

ELIE MENDY

ID : 19004664

17 Juin 2021

---

## **CHAPITRE 10 - Architecture des Machines**

---

# Table des matières

<b>1</b>	<b>Exercice 10.1</b>	<b>3</b>
<b>2</b>	<b>Exercice 10.2</b>	<b>4</b>
<b>3</b>	<b>Exercice 10.3</b>	<b>5</b>
<b>4</b>	<b>Exercice 10.4</b>	<b>5</b>
<b>5</b>	<b>Exercice 10.5</b>	<b>6</b>
<b>6</b>	<b>Exercice 10.6</b>	<b>9</b>
<b>7</b>	<b>Exercice 10.7</b>	<b>12</b>
<b>8</b>	<b>Emulateur : Ordinateur en Papier (code du programme) :</b>	<b>15</b>
8.1	Explication du projet : . . . . .	15
8.1.1	Précautions d'utilisation : . . . . .	16
8.2	librairies - Constantes préprocesseur - typedef - Variables Globales . . . . .	17
8.3	fonction main . . . . .	18
8.4	fonctions : outils du main . . . . .	20
8.4.1	usage() . . . . .	20
8.4.2	stepper() . . . . .	20
8.4.3	afficherMemoire() . . . . .	20
8.4.4	afficherProgramme() . . . . .	21
8.4.5	initialiserRegistres() . . . . .	21
8.4.6	lireProgramme() . . . . .	22
8.4.7	chargerBootstrap() . . . . .	22
8.4.8	chargerProgramme() . . . . .	23
8.4.9	executer() . . . . .	23
8.5	Fonctions Microcodes . . . . .	27
8.5.1	transfert() . . . . .	27
8.5.2	prepaCalcul() . . . . .	28
8.5.3	hexaToInt() . . . . .	28
8.5.4	intToHexa() . . . . .	29
8.5.5	intTostr() . . . . .	29

8.5.6	calcul()	30
8.5.7	lireMemoire()	31
8.5.8	ecrire()	31
8.5.9	saisir()	32
8.5.10	lire()	32
8.6	Fonctions de déroulement d'un cycle	33
8.6.1	phase1()	33
8.6.2	phase 3	33
8.6.3	incrémenterPC()	33
8.7	Fonction D'Opérations	34
8.7.1	add()	34
8.7.2	addP()	34
8.7.3	addPP()	35
8.7.4	sub()	35
8.7.5	subP()	36
8.7.6	subPP()	36
8.7.7	nand()	37
8.7.8	nandP()	37
8.7.9	nandPP()	38
8.7.10	load()	38
8.7.11	loadP()	39
8.7.12	loadPP()	39
8.7.13	storeP()	40
8.7.14	storePP()	40
8.7.15	inP()	41
8.7.16	inPP()	41
8.7.17	outP()	42
8.7.18	outPP()	42
8.7.19	jump()	43
8.7.20	brn()	43
8.7.21	brz()	44

## 1 Exercice 10.1

### A RENDRE

*Il manque les microcodes de la phase 2 pour le cas où OP contient E62. Complétez cette série de microcodes.*

On construira l'Opération NAND selon les actions suivantes :

code	signification
17	Ecrire <b>NAND</b> dans UAL
1	Ciblage d'une case mémoire PC->RS
13	Lecture de la valeur dans la case mémoire
6	Sauvegarde de la valeur dans le registre AD
7	Ciblage d'une case mémoire AD->RS
13	Lecture de la valeur dans la case mémoire
6	Sauvegarde de la valeur dans le registre AD
7	Ciblage d'une case mémoire AD->RS
13	Lecture de la valeur dans la case mémoire
12	Effectuer l'Opération Afficher dans UAL (ici : <b>NAND</b> )

L'Opération NAND \* correspondra à la suite de microcode suivante :

**17 - 1 - 13 - 6 - 7 - 6 - 13 - 6 - 7 - 13 - 12**

## 2 Exercice 10.2

### A RENDRE

Étant donné la logique qui existe entre les codes opératoires et le microcode... A quelle code pourrait correspondre l'instruction IN ?

Étant donné la logique qui existe entre les opérations :

49 -> 49 1 13 6 7 16 8 14

C9 -> 1 13 6 7 13 6 7 16 8 14

Correspondance des actions :

code	signification
1	Ciblage d'une case mémoire PC->RS
16	Capturer une saisie utilisateur
8	Sauvegarde de la valeur saisie dans le registre RM
14	Ecriture de la valeur saisie dans la case mémoire ciblée dans RS

L'Opération IN # correspondra à la suite de microcode suivante :

**1 - 16 - 8 - 14**

(On supprime les indirections successives que représentent les enchainements des microcodes 13 - 6 - 7).

### 3 Exercice 10.3

#### A RENDRE

Étant donné la logique qui existe entre les codes opératoires et le microcode... A quelle opération pourrait correspondre le code 01 ?

Étant donné la logique qui existe entre les code opératoires :

Mnémonique	Code
LOAD	00
LOAD $\alpha$	40
LOAD * $\alpha$	C0
STORE $\alpha$	48
STORE * $\alpha$	C8
IN $\alpha$	49
IN * $\alpha$	C9
XXX	01
OUT $\alpha$	41
OUT * $\alpha$	C1

L'Opération **OUT** # correspondra à l'opération 01 :

### 4 Exercice 10.4

#### A RENDRE

Dans le programme de bootstrap, les adresses 18 à 1E sont rem- plies de 00. Que se passe-t-il au cours du bootstrap quand l'ordinateur exécute ces instructions ?

Sauf erreur de ma part, il s'agit d'une partie de la mémoire qui n'est pas parcourue par le l'ordinateur à l'exécution du bootstrap étant donné qu'un jump est effectué à l'adresse indiqué par la case 20 à la fin de l'exécution.

De plus, il est normalement impossible d'écrire dans cette zone mémoire étant donné qu'elle représente la mémoire morte.

## 5 Exercice 10.5

### A RENDRE

*Écrire un programme pour l'ordinateur en papier qui lit deux nombres et affiche leur produit.*

**Nota :** le Programme de l'emulateur de l'ordinateur en papier est consultable dans la section 8 de ce rendu.

*Algorithme du programme :*

```
1  - x = 0
2
3  - saisir NB1
4  - si NB1 = 0:
5      - on retourne 0
6
7  - saisir NB2
8  - si NB2 = 0:
9      - on retourne 0
10
11 - tant que NB2 > 0:
12     - x += NB1
13
14 - retourner x
```

**Programme : (avec mnémonique)**

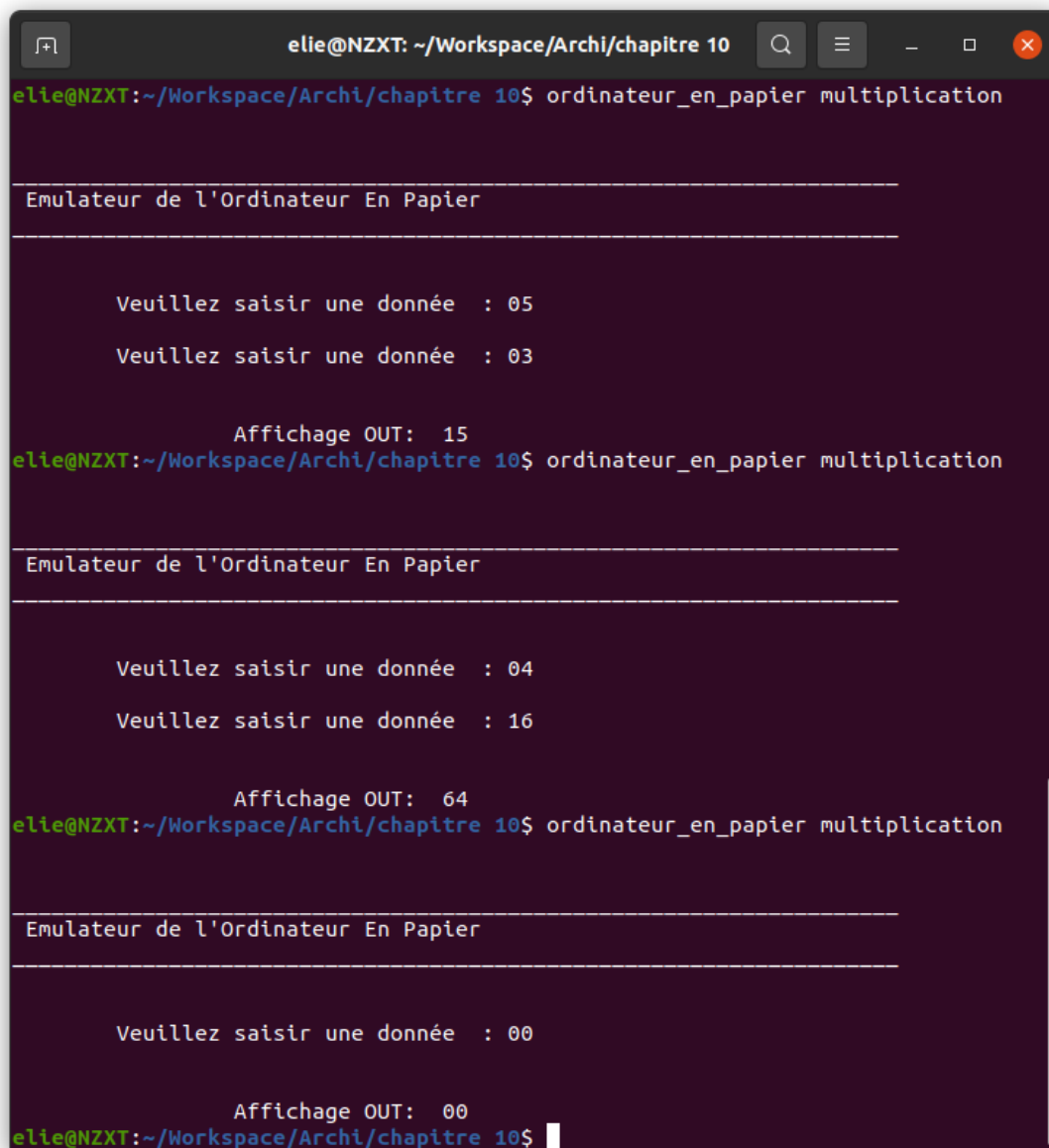
Adresse (base 10)	Adresse (hexa)	code	mnémonique	commentaire
40	28	00	LOAD 00	charger 00 dans l'accumulateur
41	29	00		
42	2A	48	STORE 60	ecrire zero dans le résultat (adresse 60)
43	2B	60		
44	2C	49	IN 61	entrer NB1
45	2D	61		
46	2E	40	LOAD 61	charger NB1 dans l'accumulateur
47	2F	61		
48	30	12	BRZ 46	si NB1 = 0, on sort en affichant le résultat (ici zéro)
49	31	46		(correspond à une multiplication par zero)
50	32	49	IN 62	entrer NB2
51	33	62		
52	34	40	LOAD 62	charger NB2 dans l'accumulateur
53	35	62		
54	36	12	BRZ 46	si NB2 = 0, on sort en affichant le résultat
55	37	46		
56	38	40	LOAD 60	charger le résultat dans l'accumulateur
57	39	60		
58	3A	60	ADD 61	ajouter NB au résultat
59	3B	61		
60	3C	48	STORE 60	écraser la valeur du résultat par sa nouvelle valeur
61	3D	60		
62	3E	40	LOAD 62	charger NB2 dans l'accumulateur
63	3F	62		
64	40	21	SUB 01	décrémenter NB2
65	41	01		
66	42	48	STORE 62	écraser la valeur de NB2 par sa nouvelle valeur
67	43	62		
68	44	10	JUMP 34	brancher sur 34 (nouveau tour de boucle)
69	45	34		
70	46	41	OUT 60	afficher le résultat
71	47	60		
72	48	10	JUMP 00	sortir du programme
73	49	6E		



**Exemple d'exécution du programme :**

***Nota : les Defines pour une tel execution sont les suivants :***

```
1 // #define DEBUG
2 // #define DEBUG_n2
3 // #define STEPPER
4 #define AUTO
5 // #define EXO
6 #define TAILLE_MEMOIRE 256
7 #define TAILLE_ADRESSE 3
8 #define TAILLE_BOOTSTRAP 32
9 #define TAILLE_MAX_PROGRAMME 200
10 #define FIRST_INSTRUCTION "28"
11 #define FIRST_INSTRUCTION_EXO_10_6 "50"
```



```
elie@NZXT: ~/Workspace/Archi/chapitre 10
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier multiplication

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 05

    Veuillez saisir une donnée  : 03

    Affichage OUT: 15
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier multiplication

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 04

    Veuillez saisir une donnée  : 16

    Affichage OUT: 64
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier multiplication

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 00

    Affichage OUT: 00
elie@NZXT:~/Workspace/Archi/chapitre 10$
```

## 6 Exercice 10.6

### A RENDRE

*En vous inspirant de l'Exercice 10.5... Écrire un programme pour l'ordinateur en papier qui lit deux nombres et affiche leur quotient "entier".*

**Nota : le Programme de l'emulateur de l'ordinateur en papier est consultable en annexe de ce rendu.**

```
1  - x = 0           // résultat
2
3  - saisir NB1      // numérateur
4  - si NB1 = 0:
5      - on retourne 0
6
7  - saisir NB2      // diviseur
8  - si NB2 = 0:
9      - on retourne 0 --> cas simplifié de la division par zero.
10
11 - y = NB1-NB2
12
13 - tant que y > 0:
14     - x ++
15     - y -= NB2
16
17 - retourner x
```

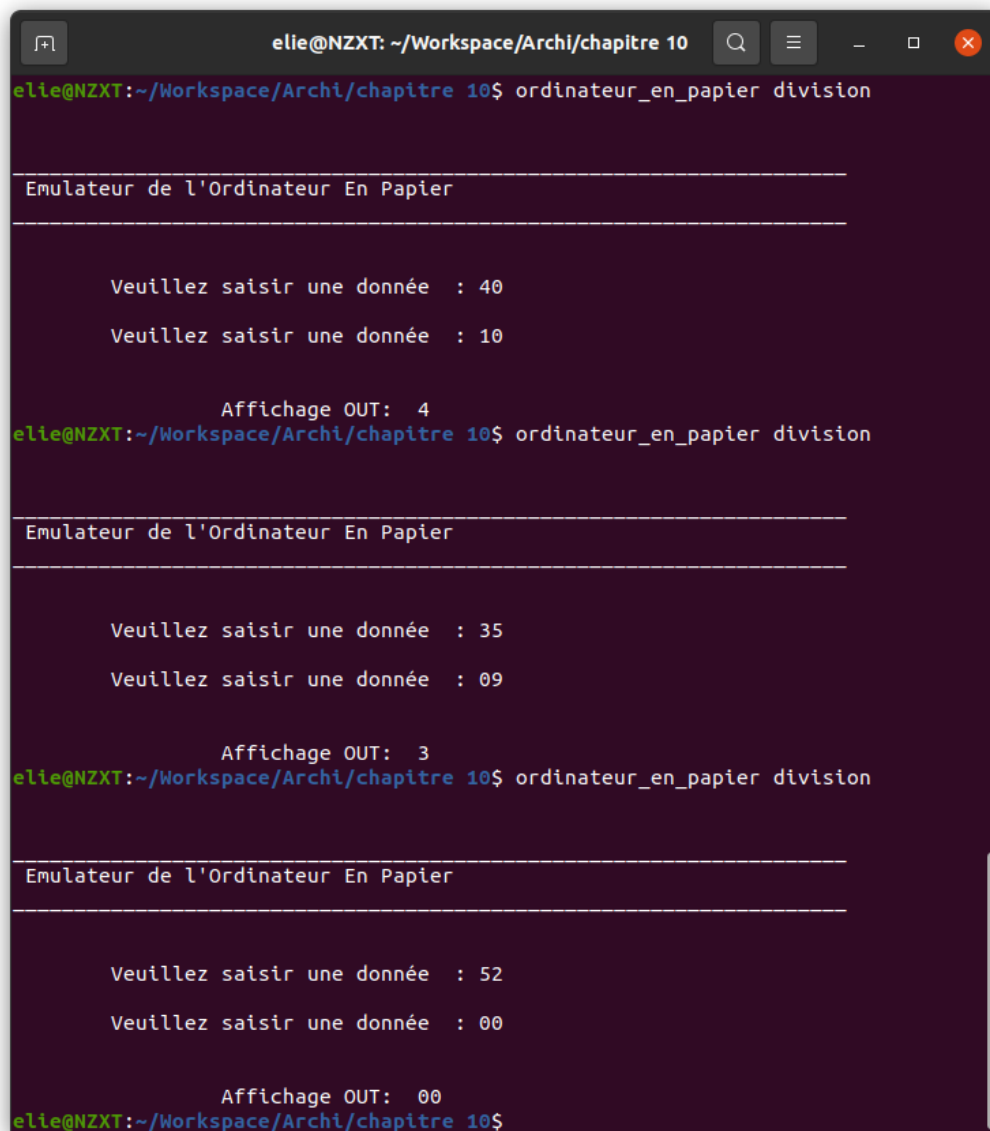
**Programme : (avec mnémonique)**

<b>Adresse</b> (base 10)	<b>Adresse</b> (hexa)	<b>code</b>	<b>mnémonique</b>	<b>commentaire</b>
40	28	00	LOAD 00	charger 00 dans l'accumulateur
41	29	00		
42	2A	48	STORE 60	ecrire zero dans le résultat (adresse 60)
43	2B	60		
44	2C	49	IN 61	entrer NB1 (adresse 61)
45	2D	61		
46	2E	40	LOAD 61	charger NB1 dans l'accumulateur
47	2F	61		
48	30	12	BRZ 50	si NB1 = 0, on sort en affichant le résultat (ici zéro)
49	31	50		(correspond à un numérateur à zero)
50	32	49	IN 62	entrer NB2
51	33	62		
52	34	40	LOAD 62	charger NB2 dans l'accumulateur
53	35	62		
54	36	12	BRZ 50	si NB2 = 0, on sort en affichant le résultat
55	37	50		
56	38	40	LOAD 61	charger NB2 dans l'accumulateur
57	39	61		
58	3A	61	SUB 62	décrémenter NB2
59	3B	62		
60	3C	48	STORE 63	ecrire b en copie (adresse 60)
61	3D	63		
62	3E	40	LOAD 63	charger B dans l'accumulateur
63	3F	63		
64	40	11	BRN 50	si B = 0, on sort en affichant le résultat
65	41	50		
66	42	40	LOAD 60	charger resultat dans l'accumulateur
67	43	60		
68	44	20	ADD 01	incrémenter le résultat
69	45	01		
70	46	48	STORE 60	écraser la valeur du résultat par sa nouvelle valeur
71	47	60		
72	48	40	LOAD 63	charger B dans l'accumulateur
73	49	63		
74	4A	61	SUB 61	soustraire NB2
75	4B	61		
76	4C	48	STORE 63	écraser la valeur de B par sa nouvelle valeur
77	4D	63		
78	4E	10	JUMP 3E	brancher sur 40 (nouveau tour de boucle)
79	4F	3E		
80	50	41	OUT 60	afficher le résultat
81	51	60		
82	52	10	JUMP 00	sortir du programme
83	53	00		

## Exemple d'exécution du programme :

*Nota : les Defines pour une tel execution sont les suivants :*

```
1 // #define DEBUG
2 // #define DEBUG_n2
3 // #define STEPPER
4 #define AUTO
5 // #define EXO
6 #define TAILLE_MEMOIRE 256
7 #define TAILLE_ADRESSE 3
8 #define TAILLE_BOOTSTRAP 32
9 #define TAILLE_MAX_PROGRAMME 200
10 #define FIRST_INSTRUCTION "28"
11 #define FIRST_INSTRUCTION_EXO_10_6 "50"
```



```
elie@NZXT: ~/Workspace/Archi/chapitre 10
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier division

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée : 40
    Veuillez saisir une donnée : 10

    Affichage OUT: 4
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier division

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée : 35
    Veuillez saisir une donnée : 09

    Affichage OUT: 3
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier division

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée : 52
    Veuillez saisir une donnée : 00

    Affichage OUT: 00
elie@NZXT:~/Workspace/Archi/chapitre 10$
```

Le dernier cas de la division par zero est simplifiée ici en renvoyant un "00"

## 7 Exercice 10.7

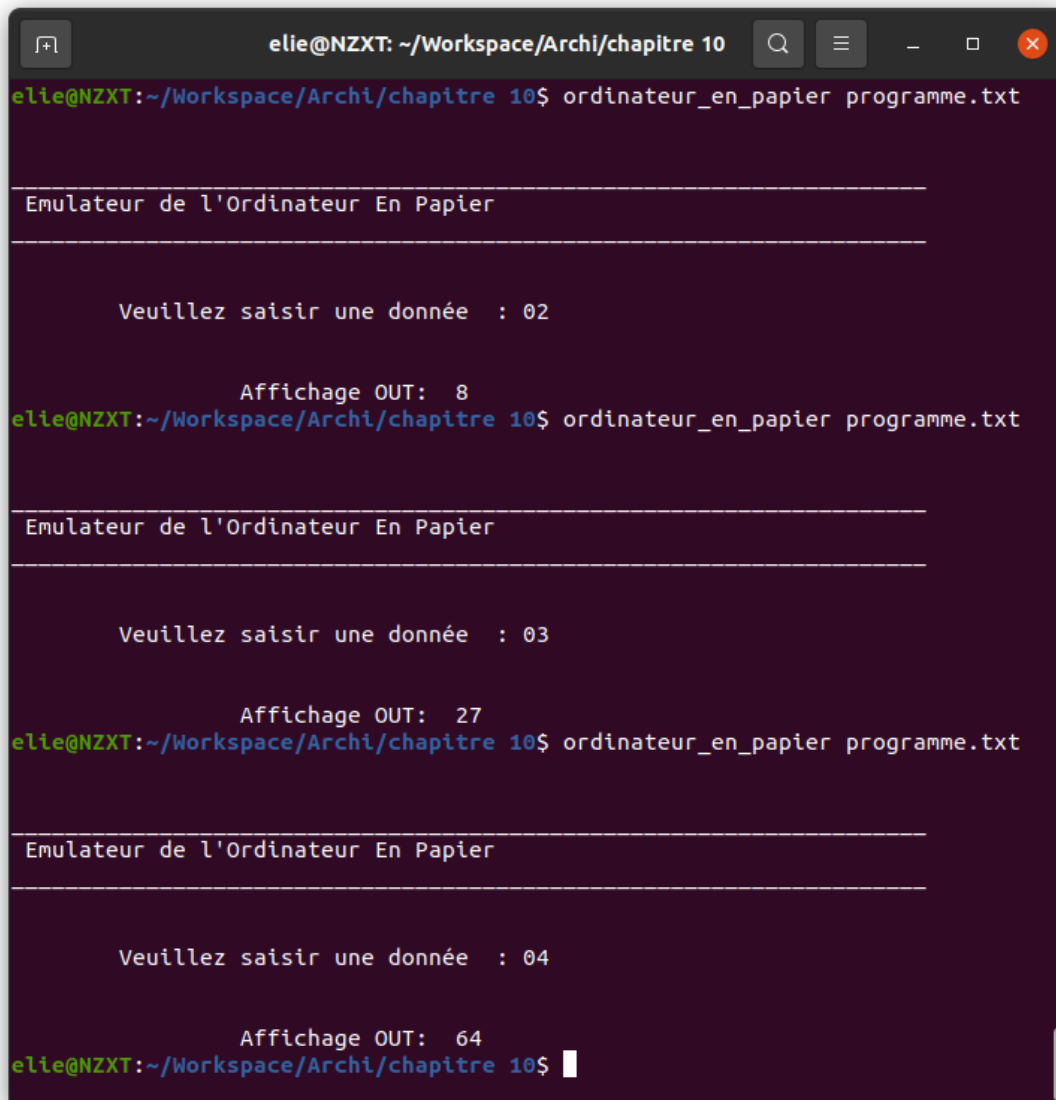
### A RENDRE

En vous inspirant de l'Exercice 10.5... Indiquez les mnémoniques qui correspondent aux valeurs suivantes dans la mémoire.

Adresse	Adresse	code	mnémonique	commentaire
80	50	49	IN 70	entrer une valeur dans adresse 70
81	51	70		
82	52	40	LOAD 70	charger cette valeur
83	53	70		
84	54	48	STORE 71	ecrire cette valeur dans l'adresse 71 (variable a)
85	55	71		
86	56	48	STORE 72	ecrire cette valeur dans l'adresse 72 aussi (variable b)
87	57	72		
88	58	00	LOAD 5E	charger la "5E" dans l'accumulateur
89	59	5E		
90	5A	48	STORE 8D	ecrire "5E" a l'adresse "8D"
91	5B	8D		
92	5C	10	JUMP 74	Jump à l'adresse 74 ->
93	5D	74		
94	5E	40	LOAD 71	charger le résultat dans l'accumulateur
95	5F	71		
96	60	48	STORE 72	l'écriture à l'adresse 72
97	61	72		
98	62	40	LOAD 70	charger le nombre initial dans l'accumulateur
99	63	70		
100	64	48	STORE 71	l'écriture à l'adresse 71
101	65	71		
102	66	00	LOAD 6C	charger "6C" dans l'accumulateur
103	67	6C		
104	68	48	STORE 8D	ecrire "6C" à l'adresse "8D" (la fin du programme)
105	69	8D		
106	6A	10	JUMP 74	jump à l'adresse 74
107	6B	74		
108	6C	41	OUT 71	
109	6D	71		
110	6E	10	JUMP 6E	
112	6F	6E		
113	70	00		
114	71	00		
115	72	00		
116	73	00		

116	74	00	LOAD 00	charger "00" dans l'accumulateur
117	75	00		
118	76	48	STORE 73	ecrire "00" a l'adresse 73 (variable c)
119	77	73		
120	78	40	LOAD 71	charger la valeur de l'adresse 71 (ici 2)
121	79	71		
122	7A	12	BRZ 88	si A = "00" -> Brancher sur 88
123	7B	88		
124	7C	21	SUB 01	decrémenter de 1 (ici 1)
125	7D	01		
126	7E	48	STORE 71	ecrire le résultat a l'adresse 71 (decrementer a)
127	7F	71		
127	80	40	LOAD 73	charger la valeur de l'adresse 73 (ici 00)
128	81	73		
129	82	60	ADD 72	additionner la valeur à l'adresse 72 (variable c + a)
130	83	72		
131	84	48	STORE 73	ecraser la valeur à l'adresse 73 avec le résultat du calcul
132	85	73		
133	86	10	JUMP 78	
134	87	78		
135	88	40	LOAD 73	charger le résultat
136	89	73		
137	8A	48	STORE 71	ecraser variable a avec le resultat
138	8B	71		
139	8C	10	JUMP 00	Jump sur 5E
140	8D	00		

Le programme renvoie le nombre saisi à la puissance 3 On le voit sur cette captures d'écrans



```
elie@NZXT: ~/Workspace/Archi/chapitre 10
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier programme.txt

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 02

    Affichage OUT:  8
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier programme.txt

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 03

    Affichage OUT:  27
elie@NZXT:~/Workspace/Archi/chapitre 10$ ordinateur_en_papier programme.txt

-----
Emulateur de l'Ordinateur En Papier
-----

    Veuillez saisir une donnée  : 04

    Affichage OUT:  64
elie@NZXT:~/Workspace/Archi/chapitre 10$
```

## 8 Emulateur : Ordinateur en Papier (code du programme) :

### 8.1 Explication du projet :

Mon objectif aura été de me rapprocher au plus près de la description de l'ordinateur en papier qui nous est donnée dans le cour. tant dans la représentation des données que dans la modélisation des ressources (mémoire, registres).

La modélisation paraîtra donc laborieuse étant donné que je fais apparaître chaque couche d'abstraction dans la structure du code.

J'ai donc décidé de recréer virtuellement la mémoire sous forme de matrice de chaînes de caractères (typé 'Hexa' pour mentionner la base 16 dans laquelle lire ces données). Les registres seront représentés sous forme de vecteurs de chaînes de caractères (eux aussi typés 'Hexa').

Je capture ensuite le traitement de chaque microcode dans des fonctions. Je me sers ensuite de ces fonctions microcodes dans la construction de fonctions qui représente les opérations à effectuer, les fonctions d'opérations seront elles-mêmes appelées dans les fonctions qui orchestre le cycle d'exécution d'une instruction. Enfin le main jouera une boucle d'exécution des fonctions de cycle jusqu'à exécution complète du programme.

Une fois avoir obtenu un modèle d'ordinateur fonctionnel. J'ai pris l'initiative d'y ajouter plusieurs fonctionnalités :

- La capacité de charger un programme automatiquement en entrant le nom du programme sur la ligne de commande. Pour activer cette fonctionnalité il faudra décommenter la ligne **AUTO** dans la section 8.2.
- La capacité d'attendre la saisie de la touche 'Enter' avant d'exécuter l'instruction suivante de manière à pouvoir analyser étape par étape l'exécution d'un programme.  
Pour activer cette fonctionnalité il faudra décommenter la ligne **STEPPER** dans la section 8.2.
- La capacité d'afficher ou pas des logs de debug avec deux niveaux de détail  
Pour activer cette fonctionnalité il faudra décommenter la ligne **DEBUG** et/ou **DEBUG\_n2** dans la section 8.2 : *(voir fichiers test\_DEBUG et test\_DEBUG\_n2, qui sont l'image de l'exécution d'une multiplication de  $20 \times 3 = 60$ )*
  - **DEBUG** (affiche l'état des registres et indique les phases de cycles en cours d'exécution, ainsi que le résultat des calculs effectués).
  - **DEBUG\_n2** (indique l'appel de chaque fonction en cours d'exécution et donne des informations supplémentaires sur les conversions de type).

Le code de l'exercice 10.7 ne respectant pas mes conventions j'ai dû créer un environnement EXO pour son exécution, cet environnement permet de charger le programme au bon endroit dans la mémoire. Pour activer cet environnement et jouer le programme de l'exercice 10.7 il faudra décommenter la ligne **EXO** dans la section 8.2.



### 8.1.1 Précautions d'utilisation :

- Avant une execution de programme de l'exercice 10.7, assurez vous d'avoir activé l'environnement **EXO** dans la section **8.2**.
- Lorsque le programme vous demande de saisir une valeur, veuillez à entrer une, veuillez à entrer une valeur numérique à deux chiffres.
- Si la valeur est inférieure à 10 , n'oubliez pas d'indiquer un zero devant le chiffre. (exemple '03')

## 8.2 bibliothèques - Constantes préprocesseur - typedef - Variables Globales

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 // #define DEBUG
6 // #define DEBUG_n2
7 // #define STEPPER
8 #define AUTO
9 // #define EXO
10 #define TAILLE_MEMOIRE 256
11 #define TAILLE_ADRESSE 3
12 #define TAILLE_BOOTSTRAP 32
13 #define TAILLE_MAX_PROGRAMME 200
14 #define FIRST_INSTRUCTION "28"
15 #define FIRST_INSTRUCTION_EXO_10_6 "50"
16
17 typedef char Hexa ;
18 typedef char * str ;
19 typedef unsigned idx ;
20 // Allocation d'espace pour :
21 // - la memoire
22 Hexa * memoire[TAILLE_MEMOIRE];
23
24 // - les registres
25 Hexa RS[TAILLE_ADRESSE];
26 Hexa RM[TAILLE_ADRESSE];
27
28 Hexa IN[TAILLE_ADRESSE];
29 Hexa OUT[TAILLE_ADRESSE];
30
31 Hexa PC[TAILLE_ADRESSE];
32
33 Hexa OP[TAILLE_ADRESSE];
34 Hexa AD[TAILLE_ADRESSE];
35
36 Hexa A[TAILLE_ADRESSE];
37
38 // - l'unité de calcul
39 int UAL;
40
41 // - le programme à charger en memoire
42 Hexa * prog[TAILLE_MAX_PROGRAMME] = { NULL };
43
44 // import des headers
45 #include "fonctions.h"
```

## 8.3 fonction main

```
1 #ifdef AUTO
2 int main(int k, char *argv[]) {
3     // test du nombre d'arguments
4     if (k < 2) usage("veuillez indiquer le nom du programme a lire");
5
6     printf("\n\
n----- \n"
);
7     printf("  Emulateur de l'Ordinateur En Papier \n");
8     printf("
----- \n\n
");
9
10    // initialisation de la mémoire
11    initialiserRegistres();
12    chargerBootstrap();
13    // afficherMemoire(55);
14
15    // chargement du programme
16    int taille = lireProgramme(argv[1]);
17    // afficherProgramme();
18    chargerProgramme(taille);
19    // afficherMemoire(250);
20
21
22    #ifdef EXO
23        Hexa fin[TAILLE_ADRESSE] = "6E";
24    #else
25        Hexa fin[TAILLE_ADRESSE] = "00";
26    #endif
27
28    do{
29        #ifdef DEBUG
30            printf("
----- \n")
;
31            #endif
32            phase1(); // recherche d'instruction
33            executer(hexaToInt(OP));
34            #ifdef DEBUG
35                printf("
----- \n")
;
36                printf("Etat des Registres:\n");
37                printf("- PC=%s \t AD=%s \t OP=%s / %i\n" , PC, AD, OP , hexaToInt(
OP));
38                printf("- RS=%s \t RM=%s \t A=%s\n" , RS, RM , A);
39                printf("- UAL=%i \t IN=%s \t OUT=%s\n" , UAL, IN , OUT);
40                printf("
----- \n\n
\n");
41            #endif
42
```

```

43
44
45
46
47
48     #ifdef STEPPER
49         stepper();
50     #endif
51 } while (strcasecmp(PC , fin)); // fin du programme quand on jump sur
00
52     return 0;
53 }
54 #else
55 int main(int k, char *argv[]) {
56     initialiserRegistres();
57     chargerBootstrap();
58     do{
59         phase1(); // recherche d'instruction
60         executer(hexaToInt(OP));
61
62     } while (strcasecmp(PC , "00")); // fin du programme quand on jump sur
00
63     return 0;
64 }
65 #endif

```

## 8.4 fonctions : outils du main

### 8.4.1 usage()

```
1  /*  fonction: usage()
2      objectif: impression de messages d'erreur (sur flux stderr)
3      parametres: une string (le messages à renvoyer)*/
4  void usage(str message) { fprintf(stderr, "Usage : %s\n", message), exit(1)
    ;}
```

### 8.4.2 stepper()

```
1  /*  fonction: stepper()
2      objectif: permettre a l'utilisateur d'utiliser la touche 'enter
3              pour passer une instruction */
4  void stepper() {
5      printf("( 'Enter ' pour passer à l'instruction suivante)");
6      char c;
7      while((c =getchar()) == '\0 ');
8  }
```

### 8.4.3 afficherMemoire()

```
1  /*  fonction: afficherMemoire()
2      objectif:
3          - afficher la memoire
4      */
5  void afficherMemoire(int adresse) {
6      #ifdef DEBUG
7          printf("- affichage de la memoire \n");
8      #endif
9      for (int i = 0; i < adresse && i < TAILLE_MEMOIRE ; i++) {
10         printf("adresse %i\t--> b16: %x:\t%s\n", i, i, memoire[i]);
11     }
12 }
```

#### 8.4.4 afficherProgramme()

```
1  /*  fonction: afficherMemoire()
2      objectif:
3          - afficher le programme
4  */
5  void afficherProgramme() {
6      #ifdef DEBUG
7          printf("- affichage du programme \n");
8      #endif
9      for (int i = 0; prog[i] ; i++) {
10         printf("instruction : %s\n",  prog[i]);
11     }
12 }
```

#### 8.4.5 initialiserRegistres()

```
1  /*  fonction: initialiserRegistres()
2      objectif: attribuer une valeur initiale aux différents registres */
3  void initialiserRegistres() {
4      #ifdef DEBUG
5          printf("- initialisation des registres  \n");
6      #endif
7
8      // registre memoire
9      strcpy(RS, "00 ");
10     strcpy(RM, "15 ");
11
12     // Entrées / sorties
13     strcpy(IN, "00 ");
14     strcpy(OUT, "00 ");
15
16     // Program Counter
17     #ifdef EXO
18         strcpy(PC, FIRST_INSTRUCTION_EXO_10_6);
19     #else
20         strcpy(PC, FIRST_INSTRUCTION);
21     #endif
22     // Registre Instruction
23     strcpy(OP, "00 ");
24     strcpy(AD, "00 ");
25
26     // unité de calcul
27     strcpy(A, "10 ");
28     UAL = 0;
29 }
```

#### 8.4.6 lireProgramme()

```
1  /*  fonction: lireProgramme()
2      objectif:
3          - prend le nom d'un programme à lire
4          - stocke les instructions du programme dans le vecteur 'programme'
5      parametres:
6          - une string (le nom du programme à lire)*/
7  int lireProgramme(str programme) {
8      FILE * fichier = fopen(programme, "r");
9      if (! fichier) usage(" stoplist illisible");
10
11      int i = 0;
12      char lu = '\0';
13      while (i < TAILLE_MAX_PROGRAMME && lu != EOF){
14          char sas[TAILLE_ADRESSE];           // sas de reception du mot
15          lu = fscanf(fichier, "%s ", sas);
16          if (lu != EOF){
17              prog[i++] = strdup(sas);
18          }
19      }
20      fclose(fichier);
21      return i - 1;
22  }
```

#### 8.4.7 chargerBootstrap()

```
1  /*  fonction: chargerBootstrap()
2      objectif:
3          - li le fichier "bootstrap"
4          - le charge en memoire */
5  void chargerBootstrap() {
6      #ifdef DEBUG
7          printf("- chargement du bootstrap \n");
8      #endif
9      // ouverture du flux
10     FILE * fichier = fopen("bootstrap", "r");
11     if (! fichier) usage("bootstrap illisible");
12     idx i = 0;
13     char lu = '\0';
14     while (i < TAILLE_MEMOIRE && lu != EOF){
15         char sas[TAILLE_ADRESSE];           // sas de reception du mot
16         lu = fscanf(fichier, "%s ", sas);
17         if (lu != EOF){
18             memoire[i++] = strdup(sas);
19         }
20     }
21     fclose(fichier);
22  };
```

### 8.4.8 chargerProgramme()

```
1  /*  fonction: chargerProgramme()
2      objectif:
3          - ecrit le programme dans la mémoire
4          - sans passer par le bootstrap manuel
5      parametres:
6          - un entier (l'adresse de la première instruction du programme) */
7  void chargerProgramme(int t) {
8      #ifdef DEBUG
9          printf("- chargement du programme\n");
10     #endif
11     //int adresse = hexaToInt(RS) + 40;
12     #ifdef EXO
13         int adresse = hexaToInt(FIRST_INSTRUCTION_EXO_10_6);
14     #else
15         int adresse = hexaToInt(FIRST_INSTRUCTION);
16     #endif
17     for (int i = 0; i < t ; i ++){
18         memoire[adresse + i] = strdup(prog[i]);
19     }
20     // ecriture de l'emplacement de la première instruction
21     #ifdef EXO
22         memoire[32] = strdup(FIRST_INSTRUCTION_EXO_10_6);
23     #else
24         memoire[32] = strdup(FIRST_INSTRUCTION);
25     #endif
26     // ecriture de la taille de la mémoire
27     Hexa taille[TAILLE_ADRESSE];
28     intToStr(taille , t);
29     strcpy(memoire[34], taille);
30 }
```

### 8.4.9 executer()

```
1  /*  fonction: executer()
2      objectif:
3          - prend en compte le microcode entré en parametre
4          - appel la fonction associé (voir page 224)
5      parametres:
6          - un entier (le code associé a une fonction) */
7  void executer(int code) {
8      #ifdef DEBUG
9          printf("==> phase 2 : exec code -> %i \n", hexaToInt(OP));
10     #endif
11     switch (code)
12     {
13
14
15
16
17
```



```

18
19
20 //
21 // ARITHMETIQUE
22 //
23 // ADD # 20
24 case 32:
25     addValeur();
26     break;
27
28 // ADD 60
29 case 96:
30     addValeurP();
31     break;
32
33 // ADD * E0
34 case 224:
35     addValeurPP();
36     break;
37
38 // SUB # 21
39 case 33:
40     subValeur();
41     break;
42
43 // SUB 61
44 case 97:
45     subValeurP();
46     break;
47
48 // SUB * E1
49 case 225:
50     subValeurPP();
51     break;
52
53
54 //
55 // LOGIQUE
56 //
57 // NAND # 22
58 case 34:
59     nand();
60     break;
61
62 // NAND 62
63 case 98:
64     nandP();
65     break;
66
67 // NAND * E2
68 case 226:
69     nandPP();
70     break;
71
72

```

```

73 //
74 // TRANSFERTS
75 //
76 // LOAD # 00
77 case 0:
78     load();
79     break;
80
81 // LOAD 40
82 case 64:
83     loadP();
84     break;
85
86 // LOAD * C0
87 case 192:
88     loadPP();
89     break;
90
91 // STORE 48
92 case 72:
93     storeP();
94     break;
95
96 // STORE * C8
97 case 200:
98     storePP();
99     break;
100
101 //
102 // ENTREE / SORTIES
103 //
104 // IN 49
105 case 73:
106     inP();
107     break;
108
109 // IN * C9
110 case 201:
111     inPP();
112     break;
113
114 // OUT 41
115 case 65:
116     outP();
117     break;
118
119 // OUT * C1
120 case 193:
121     outPP();
122     break;
123
124
125
126
127

```

```

128
129
130 //
131 // BRANCHEMENT INCONDITIONNEL
132 //
133 // JUMP      10
134 case 16:
135     jump();
136     break;
137
138 //
139 // BRANCHEMENT CONDITIONNEL
140 //
141 // BRN       11
142 case 17:
143     brn();
144     break;
145
146 // BRN       12
147 case 18:
148     brz();
149     break;
150
151 default:
152     printf(" !\\ CODE OP %i INNEXISTANT !\\ ", code);
153     exit(1);
154     break;
155 }
156 }

```

## 8.5 Fonctions Microcodes

### 8.5.1 transfert()

```
1  /*  fonction: transfert()
2      objectif:
3          - prend en compte le microcode entré en parametre
4          - effectuer le transfert mémoire associé (voir page 227)
5      parametres:
6          - un entier (le microcode)*/
7  void transfert(int code){
8      #ifdef DEBUG_n2
9          printf("— transfert -> %i \n", code);
10     #endif
11
12     switch (code)
13     {
14     case 1:
15         /* (RS)      (PC) */
16         strcpy(RS,PC);
17         break;
18     case 2:
19         /* (PC)      (RM) ; ne pas faire la phase III. */
20         strcpy(PC,RM);
21         break;
22     case 3:
23         /* (A)       (RM) */
24         strcpy(A,RM);
25         break;
26     case 4:
27         /* (RM)      (A) */
28         strcpy(RM,A);
29         break;
30     case 5:
31         /* (OP)      (RM) */
32         strcpy(OP,RM);
33         break;
34     case 6:
35         /* (AD)      (RM) */
36         strcpy(AD,RM);
37         break;
38     case 7:
39         /* (RS)      (AD) */
40         strcpy(RS,AD);
41         break;
42     case 8:
43         /* (RM)      (Entrée) */
44         strcpy(RM,IN);
45         break;
46     case 9:
47         /* (Sortie)   (RM) */
48         strcpy(OUT,RM);
49         break;
50
51 }
```

```

52
53
54     default:
55         usage("Erreur – fonction transfert() : le microcode ne correspond pas a
           un transfert ou n'existe pas.");
56         break;
57     }
58 }

```

### 8.5.2 prepaCalcul()

```

1  /*  fonction: prepaCalcul()
2      objectif:
3          – prend en compte le microcode entré en parametre
4          – préparer le calcul associé (voir page 227)
5            (sauvegarde le microcode du calcul dans UAL)
6      parametres:
7          – un entier (le calcul demandé)*/
8  void prepaCalcul(int code){
9      #ifdef DEBUG
10         printf("– preparation calcul:  %i \n", code);
11     #endif
12
13     UAL = code;
14 }

```

### 8.5.3 hexaToInt()

```

1  /*  fonction: hexaToInt()
2      objectif:
3          – traduit un hexadecimal en int
4      parametres:
5          – un hexa (l'operande a traduire)
6      retour:
7          – un entier */
8  int hexaToInt(Hexa h[TAILLE_ADRESSE] ){
9      #ifdef DEBUG_n2
10         int i = strtol(h, NULL, 16);
11         printf("    – conversion hexa: %s  --> ToInt : %i\n", h, i);
12     #endif
13
14     return strtol(h, NULL, 16);
15 }

```

### 8.5.4 intToHexa()

```
1  /*  fonction: intToHexa()
2      objectif:
3          - traduit un int en hexadecimal
4      parametres:
5          - un entier (l'operande a traduire)*/
6  void intToHexa(Hexa * registre , int code ){
7      Hexa value[TAILLE_ADRESSE];
8      sprintf(value , "%X" , code);
9
10     // gestion des valeurs < 10
11     if ( strlen(value) < 2) {
12         value[1] = value[0];
13         value[0] = '0';
14     }
15
16     // attribution de la nouvelle valeur au registre
17     strcpy(registre , value);
18     #ifdef DEBUG_n2
19         printf("    - conversion int: %i  --> ToHexa : %s\n" , code , registre);
20     #endif
21
22 }
```

### 8.5.5 intToStr()

```
1  /*  fonction: intToStr()
2      objectif:
3          - traduit un int en str
4      parametres:
5          - un entier (l'operande a traduire)*/
6  void intToStr(Hexa * registre , int code ){
7      sprintf(registre , "%d" , code);
8      #ifdef DEBUG_n2
9          printf("    - conversion int: %i  --> ToStr : %s\n" , code , registre);
10     #endif
11 }
```

### 8.5.6 calcul()

```
1  /* fonction: calcul()
2     objectif:
3         - traduit les deux operandes en entiers
4         - effectue le calcul associé (voir page 227)
5         (sauvegarde le resultat du calcul dans A)
6     parametres:
7         - un entier (le calcul demandé)*/
8 void calcul(){
9     // recuperation des operandes
10    int a = strtol(A, NULL, 10);
11    int rm = strtol(RM, NULL, 10);
12
13    // calcul
14    switch (UAL) {
15        /* addition */
16        case 10:
17            a = a + rm;
18            #ifdef DEBUG
19                printf("-calcul operandes %i + %i \n",a, rm);
20            #endif
21            break;
22
23        /* soustraction */
24        case 11:
25            a = a - rm;
26            #ifdef DEBUG
27                printf("-calcul operandes %i - %i \n",a, rm);
28            #endif
29            break;
30
31        /* logique -> NAND */
32        case 17:
33            if (!(a > 0 && rm > 0)){
34                a = 1;
35                #ifdef DEBUG
36                    printf("-calcul operandes ~(%i ET %i) --> T \n",a, rm);
37                #endif
38            } else {
39                a = 0;
40            }
41            break;
42        default:
43            usage("Erreur- fonction calcul() : le microcode ne correspond pas a
44                un opération ou n'existe pas.");
45            break;
46    }
47    // sauvegarde du resultat dans l'accumulateur
48    intToStr(A, a);
49 }
```

### 8.5.7 lireMemoire()

```
1  /*  fonction: lireMemoire()
2      objectif:
3          - decoder l'adresse contenu dans RS
4          - reporter le contenu de la mémoire dans RM
5      parametres:
6          - un hexa (l'adresse a decoder)*/
7  void lireMemoire() {
8      strcpy(RM, memoire[hexaToInt(RS)]);
9      #ifdef DEBUG_n2
10         printf("- lecture de la memoire memoire[%s]=%s\n",RS,RM);
11     #endif
12
13 };
```

### 8.5.8 ecrire()

```
1  /*  fonction: ecrire()
2      objectif:
3          - decoder l'adresse contenu dans RS
4          - reporter le contenu de la mémoire dans RM
5      parametres:
6          - un hexa (l'adresse a decoder)*/
7  void ecrire() {
8      int idx = hexaToInt(RS);
9      #ifdef DEBUG_n2
10         printf("- ecriture valeur: %s dans la case mémoire %i\n",RM, idx);
11     #endif
12
13     memoire[idx] = strdup(RM);
14 };
```



### 8.5.9 saisir()

```
1  /*  fonction: saisir()
2      objectif:
3          - saisir une donnée dans IN*/
4  void saisir(){
5      #ifdef DEBUG_n2
6          printf("- saisie d'une valeur \n");
7      #endif
8      printf("\n\tVeuillez saisir une donnée : ");
9      lire(IN, TAILLE_ADRESSE);
10 };
```

### 8.5.10 lire()

```
1  /*  fonction: lire()
2      objectif:
3          - supprime le saut de ligne en fin de saisie user
4      retour
5          - 0 si il est impossible de lire la saisie
6          - 1 si la saisie est reussi et lu*/
7  int lire(char * mot, int taille){
8      if (fgets(mot, taille, stdin)){
9          char * monPointeur = strchr(mot, '\n');
10         if (monPointeur)
11             *monPointeur = '\0';
12
13         while(getchar() != '\n' && getchar() != EOF);
14         return 1;
15     }
16     while(getchar() != '\n' && getchar() != EOF);
17     return 0; // si impossible de lire la saisie
18 }
```

## 8.6 Fonctions de déroulement d'un cycle

La phase 2 est représentée par la fonction `executer()` -> section précédente

### 8.6.1 `phase1()`

```
1  /*  fonction: phase1()
2      objectif:
3          - rechercher une instruction
4          - la stocker dans OP*/
5  void phase1() {
6      #ifdef DEBUG
7          printf("==> phase 1 \n");
8      #endif
9      transfert(1);      // 1
10     lireMemoire();      // 13
11     transfert(5);       // 5
12     incrementerPC();    // 15
13 };
```

### 8.6.2 `phase 3`

```
1  /*  fonction: phase3()
2      objectif:
3          - increpenter PC*/
4  void phase3() {
5      #ifdef DEBUG
6          printf("\n==> phase 3 \n");
7      #endif
8      incrementerPC();   // 15
9  };
```

### 8.6.3 `incrementerPC()`

```
1  /*  fonction: incrementerPC()
2      objectif:
3          - increpenter PC*/
4  void incrementerPC() {
5      #ifdef DEBUG_n2
6          printf("    - incrémentation de PC\n");
7      #endif
8      int pc = hexaToInt(PC);
9      pc ++;
10
11     intToHexa(PC , pc);
12 };
```

## 8.7 Fonction D'Opérations

### 8.7.1 add()

```
1  /*  fonction: addValeur()    20
2      description:
3          ADD# -->  A      A+V
4  */
5  void addValeur() {
6      #ifdef DEBUG
7          printf("\n\t[ ADD # ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(10);    // 10
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     calcul();           // 12
14     // phase 3
15     phase3();
16 };
```

### 8.7.2 addP()

```
1  /*  fonction: addValeur()    60
2      description:
3          ADD -->  A      A + ( )
4  */
5  void addValeurP() {
6      #ifdef DEBUG
7          printf("\n\t[ ADD      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(10);    // 10
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     transfert(6);       // 6
14     transfert(7);       // 7
15     lireMemoire();      // 13
16     calcul();           // 12
17     // phase 3
18     phase3();
19 };
```

### 8.7.3 addPP()

```
1  /*  fonction: addValeur()    E0
2      description:
3          ADD*  -->  A      A + *( )
4  */
5  void addValeurPP() {
6      #ifdef DEBUG
7          printf("\n\t[ ADD *      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(10);    // 10
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     transfert(6);       // 6
14     transfert(7);       // 7
15     lireMemoire();      // 13
16     transfert(6);       // 6
17     transfert(7);       // 7
18     lireMemoire();      // 13
19     calcul();           // 12
20     // phase 3
21     phase3();
22 };
```

### 8.7.4 sub()

```
1  /*  fonction: subValeur()    21
2      description:
3          SUB# -->  A      A-V
4  */
5  void subValeur() {
6      #ifdef DEBUG
7          printf("\n\t[ SUB #  ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(11);    // 11
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     calcul();           // 12
14     // phase 3
15     phase3();
16 };
```

### 8.7.5 subP()

```
1  /*  fonction: subValeurP()    61
2      description:
3          SUB  -->  A      A - ( )
4  */
5  void subValeurP() {
6      #ifdef DEBUG
7          printf("\n\t[ SUB      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(11);    // 11
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     transfert(6);       // 6
14     transfert(7);       // 7
15     lireMemoire();      // 13
16     calcul();           // 12
17     // phase 3
18     phase3();
19 };
```

### 8.7.6 subPP()

```
1  /*  fonction: subValeurPP()   E1
2      description:
3          SUB*  -->  A      A - *( )
4  */
5  void subValeurPP() {
6      #ifdef DEBUG
7          printf("\n\t[ SUB *      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(11);    // 11
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     transfert(6);       // 6
14     transfert(7);       // 7
15     lireMemoire();      // 13
16     transfert(6);       // 6
17     transfert(7);       // 7
18     lireMemoire();      // 13
19     calcul();           // 12
20     // phase 3
21     phase3();
22 };
```

### 8.7.7 nand()

```
1  /*  fonction: nand()      22
2      description:
3          NAND# -->  A      [A&V]
4  */
5  void nand() {
6      #ifdef DEBUG
7          printf("\n\t[ NAND # ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(17);    // 17
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     calcul();           // 12
14     // phase 3
15     phase3();
16 };
```

### 8.7.8 nandP()

```
1  /*  fonction: nandP()    62
2      description:
3          NAND -->  A      [A&( )]
4  */
5  void nandP() {
6      #ifdef DEBUG
7          printf("\n\t[ NAND      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(17);    // 17
11     transfert(1);       // 1
12     lireMemoire();      // 13
13     transfert(6);       // 6
14     transfert(7);       // 7
15     lireMemoire();      // 13
16     calcul();           // 12
17     // phase 3
18     phase3();
19 };
```

### 8.7.9 nandPP()

```
1  /*  fonction: nandPP()    E2
2      description:
3          NAND*  -->  A      [A&    (  )]
4  */
5  void nandPP(){
6      #ifdef DEBUG
7          printf("\n\t[ NAND *      ]\n");
8      #endif
9      // phase 2
10     prepaCalcul(17);    // 17
11     transfert(1);        // 1
12     lireMemoire();       // 13
13     transfert(6);        // 6
14     transfert(7);        // 7
15     lireMemoire();       // 13
16     transfert(6);        // 6
17     transfert(7);        // 7
18     lireMemoire();       // 13
19     calcul();            // 12
20     // phase 3
21     phase3();
22 };
```

### 8.7.10 load()

```
1  /*  fonction: load()    00
2      description:
3          LOAD# -->  A      V
4  */
5  void load(){
6      #ifdef DEBUG
7          printf("\n\t[ LOAD #  ]\n");
8      #endif
9      // phase 2
10     transfert(1);        // 1
11     lireMemoire();       // 13
12     transfert(3);        // 3
13     // phase 3
14     phase3();
15 };
```

### 8.7.11 loadP()

```
1  /*  fonction: loadP()      40
2      description:
3          LOAD --> A      ( )
4  */
5  void loadP() {
6      #ifdef DEBUG
7          printf( "\n\t[ LOAD      ]\n" );
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     lireMemoire();         // 13
15     transfert(3);          // 3
16     // phase 3
17     phase3();
18 };
```

### 8.7.12 loadPP()

```
1  /*  fonction: loadPP()    C0
2      description:
3          LOAD* --> A      ( )
4  */
5  void loadPP() {
6      #ifdef DEBUG
7          printf( "\n\t[ LOAD *      ]\n" );
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     lireMemoire();         // 13
15     transfert(6);          // 6
16     transfert(7);          // 7
17     lireMemoire();         // 13
18     transfert(3);          // 3
19     // phase 3
20     phase3();
21 };
```



### 8.7.13 storeP()

```
1  /*  fonction: storeP ()    48
2      description:
3          STORE --> ( )      A
4  */
5  void storeP () {
6      #ifdef DEBUG
7          printf( "\n\t[ STORE      ]\n" );
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     lireMemoire();         // 13
15     transfert(4);          // 4
16     ecrire();              // 14
17     // phase 3
18     phase3();
19 };
```

### 8.7.14 storePP()

```
1  /*  fonction: storePP ()   C8
2      description:
3          LOAD*  --> A        ( )
4  */
5  void storePP () {
6      #ifdef DEBUG
7          printf( "\n\t[ STORE *      ]\n" );
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     lireMemoire();         // 13
15     transfert(6);          // 6
16     transfert(7);          // 7
17     transfert(4);          // 4
18     ecrire();              // 14
19     // phase 3
20     phase3();
21 };
```

### 8.7.15 inP()

```
1  /*  fonction: inP()      49
2      description:
3          IN  -->  ( )      Entrée
4  */
5  void inP() {
6      #ifdef DEBUG
7          printf("\n\t[ IN      ]\n");
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     saisir();              // 16
15     transfert(8);          // 8
16     ecrire();              // 14
17     // phase 3
18     phase3();
19 };
```

### 8.7.16 inPP()

```
1  /*  fonction: inPP()     C9
2      description:
3          IN*  -->      ( )      Entrée
4  */
5  void inPP() {
6      #ifdef DEBUG
7          printf("\n\t[ IN *      ]\n");
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(6);          // 6
13     transfert(7);          // 7
14     lireMemoire();         // 13
15     transfert(6);          // 6
16     transfert(7);          // 7
17     saisir();              // 16
18     transfert(8);          // 8
19     ecrire();              // 14
20     // phase 3
21     phase3();
22 };
```

### 8.7.17 outP()

```
1  /*  fonction: outP()    41
2      description:
3          OUT  -->  Sortie    ( )
4  */
5  void outP(){
6      #ifdef DEBUG
7          printf("\n\t[ OUT    ]\n");
8      #endif
9      // phase 2
10     transfert(1);        // 1
11     lireMemoire();       // 13
12     transfert(6);        // 6
13     transfert(7);        // 7
14     lireMemoire();       // 13
15     transfert(9);        // 9
16     printf("\n\n\t\t Affichage OUT:  %s\n", OUT);
17     // phase 3
18     phase3();
19 };
```

### 8.7.18 outPP()

```
1  /*  fonction: outPP()   C1
2      description:
3          OUT*  -->  Sortie    ( )
4  */
5  void outPP(){
6      #ifdef DEBUG
7          printf("\n\t[ OUT *    ]\n");
8      #endif
9      // phase 2
10     transfert(1);        // 1
11     lireMemoire();       // 13
12     transfert(6);        // 6
13     transfert(7);        // 7
14     lireMemoire();       // 13
15     transfert(6);        // 6
16     transfert(7);        // 7
17     lireMemoire();       // 13
18     transfert(9);        // 9
19     printf("--> Affichage OUT:  %s\n", OUT);
20     // phase 3
21     phase3();
22 };
```

### 8.7.19 jump()

```
1  /*  fonction: jump()    10
2      description:
3          JUMP    -->  PC
4  */
5  void jump() {
6      #ifdef DEBUG
7          printf( "\n\t[ JUMP    ]\n" );
8      #endif
9      // phase 2
10     transfert(1);          // 1
11     lireMemoire();         // 13
12     transfert(2);          // 2
13 };
```

### 8.7.20 brn()

```
1  /*  fonction: brn()    11
2      description:
3          BRN    -->  si A < 0 alors PC
4  */
5  void brn() {
6      #ifdef DEBUG
7          printf( "\n\t[ si A < 0 : BRN    ]\n" );
8      #endif
9      // condition
10     if (hexaToInt(A) < 0) {
11         // phase 2
12         transfert(1);          // 1
13         lireMemoire();         // 13
14         transfert(2);          // 2
15     } else {
16         phase3();
17     }
18 };
```

### 8.7.21 brz()

```
1  /*  fonction: brz()    12
2      description:
3          BRN    -->    si A < 0 alors PC
4  */
5  void brz() {
6      #ifdef DEBUG
7          printf("\n\t[ si A = 0 : BRZ    ]\n");
8      #endif
9      // condition
10     if (hexaToInt(A) == 0) {
11         // phase 2
12         transfert(1);        // 1
13         lireMemoire();       // 13
14         transfert(2);        // 2
15     } else {
16         phase3();
17     }
18 };
```