Notes du cours de microprocesseurs (MIC2)

Nathan Furnal

31 mai 2021

Table des matières

1 Ouvrons le PC

En ouvrant le PC, on rencontre la carte mère et la mémoire cache, sur lesquels nous nous concentrerons.

1.1 Carte mère

La **carte mère** est le cœur de tout ordinateur. Elle est essentiellement composée de circuits imprimés et de ports de connexion, par le biais desquels elle assure la connexion de tous les composants et périphériques propres à un micro-ordinateur (disques durs, mémoire vive, microprocesseur, cartes filles, etc.) afin qu'ils puissent être reconnus et configurés par la carte lors du démarrage.

Dans la carte mère, on retrouve :

— Les connecteurs électriques

— Le support processeur (*socket*)
— Le **chipset** : Celui-ci se divise en deux parties distinctes à savoir le **northbridge** et **southbridge**.
— Les **bus**
— Les **slots** mémoire
— Les **slots** extension
— Les connecteurs de stockage
— Le panneau d'entrées/sorties

Aujourd'hui on a remplacé les architectures **northbridge** et **southbridge** par le **PCH** (platform controller hub).

1.2 Carte multiprocesseur

Une carte multiprocesseur peut accueillir plusieurs processeurs physiquement distincts. Elles sont principalement utilisées dans les architectures serveur ou les super-ordinateurs.

1.3 La mémoire cache

La mémoire cache (également appelée antémémoire ou mémoire tampon) est une mémoire rapide permettant de réduire les délais d'attente des informations stockées en mémoire vive. En effet, la mémoire centrale de l'ordinateur possède une vitesse bien moins importante que le processeur. Il existe néanmoins des mémoires beaucoup plus rapides, mais dont le coût est très élevé. La solution consiste donc à inclure ce type de mémoire rapide à proximité du processeur et d'y stocker temporairement les principales données devant être traitées par le processeur.

Les ordinateurs récents possèdent plusieurs niveaux de mémoire cache :

— La mémoire cache de premier niveau (appelée **L1 Cache**, pour Level 1 Cache) est directement intégrée dans le processeur.
— La mémoire cache de second niveau (appelée **L2 Cache**, pour Level 2 Cache) est située au niveau du boîtier contenant le processeur (dans la puce).
— La mémoire cache de troisième niveau (appelée **L3 Cache**, pour Level 3 Cache) autrefois située au niveau de la carte mère (utilisation de la mémoire centrale), est aujourd'hui intégré directement dans le CPU.

Tous ces niveaux de cache permettent de **réduire les temps de latence des différentes mémoires** lors du traitement et du transfert des informations.

2 Microprocesseurs : fonctionnement

Le programme est une suite d'instructions stockées dans la mémoire. Pour exécuter ces instructions de manière séquentielle, le microprocesseur utilise `RIP`, qui contient l'adresse de la prochaine instruction.

2.1 Composantes

ALU Unité arithmétique et logique, qui exécute les opérations et les instructions.

registres Les registres contiennent les valeurs, ils servent au stockage.

Unité de contrôle (UC) Cette unité contrôle l'exécution des instructions et contient :

RIP L'adresse de la prochaine instruction à être exécutée.

Incrémenteur Il incrémente la valeur de RIP automatiquement.

Registre d'instructions (RI) Contient l'instruction qui va être décodée avant d'être exécutée.

Décodeur Décode les instructions.

Séquenceur Contient les adresses vers lesquels RIP pointe.

Horloge Contrôle la cadence des instructions.

Bus de données Transporte les données, il aide à manipuler les données qui sont stockées en mémoire, là où le bus d'adresses indique où celles-ci se trouvent.

Bus d'adresses Le bus d'adresses collecte les adresses.

Mémoire RAM On retrouve aussi bien les instructions que les données nécessaires aux instructions.

2.2 Code machine et instructions

— `0x0102030405060708` : adresse sur 64 bits.
— `0x480556341200` : instruction (`add rax, 0x123456`);
— `0x4801D8` : instruction (`add rax, rbx`);
— `0x4801B48A34120000` : instruction (`add [rcx*4+rdx+0x1234], rsi`)

2.3 Jeu d'instructions

Les microprocesseurs sont capables d'effecteur certaines opérations élémentaires, l'ensemble de ces opérations est appelée **jeu d'instructions**.

Une instruction au niveau machine doit fournir à l'unité centrale toutes les informations nécessaires pour déclencher une opération. En général, les opérations élémentaires comportent plusieurs champs, le premier champ contient le code de l'opération, appelée **op-code**.

2.4 Cycle d'exécution d'une instruction

Lors de son exécution, une instruction est décomposée en mini-opérations élémentaires.

FETCH Recherche d'instruction.

DECODE Décodage de l'instruction.

EXECUTE Exécution de l'instruction.

Ces opérations sont cadencées au rythme d'horloge, qui pilote le séquenceur. La durée de traitement d'une instruction s'appelle **cycle d'instruction** ou **cycle machine**. Le nombre de périodes d'horloge nécessaires à l'exécution dépend de l'architecture du processeur et du mode d'adressage.

2.4.1 `Fetch`

Le contenu de `RIP` est placé dans le bus d'adresses ; l'unité de contrôle établit la connexion entre les deux. Ensuite, l'unité de contrôle émet un ordre de lecture de la case mémoire dont le contenu sera ensuite acheminé à travers le bus de données, vers le registre d'instruction (RI).

Remarque : À la mise sous tension ou après un `RESET`, le compteur de programme est initialisé par une valeur fixée par le constructeur.

2.4.2 `Decode`

Le registre d'instruction (RI) contient maintenant le code opératoire (op-code). L'unité de contrôle (UC) décode le contenu de RI pour savoir la nature de l'opération à effectuer (addition, multiplication, etc.) et incrémente RIP pour pointer sur la prochaine instruction à exécuter.

2.4.3 `Execute`

Le cycle d'exécution varie en fonction de l'architecture et du type de l'instruction. De manière générale c'est l'ALU qui exécute l'instruction en cours et positionne les indicateurs du registre d'état (le registre qui contient les *flags*) comme `RFLAGS`.

2.4.4 Exemple 1

On considère le processus représenté par le code assembleur suivant :

```
1 mov eax, 33
2 add eax, ecx
```

Le code machine correspondant est : `B82100000001C8`.

On suppose qu'au départ, `EAX` contient 17, `EBX` contient 19 et `ECX` contient 15. Le contenu des autres registres est inconnu.

D'abord on exécute la première instruction `mov eax, 33`, cela correspond à `B821000000` en code machine. On commence par `FETCH`. La porte s'ouvre et l'adresse contenue dans RIP est placée sur le bus d'adresses. La RAM fournit la donnée qui se trouve à cette adresse. Ensuite, l'instruction qui se trouvait à l'adresse donnée (`0x10..84`) se retrouve dans le registre d'instruction RI, en passant par le **bus de données**.

Puis, on décode l'instruction grâce au décodeur et au registre d'instruction. On voit que c'est un `mov` sur 5 bytes. Les portes s'ouvrent à nouveau, RIP s'incrémente de 5 bytes et vaut maintenant (`0x10..89`).

Enfin, on exécute cette première instruction.

2.4.5 Exemple 2

On considère le processus représenté par le code assembleur suivant.

```
boucle : ADD BL, 10
JMP boucle
```

Le code machine correspondant est : `80C30AEBFB`.

On suppose qu'au départ, `EAX` contient 17, `EBX` contient 05 et `ECX` contient 15. Le contenu des autres registres est inconnu.

Pour arrêter un processus qui tourne en boucle, on va devoir utiliser une **interruption**.

3 Interruptions

Pour rappel, une instruction est décomposé en `FETCH`, `DECODE`, `EXECUTE`. Mais on peut se poser la question de ce qu'il se passe si on veut interrompre ce processus. Par exemple dans le cas d'un événement extérieur (frappe au clavier, clique souris,...). Dans le cas d'une erreur exécution (instruction inconnue, division par zéro). Ou encore dans le cas de dialogue avec le système (appel système, scheduling de processus).

Dans tous ces cas, la gestion de l'arrêt des instructions passe par les **interruptions**. Une interruption est un arrêt temporaire (en général) et automatique du processus en cours. L'interruption provoque l'exécution d'une routine (un bloc de code) qu'on appelle *interrupt handler*, la routine de traitement d'interruption. En général, le programme reprend après l'interruption.

Pour savoir s'il faut interrompre un programme, on doit constamment "écouter" ou scruter les processus en cours. Ce concept de scrutation des programmes s'appelle le **polling**, qui est une attente active du CPU.

3.1 Types d'interruptions

Il y a trois types d'interruption :

- L'interruption **matérielle**, provoquée par du matériel extérieur comme le clavier ou la souris ou bien l'arrivée d'un paquet sur le réseau.
- Les **exceptions**. Elles sont provoquées par une programme, en général suite à une erreur. Par exemple la division par 0 ou une instruction inconnue.
- Les **appels-système**. Ils sont provoqués par un programme, pour demander un service au système d'exploitation. Par exemple la demande de lecture ou d'écriture d'un fichier.

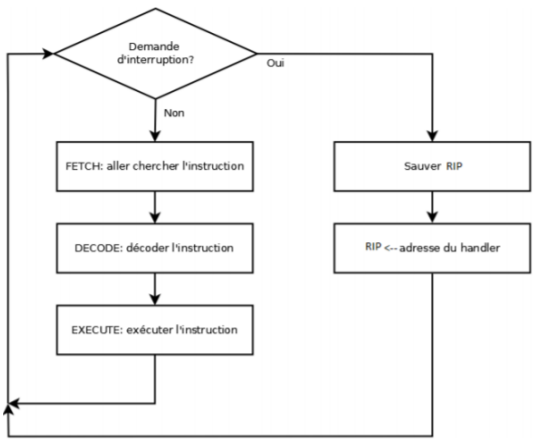
Les interruptions sont reprises dans un tableau appelé vecteur, chaque interruption aura une valeur précise. Il y en a 256.

0-31 Valeurs réservés aux exceptions.

- **0** Division par zéro.
- **6** opcode non défini.

32-255 définies par l'OS et programmées dans le contrôleur d'interruptions, le PIC (*Program interrupt controller*).

Voici le cycle du processeur quand on a une interruption, dans ce cas il n'y a pas d'attente active. On verra plutôt s'il y a une interruption après chaque instruction. L'adresse du *handler*, c'est l'adresse de la routine d'interruption.



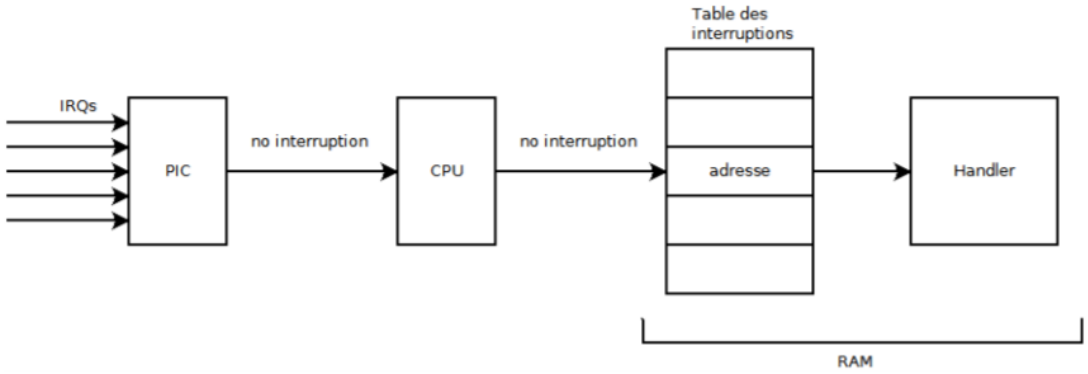
3.2 La broche `INTR` et le contrôleur d'interruptions

La broche `INTR` (*Interrupt Request*) signale au processeur l'arrivée d'une interruption matérielle, elle est unique et se trouve sur le microprocesseur. Elle gère plusieurs sources potentielles d'interruptions de manière **sérialisée**.

La broche et donc le microprocesseurs sont liés au contrôleur d'interruptions (*PIC*) qui est lui-même relié aux périphériques via les bornes `IR(Q)` (*interrupt request*). Il envoie les demandes d'interruptions une par une au CPU via la borne `INT`.

Chaque borne `IR(Q)` décrit un type spécifique d'interruption : clavier, port, disque, etc. Si on est limité par le nombre de bornes pour les interruptions, on peut mettre deux *PIC* en cascade et augmenter le nombre d'interruptions possibles.

Les adresses mémoire des routines d'interruptions et leurs requêtes d'interruptions associées sont stockées dans la RAM, suivant le schéma suivant :



Maintenant, on peut se demander comment fonctionne les *handlers* en monoprogrammation. Lors de l'interruption :

1. On `PUSH` RIP sur la pile pour le sauver.
2. `MOV RIP, adresse routine interruption`
3. On continue l'exécution jusqu'à atteindre `IRET`.
4. On exécute automatiquement l'instruction `IRET` en fin d'interruption qui restaure l'ancien `RIP` (avec `POP`) et permet de revenir au programme interrompu.

Enfin, les registres de `RFLAGS` et plus particulièrement le *flag* `IF` nous informe si le CPU est interruptible ou pas. S'il vaut 0, le CPU n'est pas interruptible et s'il vaut 1, le CPU est interruptible. L'OS utilise les instructions `CLI` et `STI` pour changer les valeurs du *flag* `IF`.

Remarque : `IRET` signifie *Interrupt return* qui rend la main au programme à la fin d'une exception ou d'une interruption. On utilise le suffixe `(Q)` pour les routines 64 bits, comme `IR(Q)` et `IRET(Q)`.

4 Mode réel et mode protégé

Cette partie du cours traite du mode réel et du mode protégé qui sont des modes de fonctionnement des processeurs *Intel* x86.

4.1 Avant le 80286

Le mode de fonctionnement protégé, souvent appelé **mode protégé** est disponible depuis le processeur 80286.

Ce mode offre des protections à deux niveaux :

— Un programme ne peut pas accéder à toute la mémoire (RAM) de l'ordinateur.
— Un programme ne peut pas toujours utiliser toutes les instructions du processeur.

Aujourd'hui, la *quasi* totalité des systèmes informatiques fonctionnent en mode protégé. Le mode réel existe toujours mais l'immense majorité du code tourne en mode protégé. À l'époque, certains programmes tournaient en mode réel et donc avaient accès à toute la mémoire et aux interruptions, ce qui représentait un risque de sécurité.

4.2 Mode protégé

Le mode protégé offre quatre niveaux de protection, appelés des anneaux de protection ou *rings*. À tout moment, le processeur travaille dans un des quatre niveaux de protection. À certains moments-clé, le processeur change de niveau de protection.

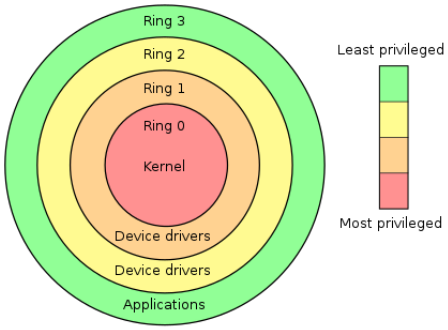
Les niveaux de protection sont numérotés de 0 à 3, du plus permissif au moins permissif.

Ring 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).

Ring 1 Les programmes ne peuvent employer qu'un nombre réduit d'instructions, et n'ont pas accès à toute la mémoire. (Ce ring n'est pas utilisé sous Windows et Linux).

Ring 2 Les programmes ne peuvent employer qu'un nombre réduit d'instructions et n'ont pas à accès à toute la mémoire. (Ce ring n'est pas utilisé Windows et Linux).

Ring 3 Les programmes ne peuvent employer qu'un nombre réduit d'instructions et n'ont pas accès à toute la mémoire (Ce mode est employé par les programmes utilisateurs sous Windows et Linux).



Voici par exemple une instruction interdite en mode protégé : l'instruction `CLI` sert à suspendre les interruptions en mettant le *flag* `IF` à 0. Elle n'est permise qu'en **ring 0**, donc seul l'OS peut l'employer. Ici, on essaie de l'utiliser dans un programme utilisateur qui tourne en ring 3, ce qui crée une erreur.

```
;cli asm
global main
section .text
main:
    cli
;; fin
    mov rax, 60
    mov rdi, 0
    syscall
```

4.2.1 Commentaire sur le mode réel

Malgré l'introduction du mode protégé, les processeurs 80286 et suivants peuvent encore travailler dans le 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 du démarrage de l'ordinateur, le processeur commence toujours à fonctionner en mode réel, puis l'OS le fait passer en mode protégé.

Attention, en mode réel la taille totale de la mémoire adressable est limitée à 1 MB (2^{20} bytes), et tous les programmes peuvent y écrire sans protection !

4.2.2 Gestion de la mémoire en mode protégé

Le mode protégé emploie 3 types d'adresses décrites ci-dessous.

1. Les adresses logiques

   Les adresses employées dans les programmes assembleur et vues par le processeur.

   Si on exécute les deux mêmes programmes dans `kdbg` et qu'on compare les adresses utilisées, on voit que les deux programmes utilisent les mêmes adresses en même temps ! Ce qui est impossible car deux programmes ne peuvent pas utiliser le même espace mémoire.

   En réalité, les adresses employées par les programmes sont des adresses logiques qui ne sont pas des véritables adresses physiques et qui ne correspondent donc pas à des adresses physiques de la mémoire centrale. On aura donc besoin d'un mécanisme de traduction qui fera le lien entre adressage logique et physique.

   Voici du code assembleur qui présente une variante de l'instruction `MOV` :

```
mov [adresse], valeur
```

Cette instruction place une valeur entre crochets à l'adresse mémoire spécifiée entre crochets. Par exemple, l'instruction suivante place la valeur de `ax` en mémoire centrale à l'adresse logique 10.

```
mov [10], ax
```

Attention, `10` est bien une adresse logique, et non la véritable adresse physique à laquelle sera placé le contenu du registre `ax`.

Ainsi, deux programmes différents peuvent effectuer `mov [10], ax` sans se marcher sur les pieds. Grâce au fait que l'OS traduit les deux adresses `10` vers des adresses **physiques** différentes.

2. Les adresses linéaires
   Issues d'une traduction des adresses logiques au travers du processus de **segmentation**.
3. Les adresses physiques
   Véritables adresses de la mémoire centrale.

4.3 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 de précise de chaque segment en mémoire est stockée dans une **table des segments**, elle tient donc une référence de l'emplacement des adresses physiques. Elle est aussi appelée **GDT** (*Global Description Table*).

Dans cette table des segments, on trouve 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).

4.3.1 Segments typiques d'un programme

Segments de code Contient les instructions du programme.

Segments de données Contient les variables globales du programmes (`.data`, `.rodata`, `.bss`).

Segments de pile Contient les variables locales du programme.

4.3.2 Registres de segment

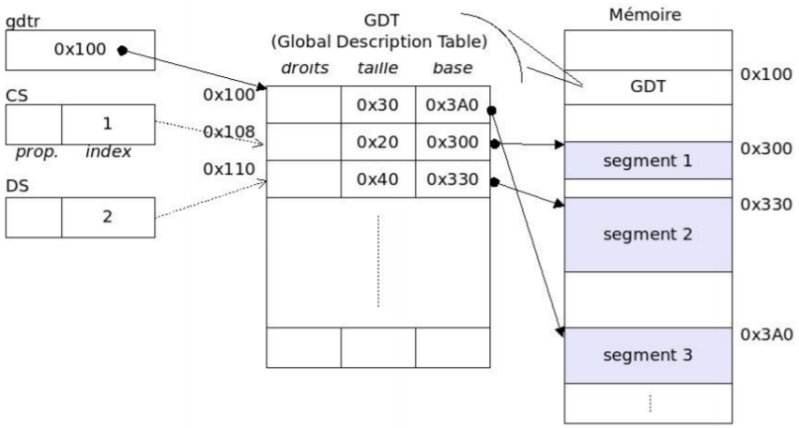
Le registre *Code selector* (CS) permet de retrouver l'adresse du segment de code dans la table des segments.

Le registre *Data selector* (DS) permet de retrouver l'adresse des segments de données dans la table des segments.

Le registre *Stack selector* (SS) permet de retrouver l'adresse du segment de pile dans la table des segments.

Ces registres (CS, DS, SS) indiquent en réalité où dans la la **table des segments** on doit aller chercher l'adresse du segment correspondant. Il y aussi d'autres registres de segment qui existent.

Voici une image qui représente l'utilisation des registres et la segmentation en mode protégé.



4.3.3 Traduction d'une adresse logique en une adresse linéaire

Les adresses logiques sont en réalités des décalages par rapport au début de leur segment. Il faut donc rajouter à chaque adresse logique l'adresse du début du segment correspondant.

$$\text{Adresse linéaire} = \text{adresse du début de segment} + \text{adresse logique}$$

Voici un exemple, on obtient l'adresse de début de segment de données grâce au registre `DS` et à la table des segments. On rajoute cette adresse à 10 pour obtenir l'**adresse linéaire**.

```
;; Mettre le contenu de ax à l'adresse 10 du segment de données
mov [10], ax
```

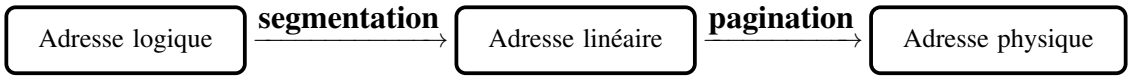
4.3.4 Pagination

La pagination est un deuxième mécanisme de gestion de la mémoire offert par le processeur x86 à partir de la version Intel 80386. Ce mécanisme 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.

Contrairement à la segmentation, la pagination peut être désactivée dans les processeurs 80386 et ceux venant après. Si on n'emploie pas la pagination, les adresses linéaires et les adresses physiques sont **égales**.

Voici un résumé des étapes de traduction des adresses.

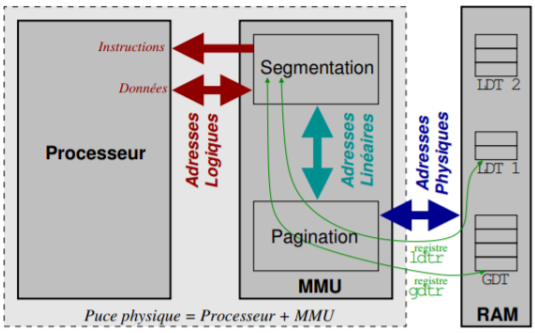


4.4 *Memory Management Unit*

Le processeur n'utilise que des adresses logiques dans ces registres (`RIP`). Ces adresses doivent être traduites en adresses physiques (segmentation, pagination) avant d'accéder véritablement à la mémoire.

La traduction d'adresse se fait grâce à une puce particulière appelée **MMU**, c'est l'unité de gestion de la mémoire ou *Memory Management Unit*.

Ici-bas, on voit une illustration du processeur et de la traduction des adresses.



4.5 Mode réel et segmentation

En mode réel, il y a une **segmentation**, mais très simplifiée. C'est une astuce pour contourner des limites techniques. Elle ne contient aucun mécanisme de la mémoire et ne se sert pas de table. Ce mode est utilisé pour démarrer le PC.

En mode réel, il n'y a pas de pagination donc la segmentation donne directement des adresses physiques.

→ Adresse physique = adresse de début du segment * 16 + offset.

Ici, l'adresse de début du segment et l'offset sont sur 16bits, ce qui donne une adresse totale 20bits.

Segment	A 2 5 F
Offset	5 2 A 6
Adresse	A 7 8 9 6

En mode réel, si on souhaite écrire à une adresse connue en mémoire, il suffit de choisir un segment et un *offset* qui mène à cette adresse. Inversement, si on a un segment et un offset, on peut calculer quelle adresse physique est obtenue.

On a à disposition les registres de segment de `code` (CS), de données `DS`, de pile (SS) et bien d'autres comme `ES`, `FS`, `GS`. Donc, tout programme peut accéder à n'importe quelle adresse physique en choisissant un segment et un offset qui mène à cette adresse.

En mode réel, un programme peut définir son propre segment grâce au registre `ES`. La segmentation en mode réel ne contient aucun mécanisme de protection de la mémoire. Ce même programme ne fonctionne pas en mode protégé, il y aura une **segmentation fault**. Car il faut être en **ring 0** pour accéder aux adresses `0xB000`.

Voici un exemple :

```
mov ax, 0xB800
mov es, ax
mov al, 'h'
mov ah, 10010111b
mov [es : 0xA0], ax
```

On donne la valeur de `0xB00` au registre de segment `ES`. On donne une valeur arbitraire à `EX` que l'on place aux positions `0xB80A0` et `0xB80A1` en mémoire.

4.6 Mode / Interruptions

Rappel : La table des interruptions est la table disant à quelle adresse se trouve la routine de gestion de l'interruption, pour chaque interruption. 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 manière différente.

En mode réel, les adresses sont données par une paire **segment** et **offset** où le segment et l'offset font chacun deux *bytes* et donc 4 *bytes* en tout. Toujours en mode réel, la table des interruptions est à l'adresse **fixe** `000:000` c'est-à-dire 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 d'interruption **i** se trouve donc à l'adresse **4*i** ;

En mode protégé, la table des interruptions **IDT** est à l'adresse donnée par le registre **IDTR** et l'entrée **i** de la table contient l'adresse de la routine de gestion de l'interruption n° **i**.

Toujours en mode protégé, les adresses sont données par une paire de segment et d'offset où :

en architecture 32 bit Le segment fait 2 *bytes* et l'offset fait 4 *bytes* donc 6 en tout.

en architecture 64 bit Le segment fait 2 *bytes* et l'offset fait 8 *bytes* donc 10 en tout.

L'architecture influence aussi les entrées de la table des interruptions.

En 32 bits Les 6 *bytes* de l'adresse de la routine de gestion d'interruption, ainsi que 2 *bytes* supplémentaires contenant d'autres informations de sécurité. Il y a donc 8 *bytes* en tout, par entrée de la table en mode protégé.

L'adresse de la routine de gestion de l'interruption **i** (le bout de code correspondant à **int i**) se trouve à l'adresse $[IDTR] + 8 * i$.

En 64 bits Les 10 *bytes* de l'adresse de la routine de gestion d'interruption, ainsi que 2 *bytes* supplémentaires contenant d'autres informations de sécurité. Il y a donc 12 *bytes* en tout, par entrée de la table en mode protégé.

L'adresse de la routine de gestion d'interruption **i** (le bout de code correspondant à **int i**) se trouve à l'adresse $[IDTR] + 12 * i$.

Pour rappel, en mode protégé le processeur peut travailler dans un des 4 **ring**.

ring 0	Peut exécuter toutes les instructions du jeu d'instructions Code d'interruption
ring 3	Ne peut exécuter qu'une partie des instructions du jeu d'instructions Code utilisateur Si interruption basculement en **ring 0** et **IRET** remet l'ancien ring

Lors d'une interruption :

— Le processeur termine l'instruction en cours.
— Il place le contenu du registre **RIP** au sommet de la pile.
— Il remplace **RIP** par une valeur située en mémoire, dans une table à l'index donné par le numéro de l'interruption.
— Le processeur bascule en **ring 0** s'il est en mode protégé pour pouvoir exécuter la routine d'interruption.

Puis, lorsqu'on rencontre l'instruction **IRET** :

— Le processeur récupère **RIP** sur la pile.
— Le processeur bascule dans l'ancien ring (par exemple le 3) s'il est en mode protégé.

5 Langage d'assemblage

Cette section traite du langage d'assemblage ou assembleur et de son fonctionnement.

L'assembleur possède plusieurs dialectes comme celui d'Intel ou de AT&T et il peut être aussi utilisé dans du code de haut niveau. À l'époque c'était le langage le plus performant car il était proche du langage machine mais aujourd'hui, les langages comme `C` ou `C++` ont des compilateurs suffisamment performants pour que ce ne soit plus le cas. Malgré tout, il reste très performant de par sa proximité avec le binaire.

On utilise l'assembleur dans une optique pédagogique, il permet de mieux comprendre la programmation. Il est néanmoins plus simple à utiliser que du langage machine direct à l'aide de texte, de mnémoniques et d'étiquettes, qui permettent de ne pas se soucier des calculs d'adresse.

Mais tout n'est pas simple en assembleur, il est difficile à écrire et maintenir et comme il est le reflet du jeu d'instructions du processeur, il y a autant d'assembleurs que de types d'architecture de processeurs.

Comme mentionné, voici quelques exemples de dialectes Intel et AT&T, respectivement.

```
mov eax, 5
mov ax, 5
mov al, 5
```

```
movl $5, %eax
movw $5, %ax
movb $5, %al
```

On peut aussi inclure de l'assembleur dans du code `C/C++`. Mais il faut utiliser le dialecte AT&T.

```
asm("movl %ebx, %eax"); /* moves the contents of ebx register to eax */
__asm__("movb %ch, (%ebx)"); /* moves the byte from ch to the memory pointed by ebx */
```

Bien qu'il soit assez vieux, il y a toujours des utilisations pour l'assembleur. Par exemple pour du code système : *drivers*, *handlers* d'interruption, `BIOS`, etc. Certaines portions du noyau Linux sont encore en assembleur. On utilise aussi l'assembleur pour les micro-contrôleurs et les systèmes embarqués même si ça se fait surtout en `C/C++`. Aussi, certaines applications qui nécessitent de lourds calculs et qui peuvent être optimisées avec des instructions particulières du processeur utilisent l'assembleur. Finalement, on peut aussi écrire des virus en assembleur.

Aujourd'hui, on utilise l'assembleur à des fins pédagogiques et pour des usages très particuliers qui nécessitent une optimisation fine mais on ne l'utilise plus pour des applications complètes.

6 Les modes d'adressage

On a souvent utiliser la manipulation de données dans le cours et dans les laboratoires. Maintenant, on peut se poser la question de ce que signifie **adresser** les données, ce qu'est l'adressage et les différentes manières de l'utiliser.

Il y a de nombreuses instructions pour manipuler les données, par exemple :

Transfert `mov rax, 0x34`

Calcul `add rax, rdx`

Test `bt r14, 0xA`

Il faut pouvoir indiquer où se trouve chaque donnée et où mettre chaque données. C'est ce qu'on appelle les modes d'adressage ou l'adressage de données.

6.1 Modes d'adressage de base

Voici quelques mots de vocabulaire et exemples utiles :

Immédiat La donnée est directement dans l'instruction, on fait directement référence à une valeur en décimal, octal, hexadécimal, binaire, etc.

```
mov r12, 0x445
```

Registre La donnée est ou doit être placée dans un registre.

```
mov rax, rbx
```

Direct La donnée est ou doit être placée à l'adresse donnée dans l'instruction.

```
mov rax, [0xB8A4]
mov [0xB8A0], rax
```

Indirect La donnée est ou doit être placée à l'adresse contenue dans le registre.

```
mov rax, [rcx]
mov [rcx], rax
```

Mais attention, il ne faut pas confondre une **adresse** avec le **contenu** d'une adresse. Par exemple :

```
1 mov rax, 0xB8A0 ; immédiat, RAX reçoit la valeur de 0xB8A0
2
3 mov rax, [0xB8A0] ; direct, RAX reçoit la valeur (8 bytes) à l'adresse 0xB8A0.
4
5 mov eax, [0xB8A0] ; direct, EAX reçoit la valeur (4 bytes) à l'adresse 0xB8A0.
6
7 mov ax, [0xB8A0] ; direct, AX reçoit la valeur (2 bytes) à l'adresse 0xB8A0.
8
9 mov al, [0xB8A0] ; direct, AL reçoit la valeur (1 byte) à l'adresse 0xB8A0.
```

Il faut aussi faire attention à l'utilisation des **labels** qui sont des noms symboliques pour une adresse.

```
1 mov rax, label1 ; immédiat, on met dans RAX l'adresse label1
2
3 mov rax, [label1] ; direct, RAX reçoit la valeur (8 bytes) à l'adresse label1.
4
5 mov al, [label1] ; direct, AL reçoit la valeur (1 byte) à l'adresse label1.
```

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

Par exemple $b \leftarrow a + 5$ pourrait se traduire par :

— `mov rax, [a]` : registre, direct
— `add rax, 5` : registre, immédiat
— `mov [b], rax` : direct, registre

Ceci est aussi valable pour les variables globales. En pratique, a et b sont probablement des variables locales. Elle seront dès lors stockées dans une pile, ce qui modifie un peu les instructions.

6.2 `RISC` vs `CISC`

Les processeurs peuvent avoir différentes architectures, typiquement les ordinateurs seront de type `CISC` et les *smartphones* de type `RISC`.

RISC *Reduced Instruction Set Computer*, utilise peu de modes d'adressage avec un processeur moins complexe.

CISC *Complex Instruction Set Compute*, utilise beaucoup de modes d'adressage avec un processeur plus complexe. L'architecture `CISC` permet de coder plus facilement des instructions de haut niveau.

6.3 Adressage indirect

Ici, on présente certains modes d'adressage indirect.

6.3.1 Registre

La donnée est une adresse donnée par un registre.

```
mov rbx, 0xB80A0 ; RBX contient l'adresse donnée

mov rax, [rbx] ; On met dans RAX la donnée (8 bytes) se trouvant à l'adresse 0xB8A0
```

Ce mode d'adressage est utile pour traduire les instructions manipulant les **points** et/ou les **références**.

6.3.2 Déplacement

L'adresse est obtenue en ajoutant un déplacement à une adresse dans un registre.

Par exemple, `mov rax, [rbx+4]` met dans `rax` la donnée se trouvant à l'adresse de `rbx` + 4. Comme on utilise le déréférencement, on a bien affaire à une donnée et pas une adresse.

Ce mode d'adressage est utile pour traduire les instructions manipulant des **structures**.

6.3.3 Indexé

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

Par exemple, `mov rax, [rbx+4*rcx]` met dans `rax` la donnée se trouvant à l'adresse de `rbx` + 4 × la valeur stockée dans `rcx`.

Ce mode d'adressage est utile pour traduire les instructions manipulant les **tableaux** comme le montre l'exemple suivant.

Imaginons un tableau `tab` pour lequel on veut `tab[3]` ← `1`.

```
mov rax, tab
mov rbx, 3
mov [rax + 8*rbx], 1 ; 8 représente la taille de la case du tableau
```

7 Le codage des instructions assembleur x86 64 bits

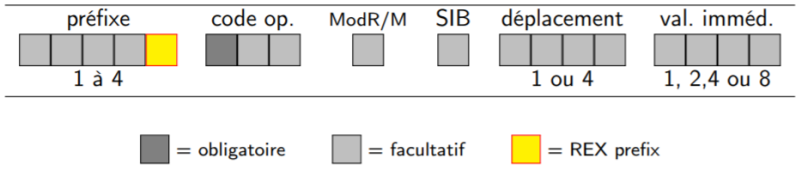
Cette section est consacrée à la conversion entre le code assembleur et le code machine et au format général d'une instruction, qu'on décomposera afin de mieux la comprendre.

7.1 Format général d'une instruction

En architecture 64 bits, le format général d'une instruction du `x86` se présente comme suit :

- les **préfixes** de l'instruction (1 à 4 *bytes* optionnels, 1 byte par préfixe).
- l'**opcode** de l'instruction (1 à 3 bytes, 1 obligatoire).
- le byte **ModR/M** (1 byte optionnel). Dans la table des codes opératoires, il est représenté par un `/r`. Il est utilisé pour les modes d'adressages complexes.
- le byte **SIB** (1 byte optionnel). Ça correspond à de l'adressage indexé.
- le **déplacement** (1 ou 4 bytes optionnels). Correspond à de l'adressage indirect avec déplacement.
- la valeur **immédiate** (1, 2, 4 ou 8 bytes optionnels).

Voici le format général d'une instruction sous forme schématisée. Ce qu'on voit c'est que le code opératoire est la seule partie obligatoire.



- Longueur variable en fonction de l'instruction
- Assez complexe ⇒ procédons par étapes

Le lien suivant montre les instructions pour pour l'instruction `MOV`. On voit que chaque cas de figure possible est repris, suivant qu'on effectue des déplacements avec des registres ou des immédiats et suivant la taille des éléments.

7.2 Les préfixes

Tout d'abord, il faut distinguer deux types de préfixes : les préfixes **hérités** et les préfixes `REX`.

Les préfixes hérités sont séparés en 4 groupes :

- groupe 1 : les préfixes de répétition (`0xF2`, `0xF3`). Par exemple les instructions qui se répètent.
- groupe 2 : les préfixes de réécriture des segments : CS(`0x2E`), SS(`0x36`), DS(`0x3E`), ES(`0x26`), FS(`0x64`), GS(`0x65`). On utilise ceux-ci quand on réécrit de l'information qui se trouve sur ces segments manuellement.
- groupe 3 : les préfixes de réécriture de la taille des opérandes (`0x66`). Par exemple quand on a des instructions qui échangent de l'information entre 32 et 64 bits ou 16 et 32.
- groupe 4 : les préfixes de réécriture de la taille des adresses (`0x67`).

Ensuite, les `REX` préfixes (`0100WRXB`) (en jaune sur le dessin précédent).

Ce sont des ensembles de 16 *opcodes* utilisés comme préfixes d'instruction en x86 64 bits. Toutes les instructions du x86 64 bits ne nécessitent pas un préfixe `REX`.

Enfin, `REX.W` peut être utilisé pour déterminer la taille des opérandes. Dans ce cas, le `.W` indique qu'en code machine, le `W` vaut 1. Donc, `01001RXB`. On définira plus tard les autres lettres.

7.3 Code opératoire

Le code opératoire ou *opcode* permet d'identifier l'opération à exécuter. Mais attention, pour un **même mnémonique** on peut avoir plusieurs *opcodes* associés. Par contre, pour un *opcode* donné, il n'y a qu'une et seule instruction associée.

7.4 Références

Des exemples et références sont donnés aux slides 165 à 167.

7.5 Code des registres

Dans les instructions du x86, les registres sont codés sur **3 bits**. Le code machine permet au processeur de déterminer s'il doit travailler des registres de 8, 16, 32 ou 64 bits.

Voici les codes qu'on retrouve et les registres correspondant :

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, RSI, R14L, R14W, R14D, R14 ;
111 BH, DI, EDI, RDI, R15L, R15W, R15D, R15 ;

Grâce à ça, on peut savoir quel registre est modifié ou inclus dans l'instruction. Comme on connaît aussi la taille de l'instruction avec les codes opératoires, on peut retomber sur le bon code de registre.

Par exemple, avec :

```
1 MOV RAX, 0x8877665544332211
```

L'instruction devient `48B8...` mais si on avait utilisé `RCX` dont le code est 001, on aurait eu `48BA....`

7.6 Environnement d'exécution

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**.

En **IA-32e**, il y a deux sous-modes :

— **Mode compatibilité** : qui permet l'exécution des applications 16 bits et 32 bits sur un OS 64 bits.
— **Mode 64 bits** : qui permet à l'OS 64 bits d'exécuter des applications écrites pour accéder à un espace linéaire 64 bits. Dans ce mode, la taille des adresses, par défaut, est de 64 bits et celle des opérandes des instructions est de 32 bits. L'utilisation des préfixes `REX` sous la forme `REX.R`, permet d'accéder aux registres `R8`, R9=, R10=, etc. Enfin, l'utilisation du préfixe `REX` (par exemple `REX.W`) promeut les opérations à 64 bits.

7.7 Code machine sur un byte

Certaines instructions ne tiennent que sur un *byte*, comme `CLC` (`F8`) ou `NOP` (`90`). C'est d'ailleurs le seul *byte* obligatoire dans l'**opcode**.

7.8 Code machine avec valeur immédiate

7.8.1 `MOV`

Voici un exemple de code avec l'opération `MOV` avec un immédiat. Dans ce cas, on a que l'opcode et la valeur immédiate.

```
1 MOV EAX, 20 ;; => B8 14 00 00 00, 32 bits
2 MOV RAX, 20 ;; => 48 B8 14 00 00 00 et on a une différence car elle est en 64 bits
```

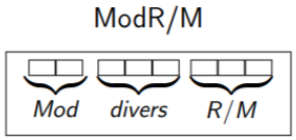
7.8.2 `ADD`

Dans cet exemple, on additionne le registre avec un immédiat.

```
1 ADD EAX, 10 ;; => 83 C0 0A
2 ADD RAX, 10 ;; => 48 83 C0 0A, encore une fois en 64 bits on a le REX préfixe de 48
```

7.9 Le byte `ModR/M`

Le `ModR/M` est un *byte* utilisé pour les modes d'adressages complexes. On le décompose de la manière suivante.



Avec,

Mod Le mode d'adressage pour la partie R/M.

R(egistre)/M(émoire) L'opérande, pouvant nécessiter d'autres *bytes*.

Divers Dépend du nombre d'opérandes.
— 1 opérande : extension du code opératoire.
— 2 opérandes : deuxième opérande, un registre.

7.9.1 Mode d'adressage sur 2 bits

Le mode d'adressage (`Mod`) est codé sur 2 bits :

— **00** : De registre à mémoire, par exemple : `MOV EAX, [EBX]`. L'adresse est dans le registre, sauf `ESP`, `RSP` qui sont réservés lors d'un adressage indirect indexé, exemple : `MOV RAX, [RBX + 4 * RCX]` et `EBP`, `RBP` qui sont réservés lors d'un adressage direct (déplacement sur 32 bits). Par exemple : `MOV EAX, [0xB8A0]`.
— **01** : De registre à mémoire, `MOV EAX, [EBX + 8]`. L'adresse est dans le registre plus un déplacement sur 1 *byte* sauf `ESP`, `RSP`.
— **10** : De registre à mémoire, comme `MOV EAX, [EBX+512]`. L'adresse est dans le registre plus un déplacement sur 4 *bytes*, sauf pour `ESP` et `RSP`.
— **11** : De registre à registre, comme `MOV EAX, EBX`.

À noter que `RSP`, `RBP` correspondent à du code en 64 bits et `ESP`, `EBP` à du code en 32 bits.

7.9.2 De registre à registre

Voici deux exemples utilisant des opérations de registre à registre. Dans les exemples suivants, on a `01/r`, ce qui signifie que le code opératoire est `01` et qu'il y a un `ModR/M`, indiqué par le `/r`. Enfin, il faut regarder la documentation de l'opération qu'on utilise (ici, `ADD`)) pour savoir qui joue le rôle de registre et qui joue le rôle de mémoire. Aussi, `/r` désigne un *byte* pour le `ModR/M`, il pourrait avoir d'autres valeurs précédés par un `/`, nous y reviendrons plus tard.

```
ADD EAX, EBX ; (01/r)

;; Op code : 01 (ADD)
;; Mode d'adressage : 11 (registre à registre)
;; Registre : EBX = 011
;; Mémoire : EAX = 000

ADD EAX, RBX ; => 01 1101 1000 => 01 D8
```

et

```
ADD RAX, EBX ; (REX.W + 01/r)

;; Préfixe REX sous la forme REX.W :01001000 = 48
;; Op code : 01 (ADD)
;; Mode d'adressage : 11 (registre à registre)
;; Registre : RBX = 011
;; Mémoire : RAX = 000

ADD RAX, RBX ; => 0100 1000 01 1101 1000 => 48 01 D8
```

7.9.3 De registre à mémoire

Voici un autre exemple de registre à mémoire, avec l'adresse dans le registre.

```
ADD [EAX], [EBX] ; (01/r)

;; Op code : 01 (ADD et indiqué au-dessus)
;; Mode d'adressage : 00 (registre à mémoire)
;; Registre : EBX = 011
;; Mémoire : EAX = 000

ADD [EAX], EBX ; => 01 0001 1000 => 01 18
```

Maintenant, en 64 bits.

```
ADD [RAX], [RBX] ; (01/r)

;; Préfixe REX sous la forme REX.W : 01001000 = 48
;; Op code : 01 (ADD et indiqué au-dessus)
;; Mode d'adressage : 00 (registre à mémoire)
;; Registre : RBX = 011
;; Mémoire : RAX = 000

ADD [RAX], RBX ; => 0100 1000 01 0001 1000 => 48 01 18
```

Voilà un autre exemple avec adresse dans le registre et un déplacement sur 1 *byte*.

Attention, l'adresse ne peut **pas** être dans `ESP` ou `RSP`. Voici un exemple :

```
ADD [ECX + 0x40], ESI ; (01/r)

;; Op code : 01 (ADD) et précisé dans l'énoncé
;; Mode d'adressage : 01 (registre à mémoire)
;; Registre : ESI = 110
```

```
;; Mémoire : ECX = 001
;; Déplacement : 0x40

ADD [ECX + 0x40], ESI ; 01 0111 0001 0100 0000 => 01 71 40
```

Et un autre exemple sur 64 bits :

```
ADD [RCX + 0x40], RSI ; (01/r)

;; Préfixe REX sous la forme REX.W : 01001000 = 48
;; Op code : 01 (ADD) et précisé dans l'énoncé
;; Mode d'adressage : 01 (registre à mémoire)
;; Registre : RSI = 110
;; Mémoire : RCX = 001
;; Déplacement : 0x40

ADD [RCX + 0x40], RSI ; 0100 1000 01 0111 0001 0100 0000 => 48 01 71 40
```

Enfin, nous arrivons au dernier cas de figure avec une opération de registre à mémoire, l'adresse se trouvant dans le registre avec un déplacement de 4 *bytes*.

Attention, l'adresse ne peut pas être dans ESP ou RSP.

```
ADD [ECX + 0x00001234], ESI ; (01/r)

;; Op code : 01
;; Mode d'adressage : 10
;; Registre : ESI = 110
;; Mémoire : ECX = 001
;; Déplacement en little endian : 0x34120000

ADD [ECX + 0x1234], ESI ; => 01 1011 0001 0011 0100 0001 0010 0000 0000 => 01 B1 34 12 00 00
```

Et en 64 bits.

```
ADD [RCX + 0x00001234], RSI ; (01/r)

;; Préfixe sur la forme REX.W => 01001000 = 48
;; Op code : 01
;; Mode d'adressage : 10
;; Registre : RSI = 110
;; Mémoire : RCX = 001
;; Déplacement en little endian : 0x34120000

ADD [RCX + 0x1234], RSI ; => 48 01 B1 34 12 00 00
```

7.9.4 De mémoire à registre

Maintenant on peut passer de mémoire vers le registre. Attention, l'adresse ne peut pas être dans ESP ou RSP.

```
ADD ESI, [ECX + 0x1234] ; (03/r)

;; Op code : 03
;; Mode d'adressage : 10
;; Registre : ESI = 110
;; Mémoire : ECX =  001
;; Déplacement, en little endian : 0x3412

ADD ESI, [ECX + 0x1234] ; => 03 1011 0001 ... => 03 B1 34 12 00 00
```

Et en 64 bits :

```
ADD RSI, [RCX + 0x1234] ; (03/r)

;; Préfixe sous la forme REX.W : 01001000 = 48
;; Op code : 03
;; Mode d'adressage : 10
;; Registre : RSI = 110
;; Mémoire : RCX = 001
;; Déplacement, en little endian : 0x3412

ADD RSI, [ECX + 0x1234] ; => 03 1011 0001 ... => 48 03 B1 34 12 00 00
```

7.9.5 De registre à mémoire avec adresse fixe sur 4 ou 8 *bytes*

Attention, on utilise le registre `EBP` ou `RBP` pour la mémoire.

Voici un exemple :

```
ADD [0x1234], ESI ; (01/r)

;; Op code : 01
;; Mode d'adressage : 00
;; Registre : ESI = 110
;; Mémoire : EBP = 101
;; Déplacement en little endian : 34120000

ADD [0x1234], ESI ; => 01 0011 0101 ... => 01 35 34 12 00 00
```

Et en 64 bits :

```
ADD [0x1234], RSI ; (01/r)

;; Préfixe sous la forme REX.W : 01001000 = 48
;; Op code : 01
;; Mode d'adressage : 00
;; Registre : RSI = 110
;; Mémoire : RBP = 101
;; Déplacement en little endian : 34120000

ADD [0x1234], RSI ; => 48 01 0011 0101 ... => 48 01 35 34 12 00 00
```

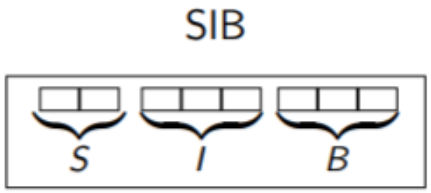
7.9.6 Récapitulatif

Voici un tableau récapitulatif pour le `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

7.10 Le byte `SIB`

Ce *byte* est utilisé en complément au `ModR/M` et est utilisé pour les modes d'adressage indexés $B + S \times I$.



Où :

S facteur multiplicatif (*scale*) :
$2^i \Rightarrow 00 = 1\times, 01 = 2\times, 10 = 4\times, 11 = 8\times$

I Registre d'index

B Registre de base

Exemple de registre à mémoire :

— L'adresse est fournie par un index, une base et un déplacement sur 1 ou 4 *bytes*.
— Le *byte* `ModR/M` spécifie s'il y a un déplacement et sur quelle taille.
— Le code du registre **ESP** ou **RSP** (`100`) est utilisé pour préciser que l'adresse est calculée sur base d'un index et d'une base.

Dans ce cas, le *byte* `ModR/M` est suivi du *byte* `SIB`.

Voici quelques exemples :

— `ADD [ECX + 2*EAX], EBX`
⇒ 01 1C 41
— `ADD [RCX + 2*RAX], RBX`
⇒ 48 01 1C 41
— `ADD [EDX + 4*ECX + 0x1234], ESI`
⇒ 01 B4 8A 34 12 00 00
— `ADD [RDX + 4*RCX + 0x1244], RSI`
⇒ 48 01 B4 8A 34 12 00 00

7.11 Les préfixes

7.11.1 *Legacy* préfixes

Il y a des préfixes de **segmentation**, par exemple :

— `ADD [ES :EAX], EBX`
⇒ 26 01 18
— `ADD [SS :ECX], ESI`
⇒ 36 01 31

On a aussi des préfixes de **répétition**, par exemple la commande `MOVSB` copie un *byte* de l'adresse `[ESI]` vers l'adresse `[EDI]` et incrémente `ESI` et `EDI`. Cette instruction peut être répétée `ECX` fois à l'aide du préfixe `REP` codé `0xF3`.

— `MOVSB`
⇒ A4
— `REP MOVSB`
⇒ F3 A4

Ensuite, on a des préfixes de **taille**. D'abord on doit préciser dans le fichier assembleur quelle taille de registre on utilise, avec les commandes `[BITS 16]`, `[BITS 32]` ou `[BITS 64]`. Puis un bit `D` qui se trouve dans le descripteur de segment précise au processeur s'il doit utiliser le registre 16 bits (D=0) ou 32 bits (D=1).

Aussi, pour une instruction, on peut changer la taille par défaut à l'aide du préfixe `0x66` pour les opérandes et `0x67` pour les adresses.

Voici un exemple :

```
1 [BITS 16]
2
3 ADD [EAX], EBX ; => 67 66 01 18
```

```
;; Car on est en mode 16 bits mais on utilise une adresse sur 32 bits
;; ET un registre opérande sur 32 bits
```

et un autre exemple :

```
[BITS 16]

MOV EBX, 0 ; => 66 BB 00 00 00 00
;; 66 en début d'instruction car l'instruction est en 32 bits

MOV ECX, [EBP] ; => 67 66 8B 4D 00
;; 67 car l'adresse sur laquelle pointe EBP n'est pas en 16 bits
;; mais en 32 bits. 66 car l'instruction est en 32 bits.
```

Et quelques exemples de préfixes de taille en plus :

Instructions	[BITS 16]	[BITS 32]	[BITS 64]
ADD[EBX],EAX	66 67 01 03	01 03	67 01 03
ADD[EBX],AX	67 01 03	66 01 03	66 67 01 03
ADD[BX],AX	01 07	66 67 01 07	Impossible
ADD[BX],AL	00 07	67 00 07	Impossible
ADD[EBX],AL	67 00 03	00 03	67 00 03
ADD[RBX],AL	Impossible	Impossible	00 03

7.11.2 `REX` préfixes (`0100WRXB`)

Les préfixes `REX` sont des préfixes d'instruction utilisés en mode 64 bits. Ils permettent entre autre de spécifier la taille des opérandes à 64 bits.

Toute les instructions ne nécessitent pas de préfixe `REX` en mode 64 bits. Il y a **un** et **un seul** préfixe `REX` par instruction. Enfin, le *byte* préfixe `REX` précède immédiatement le *byte* opcode.

Nous devons apporter quelques précisions sur les champs du préfixe `REX`.

- 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.
- Le bit `REX.R` modifie le champ **reg** (partie divers) du *byte* `ModR/M` quand ce champ encode les registres `R8` à `R15`.
- Le bit `REX.R` est ignoré sur `ModR/M` spécifie d'autres registres ou défini un extension de l'opcode.
- Le bit `REX.X` modifie le champ **index** du **SIB**.
- 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` à `R15`.

1. Exemple : Adressage par registre
   Information nécessaire pour comprendre l'instruction :

REX.W + 8B/r	MOV r64, r/m64
REX.W + 89/r	MOV r/m64, r64

```
MOV RAX, R9 ; (REX.W + 89/r)

;; Opération 64 bits => REX.W = 1
;; SIB n'est pas utilisé => REX.X = 0
;; La base du champ r/m du ModR/M = RAX => REX.B = 0
;; Le champ reg de ModR/M = R9 => REX.R = 1
;; => REX = 01001100b = 0x4C

;; REX = 0x4C
```

```
;; Opcode : 0x89
;; ModR/M ->
;; 1. Mode d'adressage = 11b
;; 2. Registre R9 = 001b
;; 3. Mémoire (r/m) = RAX = 000b
;; ModR/M => 1100 1000 => 0xC8

MOV RAX, R9 ; => 0x4C 89 C8
```

2. Exemple : Adressage par registre
   Information nécessaire pour comprendre l'instruction :

REX.W + 8B/r	MOV r64, r/m64
REX.W + 89/r	MOV r/m64, r64

```
MOV R9, RAX ; (REX.W + 89/r)

;; Opération à 64 bits => REX.W = 1
;; SIB n'est pas utilisé => REX.X = 0
;; La base du champ r/m du ModR/M (la parité mémoire)
;; est égale à R9 => REX.B = 1
;; Le champ reg de ModR/M = RAX => REX.R = 0
;; REX = 01001001b = 0x49

;; REX = 0x49
;; Opcode = 0x89
;; ModR/M ->
;; 1. Mode d'adressage = 11b
;; 2. Registre = RAX = 000b
;; 3. Mémoire (r/m) = R9 = 001b

;; ModR/M = 1100 0001 => 0xC1

MOV R9, RAX ; => 0x49 89 C1
```

3. Exemple : Adressage indirect
   Information nécessaire pour comprendre l'instruction :

REX.W + 8B/r	MOV r64, r/m64

```
MOV RAX, [R9] ; (REX.W + 8B/r)

;; Opération à 64 bits => REX.W = 1
;; SIB n'est pas utilisé : REX.X = 0
;; La base du champ r/m du ModR/M = R9 => REX.B = 1
;; Le champ reg du ModR/M = RAX => REX.R = 0
;; REX = 01001001b = 0x49

;; REX = 0x49
;; Opcode = 0x8B
;; ModR/M ->
;; 1. Mode d'adressage = 00b (indirect)
;; 2. Registre = RAX = 000b
;; 3. Mémoire (r/m) = R9 = 001b
;; ModR/M = 0000 0001 = 0x01

MOV RAX, [R9] ; => 49 8B 01
```

4. Adressage indirect indexé
   Information nécessaire pour comprendre l'instruction :

REX.W + 8B/r	MOV r64, r/m64

```
MOV RAX, [R9 + 2*RDX] ; (REX.W + 8B/r)

;; Opération 64 bits => REX.W = 1
;; SIB est utilisé (car indexation) => REX.X = 0 mais pas d'utilisation de R8..R15
;; La base du champ r/m du ModR/M = R9 => REX.B = 1
;; Le champ reg du ModR/M = RAX => REX.R = 0
;; REX = 0100 1001 = 0x49

;; REX = 0x49
;; Opcode = 0x8B
;; ModR/M ->
;; 1. Mode d'adressage = 00b (indirect indexé)
;; 2. Registre = RAX = 000b
;; 3. Mémoire (r/m) = RSP = 100b
;; ModR/M = 0000 0100b = 0x04

;; SIB :
;; S : facteur multiplicatif 01b car vaut 2^1
;; I : registre d'index = RDX = 010b
;; B : registre base = R9 = 001b
;; SIB => 0101 0001 = 0x51

MOV RAX, [R9 + 2*RDX] ; => 49 8B 04 51
```

5. Exemple : Adressage indirect indexé

```
MOV RBX, [R9 + 2*R8] ; (REX.W + 8B/r)

;; Opération sur 64 bits : REX.W = 1
;; SIB est utilisé (et avec un registre R[8-15]) => REX.X = 1
;; Base du champ r/m du ModR/M = R9 => REX.B = 1
;; Le champ reg du ModR/M = RBX => REX.R = 0
;; REX = 0100 1011b = 0x4B

;; REX = 0x4B
;; Opcode = 8B
;; ModR/M ->
;; 1. Mode d'adressage = 00b (indirect indexé)
;; 2. Registre = RBX = 011b
;; 3. Mémoire (r/m) = RSP = 100b
;; ModR/M = 0001 1100 = 0x1C

;; SIB :
;; S : facteur multiplicatif = 01b car vaut 2^1
;; I : registre d'index = R8 = 000b
;; B : registre base = R9 = 001b
;; SIB => 0100 0001 = 0x41

MOV RBX, [R9 + 2*R8] ; => 4B 8B 1C 41
```

6. Adressage indirect indexé

```
MOV R10, [R9 + 2*R8] ; (REX.W + 8B/r)

;; Opération sur 64 bits : REX.W = 1
;; SIB est utilisé (et avec un registre R[8-15]) => REX.X = 1
;; Base du champ r/m du ModR/M = R9 => REX.B = 1
;; Le champ reg du ModR/M = R10 => REX.R = 1
;; REX = 0100 1111b = 0x4F

;; REX = 0x4F
;; Opcode = 8B
;; ModR/M ->
```

```
;; 1. Mode d'adressage = 00b (indirect indexé)
;; 2. Registre = R10 = 010b
;; 3. Mémoire (r/m) = RSP = 100b
;; ModR/M = 0001 0100 = 0x14

;; SIB :
;; S : facteur multiplicatif = 01b car vaut 2^1
;; I : registre d'index = R8 = 000b
;; B : registre base = R9 = 001b
;; SIB => 0100 0001 = 0x41

MOV RBX, [R9 + 2*R8] ; => 4F 8B 14 41
```

7.12 Comment visualiser le code machine ?

Pour visualiser un code machine en mode 64 bits donc avec `bits 64` écrit dans un fichier d'extension `.asm`, on doit le compiler en binaire et puis utiliser un programme permettant d'afficher le binaire.

On le compile avec :

```
nasm monprog.asm -o monprog.bin -f bin
```

et on visualise le contenu avec :

```
od -tx1 monprog.bin
```

8 Cartographie de la mémoire en mode réel

8.1 ROM

8.2 RAM

8.3 Périphériques

8.4 IN/OUT

8.5 MOV

8.6 Cartographie de la mémoire RAM d'un PC

8.7 Exemple d'accès à un périphérique

9 Démarrage d'un ordinateur

9.1 Mise sous tension

9.2 BIOS

9.3 Secteur de *boot*

9.4 Un *process*

10 Coprocesseur mathématique

10.1 Intérêts et format des données

10.2 Les registres de données

10.3 Les registres spéciaux

10.3.1 TW

10.3.2 CW

10.3.3 SW