# CS253 Architectures II

Lecture 5

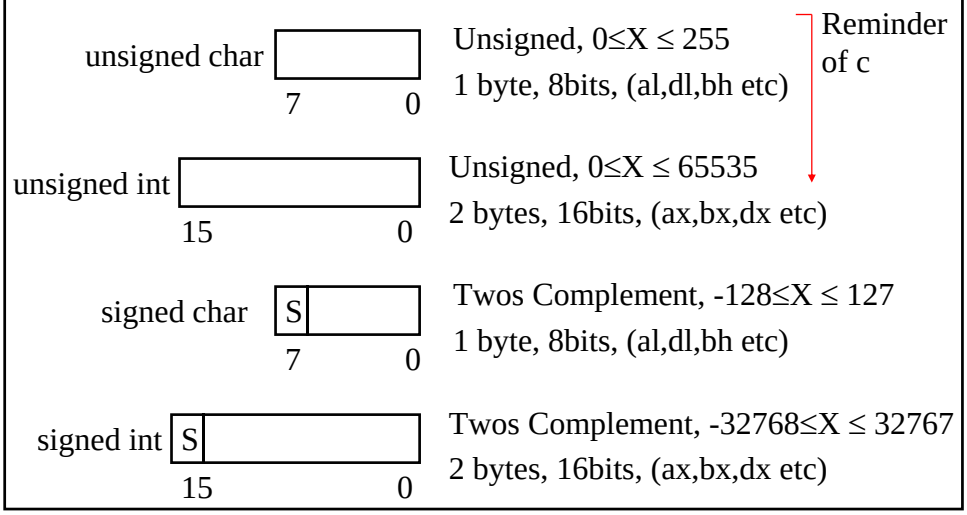
Assembly Language

(Data Types, Integers, Real and String)

Charles Markham

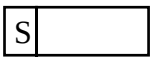
# Data format (unsigned integer) in PC's

So far we have used the data stored in the registers in the following way, the formats are very similar to that used in C.



# Data format (signed integer) in 8086

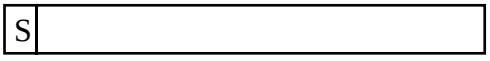
There are other data formats that instructions assume, the following formats are used by the 80386 and 80387 coprocessor.

Word 

Twos Complement, -32768<X<32768

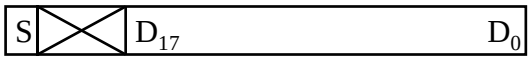
Short int 

Twos Complement, $-2x10^9<X<2x10^9$

Long int 

Twos Complement, $-9x10^{18}<X<9x10^{18}$

# Data format (integer) in PC's

Packed Decimal 


-99…99<X<99…99 (18digits)

Each byte stores two BCD digits, each digit requires 4 bits

# Twos Complement

You could use a single bit to set the sign of the number. However if you do this then you need separate hardware/software for addition and subtraction. Two's complement avoids the problem.

By inverting each bit in a number and adding one to the result. A new number is formed that has the same effect when added to a number as subtracting the original number. If the MSB is 1 then the number is negative, twos complement, otherwise positive ordinary binary.

# Twos Complement

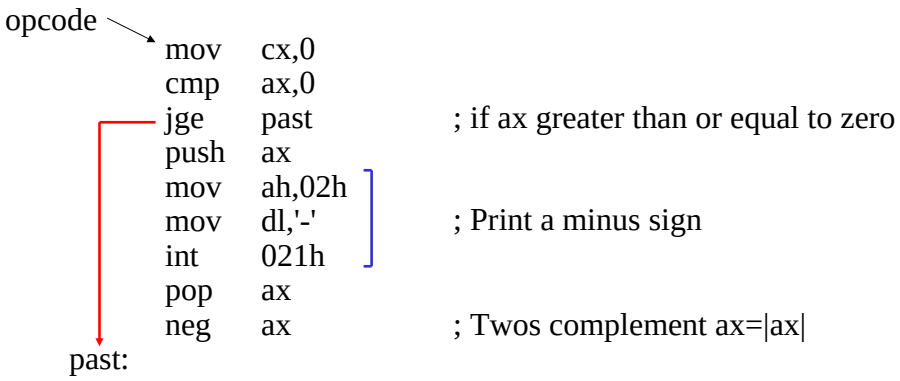
Subtract 65 from 120 using the twos complement method.

| | |
|---|---|
| 120: | 0 1 1 1 1 0 0 0 |
| 65: | 0 1 0 0 0 0 0 1 |
| Invert bits: | 1 0 1 1 1 1 1 0 |
| Add 1: | 1 0 1 1 1 1 1 1 |
| Add to 120 | 0 0 1 1 0 1 1 1 :55 |

128, 64, 32, 16, 8, 4, 2, 1

# Negative numbers

The following code segment converts negative numbers to their absolute (positive) value. Add it to the start of the print algorithm to print signed numbers, it assumes the number is stored in AX.




# Strings

A string is a group of ASCII characters used to describe text.

Two formats are used

String terminated with a NULL byte=0 character

*Charles Markham /0*

String using first byte to set length

*/15 Charles Markham*

ASCII: American standard code for information interchange.

# String operations

A string is a set of bytes stored is successive memory locations. The 8086 contains instructions for the manipulation of strings, moving them and searching for text.

Registers SI (Source) and DI(Destination) are used for string manipulation.

| | |
|---|---|
| LEA SI,*Label* | Load effective address in to SI |
| MOVSB | Move string byte |
| CMPSB | Compare string |

# movsb

Copies memory pointed to by DS:SI to the address specified in ES:DI

If DF=0 SI,DI are incremented,

If DF=1 SI,DI are decreased.

Note a modifier ‘REP’ can be placed in front of any string instruction and repeats the instruction CX times. Rep stands for repeat string prefix.

CISC!!!

# cmpsb

Compares memory byte pointed to by DS:SI to the memory byte pointed to by ES:DI. The command is normally used with a REPE modifier as part of a large string test loop.

REPE: Repeat while equal

RENE:Repeat while not equal

If DF=0 SI,DI are incremented after each comparison,

If DF=1 SI,DI are decreased after each comparison.

e.g. REPE CMPSB

JNE Strings are not equal

```
        .DATA

string1 db 'This is a test$'
string2 db 'abcdefghijklmnopqrstuvwxyz$'

        .CODE
        .STARTUP

        mov     dx, OFFSET string2
        mov     ah,09h
        int     021h

        mov     ax,ds           ; ES=DS
        mov     es,ax
        LEA     SI,string1      ; Offset
        LEA     DI,string2
        CLD                     ; Direction=0
        mov     CX,15           ; No. Bytes
        REP     MOVSB           ; Copy

        mov     dx, OFFSET string2
        mov     ah,09h
        int     021h
        .EXIT
```

Repeat instruction until CX=0, (modifier, vector?).

BIOS Print until $

Copy string

BIOS Print until $

Output: abcdefghijklmnopqrstuvwxyz This is a test

# Multiply

Many CISC processors include a multiply and divide function in the instruction set, many RISC processors don't. Can you write MASM code to multiply without using the MUL operation?

Decimal

```
  1234
     x
    12
  2468
+12340
 14808
```

Binary

```
  1101
     x
   101
  1101
+00000
+110100
 1000001
```

# Multiply (The Hard Way, RISC)

```
        .STARTUP

        mov ax,0
        mov dx,100      ;Multiply dl by bl result in ax.
        mov bx,123
        mov cx,8

back:   rcr dx,1        ;move lsb dx into c
        jnc over
        add ax,bx

over:   shl bx,1        ;multiply bx by 2

        loop back       ;repeat 8 times

        call Print
        .EXIT
```

RISC: Reduced Instruction Set Computer

# Multiply (The Easy Way, CISC)

```
.STARTUP

mov dl,100      ;Multiply dl by bl result in ax.
mov al,123

mul  dl         ;ax=dl*al

call Print

.EXIT
```

CISC: Complex Instruction Set

# Divide

Decimal

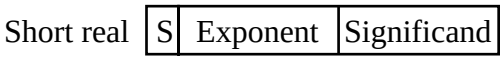
```
        0 1 1 2
   ---------
1 1 ) 1 2 3 4
      0
      ---
      1 2
      1 1
      ---
        1 3
        1 1
        ---
          2 4
          2 2
          ---
            2
```

Binary

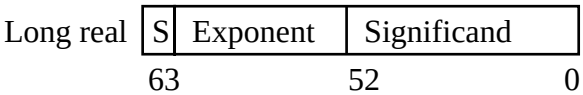
```
          0 0 1 0
       ---------
1 0 1 ) 1 1 0 1
        1 0 1
        -----
        0 0 1 1
```

Take the first digit (1) subtract 101, won't go, write down zero, take first and second digit (11) subtract 101, won't go write down zero, take first three digits (110) subtract 101, will go, write down a 1…., repeat until all digits are analysed.

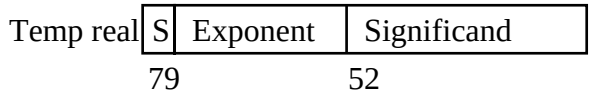
# Data format (float) in PC's




$-1.2\times10^{-38}<|X|<3.4\times10^{38}$




$-2.3\times10^{-308}<|X|<1.7\times10^{308}$




$-3.4\times10^{-4932}<|X|<1.1\times10^{4932}$

Note: exponents are offset to speed up magnitude comparison.

Short exponent biased by +127 (7FH)

*Note: Double precision division on the P90 was only calculated to single precision.*

# Floating point

Express 178.625 as a short real.

Use 4 bytes to represent the digits in the number.

178D=10110010B, Note: the binary always starts with a 1.

0.625D=0.101B,  4, 2, 1, 1/2, 1/4, 1/8,…..,

178.625D=10110010.101B

$=1.0110010101\times2^{7}$

Add bias gives $1.0110010101\times2^{7+127}=1.0110010101\times2^{134}$

134D=10000110B

Result=0,10000110,01100101010000000000000

Significand (23bits)

Exponent (8bits)

Sign

In hex 4332A000

# Floating Point Calculation

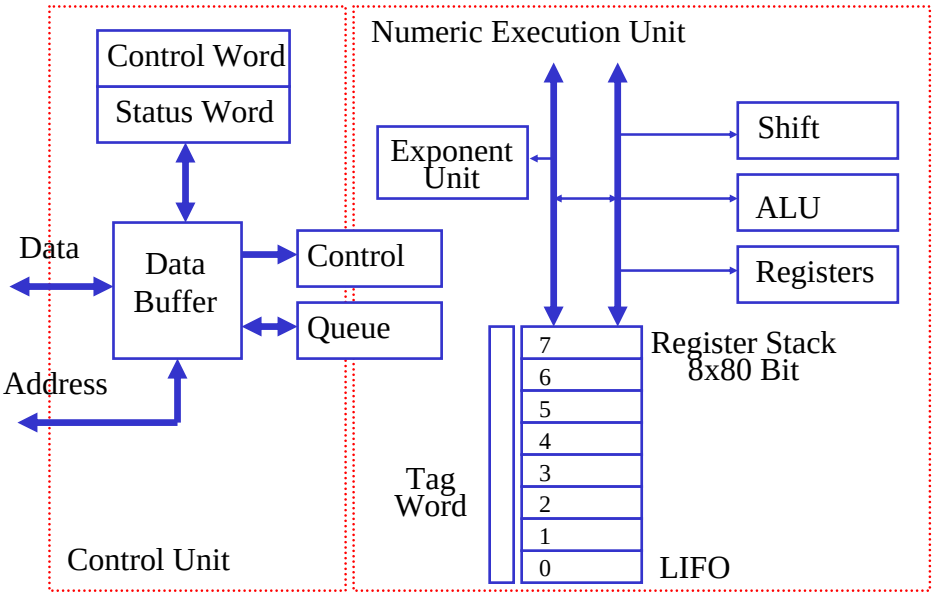
An optional coprocessor is available for many microprocessors, these chips are designed to speed up floating point calculation by as much as 100 times. From the 486 on, all Intel/PC Processors have contained an integrated co-processor.

A co-processor is a second processor, not just an enhancement to the existing processor.

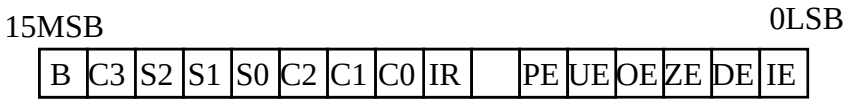
The addition of a coprocessor gives an increased instruction set.

To separate coprocessor instructions from standard instructions each coprocessor instruction starts with 11011…..

# 8087 Architecture



# The Status Word

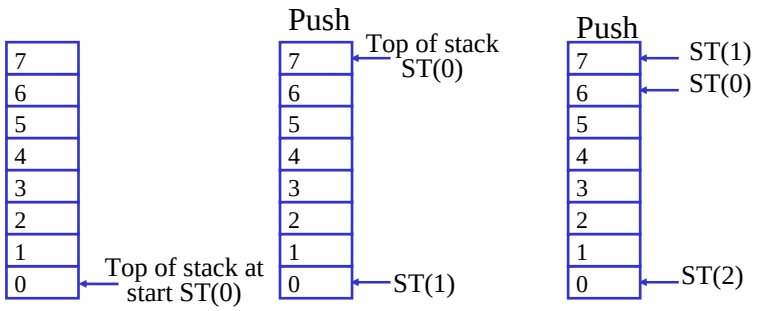



B: Busy
C2-0: Condition bits
S2-0: Top of stack
IR: Interrupt request
PE: Precision
UE: Underflow
OE: Overflow
DE: Denormalized operand
IE: Invalid operation

# The ControlWord


IC: Infinity control
R1-0: Rounding control
P1-0: Precision control
IR: Interrupt request
IE: Interrupt Enable Mask
PM: Precision
UM: Underflow
OM: Overflow
DM: Denormalized operand
IM: Invalid operation

# 8087 Stack




Each item on the stack is 10bytes, 80 bits, long double.

Stack pointer decreases on each push.

The stack can go around the clock, i.e. past 111,000.

In MASM ST(0) means the item at the top of the stack.

# 8087 Instructions

Addition

FADD *//source/destination,source*

Examples:

FADD ST(3),ST(0) ;Add ST(3) to ST(0) result in ST(3)

FADD ;Add ST(1) and ST(0) pop 1 number off the stack, result in ST(0)

FADD ST,ST(1) ;Add Stack top ST(0) to ST(1) result placed in ST(0)

# 8087 Instructions

Addition

FADD *//source/destination,source*

Examples:

```
FADD ST(3),ST(0)    ;Add ST(3) to ST(0) result in ST(3)
FADD                ;Add ST(1) and ST(0) pop 1
                     number off the stack, result in ST(0)
FADD  ST,ST(1)      ;Add Stack top  ST(0) to ST(1) result
                      placed in ST(0)
```

# 8087 Arithmetic Instructions

Addition/Subtract:

FADD *//s/d,s* : Add source to real at specified destination.

FSUB *//s/d,s* : Subtract real from specified source result in destination.

FSUBR *//s/d,s* : Subtract d-source from source, result in destination.

Multiplication:

FMUL *//s/d,s* : Multiply d-source by s-real..

Also FMULP, FIMUL

Division

FDIV *//s/d,s* : Divide d-source by s-real, result in destination.

Also FDIVR, FDIP, FIDIV etc.