MIC

COURS

S1 : Ouvrons un PC

Carte mère

Permet l'interconnection des composants d'un ordinateur.
Dans CPU il y a de la mémoire (cache interne pas RAM) plus une mémoire est proche du CPU, plus elle coûte chère, il y a L1/premier niveau(dans CPU), L2/second niveau(dans boitier contenant CPU), L3 (en dehors CPU/boitié).

Chipset : c'est un composant éléctronique qui contient plusieurs composants, il est souder à la carte mère. Le processeur est en dehors du chipset. Il contient le Northbridge et le Southbridge.

Rôle carte mère : essentielement composé de circuits imprimés et de ports de connexion, par le biais desquels elle assure la connexion de tous les composants et périphérique propres à un micro-ordinateur (HDD, RAM)
…

Structure de la carte mère : connecteurs électrique
Support proco (socket)
…

Carte multiprocesseur peut acceuillir plusieurs processeur physiquement distinct.
Principalement utilisé dans serveurs ou superordinateurs.

Mémoire cache (antémémoire/mémoire tampon) : très rapide, + que la RAM, contient une copie des information de la RAM.

Processeur

Exécute des instructions qui font partie d'un programme.
Un microprocesseur est un processeur dont tous les composants sont regroupés dans un même boitier
Communique avec les autres composants avec ses pattes (pins).
Chaque pate à une utilité et un nom.

Di	reliées au bus de données
Ai	reliées au bus d'adresses
INTR	pour les interuptions
CLK	horloge
RESET	reset du processeur
Vcc Vss	alimentation

Les processeurs se distinguent par

- Leur architecture interne
- Les instructions comprises
- La taille des registres (32/64 bits)
- Leur fréquence
- Le nombre de cœurs

Une famille de processeurs regroupe les processeurs qui partagent un même jeu d'instructions.
Quelques dates de la famille x86

- 1978 Intel 8086
- 1985 Intel 80386, AMD Am386
- 1993 Intel Pentium (Pro)
- 2000 Intel Pentium 4, Athlon
- 2008 Intel Core i7

Ils ont une rétrocompatibilité.

Processeur de type RISC et CISC

	RISC	CISC
Signification	Reduced Instruction Set Computer	Complex Instruction Set Computer
Nb d'instructions	Plus	Moins
Code	Plus gros	Moins gros
Processeur	Moins complexe	Plus complexe

Intel fais partie des CISC
Les x86 sont des CISC.

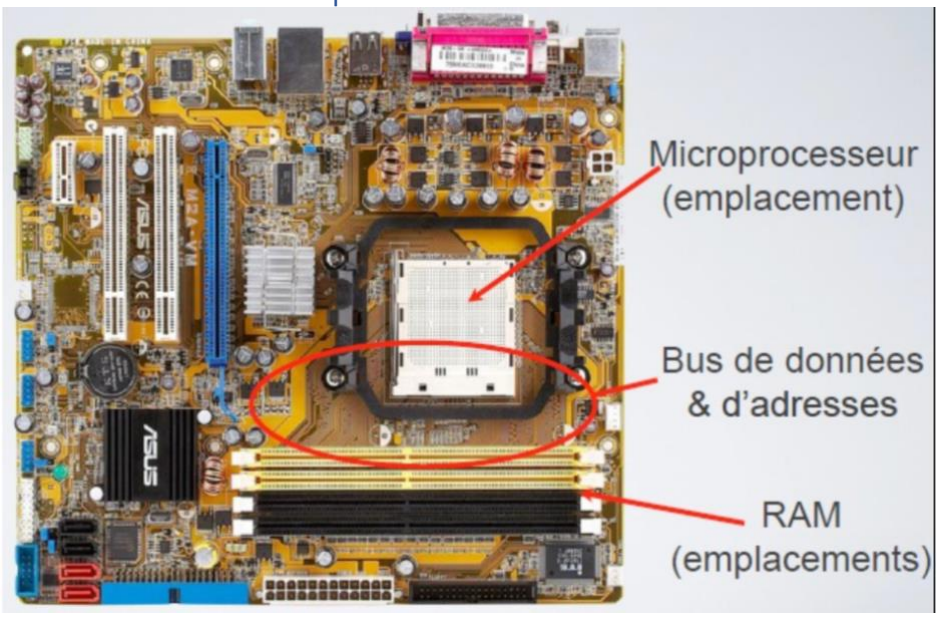
Mémoire

La RAM est la mémoire vive qui contient les processus (code + données)

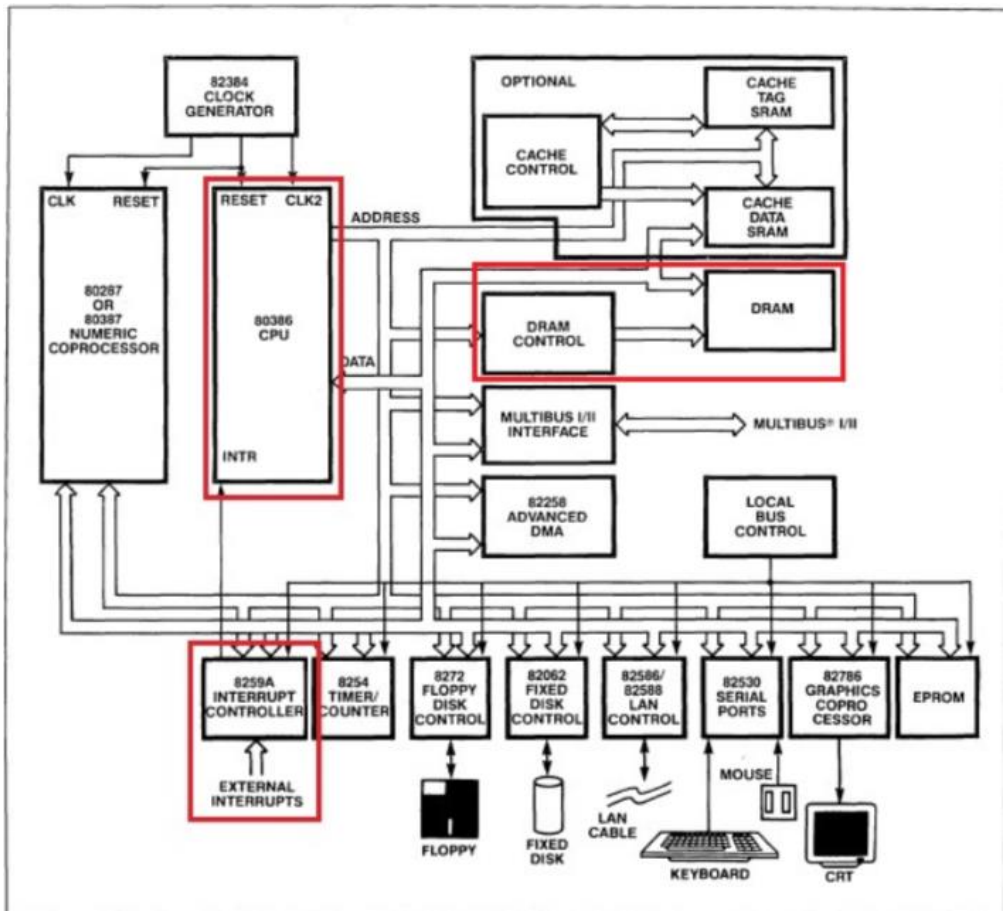
Dans schéma, le décodeur regarde si oui ou non il peut accéder à la donnée.

S2 : Microprocesseurs

Fonctionnement microprocesseur

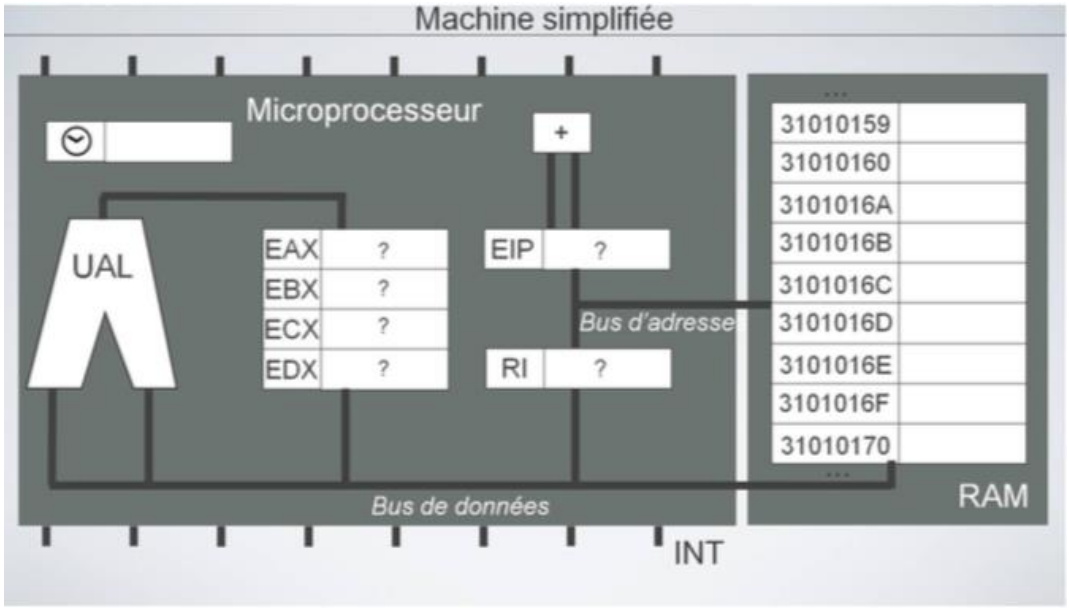


Shéma Intel 80386



La patte INTR du CPU est direct connecté au controleur d'interuption.

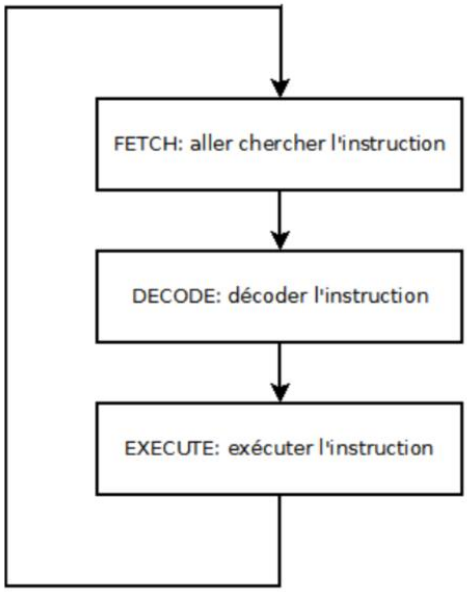
Cycle du processeur



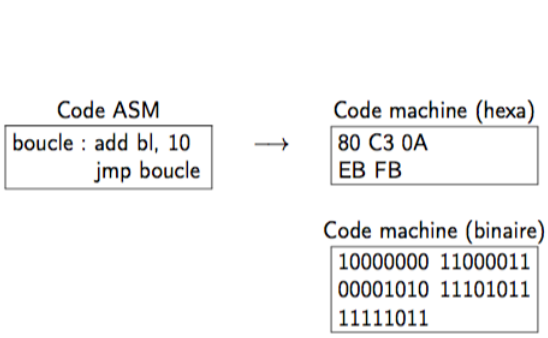
RIP : pointeur d'instruction.
RI : décode.
Registres généraux.
Bus.

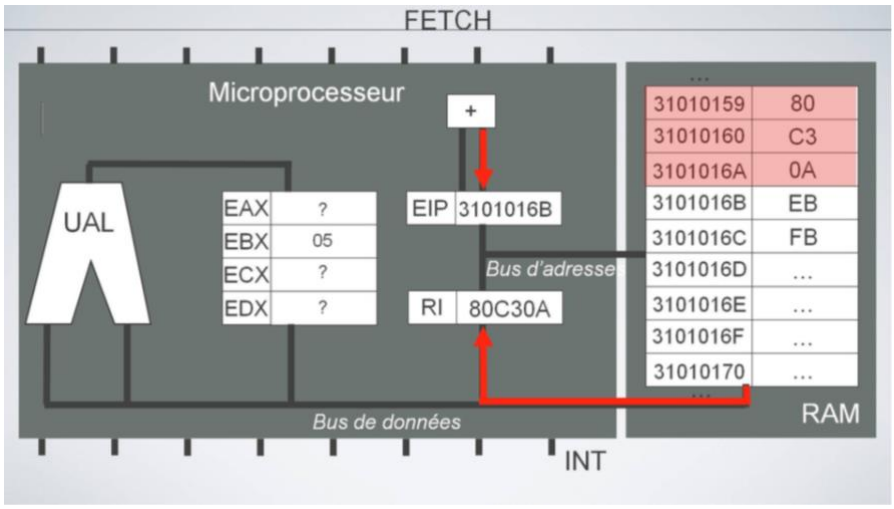
UAL : Unité Arithmétique et Logique.
Horloge.
Incrémenteur.

Cycle du processeur : Fetch puis décode et exécute en boucle.



Shéma, la 1ere exécution, exécute l'addition de 10 puis comme c'est une boucle infini et qu'on est à la fin, la 2ème exécution c'est le jump.





S3 : Interruptions

Exemple : mov ax,0X529 en mémoire, il commence par entré « mov » puis il lit de droite à gauche pour enregistrer dans la RAM, donc 29 puis 05.

Le cycle du processeur exécute tout un processus en une fois. Que ce passe-t-il si :

- Un évènement extérieur arrive ? (Frappe au clavier, click souris, …)
- Si erreur d'exécution ? (Instruction inconnue, division par 0...)
- Du dialogue avec le système ? (Appel système, scheduling de processus...)

Le polling/Scrutation

Se demande tout le temps s'il y a un caractère au clavier.
Sert pour détecter une frappe au clavier sans interruptions. Problème : attente active
Exemple : traitement de texte.

Interruptions

C'est quoi :

- Arrêt automatique du processus en cours.
- Exécution d'une « routine de traitement d'interruption » (interrupt handler)
- En général, reprise ultérieure du programme interrompu.

Types d'interruptions :

- Interruptions matérielles : Provoquées par un matériel extérieur (frappe clavier..)
- Exceptions : Provoquées par un programme, généralement suite à une erreur(÷0)
- Appels-système : Provoquées par un programme, pour demander un service à l'O.S. (demande de lecture/modification sur fichier, demande droit d'accès …)
  Exemple : syscall qui sert a arrêter le programme.

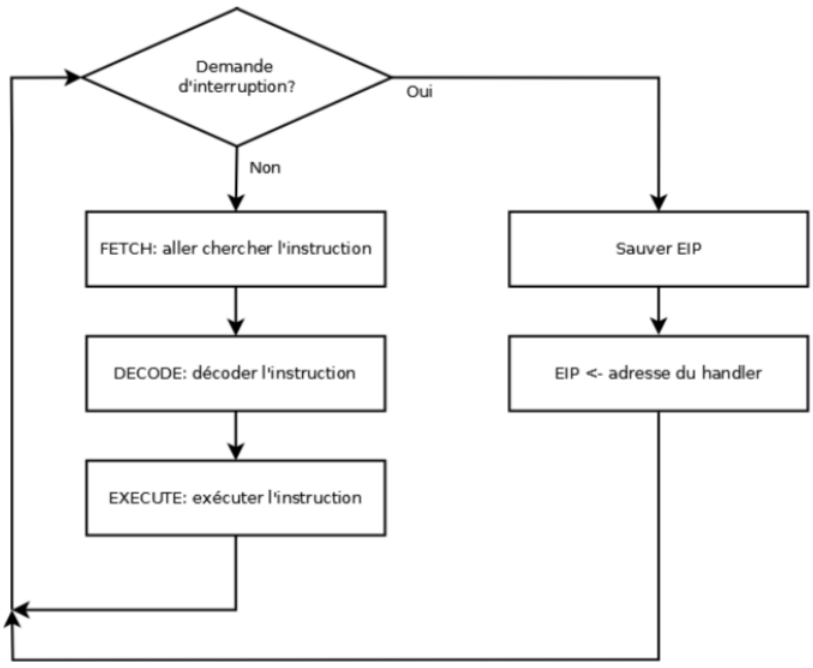
Chaque interruption à un n°, qui est un vecteur.

- 256 types différents.
- 0-31 : réservés pour des exceptions
  - 0 : division par 0
  - 6 : opcode non-défini ...
- 32-255 :
  - Définis par l'OS
  - Programmés dans le PIC (interruptions matérielles)

-F dwarf c'est pour le debogeur

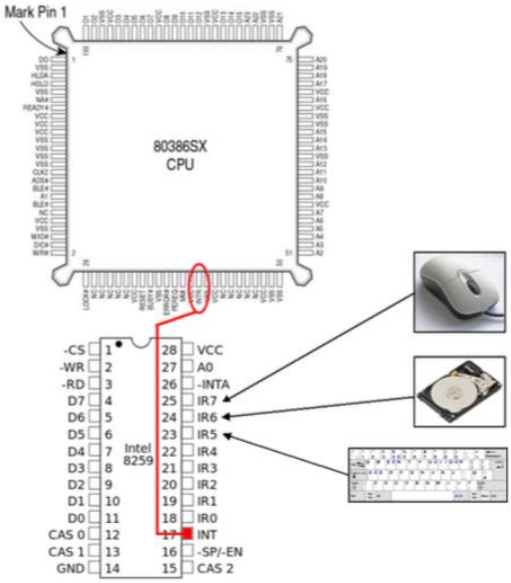
Si on ne met pas syscall « erreur de segmentation » car il dépasse notre espace réservé en RAM.

Cycle du processeur avec interruption :

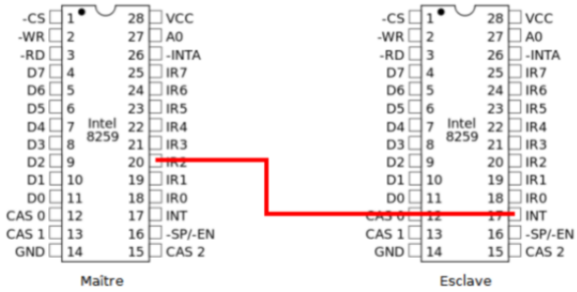


MIC

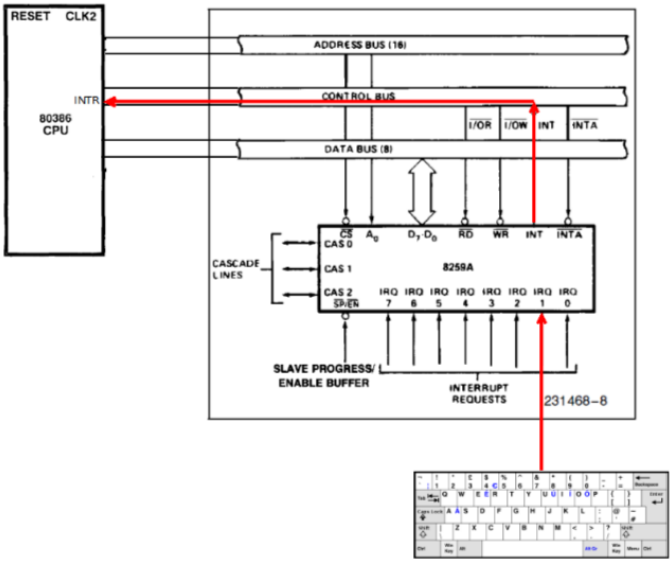
Sur le proco il y a une broche pour les interruptions.
INTR = INTerrupt Request
Signale au processeur l'arrivée d'une interruption matérielle
Cette broche est unique. Comme il y a bcp d'interruptions possible, c'est PIC qui s'en gère puis qui prévient le CPU. PIC est relié aux différents périphériques (clavier, souris, …) sur les borne IR(Q) et PIC est aussi branché à la broche du CPU. Envoie les demandes d'interruptions une à une au CPU via la broche INT. PIC est programmable pour donner des priorités différentes à chaque périphérique
PIC = Programmable Interrupt Controller



2 PIC en cascades permet d'augmenter le nombre d'IRQ



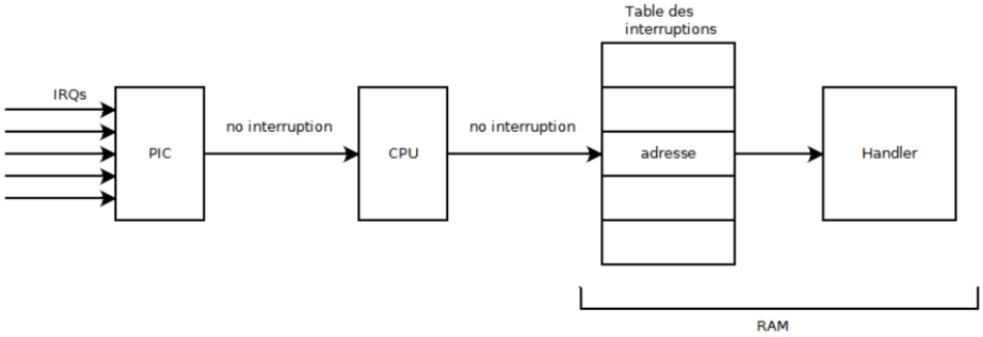
Le PIC reçois l'interruption, celui-ci, via le bus de contrôle envoie l'interruption sur INTR.



CPU envoie un accusé de réception via INTA sur le bus de contrôle.
Puis le n° d'interruption est envoyé via le bus de données au CPU

Table d'interruption :

- Contient les adresses des handlers (ce qu'il doit faire quand cette intr apparait) d'interruptions
- Indexée par le numéro d'interruption
- Dans la RAM
- Gérée par l'OS
- Détails : leçon ultérieure et cours de Systèmes



Handler : code de ce que dois faire l'interruption. Exemple : syscall.

Structure d'un handler (monoprogrammation)

1. Sauver registres utilisés (sur pile)
2. Traiter l'interruption
3. Restaurer registres utilisés (àpd pile)
4. IRET (quand il y a appel système, … on passe de mode utilisateur à privilégier et IRET rechange à la fin du mode privilégié a mode utilisateur)

IRET :

- Exécutée en fin de handler d'interruption
- Restaure automatiquement l'ancien EIP
- Permet de revenir au programme interrompu

Flag IF
Si IF=0, le CPU est non-interruptible
SI IF=1, le CPU est interruptible
CLI(clear) : instruction pour mettre IF à 0
STI(store) : instruction pour mettre IF à 1
CLI/STI : utilisés par l'OS
Ce n'est pas pour utilisateur, ne compile/exécute pas.

S4 : Mode réel et protégé

Mode réel et mode protégé son des modes de fonctionnement de notre ordinateur.
Avant le processeur Intel 80286

- Chaque programme avait accès à toute la mémoire centrale
- Le système MS-DOS travaillait dans ce mode
- Risques de sécurité :
  - Un programme peut modifier les données d'un autre programme
  - Un programme peut même modifier les données de l'OS.
- Solution : introduction du « mode de fonctionnement protégé » à partir du processeur 80286

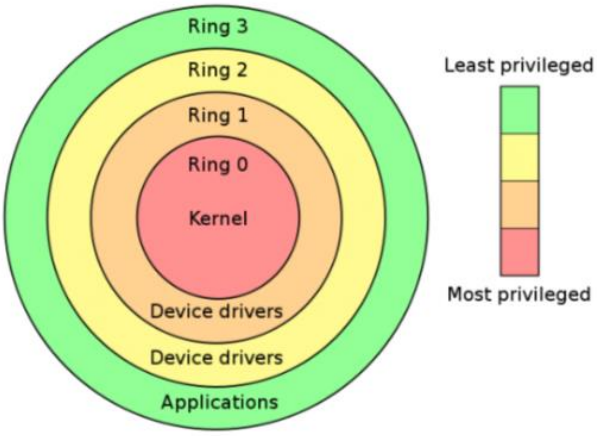
Protégé

- Disponible depuis le processeur 80286
- Souvent abrégé en « « mode protégé
- Ce mode offre des protections à deux niveaux :
  - Un programme ne peut pas accéder à toute la mémoire de l'ordinateur, on va segmenter, baliser.
  - Un programme ne peut pas toujours utiliser toutes les instructions du processeur
- La quasi-totalité des systèmes informatiques actuels fonctionnent en mode protégé

Anneaux de protection

À tout moment, le processeur travaille dans un des quatre niveaux de protection
A certains moments-clé, le processeur change de niveau de protection

Niveau	Description
0	Les programmes peuvent employer toutes les instructions du processeur et accéder à toute la mémoire centrale (employé pour le code de l'OS sous Windows et Linux)
1	Les programmes ne peuvent employer qu'un nombre réduit d'instructions, et n'ont pas accès à toute la mémoire (non employé sous Windows et Linux)
2	Les programmes ne peuvent employer qu'un nombre réduit d'instructions, et n'ont pas accès à toute la mémoire (non employé sous Windows et Linux)
3	Les programmes ne peuvent employer qu'un nombre réduit d'instructions, et n'ont pas accès à toute la mémoire (employé par les programmes utilisateurs sous Windows et Linux)



0 : le poco peut tout faire, peut accéder à toutes la mémoire.
3 : c'est les programmes que nous créons, on ne peut aller que dans la partie de mémoire qui nous es réservé.

Interdis en mode protégé, CLI (clear interrupt)

- L'instruction CLI sert à suspendre les interruptions en mettant le ag IF à 0
- Elle n'est permise qu'en ring 0, donc seul l'OS peut l'employer
- L'exemple suivant montre un essai d'utilisation de CLI dans un programme utilisateur

Si dans un code on met juste « cli » comme on n'est pas en ring0, on a une erreur de segmentation.

Mode réel

En cmd en émule le mode réel

- Malgré l'introduction du mode protégé, les processeurs 80286 et suivants peuvent encore travailler dans le mode précédent, non-protégé : il s'agit du mode réel
- Ce mode permet notamment d'exécuter des vieux OS comme MS-DOS, qui n'étaient pas prévus pour le mode protégé
- En réalité, lors de l'allumage de l'ordinateur, le processeur commence toujours à fonctionner en mode réel, puis l'OS le fait passer en mode protégé
- En mode réel, la taille totale de la mémoire est limitée à 1 Mb (220 bytes), et tous les programmes peuvent y écrire partout sans protection !

Gestion de la mémoire en mode protégé

Emploie 3 types d'adresses :

1. Les adresses logiques : les adresses employées dans vos programmes assembleur, et vues par le processeur. Se fais segmenter, devient linéaire.
2. Les adresses linéaires : issues d'une traduction des adresses logiques au travers du processus de segmentation. Se fait paginer, devient physique.
3. Les adresses physiques : véritables adresses de la mémoire centrale.

Segmentation : segmente la mémoire, les données se trouve à un endroit, les variable à un autre.

Déclaration de variable

```
section .data
var db 25       sur 8bits
var2 dw 25      sur 16bits
var3 dd 25      sur 32bits
var4 dq 25      sur 64bits
```

Adresse logiques

- Les adresses logiques sont celles employées dans vos programmes assembleur, et visibles notamment dans Kdbg
- Expérience : exécutez deux fois le même programme en même temps sur linux1 et comparez les adresses grâce à Kdbg
  - Les deux programmes emploient les mêmes adresses !
  - Or deux programmes ne peuvent occuper simultanément le même espace en mémoire !
  - En réalité, les adresses employées par votre programme (adresses logiques) ne sont pas les véritables adresses physiques employées par la mémoire centrale
  - Un mécanisme de traduction sera nécessaire pour traduire ces adresses logiques en adresses physiques

mov

mov rax,[rax], entre crochet c'est l'adresse. Met la valeur qui est à cette adresse dans
Exemple : l'instruction MOV [10], AX place la valeur de AX en mémoire centrale à l'adresse logique 10
Attention, ici 10 est bien une adresse logique, et non la véritable adresse physique à laquelle sera placé le contenu de AX
Ainsi, deux programmes différents peuvent effectuer un MOV [10], AX sans se marcher sur les pieds, car l'OS traduira leur deux adresses « 10 » vers des adresses physiques différentes.

Segmentation

La mémoire est protégée grâce à la segmentation

- En mode protégé, la mémoire d'un programme est divisée en morceaux appelés des segments
- Chaque segment d'un programme peut être situé à un endroit différent de la mémoire centrale
- La localisation précise de chaque segment en mémoire est stockée dans une table appelée table des segments
- Cette table contient l'adresse et la taille de chaque segment, ainsi qu'une information disant à quel programme appartient le segment
- Les informations de la table des segments permettent d'empêcher un programme d'accéder aux segments d'un autre programme (protection de la mémoire)

Segment typique d'un programme

Segment de code : contient les instructions du programme
Segment de données : contient les variables globales du programme (voir labo)
Segment de pile : contient les variables locales du programme

Registres de segments

- Le registre CS (code selector) permet de retrouver l'adresse du segment de code dans la table des segments
- Le registre DS (data selector) permet de retrouver l'adresse du segment de données dans la table des segments
- Le registre SS (stack selector) permet de retrouver l'adresse du segment de pile dans la table des segments

Ces registres indiquent en réalité où dans la table des segments on doit aller chercher l'adresse du segment correspondant. D'autres registres de segment existent (ES,FS,GS).

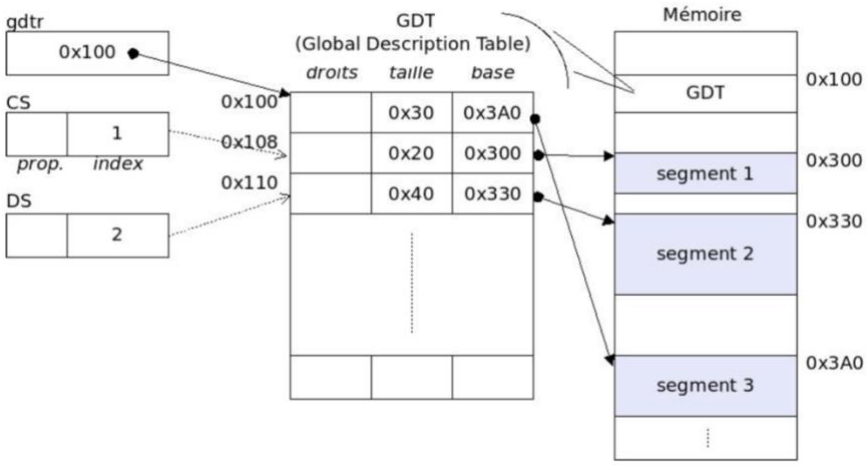
Traduction de d'une adresse logique en linéaire

Les adresses logiques sont en réalité des décalages (anglais offset) par rapport au début de leur segment, il faut donc rajouter à chaque adresse logique l'adresse du début du segment correspondant
Adresse linéaire = adresse du début du segment + adresse logique

Exemple :

- `MOV [10], AX` signifie « mettre le contenu de AX à l'adresse 10 *du segment de données* »
- On obtient l'adresse de début du segment de données grâce au registre DS et à la table des segments
- On rajoute cette adresse à 10 pour obtenir l'adresse linéaire

La segmentation en mode protégé

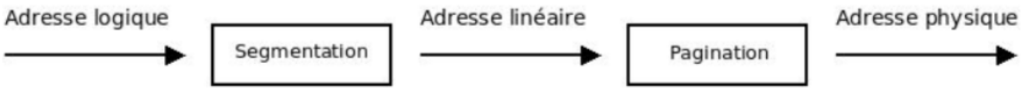


Pour obtenir l'adresse, je dois passer par la table d'index celle-ci contient les droit, la taille et le début du segment de donnée (base).
Gdtr a l'adresse de la table d'index.
La table d'index commence à 0.
On a accès à CS.

Pagination

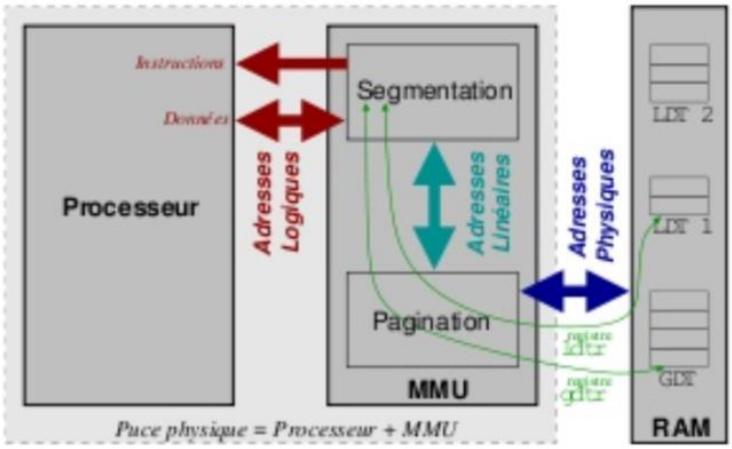
- La pagination est un deuxième mécanisme de gestion de la mémoire offerts par le processeur Intel 80386
- Il consiste à découper les programmes en morceaux de taille égale appelés des pages
- Chaque page du programme sera ensuite placée en mémoire à un endroit différent
- Ainsi, un même segment peut être divisé en plusieurs pages, situées à des endroits différents de la mémoire
- Les adresses linéaires doivent donc elles-mêmes être traduites pour retrouver les adresses physiques !
- La pagination (contrairement à la segmentation) peut être désactivée dans le processeur 80386
- Si on n'emploie pas de pagination, les adresses linéaires et les adresses physiques sont égales
- Nous ne verrons pas en détail cette année le fonctionnement de la pagination
- La pagination est fortement utilisée par les systèmes d'exploitation modernes, dont Linux

Étapes de traduction des adresses



Le processeur et la traduction des adresses

- Le processeur n'utilise que des adresses logiques dans ces registres (EIP, ...)
- Ces adresses doivent donc être traduites en adresses physiques (segmentation, pagination) avant d'accéder véritablement à la mémoire
- Cette traduction d'adresse se fait grâce à une puce particulière appelée MMU (Memory Management Unit, i.e. Unité de Gestion de la Mémoire)



Mode réel et segmentation

Segmentation mais très simplifié, pas e pagination, donne directement des adresses physiques
Adresse physique = adresse de début du segment*16 + offset
L'adresse de début du segment et l'o set sont sur 16 bits, ce qui donne une adresse totale sur 20 bits :

Segment :	A 2 5 F	sur 16 bits
Offset : +	5 2 A 6	sur 16 bits
Adresse :	A 7 8 9 6	sur 20 bits

Lors d'une multiplication par 16, on décale de 4bits à gauche.
Avant la segmentation servait pour avoir des adresses sur 20bits.

- En mode réel, la segmentation ne contient aucun mécanisme de protection de la mémoire
- Tout programme peut accéder à n'importe quelle adresse physique en choisissant un segment et un offset qui mène à cette adresse
- Il n'y a pas de table des segments
- Les registres CS, DS, SS sont associés au code, aux données, et à la pile, comme en mode protégé
- Les autres registres de segment (ES,FS,GS) sont utilisables librement (cf. page suivante)
- Exemple d'un programme en mode réel qui dé nit son propre segment grâce au registre ES, la segmentation ne contient aucun mécanisme de protection de la mémoire
- Ce programme ne fonctionne pas (segmentation fault) en mode protégé, car il faut être en ring 0 pour accéder aux adresses 0xB8000

```
mov AX, 0xB800          ; On donne la valeur B800
mov ES, AX              ; au registre de segment ES
mov AL , 'h'            ; on donne une valeur
mov AH,10010111b        ; arbitraire a AX
mov [ES:0xA0] , AX      ; que l'on place aux positions
                        ; B80A0 et B80A1 en memoire.
```

B8000+A0

Mode interruptions

Rappel : la table des interruptions est la table disant à quelle adresse se trouve la routine de gestion de l'interruption n° i
Cette table se trouve dans la mémoire centrale, dans une zone réservée appartenant au système d'exploitation
Selon qu'on est en mode réel ou protégé, cette table est gérée de façon différente

Mode réel et interruptions

- En mode réel, les adresses sont données par une paire (segment : offset), où segment et offset font 2 bytes chacun (donc 4 bytes en tout)
- En mode réel, la table des interruptions est à l'adresse fixe 0000:0000 (i.e. l'adresse 0 dans le segment 0).
- L'entrée i de la table contient l'adresse de la routine de gestion de l'interruption n° i
- L'adresse de la routine de gestion de l'interruption i se trouve donc à l'adresse $4*i$

Mode protégé et interruptions pour 32bits

- En mode protégé, la table des interruptions est à l'adresse donnée par le registre IDTR
- L'entrée i de la table contient l'adresse de la routine de gestion de l'interruption n° i
- En mode protégé, les adresses (segment : offset) ont 2 octets pour le segment et 4 octets pout l'offset (6 octets en tout) en 32 bits mais pour 64bits, le segment fait 2 bytes pour segment et 8bytes pour l'offset.
- Chaque entrée de la table des interruptions contient les 6 bytes de l'adresse de la routine de gestion, ainsi que 2 bytes supplémentaires contenant d'autres informations (donc 8 octets en tout par entrée de la table) en 64bits il y a aussi 2 octets supplémentaire donc 12 au total.
- L'adresse de la routine de gestion de l'interruption i se trouve donc à l'adresse [IDTR]+8*i pour 32 bites et pour 64bits se trouve à l'adresse [IDTR]+12*i

On ne sait pas où est la table de segmentation.

Le code d'un utilisateur est exécuté dans le ring3
Le code d'une interruption est exécuté dans le ring0
Une interruption fais basculer le processeur en ring0 et IRET le fait basculer dans l'ancien ring(mode).

Lors d'une interruption,
Le processeur termine l'instruction en cour ;
Place le contenu du registre RIP au sommet de la pile
Remplace RIP par une valeur situé en mémoire dans une table à l'index donnée par le n° d'interruption
Le processeur bascule en ring0 s'il est en mode protégé pour pouvoir exécuter la routine d'interruption.

Lors de l'exécution de l'instruction IRET,
Le proco récupère RIP sur pile
Le proco bascule dans l'ancien ring s'il est en mode protégé.

S5 : Assembleur

Le terme « assembleur » désigne :

- Un langage de programmation
- Le compilateur de ce langage

Langage

- Langage de « seconde génération » :
  - « Première génération » = code machine binaire
- Assembleur = forme de code machine humainement lisible :
  - Nommage explicite des instructions (mnémoniques : MOV, ADD, ...)
  - Nommage explicite des adresses du code (labels : main, if, else, ...)
  - Nommage explicite des adresses de données (variables : var DB 5, ...)

Compilateur

- Assembleur = compilateur du langage Assembleur
- « Assemblage » = compilation
- But : transformer le fichier source en code machine binaire
  - Mnémoniques traduites en instructions binaires (opcodes, voir chap. 7)
  - Labels traduits en adresses (ou décalages)
  - Variables traduites en adresses

Assemblage et éditions des liens

Utilisation possible de plusieurs fichiers sources pour produire un exécutable
Phase 1 : compilation de chaque fichier source =⇒fichiers objets
Phase 2 : édition des liens entre fichiers objets =⇒fichier exécutable

Pourquoi utiliser l'assembleur ?

- Pour des raisons pédagogique : permet de simuler et comprendre le fonctionnement interne d'un microprocesseur
- Plus facile à lire et à comprendre pour un humain car dans se code on retrouve :
  - Du texte
  - Des mnémoniques
- L'utilisation des étiquettes pour représenter par exemple des noms de variables, les noms de fonctions ,etc … permet de ne pas devoir réaliser des calculs d'adresses et de recalculer les adresses en cas e modification du code.

A la place des label on peut à chaque fois écrire l'adresse mais si qqchose change, on doit changer l'adresse, de plus il faut la calculer.

- L'assembleur connaît les formats standards de fichiers objet(comme elf), on ne doit pas s'en préoccuper à a rédaction du code mais juste éventuellement utiliser des directives et des options d'assemblages
- L'assembleur (au niveau de la traduction) est plus proche du langage machine, c'est un langage de bas niveau. L'effort de traduction en langage machine est donc très faible.

Les obstacles à l'utilisation de l'assembleur

- Comme il est le reflet du jeu d'instruction du processeur, il y a autant d'assembleurs que de processeur
- Quand on change de processeur …
- Clarté du code : le code ASM est moins lisible qu'un langage de plus haut niveau (C, C++, Java, . . .)
- Temps de développement : plus long car programmation plus complexe qu'en langages de haut niveau
- Portabilité : moins portable car jeu d'instruction différent pour chaque processeur

Dialectes utilisés en assembleur

Deux dialectes différents pour l'assembleur x86

- AT&T
- Intel

Différence :

- Ordre des opérateur
- Spécifications de taille

Exemple :

Intel	AT&T
mov eax, 5	movl $5, %eax
mov ax, 5	movw $5, %ax
mov al, 5	movb $5, %al

Inclure un code assembleur dans un code C

- Certains compilateurs permettent d'insérer du ode ASM à l'intéreur d'un programme en langage de haut niveau
- Exemple en C avec le compilateur gcc (syntaxe AT&T)

Performance

- On dit souvent que les programmes en assembleur sont plus rapides
- Pas toujours vrai :
  - Dépend fortement des capacités du programmeur ASM
  - Les compilateurs actuels permettent de fortement optimiser le code donne souvent code tout aussi rapide que l'assembleur
  - Exemple d'optimisation en C avec le compilateur gcc : gcc -o2 fichier_source.c

Applications de l'assembleur

- Code système : Drivers, handlers d'interruptions, BIOS, . . . Certaines portions du kernel Linux sont encore en assembleur
- Micro-contrôleurs et systèmes embarqués : Mais de plus en plus remplacé par le langage C
- Applications Multimedia : Applications nécessitant de lourds calculs (graphiques pour jeux, etc. . .) Nécessité de tirer parti des instructions les plus spécialisées du processeur (ex. SIMD, voir chap. 11)
- Virus

L'assembleur aujourd'hui

- N'est plus employé pour le développement d'applications complètes
- Employé pour certaines portions de code très précises nécessitant une optimisation fine
- Intérêt pédagogique : Permet de mieux comprendre le fonctionnement du processeur

S6 : Mode d'addressage

Adresser les données

De nombreuses instructions manipulent des données

- transferts mémoire-regitres : MOV AX, 3
- calculs : ADD EAX, EBX
- tests : BTAX,2 ...

Il faut pouvoir indiquer où se trouve / où mettre chaque donnée. C'est ce qu'on appelle les modes d'adressage.

Modes de base

- Immédiat : la donnée est directement dans l'instruction. Exemple : MOV AX,42
- Registre : la donnée est (doit être placée) dans un registre. Exemple : MOV AX, BX
- Direct : la donnée est (doit être placée) à l'adresse donnée dans l'instruction. Exemple : MOV AX, [0xB8A0] ou encore MOV [0xB8A0], AX

Pièges

Attention à ne pas confondre une adresse et ce qu'elle contient. **Comme on est en 64bits, les adresse font 64bits.** Un label est un nom symbolique pour une adresse.

```
MOV EAX, 0xB8A0   ; immédiat. EAX reçoit la valeur 0xB8A0.
MOV EAX, [0xB8A0] ; direct. EAX reçoit la valeur (4 bytes) à l'adresse 0xB8A0.
MOV AX, [0xB8A0]  ; direct. AX reçoit la valeur (2 bytes) à l'adresse 0xB8A0.
MOV AL, [0xB8A0]  ; direct. AL reçoit la valeur (1 byte) à l'adresse 0xB8A0.
```

```
MOV EAX, brol     ; immédiat. On met dans EAX l'adresse brol.
MOV EAX, [brol]   ; direct. EAX reçoit la valeur (4 bytes) à l'adresse brol.
MOV AX, [brol]    ; direct. AX reçoit la valeur (2 bytes) à l'adresse brol.
MOV AL, [brol]    ; direct. AL reçoit la valeur (1 byte) à l'adresse brol.
```

D'ailleurs

```
MOV AL, 0xB8A0    ; Ne compile pas ! AL trop petit pour y mettre la valeur
MOV 0xB8A0, AL    ; Ne compile pas ! Pas d'immédiat à gauche.
```

D'ailleurs

```
MOV [brol], AX    ; on met le contenu de AX à l'adresse brol
MOV brol, AX      ; Ne compile pas ! On ne peut pas modifier un label.
```

Utilisation

Ces trois modes permettent de traduire les instructions simples des langages de haut niveau.

1. Valable pour des variables globales. En pratique, a et b sont probablement des variables locales ; elles seront dès lors stockées dans une pile, ce qui modifie un peu les instructions.

Par exemple

```
b <- a + 2
```

pourrait se traduire [1]

```
MOV AX, [a]   ; registre , direct
ADD AX, 2     ; registre , immédiat
MOV [b], AX   ; direct , registre
```

RISC vs CISC

	RISC	CISC
Signification	Reduced Instruction Set Computer	Complex Instruction Set Computer
Modes d'addressage	Peu	Beaucoup
Processeur	Moins complexe	Plus complexe

Les architectures CISC permettent de coder plus facilement des instructions de haut niveau comme

```
maStructure.unChamp <− 1
tab[3]  <− 2
tab[3]. unChamp <− 3
```

Adressage indirect

La donnée est à une adresse donnée par un registre.
Exemple :

```
MOV EBX, 0xB8A0   ; EBX contient l'adresse donnée
MOV EAX, [EBX]    ; on met dans EAX la donnée se trouvant à l'adresse 0xB8A0
```

Utile pour traduire les instructions manipulant les pointeurs / références
Exemple : Pour traduire objet1 <− objet2 (copie des références), on pourrait avoir

```
MOV EAX, [objet2]     ; l'objet référencé par objet2
MOV [objet1], EAX     ; et si on ne met pas les crochets ?
```

(On ne peut pas modifier l'adresse de qqchose sans crochet = adresse)

Avec déplacement

Adresse obtenue en ajoutant un déplacement à une adresse dans un registre

Exemple :

```
MOV AX, [EBX + 4]  ; AX reçoit la donnée se trouvant à l'adresse
                   ; donnée par (le contenu de) EBX + 4
```

Utile pour traduire les instructions manipulant les structures.
Exemple : Pour traduire maStructure.unChamp <− 1, on pourrait avoir. 2ème pas taille

```
MOV EAX, maStructure ; pas de crochet ici
MOV [EAX + 8], 1     ; Le '8' dépend des champs précédents dans la structure
```

Indexé

Le déplacement est lui aussi dans un registre. On y applique un facteur multiplicatif.

Exemple :

```
MOV AX, [EBX + 4 * ECX]  ; AX reçoit la donnée se trouvant à l'adresse
                         ; donnée par (le contenu de) EBX + 4 ∗ (le contenu de) ECX
```

Utile pour traduire les instructions manipulant les tableaux.
Exemple : Pour traduire tab[3] <− 1, on pourrait avoir

```
MOV EAX, tab
MOV EBX, 3
MOV [EAX + 4 ∗ EBX], 1     ; Le '4' est la taille d'une case.
; on comprend mieux que les tableaux commencent à0 dans de nombreux langages
```

Remarques

Nous n'avons pas tout dit :

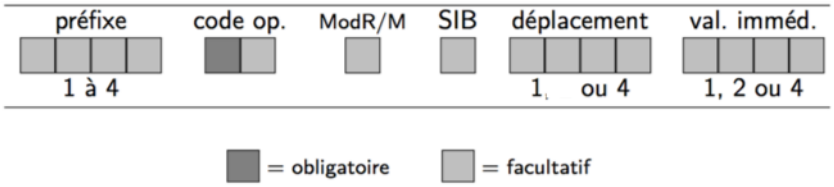
- Les modes portent parfois d'autres noms
- Il existe encore d'autres modes d'adressages
- Chaque processeur dispose d'un sous-ensemble des modes possibles
- Chaque assembleur dispose de sa propre syntaxe pour ces modes d'adressages

S7 Codage des instructions

Objectifs

Comprendre comment sont codées les instructions du x86
Être capable de traduire Assembleur ⇐⇒ Code machine

Forme générale d'une instruction (32bits !!)



Ce sont des bytes, en 64, dans préfixe il y a un byte en +, le byte de REX.
En 64, le code op a une taille de 3 byte.
Longueur variable en fonction de l'instruction
Assez complexe ⇒ procédons par étapes

Le code opératoire

Identifie l'opération à exécuter

- Souvent sur 1 byte, parfois sur 2
- Parfois, utilise une partie du byte ModR/M (on parle d'extension du code)

Un mnémonique, plusieurs code associés

- En fonction des opérandes
- Ex : INC peut se coder 40 à 47, FE ou FF

Un code opératoire, une seule instruction (extension comprise)

- Ex : CD est un INT,01estunADD

Opcode, lorsqu'il se termine par /r ➔ il y a un ModR/M, ib ➔ immédiat de 1byte, iw ➔ immédiat de 2bytes, …

Référence

Où trouver ces codes opératoires et les détails ?
Dans une référence (disponible sur poÉSI) Intel
Exemple

DEC

Opcode	Opérandes
FE /1	r/m8
FF /1	r/m16
48+rw	r16
48+rw	r32

Pas évident à lire ; nous introduirons les notations petit à petit.

Instructions

Tailles différentes, pour ça que RIP soute parfois de plusieurs.

Sans paramètres

Codée sur 1 byte : le code opératoire de l'instruction

Codage : **code op.**

NOP

Opcode	Opérandes
90	

NOP ⟹ 90 (code op.)

Avec une opérande immédiate

Codage : **code op. | immédiat sur 1, 2 ou 4 bytes**

```
INT 0x80
```

INT

Opcode	Opérandes	Explications
CD ib	imm8	imm8 est le numéro de l'interruption

ib : l'opcode est étendu par une valeur immédiate sur un byte

imm8 : l'opérande est une valeur immédiate interprétée sur 1 byte

INT 0x80 ⟹ CD (code op.) 80 (imm8)

Codage des registres

Dans les exemples suivants, nous devrons coder des registres.

- 000 : AL, AX, EAX, RAX, R8L, R8W, R8D, R8 ;
- 001 : CL, CX, ECX, RCX, R9L, R9W, R9D, R9 ;
- 010 : DL, DX, EDX, RDX, R10L, R10W, R10D, R10 ;
- 011 : BL, BX, EBX, RBX, R11L, R11W, R11D, R11 ;
- 100 : AH, SP, ESP, R12L, R12W, R12D, R12 ;
- 101 : CH, BP, EBP, R13L, R13W, R13D, R13 ;
- 110 : DH, SI, ESI, R14L, R14W, R14D, R14 ;
- 111 : BH, DI, EDI, R15L, R15W, R15D, R15 ;

Avec une opérande de registres

Codage : **code op. + n° registre**

```
INC EBX
```

INC (extrait)

Opcode	Opérandes	Explications
40 + rd	r32	incrémente le registre désigné

INC EBX ⟹ 43

40 (code op.) + 3 (EBX)

+rd : le numéro du registre 32 bits est ajouté à l'opcode.

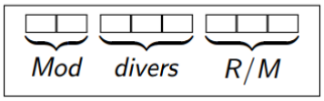
r32 : l'opérande est un registre 32 bits.

Le byte ModR/M

Utilisé pour les modes d'adressages complexes

- Mod : mode d'adressage pour la partie R/M
- R/M : opérande (peut nécessiter d'autres bytes)
- divers : dépend du nombre d'opérandes
  - 1 opérande : extension du code opératoire
  - 2 opérandes : deuxième opérande (un registre)

ModR/M



Mode d'adressage plus complexe

Analysons INC, …

INC (extrait)

Opcode	Opérandes
FF /0	r/m32

- r/m32 : l'opérande est un registre 32 bits ou une mémoire 32 bits
  - utilisation du byte ModR/M
  - permet les différents adressages complexes
  - nous allons les voir un à un
- /0 : la partie divers de ModR/M vaut 0, si /r alors dans divers c'est un registre.
  - on parle d'extension de code

Mode d'adressage par registres (11)

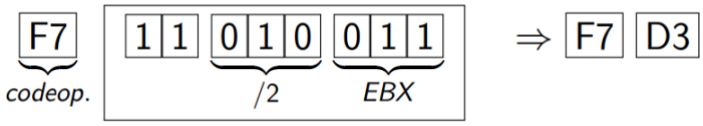
Codage : 

Sauf pour INC (inutile) car il a une autre forme. (Grand rectangle = ModR/M)

Exemple : not EBX

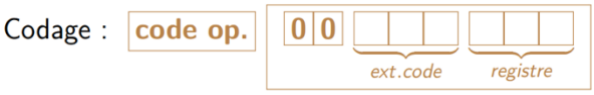
NOT (extrait)

Opcode	Opérandes
F7 /2	r/m32



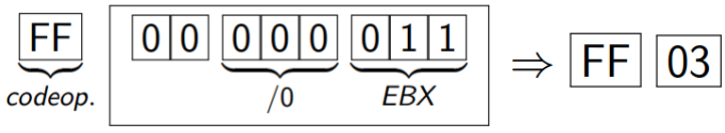
F7 D3 car on écrit l'op. code puis on coupe par 4 bytes, donc 1101 = 13 = D et 0011 = 3

Mode d'adressage indirect (00)

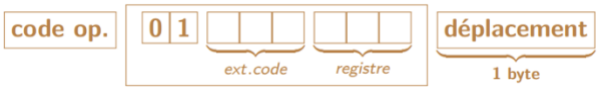
Codage : 

Le registre ne peut être **ni EBP ni ESP** (A la place de EBX)Car ça ne correspond pas à la même règles. Sont réservés. !Mais si je le fais, sa compile. !

Exemple : inc dword [EBX]



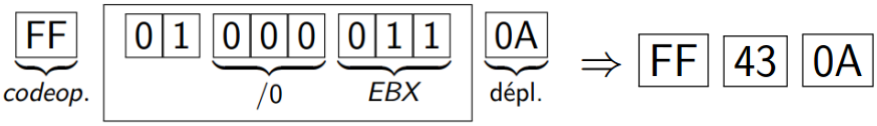
Mode d'adressage indirect avec déplacement court (01)



Codage : (déplacement : TJRS 1 bytes)

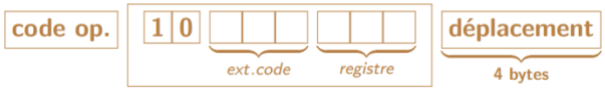
Le registre ne peut pas être **ESP**

Exemple : INC dword [EBX + 10]



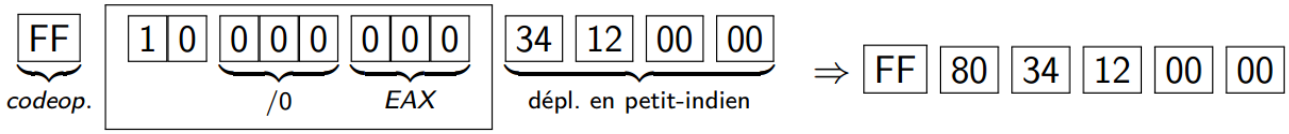
MIC

Mode d'adressage indirect avec déplacement long (10)



Codage :
Le registre ne peut pas être **ESP**
! Déplacement en little-endian.

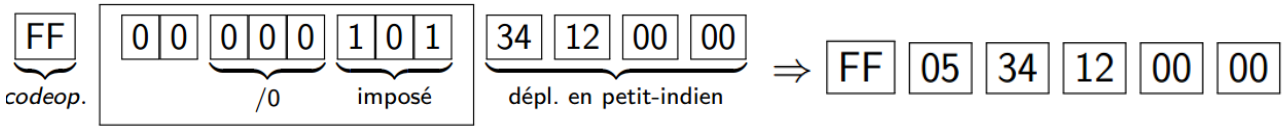
Exemple : INC dword [EAX+0x1234]



Mode adressage direct (00)

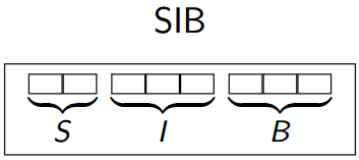
Codage :
(exc. EBP = imposé car il n'y a pas de registre donc on impose le registre EBP.)

Exemple : INC dword [0x1234]



Le byte SIB

Utilisé en complément à ModR/M. Pour les modes d'adressages indexés (B + S × I)

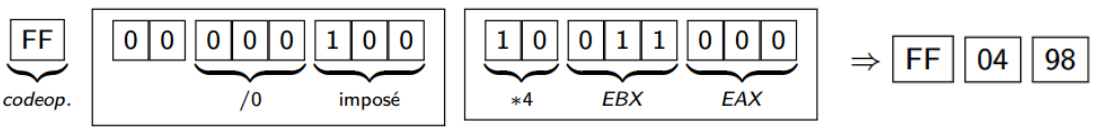


S: facteur multiplicatif (scale)
$2^i \Rightarrow 2^{00} = 1, 2^{01} = 2, 2^{10} = 4, 2^{11} = 8$
I: registre d'index
B: registre de base (celui qu'on ne doit pas multiplier)

Indirect indexé (00)

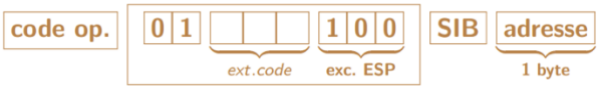
Codage :

Exemple : INC dword [EAX + 4*EBX]



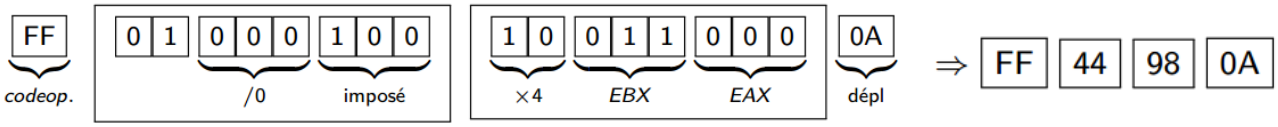
10 ➔ *4 car $2^{10} = 4$

Indirect indexé avec déplacement court (01)



Codage :

Exemple : INC dword [EAX + 4*EBX+10]



Récapitulatif ModR/M

Adressage	Exemple	ModR/M
registre	EAX	1 1 _ _ _ _ _ _
indirect	[EAX]	0 0 _ _ _ _ _ _
indirect + court	[EAX+10]	0 1 _ _ _ _ _ _
indirect + long	[EAX+800]	1 0 _ _ _ _ _ _
direct	[adresse]	0 0 _ _ _ 1 0 1
indirect indexé	[EAX+4*EBX]	0 0 _ _ _ 1 0 0
indexé + court	[EAX+4*EBX+10]	0 1 _ _ _ 1 0 0
indexé + long	[EAX+4*EBX+800]	1 0 _ _ _ 1 0 0

Instructions avec 2 opérandes

Une des deux opérandes sera toujours un registre (mode registre). Pour l'autre on aura toutes les possibilités

Exemple :

- **ADD EAX**, brol (registre, immédiat) ⇒ OK
- **ADD [EAX], EBX** (indirect, registre) ⇒ OK
- **ADD EAX, EBX** (registre, registre) ⇒ OK
- **ADD [EAX], [EBX]** (indirect, indirect) ⇒ interdit

Le code opératoire va dépendre de l'ordre des opérandes.

ADD (extrait)

Opcode	Opérandes
01 /r	r32/mem32, r32
03 /r	r32, r32/mem32

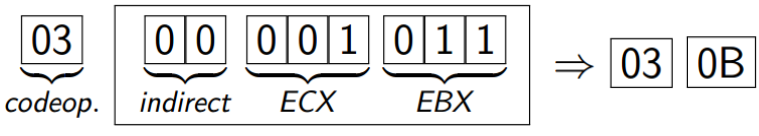
Opcode : /r pour le n° de registre ➔ r32 (peut être soit à droite soit à gauche).
Opcode : /r = registre ➔ registre sans [].
L'un le registre est à droite et la mémoire à gauche et l'autre l'inverse.

Exemples

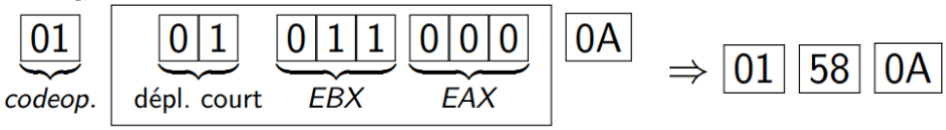
- **ADD EAX**, brol utilise le code 03
- **ADD [EAX], EBX** utilise le code 01
- **ADD EAX, EBX** utilise le code 01 ou 03

Exemple :

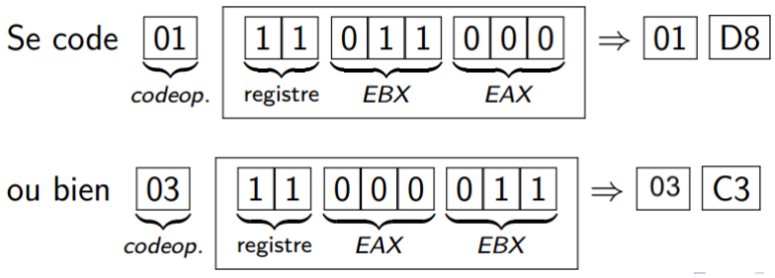
ADD ECX, [EBX]



ADD [EAX+10], EBX



ADD EAX, EBX



Les préfixes

32bits

Les préfixes modifient le comportement de l'instruction :

- Répétition : répétition automatique de l'instruction
- Taille adresse : modifie la taille par défaut des adresses (16/32)
- Taille registre : modifie la taille par défaut des registres (16/32)
- Segment : modifie le segment par défaut

À placer dans cet ordre

64bits

2 types de préfixe :

- Legacy préfixe : divisé en 4 groupes (répétition, réécriture des segments, réécriture de la taille opérandes, réécriture de la taille adresses)
- REX : 0100WRXB. Le 0100 restera toujours le même, WRXB peuvent chacun devenir un 1. Si dans la documentation : REX.W +.... Ça veut dire qu'a W on met un 1. Donc REX.W = 0100 1000 ➔ 48

io = immédiat en octal. Id = immédiat double.

Modification du segment

On ajoute le préfixe AVANT l'opcode.

Rappel : une adresse est sous la forme base : offset

- base : est l'adresse de début du segment.
- Une table reprend les descripteurs de segments.
- Des registres spécialisés CS, DS, ES, FS, GS et SS donnent l'index dans la table des segments.
- offset : est le déplacement dans le segment (spécifié par le programmeur).

Le segment utilisé dans une instruction est implicite

- INC [AX] utilisera le segment des données (DS) (adresse se trouve dans DS)
- JMP brol utilisera le segment du code (CS)

Peut être modifié (ex : **ADD [ES:EAX], EBX**) Cad la valeur EAX qui ne se trouve pas dans DS mais dans ES.

Codé dans le préfixe : (pas connaître par cœur) ➔

2E = CS	3E = DS	26 = ES
64 = FS	65 = GS	36 = SS

Exemple :

ES ➔ 26 01(add) 00(indirect)
011(EBX) 000(EAX) **ADD [ES:EAX], EBX** ⇒ 26 01 18
ES = Extra Segment

Préfixe de répétition

Permet de répéter ECX fois une instruction
En assembleur, on préfixe par REP, codé F3.

Exemple :

Répétition de l'instruction MOVSB (MOV String Byte ➔ codé A4)
MOVSB copie un byte de [ESI] vers [EDI] puis incrémente ESI et EDI.
Donc, pour copier les 10 premiers bytes de chaine1 dans chaine2 :

```
MOV ESI, chaine1
MOV EDI, chaine2
MOV ECX, 10
REP MOVSB          ; codé F3 A4
```

Taille des données et des adresses

Extrait de la référence de ADD

ADD (extrait)

Opcode	Opérandes
00 /r	r8/m8, r8
01 /r	r16/m16, r16
01 /r	r32/m32, r32

ADD EAX, EBX et ADD AX, BX se codent de la même façon ?

Un bit D dans le descripteur de segment indique si on utilise des registres/adresses 16 bits (D=0) ou 32bits (D=1).

Spécifié en assembleur via les macros [BITS 16] et [BITS 32]
On écrit ça tout au-dessus du fichier asm. Par défaut = 16bits

On peut modifier une instruction donnée par des préfixe :

- 0x66 : pour la taille d'une donnée
- 0x67 : pour la taille d'une adresse

Exemple :

```
[BITS 16]
INC AX             ; pas de préfixe
INC EAX            ; 0x66 : donnée sur 32 bits
INC word [EAX]     ; 0x67 : adresse 32 bits
INC dword [EAX]    ; 0x66 0x67 : donnée et adresse 32 bits
ADD AX, BX         ; pas de préfixe
ADD EAX, EBX       ; 0x66 : données sur 32 bits
ADD EAX, [EBX]     ; 0x66 0x67 : donnée et adresse 32 bits
```

si ça aurait été en 32 il n'y aurais pas eu de préfixes.

REX (0100WRXB)

- Les préfixes REX sont des préfixes d'instructions utilisés en mode 64-bits.
- Ils permettent entre autre de spécifier la taille des opérandes à 64-bits.
- Toutes les instructions ne nécessitent pas de préfixe REX en mode 64-bits.
- Un et un seul préfixe REX par instruction.
- Le byte préfixe REX précède immédiatement le byte opcode

REX.W

- Le bit REX.W peut être utilisé pour déterminer la taille des opérandes.
- Si un préfixe 0x66 est utilisé avec le préfixe REX et (REX.W=1), alors 0x66 est ignoré.
- Si un préfixe 0x66 est utilisé avec le préfixe REX et (REX.W=0), la taille des opérandes est 16-bits.

REX.R

- Le bit REX.R modifie le champ reg (partie divers) du Byte modR/M quand ce champ encode un registre R8 . . . R15.
- Le bit REX.R est ignoré si ModR/M spécifie d'autres registres ou définie une extension de l'opcode.

REX.X

- Le bit REX.X modifie le champ index du SIB.

REX.B

- Le bit REX.B : modifie la base du champ r/m dans le ModR/M (dans la partie mémoire) ou bien il modifie le champ base du SIB ou bien il modifie le champ reg de l'opcode utilisé pour accéder aux registres R8 . . . R15.

Exemple

Exemple : **ADD R8, [RDI + 0x3A]** (REX.W+03/r)

- Opération à 64-bits $\Longrightarrow$ REX.W=1
- SIB n'est pas utilisé $\Longrightarrow$ REX.X=0
- La base du champ r/m du ModR/M (la partie mémoire) = RDI
  $\Longrightarrow$ REX.B=0
- Le champ reg de ModR/M = R8 $\Longrightarrow$ REX.R=1

$\Longrightarrow$ REX = 01001100 = 4C

Exemple : **ADD R8, [RDI + 0x3A]** (REX.W+03/r)

- REX = 01001100 = 4C
- Opcode = 03
- ModR/M :
  - Mode adressage = 01
  - Registre = R8 = 000
  - Mémoire (r/m) = RDI = 111

  $\Longrightarrow$ ModR/M = 01000111 = 87
- Déplacement = 0x3A

d'où : **ADD R8, [RDI + 0x3A]** $\Longrightarrow$ 4C 03 87 3A

Environnement d'exécutions de base

Ce qu'il faut savoir

L'architecture IA-32 supporte 3 modes de fonctionnement de base :

- Mode protégé ;
- Mode réel ;
- Mode gestion du système (SMM)

L'architecture Intel 64 ajoute le mode IA-32e, il a 2 sous-modes :

- compatibilité : permet exécution de 16 et 32 bits sur OS 64bits
- mode 64 bits : permet à un OS 64-bits d'exécuter des applications écritent pour accéder un espace linéaire 64-bits (applications 64-bits).

En mode 64bits :

- La taille par défaut des adresses est 64 bits
- La taille par défaut des opérandes des instructions est 32 bits
- L'utilisation du préfixe REX sous la forme de REX.R permet d'accéder aux registres de R8, R9, R10,. . . , R15
- L'utilisation du préfixe REX sous la forme de REX.W met les opérations à 64 bits

Code machine sur un byte fixe

Codée sur 1 byte : le code opératoire de l'instruction
CLC = clear carry (clear le carry flag) son opcode = F8

Code machine sur un byte

Codée sur 1 byte : le code opératoire de l'instruction
NOP = no operation ➔ opcode = 90

NOP — No Operation

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
NP 90	NOP	ZO	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Code machine avec valeur immédiate

MOV

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
88 /r	MOV r/m8,r8	MR	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8***,r8***	MR	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	MR	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	MR	Valid	Valid	Move r32 to r/m32.
REX.W + 89 /r	MOV r/m64,r64	MR	Valid	N.E.	Move r64 to r/m64.
8A /r	MOV r8,r/m8	RM	Valid	Valid	Move r/m8 to r8.
REX + 8A /r	MOV r8***,r/m8***	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg**	MR	Valid	Valid	Move segment register to r/m16.
REX.W + 8C /r	MOV r16/r32/m16, Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/r64/m16.
REX.W + 8C /r	MOV r64/m16, Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
8E /r	MOV Sreg,r/m16**	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + 8E /r	MOV Sreg,r/m64**	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8*	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8*	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16*	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64*	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8***,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16*,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64*,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8***, imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.

Sreg pour dire registre de segment.
Op/En MR mémoire gauche registre droit
RM registre gauche mém droite
Moffs offset exemple la case d'un tableau.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

Exemple : MOV EAX, 20 et MOV RAX, 20

- MOV EAX, 20 $\Longrightarrow$ B8 14 00 00 00 ;
- MOV RAX, 20
  $\Longrightarrow$ 48 B8 14 00 00 00 00 00 00 00.

MOV r16, imm16 et MOV r32, imm32
Donc B8 suivit de 20 mais en hexa et sur 4bytes car imm16
B8 suivit de 20 en hexa et sur 8bytes.

ADD

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8*.
REX + 80 /0 *ib*	ADD *r/m8*, imm8*	MI	Valid	N.E.	Add *sign-extended imm8* to *r/m8*.
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16*.
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32*.
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64*.
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign-extended imm8* to *r/m16*.
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign-extended imm8* to *r/m32*.
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign-extended imm8* to *r/m64*.
00 /*r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8*.
REX + 00 /*r*	ADD *r/m8*, r8**	MR	Valid	N.E.	Add *r8* to *r/m8*.
01 /*r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16*.
01 /*r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32*.
REX.W + 01 /*r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64*.
02 /*r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8*.
REX + 02 /*r*	ADD *r8*, r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8*.
03 /*r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16*.
03 /*r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32*.
REX.W + 03 /*r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA

Exemple : ADD EAX, 10 et ADD RAX, 10

- ADD EAX, 10 ⟹ 05 0A 00 00 00 ;
- ADD RAX, 10
  ⟹ 48 05 0A 00 00 00.

ADD EAX, imm32 ➔ 05 suivit de 10 en hexa, sur 4 bytes.
ADD RAX, imm32 ➔ REX.W (48) + 05 suivit de 10 en hexa sur 4 bytes.
REX = 0100WRXB REX.W = 0100 1000 = 48

ADD ➔ ModR/M
ADD [0x1234], ESI il faut un registre et du coup c'est EBP ou RBP qui va remplacer l'immédiat, ce sera la mémoire.

Exercice

ADD [ECX + 2*EAX], EBX
01 0001 1100 01 000 001
Op code = 01
Registre = 011
Indirect indexé (00) se code : opcode 00 registre 100 SIB
SIB = 01(car 2^1 = 2) 000(EAX) 001(ECX)
Donc à la fin ➔01(OPCODE) 00 011(EBX) 100 (SIB)01 000 001
Op code puis tableau(img en dessous) puis sib

indirect indexé	[EAX+4*EBX]	0	0				1	0	0

S8 Cartographies de la mémoire-mode réel

En mode réel il n'y a qu'1méga de ram et les adresses sont fixes.
Les bus de données et d'adresse permettent au processeur de communiquer avec :

- La mémoire, il y a 2 partie de mémoire. (S.E. = Système d'exploitation)
- Les périphériques via son contrôleur(il y a donc un contrôleur de périphérique.).

ROM

- Une partie de cette mémoire est de la ROM (Read Only Memory) non volatile (lorsque l'on éteins le PC, les données reste.).
- Les premières instructions exécutées au démarrage du PC sont mémorisées en ROM.
- À cet endroit, vous trouvez un premier S.E. appelé le BIOS (Basic Input Output System)
- Aujourd'hui, cette ROM est reprogrammable à l'aide de logiciels. On dit 'flasher' la ROM. On parle de flasher, car aux premiers temps des ROM programmables, on les effaçait en les éclairant avec une lampe U.V.

RAM

- Une autre partie de cette mémoire est de la RAM (Random Acces Memory) volatile (dès qu'on éteins le PC, tout part).
- Toutes les informations en mémoire (instructions, données, adresses...) qui se modifient pendant le fonctionnement de l'ordinateur se trouvent en RAM.
- Une partie de cette RAM peut être utilisée par les process des utilisateurs.
- Conventionnellement, les 640 premiers Kb sont réservés au S.E. et aux process des utilisateurs(avant c'était beaucoup).

Périphérique

- Il n'y a pas de conflit avec les accès mémoire car une patte du processeur permet de sélectionner le périphérique ou la mémoire.(en fonction de qui le demande. Si périphérique demande le proco alors il sélectionne périph)
- Des instructions spécialisées permettent au processeur de décider s'il s'agit d'un accès mémoire (MOV par exemple) ou d'un accès périphérique (IN, OUT).

MOV

- On peut toujours décider de câbler un contrôleur à certaines adresses de la RAM. (Chaque périphérique à son numéro)
- L'utilisateur écrit dans le périphérique comme s'il écrivait en RAM.
- Dans le cas du compatible PC, on a placé le périphérique vidéo texte(des caractères) aux emplacements 0xB8000(en haut à gauche)-0xB8FFF(en bas à droite) et le périphérique vidéo graphique(pour faire des pixels) aux emplacements 0xA0000-0xAFFFF(se sont des adresse réel.).
- L'électronique du contrôleur vidéo affiche en permanence à l'écran les informations qui se trouvent à ces adresses.
- Un simple MOV à ces adresses provoquent un affichage !
- Le contrôleur vidéo peut travailler en différents modes (texte, 320x200, 640x480(se sont des pixels),...) que l'on peut changer à l'aide d'une instruction OUT dans un registre du contrôleur.

Cartographie de la RAM PC

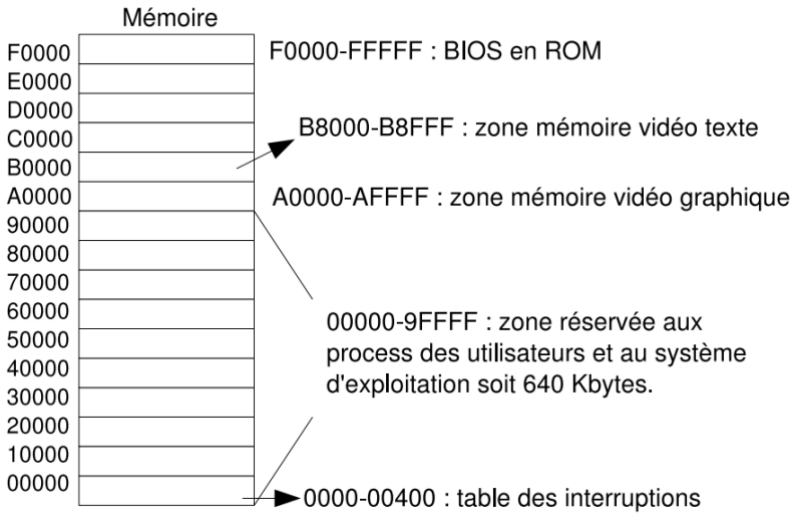


Table d'interruptions : se trouve dans la ram, 2 tables différentes : mode réel (entre 0000 et 00400) et mode protégé, on ne sait pas ou elle est

Afficher un h à l'écran

```
MOV     AX,0xB800
MOV     ES,AX
MOV     AL,'h'
MOV     AH,10010111b
MOV     [ES:0xA0],AX
```

- Un caractère = rectangle dans lequel il y a lettre. clignotant ou pas (comme curseur qui clignote)
  3 suivants : RVB du fond d'écrans (du rectangle)
  est-ce que la couleur est foncé ou clair ==> 0 foncé
  3 suivant : RVB du caractère.
- ES est dédié à l'affichage

pour 1 caractère ==> 2bytes : 1 pour caractère et 1 pour couleur

Le contrôleur vidéo affiche en mode texte tout ce qui se trouve en B8000-B8FFF sous la forme de 25 lignes de 80 words. Chacun de ces words est structuré en un byte qui est le code ansi du caractère et en 8 bits représentant sa couleur CrvbLRVB où RVB est la couleur du caractère, rvb, la couleur du fond, L pour 'plus lumineux' et C pour clignotant. Ce programme affichera un h gris clignotant sur fond bleu à 2ème lignes et 5èmes colonnes

S9 - Démarrage

A la mise sous tension

- L'alimentation met un certain temps avant de fournir un courant stabilisé/fixe. CAD pas de défaut de l'alimentation.
- Dès que le courant est stable (soit timer, soit power good détecté par la carte mère), READY est sous tension (patte du processeur) ;
- CS : IP(notre pointeur d'instructions) est chargé avec l'adresse FFFF0, le processeur démarre son cycle ;
- En FFFF0, on trouve normalement un jmp vers une adresse en ROM.

Le démarrage hardware est terminé, le software adressé par FFFF0 est exécuté.

BIOS

À la mise sous tension, un jmp est effectué vers un bout de code qui :

- Effectue un check de périphériques, de la RAM ... ;
- Complète la table des interruptions ;
- Complète les routines d'interruption ;
- Cherche un périphérique de boot ;
- S'il le trouve, charge le premier secteur de ce périphérique en mémoire à l'adresse 07C00.
- Remplace CS : IP par l'adresse 0000 : 7C00 ;

Secteur de boot d'une disquette

Le premier secteur du périphérique
Enregistrement structuré de 512 bytes

- 000-509 : libre (zone de code, on a 510bytes, et là on va placer un programme de démarrage.)
- 510-511 : 0x55AA (c'est ce qui définit un secteur de boot)

De 000 à 509, on dispose de 510 bytes pour placer un programme de boot.

Secteur de boot d'un disk (ou usb)

Le premier secteur du périphérique
Enregistrement structuré de 512 bytes

- 000-439 : libre (zone de code)
- 440-443 : signature du disque (optionnel)
- 444-445 : 0x0000 (optionnel)
- 446-509 : description des 4 partitions primaires
- 510-511 : 0x55AA (la même signature que diskette)

De 000 à 439, on dispose de 440 bytes pour placer un programme de boot.

Un process sans interruption du BIOS

Cad que lorsqu'il va lire le 07C00 Il ne va pas lancé le démarrage mais autre chose.

```
MOV     AX,0xB800
MOV     ES,AX
MOV     AL,'B'
MOV     AH,10010111b
MOV     [ES:80*2*12+40*2],AX    ; milieu
JMP     $                       ; boucle infinie
times   510-($-$$) DB 0x90      ; 0x90=NOP --> 510
DB      0x55,0xAA               ; signature boot
```

$ = adresse courante donc il saute sur lui-même.

DB : on peut sans mettre de nom de variable mais pas en protégé.

510 = taille secteur de boot.
$$ = adresse du début de la 1ère instructions. donc ça nous fais la différence d'octets entre le début et la fin. Et le reste je remplis par NOP.
Ici le but est juste de remplir les octets.
$-$$ nous donne la taille de tout jusqu'ici

après on doit placer celà dans une disquette puis le mettre dans un programme pour le transformer en binaire et le charger dans le 1er secteur d'une disquette ou USB mais a la place de 510 on aurais changé par 440.
Pour affiche juste un B à L'INFINI.

S10 - Coprocesseur

Introduction

Il fait déjà partie du processeur.
Coprocesseur : circuit destiné à ajouter une fonction à un processeur classique.

- Coprocesseurs arithmétiques (calcul en virgule flottante)
- Coprocesseurs graphiques (2D, 3D)
- Coprocesseurs spécialisés dans le chiffrement

But : augmenter les performances de la machine pour un type de calcul précis.

Le coprocesseur Intel 8087

Co-processeur arithmétique pour la gamme x86
x86 : uniquement calculs de nombres entiers

- Calculs de réels simulés avec des entiers représentant des nombres à virgule fixe implicite
- Ex : 2.5 + 3.4
- On représente les nombre comme 250 et 340 (si deux décimales implicites) La somme donne 590 ce qui correspond à 5.9.

Le 8087 introduit des instructions de calcul de nombres « à virgule flottante »
Disponible dès le 8086, comme circuit intégré séparé.
À partir du 80486DX (32 bits), le coprocesseur est intégré
dans la même puce que le processeur «classique»

Le calcul par pile

Principe

- Les opérandes sont pushées sur la pile
- Les calculs poppent les opérandes et pushent le résultat
- Le résultat final est disponible au fond de la pile

Technique très employée par

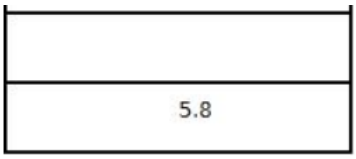
- Compilateurs
- Certains processeurs et coprocesseurs
- Coprocesseur Intel 8087

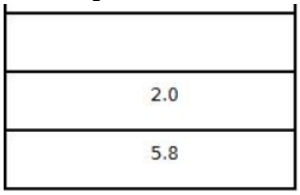
Exemple 1

Pour faire 2.5 + 3.3
On push 2.5 puis on push 3.3 ensuite on fait la somme. Le résultat est au fond de la pile.
Comme en assembleur, on met les paramètres puis on appelle la fonction somme.
En vrai ce n'est pas vraiment « Push » mais c'est pour simplifier.

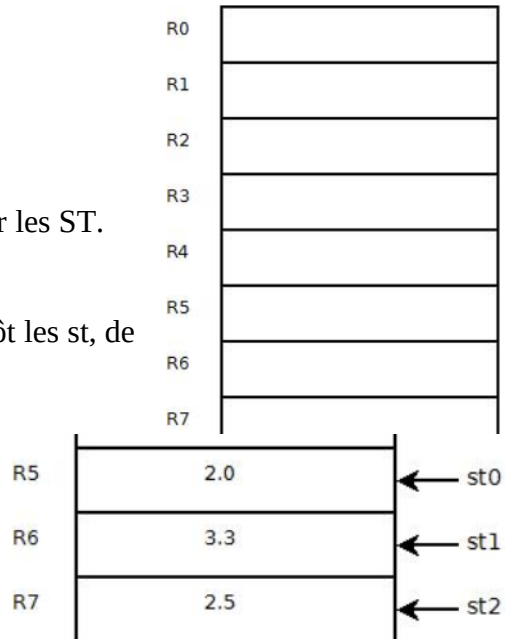
Exemple 2

Pour faire (2.5 + 3.3) × (1.2 + 0.8)
On push 2.5 puis 3.3 ensuite on fait la somme, puis on push 1.2 puis 0.8, on fait la somme puis ont fait le produit. Les opérations se font sur les 2 chiffres le + en haut.





Intel 8087



Registres

Pile hardware dans le coprocesseur
8 registres : R0 à R7 ➔ physique, registres généraux.
La pile est cyclique, après R7 on revient à R0, pareil pour les ST.
Les entiers et réel sont transformé en réel sur la pile.

Les registres ne sont pas trop employé car on utilise plutôt les st, de 0 à 7.
St0 = sommet de la pile.
st0 c'est TJRS l'élément le + au-dessus de la pile,
quand on ajoute un élément, les st se décale
la plus part des instructions utilisent st0.

sw.top vaut 0 c'est que R0 correspont à st0.
si sw.top vaut 4, il y a un décalage de 4 donc st0 correspont au resgistre R4.

FINCSTP : incrémente sw.top sw = status word
Tout celle qui comment par F c'est pour les réel.
FDECSTP : le désincrémente.

Quand pile est vide, sw.top vaut 7
sw.top n'est définit qu'une fois au tout début.

Il y a une correspondance entre flag et statu. Que pour nb non signé.
REGISTRES SPECIAUX = tout ce qu'il y a écrit après

status word registre de 16bits
il a des position ou certain flag se trouve au même endroit que les flag de registre rflags pour pouvoir faire des comparaison.

tag word : registre 16bits (décris ce que contient chaque registres.)

00:st(i) contient une donnée valide
01:st(i) contient 0
10:st(i) contient un nombre spécial (infini)
11:st(i) contient est vide.

control word (dedans il y a entre autre AA et PP)

AA Arondi
00 : arrondi au + proche
11 arrondi tronquée.
les autres pas utilisé.

PP précision lors du transfert vers la mémoire.

00: 24bits (réel court)
01: non utilisé
10: 53bits (réel long)
11: 64bits (réel temporaire)

L'encodage des réels sous Intel 8087

Les registres st0 à st7 contiennent des réels de 80 bits
Valeurs possibles dans les registres :

- Nombre réel (80 bits)
- « vide »
- « infini »
- « NAN »

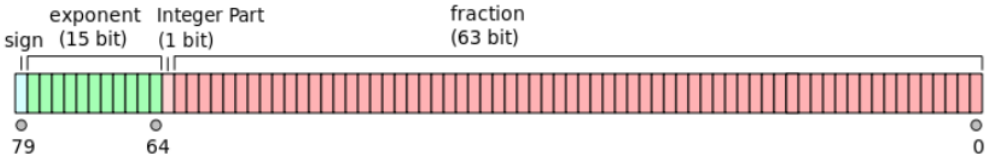
tword(comme qword etc) = 10 octets (t = ten)

- Même usage que byte, word, dword

Encodage des réels en format « x86 Extended Precision »

- Bit de signe s
- Exposant e sur 15 bits (avec biais de 16383)
- Mantisse m sur 64 bits (partie entière en bit 63, partie fractionnaire au-delà)

Signe = 1bit
Mantisse = 63bits
exposant sur 15bits
Un nbre en virgule flottante est codée sur 80bits. De 0 à 79.



Formule

$$(-1)^s \times m \times 2^{e-16383}$$

Exemple

- Exemple : 4001C000000000000000

s = 0 ➔ positif le dernier vert (à gauche)
e = $100000000000001_2 = 16385_{10}$ vert sauf le dernier.
m = $1.100000\ldots0_2 = 1.5_{10}$ ➔rose claire.rose flash

Formule :

$$\begin{aligned}(-1)^s \times m \times 2^{e-16383} &= (-1)^0 \times 1.5 \times 2^{16385-16383} \\ &= 1 \times 1.5 \times 2^2 \\ &= 6\end{aligned}$$

Instructions

FLD = Floating-point LoaD

Syntaxe : FLD param param est pushé au sommet param = immédiat, mémoire, sti
Exemple : FLD tword 2.5 ➔ st0 prend 2.5

Variante

FLDZ : PUSH 0.0 FLD1 : PUSH 1.0 FLDPI : PUSH π

FST = Floating-point STore

Syntaxe : FST param store st0 dans param.
Exemple : x DT 0.0 (définit une variable nommé x de taille ten.) PUIS
FST tword [x] enregistre st0 dans la variable x définit par avant.

Variante

FSTP = Floating-point Store Pop ➔ Le store puis le pop, enlève la valeur de la pile.

Opérations arithmétiques du 8087

Opérations binaires

- Addition
- Soustraction
- Multiplication
- Division

Opérations unaires

- Racine carrée
- Valeur absolue
- Changement de signe
- Conversion réel-entier
- Cosinus

Addition

FADD = Floating-point ADD
Syntaxe : FADD sti, stj ➔ met la somme de sti et stj dans sti.

Variante :
FADDP = Floating-point ADD Pop Pareil que FADD sauf qu'après il POP.

Raccourcis
FADDP sti ➔ raccourci pour FADDP sti, st0
FADDP ➔ raccourci pour FADDP st1, st0

Exemple

Addition de 3 nombres.
On fais 3 fois « FLD tword mon_nombre » suivit de 2fois FADDP

Soustraction

FSUB/FSUBP (Floating-point SUBstract Pop)

Multiplication

FMUL/FMULP (Floating-point MULtiply Pop)

Division

FDIV/FDIVP (Floating-point DIVide Pop)
Même syntaxe que l'addition (raccourcis inclus) pour tous

Opérations unaires

Valeur absolue

FABS = Floating-point ABSolute value st0 ← |st0|

Racine carrée

FSQRT = Floating-point SQuare RooT st0 ← $\sqrt{st0}$

Cosinus

FCOS = Floating-point COSinus st0 ← cos(st0)

Conversion reel ⇔ entier

Rajout d'un I après le F initial (I=Integer) = convertion automatique réel ⇔ entier
Ex : (FLD) FILD eax ➔eax (entier 32 bits) converti vers réel 80 bits puis pushé.
On charge un entier.
Ex : FISTP eax ➔réel 80 bits converti vers entier 32 bits puis poppé vers eax

S10 - Évolution des processeurs (Moore et Pipeline)

Moore

Les ordinateurs doublent de puissance tous les 2 ans pour le même volume et le même prix de revient. CAD tous les 2 ans on double le nombres de transistors.
Si à l'exam on doit expliquer ➔ maintenant c'est plutot tout les 18 mois. Sa ralentit au fur des années. Sa ralentit car on arrive au limite de la physique.
Cette loi dit en 65.

Pipeline

Pipeline = chaine de traitement. Utilité : gagner du temps.

Sans pipeline

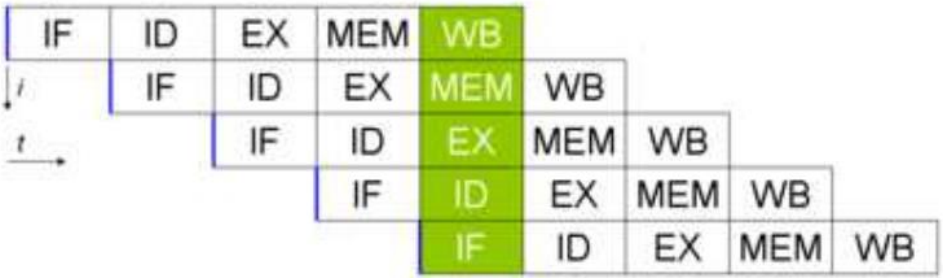
Les instructions sont effectuées les unes après les autres. Pour exécuter la prochaine instructions, l'instructions courante doit obligatoirement être terminée.
Ces instructions sont découpées en :

- IF Instruction fetch ou lecture de l'instruction (charge l'instruction à exécuter)
- ID Instruction decode ou décodage de l'instruction ;
- EX Execute ou exécution de l'instruction ;
- MEM Lecture en mémoire éventuelle ;
- WB Write Back ou stocker le résultat.

ici pour exécuter 1 instructions -> 5cycle donc 3 instructions-> 15cycles

Avec pipeline

Ici, 5 pipelines permettent d'effectuer les instructions en parallèle en utilisant des parties différentes du circuit. En augmentant encore le nombre de pipelines, on obtient des processeurs super scalaires.
Le processeur peut donc s'occuper de plusieurs instructions en même temps.



On exécute 5 instructions en 9 cycles
Le nombre d'étages d'un pipeline = ça profondeur.