

ENSEIGNEMENT DE PROMOTION SOCIALE

Cours de

STRUCTURE DES ORDINATEURS

- Logique et électronique booléenne -

H. Schyns

Mars 2011

Sommaire

1. INTRODUCTION

- 1.1. Position du problème
- 1.2. Le système binaire
- 1.3. L'arithmétique
- 1.4. La logique mathématique
- 1.5. La théorie des ensembles
- 1.6. L'électricité et l'électronique
- 1.7. Equivalence des représentations

2. OPERATEURS ET FONCTIONS LOGIQUES

- 2.1. Introduction
- 2.2. L'opérateur NOT
 - 2.2.1. Fonctionnement
 - 2.2.2. Equivalent mathématique
 - 2.2.3. Représentation dans les ensembles
- 2.3. L'opérateur AND
 - 2.3.1. Fonctionnement
 - 2.3.2. Equivalent mathématique
 - 2.3.3. Représentation dans les ensembles
- 2.4. L'opérateur OR
 - 2.4.1. Fonctionnement
 - 2.4.2. Equivalent mathématique
 - 2.4.3. Représentation dans les ensembles
- 2.5. L'opérateur NAND
 - 2.5.1. Fonctionnement
 - 2.5.2. Equivalent mathématique
 - 2.5.3. Représentation dans les ensembles
- 2.6. L'opérateur NOR
 - 2.6.1. Fonctionnement
 - 2.6.2. Equivalent mathématique
 - 2.6.3. Représentation dans les ensembles
- 2.7. L'opérateur XOR
 - 2.7.1. Fonctionnement
 - 2.7.2. Equivalent mathématique
 - 2.7.3. Représentation dans les ensembles
- 2.8. Récapitulons

3. ELEMENTS D'ALGEBRE BOOLEENNE

- 3.1. Bref rappel d'algèbre
- 3.2. Tautologies
 - 3.2.1. Définition
 - 3.2.2. Démonstration par tableau de vérité

- 3.2.3. Démonstration par les ensembles
- 3.2.4. Utilisation
- 3.2.5. Autre exemple

3.3. Egalités remarquables

4. ELECTRONIQUE BOOLEENNE

- 4.1. De l'électricité à l'électronique**
- 4.2. Le transistor MOSFET**
- 4.3. L'ampli opérationnel**
- 4.4. La porte NOT**
- 4.5. La porte AND**
- 4.6. La porte NAND**
- 4.7. La porte OR**
- 4.8. La porte NOR**
- 4.9. La porte XOR**
- 4.10. Porte tri-state**
- 4.11. Tautologies**
- 4.12. Cascades AND et OR**

5. QUELQUES CIRCUITS

- 5.1. Position du problème**
- 5.2. Retrouver la logique d'un tableau**
- 5.3. Le clavier**
- 5.4. L'additionneur 1 bit + 1 bit**
- 5.5. L'additionneur 1 bit + 1 bit + report**
- 5.6. L'additionneur 1 byte + 1 byte**
- 5.7. L'inverseur de signe**
- 5.8. L'additionneur-soustracteur**
- 5.9. L'incrémenteur**
- 5.10. La bascule R/S**
- 5.11. Le multiplicateur 2 bits x 2 bits**
- 5.12. Le décodeur d'adresse**
 - 5.12.1. Définition
 - 5.12.2. L'arbre binaire
 - 5.12.3. Le décodeur d'adresse à 1 bit
 - 5.12.4. Le décodeur d'adresse à 2^n bits
- 5.13. Le démultiplexeur**
- 5.14. Le multiplexeur**
- 5.15. Combinaison Multiplexeur-Démultiplexeur**
- 5.16. Le contrôleur de parité**
- 5.17. L'afficheur digital**

5.18. Le convertisseur Binaire-BCD

- 5.18.1. Le comparateur GE5
- 5.18.2. Le circuit "Add 3 if GE5"
- 5.18.3. L'assemblage final

5.19. Conclusion**6. EXERCICES**

- 6.1. Exercice 1
- 6.2. Exercice 2
- 6.3. Exercice 3
- 6.4. Exercice 4
- 6.5. Exercice 5
- 6.6. Exercice 6
- 6.7. Exercice 7

7. ANNEXES**7.1. Circuits équivalents**

- 7.1.1. Principe
- 7.1.2. Cas d'une seule entrée
- 7.1.3. Cas de deux entrées

7.2. Cryptage XOR**8. SOURCES**

1. Introduction

1.1. Position du problème

Comprendre le fonctionnement interne d'un ordinateur n'est pas immédiat ; la démarche passe par plusieurs étapes qui résolvent chacune une partie du problème (fig. 1.1).

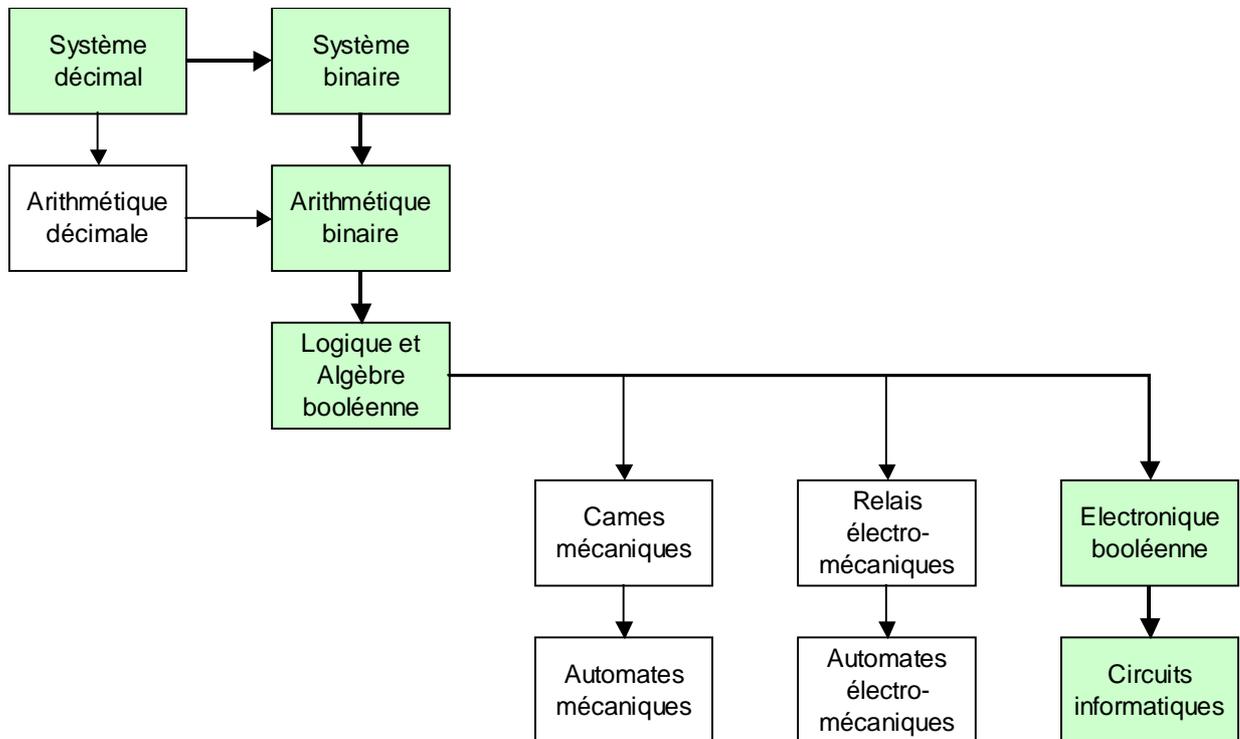


fig. 1.1 Du système décimal au circuits de l'ordinateur

Dans les modules précédents, nous sommes passés du système décimal et de l'arithmétique décimale au système binaire et à l'arithmétique binaire.

Dans ce module, nous verrons d'abord comment chaque opération de l'arithmétique binaire peut être traduite en une expression d'algèbre booléenne, c'est-à-dire en une suite d'opérations de logique mathématique.

Nous verrons ensuite comment les opérations élémentaires de l'algèbre booléenne peuvent être transposées et exécutées par des montages électroniques.

L'étape finale consistera à assembler les montages élémentaires en des circuits de plus en plus complexes qui réalisent toutes les opérations désirées, y compris le retour vers la numérotation décimale.

1.2. Le système binaire

Nous avons vu dans un chapitre précédent que le système binaire utilise deux signes (**0** et **1**) pour représenter les nombres.

Tous les nombres, qu'ils soient entiers ou réels, positifs ou négatifs, peuvent se traduire en langage binaire en utilisant plusieurs bits. Il en va de même pour les caractères, les instructions, les sons, les couleurs, etc. Habituellement, ce codage s'effectue sur un ou plusieurs octets (*ang. : bytes*) qui sont des paquets de huit bits.

1.3. L'arithmétique

Nous savons depuis notre enfance que le système décimal admet des opérations arithmétiques dont les résultats s'expriment sous la forme de nombres décimaux. Les tables d'addition et de multiplication contiennent chacune cent lignes dont voici un extrait :

Addition	Multiplication
:	:
3 + 3 = 6	3 * 3 = 9
3 + 4 = 7	3 * 4 = 12
3 + 5 = 8	3 * 5 = 15
:	:

Notons au passage que, dans certains cas, il y a report d'une ou plusieurs unités sur le rang suivant.

Nous avons vu que le système binaire admet lui aussi les opérations arithmétiques. Les tables d'addition et de multiplication se réduisent chacune à quatre lignes :

Addition	Multiplication
0 + 0 = 0	0 * 0 = 0
0 + 1 = 1	0 * 1 = 0
1 + 0 = 1	1 * 0 = 0
1 + 1 = 10	1 * 1 = 1

Ici, par contre, il n'y a qu'un seul cas de report d'une unité sur le rang suivant.

1.4. La logique mathématique

La logique mathématique - qui sera développée dans ce document - concerne l'analyse des affirmations, des propositions, des conditions.

La logique n'admet que deux résultats : une proposition est soit **vraie** (*ang.*: *true*) soit **fausse** (*ang.*: *false*) Elle ne peut pas être partiellement vraie ou partiellement fausse car cela signifierait que le problème n'a pas été correctement analysé ⁽¹⁾.

Un exemple de proposition logique serait

x est plus grand que 15

Remplacez x par un nombre quelconque (par ex. 18) vous constaterez que, pour la valeur choisie, la réponse à la proposition sera **toujours** soit **vraie** (ou **oui**) soit **fausse** (ou **non**). Vérifiez avec 15 pour x et répondez strictement à la question.

La proposition intervient généralement dans une condition qui détermine un choix :

```

Si [ x est plus grand que 15 ]
alors je fais ceci
sinon je fais autre chose
  
```

Bien que cette approche puisse sembler assez restrictive, nous verrons qu'elle fonctionne très bien dans des cas complexes.

¹ Pensez à ces films américains qui se passent dans un tribunal. Il arrive toujours un moment où le procureur demande au prévenu : "Répondez par oui ou par non". Si le prévenu veut répondre "Oui mais..." c'est que les circonstances n'ont pas été complètement analysées. L'avocat de la défense agira en conséquence en posant des questions supplémentaires.

En effet, à l'image de l'arithmétique qui définit des opérateurs s'appliquant aux nombres, la logique définit des opérateurs qui s'appliquent aux propositions. Ces opérateurs nous permettront de décomposer des propositions complexes en un certain nombre de propositions élémentaires.

1.5. La théorie des ensembles

Georg Cantor ⁽¹⁾ et John Venn ⁽¹⁾ sont deux mathématiciens qui ont développé la théorie des ensembles. On représente habituellement un ensemble par une forme elliptique (fig. 1.2). Celle-ci définit deux zones :

- la zone (1) des éléments qui **appartiennent** à l'ensemble (c'est-à-dire qui vérifient sa définition) et
- la zone (2) des éléments qui ne lui appartiennent pas (c'est-à-dire qui ne vérifient pas sa définition).

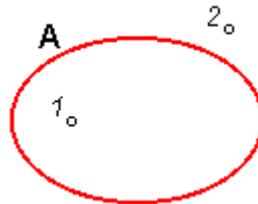


fig. 1.2 Un ensemble est aussi un système binaire

Traduit dans la théorie des ensembles, notre proposition du point 1.4 serait :

Soit A, l'ensemble des nombres supérieurs à 15

Quel que soit le nombre choisi, il se place toujours soit dans la zone (1), soit dans la zone (2).

1.6. L'électricité et l'électronique

Un bit du langage binaire peut être facilement traduit dans un langage électrique (**on** ou **off**, **haut** ou **bas**). Ainsi, un interrupteur peut être ouvert ou fermé (fig. 1.3), le courant circule ou non, un condensateur est chargé ou vide, etc.

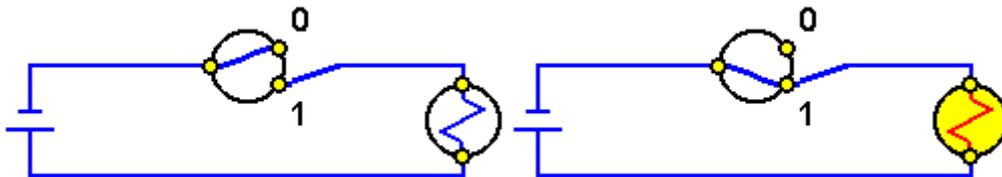


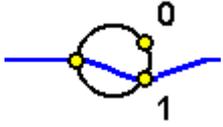
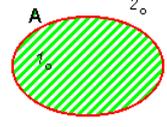
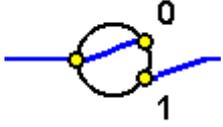
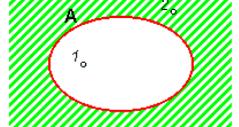
fig. 1.3 Un interrupteur implémente une variable binaire

1.7. Equivalence des représentations

Le lecteur attentif aura noté que les différents systèmes repris ci-dessus sont fort similaires : il n'y a jamais que deux états possibles. C'est pourquoi on parle de représentations **binaires**.

Cette similitude implique que toute proposition exprimée dans un système peut parfaitement se traduire dans n'importe quel autre système.

¹ Mathématicien allemand, créateur de la théorie des ensembles (1845 - 1918).

Logique		Binaire	Electrique	Ensemble
Vrai True	Oui Yes	1		
Faux False	Non No	0		

1 Mathématicien et logicien britannique, célèbre pour avoir conçu les diagrammes qui portent son nom (1834 - 1923)

2. Opérateurs et fonctions logiques

2.1. Introduction

Les systèmes numériques (binaire, décimal, hexadécimal) permettent l'emploi d'**opérateurs arithmétiques**. Ce sont les opérateurs

$$+ - * /$$

Ils définissent les opérations d'addition, de soustraction, de multiplication et de division. Ils permettent aussi de définir des fonctions beaucoup plus complexes telles que les fonctions trigonométriques, logarithmiques, exponentielles, etc.

Les systèmes binaires admettent des opérateurs supplémentaires appelés **opérateurs logiques**.

$$\neg \wedge \vee \oplus$$

Ils définissent un certain nombre d'opérations que nous allons étudier dans la suite de ce chapitre. Ils permettent également de définir des fonctions plus complexes. Pour étudier les fonctions de variables binaires on utilise une algèbre développée au XIX^{ème} siècle par un mathématicien anglais : Georges Boole (). C'est pourquoi on parle de variables booléennes, d'opérateurs booléens, d'algèbre booléenne, etc.

Nous utiliserons principalement les termes anglais en lieu et place des termes français car c'est la langue utilisée dans toute la littérature technique.

2.2. L'opérateur NOT

2.2.1. Fonctionnement

L'opérateur **NOT** (fr.: *NON* ou *PAS*) renvoie l'opposé d'une valeur logique. Il ne prend qu'un seul opérande.

Son application abonde dans le langage courant. Prenons l'affirmation

A:	Il pleut
----	----------

l'application de l'opérateur donne l'affirmation contraire

NOT A:	NOT (il pleut) = Il ne pleut pas
--------	-------------------------------------

On remarque que si la première affirmation (A) est vraie (V), alors, la seconde (NOT A) est fausse (F). Si la première est fausse, la seconde est vraie. Ceci est schématisé dans ce qu'on nomme une **table de vérité** :

<u>A</u>	<u>NOT A</u>	<u>A</u>	<u>\overline{A}</u>
F	V	0	1
V	F	1	0

La table de gauche reprend le symbolisme "logique", celle de droite reprend le symbolisme "binaire". Elles sont absolument équivalentes. Remarquez la barre horizontale au-dessus du A de la dernière colonne; il signifie NOT A. On rencontre aussi parfois la notation $\neg A$.

2.2.2. Equivalent mathématique

Dans le système binaire, on réalise l'opération NOT à l'aide d'une soustraction :

$$\text{NOT } A = \bar{A} \equiv 1 - A$$

Le tableau de vérité confirme l'équivalence :

A	1 - A
0	1
1	0

Nous avons vu dans un module précédent que, lors du codage des nombres entiers négatifs, il fallait commencer par remplacer tous les 1 par des 0 et réciproquement. Il est donc très probable que l'opérateur NOT sera impliqué dans cette opération.

2.2.3. Représentation dans les ensembles

Dans la théorie des ensembles, le domaine dans lequel la proposition est vraie est représenté par l'intérieur du contour, tandis que le domaine dans lequel la proposition est fautive est représenté par l'extérieur (fig. 2.1) :

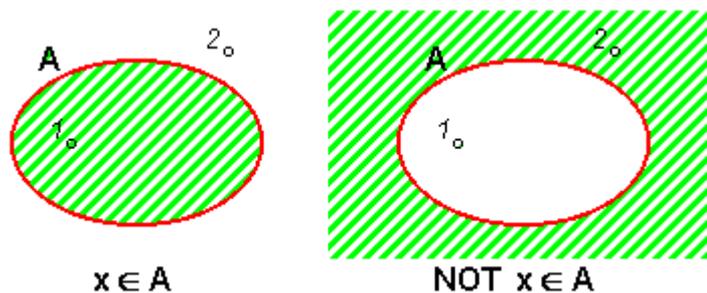


fig. 2.1 L'ensemble et son contraire

2.3. L'opérateur AND

2.3.1. Fonctionnement

L'opérateur **AND** (fr.: ET) combine **deux** valeurs logiques pour donner un résultat. Il prend donc deux opérandes.

Le fonctionnement de l'opérateur AND est conforme à celui du ET dans le langage courant :

A:	Il pleut
B:	Je suis à l'extérieur

l'application de l'opérateur AND donne

A AND B:	SI Il pleut AND je suis à l'extérieur ALORS je suis mouillé
----------	--

On remarque que si l'une des deux affirmations est fautive, alors le résultat est faux (je ne suis pas mouillé). Le résultat n'est vrai (je suis mouillé) que si les deux propositions sont vraies en même temps. Ceci est schématisé dans la **table de vérité**.

Les deux propositions sont **indépendantes** : il pleut ou non est indépendant du fait que je sois à l'extérieur ou non et réciproquement. Et, d'autre part, chaque

proposition admet deux états : vrai ou faux. Dès lors, la table de vérité comporte quatre lignes :

2 propositions x 2 états = 4 lignes

<u>A</u>	<u>B</u>	<u>A AND B</u>	<u>A</u>	<u>B</u>	<u>A ∧ B</u>
F	F	F	0	0	0
F	V	F	0	1	0
V	F	F	1	0	0
V	V	V	1	1	1

On note que le résultat du **AND** est **toujours FAUX** sauf quand les deux propositions sont **VRAIES**.

2.3.2. Equivalent mathématique

Dans le système binaire, le résultat de l'opération AND ressemble furieusement à une multiplication :

$$A \text{ AND } B = A \wedge B = A \bullet B$$

Le tableau de vérité confirme l'équivalence :

<u>A</u>	<u>B</u>	<u>A • B</u>
0	0	0
0	1	0
1	0	0
1	1	1

2.3.3. Représentation dans les ensembles

Puisque l'opérateur AND fait intervenir deux opérandes, nous devons utiliser deux ensembles qui se recouvrent partiellement :

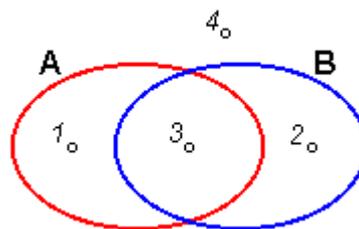


fig. 2.2 Deux propositions représentées par deux ensembles

Dans ce schéma, on distingue quatre zones correspondant aux quatre possibilités définies plus haut :

- les éléments qui appartiennent à A et pas à B [1]
- les éléments qui appartiennent à B et pas à A [2]
- les éléments qui appartiennent à A et à B [3]
- les éléments qui n'appartiennent ni à A ni à B [4]

Le domaine dans lequel l'une **et** l'autre des deux propositions sont vraies est représenté par l'**intersection** des deux ensembles :

$$x \in A \text{ AND } x \in B \equiv x \in A \cap B$$

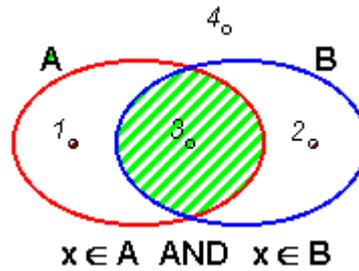


fig. 2.3 AND est représenté par l'intersection de deux ensembles

2.4. L'opérateur OR

2.4.1. Fonctionnement

L'opérateur **OR** (fr.: *OU*) combine aussi **deux** valeurs logiques. Son fonctionnement est semblable à celui du OU dans le langage courant :

A:	La route est bloquée
B:	Je pars trop tard

l'application de l'opérateur OR donne

A OR B:	SI La route est bloquée OR Je pars trop tard ALORS J'arrive en retard
---------	--

On remarque qu'il suffit que l'une des deux affirmations soit vraie pour que le résultat soit vrai (j'arrive en retard). Le résultat n'est faux (je n'arrive pas en retard) que si les deux propositions sont fausses en même temps. Ceci est schématisé dans la **table de vérité**.

A nouveau, la table de vérité comporte quatre lignes car il s'agit de deux propositions indépendantes qui admettent chacune deux états.

<u>A</u>	<u>B</u>	<u>A OR B</u>	<u>A</u>	<u>B</u>	<u>A ∨ B</u>
F	F	F	0	0	0
F	V	V	0	1	1
V	F	V	1	0	1
V	V	V	1	1	1

On note que le résultat du OR est toujours VRAI sauf quand les deux propositions sont FAUSSES .

2.4.2. Equivalent mathématique

Dans le système binaire, le résultat de l'opération OR ressemble assez bien à une addition :

$$A \text{ OR } B = A \vee B \cong A + B$$

En effet :

<u>A</u>	<u>B</u>	<u>A + B</u>
0	0	0
0	1	1
1	0	1
1	1	10

Il faut comprendre que la dernière ligne donne une valeur non nulle.

2.4.3. Représentation dans les ensembles

Le domaine dans lequel l'une **ou** l'autre des deux propositions est vraie est représenté par l'**union** des deux ensembles :

$$x \in A \text{ OR } x \in B \equiv x \in A \cup B$$

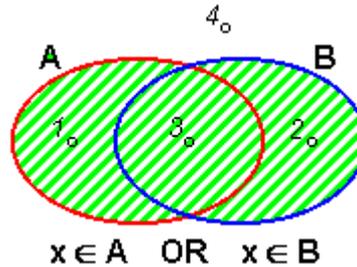


fig. 2.4 OR est représenté par l'union de deux ensembles

2.5. L'opérateur NAND

2.5.1. Fonctionnement

L'opérateur NAND (fr.: *NON ET*) résulte de l'enchaînement d'un opérateur AND, qui combine deux valeurs logiques et d'un opérateur NOT qui inverse le résultat. Son fonctionnement est semblable au ET mais avec une proposition résultante qui est *l'inverse* de ce que dicte le langage courant :

A:	Il pleut
B:	Je suis à l'extérieur

l'application de l'opérateur NAND donne

A NAND B:	SI Il pleut NAND je suis à l'extérieur ALORS je suis au sec
-----------	--

Ce qui peut aussi s'écrire :

NOT(A AND B):	SI NOT(Il pleut AND je suis à l'extérieur) ALORS je suis au sec
---------------	--

L'illustration dans le langage courant n'est pas aisée car elle est contraire à l'usage courant. Le résultat est toujours vrai (je suis au sec) sauf dans le cas où les deux propositions sont vraies en même temps (pleut-il et suis-je à l'extérieur ? oui ? ben non, c'est le contraire!). Il est plus facile d'appliquer aveuglément la **table de vérité**.

<u>A</u>	<u>B</u>	<u>A AND B</u>	<u>A NAND B</u>	<u>A</u>	<u>B</u>	<u>A ∧ B</u>	<u>¬(A ∧ B)</u>
F	F	F	V	0	0	0	1
F	V	F	V	0	1	0	1
V	F	F	V	1	0	0	1
V	V	V	F	1	1	1	0

On note que le résultat du NAND est toujours VRAI sauf quand les deux propositions sont VRAIES.

2.5.2. Equivalent mathématique

Dans le système binaire, le résultat de l'opération NAND est la combinaison d'un NOT et d'un AND :

$$A \text{ NAND } B = \text{NOT}(A \text{ AND } B) = \overline{A \cdot B} = 1 - (A \cdot B)$$

En effet :

A	B	A • B	1 - A • B
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

2.5.3. Représentation dans les ensembles

Le domaine dans lequel le résultat est vrai est l'ensemble du plan sauf l'intersection des deux ensembles :

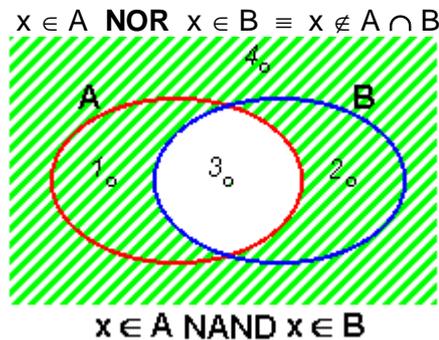


fig. 2.5 NAND est tout sauf l'intersection des deux ensembles

2.6. L'opérateur NOR

2.6.1. Fonctionnement

L'opérateur NOR (fr.: NON OU) résulte de l'enchaînement d'un opérateur OR, qui combine deux valeurs logiques et d'un opérateur NOT qui inverse le résultat. Son fonctionnement est semblable au OU mais avec une proposition résultante qui est l'inverse de ce que dicte le langage courant :

A:	La route est bloquée
B:	Je pars trop tard

l'application de l'opérateur NOR donne

A NOR B:	SI La route est bloquée NOR Je pars trop tard ALORS J'arrive à temps
----------	---

Comme dans le cas de NAND, L'illustration de NOR dans le langage courant n'est pas aisée. Le résultat est toujours faux (je n'arrive pas à temps) sauf dans le cas où les deux propositions sont fausses en même temps (ni blocage, ni parti trop tard (1)). Voyons plutôt la **table de vérité**.

1 On peut assimiler le NOR à l'usage du "ni" du langage courant mais il faut rester vigilant. Une expression telle que "il n'y a ni pluie, ni nuages" se traduit littéralement par (NOT pluie) AND (NOT nuages). Il se fait ./.

A	B	A OR B	A NOR B	A	B	A ∨ B	$\overline{A \vee B}$
F	F	F	V	0	0	0	1
F	V	V	F	0	1	1	0
V	F	V	F	1	0	1	0
V	V	V	F	1	1	1	0

On note que le résultat du NAND est toujours FAUX sauf quand les deux propositions sont FAUSSES.

2.6.2. Equivalent mathématique

Dans le système binaire, le résultat de l'opération NOR est la combinaison d'un NOT et d'un OR :

$$A \text{ NOR } B = \text{NOT}(A \text{ OR } B) = \overline{A \vee B} \cong 1 - (A + B)$$

A	B	A + B	1 - (A + B)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	10	0 !!!

Attention, pour interpréter correctement le tableau, il faut comprendre que, dans la dernière ligne, A+B donne une valeur non nulle (NOT 0).

2.6.3. Représentation dans les ensembles

Le domaine dans lequel le résultat NOR est vrai est l'ensemble du plan sauf l'union des deux ensembles :

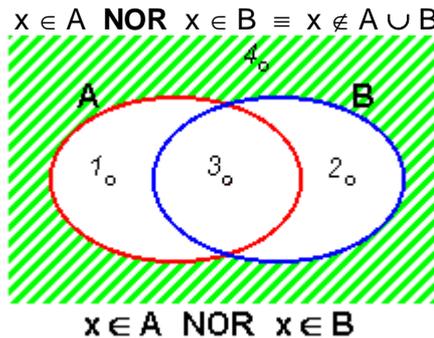


fig. 2.6 NOR est tout sauf l'union de deux ensembles

2.7. L'opérateur XOR

2.7.1. Fonctionnement

L'opérateur XOR ou Exclusive OR (*fr.:* OU Exclusif) fonctionne comme l'opérateur OR à une exception près : les deux propositions ne peuvent être vraies simultanément. On utilise implicitement un XOR dans le langage courant chaque fois que l'un des choix exclut l'autre (soit) :

que, comme on le verra plus tard, le résultat est équivalent à celui de l'expression NOT (pluie OR nuage)

A:	Je suis au cinéma
B:	Je suis au théâtre

L'application de l'opérateur XOR donne

A XOR B:	SI Je suis au cinéma XOR Je suis au théâtre ALORS Je suis sorti
----------	--

Il suffit que l'une des deux affirmations soit vraie pour que le résultat soit vrai (je suis sorti ce soir). Le résultat est faux (je ne suis pas sorti) si les deux propositions sont fausses en même temps et, finalement, les deux propositions ne peuvent être vraies simultanément (il m'est impossible d'être à la fois au cinéma et au théâtre) donc le cas ne peut être vrai.

<u>A</u>	<u>B</u>	<u>A XOR B</u>	<u>A</u>	<u>B</u>	<u>A ⊕ B</u>
F	F	F	0	0	0
F	V	V	0	1	1
V	F	V	1	0	1
V	V	F	1	1	0

On note que le résultat du XOR est **VRAI** quand l'une et seulement une des deux propositions est **VRAIE**.

2.7.2. Equivalent mathématique

Dans le système binaire, le résultat de l'opération XOR ressemble mieux à une addition que le OR si on ne considère que le dernier bit :

$$A \text{ XOR } B = A \oplus B$$

<u>A</u>	<u>B</u>	<u>A + B</u>
0	0	0
0	1	1
1	0	1
1	1	(1) 0

2.7.3. Représentation dans les ensembles

Le domaine dans lequel l'une **ou** l'autre des deux propositions est vraie à l'exclusion des deux est représenté par l'**union** des deux ensembles moins leur intersection :

$$x \in A \text{ XOR } x \in B \equiv x \in A \cup B \text{ AND } x \notin A \cap B$$

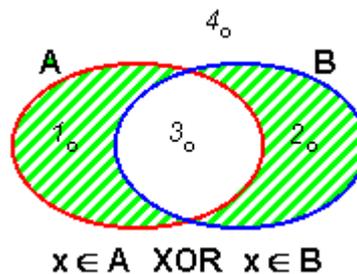


fig. 2.7 XOR est représenté par l'union de deux ensembles mais pas leur intersection

2.8. Récapitulons

Table de vérité	Ensemble	Table de vérité	Ensemble																														
<p>NOT</p> <table border="1"> <tr> <td>A</td> <td>\overline{A}</td> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	\overline{A}	0	1	1	0	<p>NOT $x \in A$</p>																										
A	\overline{A}																																
0	1																																
1	0																																
<p>AND</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$A \wedge B$</td> </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	$A \wedge B$	0	0	0	0	1	0	1	0	0	1	1	1	<p>$x \in A$ AND $x \in B$</p>	<p>NAND</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$\overline{A \wedge B}$</td> </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	$\overline{A \wedge B}$	0	0	1	0	1	1	1	0	1	1	1	0	<p>$x \in A$ NAND $x \in B$</p>
A	B	$A \wedge B$																															
0	0	0																															
0	1	0																															
1	0	0																															
1	1	1																															
A	B	$\overline{A \wedge B}$																															
0	0	1																															
0	1	1																															
1	0	1																															
1	1	0																															
<p>OR</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$A \vee B$</td> </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	$A \vee B$	0	0	0	0	1	1	1	0	1	1	1	1	<p>$x \in A$ OR $x \in B$</p>	<p>NOR</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$\overline{A \vee B}$</td> </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	$\overline{A \vee B}$	0	0	1	0	1	0	1	0	0	1	1	0	<p>$x \in A$ NOR $x \in B$</p>
A	B	$A \vee B$																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	1																															
A	B	$\overline{A \vee B}$																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	0																															
<p>XOR</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$A \oplus B$</td> </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	$A \oplus B$	0	0	0	0	1	1	1	0	1	1	1	0	<p>$x \in A$ XOR $x \in B$</p>	<p>NXOR</p> <table border="1"> <tr> <td>A</td> <td>B</td> <td>$\overline{A \oplus B}$</td> </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	$\overline{A \oplus B}$	0	0	1	0	1	0	1	0	0	1	1	1	<p>$x \in A$ NXOR $x \in B$</p>
A	B	$A \oplus B$																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	0																															
A	B	$\overline{A \oplus B}$																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	1																															

3. Eléments d'algèbre booléenne

3.1. Bref rappel d'algèbre

L'algèbre classique combine les opérateurs arithmétiques (+ - * /) afin de construire des expressions plus compliquées du genre :

$$A * (B + C) + B * (A + C) + C * (A + B)$$

Elle définit aussi des règles d'écriture et de calcul telles que :

- écriture simplifiée : $A * B = AB$
- regroupement : $AB + AB = 2 AB$
- permutation : $A + B = B + A$ $A * B = B * A$
- associativité : $(A + B) + C = A + (B + C)$ $(A * B) * C = A * (B * C)$
- distributivité : $A * (B + C) = A * B + A * C = AB + AC$

Ces règles permettent de démontrer des équivalences telles que :

$$(A + B)^2 = A^2 + 2 AB + B^2$$

C'est en appliquant ces règles lues tantôt dans le sens gauche-droite, tantôt dans le sens droite-gauche que l'on simplifie l'expression donnée plus haut :

$$\begin{aligned}
 & A * (B + C) + B * (A + C) + C * (A + B) \\
 \text{par distribution} &= A * B + A * C + B * A + B * C + C * A + C * B \\
 \text{par permutation} &= A * B + A * C + A * B + B * C + A * C + B * C \\
 \text{écriture simplifiée} &= AB + AC + AB + BC + AC + BC \\
 \text{par regroupement} &= 2 AB + 2 AC + 2 BC \\
 \text{par mise en évidence} &= 2 (AB + AC + BC)
 \end{aligned}$$

3.2. Tautologies

3.2.1. Définition

L'algèbre booléenne combine aussi les opérateurs qui lui sont propres pour construire des expressions plus compliquées. On y définit aussi des égalités remarquables nommées tautologies telles que

$$\overline{(A \wedge B)} \Leftrightarrow \bar{A} \vee \bar{B}$$

ce qui se lit

$$\text{NOT (A AND B) équivaut à (NOT A) OR (NOT B)}$$

Une des techniques de démonstration des tautologies est extrêmement simple : il suffit de construire les tableaux de vérités des deux côtés de l'équivalence et de regarder si, pour chaque ligne, le résultat du côté gauche est identique à celui du côté droit. Si c'est le cas, l'équivalence est acceptée.

On peut aussi démontrer les tautologies à l'aide de la théorie des ensembles.

3.2.2. Démonstration par tableau de vérité

Reprenons notre exemple.

$$\overline{(A \wedge B)} \Leftrightarrow \bar{A} \vee \bar{B}$$

La première étape consiste à repérer le nombre de propositions élémentaires qui y interviennent. Dans l'exemple présent, il y en a deux : A et B. Nous devons donc considérer quatre cas (voir 2.3.1).

Construisons un tableau de vérité à quatre lignes en détaillant les opérations ainsi que l'ordre dans lequel nous allons pouvoir les effectuer :

1	2	6	3	8	4	7	5
A	B	NOT	(A AND B)	⇔	NOT A	OR	NOT B
0	0						
0	1						
1	0						
1	1						

Remplissons les colonnes pas à pas en tenant compte de leur priorité respective. Nous devons remplir (3), (4), et (5) avant de traiter (6) et (7)

1	2	6	3	8	4	7	5
A	B	NOT	(A AND B)	⇔	NOT A	OR	NOT B
0	0		0		1		1
0	1		0		1		0
1	0		0		0		1
1	1		1		0		0

A présent, nous sommes en mesure d'évaluer (6) à partir de (3) et (7) à partir de (4) et (5)

1	2	6	3	8	4	7	5
A	B	NOT	(A AND B)	⇔	NOT A	OR	NOT B
0	0	1	0		1	1	1
0	1	1	0		1	1	0
1	0	1	0		0	1	1
1	1	0	1		0	0	0

Vérifions maintenant ligne par ligne si les valeurs reprises dans (6) et dans (7) sont identiques et inscrivons le résultat dans (8) . C'est effectivement le cas

1	2	6	3	8	4	7	5
A	B	NOT	(A AND B)	⇔	NOT A	OR	NOT B
0	0	1	0	V	1	1	1
0	1	1	0	V	1	1	0
1	0	1	0	V	0	1	1
1	1	0	1	V	0	0	0

On peut donc affirmer que

NOT (A AND B) équivaut à (NOT A) OR (NOT B)

CQFD ⁽¹⁾.

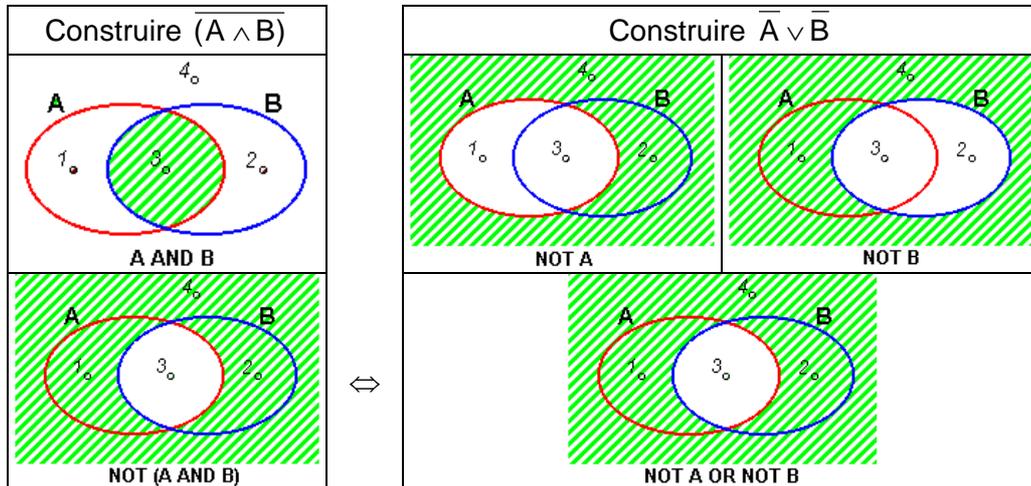
1 J'entends d'ici les voix qui s'écrient : "Bon, et alors ?". Un peu de patience, on en parlera au point 3.2.4

3.2.3. Démonstration par les ensembles

Pour démontrer que

$$\overline{(A \wedge B)} \Leftrightarrow \overline{A} \vee \overline{B}$$

Commençons par le membre de gauche; représentons l'intersection de A et de B puis prenons-en l'inverse. Pour le membre de droite; représentons NOT A et NOT B puis prenons-en l'union. On s'aperçoit que les zones hachurées finales sont identiques.



3.2.4. Utilisation

Cette équivalence est largement utilisée en informatique et dans les processeurs. En effet, on est souvent amené à exécuter une action si la proposition est vraie et une autre action si la proposition est fausse :

```
Si (condition) Alors (faire ceci) Sinon (faire cela)
```

Il est parfois plus facile de commencer par faire ce qu'il faut quand la condition n'est pas vérifiée :

```
Si (NOT condition) Alors (faire cela) Sinon (faire ceci)
```

Par exemple :

```
Si un nombre est divisible par 2 ET par 3,
alors calculer sa racine carrée
sinon passer au nombre suivant
```

Question : La racine carrée étant une opération longue et compliquée, quels sont les nombres dont je ne dois pas calculer la racine carrée ?

Exprimons la proposition en termes booléens (notez la simple flèche) :

```
(Divisible par 2) AND (Divisible par 3) => racine carrée
```

Exprimons son contraire :

```
NOT(Divisible par 2 AND Divisible par 3)
=> NOT(racine carrée)
```

Utilisons la tautologie :

```
NOT (Divisible par 2) OR NOT (Divisible par 3)
=> NOT(racine carrée)
```

Donc

Si un nombre n'est pas divisible par 2
 OU s'il n'est pas divisible par 3,
 alors passer au nombre suivant
 sinon calculer sa racine carrée.

Conclusion : si je tombe sur un nombre impair (non divisible par 2), je passe directement au suivant. L'opérateur OU me dispense de vérifier s'il est divisible par 3.

3.2.5. Autre exemple

Démontrons que l'opérateur XOR peut s'écrire à l'aide d'opérateurs AND et OR

$$A \oplus B \Leftrightarrow (\bar{A} \wedge B) \vee (A \wedge \bar{B})$$

1	2	7	11	3	8	4	10	5	9	6
A	B	A XOR B	\Leftrightarrow	(NOT A AND B)	OR	(A AND NOT B)				
0	0	0	V	1	0	0	0	0	0	1
0	1	1	V	1	1	1	1	0	0	0
1	0	1	V	0	0	0	1	1	1	1
1	1	0	V	0	0	1	0	1	0	0

Il y a deux variables : A, B, ce qui nous donne quatre lignes :

- nous pouvons remplir les colonnes (1) et (2);
- nous sommes ensuite en mesure de compléter (3), (4), (5), (6) à partir des valeurs de (1) et (2);
- évaluons (7) grâce à (1) et (2); (8) grâce à (3) et (4); et (9) grâce à (5) et (6);
- les deux parenthèses étant calculées, nous pouvons remplir (10) à partir de (8) et (9). Elle est toujours vraie.

3.3. Egalités remarquables

Les principales égalités remarquables sont reprises dans le tableau ci-dessous. La plupart d'entre elles sont assez évidentes :

1	A • 0 = 0	A AND 0 = 0	toujours faux
2	A • 1 = A	A AND 1 = A	
3	A + 0 = A	A OR 0 = A	
4	A + 1 = 1	A OR 1 = 1	toujours vrai
5	$\overline{\overline{A}} = A$	NOT (NOT A) = A	
6	A • \overline{A} = 0	A AND (NOT A) = 0	toujours faux
7	A + \overline{A} = 1	A OR (NOT A) = 1	toujours vrai
8	$\overline{(A \bullet B)} = \overline{A} + \overline{B}$	NOT (A AND B) = (NOT A) OR (NOT B)	
9	$\overline{(A + B)} = \overline{A} \bullet \overline{B}$	NOT (A OR B) = (NOT A) AND (NOT B)	
10	A • (B + C) = A • B + A • C	A AND (B OR C) = (A AND B) OR (A AND C)	
11	A + (B • C) = (A + B) • (A + C)	A OR (B AND C) = (A OR B) AND (A OR C)	
12	A ⊕ B = ($\overline{A} \bullet B$) + (A • \overline{B})	A XOR B = (NOT A AND B) OR (A AND NOT B)	

4. Electronique booléenne

4.1. De l'électricité à l'électronique

Nous avons vu plus haut qu'un bit du langage binaire peut être facilement traduit dans un langage électrique. Ainsi, un interrupteur peut être ouvert ou fermé, le courant circule ou non, un condensateur est chargé ou vide, etc (fig. 4.1).

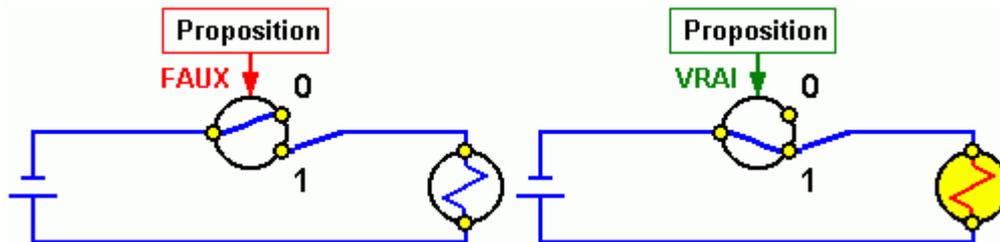


fig. 4.1 Un interrupteur commande une variable binaire

En pratique un niveau (0 ou 1) est défini par un domaine de tension ou de courant.

Par exemple, en termes de tension, on dira qu'un conducteur est à un **niveau haut** (=1) s'il présente une différence de potentiel comprise entre +2 V et +5 V par rapport à la masse. On dira qu'il est à un **niveau bas** (=0) si la différence de potentiel est inférieure à +0.8 V ⁽¹⁾.

En termes de courant, un conducteur sera à un **niveau haut** (=1) s'il est parcouru par un courant de 20 mA. Il sera à un niveau **bas** (=0) s'il est parcouru par un courant de 4 mA ⁽²⁾.

Dans un ordinateur actuel, une valeur binaire est transmise au moyen d'une tension (Volts) et non par un courant (Ampères).

Toute la question est de savoir *comment* une condition initiale (la proposition) peut agir sur l'interrupteur qui va transmettre le signal.

Dans tout système informatique, tout commence avec un doigt qui appuie sur un bouton (fig. 4.2).

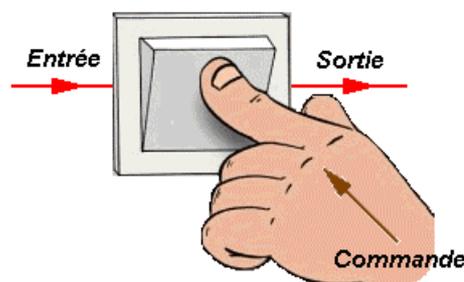


fig. 4.2 A l'origine, il y a toujours un doigt qui appuie sur un bouton

- 1 Entre les deux, l'état est indéterminé et nous verrons qu'il est indispensable de traverser cette page intermédiaire le plus rapidement possible.
- 2 Choisir 4 mA et non 0 mA pour le niveau bas permet de détecter une coupure dans le circuit de transmission.

Le module de commande élémentaire comporte trois composantes :

- une entrée qui assure l'alimentation,
- une sortie qui assure la propagation,
- un organe de commande qui provoque le déclenchement.

La sortie peut ensuite être utilisée comme organe de commande pour le module de commande suivant. Depuis la découverte de l'électricité, on a utilisé successivement :

- les relais électromécaniques,
- les tubes à vide (triodes),
- les transistors analogiques,
- les transistors MOSFET.

4.2. Le transistor MOSFET

Le transistor MOSFET ⁽¹⁾ est le constituant principal des circuits intégrés actuels. Son fonctionnement est décrit en détail dans un autre document.

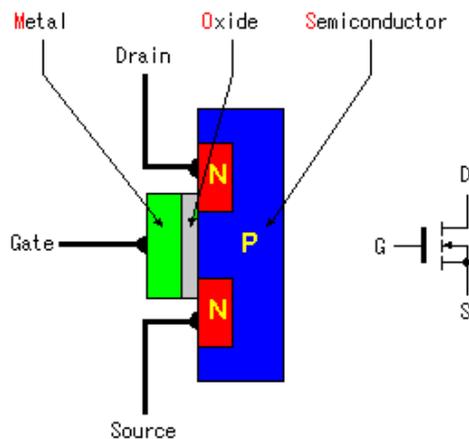


fig. 4.3 Schéma d'un transistor MOSFET

Le MOSFET comprend les trois éléments habituels (fig. 4.3) :

- la source pour l'alimentation,
- le drain pour la propagation,
- le grille (*ang. : gate*) pour la commande.

Si la grille est à la masse (état bas = 0 V), aucun courant ne circule entre la source et le drain. Si la grille est portée à l'état haut (5 V) alors le courant circule entre la source et le drain.

1 **Metal Oxide Silicon Field Effect Transistor** ou transistor à effet de champ à structure métal-oxyde-semiconducteur. Le principe théorique est démontré en 1920, le premier prototype est construit en 1960 et la première intégration dans un chip date de 1963 mais c'est dans les années 1970 qu'ils ont envahi le marché.

4.3. L'ampli opérationnel

Un premier montage consiste à utiliser le MOSFET comme relais pour assurer voire renforcer la propagation d'un signal (fig. 4.4).

Dans ce schéma, comme dans tous ceux qui suivront ⁽¹⁾, le trait supérieur représente la ligne d'alimentation ([A] = 5 V) et le trait inférieur, la ligne de masse ([C] = 0 V). Les signaux qui nous intéressent sont propagés sur une ligne médiane, qui aboutit à la grille du MOSFET et qui repart vers la borne [B]. Toute l'action se passe dans les lignes verticales qui vont de l'alimentation à la masse en traversant le MOSFET.

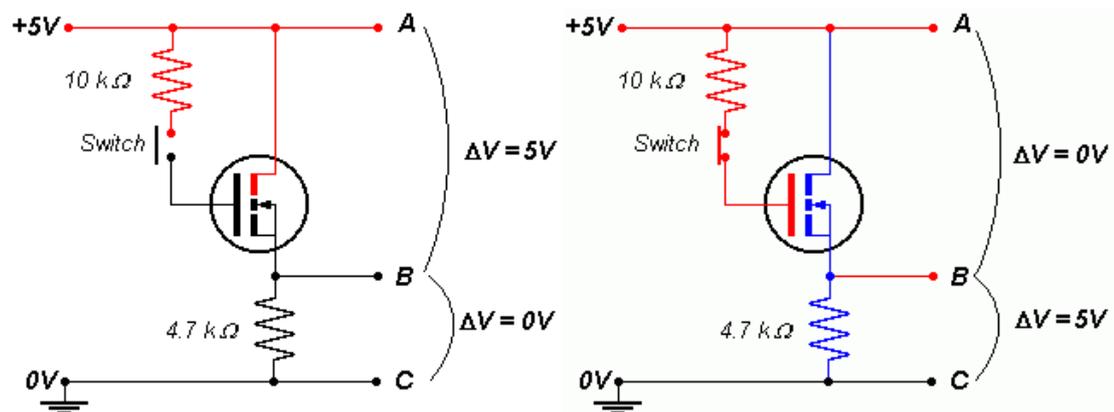


fig. 4.4 Montage simplifié d'un ampli opérationnel (OFF / ON)

Si l'interrupteur (*ang.*: *switch*) est ouvert (fig. 4.4 gauche), aucune tension n'est appliquée à la grille et le MOSFET ne laisse pas passer le courant. La borne [A] est à 5 V mais il n'y a aucune différence de potentiel ($\Delta V = 0$ V) entre la borne [C] (la masse) et la borne [B]. En effet, comme aucun courant ne circule dans la branche, il n'y a pas de chute de tension dans la résistance de 4.7 kΩ.

Si l'interrupteur est fermé (fig. 4.4 droite), le MOSFET devient passant. Le courant qui circule provoque une différence de potentiel dans la résistance. Dès lors, la borne [A] étant à 5 V et la borne [C] étant toujours à la masse à la masse (0 V), la borne [B] affiche le même potentiel que la borne [A], soit 5 V. Il y a donc maintenant une différence de potentiel ($\Delta V = 5$ V) entre les bornes [C] et [B].

Les deux cas sont résumés sur le tableau ci-dessous ⁽²⁾ :

Grille	[B]-[C]
0 V	0 V
5 V	5 V

On constate que le signal (la tension) qui arrive sur la grille est transmis à la sortie [B]. Le montage se comporte comme un propagateur voire comme un amplificateur de signal.

La résistance de 4.7 kΩ sert à limiter le courant dans la branche verticale. En vertu de la loi d'Ohm, on a

1 Attention, il s'agit toujours de schémas de principe basés sur la technologie RTL (Resistor-Transistor-Logic), très simplifiés par rapport aux schémas réels.
2 Par convention, les potentiels sont mesurés par rapport à la masse.

$$I = \frac{U}{R} = \frac{5 \text{ V}}{4700 \Omega} \cong 1 \text{ mA}$$

La puissance dissipée dans la résistance est donnée par

$$P = U \cdot I = 5 \text{ V} \cdot 1 \text{ mA} = 5 \text{ mW}$$

Une puissance de quelques milliwatts peut sembler ridiculement faible mais il ne faut pas perdre de vue qu'un circuit intégré contient vite des millions de portes. La puissance totale atteint facilement la centaine de watts.

Dans un schéma d'électronique digitale, on ne trace pas les lignes d'alimentation, mais uniquement celles qui véhiculent le signal. L'ampli est simplement représenté par un triangle dans lequel le signal entre par la base (par la gauche) et sort par la pointe (par la droite) (fig. 4.5).



fig. 4.5 Un ampli-op

4.4. La porte NOT

En électronique digitale l'opération NOT est réalisée par un **inverseur** (ang.: *invert*) appelé **NOT Gate** (fr.: *porte NON*) schématisé ci-dessous (fig. 4.6) :

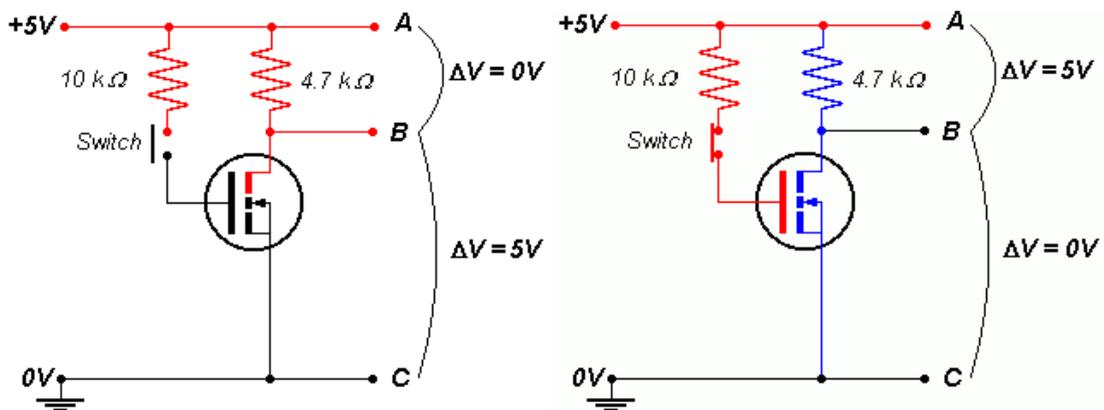


fig. 4.6 Un circuit NOT (OFF / ON)

Le montage est presque identique à celui de l'ampli sauf que la résistance de 4.7 kΩ est maintenant montée en amont du MOSFET (selon le sens conventionnel du courant)

A nouveau, si l'interrupteur est ouvert (fig. 4.6 gauche), aucune tension n'est appliquée à la grille et le MOSFET ne laisse pas passer le courant. La borne [A] est à 5 V et la borne [B] aussi car, comme aucun courant ne circule dans la branche, il n'y a pas de différence de potentiel aux bornes de la résistance. Dès lors on mesure une différence de potentiel de 5 V entre les bornes [B] et [C]

Par contre, si l'interrupteur est fermé (fig. 4.6 droite), le MOSFET devient passant et les deux extrémités de la résistance affichent une différence de potentiel. Dès lors les bornes [A] et [B] affichent une différence de 5 V tandis que la différence de potentiel entre [B] et [C] est nulle.

Les deux cas sont résumés sur le tableau ci-dessous :

Grille	[B]-[C]
0 V	5 V
5 V	0 V

Le montage se comporte comme un inverseur du signal qui arrive sur la grille.

La porte NOT est schématisée ci-dessous (fig. 4.7). Notez bien le petit rond à la pointe du triangle : en toute rigueur, c'est lui qui signifie "NOT"; le triangle, lui, représente un amplificateur



fig. 4.7 Une porte NOT

4.5. La porte AND

L'opérateur AND est représenté par un câblage en série de deux MOSFET (fig. 4.8) :

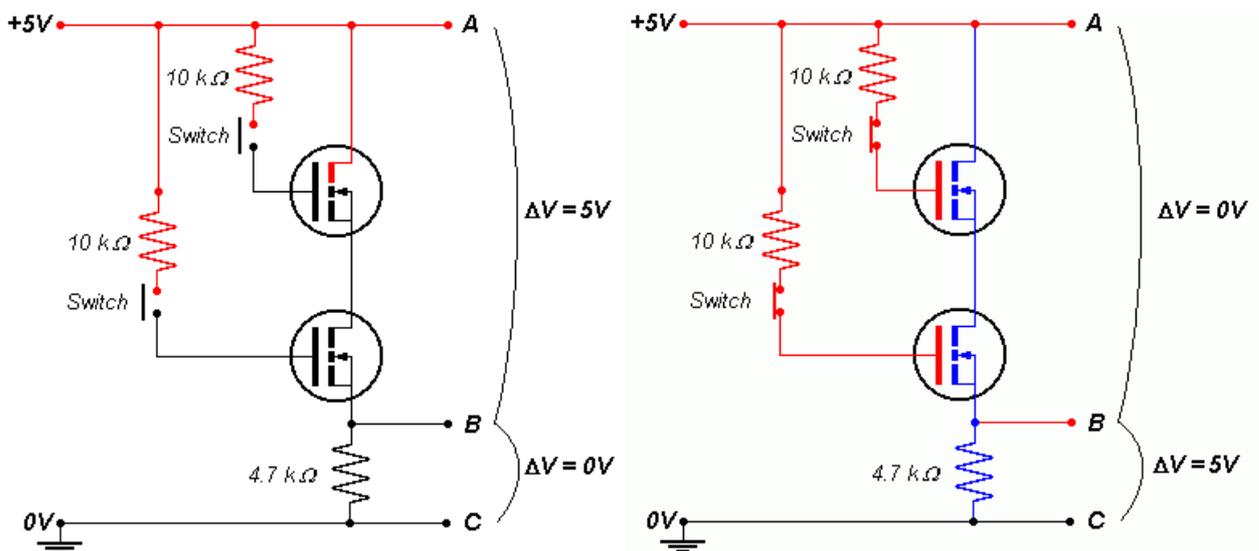


fig. 4.8 Un circuit AND (OFF / ON)

Le lecteur n'aura aucune peine à en comprendre le fonctionnement à la lumière de ce qui a été dit dans les paragraphes précédents.

La porte AND (**AND Gate**) est schématisée ci-dessous (fig. 4.9) :

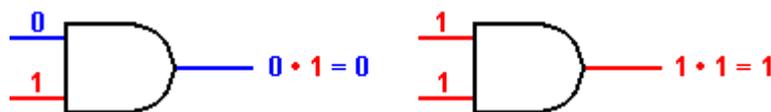


fig. 4.9 Une porte AND

Les deux signaux ou tensions à combiner entrent dans la porte par la base (par la gauche) et le signal résultant sort par la partie arrondie (par la droite). Pour qu'un signal ou une tension apparaisse à la sortie, il faut qu'un signal soit présent sur chacun des deux connecteurs d'entrée.

4.6. La porte NAND

L'ampli vu plus haut avait été transformé en porte NOT par un simple déplacement de la résistance en amont du MOSFET. C'est le même principe qui s'applique pour transformer une porte AND en porte NAND (fig. 4.10).

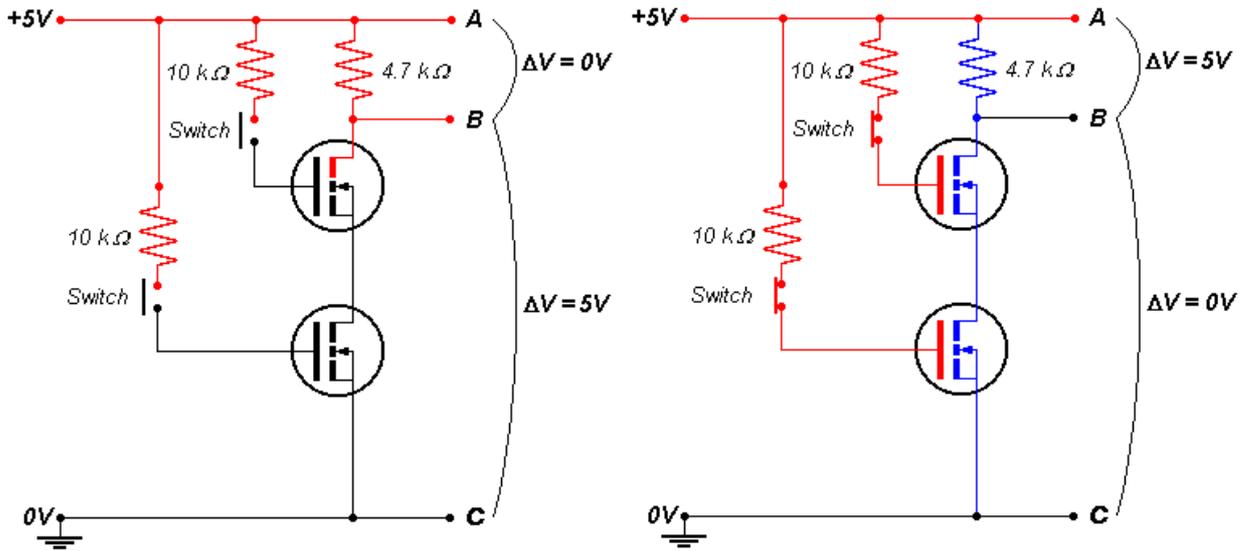


fig. 4.10 Un circuit NAND (OFF / ON)

Le symbole de la porte NAND (**NAND Gate**) est présenté ci-dessous (fig. 4.11) :

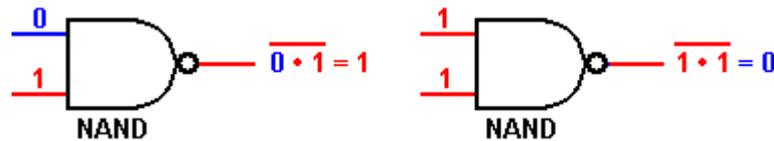


fig. 4.11 Une porte NAND

Les deux signaux ou tensions à combiner entrent dans la porte par la base (par la gauche) et le signal résultant sort par le petit rond (à droite). Comme dans le cas de la porte NOT, c'est ce petit rond qui représente l'inversion du signal. Un signal ou une tension apparaît toujours à la sortie, sauf quand les deux connecteurs d'entrée sont alimentés.

4.7. La porte OR

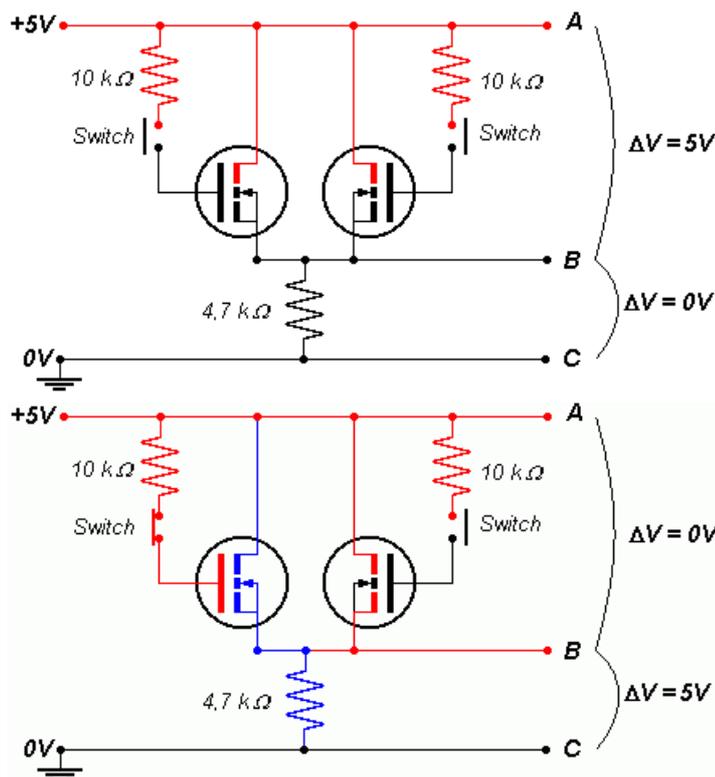


fig. 4.12 Un circuit OR (OFF / ON)

L'opérateur OR est implémenté grâce à un câblage en parallèle de deux MOSFET (fig. 4.12). Le raisonnement est identique à celui utilisé pour les portes précédentes.

Le schéma (fig. 4.13) montre le symbole de la porte OR (**OR Gate**) :

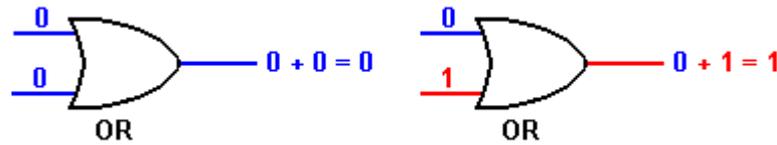


fig. 4.13 Une porte OR

Comme pour les autres portes, les deux signaux ou tensions à combiner entrent dans la porte par la base incurvée (par la gauche) et le signal résultant sort par la pointe de l'ogive (par la droite). Pour qu'un signal ou une tension apparaisse à la sortie, il faut et il suffit qu'un signal soit présent sur l'un des deux connecteurs d'entrée.

4.8. La porte NOR

A nouveau, la transformation d'une porte OR en porte NOR se fait simplement en déplaçant la résistance (fig. 4.14) :

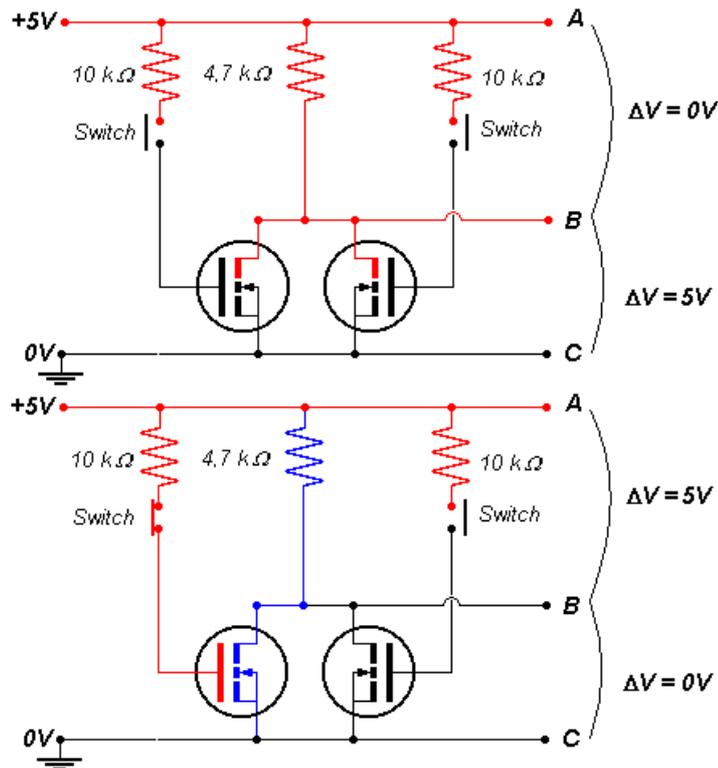


fig. 4.14 Un circuit NOR (OFF / ON)

De même, il suffit d'ajouter un petit rond à l'extrémité du symbole pour transformer une porte OR en porte NOR (**NOR Gate**) (fig. 4.15) :

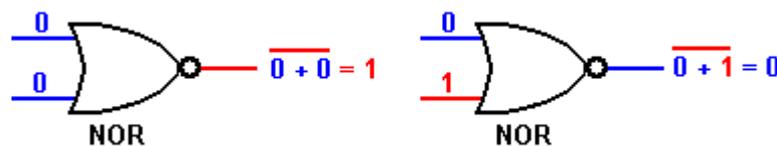


fig. 4.15 Une porte NOR

Les deux signaux ou tensions à combiner entrent dans la porte par la base (par la gauche) et le signal résultant sort par le petit rond (par la droite). Un signal ou une tension apparaît à la sortie uniquement si aucun signal n'est présent à l'entrée.

4.9. La porte XOR

La réalisation d'un circuit XOR est plus subtile. La solution présentée ici utilise la tautologie définie plus haut :

$$X \text{ XOR } Y = [(\text{NOT } X) \text{ AND } Y] \text{ OR } [X \text{ AND } (\text{NOT } Y)]$$

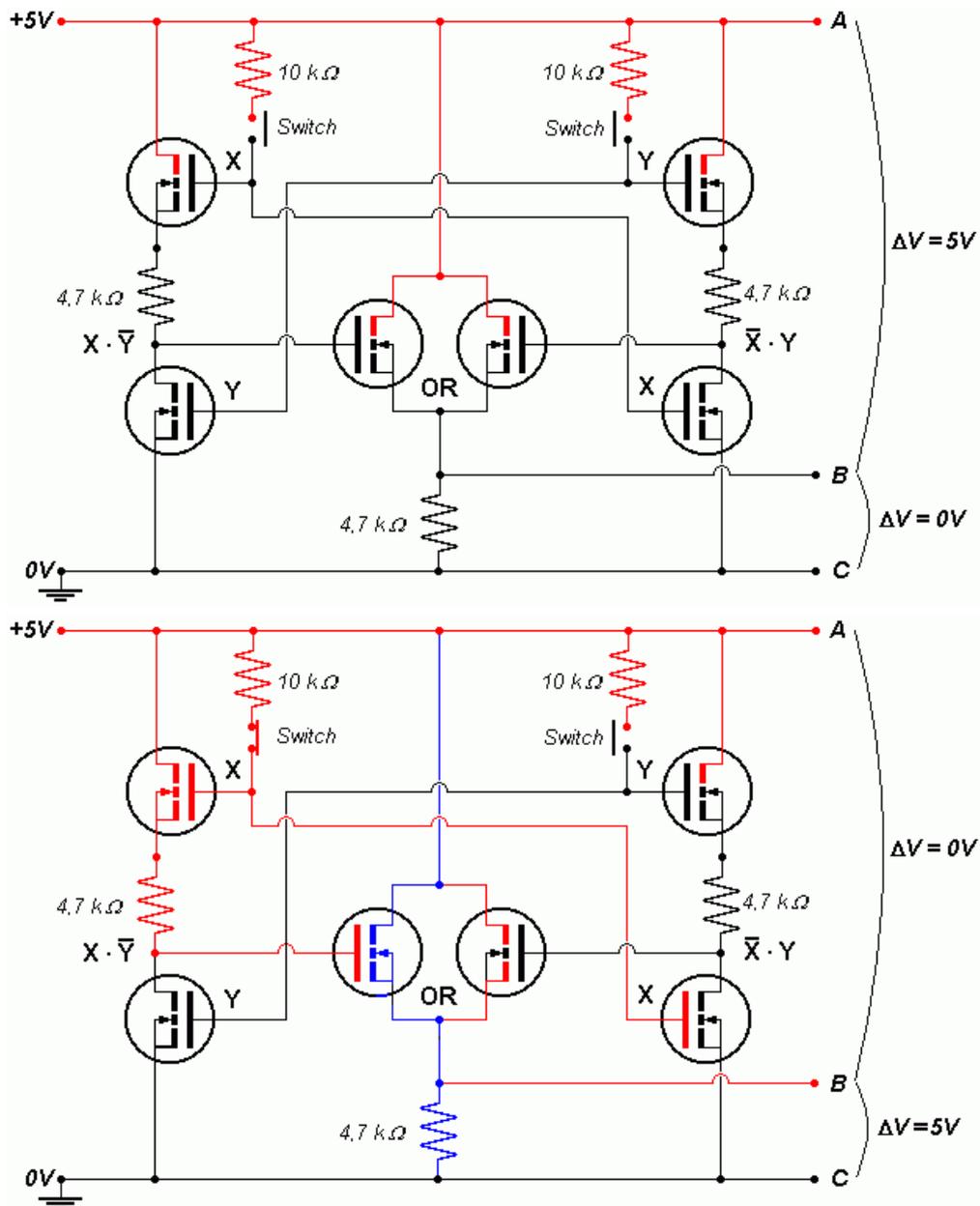


fig. 4.16 Un circuit XOR (OFF / ON)

Le circuit de la fig. 4.16 est symétrique. Les deux MOSFET de gauche et la résistance intermédiaire implémentent le premier membre de l'équation logique :

$$(\text{NOT } X) \text{ AND } Y$$

tandis que les deux MOSFET de droite et la résistance intermédiaire implémentent le deuxième membre :

$$X \text{ AND } (\text{NOT } Y)$$

Les deux tensions sont récupérées pour activer les grilles des deux MOSFET centraux qui implémentent la fonction OR vue plus haut.

Les MOSFET diagonaux [X] sont activés par l'interrupteur de gauche, tandis que la diagonale [Y] est activée par celui de droite.

Le symbole de la porte XOR (**XOR Gate**) ressemble à celui de la porte OR mais la base curviligne comprend deux traits (fig. 4.17) :

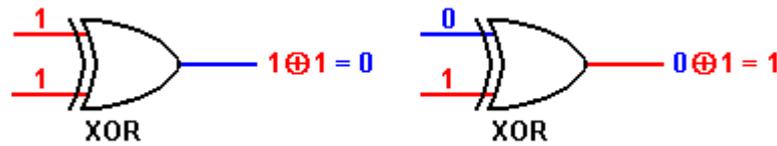


fig. 4.17 Une porte XOR

Les deux signaux ou tensions à combiner entrent dans la porte par la base (par la gauche) et le signal résultant sort par la partie pointue (par la droite). Pour qu'un signal ou une tension apparaisse à la sortie, il faut et il suffit que les deux signaux à l'entrée soient différents.

4.10. Porte tri-state

Les portes discutées dans les paragraphes précédents présentent un problème potentiel : quel que soit l'état des MOSFET, la sortie [B] est toujours connectée soit à la ligne d'alimentation (version normale), soit à la ligne de masse (version NOT). En d'autres mots, elle n'est jamais vraiment déconnectée du circuit.

La porte tri-state permet de résoudre à ce problème, en isolant totalement un étage du reste du circuit (fig. 4.18) :

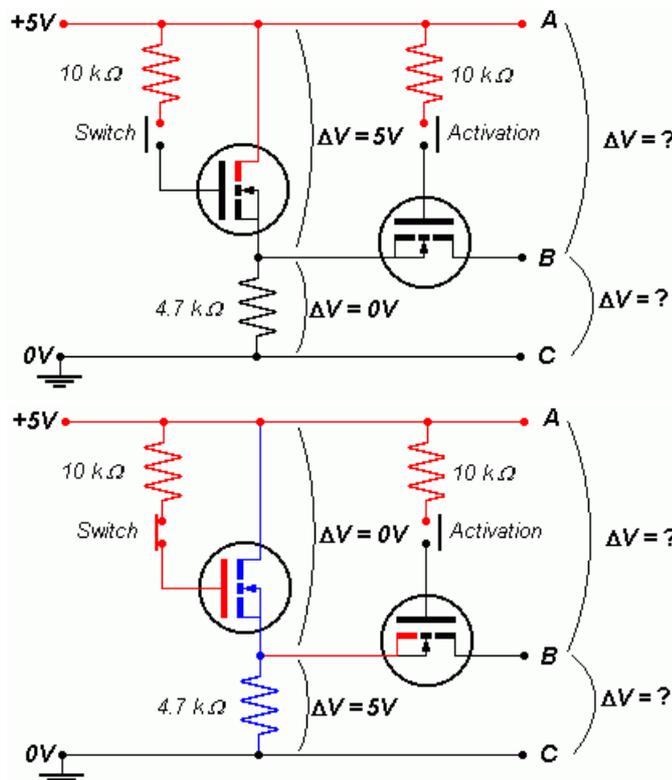


fig. 4.18 Un circuit Tri-state (OFF / Z)

Le premier étage est un ampli-op classique mais sa sortie vers [B] est bloquée par un deuxième MOSFET tant que celui-ci n'est pas activé. En absence d'activation, la borne [B] se comporte comme un conducteur qui n'est relié à rien. Elle est dite "flottante" (*ang.: floating*) et son potentiel est indéterminé.

Les différents cas sont résumés sur le tableau ci-dessous :

Grille	Activ	[B]-[C]
0 V	0 V	Z
5 V	0 V	Z
0 V	5 V	0 V
5 V	5 V	5 V

La logique de la porte tri-state ne doit pas être confondue avec celle de la porte AND. Il est vrai que la sortie n'est activée que si les deux MOSFET sont activés mais la différence est que, inactivée, la sortie présente une grande impédance (entendez : une résistance infinie) et ne transmet *aucun* signal.

Le symbole de la porte tri-state est présenté ci-dessous (fig. 4.19). Le signal entre par la base et sort par la pointe tandis que l'activation de fait par l'entrée latérale.

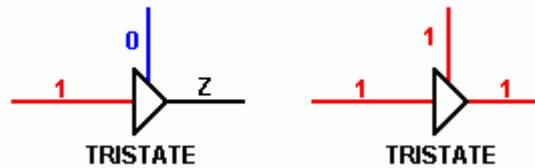


fig. 4.19 Une porte Tri-state

4.11. Tautologies

Lors des câblages électroniques, les tautologies permettent de remplacer un circuit par un autre plus simple ou n'utilisant que certaines pièces disponibles :

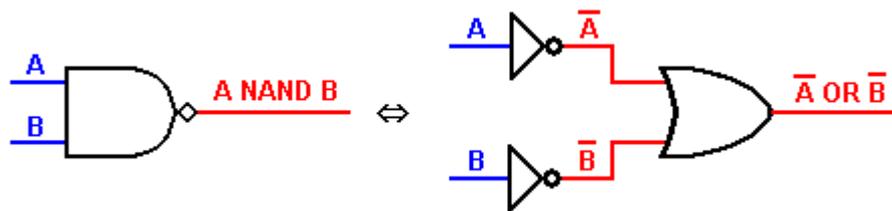


fig. 4.20 Montage équivalent à une porte NAND

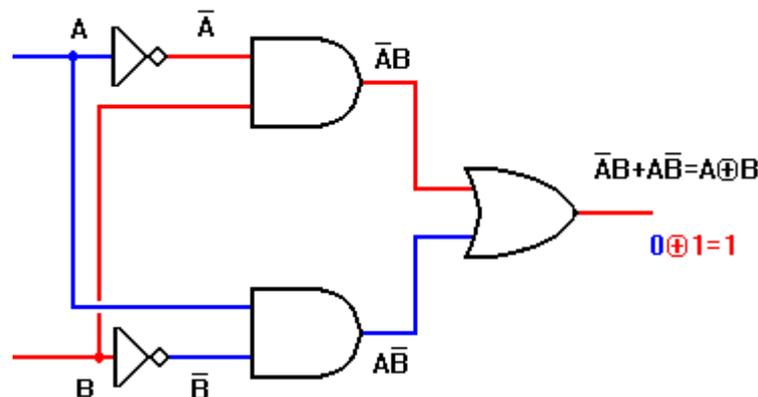
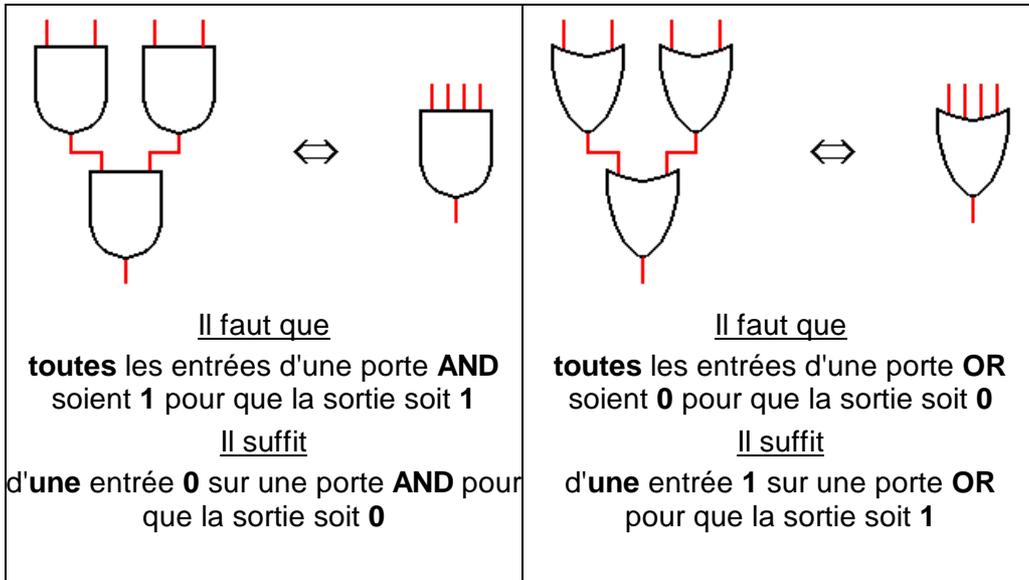


fig. 4.21 Montage équivalent à une porte XOR

On peut aussi démontrer que toutes les portes peuvent être construites uniquement à partir de portes NOR beaucoup plus économiques (voir exercices).

4.12. Cascades AND et OR

Il arrivera fréquemment que nous devons utiliser des cascades de portes AND ou de portes OR. Celles-ci peuvent être remplacées (symbolisées) par une seule porte à plusieurs entrées et une sortie :



5. Quelques circuits

5.1. Position du problème

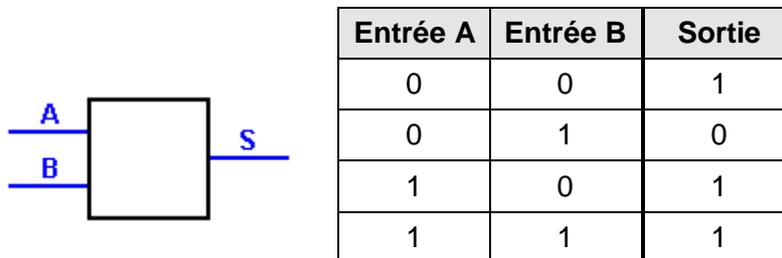
Dans ce chapitre, nous entrons au cœur du chip de silicium.

Nous allons illustrer la conception des différents circuits électroniques qui permettent d'aller de la saisie au clavier jusqu'à l'affichage du résultat en passant par la réalisation de l'opération demandée.

5.2. Retrouver la logique d'un tableau

Nous serons souvent amenés à construire un tableau à partir de l'énoncé des différents cas possibles. Tout le problème consiste alors à retrouver la logique sous-jacente et à l'exprimer soit dans un algorithme, soit dans un circuit.

Prenons l'exemple du circuit (boîte noire) et de la table ci-dessous :



Admettons que, grâce à un multimètre ou un oscilloscope, nous ayons pu voir que

- quand il n'y a pas de signal en A ni en B, on a un signal en S;
- quand il n'y a pas de signal en A mais bien en B, on n'a pas de signal en S;
- quand il y a un signal en A mais pas en B, on a un signal en S;
- quand il y a un signal en A et en B, on a un signal en S.

Nous avons noté ces résultats dans la table.

Ecrivons une expression logique qui rend compte de ces résultats. Considérons uniquement les lignes pour lesquelles la sortie vaut 1. On a 1 à la sortie si...

$$S = (\text{NOT } A \text{ AND NOT } B) \text{ OR } (A \text{ AND NOT } B) \text{ OR } (A \text{ AND } B)$$

$$S = (\bar{A} \cdot \bar{B}) + (A \cdot \bar{B}) + (A \cdot B)$$

Appliquons maintenant les égalités du point 3.3 pour essayer de simplifier l'expression. Par exemple, on peut mettre A en évidence entre le deuxième et le troisième groupe en appliquant la règle [11].

$$S = (\bar{A} \cdot \bar{B}) + A \cdot (\bar{B} + B)$$

Grâce aux règles [7] et [2], le deuxième terme se simplifie :

$$S = (\bar{A} \cdot \bar{B}) + A$$

Distribuons + (OR) sur • (AND) en utilisant la règle [11]

$$S = (\bar{A} + A) \cdot (\bar{B} + A)$$

Grâce à la règle [7], le premier terme se simplifie. On applique la règle [2] et on a :

$$S = \bar{B} + A$$

Finalement, on s'aperçoit que la logique de la boîte noire s'exprime par

$$S = \text{NOT } B \text{ OR } A = A \text{ OR NOT } B$$

Nous aurions tout aussi bien pu commencer le raisonnement en considérant les lignes pour lesquelles la sortie vaut 0 :

$$\text{NOT } S = (\text{NOT } A) \text{ AND } B$$

$$\bar{S} = \bar{A} \cdot B$$

Inversons la logique en prenant le contraire des deux membres :

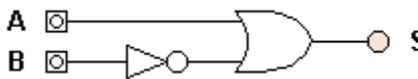
$$\bar{\bar{S}} = \bar{\bar{A} \cdot B}$$

Appliquons la règle [5] à gauche et la règle [8] qui transforme la négation d'un • (AND) en + (OR) à droite :

$$S = \bar{\bar{A}} + \bar{B}$$

$$S = A + \bar{B}$$

Nous arrivons beaucoup plus rapidement à la même conclusion !



Entrée A	Entrée B	A	OR	NOT B	↔	S
0	0	0	1	1	V	1
0	1	0	0	0	V	0
1	0	1	1	1	V	1
1	1	1	1	0	V	1

5.3. Le clavier

Le clavier est un système dans lequel la frappe d'une seule touche provoque la génération et l'envoi du motif binaire correspondant à la touche frappée.

Prenons l'exemple du clavier d'une calculatrice ou du pavé numérique d'un clavier classique : quand on frappe la touche [5], un circuit doit générer un byte contenant la valeur binaire 00001001.

Commençons par établir la table de vérité des sorties en fonction de la touche frappée. Pour simplifier, nous ne considérons que des motifs codés sur 4 bits, soit $5_d = 1001_b$, mais il est très facile d'étendre le raisonnement.

Puisque chaque frappe génère une sortie codée sur 4 bits, il nous faut 4 circuits distincts, un pour chacun des bits.

Touche	S3	S2	S1	S0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

En relisant le tableau colonne par colonne, nous établissons les équations logiques de chacune des sorties :

S0:	"1" OR "3" OR "5" OR "7" OR "9"
S1:	"2" OR "3" OR "6" OR "7"
S2:	"4" OR "5" OR "6" OR "7"
S3:	"8" OR "9"

Le circuit de notre clavier utilise exclusivement des portes OR. Si nous croisons les 10 entrées correspondant aux 10 touches avec les 4 sorties correspondant aux 4 bits, le cerveau de notre clavier ressemble à ceci :

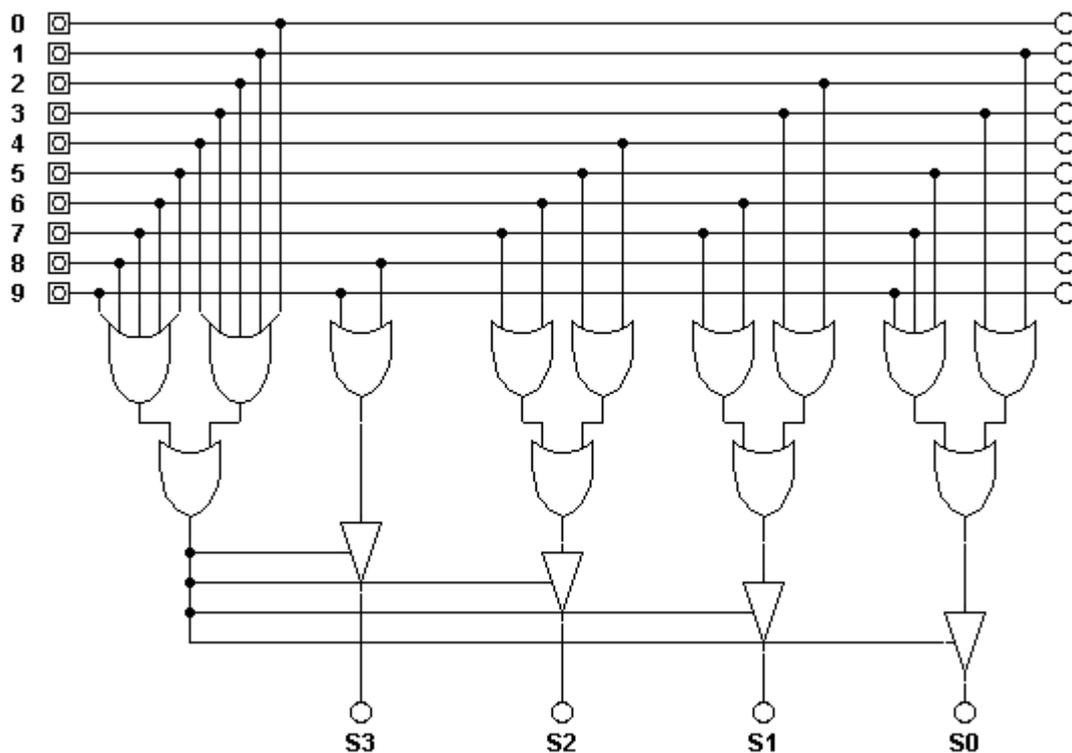


fig. 5.1 Clavier élémentaire à codage sur 4 bits

La cascade de portes OR à gauche commande les portes tri-state qui n'ouvrent le circuit que si une des touches du clavier a été frappée.

En réalité, les claviers d'ordinateurs n'envoient pas directement le code ASCII du caractère frappé mais bien un "scan code" d'un byte qui représente la position de la touche sur le clavier. La conversion du "scan code" en code ASCII est réalisée par le système d'exploitation. Ceci ne change en rien le principe de génération des codes expliqué ci-dessus.

Dans le même ordre d'idées, un code binaire est également associé à chacune des touches arithmétiques (+ - * /)

5.4. L'additionneur 1 bit + 1 bit

Le problème suivant consiste à transformer une opération *arithmétique* en une opération *logique*. Commençons par le cas le plus simple : l'addition de deux bits. Il y a quatre cas possibles :

A_0	0	0	1	1
B_0	+0	+1	+0	+1
	0	1	1	10

Il nous faut considérer deux choses, choses que nous connaissons depuis la première année de l'école primaire : le **nombre que l'on pose** (*ang.: result*) et le **nombre que l'on reporte** (*ang.: carry*). Et la règle qui définit le nombre que l'on pose n'est pas la même que celle qui détermine le report !

A	B	Poser	A	B	Reporter
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

Puisqu'il faut considérer deux sorties, qui sont définies par deux tableaux différents, l'additionneur comportera deux circuits en parallèle ⁽¹⁾.

Le tableau "Poser" nous fait immédiatement penser à l'opérateur **XOR**.
Le tableau "Reporter" nous fait penser à l'opérateur **AND**.

Dans le circuit électronique, nous devons dédoubler les signaux venant de A et de B pour les conduire respectivement vers le circuit qui détermine le nombre à poser et vers le circuit qui détermine le nombre à reporter :

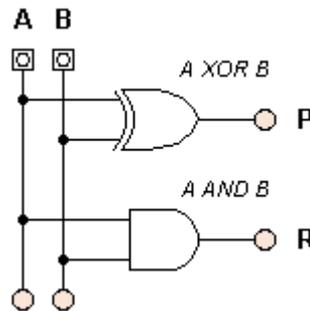


fig. 5.2 Demi-additionneur "Poser" et "Reporter"

Ce circuit est appelé demi-additionneur (*ang.: half adder*). Nous allons tout de suite comprendre pourquoi.

5.5. L'additionneur 1 bit + 1 bit + report

En général, les nombres que nous allons additionner sont codés sur plusieurs bits.

R	1 1 1 0	0 0 1
A	0 1 0 1	0 1 0 1
B	0 1 1 1	1 0 0 1
S	1 1 0 0	1 1 1 0

1 Une erreur fréquente est d'essayer de traiter en même temps le bit "Poser" et le bit "Reporter". Il ne faut pas croire que le même circuit doit gérer les deux sorties.

Le problème que nous avons résolu au paragraphe précédent concerne juste la colonne du premier bit à l'extrémité droite appelé bit de poids faible (*ang.*: **Least Significant Bit**). Pour chacune des autres colonnes, il faut considérer un report éventuel venant de la colonne précédente. Pour l'addition des nombres d'une colonne, par exemple la colonne de rang 1, il faut considérer 8 cas au lieu de 4 :

R_1	0	1	0	1	0	1	0	1
A_1	0	0	1	1	0	0	1	1
B_1	+0	+0	+0	+0	+1	+1	+1	+1
	0	1	1	10	1	10	10	11

Nous devons concevoir un circuit à trois entrées (R_1 , A_1 , B_1) et deux sorties (P_1 , R_2). L'avantage, c'est qu'il suffira de reproduire ce circuit autant de fois qu'il y a de rangs de bits à additionner.

Le problème se simplifie si nous considérons que l'addition est une opération en deux étapes :

Etape 1	R	1110 0010
	A	0101 0101
Etape 2	RI	1011 0111
	B	0111 1001
	S	1100 1110

- pour le bit "Poser", on considère d'abord l'addition du bit de Report (soit R_0) et du bit venant de A (soit A_0) (étape 1). On considère ensuite l'addition de ce résultat intermédiaire (RI) avec le bit venant de B (soit B_0) (étape 2).
- pour le bit "Reporter", on doit considérer soit le report issu de l'étape 1, soit le bit du report issu de l'étape 2. L'exemple et le tableau des 8 cas montrent qu'il n'y a jamais qu'un seul bit de report : s'il y a eu report à l'étape 1, il n'y a pas de report à l'étape 2.

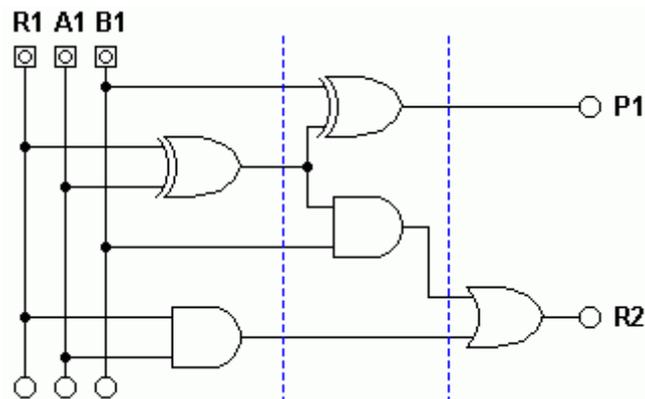


fig. 5.3 Additionneur "1 bit" complet

La figure (fig. 5.3) montre le circuit résultant appelé additionneur complet (*ang.*: *full adder*) :

- le premier étage additionne le report [R_1] venant du rang précédent avec le bit du premier nombre [A_1]. En haut, la porte XOR calcule le bit à poser [P]; en bas, la porte AND calcule un report éventuel [R];
- le deuxième étage additionne le bit [P] issu du premier étage avec le bit du deuxième nombre [B_1] (XOR). En bas, il calcule un report éventuel venant de ces deux bits [R] et [B_1] (AND);
- le troisième étage compare les bits de report issus des deux étages précédents pour construire le bit [R_2] qui sera reporté au rang suivant.

Finalement, les équations logiques des deux sorties sont :

$$P1 = (B1 \text{ XOR } (A1 \text{ XOR } R1))$$

$$R2 = (B1 \text{ AND } (A1 \text{ XOR } R1)) \text{ OR } (A1 \text{ AND } R1)$$

Le schéma ci-dessus est souvent trop encombrant pour être utilisé tel quel dans des diagrammes plus complexes. On le remplace alors par une boîte noire à trois entrées (A, B, C_{in}) et deux sorties (Sum, C_{out}) (fig. 5.4) :

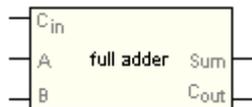


fig. 5.4 Boîte noire d'addition 1 bit (1)

5.6. L'additionneur 1 byte + 1 byte

Nous voici enfin en mesure d'additionner deux bytes. Il suffit d'arranger une cascade de 8 additionneurs 1 bit dans laquelle chaque sortie "Report" (C_{out}) est ramenée vers l'entrée de la boîte suivante (C_{in}) (fig. 5.5) :

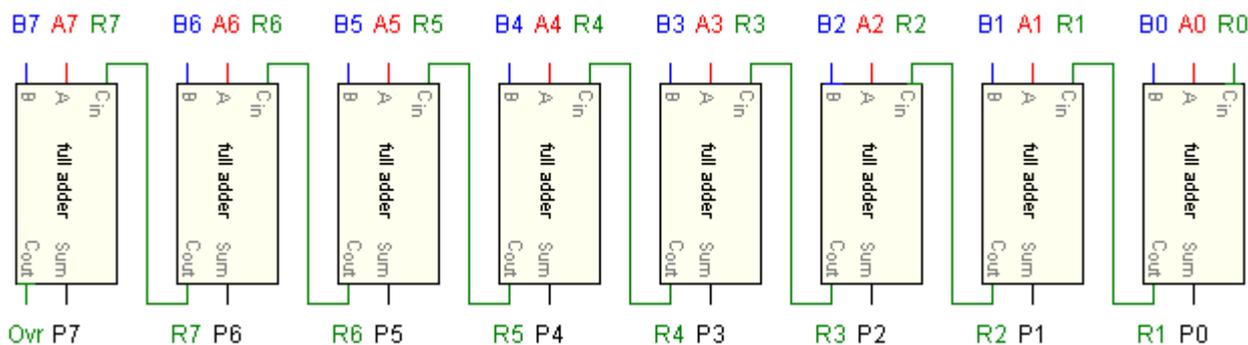


fig. 5.5 Addition de deux bytes

Le report à l'entrée de la boîte de rang 0 [R₀] est mis à 0 tandis que le report issu de la dernière boîte [Ovr] signale un dépassement de capacité (*ang.:* overflow) éventuel.

Notons qu'il faut attendre la fin du calcul d'un rang pour pouvoir calculer le rang suivant, exactement comme dans le cas du calcul manuel. Il n'est pas possible de calculer tous les rangs d'un seul coup. Le problème sera résolu à l'aide d'une horloge qui cadencera la succession des opérations (fig. 5.6).

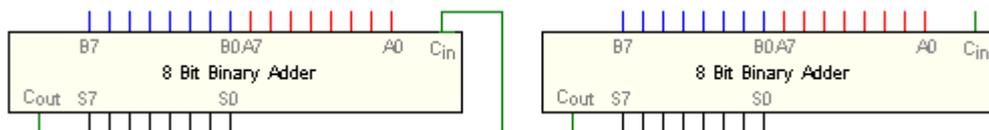


fig. 5.6 Addition de deux nombres de 2 bytes

On peut évidemment combiner plusieurs de ces boîtes pour faire des additions de nombres à 16 ou 32 bits.

5.7. L'inverseur de signe

Nous savons que pour inverser le signe d'un nombre, il faut inverser tous les bits puis ajouter 1.

1 C_{in} et C_{out} remplacent R₀ et R₁

Nous pourrions utiliser notre additionneur de la fig. 5.5 à condition d'inverser tous les bits du premier byte (byte A) avant de l'introduire dans l'additionneur et de prendre 01_h pour le deuxième byte (byte B).

Inverser tous les bits n'est pas très compliqué : il suffit de mettre une porte NOT avant chaque entrée.

Par contre, au lieu de mettre le byte B à 01_h, on pourrait le laisser à 00_h et utiliser plutôt le bit [R₀] de report à l'entrée de la première boîte (fig. 5.7) :

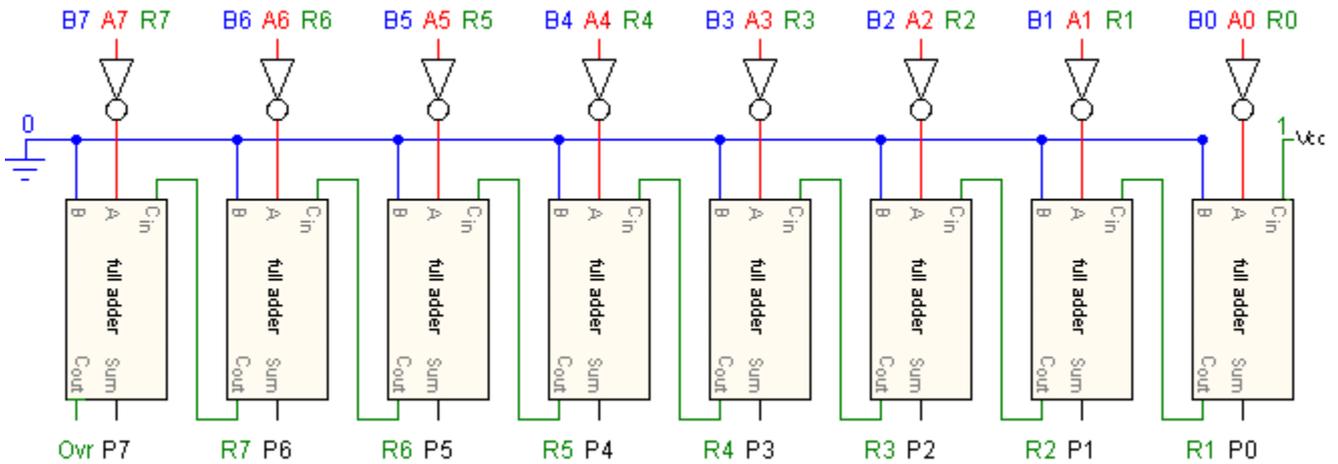


fig. 5.7 Inverseur de signe (première version)

En examinant bien cet arrangement, on s'aperçoit qu'il effectue l'opération

$$P = (-A) + 0 = -A$$

5.8. L'additionneur-soustracteur

A la réflexion, dans le schéma de la fig. 5.7, on peut parfaitement utiliser le byte B pour envoyer un autre nombre et ainsi réaliser l'opération

$$P = (-A) + B = B - A$$

Autrement dit, nous avons devant nous un circuit capable d'effectuer une soustraction. Il suffirait de mettre les portes NOT sur les entrées B plutôt que sur les entrées A pour effectuer

$$P = A + (-B) = A - B$$

L'avantage, c'est qu'on utilise un circuit bien connu pour effectuer la soustraction. Par contre, ce qui est désolant, c'est qu'il nous faut un circuit sans les portes NOT pour faire l'addition et un circuit avec les portes NOT pour faire la soustraction.

Replongeons-nous dans les tableaux de vérité des diverses fonctions logiques. Examinons plus particulièrement la fonction XOR :

M	N	M ⊕ N
0	0	0
0	1	1
1	0	1
1	1	0

- Si M est nul (lignes 1 et 2), la fonction XOR ne modifie pas la valeur de N. Elle se comporte comme si elle était transparente
- Si M vaut 1 (lignes 3 et 4), la fonction XOR inverse la valeur de N. Elle se comporte comme une fonction NOT.

Autrement dit, dans le schéma de notre inverseur de signe, nous pouvons utiliser des portes XOR à la place des portes NOT. Nous enverrons 0 ou 1 sur l'une des entrées selon que nous voulons faire une addition ou une soustraction. Or il se fait que, lors d'une addition, le bit de report introduit dans le rang 0 vaut 0 alors que ce même bit vaut 1 en cas de soustraction. Il suffit donc de récupérer ce bit et de le renvoyer vers toutes les portes XOR :

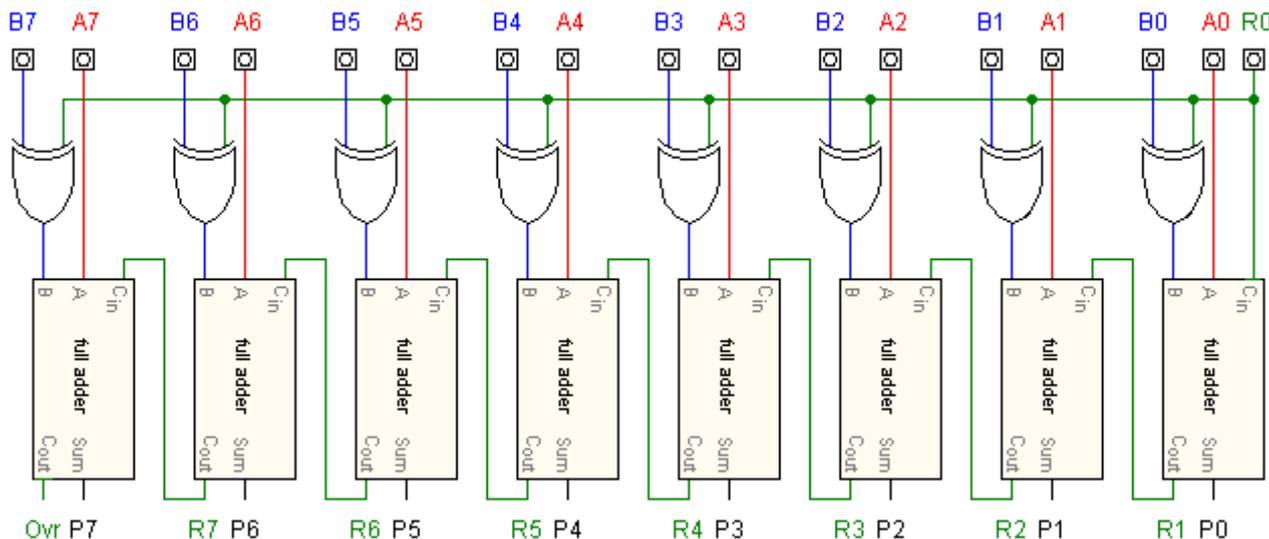


fig. 5.8 Module Additionneur ($R_0=0$) et Soustracteur ($R_0=1$)

Ce module peut encore être amélioré en utilisant un circuit "anticipateur de retenue". Nous n'en parlerons pas ici.

5.9. L'incrémenteur

L'incrémenteur ou compteur (*ang.: counter*) est un circuit qui ajoute une unité au nombre qui lui est fourni en entrée.

Il est évident qu'on peut utiliser un additionneur pour accomplir ce travail mais ce n'est pas la meilleure solution. Puisqu'on ajoute 1, on sait que tous les bits envoyés sur le byte "B" de l'additionneur sont nuls hormis le bit de rang 0. On peut donc simplifier le circuit de la fig. 5.3 qui redevient un simple incrémenteur (fig. 5.9)

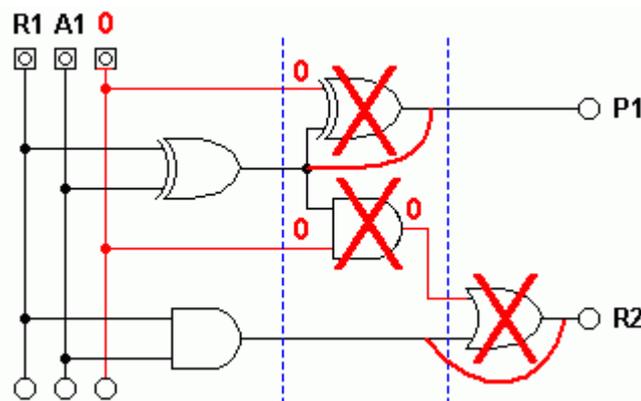


fig. 5.9 Simplification de l'additionneur

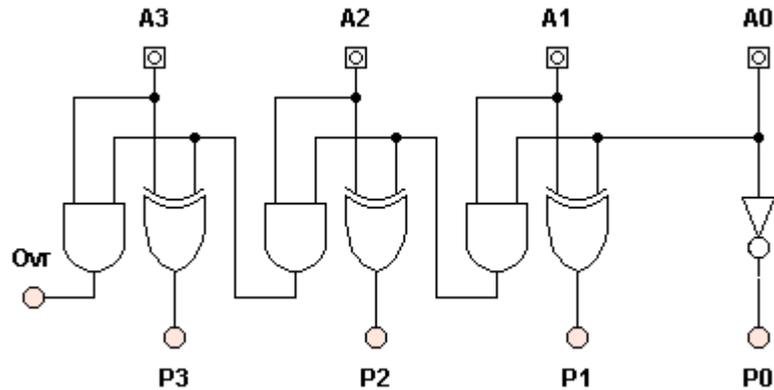


fig. 5.10 Module Incrémenteur 4 bits

En les plaçant en cascade, on obtient un incrémenteur plus simple et plus rapide que l'additionneur. Sur la fig. 5.10, le nombre binaire est introduit en [A0...A3] au et on récupère le nombre incrémenté d'une unité en [P0...3]. Le dernier report signale le dépassement de capacité.

La tentation est grande de récupérer les sorties, donc le nombre incrémenté, et de les rediriger vers les entrées, de manière à constituer un compteur. L'idée est bonne mais si nous ne prenons pas quelques précautions, le système deviendra complètement instable...

Ceci nous conduit naturellement aux concepts de mémoire et de séquenceur.

5.10. La bascule R/S

La bascule R/S est un circuit capable de mémoriser l'état d'un bit même après que ses entrées aient été déconnectées.

Elle est constituée de deux portes NOR montées en parallèle. Sa particularité est que la sortie de chaque porte est redirigée vers l'entrée de l'autre, ce qui introduit une **rétroaction** (fig. 5.11).

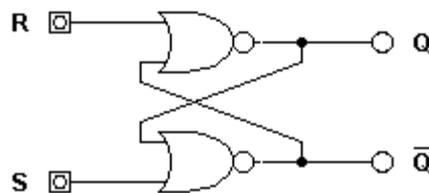


fig. 5.11 Bascule R/S

Supposons d'abord que les entrées [R] et [S] soient mises à 0. Admettons ensuite que la sortie [Q] soit à l'état 1.

Du fait de la rétroaction, le 1 est reporté à l'entrée de la deuxième porte NOR et, dès lors, sa sortie vaut 0 (d'où la notation [not Q]). Le 0 est ramené vers l'entrée de la première porte, ce qui confirme la sortie ([Q] = 1). Le circuit est donc dans un **état stable**.

Par raison de symétrie, si la sortie [Q] avait été mise à 0, nous aurions trouvé un **deuxième état stable** dans lequel la sortie [not Q] vaut 1.

Pour cette raison, ce circuit est appelé **bascule bi-stable**.

R	S	Q	not Q
0	0	1 (0)	0 (1)
0	1	1	0
1	0	0	1
1	1	?	?

Ce qui est intéressant, c'est que si nous faisons passer *momentanément* l'entrée [R] à 1, alors la sortie [Q] passe instantanément à 0 et reste dans cet état, tandis que la sortie [not Q] passe à 1 et reste dans cet état.

Si, par contre, nous faisons passer *momentanément* l'entrée [S] à 1, alors la sortie [Q] passe instantanément à 1 et reste dans cet état, tandis que la sortie [not Q] passe à 0 et reste dans cet état.

C'est pour cette raison que les entrées de la bascule sont nommées [R] (pour **Reset**) et [S] (pour **Set**).

Notons que si nous mettons les entrées [R] et [S] à 1 simultanément, le système devient instable.

Pour éviter ce risque, on n'utilise souvent qu'une seule entrée qui alimente :

- [S] directement
- [R] via une porte NOT

La bascule R/S est souvent utilisée pour mémoriser les informations entre deux étages du calculateur. On lui adjoint une porte tri-state en amont et une autre en aval de manière à la faire fonctionner comme une écluse (fig. 5.12). Ces deux portes sont commandées par une ligne parallèle R/W enable.

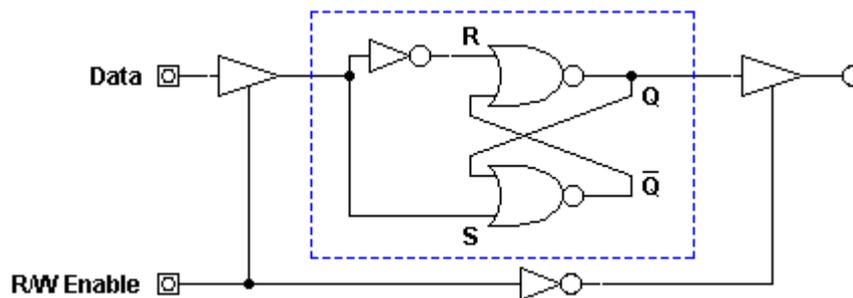


fig. 5.12 Registre R/S avec R/W enable

Lorsque l'entrée [R/W] est activée, la porte tri-state amont est active, ce qui connecte la bascule à l'entrée [Data]. Le bit présent (qu'il soit 1 ou 0) est alors stocké et "piégé" dans la bascule car la porte tri-state aval est désactivée. Le circuit est dans le mode "Ecriture" (*ang.: Write*).

Lorsque l'entrée [R/W] est désactivée, la porte tri-state amont est inactive, ce qui déconnecte la bascule et interdit l'écriture d'un autre bit. Par contre, la bascule étant dans un état stable, le bit déjà stocké dans la bascule est toujours conservé. Il peut maintenant être lu sur la ligne [Q] car la porte tri-state aval est active. Le circuit est ainsi dans le mode "Lecture" (*ang.: Read*).

Le plus souvent, l'entrée R/W enable sera pilotée par les signaux d'horloge de manière à synchroniser le flux des informations dans tout l'ordinateur.

Si nous couplons ce système avec l'incrémenteur vu plus haut, nous obtenons un compteur (voir annexe).

5.11. Le multiplicateur 2 bits x 2 bits

Après l'addition, il est logique de se pencher sur la multiplication. La table de multiplication binaire ne comporte que quatre cas possibles et il n'y a jamais de report :

A_0	0	0	1	1
B_0	x0	x1	x0	x1
	0	0	0	1

On voit qu'une simple porte AND suffit à réaliser l'opération.

Les choses se compliquent un peu quand on veut multiplier des nombres de deux bits, par exemple :

A	10
B	x11
	10
	+10_
	110

Notons que le bit qui résulte du produit des unités ($A_0 \cdot B_0$) est reporté tel quel dans le résultat final et n'intervient pas dans l'addition. Notons aussi que le plus grand résultat possible $11 \times 11 = 1001$ se note sur quatre bits.

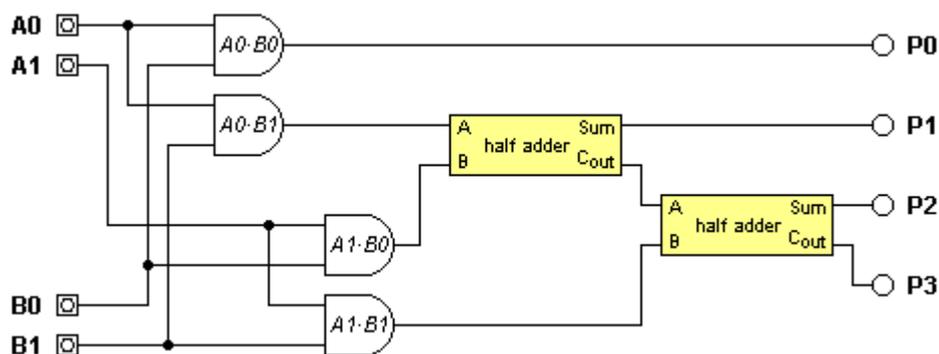


fig. 5.13 Multiplicateur 2 bits x 2 bits

On crée facilement un multiplicateur 4 bits x 4 bits en combinant quatre multiplicateurs 2 bits x 2 bits et trois additionneurs 4 bits ⁽¹⁾. En effet, chaque nombre de quatre bits peut être décomposé en deux nombres de deux bits :

$$\begin{aligned}
 A_3A_2A_1A_0 \cdot B_3B_2B_1B_0 &= (A_3A_2 \cdot 100 + A_1A_0) \cdot (B_3B_2 \cdot 100 + B_1B_0) \\
 &= (A_3A_2 \cdot B_3B_2) \cdot 10000 \\
 &\quad + (A_3A_2 \cdot B_1B_0 + A_1A_0 \cdot B_3B_2) \cdot 100 \\
 &\quad + (A_1A_0 \cdot B_1B_0)
 \end{aligned}$$

Les multiplications par 100 et 10000 sont de simples décalages de 2 et 4 bits.

On utilise la même logique pour assembler un multiplicateur 8 bits x 8 bits à partir de multiplicateurs 4 bits.

1 Mais ce n'est pas nécessairement la méthode la plus efficace du point de vue du temps de calcul.

5.12. Le décodeur d'adresse

5.12.1. Définition

Le décodeur d'adresse (*ang.*: *address decoder*) est un dispositif qui active une sortie parmi plusieurs possibles en fonction du numéro qui lui est fourni en entrée.

Un ordinateur utilise de nombreux décodeurs d'adresse, par exemple :

- pour accéder à une mémoire dont on connaît l'adresse,
- pour ouvrir le circuit de tel module arithmétique en fonction du code d'opération,
- pour sélectionner telle ou telle carte en fonction du numéro du connecteur (*ang.*: *slot*) qu'elle occupe
- pour sélectionner tel ou tel pixel d'un écran LCD

Les décodeurs d'adresse sont largement répandus dans la vie courante. On peut les comparer au système d'aiguillages placés *à l'entrée* d'une gare de triage. Son rôle consiste à activer telle ou telle voie en fonction du numéro du train à assembler. Le téléphone est un autre exemple bien connu où le décodeur d'adresse doit activer une ligne en fonction du numéro qui a été donné à l'entrée.

5.12.2. L'arbre binaire

Le principe de base du décodeur d'adresse est l'arbre binaire.

La fig. 5.14 montre un arbre binaire codé sur 4 bits. Les 16 extrémités, qui correspondent aux adresses ou lignes disponibles, sont numérotées de 0_d à 15_d , soit de 0_h à F_h ou de 0000_b à 1111_b .

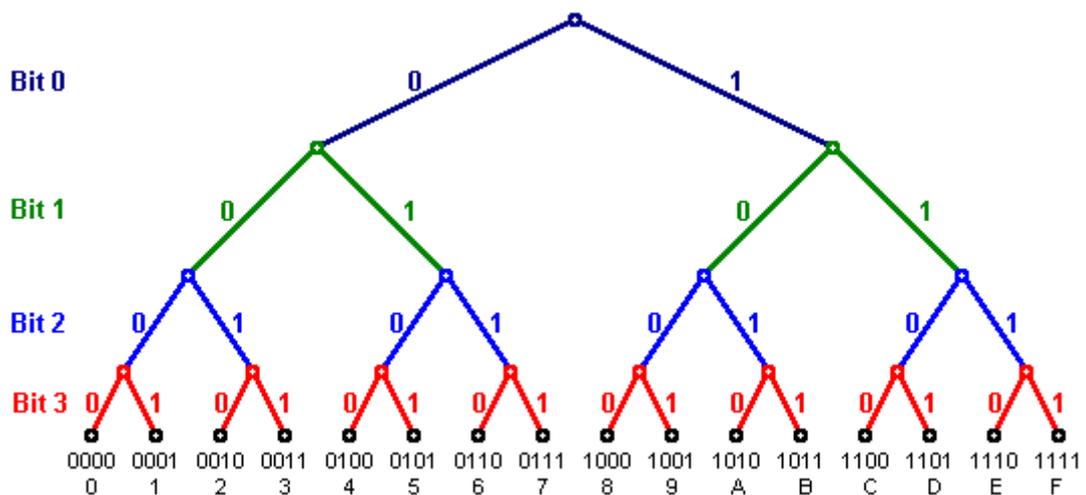


fig. 5.14 Principe d'un arbre binaire à 4 bits

L'arbre est structuré en plusieurs étages. Le nombre d'étages correspond au nombre de bits nécessaires pour coder la plus grande des adresses présentes aux extrémités ⁽¹⁾.

Chaque niveau présente un nœud et deux directions possibles ⁽²⁾. On choisit la direction en fonction de l'état du bit correspondant. Par exemple, pour arriver à

1 Ainsi, s'il y avait 200 extrémités, celles-ci seraient numérotées de 0_d à 199_d , soit de 0_b à $1100\ 0111_b$ et il faudrait 8 niveaux.

2 D'où le terme d'arbre binaire.

l'extrémité "5", on a du prendre successivement les branches 0, 1, 0, 1 or, 0101 est précisément le nombre "5" exprimée en binaire.

5.12.3. Le décodeur d'adresse à 1 bit

Voyons à présent comment construire un module qui active une adresse lorsqu'on présente son numéro à l'entrée.

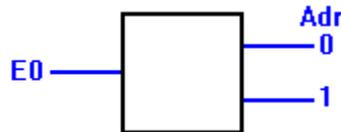


fig. 5.15 Principe d'un décodeur à 1 bit

Commençons simplement. A l'aide d'un bit, nous pouvons effectuer un choix entre deux adresses (ou lignes) portant les numéros 0 et 1 :

- si nous présentons la valeur 0 à l'entrée E_0 , la ligne 0 est sélectionnée et son état doit être porté à 1 tandis que l'état de la ligne 1 doit rester à 0;
- si nous présentons la valeur 1 à l'entrée E_0 , la ligne 1 est sélectionnée et son état est porté à 1 tandis que l'état de la ligne 0 doit rester à 0.

On peut en faire un tableau de vérité :

Entrée	Ligne 0	Ligne 1
0	1	0
1	0	1

On en déduit que la ligne 1 se comporte comme le bit d'entrée et que la ligne 0 résulte d'une opération NOT :

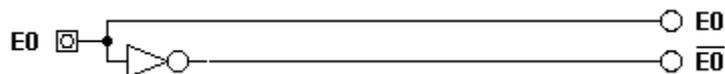


fig. 5.16 Décodeur d'adresse à 1 bit

On voit que si E_0 vaut 1, la ligne 1 transporte un 1 et la ligne 0 est à 0. Par contre, si E_0 vaut 0, alors c'est la ligne 0 qui transporte un 1 et la ligne 1 qui est à 0.

5.12.4. Le décodeur d'adresse à 2^n bits

Le principe de base - la porte NOT - étant trouvé, nous pouvons passer à un système un peu plus complexe avec deux entrées (E_0 et E_1). Avec deux bits, nous comptons de 0 à 3. Nous pouvons effectuer un choix entre quatre lignes :

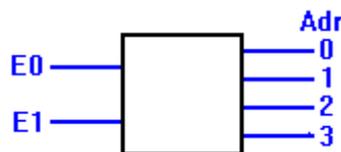


fig. 5.17 Principe d'un décodeur à 2 bits

E_1	E_0	Ligne 0	Ligne 1	Ligne 2	Ligne 3	(¹)
0	0	1	0	0	0	
0	1	0	1	0	0	
1	0	0	0	1	0	
1	1	0	0	0	1	

1 Notez la structure diagonale du tableau

On en déduit les opérations logiques ainsi qu'expliqué au point 5.2 :

Ligne 0 : $(\text{NOT } E_1) \text{ AND } (\text{NOT } E_0)$
 Ligne 1 : $(\text{NOT } E_1) \text{ AND } E_0$
 Ligne 2 : $E_1 \text{ AND } (\text{NOT } E_0)$
 Ligne 3 : $E_1 \text{ AND } E_0$

Il nous faut donc recourir à des portes AND, ce qui complique un peu le schéma précédent.

Une disposition pratique consiste à tracer un réseau de parallèles horizontales comportant toutes les lignes d'entrée (directes et NOT). Sur ces lignes, on croise un réseau de lignes verticales qui vont rechercher le signal adéquat pour le conduire à la porte AND :

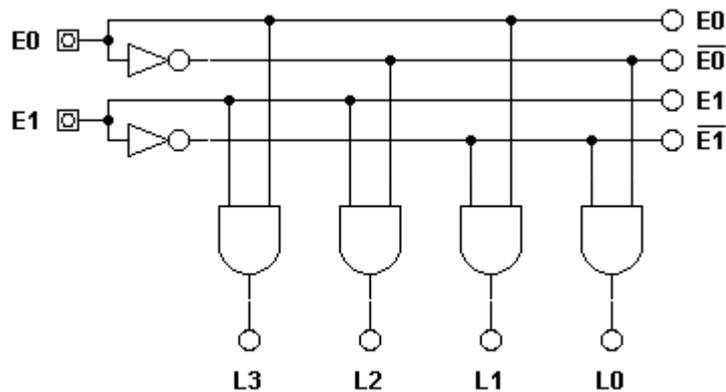


fig. 5.18 Décodeur d'adresse à 2 bits et 4 lignes

Les deux entrées E_0 et E_1 définissent le numéro de l'adresse. Ici, l'adresse est codée sur deux bits mais nous pouvons facilement étendre ce système. C'est particulièrement aisé si le nombre de bits de l'adresse est une puissance de 2.

En effet, prenons l'exemple une adresse quelconque codée sur 4 bits. Les opérations logiques qui la déterminent sont du type

Ligne 6 : $(\text{NOT } E_3) \text{ AND } E_2 \text{ AND } E_1 \text{ AND } (\text{NOT } E_0)$
 $= ((\text{NOT } E_3) \text{ AND } E_2) \text{ AND } (E_1 \text{ AND } (\text{NOT } E_0))$

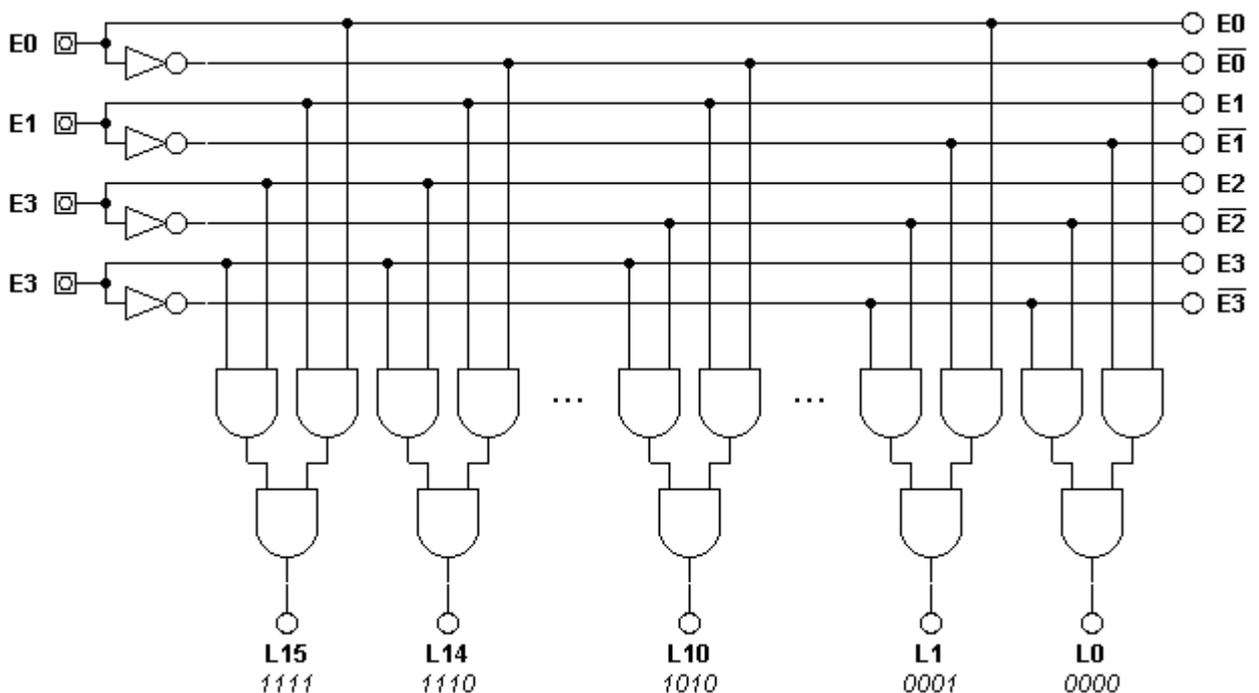


fig. 5.19 Décodeur d'adresse à 4 bits et 16 lignes (adresses)

Autrement dit, on peut combiner les bits E_3 et E_2 d'un côté; les bits E_1 et E_0 de l'autre puis combiner ces deux résultats dans une troisième porte AND. C'est exactement ce qui est fait dans la fig. 5.19.

En poursuivant le raisonnement, la phrase logique décrivant une adresse "8 bits" est décomposée en deux phrases "4 bits" reliées par une opération AND.

Il est donc très facile de combiner deux décodeurs d'adresse à n bits d'adresse pour en faire un décodeur d'adresse $2n$ bits : il suffit de les croiser et d'ajouter 2^{2n} portes AND ! Et voilà comment les constructeurs sont passés des ordinateurs "8 bits" aux ordinateurs "32 bits".

Mais le problème est là : avec l'exposant $2n$, le nombre de portes AND croît à une vitesse phénoménale !

Bits d'adresse	Nombre de Lignes	Portes AND par cascade	Portes AND
4	16	3	48
8	256	7	1 792
16	65 536	15	983 040
32	4 294 967 296	31	$1.33 \cdot 10^{11}$
64	$1.84 \cdot 10^{19}$	63	$1.16 \cdot 10^{21}$

Chaque ligne a sa cascade de portes AND. Supposons que l'on mette un million de ces cascades par millimètre carré de silicium. Un décodeur d'adresse à 32 bits demanderait alors $4\,294\,967\,296\text{ mm}^2$. Il tiendrait dans un carré de 6.5 cm de côté, ce qui est parfaitement gérable. Le même raisonnement appliqué à un décodeur d'adresse à 64 bits fournit une surface de 18.4 km^2 ! Il faudra certainement trouver une astuce...

5.13. Le démultiplexeur

Les flux d'information qui circulent dans un ordinateur sont le plus souvent des couples formés d'une adresse et d'une donnée (*ang.: address-data pair*).

Le démultiplexeur (*ang.: demultiplexor*) est la combinaison d'un décodeur d'adresse et d'une série de portes tri-state qui servent à orienter les données vers tel ou tel circuit ⁽¹⁾.

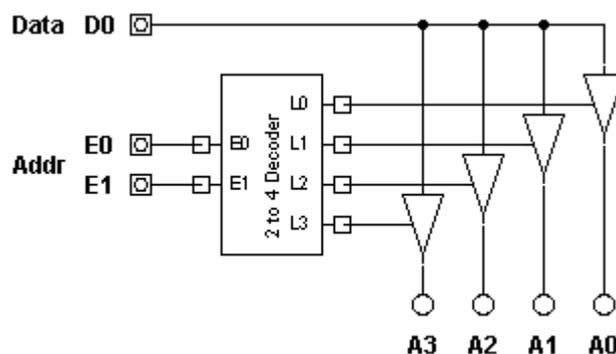


fig. 5.20 Démultiplexeur à 2 bits d'adresse et 1 bit de données

1 On pourrait utiliser des portes AND au lieu de portes tri-state mais l'avantage de ces dernières est de déconnecter physiquement les lignes non utilisées.

Il possède (fig. 5.20) :

- une entrée pour l'adresse [E0, E1],
- une entrée pour les données [D0],
- plusieurs sorties de données dont une seule est active [A0, ...A3].

Pour reprendre l'exemple de la gare de triage, le multiplexeur est un peu le système d'aiguillage **et** le système de signaux qui oblige le train à aller sur une voie donnée.

Dans un ordinateur, on retrouve un multiplexeur quand il faut :

- envoyer une donnée vers une mémoire dont on connaît l'adresse,
- pour orienter les bytes vers tel ou tel module arithmétique en fonction du code d'opération,
- etc.

En général, les données ne circulent pas bit par bit mais bien par byte. Tous les bits du byte doivent circuler en parallèle. La figure (fig. 5.21) illustre ce principe.

En attribuant un code, c'est-à-dire une adresse, à chaque type d'opération, cet arrangement permet d'envoyer un byte vers l'additionneur, vers l'incrémenteur, vers le gestionnaire de mémoire, etc.

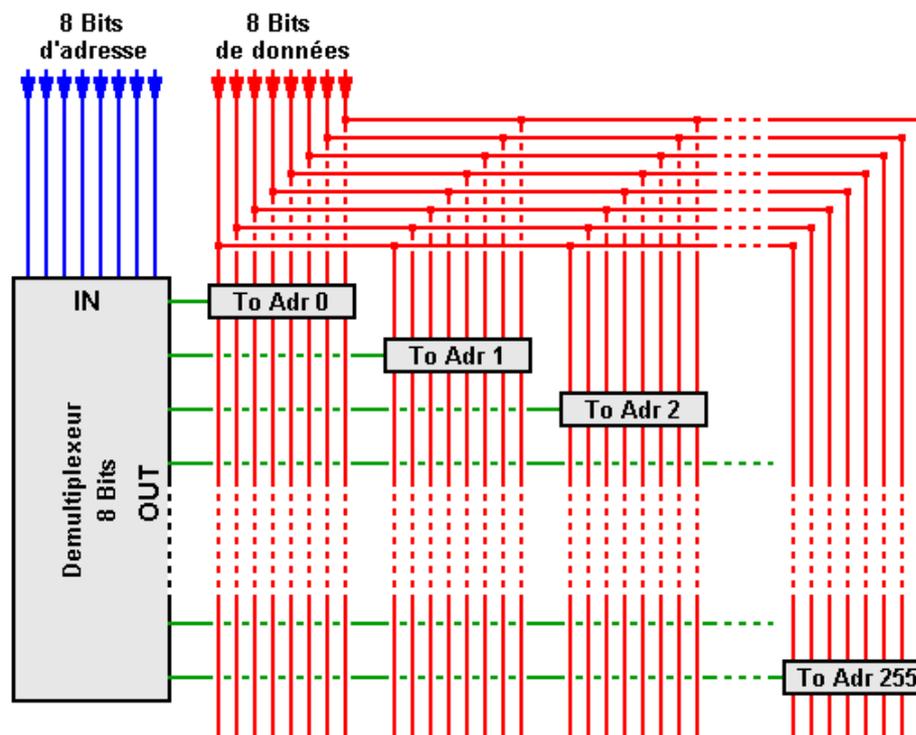


fig. 5.21 Démultiplexeur à 8 bits d'adresse et 8 bits de données

Les petits rectangles, qui étaient de simples portes tri-state dans le schéma fig. 5.20 deviennent maintenant des tableaux de portes tri-state.

Ce tableau de portes tri-state fait office d'interrupteur (fig. 5.22). En effet, si le bit qui entre par le côté [L_{in}] est nul, toutes les valeurs à la sortie sont déconnectées. Par contre, si ce bit vaut 1, les valeurs à la sortie sont identiques à celles présentées à l'entrée.

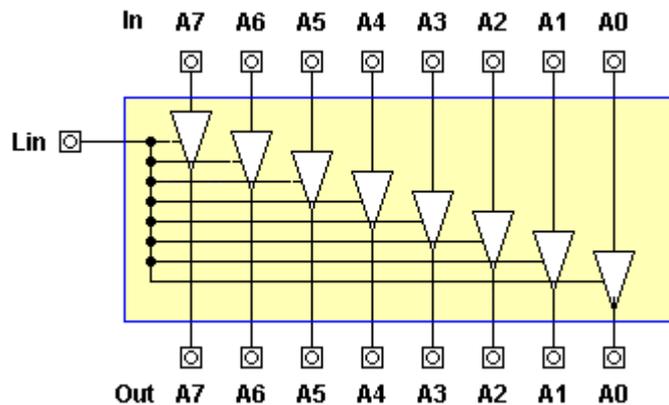


fig. 5.22 Tableau de portes tri-state faisant office d'interrupteur

5.14. Le multiplexeur

Le multiplexeur (*ang.: multiplexor*) ou collecteur est la contrepartie du démultiplexeur. On peut le comparer aux aiguillages placés à la sortie d'une gare de triage. Son rôle consiste à sélectionner le train issu d'une des voies de la gare de manière à l'envoyer sur la voie principale. Les feux de signalisation routière peuvent aussi être considérés comme des multiplexeurs. De même le téléphone et la télévision utilisent des multiplexeurs quand ils doivent envoyer des conversations ou des émissions vers un satellite.

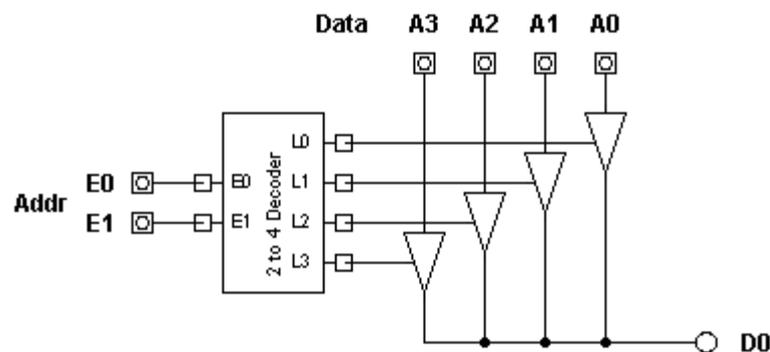


fig. 5.23 Multiplexeur à 2 bits d'adresse et 1 bit de données

Le multiplexeur ressemble fort au démultiplexeur (fig. 5.23). Il est la combinaison d'un décodeur d'adresse, d'une série de portes TriState dont une seule est active à la fois. Elles servent à sélectionner les données venant de tel ou tel circuit [A0...A3] pour les envoyer sur la ligne [D0]

Il possède :

- une entrée pour l'adresse,
- plusieurs entrées pour les données,
- une seule sortie de données.

5.15. Combinaison Multiplexeur-Démultiplexeur

On utilise une combinaison Multiplexeur-Demultiplexeur chaque fois qu'une seule structure de transport – unique, mais de grande capacité - doit transmettre des informations provenant de plusieurs sources.

C'est le cas des câbles téléphoniques, des faisceaux hertziens des satellites, des relais de GSM, des connexions ADSL, etc.

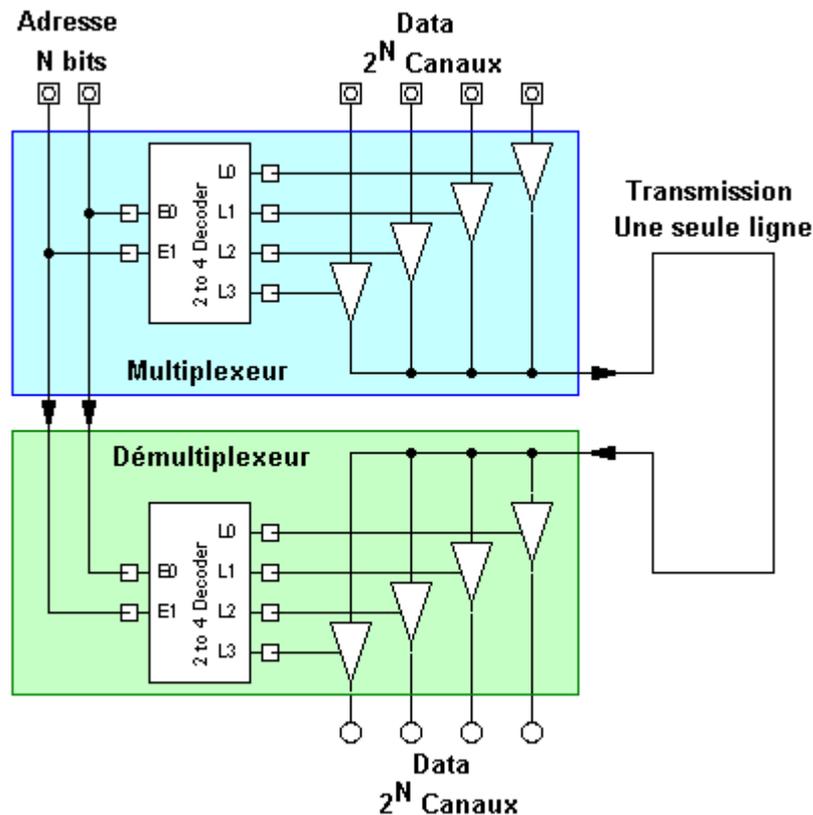


fig. 5.24 Combinaison Multiplexeur-Démultiplexeur

L'adresse est fournie au multiplexeur qui sélectionne l'une des sources (fig. 5.24). Un paquet d'informations provenant de cette source est envoyé sur le canal de transport vers le démultiplexeur.

L'adresse est envoyée de manière synchrone au démultiplexeur sur un autre canal. Ce transfert est beaucoup moins exigeant en terme de capacité du canal.

Au moment où le démultiplexeur reçoit le paquet d'informations, il reçoit aussi l'adresse. Il la décode et redirige le paquet d'informations vers le canal désiré.

Le plus souvent, on installe une deuxième combinaison similaire de manière à permettre un transfert en sens inverse. On parle alors de transmission **full duplex**.

Dans un ordinateur, ces éléments deviendront les **contrôleurs** ou **ponts**, le bus d'adresse et le bus de données. Le processeur pourra ainsi échanger des données avec ses différents périphériques.

On pressent que le point délicat ne sera pas la vitesse du processeur mais bien la capacité de transport du bus, c'est-à-dire la bande passante du canal installé entre le multiplexeur et le démultiplexeur.

5.16. Le contrôleur de parité

Le contrôle de parité est une technique simple qui permet de vérifier l'absence d'erreurs lors du stockage ou de la transmission de l'information.

Le principe consiste à compter le nombre de bits qui ont la valeur 1 dans la suite de bytes transmis. La parité se décline en deux règles :

- Parité paire : le bit de contrôle vaut 0 si le nombre de bits valant 1 dans la série transmise est pair, 0 sinon;
- Parité impaire : le bit de contrôle vaut 0 si le nombre de bits valant 1 dans la série transmise est impair, 0 sinon .

Appliquons ce principe sur un exemple. Nous devons transmettre le byte

$$A = 1011\ 0011$$

Dans ce byte, 5 bits sont mis à 1. 5 étant un nombre impair, le bit de contrôle sera mis à 1 si on travaille en parité paire de manière à ce que l'ensemble des bits mis à 1 dans le message (y compris le bit de parité) soit pair.

Par contre, il sera mis à 0 si on travaille en parité impaire de manière à ce que l'ensemble des bits mis à 1 dans le message (y compris le bit de parité) soit impair.

Où se trouve ce bit de contrôle ? Ici encore, de nombreuses solutions existent. Les plus courantes sont :

- après avoir envoyé 8 bytes d'information, on envoie un byte de contrôle qui reprend tous les bits de parité;
- l'information est recodée sur 7 bits et le huitième sert de bit de parité. Cette solution est souvent adoptée pour les échanges via les ports série.

Pour concevoir un **contrôleur de parité paire** qui travaille sur les huit bits d'un byte, raisonnons sur les deux premiers bits

A1	A0	Parité P
0	0	0
0	1	1
1	0	1
1	1	0

On voit que la table de vérité correspond à celle de la fonction XOR. On peut appliquer le même raisonnement sur les paires de bits A2-A3, A4-A5, A6-A7. Notre circuit commence donc par quatre portes XOR en parallèle.

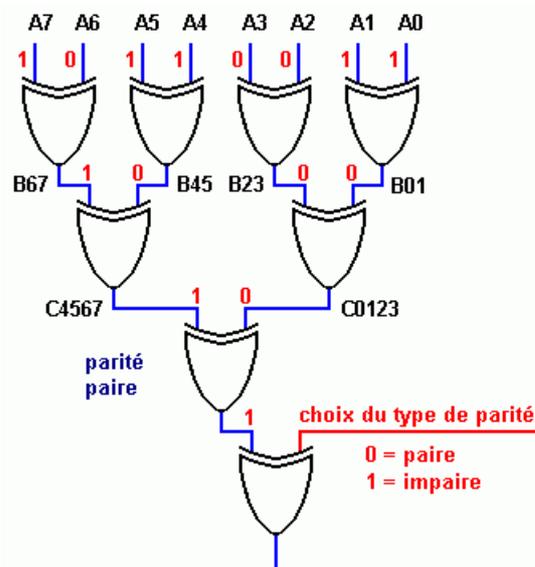


fig. 5.25 Calculateur de parité

A présent, combinons les bits de parité entre eux. Si nous combinons deux groupes qui avaient chacun un nombre impair de bits mis à 1, alors l'ensemble aura un nombre pair de bits mis à 1. En d'autres mots :

- Un nombre pair + un nombre pair = un nombre pair
- Un nombre pair + un nombre impair = un nombre impair
- Un nombre impair + un nombre pair = un nombre impair
- Un nombre impair + un nombre impair = un nombre pair

Transformons ces règles dans une table de vérité. Rappelons que nous travaillons en parité impaire :

B 23	B 01	C 0123
0	0	0
0	1	1
1	0	1
1	1	0

il s'agit encore d'un XOR. Nous pouvons donc utiliser deux portes XOR pour combiner les bits de parité venant de l'étage précédent. Le même raisonnement s'applique lorsque l'on combine les bits de parité des groupes de 4 bits (fig. 5.25).

Le contrôleur de parité paire est donc formé d'une pyramide de portes XOR. Il suffirait d'ajouter une porte NOT à la sortie pour le transformer en contrôleur de parité impaire.

Mais nous pouvons faire mieux : nous avons vu au point 5.8 qu'une porte XOR se comportait soit comme une transmission directe, soit comme une porte NOT selon que le bit de commande vaut 0 ou 1. Nous ajoutons une ultime porte XOR dont une entrée sert de bit de commande. Si ce bit vaut 0, le circuit se comporte comme un contrôleur de parité paire; si ce bit vaut 1, le circuit est un contrôleur de parité impaire.

5.17. L'afficheur digital

Un afficheur digital est un circuit qui comporte 7 segments à cristaux liquides disposés selon la forme d'un 8 (fig. 5.26). Chaque segment est relié à une borne numérotée de [A] à [G] qui permet de l'activer.

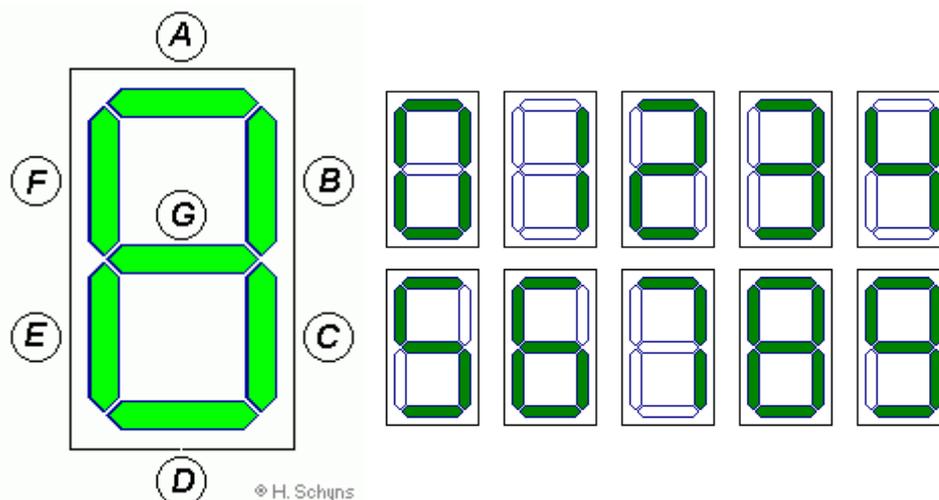


fig. 5.26 Afficheur digital à 7 segments et représentation des chiffres de 0 à 9

Un segment peut être activé seul ou en conjonction avec d'autres segments de manière à former des chiffres ou des lettres. Un afficheur à 7 segments permet de composer tous les chiffres mais seulement un nombre restreint de lettres (fig. 5.26).

Comment transformer les 10 chiffres en une combinaison de segments ?

La première étape consiste à dresser un tableau qui reprend, pour chaque chiffre, la référence des segments qui sont activés :

Segment Actif	Chiffre à afficher									
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	1	0	1	1	1	1	1
B	1	1	1	1	1	0	0	1	1	1
C	1	1	0	1	1	1	1	1	1	1
D	1	0	1	1	0	1	1	0	1	1
E	1	0	1	0	0	0	1	0	1	0
F	1	0	0	0	1	1	1	0	1	1
G	0	0	1	1	1	1	1	0	1	1

Ce tableau est rempli colonne par colonne. Nous voyons ainsi que, pour afficher un 0, nous devons activer tous les segments sauf le segment G. Par contre, dans le cas de l'affichage du 1, seuls les segments B et C sont activés.

Nous relisons ensuite le tableau ligne par ligne en écrivant les équations logiques correspondantes :

```

A:  "0" OR "2" OR "3" OR "5" OR "6" OR "7" OR "8" OR "9"
    ou bien
    NOT ( "1" OR "4" )
    
```

On voit que, selon les segments, il est plus tantôt plus facile de considérer les conditions d'allumage, tantôt les conditions d'extinction. Il est toujours possible de trouver une solution avec quatre conditions au maximum. Notons que la colonne "9" n'est jamais utilisée.

Si nous croisons les 10 entrées correspondant aux 10 chiffres avec les 7 sorties correspondant aux 7 segments, notre circuit de sélection - qui n'est pas sans rappeler notre clavier de la fig. 5.1 - ressemble à ceci :

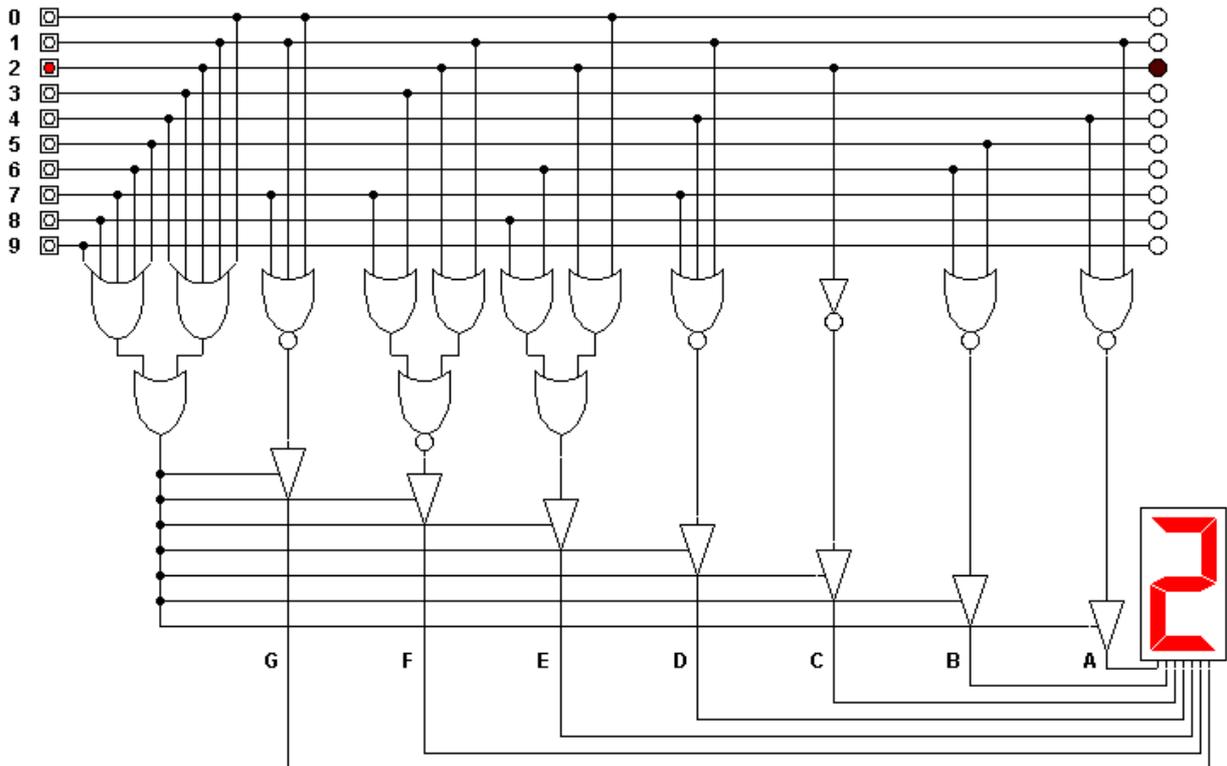


fig. 5.27 Conditions d'activation des 7 segments

Le motif binaire du chiffre le plus grand, 9, est 1001. En amont de ce circuit, nous pouvons donc utiliser un décodeur d'adresse "4 bits". Ce décodeur activera l'une

des dix lignes en fonction du motif binaire du chiffre à afficher. Les lignes 10 à 15 seront déconnectées ou, mieux, elles provoqueront l'affichage d'un motif qui exprime le dépassement de capacité de l'affichage ⁽¹⁾.

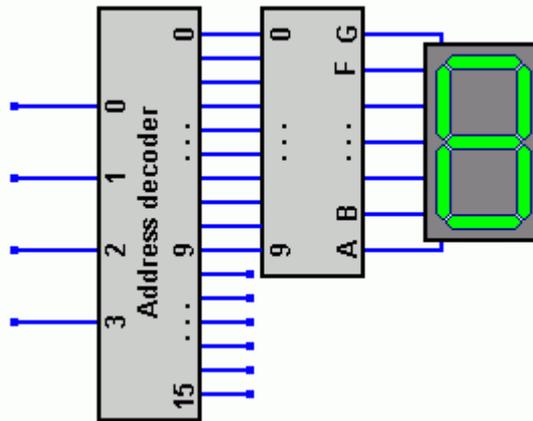


fig. 5.28 Schéma de l'afficheur

5.18. Le convertisseur Binaire-BCD

Un nombre codé sur un byte peut prendre n'importe quelle valeur comprise entre 0 et 255. Dès lors, pour pouvoir afficher tous les nombres possibles, nous avons besoin d'un affichage digital à trois modules. Mais comment convertir un nombre de huit digits binaires en un nombre de trois digits décimaux ?

Si chacun des digits décimaux pouvait être converti en binaire, le problème serait résolu : nous pourrions utiliser trois décodeurs décrits au paragraphe précédent. Ainsi, le nombre décimal 234_d est formé de trois digits décimaux 2-3-4 qui, codés indépendamment en binaire donnent 0010_b - 0011_b - 0100_b . Une telle représentation est appelée BCD (*ang.*: **B**inary **C**oded **D**ecimal). Malheureusement, la "vraie" représentation binaire de 234_d est 11101010_b .

Comment réaliser une conversion Binaire-BCD pour passer de 11101010_b à 0010_b - 0011_b - 0100_b ? Le problème a été résolu de manière fort élégante par un algorithme nommé "Shift and Add 3" (*fr.*: *décaler et ajouter 3*). Il est illustré ci-dessous dans le cas du nombre 234_d (tableau 5.1).

On commence par remplir avec des zéros les cellules (4 bits) qui traduiront les digits décimaux. Dans la colonne suivante, on écrit la représentation binaire du nombre à traduire.

Il y a autant d'étapes que de bits à convertir, soit 8 dans le cas d'un nombre codé sur un byte.

Chaque étape commence en faisant glisser (*ang.*: *to shift*) d'un rang vers la gauche (\leftarrow) tous les bits de toutes les cellules. Le cas échéant, les bits passent d'une cellule à la suivante. Ainsi, dans la première étape; le bit de poids fort du nombre à convertir pénètre dans la cellule des unités.

Ensuite, on vérifie (*ang.*: *to check*) les nombres binaires qui apparaissent dans les cellules cent-dix-unités. Si un nombre est égal ou supérieur à 5, on lui ajoute 3 (*ang.*: *add 3*), sinon, on le laisse inchangé ⁽²⁾. Attention : il n'y a pas de phase de vérification pour la dernière étape !

1 Souvent, le dépassement de capacité est représenté par un signe "+" qui a perdu une de ses branches. (segments B-C-G ou E-F-G. Parfois on utilise un petit "o" C-D-E-G.

2 Si la cellule contient 5 (0101), alors le prochain shift provoquera une multiplication par 2 ce qui donnera 10.

Opération	cent	dix	unités	Entrée	Etape
start	0000	0000	0000	11101010	
shift	0000	0000	0001	1101010	1
check	0000	0000	0001	1101010	
shift	0000	0000	0011	101010	2
check	0000	0000	0011	101010	
shift	0000	0000	0111	01010	3
check	0000	0000	1010	01010	
shift	0000	0001	0100	1010	4
check	0000	0001	0100	1010	
shift	0000	0010	1001	010	5
check	0000	0010	1100	010	
shift	0000	0101	1000	10	6
check	0000	1000	1011	10	
shift	0001	0001	0111	0	7
check	0001	0001	1010	0	
shift	0010	0011	0100		8
no check !	0010	0011	0100		
Sortie	2	3	4		

tableau 5.1 Shift and Add 3

Pour la concevoir le circuit de conversion binaire-BCD, remarquons d'abord que toutes les cellules effectuent le même travail. Nous pouvons également remplacer les huit étapes qui se succèdent dans le temps par huit modules qui s'enchaînent dans l'espace.

Au lieu de nous lancer dans l'examen d'une table de vérité complexe, nous allons aussi utiliser ce que nous avons déjà développé dans ce chapitre.

Pour réaliser un module élémentaire, nous avons besoin :

- d'un circuit comparateur pour savoir si un nombre est plus grand ou égal à 5;
- d'un circuit capable d'ajouter 3 à un nombre;

5.18.1. Le comparateur GE5

Un nombre entier est plus grand ou égal à 5 s'il est strictement plus grand que 4. Ce qui signifie que (fig. 5.29) :

- le bit valant 4 est actif ainsi que l'un des bits valant 2 ou 1
- ou
- le bit valant 8 est actif

$$T0 = (A2 \text{ AND } (A1 \text{ OR } A0)) \text{ OR } A3$$

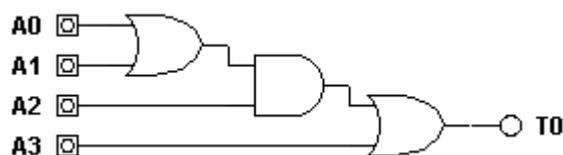


fig. 5.29 Le comparateur "GE 5"

5.18.2. Le circuit "Add 3 if GE5"

Si le nombre est supérieur ou égal à 5, il faut lui ajouter 3. Or, pour ajouter 3 à un nombre, nous pouvons utiliser un additionneur sur 4 bits.

Il est clair que les données qui entrent dans la cellule alimentent les bits $[A_0...A_3]$ de l'additionneur (fig. 5.30). Ils alimentent aussi les entrées $[A_0...A_3]$ du comparateur dont la sortie $[T_0]$ est redirigée vers les bits $[B_0B_1]$ de l'additionneur. Les bits $[B_3B_4]$ eux, seront mis à la masse (0).

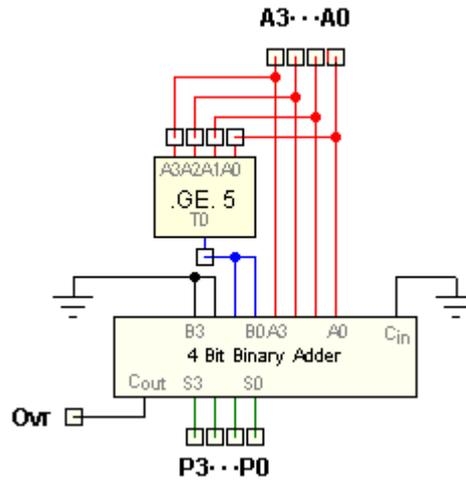


fig. 5.30 Un étage du décodeur BCD

Si le nombre entré est inférieur ou égal à 4, alors la sortie du comparateur vaut 0 ; le motif binaire entré en $[B]$ vaut 0000 et l'additionneur n'additionne rien. Si le nombre est supérieur ou égal à 5, alors la sortie du comparateur vaut 1 ; le motif binaire entré en $[B]$ vaut 0011 et l'additionneur ajoute 3 au nombre fourni en $[A]$.

5.18.3. L'assemblage final

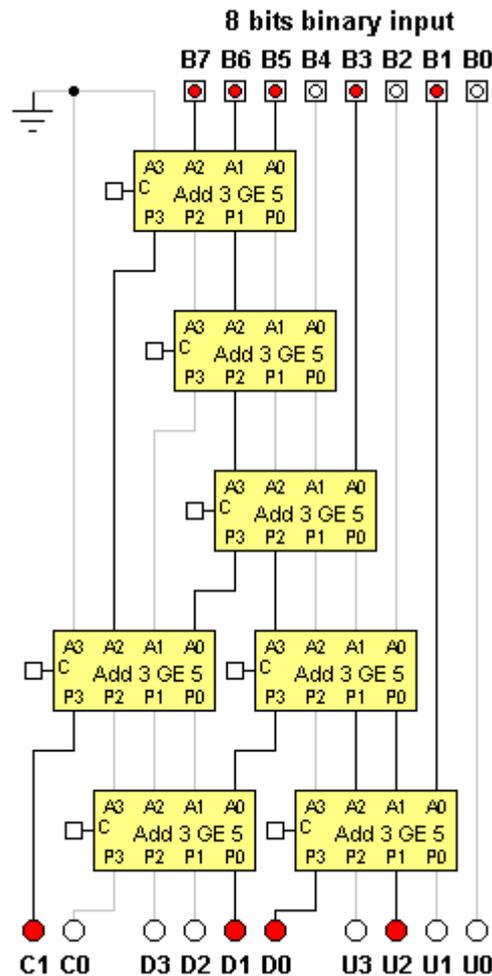


fig. 5.31 Structure complète du décodeur BCD

Il nous reste à assembler les étages pour réaliser l'algorithme. On note que rien de spécial ne se passe avant que les trois premiers bits ($B_7...B_5$) aient pénétré dans une nouvelle colonne de la matrice (fig. 5.31).

5.19. Conclusion

Ce chapitre nous a permis d'*explorer* le fonctionnement de tous les composants essentiels d'un ordinateur depuis la saisie au clavier jusqu'à l'affichage des résultats.

Nous avons vu comment un démultiplexeur permet de diriger un flux de données vers un module de calcul donné et comment un multiplexeur peut récupérer le résultat du module concerné. Nous avons vu comment une bascule permet de stocker un résultat intermédiaire ; nous avons découvert comment l'horloge permet de cadencer la progression de l'information.

Nous retrouverons ces circuits à différents endroits tant sur la carte mère, qu'au sein du processeur ainsi que dans les ponts et les cartes d'extension.

6. Exercices

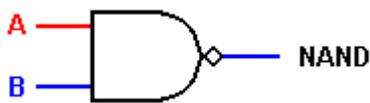
6.1. Exercice 1

Démontrer que :

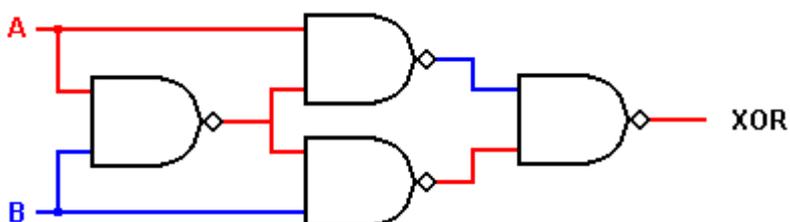
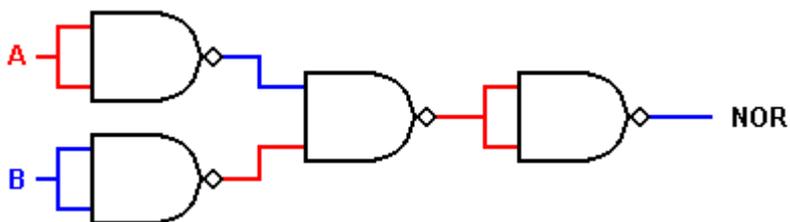
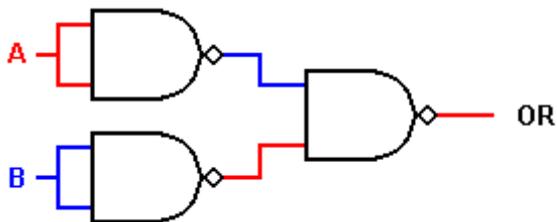
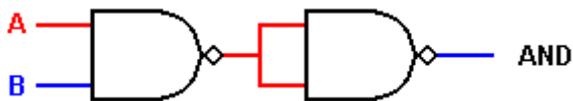
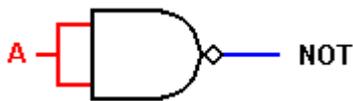
- $A \text{ XOR } A = 0$
- $(A \text{ XOR } B) \text{ XOR } A = B$
- $(A \text{ AND } B) \text{ OR } (A \text{ NOR } B) = \text{NOT } (A \text{ XOR } B)$

6.2. Exercice 2

La porte NAND est la plus simple à réaliser du point de vue technique et la plus économique.



Retrouver l'équation logique des circuits suivants construits avec des portes NAND et démontrer leur équivalence avec les fonctions indiquées



6.3. Exercice 3

Une technique de cryptage utilise une fonction XOR. Le byte crypté est la réalisation bit à bit d'une opération XOR entre le byte "source" et un byte "clé".

Source :	1011 0101
Clé :	0101 0110
Résultat :	1110 0011

Le décryptage se fait en passant le byte crypté dans la même logique, avec la même clé :

Résultat :	1110 0011
Clé :	0101 0110
Source :	1011 0101

Imaginez un circuit (similaire aux circuits d'addition, d'inversion et d'incrémentation) qui permette facilement de coder ou de décoder un byte.

Peut-on enchaîner plusieurs modules de ce type pour faire un cryptage sur 16 ou 32 bits ?

Modifiez ce circuit pour qu'il soit en mode "codage" si un bit de commande est mis à 1 et "transparent" si ce même bit de commande est mis à 0.

6.4. Exercice 4

Imaginez un circuit qui décale un byte d'un bit vers le LSB (least significant bit). Le bit le plus petit est perdu et le bit le plus grand est remplacé par 0.

6.5. Exercice 5

Modifiez le circuit de l'afficheur digital afin qu'il allume les segments B-C-G en cas de dépassement de capacité (de 10 à 15).

6.6. Exercice 6

Concevez un circuit d'affichage digital à deux digits qui puisse afficher les nombres de 00 à 99 sans passer par un convertisseur Binaire-BCD

Astuce : utiliser un décodeur d'adresse à 7 bits

6.7. Exercice 7

Concevez un circuit comparateur de deux nombres A et B qui affiche 1 si $A > B$ et 0 sinon.

Astuce pour la méthode 1 : commencer par comparer les bits de poids fort.

Astuce pour la méthode 2 : si $A > B$ alors $A - B > 0$.

7. Annexes

7.1. Circuits équivalents

7.1.1. Principe

Si un circuit présente N entrées indépendantes,
alors, la table de vérité correspondante doit prévoir $L=2^N$ lignes possibles en entrée,
et il y a $S = 2^L$ solutions possibles pour la sortie.

N entrées	L lignes 2^N	S solutions 2^L
1	2	4
2	4	16
3	8	256
4	16	65536

7.1.2. Cas d'une seule entrée

Selon le tableau, dans le cas d'une seule entrée X, celle-ci peut prendre deux états 0 ou 1 représentés par deux lignes du tableau de vérité. On peut avoir quatre solutions S possibles à la sortie.

- la sortie vaut 0 quelle que soit la valeur présentée à l'entrée,
- la sortie à toujours la même valeur que l'entrée,
- la sortie à toujours la valeur inverse de l'entrée,
- la sortie vaut 1 quelle que soit la valeur présentée à l'entrée.

Ces quatre solutions sont résumées dans le tableau suivant :

In	Out S			
X	0	1	2	3
0	0	1	0	1
1	0	0	1	1

En d'autres mots, le circuit correspondant est l'un des suivants :

Out S	Equation logique
0	$S = 0$
1	$S = \text{NOT } X$
2	$S = X$
3	$S = 1$

Par exemple, si on observe que la sortie vaut 1 quand l'entrée vaut 0, et qu'elle vaut 0 quand l'entrée vaut 1 (Solution 1) alors, le circuit correspondant est

$$S = \text{NOT } X$$

7.1.3. Cas de deux entrées

Si un circuit possède deux entrées, le tableau de vérité doit contenir quatre lignes et la configuration de sortie ne peut être que l'une des seize combinaisons suivantes :

In		Out S															
X	Y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Les équations logiques équivalentes sont :

Out S	Equation logique
0	$S = 0$
1	$S = X \text{ NOR } Y$
2	$S = (\text{NOT } X) \text{ AND } Y$
3	$S = \text{NOT } X$
4	$S = X \text{ AND } (\text{NOT } Y)$
5	$S = \text{NOT } Y$
6	$S = X \text{ XOR } Y$
7	$S = X \text{ NAND } Y$
8	$S = X \text{ AND } Y$
9	$S = \text{NOT } (X \text{ XOR } Y)$
A	$S = Y$
B	$S = (\text{NOT } X) \text{ OR } Y$
C	$S = X$
D	$S = X \text{ OR } (\text{NOT } Y)$
E	$S = X \text{ OR } Y$
F	$S = 1$

On notera avec intérêt que les solutions sont inversement symétriques : la solution F est le contraire de la solution 0; la solution E est le contraire de la solution 1 et ainsi de suite.

7.2. Cryptage XOR

Le cryptage XOR est une technique de cryptage simple et pourtant très efficace pour autant que la clé soit assez longue (voir Exercice 3). Le processus peut même être effectué manuellement à l'aide de la table ci-dessous.

- Choisissons une longueur de clé de cryptage :
Prenons une clé de 2 bytes, soit 16 bits ou 4 Hex
- Choisissons des valeurs Hex pour remplir la clé de cryptage :
Prenons 5 - 7 - A - D
- Considérons une chaîne de caractères ou suite de bytes ou de Hex à crypter :
soit 1 - 2 - 3 - 4 - 5 - 6

- Cryptons le premier Hex de la chaîne par le premier Hex de la clé; le deuxième avec le deuxième et ainsi de suite. Quand on a épuisé tous les caractères de la clé, on reprend le premier.

On sélectionne la ligne correspondant au Hex à crypter et la colonne correspondant au Hex de cryptage. Le résultat du cryptage se lit dans la cellule située à l'intersection de la ligne et de la colonne :

$$\begin{array}{lll}
 1 \text{ XOR } 5 = 4 & 2 \text{ XOR } 7 = 5 & 3 \text{ XOR } A = 9; \\
 4 \text{ XOR } D = 9 & 5 \text{ XOR } 5 = 0 & 6 \text{ XOR } 7 = 1
 \end{array}$$

La chaîne cryptée est 4 - 5 - 9 - 9 - 0 - 1

- Pour décrypter, on procède de la même manière, en partant de la chaîne cryptée et de la clé :

$$\begin{array}{lll}
 4 \text{ XOR } 5 = 1 & 5 \text{ XOR } 7 = 2 & 9 \text{ XOR } A = 3; \\
 9 \text{ XOR } D = 4 & 0 \text{ XOR } 5 = 5 & 1 \text{ XOR } 7 = 6
 \end{array}$$

- Il s'agit d'une technique très simple mais déjà très efficace à condition de prendre une clé d'au moins 16 Hex.

Table de cryptage XOR

XOR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	XOR
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0
1	1	0	3	2	5	4	7	6	9	8	B	A	D	C	F	E	1
2	2	3	0	1	6	7	4	5	A	B	8	9	E	F	C	D	2
3	3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C	3
4	4	5	6	7	0	1	2	3	C	D	E	F	8	9	A	B	4
5	5	4	7	6	1	0	3	2	D	C	F	E	9	8	B	A	5
6	6	7	4	5	2	3	0	1	E	F	C	D	A	B	8	9	6
7	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8	7
8	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8
9	9	8	B	A	D	C	F	E	1	0	3	2	5	4	7	6	9
A	A	B	8	9	E	F	C	D	2	3	0	1	6	7	4	5	A
B	B	A	9	8	F	E	D	C	3	2	1	0	7	6	5	4	B
C	C	D	E	F	8	9	A	B	4	5	6	7	0	1	2	3	C
D	D	C	F	E	9	8	B	A	5	4	7	6	1	0	3	2	D
E	E	F	C	D	A	B	8	9	6	7	4	5	2	3	0	1	E
F	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	F
XOR	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	XOR

8. Sources

- **Architecture et Technologie des Ordinateurs**
S. Tisserant
ESIL
<http://tisserant.developpez.com/cours/systeme/architecture/>
- **Algèbre de Boole et circuits logiques**
Architectures des ordinateurs Cours 3 - 2002
Eric Garcia
IUT GTR, Montbéliard
<http://www.scribd.com/doc/45877067/03-Circuits-logiques>
- **Notes de cours**
Pr. Richard E. Haskell
Oakland University
<http://www.cse.secs.oakland.edu/haskell>
- **Logic gates** (et articles connexes)
Ouvrage collectif
Wikipedia
http://en.wikipedia.org/wiki/Logic_gates
- **Resistor-Transistor Logic** (et articles connexes)
Ouvrage collectif
Wikipedia
http://en.wikipedia.org/wiki/Resistor-transistor_logic
- **Digital Works 3.04**
Un petit logiciel très intuitif et très bien conçu qui permet de dessiner et de tester des circuits logiques (tous les circuits présentés au chapitre 5 ont été réalisés avec DW 3.04).
D. J. Barker
University of Teesside.
Lien vers une version démo :
<http://www.electronics-lab.com/downloads/cnt/fclick.php?fid=52>