



Assembly

The assemblage of features

Mola mola (sunfish)



Mola mola (sunfish)



600 m

Mola mola (sunfish)

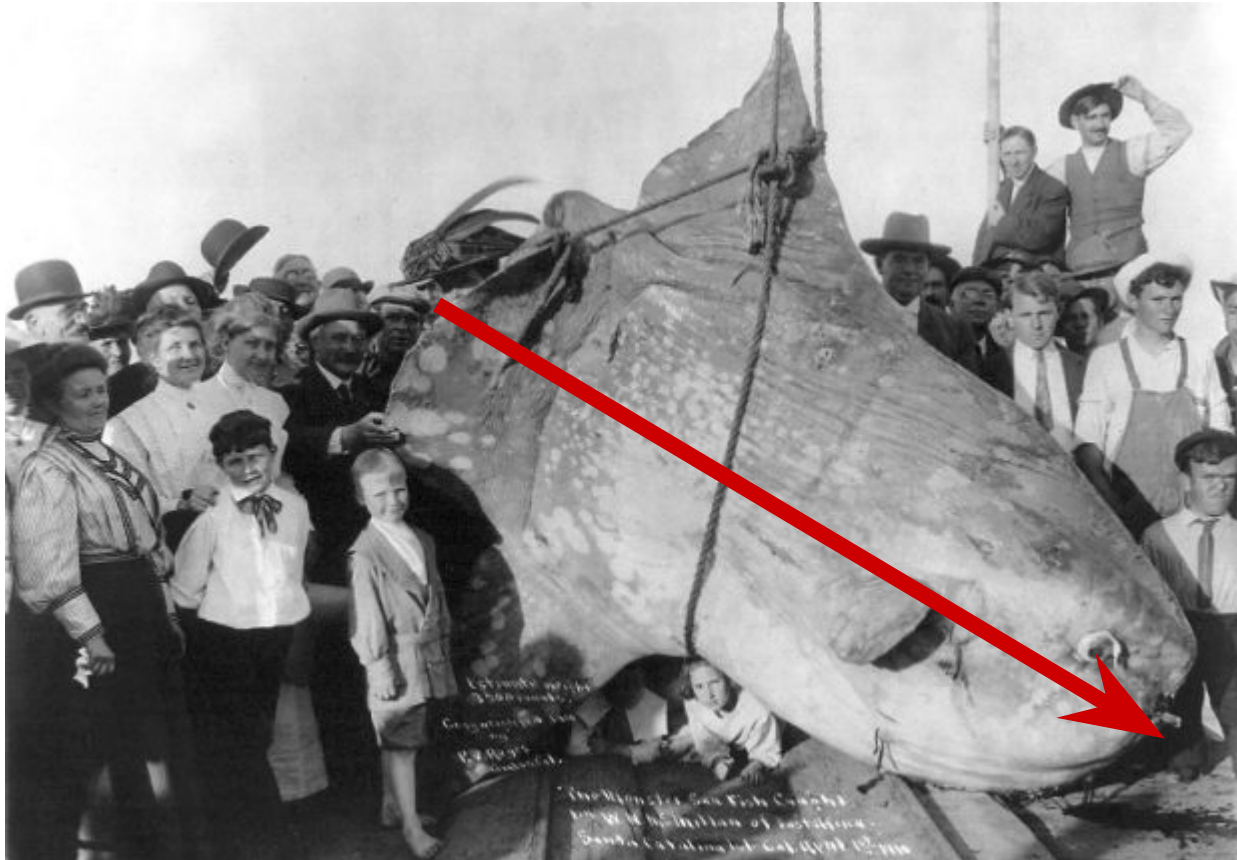


1000 kg

Mola mola (sunfish)



Mola mola (sunfish)



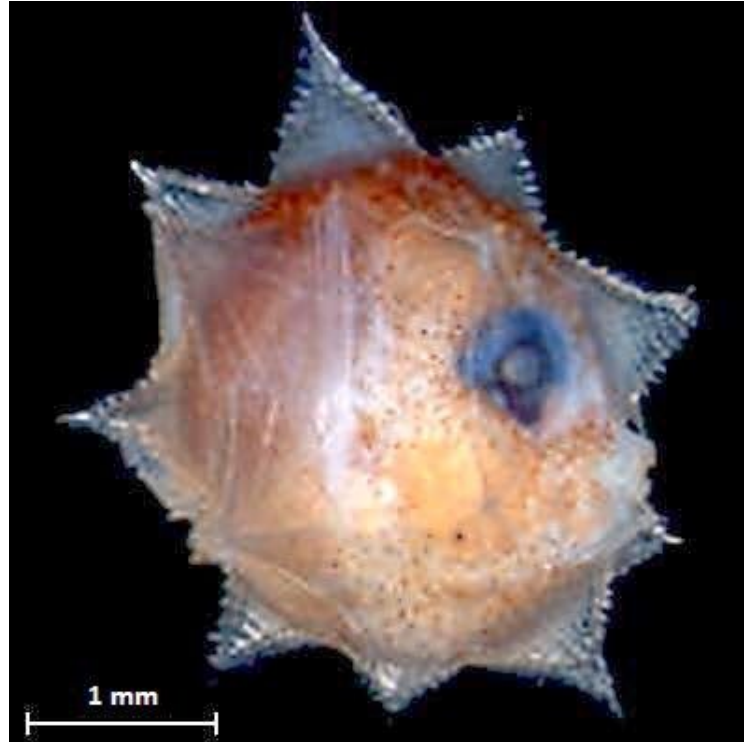
3.3 m

Mola mola (sunfish)



4.2 m

Mola mola (sunfish)





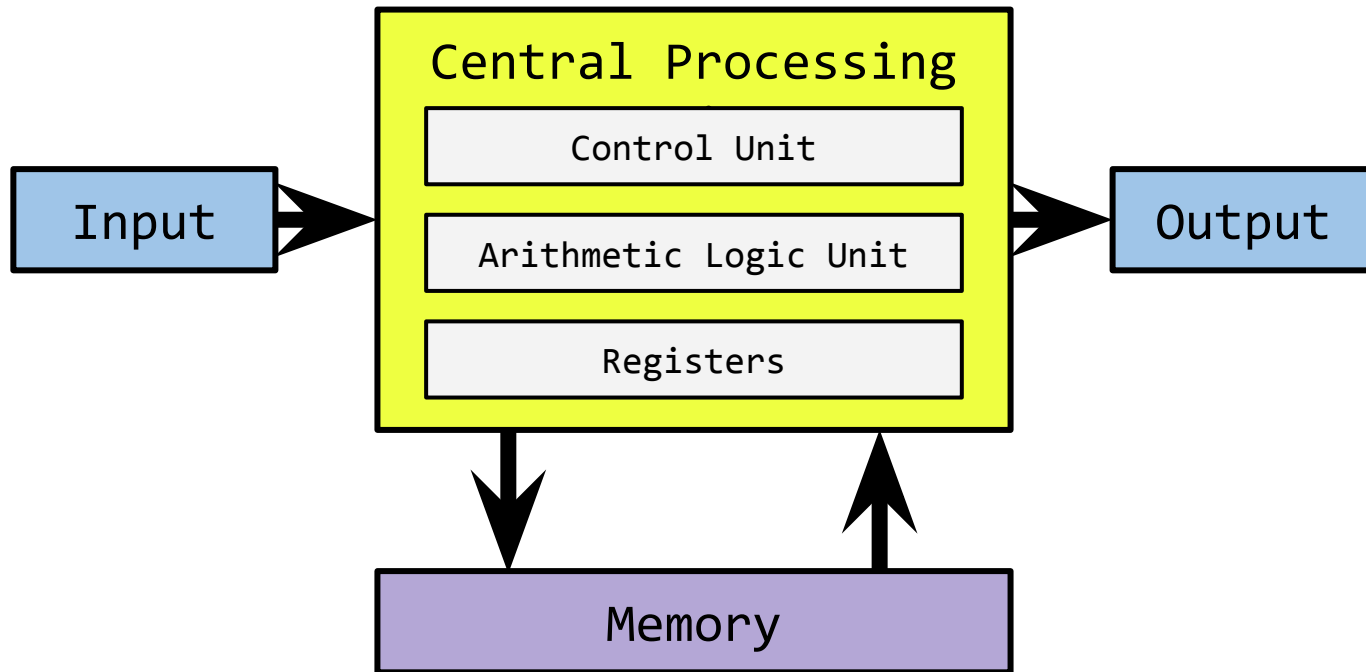
★ **A recap from the last week**

★ Scenario: Q&A

★ First and next steps in gdb

We have learnt about:

a simple computer architecture



We have learnt about:

memory problem



We have learnt about:

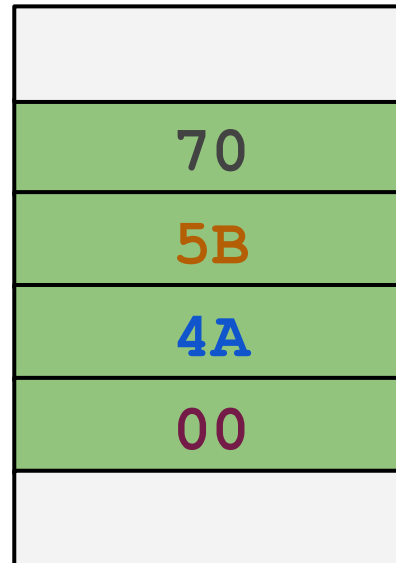
registers



We have learnt about:

main memory

(00 4A 5B 70)₁₆



...

0x0100100**9**

0x0100100**A**

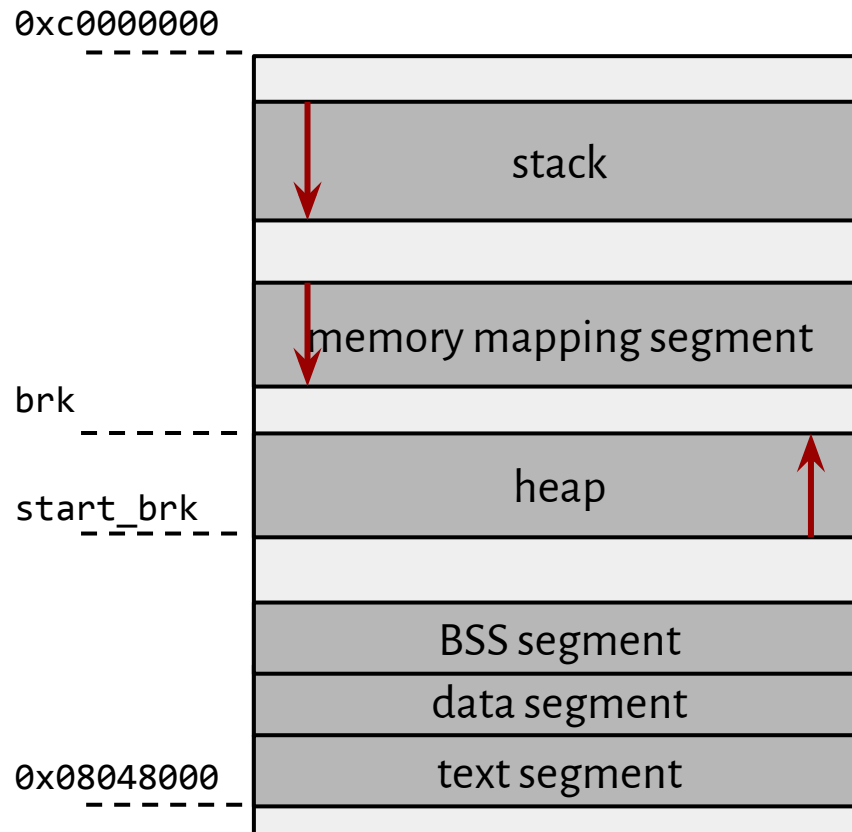
0x0100100**B**

0x0100100**C**

...

We have learnt about:

program memory layout





- ★ A recap from the last week
- ★ **Scenario: Q&A**
- ★ First and next steps in gdb

Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```


Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```

```
int64_t sddiv(int64_t x) {  
    return x << 1;  
}
```

```
uint64_t uddiv(uint64_t x) {  
    return x << 1;  
}
```

Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```

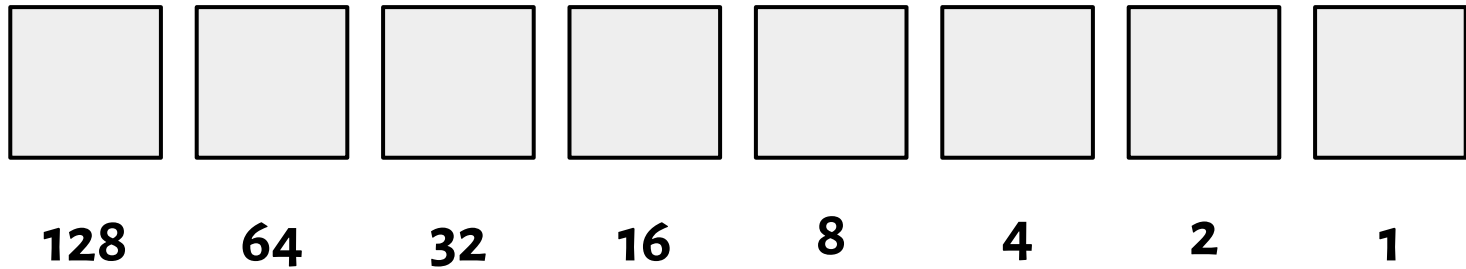
```
int64_t sddiv(int64_t x) {  
    return x << 1;  
}
```

```
mov rax,rdi  
sar rax,1
```

```
uint64_t uddiv(uint64_t x) {  
    return x << 1;  
}
```

```
mov rax,rdi  
shr rax,1
```

Unsigned integers in x86

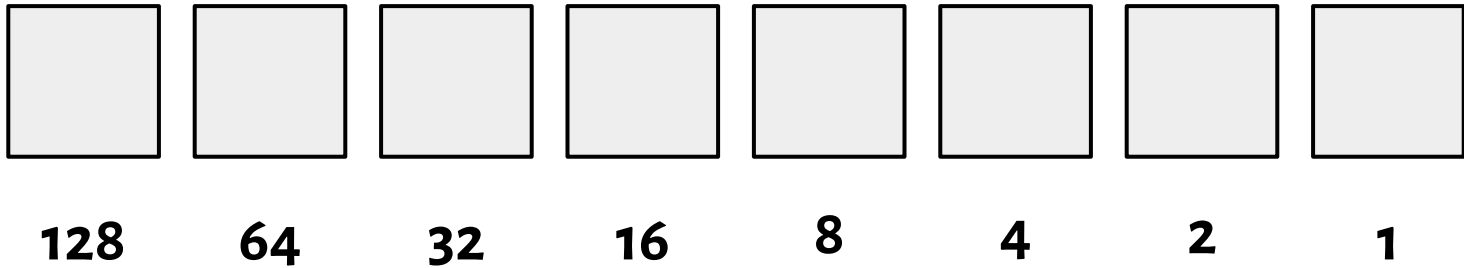


$$x = \sum_{i \in [0, n)} 2^i \cdot t[i]$$

$n = 8$

Unsigned integers in x86

21

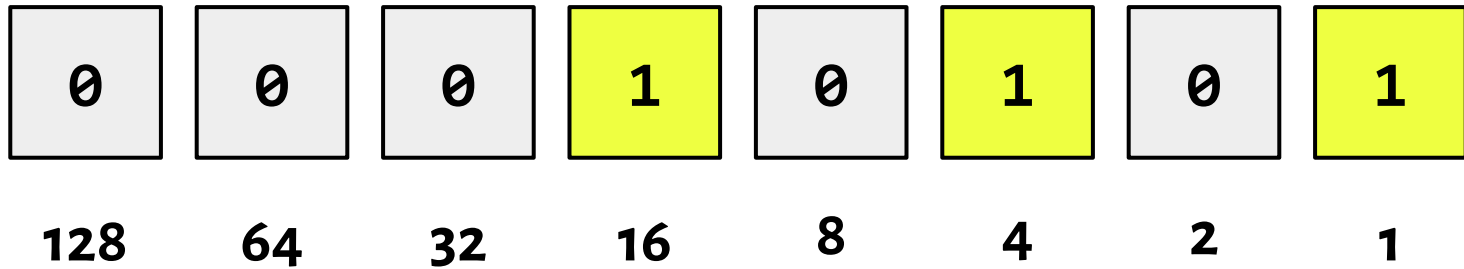


$$x = \sum_{i \in [0, n)} 2^i \cdot t[i]$$

$n = 8$

Unsigned integers in x86

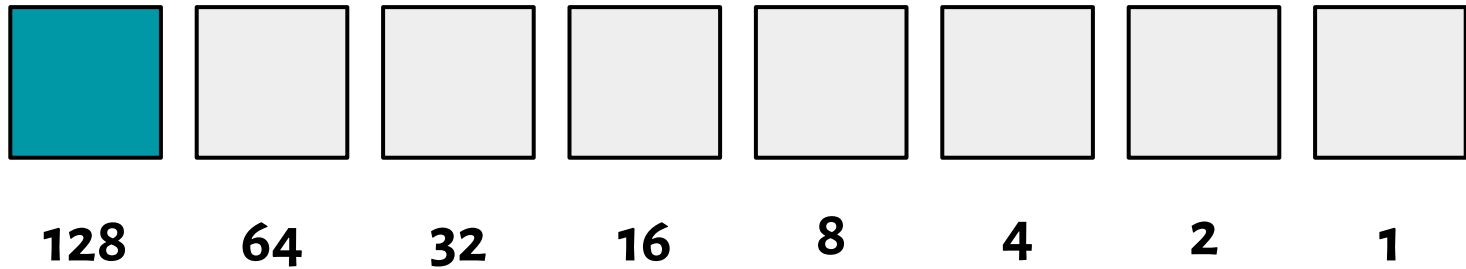
21



$$x = \sum_{i \in [0, n)} 2^i \cdot t[i]$$

$n = 8$

Signed integers in x86

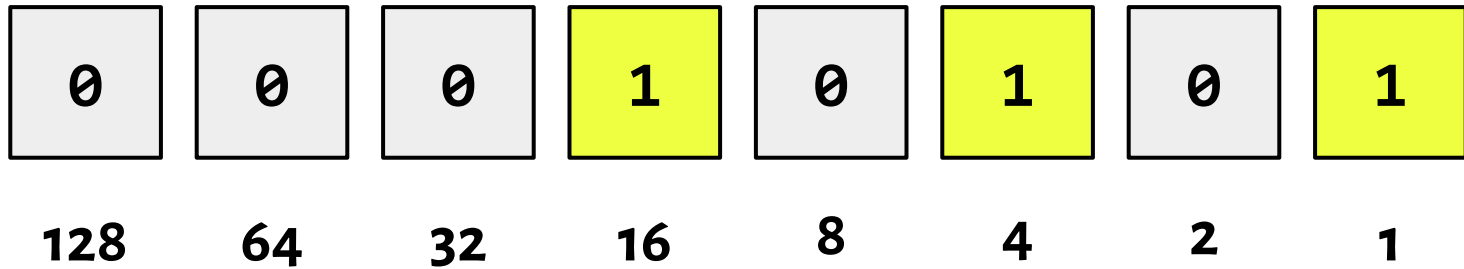


$$X = (-1)^{t[n-1]} \cdot 2^{n-1} + \sum_{i \in [0, n-1)} 2^i \cdot t[i]$$

$n = 8$

Signed integers in x86

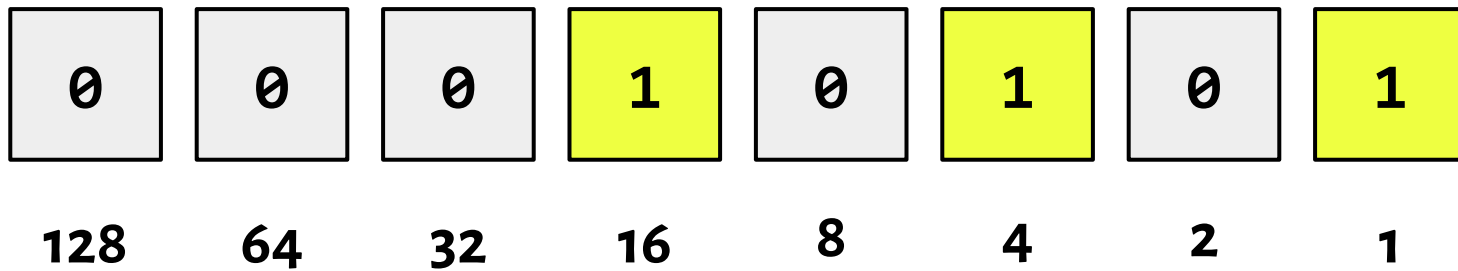
21



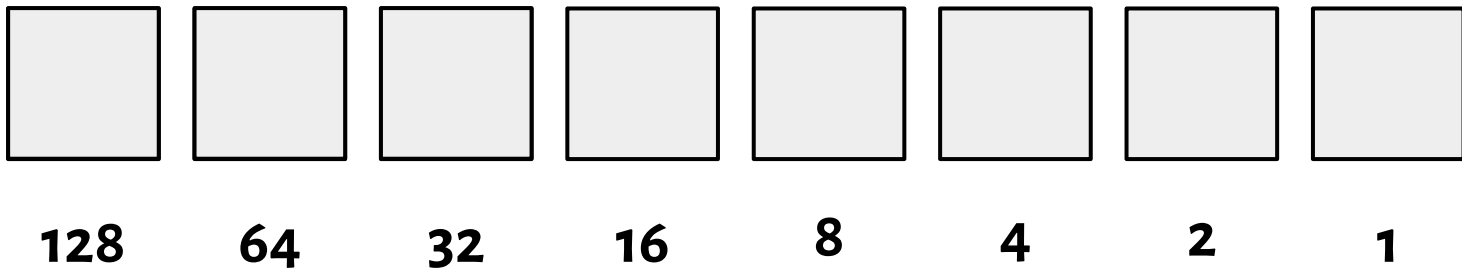
$n = 8$

Signed integers in x86

21



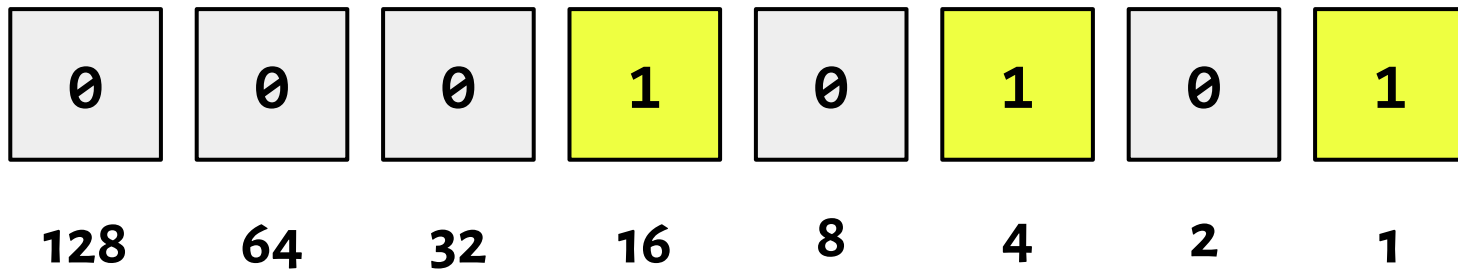
-21



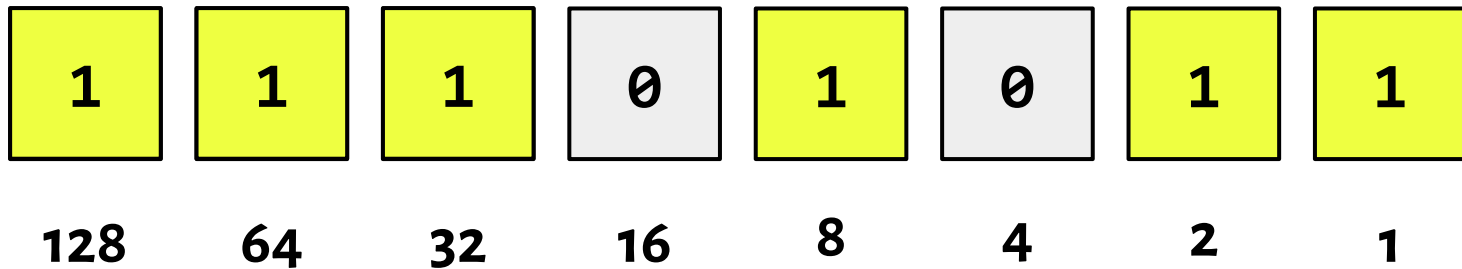
$n = 8$

Signed integers in x86

21



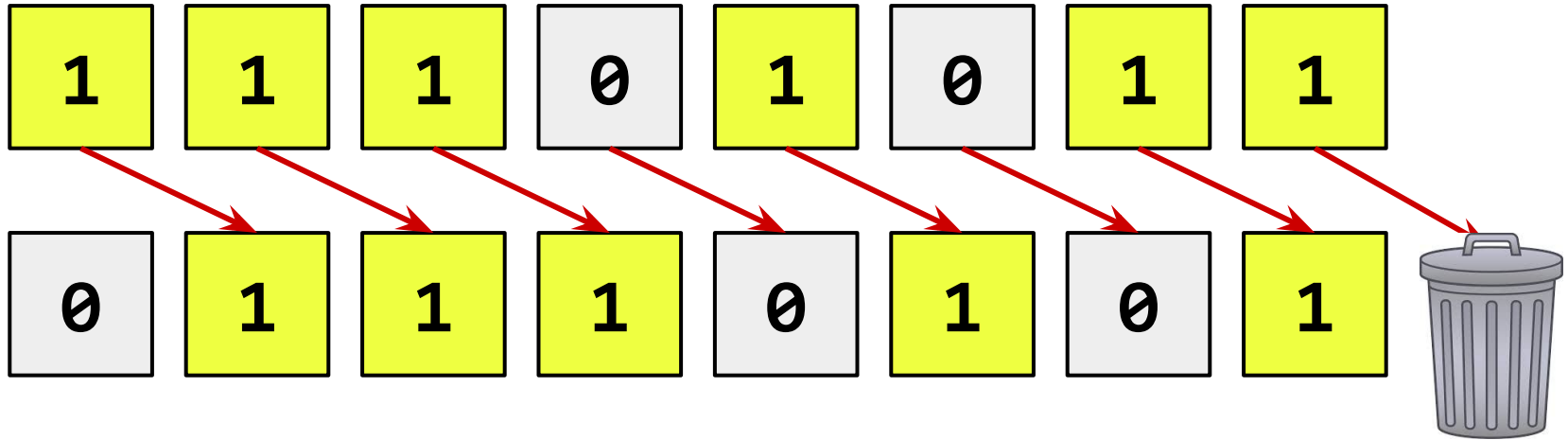
-21



$n = 8$

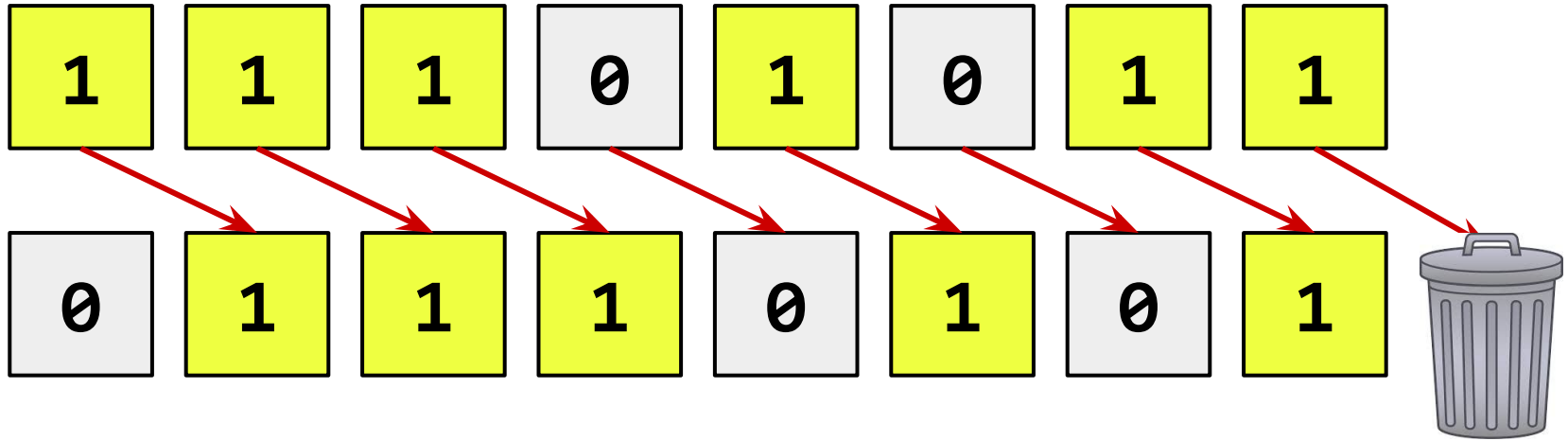
SHR vs. SAR

sHR

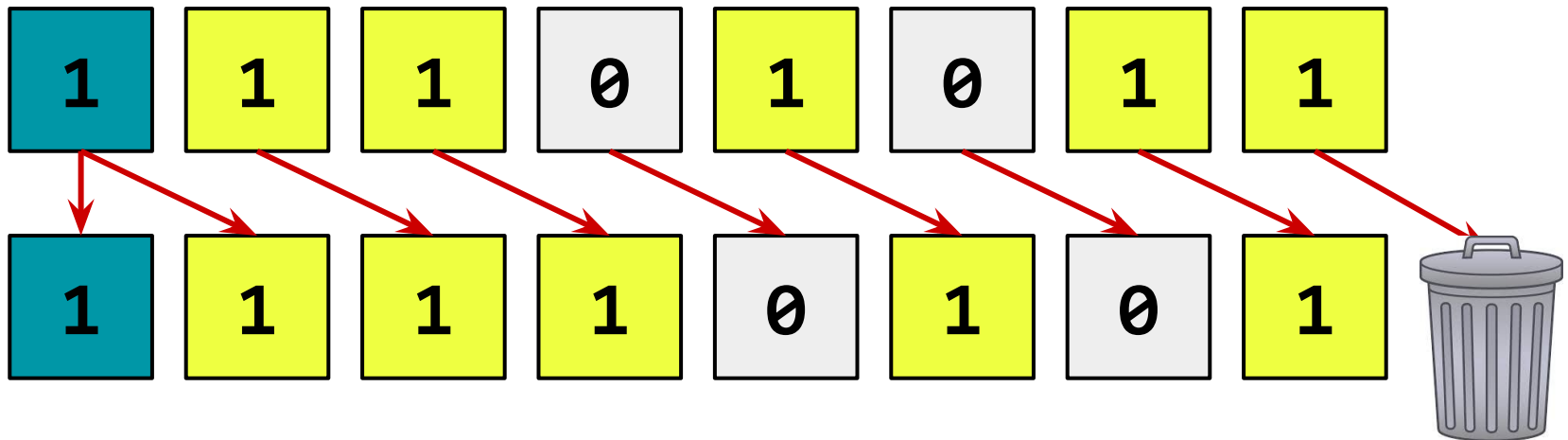


SHR vs. SAR

sHR



sAR



Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```

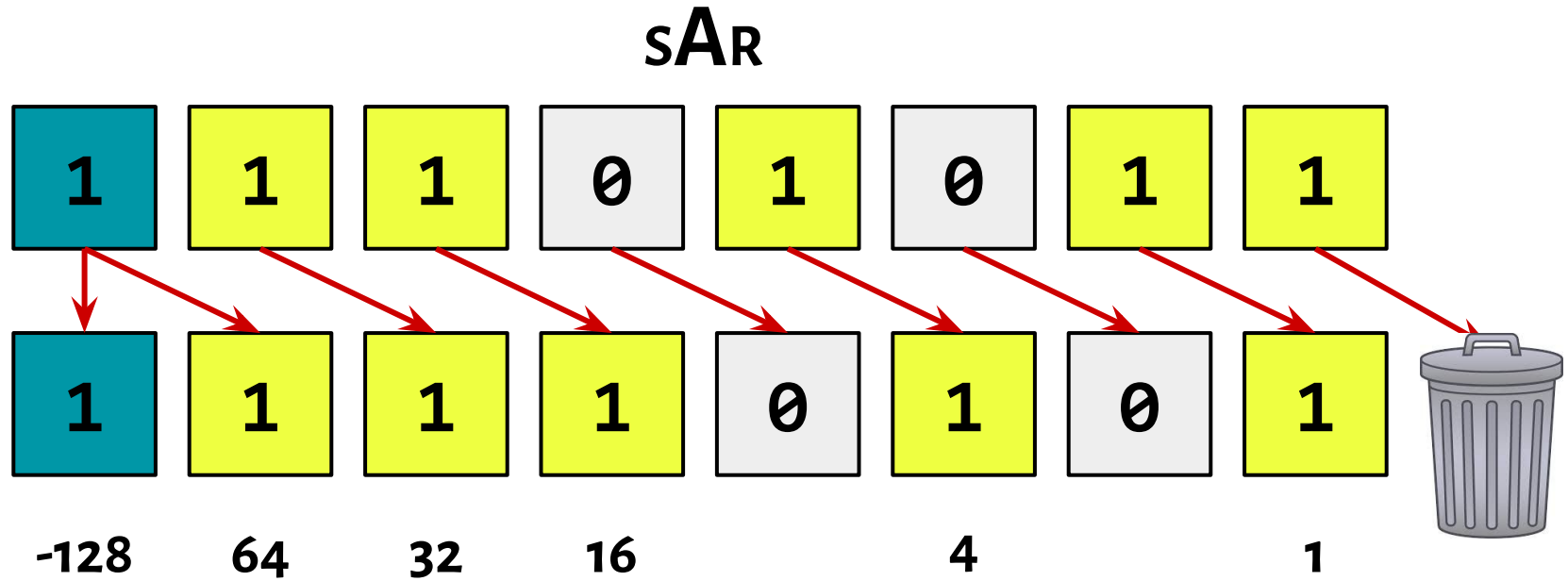
```
int64_t sddiv(int64_t x) {  
    return x / 2;  
}
```

```
mov rax,rdi  
shr rax,0x3f  
add rax,rdi  
sar rax,1
```

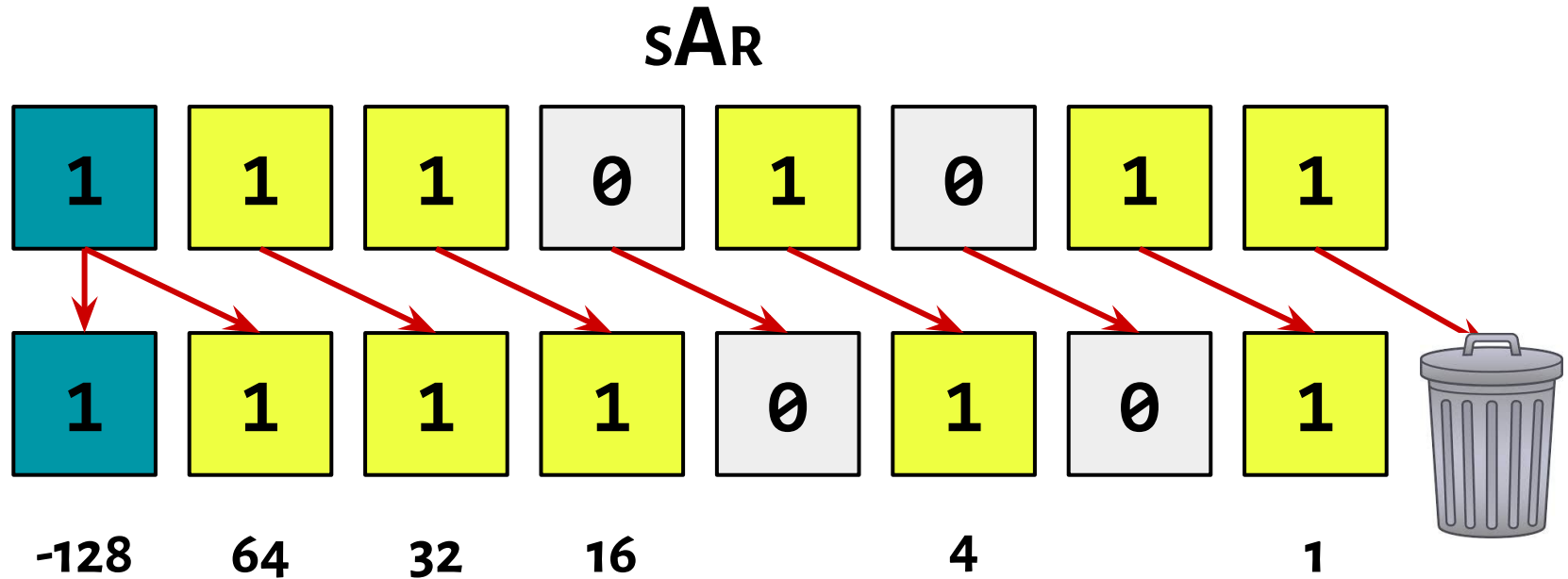
```
uint64_t uddiv(uint64_t x) {  
    return x / 2;  
}
```

```
mov rax,rdi  
shr rax,1
```

SHR vs. SAR



SHR vs. SAR



$$X = -11$$

Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```

```
int64_t smmul(int64_t x) {  
    return x * 2;  
}
```

```
lea rax,[rdi+rdi*1]  
ret
```

```
uint64_t ummul(uint64_t x) {  
    return x * 2;  
}
```

```
lea rax,[rdi+rdi*1]  
ret
```

Scenario: A1

```
$ gcc -c -Wall -O2 a1.c -o a1.o
```

```
$ objdump -d -M intel_mnemonic a1.o
```

```
int64_t smmul(int64_t x) {  
    return x << 1;  
}
```

```
lea rax,[rdi+rdi*1]  
ret
```

```
uint64_t ummul(uint64_t x) {  
    return x << 1;  
}
```

```
lea rax,[rdi+rdi*1]  
ret
```


LEA

Load Effective Address

LEA R3, [R1+a•R2+b]

a = 1, 2, 4 or 8

Sometimes faster and makes the code size smaller.

Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_a(s_t arg) {  
    return arg.a;  
}
```

Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_a(s_t arg) {  
    return arg.a;  
}
```

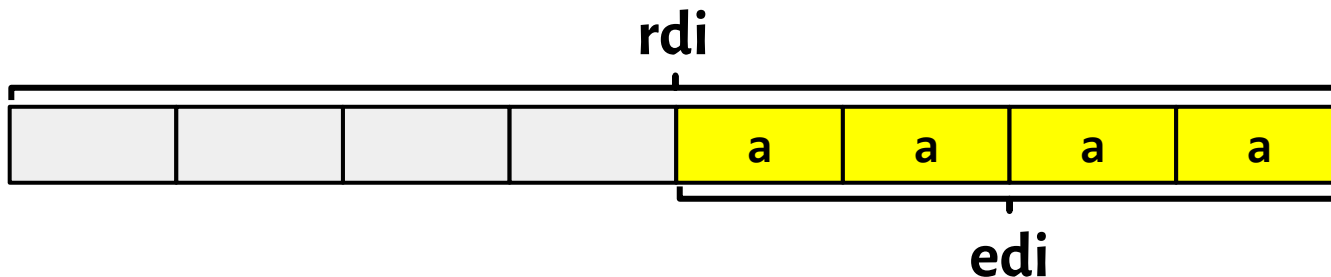
Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_a(s_t arg) {  
    return arg.a;  
}
```

```
mov eax,edi  
ret
```



Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_b(s_t arg) {  
    return arg.b;  
}
```

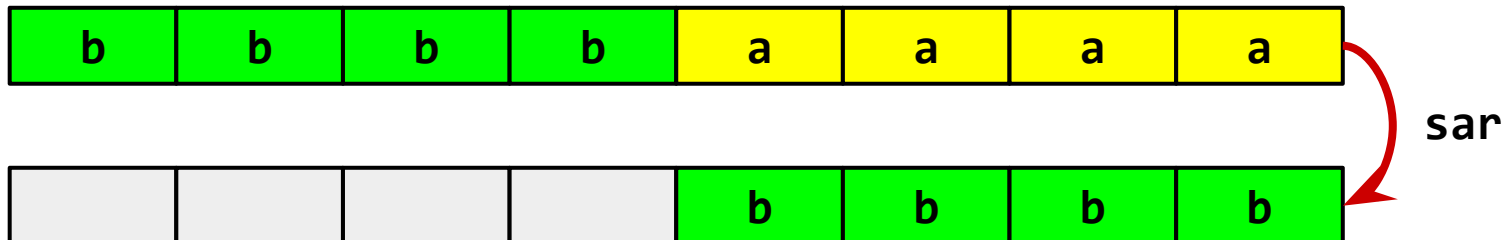
Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_b(s_t arg) {  
    return arg.b;  
}
```

```
mov rax,rdi  
sar rax,0x20  
ret
```



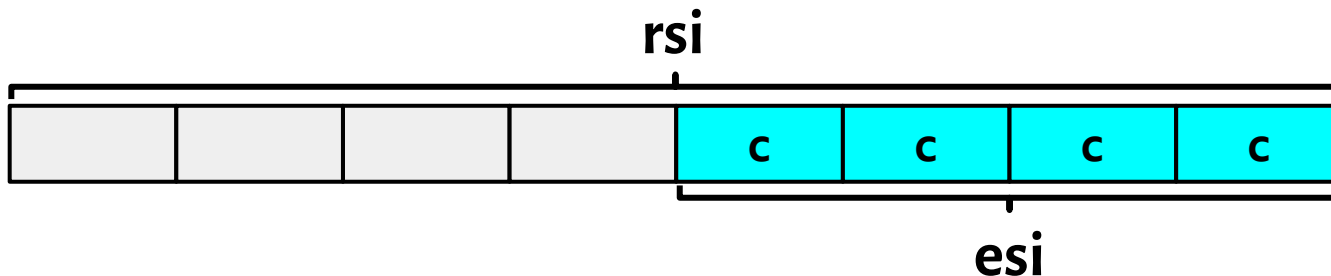
Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
int foo_c(s_t arg) {  
    return arg.c;  
}
```

```
mov eax, esi  
ret
```

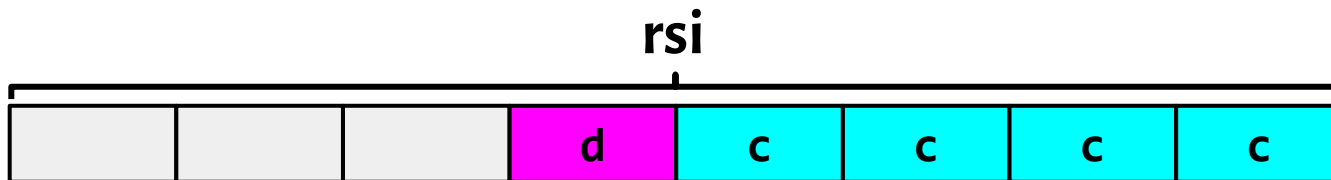


Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
char foo_d(s_t arg) {  
    return arg.d;  
}
```



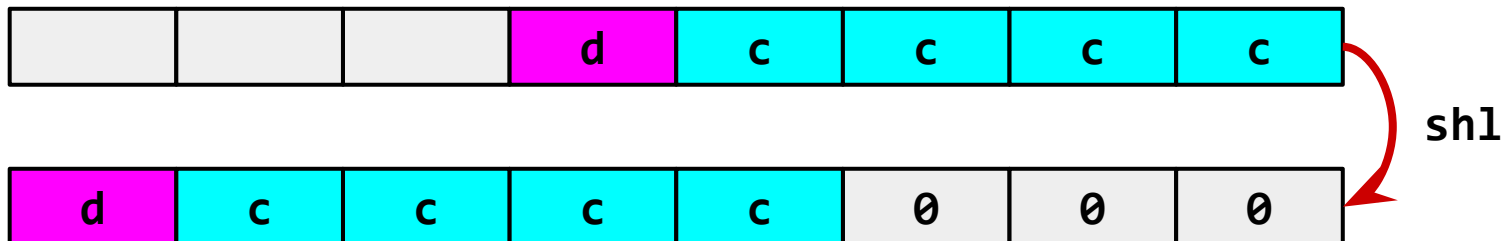
Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
char foo_d(s_t arg) {  
    return arg.d;  
}
```

```
mov rax,rsi  
shl rax,0x18
```



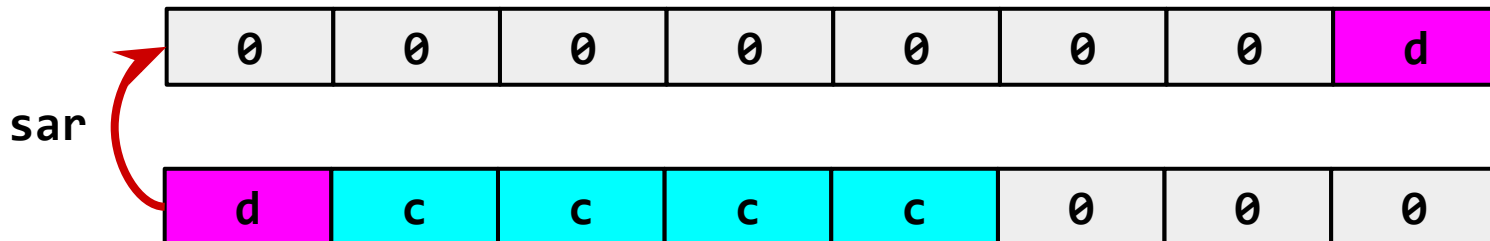
Scenario: A2

Poeksperymentuj z innymi typami argumentów: wskaźniki, struktury, wskaźniki do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

```
char foo_d(s_t arg) {  
    return arg.d;  
}
```

```
mov rax,rsi  
shl rax,0x18  
sar rax,0x38  
ret
```



Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;
```

Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {
    int a;
    int b;
    int c;
    char d;
    char e;
    char f;
} s_t;

s_t bar1() {
    s_t s;
    s.a = 1;
    s.b = 2;
    s.c = 3;
    s.d = 'a';
    s.e = 'b';
    s.f = 'c';
    return s;
}
```

Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;  
  
s_t bar1() {  
    s_t s;  
    s.a = 1;  
    s.b = 2;  
    s.c = 3;  
    s.d = 'a';  
    s.e = 'b';  
    s.f = 'c';  
    return s;  
}
```

```
movabs rax,0x200000001  
movabs rdx,0x63626100000003  
ret
```

Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
} s_t;  
  
s_t bar1() {  
    s_t s;  
    s.a = 1;  
    s.b = 2;  
    s.c = 3;  
    s.d = 'a';  
    s.e = 'b';  
    s.f = 'c';  
    return s;  
}
```

```
movabs rax,0x200000001  
movabs rdx,0x63626100000003  
ret
```

rax	b	b	b	b	a	a	a	a
rdx		f	e	d	c	c	c	c

Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {
    int a;
    int b;
    int c;
    char d;
    char e;
    char f;
    int g;
} s_t;

s_t bar1() {
    s_t s;
    s.a = 1;
    s.b = 2;
    s.c = 3;
    s.d = 'a';
    s.e = 'b';
    s.f = 'c';
    s.g = 42;
    return s;
}
```

Scenario: A3

Poeksperymentuj ze zwracaniem wartości innych typów, wskaźników, struktur, wskaźników do struktur.

```
typedef struct {  
    int a;  
    int b;  
    int c;  
    char d;  
    char e;  
    char f;  
    int g;  
} s_t;  
  
s_t bar1() {  
    s_t s;  
    s.a = 1;  
    s.b = 2;  
    s.c = 3;  
    s.d = 'a';  
    s.e = 'b';  
    s.f = 'c';  
    s.g = 42;  
    return s;  
}
```

```
mov rax,rdi  
mov DWORD PTR [rdi],0x1  
mov DWORD PTR [rdi+0x4],0x2  
mov DWORD PTR [rdi+0x8],0x3  
mov BYTE PTR [rdi+0xc],0x61  
mov BYTE PTR [rdi+0xd],0x62  
mov BYTE PTR [rdi+0xe],0x63  
mov DWORD PTR [rdi+0x10],0x2a  
ret
```


Scenario: A4

Zaimplementuj w asemblerze funkcję o sygnaturze

```
uint128_t mac (uint128_t const *a,  
               uint128_t const *x,  
               uint128_t const *y);
```

która oblicza wartość $a + x \cdot y$ modulo 2 do potęgi 128.

Scenario: A4

Zaimplementuj w assemblerze funkcję o sygnaturze

```
uint128_t mac (uint128_t const *a,  
               uint128_t const *x,  
               uint128_t const *y);
```

która oblicza wartość $a + x \cdot y$ modulo 2 do potęgi 128.

```
typedef struct {  
    uint64_t lo;  
    uint64_t hi;  
} uint128_t
```

Scenario: A4

```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

a_lo

a_hi

x_lo

x_hi

y_lo

y_hi

Scenario: A4

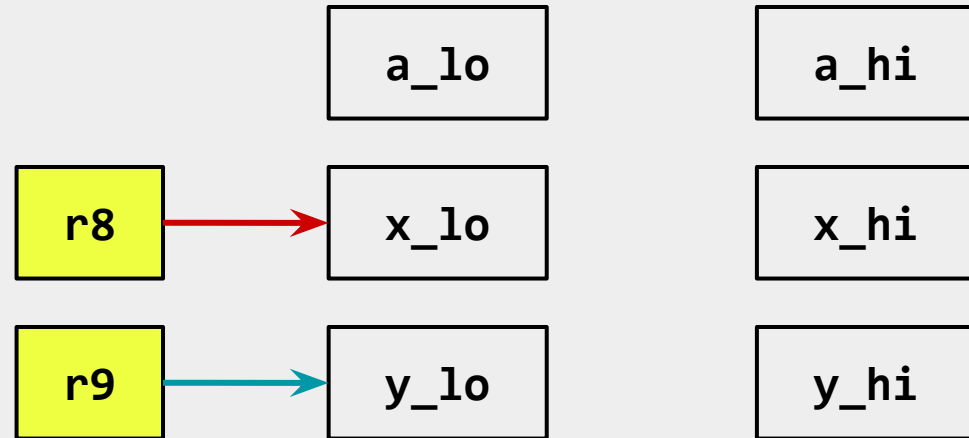
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
    mov    r8, [rsi]  
    mov    r9, [rdx]
```



Scenario: A4

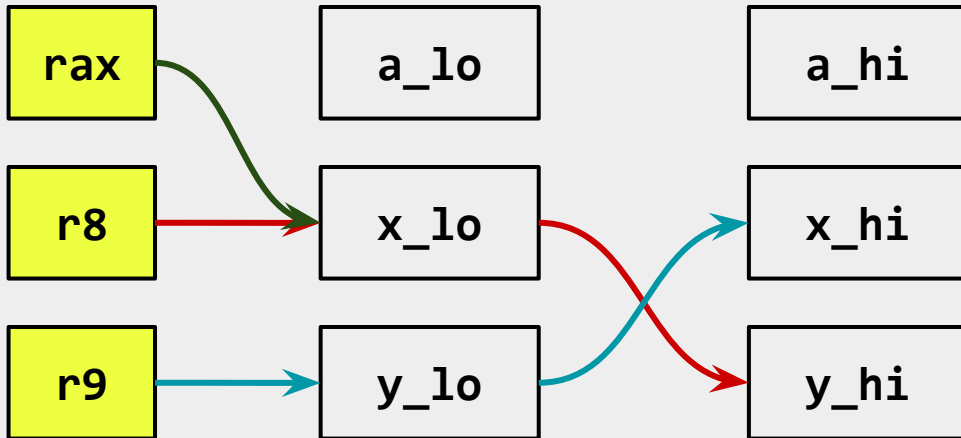
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
    mov  r8, [rsi]  
    mov  r9, [rdx]  
    imul r8, [rdx + 8]  
    imul r9, [rsi + 8]  
    mov  rax, [rsi]
```



Scenario: A4

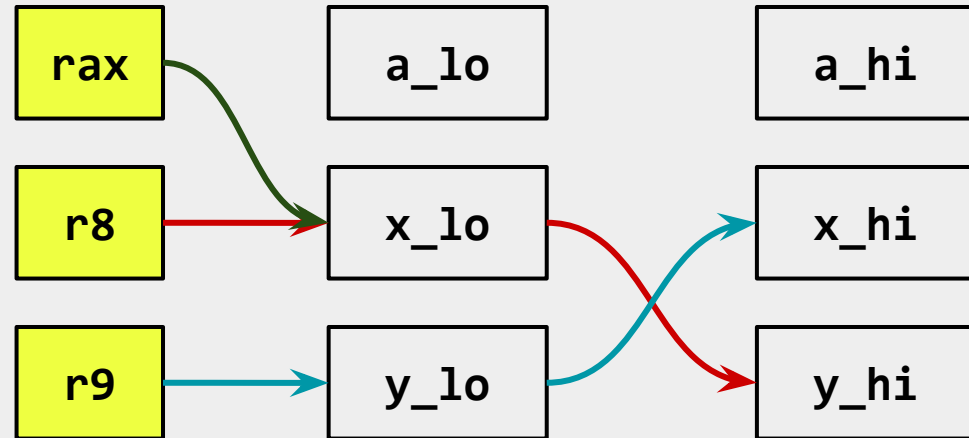
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
    mov  r8, [rsi]  
    mov  r9, [rdx]  
    imul r8, [rdx + 8]  
    imul r9, [rsi + 8]  
    mov  rax, [rsi]  
    mul  qword [rdx]
```



Scenario: A4

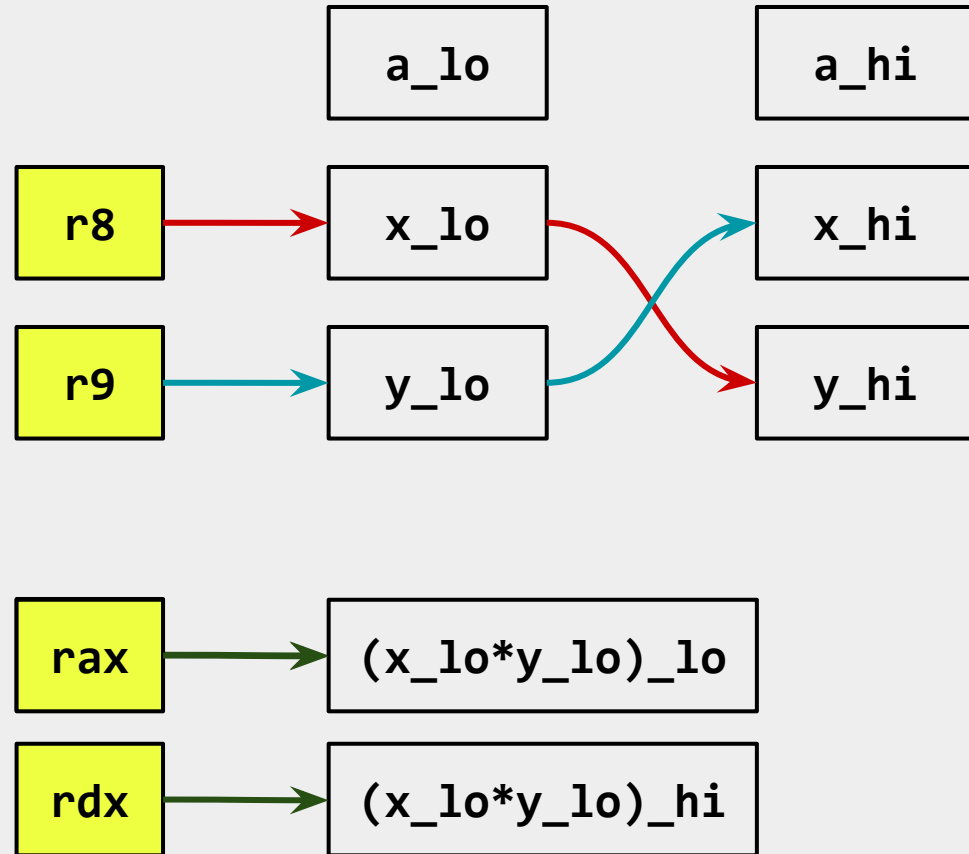
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
mov r8, [rsi]  
mov r9, [rdx]  
imul r8, [rdx + 8]  
imul r9, [rsi + 8]  
mov rax, [rsi]  
mul qword [rdx]
```



Scenario: A4

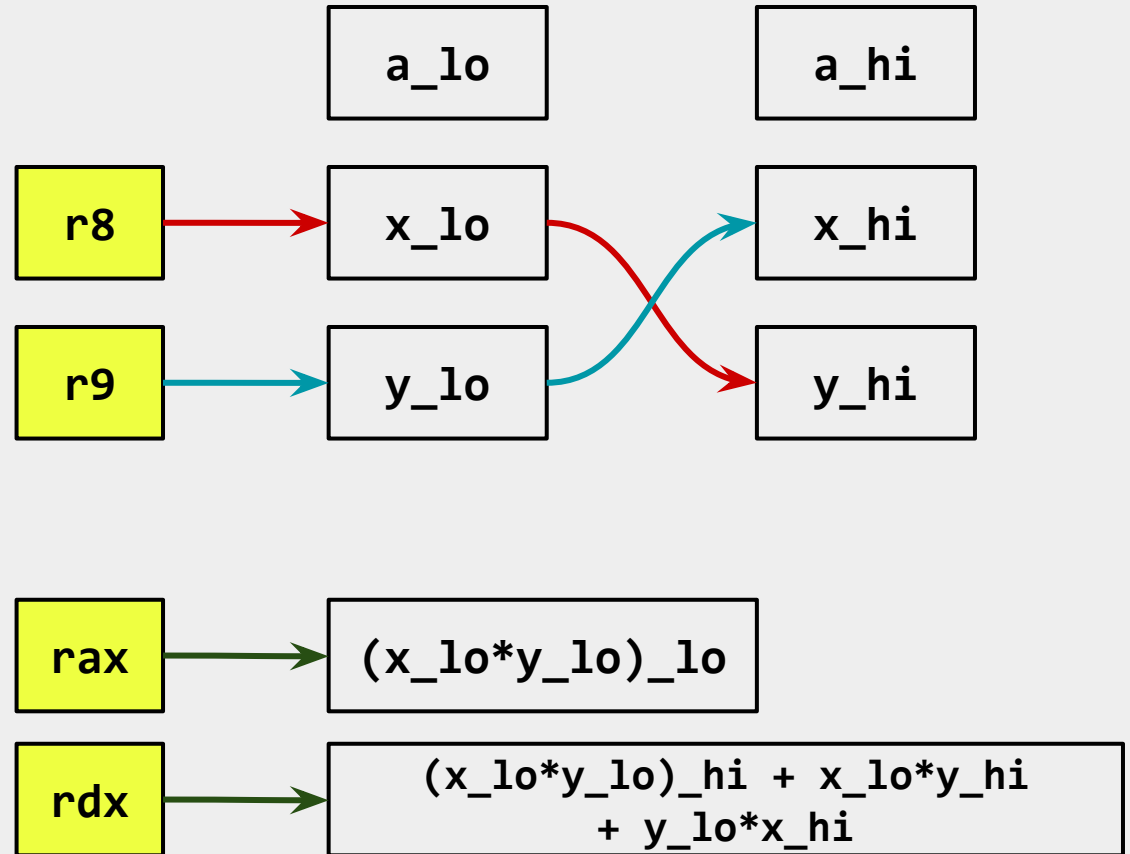
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
mov r8, [rsi]  
mov r9, [rdx]  
imul r8, [rdx + 8]  
imul r9, [rsi + 8]  
mov rax, [rsi]  
mul qword [rdx]  
add rdx, r8  
add rdx, r9
```



Scenario: A4

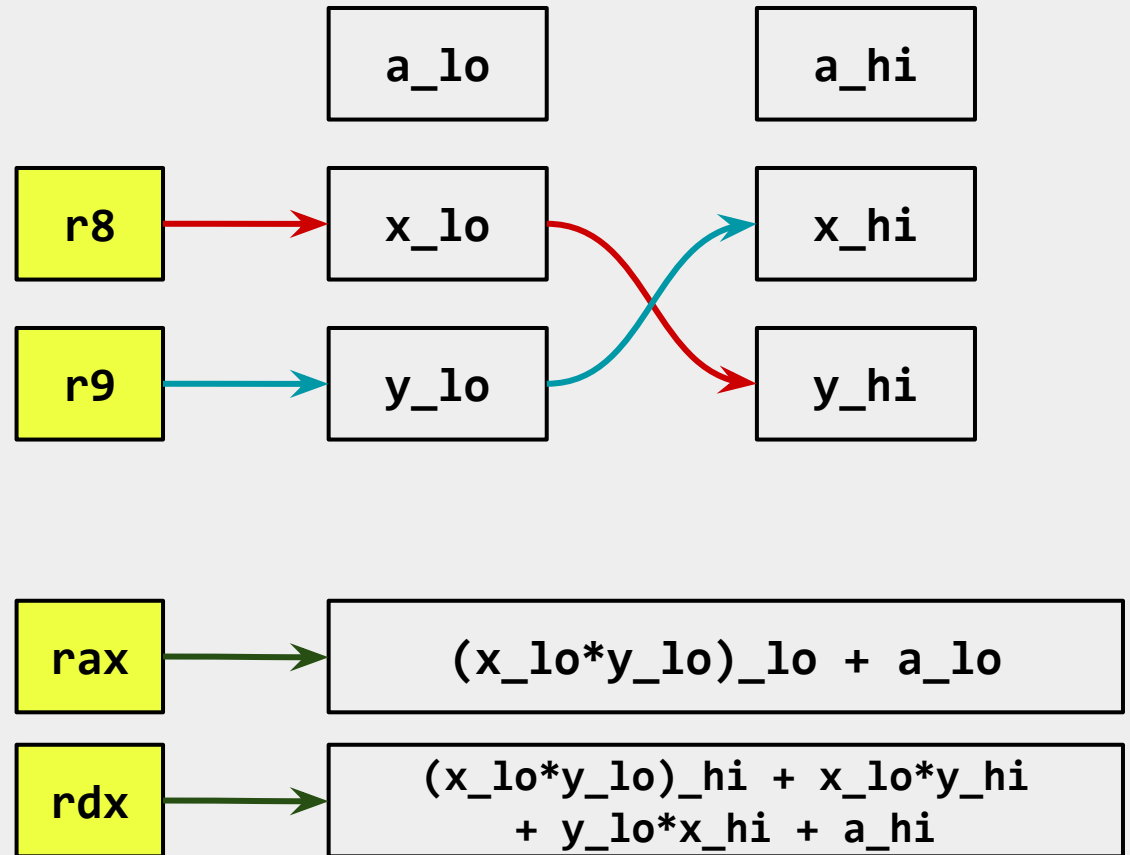
```
global mac
```

```
section .text
```

```
; rdi: a  
; rsi: x  
; rdx: y
```

```
mac:
```

```
mov r8, [rsi]  
mov r9, [rdx]  
imul r8, [rdx + 8]  
imul r9, [rsi + 8]  
mov rax, [rsi]  
mul qword [rdx]  
add rdx, r8  
add rdx, r9  
add rax, [rdi]  
adc rdx, [rdi + 8]
```



Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Co w tym przypadku robi instrukcja xor?

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Co w tym przypadku robi instrukcja xor?

Zeruje rejestr rbx.

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Po co jest nam instrukcja nop?

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Po co jest nam instrukcja nop?

Wyrównuje adres docelowy skoku (instrukcja jne).

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Dlaczego przepisujemy ebx na edi?

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Dlaczego przepisujemy ebx na edi?

Bo pierwszy argument funkcji jest przekazywany w rejestrze rdi.

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Dlaczego iterujemy po ebx?

Scenario: A5

```
void foo(void) {  
    int i;  
    for (i = 0; i < 100; ++i)  
        bar(i);  
}
```

```
push rbx  
xor ebx,ebx  
nop  
mov edi,ebx  
add ebx,0x1  
call 0x0  
cmp ebx,0x64  
jne 0xf1 ;skok o -15 bajtów, do 0x8  
pop rbx  
ret
```

Dlaczego iterujemy po ebx?

Bo ebx nie zostanie zmodyfikowany przez bar.

Scenario: A6

Napisz w asemblerze program *hello world*.

```
global _start
```

```
section .rodata ; dane tylko do odczytu  
hello db "Hello World!"
```

Scenario: A6

Napisz w asemblerze program *hello world*.

```
NEW_LINE equ 10

global _start

section .rodata ; dane tylko do odczytu
hello db "Hello World!"
new_line db NEW_LINE
```

Scenario: A6

Napisz w asemblerze program *hello world*.

```
NEW_LINE equ 10

global _start

section .rodata ; dane tylko do odczytu
hello db "Hello World!"
new_line db NEW_LINE

section .text ; kod wykonywalny
_start:
```

Scenario: A6

Napisz w asemblerze program *hello world*.

```
SYS_WRITE equ 1

STDOUT    equ 1
NEW_LINE  equ 10

global _start

section .rodata                                ; dane tylko do odczytu
hello     db "Hello World!"
new_line  db NEW_LINE

section .text                                   ; kod wykonywalny
_start:
    mov eax, SYS_WRITE
    mov edi, STDOUT
```


Scenario: A6

Napisz w asemblerze program *hello world*.

```
SYS_WRITE equ 1

STDOUT    equ 1
NEW_LINE  equ 10

global _start

section .rodata                                ; dane tylko do odczytu
hello     db "Hello World!"
new_line  db NEW_LINE

section .text                                   ; kod wykonywalny
_start:
    mov eax, SYS_WRITE
    mov edi, STDOUT
    mov rsi, hello
    mov edx, new_line - hello + 1
    syscall
```

Scenario: A6

Napisz w asemblerze program *hello world*.

```
SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1
NEW_LINE  equ 10

global _start

section .rodata                ; dane tylko do odczytu
hello     db "Hello World!"
new_line  db NEW_LINE

section .text                  ; kod wykonywalny
_start:
    mov eax, SYS_WRITE
    mov edi, STDOUT
    mov rsi, hello
    mov edx, new_line - hello + 1
    syscall
    mov eax, SYS_EXIT
    xor edi, edi
    syscall
```

Scenario: A6

Napisz w asemblerze program *hello world*.

```
SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1
NEW_LINE  equ 10

global _start

section .rodata                ; dane tylko do odczytu
hello     db "Hello World!"
new_line  db NEW_LINE

section .text                  ; kod wykonywalny
_start:
    mov eax, SYS_WRITE
    mov edi, STDOUT
    mov rsi, hello
    mov edx, new_line - hello + 1
    syscall
    mov eax, SYS_EXIT
    xor edi, edi
    syscall
```



★ A recap from the last week

★ Scenario: Q&A

★ **First and next steps in gdb**

Let us see... gdb

```
$ nasm -f elf64 -F dwarf -g hello.asm -o hello
```

```
$ ld -o hello hello.o
```

```
$ gdb --args hello 1 2 3 4
```

```
(gdb) b 28
```



set breakpoint at line 28

```
(gdb) r
```



run the program

```
(gdb) bt
```



see the backtrace

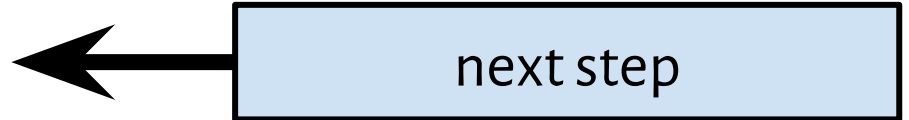
Let us see... gdb

```
$ nasm -f elf64 -F dwarf -g hello.asm -o hello
```

```
$ ld -o hello hello.o
```

```
$ gdb --args hello 1 2 3 4
```

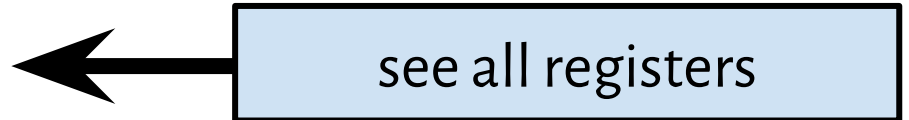
```
(gdb) si
```



```
(gdb) p $rsi
```



```
(gdb) i r
```



Let us see... gdb

```
$ nasm -f elf64 -F dwarf -g hello.asm -o hello
```

```
$ ld -o hello hello.o
```

```
$ gdb --args hello 1 2 3 4
```

```
(gdb) p *(char *) [addr]
```



print a char

```
(gdb) x/s [addr]
```



examine a string

```
(gdb) info symbol [addr]
```



find the nearest symbol

Assignment #1

Expected in:

<https://svn.mimuw.edu.pl/repos/SO/studenci/<login>/zadanie1>

with a proper filename and no additional files

Expected by:

12 March 2018, 8 p.m.