
        ret
```

```
x = (x >> 16) | (x << 16)
x = ((x >> 8) & 0x00FF00FF) | ((x & 0x00FF00FF) << 8)
```

Zadanie 6

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`».

x = RDI:RSI y = RDX:RCX

```
superadd:   addq %rsi %rcx
            adc %rdi %rdx
            moveq %rcx %rax
            ret
```

Czyli dodajemy najpierw rejestry młodszych bitów, a potem starszych z uzwględnieniem przeniesienia.

Zadanie 7

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 $n = n_{127\ldots64} \cdot 2^{64} + n_{63\ldots0}$, 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`.

Niech $x_h = x_{[127:64]}, x_l = x_{[63:0]}, y_h = y_{[127:64]}, y_l = x_{[63:0]}$

Wiemy, że

- RDI <- x_h
- RSI <- x_l
- RDX <- y_h
- RCX <- y_l

Zobaczmy teraz co się stanie, gdy wymnożymy x z y $x*y = (x_h * 2^{64} + x_l) * (y_h * 2^{64} + y_l)$ $x*y = x_h * y_h * 2^{128} + (x_h * y_l + y_h * x_l)*2^{64} + x_l * y_l$

Niech $A = x_h * y_h * 2^{128}, B = (x_h * y_l + y_h * x_l)*2^{64}, C = x_l * y_l$

Zauważmy, że A nie mieści się na wynikowych 128 bitach (lub jest zerem), więc można je pominąć.

Zauważmy też, że B również może wyjść poza zadane 128 bitów. Jako, że dolne 64 bity B_l są zerami to wystarczy dodać tylko górne bity B_h do górnych bitów C_h i przepisać C_l.

Czyli wynik będzie postaci: RDX <- B_h + C_h RAX <- C_l

Widząc, że tylko raz wykonamy obliczenia na RDX możemy zaczać od tych obliczeń i przechować je w RDX

(MUL r/m64: Unsigned multiply (RDX:RAX ← RAX ∗ r/m64))

- RDI <- x_h
- RSI <- x_l
- RDX <- y_h
- RCX <- y_l

```
mult128b: //RDX * RSI
          movq %rdx,%rax
          mulq %rsi
          movq %rax,%r8

          //RDI * RCX
          movq %rdi,%rax
          mulq %rcx

          //(RDI*RCX + RDX*RSI) << 64
          addq %rax,%r8

          //RSI * RCX
          movq %rsi,%rax
          mulq %rcx

          //(RDI*RCX + RDX*RSI) << 64 + RSI * RCX
          addq %r8, %rdx
          ret
```

Zadanie 8

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 przepisz je używając instrukcji «sbb».

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

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

Rozwiązanie wykorzystujące skok warunkowy

```
        addq  %rdi, %rsi
        jae   JUMP
        movq  $ULONG_MAX, %rsi
JUMP :  movq  %rsi, %rax
        ret
```

Dodamy $x+y$, flaga CF ustawia się na 1 lub 0 w zależności czy nastąpił nadmiar. Następnie sprawdzamy czy CF = 0 czyli czy nie nastąpił nadmiar. Jeżeli nie nastąpił to skaczemy do jump1 i zwracamy x+y, jezeli nastąpił to wsadzamy do %rsi $ULONG_MAX i zwracamy jako wynik.

Rozwiązanie bez skoku warunkowego

```
addq   %rdi, %rsi
sbbq   %rax, %rax
orq    %rsi, %rax
```

Dodajemy $x+y$ i ustawia się nasza flaga CF. Cała magia dzieje się teraz w sbbq %rax %rax bo jakby robimy tam 0 - (0 + CF), co da nam ULONG_MAX gdy się przepełni bo nastąpi przepełnienie i 0 gdy nie nastąpi te przepełnienie. Operacja orq %rsi, %rax za to gdy sbbq wpisało do %rax 0 zwróci nam $x+y$. Gdy wpisało ULONG_MAX to oczywiście zwraca też ULONG_MAX bo jego zapis binarny to same jedynki.

Zadanie 9

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)`».

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

x i y otrzymujemy odpowiednio w rejestrach RDI i RSI

```
cmp: subq %rsi, %rdi
     sbbq %rax, %rax
     negq %rdi
     adcq %rax, %rax
     ret
```

Rozważmy przypadki:

- x < y `subq %rsi, %rdi` ustawia CF na 1 `sbbq %rax, %rax` ustawia RAX na -CF (czyli -1) `negq %rdi` ustawia CF na 1 `adcq %rax, %rax` RAX = -1 + (-1 + CF) = -1
- x = y `subq %rsi, %rdi` ustawia CF na 0 `sbbq %rax, %rax` ustawia RAX na 0 `negq %rdi` ustawia CF na 0 `adcq %rax, %rax` RAX = 0 + (0 + 0) = 0
- x > y `subq %rsi, %rdi` ustawia CF na 0 `sbbq %rax, %rax` ustawia RAX na 0 `negq %rdi` ustawia CF na 1 `adcq %rax, %rax` RAX = 0 + (0 + 1) = 1