L'assembleur x86 32 bits

Architecture des ordinateurs

Guillaume Blin

IGM-LabInfo UMR 8049,
Bureau 4B066
Université de Marne La Vallée
gblin@univ-mlv.fr
http://igm.univ-mlv.fr/~gblin

Références

- L'excellent "PC Assembly Language" de Paul A. Carter http ://www.drpaulcarter.com/

Plan

Plan

Donnée et mémoire : rappel

- Unité mémoire de base = octet
- Chaque octet en mémoire est étiqueté par un nombre unique (i.e. son adresse)

Adresse	0	1	2	3	4	5	6	7
Mémoire	2A	45	B8	20	8F	CD	12	2E

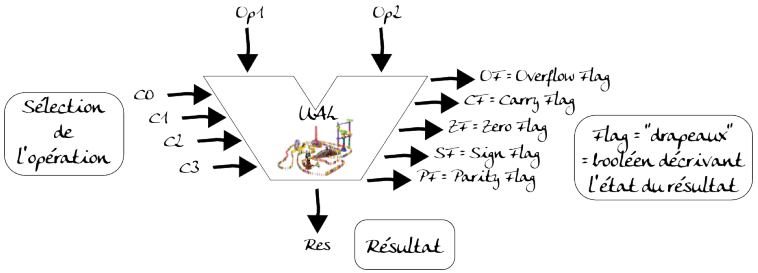
- Souvent, la mémoire est utilisée par bouts plus grand que des octets isolées.

mot	2 octets
double mot	4 octets
quadruple mot	8 octets
paragraphe	16 octets

- Toute donnée en mémoire est numérique
  - Caractères = code caractère
  - Nombre = en base 2

UAL : rappel

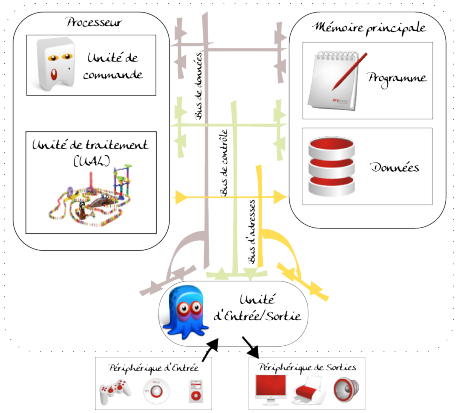
- Unité chargée
  - *des opérations arithmétiques*
  - ADD (+), SUB (-), MUL (*), DIV (:), INC (+ 1), DEC (- 1)
  - *des opérations logiques*
  - AND, OR, XOR, NOT, CMP
  - LSL, LSR, ASR (décalages)



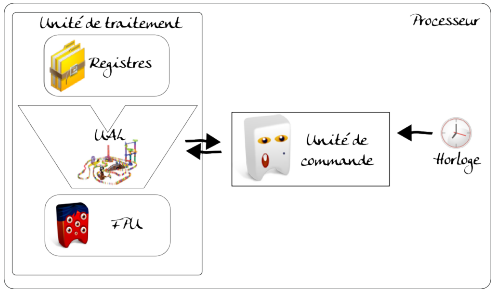
Vision abstraite d'un ordinateur

Bus : nappe de fils conduisant de l'information entre les éléments

- *Bus de données* : transporte les données échangées
- *Bus d'adresses* : transporte des adresses en mémoire
  - Position d'un élément requis par le CPU
  - Position où écrire un élément envoyé par le CPU
- *Bus de contrôle* : transporte les informations de contrôle entre le CPU et les autres organes



Vision abstraite du processeur



- *Registres* : emplacements de stockage d'accès rapide
- *UAL* : unité de calcul (entiers et booléens)
- *FPU* : unité de calcul sur les "réels"
- *Unité de contrôle* : interprète les instructions
- *Horloge* : cadence le processeur (fréquence en MHz/GHz)

Le Central Processing Unit

- Dispositif physique exécutant les instructions (assez simples)
- Pouvant nécessiter la présence de données dans des emplacements de stockage spécifiques (i.e. registres)
- Accès plus rapide aux registres qu'en mémoire
- Mais, le nombre de registres est limité
  $\Rightarrow$ ne conserver que les données actuellement utilisées
- $\{$instructions d'un processeur$\}$ = son langage machine

Le langage machine

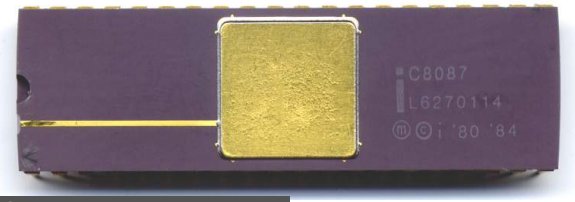
- Les programmes machine
  - structure basique
  - instructions encodées en hexa (pas facilement lisible)
  - doivent permettre au CPU de décoder rapidement les instructions
- Les programmes écrits dans d'autres langages $\rightarrow$ en langage machine natif du processeur pour s'exécuter sur l'ordinateur
- $\Rightarrow$ c'est le rôle du compilateur
- Chaque type de processeur a son propre langage machine
- C'est une des raisons pour lesquelles un programme écrit pour Mac ne peut pas être exécuté sur un PC.

1978 - Le 8086

- Architecture avec tous les registres internes sur 16 bits
- Bus d'adressage de 20 bits (pouvant adresser en mode dit "réel" jusqu'à 1Mo)
  - Dans ce mode, un programme peut accéder à n'importe quelle adresse mémoire, même la mémoire des autres programmes !
  - Pb de débogage et de sécurité
  - La mémoire du programme doit être divisée en segments $\leq$ 64Ko.
- Extension du 8080 (8 bits avec simple accumulateur)
- Ajout de registres supplémentaires avec utilisation déterminiée
- $\Rightarrow$ *Machine à accumulateur étendu*

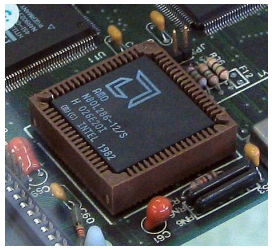
1980 - Le 8087

- Co-porcesseur flottant Intel
- Extension du 8086 avec près de 60 opérations flottantes toutes commençant par "F" (e.g. FADD/FMUL)
- Basé sur organisation hybride avec pile et registres
- Pas un ensemble linéaire de registres tq AX/BX/CX/DX
- Mais structurés sous une certaine forme de pile s'étendant de ST0 à ST7.
- Conduit à l'introduction de la première norme de virgule flottante pour les PC à base de x86 : l'IEEE 754
- $\Rightarrow$ *Machine à pile étendue*



1982 - Le 80286

- Bus d'adressage de 24 bits (pouvant adresser jusqu'à 16Mo)
- Ajout d'instructions
- 2 fois plus rapide que le 8086 par cycle d'horloge.
- 2 modes de fonctionnement possibles :
  - Mode d'adressage dit réel, dans lequel il se comporte comme un 8086 amélioré
  - Un nouveau mode dit protégé 16 bits, permettant l'accés à 16Mo de mémoire et empêcher les prog d'accéder à la mémoire des uns et des autres. Mais toujours division en segments $\leq$ 64Ko.



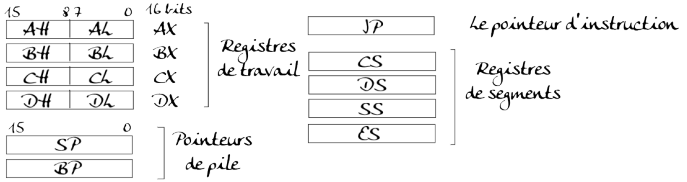
1985 - Le 80386

- Architecture à 32 bits avec registres 32 bits
- Bus d'adressage de 32 bits (pouvant adresser jusqu'à 4Go)
  - Les programmes sont toujours divisés en segments mais $\leq$ 4Go !
- Nouveaux modes d'adressage incluant la prise en charge de la pagination
- Ajout d'instructions

1989 - Le 80486

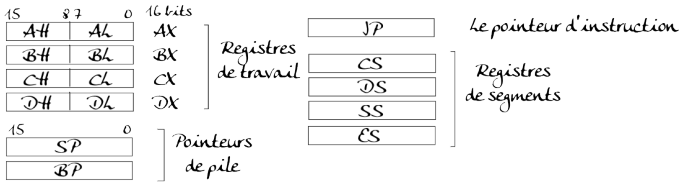
- Du point de vue de l'architecture, c'est une grande amélioration. Il y a un cache d'instruction et de donnée unifiée intégré.
- Unité de calcul en virgule flottante intégrée (FPU),
- Une unité d'interface de bus améliorée

Registres 16 bits du 8086



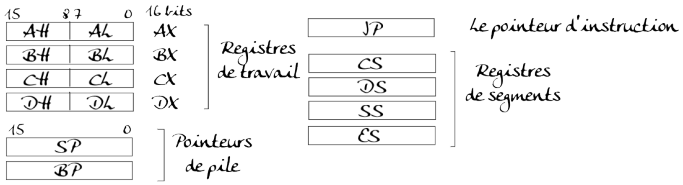
- 4 registres généraux : AX, BX, CX et DX.
  décomposables en 2 registres de 8 bits pouvant être utilisés comme des registres d'un octet indépendants

Registres 16 bits du 8086



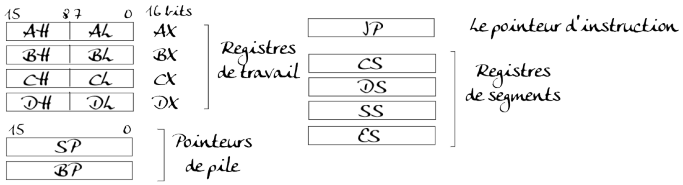
- 4 registres généraux : AX, BX, CX et DX.
  décomposables en 2 registres de 8 bits pouvant être utilisés comme des registres d'un octet indépendants
  Changer la valeur de AX changera les valeurs de AL et AH et vice versa

Registres 16 bits du 8086



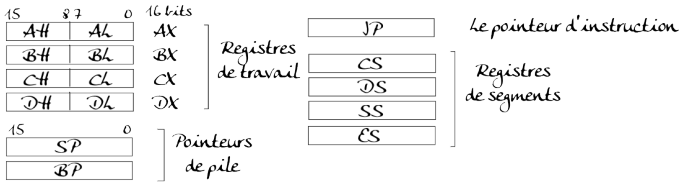
- 4 registres généraux : AX, BX, CX et DX.
  décomposables en 2 registres de 8 bits pouvant être utilisés comme des registres d'un octet indépendants
  Changer la valeur de AX changera les valeurs de AL et AH et vice versa
  utilisés dans beaucoup de déplacements de données et instructions arithmétiques.

Registres 16 bits du 8086



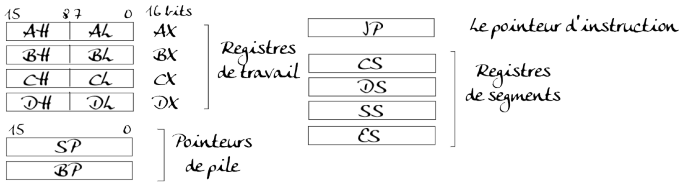
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
  utilisés comme des pointeurs, mais également comme les registres généraux

Registres 16 bits du 8086



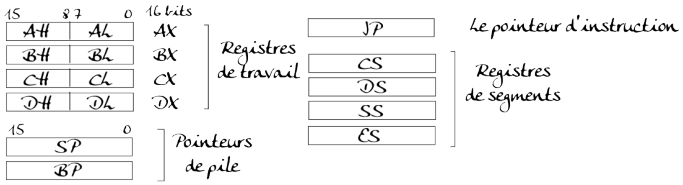
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
  utilisés comme des pointeurs, mais également comme les registres généraux
  mais non décomposables en registres de 8 bits

Registres 16 bits du 8086



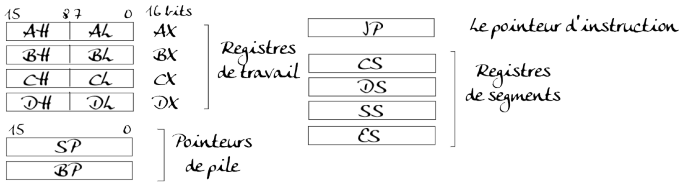
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
  utilisés pour pointer sur des données dans la pile du langage machine et appelés, resp. pointeur de base et de pile

Registres 16 bits du 8086



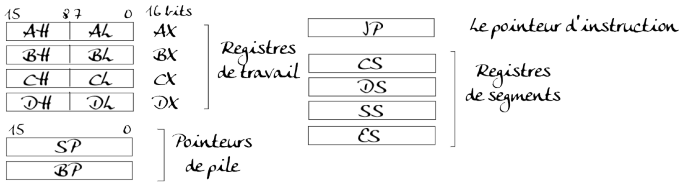
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
  indiquent la zone de la mémoire utilisée par les différentes parties d'un programme

Registres 16 bits du 8086



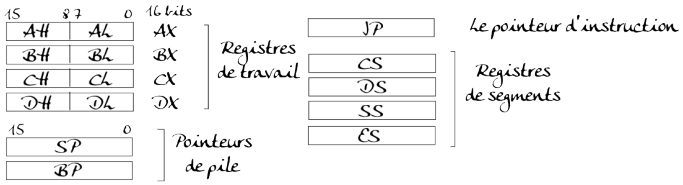
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
  indiquent la zone de la mémoire utilisée par les différentes parties d'un programme
  CS = Code Segment, DS = Data Segment, SS = Stack Segment et ES = Extra Segment

Registres 16 bits du 8086



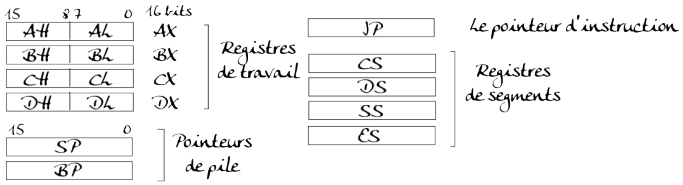
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
  indiquent la zone de la mémoire utilisée par les différentes parties d'un programme
  CS = Code Segment, DS = Data Segment, SS = Stack Segment et ES = Extra Segment
  ES est utilisé en tant que registre de segment temporaire

Registres 16 bits du 8086



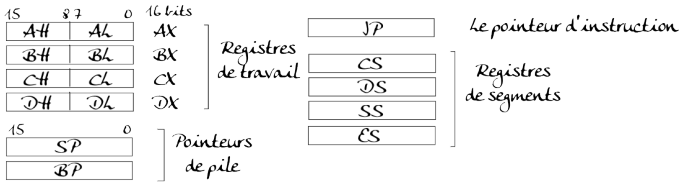
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
- 1 registre de pointeur d'instruction : IP
  utilisé avec le registre CS pour mémoriser l'adresse de la prochaine instruction à exécuter

Registres 16 bits du 8086



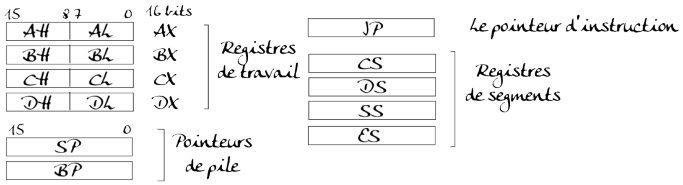
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
- 1 registre de pointeur d'instruction : IP
  utilisé avec le registre CS pour mémoriser l'adresse de la prochaine instruction à exécuter
  normalement, incrémenté lorsqu'une instruction est exécutée

Registres 16 bits du 8086



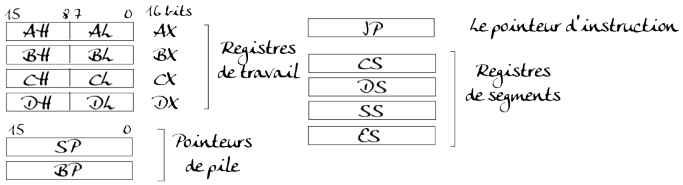
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
- 1 registre de pointeur d'instruction : IP
- 1 registre FLAGS stockant des informations importantes sur les résultats d'une instruction précédente comme des bits individuels dans le registre

Registres 16 bits du 8086



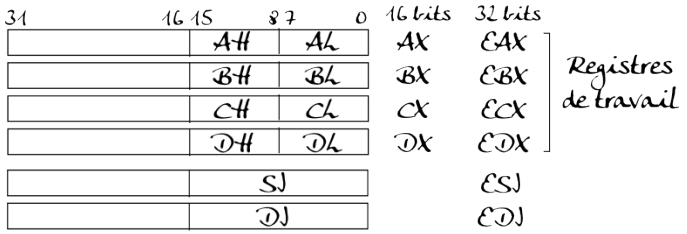
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
- 1 registre de pointeur d'instruction : IP
- 1 registre FLAGS
  par exemple, le bit Z est positionné à 1 si le résultat de l'instruction précédente était 0 ou à 0 sinon

Registres 16 bits du 8086



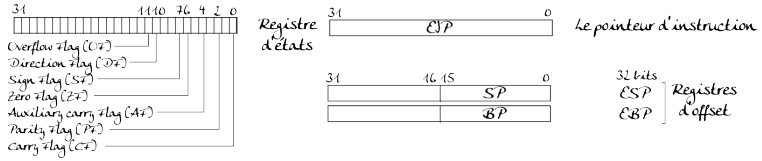
- 4 registres généraux : AX, BX, CX et DX.
- 2 registres d'index : SI et DI.
- 2 registres de pile : BP et SP
- 4 registres de segment : CS, DS, SS et ES
- 1 registre de pointeur d'instruction : IP
- 1 registre FLAGS
  par exemple, le bit Z est positionné à 1 si le résultat de l'instruction précédente était 0 ou à 0 sinon
  toutes les instructions ne modifient pas les bits dans FLAGS

Registres 32 bits du 80386



- Les processeurs 80386 et plus récents ont des registres étendus.
- Par exemple, le registre AX 16 bits est étendu à 32 bits.
- Pour la compatibilité ascendante,
  - AX, AL et AH existent toujours
  - font référence à des parties de EAX
  - pas d'accès aux 16 bits de poids fort de EAX

Registres 32 bits du 80386



- La plupart des autres registres sont également étendus.
- BP devient EBP ; SP devient ESP ; FLAGS devient EFLAGS et IP devient EIP.
- Mais plus d'accès aux registres 16 bits en mode protégé 32 bits

Registres 32 bits du 80386

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

Evolution du "mot"

- Initialement le terme "mot" = la taille des registres de données du processeur
- En 80x86, le terme est désormais un peu confus
- Historiquement, il reste donc défini comme faisant 2 octets (ou 16 bits)

Mode Réel

- En mode réel (8086), la mémoire est limitée à seulement 1Mo (2^{20})
- Les adresses valides vont de 00000 à FFFFF (20 bits en hexa)
- 20 bits > capacité des registres 16 bits du 8086.
- $\Rightarrow$ L'adresse = un registre segment (selecteur, 16 bits de poids fort) + un de déplacement (offset, 16 bits de poids faible)
- L'adresse physique $\equiv$ déplacement 32 bits égale à $16 * \textit{selecteur} + \textit{deplacement}$
- Multiplier par 16 en hexa $\equiv$ ajouter un 0 à la droite du nombre (donc sur 20 bits).
- Par exemple, l'adresse physique référencée par 047C :0048 est obtenue de la façon suivante :

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

Mode Réel

- La valeur du sélecteur = numéro de paragraphe (16o)
- Les adresses réelles segmentées ont des inconvénients :

Mode Réel

- La valeur du sélecteur = numéro de paragraphe (16o)
- Les adresses réelles segmentées ont des inconvénients :
  - Une valeur de sélecteur référence 64Ko de mémoire
  - Que se passe-t-il si un programme a plus de 64Ko de code ?

Mode Réel

- La valeur du sélecteur = numéro de paragraphe (16o)
- Les adresses réelles segmentées ont des inconvénients :
  - Une valeur de sélecteur référence 64Ko de mémoire
  - Que se passe-t-il si un programme a plus de 64Ko de code ?
    ⇒ Le programme doit être divisé en segments de moins de 64Ko. Le passage d'un segment à l'autre nécessite le changement de CS

Généralité Langage Assembleur Programmation assembleur Organisation 80x86 Regitres Adressage Interruptions

Mode Réel

- La valeur du sélecteur = numéro de paragraphe (16o)
- Les adresses réelles segmentées ont des inconvénients :
  - Une valeur de sélecteur référence 64Ko de mémoire
  - Que se passe-t-il si un programme a plus de 64Ko de code ?
  - Problèmes similaires pour de grandes quantités de données et le registre DS.

Généralité Langage Assembleur Programmation assembleur Organisation 80x86 Regitres Adressage Interruptions

Mode Réel

- La valeur du sélecteur = numéro de paragraphe (16o)
- Les adresses réelles segmentées ont des inconvénients :
  - Une valeur de sélecteur référence 64Ko de mémoire
  - Que se passe-t-il si un programme a plus de 64Ko de code ?
  - Problèmes similaires pour de grandes quantités de données et le registre DS.
  - Un octet en mémoire = pas une adresse segmentée unique. L'adresse physique 04808 peut être référencée par 047C :0048, 047D :0038, 047E :0028 ou 047B :0058.
    Cela complique la comparaison d'adresses segmentées.

Généralité Langage Assembleur Programmation assembleur Organisation 80x86 Regitres Adressage Interruptions

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- En mode réel, selecteur = num de paragraphe en mémoire. En mode protégé, c'est un indice dans un tableau de descripteurs

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- selecteur = un indice dans un tableau de descripteurs
- Les programmes sont divisés en segments ; En mode réel, les segments sont à des positions fixes en mémoire et le sélecteur indique le numéro de paragraphe auquel commence le segment. En mode protégé, ils ne sont pas à des positions fixes en mémoire physique. De fait, pas besoin d'être en mémoire du tout

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- selecteur = un indice dans un tableau de descripteurs
- les segments pas en mémoire du tout
- Utilise une technique appelée *mémoire virtuelle*
  - Idée de base = ne garder en mémoire que les programmes et les données actuellement utilisés
  - Le reste étant stocké temporairement sur le disque jusqu'à leur utilisation
  - Les segments sont déplacés entre la mémoire et le disque selon les besoins
  - Le placement des segments en mémoire n'est pas constant mais effectué de façon transparente par le système d'exploitation

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- selecteur = un indice dans un tableau de descripteurs
- les segments pas en mémoire du tout
- Utilise une technique appelée *mémoire virtuelle*
- Le programme n'a pas a être écrit différemment pour que la mémoire virtuelle fonctionne.

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- selecteur = un indice dans un tableau de descripteurs
- les segments pas en mémoire du tout
- Utilise une technique appelée *mémoire virtuelle*
- Le programme n'a pas a être écrit différemment pour que la mémoire virtuelle fonctionne.
- Chaque segment est assigné à une entrée dans un tableau de descripteurs.
  - Cette entrée contient toutes les informations dont le système a besoin à propos du segment.
  - e.g. est-il actuellement en mémoire ? à quel endroit ? ses droits d'accès ?
  - L'indice de l'entrée du segment est la valeur du sélecteur stockée dans les registres de segment.

Mode Protégé 16 bits

- Dans mode protégé 16 bits du 80286, les valeurs du sélecteur sont interprétées de façon totalement différente par rapport au mode réel.
- selecteur = un indice dans un tableau de descripteurs
- les segments pas en mémoire du tout
- Utilise une technique appelée *mémoire virtuelle*
- Le programme n'a pas a être écrit différemment pour que la mémoire virtuelle fonctionne.
- Chaque segment est assigné à une entrée dans un tableau de descripteurs.
- L'inconvénient est que les déplacements sont toujours des quantités sur 16 bits.
  $\Rightarrow$ taille de segment toujours $\leq$ 64Ko.
  Cela rend l'utilisation de grands tableaux problématique

Mode Protégé 32 bits

- Mode introduit dès le 80386
- 2 différences avec le 16 bits du 286 :
  - Déplacements étendus à 32 bits.
    Permettant un déplacement jusqu'à 4 milliards
    taille des segments $\leq$ 4Go.
  - Segments divisables en pages (i.e. unité de 4Ko)
    Le système de mémoire virtuelle utlise les pages plutôt que les segments
    $\Rightarrow$ Chargement en mémoire que de certaines parties d'un segment
    En mode 16 bits, tout le segment ou rien
    Ce qui n'aurait pas été pratique avec les segments plus grands du mode 32 bits
- Dans Windows 3.x, le mode standard (resp. amélioré) $\equiv$ mode protégé 16 (resp. 32) bits
- Windows 9X, Windows NT/2000/XP, OS/2 et Linux fonctionnent tous en mode protégé 32 bits paginé

Interruptions

- Le flot ordinaire d'un programme doit pouvoir être interrompu pour traiter des évènements néccessitant une réponse rapide
- $\Rightarrow$ un mécanisme appelé interruptions
- e.g. lorsqu'une souris est déplacée, elle interrompt le programme en cours pour gérer le déplacement de la souris

Interruptions

- Elles provoquent le passage du contrôle à un gestionnaire d'interruptions
- Les gestionnaires d'interruptions = routines traitant une interruption
- un type d'interruption = un nombre entier
- Au début de la mémoire physique, réside un tableau de vecteurs d'interruptions contenant les adresses segmentées des gestionnaires d'interruption.
- Le numéro d'une interruption = un indice dans ce tableau.

Interruptions

- Les interruptions externes proviennent de l'extérieur du processeur (e.g. la souris)
- Le cas de beaucoup de périphériques d'E/S (e.g. le clavier, CD-ROM, ...)
- Les interruptions internes sont soulevées depuis le processeur, à cause d'une erreur ou d'une instruction d'interruption
- Les interruptions erreur sont également appelées traps
- Les interruptions générées par l'instruction d'interruption sont également appelées interruptions logicielles

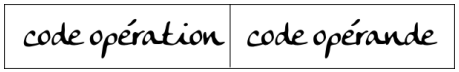
Interruptions

- Le DOS utilise ce type d'interruption pour implémenter son Application Programming Interface
- Les systèmes d'exploitation plus récents (comme Windows et Unix) utilisent une interface basée sur C
- Beaucoup de gestionnaires d'interruptions redonnent le contrôle au programme interrompu lorsqu'ils se terminent
- Ils restaurent tous les registres aux valeurs qu'ils avaient avant l'interruption
- Le programme interrompu s'exécute comme si rien n'était arrivé (excepté qu'il perd quelques cycles processeur)
- Les traps ne reviennent généralement jamais et arrêtent le programme

Plan

Le langage machine

- un type de processeur = son propre langage machine
- Instructions et opérandes stockés en mémoire principale.
- Taille totale d'une instruction dépend de son type et du type d'opérande.
- Toujours codée sur un nombre entier d'octets (facilite décodage)
- Composée de deux champs :
  - le code opération, identifiant de l'instruction à réaliser ;
  - le champ opérande, contenant la donnée, ou sa référence en mémoire



Le langage machine

- Programmation directe du processeur avec les instructions machines :
  - Difficile et long
  - Compréhension quasi-impossible
- Par exemple, EAX = EAX + EBX est encodée par :

  `03 C3`
- Solution : Utilisation d'un langage de plus haut niveau associant un mnémonique à chaque instruction machine : l'assembleur

Langage Assembleur

- Un programme en langage d'assembleur = fichier texte
- Une instruction assembleur = une instruction machine
- Par exemple, EAX = EAX + EBX ≡ `add eax, ebx`
- Signification plus claire qu'en code machine
- Le mot `add` = mnémonique pour l'instruction d'addition
- Forme générale d'une instruction assembleur :

  `mnémonique opérande(s)`

Langage Assembleur

- Un assembleur = programme convertissant un fichier texte contenant des instructions assembleur en code machine
- Un compilateur = programme opérant des conversions similaires pour les langages de haut niveau
- Assembleur = plus simple qu'un compilateur
- Une instruction assembleur = une instruction machine (pas le cas des langages de haut niveau)
- Chaque type de processeur ayant son propre langage machine, il a également son propre langage d'assemblage
- ⇒ Portage entre différentes architectures d'ordinateur plus difficile
- Dans ce cours, on étudie le Netwide Assembler (NASM)

Opérandes d'instruction

- Les instructions ont un nombre et un type variables d'opérandes
- Mais, en général, en nombre fixé (0 à 3)
- Les opérandes peuvent avoir les types suivants :
  - registre : directement référence au contenu des registres
  - mémoire : référence aux données en mémoire
    L'adresse = constante codée en dur ou calculée en utilisant les valeurs des registres
    Les adresses = toujours des déplacements relatifs au début d'un segment
  - immédiat : = des valeurs fixes listées dans l'instruction elle-même
    stockées dans l'instruction (dans le segment de code), pas dans le segment de données
  - implicite : pas entrés explicitement
    e.g. l'incrémentation (le un est implicite)

Instructions de base

- L'instruction la plus basique = `MOV`
- Déplace les données d'un endroit à un autre

```
mov dest, src
```

- La donnée spécifiée par `src` est copiée vers `dest`
- Restriction 1 = `src` et `dest` pas tous deux des opérandes mémoire
- Restriction 2 = `src` et `dest` même taille
  e.g. AX ne peut pas être stockée dans BL
- ⇒ Souvent des règles quelque peu arbitraires sur la façon dont les différentes instructions sont utilisées

Instructions de base

- Quelques exemples (point-virgule = un commentaire) :

```
mov eax, 3 ; stocke 3 dans le registre EAX
mov bx, ax ; stocke la valeur de AX dans BX
```

- ADD = additionner des entiers

```
add eax, 4 ; eax = eax + 4
add al, ah ; al = al + ah
```

- SUB = soustraire des entiers

```
sub bx, 10 ; bx = bx - 10
sub ebx, edi ; ebx = ebx - edi
```

- INC et DEC incrémentent ou décrémentent les valeurs de 1
  Le un étant implicite, le code machine pour INC et DEC est plus petit que ADD et SUB équivalentes

```
inc ecx ; ecx++
dec dl ; dl--
```


Directives

- Directive destinée à l'assembleur, pas au processeur
- Pour indiquer à l'assembleur une tâche à faire ou l'informer
- Pas traduites en code machine
- Utilisations courantes :
  - Définition de constantes
  - Définition de mémoire pour stocker des données
  - Grouper la mémoire en segment
  - Inclure des codes sources de façon conditionnelle
  - Inclure d'autres fichiers
- Le code NASM est analysé par un préprocesseur
- Semblable au préprocesseur C, mais elles commencent par un % au lieu d'un # comme en C

La directive equ

- equ peut être utilisée pour définir un symbole
- Un symbole = constantes nommées utilisables dans le programme assembleur
- Format :

```
symbole equ valeur
```

- Valeur d'un symbole n'est pas redéfinissable

La directive %define

- Semblable à la directive #define du C
- Souvent utilisée pour définir des macros

```
%define SIZE 100
mov eax, SIZE
```

- Macro plus flexible que symbole
  - Redéfinissable
  - Plus complexe que des nombres constants

Directives de données

- Utilisées dans les segments de données pour réserver de la place en mémoire
- 2 façons de réserver
  - Une des directives RESX

Directives de données

- Utilisées dans les segments de données pour réserver de la place en mémoire
- 2 façons de réserver
  - Une des directives RESX
    ne fait qu'allouer la place pour les données dont la taille est donnée par X

Unité	Lettre
octet	B
mot	W
double mot	D
quadruple mot	Q
dix octets	T

Directives de données

- Utilisées dans les segments de données pour réserver de la place en mémoire
- 2 façons de réserver
  - Une des directives RESX
    ne fait qu'allouer la place pour les données dont la taille est donnée par X

Unité	Lettre
octet	B
mot	W
double mot	D
quadruple mot	Q
dix octets	T

  - Une des directives DX
    alloue la place et donne une valeur initiale suivant X

Directives de données

- Utilisées dans les segments de données pour réserver de la place en mémoire
- 2 façons de réserver
  - Une des directives RESX
    ne fait qu'allouer la place pour les données dont la taille est donnée par X

Unité	Lettre
octet	B
mot	W
double mot	D
quadruple mot	Q
dix octets	T

  - Une des directives DX
    alloue la place et donne une valeur initiale suivant X
- Très courant de marquer les emplacements mémoire avec des labels ; cela permet de faire référence facilement aux emplacements mémoire dans le code

Directives de données

- Exemples :

```
L1 db 0 ; octet libelle L1 = 0
L2 dw 1000 ; mot labelle L2 = 1000
L3 db 110101b ; octet = valeur binaire 110101 (53_10)
L4 db 12h ; octet = valeur hexa 12 (18_10)
L5 db 17o ; octet = valeur octale 17 (15_10)
L6 dd 1A92h ; double mot = valeur hexa 1A92
L7 resb 1 ; 1 octet non initialise
L8 db "A" ; octet = code ASCII du A (65)
```

- Les doubles et simples quotes sont traitées de la même façon
- Les définitions de données consécutives sont stockées séquentiellement en mémoire
- $\Rightarrow$ L2 est stocké immédiatement après L1 en mémoire

Directives de données

- Définition de séquences de mémoire

```
L9 db 0, 1, 2, 3 ; definit 4 octets
L10 db "w", "o", "r", ‘‘d’’, 0 ; definit une
chaine "word"
L11 db ’word’, 0 ; idem L10
```

- DD peut être utilisée pour définir à la fois des entiers et des réels en simple précision
- DQ en revanche n'est que pour les réels à double précision
- Pour les grandes séquences, on utilise la directive TIMES qui répète son opérande un certain nombre de fois

```
L12 times 100 db 0 ; equivalent a 100 (db 0)
L13 resw 100 ; reserve de la place pour 100 mots
```


Utilisation des labels

- 2 façons d'utiliser les labels
  - label simple fait référence à l'adresse (ou offset) de la donnée
  - label entre crochets ([]) est interprété comme la donnée à cette adresse

Utilisation des labels

- 2 façons d'utiliser les labels
- label = pointeur vers la donnée et les crochets déréférencent le pointeur

Utilisation des labels

- 2 façons d'utiliser les labels
- label = pointeur vers la donnée et les crochets déréférencent le pointeur
- En mode 32 bits, les adresses sont sur 32 bits. Voici quelques exemples :

```
1 mov al, [L1] ; Copie l'octet situe en L1 dans AL
2 mov eax, L1 ; EAX = addresse de l'octet en L1
3 mov [L1], ah ; copie AH dans l'octet en L1
4 mov eax, [L6] ; copie le double mot en L6 dans EAX
5 add eax, [L6] ; EAX = EAX + double mot en L6
6 add [L6], eax ; double mot en L6 += EAX
7 mov al, [L6] ; copie le premier octet du double mot
en L6 dans AL
```


Utilisation des labels

- 2 façons d'utiliser les labels
- label = pointeur vers la donnée et les crochets déréférencent le pointeur
- En mode 32 bits, les adresses sont sur 32 bits. Voici quelques exemples :

```
1 mov al, [L1] ; Copie l'octet situe en L1 dans AL
2 mov eax, L1 ; EAX = addresse de l'octet en L1
3 mov [L1], ah ; copie AH dans l'octet en L1
4 mov eax, [L6] ; copie le double mot en L6 dans EAX
5 add eax, [L6] ; EAX = EAX + double mot en L6
6 add [L6], eax ; double mot en L6 += EAX
7 mov al, [L6] ; copie le premier octet du double mot
en L6 dans AL
```

- La ligne 7 montre une propriété importante de NASM. L'assembleur ne garde pas de trace du type de données auquel se réfère le label. C'est au programmeur de s'assurer de sa bonne utilisation.

Utilisation des labels

- Adresses souvent stockées dans registres (≡pointeur en C)
- ⇒ Là encore, aucune vérification
- Par exemple,

```
mov [L6], 1 ; stocke 1 en L6
```

qui produit une erreur `operation size not specified`

- Car l'assembleur ne sait pas si "1" est un octet, un mot ou un double mot
- Pour réparer cela, il faut ajouter un spécificateur de taille :

```
mov dword [L6], 1 ; stocke 1 en L6
```

- Cela indique à l'assembleur de stocker un 1 dans le double mot qui commence en L6.
- Les autres spécificateurs de taille sont : BYTE, WORD, QWORD et TWORD (10 o.) .

Plan

E/S et langage C

- E/S ≡ activités très dépendantes du système
- ⇒ interfaçage avec le matériel
- Le C fournit des bibliothèques standards de routines pour les E/S
- Ce que n'offre pas l'assembleur
- ⇒ les routines assembleur sont interfacées avec du C
- Cependant, il faut connaître les conventions d'appels des routines que le C utilise
- Assez compliqué ...

E/S et langage C

- Pour simplifier les E/S, on utilisera les routines de Paul A. Carter (masquent les conventions)

`print_int`	affiche à l'écran la valeur d'un entier stocké dans EAX
`print_char`	affiche à l'écran le caractère dont le code ASCII est stocké dans AL
`print_string`	affiche à l'écran le contenu de la chaîne à l'adresse stockée dans EAX. La chaîne doit être une chaîne de type C (i.e. terminée par 0).
`print_nl`	affiche à l'écran un caractère de nouvelle ligne.
`read_int`	lit un entier au clavier et le stocke dans le registre EAX.
`read_char`	lit un caractère au clavier et stocke son code ASCII dans le registre EAX.

- Ces routines préservent les valeurs de tous les registres, excepté les routines read qui modifient EAX

E/S : utilisation

`print_int`	affiche à l'écran la valeur d'un entier stocké dans EAX
`print_char`	affiche à l'écran le caractère dont le code ASCII est stocké dans AL
`print_string`	affiche à l'écran le contenu de la chaîne à l'adresse stockée dans EAX. La chaîne doit être une chaîne de type C (i.e. terminée par 0).
`print_nl`	affiche à l'écran un caractère de nouvelle ligne.
`read_int`	lit un entier au clavier et le stocke dans le registre EAX.
`read_char`	lit un caractère au clavier et stocke son code ASCII dans le registre EAX.

- Utilisez la directive du préprocesseur `%include`

```
%include "asm_io.inc"
```

- Pour chaque routine,
  - il faut charger EAX avec la valeur correcte
  - utiliser une instruction CALL pour l'invoquer

Débogage

- En assembleur, le débogage consiste essentiellement à tracer l'état des registres et de la mémoire
- Macros utilisées comme des instructions ordinaires dont les opérandes sont séparés par des virgules
- `dump_regs A` affiche les valeurs des registres (en hexadécimal) ainsi que les bits positionnés du registre FLAGS. "A" est un entier qui est affiché également (pour distinguer 2 appels)

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Débogage

- En assembleur, le débogage consiste essentiellement à tracer l'état des registres et de la mémoire
- Macros utilisées comme des instructions ordinaires dont les opérandes sont séparés par des virgules
- `dump_regs A`
- `dump_mem A, B, C` affiche les valeurs d'une région de la mémoire (en hexadécimal et ASCII). "A" est un entier utilisé pour étiqueter la sortie, "B" est l'adresse à afficher (cela peut être un label), "C" est le nombre de paragraphes de 16 octets à afficher après

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Débogage

- En assembleur, le débogage consiste essentiellement à tracer l'état des registres et de la mémoire
- Macros utilisées comme des instructions ordinaires dont les opérandes sont séparés par des virgules
- `dump_regs A`
- `dump_mem A, B, C`
- `dump_stack A, B, C` affiche les valeurs de la pile du processeur autour de l'adresse contenue dans EBP (B double mots après et C double mots avant). "A" est un entier utilisé pour étiqueter la sortie

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Débogage

- En assembleur, le débogage consiste essentiellement à tracer l'état des registres et de la mémoire
- Macros utilisées comme des instructions ordinaires dont les opérandes sont séparés par des virgules
- `dump_regs A`
- `dump_mem A, B, C`
- `dump_stack A, B, C`
- `dump_math A` affiche les valeurs des registres du coprocesseur arithmétique. "A" est un entier utilisé pour étiqueter la sortie

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Créer un Programme

- Aujourd'hui, il est très rare de créer un programme autonome écrit complètement en langage assembleur
- L'assembleur est utilisé pour optimiser certaines routines critiques
- Pourquoi apprendre l'assembleur ?
  1. Quelques fois, le code écrit en assembleur peut être plus rapide et plus compact que le code généré par un compilateur.
  2. L'assembleur permet l'accès à des fonctionnalités matérielles du système directement qu'il pourrait être difficile ou impossible à utiliser depuis un langage de plus haut niveau.
  3. Apprendre à programmer en assembleur aide à acquérir une compréhension plus profonde de la façon dont fonctionne un ordinateur.
  4. Apprendre à programmer en assembleur aide à mieux comprendre comment les compilateurs et les langage de haut niveau comme C fonctionnent.

Programme de lancement en C

- Programme de lancement en `launch.c`

```
int main()
{
 int ret_status;
 ret_status = asm_main();
 return ret_status;
}
```

- Appel simplement `asm_main`
- C'est la routine qui sera écrite en assembleur
- Avantages :
  - Cela laisse le système du C initialiser le programme de façon à fonctionner correctement en mode protégé
  - Tous les segments et les registres correspondants seront initialisés par le C
  - La bibliothèque du C pourra être utilisée par le code assembleur (e.g. les routines d'E/S)

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- Programme affichant un nombre entré au clavier

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `%include "asm_io.inc"` : inculsion de la bibliotèque d'E/S

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `segment .data` :
  définit une section spécifiant la mémoire à stocker dans le segment de données. Seules les données initialisées doivent être définies dans ce segment

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `segment .data` :
  définit une section spécifiant la mémoire à stocker dans le segment de données. Seules les données initialisées doivent être définies dans ce segment
- `prompt1 db "Enter a number : ", 0` et
  `outmsg1 db "You entered ", 0` :
  déclaration de chaînes de caractères (terminées par un octet nul comme en C)

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `segment .bss` :
  définit un segment contenant les données non initialisées
  `input1 resd 1` est un label $\equiv$ double mots

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `segment .text` :
  définit le segment de code où sont placées les instructions sont placées.

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `segment .text` : définit le segment de code où sont placées les instructions sont placées.
- La directive `global` indique à l'assembleur de rendre le label `asm_main` global (portée interne par défaut)
  Cela permet à ce label d'être accédé de l'extérieur

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `enter 0,0` et `pusha` liés à la convention d'appel de fonction (vus plus tard)

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `mov eax, prompt1`
  `call print_string` :
  charge l'@ de prompt1 dans le reg. eax puis demande son affichage

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

- `call read_int`
  `mov [input1], eax` :
  demande la lecture d'un entier stocké dans reg. eax puis copier dans `input1`

first.asm

```
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0
outmsg1 db    "You entered ", 0

segment .bss
input1  resd 1

segment .text
        global  asm_main
asm_main:
        enter   0,0
        pusha
        mov     eax, prompt1
        call    print_string
        call    read_int
        mov     [input1], eax
        mov     eax, outmsg1
        call    print_string
        mov     eax, [input1]
        call    print_int
        call    print_nl
        popa
        mov     eax, 0
        leave
        ret
```

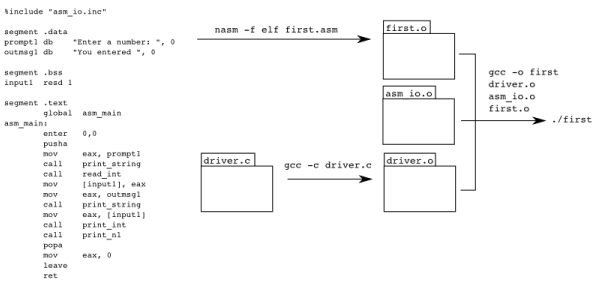
- `popa`
  `mov eax, 0`
  `leave`
  `ret` : liés à la convention d'appel de fonction

Assembler le code

- La première étape consiste à assembler le code
- ⇒ Créer un fichier objet correspondant

```
nasm -f format-objet first.asm
```

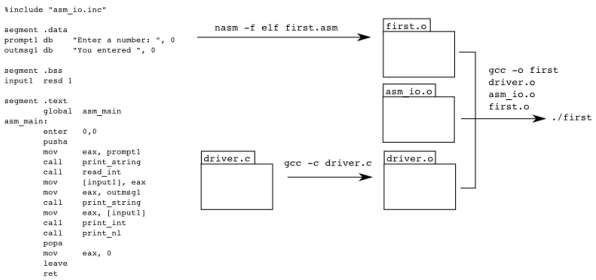
`format-objet` dépend de la platforme et du compilateur C (pour nous `elf` (Executable and Linkable Format))



Compiler le code C

- Compilez le fichier `driver.c` en utilisant un compilateur C
- ⇒ Créer un fichier objet correspondant

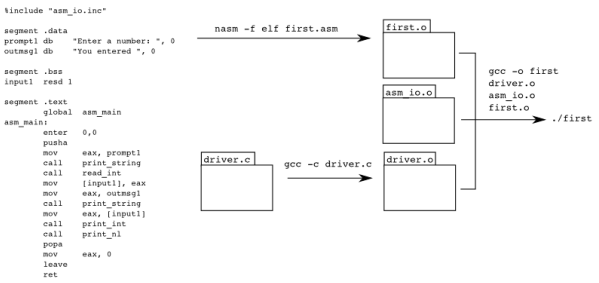
```
gcc -c driver.c
```



Lier les fichiers objets

- L'édition de liens est le procédé qui consiste à combiner le code machine et les données des fichiers objet et des bibliothèques
- ⇒ Créer un fichier exécutable

```
gcc -o first driver.o first.o asm_io.o
```



Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing montrant comment le code a été assemblé (à voir en TD)

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing
- On y retrouve pour les données (1) le déplacement (en hexa) de la donnée dans le segment, (2) les données hexa brutes (valeur ou codes ASCII) ainsi que (3) le code source correspondant

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing
- On y retrouve pour les données (1) le déplacement , (2) les données brutes ainsi que (3) le code source
- ATTENTION : les déplacements ne sont pas les déplacements réels !
  Chaque module (ou ss-prog) peut définir ses propres labels dans le segment de données (et les autres segments également). C'est lors de l'éditions des liens que les déplacements réels sont calculés

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing
- On y retrouve pour les données (1) le déplacement , (2) les données brutes ainsi que (3) le code source
- ATTENTION : les déplacements ne sont pas réels !
- Exemple :

```
94 0000002C A1[00000000] mov eax, [input1]
95 00000037 89C3 mov ebx, eax
```

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing
- On y retrouve pour les données (1) le déplacement , (2) les données brutes ainsi que (3) le code source
- ATTENTION : les déplacements ne sont pas réels !
- Exemple :
  ```
  94 0000002C A1[00000000] mov eax, [input1]
  95 00000037 89C3 mov ebx, eax
  ```
- Souvent le code complet pour une instruction ne peut pas encore être calculé ; e.g. l'adresse de input1 ne sera pas connu avant que le code ne soit lié.
  L'assembleur se contente de calculer l'opcode ici A1 pour mov `[0...0]` correspond à un déplacement temporaire relatif au segment de ce module

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Comprendre un listing assembleur

- L'option `-l fichier-listing` indique à `nasm` de créer un fichier listing
- On y retrouve pour les données (1) le déplacement , (2) les données brutes ainsi que (3) le code source
- ATTENTION : les déplacements ne sont pas réels !
- Exemple :
  ```
  94 0000002C A1[00000000] mov eax, [input1]
  95 00000037 89C3 mov ebx, eax
  ```

- Souvent le code complet pour une instruction ne peut pas encore être calculé
- Le code machine d'instruction ne faisant pas intervenir de label est, en revanche, complet

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Arithmétique en complément à deux

- Un add (ou sub) positionne les bits `carry` et `overflow` de EFLAGS
- En signé, `overflow = 1` si pas assez de bits dans la destination pour coder résultat
- `carry = 1` s'il y a une retenue dans le bit le plus significatif
- $\Rightarrow$ utilisable pour détecter un dépassement de capacité en non signé
- En complément à 2, c'est l'interprétation du résultat qui fait la différence (pas le cas pour mul et div)

002C	44	44
+ FFFF	+ (1)	+ 65535
002B	43	65579*

Il y a une retenue de générée mais elle ne fait pas partie de la réponse en signé

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Généralité Langage Assembleur Programmation assembleur Bases Opérations et contrôles Sous-Programmes

Arithmétique en complément à deux

- Il y a deux instructions de multiplication et de division différentes.
- Pour multiplier, MUL (non signé) ou IMUL (signé)
- Pourquoi deux instructions ?
- Parceque le résultat n'est pas le même !

FF	-1	255
* FF	* (-1)	* 255
????	1 (0001)	65025 (FE01)

G. Blin - Architecture des ordinateurs Assembleur x86 32 bits

Arithmétique en complément à deux

- +eurs formes pour la multiplication
- La plus ancienne

```
mul source
```

où source = registre/ mais pas un immédiat

taille source	opération		résultat
un octet	source * AL	$\rightarrow_{16b}$	AX
16 bits	source * AX	$\rightarrow_{32b}$	DX : AX
32 bits	soure * EAX	$\rightarrow_{64b}$	EDX : EAX

Arithmétique en complément à deux

- +eurs formes pour la division
- La plus ancienne

```
div source
```

où source = registre/ mais pas un immédiat

taille source	opération		reste	quotient
un octet	AX / source	$\rightarrow_{8b}$	AH	AL
16 bits	DX : AX / source	$\rightarrow_{16b}$	DX	AX
32 bits	EDX : EAX / source	$\rightarrow_{32b}$	EDX	EAX

- Attention : il faut donc penser à initialiser DX ou EDX avant la division
- Division par 0 ou un quotient trop grand pour destination interromp le programme

Arithmétique des grands nombres

- Addition et soustraction sur des nombres de plus de 32 bits ?
- $\Rightarrow$ Idée simple : utiliser la retenue générée par add et sub en morcelant l'opération en sous-opérations de 32 bits
- ADC op1, op2 : op1 = op1 + carry + op2
- SBB op1, op2 : op1 = op1 - carry - op2
- Exemple :
  EDX : EAX = EDX : EAX + EBX : ECX (sur 64 bits)

```
add eax, ecx ; eax = eax + ecx
adc edx, ebx ; edx = edx + ebx + carry
```

- Avec une boucle et l'instruction CLC (clear carry) pour de + grands nombres

Avant propos

- Le processeur n'a aucune idée de ce qu'un octet en particulier (ou un mot ou un double mot) est supposé représenté.
- L'assembleur n'a pas le concept de types qu'un langage de plus haut niveau peut avoir.
- La façon dont les données sont interprétées dépendent de l'instruction dans laquelle on les utilise.

Structures de Contrôle

- Les langages de haut niveau fournissent des structures de contrôle de haut niveau (e.g. if et while) qui contrôlent le flot d'exécution
- Le langage assembleur ne fournit pas de structures de contrôle aussi complexes.
- Les structures de contrôle décident ce qu'il faut faire en comparant des données.
- En assembleur, le résultat d'une comparaison est stocké dans le registre FLAGS pour être utilisé plus tard.
- Le 80x86 fournit l'instruction CMP pour effectuer des comparaisons.

L'instruction CMP

- Le registre FLAGS est positionné selon la différence entre les deux opérandes de l'instruction CMP.
- Les opérandes sont soustraites et le registre FLAGS est positionné selon le résultat, mais ce résultat n'est stocké nulle part.
- Si vous avez besoin du résultat, utilisez SUB
- Pour les entiers non signés, il y a 2 drapeaux importants : zéro (ZF) et retenue (CF)
- Pour `cmp A, B`
  - A=B $\Rightarrow$ ZF=1, CF=0
  - A>B $\Rightarrow$ ZF=CF=0
  - A<B $\Rightarrow$ ZF=0, CF=1

L'instruction CMP

- Le registre FLAGS est positionné selon la différence entre les deux opérandes de l'instruction CMP.
- Les opérandes sont soustraites et le registre FLAGS est positionné selon le résultat, mais ce résultat n'est stocké nulle part.
- Si vous avez besoin du résultat, utilisez SUB
- Pour les entiers non signés, zéro (ZF) et retenue (CF)
- Pour les entiers signés, zéro (ZF) overflow (OF) et signe (SF) [1 si < 0]
- Pour `cmp A, B`
  - A=B $\Rightarrow$ ZF=1
  - A>B $\Rightarrow$ ZF=0, SF=OF
  - A<B $\Rightarrow$ ZF=0, SF$\neq$OF

Instructions de branchement

- Elles peuvent transférer l'exécution à n'importe quel point du programme
- De deux types : conditionnel et inconditionnel
- inconditionnel = effectue toujours le branchement
- conditionnel = effectue le branchement selon les drapeaux du registre FLAGS.
- Si le branchement n'est pas effectué, le contrôle passe à l'instruction suivante.

L'instruction JMP

- Effectue les branchements inconditionnels
- Son seul argument est le label de code où se brancher
- L'instruction immédiatement après l'instruction JMP ne sera jamais exécutée à moins qu'une autre instruction ne se branche dessus !
- Plusieurs variantes de saut en utilisant le mot clé immédiatement avant l'étiquette :
  - SHORT limité en portée (moins de 128o) ; utilise moins de mémoire
  - NEAR par défaut ; pour sauter vers n'importe quel emplacement dans un segment
  - FAR permet de passer le contrôle à un autre segment de code ; très rare en mode protégé 386.

Branchement conditionnel

- Beaucoup d'instructions différentes prennant toutes une étiquette de code comme seule opérande
- Les plus simples se contentent de regarder un drapeau dans le registre FLAGS pour déterminer si elles doivent sauter ou non.

JZ	si ZF est allumé	JNS	si SF est éteint
JNZ	si ZF est éteint	JC	si CF est allumé
JO	si OF est allumé	JNC	si CF est éteint
JNO	si OF est éteint	JP	si PF est allumé
JS	si SF est allumé	JNP	si PF est éteint

- PF est le drapeau de parité qui indique si le nombre de bits dans les 8 bits de poids faible du résultat est pair ou impair

Branchement conditionnel

- Le pseudo-code suivant :

```
if ( EAX == 0 )
 EBX = 1 ;
else
 EBX = 2 ;
```

- peut être écrit de cette façon en assembleur :

```
  cmp eax, 0 ; ZF=1 si eax - 0 = 0
  jz thenblock ; si ZF=1, branchement vers
thenblock
  mov ebx, 2 ; partie ELSE du IF
  jmp next ; saute par dessus la partie THEN du IF
thenblock :
  mov ebx, 1 ; partie THEN du IF
next :
```


Branchement conditionnel

- Plus dur ...

```
if ( EAX >= 5 )
 EBX = 1 ;
else
 EBX = 2 ;
```

- Compliqué car si EAX est plus grand ou égal à 5, ZF peut être allumé ou non et SF sera égal à OF.
- Heureusement, le 80x86 fournit des instructions additionnelles

Signé	Non Signé	
JE	JE	si A=B
JNE	JNE	si A $\neq$ B
JL, JNGE	JB, JNAE	si A $<$ B
JLE, JNG	JBE, JNA	si A $\leq$ B
JG, JNLE	JA, JNBE	si A $>$ B
JGE, JNL	JAE, JNB	si A $\geq$ B

Branchement conditionnel

- Plus dur ...

```
if ( EAX >= 5 )
 EBX = 1;
else
 EBX = 2;
```

-
```
  cmp eax, 5;
  jge thenblock
  mov ebx, 2
  jmp next
thenblock :
  mov ebx, 1
next :
```


Les instructions de boucle

- Le 80x86 fournit plusieurs instructions destinées à implémenter des boucles de style for.
- Chacune prend une étiquette de code comme seule opérande.
  - LOOP Decrémente ECX, si ECX $= 0$, saute vers l'étiquette
  - LOOPE, LOOPZ Décremente ECX (le registre FLAGS n'est pas modifié), si ECX $\neq 0$ et ZF $= 1$, saute
  - LOOPNE, LOOPNZ Décremente ECX (FLAGS inchangé), si ECX $\neq 0$ et ZF $= 0$, saute
  - Exemple :

```
sum = 0;
for ( i=10; i >0; i--)
 sum += i;
```

peut être traduit en :

```
  mov eax, 0; eax est sum
  mov ecx, 10; ecx est i
loop_start :
  add eax, ecx
  loop loop_start
```


Structures de Contrôle Standards

- Instructions if

```
if ( condition )
 then_block;
else
 else_block;
```

```
; code pour positionner FLAGS
  jxx else_block
  ;;code du bloc then
  jmp endif
else_block :
  ;;; code du bloc else
endif :
```


Structures de Contrôle Standards

- Instructions if
- Boucle while

```
while ( condition ) {
 corps de la boucle;
}
```

```
while :
; code pour positionner FLAGS
  jxx endwhile
  ; body of loop
  jmp while
endwhile :
```

Structures de Contrôle Standards

- Instructions if
- Boucle while
- Boucle do while

```
do {                          | do :
  corps de la boucle ;        |    ; body of loop
                              | ; code pour positionner FLAGS
}while ( condition ) ;        |   jxx do
```


Opérations de Décalage

- En assembleur le programmeur peut manipuler individuellement les bits des données
- Une opération courante sur les bits = `décalage`
- Déplace la position des bits d'une donnée dans une direction donnée
- 2 types de décalages :
  - logique
  - arithmétique

Décalages logiques

- Le type le plus simple de décalage

Original	1 1 1 0 1 0 1 0
Décalé à gauche	1 1 0 1 0 1 0 0
Décalé à droite	0 1 1 1 0 1 0 1

- Notez que les nouveaux bits sont toujours à 0
- `SHL` (resp. `SHR`) décale à gauche (resp. droite)
- Le nombre de positions à décaler peut soit être une constante, soit être stocké dans `CL`
- Le dernier bit décalé de la donnée est stocké dans le drapeau de retenue

Décalages logiques

- Le type le plus simple de décalage

Original	1 1 1 0 1 0 1 0
Décalé à gauche	1 1 0 1 0 1 0 0
Décalé à droite	0 1 1 1 0 1 0 1

- Voici quelques exemples :

```
mov ax, 0C123H
shl ax, 1 ; décale d'1 bit à gauche, ax = 8246H, CF = 1
shr ax, 1 ; décale d'1 bit à droite, ax = 4123H, CF = 0
shr ax, 1 ; décale d'1 bit à droite, ax = 2091H, CF = 1
mov ax, 0C123H
shl ax, 2 ; décale de 2 bits à gauche, ax = 048CH, CF =
1
mov cl, 3
shr ax, cl ; décale de 3 bits à droite, ax = 0091H, CF
= 1
```

Décalages logiques

- Souvent utilisés pour la multiplication et la division rapides
- La multiplication et la division par une puissance de deux sont simples, on dé- cale simplement les chiffres
- Multiplication par $2^k = k$ décalages à gauche
- Division par $2^k = k$ décalages à droite en binaire
- Très basiques mais beaucoup plus rapides que les instructions MUL et DIV correspondantes !
- Utilisés pour multiplier ou diviser des valeurs non signées
- Que faire en cas de valeurs signées ? $\Rightarrow$ un autre type de décalage

Décalages arithmétiques

- Conçus pour permettre à des nombres signés d'être rapidement multipliés et divisés par des puissances de 2
- Assurent un traitement adequat pour le bit de signe
- `SAL` (Shift Arithmetic Left) - = à `SHL` avec un test supplémentaire : tant que le bit de signe n'est pas changé par le décalage, le résultat sera correct.
- `SAR` (Shift Arithmetic Right) - une nouvelle instruction où les nouveaux bits qui entrent par la gauche sont des copies du bit de signe).

Opérations Booléennes Niveau Bit

- Il y a quatre opérateurs booléens courants : AND, OR, XOR et NOT
- Les processeurs supportent ces opérations comme des instructions agissant de façon indépendante sur tous les bits de la donnée en parallèle
- Considérons que l'instruction `mov ax, 0C123H` est effectuée avant chacune des opérations suivantes

```
and   ax, 82F6H    ; ax = 8022H
or    ax, 0E831H   ; ax = E933H
xor   ax, 0E831H   ; ax = 2912H
not   ax           ; ax = 3EDCH
```


L'instruction TEST

- L'instruction TEST effectue un ADD sans stocker le résultat
- Elle positionne le registre FLAGS selon ce que ce dernier aurait été après un AND
- Similaire à CMP qui effectue un SUB sans stocker le résultat

PLEIN d'utilisations qu'on peut voir en L3 mais pas en IR1

Sous-Programmes

- Les fonctions et les procédures sont des exemples de sous-programmes dans les langages de haut niveau.
- Le code appelant le sous-programme et le sous-programme lui-même doivent se mettre d'accord sur la façon de se passer les données
- Ces règles sont appelées *conventions d'appel*
- Nous nous focaliserons sur celles du C qui passe souvent les adresses des données (i.e. des pointeurs) pour permettre au sous-programme d'accéder aux données en mémoire.

Adressage Indirect

- L'adressage indirect permet aux registres de se comporter comme des pointeurs.
- Pour indiquer qu'un registre est utilisé indirectement comme un pointeur, il est entouré par des crochets ([]).
- Par exemple :

```
mov ax, [Data] ; adressage mémoire direct d'un label
mov ebx, Data ; ebx = &Data
mov ax, [ebx] ; ax = *ebx
```

- Comme AX contient un mot, `mov ax, [ebx]` lit un mot commençant à l'adresse stockée dans EBX
- Si AX était remplacé par AL, un seul octet serait lu
- Les registres n'ont pas de types comme les variables en C ; ce sur quoi EBX est censé pointer est totalement déterminé par les instructions utilisées.
- Si EBX est utilisé de manière incorrecte, = souvent pas d'erreur signalée mais le programme ne fonctionnera pas correctement

Exemple de Sous-Programme Simple

- Un sous-programme = comme une fonction en C.
- Un saut peut être utilisé pour appeler le sous-programme, mais le retour présente un problème.
- Si le sous-programme est destiné à être utilisé par différentes parties du programme, il doit revenir à la section de code qui l'a appelé.
- Donc, le retour du sous-programme ne peut pas être codé en dur par un saut vers une étiquette.
- Le code ci-dessous montre comment cela peut être réalisé en utilisant une forme indirecte de l'instruction JMP.
- Cette forme de l'instruction utilise la valeur d'un registre pour déterminer où sauter (donc, le registre agit plus comme un pointeur de fonction du C).

Exemple de Sous-Programme Simple

```
 mov ebx, input1 ; stocke l'adresse de input1 dans ebx
 mov ecx, ret1 ; stocke l'adresse de retour dans ecx
 jmp short get_int ; lit un entier
ret1 :
 mov ebx, input2
 mov ecx, $ + 7 ; ecx = cette addresse + 7
 jmp short get_int
...
 ; subprogram get_int
 ; Paramètres :
 ; ebx - addresse du dword dans lequel stocker l'entier
 ; ecx - addresse de l'instruction vers laquelle retourner
 ; Notes :
 ; la valeur de eax est perdue
get_int :
 call read_int
 mov [ebx], eax ; stocke la saisie en mémoire
 jmp ecx ; retour à l'appelant
```

Exemple de Sous-Programme Simple

- Le sous-programme `get_int` utilise une convention d'appel simple, basée sur les registres.
- EBX doit contenir l'adresse du DWORD dans lequel stocker le nombre saisi et ECX l'adresse de l'instruction vers laquelle retourner.
- Pour renseigner ECX, on peut utiliser un label (e.g. `ret1`) ou l'opérateur $ qui retourne l'adresse de la ligne sur laquelle il apparaît
- Un peu compliqué ou requiérant autant d'étiquettes que d'appels du sous-programme.
- $\Rightarrow$ Utilisation de la pile.

La pile

- Beaucoup de processeurs ont un support intégré pour une pile.
- Une pile est une liste Last-In First-Out
- La pile est une zone de la mémoire qui est organisée de cette façon.
- L'instruction PUSH ajoute des données à la pile et l'instruction POP retire une donnée.
- La donnée retirée est toujours la dernière donnée ajoutée.
- Le registre de segment SS spécifie le segment qui contient la pile.
- Le registre ESP contient l'adresse de la donnée qui sera retirée de la pile.
- On dit que cette donnée est au *sommet* de la pile.

La pile

- Les données ne peuvent être ajoutées que par unités de double mots.
- L'instruction PUSH insère un double mot sur la pile en ôtant 4 de ESP puis en stockant le double mot en [ESP].
- L'instruction POP lit le double mot en [ESP] puis ajoute 4 à ESP.
- Le code ci-dessous montre comment fonctionnent ces instructions en supposant que ESP vaut initialement `1000H`.

```
push dword 1 ; 1 est stocké en 0FFCh, ESP = 0FFCh
push dword 2 ; 2 est stocké en 0FF8h, ESP = 0FF8h
push dword 3 ; 3 est stocké en 0FF4h, ESP = 0FF4h
pop eax ; EAX = 3, ESP = 0FF8h
pop ebx ; EBX = 2, ESP = 0FFCh
pop ecx ; ECX = 1, ESP = 1000h
```


La pile

- La pile peut être utilisée comme un endroit approprié pour stocker des données temporairement.
- Elle est également utilisée pour effectuer des appels de sous-programmes, passer des paramètres et des variables locales.
- Le 80x86 fournit également une instruction, `PUSHA`, qui empile les valeurs des registres EAX, EBX, ECX, EDX, ESI, EDI et EBP (pas dans cet ordre).
- L'instruction `POPA` peut être utilisée pour les dépiler tous.

Les Instructions CALL et RET

- Le 80x86 fournit deux instructions qui utilisent la pile pour effectuer des appels de sous-programmes rapidement et facilement.
- L'instruction CALL effectue un saut inconditionnel vers un sous-programme et empile l'adresse de l'instruction suivante
- L'instruction RET dépile une adresse et saute à cette adresse.
- Lors de l'utilisation de ces instructions, il est très important de gérer la pile correctement afin que l'adresse dépilé par l'instruction RET soit correcte.
- Exemple :

```
 mov ebx, input1
 call get_int
get_int :
...
 call read_int
 mov [ebx], eax
 ret
```


Les Instructions CALL et RET

- Avantages à utiliser CALL et RET :
  - C'est plus simple !
  - Cela permet d'imbriquer des appels de sous-programmes facilement.
- Notez que get_int appelle read_int.
- Cet appel empile une autre adresse.
- A la fin du code de read_int se trouve un RET qui dépile l'adresse de retour et saute vers le code de get_int.
- Puis, lorsque le RET de get_int est exécuté, il dépile l'adresse de retour qui revient vers asm_main.
- Cela fonctionne correctement car il s'agit d'une pile LIFO.
- Souvenez vous, il est très important de dépiler toute donnée qui est empilée.

Les Instructions CALL et RET

- Par exemple, considérons le code suivant :

```
get_int :
 call read_int
 mov [ebx], eax
 push eax
 ret ; dépile la valeur de EAX, pas l'adresse de
retour ! !
```

- Ce code ne reviendra pas correctement !

Conventions d'Appel

- Lorsqu'un sous-programme est appelé, le code appelant et le sous-programme doivent s'accorder sur la façon de se passer les données.
- Les langages de haut niveau ont des manières standards de passer les données appelées *conventions d'appel*.
- Pour interfacer du code de haut niveau avec le langage assembleur, le code assembleur doit utiliser les mêmes conventions que le langage de haut niveau.
- Les conventions d'appel peuvent différer d'un compilateur à l'autre ou peuvent varier selon la façon dont le code est compilé.
- Une convention universelle est que le code est appelé par une instruction CALL et revient par un RET.

Passer les paramètres via la pile

- Ils sont empilés avant l'instruction CALL.
- Si le paramètre doit être modifié par le sous-programme, c'est l'adresse de la donnée qui doit être passée.
- Si la taille du paramètre est inférieure à un double mot, il doit être converti en un double mot avant d'être empilé.
- Les paramètres sur la pile ne sont pas dépilés par le sous-programme mais accédés depuis la pile elle-même.
- $\Rightarrow$ Pourquoi ?
  - Comme ils doivent être empilés avant l'instruction CALL, l'adresse de retour devrait être dépilée avant tout (puis réempilée ensuite).
  - Souvent, les paramètres sont utilisés à plusieurs endroits dans le sous-programme.
    Difficilement conservés dans un registre durant toute la durée du sous-programme.
    Les laisser sur la pile conserve une copie de la donnée en mémoire qui peut être accédée depuis n'importe quel endroit du sous-programme.

Passer les paramètres via la pile

- Considérons un sous-programme auquel on passe un paramètre unique via la pile.
- Lorsque le sous-programme est appelé, la pile ressemble à

ESP + 4	Paramètre
ESP	Adresse de retour

- On peut accéder au paramètre en utilisant l'adressage indirect ([ESP+4])

Passer les paramètres via la pile

- Si la pile est également utilisée dans le sous-programme pour stocker des données, le nombre à ajouter à ESP changera.
- Par exemple, si un DWORD est empilé la pile ressemble à

ESP + 8	Paramètre
ESP + 4	Adresse de retour
ESP	Donnée du sous-programme

- Maintenant, le paramètre se trouve en ESP + 8, et non plus en ESP + 4.
- Donc, utiliser ESP lorsque l'on fait référence à des paramètres peut être une source d'erreurs.

Passer les paramètres via la pile

- Pour résoudre ce problème, le 80386 fournit un autre registre à utiliser : EBP.
- La seule utilité de ce registre est de faire référence à des données sur la pile.
- La convention d'appel C stipule qu'un sous-programme doit d'abord empiler la valeur de EBP puis définir EBP pour qu'il soit égal à ESP.
- Cela permet à ESP de changer au fur et à mesure que des données sont empilées ou dépilées sans modifier EBP.
- A la fin du programme, la valeur originale de EBP doit être restaurée (c'est pourquoi elle est sauvegardée au début du sous-programme).

Passer les paramètres via la pile

- La forme générale d'un sous-programme qui suit ces conventions :

```
etiquette_sousprogramme :
 push ebp ; empile la valeur originale de EBP
 mov ebp, esp ; EBP = ESP
  ; code du sousprogramme
 pop ebp ; restaure l'ancienne valeur de EBP
 ret
```

- Les lignes 2 et 3 constituent le prologue générique d'un sous-programme ; les lignes 5 et 6 son épilogue.
- La pile immédiatement après le prologue ressemble à

ESP + 8	EBP + 8	Parametre
ESP + 4	EBP + 4	Adresse de retour
ESP	EBP	EBP sauvegardé

- Le paramètre est accessible avec [EBP + 8] depuis n'importe quel endroit du programme sans se soucier de ce qui a été empilé entre temps par le sous-programme.

Passer les paramètres via la pile

- Une fois le sous-programme terminé, les paramètres qui ont été empilés doivent être retirés.
- La convention d'appel C spécifie que c'est au code appelant de le faire.
- D'autres conventions sont différentes.
- Par exemple, la convention d'appel Pascal spécifie que c'est au sous-programme de retirer les paramètres ...

Passer les paramètres via la pile

- Un sous-programme utilisant la convention d'appel C peut être appelé comme suit :

```
push dword 1 ; passe 1 en paramètre
call fun
add esp, 4 ; retire le paramètre de la pile
```

- La ligne 3 retire le paramètre de la pile en manipulant directement le pointeur de pile.
- Une instruction POP pourrait également être utilisée, mais cela nécessiterait le stockage d'un paramètre inutile dans un registre.
- En fait, dans ce cas particulier, beaucoup de compilateurs utilisent une instruction POP ECX pour retirer le paramètre.
- Le compilateur utilise un POP plutot qu'un ADD car le ADD nécessite plus d'octets pour stocker l'instruction.
- Cependant, le POP change également la valeur de ECX !

Variables locales sur la pile

- La pile peut être utilisée comme un endroit pratique pour stocker des variables locales.
- C'est exactement ce que fait le C pour les variables normales.
- Utiliser la pile pour les variables est important si l'on veut que les sous-programmes soient réentrants.
- Un programme réentrant fonctionnera qu'il soit appelé de n'importe quel endroit, même à partir du sous-programme lui-même.
- En d'autres termes, les sous-programmes réentrants peuvent être appelés récursivement.

Variables locales sur la pile

- Utiliser la pile pour les variables économise également de la mémoire.
- Les données qui ne sont pas stockées sur la pile utilisent de la mémoire du début à la fin du programme (le C appelle ce type de variables global ou static).
- Les données stockées sur la pile n'utilisent de la mémoire que lorsque le sous-programme dans lequel elles sont définies est actif.
- Les variables locales sont stockées immédiatement après la valeur de EBP sauvegardée dans la pile.
- Elles sont allouées en soustrayant le nombre d'octets requis de ESP dans le prologue du sous-programme.

Variables locales sur la pile

- Le nouveau squelette du sous-programme est :

```
etiquette_sousprogramme :
 push ebp ; empile la valeur originale de EBP
 mov ebp, esp ; EBP = ESP
 sub esp, LOCAL_BYTES ; = nb octets nécessaires pour
les locales
  ; code du sousprogramme
  mov esp, ebp ; désalloue les locales
  pop ebp ; restaure la valeur originalede EBP
 ret
```

- Le registre EBP est utilisé pour accéder à des variables locales.

Variables locales sur la pile

- Considérons la fonction C suivante :

```
void calc_sum( int n, int * sump )
{
 int i , sum = 0 ;
 for ( i=1 ; i <= n ; i++ )
  sum += i ;
 *sump = sum ;
}
```


Variables locales sur la pile

- En assembleur.

```
cal_sum :
 push ebp
 mov ebp, esp
 sub esp, 4 ; fait de la place pour le sum local
 mov dword [ebp - 4], 0 ; sum = 0
 mov ebx, 1 ; ebx (i) = 1
for_loop :
 cmp ebx, [ebp+12] ; i <= n ?
 jnle end_for
 add [ebp-4], ebx ; sum += i
 inc ebx
 jmp short for_loop
end_for :
 mov ebx, [ebp+8] ; ebx = sump
 mov eax, [ebp-4] ; eax = sum
 mov [ebx], eax ; *sump = sum ;
 mov esp, ebp
 pop ebp

 ret
```

Variables locales sur la pile

- La pile après le prologue du programme ressemble à

ESP + 16	EBP + 12	n
ESP + 12	EBP + 8	sump
ESP + 8	EBP + 4	Adresse de retour
ESP + 4	EBP	EBP sauvé
ESP	EBP - 4	sum

- Cette section de la pile qui contient les paramètres, les informations de retour et les variables locales est appelée *cadre de pile* (stack frame).

Variables locales sur la pile

- Chaque appel de fonction C crée un nouveau cadre de pile sur la pile.
- Le prologue et l'épilogue d'un sous-programme peuvent être simplifiés en utilisant deux instructions spéciales qui sont conçues spécialement dans ce but.
- L'instruction ENTER effectue le prologue et l'instruction LEAVE l'épilogue.
- L'instruction ENTER prend deux opérandes immédiates.
- Dans la convention d'appel C, la deuxième opérande est toujours 0.
- La première opérande est le nombre d'octets nécessaires pour les variables locales.
- L'instruction n'a pas d'opérande.

Variables locales sur la pile

- Exemple :

```
etiquette_sousprogramme :
 enter LOCAL_BYTES, 0 ; = nb octets pour les locales
 ; code du sous-programme
 leave
 ret
```


Programme multimodules

- C'est un programme composé de plus d'un fichier objet.
- Souvenez vous que l'éditeur de liens combine les fichier objets en un programme exécutable unique.
- L'éditeur de liens doit rapprocher toutes les références faites à chaque étiquette d'un module (i.e. un fichier objet) de sa définition dans un autre module.
- Afin que le module A puisse utiliser une étiquette définie dans le module B, la directive `extern` doit être utilisée.
- Après la directive `extern` vient une liste d'étiquettes délimitées par des virgules.
- La directive indique à l'assembleur de traiter ces étiquettes comme externes au module.
- C'est-à-dire qu'il s'agit d'étiquettes qui peuvent être utilisées dans ce module mais sont définies dans un autre.

Programme multimodules

- En assembleur, les étiquettes ne peuvent pas être accédées de l'extérieur par défaut.
- Si une étiquette doit pouvoir être accédée depuis d'autres modules que celui dans lequel elle est définie, elle doit être déclarée comme globale dans son module, par le biais de la directive `global`.

Interfacer de l'assembleur avec du C

- Aujourd'hui, très peu de programmes sont écrits complètement en assembleur.
- Les compilateurs sont très performants dans la conversion de code de haut niveau en code machine efficace.
- Comme il est plus facile d'écrire du code dans un langage de haut niveau, ils sont plus populaires.
- De plus, le code de haut niveau est beaucoup plus portable que l'assembleur !
- Lorsque de l'assembleur est utilisé, c'est souvent pour de petites parties du code.

Interfacer de l'assembleur avec du C

- Cela peut être fait de deux façons : en appelant des sous-routines assembleur depuis le C ou en incluant de l'assembleur.
- Inclure de l'assembleur permet au programmeur de placer des instructions assembleur directement dans le code C.
- Cela peut être très pratique ; cependant, il y a des inconvénients à inclure l'assembleur.
- Le code assembleur doit être écrit dans le format que le compilateur utilise.
- Aucun compilateur pour le moment ne supporte le format NASM.

Sous-Programmes Réentrants et Récursifs

- Un sous-programme réentrant rempli les critères suivants :
  - Il ne doit pas modifier son code. Dans un langage de haut niveau, cela serait diffcile mais en assembleur, il n'est pas si dur que cela pour un programme de modifier son propre code (en tout cas en mode réel). Par exemple :
    ```
    mov word [cs :$+7], 5 ; copie 5 dans le mot 7 octets plus loin
    add ax, 2 ; l'expression précédente change 2 en 5 !
    ```
    En mode protégé, le segment de code est marqué en lecture seule. Lorsque la première ligne ci-dessus s'exécute, le programme est interrompu.
  - Il ne doit pas modifier de données globales (comme celles qui se trouvent dans les segments data et bss). Toutes les variables sont stockées sur la pile.

Sous-Programmes Réentrants et Récursifs

- Il y a plusieurs avantages à écrire du code réentrant.
  - Un sous-programme réentrant peut être appelé récursivement.
  - Un programme réentrant peut être partagé par plusieurs processus. Pour plusieurs instances d'un programme en cours, seule une copie du code se trouve en mémoire.
  - Les sous-programmes réentrants fonctionnent beaucoup mieux dans les programmes multi-threadés.

Sous-programmes récursifs

- Ce type de sous-programmes s'appellent eux-mêmes.
- La récursivité peut être soit directe soit indirecte.
- La récursivité directe survient lorsqu'un sous-programme, disons foo, s'appelle lui-même dans le corps de foo.
- La récursivité indirecte survient lorsqu'un sous-programme ne s'appelle pas directement lui-même mais via un autre sous-programme qu'il appelle.
- Par exemple, le sous-programme foo pourrait appeler bar et bar pourrait appeler foo.
- Les sous-programmes récursifs doivent avoir une condition de terminaison.
- Lorsque cette condition est vraie, il n'y a plus d'appel récursif.
- Si une routine récursive n'a pas de condition de terminaison ou que la condition n'est jamais remplie, la récursivité ne s'arrêtera jamais (exactement comme une boucle infinie).

Sous-programmes récursifs

- Exemple du calcul de factorielle :

```
; trouve n !
segment .text
 global _fact
_fact :
 enter 0,0
  mov eax, [ebp+8] ; eax = n
 cmp eax, 1
 jbe term_cond ; si n <= 1, terminé
 dec eax
 push eax
 call _fact ; appel fact(n-1) récursivement
 pop ecx ; réponse dans eax
 mul dword [ebp+8] ; edx :eax = eax * [ebp+8]
 jmp short end_fact
term_cond :
 mov eax, 1
end_fact :
 ret
```


Sous-programmes récursifs

- La pile au point le plus profond pour l'appel de fact :

cadre n=3	n(3) Adresse de retour EBP Sauvé
cadre n=2	n(2) Adresse de retour EBP Sauvé
cadre n=1	n(1) Adresse de retour EBP Sauvé