Architecture Matérielle des Ordinateurs

Troisième Partie : Assembleur

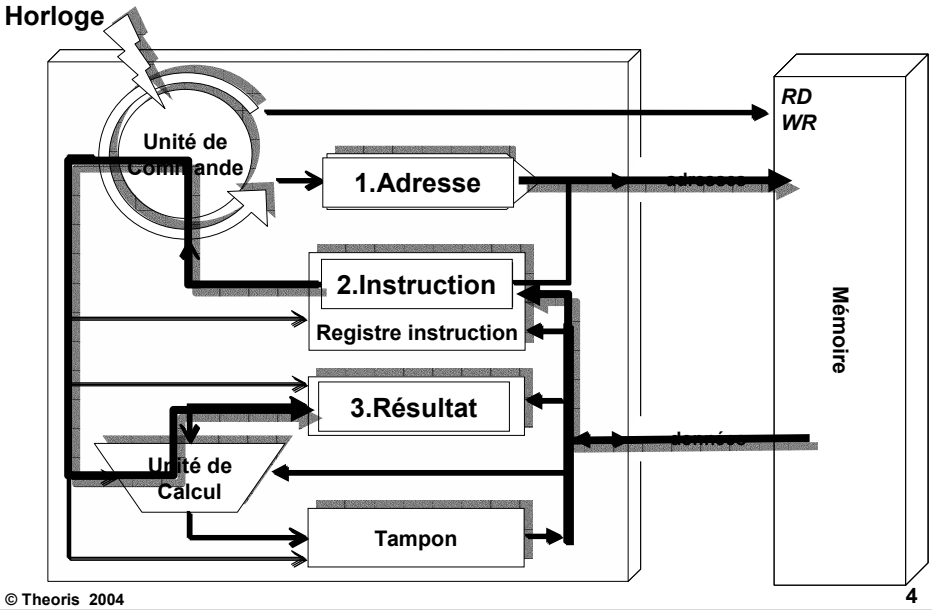
Plan

- **Le calculateur numérique**
- **Les Entrées - Sorties**
- **Notions d'assembleur**
- **Architectures évoluées**

Plan de la troisième partie

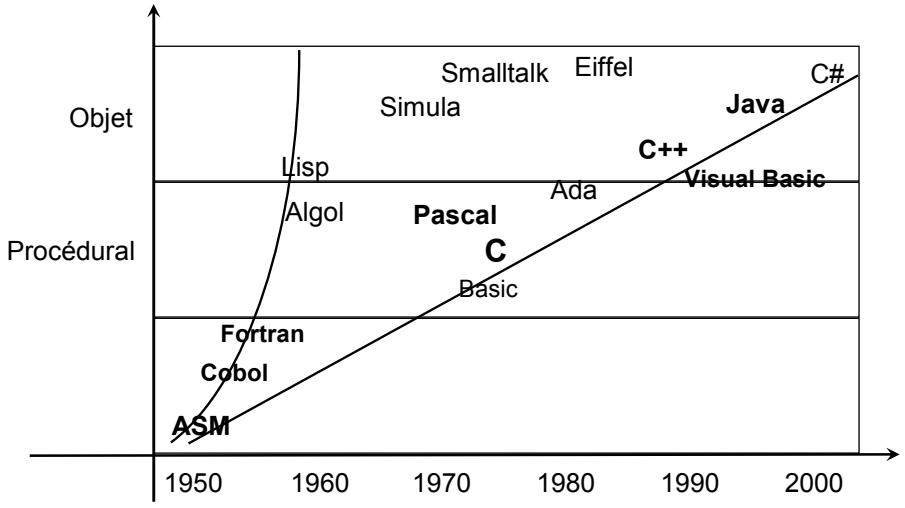
- Langage machine et assembleur
- Un exemple : les processeurs 80x86
- Calculs
- Sauts
- Adressage
- Pile et procédures
- Exécution conditionnelle

Fonctionnement de l'UC



4

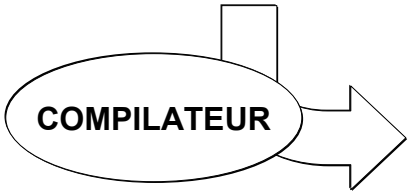
Langages



Langage machine

```
#include "stdafx.h"

int main(int argc, char* argv[])
{
  printf("Hello World!\n");
  return 0;
}
```




```
00401010  55 8B EC 83
00401014  EC 40 53 56
00401018  57 8D 7D C0
0040101C  B9 10 00 00
00401020  00 B8 CC CC
00401024  CC CC F3 AB
00401028  68 1C 00 42
0040102C  00 E8 2E 00
00401030  00 00 83 C4
00401034  04 33 C0 5F
00401038  5E 5B 83 C4
0040103C  40 3B EC E8
00401040  9C 00 00 00
00401044  8B E5 5D C3
```

Langage machine

- Le code machine n'est qu'une suite de nombres
  - Données et programme sont « mélangées »
  - C'est uniquement la valeur du pointeur de programme qui décide de ce que le processeur considère comme l'instruction courante
  - La génération des valeurs qui constituent un programme valide est pratiquement impossible à main nue
  - La localisation des données, du code et de la pile en mémoire doit se plier aux conventions générales du système d'exploitation

Langage assembleur

```
55                  push        ebp
8B EC               mov         ebp,esp
83 EC 40            sub         esp,40h
53                  push        ebx
56                  push        esi
57                  push        edi
8D 7D C0            lea         edi,[ebp-40h]
B9 10 00 00 00      mov         ecx,10h
B8 CC CC CC CC      mov         eax,0CCCCCCCCh
F3 AB               rep stos    dword ptr [edi]
68 1C 00 42 00      push        offset string "Hello World!\n" (0042001c)
E8 2E 00 00 00      call        printf (00401060)
83 C4 04            add         esp,4
33 C0               xor         eax,eax
5F                  pop         edi
5E                  pop         esi
5B                  pop         ebx
83 C4 40            add         esp,40h
3B EC               cmp         ebp,esp
E8 9C 00 00 00      call        __chkesp (004010e0)
8B E5               mov         esp,ebp
5D                  pop         ebp
C3                  ret
```

Langage assembleur

- Le langage d'assemblage rend le code machine manipulable
  - ✓ Les instructions reçoivent des appellations symboliques
  - ✓ Les adresses code et données sont représentées par le biais d'étiquettes
- L'assembleur fabrique le code machine
  - ✓ Le placement du code et des données est calculé automatiquement
  - ✓ Les liens avec le système d'exploitation sont gérés

Un exemple : Intel 80x86

- Une évolution constante sur plus de 20 ans
  - ✓ 8080 : registres 8/16 bits, adresse 64 Ko
  - ✓ 8086 : registres 16 bits, adresse 1 Mo (segmentation)
  - ✓ 80186 : extension du jeu d'instructions
  - ✓ 80286 : apparition du mode protégé (MMU)
  - ⇨ 80386 : registres 32 bits, adresse 4 Go
  - ✓ 80486 : intégration du calcul flottant (FPU)
  - ✓ Pentium : refonte de la FPU, mais le jeu d'instruction évolue peu
  - ✓ Extensions diverses du jeu d'instruction (MMX)

☞ On se limitera aux instructions de base du 80386

- « presque » pas de problèmes de segmentation
- registres 32 bits
- aspects système gérés par la chaîne de programmation

Le problème de la segmentation

Une manière de partitionner l'espace mémoire

- **CS : zone de code**
- **DS : zone de données**
- **SS : zone de pile**

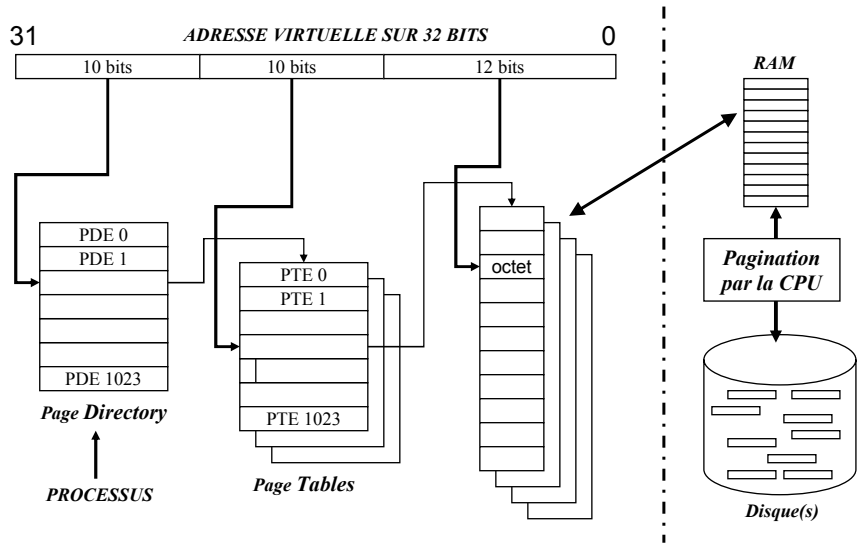
☞ **La gestion des segments est du domaine de l'OS**

- **Détection d'accès non autorisés (écriture dans le code…)**
- **Gestion des débordements de pile (extension de la pile)**

La localisation des données est virtuelle

- **Adresse virtuelle = segment + offset**
- **Adresse physique calculée par la MMU**

Mémoire Virtuelle



Impacts de la segmentation

- **Vue du programmeur**
  - Écriture du code dans des sections prédéfinies
  - Les zones de données, programme et pile sont disjointes
  - Aucune manipulation directe des segments !!
- **Vue de la chaîne de développement**
  - Stockage des différentes sections dans un fichier exécutable
  - Lien avec le code système capable de lancer l'exécution
- **Vue du système d'exploitation**
  - Allocation mémoire lors du chargement du programme
  - Initialisation des différents segments
  - Gestion des protections d'accès

Registres

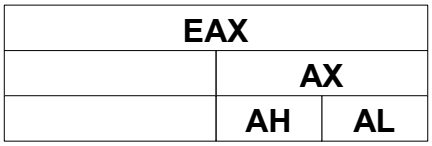
- données : EAX, EBX, ECX, EDX, ESI, EDI
- Tests : FLAGS (carry,zero,sign,overflow,direction)
- pile : ESP, EBP
- code : manipulé par le biais des instructions de saut

Les registres 32 bits EAX à EDX sont décomposables

AX : partie basse de EAX (16 bits)

AL : octet de poids faible de AX

AH : octet de poids fort de AX



Flags

- Bits regroupés dans un même registre
  - Contrôle du fonctionnement du processeur
  - Informations sur la dernière opération effectuée
  - Restrictions d'accès en mode utilisateur
- Contrôle du fonctionnement du processeur
  - Direction des instructions de boucle
  - Masquage des interruptions, changement de mode
- Informations sur la dernière opération effectuée
  - Déroutement de code avec les sauts conditionnels
  - Propagation de la retenue (carry)

Détail des flags

- ZF : le résultat de la dernière opération est nul
- CF : la dernière opération a produit une retenue
- OF : la dernière opération a causé un débordement
- SF : le résultat de la dernière opération est négatif
- DF : définit le sens de déplacement des boucles

☞Les autres flags sont inaccessibles en mode utilisateur

Instructions

- Calcul : quatre opérations, logique, décalages
- Stockage : déplacements en mémoire
- Tests : comparaison de valeurs sans stockage
- Sauts : branchement ou appel de fonction
- Boucles : tests ou transferts sur une zone mémoire
- Divers : positionnement des flags

☞ Le 80386 est une architecture ancienne (CISC)

- Opérations effectuant à la fois un calcul et un stockage
- Primitives de boucles complexes
- Registres peu nombreux et peu polyvalents

Spécialisation des registres

- EAX : accumulateur, supporte tous types de calculs
- EBX : supporte des modes d'adressage complexes
- ECX : compteur de boucles
- EDX : intervient dans les calculs 64 bits et les E/S
- ESI : adressage source des opérations de boucles
- EDI : adressage destination des opérations de boucles

- ESP : pointeur de pile (évoluant avec PUSH et POP)
- EBP : seul autre registre capable d'adresser la pile

Instructions de calcul

Arithmétique
- **addition, soustraction, multiplication, division**
- **négation arithmétique (complément à 2)**
- **incrémentation, décrémentation**
- **extensions de signe (AX vers EAX par exemple)**
- **conversion en décimal codé binaire (obsolète)**

Logique
- **décalages avec ou sans extension de signe**
- **calculs booléens (et, ou, ou exclusif, négation)**

☞ **Les calculs flottants sortent du cadre de cette présentation**

☞ **Idem pour les instructions étendues de type MMX**

Instructions de stockage

Le stockage est un déplacement de mémoire à registre (MOV)

La plupart des calculs opèrent sur des registres.

Il est donc souvent nécessaire :
- **d'initialiser le contenu d'un registre (lecture),**
- **de stocker le résultat d'un calcul en mémoire (écriture).**

Instructions de test

Deux opérations de base
- **soustraction (CMP)**
- **ET logique (TEST)**

But : effectuer une opération uniquement pour positionner les flags, sans stocker le résultat.

L'état des flags après l'opération permet de comparer deux valeurs, ce qui sera en général exploité par une instruction de saut conditionnel.

Le test est donc identique à une opération de calcul, à ceci près qu'il ne nécessite pas de détruire le (précieux) contenu d'un registre.

Registre → Flags

Instructions de saut

- **Saut inconditionnel (JMP)**
  - **Modifie le pointeur de programme**
  - **L'exécution reprendra à l'adresse indiquée**
- **Saut conditionnel (JZ, JNZ, …)**
  - **Même comportement que le saut inconditionnel**
  - **MAIS ne prend effet que si une condition est satisfaite**
  - **Si la condition n'est pas vraie, l'exécution continue en séquence**
- **Appel de fonction (CALL - RET)**
  - **Même comportement que le saut inconditionnel**
  - **MAIS mémorise dans la pile l'adresse de retour**
  - **Une instruction en fin de fonction permet de revenir à l'appelant**

Adressage - 1

- **Immédiat**

  Chargement direct de la valeur indiquée

```
MOV EAX, 10  // charge la valeur 10 dans EAX
```

- **Indirect**

  On indique l'adresse de la donnée à charger

```
MOV EAX, [10] //charge le contenu de
  l'adresse 10 dans EAX (label)
```


Adressage - 2

- **Problème de taille**
  - **Lorsque la source est une constante, il faut préciser sa taille**

    `MOV [EAX],0` **est ambigu :**

    s'agit-il de 1, 2 ou 4 octets ?

  Pour lever l'ambiguïté, on précise la taille

```
MOV byte ptr EAX, 0
```

Adressage - 3

- **Indirect indexé**

  Chargement direct de la valeur indiquée (structures C)

```
MOV EAX, [EBX+10] // charge le contenu de
  l'adresse (EBX+10)
```

- **Encore plus compliqué**

  La famille 80x86 a hérité de son passé CISC, et propose des combinaisons très compliquées de calculs d'adresse

```
MOV EAX, [EBX+ESI]
MOV EAX, [EBX*2]
```

  etc.

Étiquettes

- **Toute donnée peut être manipulée par un symbole qui représente l'endroit où elle est stockée, que ce soit son adresse en mémoire ou un registre**

```
locale1  EQU [EBP+4]
temp     EQU ECX
MOV      temp, locale1
```

- **L'assembleur calcule automatiquement les adresses des données et du code**

```
chaine   DB 'Bonjour, monde !',0
entier   DW 12345
```

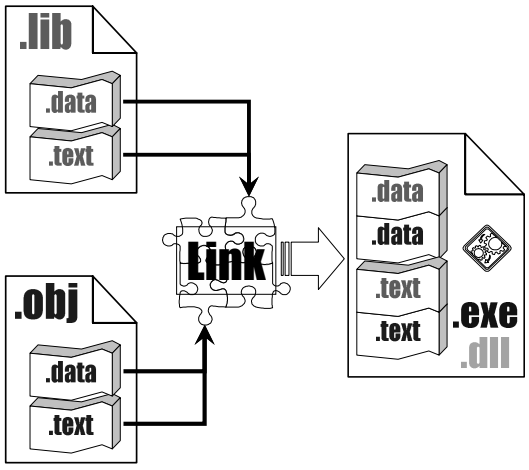
Sections

- **Les sections permettent de ranger code et données dans des zones contiguës de mémoire**
  - **text** : code exécutable
  - data : données initialisées
  - bss : données non initialisées (remises à zéro au démarrage)
  - const : données non modifiables (chaînes de caractères etc.)

```
.text
ne_rien_faire      PROC
                   RET
ne_rien_faire      ENDP
.data
initialisee        DW 12345
.bss
Pas_initialisee    DW ?
.const
chaine             DB 'Bonjour, monde !',0
```


Link & Sections

- **Construction d'un espace d'adressage unique**
- **Fusion des sections**

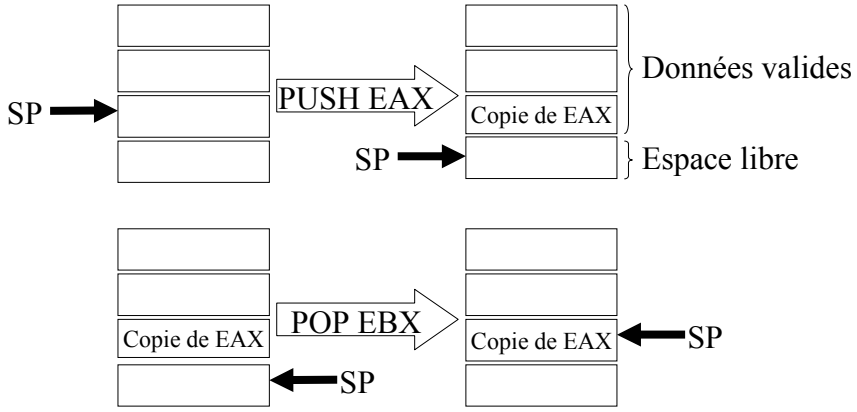


La pile

- **Zone de stockage temporaire**
  - **Permet de sauvegarder le contenu des registres**
- **Mécanisme indispensable aux appels**
  - **Retient l'adresse de retour d'une procédure**
  - **Le même mécanisme sert de base à la gestion des interruptions**
    - **CS, IP, FLAGS → Pile**
- **Outil de base des langages évolués**
  - **Permet le passage de paramètres**
  - **Autorise la création de variables locales**

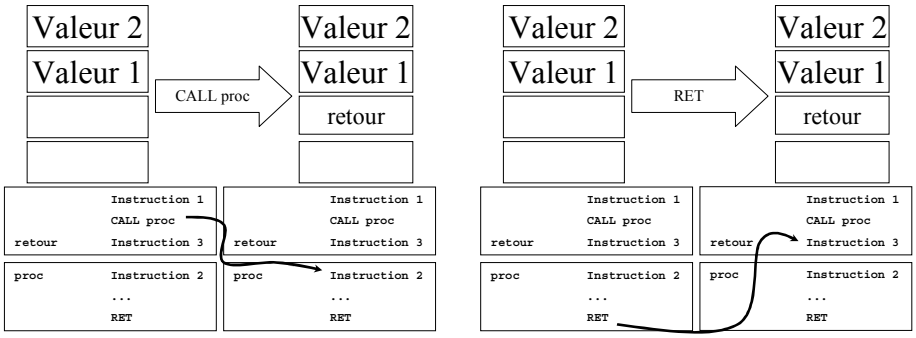
Pile : fonctionnement général

- **Gestion en LIFO (last in – first out)**
- **Croissance « vers le bas »**



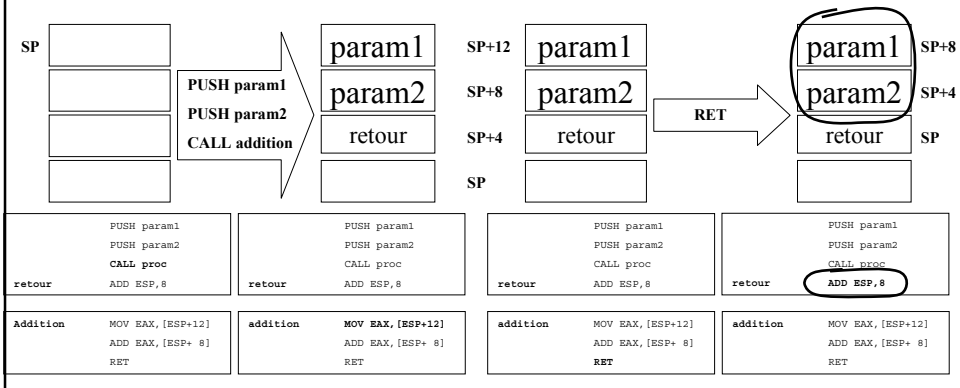
La pile : appel de fonction

- **Sauvegarde de l'adresse de retour**
- **Modification du pointeur d'instructions**



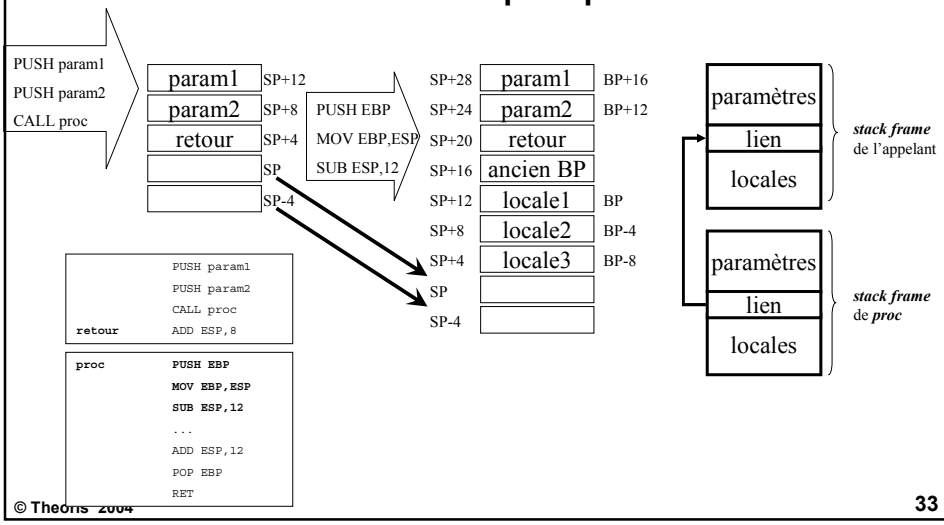
La pile : passage de paramètres

- **Nécessaire pour palier au manque de registres**
- **« Nettoyage » de la pile après l'appel**



La pile : variables locales

- **Notion de *stack frame***
- **Référence commune pour paramètres et locales**



Exécution conditionnelle

- **Tout n'est qu'une suite d'alternatives**

  les structures de contrôle des langages évolués se ramènent (presque) toujours à une série de sauts conditionnels

- **Des algorithmes de génération de code peuvent rendre les structures d'exécution conditionnelles peu lisibles pour des raisons d'optimisation.**

Structure si/sinon

```
if (a > b)
{
      gagne = 1;
}
else
{
      gagne = 0;
}
```

```
                MOV EAX, a
                CMP EAX, b
                JLE test_faux
test_vrai:      MOV EAX, 1
                JMP test_fin
test_faux:      MOV EAX, 0
test_fin:       MOV gagne, EAX
```


Structure for

```
for ( i=0 ; i != 10 ; i++ )
{
      resultat+=tableau[i];
}
```

```
        MOV EAX, resultat
        MOV EBX, 0
boucle: ADD EAX, tableau[EBX]
        INC EBX
        CMP EBX, 10
        JNE boucle
fin:    MOV resultat, EAX
```