LISTA 4

ZAD. 1

w zależności od tego jak porównuje cmp ustawia flagi

setb - set byte if below (unsigned) CF
setl - set byte if less (signed) SF^OF
↳ daje 1 kiedy b-a<0

FLAGI:

setb - 1 jeśli CF=1

- CF=1 if carry out from MSB (used for unsigned comparison)
- SF=1 if (b-a)<0 (signed)
- OF=1 if two's complement (signed) overflow
(b>0 && a<0 && (b-a)<0 || (b<0 && a>0 && (b-a)>0)
↳ kiedy wynik + lub - odejmowanie nie mieści się w zakresie

INSTRUKCJA CMP a b

zwraca 1 gdy a > b
0 > b-a

(zapali 1)
CF (carry flag - bez znaku) = 1 wtw, kiedy wystąpi borrow bo carry
ZF (zero flag) = 1 x-y=0 → (x=y)
SF (sign-flag - ze znakiem) x-y<0
OF (overflow - ze znakiem) gdy wystąpi overflow/underflow
b-a<0

1. dla signed (setl):

- a ≥ 0 && b ≥ 0
Nie wystąpi overflow, czyli OF=0
  - a > b
  b-a<0, więc SF=1, a SF^OF=1
  setl ustawia na 1, a
b jest mniejsze niż a, więc działa zgodnie z oczekiwaniem
  - a ≤ b
  b-a>0, więc SF=0, a SF^OF=0
  setl ustawia na 0, a
b nie jest mniejsze niż a, więc działa zgodnie z oczekiwaniem
- a < 0 && b < 0
Nie może wystąpić overflow, czyli OF=0
  - a > b
  b-a<0, więc SF=1, OF=0, SF^OF=1,
  setl ustawia 1, a
b jest mniejsze niż a, więc działa zgodnie z oczekiwaniem
  - a ≤ b
  b-a≥0, więc SF=0, OF=0, SF^OF=0,
  setl ustawia 0, a
b nie jest mniejsze niż a, więc działa zgodnie z oczekiwaniem.
- a ≥ 0 && b < 0
  - Jeśli wystąpi overflow (odejmowanie będzie dodatnie chociaż niepoprawne) to SF=0, OF=1, więc setl zadziała poprawnie SF^OF=1
  - Jeśli overflow nie wystąpi to OF=0, SF=1, więc setl działa poprawnie SF^OF=1

b-a

- a < 0 && b ≥ 0

- Jeśli wystąpi overflow (różnica będzie ujemna, chociaż nie powinna) wtedy SF=0 a OF=1, więc OF^SF=0, więc setl działa poprawnie.

- Jeśli overflow nie wystąpi, wtedy SF=0 oraz OF=0, więc setl nadal działa poprawnie

2. dla unsigned (setb)

- a ≤ b

b-a ≥ 0 wykona się poprawnie niezależnie od wielkości a i b (nie może nastąpić pożyczenie z najbardziej znaczącego bitu, ponieważ byłoby to zaprzeczeniem, że a≤b), stąd CF=0 i setb zadziała zgodnie z przewidywaniem.

- a > b

b-a < 0 w tym przypadku konieczne jest zapożyczenie z bitu, który jest już poza długością liczby, niezależnie ile ta długość wynosi (następuje carry), więc zapali się CF=1, więc setb ustawi 1 i zadziała zgodnie z oczekiwaniem.

Każdego rozkazu set dotyczy! Każda instrukcja set nie zostawia flag, decyduje czy warunek jest spełniony, jest też taka [illegible] żeby [illegible] wszystkie przypadki

ZAD. 2

```
1  who:  subq  $1, %rsi          // n = n-1
2        movl  $0, %ecx          // i = 0, zerujemy 4 ostatnie bajty rcxa
3  .L2:  cmpq  %rsi, %rcx        // aby wróciło CF=1 dopóki n>i
4        jnb   .L4               // wtedy (n>i)
5        leaq  (%rdi,%rcx,2), %rcx   // w rcx znajduje się rdi+2*rcx, czyli wskazuje i-ty element w tablicy
6        movzwl (%rcx), %r8d     // x = rdi[i]
7        leaq  (%rdi,%rsi,2), %rdx
8        movzwl (%rdx), %r9d     // y = rdi[n]
9        movw  %r9w, (%rcx)      // rdi[i] = y
10       movw  %r8w, (%rdx)      // rdi[n] = x
11       addq  $1, %rcx          // i++
12       subq  $1, %rsi          // n--
13       jmp   .L2
14 .L4:  ret
```

zamiana 26→46

*2 bo jesteśmy w short, czyli przesuwamy o 2 bajty

void who(short v[], size_t n)
// rdi -> wskaźnik na adres początku tablicy (na początek tablicy shortów)
// rsi -> n
// ecx - ostatnie 4 bajty rcxa
// jnb bierze ~CF
// r8d - 4 dolne bajty r8
// r9d - 4 dolne bajty r9
// rcx -> wskaźnik na i-ty element tablicy

Czyli funkcja odwraca kolejność w tablicy.

```c
void who (short v[], size_t n){
    int j = n-1;
    for (int i=0; i<=j; i++){
        short temp = v[i];
        v[i] = v[j];
        v[j] = temp;
        j--;
    }
}
```

Znajomość sygnatury nie jest istotna, gdyż z naszej decyzji mamy do czynienia z tablicą, a instrukcja movw wskazuje, że operujemy na shortach.

ZAD 3.

kopiuje bit znaku

ZF=1 wtw x-y=0 (x==y)

```
zonk:  movl    $0, %ecx           // rcx=0
.L2:   movslq  %ecx, %rax         // long long rax = rcx;
       movzbl  (%rdi,%rax), %edx  // rdx = a[rax];
       testb   %dl, %dl    --> string kończy się zerem'
       je      .L6  <------------ // if (dl==0)
       cmpb    %dl, (%rsi,%rax)
       jne     .L5                // if(a[rax] != b[rax])
       addl    $1, %ecx           // rcx ++
       jmp     .L2
.L6:   orb     (%rsi,%rax), %dl   // ZF ustawione jeśli
                                  // oba stringi się skończyły
       sete    %al  --> daje AL w zależności jak orb ustawiło flagi
       ret
.L5:   movl    $0, %eax           // return false
       ret
```

a: w ostatnim bajcie

orb zwróci jeżeli y też się skończył

zwróci prawdę jeśli jeśli ZF = 1, a jeśli ZF = 1 jeśli orb zwróci false

// rdi -> a
// rsi -> b
// char jednobajtowy
// dl to ostatni bajt rdx
// al to ostatni bajt rax
// je bierze ZF
// jne bierze ~ZF
// sete bierze ZF
// test x,y ; set ZF to 1 if x&y=0
// test x,x ; ustawi ZF na 1 gdy x=0

Funkcja sprawdza, czy string x jest prefiksem y (przedrostkiem)

```c
bool zonk(char* a, char* b){
    for (int i=0; ; i++){
        char x = *(a+i);   // char x = a[i];
        char y = *(b+i);   // char y = b[i];
        if (x == '\0'){
            return x==y; true
        }
        if(x != y)
            return false
    }
}
```

ZAD. 4

%rsp wskaźnik wierzchołka stosu

```
pushq src:
    subq $8, %rsp  // odjęcie 8 bajtów
    movq src, (%rsp)

popq dest:
    movq (%rsp), dest
    addq $8, %rsp
```

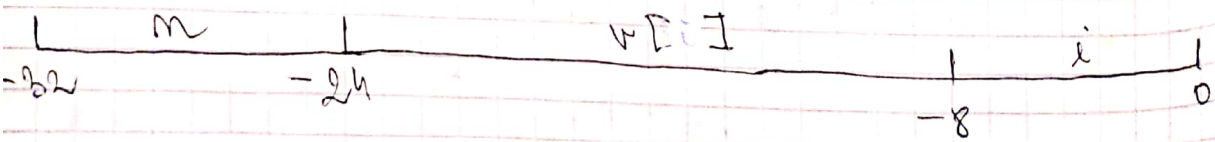
ZAD. 5

wskaźnika na ramkę stosu ogólnie zaczyna się później dla tej funkcji

```
1  foo:  pushq %rbp          //wrzucamy rbp na stos
         movq %rsp, %rbp     //początek stosu w rbp
         movq %rdi, -24(%rbp)
         movq %rsi, -32(%rbp)   przygotowanie stosu
         movq $0, -8(%rbp)      0 ≤ i  8-23: v[]  24-31: n;
         jmp .L2
.L3:     movq -8(%rbp), %rax    // w rax wskaźnik na i
         leaq (%rax,%rax), %rdx //rax = i+i = 2i
         movq -24(%rbp), %rax   // w rax wskaźnik na początek tablicy
         addq %rdx, %rax        // w rax wskaźnik na i-ty element tablicy
         movzwl (%rax), %ecx    // w rax i-ty element tablicy
         leal (%rax,%rax), %edx // w rdx 2·ity element tablicy
         movq -8(%rbp), %rax    // w rax wskaźnik na i
         leaq (%rax,%rax), %rcx // rcx = 2i
         movq -24(%rbp), %rax   // w rax wskaźnik na początek tablicy
         addq %rcx, %rax        // w rax wskaźnik na ity element tablicy
         movw %dx, (%rax)       //do adresu mam. wpisz 2·v[i]
         addq $1, -8(%rbp)      // i++
.L2:     movq -8(%rbp), %rax    // rax = i
         cmpq -32(%rbp), %rax   //(i < n)
         jb .L3                 // while (i<n)
         nop
         popq %rbp              //zwolnij stos
         ret
```

rsp -> wskaźnik wierzchołka stosu
// rdi -> wskaźnik na adres początku tablicy
// rsi -> n
// jb bierze CF
// rbp to frame pointer
// eax to dolne bajty rdx
// ecx to dolne bajty rax

```
void foo (int16_t v[], size_t n){
    for(int i=0; i<n; i++){
        v[i] = 2* v[i];
    }
}
```



ZAD. 6

```
reccur:
        pushq   %rbp              // 16 bajtów wyjęcia
        movq    %rsp, %rbp
        subq    $16, %rsp
        movl    %edi, -4(%rbp)    -> jeśli x==0 to ZF=1
        cmpl    $0, -4(%rbp)      // (0 == x)?   if (x!=0) jmp L2
        jne     .L2
        movl    $1, %eax
        jmp     .L3
.L2:    movl    -4(%rbp), %eax    // eax = x
        subl    $1, %eax          // eax = x-1
        movl    %eax, %edi        // rdi = x-1
        call    reccure           // wynik w rax
        imull   -4(%rbp), %eax    // eax * reccure(eax-1)
                      (rax)       rax = x*(x-1)
.L3:    leave     // -> przesuwa wskaźnik wierzchołka stosu (%rsp) na miejsce w pamięci dla tej funkcji, czyli tam gdzie zaczyna się ramka stosu dla funkcji (%rbp)
        ret
```

//edi -> ostatnie 4 bajty rdi
// rdi -> x
// jne bierze ~ZF, jump if not equal zero

```
int reccur (int x){
        if (x == 0){
            return 1
        }
        return x * reccur(x-1);
}
```

ZAJĘCIA

ZAD. 4

```
# pushq %reg1
  leaq  -8(%rsp), %rsp
  movq  %reg1, (%rsp)
```

przez leaq, bo addq i subq ustawiają flagi

```
# popq %reg2
  movq  (%rsp), %reg2
  leaq  8(%rsp), %rsp
```

ZAD. 7

Ramka funkcji bar wygląda tak:

itd.
ósmy argument
siódmy argument (%rsp + 16)
adres powrotu (%rsp + 8)
%rbp <- na ten adres wskazuje %rsp

widzimy gdzie wpisuje się kolejne argumenty danej funkcji

	MIEJSCE ZAPISU	W RBP POWOŁANIU BAR
arg 1	rdi	rbp-4
arg 2	rsi	rbp-8
arg 3	rdx	rbp-12
arg 4	rcx	rbp-16
arg 5	r8d	rbp-20
arg 6	r9d	rbp-24
arg 7	rbp+16	rbp+16

Od 7 argumentu kolejne są zapisywane na stosie rbp. Począwszy od 7 zapisanego na rbp+16. Widzimy że podany w zadaniu kod assembler odnosi się właściwie

do tego miejsca w pamięci. Co oznacza, że funkcji bar musiało zostać podane conajmniej 7 argumentów.

$n \geq 7$

Jak zmieni się kod gdyby n było większe?

Każdy kolejny argument byłby przekazany na stos.

```
bar: pushq %rbp
     movq %rsp, %rbp
     ...
     movl 16(%rbp), %eax
     popq %rbp
     ret
```