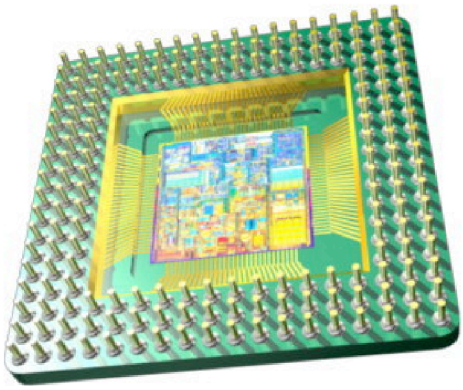
Chapitre 11:
Logique combinatoire asynchrone

11.1 - Introduction

En électronique, le terme "logique" désigne l'une ou l'autre famille de circuits numériques. On parle par exemple de logique combinatoire, de logique séquentielle, de logique synchrone, de logique CMOS, etc… qui sont autant de critères utilisés pour classifier les circuits numériques.

Nous abordons ici la logique combinatoire asynchrone (termes que nous expliquerons plus loin), qui correspond aux opérations les plus simples réalisables sur des signaux numériques.

Le but de l'électronique, c'est de traiter un signal…



2

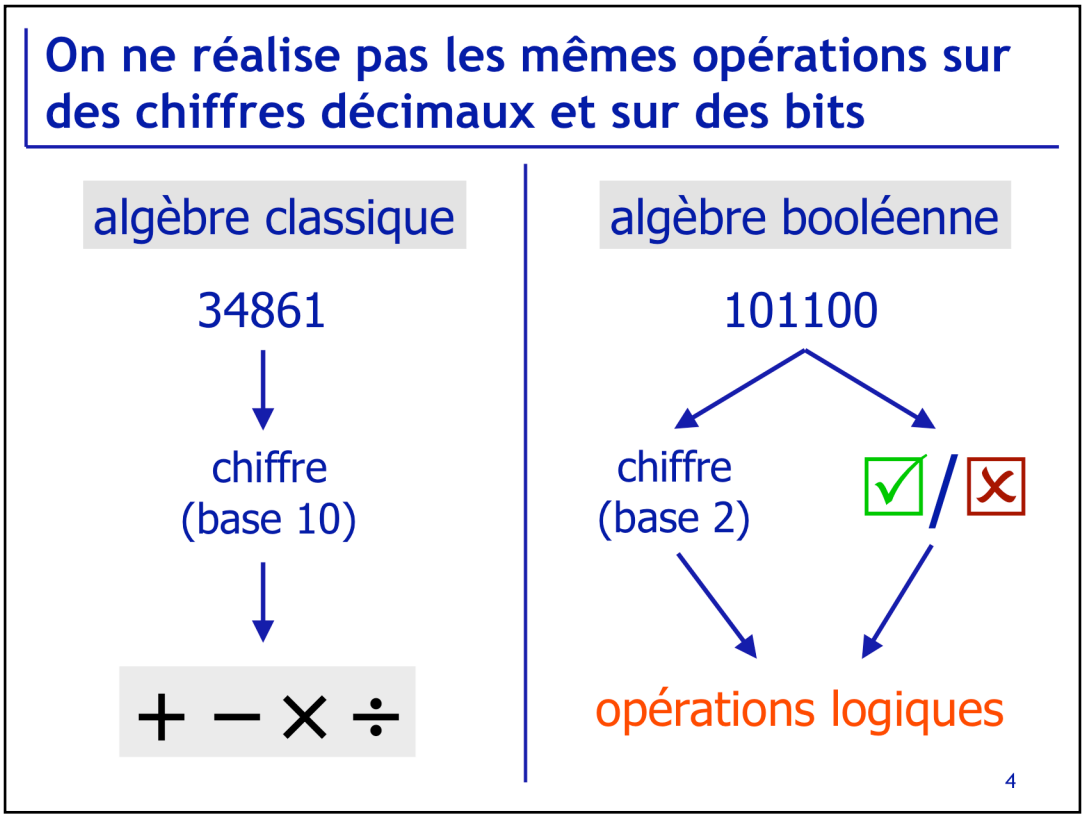
Revenons à notre exemple initial de système numérique: le PC. La première chose dont on parle dans un PC, c'est le processeur (entouré dans la figure de gauche et représenté à droite). Ceci nous rappelle que… **Le but de l'électronique, c'est de traiter un signal**

Après avoir précisé (dans le chapitre précédent) comment une information est représentée dans le signal électrique, nous allons donc maintenant nous intéresser au **traitement** de ce signal. Nous allons cependant commencer par des circuits beaucoup plus simples qu'un processeur: les circuits logiques combinatoires asynchrones.

Chapitre 11:
Logique combinatoire asynchrone

11.2 – Opérations logiques élémentaires: portes logiques

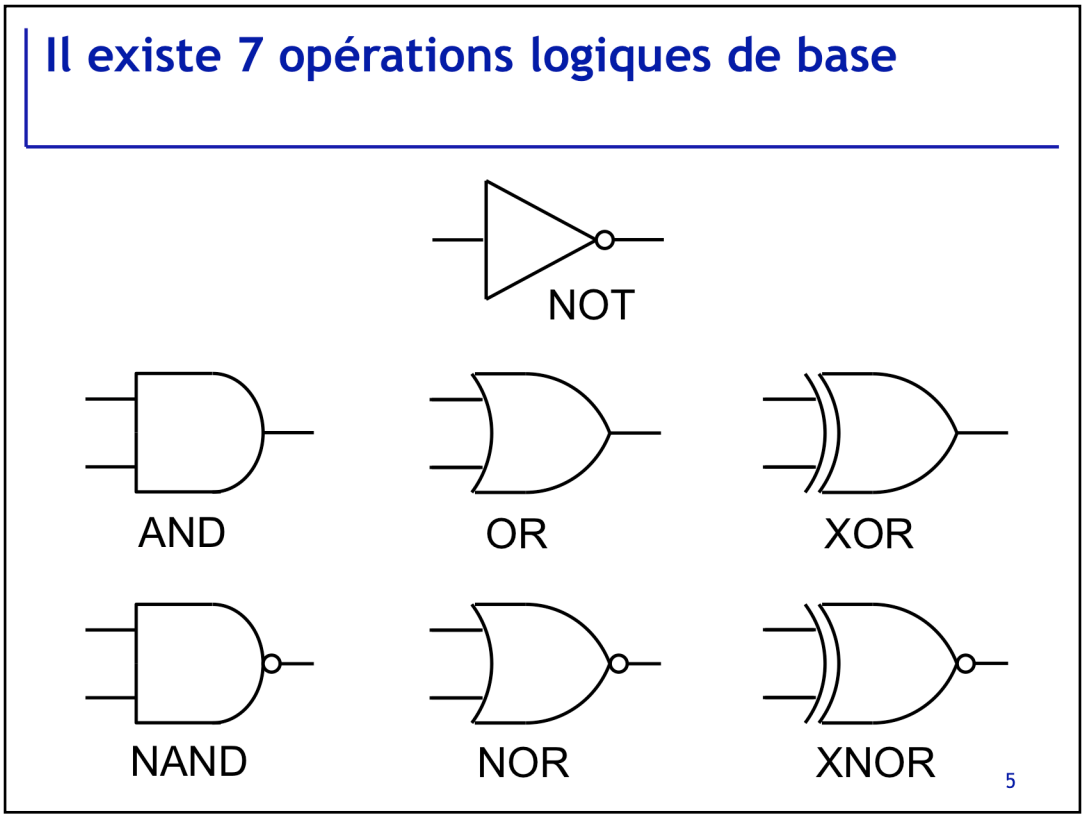
Nous nous intéressons d'abord aux opérations les plus élémentaires réalisables sur des bits.



On ne réalise pas les mêmes opérations sur des chiffres décimaux et sur des bits

En algèbre classique (base 10), un certain nombre d'opérations de base ont été définies pour manipuler l'information (qui apparaît sous forme de chiffres ou de nombres en base 10): il s'agit des opérations classiques d'addition, de soustraction, de multiplication, de division, etc.

En algèbre booléenne, l'information est représentée sous forme de bits. Or on a déjà souligné qu'un bit a une double nature: il peut à la fois représenter un chiffre (en base 2) ou une valeur booléenne (soit "vrai" soit "faux"). Les traitements que peuvent subir les variables booléennes –et donc les bits- ne sont pas les mêmes qu'en algèbre classique. Ces opérations de base sont présentées dans les pages suivantes.

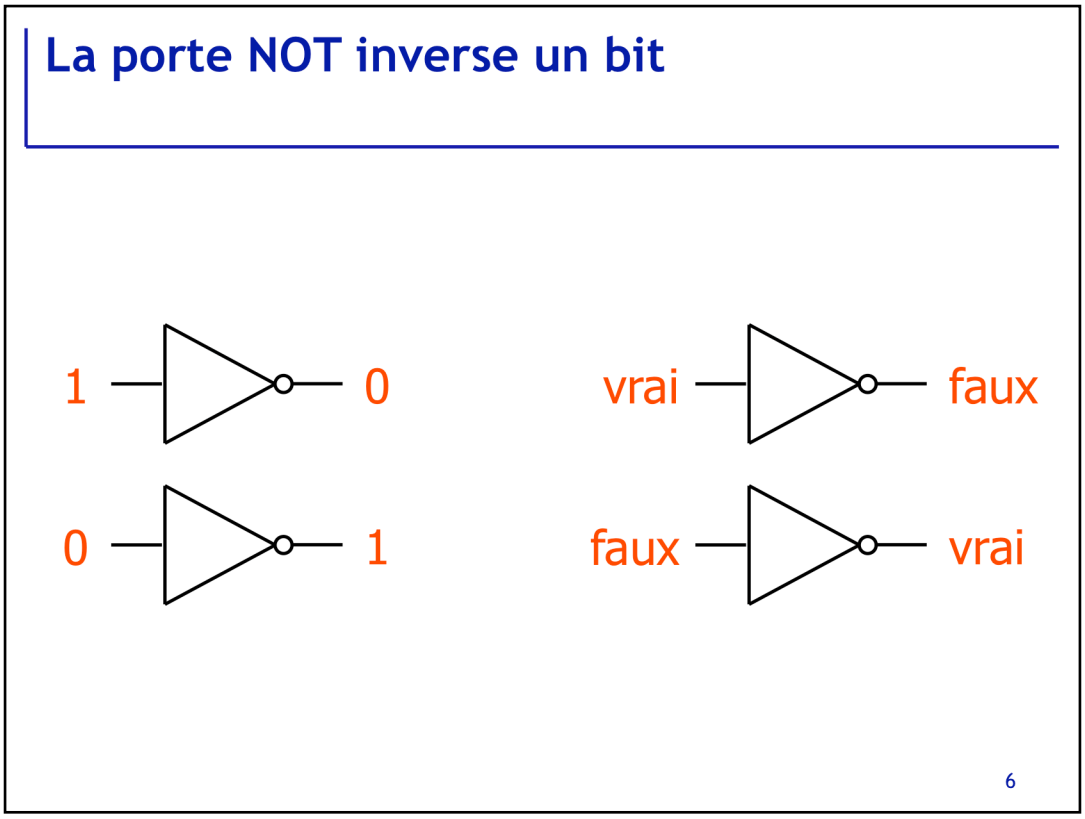


Il existe 7 opérations logiques de base

Les opérations de base de l'algèbre booléenne sont au nombre de 7. Il y a plusieurs manières de définir ou de noter chacune de ces opérations. Nous avons représenté ci-dessus la notation en "portes logiques". Ces portes sont à considérer pour l'instant comme de simples opérateurs mathématiques. Nous verrons néanmoins qu'elles existent aussi physiquement, sous forme de circuits électroniques.

Ces 7 portes peuvent être réparties en trois groupes:
- la porte NOT qui n'agit que sur un bit
- les portes AND, OR (=OU inclusif) et XOR (=OU exclusif) qui représentent les trois opérations de base possibles sur un ensemble de deux bits
- les portes NAND, NOR et XNOR qui sont chacune une combinaison d'une porte NOT avec une porte du second groupe ci-dessus (AND, OR ou XOR) (*)

(*) cette manière de définir les NAND, NOR et XNOR n'est valable que pour des portes à deux bits d'entrée (on peut définir des portes à plus de deux bits d'entrée, mais alors la définition ci-dessus doit être modifiée).



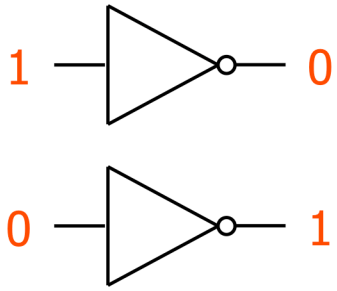
La porte NOT inverse un bit

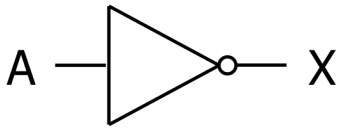
Comme représenté ci-dessus, la porte NOT est *définie* par le fait qu'elle inverse un bit, c'est-à-dire qu'elle transforme un 0 en 1 et un 1 en 0.
Cette opération peut également être vue sous l'angle de la logique booléenne: un VRAI devient FAUX et un FAUX devient VRAI.

Dans le symbole de la porte, on notera que c'est la "boule" qui représente la négation logique (cette même "boule" apparaîtra également par la suite dans d'autres symboles).

N.B.: Dans la suite, nous représenterons uniquement les valeurs 1 et 0, en supposant que la valeur 1 représente également la valeur booléenne VRAI et que la valeur 0 représente également la valeur logique FAUX.

Le comportement d'une porte peut être résumé sous forme de *table de vérité*...





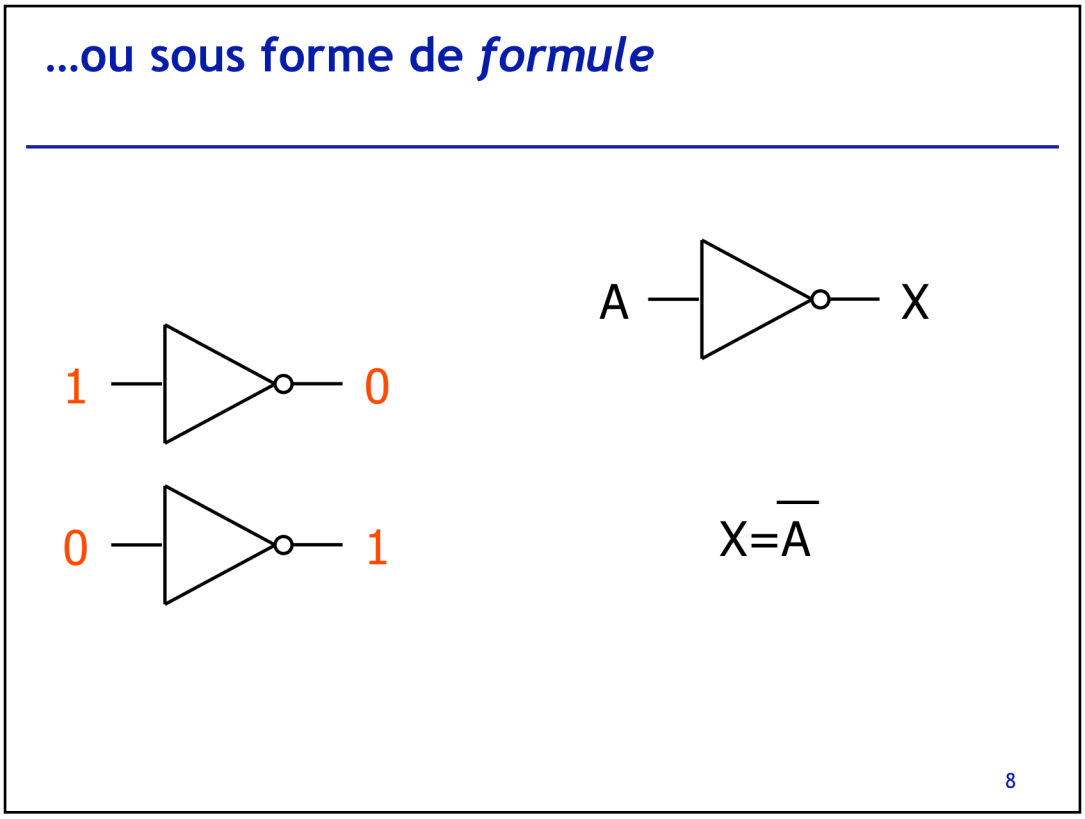
A	X
0	1
1	0

7

Le comportement d'une porte peut être résumé sous forme de table de vérité

Le comportement d'une porte logique (ou plus largement d'un circuit logique) peut être représenté par une table de vérité.
Une table de vérité est un tableau reprenant:
- autant de colonnes qu'il y a de signaux d'entrée et de sortie de la porte (ou du circuit)
- autant de lignes qu'il y a de combinaisons possibles des entrées (soit 2^N lignes de la table pour N entrées)

La table de vérité, qui est un moyen de définir la fonction d'un circuit ou d'une porte logique, contient simplement l'énumération des différents cas possibles et le résultat délivré par la porte ou le circuit pour chacun de ces cas.

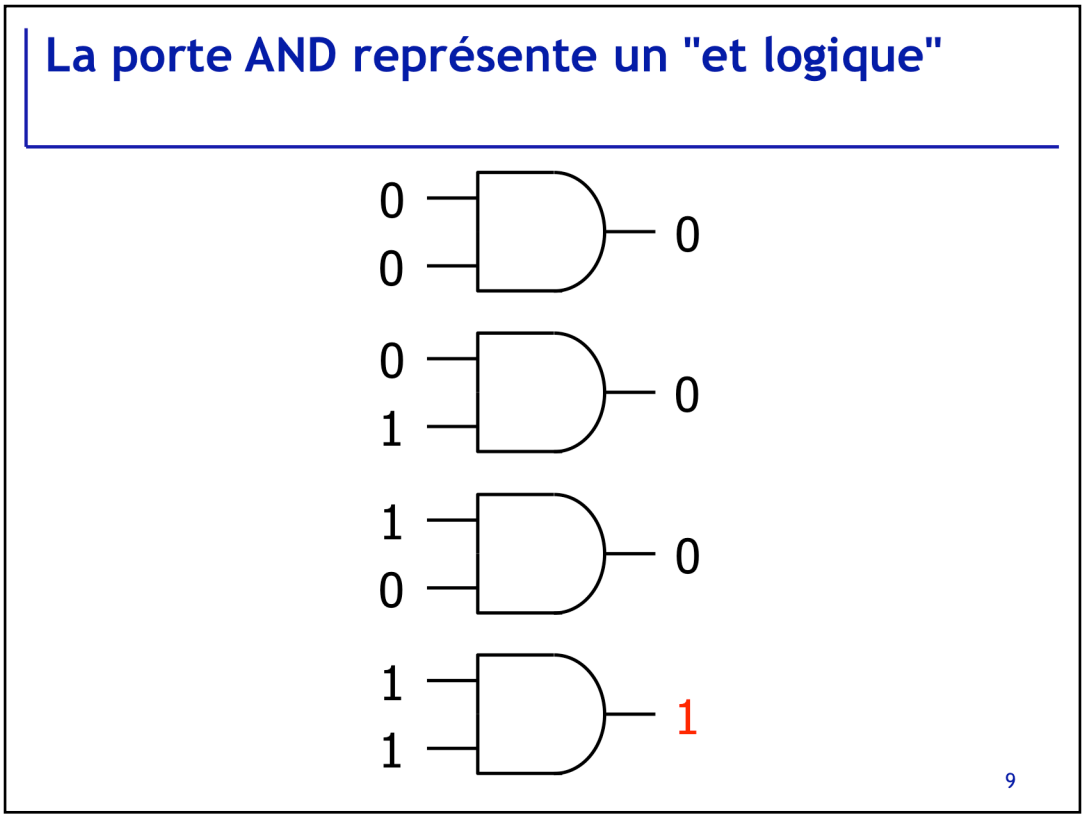


Le comportement de la porte peut également être résumé sous forme de formule

Il existe une notation encore beaucoup plus concise que la table de vérité: comme en algèbre classique, les opérations logiques de base se sont vu attribuer des notations spécifiques.

L'opération NOT est représentée par le fait de surligner la variable à laquelle elle est appliquée (N.B.: cette notation est aussi utilisée pour les signaux actifs à l'état bas: voir plus loin).

Comme le surlignement est parfois difficile à mettre en œuvre dans certains traitements de texte, de nombreuses autres notations sont également utilisées pour l'opération NOT, par exemple:
A'
!A
~A
#A
NOT_A
A_NOT



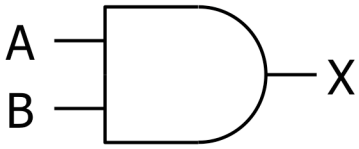
Voyons maintenant la seconde opération logique de base, qui porte cette fois sur deux bits:

La porte AND représente un "et logique"

Comme la porte AND reçoit deux bits en entrée et que chacun de ces bits peut prendre deux états possibles, il y a au total quatre cas possibles.
Par définition, une porte AND délivre la valeur 1 si et seulement si ses deux entrées ont-elles-mêmes simultanément la valeur 1. Dans tous les autres cas (c'est-à-dire si au moins une des entrées porte la valeur 0), une porte AND délivre la valeur 0.

Une porte AND permet donc de tester si deux propositions sont vraies simultanément ("et logique"): dans ce cas, elle délivre la valeur "vrai" (=1).

Le comportement de la porte AND peut être résumé sous les formes suivantes



X=A.B

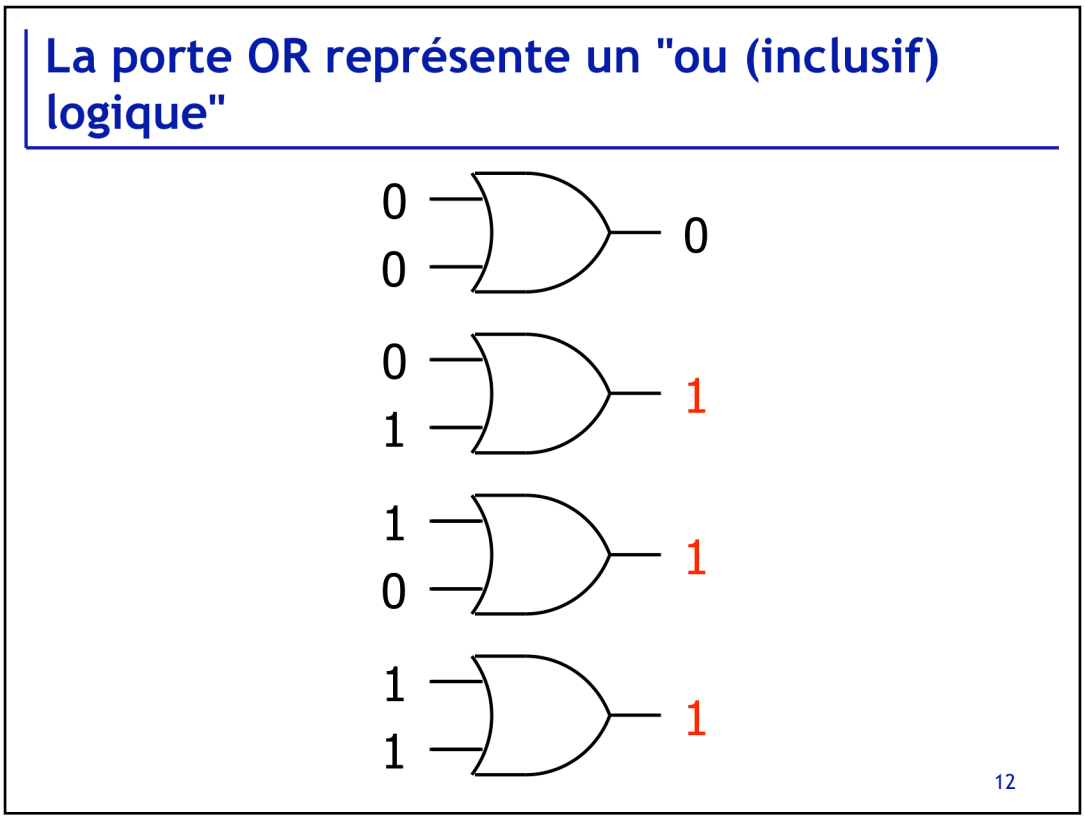
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

11

On a représenté ci-dessus:

1) la table de vérité de la porte AND (qui peut être considérée comme la *définition* de l'opération AND)

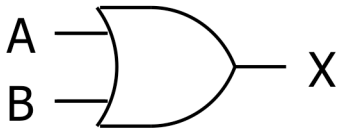
2) la notation mathématique de l'opération AND, qui est identique à la notation de la multiplication (".") en algèbre classique. Ce choix est légitime: si on considère en effet que les deux signaux d'entrée de la porte AND sont des chiffres, on peut vérifier que le résultat délivré par la porte est bien celui qu'on aurait obtenu en multipliant, au sens classique du terme, les deux signaux d'entrée. (Ceci illustre bien l'ambiguïté qui existe dans la signification d'un bit puisqu'une opération définie initialement sur base d'une interprétation booléenne de ces bits se voit représentée par un symbole traduisant ce qui se passe si on interprète ces bits comme des chiffres.)



La porte OR représente un "ou logique"

Par définition, une porte OR délivre la valeur 1 si une des ses entrées au moins porte la valeur 1. En d'autres termes, une porte OR délivre la valeur 0 si et seulement si ses deux entrées ont-elles-mêmes simultanément la valeur 0.

Le comportement de la porte OR peut être résumé sous les formes suivantes...



X=A+B

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

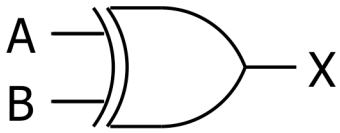
14

On a représenté ci-dessus:

1) la table de vérité de la porte OR (qui peut être considérée comme la *définition* de l'opération OR)

2) la notation mathématique de l'opération OR, qui est identique à la notation de l'addition ("+") en algèbre classique. Ceci est cohérent avec le fait que, si on interprète les signaux logiques comme des chiffres, l'opération OR revient en effet à additionner les deux signaux d'entrée, SAUF pour la dernière ligne de la table de vérité.

La porte XOR représente un "ou exclusif"



$X = A \oplus B$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

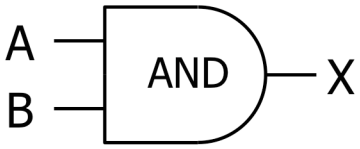
15

Enfin on a représenté ci-dessus:

1) la table de vérité de la porte XOR (qui peut être considérée comme la *définition* de l'opération OR). Celle-ci est identique à celle de la porte OR, sauf pour le cas où les deux signaux d'entrée valent simultanément 1: dans ce cas, la porte délivre le signal 0 (d'où le nom de ou EXCLUSIF). On peut encore formuler l'opération XOR de la manière suivante: si les deux bits sont identiques la sortie vaut 0, tandis que si les deux bits sont différents la sortie vaut 1.

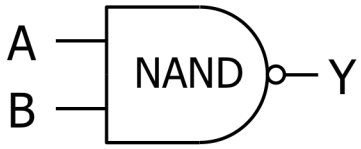
2) la notation mathématique de l'opération XOR, qui est un "+" entouré d'un cercle (pour le distinguer du symbole de l'opération OR).

Porte NAND = NOT(AND)



$X=A.B$

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



$Y=\overline{A.B}=\overline{X}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

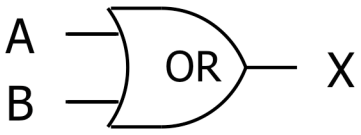
17

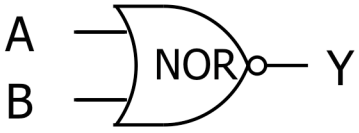
Voyons maintenant les trois dernières opérations logiques.

L'opération **NAND** est définie par la table de vérité ci-dessus (à droite): la dernière colonne de cette table de vérité est en fait simplement l'inverse de la dernière colonne de la table de vérité d'une porte AND (à gauche). L'opération NAND est donc tout-à-fait équivalente à un AND suivi d'un NOT.

En conséquence:
- le symbole d'une porte NAND est celui d'une porte AND à la sortie de laquelle on ajoute la "boule" représentant la négation
- il n'y a pas de symbole mathématique particulier pour représenter l'opération NAND

Porte NOR = NOT(OR)





$X=A+B$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

$Y=\overline{A+B}=\overline{X}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

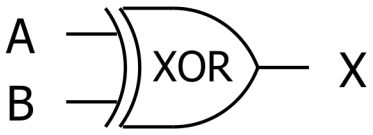
18

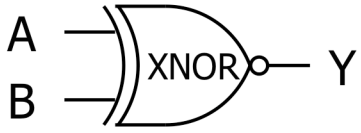
De la même manière…

L'opération **NOR** est définie par la table de vérité ci-dessus (à droite): la dernière colonne de cette table de vérité est en fait simplement l'inverse de la dernière colonne de la table de vérité d'une porte OR (à gauche). L'opération NOR est donc tout-à-fait équivalente à un OR suivi d'un NOT.

En conséquence:
- le symbole d'une porte NOR est celui d'une porte OR à la sortie de laquelle on ajoute la "boule" représentant la négation
- il n'y a pas de symbole mathématique particulier pour représenter l'opération NOR

Porte XNOR = NOT(XOR)





$X = A \oplus B$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

$Y = \overline{A \oplus B} = \overline{X}$

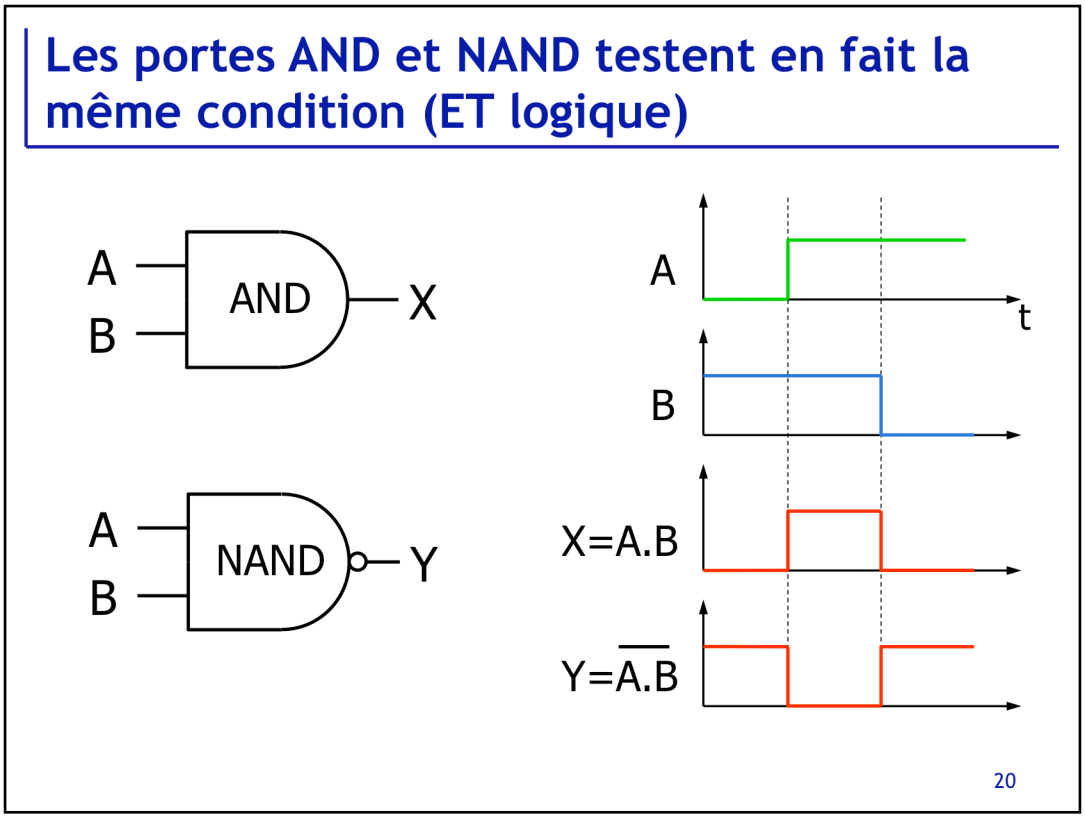
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

19

De la même manière…

L'opération **XNOR** est définie par la table de vérité ci-dessus (à droite): la dernière colonne de cette table de vérité est en fait simplement l'inverse de la dernière colonne de la table de vérité d'une porte XOR (à gauche). L'opération XNOR est donc tout-à-fait équivalente à un XOR suivi d'un NOT.

En conséquence:
- le symbole d'une porte XNOR est celui d'une porte XOR à la sortie de laquelle on ajoute la "boule" représentant la négation
- il n'y a pas de symbole mathématique particulier pour représenter l'opération XNOR



Les portes AND et NAND testent en fait la même condition (*)

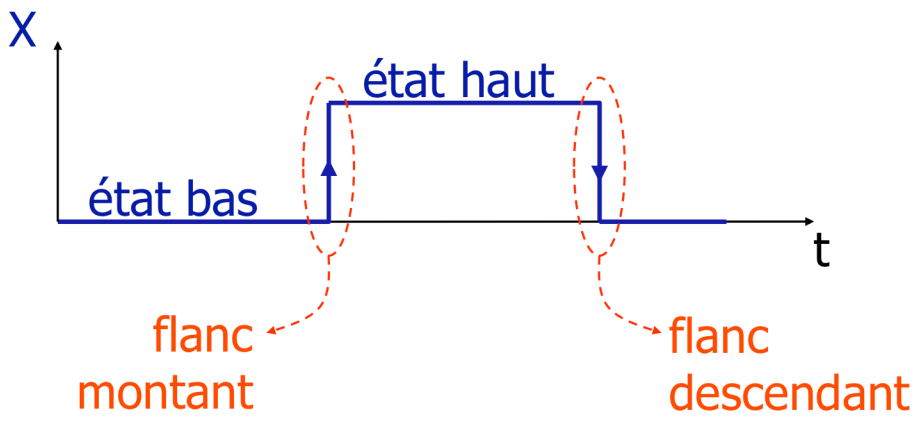
Insistons sur le fait qu'une porte NAND remplit en fait la même fonction qu'une porte AND (**): les deux portes réalisent un "ET logique" (c'est-à-dire qu'elles testent toutes deux que deux conditions A et B sont vraies simultanément), mais le résultat du test est exprimé différemment. Ceci apparaît bien sur les chronogrammes ci-dessus: les deux portes ne passent dans un certain état (1 pour la porte AND, 0 pour la porte NAND) que si les entrées A et B valent 1 simultanément.

Ceci nous amène aux notions de signal actif à l'état haut et de signal actif à l'état bas (voir slides suivants)…

(*) Le même raisonnement est applicable aux couples de portes OR/NOR et XOR/XNOR.

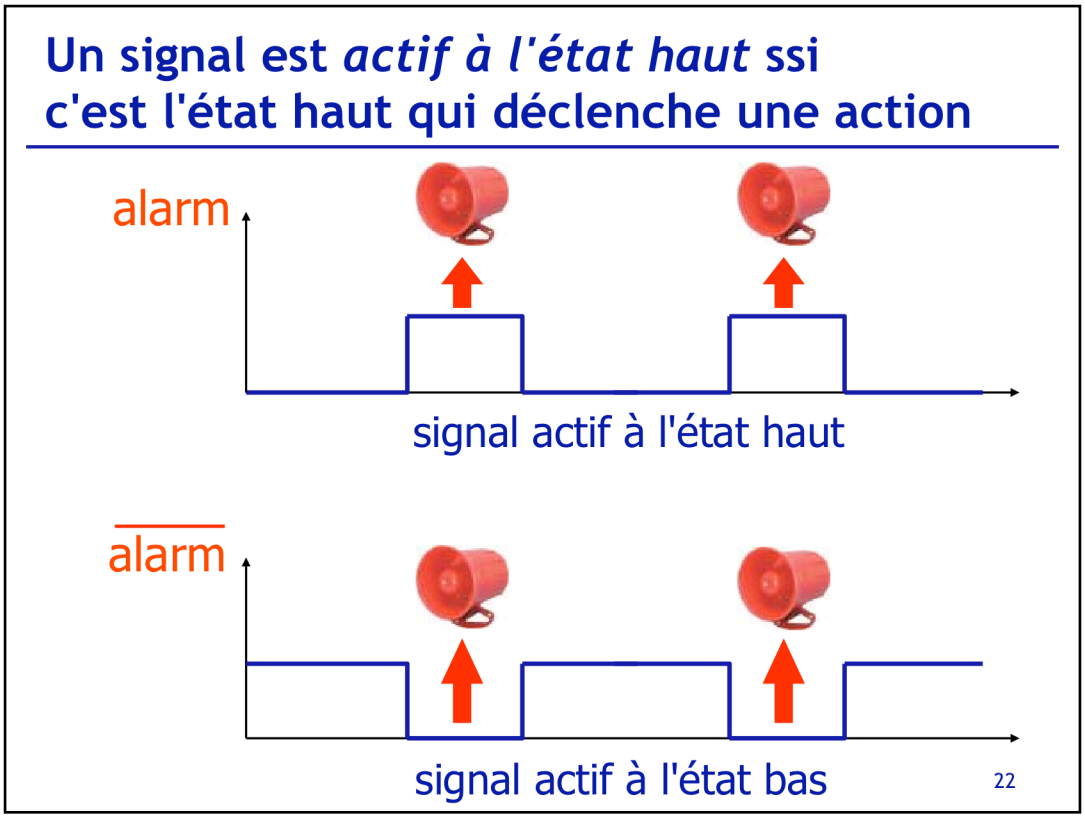
(**) Pourquoi dans ce cas définir la fonction NAND en plus de la fonction AND? Réponse: sur le plan purement logique, la fonction NAND n'apporte effectivement rien de nouveau. Mais nous verrons dans la suite qu'il est plus facile, techniquement parlant, de réaliser la fonction NAND que la fonction AND.

Dans l'évolution d'un signal logique, on distingue des *états* et des *flancs*



21

Tout signal logique apparaît comme une suite de transitions entre deux états. Sur cette base, on peut dire que tout signal:
- soit se trouve à l'état haut
- soit se trouve à l'état bas
- soit subit un flanc montant (=transition de l'état bas vers l'état haut)
- soit subit un flanc descendant (=transition de l'état haut vers l'état bas)

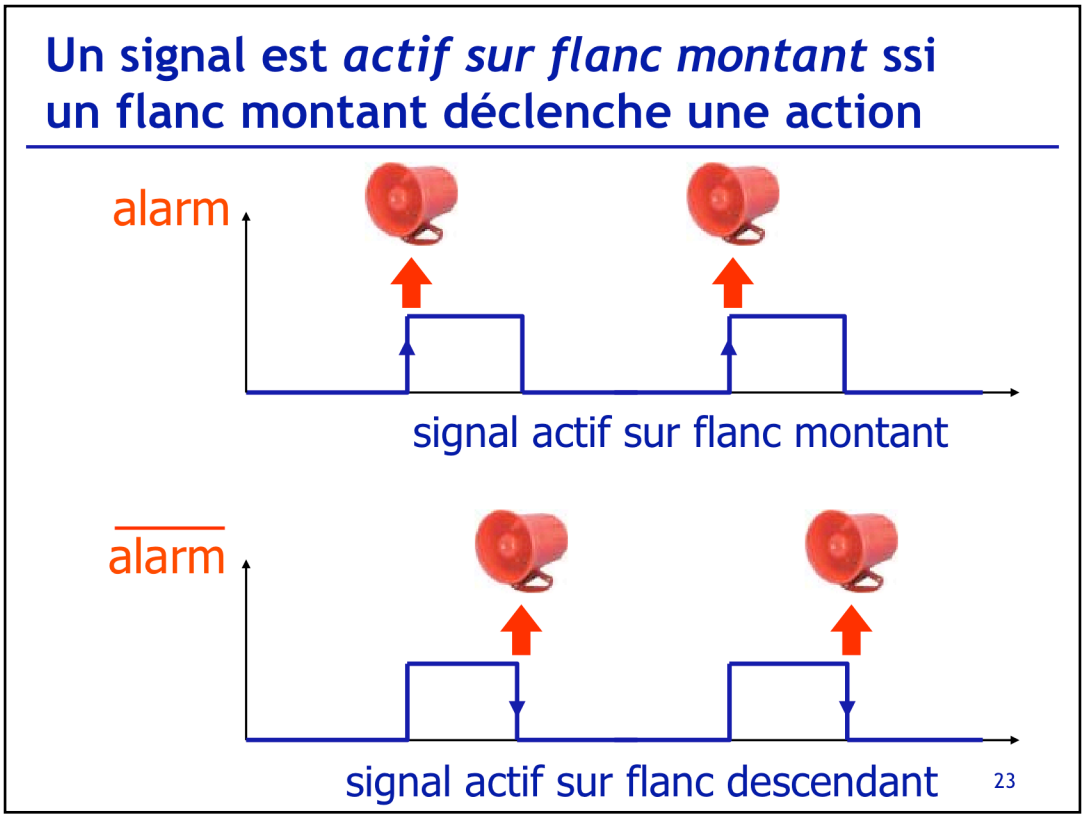


Les états "haut" et "bas" d'un signal ne sont en général pas équivalents pour l'utilisateur: pour de nombreux dispositifs, on peut en effet considérer qu'il existe un état où le dispositif est "actif" (= fonctionne, est allumé, etc) et un état où le dispositif est "inactif" ou "au repos" (= ne fonctionne pas, est éteint, etc). Il existe de ce fait deux possibilités suivant le codage utilisé:
- lorsque l'état actif du dispositif est représenté par l'état haut du signal (et l'état inactif du dispositif par l'état bas du signal), **on parle de signal actif à l'état haut**
- inversément, lorsque l'état actif du dispositif est représenté par l'état bas du signal (et l'état inactif du dispositif par l'état haut du signal), **on parle de signal actif à l'état bas**
On notera que le fait de dire qu'un signal donné est actif à l'état haut ou à l'état bas résulte uniquement d'une interprétation faite par l'utilisateur de la fonction qu'il donne à ce signal. Il s'agit donc purement d'une convention (techniquement, un signal n'est pas intrinsèquement à l'état haut ou à l'état bas).

La différence entre une porte AND et une porte NAND (ainsi qu'entre les paires OR/NOR et XOR/XNOR) est simplement de cet ordre: en supposant que le signal de sortie de la porte doit être actif lorsque ses deux entrées sont à l'état 1, on peut dire qu'une porte AND délivre un signal actif à l'état haut tandis qu'une porte NAND délivre un signal actif à l'état bas.

On parlera encore d'ACTIVER un signal (= le mettre dans son état actif, que ce soit l'état haut ou l'état bas) ou de le DESACTIVER (le mettre dans son état inactif ou "état de repos").

N.B.: un signal actif à l'état bas est généralement surligné (même notation que la négation logique).



Comme on le verra plus tard, certains dispositifs ne sont pas activés par l'un ou l'autre état du signal qui les commande mais uniquement par une *transition* de ce signal. On parle dans ce cas de:
- signal actif sur flanc montant (lorsque c'est un flanc montant qui déclenche le dispositif)
- signal actif sur flanc descendant (lorsque c'est un flanc descendant qui déclenche le dispositif)

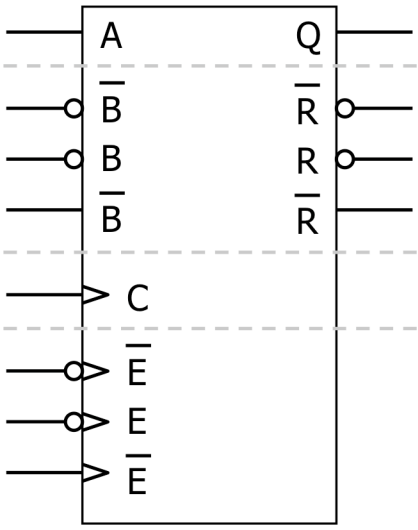
Conventions de représentation de l'activité des signaux

signal actif à l'état haut

signal actif à l'état bas

signal actif sur flanc montant

signal actif sur flanc descendant

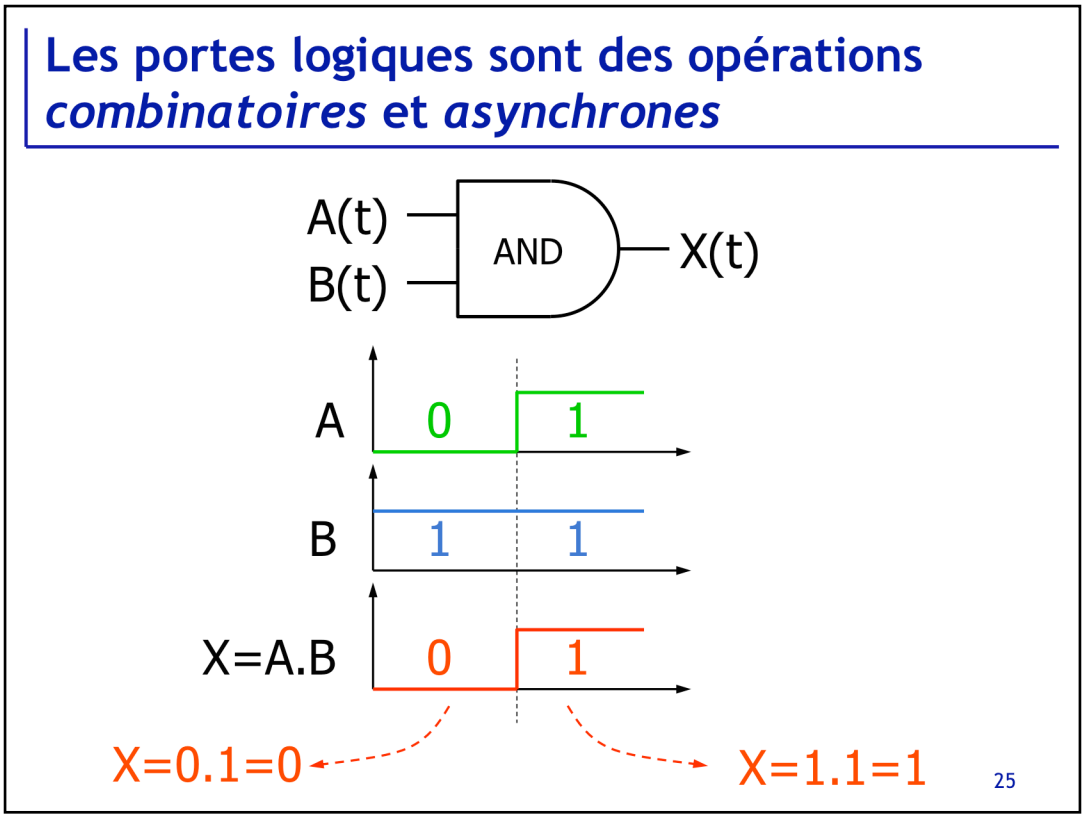


24

Lorsqu'on lit un schéma numérique, il est préférable de connaître la signification des différents signaux (la compréhension du schéma en est facilitée). Il existe à cette fin des notations standardisées, qui ne sont malheureusement pas toujours respectées. On a représenté ci-dessus un circuit avec un certain nombre de signaux d'entrée et de sortie afin d'illustrer ces notations.

Dans la symbolique IEEE (une des normes les plus usuelles), la marque d'un signal actif à l'état bas est à la fois:
- un rond marquant l'inversion
- le surlignement du nom de la variable logique
Il arrive toutefois que l'une de ces deux marques soit absente.

Par ailleurs, le fait qu'un signal soit actif sur une transition (flanc montant ou descendant) se marque par un petit triangle à l'entrée du circuit sur lequel ce signal aboutit:
- un triangle seul désigne un signal actif sur flanc montant
- un triangle accompagné d'une (ou plusieurs) marque(s) de la négation logique désigne un signal actif sur flanc descendant



Voyons maintenant quelques propriétés des portes logiques, en commençant par les notions de logique "combinatoire" et de logique "asynchrone" qui constituent le titre de ce chapitre.

Les portes logiques sont des opérations combinatoires
Un circuit **combinatoire** est un circuit dont le résultat (= la valeur de sortie) à un instant donné ne dépend que des valeurs présentes à l'entrée du circuit à ce même instant (voir figure ci-dessus).
Cette notion est à opposer aux circuits **séquentiels** (voir chapitre correspondant), dont le résultat à un instant donné dépend des valeurs présentes à l'entrée du circuit à ce même instant MAIS AUSSI des valeurs manipulées aux instant précédents (donc de "l'histoire" du circuit).

Par ailleurs, les portes logiques sont des opérations asynchrones
Un circuit **asynchrone** est un circuit qui réagit "immédiatement" (*) à toute variation de ses signaux d'entrée (en modifiant le cas échéant sa valeur de sortie).
Cette notion est à opposer aux circuits **synchrones** dans lesquels certains signaux sont synchronisés sur une horloge, ce qui a pour conséquence que ces circuits ne réagissent pas immédiatement à une modification de leurs entrées mais attendent pour cela le "coup d'horloge" suivant.

En d'autres termes, un circuit combinatoire asynchrone réagit "immédiatement" à toute modification de ses entrées et le résultat obtenu ne dépend que de la valeur des entrées à cet instant.

(*) on verra plus tard qu'il existe quand même un délai, mais le "immédiatement" qui figure ici oppose en fait les circuits asynchrones aux circuits synchrones (voir plus loin chap. 11.4) fonctionnant eux sur le principe d'une horloge.

Propriété: toutes les portes peuvent être réalisées sur base de seules NAND

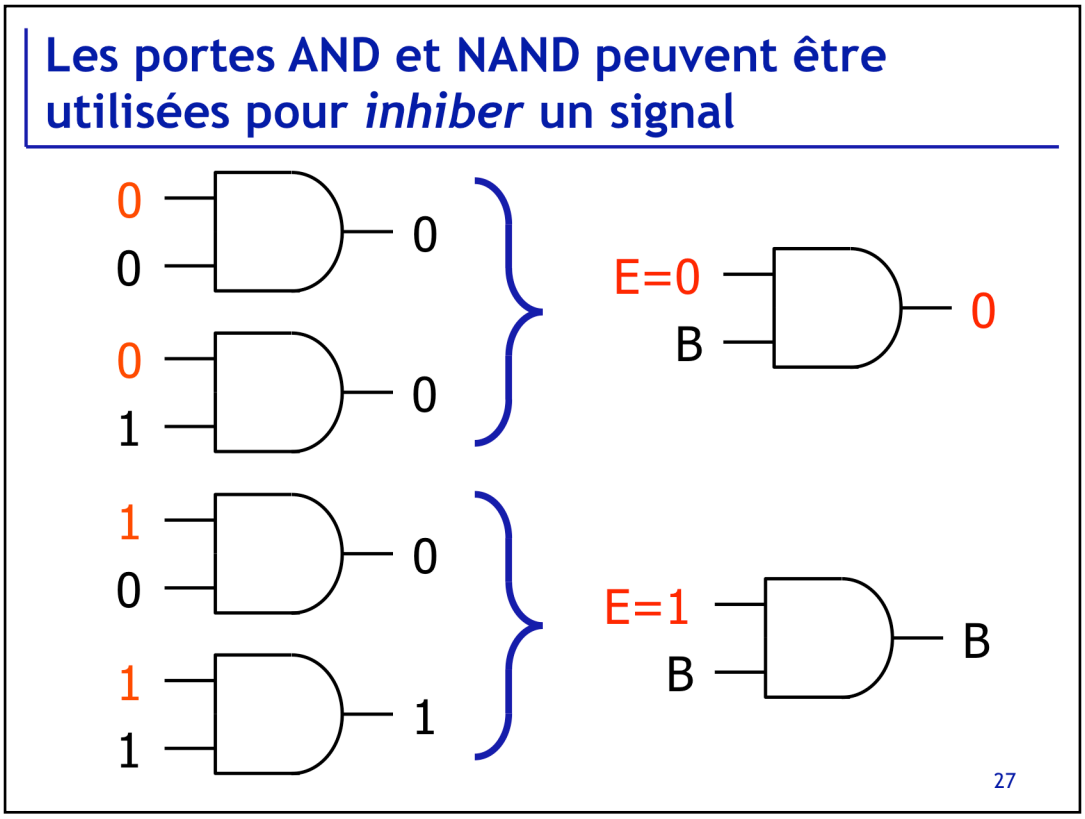
A	X
0	1
1	0

26

On notera une propriété particulière des portes logiques (sans démonstration): toutes les portes élémentaires peuvent être elles-mêmes "fabriquées" en utilisant uniquement des portes NAND (*). Cette propriété est mise à profit dans la fabrication des circuits électroniques.

On retiendra en particulier l'application de cette propriété à la porte NOT: une porte NOT peut être réalisée en appliquant le même signal aux deux entrées d'une porte NAND. Il suffit de le vérifier au moyen des tables de vérité pour s'en convaincre.

(*) en fait cette propriété est également valable pour les NOR: toute porte logique élémentaire peut être remplacée par un ensemble de portes NOR uniquement.



En électronique numérique, il est parfois nécessaire de "masquer" ou "inhiber" un signal, c'est-à-dire de le rendre invisible pour les circuits situés en aval.
Ceci peut se faire au moyen d'une porte AND (ou NAND), en donnant simplement une interprétation particulière aux signaux d'entrée de cette porte:
- la première entrée, notée E (pour "enable" ou "autoriser"), sera considérée comme le signal de commande permettant d'inhiber ou non l'autre signal
- la seconde entrée, notée ici B (entrée du bas dans les figures ci-dessus), sera le signal à masquer

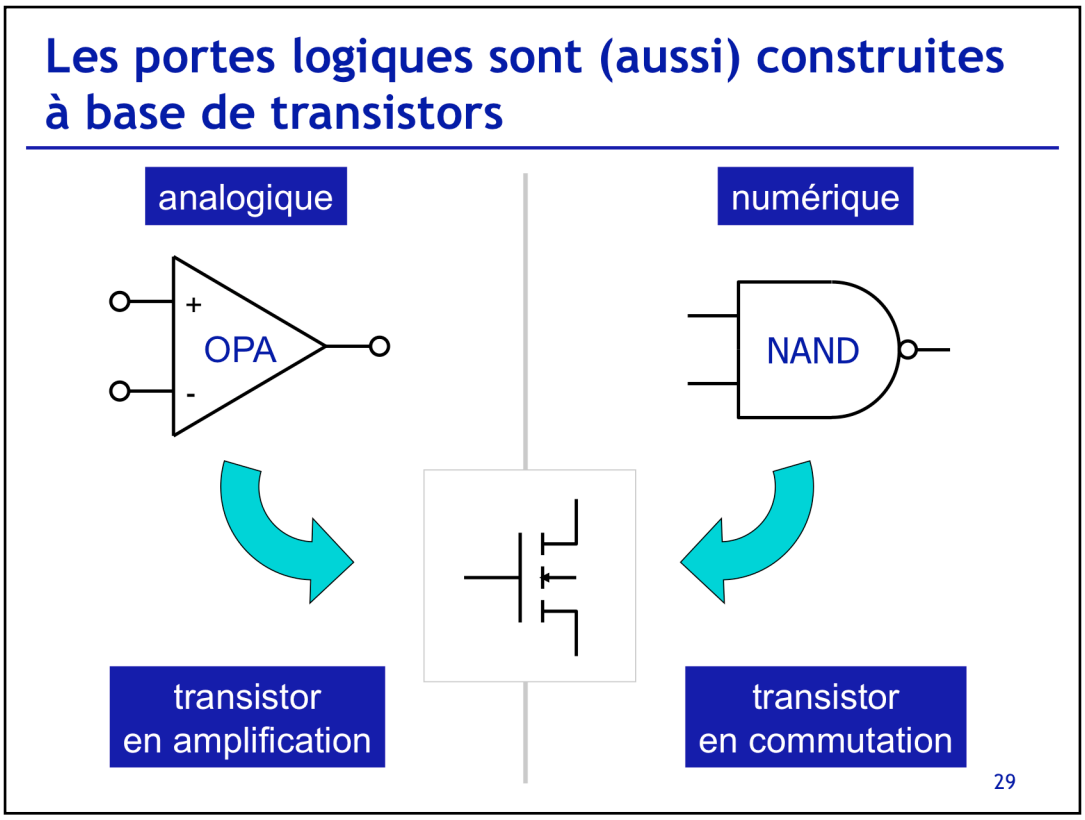
En supposant qu'on utilise une porte AND (slide ci-dessus), on constate dans ce cas que:
- si E=0 (moitié supérieure de la figure), la sortie de la porte vaut toujours 0 quelle que soit la valeur de B: B est donc "masqué" pour la suite du circuit
- si E=1 (moitié inférieure de la figure), la sortie de la porte est égale à B (quelle que soit la valeur de B)
On peut donc masquer ou rendre visible à volonté la valeur B en agissant simplement sur la valeur E. Cette utilisation découle simplement de la table de vérité de la porte AND.

Le "masquage" ou "inhibition" du signal consiste en fait à le rendre inactif (au sens défini précédemment) pour les circuits situés en aval. On utilisera donc:
- une porte AND si le signal B est actif à l'état haut (car dans ce cas E=0 implique que la sortie de la porte vaut 0 quel que soit B)
- une porte NAND si le signal B est actif à l'état bas (car dans ce cas E=0 implique que la sortie de la porte vaut 1 quel que soit B)

N.B.: De manière plus générale, de nombreux circuits possèdent une entrée "enable" (ou son contraire: une entrée "disable") qui permet d'inhiber à volonté les sorties du circuit en question.

Chapitre 11:
Logique combinatoire asynchrone

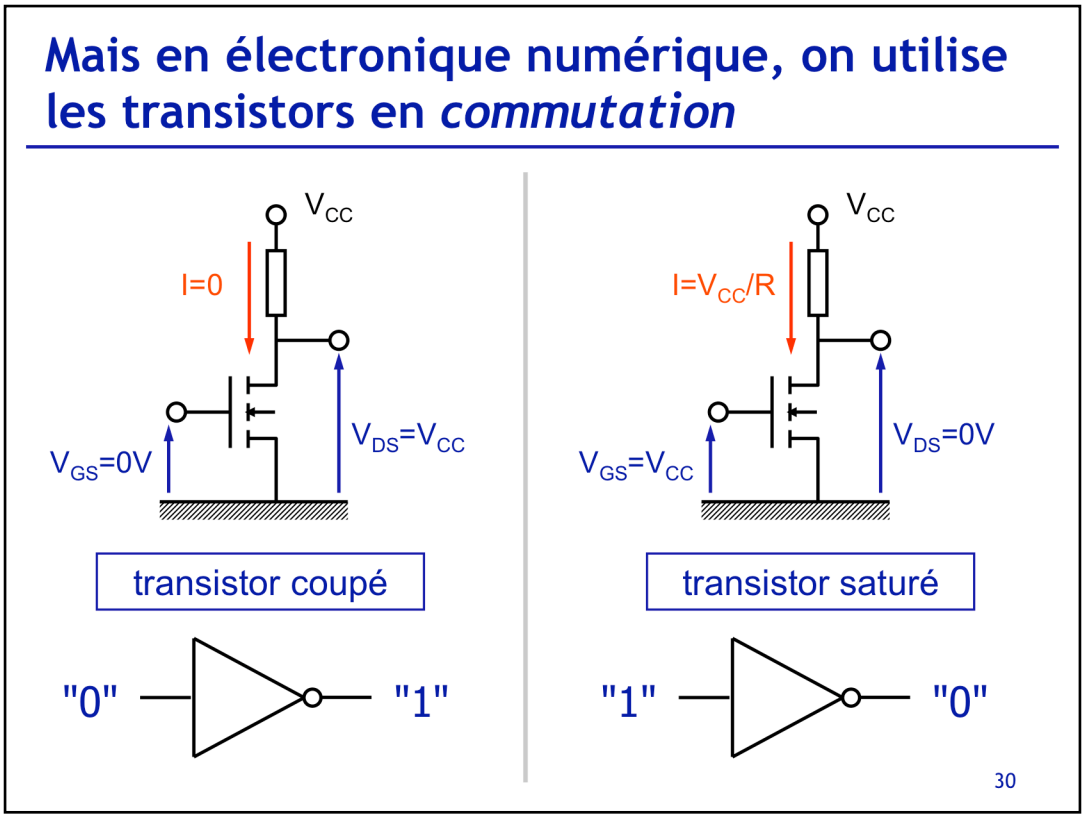
11.3 – Portes logiques et transistors
(le transistor en commutation)



Nous avons déjà vu que le transistor est (avec la diode) LE composant de base de l'électronique analogique. Les ampli-ops notamment sont réalisés à base de transistors.

En électronique numérique également, le transistor est la base de tout. Les portes logiques sont en effet également réalisées à base de transistor.

Si les transistors utilisés dans les deux cas sont similaires (dans 99% des cas des transistors MOSFET), ils sont utilisés de deux manières différentes:
- en analogique, on utilise les transistors "en amplification"
- en numérique, on utilise les transistors "en commutation" (voir slides suivants)



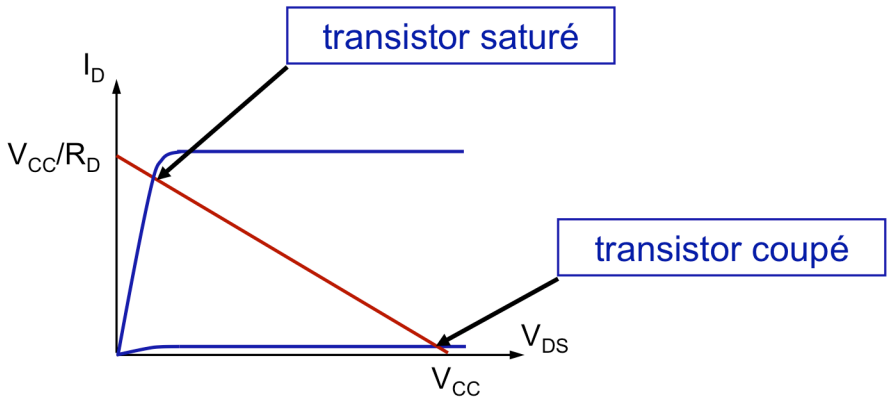
Le principe fondamental d''utilisation des transistors pour traiter des signaux logiques (=transistor en commutation) est expliqué ci-dessus.

On peut remarquer que le circuit de base est le même qu'en analogique: un transistor MOS monté en "source commune", et dont le drain est connecté à une alimentation par l'intermédiaire d'une résistance de charge.

La différence par rapport à l'analogique est que le signal V_{GS} fourni au transistor est un signal logique, c'est-à-dire un signal en "tout ou rien". En conséquence, le transistor fonctionne lui-même en tout ou rien:
- dans le schéma de gauche, on fournit au transistor un signal V_{GS}=0V (état logique "0"): en conséquence, le transistor est coupé et il se comporte en sortie comme un circuit ouvert => aucun courant ne circule dans la résistance de charge et la sortie du montage (V_{DS}) est égale à sa tension d'alimentation: V_{CC}. Cette sortie peut être interprétée comme un état logique "1".
- dans le schéma de droite, on fournit au transistor un signal "maximal": V_{GS}=V_{CC} (état logique "1"): en conséquence, le courant traversant la résistance de charge et le transistor est maximal. Un tel courant est obtenu lorsque le transistor est dit "saturé", c'est-à-dire que sa tension V_{DS} est très faible (ou en première approximation: nulle). Le courant vaut alors V_{CC}/R_D. Cette tension de sortie nulle peut être interprétée comme un état logique "0".

Puisque les états logiques obtenus en entrée et en sortie sont chaque fois opposés, on peut considérer que le montage ci-dessus, utilisé en commutation (c'est-à-dire de sorte que le transistor est soit coupé soit saturé), réalise la fonction logique "NOT".

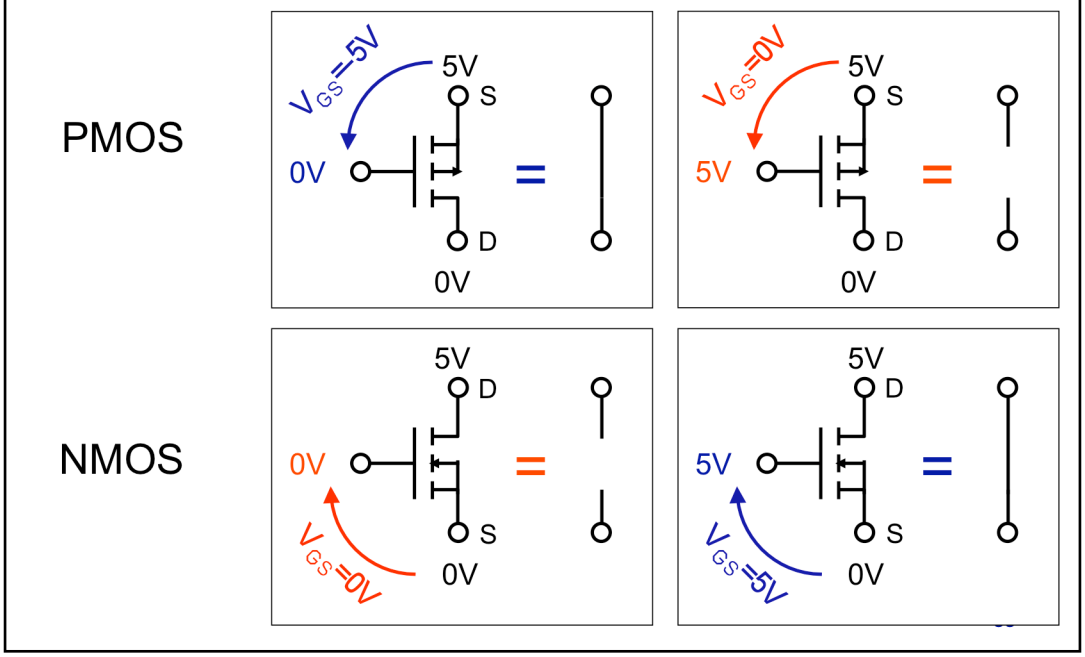
Le transistor en commutation exploite d'autres zones de la caractéristique



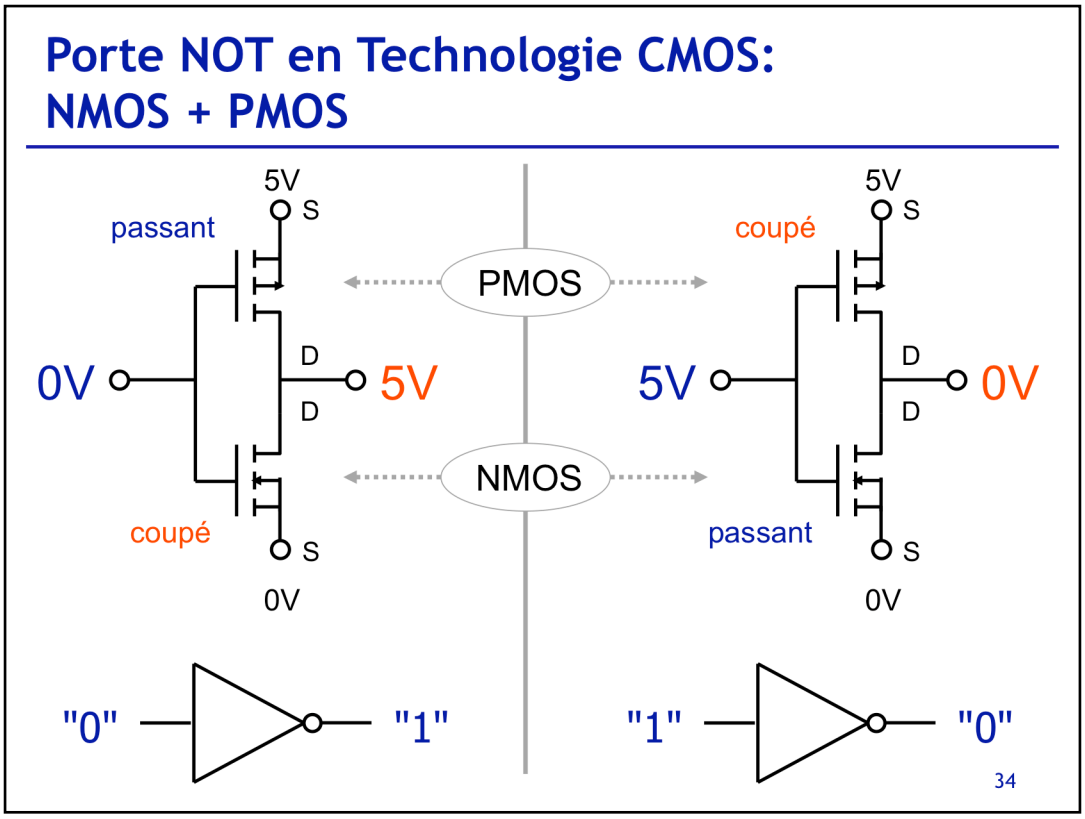
31

Alors qu'en analogique, on utilise la zone centrale de la caractéristique de sortie du transistor, en numérique on utilise les deux zones extrêmes de cette même caractéristique:
- à tension V_{GS} nulle, la caractéristique se confond avec l'axe horizontal: le transistor est coupé (courant nul)
- à tension V_{GS} maximale, la droite de charge coupe la caractéristique dans la zone ohmique, ce qui correspond à une tension de sortie V_{DS} très faible: le transistor est saturé. Cette situation correspond à un courant maximal dans le transistor.

Rappel: pour une même tension V_{GS}, PMOS et NMOS sont dans des états opposés

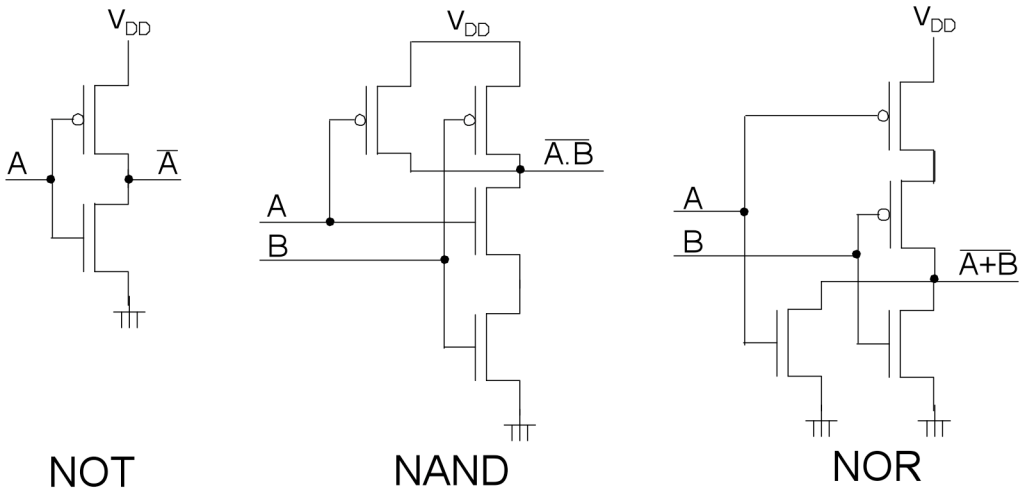


Notez la flèche retournée au centre du symbole des transistors PMOS



En réalité, les portes sont réalisées en technologie CMOS, une technologie qui exploite des paires de transistors complémentaires (NMOS + PMOS).

Portes NAND et NOR en technologie CMOS



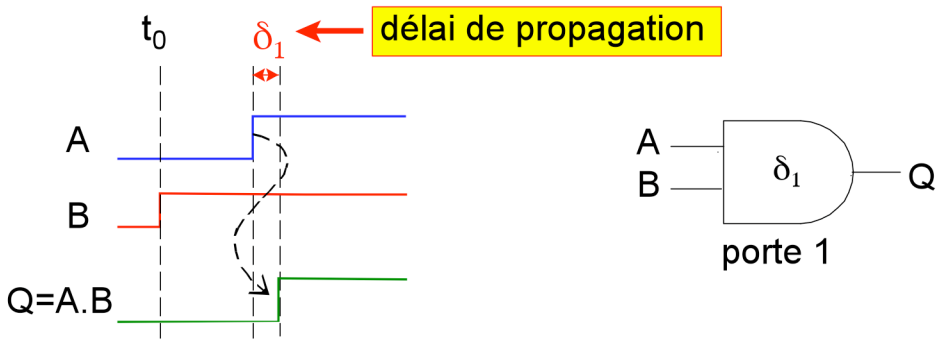
NOT NAND NOR

35

Chapitre 11:
Logique combinatoire asynchrone

11.4 – Propriétés des circuits logiques réels

Le temps de réaction d'une porte n'est pas nul



37

Le mode de fonctionnement des circuits que nous avons vu jusqu'ici est dit "asynchrone". Que signifie ce terme?

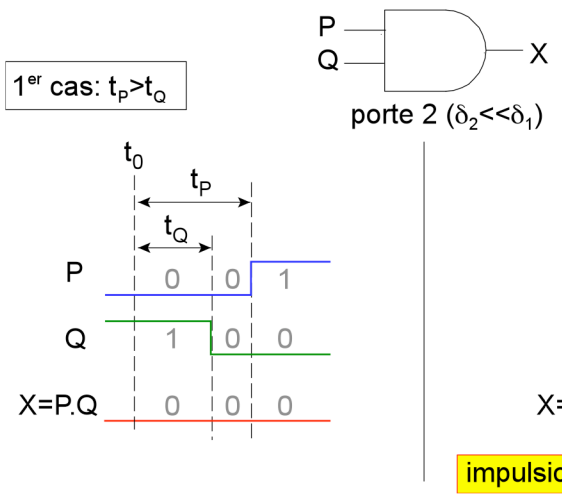
Lorsque les signaux d'entrée d'une porte logique sont modifiés, la sortie de la porte (si elle varie) ne varie pas instantanément: la porte logique possède un certain "temps de réaction" (appelé temps de propagation et généralement noté δ) qui vient lui même du délai nécessaire aux transistors qui la constituent pour commuter d'un état à l'autre. Ce temps est très faible: quelques nanosecondes dans les technologies actuelles.

Les portes logiques qui constituent un circuit ont chacune un temps de propagation différent. En conséquence, les transitions des différents signaux existants dans un circuit n'ont aucune raison d'avoir lieu au même moment: elles sont désynchronisées à cause des différents délais de propagation des différentes portes franchies par chaque signal: on parle de logique "asynchrone".

Nous verrons plus tard qu'il est possible de "synchroniser" d'une certaine manière les transitions des signaux (logique "synchrone"), mais voyons d'abord les conséquences de cette désynchronisation et la notion de "condition de course".

N.B.: Dans le chronogramme (=graphique de l'évolution de signaux logiques en fonction du temps) ci- dessus, la flèche courbe pointillée symbolise le fait que la transition du signal Q est causée par la transition du signal A.

Conséquence: conditions de course et impulsions parasites



38

Considérons maintenant une seconde porte, également de type AND. Nous ne nous intéressons plus à son délai de propagation (que nous pouvons supposer négligeable) mais à son résultat, sachant qu'elle est "nourrie" par des signaux P et Q désynchronisés venant d'autres portes situées en amont (comme la porte 1 par exemple).

Deux cas sont possibles: à gauche, la transition du signal P est en retard sur celle du signal Q. A droite, c'est l'inverse.

Dans les deux cas, les signaux ont un comportement identique: P passe de 0 à 1 et Q passe de 1 à 0. Il en résulte donc, puisque X = (P AND Q) que:
- avant les 2 transitions, X vaut 0
- après les 2 transitions aussi
puisque dans chaque cas un des deux signaux vaut 0.

On voit néanmoins que entre la première et la seconde transition:
- X vaut 0 à gauche (ce qui serait le résultat normal du circuit si on supposait que les transitions étaient vraiment simultanées)
- X vaut 1 à droite (ce qui peut-être interprété comme une impulsion parasite, ou "glitch" en anglais).

Nous arrivons donc à la conclusion que pour un même comportement "fonctionnel" des entrées, le résultat du circuit dépend de l'ordre dans lequel ont lieu les transitions de ces différents signaux. Comme en pratique les délais de propagation des différentes portes sont imprévisibles, on ne peut pas s'assurer que la porte 2 ne générera pas de "glitches". Or une porte située en aval pourrait interpréter ces parasites comme des signaux utiles et réagir en conséquence, propageant un résultat erroné dans le circuit.

En conséquence, on peut résumer la situation d'un circuit asynchrone (c'est-à-dire d'un circuit dans lequel on ne prend aucune précaution particulière pour synchroniser les transitions des signaux) comme suit:
- les délais de progation des portes, différents d'une porte à l'autre, ont tendance à désynchroniser les signaux de manière imprévisible
- des transitions désynchronisées à l'entrée d'une porte peuvent, suivant leur ordre d'arrivée, provoquer l'apparition d'un résultat erroné en sortie de la porte (pendant la durée où toutes les transitions des signaux d'entrée n'ont pas encore été reçues).

En pratique, il est donc fondamentalement inconfortable de travailler en logique asynchrone puisque le résultat du circuit peut être momentanément erroné lors des transitions des signaux.
Il faut cependant noter que ce type de fonctionnement peut être tout-à-fait satisfaisant dans certains cas. La logique asynchrone garde donc toute son utilité.

Chapitre 11:
Logique combinatoire asynchrone

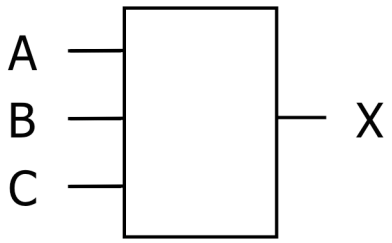
11.5 – Principales fonctions logiques combinatoires

9/10/09 50

Après avoir vu les "portes" de base (opérations logiques combinatoires élémentaires), nous allons passer en revue quelques grands types de circuits combinatoires, pouvant être réalisés au moyen de telles portes.

Le but ici n'est pas de détailler le fonctionnement de ces circuits mais de montrer par quelques exemples le type de fonctions qu'on peut réaliser en logique combinatoire.

Une *fonction logique* combinatoire, c'est…



$X=\overline{C}B\overline{A}+C\overline{B}A+CBA$

C	B	A	X
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

51

Dans les pages qui précèdent, nous avons présenté les 7 opérations combinatoires élémentaires (portes logiques).

De manière plus générale, une **fonction logique combinatoire** fait correspondre à chaque combinaison de M variables d'entrée (il y a donc 2^M combinaisons) une valeur d'une variable logique de sortie (donc un 0 ou un 1).

Une telle fonction peut être définie de différentes manières, par exemple par une formule utilisant les notations de l'algèbre booléenne ou encore par sa table de vérité.

Le slide ci-dessus montre une fonction logique de trois variables A, B et C. On notera que pour énumérer, dans la table de vérité, tous les cas possibles des trois variables A, B et C, il suffit de compter en binaire. Pour cela, il suffit:
- de faire alterner (entre 0 et 1) *à chaque ligne* la variable d'entrée la plus à droite de la table (ici A)
- de faire alterner (entre 0 et 1) *toutes les deux lignes* la variable d'entrée suivante (ici B)
- de faire alterner (entre 0 et 1) *toutes les quatre lignes* la variable d'entrée suivante (ici C)
- etc si d'autres variables sont présentes

Synthétiser une fonction, c'est trouver comment combiner les portes logiques

52

Un des problèmes classiques en électronique numérique est la synthèse d'une fonction logique: synthétiser une fonction logique, c'est déterminer comment la réaliser en pratique (la fonction logique étant définie par une formule ou par sa table de vérité).
Le problème de la synthèse d'une fonction logique ne sera pas vu en détail dans ce cours. On retiendra simplement que:
- toute fonction logique combinatoire écrite sous forme de formule d'algèbre booléenne ou sous forme de table de vérité peut être réalisée en utilisant un nombre fini de portes logiques
- pour chaque formule de ce type, il existe plusieurs combinaisons de portes logiques qui reproduisent cette formule

Les principales fonctions combinatoires sont…

- les additionneurs
  - sur 1 bit: "half adder" et "full adder"
  - sur N bits: parallel binary adder
- [les comparateurs]
  - comparateur 1 bit (XOR) et N bits
- les encodeurs / les décodeurs
- les convertisseurs de code ("code converters")
- les multiplexeurs / les démultiplexeurs
- [les générateurs et testeurs de parité]

53

Addition binaire: le *half-adder* permet d'additionner 2 bits

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 10 \end{array}$$

A	B	C	Z
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

54

Un des traitements de base en électronique numérique consiste évidemment à additionner deux bits (on considère dans ce cas que les bits représentent des chiffres et non des valeurs logiques).

L'opération d'addition la plus simple, représentée ci-dessus, est une fonction à deux bits d'entrée. Il y a donc quatre possibilités (voir moitié gauche du slide ci-dessus). On remarque néanmoins que l'opération "1+1" en binaire donne le résultat binaire "10" (ce qui représente bien la valeur 2 en décimal). Ce résultat ne peut pas s'écrire en un seul chiffre binaire: il y a un "report" ("carry" en anglais).

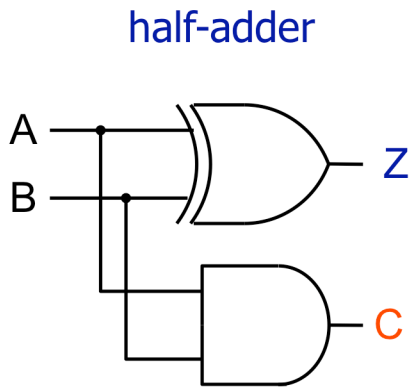
Le résultat d'une addition de deux bits doit donc en fait être exprimé sur deux bits. L'addition binaire sur deux bits doit donc en fait générer deux valeurs: le bit des "unités" (Z dans la table de vérité ci-dessus) et le bit des "deuxaines" (*) (C pour "carry" dans la table de vérité ci-dessus).

(*) "deuxaine" = l'équivalent dans un nombre binaire d'une dizaine dans un nombre décimal (autrement dit: le deuxième chiffre en partant de la droite) :-)

Addition binaire: le *half-adder* permet d'additionner 2 bits

A	B	C	Z
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

AND XOR



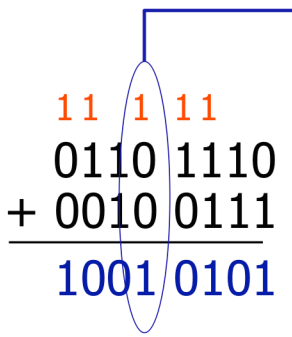
55

Le slide ci-dessus montre la manière la plus simple de générer les valeurs de la table de vérité montrée à gauche.

On peut facilement vérifier que:
- le bit Z est en fait une simple opération XOR entre les bits A et B à additionner
- le bit C est en fait une simple opération AND entre les bits A et B à additionner

Le circuit de droite, qui est donc le résultat de la synthèse (en portes logiques) des fonctions C et Z, est appelé "half-adder".

Addition binaire: le *full-adder* permet d'additionner 2 bits avec report



full-adder

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

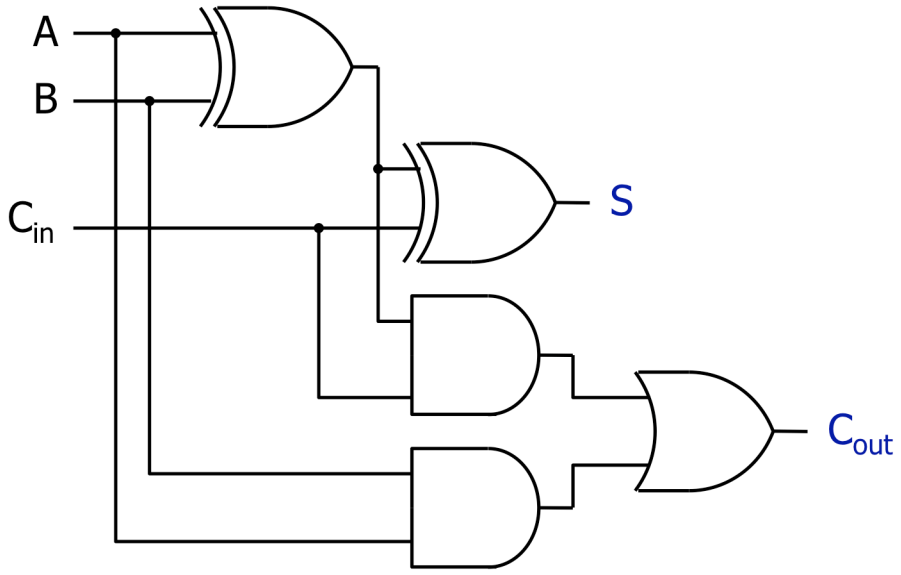
half-adder

56

Lorsqu'on désire additionner deux *nombres* binaires, le circuit half-adder ne suffit pas. En effet, pour tous les poids autres que les unités, il faut être capable de prendre en compte un éventuel report provenant de l'addition de deux bits de poids directement inférieur. En d'autres termes, lorsqu'on additionne deux nombres (en binaire aussi bien qu'en décimal), on additionne en fait trois quantités pour chaque poids: un chiffre de chaque nombre et un éventuel report provenant du poids inférieur.

La table de vérité de droite ci-dessus correspond à cette situation. Elle comporte trois variables d'entrée (A, B et C_{in} pour "carry in", représentant un report "entrant" dans le calcul) et deux variables de sortie (le bit de sortie S ainsi que la valeur du report C_{out} pour le poids supérieur). La première moitié de cette table de vérité (moitié pour laquelle C_{in} vaut 0) est identique à la table de vérité du half-adder.
Le circuit qui implémente la table de vérité complète ci-dessus est appelé "full-adder". C'est le circuit de base qui permet d'additionner deux nombres binaires.

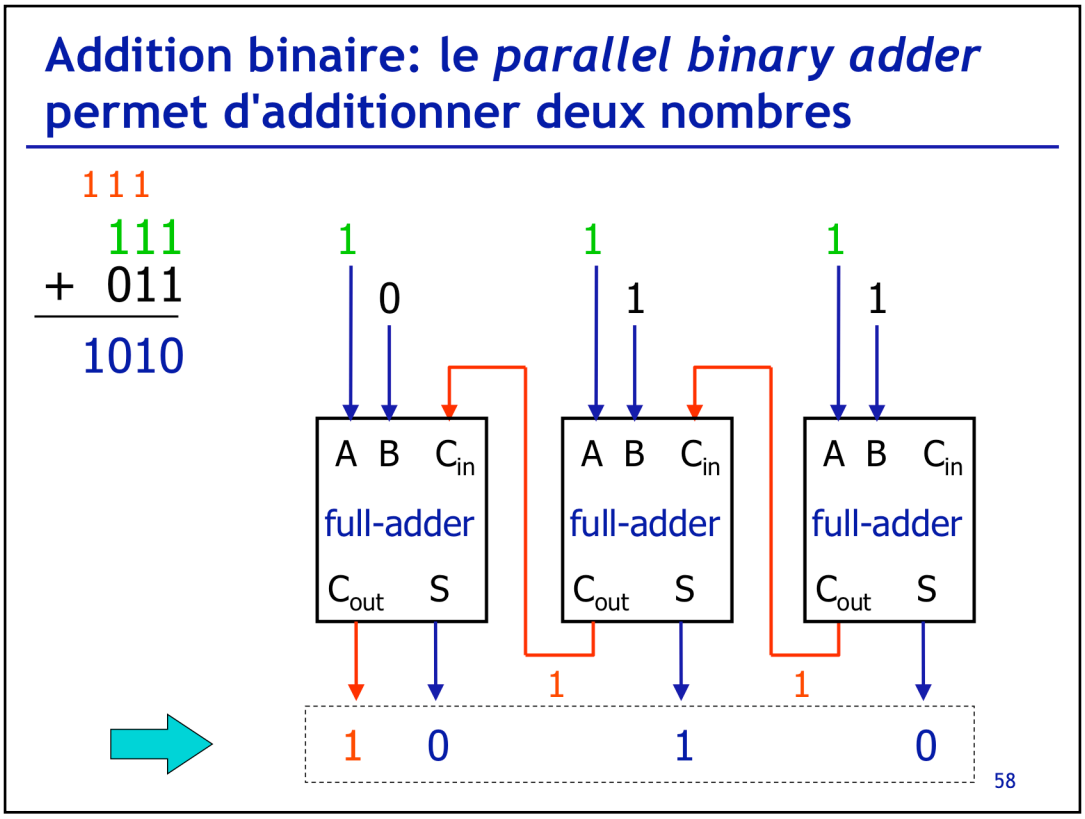
Addition binaire: le *full-adder* permet d'additionner 2 bits avec report



57

A titre d'exemple, on donne ci-dessus le schéma d'un circuit full-adder (trois bits d'entrée et deux bits de sortie).

On pourra vérifier à titre d'exercice que ce schéma satisfait bien la table de vérité du slide précédent.



En supposant qu'on dispose de circuits full-adder (capables d'additionner chacun deux bits avec reports en entrée et en sortie), le schéma ci-dessus montre comment réaliser un circuit (appelé *parallel binary adder*) qui permet d'additionner deux nombres binaires quelconques de 3 bits: il suffit de "cascader" les circuits full-adder (c'est-à-dire de connecter la sortie C_{out} d'un circuit à l'entrée C_{in} du circuit suivant pour transmettre les reports éventuels).
Le résultat est constitué des trois sorties S des circuits full-adder ainsi que de la sortie Cout du circuit de poids le plus fort (soit 4 bits au total).

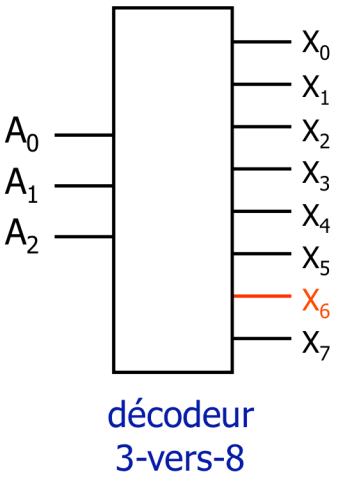
On notera que dans la représentation ci-dessus les circuits full-adder calculant les bits de poids faible se trouvent le plus à droite et les circuits full-adder calculant les bits de poids fort se trouvent le plus à gauche (c'est une bonne habitude à prendre lorsque les circuits manipulent des nombres, qui s'écrivent de gauche à droite).

Ce schéma peut facilement être étendu à un nombre de bits quelconques.

De manière plus globale, cette série de transparents détaillant les circuits d'addition illustrent comment il est possible de "reconstruire" les opérations mathématiques de l'algèbre classique (ici l'addition sur des entiers) en utilisant les portes logiques, opérations élémentaires de l'algèbre booléenne.

Par ailleurs on n'oubliera pas que chacune des connexions représentées dans le circuit ci-dessus représente un bit (0 ou 1) mais que ce bit est en fait représenté physiquement par une valeur de potentiel électrique (par exemple 0V ou 5V) sur cette connexion.

Un *décodeur* active une de ses sorties en fonction du nombre présent à son entrée



A_2	A_1	A_0	X_7	X_6	X_5	...	X_1	X_0
0	0	0	0	0	0		0	1
0	0	1	0	0	0		1	0
0	1	0	0	0	0		0	0
0	1	1	0	0	0	...	0	0
1	0	0	0	0	0		0	0
1	0	1	0	0	1		0	0
1	1	0	0	1	0		0	0
1	1	1	1	0	0		0	0

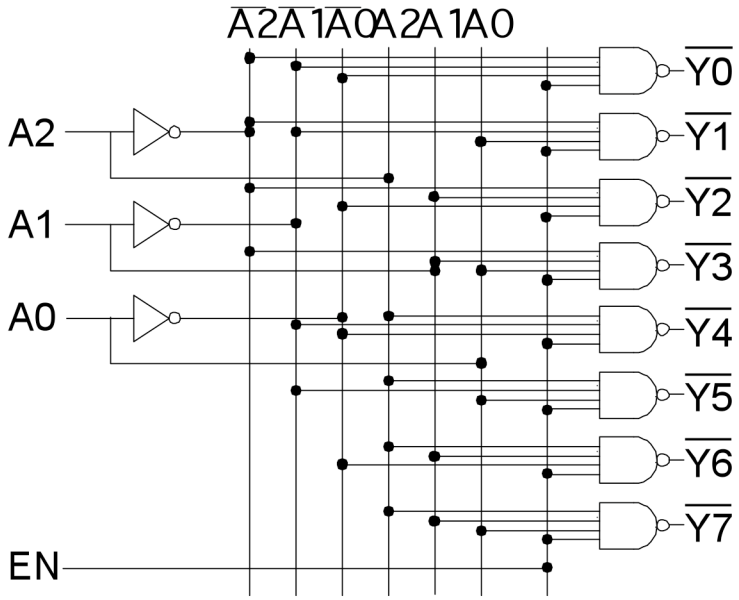
59

Passons maintenant à un second type de circuit combinatoire très courant en pratique: le décodeur.

Il existe plusieurs types de décodeurs. Ce slide illustre le type le plus simple: le décodeur binaire. Un **décodeur binaire** est un circuit à M entrées (représentant ensemble un nombre à M bits) et 2^M sorties numérotées. Sa fonction est d'activer (au sens d'un signal actif à l'état haut ou à l'état bas) LA sortie qui correspond à la valeur binaire représentée par ses M bits d'entrée.
Le slide ci-dessus montre par exemple un décodeur "3-vers-8" (M=3). On peut voir dans la table de vérité de droite que pour la valeur binaire "110" (soit 6 en décimal), toutes les sorties sont à 0 sauf la sortie X_6 qui est à 1. On peut donc en déduire que les sorties de ce décodeur sont actives à l'état haut. Inversément, il existe des décodeurs dont les sorties sont actives à l'état bas.

Ce type de circuit est utilisé pour activer un circuit parmi 2^M sur base d'un nombre désignant ce circuit.

Exemple: schéma de décodeur binaire

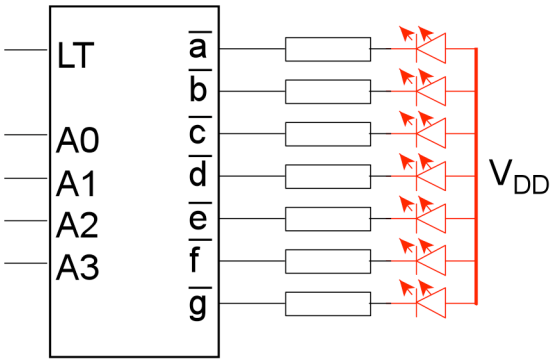


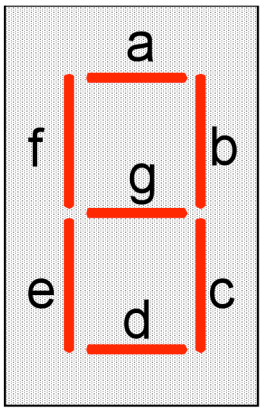
60

A titre d'exemple, on a représenté ci-dessus le schéma interne d'un décodeur binaire "3-vers-8". Comme on le voit un tel décodeur est simplement composé de trois portes NOT et de 8 portes NAND (à 4 entrées chacune) connectées de manière appropriée.

Le décodeur illustré ci-dessus possède deux particularités par rapport à celui présenté dans le slide précédent:
- il possède une entrée supplémentaire ENABLE (EN) qui inhibe (=masque) toutes les sorties lorsqu'elle est inactive (en effet: le signal EN aboutit sur des portes NAND, donc quand EN=0, les portes délivrent toutes la valeur logique "1" quelles que soient leurs autres entrées, valeur qui correspond ici à l'état inactif des sorties du décodeur).
- ses sorties sont actives à l'état bas (ce qu'on peut vérifier par exemple de la manière suivante: lorsque EN est à 1, toutes les portes NAND de sortie sont forcément à 0 quelles que soient les valeurs des bits d'entrée A0, A1 et A2). Ces sorties sont d'ailleurs surlignées (donc actives à l'état bas).

Exemple: décodeur "BCD / 7-segments"





61

De manière plus générale, un décodeur est un circuit:
- qui possède plus de sorties que d'entrées
- et qui, pour chaque combinaison de bits présents à son entrée, active une combinaison particulière de ses sorties.

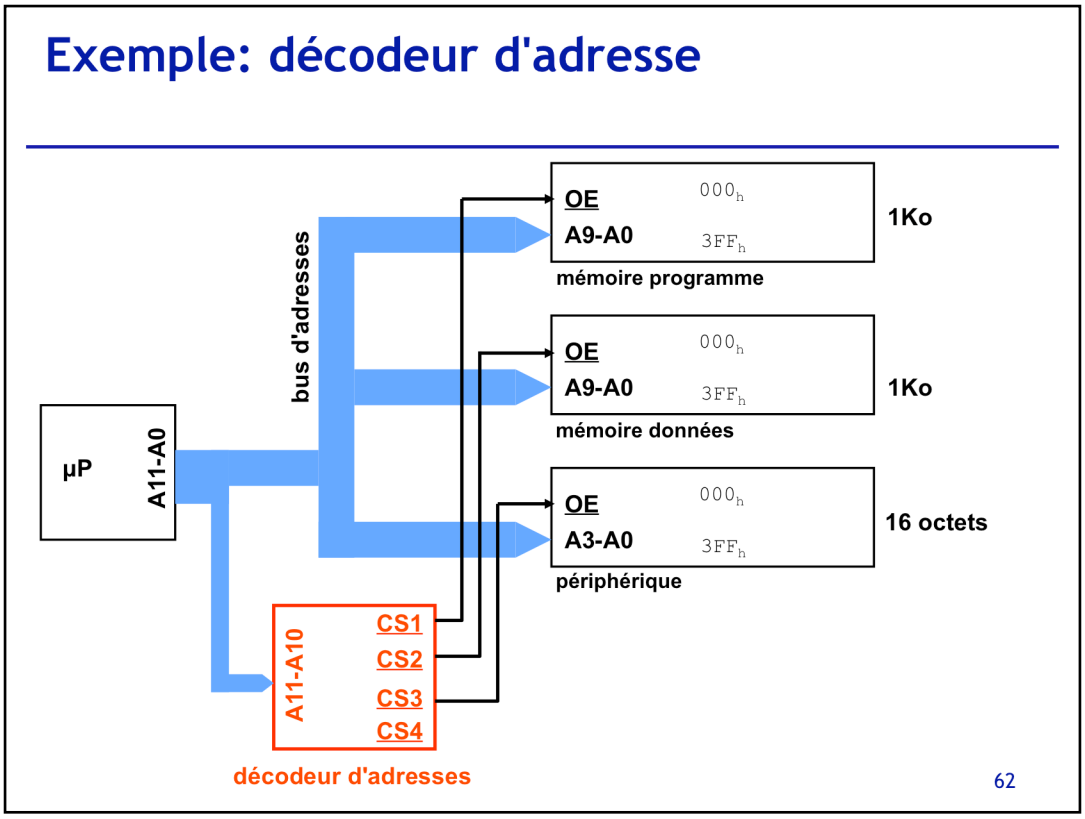
On voit ici un exemple de décodeur utilisé dans une application particulière: l'affichage d'un chiffre à destination d'un utilisateur.

Lorsqu'on désire afficher un chiffre décimal pour un utilisateur humain, il est courant d'utiliser un codage particulier appelé "BCD" pour "binary coded decimal" (décimal codé binaire). Ce codage consiste à utiliser quatre bits pour représenter les chiffres de 0 à 9 (et non de 0 à 15 comme on pourrait le faire avec 4 bits).

Par ailleurs, l'affichage d'un chiffre pour un utilisateur "humain" se fait souvent au moyen d'un "afficheur 7 segments", représenté à droite ci-dessus. Un tel afficheur comporte 7 segments (indiqués par les lettres "a" à "f") qui peuvent être rendus individuellement visibles ou non, de manière à former ensemble un chiffre en écriture arabe. Par exemple, le chiffre 4 s'obtient en allumant les segments b, c, g et f.
On a représenté sur le dessin de gauche un circuit où chaque segment est réalisé au moyen d'une diode électroluminescente (LED). Les anodes des 7 diodes sont connectées ensemble (anode commune) et portées au potentiel positif V_{DD}. Pour allumer une diode, il suffit donc de porter sa cathode à un potentiel inférieur à V_{DD}-V_{TH}. En pratique, cette ddp sera appliquée au travers d'une résistance de limitation de courant (voir "circuits de polarisation" dans le chapitre sur les diodes). En faisant passer la sortie correspondante du décodeur à la masse (sortie active à l'état bas), on allume la LED (segment) connectée à cette sortie.

Dans ce contexte, le rôle du décodeur est de convertir un chiffre décimal (codé sur 4 bits, A3 à A0, au moyen du format BCD) en sept signaux de commande permettant d'allumer les segments appropriés d'un afficheur 7-segments (cfr exemple déjà donné ci-dessus: le chiffre 4 s'obtient en allumant les segments b, c, g et f). Ce type particulier de décodeur peut être vu comme un ensemble de 7 fonctions logiques à 4 entrées chacune.

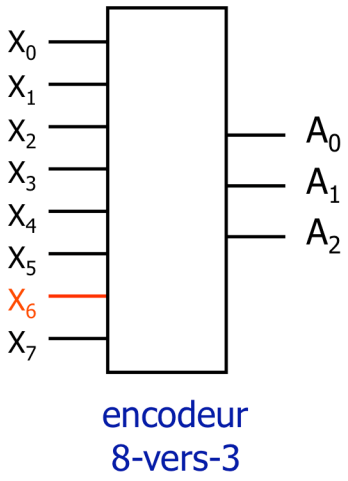
Dans le circuit représenté ci-dessus, l'entrée supplémentaire LT (Lamp Test) permet d'allumer tous les segments indépendemment du nombre présent en entrée du décodeur.



Une autre application particulière des décodeurs est le décodeur d'adresses, qui sera vu dans le chapitre consacré aux systèmes à microprocesseurs.

Un système à microprocesseur comporte plusieurs mémoires (à droite ci-dessus). Sur base d'un nombre binaire appelé "adresse" et fourni par le microprocesseur, le décodeur d'adresse calcule quelle mémoire est concernée et active la sortie correspondante.

Un *encodeur* délivre la valeur désignant son entrée active

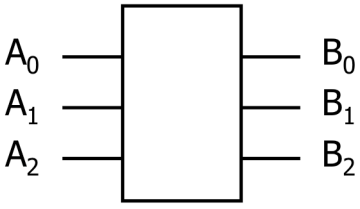


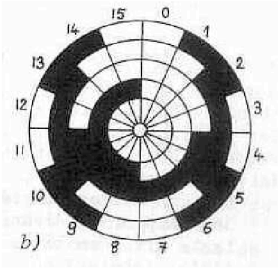
63

Un encodeur réalise la fonction inverse du décodeur.

En particulier, un **encodeur binaire** est un circuit à 2^M entrées numérotées et M sorties représentant ensemble un nombre à M bits. Sa fonction est d'envoyer sur ses sorties le numéro de l'entrée qui est active.

Les *convertisseurs* traduisent un code en un autre





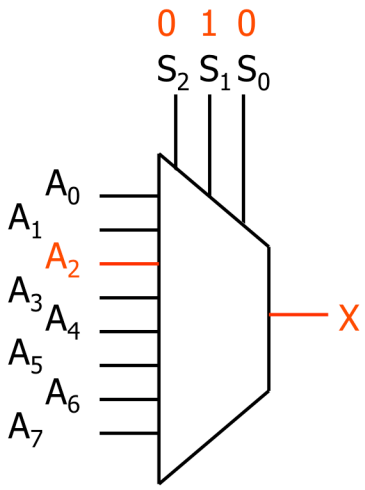
A	B
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

65

Enfin, sachant qu'il existe de nombreuses manières de coder un nombre ou une variable numérique en bits, un **convertisseur** ("code converter" en anglais) est simplement un circuit qui joue le rôle de "traducteur" entre deux codes différents.

exemple:
- hexadécimal en deux valeurs BCD (ou l'inverse)
- code binaire vers code GRAY (ou l'inverse)
etc

Le *multiplexeur* permet de sélectionner une entrée parmi 2^N

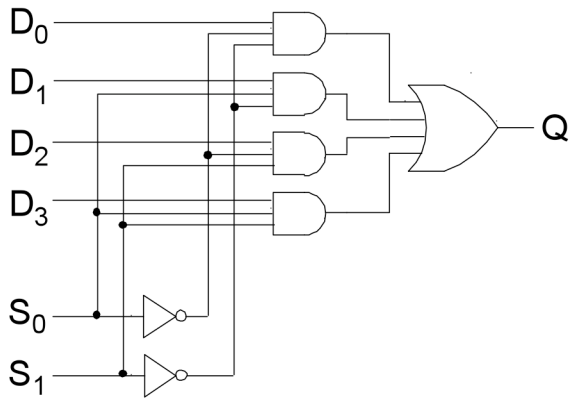


S_2	S_1	S_0	X
0	0	0	A_0
0	0	1	A_1
0	1	0	A_2
0	1	1	A_3
1	0	0	A_4
1	0	1	A_5
1	1	0	A_6
1	1	1	A_7

66

Le **multiplexeur** est un "aiguillage numérique" à 2^N entrées et 1 sortie. La sortie reflète l'entrée qui porte le numéro codé par N bits de sélection (bits S_0 à S_{N-1}).

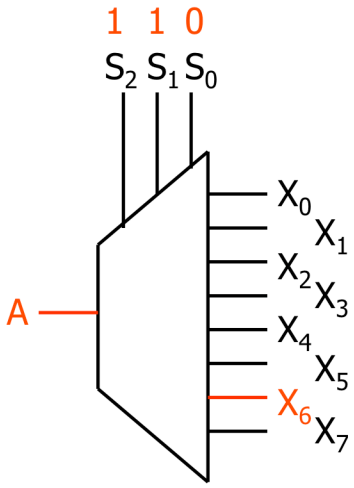
Exemple: schéma d'un multiplexeur à 4 entrées



67

A titre d'exemple, on voit ici le schéma interne d'un multiplexeur 4 vers 1.

Le *démultiplexeur* permet de sélectionner une sortie parmi 2^N

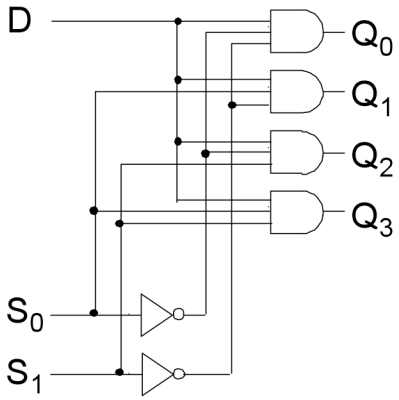


S_2	S_1	S_0	X_7	X_6	X_5	...	X_1	X_0
0	0	0	0	0	0		0	A
0	0	1	0	0	0		A	0
0	1	0	0	0	0		0	0
0	1	1	0	0	0	...	0	0
1	0	0	0	0	0		0	0
1	0	1	0	0	A		0	0
1	1	0	0	A	0		0	0
1	1	1	A	0	0		0	0

68

L'aiguillage inverse du multiplexeur est appelé **démultiplexeur** et possède 1 entrée et 2^N sorties. Sa table de vérité est simple:
- si l'entrée A est inactive, aucune sortie n'est active
- si l'entrée A est active, la sortie active est celle dont l'indice est le numéro codé par les bits de sélection S_{N-1} à S_0.

Exemple: schéma d'un démultiplexeur à 4 sorties



69

A titre d'exemple, on voit ici le schéma interne d'un démultiplexeur à 4 sorties.