

Assembleur x86

Eric Cariou

*Université de Pau et des Pays de l'Adour
Département Informatique*

Eric.Cariou@univ-pau.fr

Niveaux de programmation

- ◆ Du plus bas niveau (1) au plus abstrait (6)
 - 1) Impulsions électriques : 0 et 1 dans portes/composants
 - 2) Micro-instructions
 - ◆ Séquencement via unité de commande, chemin de données
 - 3) Micro-opérations internes
 - ◆ Traduction opérations complexes CISC en μ OP de type RISC
 - 4) Langage machine
 - ◆ Codage binaire des opérations : code opération + codes opérands
 - 5) Assembleur
 - ◆ Langage machine avec codes mnémotechniques à la place des codes opération en binaire
 - 6) Langage haut niveau : C, JAVA ...

Niveaux de programmation

- ◆ Le compilateur
 - ◆ Traduit un programme écrit en assembleur ou langage de haut niveau
 - ◆ En un programme en langage machine exécutable par le processeur
- ◆ Assembleur
 - ◆ Le langage de plus bas niveau utilisable par un être humain
 - ◆ Utilise le jeu d'instructions du processeur

Jeu d'instructions

- ◆ Jeu d'instruction
 - ◆ Ensemble des instructions machine (et donc assembleur) qu'un processeur sait exécuter
- ◆ Deux grandes familles de jeu d'instruction
 - ◆ CISC : Complex Instruction Set Computing
 - ◆ Instructions nombreuses et pouvant être complexes et de taille variable
 - ◆ Peu de registres
 - ◆ Exemple : Intel x86
 - ◆ RISC : Reduced Instruction Set Computing
 - ◆ Instructions limitées et simples et de taille fixe
 - ◆ Adressage mémoire : chargement/rangement (que via des registres)
 - ◆ Nombre relativement importants de registres
 - ◆ Exemples : Sun SPARC, IBM PowerPC

Architecture Intel x86

◆ Architecture Intel x86

- ◆ Date du processeur 8086 en 1978
 - ◆ ... et toujours en vigueur aujourd'hui dans les processeurs Intel et AMD
- ◆ Données sur 16 bits
- ◆ Adresses mémoire sur 20 bits : 1 Mo addressable
- ◆ 8 registres généraux de 16 bits

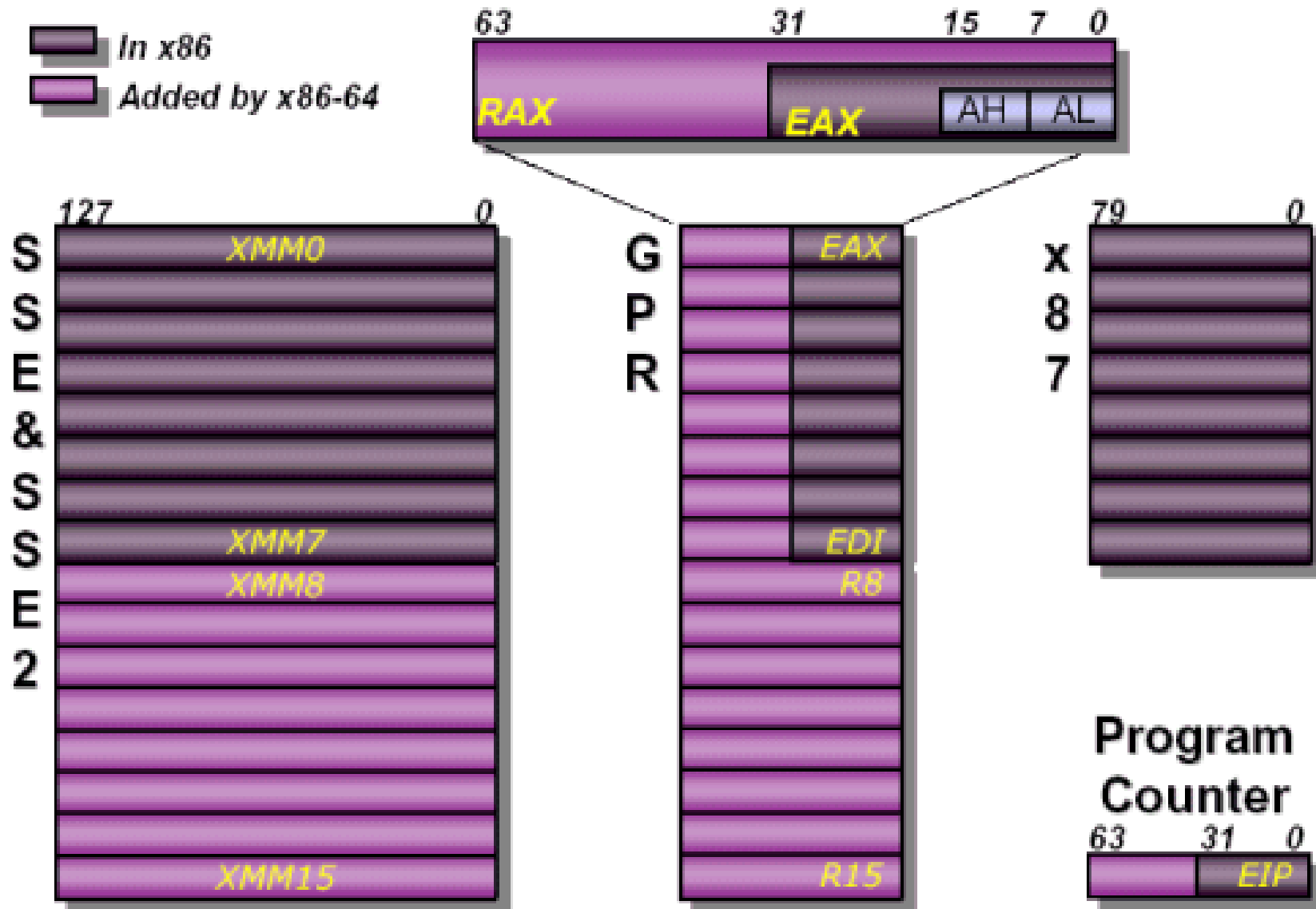
◆ Architecture vieillissante

- ◆ Trop peu de registres et de taille trop petite
- ◆ Evolution de cette architecture avec les nouvelles gammes de processeurs Intel et compatibles
- ◆ En gardant la compatibilité avec architecture originelle x86

Architecture Intel x86

- ◆ Première évolution
 - ◆ Architecture IA32 du processeur Intel 386 en 1985
 - ◆ Passage au 32 bit : 8 registres généraux 32 bits
 - ◆ Adresses sur 32 bits
 - ◆ Ajout de registres, opérations et unités FPU avec le 486 en 1989
- ◆ Evolution récente : exemple du AMD Athlon 64
 - ◆ 16 registres 64 bits
 - ◆ Dont 8 utilisables en version 32 bits en mode IA32
 - ◆ Et 8 utilisables en version 16 bits du x86 originel
 - ◆ Registres multimédia : 16 registres 128 bits
 - ◆ Registres flottants (x87) : 8 registres 80 bits

Registres de l'AMD 64



Registres x86

- ◆ 8 registres 16 bits généraux
 - ◆ AX, BX, CX, DX, DI, SI, BP, SP
- ◆ Chaque registre peut avoir une fonction particulière implicite selon l'instruction utilisée
 - ◆ AX : accumulateur
 - ◆ BX : base (adresse mémoire)
 - ◆ CX : compteur
 - ◆ DX : donnée, entrées/sorties
 - ◆ SI : index source
 - ◆ DI : index destination
 - ◆ BP : base de la pile
 - ◆ SP : pointeur sur le sommet de la pile

Registres x86

- ◆ Les registres en « X » peuvent être utilisés sous la forme de 2 registres 8 bits
 - ◆ $xX = xH + xL$
 - ◆ H : partie haute, L : partie basse
 - ◆ $xX = 256 * xH + xL$
 - ◆ AX = AH et AL
 - ◆ BX = BH et BL
 - ◆ CX = CH et CL
 - ◆ DX = DH et DL

Registres x86

- ◆ Autres registres
 - ◆ IP : pointeur d'instruction
 - ◆ FLAGS : flags (drapeaux 1 bit)
 - ◆ Registres de segments mémoires
 - ◆ CS : code
 - ◆ DS : données (data)
 - ◆ SS : pile (stack)
 - ◆ ES : extra
- ◆ Registres IA32
 - ◆ 8 registres généraux sur 32 bits
 - ◆ Extensions des 8 registres 16 bits en 32 bits
 - ◆ Ajout de la lettre E devant : EAX, EBP, ...

Segments et adresses mémoire

- ◆ Bus d'adresse de l'architecture x86
 - ◆ 20 bits soit 1 Mo de mémoire adressable
 - ◆ Mais registres n'ont que 16 bits
 - ◆ 64 Ko de mémoire adressable
- ◆ Pour avoir des adresses mémoire sur 20 bits avec des registres de 16 bits
 - ◆ On combine 2 registres
 - ◆ Le premier donne l'adresse de base d'un segment
 - ◆ Le second donne le déplacement dans ce segment
 - ◆ Segment = zone mémoire de 64 Ko

Segments et adresses mémoire

- ◆ Adressage sur 20 bits avec 2 registres
 - ◆ 2 registres 16 bits : peut coder adresses sur 32 bits
 - ◆ Pour uniquement 20 bits
 - ◆ Décale le premier registre de 4 bits et l'additionne au second
 - ◆ Adresse notée A:B
 - ◆ Adresse réelle : $A * 16 + B$
 - ◆ Exemple (les nombres sont en hexa)
 - ◆ 3100:27EE correspond à l'adresse $31000 + 27EE = 337EE$
 - ◆ Décaler de 4 bits en binaire revient à décaler d'un chiffre en hexa

Segments et adresses mémoire

- ◆ 4 segments d'adresses : CS, DS, SS, ES
 - ◆ Peut utiliser 2 registres pour adresser des mots mémoires
 - ◆ Le premier est le registre de segment
 - ◆ Le second un registre général
 - ◆ On l'appelle l'offset (décalage)
 - ◆ Adresse : *segment:offset*
 - ◆ Exemples
 - ◆ CS:IP : adresse de la prochaine instruction à exécuter
 - ◆ DS:SI : adresse d'une donnée
 - ◆ SS:SP : adresse du haut de la pile

Segments et adresses mémoire

- ◆ Addressage mémoire en IA32
 - ◆ Adresses mémoire codables sur 32 bits
 - ◆ Segment est codé sur 16 bits
 - ◆ Offset codé sur 32 bits
 - ◆ Extensions possibles pour coder des adresses sur 36 bits
 - ◆ Depuis Pentium Pro en 1995
 - ◆ 64 Go adressables
- ◆ Deux segments supplémentaires en IA32
 - ◆ FS et GS : données
 - ◆ 4 segments de données au total : DS, ES, FS, GS

Segments et adresses mémoire

- ◆ Addressage à plat possible également
 - ◆ Sans passer par les segments
- ◆ Pourquoi des segments ?
 - ◆ A l'origine
 - ◆ Pour pouvoir adresser plus de 64 Ko de mémoire dans un programme
 - ◆ Car registres 16 bits
 - ◆ En pratique aujourd'hui
 - ◆ Permet de séparer clairement des zones mémoires selon leur rôle
 - ◆ Exemple : la pile ne peut pas être écrasée par des données ou déborder sur des données/code
 - ◆ Mais (très) contraignant ...

Modes d'adressage

- ◆ Plusieurs modes d'adressage des données
 - ◆ Par registre
 - ◆ ADD AX, BX
 - ◆ Valeur immédiate
 - ◆ ADD AX, 2
 - ◆ « h » et « b » pour noter que le nombre est en hexa ou binaire
 - ◆ 20h, 100101b
 - ◆ Adressage mémoire
 - ◆ ADD AX, [1000h]
 - ◆ ADD AX, [BX]
 - ◆ Adressage mémoire indexée
 - ◆ ADD AX,[BX+DI]

Registre d'états

- ◆ Le registre d'état
 - ◆ Registre sur 16 bits voire 32 en IA32
 - ◆ Un bit est un drapeau
 - ◆ Propriété vérifiée (=1) ou pas (=0) après l'appel d'une opération
 - ◆ Certains bits n'ont aucune signification
 - ◆ Principaux drapeaux (flags)
 - ◆ CF (bit 0) : opération génère une retenue
 - ◆ Débordement si entiers non signés
 - ◆ PF (bit 2) : résultat de l'opération est pair
 - ◆ ZF (bit 6) : résultat de l'opération est 0
 - ◆ SF (bit 7) : signe du résultat de l'opération (0=positif, 1=négatif)
 - ◆ OF (bit 11) : si opération génère un débordement de capacité (overflow)
 - ◆ Gère les débordements en entiers signés

Opérations

- ◆ Opérations sont classées en plusieurs grandes catégories
 - ◆ Arithmétiques
 - ◆ Addition, multiplication ...
 - ◆ Logiques
 - ◆ Et, ou, xor ...
 - ◆ Transfert de données
 - ◆ Mémoire vers registre, registre vers registre, gestion de la pile...
 - ◆ Décalage et rotations de bits
 - ◆ Instructions de contrôle
 - ◆ Sauts conditionnels, appels de procédures ...
 - ◆ Structure de données
 - ◆ Bit à bit, chaîne de caractères ...

Opérations

- ◆ Signification des opérandes, format général
 - ◆ 2 opérandes au plus
 - ◆ Les opérandes servent de paramètres
 - ◆ Si résultat calculé, la première opérande contient ce résultat
 - ◆ Implicitement, selon l'opération, des registres peuvent être modifiés ou utilisés pour stocker le résultat
 - ◆ Exemple
 - ◆ `ADD AX, BX` ; $AX = AX + BX$
; modifie les flags selon résultat
; (débordement, résultat nul ...)

Opérations arithmétiques (entiers)

- ◆ ADD : addition
- ◆ ADC : addition avec retenue de 1 ajoutée si CF = 1
- ◆ SUB : soustraction
- ◆ SBB : soustraction avec retenue de -1 ajoutée si CF = 1
- ◆ IMUL : multiplication avec entiers signés (résultat sur taille double)
- ◆ MUL : multiplication avec entiers non signés (résultat sur taille double)
- ◆ IDIV : division entière signée
- ◆ DIV : division entière non signée
- ◆ INC : incrémentation de 1
- ◆ DEC : décrémentation de 1
- ◆ NEG : opposé
- ◆ CMP : comparaison sous forme de soustraction sans stockage résultat

Opérations logiques

- ◆ 4 opérations logiques : fonctionnement bit à bit
 - ◆ AND : Et logique
 - ◆ Application d'un masque sur une donnée : garde les bits de la donnée pour lesquels le masque est à 1, sinon met le bit à 0
 - ◆ OR : OU logique
 - ◆ XOR : XOR logique
 - ◆ NOT : inverse les bits
- ◆ Exemples
 - ◆ $0110 \text{ AND } 0011 = 0010$
 - ◆ $0110 \text{ OR } 0011 = 0111$
 - ◆ $0110 \text{ XOR } 0011 = 0101$
 - ◆ $\text{NOT } 0110 = 1001$

Transferts de données

- ◆ Copies de données entre mémoire centrale et registres
- ◆ Une opération unique pour copie : MOV
 - ◆ Plusieurs combinaisons
 - ◆ MOV registre, mémoire
 - ◆ MOV mémoire, registre
 - ◆ MOV registre, registre
 - ◆ MOV mémoire, immédiate
 - ◆ MOV registre, immédiate
 - ◆ Exemples
 - ◆ MOV AX, [BX] ; AX = contenu adresse référencée par BX
 - ◆ MOV [3000h], 'D' ; met caractère D à l'adresse 3000
 - ◆ MOV [BX], DX ; copie la valeur de DX à l'adresse référencée par BX

Décalage et rotations

◆ Décalage

- ◆ Les bits de la donnée sont décalés vers la droite ou la gauche, avec remplissage par 0

◆ Exemple

- ◆ 110011 -> décalage gauche -> 100110
- ◆ On perd le bit 1 de poids fort et on ajoute un 0 en poids faible

◆ Rotation

- ◆ Les bits sont décalés vers la droite ou la gauche
- ◆ Les bits qui « débordent » d'un côté sont copiés de l'autre

◆ Exemple

- ◆ 110011 -> rotation gauche -> 100111
- ◆ Le 1 de poids fort devient 1 de poids faible

Décalage et rotations

- ◆ 8 opérations
 - ◆ SAR : décalage arithmétiques droite
 - ◆ SHR : décalage logique droite
 - ◆ SAL : décalage arithmétique gauche
 - ◆ SHL : décalage logique gauche
 - ◆ ROL : rotation gauche
 - ◆ ROR : rotation droite
 - ◆ RCL : rotation gauche de la donnée + la retenue
 - ◆ RCR: rotation droite de la donnée + la retenue
- ◆ Dans les 8 cas : CF contient le bit qui sort de la donnée
- ◆ Décalage arithmétique : conserve le signe

Instructions de contrôle

- ◆ Par défaut, exécution séquentielle des instructions
- ◆ Possibilité de sauter à d'autres endroits du programme de 2 façons
 - ◆ Saut à une adresse du programme
 - ◆ Saut conditionnel à une adresse du programme
- ◆ Pas de structures de contrôle plus évoluées
 - ◆ Pas de boucles
 - ◆ Pas de *if then else* ou de *switch*

Instructions de contrôle

- ◆ Dans le code du programme en assembleur
 - ◆ Peut placer des labels pour marquer un endroit de la séquence d'instructions
 - ◆ Instruction de saut : JMP
 - ◆ JMP label
 - ◆ Exemple
 - ◆ MOV AX, [1000h]
MOV BX, [1004h]
JMP calcul
suite:
MOV [BX],AX
..
calcul:
ADD AX, BX
JMP suite

Instructions de contrôle

◆ Sauts conditionnels

- ◆ En fonction de l'état de drapeaux positionnés par l'exécution d'une opération
- ◆ Ou après une opération de comparaison : CMP
- ◆ 33 types de saut conditionnel !
- ◆ Exemples de quelques instructions de sauts conditionnels
 - ◆ JE : saut si égal
 - ◆ JZ : saut si résultat est zéro
 - ◆ JG : saut si plus grand
 - ◆ JEG : saut si plus grand ou égal (equal or above)
 - ◆ JNGA : saut si pas égal ou pas plus grand (not equal or above)
 - ◆ JL : saut si plus petit (less)
 - ◆ JC : saut si retenue (carry)
 - ◆ JNO : saut si pas débordement (not overflow)

Instructions de contrôle

◆ On peut avec des sauts conditionnels reproduire des instructions de contrôle de plus haut niveau

◆ If ... then ... else

◆ if (AX = 3)
then BX = 5 else BX = DX

CMP AX, 3

JNE else ; si pas égal, on saute à else

MOV BX, 5 ; sinon on est dans le then

JMP endif ; on saute le then pour aller à la fin du if

else:

MOV BX, DX

endif:

Instructions de contrôle

◆ for (cx=0; cx<5; cx++)
 ax = ax + cx

```
MOV AX,0      ; initialisation de AX et CX à 0
MOV CX,0      ; CX est le compteur de boucle
for:
CMP CX, 5     ; compare CX à 5
JGE endfor    ; si CX >= 5, on sort de la boucle
ADD AX, CX    ; fait le calcul
INC CX       ; CX est incrémenté de 1
JMP for       ; on reprend au début de la boucle
endfor:
```

Instructions de contrôle

- ◆ Pour boucle for précédente, peut utiliser une autre instruction
- ◆ LOOP label
 - ◆ Décrémente CX et fait le saut à label si $CX > 0$
- ◆ for (cx=5; cx>0; cx--)
 ax = ax + cx

```
MOV AX,0
```

```
MOV CX,5 ; CX est le compteur de boucle
```

```
for:
```

```
ADD AX,CX ; fait le calcul
```

```
LOOP for ; décrémente CX. Si  $CX > 0$  fait  
          ; le saut à for
```

Instructions de contrôles

◆ Exemple avec saut sur drapeau

ADD AX, BX

JO erreur

...

erreur:

MOV AX, 0

- ◆ Si débordement (overflow) lors de l'addition de AX et BX, on saute au label erreur

Procédures

- ◆ On peut définir des procédures en assembleur x86
- ◆ Pour les paramètres des procédures et la valeur de retour, 2 méthodes
 - ◆ Utiliser des registres
 - ◆ Simple
 - ◆ Mais registres peu nombreux et doit toujours utiliser les mêmes à chaque appel donc assez contraignant
 - ◆ Passer par la pile
 - ◆ On empile les paramètres de la procédure
 - ◆ On appelle la procédure
 - ◆ On dépile les paramètres et résultats
 - ◆ Plus compliqué mais plus général

Gestion de la pile

- ◆ Gestion de la pile, 4 opérations
 - ◆ PUSH : empiler une valeur sur la pile
 - ◆ POP : dépiler une valeur de la pile
 - ◆ PUSHA : empiler les 8 registres généraux sur la pile
 - ◆ POPA : positionne les valeurs des 8 registres à partir des 8 valeurs au sommet de la pile
 - ◆ PUSHA et POPA permettent de sauvegarder et restaurer simplement l'état des registres
 - ◆ Avant un appel de procédure par exemple
 - ◆ Egalement des opérations pour dépiler/empiler registre d'état
- ◆ Pour lire des éléments de la pile sans les dépiler
 - ◆ `MOV AX, [SP]` ; AX prend la première valeur en sommet de pile
 - ◆ `MOV AX, [SP+2]` ; AX prend la deuxième valeur en sommet de pile

Procédures

◆ Déclaration d'une procédure

```
nomProc PROC
    ...
    instructions de la procédure
    ...
    RET
nomProc ENDP
```

◆ Appel d'une procédure

```
CALL nomProc
```

◆ Quand appelle une procédure, le registre IP est empilé sur la pile pour savoir où revenir

Procédures

◆ Exemple par registre

- ◆ Procédure qui retourne le double de la somme de AX et DX

```
calcul PROC
```

```
    ADD AX, DX    ; addition
```

```
    SHL AX, 1     ; décalage gauche = X 2
```

```
    RET
```

```
calcul ENDP
```

```
.....
```

```
MOV AX, 10
```

```
MOV DX, 20
```

```
CALL calcul      ; après l'appel, AX contient le résultat
```

Procédures

- ◆ Même exemple avec passage propre par pile
- ◆ Propre car les registres restent dans le même état

calcul PROC

PUSH BP	; sauvegarde base de la pile
MOV BP, SP	; nouvelle base de pile = sommet pile
PUSH AX	; sauvegarde AX
MOV AX, [BP+4]	; récupère argument 2
ADD AX, [BP+6]	; addition AX et argument 1
SHL AX, 1	; décalage gauche = X 2
MOV [BP+6],AX	; remplace argument 1 par résultat
POP AX	; restaure AX
POP BP	; restaure la base de la pile
RET	

calcul ENDP

Procédures

- ◆ Appel de la procédure du transparent précédent

```
PUSH 10      ; empile l'argument 1
PUSH 20      ; empile l'argument 2
CALL calcul
POP AX       ; dépile l'argument 2
POP AX       ; AX contient le résultat
```

Définitions segments et variables

- ◆ Dans segment de données : peut définir des variables
- ◆ Variable = zone mémoire

```
Assume CS:Code, DS:Donnees      ; CS est le segment Code  
                                ; DS est le segment Donnees
```

```
Donnees SEGMENT                ; définit segment données  
    var1 dw 12                  ; var1 : variable sur 16 bits (dw) init à 12  
    var2 db ?                    ; var2 : variable sur 8 bits (db) non initialisé  
Donnees ENDS                    ; fin de déclaration du segment
```

```
Code SEGMENT                    ; définit segment code  
    main:                       ; main sera le début du programme
```

...

```
    MOV AX, var1                ; copie de var1 dans AX
```

```
    MOV var2, DH                ; copie de DH dans var2
```

....

```
Code ENDS                        ; fin du segment
```

```
END main                          ; précise que le programme commence au label main
```

Interruptions

- ◆ Interruption
 - ◆ Pour appeler une opération système ou matérielle
 - ◆ Généralement d'entrée/sortie
- ◆ Exemple d'interruption : interruption système DOS 21h
- ◆ Affectation des registres selon la fonction de l'interruption à utiliser

```
MOV DL, "E"
```

```
MOV AH, 2 ; fonction 2 : affichage à l'écran du caractère  
; stocké dans DL
```

```
INT 21h ; appel de l'interruption
```

```
MOV AH, 1 ; fonction 1 : lecture du caractère entré au clavier
```

```
INT 21h ; le caractère lu est placé dans AL
```

Conclusion

- ◆ Intérêt de programmer en assembleur de nos jours
 - ◆ Pour une programmation « standard » : aucun
 - ◆ Les compilateurs produiront toujours un code plus optimisé que celui qu'on écrirait à la main
 - ◆ Le compilateur est généralement optimisé pour une architecture de CPU cible, il sait comment optimiser le code pour un type de CPU donné
 - ◆ Mais utiles dans certains cas précis
 - ◆ Programmation bas-niveau de pilotes de périphériques
 - ◆ Utilisation de processeurs dans des cartes électroniques