

TP1 Architecture des Ordinateurs

INTRODUCTION À LA PROGRAMMATION EN ASSEMBLEUR

Sommaire

0. Introduction.....	3
1. Représentation des données.....	4
1.1 Représentation des entiers.....	4
1.2 Opérations en complément à deux.....	6
1.3 Représentation des flottants.....	6
2. Le processeur et les registres.....	7
2.1 Registres et sous-registres.....	7
2.2 Opérations sur les registres.....	7
2.3 Lecture/écriture en mémoire.....	8
2.3.1 Lecture en mémoire.....	8
2.3.2 Écriture en mémoire.....	9
3. Premier programme.....	9
3.1 Compilation.....	9
3.3 Assemblage du programme.....	9
4. Conclusion.....	10

Introduction

Le processeur est l'unité de calcul de l'ordinateur ; en effet, il s'agit d'un composant capable de réaliser des calculs simples sur une suite de chiffres en base binaire (nombres composés de 0 et de 1) stockés dans sa mémoire. Ces nombres peuvent être convertis en d'autres bases pour une meilleure clarté et compréhension pour l'humain, telles que l'hexadécimal ou le décimal. Bien qu'à l'heure actuelle de nombreux outils nous permettent de faire les conversions automatiquement, il est important de bien comprendre comment elles sont réalisées pour mieux comprendre le fonctionnement de la machine, surtout à un si bas niveau.

En tant que programmeurs, nous sommes capables de communiquer avec le processeur presque directement avec le langage assembleur. L'assembleur va être converti en langage machine, langage capable de donner des instructions au processeur, mais pas pratique à utiliser pour coder, car il n'est pas vraiment clair. Le langage assembleur étant différent pour chaque modèle de processeur existant, il existe néanmoins une compatibilité des vieux langages assembleur sur les nouveaux processeurs, ce pourquoi on étudiera l'assembleur de l'Intel2000, plus ou moins simple à prendre en main et assez compatible avec la majorité de processeurs actuels.

Dans le cadre de ce TP, nous apprendrons à faire des conversions de données manuellement entre les bases binaire, décimale et hexadécimale, puis nous prendrons en main le langage assembleur avec l'étude de petits programmes qui afficheront une ou plusieurs chaînes de caractères sur le terminal.

1. Représentation des données

Exercice 1. (Nombre de données)

1.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2 ⁿ	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536

- On peut coder 2ⁿ données différentes sur n bits, car on a deux éléments (soit 1, soit 0) avec 2ⁿ combinaisons possibles.
- On peut coder 2^{8*n} données différentes sur n octets, car un octet équivaut à 8 bits ; on code sur 8*n bits.
- On peut coder 2^{1024*8*n} données différentes sur n Kio, car 1 Kio = 1024 octets.
- On peut coder 2^{1048576*8*n} données différentes sur n Mio.
- On peut coder 2^{1073741824*8*n} données différentes sur n Gio.

1.1 Représentation des entiers

Exercice 2. (Changement de base)

1. (999)_{dix} = (0011 1110 0111)_{deux} = (33213)_{quatre} = (3E7)_{seize}

x	x%2	x/2	L
(999) _{dix}	(1) _{dix}	(499) _{dix}	[1]
(499) _{dix}	(1) _{dix}	(249) _{dix}	[1, 1]
(249) _{dix}	(1) _{dix}	(124) _{dix}	[1, 1, 1]
(124) _{dix}	(0) _{dix}	(62) _{dix}	[0, 1, 1, 1]
(62) _{dix}	(0) _{dix}	(31) _{dix}	[0, 0, 1, 1, 1]
(31) _{dix}	(1) _{dix}	(15) _{dix}	[1, 0, 0, 1, 1, 1]
(15) _{dix}	(1) _{dix}	(7) _{dix}	[1, 1, 0, 0, 1, 1, 1]
(7) _{dix}	(1) _{dix}	(3) _{dix}	[1, 1, 1, 0, 0, 1, 1, 1]
(3) _{dix}	(1) _{dix}	(1) _{dix}	[1, 1, 1, 1, 0, 0, 1, 1, 1]
(1) _{dix}	(1) _{dix}	(0) _{dix}	[1, 1, 1, 1, 1, 0, 0, 1, 1, 1]
(0) _{dix}	-	-	[1, 1, 1, 1, 1, 0, 0, 1, 1, 1]

x	x%4	x/4	L
(999) _{dix}	(3) _{dix}	(249) _{dix}	[3]
(249) _{dix}	(1) _{dix}	(62) _{dix}	[1, 3]
(62) _{dix}	(2) _{dix}	(15) _{dix}	[2, 1, 3]
(15) _{dix}	(3) _{dix}	(3) _{dix}	[3, 2, 1, 3]
(3) _{dix}	(3) _{dix}	(0) _{dix}	[3, 3, 2, 1, 3]
(0) _{dix}	-	-	[3, 3, 2, 1, 3]

x	x%16	x/16	L
(999) _{dix}	(7) _{dix}	(62) _{dix}	[7]
(62) _{dix}	(14) _{dix}	(3) _{dix}	[14, 7]
(3) _{dix}	(3) _{dix}	(0) _{dix}	[3, 14, 7]
(0) _{dix}	-	-	[3, 14, 7]

2. (1002)_{trois} = (1003)_{dix}

$$(1002)_{\text{trois}} = (2 \times 3^0 + 0 \times 3^1 + 0 \times 3^2 + 1 \times 3^3) = (2 + 0 + 0 + 27) = (29)_{\text{dix}}$$

Exercice 3. (Binaire et hexadécimal)

1. $(0101\ 0101\ 1111\ 0000\ 0000\ 0001)_{\text{deux}} = (55F001)_{\text{seize}}$
2. $(2BABA101F9)_{\text{seize}} = (0010\ 1011\ 1010\ 1011\ 1010\ 0001\ 0000\ 0001\ 1111\ 1001)_{\text{deux}}$

Exercice 4. (Représentation avec biais)

- | | |
|---|--|
| <ol style="list-style-type: none">1. $n := 8$ bits
$k := 1000$
$-950 + 1000 = 50 \quad (50)_{\text{dix}} = (110010)_{\text{deux}}$
$(-950)_{\text{dix}} = (0011\ 0010)_{\text{biais}=1000}$2. $n := 16$ bits
$k := 256$
$0 + 256 = 256 \quad (256)_{\text{dix}} = (100000000)_{\text{deux}}$
$(0)_{\text{dix}} = (0000\ 0001\ 0000\ 0000)_{\text{biais}=1000}$3. $n := 8$ bits
$k := 21$
$(0)_{\text{deux}} = (0)_{\text{dix}} \quad 0 - 21 = 21$
$(0000\ 0000)_{\text{biais}=21} = (-21)_{\text{dix}}$ | <ol style="list-style-type: none">4. $n := 16$ bits
$k := 17$
$(10001)_{\text{deux}} = (17)_{\text{dix}} \quad 17 - 17 = 0$
$(0000\ 0000\ 0001\ 0001)_{\text{biais}=17} = (0)_{\text{dix}}$5. $n := 8$ bits
$k := 1777$
$(10001)_{\text{deux}} = (17)_{\text{dix}} \quad 17 - 1777 = -1760$
$(0001\ 0001)_{\text{biais}=1777} = (-1760)_{\text{dix}}$ |
|---|--|

Exercice 5. (Représentation en complément à deux)

1. $n := 8$ bits
 $(17)_{\text{dix}} = (0001\ 0001)_{c2}$
2. $n := 8$ bits
 $(1)_{\text{dix}} = (0000\ 0001)_{\text{deux}} = (1111\ 1110)_{c1} = (1111\ 1111)_{c2}$
 $(-1)_{\text{dix}} = (1111\ 1111)_{c2}$
3. $n := 16$ bits
 $(1021)_{\text{dix}} = (0000\ 0011\ 1111\ 1101)_{\text{deux}} = (1111\ 1100\ 0000\ 0010)_{c1} = (1111\ 1100\ 0000\ 0011)_{c2}$
 $(-1021)_{\text{dix}} = (1111\ 1100\ 0000\ 0011)_{c2}$
4. $n := 8$ bits
 $(1000)_{\text{dix}} = (0000\ 0011\ 1110\ 1100)_{\text{deux}} = (1111\ 1100\ 0001\ 0011)_{c1} = (1111\ 1100\ 0000\ 1111)_{c2}$
 $(-1000)_{\text{dix}} = (1111\ 1100\ 0000\ 1111)_{c2}$

On ne peut pas représenter -1000 sur 8 bits car il nous en faut au moins 11 ; 1000 est codé sur 10 bits et il nous faut un bit supplémentaire pour avoir le bit de signe.

5. $n := 8$ bits
 $(0000\ 1001)_{c2} = (9)_{\text{dix}}$

6. $n := 8$ bits
 $(1000\ 1001)_{\text{deux}} = (0111\ 0110)_{c1} = (0111\ 0111)_{c2} = (119)_{\text{dix}}$
 $(1000\ 1001)_{c2} = (-119)_{\text{dix}}$
7. $n := 8$
 $(1100\ 1011)_{\text{deux}} = (0011\ 0100)_{c1} = (0011\ 0101)_{c2} = (53)_{\text{dix}}$
 $(1100\ 1011)_{c2} = (-53)_{\text{dix}}$

1.2 Opérations en complément à deux

Exercice 6. (Dépassement de capacité (D.C.) et retenue de sortie (R.S.))

		Retenue de sortie	Dépassement de capacité
$v1 + v2$	$\begin{array}{r} 10011100 \\ + 00000101 \\ \hline 10100001 \end{array}$	0	non
$v3 + v4$	$\begin{array}{r} 10001000 \\ + 11111001 \\ \hline (1) 10000001 \end{array}$	1	non
$v5 + v6$	$\begin{array}{r} 00110101 \\ + 11100000 \\ \hline (1) 00110101 \end{array}$	1	non
$v7 + v8$	$\begin{array}{r} 01001111 \\ + 01011001 \\ \hline 10101000 \end{array}$	0	oui
$v9 + v10$	$\begin{array}{r} 10000000 \\ + 11000000 \\ \hline (1) 01000000 \end{array}$	1	oui
$v1 + v10$	$\begin{array}{r} 10011100 \\ + 11000000 \\ \hline (1) 01011100 \end{array}$	1	oui

1.3 Représentation des flottants

Exercice 7. (Flottants)

1. $n := 8$ bits $m := 7$ bits
 $b := 2$
 $x := (0.40625)_{\text{dix}}$

x	$x*2$	$x - [x]$	L
$(0.40625)_{\text{dix}}$	$(0.8125)_{\text{dix}}$	$(0.8125)_{\text{dix}}$	$[0]$
$(0.8125)_{\text{dix}}$	$(1.625)_{\text{dix}}$	$(0.625)_{\text{dix}}$	$[1, 0]$
$(0.625)_{\text{dix}}$	$(1.25)_{\text{dix}}$	$(0.25)_{\text{dix}}$	$[1, 1, 0]$
$(0.25)_{\text{dix}}$	$(0.5)_{\text{dix}}$	$(0.5)_{\text{dix}}$	$[0, 1, 1, 0]$
$(0.5)_{\text{dix}}$	$(1.0)_{\text{dix}}$	$(0.0)_{\text{dix}}$	$[1, 0, 1, 1, 0]$
$(0)_{\text{dix}}$	-	-	$[0, 1, 1, 0, 1]^*$

* Je me suis trompé de sens, je l'ai donc inversé à la fin ; le calcul est globalement correct.

$$(52.40625)_{\text{dix}} = (0011\ 0100.011\ 0100)_{\text{vf}}$$

2. IEEE₇₅₄ SINGLE

$s := 0$ $\text{vale}(e) = (5)_{\text{dix}} = (0000\ 0101)_{\text{deux}}$ $e = (1000\ 0100)_{\text{biais}=127}$

$B := 127$

$|x| = (110100.1011)_{\text{vf}} = \text{valm}(1101001011) * 2^5$ (5 sauts jusqu'au 1 de poids le plus fort)

$m = 101000110100000000000000$

$(52.40625)_{\text{dix}} = (0\ 10000100\ 101001101000000000000000)_{\text{IEEE754single}}$

2. Le processeur et les registres

2.1 Registres et sous-registres

Exercice 8.

Registre	Décimal (non signé)		Décimal (signé)		Hexadécimal	
	Max	Min	Max	Min	Max	Min
eax	4 294 967 295	0	2 147 483 647	-2 147 483 647	0xFFFFFFFF	0x0
ax	65 535	0	32 767	-32 767	0xFFFF	0x0
ah	255	0	127	-127	0xFF	0x0
al	255	0	127	-127	0xFF	0x0

Le décimal signé correspond au type **int** codé sur 32, 16 et 8 bits respectivement.

2.2 Opérations sur les registres

Exercice 9.

mov eax, 134512768 va recopier :

eax = 0x8048080

ax = 0x8080

ah = 0x80

al = 0x80

mov al, 0 va recopier 0x0 dans le registre al :

eax = 0x8048000

ax = 0x8000

ah = 0x80

al = 0x00

Exercice 10.

mov ebx, 0x0 ; ebx = 0x0

mov bh, 0xAB ; ebx = 0x0000AB

mov bh, 0xCD ; ebx = 0x0000ABCD

Exercice 11.

mov eax, 0xF0000F0

eax = 0xF0000F0			
		ax = 0x00F0	
ah = 0x00		al = 0xF0	

`add ax, 0xF000`

<code>eax = 0xF000F0F0</code>			
<code>ax = 0xF0F0</code>			
<code>ah = 0xF0</code>		<code>al = 0xF0</code>	

`add ah, al`

<code>eax = 0xF000F001E0F0</code>			
<code>ax = 0x8001</code>			
<code>ah = 0xE0</code>		<code>al = 0xF0</code>	

F0 + F0 = 1E0. Ce nombre, converti en binaire, tient sur 3 octets ... Il y a donc un dépassement de capacité.

2.3 Lecture/écriture en mémoire

Exercice 12. (Mémoire)

1. Une case mémoire peut représenter 2^8 valeurs différentes ; il s'agit du nombre de combinaisons possibles de 0 et 1 sur un octet.

2. $2^{32}/(1024^3) = 4$ Gio. Une mémoire est codée sur 32 bits.

3. $0x0000100A = 4106$

Il y a 4106 cases avant celle-ci, car on compte aussi la case d'indice 0.

2.3.1 Lecture en mémoire

Exercice 13.

`mov eax, 3` va recopier sur le registre `eax` la valeur 3.

`mov eax, [3]` va recopier sur le registre les valeurs stockées en mémoire à l'adresse 3, 4, 5 et 6, car on aura besoin de remplir `eax` avec 4 octets.

Exercice 14.

En convention *little-endian*, les instructions suivantes vont recopier :

1. `mov ax, [1]` → les valeurs pointées par [1] et par [2] sur `ax`, soit : 0x5A1E

2. `mov ah, [0]` → la valeur pointée par [0] sur `ah`, soit : 0x3

3. `mov eax, [0]` → les valeurs pointées par [0], [1], [2] et [3] sur `eax`, soit : 0xA5A1E3

4. `mov eax, [1]` → les valeurs pointées par [1], [2], [3] et [4] sur `eax`, soit : 0x10A5A1E

Exercice 15.

`eax` → 0100 0100

2.3.2 Écriture en mémoire

Exercice 16.

mov eax, 0x04030201

<i>eax</i> = 0000 0100 0000 0011 0000 0010 0000 0001			
		<i>ax</i> = 0000 0010 0000 0001	
		<i>ah</i> = 0000 0010	<i>al</i> = 0000 0001

mémoire	<i>mov [0], eax</i>	<i>mov [2], ax</i>	<i>mov [3], al</i>	mémoire finale
[0]	0000 0001	0000 0001	0000 0001	0000 0001
[1]	0000 0010	0000 0010	0000 0010	0000 0010
[2]	0000 0011	0000 0001	0000 0001	0000 0001
[3]	0000 0100	0000 0010	0000 0001	0000 0001

Exercice 17.

mémoire	<i>mov dword [0], 0x02001</i>	<i>mov byte [1], 0x21</i>	<i>mov word [2], 0x1</i>	mémoire finale
[0]	0000 0001	0000 0001	0000 0001	0000 0001
[1]	0010 0000	0010 0001	0010 0001	0010 0001
[2]	0000 0000	0000 0000	0000 0000	0000 0000
[3]	0000 0000	0000 0000	0000 0000	0000 0000

Exercice 18.

eax → 0x1

eax → 0x1000000

3. Premier programme

3.1 Compilation

nasm -f elf32 Hello.asm ; *nasm* est le compilateur des fichiers .asm

ld -o Hello -melf_i386 -e main Hello.o ; *ld* va faire le lien entre les fonctions et les fichier objet

Pour compiler sur linux, il suffit d'écrire la commande : *./make.sh NomExecutable*. Ce script va recompiler le programme ; il va également créer le fichier .lst puis éliminer les fichiers objet.

3.3 Assemblage du programme

Exercice 20.

La section *.data* de E19 est plus grande que celle de Hello, ce qui est lié au fait que la chaîne de caractères déclarée est plus grande. On va aussi modifier le nombre de caractères affichés placé dans *edx*.

Exercice 21.

La dernière instruction du code se trouve à l'adresse 0x36 et tient sur deux octets : le code s'étend donc sur 54 + 1 = 55 octets. Réellement, le code occupera **56 octets** car il faut un multiple de 4.

Conclusion

L'assembleur est un langage bas niveau qui nécessite une bonne maîtrise de la mémoire du processeur et de la machine pour pouvoir le prendre en main. Le code sera divisé en trois sections principales : la première *.data* qui va avoir des constantes qui ne pourront pas être modifiées au runtime, seulement utilisées ; la seconde *.bss* qui va servir à déclarer des variables qui pourront être utilisées et modifiées dans le programme ; et finalement, *.text*, qui aura les fonctions exécutées dans le *main* du programme.

Le langage assembleur n'ayant que des instructions de calcul très simples, il va devoir appeler des fonctions enregistrées dans le kernel du système d'exploitation avec une instruction d'interruption, *int 0x80*. Avant d'appeler les fonctions, il faudra placer dans des registres précis les paramètres nécessaires pour la fonction.

Dans le cadre de ce TP, seules des constantes déclarées dans *.data* ont été utilisées pour afficher un petit texte sur le terminal de la machine ; ensuite une petite étude de la taille du programme et de l'emplacement en mémoire de chaque partie a été réalisée.