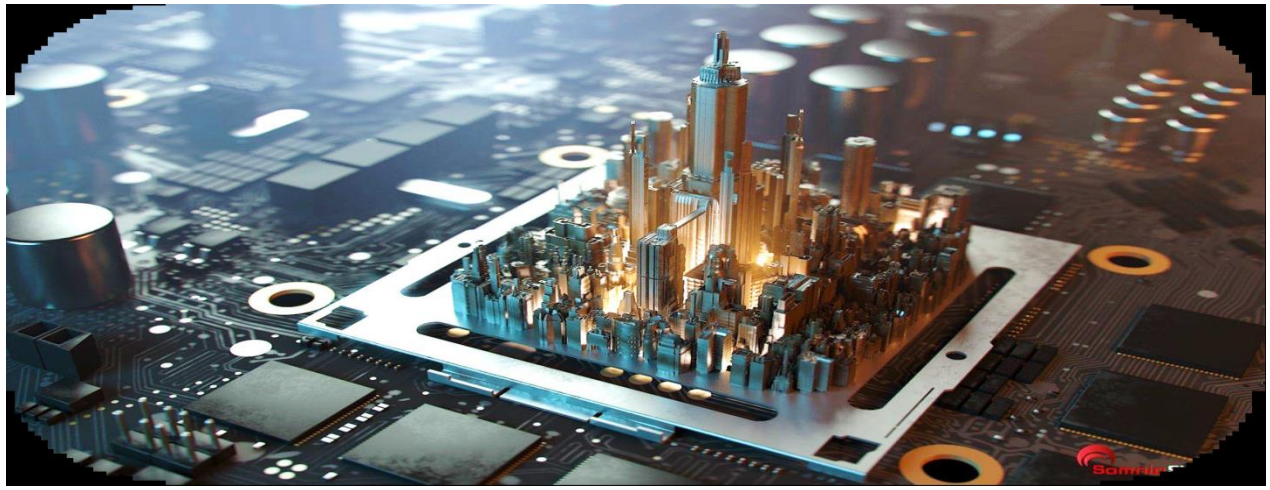
Université A. Mira Bejaia
Faculté des sciences exactess
Département d'informatique

Cours 4: Étude de cas «Processeurs 32 et 64 bits»



NIVEAU: INGÉNIEUR I ANNÉE

PRÉSENTÉE PAR: DR. SAAD NARIMANE

2024/2025

Introduction

- Les deux architectures les plus populaires connues pour les microprocesseurs (puces) sont IA-32 et AMD64 ou **32 et 64 bits**,
- L'architecture **IA-32** a été développée dans les premiers temps du PC, cette architecture présentait certains inconvénients,
- La seconde, **AMD64**, est moderne et a été créée relativement récemment
- Les utilisateurs qui viennent d'acquérir un ordinateur se demandent souvent ce qui est le mieux : 32 ou 64 bits ?
- Quelle architecture est préférable pour un PC ?

I. Evolution des architectures (IA-32 à AMD64)

I.1. Qu'est-ce que le débit binaire ?

- Le débit binaire de Windows = l'architecture du processeur,
- Détermine le nombre de bits d'information que le **processeur** peut traiter en même temps.
- Les versions 32 bits et 64 bits du système d'exploitation Windows.

➢ **Windows 32 bits :**

✓ Les processeurs et les systèmes d'exploitation dotés d'une architecture 32 bits peuvent **traiter des informations sur 32 bits à la fois.**

✓ Cela signifie qu'ils peuvent adresser jusqu'à **4 giga-octets de mémoire vive** et exécuter des **programmes sur 32 bits**.

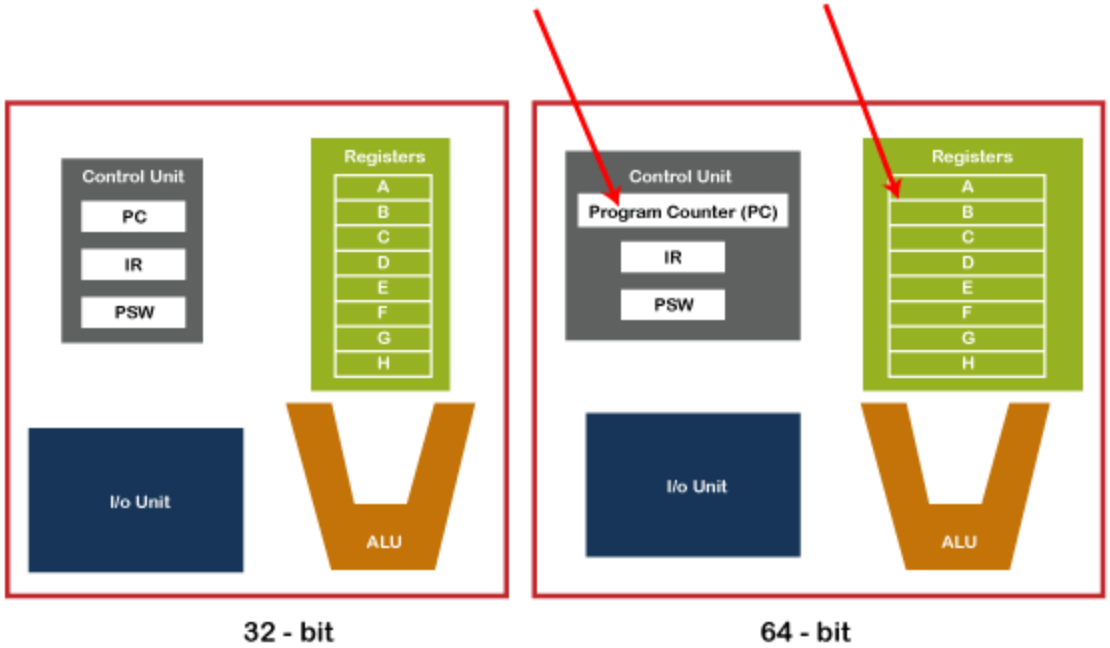
➢ **Windows 64 bits :**

✓ Les processeurs et les systèmes d'exploitation de l'architecture 64 bits peuvent **traiter des informations sur 64 bits à la fois**.

✓ Cela leur permet d'adresser beaucoup plus de mémoire vive (**plus de 4 giga-octets**) et d'offrir de **meilleures performances** lors de **l'exécution d'applications 64 bits**.

Remarque :

- Le choix entre les versions **32 bits** et **64 bits** de **Windows** dépend de vos exigences en matière d'utilisation **du processeur** et de **la mémoire vive**.
- Actuellement, la plupart des nouveaux ordinateurs et portables sont livrés avec **Windows 64 bits** en raison de sa capacité à utiliser efficacement de grandes quantités de mémoire vive et à prendre en charge les applications modernes



- **Exemples de systèmes d'exploitation 64 bits:**
- Les versions **64 bits** de **Windows XP** et **Vista** étaient bien moins nombreuses lorsqu'elles étaient populaires,
- **L'architecture 64 bits** peut fonctionner sur de nombreux systèmes d'exploitation tels que: Windows 10, Windows 8, Windows 7, ainsi que Windows Vista et Windows XP.
- **Exemples de processeurs 64 bits :**

✓ Fusion, FX, Phenom, Turion 64, AMD Opteron, Sempron et Athlon 64.

✓ Les processeurs Intel Celeron et Pentium 4.

✓ Les processeurs Intel Pentium double cœur, Core i3, Core i5 et Core i7……

- ***1.2. Présentation de l'architecture 32 et 64 bits :***

➢ **L'architecture 32 bits**

- L'architecture 32 bits ou 80386 ou IA-32 est une **architecture de processeur identique *avec*** laquelle **le système d'exploitation** est conçu ***pour*** **fonctionnerx86** a été utilisé pour la première fois pour les puces fabriquées par Intel.
- **L'architecture a reçu le nom** correspondant en raison **des premiers processeurs** avec lesquels elle a été utilisée **-** Intel 80386.
- Par la suite, elle a trouvé sa place chez **AMD et x86** est devenu un standard pour les PC. L'architecture a été **constamment améliorée et finalisée.**

L’architecture 64 bits

- **L'architecture 64 bits** est apparue plus tard qu'**AMD**
- Elle est également appelée **x86-64** ou **AMD64.**
- Elle fonctionne avec les puces **Intel** et **AMD**.
- En même temps, elle est **considérée** comme entièrement **compatible avec l'architecture x32.**
- La principale différence entre les deux architectures **32 et 64 bits** est le **nombre de bits.**

1.3. Les différences entre l'architecture 32 et 64 bits

- Un **microprocesseur** (**une puce**) : le composant le plus important de l'ordinateur;le cerveau du PC.
- Le processeur **gère les données** à traiter, **contrôle les périphériques externes**, leur **envoie des commandes, reçoit des données et interagit avec la mémoire.**
- Toutes les adresses et instructions exécutées **par le processeur** doivent être stockées dans la mémoire.
- Une puce de processeur est nécessaire pour **accomplir ces tâches**.

Quelle est la différence générale entre l'architecture 32 bits et 64 bits ?

- Dans les processeurs **32 bits**, la taille des cellules est de 32 bits. Dans les puces à architecture **64 bits**, la taille est de 64.
- Plus **la cellule est grande**, **plus elle peut contenir de données**.

- Les puces à architecture **32 bits** sont limitées en ce sens qu'elles ne peuvent obtenir des adresses que dans la limite de 2^{32} degrés.
- Une adresse plus grande ne tient tout simplement pas dans la cellule..Puisque seulement 2^{32} **bits** ou **4 GB** de mémoire sont dans ces limites
- Une puce avec une taille de registre de **64 bits** est déjà orientée pour travailler avec des adresses de 2^{64}**= 1 milliard de Go.**

- **Aucun système d'exploitation** moderne ne peut prendre en charge une telle quantité de mémoire vive Même le populaire **Linux** n'est pas conçu pour cela. « C'est le microprocesseur qu'il va le faire pour le traitement »
- Une puce dotée d'un système **x32 bits** peut traiter **32 bits ou 4 octets de données**.
- Si **la taille des données est supérieure à 4 octets**, la puce doit exécuter plusieurs cycles simultanément pour les traiter.
- Si **la puce est de 64 bits**, **la taille des données à traiter sera multipliée par deux et sera égale à 8 octets**.(La puce devra consacrer moins de temps à la tâche à accomplir, à savoir le traitement des données)

Différences principales : Quelle est la meilleure taille de bit ?

Quelle taille de bit du système d'exploitation est la meilleure dépend de vos besoins et des exigences de performance de votre ordinateur. Voici quelques facteurs qui peuvent vous aider à prendre une décision :

- ✓ ***Performances :*** les systèmes **64 bits** sont capables de traiter de grandes quantités de données de manière plus efficace, ce qu peut améliorer les performances lors de l'exécution des tâches et d'applications complexes.
- ✓ ***Prise en charge de la mémoire :*** un système **64 bits** peut adresser beaucoup plus de mémoire vive qu'un système **32 bits** Si vous devez utiliser de grandes quantités de mémoire vive (plus de 4 Go), **un système 64 bits** sera un meilleur choix.
- ✓ ***Compatibilité logicielle :*** certaines applications peuvent être **optimisées** pour fonctionner sur des systèmes **64 bits**, ce qui peut affecter leurs performances et leurs fonctionnalités.
- ✓ ***Mises à jour futures :*** à mesure que **la technologie** et **les logiciels** évoluent, il est probable que de plus en plus d'applications fonctionneront mieux sur les systèmes 64 bits.

Remarque : Si vous avez la possibilité d'utiliser une version **64 bits** de Windows et que votre ordinateur prend en charge cette capacité de bits, il s'agit probablement ***d'un meilleur choix en raison de:***

- ***L'amélioration des performances***
- ***Et de la possibilité d'utiliser plus de mémoire vive***.

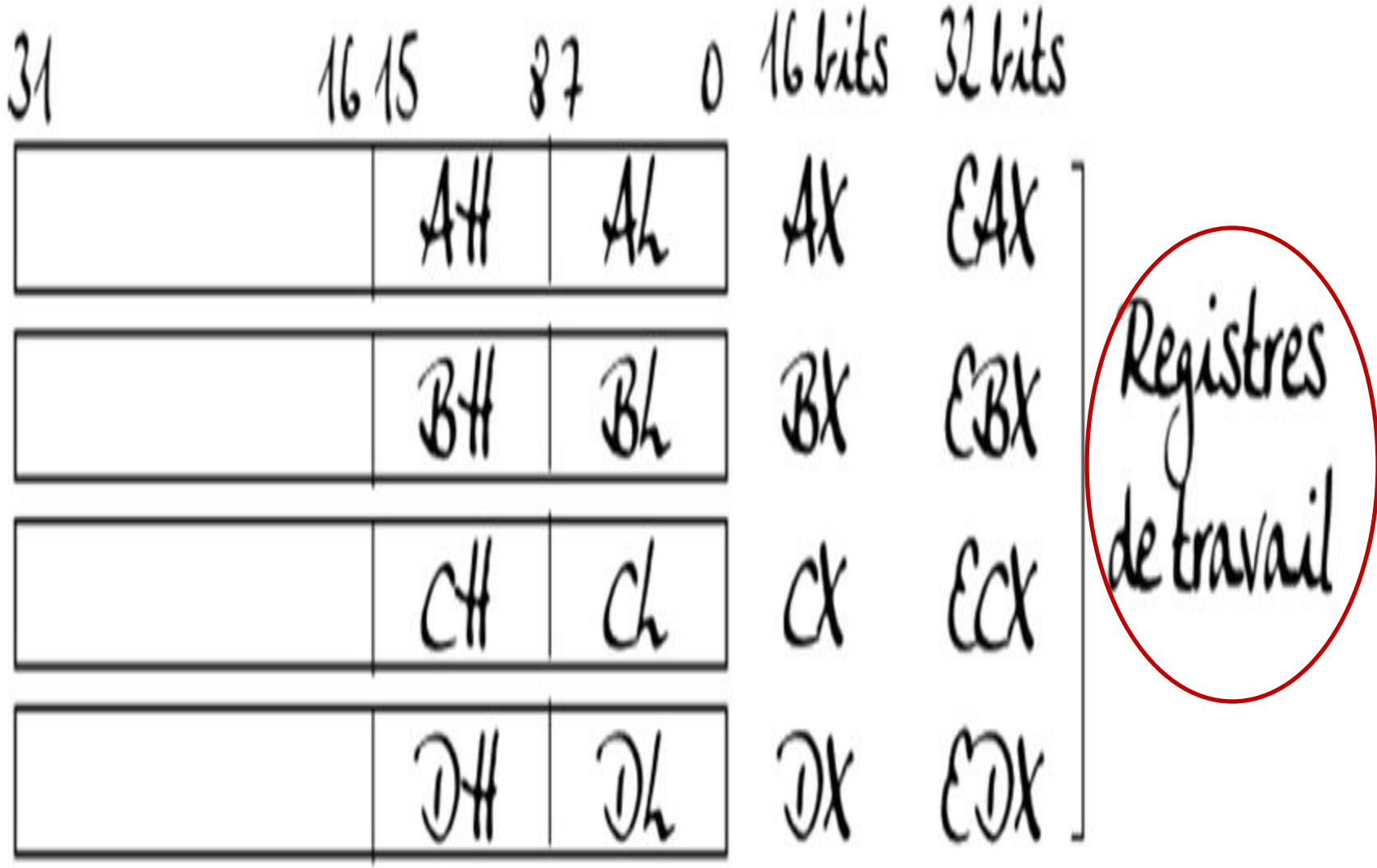
2. Registres dans les processeurs 32 et 64 bits

- Le modèle de programmation du **80386** et des versions supérieures contient des registres étendus de **8**, **16** et **32 bits**, ainsi que deux ***registres de segments*** supplémentaires de **16 bits** : **FS** et **GS**.
- Certaines parties des registres sont accessibles à partir des identifiants suivants :

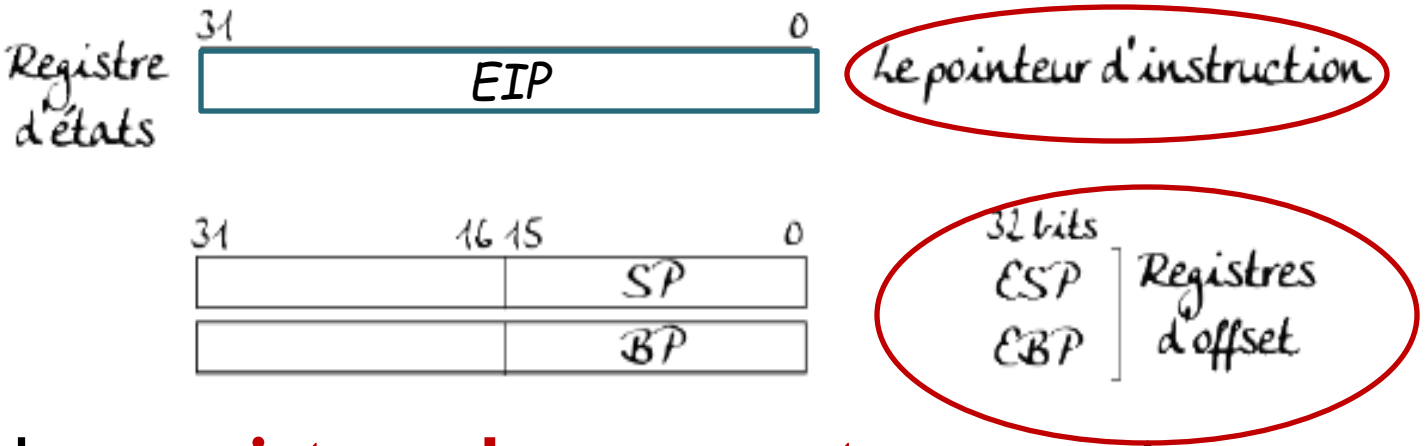
rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp, r8, r9, ... ,r15	64 bits
eax, ebx, ecx, edx, edi, esi, ebp, esp, r8d, r9d, ... ,r15d	32 bits

2.1.Les principaux registres 32 bits

- Les processeurs 80386 et plus récents ont des registres étendus.
- **Par exemple:** le registre **AX 16 bits** est étendu à 32 bits.
- Les registres **16 bits** sont AX, BX, CX, DX, SP, BP, DI et SI
- Les registres **étendus 32 bits** sont EAX, EBX, ECX, EDX, ESP, EBP, EDI et ESI.
- Pour la **compatibilité ascendante** :
- AX; AL et AH existent toujours,
- Font référence à des parties de **EAX (accumulateur),**
- Pas d'accès aux 16 bits de poids fort de **EAX**


31
16
15
8
7
0
16 bits
32 bits
AH
AL
AX
EAX
BH
BL
BX
EBX
CH
CL
CX
ECX
DH
DL
DX
EDX
Registres de travail

- Avec extensions **32 bits: BP** devient **EBP** ; **SP** devient **ESP** ; **FLAGS** devient **EFLAGS**
- Et **IP** devient **EIP**



- Les **registres de segment** sont toujours sur **16 bits** dans le **80386**, et ***2 nouveaux registres temporaires de segment*** : **FS et GS**.

CF Carry Flag (bit 0)	retenue
PF Parity Flag (bit 2)	
AF Auxiliary Carry Flag (bit 4)	
ZF Zero Flag (bit 6)	vaut 1 lorsque le résultat est 0
SF Sign Flag (bit 7)	bit de signe du résultat
OF Overflow Flag (bit 11)	dépassement, le résultat contient trop de bits
DF Direction Flag (bit 10)	sens d'incrémentation de ESI et EDI
TF Task Flag (bit 8)	active la gestion de tâche en mode protégé
IF Interrupt Flag (bit 9)	interruption
IOPL I/O Privilege Level (bits 12 et 13)	
NT Nested Task (bit 14)	
RF Resume Flag (bit 16)	active le mode debug
VM Vitrual 8086 Mode (bit 17)	

Le registre **EFLAGS** contient des **informations** concernant le résultat de l'exécution d'une instruction

2.2. Les principaux registres 64 bits

- Les registres **64 bits** d'un ***Pentium 4*** avec extensions **64 bits** sont **RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI** et **R8 à R15**.
- La plupart des autres registres sont **également étendus**: **EBP, ESP, EFLAGS, EIP**
- Avec extensions **64 bits: BP** devient **RBP** ; **SP** devient **RSP** ; **FLAGS** devient **RFLAGS,** et **IP** devient **RIP**.
- Mais plus d'accès aux registres **16 bits** en **mode protégé de l'architecture 32 bits**

rax	registre général, accumulateur, contient la valeur de retour des fonctions
rbx	registre général
rcx	registre général, compteur de boucle
rdx	registre général, partie haute d'une valeur 128 bits
rsi	registre général, adresse source pour déplacement ou comparaison
rdi	registre général, adresse destination pour déplacement ou comparaison
rsp	registre général, pointeur de pile (stack pointer)
rbp	registre général, pointeur de base (base pointer)
r8	registre général
r9	registre général
⋮	
r15	registre général
rip	compteur de programme (instruction pointer)

Le registre RFLAGS

- Le registre **RFLAGS** contient des **informations** concernant le résultat de l'exécution d'une instruction.
- Certains **bits** du registre appelés **drapeaux** ont une signification particulière.
- Seuls les 32 bits de la partie EFLAGS **sont utilisées.**
- De plus, le microprocesseur **64 bits** contient un pointeur d'instruction (IP/EIP/RIP) et un registre d'indicateurs (FLAGS, EFLAGS ou RFLAGS).
- **Toutes les adresses mémoire en mode réel sont une combinaison d'une adresse de segment et d'une adresse de déplacement.**

17 indicateurs

CF Carry Flag (bit 0)	retenue
PF Parity Flag (bit 2)	
AF Auxiliary Carry Flag (bit 4)	
ZF Zero Flag (bit 6)	vaut 1 lorsque le résultat est 0
SF Sign Flag (bit 7)	bit de signe du résultat
OF Overflow Flag (bit 11)	dépassement, le résultat contient trop de bits
DF Direction Flag (bit 10)	sens d'incrémentation de ESI et EDI
TF Task Flag (bit 8)	active la gestion de tâche en mode protégé
IF Interrupt Flag (bit 9)	interruption
IOPL I/O Privilege Level (bits 12 et 13)	
NT Nested Task (bit 14)	
RF Resume Flag (bit 16)	active le mode debug
VM Vitrual 8086 Mode (bit 17)	
AC Alignement Check (bit 18)	
VIF Virtual Interrupt Flag (bit 19)	
VIP Virtual Interrupt Pending (bit 20)	
ID Identification Flag (bit 21)	

3. Modes d'adressage avancés

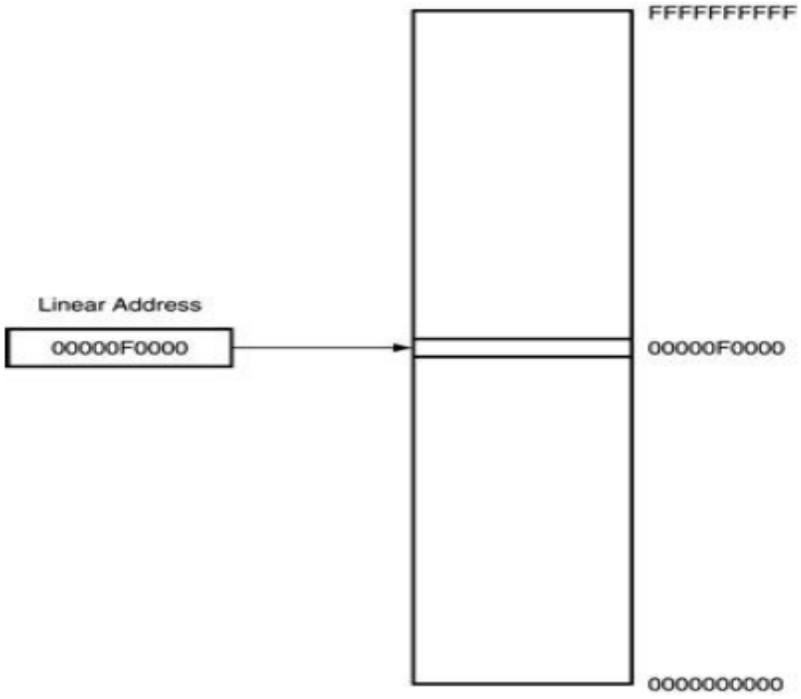
- **3.1. Passage de l'adressage segmenté à l'adressage plat**
- Le microprocesseur 64 bits, manipule des adresses physiques sur 64 bits, d'accéder à 64 GO de mémoire vive
- L'architecture 64 bits: Le modèle de mémoire utilisé = modèle plat (flat mode memory) dans lequel **il n'y a pas de segmentation**
- L'adresse physique du premier octet de la mémoire est : 00 0000 0000H et le dernier est FF FFFF FFFF H, l'adresse est de 40 bits

- Les **registres de segment** n'ont donc **plus d'utilité** a priori pour *adresser un emplacement en mémoire*.
- En fait le registre **de segment de code CS** est utilisé pour spécifier un descripteur précisant les droits d'accès (**mais non son adresse de base et sa limite**).
- **Les données** et **la pile** sont *contenues dans* le segment de code.
- Le **registre CS** est toujours utilisé en mode protégé en mode **64 bits**.

- **La protection** et **la pagination** **sont autorisées en mode 64 bits**.
- ***La pagination mémoire:*** d'attribuer n'importe quel emplacement mémoire physique à n'importe quelle adresse linéaire.
- **Une adresse linéaire :** l'adresse générée par un programme.
- **L'adresse physique:** est l'emplacement mémoire **réel** auquel accède un programme.

- **Grâce à la pagination** mémoire, l'adresse linéaire est traduite de manière invisible en n'importe quelle adresse physique.

- La plupart des programmes fonctionnent en mode compatible IA32.
- Les logiciels actuels fonctionnent correctement mais cela changera dans quelques années avec **l'augmentation de la mémoire** et **l'arrivée massive d'ordinateurs 64 bits.**

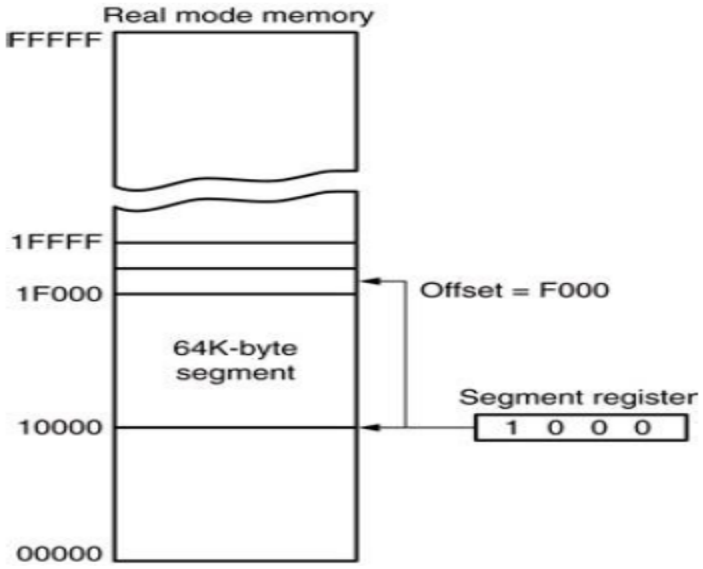


- **3.2. Adressage mémoire en mode réel**
- Les processeurs 80286 et supérieurs fonctionnent **en mode réel** ou **protégé** :
- Le fonctionnement en mode réel: permet **d'adresser** uniquement **le premier Mo d'espace mémoire**, même sur les microprocesseurs Pentium 4 ou Core2 (l'architecture 64 bits).
- Le premier Mo de mémoire est appelé mémoire réelle, mémoire conventionnelle ou mémoire système DOS.

3.3. Segments et Offsets (spécificités 64 bits)

- Toutes les **adresses mémoire en mode réel** doivent être composées **d'une adresse de segment et d'une adresse de déplacement (Offset).**
- **L'adresse de segment:** définit l'adresse de **début de tout segment mémoire** de **64 Ko.**
- **L'adresse de déplacement (Offset):** sélectionne **n'importe quel emplacement** dans le segment mémoire de 64 Ko.

- *Comment le schéma d'adressage segment plus déplacement sélectionne un emplacement mémoire :*
- *Ceci montre un segment de mémoire commençant à 10 000 H et se terminant à l'emplacement IFFFFH. Longueur : 64 Ko*
- *Montre également comment une adresse de Offset, appelée déplacement, de F000H sélectionne l'emplacement 1F000H dans la mémoire.*



- L'emplacement **de départ d'un segment:** est défini par le nombre de **16 bits** du registre de segment, **suivi d'un zéro hexadécimal** à son extrémité droite.

- **L'adresse de déplacement (offset)** : est **un nombre de 16 bits *ajouté* à l'adresse de segment de 20 bits** pour former **l'adresse mémoire en mode réel**.

Adresse effective = 16 × segment + déplacement

- L'adresse de déplacement est toujours ajoutée à l'adresse de début du segment pour localiser les données.
- La forme 1000:2000 ➪ L' adresse de segment de 1000H ; un décalage de 2000H
- **L'accès à toutes les instructions (code)** se fait par **la combinaison de CS (adresse de segment)** et **IP ou EIP (adresse de déplacement)**.

- **Adressage: Basé, Indexé (avec ou sans déplacement), adressage relatif (32, 64 bits)**
- [EDI] et [ESI] : adressage indexé (sur32 bits)
- [EDI]+[offset] et [ESI]+[offset] : adressage indexé avec déplacement « relatif » (sur32 bits)
- [EBP] et [EBX]: adressage basé (sur32 bits)
- [EBP]+[offset] et [EBX]+[offset]: adressage basé avec déplacement « relatif » (sur32 bits)
- **Il y'a aussi l'adressage basé indexé et basé indexé relatif**
- **Pareil avec les registres en mode réel (sur 64 bits) RBP, RBX, RDI, RSI avec ou sans déplacement**

4. Jeu d'instructions

4.1. Unités vectorielles

- **Les unités vectorielles :**de vectoriser le code, de le paralléliser au sein du **microprocesseur.**
- On exécutera la même instruction sur plusieurs données différentes stockées dans un **registre** de type MMX (64 bits), **SSE (128 bits)** ou **AVX (256 bits)**.
- Par exemple avec la technologie **SSE**, au lieu d'écrire :

```
1  float v1[4], v2[4], v3[4];
2
3  void vector_sum(float *x, float *y, float *z, int size) {
4      for (int i = 0; i < 4; ++i) {
5          z[i] = x[i] + y[i]
6      }
7  }
8
9  vector_sum(v1, v2, v3, 4);
```

- On réalise une seule opération en parallèle sur un registre capable de contenir **4 floats**, ce que l'on note :

```
z[0:3] = x[0:3] + y[0:3]
```

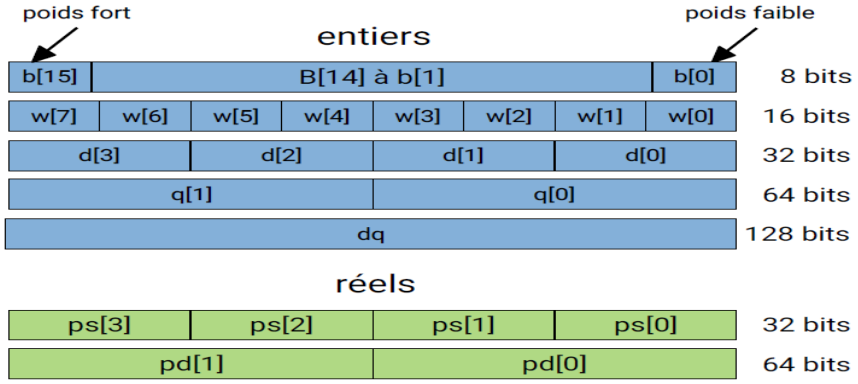
- On parle alors de **traitement** SIMD « Single Instruction Multiple Data », cela signifie que la même instruction est appliquée sur des données différentes **en parallèle:** (technologies SSE et AVX)

4.2. Particularités et extensions modernes : SSE

- Sous le sigle **SSE** (Streaming SIMD Extensions) tous les jeux d'instructions successifs SSE, SSE2, SSE3, SSSE3, SSE4A, SSE4.1 et SSE4.2
- Un jeu de 70 instructions apparu en 1999 sur le Pentium III, traitent des entiers ou des réels.
- Représentation **Intel** pour les instructions SSE qui consiste à : écrire les valeurs d'un vecteur en mémoire ou d'un registre en plaçant la partie haute à gauche et la partie basse à droite.

- L'efficacité du SSE était sur le Pentium 4, bien que le Pentium III, car il dispose des registres **de stockage de 128 bits**, il ne possédait que de registres **64 bits pour réaliser les calculs.**
- **Une instruction SSE** était traitée en deux fois 64 bits: on commençait par traiter la partie basse, puis la partie haute, ce qui est moins efficace que de traiter 128 bits en une seule fois.
- En architecture **32 bits**, il existe 8 registres SSE de 128 bits nommés **xmm0** à **xmm7**. Ce nombre de registres est doublé en architecture **64 bits** avec l'ajout des registres **xmm8** à **xmm15**

- Les registres SSE possèdent des instructions qui traitent les valeurs qu'ils contiennent:
- **Soit sous forme d'entiers au format :**
- 16 octets
- 8 mots
- 4 double mots (4 entiers 32 bits signés ou non)
- 2 quadruplés mots (2 entiers 64 bits signés ou non)
- double quadruple mot (double quad word) soit un total de 128 bits
- **Soit sous forme de nombres à virgule flottante (32 et 64 bits)**
- 4 flottants simple précision (float)
- 2 flottants double précision (double)



- L'instruction **paddb** réalise une addition entière en **parallèle** entre les **16 octets** de **ses deux opérandes**,
- Alors que **paddq, paddw, paddd** réalise une addition entière en **parallèle** sur **4 entiers 32 bits**.
- De la même manière **addps, addpd** réalise une addition en parallèle sur **4 flottants** en simple précision et **addpd** traite **2 flottants** en double précision, respectivement.
- On note également pour **les flottants** les suffixes **ss** et **sd** qui ne traitent que **la partie basse du registre SSE** (respectivement 32 et 64 bits).

Type	Taille en octets	Nom	Quantité dans 128 bits	Suffixe
entier	1	byte	16	b
entier	2	word	8	w
entier	4	double word	4	d
entier	8	quad word	2	q
flottant	4	float	4	ps
flottant	8	double	2	pd
flottant	4	float	1	ss
flottant	8	double	1	sd

- Accéder au i^{ème} élément et manipuler un registre vectoriel sous forme de petits programmes C :
- **xmm0.T[i]** ou : **T** représente le type (b, w, d, q, ps, pd) et **i** le i^{ème} élément. Ainsi **xmm0.b[15]** représente le dernier octet du registre **xmm0** (donc l'octet de poids fort), l'octet de poids faible étant: **xmm0.b[0]**.

➢ **Chargement et stockage des données**

- Pour **les entiers**, on utilisera **movdqu** ou **movdqa**

```
1    movdqa  xmm1, [ebx]            ; opérande SSE et référence
2                                   ; mémoire (Load)
3    movdqa  [edi + ecx * 4], xmm7  ; idem (Store)
4    movdqa  xmm3, xmm1             ; deux opérandes SSE
```

- **Les flottants**, on utilisera les instructions **movups** ou **movaps** , et pour traiter que **la partie basse** du registre SSE comme **movd** pour **les entiers** et **movss, movsd** pour **les flottants** simple et double précision

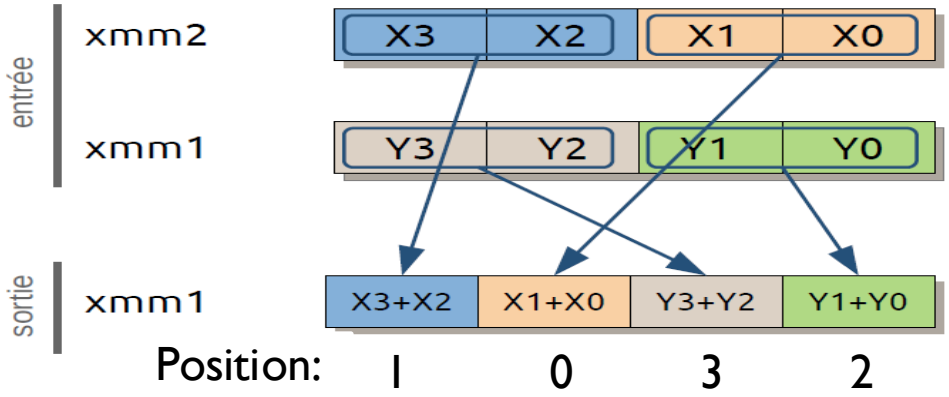
```
1  mov     eax, 0x01010101
2  movd    xmm1, eax        ; xmm1.d[0] = 0x01010101, xmm1.d[1:3] = ?
3  movss   xmm2, [edi]      ; xmm2.ps[0] = [edi], xmm2.ps[1:3] = ?
4  movsd   xmm2, [edi]      ; xmm2.pd[0] = [edi], xmm2.pd[1] = ?
```

Instructions arithmétiques

- Pour **les entiers**, on utilisera les instructions **padd** pour l'addition, **psub** pour la soustraction et **pmull** pour la multiplication, et **il n'existe pas** d'instruction **pdiv** qui réaliserait une division entière!!!.
- **Exemple: padd eax, xmm1**
- Pour **les flottants**, on trouve les instructions **addps, subps, mulps, divps** ainsi que **addpd, etc** …….. Pour la partie basse des registres
- **Exemple: addpd eax, xmm1.b[0]**
- L'addition de l'octet de poids faible de **xmm1** au registre **eax** et la partie fort « à droite » de registre **eax** sera complété par des zéro

- **haddps** l'utilisation de deux fois cette instruction sur le même registre permet de calculer la somme des quatre valeurs du registre **SEE**

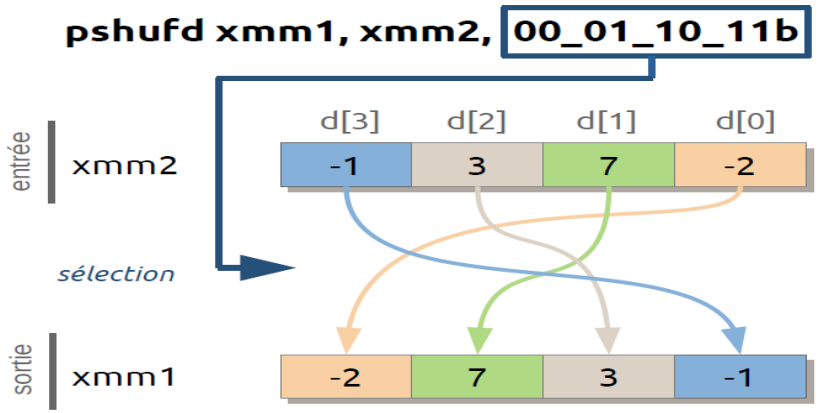
haddps xmm1, xmm2




```
1  ; haddps xmm1, xmm2
2  ; on utilise un registre temporaire xmmt
3  xmmt.ps[0] = xmm1.ps[0] + xmm1.ps[1]
4  xmmt.ps[1] = xmm1.ps[2] + xmm1.ps[3]
5  xmmt.ps[2] = xmm2.ps[0] + xmm2.ps[1]
6  xmmt.ps[3] = xmm2.ps[2] + xmm2.ps[3]
7  xmm1 = xmmt
```

Instructions de réarrangement

- Les instructions **pshufd** pour **les entiers** et **shufps** pour **les flottants** permettent de sélectionner ou réorganiser les données au sein d'un registre **SSE.**
- Ces instructions utilisent un troisième opérande qualifié de masque et notée **imm8** ce qui signifie qu'il s'agit **d'une constante** sur 8 bits et elle est utilisée pour indiquer quels champs sélectionner.
- **Exemple: pshufd xmm1, xmm2, imm8** réalise une sélection et réorganisation des valeurs de **xmm2** vers **xmm1 et imm8** pour indiquer quel champs utilisé sur 8bits



- Une autre instruction intéressante est **blendps**, mais elle n'utilise que les **4 premiers bits** de la constante **imm8**. Elle permet de remplacer les valeurs du registre de destination par des valeurs du registre source :

```
1  // blendps xmm1, xmm2, imm8
2  for (int index = 0; index <= 3; ++index) {
3      xmm1.ps[ index ] = (imm8 & (1 << index)) == 0
4          ? xmm1.ps[ index ] : xmm2.ps[ index ];
5  }
```

- Ainsi, le code suivant remplacera **xmm1.ps[1]** par **xmm2.ps[1]** :

```
1  blendps xmm1, xmm2, 00000010b
```

- Il existe une série d'instructions **vpbroadcast(b,w,d,q)** qui permettent de **recopier une valeur** dans plusieurs emplacements d'un registre SSE ou AVX.
- **Exemple: vpbroadcastb xmm1, xmm2** recopie le **premier octet** du registre **xmm2** dans les **15** autres emplacements du registre:

```
1  // vpbroadcastb xmm1, xmm1
2  for (int index = 1; index <= 15; ++index) {
3    xmm1.b[ index ] = xmm1.b[ 0 ];
4  }
```

- Enfin, l'instruction **insertps xmm1, xmm2, imm8** réalise plusieurs opérations
- **Exemple:** On commence par charger dans **xmm1** le vecteur [4.0, 3.0, 2.0, 1.0], puis dans **xmm2** le vecteur [8.0, 7.0, 6.0, 5.0]. On choisit alors la valeur d'indice 10b de **xmm2**, c'est à dire 7.0 et on la recopie en position 11b de **xmm1**. La partie basse de la constante **imm8**, soit 0011b indique que les valeurs d'indices 0 et 1 de **xmm1** doivent être mises à zéro.

4.3. Particularités et extensions modernes : AVX

➢ Spécificités

- Sous AVX nous plaçons les jeux d'instructions AVX (*Advanced VectoreXtensions*) et AVX2 256 bits.
- Tout comme en **architecture 32 bits**, il existe 8 registres AVX de **256 bits** nommés **ymm0** a **ymm7**.
- Ce nombre de registres est doublé en architecture **64 bits** avec l'ajout de **ymm8** a **ymm15**. Les principaux changements par rapport au SSE sont les suivants :
- **Les instructions AVX** commencent par la lettre **v** pour les distinguer des ***instructions SSE***
- **Les instructions AVX** peuvent agir sur les registres **ymm** ou **xmm** et vont utiliser ***la même syntaxe***
- Cependant, une instruction AVX peut prendre un opérande supplémentaire qui sera **le registre de destination**

- Par exemple, en **SSE**, si on écrit: **paddd xmm1, xmm2**, les quatre entiers de **xmm2** sont ajoutés à **xmm1**, en d'autres termes on a **xmm1.d[0:3] += xmm2.d[0:3]**.
- Les valeurs présentes dans **xmm1** sont donc perdues. On aura le même comportement si on utilise **vpaddd xmm1, xmm2 , en AVX**

```
1  ; avec deux opérandes
2   paddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
3  vpaddd  xmm1, xmm2  ; xmm1.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

- Par contre, si on écrit **vpaddd xmm3, xmm1, xmm2**, le registre **xmm3** recevra le résultat de la somme de **xmm1** et **xmm2**. Les registres **xmm1** e **xmm2** ne seront donc pas modifies.

```
1  ; avec trois opérandes
2  vpaddd  xmm3, xmm1, xmm2  ; xmm3.d[0:3] = xmm1.d[0:3] + xmm2.d[0:3]
```

Partie haute

- Certaines instructions, comme **insertps**, travaillent uniquement avec la partie basse des registres **AVX**. Cela est du à la constante **imm8** qui interagit avec l'un des quatre flottants simple précision d'un registre SSE.
- **L'extension AVX** de cette instruction **vinsertps** ne permet pas d'identifier les flottants dans la partie haute d'un registre AVX.
- Il est donc nécessaire pour transposer l'utilisation du **SSE vers l'AVX** de travailler sur la partie basse du **registre AVX** puis de déplacer la partie basse vers la partie haute.
- On dispose par exemple des instructions **vinsertf128**, **vextractf** ou **vpbroadcast** qui réalisent ces manipulations.
- La série d'instructions vpbroadcast**(b/w/d/q) x/ymm, reg** recopie les **8/16/32 ou 64 bits** d'un registre général respectivement vers tous les octets, mot, double mots ou quadruples mots d'un registre SSE ou AVX.
- Ainsi pour recopier **8 fois l'octet 0x85** dans le registre **ymm1**, on écrira :

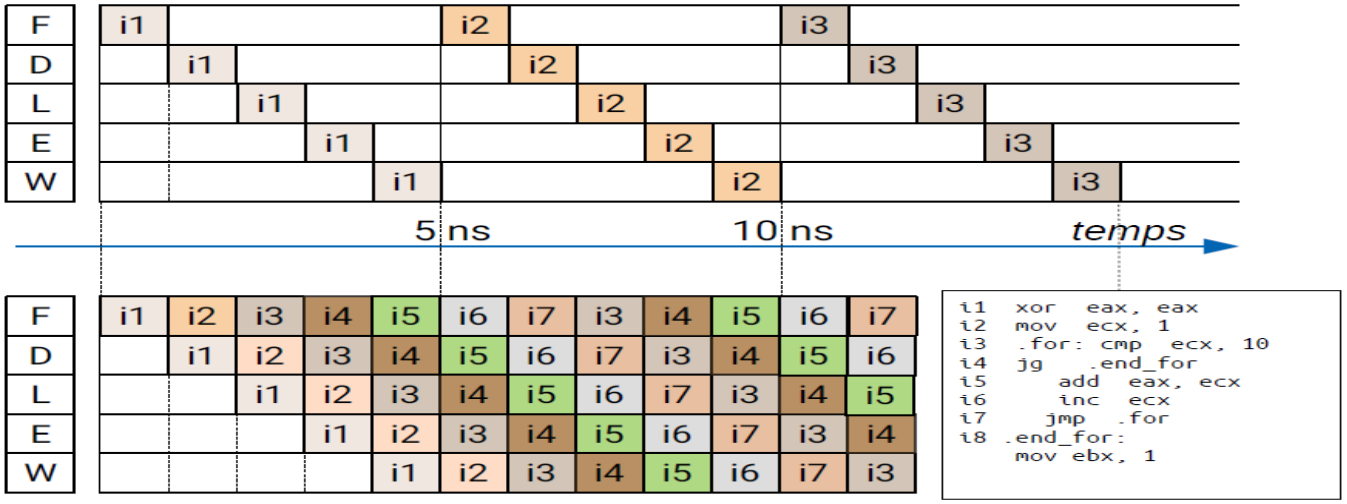
```
1  mov          eax, 0x85    ; ou mov al, 0x85
2  vpbroadcastb ymm1, eax
```

Instructions singulières

- Certaines instructions n'ont pas le même comportement en **AVX** et en **SSE**.
- C'est le cas de **haddps** l'utilisation de deux fois cette instruction sur le même registre permet de calculer la somme des quatre valeurs de registre SEE
- Cela ne fonctionne pas avec les 8 valeurs **32 bits** qui contient un registre **ymm** lorsque l'on utilise **vhaddps**.

5. Cycle d'exécution d'une instruction

- **5.1.Étapes du pipeline pour une instruction donnée**
- Afin **d'améliorer la vitesse de traitement** des instructions un mécanisme de **pipeline** a été mis en place.
- Pipeline : **ne pas attendre que l'ensemble des étapes de traitement aient été réalisées avant de passer à l'instruction suivante**. chaque étape de traitement indépendante.
- **Principe:** Une première instruction passe dans l'étape de chargement au temps **t = 0**, puis au temps **t + 1**, elle passe dans l'étape de décodage, pendant que l'instruction suivante passe dans l'étape de chargement et ainsi de suite.



- Quel **gain** apporte le **pipeline** ? Pour répondre a cette question il suffit de **comparer** les temps d'exécution avec et sans pipeline pour traiter n instructions :
- *Sans pipeline* une instruction est exécutée toutes les **5 ns**, si on à **n** instructions à exécuter il faut donc **5 × n ns**.
- *Avec pipeline*, il faut **5 ns** pour que la première instruction soit exécutée, puis **n − 1** ns pour exécuter les **n − 1** instructions restantes
- Le gain obtenu est donné par le rapport du **temps d'exécution** sans pipeline par le temps d'exécution avec pipeline :

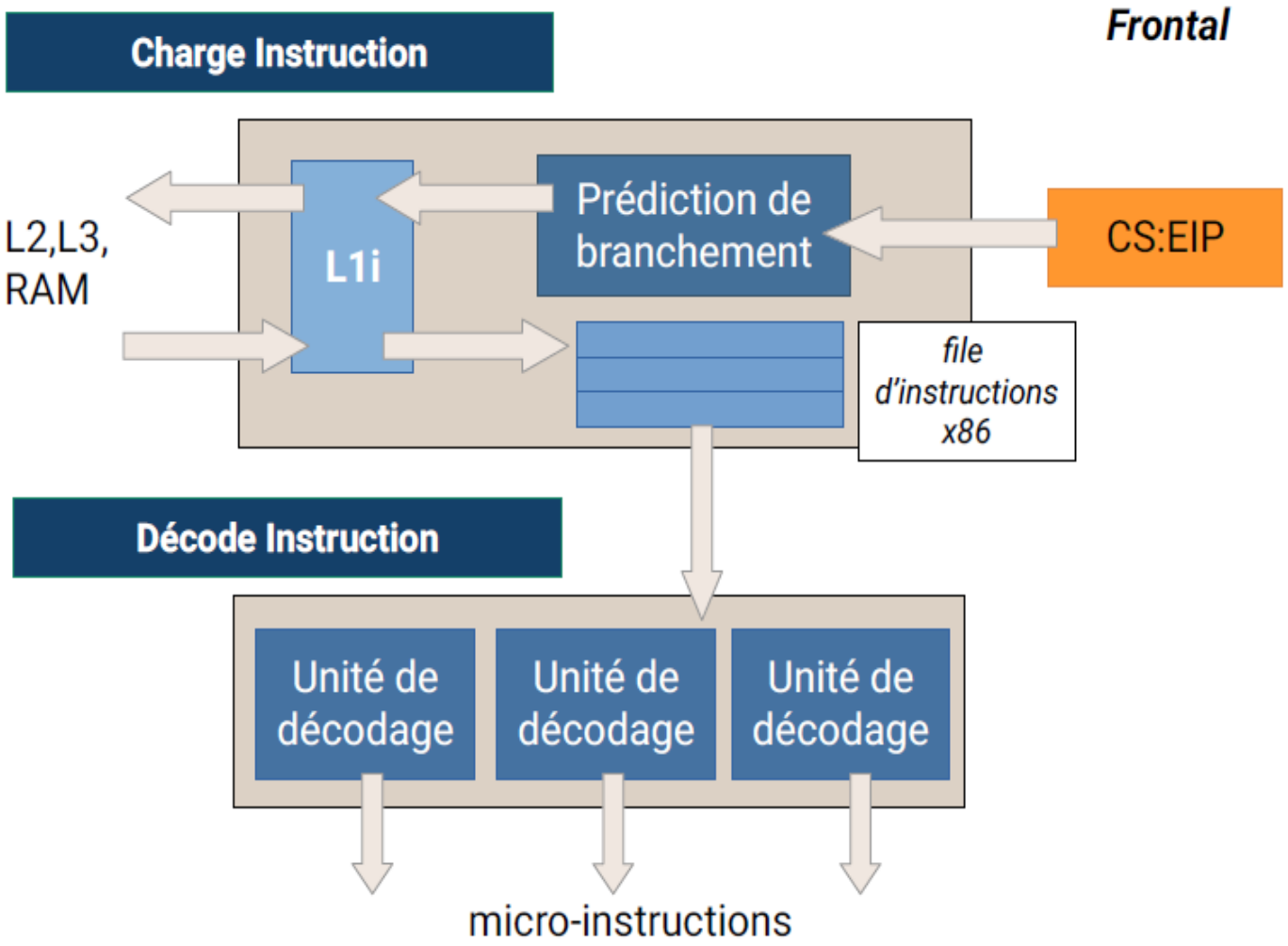
$$\text{gain} = \lim_{n\to\infty}\left(\frac{5n}{5+n-1}\right) \simeq \frac{5n}{n} \simeq 5$$

Micro architecture	Pipeline	Micro architecture	Pipeline
P5 (Pentium)	5	NetBurst (Cedar Mill)	31
P6 (Pentium 3)	10	Core	14
P6 (Pentium Pro)	14	Sandy Bridge	14
NetBurst (Willamette)	20	Haswell	14
NetBurst (Northwood)	20	Skylake	14
NetBurst (Prescott)	31	Kabylake	14

Nombre d'étages de pipeline pour différentes architectures Intel

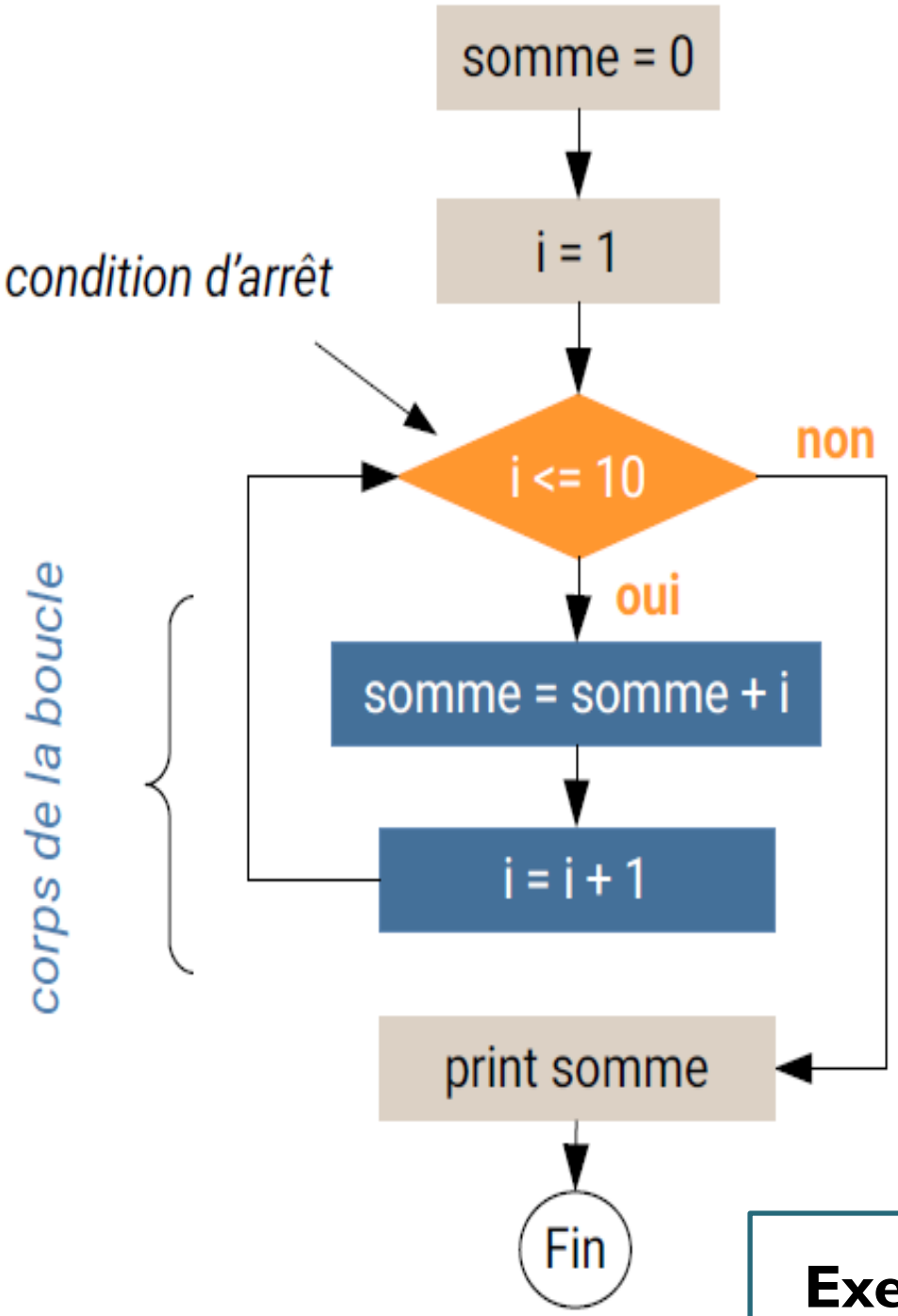
- Plus le **pipeline est long**, plus il est **couteux** de le vider et le réalimenter, c'est ce qui arrive lors de l'exécution des instructions conditionnelles ou lors du traitement des boucles.
- Il se limite à une **quinzaine d'étages** sur la plupart des **microprocesseurs actuels**.

- **1. Frontal : chargement et décodage**
- A partir de **cs:eip**  l'adresse de la prochaine instruction à exécuter.
- Utiliser un mécanisme de prédiction de branchement, (**BPU)** : lire l'instruction suivante ou si on devra se déplacer à une autre adresse du code
- La bonne adresse: on récupère l'instruction à exécuter dans **le cache** L1 d'instructions (L1i).
- Si elle n'est présente dans aucun cache, il faudra lancer une **requête d'accès en mémoire** pour récupérer les octets situés à l'adresse à lire et les charger dans les différents caches ou dans le cache L1i uniquement



Chargement et décodage

- **1.1. Chargement et prédiction de branchement**
- Le chargement d'une instruction : Appel des mécanismes de *prediction de branchement :* prédire à quelle adresse le (**eip**) se placer « l'instruction suivante ».
- Cas de **branchements**, d'une boucle **for** par exemple: revenir au début de la boucle après l'exécution **ou** sortir de la boucle lorsque la condition d'arrêt est atteinte (existe plusieurs chemins d'exécution)



Code C

```
int i, somme = 0;
for (i=1; i<=10; ++i) {
   somme = somme + i;
}
printf("%d",somme);
```

Code Assembleur

```
    xor   eax, eax      ;i1
    mov   ecx, 1        ;i2
.for:
    cmp   ecx, 10       ;i3
    jg    .end_for      ;i4
    add   eax, ecx      ;i5
    inc   ecx           ;i6
    jmp   .for          ;i7
.end_for:
    push    eax         ;i8
    push    dword msg   ;i9
    call    printf      ;i10
```

Exécutinon

Charger dans le pipeline

Exemple de boucle *for*

Considérons un code C pour lequel on calcule la somme des entiers de 1 a 10. On voit qu'il existe deux chemins :
Le premier est pris lorsque i ≤ 10 et le second lorsque i > 10.
Le registre **eax** contient **la somme des valeurs** et le registre **ecx représente la variable de boucle** (**i**).
Après l'utilisation de l'instruction **cmp ecx, 10 :** compare le registre **ecx** à la constante 10, on place une instruction de branchement conditionnel jg .end_for, qui signifie ***jump on greater***.

Si **ecx** est supérieur à**10** il faut sortir de la boucle et modifier **eip** pour qu'il pointe sur l'instruction après le label **.end_for**, c'est à dire l'instruction **i8** .Cependant les instructions suivant la comparaison (***i5, i6, i7***) ont déjà été chargées dans **le pipeline** pendant le traitement de ***i3*** et ***i4***.

- Ces **instructions de branchement conditionnel** sont source de ralentissement au sein du pipeline puisqu'il est nécessaire de vider le pipeline si le chemin d'exécution suivi n'est pas le bon.

1.2. Décodage d'instructions

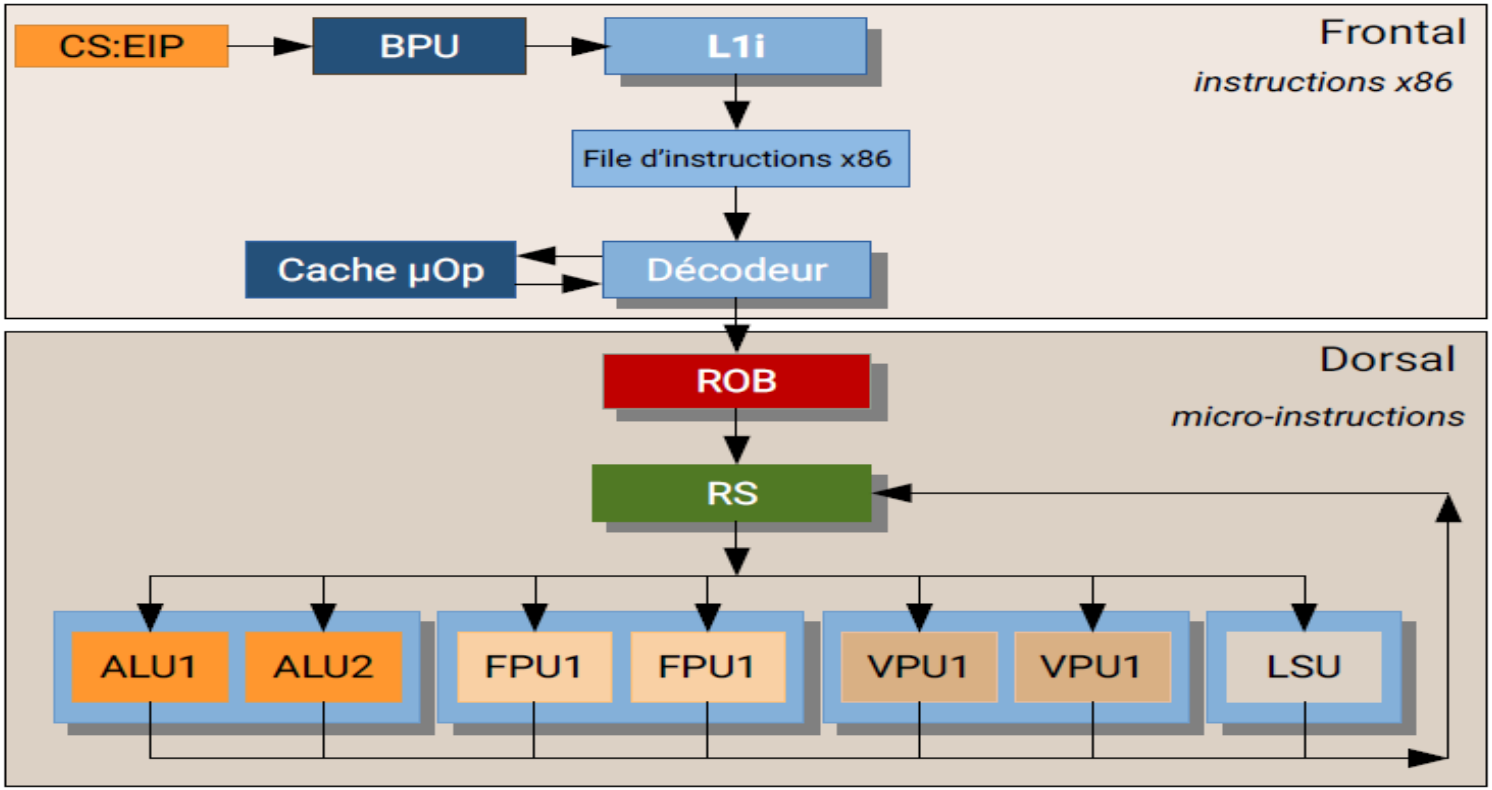
- Les instructions assembleur ⇨ macro-instructions pour traitements parfois très complexes.
- Macro-instructions : décomposées en une série d'instructions plus simples appelées **micro-opérations (µ-ops).**
- L'instruction **add [ebx + ecx * 4 + 8], eax**

Cette instruction décomposée en plusieurs micro-opérations beaucoup plus simples afin d'être exécutée :

- 1) µ-op1 : calcul de l'adresse **@ = ebx + ecx * 4 + 8**
- 2) µ-op2 : chargement de la donnée à l'adresse mémoire @ dans le registre **R**
- 3) µ-op3 : exécution de l'addition **R + eax** et stockage dans **R**
- 4) µ-op4 : stockage de **R** à l'adresse mémoire @

➢ Mécanisme **de cache de traduction représente** le **µ-Ops Cache**, stocker la série de micro-instructions générées par **le décodage d'une instruction x86**

2. Exécution des instructions



- **Dorsal:** C'est un ensemble de **µ-ops associées à des instructions x86** que l'on doit traiter. diminuer les temps d'attentes et accélérer l'exécution du traitement on utilise un mécanisme d'exécution dans **le désordre (*Out Of Order*)** qui consiste à traiter les µ-ops.

- **Exécution dans le désordre pose un problème crucial à résoudre:** les instructions x86 soient traitées dans l'ordre dans lequel elles sont entrées dans le pipeline de traitement.
- Pour ce faire, on utilise deux (*buffers*) appelés **RS** « *Reservation Station* » et **ROB** « *ReOrder Buffer* »
- **ROB:** garder la cohérence et maintenir l'ordre d'exécution, et l'*allocation des registres*
- **RS:** stocker les instructions et de les garder jusqu'a ce qu'elles soient exécutées
- L'*allocation avec renommage des registres* est une technique essentielle pour **traiter les instructions dans le désordre** (établit une correspondance entre les registres visibles par le programmeur (**eax**, **ebx**, etc…, et ses registres internes pour traiter chaque instruction de manière indépendante)

- **5.2. Exemple d'instructions en 32 bits et 64 bits**
- La majorité des instructions x86 sont de la forme :
- **opération** *destination*, *source*
- **opération** est un mnémonique : un symbole court de quelques lettres facilement compréhensible et mémorisable qui représente l'opération à exécuter, par exemple **add** pour l'addition
- *source* est une donnée **en lecture** qui ne sera donc pas modifiée, ce peut être une constante, un registre ou une adresse mémoire
- *destination* est une donnée **en écriture** qui peut être un registre ou une adresse mémoire

- **Remarque:** utilisant l'architecture **64 bits** avec les registres de 64 bits **rax, rbx, …, puisque l'utilisation des registres de 32 bits avec l'architecture 64 bits signifie le reste des bits sera remplis est complété par des zéros**

- **add eax**, **ebx** signifie que l'on doit réaliser le calcul **eax** = **eax** + **ebx** : opération d'addition en 32 bits,
- **add rax**, **rbx** opération d'addition en 64 bits
- Nous verrons qu'il existe d'autres variantes de format d'instruction comme pour les instructions **not**, **cmp**, **div**, **mul**, etc.
- Contrainte de codage des instructions x86 à n'avoir qu'une seule référence mémoire
  cela implique que l'on ne peut pas écrire :

```
1    operation    [adresse1], [adresse2]     ; !!! non autorisé !!!
```

- Il faudra alors passer par un registre et écrire:

```
1    mov          registre, [adresse2]
2    operation    [adresse1], registre
```

L'instruction de chargement et stockage des données : Mov et ses variantes

- L'instruction **mov** déplace la donnée située à l'adresse2 dans un registre, on place l'adresse entre crochets [ ]. Ainsi :
- **mov eax**, [**addr**] signifie placer la valeur 32 bits située à l'adresse **addr** dans le registre **eax**
- **mov eax**, **val** signifie placer la valeur constante codée sur 32 bits **val** dans le registre **eax**
- **movsx** (*Mov with Sign eXtension*) qui transforme une valeur sur **8 bits** en une valeur **16 bits ou** une valeur sur **16 bits** en une valeur **32 bits** en préservant le fait que la valeur soit négative ou positive
- **movzx** (*Mov with Zero eXtend*) qui transforme une valeur sur **8 ou 16 bits** en une valeur **16, 32 ou 64 bits** en la complétant avec des **0**. L'instruction **movzx** est parfois plus rapide que **mov.**

- Il est donc préférable d'écrire, afin de lire **l'octet** à l'adresse **edi** en mémoire et le stocker dans le registre **al** :

```
1    movzx eax, byte [edi]
```

- Plutôt que :

```
1    mov   al, [edi]
```

- La différence: **movzx** va modifier **eax** en mettant dans **al** l'octet pointé par **edi** et en mettant à 0 les 24 autres bits.
- Dans ce cas il faut préciser la quantité chargée : **byte** pour un octet, **word** pour un mot et dans d'autres instructions **dword** pour un double mot (32bits)
- **Exemples :**
- **mov eax, 0** : affecter la valeur 0 au registre **eax**
- **mov eax, ebx** : affecter le contenu de **ebx** au registre **eax**
- **mov al, bh** : affecter le contenu de **bh** au registre **al**
- **mov eax, [ebx + ecx * 4]** : affecter au registre **eax** la valeur située à l'adresse mémoire indiquée, il s'agit d'une référence mémoire comme **eax** est un registre **32 bits** on lit le double mot situé à l'adresse indiquée
- **mov [edi + esi], edx** : stocker à l'adresse **edi + esi** la valeur contenue dans le registre **edx**

Instructions arithmétiques

- **Instructions add, sub, inc et dec**
- Les instructions **add** et **sub :** l'addition et la soustraction de deux valeurs entières signées ou non signées et prennent deux opérandes.
- Les instructions **inc** et **dec** : respectivement l'incrémentation et la décrémentation de leur unique opérande.
- Les deux instructions suivantes sont donc équivalentes :

```
1    add   eax, 1
2    inc   eax
```

- On trouve également l'instruction **sbb** (*SuBstract with Borrow*) pour faire des soustractions si on utilise deux registres **16** ou **32 bits**.
- Concernant les instructions **inc** et **dec**, elles **modifient les flags** OF, SF, ZF, AF et PF, mais pas le Carry Flag. Il est de plus conseillé de ne pas les utiliser car elle peuvent produire dans certaines situations des *false dependencies*.

L'instruction mul

- L'instruction **mul** n'accepte qu'un seul opérande source et réalise la multiplication non signée entre un registre 8, 16 ou 32 bits et respectivement **al**, **ax**, **eax**

Opération	Source	Résultat
mul reg8	al	ax
mul reg16	ax	dx:ax
mul reg32	eax	edx:eax

- **mul bh**, c'est le registre **al** qui est multiplie par **bh** et le résultat est place dans **ax**.
- En architecture **32 bits**, on notera qu'avec un opérande source de **16 bits**, le registre **dx** est modifié et si opérande source de **32 bits**, le registre **edx** est modifié . Il ne faudra pas stocker de donnée dans **edx**, avant la multiplication ou en la plaçant dans la pile, puis après la multiplication, la récupérer depuis la pile. Pour calculer 7 × 5 ou 5 × 7, on écrira donc :

```
1    push  edx       ; on sauvegarde edx
2    mov   eax, 5
3    mov   ebx, 7
4    mul   ebx       ; edx:eax= 0:35
5    pop   edx       ; on restaure edx
```

L'instruction div

- L'instruction **div** réalise la division entière **non signée** entre une valeur **64, 32 ou 16 bits** par un diviseur sur **32, 16 ou 8 bits** respectivement, le reste de la division est également calculé.

Dividende	Diviseur	Quotient	Reste
edx:eax	div reg32	eax	edx
eax	div reg16	ax	dx
ax	div reg8	al	ah

- En architecture 32 bits: **une valeur sur 64 bits contenue dans deux registres 32 bits edx:eax** que l'on divise par un opérande 32 bits contenu dans un autre registre.
- Si on désire travailler avec des valeurs 32 bits, il faut mettre **edx** à 0 avant de faire la division. Pour diviser 1024 par 3, on écrira donc :

```
1    mov   eax, 1024
2    xor   edx, edx    ; mise à zéro de edx pour rester
3                        ; en 32 bits
4    mov   ebx, 3
5    div   ebx
```

Comparaison : L'instruction cmp

- L'instruction **cmp** (*CoMPare*): la comparaison **non signée** de deux opérandes en calculant leur différence. Les opérandes ne seront pas modifiées seule la différence sera utilisée pour mettre à jour le registre des **flags**.

cmp eax, ebx	CF	SF	ZF
eax == ebx	0	0	1
eax < ebx	1	1	0
eax > ebx	0	0	0

Voici quelques utilisations de cette instruction:

- **cmp eax, 10** : compare **eax** à la constante 10
- **cmp eax, edx** : compare **eax** à **edx**
- **cmp ecx, [ebp-12]** : compare **ecx** à l'entier 32 bits situe en mémoire à l'adresse [ss:ebp-12]

Instructions de branchement conditionnel

- **Loop (conditionnel)**
- Il existe également une instruction spécifique **loop *address*** : décrémente le registre **ecx** , se branche à l'adresse indiquée.
- Elle est donc équivalente aux deux instructions suivantes :

```
1  .begin:
2      ...
3      dec   ecx     ; à remplacer par
4      jnz   .begin  ; loop .begin
```

Autres instructions de branchement (inconditionnel)

- **jmp address** : modifie (*JuMP*) le pointer d'instruction pour qu'il soit égal à l'opérande ***address***
- **call address** : réalise un appel de sous-programme
- **ret** : réalise le retour de sous-programme
- L'instruction **call** empile l'adresse de l'instruction qui lui succède puis modifie le registre **eip** pour qu'il soit égal à **address**. L'instruction **ret**, utilisée lors du retour de sous-programme, dépile l'adresse en sommet de pile (placée par **call**) et l'affecte à **eip**.

Résumé

En architecture **32 bits**, on dispose de huit registres généraux, cependant seuls 6 sont utilises pour faire des calculs ou stocker des données (**eax**, **ebx**, **ecx**, **edx**, **edi**, **esi**) ; le registre **esp** contient le sommet de pile et ne doit pas être modifie directement alors que **ebp** est utilisé afin de récupérer les arguments d'un sous-programme :

- En architecture **32 bits**, si l'on doit réaliser des multiplications ou des divisions les registres **eax** et **edx** seront impactes, ce qui ne laisse plus que 4 registres pour faire les calculs

- En architecture **64 bits**, on dispose de 8 registres généraux supplémentaires (**r8** a **15**), ce qui permet de lever le verrou des limitations du **32 bits**

- L'assembleur ne dispose pas de structures de contrôle comme la conditionnelle **if**, les boucles **for**, **while**. Ecrire en assembleur est donc une tache difficile.

- Les techniques de dépliage de boucle ou de tuilage permettent d'améliorer l'efficacité du traitement des boucles

- Positionner un **if** a l'intérieur d'une boucle (**for** ou **while**) conduit à ralentir l'exécution du traitement, il faut alors être en mesure de pouvoir éliminer le **if** soit en le remplaçant par des instructions spécifiques, soit en le vectorisant

✓**Compétences à acquérir**

On doit être capable de traduire :

- **Une instruction arithmétique : addition soustraction, multiplication, une division**
- **Une instruction de branchement conditionnelle ou inconditionnelle**