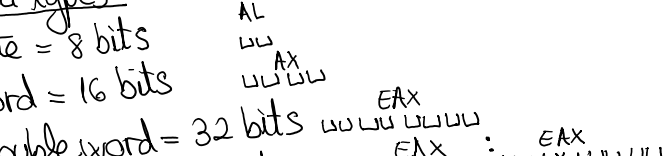
I. 1) Converting numbers between bases 2, 10 and 16
2) Representation of numbers (signed / unsigned)

## Data types

1) byte = 8 bits
2) word = 16 bits
3) doubleword = 32 bits
4) quadword = 64 bits

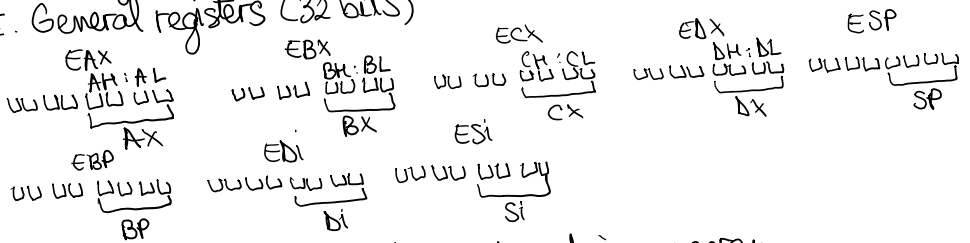



## Constants

1) numbers
   ex. 1016, A061h, -10, 123d
2) characters
   ex. 'a', 'B', 'd'
3) strings
   ex. 'abcd', "test"

## Variables

1) Pre-defined variables
   CPU registers = memory areas
   I. General registers (32 bits)




   II. Segment registers (16 bits) – not used in a program
   CS, DS, SS, ES, FS, GS

   III. EIP and Flags

2) User-defined variables

```
a db 3
b dw 1010111110106
c dd -101
```

## Instructions

1) MOV dest, source
   where dest = register / var of all types (works on segment reg too)

source = register/var(const of all types

2) ADD dest, source
where dest = register/var of all types (at most one can be var)
source = register/var(const of all types

3) SUB dest, source
where dest = register/var of all types (at most one can be var)
source = register/var(const of all types

→ Boilerplate

```
bits 32
global start

extern exit
import exit msvcrt.dll

segment data use32 class=data
    ;...

segment code use32 class=code
start:
    ;...
    push dword 0
    call [exit]
```

→ Conversions from numeration bases 2, 10 and 16

2) b2 → b10; $10111_{(2)} = 23_{(10)}$

$10111_{(2)} = 1_{(2)} * 2^0 + 1_{(2)} * 2^1 + 1_{(2)} * 2^2 + 0_{(2)} * 2^3 + 1_{(2)} * 2^4 = 1+2+4+16 = 23$

Binary digit = bit = smallest quantity of information

→ Representation of integer numbers

= the representation in base 2

1) Unsigned representation = the representation in base 2
ex. $39_{(10)} = 00010111_{(2)}$ (on 8 bits => byte/AL)
- operations: mov, add, sub, mul, div
2) Signed representation => 2's complementary code
- $-1-1x1_{(2)}$ or set the sign bit to 1 and flip all the bits from the left of the first bit of 1 from the right ($-1-1x1_{(10)}$ works too);
ex. $100_{(10)} = 01100100_{(2)} \Rightarrow 10011100_{(2)} = 2sc$ (on 8 bits)
or $-1-1-1-1-1-1-1-100$
$10000\ 0000 -$
$0110\ 0100$
$= 1001\ 1100$
- interpretation = reverse of representation;
- 2sc → number: flip all bits from the left of the first bit of 1 from the right.
ex. $10011100_{(2)} \longrightarrow 01100100$
- operations: mov, add, sub, imul, idiv, cbw, cwd, cwde;
- signed $-2^{n-1}$ = unsigned $2^{n-1} = 100...0$ ($n-1$ bits);

II. Important rules
1) All operands must have the same size/type.
2) At least one operand must be a general register or a constant and if it's a constant
ex. a db 10
...
add ax, [a] ; moves into AX (word) a word from the memory that ; starts where 'a' is. (rule #1 is broken).

Instructions
1) MUL source
where source = register/var of b/w/d. (! cannot be const)
- source = byte => AX = AL * source
- source = word => DX:AX = AX * source
- source = dword => EDX:EAX = EAX * source
- results are not stored on EAX instead of DX:AX because of rollback

compatibility (previous CPUs were on 16 bits).

ex. 12345678h
highpart (DX) lowpart (AX)

2) DIV source
where source = register/var of b/w/d (!, cannot be const)
- source = byte => AX = AL / source
- source = word => DX:AX = AX / source
- source = dword => EDX:EAX = EAX / source
- EDX stores the remainder, EAX the quotient.

Obs. • When you MUL/DIV directly by a variable, you must specify its size before ([var] is just a memory reference).
ex. a db 5
...
mul byte [a]

• When working with unsigned representation, let's say we have to subtract AX from BL. We have to convert it to BX by doing:
mov BH, 0 ; adding insignificant zeros to the left.

• When the number is negative, we have to fill it with 1. So for signed representation, we have to determine whether the number is positive (fill with 0) or negative (fill with 1). We can make use of the following instructions:

3) CBW = convert byte to word (AL to AX)
4) CWD = convert word to doubleword (AX to DX:AX)
5) CWDE = convert word to doubleword extended (AX to EAX)
6) CDQ = convert doubleword to quadword (EAX to EDX:EAX)

• For unsigned:
AL → AX : mov AH, 0
AX → DX:AX : mov DX, 0
EAX → EDX:EAX : mov EDX, 0

7) INC dest
where dest = register/var of all types
- increments dest.

8) DEC dest
where dest = register/var of all types

- decrements dest.

9) NEG dest
where dest = register/var of all types
⇒ dest = -dest

10) ADC dest, source
where dest = register/var of all types (at most one can be var)
source = register/var/const of all types
⇒ dest = dest + source + CF

11) SBB dest, source
where dest = reg/var of all types (at most one can be var)
source = reg/var/const of all types
⇒ dest = dest - source - CF

12) iMUL source
where source = reg/var of b/w/d. (!, cannot be const)

13) iDIV source
where source = reg/var of b/w/d. (! cannot be const)

## Declaring variables without initial values

1) a RESB 1 ; reserving 1 byte
b RESB 64 ; reserving 64 bytes
2) c RESW 1 ; reserving 1 word

## Declaring constants

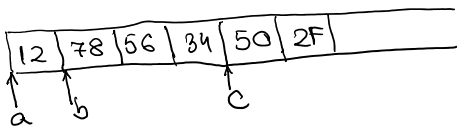
1) a EQU 5

## Little endian representation

One byte is 8 bits = 2 hex digits (so in the memory it is ww)
All bytes are reversed in representation.

ex. a db 12h
b dd 345678h
c dw 0010111101010000b ; = 2F50h

⇒




1) CLC = clear carry flag

## Putting a quadword 'a' on reg

mov EAX, [a+0] ; little endian!!
mov EDX, [a+4]

## Bitwise operations

1) AND dest, source
   where dest = reg/var of all types
   source = reg/var/const of all types

2) OR dest, source
   where dest = reg/var of all types
   source = reg/var/const of all types

3) XOR dest, source
   where dest = reg/var of all types
   source = reg/var/const of all types

4) TEST dest, source
   where dest = reg/var of all types
   source = reg/var/const of all types
   - executes AND without storing the result in dest.

5) NOT dest
   where dest = reg/var of all types

6) SHL = SAL dest, source
   where dest = reg/var of all types
   source = const/CL
   - last disappearing bit is kept in CF

7) SHR dest, source
   where dest = reg/var of all types
   source = const/CL
   - last disappearing bit is kept in CF.

8) SAR dest, source
   where dest = reg/var of all types
   source = const/CL
   - last disappearing bit is kept in CF;
   - leftmost bits are filled with the sign bit.

9) ROL, ROR -11-

10) RCL, RCR -11-
    - bit that goes out is stored in CF and then added

11) STC = set carry flag (with 1)

ex. (10+11)
```
stc ; CF=1
mov al, 00110011b
mov cl, 2                    ; CF=0
```

rcl al, cl ; AL= [illegible]

Isolating bits 2-4

```
mov al, [a]
and al, 00011100b
```

Setting bits 2-4 to 0

```
mov al, [a]
and al, 11100011b
```

Setting bits 2-4 to 1

```
mov al, [a]
or al, 00011100b
```

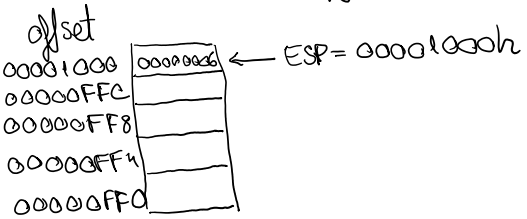
Move from $x_1$-$y_1$ to $x_2$-$y_2$

- compute the difference and rotate properly.

## Stack

runtime stack = memory arranged directly by the CPU, using ESP (extended stack pointer)

- ESP holds 32-bit offset. We rarely manipulate it directly;

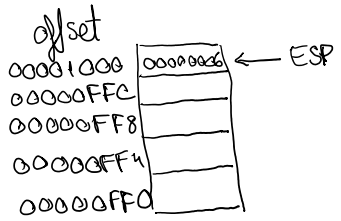



1) PUSH

- a 32-bit push operation decrements ESP by 4.


2) POP

- a 32-bit pop operation increments ESP by 4;
- the area of the stack below ESP is logically empty.




3) Uses

- temporary save area;

- when CALL executes, CPU saves current [illegible]
  on the STACK;
- when calling a subroutine, its arguments are pushed on the stack;
- temporary storage for local variables in subroutines.

PUSH → register;
  ↳ memory variable;
  ↳ constants.

POP → register;
  ↳ memory variable.

PUSHFD = pushes 32-bit EFLAGS register
POPFD = pops 32-bit EFLAGS register
PUSHF = pushes 16-bit FLAGS register
POPF = pops 16-bit FLAGS register

- MOV cannot be used to copy the flags of a variable.
- best practice: push and then pop ASAP (or immediately).

ex. .data
saveFlags DW 0
.code
pushfd
pop saveFlags
...
push saveFlags
popfd

or

pushfd
...
popfd

(more prone to errors)

PUSHAD = pushes in this order: EAX, ECX, EDX, EBX, ESP (before executing PUSHAD), EBP, ESI, EDI
POPAD = pops them in reverse order
PUSHA = AX, CX, DX, ... (16-bit)
POPA = — " —
- bad practice to be used in subroutines (POPAD overwrites EAX).
- EBP is used in high-level languages to reference function parameters and local variables. It should not be used for basic arithmetic or data transfer except at an advanced level. (extended frame pointer)

Jumps and loops
CMP = fictional subtraction, affecting flags OF, SF, ZF, AF, PF, CF and source → register;

## Jumps and loops

CMP = fictional subtraction, affecting flags OF, SF, ZF, AF, PF, CF

CMP dest, source where dest → register; / memory variable and source → register; / memory v.; / constant

- dest, source of same size

TEST = OF = 0, CF = 0, SF, ZF, PF - modified, AF - undefined

- comparing two signed numbers: less than, greater than;
- comparing two unsigned numbers: below, above.

## String operations

1) dest + source: MOVSB, MOVSW, MOVSD, CMPSB, CMPSW, CMPSD;
2) dest: STOSB, STOSW, STOSD, SCASB, SCASW, SCASD;
3) source: LODSB, LODSW, LODSD.

## A string is characterized by:

1) the type of the elements (bytes or words) - both strings have the same type
2) the address of the first element - FAR:
   - dest: ES:EDI
   - source: DS:ESI
3) the parsing direction - DF flag (0 - from small addresses to large)
4) the number of elements - CX or ECX

LODS + STOS = MOVS

- if DF = 0 : increase ; else: decrease

LODSB: <DS:ESI> → AL
LODSW: <DS:ESI> → AX
LODSD: <DS:ESI> → EAX

STOSB: AL → <ES:EDI>
STOSW: AX → <ES:EDI>
STOSD: EAX → <ES:EDI>

MOVSB / MOVSW / MOVSD: <DS:ESI> → <ES:EDI>

SCASB: CMP AL, <ES:EDI>
SCASW: CMP AX, <ES:EDI>
SCASD: CMP EAX, <ES:EDI>

CMPSB / CMPSW / CMPSD: CMP <DS:ESI>, <ES:EDI>

repetitive prefix string_instruction (=) Again: string_instruction

repetitive_prefix string_instruction (=) Again:
string_instruction
Loop Again

repetitive_prefix → REP, REPE, REPZ ... on SCAS/CMPS ... ECX=0 / ZF=0 (a mismatch occurs)
↳ REPNE, REPNZ ... on SCAS/CMPS ... ECX = 0 / ZF = 1 (a match occurs)

!! LODS, STOS, MOVS - do not change flag

ADDRESS COMPUTATION FORMULA

offset = base + (index * scale) + displacement

| | | | |
|---|---|---|---|
| EAX | EAX | 1 | none |
| EBX | EBX | 2 | 8-bit |
| ECX | ECX | 4 | 16-bit |
| EDX | EDX | 8 | 32-bit |
| ESP | EBP | | |
| EBP | ESI | | |
| ESI | EDI | | |
| EDI | | | |

ex. mov ax, [a] - only displacement
mov ax, [eax] - only base or index
mov ax, [a+eax+ebx] - base, index and displacement
mov ax, [eax+eax+a+2] - base, index and displacement
mov ax, [a+4+ebx*2] - index, scale and displacement
mov ax, [eax+ebx*4+20] - base, index, scale and displacement