Travaux Dirigés d'architecture des machines

1 Planning du cours Architecture des Machines

- I. Histoire de l'informatique
- II. Représentation des informations et arithmétique
- III. Circuits logiques et algèbre de Boole
- IV. Présentation générale
- V. Le processeur
- VI. Assembleur
- VII. La mémoire
- VIII. Les entrées/sorties
- IX. Performances

2 Bibliographie

Carter Nicholas P., *Architecture de l'ordinateur*, EdiScience, 2002.
Cazes A. & Delacroix J., *Architecture des machines et des systèmes informatiques*, 4e édition, Dunod, 2011.
Hennessy J.L. & Patterson D.A., *Architecture des ordinateurs*, 3e édition, Vuibert, 2003.
Lazard E., *Collection Synthex, Architecture de l'ordinateur*, Pearson Education, 2006.
Tanenbaum A., *Architecture de l'ordinateur*, 5e édition, Pearson Education, 2005.
Zanella P., Ligier Y. & Lazard E., *Architecture et technologie des ordinateurs*, 6e édition, Dunod, 2018.

3 Exercices représentation des nombres

Exercice 1 *Exprimer le nombre décimal* 100 *dans toutes les bases de* 2 *à* 9 *et en base* 16.

CORRIGÉ : 1100100_2 10201_3 1210_4 400_5 244_6 202_7 144_8 121_9 64_{16}

Exercice 2 *Multiplier* 0111 *et* 0101 *en binaire.*

CORRIGÉ :

		0	1	1	1
		0	1	0	1
		0	1	1	1
0	1	1	1	0	0
1	0	0	0	1	1

Exercice 3

1. *Exprimer les nombres décimaux* 94, 141, 163 *et* 197 *en bases* 2, 8 *et* 16.
2. *Donner sur* 8 *bits les représentations*
   (a) *signe et valeur absolue ;*
   (b) *complément à 1 ;*
   (c) *complément à 2*

   des nombres décimaux 45, 73, 84, −99, −102 *et* −118.
3. *Effectuer la soustraction de* 122 *et* 43 *dans la représentation en complément à 2 en n'utilisant que l'addition.*

CORRIGÉ :

1. $94_{10} = 1011110_2 = 136_8 = 5E_{16}$
   $141_{10} = 10001101_2 = 215_8 = 8D_{16}$
   $163_{10} = 10100011_2 = 243_8 = A3_{16}$
   $197_{10} = 11000101_2 = 305_8 = C5_{16}$

2.

	S et V	Comp. à 1	Comp. à 2
45	0010 1101	0010 1101	0010 1101
73	0100 1001	01001 001	0100 1001
84	0101 0100	0101 0100	0101 0100
-99	1110 0011	1001 1100	1001 1101
-102	1110 0110	1001 1001	1001 1010
-118	1111 0110	1000 1001	1000 1010

3. $122 - 43 = 122 + (-43)$

122 :	01111010
−43 :	11010101
79 :	01001111

Exercice 4

1. *On considère l'opération décimale* $101 + 99$. *Expliquer pourquoi un programme peut « donner » deux résultats différents. Quels sont ces résultats et de quoi dépend le fait que le programme donne l'un ou l'autre ?*

2. *Votre nouveau programme est en train de compter le nombre de caractères d'un fichier. Vous savez qu'il y en a environ 50 000. Finalement votre programme s'arrête et affiche `-14532` caractères. Quel bug avez-vous fait et quel est le nombre de caractères du fichier ?*
3. *On considère le programme C suivant :*

Listing 1. Calculs

```c
#include <stdio.h>

main() {
    char c;
    char str[500];
    gets(str);       /* récupère une chaîne au clavier et la met dans str */
    c = strlen(str); /* renvoie la longueur de la chaîne */
    printf("longueur : %d\n", c);
}
```

Pensez-vous que ce programme affiche toujours la bonne valeur ?
Quelles sont ses limites de fonctionnement ?

4. *On considère le programme C suivant :*

Listing 2. Signé/non-signé

```c
#include <stdio.h>

main() {
    short i;
    unsigned short j;

    i = -1;
    j = i;
    printf("%d\n", j);
}
```

Quelle est la valeur affichée ?
Quelle serait la valeur affichée si la représentation des nombres était « signe et valeur absolue » ? Et si c'était la représentation en complément à un ?

5. *Le langage C possède plusieurs types pour représenter les nombres entiers. Lesquels ?*
Quels sont les types entiers en Java ?
Quelle question vous posez-vous avant de choisir un type pour votre variable ? Quelles sont les valeurs critiques ?

Corrigé :

1. Si tout se passe bien, le résultat est celui attendu, c'est-à-dire 200 mais si les deux nombres 101 et 99 sont stockés sur un octet signé, le résultat aussi sera stocké sur un octet signé, qui ne peut pas représenter 200 mais donnera comme résultat −56. La représentation binaire du résultat sera toujours la même (11001000) mais le programme « lira » cette valeur de deux manières différentes suivant le type de la variable. Le seul type posant problème est l'octet signé (`char` en C) car il représente les nombres de −128 à 127 ; tous les autres types donneront un résultat correct, entre autres le type `unsigned char`. En Java, l'addition porte forcément sur des entiers de type `int`, donc sur 32 bits. Le résultat du calcul ci-dessus est donc toujours correct mais si le résultat dépassait 2^{31}, il y aurait le même problème.
2. En C ou en Java, on a utilisé une variable de type `short` qui stocke un nombre signé sur 16 bits, donc une valeur de −32 768 à 32 767. Il est impossible d'afficher une valeur proche de 50 000. Le nombre affiché est donc le complément à 2 du nombre réel, qui est de 65 536 − 14 532 = 51 004 caractères.

3. Une variable de type `char` en C peut stocker une valeur de -128 à 127. Si la chaîne récupérée fait moins de 128 caractères, le programme affichera la bonne valeur ; si elle fait plus, le programme affichera soit une valeur négative (par exemple si la chaîne a une longueur entre 128 et 255), soit le modulo 256 de la longueur (par exemple si la chaîne a une longueur entre 256 et 384), voire même une vague combinaison des deux si la chaîne est plus grande... Si on veut pouvoir mettre une chaîne de longueur inférieure à 255, il faut au minimum utiliser une variable de type `unsigned char` et si la chaîne peut être plus longue, un `short` ou un `int` est nécessaire. Ici, *théoriquement*, la chaîne est limitée à 500 caractères par la taille du tableau mais `gets` ne fait aucune vérification et si l'utilisateur rentre une chaîne plus longue, il y aura des problèmes d'écriture mémoire sur une zone non-prévue, mais ce n'était pas le point de l'exercice.
4. -1 se représente en binaire comme $11\ldots11$ (sur 16 bits). Comme `j` est un `unsigned short`, on doit le lire comme un nombre positif ; le programme affiche donc 65535. Si la représentation est « signe et valeur absolue », -1 s'écrit $100\ldots001$ et donc le programme affiche 32769 ; si c'est la représentation en complément à un, -1 s'écrit $111\ldots110$ et le programme affiche 65534.
5. On peut distinguer six types : `char`, `short`, `long` qui peuvent être spécifiés `unsigned` ; le type `int` est équivalent à `short` ou `long` suivant le compilateur.
   En Java, on a les types `byte`, `char`, `short`, `int` et `long`.
   Avant de choisir un type pour une variable, il faut savoir quelles sont les valeurs qu'elle peut prendre. Voici les valeurs extrêmes des différents types :

type C	type java	val. min	val. max
`char`	`bytes`	-128	127
`unsigned char`		0	255
`short`	`short`	-32768	32767
`unsigned short`	`char`	0	65535
`long`	`int`	env. -2 milliards	env. 2 milliards
`unsigned long`		0	env. 4 milliards
`long long`	`long`	env. -10^{19}	env. 10^{19}

Exercice 5 *On considère une représentation simplifiée des réels en virgule flottante.*

Un nombre réel X *est représenté par* 10 *bits* $\boxed{s\,eeee\,mmmmm}$ *où* $X = (-1)^s * 1{,}m * 2^{e-7}$ *avec un exposant sur 4 bits (*$0 < e \leqslant 15$*, un exposant égal à* 0 *désigne le nombre* 0 *quelle que soit la mantisse) et une pseudo-mantisse sur 5 bits (représentant les puissances négatives de* 2*).*

1. *Quels sont le plus grand nombre et le plus petit nombre strictement positifs représentables ?*
2. *Comment se représente le nombre* 1 *? La précision* ϵ *d'une représentation est l'écart entre* 1 *et le nombre représentable suivant. Combien vaut* ϵ *pour cette représentation ?*
3. *Peut-on représenter* 7,2 *et* 57,6 *Quels sont les nombres qui encadrent* 7,2 *et* 57,6 *dans cette représentation ?*
4. *Que donne en décimal la multiplication de* 7,25 *et* 28,5 *? Écrivez* 7,25 *et* 28,5 *dans la représentation proposée et effectuez la multiplication. Quels sont les deux arrondis possibles pour le résultat et leur valeur décimale ?*

Corrigé :

1. $M = \boxed{0\,1111\,11111} = (1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32) \times 2^8 = 504$
   $m = \boxed{0\,0001\,00000} = 1 \times 2^{-6} = 1/64 = 0{,}015625$
2. $1 = 1 \times 2^0 = \boxed{0\,0111\,00000}$. Le nombre représentable suivant est $\boxed{0\,0111\,00001} = 1 + 1/32$; donc $\epsilon = 1/32$.
3. $7{,}2 = 1{,}8 \times 2^2$. Mais on ne peut pas écrire une pseudo-mantisse égale à 0,8. Elle se situe entre 11001_2 (qui vaut 0,78125) et 11010_2 (qui vaut 0,8125). Les deux nombres qui encadrent 7,2 dans cette représentation sont donc $1{,}78125 \times 2^2 = 7{,}125$ et $1{,}8125 \times 2^2 = 7{,}25$. De même, $57{,}6 = 1{,}8 \times 2^5$ et on se retrouve dans

la même situation, les deux nombres encadrant 57,6 étant alors $1{,}78125 \times 2^5 = 57$ et $1{,}8125 \times 2^5 = 58$. Remarquez que l'écart entre deux nombres de la représentation n'est pas constant mais augmente avec la valeur des nombres.

4. $7{,}25 \times 28{,}5 = 206{,}625$. On a vu que 7,25 se représente exactement par 0 1001 11010 ($1{,}8125 \times 2^2$) et que 28,5 se représente exactement par 0 1011 11001 ($1{,}78125 \times 2^4$). La multiplication des deux mantisses $1{,}11010_2$ et $1{,}11001_2$ donne $11{,}001110101_2$ comme résultat, qu'il faut normaliser à $1{,}1001110101_2$ en ajoutant 1 à l'exposant qui devient $2 + 4 + 1 = 7$. Le problème est que cette mantisse a trop de bits significatifs pour cette représentation et qu'il faut donc l'arrondir. On peut tronquer la mantisse à $1{,}10011_2$ ou arrondir supérieurement à $1{,}10100_2$. Le premier cas donne $1{,}10011_2 \times 2^7 = 204$ et le second $1{,}10100_2 \times 2^7 = 208$. On s'aperçoit alors que même si deux nombres sont exactement représentables en virgule flottante, leur produit ne l'est pas forcément, d'où la nécessité d'avoir des règles d'arrondi très précises pour être sûr que les résultats soient reproductibles d'une machine à une autre.

Exercice 6 *On considère une représentation simplifiée des réels en virgule flottante.*

Un nombre réel X *est représenté par* 10 *bits* $\boxed{s\,eeee\,mmmm}$ *où* $X = (-1)^s * 1{,}m * 2^{e-7}$ *avec un exposant sur 4 bits (*$0 < e \leqslant 15$*, un exposant égal à* 0 *désigne le nombre* 0 *quelle que soit la pseudo-mantisse) et une pseudo-mantisse sur 5 bits (représentant les puissances négatives de* 2*).*

1. *Représenter* 16, 10 *et* 0,75 *en virgule flottante.*
2. *Pour additionner deux nombres en virgule flottante, il faut décaler la mantisse d'un des deux nombres pour égaliser les exposants, additionner les mantisses (sans oublier le* 1 *débutant la mantisse), puis renormaliser le nombre en arrondissant éventuellement la pseudo-mantisse.*
   Ainsi, on a $7 = 1{,}11000_2 \times 2^2$ *et* $1{,}25 = 1{,}01000_2 \times 2^0 = 0{,}0101000_2 \times 2^2$ *dont l'addition donne* $10{,}0001_2 \times 2^2$ *qui se normalise en* $1{,}00001_2 \times 2^3 = 8{,}25$.
   Expliciter l'addition flottante de 16 *et* 10*. Que donne l'addition de* 16 *et* 0,75 *si on arrondit inférieurement ?*
3. *On écrit le code suivant :*

Listing 3. Calculs

```
float f = 16.0;
for (i = 0; i < 10; i++)
    f = f + 0.75;
```

Quelle valeur aura la variable `f` *après l'exécution ? Et quelle aurait été sa valeur si on avait simplement écrit :*

```
float f = 16.0 + (10 * 0.75);
```

Corrigé :

1. $16 = 1 \times 2^4 = \boxed{0\,1011\,00000}$
   $10 = 1{,}25 \times 2^3 = \boxed{0\,1010\,01000}$
   $0{,}75 = 1{,}5 \times 2^{-1} = \boxed{0\,0110\,10000}$
2. Il faut décaler la mantisse de 10 d'un bit vers la droite pour égaliser les exposants à 1011. On additionne donc $1{,}00000_2$ et $0{,}101000_2$, ce qui donne une mantisse égale à $1{,}101000_2$. La somme est donc $\boxed{0\,1011\,10100}$ soit $1{,}625 \times 2^4 = 26$.
   Pour additionner 16 et 0,75, il faut décaler la mantisse de ce dernier de cinq bits vers la droite, ce qui donne $0{,}75 = 1{,}10000_2 \times 2^{-1} = 0{,}0000110000_2 \times 2^4$. L'addition donne un résultat égal à $1{,}00001\not{1}_2 \times 2^4$ et le dernier bit de la mantisse saute à cause de l'arrondi. Le résultat obtenu est alors $1{,}00001_2 \times 2^4 = 16{,}5$.

3. Dans le premier cas, à chaque passage dans la boucle, on effectue une addition qui s'arrondit en laissant tomber le dernier bit de 0,75... On additionne en fait 0,5 et pas 0,75. La variable `f` va donc avoir 21 pour valeur.
   Dans le deuxième cas, le calcul est exact car $10 \times 0{,}75 = 7{,}5 = 1{,}11100_2 \times 2^2$ qui se décale en $0{,}01111_2 \times 2^4$, dont l'addition avec 16 donne $1{,}01111_2 \times 2^4 = 23{,}5$.

4 Exercices circuits logiques

Exercice 7 *Soit un entier de* 0 *à* 7 *représenté par* 3 *bits* $b_2b_1b_0$. *Soit* F *la fonction de* $[0,7] \longrightarrow \{0,1\}$ *telle que* $F(0) = F(3) = F(4) = F(7) = 1$ *et* $F(1) = F(2) = F(5) = F(6) = 0$.

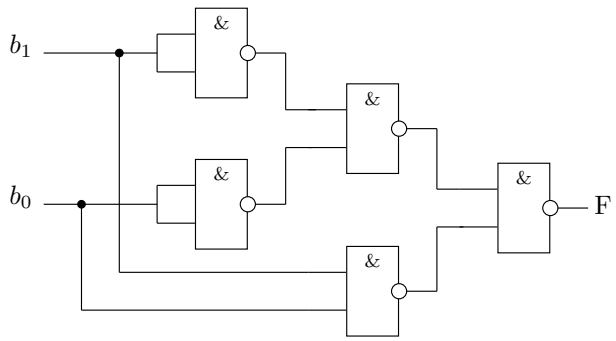
1. *Donner la table de vérité de* F.
2. *Donner l'expression algébrique de* F.
3. *Simplifier* F.
4. *Représenter le circuit logique de* F *uniquement à l'aide de portes NAND.*

Corrigé :

1.

b_2	b_1	b_0	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

2. $F = \overline{b_2}\,\overline{b_1}\,\overline{b_0} + \overline{b_2}b_1b_0 + b_2\overline{b_1}\,\overline{b_0} + b_2b_1b_0$
3. $F = (\overline{b_2} + b_2)(\overline{b_1}\,\overline{b_0}) + (\overline{b_2} + b_2)(b_1b_0) = \overline{b_1}\,\overline{b_0} + b_1b_0 = \overline{b_1 \oplus b_0}$
4. $F = \overline{\overline{\overline{b_1}\,\overline{b_0}}.\overline{b_1b_0}}$



Exercice 8 *Soit une machine qui travaille sur des nombres binaires signés de 3 bits. La valeur signée d'un mot* $a_2a_1a_0$ *est égale à* $a_0 + 2a_1 - 4a_2$ *(c'est la représentation classique en complément à 2).*

On désire construire un circuit qui donne en sortie $b_2b_1b_0$ *l'opposé de la valeur binaire d'entrée (par exemple, si on a* 2 *en entrée, on veut la valeur binaire* -2 *à la sortie).*

1. *Toutes les valeurs sont-elles autorisées en entrée ? Autrement dit, le circuit donne-t-il toujours une valeur correcte ?*
2. *Donner la table de vérité du circuit pour toutes les valeurs autorisées.*
3. *Donner les expressions logiques des* 3 *bits de sortie en fonction des* 3 *bits d'entrée en simplifiant au maximum les expressions et en ne tenant pas compte des valeurs interdites.*
4. *Donner l'expression logique d'un bit supplémentaire en sortie,* e (débordement), *qui indiquerait qu'une valeur interdite est présente sur les entrées.*

5. *Pouvez-vous généraliser certaines de ces expressions à des nombres de n bits ? Plus précisément, montrer que pour un circuit donnant l'opposé d'un nombre signé sur n bits, le bit de poids fort et le bit de poids faible de la sortie s'expriment facilement en fonction des bits d'entrée. Même question pour le bit de débordement.*

Corrigé :

1. Toutes les valeurs binaires sur 3 bits ont un opposé sauf $-4 = 100_2$ dont l'opposé, 4, ne s'exprime pas sur 3 bits en complément à deux.
2.

a_2	a_1	a_0	b_2	b_1	b_0	e
0	0	0	0	0	0	0
0	0	1	1	1	1	0
0	1	0	1	1	0	0
0	1	1	1	0	1	0
1	0	0	–	–	–	1
1	0	1	0	1	1	0
1	1	0	0	1	0	0
1	1	1	0	0	1	0

3. On voit donc que l'on a :
   $b_0 = a_0$
   $b_1 = \overline{a_2}\,\overline{a_1}a_0 + \overline{a_2}a_1\overline{a_0} + a_2\overline{a_1}a_0 + a_2a_1\overline{a_0} = \overline{a_2}(a_1 \oplus a_0) + a_2(a_1 \oplus a_0) = (a_1 \oplus a_0)$
   $b_2 = \overline{a_2}(a_1 + a_0)$ car le signe s'inverse sauf si le nombre est égal à 0.
4. $e = a_2\overline{a_1}\,\overline{a_0}$ c'est-à-dire pour la valeur -4.
5. Pour un nombre de n bits, la parité de son opposé est la même, on a donc toujours $b_0 = a_0$. De même, le signe s'inverse toujours, sauf pour 0, donc $b_{n-1} = \overline{a_{n-1}}(a_{n-2} + \ldots + a_0)$.
   Le bit de débordement est mis à 1 pour la seule valeur interdite, -2^{n-1} : $e = a_{n-1}\overline{a_{n-2}} \ldots \overline{a_0}$.

   On peut aussi trouver le même résultat pour $b_1 = a_1 \oplus a_0$. En effet, si $a_1 \oplus a_0 = 1$, c'est que le nombre s'écrit $4p + 1$ ou $4p + 2$; dans ce cas, son opposé s'écrit $2^n - 4p - 1 = 4p' + 3$ ou $2^n - 4p - 2 = 4p' + 2$ et dans les deux cas, le bit b_1 est à 1.
   Et de façon plus générale, on trouve que $b_i = a_i.(\overline{a_{i-1}} \ldots \overline{a_0}) + \overline{a_i}.(a_{i-1} + \ldots + a_0)$, ce qui donne $b_i = a_i \oplus (a_{i-1} + \ldots + a_0)$.

Exercice 9 *Soit une machine qui travaille sur des nombres binaires signés de 4 bits. La valeur signée d'un mot $A = a_3a_2a_1a_0$ est égale à $a_0 + 2a_1 + 4a_2 - 8a_3$ (c'est la représentation classique en complément à 2).*

On désire construire un circuit qui donne en sortie $B = b_3b_2b_1b_0 = -2 * A$ (par exemple, si on a 2 en entrée, on veut la valeur binaire -4 à la sortie).

1. *Toutes les valeurs sont-elles autorisées en entrée ? Autrement dit, le circuit donne-t-il toujours une valeur correcte ?*
2. *Donner la table de vérité du circuit pour toutes les valeurs autorisées.*
3. *Donner les expressions logiques des 4 bits de sortie en fonction des 4 bits d'entrée en* **simplifiant au maximum les expressions** *et en ne tenant pas compte des valeurs interdites (c'est-à-dire que les sorties peuvent être quelconques dans les cas interdits).*
4. *Donner l'expression logique d'un bit supplémentaire en sortie,* e (débordement)*, qui indiquerait qu'une valeur interdite est présente sur les entrées.*

Corrigé :

1. Les valeurs de -3 à 4 sont autorisées car, par exemple, $-2*-4 = 8$ et $-2*5 = -10$ qui ne se représentent pas sur 4 bits en complément à 2.

2.

a_3	a_2	a_1	a_0	b_3	b_2	b_1	b_0	e	e	b_3	b_2	b_1	b_0	a_3	a_2	a_1	a_0
0	0	0	0	0	0	0	0	0	1	–	–	–	–	1	0	0	0
0	0	0	1	1	1	1	0	0	1	–	–	–	–	1	0	0	1
0	0	1	0	1	1	0	0	0	1	–	–	–	–	1	0	1	0
0	0	1	1	1	0	1	0	0	1	–	–	–	–	1	0	1	1
0	1	0	0	1	0	0	0	0	1	–	–	–	–	1	1	0	0
0	1	0	1	–	–	–	–	1	0	0	1	1	0	1	1	0	1
0	1	1	0	–	–	–	–	1	0	0	1	0	0	1	1	1	0
0	1	1	1	–	–	–	–	1	0	0	0	1	0	1	1	1	1

3. On voit donc que l'on a :

$$b_0 = 0$$

$$b_1 = \overline{a_3}\,\overline{a_2}\,\overline{a_1}a_0 + \overline{a_3}\,\overline{a_2}a_1a_0 + a_3a_2\overline{a_1}a_0 + a_3a_2a_1a_0 = \overline{a_3}\,\overline{a_2}a_0 + a_3a_2a_0 = \overline{(a_3 \oplus a_2)}a_0$$

que l'on peut simplifier en

$$b_1 = a_0$$

car on ne s'intéresse pas aux cas non autorisés.

$$\begin{aligned} b_2 &= \overline{a_3}\,\overline{a_2}\,\overline{a_1}a_0 + \overline{a_3}\,\overline{a_2}a_1\overline{a_0} + a_3a_2\overline{a_1}a_0 + a_3a_2a_1\overline{a_0} = \overline{a_3}\,\overline{a_2}(a_1 \oplus a_0) + a_3a_2(a_1 \oplus a_0) \\ &= \overline{(a_3 \oplus a_2)}(a_1 \oplus a_0) \end{aligned}$$

que l'on peut simplifier en

$$b_2 = a_1 \oplus a_0$$

car on ne s'intéresse pas aux cas non autorisés.

$$b_3 = \overline{a_3}(a_2 + a_1 + a_0)$$

car le signe s'inverse sauf si le nombre est égal à 0.

4. En regroupant les cas de débordement en trois (-8 à -5 ; 5 à 7 ; -4), on trouve :

$$e = a_3\overline{a_2} + \overline{a_3}a_2(a_1 + a_0) + a_3a_2\overline{a_1}\,\overline{a_0}$$

Exercice 10

1. *On construit un circuit qui a 2 entrées (*a *et* b*) et 2 sorties (*s *et* r*) et une ligne de commande* F *tel que :*
   - *si* $F = 0$*, le circuit effectue une addition (*$a + b$ *avec une sortie* s *et une retenue* r*);*
   - *si* $F = 1$*, le circuit effectue une soustraction (*$a - b$ *avec une sortie* s *et une retenue* r*).*

   Donner la table de vérité (s *et* r *en fonction de* a *et* b*) pour* $F = 0$ *et celle pour* $F = 1$*.*
   Montrer que s *se calcule facilement avec une porte et* r *avec deux (on autorise NON, OU, ET et XOR). Dessiner le circuit de ce 1/2-additionneur/soustracteur.*

2. *On construit maintenant un additionneur/soustracteur complet, c'est-à-dire un circuit à trois entrées (a, b et r), deux sorties (s et r') et une ligne de commande F tel que :*
   - *si $F = 0$, le circuit effectue une addition ($a + b + r$ avec une sortie s et une retenue r');*
   - *si $F = 1$, le circuit effectue une soustraction ($a - (b + r)$ avec une sortie s et une retenue r').*

   Montrer que s s'exprime facilement en fonction de a, b et r.
   Donner la valeur de r' (éventuellement en fonction de a et F) dans les cas suivants :
   - $b = r = 1$;
   - $b = r = 0$;
   - $b = \overline{r}$.

   Donner une expression possible pour calculer r'.
3. *À partir des deux circuits précédents, proposer une construction pour un additionneur/soustracteur sur n bits, c'est-à-dire un circuit avec 2 fois n bits en entrée et qui effectue l'addition ou la soustraction de ces deux nombres suivant la valeur d'une ligne de commande.*

Corrigé :

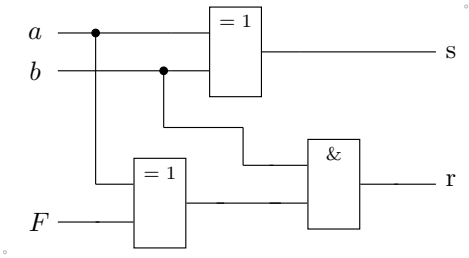
1.

a	b	F	s	r
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	0	0
0	1	1	1	1
1	0	1	1	0
1	1	1	0	0

$$s = a \oplus b$$

$$r = ab\overline{F} + \overline{a}bF = b(a\overline{F} + \overline{a}F) = b(a \oplus F)$$

Voici le dessin du $\frac{1}{2}$-add/sub.



2.

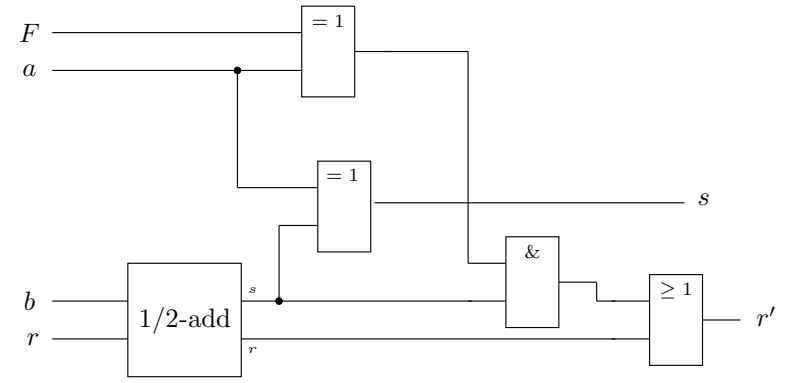
$$s = a \oplus b \oplus r$$

$$\begin{array}{rcl} b = r = 1 & \Rightarrow & r' = 1 \\ b = r = 0 & \Rightarrow & r' = 0 \\ b = \overline{r} & \Rightarrow & \left\{ \begin{array}{l} F = 0 \Rightarrow r' = a \text{ donc } r' = \overline{F}a \\ F = 1 \Rightarrow r' = \overline{a} \text{ donc } r' = F\overline{a} \end{array} \right\} \text{ donc } r' = F \oplus a \end{array}$$

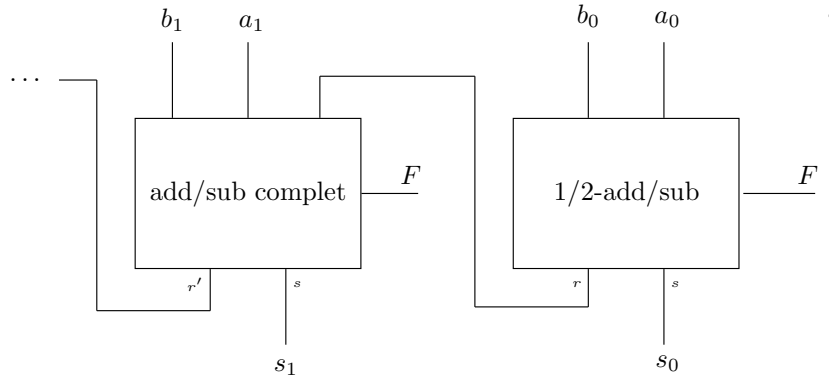
Au final, cela donne

$$r' = (b \oplus r)(F \oplus a) + br$$

Voici le dessin de l'add/sub complet.



3. C'est la même construction que pour un additionneur n bits.



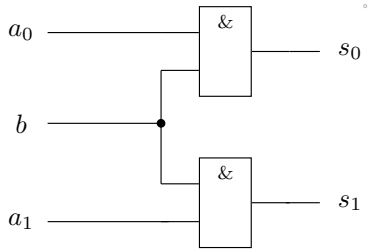
Exercice 11

1. *On souhaite construire un multiplicateur 2 bits par 1 bit (représentant des nombres entiers positifs), c'est-à-dire un circuit à trois entrées a_0, a_1 et b, et deux sorties s_0 et s_1 tel que :*
   - *si* $b = 0$, $s_0 = s_1 = 0$;
   - *si* $b = 1$, $s_0 = a_0$ *et* $s_1 = a_1$.

   Donner le schéma d'un tel circuit en utilisant 2 portes ET. On appelle M ce circuit.
2. *On veut maintenant construire un multiplicateur 2 bits par 2 bits. En décomposant cette multiplication en multiplications plus simples et en additions, montrer que l'on peut le faire avec deux circuits M et 2 demi-additionneurs. Donner le schéma du circuit correspondant.*
3. *Donner le schéma d'un multiplicateur 2 bits par n bits utilisant des circuits M, demi-additionneurs et additionneurs complets.*

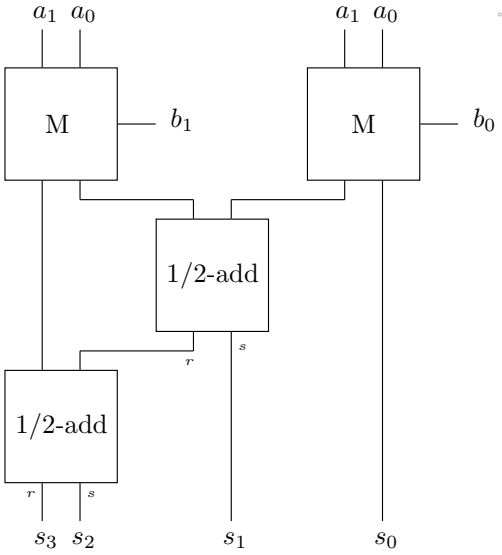
Corrigé :

1.



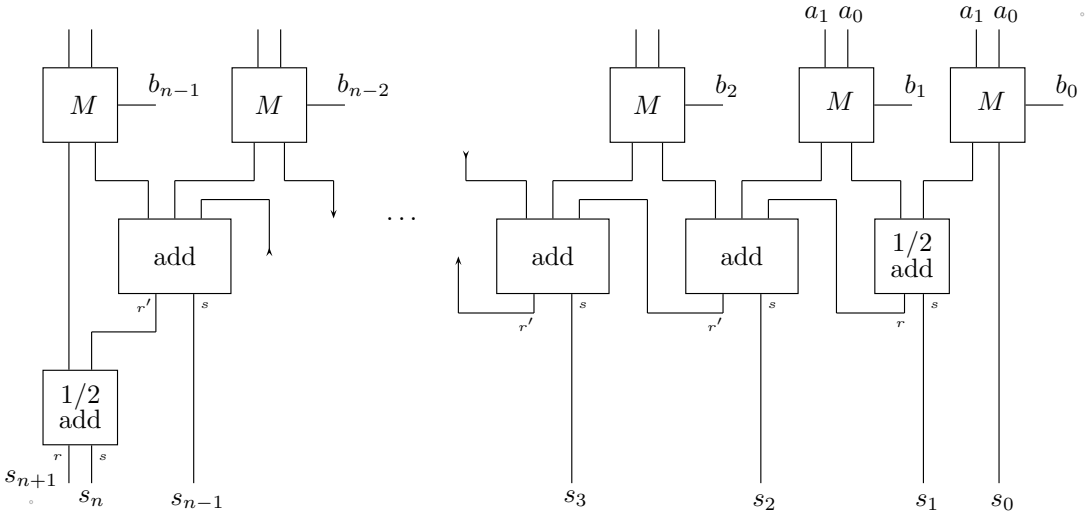
2.

		a_1	a_0
		b_1	b_0
		a_1b_0	a_0b_0
	a_1b_1	a_0b_1	
s_3	s_2	s_1	s_0



3.

					a_1	a_0
				b_{n-1} … …	b_1	b_0
					a_1b_0	a_0b_0
				a_1b_1	a_0b_1	
				a_0b_2		
			…			
		a_1b_{n-2}				
	a_1b_{n-1}	a_0b_{n-1}				
s_{n+1}	s_n	s_{n-1}	…	s_2	s_1	s_0



Soit n circuits M, deux demi-additionneurs et $n-2$ additionneurs complets.

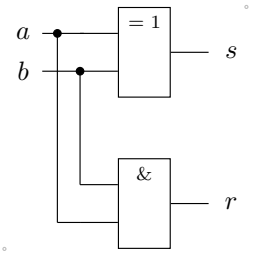
Exercice 12

1. *Donner la table de vérité d'un demi-additionneur 1 bit. Donner le schéma logique d'un demi-additionneur en utilisant une porte ET et une porte XOR.*
2. *Donner la table de vérité d'un additionneur complet. Donner le schéma logique d'un additionneur complet en utilisant deux demi-additionneurs et une porte OU.*
3. *Donner le schéma d'un additionneur 4 bits.*
4. *On suppose que le passage d'un signal électrique dans une porte « coûte » 10 ns. On dit alors qu'un circuit travaille en p ns si tous les signaux en sortie sont disponibles en au maximum p ns. En combien de temps un demi-additionneur, un additionneur complet et un additionneur 4 bits travaillent-ils ?*
5. *On va construire un additionneur 4 bits à retenue anticipée, c'est-à-dire un circuit où le calcul des retenues intermédiaires se fait plus rapidement. Soit une fonction de génération de retenue $G = ab$ et une fonction de propagation de retenue $P = a + b$; montrer que la retenue de sortie d'un additionneur complet peut se calculer par la formule $r' = Pr + G$, où a et b sont les entrées et r la retenue d'entrée.*
6. *Dans notre additionneur 4 bits, on note $G_i = a_i b_i$ et $P_i = a_i + b_i$, et r_i la retenue intermédiaire d'un étage, normalement calculée à partir des entrées a_i, b_i et de la retenue précédente r_{i-1}. Exprimer r_0, r_1, r_2 et $r_3 = r$ en fonction de G_0, G_1, G_2, G_3, P_0, P_1, P_2 et P_3.*
7. *En supposant que l'on dispose de plusieurs portes OU et ET à 2, 3 et 4 entrées, et que chacune travaille en 10 ns, en combien de temps se fait le calcul des G_i, P_i et r_i ? Est-ce intéressant ? Quelle différence y a-t-il entre le temps de travail d'un additionneur 8 bits normal et d'un additionneur 8 bits à retenue anticipée (avec les bonnes portes OU et ET) ?*

Corrigé :

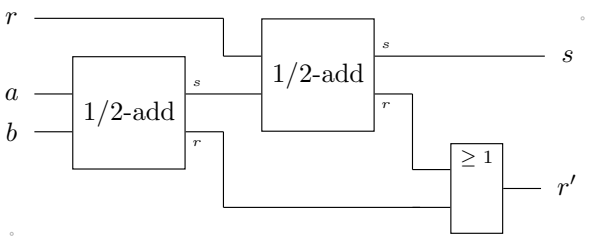
1.

a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

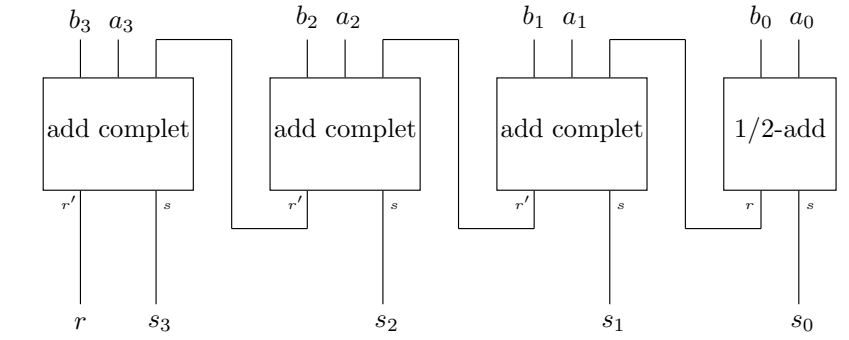


2.

a	b	r	s	r′
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



3.



4. Le demi-additionneur travaille en 10 ns, un additionneur complet travaille en 30 ns pour r' et 20 ns pour s, soit 30 ns au total (mais il faut différencier les sorties pour la question d'après).
   Pour l'additionneur 4 bits, on a les chiffres suivant pour chaque sortie :
   $s_0 : 10$ $s_1 : 20$ $s_2 : 40$ $s_3 : 60$
   $r_0 : 10$ $r_1 : 30$ $r_2 : 50$ $r_3 : 70$
   soit 70 ns au total.
5. $$r' = ab + ar + br + abr = ar + br + ab(r + 1) = r(a + b) + ab = rP + G$$
6. $$\begin{aligned} r_0 &= a_0 b_0 = G_0 \\ r_1 &= G_1 + r_0 P_1 = G_1 + G_0 P_1 \\ r_2 &= G_2 + r_1 P_2 = G_2 + (G_1 + G_0 P_1) P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 \\ r_3 &= G_3 + r_2 P_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 \end{aligned}$$
7. $G_i = a_i b_i$ et $P_i = a_i + b_i$ donc chacun est disponible en 10 ns. Les r_i sont formés de OU à partir de ET sur les G_i et P_i, ils se calculent donc tous en 30 ns. Donc s_0 se calcule en 10 ns, s_1 en 20 ns, s_2 en 40 ns et s_3 en 40 ns et c'est meilleur.
   Pour un additionneur 8 bits, le temps normal est de 150 ns ; avec un additionneur à retenue anticipée, le calcul se fait toujours en 40 ns mais il faut des portes ET et OU à 8 entrées.

Exercice 13 *On souhaite construire une machine à vote majoritaire sur 3 entrées* a_2, a_1 *et* a_0, *qui a deux sorties* M *et* U *telles que* M *représente le vote majoritaire des entrées (*M *vaut* 0 *s'il y a plus d'entrées à* 0, *et* 1 *s'il y a plus d'entrées à* 1*) et* U *(Unanimité) vaut* 1 *si les trois entrées sont égales,* 0 *sinon.*

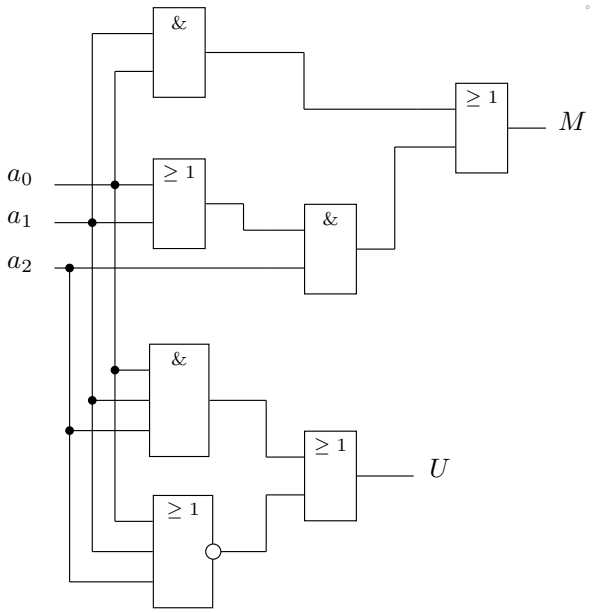
1. *Donner la table de vérité de la machine.*
2. *Donner les expressions logiques des* 2 *bits de sortie en fonction des* 3 *bits d'entrée.*
3. *Dessiner la machine en utilisant* exactement *:*
   - *2 portes OU et 2 portes ET pour* M *;*
   - *une porte OU, une porte NON-OU à trois entrées et une porte ET à trois entrées pour* U.
4. *On souhaite étendre la machine à 5 bits d'entrée et avoir un bit de sortie* M' *représentant le vote majoritaire des entrées. On suppose que l'on dispose de la machine majoritaire à 3 entrées ci-dessus donnant un bit* M *de sortie. Donner une expression logique pour* M' *en décomposant en trois cas : soit les trois premiers bits sont à* 1, *soit il y a une majorité de* 1 *dans les trois premiers bits, soit les* 2 *derniers bits sont à* 1.

Corrigé :

1.

a_2	a_1	a_0	M	U
0	0	0	0	1
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

2. On voit donc que l'on a :
   $M = \overline{a_2}a_1a_0 + a_2\overline{a_1}a_0 + a_2a_1\overline{a_0} + a_2a_1a_0 = \overline{a_2}a_1a_0 + a_2(a_1 + a_0) = a_1a_0 + a_2(a_1 + a_0)$
   $U = \overline{a_2}\,\overline{a_1}\,\overline{a_0} + a_2a_1a_0 = \overline{a_2 + a_1 + a_0} + a_2a_1a_0$
3.



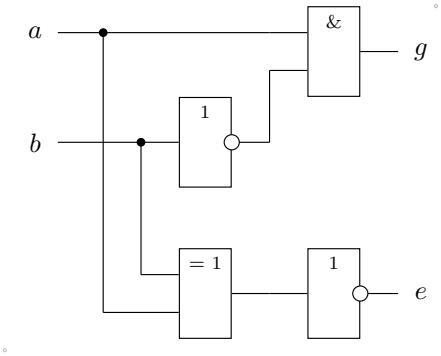
4. $M' = a_2a_1a_0 + M(a_3 + a_4) + a_3a_4(a_2 + a_1 + a_0)$

Exercice 14 *On veut concevoir un circuit permettant de comparer deux nombres* A *et* B *de* 4 *bits (valant de* 0 *à* 15*),* $A = a_3a_2a_1a_0$ *et* $B = b_3b_2b_1b_0$. *Le circuit comporte deux sorties :* G *valant* 1 *si* $A > B$ *et* E *valant* 1 *si* $A = B$.

1. *Soit* a *et* b *deux nombres de* 1 *bit. Soit un circuit à deux sorties :* g *valant* 1 *si* $a > b$ *et* e *valant* 1 *si* $a = b$. *Donner les fonctions logiques* g *et* e*, dessiner le circuit.*
2. *En utilisant des circuits de la question précédente ainsi que des portes logiques OU, ET (à* 2*,* 3 *ou* 4 *entrées) et NON, concevoir un circuit permettant de comparer deux nombres de* 4 *bits. Dessiner le schéma en expliquant votre raisonnement.*

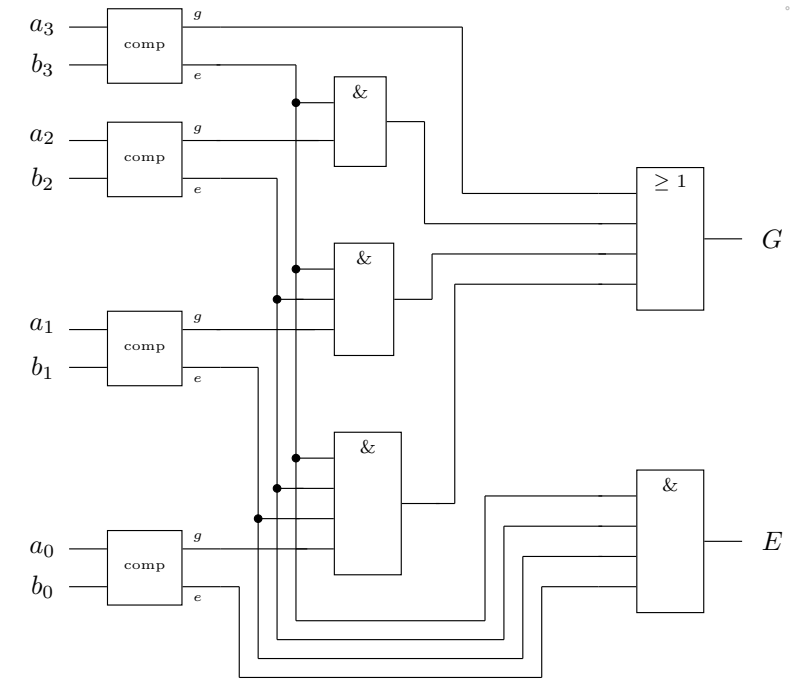
Corrigé :

1. $g = a\overline{b}$ et $e = ab + \overline{a}\,\overline{b} = \overline{a \oplus b}$. Soit le circuit `comp`.



2. $E = 1$ si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 = b_1$ et $a_0 = b_0$.

   $G = 1$ si $a_3 > b_3$
   ou si $a_3 = b_3$ et $a_2 > b_2$
   ou si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 > b_1$
   ou si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 = b_1$ et $a_0 > b_0$

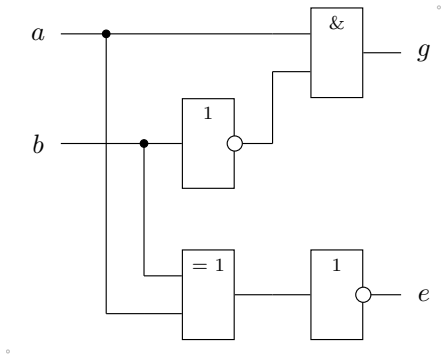


Exercice 15 *On veut concevoir un circuit permettant de comparer deux nombres* **signés** A *et* B *de* 4 *bits (valant de* -8 *à* 7*),* $A = a_3a_2a_1a_0$ *et* $B = b_3b_2b_1b_0$*. Le circuit comporte deux sorties :* G *valant* 1 *si* $A > B$ *et* E *valant* 1 *si* $A = B$*.*

1. *Soit* a *et* b *deux nombres non-signés de* 1 *bit. Soit un circuit à deux sorties :* g *valant* 1 *si* $a > b$ *et* e *valant* 1 *si* $a = b$*. Donner les fonctions logiques* g *et* e*, dessiner le circuit à l'aide de portes OU, ET, NON et XOR.*
2. *En utilisant des circuits de la question précédente ainsi que des portes logiques OU, ET (à* 2*,* 3 *ou* 4 *entrées) et NON, concevoir un circuit permettant de comparer deux nombres signés de* 4 *bits. Dessiner le schéma en expliquant votre raisonnement.*

Corrigé :

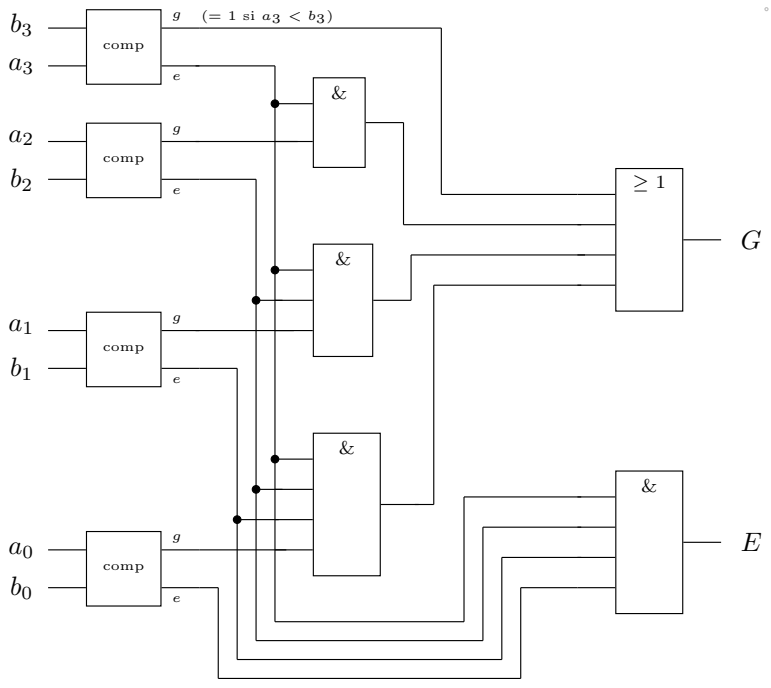
1. $g = a\overline{b}$ et $e = ab + \overline{a}\,\overline{b} = \overline{a \oplus b}$. Soit le circuit `comp`.



2. $E = 1$ si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 = b_1$ et $a_0 = b_0$.

   $G = 1$ si $A \geqslant 0$ et $B < 0$ ($\Longleftrightarrow a_3 = 0$ et $b_3 = 1$)
   ou si $a_3 = b_3$ et $a_2 > b_2$
   ou si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 > b_1$
   ou si $a_3 = b_3$ et $a_2 = b_2$ et $a_1 = b_1$ et $a_0 > b_0$

   C'est donc le même dessin que l'exercice précédent, en inversant les entrées du premier comparateur. La comparaison des trois bits suivants est la même que A et B soient positifs tous les deux ou négatifs tous les deux. C'est une propriété du complément à 2.

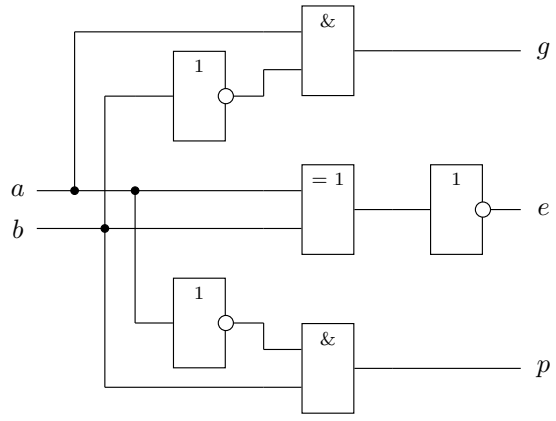


Exercice 16 *On veut concevoir par récurrence un circuit permettant de comparer deux nombres non-signés* A *et* B *de* n *bits,* $A = a_{n-1} \dots a_0$ *et* $B = b_{n-1} \dots b_0$. *Le circuit comporte trois sorties :* G *valant* 1 *si* $A > B$, E *valant* 1 *si* $A = B$ *et* P *valant* 1 *si* $A < B$.

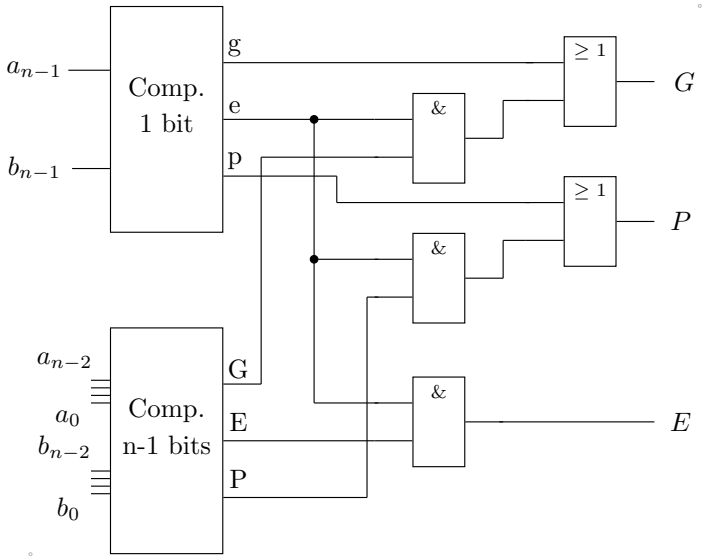
1. *Soit* a *et* b *deux nombres de* 1 *bit. Soit un circuit à trois sorties :* g *valant* 1 *si* $a > b$, e *valant* 1 *si* $a = b$ *et* p *valant* 1 *si* $a < b$. *Donner les fonctions logiques* g, e *et* p, *dessiner le circuit à l'aide de portes OU, ET, NON et XOR (celles que vous voulez).*
2. *En supposant que l'on dispose de portes logiques OU, ET et NON, d'un circuit logique comparant deux nombres non-signés de* $n - 1$ *bits (avec trois sorties G, E et P), ainsi que du circuit de la question précédente, concevoir un circuit permettant de comparer deux nombres non-signés de* n *bits en séparant le travail en deux : comparaison des deux bits de poids fort et comparaison du reste. Dessiner le schéma en expliquant votre raisonnement.*
3. *Toujours à l'aide des circuits précédents, concevoir un circuit permettant de comparer deux nombres* **signés** *de* n *bits.*

Corrigé :

1. $g = a\overline{b}$; $e = ab + \overline{a}\,\overline{b} = \overline{a \oplus b}$; $p = \overline{a}b$



2. $G = a_{n-1} > b_{n-1}$ OU $a_{n-1} = b_{n-1}$ et $(a_{n-2} \dots a_0) > (b_{n-2} \dots b_0)$.
   $P = a_{n-1} < b_{n-1}$ OU $a_{n-1} = b_{n-1}$ et $(a_{n-2} \dots a_0) < (b_{n-2} \dots b_0)$.
   $E = a_{n-1} = b_{n-1}$ et $(a_{n-2} \dots a_0) = (b_{n-2} \dots b_0)$.



3. Il suffit d'intervertir les entrées du premier circuit. En effet :
   $G = a_{n-1} = 0$ et $b_{n-1} = 1$ OU $a_{n-1} = b_{n-1}$ et $(a_{n-2} \dots a_0) > (b_{n-2} \dots b_0)$. D'où :
   $G = a_{n-1} < b_{n-1}$ OU $a_{n-1} = b_{n-1}$ et $(a_{n-2} \dots a_0) > (b_{n-2} \dots b_0)$.
   Et c'est la même chose pour P.

Exercice 17 *On désire construire un comparateur arithmétique binaire de nombres de n bits. C'est-à-dire un circuit à 2 fois* n *entrées* $X = x_{n-1} \dots x_0$ *et* $Y = y_{n-1} \dots y_0$ *représentant deux nombres binaires non-signés sur* n *bits et 2 sorties,* Sup *et* Inf. *On veut que* $\text{Sup} = (X > Y)$ *et* $\text{Inf} = (X < Y)$ *(autrement dit,* Sup *(resp.* Inf*) vaut* 1 *si* $X > Y$ *(resp.* $X < Y$*) et* 0 *sinon).*

On effectue une comparaison bit par bit en commençant par ceux de poids fort.
On pose

$$\text{Sup}_i = (x_{n-1}x_{n-2} \dots x_i > y_{n-1}y_{n-2} \dots y_i)$$

$$\text{Inf}_i = (x_{n-1}x_{n-2} \dots x_i < y_{n-1}y_{n-2} \dots y_i)$$

C'est-a-dire est-ce que la valeur binaire formée par les premiers bits de poids fort de X est plus grande (resp. plus petite) que la valeur binaire formée par les premiers bits de poids fort de Y.

1. *Trouver les formules de récurrence*

$$\text{Sup}_i = f(\text{Sup}_{i+1}, \text{Inf}_{i+1}, x_i, y_i)$$

$$\text{Inf}_i = g(\text{Sup}_{i+1}, \text{Inf}_{i+1}, x_i, y_i)$$

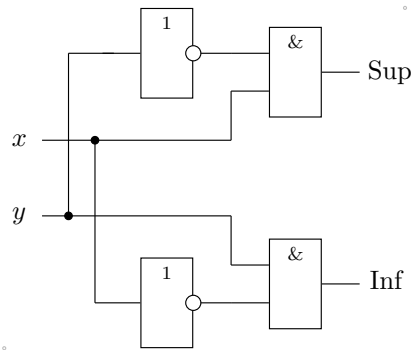
   C'est-à-dire exprimer Sup_i *et* Inf_i *en fonction de* Sup_{i+1}, Inf_{i+1}, x_i *et* y_i.
2. *Donner le schéma d'un comparateur 1 bit. Donner le schéma d'un comparateur* n *bits à partir de comparateur(s)* $n-1$ *bits, de comparateur(s) 1 bit, de porte(s) NON, ET et OU à 2 entrées.*
3. *On suppose que le passage d'un signal électrique dans une porte « coûte » 10 ns. On dit alors qu'un circuit travaille en* p *ns si tous les signaux en sortie sont disponibles en au maximum* p *ns. En combien de temps un comparateur 1 bit et un comparateur* n *bits travaillent-ils ?*

Corrigé : Deux solutions sont possibles :
1.

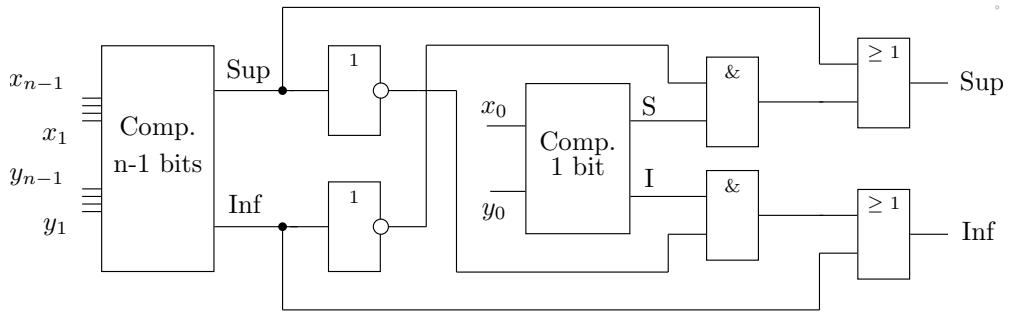
$$\text{Sup}_i = \text{Sup}_{i+1} + \overline{\text{Inf}_{i+1}} x_i \overline{y_i}$$

$$\text{Inf}_i = \text{Inf}_{i+1} + \overline{\text{Sup}_{i+1}}\ \overline{x_i} y_i$$

2. Pour le comparateur 1 bit, on a $Sup = x\overline{y}$ et $Inf = \overline{x}y$.



Le comparateur n bits se construit à partir de la formule :



3. Le comparateur 1 bit travaille en 20 ns, le comparateur 2 bits en 50 ns et le comparateur n bits en $20 + 30(n-1)$ ns.

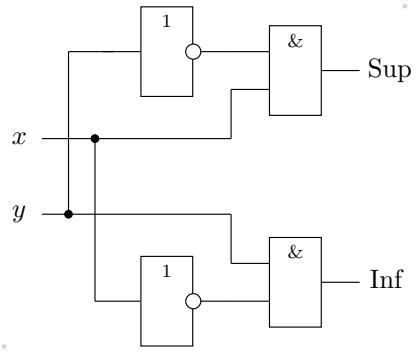
Ou

1.

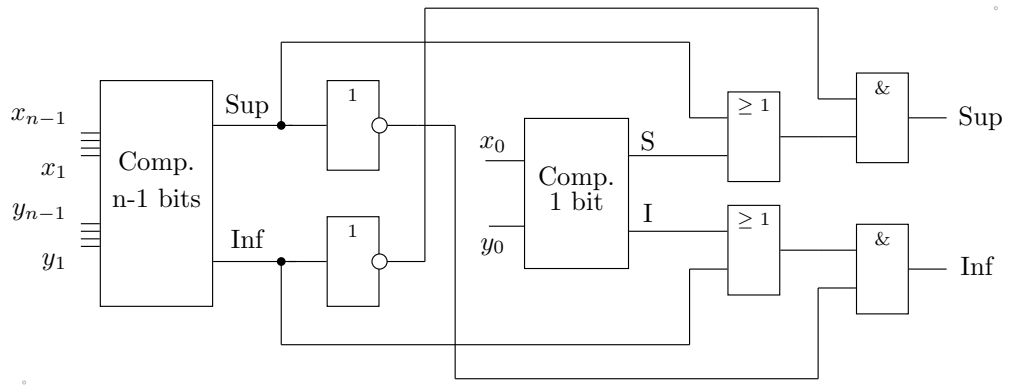
$$\text{Sup}_i = \overline{\text{Inf}_{i+1}}(\text{Sup}_{i+1} + x_i\overline{y_i})$$

$$\text{Inf}_i = \overline{\text{Sup}_{i+1}}(\text{Inf}_{i+1} + \overline{x_i}y_i)$$

2. Pour le comparateur 1 bit, on a $Sup = x\overline{y}$ et $Inf = \overline{x}y$.



Le comparateur n bits se construit à partir de la formule :



3. Le comparateur 1 bit travaille en 20 ns, le comparateur 2 bits en 40 ns et le comparateur n bits en $20 + 20(n-1)$ ns.

5 Exercices Assembleur

Exercice 18 *Dans* `r0` *se trouve l'adresse d'une chaîne de caractères terminée par l'octet nul. Écrire une procédure assembleur qui, lorsqu'elle se termine, permet de récupérer le nombre de caractères de la chaîne dans* `r1`.

Corrigé :

Listing 4. Comptage

```
        MVI     r1,#0           ; compteur = 0
ici:    LDB     r2,(r0)         ; récupérer caractère
        JZ      r2,fin          ; fin chaîne ?
        ADD     r1,r1,#1        ; sinon, compteur ++
        ADD     r0,r0,#1        ; caractère suivant
        JMP     ici             ; retour
fin:
```

Exercice 19

1. *Un tableau d'entiers est stocké en mémoire. Chaque élément a une valeur de 1 à 127 et est donc stocké sur un octet. L'octet qui suit le dernier élément du tableau est nul. L'adresse du premier élément du tableau se trouve dans le registre* `r0`*. Écrire une procédure assembleur qui, lorsqu'elle se termine, permet de récupérer le plus grand élément du tableau dans le registre* `r1`.
2. *Rajouter au programme précédent le calcul de la somme de tous les éléments du tableau dans le registre* `r2`.

Corrigé :

1.

Listing 5. Plus grand

```
        MVI     r1,#0           ; meilleur égal 0
ici:    LDB     r3,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur caractère suivant
        JZ      r3,fin          ; fini ?
        SUB     r31,r3,r1       ; comparer r3 et r1 dans r31
        JLE     r31,ici         ; char plus petit que meilleur ?
        MOV     r1,r3           ; sinon, meilleur = char
        JMP     ici
fin:
```

2.

Listing 6. Somme

```
        MVI     r1,#0           ; meilleur égal 0
        MVI     r2,#0           ; somme égale 0
ici:    LDB     r3,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur caractère suivant
        JZ      r3,fin          ; fini ?
        ADD     r2,r2,r3        ; additionner char à la somme.
        SUB     r31,r3,r1       ; comparer r3 et r1 dans r31
        JLE     r31,ici         ; char plus petit que meilleur ?
        MOV     r1,r3           ; sinon, meilleur = char
        JMP     ici
fin:
```

Exercice 20 *Un tableau d'entiers est stocké en mémoire, chacun étant stocké sur un octet (et représente un nombre en complément à 2) et aucun n'est nul. L'octet qui suit le dernier élément du tableau est nul. L'adresse du premier élément du tableau se trouve dans le registre* `r0`.

Écrire une procédure assembleur qui, lorsqu'elle se termine, permet de récupérer le nombre d'entiers positifs du tableau dans le registre `r1` *et le nombre d'entiers du tableau égaux à 40 (décimal) dans le registre* `r2`.

Corrigé :

Listing 7. Comptage

```
        MVI     r1,#0           ; nombre positifs égal à 0
        MVI     r2,#0           ; nombre égaux à 40 égal 0
ici:    LDB     r3,(r0)         ; charger entier
        ADD     r0,r0,#1        ; pointeur sur entier suivant
        JZ      r3,fin          ; fini ?
        JLT     r3,ici          ; nombre négatif ? si oui, nombre suivant
        ADD     r1,r1,#1        ; si non, incrémenter compteur
        SUB     r31,r3,#40      ; comparer r3 et 40
        JNZ     r31,ici         ; égal à 40 ?, si non ,nombre suivant
        ADD     r2,r2,#1        ; si oui, incrémenter compteur
        JMP     ici
fin:
```

Exercice 21 *Une chaîne de caractères est stockée en mémoire. Chaque élément est stocké sur un octet et aucun n'est nul. L'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre* `r0`. *On souhaite effectuer une opération de remplacement sur certains caractères de la chaîne. Plus précisément, le registre* `r1` *contient le caractère dont on souhaite remplacer chaque occurence dans la chaîne par le caractère se trouvant dans le registre* `r2`.

Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que toutes les occurences (dans la chaîne) du caractère contenu dans `r1` *sont remplacées par le caractère contenu dans* `r2`. *On souhaite également avoir à la fin dans le registre* `r3` *le nombre de remplacements effectués.*

Corrigé :

Listing 8. Remplacements

```
        MVI     r3,#0           ; nombre remplacements égal 0
ici:    LDB     r4,(r0)         ; charger caractère
        JZ      r4,fin          ; fini ?
        SUB     r4,r4,r1        ; comparer r4 et r1
        JNZ     r4,next         ; sauter remplacement si différent
        STB     (r0),r2         ; si oui, remplacer par r2
        ADD     r3,r3,#1        ; nombre de remplacements ++
next:   ADD     r0,r0,#1        ; pointeur char. suivant
        JMP     ici
fin:
```

Exercice 22 *Dans les 16 bits de poids faible de* `r1` *se trouve un nombre entier non-signé ; les 16 autres bits de* `r1` *sont nuls. Dans les 16 bits de poids faible de* `r2` *se trouve un nombre entier non-signé ; les 16 autres bits de* `r2` *sont nuls. Écrire de deux manières différentes une procédure assembleur qui, lorsqu'elle se termine, permet de récupérer dans les 32 bits de* `r0` *le produit des deux nombres (sans bien sûr utiliser directement une instruction de multiplication).*

1. *La suite d'instructions est une simple boucle effectuant autant d'additions que nécessaire.*

2. *La suite d'instructions implémente la procédure classique de multiplication comme suite d'additions des produits partiels.*

Corrigé : Une première version simple.

Listing 9. Multiplication

```
        MVI     r0,#0           ; résultat égal 0
ici:    JZ      r1,fin          ; fini ?
        ADD     r0,r0,r2        ; on additionne une fois
        SUB     r1,r1,#1        ; décrémente multiplicateur
        JMP     ici
fin:
```

Mais on peut surtout changer l'algorithme pour qu'il soit beaucoup plus rapide.

Listing 10. Multiplication v2

```
        MVI     r0,#0           ; résultat égal 0
ici:    LLS     r1,r1,#-1       ; décalage r1 un bit vers la droite
        CCR     r31             ; récupérer le bit C dans r31
        JZ      r31,decal       ; passer au bit suivant si 0
        ADD     r0,r0,r2        ; si non, on additionne
decal:  LLS     r2,r2,#1        ; décalage de r2
        JNZ     r1,ici          ; encore des bits à tester ?
```

Exercice 23 *Une chaîne de caractères est stockée en mémoire. Chaque caractère est stocké sur un octet et l'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre* `r0`.

On souhaite renverser la chaîne, c'est-à-dire l'écrire en commençant par le dernier caractère de la chaîne et stocker cette nouvelle chaîne en mémoire à une adresse contenue dans le registre `r1`. *(Ainsi, si la première chaîne est* `truc`*, on souhaite écrire la chaîne* `curt` *en mémoire avec le premier élément à l'adresse qui se trouve dans* `r1` *et avec l'octet nul final.)*

Écrire une procédure assembleur qui renverse la chaîne pointée par `r0` *et la stocke à l'adresse pointée par* `r1`.

Corrigé :

Listing 11. Renversement

```
        MVI     r2,#0           ; compte le nombre de   caractères
comp:   LDB     r3,(r0)         ; charge le caractère suivant
        JZ      r3,suite        ; fin chaîne ?
        ADD     r2,r2,#1        ; sinon on incrémente
        ADD     r0,r0,#1        ; avancer pointeur
        JMP     comp            ; et on boucle
suite:  SUB     r0,r0,#1        ; reculer pointeur chaîne 1
        JZ      r2,fin          ; on a tout renversé ?
        LDB     r3,(r0)         ; charger caractère
        STB     (r1),r3         ; et le recopier
        ADD     r1,r1,#1        ; avancer pointeur chaîne 2
        SUB     r2,r2,#1        ; décrémenter compteur
        JMP     suite           ; on boucle
fin:    STB     (r1),r2         ; zéro final à stocker (r2 vaut zéro !)
```

Une seule petite subtilité : il ne faut pas oublier le cas où la première chaîne est vide, c'est pourquoi on teste tout de suite le compteur, avant de commencer à recopier.

Exercice 24 *Une chaîne de caractères est stockée en mémoire. Chacun est stocké sur un octet et aucun n'est nul ; l'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre* `r0`*. On souhaite générer des acronymes dans cette chaîne, c'est-à-dire mettre un* `'.'` *après chaque lettre, par exemple en transformant* `IBM` *en* `I.B.M.` *Cette chaîne modifiée se mettra à l'adresse contenue dans* `r1`*.*

Pour indiquer que la transformation est souhaitée, on place un caractère `#`*, pour indiquer qu'elle n'est plus nécessaire une autre* `#` *doit apparaître. Les* `#` *disparaissent de la transformation. On considérera qu'il n'y a pas de blancs ni de caractères bizarroïdes entre les* `#`*. Par exemple* `"je suis chez #IBM# pour toujours"` *deviendra* `"je suis chez I.B.M. pour toujours"` *; la chaîne peut commencer par le caractère* `#` *:* `"#RATP#, #SNCF# : même combat"` *deviendra* `"R.A.T.P., S.N.C.F. : même combat"`*. Il peut y avoir aucun ou plusieurs* `#` *et si c'est le dernier caractère de la chaîne, il pourra ne pas être présent.*

Écrire une procédure assembleur qui, lorsqu'elle se termine, laisse à l'adresse qui était pointée par `r1` *(qui peut avoir changé) la nouvelle chaîne modifiée.*

Corrigé :

Listing 12. Acronymes

```
        MVI     r3,#'.'         ; le caractère à rajouter
debut:  LDB     r2,(r0)         ; récupérer caractère
        JZ      r2,fin          ; fini ?
        SUB     r31,r2,#'#'     ; comparer à '#'
        JZ      r31,start       ; début transformation ?
        STB     (r1),r2         ; copier caractère
        ADD     r0,r0,#1        ; avancer pointeur 0
        ADD     r1,r1,#1        ; avancer pointeur 1
        JMP     debut           ; caractère suivant
start:  ADD     r0,r0,#1        ; avancer pointeur 0
        LDB     r2,(r0)         ; récupérer caractère
        JZ      r2,fin          ; fini ?
        SUB     r31,r2,#'#'     ; comparer à '#'
        JZ      r31,end         ; fin de la transformation ?
        STB     (r1),r2         ; copier caractère
        ADD     r1,r1,#1        ; avancer pointeur 1
        STB     (r1),r3         ; mettre un '.'
        ADD     r1,r1,#1        ; avancer pointeur 1
        JMP     start           ; continuer la recopie
end:    ADD     r0,r0,#1        ; avancer pointeur 0 (sauter le #)
        JMP     debut           ; caractère suivant
fin:    STB     (r1),r2         ; zéro final à stocker (r2 vaut zéro !)
```

Exercice 25 *Une chaîne de caractères est stockée en mémoire. Elle a moins de 128 éléments, chacun est stocké sur un octet et aucun n'est nul ; elle est composée de caractères* `a`*, de caractères* `b` *et d'autres caractères. L'octet qui suit le dernier élément de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre* `r0`*. On souhaite comparer le nombre de caractères* `a` *et le nombre de caractères* `b` *de la chaîne. Plus précisément, on souhaite avoir dans* `r1`*, à la fin de la procédure,* `0` *s'il y a plus de* `a` *que de* `b` *et* `1` *s'il y a autant ou plus de* `b` *que de* `a`*.*

Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que le registre `r1` *contient* `0` *s'il y a plus de* `a` *que de* `b` *et* `1` *sinon.*

Corrigé :

Listing 13. Comptage

```
        MVI     r2,#0           ; nombre de 'a'
        MVI     r3,#0           ; nombre de 'b'
ici:    LDB     r4,(r0)         ; charger caractère
```

```
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        JZ      r4,fin          ; fin chaîne ?
        SUB     r31,r4,#'a'
        JZ      r31,CARa        ; caractère égal 'a' ?
        SUB     r31,r4,#'b'
        JNZ     r31,ici         ; caractère diff. 'b' ?
        ADD     r3,r3,#1        ; nombre de 'b' ++
        JMP     ici             ; caractère suivant
CARa:   ADD     r2,r2,#1        ; nombre de 'a' ++
        JMP     ici             ; caractère suivant
fin:    MVI     r1,#1           ; résultat égal 1
        SUB     r31,r3,r2       ;
        JGE     r31,fin2        ; 'b' >= 'a' ?
        MVI     r1,#0           ; si non, résultat égal 0
fin2:
```

Exercice 26 *On désire faire l'addition de deux nombres entiers positifs stockés sous la forme de deux chaînes de caractères ASCII. Chaque caractère d'une chaîne représente un chiffre de* `0` *à* `9` *stocké sur un octet par son code ASCII de valeur décimale* 48 *(pour* `0`*) à* 57 *(pour* `9`*). Les deux chaînes comportent le même nombre de caractères et se terminent chacune par un octet nul. L'adresse du premier caractère de la première chaîne se trouve dans le registre* `r0`*. L'adresse du premier caractère de la deuxième chaîne se trouve dans le registre* `r1`*.*

On désire stocker l'addition des deux nombres sous la forme d'une chaîne de caractères ASCII en mémoire dont le premier caractère sera mis à l'adresse contenue dans le registre `r2` *et qui sera terminée par un octet nul. Les deux chaînes sont stockées à l'envers de leur affichage : le premier caractère représente le chiffre le moins significatif et le dernier le plus significatif. On peut donc effectuer l'addition en parcourant les chaînes du premier au dernier caractère.*

Écrire une procédure assembleur qui, lorsqu'elle se termine, laisse en mémoire à l'adresse initialement pointée par `r2` *la chaîne de caractères ASCII correspondant à la somme des deux nombres initiaux.*

Corrigé :

Listing 14. Addition ASCII

```
        MVI     r3,#0           ; retenue égale 0
ici:    LDB     r4,(r0)         ; charger les deux caractères
        LDB     r5,(r1)
        ADD     r0,r0,#1        ; et pointer sur les
        ADD     r1,r1,#1        ; caractères suivants
        JZ      r4,fin          ; fin chaîne ?
        ADD     r6,r4,r5        ; additionner les 2 caractères
        SUB     r6,r6,#'0'      ; revenir à un caractère ascii
        ADD     r6,r6,r3        ; rajouter la précédente retenue
        MVI     r3,#0           ; retenue à 0
        SUB     r31,r6,#'9'     ; le caractère est-il
        JLE     r31,inf         ; plus petit ou égal à 9 ?
        SUB     r6,r6,#10       ; si oui, retirer 10
        MVI     r3,#1           ; et mettre une retenue
inf:    STB     (r2),r6         ; stocker somme
        ADD     r2,r2,#1        ; avancer pointeur
        JMP     ici
fin:    JZ      r3,fin2         ; retenue finale ?
        MVI     r7,#'1'         ; si oui mettre un 1
        STB     (r2),r7
        ADD     r2,r2,#1        ; et avancer pointeur
fin2:   MVI     r7,#0           ; mettre le 0 final
        STB     (r2),r7
```

Exercice 27 *Une chaîne de caractères est stockée en mémoire. Chaque caractère est stocké sur un octet et est soit un caractère 'x' soit un caractère 'y'. L'octet qui suit le dernier caractère de la chaîne est nul. L'adresse du premier élément de la chaîne se trouve dans le registre* `r0`.

On souhaite comparer le nombre de groupes de caractères `x` *et le nombre de groupes de caractères* `y` *de la chaîne (un groupe de caractères* `x` *[resp.* `y`*] est simplement un ou plusieurs caractères* `x` *[resp.* `y`*] qui se suivent). Plus précisément, on souhaite avoir dans* `r1`*, à la fin de la procédure,* `0` *s'il y a plus de groupes de* `x` *que de groupes de* `y` *et* `1` *s'il y a autant ou plus de groupes de* `y` *que de groupes de* `x`.

Écrire une procédure assembleur qui, lorsqu'elle se termine, assure que le registre `r1` *contient* `0` *s'il y a strictement plus de groupes de* `x` *que de groupes de* `y` *et* `1` *sinon.*

Corrigé :

Le premier algorithme auquel on peut penser est simplement de parcourir la chaîne en bouclant sur chaque groupe et en incrémentant un compteur dans un registre à la fin de chaque groupe. Quand la chaîne est terminée *(ne pas oublier de vérifier à chaque caractère chargé)*, on compare le nombre de groupes de `x` au nombre de groupes de `y`. Cela donne le code suivant :

Listing 15. Comparaison de caractères

```
        MVI     r2,#0           ; nombre de groupeX
        MVI     r3,#0           ; nombre de groupeY
        LDB     r4,(r0)         ; charger premier caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        JZ      r4,fin          ; fin chaîne ?
        SUB     r31,r4,#'y'     ; caractère égal à 'y' ?
        JZ      r31,CARy        ; sinon c'est un 'x'
CARx:   LDB     r4,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        SUB     r31,r4,#'x'     ; caractère égal à 'x' ?
        JZ      r31,CARx        ; si oui reboucler
        ADD     r2,r2,#1        ; sinon on a fini un groupeX
        JZ      r4,fin          ; fin chaîne ?
CARy:   LDB     r4,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        SUB     r31,r4,#'y'     ; caractère égal à 'y' ?
        JZ      r31,CARy        ; si oui reboucler
        ADD     r3,r3,#1        ; sinon on a fini un groupeY
        SUB     r31,r4,#'x'     ; caractère égal à 'x' ?
        JZ      r31,CARx        ; sinon c'est une fin de chaîne
fin:    MVI     r1,#1           ; résultat égale 1
        SUB     r31,r3,r2       ; comparer les deux nombres
        JGE     r31,fin2        ; groupeY >= groupeX ?
        MVI     r1,#0           ; si non, résultat égal 0
fin2:
```

Mais une analyse un peu plus fine permet de s'apercevoir qu'il y a au maximum une différence de 1 entre le nombre de groupes de `x` et de `y`. En effet, *chaque groupe de l'un est forcément suivi d'un groupe de l'autre, sauf éventuellement le dernier*. Plus précisément :

- si le premier caractère est un 'y', il y a autant de groupes de `y` que de groupes de `x` (si le dernier caractère est un 'x') ou un de plus (si le dernier caractère est un 'y') ; dans ces deux cas, il faut renvoyer 1 ;
- si le premier caractère est un 'x', il y a autant de groupes de `x` que de groupes de `y` (si le dernier caractère est un 'y') ou un de plus (si le dernier caractère est un 'x') ; dans le premier cas on renvoie 1, et 0 dans le deuxième.

D'où le code suivant :

Listing 16. Comparaison de caractères v2

```
        MVI     r1,#1           ; le cas par défaut
        LDB     r2,(r0)         ; charger premier caractère
        JZ      r2,fin          ; fin chaîne ?
        SUB     r31,r2,#'y'     ; premier caractère égal à 'y' ?
        JZ      r31,fin         ; si oui, c'est fini
ici:    MOV     r3,r2           ; sauvegarder le dernier caractère dans r3
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        LDB     r2,(r0)         ; charger caractère
        JNZ     r2,ici          ; ce n'est pas la fin de la chaîne ?
        SUB     r31,r3,#'y'     ; le dernier caractère était un 'y' ?
        JZ      r31,fin         ; sinon, c'est le dernier cas,
        MVI     r1,#0           ; 'x' en tête et fin
fin:
```

*Si on veut éliminer cette deuxième solution plus simple, on peut introduire d'autres caractères que '*x*' et '*y*' dans la chaîne ; cela force alors à la parcourir en comptant les 2 groupes avec le code suivant :*

Listing 17. Comparaison de caractères v3

```
        MVI     r2,#0           ; nombre de groupeX
        MVI     r3,#0           ; nombre de groupeY
ici:    LDB     r4,(r0)         ; charger premier caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        JZ      r4,fin          ; fin chaîne ?
        SUB     r31,r4,#'y'     ; caractère égal à 'y' ?
        JZ      r31,CARy        ; si oui, aller en CARy
        SUB     r31,r4,#'x'     ; caractère égal à 'x' ?
        JNZ     r31,ici         ; sinon reboucler
CARx:   LDB     r4,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        SUB     r31,r4,#'x'     ; est-ce toujours un 'x' ?
        JZ      r31,CARx        ; si oui reboucler
        ADD     r2,r2,#1        ; sinon on a fini un groupeX
        JZ      r4,fin          ; fin chaîne ?
        SUB     r31,r4,#'y'     ; caractère égal à 'y' ?
        JNZ     r31,ici         ; sinon reboucler
CARy:   LDB     r4,(r0)         ; charger caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        SUB     r31,r4,#'y'     ; est-ce toujours un 'y' ?
        JZ      r31,CARy        ; si oui reboucler
        ADD     r3,r3,#1        ; sinon on a fini un groupeY
        SUB     r31,r4,#'x'     ; est-ce un 'x' ?
        JZ      r31,CARx        ; si oui aller en CARx
        JNZ     r4,ici          ; on reboucle si ce n'est pas un fin de chaîne
fin:    MVI     r1,#1           ; résultat égale 1
        SUB     r31,r3,r2       ; comparer les deux nombres
        JGE     r31,fin2        ; groupeY >= groupeX ?
        MVI     r1,#0           ; si non, résultat égal 0
fin2:
```

Mais il suffit en fait de tester les changements d'un caractère par rapport au caractère précédent pour détecter les groupes :

- *si le caractère est identique au précédent, on passe au suivant ;*
- *sinon, s'il est égal à '*x*', c'est le début d'un groupe de '*x*' ;*
- *sinon, s'il est égal à '*y*', c'est le début d'un groupe de '*y*' ;*

– *on passe au caractère suivant.*

Listing 18. Comparaison de caractères v4

```
        MVI     r2,#0           ; nombre de groupeX
        MVI     r3,#0           ; nombre de groupeY
        MVI     r4,#0           ; caractère précédent vide
ici:    MOV     r5,r4           ; sauvegarde caractère précédent
        LDB     r4,(r0)         ; charger premier caractère
        ADD     r0,r0,#1        ; pointeur sur caractère suivant
        JZ      r4,fin          ; fin chaîne ?
        SUB     r31,r5,r4       ; identique au précédent ?
        JZ      r31,ici         ; si oui passer au prochain
        SUB     r31,r4,#'x'     ; est-il différent de 'x' ?
        JNZ     r31,testY       ; si oui tester pour 'y'
        ADD     r2,r2,#1        ; on commence un groupeX
        JMP     ici             ; on boucle sur le caractère suivant
testY:  SUB     r31,r4,#'y'     ; comparer à 'y'
        JNZ     r31,ici         ; sinon reboucler
        ADD     r3,r3,#1        ; on commence un groupeY
        JMP     ici             ; on boucle sur le caractère suivant
fin:    MVI     r1,#1           ; résultat égale 1
        SUB     r31,r3,r2       ; comparer les deux nombres
        JGE     r31,fin2        ; groupeY >= groupeX ?
        MVI     r1,#0           ; si non, résultat égal 0
fin2:
```

6 Exercices mémoire cache

Exercice 28 *Un programme se compose d'une boucle de 36 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 8, 77 à 84, 9 à 16, 77 à 84, 17 à 20. Ce programme doit tourner sur une machine possédant un cache d'une taille de 16 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C. Le cache est associatif (un bloc mémoire peut venir dans n'importe quel bloc du cache) et la stratégie de remplacement utilisée est LRU (on remplace le bloc le moins récemment utilisé).*

1. *Le cache possède 4 blocs de 4 instructions : les blocs que l'on peut transférer sont 1-4, 5-8, ..., 73-76, 77-80, 81-84... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
2. *Le cache possède 2 blocs de 8 instructions : les blocs que l'on peut transférer sont 1-8, 9-16, ..., 73-80, 81-88... Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
3. *Refaire les deux questions précédentes en supposant que les instructions sont aux adresses mémoire 1 à 8, 75 à 82, 9 à 16, 75 à 82, 17 à 20. Le cache est vide au départ.*

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1.

$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 2	$1 \to 4$	$M + 3C$	bloc 3
$5 \to 8$	$M + 3C$	bloc 2	$5 \to 8$	$M + 3C$	bloc 3	$5 \to 8$	$M + 3C$	bloc 4
$77 \to 80$	$M + 3C$	bloc 3	$77 \to 80$	$M + 3C$	bloc 4	$77 \to 80$	$M + 3C$	bloc 1
$81 \to 84$	$M + 3C$	bloc 4	$81 \to 84$	$M + 3C$	bloc 1	$81 \to 84$	$M + 3C$	bloc 2
$9 \to 12$	$M + 3C$	bloc 1	$9 \to 12$	$M + 3C$	bloc 2	$9 \to 12$	$M + 3C$	bloc 3
$13 \to 16$	$M + 3C$	bloc 2	$13 \to 16$	$M + 3C$	bloc 3	$13 \to 16$	$M + 3C$	bloc 4
$77 \to 80$	$4C$	bloc 3	$77 \to 80$	$4C$	bloc 4	$77 \to 80$	$4C$	bloc 1
$81 \to 84$	$4C$	bloc 4	$81 \to 84$	$4C$	bloc 1	$81 \to 84$	$4C$	bloc 2
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 2	$17 \to 20$	$M + 3C$	bloc 3

Soit un total de $3 \times (7M + 29C) = 21M + 87C$.

2.

$1 \to 8$	$M + 7C$	bloc 1	$1 \to 8$	$M + 7C$	bloc 2	$1 \to 8$	$M + 7C$	bloc 1
$77 \to 80$	$M + 3C$	bloc 2	$77 \to 80$	$M + 3C$	bloc 1	$77 \to 80$	$M + 3C$	bloc 2
$81 \to 84$	$M + 3C$	bloc 1	$81 \to 84$	$M + 3C$	bloc 2	$81 \to 84$	$M + 3C$	bloc 1
$9 \to 16$	$M + 7C$	bloc 2	$9 \to 16$	$M + 7C$	bloc 1	$9 \to 16$	$M + 7C$	bloc 2
$77 \to 80$	$M + 3C$	bloc 1	$77 \to 80$	$M + 3C$	bloc 2	$77 \to 80$	$M + 3C$	bloc 1
$81 \to 84$	$M + 3C$	bloc 2	$81 \to 84$	$M + 3C$	bloc 1	$81 \to 84$	$M + 3C$	bloc 2
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 2	$17 \to 20$	$M + 3C$	bloc 1

Soit un total de $3 \times (7M + 29C) = 21M + 87C$.

3.

$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 4	$1 \to 4$	$M + 3C$	bloc 3
$5 \to 8$	$M + 3C$	bloc 2	$5 \to 8$	$M + 3C$	bloc 1	$5 \to 8$	$M + 3C$	bloc 4
$75 \to 76$	$M + C$	bloc 3	$75 \to 76$	$M + C$	bloc 2	$75 \to 76$	$M + C$	bloc 1
$77 \to 80$	$M + 3C$	bloc 4	$77 \to 80$	$M + 3C$	bloc 3	$77 \to 80$	$M + 3C$	bloc 2
$81 \to 82$	$M + C$	bloc 1	$81 \to 82$	$M + C$	bloc 4	$81 \to 82$	$M + C$	bloc 3
$9 \to 12$	$M + 3C$	bloc 2	$9 \to 12$	$M + 3C$	bloc 1	$9 \to 12$	$M + 3C$	bloc 4
$13 \to 16$	$M + 3C$	bloc 3	$13 \to 16$	$M + 3C$	bloc 2	$13 \to 16$	$M + 3C$	bloc 1
$75 \to 76$	$M + C$	bloc 4	$75 \to 76$	$M + C$	bloc 3	$75 \to 76$	$M + C$	bloc 2
$77 \to 80$	$M + 3C$	bloc 1	$77 \to 80$	$M + 3C$	bloc 4	$77 \to 80$	$M + 3C$	bloc 3
$81 \to 82$	$M + C$	bloc 2	$81 \to 82$	$M + C$	bloc 1	$81 \to 82$	$M + C$	bloc 4
$17 \to 20$	$M + 3C$	bloc 3	$17 \to 20$	$M + 3C$	bloc 2	$17 \to 20$	$M + 3C$	bloc 1

Soit un total de $3 \times (11M + 25C) = 33M + 75C$.

$1 \to 8$	$M + 7C$	bloc 1	$1 \to 8$	$M + 7C$	bloc 2	$1 \to 8$	$M + 7C$	bloc 1
$75 \to 80$	$M + 5C$	bloc 2	$75 \to 80$	$M + 5C$	bloc 1	$75 \to 80$	$M + 5C$	bloc 2
$81 \to 82$	$M + C$	bloc 1	$81 \to 82$	$M + C$	bloc 2	$81 \to 82$	$M + C$	bloc 1
$9 \to 16$	$M + 7C$	bloc 2	$9 \to 16$	$M + 7C$	bloc 1	$9 \to 16$	$M + 7C$	bloc 2
$75 \to 80$	$M + 5C$	bloc 1	$75 \to 80$	$M + 5C$	bloc 2	$75 \to 80$	$M + 5C$	bloc 1
$81 \to 82$	$M + C$	bloc 2	$81 \to 82$	$M + C$	bloc 1	$81 \to 82$	$M + C$	bloc 2
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 2	$17 \to 20$	$M + 3C$	bloc 1

Soit un total de $3 \times (7M + 29C) = 21M + 87C$.

Exercice 29 *Un programme se compose d'une boucle de 20 instructions à exécuter 4 fois ; cette boucle se trouve aux adresses mémoire 1 à 20. Ce programme doit tourner sur une machine possédant un cache d'une taille de 16 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.*

1. *Le cache est à correspondance directe, formé de 8 blocs de 2 instructions. On rappelle que cela veut dire que les mots mémoire 1 et 2, 17 et 18... vont dans le premier bloc, que les mots 3 et 4, 19 et 20... vont dans le deuxième, etc. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ?*
2. *Le cache est maintenant associatif (un bloc mémoire peut venir dans n'importe quel bloc du cache) et la stratégie de remplacement utilisée est LRU (on remplace le bloc le moins récemment utilisé). Quel est le temps d'exécution du programme ?*
3. *Refaire les deux premières questions en prenant un cache composé de 4 blocs de 4 mots.*

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1. Les deux premiers blocs sont partagés ($1 - 2$ et $17 - 18$, $3 - 4$ et $19 - 20$), donc :

$$T = 8(M + C) + 2(M + C) + 3[4(M + C) + 12C] = 22M + 58C$$

2. On ne réutilise jamais un bloc, donc :

$$T = 4[10(M + C)] = 40M + 40C$$

3. (a) Le premier bloc est partagé ($1 \to 4$ et $17 \to 20$), donc :

$$T = 4(M + 3C) + (M + 3C) + 3[2(M + 3C) + 12C] = 11M + 69C$$

(b) On ne réutilise jamais un bloc, donc :

$$T = 4[5(M + 3C)] = 20M + 60C$$

Exercice 30 *Un programme se compose d'une boucle de 16 instructions consécutives à exécuter 5 fois. Ce programme doit tourner sur une machine possédant un cache d'une taille de 16 instructions. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C. Le cache est associatif (un bloc mémoire peut venir dans n'importe quel bloc du cache) et la stratégie de remplacement utilisée est LRU (on remplace le bloc le moins récemment utilisé).*

1. *Le cache possède 4 blocs de 4 instructions : les blocs mémoire que l'on peut transférer dans le cache sont 1-4, 5-8, 9-12, 13-16, 17-20...*
   (a) *Le programme se place aux adresses 1 à 16 en mémoire. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ?*
   (b) *Le programme se place aux adresses 3 à 18 en mémoire. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? A-t-on le même résultat si le programme se place aux adresses 2 à 17 ou 4 à 19 ?*
2. *Le cache possède 1 bloc de 16 instructions : les blocs mémoire que l'on peut transférer dans le cache sont 1-16, 17-32... Reprendre les questions ci-dessus.*
3. *Sachant que l'adresse de placement du programme est tirée aléatoirement, pouvez-vous calculer l'efficacité du cache ?*

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1. (a)

$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$4C$	bloc 1	$1 \to 4$	$4C$	bloc 1
$5 \to 8$	$M + 3C$	bloc 2	$5 \to 8$	$4C$	bloc 2	$5 \to 8$	$4C$	bloc 2
$9 \to 12$	$M + 3C$	bloc 3	$9 \to 12$	$4C$	bloc 3	$9 \to 12$	$4C$	bloc 3
$13 \to 16$	$M + 3C$	bloc 4	$13 \to 16$	$4C$	bloc 4	$13 \to 16$	$4C$	bloc 4
$1 \to 4$	$4C$	bloc 1	$1 \to 4$	$4C$	bloc 1			
$5 \to 8$	$4C$	bloc 2	$5 \to 8$	$4C$	bloc 2			
$9 \to 12$	$4C$	bloc 3	$9 \to 12$	$4C$	bloc 3			
$13 \to 16$	$4C$	bloc 4	$13 \to 16$	$4C$	bloc 4			

Soit un total de $4M + 76C$.

(b)

$3 \to 4$	$M + C$	bloc 1	$3 \to 4$	$M + C$	bloc 3	$3 \to 4$	$M + C$	bloc 1
$5 \to 8$	$M + 3C$	bloc 2	$5 \to 8$	$M + 3C$	bloc 4	$5 \to 8$	$M + 3C$	bloc 2
$9 \to 12$	$M + 3C$	bloc 3	$9 \to 12$	$M + 3C$	bloc 1	$9 \to 12$	$M + 3C$	bloc 3
$9 \to 12$	$M + 3C$	bloc 4	$9 \to 12$	$M + 3C$	bloc 2	$9 \to 12$	$M + 3C$	bloc 4
$17 \to 18$	$M + C$	bloc 1	$17 \to 18$	$M + C$	bloc 3	$17 \to 18$	$M + C$	bloc 1
$3 \to 4$	$M + C$	bloc 2	$3 \to 4$	$M + C$	bloc 4			
$5 \to 8$	$M + 3C$	bloc 3	$5 \to 8$	$M + 3C$	bloc 1			
$9 \to 12$	$M + 3C$	bloc 4	$9 \to 12$	$M + 3C$	bloc 2			
$9 \to 12$	$M + 3C$	bloc 1	$9 \to 12$	$M + 3C$	bloc 3			
$17 \to 18$	$M + C$	bloc 2	$17 \to 18$	$M + C$	bloc 4			

Soit un total de $5 \times (5M + 11C) = 25M + 55C$. Le résultat est le même si le programme se place aux adresses 2 à 17 ou 4 à 19.

2. (a)

$1 \to 16$	$\mathrm{M} + 15\mathrm{C}$	bloc 1
$1 \to 16$	$16\mathrm{C}$	bloc 1
$1 \to 16$	$16\mathrm{C}$	bloc 1
$1 \to 16$	$16\mathrm{C}$	bloc 1
$1 \to 16$	$16\mathrm{C}$	bloc 1

Soit un total de $\mathrm{M} + 79\mathrm{C}$.

(b)

$3 \to 16$	$\mathrm{M} + 13\mathrm{C}$	bloc 1	$3 \to 16$	$\mathrm{M} + 13\mathrm{C}$	bloc 1	$3 \to 16$	$\mathrm{M} + 13\mathrm{C}$	bloc 1
$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1	$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1	$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1
$3 \to 16$	$\mathrm{M} + 13\mathrm{C}$	bloc 1	$3 \to 16$	$\mathrm{M} + 13\mathrm{C}$	bloc 1			
$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1	$17 \to 18$	$\mathrm{M} + \mathrm{C}$	bloc 1			

Soit un total de $5 \times (2\mathrm{M} + 14\mathrm{C}) = 10\mathrm{M} + 70\mathrm{C}$. Le résultat est le même si le programme se place aux adresses 2 à 17 ou 4 à 19.

3. Le taux moyen de défaut de cache est de

$$\frac{1}{4} \times \frac{4}{80} + \frac{3}{4} \times \frac{25}{80} \approx 0{,}25$$

dans le premier cas et de

$$\frac{1}{16} \times \frac{1}{80} + \frac{15}{16} \times \frac{10}{80} \approx 0{,}11$$

dans le deuxième.

Exercice 31 *Notre machine possède un cache d'une taille de 16 instructions regroupées en 4 blocs de 4 instructions. Le cache est à correspondance directe. On rappelle que cela veut dire que les mots mémoire 1-4, 17-20... vont dans le premier bloc du cache, que les mots 5-8, 21-24... vont dans le deuxième, etc. Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.*

1. *Un programme se compose d'une boucle de 40 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 20, 21 à 24, 1 à 4, 21 à 24, 1 à 4, 21 à 24. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
2. *Le compilateur a réussi à optimiser votre programme et en a réduit la taille en gagnant quelques intructions. Il se compose d'une boucle de 36 instructions à exécuter 3 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 16, 17 à 20, 1 à 4, 17 à 20, 1 à 4, 17 à 20. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
3. *Que concluez-vous des deux précédents résultats ?*

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1.

$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$1 \to 4$	$4\mathrm{C}$	bloc 1	$1 \to 4$	$4\mathrm{C}$	bloc 1
$5 \to 8$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$5 \to 8$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$5 \to 8$	$\mathrm{M} + 3\mathrm{C}$	bloc 2
$9 \to 12$	$\mathrm{M} + 3\mathrm{C}$	bloc 3	$9 \to 12$	$4\mathrm{C}$	bloc 3	$9 \to 12$	$4\mathrm{C}$	bloc 3
$13 \to 16$	$\mathrm{M} + 3\mathrm{C}$	bloc 4	$13 \to 16$	$4\mathrm{C}$	bloc 4	$13 \to 16$	$4\mathrm{C}$	bloc 4
$17 \to 20$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$17 \to 20$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$17 \to 20$	$\mathrm{M} + 3\mathrm{C}$	bloc 1
$21 \to 24$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$21 \to 24$	$\mathrm{M} + 3\mathrm{C}$	bloc 2	$21 \to 24$	$\mathrm{M} + 3\mathrm{C}$	bloc 2
$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 1	$1 \to 4$	$\mathrm{M} + 3\mathrm{C}$	bloc 1
$21 \to 24$	$4\mathrm{C}$	bloc 2	$21 \to 24$	$4\mathrm{C}$	bloc 2	$21 \to 24$	$4\mathrm{C}$	bloc 2
$1 \to 4$	$4\mathrm{C}$	bloc 1	$1 \to 4$	$4\mathrm{C}$	bloc 1	$1 \to 4$	$4\mathrm{C}$	bloc 1
$21 \to 24$	$4\mathrm{C}$	bloc 2	$21 \to 24$	$4\mathrm{C}$	bloc 2	$21 \to 24$	$4\mathrm{C}$	bloc 2

Soit un total de $15\mathrm{M} + 105\mathrm{C}$.

2.

$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1
$5 \to 8$	$M + 3C$	bloc 2	$5 \to 8$	$4C$	bloc 2	$5 \to 8$	$4C$	bloc 2
$9 \to 12$	$M + 3C$	bloc 3	$9 \to 12$	$4C$	bloc 3	$9 \to 12$	$4C$	bloc 3
$13 \to 16$	$M + 3C$	bloc 4	$13 \to 16$	$4C$	bloc 4	$13 \to 16$	$4C$	bloc 4
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1
$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1
$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1	$1 \to 4$	$M + 3C$	bloc 1
$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1	$17 \to 20$	$M + 3C$	bloc 1

Soit un total de $21M + 87C$.

3. On voit que « l'amélioration » effectuée par le compilateur a augmenté le nombre d'accès mémoire par une mauvaise répartition des instructions dans les blocs même si on a gagné en nombre d'instructions. On a $6M$ en plus et $18C$ en moins, ce qui fait que si $M > 3C$, le programme « optimisé » s'exécute plus lentement !

Exercice 32 *Notre machine possède un cache d'une taille de 24 instructions regroupées en 6 blocs de 4 instructions. Le cache est associatif (un bloc mémoire peut venir dans n'importe quel bloc du cache) et la stratégie de remplacement utilisée est LRU (on remplace le bloc le moins récemment utilisé). Le temps de cycle de la mémoire principale est M et le temps de cycle du cache est C.*

1. *Un programme se compose d'une boucle de 28 instructions à exécuter 5 fois ; les instructions se trouvant, dans l'ordre, aux adresses mémoire 1 à 28. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
2. *Vous avez réorganisé votre code et votre boucle est maintenant découpée en deux : on exécute d'abord 5 fois les instructions 1 à 14* puis *5 fois les instructions 15 à 28. Quel est le temps total d'exécution du programme en ne tenant pas compte des temps de calcul ? Le cache est vide au départ.*
3. *Que concluez-vous des deux précédents résultats ?*
4. *Refaire uniquement la première question en supposant que le cache est à correspondance directe. On rappelle que cela veut dire que les mots mémoire 1-4, 25-28... vont dans le premier bloc du cache, que les mots 5-8, 29-32... vont dans le deuxième, etc.*

Rappel : Lorsque l'on va chercher un mot en mémoire, pendant M on ramène le mot pour le processeur et tout le bloc correspondant dans le cache.

Corrigé :

1.

$1 \to 4$	$M + 3C$	(1)	$M + 3C$	(2)	$M + 3C$	(3)	$M + 3C$	(4)	$M + 3C$	(5)
$5 \to 8$	$M + 3C$	(2)	$M + 3C$	(3)	$M + 3C$	(4)	$M + 3C$	(5)	$M + 3C$	(6)
$9 \to 12$	$M + 3C$	(3)	$M + 3C$	(4)	$M + 3C$	(5)	$M + 3C$	(6)	$M + 3C$	(1)
$13 \to 16$	$M + 3C$	(4)	$M + 3C$	(5)	$M + 3C$	(6)	$M + 3C$	(1)	$M + 3C$	(2)
$17 \to 20$	$M + 3C$	(5)	$M + 3C$	(6)	$M + 3C$	(1)	$M + 3C$	(2)	$M + 3C$	(3)
$21 \to 24$	$M + 3C$	(6)	$M + 3C$	(1)	$M + 3C$	(2)	$M + 3C$	(3)	$M + 3C$	(4)
$25 \to 28$	$M + 3C$	(1)	$M + 3C$	(2)	$M + 3C$	(3)	$M + 3C$	(4)	$M + 3C$	(5)

Soit un total de $35M + 105C$.

2.

$1 \to 4$	$M + 3C$	(1)	$4C$	(1)	$4C$	(1)	$4C$	(1)	$4C$	(1)
$5 \to 8$	$M + 3C$	(2)	$4C$	(2)	$4C$	(2)	$4C$	(2)	$4C$	(2)
$9 \to 12$	$M + 3C$	(3)	$4C$	(3)	$4C$	(3)	$4C$	(3)	$4C$	(3)
$13 \to 14$	$M + C$	(4)	$2C$	(4)	$2C$	(4)	$2C$	(4)	$2C$	(4)
$15 \to 16$	$2C$	(4)	$2C$	(4)	$2C$	(4)	$2C$	(4)	$2C$	(4)
$17 \to 20$	$M + 3C$	(5)	$4C$	(5)	$4C$	(5)	$4C$	(5)	$4C$	(5)
$21 \to 24$	$M + 3C$	(6)	$4C$	(6)	$4C$	(6)	$4C$	(6)	$4C$	(6)
$25 \to 28$	$M + 3C$	(1)	$4C$	(1)	$4C$	(1)	$4C$	(1)	$4C$	(1)

Soit un total de $7M + 133C$.

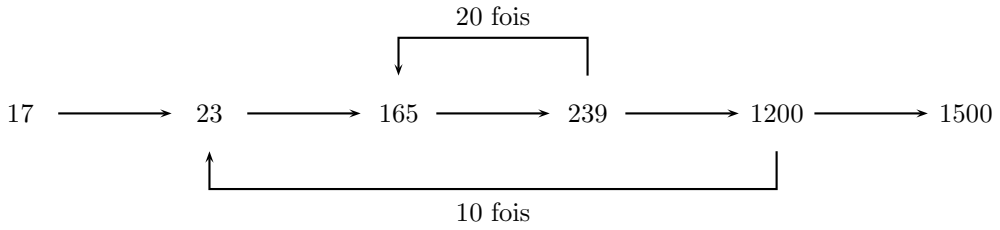
3. On voit que l'écriture du programme a une influence sur la performance du cache. Si une boucle tient entièrement dans le cache, celui-ci est beaucoup plus efficace.

4.

$1 \rightarrow 4$	$M+3C$	(1)	$M+3C$	(1)	$M+3C$	(1)	$M+3C$	(4)	$M+3C$	(4)
$5 \rightarrow 8$	$M+3C$	(2)	$4C$	(2)	$4C$	(2)	$4C$	(5)	$4C$	(5)
$9 \rightarrow 12$	$M+3C$	(3)	$4C$	(3)	$4C$	(3)	$4C$	(6)	$4C$	(6)
$13 \rightarrow 16$	$M+3C$	(4)	$4C$	(4)	$4C$	(4)	$4C$	(1)	$4C$	(1)
$17 \rightarrow 20$	$M+3C$	(5)	$4C$	(5)	$4C$	(5)	$4C$	(2)	$4C$	(2)
$21 \rightarrow 24$	$M+3C$	(6)	$4C$	(6)	$4C$	(6)	$4C$	(3)	$4C$	(3)
$25 \rightarrow 28$	$M+3C$	(1)	$M+3C$	(1)	$M+3C$	(1)	$M+3C$	(4)	$M+3C$	(4)

Soit un total de $15M + 125C$.

Exercice 33 *Un programme se compose de deux boucles* `for` *imbriquées : une petite boucle interne et une plus grande boucle externe. La structure générale du programme est donnée dans la figure suivante.*



Les adresses décimales données déterminent l'emplacement des deux boucles ainsi que le début et la fin du programme. Tous les emplacements mémoire des différents blocs, 17-22, 23-164, 165-239, etc., contiennent des instructions devant être exécutées séquentiellement. Le programme doit s'exécuter sur une machine possédant un cache. Le cache est organisé sous la forme de correspondance directe et les paramètres en sont :

- *taille de la mémoire principale : 64K mots ;*
- *taille du cache : 1K (soit 1 024) mots ;*
- *taille d'un bloc : 128 mots.*

Le temps de cycle de la mémoire principale est 100 ns et le temps de cycle du cache est 10 ns. Une instruction occupe un mot en mémoire.

En ne tenant pas compte des temps de recherche et de rangement d'un opérande ou d'un résultat associés aux opérations de lecture et d'écriture, calculer le temps total nécessaire à la recherche des instructions dans le programme. Estimer le facteur d'amélioration résultant de l'utilisation du cache.

Corrigé : Le cache faisant 1K mots et les blocs étant de taille 128 mots, il y a 8 blocs dans le cache. La correspondance est directe, donc les mots mémoire 1 à 128 peuvent aller dans le bloc 0, les mots 129 à 256 dans le bloc 1,..., les mots 897 à 1024 dans le bloc 7, les mots 1025 à 1152 vont dans le bloc 0, les mots 1153 à 1280 dans le bloc 1...

Soient M le temps d'accès mémoire et C le temps d'accès au cache.

Première itération

- L'exécution des instructions de 17 à 128 prend $M + 111C$ car lors de l'accès à 17 on charge tout le premier bloc dans le cache.
- Lors de l'accès au mot 129, on charge tous les mots du bloc 2, c'est-à-dire tous les mots de 129 à 256, donc entre autre, toute la boucle interne est chargée. L'exécution prend : $M + 127C + 19 \times 75C$. En effet, on va exécuter une fois toutes les instructions de 129 à 256, soit $M + 127C$ et encore 19 fois la boucle interne qui se trouve en entier dans le cache et comprend 75 instructions.
- Les instructions 257 à 384 prennent $M + 127C$ (comme d'habitude). De même $385 \rightarrow 512$, $513 \rightarrow 640$, $641 \rightarrow 768$, $769 \rightarrow 896$, $896 \rightarrow 1024$. Soit un total de $6M + 6 \times 127C$ pour les instructions de 257 à 1024.

- Lorsque l'on accède à l'instruction 1025, il faut la chercher en mémoire et transférer le bloc dans le cache. Or ce bloc va dans le bloc 0 du cache, on enlève donc le bloc de mots d'adresses $1 \rightarrow 128$ du cache. Cela prend $\mathrm{M} + 127\mathrm{C}$ pour exécuter de 1024 à 1152.
- De même, l'accès au mot 1153 oblige à transférer le bloc $1153 \rightarrow 1280$ dans le bloc 1 du cache, à la place des mots d'adresses $129 \rightarrow 256$. Cela prend $\mathrm{M} + 47\mathrm{C}$ d'exécuter ces instructions (car à 1200 on boucle).

Pour la première itération, on a un total de : $10\mathrm{M} + 2\,599\mathrm{C}$.

Deuxième itération et suivante

- Le bloc 0 contient les adresses 1025 à 1152, il faut donc récupérer le bloc $1 \rightarrow 128$ pour accéder à l'instruction à l'adresse 23. D'où un temps de $\mathrm{M} + 105\mathrm{C}$ pour les instructions 23 à 128.
- De même, les instructions 129 à 256 ne sont pas dans le cache (car le bloc 1 du cache comprend les instructions mémoire 1153 à 1280). Donc, comme avant, il faut $\mathrm{M} + 127\mathrm{C} + 19 \times 75\mathrm{C}$ pour toutes ces instructions en comptant la boucle interne.
- Par contre, les instructions 257 à 1024 sont dans le cache, elles n'ont pas été remplacées. Il faut donc $6 \times 128\mathrm{C}$ pour les exécuter.
- Le bloc 0 du cache ne contient plus les mots 1025 à 1152 : il faut $\mathrm{M} + 127\mathrm{C}$.
- De même, le bloc 1 ne contient pas ce qu'il faut : $\mathrm{M} + 47\mathrm{C}$.

La deuxième itération prend : $4\mathrm{M} + 2\,599\mathrm{C}$.
Les 8 itérations suivantes sont pareilles : $8 \times (4\mathrm{M} + 2\,599\mathrm{C})$

Fin du programme

- Les instructions 1201 à 1280 sont dans le cache : $80\mathrm{C}$
- On transfert 1281 à 1408 : $\mathrm{M} + 127\mathrm{C}$.
- On transfert 1409 à 1500 (en fait 1536) : $\mathrm{M} + 91\mathrm{C}$.

La fin du programme prend : $2\mathrm{M} + 298\mathrm{C}$.

Total

Temps total : $10\mathrm{M} + 2\,599\mathrm{C} + 9 \times (4\mathrm{M} + 2599\mathrm{C}) + 2\mathrm{M} + 298\mathrm{C} = 48\mathrm{M} + 26\,288\mathrm{C} = 267\,680\,\mathrm{ns}$.
Efficacité : $26\,288/(26\,288 + 48) = 99{,}8\%$
Si l'on avait pas eu de cache, on aurait fait $48 + 26\,288$ accès mémoire, soit un temps de $2\,633\,600\,\mathrm{ns}$, donc un gain de temps d'un facteur $9{,}8 = (2\,633\,600/267\,680)$.