Chapitre 22

Circuits et logique booléenne

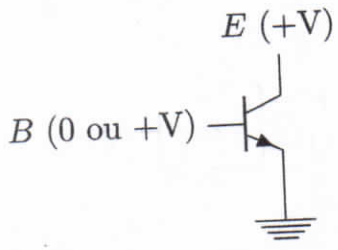
Notions introduites

- transistors, portes logiques
- tables de vérité
- fonctions et expressions booléennes
- circuits combinatoires

22.1 Portes logiques

Les circuits d'un ordinateur (mémoire, microprocesseur, etc.) manipulent uniquement des chiffres binaires 0 et 1 qui, en interne, sont simplement représentés par des tensions électriques. Ainsi, le chiffre 0 est représenté par une tension basse (proche de 0 volt) et le chiffre 1 par une tension haute (que l'on notera $+V$ volts, car cette tension varie selon les circuits).

Les opérateurs (logiques ou arithmétiques) sur ces nombres binaires sont construits à partir de *circuits électroniques* dont les briques élémentaires sont appelées *transistors*. Les transistors que l'on trouve dans les circuits des ordinateurs se comportent comme des interrupteurs qui laissent ou non passer un courant électrique, selon le mode du *tout ou rien*, comme représenté graphiquement de la manière suivante.



Dans ce schéma, la commande de l'interrupteur est jouée par la broche B. Lorsqu'elle est sous tension haute, la broche B laisse passer le courant entre

la broche E (sous tension haute) et la masse (le sens du courant est indiqué par la petite flèche), ce qui a pour effet de passer E sous la tension basse (en pratique, il y a des résistances placées aux bons endroits pour ne pas avoir de courts-circuits). Inversement, quand B est sous tension basse, la broche E reste sous tension haute.

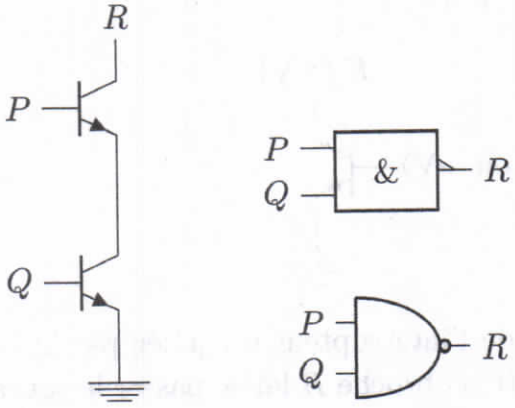
Ce simple transistor permet de réaliser une opération élémentaire appelée *porte logique* NON (appelée NOT en anglais). Une porte logique est une fonction qui prend un ou plusieurs *bits* en entrée et qui produit un *bit* en sortie.

La porte NOT implantée par un transistor est la plus simple de toutes les portes. Elle n'a qu'un seul *bit* en entrée (P) et sa sortie (Q) vaut 0 quand l'entrée vaut 1, et inversement elle vaut 1 quand son entrée est à 0. Graphiquement, on représente la porte NOT comme dans le schéma ci-dessous, avec à gauche la notation américaine et à droite la notation européenne.

Pour représenter le calcul réalisé par une porte logique, on utilise une *table logique* qui relie les valeurs des entrées à la valeur du résultat. La table logique de la porte NOT est donnée ci-dessous.

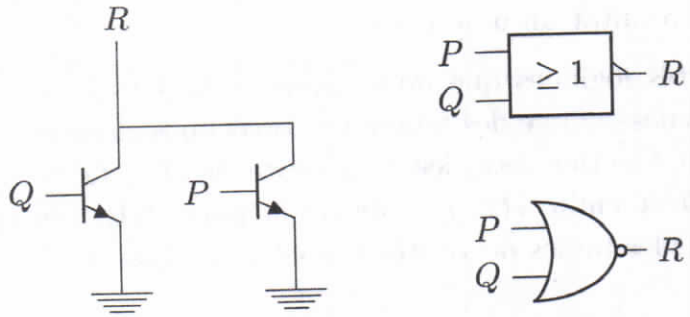
Porte NOT	
P	Q
0	1
1	0

On peut fabriquer d'autres portes logiques en combinant plusieurs transistors. Par exemple, en combinant deux transistors en série comme ci-dessous (schéma de gauche) on peut fabriquer la porte NON ET (appelée NAND en anglais) qui, pour deux entrées P et Q, produit un résultat R dont le calcul est donné par la table de vérité à droite. On donne également ci-dessous les schémas européen (en haut) et américain (en bas) pour cette porte.



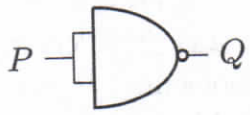
Porte NAND		
P	Q	R
0	0	1
0	1	1
1	0	1
1	1	0

De la même manière, en combinant deux transistors en parallèle, on obtient la porte NON OU (NOR en anglais) donnée ci-dessous (avec les schémas pour la représenter et sa table de vérité).



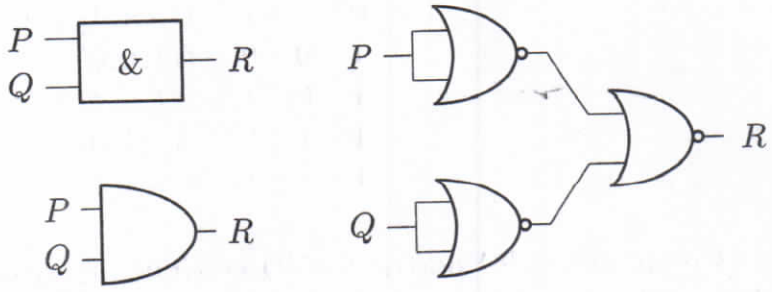
Porte NOR		
P	Q	R
0	0	1
0	1	0
1	0	0
1	1	0

Les portes NAND et NOR sont fondamentales dans les circuits électroniques car elles sont *complètes*, c'est-à-dire que n'importe quel circuit peut être conçu en utilisant *uniquement* ces deux portes. Par exemple, la porte NOT peut être fabriquée à partir d'une porte NAND en reliant les deux entrées de cette porte, comme ci-dessous.



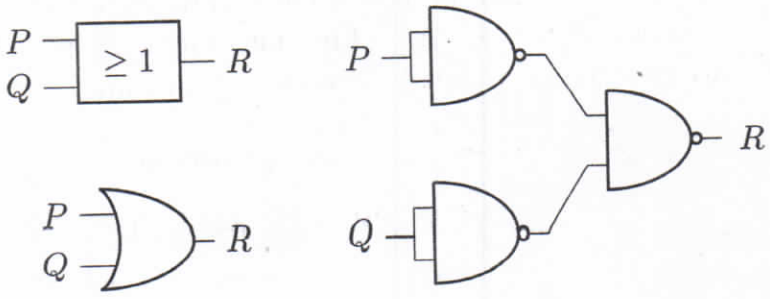
Une autre porte logique très importante est la porte ET (AND en anglais). Elle peut aussi être construite avec plusieurs portes NOR. Voici ci-dessous la table de vérité de la porte ET, ses représentations symboliques américaine (en bas) et européenne (en haut) ainsi qu'un schéma de construction avec des portes NOR.

Porte AND		
P	Q	R
0	0	0
0	1	0
1	0	0
1	1	1



De la même manière, la porte OU (OR en anglais) peut aussi être construite avec plusieurs portes NAND.

Porte OR		
P	Q	R
0	0	0
0	1	1
1	0	1
1	1	1



22.2 Fonctions booléennes

Certains circuits électroniques peuvent être décrits comme des fonctions booléennes, c'est-à-dire des fonctions qui prennent en entrée un ou plusieurs *bits* et qui produisent en résultat un unique *bit*.

Par exemple, les portes logiques que nous avons vues dans la section précédente peuvent être vues comme des fonctions booléennes élémentaires. Ainsi, si on note $\neg(x)$ la fonction associée à la porte NOT, $\wedge(x, y)$ celle associée à la porte AND et enfin $\vee(x, y)$ celle de la porte OR, ces trois fonctions sont définies par les tables de vérités rappelées ci-dessous.

x	$\neg(x)$
0	1
1	0

x	y	$\wedge(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$\vee(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

Plus généralement, la table de vérité d'une fonction avec n *bits* en entrée aura 2^n lignes correspondant aux 2^n combinaisons possibles des entrées. Par exemple, une fonction booléenne f avec trois entrées x, y et z sera entièrement définie par une table de vérité à $2^3 = 8$ lignes, de la forme suivante.

x	y	z	$f(x, y, z)$
0	0	0	$f(0,0,0)$
0	0	1	$f(0,0,1)$
0	1	0	$f(0,1,0)$
0	1	1	$f(0,1,1)$
1	0	0	$f(1,0,0)$
1	0	1	$f(1,0,1)$
1	1	0	$f(1,1,0)$
1	1	1	$f(1,1,1)$

Comme pour les portes NAND et NOR, les trois fonctions booléennes élémentaires $\neg(x)$, $\wedge(x, y)$ et $\vee(x, y)$ forment une base complète pour la constructions des autres fonctions booléennes, c'est-à-dire que toute fonction booléenne peut être définie à l'aide d'une combinaison de ces trois fonctions sur ses entrées. Aussi, pour simplifier la définition des fonctions booléennes, on utilisera plutôt ces trois fonctions élémentaires comme des opérateurs (unaire ou binaires). Ainsi, on écrira et dira que

$\neg x$ est la *négation* de x,

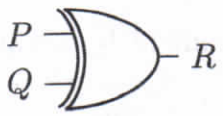
$x \wedge y$ est la conjonction de x et y,

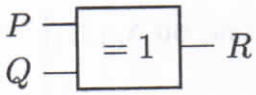
$x \vee y$ est la *disjonction* de x et y.

Ces opérateurs sont appelés *opérateurs booléens* pour rendre hommage au mathématicien et philosophe Georges Boole qui inventa au 19e siècle ce calcul (on dit aussi *algèbre*) basé sur ces opérateurs et ces chiffres binaires, qu'on appelle aussi des chiffres booléens.

Parmi les opérateurs élémentaires que nous n'avons pas encore présentés figure le *ou exclusif* qui est fréquemment utilisé dans les définitions de fonctions. Cet opérateur binaire, noté $x \oplus y$, est défini par la table de vérité ci-dessous (on donne également la représentation symbolique américaine et européenne de la porte nommée XOR en anglais). Il correspond au *ou* de la carte d'un restaurant, quand il est indiqué dans un menu que l'on peut prendre *fromage ou dessert*. Comme chacun sait, cela veut dire que l'on peut prendre soit un fromage, soit un dessert, mais pas les deux en même temps. Cela se traduit par une porte logique dont le résultat est à 1 quand *une et une seule* de ses entrées vaut 1, et qui renvoie 0 dans les deux autres cas.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0





Expressions booléennes. En utilisant ces opérateurs, on peut définir n'importe quelle fonction booléenne comme une expression booléenne sur ses entrées. Par exemple, l'égalité suivante définit une fonction f avec trois paramètres x, y et z à l'aide d'une expression booléenne sur ces variables :

$$f(x, y, z) = (x \wedge y) \oplus (\neg y \vee z)$$

Pour calculer la table de vérité associée à la fonction f, il suffit alors de procéder comme pour un calcul arithmétique ordinaire en calculant les résultats pour toutes les sous-expressions, en commençant par les calculs en profondeur puis en remontant. Sur notre exemple, cela revient tout d'abord à calculer les résultats des expressions $x \wedge y$ et $\neg y$, puis $\neg y \vee z$ et enfin le résultat final.

x	y	z	$(x \wedge y)$	$\neg y$	$(\neg y \vee z)$	$(x \wedge y) \oplus (\neg y \vee z)$
0	0	0	0	1	1	1
0	0	1	0	1	1	1
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	0	0	1
1	1	1	1	0	1	0

Une expression booléenne peut aussi être manipulée comme n'importe quelle expression arithmétique. Pour cela, le calcul avec les opérateurs booléens obéit à quelques identités élémentaires dont certaines sont données dans le tableau ci-dessous.

involutif :	$\neg(\neg x) = x$	
neutre :	$1 \wedge x = x$	$0 \vee x = x$
absorbant :	$0 \wedge x = 0$	$1 \vee x = 1$
idempotence :	$x \wedge x = x$	$x \vee x = x$
complément :	$x \wedge \neg x = 0$	$x \vee \neg x = 1$
commutativité :	$x \wedge y = y \wedge x$	$x \vee y = y \vee x$
associativité :	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	$x \vee (y \vee z) = (x \vee y) \vee z$
distributivité :	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
De Morgan :	$\neg(x \wedge y) = \neg x \vee \neg y$	$\neg(x \vee y) = \neg x \wedge \neg y$

Par exemple, en utilisant ces lois, on peut montrer l'égalité suivante.

$$\neg(y \wedge (x \vee \neg y)) = (\neg x \vee \neg y)$$

Pour cela, on applique successivement les identités indiquées à droite.

$\neg(y \wedge (x \vee \neg y))$	$=$	$\neg y \vee \neg(x \vee \neg y)$	De Morgan
	$=$	$\neg y \vee (\neg x \wedge y)$	De Morgan et involutif
	$=$	$(\neg y \vee \neg x) \wedge (\neg y \vee y)$	distributivité
	$=$	$(\neg y \vee \neg x) \wedge 1$	complément
	$=$	$(\neg y \vee \neg x)$	neutre
	$=$	$(\neg x \vee \neg y)$	commutativité

22.3 Circuits combinatoires

D'une manière générale, les circuits électroniques possèdent plusieurs entrées et plusieurs sorties. Quand les sorties dépendent *directement* et *uniquement* des entrées, on parle de *circuits combinatoires*. Il existe d'autres types de circuits, comme les *circuits séquentiels*, où les sorties peuvent dépendre également des valeurs précédentes des entrées (reçues dans le passé). Ces circuits, que nous n'aborderons pas dans ce livre, possèdent donc une capacité de mémorisation (appelée *état du circuit*) qui est utilisée pour construire des composants mémoires (RAM, registres, etc.).

La construction des circuits (combinatoires ou autres) ressemble à un jeu de Lego dans lequel on assemble des circuits élémentaires pour former des circuits plus complexes. En général, on ne part pas des portes logiques élémentaires vues précédemment, mais de circuits d'un peu plus haut niveau, qui sont conçus à partir de portes logiques. Parmi ces circuits, il y a par exemple des *décodeurs* (pour sélectionner une sortie à partir des entrées), des *multiplexeurs* (pour sélectionner une entrée de données à partir

des entrées de contrôle), des circuits pour *comparer* les entrées entre elles, ou bien encore des circuits arithmétiques (addition, soustraction, décalage, etc.). Nous présentons dans la suite deux de ces circuits.

Décodeur. Un décodeur n *bits* possède n entrées et 2^n sorties. Les n *bits* en entrée sont utilisés pour mettre à 1 la sortie dont le numéro est égal au nombre codé (en base 2) par les entrées et mettre les autres sorties à 0.

Par exemple, un décodeur 2 *bits*, avec 2 entrées (e_0 et e_1) et 4 sorties (s_0, s_1, s_2, s_3), correspond à la table de vérité suivante.

entrées		sorties			
e_0	e_1	s_0	s_1	s_2	s_3
0	0	1	0	0	0
1	0	0	1	0	0
0	1	0	0	1	0
1	1	0	0	0	1

Le fonctionnement d'un décodeur peut aussi se définir à l'aide des quatre expressions booléennes suivantes :

$$\begin{aligned} s_0 &= \neg e_0 \wedge \neg e_1 \\ s_1 &= e_0 \wedge \neg e_1 \\ s_2 &= \neg e_0 \wedge e_1 \\ s_3 &= e_0 \wedge e_1 \end{aligned}$$

Ainsi, on peut construire un décodeur en assemblant des portes NOT et AND comme décrit dans la figure 22.1.

Additionneur. Le circuit pour additionner des nombres binaires de n *bits* est construit en mettant en cascades des additionneurs 1 *bit* qui réalisent chacun l'addition de deux nombres de 1 *bit*. Ces additionneurs 1 *bit* sont eux-mêmes construits à partir d'un circuit encore plus simple appelé *demi-additionneur*.

Un demi-additionneur 1 *bit* prend en entrée deux *bits* e_0 et e_1 et il envoie sur une sortie s la somme $e_0 + e_1$. Si ce calcul provoque une retenue, il positionne alors à 1 une autre sortie c, qui indique la retenue éventuelle. La table de vérité d'un demi-additionneur 1 *bit* est donnée ci-dessous.

entrées		sorties	
e_0	e_1	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

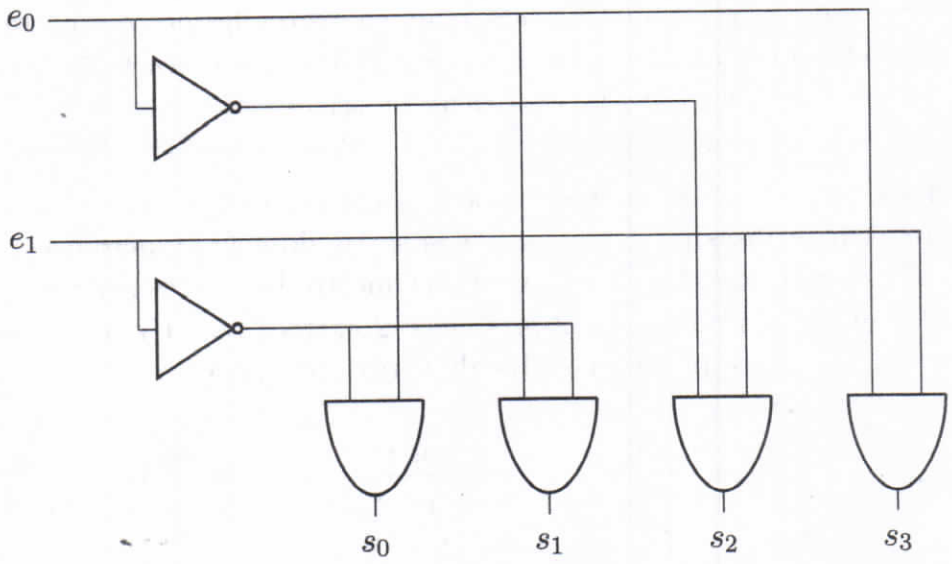


Figure 22.1 – Schéma d'un décodeur 4 *bits*.

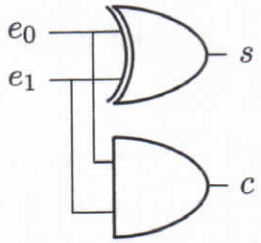


Figure 22.2 – Schéma d'un demi-additionneur 1*bit*.

Comme on peut le voir sur cette table, le fonctionnement d'un demi-additionneur peut aussi se définir à l'aide des deux expressions booléennes suivantes.

$$\begin{aligned} s &= e_0 \oplus e_1 \\ c &= e_0 \wedge e_1 \end{aligned}$$

En utilisant ces expressions, on peut facilement déduire le schéma de construction d'un demi-additionneur en assemblant deux portes XOR et AND, comme décrit dans la figure 22.2.

Pour réaliser un additionneur 1 *bit* complet, il faut prendre en compte la retenue éventuelle de l'addition de deux *bits* précédents. Ainsi, un additionneur 1 *bit* est un circuit qui prend en entrées deux *bits* e_0, e_1 et une retenue c_0, qui produit le résultat $s = e_0 + e_1 + c_0$, en positionnant éventuellement la retenue c finale.

On construit cet additionneur 1 *bit* en utilisant deux demi-additionneurs. Le premier demi-additionneur calcule la somme $s_0 = e_0 + e_1$, en positionnant éventuellement à 1 une retenue c_1. Le second demi-additionneur calcule la

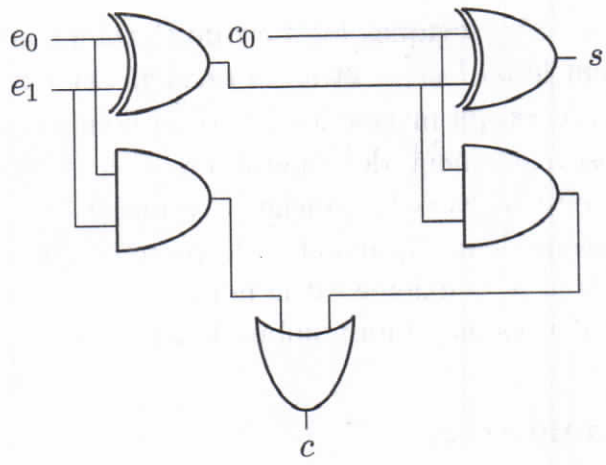


Figure 22.3 – Schéma d'un additionneur 1*bit*.

somme $s = c_0 + s_0$, en positionnant éventuellement à 1 une retenue c_2. La retenue finale c vaut 1 si l'une des deux retenues c_1 ou c_2 est à 1. La table de vérité d'un additionneur 1 *bit* complet est donnée ci-dessous.

entrées			sorties	
e_0	e_1	c_0	s	c
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

En utilisant deux schémas de demi-additionneur 1 *bit*, on peut facilement déduire le schéma de construction d'un additionneur 1 *bit* complet comme décrit dans la figure 22.3.

Opérations sur les *bits* en Python. Python dispose de nombreux opérateurs qui agissent directement sur les nombres au niveau des *bits*. Ces opérateurs sont appélés *opérateurs bit-à-bit* (opérateurs *bitwise* en anglais). Par exemple, l'opérateur & réalise un *et logique* entre les *bits* de deux nombres.

```
>>> bin (0b0101 & 0b1100)
'0b100'
```

Comme on le voit dans cet exemple, Python n'affiche les *bits* d'un nombre binaire qu'à partir du premier *bit* non nul. Cela est dû au fait que les nombres binaires sont infinis en Python.

Les autres opérateurs logiques *bit-à-bit* de Python sont `x | y` et `x ^ y` pour respectivement le *ou logique* et le *ou exclusif* entre `x` et `y`, ainsi que le complément à 1 noté `~x` qui inverse les *bits* d'un nombre `x`.

Python propose également des opérateurs pour décaler les *bits* d'un nombre vers la droite ou vers la gauche. Ces opérateurs de *décalage*, notés `<<` et `>>` prennent deux arguments : le premier est le nombre dont il faut décaler les *bits* et le deuxième est le nombre de position à décaler. Par exemple, pour décaler les *bits* d'un nombre de 2 positions vers la gauche, on écrira

```
>>> bin(0b0001010 << 2)
'0b101000'
```

Il est intéressant de noter que ces opérations de décalage sont des opérateurs extrêmement efficaces pour multiplier ou diviser un nombre par une puissance de 2.

À retenir. Les circuits d'un ordinateur sont fabriqués à partir de **portes logiques** élémentaires, elles-mêmes construites à partir de **transistors** où les valeurs booléennes 0 et 1 sont matérialisées par des courants électriques. Les portes logiques permettent de construire des **fonctions booléennes** arbitraires, telles que des décodeurs ou des additionneurs. Le langage Python permet d'appliquer des opérations **bit-à-bit** sur la représentation en base 2 des entiers.

Exercices

Exercice 231 Écrire en Python une fonction `xor(x,y)` qui réalise l'opérateur du même nom sur les booléens.

Solution page 490 □

Exercice 232 Montrer de deux manières différentes l'égalité suivante.

$$(x \wedge y) \vee (\neg y \wedge z) = (x \vee \neg y) \wedge (y \vee z)$$

Solution page 490 □

Exercice 233 Définir une fonction booléenne f sur deux variables x et y qui vaut 1 si et seulement si deux variables ont la même valeur (qu'elle soit 0 ou 1), en utilisant uniquement les opérations NON, ET, OU ou OU exclusif. Donner sa table de vérité.

Solution page 490 □

Exercice 234 On considère la fonction booléenne à trois variables suivante :

$$f(x, y, z) = (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z)$$

Donner sa table de vérité. Que fait cette fonction ? Donner une expression booléenne plus simple pour cette fonction. Solution page 491 □

Exercice 235 Expliquer comment combiner deux additionneurs 1 *bit* pour obtenir un additionneur 2 *bits*. Donner la table de vérité de cet additionneur 2 *bits*. Elle doit avoir seize lignes.

Solution page 491 □

Exercice 236 Écrire un programme Python qui affiche la table d'une opération logique pour tous les entiers de n *bits*. Par exemple, pour l'opération `&` (ET logique) et $n = 3$, le programme devra afficher ceci :

```
    000 001 010 011 100 101 110 111
000 000 000 000 000 000 000 000 000
001 000 001 000 001 000 001 000 001
010 000 000 010 010 000 000 010 010
```

Pour afficher un entier `x` en base 2, on peut se servir de `format(x, 'b')`. Si on veut que cela soit joli, avec exactement n chiffres, il faut se fatiguer un peu. Solution page 492 □

Exercice 237 Un multiplexeur k *bits* permet de sélectionner une entrée parmi 2^k disponibles. Un tel circuit a $k + 2^k$ *bits* en entrée. Les k premiers $(a_0, \ldots, a_{k-1})$ sont les *bits* d'adresse, ils servent à coder le numéro de l'entrée à sélectionner parmi les 2^k entrées suivantes $(b_0, \ldots, b_{2^k-1})$. La sortie s du multiplexeur est égale à la sortie sélectionnée.

Donner tout d'abord la table de vérité d'un multiplexeur 1 *bit* ainsi que l'expression booléenne qui définit sa sortie. À l'aide de portes logiques, définir ensuite le schéma de ce multiplexeur. Solution page 492 □