

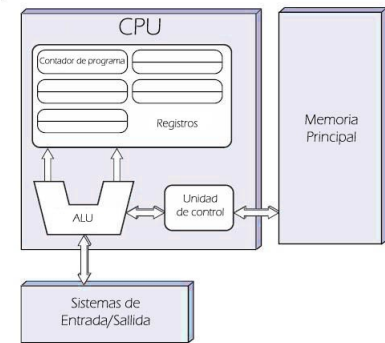
# FONAMENTS D'ORDINADORS

## TEMA 5: IA32

Manel Guerrero

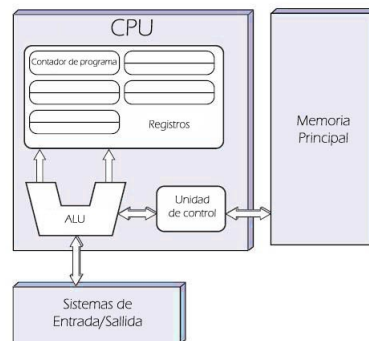
# [H1] Arquitectura Von Neumann

- La majoria dels ordinadors segueixen l'estructura de Von Neumann (circa 1950): Processador, memòria i E/S connectats per un bus.
- La memòria és una matriu de bits de 'N' files i 'm' columnes.
- El bus es té tres parts:
  - dades (m bits (o 2m, o 4m, ... ))
  - adreces (n bits,  $n=\log_2(N)$ )
  - control (1 bit).



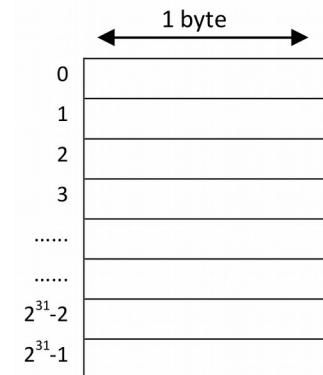
## Unitat Central de Procés (CPU)

- Unitat Aritmètic Lògica (ALU): És com una mini calculadora que fa operacions aritmètiques i de lògica booleana.
- Unitat de Control (CU): interpreta les instruccions i genera els senyals de control necessaris per executar-les.
- Registres: Són com variables del processador. Llegir o escriure d'un registre és molt més ràpid que llegir o escriure a la memòria.



## La memòria del IA32

- El IA32 (Intel 386 i posteriors) té la memòria organitzada en N paraules de 8 bits (1 byte). Cadascun d'aquests bytes s'adreça del 0 al N-1.
- Al ser una arquitectura de 32 bits, la mida del bus de dades/adreces i dels registres és de 32 bits.
- Quan llegeix o escriu sempre són 4 bytes en adreces múltiples de 4.
- Al tenir un bus d'adreces de 32 bits, podem adreçar 4Gigabytes ( $2^{32}$  bytes).



Modelo plano de memoria

## Les dades a la memòria

- Les instruccions dels programes i les dades amb les que treballen es guarden a memòria.
- Les dades:
  - Naturals (enters positius): en binari pur.
  - Caràcters: en ASCII
  - Enters amb signe: en Complement a 2 (complement a 1 + 1)

n	bin	n	Ca1	n	Ca2
0	000	3	011	3	011
1	001	2	010	2	010
2	010	1	001	1	001
3	011	0	000	0	000
4	100	-0	111	-1	111
5	101	-1	110	-2	110
6	110	-2	101	-3	101
7	111	-3	100	-4	100

## Complement a 2

- Els positius es guarden en binari pur i els negatius en Ca2.
- $Ca2 = Ca1 + 1$
- Conversió ràpida per humans: #1: començant per la dreta copiar el número original fins el primer 1 (primer 1 inclòs). #2: Després, es neguen (complementen) els dígitos restants (es a dir, substituïm 0 per 1 i viceversa).
  - Ex: 92  $\rightarrow$  -92
  - #1 01011100  $\rightarrow$  ?????100
  - #2 01011100  $\rightarrow$  10100100
- El Complement a 2 és molt pràctic per als ordinadors per que el bit de més pes determina el signe i podem sumar números sense pensar si són amb signe o sense i funciona.
- A més per restar A-B podem fer  $A+Ca2(B)$ .
- Exemple de suma:
 

(181)	1 0 1 1 0 1 0 1	(-75)
( 59)	0 0 1 1 1 0 1 1	(+59)
	+--- +----- +---	
(240)	1 1 1 1 0 0 0 0	(-16)

## Els programes a la memòria

- Els programes són seqüències d'instruccions.
- Les instruccions dels programes es codifiquen com una seqüència de bytes a memòria. Cada instrucció té:
  - Codi d'Operació.
  - Operador Font #1
  - Operador Font #2
  - Operador Destí
- Alguns operadors poden ser implícits. Exemple:
  - C:  $EAX = EAX + 2;$
  - ASM:  $ADDL \$2, \%EAX$
  - CM:  $0x\ 83\ C0\ 02$
  - CO:  $83\ (ADDL); OF1: 02;$   
 $OF2: C0\ (\%EAX); OD: C0.$
- Per cada operand es diu el mode d'adreçament (si és un està en un registre, o en memòria, o si donem el valor directament) i el valor.

## Execució d'un programa

- El S.O. carrega el programa a memòria.
- El registre %eip guarda el valor de la primera instrucció a executar.
- La CPU va executant les instruccions en ordre seqüencial
- %eip sempre guarda el valor de la següent instrucció a executar.
- Existeixen instruccions que poden trencar l'execució seqüencial: salts i crides a subrutines).
- El programa acaba amb una instrucció especial que retornen el control al S.O.
- Per cada instrucció la CPU fa:
  - Fetch: Anar a buscar a memòria la instrucció a executar.
  - Descodificació: Determinar el tipus d'operació a realitzar i on es troben els operands.
  - Obtenir operands font.
  - Execució.
  - Escriure el resultat a l'operand destí.
  - Incremento %eip.
  - En anglès: "fetch, decode, execute, and store".

# [H2] Registres

%EAX <sub>0..31</sub>		%AH <sub>8..15</sub>	%AL <sub>0..7</sub>	%AX <sub>0..15</sub>
%EBX <sub>0..31</sub>		%BH <sub>8..15</sub>	%BL <sub>0..7</sub>	%BX <sub>0..15</sub>
%ECX <sub>0..31</sub>		%CH <sub>8..15</sub>	%CL <sub>0..7</sub>	%CX <sub>0..15</sub>
%EDX <sub>0..31</sub>		%DH <sub>8..15</sub>	%DL <sub>0..7</sub>	%DX <sub>0..15</sub>
%EBP <sub>0..31</sub>				%BP <sub>0..15</sub>
%ESP <sub>0..31</sub>				%SP <sub>0..15</sub>
%ESI <sub>0..31</sub>				%SI <sub>0..15</sub>
%EDI <sub>0..31</sub>				%DI <sub>0..15</sub>
%EFLAGS <sub>0..31</sub>				%FLAGS <sub>0..15</sub>
%EIP <sub>0..31</sub>				%IP <sub>0..15</sub>

## General-Purpose Registers (GPR)

- AX (Accumulator). Per operacions aritmètiques.
- CX (Counter). Per bucles (com la var 'i' en C).
- DX (Data). Per operacions aritmètiques i de I/O.
- BX (Base). Punter a dades o a primer element de vector.
- SP (Stack Pointer). Punter a la pila (al sostre de).
- BP (Stack Base Pointer). Punter a la base de la pila.
- SI (Source Index). Punter a origen o índex d'origen.
- DI (Destination Index). Punter a destinació o índex de destí.

# Registres

%EAX <sub>0..31</sub>		%AH <sub>8..15</sub>	%AL <sub>0..7</sub>	%AX <sub>0..15</sub>
%EBX <sub>0..31</sub>		%BH <sub>8..15</sub>	%BL <sub>0..7</sub>	%BX <sub>0..15</sub>
%ECX <sub>0..31</sub>		%CH <sub>8..15</sub>	%CL <sub>0..7</sub>	%CX <sub>0..15</sub>
%EDX <sub>0..31</sub>		%DH <sub>8..15</sub>	%DL <sub>0..7</sub>	%DX <sub>0..15</sub>
%EBP <sub>0..31</sub>				%BP <sub>0..15</sub>
%ESP <sub>0..31</sub>				%SP <sub>0..15</sub>
%ESI <sub>0..31</sub>				%SI <sub>0..15</sub>
%EDI <sub>0..31</sub>				%DI <sub>0..15</sub>
%EFLAGS <sub>0..31</sub>				%FLAGS <sub>0..15</sub>
%EIP <sub>0..31</sub>				%IP <sub>0..15</sub>

## Flags

- 0. CF : Carry Flag. 1 si l'última operació aritmètica ha tingut un 'carry' (suma) o un 'borrow' (resta) més enllà de l'últim bit. Considerant que era una operació sense signe.
- 6. ZF : Zero Flag. 1 si el resultat de l'última operació es 0.
- 7. SF : Sign Flag. 1 si el resultat de l'última operació es un número negatiu.
- 11. OF : Overflow Flag. 1 si el valor resultant de l'última operació és massa gran per caber en un registre en el cas de que fos una operació amb signe.

## Instruction Pointer (IP)

- Conté l'adreça de la propera instrucció a executar, a no ser que es faci un salt o una crida a una funció.

# Carry vs Overflow

CF	ss	bits	as	OF	CF	ss	bits	as	OF
	13	1101	-3			5	0101	5	
	+ 5	+0101	+ 5			- 2	-0010	- 2	
	---	-----	---			---	-----	---	
1	18!=2	10010	2 = 2	0	0	3= 3	00011	3 = 3	0
CF	ss	bits	as	OF	CF	ss	bits	as	OF
	11	1011	-5			6	0110	6	
	+ 9	+1001	+ -7			-15	-1111	--1	
	---	-----	---			---	-----	---	
1	20!=4	10100	4!=-12	1	1	-9!=7	10111	7 = 7	0

# Carry vs Overflow 2

- Una altra manera d'entendre Overflow:
  - El bit més alt d'un número negatiu és 1. El bit més alt d'un número positiu és 0.
  - Overflow quan: pos + pos = neg OR neg + neg = pos
  - Podem deduir què en la resta tindrem Overflow quan: pos - neg = neg OR neg - pos = pos
- Feu l'exercici 1.

## Tipus de dades i endianisme

- Tipus de dades en IA32:
 

@ Val	
0 0x34	Byte: 8 bits. (b)
1 0xAF	Word: 16 bits. (w)
2 0x00	Long: 32 bits. (l)
3 0xFF	Posicions de memòria. (l)
- Com es guarden en una memòria lineal?
 

4 0x00	Format little endian: byte de major pes a la posició més alta de memòria (intel, alpha)
5 0x01	0x03020100
6 0x02	Format big endian: byte de major pes a la posició més baixa de memòria (IBM, Sun)
7 0x03	

## Alineació

- Alineació: Com que degut a l'arquitectura el bus de dades de 32 bits només pot accedir a adreces múltiples de 4, per eficiència, alinearem els words en adreces múltiples de 2 i els longs en adreces múltiples de 4. Si no ho fem així accedir a aquestes variables requerirà dos accessos a memòria.
 

@ Val	
0 0x34	low a @0: OK
1 0xAF	la @1: 2 accessos
2 0x00	wa @2: OK
3 0xFF	wa @3: 2 accessos
-----	
4 0x00	low a @4: OK
5 0x01	la @5: 2 accessos
6 0x02	wa @6: OK
7 0x03	
-----	

## Modes d'adreçament

- Per indicar on està un operador d'una instrucció. La instrucció mov:
 

```
movl %eax, %ebx
```
- 1. Immediat: \$<valor>
 

```
[%ebx = %eax]
```

  - Trampa: l'operador de destí no pot ser immediat.
 

```
movw $1000, %bx
```

```
[%bx = 1000]
```
- 2. Registre: %<reg>
 

```
movb $'A', %al
```

```
[%al = 'A' (=67)]
```

  - Ex: Assignar un 0 al registre eax:
 

```
movl $0, %eax
```

## Modes d'adreçament 2

- 3.a. Memòria, Normal
 

@ Val	(ecx=0x00000002)
0 0x97	movb -2(%ecx), %al
1 0xAF	al=0x97 ('a')
- 3.b. Memòria, Base + Desplaçament
 

2 0x00 \	movw (%ecx), %ax
3 0xFF /	ax=0xFF00
4 0x00 \	movl 2(%ecx), %eax
5 0x01	eax=0x03020100
- Trampa: Solament 1 operand pot estar en memòria.
 

6 0x02	
7 0x03 /	

  - movl (%eax), (%ebx) -> ERROR!
  - movl a, b -> ERROR!

## “Punters” en ASM

ASM	C	Descripció
\$1	1	Número 1
1	&1 (!)	Valor en la posició de memòria 1
i	i	Valor de la variable i
\$i	&i	Posició de memòria on està i
%eax	eax	Valor guardat en el registre
(%eax)	*eax	Valor guardat en la posició de memòria indicada pel registre

## [H3] Modes d'adreçament 3

- 3. Memòria, Indexat: D(Rb,Ri,Fe)  
L'operand és a l'@ de memòria  
D+Rb+Ri\*S  
Copia @[4+%ebp+%ebx] a %eax
  - D: Desplaçament: Constant o variable.
  - Rb: Registre base: Qualsevol registre.
  - Ri: Registre índex: Qualsevol registre menys %esp.
  - S: Scale factor: 1, 2, o 4 (b, w, o l).
- Si un d'ells no apareix, per defecte, D=0, Rb=0, Ri=0 i Fe=1.  
Copia @[eax+%ebx] a %edx

## El meu primer programa en I32

```
t5_p01_add.s:
# int a=2, b=3, r;
# r = a + b;
#===[Execute]=====
# $ gcc -o p01_sum p01_sum.s -g
# $ ./p01_sum
# $ echo $?
# 5
.data      # initialized
a: .long 2 # int a = 2;
b: .long 3 # int b = 3;
.bss      # uninitialized
.comm r,4,4 # int r;

.text      # code segment
.global main # For the OS
main: # main() {
    movl a, %eax # a -> eax
    addl b, %eax # b+eax -> eax
    movl %eax, r # eax -> r
    # LINUX EXIT
    #movl $0, %ebx # Return 0
    movl r, %ebx # Ret r (%bl)
    movl $1, %eax
    int $0x80
```

## Instruccions 1: Transferència de dades

- mov[b|w|l] op1, op2
  - Copia byte, word o long de l'op1 a l'op2
  - Ex: movl var, %ebx
- movs[bw|bl|wl] op1, op2
  - Copia l'op1 a l'op2 estenent el seu signe de byte a word o long, i de word a long (conversió de tipus)
  - Ex: movl \$-1, %al movsbl %al, %ebx
  - Perquè -1 = 0xFF, i volem que %ebx valgui -1 (0xFFFFFFFF) i no 0x000000FF (+255)
  - movz enlloc de movs fa “zero extension” enlloc de “sign extension”.
- push i pop. Les veurem més endavant quan estudiem la pila i com fer crides a funcions.

## Instruccions 2: Operacions Aritmètiques

Instrucció	Operació
add[b w l] op1, op2	# op2 = op2 + op1
sub[b w l] op1, op2	# op2 = op2 - op1
cmp[b w l] op1, op2	# op2 - op1 (op2 no pot ser immediat!)
inc[b w l] op1	# op1++
dec[b w l] op1	# op1--
neg[b w l] op1	# op1 = -op1
imul[b w l] op1	# %ax = %al * op1 (b) # %dx:%ax = %ax * op1 (w) # %edx:%eax = %eax * op1 (l)
imul[b w l] op1, op2	# op2 = op1 * op2 (op2 registre)
imul[b w l] op1, op2, op3	# op3 = op1 * op2 (op1 immediat, op2 reg/mem, op3 reg)

## Ex: El meu segon programa en I32

```

t5_p02_inc.s                                main: # main() {
# int i=0, j=3, r;                            incl i # i++
# i++; j--; j = -j;                          decl j # j--
# r = i - j;                                negl j # j = -j
.data # initialized                          movl i, %eax # i-> eax
i: .long 0 # int i = 0;                      subl j, %eax # i-j -> eax
j: .long 3 # int j = 3;                      movl %eax, r # i-j -> r
.bss # uninitialized                         # LINUX EXIT
.comm r,4,4 # int r;                          movl r, %ebx # Return r
.text # code segment                         movl $1, %eax
.global main                                  int $0x80

```

## [H4] El meu tercer programa en I32

```

t5_p03_imul.s:                               movl i, %edx
# #define N = 10                             addl j, %edx
# int i=2, j=3, k=6;                         movl %edx, %eax
# r = (i+j) * ( k+(-2)) + N;                 movl k, %edx
N = 10 # define N 10                         addl $-2, %edx
.data # initialized                          imul %edx, %eax
i: .long 2 # int i = 2;                       addl $N, %eax
j: .long 3 # int j = 3;                       movl %eax, r
k: .long 4 # int k = 4;
.bss # uninitialized
.comm r,4,4 # int r;
.text # code segment
.global main
main: # main() {

```

## Instruccions 3: Control de flux

Instrucció	Comportament	Flags
cmp op1, op2	Abans de salts condicionals (op2 - op1)	Seteja els flags ZF, SF i OF
jmp etiq	Salta a etiq	
je etiq	Salta si op2==op1	ZF = 1
jne etiq	Salta si op2!=op1	ZF = 0
jg etiq	Salta si op2>op1	ZF = 0 i SF = OF
jge etiq	Salta si op2>=op1	SF = OF
jl etiq	Salta si op2<op1	SF != OF
jle etiq	Salta si op2<=op1	ZF = 1 o SF != OF
call etiq	Crida a funció	

## Ex: Com fer "if"s en ASM

```
t5_if.s
# int a=3, b=3, r;
# if (a >= b) r=1; else r=0;
.data # initialized
a: .long 3 # int a = 3;
b: .long 3 # int b = 3;
.bss # uninitialized
.comm r,4,4 # int r;

.text # code segment
.global main
main: # main() {
    movl a, %eax
    cmpl b, %eax # if (a >= b)
    jle else # {
    movl $1, r # r=1;
    jmp endif # }
else: # else {
    movl $0, r # r=0;
endif: # }
    # LINUX EXIT
    movl r, %ebx # Return r
    movl $1, %eax
    int $0x80
```

## Ex: Com fer "do\_loop" en ASM

```
t5_do_loop.s
# int f=1, n=3, i=1;
# // f -> %eax, i -> %ecx
# do {f=f*i;i++;}while(i<=n);
# retornar f; // f <- n
factorial
.data
n: .long 3
.text
.global main
main:
    movl $1, %eax # f=1;
    movl $1, %ecx # i=1;
do_loop: # do {
    imull %ecx, %eax # f*=i;
    incl %ecx # i++;
    cmpl n, %ecx #} while(i<n);
jle do_loop

    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

## Ex: Com fer "loop" en ASM

```
t5_loop.s
# int f=1, n=3, i;
# i=1; while(i<=n) {f=f*i;i++;}
# retornar f;
# // f <- n factorial
.data # initialized
f: .long 1 # int f = 1;
n: .long 3 # int n = 3;

.text # code segment
.global main
main: # main() {
    movl f, %eax
    movl $1, %ecx # i=1;
loop: # while
    cmpl n, %ecx # (i<=n) {
    jg end_loop
    imull %ecx, %eax # f=f*i;
    # Op2 en imul ha de ser registre!
    incl %ecx # i++;
    jmp loop
end_loop: # }
    movl %eax, %ebx # Ret r
    movl $1, %eax
    int $0x80
```

## [H5] Ex: while (a && b) en ASM

```
t5_loop_and.s
# int n=1, i=0;
# while (n<100 && i<64) {
# n=n*2; i++;} retornar n;
.data # initialized
n: .long 1 # int n = 1;
.text # code segment
.global main
Main:
    movl n, %eax
    movl $0, %ecx
loop: #((%eax<100)&&(%ecx<64))
    cmpl $100, %eax
    jge end_loop # %eax>=100
    cmpl $64, %ecx
    jge end_loop # %ecx>=64
    imull $2, %eax # Op2 reg!
    incl %ecx
    jmp loop
end_loop:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

## Ex: while (a || b) en ASM

```
t5_loop_or.s
# int n=7; m=1;
# while (n<100||m<100) {
# n=n*2; m=m*3;} ret n;
.data      # initialized
n: .long 7 # int n = 7;
m: .long 1 # int m = 1;
.text      # code segment
.global main
main:
    movl n, %eax
    movl m, %ebx
    # while (%eax<100||%ebx<100)

loop:
    cmpl $100, %eax
    j1 body_loop # %eax<100
    cmpl $100, %ebx
    jge end_loop # %ebx>=100
body_loop: # cuerpo_bucle
    imull $2, %eax # Op2 reg!
    imull $3, %ebx # Op2 reg!
    jmp loop
end_loop:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

## [H6] Ex: bucles niuats en ASM

```
t5_loop_loop.s
# n=0;
# for (i=1;i<5;i++)
#   for (j=i;j<5;j++)
#     n++; ret n;
.text # code segment
.global main
main: # main() {
    movl $0, %eax
    movl $1, %esi # %esi=1
loop_esi:
    cmpl $5, %esi
    jge end_for_esi # %esi>=5
    movl %esi, %edi # %edi=%esi

loop_edi:
    cmpl $5, %edi
    jge end_for_edi # %edi>=5
    incl %eax
    incl %edi # %edi++
jmp loop_edi
end_loop_edi:
    incl %esi # %esi++
jmp loop_esi
end_loop_esi:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

## Estructura d'un programa en ASM

```
# Definició de constants
N = 20      # decimal
X = 0x4A    # hexadecimal
B = 0b010011 # binari
C = 'J'     # caràcter
.data      # initialized
ii: .long 1 # int ii = 1;
ss: .word 2 # short ss = 2;
cc: .byte 3 # char cc = 3;
s1: .ascii "Hola\n" # Chars
s2: .asciz "Hola\n" # Chars+\0
.bss # uninitialized
.global nom_var # variable global
#(visible des de fitxers externs)

# .comm nom_var, tamany, alineació
.comm i, 4, 4 # int i;
.comm s, 2, 2 # short s;
.comm c, 1, 1 # char c;

.text # Secció de codi
.global main # main visible pel SO
main:
...
# Finalitzar el programa en Linux
movl $0, %ebx # Ret %bl al SO
movl $1, %eax
# $1 = funció sys_exit del SO
int $0x80 # Executa la interrup
```

## Problemes

- Fer un programa complet que faci el que se'ns demana al problema 6.
- Transformar el programa perquè calculi el factorial d'un número.
- Dubtes?