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.
Single Operand MUST be a register or a memory-ref

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 (numerator) is 32 bits with
MSW in dx
LSW in ax

OP is the 16bit divisor (denominator)

DX AX is divided by the 16-bit 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!!

8 bit version

The dividend (numerator) is 16 bits and is in AX
OP is the 8 bit Divisor (denominator)

AX is divided by 8-bit OP

Quotient stored as 8 bit unsigned int in AL
Remainder stored as 8 bit unsigned int in AH

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

Looking at Appendix 1 in the book,
We see that the carry flag is not affected by INC/DEC but is definitely affected by ADD

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

More later...

Stack manipulation Instructions:

LIFO : Last In First Out
Bottom of Stack is in High Mem
Top of stack is in Low Mem

Only WORD sized operands (16-bit for 8086) can be stored on or removed from the stack

SS and SP registers are dedicated to the run-time stack
SS:SP points to ==> Top of Stack
(BP is also used for stack manipulation.)

Instructions that manipulate the stack:

```
push    OP
pop     OP
```

Push and Pop automatically do the right things to SP
Push:

SP is DECREMENTED by 2 (why?) and OP is stored at location pointed to by SS:SP

Pop:

One word (two bytes) popped of the stack and stored in OP
AND
SP is INCREMENTED by 2

OP must be register or Memory-ref AND
MUST be of size: One word = two bytes = 16 bits

Examples

1)

```
        push    bx
        pop     ax
```

same as

```
        mov     ax, bx
```

Well a better use might be to move the value of one segment register to another. Remember segment reg to segment reg moves are not allowed. we want to mov contents of DS to ES

```
        mov     es, ds                  ; ILLEGAL
```

instead

```
        mov     ax, ds
        mov     es, ax
```

but AX might be in use (that is, contain a value that the program is using)

therefore

```
        push    ds
        pop     es
```

would do the job.

PROGRAM CONTROL

The JMP family

Unconditional Jump:

```
1)      jmp     Target          ; goto Target
```

Target is a labeled location in the code segment

Conditional Jumps:

```
2)      je      ShortTarget   ; goto Target IF some pair of
                              ; quantities is equal


        cmp     op1, op2
        je      ShortTarget
```

; Compare op1 and op2
; AND
; IF Op1 equals Op2 THEN goto Target

So this is equivalent to jumping if the
ZERO flag is set (to 1)

JE   can also be used in other situations

```
        add     cx, 0
        je      ShortTarget

        div     cx
```

Add 0 to ax, IF contents of AX are now 0 the zero flag will be set and THEN the program will jump to ShortTarget ELSE the program will not jump to ShortTarget but do the next instruction

Better program:

```
        add     cx, 0
        je      ErrorLabel

        div     cx

        .
        .
        .

ErrorLabel:
        .
        .
        .
```

Other jumps (Total of 17 jmp instructions)

```
jne     ; Jump if not equal
jge     ; Jump greater than or equal
jg      ; Jump greater than
jle     ; Jump less than or equal
jl      l Jump less than
```

With conditional jmp intstructions

```
je      ShortTarget
```

**the instruction with the label to be jumped to
MUST NOT be more than 127 bytes from
the conditional jmp instruction**

Why do you think?

Other useful instructions

```
        lea     op1, op2
```

LOAD Effective Address of op2 into op1

op1 (destination) must be 16 bit register
op2 (source) must be a memory reference

Loads the Offset of op2 into op1

Example:

```
.DATA
        msg1    db      "Please input a string"

        .
        .
        .
.CODE
        .
        .
        .
        lea     dx, msg1
        call    WRITE
```

Addressing Modes:

Thinking about Pointers.....

In Intel Assembly you can store the address (offset) of a memory location in a register and refer to that memory location using the register

```
.DATA
        BUF     db      80 dup('$')

        .
        .
        .
.CODE
        .
        .
        .
        lea     dx, BUF
        call    READ

        lea     bx, BUF
        mov     ax, [bx]                ; moves a WORD

; or you could try

        mov     al, [bx]                ; moves a BYTE
        mov     [bx + 1], al            ; moves a BYTE
```