Année 2009-2010

Examen de
Circuits et Architecture des Ordinateurs
éléments de corrections


corrigé partiel et non officiel,
à lire d'un œil critique


Exercice 1 :

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23		50		4	
50		80					
-27		-80					
-70		-12					

Méthode :

On ne détaillera pas les calculs du niveau primaire :

si n = 50 et m = 80
n + m = 50 + 80 = 130
etc

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23		50		4	
50		80		130		-30	
-27		-80		-107		53	
-70		-12		-82		-58	

Rappels de L1 Info

- Le principe du binaire :

Système Décimal	Système Binaire
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
...	...

On note $(x)_2$ pour préciser que x est en base 2 (binaire)
$(x)_{10}$ pour préciser que x est en base 10 (système décimal courant)
$(x)_{16}$ pour préciser que x est en base 16
etc

- Pour passer d'un binaire à un décimal (par exemple 00011011) :

position des chiffres	7	6	5	4	3	2	1	0
nombre binaire	0	0	0	1	1	0	1	1

Pour chaque 1 en bas, on ajoute 2^i où i est la position du bit indiquée en haut
Donc $(00011011)_2 = 2^4 + 2^3 + 2^1 + 2^0 = 16+8+2+1 = 27$ ($(27)_{10}$ pour être précis)

- Pour passer d'un décimal à un binaire (par exemple 50) :

On fait la division entière de ce nombre par 2, on note le reste (forcement 0 ou 1), et on répète le processus sur le quotient, jusqu'à atteindre 0.

50/2 = 25, reste 0
25/2 = 12, reste 1
12/2 = 6, reste 0
6/2 = 3, reste 0
3/2 = 1, reste 1
1/2 = 0, reste 1

Ensuite, on lit les restes du bas vers le haut, et on obtient la conversion en binaire de 50 : 110010 (ce qui remplit la 2ème ligne de la 2ème colonne du tableau)

Autre méthode :

Avec des nombres assez petits, et si on connait bien ses puissances de 2, on peut aussi prendre la puissance de 2 immédiatement inférieure et procéder ainsi :

$50 \geq 32 \rightarrow$ oui : 1 | $50 - 32 = 18$
$18 \geq 16 \rightarrow$ oui : 1 | $18 - 16 = 2$
$2 \geq 8 \rightarrow$ non : 0
$2 \geq 4 \rightarrow$ non : 0
$2 \geq 2 \rightarrow$ oui : 1 | $2 - 2 = 0$
$0 \geq 1 \rightarrow$ non : 0

(revient à voir que 50 = 32 + 16 + 2)

Ce qui donne aussi que 50 est équivalent à 110010

De la même façon :

- Pour 23 :

Division	Quotient	Reste
23/2	11	1
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

23 → 10111

- Et pour 80 :

Division	Quotient	Reste
80/2	40	0
40/2	20	0
20/2	10	0
10/2	5	0
5/2	2	1
2/2	1	0
1/2	0	1

80 → 1010000

Comme ces nombres sont codés sur 8 bits, on complète avec des 0 devant, jusqu'à avoir 8 chiffres.

Ce qui donne :

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23	00010111	50		4	
50	00110010	80	01010000	130		-30	
-27		-80		-107		53	
-70		-12		-82		-58	

Rappels de cours

Une façon de noter les nombres binaires négatifs est de les coder en complément à 1, c'est-à-dire, pour passer de x à -x, d'inverser tous les chiffres de x (les 1 deviennent des 0, et les 0 deviennent des 1).
Par exemple, sur 8 bits :
27 → 00011011
-27 → 11100100

Dans cette représentation, les entiers commençant par un 0 sont des nombres positifs, et ceux commençant par un 1 des nombres négatifs.
Les entiers codés en complément à 1 sur n bits vont donc de $-2^{n-1}-1$ à $2^{n-1}-1$ (de -127 à 127 sur 8 bits).
Remarque : Ici, 00000000 et 11111111 sont tous les 2 des représentations de 0.

Une autre façon de noter les nombres binaires négatifs (très proche, mais plus puissante, et beaucoup plus utilisée aujourd'hui) est de les coder en complément à 2, c'est-à-dire, pour passer de x à -x, de noter d'abord x en complément à 1, puis d'additionner 1.
Par exemple, sur 8 bits :
27 → 00011011
-27 → 11100100 +1 = 11100101
(pour poser une addition en binaire, c'est comme en décimal)

Là aussi, les entiers commençant par un 0 sont des nombres positifs, et ceux commençant par un 1 des nombres négatifs (et même *strictement* négatifs cette fois).
Les entiers codés en complément à 2 sur n bits vont donc de -2^{n-1} à $2^{n-1}-1$ (de -128 à 127 sur 8 bits).
Remarque : Ici, 00000000 représente 0, mais 11111111 représente -1.

- pour -70 :

Calculons 70 :

Division	Quotient	Reste
70/2	35	0
35/2	17	1
17/2	8	1
8/2	4	0
4/2	2	0
2/2	1	0
1/2	0	1

70 → 1000110
= 01000110 (sur 8 bits)

-70 → 10111001 +1 = 10111010

- pour -80 :

On a déjà calculé 80

80 → 1010000
= 01010000

-80 → 10101111 +1 = 10110000

- pour -12 :

Calculons 12 :

Division	Quotient	Reste
12/2	6	0
6/2	3	0
3/2	1	1
1/2	0	1

12 → 1100
= 00001100

-12 → 11110011 +1 = 11110100

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23	00010111	50		4	
50	00110010	80	01010000	130		-30	
-27	11100101	-80	10110000	-107		53	
-70	10111010	-12	11110100	-82		-58	

Rappels de cours

Un additionneur 8 bits fait l'addition de 2 nombre de 8 bits et renvoie un résultat sur 8 bits. Si le résultat fait 9 bits, ce qui arrive parfois, le bit de poids fort (c'est-à-dire le plus à gauche) est ignoré, on dit alors qu'il y a un *dépacement*.
Remarque : Le résultat de l'addition peut parfois être le bon même avec un dépacement, ou faux sans dépacement : la validité du résultat dépend juste de ce qu'il soit compris entre -2^{n-1} et $2^{n-1}-1$

---

27 + 23 = 00011011 + 00010111 :

```
  00011011
+ 00010111
  00110010
```

Autre méthode :

Tant que la somme est comprise entre -128 et 127, l'additionneur 8 bits donnera une valeur correcte, autrement dit, on pouvait aussi reprendre la valeur qu'on avait trouvé pour 50.

De la même façon :

50 + 80 :

```
  00110010
+ 01010000
  10000010
```

Remarque : Ça n'était pas demandé, mais on peut remarquer que l'additionneur donne une valeur fausse si la somme n'est pas comprise entre -128 et 127 (forcement, puisqu'on ne peut pas coder ces nombres sur 8 bits), ici, il donne 10000010, ce qui vaut normalement -126, alors que 50 + 80 = 130. C'est une sorte de boucle (comme le cercle trigo, où même avec le bon *cos* et le bon *sin* on peut se tromper de 2π), il va généralement se tromper de + ou – 256, et 130 - 256 = -126.

-27 + (-80) :

```
  11100101
+ 10110000
 ¹10010101
```

(on ignore le 1 qui est en dehors des 8 bits, et on obtient bien -107)

-70 + (-12) :

```
  10111010
+ 11110100
 ¹10101110
```

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23	00010111	50	00110010	4	
50	00110010	80	01010000	130	10000010	-30	
-27	11100101	-80	10110000	-107	10010101	53	
-70	10111010	-12	11110100	-82	10101110	-58	

En ce qui concerne la dernière colonne, l'énoncé est un peu obscur, puisqu'il parle de faire la soustraction de 2 nombre à l'aide d'un additionneur, sans plus de précision.
Nous allons supposer qu'il s'agit d'un aditionneur comme celui ce la page 51 du cours, qui prend simplement la négation du second nombre et additionne le tout.
Donc, comme l'additinneur précédent, il donne le véritable résultat m-n tant que celui-ci est compris entre -128 et 127.

Puisque les résultats de la 7ème colonne sont compris entre -128 et 127, pas la peine de prendre les opposés des *n* et de les additionner aux *m*, on va simplement convertir la 7ème colonne en binaire pour aller plus vite (méthode rapide, avec les puissance de deux) :

$4 = 2^2 \rightarrow 100 =$ 00000100

$30 = 16 + 8 + 2 = 2^4 + 2^3 + 2^1 \rightarrow 11010 = 00011010$
$-30 \rightarrow 11100101 + 1 =$ 11100110

$53 = 32 + 16 + 4 + 1 = 2^5 + 2^4 + 2^2 + 2^0 \rightarrow 110101 =$ 00110101

$58 = 32 + 16 + 8 + 2 = 2^5 + 2^4 + 2^3 + 2^1 \rightarrow 111010 = 00111010$
$-58 \rightarrow 11000101 + 1 =$ 11000110

Ce qui complète le tableau

Correction :

m		*n*		*m* + *n*	*m* ⊕ *n*	*m* - *n*	*m* ⊖ *n*
décimal	binaire	décimal	binaire	décimal	binaire	décimal	binaire
27	00011011	23	00010111	50	00110010	4	00000100
50	00110010	80	01010000	130	10000010	-30	11100110
-27	11100101	-80	10110000	-107	10010101	53	00110101
-70	10111010	-12	11110100	-82	10101110	-58	11000110

Note :

Pour comprendre les exercices 2 à 4, il faut lire le chapitre 10 du cours, sur le processeur LC-3, et en particulier le point 10.4.2 (le tableau de la page 73) qui récapitule les instructions du processeur.

Exercice 2 :

Question a :

Il y avait plusieurs solutions, en voici une.
Les phrases entre parenthèses et les commentaires du code ne sont peut-être pas indispensables pour l'examen.

Correction :

Idée d'algorithme :

(1) On compte le nombre de 1 dans R0
(2) On en déduit la parité du nombre de 1

(1) : On initialise un autre registre à 0, le registre R1 par exemple, il sera le compteur du nombre de 1 dans R0.
Tant que R0 est différent de 0 (car alors il vaut 00...000, il n'y a plus aucun 1)
Si R0 est négatif (c'est-à-dire s'il commence par un 1, cf exercice 1)
Alors on incrémente R1 de 1
On multiplie R0 par 2 (ce qui, sur 8 bits, décale les chiffres vers la gauche, en faisant disparaître le plus à gauche, et en ajoutant un 0 à droite)
(Note : la boucle se répète au maximum 16 fois)

(2) : On récupère le bit de parité du compteur R1 (le bit le plus à droite de R1) et on l'enregistre dans R0.

Rappels de cours page 73

ADD DR, SR1, SR2 :
Fait l'addition binaire bit à bit des 2 valeurs sur 16 bits contenus respectivement dans les registres SR1 et SR2, et met le résultat dans le registre DR (DR = SR1 + SR2).

ADD DR, SR1, Imm5 :
Fait la même chose que le précédent, mais Imm5 est une contante codée sur 5 bits.

AND DR, SR1, SR2 :
Fait le *and* (et) logique bit à bit des 2 valeurs sur 16 bits contenus respectivement dans les registres SR1 et SR2, et met le résultat dans le registre DR.

AND DR, SR1, Imm5 :
Fait la même chose que le précédent, mais Imm5 est une contante codée sur 5 bits.

BRn label :
Teste si le dernier registre dans lequel on a écrit est négatif (commence par un 1), si oui, il conduit à la ligne commençant par "label :", sinon, il ne fait rien.

BRz label :
Pareil que BRz, mais il teste si le dernier registre est nul (ne contient que des 0).

BRp label :
Pareil, mais teste s'il est positif (commence par un 0).

BRnz label :
Teste s'il est négatif *ou* nul.

etc, avec les autres combinaisons de [n][z][p].

Code de l'algo :

```
parity :    AND R1, R1, 0       ; écrit 00...000 dans R1
            AND R0, R0, -1      ; instruction neutre, pour mettre les BR sur R0
            BRz fin             ; si R0 est nul, on saute à la branche fin

loop :      BRp suite           ; si R0 est positif (commence par un 0), on
                                  saute la ligne suivante
            ADD R1, R1, 1       ; on incrémente le compteur de 1 (R0  négatif)

suite :     ADD R0, R0, R0      ; R0 = R0 + R0, on double R0 pour décaller les
                                  chiffres vers la gauche
            BRnp loop           ; si R0 n'est pas nul, on retourne dans la boucle
                                  (sinon, on passe à la ligne suivante : fin)

fin :       AND R0, R1, 1       ; on fait un et logique entre R1 et 1 pour avoir
                                  son bit de poids faible (ou de parité) et on le
                                  met dans R0
            RET
```

(Question b et c non traitées pour l'instant)

Exercice 3 :

Question a :

Méthode :

On sait que le processeur LC-3 a un op-code non utilisé, on le trouve en bas du tableau de la page 73 du cours : 1101.
La première solution était donc de l'utiliser pour coder le XOR.

Pour la seconde solution, il fallait trouver une opération que l'on pouvait remplacer par XOR, c'est-à-dire une opération dont on puisse ensuite se passer, donc une opération que l'on puisse facilement simuler avec le XOR.
Il y avait peut-être plusieurs réponses possible, la plus simple était d'utiliser le NOT (op-code 1001), car NOT(x) est équivalent à XOR(x,1).

Correction :

On peut utiliser l'op-code libre (1101) ou le l'op-code du NOT (1001), car on pourra toujours faire un NOT de R0 en faisant XOR de R0 et de R1, où R1 est un registre rempli de 1.

Question b :

Méthode :

Puisqu'il est demandé que le codage de cette instruction soit cohérent avec le codage des autres insctructions du LC-3, on regarde les autres pour s'en inspirer, et en particulier, l'instruction la plus proche : AND (le seul autre opérateur logique binaire).
On voit, dans le tableau de la page 73 du cours, que AND existe sous 2 formes, l'une prend 2 registres (4ème ligne du tableau) et l'autre prend un registre et une constante (5ème ligne), ce qui est précisement ce qu'il est demandé de faire pour le XOR dans l'énoncé, on recopie donc ces 2 lignes en les adaptant un peu (on remplace AND par XOR, et l'op-code de AND par l'un des 2 qu'on a choisi à la question 1).

Correction :

Si on choisi, par exemple, l'op-code 1101 pour XOR :

Pour l'opération entre 2 registres :
XOR DR,SR1,SR2 : 1101 DR SR1 000 SR2

Pour l'opération entre un registre et une constante :
XOR DR,SR1,Imm5 : 1101 DR SR1 1 Imm5

(Exercices 4 et 5 traités prochainement)