

Day 6

354 :

①

~~X~~ ⇐ most "homeworks"

x86 assembly

Today

Basics of Processing

x86 assembly

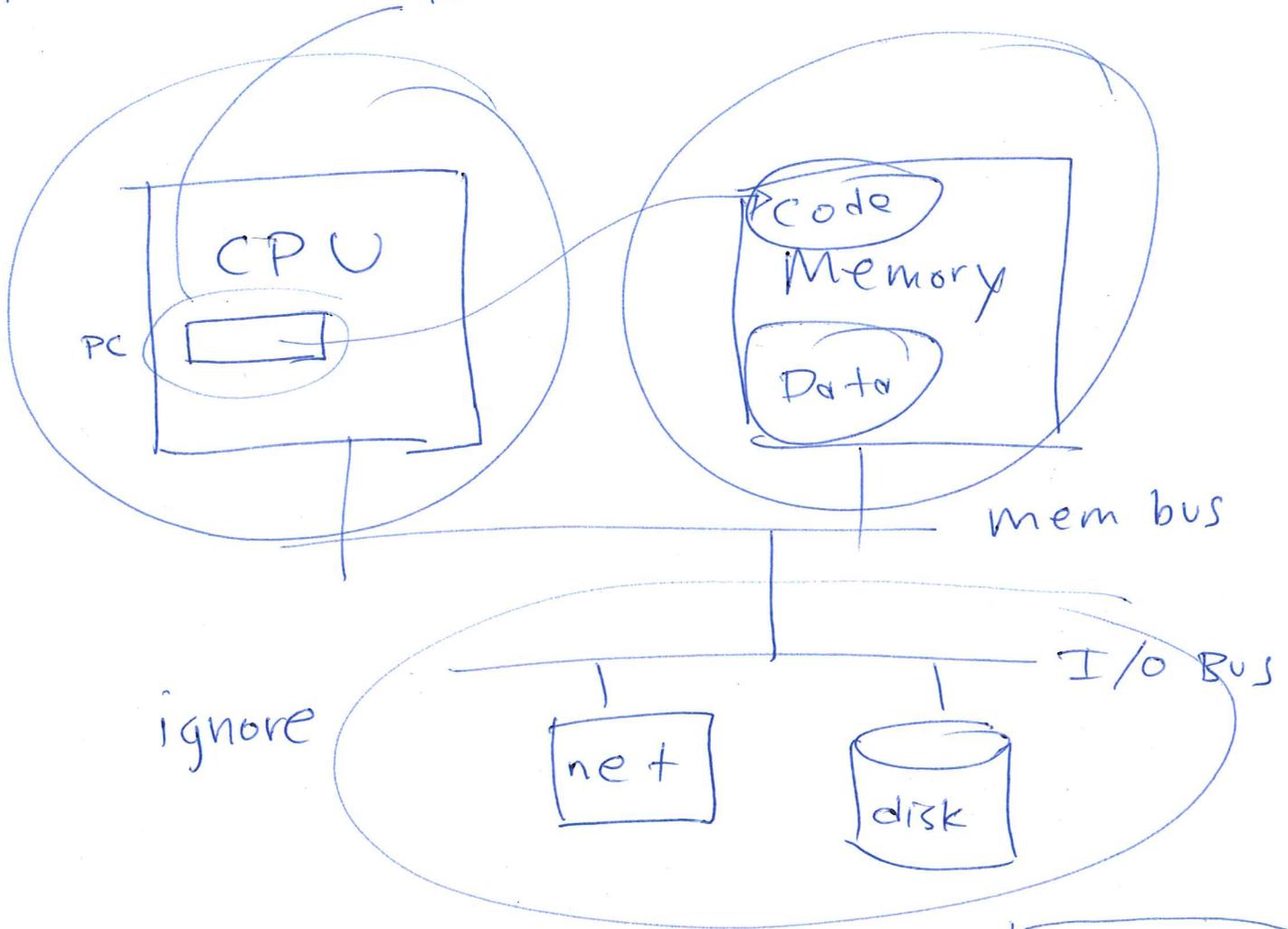
registers

basic instructions

- ⇒ numbers and expressions
- ⇒ variables / memory
- ⇒ decisions / loops

Review

little memory: [register]



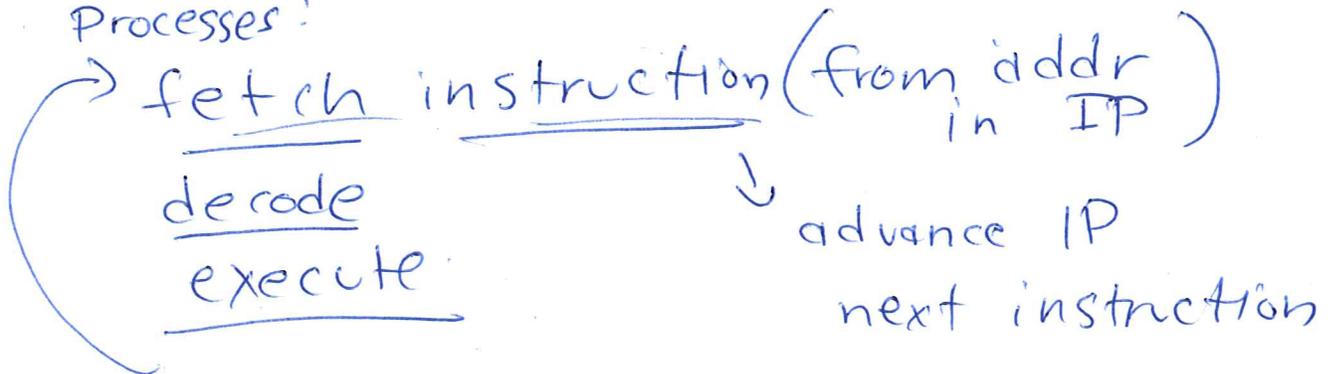
"Processor"

executes instructions

x86
instruction
set

Program counter / instruction pointer

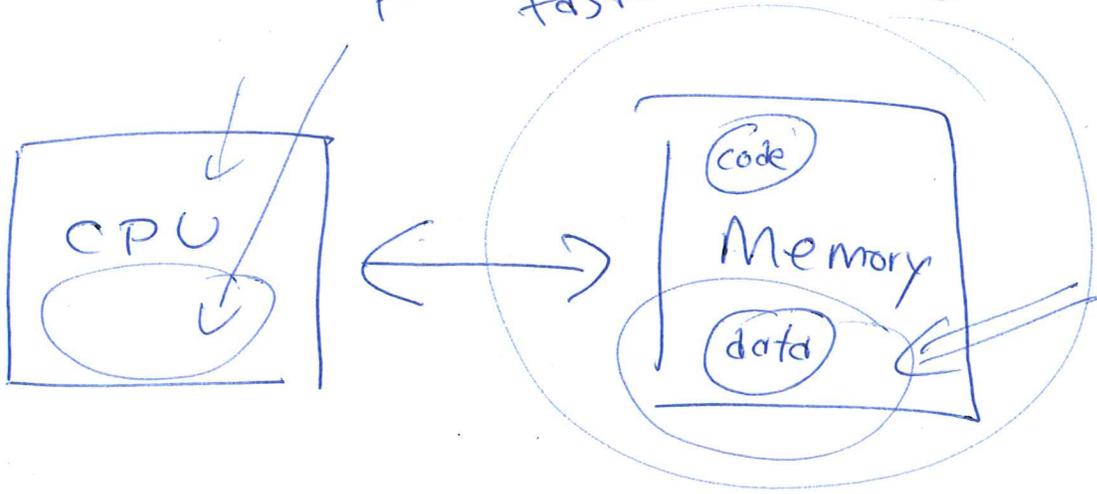
Processes:



Problem:

put smaller, faster memory here (in CPU)

3



large memory

⇒ fast, but not fast enough

CPU: billions inst / sec

1 or more
inst / ns

Memory: latency ~ 100 ns

Fetch inst "very often"

Solution: general purpose registers

some small # of registers
that instructions can
use to put data in,
+ get data from

x86 :

(4)

registers: general purpose

little pieces of memory

each have name

can read / write

▷ 32-bit x86 ⇒ 32 bits
(4 bytes)

⇒ int

"normal"

x86 register names:

⇒ %eax
%ebx
%ecx
%edx
%esi
%edi

32 bits

~~inst set~~

~~r1~~

~~r2~~

~~r3~~

~~r4~~

%eax ⇒ 32 bits

31	...	10
----	-----	----

%ax (16 bits)

%ah %al

8 bits 8 bits

eax

program counter ⇒ %eip

Instruction Sets

5

instructions themselves

⇒ arithmetic

⇒ memory

⇒ control flow

(if, while, call/return)

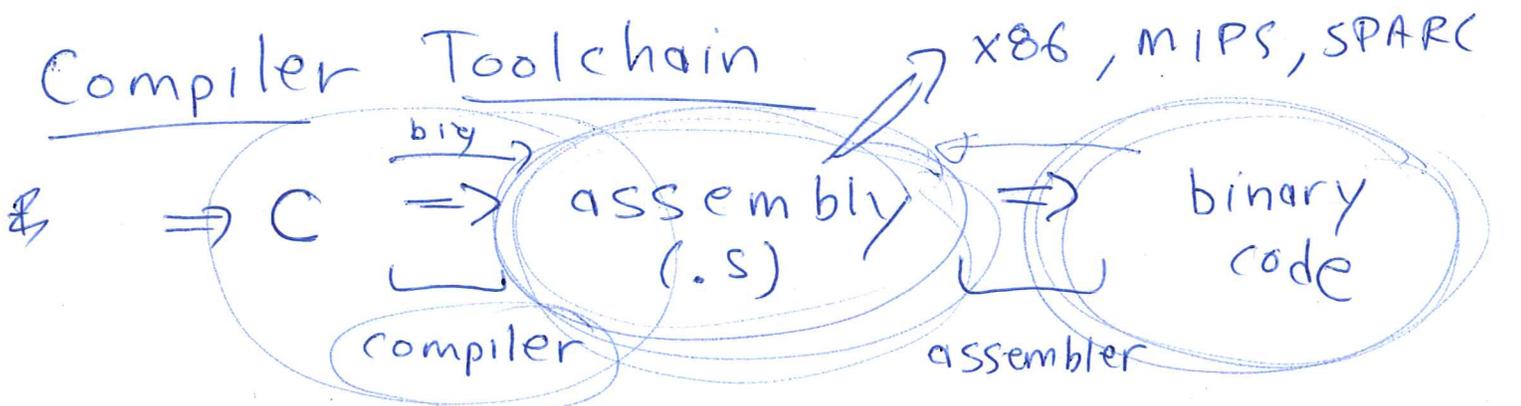
basic data types

⇒ 32-bit numbers,
etc.

other stuff

⇒ Input/Output, support OS
(S37)

Compiler Toolchain



add instruction ⇒

110 | 0001 | ...
add

Have registers, want instructions (6)

Number => register

mov instruction

we're using

=> gnu / AT&T

=> ~~NASM~~

MOV

source, destination

name

operand(s) [1 or more]

variants: different sizes of data

movb

byte

movl

"long" (int) 4 bytes

movq

8 bytes

indicate

numbers in .S

registers

immediate value

movl \$15, %eax

it's called move

it's a copy

mov %eax, %ebx



(%eax ~~is~~ has
same contents
after]

operators

(8)

addl source, destination

$$\text{dest} = \text{dest} + \text{src}$$

subl source, dest

$$\text{dest} = \text{dest} - \text{src}$$

imull source, dest

$$\text{dest} = \text{dest} * \text{source}$$

(alt: imull aux, source, dest)

$$\text{dest} = \text{aux} * \text{source}$$

idivl arg

number to be divided: $\frac{[\%edx:\%eax]}{(\text{arg})}$

result of div \Rightarrow $\%eax$ \nearrow reg

mod \Rightarrow $\%edx$

Problem #1

Write assembly to:

- move value 1 into %eax
- add 10 to it and put result into %eax

```
movl $1, %eax
addl $10, %eax
```

movl addl

Problem #2

Expression: $3 + 6 * 2$

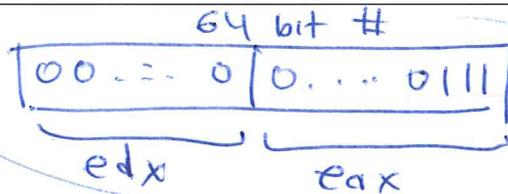
or fewer

Use one register (%eax), and 3 instructions to compute this piece-by-piece

```
movl $6, %eax
imull $2, %eax
addl $3, %eax
```

Problem #3

```
movl $0, %edx
movl $7, %eax
movl $3, %ebx
idivl %ebx
movl %eax, %ecx
movl $0, %edx
movl $9, %eax
movl $2, %ebx
idivl %ebx
movl %edx, %eax
addl %ecx, %eax
```



7/3

Write simple C expression that is equivalent to these instructions

$$7/3 + 9\%2$$

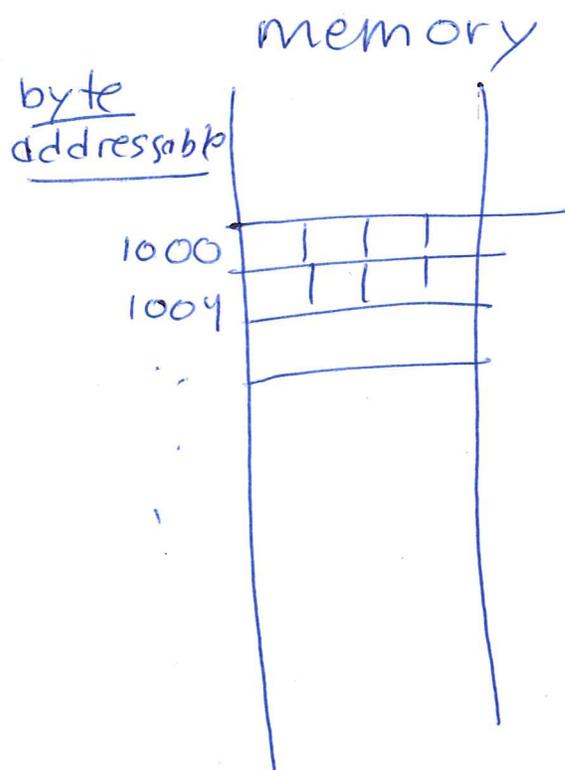
$$2 + 1 = 3$$

memory : how to access memory

9

=> many instructions take
mem addr (in various forms)
as one operand

=> x86 has many different
ways of specifying
a memory address



Memory Address Forms:

(10)

Absolute address: (a number)

movl 1000, %eax

load from addr: 1000
value \Rightarrow %eax

~~addl~~ addl \$1, %eax

movl %eax, 1000

1000	23
1004	3
1008	10

Indirect address:

\rightarrow address is in a register
e.g. %eax

%eax: 1004

movl (%eax), %ebx \Downarrow 3

Base + displacement

\Rightarrow movl 8(%eax), %ebx

addr: contents of eax + 8 \Rightarrow 1008

%eax: 1000

Indexed :

11

`movl 4(%eax, %ebx), %ecx`

addr : 4 + contents_{eax} + contents_{ebx}

Most
General :

`movl 8(%eax, %ebx, 4), %ecx`

addr : 8 + contents_{eax} + 4 * contents_{of ebx}

Problem #4 (from CSAPP 3.1)

Memory

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Registers
 %eax
 %ecx
 %edx

0x100
0x1
0x3

Value of:	
%eax	0x100
0x104	0xAB
0x108	0x108
(%eax)	0xFF
4(%eax)	0xAB
9(%eax, %edx)	0x11
260(%ecx, %edx)	0x13
0xFC(,%ecx, 4)	0xFF
(%eax, %edx, 4)	0x11

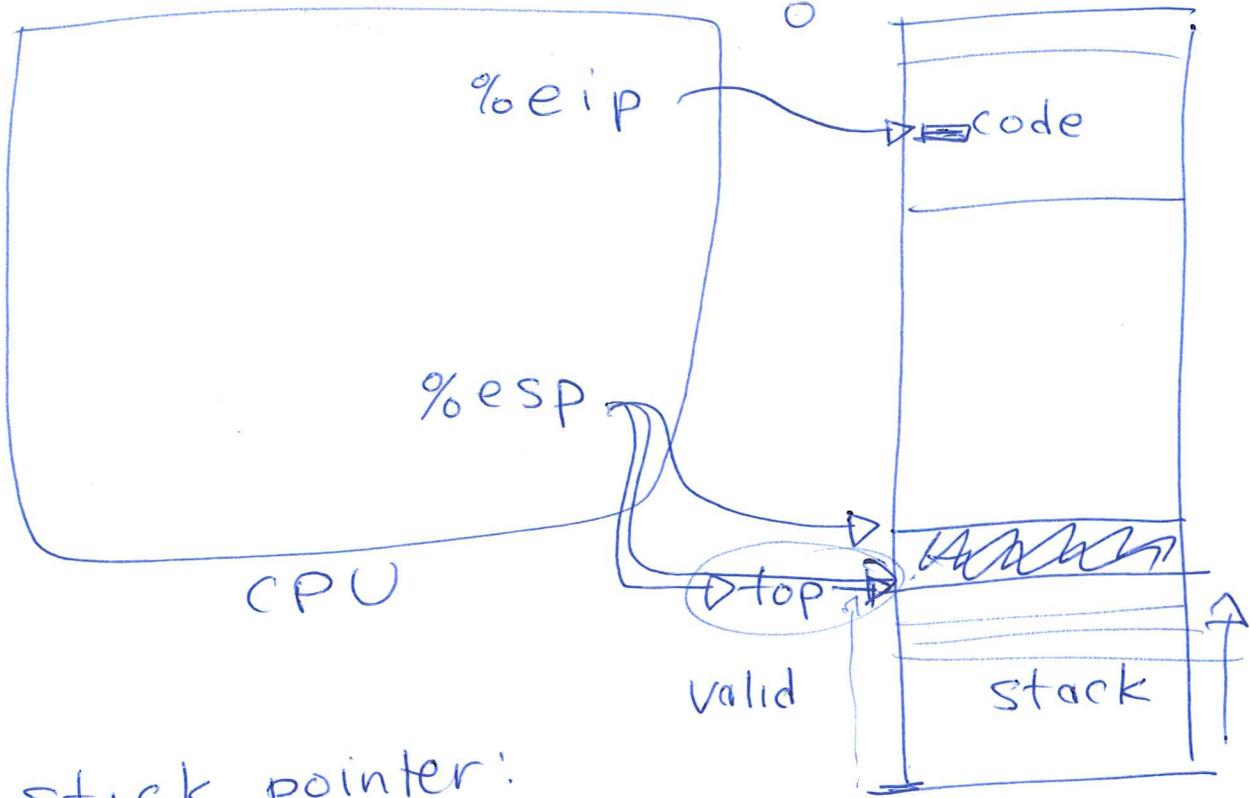
"pointer access"

movl , %reg
 what ends up
 in dest
 reg?

0x100
 264 256
 0 0x108

Stack :

Address Space



Stack pointer:

%esp

(another register)

can read/write

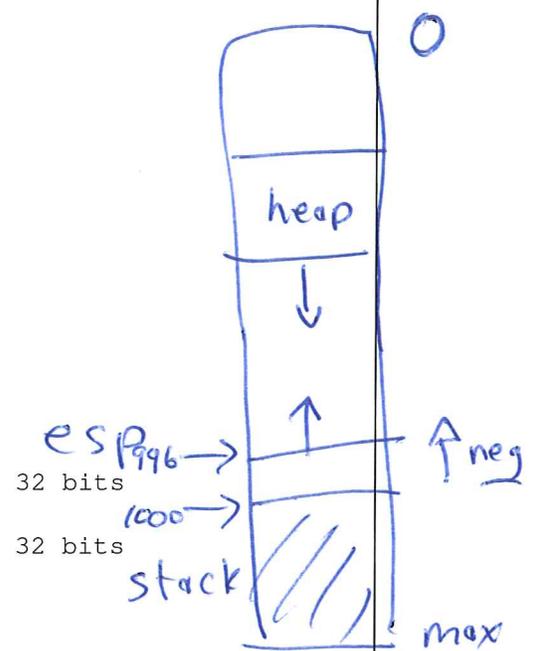
Stack:

local variables

New register to help with stack: esp (extended stack pointer)

Referred to as %esp

[.....]	eax	32 bits
[.....]	ax	16 bits
[.....]	ah	8 bits
[.....]	al	8 bits
[.....]	ebx	
[.....]	bx	
[.....]	bh	
[.....]	bl	
[.....]	ecx	
[.....]	cx	
[.....]	ch	
[.....]	cl	
[.....]	edx	
[.....]	dx	
[.....]	dh	
[.....]	dl	
[.....]	esi	
[.....]	edi	
[.....]	esp	
[.....]	eip	



Points to "top of stack" when program is running
Changes often (room for local variables, function call/return, etc.)

Can use normal instructions to interact with it, e.g., addl, subl
Can also use special instructions (we'll see this later)

Problem #5

Use instructions to:

- Increase size of stack by 4 bytes
- Store an integer value 10 into the top of the stack
- Retrieve that value and put it into %ecx
- Add 5 to it
- Put final value into %eax

```

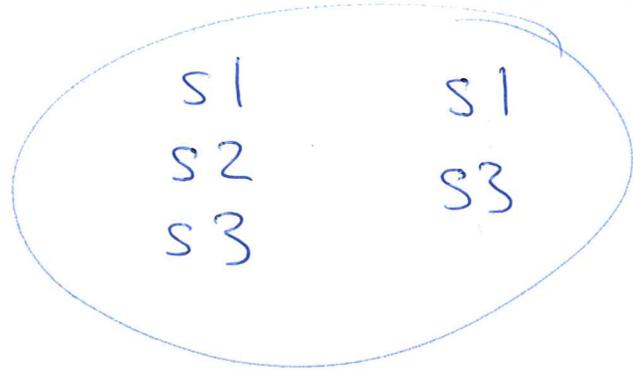
addl $-4, %esp
movl $10, *(%esp)
movl (%esp), %ecx
addl $5, %ecx
movl %ecx, %eax
addl $4, %esp
    
```

```

→ statement 1;
→ if ( expr ) {
    statement 2;
}
→ statement 3;

```

2 possible control flows: (13)

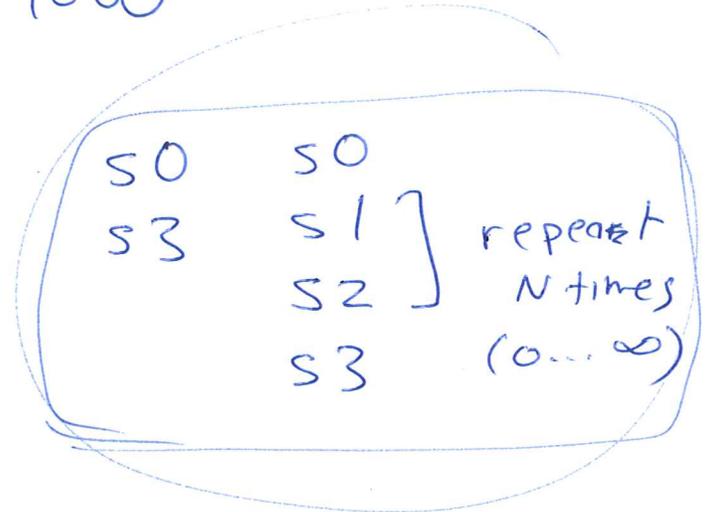


need assembly help
w/ control flow

```

stmt 0;
while ( expr ) {
    stmt 1;
    stmt 2;
}
stmt 3;

```



Needs:

→ Need to compute expression
(perform comparison)
if (a > 3) {

→ Need to change %eip
(control flow)

Instructions

⇒ Comparison

~~cmpb~~

cmpl b, a

internally:

computing (a-b)
(no destination)

what it does:

if (a-b) == 0

ZF=1

else

ZF=0

if (a-b) < 0

SF=1

else

SF=0

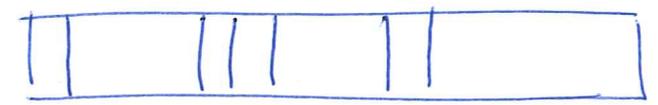
"state" on CPU:

%eip

%esp

gen purpose registers (%eax, etc.)

%eflags



0	6 7	11	
Carry Flag	Zero Flag	Sign Flag	Overflow Flag
(CF)	(ZF)	(SF)	(OF)
	<u> </u>	<u> </u>	

Condition codes: new bits in hidden %eflags register.

Some instructions set those bits based on comparisons:

cmp, test

Other instructions change control flow (%eip) based on results:

jmp family

INSTRUCTION: cmpl B, A

computes A-B (but doesn't put result anywhere)

condition codes (incomplete):

zero flag : ZF=1 if (A-B) == 0 otherwise ZF=0

signed flag : SF=1 if (A-B) < 0 otherwise SF=0

ZF
SF

INSTRUCTION: jmp TARGET always changes %eip to TARGET

INSTRUCTION: je TARGET %eip=TARGET if ZF==1

INSTRUCTION: jne TARGET %eip=TARGET if ZF==0

INSTRUCTION: jg TARGET %eip=TARGET if SF==0 and ZF==0

INSTRUCTION: jge TARGET %eip=TARGET if SF==0

INSTRUCTION: j1 TARGET %eip=TARGET if SF==1

INSTRUCTION: jle TARGET %eip=TARGET if SF==1 or ZF==1

⇒ (doesn't quite work in all cases)

Instruction

Idiom:

(15)

=> comparison

=> jump / branch

jump instruction:

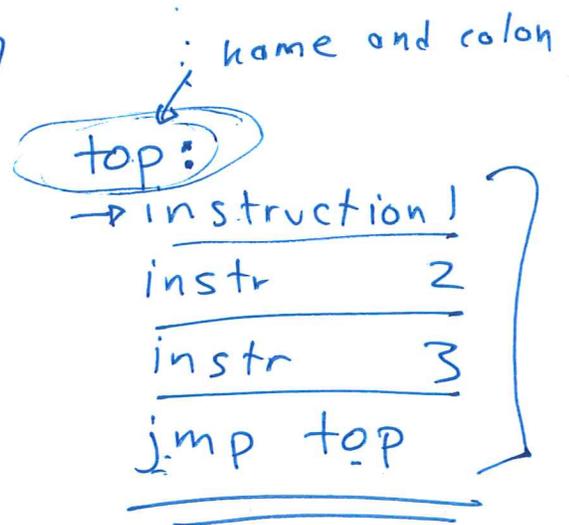
Change flow of control

many forms: [label]

jmp target

(unconditional branch)

[Changes %eip to target addr]

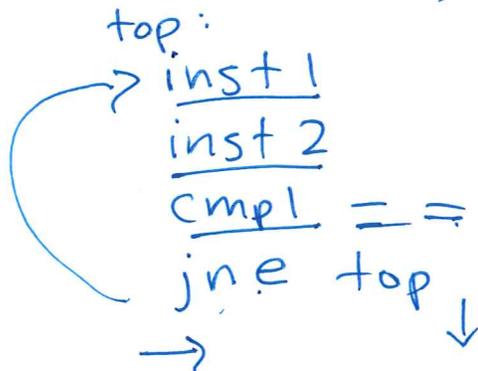


conditional: branch based on values of flags in %eflags

(condition codes)

jne target (jump not equal)

CF, ZF, SF, OF



otherwise "fall through" (go to next inst)

je

jg jge

jl jle

HW3 :

TAs

extra office hours



Test scripts:

⇒ try to run them

HW2 (regrade)

hw2-regrade/

✓ (expect
mail)

put slightly mod'd code
here

+ get regrade

```
struct list_t {
    node_t * array;
    int num_elements;
}
```

```
void list_init (list_t *l, ...) {
    l -> num_elements = 0;
```

stack

```
node_t a [chunk_size];
```

```
l -> array = &a[0];
```

```
return;
```

```
}
```

type check ✓

I'm C
looks
good!

⇒ C is fun

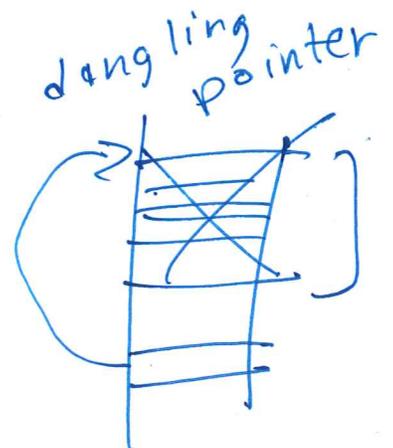
so close!

called:

```
list_t L;
```

```
list_init (&L, ...);
```

(OK)



```

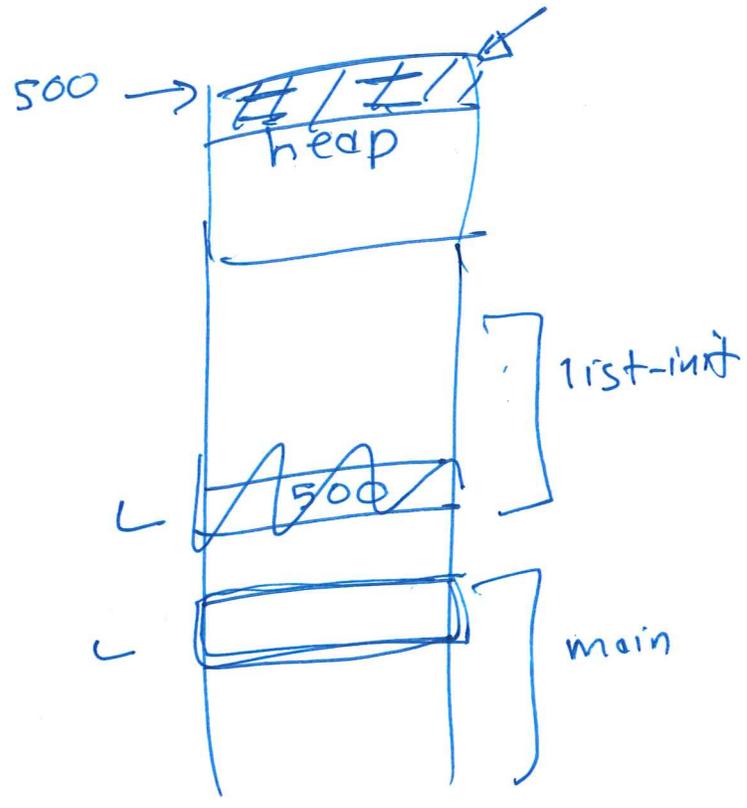
list-init
void (List_t *L, ...) {
    L = malloc(sizeof(List_t));
    L -> size = 0;
    return;
}

```

crash?
undefined
? }

(not undefined)

memory
leak



Bug #1:

struct list-t *L;

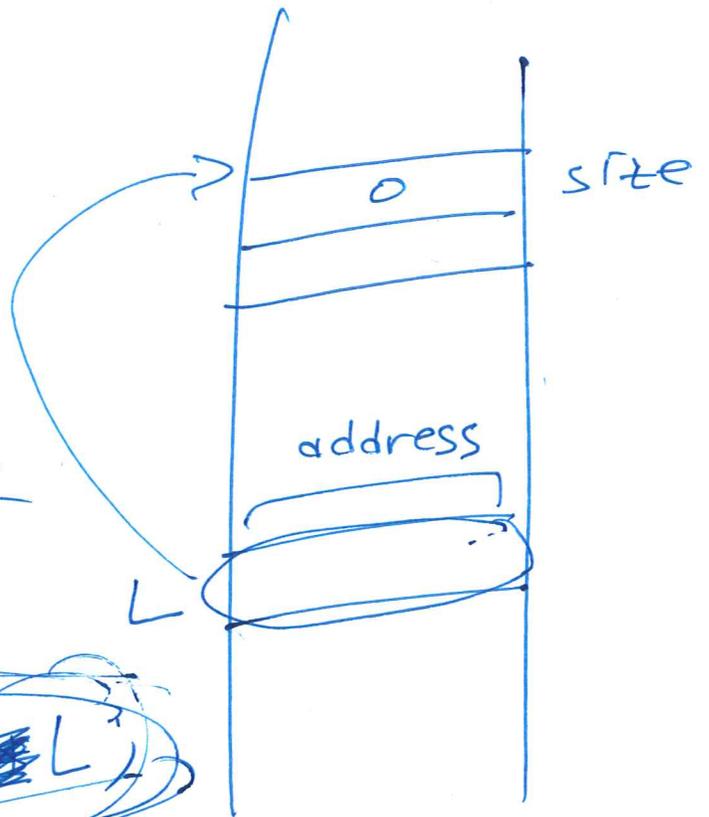
= malloc ()
*

L -> size = 0;

dereference

(*L).size

int *P;
int x = *P;



Bug #2:

struct list-t ~~L~~;

list-init (~~L~~ chunk-size);

~~L~~

X

void list-init (L -> size = 0;) }