TD1: le M99 (l'ordinateur en papier) – Correction

ENS-Rennes

Objectifs pédagogiques:

- Comprendre le principe de base d'un processeur informatique
- Comprendre l'encodage des opcodes dans la mémoire, et le cycle fetch/decode/execute (ex 1, 4)
- Modifier un programme en assembleur, entrevoir le travail d'un compilateur (ex 2, 4, 5)
- Voir en pratique une implémentation possible du mécanisme de pile d'exécution (ex 3)

Les exercices indiqués comme optionnels (ex 4 et 5) ne sont là que pour votre culture et pour amuser les élèves qui le souhaitent. Ne les faites pas si ce n'est pas un jeu pour vous. Ils ne seront pas corrigés en classe, et ne donneront lieu à aucune évaluation.
Des questions issues d'exercices non-optionnels peuvent être posées en examen.

Le M99 (page séparée) est doté de 100 cases mémoires (la grille en haut), et d'un processeur (en bas).

La mémoire est composée de 100 cases mémoires de 3 chiffres (valeur de 000 à 999). Chaque case mémoire peut être désigné par une adresse codée sur deux chiffres. Cette mémoire va contenir données et instructions.

Unité Arithmétique et Logique (ALU). Le processeur dispose de trois registres directement utilisables: A et B sont utilisés pour les opérandes des opérations tandis que R est pour le résultat. Ces registres sont de 3 chiffres, mais contrairement à la mémoire, ils ont un signe. Leurs valeurs sont donc comprises entre -999 et 999.

Le processeur dispose aussi d'un quatrième registre nommé PC (Program Counter). C'est le pointeur d'instruction, contenant l'adresse mémoire de la prochaine instruction à exécuter. Lorsqu'on utilise le M99, on peut noter le numéro de l'instruction à exécuter dans la case prévue à cette effet, mais en pratique, il est plus simple de le matérialiser avec un "pion" sur les cases de la grille mémoire, ou de suivre avec son doigt.

Unité de commande. Elle pilote l'ordinateur. Son cycle de fonctionnement comporte 3 étapes :

1. *(fetch)* charger l'instruction depuis la case mémoire pointée par PC dans la case `instr` (sous le PC).
2. *(decode)* identifier l'opération à réaliser à partir des 3 chiffres la codant et grâce au pense-bête de droite.
3. *(execute)* exécuter cette instruction, puis incrémenter ensuite le PC.

Premier jeu d'instructions. D'autres instructions seront rajoutées au fil des exercices.

Code	*Mnémonique*	*Instruction à réaliser*
0 x y	`STR xy`	Copie le contenu du registre R dans la case mémoire d'adresse *xy*
1 x y	`LDA xy`	Copie la case mémoire d'adresse *xy* dans le registre A
2 x y	`LDB xy`	Copie la case mémoire d'adresse *xy* dans le registre B
3 x y	`MOV x y`	Copie registre Rx dans Ry (R0: R; R1: A; R2: B)
4 0 0	`ADD`	Ajoute les valeurs des registres A et B, produit le résultat dans R
4 0 1	`SUB`	Soustrait la valeur du registre B à celle du registre A, produit le résultat dans R
4 x y	etc	(d'autres opérations arithmétiques et logiques peuvent être encodées ainsi)
5 x y	`JMP x y`	Branche (saute) en *xy* (PC reçoit la valeur *xy*)
6 x y	`JPP x y`	Branche en *xy* si la valeur du registre R est positive
7 x y	`JEQ x y`	Saute une case (PC := PC+1) si la valeur du registre R est égale à *xy*
8 x y	`JNE x y`	Saute une case (PC := PC+1) si la valeur du registre R est différent de *xy*

PC est incrémenté deux fois si la condition de JEQ ou JNE est vraie: par l'unité de commande et par l'opération. Cela signifie que le PC augmente au final de 2 si la condition est vérifiée.
On peut utiliser le mnemotechnique `DAT x y z` pour stocker une valeur numérique arbitraire *xyz* en mémoire. Les dépassements de capacité *(overflow)* sont gérés de manière cyclique: 999 + 1 = -999; -999 - 10 = 990. Lorsqu'on stocke une valeur négative d'un registre vers la mémoire, le signe est perdu.

Boot et arrêt. La machine démarre avec la valeur nulle comme pointeur d'instruction (`PC=0`) et elle s'arrête si le pointeur d'instruction vaut 99. On peut donc utiliser le mnémonique `HLT` comme synonyme de `JMP 99`.

En cas de faute (comme par exemple une division par zéro), le programme est interrompu (`PC := 99`).

Entrées/sorties Les entrées/sortries sont "mappées" en mémoire: Écrire à l'adresse mémoire 99 écrit sur le terminal, tandis que les valeurs saisies sur le terminal seront lues à cette adresse 99.

★ Exercice 1: Prise en main, opcode et cycle fetch/decode/execute.

▷ **Question 1:** Quelle valeur se trouve à la case 17 ? Quelle est l'instruction correspondante ?

Réponse

310, donc `MOV r1 r0`, donc `MOV A R`. On déplace le contenu du registre A dans le registre R.

Fin réponse

▷ **Question 2:** Que fait le programme chargé à l'adresse 0 ? Combien d'instructions sont exécutées ?

Réponse

Pour répondre, il faut appliquer le cycle fetch/decode/exec aux données qui sont dans les premières adresses de la mémoire. Traduire les valeurs numériques en mémoire est nécessaire.

```
@00: 110; LDA 10  // Charge le contenu de la case 10 dans le registre A
@01: 211; LDB 11  // Charge le contenu de la case 11 dans le registre B
@02: 401; SUB     // R := A - B
@03: 607; JPP 7   // Si R > 0 alors PC := 7
@04: 320; MOV B R // R := B
@05: 099; STR 99  // Copie R en 99, c'est-à-dire, affiche R à l'écran
@06: 599; JMP 99  // Arrête le programme
@07: 310; MOV A R // R := A
@08: 099; STR 99  // Copie R en 99, donc affiche R à l'écran
@09: 599; JMP 99  // Arrête le programme
@10: 123; DAT 123 // Utilisé seulement comme une donnée, sans signification
@11: 042; DAT 42  // Utilisé seulement comme une donnée, sans signification
```

Donc au final, ce programme affiche 123 (car $123 > 42$) en 10 instructions.

Fin réponse

▷ **Question 3:** Que fait le programme débutant à l'adresse 13?

Réponse

```
@13: 199; LDA 99  // Charge une entrée utilisateur dans A
@14: 310; MOV A R // R := A
@15: 010; STR 10  // Copie l'entrée utilisateur en 10
@16: 199; LDA 99  // Charge une entrée utilisateur dans A
@17: 310; MOV A R // R := A
@18: 011; STR 11  // Copie l'entrée utilisateur en 11
@19: 500; JMP 0   // Branche l'exécution sur 0, au début du programme précédent
```

Donc au final, ce programme demande deux entrées à l'utilisateur avant d'exécuter le programme précédent (qui affichera le plus grand d'entre eux).

Fin réponse

▷ **Question 4:** À quoi correspondent les cases 10 et 11 dans ces deux programmes?

Ce sont des données, il ne faut pas les interpréter comme des instructions. On ne peut pas faire la différence à priori en inspectant la case mémoire, mais on le voit quand on exécute le programme.

Fin réponse

▷ **Question 5:** Écrivez à partir de l'adresse 20 un programme affichant le minimum de deux entrées clavier.

Réponse

On n'a pas besoin d'écrire ce qu'on lit en mémoire puisqu'on l'utilise immédiatement.

```
@20: 199; LDA 99  // input A
@21: 299; LDB 99  // input B
@22: 401; SUB
@23: 627; JPP 27  // JMP 27 si R>0, ie si A>B
@24: 310; MOV A R // Copie A dans R
@25: 099; STR 99  // Affiche A
@26: 599; JMP 99  // Halt
@27: 320; MOV B R // Copie B dans R
@28: 099: STR 99  // Affichage B
@29: 599: JMP 99  // Halt
```

Fin réponse

★ Exercice 2: Premiers programmes en assembleur.

▷ **Question 1:** Que fait le programme débutant à l'adresse 40 (pour les entrées 5 et 2)?

Il calcule le produit des deux entrées et affiche le résultat.

Fin réponse

▷ **Question 2:** Raccourcissez ce programme pour l'écrire avec aussi peu d'instructions que possible.

Réponse

On peut passer à 21 cases en stockant x et y dans les cases 40 et 41 car on n'a plus besoin de ce code une fois qu'on l'a exécuté. Oui, c'est pas très propre, mais on est en assembleur, là.

Il faut ensuite déplacer les cases 61 et 62 (qui sont respectivement la valeur 1 et le résultat) dans les cases 59 et 60. Cette opération est plus simple en **nommant les variables**, c'est à dire en écrivant par exemple "LDB un" à la place de "LDB 61" dans le mnemotechnique la case 51. C'est ensuite le travail du compilateur de faire ce qu'il faut.

On peut même tomber à 19 cases en utilisant une ruse: au lieu d'énumérer les trois opérations nécessaires pour afficher le contenu de A avant de s'arrêter, on branche vers l'endroit du programme 1 qui fait cela (JMP 04).

Notez que réutiliser les bouts d'un autre programme en sautant au milieu de son code est considéré comme **très sale** par la plupart des programmeurs. En pratique on ne veut absolument jamais faire quelque chose d'aussi dangereux avec de vrais programmes car cela les rend difficile à lire et à comprendre. On ne fait pas des programmes que pour la machine, mais aussi (surtout) pour que d'autres humains les comprennent.

```
@56: 310: MOV A R     ->   @56: 504: JMP 04
@57: 099: STR 99
@58: 599: HLT
```

Fin réponse

▷ **Question 3:** Corrigez ce programme quand la seconde entrée vaut 0.

Réponse

En effet, notre programme calcule par exemple 5*0 = 5 car il ajoute x au résultat dans tous les cas. Pour corriger, il faut d'abord vérifier s'il y a besoin d'ajouter x et ensuite seulement le faire.

La première solution est d'ajouter un JMP juste avant la boucle pour entrer au bon endroit de la boucle (sur le décrément de y). Mais cela fait un code spagetti assez désagréable.

La seconde solution est de réécrire le corps de boucle pour faire le décrément du compteur avant l'addition au résultat, au prix de légères contortions pour sortir de la boucle au bon moment

```
@46: 162: LDA res    ->  @46: 162: LDA y
@47: 259: LDB x          @47: 261: LDB un
@48: 400: ADD            @48: 401: SUB
@49: 062: STR res        @49: 899: JPP fin // si y est passé sous 0, on arrête tout
@50: 160: LDA y          @50: 162: LDA res
@51: 261: LDB un         @51: 259: LDB x
@52: 401: SUB            @52: 400: ADD
@53: 060: STR y          @53: 062: STR res
@54: 646: JPP 46         @54: 546: JMP 46 // Retour début boucle
                         @55: 162: LDA res
                         @56: 504: JMP 04 // Utilise la fin du prog 1
```

Au final, les deux solutions sont assez difficiles à relire: soit on commence la première boucle en sautant au milieu, soit on sort de la dernière boucle en sautant depuis le milieu. C'est quand même plus simple d'utiliser un langage de haut niveau et un compilateur :)

Fin réponse

★ Exercice 3: Pile et fonctions.

Dans cet exercice, nous allons doter le M99 d'une pile d'exécution afin de permettre l'écriture de fonctions récursives. On ajoute pour cela deux nouveaux registres. r3 est nommé `SB` (Stack Base – initialisé à 98), tandis que r4 est nommé `RA` (Return Address – initialisé à 0). On ajoute également les opérations suivantes :

Code	*Mnémo*	*Instruction à réaliser*
48 x	`PSH x`	`mem(SB):=Rx; SB--` (Rx est écrit dans la case pointée par SB, qui est ensuite décrémenté)
49 x	`POP x`	`SB++; Rx:=mem(SB)` (SB est incrémenté puis le contenu de mem(SB) est écrit dans Rx)
9 xy	`CAL xy`	`RA:=PC+1; PC:=xy` (PC+1 est copié dans RA, puis l'exécution saute en xy)
409	`RET`	`PC:=RA` (on récupère dans RA l'adresse où reprendre l'exécution)

▷ **Question 1:** Que fait le code qui commence à l'adresse 64? Combien d'instructions sont exécutées?

Réponse

Ce code propose un corps général (cases 64 à 74) qui charge sur la pile 3 entiers lues au clavier, et une routine (cases 75 à 82) qui remplace les deux entiers au sommet de la pile par le maximum de ces deux entiers. Après avoir appelé la routine deux fois, le code général affiche le résultat au sommet de la pile et s'arrête. Le main contient 11 instructions et la routine exécute 6 instructions dans tous les cas. Au total, on exécute donc $11 + 6 + 6 = 23$ instructions.

```
@64: 199: LDA 99
@65: 481: PSH A  // Empile ce qu'on vient de lire
@66: 199: LDA 99
@67: 481: PSH A  // Empile ce qu'on vient de lire
@68: 199: LDA 99
@69: 481: PSH A  // Empile ce qu'on vient de lire
@70: 975: CALL @max // Appelle cette routine
@71: 975: CALL @max // Appelle cette routine
@72: 490: POP R // Dépile dans R
@73: 099: STR 99 // Affiche le contenu de R
@74: 599: JMP 99 == HALT
# Début de la routine
@75: 491: POP A
@76: 492: POP B
@77: 401: SUB (R:= A-B)
@78: 681: JPP (jump si R positif)
@79: 481: PSH A
@80: 409: RET
@81: 481: PSH B
@82: 409: RET
# Fin de la routine
```

Fin réponse

▷ **Question 2:** Que fait le code suivant si on le charge à partir de l'adresse 20. Le nombre hors de l'encadré est l'adresse de la case mémoire, tandis que celui entre parenthèses est la représentation numérique de l'instruction.

```
20 (924) CAL 24
21 (599) JMP 99
22 (042) DAT 42
23 (001) DAT 1
```

```
24 (122) LDA 22
25 (481) PSH A
26 (928) CAL 28
27 (409) RET
```

```
28 (491) POP A
29 (223) LDB 23
30 (401) SUB
31 (636) JPP 36
32 (484) PSH RA
33 (480) PSH R
34 (928) CAL 28
35 (494) POP RA
36 (409) RET
```

Réponse

Le code appelle une première routine définie entre 24 et 27. Celle-ci pousse 42 sur la pile, puis appelle une seconde routine définie entre 28 et 36, avant de retourner le contrôle au code appelant (c'est à dire en 21).

La seconde routine dépile un paramètre, et retranche 1 à sa valeur. Si c'est positif, on empile la valeur décrémentée puis on fait un appel récursif à la fonction. Pour assurer l'appel récursif, on empile d'abord l'adresse de retour avant le paramètre. À la fin de la routine, on dépile l'adresse de retour puis on l'utilise.

La ruse consistant à empiler l'ancienne adresse de retour permet à la seconde routine d'être appelable depuis une routine (comme c'est le cas ici). Au contraire, la première routine ne peut pas être utilisée récursivement.

Notez que cette fonction empile 42 fois l'adresse de retour "26". C'était une bonne idée de le charger en position 20 pour éviter les dépassements.

Fin réponse

▷ **Question 3:** Écrivez le code de la factorielle sous forme d'une fonction récursive. Vous pourrez soit modifier le programme déjà écrit, ou ajouter une opération `MUL` de code 402 pour stocker le produit de `A` et `B` dans `R`.

★ **Exercice 4: En Python** (exercice optionnel – d'après Emeric Tourniaire). On va coder l'état d'un ordinateur M99 par un couple de deux listes `(mem, reg)`. On pourra ainsi écrire `ordi[1][3]` pour connaître la valeur de `PC`, ou bien `ordi[0][55] = 162` pour changer l'état de la mémoire.

▷ **Question 1:** Écrivez une fonction `instruction(ordi, action)` qui exécute l'instruction action sur l'ordinateur ordi (étape 3 du fonctionnement). Le code de cette fonction sera sans doute une suite de if. On peut utiliser `return` (sans rien d'autre) pour sortir quand l'instruction a été exécutée.

▷ **Question 2:** Écrivez une fonction `execute(ordi)` qui prend en argument un ordinateur, exécute le programme qui s'y trouve, et renvoie la liste de valeurs que l'ordinateur afficherait.

▷ **Question 3:** Écrivez une fonction `parse()` lisant fichier texte contenant un programme écrit comme une suite de mnémos (comme `JPP 56` ou `LDA 13`), et retournant un objet `ordi`. Supposez d'abord que chaque ligne ne contient qu'un mnémo, écrit sur 3 lettres, et que les éventuels paramètres sont séparés par une seule espace.

On peut ajouter la possibilité de définir des *labels* pour nommer une case mémoire ainsi `AZE: LDA 13`. Comme le quatrième caractère est un ':', les trois précédents sont le nom de cette case. Le mnémo de la case commence après le ':'. Ailleurs dans le programme, on peut utiliser ce label ainsi `JPP AZE`, pour signifier que le branchement doit se faire vers la case où le label `AZE` est défini.

★ **Exercice 5: Compilation avancée** (exercice optionnel – d'après Emeric Tourniaire).

▷ **Question 1:** Soit un programme comme celui du fragment de code 1 ci-dessous. Écrivez une fonction permettant d'écrire un tel programme dans la mémoire d'un `ordi`. Les paramètres de cette fonction sont k, le nombre de tour de boucle à réaliser, et *liste*, les mnémos correspondants au corps de la boucle (dont on supposera qu'ils peuvent s'écrire n'importe où sans changement d'exécution).

▷ **Question 2:** On souhaite maintenant pouvoir compiler le second fragment de code. En supposant que x et y sont à une case prédéterminée de la mémoire et que *liste*1 et *liste*2 sont les listes de mnémos des corps des deux branches, expliquez comment traduire le programme en une nouvelle liste.

Fragment de code 1

```
1 Pour i de 0 à k:
2   Faire instructions.
```

Fragment de code 2

```
1 Si x > y Alors:
2   Faire instructions1
3 Sinon:
4   Faire instructions2
```

▷ **Question 3:** (grand jeu "Busy Beaver") Écrivez un ordinateur M99 qui exécute un nombre maximal d'instructions, puis s'arrête. Donnez un minorant du nombre d'instructions réalisées en détaillant son fonctionnement.

Réponse

Cette question est un clin d'oeil à un problème insoluble algorithmiquement au sujet des machines de Turing. Vous pouvez lire la page wikipédia du Castor affairé pour lus d'informations.

Fin réponse

★ **Réalisme du M99.** Cet ordinateur en papier est assez simple à utiliser avec seulement un crayon, mais il a été pensé pour être relativement réaliste par rapport aux vrais ordinateurs.

▷ **Mémoire.** La mémoire d'un vrai ordinateur est également découpée en cases mémoires, chacune étant dotée d'une adresse unique. En général, les vrais ordinateurs utilisent des cases de 1 octet (8 bits). Les ordinateurs 32 bits peuvent avoir jusqu'à 2^{32} cases (soit un peu plus de 4 Go de mémoire) tandis que les ordinateurs 64 bits peuvent avoir jusqu'à 2^{64} cases en théorie (18 Exaoctets, $18 \cdot 10^{18}$ octets).

▷ **32bits ou 64bits.** Comme on ne peut coder que 256 entiers sur un octet, il faut agréger plusieurs cases mémoires pour encoder des entiers plus grands. Un processeur 32bits utilise des mots de 4 cases (= l'encodage naturel des entiers y est fait sur 4 octets, càd 32 bits). Les mots font 8 octets en 64 bits. Le M99 utilise des mots de une case, car l'intervalle de valeurs est suffisant. Merci au bus mémoire du M99, arbitrairement large.

▷ **Registres et caches.** Les vrais processeurs ont également des registres afin de gérer au mieux le problème de la barrière mémoire. Ils ont également des caches pour optimiser les échanges entre la mémoire et le CPU. Là où lire en mémoire peut demander une centaine de cycles CPU, lire en cache prend entre 10 et 30 cycles. Pour simplifier, le M99 a un nombre très limité de registres, et aucun cache.

▷ **Opcodes.** Les vrais programmes sont également écrits sous forme d'opcodes en mémoire des vrais ordinateurs, avec le préfixe indiquant l'opération tandis que le suffixe indique les opérandes. Le jeu d'opérations élémentaires disponibles varie beaucoup d'une famille de processeurs à l'autre.

Pour le M99, nous avons choisi d'utiliser des cases mémoires de trois positions décimales, ce qui contraint fortement le nombre d'instructions disponibles. Ces contraintes sont parfaitement réalistes de celles que doivent résoudre les fabriquants de CPU. Ajouter des instructions simplifie l'écriture de programmes efficaces, mais complique le processeur, qui devient plus gros et donc plus cher et plus énergivore.

Les processeurs de la famille RISC (reduced instruction set CPU) visent la simplicité et n'offrent que peu d'instructions tandis que ceux de la famille CISC (complex instruction set CPU) offrent des opérations optimisées plus spécialisées, comme des opérations vectorielles.

Il serait faux de dire que l'une des familles est vraiment préférable à l'autre. Il s'agit plutôt de deux compromis différents entre complexité du processeur et complexité des programmes. Les processeurs des téléphones portables sont souvent des RISC (par exemple ceux du constructeur ARM) tandis que ceux des ordinateurs sont souvent des CISC (par exemple ceux des constructeurs Intel ou AMD).

▷ **Dépassements de capacité.** Sur M99, il n'est pas possible de savoir quand un dépassement de capacité a eu lieu. Sur certains processeurs, un tel événement déclenche l'exécution d'un bout de code spécifique. Sur d'autres, un registre d'état est modifié quand cela arrive et une opération de branchement similaire à `JPP` permet de sauter à une adresse si ce bit est vrai pour gérer correctement ces cas d'erreurs.

▷ **Entrées/sorties.** Gérer les entrées/sorties au travers d'adresses particulières de l'espace d'adressage du bus mémoire est parfaitement réaliste. En revanche, il n'est pas courant d'avoir plusieurs périphériques à la même adresse. On aurait pu séparer les lectures du clavier et les écritures à l'écran dans des zones mémoire différentes, mais le M99 n'a que 100 cases mémoires disponibles. De plus, nous avons ignoré toute la synchronisation qu'un vrai processeur doit faire pour échanger avec les périphériques, souvent bien plus lents que le processeur.

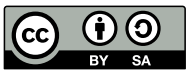
▷ **Pile et fonctions.** Les registres RA et SB sont inspirés de ce qu'on trouve sur les architectures Mips ou Risc-V. La pile d'exécution du M99 est cependant une simplification assez peu pratique de la réalité. La plupart des processeurs permettent d'accéder à une case de la pile directement en précisant un décalage par rapport au sommet de pile. Cela permet d'écrire rapidement `A := @(SB + 1); B := @(SB + 3)`, et de dépiler plusieurs cases d'un coup au moment du retour de fonction. Cela permet d'écrire du code plus compact que celui du M99, mais n'avoir qu'un seul mode d'adressage permet de simplifier le M99.

Les piles des vrais systèmes font également en sorte qu'on ne puisse pas modifier l'adresse de retour afin éviter les problèmes de sécurité. Mais qui a besoin de sécurité dans un ordinateur en papier ?

Notez que certaines machines virtuelles comme Forth ou uxn utilisent intensément la pile pour simplifier le travail des programmeurs. Le résultat est assez amusant à utiliser, pour ceux qui aiment.

Segmentation mémoire. Dans le M99, tous les programmes s'exécutent directement en mémoire. Dans un vrai système, les programmes sont isolés les uns des autres par le système d'exploitation. Chaque programme s'exécute dans un espace d'adressage virtuel, et tout accès à la mémoire physique est traduit un composant matériel dédié : le MMU (memory management unit). Introduire un système d'exploitation dans le M99 rendrait certainement les choses trop fastidieuses pour être faisables à la main.

À propos du M99. Cette activité de découverte a été inventée par le groupe InfoSansOrdi. Plus d'informations et de ressources en ligne: `https://github.com/InfoSansOrdi/M999`



	0	10	20	30	40	50	60	70	80	90	
0	110 LDA 10	042			199 LDA 99	160 LDA 60	002 DAT 2	975 CAL 75	409 RET		0
1	211	010			310 MOV A R	261 LDB 61	001 DAT 1	975 CAL 75	482 PSH B		1
2	401 SUB				059 STR 59	401 SUB	000 DAT 0	490 POP R	409 RET		2
3	607	199			299 LDB 99	060 STR 60		099 STR 99			3
4	320	310			320 MOV B R	646 JPP 46	199 LDA 99	599 JMP 99			4
5	099	010			060 STR 60	162 LDA 62	481 PSH A	491 POP A			5
6	599	199			162 LDA 62	310 MOV A R	199 LDA 99	492 POP B			6
7	310	310			259 LDB 59	099 STR 99	481 PSH A	401 SUB			7
8	099	011			400 ADD	599 JMP 99	199 LDA 99	681 JPP 81			8
9	599	500			062 STR 62	003 DAT 3	481 PSH A	481 PSH A		INPUT OUTPUT	9
	0	10	20	30	40	50	60	70	80	90	

M99

ordinateur en papier

v230910

R	A	B	PC
R0	R1	R2	instr

Code	*Mnémo*	*Signification*
0 x y	STR xy	R → mem(xy)
1 x y	LDA xy	A ← mem(xy)
2 x y	LDB xy	B ← mem(xy)
3 x y	MOV x y	Copie le registre Rx dans Ry

310 (MOV A R) R := A

4 0 0	ADD	R := A + B
4 0 1	SUB	R := A − B

5 x y	JMP xy	PC := xy
6 x y	JPP xy	R > 0 ⇒ PC :=xy
7 x y	JEQ xy	R=xy ⇒ PC+=2
8 x y	JNE xy	R≠xy ⇒ PC+=2