

Institut d'enseignement à distance
Licence 1 - Architecture des ordinateurs
Chapitre 13

BLANCHARD Patrice
Numéro étudiant : 18904701

14 mai 2020

Table des matières

1 - Il manque les microcodes de la phase 2 pour le cas où OP contient E2. Compléter.	3
2 - Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle code pourrait correspondre l'instruction <code>IN #</code> ?	4
3 - Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle opération pourrait correspondre le code 01 ?	6
4 - (Piégé) Dans le programme de bootstrap, les adresses 18 à 1E sont remplies de 0. Que se passe-t-il au cours du bootstrap quand l'ordinateur exécute ces instructions ?	7
5 - Écrire un programme pour l'ordinateur en papier qui lit deux nombres et affiche leur produit. Comme pour le programme de bootstrap donné dans le chapitre, indiquer les adresses, les codes, les mnémoniques et les commentaires. (on peut calculer le produit avec une suite d'additions simples ; faute de décalage à droite, l'ordinateur en papier ne nous permet pas d'utiliser l'algorithme plus rapide vu en cours).	8
6 - Même question que précédemment pour la division, avec des soustractions successives. On pourra s'inspirer de l'algorithme suivant :	10
7 - Quelles parties de l'ordinateur faut-il modifier pour ajouter une instruction <code>SHIFT</code> de décalage à droite ?	12
8 - (Assez difficile) Dans la mémoire, on trouve de l'adresse 30 à 45 les valeurs suivantes : 10, 3a, 00 31 40 32 60 33 48 32 49 33 40 33 22 ff 12 34 41 32 10 44. Dans PC se trouve 30. Que fera ce programme quand l'ordinateur en papier va tourner et l'exécuter ?	13
9 - (Assez facile) Indiquer les mnémoniques qui correspondent aux valeurs suivantes dans la mémoire.	20
10 - (Amusant mais difficile) Écrire un programme qui met la plus grande partie possible de la mémoire à 0 (en supposant qu'il y a pas de mémoire morte).	24
11 - (Long, pas de corrigé) Écrire un programme d'extraction de la racine carré d'un nombre, avec la méthode de Newton pour l'ordinateur en papier.	26
12 - Programme réalisé en C et correspondant à l'exercice cx25.0 du cours de programmation impérative	26
13 - Programme réalisé en C et correspondant à l'exercice cx25.1 (STEPPER) du cours de programmation impérative	30

1 - Il manque les microcodes de la phase 2 pour le cas où OP contient E2. Compléter.

Tous les codes opératoires sont représentés sous la forme de microcode en page 138 sauf l'instruction : $NAND * \alpha$ de valeur E2 et décrite sous la forme : $A \leftarrow \neg[A \ \& \ *(\alpha)]$ en page 134.

Son microde correspond à :



Où :

- 17 Écrire $NAND$ dans l'Unité Arithmétique et Logique.
- 1 $(RS) \leftarrow (PC)$ (\leftarrow représente une affectation)
- 13 Décoder l'adresse contenue dans le registre de sélection (RS) et reporter le contenu de la case mémoire ainsi désignée dans le registre mot (RM).
- 6 $(AD) \leftarrow (RM)$
- 7 $(RS) \leftarrow (AD)$
- 13 Décoder l'adresse contenue dans le registre de sélection (RS) et reporter le contenu de la case mémoire ainsi désignée dans le registre mot (RM).
- 6 $(AD) \leftarrow (RM)$
- 7 $(RS) \leftarrow (AD)$
- 13 Décoder l'adresse contenue dans le registre de sélection (RS) et reporter le contenu de la case mémoire ainsi désignée dans le registre mot (RM).
- 12 Effectuer l'opération affichée sur l'UAL en prenant le contenu de (A) comme premier opérande et le contenu de (RM) comme second. Écrire le résultat dans l'accumulateur.

Observations :

Le microcode de E2 : $NAND * \alpha$ est presque similaire au microcode de E0 : $ADD * \alpha$ et E1 : $SUB * \alpha$. En effet, chaque instruction se distingue seulement par l'opération de préparation du calcul intégré dans leur toute première opération : E0 additionne le contenu de la case dont l'adresse est dans la case d'adresse alpha avec le contenu du registre (A), autrement dit l'accumulateur, tandis que E1 soustrait et E2 effectue l'opération logique $NAND$ tous les deux sur ces mêmes opérandes.

Conclusion :

Toutes ces observations logiques viennent appuyées l'exactitude du microde de E2 présenté en haut de cette page.

2 - Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle code pourrait correspondre l'instruction IN # ?

1. LA LOGIQUE DES CODES OPÉRATOIRES :

ADD #	20	ADD α	60	ADD $*\alpha$	E0
SUB #	21	SUB α	61	SUB $*\alpha$	E1
NAND #	22	NAND α	62	NAND $*\alpha$	E2
LOAD #	00	LOAD α	40	LOAD $*\alpha$	C0
IN #	??	IN α	49	IN $*\alpha$	C9
OUT #	??	OUT α	41	OUT $*\alpha$	C1

Observations :

Les codes opératoires LOAD, IN, OUT, provenant des catégories transferts et entrées-sorties, lorsqu'ils sont associés à la valeur α affichent tous le même chiffre de gauche : le numéro 4. De plus, ces trois codes opératoires suivi de $*\alpha$ présente également tous la même indication à cette même position : la lettre C. Sachant que l'opérateur LOAD # manifeste comme valeur le chiffre 0 comme premier élément en partant de la gauche, on peut en déduire que le code opératoire de l'opération IN # disposera également d'un 0 comme chiffre des dizaines.

Concernant l'élément le plus à droite de chaque code opératoire suivant : LOAD #, LOAD α , LOAD $*\alpha$, celui-ci se trouve être le même : le chiffre 0. En suivant la même logique et étant donné que les codes opératoires de IN α et IN $*\alpha$ comportent le chiffre 9 à l'emplacement des unités, tout porte à laisser croire que le IN # disposera également de ce même chiffre à ce même emplacement.

Conclusion :

Vraisemblablement, d'après la représentation ci-dessus du regroupement de mnémoniques arithmétiques associés à leur code opératoire par catégorie, le code 09 conviendrait à l'instruction IN #.

2. LA LOGIQUE DES MICROCODES :

Quelques exemples :

00	LOAD #	1	13	3							
40	LOAD α	1	13	6	7	13	3				
C0	LOAD $*\alpha$	1	13	6	7	13	6	7	13	3	
20	ADD #	10	1	13	12						
60	ADD α	10	1	13	6	7	13	12			
E0	ADD $*\alpha$	10	1	13	6	7	13	6	7	13	12
09	IN #	??	??	??						
49	IN α	1	13	6	7	16	8	14			
C9	IN $*\alpha$	1	13	6	7	13	6	7	16	8	14

Observations :

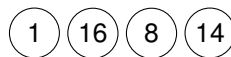
L'agencement des microcodes met en lumière la présence d'une même combinaison de microcodes :



Quelques soient les codes opératoires associés à α , cette même combinaison est présente. Elle est même doublée dans le cas de l'utilisation de $*\alpha$ et inexistante pour tout opérateur agrémenté du symbole $\#$. En effet, ⑥ est une étape qui consiste à affecter le registre mémoire (RM) au registre d'adresse (AD), ⑦ assigne AD au registre de sélection (RS) et ⑬ décode RS et reporte le contenu de sa case mémoire dans le registre RM. Cette combinaison est nécessaire pour charger le contenu d'une case mémoire autant en cas d'adressage absolu désigné par le signe α et dans le cas d'adressage indirect, cette combinaison est doublée lors de cet adressage remarquable par l'association $*\alpha$ ou une valeur sera sélectionné à partir de l'adresse d'une case qui contient l'adresse de la valeur recherché. Cette combinaison est inexistante pour les codes opératoires d'adressage immédiat révélés par le symbole $\#$.

Conclusion :

Le microcode de IN $\#$ correspond au microcode de IN α ou IN $*\alpha$ sans aucune combinaison ⑬ ⑥ ⑦, de représentation :



3 - Étant donné la logique qui existe entre les codes opératoires et le microcode, à quelle opération pourrait correspondre le code 01 ?

1. LA LOGIQUE DES CODES OPÉRATOIRES :

ADD #	20	ADD α	60	ADD * α	E0
SUB #	21	SUB α	61	SUB * α	E1
NAND #	22	NAND α	62	NAND * α	E2
LOAD #	00	LOAD α	40	LOAD * α	C0
IN #	09	IN α	49	IN * α	C9
OUT #	??	OUT α	41	OUT * α	C1

Observations :

En admettant que le code opératoire IN # est comme valeur 09. Le regroupement des codes LOAD # et IN # de catégories transferts et entrées-sorties présente le chiffre 0 à l'indice des dizaines et tous les codes opératoires OUT α et OUT * α disposent du chiffre 1 à l'emplacement des unités.

Conclusion :

Étant donné la logique qui existe entre les codes opératoires, le code 01 correspondrait à OUT #

2. LA LOGIQUE DES MICROCODES :

01 OUT # ?? ?? ...

41 OUT α (1) (13) (6) (7) (13) (9)

C1 OUT * α (1) (13) (6) (7) (13) (6) (7) (13) (9)

Observations :

En établissant le même constat que dans l'exercice 13.2, le microcode de OUT # revient à celui de OUT α ou OUT * α sans aucune combinaison de (13) (6) (7).

Conclusion :

Le microcode de OUT # est le suivant :

(1) (13) (9)

4 - (Piégé) Dans le programme de bootstrap, les adresses 18 à 1E sont remplies de 0. Que se passe-t-il au cours du bootstrap quand l'ordinateur exécute ces instructions ?

Le bootstrap est implémenté de sorte à ignorer ces adresses. En effet, son rôle est de charger un programme, qui débutera à une certaine adresse (dont l'adresse est stockée à l'adresse 20). Lorsque le chargement de l'ensemble des instructions constituant le bootstrap est achevé, le branchement conditionnel BRZ 1F intégré dans celui-ci à l'adresse 12 a pour rôle d'assigner l'adresse 1F au Program Counter (l'adresse de la prochaine instruction) induisant l'effet d'ignorer les adresses comprises de la 18 à la 1E incluse. Aucune instructions fournies dans cet intervalle seront exécutées.

- 5 - Écrire un programme pour l'ordinateur en papier qui lit deux nombres et affiche leur produit. Comme pour le programme de bootstrap donné dans le chapitre, indiquer les adresses, les codes, les mnémoniques et les commentaires. (on peut calculer le produit avec une suite d'additions simples ; faute de décalage à droite, l'ordinateur en papier en nous permet pas d'utiliser l'algorithme plus rapide vu en cours).**

ADD	CODE	MNÉMONIQUE	COMMENTAIRE
23	00	LOAD #00	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR 0
24	00		
25	48	STORE 47	RANGER LA VALEUR DU REGISTRE DE L'ACCUMULATEUR À L'ADRESSE 47 (RÉSULTAT)
26	47		
27	49	IN 45	SAISIR LA VALEUR DU PREMIER NOMBRE (NB1), L'ENREGISTRER À L'ADRESSE 45
28	45		
29	40	LOAD 45	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE NB1
2A	45		
2B	12	BRZ 41	SI NB1 = 0 ALORS PC ← 41 : AFFICHAGE DE LA VALEUR À L'ADRESSE 47 (RÉSULTAT)
2C	41		
2D	49	IN 46	SAISIR LA VALEUR DU SECOND NOMBRE : NB2
2E	46		
2F	40	LOAD 46	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE L'ADRESSE 46 : NB2
30	46		
31	12	BRZ 41	SI NB2 = 0 ALORS PC ← 41 : AFFICHAGE DU RÉSULTAT PRÉSENT À L'ADRESSE 47
32	41		
33	48	STORE 46	RANGER LA VALEUR DU REGISTRE DE L'ACCUMULATEUR À L'ADRESSE 46 (NB2)
34	46		
35	40	LOAD 47	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE L'ADRESSE 47 (RÉSULTAT)
36	47		
37	60	ADD 45	AJOUTER NB1 AU RÉSULTAT
38	45		
39	48	STORE 47	RANGER LA VALEUR DU REGISTRE DE L'ACCUMULATEUR À L'ADRESSE 47
3A	47		
3B	40	LOAD 46	CHARGER LA VALEUR DE L'ADRESSE 46 (NB2)
3C	46		
3D	21	SUB # 1	SOUSTRAIRE UN À LA VALEUR DE NB2 PRÉSENT DANS LE REGISTRE DE L'ACCUMULATEUR
3E	01		
3F	10	JUMP 31	RÉALISER UN SAUT À L'ADRESSE 31
40	31		
41	41	OUT 47	AFFICHER LA VALEUR DE L'ADRESSE 47 (RÉSULTAT)
42	47		
43	10	JUMP 23	RÉALISER UN SAUT À L'ADRESSE 23 (DÉBUT DU PROGRAMME)
44	23		
45	??		LE PREMIER NOMBRE : NB1
46	??		LE DEUXIÈME NOMBRE : NB2
47	??		LE RÉSULTAT FINAL OU TEMPORAIRE


```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
(base) [patrice@g3-3590 cx25.0]$ gcc -g -Wall cx25.c
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.5.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 023 | A : 000 | LOAD #00    A = 00
PC : 025 | A : 000 | STORE47    data[47] = A
PC : 027 | A : 000 | IN 45      data[45] = input(val ?)
Saisir une valeur en décimal : 3

PC : 029 | A : 000 | LOAD 45    A = data[45]
PC : 02B | A : 003 | BRZ 41     Si A = 0 alors PC = 41
PC : 02D | A : 003 | IN 46      data[46] = input(val ?)
Saisir une valeur en décimal : 2

PC : 02F | A : 003 | LOAD 46    A = data[46]
PC : 031 | A : 002 | BRZ 41     Si A = 0 alors PC = 41
PC : 033 | A : 002 | STORE46    data[46] = A
PC : 035 | A : 002 | LOAD 47    A = data[47]
PC : 037 | A : 000 | ADD 45     A += data[45]
PC : 039 | A : 003 | STORE47    data[47] = A
PC : 03B | A : 003 | LOAD 46    A = data[46]
PC : 03D | A : 002 | SUB #01    A -= 01
PC : 03F | A : 001 | JUMP 31    PC = 31
PC : 031 | A : 001 | BRZ 41     Si A = 0 alors PC = 41
PC : 033 | A : 001 | STORE46    data[46] = A
PC : 035 | A : 001 | LOAD 47    A = data[47]
PC : 037 | A : 003 | ADD 45     A += data[45]
PC : 039 | A : 006 | STORE47    data[47] = A
PC : 03B | A : 006 | LOAD 46    A = data[46]
PC : 03D | A : 001 | SUB #01    A -= 01
PC : 03F | A : 000 | JUMP 31    PC = 31
PC : 031 | A : 000 | BRZ 41     Si A = 0 alors PC = 41
PC : 041 | A : 000 | OUT 47     print(data[47])
OUT en décimal : 6
(Continuer : O/N): n
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 1 – Exécution du programme de l'exercice 13.5

6 - Même question que précédemment pour la division, avec des soustractions successives. On pourra s'inspirer de l'algorithme suivant :

Etant donné x et y , calculer a et b tels que $a \times b + b = y$

$a \leftarrow 0$

$b \leftarrow y$

tant que $b > x$

$a \leftarrow a + 1$

$b \leftarrow b - x$

ADD	CODE	MNÉMONIQUE	COMMENTAIRE
23	49	IN 48	SAISIR X (LE DIVIDENDE), L'ENREGISTRER À L'ADRESSE 48
24	48		
25	49	IN 47	SAISIR Y (LE DIVISEUR), L'ENREGISTRER À L'ADRESSE 47
26	47		
27	00	LOAD #00	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR 00
28	00		
29	48	STORE 45	RANGER LA VALEUR DU REGISTRE A À L'ADRESSE 45 (Q, LE QUOTIENT)
2A	45		
2B	40	LOAD 48	CHARGER LA VALEUR X (LE DIVIDENDE) PRÉSENT À L'ADRESSE 48
2C	48		
2D	48	STORE 46	RANGER LA VALEUR X À L'ADRESSE 46 (R, LE RESTE)
2E	46		
2F	40	LOAD 45	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE L'ADRESSE 45 (Q, LE QUOTIENT)
30	45		
31	20	ADD# 01	INCRÉMENTER LA VALEUR PRÉSENT DANS LE REGISTRE DE L'ACCUMULATEUR
32	01		
33	48	STORE 45	RANGER LA VALEUR PRÉSENT DANS LE REGISTRE DE L'ACCUMULATEUR À L'ADRESSE 45
34	45		
35	40	LOAD 46	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE L'ADRESSE 46 (B, LE RESTE)
36	46		
37	61	SUB 47	SOUSTRAIRE LA VALEUR DU REGISTRE DE L'ACCUMULATEUR (R) ET Y (LE DIVISEUR)
38	47		
39	48	STORE 46	RANGER LA VALEUR PRÉSENT DANS LE REGISTRE DE L'ACCUMULATEUR À L'ADRESSE 46 (R)
3A	46		
3B	12	BRZ 3F	SI A = 0 ALORS PC ← 3F : AFFICHAGE DE A
3C	3F		
3D	10	JUMP 2F	SAUT À L'ADRESSE 2F
3E	2F		
3F	41	OUT 45	AFFICHER LA VALEUR DE L'ADRESSE 45 (LE QUOTIENT)
40	45		
41	41	OUT 46	AFFICHER LA VALEUR DE L'ADRESSE 46 (LE RESTE)
42	46		
43	10	JUMP 23	SAUT À L'ADRESSE 23
44	23		
45	??		LA VARIABLE : Q, LE QUOTIENT
46	??		LA VARIABLE : R, LE RESTE
47	??		LA VARIABLE : Y, LE DIVISEUR
48	??		LA VARIABLE : X, LE DIVIDENDE

```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.6.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 023 | A : 000 | IN 48      data[48] = input(val ?)
Saisir une valeur en décimal : 15

PC : 025 | A : 000 | IN 47      data[47] = input(val ?)
Saisir une valeur en décimal : 5

PC : 027 | A : 000 | LOAD #00   A = 00
PC : 029 | A : 000 | STORE45   data[45] = A
PC : 02B | A : 000 | LOAD 48    A = data[48]
PC : 02D | A : 015 | STORE46   data[46] = A
PC : 02F | A : 015 | LOAD 45    A = data[45]
PC : 031 | A : 000 | ADD #01    A += 01
PC : 033 | A : 001 | STORE45   data[45] = A
PC : 035 | A : 001 | LOAD 46    A = data[46]
PC : 037 | A : 015 | SUB 47     A -= (data[47]
PC : 039 | A : 010 | STORE46   data[46] = A
PC : 03B | A : 010 | BRZ 3F     Si A = 0 alors PC = 3F
PC : 03D | A : 010 | JUMP 2F    PC = 2F
PC : 02F | A : 010 | LOAD 45    A = data[45]
PC : 031 | A : 001 | ADD #01    A += 01
PC : 033 | A : 002 | STORE45   data[45] = A
PC : 035 | A : 002 | LOAD 46    A = data[46]
PC : 037 | A : 010 | SUB 47     A -= (data[47]
PC : 039 | A : 005 | STORE46   data[46] = A
PC : 03B | A : 005 | BRZ 3F     Si A = 0 alors PC = 3F
PC : 03D | A : 005 | JUMP 2F    PC = 2F
PC : 02F | A : 005 | LOAD 45    A = data[45]
PC : 031 | A : 002 | ADD #01    A += 01
PC : 033 | A : 003 | STORE45   data[45] = A
PC : 035 | A : 003 | LOAD 46    A = data[46]
PC : 037 | A : 005 | SUB 47     A -= (data[47]
PC : 039 | A : 000 | STORE46   data[46] = A
PC : 03B | A : 000 | BRZ 3F     Si A = 0 alors PC = 3F
PC : 03F | A : 000 | OUT 45     print(data[45])
OUT en décimal : 3
(Continuer : O/N): o

PC : 041 | A : 000 | OUT 46     print(data[46])
OUT en décimal : 0
(Continuer : O/N): n
(base) [patrice@g3-3590 cx25.0]$
```

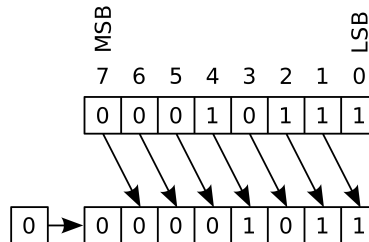
FIGURE 2 – Exécution du programme de l'exercice 13.6

7 - Quelles parties de l'ordinateur faut-il modifier pour ajouter une instruction SHIFT de décalage à droite ?

Décrire ces modifications, choisir un code opératoire, détailler les modifications à apporter au séquenceur.

D'après la source wikipedia suivante https://fr.wikipedia.org/wiki/Op%C3%A9ration_bit_%C3%A0_bit#D%C3%A9calage_%C3%A0_droite :

Le décalage à droite représente une division entière par 2. Si le bit de poids faible est à 1, c'est-à-dire que le nombre est impair, celui-ci sera perdu, conformément au principe de la division entière dans laquelle il ne peut pas y avoir de partie fractionnaire.



Exemple :

00010111 (+23) RIGHT-SHIFT
= 00001011 (+11)

Réponse :

Pour ajouter une instruction SHIFT, il convient de :

- intégrer la possibilité d'effectuer un décalage à droite en ajoutant cette opération à l'unité arithmétique logique.
- ajouter au séquenceur le code opcode (18) relatif au décalage à droite ainsi que sa séquence de microcodes.

Présentation des différents code op SHIFT :

MNÉMONIQUES :	CODE OP	DESCRIPTION	MICROCODES
SHIFT # :	30	$A \leftarrow \text{SHIFT}[V]$	(18) (1) (13) (12)
SHIFT α :	31	$A \leftarrow \text{SHIFT}[(\alpha)]$	(18) (1) (13) (6) (7) (13) (12)
SHIFT $\ast \alpha$:	E3	$A \leftarrow \text{SHIFT}[\ast(\alpha)]$	(18) (1) (13) (6) (7) (13) (6) (7) (13) (12)

Microcode :

- (18) Écrire SHIFT dans l'UAL
- (1) (RS) \leftarrow (PC)
- (13) Décoder l'adresse contenue dans RS et reporter le contenu dans (RM)
- (12) Effectuer l'opération affichée sur l'UAL entre A et RM et écrire le résultat dans A (le registre de l'accumulateur)
- (6) (AD) \leftarrow (RM)
- (7) (RS) \leftarrow (RM)

8 - (Assez difficile) Dans la mémoire, on trouve de l'adresse 30 à 45 les valeurs suivantes : 10, 3a, 00 31 40 32 60 33 48 32 49 33 40 33 22 ff 12 34 41 32 10 44. Dans PC se trouve 30. Que fera ce programme quand l'ordinateur en papier va tourner et l'exécuter ?

ADRESSE	VALEUR	MNÉMONIQUES	COMMENTAIRE
30	10	JUMP 3A	SAUT À L'ADRESSE 3A
31	3A		
32	00		LA VARIABLE B
33	31		LA VARIABLE C (DE VALEUR 31)
34	40	LOAD 32	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR À L'ADRESSE 32 (B)
35	32		
36	60	ADD 33	ADDITIONNER B ET C
37	33		
38	48	STORE 32	RANGER LE RESULTAT À L'ADRESSE 32 (B)
39	32		
3A	49	IN 33	SAISIR LE NOMBRE C
3B	33		
3C	40	LOAD 33	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR À L'ADRESSE 33 (C)
3D	33		
3E	22	NAND# FF	CALCULER C NAND FF
3F	FF		
40	12	BRZ 34	SI A NAND FF = 0 ALORS PC ← 34
41	34		
42	41	OUT 32	AFFICHER LA VALEUR DE L'ADRESSE 32 (B)
43	32		
44	10	JUMP 44	SAUT À L'ADRESSE 44
45	44		

En l'état, ce programme correspond à l'algorithme suivant :

Initialisation au préalable de B = 0 et C = 31.

```

C = IN (C)
if (C NAND FF) = 0
    B = B + C
    C = IN(C)
else
    OUT (B)
    while (1)

```

Observations :

Au démarrage du programme lors de la première saisie de la valeur de C, celui-ci affiche la valeur de B à zéro si (C NAND FF) est différent de zéro (valeur de C autre que 255(FF)) ou sinon si (C NAND FF) est égale 0 (valeur de C égale à 255 (FF)), la valeur de B sera dans ce cas additionnée à celle de C puis enregistrée en B, suivant le nombre de fois ou (C NAND FF) sera égale à zéro, avant d'être affichée si une saisie de la valeur C viendrait à engendrer (C NAND FF) différent de zéro. Quoiqu'il en soit l'affichage de la valeur de B est suivi d'une boucle infinie qui monopolise le terminal en affichage, comme on peut le constater après l'observation de la capture présente trois pages plus loin.

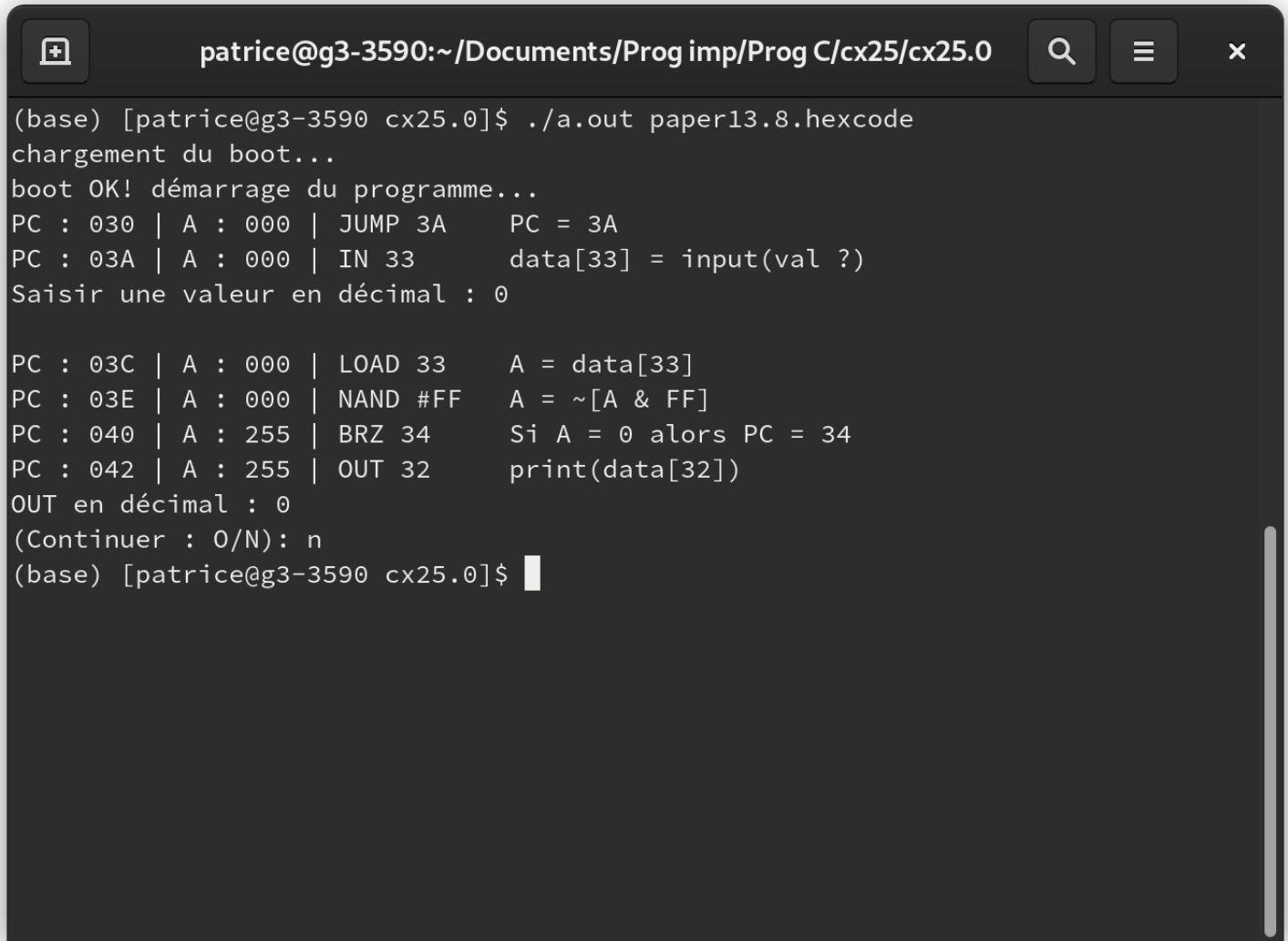
Conclusion :

Deux cas possibles :

- Au démarrage du programme, si (C NAND FF) != 0 donc C != 255, B sera affiché à zéro.
- Sinon si (C NAND FF) == 0 donc C = 255, B prendra la valeur 255 multipliée par le nombre de fois ou cette valeur sera proposée en saisie de C avant que le résultat soit affiché par une saisie autre que C = 255.

Dans les deux cas, l'affichage est suivi d'une boucle infinie qui engendre l'affichage continué suivant :

PC : 044 | A : 253 | JUMP 44 PC=44

A terminal window with a dark background and light gray text. The title bar shows the user 'patrice@g3-3590' and the current directory '~/Documents/Prog imp/Prog C/cx25/cx25.0'. The terminal content shows the execution of a program named 'paper13.8.hexcode'. It starts with a boot sequence, then enters a loop where it reads input from the user. The first input is '0', which is stored in memory location 33. Subsequent instructions load this value into register A, perform a bitwise NOT operation, and then check if A is zero. Since A is zero, the program jumps to instruction 34. Finally, it prints the value at memory location 32 (which is 0) and asks the user if they want to continue. The user enters 'n', and the program returns to the shell prompt.

```
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.8.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 030 | A : 000 | JUMP 3A      PC = 3A
PC : 03A | A : 000 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 0

PC : 03C | A : 000 | LOAD 33     A = data[33]
PC : 03E | A : 000 | NAND #FF   A = ~[A & FF]
PC : 040 | A : 255 | BRZ 34      Si A = 0 alors PC = 34
PC : 042 | A : 255 | OUT 32      print(data[32])
OUT en décimal : 0
(Continuer : 0/N): n
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 3 – 1er cas : Exécution du programme de l'exercice 13.8 avec une saisie de C égale à zéro au démarrage du programme

```
patrice@g3-3590:~/CloudStation/IED - Info - Licence 1/architectu...
(base) [patrice@g3-3590 prog]$ ./a.out paper13.8.hexcode
Chargement du bootstrap...
Chargement du bootstrap terminé !
Démarrage du programme...
PC : 030 | A : 000 | JUMP 3A    PC = 3A
PC : 03A | A : 000 | IN 33     data[33] = input(val ?)
Saisir une valeur en décimal : 255

PC : 03C | A : 000 | LOAD 33   A = data[33]
PC : 03E | A : 255 | NAND #FF  A = ~(A & FF)
PC : 040 | A : 000 | BRZ 34    Si A = 0 alors PC = 34
PC : 034 | A : 000 | LOAD 32   A = data[32]
PC : 036 | A : 000 | ADD 33    A += data[33]
PC : 038 | A : 255 | STORE 32  data[32] = A
PC : 03A | A : 255 | IN 33     data[33] = input(val ?)
Saisir une valeur en décimal : 0

PC : 03C | A : 255 | LOAD 33   A = data[33]
PC : 03E | A : 000 | NAND #FF  A = ~(A & FF)
PC : 040 | A : 255 | BRZ 34    Si A = 0 alors PC = 34
PC : 042 | A : 255 | OUT 32    print(data[32])
OUT en décimal : 255
(Continuer : 0/N):
```

FIGURE 4 – 2ème cas : Exécution du programme de l'exercice 13.8 avec une saisie de C égale à 255 au démarrage du programme

ADRESSE	VALEUR	MNÉMONIQUES	COMMENTAIRE
30	10	JUMP 3A	SAUT À L'ADRESSE 3A
31	3A		
32	00		La variable B
33	31		La variable C (de valeur 31)
34	40	LOAD 32	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR À L'ADRESSE 32 (B)
35	32		
36	60	ADD 33	ADDITIONNER B ET C
37	33		
38	48	STORE 32	RANGER LA VALEUR DE B À L'ADRESSE 32
39	32		
3A	49	IN 33	SAISIR LE NOMBRE C
3B	33		
3C	40	LOAD	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR À L'ADRESSE 33 (C)
3D	33		
3E	22	NAND# FF	CALCULER C NAND FF
3F	FF		
40	12	BRZ 34	SI C NAND FF = 0 ALORS PC ← 34
41	34		
42	41	OUT 32	AFFICHER LA VALEUR DE L'ADRESSE 32 (B)
43	32		
44	10	JUMP	SAUT À L'ADRESSE 34
45	34		

Quelques informations au sujet de la porte NAND : https://en.wikipedia.org/wiki/NAND_gate

The NAND gate has the property of functional completeness, which it shares with the NOR gate. That is, any other logic function (AND, OR, etc.) can be implemented using only NAND gates. An entire processor can be created using NAND gates alone. In TTL ICs using multiple-emitter transistors, it also requires fewer transistors than a NOR gate.

As NOR gates are also functionally complete, if no specific NAND gates are available, one can be made from NOR gates.

Traduction :

La porte NAND a la propriété d'exhaustivité fonctionnelle, qu'elle partage avec la porte NOR. Autrement dit, toute autre fonction logique (AND, OR, etc.) peut être implémentée en utilisant uniquement des portes NAND. Un processeur entier peut être créé en utilisant uniquement des portes NAND. Dans les circuits intégrés TTL utilisant des transistors à émetteurs multiples, il nécessite également moins de transistors qu'une porte NOR.

Comme les portes NOR sont également fonctionnellement complètes, si aucune porte NAND spécifique n'est disponible, une peut être faite à partir des portes NOR.

Table de vérité :

a	b	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.82.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 030 | A : 000 | JUMP 3A      PC = 3A
PC : 03A | A : 000 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 255

PC : 03C | A : 000 | LOAD 33     A = data[33]
PC : 03E | A : 255 | NAND #FF   A = ~(A & FF)
PC : 040 | A : 000 | BRZ 34     Si A = 0 alors PC = 34
PC : 034 | A : 000 | LOAD 32     A = data[32]
PC : 036 | A : 000 | ADD 33     A += data[33]
PC : 038 | A : 255 | STORE32    data[32] = A
PC : 03A | A : 255 | IN 33     data[33] = input(val ?)
Saisir une valeur en décimal : 0

PC : 03C | A : 255 | LOAD 33     A = data[33]
PC : 03E | A : 000 | NAND #FF   A = ~(A & FF)
PC : 040 | A : 255 | BRZ 34     Si A = 0 alors PC = 34
PC : 042 | A : 255 | OUT 32     print(data[32])
OUT en décimal : 255
(Continuer : O/N): n
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 6 – Exécution du programme de l'exercice 13.8 avec un jump relié à l'adresse 34

```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.82.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 030 | A : 000 | JUMP 3A      PC = 3A
PC : 03A | A : 000 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 255

PC : 03C | A : 000 | LOAD 33     A = data[33]
PC : 03E | A : 255 | NAND #FF    A = ~[A & FF]
PC : 040 | A : 000 | BRZ 34      Si A = 0 alors PC = 34
PC : 034 | A : 000 | LOAD 32     A = data[32]
PC : 036 | A : 000 | ADD 33      A += data[33]
PC : 038 | A : 255 | STORE32     data[32] = A
PC : 03A | A : 255 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 23

PC : 03C | A : 255 | LOAD 33     A = data[33]
PC : 03E | A : 023 | NAND #FF    A = ~[A & FF]
PC : 040 | A : 232 | BRZ 34      Si A = 0 alors PC = 34
PC : 042 | A : 232 | OUT 32      print(data[32])
OUT en décimal : 255
(Continuer : O/N): o

PC : 044 | A : 232 | JUMP 34      PC = 34
PC : 034 | A : 232 | LOAD 32     A = data[32]
PC : 036 | A : 255 | ADD 33      A += data[33]
PC : 038 | A : 278 | STORE32     data[32] = A
PC : 03A | A : 278 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 255

PC : 03C | A : 278 | LOAD 33     A = data[33]
PC : 03E | A : 255 | NAND #FF    A = ~[A & FF]
PC : 040 | A : 000 | BRZ 34      Si A = 0 alors PC = 34
PC : 034 | A : 000 | LOAD 32     A = data[32]
PC : 036 | A : 278 | ADD 33      A += data[33]
PC : 038 | A : 533 | STORE32     data[32] = A
PC : 03A | A : 533 | IN 33       data[33] = input(val ?)
Saisir une valeur en décimal : 0

PC : 03C | A : 533 | LOAD 33     A = data[33]
PC : 03E | A : 000 | NAND #FF    A = ~[A & FF]
PC : 040 | A : 255 | BRZ 34      Si A = 0 alors PC = 34
PC : 042 | A : 255 | OUT 32      print(data[32])
OUT en décimal : 533
(Continuer : O/N): n
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 7 – Un deuxième exemple

9 - (Assez facile) Indiquer les mnémoniques qui correspondent aux valeurs suivantes dans la mémoire.

(Difficile) Commenter le programme. Que fait-il ?

L'ensemble de ces instructions forme l'algorithme suivant :

Saisir A

```
X = A
Y = A
Z = 0
while (X != 0)
    X = X - 1
    Z = Z + Y
X = Z
Y = X
X = A
Z = 0
while (X != 0)
    X = X - 1
    Z = Z + Y
```

X = Z

Afficher X

Conclusion :

La capture d'écran en page suivante révèle que ce programme demande à l'utilisateur de saisir un nombre dans le but de calculer et d'afficher le cube de celui-ci.

ADRESSE	VALEUR	MNÉMONIQUES	COMMENTAIRE
50	49	IN 70	SAISIR A, L'ENREGISTRER À L'ADRESSE 70
51	70		
52	40	LOAD 70	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR À L'ADRESSE 70 (A)
53	70		
54	48	STORE 71	RANGER LA VALEUR DE A À L'ADRESSE 71 (X)
55	71		
56	48	STORE 72	RANGER LA VALEUR DE A À L'ADRESSE 72 (Y)
57	72		
58	00	LOAD #5E	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR 5E
59	5E		
5A	48	STORE 8D	RANGER LA VALEUR 5E À L'ADRESSE 8D
5B	8D		
5C	10	JUMP 74	SAUT À L'ADRESSE 74
5D	74		
5E	40	LOAD 71	CHARGER LA VALEUR DE L'ADRESSE 71 (X)
5F	71		
60	48	STORE 72	RANGER LA VALEUR DE X À L'ADRESSE 72 (Y)
61	72		
62	40	LOAD 70	CHARGER LA VALEUR DE L'ADRESSE 70 (A)
63	70		
64	48	STORE 71	RANGER LA VALEUR DE A À L'ADRESSE 71 (X)
65	71		
66	00	LOAD #6C	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR 6C
67	6C		
68	48	STORE 8D	RANGER LA VALEUR 6C À L'ADRESSE 8D
69	8D		
6A	10	JUMP 74	SAUT À L'ADRESSE 74
6B	74		
6C	41	OUT 71	AFFICHER LA VALEUR DE L'ADRESSE 71
6D	71		
6E	10	JUMP 6E	SAUT À L'ADRESSE 6E
6F	6E		
70	00		LA VARIABLE : A
71	00		LA VARIABLE : X
72	00		LA VARIABLE : Y
73	00		LA VARIABLE : Z
74	00	LOAD #00	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR 00
75	00		
76	48	STORE 73	RANGER LA VALEUR 00 À L'ADRESSE 73 (Z)
77	73		
78	40	LOAD 71	CHARGER LA VALEUR DE L'ADRESSE 71 (X)
79	71		
7A	12	BRZ 88	SI X = 0 ALORS PC ← 88
7B	88		
7C	21	SUB #02	SOUSTRAIRE 1 À X
7D	01		
7E	48	STORE 71	RANGER LE RÉSULTAT DE LA SOUSTRACTION À L'ADRESSE 71 (X)
7F	71		
80	40	LOAD 73	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE L'ADRESSE 73 (Z)
81	73		
82	60	ADD 72	ADDITIONNER Z ET Y
83	72		
84	48	STORE 73	RANGER LA VALEUR DE L'ADDITION À L'ADRESSE 73 (Z)
85	73		
86	10	JUMP 78	SAUT À L'ADRESSE 78
87	78		
88	40	LOAD 73	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR DE 73 (Z)
89	73		
8A	48	STORE 71	RANGER LA VALEUR DE Z À L'ADRESSE 71 (X)
8B	71		
8C	10	JUMP 00	SAUT À L'ADRESSE 5E DANS UN PREMIER TEMPS PUIS EN 6C
8D	00		

```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
(base) [patrice@g3-3590 cx25.0]$ ./a.out paper13.9.hexcode
chargement du boot...
boot OK! démarrage du programme...
PC : 050 | A : 000 | IN 70      data[70] = input(val ?)
Saisir une valeur en décimal : 3

PC : 052 | A : 000 | LOAD 70    A = data[70]
PC : 054 | A : 003 | STORE71    data[71] = A
PC : 056 | A : 003 | STORE72    data[72] = A
PC : 058 | A : 003 | LOAD #5E   A = 5E
PC : 05A | A : 094 | STORE8D    data[8D] = A
PC : 05C | A : 094 | JUMP 74    PC = 74
PC : 074 | A : 094 | LOAD #00    A = 00
PC : 076 | A : 000 | STORE73    data[73] = A
PC : 078 | A : 000 | LOAD 71    A = data[71]
PC : 07A | A : 003 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 003 | SUB #01    A -= 01
PC : 07E | A : 002 | STORE71    data[71] = A
PC : 080 | A : 002 | LOAD 73    A = data[73]
PC : 082 | A : 000 | ADD 72     A += data[72]
PC : 084 | A : 003 | STORE73    data[73] = A
PC : 086 | A : 003 | JUMP 78    PC = 78
PC : 078 | A : 003 | LOAD 71    A = data[71]
PC : 07A | A : 002 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 002 | SUB #01    A -= 01
PC : 07E | A : 001 | STORE71    data[71] = A
PC : 080 | A : 001 | LOAD 73    A = data[73]
PC : 082 | A : 003 | ADD 72     A += data[72]
PC : 084 | A : 006 | STORE73    data[73] = A
PC : 086 | A : 006 | JUMP 78    PC = 78
PC : 078 | A : 006 | LOAD 71    A = data[71]
PC : 07A | A : 001 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 001 | SUB #01    A -= 01
PC : 07E | A : 000 | STORE71    data[71] = A
PC : 080 | A : 000 | LOAD 73    A = data[73]
PC : 082 | A : 006 | ADD 72     A += data[72]
PC : 084 | A : 009 | STORE73    data[73] = A
PC : 086 | A : 009 | JUMP 78    PC = 78
PC : 078 | A : 009 | LOAD 71    A = data[71]
PC : 07A | A : 000 | BRZ 88     Si A = 0 alors PC = 88
PC : 088 | A : 000 | LOAD 73    A = data[73]
PC : 08A | A : 009 | STORE71    data[71] = A
PC : 08C | A : 009 | JUMP 5E    PC = 5E
PC : 05E | A : 009 | LOAD 71    A = data[71]
PC : 060 | A : 009 | STORE72    data[72] = A
PC : 062 | A : 009 | LOAD 70    A = data[70]
PC : 064 | A : 003 | STORE71    data[71] = A
PC : 066 | A : 003 | LOAD #6C   A = 6C
PC : 068 | A : 108 | STORE8D    data[8D] = A
PC : 06A | A : 108 | JUMP 74    PC = 74
PC : 074 | A : 108 | LOAD #00    A = 00
PC : 076 | A : 000 | STORE73    data[73] = A
PC : 078 | A : 000 | LOAD 71    A = data[71]
PC : 07A | A : 003 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 003 | SUB #01    A -= 01
```

FIGURE 8 – Exécution du programme - Calcul de 3 au cube

```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0

PC : 07E | A : 002 | STORE71    data[71] = A
PC : 080 | A : 002 | LOAD 73    A = data[73]
PC : 082 | A : 000 | ADD 72     A += data[72]
PC : 084 | A : 009 | STORE73    data[73] = A
PC : 086 | A : 009 | JUMP 78    PC = 78
PC : 078 | A : 009 | LOAD 71    A = data[71]
PC : 07A | A : 002 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 002 | SUB #01    A -= 01
PC : 07E | A : 001 | STORE71    data[71] = A
PC : 080 | A : 001 | LOAD 73    A = data[73]
PC : 082 | A : 009 | ADD 72     A += data[72]
PC : 084 | A : 018 | STORE73    data[73] = A
PC : 086 | A : 018 | JUMP 78    PC = 78
PC : 078 | A : 018 | LOAD 71    A = data[71]
PC : 07A | A : 001 | BRZ 88     Si A = 0 alors PC = 88
PC : 07C | A : 001 | SUB #01    A -= 01
PC : 07E | A : 000 | STORE71    data[71] = A
PC : 080 | A : 000 | LOAD 73    A = data[73]
PC : 082 | A : 018 | ADD 72     A += data[72]
PC : 084 | A : 027 | STORE73    data[73] = A
PC : 086 | A : 027 | JUMP 78    PC = 78
PC : 078 | A : 027 | LOAD 71    A = data[71]
PC : 07A | A : 000 | BRZ 88     Si A = 0 alors PC = 88
PC : 088 | A : 000 | LOAD 73    A = data[73]
PC : 08A | A : 027 | STORE71    data[71] = A
PC : 08C | A : 027 | JUMP 6C    PC = 6C
PC : 06C | A : 027 | OUT 71     print(data[71])
OUT en décimal : 27
(Continuer : O/N): n
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 9 – Suite et fin du calcul de 3 au cube

10 - (Amusant mais difficile) Écrire un programme qui met la plus grande partie possible de la mémoire à 0 (en supposant qu'il y a pas de mémoire morte).

UNE PREMIÈRE VERSION SÉQUENTIELLE ET AUTOMATIQUE :

ADRESSE	VALEUR	MNÉMONIQUES	COMMENTAIRE
00	40	LOAD 0C	CHARGER LE REGISTRE DE L'ACCUMULATEUR AVEC LA VALEUR A L'ADRESSE 0C
01	0C		
02	20	ADD#1	AJOUTER LA VALEUR 01 À LA VALEUR DE L'ACCUMULATEUR
03	01		
04	48	STORE 0C	RANGER LE RÉSULTAT À L'ADRESSE 0C
05	0C		
06	00	LOAD #00	CHARGER LA VALEUR 00 DANS LE REGISTRE DE L'ACCUMULATEUR
07	00		
08	C8	STORE *0C	RANGER LE RÉSULTAT À L'ADRESSE DONT LA VALEUR SE TROUVE À L'ADRESSE 0C
09	0C		
0A	10	JUMP 00	SAUT À L'ADRESSE 00
0B	00		
0C	0C		

Ce programme revient à :

Initialisation au préalable de l'adresse_0C à 0C

```
while (1)
    0C = 0C + 1
    *0C = 00
```

Quelques remarques :

Ce programme occupe 13 adresses mémoires, les suivantes sont initialisées à zéro jusqu'à ce que l'exécution du programme provoque l'affichage d'un message d'erreur de segmentation lors du chargement de la 33624^{ème} adresse, d'après la capture d'écran en page suivante. En effet, ce plantage est lié au fait que ce programme a tenté d'accéder à un emplacement mémoire qui ne lui était pas alloué.


```
patrice@g3-3590:~/Documents/Prog imp/Prog C/cx25/cx25.0
PC : 000 | A : 000 | LOAD 0C    A = data[0C]
PC : 002 | A : 33620 | ADD #01   A += 01
PC : 004 | A : 33621 | STORE 0C   data[0C] = A
PC : 006 | A : 33621 | LOAD #00   A = 00
PC : 008 | A : 000 | STORE *0C  data[data[0C]] = A
PC : 00A | A : 000 | JUMP 00    PC = 00
PC : 000 | A : 000 | LOAD 0C    A = data[0C]
PC : 002 | A : 33621 | ADD #01   A += 01
PC : 004 | A : 33622 | STORE 0C   data[0C] = A
PC : 006 | A : 33622 | LOAD #00   A = 00
PC : 008 | A : 000 | STORE *0C  data[data[0C]] = A
PC : 00A | A : 000 | JUMP 00    PC = 00
PC : 000 | A : 000 | LOAD 0C    A = data[0C]
PC : 002 | A : 33622 | ADD #01   A += 01
PC : 004 | A : 33623 | STORE 0C   data[0C] = A
PC : 006 | A : 33623 | LOAD #00   A = 00
PC : 008 | A : 000 | STORE *0C  data[data[0C]] = A
PC : 00A | A : 000 | JUMP 00    PC = 00
PC : 000 | A : 000 | LOAD 0C    A = data[0C]
PC : 002 | A : 33623 | ADD #01   A += 01
PC : 004 | A : 33624 | STORE 0C   data[0C] = A
PC : 006 | A : 33624 | LOAD #00   A = 00
Erreur de segmentation (core dumped)
(base) [patrice@g3-3590 cx25.0]$
```

FIGURE 10 – Exécution du programme paper.13.10.hexcode

Cette capture d'écran m'a fait pensé à une autre solution :

UNE DEUXIÈME VERSION SÉQUENTIELLE MAIS AVEC UNE SAISIE MANUELLE :

ADRESSE	VALEUR	MNÉMONIQUES	COMMENTAIRE
00	C9	IN * 8357	SAISIR À L'ADRESSE CORRESPONDANT À LA VALEUR DE L'ADRESSE 8357(33623)
01	8357		
02	40	LOAD 8357	CHARGER LA VALEUR DE L'ADRESSE 8357
03	8357		
04	21	SUB #1	DÉCRÉMENTER DE 1 LA VALEUR DE L'ACCUMULATEUR
05	01		
06	48	STORE 8357	RANGER LA VALEUR DE L'ACCUMULATEUR À L'ADRESSE 8357
07	10	JUMP 00	SAUT À L'ADRESSE 00
08	00		
.			
.			
8357	8356		

Cette nouvelle version est implémentée sur seulement dix adresses et permet d'initialiser à zéro à partir de l'adresse 8356 (33623) jusque l'adresse 06 incluse.

- 11 - (Long, pas de corrigé) Écrire un programme d'extraction de la racine carré d'un nombre, avec la méthode de Newton pour l'ordinateur en papier.
- 12 - Programme réalisé en C et correspondant à l'exercice cx25.0 du cours de programmation impérative

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# define DEBUG if (0) fprintf

//Déclaration, définitions et initialisation
enum{MAX_MEM = 256};
// Le nombre d'index présent en mémoire
unsigned int *data;
// Un vecteur d'entiers qui seront enregistrés les opcodes et leur valeur

// Les prototypes de fonctions :
void usage(char *, char *);
unsigned int charger_fichier(char *);
void run(unsigned int);
void commandes(unsigned int, unsigned int, int *, unsigned int *);
int oui_non();

//Traitement du programme :
int main(int argc, char *argv[]){
    // Charge un bootstrap et un fichier donné en argv[1]
    data = malloc(sizeof(unsigned int *) * MAX_MEM);
    // Allocation du vecteur data de taille MAX_MEM
    if (argc != 2) usage("", "Usage : <nom du programme> [-d] <nom du fichier >\n");
    puts("Chargement du bootstrap...");
    //run(charger_fichier("paperboot.hexcode"));
    // Chargement du bootloader
    puts("Chargement du bootstrap terminé !\nDémarrage du programme...\n");
    run(charger_fichier(argv[1]));
    // Chargement du programme
    return 0;
}

void usage(char *nom_fichier, char *message){
    // Affichage un message d'erreur
    perror(nom_fichier);
    fprintf(stderr, message);
    exit(1);
}

unsigned int charger_fichier(char * fichier){
    // Charger un fichier et initialise le vecteur data
    unsigned int offset;
    int i = 0;
    FILE *flux = fopen(fichier, "r");
    // Ouverture du fichier
    if (!flux)
        usage(fichier, "Une erreur s'est produite lors de l'ouverture du fichier\n");
    if (fscanf(flux, "%*s %X %*s", &offset) == EOF)
        usage(fichier, "Une erreur s'est produite lors de la lecture du fichier\n");
    // Récupérer l'offset en ligne 2 dans le fichier
    while(!feof(flux)) fscanf(flux, "%X", &data[offset + i++]);
    // Enregistrer les nombres sous forme hexadécimale dans le vecteur data
    if (fclose(flux))
```

```

        usage(fichier, "Echec lors de la fermeture du fichier, voir la doc\n");
    for (int i = 0; i < 256; ++i) DEBUG(stderr, "%i.%X\n", i, data[i]);
    return offset;
}

void run (unsigned int PC){
    // Envoyer les opcodes présents dans le vecteur data à la fonction commandes pour interprétation
    int A = 0;
    // Variable simulant la valeur du registre de l'accumulateur (A)
    int *ptr_A = &A;
    unsigned int *ptr_PC = &PC;
    // un pointeur sur PC, simulant la valeur du Program Counter
    while(PC < MAX_MEM){
        // Boucler sur les opcodes présent dans le vecteur data
        printf("PC : %03X | A : %03i | ", PC, A);
        PC += 2;
        // Incrémenter de 2 pour avoir accès à la prochaine instruction du Programm Counter
        commandes(data[PC - 2], data[(PC - 2) + 1], ptr_A, ptr_PC);
        // Envoyer les opcodes et leur valeur à la fonction commandes
        printf("\n");
    }
}

void commandes(unsigned int opcode, unsigned int arg, int *ptr_A, unsigned int *ptr_PC){
    // Interpréter les opcodes reçus en argument et modifie en conséquence les pointeurs A et PC
    switch (opcode){
        case 0x00:
            printf("LOAD #%02X    A = %02X", arg, arg);
            *ptr_A = arg;
            break;
        case 0x10:
            printf("JUMP %02X    PC = %02X", arg, arg);
            *ptr_PC = arg;
            break;
        case 0x11:
            printf("BRN %02X    Si A < 0 alors PC = %02X", arg, arg);
            if (*ptr_A < 0)
                *ptr_PC = arg;
            break;
        case 0x12:
            printf("BRZ %02X    Si A = 0 alors PC = %02X", arg, arg);
            if (!*ptr_A)
                *ptr_PC = arg;
            break;
        case 0x20:
            printf("ADD #%02X    A += %02X", arg, arg);
            *ptr_A += arg;
            break;
        case 0x21:
            printf("SUB #%02X    A -= %02X", arg, arg);
            *ptr_A -= arg;
            break;
        case 0x22:
            printf("NAND #%02X    A = ~[A & %02X]", arg, arg);
            *ptr_A = (~(*ptr_A & arg))% MAX_MEM;
            break;
        case 0x40:
            printf("LOAD %02X    A = data[%02X]", arg, arg);
            *ptr_A = data[arg];
            break;
        case 0x41:
            printf("OUT %02X    print(data[%02X])\nOUT en décimal : %i\n", arg, arg, data[arg]);
            if (!oui_non())
    
```

```

        exit(0);
    else break;
case 0x48:
    printf("STORE %02X    data[%02X] = A", arg, arg);
    data[arg] = *ptr_A;
    break;
case 0x49:
    printf("IN %02X      data[%02X] = input(val ?)\nSaisir une valeur en décimal : ", arg, arg);
    if (!scanf("%i", &data[arg]))
        usage("", "Erreur de saisie, la valeur doit être un chiffre\n");
    break;
case 0x60:
    printf("ADD %02X      A += data[%02X]", arg, arg);
    *ptr_A += data[arg];
    break;
case 0x61:
    printf("SUB %02X      A -= (data[%02X]", arg, arg);
    *ptr_A -= data[arg];
    break;
case 0x62:
    printf("NAND %02X     A = ~(A & %02X)", arg, arg);
    *ptr_A = (~(*ptr_A & data[arg]))% MAX_MEM;
    break;
case 0xC0:
    printf("LOAD %02X     A = data[data[%02X]]", arg, arg);
    *ptr_A = data[data[arg]];
    break;
case 0xC1:
    printf("OUT %02X      print(data[data[%02X]]\nOUT en décimal – %i\n)", arg, arg, data[data[arg]]);
    if (!oui_non())
        exit(0);
    else break;
case 0xC8:
    printf("STORE %02X  data[data[%02X]] = A", arg, arg);
    data[data[arg]] = *ptr_A;
    break;
case 0xC9:
    printf("IN %02X      data[data[%02X]] = input(val ?)\n Saisir une valeur en décimal : ",
arg, arg);
    puts("val (en décimal) : ");
    if (!scanf("%i", &data[data[arg]]))
        usage("", "Erreur de saisie : la valeur doit être un chiffre\n");
    break;
case 0xE0:
    printf("ADD %02X     A += data[data[%02X]]", arg, arg);
    *ptr_A += data[data[arg]];
    break;
case 0xE1:
    printf("SUB %02X     A -= data[data[%02X]]", arg, arg);
    *ptr_A -= data[data[arg]];
    break;
case 0xE2:
    printf("NAND %02X    A = ~(A & data[data[%02X]])", arg, arg);
    *ptr_A = (~(*ptr_A & data[data[arg]]))% MAX_MEM;
    break;
default:
    usage("", "Erreur : commande inexistante"); exit(0);
}
}

int oui_non() {
    // Cette fonction permet d'arrêter ou de continuer le programme ou autoriser une option
    char option;

```

```
while (1) {  
    printf("(Continuer : O/N): ");  
    if (scanf(" %c", &option) != 1)  
        usage("", "Une erreur s'est produite lors de la lecture de l'option\n");  
    else if (option == 'o' || option == 'O') return 1;  
    else if (option == 'n' || option == 'N') return 0;  
    else printf("Seulement O ou N sont acceptables.\n");  
}
```

```
}
```

13 - Programme réalisé en C et correspondant à l'exercice cx25.1 (STEPPER) du cours de programmation impérative

Le STEPPER ajouté au code du cx25.0 comporte les options suivantes :

- display all : affiche l'ensemble du programme.
- display <adresse> : affiche la valeur enregistré à l'adresse demandée.
- quit : permet de quitter le programme.
- help : affiche l'aide.
- store <adresse> : Enregistre une nouvelle valeur à l'adresse précisée et créer un fichier dénommé new_version dans le répertoire, suite à cet enregistrement

Pour activer le stepper, il suffit d'exécuter le programme de cette manière : ./cx25.1 -d <nom_fichier>

J'ai ajouté une documentation utilisateur très synthétique avec le programme.

LE CODE SOURCE DU PROGRAMME CX25.1

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# define DEBUG if (0) fprintf

//Déclaration, définitions et initialisation
enum{MAX_MEM = 256};
// Le nombre d'index présent en mémoire
unsigned int *data;
// Un vecteur d'entiers oui seront enregistrés les opcodes et leur valeur
unsigned int taille = 0;
// La taille du programme lu dans un fichier
unsigned int *ptr_taille = &taille;
// Un pointeur sur taille

// Les prototypes de fonctions :
void traiter_ldc(int, char **);
void usage(char *, char *);
unsigned int charger_fichier(char *, unsigned int *);
void run(unsigned int, int);
void commandes(unsigned int, unsigned int, int *, unsigned int *);
int oui_non();
void help();
int stepper(unsigned int);
void enregistrer_fichier(unsigned int);

//Traitement du programme :
int main(int argc, char *argv[]){
    // Chargement du bootloader, et d'un programme passé en argument via la LDC
    data = malloc(sizeof(unsigned int *) * MAX_MEM);
    // Allocation du vecteur data de taille MAX_MEM
    puts("Chargement du bootstrap...");
    //run(charger_fichier("paperboot.hexcode", ptr_taille), step_on);
    // Chargement du bootloader
    puts("Chargement du bootstrap terminé !\nDémarrage du programme...");
    traiter_ldc(argc, argv);
    // chargement du programme et de ses options suivant les arguments de la ldc
    return 0;
}
```

```

void traiter_ldc(int argc, char ** argv){
    // Traitement des différents possible cas en ligne de commande
    int step_on;
    switch (argc){
        case 1 :
            usage("", "Usage : <nom du programme> [-d] <nom du fichier>\n"); break;
        case 2 :
            if (!strcasecmp(argv[1], "-d"))
                usage("", "Usage : <nom du programme> [-d] <nom du fichier>\n");
            else {
                step_on = 0;
                run(charger_fichier(argv[1], ptr_taille), step_on);
                // Chargement du programme sans stepper
            }
            break;
        case 3 :
            if (!strcasecmp(argv[1], "-d")){
                step_on = 1;
                run(charger_fichier(argv[2], ptr_taille), step_on);
                // Chargement du programme avec stepper
            }
            else
                usage("", "Usage : <nom du programme> [-d] <nom du fichier>\n");
        default :
            usage("", "Usage : <nom du programme> [-d] <nom du fichier>\n");
    }
}

```

```

void usage(char *nom_fichier, char *message){
    // Affichage un message d'erreur
    perror(nom_fichier);
    fprintf(stderr, message);
    exit(1);
}

```

```

unsigned int charger_fichier(char * fichier, unsigned int *ptr_taille){
    // Charger un fichier et initialise le vecteur data
    unsigned int offset;
    unsigned int i = 0;
    FILE *flux = fopen(fichier, "r");
    // Ouverture du fichier
    if (!flux) usage(fichier, "Une erreur s'est produite lors de l'ouverture du fichier\n");
    if (fscanf(flux, "%*s %X %*s", &offset) == EOF)
        usage(fichier, "Une erreur s'est produite lors de la lecture du fichier\n");
    // Récupérer l'offset en ligne 2 dans le fichier
    while (!feof(flux)) fscanf(flux, "%X", &data[offset + i++]);
    // Enregistrer les nombres sous forme hexadécimale dans le vecteur data
    *ptr_taille = i - 1;
    // Assigne la taille du programme
    if (fclose(flux))
        usage(fichier, "Echec lors de la fermeture du fichier, voir la doc\n");
    for (int i = 0; i < 256; ++i) DEBUG(stderr, "%i.%X\n", i, data[i]);
    return offset;
}

```

```

void run (unsigned int PC, int step_on){
    // Envoyer les opcodes présents dans le vecteur data à la fonction commandes pour interprétation
    int A = 0;
    // Variable simulant la valeur du registre de l'accumulateur (A)
    int *ptr_A = &A;
    unsigned int cp_PC = PC;
    // une copie de la valeur simulant celle du Program Counter (PC)
    unsigned int *ptr_PC = &PC;
}

```

```

puts("gdb - cx25.1\nVoulez-vous afficher la valeur de PC?");
int test_PC = oui_non();
// Affichage ou non de PC
puts("Voulez-vous afficher les la valeur de A?");
int test_A = oui_non();
// Affichage ou non de A
if (step_on){puts("Voulez-vous afficher l'aide avant de commander ?"); if (oui_non()) help();}
puts("\nDébut du programme :\n");
getchar();
while(PC < MAX_MEM + 1){
// Boucler sur les opcodes présent dans le vecteur data

    if (test_PC && !test_A) printf("PC : %03i | ", PC);
    else if (test_A && !test_PC) printf("A : %03i | ", A);
    else if (!test_A && !test_PC) ;
    else printf("PC : %03i | A : %03i | ", PC, A);
    PC += 2;
    // Incrémentation de 2 pour avoir accès à la prochaine instruction du Program Counter
    commandes(data[PC -2], data[(PC -2) + 1], ptr_A, ptr_PC);
    // Envoyer les opcodes et leur valeur à la fonction commandes
    printf("\n");
    if (step_on)stepper(cp_PC);
    // Affichage du stepper si step_on == 1

}
}

int oui_non() {
// Cette fonction permet d'arrêter ou de continuer le programme, ou d'autoriser une option
char option;
while (1) {
    printf("(Continuer : O/N): ");
    if (scanf(" %c", &option) != 1)
        usage("", "Une erreur s'est produite lors de la lecture de l'option\n");
    else if (option == 'o' || option == 'O') return 1;
    else if (option == 'n' || option == 'N') return 0;
    else printf("Seulement O ou N sont acceptables.\n");
}
}

void help(){
// Cette fonction affiche l'aide
puts("\n<Bienvenue dans l'aide de gdb cx35.1>\n");
puts("Voici la liste des commandes : ");
puts("\ndisplay <adresse>\nSignification : affiche la valeur à l adresse désignée.\nExemple d u");
puts("display all\nSignification : affiche toutes les valeurs de toutes les adresses du program");
printf("store <adresse> <valeur>;\nSignification : enregistre une valeur saisie en hexadecimal");
printf("l adresse designee. ");
printf("Un fichier denomme : new_version est automatiquement cree avec la modification apportee");
puts("quit");
puts("Signification : Arrêt du programme.\n");
puts("help");
puts("Signification : afficher l'aide.");
}

void commandes(unsigned int opcode, unsigned int arg, int *ptr_A, unsigned int *ptr_PC){
// Interpréter les opcodes reçus en argument et modifie en conséquence les pointeurs A et PC
switch (opcode){
case 0x00:
    printf("LOAD #%02X A = %02X", arg, arg);
    *ptr_A = arg;
    break;
case 0x10:

```



```

printf("JUMP %02X      PC = %02X", arg, arg);
* ptr_PC = arg;
break;
case 0x11:
printf("BRN %02X      Si A < 0 alors PC = %02X", arg, arg);
if (*ptr_A < 0)
    *ptr_PC = arg;
break;
case 0x12:
printf("BRZ %02X      Si A = 0 alors PC = %02X", arg, arg);
if (!*ptr_A)
    *ptr_PC = arg;
break;
case 0x20:
printf("ADD #%02X      A += %02X", arg, arg);
*ptr_A += arg;
break;
case 0x21:
printf("SUB #%02X      A -= %02X", arg, arg);
*ptr_A -= arg;
break;
case 0x22:
printf("NAND #%02X     A = ~[A & %02X]", arg, arg);
*ptr_A = (~(*ptr_A & arg))% MAX_MEM;
break;
case 0x40:
printf("LOAD %02X      A = data[%02X]", arg, arg);
*ptr_A = data[arg];
break;
case 0x41:
printf("OUT %02X      print(data[%02X])\nOUT en décimal : %i\n",arg, arg, data[arg]);
if (!oui_non())
    exit(0);
else break;
case 0x48:
printf("STORE %02X     data[%02X] = A", arg, arg);
data[arg] = *ptr_A;
break;
case 0x49:
printf("IN %02X       data[%02X] = input(val ?)\nSaisir une valeur en décimal : ", arg, arg);
if (!scanf("%i", &data[arg]))
    usage("", "Erreur de saisie, la valeur doit être un chiffre\n");
break;
case 0x60:
printf("ADD %02X      A += data[%02X]", arg, arg);
*ptr_A += data[arg];
break;
case 0x61:
printf("SUB %02X      A -= (data[%02X]", arg, arg);
*ptr_A -= data[arg];
break;
case 0x62:
printf("NAND %02X     A = ~[A & %02X]", arg, arg);
*ptr_A = (~(*ptr_A & data[arg]))% MAX_MEM;
break;
case 0xC0:
printf("LOAD *%02X     A = data[data[%02X]]", arg, arg);
*ptr_A = data[data[arg]];
break;
case 0xC1:
printf("OUT *%02X      print(data[data[%02X])\nOUT en décimal – %i\n)", arg, arg, data[data[a
if (!oui_non())
    exit(0);

```

```

        else break;
    case 0xC8:
        printf("STORE %%02X  data[data[%02X]] = A", arg, arg);
        data[data[arg]] = *ptr_A;
        break;
    case 0xC9:
        printf("IN %%02X)    data[data[%02X]] = input(val ?)\n Saisir une valeur en décimal : ",
arg, arg);
        puts("val (en décimal) : ");
        if (!scanf("%i", &data[data[arg]]))
            usage("", "Erreur de saisie : la valeur doit être un chiffre\n");
        break;
    case 0xE0:
        printf("ADD %%02X    A += data[data[%02X]]", arg, arg);
        *ptr_A += data[data[arg]];
        break;
    case 0xE1:
        printf("SUB %%02X    A -= data[data[%02X]]", arg, arg);
        *ptr_A -= data[data[arg]];
        break;
    case 0xE2:
        printf("NAND %%02X  A = ~(A & data[data[%02X]])", arg, arg);
        *ptr_A = (~(*ptr_A & data[data[arg]]))% MAX_MEM;
        break;
    default:
        usage("", "Erreur : commande inexistante\n"); exit(0);
}
}

```

```

int stepper(unsigned int cp_PC){
// Une fonction qui developpe quelques fonctionnalités de gdb
char phrase[20];
fgets(phrase, 20, stdin);
// Saisir une commande
int i = 0;
char *mot[3];
char * token = strtok(phrase, " ");
// split la phrase en un ou plusieurs mot

while(token != NULL){
    mot[i] = token;
    token = strtok(NULL, " ");
    i++;
}
if (!strcmp(mot[0], "quit\n") && i == 1) exit(0);
// Si la commande saisie est : quit, le programme s'arrête
else if (!strcmp(mot[0], "display") & i == 2){

    if (!strcasecmp(mot[1], "all\n")) {
        // Si la commande saisie est : display all, le programme affiche l'ensemble du programme
        // présent dans le vecteur data
        unsigned int j=cp_PC;
        for (int i = cp_PC; i < (cp_PC + *ptr_taille) ; ++i)
            fprintf(stdout, "data[%i] == %X\n", i, data[i]);
        getchar();
    }
    else{
        int adresse;
        adresse = strtoul(mot[1], NULL, 10);
        fprintf(stderr, "%X\n", data[adresse]);
        // si la commande saisie est display suivie d'une adresse, le programme
        // affiche la valeur à l'adresse concerné
        getchar();
    }
}
}

```

```

        return 0;
    }
}
else if (!strcmp(mot[0], "store") && i == 3){
    // Si la commande saisie est store suivie d'une adresse, le programme enregistre la valeur
    // à l'adresse choisie
    int adresse2;
    adresse2 = strtoul(mot[1], NULL, 10);
    unsigned int valeur = strtoul(mot[2], NULL, 16);
    data[adresse2] = valeur;
    enregistrer_fichier(cp_PC);
    // Suite à la modification de l'adresse, l'ensemble du vecteur data est enregistré
    // dans un nouveau fichier
    getchar();
}
else if (!strcasecmp(mot[0], "help\n") && i == 1){
    // Si la commande saisie est help, le programme affiche l'aide
    help();}
else;
}

void enregistrer_fichier (unsigned int cp_PC){
    // Cette fonction enregistre le vecteur data dans un fichier dénommé : new_version
    FILE* flux = fopen("new_version","w+");
    if (!flux)
        usage ("","Une erreur s'est produite lors de l'ouverture du fichier\n");
    fprintf(flux, "%s\n%X\n%s\n", "offset", cp_PC,"code");
    for (int i = cp_PC; i < (cp_PC + (taille - 1)); i = i + 2)
        fprintf(flux, "%X %X\n", data[i], data[i + 1]);
    if (fclose(flux))
        usage("new_version", "Echec lors de la fermeture du fichier, voir la doc\n");
}

```