PIERRE_24

Zeste de savoir

De la logique aux processeurs

10 juillet 2020

Table des matières

Amis de Zeste de Savoir, bonjour !

Est-ce que vous vous êtes un jour demandé comment un ordinateur fonctionne, à son plus bas niveau ? Comment avec quelques composants très simples, on peut réaliser des choses aussi complexes que l'appareil sur lequel vous êtes en train de lire ce tutoriel ? Si oui, alors soyez les bienvenues.

Prérequis
Ce tutoriel se veut accessible au plus grand nombre. Normalement, pas besoin de maths poussées ou de connaissances complexes en électronique pour l'aborder
Objectifs
Appréhender la logique formelle et le binaire.
Expliquer comment on passe d'une expression mathématique à un circuit électrique qui fait la même chose.
Voir quelques circuits intéressants, qui entrent dans la composition d'un ordinateur.

1. La logique des propositions et des prédicats

Dans ce premier chapitre, nous allons découvrir les bases de la logique combinatoire. Même si ce mot peu faire peur, il désigne en fait quelque chose d'assez simple

1.1. C'est quoi, une proposition ?

Pour faire de la logique comme on en aura besoin par la suite, il s'agit d'abord de définir ce qu'est une **proposition** : c'est simplement un énoncé qui est soit vrai, soit faux, sans ambiguïté. Par exemple, les phrases suivantes sont des propositions :

— « Jules César est mort » (proposition vraie) ;
— « Mon chat est violet » (proposition à priori fausse, mais il suffit de le regarder pour en être sûr) ;
— « $2 + 2 = 5$ » (c'est une proposition, bien entendu fausse) .

Par contre, les phrases suivantes ne sont pas des propositions :

— « Assieds-toi » ;
— « T'as du feu ? » (ce n'est pas une proposition, c'est une question) ;
— « $x + 3 = z$ » (la valeur de vérité de cette phrase dépend de x et z, ce n'est pas une proposition).

La dernière phrase est en fait un **prédicat** : la valeur de vérité dépend des variables qu'elle contient (en l'occurrence x et z). Si cet exemple parait un peu trop mathématique, un autre exemple de prédicat serait la phrase « votre pays se situe en Europe », dont la vérité dépend de « votre pays ». Ainsi, pour les lecteurs européens, ce prédicat sera vrai, tandis qu'il sera faux pour les autres.

Évidement, dans un souci d'abstraction, plutôt que de travailler avec des phrases, on travaille avec des **variables propositionnelles**, dont la valeur ne peut être que *vrai* ou *faux*. On peut utiliser différent symboles pour exprimer la valeur de vérité ou de fausseté (vert/rouge, oui/non, V/F, allumé/éteint, et j'en passe), mais on choisira pour la suite de représenter vrai en notant 1 et faux en notant 0. C'est une notation commune, qui rappelle le binaire qu'on aura l'occasion de rencontrer par la suite.

Ainsi, si p représente la proposition « hier, à 15h, il pleuvait à Paris », alors $p = 1$ s'il pleuvait hier, à l'heure sus-mentionnée, là-bas , 0 sinon. Un prédicat tel que « le lecteur vit en Allemagne » pourrait alors s'écrire $q(x)$, où x est la nationalité du lecteur (oui, vous), et $q(x) = 1$ si vous vivez en Allemagne, 0 sinon.

Pour être plus formel, on admet en fait les 2 axiomes (règles) suivants, qu'on a en fait déjà énoncés.

- Le *principe du tiers-exclu* : une proposition est vraie ou alors sa négation est vraie, et inversement. Autrement dit, la négation de la négation d'une proposition est équivalente à la proposition initiale (autrement dit, $\neg(\neg p) \Leftrightarrow p$, je reviens sur les notations ci-dessous !).
- Le *principe de non-contradiction* : une proposition ne peut être vraie et fausse en même temps, autrement dit une variable propositionnelle ou un prédicat vaut soit « 0 », soit « 1 ». C'est pour ça que j'ai commencé par dire que la valeur d'une proposition devait être déterminée de manière non-ambiguë.

Il existe d'autres formes de logiques, qui n'admettent pas l'un ou l'autre de ces axiomes, par exemple la logique floue , qui définit qu'une variable propositionnelle peut valoir n'importe quel réel entre 0 et 1. Bien que ce soit très intéressant, ce n'est pas le sujet ici

1.2. Assembler des propositions avec les connecteurs

Évidement, dans une conversation, on débite un nombre impressionnant de propositions ou de prédicats, dont la valeur de vérité est d'ailleurs parfois laissée à l'appréciation des interlocuteurs. Or, nous ordonnons naturellement ces propositions en utilisant ce qu'on appelle des connecteurs.

Par exemple, dans la phrase « je n'aime ni courir, ni jouer au tennis », on peut constater l'usage de deux propositions « j'aime courir » (p, qui peut être soit vraie soit fausse) et « j'aime jouer au tennis » (q), toute deux niées (puisque je dis que je n'aime PAS ça), et assemblées par un connecteur « et » (puisque je dis que je n'aime pas courir ET que je n'aime pas jouer au tennis). On constate également que le langage humain n'exprime pas toujours clairement les choses.

Cet ensemble de propositions, rassemblées par des connecteurs logiques, s'appelle une **formule**.

1.2.1. Les connecteurs de base

Dans l'exemple précédent, on a déjà rencontré deux des trois connecteurs que j'appellerai *de base*. En effet, le premier des trois est **la négation** : soit une proposition p, sa négation peut s'écrire « $\neg p$ ».

Afin de décrire les différentes situations, on utilise ce qu'on appelle une **table de vérité** : pour chacune des variables propositionnelles, on écrit les différentes valeurs qu'elles peuvent prendre, puis on écrit le résultat de la formule. Ainsi, p peut prendre deux valeurs, qui sont 1 et 0. On a donc la table de vérité suivante :

p	$\neg p$
1	0
0	1

On voit que, de manière assez logique, si $p = 1$, $\neg p = 0$, et inversement.

On a ensuite le **connecteur ET** (qu'on appelle aussi connecteur de **conjonction**), qui relie deux variables propositionnelles. Ici, la table de vérité comporte 4 lignes, car il existe 4 possibilités : soit p et q sont toute deux vraies ou fausses, soit elles ont des valeurs opposées. Et on obtient la table suivante :

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

On constate que le ET peut s'écrire $\wedge$ et que la formule $p \wedge q = 1$ seulement si $p = q = 1$.

Finalement, le dernier connecteur de base qu'on a pas encore rencontré est le **connecteur OU**, encore appelé **connecteur de disjonction**. La table de vérité de ce connecteur est la suivante :

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

On constate que le OU peut s'écrire $\vee$ et que la formule $p \vee q = 0$ seulement si $p = q = 0$. Il s'agit en fait du *ou non-exclusif*, qu'on traduirait en français par « ou » de manière un peu ambiguë (on devrait écrire « et/ou », mais cela porterait à confusion), comme dans la phrase « le magasin est fermé les jours fériés ou durant nuit » (puisque s'il ne s'agissait pas d'un ou non-exclusif, le magasin devrait être ouvert la nuit des jours fériés).

i

Entre autres propriétés qu'on verra au chapitre prochain, ET et OU sont commutatifs ($p \wedge q = q \wedge p$, par exemple) et associatifs [c'est-à-dire que $p \wedge (q \wedge r) = (p \wedge q) \wedge r$, par exemple]. Les parenthèses gardent leur sens ici (on effectue d'abord l'opération entre parenthèse).

1.2.2. Vérité d'une formule

Pour déterminer la vérité d'une formule, on doit déterminer la valeur de vérité pour les différentes valeurs des variables propositionnelles qui la composent. Par exemple, soit la formule $f(a, b, c) = a \wedge (b \vee c)$. Il est cette fois nécessaire de construire une table de vérité à 8 lignes (de

manière générale, s'il y a n variables propositionnelles, il y a aura 2^n lignes dans la table de vérité) :

a	b	c	$b \vee c$	$f(a,b,c) = a \wedge (b \vee c)$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

On peut constater plusieurs choses dans cet exemple :

- On peut tout à fait réaliser la table de vérité pour des morceaux de la formule (ici $b \vee c$) avant de calculer la valeur de vérité de la formule elle même. Ici, il suffit ensuite de faire un ET entre la quatrième $(b \vee c)$ et la première colonne pour connaître la valeur de vérité de la formule complète.
- Remarquez la manière de remplir la table pour les variables a, b et c : quatre « 0 » suivit de 4 « 1 » pour la première colonne, une alternance de 2 « 0 » et 2 « 1 » pour la deuxième et une alternance de « 0 » et de « 1 » pour la troisième. En procédant comme cela, vous êtes sûr de couvrir les 8 cas de figure possible.
- On voit ici que $f(a,b,c)$ ne peut être vraie que si $a = 1$. Cela provient de ce qu'on a observé précédemment : $a \wedge b = 1$ si $a = b = 1$.

1.2.3. D'autres connecteurs

On retrouve ensuite une série de connecteurs définis à partir des trois connecteurs de base[1] qu'on a vu plus haut.

1.2.3.1. L'implication

On utilise souvent, sous différentes formes, la phrase « p **implique** q », même si c'est plus souvent sous la forme « si ... alors ... », par exemple « s'il pleut, alors je reste à la maison » ou encore « $n = 0 \Rightarrow n + 1 > 0$ ». La table de vérité de cet opérateur est la suivante :

p	q	$p \Rightarrow q$	$\neg p \vee q$
0	0	1	1

1. On peut en fait très bien définir « ET » à partir de « OU » et de la négation, ou « OU » à partir de « ET » et de la négation. Mais vous verrez par la suite pourquoi j'insiste sur ces 3 connecteurs.

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

On peut constater que :

- L'opérateur d'implication peut s'écrire $\Rightarrow$;
- $p \Rightarrow q$ est équivalent à $\neg p \vee q$ (dernière colonne) ;
- La formule est fausse lorsque $p = 1$ et $q = 0$, ce qui permet de lire $p \Rightarrow q$ comme « p seulement si q ». Autrement dit, « du vrai n'implique jamais du faux » et « du faux implique n'importe quoi ».
- On peut encore dire « p suffit à q » ou encore « q est nécessaire pour p ». Si je reprends l'exemple « $n = 0 \Rightarrow n + 1 > 0$ », alors on obtient « que $n = 0$ suffit pour que $n + 1 > 0$ », et « $n + 1 > 0$ est nécessaire pour que $n = 0$ » ;
- Dans $p \Rightarrow q$, p est aussi nommé l'antécédent (ou l'hypothèse) et q est nommé le conséquent (ou la thèse).

À noter que si on a $p \Rightarrow q$, on peut retrouver

- Sa réciproque, qui s'écrit $q \Rightarrow p$;
- Sa contraposée, qui s'écrit $\neg q \Rightarrow \neg p$;
- Son inverse, qui s'écrit $\neg p \Rightarrow \neg q$.

Néanmoins, ce n'est pas parce qu'une implication est vraie que sa réciproque ou son inverse l'est également ! Ainsi si je reprends l'implication $n = 0 \Rightarrow n + 1 > 0$ est vraie, mais sa récriproque, $n + 1 > 0 \Rightarrow n = 0$ n'est pas vraie (du vrai n'implique jamais du faux) ! De même, son inverse, $n \neq 0 \Rightarrow n+1 \leq 0$ n'est pas vraie non plus (pour la même raison). Par contre, et on le verra plus loin, la contraposée d'une implication est toujours de même valeur de vérité que l'implication elle-même, et dans notre cas l'implication $n + 1 \leq 0 \Rightarrow n \neq 0$ est bel et bien vraie aussi.

Attention à la négation des propositions mathématiques. Si p est la proposition $n = 0$ et q est la proposition $n < 0$, $\neg p$ équivaut à $n \neq 0$, mais $\neg q$ correspond à $n \geq 0$ (si ce n'est pas strictement plus petit que 0, c'est soit égal à 0, soit supérieur) !

1.2.3.2. L'équivalence

Second connecteur relativement important, l'équivalence. Sa table de vérité est la suivante :

p	q	$p \Leftrightarrow q$	$(p \Rightarrow q) \wedge (q \Rightarrow p)$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	1	1

On constate, assez logiquement, que :

— $p \Leftrightarrow q$ est vrai si $p = q$;
— $p \Leftrightarrow q$ équivaut à ce que $p \Rightarrow q$ et sa réciproque soient toutes deux vraies (ce n'est pas forcément le cas, je le répète une fois encore). Autrement dit, « p implique q et q implique p », ou encore « p est une condition nécessaire et suffisante à q » ou encore « p si et seulement si q » ;
— On retrouve parfois ce connecteur écrit $p \equiv q$ (littéralement, $p = q$).

Ce connecteur est très employé en mathématiques, dans les démonstrations.

1.2.3.3. ... Et d'autres

On retrouve finalement trois connecteurs logiques ayant leur utilité en électronique (voir plus loin) ou en informatique.

— NAND (« NON ET ») et NOR (« NON OU »), respectivement $p \uparrow q$ et $p \downarrow q$;
— XOR (le *ou exclusif* ou *disjonction exclusive*), qu'on écrit souvent $p \oplus q$ (ou encore $p \neq q$).

Les tables de vérités de ces connecteurs sont :

p	q	$p \uparrow q$	$p \downarrow q$	$p \oplus q$
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

On peut par ailleurs en déduire que

— $p \uparrow q \Leftrightarrow \neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$;
— $p \downarrow q \Leftrightarrow \neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$;
— $p \oplus q \Leftrightarrow \neg(p \Leftrightarrow q)$;

1.2.4. Tautologies

Une tautologie est une formule qui est toujours vraie quelle que soit la valeur des variables qui la compose. Ainsi, les trois dernières définitions du paragraphe précédent sont des tautologies. On peut le vérifier par tables de vérité.

Pour vous entraîner, vous pouvez vérifier que $\vDash \neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$ (le symbole $\vDash$ permet d'indiquer qu'il s'agit d'une tautologie).

Contenu masqué n°1

Vous pouvez également vérifier que $\vDash p \Rightarrow q \Leftrightarrow \neg q \Rightarrow \neg p$:

Contenu masqué n°2

1.3. C'est toujours utile : les prédicats et les quantificateurs

Il est tout à fait possible de faire des tables de vérités avec des prédicats, comme on l'a fait pour les propositions : ce n'est pas parce que la valeur de $p(x)$ change avec x que $p(x)$ ne peut pas être vrai ou faux.

Par contre, si on rajoute un **quantificateur** devant un prédicat qui caractérise la (ou les) variable(s) de ce prédicat, alors on obtient une proposition. Et on va distinguer 3 quantificateurs :

- Le **quantificateur universel**, représenté par $\forall$. Ainsi, $\forall x : p(x)$ est une **proposition** vraie si pour tout x, $p(x) = 1$. On le retrouve en général en mathématique, par exemple dans la proposition $\forall x \in \mathbb{N} : x \geq 0$ (*aka* « tout nombre appartenant à l'ensemble[2] des naturels est positif »).
- Le **quantificateur existentiel**, représenté par $\exists$. Ainsi la proposition $\exists x : p(x)$ est vraie s'il existe au moins un x tel que $p(x) = 1$. Une version *plus dure* de ce quantificateur est celui de **stricte existentialité**, $\exists !$, la proposition $\exists ! x : p(x)$, n'étant alors vraie que s'il existe un et un seul x tel que $p(x) = 1$.

À noter qu'un quantificateur a une priorité absolue, ce qui signifie qu'il s'applique sur tout ce qui suit :

$$\underbrace{\forall a, \forall b : \underbrace{p(a,b)}_{\text{portée de } \forall b}}_{\text{portée de } \forall a} .$$

Ainsi, l'ordre des quantificateurs a de l'importance. Dès lors, les deux propositions suivantes ne sont pas équivalentes :

1. $\forall x \in \mathbb{R}, \exists y \in \mathbb{R} : x^2 + y < 0$ (pour tout réel x, il existe un réel y tel que $x^2 + y < 0$) ;
2. $\exists y \in \mathbb{R}, \forall x \in \mathbb{R} : x^2 + y < 0$ (il existe un réel y tel que pour tout réel x, $x^2 + y < 0$).

La première proposition est vraie (il suffit de prendre $y = -x^2 - 1$) alors que la seconde est bien entendu fausse (il n'existe pas de réel y qui satisfera le prédicat pour **tout** les x possibles parmi les réels).

Dans ce premier chapitre, on a donc vu qu'il existait des propositions (et des prédicats) que l'on pouvait représenter en utilisant des variables propositionnelles, qu'on pouvait associer en formules grâce à des connecteurs logiques, dont les principaux sont le ET, le OU et la négation

2. Je ne vais pas m'étendre sur la notion d'ensemble, qui a déjà été abordée par c_page , mais je vous rappelle qu'il s'agit d'un « paquet » d'objets mathématiques quelconques (un ensemble de valeurs possibles, pour le dire autrement).

(même si on a pu ensuite en dériver d'autre). Je ne souhaitais, ceci dit, pas trop m'appesantir là-dessus, car le prochain chapitre va présenter les choses de manière un peu plus systématique.

À tout de suite

Contenu masqué

Contenu masqué n°1

p	q	$p \wedge q$	$\neg(p \wedge q)$	$\neg p \vee \neg q$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

On voit bien que les deux dernières colonnes sont équivalentes, donc la tautologie $\vDash \neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$ est respectée. Cette tautologie est d'ailleurs une des deux lois de De Morgan qu'on verra au prochain chapitre. Retourner au texte.

Contenu masqué n°2

p	q	$p \Rightarrow q$	$\neg q$	$\neg p$	$\neg q \Rightarrow \neg p$
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	0	0
1	1	1	0	0	1

On voit que la troisième et la dernière colonne sont équivalentes, il s'agit donc bien d'une tautologie, une implication et sa contraposée sont bien équivalentes. Retourner au texte.

2. L'algèbre de Boole

Vous allez voir qu'on va ré-exprimer ce qu'on a vu au chapitre précédent sous une forme légèrement différente, mais qui va nous permettre de travailler de manière plus efficace avec ces variables propositionnelles. Vous allez voir que c'est moins compliqué qu'il n'y parait.

L'algèbre de Boole, telle qu'utilisée ici, n'est en fait qu'une façon légèrement différente de formaliser les choses : on se base en fait sur la théorie des ensembles (en travaillant avec un ensemble à deux éléments, $\mathbb{B} = \{0,1\}$), et on rajoute deux opérations (ou plutôt deux lois), qui sont le $+$ et le $\times$. L'objet qu'on crée alors s'appelle une « algèbre de Boole à deux éléments » et possède différentes propriétés intéressantes qu'on va exploiter ici.

2.1. Propriétés de l'algèbre de Boole

Dans l'introduction, on a parlé d'ensemble et de lois. Ça, c'est pour que les matheux soient contents ... Mais de quoi on parle ?

Déjà, on parle d'ensemble, ce qui signifie que toute opération sur un élément de cet ensemble **doit** être un élément de cet ensemble (pas forcément le même). Ici, on travaille avec un ensemble composé de deux éléments, donc le choix est assez restreint.

Mais quelles sont ces opérations ?

— La **disjonction**, $a+b$. Cette opération est équivalente au connecteur OU, donc $a+b=0$ si $a=b=0$.
— La **conjonction**, $a \times b$, qu'on peut également écrire $a \cdot b$ ou encore plus simplement ab. C'est l'équivalent du connecteur ET, donc $a \cdot b = 1$ si $a=b=1$.

Et mine de rien, le fait qu'on utilise ces deux opérations et la théorie des ensembles veut dire qu'on a déjà un certain nombre d'axiomes (de règles) associés :

Nom	Axiome pour $\times$	Axiome pour $+$
Neutre	$a \cdot 1 = a$	$a + 0 = a$
Inverse	$a \cdot \bar{a} = 0$	$a + \bar{a} = 1$
Commutativité	$ab = ba$	$a + b = b + a$
Distributivité	$a\,(b+c) = ab + ac$	$a + bc = (a+b)\,(a+c)$

Ici, a, b et c sont des variables propositionnelles telles qu'on les a rencontrées au chapitre précédent, on parle également **d'atome**[3] dans le cadre de l'algèbre de Boole.

3. Pour rappel, je suis chimiste de formation, donc ça me fait sourire

Qu'est-ce qu'on a là ?

— L'axiome du neutre est assez logique au vu de ce que vous avez appris en math : il existe, pour chacune des deux opérations, une valeur qui ne change pas le résultat. Ces valeurs sont d'ailleurs les mêmes que d'habitude, puisque n'importe quoi multiplié par 1 ou additionné de 0 reste le même.

— L'axiome suivant (aussi appelé axiome de complémentation) signale la présence d'un inverse. Ça a l'air très compliqué, mais ça nous permet en fait de retrouver quelque chose qu'on a déjà vu : pour tout atome, a, il existe sa négation, qu'on note ici $\bar{a}$ (certains notent aussi a'). Comme on l'a vu au chapitre précédent, $\bar{1} = 0$ et $\bar{0} = 1$. Là où c'est moins intuitif, c'est que dans les mathématiques classiques, la multiplication par l'inverse (a^{-1} pour la multiplication) donne 1, et l'addition de l'inverse ($-a$ pour l'addition) donne 0, alors que c'est l'inverse ici.

— Les troisième et quatrième axiomes, la commutativité et la distributivité, sont assez classiques pour la multiplication, moins pour l'addition (l'addition n'est pas distributive en algèbre classique). Mais ce n'est pas l'addition classique, donc ça tombe bien 🍊

Bref, on a des règles légèrement différentes de ce qu'on a l'habitude de voir, mais qui sont néanmoins logiques au vu de ce qu'on a vu au chapitre précédent. Par ailleurs, ces règles sont définies d'une manière particulière, au vu d'une propriété intéressante de l'algèbre de Boole, la **dualité** : si on définit une règle pour un des deux opérateurs, on peut obtenir la règle pour l'autre opérateur en interchangeant $1 \leftrightarrow 0$ et $+ \leftrightarrow \times$. Ainsi, si on reprend le premier axiome pour $\times$, $a \cdot \bar{a} = 0$, on voit qu'on obtient bien l'axiome correspondant pour $+$ (son **dual**) en remplaçant $\cdot$ par $+$ et 0 par 1.[4]

À noter qu'on nomme **littéral** un atome ou son opposé. Ainsi, $\bar{a} + b$ est une disjonction de deux littéraux, $\bar{a}$ et b.

De là, on tire un certain nombre de propriétés[5] :

Nom	Propriété de $\times$	Propriété de $+$
Associativité	$a\,(b \cdot c) = (a \cdot b)\,c$	$a + (b + c) = (a + b) + c$
Involution	$\bar{\bar{a}} = a$	*de même*
Idempotence	$a \cdot a = a$	$a + a = a$
Élément absorbant	$a \cdot 0 = 0$	$a + 1 = 1$
Loi de De Morgan	$\overline{ab} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \cdot \bar{b}$
Théorème d'absorption	$a\,(a + b) = a$	$a + ab = a$
Théorème d'allègement	$a\,(\bar{a} + b) = a \cdot b$	$a + \bar{a} \cdot b = a + b$

4. Ça va même plus loin que ça, puisque si on réussit à prouver une propriété pour un opérateur, son dual est également prouvé.

5. Les preuves des différentes propriétés sont disponibles à différents endroits sur internet, par exemple ici (en) . Vous pouvez par ailleurs vous amuser à les vérifier en écrivant des tables de vérité, même si certaines sont triviales.

Théorème du consensus	$ab + \bar{a}c + bc = ab + \bar{a}c$	$(a+b)(\bar{a}+c)(b+c) = (a+b)(\bar{a}+c)$

Ces propriétés vont se révéler utiles lorsqu'on va travailler, juste en dessous, sur des fonctions booléennes. Par ailleurs, vous pouvez facilement vérifier que ces propriétés respectent la dualité exprimée ci-dessus.

2.2. Fonctions booléennes

Pour la faire courte, une fonction, c'est une application qui fait correspondre un ensemble à un autre, c'est-à-dire qui, à un objet mathématique dans un ensemble de départ, en fait correspondre un autre dans l'ensemble d'arrivée :

$$f : \mathcal{E}^m \to \mathcal{G}^n : e_1, \ldots, e_m \mapsto g_1, \ldots, g_n$$

qui signifie que la fonction f est une application qui fait correspondre à un ensemble de m éléments (un m-uplet) de l'ensemble $\mathcal{E}$ à un ensemble de n éléments de l'ensemble $\mathcal{G}$. Dans le cas qui nous occupe, on va travailler avec des fonctions de $\mathbb{B}^n \to \mathbb{B}$, donc des fonctions de n variables en entrées et qui donne un résultat unique (0 ou 1).

Micmaths a déjà consacré un excellent cours aux fonctions , que je vous conseille de relire pour vous rafraîchir la mémoire, le cas échéant

Encore une fois, un mot très compliqué pour désigner ce qu'on a déjà rencontré dans le chapitre précédent sous le terme de *formule*. Une fonction booléenne, c'est donc une série d'atomes liés entre eux par des opérations vues plus haut. Par exemple, la formule $a \wedge (b \vee c)$ est « traduite », dans l'algèbre de Boole, par $f(a,b,c) = a\,(b+c)$. De par les axiomes qu'on a vu plus haut, on sait déjà que c'est équivalent à $f(a,b,c) = a \cdot b + a \cdot c$.

Mais on peut complexifier les choses et travailler sur des fonctions booléennes beaucoup plus complexes (la complexité n'ayant pour limite que votre imagination), telles que

$$f(a,b,c) = a \cdot \overline{(b \cdot c)} + \overline{(\bar{a} + b)}.$$

Une telle fonction, bien que tout à fait correcte, peut être réécrite sous deux formes :

- Forme normale conjonctive (FNC) : la fonction est exprimée sous forme d'une conjonction de formes disjonctives : $f = m_1 \cdot m_2 \ldots m_n$, avec m_i la clause disjonctive, c'est-à-dire une disjonction de littéraux $(q_1 + q_2 + \ldots + q_n)$. Si tous les atomes sont présents (sous forme de littéraux) dans chacune des formes disjonctives, on parle de forme **normale** (ou canonique).
- Forme normale disjonctive (FND) : la fonction est exprimée sous la forme d'une disjonction de formes conjonctives (donc $f = m_1 + m_2 \ldots + m_n$, $m_i = q_1 \cdot q_2 \ldots q_n$).

Pour obtenir de telles formes, il est nécessaire de faire porter la complémentation sur les variables (en utilisant la loi de De Morgan si nécessaire), puis d'utiliser la distributivité de $\times$ sur $+$ (pour une forme disjonctive) où de $+$ sur $\times$ (pour une forme conjonctive).

Dans le cas de la fonction donnée plus haut, on aurait :

$$\begin{aligned} f(a,b,c) &= a \cdot \overline{(b \cdot c)} + \overline{(\bar{a} + b)} \\ &= a\,(\bar{b} + \bar{c}) + \bar{\bar{a}} \cdot \bar{b} && \text{(De Morgan)} \\ &= a\,(\bar{b} + \bar{c}) + a \cdot \bar{b} && \text{(involution)} \\ &= a \cdot \bar{b} + a \cdot \bar{c} + a \cdot \bar{b} && \text{(distributivité)} \\ &= a \cdot \bar{b} + a \cdot \bar{c} && \text{(idempotence)} \end{aligned}$$

où la dernière ligne est bien une forme disjonctive, mais pas *canonique* (on obtient en fait la fonction simplifiée, comme on le verra plus loin). On voit qu'il est normalement nécessaire de se servir des différentes propriétés pour arriver à une expression qui est relativement plus simple à traiter.

2.2.1. Obtention des formes normales

Il existe cependant un moyen d'obtenir les FND ou FNC facilement en travaillant sur la table de vérité. On s'intéresse aux différents cas pour a, b et c et on regarde ce que cette fonction retourne :

a	b	c	$f(a,b,c)$
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	0
0	0	0	0

2.2.1.1. Obtenir la FND

Pour la FND, on s'intéresse aux cas pour lesquels la fonction vaut 1 et pour lesquels on écrit la conjonction des littéraux (qu'on appelle des *minterms*) correspondante :

1. $a = b = 1$ et $c = 0$. Ce premier cas donne la conjonction $a \cdot b \cdot \bar{c}$, et la table de vérité de la fonction nous dis que celle-ci donne 1 pour résultat lorsque $a = 1$, $b = 1$ et $c = 0$, or $1 \cdot 1 \cdot \bar{0} = 1$.

2. $a = 1$ et $b = 0$, soit $c = 1$ et on a la conjonction $a \cdot \bar{b} \cdot c$, soit $c = 0$ et on a la conjoinction $a \cdot \bar{b} \cdot \bar{c}$.

La forme normale disjonctive (FND) est obtenue en prenant la disjonction de ces différentes conjonctions, donc $f(a,b,c) = a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c}$. On aurait ceci dit pu l'obtenir comme suis :

$$\begin{aligned} f(a,b,c) &= a \cdot \bar{b} + a \cdot \bar{c} \\ &= a \cdot \bar{b} \cdot 1 + a \cdot \bar{c} \cdot 1 && \text{(neutre)} \\ &= a \cdot \bar{b}\,(c + \bar{c}) + a \cdot \bar{c}\,(b + \bar{b}) && \text{(complémentation)} \\ &= a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} && \text{(distributivité)} \\ &= a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c} && \text{(idempotence)} \end{aligned}$$

(mais c'est quand même plus facile avec la table de vérité)

2.2.1.2. Obtenir la FNC

On s'intéresse aux cas où la fonction vaut 0 (qu'on appelle des *maxterms*) , et de la même manière, on note les différentes conjonctions correspondantes. On obtient alors

$$\overline{f(a,b,c)} = a \cdot b \cdot c + \bar{a} \cdot b \cdot c + \bar{a} \cdot b \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot \bar{b} \cdot \bar{c},$$

qui correspond à l'inverse de la fonction $f(a,b,c)$. Si on en prend la conjonction, la propriété d'involution assure que $\overline{\overline{f(a,b,c)}} = f(a,b,c)$, mais la loi de De Morgan nous permet alors d'obtenir :

$$\begin{aligned} \overline{\overline{f(a,b,c)}} &= \overline{a \cdot b \cdot c + \bar{a} \cdot b \cdot c + \bar{a} \cdot b \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot \bar{b} \cdot \bar{c}} \\ &= (\bar{a} + \bar{b} + \bar{c})\,(a + \bar{b} + \bar{c})\,(a + \bar{b} + c)\,(a + b + \bar{c})\,(a + b + c). \end{aligned}$$

... Qui est bien la forme normale conjonctive (FNC).

2.3. Simplifier une forme normale : la méthode de Karnaugh

Obtenir la forme normale est certes pratique, mais le but final est en fait de simplifier cette forme, afin d'avoir le moins d'opérations possibles à faire. On a vu plus haut qu'il était en fait possible, à l'aide des différentes propriétés, de simplifier une fonction booléenne, néanmoins, il existe une manière de faire plus simple, basée sur les diagrammes de Karnaugh (ou tables de Karnaugh), qui a le bon goût d'être une méthode graphique.

2.3.1. La méthode

Tout d'abord, il faut savoir comment construire ces fameux diagrammes de Karnaugh : il s'agit d'une autre manière de représenter la table de vérité ci-dessus.

1. Pour trois variables, un diagramme contient 8 cellules (16 cellules pour 4 variables).
2. Chacune de ces cases correspond à une série de valeur pour les différentes variables : ainsi, pour 3 variables, a, b et c, on aurait par exemple, le triplet 001 (donc $a = 0, b = 0$ et $c = 1$).
3. La règle est que quand on passe d'une case de ce diagramme à une autre, seule une variable peut changer de valeur (la numération suit un code de Gray). Ainsi, on peut passer de 001 à 011 mais pas à 010. Dès lors, on ordonne le diagramme de telle manière à ce que les deux premières colonnes aient une valeur de $a = 1$, tandis que ce sont les colonnes du milieu qui auront une valeur pour b égale à 1. Comme ça, on a la série $10, 11, 01, 00$ pour a et b. On fait de même pour les lignes (c'est tout à fait arbitraire).
4. De manière évidente, un triplet ou quadruplet donné ne peut se retrouver que dans une seule cellule.

On obtient par exemple les diagrammes suivants :

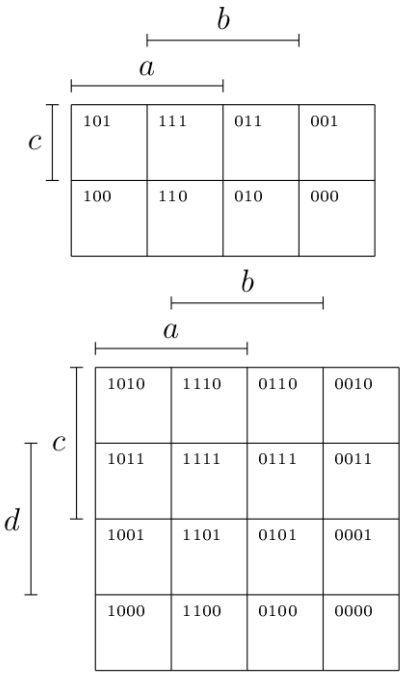


FIGURE 2.1. – Diagrammes de Karnaugh pour 3 variables (dessus) et 4 variables (dessous). Les lignes et colonnes marquées par une variable et une ligne désigne les zones où ces variables valent 1. **Le choix est tout à fait arbitraire**, on pourrait très bien décider de mettre c et d sur les lignes à la place de a et b, voir de les inverser. L'important c'est de bien respecter la règle qui dit que le changement d'une cellule à une autre (vers le haut, le bas, la gauche ou la droite) doit s'accompagner d'un et d'un seul changement de valeur.

Il suffit alors de remplir ce diagramme en plaçant 1 aux endroits où la fonction vaut 1, 0 ailleurs. Il suffit de reprendre la table de vérité et de mettre ce qu'il faut au bon endroit. Dans le cas de notre fonction, on obtient le diagramme suivant.

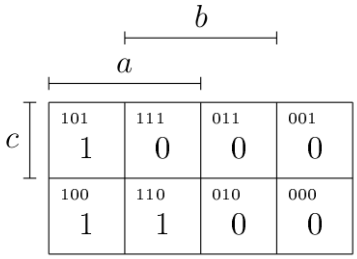


FIGURE 2.2. – Diagramme de Karnaugh pour la fonction $f(a, b, c)$, en accord avec la table de vérité écrite plus haut.

La règle de simplification est assez simple : si deux cellules contenant la même valeur sont contiguës, c'est qu'elles peuvent être simplifiées. En effet, ces deux cellules contiennent forcément une même série de littéraux, f, multipliée par un atome, p, nié dans une cellule et non-nié dans l'autre. Dès lors,

$$f \cdot p + f \cdot \bar{p} = f\,(p + \bar{p}) = f.$$

Ce qui signifie qu'on élimine les atomes qui varient d'une cellule à l'autre, ou encore **qu'on ne conserve que les littéraux qui ne changent pas**. De la même manière, on peut simplifier 4 cellules contiguës (en ligne, en colonne ou en « carré «) d'un coup, voir également 8 cellules contiguës, si elles sont présentes (2 lignes ou 2 colonnes dans un diagramme 4x4, du coup). On peut considérer plusieurs fois une même cellule, puisque $f = f + f$ par idempotence.

2.3.1.1. Simplifier la FND

Pour obtenir une fonction disjonctive simplifiée, on travaille sur les endroits du diagramme où la fonction vaut 1.

Dès lors, dans le cas de $f(a, b, c)$, on constate que les deux cellules de la première colonne valent 1 (entourées en vert), ainsi que les deux premières cellules de la deuxième ligne (entourées en bleu).

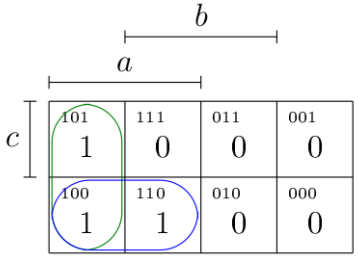


FIGURE 2.3. – Simplification de la FND : 3 cellules sont contiguës. On constate que la cellule correspondante au triplet 100 est considérée deux fois.

1. Pour la simplification des cellules entourées *en vert* : il s'agit de la formule $a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c}$, où on voit que c'est c qui est l'atome nié et non nié. On obtient donc la conjonction $a \cdot \bar{b}$
2. Pour la simplification *en bleu* : $a \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot \bar{c} = a \cdot \bar{c}$, puisque b est l'atome nié et non-nié dans ce cas.

On obtient dès lors la forme disjonctive simplifiée suivante : $f(a, b, c) = a \cdot \bar{b} + a \cdot \bar{c}$, comme prévu.

2.3.1.2. Simplifier la FNC

On travaille sur la partie du diagramme où la fonction vaut 0, ce qui permet en fait d'obtenir une forme simplifiée pour $\overline{f(a, b, c)}$ sous forme disjonctive, puis forme conjonctive simplifiée par la loi de De Morgan, ainsi qu'expliqué plus haut.

Au niveau de notre exemple, on constate qu'on a une série de 4 cellules contiguës (en rose), et deux cellules contiguës à la première ligne, en deuxième et troisième colonne (en orange).

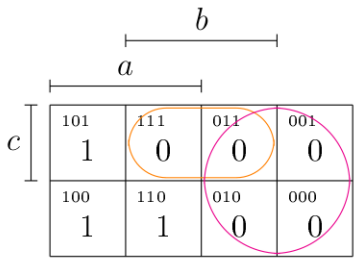


FIGURE 2.4. – Simplification de la FNC : on constate que 4 cellules sont contiguës.

1. Pour la simplification *en rose* : $\bar{a} \cdot b \cdot c + \bar{a} \cdot b \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot \bar{b} \cdot \bar{c} = \bar{a}$, car on voit que sont niés et non-niés les atomes b et c, ce qui permet de tous les simplifier.
2. Pour la simplification *en orange* : $a \cdot b \cdot c + \bar{a} \cdot b \cdot c = b \cdot c$, puisqu'ici, c'est l'atome a qui est nié et non-nié.

On obtient alors $\overline{f(a, b, c)} = \bar{a} + b \cdot c$, ce qui permet d'obtenir une forme conjonctive simplifiée comme suit :

$$\overline{\overline{f(a, b, c)}} = \overline{\bar{a} + b \cdot c} = a\,(\bar{b} + \bar{c}).$$

À noter qu'on aurait pu l'obtenir à partir de la forme disjonctive simplifiée en mettant en évidence a.

2.3.2. Attention, le diagramme de Karnaugh est cyclique !

Et j'insiste :

Puisque l'ordre est arbitraire (tant qu'on respecte la règle du 1-seul-changement-de-valeur-d'une-cellule-à-une-autre), ce qui signifie qu'un bord du diagramme est en fait le début de l'autre bord, et que ce qu'on considère comme plat est en fait un tore.

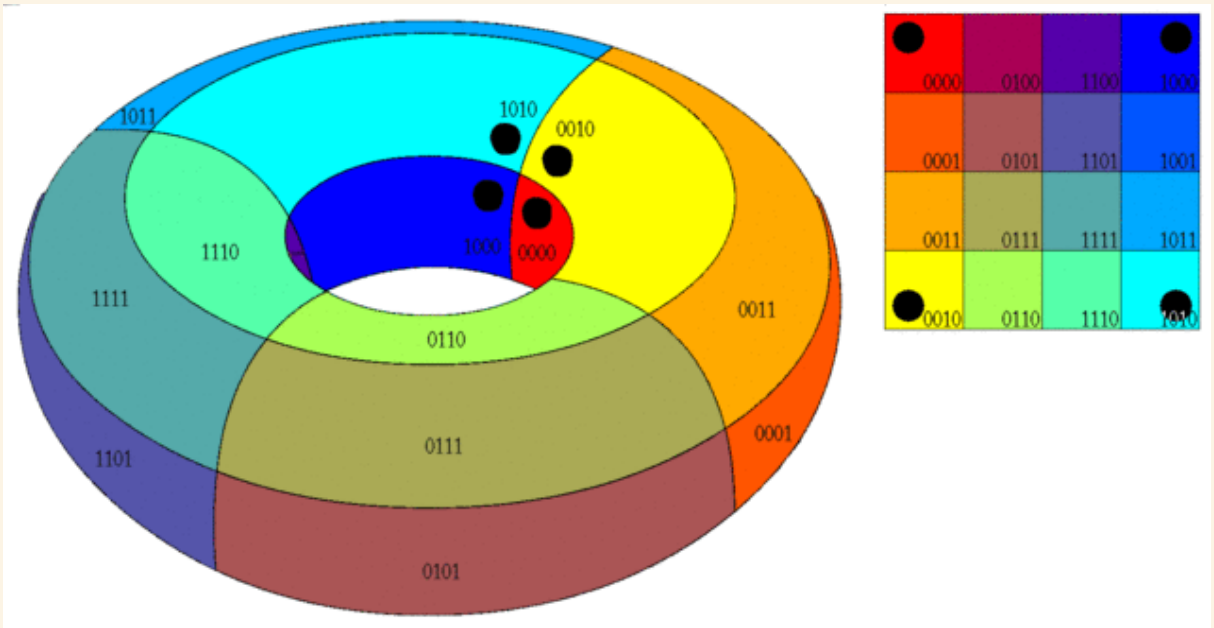


FIGURE 2.5. – Le diagramme de Karnaugh représente en fait un tore (source : Wikipédia).

Il faut donc bien regarder sur les côtés si un plus grand groupement n'est pas possible.

Voyons ça sur un second exemple : soit une fonction $g(a, b, c, d)$ qui donnerait le diagramme de Karnaugh suivant.

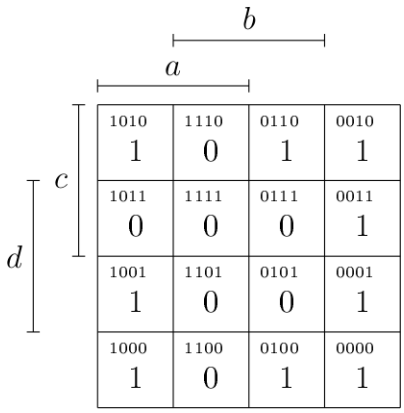


FIGURE 2.6. – Une autre fonction, cette fois à 4 variables, $g(a, b, c, d)$, pour varier les plaisirs.

Le nombre de possibilité pour former des groupes de 2 ou 4 cellules est considérablement accru à cause de ça. Par exemple, on peut très bien proposer les groupements suivants.

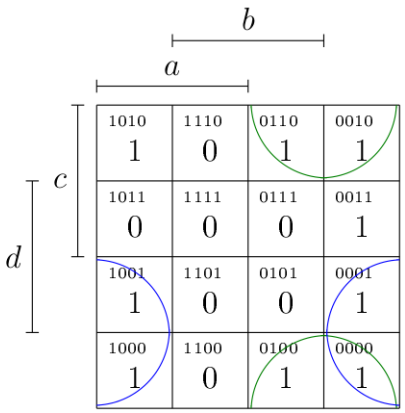


FIGURE 2.7. – Groupement *possibles* de variables (évidement, il en manque). Les deux demi-cercles verts (et bleus) sont connectés, et se simplifient ensemble.

Du coup, la stratégie va être :

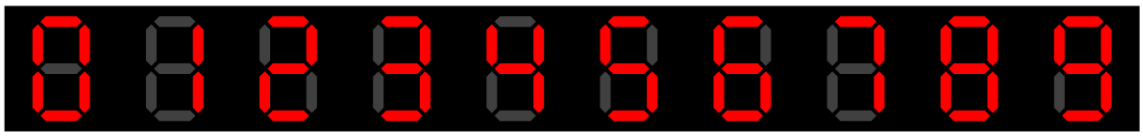
1. D'abord tenter de faire des groupes de 8, puis de 4 et puis de 2 cellules (et attention à ce qui se passe sur les bords, donc).
2. Pas besoin de former un groupe si toutes les cases en questions appartiennent déjà à d'autres groupes.

Sachant cela, quelle serait la forme disjonctive et conjonctive réduite correspondante au diagramme donné plus haut ?

Contenu masqué n°3

2.3.3. « *Don't cares* » ?

Un autre cas intéressant à traiter, c'est le cas où on ne s'intéresse pas au résultat pour certaines valeurs. L'exemple classique, c'est l'afficheur 7 segments , qui permet d'afficher les nombres sur une calculette ou un réveil digital (ou un écran du même type).



0	1	2	3	4	5	6	7	8	9
(0_2)	(1_2)	(10_2)	(11_2)	(100_2)	(101_2)	(110_2)	(111_2)	(1000_2)	(1001_2)

FIGURE 2.8. – Afficheur 7 segments et représentation des 10 chiffres (la conversion en binaire de chacun des chiffres est donnée entre parenthèses).

Chaque « segment » est une LED qu'on allume ou éteint en fonction du chiffre qu'on souhaite afficher. Ces segments sont désignés par des lettres, par facilité.

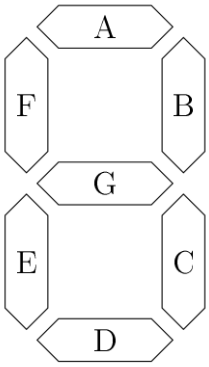


FIGURE 2.9. – Lettres utilisées pour désigner les segments dans un afficheur 7 segments (dixit Wikipédia).

Il s'agit d'un bon exemple, puisqu'on pourrait par exemple se demander, en fonction du nombre à afficher, s'il est nécessaire d'allumer, disons, le segment G. On voit dans l'image ci-dessus que pour représenter les nombres de 0 à 9, 4 *bits* sont nécessaires, appelons-les a_1, a_2, a_3 et a_4. Par exemple, pour représenter 5, on aura $a_1 = 0, a_2 = 1, a_3 = 0$ et $a_4 = 1$. Et la suite, c'est évidement une petite table de vérité, où on va chercher la fonction $G(a_1, a_2, a_3, a_4)$, qui vaudra 1 quand le segment G sera allumé, 0 sinon.

Nombre	a_1	a_2	a_3	a_4	$G(a_1, a_2, a_3, a_4)$
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1

Oui, mais quand on va réaliser notre table de Karnaugh, quelle valeur mettre pour, par exemple, 1011 (11 en décimal) ? L'afficheur ne peut pas afficher de nombre plus grand que 9, évidement[6] ... *Eh* bien on s'en fout (c'est un comportement indéterminé, qui ne devrait pas arriver), on met un *don't care*, qu'on va symboliser par un « X ».

Notez qu'on écrit également cette fonction

$$G(a_1, a_2, a_3, a_4) = \sum m(2,3,4,5,6,8,9) + \sum d(10,11,12,13,14,15)$$
$$= \sum m(0010, 0011, 0100, 0101, 0110, 1000, 1001) + \sum d(1010, 1011, 1100, 1101, 1110, 1111$$

où $m()$ (pour *minterms*) sont les valeurs pour lesquelles la fonction vaudra 1 et $d()$ (pour *don't care*) sont les valeurs qui ne nous intéressent pas. L'écrire en binaire (dessous) ou en décimal (dessus) revient évidement au même. Les termes qui ne sont pas cités (0, 1 et 7) sont ceux pour lesquels la fonction vaut 0.

On a donc le diagramme suivant.

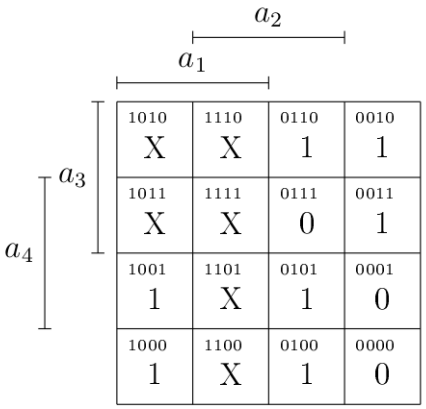


FIGURE 2.10. – Il y a bien 7 « X » en tout. Attention, pour remplir le diagramme, ce n'est pas dans l'ordre numérique, il faut bien placer les valeurs correspondantes aux bons quadruplets a_1, a_2, a_3, a_4.

L'avantage des « X », c'est qu'ils peuvent valoir 0 ou 1, au choix. Voilà qui ouvre évidement plus de possibilités de simplifications ! Je vous laisse réfléchir au résultat.

Contenu masqué n°4

2.3.4. Conclusion

La méthode de Karnaugh est relativement simple à mettre en place et fonctionne bien ... Pour peu que le nombre de variables soit faible (puisqu'il y a 2^n possibilités pour n variables). On va voir une seconde méthode légèrement plus simple à expliquer à un ordinateur pour qu'il fasse le taf'.

6. En fait, certains exploitent les nombres suivant pour afficher les 7 premières lettres de l'alphabet (a, b ... f) afin de faire de l'hexadécimal. Mais pour l'exemple, on disait que.

Néanmoins, si vous voulez vérifier vos résultats, le site http://www.32x8.com/ le permet de manière simple et efficace, et ce jusqu'à 8 variables si vous êtes courageux (par contre il est en anglais).

2.4. Simplifier une forme normale : la méthode de Quine-Mc Cluskey

2.4.1. Présentation de l'algorithme

La méthode de Quine-Mc Cluskey se déroule en deux temps : la recherche de termes dit « générateurs » et l'élimination des termes redondants (suivi éventuellement de la méthode de Petrick pour choisir entre termes). C'est un algorithme taillé pour être traduit en code informatique, mais dont l'explication n'est pas très compliquée (son application est par contre un peu fastidieuse). Voyons ça sur un exemple.

Reprenons la fonction pour le segment G et voyons comment générer sa forme simplifiée. Pour rappel :

$$\begin{aligned} G(a_1, a_2, a_3, a_4) &= \sum m(2,3,4,5,6,8,9) + \sum d(10,11,12,13,14,15) \\ &= \sum m(0010, 0011, 0100, 0101, 0110, 1000, 1001) + \sum d(1010, 1011, 1100, 1101, 1110, 1111 \end{aligned}$$

2.4.1.1. Première étape

Nous allons commencer par regrouper les *minterms* (et les valeurs « X ») en fonction du nombre de 1 qu'ils contiennent. Par facilité, on va aussi les classer par ordre croissant à l'intérieur de chaque groupe :

$$
\begin{array}{ll}
G_1 & \left\{\begin{array}{ll} m(2) & 0010 \\ m(4) & 0100 \\ m(8) & 1000 \end{array}\right. \\
G_2 & \left\{\begin{array}{ll} m(3) & 0011 \\ m(5) & 0101 \\ m(6) & 0110 \\ m(9) & 1001 \\ d(10) & 1010 \\ d(12) & 1100 \end{array}\right. \\
G_3 & \left\{\begin{array}{ll} d(11) & 1011 \\ d(13) & 1101 \\ d(14) & 1110 \end{array}\right. \\
G_4 & \left\{\begin{array}{ll} m(15) & 1111 \end{array}\right.
\end{array}
$$

On va ensuite combiner les termes qui ne varient que d'un chiffre, pour générer l'ordre 1. Par exemple, $m(2)$ (notation pour *minterm* 2) peut être combiné avec $m(3)$, puisque ceux-ci ne varient que d'un chiffre. On écrira ce nouveau terme $m(2,3) = 001x$ (comme dans la méthode de Karnaugh, ces termes peuvent être combinés de telle manière que a_4 disparaisse, d'où le x[7]). À noter qu'une fois qu'un terme est utilisé, il est marqué comme tel (mais peut être réutilisé si nécessaire). L'avantage d'avoir trié en fonction du nombre de 1, c'est que les termes du groupe 1 ne peuvent être combinés qu'avec les termes du groupe 2, ceux du groupe 2 uniquement avec ceux du groupe 3, et ainsi de suite. En effet, on élimine les termes redondants, et $m(3,2) = 001x$ donnant la même chose que $m(2,3)$, il n'est pas nécessaire de le garder.

Dès lors, on obtient :

7. On peut également mettre un tiret à la place du x.

$$
\begin{array}{rl}
G_1 + G_2 \left\{ \begin{array}{ll}
m(2,3) & 001x \\
m(2,6) & 0x10 \\
m(2,10) & x010 \\
m(4,5) & 010x \\
m(4,6) & 01x0 \\
m(4,12) & x100 \\
m(8,9) & 100x \\
m(8,10) & 10x0 \\
m(8,12) & 1x00
\end{array} \right. \\
G_2 + G_3 \left\{ \begin{array}{ll}
m(3,11) & x011 \\
m(5,13) & x101 \\
m(6,14) & x110 \\
m(9,11) & 10x1 \\
m(9,13) & 1x01 \\
m(10,11) & 101x \\
m(10,14) & 1x10 \\
m(12,13) & 110x \\
m(12,14) & 11x0
\end{array} \right. \\
G_3 + G_4 \left\{ \begin{array}{ll}
m(11,15) & 1x11 \\
m(13,15) & 11x1 \\
m(14,15) & 111x
\end{array} \right.
\end{array}
$$

Pour générer l'ordre 2, on regroupe les termes (encore une fois, avec ceux du groupe suivant) de telle manière à ce que les x soient à la même position, et que, encore une fois, le reste ne diffère que d'un chiffre. Ainsi, $m(2,3) = 001x$ peut être combiné avec $m(10,11)101x$ puisque le x est en dernière position dans les deux cas et le chiffre qui varie est le premier. On aura donc $m(2,3,10,11) = x01x$. Pareil, notez qu'on obtient la même chose en combinant $m(2,10) = x010$ avec $m(3,11) = x011$, d'où $m(2,10,3,11) = x01x$. Une fois encore, comme ce terme est redondant, on l'élimine. Notez que des termes redondants contiennent à chaque fois les mêmes *minterms* de base (ici : 2, 3, 10 et 11), il suffit donc de garder la combinaison dans l'ordre croissant et d'éliminer toutes les autres. Les autres termes de l'ordre 2 ne peuvent pas être combinés.

$$
\begin{array}{rl}
G_1 + G_2 + G_3 \begin{cases} m(2,3,10,11) & x01x \\ m(2,6,10,14) & xx10 \\ m(4,5,12,13) & x10x \\ m(4,6,12,14) & x1x0 \\ m(8,9,10,11) & 10xx \\ m(8,9,12,13) & 1x0x \\ m(8,10,12,14) & 1xx0 \end{cases} \\
G_2 + G_3 + G_4 \begin{cases} m(9,11,13,15) & 1xx1 \\ m(10,11,14,15) & 1x1x \\ m(12,13,14,15) & 11xx \end{cases}
\end{array}
$$

Finalement, on refait l'étape une troisième fois (et dernière fois, comme il n'y a que 4 variables). Dans ce cas-ci, on peut grouper $m(8,9,10,11)$ avec $m(12,13,14,15)$ pour obtenir $m(8,9,10,11,12,13,14,15) = 1xxx$. On obtient également la même chose en combinant $m(8,10,12,14)$ avec $m(9,11,13,15)$, donc pas besoin de le retenir. Et finalement, on obtient encore une fois la même chose avec $m(8,9,12,13)$ et $m(10,11,14,15)$.

Conclusion de tout ça, on se retrouve avec les termes suivants : $m(2,3,10,11) = x01x$, $m(2,6,10,14) = xx10$, $m(4,5,12,13) = x10x$, $m(4,6,12,14) = x1x0$ (termes non utilisés de l'ordre 2) et $m(8,9,10,11,12,13,14,15) = 1xxx$. Il s'agit des **termes générateurs** (ou implicants premiers), la première étape de l'algorithme est complète. On peut constater que le terme $m(8,9,10,11,12,13,14,15)$ avait déjà été obtenu par la méthode de Karnaugh plus haut. En effet, et même si c'est pas forcément visible, la méthode employée **reste la même** (par contre, elle est plus facile à expliquer à un ordinateur) !

i

À noter qu'un implicant (en) , c'est une combinaison de *minterms* d'une fonction booléenne donnée. Par exemple, si on a une fonction $f(a,b,c) = a \cdot b + \bar{b} \cdot c + a \cdot c$, alors $a \cdot b$ ou $a \cdot b \cdot c$ sont des implicants de f. Un **implicant premier**, c'est un implicant qu'on ne peut plus simplifier par les lois de l'algèbre de Boole (qu'on ne peut pas exprimer avec moins de littéraux). Par exemple, $a \cdot b$ est un implicant premier de f, par contre $a \cdot b \cdot c$ ne l'est pas : on peut soit retirer le c pour obtenir l'implicant premier précédent, soit retirer le b pour obtenir le troisième des implicant premier de f, $a \cdot c$. Ce que l'étape précédente nous à permis, c'est donc d'obtenir ces fameux implicants premiers (puisqu'on ne peut plus les combiner pour les simplifier).

2.4.1.2. Deuxième étape

La deuxième étape d'élimination des termes redondants passe par la réalisation d'une table des implicants (ou monômes) premiers.

On réalise un tableau où les entrées des lignes sont composées des termes générateurs, et les entrées des colonnes sont composées des *minterms* (on élimine cette fois les membres de $d()$, puisqu'ils ne sont pas importants). On remplit alors le tableau en indiquant si le terme générateur

peut exprimer le *minterm*. Par exemple, dans le cas de $m(2,3,10,11) = x01x$, celui-ci permet d'exprimer 2 et 3 (10 et 11 ne nous intéressent pas).

Dans ce cas, on obtient :

Termes générateurs	0010 **(2)**	0011 **(3)**	0100 **(4)**	0101 **(5)**	0110 **(6)**	1000 **(8)**	1001 **(9)**
$m(2,3,10,11) = x01x$	X	**X**					
$m(2,6,10,14) = xx10$	X				X		
$m(4,5,12,13) = x10x$			**X**	X			
$m(4,6,12,14) = x1x0$			X		X		
$m(8,9,10,11,12,13,14,15) = 1xxx$						**X**	**X**

TABLE 2.6. – Table des implicants premiers pour $G(a_1, a_2, a_3, a_4)$. Les croix indiquent ce que les termes générateurs peuvent exprimer. Les croix impliquant des termes essentiels (voir ci-dessous) sont en gras.

La première chose à faire, c'est de regarder quels sont les termes essentiels, en regardant quelles sont les colonnes dans lesquelles il n'y a qu'une seule croix. Ici, on voit que le terme $m(8,9,10,11,12,13,14,15) = 1xxx$ est le seul permettant d'exprimer 8 et 9, il est donc essentiel. On voit également que 3 ne peut être exprimé que par $m(2,3,10,11) = x01x$ et que 5 ne peut être exprimé que par $m(4,5,12,13) = x10x$, qui sont donc essentiels. Pour les traduire en fonction booléenne, on ne considère que les termes qui ne sont pas x, et on a donc $m(8,9,10,11,12,13,14,15) = a_1$, $m(2,3,10,11) = \bar{a}_2 \cdot a_3$, $m(4,5,12,13) = a_2 \cdot \bar{a}_3$. Il s'agit bien d'éléments de notre fonction simplifiée obtenue par Karnaugh.

Le « problème », c'est que aucun de ces 3 termes essentiels ne permet d'exprimer 6, pour lequel il faut choisir entre $m(2,6,10,14) = xx10$ et $m(4,6,12,14) = x1x0$, dont l'un des deux est redondant. On pourrait choisir au hasard (et tomber dans ce cas-ci sur la bonne réponse quoiqu'il arrive), mais une méthode plus sûre est d'utiliser soit **la réduction par domination**, soit **la méthode de Petrick** (en) , qui seront expliquées plus bas.

Ici, on peut en fait éliminer n'importe laquelle des deux lignes. Si on élimine la deuxième, on voit qu'on tombe sur $m(2,6,10,14) = a_3 \cdot \bar{a}_4$, qui est bien le dernier terme manquant à notre fonction simplifiée telle qu'obtenue par Karnaugh. Sinon, on peut éliminer la première ligne, qui permet d'obtenir $m(4,6,12,14) = a_2 \cdot \bar{a}_4$, qui est également possible. En effet, si on regarde le diagramme de Karnaugh, on voit qu'il y a deux possibilités.

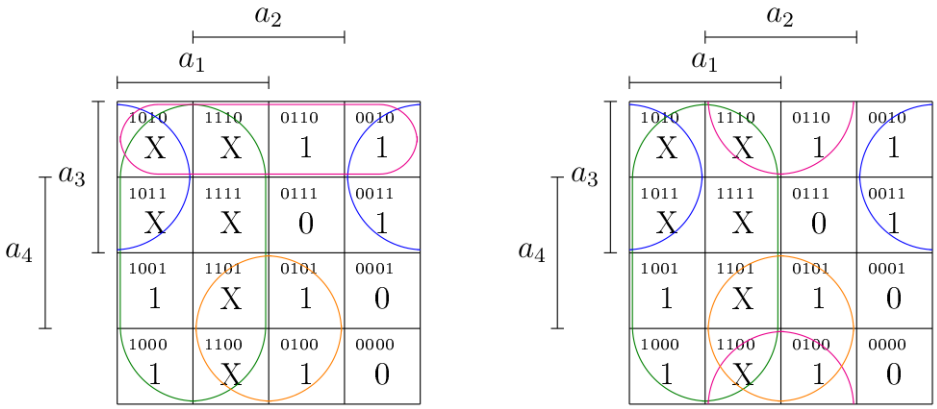


FIGURE 2.11. – Les deux possibilités de simplification pour $G(a_1, a_2, a_3, a_4)$: $a_1 + a_3 \cdot \bar{a}_4 + \bar{a}_2 \cdot a_3 + a_2 \cdot \bar{a}_3$ à gauche, $a_1 + a_2 \cdot \bar{a}_4 + \bar{a}_2 \cdot a_3 + a_2 \cdot \bar{a}_3$ à droite.

2.4.2. Un autre exemple pour bien comprendre la deuxième étape de l'algorithme

Intéressons-nous maintenant à la fonction suivante (qui n'a probablement pas d'intérêt autre que didactique) :

$$Q(a, b, c, d) = \sum m(0, 3, 5, 6, 7, 8, 9, 13, 14).$$

dont je vous épargne la recherche des termes générateurs (en fait ça s'arrête à l'ordre 1) pour vous donner tout de suite la table des implicants premiers.

Termes générateurs	**0**	**3**	**5**	**6**	**7**	**8**	**9**	**13**	**14**
$m(0,8) = x000$	**X**					X			
$m(3,7) = 0x11$		**X**			X				
$m(5,7) = 01x1$			X		X				
$m(5,13) = x101$			X					X	
$m(6,7) = 011x$				X	X				
$m(6,14) = x110$				X					**X**
$m(8,9) = 100x$						X	X		
$m(9,13) = 1x01$							X	X	

TABLE 2.8. – Tableau des implicants premiers pour $Q(a, b, c, d)$. Sont mises en gras les croix qui conduisent à des termes essentiels.

Tout d'abord, on voit que les termes essentiels sont $m(0, 8)$, $m(3, 7)$ et $m(6, 14)$, car ils sont les seuls à pouvoir exprimer 0, 3 et 14, respectivement.

On peut obtenir une table réduite en enlevant les termes essentiels et en éliminant les colonnes correspondantes (ainsi que les colonnes pour 6, 7 et 8, qui sont déjà couverts par ces termes

essentiels). Ici, on peut également enlever le terme $m(6,7)$ qui n'exprime rien de plus que ces 3 termes essentiels.

Termes générateurs	**5**	**9**	**13**
$m(5,7) = 01x1$	X		
$m(5,13) = x101$	X		X
$m(8,9) = 100x$		X	
$m(9,13) = 1x01$		X	X

TABLE 2.10. – Tableau simplifié des implicants premiers pour $Q(a,b,c,d)$.

Voyons maintenant comment faire le choix parmi touts ces termes redondants.

2.4.2.1. Par dominance

La règle de dominance est ainsi énoncée : « une colonne (ou ligne) est dominante relativement à une autre colonne (ou rangée) si elle contient toutes les "x" de l'autre, et au moins un "x" en plus ».

Méthode de la dominance

1. On peut éliminer une colonne égale ou dominante par rapport à une autre (afin de réduite le nombre de termes).
2. On peut éliminer une ligne égale ou dominée par rapport à une autre.
3. On extrait alors les nouveaux termes essentiels, puis on recommence si nécessaire.

Ici, aucune colonne n'égale ou ne domine une autre (elles contiennent le même nombre de « X » plus au moins 1). Par contre, on a de la domination dans les lignes : $m(5,7)$ est dominée par $m(5,13)$ et peut donc être éliminée. De même, $m(8,9)$ est dominée par $m(9,13)$, donc elle disparaît. On obtient alors la table suivante.

Termes générateurs	**5**	**9**	**13**
$m(5,13) = x101$	X		X
$m(9,13) = 1x01$		X	X

TABLE 2.12. – Tableau simplifié des implicants premiers pour $Q(a,b,c,d)$ après l'application de l'élimination par domination.

On voit alors que les deux termes restants sont essentiels, puisqu'ils permettent d'exprimer 5 et 9, respectivement. La fonction simplifiée est donc :

$$Q(a,b,c,d) = \bar{b}\cdot\bar{c}\cdot\bar{d} + \bar{a}\cdot c\cdot d + b\cdot c\cdot\bar{d} + b\cdot\bar{c}\cdot d + a\cdot\bar{c}\cdot d.$$

2.4.2.2. Par la méthode de Petrick

Méthode de Petrick

1. On nomme chaque ligne (chaque terme) de la table des implicants réduite L_i, où i est le numéro de la ligne (qui correspond donc à un terme redondant).
2. Chaque colonne est représentée par une disjonction (une somme) de ligne.
3. La fonction logique totale est représentée par la conjonction (un produit) des lignes.
4. On distribue ensuite les termes, et on réduit par le théorème d'absorption (pour rappel, $a + a \cdot b = a$). On obtient donc une conjonction (une somme) de disjonction (un produit) de lignes.
5. On sélectionne les produits **avec le moins de littéraux**. S'il y a plusieurs possibilités, on pourra alors choisir parmi ceux-ci.

On repart donc de la table simplifiée des implicants premiers, dans laquelle ont été rajoutés les numéros de lignes (étape 1) :

Termes générateurs	**Conjonction**	**5**	**9**	**13**
$m(5,7) = 01x1$	$L_1 = \bar{a} \cdot b \cdot d$	X		
$m(5,13) = x101$	$L_2 = b \cdot \bar{c} \cdot d$	X		X
$m(8,9) = 100x$	$L_3 = a \cdot \bar{b} \cdot \bar{c}$		X	
$m(9,13) = 1x01$	$L_4 = a \cdot \bar{c} \cdot d$		X	X

TABLE 2.14. – Tableau simplifié des implicants premiers pour $Q(a, b, c, d)$.

On voit que la fonction qu'on obtient (étape 2 et 3) est :

$$P = \underbrace{(L_1 + L_2)}_{5} \cdot \underbrace{(L_3 + L_4)}_{9} \cdot \underbrace{(L_2 + L_4)}_{13},$$

qui exprime simplement que la fonction finale doit être composée de tout les *minterms* restant (5, 9 et 13), d'où la conjonction, tandis que les disjonctions pour chacun de ceux-ci exprime qu'on peut choisir parmi les termes.
On la simplifie (étape 4) comme suit :

$$\begin{aligned}
P &= (L_1 + L_2)(L_3 + L_4)(L_2 + L_4) \\
&= (L_1 \cdot L_3 + L_1 \cdot L_4 + L_2 \cdot L_3 + L_2 \cdot L_4)(L_2 + L_4) \\
&= L_1 \cdot L_3 \cdot L_2 + L_1 \cdot L_4 \cdot L_2 + L_2 \cdot L_3 \cdot L_2 + L_2 \cdot L_4 \cdot L_2 + L_1 \cdot L_3 \cdot L_4 + L_1 \cdot L_4 \cdot L_4 + L_2 \cdot L_3 \cdot L_4 + L_2 \cdot L \\
&= L_1 \cdot L_2 \cdot L_3 + L_1 \cdot L_2 \cdot L_4 + L_2 \cdot L_3 + L_2 \cdot L_4 + L_1 \cdot L_3 \cdot L_4 + L_1 \cdot L_4 + L_2 \cdot L_3 \cdot L_4 \\
&= L_2 \cdot L_3 + L_2 \cdot L_4 + L_1 \cdot L_4
\end{aligned}$$

On a donc 3 combinaisons possibles (étape 5) :

1. L_2 et L_3, qui donnent la somme $b \cdot \bar{c} \cdot d + a \cdot \bar{b} \cdot \bar{c}$, soit 5 littéraux différents (b et $\bar{b}$ sont deux littéraux différents) ;
2. L_2 et L_4, qui donnent la somme $b \cdot \bar{c} \cdot d + a \cdot \bar{c} \cdot d$, soit 4 littéraux différents ;

3. L_1 et L_4, qui donnent la somme suivante $\bar{a} \cdot b \cdot d + a \cdot \bar{c} \cdot d$, soit 5 littéraux différents.

On voit donc bien que c'est L_2 et L_4 qui doivent être sélectionnées, ce qui était ce qu'on avait obtenu avec la réduction par dominance, et donc la fonction booléenne simplifiée est la même.

2.4.3. Conclusion

Encore une fois, cet algorithme n'est qu'une variation de la méthode de Karnaugh, mais est plus simple à expliquer à un ordinateur afin qu'il simplifie les fonctions booléennes pour nous. Une version encore plus efficace existe ceci dit, nommée Espresso (en) , exploitée par le programme du même nom et dont un certain nombre d'implémentations existent.

> *i* À nouveau, tout ce travail peut être réalisé en ligne. Parmi le nombre d'implémentations disponibles, je me suis pris d'affection pour celle-ci , qui est une fois encore en anglais et utilise des tirets à la place des x, mais qui présente bien chacune des étapes !

Dans ce chapitre, on a donc vu que l'algèbre de Boole permettait d'exprimer les mêmes choses que dans le chapitre précédent, mais sous une forme légèrement différente. On a aussi vu, et c'est très important, que ça nous permettait de simplifier ces fonctions logiques. On peut maintenant continuer notre périple pour aller voir comment on peut faire comprendre ça à une machine. À tout de suite

Contenu masqué

Contenu masqué n°3

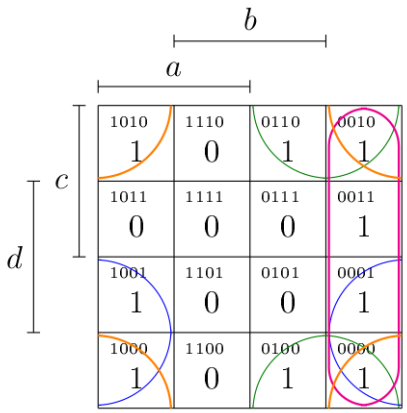


FIGURE 2.12. – Possibilités de groupement. J'avoue que la simplification en orange, il faut la voir.

Groupes de 4 cellules :

1. En vert : $\bar{a} \cdot b \cdot c \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot c \cdot \bar{d} + \bar{a} \cdot b \cdot \bar{c} \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} = \bar{a} \cdot \bar{d}$;
2. En bleu : $a \cdot \bar{b} \cdot \bar{c} \cdot d + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot d + a \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} = \bar{b} \cdot \bar{c}$;
3. En rose : $\bar{a} \cdot \bar{b} \cdot c \cdot d + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot d + \bar{a} \cdot \bar{b} \cdot c \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} = \bar{a} \cdot \bar{b}$;
4. En orange : $a \cdot \bar{b} \cdot c \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot c \cdot \bar{d} + a \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} + \bar{a} \cdot \bar{b} \cdot \bar{c} \cdot \bar{d} = \bar{b} \cdot \bar{d}$.

Une forme disjonctive simplifiée serait donc :
$g(a,b,c,d) = \bar{a}\cdot\bar{d} + \bar{b}\cdot\bar{c} + \bar{a}\cdot\bar{b} + \bar{b}\cdot\bar{d}$.
Pour ce qui est de la forme conjonctive, on groupe les cases de la manière suivante.

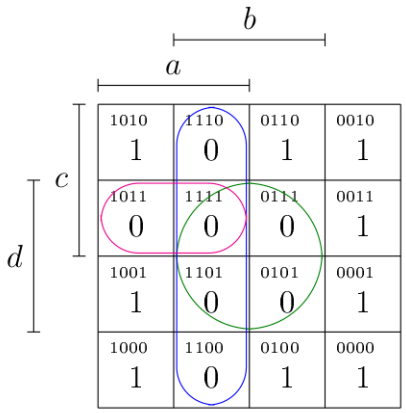


FIGURE 2.13. – Groupement pour la FNC.

On trouve donc que $\overline{g(a,b,c,d)} = a\cdot b + b\cdot d + a\cdot c\cdot d$, dès lors, une forme conjonctive simplifiée est :
$g(a,b,c,d) = \overline{a\cdot b + b\cdot d + a\cdot c\cdot d} = (\bar{a}+\bar{b})\,(\bar{b}+\bar{d})\,(\bar{a}+\bar{c}+\bar{d})$.
Ce n'est pas des plus simple, mais il faut toujours chercher à faire des groupes avec la plus grande taille possible. Retourner au texte.

Contenu masqué n°4

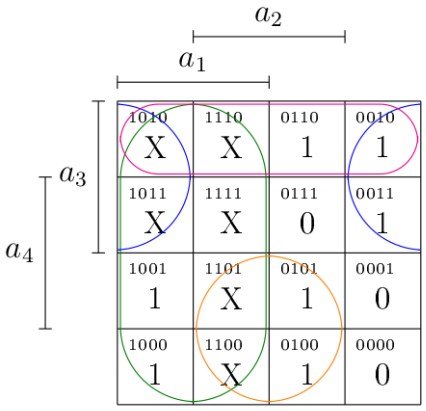


FIGURE 2.14. – Groupements pour simplification de la FND.

Dès lors,

$$G(a_1,a_2,a_3,a_4) = a_1 + a_3\cdot\bar{a}_4 + \bar{a}_2\cdot a_3 + a_2\cdot\bar{a}_3.$$

Évidemment, rien ne vous empêche de faire le travail pour les autres segments. Retourner au texte.

3. Des maths à l'électronique

Et maintenant, on va voir quels composants sont nécessaires pour réaliser des circuits logiques au niveau électronique, et un peu s'intéresser à ce qu'il y a dedans.

Ce chapitre tient, pour la plupart, de la culture générale, donc si vous le souhaitez, vous pouvez directement sauter à la dernière partie qui présente les portes logiques sans s'inquiéter de ce qu'il y a dedans (parce qu'en pratique, peu importe). Par ailleurs, différents chapitre du MOOC fabrication numérique explorent de manière plus détaillée la partie « circuits », tandis que le tutoriel d'Aabu sur les panneaux photovoltaïques explique le principe de fonctionnement d'un semi-conducteur.

3.1. La version « interrupteurs »

Comment représenter une variable propositionnelle au niveau de la machine ? Cela n'est pas très compliqué en soi, il suffit d'avoir un objet ou un signal possédant 2 états différents, un *on* (ou 1, ou « vrai ») et un *off* (ou 0, ou « faux »). L'idée la plus simple est alors de dire, par exemple, que la présence du courant correspond à 1 et l'absence de courant à 0.
Pour visualiser ça, pas besoin d'aller très loin. Il suffit d'imaginer un circuit électrique tout simple :

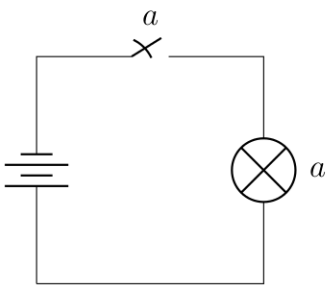


FIGURE 3.1. – Le plus simple circuit électrique du monde : une source de courant (à gauche), une lampe (à droite) et un interrupteur (au milieu). Ne faites pas attention à la résistance, elle est juste là pour éviter de provoquer un court-circuit.

Bon, pas besoin d'avoir fait Bac+5 ingénieur pour imaginer ce que fait ce circuit, non ? Si ? Allez : si l'interrupteur est ouvert, la lampe ne s'allume pas, s'il est fermé, le courant passe et elle s'allume. Sauf que c'est beaucoup moins idiot qu'il n'y parait : si l'interrupteur représente une variable a, la lampe représente le résultat (donc ici a). Et avec un tout petit peu plus de matériel, on peut commencer à faire des choses intéressantes. Par exemple, le circuit suivant.

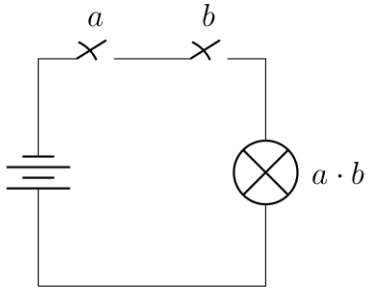


FIGURE 3.2. – Un circuit avec deux interupteurs.

Repartons du principe que chaque interrupteur représente une variable. Qu'est ce qui se passe si on teste les différentes possibilités de positions pour chacun des interrupteurs ?

Interrupteur a	**Interrupteur** b	**Lampe**
Ouvert	Ouvert	Éteinte
Ouvert	Fermé	Éteinte
Fermé	Ouvert	Éteinte
Fermé	Fermé	Allumée

Ceux qui ont une bonne mémoire se souviendront alors que ça ressemble fortement à la table de vérité du connecteur ET , pour la simple et bonne raison que c'est comme ça qu'on s'en sortirait en utilisant des interrupteurs. De la même manière, le circuit suivant donne la version électrique du connecteur OU.

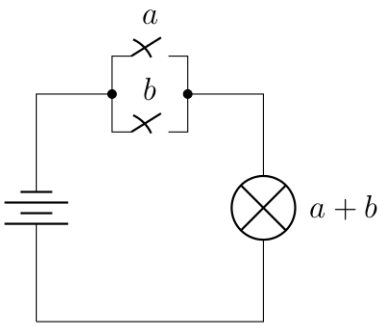


FIGURE 3.3. – Circuit implémentant la logique OU.

Vous pouvez vérifier, ça correspond bien à la même table de vérité 🍊

Contenu masqué n°5

Pour être totalement complet, voici finalement un circuit possible pour la négation (on est obligé d'utiliser un interrupteur un peu différent).

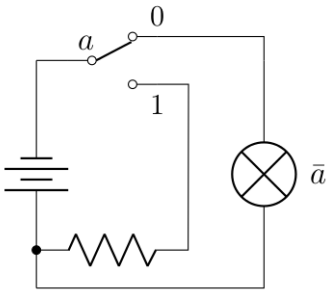


FIGURE 3.4. – Une manière d'implémenter la négation dans un circuit électrique : en fonction de la valeur de a, le circuit passe par la lampe (quand $a = 0$) ou pas (quand $a = 1$).

Donc, avec un certain nombre d'interrupteurs, des lampes et beaucoup de fils, on est capable de transformer n'importe quelle fonction logique en un circuit équivalent. Pas mal, non ? Pas mal, en effet, mais un peu trop beau pour être vrai. Même si le principe est à peu près équivalent, on a développé des solutions beaucoup plus facile à utiliser (mais rien ne vous empêche d'essayer, je vous renvoie au MOOC fabrication numérique pour les motivés).

3.2. Semi-conducteurs et diodes

3.2.1. Petits rappels d'électricité

Vous savez sans doute que l'électricité, ce n'est jamais qu'un déplacement d'électrons, ces petites charges négatives qui constituent la matière qui nous entoure en étant une des particules de l'atome (avec le proton et le neutron). Dans tout circuit électrique, les électrons se déplacent avec une certaine intensité, qui équivaut au nombre d'électron déplacés de la source à la destination. Cette intensité, I est mesurée en ampère, qui correspond au nombre de charges par unités de seconde, donc à un « débit » (voir une vitesse) de charge. Un peu comme un robinet que vous ouvririez plus ou moins fort.
À noter que le sens conventionnel du courant est défini comme étant l'inverse du sens physique : les électrons se déplacent du pôle négatif au pôle positif, mais **le courant part de la borne positive vers la borne négative**.
Une seconde grandeur intervient, c'est la tension, U, qui est l'énergie qu'on applique pour déplacer les électrons, que vous connaissez par son unité, les volts. Une manière autre de le voir, c'est celle du pôle positif/négatif (+/-), les électrons étant attirés par la tension positive et repoussés par la tension négative. Cette tension se mesure entre deux points dans un circuit électrique. Elle est fonction de l'intensité du courant qui circule entre les deux points et de la résistance du circuit entre ces deux mêmes points, par la célèbre loi $U = R \cdot I$, où R est la résistance (en ohms).
Les lois de l'électricité énoncent que la somme des courants entrant dans un circuit est égale à la somme des courant en sortant, tandis que le potentiel global d'un circuit fermé doit être nul (parce qu'il y a conservation de l'énergie). Dès lors,

- En *série* , l'intensité est la même en tout points du circuit, et les potentiels s'additionnent.
- En *parallèle* , la tension entre les branches est la même, mais les intensités s'additionnent et leur somme est égale à l'intensité au niveau des nœuds.

3.2.2. Des semi-conducteurs tout partout !

Dans la réalité un électron ne se déplace pas réellement d'un bout à l'autre du circuit. La réalité serait en effet plus proche d'une grande file où les électrons avanceraient petit à petit, en rencontrant plus ou moins de résistance. Et de résistance, il va justement en être question 🍊 En effet, un matériau dit **résistant**, c'est donc un matériau dans lequel le déplacement des électrons est difficile[8], voire impossible. À l'inverse, un matériau est dit conducteur si le déplacement des électrons dans celui-ci est facile. Les matériaux conducteurs, vous les connaissez bien, c'est par exemple le cuivre (le moins cher des trois), l'argent ou l'or. Et un matériau **semi-conducteur**, ben c'est entre les deux : là où une tension très faible suffit à faire circuler les électrons dans un conducteur, il faut atteindre une certaine tension seuil pour faire circuler le courant dans un semi-conducteur.
Pour visualiser ce qu'il se passe dans ces matériaux semi-conducteurs, on peut imaginer que les électrons circulent en passant de trous en trous, qui sont autant de défauts dans la structure du semi-conducteur. Ces « trous » peuvent être vu comme des charges positives, parce qu'il s'agit d'endroit microscopiques dans la structure du matériau où on constate une absence d'électron (défaut de charge négative = charge positive).
Par défaut, la tension à appliquer pour qu'un semi-conducteur soit utilisable est un peu trop importante pour qu'elle soit intéressante. ~~Comme dans le sport~~, on a alors recours au dopage et il existe deux manière de faire :

- Dopage n : on augmente le nombre d'électrons disponibles dans le semi-conducteur en question. On parle alors de porteurs de charge majoritaires négatifs et de porteurs de charge minoritaires positifs.
- Dopage p : à l'inverse, on augmente le nombre de trous.[9]

Là où ça devient intéressant, c'est qu'on peut alors créer une « jonction pn », soit un assemblage entre un semi-conducteur dopé p et un semi-conducteur dopé n. Schématiquement, on obtient quelque chose du genre.

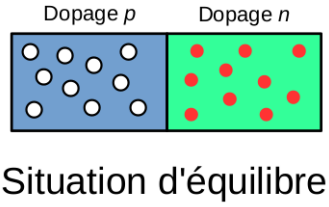


FIGURE 3.5. – Représentation d'une jonction entre semi-conducteurs dopés p (trous majoritaires, en blanc) et n (électrons majoritaires, en rouge). Notez bien que l'échelle est complètement exagérée.

Quelque chose d'intéressant à noter, c'est que cette jonction ne réagit pas de la même manière si on lui applique une tension dans un sens ou dans l'autre.

8. À l'impossible, nul n'est tenu : quand la tension est très importante, tout est possible.
9. En vrai : on a un matériau fait de SiO_2 cristallin, le silicium ayant 4 électrons de valence chacun liés à un oxygène. Si on inclut dans le réseau cristallin des atomes de la $5^{\text{ième}}$ colonne du tableau périodique (5 électrons de valence), il y a donc un électron de valence « de trop », donc un excès. À l'inverse, si on inclut dans la structure des atomes de la $3^{\text{ième}}$ colonne du tableau périodique (3 électrons de valence), on crée un trou. Rien de compliqué. 🍊

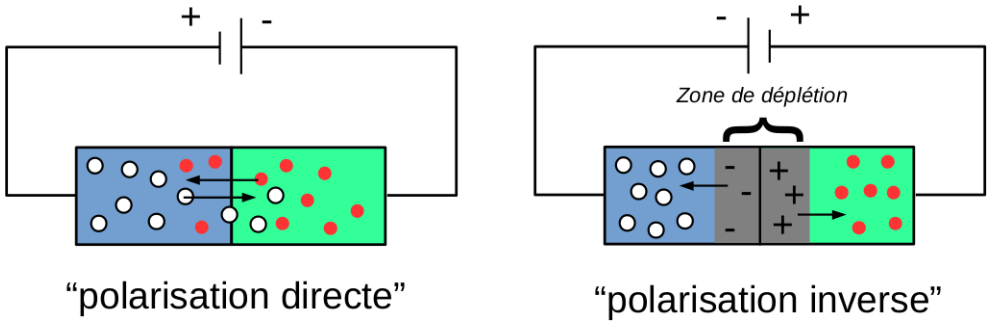


FIGURE 3.6. – La « polarisation » d'une jonction *pn* en fonction du sens de la tension, le comportement n'est pas le même : quand on est en polarisation inverse, le courant est bloqué (schéma très clairement inspiré de celui de Wikipédia).

Qu'est ce qui se passe ? Quand on est dans le sens de la polarisation directe, la borne positive attire les charges négatives, les électrons, à travers le semi-conducteur *p*. Les trous, quand a eux, ne se déplacent pas, mais comme les électrons sont attirés, ils laissent derrière eux des trous inoccupés, d'où l'impression qu'ils se déplacent également vers la borne négative. Mais quand on applique la tension dans l'autre sens (polarisation inverse), rien ne va plus ! Les électrons, toujours attirés par la borne positive, se déplacent dans cette direction, laissant des trous dans le semi-conducteur *p* derrière eux, et les électrons du semi-conducteur *n* font de même, laissant derrière eux une zone appelée *zone de déplétion* et qui contient les porteurs de charges minoritaires (de signes opposés, mais en nombre moins importants). Autrement dit, dans ce sens-là, les porteurs de charge majoritaires s'éloigne de la jonction, ce qui a pour effet que le courant ne passe plus dans le matériau.
On a donc un matériau qui laisse passer le courant dans un sens (quand le semi-conducteur *p* est relié à la borne **P**ositive et le semi-conducteur *n* est relié à la borne **N**égative, notez la mise en évidence) et pas dans l'autre, ce qu'on appelle aussi une **diode**.

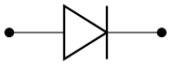


FIGURE 3.7. – Représentation d'une diode dans un schéma électrique

Dès lors, dans ce schéma de la diode, qui représente une flèche, définit le « sens passant » si le potentiel à gauche est plus important (plus positif) que celui de droite (sens conventionnel du courant).

i

Historiquement, les diodes étaient à la base des tubes sous vides dans lesquels un flux d'électron parcouraient le vide entre une anode et une cathode à condition que la tension entre ces deux composants soit suffisamment importante. On les retrouve encore aujourd'hui, par exemple dans les amplis dit « à lampe » pour joueurs de guitare.

3.2.3. Faire des circuits logiques avec des diodes

Ces diodes permettent déjà d'implémenter d'une manière différente les portes logiques citées plus haut (voir à ce sujet l'article Wikipédia correspondant (en) si ça vous intéresse). Ainsi, voici par exemple un montage qui représente un connecteur OU et un ET.

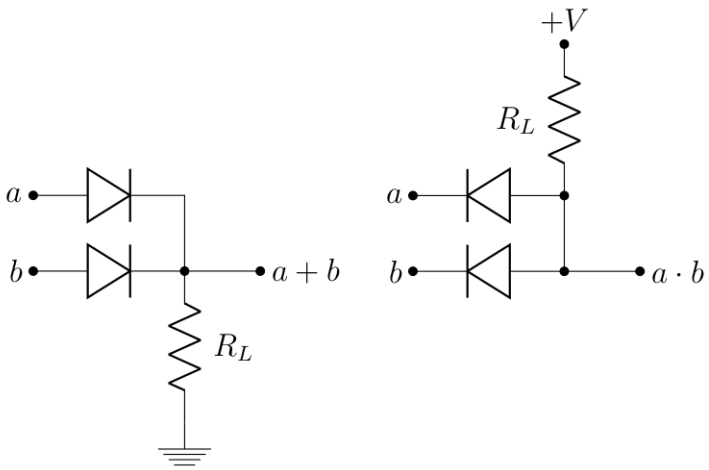


FIGURE 3.8. – Circuits implémentant les logiques OU (gauche) et ET (droite). Notez que le sens des diodes est inversé d'un cas à l'autre. La résistance R_L est là pour « protéger » la diode contre un courant trop important qui « grillerait » celle-ci (voir, une fois encore, le MOOC fabrication numérique pour plus d'explications).

Le schéma associé à OU est plus facile à comprendre : s'il n'y a aucune tension appliquée sur a ou b, aucun courant ne passe et la sortie est également à une tension de 0 volt, ce qui correspond à « faux ». Si on applique cette fois une tension positive $+V$ sur a et/ou b, la tension mesurée en sortie sera également positive, ce qui correspondra à « vrai ».
Le schéma associé à ET est juste un tout petit peu plus complexe : s'il n'y a aucune tension appliquée sur a ou b, la tension $+V$ est dissipé à travers les diodes, par les lois associées aux tensions, il n'y a pas de tension en sortie. Cela reste vrai tant qu'une tension positive $+V$ n'est pas appliquée sur chacune des deux entrées a et b.

i

En pratique, les montages utilisant les diodes ne sont plus utilisés, les diodes possédant une certaine résistance, provenant du passage des charges dans le semi-conducteur (autrement dit, elles ont besoin d'une tension minimale pour être actives, tout comme leur équivalent sous forme de tube à vide, quoique moins importante que ces derniers).

3.3. Jonctions et transistors à effet de champ

L'idée du semi-conducteur était quand même bonne et on va voir comment il est possible de l'exploiter de manière plus efficace.

3.3.1. Les transistors bipolaires

On va donc assembler, à la suite, un premier semi-conducteur qu'on va doper de manière assez importante et qu'on va appeler, de ce fait, **l'émetteur** (E), suivit d'un semi-conducteur de dopage opposé, relativement fin, et qu'on appellera **la base** (B), puis finalement un troisième émetteur de même type de dopage que le premier, qu'on appellera **le collecteur** (C). La jonction peut donc être de type NPN ou PNP, en fonction du dopage du premier semi-conducteur[10].

10. À priori et puisque la vitesse de diffusion des électrons est plus importante que celle des trous, c'est la jonction NPN qui est préférée.

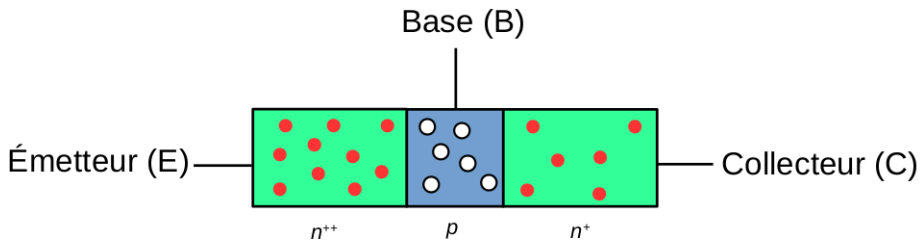


FIGURE 3.9. – Jonction de type NPN. Les $^{+}$ et $^{++}$ n'indiquent pas la charge, mais le fait que l'émetteur est plus dopé que le collecteur. À noter qu'en pratique, le montage n'est pas linéaire .

On forme alors des transistors[11] dit « bipolaires » du fait qu'ils possèdent 3 points de contacts, contre deux pour les éléments de circuits plus classiques qu'on a rencontré jusqu'ici.
On a vu plus haut que pour qu'il y ait passage du courant, la jonction doit être polarisée dans le sens direct. On se doute donc bien que si le potentiel de la base (U_B) est plus négatif que celui de l'émetteur (U_E) et du collecteur (U_C), rien ne se passe ($U_E > U_B < U_C$). Néanmoins, c'est en jouant sur le potentiel de la base que cette jonction révèle tout son... Potentiel.

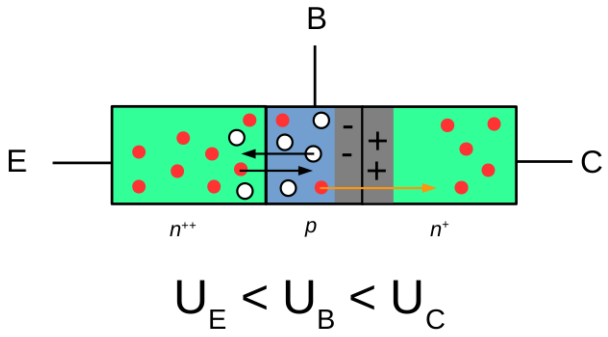


FIGURE 3.10. – Jonction NPN en mode actif : le potentiel entre la base et l'émetteur est positif, donc cette jonction est en polarisation directe. À l'inverse, la jonction entre la base et le collecteur est en polarisation inverse, mais la faible épaisseur de la base permet aux électrons amené par l'émetteur de passer dans le collecteur (flèche orange). Le principe serait le même avec un dispositif PNP, mais ce serait alors les trous qui se « déplaceraient ».

Ce qui est très intéressant, c'est qu'on a un dispositif qui peut alors jouer le rôle d'un interrupteur, en laissant passer le courant ou non en fonction du potentiel appliqué à la base[12]. Ainsi, si on reste dans le cas d'un transistor NPN, le potentiel du collecteur étant fixé comme supérieur à celui de l'émetteur et de la base, l'interrupteur est fermé si la tension appliquée à la base est supérieure à celle de l'émetteur ($U_{BE} = U_B - U_E > 0$) et ouvert si cette tension est inférieure ($U_{BE} < 0$). Il est donc assez logique d'imaginer qu'on puisse s'en servir pour créer un circuit. Dans ledit circuit, une jonction npn se représente comme suis.

11. En anglais, « *bipolar junction transistor* », ou plus simplement BJT.
12. Et même si ça ne nous intéresse pas ici, d'amplificateur, car ce type de jonction est créée de telle manière à ce que le courant collecté réagisse en fonction du courant appliqué à la base, mais avec un facteur 100 ou 1000, par exemple. Vu que les MOSFET sont préférés dans les portes logiques, c'est dans les amplificateurs qu'on retrouve ces transistors.

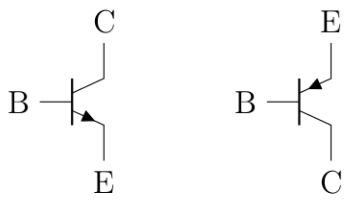


FIGURE 3.11. – Représentation d'une jonction NPN (à gauche) et PNP (à droite). Notez que la flèche, comme pour la diode, indique le sens de polarisation direct (émetteur-base, comme décrit plus haut) dans le sens **conventionnel** du courant. Notez aussi que dès lors, émetteur et collecteurs sont inversés.

Et voici donc les circuits correspondant aux différents composants logiques.

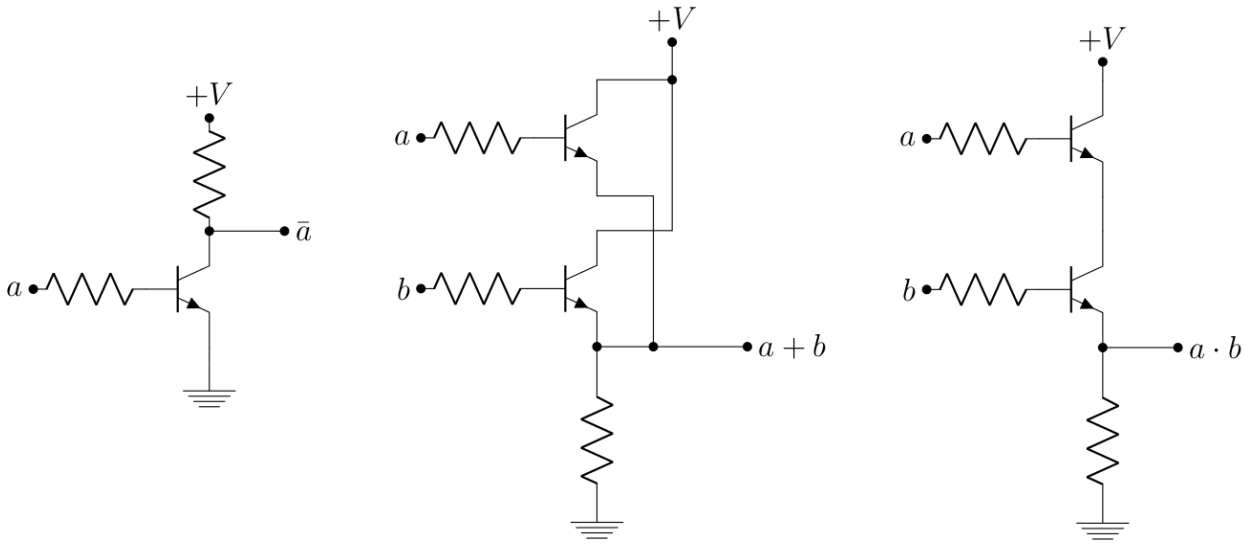


FIGURE 3.12. – Implémentation de la logique de négation (gauche, source), OU (milieu, source) et ET (droite, source) avec des transistors NPN. Notez bien que si deux fils se croisent mais qu'il n'y a pas de point à leur intersection (ici dans le schéma associé à OU, au milieu), **alors il n'y a pas de connexion** entre eux. Il est bien entendu possible de faire de même avec des transistors PNP. Oh, et comme d'habitude, les résistances sont là pour des raisons qui tiennent de l'électronique et non de la logique 🍊

Si vous avez bien intégré que le transistor fonctionne à peu près comme un interrupteur dont le potentiel sur la base déterminerait l'état ouvert ou fermé, vous devriez normalement ne pas avoir trop de difficulté à comprendre ses schémas. Ainsi, si on prend par exemple le cas de la négation, si on applique pas de tension sur a (ce qui correspond à « faux »), l'interrupteur est alors « ouvert » et alors la sortie sera à une tension de $+V$, soit « vrai ». À l'inverse, si on applique une tension sur a, l'interrupteur est « fermé », et le potentiel en sortie est alors nul (la terre est par définition à 0 volt). C'est bien le principe de la négation (ou de ce que les Anglais appellent parfois *inverter*).
Le même principe d'interrupteur s'applique pour les schémas associés à ET et OU, qui reviennent alors à ce qu'on avait montré au début de ce chapitre , vu que c'est la tension appliquée à la base (donc les entrées a et b) qui déterminent l'ouverture de l'interrupteur ou pas.

En pratique, ces composants sont plus gourmands en énergie (et dissipent plus de chaleur) que leurs équivalent à effet de champ, même si plus rapides. À l'heure où on aimerait bien qu'un portable tienne plus d'une heure, c'est bel et bien ces derniers qui sont utilisés la plupart du temps.

3.3.2. MOS (*metal-oxyde-semiconductor*), FET (*field-effect transistor*) et CMOS (*complementary MOS*)

Le concept d'un interrupteur dont l'ouverture et la fermeture est commandée par la tension appliquée à la base étant quand même vraiment intéressant, à peu près à la même époque a été imaginée la technologique dite des transistors à effet de champ[13]. L'effet est en pratique similaire, mais la mise en oeuvre est légèrement différente.
Pour ça, il faut explorer un autre effet, celui qu'on trouve en fait dans un autre élément de circuit classique, les condensateurs . Un condensateur est formé de deux plaques de métal, séparées d'un isolant électrique. Lorsqu'on applique un potentiel, disons positif, sur une des faces du condensateur, on génère des charges négatives sur l'autre plaque, générant de ce fait un champ électrique entre les deux plaques du condensateur.

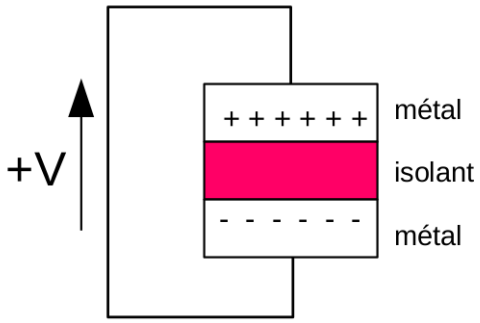


FIGURE 3.13. – Principe de fonctionnement d'un condensateur en charge : des charges positives sont formées sur la face supérieure du condensateur, induisant alors des charges négatives sur la face inférieure. On crée donc un champ électrique.

Or, cet effet peut être utilisé de manière intéressante si on construit le bon montage. Voici comment on réalise un transistor à effet de champ (**FET**).

13. Les transistors à effet de champs (principe breveté en 1925, CMOS breveté en 1936) sont même une invention antérieure aux transistors bipolaires (1947 et 1948, aux laboratoires Bell). Sauf que ni l'un, ni l'autre n'étaient facile à mettre en œuvre industriellement.

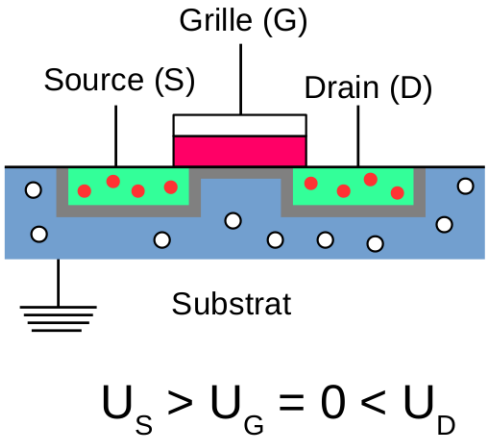


FIGURE 3.14. – Illustration d'un transistor à effet de champ (FET) de type nMOS (voir texte). Le potentiel du drain est bien entendu supérieur à celui de la source.

Tout comme le transistor bipolaire NPN, on retrouve ici le mélange des deux types de semi-conducteurs. Ici, on a un transistor dit « nMOS », car ce sont des semi-conducteurs de type n qui vont fournir les charges. Le terme MOS provient du fait qu'on utilise un oxyde métallique (*metal oxyde*) comme isolant entre la couche de métal et le semi-conducteur p, afin de former la **grille** (G, *gate* en anglais, qui signifie alors « porte »). Les deux semi-conducteurs n sont ici appelés **la source** (S) et **le drain** (D, *to drain* signifiant en anglais « vider » ou « vidanger »), tandis que le semi-conducteur p est appelé le substrat.
On voit qu'on s'arrange pour placer les jonctions pn en polarisation inverse, en plaçant un potentiel positif sur la source et le drain, tandis que le substrat est placé à la terre (0 volt). Dans cette configuration, aucun courant ne circule donc entre la source et le drain. Mais cette fois-ci, le rôle de la grille ne va pas être de placer une jonction en polarisation directe ! Appliquer un potentiel sur celle-ci va plutôt créer des charges négatives dans la jonction entre l'oxyde (rose sur le schéma) et le substrat, induisant un champ électrique (d'où le terme FET, *field induced transistor*) qui va alors s'opposer aux porteurs de charges minoritaires du substrat, créant de ce fait un « canal » de conduction où les électrons de la source peuvent alors se promener librement vers le drain. En d'autres termes, le courant passe !

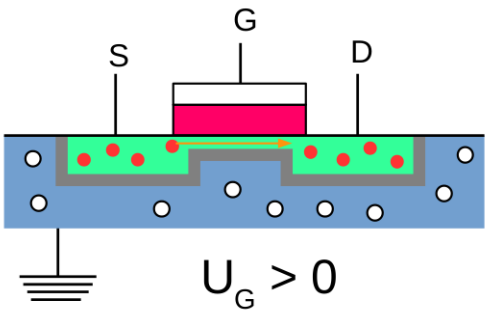


FIGURE 3.15. – Transistor à effet de champ nMOS avec une tension de grille positive, ce qui induit le passage des électrons de la source au drain.

Il est évident que ce type de transistor permet de réaliser le même genre de circuit que ceux présentés pour la jonction npn (voir par exemple ici). Les représentations de ces deux transistors sont les suivantes.

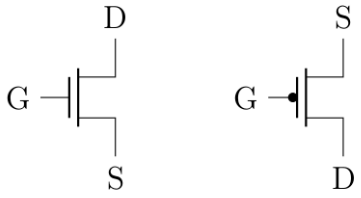


FIGURE 3.16. – Représentation d'un transistor nMOS (gauche) et pMOS (droite) dans un circuit. Notez que là où un potentiel positif sur la grille permet le passage du courant pour un transistor nMOS, c'est l'inverse pour un transistor pMOS.

En soi, rien de neuf sous le soleil, il s'agit toujours d'un transistor qui s'active quand on lui applique un potentiel. Mais qu'est ce qui se passe lorsqu'on met des nMOS et pMOS ensemble ? Observons par exemple le schéma suivant.

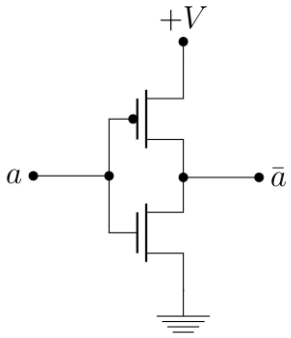


FIGURE 3.17. – Implémentation d'une logique d'inversion à l'aide d'un nMOS (en dessous) et d'un pMOS (au-dessus).

Pourquoi est-ce intéressant ? Parce que si on applique pas de tension sur a, alors c'est le pMOS qui laisse passer le courant (on vient de voir que le nMOS était alors bloquant), donc la sortie est a $+V$ («vrai»). À l'inverse, à l'application d'une tension sur a, c'est le nMOS qui laisse passer le courant, ce qui place la sortie à un potentiel nul («faux»). C'est ce qu'on appelle un circuit **CMOS**, le «C» étant pour complémentaire (puisqu'on utilise une paire de pMOS et de nMOS).
Le gros avantage de cette manière de fonctionner, c'est que $+V$ et la terre ne sont jamais en contact. En effet, dans les montages impliquant des transistors bipolaires, l'ouverture de ceux-ci finissait toujours par mettre en contact le pôle positif et négatif, ce qui consomme de l'énergie. Là, ce n'est jamais le cas.[14] C'est pour cela qu'on a finalement favorisé ce genre d'architecture et que les CMOS sont aujourd'hui utilisés dans les CPU et les mémoires (voir chapitre suivant).

Contenu masqué n°6

14. ... Sauf quand les deux MOS sont en train de changer de sens, là, on a une perte d'énergie, puisqu'il arrive un bref moment où les deux dispositifs sont passants. Du coup, si on change régulièrement d'état, on consomme d'autant plus d'électricité. Il y a tout de même moyen de contrôler ce phénomène (en) , mais jamais en dessous d'une certaine limite.

3.4. Des portes, des portes et des portes !

Bon, en pratique, tout ce qui vient avant n'est pas réellement nécessaire. Tout ce qu'il est important de savoir (sauf si vous voulez réaliser un circuit ayant une application commerciale réelle), c'est qu'il est possible de réaliser des circuits logiques. Pour la suite, on va donc ranger sous le tapis ces histoires de potentiel et de semi-conducteurs pour se concentrer sur les circuits eux-mêmes. Pour ça, on a besoin de composants logiques, qu'on va nommer **portes logiques** (ou, plus simplement, porte). Voici les 3 portes de bases :

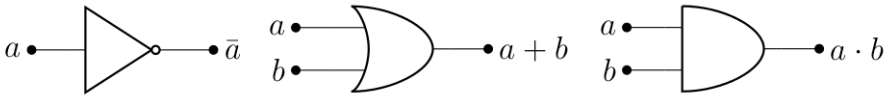


FIGURE 3.18. – Les 3 portes logiques principales : la négation (gauche), la porte OU (milieu) et la porte ET (droite).

À ça, on rajoute 3 portes qu'on a déjà rencontré dans le premier chapitre et qui se retrouvent dans certains circuits[15] (même si le processus de simplification de Karnaugh ne permet pas de les faire apparaître).

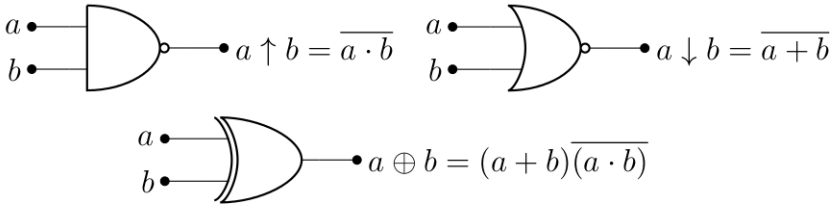


FIGURE 3.19. – 3 portes logiques secondaires : NAND (gauche), NOR (droite) et XOR (dessous). Remarquer que les deux premières portes sont semblables à leur équivalent non-nié, mais qu'il y a une bulle supplémentaire à la fin.

Ce qu'il est important de savoir, c'est que **passer au travers d'une porte demande du temps**. Bien entendu, on ne parle pas d'un temps mesuré en seconde, mais à l'échelle de la fréquence d'un processeur (qui se mesure en MHz, donc en millions d'instructions à la seconde), ça devient limitant, ce qui signifie qu'il faut que le nombre de porte par lesquelles voyage un signal devrait être le plus réduit possible, légitimant alors le procédé de simplification vu plus haut (de plus, ça rend les circuits électriques plus lisibles).
Ainsi, si on reprend l'exemple du chapitre précédent, on avait

$$\begin{aligned} f(a,b,c) &= a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c} && \text{(FND)} \\ &= a \cdot \bar{b} + a \cdot \bar{c} && \text{(forme disjonctive, après simplification)} \\ &= a\,(\bar{b} + \bar{c}) && \text{(forme conjonctive, après simplification)} \end{aligned}$$

Voici tout d'abord le circuit correspondant à la forme disjonctive.

15. Il est même possible d'implémenter toutes les autres portes avec juste une porte NAND ou NOR. Par exemple, le système de guidage d'Apollo (utilisé par la mission Apollo 11) était construit uniquement avec des NOR.

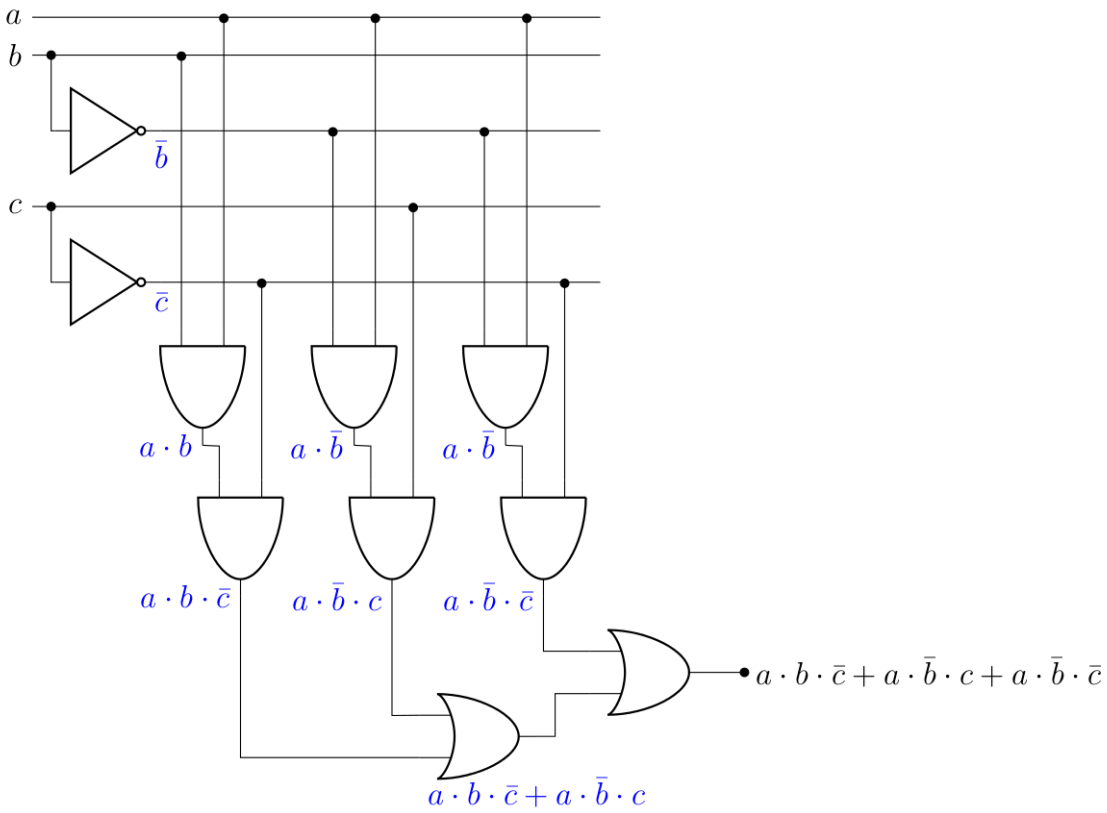


FIGURE 3.20. – Circuit correspondant à $a \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot c + a \cdot \bar{b} \cdot \bar{c}$ (forme normale disjonctive). La formule à la sortie de chacune des portes est notée en bleue.

C'est long, mais assez systématique :

1. Pour chaque variable, on «tend un câble» pour celle-ci et sa négation, si c'est nécessaire (ici, $\bar{a}$ n'est pas nécessaire).
2. On construit ensuite les termes conjonctifs (pour rappel, c'est aussi ce qu'on appelle les *minterms*, $x \cdot y$) en utilisant les portes «ET».
3. Puis finalement, on crée les disjonctions $(x + y)$ en rassemblant les conjonctions à l'aide de portes «OU».

Si vous comptez, vous verrez que chaque signal passe en moyenne par 4 portes (5 dans le pire des cas, 3 dans le meilleur des cas). Si on compare maintenant à la version disjonctive simplifiée, construite selon le même principe...

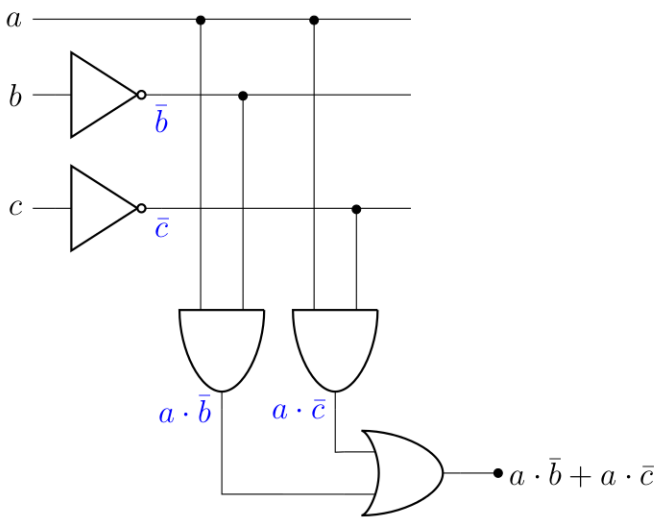


FIGURE 3.21. – Circuit correspondant à la formule $a \cdot \bar{b} + a \cdot \bar{c}$, tout à fait équivalente à la précédente.

...On constate que le signal passe cette fois en moyenne par 3 portes (2 dans le meilleur des cas), donc un gain d'au moins une porte. Par ailleurs, le nombre total de portes nécessaire pour implémenter le circuit passe de 10 à 5, soit une réduction de moitié. Très clairement, la simplification est intéressante dans ce cas et d'ailleurs dans la plupart des cas.
Néanmoins, il est possible d'obtenir une version de ce circuit qui n'utilise que 4 portes en dessinant la forme conjonctive. Saurez-vous le dessiner ?

Contenu masqué n°7

Bien entendu, la forme conjonctive n'est pas forcément plus courte que la disjonctive et il faut tester les deux pour le savoir (ou, avec un peu d'entraînement, compter directement dans l'expression de base).
Et si on s'autorise l'utilisation des fameuses «portes logiques secondaires», il est encore possible d'obtenir quelque chose de plus simple.

Contenu masqué n°8

On voit donc ici que la simplification de Karnaugh est un outil intéressant, même s'il est parfois possible d'aller au-delà.

Au final, ce qu'il est réellement important de retenir de ce chapitre, c'est sa fin, à savoir :

1. Il existe des symboles pour les 3 opérations logiques de bases, c'est-à-dire la négation, ET et OU.
2. La simplification de Karnaugh, vu au chapitre précédent, permet de réduire drastiquement le nombre de portes à utiliser pour implémenter un circuit logique (même s'il est possible d'aller plus loin grâce aux portes NAND, NOR ou XOR).

Nous sommes à la moitié du parcours 🍹 ! Dans le prochain chapitre, on verra comment il est possible de représenter des nombres afin de les manipuler dans des circuits logiques, donc ce qui se cache derrière la représentation binaire. 🍊

Contenu masqué

Contenu masqué n°5

Interrupteur a	**Interrupteur** b	**Lampe**
Ouvert	Ouvert	Éteinte
Ouvert	Fermé	Allumée
Fermé	Ouvert	Allumée
Fermé	Fermé	Allumée

C'est bien équivalent à la table de vérité du connecteur OU : éteint uniquement si les deux interrupteurs sont ouverts, autrement dit $a + b = 0 \Leftrightarrow a = b = 0$. Retourner au texte.

Contenu masqué n°6

En se grattant un peu la tête, on se rend compte qu'un cMOS correspond à une porte logique de type NOR.

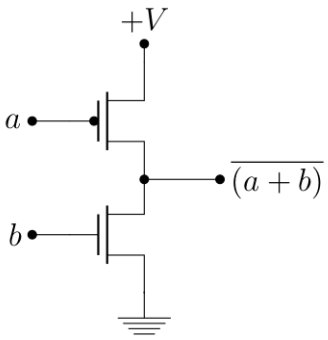


FIGURE 3.22. – Un circuit CMOS

Pour preuve, voici la table de vérité du CMOS, exprimé en termes de potentiel.

a	b	**sortie CMOS**
0	0	$+V$
0	$+V$	0 (et énergie perdue)
$+V$	0	0
$+V$	$+V$	0

Et revoici la table de vérité du NOR.

a	b	$a+b$	$\overline{a+b}$
0	0	0	1

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

On voit bien que ça correspond. Dès lors, l'exercice revient à exprimer un circuit logique en termes de portes NOR. Par exemple, voici une implémentation pour NAND.

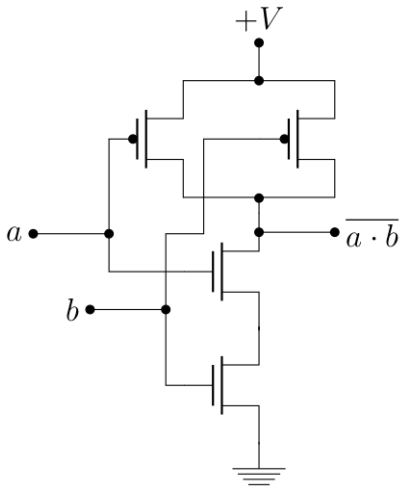


FIGURE 3.23. – Implémentation de la logique NAND («NON-ET»). Ça commence à tenir du jeu de piste, par contre on respecte toujours ce principe de la connexion d'une entrée à un nMOS et un pMOS.

Retourner au texte.

Contenu masqué n°7

La forme normale conjonctive demande en effet 4 portes pour être implémentée :

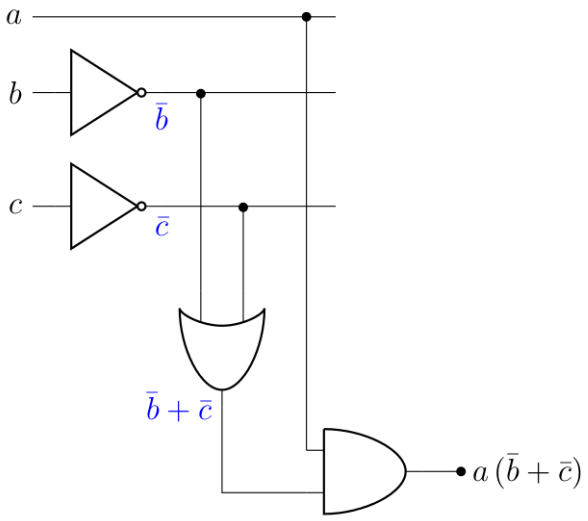


FIGURE 3.24. – Forme conjonctive, $a\,(\bar{b} + \bar{c})$

Retourner au texte.

Contenu masqué n°8

Il suffit de partir de la version conjonctive et d'utiliser une porte NAND («NON ET»). En effet, la loi de De Morgan énonce que $b \uparrow c = \overline{b \cdot c} = \bar{b} + \bar{c}$, ce qui est bien le terme disjonctif rencontré dans l'expression précédente. On obtient alors :

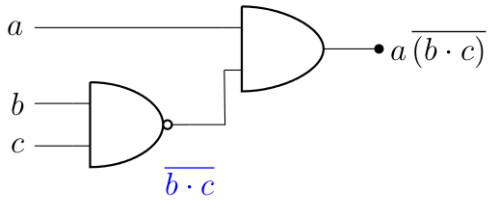


FIGURE 3.25. – Version «ultime» du circuit précédent, en utilisant que deux portes. Même si on a vu plus haut que la porte «NON ET» peut être légèrement plus complexe au niveau électronique, ça reste plus avantageux que le circuit précédent.

Retourner au texte.

4. Un détour nécessaire : le binaire

Un ordinateur, vous commencez à le sentir, c'est entre autre un très long et très complexe circuit logique (voir même plusieurs). Néanmoins, le but de ce circuit est de réaliser des opérations et entre autre des opérations mathématiques «classiques» sur des nombres quelconques. La question qu'on va se poser ici, c'est comment on peut représenter ces nombres au niveau de la machine ?

4.1. C'est la base !

Sans refaire tout un cours de numération, nous travaillons, dans la vie de tout les jours, avec la base 10. Ça signifie que «10» est le premier **nombre** (c'est-à-dire une suite de chiffre) qui nécessite deux **chiffres** pour être représenté. Par exemple, si on travaillait en base 8, ça signifie que pour représenter 8, il faudrait écrire... «10». Pour ne pas confondre, on va plutôt écrire 10_8, c'est-à-dire *«10» en base 8*. Et en binaire, c'est-à-dire en base 2, 2_{10} est représenté par 10_2 (c'est-à-dire *«10 » en base 2*).

4.1.1. Conversion en base 10

Instinctivement, on sait que si on a par exemple 1235_{10}, c'est équivalent à $\{1 \times 1000 + 2 \times 100 + 3 \times 10 + 5\}_{10}$ (j'utilise ici les accolades pour indiquer que tous les nombres sont en base 10). Mais s'il me prend l'envie de le réécrire d'une manière légèrement différente, on se rend compte de l'usage de la base :

$$1235_{10} = \{1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0\}_{10},$$

ce qui signifie qu'à chaque fois, le chiffre est multiplié par la base exposant la position dans le nombre, lu de droite à gauche (et en commençant à 0). Un nombre a composés de n chiffres a_i dans une base donnée peut donc s'écrire comme $a_{n-1}a_{n-2}\dots a_0$. Un nombre $a_{\mathfrak{B}}$ dans une base $\mathfrak{B}$ correspond, en base 10, à :

$$a_{10} = \left\{ \sum_{i=0}^{n-1} a_i \, \mathfrak{B}^i \right\}_{10}$$

où a_i représente le chiffre à la position i. Cette formule est en fait très pratique pour passer d'une base $\mathfrak{B}$ à la base 10. Ainsi,

$$125_7 = \{1 \times 7^2 + 2 \times 7^1 + 5 \times 7^0\}_{10} = 68_{10},$$
$$100101_2 = \{1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\}_{10} = \{2^5 + 2^2 + 2^0\}_{10} = 37_{10}.$$

Comme on le voit, c'est assez systématique et à condition de jongler avec les puissances, c'est assez rapide à mettre en place.
Pour vous entraîner, convertissez 1085_9, 11213_4 et $1F75_{16}$ en base 10, sachant que pour la base 16 (hexadécimale), on considère que $10_{10} = A_{16}$, $11_{10} = B_{16}$, et ainsi de suite.

Contenu masqué n°9

Dans le cas du binaire, on a une suite de *bits*, valant 0 ou 1. Le bit le plus à droit est nommé *bit de poids faible* tandis que celui le plus à gauche est nommé *bit de poids fort*.[16]

4.1.2. Conversion d'une base 10 vers une base quelconque

Si on veut convertir un nombre donné en base 10 vers une base quelconque, disons $\mathfrak{B}$, il existe une méthode (puis une seconde dans le cas du binaire, même si elle repose sur la même astuce).

4.1.2.1. Méthode générale

La méthode consiste à diviser (**par division euclidienne**, donc entière) le nombre en question par $\mathfrak{B}$, puis à écrire le reste de cette division à côté, pour ensuite diviser le quotient par $\mathfrak{B}$ et ainsi de suite jusqu'à ce que le quotient obtenu soit égal à 0.
Par exemple, si on veut convertir 137 en base 5, on divise 137 par 5. On obtient 27, avec 2 comme reste. On divise 27 par 5, on obtient 5, avec 2 pour reste et ainsi de suite.

Quotient	**Reste**
137	2
27	2
5	0
1	1
0	

Le nombre converti s'obtient en lisant ensuite la colonne des restes à l'envers. On a donc 1022_5, et vous pouvez vérifier , c'est correct. Vous pouvez donc vous entrainer à convertir 1256_{10} en base 7, puis 645_{10} en base 2 (en binaire, quoi).

Contenu masqué n°10

Encore une fois, c'est assez systématique. Par contre, c'est relativement long, surtout pour les petites bases.

4.1.2.2. Le cas particulier du binaire

La seconde méthode consiste à écrire toutes les puissances de 2 dans un tableau de droite à gauche, tant que 2^x est plus petit que le nombre que vous devez convertir. Ensuite, on soustrait la plus grande puissance écrite au résultat, et on écrit 1 en dessous de cette puissance. On sélectionne alors la plus grande puissance inférieure à ce résultat, qu'on soustrait, en indiquant

16. Dans un scénario de type *big-endian*. Ce qui est la méthode naturelle pour écrire les nombres (chiffre de poids le plus fort à gauche), mais n'est pas la méthode employée sur un certain nombre d'architectures modernes, qui sont *little-endian*. Ici, **on va travailler en *big-endian*** et ne pas s'en soucier pour ne pas complexifier les explications (on garde donc l'ordre usuel).

1 en dessous de la puissance. Et ainsi de suite, jusqu'à ce qu'on obtienne 0. Les puissances qui ne sont pas utilisées sont marqués à 0 et le nombre en binaire est ainsi obtenu.
Par exemple, convertissons 724_{10} en binaire. La plus grande puissance de 2 inférieure à 724, c'est 512 (2^9). On a donc :

512	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	.	.	.	.	.	.	.	.	.

si on soustrait 512 à 724, on obtient 212. On voit que la plus grande puissance de 2 inférieure à 212, c'est 128. On marque 1 dans la colonne 128 (et zéro dans la colonne 256), et on soustrait 128 à 212 (on obtient 84).

512	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	1	.	.	.	.	.	.	.

Et ainsi de suite. Finalement, on obtient le tableau suivant :

512	**256**	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	0	1	1	0	1	0	1	0	0

ce qui signifie que $724_{10} = 1011010100_2$, ce qui est la bonne réponse . C'est un peu plus rapide à faire, selon moi, surtout si le nombre une fois converti en binaire ne contient pas que des 1_2 (auquel cas c'est juste aussi long que la méthode ci-dessus). Néanmoins, les deux méthodes donnent le même résultat, donc c'est comme vous souhaitez.

4.1.3. Binaire et représentation des nombres

En mathématiques, on ne s'inquiète pas trop de la place que prend un nombre. On sait à peu près qu'il en existe une infinité et qu'on obtient toujours le suivant en faisant « +1 » et le précédent en faisant « -1 ». Bon, parfois, il faut écrire très petit sur sa feuille, parfois il faut utiliser de moyens alternatifs (comme la notation scientifique , qu'on retrouve aussi pour représenter les nombres à virgule en binaire), mais dans l'idée c'est ça.
Néanmoins, un ordinateur fonctionne (généralement) en représentant les nombres en utilisant un nombre de *bits* constant. En effet, c'est plus simple de ne pas avoir à se demander combien de *bit* fait le nombre avant de travailler avec et beaucoup plus efficace d'effectuer une opération sur deux nombres si on sait quelle taille ils font.
En fonction du processeur dont est fait votre ordinateur, la taille (le nombre de *bits*) allouée à la représentation des **nombres entiers** varie, mais est généralement de 64 *bits* (ou 32 *bits* pour certains processeurs plus vieux) et les circuits sont construits de manière à refléter cette taille, pour une raison assez pratique : ces nombres entiers servent également à représenter des adresses mémoires au niveau du processeur et une bonne partie d'un programme informatique revient à manipuler des adresses mémoires.
Lorsqu'on a un nombre N quelconque, il «suffit» de calculer $n = \lceil log_2 N \rceil$ [17] pour savoir combien de *bits* sont nécessaires pour le représenter (au minimum). Dès lors, lorsqu'on représente un entier sur n *bits* (autrement dit, à l'aide de n *bits*), on peut représenter en pratique les entiers de 0 à $2^n - 1$. Ça signifie que 2^n ou $2^n + 1$ ne possèdent pas de représentation si on utilise n

bits (ce serait équivalent à l'infini), on *sort de l'espace de représentation*. Et on va voir juste en dessous qu'on peut exploiter ce phénomène une fois qu'on en est conscient. 🍊

4.2. Addition, soustraction et nombres négatifs en binaire

Comme on va le voir par la suite, les opérations avec le binaire sont en fait équivalentes à ce que vous avez fait en primaire, mais dans une base différente.

4.2.1. L'addition

Ainsi, l'addition de deux nombres peut s'effectuer comme un *calcul écrit*[18]. Par exemple, sur 5 *bits*, $\{11+13\}_{10} = \{01011+01101\}_2$ se calcule comme suit.

$$\begin{array}{cccccc} & \color{blue}{1} & \color{blue}{1} & \color{blue}{1} & \color{blue}{1} & \\ & 0 & 1 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 & 0 \end{array}$$

En effet, lorsqu'on a $\{1+1\}_2$, on dépasse la base, donc on écrit 0_2 et on reporte 1_2 (les reports sont indiqués en bleu). Et ainsi de suite. Lorsqu'on se retrouve avec $\{1+1+1\}_2$, on écrit et reporte 1_2, tout simplement. On obtient au final 11000_2, ce qui correspond bien à 24_{10}.

4.2.2. Nombres négatifs

On pourrait tout simplement écrire -1011_2, comme on le fait classiquement, mais on souhaite une représentation qu'on puisse facilement exploiter. En particulier, on veut que l'addition de nombres dans cette représentation soit aussi simple que l'addition en binaire qu'on a vu plus haut (et qu'on ait donc pas à s'inquiéter du signe). Ce qui permettrait, si c'est le cas, de ne pas avoir à s'inquiéter d'écrire un circuit pour la soustraction, puisque $a-b$ se calcule comme $a+(-b)$.

4.2.2.1. Première solution : le bit de signe

Une solution, pourrait être d'utiliser un *bit de signe*, disons le premier (celui de poids fort), et le reste des *bits* seraient la représentation sur $n-1$ *bits* de la valeur absolue du nombre.
Ainsi, -7_{10} serait écrit, sur 5 *bits* (par exemple), $\underline{1}0111_2$, dont on saurait que le premier *bit* serait à 1 s'il s'agit d'un nombre négatif, 0 pour un nombre positif[19]. Notez que par facilité, je vais **souligner le premier *bit* quand on sera dans une suite de *bits* représentant un nombre négatif**, afin de les différentier d'une représentation non signée.
Le problème de cette représentation, c'est que 0 possède alors deux représentations, qui, sur 5 *bits*, seraient donc $\underline{0}0000_2$ et $\underline{1}0000_2$ (qui correspond à -0). Ce n'est pas quelque chose qui est forcément souhaitable (même si on pourrait s'en sortir). Par contre, cette représentation rend

17. Les parenthèses $\lceil\rceil$ signifient qu'il faut arrondir à l'entier supérieur, c'est l'équivalent de la fonction `ceil()` d'un grand nombre de langages de programmation.
18. Et d'ailleurs, ça fonctionne pour n'importe quelle base.

l'addition (ou la soustraction) de nombres signés relativement complexe à écrire, car l'addition de deux nombres signés de cette manière ne correspond pas à leur addition purement binaire. Par exemple, soit le calcul $\{3-7\}_{10}$, qui dans cette représentation (sur 5 *bits*) s'écrit $\{\underline{0}0011 + \underline{1}0111\}_2$ (on calcule en fait $\{3+(-7)\}_{10}$) :

$$\begin{array}{cccccc} & & 1 & 1 & 1 & \\ & 0 & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 1 & 0 \end{array}$$

Si on considère l'addition purement binaire, c'est correct (c'est comme si on calculait $\{3+23\}_{10} = 26_{10} = 11010_2$). Par contre, dans notre représentation, on a un nombre négatif (le premier *bit* est égal à 1_2), mais on obtient -10_{10}, ce qui n'est pas la bonne réponse. Utiliser un *bit* de signe n'est donc pas une bonne idée dans ce cas ci.

4.2.2.2. Le complément à 1

Qu'à cela ne tienne, on a en fait déjà vu la solution : $a + \bar{a} = 0$, ce qui est bien la propriété que doit respecter un nombre et son opposé négatif. Cette méthode s'appelle **le complément à 1** et consiste à représenter le nombre négatif correspondant en inversant tous les bits ($1 \leftrightarrow 0$). On constate que dans la pratique, les nombres négatifs ont alors leur *bit* de poids fort (le *bit* le plus à gauche) qui vaut 1_2, comme dans la technique du *bit* de signe.
Dès lors, si 7_{10} correspond à $\underline{0}0111_2$ sur 5 bits, alors -7_{10} correspond, dans cette représentation, à $\overline{\underline{0}0111}_2 = \underline{1}1000_2$ (tous les *bits* sont inversés et on voit que le *bit* de poids fort est égal à 1, comme avec le *bit* de signe, raison pour laquelle je vais continuer à le souligner). Et cette fois-ci, pour l'addition, ça fonctionne ! Si on reprend $\{3-7\}_{10}$, on a donc cette fois $\{\underline{0}0011 + \underline{1}1000\}_2$, ce qui donne

$$\begin{array}{cccccc} & 0 & 0 & 0 & 1 & 1 \\ + & 1 & 1 & 0 & 0 & 0 \\ \hline & 1 & 1 & 0 & 1 & 1 \end{array}$$

On obtient cette fois $\underline{1}1011_2$, ce qui équivaut bien à -4_{10}, puisque $\overline{4}_{10} = \overline{\underline{0}00100}_2 = \underline{1}1011_2$. On a donc une méthode qui permet de réaliser l'addition (et la soustraction) sans avoir à se soucier du signe (et, encore mieux, sans avoir à se soucier de savoir si les nombres qu'on manipule sont signés ou pas). Par contre, 0_{10} possède toujours deux représentations (sur 5 *bits*, $\underline{1}1111_2$ et $\underline{0}0000_2$), ce qui n'a l'air de rien, mais demande en fait de tester deux valeurs pour savoir si un résultat est nul ou pas.

4.2.2.3. La solution finale : le complément à 2

On utilise donc la technique du **complément à 2** : $-a = \bar{a} + 1$. Tous les avantages du complément à 1 sont conservés, mais en plus, pour zéro, ça fonctionne, parce que, toujours sur 5 *bits*, $0_{10} = \underline{0}0000_2$ et $-0_{10} = \{\overline{\underline{0}0000} + 1\}_2 = \{\underline{1}1111 + 1\}_2$, ce qui équivaut à...

19. C'est une convention, qui en vaut une autre. D'ailleurs peu importe.

$$\begin{array}{c|cccccc} & 1 & 1 & 1 & 1 & 1 & \\ & & 1 & 1 & 1 & 1 & 1 \\ + & & & & & & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & \end{array}$$

ce qui fonctionne, par ce qu'on travaille sur 5 *bits* (représenté par la barre verticale dans le calcul ci-dessus), on oublie donc le report sortant (report induit par l'addition des *bits* de poids fort, et qui « sort » des 5 *bits*), et on obtient $\underline{0}0000_2$, donc on a bien $0_{10} = -0_{10}$ dans le complément à 2. Pareil, ça fonctionne également dans le cas de $\{3-7\}_{10}$, sauf que cette fois-ci, $-7_{10} = \{\overline{\underline{0}0111}+1\}_2 = \underline{1}1001_2$, et

$$\begin{array}{cccccc} & & & 1 & 1 & \\ & 0 & 0 & 0 & 1 & 1 \\ + & 1 & 1 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 0 & 0 \end{array}$$

Ce qui est bien la représentation de -4_{10} en complément à 2, puisque $\{\overline{\underline{0}00100}+1\}_2 = \underline{1}1100_2$.

> *i* En fait, calculer la représentation de $-a$ comme le complément à deux de a revient à calculer $2^{n-1} - |a|$, où n est le nombre de *bits* qu'on se donne pour représenter les nombres et $|a|$ la valeur absolue de a. Autrement dit, si un nombre est écrit en complément à deux, alors le *bit* de poids fort correspond à -2^{n-1}, puis les autres puissances de 2 s'additionnent, comme vu précédemment. De ce fait, cette représentation permet de stocker des nombres allant de -2^{n-1} à $2^{n-1} - 1$.

Ainsi, si on reprend -4_{10} en complément à 2, c'est-à-dire $\underline{1}1100_2$, cela correspond en base 10 à

$$\begin{aligned} \underline{1}1100_2 &= \{-1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\}_{10} \\ &= \{-16 + 8 + 4\}_{10} \\ &= -4_{10}. \end{aligned}$$

Et on voit bien que les deux correspondent. On peut donc rapidement convertir un nombre négatif en base 10 selon la technique vue plus haut, en calculant $2^{n-1} - |a|$ et en représentant ce nombre dans les $n-1$ *bits* restants. Ainsi, pour représenter -72_{10} en complément à deux, sur 8 *bits*, on calcule $\{2^7 - 72\}_{10}$, ce qui fait 56_{10}, et on représente 56_{10} sur les 7 *bits* restants (par la technique de votre choix), donc 0111000_2. $-72_{10} = \underline{1}0111000_2$ sur 8 *bits* en complément à 2. Pour vous entraîner, vous pouvez convertir -33_{10}, -124_{10} et -1_{10} en complément à 2 sur 8 *bits*.

Contenu masqué n°11

4.2.3. « Il y en a un peu plus, je vous le mets ? » (*l'overflow*)

On est passé très vite (exprès) sur le fait que dans certain calculs (entre autre le résultat de -0_{10} en complément à 2), on sorte de l'espace de représentation (qu'on ait un report sortant

qui dépasse les n *bits* qu'on s'est donné pour représenter notre nombre) n'était pas un problème. Sauf que des fois, c'est bien entendu quelque chose qu'on ne souhaite pas, et ce problème arrive quand le résultat du calcul n'est pas représentable avec l'espace de représentation qu'ont s'est fixé, c'est ce qu'on appelle l'*overflow*.
Évidement, ça ne peut jamais arriver lorsque on additionne des nombres de signes différents, ou qu'on soustrait des nombres de même signe, puisque dans ces cas, le résultat est forcément représentable si les opérandes le sont. Il reste donc à s'intéresser aux autres cas possibles (opérandes de même signe pour l'addition, de signes opposés pour la soustraction).

a	b	***overflow* sur $a+b$?**	***overflow* sur $a-b$?**
>0	>0	*peut être*	NON
>0	<0	NON	*peut être*
<0	>0	NON	*peut être*
<0	<0	*peut-être*	NON

Voyons un peu quand ces cas aboutissent à un *overflow* : restons sur 5 *bits* et comparons les différents calculs suivants (la barre indique la limite des 5 *bits*).

	0	0	0	1	1
+	0	1	0	0	0
	0	1	0	1	1

				1	1
	1	1	1	0	1
+	1	1	0	0	0
1	1	0	1	0	1

	1				
	0	1	0	1	1
+	0	1	1	0	0
	1	0	1	1	1

1		1			
	1	0	1	0	1
+	1	0	1	0	0
1	0	1	0	0	1

— Le premier et le deuxième cas (correspondant respectivement à $\{3+8\}_{10}$ et $\{-3-8\}_{10}$) ne posent pas de problème : bien que l'on a un report sortant, les résultats sont corrects si on se limite aux 5 bits de notre représentation (ce qui est logique, puisque 11_{10} et -11_{10} sont tous les deux représentables sur 5 bits, puisque compris dans l'intervalle $[-16_{10}, 15_{10}]$).
— Les deux cas suivants (correspondants respectivement à $\{11+12\}_{10}$ et $\{-11-12\}_{10}$) sont eux problématiques (ce qui, encore une fois, est logique).

La clé, c'est de regarder s'il y a un report entrant généré par l'addition impliquant le *bit* de poids fort (le *bit* de signe) : si on a un report entrant et sortant ou pas de report du tout, il n'y a pas de problème et il suffit de lire les *bits* sans s'inquiéter. S'il y a seulement un des deux reports qui est présent, alors on est dans un cas d'*overflow*.
Ainsi, dans le cas de la deuxième addition, il n'y a pas de problèmes car il y a report entrant et sortant pour le *bit* de signe, ce qui n'est pas le cas pour la troisième (report entrant, mais pas de report sortant) ou la quatrième (uniquement un report sortant). Si on note r_s le report généré par le *bit* de signe et r_e le report entrant pour cette même addition de *bits*, alors,

r_e	r_s	***overflow***
0	0	0
1	0	1

0	1	1
1	0	0

Ce qui correspond à la fonction logique $r_e \cdot \overline{r_s} + \overline{r_e} \cdot r_s$, c'est à dire à un XOR ($r_e \oplus r_s$).

i

Il y a donc une différence à ce niveau-là entre les opérations sur entiers signés et non signés. En effet, dans le cas d'une opération sur nombres non-signés, l'absence de report entrant n'est pas un problème, seul la présence de report sortant l'est (puisque ça signifie qu'on sort de l'espace de représentation des entiers non-signés). Comme le processeur n'a pas moyen de savoir a priori s'il manipule des entiers signés ou pas (et que tout le but, c'est justement de ne pas trop à avoir à s'en inquiéter), les instructions processeurs correspondantes sont différentes pour les entiers signés ou pas.

4.2.4. La soustraction

On a donc vu que $a - b$ équivalait à $a + (-b)$ et qu'on pouvait représenter $-b$ en utilisant le complément à deux, ce qui, moyennant qu'il n'y ait pas d'*overflow*, fonctionne. On a vu que pour obtenir la représentation de $-b$ en complément à deux, on calculait $\bar{b} + 1$. Alors, pour que ça soit plus rapide, pourquoi ne pas calculer $a + \bar{b}$, en partant avec un report de 1_2 sur l'addition du *bit* de poids le plus faible ?
Prenons, par exemple, le calcul $\{15 - 6\}_{10}$ (pas de problème d'*overflow* à l'horizon). Pour rappel $\bar{6}_{10} = \overline{00110}_2 = \underline{1}1001_2$, et donc

1	1	1	1	1	1
	0	1	1	1	1
+	1	1	0	0	1
1	0	1	0	0	1

Le report entrant a été indiqué en rouge. On voit que

1. On a effectivement pas de problème d'*overflow*, comme promis, puisque report généré par les deux dernières additions ;
2. Que dès lors, si on se limite à 5 *bits*, on a bien la bonne réponse puisque $01001_2 = 9_{10}$.

i

Ce qui semble tenir de l'astuce ici est en fait beaucoup plus intelligent que ça : en remarquant ça, on se rend compte que pour calculer $a \pm b$, on a en fait besoin que d'un seul circuit logique réalisant l'addition, et qui prendrait en entrée b ou $\bar{b}$ et un report entrant ou pas pour avoir un circuit qui réalise l'addition et la soustraction sans trop avoir à se fatiguer (puisque sinon, il faudrait d'abord calculer $\bar{b} + 1$ avant de l'additionner à a). D'ailleurs, c'est ce qu'on va exploiter au prochain chapitre.

4.2.5. Bonus : ce qu'implique le complément à 2

Voici quelques effets, parfois surprenants, de l'utilisation de la représentation en complément à 2 pour les nombres négatifs :

1. **Il est facile d'étendre un nombre dans une représentation donnée à une autre représentation** : il suffit de recopier le *bit* de signe (celui que je souligne) autant de fois que nécessaire (c'est ce qu'on appelle **l'extension de signe**). Ainsi, sur 5 *bits*, $7_{10} = \underline{0}00111_2$, et $-7_{10} = \underline{1}1001_2$. *Eh* bien sur 8 *bits*, ces mêmes nombres sont $\underline{0}0000111_2$ et $\underline{1}1111001_2$, respectivement (l'extension de signe est indiquée en bleu).
2. **La comparaison d'entier non-signés et d'entier signés n'est plus là même**, puisque une comparaison lexicographique classerait alors les nombres négatifs avant les positifs. Néanmoins, en pratique, la comparaison d'entiers se fait en effectuant une soustraction : si le résultat est différent de 0 (ou qu'il y a eu *overflow*), alors les deux nombres ne sont pas les mêmes (le signe du résultat et la présence d'un *overflow* indique même si les deux nombres comparés sont plus grands ou plus petits l'un par rapport à l'autre). Et cette approche fonctionne que l'entier soit signé ou non.
3. **Le nombre le plus négatif**, -2^{n-1}, **est égal à son complément à 2** (de la même manière que le calcul de -0_{10} génère 0_{10}). Du coup, dans certains langages de programmation, la fonction `abs()` (ou équivalent), qui calcule la valeur absolue d'un nombre entier, peut avoir un comportement inattendu lorsqu'on calcule la valeur absolue de ce nombre en question.

> Contenu masqué n°12

4.3. Multiplication et divison binaire

Au détail du complément à deux, on a vu que les vieilles recettes de primaire fonctionnaient bien ici. *Eh* bien la bonne nouvelle, c'est que c'est toujours le cas pour les deux dernières opérations mathématiques !

4.3.1. La multiplication (*shift, add, shift, add*, ...)

Pour rappel, la multiplication de deux nombres, $\{a \times b\}_{\mathfrak{B}} = \{a_{n-1}a_{n-2}\dots a_0 \times b_{m-1}b_{m-2}\dots b_0\}_{\mathfrak{B}}$, quelles que soient leurs bases (disons donc $\mathfrak{B}$), se cacule comme $b_0 \times a + b_1 \times a \times \mathfrak{B} + b_2 \times a \times \mathfrak{B}^2 + \dots$, soit :

$$a \times b = \sum_{i=0}^{m-1} b_i \times a \times \mathfrak{B}^i$$

où m et n sont le nombre de chiffes dans b et a, respectivement. Maintenant, cette écriture mathématique « cache » le fait que multiplier un nombre par $\mathfrak{B}^i$, ce n'est jamais que décaler le nombre de i positions vers la gauche : $\{32 \times 10^3\}_{10} = 32000_{10}$, je ne vous apprends normalement rien. Et ce qui est vrai dans n'importe quelle base est vrai en binaire, dans lequel décaler les *bits* vers la droite multiplie le nombre par 2 (pour ce qui est des entiers non signés, en tout

20. Mais aussi au moins en C++ et en Java. J'ai pas testé pour les autres, et c'est pas réellement grave.

cas). Dès lors, la multiplication de nombres entiers non-signés revient à une série d'addition du multiplicande[21], de plus en plus décalé vers la droite. L'avantage du binaire, c'est que si le *bit* correspondant dans le multiplicateur est 0_2, pas besoin de réaliser l'addition.

$$a \times b = \sum_{i=0}^{m-1} \begin{cases} 0 & \text{si } b_i = 0, \\ a \ll i & \text{si } b_i = 1. \end{cases}$$

où $a \ll i$ représente ici a décalé de i positions vers la gauche[22].
Ainsi, $\{7 \times 13\}_{10} = \{00111 \times 01101\}_2$ (sur 5 *bits*, et en non-signé) équivaut à :

$$\begin{array}{cccccccccc}
 & & & & & 0 & 0 & 1 & 1 & 1 \\
 & & & & \times & 0 & 1 & 1 & 0 & 1 \\
\hline
 & & & & & 0 & 0 & 1 & 1 & 1 \\
 & & & 0 & 0 & 1 & 1 & 1 & & \\
+ & & 0 & 0 & 1 & 1 & 1 & & & \\
\hline
 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1
\end{array}$$

Le produit, $0001011011_2 = 91_{10}$ correspond bien au résultat de $\{7 \times 13\}_{10}$. Par contre, on voit que la multiplication de deux nombres constitués de n *bits* génère en fait un nombre composé de $2n$ *bits*, ce qui est relativement logique, et d'ailleurs les multiplications ne génèrent pas d'erreur d'*overflow* au niveau du processeur, dès lors stocké dans deux nombres de n *bits* chacun[23] (par contre, ce que le programme reçoit en retour est souvent la partie de n *bits* la plus à droite, mais ça dépend du langage de programmation).
Pour ce qui est de la multiplication d'entiers signés, le seul cas problématique (au delà de l'*overflow*) est le cas où le multiplicateur est négatif. En effet, si c'est le multiplicande qui est négatif, cela fonctionne quand même (pour peu qu'on ne sorte pas de la représentation). Pour preuve, $\{-3 \times 3\}_{10} = \{\underline{1}1101 \times \underline{0}0011\}_2$ (sur 5 *bits*, pour pas changer) vaut (la barre verticale indique la limite de 5 *bits*) :

$$\begin{array}{cccccc|ccccc}
 & & & & & & 1 & 1 & 1 & 0 & 1 \\
 & & & & & \times & 0 & 0 & 0 & 1 & 1 \\
\hline
 & & & & & & 1 & 1 & 1 & 0 & 1 \\
+ & & & & & 1 & 1 & 1 & 0 & 1 & \\
\hline
 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1
\end{array}$$

Et ça fonctionne (si on se limite aux 5 *bits* les plus à droite), puisque $\underline{1}0111_2 = -9_{10}$.
Dès lors, une solution lorsqu'on a un multiplicateur négatif, ben ... C'est de prendre le négatif du multiplicateur et du multiplicande[24]. Mathématiquement, ça se tient, puisque $a \times -b =$

21. Oui, moi aussi je l'ai découvert en écrivant ce tuto, si $a \times b$, a est le multiplicande (et oui, c'est un nom masculin) et b le multiplicateur.
22. Notation empruntée à différents langages de programmation, car il n'existe pas de symboles mathématiques pour ça, ou en tout cas pas que je connaisse.
23. Deux registres de n bits, quoi. Bien entendu, si on écrit un peu de code assembleur, il y a moyen de récupérer le second registre.

$(-1) \times a \times b = -a \times b$, mais on voit que le multiplicateur est alors positif. Et ça, on vient de voir que ça fonctionnait.
On peut donc calculer sans problème $\{5 \times -3\}_{10}$ comme $\{-5 \times 3\}_2 = \{\underline{1}1011 \times \underline{0}0011\}_2$ (la barre verticale indique encore et toujours la limite des 5 *bits*),

						1	1	0	1	1
					×	0	0	0	1	1
						1	1	0	1	1
					1	1	0	1	1	
+										
	0	0	0	0	1	1	0	0	0	1

Bien entendu, $\underline{1}0001_2 = -15_{10}$.

i

Pendant très longtemps, les processeurs ne possédaient pas de circuits logiques spécifiques[25] pour la multiplication. Quand on y réfléchit, on se rend compte que la manière vue ci-dessus est quand même très lente, puisque faire une multiplication de deux nombres de n *bits* requiert au final $2n$ additions (et, si on y réfléchit pas, un additionneur qui travaille avec $2n$ *bits*, mais il y a moyen de s'en passer). Heureusement, il existe des algorithmes beaucoup plus efficaces, réduisant le nombre d'addition à effectuer (la page Wikipédia de la multiplication donne quelques pistes à ce sujet). Néanmoins, le message reste vrai : une multiplication, c'est une série de décalages et d'additions.

4.3.2. La division entière (*shift, sub, shift, sub* ...) et le modulo

Pour commencer, rappelons que la division revient à une série de soustraction du dividende par le diviseur (autant de fois qu'il faut pour obtenir un reste qui soit plus petit que le diviseur). Bien entendu, ça, c'est la manière la plus inefficace de faire (imaginez le temps que ça prendrait de diviser par 2_{10} des très grands nombres).
Ici, le calcul écrit nous permet d'aller plus vite que ça. Si on calcule par exemple $\{23 \div 4\}_{10}$ en non-signé, $\{10111 \div (00)100\}_2$, on a

	1	0	1	1	1	1	0	0
					—	—	—	
−	1	0	0			1	0	1
—	—	—						
			1	1	1			
		−	1	0	0			
		—	—	—				
				1	1			

24. Une autre solution ça serait de faire une extension de signe des nombres de n *bits* sur $2n$ *bits*, mais déjà que ce qu'on voit là n'est pas hyper efficace, ça le serait encore moins.
25. Et pas d'instruction assembleur correspondante non plus.

Néanmoins, quand on écrit ça, même si c'est correct, on fait plein de simplifications dans notre tête. Ainsi, on décale le diviseur vers la droite petit à petit. Mais surtout, on ne fait pas la soustraction lorsque on voit bien que le reste issu de la soustraction précédente est plus petit que le diviseur, et on continue alors de décaler le diviseur vers la droite, tout en considérant le *bit* suivant du dividende. Une implémentation basique de la division consiste à effectuer cette différence, mais à ne pas conserver le reste si celui-ci est inférieur à 0. Les *bits* du quotient sont alors mis en fonction de si le reste est supérieur ou strictement inférieur à 0.
On constate que la division de deux nombres de $2n$ *bits* donne, tout au plus, un quotient de n *bits* et un reste de n *bits*. Le quotient est le résultat de la division entière tandis que le reste est le résultat du modulo du dividende et du quotient, ce qu'on écrit très souvent en programmation `reste = a % b`.

Une fois encore, ce n'est pas la manière la plus efficace de faire, et des algorithmes plus efficaces existent (voir à ce sujet la page Wikipédia sur les algorithmes de division (en)).

Les nombres flottants ont été écartés de ce tutoriel afin de ne pas complexifier les choses. En effet, la représentation de ces nombres en utilisant la virgule flottante est très intéressante, mais dépasse de très loin ce dont on a besoin ici. Si ça vous intéresse, je vois renvois à la page Wikipédia correspondante , mais aussi à ce tutoriel qui présente une représentation alternative.

Au-delà de la représentation binaire des nombres entiers (positifs et négatifs), on a vu qu'il suffisait d'un circuit réalisant l'addition (et la soustraction, mais l'utilisation du complément à deux rend ça compatible avec un circuit réalisant l'addition) pour réaliser les 4 (+1, le modulo) opérations mathématiques de bases. En combinant ce circuit avec l'une ou l'autre fonctionnalités supplémentaires (quelques opérations logiques), on peut réaliser ce qu'on appelle une ALU, ou *arithmetic-logic unit*. Et comme on a vu tous les outils pour en réaliser une version simple, c'est ce que je me propose de vous montrer dans le prochain et dernier chapitre.
On s'y voit de suite.
Et si vous souhaitez voir comment on peut exploiter le binaire de façon plus complète dans vos codes, vous pouvez également lire ce tutoriel de Lucas-84 .

Contenu masqué

Contenu masqué n°9

$$1085_9 = \{1 \times 9^3 + 8 \times 9 + 5\}_{10} = 806_{10}$$
$$11213_4 = \{1 \times 4^4 + 1 \times 4^3 + 2 \times 4^2 + 1 \times 4 + 3\}_{10} = 359_{10}$$
$$1F75_{16} = \{1 \times 16^3 + 15 \times 16^2 + 7 \times 16 + 5\} = 8053_{10}$$

Retourner au texte.

Contenu masqué n°10

Quotient	Reste
1256	3
179	4
25	4
3	3
0	

Donc $1256_{10} = 3443_7$

Quotient	Reste
645	1
322	0
161	1
80	0
40	0
20	0
10	0
5	1
2	0
1	1
0	

Donc, $645_{10} = 1010000101_2$ Retourner au texte.

Contenu masqué n°11

— $\{2^7 - 33\}_{10} = 95_{10}$, or $95_{10} = 1011111_2$ (sur 7 bits), donc $-33_{10} = \underline{1}1011111_2$;
— $\{2^7 - 124\}_{10} = 4_{10}$, or $4_{10} = 0000100_2$ (sur 7 bits), donc $-124_{10} = \underline{1}0000100_2$;
— $\{2^7 - 1\}_{10} = 127_{10}$, or $127_{10} = 1111111_2$ (sur 7 bits), donc $-1_{10} = \underline{1}1111111_2$.

Retourner au texte.

Contenu masqué n°12

Le dernier point est par exemple vrai en C[20], où on utilise `INT_MIN` pour représenter le nombre le plus négatif représentable dans le type `int` (entiers), et où `INT_MIN == -INT_MIN`. Dès lors, on peut avoir ce genre de comportement.

```c
// petit code à compiler par exemple avec "gcc -o test main.c"

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main() {
    printf("abs(%d) = %d\n", -16, abs(-16));
    // donne "abs(-16) = 16"

    printf("abs(%d) = %d\n", 8, abs(8));
    // donne "abs(8) = 8"

    printf("abs(%d) = %d\n", INT_MIN, abs(INT_MIN));
    // donne, chez moi, "abs(-2147483648) = -2147483648",
    // mais ça dépendra de la valeur de INT_MIN chez vous.
}
```

Ce «*bug*» provient de l'implémentation de `abs()` en C, qui pourrait ressembler à ceci.

```c
/* Ceci est une implémentation naïve et très peu efficace. */
int abs(int n) {
  if (n >= 0)
    return n;
  else
    return -n; // ici, -a != |a| si a = INT_MIN
}
```

Comme ça, vous savez ! 🤘

Retourner au texte.

5. Vers la pratique : un gros paquet de portes

Armé de toutes ces connaissances, il est finalement temps de terminer notre voyage avec un petit aperçu de quelques circuits présents dans les processeurs modernes.

5.1. Multiplexeurs et démultiplexeurs

5.1.1. Multiplexeur (MUX)

Premier élément d'électronique utile pour la suite, le *multiplexeur* : plusieurs entrées, un sélecteur et une sortie. En fonction de la valeur du sélecteur, c'est une des entrées qui apparaît en sortie. Ainsi, on numérote les entrées et on donne au sélecteur la valeur de l'entrée qu'on veut voir en sortie.
Par exemple, prenons un multiplexeur simple, dit 2 vers 1 : on a deux entrées, e_0 et e_1 (pouvant chacune valoir 0 ou 1), un sélecteur, s_0 (valant 0 pour la sélection de e_0, 1 pour e_1) et une sortie q (dont la valeur dépend de l'entrée sélectionnée et de la valeur de cette entrée).

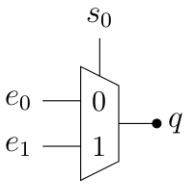


FIGURE 5.1. – Représentation schématique d'un multiplexeur (noté MUX).

À quoi ressemble le circuit électrique correspondant à cet élément ? Pour le savoir, vous pouvez constater qu'on a ici 3 entrées et une sortie. Il s'agit donc d'une fonction booléenne de type $q(e_0, e_1, s_0)$. Le reste, c'est la création d'une table de vérité, puis la simplification par Karnaugh, comme on l'a vu ici . Vous essayez ?

Contenu masqué n°13

Évidement, il existe des multiplexeurs 4 vers 1, 8 vers 1, 4 vers 2, etc. Néanmoins, le nombre de *bits* de sélection augmente avec le nombre d'entrée (4 entrées nécessitent 2 *bits* de sélections, 8 entrées nécessitent 3 *bits* de sélection, et ainsi de suite). Bien que ce soit ensuite un peu moins marrant à traiter avec Karnaugh, il est tout à fait possible d'arriver à des simplifications (en utilisant directement les règles de l'algèbre booléenne). Une autre manière de faire est de *chaîner* les multiplexeurs. Voici un exemple de multiplexeur 4 vers 1 réalisé à partir de multiplexeurs 2 vers 1.

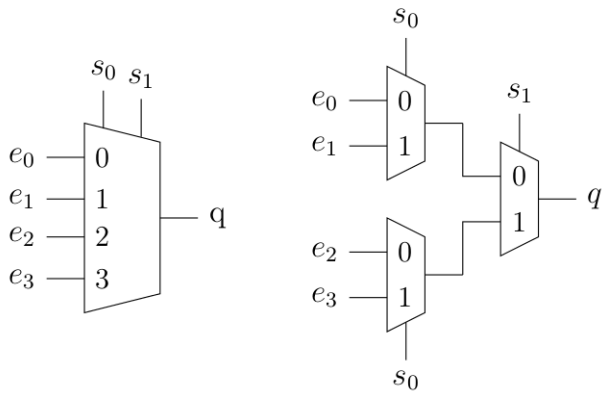


FIGURE 5.2. – Représentation schématique d'un MUX 4 vers 1 utilisant 2 *bits* de sélection, s_0 et s_1 (gauche) et sa décomposition (droite), réalisé à partir de 3 MUX 2 vers 1, ce qui correspond à la fonction logique $q(e_0, e_1, e_2, e_3, s_0, s_1) = \overline{s_1}\,(\overline{s_0} \cdot e_0 + s_0 \cdot e_1) + s_1\,(\overline{s_0} \cdot e_2 + s_0 \cdot e_3)$.

i

Ces composants sont très utiles pour piloter un composant électronique. En effet, comme on le verra ci-dessous, il permet de choisir entre différentes entrées en fonction de l'opération à réaliser.

5.1.2. Démultiplexeur (DEMUX)

À l'inverse, le démultiplexeur prend une entrée et la redirige vers la bonne sortie en fonction du sélecteur, ce qui donne l'équivalent électronique de l'aiguillage.

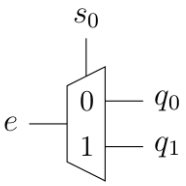


FIGURE 5.3. – Représentation schématique d'un démultiplexeur 1 vers 2 (DEMUX).

Cette fois, la fonction logique sera une série de « ET ». En effet, pour être active, une sortie donnée devra correspondre au sélecteur. Par exemple, dans le démultiplexeur donné plus haut, la fonction correspondante à la sortie « 0 » serait $q_0 = \overline{s_0} \cdot e$ tandis que $q_1 = s_0 \cdot e$. Dès lors, à quoi ressemblera le circuit logique ?

Contenu masqué n°14

Bien entendu, il existe des démultiplexeurs 1 vers 4, 8...

i

Couplés aux multiplexeurs, ces composés permettent de transmettre différentes informations par le même canal.

FIGURE 5.4. – Transmission de plusieurs informations, ici des conversations téléphoniques, par le même canal (source : Wikipédia)

5.2. Unité arithmétique et logique (ALU)

Bon, maintenant qu'on a vu les (dé)multiplexeur, on peut s'attaquer à un composant de base d'un processeur, c'est l'ALU, pour *arithmetic logic unit* (unité arithmétique et logique, en bon français). Dans un processeur, c'est ce type de circuit qui réalise toutes les opérations de base.

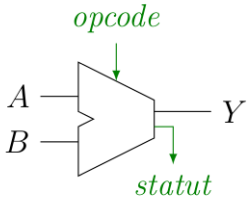


FIGURE 5.5. – Représentation schématique d'une ALU. Les flèches vertes représentent des *bits* de contrôles, qu'on aura l'occasion d'expliquer par la suite.

Une ALU prend un certain nombre de choses en entrée : deux *mots*, représentant des entiers et dont le nombre de *bit* dépend du processeur (32 ou 64 *bits*), A et B, ainsi que l'opération à réaliser par l'ALU, l'*opcode* (pour *operation code*). On retrouve alors le résultat de l'opération en sortie, Y (mot d'autant de *bits* que les entrées), ainsi qu'un certain nombre d'informations, regroupées sous le terme de *statut* (par exemple, pour citer quelque chose qu'on a déjà rencontré, l'*overflow*, mais aussi d'autres choses intéressantes).
En fonction de l'ALU, on peut réaliser un certain nombre d'opérations sur les entrées A et B, telles que :

- Les mathématiques de base : addition, soustraction, complément à 2, éventuellement les multiplications et divisions, ...
- Les opérations logiques : « ET », « OU », ...
- Les comparaisons : $A \neq B$, $A < B$, ...

D'autres opérations sont parfois disponibles dans des ALU spécialisées, telles que le calcul flottant (nombres à virgule), les décalages et rotations de *bits*, des opérations mathématiques plus complexes (racines carrées, exponentielle, logarithme et autres joyeusetés), mais le principe reste le même. Évidement, plus on rajoute d'opérations, plus les circuits associés sont complexes, le but ne va donc pas être ici de les passer en revue. Néanmoins, on va voir deux ou trois morceaux intéressants.

5.2.1. L'additionneur binaire

On avait fini le chapitre précédent en montrant qu'on pouvait implémenter les 5 opérations mathématiques de base à l'aide de l'addition et de quelques astuces. Il nous faut donc un circuit

permettant de réaliser l'addition de 2 *bits* en position i de A et B, qu'on va noter a_i et b_i, mais en gérant le report entrant, noté c_i, et pour stocker le tout dans le *bit* i de Y, noté y_i, et en calculant s'il est nécessaire d'avoir un report sortant, c_{i+1}, pour la prochaine opération. Autrement dit, on est face à la table de vérité ci-dessous.

c_i	a_i	b_i	$y_i(a_i, b_i, c_i)$	$c_{i+1}(a_i, b_i, c_i)$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Il n'y a plus qu'à regarder quelles fonctions logiques on obtient alors.

— Pour ce qui est du résultat de l'addition, $y_i(a_i, b_i, c_i) = \overline{c_i}\cdot\overline{a_i}\cdot b_i + \overline{c_i}\cdot a_i\cdot\overline{b_i} + c_i\cdot\overline{a_i}\cdot\overline{b_i} + c_i\cdot a_i\cdot b_i$, ce qui n'est pas simplifiable en utilisant Karnaugh (parce que ça donne un magnifique damier). Néanmoins, si on fait intervenir la porte XOR, $p \oplus q = \bar{p}\cdot q + p\cdot\bar{q}$, on peut obtenir une simplification en termes de nombre de portes logiques. Ainsi, cela revient à $y_i(a_i, b_i, c_i) = c_i \oplus (a_i \oplus b_i)$. Vous pouvez même le vérifier par vous-mêmes[7]. 🍹

— Pour le report sortant, on obtient cette fois quelque chose de simplifiable par Karnaugh. Une fois que c'est fait, on se rend compte que $c_{i+1}(a_i, b_i, c_i) = a_i \cdot b_i + c_i\,(a_i + b_i)$ (on gagne une porte en mettant c_i en évidence).

Ne reste plus qu'à faire le circuit correspondant, ce que je vous laisse essayer :

> 👁 Contenu masqué n°15

Pour la suite, on va simplifier les choses en représentant l'additionneur binaire comme suit :

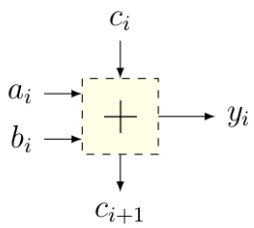


FIGURE 5.6. – Représentation simplifiée de l'additionneur binaire, équivalent du circuit présenté ci-dessus. Les flèches sont là pour visualiser plus facilement ce qui rentre et ce qui sort.

7. C'est tout à fait équivalent, puisque $c_i \oplus (a_i \oplus b_i) = c_i\,\overline{(a_i \oplus b_i)} + \overline{c_i}\,(a_i \oplus b_i)$, avec $\overline{(a_i \oplus b_i)} = \overline{(\overline{a_i}\cdot b_i + a_i\cdot\overline{b_i})} = \overline{(\overline{a_i}\cdot b_i)}\,\overline{(a_i\cdot\overline{b_i})} = (a_i + \overline{b_i})\,(\overline{a_i} + b_i) = a_i\cdot b_i + \overline{a_i}\cdot\overline{b_i}$. Il faut juste aimer De Morgan.

5.2.2. Une petite ALU 1 *bit*

On entre clairement dans le domaine du spéculatif : peu de chance qu'un vrai circuit de processeur ressemble à ceux que je vais vous présenter. Mais conceptuellement, ça tient la route.

Jusqu'ici, on peut juste réaliser l'addition. Qu'à cela ne tienne, on a vu que pour réaliser la soustraction, $A - B$, il suffisait de placer c_0 à 1, et d'additionner avec $\overline{b_i}$. On a donc besoin d'un signal, qu'on va appeller inv_B, qui donne b_i si $inv_B = 0$ et $\overline{b_i}$ si $inv_B = 1$ autrement dit, $inv_B \oplus b_i$. Et le circuit permettant de réaliser la soustraction est donc simplement le suivant.

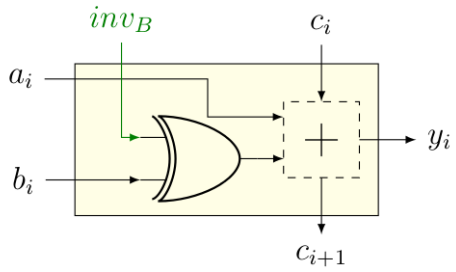


FIGURE 5.7. – ALU 1 *bit* (carré jaune) permettant de réaliser l'addition et la soustraction.

Bon, on aimerait aussi que notre ALU face un peu plus que l'addition et la soustraction, par exemple qu'elle face aussi le «ET» et le «OU». Néanmoins, pour faire ça, on va avoir besoin d'un multiplexeur qui réalise le choix entre les différentes opérations à réaliser par l'ALU, qu'on va commander par le signal *op*. Tant qu'on est dedans, on va rajouter un signal inv_A qui va permettre d'obtenir «NON ET» (comme $\overline{a_i} + \overline{b_i}$) et «NON OU» (comme $\overline{a_i} \cdot \overline{b_i}$). Et l'histoire de préparer la suite, on va également rajouter une entrée p_i, dont vous allez comprendre l'intérêt un peu plus bas.

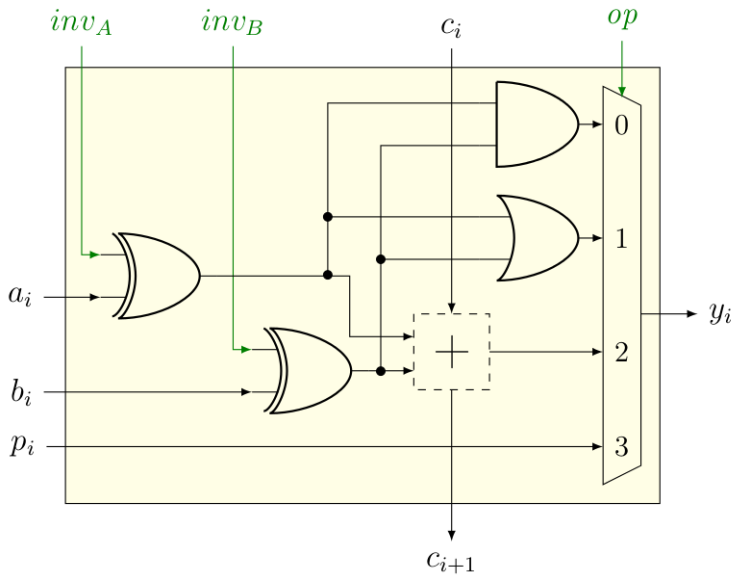


FIGURE 5.8. – Une version beaucoup plus intéressante de l'ALU 1 *bit*, qui permet déjà de faire plus de choses. 🍹 Évidement, on pourrait lui ajouter d'autres fonctionnalités (comme une porte XOR, par exemple), mais on peut calculer déjà pas mal avec ça. Notez que les valeurs de *op* sont purement arbitraires.

Pour la suite, on va encore dézoomer d'un cran et représenter ce type d'ALU de la manière suivante.

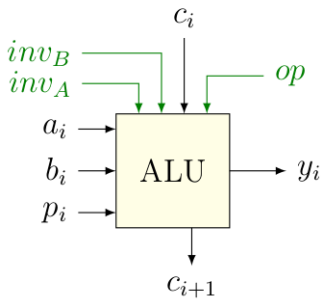


FIGURE 5.9. – Représentation schématique d'une ALU 1 *bit* capable de faire l'addition, la soustraction, et quelques opérations logiques.

5.2.3. Une « petite » ALU n *bits*

Maitenant qu'on a une ALU 1 *bit*, on peut l'utiliser pour faire des calculs sur des nombres représentés sur plusieurs *bits*. Combien ? Comme ça a été dit dans le chapitre précédent, ça dépend du processeur, mais généralement 32 ou 64 *bits*, qui est la taille de la représentation des nombres entiers. De manière assez logique[26], une telle ALU sera alors formée de n ALU 1 *bit* mises bout à bout pour traiter chacun des n *bits* de A et de B. Dès lors, on peut créer notre ALU comme suit.

26. Aha.

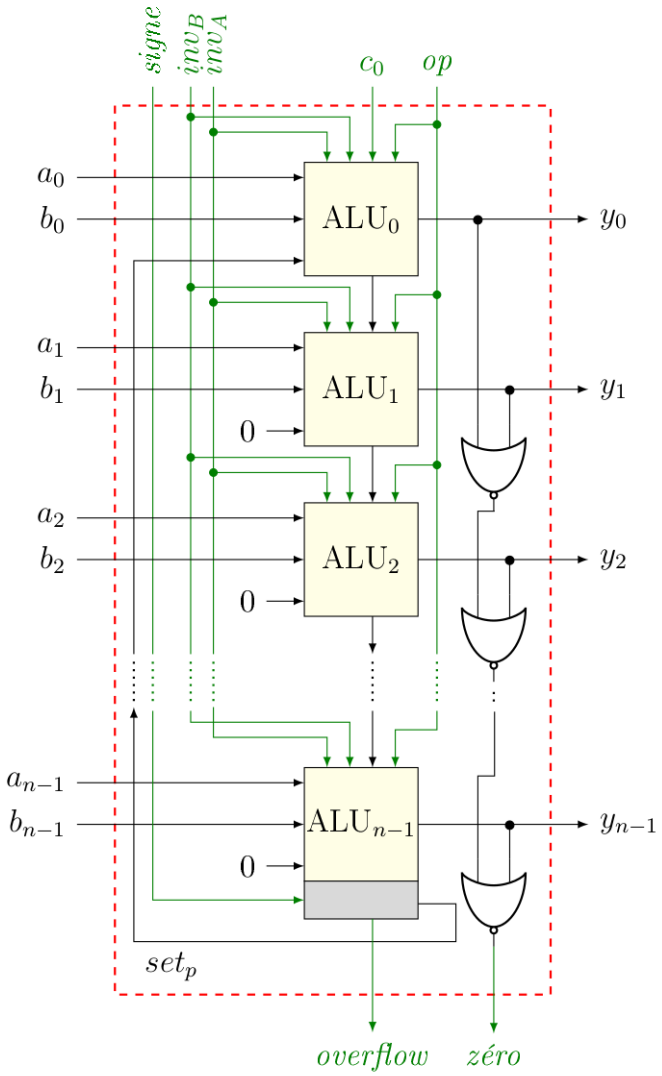


FIGURE 5.10. – Et voici notre ALU n *bits*. Et là je vous dois quelques explications.

Donc, qu'est-ce qu'on a là ? Vous devez reconnaître le centre : c'est là que se font les différentes opérations, *bit* par *bit*, comme promis. Pour que ça fonctionne, il faut évidement donner à chacune de ces ALUs les différents signaux *op*, inv_A et inv_B.
Par contre, deux modifications ont été apportées afin que cette ALU soit capable de faire les comparaisons :

— La première, c'est pour réaliser la comparaison $A = B$. Pour ça, l'astuce est de calculer $A - B$, puisque si A est égal à B, le résultat sera de... zéro. Ensuite, pour vérifier que c'est bien le cas, on fait passer le résultat de chaque soustraction *bit* à *bit* dans des portes « NON OU « (à droite dans le schéma). En effet, $x \downarrow y = 1 \Leftrightarrow a = b = 0$ (voir le premier chapitre). Le résultat, c'est le signal *zéro*, qui vaut donc 1 si $A - B = 0$, c'est-à-dire que $A = B$.

— La deuxième, basée sur la même astuce, c'est pour réaliser les comparaisons du type $A < B$. En effet, si $A < B$, $A - B$ est un nombre négatif. À l'inverse si $A > B$, $A - B$ est un nombre positif. Or on a vu que si un nombre était négatif, son *bit* en première position (le *bit* de poids fort) valait 1. C'est à ça que va servir la fameuse sortie p qu'on a rajoutée plus haut. Si cette sortie est sélectionnée, tous les *bits* seront mis à zéro, sauf le premier, dont la valeur sera égale au *bit* de poids fort résultant de la soustraction de A

et B (set_p dans le schéma), ce qui explique en partie pourquoi l'ALU$_{\text{n-1}}$ est différente des autres (la partie «grise» contient quelques circuits supplémentaires pour gérer cela[27]). Autrement dit, si $A < B$, $Y = 00\ldots001_2$, $00\ldots000_2$ sinon. Dans certains processeurs, c'est également un signal, comme *zéro*. À noter que le calcul dépend de si on compare des nombres signés ou pas (d'où le signal supplémentaire *signe*).

Et finalement, la dernière modification apportée est là pour permettre la détection de l'*overflow*, comme expliqué dans le chapitre précédent. Pour cela, il est nécessaire de connaître le report entrant et sortant (c_{n-2} et c_{n-1}), ainsi que le fait qu'on travaille avec des nombres signés ou non. De même, c'est l'ALU$_{\text{n-1}}$ qui a été modifiée de manière à indiquer s'il y a dépassement ou non.

Et on a donc une ALU qui est capable de faire des opérations sur les nombres entiers signés ou pas, y compris des comparaisons. C'est rudimentaire, mais vous n'imaginez pas tout ce qu'on peut déjà faire avec ça.

Tout ça n'est toutefois pas très efficace. En effet, tout ça est très joli sur le papier, mais j'ai volontairement fait en sorte de ne pas complexifier la chose en omettant des aspects très importants qui reposent sur la petite note faite il y a deux chapitres : **passer par une porte prend du temps** ! Deux conséquences à cela :

— On a créé des **scénarios de course** (*race condition*, que les développeurs parmi vous connaissent peut-être déjà, parce que ce n'est pas limité à l'électronique). Par exemple :

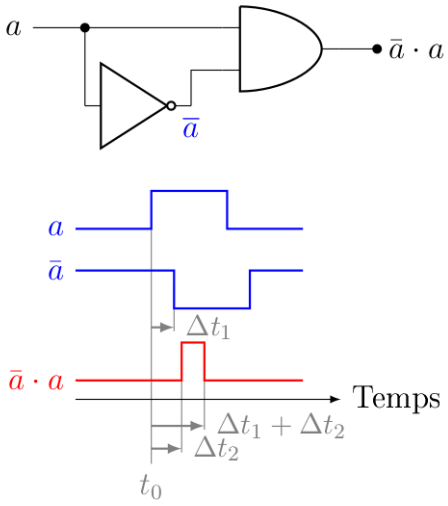


FIGURE 5.11. – Scénario de course simple entre deux composants électroniques (inspiré de Wikipédia). On verra plus loin que ce scénario a ceci dit un intérêt.

Imaginons qu'on change la valeur de a. Le passage par la porte «NON» implique un délai Δt_1 de temps, par rapport à la première entrée de la porte «ET», ce qui signifie que la valeur de sortie de cette porte «ET» n'est correcte qu'après un temps équivalent à $\Delta t_1 + \Delta t_2$ (le passage par la porte «ET» impliquant un second délai Δt_2). Autrement dit, comme les changements ne sont pas instantanés, il faut attendre un certain temps pour que le système se «stabilise» avant de lire une valeur correcte. Pire, durant l'intervalle, la valeur en sortie de la porte «ET» est incorrecte. Il faut donc attendre un certain temps... Pour ceux que ça intéresse, ça se traite généralement en ajoutant de la redondance et/ou en simplifiant le circuit (comme c'est le cas ici).

— La création d'un circuit comme celui présenté plus haut où les unités sont mises à la chaîne induit **un énorme problème généré par le calcul du report**. En effet, le calcul du

27. Je ne suis pas certain de la forme des circuits impliqués dans la partie grise de l'ALU$_{\text{n-1}}$, je ne préfère donc pas m'avancer.

dernier reste (c_n) requiert de passer par rien de moins que $3n$ portes, sans compter le temps qu'il faut pour réaliser les opérations à la suite : le calcul de y_i requiert le calcul de c_i, qui requiert le calcul de c_{i-1}, qui requiert celui de c_{i-2}, qui... rends le temps très long ! Ça impacte donc le nombre d'opération qu'on peut réaliser par seconde, et ce n'est pas quelque chose de souhaitable. Il existe cependant des solutions, par exemple le *carry lookahead adder* (« additionneur à retenue/report anticipée »), une alternative assez intelligente pour régler le problème du report et qui est une fois encore une application assez jolie de l'algèbre de Boole.

5.2.4. Bonus : et ainsi, on en arrive presque à l'assembleur

Si on résume, on a besoin de différents *bits* de contrôle nous permettant de réaliser tout un tas d'opérations. Et c'est là qu'intervient l'Assembleur. C'est quoi ? C'est le langage de «programmation» le plus proche du pour communiquer avec le processeur. Ainsi, les instructions en Assembleur sont directement lues et interprétées par le CPU, qui va agir en fonction. Une instruction Assembleur contient un *opcode* (qui dit au CPU quoi faire) et des *opérandes* (qui dit au CPU quoi utiliser pour le faire). Évidement, cet *opcode* n'est pas un mot, mais un nombre en binaire, qui peut alors être passé à la moulinette de portes logiques pour déterminer de quelle opération il s'agit. Ainsi, dans un CPU, il ne serait pas illogique de retrouver sous une forme ou une autre dans cet *opcode* les différentes valeurs de $signe$, inv_A, inv_B et op.
Le processeur trouve les données qu'il manipule dans des registres : il s'agit de zones mémoires très petites (typiquement, 1 mot) et à accès très rapide (à l'échelle du processeur). Quand on manipule une variable dans un langage de programmation, elle est en pratique copiée dans un registre avant d'être manipulée. Il existe, en fonction de l'architecture du processeur, un nombre de registre différent, mais ce nombre est en fait très faible comparé aux nombres de variables que peut manipuler un programme.

i **Par exemple, le MIPS32**

Prenons par exemple le MIPS32 (en) (un certain type de processeur, aujourd'hui plus scolaire qu'encore utilisé), on voit que les instructions processeurs sont séparées en différents «champs».

Instruction MIPS (32 bits)					
Opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	func (6 bits)

TABLE 5.3. – Répartition des différents *bits* dans une instruction MIPS (de type R). Pour les opérations mathématiques, l'opcode est le même. Le *function code* (func) contient les différents *bits* de contrôle pour l'ALU. rs, rt et rd (registres *source*, *target* et *destination*) sont les registres que l'instruction doit manipuler (en MIPS, il y a 32 registres, d'où le code sur 5 *bits* qui représente le numéro du registre). Le *shift amount* (shamt) est utile pour les opérations de décalage de *bits*. Typiquement, pour l'instruction **add** et **sub** (qui font, comme leur nom l'indique, l'addition et la soustraction de rs et rt pour inscrire le résultat dans rd), l'opcode vaut 0_{16} dans les deux cas, tandis que le *function code* vaut 20_{16} (32_{10}) et 22_{16} (34_{10}), respectivement (pour les additions et soustractions non-signées, les *function codes* valent 21_{16} et 23_{16}, de manière logique).
Plus d'exemples ici (en)

Le monde de l'Assembleur nécessiterait cependant un tutoriel à lui tout seul. En effet, les instructions sont spécifiques à un type de processeur (ARM ou intel, par exemple) et certaines subtilités existent (par exemple au niveau des registres). Néanmoins, le principe de base reste le même.

5.3. La mémoire (et l'horloge)

C'est quoi de la mémoire, en fait ? D'un point de vue tout à fait théorique, ce n'est jamais qu'une grosse fonction booléenne, pour laquelle la variable d'entrée est l'adresse, et la sortie est la valeur du *bit* situé à l'adresse demandée[28] (c'est le concept table de correspondance, *look-up table* (en) , abrégé LUT).
Évidement, vous connaissez des éléments qui constituent « de la mémoire » pour un ordinateur. De tête, on pourrait probablement citer les CD/DVD/Blue Ray, les clés USB et autres cartes flash, les disques durs et SSD, ou encore la fameuse RAM (*Random-Access Memory*). Là-dedans, on peut quand même distinguer :

— La mémoire *volatile* et non-volatile. La mémoire volatile disparaît en l'absence d'alimentation électrique (c'est le cas de la RAM), la mémoire non-volatile existe indépendamment de l'alimentation (c'est le cas de tout les autres composants cités).
— La mémoire à accès séquentiels ou aléatoire (*random*). Une mémoire à accès aléatoire permet, en un temps rapide, d'accéder à un endroit précis, là ou une mémoire à accès séquentiels... Le permet, mais en un temps plus long. Un exemple typique du dernier cas sont les bandes magnétiques (oui, les cassettes audio, mais ça a longtemps servi comme support de sauvegarde pour les ordinateurs aussi, des programmes étaient même vendus comme ça). On se rend également compte que le terme «RAM» n'est absolument pas précis, puisqu'on pense aux barrettes mémoires de notre ordi, alors que le terme s'applique en fait très bien à un disque dur (qui permet aussi d'accéder à un endroit arbitraire en un temps court). Le terme exact pour les barrettes de RAM est en fait DRAM (*dynamic random-access memory*) (voir de la SDRAM si on veut pinailler sur les détails), comme on aura l'occasion de le voir plus loin.
— La manière de stocker les *bits* varie également d'un support à un autre. En ce qui concerne les disques optiques (45 tours, CD, DVD), il s'agit bien entendu de creux et de bosses. Le disque dur exploite lui les propriétés magnétiques de métaux (c'était également le cas des disquettes et de la bande magnétique, qui se base sur les propriétés magnétiques de l'oxyde de fer, plus communément appelé la rouille, d'où la couleur des bandes magnétiques

d'ailleurs). Quant aux autres types de mémoire, on aura l'occasion d'en parler.

5.3.1. ROM ?

Avant de parler de mémoire volatile, passons un instant sur les ROM (*read only memory*). On leur donne aussi le petit nom de mémoire morte en français. Il s'agit simplement d'un circuit implémentant la fameuse fonction booléenne qui, pour une adresse entrée (dans les *word lines*), donne la valeur des *bits* en sortie (dans les *bit lines*), les unes et les autres étant reliés par ce qu'il faut de porte logique pour obtenir la fonction booléenne. Dans une version naïve, ces fonctions peuvent être implémentées simplement à l'aide de multiplexeurs, mais ça en ferait vite beaucoup. Une autre idée, c'est d'être plus systématique et en accord avec ce qu'on a vu dans le chapitre sur la logique booléenne : une fonction booléenne, c'est une disjonction de *minterms* (qui sont des conjonctions de littéraux, pour rappel). Du coup, on peut schématiser les choses comme suit.

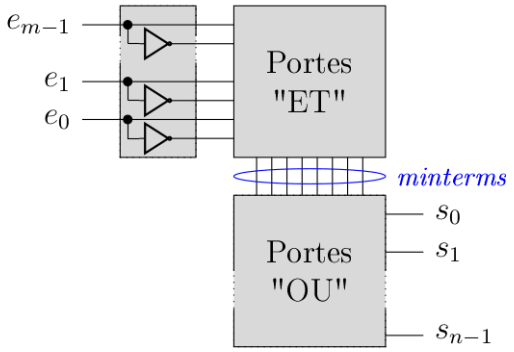


FIGURE 5.12. – Schéma simplifié d'une ROM, composé de m entrées et de n sorties (qu'on appelle une $2^m \times n$ *bit* ROM, donc si par exemple $m = n = 2$, alors ce sera une ROM 4 x 2 *bits*).

Écrire une ROM demande donc de créer un circuit logique (c'est là que l'algorithme de Mc-Cluskey prend tout son sens, afin de demander à un ordinateur de faire le travail de simplification). Pour rendre ça un peu plus systématique, on a inventé des ROM «programmables» (les PROMS). L'idée de base, c'est un *array* de portes «ET», suivis d'un *array* de portes «OU», et par défaut, tout est interconnecté, mais la programmation de la ROM permet de changer ça. Voyons ça sur un exemple d'une ROM 4x2 bits :

28. En vrai, on travaille plutôt au niveau de l'octet, voir du mot (64 *bits* sur les processeurs modernes). Mais c'est pareil.

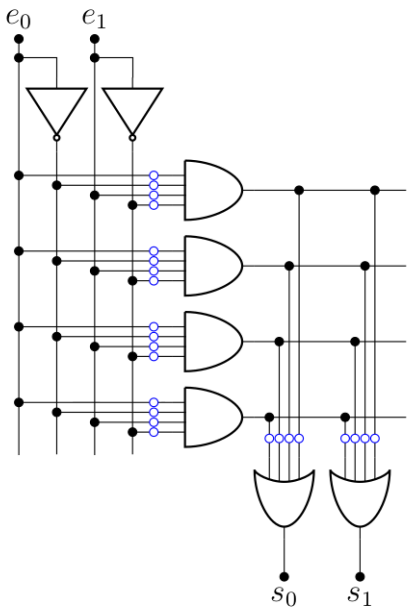


FIGURE 5.13. – Schéma d'une ROM 4x2 bits non-programmée. Il s'agit plus précisément d'un *programmable logic array* (en) (PLA). Notez qu'une PLA pour une ROM avec m entrées et n sorties contient 2^m portes « ET » (ou équivalent) et n portes « OU ».

Chaque petit rond bleu est une connexion qui peut être brisée (pour ça, on peut utiliser des fusibles que l'application d'une grande tension brise). C'est le processus de "programmation" d'une ROM : passer d'un circuit comme celui présenté plus haut à un circuit correspondant à la fonction logique qu'on souhaite implémenter. En l'occurrence, pour implémenter la fonction logique $f(a,b) = a \cdot b + \bar{a} \cdot b$ et $g(a,b) = \bar{a} \cdot \bar{b} + a \cdot \bar{b}$,

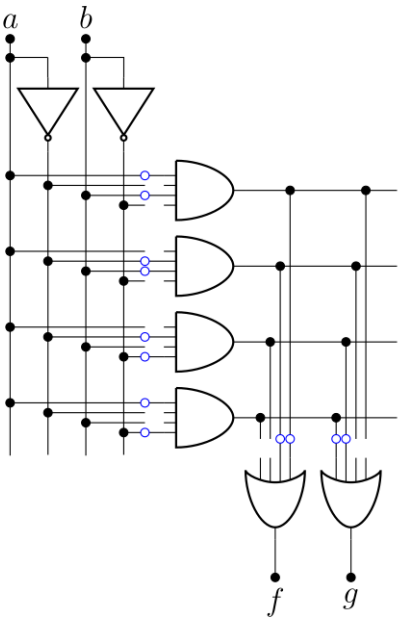


FIGURE 5.14. – ROM programmée pour que les sorties correspondent aux fonctions booléennes f et g.

On trouve généralement un schéma simplifié pour exprimer la même chose.

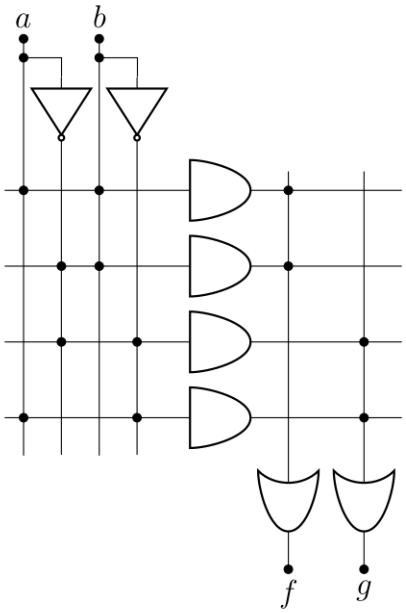


FIGURE 5.15. – Schéma simplifié de la PLA pour f et g.

Dans le cas d'une simple PROM (avec des fusibles), la programmation ne peut être effectuée qu'une fois (au moment de sa création). Mais on a par la suite inventé les EPROM et EEPROM, qui peuvent être reprogrammées après une exposition aux rayons UV pour la première et l'aide d'un courant électrique pour la seconde (ce qui est utile lorsqu'on fait du développement de circuit).
Les ROM étaient par exemple présentes dans les anciens BIOS ou était employée dans les cartouches de jeux vidéos auparavant[29] (en anglais, le mot pour les désigner est *ROM cartridge*, *cartridge* signifiant cartouche). On s'en sert toujours pour créer des tables de correspondances entre valeurs ou dans l'informatique embarquée.
Mais plus intéressant, les mémoires flashs et disque dur SSD sont en fait composés de ROM reprogrammables à l'aide de courant électriques.

5.3.2. Circuit avec rétroaction et logique combinatoire

Que vaut $r(a)$ dans le circuit suivant ?

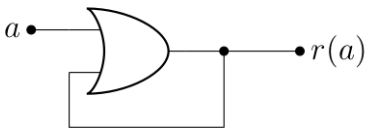


FIGURE 5.16. – Circuit avec rétroaction : la sortie de la porte «OU» est passée en entrée de cette même porte.

29. En parlant de ça, puisque la ROM est non-modifiable, elle ne permet pas la sauvegarde, à l'inverse des mémoires volatiles qu'on verra plus loin. C'est pourquoi il y avait une pile dans les cartouches de Game Boy (et que la plupart ne fonctionnent plus aujourd'hui à moins de changer la pile).

Eh bien... Ça dépend. Ça dépend en fait de ce que vaut a au temps t, bien entendu, ce qu'on va noter a_t, mais ça dépend également ce que valait a avant, ce qu'on va noter a_{t-1}. En fait, on peut écrire le petit tableau suivant.

a_{t-1}	a_t	$r(a_t)$
0	0	0
1	0	1
0	1	1
1	1	1

TABLE 5.5. – Table de vérité du circuit ci-dessus.

En fait, le plus intéressant, c'est que si $a_t = 0$, alors le circuit ne change pas de résultat ($r(0) = a_{t-1}$). À l'inverse, si $a = 1$, alors le circuit change son état pour 1 ($r(1) = 1$), et rien ne pourra plus l'en faire sortir (sauf évidement de couper l'alimentation électrique). Le circuit est donc verrouillé (on parle d'un *latch*, en anglais).

Et ça, c'est très intéressant : d'une part parce qu'on vient de créer de la mémoire volatile (qui, ici, n'est pas réinscriptible, puisque si on inscrit 1, on ne peut plus en sortir), et d'autre par parce qu'on vient de mettre le doigt sur ce qu'on appelle de la logique **séquentielle**. En effet, jusqu'ici, on est resté dans le domaine de la logique **combinatoire**, ou la valeur d'une fonction booléenne dépend uniquement de ses entrées. Sauf que dans l'exemple ci-dessus, on a vu qu'il fallait également introduire la notion de temps, puisque la sortie de la fonction booléenne ne dépend plus seulement des valeurs d'entrée, mais également des valeurs précédentes (ou, comme on le verra plus loin, de l'instant où elles sont mesurées).

Bon, jusqu'ici, on a un circuit qui n'est pas très utile, mais on peut l'améliorer pour qu'il le soit en créant des « verroux » . L'un des exemples les plus utilisés, c'est celui qu'on peut créer avec deux portes NOR assemblées comme suit.

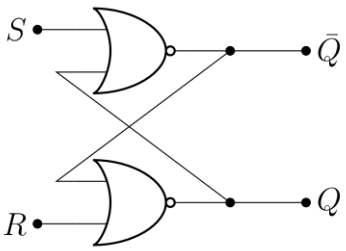


FIGURE 5.17. – Verrou SR. Vu que transistors et CMOS agissent comme des portes NOR, c'est assez facile à mettre en place.

La table de vérité de ce circuit est la suivante.

S	R	Q_t	**Note**
0	0	Q_{t-1}	Lecture de la valeur de Q
0	1	0	Mise à 0 de Q
1	0	1	Mise à 1 de Q
1	1	X	Interdit

TABLE 5.7. – Table de vérité du verrou SR (*SR latch* en anglais).

Là, on a un élément mémoire digne de ce nom. On voit que si on met S à 1 (avec $R = 0$), on change la valeur de Q telle qu'elle soit 1 (et ce quelle que soit sa valeur précédente) : $\bar{Q} = S \downarrow Q_{t-1} = 0$ **et ce quel que soit la valeur précédente de** Q_{t-1} (allez revoir la table de vérité de NOR si vous ne me croyez plus). Le reste suit logiquement : $R \downarrow \bar{Q} = 1$, donc on arrive bien à la situation où $Q = 1$.

À l'inverse, R à 1 (avec $S = 0$) change la valeur de Q pour 0. S et R sont simplement les premières lettres de *set* et *reset*. La valeur de Q est lue en utilisant $R = S = 0$, c'est l'état verrouillé, comme dans le circuit précédent. À noter que $R = S = 1$ est interdit, parce que dans ce cas, on a droit à un magnifique scénario de course (enfin, grosso modo, $Q = \bar{Q} = 0$, ce qui n'a pas beaucoup de sens).

i

Ce circuit est nommé multivibrateur bistable , c'est-à-dire un circuit dont la sortie change d'état lors de la modification de l'état d'une de ces entrées, et qui garde cette position jusqu'à ce qu'on change les entrées. Le terme «multivibrateur» vient du fait que ce circuit peut également être vu comme un oscillateur électrique, produisant un signal (idéalement) carré.

Ce verrou est dit **transparent**, dans le sens où un changement des entrées se répercute immédiatement sur la sortie. Et ça, ça peut être un problème à cause des scénarios de courses déjà mentionnés plus haut. Idéalement, on souhaiterait que le changement ne soit possible qu'à certains moments, on va donc ajouter une entrée qui permettra d'activer le changement, qu'on va noter E (pour *enable*). Facile, il suffit d'ajouter des portes «ET» :

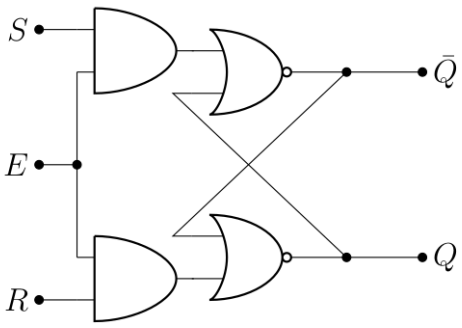


FIGURE 5.18. – Verrou SR modifié pour ajouter une protection contre la modification (*gated SR latch* en anglais).

E	S	R	Q_t	**Note**
0	X	X	Q_{t-1}	Pas de changements
1	0	0	Q_{t-1}	Lecture de la valeur de Q
1	0	1	0	Mise à 0 de Q
1	1	0	1	Mise à 1 de Q
1	1	1	X	Interdit

TABLE 5.9. – Table de vérité du verrou SR avec protection.

Sauf qu'il y a une simplification possible ! En effet, lorsqu'on change l'état du verrou (avec $E = 1$), alors S et R sont opposés. Il suffit alors tout simplement de retirer l'une des deux entrées (on va renommer l'autre D pour *data*), et d'utiliser une porte « NON » :

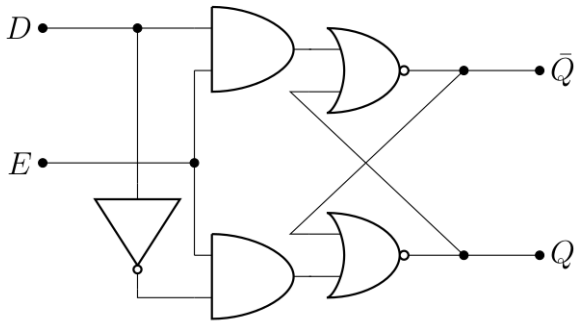


FIGURE 5.19. – Verrou D (*D latch* en anglais).

Ce qui simplifie la table de vérité :

E	D	Q_t	**Note**
0	X	Q_{t-1}	Pas de changements
1	0	0	Mise à 0 de Q
1	1	1	Mise à 1 de Q

TABLE 5.11. – Table de vérité du verrou D. Cette fois, le verrou est transparent lorsque $E = 1$, opaque dans le cas contraire. Notez qu'il n'y a pas de cas interdit.

5.3.3. L'horloge et les *flip-flops*

5.3.3.1. La gestion du temps

L'horloge, c'est tout d'abord un circuit capable de passer d'un état bas (0) à un état haut (1) et inversement f fois par seconde, c'est ce qu'on appelle la fréquence. La période de l'horloge est donnée par $T = 1/f$, c'est le temps qu'il faut au circuit pour partir de 0 et revenir à 0 (même chose avec 1). Avec un dessin,

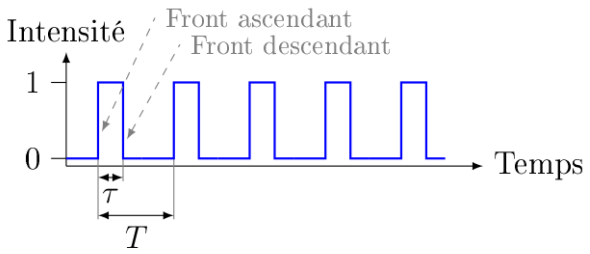


FIGURE 5.20. – Signal périodique d'horloge. Le temps τ est le temps entre les deux côtés du signal positif, le front ascendant (*rising edge*, qu'on appelle aussi front montant) et le front descendant (*falling edge*, en anglais).

Derrière tout ça se cache un matériau piézo-électrique, c'est-à-dire un matériau qui possède la capacité de changer de forme si on lui applique un potentiel électrique (c'est un cristal de quartz qui est employé ici, et si on connaît sa forme, on peut connaître assez précisément la fréquence).
On peut également introduire le rapport cyclique : c'est la fraction de la période durant laquelle le signal est haut. Ce rapport se calcule simplement comme $1\alpha = \tau/T$. Il est généralement de $^1/_2$. L'idée, c'est que c'est ce signal qui (en plus de donner l'heure) va synchroniser les circuits électriques. Autrement dit, on va s'arranger pour que les changements n'arrivent qu'à un instant précis (tout les T intervalles de temps, évidement), puis on va laisser le temps au circuit pour se stabiliser, et on est assuré qu'après un certain temps (inférieur à T, ce qui fait que le choix de la période est crucial et dépend du plus lent des composants du circuit, rappelez-vous la remarque sur le calcul du report fait plus haut), la réponse qu'on mesurera(it) sera(it) la bonne.
Une fois qu'on possède un tel circuit d'horloge, on peut s'en servir pour synchroniser tous les circuits. Évidement, chacun de ceux-ci ne travaille pas à la même fréquence (T plus ou moins long), mais on peut s'arranger (la mémoire vive fonctionne par exemple plus lentement que le processeur, et c'est sans parler des mémoires plus lentes telles que les disques [durs]). Notez que la modification de la fréquence d'horloge d'un processeur est tout à fait possible, on peut ainsi l'augmenter par *overclocking* (sur-cadençage pour les plus français d'entre nous), mais le processeur peut également faire le travail lui-même, souvent en réduisant la fréquence pour réduire la quantité de chaleur mais aussi en l'augmentant en cas de besoin (avec des technologies comme Intel Turbo Boost (en) et AMD PowerTune (en)).
Donc la première chose à tenter, c'est de brancher le signal E de notre verrou D au signal d'horloge. Quand on fait ça, on a alors un système dont les valeurs ne peuvent être modifiées que quand le signal d'horloge est haut la valeur associée au signal est de 1). On passe alors d'un verrou à une bascule (*flip-flop* en anglais, rien à voir avec les «chaussures» du même nom), qui se distingue d'un verrou par le fait que le circuit correspondant est **synchrone**.

5.3.3.2. *Edge-triggered flip-flops*

C'est une bonne idée, mais... puisque durant tout cet intervalle de temps τ, la valeur du verrou peut être modifiée, on est pas à l'abri d'un cas où la valeur de D changerait plusieurs fois au cours de cet intervalle (à moins d'avoir un τ extrêmement faible). Du coup, on va ruser et permettre le changement uniquement quand le signal d'horloge change. Et pour ça, on peut imaginer plusieurs manières de faire. La première est de réaliser un circuit qui ne vaut 1 que quand le signal d'horloge change de valeur. Et pour ça, il suffit d'exploiter... Le scénario de course qu'on a vu plus haut, tout simplement. Imaginons que le temps pour passer au travers d'une porte «NON» soit de Δt_1 tandis que le passage dans une porte «ET» soit de Δt_2. On peut alors mesurer la réponse du circuit suivant au cours du temps :

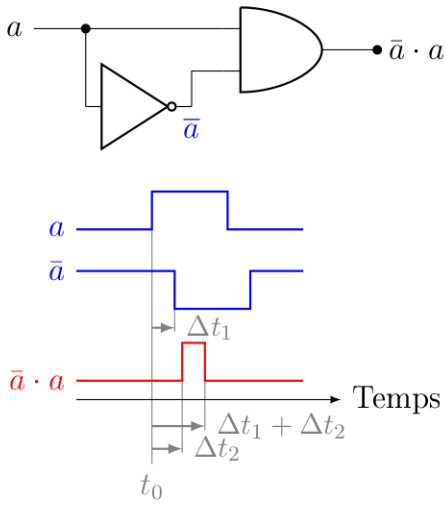


FIGURE 5.21. – Le retour du scénario de course :magicien : (notez que les échelles sont un peu exagérées).

On voit que le signal de sortie (en rouge) ne vaut 1 que durant un bref instant Δt_1 qui suis le front ascendant de a (on voit qu'il n'y a pas de pic correspondant au front descendant de a). On a donc une espèce de détecteur de pulsation, donc il «suffit» de minimiser Δt_2 et de contrôler Δt_1 pour obtenir quelque chose d'utilisable pour contrôler précisément quand la valeur de la bascule peut être modifiée. Notez que si on veut réaliser un circuit qui détecte le front descendant, il suffit d'utiliser une porte «NOR» à la place du «ET». On peut donc placer un tel circuit avant l'entrée E d'un verrou D pour obtenir un système tout à fait décent.
Une autre manière de faire, c'est d'assembler en série deux verrous D, contrôlés par le même signal d'horloge, dans une configuration dite «maitre-esclave» :

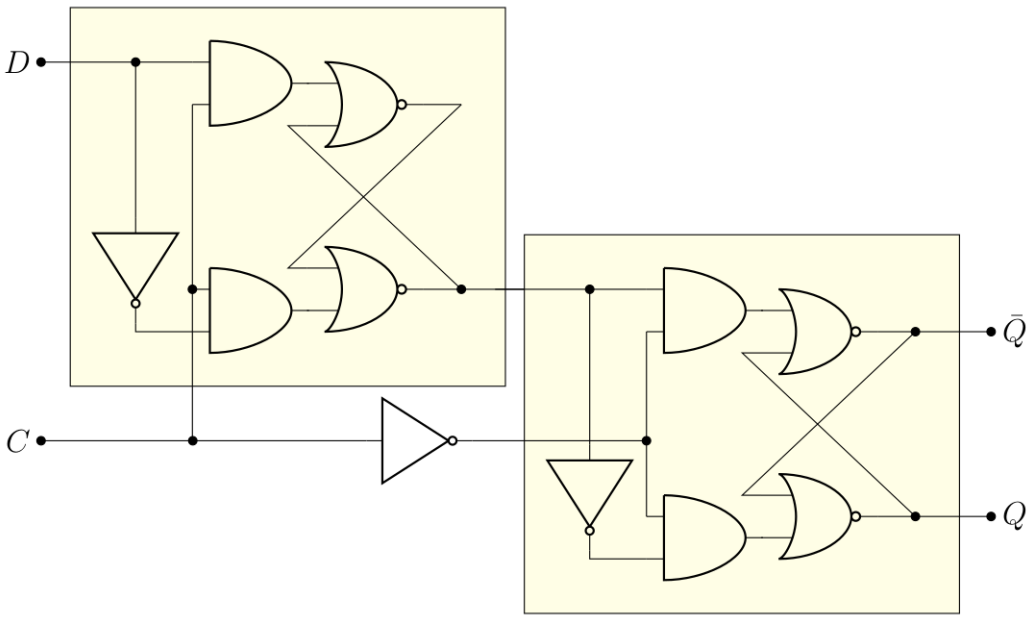


FIGURE 5.22. – Chaine de deux verrous D (maître puis esclave) formant une bascule D maître-esclave se déclanchant sur le front descendant (*Master–slave negative edge-triggered D flip-flop*). L'entrée C est l'horloge (*Clock*).

Déjà, on peut noter que le second verrou (l'esclave) ne change de valeur que si le premier (le maître) change. On peut également noter que ce changement ne peut pas être simultané, puisque les entrées permettant la modification (E) sont inversée sur les deux verrous. Autrement dit, on ne peut éditer la valeur du maître que si le signal d'horloge vaut 1 (mais la lecture de Q donnera toujours la valeur précédente), et une fois que le signal d'horloge passe à 0, l'esclave est mis à jour en conséquence (mais le maître ne peut plus être modifié). Autrement dit, ce système change de valeur sur le front descendant du signal d'horloge. Bien entendu, il y a moyen d'obtenir le même genre de circuit pour le front ascendant (en fait, il suffit de mettre une porte «NON» avant le signal d'entrée de l'horloge), mais l'idée reste la même.
Il existe bien entendu un symbole simplifié pour ces composants, que voici :

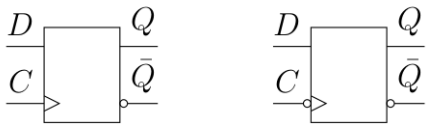


FIGURE 5.23. – Symbole pour les bascules D maître-esclave se déclenchant sur le front ascendant (gauche, dit positif) ou descendant (droite, dit négatif).

Notez qu'en pratique, c'est ce genre de bascules qui sont généralement utilisées (configuration maître-esclave). Il en existe également d'autre, telles que la bascule JK (deux verroux SR mit en série, donc un fonctionnement proche de la version D) et la bascule T, qui change de valeur de sortie (*toogle*, d'où le T) si son entrée $T = 1$ (la bascule a pour équation $Q_t = T \cdot \bar{Q}_{t-1} + \bar{T} \cdot Q_{t-1}$). Ce qui est drôle, c'est que si l'entrée T est constamment maintenue à 1, alors le signal varie toutes les 2 périodes d'horloges (donc on divise la fréquence d'horloge par deux).
Plus d'info sur les bascules sur Wikipédia .

5.3.4. SRAM et DRAM

5.3.4.1. Registres et caches : SRAM

En fait, c'est comme pour l'ALU : le plus dur, c'est de faire le travail pour 1 *bit*, ensuite il suffit d'en mettre plusieurs d'affilée. Les registres n'y échappent pas :

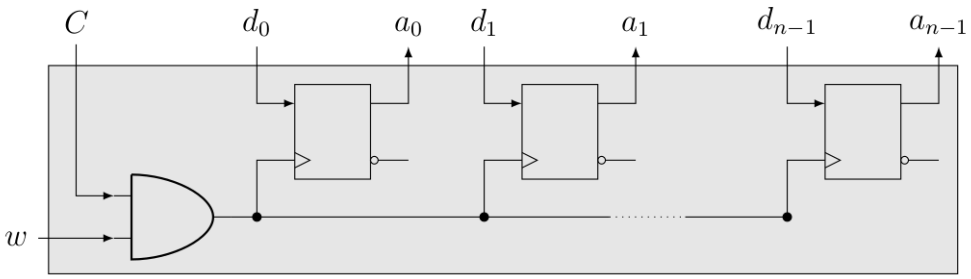


FIGURE 5.24. – Registre de n *bits*, composés de n bascules D. Le signal w sert à autoriser l'écriture, tandis que le signal C est l'horloge (du processeur). Notez que ce circuit permet de lire et d'écrire en même temps.

On vient de créer ce qu'on appelle de la *static random-access memory*, ou SRAM (même si en pratique, ce n'est plus des bascules D qui sont utilisées). Celle-ci est plus rapide que la DRAM qu'on verra juste après, bien qu'elle prenne plus de place que cette dernière (et est plus coûteuse). C'est pour ça qu'on la retrouve principalement dans les registres (pour taper sur le clou : zones de mémoires directement manipulées par le processeur), ainsi que les mémoires caches (qui ne sont pas manipulées directement par le processeur, mais contiennent des données qui sont souvent manipulées par celui-ci, ce qui accélère le calcul parce que les données sont plus rapidement accessibles que s'il fallait aller les rechercher en mémoire vive, par exemple).

5.3.4.2. Bon, et la (S)DRAM, alors ?

Rien à voir. Une des causes du coût de la SRAM, c'est le nombre de portes logiques (et donc de transistors) qu'il est nécessaire pour les créer (entre 4 et 6). *Eh* bien la DRAM demande deux composant pour stocker un *bit* : un transistor (nMOS) et un condensateur.

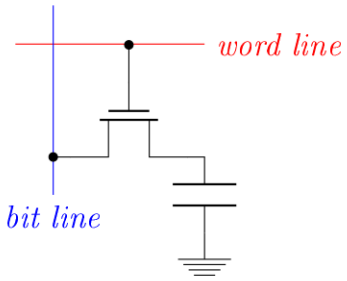


FIGURE 5.25. – Schéma d'une cellule de DRAM, permettant de stocker un *bit*. Pour rappel, la *word line* est celle qui provient des entrées (l'adresse) et la *bit line* symbolise la sortie.

Pour rappel, un transistor nMOS ne laisse passer le courant que si un potentiel est appliqué sur sa grille (c'est comme un interrupteur). Par ailleurs, on retrouve un condensateur, dont on avait déjà expliqué le fonctionnement il y a deux chapitres, mais disons simplement qu'il permet de stocker de l'électricité, comme une micro-batterie. Bref, le concept, c'est que le condensateur stocke la présence ou l'absence de *bit* par présence ou absence d'électricité. À partir de là, c'est facile : la charge du condensateur n'est modifiée que quand un potentiel haut (équivalent à 1) provient de la *word line* (ça signifie que la cellule a été choisie). Pour écrire dans la cellule DRAM, c'est le potentiel de la *bit line* qui va déterminer si le condensateur va être chargé (potentiel haut) ou déchargé (potentiel bas). Pour la lecture, c'est la décharge ou nom du condensateur qui va modifier le potentiel de la *bit line* (qui est préalablement placé à un potentiel intermédiaire entre haut et bas) et qui va permettre de savoir si le condensateur était chargé (1) ou déchargé (0).
Quelques petites choses à préciser, tout de même :

1. Ça signifie que quoiqu'il arrive, à la fin d'une opération de lecture, l'état du condensateur est inversé. Donc, il est nécessaire de faire suivre une opération de lecture par «une opération d'écriture» (recharger ou décharger le condensateur pour le replacer dans son état initial) !

2. Le problème des condensateurs, c'est qu'ils ne sont pas des batteries très efficaces et qu'ils laissent du coup «échapper» le courant très rapidement. Toutes les 64 ms (ou moins), une lecture est faite afin de réécrire les données à la suite et ainsi préserver l'état de la mémoire. C'est de ce besoin périodique de rafraîchissement que provient le «D» de DRAM (*dynamic*), là où la SRAM n'a pas besoin d'être rafraîchie.
3. On ne travaille pas par cellules individuelles, mais par «rangées» de cellules. Donc on lit par rangée, on rafraîchit l'état de toutes les cellules de la rangée en même temps et on profite d'une écriture pour également rafraîchir l'état de toute la rangée (pour être sûr). À cause de tout ça, la DRAM possède son propre circuit de gestion, indépendant du processeur et sa propre horloge (qui va normalement moins vite que celle du processeur). C'est la présence de cette horloge qui fait la SDRAM (pour *synchroneous*). On ne trouve aujourd'hui quasi plus que des SDRAM.
4. Le terme "DDR " signifie "*double data rate*" et s'applique uniquement à un certain type de SDRAM. Pour réaliser ce miracle, ces mémoires transfèrent des données non seulement sur le front montant, mais aussi sur le front descendant de l'horloge (évidement, l'électronique est adaptée en fonction). Les itérations suivantes (DDR2 , DDR3 et DDR4) jouent sur le nombre de données qui peuvent être transférées simultanément, l'augmentation de la fréquence d'horloge et la réduction du voltage (qui entraîne sinon des surchauffes à cause de l'effet joule).

À noter que la gestion de la mémoire par un ordinateur est un sujet crucial pour les performances, d'une part au niveau du cache (pour que les données les plus utiles soient rapidement accessibles), d'autre par parce que la demande en mémoire d'un programme excède souvent la mémoire disponible (ce qui n'est pas un problème si on le gère convenablement, avec ce qu'on appelle la mémoire virtuelle , où on va mettre l'excédent moins utile sur le disque). Mais tout ça sort du cadre de ce tutoriel.

Et voilà ! Il s'agissait d'un aperçu assez général et succin, et beaucoup plus pourrait être dit sur le sujet. Pour ceux que le sujet aurait intéressé, je leur conseille de faire un tour sur ce textbook (en) , qui est, en plus d'être vachement complet, accessible gratuitement en ligne. Toute la section matériel et électronique se propose également à vous, dont l'incontournable tutoriel Arduino .
Sinon, après avoir été si bas dans les détails de l'électronique d'un ordinateur, il est éventuellement temps de regarder du côté du langage assembleur, qui vous permettrait d'apprendre comment un processeur (et un programmeur) réussi à exploiter tout ça. C'est également très intéressant, et il n'est pas impossible que je rajoute un jour une seconde partie à ce tutoriel pour expliquer ça.
D'ici là, bon amusement

J'espère que vous avez appris plein de choses, et que vous avez ressenti qu'il suffit en fait d'outils très simples et d'un peu d'ingéniosité pour faire des choses très complexes, telles que le support que vous êtes en train d'utiliser pour me lire.
Dans tous les cas, je vous remercie de m'avoir suivi. Merci également à @Taurre pour la validation, ainsi que tous les gens qui m'ont fait part de retours lors de la bêta d'une manière ou d'une autre : @Aabu, @d3m0t3p, @**Ksass'Peuk**, @unidan et @Vayel.
Sources et notes :

— Le livre *computer organization and design* de M. Patterson et J.L. Hennessy, qui, en plus de réexpliquer tout ce que je fais dans ce tutoriel, va plus loin et tente d'expliquer comment fonctionne vraiment un ordinateur (incluant le célèbre processeur MIPS32).

Merci donc à mon prof de m'avoir indiqué ce bouquin. Nombre des schémas du dernier chapitres en sont directement inspirés.
— Wikipédia, surtout dans sa version anglaise, reste une source inépuisable de savoir, même s'il est parfois nécessaire de jongler entre les onglets. En particulier, ça m'a bien aidé pour écrire toute la partie liée aux semi-conducteurs (en) , aux transistor bipolaires (en) mais aussi les CMOS (en) , et donc les MOSFETs (en) .
— Pour l'algorithme de Quine-Mc Cluskey, outre la page Wikipédia (en) , je me suis également aidé de cette présentation (en) et de celle-ci (en) . La méthode de Petrick est en outre expliquée ici (mais également dans les deux documents précédents).
— De manière générale, ce *textbook* (en) de « *All About Circuits* » est une mine d'information géniale. Si le sujet vous intéresse, foncez.
— Par ailleurs, cette présentation de F. Anceau, qui m'a été indiquée par Aabu lors de la bêta , retrace très bien l'historique du développement des transistors, et mentionne deux ou trois choses que je n'ai pas eu le temps et la place de développer.
— En l'absence de mieux, les schémas électriques ont été réalisés à l'aide de circuitikz (en) (ça fonctionne vraiment pas mal, en fait !), les tables de Karnaugh sont issues d'une réécriture en Tikz des fonctionnalités de ce *package* (en) , tandis que l'afficheur 7 segments est une réécriture des fonctionnalités de ce *package* (mais avec des segments un peu plus réalistes et modulables à mon gout). L'ensemble de ces schémas est distribué sous la même licence que le tutoriel sur un dépôt Github dédié .

À bientôt ! 🤠

Contenu masqué

Contenu masqué n°13

s_0	e_0	e_1	q
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0

Vous devriez assez facilement tomber sur $q(e_0, e_1, s_0) = \overline{s_0} \cdot e_0 + s_0 \cdot e_1$, ce qui est assez logique. Le circuit associé est donc :

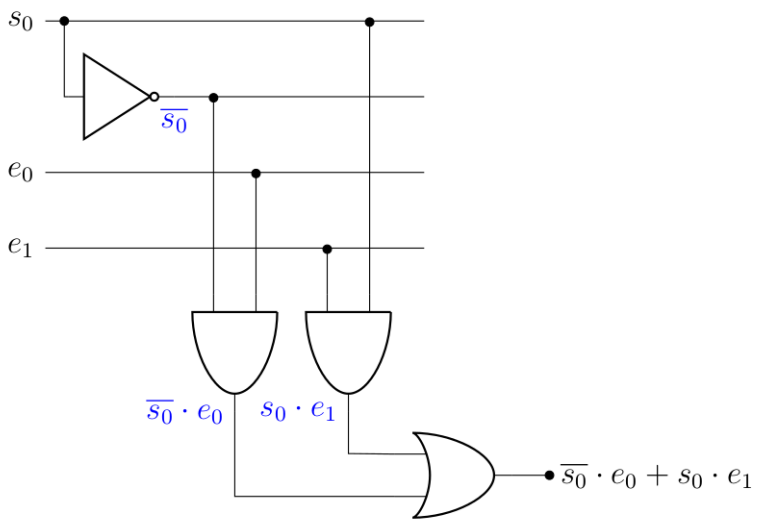


FIGURE 5.26. – Circuit d'un MUX 2 vers 1.

Retourner au texte.

Contenu masqué n°14

Sachant cela :

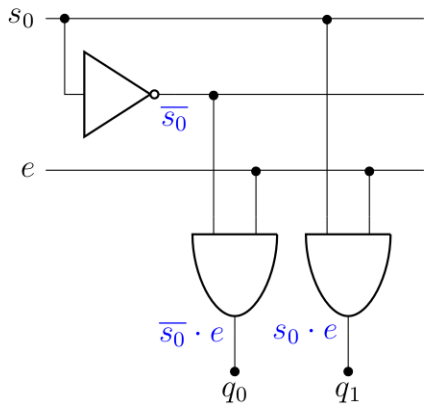


FIGURE 5.27. – Schéma électrique d'un DEMUX 1 vers 2.

Retourner au texte.

Contenu masqué n°15

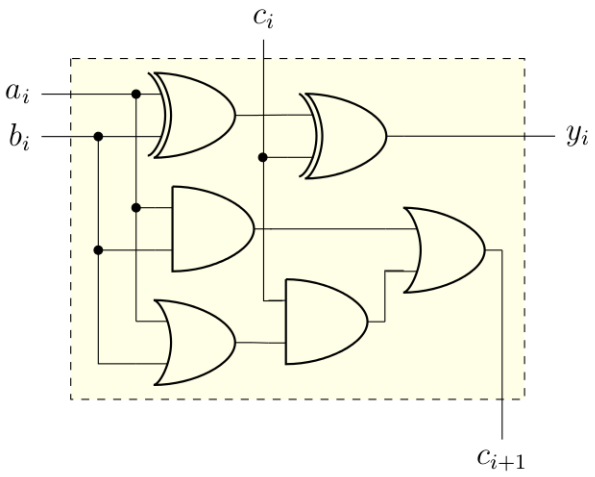


FIGURE 5.28. – Circuit électronique d'un additionneur binaire. Les deux portes XOR s'occupent de l'addition, tandis que les portes du dessous gèrent le report sortant. C'est un peu plus compressé que d'habitude, mais peu importe.

Retourner au texte.