Booléens et addition binaire

Doc. 4,4

Objectif

Dresser la table de vérité d'une expression booléenne, découvrir le ou exclusif, et s'en servir pour programmer l'addition binaire de deux octets, comme le fait un circuit du processeur.

Q1. Un processeur additionne deux bits a et b. Combien de cas différents peut-il rencontrer ? Pour chacun, écrire le résultat de $a + b$ en base 2. Combien de bits faut-il pour écrire ce résultat ?

Réponse : Il y a quatre cas : $0+0=0$, $0+1=1$, $1+0=1$ et $1+1=(10)_2$. Le dernier résultat demande deux bits : un chiffre des unités, appelé **somme**, et une **retenue**. Un circuit d'addition doit donc produire deux bits de sortie à partir de deux bits d'entrée.

§1. Tables de vérité

Un bit et un booléen sont deux façons de voir la même information : 1 correspond à `True` (vrai) et 0 à `False` (faux). En Python, `int(True)` vaut `1` et `bool(0)` vaut `False`.

Les opérateurs `and`, `or` et `not` ont été rencontrés dans le Doc. 2,3. Une expression booléenne qui dépend de deux variables a et b ne peut prendre que quatre combinaisons de valeurs d'entrée. On peut donc décrire complètement son comportement dans un tableau qui donne sa valeur pour chaque combinaison : sa **table de vérité**.

Exemple résolu : table de vérité de « a and b »

On écrit toutes les combinaisons des entrées, dans l'ordre où l'on compte en binaire (00, 01, 10, 11), puis la valeur de l'expression :

a	b	`a and b`
0	0	0
0	1	0
1	0	0
1	1	1

`a and b` n'est vrai que si les deux entrées sont vraies.

Python peut produire ces tables à l'aide de deux boucles imbriquées :

```python
for a in (False, True):
    for b in (False, True):
        print(int(a), int(b), int(a and b))
```

Q2.
1. Dresser à la main la table de vérité de `a or b` et celle de `not a`.
2. Vérifier la table de `a or b` en modifiant le programme ci-dessus.

Réponse :

1. Tables de `a or b` et de `not a` :

a	b	`a or b`
0	0	0
0	1	1
1	0	1
1	1	1

a	`not a`
0	1
1	0

`a or b` est vrai dès qu'au moins une des entrées est vraie.

2. Il suffit de remplacer `a and b` par `a or b` dans le `print`.

Q3. Dresser les tables de vérité de `not (a and b)` et de `(not a) or (not b)`. Que constate-t-on ? Reformuler ce résultat par une phrase en français, en partant de : « Il est faux que a et b soient tous les deux vrais… ».

Réponse :

a	b	`not (a and b)`	`(not a) or (not b)`
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Les deux tables sont identiques : les deux expressions sont équivalentes. « Il est faux que a et b soient tous les deux vrais » signifie « au moins l'un des deux est faux ». Cette équivalence est une des lois de De Morgan. Erreur fréquente : croire que `not (a and b)` équivaut à `(not a) and (not b)`, qui ne vaut 1 que sur la première ligne.

§2. Le ou exclusif

Dans la langue courante, « ou » a deux sens. « Fromage ou dessert » au restaurant signifie qu'on ne peut pas avoir les deux. L'opérateur `or` de Python, lui, est vrai aussi quand les deux entrées sont vraies : c'est un **ou inclusif**.

Le **ou exclusif**, noté **xor** (de l'anglais *exclusive or*), est vrai lorsqu'une seule des deux entrées est vraie, mais pas les deux.

Q4.
1. Dresser la table de vérité du ou exclusif.
2. Pour deux booléens, le ou exclusif peut s'écrire `a != b`. Expliquer pourquoi.
3. Écrire une expression qui donne le ou exclusif en n'utilisant que `and`, `or` et `not`. Vérifier sa table de vérité avec Python.

Coup de pouce

Traduire mot à mot : « a ou b est vrai, et il est faux que a et b soient vrais tous les deux ».

Réponse :

1.

a	b	a xor b

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

2. Le ou exclusif est vrai exactement quand les deux entrées sont différentes, c'est-à-dire quand `a != b`.
3. `(a or b) and not (a and b)`. On peut aussi écrire `(a and not b) or (not a and b)`. On vérifie avec :

```python
for a in (False, True):
    for b in (False, True):
        print(int(a), int(b), int((a or b) and not (a and b)))
```

En Python, l'opérateur `^` calcule le ou exclusif de deux entiers égaux à 0 ou 1 : `1 ^ 1` vaut `0` et `1 ^ 0` vaut `1`.

§3. Additionner deux bits avec des opérateurs booléens

Q5. Compléter la table de l'addition de deux bits a et b, en séparant le résultat en deux bits : la somme (chiffre des unités) et la retenue. Comparer les deux colonnes de sortie aux tables de vérité déjà rencontrées.

a	b	**somme**	**retenue**
0	0		
0	1		
1	0		
1	1		

Réponse :

a	b	**somme**	**retenue**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

La colonne « somme » est la table du ou exclusif, et la colonne « retenue » est celle de `and` : somme $=$ a xor b et retenue $=$ a and b. Un circuit qui réalise ces deux opérations s'appelle un **demi-additionneur**.

Pour additionner deux nombres de plusieurs bits, chaque colonne reçoit en plus la retenue r venue de la colonne de droite. Il faut donc additionner **trois** bits. On peut démontrer que :

— la somme vaut a xor b xor r ;
— la retenue sortante vaut 1 lorsqu'au moins deux des trois bits valent 1.

Q6. Vérifier ces deux règles sur la ligne $a = 1, b = 1, r = 1$, puis sur la ligne $a = 1, b = 0, r = 1$, en comparant avec le calcul $a + b + r$ écrit en base 2.

Réponse :

- $1+1+1=3=(11)_2$: somme 1, retenue 1. Règles : 1 xor 1 = 0, puis 0 xor 1 = 1 ; trois bits valent 1, donc retenue 1. C'est bien le cas.
- $1+0+1=2=(10)_2$: somme 0, retenue 1. Règles : 1 xor 0 = 1, puis 1 xor 1 = 0 ; deux bits valent 1, donc retenue 1. C'est bien le cas.

Q7. Écrire la fonction `additionne_bits`, qui retourne un tuple :

```python
def additionne_bits(a: int, b: int, r: int) -> tuple[int, int]:
    """
    Retourne la somme et la retenue de l'addition des trois
    bits a, b et r (chacun égal à 0 ou à 1).

    >>> additionne_bits(0, 1, 0)
    (1, 0)
    >>> additionne_bits(1, 1, 0)
    (0, 1)
    >>> additionne_bits(1, 1, 1)
    (1, 1)
    """
```

Réponse :

```python
def additionne_bits(a: int, b: int, r: int) -> tuple[int, int]:
    somme = a ^ b ^ r
    if a + b + r >= 2:
        retenue = 1
    else:
        retenue = 0
    return (somme, retenue)
```

On accepte aussi `somme = (a + b + r) % 2` et `retenue = (a + b + r) // 2` : ce sont le chiffre des unités et le quotient de la division par 2, comme dans la méthode des divisions successives.

§4. Additionner deux octets

On additionne deux octets comme on pose l'addition à la main (Doc. 4,2) : de la droite vers la gauche, en reportant la retenue d'une colonne sur la suivante. La retenue qui sort de la colonne de gauche est perdue, puisque le résultat est rangé sur 8 bits.

Pour parcourir les indices de droite à gauche, on utilise `range` avec trois arguments : `range(7, -1, -1)` produit 7, 6, 5, ..., 0.

Q8. Écrire la fonction `addition`, qui doit utiliser `additionne_bits` :

```python
def addition(bits1: str, bits2: str) -> str:
    """
    Retourne la somme sur 8 bits des octets bits1 et bits2.
```

```
    La retenue finale est perdue.


    >>> addition("00000110", "00000111")
    '00001101'
    >>> addition("11111111", "00000001")
    '00000000'
    """
```

Coup de pouce

1. Avant la boucle : un résultat vide et une retenue nulle.
2. Dans la boucle, pour l'indice `i` : convertir `bits1[i]` et `bits2[i]` en entiers, appeler `additionne_bits`, déballer le tuple obtenu.
3. Chaque nouveau chiffre se place à gauche du résultat déjà construit.

Réponse :

```python
def addition(bits1: str, bits2: str) -> str:
    resultat = ""
    retenue = 0
    for i in range(7, -1, -1):
        somme, retenue = additionne_bits(int(bits1[i]), int(bits2[i]), retenue)
        resultat = str(somme) + resultat
    return resultat
```

Le déballage `somme, retenue = ...` met à jour la retenue, qui servira à la colonne suivante. À la fin de la boucle, la dernière retenue n'est écrite nulle part : c'est le dépassement de capacité du Doc. 4,2. Erreur fréquente : écrire `resultat + str(somme)`, qui donne le résultat à l'envers.

Q9. Dans le Doc. 4,3, 5 s'écrit 00000101 et -5 s'écrit 11111011 sur 8 bits.

1. Prévoir le résultat de `addition("00000101", "11111011")`, puis vérifier.
2. Calculer `addition("00000011", "11111011")`. Quel entier relatif ce résultat représente-t-il ? Le calcul est-il juste ?

Réponse :

1. Le résultat est `'00000000'` : la retenue finale est perdue, et $5 + (-5) = 0$.
2. Le résultat est `'11111110'`. Il commence par 1 : complément à 1 `00000001`, plus 1 : `00000010`, donc il représente -2. Et $3 + (-5) = -2$: la même fonction additionne correctement les entiers positifs et les entiers négatifs écrits en complément à 2. C'est la raison pour laquelle les ordinateurs utilisent cette représentation.

Q10. Recopier la fonction `complement_a_1` du Doc. 4,3, puis écrire en une ligne la fonction `complement_a_2` :

```python
def complement_a_2(bits: str) -> str:
    """
    Retourne le complément à 2 de l'octet bits.
```

```
    >>> complement_a_2("00000101")
    '11111011'
    >>> complement_a_2("11111011")
    '00000101'
    """
```

Réponse :

```python
def complement_a_1(bits: str) -> str:
    resultat = ""
    for bit in bits:
        if bit == "0":
            resultat = resultat + "1"
        else:
            resultat = resultat + "0"
    return resultat

def complement_a_2(bits: str) -> str:
    return addition(complement_a_1(bits), "00000001")
```

Le deuxième exemple montre que le complément à 2 du complément à 2 redonne l'octet de départ : l'opposé de -5 est bien 5.

§5. Un détail qui compte : l'évaluation séquentielle

Dans le Doc. 2,3, on a vu que Python évalue `a and b` de gauche à droite et s'arrête dès que le résultat est connu : si `a` est faux, `b` n'est même pas évalué. De même, si `a` est vrai, `a or b` est vrai sans que `b` soit évalué. La table de vérité ne dit rien de cet ordre, mais il change le comportement d'un programme.

Q11. On exécute `x = 0` dans la console. Prévoir le résultat de chacune des deux expressions, puis vérifier :

```
>>> x != 0 and 10 / x > 1
>>> 10 / x > 1 and x != 0
```

D'après leurs tables de vérité, ces deux expressions devraient être équivalentes. Pourquoi ne le sont-elles pas dans un programme ?

Réponse :

```
>>> x != 0 and 10 / x > 1
False
>>> 10 / x > 1 and x != 0
ZeroDivisionError: division by zero
```

Dans la première expression, `x != 0` est faux : Python n'évalue pas `10 / x` et retourne `False`. Dans la seconde, Python commence par `10 / x` et la division par zéro provoque une erreur avant que la condition `x != 0` soit

examinée. Les tables de vérité décrivent les valeurs, pas l'ordre des calculs : on place en premier la condition qui protège la seconde.

§6. Bilan

À l'issue de cette activité, je sais…

- associer les bits 0 et 1 aux booléens `False` et `True` ;
- dresser la table de vérité d'une expression booléenne, à la main et avec deux boucles imbriquées ;
- reconnaître deux expressions équivalentes en comparant leurs tables de vérité ;
- définir le ou exclusif et l'écrire avec `!=`, `^` ou les opérateurs `and`, `or` et `not` ;
- expliquer que la somme de deux bits est un ou exclusif et la retenue un `and` ;
- programmer l'addition de deux octets en propageant la retenue, et constater qu'elle fonctionne aussi en complément à 2 ;
- expliquer pourquoi l'ordre des opérandes de `and` et de `or` peut éviter une erreur.

Dans le Doc. 4,5, on quitte les entiers pour les nombres à virgule, et on découvre pourquoi `0.1 + 0.2` n'est pas égal à `0.3`.