

Cours Architecture des Ordinateurs

Programmation Assembleur

Jean-Claude Bajard

IUT - université Montpellier 2

Présentation du Pentium

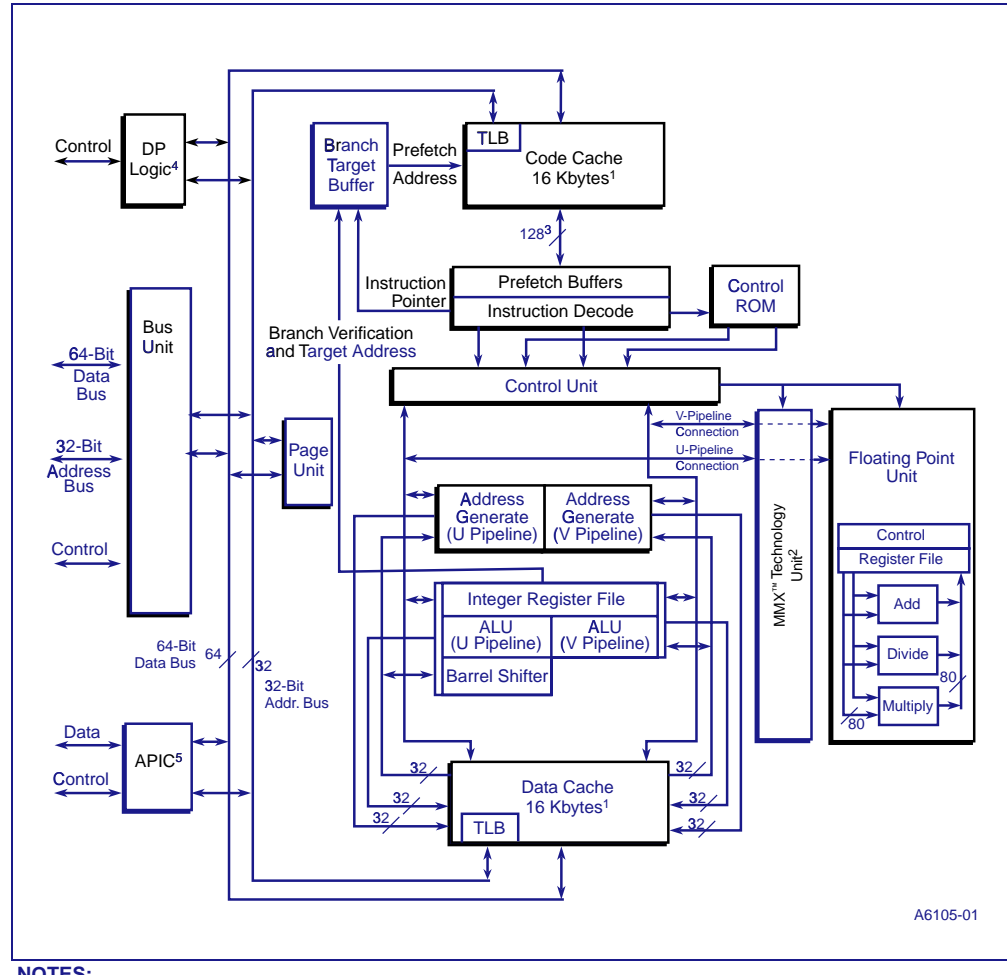
<http://developer.intel.com/design/pentium4/manuals/>

...

Table 2-1. Processor Performance Over Time and Other Intel Architecture Key Features

Intel Processor	Date of Product Introduction	Performance in MIPS ¹	Max. CPU Frequency at Introduction	No. of Transistors on the Die	Main CPU Register Size ²	Extern. Data Bus Size ²	Max. Extern. Addr. Space	Caches in CPU Package ³
8086	1978	0.8	8 MHz	29 K	16	16	1 MB	None
Intel 286	1982	2.7	12.5 MHz	134 K	16	16	16 MB	Note 3
Intel386™ DX	1985	6.0	20 MHz	275 K	32	32	4 GB	Note 3
Intel486™ DX	1989	20	25 MHz	1.2 M	32	32	4 GB	8KB L1
Pentium®	1993	100	60 MHz	3.1 M	32	64	4 GB	16KB L1
Pentium® Pro	1995	440	200 MHz	5.5 M	32	64	64 GB	16KB L1; 256KB or 512KB L2
Pentium II®	1997	466	<u>266</u>	7 M	32	64	64 GB	32KB L1; 256KB or 512KB L2
<u>Pentium® III</u>	<u>1999</u>	<u>1000</u>	<u>500</u>	<u>8.2 M</u>	<u>32 GP</u> <u>128</u> <u>SIMD-FP</u>	<u>64</u>	<u>64 GB</u>	<u>32KB L1;</u> <u>512KB L2</u>

Figure 2-1. Embedded Pentium® Processor Block Diagram



NOTES:

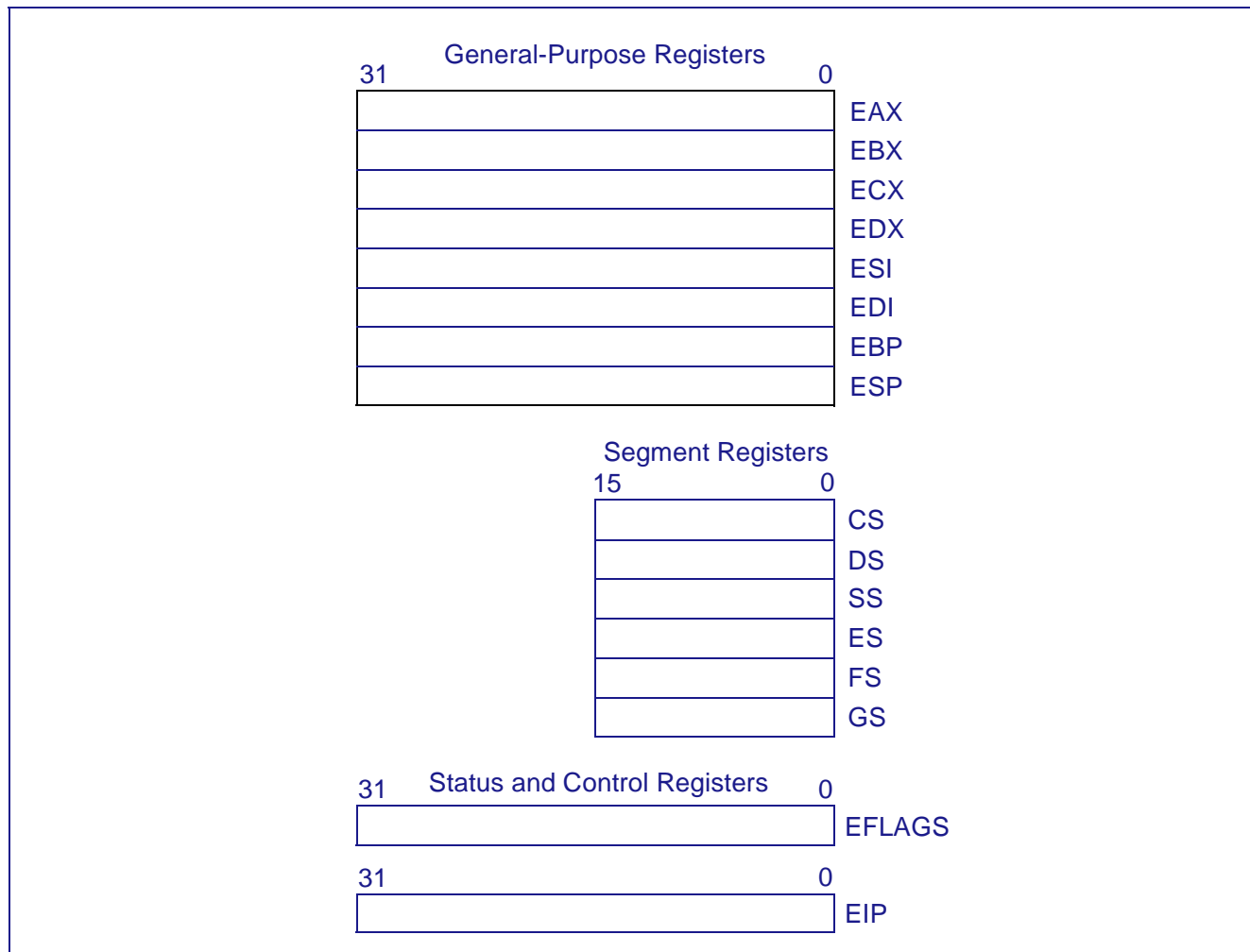


Figure 3-3. Application Programming Registers

Programmation Assembleur format ATT- GNU

<http://www.gnu.org/manual/>

...

Format d'une instruction assembleur (gnu ATT)

étiquette:	mnémonique	opérandes	#commentaires
-------------------	-------------------	------------------	----------------------

- **étiquette:** *adresse effective de l'instruction, utile pour les branchements*
- **mnémonique** *nom générique donné à une instruction : ADD, JUMP...*
- **opérandes** *arguments de l'instruction : 0, 1 ou 2*
- **#commentaires** *non pris en compte au moment de l'assemblage*

Type des opérandes

opérande 1 : source	opérande 2 : destination
registre	registre
immédiat	registre
mémoire	registre
registre	mémoire
immédiat	mémoire

General-Purpose Registers					
31	16 15	8 7	0	16-bit	32-bit
	AH	AL		AX	EAX
	BH	BL		BX	EBX
	CH	CL		CX	ECX
	DH	DL		DX	EDX
	BP				EBP
	SI				ESI
	DI				EDI
	SP				ESP

Figure 3-4. Alternate General-Purpose Register Names

Les modes d'adressages de la mémoire

- absolu \Rightarrow 0x8049514
- registre indirecte :
 - * base \Rightarrow (%eax)
 - * base + déplacement \Rightarrow 4(%eax)
 - * base + index * pas + déplacement \Rightarrow dep(base,index,scale)
 \Rightarrow 4(%eax,%ebx,2)

Les modes d'adressages

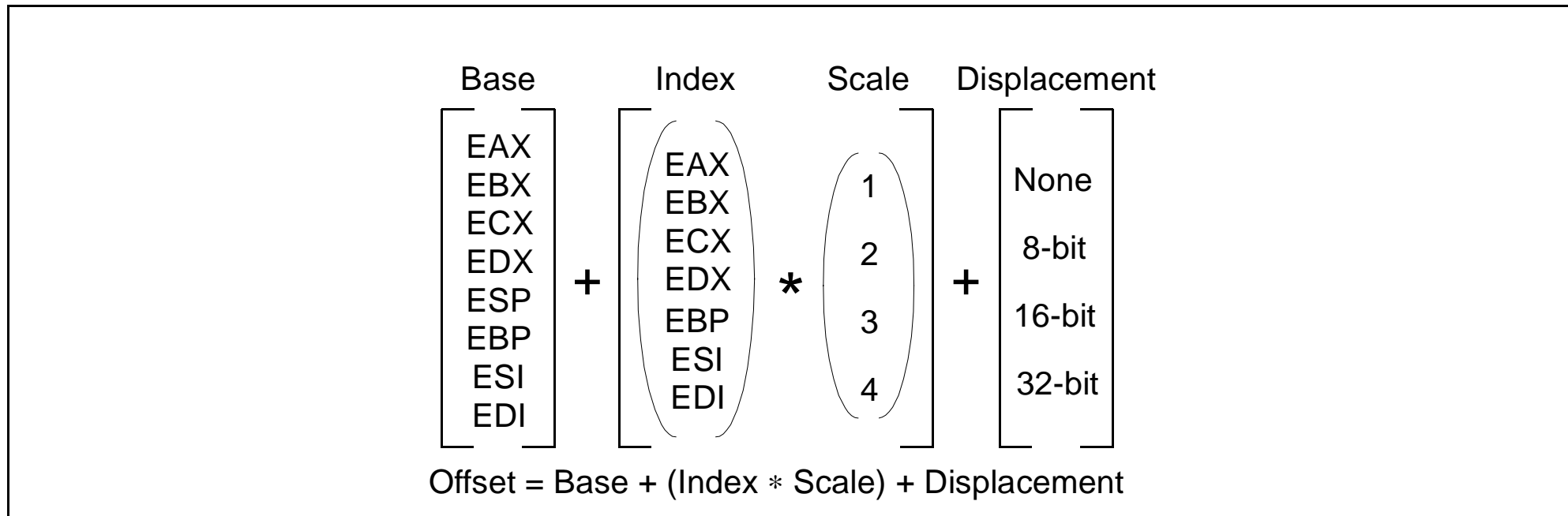


Figure 5-6. Offset (or Effective Address) Computation

Types des instructions

- Transfert : *mouvement de données, MOV*
- Arithmétique et logique : *ADD, AND...*
- Contrôle :
 - * *saut conditionnel*
 - * *appel de procédure*
- Interruption : *en général : int numéro de 0 à 255*

Format d'une instruction machine

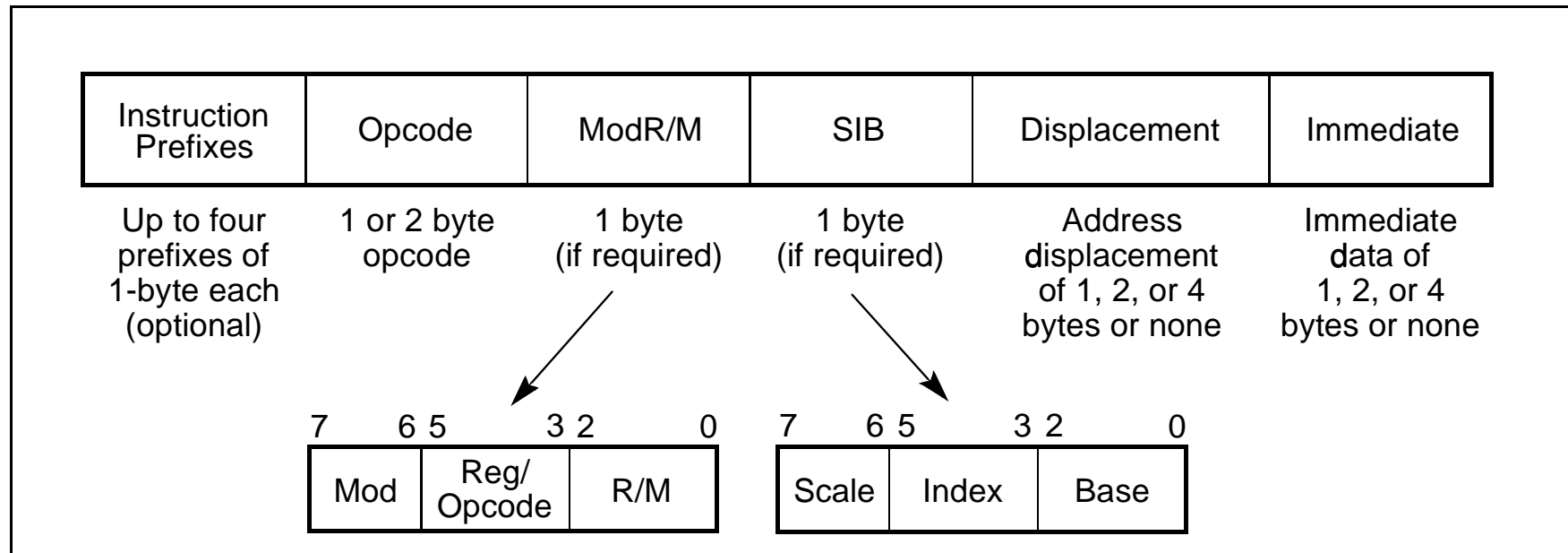


Figure 2-1. Intel Architecture Instruction Format

- longueur d'une instruction de un à 16 octets

Structure d'un programme : les segments

- A l'origine un programme comportait au moins trois segments associés à des registres (16 bits):
 - * code : CS EIP => .text
 - * données : DS => .data
 - * pile : SS ESP=> .bss
- La gestion de la mémoire est fortement liée au système d'exploitation. Elle peut être de deux types : segmentée (8086 avec msdos) ou paginée (linux), l'utilisation des registres "segment" pour l'adressage physique dépend de cette gestion (voir doc Pentium <http://developer.intel.com/design/pentium4/manuals/>).

Structure d'un programme :

```
.data
bonjour:
    .string "hello world!\n"
taille:
    .long . - bonjour
.text
.globl main
main:
    ## appel système de write ()
    movl $4, %eax           # write () system call
    movl $1,%ebx           # %ebx = 1, fd = stdout
    leal bonjour, %ecx     # %ecx ---> bojour
    movl taille, %edx      # %edx = count
    int $0x80              # execute write () system call
```

```
## appel système de exit ()
xorl %eax, %eax          # %eax = 0
incl %eax                # %eax = 1 system call _exit ()
xorl %ebx, %ebx         # %ebx = 0 normal program return code
int $0x80                # execute system call _exit ()
```

Structure d'un programme : les données

- directives :
 - `.byte` octet : 8 bits
 - `.word` mot : 16 bits
 - `.long` double mot : 32 bits
 - `.quad` 64 bits
 - `.ascii` caractère ou chaîne
 - `.string` chaîne de caractères terminée par 0

- valeurs :
 - binaire 001110
 - octal 012345670
 - décimal 1234567890
 - hexadécimal 0x123456789ABCDEF
 - décimal réel 0f 120.121e-15
 - ASCII " ... "

Programmation Assembleur

Les instructions

...

...

Instructions de transfert : mov

mov? Source, Destination

- Destination et Source : registres, variables, adresses
- Source : immédiat

movw \$36, %eax 36 → eax

.....?.....36.....
-------------	-------------

movl %ebx, %eax ebx

.....12.....	→	eax12.....
--------------	---	-----	--------------

Instructions de tranfert : mov

movl (0x8014), %eax 0x8014 0c020381 → eax 8103020c

movw (%ebx), %eax (ebx 00000012) 1201 → eax ????0112

movb \$5, (%ebx) ebx 00000012 → 0x12 05??

movw \$2, (%ebx) ebx 00000012 → 0x12 0200

movl 5(%ebx,%esi,1), %eax

(ebx 00000012 esi 00000003) 20 0510e1a2) → eax a2e11005

Instructions de transfert : XCHG, LEA

xchg Source, Destination

échange le contenu de Destination et de Source

lea Mémoire, Registre

Chargement d'une adresse effective

.data

table .fill 9 2 5; neuf 5 sur des mots de 2 octets

toto .long 0x1a ; adresse donnee + 18

lea toto, %eax eax 00000012

Alors que

movl toto, %eax, eax 0000001a

Instructions de transfert : propre au pentium

movsx Destination, Source

avec MOV la source et la destination sont de même type
ici, la source peut être d'un type plus petit

SX, signe extension, le signe de la source est conservé

movsx %al,%ebx al a3 → ebx ffffffa3

movzx Source Destination,

ZX, zero extension,

movzx %al ,ebx al a3 → ebx 000000a3

Programmation Assembleur

Instructions arithmétiques

<http://www.gnu.org/manual/>

...

L'addition

add Source, Destination

- Destination + Source → Destination
- Source : immédiat, registre, mémoire
- Destination : registre, mémoire
- opérandes : nombres signés ou non signés
- flag : OF, SF, ZF, AF, CF, et PF

L'addition (exemple)

addl %eax, %ebx

eax 00001113 + ebx 000000a9 → ebx 000011bc

eflags	OF	SF	ZF	0	AF	0	PF	1	CF
	0	0	0	0	0	0	0	1	0

eax 00001113 + ebx ffffffff → ebx 00001112

eflags	OF	SF	ZF	0	AF	0	PF	1	CF
	0	0	0	0	1	0	1	1	1

eax fff1113 + ebx a00000a9 → ebx 9fff11bc

OF	SF	ZF	0	AF	0	PF	1	CF
0	1	0	0	0	0	0	1	1

$\text{eax} \boxed{\text{ffff1113}} + \text{ebx} \boxed{\text{fffffff}} \rightarrow \text{ebx} \boxed{\text{ffff1112}}$

OF	SF	ZF	0	AF	0	PF	1	CF
0	1	0	0	1	0	1	1	1

$\text{eax} \boxed{\text{a000013}} + \text{ebx} \boxed{\text{800000a9}} \rightarrow \text{ebx} \boxed{\text{200000bc}}$

OF	SF	ZF	0	AF	0	PF	1	CF
1	0	0	0	0	0	0	1	1

La soustraction

sub Source, Destination

- Destination — Source → Destination
- Source : immédiat, registre, mémoire
- Destination : registre, mémoire
- opérandes : nombres signés ou non signés
- flag : OF, SF, ZF, AF, CF, et PF

Incrémenter - Décrémenter

`inc` Destination

`dec` Destination

- `inc` : Destination + 1 → Destination
- `dec` : Destination - 1 → Destination
- Destination : registre, mémoire
- flag : OF, SF, ZF, AF et PF, par contre CF n'est pas modifié

La multiplication

mul Source 2

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

- Source 1 * Source 2 → Destination
- Source 2 : registre, mémoire
- Source 1 : registre AL, AX, EAX
- Destination : registre AX, DX:AX, EDX:EAX

- opérandes : non signés
- **flag** : OF et CF flags sont mis à 0 si la partie haute du résultat est nulle, sinon ils sont mis à 1.
SF, ZF, AF, et PF ne sont pas définis.

La multiplication signée

- **imul** Source 2

Source 2 (reg or mem) * AL, AX, or EAX register → AX, DX:AX, or EDX:EAX.

CF et OF = 0 si le résultat tient dans la partie basse, sinon CF et OF = 1.

- **imul** Source Destination

Destination (register) * Source (register or memory or immédiat) → Destination

- **imul** Source 1 Source 2 Destination

Source 1 (register or memory) * Source 2 (immédiat) → Destination (register)

- Immediate value is sign-extended to the length of the destination operand format.

La division

`div` Diviseur

Operand Size	Dividende	Diviseur	Quotient	Reste	Max Quotient
Word/byte	AX	r/m8	AL	AH	255
Doublew/word	DX:AX	r/m16	AX	DX	65,535
Quadw/doublew	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

- $\text{Dividende} = \text{Diviser} * \text{Quotient} + \text{Reste}$

La division (suite)

- **Dividende** : AX, DX:AX, EDX:EAX
- **Diviseur**: registre ou mémoire
- **Quotient** : AL, AX ou EAX
- **Rest** : AH, DX ou EDX
- The CF, OF, SF, ZF, AF, and PF flags are undefined.
- Si **Quotient** trop gros pour AL, AX ou EAX, alors interruption

La division signée

idiv

Diviseur

Operand Size	Dividende	Diviseur	Quotient	Reste	Quotient Range
Word/byte	AX	r/m8	AL	AH	-128 to +127
Doublew/word	DX:AX	r/m16	AX	DX	-2^{15} to $2^{15} - 1$
Quadw/doublew	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31} - 1$

- $\text{Dividende} = \text{Diviseur} * \text{Quotient} + \text{Reste}$

La division signée (suite)

- Dividende, Diviseur, Quotient , Reste : comme DIV
- The CF, OF, SF, ZF, AF, and PF flags are undefined.
- The sign of the remainder is always the same as the sign of the dividend.
- absolute value of remainder $<$ absolute value of divisor.

Extension de signe

- **cbw** : extension signée de **AL** → **AX**
- **cwde** : extension signée de **AX** → **EAX**
- **cwd** : extension signée de **AX** → **DX:AX**
- **cdq** : extension signée de **EAX** → **EDX/EAX**
- CF, OF, SF, ZF, AF, and PF flags are undefined.

Autres opérateurs

- **adc** Source, Destination

Destination + Source + CF → Destination

- **sbb** Source, Destination

Destination - (Source + CF) → Destination

- **Source** : immédiat, registre, mémoire
- **Destination** : registre, mémoire
- **flag** : OF, SF, ZF, AF, CF, et PF

Opérateurs Logiques

- **and** Source, Destination

Destination *AND* Source \rightarrow Destination

- **or** Source, Destination

Destination *OR* Source \rightarrow Destination

- **xor** Source, Destination

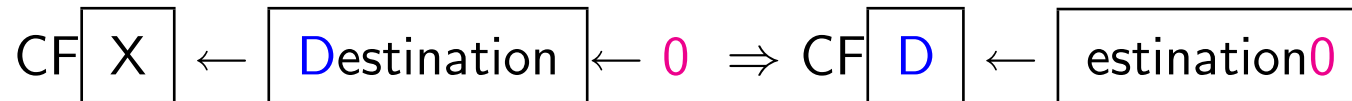
Destination *XOR* Source \rightarrow Destination

Opérateurs Logiques (suite)

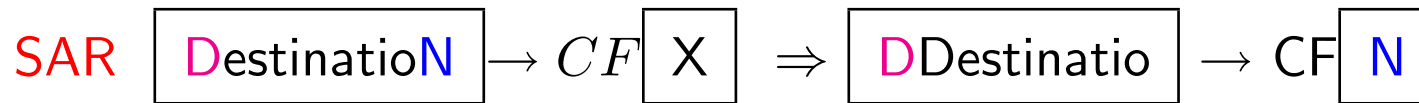
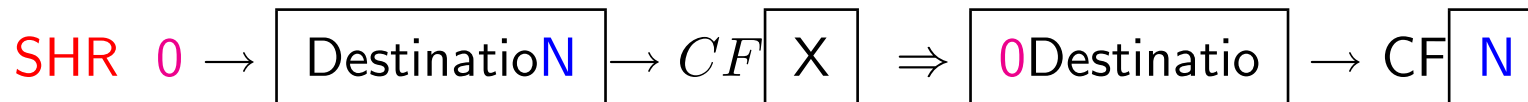
- The OF and CF flags are cleared;
- the SF, ZF, and PF flags are set according to the result.
- The state of the AF flag is undefined.
- **NOT Source** : complément à un

Décalages

- **shl** Source, Destination
- sal** Source, Destination



- **shr** Source, Destination
- sar** Source, Destination



- **Source** : CL ou immédiat, au plus 31 **Destination,** : registre ou mémoire

Rotations

rol	Source,	Destination
ror	Source,	Destination
rcl	Source,	Destination
rcr	Source,	Destination

• **ROL** : $CF \boxed{X} \leftarrow \boxed{\text{Destination}} \Rightarrow CF \boxed{D} \leftarrow \boxed{\text{estination}D}$

• **ROR** : $\boxed{\text{Destinatio}N} \rightarrow CF \boxed{X} \Rightarrow \boxed{N\text{Destinatio}} \rightarrow CF \boxed{N}$

• **RCL** : $CF \boxed{X} \leftarrow \boxed{\text{Destination}} \Rightarrow CF \boxed{D} \leftarrow \boxed{\text{estination}X}$

• **RCR** : $\boxed{D\text{estinatio}N} \rightarrow CF \boxed{X} \Rightarrow \boxed{X\text{Destinatio}} \rightarrow CF \boxed{N}$

• **Source** : CL ou immédiat, au plus 31 **Destination**, : registre ou mémoire

Les branchements

Programmation Assembleur

Jean-Claude Bajard

IUT - université Montpellier 2

Instructions de préparation

CMP Source, Destination

- Destination, : registre, mémoire
- Source : registre, mémoire , immédiat
- effectue Destination - Source : sans affecter Destination
- flag : OF, SF, ZF, AF, CF, et PF

Instructions de préparation

TEST Source, Destination

- Destination, : registre, mémoire
- Source : registre, mémoire , immédiat
- effectue Destination AND Source : sans affecter Destination
- flag : OF=0, SF, ZF, AF, CF=0, et PF

Les Sauts

Deux catégories

- Saut inconditionnel

JMP Adresse

- Saut conditionnel

Jcc Adresse

cc représente la condition

Les Sauts (suite)

- Adresse déplacement relatif à EIP
- Adresse généralement une étiquette dans le code
- Jcc ne supporte pas les sauts lointains

Exemple: si nous ne pouvons pas faire le saut suivant

```
JZ FARLABEL;
```

alors on le remplace par

```
    JNZ  BEYOND;
```

```
    JMP  FARLABEL;
```

```
BEYOND:
```

Les conditions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF and ZF)=0	Avant/non après ou égal
JAE/JNB	CF=0	Avant ou égal/non après
JB/JNAE	CF=1	Après/non avant ou égal
JBE/JNA	(CF or ZF)=1	Après ou égal/non avant
JC	CF=1	retenue
JE/JZ	ZF=1	égal/nul
JNC	CF=0	pas de retenue
JNE/JNZ	ZF=0	Non égal/non nul
JNP/JPO	PF=0	Non pair/parité impaire
JP/JPE	PF=1	Pair/parité paire
JCXZ	CX=0	Registre CX est zero
JECXZ	ECX=0	Registre ECX est zero

Les conditions (suite)

Instruction Mnemonic	Condition (Flag States)	Description
Signed Conditional Jumps		
JG/JNLE	$((SF \text{ xor } OF) \text{ or } ZF) = 0$	Plus grand/no plus petit ou égal
JGE/JNL	$(SF \text{ xor } OF) = 0$	Plus grand ou égal/non plus petit
JL/JNGE	$(SF \text{ xor } OF) = 1$	Plus petit/non plus grand ou égal
JLE/JNG	$((SF \text{ xor } OF) \text{ or } ZF) = 1$	Plus petit ou égal/non plus grand
JNO	$OF = 0$	Non overflow
JNS	$SF = 0$	Non signe (non-negatif)
JO	$OF = 1$	Overflow
JS	$SF = 1$	Signe (negatif)

Si ??? Alors ... Sinon ...

Exemple

```

...
suite 1
...
Si      A < B
      Alors
          suite 2
          ...
      Sinon
          suite 3
          ...
...
suite 4
...

```

Traduction

```

...
suite 1
...
MOV    $A,%EAX
CMP    $B,%EAX
JNL    Sinon
Alors: suite 2
...
JMP    Finsi
Sinon: suite 3
...
Finsi: suite 4
...

```

Faire...Tant que ???

Exemple

```

...
suite 1
...
Faire
...
suite 2
...
Tant que  $A \geq B$ 
...
suite 3
...

```

Traduction

```

...
suite 1
...
Faire:
...
suite 2
...
MOV    $A,%EAX
CMP    $B,%EAX
JGE    Faire
Finftq:
...
suite 3
...

```

Tant que ??? Faire ...

Exemple

```

...
suite 1
...
Tant que  A ≥ B  Faire
           ...
           suite 2
           ...
...
suite 3
...

```

Traduction

```

...
suite 1
...
Tantque:  MOV    $A,%EAX
          CMP    $B,%EAX
          JNGE   Fintq
...
suite 2
...
          JMP    Tantque
...
Fintq :   ...
          suite 3
...

```

Pour $i = n$ à m Faire ...

Exemple

```

...
suite 1
...
Pour   i = n à m (pas =t)
Faire  ...
       suite 2
       ...
...
suite 3
...

```

Traduction

```

...
suite 1
...
Pour:  MOV   $n,%ECX
       CMP   $m,%ECX
       JG   Finpour
...
suite 2
...
       ADD   $t,%ECX
Finpour: JMP   Pour
...
suite 3
...

```

Instruction LOOP

LOOP Adresse (8bits: -128 à 127)

- ECX est utilisé comme compteur
- ECX est décrémenté,
- puis Si $ECX \neq 0$ alors saut vers Adresse ,
- Sinon passage à la suite

```
...
suite 1
...
Pour    i = n à 1
Faire  ...
        suite 2
        ...
...
suite 3
...
```

Traduction

```
...
suite 1
...
MOV    $n,%ECX
Pour:  ...
        suite 2
        ...
        LOOP   Pour
Finpour : ...
        suite 3
        ...
```

Gestion de la PILE

Appel de fonction

Principe d'utilisation de la Pile

- Place mémoire réservée pour lors de l'exécution d'un programme (sous linux par défaut $2Mo$) : pour les variables locales, pour les sauvegardes d'environnement
- **ESP** pointe à l'initialisation sur le premier octet hors de la pile
- **ESP** pointe toujours sur le sommet de la zone occupée
- **EBP** pour pointer les variables stockées dans la pile

Exemple

- Réserve de quatre octets dans la pile

- ```
movl %esp,%ebp ⇒
subl $4,%esp
movw $4,-4(%ebp)
movw $12,-2(%ebp)
```

ESP → 04 00 12 00  
ESP →

# Opérations sur la pile

**PUSH** Source

- **PUSH** décrémente **ESP** de 2 ou de 4 suivant le type de **Source**
- Puis met la valeur de **Source** à l'adresse **[ESP]**

• `mov $0x23,%eax`  $\Rightarrow$  `push %eax`  $\Rightarrow$

|    |    |    |    |
|----|----|----|----|
| 00 | 00 | 00 | 23 |
|----|----|----|----|

 $\Rightarrow$ 

|                          |    |    |    |    |
|--------------------------|----|----|----|----|
| <b>esp</b> $\rightarrow$ | 23 | 00 | 00 | 00 |
| <b>esp</b> $\rightarrow$ |    |    |    |    |

`esp  $\leftarrow$  esp - 4`  
`mov %eax,(%esp)`

# Opérations sur la pile (suite)

## POP Source

- POP met la valeur pointée à l'adresse [ESP] dans Source
- Puis incrémente ESP de 2 ou de 4 suivant le type de Source

• `pop %eax`  $\Rightarrow$  `mov (%esp),%eax`  $\Rightarrow$

|     |    |    |    |    |
|-----|----|----|----|----|
| eax | 00 | 00 | 00 | 23 |
|-----|----|----|----|----|

`esp`  $\leftarrow$  `esp` + 4

ESP  $\rightarrow$

ESP  $\rightarrow$

|    |    |    |    |
|----|----|----|----|
| 23 | 00 | 00 | 00 |
|----|----|----|----|

# Appel de fonctions

# Appel d'une fonction

CALL      nomfonction

- **CALL** : sauvegarde de l'adresse de l'instruction suivant le **CALL** dans la pile

traduction :

```
push %eip
jmp nomfonction
```

# Déclaration d'une fonction

nomfonction:

```
...
suite instructions
...
ret
```

- `ret` assure le retour en dépilant l'adresse de l'instruction suivant l'appel

`RET`     `$val`

- traduction :

```
pop %eip
add $val, %esp ;(si arguments)
```

# Retour sécurisé

- Pour être certain de dépiler l'adresse de l'instruction suivant le `call`

- En début de fonction :  
`push %ebp`  
`mov %esp,%ebp`

- En fin de fonction :  
`leave`  
`ret`

- traduction de `LEAVE` :

```
mov %ebp, %esp
pop %ebp
```

# Sauvegarde de l'environnement

- **Avant l'appel** empiler le contenu de tous les registres risquant d'être modifiés dans la procédure

**PUSHAD** ;empile : EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

- **Avant le retour** dépiler pour restituer le contenu de tous les registres empilés lors de l'appel POPAD Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX

**POPAD** ; dépile : EDI, ESI, EBP, EBX, EDX, ECX, and EAX

Directives :

- `.globl` déclaration pour l'édition de lien
- `.type nomfonction,@function` déclaration du type

Remarque:

- le corps du programme (`main:`) peut lui aussi être déclaré comme fonction. C'est même conseillé.

## Utilitaire:

- `readelf` pour lire les différentes informations liées au format **ELF**: par exemple `readelf -s a.out` pour lire la table des symboles.

## Construction de l'exécutable:

- avec `gcc`  
`gcc toto.s -o toto`
- avec `as` et `ld`  
`as toto.s -o toto.o`  
`ld -e main toto.o -o toto`

## résumé

|                     |                             |                           |                                    |
|---------------------|-----------------------------|---------------------------|------------------------------------|
| <code>.globl</code> | <code>main</code>           | <code>.globl</code>       | <code>nomfonction</code>           |
| <code>.type</code>  | <code>main,@function</code> | <code>.type</code>        | <code>nomfonction,@function</code> |
| <code>main:</code>  |                             | <code>nomfonction:</code> |                                    |
| <code>push</code>   | <code>%ebp</code>           | <code>push</code>         | <code>%ebp</code>                  |
| <code>mov</code>    | <code>%esp,%ebp</code>      | <code>mov</code>          | <code>%esp,%ebp</code>             |
| <code>...</code>    |                             | <code>...</code>          |                                    |
| <code>pushad</code> |                             | <code>leave</code>        |                                    |
| <code>call</code>   | <code>nomfonction</code>    | <code>ret</code>          |                                    |
| <code>popad</code>  |                             |                           |                                    |
| <code>...</code>    |                             |                           |                                    |
| <code>leave</code>  |                             |                           |                                    |
| <code>ret</code>    |                             |                           |                                    |

# Passage des arguments

Principe d'utilisation de la pile lors de l'appel:

- empiler les arguments dans l'ordre inverse: argN,...,arg2,arg1  
push argN  
...  
push arg2  
push arg1
- faire l'appel  
call nomfonction
- au retour rétablir le pointeur de pile  
ret taillearg

## Passage des arguments: suite

Principe d'utilisation de la pile dans la fonction:

- sauvegarde du pointeur de pile:  
`push %ebp`  
`mov %esp,%ebp`
- reservation de l'espace nécessaire pour les variables locales:  
`sub $val,%esp`
- localisation par rapport au registre `%ebp`:  
`+8(%ebp)`    pointe sur `arg1`  
`(%ebp)`      pointe `%esp` d'origine  
`-4(%ebp)`    pointe sur `varloc1`
- **leave** replace le pointeur de pile sur l'appel (`%eip` suivant)

# Variables Locales

- Réserve d'espace dans la pile pour les variables locales :

