Cours Architecture (ASR 2)

Gaëtan Rey

IUT de Nice - Côte d'Azur
Département Informatique

Gaetan.Rey@unice.fr

Objectif du cours

- ✓ De quoi est composé un ordinateur ?
- ✓ Quels sont les modèles sous-jacents au fonctionnement d'une machine ?
- ✓ Comment s'exécutent les programmes ?
- ✓ Quel est le lien entre le logiciel et le matériel ?
- ✓ Comment fonctionnent les divers périphériques ?

Organisation du Cours

- ✓ Les séances
  - 7 x 1h30 de cours
  - 10 x 1h30 de TP
- ✓ Les intervenants
  - Marcela Rivera (4 groupes TP)
  - Gaëtan Rey (1 groupe TP + cours)
- ✓ Les évaluations
  - un devoir surveillé
  - un contrôle possible à chaque séance TP
  - un projet bibliographique

Le Projet (1)

- ✓ Rédiger un article sur un sujet en rapport avec le cours
  - Article technique
  - Sujets complexes et non traités en cours
- ✓ Consignes
  - Respecter les consignes « Hermès »
  - Taille limitée de 8 à 12 pages
- ✓ Attention
  - S'approprier le sujet sans faire du plagiat
  - Référencer et diversifier vos sources.

Le Projet (2)

- ✓ Publication en ligne des meilleurs articles
  - Objectif est de créer une mini revue technique
  - Des questions à l'exam sur les meilleurs articles
- ✓ Pour la semaine prochaine
  - Établir les groupes
  - Choisir et faire valider un sujet par groupe
- ✓ Le reste du projet
  - À faire en dehors des cours
  - A rendre avant le 19 avril à 12h00

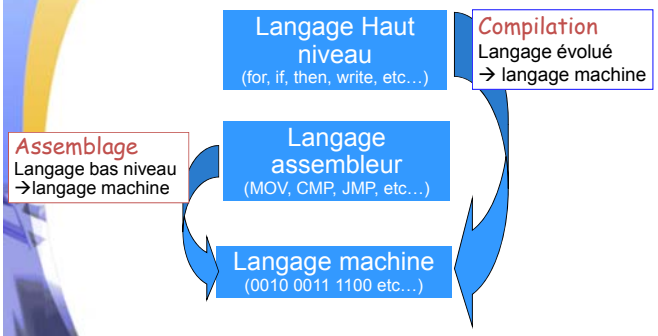
Les sujets possible pour le Projet

- ✓ Les Ecrans plats (Plasma, LCD, SED, OLED…)
- ✓ Les Disques Durs
- ✓ Les CD-ROM, DVD-ROM et BD-ROM
- ✓ Architecture 680x0
- ✓ Architecture SPARC
- ✓ Architecture POWER
- ✓ Architecture ARM
- ✓ Architecture Cell
- ✓ Architecture IA-64
- ✓ Architecture MIPS
- ✓ Le Front Site Bus
- ✓ Les bus PCI / PCI-X
- ✓ Les bus ISA / EISA / VLB
- ✓ La loi de Moore et la Loi de Rock
- ✓ Le standard SCSI
- ✓ Le standard ATA (IDE et ATAPI)
- ✓ Le standard SATA
- ✓ Les mémoires statiques (SRAM…)
- ✓ Les mémoires Dynamiques (DRAM…)
- ✓ Les mémoires Vidéos (VRAM…)
- ✓ …

Plan

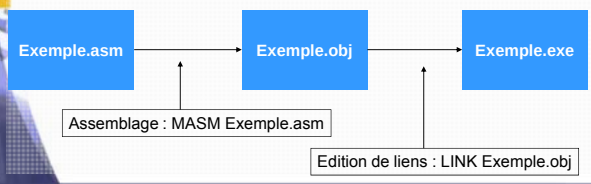
- ✓ Introduction
- ✓ Représentation de l'information
- ✓ Arithmétique binaire
- ✓ Algèbre de Boole
- ✓ Circuits séquentiels/Automates
- ✓ Architecture type Von Neumann
- ✓ **Couche d'assemblage / Assembleur 8086**
- ✓ Mécanismes d'interruptions
- ✓ Fonctions avancées des processeurs modernes
- ✓ Gestion de la mémoire
- ✓ Traduction de programmes, compilation édition de liens, chargement

Définitions



Définitions

- ✓ **Assembleur** : mnémoniques associées au langage machine (JMP,MOV,CMP…)
- ✓ **Langage machine** : ordres (en binaire) compréhensible par un processeur donné.
- ✓ **Exécutable** : suite d'instruction en langage machine



Processeurs 80x86

- ✓ Micro-processeurs 80x86 équipent les PC et compatibles
  - Premiers PC (début 80) = 8086, micro-processeur 16 bits
  - Puis 80286, 80386, 80486, Pentium…
- ✓ Améliorations
  - Augmentation de la fréquence d'horloge, de la largeur des bus d'adresses et de données
  - Ajout de nouvelles instructions et de registres
- ✓ Compatibilité ascendante
  - Un programme écrit dans le langage machine du 286 peut s'exécuter sur un 486 (l'inverse est faux)

Les bus du 8086

- ✓ Les Bus
  - Bus de données : 16 bits
  - Bus d'adresses : 20 bits
- ✓ Exemples

MICROPROCESSEUR	BUS DE DONNEES	BUS D'ADRESSES
8088	8	20
8086	16	20
80286	16	24
80386SX	16	32
80386DX	32	32
80486	32	32
Pentium	64	32

Registres du 8086 (1)

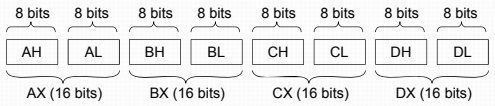
- ✓ Un registre sert à stocker les données nécessaires à l'exécution d'un programme par le microprocesseur.
  - On n'accède pas à un registre avec une adresse, mais avec son appellation.
- ✓ Exemple : l'instruction MOV registre1,registre2
  - elle a pour effet de copier le contenu du registre2 dans le registre1 (Le contenu du registre1 est écrasé).
  - Registre1 et registre2 doivent être de même taille

Registres du 8086 (2)

- Registres de travail :
  - **AX** : Accumulateur utilisé pour les opérations arithmétiques.
  - **BX** : Base pour les adressages
  - **CX** : Compteur pour le comptage
  - **DX** : Data pour les données
- Registres de travail ont une taille de 16 bits

Registres du 8086 (3)

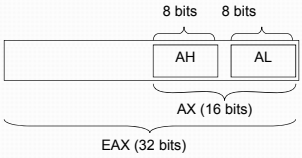
- Les Registres AX, BX, CX et DX (16 bits) peuvent également être utilisés comme 2 registres 8 bits



- AX et AL (AH) sont le même registre
  - Si on écrit dans AL (AH), on efface une partie de AX
  - Il en est de même pour BX et (BL, BH), CX et (CL, CH) et DX et (DL, BDH)

Registres du 8086 (4)

- Depuis les 386 ont dispose également de 4 nouveaux registres de données de 32 bits
  - EAX, EBX, ECX et EDX



Registres du 8086 (5)

- Registres d'offset :
  - **SI** : Source Index utilisé lors d'opérations sur chaînes de caractères, associé à DS.
  - **DI** : Destination Index utilisé lors d'opérations sur les chaînes de caractères, associé à DS. si utilisé comme index et à ES lors de manipulation de chaînes.
  - **IP** : Instruction Pointer associé à CS, il indique la prochaine instruction à exécuter, et ne peut être modifié.
  - **BP** : Base Pointer associé à SS pour accéder aux données de la pile lors d'appels d'un sous-programme.
  - **SP** : Stack Pointer associé à SS, il indique l'adresse du dernier élément de la pile.

Le registre IP

- IP : pointeur d'instruction
- Le registre IP du processeur conserve l'adresse de la prochaine instruction à exécuter
- Le processeur effectue les actions suivantes pour chaque instruction :
  - lire et décoder l'instruction à l'adresse IP;
  - IP ←IP + taille de l'instruction;
  - exécuter l'instruction.

Registres du 8086 (6)

- Registres de segment :
  - **CS** : Code Segment
    - il contient l'adresse du segment de la mémoire de programme
  - **DS** : Data Segment
    - il contient l'adresse du segment de mémoire de données
  - **SS** : Stack Segment
    - il contient l'adresse du segment de mémoire de la pile
  - **ES** : Extra Segment
    - segment supplémentaire pour adresser des données.

Sens des mouvements de données

- ✓ La plupart des instructions spécifient des mouvements de données entre la mémoire principale et le microprocesseur.
- ✓ En langage symbolique, on indique toujours la destination, puis la source.
  - L'instruction **MOV AX, [0110]** transfère le contenu de l'emplacement mémoire 0110H dans l'accumulateur,
  - L'instruction **MOV [0112], AX** transfère le contenu de l'accumulateur dans l'emplacement mémoire 0112.
  - L'instruction MOV s'écrit donc toujours :
    - MOV destination, source

Jeu d'instructions du 8086 (1)

- ✓ Instructions d'affectation :
  - Déclenchent un transfert de données entre l'un des registres du processeur et la mémoire principale (MP)
    - ***Lecture en MP*** : de la Mémoire Principale vers le CPU
    - ***Ecriture en MP***: du CPU vers la Mémoire Principale
  - L'instruction MOV
    - MOV Registre, Registre → Par ex: MOV CX, DX
    - MOV Registre, Mémoire → Par ex: MOV AX, [MEM]
    - MOV Registre, Constante → Par ex: MOV AH, 9
    - MOV Mémoire, Registre → Par ex: MOV [MEM], AX
    - MOV Mémoire, Constante → Par ex: MOV [MEM], 5
    - …

Jeu d'instructions du 8086 (2)

- ✓ Instructions arithmétiques et logiques
  - Opération entre une donnée et l'accumulateur. Le résultat est placé dans le premier paramètre.
  - L'instruction Addition (**ADD**): AX ← AX + donnée
    - ADD Registre, Registre → Par ex: ADD AX, BX
    - ADD Registre, Mémoire → Par ex: ADD BX, [MEM]
    - ADD Registre, Constante → Par ex: ADD AH, 12
    - ADD Mémoire, Registre → Par ex: ADD [MEM], BX
    - ADD Mémoire, Constante → Par ex: ADD [MEM], 11
  - L'instruction Soustraction (**SUB**): AX ← AX + donnée
    - SUB Registre, Registre → Par ex: SUB AX, BX
    - SUB Registre, Mémoire → Par ex: SUB BX, [MEM]
    - SUB Registre, Constante → Par ex: SUB AH, 12
    - SUB Mémoire, Registre → Par ex: SUB [MEM], BX
    - SUB Mémoire, Constante → Par ex: SUB [MEM], 11

Jeu d'instructions du 8086 (3)

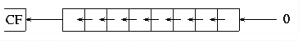
- ✓ Instructions arithmétiques et logiques
  - Opération entre une donnée et l'accumulateur. Le résultat est placé dans le premier paramètre.
  - L'instruction Incrémentation (**INC**): AX ← AX + 1
    - INC Registre → Par ex: INC CX
    - INC Mémoire → Par ex: INC [MEM]
  - L'instruction Décrémentation (**DEC**): AX ← AX – 1
    - DEC Registre → Par ex: DEC CX
    - DEC Mémoire → Par ex: DEC [MEM]
  - L'instruction Négation (**NEG**) : AX ← - AX
    - NEG Registre → Par ex: NEG CX
    - NEG Mémoire → Par ex: NEG [MEM]

Jeu d'instructions du 8086 (4)

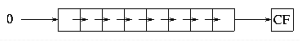
- ✓ Instructions arithmétiques et logiques
  - Les instructions de décalage et de rotations (Sxx / Rxx)
    - Sxx Registre, 1 → Par ex: Sxx DL,1
    - Sxx Registre, CL → Par ex: Sxx AX, CL
    - Sxx Mémoire, 1 → Par ex: Sxx [MEM], 1
    - Sxx Mémoire, CL → Par ex: Sxx [MEM], CL
  - Décalage vers la gauche ou vers la droite les bits de l'accumulateur. Opérations utilisées
    - pour décoder bit à bit des données
    - pour diviser ou multiplier rapidement par une puissance de 2.
    - décaler AX de n bits vers la gauche revient multiplier AX par 2^n (un décalage vers la droite revient à diviser par 2^n)

Décalage et Rotations (1)

- ✓ **SHL *registre*, 1** *(Shift Left)*
  - Décale les bits du registre d'une position vers la gauche. Le bit de gauche est transféré dans l'indicateur CF. Les bits introduits à droite sont à zéro.



- ✓ **SHR *registre*, 1** *(Shift Right)*
  - Comme SHL mais vers la droite. Le bit de droite est transféré dans CF. SHL et SHR peuvent être utilisé pour multiplier/diviser des entiers *naturels* (et non des relatifs car le bit de signe est perdu)



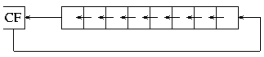
Décalage et Rotations (2)

- ROL *registre*, 1 *(Rotate Left)*
  - Rotation vers la gauche : le bit de poids fort passe à droite, et est aussi copié dans CF. Les autres bits sont décalés d'une position.

- ROR *registr*
  - Rotation vers la droite : le bit de poids faible passe à gauche, et est aussi copié dans CF. Les autres bits sont décalés d'une position

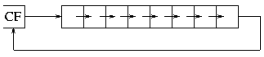
Décalage et Rotations (3)

- RCL *registre*, 1 *(Rotate Carry Left)*
  - Rotation vers la gauche en passant par l'indicateur CF. CF prend la place du bit de poids faible; le bit de poids fort part dans CF.



- RCR *registre*, 1 *(Rotate Carry Right)*
  - Rotation vers la droite en passant par l'indicateur CF. CF prend la place du bit de poids fort; le bit de poids faible part dans CF

- RCL et RCI t le contenu d'un registre.



Jeu d'instructions du 8086 (5)

- Instructions arithmétiques et logiques
  - Les instructions logiques (AND, OR, XOR et TEST)
    - AND Registre, Registre → Par ex: AND AX, DX
    - AND Registre, Mémoire → Par ex: AND BX, [MEM]
    - AND Registre, Constante → Par ex: AND AH, 7
    - AND Mémoire, Registre → Par ex: AND [MEM], AX
    - AND Mémoire, Constante → Par ex: AND [MEM], 11
  - Pas de propagation de retenue lors de ces opérations
    - chaque bit du résultat est calculé indépendamment des autres
    - AX <- AX ou BX
  - Les instructions logiques (NOT)
    - AND Registre → Par ex: NOT CX
    - AND Mémoire → Par ex: NOT [MEM]

Opérations logiques

- AND
  - AND est souvent utilisé pour forcer certains bits à 0.
  - Après ***AND AX, FF00***, l'octet de poids faible de AX vaut 00, tandis que l'octet de poids fort est inchangé.
- OR
  - OR est souvent utilisé pour forcer certains bits à 1.
  - Après ***OR AX, FF00***, l'octet de poids fort de AX vaut FF, tandis que l'octet de poids faible est inchangé.
- XOR
  - XOR est souvent utilisé pour inverser certains bits.
  - Après ***XOR AX, FFFF***, tous les bits de AX sont inversés

Jeu d'instructions du 8086 (6)

- Instructions de comparaison
  - Compare un registre avec une donnée et positionne les indicateurs. La donnée et la valeur du registre ne sont pas modifiées
  - L'instruction Comparaison (**CMP**):
    - CMP Registre, Registre → Par ex: CMP CX, DX
    - CMP Registre, Mémoire → Par ex: CMP AX, [MEM]
    - CMP Registre, Constante → Par ex: CMP AH, 12
    - CMP Mémoire, Registre → Par ex: CMP [MEM], AX
    - CMP Mémoire, Constante → Par ex: CMP [MEM], 11

Jeu d'instructions du 8086 (7)

- Instructions de branchement
  - Un programme exécute normalement les instructions les une à la suite des autres. En assembleur, le registre IP indique l'adresse de la prochaine instruction à modifier
  - Une instruction de branchement modifie la valeur de IP pour exécuter une autre instruction (cas des boucles, tests, …)
  - Il y a 2 catégories de branchement
    - ***Branchements inconditionnels*** : on effectue toujours le saut
    - ***Branchements conditionnels*** : on effectue le saut si une condition est satisfaite

Jeu d'instructions du 8086 (8)

- ✓ Instructions de branchement inconditionnels (JMP):
  - JMP Mémoire → Par ex: JMP [MEM]
  - JMP Offset → Par ex: JMP ADRESSE
- ✓ Instructions de branchement conditionnels (Jxxx):
  - Jxxx label → Par ex: JMP fin
  - Attention, l'espace franchi par les instructions de branchement conditionnels ne doit pas dépasser 127 octets

Jeu d'instructions du 8086 (9)

- ✓ Instructions de branchement conditionnels (Jxxx):
  - **JE** *Jump if Equal (ou JZ)* : saut si ZF = 1;
  - **JNE** *Jump if Not Equal (ou JNZ)* : saut si ZF = 0;
  - **JG** *Jump if Greater* : saut si ZF = 0 et SF = 0;
  - **JLE** *Jump if Lower or Equal* : saut si ZF=1 ou SF ≠ OF;
  - **JS** *Jump if Sign* : saut si SF=1;
  - **JNS** *Jump if Not Sign* : saut si SF=0;
  - **JA** *Jump if Above* : saut si CF=0 et ZF=0;
  - **JBE** *Jump if Below or Equal :* saut si CF=1 ou ZF=1.
  - **JB** *Jump if Below :* saut si CF=1.
  - **JO** *Jump if Overflow :* saut si OF=1

Les Indicateurs (1)

- ✓ Les instructions de branchement conditionnels utilisent les indicateurs, qui sont des bits spéciaux positionnés par l'UAL après certaines opérations.
- ✓ Les indicateurs sont regroupés dans le registre d'état du processeur. Ce registre n'est pas accessible globalement par des instructions ; chaque indicateur est manipulé individuellement par des instructions spécifiques.
- ✓ On se concentrera ici les indicateurs suivant
  - ZF, CF, SF et OF.

Les Indicateurs (2)

- ✓ ZF Zero Flag
  - Cet indicateur est mis à 1 lorsque le résultat de la dernière opération est zéro.
  - Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), ZF=1 indiqueque les deux opérandes étaient égaux. Sinon, ZF est positionné à 0.

Les Indicateurs (3)

- ✓ CF Carry Flag
  - C'est l'indicateur de report (retenue), qui intervient dans les opérations d'addition et de soustractions sur des entiers naturels. Il est positionné en particulier par les instructions ADD, SUB et CMP.
  - CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes.
  - Exemples (sur 4 bits pour simplifier) :

0100 + 0110 = 1010	1100 + 0110 = 0010	1111 + 0001 = 0000
CF = 0	CF = 1	CF = 1

Les Indicateurs (4)

- ✓ SF Sign Flag
  - SF est positionné à 1 si le bit de poids fort du résultat d'une addition ou soustraction est 1 ; sinon SF=0.
  - SF est utile lorsque l'on manipule des entiers relatifs, car le bit de poids fort donne alors le signe du résultat.
  - Exemples (sur 4 bits) :

0100 + 0110 = 1010	1100 + 0110 = 0010	1111 + 0001 = 0000
SF = 1	SF = 0	SF = 0

Les Indicateurs (5)

- ✓ OF Overflow Flag
  - – Indicateur de débordement OF=1 si le résultat d'une addition ou soustraction donne un nombre qui n'est pas codable en relatif dans l'accumulateur
    - ▪ Par exemple si l'addition de 2 nombres positifs donne un codage négatif
  - – Overflow = Carry XOR Retenue$_{n-2 \rightarrow n-1}$

0100	1100	1111
+ 0110	+ 0110	+ 0001
1010	0010	0000
OF = 1	OF = 0	OF = 0

Jeu d'instructions du 8086 (10)

- ✓ Instructions de modification des indicateurs
  - – L'instruction CLC (sans opérande)
    - ▪ Elle met à 0 l'indicateur CF
  - – L'instruction SLC (sans opérande)
    - ▪ Elle met à 1 l'indicateur CF
  - – L'instruction CMC (sans opérande)
    - ▪ Elle inverse l'indicateur CF (opération NOT)

Correspondance avec le langage C

- ✓ La table suivante établi un parallèle entre les instructions arithmétiques et logiques du 8086 et les opérateurs du langage C (lorsque ces derniers agissent sur des variables non signées).

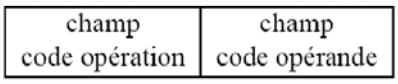
Opérateur C	Instruction 80x86
+	ADD
-	SUB
<<	SHL
>>	SHR
|	OR
&	AND
^	XOR

Codage des instructions (1)

- ✓ Les instructions et leurs opérandes (paramètres) sont stockés en mémoire principale.
- ✓ La taille totale d'une instruction (nombre de bits nécessaires pour la représenter en mémoire) dépend du type d'instruction et aussi du type d'opérande.
- ✓ Chaque instruction est toujours codée sur un nombre entier d'octets, afin de faciliter son décodage par le processeur.

Codage des instructions (2)

- ✓ Une instruction est composée de deux champs :
  - – **le code opération (Obligatoire)**, qui indique au processeur quelle instruction réaliser ;
  - – **le champ opérande** qui contient la donnée, ou la référence à une donnée en mémoire (son adresse).



- ✓ Selon la manière dont la donnée est spécifiée, c'est à dire selon le mode d'adressage de la donnée, une instruction sera codée par 1, 2, 3 ou 4 octets.

Code opération : assembleur

Symbole	Code Op.	Octets	
MOV AX, *valeur*	B8	3	AX ← *valeur*
MOV AX, [*adr*]	A1	3	AX ← contenu de l'adresse *adr.*
MOV [*adr*], *AX*	A3	3	range AX à l'adresse *adr.*
ADD AX, *valeur*	05	3	AX ← AX + *valeur*
ADD AX, [*adr*]	03 06	4	AX ← AX + contenu de *adr.*
SUB AX, *valeur*	2D	3	AX ← AX - *valeur*
SUB AX, [*adr*]	2B 06	4	AX ← AX - contenu de *adr.*
SHR AX, 1	D1 E8	2	décale AX à droite.

On utilise des programmes spéciaux, appelés *assembleurs*, pour traduire automatiquement le langage symbolique en code machine.

Exemple de programme

- Programme en langage machine implanté à l'adresse mémoire 0100H
  - A1 10 01 03 06 12 01 A3 14 01
  - Ce programme additionne le contenu de deux cases mémoire et range le résultat dans une troisième

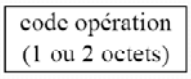
Adresse	Contenu MP	Langage Symbolique	Explication
0100H	A1 10 01	MOV AX, [0110H]	;Charger AX avec le contenu de 0110H.
0103H	03 06 12 01	ADD AX, [0112H]	;Ajouter le contenu de 0112H à AX (résultat dans AX).
0107H	A3 14 01	MOV [0114H],AX	;Ranger AX en 0114H.

Modes d'adressage

- **Adressage** = méthode de localisation des opérandes
- **Mode d'adressage** = manière d'interpréter les bits d'un champs d'adresse en vue de la localisation de l'opérande

Adressage implicite (par registre)

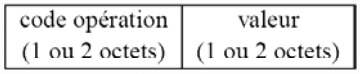
- Pas d'accès mémoire pour les opérandes
- L'instruction contient seulement le code opération, sur 1 ou 2 octets.



- L'instruction porte sur des registres ou spécifie une opération sans opérande
  - Exemple : `INC AX`

Adressage immédiat

- Adressage dans lequel la valeur de l'opérande figure directement dans l'instruction sans avoir besoin de faire un nouvel accès mémoire.

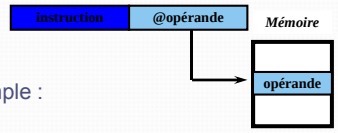


- Exemple :

```
MOV AL, 10 ; Met la valeur 10 dans AL
ADD AX, 5 ; Ajoute 5 à la valeur contenue dans AX
```


Adressage direct

- Un des opérandes est l'emplacement mémoire dont l'adresse figure dans l'instruction.
  - Une fois l'instruction lue, il faut aller faire un accès mémoire pour obtenir l'opérande désiré.

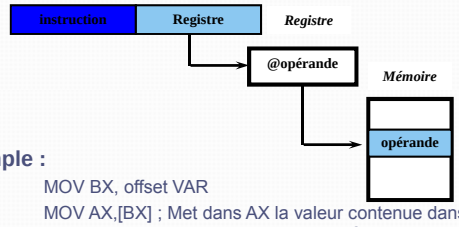


- Exemple :

`MOV AX,[130h];` Met la valeur contenue dans l'emplacement d'adresse 130h dans AX

Adressage indirect

- L'adresse de la variable est contenue dans un registre de base ou d'index : 2 accès mémoires.



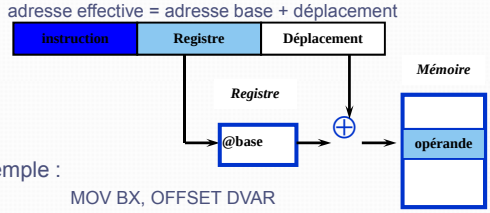
- **Exemple :**

```
MOV BX, offset VAR
MOV AX,[BX] ; Met dans AX la valeur contenue dans
l'emplacement mémoire dont l'adresse est spécifiée dans
l'emplacement mémoire d'adresse BX.
```

  - Le type de la donnée doit être accordé avec le registre utilisé

Adressage basé

- ✓ Similaire à l'adressage indirect par registre sauf qu'un déplacement est ajouté à la base.
  - adresse effective = adresse base + déplacement

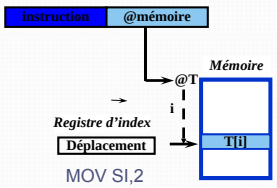


- ✓ Exemple :

```
MOV BX, OFFSET DVAR
MOV AX, [BX+4]
ADD AX, [BP+4]
```


Adressage indexé

- ✓ On utilise un registre d'index SI ou DI plus un déplacement
  - L'adressage est noté par des crochets
  - Utile pour le parcours de tableaux



- ✓ Exemple :

```
MOV SI,2
MOV AX, T[SI]
```


Modes d'adressage : résumé

- ✓ Par registre ou implicite
  - `Add AX,BX`
- ✓ Immédiat
  - `Add AX,valeur`
- ✓ Direct
  - `Add AX,[adresse]`
- ✓ Indirect
  - `Mov BX,adresse`
  - `Add AX,[BX]`
- ✓ Indexé
  - `Add,[adresse+index]`

Temps d'exécution

- ✓ Chaque instruction nécessite un certain nombre de cycles d'horloges pour s'effectuer.
- ✓ Le nombre de cycles dépend de la complexité de l'instruction et aussi du mode d'adressage
  - il est plus long d'accéder à la mémoire principale qu'à un registre du processeur.
- ✓ La durée d'un cycle dépend bien sur de la fréquence d'horloge de l'ordinateur.
  - Plus l'horloge bat rapidement, plus un cycle est court et plus on exécute un grand nombre d'instructions par seconde.

Structure d'un programme (1)

```
data            SEGMENT     ; data est le nom du segment de donnees

  ; directives de declaration de donnees

data            ENDS        ; fin du segment de donnees

                ASSUME DS:data, CS:code

code            SEGMENT     ; code est le nom du segment d'instructions

debut:                      ; 1ere instruction, avec l'etiquette debut

  ; suite d'instructions

code            ENDS

                END debut   ; fin du programme, avec l'etiquette
                            ; de la premiere instruction.
```


Structure d'un programme (2)

- ✓ Un programme écrit en assembleur comprend des définitions de données et des instructions, qui s'écrivent chacune sur une ligne de texte.
- ✓ Les données sont déclarées par des directives, mots clef spéciaux que comprend l'assembleur.
- ✓ Les directives qui déclarent des données sont regroupées dans le segment de données, qui est délimité par les directives SEGMENT et ENDS.
- ✓ Les instructions sont placées dans un autre segment, le segment de code.

Structure d'un programme (3)

- ✓ La première instruction du programme (dans le segment d'instruction) doit toujours être repérée par une étiquette.
- ✓ Le fichier doit se terminer par la directive END avec le nom de l'étiquette de la première instruction
  - ceci permet d'indiquer à l'éditeur de liens quelle est la première instruction à exécuter lorsque l'on lance le programme.
- ✓ Les points-virgules indiquent des commentaires.

Déclaration de variables (1)

- ✓ On déclare les variables à l'aide de directives.
- ✓ L'assembleur attribue a chaque variable une adresse.
  - Dans le programme, on repère les variables grâce à leur nom.
- ✓ Les noms des variables (comme les étiquettes) sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre.
  - Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les 3 caractères @ ? _
- ✓ Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

Déclaration de variables (2)

- ✓ 2 directives permettent de déclarer des variables
  - DB (Define Byte) : 1 octet
  - DW (Define Word) : 2 octets
  - Les valeurs initiales peuvent être données en hexadécimal (terminée par H) ou en binaire (terminée par b)
- ✓ Exemple d'utilisation :

```
data SEGMENT
  entree DW 15 ; 2 octets initialises a 15
  sortie DW ? ; 2 octets non initialises
  cle DB ? ; 1 octet non initialise
  nega DB -1 ; 1 octet initialise a -1
  truc DW 0F0AH ; en hexa
  masque DB 01110000b ; en binaire
data ENDS
```


Déclaration de variables (3)

- ✓ Tableaux
  - Il est aussi possible de déclarer des tableaux, c'est à dire des suite d'octets ou de mots consécutifs.
- ✓ Pour cela, utiliser plusieurs valeurs initiales :

```
data SEGMENT
  machin db 10, 0FH      ; 2 fois 1 octet
  chose db -2, 'ALORS'
data ENDS
```

  - la déclaration de la variable chose :
    - un octet à -2 (=FEH),
    - une suite de caractères.
  - L'assembleur n'impose aucune convention pour la représentation des chaînes de caractères

Déclaration de variables (4)

- ✓ Tableaux
  - Après chargement de ce programme, la mémoire aura le contenu suivant :

0AH	← machin
OFH	← machin + 1
FEH	← chose
41H	← chose + 1
4CH	← chose + 2
4FH	← chose + 3
52H	← chose + 4
53H	← chose + 5

  - Pour changer 'ALORS' en 'OLARS', on peut écrire :

```
MOV chose + 1, 'O'
MOV machin + 5, 'A'
```

  - chose+1 est une constante (valeur connue au moment de l'assemblage) : l'instruction générée par l'assembleur pour MOV chose+1, AL est MOV [adr], AL .

Déclaration de variables (5)

- ✓ Tableaux et la directive dup
  - on utilise la directive dup lorsque l'on veut déclarer un tableau de n cases soit
    - Non initialisées
    - Toutes initialisées à la même valeur,
  - Exemples

```
tab DB 100 dup (15) ; 100 octets valant 15
zzz DW 10 dup (?) ; 10 mots de 16 bits non initialises
```

- ✓ Constantes
  - La directive EQU (ou =) permet de déclarer une constante

```
Const1 EQU "Avec du Texte"
Const2 = 1000
```

Segmentation de la mémoire (1)

- ✓ Largeur du bus d'adresse est de 20 bits
  - Possibilité d'adressage mémoire 2^{20} = 1 Mo
- ✓ Le pointeur d'instruction (IP) fait 16 bits
  - Possibilité d'adresser 2^{16} = 64 Ko
  - Cela ne couvre pas la mémoire

- ✓ On utilise deux registres pour indiquer une adresse au processeur

Registre de Segment	Registre de Déplacement

- ✓ Un segment de mémoire est une zone mémoire adressable avec une valeur fixée du registre de segment
  - Un segment a donc une taille maximale de 64 Ko.

Segmentation de la mémoire (2)

- ✓ Chaque segment débute à l'endroit spécifié par un registre spécial (registre segment)
- ✓ Le déplacement permet de trouver une information à l'intérieur du segment.
- ✓ Exemples
  - lecture du code d'une instruction : CS:IP
    - CS registre segment
    - IP déplacement
  - accès aux données : DS
    - MOV AX,[1045] = lecture du mot mémoire d'adresse DS:1045H

Segmentation de la mémoire (3)

- ✓ Registres de déplacement sert à sélectionner une information dans un segment.
  - Dans le segment de code défini par CS
    - Le compteur de programme IP joue ce rôle.
    - CS:IP permet d'accéder à une information dans le segment de code.
  - Dans les segments de donné défini par DS
    - Les deux index SI ou DI jouent ce rôle.
    - Le déplacement peut être aussi une constante.
    - DS:SI ou DS:DI permettent d'accéder à une information dans le segment de données.
  - Dans le segment de pile défini par SS
    - Les registres SP (stack pointer) et BP (base pointer) jouent ce rôle.
    - SS:SP ou SS:BP permettent d'accéder à une information dans le segment de pile.

Segmentation de la mémoire (4)

- ✓ Initialisation des registres segment
  - Dans ce cours, nous n'écrirons pas de programmes utilisant plus de 64 Ko de code et 64 Ko de données, ce qui nous permettra de n'utiliser qu'un seul segment de chaque type.
    - La valeur des registres CS et de DS sera fixée une fois pour toute au début du programme.
  - Le programmeur en assembleur doit se charger de l'initialisation de DS, c'est-à-dire de lui affecter l'adresse du segment de données à utiliser.
  - Par contre, le registre CS sera automatiquement initialisé sur le segment contenant la première instruction au moment du chargement en mémoire du programme

Segmentation de la mémoire (5)

- ✓ La directive ASSUME permet d'indiquer à l'assembleur quel est le segment de données et celui de code, afin qu'il génère des adresses correctes.
- ✓ Le programme doit commencer, avant toute référence au segment de données, par initialiser le registre segment DS, de la façon suivante :

  MOV AX, nom_segment_de_donnees
  MOV DS, AX

- ✓ Il serait plus simple de faire **MOV DS, nom_segment_de_donnees** mais cette instruction n'existe pas.

Structure du programme (1)

```
PILE SEGMENT STACK
DW 256 DUP(?)
Base EQU 0
PILE ENDS

DATA SEGMENT

DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE

main:
MOV AX, DATA
MOV DS,AX

MOV AX,PILE
MOV SS,AX
MOV SP,Base

CODE ENDS
END main
```

Structure du programme (2)

```
PILE SEGMENT STACK
DW 256 DUP(?)
Base:
PILE ENDS

DATA SEGMENT

DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE

main:
MOV AX, DATA
MOV DS,AX

MOV AX,PILE
MOV SS,AX
MOV SP,Base

CODE ENDS
END main
```

Déclaration du segment de pile

Structure du programme (3)

```
PILE SEGMENT STACK
DW 256 DUP(?)
Base:
PILE ENDS

DATA SEGMENT

DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE

main:
MOV AX, DATA
MOV DS,AX

MOV AX,PILE
MOV SS,AX
MOV SP,Base

CODE ENDS
END main
```

Déclaration du segment de données

Structure du programme (4)

```
PILE SEGMENT STACK
DW 256 DUP(?)
Base:
PILE ENDS

DATA SEGMENT

DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE

main:
MOV AX, DATA
MOV DS,AX

MOV AX,PILE
MOV SS,AX
MOV SP,Base

CODE ENDS
END main
```

Déclaration du segment de code
Avec initialisation des registres de segment

Structure du programme (5)

```
PILE SEGMENT STACK
DW 256 DUP(?)
Base:
PILE ENDS

DATA SEGMENT

DATA ENDS
CODE SEGMENT
ASSUME CS:CODE, DS:DATA, SS:PILE

main:
MOV AX, DATA
MOV DS,AX

MOV AX,PILE
MOV SS,AX
MOV SP,Base

CODE ENDS
END main
```

Etiquette du programme principal

Indication de l'étiquette où commence le programme principal

La pile (1)

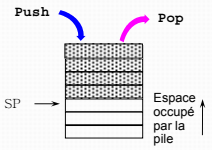
- ✓ Les piles offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs.
- ✓ Une pile est une zone de mémoire particulière définie dans son propre segment dont le registre est SS.
- ✓ Un pointeur (SP) qui conserve l'adresse du sommet de la pile.
- ✓ Elle mémorise les adresses d'appel et retour de sous programmes (cf la partie sur les procédures)

La pile (2)

- ✓ Structure de « rangement » de données
  - zone mémoire spécialisée
  - fonctionnement LIFO (Last In, First Out)
- ✓ 2 instructions pour accéder à la pile

La pile (3)

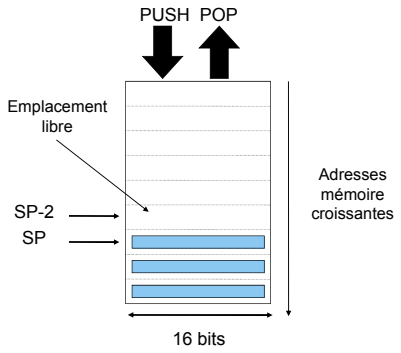
- ✓ Le registre SS (Stack Segment)
  - – C'est le registre segment qui contient l'adresse du segment de pile courant
  - – Il est initialisé au début du programme et reste fixé par la suite.
- ✓ Le registre SP (Stack Pointer) pointe sur le dernier bloc occupé de la pile



La pile (4)

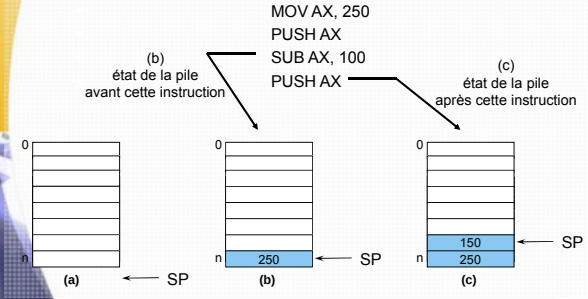
- ✓ PUSH registre
  - – Empile le contenu du registre sur la pile.
  - – PUSH : SP ← SP- 2
- ✓ POP registre
  - – Retire la valeur en haut de la pile et la place dans le registres spécifié.
  - – POP : SP ← SP+ 2
- ✓ Exemple : transfert de AX vers BX en passant par la pile.
  - – PUSH AX ; empile le contenu de AX sur la pile
  - – POP BX ; dépile le 1er élément dans la pile dans BX

La pile (5)



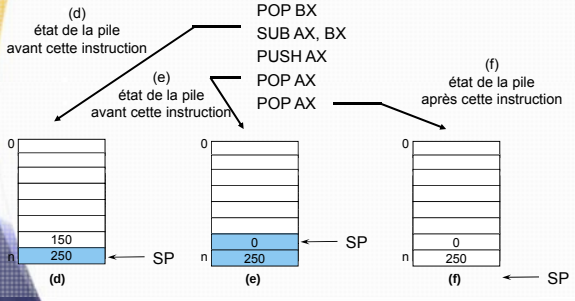
La pile (6)

- ✓ Illustration du fonctionnement de la pile
  - – Quand la pile est vide (a), SP pointe sous la pile



La pile (7)

- ✓ Illustration du fonctionnement de la pile
  - – Quand la pile est vide (a), SP pointe sous la pile



La pile (8)

- ✓ Déclaration d'une pile
  - – Pour utiliser une pile en assembleur, il faut déclarer un segment de pile, et y réserver un espace suffisant. Ensuite, il est nécessaire d'initialiser les registres SS et SP pour pointer sous le sommet de la pile.
  - – Exemple : déclaration d'une pile de 512 octets :

```
seg_pile SEGMENT stack ; mot clef stack car pile
  DW 256 dup (?)
  Base_Pile:    ; étiquette base de la pile
seg_pile ENDS
```

  - – Le mot clef "stack" après la directive SEGMENT, qui indique à l'assembleur qu'il s'agit d'un segment de pile.

La pile (9)

- ✓ Initialisation de la pile
  - Après les déclarations précédentes, on utilisera la séquence d'initialisation :

```
ASSUME SS:seg_pile
MOV AX, seg_pile
MOV SS, AX ; init Stack Segment
MOV SP, Base_Pile      ; pile vide
```

  - Le registre SS s'initialise de façon similaire au registre DS ; par contre, on peut accéder directement au registre SP.
  - Afin d'initialiser SP, il faut repérer l'adresse du bas de la pile ; c'est le rôle de l'étiquette **Base_Pile**

Appels systèmes DOS (1)

- ✓ SAISIR UN CARACTERE
  - Cette fonction permet de lire un caractère sur l'entrée standard. Ce caractère est récupéré sous la forme de son code ASCII dans AL.
  - Entrée
    - AH = 01h
  - Sortie
    - AL = caractère entré au clavier
  - Exemple

```
MOV AH,01
INT 21H ; résultat dans AL sous forme de car ASCII
```


Appels systèmes DOS (2)

- ✓ AFFICHER UN CARACTERE
  - Cette fonction permet de sortir (afficher) un caractère sur la sortie standard. Ce caractère doit être stocké sous la forme de son code ASCII dans DL.
  - Entrée
    - AH = 02h
    - DL = Code ASCII du caractère à afficher
  - Sortie
    - Aucune
  - Exemple

```
MOV DL,'A'
MOV AH,02
INT 21H
```


Appels systèmes DOS (3)

- ✓ AFFICHER UNE CHAINE DE CARACTERES
  - Cette fonction permet de sortir (afficher) une chaîne de caractères sur la sortie standard. Cette chaîne doit être stockée sous la forme d'une séquence d'octets correspondant aux codes ASCII des caractères composant la chaîne. La fin de la chaîne de caractères doit être signalée à l'aide du caractère '$'.
  - Entrée
    - AH = 09h
    - DS = Adresse de segment de la chaîne de caractères
    - DX = Adresse d'offset de la chaîne de caractères
  - Sortie
    - Aucune
  - *Note*: si vous insérez a la suite dans la chaîne les deux caractères ASCII n° 10 et 13; vous obtiendrez un retour à la ligne.

Les Boucles (1)

- ✓ **IF THEN ELSE**

```
Si (ax==1)
   bx = 10;
Sinon {
    bx = 0;
    cx  =10;
}
```

- ✓ **En assembleur, on obtient**

```
If:     CMP AX, 1
        JNE Else
Then:   MOV BX,10
        JMP EndIf
Else:   MOV BX,0
        MOV CX,10
EndIf:
```


Les Boucles (2)

- ✓ **La boucle FOR**

```
bx = 0 ;
Pour K = 0 jusqu'à 10
    bx  = bx + K ;
```

- ✓ **En assembleur, on obtient**

```
        MOV BX, 0
        MOV CX,0
For:    CMP CX,10
        JG EndFor
        ADD BX, CX
        INC CX
        JMP For
EndFor:
```

Les Boucles (3)

✓ **La boucle WHILE**

```
bx = 5 ;
Tant que bx > 0
    faire bx = bx-1;
```

✓ **En assembleur, on obtient**

```
            MOV BX,5
While :     CMP BX,0
            JLE EndWhile
            DEC BX
            JMP While
EndWhile:
```


Les Boucles (4)

✓ **La boucle REPEAT**

```
bx =  10 ;
Répéter
    bx = bx – 1 ;
jusqu'à bx <= 0
```

✓ **En assembleur, on obtient**

```
            MOV BX,10
Repeat1:    DEC BX
            CMP BX,0
            JG Repeat1
Endrepeat1:
```


Les Boucles (5)

✓ **La boucle REPEAT**

```
bx =  10 ;
Répéter
    bx = bx – 1 ;
jusqu'à bx <= 0
```

✓ **En assembleur, on obtient**

```
            MOV BX, 0
            MOV CX, 10
Repeat2:    ADD BX,CX
            LOOP Repeat2;
EndRepeat2:
```

- Si le nombre de répétitions est connu et au moins égal à 1
- Ici on utilise BX et CX pour remplacer la variable bx. Donc on doit faire ADD BX, CX

Les Boucles (6)

✓ **Le test SWITCH**

```
Switch (bx) {
  case 1: ax =1 ; break;
  case  2: ax = 5 ; break ;
  case  3: ax  = 10; break;
  default: ax = 0;
}
```

✓ **En assembleur, on obtient**

```
            CMP BX,1
            JNZ case2
            MOV AX,1
            JMP endswitch
case2:      CMP BX,2
            JNZ case3
            MOV AX,5
            JMP endswitch
case 3:     CMP BX, 3
            JNZ default
            MOV AX,10
            JMP endswitch
default:    MOV AX,0
endswitch:
```


Spécification de la taille des données

✓ Dans certains cas, l'adressage indirect est ambigu. Par exemple, si l'on écrit l'exemple si dessous, l'assembleur ne sait pas si l'instruction concerne 1 ou 2 octets consécutifs.

MOV [BX], 0 ; range 0 a l'adresse spécifiée par BX

✓ Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

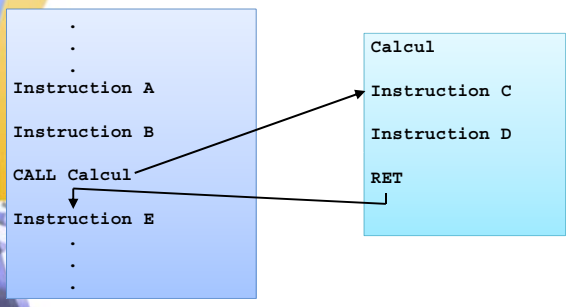
MOV byte ptr [BX], val ; concerne 1 octet
MOV word ptr [BX], val ; concerne 1 mot de 2 octets

Procédures (1)

✓ Notion de procédure équivalente
- à la notion de fonction en C
- à la notion de sous programme dans d'autres langages

✓ Une procédure
- est une suite d'instruction effectuant une action précise
  - Regroupées par commodité
  - Permet d'éviter de les écrire plusieurs fois
- est repérée par l'adresse de sa première instruction
- peut être appelé par
  - un programme
  - une autre procédure

Procédures (2)



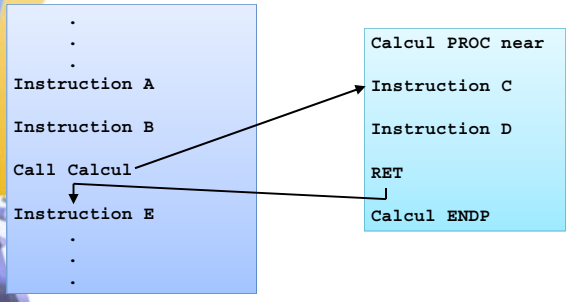
Procédures (3)

- ✓ Appel d'une procédure est effectuée par
  CALL adresse_debut_procedure ;
- ✓ La fin d'une procedure est marqué par
  RET
- ✓ Déclaration d'une procédure :
  Procedure PROC [NEAR/FAR]
  …
  Procedure ENDP
- ✓ La procédure et programme principal sont
  - dans le même segment de code : NEAR
  - dans 2 segments différents : FAR

Procédures (4)

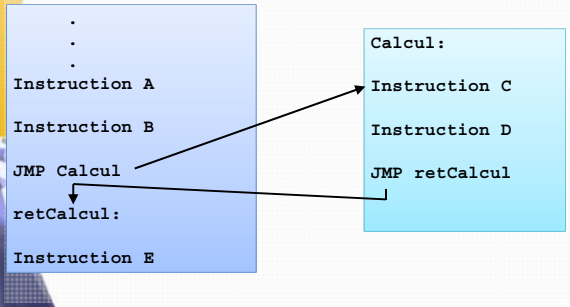
- ✓ Au moment de l'appel de la fonction, l'adresse de l'instruction suivante est sauvegardée dans la pile
  - CALL = sauvegarde de IP
- ✓ A la fin de la procédure, l'unité de traitement récupère les valeurs sauvegardées pour retourner au programme principal
  - RET = dépilement de IP
- ✓ Si la procédure est de type FAR, CS est aussi sauvegardé sur la pile lors du CALL, RET dépile IP puis CS

Procédures (5)



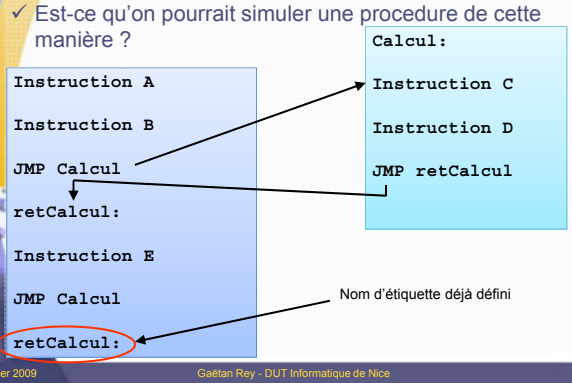
Exercice (1)

- ✓ Est-ce qu'on pourrait simuler une procedure de cette manière ?



Réponse (1)

- ✓ Est-ce qu'on pourrait simuler une procedure de cette manière ?



Passage de paramètres

- ✓ En général, une procédure
  - effectue un traitement sur des données (*paramètres*) qui sont fournies par le programme appelant, et
  - produit un résultat qui est transmis à ce programme.
- ✓ 2 méthodes de passage de paramètres :
  - par registre
  - par la pile

Passage par registre (1)

- ✓ Passage de paramètres par registre
  - les paramètres d'entrée de la procédure sont mis dans des registres avant l'appel de la procédure
  - les paramètres de sortie sont aussi rangés dans des registres
- ✓ Avantages
  - rapidité,
  - simplicité de gestion
- ✓ Inconvénient
  - peu de registres

Passage par registre (2)

```
      .
      .
      .
MOV AX, var1

MOV BX, var2

Call Min

MOV varMin, AX
      .
      .
      .
```

```
Min PROC near

  CMP AX,BX
  JLE axMin
  MOV AX,BX
axMin:

RET

Min ENDP
```


Passage par la pile (1)

- ✓ Passage de paramètres par la pile
  - Les paramètres d'entrée sont empilés avant l'appel de la procédure
  - Les paramètres de sortie sont dépilés par le programme principal
- ✓ Avantage
  - pas de limite au nombre de paramètres
- ✓ Inconvénient
  - récupération des paramètres plus « lourde »

Passage par la pile (2)

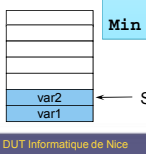
```
      .
      .
      .
PUSH var1

PUSH var2

Call Min

POP varMin
      .
      .
      .
```

```
Min PROC near
  MOV BP, SP
  MOV AX, [BP+4]
  MOV BX, [BP+2]
  CMP AX,BX
  JLE axMin
  MOV AX,BX
axMin:
  MOV [BP+4], AX
  RET 2

Min ENDP
```

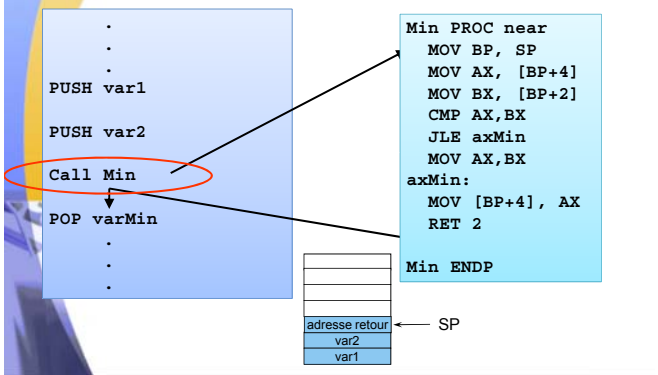

Passage par la pile (3)

```
      .
      .
      .
PUSH var1

PUSH var2

Call Min

POP varMin
      .
      .
      .
```

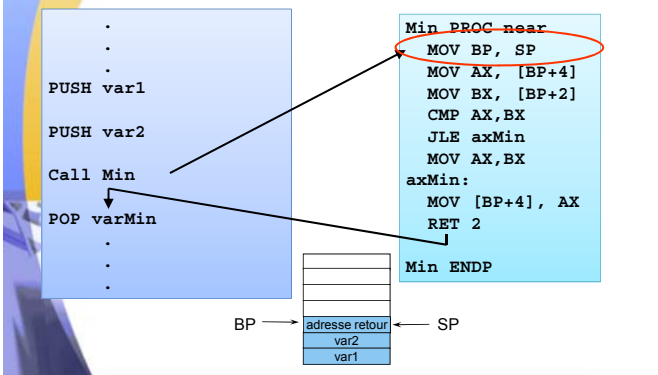
```
Min PROC near
  MOV BP, SP
  MOV AX, [BP+4]
  MOV BX, [BP+2]
  CMP AX,BX
  JLE axMin
  MOV AX,BX
axMin:
  MOV [BP+4], AX
  RET 2

Min ENDP
```



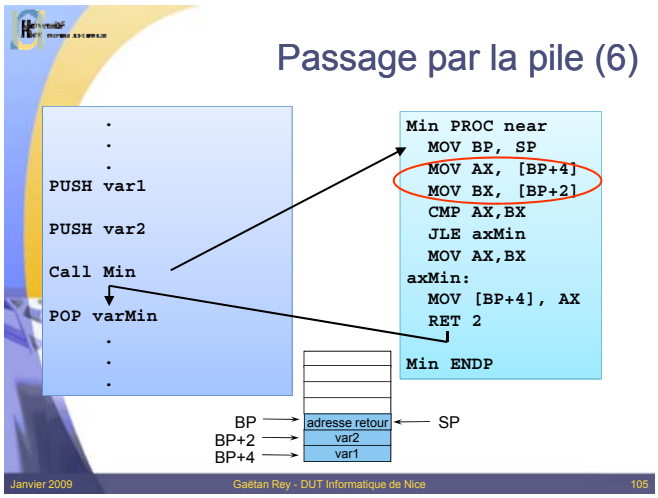
Passage par la pile (4)



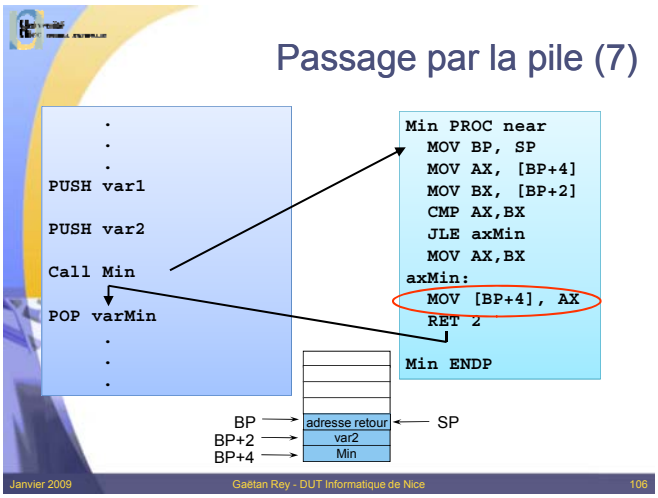
Passage par la pile (5)



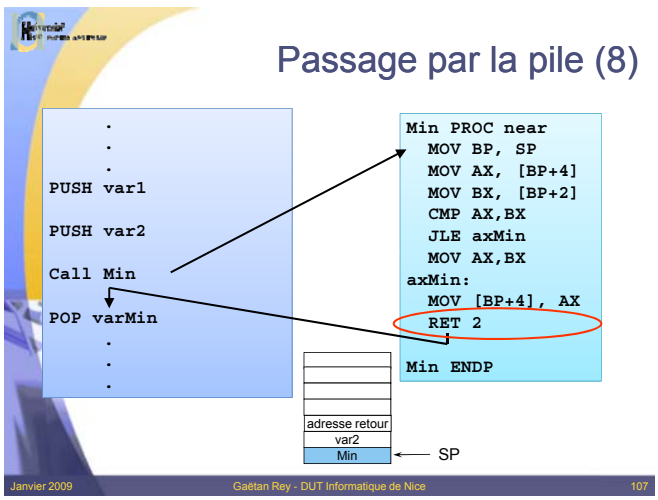
Passage par la pile (6)



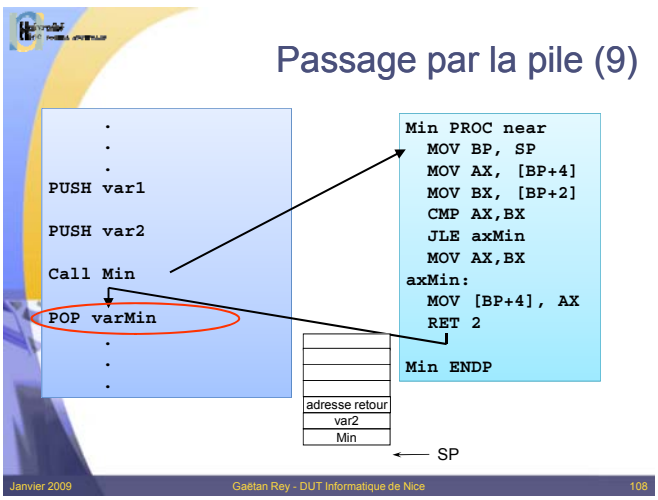
Passage par la pile (7)



Passage par la pile (8)



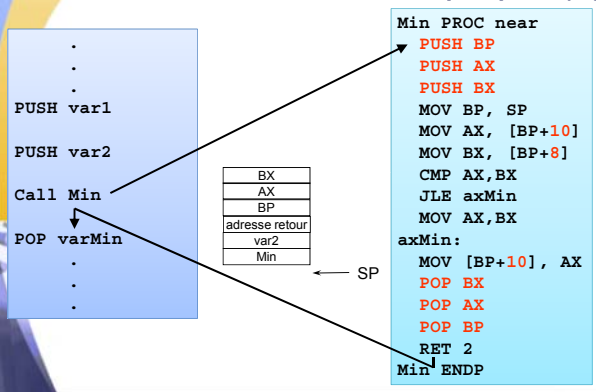
Passage par la pile (9)



Procédure propre (1)

- ✓ La procédure ne doit pas modifier les valeurs des registres du point de vue du programme appelant
  - – Sauvegarder les registres au début la procédure.
  - – Exécuté la procédure
  - – Restaurer les valeurs des registres
- ✓ La sauvegarde des registres se fait en utilisant la pile.

Procédure propre (2)



Références bibliographique (1)

- ✓ « Architecture de l'ordinateur », Andrew Tanenbaum chez Pearson Education (5eme édition)
- ✓ « Organisation et architecture de l'ordinateur », William Stallings chez Pearson Education (6eme édition)
- ✓ « Programmer en Assembleur sur PC », Holger Schakel chez Micro application
- ✓ « Compilateur », Dick Grune, Henri E. Bal chez Dunod
- ✓ « Architecture des Ordinateurs », Cecile Germain et Daniel Etiemble

Références bibliographique (2)

- ✓ « Une introduction au langage assembleur », Djamal Rebaïne
- ✓ « Architecture des ordinateurs », Emmanuel Viennet
- ✓ « Architecture », Michel Meynard
- ✓ « Introduction à la programmation en Assembleur », Pierre Jourlin
- ✓ « ASR 1 » et « ASR 2 », Joanna Moulierac
- ✓ http://www.commentcamarche.net/