

R2.04: **Communication et fonctionnement bas niveau:** **Assembleur**

U.E.2.2 et U.E.2.3

Sophie Voisin



Introduction

► Séquence pédagogique:

- 4 cours de 1H: 1 x 4 semaines
- 4 TD de 1H30: 1 x 4 semaines
- 5 TP de 1H30: 1 x 5 semaines
- 1 contrôle (1H30) la semaine du 5 juin

► Un seul intervenant: Sophie Voisin



Introduction

► Objectif de la ressource R2.04 :

- Comprendre le fonctionnement des couches systèmes et réseaux bas niveau
- Découvrir les multiples technologies et fonctions mises en œuvre dans un réseau informatique
- Comprendre les rôles et structures des mécanismes bas niveau mis en œuvre pour leur fonctionnement

3

Introduction

► Savoirs de référence à étudier:

- Étude d'un système à microprocesseur ou microcontrôleur avec ses composants (mémoires, interfaces, périphériques, etc.)
- Langages de programmation de bas niveau et mécanismes de bas niveau d'un système informatique
- Étude d'architectures de réseaux et notion de pile protocolaire – Technologie des réseaux locaux : Ethernet, WiFi (Wireless Fidelity), TCP/IP , routage, commutation, adressage, transport

► Mots clés: Protocoles Pointeurs Interruptions
Langage bas niveau

4

Plan du cours

- ▶ Introduction
- ▶ **Modèle de microprocesseur**
- ▶ Programmation en assembleur
- ▶ Tests
- ▶ Entrées Sorties
- ▶ Pointeurs
- ▶ Procédures et fonction

5

Modèle de microprocesseur

- ▶ R1.03: Introduction à l'architecture des ordinateurs
 - Ordinateur \supset Carte mère \supset Microprocesseur
 - Microprocesseur = Unité Centrale de Traitement = Central Processing Unit (CPU)
 - **CPU= UC + UT + registres**
 - UC (Unité de contrôle):
 - **Séquenceur**: description en séquence des opérations élémentaires qui permettent l'exécution d'une instruction
 - Ordres à tous les organes du microprocesseur
 - UT (Unité de traitement) \leftrightarrow UAL (Unité Arithmétique et Logique) \leftrightarrow Unité de calcul:
 - > Réalisation des calculs
 - > Résultats dans l'accumulateur

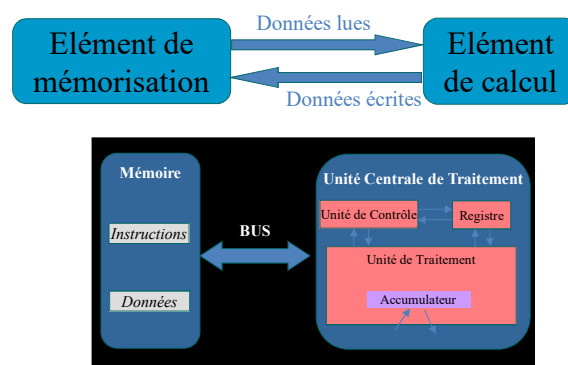


6

Modèle de microprocesseur

► R1.03: Introduction à l'architecture des ordinateurs

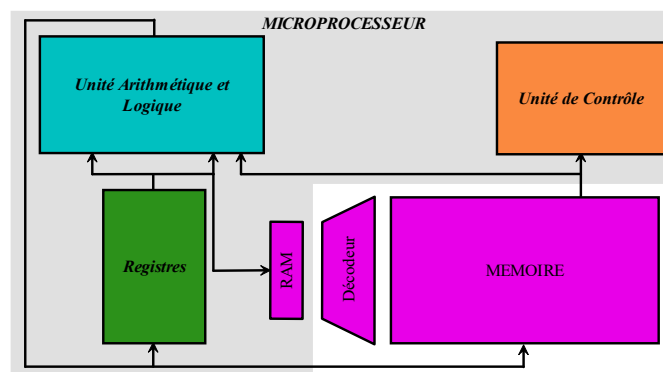
- Modèle de Von Neumann



7

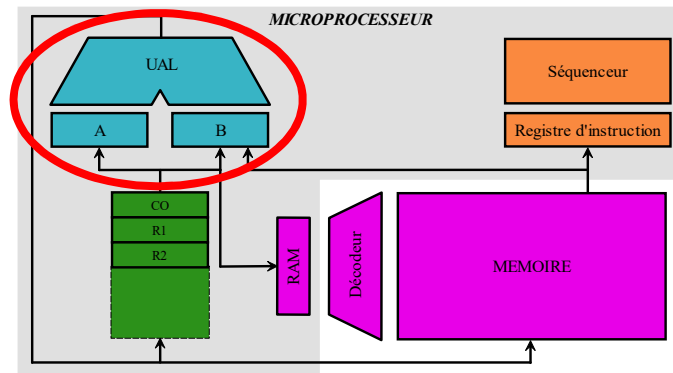
Modèle de microprocesseur

► R1.03: Introduction à l'architecture des ordinateurs



8

Modèle de microprocesseur



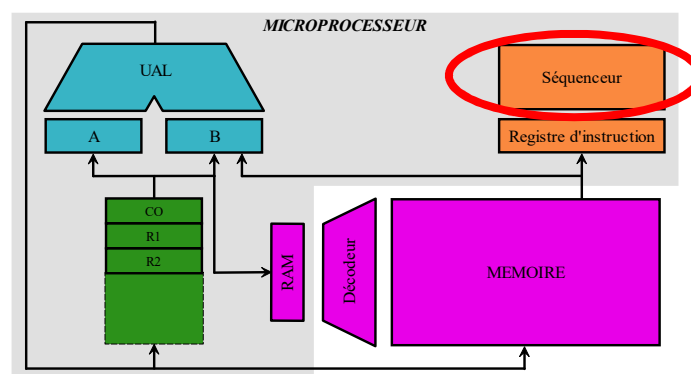
► UAL (Unité Arithmétique et Logique)

- Réalisation des calculs
- Résultats dans les accumulateurs
- A et B registres d'entrée



9

Modèle de microprocesseur



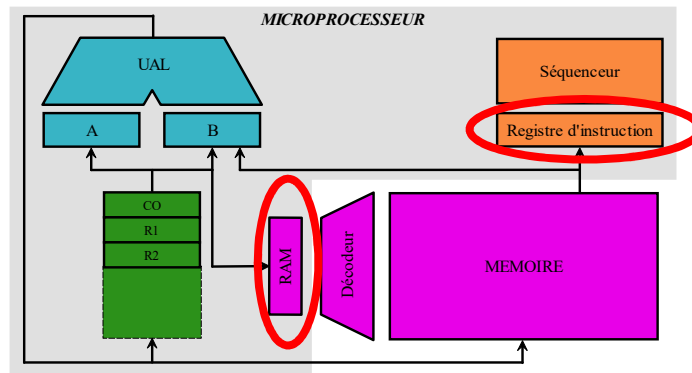
► UC (Unité de contrôle):

- Séquenceur: description en séquence des opérations élémentaires qui permettent l'exécution d'une instruction
- Câblé ou microprogrammé
- Ordres à tous les organes du microprocesseur



10

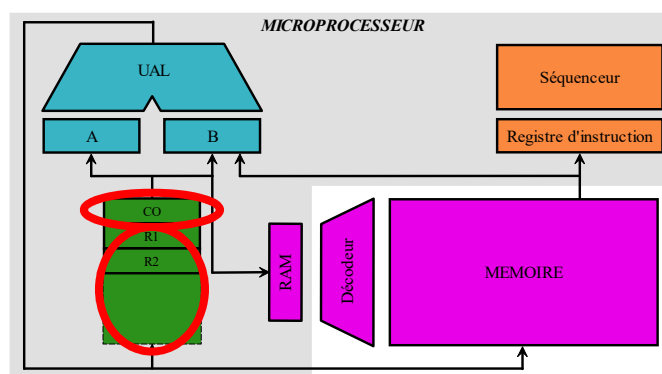
Modèle de microprocesseur



- ▶ **Registre d'instructions (RI):**
Instruction en cours d'exécution
- ▶ **Registre d'adresses mémoire (RAM)**
 - Adresse mémoire de la donnée à laquelle accéder

11

Modèle de microprocesseur

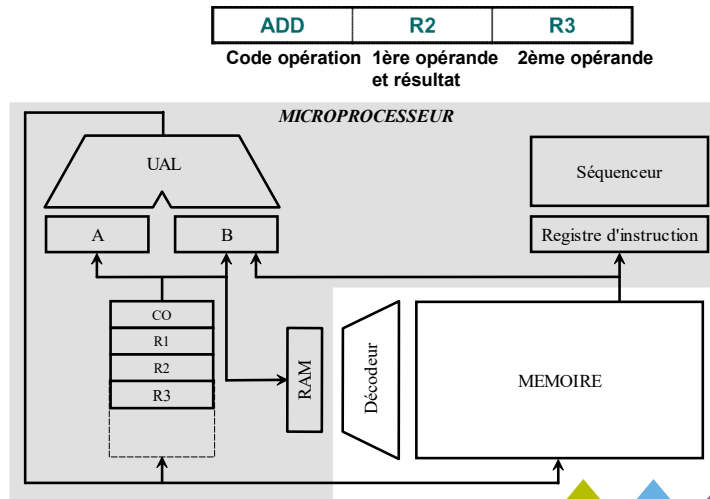


- ▶ **Compteur Ordinal (CO):**
 - Adresse de la prochaine instruction à exécuter
- ▶ **Registres de données (Ri):**
 - Stockage des informations à stocker ou extraire de la mémoire centrale

12

Modèle de microprocesseur

► Exécution de l'instruction: $R2 \leftarrow R2 + R3$



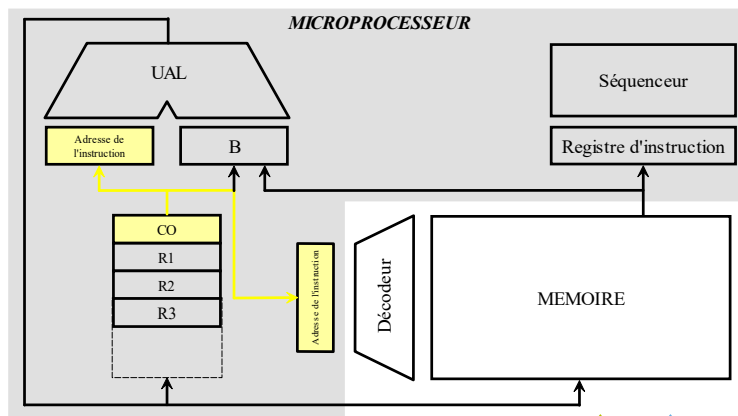
13

Modèle de microprocesseur

► Boucle d'exécution: étapes permettant de réaliser l'instruction



1. Compteur Ordinal: $CO \rightarrow A$ et $CO \rightarrow RAM$

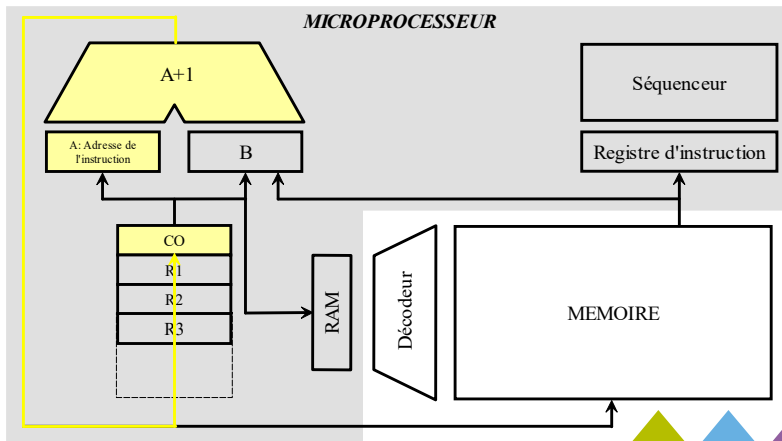


14

Modèle de microprocesseur

- Boucle d'exécution: étapes permettant de réaliser l'instruction

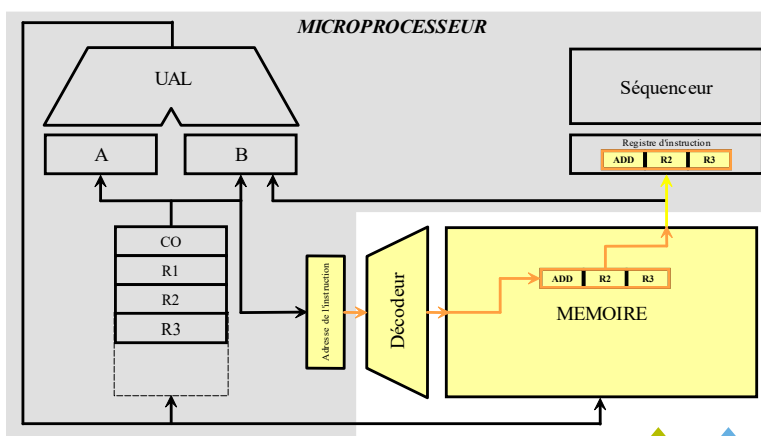
2. UAL incrémente le registre A (CO) et le replace dans CO



Modèle de microprocesseur

- Boucle d'exécution: étapes permettant de réaliser l'instruction

3. Lecture en mémoire de l'instruction **ADD R2 R3** et chargement dans le registre d'instruction **RI**

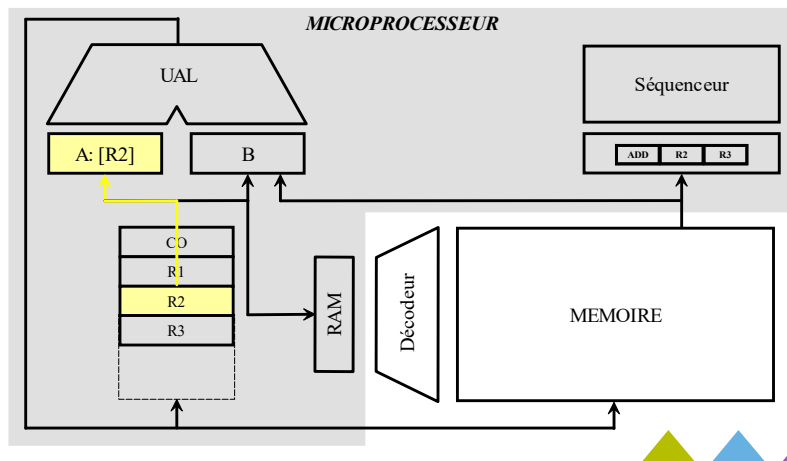


Modèle de microprocesseur

- Boucle d'exécution: étapes permettant de réaliser l'instruction

4. $R2 \rightarrow A$

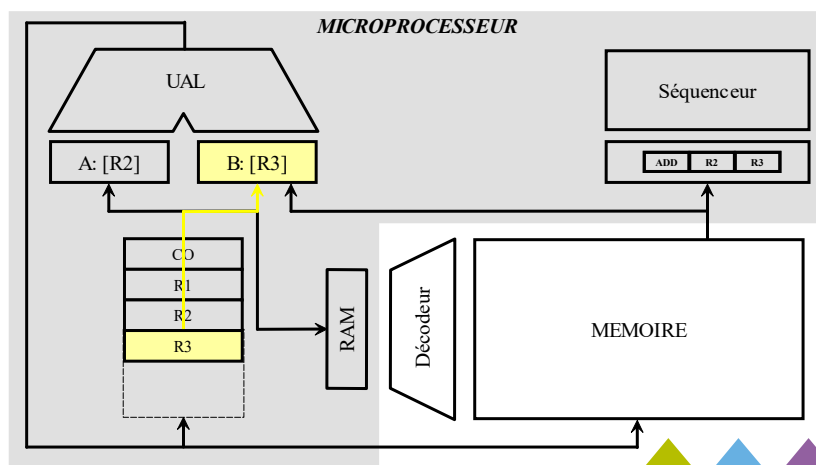
notation: [R2] contenu de R2



Modèle de microprocesseur

- Boucle d'exécution: étapes permettant de réaliser l'instruction

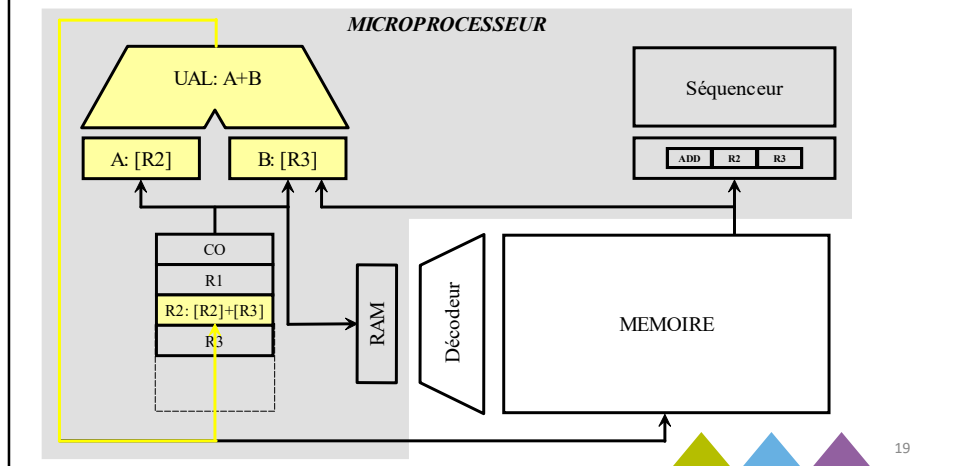
5. $R3 \rightarrow B$



Modèle de microprocesseur

- Boucle d'exécution: étapes permettant de réaliser l'instruction

6. UAL calcule $R2+R3$ et résultat placé dans R2



19

Modèle de microprocesseur

- Boucle d'exécution:

Permet le passage à l'instruction suivante

Indépendants de l'instruction

1. $CO \rightarrow A$ et $CO \rightarrow RAM$
2. **Incrémentation du CO:**
UAL incrémente le registre A (CO) et le replace dans CO
3. Lecture en mémoire de l'instruction (ici ADD R2 R3) et chargement dans le registre d'instruction RI

4. $R2 \rightarrow A$
5. $R3 \rightarrow B$
6. UAL calcule $R2+R3$ et résultat placé dans R2

Dépendant de l'instruction: orchestration par le séquenceur

20

Plan du cours

- ▶ Introduction
- ▶ Modèle de microprocesseur
- ▶ **Programmation en assembleur**
- ▶ Tests
- ▶ Entrées Sorties
- ▶ Pointeurs
- ▶ Procédures et fonction

21

Programmation en assembleur

- ▶ Le μp vu par le programmeur
 - μp : exécute des instructions
 - Jeu d'instruction

> Transferts	→	Mémoire
> Traitement	→	UT=UAL
> E/S	→	Unité d'échange
> Ruptures de séquences	→	UC
 - Utilisation de registres comme opérandes à ces instructions \Rightarrow accès plus rapide qu'en mémoire
 - > RISC (*reduced instruction-set computer*) : registres généraux
 - > CISC (*complex instruction-set computer*) : registres spécialisés
 - **Registres inaccessibles directement au programmeur: CO, RI, RAM**

22

Programmation en assembleur

► L'UE vue par le programmeur

- UE = registres accessibles par les instructions d'E/S
- Lecture des registres d'état → ce que fait ou a fait l'UE
- Ecriture dans les registres de commande de l'UE:
 - > paramétrer le fonctionnement de l'UE
 - > commander le fonctionnement de l'UE (faire faire)
- Registres de données de l'UE: échange d'information avec les périphériques

23

Programmation en assembleur

► Programmation: suite d'instructions

- Instruction=

code de l'opération + désignation des opérandes

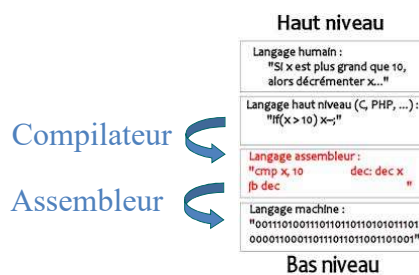
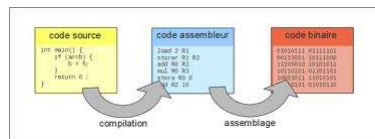
LD R0,4
- Instructions connues par le µp:
 - > en binaire
 - > en assembleur:
 - Code opération: mnémoniques (LD, ADD, JMP...)
 - Opérandes: noms de registres (R5...), variables

24

Programmation en assembleur

► Passage du langage aux instructions du μp :

- Langage de programmation: un compilateur
- Langage machine: un assembleur



25

Programmation en assembleur

► Description des opérandes

- par registre:
 - > opérande dans un registre
 - > décrite par le registre la contenant

LD R0,4
- immédiat: valeur contenue dans l'instruction
- direct:
 - > opérande en mémoire
 - > décrite par son adresse en mémoire (opérationnelle: adop): nom de variable pour le codeur mais adresse pour le microprocesseur

LD R0,adop
- indirect:
 - > opérande en mémoire
 - > décrite par le registre ou la variable contenant son adresse (*pointeur*)

LD R0,pt
- indirect avec déplacement:
 - > opérande en mémoire
 - > décrite par le registre ou la variable contenant une adresse et un déplacement à partir de cette adresse

LD R0, R2+2

26

Programmation en assembleur

► Description des opérandes

- indirect avec, en post ou pré, une incrémentation ou une décrémentation:
 - > opérande en mémoire
 - > décrite par le registre ou la variable contenant une adresse et une incrémentation ou décrémentation soit de l'adresse soit de la variable
- avec un segment \Rightarrow protection des données
- une combinaison de tout cela:
 - > Segment + indirect + déplacement + post incrément
 - > ...

27

Programmation en assembleur

► Instructions en langage machine:

- Syntaxe:

[etiquette:] **COP** [OP1[,OP2]] ;commentaire

28

Programmation en assembleur

► Différents types d'instruction en langage machine:

- Déplacement d'information:
 - > De mémoire à mémoire
 - > De mémoire à registre
 - > De registre à registre
- Traitements:
 - > Arithmétiques
 - > Logiques
 - > Comparaisons
 - > Décalages
- Traitements spécifiques à certains μp :
 - > Mathématiques
 - > Vectoriels
 - > Chaînes de caractères
 - > Traitement d'images

29

Programmation en assembleur

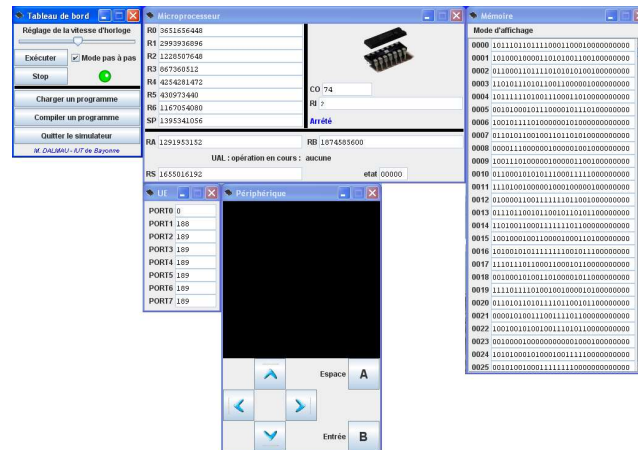
► Différents types d'instruction en langage machine:

- Rupture de séquences:
 - > Passage d'une instruction à une autre selon une condition
 - > Condition sur le registre d'état de l'Unité de Traitement (UAL)
 - > Description de l'instruction destinataire: étiquette (adresse)
- Contrôle:
 - > Appel, retour de procédures
 - > Manipulation de pile
 - > Gestion des interruptions

30

Programmation en assembleur

► Exemple de μ p : simulateur de Marc Dalmau

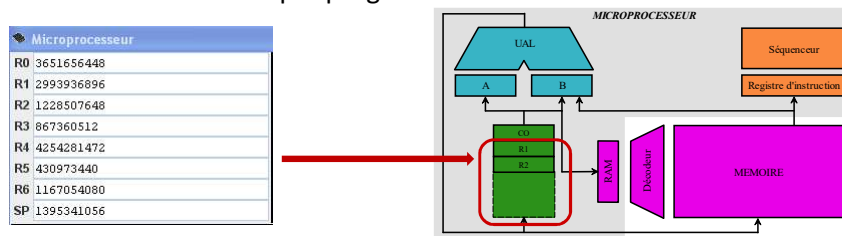


31

Programmation en assembleur

► Exemple de μ p : simulateur de Marc Dalmau

- Mémoire: mots de 32 bits
- Registres:
 - > Accessibles par programmeur:



- R0 à R6: 7 registres d'usage général
- SP: pointeur de pile (Stack Pointer)

32

Programmation en assembleur

► Exemple de μp : simulateur de Marc Dalmau

- Non accessibles par programmeur:



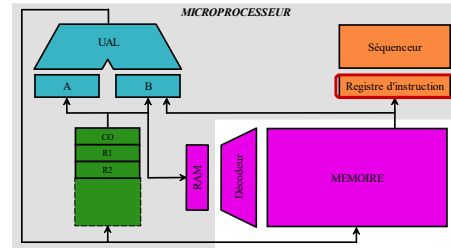
> CO

> RI

RA 1291953152	RB 1874585600
UAL : opération en cours : aucune	
RS 1655016192	etat 00000

> Registre d'état

> RA, RB, RS: registres de l'UT (UAL)



33

Programmation en assembleur

► Exemple de μp : simulateur de Marc Dalmau

- UT:
 - Opérations de type arithmétique, logique et décalage uniquement
 - Entiers naturels, entiers relatifs en complément à 2
- Exemple: -7 en complément à 2
 - 7: 0111
 - Complément: 1000
 - Complément +1 → complément à 2 soit -7: 1001

34

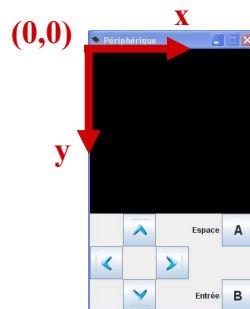
Programmation en assembleur

► Exemple de μp : simulateur de Marc Dalmau

- UE:

- > Gestion:

- Clavier: 6 touches + 1 souris
 - Écran graphique 256x256 en couleur



35

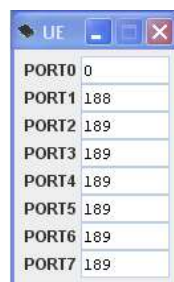
Programmation en assembleur

► Exemple de μp : simulateur de Marc Dalmau

- UE:

- > 8 registres appelés ports:

- Port 0: touches du clavier et souris
 - Port 1 à 5: écran graphique
 - Port 6: coordonnée x de la souris
 - Port 7: coordonnée y de la souris



36

Programmation en assembleur

► Programme en langage machine:

- Définition
 - > variables : adresse en mémoire + taille
 - > instructions : COP + opérandes
- 3 zones en mémoire:
 - > Code
 - > Variables
 - > Pile: utilisée pour les procédures
- Organisations possibles des 3 zones mémoires:
 - > mélangées
 - > par segments si le μp les gère \Rightarrow protection

37

Programmation en assembleur

► Programme en langage machine:

- Squelette d'un programme pour simulateur:

.DATA

déclaration des variables et constantes

.CODE

écriture du code

.STACK

réserve de place pour la pile

38

Programmation en assembleur

► Programme en langage machine:

- Déclaration des variables et constantes

> Définition:

- Réserve de place en mémoire pour une variable
- Attribution d'un nom

> Absence de typage:

- Connu seulement par programmeur
- **Au programmeur à effectuer les bons traitements**

39

Programmation en assembleur

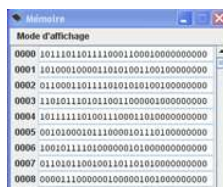
► Programme en langage machine:

- Déclaration des variables et constantes

> Manipulation des informations par leur nom = adresse des variables

- Dans le programme: nom des variables, STACK
- Microprocesseur: remplace nom des variables par adresse des variables ou de la pile
- Possibilité de manipuler toute la mémoire: variables, instructions

– **Responsabilité du programmeur**



40

Programmation en assembleur

Exemple:

```

34 ;*****
35 ;CODE PROPREMENT DIT*****
36 ;*****
37 .CODE
38
39 ;initialisation du jeu-----
40 LEA SP, STACK
41 LD R0, %0000
42 OUT R0, 5
43
44 ;affichage du message de dpart--
45 LD R0, 70
46 LD R1, 100
47 LEA R2, messageDebut
48 CALL afficheTexte
49
50

```

Adresse de la pile = 436

Adresse de la variable messageDebut = 41

Mem(41) désigne:

- La case mémoire d'adresse 41
- Le contenu de cette case

41

Programmation en assembleur

► Programme en langage machine:

- Déclaration des variables et constantes

> Programme:

– .DATA

- Positionnement des variables en mémoire
- *Equivalence nom = adresse*

– .STACK

- Positionnement de la pile
- Réservation de place pour la pile

> Deux cas:

- Variables non initialisées
- Variables initialisées

42

Programmation en assembleur

► Programme en langage machine:

- Déclaration des variables et constantes

> Variables non initialisées:

	Nom	DSW	taille (en mots mémoire)
Ex:	NbrTour	DSW	1

> Variables initialisées:

	Nom	DW	valeur
Ex:	NbrTour	DW	1

Valeur: Convertie en binaire par le compilateur

> une variable = un mot mémoire (sauf pour chaîne)

43

Programmation en assembleur

► Programme en langage machine:

- Types des valeurs:

> Entier décimal positif ou négatif: **1225 ou -6**

- de 0 à 4 294 967 295 ($2^{32}-1$)
- ou de -2 147 483 647 à 2 147 483 647 ($2^{31}-1$)
- Valeur sur 32 bits en complément à 2

> Entier hexadécimal: **\$1a ou \$ffa3**
valeur sur 32 bits (8 digits hexadécimaux maximum)

Rappel sur le codage hexadécimal

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

44

Programmation en assembleur

► Programme en langage machine:

- Valeur binaire: **%10111010**
valeur sur 32 bits (32 chiffres binaires maximum)
- Caractère ASCII: **'v'**
> code ASCII du caractère étendu à 32 bits
> ' et ; inutilisables car délimitent les caractères et les commentaires
- Chaîne de caractères : **« egun on »**
un code ASCII étendu à 32 bits par mot mémoire
un mot mémoire pour chaque caractère

45

Programmation en assembleur

► Programme en langage machine:

- Exemple de déclaration:

var1	DW	12	var 1 occupe 1 mot qui contient 12
car	DW	'!'	1 mot qui contient le code ASCII de '!'
var2	DW	-1	1 mot qui contient -1
var3	DSW	12	12 mots non initialisés
var4	DW	"ABC"	3 mots dont le 1er contient le code ASCII de 'A', le deuxième celui de 'B' et le dernier celui de 'C'.
var5	DW	11	2 mots initialisés à 11 pour le premier
	DW	22	et 22 pour le second
var6	DW	"je"	3 mots dont le 1er contient le code ASCII de 'j', le 2ème celui de 'e' et le
	DW	0	dernier la valeur 0.

46

Fin du premier cours

47

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code

> Désignation des opérandes

- Immédiat : valeur **Val**
- Registre : nom du registre **RG**
ex: R0 ou r1
- Direct : nom de la variable **Var**
 - ex: var1
 - Nom commence par une lettre
 - délimiteurs (' " ; , []), mot clé STACK, nom de registres interdits
 - Casse (majuscules ≠ minuscules)
 - Accent possible.
- Indirect : **[RG+d]**
 - Registre contient l'adresse de l'opérande: exemple [R1+2]
 - déplacement positif: $0 \leq d \leq 1023 = 2^{10}-1$
 - adresse dans le registre: 10 bits de faible poids (sur 32)

48

Programmation en assembleur

► Programme en langage machine:

- Attention:

> Dans les exemples de déclaration (Var1, var2,): variables sur 1, 2 ou x mots mémoires

> Dans la description des instructions de ce cours:

- **Var**

- nom générique d'une variable servant d'opérande

- **uniquement sur 1 mot mémoire**

- Var ≠ chaîne

- Var ≠ tableau

49

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code

- > **Déplacement de données**

- **LD dest, source** ; écriture dans un registre

- chargement $\text{dest} \leftarrow \text{source}$
 - destination: RG ou [RG]
 - source: Val, Var, RG, [RG], [RG+d]

- **ST source, Var** ; lecture d'un registre

- chargement $\text{Var} \leftarrow \text{source}$
 - source: RG, [RG]

- **SWP oper**

- échange les 16 bits de fort poids et les 16 bits de faible poids
 - oper: Var, RG, [RG], [RG+d]

- **LEA dest, var**

- chargement $\text{dest} \leftarrow \text{adresse de la variable}$
 - dest: RG, [RG]
 - initialisation de pointeur (SP)

- **Adresse du premier élément de la variable**

50

Programmation en assembleur

► Programme en langage machine:

- Exemples d'écriture du code

Déplacement de données avec LD

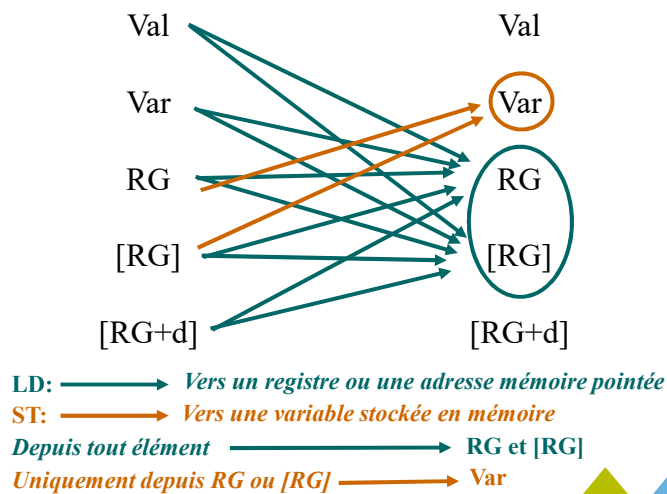
```
> LD  R0,r1          ; R0 ← R1
> LD  R1,$ff03       ; R1 ← FF03 en hexadécimal
> LD  R2,'A'         ; R2 ← code ASCII de 'A'
> LD  R0,var1        ; R0 ← contenu de var1
> ST  R2,var2        ; var2 ← contenu de R2
> LD  R0,[SP]        ; R0 ← contenu de la mémoire à
                    ; l'adresse contenue dans SP
> LD  [R1],12        ; en mémoire à l'adresse contenue dans R1
                    ; ← 12
> LD  [r1],var        ; en mémoire à l'adresse contenue dans R1
                    ; ← contenu de var
> LD  R0,[R0]        ; R0 ← contenu de la mémoire à l'adresse
                    ; contenue dans R0
```

51

Programmation en assembleur

► Programme en langage machine:

- Différence entre ST et LD pour les déplacements:



52

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code:

> Instructions arithmétiques

– Indicateurs de débordements des entiers naturels et relatifs du registre d'état de l'UT (sauf NEG)

– Type: **XXX op1, op2**

- $op1 \leftarrow op1 \text{ XXX } op2$
- XXX = **ADD**, **SUB**, **MUL** (entiers relatifs), **MULU** (entiers naturels), **DIV**, **DIVU**
- op1: RG, [RG]
- op2: Val, Var, RG, [RG], [RG+d]

– Type: **YYY oper**

- $oper \leftarrow \text{résultat de YYY sur oper}$
- YYY = **INC**, **DEC**, **NEG** (-oper)
- oper: Var, RG, [RG], [RG+d]

53

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code:

> Instructions de rupture de séquence: branchements

- **JMP etiq**: branchement inconditionnel
- **Bxx etiq**: branchement conditionnel si la condition est vérifiée

Condition	Comparaison d'entiers naturels	Comparaison d'entiers relatifs
$op1 = op2$	BEQ	BEQ
$op1 \neq op2$	BNE	BNE
$op1 < op2$	BLTU	BLT
$op1 \leq op2$	BLEU	BLE
$op1 > op2$	BGTU	BGT
$op1 \geq op2$	BGEU	BGE
débordement	BDEBU	BDEB

54

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code:
 - > Instructions de rupture de séquence: branchements
 - Mnémotechnique
 - **BEQ**: Branchement si **E**Qual
 - **BNE**: Branchement si **N**ot **E**qual
 - **BLT**: Branchement si **L**esser **T**han
 - **BLE**: Branchement si **L**esser or **E**qual
 - **BGT**: Branchement si **b**i**G**ger **T**han
 - **BGE**: Branchement si **b**i**G**ger or **E**qual
 - **BDEB**: Branchement si **DE**Bordement
 - **XXXU** (**BLTU**, **BLEU**, **BGTU**, **BGEU**, **BDEBU**):
XXX **U**nsigned

55

Programmation en assembleur

► Programme en langage machine:

- Ecriture du code:
 - > Instructions de rupture de séquence: branchements
 - **Etiquette**
 - désigne l'instruction résultante
 - mot alphanumérique
 - sans délimiteur du langage: ' " ; , []
 - sans espace
 - caractères accentués possibles
 - terminé par :
 - Ne pas la faire suivre d'un espace
 - Exemple:

Début:	LD	R0, val	; valeur inconnue
	CMP	R0, 0	
	BNE	debut	
	INC	R0	
 - Après le CMP, CO incrémenté donc:
 - si R0 ≠ 0 retour à étiquette Début
 - si R0 = 0 passage à la suite INC R0

56

Programmation en assembleur

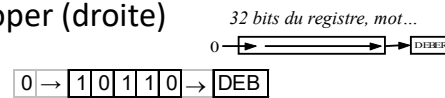
► Programme en langage machine:

- Ecriture du code:

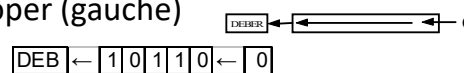
> Instructions de décalage

- Oper: Var, RG, [RG], [RG+d]
- DEBER: indicateur de débordement des entiers naturels du registre d'état de l'UT
- logique:

- **SHR** oper (droite)



- **SHL** oper (gauche)



57

Programmation en assembleur

► Programme en langage machine:

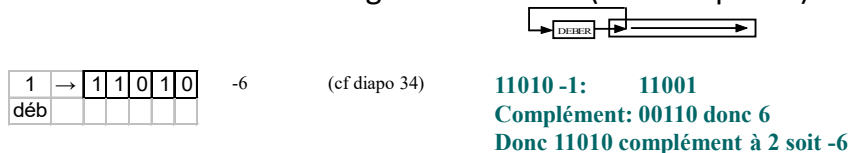
- Ecriture du code:

> Instructions de décalage

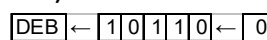
- Arithmétique

- **SAR** oper (droite)

conserve le signe du nombre (division par 2^n)



- **SAL** oper (gauche)
- (identique SHL)

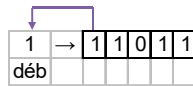


58

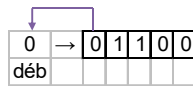
Programmation en assembleur

► Programme en langage machine:

> Exemple: décalage arithmétique droite



11011 -1: 11010
 Complément: 00101 donc 5
 Donc 11011 complément à 2 soit -5



01100 positif donc en binaire naturel: 12

59

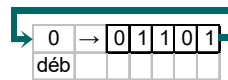
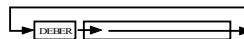
Programmation en assembleur

► Programme en langage machine:

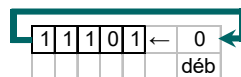
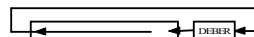
- Ecriture du code:

> Cyclique

– ROR oper (droite)



– ROL oper (gauche)



60

Programmation en assembleur

► Structures de contrôle en langage machine

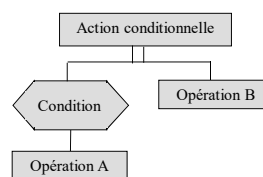
- PB: Traduction des structures de contrôle en langage machine
- Traduites avec les opérations de rupture de séquence conditionnelles ou inconditionnelles
- Réalisation de la traduction par le compilateur
- Contraintes possibles imposées par les langages sur la machine :
 - > vérification des types des variables
 - > interdiction de la modification du code
 - > limitation des structures de contrôle
- Machine seule: pas de contraintes, tout est possible

61

Programmation en assembleur

► Structures de contrôle en langage machine

- Action conditionnelle



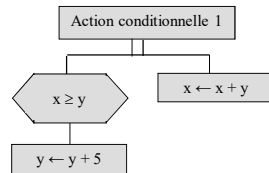
- > Évaluation de la condition
- > Si faux: branchement à instruction associée (B)
- > Si vrai :
 - effectuer l'opération associée à condition vérifiée (A)
 - puis effectuer l'opération non conditionnelle (B)

62

Programmation en assembleur

► Structures de contrôle en langage machine

• Exemple d'action conditionnelle



1. Évaluation de la condition $x \geq y$
2. Si faux branchement à instruction associée $x \leftarrow x + y$
3. Si vrai:
 1. effectuer l'opération associée à condition vérifiée $y \leftarrow y + 5$
 2. Puis effectuer l'opération non conditionnelle $x \leftarrow x + y$

Si condition de l'algorithme vraie (\geq) Si condition fausse ($<$)

```

suite:  CMP    x,y
        BLTU   suite      ; si <
        ADD    y,5        ; si ≥
        ADD    x,y
  
```

63

Programmation en assembleur

► Structures de contrôle en langage machine

• Remarque:

> limitations de notre simulateur: contraintes sur les types d'opérandes des instructions

```

suite:  CMP    x,y      Variables ≠ RG ou [RG]
        BLTU   suite      ; si <
        ADD    y,5        ; si ≥
        ADD    x,y
  
```

> Nouvelle version du code

```

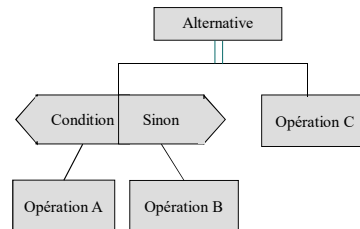
suite:  LD      R0,x
        LD      R1,y
        CMP    R0,y
        BLTU   suite      ; si <
        ADD    R1,5        ; si ≥
        ADD    R0,R1
        ST      R0,x
        ST      R1,y
  
```

64

Programmation en assembleur

► Structures de contrôle en langage machine

- Alternative:



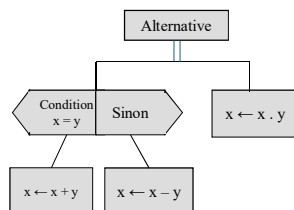
- > Évaluation de la condition
- > Si faux branchement à instruction associée au sinon (B)
- > Effectuer l'opération associée à condition vérifiée (A) et branchement après opération pour faux (à C)
- > Effectuer l'opération (C)

65

Programmation en assembleur

► Structures de contrôle en langage machine

- Exemple d'alternative:



1. Évaluation de la condition $x = y$
2. Si faux branchement à instruction associée $x \leftarrow x - y$
3. Si vrai:
 1. effectuer l'opération associée à condition vérifiée $x \leftarrow x + y$
 2. brancher après opération pour faux avant $x \leftarrow x . y$
4. Dans tous les cas, effectuer l'opération non conditionnelle $x \leftarrow x . y$

Si condition vraie (=)		Si condition fausse (≠)		
	CMP	x,y		
	BNE	diff		
	ADD	x,y		; si ≠
	JMP	suite		; si =
diff:	SUB	x,y		
suite:	MULU	x,y		

66

Programmation en assembleur

► Structures de contrôle en langage machine

- Remarque:
 - > limitations de notre simulateur: contraintes sur les types d'opérandes des instructions

	CMP	x y	Variables ≠ RG ou [RG]
	BNE	diff	; si ≠
	ADD	x y	; si =
	JMP	suite	
diff:	SUB	x y	
suite:	MULU	x y	

– Nouvelle version du code

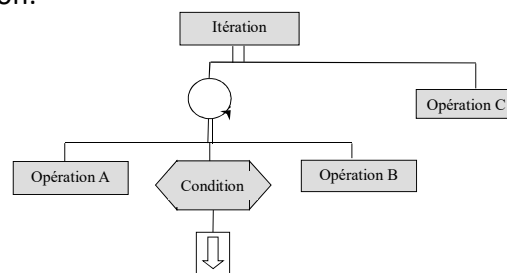
	LD	R0,x	
	CMP	R0,y	
	BNE	diff	; si ≠
	ADD	R0,y	; si =
	JMP	suite	
diff:	SUB	R0,y	
suite:	MULU	R0,y	
	ST	R0,x	

67

Programmation en assembleur

► Structures de contrôle en langage machine

- Itération:



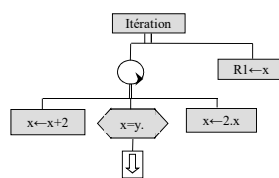
- > Première opération de la boucle (A)
- > Évaluation de la condition de sortie
- > Branchement de sortie si condition de sortie vérifiée (à C)
- > Autrement:
 - Autres opérations (B)
 - Branchement à boucle (A)
- > Si sortie, opération associée à la sortie de la boucle (C)

68

Programmation en assembleur

► Structures de contrôle en langage machine

- Exemple d'itération:



1. Première opération de la boucle $x \leftarrow x+2$
2. Evaluation de la condition de sortie $x = y$
3. Branchement si condition de sortie vérifiée (vers $R1 \leftarrow x$)
4. Autrement:
 1. Autres opérations $x \leftarrow 2.x$
 2. Branchement à boucle (vers $x \leftarrow x+2$)
5. Opération associée à la sortie de la boucle $R1 \leftarrow x$

Si condition vraie (=) Si condition fausse (≠)

```

boucle:  ADD    x,2
         CMP    x,y
         BEQ    sortie      ; si =
         MULU   x,2          ; si ≠
         JMP    boucle
sortie:  LD     R1,x
  
```

69

Programmation en assembleur

► Structures de contrôle en langage machine

- limitations de notre simulateur: contraintes sur les types d'opérandes des instructions

```

boucle:  ADD    x,2
         CMP    x,y
         BEQ    sortie      ; si =
         MULU   x,2          ; si ≠
         JMP    boucle
sortie:  LD     R1,x
  
```

Variables ≠ RG ou [RG]

- Nouvelle version du code

```

boucle:  LD      R0,x
         ADD     R0,2
         CMP     R0,y
         BEQ     sortie      ; si =
         MULU    R0,2         ; si ≠
         JMP     boucle
sortie:  LD      R1, R0
         ST      R0,x
  
```

70

Programmation en assembleur

► A partir d'un algorithme

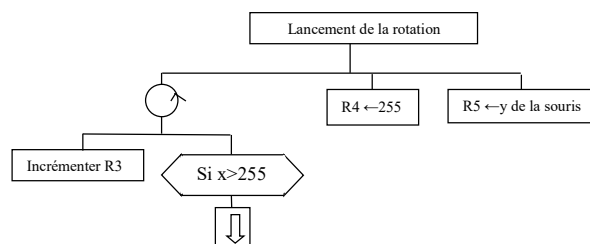
1. Identifier les instructions nécessaires à la réalisation de l'algorithme
2. Identifier les contraintes syntaxiques liées à ces instructions (types d'opérandes)
3. Rajouter les lignes de code permettant de respecter ces contraintes (transfert de données de ou vers des registres)

71

Programmation en assembleur

► Exercice d'application 1 (Hand spinner) :

- Ecrire un programme qui:
 - > initialise R3 à 0
 - > réalise l'algorithme suivant sachant:
 - R1: coordonnée en x de la souris
 - R2: coordonnée en y de la souris



72

Programmation en assembleur

► Exercice d'application 1 (Hand spinner):



Fin du deuxième cours



Plan du cours

- ▶ Introduction
- ▶ Modèle de microprocesseur
- ▶ Programmation en assembleur
- ▶ Tests
- ▶ Entrées Sorties
- ▶ Pointeurs
- ▶ Procédures et fonction

75

Tests

- ▶ Définition:
“Processus d’analyse d’un programme avec l’intention de détecter des anomalies dans le but de le valider”
(Fornari IRIT)
- ▶ Tester un logiciel = valider sa conformité par rapport à des exigences.
- ▶ Types de test:
 - Fonctionnels: conformité à la spécification
 - Non-fonctionnels: conformité en matière de configuration, de compatibilité, de documentation,
 - Structurels: codage correct ?
- ▶ Coût du test: 30 à 40% du coût de développement au minimum

76

Tests

► Principes de base:

- Indépendance: tester par quelqu'un d'autre que le programmeur
- Paranoïa: un test doit retourner erreur par défaut (partir de l'hypothèse qu'il y a une erreur) et forcer à retourner ok
- **Prédiction: définir les sorties/résultats attendus à partir des spécifications et avant l'exécution des tests.**
- Vérification: inspection minutieuse des résultats de chaque test.
- Robustesse: tests avec des jeux valides, invalides et incohérents
- Complétude: vérifier ce que fait le programme lorsqu'il n'est pas supposé le faire (jeux incohérents de données)

77

Tests

► Correction de bug:

- vérifier que le test est bien correct
- vérifier que le problème n'est pas déjà répertorié
- établir un rapport de bug
 - > donner un synopsis succinct et précis
 - > donner une description claire, avec tous les détails de reproduction du bug
 - > si possible, essayer de réduire l'exemple.

78

Tests

► Test et correction de bug avec notre simulateur:

- Coder
- Définir le test validant le programme
- Compiler le programme
- L'exécuter en vitesse rapide sans pas à pas
- Voir si le test est réussi
- Sinon, l'exécuter en pas à pas à vitesse rapide:
 - > Avant de réaliser une instruction, définir ce qui en est attendu
 - > Exécuter cette instruction et vérifier le résultat obtenu
- Si l'erreur n'est pas identifiée avant la fin du programme, exécuter en pas à pas à vitesse lente:
 - > Avant chaque instruction, définir les lectures et écritures qu'elle devrait réaliser
 - > Exécuter en pas à pas à vitesse lente et vérifier chaque lecture (en vert dans le simulateur) ou écriture (en rouge)

79

Plan du cours

- Introduction
- Modèle de microprocesseur
- Programmation en assembleur
- Tests
- Entrées Sorties
- Pointeurs
- Procédures et fonction

80

Entrées Sorties

- ▶ Unité d'échange (U.E.): constituée de contrôleurs de périphériques
- ▶ Contrôleur:
 - pilote un ou plusieurs périphériques
 - deux visions:
 - > ensemble de registres appelés PORTS qui sont accessibles par des instructions spéciales
 - > mots en mémoire avec une adresse
- ▶ UE:
 - registres de contrôles: pilotage des périphériques
 - registres d'état: surveillance des périphériques
 - registres de données: communication avec les périphériques

81

Entrées Sorties

- ▶ UE du Simulateur:
 - 8 registres de 8 bits correspondant aux n° de ports 0 à 7
 - UE:
 - > connaître l'état des touches
 - > dessiner ou écrire dans la fenêtre graphique du périphérique
 - > détecter les mouvements et les clics de souris dans la zone graphique de l'écran du périphérique
 - 2 instructions:
 - > **IN oper, port** ; oper ← octet du registre de l'UE désigné par son n°
 - > **OUT oper, port** ; octet du registre de l'UE désigné par son n° ← oper
 - > Oper: RG ou [RG] sur 8 bits (plus faible poids)
 - > Port: entier naturel désignant le n° de port de l'UE

82

Entrées Sorties

► Clavier:

- Port 0 = registre d'état et de données

Etat du clavier		Numéro de la touche					
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

► Etat du clavier:

- Action qui a eu lieu sur le clavier depuis la dernière fois que le port 0 a été lu
- Lecture du port 0 \Rightarrow remise à 0 (de B7 – B0)

00	aucune action
11	une touche appuyée
10	une touche relâchée

- B7: une action? B6: laquelle?

83

Entrées Sorties

► Simulateur:

- Clavier:

> N° de la touche sur B5 – B0:



- Souris:

> Port 0 pour l'état:

- appui sur bouton souris = appui touche n°7
- remise à 0 après lecture de son contenu

> Port 6 et 7 pour les données:

- en permanence les coordonnées de la souris
- en x: port 6, en y: port 7

84

Entrées Sorties

► Simulateur:

- Ecran:

- > Ports 1 à 5

- Ports 1 à 4:

- registres de données
 - paramètres de l'opération à exécuter

- Port 5:

- registre de commande
 - opération à exécuter

Couleur du tracé				Commande à exécuter			
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

85

Entrées Sorties

► Simulateur:

- Ecran:

- > Commandes graphiques (voir guide de programmation):

- 0000 : effacer l'écran
 - 0001 : tracer le point
 - Port1: x
 - Port2: y
 - Couleur: 4 bits *Couleur de tracé*.
 - tracer une ligne (ports 1, 2, 3 et 4)
 - tracer un rectangle (ports 1, 2, 3 et 4)
 - tracer un ovale (ports 1, 2, 3 et 4)
 - tracer un rectangle plein (ports 1, 2, 3 et 4)
 - tracer un ovale plein (ports 1, 2, 3 et 4)
 - écrire un caractère ASCII (ports 1, 2, 3)

86

Entrées Sorties

► Simulateur:

- Méthode de tracé

1. Paramétrage du tracé sur les ports 1, 2, 3, 4 selon la définition des commandes graphiques
2. Envoi de la commande de tracé sur le port 5 :
 - Couleur sur les bits de fort poids
 - Figure sur les bits de faible poids

87

Entrées Sorties

► Simulateur:

- Exemple de tracé d'un rectangle bleu clair à l'écran:
 - > Port 1: coordonnée en x du coin supérieur gauche
 - > Port 2: coordonnée en y du coin supérieur gauche
 - > Port 3: largeur
 - > Port 4 : hauteur
 - > Port 5: couleur 0101 et commande graphique 0011

	LD	R0,200	
	OUT	R0,1 ;x	
	LD	R0,100	
	OUT	R0,2 ; y	
	LD	R0,20 ;	
	OUT	R0,3 ;largeur	
	LD	R0,80	
	OUT	R0,4 ;hauteur	
Envoi de la commande	LD	r0,\$53 ; rectangle bleu soit 0101 0011	Paramétrage
	OUT	R0,5	

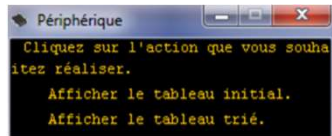
88

Entrées Sorties

► Exercice d'application 2: Entrées Sorties

L'étude porte sur un programme qui:

- affiche un message demandant à l'utilisateur de choisir entre l'affichage d'un tableau simple et l'affichage d'un tableau trié



- détermine où l'utilisateur clique pour afficher le tableau demandé:
 - > le programme attend que l'utilisateur clique
 - > si l'utilisateur clique au-dessus du premier choix (au-dessus de $y1$) ou en-dessous du second choix (en $y5$ ou en-dessous de $y5$), le programme attend que l'utilisateur clique à nouveau.
 - > si l'utilisateur clique sur le premier message ($y1 \leq y < y3$), le programme va à l'instruction dont l'étiquette est *choix1* pour afficher le tableau initial.
 - > Si l'utilisateur clique sur le second message ($y3 \leq y < y5$), le programme va à l'instruction dont l'étiquette est *choix2* pour afficher le tableau trié.

89

Entrées Sorties

► Exercice d'application 2: Entrées Sorties

- Question 1: Ecrire le code qui permet de savoir où l'utilisateur a cliqué

90

Plan du cours

- ▶ Introduction
- ▶ Modèle de microprocesseur
- ▶ Programmation en assembleur
- ▶ Tests
- ▶ Entrées Sorties
- ▶ **Pointeurs**
- ▶ Procédures et fonction

91

Pointeurs

▶ Accès aux variables:

- Directement: nom \Leftrightarrow adresse

> Exemple 1:

```
LD R0,25      ; R0 ← 25
ST R0,total   ; total ← R0
```

} donc 25 dans total
Accès direct et par registre

- Indirectement:

> adresse \Leftrightarrow **pointeur**

> [Reg]: Reg = registre contenant l'adresse de la variable

> Exemple 2:

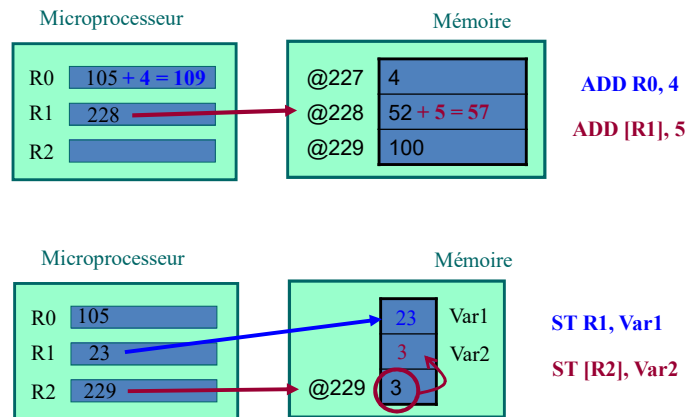
```
LEA R1,total   ; R1 ← @ de total
LD [R1],25     ; variable dont l'@ est dans R1 ← 25
```

} donc 25 dans total comme exemple 1
Mais accès indirect

92

Pointeurs

► Différence entre les deux modes d'accès:



93

Pointeurs

► Exemple 3:

copie de var 1 dans var2

- Première version:

```
LD    R0, var1    ; R0 ← var1
ST    R0, var2    ; var2 ← R0
```

} accès direct et par registre

- Seconde version:

```
LEA R1, var1      ; R1 ← @ de var1
ST   [R1], var2   ; var2 ← variable dont l'@ est dans R1
```

} accès indirect

- Troisième version

```
LEA R1, var1      ; R1 ← @ de var1
LEA R2, var2      ; R2 ← @ de var2
LD   [R2], [R1]   ; var dt @ ds R2 ← var dt @ ds R1
```

} accès indirect

94

Pointeurs

► Intérêt des pointeurs: accès à des variables composées:

- Enregistrements
- Tableaux

► Exemples:

- Coordonnées d'un point (2 entiers)

> Déclaration:

Coord DSW 2 ; taille=2 mots mémoire (2x32 bits)

> Initialisation à (10,40):

LEA R0,coord ; adresse de coord dans R0

LD [R0],10 ; variable dont @ dans R0 ← 10

LD [R0+1],40 ; variable dont @-1 dans R0 ← 40

95

Pointeurs

► Attention à la notation:

[R0] variable dont l'adresse est contenue dans R0



[R0+1] variable dont l'adresse est celle contenue dans R0 à laquelle on rajoute 1



NB: il aurait été plus logique de noter [R0]+1 mais le code aurait été illisible

96

Pointeurs

► Systèmes d'exploitation

- Pointeurs utilisés par systèmes d'exploitation:
 - > Processus
 - > Fichiers
 - > Mémoire
- Utilisation de structures de données très complexes
 - > Ex: tableau de pointeurs vers d'autres tableaux
 -

97

Pointeurs

► Tableaux utilisés par les systèmes d'exploitations:

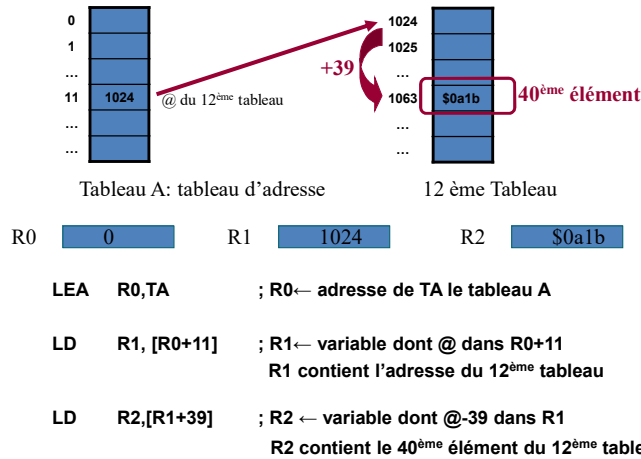
- Élément:
 - > variables de type enregistrement
 - > certains champs de l'enregistrement: pointeurs
- Exemple d'élément de tableau pour un processus:
 - > n° du processus: PID Process Identifier
 - > n° de l'utilisateur: UID User Identifier
 - > adresse de l'instruction à exécuter
 - > tableau de n éléments (sauvegarde registres)
- Accès à cet élément par un pointeur:
 - > n+4 pour accéder à l'élément suivant
 - > accès par déplacement : +1 UID, +2 adresse

98

Pointeurs

► Exemple d'utilisation de tableaux:

Comment récupérer le 40ème élément du 12ème tableau ?

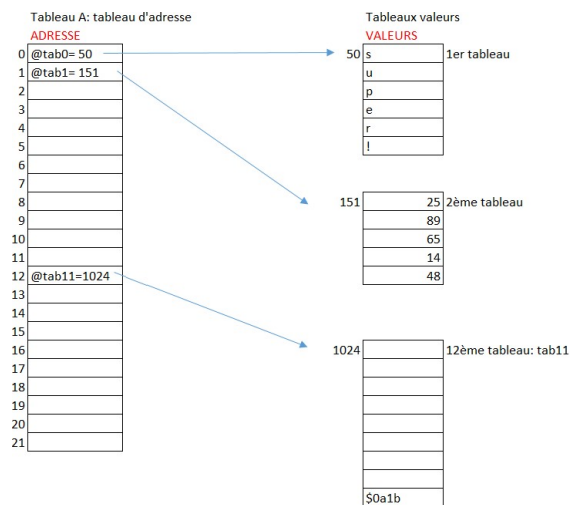


99

Pointeurs

► Exemple d'utilisation de tableaux:

Comment récupérer le 40ème élément du 12ème tableau ?



100

Pointeurs

► EXERCICE 3 : Calcul de moyenne

- Objectif:

- > écrire un programme en assembleur qui calcule la moyenne des nombres entrés dans un tableau.
- > tableau = suite de mots mémoires qui se terminent par un mot contenant la valeur 0.

1. Partie du code dans .DATA :

- > déclarer la variable moyenne appelée moy
- > initialiser à 0

101

Pointeurs

► EXERCICE 3 : Calcul de moyenne

2. Partie du code dans .DATA :

- > déclarer la variable tableau T
- > initialiser à 3-6-7-4-2 (+ 0 marquant la fin du tableau)

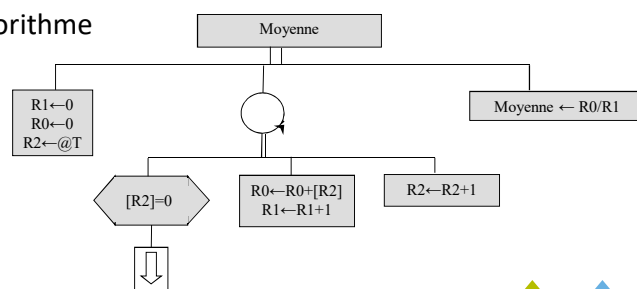
102

Pointeurs

► EXERCICE 3 : Calcul de moyenne

3. Partie du code dans le .CODE: calcul de la moyenne (pas de procédure)
 - > R0: valeur courante de la somme des nombres
 - > R1: nombre courant de chiffres pris en compte
 - > R2: pointeur sur le tableau

Algorithme



103

Pointeurs

► EXERCICE 3: Calcul de moyenne

3. Partie du code dans le .CODE: calcul de la moyenne (pas de procédure)

104

Fin du troisième cours

105

Plan du cours

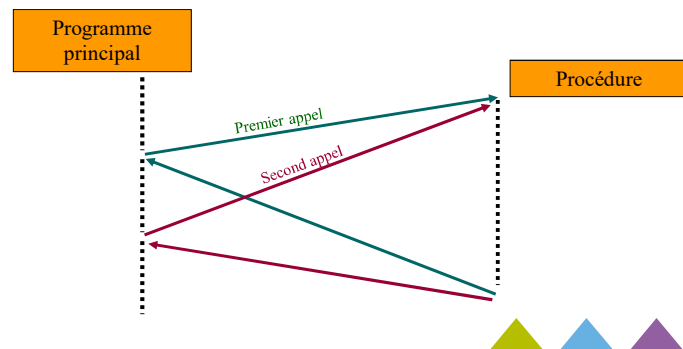
- ▶ Introduction
- ▶ Modèle de microprocesseur
- ▶ Programmation en assembleur
- ▶ Tests
- ▶ Entrées Sorties
- ▶ Pointeurs
- ▶ Procédures et fonction

106

Procédures et fonctions

► Définitions

- Principe:
 - > Morceau de code exécutable à la demande
 - > Transmission de paramètres
 - > Restitution d'une valeur si besoin
- Intérêt: réutilisation d'un morceau de code à différents endroits du programme

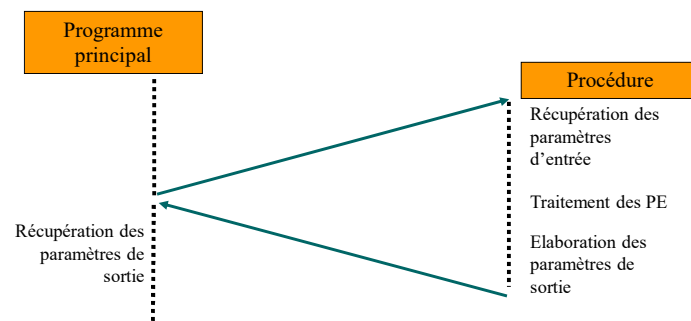


107

Procédures et fonctions

► Gestion du contexte

- Gestion des paramètres:

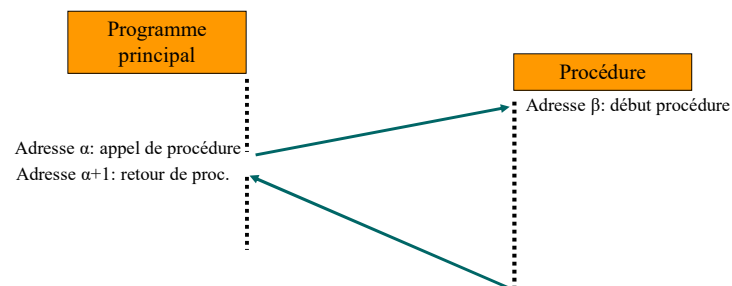


108

Procédures et fonctions

► Gestion du contexte

- Appel de procédure:
 - > quitter programme en cours
 - > exécuter procédure
 - > revenir au programme là où on l'a laissé
 - ⇒ nécessité de conserver l'adresse de retour
- Exemple:

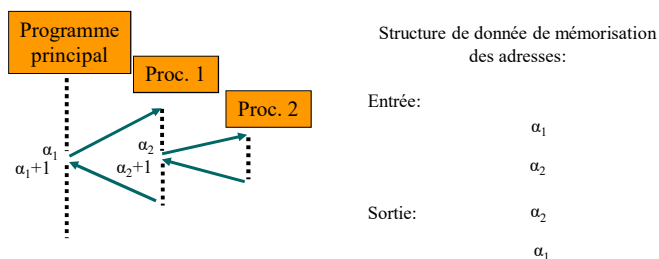


109

Procédures et fonctions

► Gestion du contexte

- Solutions:
 - > Registre spécial contenant l'adresse de retour
 - ⇒ pb: impossible d'appeler la procédure dans la procédure
 - > Mémorisation de l'adresse de retour dans une structure de données adéquate: *laquelle?*



- > Retour à l'adresse la plus récemment mémorisée
- > Last in, First out LIFO PILE

110

Procédures et fonctions

► Gestion du contexte

- Gestion de pile:
 - > Registre spécial **SP**:
 - SP: Sommet de pile
 - 1ère entrée occupée
 - > Réserve de place mémoire pour la pile:
 - directive: **.STACK taille**
 - fin de code
 - > Initialisation du registre SP sur le début de la pile:
 - instruction: **LEA SP,STACK**
 - en début de code
 - STACK en majuscule

111

Procédures et fonctions

► Programme

- Squelette du programme:
 - **DATA**
déclaration des variables et constantes
 - **CODE** ; CO initialisé sur l'instruction qui suit .CODE
LEA SP,STACK
écriture du code
 - **STACK taille**
réserve de place pour la pile

Rq: possibilité d'utiliser une instruction **HLT** pour arrêter le processeur en fin de programme

112

Procédures et fonctions

► Programme

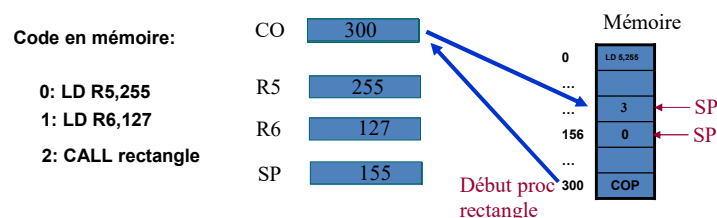
- Certains processeurs: Utilisation de deux registres pour désigner le début et la fin de la pile
 - > Possibilité de détecter les débordements
 - > En veillant à ne pas:
 - Dépiler une pile déjà vide
 - Empiler dans une pile déjà pleine
 - > Possibilité d'arrêter les programmes en cours comme dans certains systèmes d'exploitation
- Notre simulateur: bonne gestion de la pile de la responsabilité du programmeur

113

Procédures et fonctions

► Instructions dédiées

- Instruction de type rupture de séquence pour les procédures:
 - > Appel: **CALL étiquette**
 - empile l'adresse de retour contenue dans le compteur ordinal
 - $SP \leftarrow SP - 1$
 - $[SP] \leftarrow CO$
 - Branche à la procédure \approx JMP étiquette (@ de la procédure dans le CO)
 - > Exemple:



114

Procédures et fonctions

► Instructions dédiées

- Instruction de type rupture de séquence pour les procédures:

> Retour: **RET**

– dépile l'adresse de retour

- $CO \leftarrow [SP]$

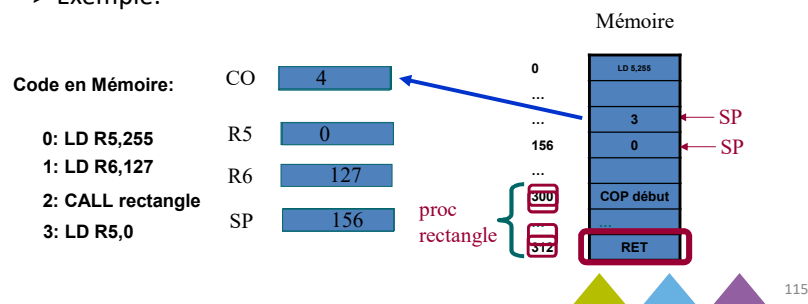
- $SP \leftarrow SP+1$

– A n'utiliser qu'une seule fois dans une procédure sinon:

- 1ère fois: @ de retour dans CO

- 2ème fois: valeur inconnue placée dans CO

> Exemple:



Procédures et fonctions

► Méthode de programmation

- Inconvénients liés aux procédures:

> double accès aux registres de la machine et aux registres de variables globales:

– depuis le programme

– depuis les procédures

> danger: modification par la procédure d'un registre utilisé par le programme qui l'appelle

⇒ perte de données

- Exemple:

```

Boucle:  LD    R0,20    ; R0 ← 20 nombre d'itération
        CALL  calcul  ; appel de la procédure calcul qui
                       ; utilise R0 (R0 ← 356)

        DEC   R0       ; R0 ← R0-1=355
        CMP   R0,0     ; comparaison de 355 au lieu de 20 avec 0
        BNE   boucle   ; si R0 ≠ 0 on boucle
  
```

Conclusion: perte du nombre de répétition de la boucle (infini)

116

Procédures et fonctions

► Méthode de programmation

- Solution 1:

- > utiliser des registres différents pour les procédures et pour le programme principal

- > problématique:

- nombre limité de registres
 - spécialisation de certains registres pour instructions
 - difficulté alors de réutilisation: appel de procédures écrites par d'autres (primitives du système d'exploitation)

117

Procédures et fonctions

► Méthode de programmation

- Solution 2:

- > Sauvegarder les registres en début de procédure et les restituer à la fin

- > Hypothèse : sauvegarde dans des variables

- Trop de variables
 - Impossible de s'assurer que chaque procédure utilise des variables différentes
 - Interdit la récursivité
- ⇒ Impossible

- > Conclusion:

Utilisation de la pile pour sauvegarder les registres

- début de procédure: empilage des registres
 - fin de procédure: dépilage des registres

118

Procédures et fonctions

► Méthode de programmation

• Instructions:

> Empiler:

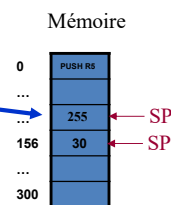
- **PUSH oper** ; empiler oper
- Réalise:
 - $SP \leftarrow SP-1$
 - $[SP] \leftarrow oper$
- oper: Var, RG, [RG], [RG+d]
- Exemple

Procédure en mémoire:

0: PUSH R5

1: LD R6,127

CO	2
R5	255
SP	155



119

Procédures et fonctions

► Méthode de programmation

• Instructions:

> Dépiler:

- **PULL oper** ; dépiler oper
- Réalise:
 - $oper \leftarrow [SP]$
 - $SP \leftarrow SP+1$
- oper: Var, RG, [RG], [RG+d]

Procédure en mémoire:

0: PUSH R5

1: LD R6,127

...

7: ADD R5,2

8: PULL R5

9:

CO	10
R5	255
SP	156



120

Procédures et fonctions

► Méthode de programmation

- Nécessité de prévoir une taille de pile suffisante:
 - > Stocker les adresses de retour
 - > Stocker les sauvegardes de registres
- Pour certains microprocesseurs:
 - > existence d'instructions pour empiler et dépiler tous les registres
 - > en pratique inutile: aucun procédure n'utilise tous les registres
- Variable locale des langages de programmation:
 - > Utilisation de la pile
 - > Mécanisme:
 - début de procédure: réservation d'espace mémoire dans la pile pour ces variables
 - fin de procédure: libération de cet espace

121

Procédures et fonctions

► Passage de paramètres

- Passage de paramètres à une procédure dans les langages habituels:
 - > par **valeur** :
 - en entrée
 - pas de possibilité de modification
 - > par **référence**:
 - en entrée et sortie
 - possibilité de modification
- Ici étude du passage de paramètres en langage machine

122

Procédures et fonctions

► Passage de paramètres

- Passage de paramètres par **valeur** :

> avant l'appel de la procédure: valeur placée dans un **registre**

– utilisation du registre par la procédure: modification possible du registre mais valeur du paramètre inchangée

– exemple

```
LD    R0,param    ; R0 ← param
CALL  calcul      ; paramètre d'entrée dans R0
```

> avant l'appel de la procédure: valeur placée dans la **pile**

– utilisation de la pile par la procédure: modification possible de la pile mais valeur du paramètre inchangée

– Exemple

```
PUSH  param    ; SP ← SP-1 et [SP] ← param
CALL  calcul    ; paramètre d'entrée dans pile
```

123

Procédures et fonctions

► Passage de paramètres

- Passage de paramètres par **référence** :

> avant l'appel de la procédure: adresse du paramètre placée dans un **registre**

– procédure: accès au paramètre par indirection

– exemple:

```
- LEA  R0, param
- CALL calcul      ; adresse du paramètre dans R0
```

> avant l'appel de la procédure: adresse placée dans la **pile**

– utilisation de la pile par la procédure

– exemple:

```
- LEA  R0, param
- PUSH R0
- CALL calcul      ; adresse du paramètre dans pile via R0
```

124

Procédures et fonctions

► Passage de paramètres

- Passage de paramètres à une procédure en langage machine:

> par registre:

- plus rapide
- limité en nombre et en taille des registres

> par pile:

- pas de limite en nombre et en taille
- utilisé par les compilateurs

125

Procédures et fonctions

► Passage de paramètres

- Retour de la valeur d'une fonction: retour d'une unique valeur

⇒ différent du cas des procédures

- > utilisation d'un registre: plus simple
- > pile: plus compliqué, pas rentable pour une seule valeur

- Utilisation de l'instruction: **RET** ou **RET n**

> retour avec vidage de la pile de n mots

> RET: (exemple diapo suivante)

- $CO \leftarrow [SP]$
- $SP \leftarrow SP+1$

> RET n: (exemple diapo 112)

- $CO \leftarrow [SP]$
- $SP \leftarrow SP+n+1$

126

Procédures et fonctions

► Passage de paramètres

- Exemple pour RET:

> CO \leftarrow [SP]

> SP \leftarrow SP+1

Procédure en mémoire:

0: PUSH R5

1: LD R6,127

...

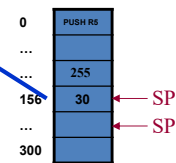
7: ADD R5,2

8: PULL R5

9: RET

CO	30
R5	255
SP	157

Mémoire



127

Procédures et fonctions

► Ecriture du programme

- Quelle est l'écriture finale d'une procédure/fonction?

> Exemple de la fonction calcul (a,b): $a \leftarrow a+2.b$

> Etude pour:

- Passage des paramètres par valeur
 - Par registre
 - Par la pile
- Passage des paramètres par références
 - Par registre
 - Par la pile

> Objectif: définir la structure d'écriture d'une procédure/fonction

128

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par valeur

- > Par registre:

- Paramètres:

- a dans R0
 - b dans R1
 - Valeur retournée par R0

- Programme:

```
LD    R0, a    ; 1er paramètre
LD    R1, b    ; 2ème paramètre
CALL  calcul   ; appel de la fonction
```

- Procédure:

```
Calcul:  ADD    R0,R1
          ADD    R0,R1
          RET                    ; retour de sous-prog
```

Attention on verra plus loin que ce code n'est pas utilisable.

129

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par valeur

- > Par la pile:

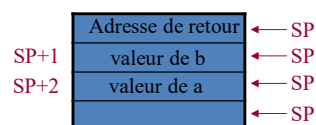
- Programme:

```
PUSH    a      ; 1er paramètre
PUSH    b      ; 2ème paramètre
CALL    calcul ; appel de la fonction
ST      R0,a
```

- Procédure:

```
Calcul: LD    R0,[SP+2]    ; R0 ← a
          ADD    R0,[SP+1] ; ajout de b
          ADD    R0,[SP+1] ; ajout de b
          RET
```

- Etat de la pile:



Attention on verra plus loin que ce code n'est pas utilisable.

130

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par valeur

> Par la pile:

- Inconvénient:

- Pile non vide en fin de procédure (2 paramètres)
- Remplissage progressif de la pile
- Danger de débordement

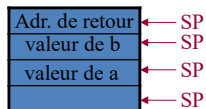
- Solution:

- Programme appelant fait ADD SP,2 après l'appel de la procédure (vidage de la pile)
- Utilisation par certains processeurs de RET n

- Nouveau code de la procédure

Calcul:	LD	R0,[SP+2]	; R0 ← a
	ADD	R0,[SP+1]	; ajout de b
	ADD	R0,[SP+1]	

RET 2



Pile vide au moment du retour au programme

Attention on verra plus loin que ce code n'est pas utilisable.

131

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par référence

> Par registre:

- Paramètres:

- Adresse de a dans R0 (par référence)
- b dans R1 (par valeur)
- Résultat dans a (donc adresse de a en donnée)

- Programme:

LEA	R0, a	; adresse de a dans R0
LD	R1, b	; 2ème paramètre
CALL	calcul	; appel de la fonction

- Procédure:

```

Calcul:  ADD    [R0],R1 ; a ← a + b
         ADD    [R0],R1
         RET      ; retour de sous-prog

```

Attention on verra plus loin que ce code n'est pas utilisable.

132

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par référence

> Par la pile:

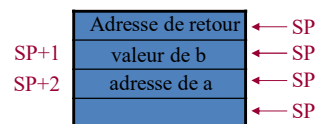
– Programme:

```
LEA    R0,a    ; R0 ← adresse de a
PUSH   R0      ; 1er paramètre
PUSH   b       ; 2ème paramètre
CALL   calcul  ; appel de la fonction
```

– Procédure:

```
Calcul: LD    R0,[SP+2]    ; R0 ← adresse de a
        ADD   [R0],[SP+1]  ; ajout de b
        ADD   [R0],[SP+1]  ; ajout de b
        RET   2
```

– Etat de la pile:



– Attention on verra plus loin que ce code n'est pas utilisable.

133

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par référence

> Par la pile:

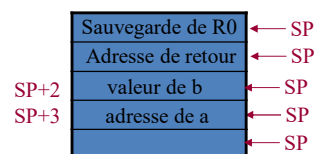
– Pas de retour de valeur par la procédure: modification directe de a

⇒ Aucune raison de modifier R0 car ne sert pas à renvoyer le résultat au programme principal

⇒ Nécessité de sauvegarder R0

– Nouveau code de la procédure:

```
Calcul: PUSH   R0
        LD     R0,[SP+3]
        ADD    [R0],[SP+2]
        ADD    [R0],[SP+2]
        PULL   R0
        RET    2
```



Code définitif: méthode que nous utiliserons

134

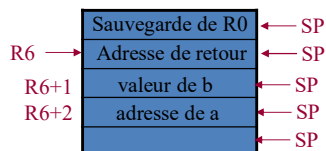
Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par référence

> Par la pile:

- Problème: décalage des références dans la pile à chaque sauvegarde d'un nouveau registre
- Solution:
 - Accès à la pile par une copie de SP et non SP
 - Copie de SP faite en début de procédure avant la sauvegarde des registres
- Nouveau code de la procédure:



```
Calcul: LD    R6, SP
        PUSH  R0
        LD    R0,[R6+2]
        ADD   [R0],[R6+1]
        ADD   [R0],[R6+1]
        PULL  R0
        RET   2
```

Nous n'utiliserons pas cette technique.

135

Procédures et fonctions

► Ecriture du programme

- Passage des paramètres par référence

> Par la pile:

- Problème: modification de R6 sans l'avoir sauvegardé avant
 - Solution:
 - Sauvegarde de R6 au début de la procédure
- ⇒ décalage de 1 dans la pile ∇ le nombre de registres sauvegardés dans la pile

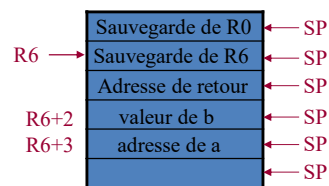
136

Procédures et fonctions

► Ecriture du programme

> Forme finale de l'écriture d'une fonction/procédure avec passage des paramètres par référence:

```
Calcul:  PUSH  R6
         LD   R6, SP
         PUSH R0
         LD   R0,[R6+3]
         ADD  [R0],[R6+2]
         ADD  [R0],[R6+2]
         PULL R0
         PULL R6
         RET  2
```



Nous n'utiliserons pas cette technique.

137

Procédures et fonctions

► Ecriture du programme

- En TD et TP: Utilisation d'une version simplifiée du passage des paramètres par référence par la pile

Pas de copie du pointeur de pile

```
.DATA    a DW 5           ;déclaration des variables et constantes
         b DW 6
```

```
.CODE    ;écriture du code
```

```
LEA SP,STACK
```

```
LEA R0,a
```

```
PUSH R0
```

```
PUSH b
```

```
CALL Calcul ;appel de la procédure
```

```
HLT
```

```
;code de la procédure
```

```
Calcul:  PUSH R1
```

```
LD R1,[SP+3] ; R1= @ de a
```

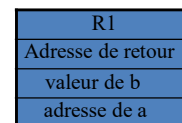
```
ADD [R1],[SP+2] ; a'=a+b
```

```
ADD [R1],[SP+2] ; a''= a'+b= 2a+b
```

```
PULL R1
```

```
RET2
```

```
.STACK 4
```



138

Procédures et fonctions

► Exercice d'application 4: fonction XOR

- Objectif:

- > Calcul du résultat du XOR, OU EXCLUSIF, entre deux nombres binaires.
- > Utilisation d'une procédure.

1. Partie du code:

- > déclarer les variables A, B et f
- > les initialiser à 0.

139

Procédures et fonctions

► Exercice d'application 4: fonction XOR

2. Code de la procédure lorsque le passage des valeurs des paramètres se fait grâce aux registres suivants :

R0 pour A

R1 pour B

R3 pour f

140

Procédures et fonctions

- Exercice d'application 4: fonction XOR
- 3. Programme appelant cette procédure.



Procédures et fonctions

- Exercice d'application 4: fonction XOR
- 3. Intégralité du code du programme lorsque le passage des paramètres se fait par la pile.



Place à la programmation en assembleur...

