Lecture 7
(In-class)

Review

Decisions

Function Call

Review:

human ⇒ .c — C language
↓ compiler
.S — assembly
↓ assembler
.o (binary code) → executable
linker

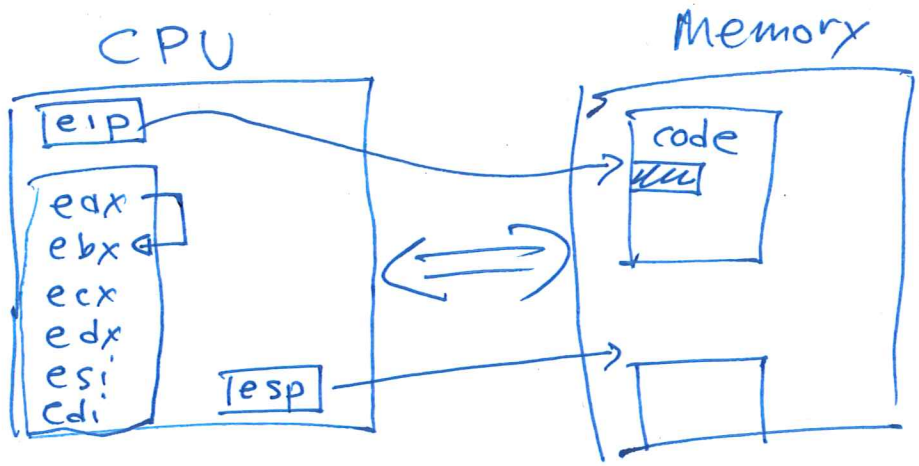
[x86] 32-bit

Registers

memory on CPU

⇒ gen purpose

32-bit registers



⇒ eip: hold address of next instruction to fetch

⇒ stack pointer: esp ⇒ top of stack

Last Time:

mov

e.g. movl $10, %eax (immediate)

FAST movl %eax, %ebx

(copy: eax still has value after)

might be slow (memory)

movl (%eax), %ebx

compute address:
contents of eax
fetch 4 bytes @ addr
put result => ebx

depends on cache

movl offset(reg1, reg2, mul)

mul → 1, 2, 4, 8

compute addr:
offset + reg1 + reg2 * mul

useful in array access

Decisions :

if

while

⇒ 2 types of instructions

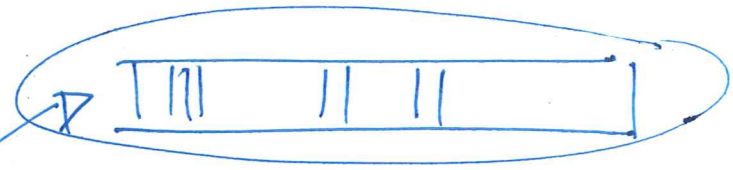
1) perform comparison :

cmpl b, a

2) change control flow of prog. based on result of comparison

jmp family of instructions

registers:
one more
%eflags



32-bit reg

[state about CPU]

cmpl b, a

=> result of comparison

=> condition codes

bits in that we can use to make [decisions]

[e.g. zero flag, sign flag, overflow flag]

jmp target => label in .S (equivalent to some inst. address)

change eip to target

unconditionally
jmp target

conditionally
cmpl b, a
jg target
inst3
target:

if a > b
else
fall through to next inst.

transfer control

```
cmpl  b, a

je    a == b
jne   a != b
jg    a > b
jge   a ≥ b
jl    a < b
jle   a ≤ b
  ⋮
```

Example 7:

assume
a ⇒ %eax
b ⇒ %ebx

```
if (a > b)
    [a++;]
```

```
                b      a
    cmpl %ebx, %eax
if a>b
    jg do        if a≤b
    jmp dont
do:
    addl $1, %eax
dont:
    inst
```

eax = eax+1

target:
inst1
inst2

```
cmpl %ebx, %eax
jle dont
addl $1, %eax
dont:
```

```
if (a > b)
    a++; ]
else
    a = b;
```

```
        cmpl %ebx, %eax
        jg doadd
        jmp domov
doadd:
        addl $1, %eax
        jmp end
domov:
        movl %ebx, %eax
end:
        ingt
```

zero

brace

```
stmt1;
while (b>0) {
    a++;
    b--;
}
stmt2;
```

semi-colons

```
    jmp loop test
top:
    addl $1, %eax
    subl $1, %ebx
loop test:
    cmpl $0, %ebx
    jg top
```

branches or conditional branches

⋮

other instructions
that set condition codes

thus far:

cmpl b, a ⇒ a-b ⇒

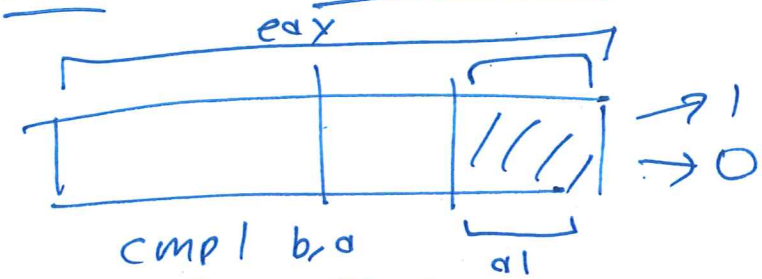
arithmetic instructions do
e.g. addl

also: test b, a ⇒ a & b
bitwise and

⋮

see condition codes

set instruction



cmpl b, a
sete %al

function call / return

in C : easy

to call:

```
foo(1, c);
```

define: → int

```
void foo(int a, int b){
    int c; // stack
    :
    return;
}
```

Goal: same @ assembly level

Questions:

- ⇒ How to call a function?
- ⇒ How to return from function to inst after call?
- ⇒ How to obtain return value?
- ⇒ How to pass parameters?
- ⇒ How to alloc space for local vars?
- ⇒ How to access parameters/local variables?
- ⇒ How to use registers w/ in function?

Calling Convention

set of rules to follow
when call / return

x86 convention: cdecl

gcc on x86

new instructions:

call target

similar to jmp

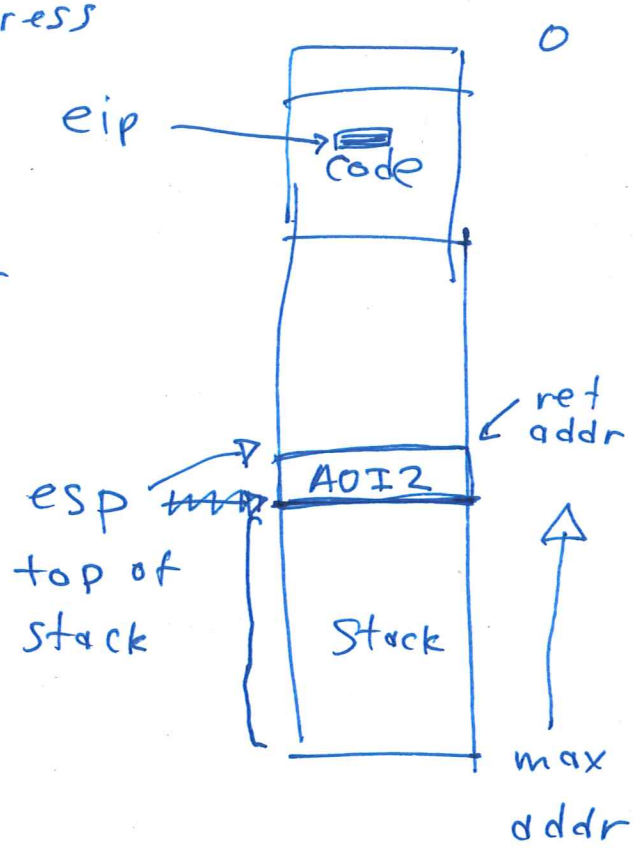
⇒ transfers control to target (jmp)

⇒ push return address onto stack

ret

⇒ pops ret addr off stack

⇒ transfers control ⇒ ret addr (jmp)



example

inst1.

call foo.

inst2 ← addr of inst 2 ⇒ AOI2

Return value:

how to return a value to caller?

Define a Convention

goal: use stack

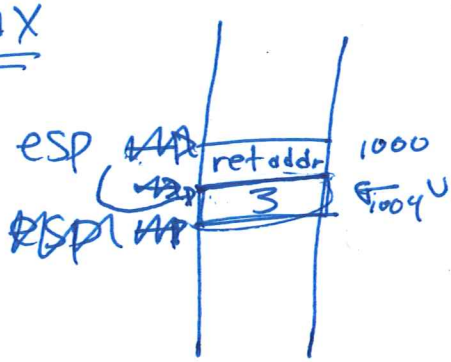
```
int foo( ) {
    :
    return 3;
}
```

```
foo:
    movl $3, 4(%esp)
    ret
```

```
int rc = foo();
printf("%d\n", rc);
```

⇒ print 3

```
main:
→ subl $4, %esp
  call foo
→ movl (%esp), %eax
```

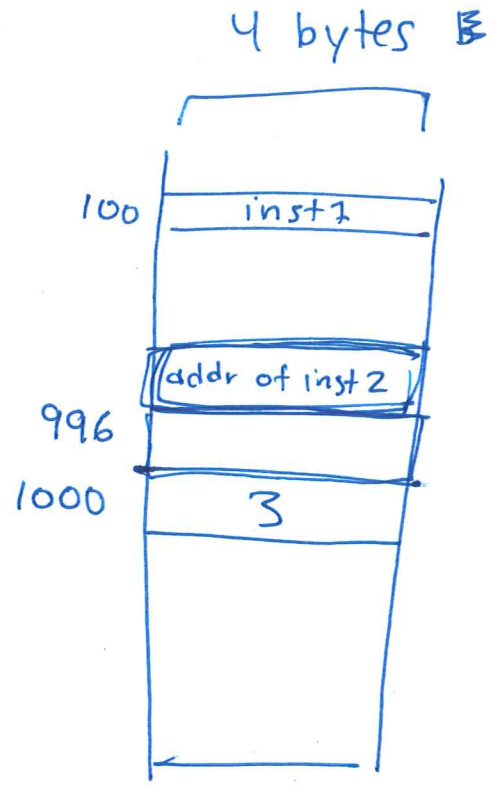


use this for ret value

eip: 100

992
~~996~~

esp: ~~1000~~



foo:
⋮
ret
⇒ eip

mov1 $3, (%esp)

call foo
inst2;

mov1 $996, %esp

return value convention #2:

put return value in register %eax (not memory)

⇒ use this convention

```
func:
    movl $3, %eax
    ret

main:
    call foo
```

→ ← ret val in %eax

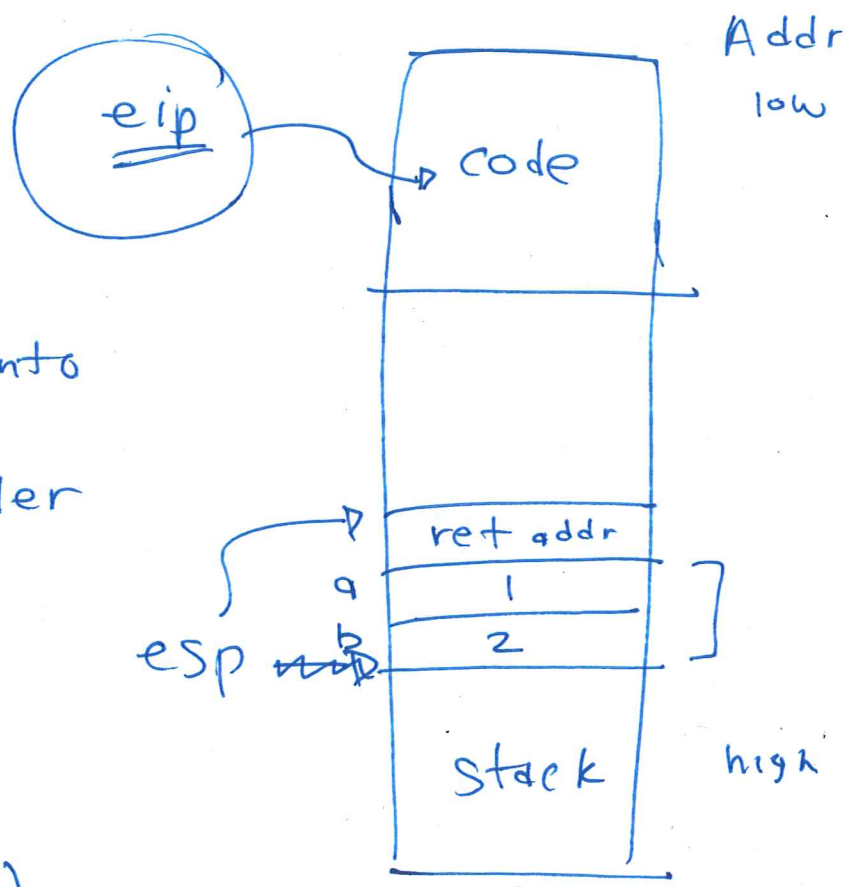
implication

⇒ ~~cant~~ can assume eax will be overwritten

Passing parameters:

```
int func (int a, int b) {
    return a+b;
}
```

use stack:



before call:

push params onto stack in "reverse" order

param N-1

⋮

param 0

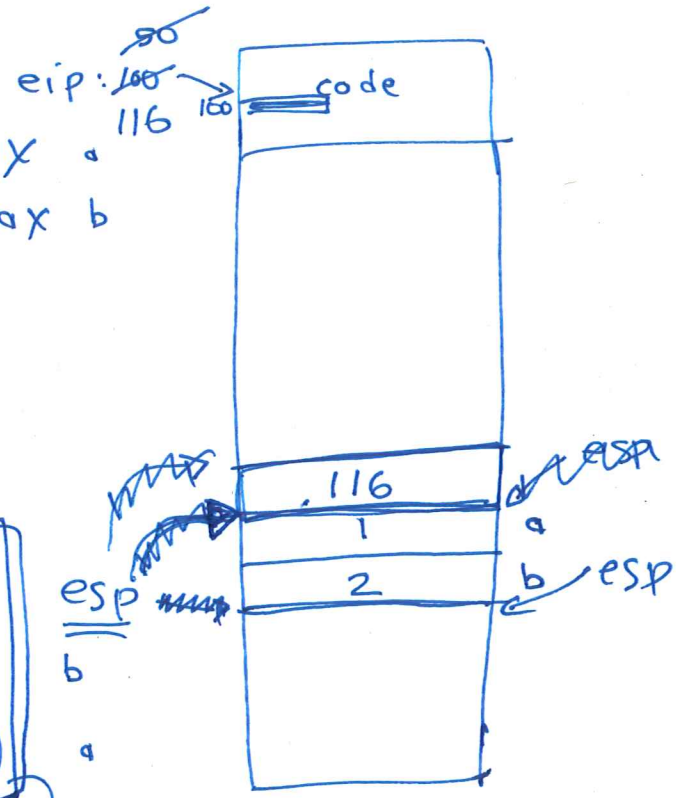
call

func (1, 2)

```
int func (int a, int b) {
    return a+b;
}
```

```
func(1,2);
```

```
func:
  50 : movl  4(%esp), %eax   a
       addl  8(%esp), %eax   b
       ret
```



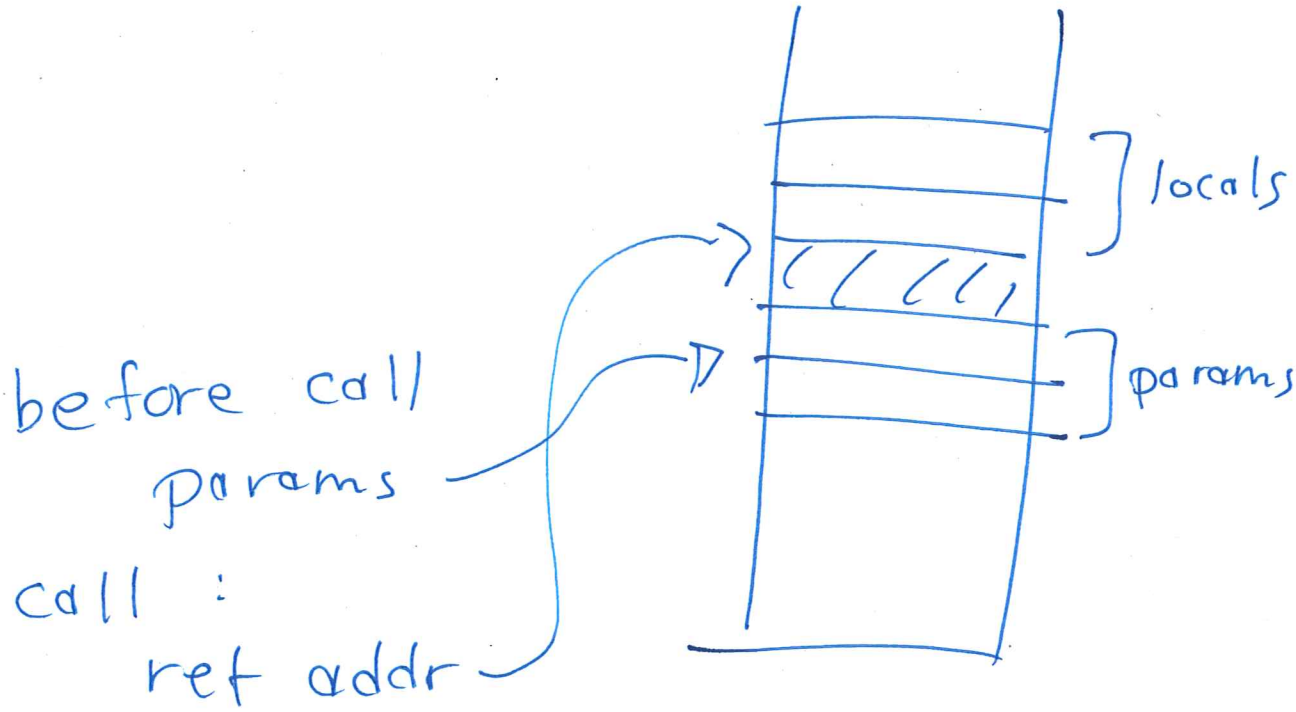

```
main:
100: [subl  $8, %esp]
104: movl  $2, 4(%esp)
108: movl  $1, 0(%esp)
112: call  func
116 : addl  $8, %esp
```

esp
b
a

ret

push1 $2
push1 $1

Space for local variables:



how to make space for
local vars on stack?

```
int func (int a, int b) {
    [int c = 1;] ⇐ local
    return a+b+c;
}
```

allocate room for c on entry, deallocate on return (automatic)

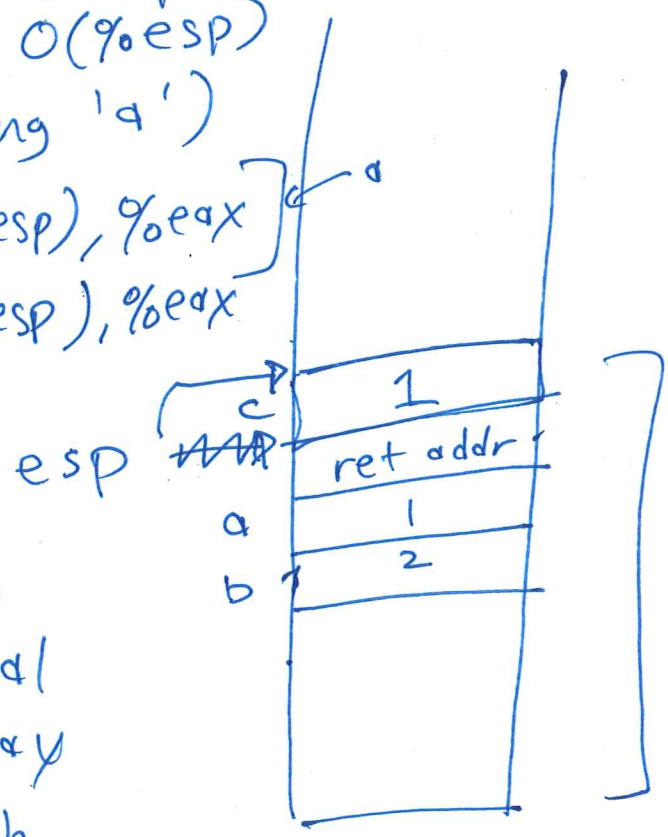
inside func:

need space for c

```
subl $4, %esp
movl $1, 0(%esp)
```

before: (accessing 'a')

```
[movl 4(%esp), %eax]
 add 8(%esp), %eax
```



goal:

to refer to params/local in same way even though esp may be changing during function

base pointer (frame pointer)

new register: (%ebp)

stable (non changing)
pointer into stack
during function call

a : 4 (%ebp)
b : 8 (%ebp)
c : -4 (%ebp)
⋮