**LOGIC GATES**

Georges **BOOLE** 1847 remplace les raisonnements logiques par des calculs sur deux éléments :

False etTrue ou bien 0 et 1. C'est l'algèbre de BOOLE.

Claude **SHANNON** en 1938 prouve que des circuits électriques peuvent résoudre tous les problèmes posés dans l'algèbre de Boole.

**Alan TURING** en 1936 propose un modèle qui utilise ces concepts pour résoudre des problèmes complexes : la machine de Turing.

**Opérations fondamentales** : négation NOT a non a conjonction & AND a et b disjonction | OR a ou b

**Tables de vérité**

Construire une porte **Nand**

|  |  |  |
| --- | --- | --- |
| a | b | A Nand b |
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Construire une porte **Xor** : si a=b alors retourne 0 sinon retourne 1

|  |  |  |
| --- | --- | --- |
| a | b | A Xor b |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Construire un **semi additionneur** 1 bit qui prend en entrée deux bits a et b et qui renvoie leur somme s= a+b et la retenue c

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | s | c |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Construire un **additionneur** 1 bit qui prend en entrée (in) deux nombres a et b et aussi la retenue précédente c et qui calcule la somme s=a+b+c et la retenue finale c1 (out)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | b | c | s | c1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Construire un **multiplexor** 1 bit qui prend trois entrées a, b, sel (select) et qui a une unique sortie out : si sel=0 alors out=a sinon out=b

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | sel | out |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Un multiplexor permet de n'utiliser qu'un seul bus pour plusieurs entrées. Il est très largement répandu.

|  |  |  |  |
| --- | --- | --- | --- |
| in | sel | a | b |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |

Construire un **demultiplexor** 1 bit qui prend deux entrées in et sel et a deux sorties a et b :

Le demultiplexor est le mécanisme inverse du précédent : à la sortie il rétablit les données.

Construire des fonctions And8, Or8, Not8, Nand8, Xor8, Add8 qui prennent en entrée des tableaux a et b de 8 bits.

**Correction**

# Créé par prof, le 28/11/2020

from \_\_future\_\_ import division

from lycee import \*

def Nand(a,b):

c= not(a and b)

return(c) #fonction logique essentielle qui permet de construire toutes les autres:

#Nand(a,a)=Not(a) Nand(Nand(a,a),Nand(b,b))=a Or b

def Xor(a,b):

if a==b: #visualiser cette condition simple

return(0)

else:

return(1)

def Halfadder(a,b):

s=Xor(a,b)

c=(a and b)

return(s,c) # on récupère la somme et la retenue

def Adder(a,b,c): #on retrouve les tables de vérité de Xor

s=Xor(a,b) #on calcule en deux fois

s=Xor(s,c)

c1=(a and b) or (a and c) or (b and c)

return(s,c1) #on peut calculer cette valeur par étapes

def Multiplexor(a,b,sel):

if sel==0:

return(a)

else:

return(b)

def Demultiplexor(In,sel):

a=0

b=0

if sel==0:

a=In

else:

b=In

return(a,b)

def And8(a,b):

c=[0]\*8

for k in range(8):

c[k]=a[k] and b[k]

return(c)

def Not8(a):

c=[0]\*8

for k in rang(8):

c[k]=not(a[k])

return(c)

def Or8(a,b):

c=[0]\*8

for k in rang(8):

c[k]=(a[k] or b[k])

return(c)

def Nand8(a,b):

c=[0]\*8

for k in range(8):

c[k]=Nand(a[k],b[k])

return(c)

def Xor8(a,b):

c=[0]\*8

for k in range(8):

c[k]=Xor(a[k],b[k])

return(c)

def Add8(a,b):

s8=[0]\*8 #addition commençant à gauche

s8[0]=Halfadder(a[0],b[0])[0]

c=Halfadder(a[0],b[0])[1]

for k in range(1,8):

s8[k]=Adder(a[k],b[k],c)[0]

c=Adder(a[k],b[k],c)[1]

return(s8)

def Adder8(a,b): #fonction incorrecte la retenue n'est pas prise

s=[0]\*8 #en compte

c=[0]\*8 #addition de gauche à droite

for k in range(8): # pour rétablir la disposition naturelle

s[k]=Xor(a[k],b[k]) # on change k en 7-k

c[k]= a[k] and b[k]

return(s)

# TESTS

print(Xor(1,0))

print(Xor(1,1))

print(Halfadder(1,0))

print(Halfadder(1,1))

print(Adder(1,0,1))

print(Adder(1,1,1))

print(Multiplexor(0,1,0))

print(Demultiplexor(1,1))

a=[1,1,0,0,1,1,0,1]

b=[0,1,1,1,0,0,0,1]

print(Add8(a,b))

print(Adder8(a,b))

a=[1,1,0,0,1,1,1,1]

b=[0,1,1,1,0,1,1,1]

print(Add8(a,b))

print(Adder8(a,b))

**LOGIQUE ALGEBRE DE BOOLE PROGRAMMATION EN PYTHON.**

**Programmer** les quatre fonctions suivantes :

a et b sont deux variables qui prennent des valeurs booléennes,

c’est à dire Vrai (1) ou Faux (0)

def Prem(a,b) : non a et b

def Second(a,b) : non a et non b

def Tierce(a,b) : a et non b

def Quarte(a,b) : non(a ou b)

**Tester** chaque fonction par tous les cas possibles et

noter les résultats :

Faux-Faux Vrai-Faux Faux-Vrai et Vrai-Vrai

**Constater** que deux fonctions produisent

les mêmes résultats pour les mêmes données..

Quelle égalité entre deux expressions logiques peut-on écrire ?

Noter cette égalité avec **les symboles**:

~a&b a&~b ~a&~b ~(a|b)

C’est une des **égalités de Morgan.**

Déduire une autre égalité de Morgan.