Booléens : opérateurs et portes logiques

Dernière mise à jour le : 14/08/2023

Introduction

En Python, comme dans les autres langages de programmation, une comparaison ou un test d'égalité est une expression qui peut être soit vraie, soit fausse. Les deux résultats possibles, vrai et faux, s'appellent des valeurs **booléennes**.

Ce terme tire son nom du mathématicien **Georges Boole** qui a défini, entre autre, une représentation des opérations logiques à l'aide d'opérations mathématiques. Cette modélisation s'appelle **l'algèbre de Boole** et elle est à la base du fonctionnement de tous les circuits électroniques, et donc des ordinateurs.

George Boole (1815-1864)

Crédit : Voir la page pour l'auteur, Domaine public, via Wikimedia Commons

Dans l'algèbre de Boole, il n'existe que deux éléments : 0 et 1. C'est pourquoi les valeurs booléennes sont également notées 0 (pour faux) et 1 (pour vrai) et on nous allons voir que l'on peut effectuer des opérations sur ces booléens.

Les ordinateurs, qui manipulent uniquement des bits 0 et 1, s'appuient sur la théorie de Boole pour effectuer des opérations logiques grâce à des **opérateurs booléens** qui sont réalisés par des circuits électroniques que l'on appelle des **portes logiques** (ou portes booléennes).

Nous allons d'abord évoquer les opérateurs booléens qui permettent de faire des opérations logiques sur les booléens, puis nous parlerons des circuits électroniques d'un ordinateur qui réalisent effectivement les opérations correspondantes.

Opérateurs booléens

> Un **opérateur booléen** (on dit aussi *fonction booléenne*) est une fonction mathématique prenant en entrée un ou plusieurs booléens et donnant en sortie un unique booléen.

Il existe différents opérateurs booléens mais nous allons nous concentrer principalement sur trois d'entre eux, à partir desquels on peut construire tous les autres : *non*, *ou* et *et*.

Les opérateurs *non*, *ou* et *et*

Comme les booléens sont en nombre fini, on peut représenter chaque opérateur booléen par ce qu'on appelle une **table de vérité**, qui n'est autre qu'un tableau dans lequel on indique *tous* les cas possibles d'entrée(s) et de sortie(s).

Opérateur *non* (négation)

C'est un opérateur qui transforme vrai en faux et inversement. Sa table de vérité est donc :

a	***non a***
0	1
1	0

Opérateur *ou* (disjonction)

L'opérateur ***ou*** s'applique à *deux* booléens, notés ici ***a*** et ***b***. On note ***a ou b*** le résultat de l'opération, et ***a ou b*** est vrai si et seulement si ***a*** est vrai ou ***b*** est vrai (ou si les deux sont vrais). Sa table de vérité est donc :

a	***b***	***a ou b***
0	0	0
1	0	1
0	1	1
1	1	1

Opérateur *et* (conjonction)

L'opérateur ***et*** s'applique à *deux* booléens ***a*** et ***b***. On note ***a et b*** le résultat de l'opération, et ***a et b*** est vrai si et seulement si ***a*** est vrai et ***b*** est vrai. Sa table de vérité est donc :

a	***b***	***a ou b***
0	0	0
1	0	0
0	1	0
1	1	1

Opérateur *xor* (= ou exclusif)

L'opérateur ***xor*** (pour exclusive or) signifie « ou exclusif ». Il possède *deux* booléens ***a*** et ***b*** en entrée et ***a xor b*** est vrai si et seulement si ***a*** est vrai et ***b*** est vrai, mais pas si les deux sont vrais. Sa table de vérité est donc :

a	***b***	***a xor b***
0	0	0
1	0	1
0	1	1
1	1	0

Cet opérateur est très utilisé en informatique également (par exemple pour chiffrer des messages ou pour déterminer si deux ordinateurs appartiennent au même sous-réseau ; ceci sera abordé en classe de Terminale).

Expressions booléennes

Définition

Une **expression booléenne** est une expression faisant intervenir des opérateurs booléens et des booléens.

Par exemple, si *a* et *b* sont des booléens, alors ***non(non a ou non b)*** est une expression booléenne. Si on veut connaître la table de vérité de cette expression booléenne, il suffit de la construire étape par étape :

a	***b***	***non a***	***non b***	***non a ou non b***	***non(non a ou non b)***
0	0	1	1	1	0
1	0	0	1	1	0
0	1	1	0	1	0
1	1	0	0	0	1

Explications :

- La colonne ***non a*** est complétée en prenant la négation des valeurs de ***a***
- Même principe pour la colonne ***non b***
- La colonne ***non a ou non b*** s'obtient en appliquant l'opérateur « ou » entre ***non a*** et ***non b***.
- Enfin la dernière colonne, celle que l'on souhaite, s'obtient en prenant la négation de la colonne précédente.

On se rend compte que la table de vérité de ***non(non a ou non b)*** est identique à celle de ***a et b***, cela veut dire que : ***non(non a ou non b) = a et b***

En particulier, cela signifie qu'il est possible d'exprimer l'opérateur ***et*** à partir des opérateurs ***non*** et ***ou***, et donc qu'en réalité on peut exprimer tous les opérateurs uniquement à partir des opérateurs ***non*** et ***ou***.

L'opérateur ***xor***, comme tous les autres, peut aussi être construit à partir des trois opérateurs précédents. En effet, on peut montrer que l'on a l'égalité suivante : ***a xor b = (a et non b) ou (non a et b)*** (sera fait en exercices).

En Python

Les booléens *Vrai* et *Faux* s'expriment respectivement en Python :

- par `True` et `False`
- ou par `1` et `0`

Les trois opérateurs ***non***, ***et*** et ***ou*** peuvent être utilisés en Python, respectivement avec : `not`, `and` et `or`. On peut évaluer sans problème des expressions booléennes en Python :

```
>>> not True
False
>>> not False
True
>>> not 1 == 0
True
>>> True or False
True
>>> 0 or 0
False
>>> 1 and 1
True
>>> True and False
False
```

On peut bien sûr combiner plusieurs opérateurs :

```
>>> a = 2
>>> b = 3
>>> a > 0 and a < b
True
>>> (True and False) or (False and False)
False
>>> (a > 0 and not True) or (not True and a < b)
False
>>> (1 or 0) and (0 and 1)
0
```

⚠ **Attention** : L'opérateur `and` est prioritaire sur `or`. Ce qui fait qu'il n'est pas nécessaire d'écrire certaines parenthèses même si *cela reste conseillé pour plus de clarté* :

```
>>> True or True and False  # équivaut à True or (True and False)...
True
>>> (True or True) and False
False
>>> True or (True and False)  # ... comme vous pouvez le constater
True
```

⚠ **Attention** : L'opérateur `not` est quant à lui prioritaire sur les autres opérations :

```
>>> not False and False   # équivaut à (not False) and False ...
False
>>> not (False and False)
True
>>> (not False) and False  # ... comme vous pouvez le constater
False
```

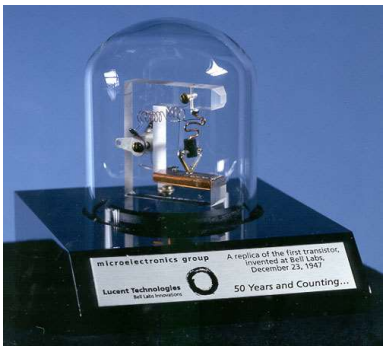
Transistors et portes logiques

> *Au commencement était le transistor,*
> *puis nous créâmes les portes booléennes*
> *et, à la fin de la journée, les ordinateurs.*

Concrètement, les ordinateurs sont capables d'effectuer des opérations logiques comme toutes celles que nous avons décrites. De plus, ils sont capables d'effectuer des opérations arithmétiques (additions, multiplications, etc.) également en utilisant des opérations logiques sur les bits. Nous allons voir comment dans cette dernière partie.

Transistors

C'est l'invention du **transistor** qui a permis de construire des circuits logiques qui permettent de réaliser les opérations booléennes décrites plus haut (*non*, *ou* , *et*, etc.). Il a été inventé en 1947 par les américains *John Bardeen*, *William Shockley* et *Walter Brattain*.



Réplique du premier transistor inventé en 1947

Crédit : Federal employee, Domaine public, via Wikimedia Commons

Quelques transistors

Crédit : ArnoldReinhold, CC BY-SA 3.0, via Wikimedia Commons

Les circuits électroniques d'un ordinateur sont en réalité composés d'un assemblage de transistors reliés entre eux. Ces circuits ne manipulent que des chiffres binaires : 0 correspond à une *tension basse* (proche de 0 volt) et 1 correspond à une *tension haute* (dont la valeur dépend du circuit).

> Aujourd'hui, les transistors sont directement gravés dans des plaques de silicium des circuits intégrés et connectés directement entre eux. Un processeur actuel compte des millions, voire des milliards de transistors.

Un transistor est un composant électronique très simple qui se comporte comme un interrupteur qui laisse ou non passer le courant électrique. Dans le schéma ci-dessous, on a représenté un transistor relié à un générateur E dont la tension est haute (+ V volts). En réalité, il y a aussi des résistances pour éviter les court-circuits mais elles n'ont pas été représentées pour simplifier.

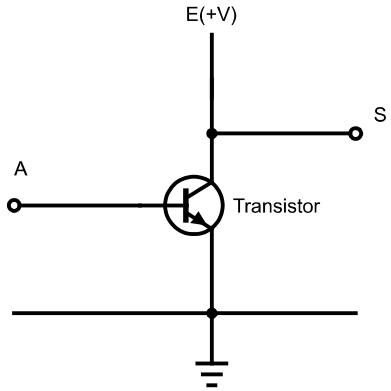
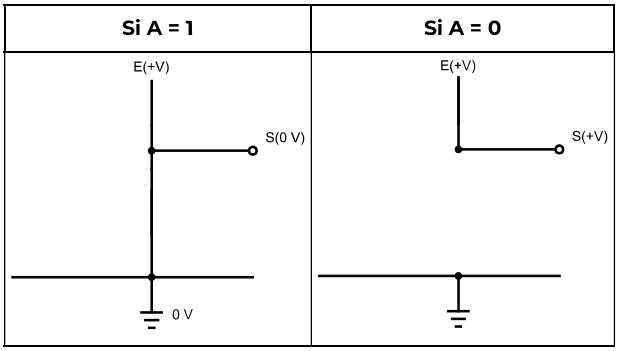


Schéma d'un transistor

Crédit : Modification personnelle du fichier créé par Pradana Aumars, CC0, via Wikimedia Commons

L'interrupteur est commandé par la broche A :

- si on applique une tension haute à l'entrée A, alors le transistor laisse passer le courant (l'interrupteur est fermé) et la sortie S est reliée à la masse, donc sa tension est basse (voir schéma 1 ci-dessous)
- en revanche, si on applique une tension basse à l'entrée A, alors le transistor bloque le courant (l'interrupteur est ouvert) et la sortie S est reliée à E (sous tension haute), donc elle reste sous tension haute (voir schéma 2 ci-dessous)



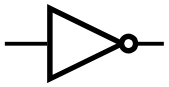
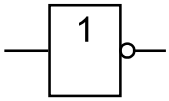
Portes logiques

En connectant de la bonne manière des transistors entre eux, on obtient des *circuits électroniques* qui permettent à une machine de réaliser concrètement les opérateurs booléens vus précédemment (*non*, *ou*, *et*, ...). Ces circuits électroniques classiques s'appellent des **portes logiques**.

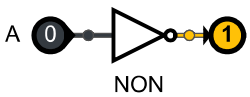
Chaque porte logique possède la même table de vérité que l'opérateur booléen qu'elle représente. On donne ci-dessous les symboles américains et européens des portes logiques (même si ce sont surtout les symboles américains qui sont utilisés).

Porte NON

Une **porte logique NON** est un circuit électronique dont la valeur en sortie est l'inverse de celle de son entrée. Cette porte peut être réalisée avec un simple transistor. En effet, comme nous l'avons vu avec le circuit précédent : si l'entrée A vaut 1 alors la sortie S vaut 0, et si l'entrée vaut 0 alors la sortie est égale à 1. Cela donne donc bien une porte NON.

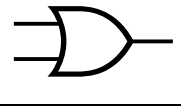
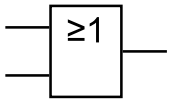
Symbole américain	Symbole européen
	

*Vérifiez, en cliquant sur l'entrée A, que la sortie correspond bien à la table de vérité de l'opérateur **non**.*

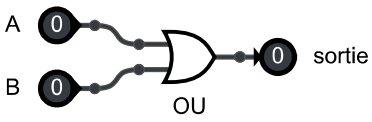


Porte OU

Une **porte logique OU** est un circuit électronique qui réalise l'opérateur ***ou*** : autrement dit un circuit possédant deux entrées A et B (deux tensions, chacune étant soit haute soit basse) et dont la tension en sortie est égale à ***A ou B***.

Symbole américain	Symbole européen
	

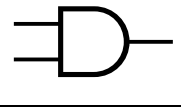
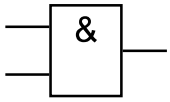
*Vérifiez, en cliquant sur les entrées A et B, que la sortie correspond bien à la table de vérité de l'opérateur **ou**.*



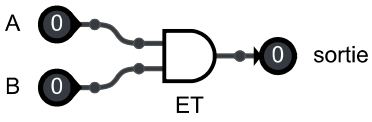
Porte ET

Une **porte logique ET** est un circuit électronique qui réalise l'opérateur ***et*** : autrement dit un circuit possédant deux entrées A et B (deux tensions, chacune étant soit haute soit basse) et dont la tension en sortie est égale à ***A et B***.

> La porte ET est un circuit électronique qui peut être construit à partir des portes OU et NON (sera fait en exercice).

Symbole américain	Symbole européen
	

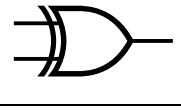
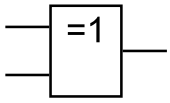
*Vérifiez, en cliquant sur les entrées A et B, que la sortie correspond bien à la table de vérité de l'opérateur **et**.*



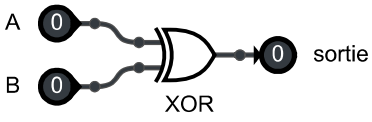
Porte XOR

Une **porte logique XOR** est un circuit électronique qui réalise l'opérateur ***xor*** : autrement dit un circuit possédant deux entrées A et B (deux tensions, chacune étant soit haute soit basse) et dont la tension en sortie est égale à ***A xor B***.

> La porte XOR est un circuit électronique qui peut être construit à partir des portes ET, OU et NON (sera fait en exercice).

Symbole américain	Symbole européen
	

*Vérifiez, en cliquant sur les entrées A et B, que la sortie correspond bien à la table de vérité de l'opérateur **xor**.*



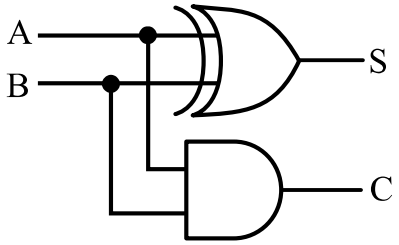
Addition binaire

Enfin, pour terminer nous allons voir qu'en combinant les portes logiques de la bonne façon on peut créer un circuit électronique qui permet de faire des additions.

En suivant le même principe, on peut créer d'autres circuits capables de soustraire, de multiplier, de diviser des nombres.

Demi-additionneur (1 bit)

Le circuit suivant, composé d'une porte **ET** et d'une porte **XOR**, est un *demi-additionneur* permettant d'additionner deux bits.



Demi-additionneur (1 bit)

Crédit : Cburnett, CC BY-SA 3.0, via Wikimedia Commons

Il prend deux bits A et B en entrée, et possède deux sorties :

- S qui est égale à la somme de A et B (on a ***S = A xor B***)
- C qui est la retenue (*carry*, en anglais) de cette somme (on a ***C = A et B***)

Sa table de vérité est la suivante :

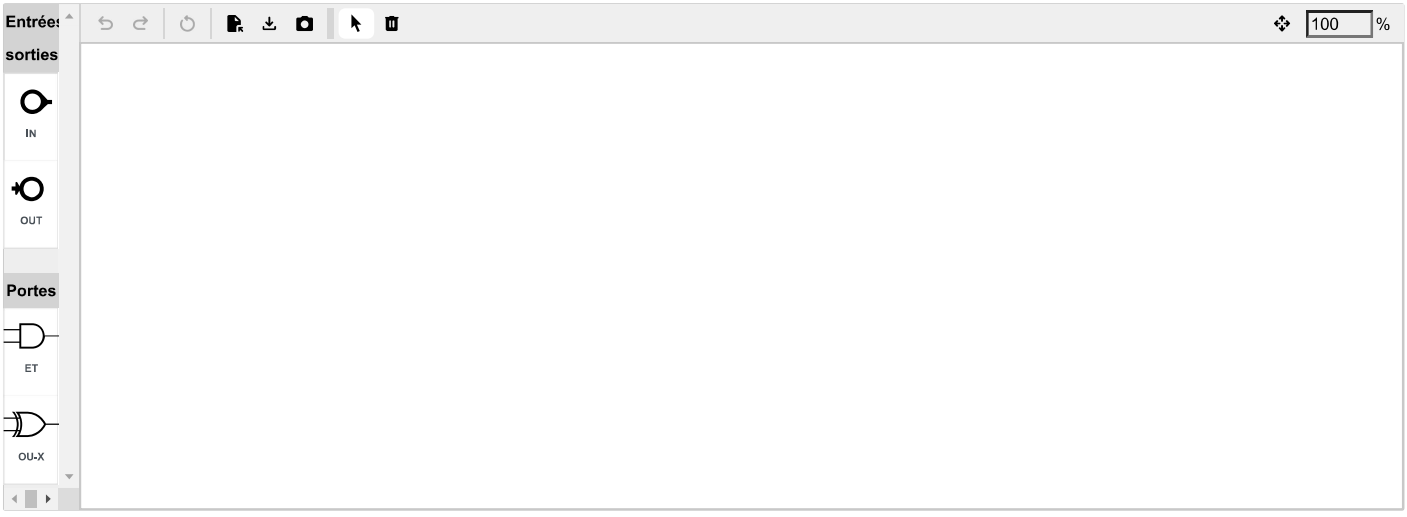
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Cela correspond bien à l'addition car :

- $(0)_2 + (0)_2 = (0)_2$ et il n'y a pas de retenue
- $(0)_2 + (1)_2 = (1)_2 + (0)_2 = (1)_2$ et il n'y a pas de retenue
- $(1)_2 + (1)_2 = (10)_2$ donc la somme (sur 1 bit !) fait 0 et on retient 1

À faire

Réalisez le circuit du demi-additionneur avec le simulateur ci-dessous, puis vérifiez la table de vérité donné ci-dessus en modifiant les valeurs des deux entrées. Pour ajouter les fils, il suffit de laisser cliquer d'une broche à une autre.

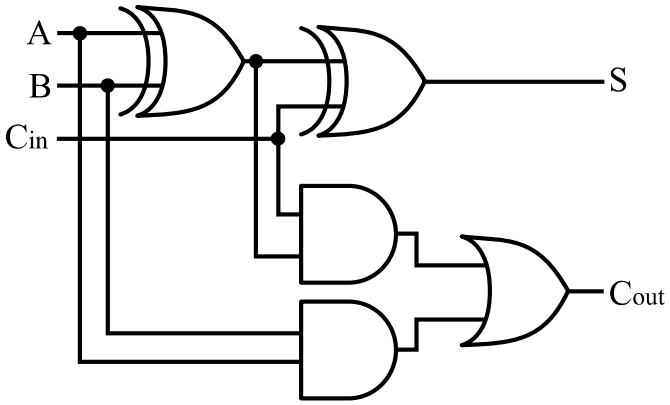


Additionneur complet

Pour obtenir un additionneur (sur 1 bit) complet, il faut une entrée supplémentaire : la retenue entrante (notée $\mathrm{C_{in}}$ par la suite).

Ce circuit logique, dont le schéma est donné ci-dessous, possède :

- trois entrées : les deux bits A et B ainsi que la retenue entrante C_{in} à additionner (celle résultant de la somme des deux bits précédents);
- et deux sorties : la valeur S de la somme (sur 1 bit) et la retenue sortante C_{out} (qui servira au calcul de la somme des deux bits suivants)



Additionneur complet (1 bit)

Crédit : Cburnett, CC BY-SA 3.0, via Wikimedia Commons

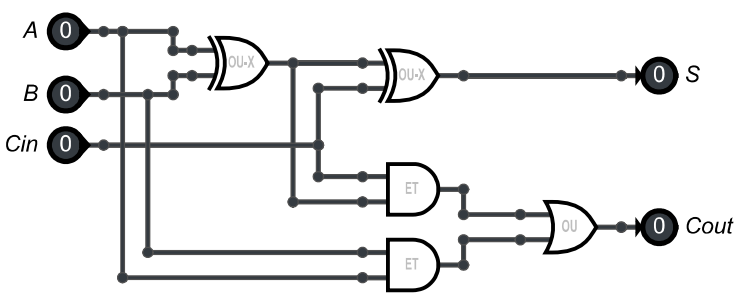
À faire

En utilisant le schéma de l'additionneur et les tables de vérité des portes logiques OU, ET et XOR, complétez la table de vérité de l'additionneur ci-dessous.

Vous pouvez vérifier avec le simulateur de circuits en-dessous.

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Table de vérité de l'additionneur



Réalisé avec le simulateur *logic.modulo-info.ch*.

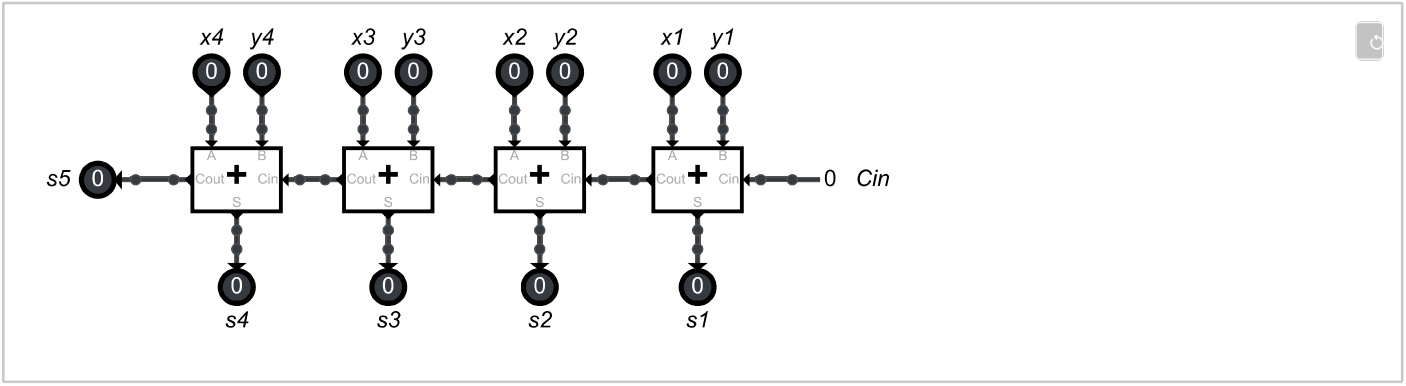
Ouvrir ce circuit avec le simulateur

En combinant des circuits additionneurs 1 bit de ce type, on peut créer des additionneurs capables d'additionner sur le nombre de bits que l'on souhaite. Par exemple, dans le schéma ci-dessous permet de créer un additionneur sur 4 bits, en combinant 4 additionneurs 1 bit.

Vous noterez qu'ici, plutôt que de reproduire quatre fois tout le circuit additionneur vu précédemment, on utilise le composant prédéfini du simulateur (où l'on retrouve les 3 entrées et les deux sorties).

Plus précisémenent, le circuit ci-dessous permet de faire l'addition de deux nombres $x = (x_4\, x_3\, x_2\, x_1)_2$ et $y = (y_4\, y_3\, y_2\, y_1)_2$ de 4 bits, le résultat de la somme étant égal à $s = (s_5\, s_4\, s_3\, s_2\, s_1)_2$.

Vous pouvez modifier les valeurs des bits d'entrée (en haut) pour vérifier que les sorties (en bas) correspondent bien à la somme. Par exemple, vérifiez que $(0110)_2 + (1011)_2$ *est bien égal à* $(10001)_2$ *en observant bien la retenue sortante qui devient la retenue entrante de l'additionneur suivant.*



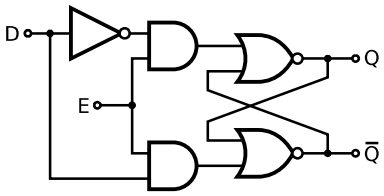
Réalisé avec le simulateur *logic.modulo-info.ch*.

Ouvrir ce circuit avec le simulateur

Les circuits *mémoires* (hors programme)

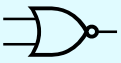
Ce n'est pas du tout au programme, mais sachez qu'avec les portes logiques, on peut aussi créer des circuits qui *stockent* des informations. Par exemple, un *verrou* est un circuit permettant mémoriser des informations. Autrement dit, c'est un circuit qui permet, dans certaines conditions, de maintenir la valeur de sa sortie malgré des changements de valeurs d'entrée.

En assemblant des verrous, on peut réaliser des *compteurs*, des *registres* ou encore des *mémoires*. Par exemple, le verrou ci-dessous permet de mémoriser 1 bit en mémoire, et il s'agit juste d'une combinaison de portes logiques :

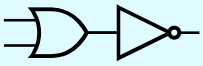


Un circuit *verrou-D*

Sur ce schéma, la porte

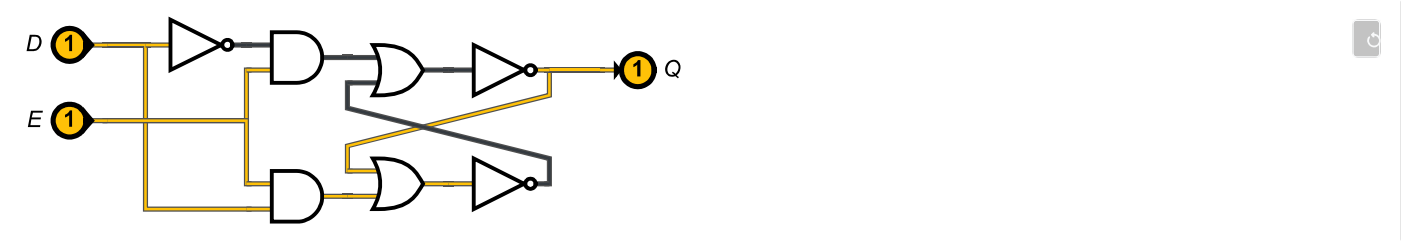


correspond à une porte logique NON OU (ou NOR en anglais, pour *Not OR*), qui n'est autre que la succession d'une porte OU puis d'une porte NON, donc équivalent à :



En pratique, lorsque l'entrée E est mise dans l'état 1, la valeur de l'entrée D (comme *data*) est mémorisée par le circuit et cette valeur peut être lue sur la sortie Q. Si on remet E à 0 et que l'on modifie la valeur de D, alors la sortie Q n'est pas modifiée, la valeur est donc bien mémorisée, jusqu'à ce que l'on provoque la mémorisation d'une nouvelle valeur en mettant E à 1.

Le circuit a été reproduit ci-dessous (sans la sortie $\overline{Q}$*, qui n'a pas d'importance dans notre cas), cela vous permet de tester et comprendre ce principe de mémorisation.*



Un circuit *verrou-D*

Réalisé avec le simulateur *logic.modulo-info.ch*

Ouvrir ce circuit avec le simulateur

> Pour en savoir plus sur ces notions, vous pouvez lire le chapitre 15 (page 182) du livre suivant : Informatique et Sciences du Numérique, Gilles Dowek.

Bilan

- Les **booléens** sont des variables qui ne possèdent que deux valeurs : vrai et faux.
- La théorie mathématique derrière l'utilisation et les calculs sur les booléens a été élaborée par George Boole (d'où le nom de ce type de variables).
- Des **opérateurs booléens** permettent de faire des opérations logiques sur les booléens : les trois opérateurs de base sont ***non*** (la négation), ***ou*** (la disjonction), ***et*** (la conjonction).
- Un opérateur booléen peut se décrire par sa **table de vérité** qui liste tous les résultats possibles en fonction des différentes combinaisons d'entrées.
- Un ordinateur est composé de **transistors**, qui sont des composants électroniques agissant comme des interrupteurs qui laissent ou non passer le courant. Ces deux états possibles se notent respectivement 1 et 0. Ainsi, un ordinateur peut manipuler des bits en s'appuyant sur la théorie développée par George Boole.
- En reliant des transistors de la bonne manière on peut créer des circuits appelés **portes logiques**. Ces portes permettent de réaliser les opérateurs booléens, et donc de réaliser des opérations logiques entre les bits.
- À partir de ces portes logiques, on peut créer des circuits plus complexes permettant d'effectuer des additions, des multiplications, ... L'*unité arithmétique et logique* d'un processeur contient ces différents circuits, absolument nécessaires au fonctionnement de l'ordinateur.
- (On peut même mémoriser des informations avec des portes logiques, et donc réaliser des composants mémoires (RAM, registres,etc.), ce qui montre bien que les portes logiques constituent les briques de base d'un ordinateur.)

La vidéo ci-dessous est un excellent résumé sur les **portes logiques** :

Lien vers la vidéo : https://youtu.be/iTH39L2d7bg

Références :

- Livre *Informatique et Sciences du Numérique, Terminale S*, de Gilles Dowek, éditions Eyrolles, disponible en version PDF.
- Cours et exercices de Romain Janvier sur les Circuits logiques et sur Python - Algèbre de Boole.
- Le simulateur logique en ligne https://logic.modulo-info.ch/ développé par Jean-Philippe Pellet.
- Article Wikipédia sur l'Additionneur.
- Livre *Numérique et Sciences Informatiques, Première NSI* de T. Balabonski, S. Conchon, J.-C. Filliâtre et K. Nguyen, éditions Ellipses. Site du livre : www.nsi-premiere.fr.

Germain Becker, Lycée Emmanuel Mounier, Angers.

Voir en ligne : info-mounier.fr/premiere_nsi/types_base/booleens.php