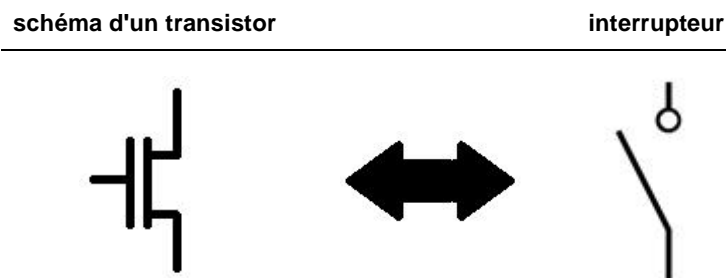


Un peu d'électronique : considération Hardware

Rappel : comme tous les constituants d'un ordinateur, le [processeur](https://fr.wikipedia.org/wiki/Processeur) (<https://fr.wikipedia.org/wiki/Processeur>) est fabriqué à base de [transistors](https://fr.wikipedia.org/wiki/Transistor) (<https://fr.wikipedia.org/wiki/Transistor>) qui sont des "nano-interrupteurs électroniques" ne pouvant prendre que 2 états : ouvert / fermé.



Chaque transistor peut "générer" une tension électrique de 0Volt ou 5Volt (par exemple) selon qu'il est ouvert ou fermé. Ces deux états possibles sont "informatiquement représentés" par :

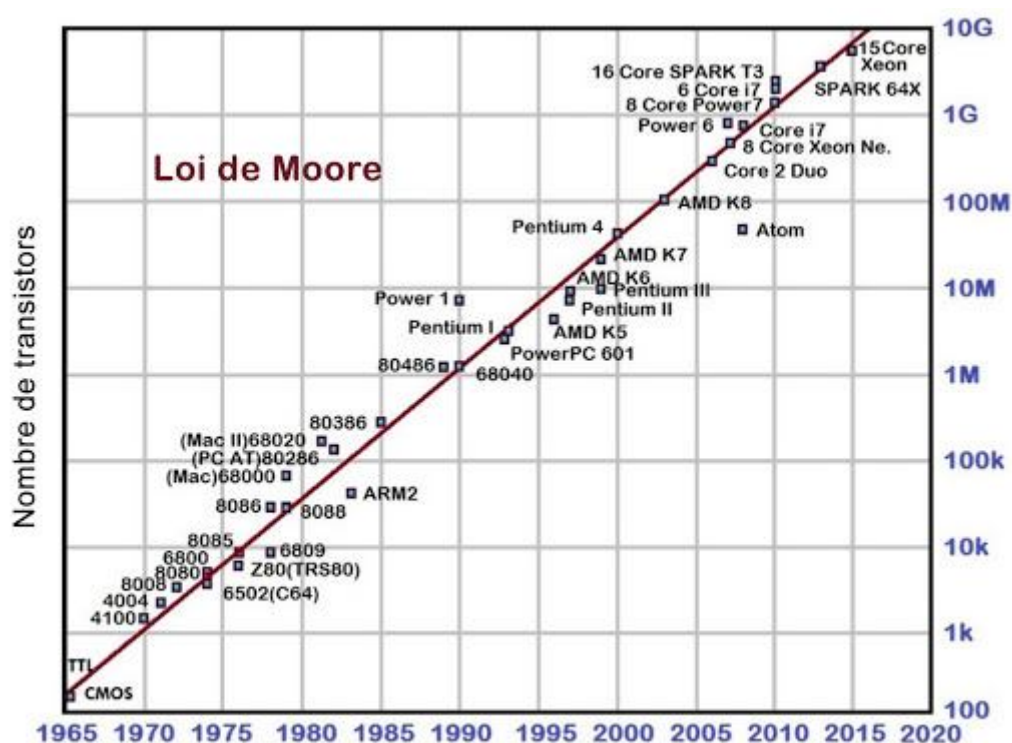
- le booléen `False` ou l'entier 0 correspondant à une tension de 0Volt.
- le booléen `True` ou l'entier 1 correspondant à une tension de 5Volt.

Pour information, le [Intel 4004](https://fr.wikipedia.org/wiki/Intel_4004) (https://fr.wikipedia.org/wiki/Intel_4004), premier processeur commercialisé de l'histoire en 1971 intégrait 2300 transistors sur sa puce. (Voir le [schéma interne du 4004](#) ([Intel-4004-schematics.pdf](#)))



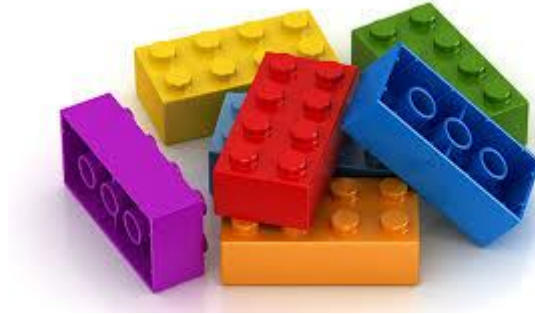
(Photo du 4004 - Source : wikipedia)

Avec les progrès de la microélectronique, les processeurs ont intégré de plus en plus de transistors, pour dépasser maintenant le milliard de transistors...



Comment peut-on assembler plus d'un milliard de transistors pour que cela fasse un processeur qui fonctionne ??

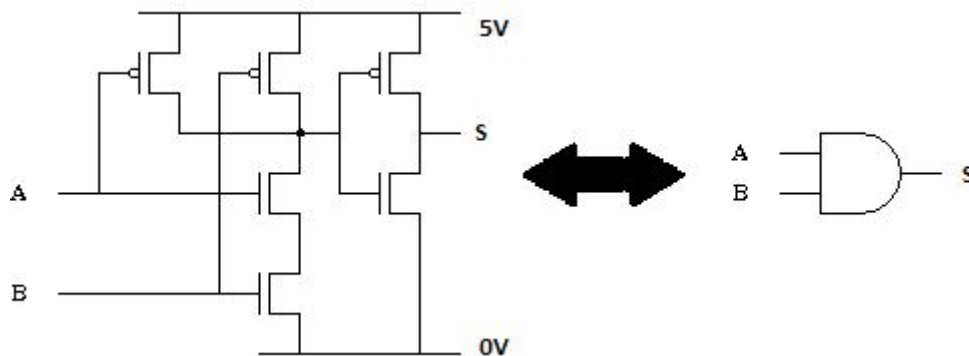
Le processeur : un "Lego" de transistors



On ne construit pas une structure de plus d'un milliard d'éléments "d'un seul coup" mais on construit des blocs de plus en plus évolués à partir de blocs plus simples et on répète l'opération. C'est finalement le même principe d'**encapsulation** et de **modularité** vu avec les **fonctions**

1. La pièce de base : Les transistors

1. En assemblant quelques transistors on fabrique des portes logiques ou opérateurs logiques (voir suite du cours)

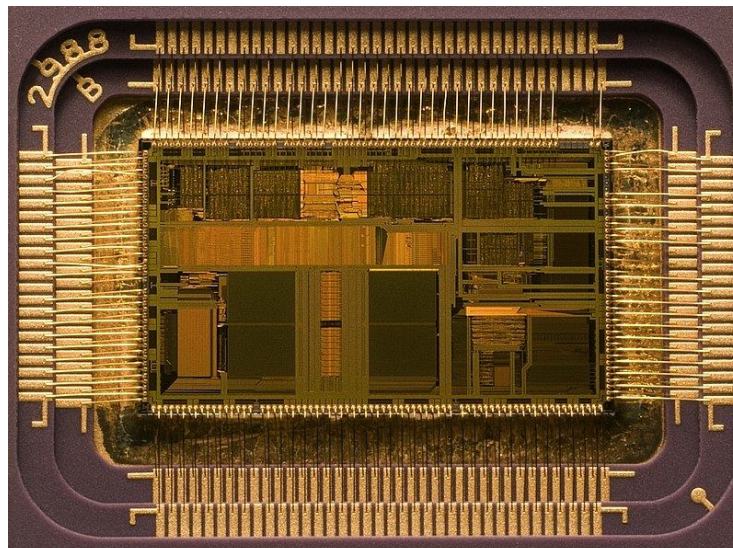


Exemple : Assemblage de 6 transistors pour faire une porte logique and

1. En assemblant quelques portes logiques on fabrique des additionneurs, comparateurs, registres, etc...

1. En assemblant ces structures plus complexes, on fabrique des Unités Arithmétique et Logique, des Unité de contrôle, etc... (voir cours sur l'architecture des ordinateurs)

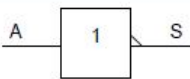
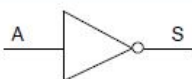

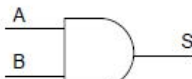
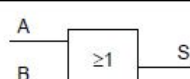

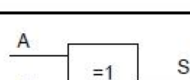







1. Un processeur est l'assemblage. Ce que l'on observe sur la photo ci-dessous :



(Photo - Wikipedia)

Les portes logiques

Les portes logiques sont des circuits électroniques permettant de réaliser des fonctions binaires élémentaires. Ces fonctions prennent 1 ou plusieurs bits en entrée et produisent 1 bit en sortie dont l'état (0 ou 1) ne dépend que de l'état des entrées (les fonctions sont dites combinatoires)

	Description	Symbole international	Symbole US	Table de Vérité	Equation															
not (non)	S est l'inverse de A			<table><tr><th>A</th><th>S</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	S	0	1	1	0	$S = \text{not}(A)$									
A	S																			
0	1																			
1	0																			
and (et)	Pour avoir $S = 1$ il faut $A=1$ ET $B=1$			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	S	0	0	0	0	1	0	1	0	0	1	1	1	$S = A \text{ and } B$
A	B	S																		
0	0	0																		
0	1	0																		
1	0	0																		
1	1	1																		
or (ou)	Pour avoir $S = 1$ il faut $A=1$ OU $B=1$			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	S	0	0	0	0	1	1	1	0	1	1	1	1	$S = A \text{ or } B$
A	B	S																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	1																		
xor (ou-exclusif)	Pour avoir $S = 1$ il faut $A=1$ OU BIEN $B=1$ (une seule entrée à 1)			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	S	0	0	0	0	1	1	1	0	1	1	1	0	$S = A \text{ xor } B$
A	B	S																		
0	0	0																		
0	1	1																		
1	0	1																		
1	1	0																		
nand (et-non)	On inverse la sortie du and			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	S	0	0	1	0	1	1	1	0	1	1	1	0	$S = \text{not}(A \text{ and } B)$
A	B	S																		
0	0	1																		
0	1	1																		
1	0	1																		
1	1	0																		
nor (ou-non)	On inverse la sortie du or			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	S	0	0	1	0	1	0	1	0	0	1	1	0	$S = \text{not}(A \text{ or } B)$
A	B	S																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	0																		
xnor (ou-exclusif-non)	On inverse la sortie du and			<table><tr><th>A</th><th>B</th><th>S</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	S	0	0	1	0	1	0	1	0	0	1	1	1	$S = \text{not}(A \text{ xor } B)$
A	B	S																		
0	0	1																		
0	1	0																		
1	0	0																		
1	1	1																		

Comment apprendre ce tableau ?

- Il suffit de retenir le nom des 4 premières portes logiques dites fondamentales (not, and, or, xor) ainsi que leurs symboles.
- Tout le reste du tableau doit se retrouver facilement à partir de ces 4 noms !!
- Si vous avez compris ce principe, alors ce sera exactement la même chose pour des portes à 3, 4 ou n entrées !!

Table de vérité

- Il y a $2^{nb \text{ d'entrées}}$ lignes à la table de vérité (voir [premiere/bloc1/ecriture_entier_positif/cours_representation_entier_positif.ipynb](#))
- Les lignes doivent être classées dans l'ordre croissant de leur conversion décimale et n'apparaître qu'une seule fois

Activité : Exercice 1 et 2 du TD

Utilisation des expressions booléennes

Dans une expression booléenne non parenthésée, le **and** est prioritaire sur le **or**. (Il est préférable tout de même de mettre des parenthèses pour lever toute ambiguïté).

Certains élèves se permettent de faire des manipulations qui change l'équation booléenne. Un conseil : lorsqu'on vous demande de calculer une expression booléenne, faites-le directement sans "transformer" l'équation...

Utilisation dans un test / if

```
In [1]: def oral_rattrapage(note_bac):  
        if (note_bac < 8) or (note_bac > 10):  
            print("Vous n'êtes pas convoqué à l'oral de rattrapage")  
        else :  
            print("Vous allez à l'oral de rattrapage")
```

```
In [2]: oral_rattrapage(10.2)
```

Vous n'êtes pas convoqué à l'oral de rattrapage

Utilisation dans des fonctions "prédicat"

Une fonction "prédicat" est une fonction renvoyant un résultat de type booléen. Le code d'une fonction prédicat peut donc s'écrire en une seule ligne : `return expression_boolenne`

Exemple : La fonction `est_liquide` est un prédicat

```
In [3]: def est_liquide(temperature):  
        """  
        Description de la fonction : Détermine s'il est possible que de l'eau soit  
        liquide  
        temperature (float) : température en degré centigrade.  
        return (bool)  
        """  
        return temperature >= 0 and temperature <= 100
```

```
In [4]: est_liquide(200)
```

Out[4]: False

```
In [5]: est_liquide(50)
```

Out[5]: True

Remarque : Certains seraient tenté d'écrire `return 0 <= temperature <= 100`.

Cette syntaxe est autorisée en python mais interdite dans la plupart des autres langages informatiques. Comme en NSI, le but est d'apprendre la programmation avant d'apprendre le langage python, **j'interdis cette syntaxe**. D'autant plus qu'elle peut ouvrir la porte à des écritures franchement bizarres, et qu'elle n'aide vraiment pas à comprendre comment sont évaluées les instructions par un ordinateur. Plus d'info [ici \(https://docs.python.org/fr/3/reference/expressions.html#comparisons\)](https://docs.python.org/fr/3/reference/expressions.html#comparisons)

Lorsqu'on débute en programmation, écrire un `return` suivi d'une expression booléenne n'est souvent pas naturel. Souvent c'est parce que la "nature booléenne" des opérateurs `==`, `!=`, `>`, `<`, `>=`, `<=` n'a pas été assimilée. Les débutants leur préfèrent des structures conditionnelles (`if`) qui certes, fonctionnent mais sont inutiles et indigestes... comme ci dessous :

```
In [ ]: # Version 2 : pas top !
def est_liquide(temperature):
    """
    Description de la fonction : Détermine s'il est possible que de l'eau soit
    liquide
    temperature (float) : température en degré centigrade.
    return (bool)
    """
    if temperature >= 0 and temperature <= 100 :
        return True
    else:
        return False
```

```
In [ ]: # Version 3 : vraiment pas top !
def est_liquide(temperature):
    """
    Description de la fonction : Détermine s'il est possible que de l'eau soit
    liquide
    temperature (float) : température en degré centigrade.
    return (bool)
    """
    if temperature >= 0 :
        if temperature <= 100 :
            return True
        else:
            return False
    else:
        return False
```

Séquentialité des opérateurs `and` et `or`

En Python, les opérateurs `and` et `or` sont séquentiels : la partie gauche est évaluée en premier, et la partie droite n'est évaluée ensuite que si c'est nécessaire.

- `expression_gauche and expression_droite` : `expression_gauche` est évaluée, et si elle vaut `False`, il n'est pas nécessaire d'évaluer `expression_droite` (en effet, `False and` n'importe quoi vaut toujours `False`).
- `expression_gauche or expression_droite` : `expression_gauche` est évaluée, et si elle vaut `True`, il n'est pas nécessaire d'évaluer `expression_droite` (en effet, `True or` n'importe quoi vaut toujours `True`).

Exemple : Lorsque `a = 0`, `1/a` n'a aucun sens. Ce que l'on observe dans le premier cas.

Mais dans le deuxième cas, `1/a` n'est même pas évalué (car c'est inutile) : aucune erreur n'est générée !!

En effet, avec `a = 0`, l'expression `(a != 0)` est donc `False`, et donc `(a != 0) and quelquechose` sera nécessairement `False`. Le "quelquechose"; c'est-à-dire `1/a` n'a donc pas besoin d'être calculé.

```
In [6]: a = 0

(1/a) and (a != 0) > 0
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-6-14238006af52> in <module>
      1 a = 0
      2
----> 3 (1/a) and (a != 0) > 0

ZeroDivisionError: division by zero
```

```
In [7]: a = 0

(a != 0) and (1/a) > 0
```

Out[7]: False

Quelques compléments

Autre notation utilisée

Il existe d'autres symboles mathématiques pour noter les expressions booléennes :

not(a)	\bar{a}	$\neg a$
a and b	$a.b$	$a \wedge b$
a or b	$a + b$	$a \vee b$
a xor b	$a \oplus b$	

Exemple $S = A \text{ and } (B \text{ or not}(C)) = A.(B + \bar{C}) = A \wedge (B \vee \neg C)$

Opérateurs binaires en python

Ces opérateurs s'effectuent entre objets s'écrivant sur plusieurs bits (exemple : variable de type entier)

- >> : décalage à droite
- << : décalage à gauche
- & : ET bit-à-bit
- | : OU bit-à-bit
- ^ : XOR bit-à-bit
- ~ : NOT bit-à-bit (en codage complément à 2)

```
In [8]: # on décale le nombre 10110 (exprimé en base 2) à droite de 2 rangs
resultat = 0b10100 >> 2

# On affiche le résultat en binaire
bin(resultat)
```

Out[8]: '0b101'


```
In [9]: # la même chose en exprimant les nombres en base 10
20 >> 2
```

Out[9]: 5

```
In [10]: # décalage à gauche de 2 rangs
bin(0b10100 << 2)
```

Out[10]: '0b1010000'

```
In [11]: # la même chose en exprimant les nombres en base 10
20 << 2
```

Out[11]: 80

```
In [12]: # Décaler un nombre d'un rang à gauche revient à le multiplier par 2

a = 53
(a * 2) == (a << 1)
```

Out[12]: True

```
In [13]: # Décaler un nombre d'un rang à droite revient à le diviser par 2 (division e
ntière !!)

a = 53
(a // 2) == (a >> 1)
```

Out[13]: True

```
In [14]: # L'opérateur ET est très utile pour isoler un "paquet de bit"
# Exemple d'un pixel dont la couleur est codée en RVB

couleur = 0b10000000_00001110_10101010
```

```
In [15]: r = couleur & 0b11111111_00000000_00000000
rouge = r >> 16
print(rouge)
```

128