

# Circuits logiques

Terminale ISN Lycée du Parc

## Table des matières

<b>Crédits</b>	<b>1</b>
<b>Préambule</b>	<b>1</b>
<b>1 Portes logiques</b>	<b>2</b>
1.1 Le transistor porte logique de base . . . . .	2
1.2 D'autres portes logiques . . . . .	3
1.2.1 Transistors en série ou en parallèle . . . . .	3
1.2.2 Portes logiques et fonctions logiques élémentaires . . . . .	5
<b>2 Fonctions booléennes</b>	<b>8</b>
2.1 Fonctions booléennes . . . . .	8
2.2 Dresser la table de vérité d'une fonction booléenne . . . . .	10
2.3 Exprimer une fonction booléenne à partir de sa table de vérité . . . . .	10
<b>3 Circuits combinatoires</b>	<b>11</b>
3.1 Définition . . . . .	11
3.2 Décodeur avec 2 bits d'entrées . . . . .	12
3.3 Demi-additionneur et additionneur 1 bit . . . . .	12
3.4 Simuler le hasard . . . . .	14
<b>4 Opérations bit à bit en Python</b>	<b>16</b>

## Crédits

*Ce cours est largement inspiré du chapitre 22 du manuel NSI de la collection Tortue chez Ellipsen auteurs : Ballabonski, Conchon, Filliatre, N'Guyen.*

## Préambule

Les circuits d'un ordinateur manipulent uniquement des 0 ou des 1 représentés en interne par des tensions hautes ou basses. Les premiers ordinateurs construits dans la période 1945-1950 sont basés sur une technologie de tube à vide ou tube électrique. En 1947, aux laboratoires Bell, [Shockley](#), [Bardeen](#)

# Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

Transistor count

50,000,000,000  
10,000,000,000  
5,000,000,000  
1,000,000,000  
500,000,000  
100,000,000  
50,000,000  
10,000,000  
5,000,000  
1,000,000  
500,000  
100,000  
50,000  
10,000  
5,000  
1,000

1970 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012 2014 2016 2018

72-core Xeon Phi  
SPARC M7  
IBM z13 Storage Controller  
18-core Xeon Haswell-E5  
Xbox One main SoC  
61-core Xeon Phi  
12-core POWER7  
8-core Xeon Nehalem-EX  
86-core Xeon 7400  
Dual-core Itanium 2  
Pentium D Prester  
Ranger-2 with 9 MB cache  
Itanium 2 Madison 6M  
Pentium D Smithfield  
Itanium 2 McKinley  
Pentium 4 Prescott 2M  
AMD K8  
Pentium 4 Northwood  
Pentium 4 Willamette  
Pentium 4 Mobile Dison  
AMD K8  
Pentium III Tualatin  
Pentium III Coppermine  
AMD K8  
Pentium III Kalmal  
Pentium III Deschutes  
Pentium Pro  
Pentium  
AMD K5  
SA1110  
ARM1100  
ARM1000  
ARM6  
ARM1  
ARM2  
ARM3  
ARM4  
ARM5  
ARM6  
ARM7  
ARM8  
ARM9  
ARM10  
ARM11  
ARM12  
ARM13  
ARM14  
ARM15  
ARM16  
ARM17  
ARM18  
ARM19  
ARM20  
ARM21  
ARM22  
ARM23  
ARM24  
ARM25  
ARM26  
ARM27  
ARM28  
ARM29  
ARM30  
ARM31  
ARM32  
ARM33  
ARM34  
ARM35  
ARM36  
ARM37  
ARM38  
ARM39  
ARM40  
ARM41  
ARM42  
ARM43  
ARM44  
ARM45  
ARM46  
ARM47  
ARM48  
ARM49  
ARM50  
ARM51  
ARM52  
ARM53  
ARM54  
ARM55  
ARM56  
ARM57  
ARM58  
ARM59  
ARM60  
ARM61  
ARM62  
ARM63  
ARM64  
ARM65  
ARM66  
ARM67  
ARM68  
ARM69  
ARM70  
ARM71  
ARM72  
ARM73  
ARM74  
ARM75  
ARM76  
ARM77  
ARM78  
ARM79  
ARM80  
ARM81  
ARM82  
ARM83  
ARM84  
ARM85  
ARM86  
ARM87  
ARM88  
ARM89  
ARM90  
ARM91  
ARM92  
ARM93  
ARM94  
ARM95  
ARM96  
ARM97  
ARM98  
ARM99  
ARM100  
ARM101  
ARM102  
ARM103  
ARM104  
ARM105  
ARM106  
ARM107  
ARM108  
ARM109  
ARM110  
ARM111  
ARM112  
ARM113  
ARM114  
ARM115  
ARM116  
ARM117  
ARM118  
ARM119  
ARM120  
ARM121  
ARM122  
ARM123  
ARM124  
ARM125  
ARM126  
ARM127  
ARM128  
ARM129  
ARM130  
ARM131  
ARM132  
ARM133  
ARM134  
ARM135  
ARM136  
ARM137  
ARM138  
ARM139  
ARM140  
ARM141  
ARM142  
ARM143  
ARM144  
ARM145  
ARM146  
ARM147  
ARM148  
ARM149  
ARM150  
ARM151  
ARM152  
ARM153  
ARM154  
ARM155  
ARM156  
ARM157  
ARM158  
ARM159  
ARM160  
ARM161  
ARM162  
ARM163  
ARM164  
ARM165  
ARM166  
ARM167  
ARM168  
ARM169  
ARM170  
ARM171  
ARM172  
ARM173  
ARM174  
ARM175  
ARM176  
ARM177  
ARM178  
ARM179  
ARM180  
ARM181  
ARM182  
ARM183  
ARM184  
ARM185  
ARM186  
ARM187  
ARM188  
ARM189  
ARM190  
ARM191  
ARM192  
ARM193  
ARM194  
ARM195  
ARM196  
ARM197  
ARM198  
ARM199  
ARM200  
ARM201  
ARM202  
ARM203  
ARM204  
ARM205  
ARM206  
ARM207  
ARM208  
ARM209  
ARM210  
ARM211  
ARM212  
ARM213  
ARM214  
ARM215  
ARM216  
ARM217  
ARM218  
ARM219  
ARM220  
ARM221  
ARM222  
ARM223  
ARM224  
ARM225  
ARM226  
ARM227  
ARM228  
ARM229  
ARM230  
ARM231  
ARM232  
ARM233  
ARM234  
ARM235  
ARM236  
ARM237  
ARM238  
ARM239  
ARM240  
ARM241  
ARM242  
ARM243  
ARM244  
ARM245  
ARM246  
ARM247  
ARM248  
ARM249  
ARM250  
ARM251  
ARM252  
ARM253  
ARM254  
ARM255  
ARM256  
ARM257  
ARM258  
ARM259  
ARM260  
ARM261  
ARM262  
ARM263  
ARM264  
ARM265  
ARM266  
ARM267  
ARM268  
ARM269  
ARM270  
ARM271  
ARM272  
ARM273  
ARM274  
ARM275  
ARM276  
ARM277  
ARM278  
ARM279  
ARM280  
ARM281  
ARM282  
ARM283  
ARM284  
ARM285  
ARM286  
ARM287  
ARM288  
ARM289  
ARM290  
ARM291  
ARM292  
ARM293  
ARM294  
ARM295  
ARM296  
ARM297  
ARM298  
ARM299  
ARM300  
ARM301  
ARM302  
ARM303  
ARM304  
ARM305  
ARM306  
ARM307  
ARM308  
ARM309  
ARM310  
ARM311  
ARM312  
ARM313  
ARM314  
ARM315  
ARM316  
ARM317  
ARM318  
ARM319  
ARM320  
ARM321  
ARM322  
ARM323  
ARM324  
ARM325  
ARM326  
ARM327  
ARM328  
ARM329  
ARM330  
ARM331  
ARM332  
ARM333  
ARM334  
ARM335  
ARM336  
ARM337  
ARM338  
ARM339  
ARM340  
ARM341  
ARM342  
ARM343  
ARM344  
ARM345  
ARM346  
ARM347  
ARM348  
ARM349  
ARM350  
ARM351  
ARM352  
ARM353  
ARM354  
ARM355  
ARM356  
ARM357  
ARM358  
ARM359  
ARM360  
ARM361  
ARM362  
ARM363  
ARM364  
ARM365  
ARM366  
ARM367  
ARM368  
ARM369  
ARM370  
ARM371  
ARM372  
ARM373  
ARM374  
ARM375  
ARM376  
ARM377  
ARM378  
ARM379  
ARM380  
ARM381  
ARM382  
ARM383  
ARM384  
ARM385  
ARM386  
ARM387  
ARM388  
ARM389  
ARM390  
ARM391  
ARM392  
ARM393  
ARM394  
ARM395  
ARM396  
ARM397  
ARM398  
ARM399  
ARM400  
ARM401  
ARM402  
ARM403  
ARM404  
ARM405  
ARM406  
ARM407  
ARM408  
ARM409  
ARM410  
ARM411  
ARM412  
ARM413  
ARM414  
ARM415  
ARM416  
ARM417  
ARM418  
ARM419  
ARM420  
ARM421  
ARM422  
ARM423  
ARM424  
ARM425  
ARM426  
ARM427  
ARM428  
ARM429  
ARM430  
ARM431  
ARM432  
ARM433  
ARM434  
ARM435  
ARM436  
ARM437  
ARM438  
ARM439  
ARM440  
ARM441  
ARM442  
ARM443  
ARM444  
ARM445  
ARM446  
ARM447  
ARM448  
ARM449  
ARM450  
ARM451  
ARM452  
ARM453  
ARM454  
ARM455  
ARM456  
ARM457  
ARM458  
ARM459  
ARM460  
ARM461  
ARM462  
ARM463  
ARM464  
ARM465  
ARM466  
ARM467  
ARM468  
ARM469  
ARM470  
ARM471  
ARM472  
ARM473  
ARM474  
ARM475  
ARM476  
ARM477  
ARM478  
ARM479  
ARM480  
ARM481  
ARM482  
ARM483  
ARM484  
ARM485  
ARM486  
ARM487  
ARM488  
ARM489  
ARM490  
ARM491  
ARM492  
ARM493  
ARM494  
ARM495  
ARM496  
ARM497  
ARM498  
ARM499  
ARM500  
ARM501  
ARM502  
ARM503  
ARM504  
ARM505  
ARM506  
ARM507  
ARM508  
ARM509  
ARM510  
ARM511  
ARM512  
ARM513  
ARM514  
ARM515  
ARM516  
ARM517  
ARM518  
ARM519  
ARM520  
ARM521  
ARM522  
ARM523  
ARM524  
ARM525  
ARM526  
ARM527  
ARM528  
ARM529  
ARM530  
ARM531  
ARM532  
ARM533  
ARM534  
ARM535  
ARM536  
ARM537  
ARM538  
ARM539  
ARM540  
ARM541  
ARM542  
ARM543  
ARM544  
ARM545  
ARM546  
ARM547  
ARM548  
ARM549  
ARM550  
ARM551  
ARM552  
ARM553  
ARM554  
ARM555  
ARM556  
ARM557  
ARM558  
ARM559  
ARM560  
ARM561  
ARM562  
ARM563  
ARM564  
ARM565  
ARM566  
ARM567

tension haute ou basse qu'on peut assimiler aux valeurs binaires 1 et 0 d'un **bit**. Si la tension appliquée sur la grille est haute (bit à 1) alors le transistor laisse passer le courant entre la source d'énergie et la sortie et ce dernier passe à l'état de tension basse (bit à 0), sinon la sortie reste en tension haute (bit 1).

Une **fonction logique** prend un ou plusieurs bits en entrée et retourne un ou plusieurs bits en sortie. Une **table logique** représente toutes les sorties produites par une fonction logique pour toutes les entrées possibles.

Un transistor représente une fonction logique dont le bit d'entrée est l'état de tension de la grille et le bit de sortie, l'état de tension de la sortie. La **table logique** (table 1) associée est celle du **NON logique** ou **Inverseur**.

Fichier de test Logisim : [transistor.circ](#).

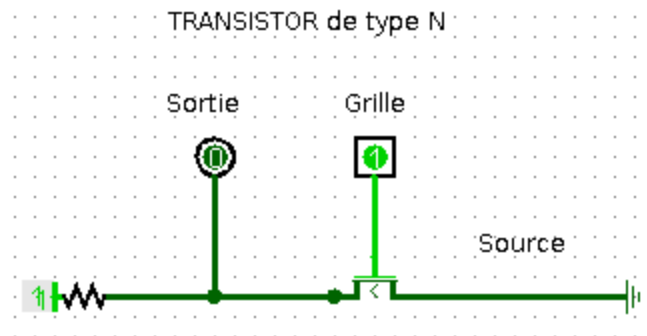
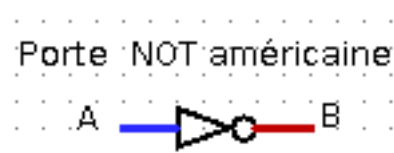
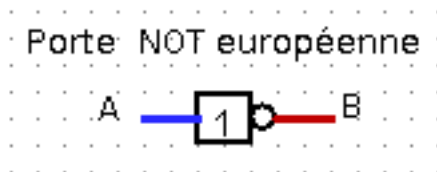


Table 1: Table logique d'une porte NON

A	B = NON(A)
0	1
1	0

Il existe deux conventions de représentation symbolique des portes logiques, une européenne et une américaine.



## 1.2 D'autres portes logiques

### 1.2.1 Transistors en série ou en parallèle



## Exercice 1

On donne ci-dessous les représentations de deux portes logiques :

- La **porte NAND** constituée de deux transistors en série
- La **porte NOR** constituée de deux transistors en parallèle

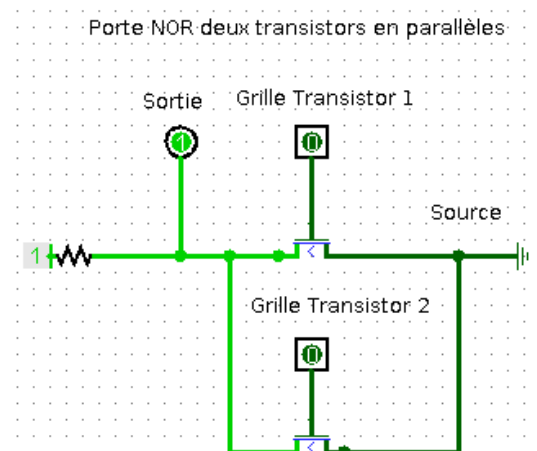
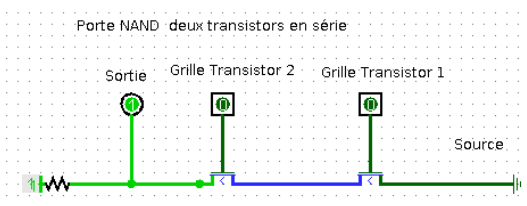
Chacune de ces portes logiques comportent deux bits d'entrée : A pour la grille du transistor 1 et B pour la grille du transistor 2 et un bit de sortie.

Compléter leurs tables logiques.

Vérifier avec [Logisim](#) et les fichiers [porte\\_NAND.circ](#) et [porte\\_NOR.circ](#).

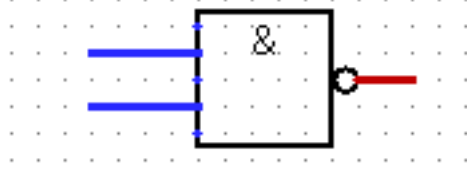
A	B	NAND(A, B)
0	0	
0	1	
1	0	
1	1	

A	B	NOR(A, B)
0	0	
0	1	
1	0	
1	1	

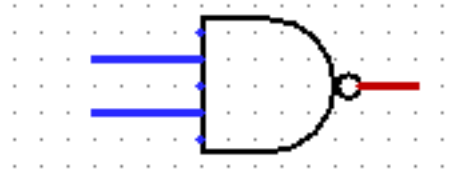


Voici les représentations symboliques des portes logiques NAND et NOR :

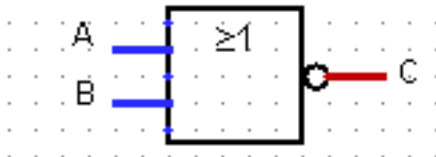
Porte NAND européenne



Porte NAND américaine



Porte NOR européenne



Porte NOR américaine



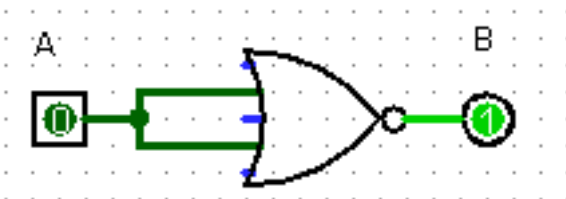
### 1.2.2 Portes logiques et fonctions logiques élémentaires



#### Exercice 2

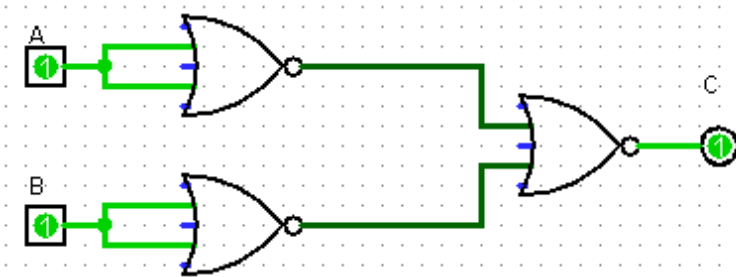
Fichier de test Logisim : [exercice2.circ](#).

1. Compléter la table logique de la porte logique représentée par le circuit ci-dessous. Quelle porte logique peut-on ainsi représenter ?



A	B = f(A)
0	
1	

2. Compléter la table logique de la porte logique représentée par le circuit ci-dessous. Quelle fonction logique correspond à cette porte logique ?



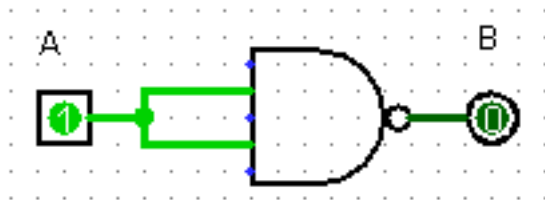
A	B	$C = g(A, B)$
0	0	
0	1	
1	0	
1	1	



### Exercice 3

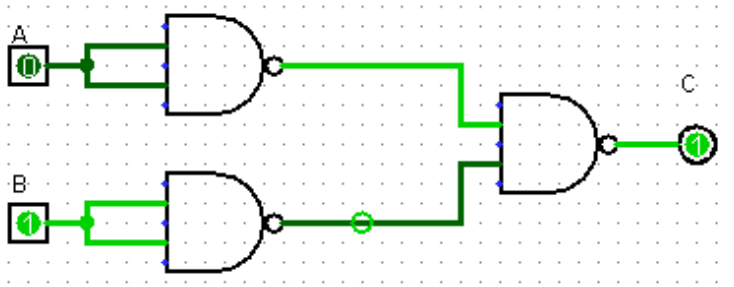
Fichier de test Logisim : [exercice3.circ](#).

1. Compléter la table logique de la porte logique représentée par le circuit ci-dessous. Quelle porte logique peut-on ainsi représenter ?



A	$B = f(A)$
0	
1	

2. Compléter la table logique de la porte logique représentée par le circuit ci-dessous. Quelle fonction logique correspond à cette porte logique ?



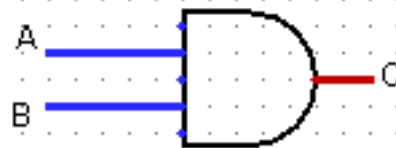
A	B	$C = g(A, B)$
0	0	
0	1	
1	0	
1	1	

Voici les représentations symboliques des portes logiques AND et OR :

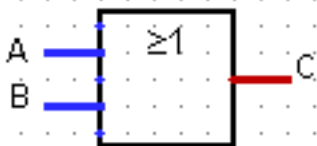
Porte AND européenne



Porte AND américaine



Porte OR européenne



Porte OR américaine



#### Exercice 4

1. Construire un circuit représentant une porte OR uniquement avec des portes NOR.
2. Construire un circuit représentant une porte AND uniquement avec des portes NAND.

Ainsi chacune des portes, NAND ou OR permet de construire les portes NOT, OR, AND. Toute porte logique pouvant s'exprimer à l'aide de ces trois portes, les portes NAND et OR sont dites *universelles*.

## 2 Fonctions booléennes

### 2.1 Fonctions booléennes



#### Définition 2

- Un **booléen** est un type de données pouvant prendre deux valeurs **True** (Vrai) ou **False** (Faux) qu'on représente numériquement par un **bit** de valeur 1 pour **True** ou 0 pour **False**. Electroniquement, les valeurs 1 et 0 se traduisent respectivement par des tensions haute ou basse.
- Une **fonction booléenne**  $f$  associe un booléen à un ou plusieurs booléens.
- Une **fonction booléenne** avec  $n$  arguments est définie sur un ensemble  $\{0;1\}^n$  à  $2^n$  valeurs et prend ses valeurs dans  $\{0;1\}$  qui a 2 éléments. On peut recenser les  $2^n$  évaluations d'une fonction booléenne à  $n$  arguments dans une **table de vérité** qui la définit entièrement. Il existe  $2^{2^n}$  fonctions booléennes à  $n$  arguments.
- Une **porte logique** est la représentation sous forme de circuit d'une fonction booléenne et sa **table logique** est la **table de vérité** de cette fonction.



#### Exercice 5

1. Compléter la fonction Python ci-dessous pour qu'elle affiche la table de vérité d'une fonction booléenne à deux entrées. Expliquer le rôle de la fonction `int`.

```
def table_verite_2bits(fonction):
    print('|{:~10}|{:~10}|{:~15}|'.format('a','b',fonction.__name__+'(a,b)'))
    for a in .....:
        for b in .....:
            print('|{:~10}|{:~10}|{:~15}|'.format(....., .....,
            int(fonction(bool(a),bool(b)))))
```

1. Vérifier que les tables de vérité affichées pour les fonctions `bool.__or__`, `bool.__and__` et `bool.__not__` sont correctes.

```
In [4]: table_verite_2bits(bool.__or__)
|    a    |    b    |  __or__(a,b) |
|    1    |    1    |             1 |
|    1    |    0    |             1 |
|    0    |    1    |             1 |
|    0    |    0    |             0 |
```





## Propriété 1

On peut exprimer toute fonction booléenne à l'aide de trois fonctions booléennes élémentaires :

- La *négation* de  $x$  est une fonction à 1 bit d'entrée (unaire) notée  $\neg x$  ou  $\bar{x}$ .  
Si  $x$  est un booléen, sa *négation* est **not**  $x$  en Python.

$x$	$\neg x$
0	1
1	0

- La *conjonction* de  $x$  et  $y$  est une fonction à 2 bits d'entrée (binaire) notée  $x \wedge y$  ou  $x.y$ .  
Si  $x$  et  $y$  sont des booléens, leur *conjonction* est  $x$  **and**  $y$  en Python.

$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

- La *disjonction* de  $x$  et  $y$  est une fonction à 2 bits d'entrée (binaire) notée  $x \vee y$  ou  $x + y$ .  
Si  $x$  et  $y$  sont des booléens, leur *disjonction* est  $x$  **or**  $y$  en Python

$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1



## Propriété 2

- Les fonctions booléennes élémentaires respectent un certain nombre de règles qui permettent de simplifier les expressions booléennes complexes :
  - opérateur involutif* :  $\neg(\neg x) = x$  et  $\bar{\bar{x}} = x$
  - élément neutre* :  $1 \wedge x = x$  et  $1.x = x$  ou  $0 \vee x = x$  et  $0 + x = x$
  - élément absorbant* :  $0 \wedge x = 0$  et  $0.x = 0$  ou  $1 \vee x = x$  et  $1 + x = 1$
  - idempotence* :  $x \wedge x = x$  et  $x.x = x$  ou  $x \vee x = x$  et  $x + x = x$
  - complément* :  $x \wedge (\neg x) = 0$  et  $x.(\bar{x}) = 0$  ou  $x \vee (\neg x) = 1$  et  $x + \bar{x} = 1$

- *commutativité* :  $x \wedge y = y \wedge x$  et  $x.y = y.x$  ou  $x \vee y = y \vee x$  et  $x + y = y + x$
- *associativité* :  $x \wedge (y \wedge z) = (x \wedge y) \wedge z$  et  $x.(y.z) = (x.y).z$  ou  $x \vee (y \vee z) = (x \vee y) \vee z$  et  $x + (y + z) = (x + y) + z$
- *distributivité* :  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$  et  $x.(y + z) = x.y + x.z$  ou  $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$  et  $x + (y.z) = (x + y).(x + z)$
- *loi de Morgan* :  $\neg(x \wedge y) = \neg x \vee \neg y$  et  $\overline{x.y} = \overline{x} + \overline{y}$  ou  $\neg(x \vee y) = \neg x \wedge \neg y$  et  $\overline{x + y} = \overline{x}.\overline{y}$

2. Les fonctions booléennes élémentaire respectent des règles de priorité : la *négarion* est prioritaire sur la *conjonction* qui est prioritaire sur la *disjonction*.

**Il est recommandé de mettre des parenthèses plutôt que d'appliquer les règles de priorité dans l'écriture des expressions booléennes.**

## 2.2 Dresser la table de vérité d'une fonction booléenne



### Exercice 6

Démontrer dans chaque cas l'égalité des expressions booléennes en utilisant les deux méthodes suivantes :

- **Méthode 1** : en comparant les tables de vérité des deux expressions booléennes ;
- **Méthode 2** : en utilisant les règles de simplification de la propriété 2.

1.  $x + x.y = x$
2.  $x + \overline{x}.y = x + y$
3.  $\overline{x.z + \overline{x}.y + y.z} = \overline{x.z} + \overline{\overline{x}.y}$
4.  $\overline{y.(x + \overline{y})} = \overline{y} + \overline{y}$
5.  $x.(\overline{x} + \overline{y}).(x + y) = x.\overline{y}$

## 2.3 Exprimer une fonction booléenne à partir de sa table de vérité



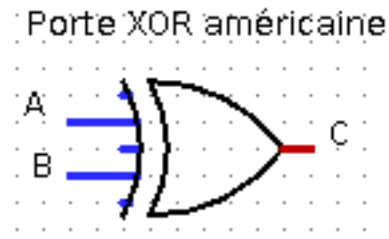
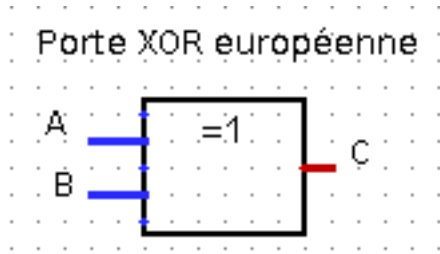
### Exercice 7

On considère la fonction booléenne dont la table de vérité est :

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

1. Exprimer chacune des lignes où la fonction prend la valeur 1 comme la *conjonction* des entrées en remplaçant chaque 1 par la variable qu'il représente et chaque 0 par la négation de la variable. Par exemple le 1 de la deuxième ligne s'écrira  $\bar{x}.y$ .
2. On peut alors écrire  $f(x, y)$  comme la *disjonction* des *formes conjonctives* obtenues à la question précédente. En déduire une expression booléenne de  $f(x, y)$ .
3. Ouvrir le logiciel [Logisim](#) et construire une porte logique représentant cette fonction booléenne.
4. Cette fonction s'appelle OU EXCLUSIF ou XOR. Ce nom vous paraît-il bien choisi ?

Voici les représentations symboliques de la porte logique XOR :



### 3 Circuits combinatoires

#### 3.1 Définition



##### Définition 3

Un **circuit logique combinatoire** permet de réaliser une ou plusieurs fonctions booléennes : ses sorties ne dépendent que de l'état actuel de ses entrées. Les portes logiques NOT, NOR, NAND, AND, OR et XOR sont des circuits combinatoires.

Il existe d'autres circuits, dits séquentiels, dont les sorties se calculent non seulement à partir de leurs valeurs d'entrée actuelles mais aussi à partir de leurs états précédents : le facteur temps intervient. Ils utilisent des circuits de mémoire pour mémoriser leurs états antérieurs.



##### Exercice 8

On considère la fonction booléenne  $f$  dont la table de vérité est donnée ci-dessous :

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

$$\frac{1 \quad 1 \quad 1}{\quad}$$

1. En utilisant la méthode exposée dans l'exercice , déterminer une expression booléenne de la fonction  $f$ .
2. Ouvrir le logiciel [Logisim](#) et construire un circuit combinatoire représentant cette fonction booléenne :
  - En utilisant les portes logiques NOT, NOR, NAND, AND, OR ou XOR.
  - En n'utilisant que des portes logiques NOT, AND ou OR.
  - En n'utilisant que des portes logiques NOR.

### 3.2 Décodeur avec 2 bits d'entrées



#### Exercice 9

On considère un circuit combinatoire qui possède deux entrées  $e_0$  et  $e_1$  et quatre sorties  $s_0$ ,  $s_1$ ,  $s_2$  et  $s_3$ .

La sortie indexée par le nombre dont le bit de poids faible est  $e_0$  et le bit de poids fort  $e_1$  est positionnée à 1 et les autres sorties à 0. Ce circuit est ainsi appelé **décodeur 2 bits**.

1. Compléter la table de vérité de ce circuit combinatoire.

$e_0$	$e_1$	$s_0$	$s_1$	$s_2$	$s_3$
0	0				
0	1				
1	0				
1	1				

2. En utilisant la méthode exposée dans l'exercice 7, déterminer une expression booléenne de chacune des sorties  $s_0$ ,  $s_1$ ,  $s_2$  et  $s_3$ , en fonction des entrées  $e_0$  et  $e_1$ .
3. Ouvrir le logiciel [Logisim](#) et construire un circuit combinatoire représentant un **décodeur 2 bits**.

### 3.3 Demi-additionneur et additionneur 1 bit



#### Exercice 10

1. Effectuer les additions binaires :  $0 + 0$ ,  $0 + 1$ ,  $1 + 0$  et  $1 + 1$ .
2. Un **demi-additionneur binaire 1 bit** est un circuit combinatoire qui possède :

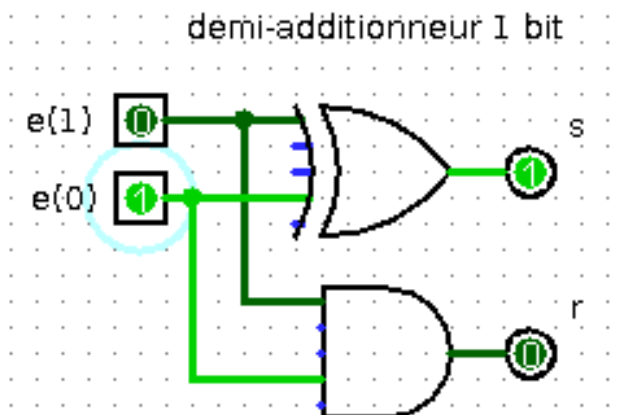
- deux entrées : deux bits d'opérande  $e_0$  et  $e_1$  ;
- deux sorties : un bit de résultat  $s$  et un bit de retenue sortante  $r$ .

La sortie  $s$  prend pour valeur le bit des unités et la sortie  $r$  le bit de retenue sortante, lorsqu'on additionne les deux bits d'entrée  $e_0$  et  $e_1$ .

1. Compléter la table de vérité de ce circuit combinatoire :

$e_0$	$e_1$	$s$	$r$
0	0		
0	1		
1	0		
1	1		

4. Justifier qu'un **demi-additionneur binaire 1 bit** peut être représenté par le circuit ci-dessous.



5. Ouvrir le logiciel [Logisim](#) et construire un circuit combinatoire représentant un **demi-additionneur binaire 1 bit**.



## Exercice 11

Un **additionneur binaire 1 bit** est un circuit combinatoire qui possède :

- trois entrées : deux bits d'opérande  $e_0$  et  $e_1$  et un bit de retenue entrante  $r_0$
- deux bits de sortie : un bit de résultat  $s_2$  et un bit de retenue sortante  $r_3$ .

1. Compléter les colonnes de la table de vérité d'un **additionneur binaire 1 bit** pour le bit de résultat  $s_2$  et le bit retenue sortante  $r_3$ .

$e_0$	$e_1$	$r_0$	$s_1 = \dots\dots$	$r_1 = \dots\dots$	$s_2 = \dots\dots$	$r_2 = \dots\dots$	$r_3 = \dots\dots$
0	0	0					
0	1	0					
1	0	0					

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

2. Un **additionneur binaire 1 bit** peut être réalisé à l'aide de deux **demi-additionneurs binaires 1 bit** :

- Le premier **demi-additionneur binaire 1 bit** prend en entrée les bits d'opérande  $e_0$  et  $e_1$  et retourne en sortie un bit de résultat intermédiaire  $s_1$  et un bit de retenue sortante intermédiaire  $r_1$ . Donner une expression booléenne de  $s_1$  et  $r_1$  en fonction de  $e_0$  et  $e_1$ .
- Le second **demi-additionneur binaire 1 bit** prend en entrée le bit de résultat  $s_1$  et le bit de retenue entrante  $r_0$  et retourne en sortie le bit de résultat final  $s_2$  et un bit de retenue sortante intermédiaire  $r_2$ . Donner une expression booléenne de  $s_2$  et  $r_2$  en fonction de  $s_1$  et  $r_0$ .
- Enfin, la retenue sortante  $r_3$  s'obtient à partir de la retenue sortante  $r_1$  du premier demi-additionneur et de la retenue sortante  $r_2$  du second. Donner une expression booléenne de  $r_3$  en fonction de  $r_1$  et  $r_2$ .

Compléter les colonnes  $s_1$ ,  $r_1$  et  $r_2$  de la table de vérité de **additionneur binaire à 1 bit**.

3. Avec le logiciel [Logisim](#) ouvrir le fichier contenant le demi-additionneur de l'exercice précédent.

- Ajouter un nouveau circuit avec **Add a circuit**, le nommer **additionneur1bit** puis copier/coller dedans le circuit du **demi-additionneur binaire 1 bit**. Compléter le circuit pour obtenir un **additionneur binaire 1 bit**.
- Ajouter un nouveau circuit avec **Add a circuit**, le nommer **additionneur2bits** puis copier/coller dedans le circuit de l' **additionneur binaire 1 bit**. Compléter le circuit pour obtenir un **additionneur binaire 2 bits**.

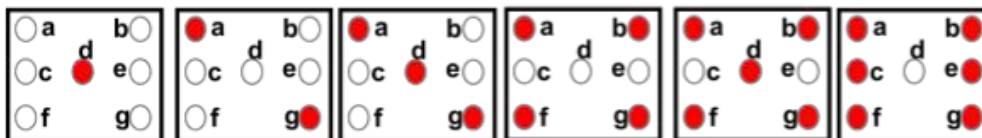
### 3.4 Simuler le hasard



#### Exercice 12

Dans cet exercice, on veut réaliser un circuit logique qui simule un dé électronique à diodes (LED).

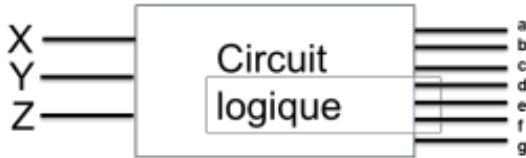
Les différentes combinaisons d'affichage du dé électronique sont représentées dans la figure ci-dessous :



Par exemple, si on veut afficher 3, il faut allumer les diodes a, d et g. Pour les combinaisons d'entrée  $(x,y,z) = (0,0,0)$  et  $(X,Y,Z) = (1,1,1)$  aucune diode ne doit être allumée.

Il s'agit d'une forme de **transcodeur 3 bits vers 8 bits**. Les 3 bits d'entrée représentent le codage d'un nombre sur 3 bits en notation positionnelle et les 8 bits de sortie représentent le codage de ce même nombre en notation additive. Par exemple si  $(x,y,z) = (1,0,1)$  en entrée, le nombre est  $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$  et il est représenté par cinq bits de sortie positionnés à 1. Pour simuler un dé à 6 faces, deux entrées,  $(x,y,z) = (0,0,0)$  et  $(x,y,z) = (1,1,1)$ , correspondent à la même sortie (tous les bits de sortie à 0), c'est pourquoi on peut réduire le nombre de sorties de 8 à 7.

Le circuit à réaliser doit donc comporter 7 sorties, soit une sortie par diode (a, b, c, d, e, f, g) et 3 entrées x, y, z.

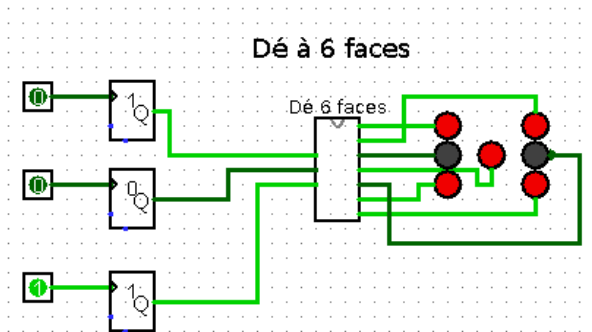


1. Compléter la table de vérité de ce circuit :

x	y	z	a	b	c	d	e	f	g
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
1	0	1							
1	1	0							
1	1	1							

2. Ouvrir le logiciel [Logisim](#). Aller dans Windows - Combinational - Analysis :

- dans l'onglet **Input** , indiquer les variables d'entrée du transcodeur ( x , y , z ) ;
- dans l'onglet **Output**, indiquer les variables de sortie ( a , . . . , g ) ;
- dans **Table** , saisir la table de vérité de chaque sortie ;
- dans **Expression** , on peut obtenir une expression booléenne pour chaque de sortie, et dans **Minimized** une expression simplifiée.
- construire le circuit avec le bouton **Build circuit** et le nommer **de6faces**
- compléter le circuit avec des **Random Generator** (outils **Memory**) en entrée et des **LED** (outils **Input - Output**) en sortie comme dans la figure ci-dessous.



## 4 Opérations bit à bit en Python



### Propriété 3

Les fonctions booléennes élémentaires (OR, AND, NOT, XOR) existent en **Python** sous la forme d'opérateurs booléens mais sont également implémentés sous la forme d'opérateurs bit à bit sur les nombres. Un *opérateur bit à bit* (*bitwise* en anglais) s'applique sur les bits de même poids des représentations binaires de ses opérandes.

Opérateur booléen	Opérateur bit à bit	Exemple
<code>and</code> , ET	<code>&amp;</code>	<pre>&gt;&gt;&gt; bin(0b101001 &amp; 0b101010) '0b101000'</pre>
<code>or</code> , OU	<code> </code>	<pre>&gt;&gt;&gt; bin(0b101001   0b101010) '0b101011'</pre>
<code>xor</code> , OU EXCLUSIF	<code>^</code>	<pre>&gt;&gt;&gt; bin(0b101001 ^ 0b101010) '0b000011'</pre>
<code>not</code> , NEGATION	<code>~</code>	<pre>&gt;&gt;&gt; ~5 #~x retourne -x - 1 -6</pre>

Exemples d'utilisation d'opérateurs bit à bit :

- On peut utiliser le ET bit à bit pour sélectionner uniquement certains bits, par exemple les bits de rang pairs :

```
>>> bits_pairs = sum(2 ** k for k in range(0, 8, 2))
>>> bin(bits_pairs)
'0b1010101'
>>> bin(183)
'0b10110111'
>>> bin(183 & bits_pairs)
'0b10100010'
```



- Le OU EXCLUSIF peut servir à masquer / démasquer une partie de la représentation binaire d'un nombre (on peut l'employer avec tout objet codé numériquement comme une image ou un caractère).

```
>>> diego = 69
>>> masque = 42
>>> zorro = diego ^ masque
>>> zorro
111
>>> zorro ^ masque
69
```



### Exercice 13

Dans un réseau IP l'adresse IP d'une machine est constitué d'un préfixe correspondant à l'adresse du réseau (commune à toutes les machines du réseau) et à un suffixe machine, identifiant la machine sur le réseau.

Le préfixe réseau s'obtient à partir de l'adresse IP de la machine en faisant un ET bit à bit avec le masque de sous-réseau.

Par exemple si l'adresse est 192.168.11.12 de représentation binaire 11000000.10101000.00001011.00001011 et le masque de sous-réseau est 255.255.252.0 de représentation binaire

11111111.11111111.11111100.00000000 alors le préfixe réseau est 11000000.10101000.00001000.00000000 soit 192.168.8.0.

On donne ci-dessous deux fonctions outils :

```
def ip2liste(ip):
    "Transforme une adresse IP V4 (type str) en liste d'entiers"
    return [int(champ) for champ in ip.split('.')]

def liste2ip(ipliste):
    "Transforme une liste d'entiers en adresse IP V4 (type str)"
    return '.'.join(str(n) for n in ipliste)
```

1. Écrire une fonction de signature `prefixe_reseau(ip, masque)` qui retourne le préfixe réseau sous forme d'adresse IP V4 (type `str`) à partir d'une adresse IP V4 et d'un masque de sous-réseau.
2. Écrire une fonction de signature `suffixe_machine(ip, masque)` qui retourne le suffixe machine sous forme d'adresse IP V4 (type `str`) à partir d'une adresse IP V4 et d'un masque de sous-réseau.

Voici un exemple de résultat attendu :

```
>>> prefixe_reseau('145.245.11.254', '255.255.252.0')
'145.245.8.0'
>>> suffixe_machine('145.245.11.254', '255.255.252.0')
```



#### Propriété 4

Python définit également des opérateurs sur les bits d'un nombre, plus efficaces que les opérations mathématiques équivalentes :

- Le décalage de `nombre` de `n` bits vers la gauche multiplie `nombre` par  $2^n$  et s'écrit `nombre << n`.
- Le décalage de `nombre` de `n` bits vers la droite divise `nombre` par  $2^n$  et s'écrit `nombre >> n`.



#### Exercice 14

Dans l'algorithme de recherche dichotomique, après division en deux de la zone de recherche, l'algorithme s'appelle lui-même sur l'une des deux moitiés. C'est un algorithme de type *Diviser pour régner* qui peut se programmer récursivement comme nous le verrons dans le chapitre sur la récursivité.

Si on note `n` la taille de la liste, une autre implémentation, non récursive, est la suivante :

- on commence la recherche au début de la liste et on avance avec un `pas = n // 2` ou `pas = n >> 1` jusqu'au premier élément supérieur à l'élément cherché ;
- on repart de l'élément précédent le point d'arrêt et on avance désormais avec un `pas = pas >> 1` ;
- on répète en boucle ces instructions jusqu'à ce que le `pas` atteigne 1.

A la fin de la boucle, on détermine si l'élément sur lequel on s'est arrêté est l'élément recherché.

Compléter le code de la fonction `recherche_dicho2` qui implémente cet algorithme.

```
def recherche_dicho2(L, e):
    x, n = 0, len(L)
    pas = n >> 1
    while pas >= 1:
        while x + pas < n and .....:
            x = .....
        pas = .....
    return .....
```