Architecture des Ordinateurs et Systèmes d'Exploitation

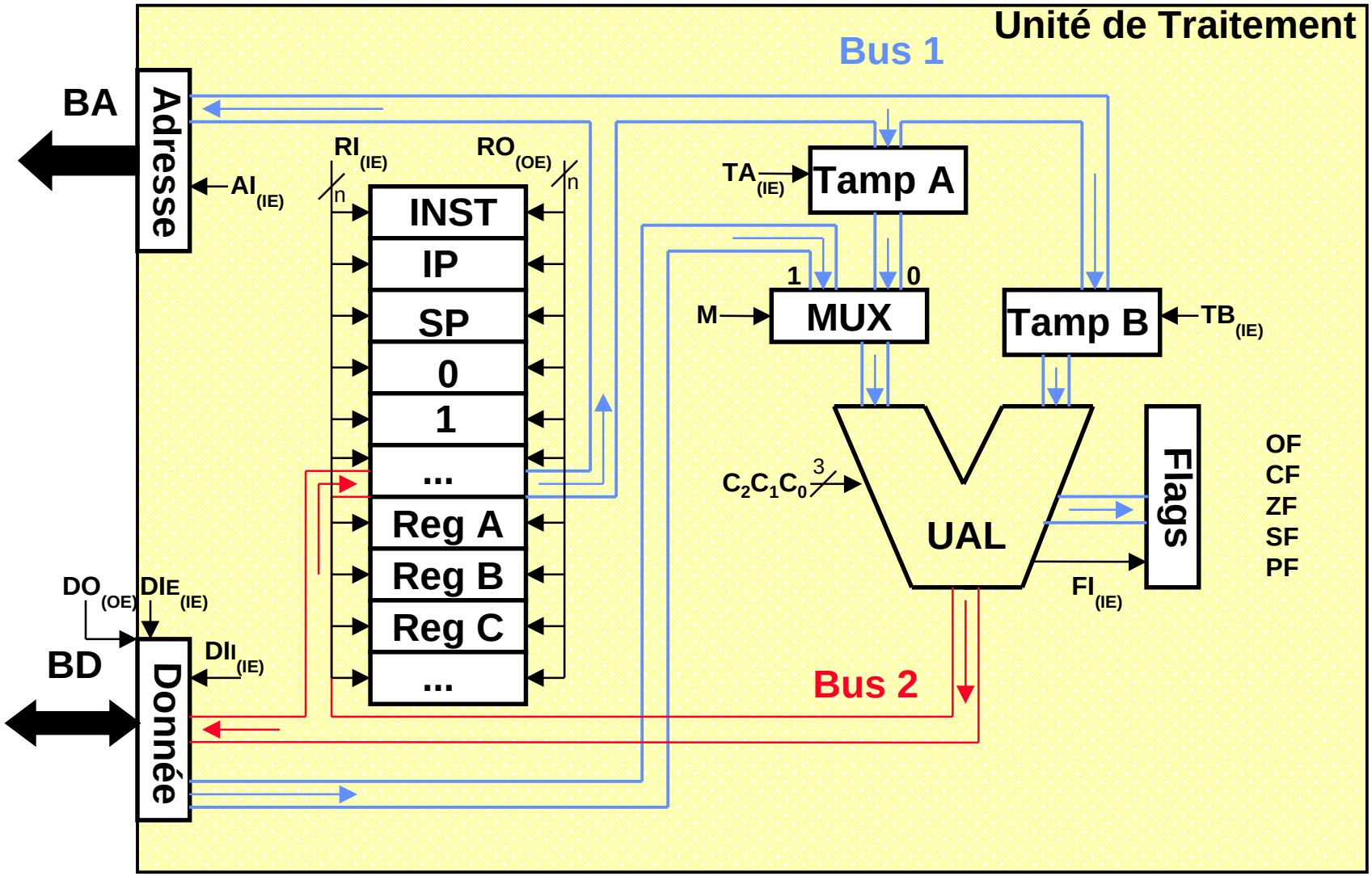
Cours n°3

La couche micro-programmée
De la micro-instruction au langage machine

Ph. Leray

3ème année

Chemin de données : rappel



Séquencement des actions : μ-instruction

Exemple :	**Reg A ← Reg A + Reg B**	Signaux
Actions :	**Transfert (Bus 1) Reg A → Tamp A**	**RO_{10} , TA**
	Transfert (Bus 1) Reg B → Tamp B	**RO_{11} , TB**
	Choix UAL (C=100, M=0, Flags)	**$C_2C_1C_0$,M, FI**
	Transfert (Bus 2) UAL → Reg A	**RI_{10}**

Une micro-instruction = ensemble des actions faisant fonctionner une fois le chemin de données

μ- instruction = 5 sous-cycles:

1. **Transfert n°1 par le Bus 1**
2. **Transfert n°2 par le Bus 1**
3. **Opération UAL**
4. **Transfert résultat par le Bus 2 (+ observation des FLAGS)**
5. **Accès à la mémoire (READ / WRITE)**

µ-programmes simples 1/2

1. Faire l'addition de 2 registres et mettre le résultat dans l'un des deux

ex : IP ← IP + Reg1

10: TampA←IP ; TampB←Reg1 ; UAL←TampA+TampB ; IP←UAL;

(signaux: RO_1 , TA ; RO_4 , TB ; C=100 , M=0 , FI ; RI_1)

2. Faire des sauts conditionnels :

Conditions possibles = FLAGS en sortie de l 'UAL (Bus 2)

ex : Si RegA=0 aller en 30

10: TampA←RegA ; TampB←Reg0 ; UAL←TampA+TampB; on ZF jmp 30 ;

(signaux: RO_{10} , TA ; RO_3 , TB ; C=100 , M=0 , FI ;)

(NB: Il faut faire une opération pour calculer ZF mais UAL ←TampA n 'en est pas une)

µ-programmes simples 2/2

3. Envoyer le contenu d'une case mémoire d'adresse XX dans un registre

(Hypothèse: l'adresse XX est dans un autre registre)

ex : RegA ← [RegB]

– il faut commencer par mettre **RegB** dans **Adresse** et déclencher une opération de lecture (Un accès à la mémoire (READ / WRITE) prend 2 cycles)

10: Adresse←RegB ;	;	;	; READ
(signaux: AI , TB	;	;	;	; DI_E)
11:	;	;	;	; READ

– la valeur est ensuite disponible dans **Donnée**. Pour l 'envoyer dans un autre registre, il faut passer par l 'UAL

12:	;	; UAL←Donnée	; RegA←UAL ;
(signaux:	;	; C=001, M=1	; RI_{10})

« Optimisation »

= Occuper les bus le + possible :

ex : RegA ← [RegB]

10: Adresse←RegB ;	;	;	;	READ
11:	;	;	;	;	READ
12:	;	;	UAL←Donnée ;	RegA←UAL;	

IP ← IP + Reg1

13:	TampA←IP ;	TampB←Reg1;	UAL←TampA+TampB;	IP←UAL;

→ Ici, on peut faire les 2 opérations en même temps

10: Adresse←RegB ;	;	;	;	READ
11:	TampA←IP ;	TampB←Reg1 ;	UAL←TampA+ TampB ;	IP ←UAL ;	READ
12:	;	;	UAL←Donnée ;	RegA←UAL ;	

Fonctionnement de l'unité de commande

L'unité de commande exécute les opérations suivantes :

- **Lecture du code de l'opération**
- **Lecture des arguments**
- **Traitement**
- **Sauvegarde du résultat**
- **Passage à l'instruction suivante**

μ-programme de lecture du code de l 'opération

(l 'opération à effectuer est en mémoire à l 'adresse IP)

- lecture de l 'opération

0: Adresse←IP ;	;	;	; *READ*

- on en profite pour passer à la case mémoire suivante

1: TampA←IP ; TampB←Reg1 ; UAL←TampA+TampB ; IP←UAL	; *READ*

- exécution du μ-programme correspondant à l'opération

2: ;	; UAL←Donnée	; INST←UAL, jmp μ-Prog[INST];

Exemple n°1

Add RegA, XX **(RegA ← RegA + [XX])**

1. **L 'UC lit le « codop » de Add RegA dans la mémoire et déclenche le μ-programme correspondant (0: 1: 2:)**
2. **Exécution de μ-programme[Add RegA,XX]**
   - 2.1 : Lecture de l 'argument XX en mémoire
     - » lecture de la valeur XX en mémoire (INST ← [IP])
     - » lecture de la valeur située à l'adresse XX (Donnée ← [INST])
   - 2.2 : Traitement de l 'opération (RegA ← RegA + Donnée)
   - (2.3 : Sauvegarde du résultat)
   - 2.4 : Passage à l 'instruction suivante (IP ← IP + 1)

20: Adresse←IP	;	;	;	; READ
21: TampA←IP	; TampB←Reg1	; UAL←TampA+TampB	; IP ←UAL	; READ
22:	;	; UAL←Donnée	; INST←UAL	;
23: Adresse←INST	;	;	;	; READ
24:	;	;	;	; READ
25:	; TampP ←RegA	; UAL←Donnée+TampB	; RegA ←UAL, jmp 0;	

Exemple n°2

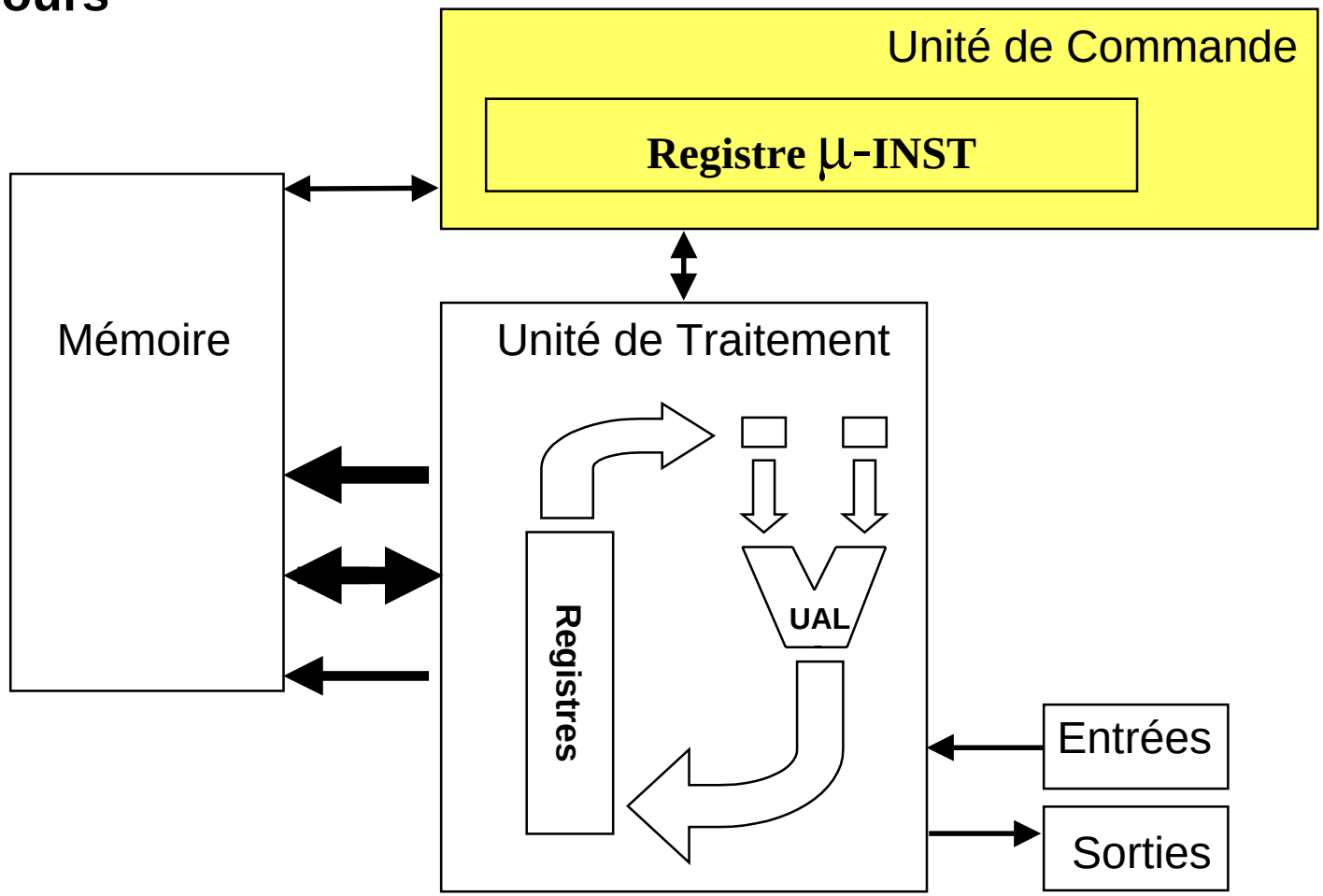
JZ xx (va en IP+xx+1 si le résultat de l 'UAL est nul)

1. **L 'UC lit le « codop » de JZ dans la mémoire et déclenche le μ-programme correspondant (0: 1: 2:)**
2. **Exécution de μ-programme[JZ XX]**
   - 2.1 : Lecture de l 'argument XX en mémoire (Donnée ← [IP])
   - 2.2 : Traitement de l 'opération (Test sur ZF, IP←IP+1+xx)
   - (2.3 : Sauvegarde du résultat)
   - 2.4 : Passage à l 'instruction suivante si la condition n 'est pas réalisée

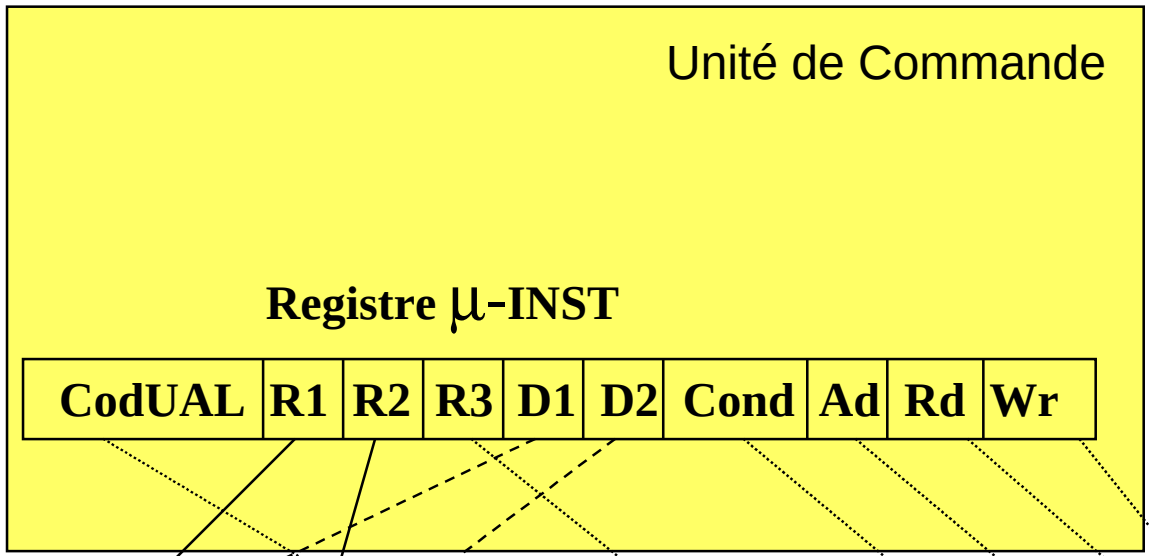
30: Adresse←IP	;	;	; on ZF jmp 32	; READ
31: TampA←IP	; TampB←Reg1	; UAL←TampA+TampB	; IP←UAL, jmp 0	;
32: TampA←IP	; TampB←Reg1	; UAL←TampA+TampB	; IP←UAL	; READ
33:	; TampB←IP	; UAL←Donnée+TampB	; IP←UAL, jmp 0	;

Fonctionnement de l'unité de commande (2)

- **L 'UC possède un registre spécial pour stocker la μ-instruction en cours**



Fonctionnement de l'unité de commande (3)



32: TampA←IP ; TampB←Reg1 ; UAL←TampA+TampB ; IP←UAL, on ZF jmp 0; READ

➔ A partir du registre de µ-instruction, il faut générer les signaux de commande de l 'UAL dans un ordre précis …

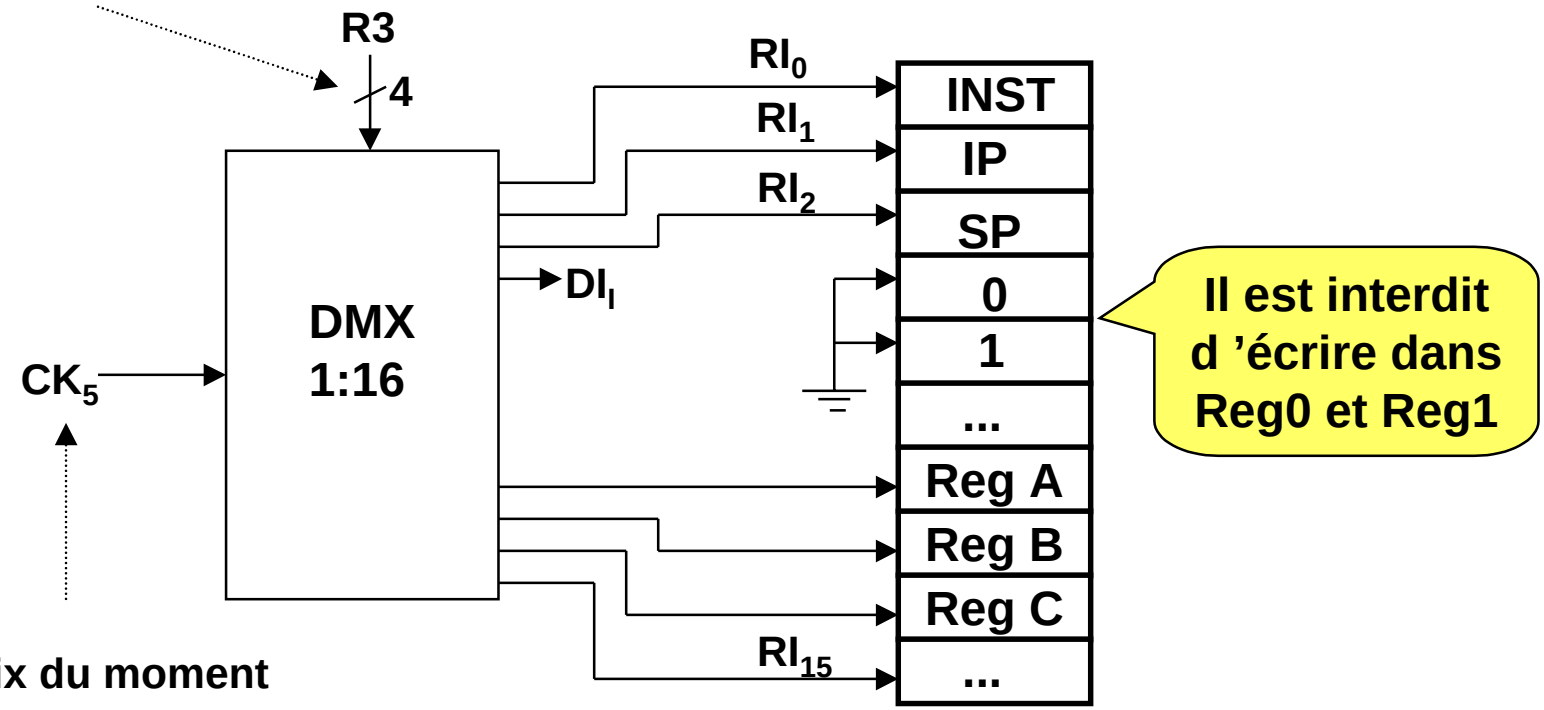
Comment déclencher les opérations séquentiellement ?

Séquencement des opérations (1)

Chaque partie de μ-INST est reliée à un DCD (ou un DMX) qui déclenchera le bon signal de commande, au bon moment :

Ex: R3 = registre destination de la sortie de l 'UAL par le Bus 2 (DI_I, RI_0… RI_{15})

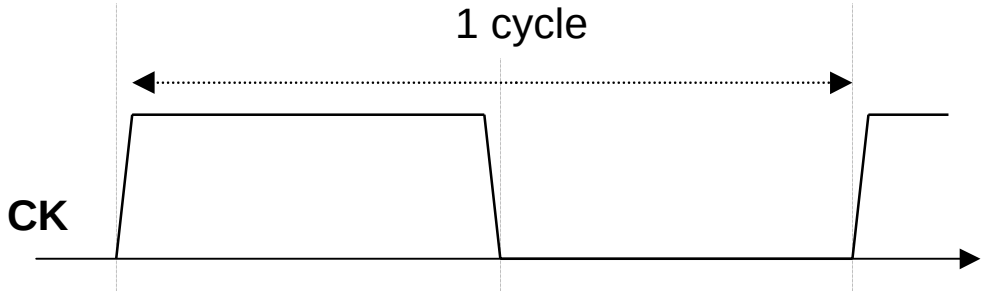
Choix du signal de commande



Choix du moment

Séquencement des opérations (2)

- **CK = Signal d ’horloge**



- **CK_i = Sous-cycles de CK correspondants aux sous-cycles des µ-instructions**

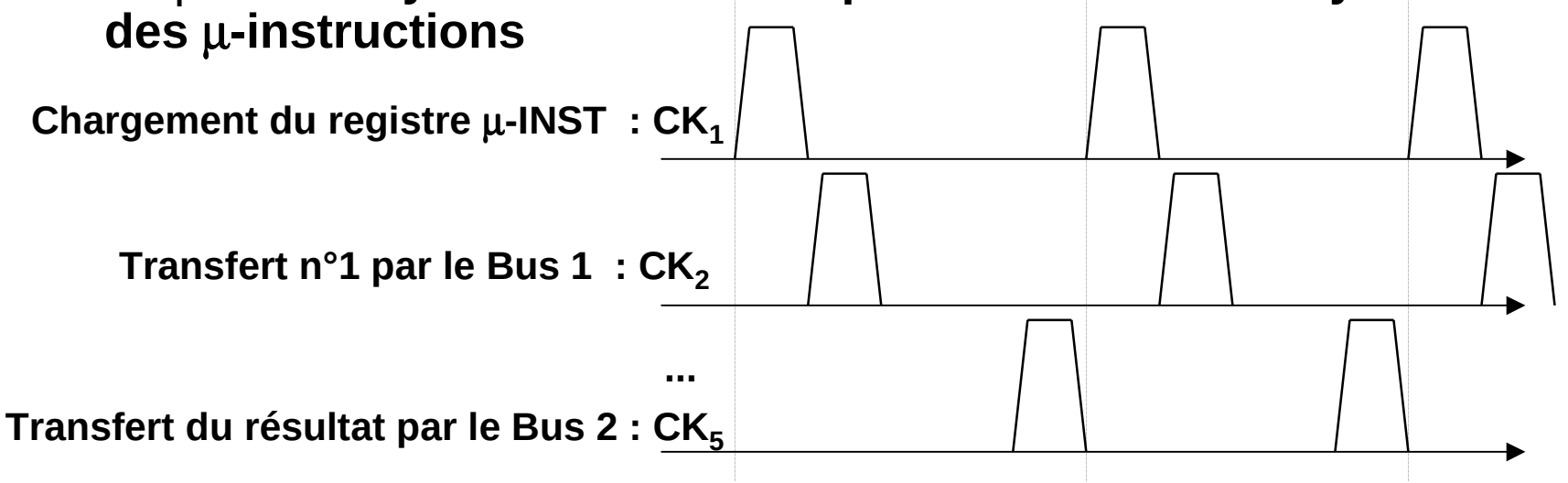
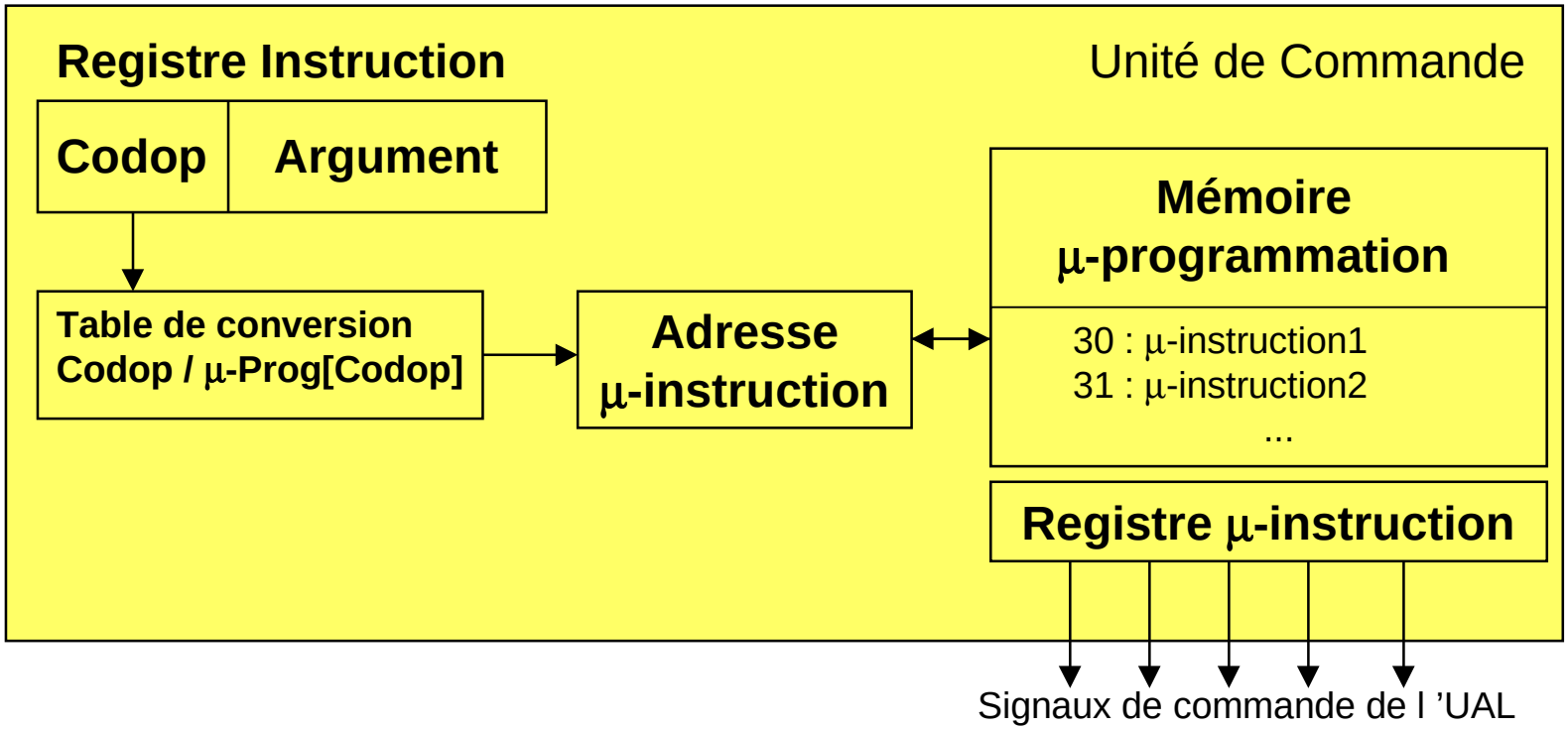


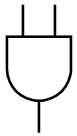
Schéma de l'unité de commande



• **Lorsqu'une instruction arrive dans RI, son Codop est envoyé dans une table de conversion qui indique l'adresse du µ-Programme correspondant.**

• **Ce µ-Programme est exécuté ligne par ligne : chaque µ-instruction est envoyée dans le registre de µ-instruction où sont générés séquentiellement les signaux de commande de l 'UAL**

Niveaux de programmation



MUX ... 0/1

Unité de Traitement
(UAL, Chemin de données)

Unité de Commande

μ-instructions
= suite de 0/1

μ-programme
= suite de μ-instructions

Codop

101010	000010100

Langage machine = suite de 0/1

ADD A,20
JZ 13

Assembleur = remplacer les codop par
des codes mnémotechniques

Architecture des Ordinateurs et Systèmes d'Exploitation

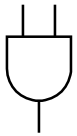
Cours n°4

Le langage assembleur : Introduction et Présentation de l'assembleur du 8086

Ph. Leray

3ème année

Niveaux de programmation (rappel)



MUX ... 0/1

Unité de Traitement
(UAL, Chemin de données)

Unité de Commande

μ-instructions
= suite de 0/1

μ-programme
= suite de μ-instructions

Codop

101010	000010100

Langage machine = suite de 0/1

ADD A,20
JZ 13

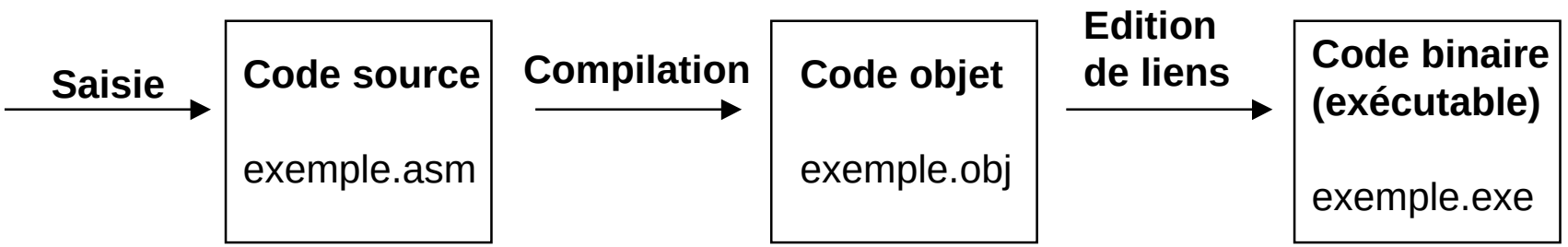
Assembleur = remplacer les codop par
des codes mnémotechniques

Quelques définitions

- **Langage Machine = ordres (en binaire) compréhensibles par un processeur donné**
- **Exécutable = suite d'instructions en langage machine**
- **Langage Assembleur = mnémoniques associées au langage machine (JUMP, ADD, MOV…)**
- **Assemblage = utilisation d'un logiciel spécifique (logiciel d'assemblage ou *assembleur*) pour transformer une suite d'instructions écrites en langage assembleur en un exécutable**

Processus d'assemblage

- **3 phases :**



1. **Saisie du code source avec un éditeur de texte**
2. **Compilation du code source**
3. **Edition des liens**
   - » **permet de lier plusieurs codes objets en un seul exécutable**
   - » **permet d 'inclure des fonctions prédéfinies dans des bibliothèques**

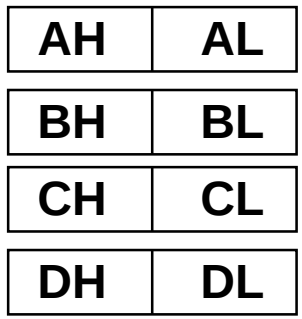
Les microprocesseurs

- **Deux grandes familles :**
  - **Les microprocesseurs CISC (Complex Instruction Set Computer)**
    - » **Jeu d'instruction étendu ⇒ taille « moyenne »**
    - » **2 grands constructeurs :**
      - **INTEL (8088, 8086, 80286… Pentium…)**
      - **MOTOROLA (68000… 68040)**
  - **Les microprocesseurs RISC (Reduced Instruction Set Computer)**
    - » **Jeu d'instruction réduit ⇒ taille « faible »**
    - » **cela permet de rajouter plus de registres, de mémoire cache, … sur le processeur**
      - **IBM/MOTOROLA (PowerPC)**
      - **SUN (Supersparc)**
      - **DIGITAL (Alpha)**

Le microprocesseur 8086

- **Microprocesseur des premiers PC et compatibles**
- **Compatibilité ascendante : un programme écrit pour le 8086 marche pour les processeurs suivants**
- **Caractéristiques :**
  - **Bus de données : 16 bits**
  - **Bus d'adresse : 20 bits**
  - **Registres : 16 bits**
    - » **Accumulator (AX)**
    - » **Base (BX)**
    - » **Counter (CX)**
    - » **Data (DX)**
    - » **Pointeur d'instruction (IP)**
    - » **Registres segments code (CS), data (DS), extra (ES), stack (SS)**
    - » **Pointeur de pile (stack) (SP), de base (BP)**
    - » **Index source (SI), Index destination (DI)**

Certains registres 16 bits peuvent être utilisés comme deux registres 8 bits



L'adressage par segment

- **Largeur du bus d 'adresse = 20 bits**

⇒ **Adressage mémoire = 2^{20} = 1 Mo**

- **Le pointeur d 'instruction fait 16 bits**

⇒ **Adressage = 2^{16} = 64 Ko** cela ne couvre qu 'une partie de la mémoire

= 1 segment de mémoire

- **Il faut 2 registres pour indiquer une adresse au processeur**

Ex: CS:IP = (4055:3192) = 40550 + 3192 = 436E2

Adresse de début de segment — Offset — Adresse pour le bus de donnée

CS:IP, (index) DS:SI, DS:DI, ES:SI, ES:DI, (pile) SS:SP, SS:BP

Types d'adressage 8086

- **Adressage par registre (ou implicite)**
  Add AX, BX
  Inc AX **(pas d 'accès mémoire pour les opérandes)**

- **Adressage immédiat**
  Add AX, valeur **(1 accès mémoire pour lire la valeur)**

- **Adressage direct**
  Add AX, [adresse] **(2 accès mémoire : adresse puis valeur [adresse])**

- **Adressage indexé (ou relatif)**
  Add AX, [adresse+index] **(2 accès mémoire)**

Jeu d'instructions 8086 (1/2)

- **Instruction d'affectation** **MOV**

- **Instructions arithmétiques**
  INC (incrémentation)
  DEC (décrementation)
  ADD (addition)
  SUB (soustraction)
  CMP (soustraction sans sauvegarde)
  NEG

- **Instructions logiques**
  NOT, OR, XOR
  AND, TEST (= AND sans sauvegarde)
  SHL (SHR), SAL (SAR)
  ROL (ROR), RCL (RCR)

Jeu d'instructions 8086 (2/2)

- **Branchement :** **JMP**

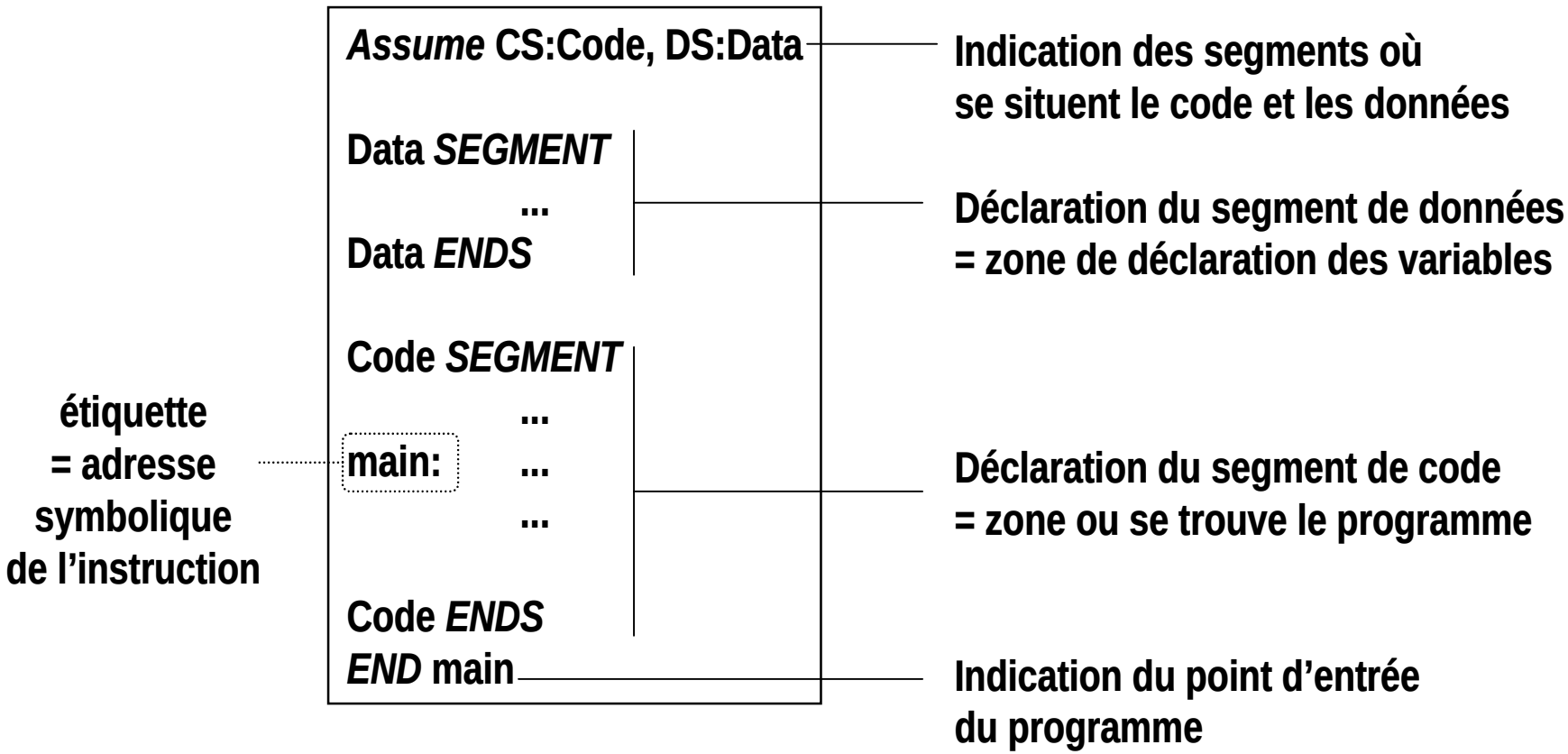
- **Branchements conditionnels :** **JE/JZ (JNE/JNZ) : Jump if zero**
  JO (JNO) : Jump if overflow
  JS (JNS) : Jump if sign

- **Comparaison de valeurs**
  CMP AX, BX suivi d 'un test :

	(entiers naturels)	(complément à 2)
AX > BX ?	JA (≥ JAE)	JG (JGE)
AX < BX ?	JB (JBE)	JL (JLE)

Programmation 8086 (1/2)

- **Structure du programme**



Programmation 8086 (2/2)

- **Déclaration des variables (segment de données)**

```
Assume CS:Code, DS:Data

Data SEGMENT
  N1        dw      25h
  N2        dw      4
  Prod      dw      ?
            ...
  NP        dw      2,3,5,7,11,13
  Table     dw      10,20,30,40,50


TabVide     dw      100 dup (?)
  Tab1      dw      50 dup (1b)

Chaine      db      'TEXTE',0Dh,0Ah

Data ENDS
```

db = Define Byte (Byte = Octet = 8 bits)
dw = Define Word (Word = Mot = 16 bits)
dd = Define Double (Double = 32 bits)
+ valeur initiale (ou liste de valeurs) ou de ?

N1 est une variable de type WORD (dw = Define Word) initialisée à 25 (en héxa)

Table est un tableau (une suite) de WORD initialisé à 10…50 (en décimal)

TabVide est un tableau de 100 WORD non initialisé

Tab1 est un tableau de 50 WORD initialisés à 1 (en binaire)

Les appels système DOS

Pour réaliser les opérations standards (affichage, saisie, …) le système d'exploitation (ici DOS) fournit des fonctions pré-écrites :

– Affichage d'un caractère	Mov DL,"A"	; caractère
	Mov AH, 2	; fonction n°2
	Int 21h	; appel système
– Saisie d'un caractère (avec écho)	Mov AH,1	
	Int 21h	; (résultat dans AL)
– Saisie d 'un caractère (sans écho)	Mov AH,7	
	Int 21h	; (résultat dans AL)
– arrêt du programme	Mov AH, 4Ch	; A mettre a la fin de
	Int 21h	; chaque programme

Mon premier programme

```
Assume CS:Code, DS:Data
Data SEGMENT
   N1          dw          25h
   N2          dw          4
   Prod        dw          ?
Data ENDS
Code SEGMENT
  main :       Mov AX, Data
               Mov DS,AX
               Mov AX, N1
               Mov CX, N2
               Mov DX, 0
  boucle :     Add DX, AX
               Loop boucle
               Mov Prod, DX
               Mov AH, 4Ch
               Int 21h
Code ENDS
END main
```

Indication du segment de données à utiliser dans le programme

Loop = instruction de boucle qui équivaut à
Dec CX
JNZ boucle
(CX sert de compteur de boucle)

A quoi sert ce programme ???

Architecture des Ordinateurs et Systèmes d'Exploitation

Cours n°5

Le langage assembleur (2): Pile, Procédures. Les Interruptions

Ph. Leray

3ème année

L'assembleur 8086 : la pile

= structure de « rangement » de données

- **définie dans un segment de mémoire particulier**
- **fonctionnement *LIFO* (Last In, First Out)** **ex: pile d'assiettes**
- **comment y accéder :**
  - **adresse du dernier élement posé : SS:SP**
  - **empiler : PUSH Reg16 (registre 16 bits)**
  - **dépiler : POP Reg16 (registre 16 bits)**
- **« la pile augmente vers les adresses faibles »**

  (PUSH : SP←SP-2, POP : SP←SP+2)
- **Déclaration du segment de pile :**

Déclaration d'une pile de 1024 octets (=1 Ko) initialisés à 0

```
Assume SS:Pile

Pile SEGMENT STACK
                dw 1024 dup (0)
Pile ENDS
```

L'assembleur 8086 : les procédures (1/5)

- **Déclaration d'une procédure :**
  - **appel proche (near) : la procédure et le programme sont dans le même segment de code**
  - **appel lointain (far) : la procédure et le programme sont dans deux segments de code différents**

```
Moyenne PROC [NEAR/FAR]
        instructions
Moyenne ENDP
```

- **Appel de la procédure dans le programme**

```
Call Moyenne
```

Comment l'unité de traitement arrive-t-elle à retourner au programme principal à la fin de la fonction ?

- **Au moment de l'appel de la fonction, l'adresse de l'instruction suivante est sauvegardée dans la pile :**
  - **appel proche (near) : sauvegarde de IP**
  - **appel lointain (far) : sauvegarde de CS puis de IP**

⇒ ***A la fin de la procédure, l 'UT récupère les valeurs sauvegardées pour retourner au programme principal***

L'assembleur 8086 : les procédures (2/5)

- **2 méthodes de passage de paramètres : * par registre * par la pile**

- **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**
  - **avantage = rapidité inconvénients = peu de registres pas de « portabilité »**
  - **Ex:**

```
; programme principal
            ...
            Mov AX, N1
            Mov BX, N2
            Call Moyenne
            Mov Res, AX
            ...
```

```
Moyenne PROC
            Add AX, BX
            SHR AX, 1
            Ret
Moyenne ENDP
```

Ret = dépilement de IP pour retourner au programme principal

L'assembleur 8086 : les procédures (3/5)

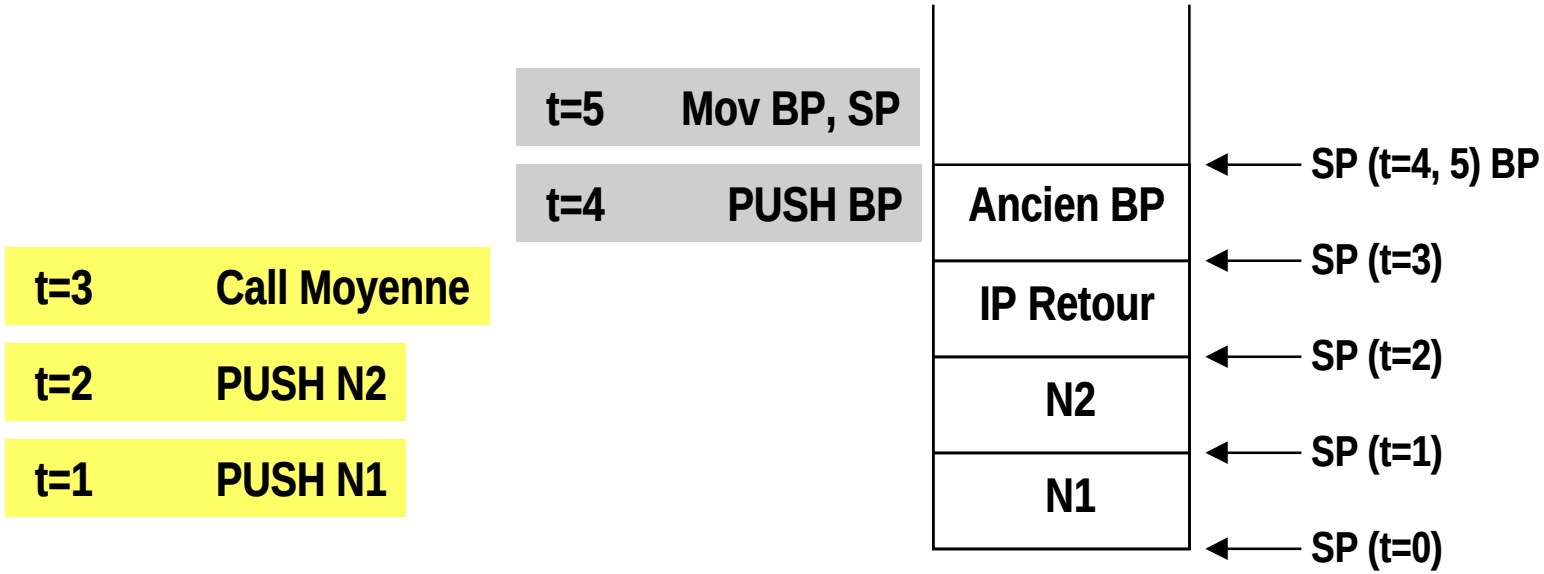
- **Passage de paramètres par la pile :**
  - **les paramètres d'entrée sont empilés avant l 'appel de la procédure**
  - **les paramètres de sortie sont dépilés par le programme principal**
  - **avantage = « portabilité »** **inconvénients = récupération des paramètres plus « lourde »**
  - **Ex:**

```
; programme principal
        ...
        PUSH N1
        PUSH N2
        Call Moyenne
        POP Res
        ...
```

```
Moyenne PROC
        PUSH BP
        Mov BP, SP
        Mov AX, [BP+4]
        Add AX, [BP+6]
        SHR AX, 1
        Mov [BP+6], AX
        POP BP
        Ret 2
Moyenne ENDPc
```

L'assembleur 8086 : les procédures (4/5)

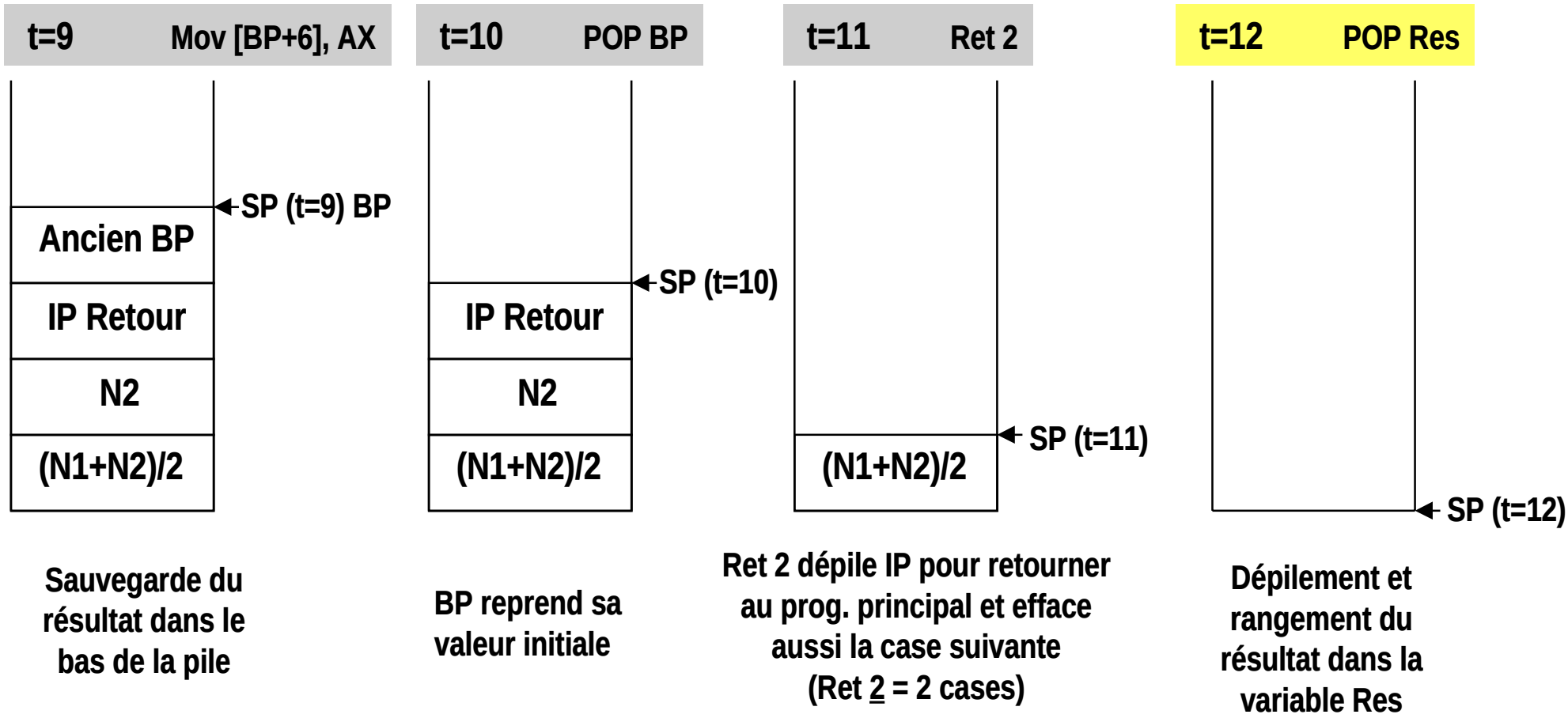
- **Passage de paramètres par la pile : évolution de la pile**



- **A quoi sert BP ?**
  - **BP = référence de la pile au début de la procédure**
  - **permet d'avoir une référence fixe pour récupérer les paramètres**
    ici : [BP+4] pour N2 et [BP+6] pour N1

L'assembleur 8086 : les procédures (5/5)

– **Sauvegarde du résultat et « nettoyage de la pile » :**



Quels paramètres ?

- **On peut utiliser deux paramètres différents :**
  - **la valeur de la variable**
  - **l 'adresse de la variable**

```
; passage des valeurs par registre
        ...
        Mov AX, N1
        Mov BX, N2
        Call Moyenne_par_valeur
        Mov Res, AX
        ...
```

```
Moyenne_par_valeur PROC
        Add AX, BX
        SHR AX, 1
        Ret
Moyenne_par_valeur ENDP
```

```
; passage des adresses par registre
        ...
        Mov SI, Offset N1
        Mov DI, Offset N2
        Mov BX, Offset Res
        Call Moyenne_par_adresse
        ...
```

```
Moyenne_par_adresse PROC
        Mov AX, [SI]
        Add AX, [DI]
        SHR AX, 1
        Mov [BX],AX
        Ret
Moyenne_par_adresse ENDP
```

(Idem pour le passage par pile)

Procédure sale / procédure propre

Procédure sale = modifie les registres

```
Moyenne PROC
        PUSH BP
        Mov BP, SP
        Mov AX, [BP+4]
        Add AX, [BP+6]
        SHR AX, 1
        Mov [BP+6], AX
        POP BP
        Ret 2
Moyenne ENDP
```

A la fin de la procédure, AX est modifié

Procédure propre = ne modifie pas (ou restaure en fin de procédure) les registres

```
Moyenne PROC
        PUSH BP
        Mov BP, SP
        PUSH AX
        Mov AX, [BP+4]
        Add AX, [BP+6]
        SHR AX, 1
        Mov [BP+6], AX
        POP AX
        POP BP
        Ret 2
Moyenne ENDP
```

AX est sauvegardé en début de procédure et restauré à la fin

Les interruptions

- **En général, l 'Unité de Traitement exécute séquentiellement les instructions ou effectue des sauts programmés (JMP, CALL)**

- **Il existe des situations où l 'U.T. est « déroutée » de sa tâche :**
  - **Reset : signal envoyé au processeur pour un (re-)démarrage**
  - **Exceptions (interruptions internes) : débordement de pile, dépassement de capacité, div /0, …**
  - **Appels systèmes (int. logicielles) : appels du programme lui-même « int 21h » …**
  - **Interruptions physiques (int. externes) : appels d'autres périphériques**

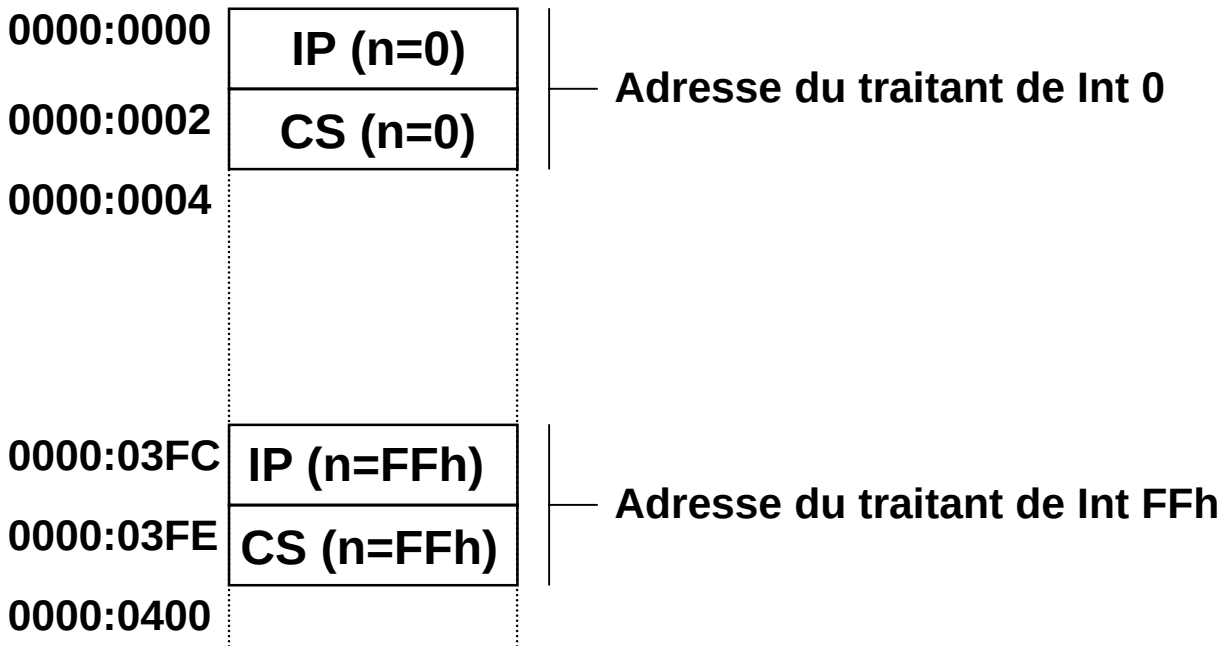
- **Les interruptions sont +/- prioritaires**
  - **Int. non masquables : l 'UT est obligée de s 'en occuper (Reset,…)**
  - **Int. masquables : l 'UT peut ne pas en tenir compte (grâce au drapeau IF)**

Les interruptions : fonctionnement 1/2

- **Chaque interruption possède un numéro de 0 à 255 (FFh) (ex: Int 21h)**
- **Lorsque l'interruption est déclenchée, l 'U.T. doit exécuter un bout de programme bien précis : le traitant de l 'interruption.**

⇒ **l 'U.T. doit savoir où se trouve l 'adresse *(CS:IP)* du traitant numéro *n***

- **Cette information est stockée en mémoire (0000:0000 à 0000:03FFh)**



Les interruptions : fonctionnement 2/2

- **Au démarrage, la table des interruptions contient des valeurs « par défaut »**
- **Il est possible de modifier cette table :**
  - **pour ajouter/supprimer un traitant**
  - **surcharger un traitant (ex: ajouter un message d 'erreur en plus du div/0)**

- **Comment ? : appels systèmes (ici l 'exemple du DOS)**
  - **Lecture de l 'adresse du traitant nn :**

    Fonction DOS 35h (entrée : AH=35h, AL=nn sortie : ES=CS(nn), BX=IP(nn))

  - **Modification de l 'adresse du traitant nn :**

    Fonction DOS 25h (entrée : AH=25h, AL=nn, DS=CS(nn), DX=IP(nn))

  - **Retour au dos en laissant le traitant en mémoire (TSR = Terminate and Stay Resident)**

    Fonction DOS 31h (entrée : AH=31h, AL=nn, DX=Taille du traitant en blocs de 16 octets)