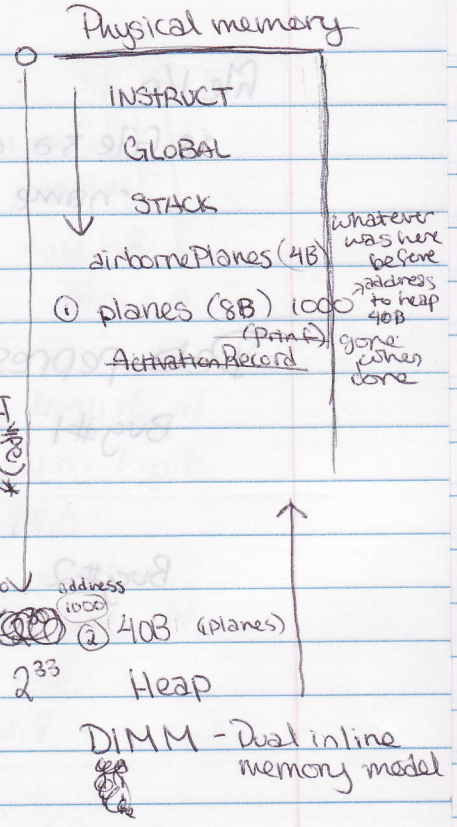
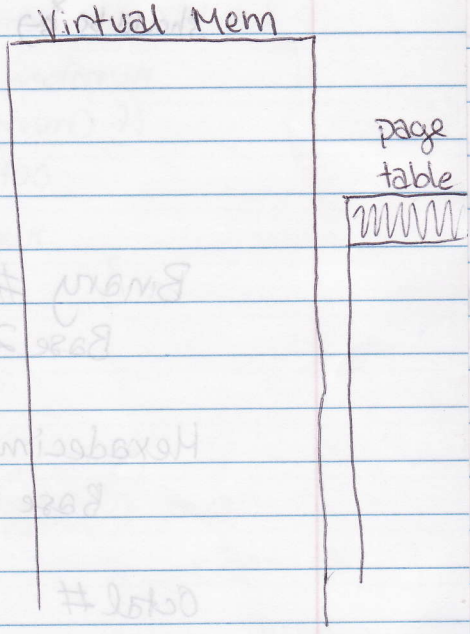
Nov 4 Midterm

```
int airbornePlanes;
struct FlightType *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes = (struct FlightType *) malloc(sizeOf(struct FlightT
                                      airbornePlanes));

if (planes == null) {
    printf("Error in allocating the data array.\n");
    ...
}
planes[0].altitude = ...
...
free(planes);
```

"Breaks link ① & ②" OS ignores address to heap "frees" pointer





malloc calls = free call

may

Command line

System Calls

The OS extends the functionality of the underlying hardware

- OS functionalities exported as a set of system calls
- In C system calls are "wrapped by" C functions (looks like C function calls)
- System calls are described in sec 2 of online manual ex man 2 open

file I/O

- a file is a contiguous set of bytes
  - name / can create, remove, read, write / append.

Data representation

Bug #1

```
scanf("%d", val);
```

Bug #2 no initializing of y

```
/* return y = Ax */
int * matvec(int **A, int *x) {
    int *y = malloc(N * sizeof(int));   type casting
    int i, j;
```

Bug #3

```
int **p;
p = malloc(N * sizeof(int));      wrong mem allocated
for (i=0; i<N; i++) {
    p[i] = malloc(M * sizeof(int));
}
```

should be *p

Binary #

Base 2 #'s are 0 or 1 $\sum_{i=0}^{n} d_i \times 2^i$

Hexadecimal #

Base 16 0-9 A-F 10A 12C 14E 11B 13D 15F $\sum_{i=0}^{n} d_i \times 16^i$

Octal #

Base 8 0-7 $\sum_{i=0}^{n} d_i \times 8^i$

0x 1003 B or 0h 1003B

hexadecimal hexadecimal

1003B

16^4 16^3 $\times 16^2$ $\times 16^1$ $\times 16^0$

16^4 : : 16 1

x3 x11

0 0 48 11

$16^4 + 0 + 0 + 3(16^2) + 11(16^0) = 16^4 + 59$

18 => 0x12

hexa

16^1 : 2

$1(16^1) + 2(16^0)$

27 16 + 11 + 0

hex → binary

0x1B → 8 + 2 + 1 1011

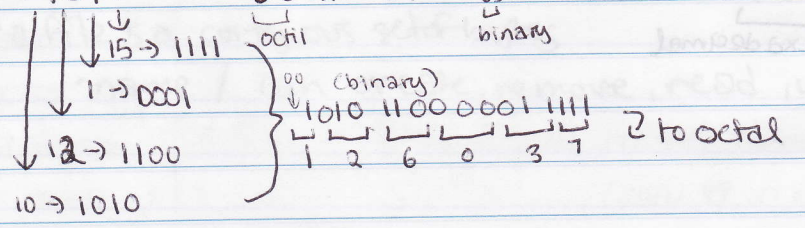
16 + 8 + 2 + 1 + 0

1 8

10001 1011

11011

0xACIF = 0o126037 = 0b1010110000011111



in decimal, digits to the right of radix point have value $1/10^i$ for each digit in the i^{th} place

- 0.25 is $2/10 + 5/100$

Similarly, in binary, digits to the right of radix point have value $1/2^i$ for each i^{th} place

- just base is different
- 8.625 is 1000.

Number = decimal Fraction

```
while (number > 0) {
    number = number * 2
    if (number >= 1) {
        output 1;
        number = number - 1;
    }
```

C declaration

- char
- short int
- int
- pointer (only one that changes # bytes needed from 32→64 bit machine)
- float
- double

Big Endian vs Little Endian

A0	BC	00	12

(BE) A0BC0012 or 1200BCA0 (LE)

- Big (BE) Endian - Most significant byte first
- Little (LE) Endian - Least significat byte first

(no diff in comp arch)

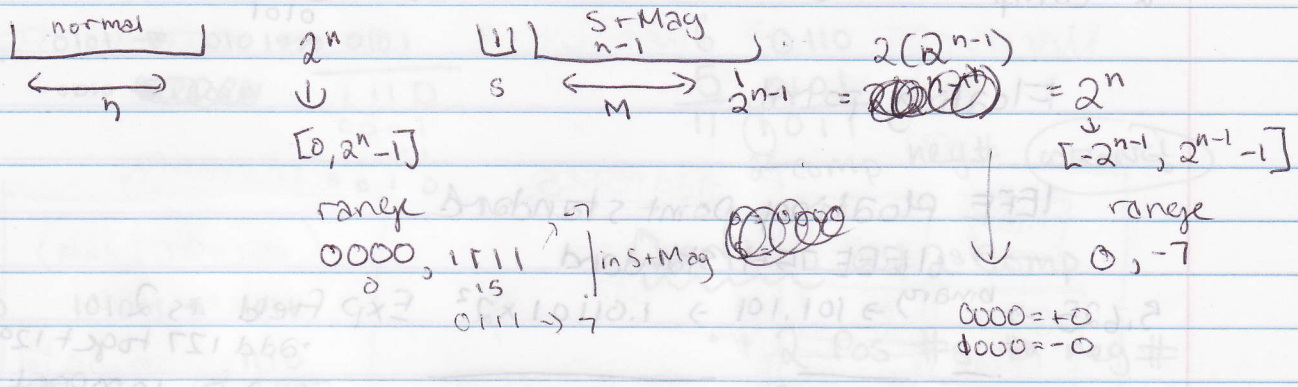
- Interprets code & values
- 1 comp (BE) sends info to another comp (LE)
- Convert into standard form before transmitting

Signed Magnitude

0+ 0100 = 4 1100 = -4

1- Sign Magnitude Sign Magnitude

0011 = 3 1011 = -3



One's Complement - represent negative numbers by complementing positive numbers

110 = 6

↳ in One's Complement 001 = -6

1110	0110
normal - 14	6
S+Mag = -6	6
1's Comp = -1	6

0001 "flip each bit" b/c 1st bit is 0

0 - pos
0 - don't need to flip

Two's Complement

- 1st Comp plus 1
- most sig bit gives sign
- copy all '0' bits from LSB (start sig) till first '1' bit. Copy '1' bits, flip remaining till MSB

Advantages: only 1 zero
convenient for arithmetic computations.
used on almost all comps

@.@ What..?

1110 = -2
0001 = -1
0010

111 carryover
0001
+1
0010

0110
+6

	0011	1011
NORMAL	3	11
S+Mag	+3	-3
1st Comp	+3	-4
2st Comp	+3	-5

0100
0100
+ 1
0101

Floating point

IEEE floating point standard

- IEEE 754 standard

5.625 → binary → 101.101 → 1.01101×2^2

Exp field is 2
- add 127 to get 129

exp is 10000001
Mantissa 01101 0000 (add 0's)
Sign bit is 0

S	Exp	Mantissa

-5 1101 → Sign + Mag

0101 → 1's comp

1010

1001 NORM 9 0001 1

SMag -1 1

1's Comp -6 1

2's Comp -7 1

7 → 0111 -7 → 0111 → 1000 +1 → 1001

Norm +2s comp

IEEE Floating point standard

Most comp follow IEEE 754 standard

Single precision (32 bits)

Double precision (64 bits)

Extended precision (80 bits)

10 → 2 in binary

$10.005 \rightarrow 1.0005 \times 10^{1} \rightarrow 1.a \times 10^{b} \rightarrow (+)\ 1.m \times 2^{n}$

Sign, Mantissa

$= \frac{1}{2^1} + \frac{1}{2^3}$

5.625

101. 101 → $1.01101 \times 10^{2} \rightarrow 1.01101 \times 2^{2}$

mantissa

n+127

Sign | Exp | Mantissa

0 | 10000001 (129) | 01101000000000000000000

23 digits

-7 -0111 → 1000+1 → 1001

5 0101 → 0101

-2 -0010 → 1101 → 0001 +1 → 0010

6 0110

5 0101

11 1011

2's comp neg # (WRONG)

Overflow w/ 2's Comp

- + 2 Pos #s → neg #
- - 2 neg #s → pos #

-5 1010+1 → 1011

-2 1110

-7 X1001 → -7

0110 +1 = 0111

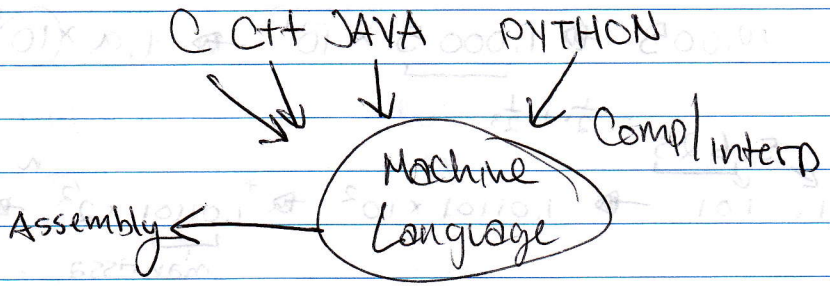
Sign Extension

Signed int	4-bit representation	8-bit representation	16-bit representation
+1	0001	00000001	0000000000000001
-1	1111	11111111	1111111111111111

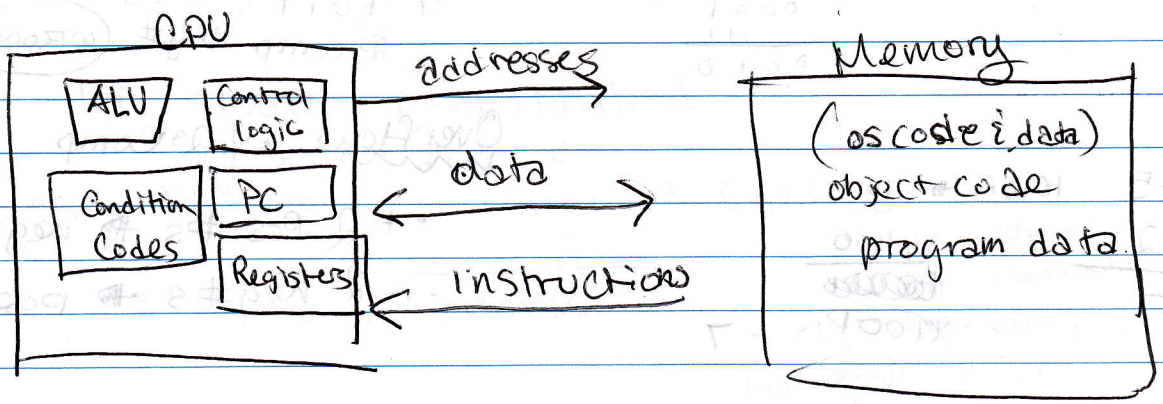
Assembly Programming

Why?
- machine interface: where software meets hardware
- to understand how hardware works, we have to understand the interface it exports

why not binary language?
- easier for humans to read & reason
- major differences
  - human readable language instead of binary
  - relative instead of absolute addresses.

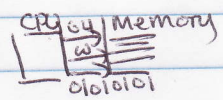


Macro ops but hardware uses microops



registers - store memory to be used in main memory

READ = LOAD
WRITE = STORE




```
10x  for(i=0; i<10; i++)
       a[i] = b[i] + c[i];
```

W R R+W

R = read
W = write

WR RR RR