CS 354 – Lecture 9

(1)

Review

1. call
2. ret
3. Locals
4. parameters

%esp → main ← %ebp

%esp → func

func
-8
-4
old ebp (+4) ← %ebp
ret addr (+8)
parameter
main

After func is done,

%esp → main ← %ebp

```
push %ebp
movl %esp, %ebp
```

②

increment:

```
push %ebp
movl %esp, %ebp
movl 8(%ebp), %ecx
addl $1, %ecx
movl %ecx, %eax
movl %ebp, %esp
popl %ebp
ret
```

%ebp → [old %ebp] ← %esp
+4 ret. addr
+8 10 f
main:

After: pop %ebp

ret addr ← %esp
10 f
main.
%ebp →

main:

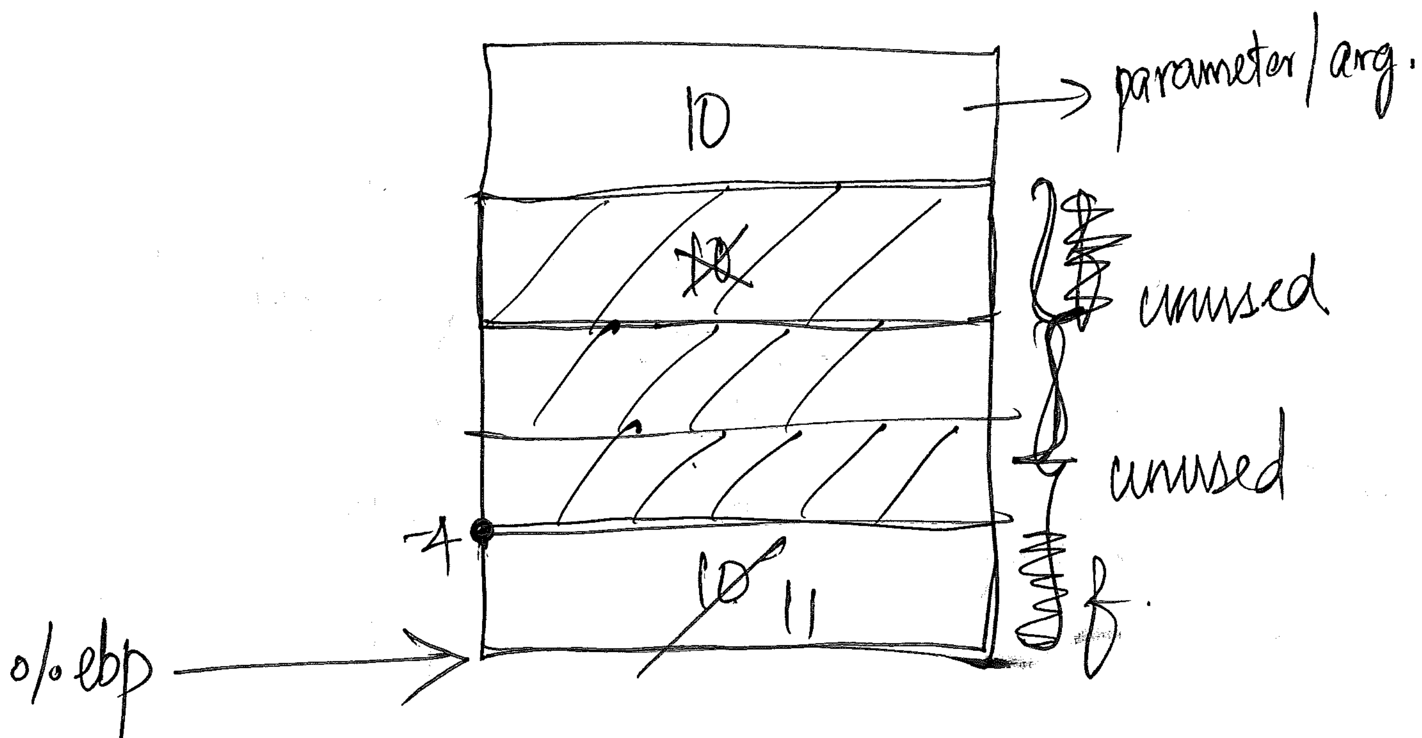
~~movl $10, 4(%esp)~~

push $10

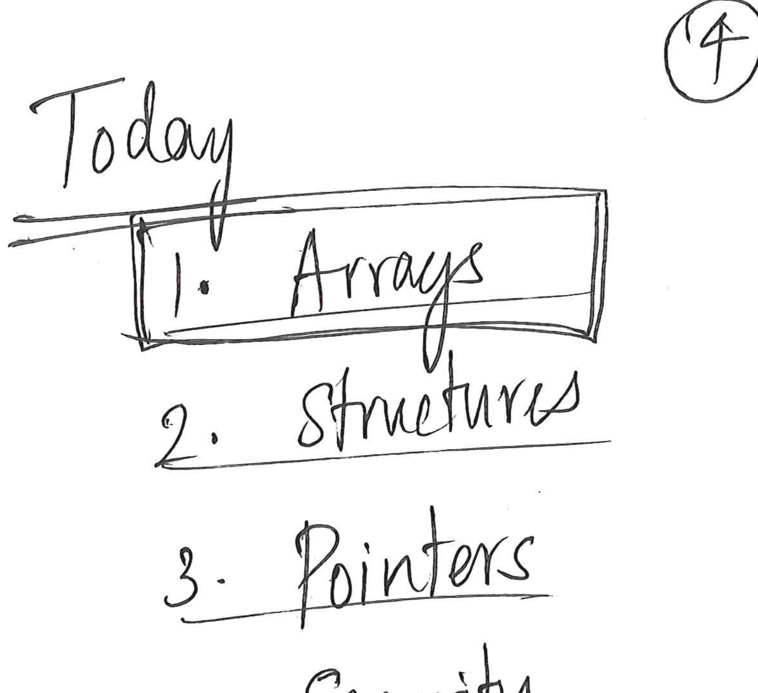
```
subl $4, %esp
call increment
addl $4, %esp
```

increment:

main:



④



int a[3];

a
0 1 2
0x100 0x104 0x108

stack.

0x100 addr incr

a[0] 10 0x104

a[1] 20 0x108

a[2] 30

%ebp

(5)

int a[3] = {10, 20, 30};

%ebx = 0x100

int a[3];	
a[0] = 10;	movl $10, (%ebx)
a[1] = 20;	movl $20, 4(%ebx)
a[2] = 30;	movl $30, 8(%ebx)

Structures

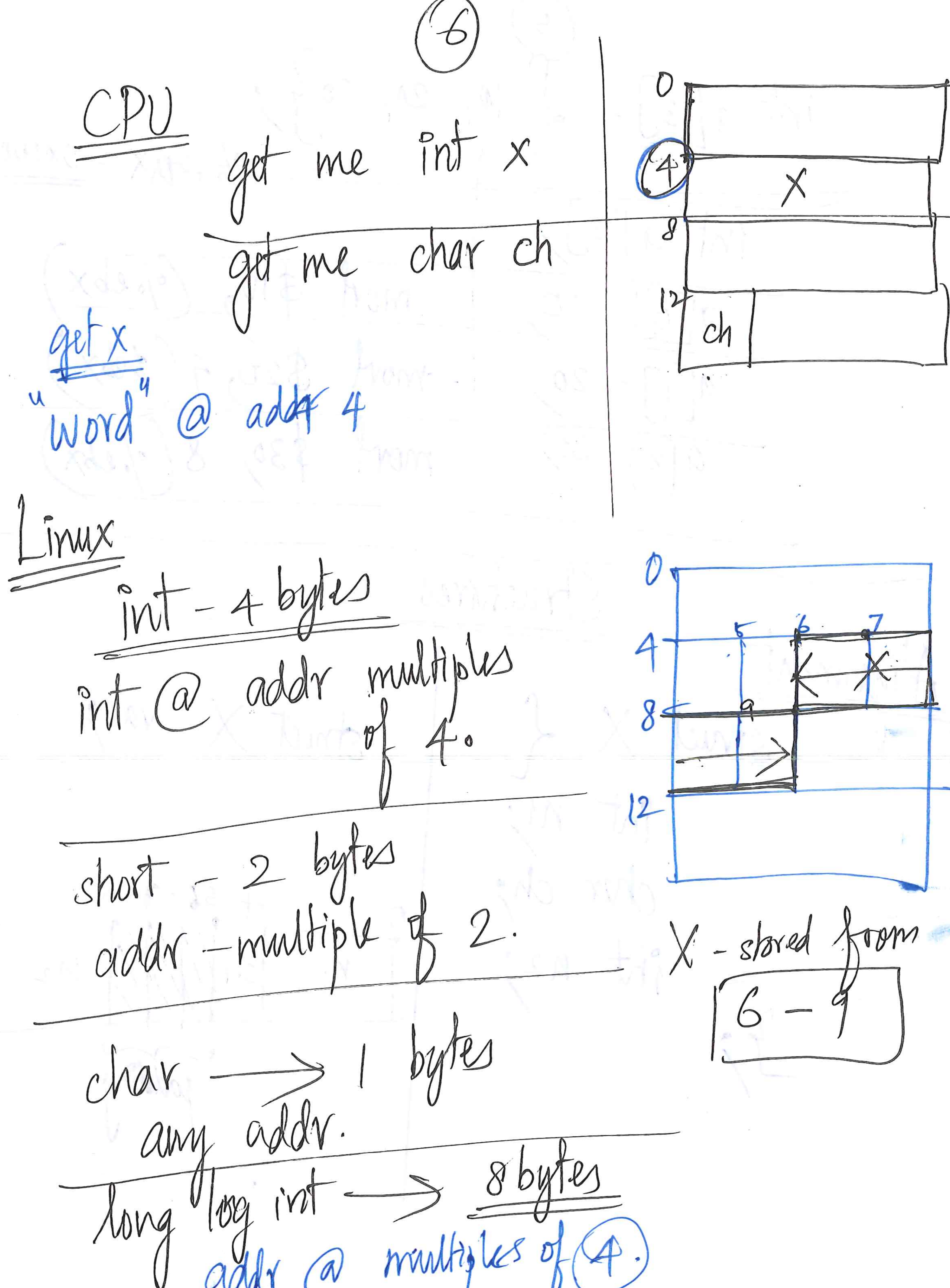
Alignment

```
struct X {
    int n1;
    char ch;
    int n2;
};
```

```
struct X var;
```

n1	ch	padding	n2

Offsets: 0 (n1), 4 (ch), 5, 6, 7 (padding), 8 (n2)


6
CPU
get me int x
get me char ch
get x
"word" @ addr 4
0
4
8
12
x
ch
Linux
int - 4 bytes
int @ addr multiples of 4.
short - 2 bytes
addr - multiple of 2.
char → 1 bytes
any addr.
long long int → 8 bytes
addr @ multiples of 4.
0
4
8
12
5
6
7
9
X
X - stored from
6 - 9

```
struct X {
    int n1;
    int n2;
    char ch;
}
```

```
struct X a[3];
```

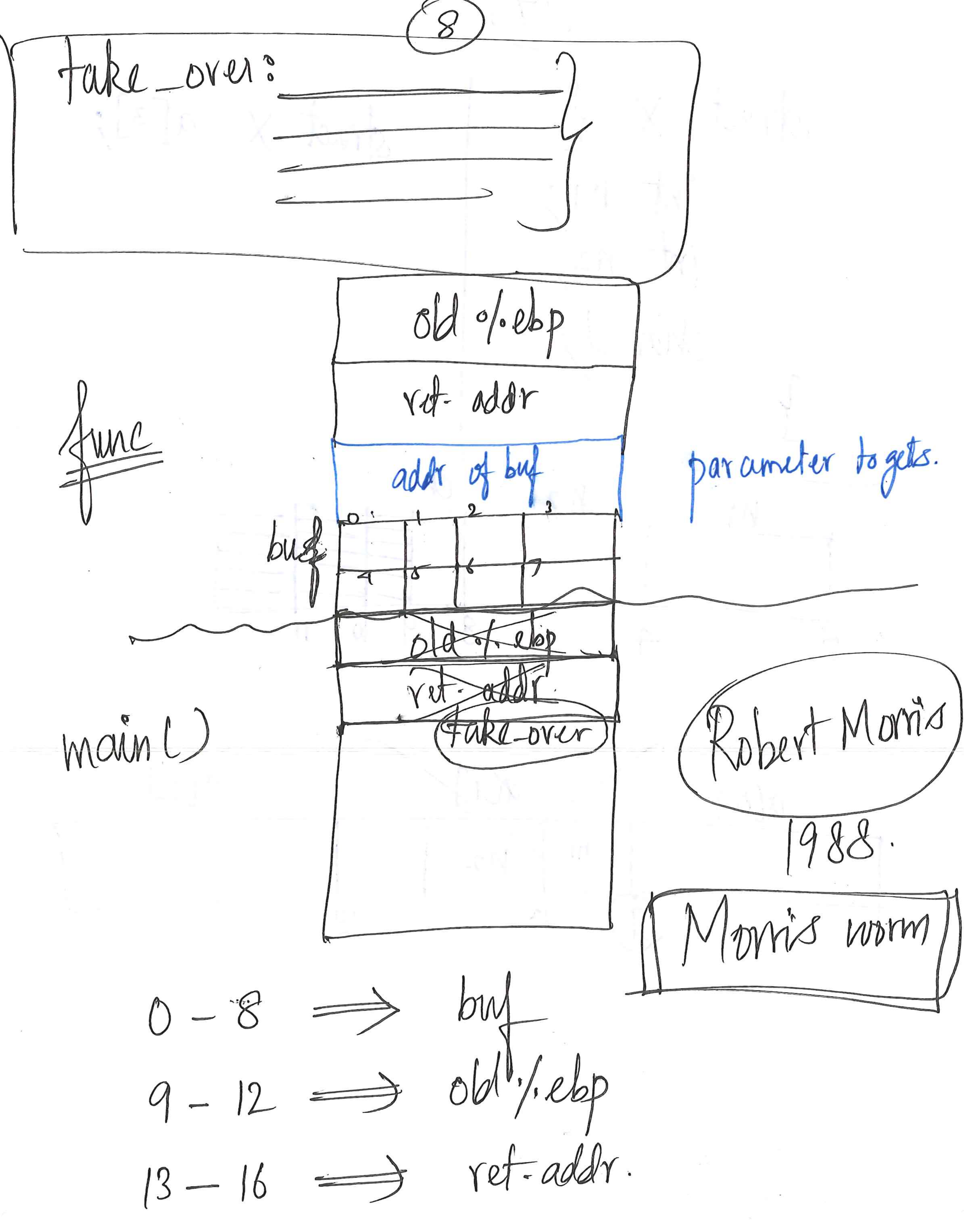
n1 | n2 | ch

0 4 8 9 10 11

a[0] a[1] a[2]

n1 n2

0 9 13 18


8
take_over:
old %ebp
ret-addr
func
addr of buf
parameter to gets.
buf
old %ebp
ret-addr
take_over
main()
Robert Morris
1988.
Morris worm
0 - 8 ⟹ buf
9 - 12 ⟹ old %ebp
13 - 16 ⟹ ret-addr.

Security:

1. ASLR - Address Space Layout Randomization

0XFFFF000D

addr of stack

random addr

0

16

14

12

start of ~~start~~ stack

2. Canary values

(10)

special loc.

canary

0	1	2	3
4	5	6	7

canary

old %ebp

ret addr

"Stack Smashing"

Arrays

```
000004ed <mystery>:
 4ed: 55                    push   %ebp
 4ee: 89 e5                 mov    %esp,%ebp
 4f0: 83 ec 10              sub    $0x10,%esp
 4f3: c7 45 f8 00 00 00 00  movl   $0x0,-0x8(%ebp)
 4fa: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%ebp)
 501: eb 18                 jmp    51b <mystery+0x2e>
 503: 8b 45 fc              mov    -0x4(%ebp),%eax
 506: 8d 14 85 00 00 00 00  lea    0x0(,%eax,4),%edx
 50d: 8b 45 08              mov    0x8(%ebp),%eax
 510: 01 d0                 add    %edx,%eax
 512: 8b 00                 mov    (%eax),%eax
 514: 01 45 f8              add    %eax,-0x8(%ebp)
 517: 83 45 fc 01           addl   $0x1,-0x4(%ebp)
 51b: 8b 45 fc              mov    -0x4(%ebp),%eax
 51e: 3b 45 0c              cmp    0xc(%ebp),%eax
 521: 7c e0                 jl     503 <mystery+0x16>
 523: 8b 45 f8              mov    -0x8(%ebp),%eax
 526: c9                    leave
 527: c3                    ret

00000528 <main>:
 528: 55                    push   %ebp
 529: 89 e5                 mov    %esp,%ebp
 52b: 83 ec 10              sub    $0x10,%esp
 52e: c7 45 f0 01 00 00 00  movl   $0x1,-0x10(%ebp)
 535: c7 45 f4 02 00 00 00  movl   $0x2,-0xc(%ebp)
 53c: c7 45 f8 03 00 00 00  movl   $0x3,-0x8(%ebp)
 543: 6a 03                 push   $0x3
 545: 8d 45 f0              lea    -0x10(%ebp),%eax
 548: 50                    push   %eax
 549: e8 9f ff ff ff        call   4ed <mystery>
 54e: 83 c4 08              add    $0x8,%esp
 551: 89 45 fc              mov    %eax,-0x4(%ebp)
 554: b8 00 00 00 00        mov    $0x0,%eax
 559: c9                    leave
 55a: c3                    ret
```

Structures

What is the size (in bytes) for the following structures on a 32-bit machine installed with the Linux Operating System?

sizeof(int) = 4 bytes
sizeof(short) = 2 bytes
sizeof(char) = 1 byte

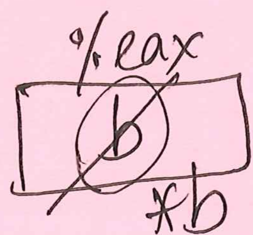
Structure	Size (in bytes)

`struct foo {`
`int d1;`
`char c1;`
`int d2;`
`}`
	4 + 4 + 4 = 12

`struct foo {`
`int d1;`
`char c1;`
`int d2;`
`char c2;`
`short s;`
`};`
	4 + 4 + 4 + 2 + 2 = 16
	0 4 8 12 16
	d1 c1 d2 c2 s

`struct foo {`
`int d1;` — 4
`int d2;` — 4
`char c1;` — 1
`char c2;` — 1
`short s;` — 2
`};`
	4 + 4 + 4 = 12

`struct foo {`
`char c1;` → 4
`int d1;` → 4
`short s;` → 4
`int d2;` → 4
`char c2;` → 4
`};`
	20



Pointers

Assume variables a and b are stored at `-0x8(%ebp)` and `-0x4(%ebp)` respectively. Write the equivalent C expressions for the following assembly snippets.

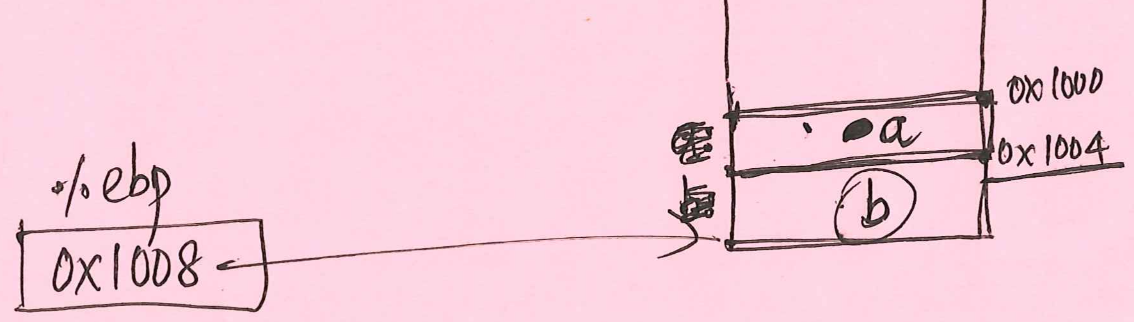
S.No.	Assembly Instruction	Corresponding C code

1	`movl -0x4(%ebp),%eax`
	`movl (%eax),%eax`
	`movl %eax,-0x8(%ebp)`
		a = *b

2	`leal -0x4(%ebp),%eax` `movl %eax,-0x8(%ebp)`	a = &b
3	`movl -0x4(%ebp),%eax` `movl %eax,-0x8(%ebp)`	a = b

4	`movl -0x4(%ebp),%eax`
	`movl (%eax),%edx`
	`movl -0x8(%ebp),%eax`
	`movl %edx,(%eax)`
		*a = *b

5	`movl -0x4(%ebp), %eax`
	`movl (%eax), %edx`
	`movl -0x8(%ebp), %eax`
	`movl (%eax), %eax`
	`addl %eax, %edx`
	`movl -0x4(%ebp), %eax`
	`movl %edx, (%eax)`	



Jumps

In the disassembled version of the machine code shown below, instructions at addresses 0x66, 0x6c, and 0x73 use relative encoding to encode jump targets. All the values in the disassembled code below (including the address column) are represented using hexadecimal numbers.

Address	Opcode	Assembly Code	
0x62:	83 7d 08 00	cmpl	$0x0,0x8(%ebp)
0x66:	74 0d	je	**X**
0x68:	83 7d 0c 00	cmpl	$0x0,0xc(%ebp)
0x6c:	74 07	je	**Y**
0x6e:	b8 01 00 00 00	mov	$0x1,%eax
0x73:	eb 05	jmp	**Z**
0x75:	b8 00 00 00 00	mov	$0x0,%eax
0x7a:	85 c0	test	%eax,%eax
0x7c:	0f 9e c0	setle	%al
0x7f:	88 45 ff	mov	%al,-0x1(%ebp)

What is the value of the addresses X, Y and Z? The values of X, Y, and Z should be written in hexadecimal notation. Remember the addresses X, Y, and Z, in assembly code will refer to actual addresses (e.g. 0x75) even though relative encoding is used in the machine code (opcode).

ANSWER:

X = 0x________ Y = 0x________ Z = 0x________