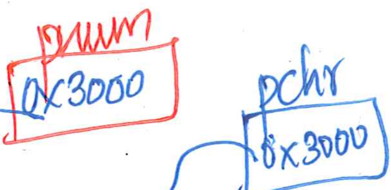


5. More low-level programming! How low are we going to go?!

Assume a **little-endian** machine in which the size of a **char** and **short int** are **1 byte** and **2 bytes** respectively. An **int** and a **pointer** take **4** bytes of storage. Consider the following code snippet. Assume that the variable **num** is allocated starting from memory address **0x3000**.

```
int num = 0x0FF1CE;
int *pnum = &num;
char *pchr = (char *)pnum;
```

(a) What are the values of the following expressions? In other words what is the value that will be printed if these expressions are used in a print statement. e.g., `printf("0x%x", *pnum);` Write all values in hexadecimal. [7 points]

0x00 0F F1CE

Expression	Value
`*pnum`	0x FF1CE
`(short)num`	0x F1CE
`pchr[1]`	0x F1
`(char)num`	0x CE
`++(*pchr)`	0x CF
`++pchr`	0x 3001
`++pnum`	0x 3004

→ *(pchr+1)

(b) The function `is_str_longer()` compares the length of two strings s and t and should return:

i. 1 (one), if strlen(s) > strlen(t)
ii. 0 (zero), otherwise

The function signature of strlen is: `unsigned int strlen(char *s);` Will the function `is_str_longer()` work as expected? If yes, just write "WORKS". If not, mention the issue and provide a fix for the same. You may assume that the pointers s and t point to valid strings and are NOT NULL.

```
int is_str_longer(char *s, char *t) {
    if (strlen(s) - strlen(t) > 0)
        return 1;
    else
        return 0;
}
```

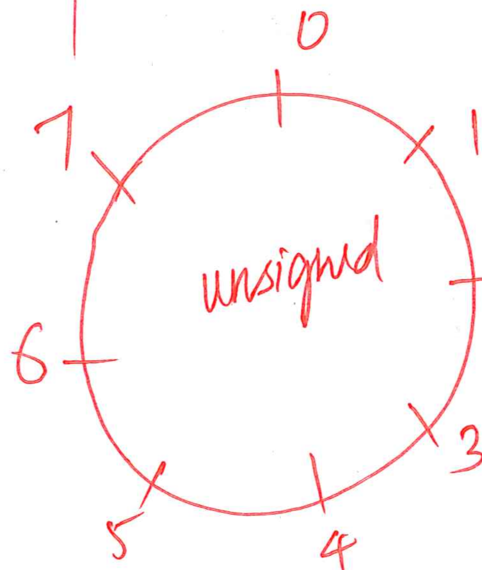
ANSWER:

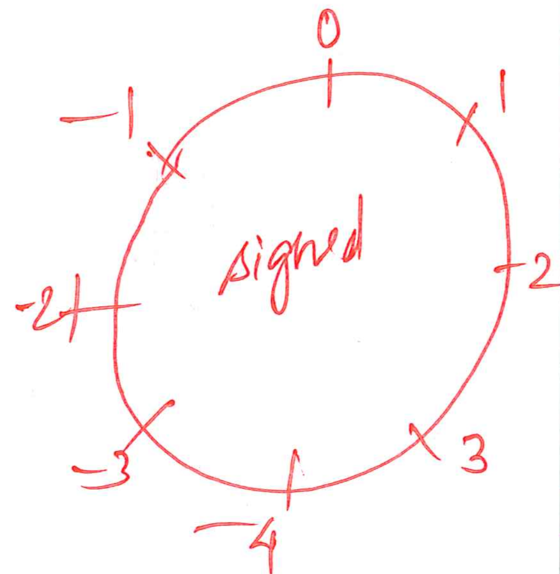
CS 354 - Lecture 6

Review

int → 3 bits

4	2	1	unsigned	signed. (2's complement)
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	4	−4
1	0	1	5	−3
1	1	0	6	−2
1	1	1	7	−1





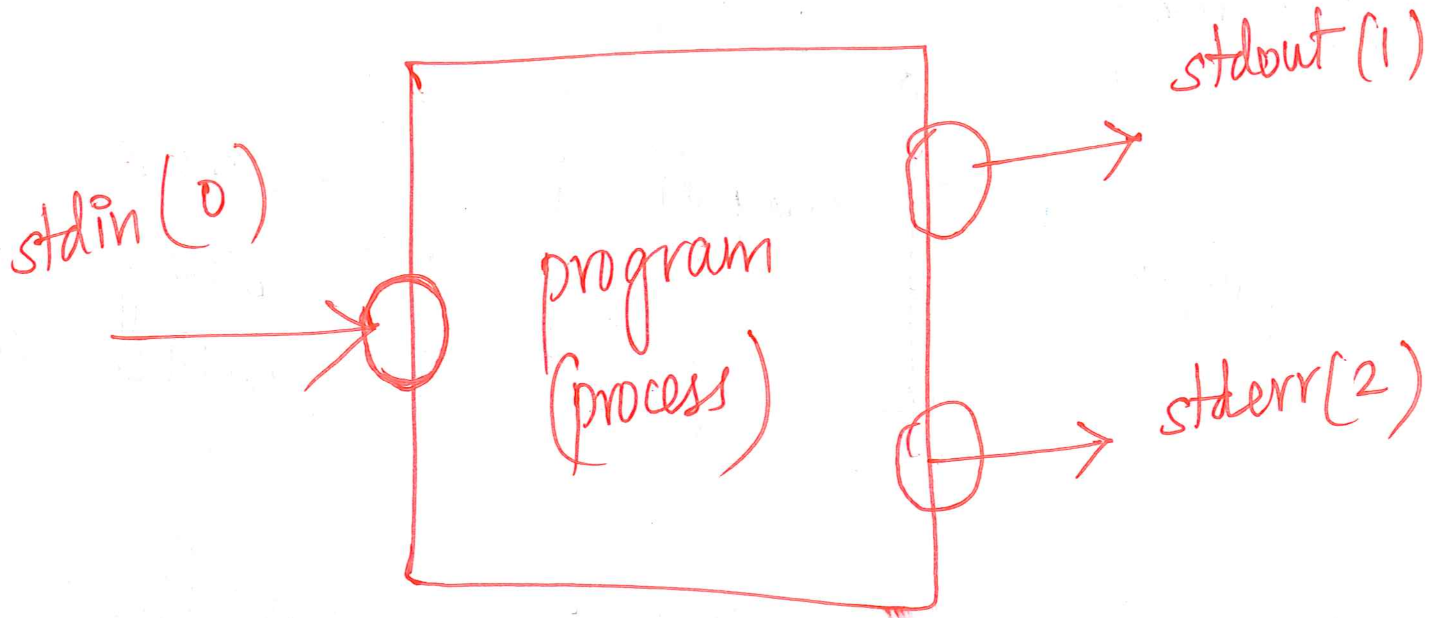
②

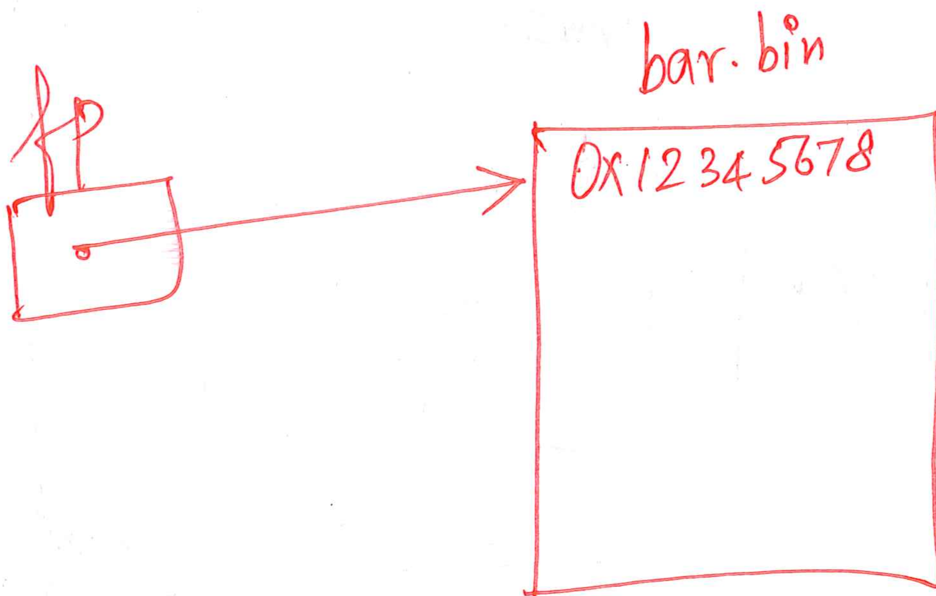
unsigned	signed
1 : 001	2 : 010 (0 ② → 10)
+ 2 : 010	− 1 : 001
3 : 011 ✓	1 : 001
carry ① 11 → lost.	① 2 1 0 2 2
5 : 101	2 : 010
+ 3 : 011	− 3 : 011
0 ~~8~~ : 000	−1 : 111
	0 0 2 2
	−2 : 110 (−2 −(3))
	−3 : 011
	~~3~~ : 011
	+3. 1

1 + 1 = 10

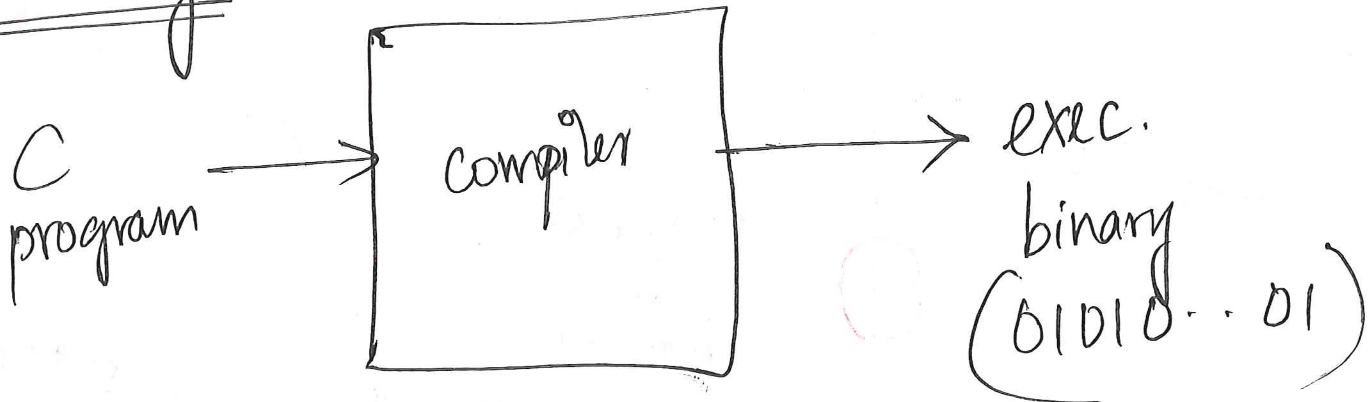
$t = a + b$

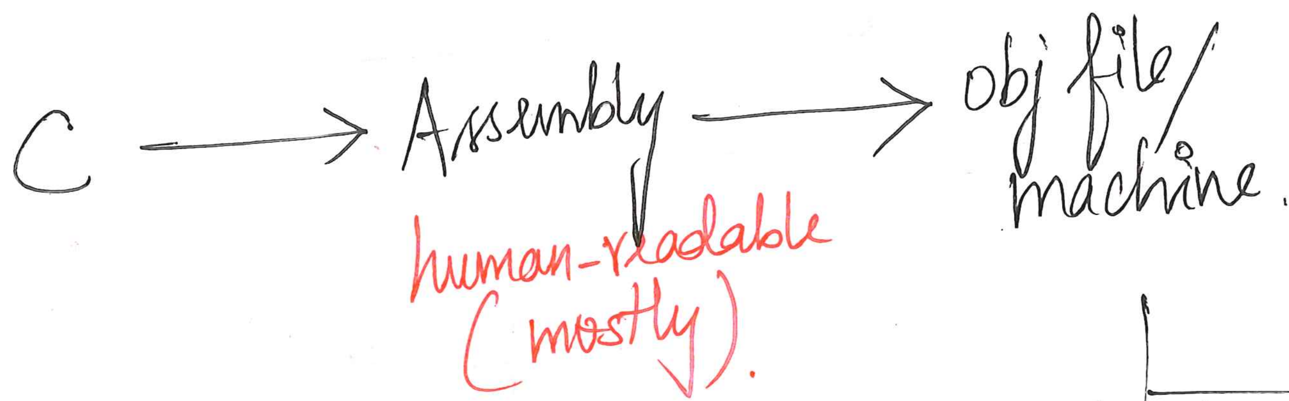
③





Assembly





sum = x + y;

instr1

2

3 add r_1, r_2

4

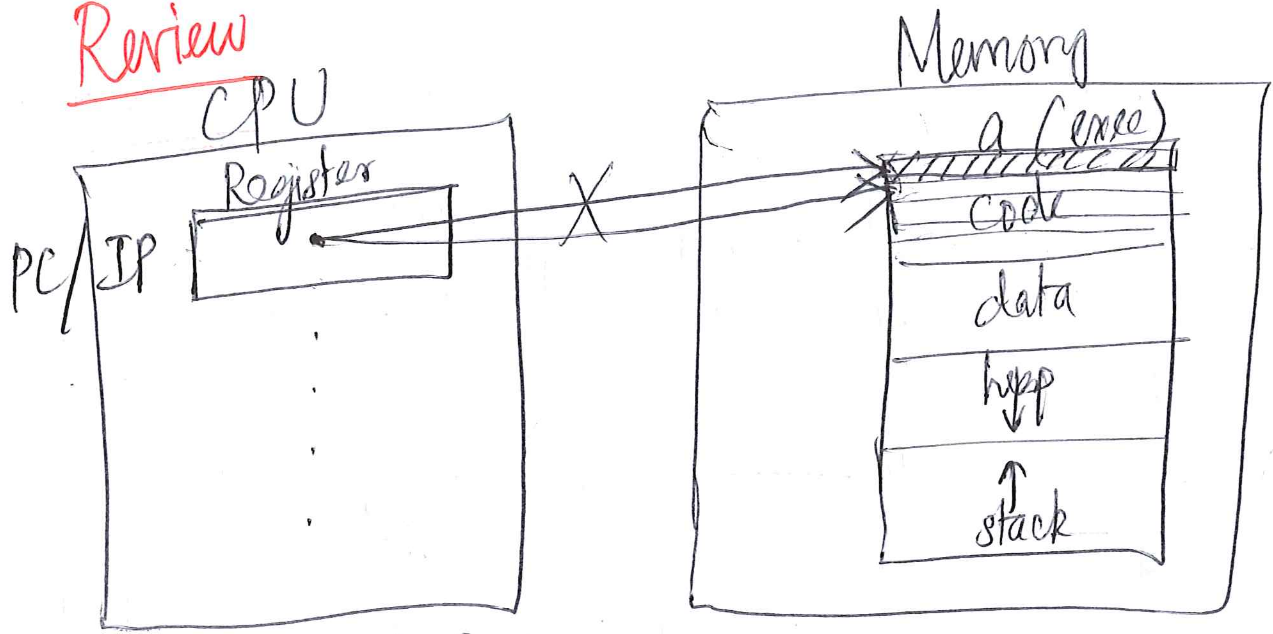
5

x: 10

y: 20

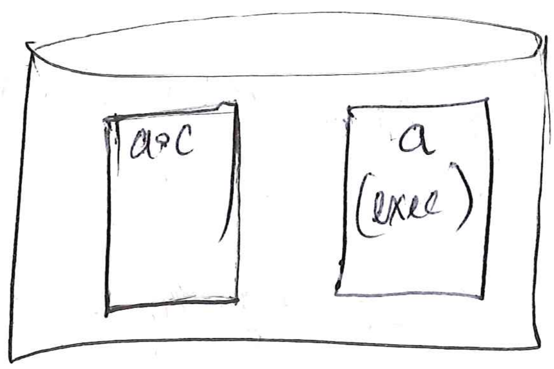
sum: 30

Review



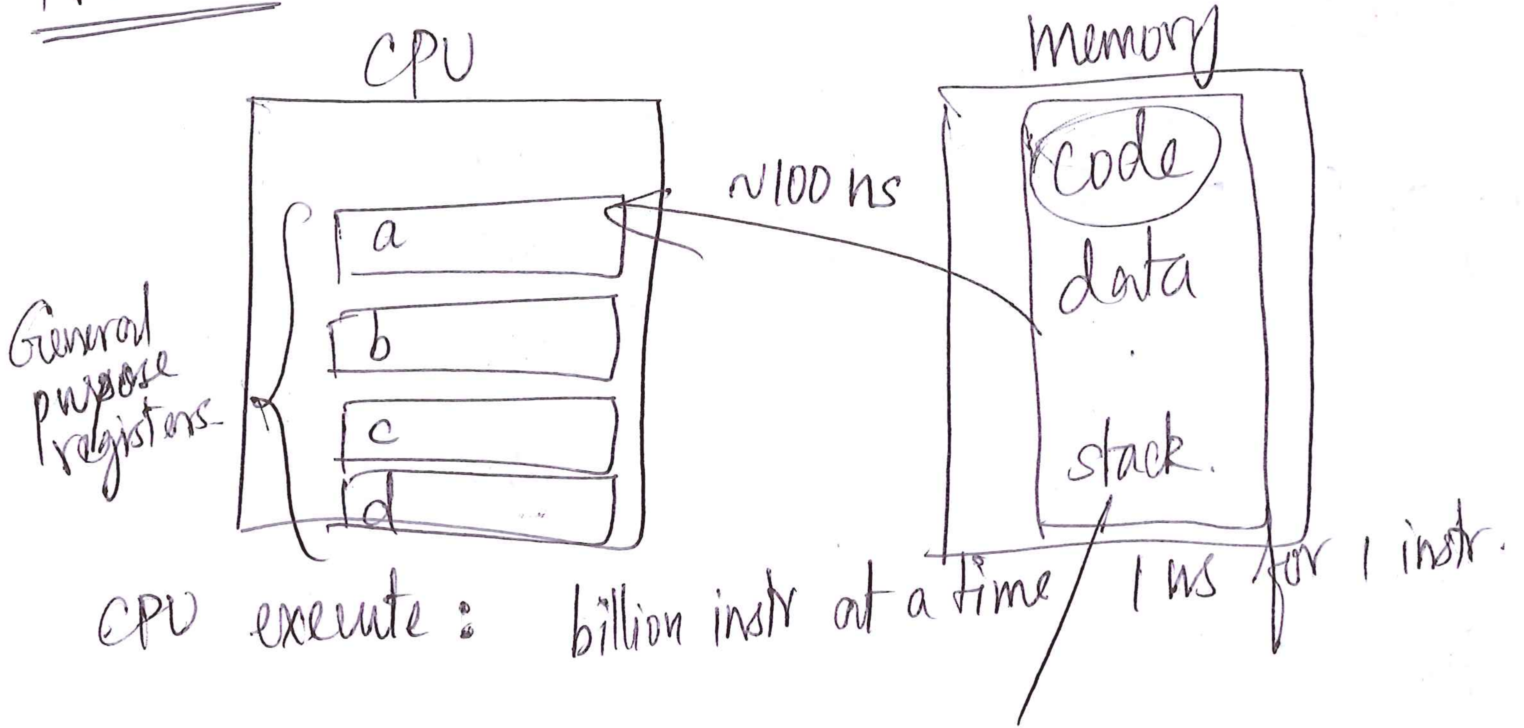
How CPU works?

1. fetch
2. decode
3. execute



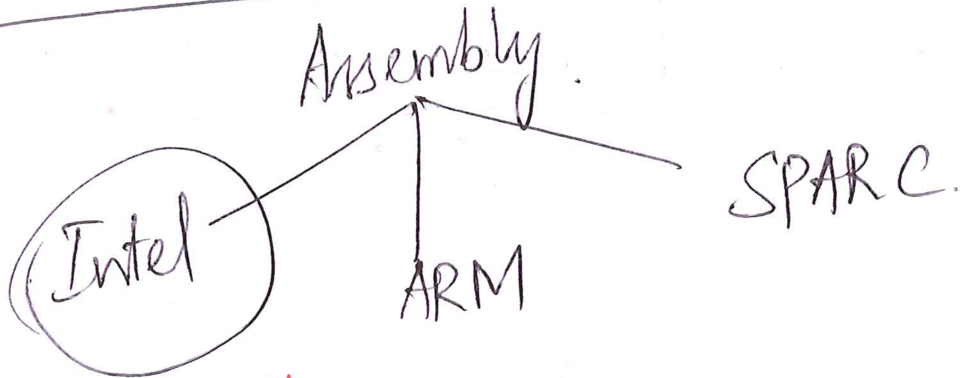
add r1, r2 ⟹ 07 10 11

Problem:



CPU execute: billion instr at a time / 1 ns for 1 instr.

Solution: general purpose registers



X86 Assembly.

8086, 80186, 80286, Pentium.

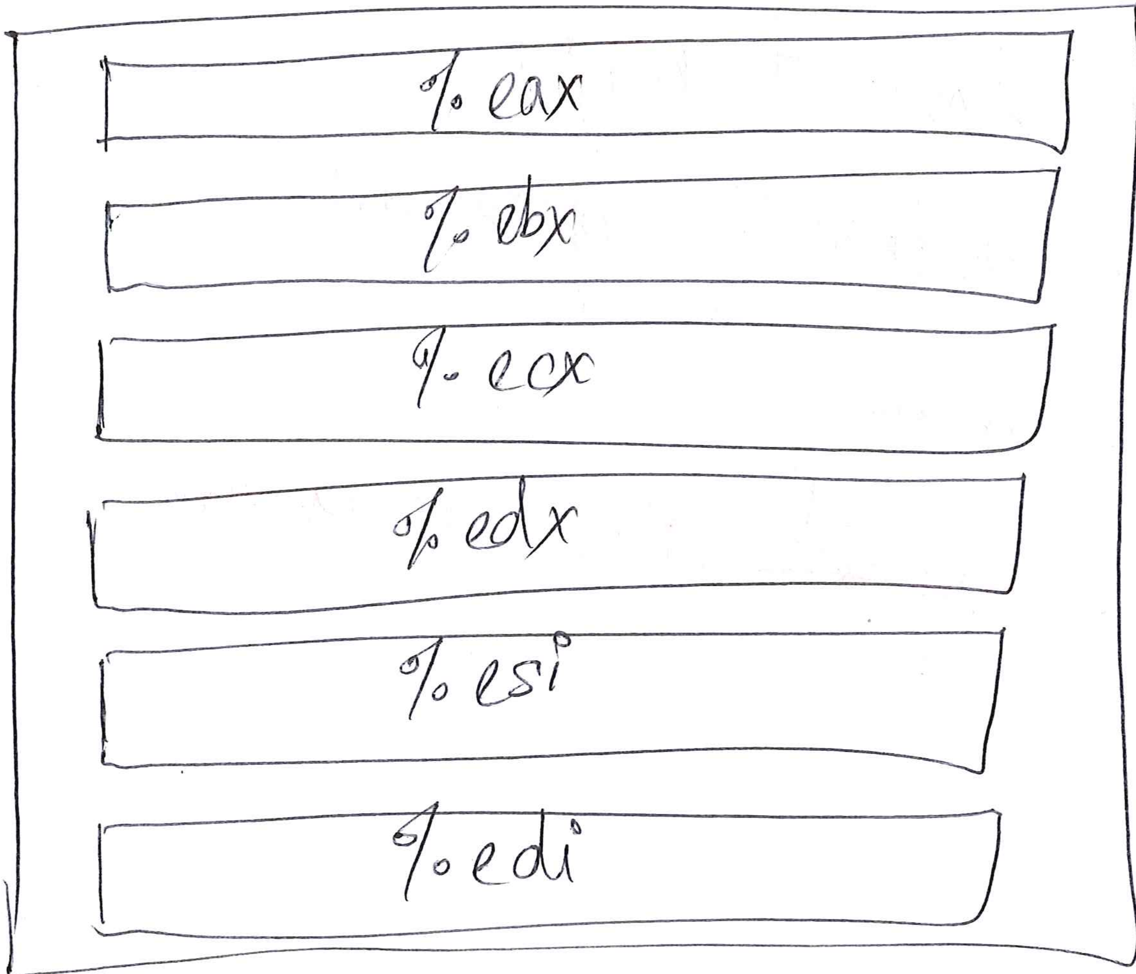
X86

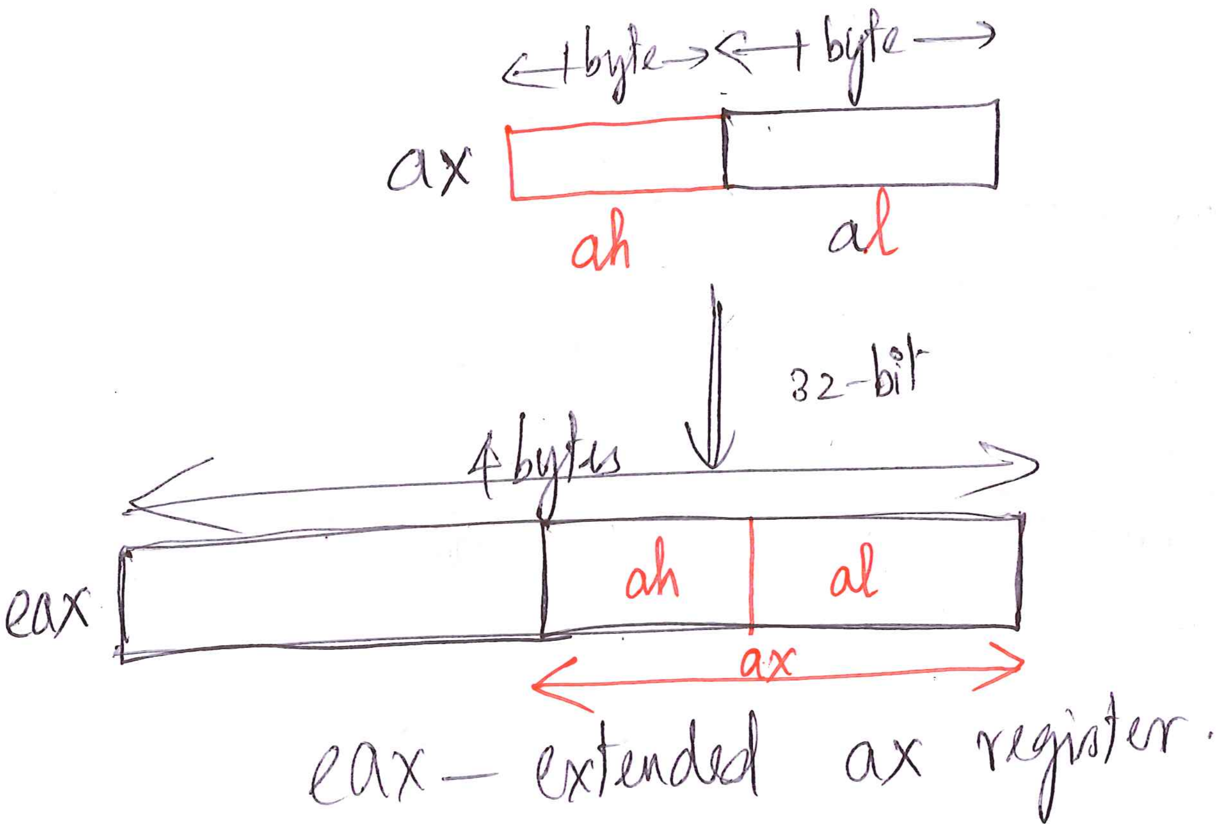
X86 (7)

AT & T (Unix) ✓	Intel — MS-DOS, Windows ✗
add A, B (A = source, B = dest.)	add A, B (A = dest, B = src.)
B ← B + A	A ⇐ A + B

CPU

32-bit





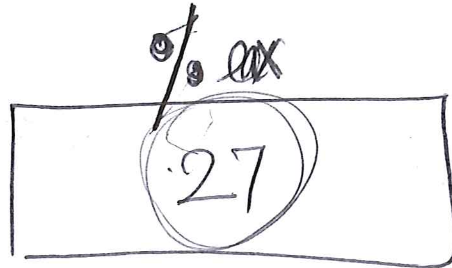
eax – extended ax register.

Instructions

1. data transfer.
2. arithmetic operations
3. memory.
4. ~~logical~~ bitwise operations.

Data transfer

mov source , destination.



mov $27 , %eax

↓ immediate operand.

Variants

movl $30, %ebx	l - long word (4 bytes)
movw $30, %bx	w - word (2 bytes)
movb $30, %al.	b - byte (1 byte).

32-bit machine: word size
32 bits (4 bytes)

movl $0XFF, %eax.

Arithmetic instruction

1. addl s, d

   $d \leftarrow d + s$

2. subl s, d

   $d \leftarrow d - s$

3. imull s, d

   int

   $d \leftarrow d * s$

   alt: imull aux, s, d

   $d \leftarrow aux * s$

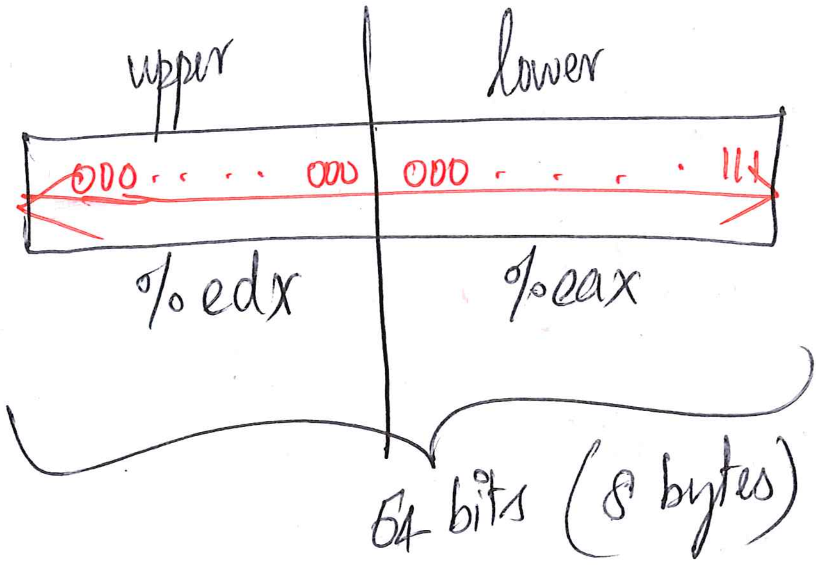
4. idivl operand

operand → divisor

7 / 2

7 → dividend

2 → divisor



quotient → %eax

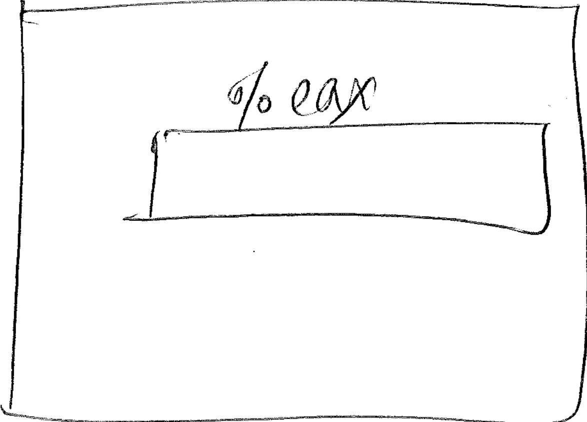
reminder → %edx

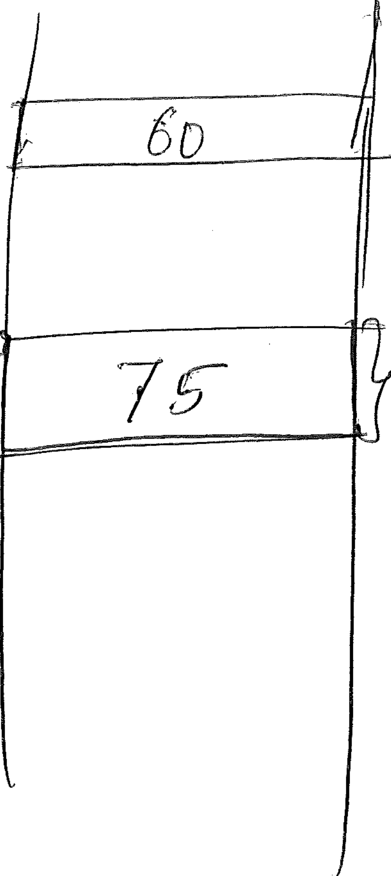
%ecx

2

idivl %ecx

Memory



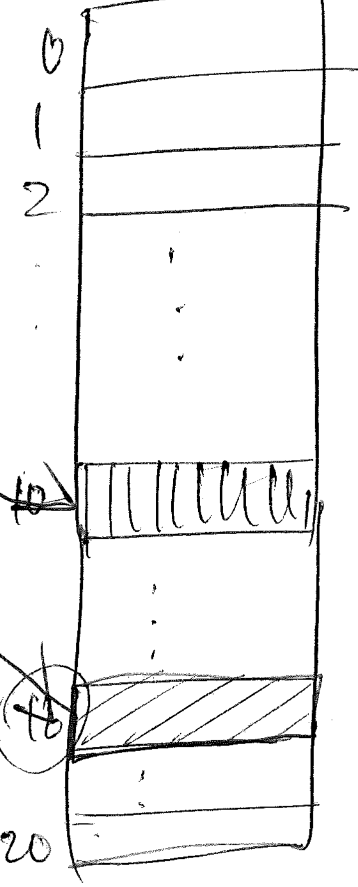


1. Absolute address

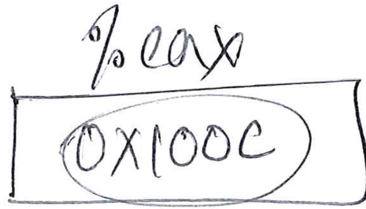
movl 0x1000, %eax

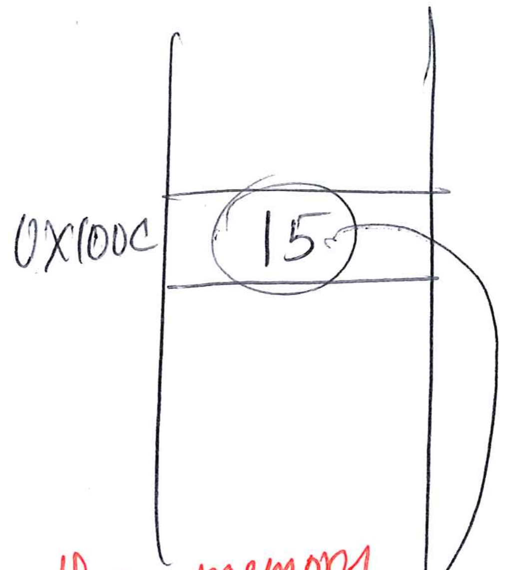
movl 1000, %eax

movl 0x10, %ecx
movl 10, %ecx



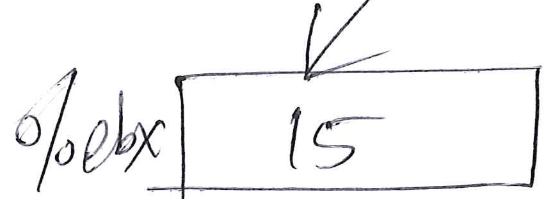
2. Indirect





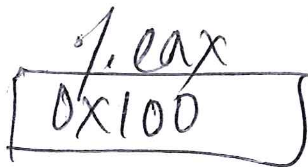
movl (%eax), %ebx

move the contents of the memory location stored in %eax to %ebx.

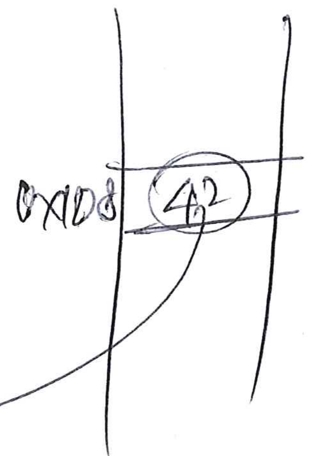


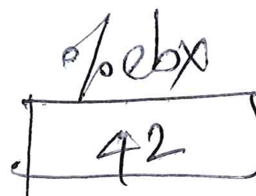
3. Base + disp increment

movl 8(%eax), %ebx



0X100 + 8 = 0X108 (addr)





4. Indexed

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

addr: 4 + contents of %eax + contents of %ebx

%eax: 0x700 %ebx: 0x100

addr: 4 + 0x700 + 0x100 = 0x804

5. Most general (Scaled indexing)

movl 0x10(%eax, %ebx, 4), %ecx

addr: 0x10 + contents of %eax + (contents of %ebx) * 4

%eax: 0x70 %ebx: 0x1

addr: 0x10 + 0x70 + (0x1 * 4)

= 0x10 + 0x70 + 0x4 = 0x84

(15)

movl d (R_1 , R_2 , S)

---

addr: $d + R_1 + R_2 * S$

movl 3 (, %ebx, 4)

movl (%eax, %ebx, 8)

---

bitwise

andl S, D

D ← D & S

orl S, D

D ← D | S

xorl S, D

D ← D ^ S

notl D

D ← ~D

CS 354.

Assembly Worksheet -1

Prepared by: Remzi Arpaci-Dusseau

```
                  x86 general-purpose registers

(most significant)                        (least)
      [........ ........ ........ ........] 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

Referred to as %eax, %ebx, %ecx, %edx, %esi, %edi, etc.
```

```
INSTRUCTION: mov SOURCE, DESTINATION

  definition: moves "SOURCE" into "DESTINATION"

  commonly has trailing character that indicates size of move, e.g.,
    movb - move a byte        movw - move 2 bytes.
    movl - move "long" or 4 bytes (that's an L after mov, not a one)
    movq - quad or 8 bytes

  our focus: movl (mostly)

  Initial (limited) usage
  - source=number ("immediate")   destination=register
    e.g., mov $10, %eax

  - source=register               destination=register
    e.g., mov %eax, %ebx

  Later, we will add different types of operands for mov
```

```
INSTRUCTION: addl SOURCE, DESTINATION

  definition: adds SOURCE and DESTINATION, puts result into DESTINATION
              i.e., DESTINATION = DESTINATION + SOURCE

  limited usage (for now):
  - source=number ("immediate")    destination=register
  - source=register                destination=register
```

```
INSTRUCTION: subl SOURCE, DESTINATION

  definition: DESTINATION = DESTINATION - SOURCE

  limited usage (for now):
  - source=number ("immediate")    destination=register
  - source=register                destination=register
```

```
INSTRUCTION: imull SOURCE, DESTINATION

  definition: DESTINATION = DESTINATION * SOURCE

  alternate:
    imull AUX, SOURCE, DESTINATION
    definition: DESTINATION = AUX * SOURCE

  limited usage (for now):
  - source=number ("immediate")    destination=register
  - source=register                destination=register
  - (aux=immediate)
```

```
INSTRUCTION: idivl DIVISOR

  definition: contents of %edx:%eax (64 bit number) divided by DIVISOR
    quotient  -> %eax
    remainder -> %edx

  limited usage (for now):
  - divisor=register

  Notes: A bit weird in its usage of VERY SPECIFIC registers!
```

src dest

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
```

Problem #2

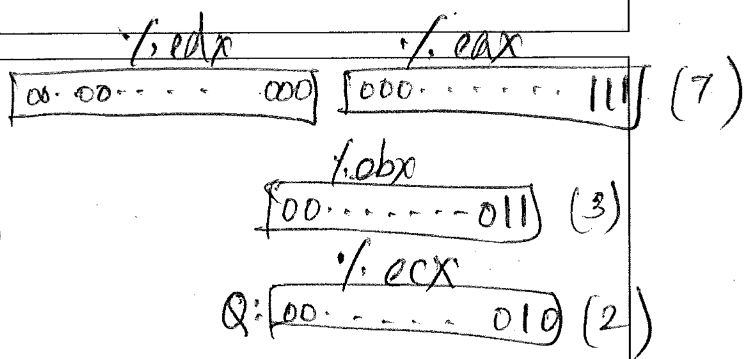
Expression: 3 + 6 * 2

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

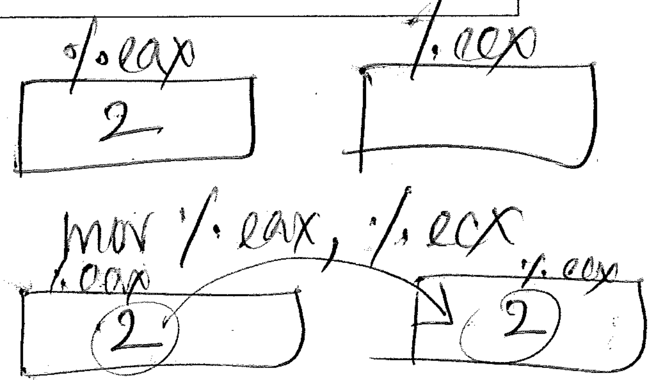


Write simple C expression that is equivalent to these instructions

7/3 + 9%2

%eax = 2 + 1 = 3

mov | copy



Many x86 instructions can refer to **memory addresses;** these addresses take on many different forms.

ABSOLUTE/DIRECT addressing

definition: just use a number as an address

```
movl 1000, %eax
  gets contents (4 bytes) of memory at address 1000, puts into %eax

NOTE: DIFFERENT than movl $1000, %eax
(which just moves the VALUE 1000 into %eax)
```

INDIRECT addressing

definition: address is in register

```
movl (%eax), %ebx
  treat contents of %eax as address, get contents from that address,
  put into %ebx
```

BASE + DISPLACEMENT addressing

definition: address in register PLUS displacement value (an offset)

```
movl 8(%eax), %ebx
  address = 8 + contents of eax
  get contents from that address, put into %ebx
```

INDEXED addressing

definition: use one register as base, other as index

```
movl 4(%eax, %ecx), %ebx
  address = 4 + contents[eax] + contents[ecx]
  get contents from that address, put into %ebx
```

SCALED INDEXED addressing (most general form)

definition: use one register as base, other as index, scale index by constant (e.g., 1, 2, 4, 8)

```
movl 4(%eax, %ecx, 8), %ebx
  address = 4 + contents[eax] + 8*contents[ecx]
  get contents from that address, put into %ebx
```

Problem #4 (from CSAPP 3.1)

Memory

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

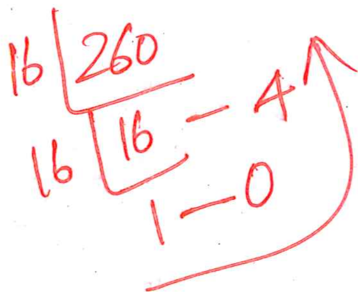
movl ____, %eax

Registers

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Value of:

Operand	Value	Notes
%eax	0x100	
0x104	0XAB	movl 0x104, ~~%eax~~ R
$0x108	0X108	
(%eax)	0XFF	movl (%eax), R
4(%eax)	0XAB	4 + 0x100 = 0x104 (addr)
9(%eax, %edx)	0x11	9 + 0x100 + 0x3 = 0x10C (addr)
260(%ecx, %edx)	0x13	260 + 0x1 + 0x3 = 0x108; 260 = 0x104
0xFC(,%ecx, 4)	0XFF	0xFC + (0x1 * 4) = 0x100
(%eax, %edx, 4)	0X11	0x100 + (0x3 * 4) = 0x10C; 0x3 * 4 = 0xC



$260_{10} = 0x104$

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        32 bits

[........ ........ ........ ........] eip        32 bits
```

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
- Decrease size of stack by 4 bytes.

Edited by: Gerald.