Machine L versus Assembly L

Machine Language is a stream of bits

Instructions and Data are both part of the bit stream.

B241B402CD21CD20

Even written in Hex it is too difficult to decipher

and there are about 100 different instructions

But written like :

```
mov     dl, 41h
mov     ah, 02h
int     21h
int     20h
```

it makes more sense.

An ASSEMBLER (a program) translates from this to machine language leaving us free to work with mov, add etc.

Syntax:

LABEL MNEMONIC OPERANDS COMMENT

```
Start:   mov      ax, bx          ; example instruction


Start:

         mov ax,bx

         ; labels are OPTIONAL
         ; mnemonic must be followed by operand
         ; comment is OPTIONAL
         ; white space (space, Horizontal tab) between
         ; each field



         movax,bx                 ; not legal
```

Two operand instructions

```
        inst        op1, op2
```

interpreted as

```
        inst        destination, source

        add         ax, bx
```

; contents of ax <-- contents of ax + contents of bx
; contents of ax change
; contents of bx do not change
; instructions affect the DESTINATION

Immediate data   === constant data
Any quantity whose value is known or can be computed before execution of the program begins.

Memory Reference === Refers to a location in memory (obviously)!

General Rules:

1) In most one OP instructions, the OP must be a register (8 or 16 bit) or a memory reference. CANNOT be Immediate data

2) In two OP instructions, BOTH OPS must be of SAME SIZE (both are 8 bits or both are 16 bits)

3) In a two OP instruction, one OP must be a register unless the source is immediate, in which case the destination can be either a register or a memory reference

```
        inst        mem, mem            ; NOT allowed
```

Special Segment register Rules

1) A segment register can only appear in a mov, push, pop instructions

2) An immediate value cannot be moved into a segment register ? Why

3) Moves from one segment register to another segment register are Illegal

4) The cs register cannot be the destination of a mov or pop

Examples,

bdata is byte in memory
wd2, wdata are words in memory

```
mov     ax, bx

mov     ax, bl

mov     cx, bdata

mov     dx, 10

mov     bdata, 10

mov     10, bdata

mov     es, 10

mov     cs, ax

mov     dx, wdata

mov     wdata, bdata

mov     wdata, wd2
```

One consequence of segment register restrictions is that MASM standard convention is that @DATA contains data segment start, so

```
        mov     ds, @DATA
```

should intialize the data segment register.

NO, because it is illegal

instead

```
        mov     ax, @DATA
        mov     ds, ax
```

The only way to change cs is to use the equivalent of a goto or a procedure call.

Arithmetic

suppose ah = 1111 1111

```
        add     ah, ah
```

what is the result

Processor just adds the bit patterns

result = 1 1111 1110

When doing 8-bit arithmetic (the operands are 8 bit operands) the processor discards any carry out of the MSB

ah contains 1111 1110

How do we interpret the answer?

Depends on whether we view the operands as numbers or as addresses; as SIGNED or UNSIGNED

UNSIGNED Interpretation
If the operands are viewed as unsigned, then the computation in decimal is

255 + 255 = 510

too large to fit in 8 bits (0 - 255)

Consequently the PROCESSOR sets the CARRY flag to signify that there was a carry out of the MSB

if this is an important consideration, the PROGRAMMER, THIS MEANS YOU, would have inserted error-checking code after the add ah, ah instruction.

The error checking would check the CARRY flag

SIGNED Interpretation

If viewed as signed numbers

-1 + -1 = -2 no problem

no signed overflow and the machine sets the overflow flag to 0 (resets the flag)

THIS is basically the story of how the carry and overflow flags function in your processor.

For multiplication the situation is a bit different

Multiplying 2 8bit numbers give 16 bit result
Multiplying 2 16bit numbers give 32 bit result

There is also a problem with signs.

straight multiplication gives:

1111 1111 * 1111 1111 = 1111 1110 1111 1111

Unsigned:

Decimal
255 * 255 = 65025 which is

Binary
1111 1110 1111 1111

But

Signed interpretation:

Decimal
-1 * -1 = 1 which is

Binary

0000 0000 0000 0001

in other words

1111 1111 * 1111 1111 = 0000 0000 0000 0001

for signed interpretations and

1111 1111 * 1111 1111 = 1111 1110 1111 1111

for unsigned interpretations

Solution:

There are two multiplication instructions!

One for signed and one for unsigned

unsigned:

```
        mul     OP
```

signed:

```
        imul    OP
```

Single OP instruction.

Assumes that the other OP is in register   ax

16 bit numbers

```
        mul     si
```

contents of si * contents of ax

Higher order word in   dx
Low order word in   ax

Overwrites ax and dx

8 bit numbers

```
        mul     bl
```

Contents of bl * contents of al

16 bit result is in ax

```
        mul     20          ; Illegal
; One operand instructions: OP must be mem-ref or reg
```

Same problem with division:

signed and unsigned versions:

div unsigned version
idiv signed version

16 bit and 8 bit versions

Divide a 32-bit number by 16 bit number and get a 16 bit quotient and a 16 bit remainder

```
        div        OP
```

OP must be mem-ref or register

The dividend is 32 bits
MSW in dx
LSW in ax

This quantity is divided by OP
Quotient stored as 16 bit unsigned int in ax
remainder stored as 16 bit unsigned int in dx

When dividing a 32 bit quantity by 16 bit quantity result may not fit in 16 bits

processor exits program with "Divide Overflow " error!!

Other instructions:

Increment and Decrement

```
        inc     OP

        dec     OP
```

inc is the same as

```
        add     OP, 1
```

but 16 bit inc is faster than add, and flags are affected differently

Compare

```
        cmp     OP1, OP2
```

special instruction:
OP1 and OP2 not affected

flags set depending on the result of

OP1 - OP2

if OP1 - OP2 = 0 the ZERO flag is set (to 1)

if OP1 > OP2, the zero flag is reset (set to 0)

Compare is always mostly used in conjunction with other instructions to implement if...then constructs

Stack manipulation Instructions:

LIFO : Last In First Out

ss and sp registers are dedicated to the run-time stack
bp   is primarily used for stack manipulation

push   OP

pop   OP