M3.L1: Série d’exercices sur l’architecture de l’ordinateur [solution]

Rappel : Soit x une variable logique, la notation $\bar{x}$ signifie la négation de x : si x=0 alors $\bar{x}$ =1 et vice versa.

1. Introduction aux portes logiques

1.1 La table de vérité du XOR présente deux combinaisons des entrées pour lesquelles ce circuit produit 1.

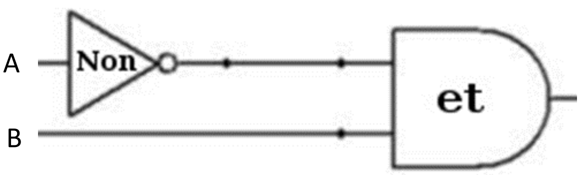
Il s’agit de :

« A valant 0 » avec « B valant 1 »

« A valant 1 » avec « B valant 0 »

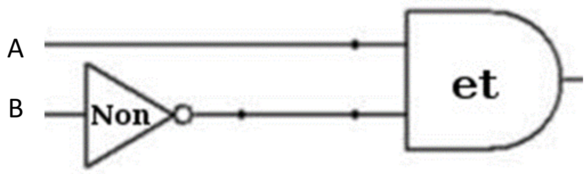
Chaque combinaison peut être mise en œuvre avec une porte logique AND qui traduit le « avec » ci-dessus.
Pour les cas où une variable est utile lorsqu’elle vaut 0, il suffit de prendre sa négation avant de l’envoyer sur l’entrée de la porte AND. On obtient :

<table>
<tr>
<td>
1.1.1) « A valant 0 » avec « B valant 1 »



Ce circuit produit 1 SEULEMENT si :
A vaut 0 et B vaut 1

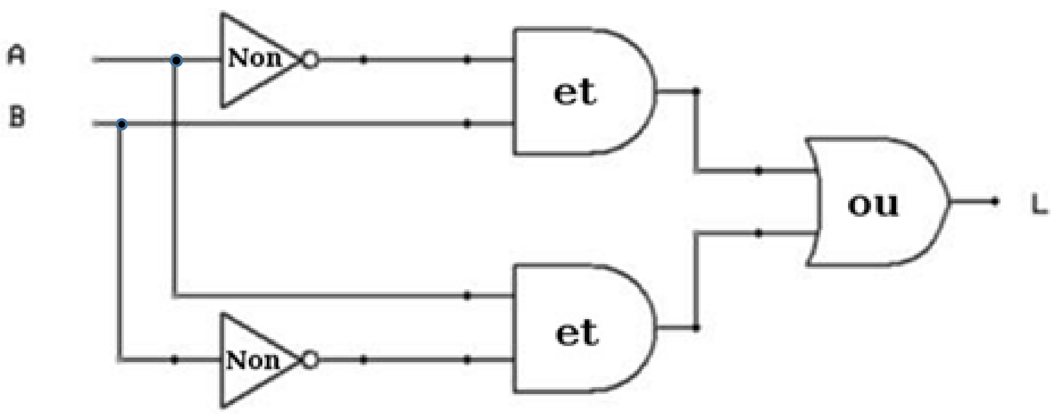
L’équation logique de sa sortie S est :
S = $\bar{A}$ et B
</td>
<td>
1.1.2) « A valant 1 » avec « B valant 0 »



Ce circuit produit 1 SEULEMENT si :
A vaut 1 et B vaut 0

L’équation logique de sa sortie S est :
S = A et $\bar{B}$
</td>
</tr>
</table>

Il suffit ensuite d’assembler ces deux circuits avec une porte OR pour obtenir tous les cas pour lesquels la porte XOR produit 1 :



1.2 Expression logique et table de vérité complète :

L'expression logique de XOR en fonction de la négation logique, AND et OR est alors :

XOR = $(\bar{A}\ et\ B)\ ou\ (A\ et\ \bar{B})$

On utilise parfois l'écriture condensée suivante XOR = $\bar{A}B \quad + \quad A\bar{B}$

Table de vérité de XOr et des sous-expression logiques :

A	**B**	$(\bar{A}\ et\ B)$	$(A\ et\ \bar{B})$	**L**
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

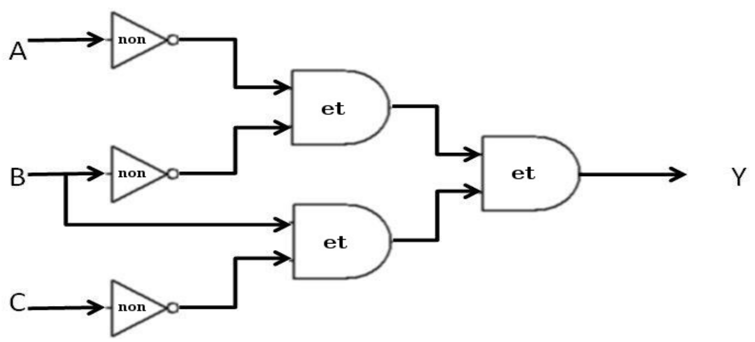
2. Table de vérité d'une expression logique quelconque

1.1 Avec 5 entrées binaires, on peut construire 2^5 combinaisons distinctes des entrées. Réponse: **32**

1.2.1 Donnez la table de vérité et dessinez le circuit logique correspondant à l'expression suivante :

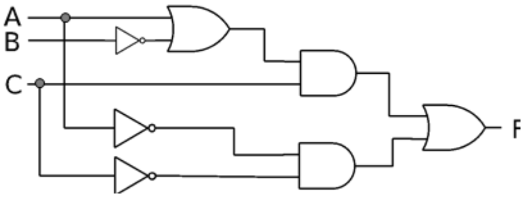
$$Y = (\bar{A}\ et\ \bar{B})\ et\ (B\ et\ \bar{C})$$

A	B	C	$(\bar{A}\ et\ \bar{B})$	$(B\ et\ \bar{C})$	Y
0	0	0	1	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	0	0	0



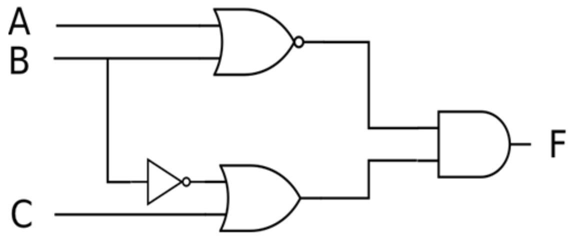
1.2.1 On aurait pu deviner que ce circuit produit toujours une sortie à 0 en observant que l'entrée B et sa négation logique sont impliquées dans des expressions avec seulement des portes AND : la sortie ne peut être que 0 car une porte AND donne 1 seulement si les deux entrées sont à 1.

1.3 L'expression logique correspondant au circuit est :



F = ((A or not B) and C) or (not A and not C) = $\big((A\ or\ \bar{B}) and\ C\big) or\ (\bar{A}\ and\ \bar{C})$ = $(A+\bar{B})C+\bar{A}\bar{C}$

1.4 le circuit qui correspond à l'expression logique est:



3. Compréhension des programmes en assembleur

On considère le programme assembleur suivant:

```
1: charge        r4, r1
2: charge        r3, 0
3: somme         r4, r4, -1
4: cont_neg      r4, 7
5: somme         r3, r2, r3
6: continue      3
7: stop
```

Si r1 = 3 et r2 = 8, quelle est la sortie de ce programme? **r3 = 24**

En général, que fait ce programme?

La boucle (ligne 6) va répéter "r4" fois. Chaque itération, r3 = r2 + r3, où r3 agit comme un accumulateur. r4 commence avec une valeur égale à r1, donc le programme calcule r1 * r2.

4. Programme de multiplication de nombres complexes

Ecrivez un programme assembleur pour calculer le produit de deux nombres complexes y et z. Utilisez les instructions similaires à celles vues au cours (p.ex. *multiplie r1, r2, r3* pour déposer dans r1 le résultat de la multiplication de r2 et r3). Vous pouvez utiliser les registres de r0 à r9. Initialement, le registre r0 contient Réel(y), r1 contient Imag(y), r2 contient Réel(z), r3 contient Im(z). A la fin de l'exécution du programme, la partie réelle du résultat doit se trouver dans r4 et la partie imaginaire dans r5. Pour rappel: (a + bi)(c + di) = (ac - bd) + (ad + bc)i.

Solution

```
// ac-bd → store in r4 as real value
0: multiplie r6, r0, r2 // ac
1: multiplie r7, r1, r3 // bd
2: soustrait r4, r6, r7 // ac-bd
// ad+bc part → store in r5 as imaginary value
3: multiplie r6, r0, r3 // ad
4: multiplie r7, r1, r2 // bc
5: somme r5, r6, r7 // ad+bc
6: stop
```

5. Programme de comparaison de valeurs horaires

Ecrivez un programme assembleur qui détermine laquelle de deux heures, A et B, exprimées en heures et en minutes est la plus petite (c.à.d. arrive le plus tôt dans la journée). Toutes les heures données sont entre 00:00 et 24:00 heures (donc p.ex. 13:45 et pas 1:45). Vous pouvez utiliser les registres de r0 à r9. Le registre r0 (resp. r1) contient le nombre des heures de A (resp. B). Le registre r2 (resp. r3) contient les minutes de A (resp. B).

A la fin de l'exécution du programme, r9 doit contenir 1 si A est une heure strictement plus petite que B et 0 sinon.

Exemple: Si on doit comparer 8h10 et 21h45, on vous donne r0 = 8, r1 = 10, r2 = 21 et r3 = 45 et à la fin de l'exécution, r9 devra contenir 1.

Rappel: L'instruction continue pp a, b, c fait continuer l'exécution à la ligne c si a est un nombre strictement plus petit que b.

Solution

```
0: continue_pp r0, r2, 5  // are the hours in ascending order (A.hour < B.hour) ?
1: continue_pp r2, r0, 3 // are the hours in descending order (A.hour > B.hour) ?
2: continue_pp r1, r3, 5 // hours are same → are minutes in strict ascending order ( A.minute < B.minute ) ?
3: charge r9, 0 // else → “catch all” output 0
4: stop
5: charge r9, 1
6: stop
```

6. Circuit = C'est un circuit ou-exclusif (XOR)

A	B	sortie
0	0	0
0	1	1
1	0	1
1	1	0