Elec201

# Processeurs et Architectures Numériques

Jean-Luc Danger
Guillaume Duc
Tarik Graba
Philippe Matherat
Yves Mathieu
Lirida Naviner
Alexis Polti
Jean Provost

Polycopié des cours du module PAN (Processeurs et Architectures Numériques)
de Télécom-ParisTech.

Le polycopié est disponible au format PDF à l'adresse suivante :
https ://sen.enst.fr/elec201-processeurs-et-architectures-numeriques-2a

Les textes des TP sont accessibles à l'adresse suivante : https ://sen.enst.fr/node/148

# Table des matières

# Liste des tableaux

# Table des figures

# Première partie

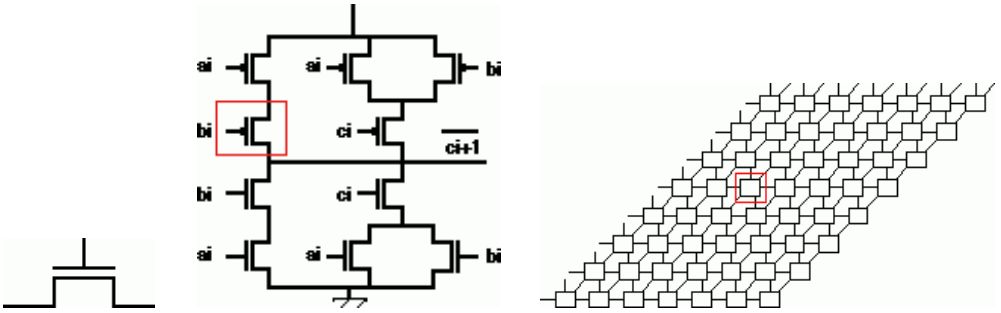
# Leçons

# Chapitre 1

# Traitement matériel de l'information

## 1.1 Du composant aux systèmes numériques

Alors que le cerveau de l'être humain qui a écrit ce texte comporte $10^{12}$ neurones, le micro-ordinateur qui a servi à le mettre en forme comporte au maximum $10^{10}$ composants de base : les *transistors*. Malgré cette relative simplicité, la réalisation de tels systèmes de traitement est difficilement concevable par un même individu dans sa globalité. La maîtrise de cette complexité est le résultat d'un découpage hiérarchique aboutissant à des étapes ayant une cohérence soit logique (fonction) soit physique (composant). La figure 1.1 représente les trois premiers niveaux de cette hiérarchisation.




(Transistor 1) (Porte logique 10) (Opérateur 1000)

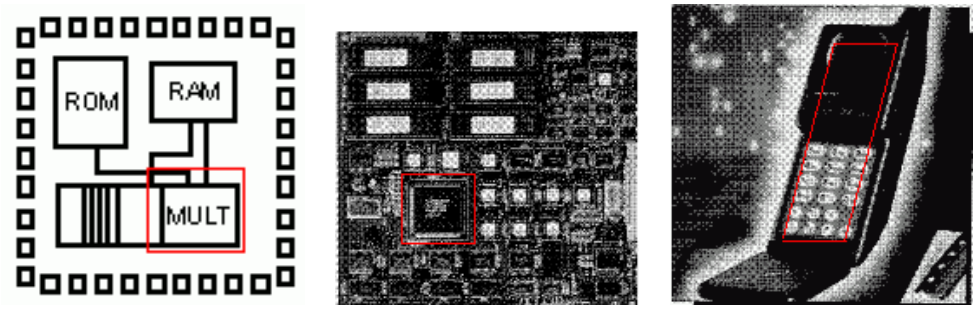
**Figure 1.1:** *Complexité des niveaux hiérarchiques.*

L'assemblage judicieux de moins d'une dizaine de transistors permet la réalisation des briques de base du traitement logique : les *portes logiques* à quelques entrées. Avec le même nombre de transistors, nous pouvons stocker une information binaire (0/1) dans un *point mémoire* et l'y maintenir tant que nous fournissons de l'énergie.

Il est possible par l'assemblage de quelques milliers de portes logiques et de fonctions de mémorisation de créer des *opérateurs* de calcul ou de traitement tels que des multiplieurs ou des unités de contrôle.

L'assemblage d'opérateurs permet la création d'un nouveau composant : le *circuit intégré*. Les millions de transistors des circuits intégrés sont réalisés sur un unique carré de matériau semiconducteur (le silicium) de la taille d'un ongle. Parmi les exemples les plus connus de circuits intégrés citons le *microprocesseur* et la *mémoire dynamique (DRAM)* qui sont au cœur des micro-ordinateurs.

Nous quittons maintenant le domaine de la micro-électronique numérique pour passer à une vision macroscopique illustrée dans la figure 1.2.




(Circuit intégré $10^8$) (Micro-ordinateur $10^{10}$) (Système Télécom $10^{xx}$)

**Figure 1.2:** *Complexité des niveaux hiérarchiques.*

L'assemblage de circuits intégrés sur des cartes ou *circuits imprimés* de quelques centaines de $cm^2$ permet la réalisation de systèmes de traitements numériques autonomes tels qu'une calculatrice, un agenda électronique ou un ordinateur de bureau.

Enfin, ces systèmes électroniques numériques permettent de réaliser la plupart des fonctions de traitement des réseaux de *télécommunications*. Bien malin qui peut savoir combien de transistors ont participé à votre dernière conversation téléphonique...

La maîtrise parfaite d'un de ces niveaux de complexité dans ses aspects techniques, scientifiques ou économiques, nécessiterait, à elle seule, une formation d'ingénieur électronicien... De façon réaliste, nos objectifs sont, dans le cadre d'une « base de connaissances indispensables », d'une part de vous permettre de comprendre le domaine de *l'électronique numérique intégrée* dans son ensemble et d'autre part, d'acquérir une première expérience concrète de la réalisation d'opérateurs numériques, ce que nous déclinons de manière plus précise par :

- compréhension de la technologie de fabrication et du fonctionnement des transistors ;
- maîtrise d'une technique de réalisation de portes logiques ;
- maîtrise de techniques de réalisation d'opérateurs ;
- compréhension de l'influence de la technologie sur les performances des circuits intégrés ;
- compréhension des enjeux techniques et économiques de l'industrie micro-électronique.

## 1.2 Signal et Information

Avant de nous intéresser au composant électronique il convient de définir son usage, c'est à dire de définir les caractéristiques des *signaux* qu'il est sensé générer, transmettre ou modifier.

### 1.2.1 Signal électrique et traitement de l'information

Le signal électrique est actuellement le support nécessaire à l'ensemble des systèmes de *traitement* de l'information qui nous sont familiers. Le mot information prend ici un sens très large que nous expliciterons ultérieurement. Votre télévision (information visuelle), votre chaîne HiFi (information musicale) ou votre micro-ordinateur (programmes...) sont des exemples concrets de systèmes de traitement de l'information utilisant le support électrique.

**Figure 1.3:** *Le signal électrique support de l'information...*

On peut envisager évidemment d'autres supports physiques que le signal électrique (papier, lumière, champs électromagnétiques) il n'en reste pas moins vrai que ces supports, s'ils sont très adaptés à la transmission et au stockage de l'information (livres, cédéroms, fibre optiques, téléphone portable ou par satellite), ne permettent guère de réaliser des fonctions de traitement élaborées.

**Figure 1.4:** *Le signal mécanique support de l'information...*

Nous pouvons mesurer les valeurs instantanées (tension, courant, charges...) d'un signal électrique quelconque ainsi que l'évolution de ces valeurs instantanées au cours du temps. Il est possible de caractériser un tel signal par des grandeurs de forme. La fréquence, la phase et l'amplitude sont, par exemple, trois paramètres caractérisant la forme d'un signal électrique sinusoïdal. Il est facile d'imaginer l'utilisation des variations de ces grandeurs ou paramètres pour représenter une information dont le signal électrique serait porteur, la restitution de l'information se faisant en interprétant la mesure de ces grandeurs.

Dans un article considéré comme fondateur de la théorie de l'information C.E. Shannon a proposé en 1948 un schéma de communication : l'information est définie comme un élément de connaissance de l'état d'un système. Une chaîne de communication comporte une source (ou émetteur) qui émet (*code*) des messages vers une destination (récepteur). Le récepteur ne peut décoder les informations émises par l'émetteur que s'il connaît l'ensemble des états possibles de la source. Le but de la théorie de l'information est de dégager les lois théoriques qui limitent les performances des systèmes de traitement et de communication. Elle permet également l'optimisation des codages en fonction des contraintes matérielles des systèmes.

**Figure 1.5:** *Claude Shannon*

### 1.2.2 Codage analogique de l'information : l'exemple du signal sonore

Jusqu'au lancement du « disque compact numérique », mis au point par les sociétés Sony et Philips au début des années 80, les technologies de stockage et de transfert du son appartenaient au domaine du traitement analogique (signal analogique à temps continu). En clair, depuis la première description du téléphone à ficelle (Robert Hooke en 1667) jusqu'à la fin des années 1970, ces techniques se basaient toutes sur la transformation d'un phénomène physique (par exemple une variation de pression) en un autre phénomène physique (par exemple vibration d'une membrane) se comportant de manière analogue au premier. Le signal électrique analogique sortant d'une tête de lecture d'un lecteur de disque microsillon est un exemple typique de ce codage analogique : la valeur instantanée de la tension à la sortie de la tête de lecture varie comme le signal audio enregistré mécaniquement sur le disque.

Le signal analogique électrique est malheureusement sujet à de nombreux phénomènes qui viennent détériorer la qualité de l'information transmise (atténuation, distortion, bruits parasites...), la correction de ces phénomènes n'est pas chose aisée et rarement satisfaisante. Ce n'est, de plus, pas un support très pratique dès que l'on envisage d'effectuer des traitements complexes. Les calculateurs analogiques ont eu une brève existence dans les années 60 à 80 lorsque leurs homologues numériques étaient à leurs balbutiements.

N'oublions cependant pas, en reprenant l'exemple de la chaîne de transmission du son, que le capteur (microphone) de même que l'enceinte acoustique ont un fonctionnement analogique ; le traitement analogique de l'information reste l'indispensable interface avec le monde « réel ».

### 1.2.3 Codage numérique de l'information

Maintenant, revenons à la théorie de l'information et reprenons l'exemple du signal sonore. Nous ne désirons transmettre et stocker que l'information pertinente à notre oreille. Comme le montre le diagramme de la figure 1.6, nous savons que celle-ci ne peut pas distinguer de variations de pression inférieures à $2 \times 10^{-5}$ Pa et qu'elle ne peut supporter de variations de pression supérieures à 20 Pa. On appelle ce rapport la dynamique du signal que l'on a coutume de mesurer en décibels :

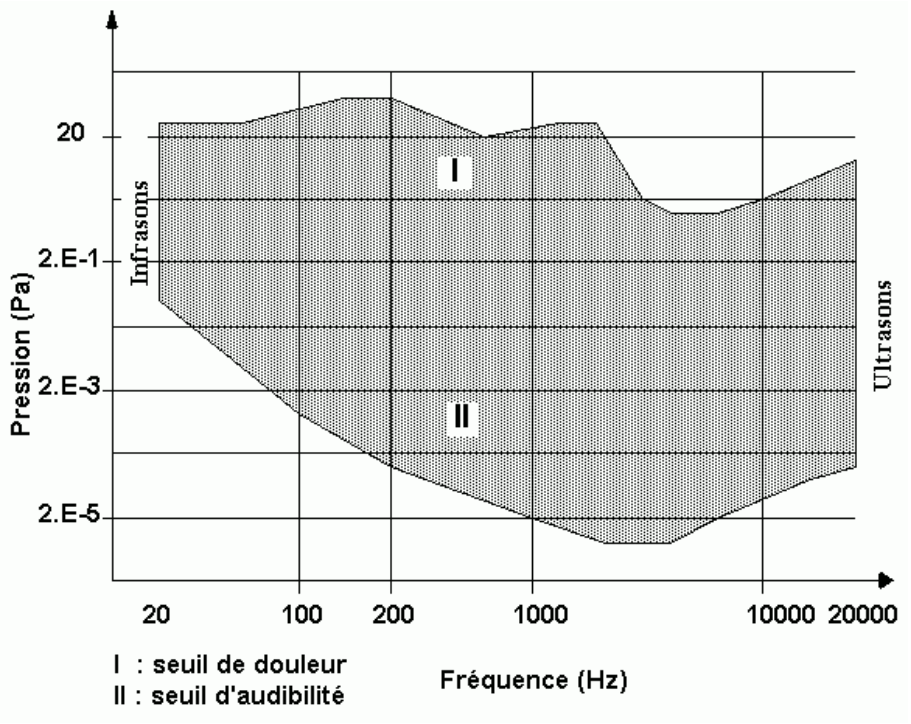



**Figure 1.6:** *L'audition moyenne d'un être humain*

$$\text{I décibels (dB)} = 20 \times \log_{10}(P/P_0)$$

Notre source audio nécessite une dynamique de 120 dB. Cela signifie aussi (au sens de la théorie de l'information) que le récepteur (l'oreille) ne peut décoder que $10^6$ états différents par pas de $2 \times 10^{-5}$ Pa. D'autre part, nous ne pouvons entendre que des signaux ayant une bande de fréquence limitée de 20 Hz à 20 000 Hz. Le *théorème d'échantillonnage* indique qu'il est possible de reconstruire avec exactitude un signal à bande limitée à partir d'échantillons de ce signal pris à intervalles réguliers à une fréquence double de la fréquence maximale du signal original. Nous avons donc montré que notre signal sonore pouvait être représenté par une suite de nombres entiers : c'est un signal prenant un nombre discret d'états de manière discrète dans le temps. Nous qualifierons ce signal de *signal numérique*. L'opération de quantification (discrétisation des niveaux) et d'échantillonnage (discrétisation du temps) nous conduit à redéfinir la notion de dynamique que l'on mesurera en bits (pour « binary digit ») :

$$\text{N bits} = \log_2(\text{nombre de niveaux})$$

Nous pouvons maintenant associer au signal numérique un *débit* de données mesuré en bits par seconde. Un lecteur de disque compact audio a, par exemple, un débit correspondant à 2 canaux (stéréophonie) échantillonnés à 44,1 kHz sur une dynamique de 16 bits soit 1,41 Mb/sec.

Nous allons donner un support électrique à ce signal numérique. L'idée la plus simple consiste à associer à chaque niveau du signal une tension et de faire évoluer dans le temps cette tension pour représenter la suite de symboles. La figure 1.7 montre un signal électrique *multivalué* support d'un signal numérique à trois niveaux prenant successivement les états

« 2 », « 1 » et « 0 ». Les plages hachurées représentent des plages de tensions pour lesquelles il n'est pas possible de déterminer l'état du signal ce qui est matérialisé par le symbole « X ».

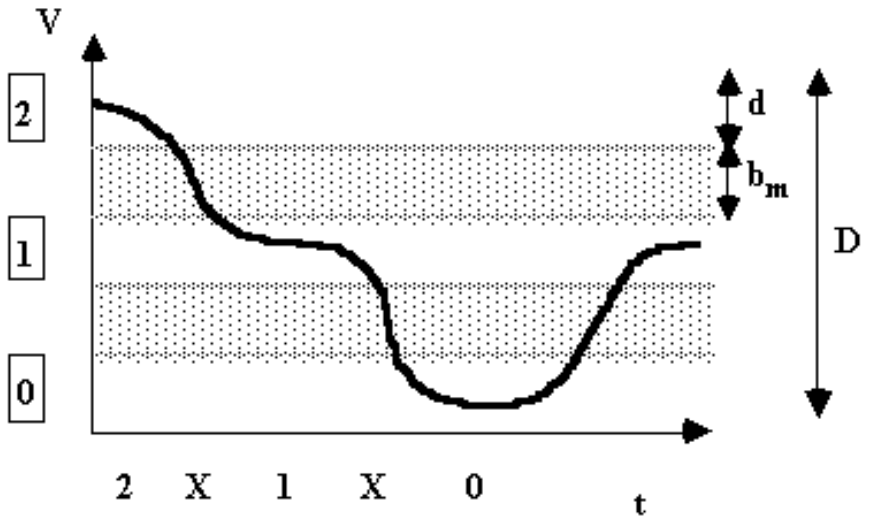



**Figure 1.7:** *Un signal multivalué codant une valeur numérique*

Remarquons (voir figure 1.8), qu'à débit d'information constant nous pouvons jouer sur le nombre de signaux physiques (ou de fils de liaison) utilisés, le nombre de niveaux codés et la fréquence de changement des symboles. Ce choix est essentiellement guidé par des consi-

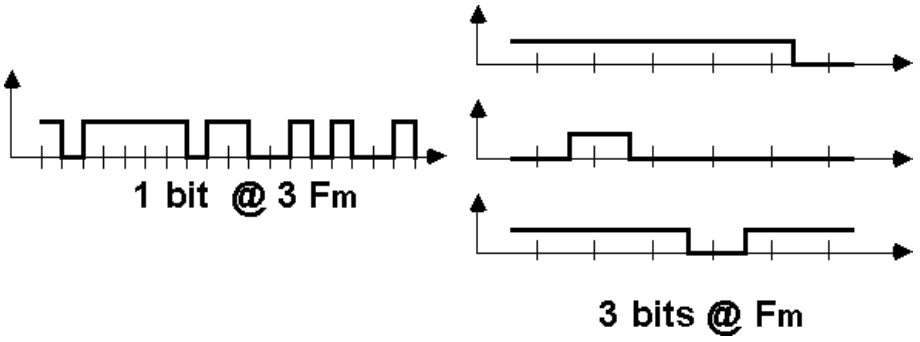



**Figure 1.8:** *Une liaison à la fréquence* $3 \cdot F_m$ *est équivalente à 3 liaisons opérants à la fréquence* $F_m$

dérations de facilité de traitement et sur la robustesse du codage au regard du bruit ou de l'atténuation du signal électrique. Sans entrer dans le détail de ces considérations, il est possible de justifier l'usage généralisé du codage binaire de la manière suivante.

Considérons la représentation de nombres sous la forme de $n$ canaux physiques ($n$ fils) utilisant $b$ niveaux électriques. Le nombre total de symboles représentables sous cette forme est $Q = b^n$. La réalisation de cette représentation a un coût matériel ; il faut en effet mettre en place des dispositifs de détection de niveaux plus ou moins complexes en fonction de la base choisie et adapter le nombre de canaux pour obtenir la qualité $Q$ choisie. Ce coût est évidemment proportionnel au nombre $n$ de canaux ; nous allons en première approximation considérer qu'il est aussi proportionnel à la base $b$ choisie : $C = b \times n$.

Nous pouvons en déduire la base optimale qui minimise le coût $C$ pour une qualité $Q$ donnée. Nous avons

$$n = \ln(Q)/\ln(b)$$

D'où

$$C = b \times \ln(Q)/\ln(b)$$

L'expression de $C$ possède un minimum en $b = e$ (base des logarithmes népériens) quelle que soit la qualité $Q$ souhaitée. Cela nous conduit à ne considérer que les bases « 2 » ou « 3 » comme candidates possibles. Dans la réalité, la complexité d'un opérateur de calcul physique en base « 3 » est plus que 1,5 fois plus grande que celle d'un opérateur en base « 2 » (la fonction $C$ dépend de $b$ de façon grandement non linéaire). Cela justifie le choix quasi universel de la base « 2 » dans les systèmes de traitement numérique. Il arrive que ce choix soit remis en cause dans des cas particuliers.

### 1.2.4 L'exemple du signal binaire

Dans un signal binaire, seuls deux états sont possibles : 0/1. Nous choisissons deux plages de tensions disjointes pour représenter les deux états, un symbole ne pouvant être à la fois dans l'état « 1 » et l'état « 0 ». Lorsque le signal électrique évolue dans le temps, il passe successivement et continuement d'une plage de définition d'état à l'autre en croisant une plage intermédiaire pour laquelle on qualifie le signal d'*indéterminé*. Une vision simplifiée du signal consiste à ne représenter que les états détectés « 0 » et « 1 » reliés par des segments de droite représentant leur *transition* d'un état à un autre (figure 1.9).

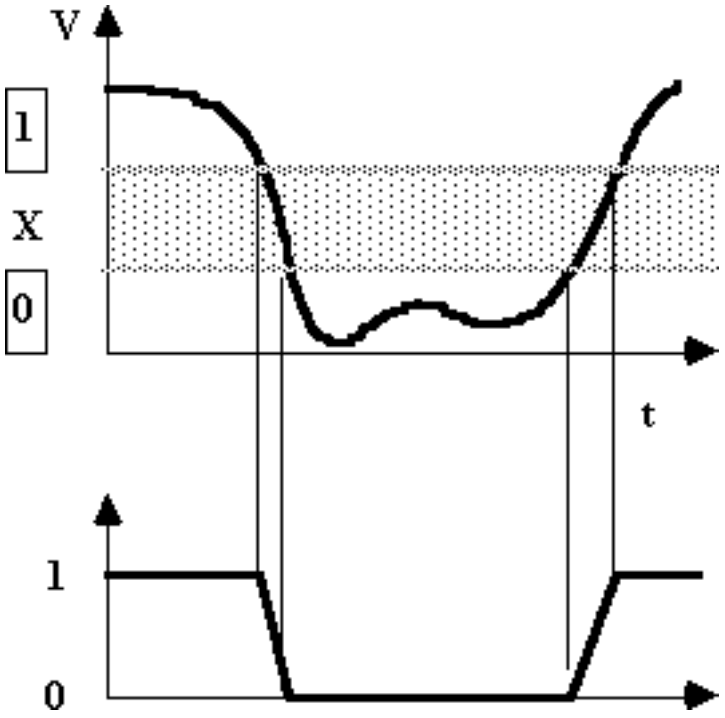



**Figure 1.9:** *Un signal binaire : signal électrique et interprétation.*

Même si le signal électrique subit des distortions, une atténuation, ou s'il est entaché de bruit, il est possible de reconstruire avec exactitude les symboles émis jusqu'à un certain niveau

de dégradation (figure 1.10). Remarquons que pour une excursion totale de tension donnée, la multiplication des états possibles (codage multivalué) diminue l'amplitude des plages de tensions associées à chaque état et donc augmente la difficulté de détection ; le codage binaire est à nouveau , de ce point de vue, le codage le plus robuste.

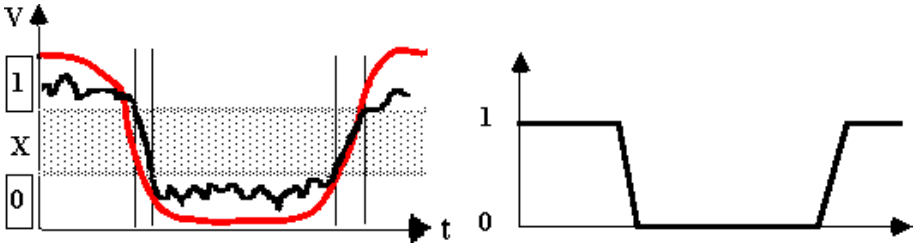



**Figure 1.10:** *Un signal binaire distordu, atténué et bruité, mais reconstruit.*

Pour conclure sur le signal binaire, nous pouvons lister les quelques caractéristiques qui ont conduit à l'imposer dans le monde de l'électronique numérique :

- C'est d'une part le symbole du raisonnement logique, de la prise de décision et du contrôle (si /alors/ sinon).
- C'est d'autre part une base de représentation des nombres entiers permettant d'effectuer tous calculs arithmétiques à partir d'opérations simples.
- Il est, comme tout signal numérique, utilisable pour coder de l'information comme le son ou l'image.
- Il peut utiliser un support électrique très simple (codage direct en amplitude).

## 1.3 Le signal binaire représenté par une grandeur électrique

Nous allons, dans ce chapitre, montrer quelques techniques simples permettant de dégager les caractéristiques des composants nécessaires à la génération, la détection et le traitement de signaux binaires électriques ; notre but n'est pas de présenter des implantations réalistes de fonction logiques.

### 1.3.1 Comment peut-on générer un signal électrique binaire ?

Il s'agit là de créer un signal électrique pouvant se stabiliser dans deux plages de tension prédéfinies correspondant aux deux états « 0 » et « 1 ». Nous pouvons partir d'une source d'alimentation continue fournissant à ses bornes une tension $V_{dd}$ donnée. Le montage de la figure 1.11 basé sur un simple interrupteur et une résistance permet de générer la tension « 0 V »(resp. $V_{dd}$) lorsque l'interrupteur est fermé (resp. ouvert) sur la sortie.

### 1.3.2 Comment peut-on extraire un symbole binaire d'un signal électrique ?

Supposons que nous disposons d'un composant, que nous appellerons « transistor », composé d'un premier dispositif capable de comparer une tension à une référence donnée, couplé à un second dispositif se comportant comme un interrupteur commandé en fonction du résultat de la comparaison. Le symbole et la fonction de cet interrupteur sont représentés en figure 1.12.

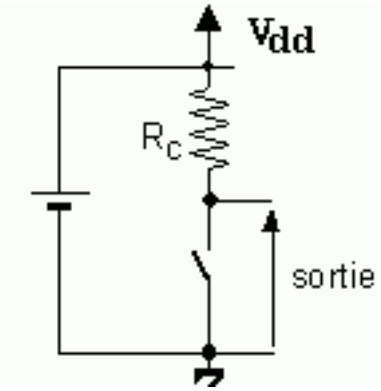
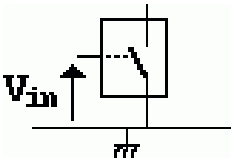


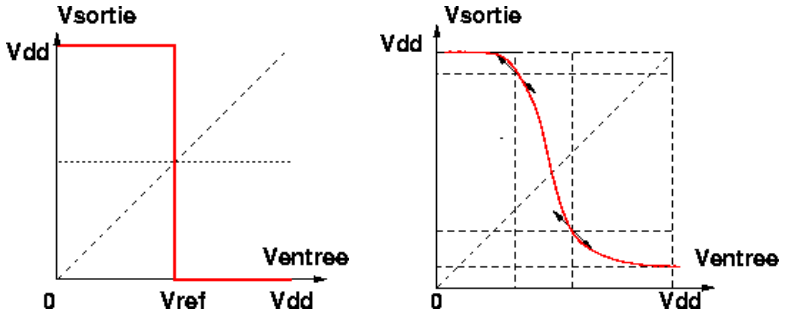

**Figure 1.11:** *Génération d'un signal binaire avec une source de tension, une résistance et un interrupteur.*




- Si $V_{in} < V_{ref}$ alors l'interrupteur est ouvert.
- Si $V_{in} > V_{ref}$ alors l'interrupteur est fermé.

**Figure 1.12:** *Le transistor interrupteur.*

Remplaçons l'interrupteur de la section précédente par notre transistor. Pour toute tension d'entrée inférieure (resp. supérieure) à $V_{ref}$, la tension de sortie prend la valeur $V_{dd}$ (resp. « 0 V »). Nous disposons bien d'un dispositif capable de détecter l'état du signal d'entrée même entaché de bruit ou de distortions. La figure 1.13 présente tout d'abord la *fonction de transfert* théorique du dispositif, c'est-à-dire la relation liant la tension d'entrée à la tension de sortie. Ce dispositif est appelé « inverseur ».




Fonction de transfert idéale Fonction de transfert réaliste

**Figure 1.13:** *Fonctions de transfert de l'inverseur.*

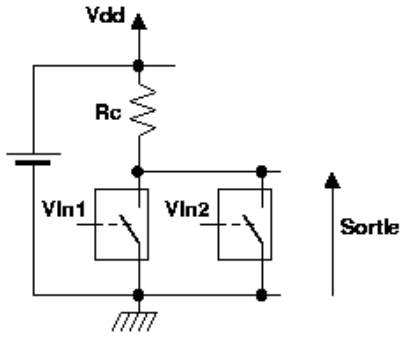
Dans la pratique, il n'est pas possible de créer des dispositifs électroniques aussi sélectifs ; nous pouvons retenir les deux imperfections suivantes :

- Passage « continu » entre le mode ouvert et le mode fermé ;
- Résistance non nulle en mode fermé.

La deuxième fonction de transfert de la figure 1.13 présente un comportement plus « réaliste » d'un tel opérateur. Tant que le signal reste dans les plages où la pente de la fonction de transfert est faible (en valeur absolue, c'est-à-dire ici supérieure à $-1$), l'immunité au bruit est maximale et le signal est régénéré de façon convenable.

### 1.3.3 Comment peut-on créer un opérateur de traitement binaire ?

Considérons maintenant le montage de la figure 1.14, composé de deux transistors et une résistance. Nous pouvons construire une table représentant la valeur de la tension en sortie du montage en fonction des tensions en entrée. Nous pouvons traduire cette table en une *table de vérité* en remplaçant les valeurs de tension par les états « 0 » ou « 1 » correspondants. Nous avons créé un opérateur de traitement binaire (la fonction « non-ou ») qui prend la valeur « 0 » en sortie si l'une ou l'autre des entrées est à « 1 ».




| $V_{in1}$ | $V_{in2}$ | $V_{sortie}$ |
|---|---|---|
| $<V_{ref}$ | $<V_{ref}$ | $V_{dd}$ |
| $<V_{ref}$ | $>V_{ref}$ | « 0 V » |
| $>V_{ref}$ | $<V_{ref}$ | « 0 V » |
| $>V_{ref}$ | $>V_{ref}$ | « 0 V » |

| In1 | In2 | Sortie |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**Figure 1.14:** *Fonction NOR2 : schéma et table de vérité*

Nous pouvons évidemment élaborer des fonctions plus complexes soit par construction soit par combinaisons de différentes fonctions déjà créées. La fonction « ou » peut se construire, par exemple, en connectant une fonction « inverseur » derrière la fonction « non-ou ». Une autre fonction importante du traitement numérique est la mémorisation des informations. La figure 1.15 montre comment au moyen de deux inverseurs connectés l'un à l'autre il est possible de créer un dispositif possédant deux états stables que l'on assimilera au stockage d'une information binaire. Comme indiqué dans la représentation des fonctions de transfert des deux inverseurs le couple de tensions ($V_a$, $V_b$) ne peut prendre que les valeurs ($V_{dd}$, $V_{min}$) ou ($V_{min}$, $V_{dd}$) et ce tant que le dispositif est alimenté par la source de tension $V_{dd}$. Nous ne traiterons pas dans ce cours de la manière de forcer cette mémoire à prendre un état désiré 0 ou 1.

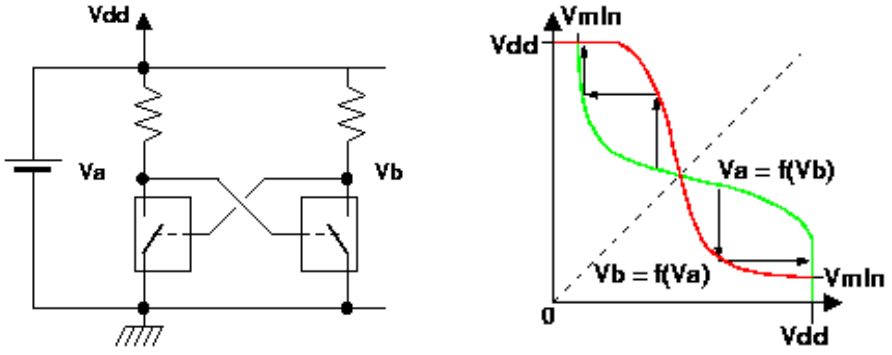



**Figure 1.15:** *Fonction mémorisation : schéma et fonctions de transfert*

## 1.4 La technologie micro-électronique

Le rôle de la technologie micro-électronique est la réalisation et l'intégration des transistors nécessaires à la réalisation des opérateurs dont nous avons vu un premier aperçu. Il se trouve que le traitement ou le stockage de données manipule une matière première sans dimensions physiques : l'information. Par conséquent, le dispositif de traitement ou de stockage peut être aussi petit que l'on peut le souhaiter dans la limite de nos capacités ou de nos connaissances scientifiques à une période donnée. Toutes les bases techniques de la fabrication des circuits intégrés électroniques ont été établies vers 1960 ; depuis les progrès ne sont que des améliorations successives sans remise en question fondamentale du procédé initial.

### 1.4.1 Quelles propriétés des matériaux peut-on exploiter pour créer des transistors ?

Nous désirons créer un dispositif passant d'un mode isolant (interrupteur ouvert) à un mode conducteur (interrupteur fermé) en fonction d'une commande électrique extérieure. Un relai électro-mécanique ferait l'affaire, mais il semble plus judicieux de chercher à exploiter des phénomènes physiques qui permettraient de modifier directement les caractéristiques conductrices d'un matériau. Les matériaux *semi-conducteurs* (silicium, germanium, arséniure de gallium...) sont des matériaux rêvés pour cet usage. Ces matériaux ont les deux propriétés fondamentales suivantes :

- Il est possible de modifier « statiquement » les densités de charges libres, et donc susceptibles de créer un courant électrique, dans le matériau en injectant des impuretés (bore, arsenic, phosphore...) dans leur structure : on appelle cela le *dopage*.
- Il est possible de modifier « dynamiquement » les densités de charges libres dans le matériau sous l'influence de champs électriques.

En combinant ces deux phénomènes, nous pouvons créer des transistors ayant le comportement demandé. Le transistor le plus couramment utilisé actuellement est le transistor MOS (pour Métal/Oxyde/Semi-conducteur). La figure 1.16 présente une vue en coupe et en perspective d'un transistor MOS de type N (vous verrez en électronique analogique qu'il existe deux types de transistors MOS, les N et les P).

Les éléments essentiels constituant un transistor MOS de type N sont les suivants :

- Un *substrat* faiblement dopé avec des dopants de type $P$ (atomes de dopants accepteurs d'électrons). On note $P^-$ce type de dopage. Polarisé correctement, ce substrat est isolant.
- Au sein de ce substrat, deux zones approximativement parallélépipédiques, fortement dopées avec des dopants de type $N$ (atome de dopants donneurs d'électrons). On note $N^+$ce type de dopage. Ces zones dopées sont nommées *Source* et *Drain* du transistor. Notons que Source et Drain sont indiscernables : le transistor est physiquement symétrique.
- La zone de substrat située entre Source et Drain se nomme le *canal* du transistor. La longueur L et la largeur W (de l'anglais Width) du canal étaient dans les technologies courantes en 2004 de l'ordre du dixième de micron. En 2014, c'est le centième de micron. En jouant sur ces deux dimensions le concepteur peut contrôler les performances du transistor.
- Au dessus du canal, une fine couche isolante, constituée d'oxyde de silicium ($SiO_2$).

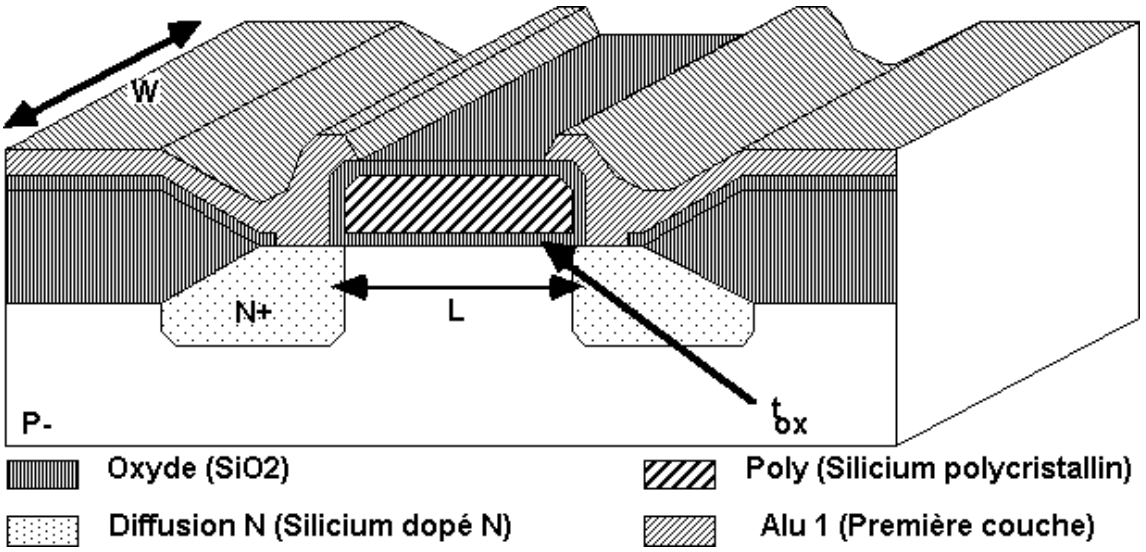



**Figure 1.16:** *Vue en coupe d'un transistor NMOS*

L'épaisseur de cette couche est actuellement de moins de 10 nm.

- Enfin, au dessus de cette couche d'oxyde, un dépôt de silicium poly-cristallin, aligné avec la canal du transistor. Il s'agit de la *Grille* du transistor. Le silicium poly-cristallin est un agglomérat de petits cristaux de silicium (c'est un matériau conducteur).
- La perspective nous montre, de plus, les connecteurs métalliques permettant de « raccorder » Source et Drain du transistor au reste du circuit.

Dans le transistor MOS, le champ électrique créé en polarisant convenablement la Grille permet de moduler le courant passant entre Drain et Source.

### 1.4.2 Quelles sont les différentes étapes de la fabrication des circuits intégrés ?

La fabrication d'un circuit intégré doit satisfaire à trois besoins :

- créer des transistors les plus performants possibles ;
- offrir les moyens d'interconnecter entre eux ces transistors, ainsi que d'interconnecter le circuit intégré avec le monde macroscopique ;
- offrir une protection, contre les agressions liées aux conditions d'utilisation, suffisante pour garantir une fiabilité satisfaisante.

Le matériau de base est le silicium, l'élément le plus commun sur Terre après l'oxygène. Ce matériau est purifié et transformé en un *lingot* mono-cristallin de quelques dizaines de cm de diamètre. Le silicium est purifié pour comporter moins d'une impureté pour 100 millions d'atomes de silicium, en effet le niveau des dopages destinés à ajuster les propriétés semi-conductrices du matériau sont de l'ordre de 10 atomes de dopant pour un million d'atomes de silicium. Le lingot est découpé en *tranches* (wafer pour les anglo-saxons) de faible épaisseur. Ces tranches sont polies jusqu'à ce que les défauts de surface n'excèdent pas quelques couches atomiques. Les tranches sont ensuite envoyées en fonderie pour la fabrication proprement dite des circuits intégrés. On fabrique ainsi en parallèle plusieurs dizaines de circuits sur la même tranche.

Les traitements effectués sur les tranches se résument à quelques étapes simples plusieurs fois répétées :

- *Croissances* ou dépôts de silice sur la surface de tranche : il s'agit de réaliser des isolations entre éléments de différentes couches ou des grilles de transistors.
- *Lithographie* : il s'agit de dessiner les motifs désirés dans le matériaux. Cela commence par le dépôt d'une résine photo-sensible sur la surface de la tranche. Après exposition à travers un masque et développement la résine est éliminée des endroits désirés. La résine restante servira de protection pour une attaque chimique sélective de la tranche.
- *Implantation ionique* : il s'agit de réaliser les dopages nécessaires au fonctionnement des transistors. La silice au préalablement gravée par lithographie sert de masque naturel pour définir les zones où seront les transistors.
- *Dépôts* de métaux : il s'agit là de déposer uniformément sur la tranche une couche de métal qui servira à réaliser des connections entre transistors.

La dernière étape de traitement de la tranche consiste à noyer les circuits dans un épais matériau de protection, sauf aux endroits où l'on voudra souder des fils les reliant au monde extérieur. Après différents test sur les tranches, les circuits sont découpés et, après être de nouveau testés, montés dans un boîtier comme indiqué figure 1.17.

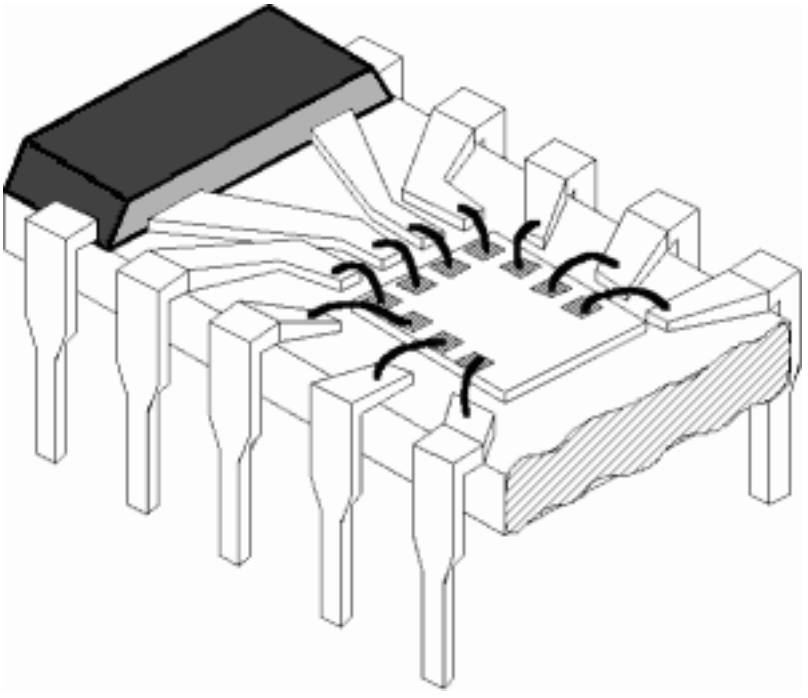


**Figure 1.17:** *Encapsulation d'un circuit intégré dans un boîtier.*

## 1.5 Les filières technico-économiques

### 1.5.1 La recherche d'un optimum de rendement dans l'utilisation de la technologie

Pour finir nous abordons, dans ce chapitre, différentes applications qui sont faites de la technologie micro-électronique et qui conduisent pour des raisons techniques et économiques

à différentes filières de réalisation des systèmes électroniques numériques. Nous ne nous intéressons ici qu'à la fonction « traitement », l'industrie des mémoires (fonction « stockage ») n'étant pas abordée.

### 1.5.2 Les circuits « universels » tels les microprocesseurs

Pour minimiser l'impact du coût de conception et de fabrication des circuits intégrés les plus complexes, il est intéressant de leur donner une gamme d'applications permettant de s'adresser une clientèle la plus large possible. Dans cet esprit, il s'agit de créer un circuit de traitement numérique dont l'usage final (l'application) n'est pas connue à la fabrication. Pour cela, il suffit de réaliser un circuit intégré ayant quelques ressources de traitement assez génériques (addition de deux nombres, stockage d'un nombre en mémoire, lecture d'un nombre d'une mémoire...) associées à un dispositif de contrôle capable d'interpréter des ordres simples qui sont stockés dans une mémoire extérieure au circuit. Il suffit de changer le contenu de cette mémoire (le programme) pour changer l'enchaînement des traitements effectués par le circuit, donc l'application. Avec de tels circuits l'augmentation de complexité des applications est gérée simplement par l'augmentation de la taille des programmes.

### 1.5.3 Les circuits spécifiques à une application

Considérons maintenant une application très spécifique, faiblement complexe mais nécessitant beaucoup de puissance de calcul. Un microprocesseur peut ne pas suffire à fournir la puissance de calcul nécessaire. On peut tenter d'utiliser plusieurs microprocesseurs, mais la gestion des échanges est malaisée et le coût du système risque de devenir rapidement prohibitif. Une alternative au microprocesseur est le câblage direct des applications sur le silicium. Les circuits intégrés réalisés de cette façon s'appellent des circuits intégrés spécifiques à une application (ASICs pour les anglo-saxons). Par exemple, une opération cruciale en télévision numérique, l'estimation de mouvement est réalisée actuellement par un seul circuit ASIC capable de calculer plus de dix milliards d'additions par seconde, ce qui dépasse de loin les performances des meilleurs microprocesseurs. La contrepartie à cette performance est que ce circuit ne peut servir qu'à la télévision numérique...

### 1.5.4 Les circuits logiques programmables

Les circuits logiques programmables (CLP) visent à un compromis entre les avantages des ASIC et des microprocesseurs. Il s'agit comme pour un ASIC de viser des applications relativement simples mais demandant de fortes puissances de calcul tout en conservant la souplesse de la programmation. Pour arriver à cela, ces circuits sont composés de milliers de fonctions logiques dont les équations sont stockées sous forme de tables de vérité dans des mémoires internes au circuit. Des centaines de milliers de fils de connexions parcourent le circuit en tous sens et sont potentiellement connectables aux fonctions logiques via des transistors servant d'interrupteurs. Les états de ces interrupteurs sont à leur tour stockés dans des mémoires internes au circuit. En résumé, ces circuits peuvent changer de fonction et de câblage par simple modification du contenu de mémoires. Cette souplesse est évidemment très avantageuse, très utilisée pour la réalisation de prototypes ou de petites séries. Les ASIC remplacent cependant systématiquement les CLP dès qu'il s'agit de produire en masse, notamment pour des questions de rendement d'utilisation du silicium.

### 1.5.5 Les systèmes intégrés sur puces

Petits derniers des évolutions de l'industrie micro-électronique, les systèmes sur puces (ou *SOC* pour « System On Chip ») tirent avantage des taux d'intégration faramineux atteints ces dernières années (plus de 400 000 portes logiques par $mm^2$ de silicium) pour intégrer sur une seule puce de Silicium toutes les fonctionnalités « logicielles » et « matérielles » nécessaires à la réalisation de systèmes de traitements totalement autonomes. Ces puces intègrent non seulement des fonctionnalités communes aux trois variantes précédemment présentées mais aussi des capteurs et éventuellement dans un futur proche des éléments mécaniques (moteurs, pompes...).

## 1.6 Bibliographie

Le site Web `http://jas2.eng.buffalo.edu/applets` du professeur Chu Ryang WIE de l'université de Buffalo (état de New-York) permet d'exécuter quelques démonstrations interactives sur l'usage des matériaux-conducteurs, sur le fonctionnement des transistors ainsi que sur le fonctionnement de quelques montages de bases de l'électronique. Les liens suivants sont particulièrement en rapport avec ce chapitre :

- n-channel MOSFET, both side-view and top-view and full photoresist steps ;
- CMOS Inverter, side-view, device fabrication steps ;
- Fabrication @ various companies.

# Chapitre 2

# Fonctions combinatoires

## 2.1 Introduction

Nous avons évoqué, dans le chapitre 1, la possibilité de réaliser physiquement des fonctions de calcul utilisant une représentation binaire des données. Avant de poursuivre plus avant l'étude de la réalisation physique de ces fonctions, nous allons développer les bases mathématiques des fonctions logiques (Algèbre de Boole) ainsi que les méthodes de représentation et de manipulation associées.

Le choix d'une structure physique optimale pour construire une fonction logique est une opération complexe dépendant de nombreuses contraintes telles que l'optimisation de la vitesse de traitement, la minimisation de l'énergie dissipée par opération ou tout simplement le coût de fabrication.

Nous nous contenterons dans ce chapitre d'envisager le critère suivant qui pourra être remis en cause dans la suite du cours :

- La construction de fonctions combinatoires complexes est basée sur l'utilisation d'une bibliothèque de fonctions logiques (ou portes) élémentaires telles que l'inversion, le « ou » logique, et le « et » logique.
- L'optimisation des fonctions complexes est basée sur la minimisation du nombre des portes élémentaires utilisées qui correspond à une simplification des équations booléennes associées.

## 2.2 Variables et fonctions logiques, tables de vérité

Considérons l'ensemble $E$ à 2 éléments $(0, 1)$.

1. Une **variable logique** est un élément de $E$ et ne possède ainsi que 2 valeurs 0 et 1. Elle est représentée par des lettres $(A, b, e, X, \cdots)$.
2. Une **fonction logique** de plusieurs variables applique $E \times E \times \cdots E$ dans $E$. Elle associe à un $n$-uplet de variables booléennes $(e_0, e_1, \cdots, e_{n-1})$ une valeur $F(e_0, e_1, \cdots, e_{n-1})$.
3. Il existe différentes manières d'exprimer une fonction booléenne. Une fonction de $n$ variables est entièrement décrite par l'énoncé des valeurs de cette fonction pour l'ensemble (ou le sous-ensemble de définition) des combinaisons du $n$-uplet de variables :

$$F(0, \cdots, 0, 0), F(0, \cdots, 0, 1), F(0, \cdots, 1, 0), \cdots, F(1, \cdots, 1, 1)$$

Cet énoncé prend généralement la forme d'un tableau à $n+1$ colonnes et au plus $2^n$ lignes, chaque ligne exposant une combinaison des variables et la valeur correspondante de la fonction. Les tableaux 2.1 et 2.2 suivants donnent la forme générale de **tables de vérité** de fonctions de trois variables totalement (fonction $F$) ou partiellement (fonction $G$) définies.

| $A$ | $B$ | $C$ | $F(A,B,C)$ |
|---|---|---|---|
| 0 | 0 | 0 | $F(0,0,0)$ |
| 0 | 0 | 1 | $F(0,0,1)$ |
| 0 | 1 | 0 | $F(0,1,0)$ |
| 0 | 1 | 1 | $F(0,1,1)$ |
| 1 | 0 | 0 | $F(1,0,0)$ |
| 1 | 0 | 1 | $F(1,0,1)$ |
| 1 | 1 | 0 | $F(1,1,0)$ |
| 1 | 1 | 1 | $F(1,1,1)$ |

**Table 2.1:** *Table de vérité d'une fonction de 3 variables.*

| $A$ | $B$ | $C$ | $G(A,B,C)$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | non définie |
| 1 | 0 | 0 | non définie |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

**Table 2.2:** *Table de vérité d'une fonction partiellement définie.*

## 2.3 Représentations des fonctions logiques

### 2.3.1 Formes algébriques

Nous associons, à l'ensemble $E$, l'algèbre de Boole basée sur trois opérateurs logiques :

- Opérateur NON : réalise la complémentation (ou inversion) représentée ici par une barre horizontale : « $\overline{x}$ ».
- Opérateur OU : réalise l'union (ou addition logique) notée ici : « $+$ ».
- Opérateur ET : réalise l'intersection (ou multiplication logique) notée ici : « $\cdot$ ».

| $A$ | $\overline{A}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Table 2.3:** *Opérateur NON.*

| $A$ | $B$ | $A+B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Table 2.4:** *Opérateur OU.*

| $A$ | $B$ | $A \cdot B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table 2.5:** *Opérateur ET.*

Les tables de vérité de ces trois fonctions logiques sont données dans les tableaux 2.3, 2.4 et 2.5.

Une fonction logique booléenne se présente comme une association des opérations algébriques précédentes sur un ensemble de variables. Elle peut s'écrire de plusieurs façons.

### 2.3.2 Forme disjonctive

Elle correspond à une somme de produits logiques : $F = \Sigma\Pi(e_i)$, où $e_i$ représente une variable ou son complément. Exemple :

$$F_1(X, Y, Z) = X \cdot Y + X \cdot \overline{Z} + \overline{X} \cdot \overline{Y} \cdot Z$$

Si chacun des produits contient toutes les variables d'entrée sous une forme directe ou complémentée, alors la forme est appelée « **première forme canonique** » ou « **forme canonique disjonctive** ». Chacun des produits est alors appelé **minterme**. Exemple de forme canonique disjonctive :

$$F_2(X, Y, Z) = \overline{X} \cdot \overline{Y} \cdot Z + \overline{X} \cdot Y \cdot Z + X \cdot \overline{Y} \cdot \overline{Z}$$

### 2.3.3 Forme conjonctive

Elle fait référence à un produit de sommes logiques : $F = \Pi\Sigma(e_i)$. Voici un exemple :

$$F_3(X, Y, Z) = (X + Y) \cdot (\overline{X} + Z) \cdot (\overline{X} + Y + \overline{Z})$$

Si chacune des sommes contient toutes les variables d'entrée sous une forme directe ou complémentée, alors la forme est appelée « **deuxième forme canonique** » ou « **forme canonique conjonctive** ». Chacune des sommes est alors appelée **maxterme**. Exemple de forme canonique conjonctive :

$$F_4(X, Y, Z) = (X + Y + Z) \cdot (\overline{X} + \overline{Y} + Z) \cdot (\overline{X} + Y + \overline{Z})$$

### 2.3.4 Équivalence entre la table de vérité et les formes canoniques

Nous avons défini la table de vérité d'une fonction comme la correspondance entre chaque combinaison des variables (du domaine de définition de la fonction) et la valeur (0 ou 1) associée de cette fonction.

Chacune des combinaisons des variables définit une **configuration** des entrées, on peut donc associer une configuration à chaque ligne d'une table de vérité.

### 2.3.5 Forme canonique disjonctive

Une fonction logique est représentée par l'ensemble des configurations pour lesquelles la fonction est égale à « 1 ».

Considérons maintenant une configuration des entrées pour laquelle une fonction booléenne vaut « 1 » : il existe un minterm unique prenant la valeur « 1 » dans cette configuration.

Il suffit donc d'effectuer la somme logique (ou réunion) des minterms associés aux configurations pour lesquelles la fonction vaut « 1 » pour établir l'expression canonique disjonctive de la fonction.

Exemple d'une fonction $H$ à trois variables entièrement définie :

| $A$ | $B$ | $C$ | $H(A, B, C)$ | **État** | **Minterme** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | $\overline{A} \cdot \overline{B} \cdot \overline{C}$ |
| 0 | 0 | 1 | 1 | 1 | $\overline{A} \cdot \overline{B} \cdot C$ |
| 0 | 1 | 0 | 0 | 2 | $\overline{A} \cdot B \cdot \overline{C}$ |
| 0 | 1 | 1 | 1 | 3 | $\overline{A} \cdot B \cdot C$ |
| 1 | 0 | 0 | 0 | 4 | $A \cdot \overline{B} \cdot \overline{C}$ |
| 1 | 0 | 1 | 1 | 5 | $A \cdot \overline{B} \cdot C$ |
| 1 | 1 | 0 | 0 | 6 | $A \cdot B \cdot \overline{C}$ |
| 1 | 1 | 1 | 0 | 7 | $A \cdot B \cdot C$ |

**Table 2.6:** *Table de vérité de la fonction $H$ : états associés et mintermes.*

On remarque que $H(A, B, C) = 1$ pour les états 0, 1, 3, 5. On écrit la fonction ainsi spécifiée sous une forme dite numérique : $H = \mathbf{R}(0, 1, 3, 5)$, **R**éunion des états 0, 1, 3, 5. La première forme canonique de la fonction $H$ s'en déduit directement :

$$H(A, B, C) = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C$$

### 2.3.6 Forme canonique conjonctive

Considérons maintenant une configuration des entrées pour laquelle la fonction vaut « 0 ».

Il existe un maxterm unique prenant la valeur « 0 » en cette configuration. Ce maxterm prend donc la valeur « 1 » dans toutes les autres configurations des entrées.

Il suffit donc d'effectuer le produit logique (ou intersection) des maxterms associés aux configurations pour lesquelles la fonction vaut « 0 » pour établir l'expression canonique conjonctive de la fonction.

Reprenons l'exemple de la fonction $H$ :

| $A$ | $B$ | $C$ | $H(A,B,C)$ | **Etat** | **Maxterme** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | $A+B+C$ |
| 0 | 0 | 1 | 1 | 1 | $A+B+\overline{C}$ |
| 0 | 1 | 0 | 0 | 2 | $A+\overline{B}+C$ |
| 0 | 1 | 1 | 1 | 3 | $A+\overline{B}+\overline{C}$ |
| 1 | 0 | 0 | 0 | 4 | $\overline{A}+B+C$ |
| 1 | 0 | 1 | 1 | 5 | $\overline{A}+B+\overline{C}$ |
| 1 | 1 | 0 | 0 | 6 | $\overline{A}+\overline{B}+C$ |
| 1 | 1 | 1 | 0 | 7 | $\overline{A}+\overline{B}+\overline{C}$ |

**Table 2.7:** *Table de vérité de la fonction $H$ : états associés et maxtermes.*

On remarque que $H(A,B,C) = 0$ pour les états 2, 4, 6, 7. On écrit la fonction ainsi spécifiée sous une forme dite numérique : $H = \mathbf{I}(2,4,6,7)$ **I**ntersection des états 2, 4, 6, 7.

La deuxième forme canonique de la fonction $H$ s'en déduit directement :

$$H(A,B,C) = (A+\overline{B}+C)\cdot(\overline{A}+B+C)\cdot(\overline{A}+\overline{B}+C)\cdot(\overline{A}+\overline{B}+\overline{C})$$

## 2.4 Description de méthodes de simplification

On cherche ici à obtenir une expression algébrique comportant un nombre minimal de termes, ainsi qu'un nombre minimal de variables dans chaque terme dans le but de simplifier la réalisation matérielle.

**Attention :** Comme nous l'avons indiqué en introduction, l'optimisation d'une fonction logique dépend de paramètres tels que la performance en vitesse désirée, la consommation maximale autorisée ou l'obligation d'utiliser des bibliothèques de fonctions élémentaires prédéfinies. La complexité de la représentation algébrique n'est donc qu'un critère d'optimisation parmi d'autres.

### 2.4.1 Utilisation des propriétés de l'algèbre de Boole

Les propriétés, lois et théorèmes fondamentaux de l'algèbre de Boole sont à notre disposition pour manipuler les équations.

**Attention !** Il n'y a pas de structure de groupe, ni pour ET, ni pour OU ! En effet, il n'y a pas d'élément opposé pour ces deux opérations. C'est avec le OU EXCLUSIF (XOR) qu'il y a une structure de groupe.

La structure d'anneau s'obtient en prenant le groupe obtenu avec le XOR et en ajoutant comme loi multiplicative, soit le ET, soit le OU.

| | | |
|---|---|---|
| **Complémentarité :** | $a + \overline{a} = 1\,, \qquad a \cdot \overline{a} = 0\,,$ | $\overline{\overline{a}} = a$ |
| **Idempotence :** | $a + a + a + \cdots = a\,,$ | $a \cdot a \cdot a \cdots = a$ |
| **Éléments neutres :** | $a + 0 = a\,,$ | $a \cdot 1 = a$ |
| **Élts absorbants :** | $a + 1 = 1\,,$ | $a \cdot 0 = 0$ |
| **Commutativité :** | $a + b = b + a\,,$ | $a \cdot b = b \cdot a$ |
| **Associativité :** | $(a + b) + c = a + (b + c) = a + b + c\,,$ | $(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$ |
| **Distributivité :** | $(a + b) \cdot c = (a \cdot c) + (b \cdot c)\,,$ | $(a \cdot b) + c = (a + c) \cdot (b + c)$ |
| **Th. d'absorption (1) :** | $a + (a \cdot b) = a\,,$ | $a \cdot (a + b) = a$ |
| **Th. d'absorption (2) :** | $a \cdot \overline{b} + b = a + b\,,$ | $(a + \overline{b}) \cdot b = a \cdot b$ |
| **Th. d'adjacence :** | $(a + \overline{b}) \cdot (a + b) = a\,,$ | $a \cdot \overline{b} + a \cdot b = a$ |

**Remarque** : Deux termes sont dits **adjacents** logiquement s'ils ne diffèrent que par une variable.

**Théorème de De Morgan** :

$$\overline{a + b} = \overline{a} \cdot \overline{b}\,, \qquad \overline{a \cdot b} = \overline{a} + \overline{b}$$

**Premier théorème d'expansion** :

$$F(e_0, e_1, \cdots, e_i, \cdots, e_{n-1}) = e_i \cdot F(e_0, e_1, \cdots, 1, \cdots, e_{n-1}) + \overline{e_i} \cdot F(e_0, e_1, \cdots, 0, \cdots, e_{n-1})$$

**Second théorème d'expansion** :

$$F(e_0, e_1, \cdots, e_i, \cdots, e_{n-1}) = [e_i + F(e_0, e_1, \cdots, 0, \cdots, e_{n-1})] \cdot [\overline{e_i} + F(e_0, e_1, \cdots, 1, \cdots, e_{n-1})]$$

### 2.4.2 Simplification à partir de la forme algébrique

Les méthodes algébriques employées se rapportent aux relations fondamentales d'absorption, d'adjacence, de mise en facteur et aux théorèmes de De Morgan. On distingue plusieurs procédés permettant d'aboutir au but recherché :

**Regroupement des termes et mises en facteur**

$$\begin{aligned} Z &= \overline{a} \cdot \overline{c} \cdot d + \overline{a} \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot c \cdot \overline{d} = \overline{a} \cdot \overline{c} \cdot (d + \overline{d}) + \overline{a} \cdot b \cdot c \cdot \overline{d} \\ &= \overline{a} \cdot \overline{c} + \overline{a} \cdot b \cdot c \cdot \overline{d} = \overline{a} \cdot (\overline{c} + c \cdot b \cdot \overline{d}) = \overline{a} \cdot (\overline{c} + b \cdot \overline{d}) \end{aligned}$$

Nous avons successivement utilisé une mise en facteur, la complémentarité, une deuxième mise en facteur et enfin le théorème d'absorption.

**Réplication de termes existants**

$$\begin{aligned} Z &= \overline{a}\cdot b\cdot c + a\cdot \overline{b}\cdot c + a\cdot b\cdot \overline{c} + a\cdot b\cdot c \\ &= \overline{a}\cdot b\cdot c + a\cdot b\cdot c + a\cdot \overline{b}\cdot c + a\cdot b\cdot c + a\cdot b\cdot \overline{c} + a\cdot b\cdot c \\ &= (\overline{a}+a)\cdot b\cdot c + (\overline{b}+b)\cdot a\cdot c + (\overline{c}+c)\cdot a\cdot b \\ &= b\cdot c + a\cdot c + a\cdot b \end{aligned}$$

La réplication du terme $a\cdot b\cdot c$ permet de simplifier chacun des trois premiers termes en utilisant une mise en facteur et la complémentarité.

**Suppression de termes superflus**

$$\begin{aligned} Z &= \overline{a}\cdot\overline{b} + b\cdot\overline{c} + \overline{a}\cdot\overline{c} = \overline{a}\cdot\overline{b} + b\cdot\overline{c} + \overline{a}\cdot\overline{c}\cdot(b+\overline{b}) \\ &= \overline{a}\cdot\overline{b} + \overline{a}\cdot\overline{b}\cdot\overline{c} + b\cdot\overline{c} + \overline{a}\cdot b\cdot\overline{c} = \overline{a}\cdot\overline{b}\cdot(1+\overline{c}) + b\cdot\overline{c}\cdot(1+\overline{a}) \\ &= \overline{a}\cdot\overline{b} + b\cdot\overline{c} \end{aligned}$$

Nous avons ici réintroduit la variable $b$ dans le troisième terme par l'intermédiaire de la propriété de complémentarité, nous avons ensuite utilisé la propriété d'absorption pour simplifier les produits.

**Simplification par utilisation des formes canoniques**

Si l'on dispose de la table de vérité de la fonction, on prend pour équation algébrique de départ la forme canonique comportant le minimum de termes. Cette équation sera ensuite simplifiée en utilisant les méthodes décrites précédemment. En effet, pour une fonction à $N$ entrées, la forme canonique disjonctive comportera $P$ mintermes (avec $P \leq 2^N$), alors que la forme conjonctive comportera $2^N - P$ maxtermes.

### 2.4.3 Méthode des tables de Karnaugh

Les tables de Karnaugh sont des représentations sous forme d'un tableau à deux dimensions de la table de vérité. Elles sont construites de façon à ce que **les termes logiquement adjacents soient géométriquement adjacents**. Chaque ligne de la table de vérité est représentée par une case du tableau de Karnaugh dans laquelle on indique la valeur de la fonction.

La contrainte d'adjacence géométrique est réalisée par un ordonnancement des lignes (resp. colonnes) du tableau pour lequel le nombre de bits modifiés d'un code au suivant est constant et égal à un (code de Gray). Cette propriété est respectée entre le code de la dernière ligne (resp. colonne) et celui de la première ligne (resp. colonne). Prenons par exemple le cas d'une fonction $F$ de trois variables, spécifiée dans le tableau 2.8.

Nous constatons que la fonction $F$ est égale à « 1 » pour :

- les 4 cases (adjacentes) qui constituent la ligne « $a = 1$ » (Fig. 2.11) ;
- les 4 cases (adjacentes) qui constituent le carré « $c = 1$ » (Fig. 2.12).

Les deux zones déterminées « recouvrant » exactement les cases du tableau où la fonction $F$ vaut 1, nous pouvons en déduire directement que : $F = a + c$.

Nous allons maintenant généraliser la méthode exposée dans l'exemple.

| $A$ | $B$ | $C$ | $F(A,B,C)$ | **minterme** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $\overline{A}\cdot\overline{B}\cdot\overline{C}$ |
| 0 | 0 | 1 | 1 | $\overline{A}\cdot\overline{B}\cdot C$ |
| 0 | 1 | 0 | 0 | $\overline{A}\cdot B\cdot\overline{C}$ |
| 0 | 1 | 1 | 1 | $\overline{A}\cdot B\cdot C$ |
| 1 | 0 | 0 | 1 | $A\cdot\overline{B}\cdot\overline{C}$ |
| 1 | 0 | 1 | 1 | $A\cdot\overline{B}\cdot C$ |
| 1 | 1 | 0 | 1 | $A\cdot B\cdot\overline{C}$ |
| 1 | 1 | 1 | 1 | $A\cdot B\cdot C$ |

**Table 2.8:** *Table de vérité de la fonction $F$ : états associés et mintermes.*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Table 2.9:** *Table de Karnaugh de la fonction $F$.*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $\overline{A}\cdot\overline{B}\cdot\overline{C}$ | $\overline{A}\cdot\overline{B}\cdot C$ | $\overline{A}\cdot B\cdot C$ | $\overline{A}\cdot B\cdot\overline{C}$ |
| 1 | $A\cdot\overline{B}\cdot\overline{C}$ | $A\cdot\overline{B}\cdot C$ | $A\cdot B\cdot C$ | $A\cdot B\cdot\overline{C}$ |

**Table 2.10:** *Correspondance des mintermes.*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Table 2.11:** *Adjacence : $a = 1$*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Table 2.12:** *Adjacence : $c = 1$*

### 2.4.4 Construction du tableau de Karnaugh

- Il y a $2^n$ cases pour $n$ variables.
- À chaque case est associé un minterm égal à 1 pour la combinaison considérée.
- Le passage d'une case à une de ses voisines se fait par changement d'une seule variable à la fois.

### 2.4.5 Règles de simplification

Il s'agit de « paver » le tableau de Karnaugh en regroupant les « 1 » adjacents de telle manière que :

- chaque « 1 » de la fonction appartient à au moins un pavé,
- les pavés sont rectangulaires,
- à la fois la longueur et la largeur de chaque pavé sont une puissance de 2.

Chaque pavé ainsi constitué représente un terme produit de la fonction ne contenant que les variables « stables » (par rapport au codage des lignes et des colonnes du tableau).

**Attention !**

- De même qu'il n'y a pas unicité de la représentation algébrique d'une fonction booléenne, il n'y a pas unicité des regroupements géométriques dans le tableau de Karnaugh.
- Il ne faut pas oublier les adjacences possibles entre colonnes et lignes extrêmes du tableau de Karnaugh.

L'exemple suivant (Tab. 2.13) illustre ces deux principes :

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

**Table 2.13:** *Table de Karnaugh*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

**Table 2.14:** *Premier pavage*

| $A$ \ $BC$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |

**Table 2.15:** *Deuxième pavage*

Selon le premier pavage (Tab. 2.14), l'expression obtenue est :

$$\overline{b} \cdot \overline{c} + \overline{a} \cdot c + a \cdot b$$

Le second pavage (Tab. 2.15), qui utilise une adjacence entre deux cases extrêmes, donne :

$$\overline{a} \cdot \overline{b} + b \cdot c + a \cdot \overline{c}$$

### 2.4.6 Fonctions non complètement définies

Certaines combinaisons peuvent ne pas se produire : elles n'ont pas d'effet sur la valeur de la fonction. Ces états indifférents, notés $X$ ou $-$, peuvent être utilisés partiellement ou totalement pour simplifier la fonction, comme illustré dans l'exemple de la Fig. 2.16.

| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | X | X | X |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

**Table 2.16:** *Table de Karnaugh*

| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 1 | X | X |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

**Table 2.17:** *Premier pavage*

| $AB$ \ $CD$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 1 | 1 | X |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 0 |

**Table 2.18:** *Deuxième pavage*

On profite du fait que les états indifférents peuvent être interprétés au choix comme des 1 ou des 0 pour réaliser les regroupements les plus pertinents permettant d'aboutir à une expression logique minimale. Ici, les deux regroupements en carrés retenus dans les tableaux 2.17 et 2.18 s'imposent naturellement.

L'expression obtenue finalement est :

$$\overline{a} \cdot \overline{c} + b \cdot d$$

### 2.4.7 Pertinence de la méthode

Remarquons que l'ordre des lignes et des colonnes est celui d'un "code de Gray". C'est-à-dire que d'une colonne à la suivante, un seul bit de numero de colonne est modifié. Idem entre les extrémités droite et gauche du tableau. Idem pour les lignes. Ceci est indispensable pour que les regroupements de cases adjascentes soit équivalents à l'application du "théorème d'adjascence". Ceci fonctionne bien avec des adresses de lignes et de colonnes sur 2 bits car il n'y a qu'un seul code de Gray sur 2 bits (à une symétrie ou permutation circulaire près). Or cette propriété d'unicité du code de Gray n'existe plus sur 3 bits. Il y a donc plusieurs façons de construire le "cube de Karnaugh", et il faudrait toutes les essayer... sans compter que c'est difficile de voir les regroupements dans un tel cube...

Cette méthode n'est donc pas utilisée avec plus de 4 variables.

## 2.5 Représentation schématique des fonctions logiques

Notre bibliothèque de fonctions ou portes logiques élémentaires n'est pour l'instant constituée que des trois opérateurs inversion (**NOT**), « et » logique (**AND**) et « ou » logique (**OR**). Nous allons compléter cette bibliothèque par quelques éléments supplémentaires dont l'objectif est de mettre en place une représentation schématique des fonctions logiques.

Nous distinguerons ainsi :

- la fonction NON-ET ou NAND dont la sortie vaut 0 si et seulement si toutes les entrées sont à 1,
- la fonction NON-OU ou NOR dont la sortie vaut 1 si et seulement si aucune entrée n'est à 1.

Ces deux fonctions sont la simple négation des fonctions AND et OR.

Nous associons maintenant à chacune des fonctions NOT, OR, AND, NAND et NOR un symbole graphique. La négation sera représentée systématiquement par un cercle :

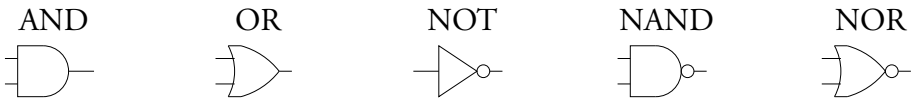



**Figure 2.1:** *Symboles des portes élémentaires.*

Exemple de schéma et équation algébrique correspondante :

$$Y = \left((\overline{A + B}) \cdot C\right) + \overline{C}$$

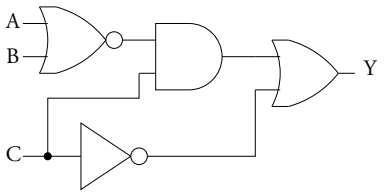



**Figure 2.2:** *Un exemple de schéma.*

## 2.6 Quelques fonctions combinatoires importantes

### 2.6.1 Fonctions d'aiguillage : multiplexeurs

La fonction « multiplexeur à $N$ entrées » consiste à aiguiller vers la sortie de la fonction une entrée parmi $N$. Le multiplexeur à 2 entrées est le multiplexeur le plus simple à concevoir. Son équation algébrique est de la forme :

$$Y = \overline{A} \cdot E_0 + A \cdot E_1 \,,$$

où $(E_0, E_1)$ sont les entrées à multiplexer et $A$ est une entrée de sélection.

La fonction multiplexeur est une traduction directe d'une instruction de type « `if` $\cdots$ `then` $\cdots$ `else` $\cdots$ » dans le cadre de langages informatiques. Elle permet aussi de décomposer une fonction booléenne complexe en utilisant les théorèmes d'expansion.

Le multiplexeur à deux entrées est souvent symbolisé de la façon suivante :

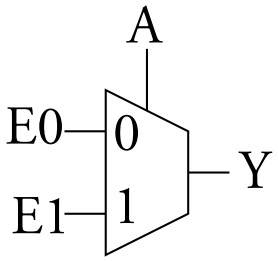



**Figure 2.3:** *Multiplexeur à deux entrées (Mux2).*

Le multiplexeur à $2^N$ entrées nécessite $N$ entrées de sélection pour distinguer les $2^N$ configurations des entrées.

Nous allons tenter maintenant de construire un multiplexeur à 4 entrées à partir des portes de base définies dans le chapitre précédent. L'expression algébrique de la sortie est de la forme :

$$S = \overline{A_0} \cdot \overline{A_1} \cdot E_0 + A_0 \cdot \overline{A_1} \cdot E_1 + \overline{A_0} \cdot A_1 \cdot E_2 + A_0 \cdot A_1 \cdot E_3$$

Cette formulation fait apparaître tous les minterms possibles à partir des entrées de sélection $(A_0, A_1)$. Le schéma de la Fig. 2.4 présente un multiplexeur à 4 entrées muni en outre d'une entrée supplémentaire $V$ permettant de valider ou d'invalider ($S = 0$) la sortie de la fonction.

**Remarque** : Il est possible de construire un multiplexeur à 4 entrées à partir de 3 multiplexeurs à 2 entrées. On se base pour cela sur la reformulation suivante, illustrée dans la Fig. 2.5.

$$S = \overline{A_1} \cdot (\overline{A_0} \cdot E_0 + A_0 \cdot E_1) + A_1 \cdot (\overline{A_0} \cdot E_2 + A_0 \cdot E_3)$$

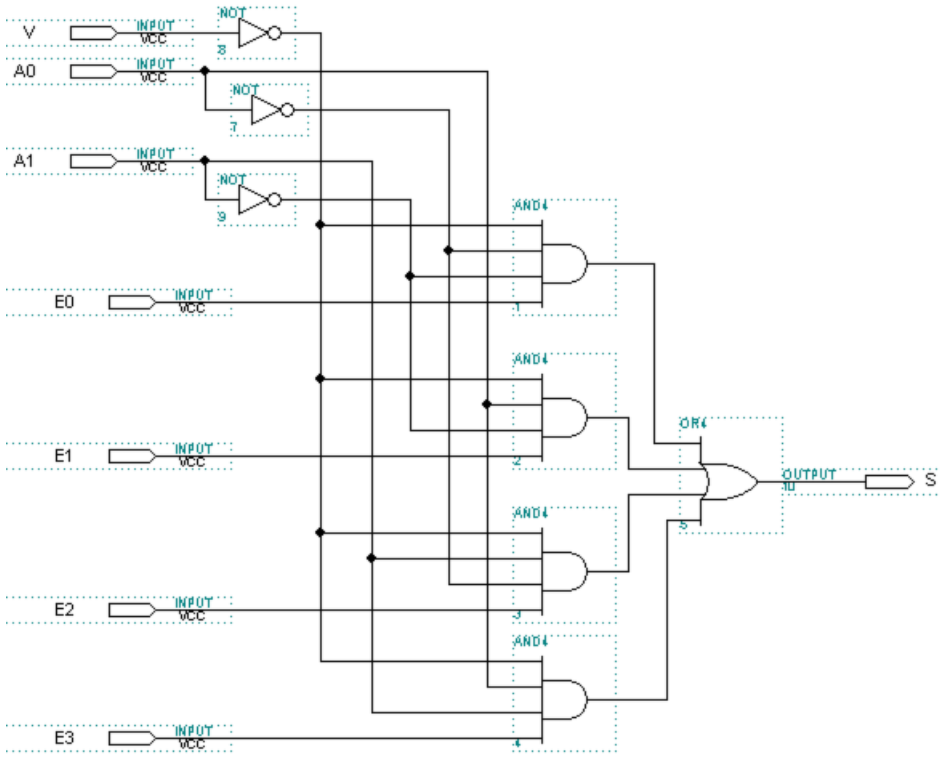



**Figure 2.4:** *Schéma interne d'un multiplexeur à 4 entrées avec entrée de validation.*

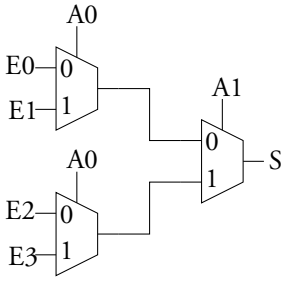



**Figure 2.5:** *Reformulation du multiplexeur à* 4 *entrées.*

### 2.6.2 Opérateurs de comparaison

Les fonctions de comparaison les plus simples sont le test de l'égalité de deux variables booléennes ainsi que le test de complémentarité de deux variables booléennes. Ces deux fonctions ont pour équations algébriques respectives :

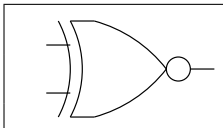
$$\begin{aligned} \text{Égalité :} \quad S &= \overline{A} \cdot \overline{B} + A \cdot B \\ \text{Complémentarité :} \quad S &= A \cdot \overline{B} + \overline{A} \cdot B \end{aligned}$$

Ces fonctions étant très souvent utilisées, il a été jugé utile de définir un nouvel opérateur booléen pour les représenter. Il s'agit de l'opérateur « OU exclusif » que nous représenterons par le symbole suivant : $\oplus$.

Les relations précédentes deviennent :

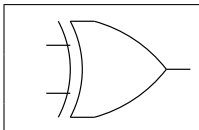
$$\begin{aligned} \text{Égalité (NON-OU-exclusif ou XNOR) :} \quad S &= \overline{A} \cdot \overline{B} + A \cdot B = \overline{A \oplus B} \\ \text{Complémentarité (OU-exclusif ou XOR) :} \quad S &= A \cdot \overline{B} + \overline{A} \cdot B = A \oplus B \end{aligned}$$

Les tables de vérité et symboles associés à ces fonctions sont donnés dans les tableaux 2.19 et 2.20.



| $A$ | $B$ | $\overline{A \oplus B}$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table 2.19:** *Table de vérité et symbole des opérateurs XNOR*



| $A$ | $B$ | $A \oplus B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table 2.20:** *Table de vérité et symbole des opérateurs XOR*

Disposant du comparateur d'égalité à deux entrées, il est possible de généraliser l'opérateur à la comparaison de deux mots de $N$ bits. L'exemple suivant montre un comparateur opérant sur 2 mots de 4 bits.

Nous disposons de 2 nombres codés sur 4 bits : $A = a_3a_2a_1a_0$ et $B = b_3b_2b_1b_0$.

Alors $A = B$ si et seulement si $(a_3 = b_3)$ et $(a_2 = b_2)$ et $(a_1 = b_1)$ et $(a_0 = b_0)$, ce qui justifie le montage de la Fig. 2.6.

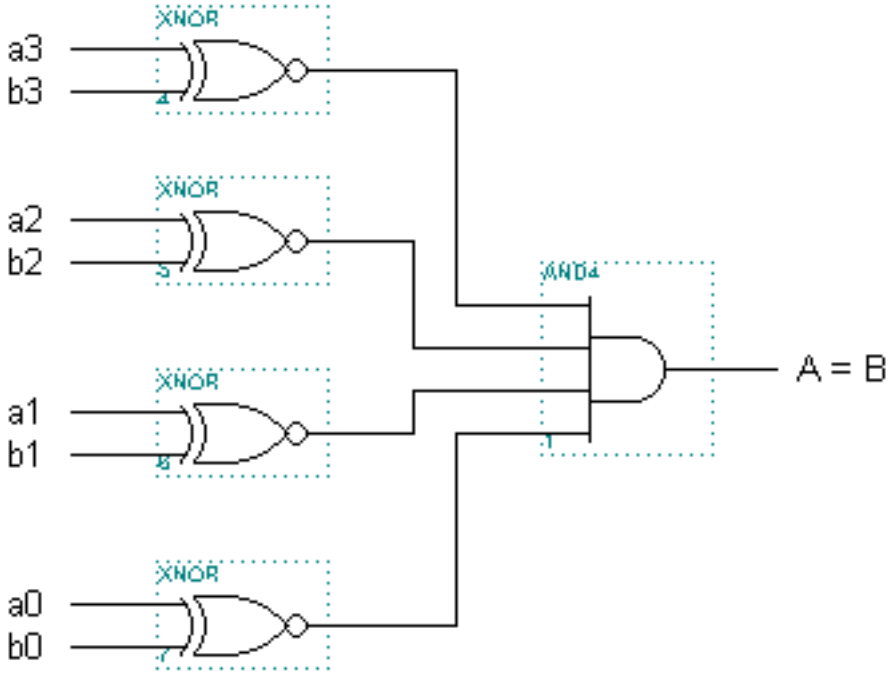



**Figure 2.6:** *Test d'égalité de deux mots de 4 bits.*

## 2.7 Annexes

### 2.7.1 Exercice de consolidation

L'expérience a montré que parmi les notions qui viennent d'être exposées, celle dans laquelle on se prend le plus fréquemment les « pieds dans le tapis » et qui par ailleurs sert le plus dans le cadre du module est la simplification des tables de Karnaugh. L'exercice suivant constitue en conséquence un petit entraînement qui pourrait s'avérer salutaire afin d'être au point sur ce sujet.

On désire réaliser un afficheur 7 segments ($a, b, c, d, e, f, g$, voir Fig. 2.7) traduisant un nombre binaire exprimé sur 4 bits $A$, $B$, $C$, $D$ en un symbole hexadécimal :
$(0, \cdots, 9, A, b, C, d, E, F)$.

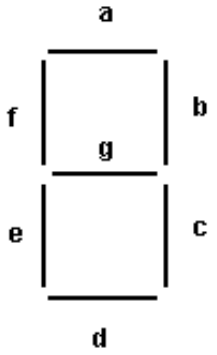



**Figure 2.7:** *Afficheur 7 segments. Un segment = une diode électro-luminescente.*

La transcription de cette réalisation en tables de vérité si l'on considère que l'on travaille en logique positive (segment allumé = « 1 ») donne une table par segment.

Pour vous aider, il vous est proposé de vous donner directement le tableau de Karnaugh correspondant au segment « $a$ » ainsi que le résultat obtenu après simplification.

| **AB \ CD** | **00** | **01** | **11** | **10** |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | 1 | 1 |
| **11** | 1 | 0 | 1 | 1 |
| **10** | 1 | 1 | 0 | 1 |

**Figure 2.8:** *Tableau de Karnaugh de* $a = F(A, B, C, D)$.

Le résultat à trouver est :

$$a = A \cdot \overline{D} + B \cdot C + \overline{A} \cdot C + \overline{B} \cdot \overline{D} + \overline{A} \cdot B \cdot D + A \cdot \overline{B} \cdot \overline{C}$$

Comment l'obtient-on ? Détaillons pour les sceptiques !

**Regroupement :** $B \cdot C$

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | 1 | |
| **11** | 1 | 0 | | |
| **10** | 1 | 1 | 0 | 1 |

**Regroupement :** $\overline{A} \cdot C$

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | |
| **01** | 0 | 1 | | |
| **11** | 1 | 0 | 1 | 1 |
| **10** | 1 | 1 | 0 | 1 |

**Regroupement :** $A \cdot \overline{D}$

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | 1 | 1 |
| **11** | 1 | 0 | 1 | 1 |
| **10** | | 1 | 0 | |

**Regroupement :** $\overline{B} \cdot \overline{D}$**.**
**Les extrémités sont logiquement adjacentes !**

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | 1 | 1 |
| **11** | 1 | 0 | 1 | 1 |
| **10** | 1 | 1 | 0 | 1 |

**Regroupement :** $\overline{A} \cdot B \cdot D$

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | | 1 |
| **11** | 1 | 0 | 1 | 1 |
| **10** | 1 | 1 | 0 | 1 |

**Enfin, Regroupement :** $A \cdot \overline{B} \cdot \overline{C}$**.**

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 1 | 0 | 1 | 1 |
| **01** | 0 | 1 | 1 | 1 |
| **11** | 1 | 0 | 1 | 1 |
| **10** | 1 | | 0 | 1 |

L'équation du premier segment $a$ est ainsi obtenue à l'aide des 6 regroupements précédents. A vous de les effectuer pour les autres segments. . .

Si vous arrivez au bout de cet exercice dans un temps raisonnable, vous pouvez considérer que vous ne rencontrerez pas beaucoup de problèmes dans le futur !

### 2.7.2 Bibliographie


- Groupe numérique *Polycopié Composants et fonctions de l'électronique numérique*, ENST
- Ronald Tocci *« Digital Systems »*, Prentice Hall
- Eugene D. Fabricius, *« Modern Digital Design and Switching Theory »*, CRC Press, 1992.

- Barry Wilkinson, *« Digital System Design »*, Prentice Hall, 1992.

# Chapitre 3

# Opérateurs arithmétiques

## 3.1 Introduction

Jusqu'à présent, nous avons principalement travaillé sur des bits simples (comprendre : des nombres de 1 bit), ce qui ne nous permet de représenter que les valeurs 0 et 1. Dans ce chapitre nous introduirons la représentation des nombres entiers plus grands que 1 et fractionnaires, ainsi que les opérateurs associés. Nous verrons donc :

1. Représentation des nombres (codage des nombres)
   (a) Représentation Simples de Position
   (b) Conversions entre Bases
   (c) Représentation en Signe et Valeur Absolue
   (d) Représentation en Complément à 2
   (e) Autres Codes
2. Opérateurs arithmétiques
   (a) Additionneur
   (b) Soustracteur

## 3.2 Représentation (codage) des nombres

### 3.2.1 Représentation Simples de Position

Un nombre positif $N$ dans un système de base $b$ peut être exprimé sous la forme polynomiale $N = a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \ldots + a_1 \cdot b^1 + a_0 \cdot b^0 + \ldots + a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + a_{-m+1} \cdot b^{-m+1} + a_{-m} \cdot b^{-m}$

La représentation simples de position correspondante est $a_{n-1}\ a_{n-2}\ \ldots a_1\ a_0, a_{-1}\ \ldots a_{-m}$
La position d'un chiffre rappelle quelle puissance de la base multiplie ce chiffre :

- $a_i$ est le chiffre de rang $i$ ($a_i$ appartient à un ensemble de $b$ symboles)
- $a_{n-1}$ est le chiffre le plus significatif
- $a_{-m}$ est le chiffre le moins significatif

S'il s'agit du système héxadécimal (b = 16) :

- $a_i$ appartient à l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$

S'il s'agit du système octal (b = 8) :

- $a_i$ appartient à l'ensemble $\{0, 1, 2, 3, 4, 5, 6, 7\}$

S'il s'agit du système binaire (b = 2), les chiffres sont appelés bits :

- $a_i$ appartient à l'ensemble $\{0, 1\}$

Si l'on se déplace d'un rang vers la gauche, le poids est augmenté d'un facteur $b$. Si le déplacement se fait vers la droite, il y a une division par $b$.
Le tableau 3.1 montre la représentation simples de position pour les nombres décimaux de 0 à 8 et leurs correspondances en binaire. Les décalages correspondant aux multiplications et divisions par 2 peuvent être vus :

- en comparant $1_{10}$ et $2_{10}$ ($1_2$ et $10_2$ )
- en comparant $3_{10}$ et $6_{10}$ ($11_2$ et $110_2$)
- en comparant $4_{10}$ et $8_{10}$ ($100_2$ et $1000_2$)

| Décimal | Binaire |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |

**Table 3.1:** *Exemple conversion binaire-décimal*

### 3.2.2 Conversions entre Bases

**Base $b$ vers base** 10

Pour cette conversion, il suffit de substituer la valeur $b$ dans l'expression polynomiale par la valeur de la base : $N = a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \ldots + a_1 \cdot b^1 + a_0 \cdot b^0 + \ldots + a_{-1} \cdot b^{-1} + a_{-2} \cdot b^{-2} + a_{-m+1} \cdot b^{-m+1} + a_{-m} \cdot b^{-m}$

Par exemple, pour trouver le correspondant de $(A1C)_{16}$ dans le système décimal, il suffit de faire : $10 \cdot 16^2 + 1 \cdot 16^1 + 12 \cdot 16^0 = 2768_{10}$

**Base** 10 **vers base** $b$

**Partie Entière**
Cette conversion consiste à faire des divisions successives du nombre décimal par $b$, jusqu'à

obtenir un quotient nul. Le nombre dans la base $b$ correspond aux restes des divisions faites, dans le sens inverse où ils ont été obtenus.

Soit la conversion $57_{10}$ vers base 2 :

| Division | Quotient | Reste |
|---|---|---|
| $57/2$ | $28$ | $1(a_0)$ |
| $28/2$ | $14$ | $0(a_1)$ |
| $14/2$ | $7$ | $0(a_2)$ |
| $7/2$ | $3$ | $1(a_3)$ |
| $3/2$ | $1$ | $1(a_4)$ |
| $1/2$ | $0$ | $1(a_5)$ |

Le résultat est donc $(111001)_2$, où l'on n'affiche que les "restes".

**Partie Fractionnaire**

Cette conversion consiste à faire des multiplications successives du nombre décimal par $b$. Le nombre dans la base $b$ correspond aux parties entières des produits des multiplications faites, dans le sens direct où ils ont été obtenus.

Soit la conversion $0,57_{10}$ vers base 2 :

| Multiplication | Produit | Partie entière |
|---|---|---|
| $0,57 \cdot 2$ | $1,14$ | $1(a_{-1})$ |
| $0,14 \cdot 2$ | $0,28$ | $0(a_{-2})$ |
| $0,28 \cdot 2$ | $0,56$ | $0(a_{-3})$ |
| $0,56 \cdot 2$ | $1,12$ | $1(a_{-4})$ |
| $0,12 \cdot 2$ | $0,24$ | $0(a_{-5})$ |
| $0,24 \cdot 2$ | $0,48$ | $0(a_{-6})$ |

Le résultat est donc $(0,100100)_2$.

**Base $2^n$ vers base $2$ et base $2$ vers base $2^n$**

A l'aide de $n$ bits, la conversion se fait sur chaque chiffre en base 2 pour ensuite les juxtaposer :

Par exemple, $(3A9)_{16} = (001110101001)_2$ et $(264)_8 = (010110100)_2$

**Base $i$ vers base $j$**

Si les deux bases sont des puissances de 2, la conversion se fait en utilisant 2 comme base relais ($i$ vers 2 et ensuite 2 vers $j$). Sinon, la base relais est la base 10.

### 3.2.3 Représentation en Signe et Valeur Absolue

La représentation en signe et valeur absolue consiste à ajouter un bit $s$ à la représentation simple de position afin de pouvoir représenter des nombres négatifs :
$(s\ a_{n-1}\ a_{n-2}\ \ldots a_1\ a_0, a_{-1}\ldots a_{-m})$.

La convention adoptée est $s = 0$ pour un nombre positif et $s = 1$ pour un nombre négatif. Ainsi, pour une représentation sur 4 bits, $+5 = 0101$ et $-5 = 1101$.

Du fait que cette représentation implique un traitement différent pour le bit de signe, elle est peu intéressante pour l'implantation d'opérateurs arithmétiques.

### 3.2.4 Représentation en Complément à 2

Il existe une forme de représentation des nombres signés plus efficace que la représentation en Signe et Valeur Absolue : le complément à deux. Le principe du complément à deux est simple : dans la représentation non signée (simples de position, ou notation binaire habituelle) sur $n$ bits, on travaille *implicitement modulo* $2^n$. Ainsi, sur $n$ bits :

- $2^n$ a la même représentation que $0$,
- $(2^n + 1)$ la même que $1$,
- etc.

C'est pour cela que pour éviter les ambiguïtés, on se limite (en binaire non signé) à la représentation des nombres allant de $0$ à $(2^n - 1)$ (soit $2^n$ nombres au total).

Ce principe du modulo peut être étendu aux nombres négatifs. Toujours sur $n$ bits :

- si $0$ a la même représentation que $2^n$ (soit $00...00$),
- $-1$ devra donc avoir la même que $(2^n - 1)$ (soit $11...11$),
- $-2$ la même que $(2^n - 2)$ (soit $11...10$),
- etc.

Le complément à deux n'est qu'une convention, consistant à dire qu' *on décale la plage des nombres représentables, en mettant le* $0$ *au centre, et que, par compatibilité avec la représentation binaire non signée, les nombres commençant par* $0$ *seront considérés positifs, et ceux commençant par* $1$ *négatifs*.
Autrement dit, au lieu de représenter des nombres non-signés allant de $0$ à $(2^n - 1)$, on représentera des nombres signés allant de $-(2^{n-1})$ à $+(2^{n-1} - 1)$, soit $2^n$ nombres au total dans les deux cas.

Remarques :

- Par convention, le 0 est donc classé dans les nombres positifs.
- Les nombres positifs ont la même représentation en binaire non signé qu'en complément à 2.
- Le complément à 2 permet de représenter moins de nombres positifs que le binaire non signé (c'est normal, l'intervalle de $2^n$ nombres a été séparé en deux parties de même longueur, une pour les positifs, une pour les négatifs).
- Le nombre d'entiers non nuls représentables en CA2 etant impair, une des deux plages sera plus grande que l'autre. Le 0 étant considéré positif (car sa représentation commence par un 0), il y aura donc un nombre strictement négatif de plus que de nombres strictement positifs.

Par exemple, pour une représentation sur 4 bits ($n = 4$),

- on peut représenter les nombres signés allant de $-8$ à $+7$,
- $+5 \equiv 0101_2$,
- $-5 \equiv 2^4 + (-5) \equiv +11 \equiv 1011_2$
- $+7 \equiv 0111_2$
- $-7 \equiv 2^4 + (-7) \equiv +9 \equiv 1001_2$
- $-8 \equiv 2^4 + (-8) \equiv +8 \equiv 1000_2$
- $+8$ est non représentable (car il aurait la même représentation que $-8$).

Le complément à deux permet que l'opérateur utilisé pour faire des additions puisse aussi faire des soustractions : au lieu de soustraire un nombre à l'aide d'un opérateur spécialisé de soustraction, il suffit d'ajouter son opposé (attention : "opposé" n'est pas la négation binaire bit à bit, mais l'opposé arithmétique) en complément à deux, et de travailler à nombre de bits constant (pour rester modulo $2^n$).

Exemple : soit l'opération $7 - 5$ en décimal. En binaire, pour une représentation sur 4 bits, cela correspond à $0111_2 - 0101_2$.

La soustraction de $0101_2$ peut être remplacée par une addition de son complément à 2, soit $1011_2$.

$7 - 5 = 7 + (-5) = 0111_2 + 1011_2 = 10010_2$. Il ne faut garder que les 4 bits de poids faible, pour obtenir la réponse exacte, c'est-à-dire, $0010_2 = 2$.

### 3.2.5 Autres Codes

Ci-après sont présentés quelques codes utilisés dans les systèmes numériques. Les codes de *Gray*, *p parmi n* et le code *à bit de parité* ne sont pas pondérés, c'est-à-dire qu'il n'y a pas de poids attribué à chaque position (rang). Les rapports entre les symboles des codes et les nombres sont de simples tableaux de correspondances convenus. De ce fait, ils sont moins appliqués aux opérations arithmétiques. Ils sont principalement rencontrés dans les systèmes de communication pour le contrôle de transmission/réception de données.

#### Code BCD (Binary Coded Decimal)

Dans le code BCD chaque chiffre décimal $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ est codé en binaire à l'aide de 4 bits. Pour la conversion BCD vers Binaire, il suffit de convertir chaque chiffre individuellement. La conversion Binaire vers BCD se fait en regroupant les bits 4 par 4. Ainsi, $178_{BCD} = 000101111000_2$.

#### Code de Gray

Dans le code de Gray, deux termes successifs ne diffèrent que par un seul bit. Les termes ne différant que par un seul bit sont appelés adjacents.

#### Code p parmi n

Dans ce code, à chaque nombre décimal correspondent n bits, dont p valent 1 et $n - p$ valent 0. Il permet de détecter jusqu'à une erreur : si lors d'une communication, il y a réception

d'un nombre de 1 différent de p, cela signifie qu'il y a eu une erreur de transmission. Le tableau 3.2 illustre un exemple de ce code pour le cas $n = 5$ et $p = 2$.

#### Code à bits de parité

Dans ce code, un bit est ajouté aux symboles de départ de sorte que le nombre total de 1's soit pair (impair), si la parité convenue est paire (impaire). Le tableau 3.2 donne l'exemple pour 4 bits d'information et une parité paire.

| **Décimal** | **Binaire** | **BCD** | **Gray** | **p parmi n** | **parité** |
|---|---|---|---|---|---|
| n | DCBA | DCBA | DCBA | EDCBA | DCBAP |
| 0 | 0000 | 0000 | 0000 | 00011 | 00000 |
| 1 | 0001 | 0001 | 0001 | 00101 | 00011 |
| 2 | 0010 | 0010 | 0011 | 01001 | 00101 |
| 3 | 0011 | 0011 | 0010 | 10001 | 00110 |
| 4 | 0100 | 0100 | 0110 | 00110 | 01001 |
| 5 | 0101 | 0101 | 0111 | 01010 | 01010 |
| 6 | 0110 | 0110 | 0101 | 10010 | 01100 |
| 7 | 0111 | 0111 | 0100 | 01100 | 01111 |
| 8 | 1000 | 1000 | 1100 | 10100 | 10001 |
| 9 | 1001 | 1001 | 1101 | 11000 | 10010 |
| 10 | 1010 | - | 1111 | - | 10100 |
| 11 | 1011 | - | 1110 | - | 10111 |
| 12 | 1100 | - | 1010 | - | 11000 |
| 13 | 1101 | - | 1011 | - | 11011 |
| 14 | 1110 | - | 1001 | - | 11101 |
| 15 | 1111 | - | 1000 | - | 11110 |

**Table 3.2:** *Exemple de différents codes*

## 3.3 Fonctions arithmétiques

La réalisation de fonctions arithmétiques est basée sur la décomposition de ces fonctions en opérations booléennes élémentaires.

### 3.3.1 Additionneur

Considérons l'addition de deux nombres $a_i$ et $b_i$ codés sur 1 bit. Le résultat peut prendre les valeurs 0, 1 ou 2, que l'on peut coder en binaire comme 00, 01 et 10. Les deux bits de ce code sont appelés bit de somme $s_i$ (poids faible) et bit de retenue $r_{i+1}$ (poids fort). A l'aide

de ces deux bits, l'addition s'exprime de la façon suivante :

$$a_i + b_i = 2 \cdot r_{i+1} + s_i$$

**Attention !** Ici, les signes "+" et "·" ne représentent pas le OU et le ET, mais l'addition et la multiplication arithmétiques sur plusieurs bits !

L'addition peut être vue comme une fonction booléenne à deux entrées ($a_i$ et $b_i$) et à deux sorties ($s_i$ et $r_{i+1}$). Cette fonction est appelée demi-additionneur. Ses caractéristiques sont résumées dans le tableau 3.1.

| $a_i$ | $b_i$ | $r_{i+1}$ | $s_i$ | **Valeur décimale** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 2 |

$$r_{i+1} = a_i \cdot b_i$$
$$s_i = a_i \oplus b_i$$

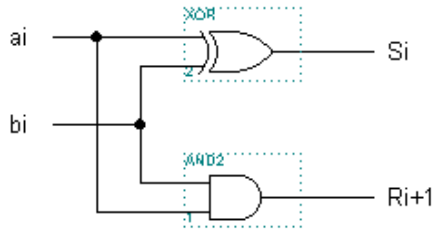



**Figure 3.1:** *Table de vérité, équations algébriques et schéma d'un demi-additionneur.*

Nous pouvons généraliser cette structure pour décrire l'addition de deux mots $A$ et $B$ de taille supérieure à 1. Chacun des bits $a_i$ et $b_i$ sont additionnés un par un en commençant par les bits de poids faible. Il faut pour cela répercuter à l'étape $i+1$ l'éventuelle retenue provenant de l'addition de $a_i$ et $b_i$. Une variable supplémentaire $r_i$ représentant une retenue entrante est donc introduite. Par analogie, le bit $r_{i+1}$ est appelé retenue sortante.

A chaque itération $i$, le résultat de cette addition des nombres $a_i$, $b_i$ et $r_i$ peut prendre les valeurs 0, 1, 2 ou 3, que l'on code en binaire comme 00, 01 et 10, 11. En utilisant les notations précédentes, l'équation arithmétique de l'additionneur « 1 bit » avec retenue entrante (ou additionneur « complet ») est alors :

$$a_i + b_i + r_i = 2 \cdot r_{i+1} + s_i$$

Nous présentons dans les tableaux suivants (3.3, 3.4, 3.5) la table de vérité de l'additionneur complet ainsi que les tables de Karnaugh associées à $r_{i+1}$ et $s_i$.

| $a_i$ | $b_i$ | $r_i$ | $r_{i+1}$ | $s_i$ | **Valeur décimale** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 2 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 2 |
| 1 | 1 | 0 | 1 | 0 | 2 |
| 1 | 1 | 1 | 1 | 1 | 3 |

**Table 3.3:** *Table de vérité de l'additionneur complet*

| $r_i \backslash a_i b_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

**Table 3.4:** *Table* $s_i$

| $r_i \backslash a_i b_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |

**Table 3.5:** *Table* $r_{i+1}$

Nous observons sur les tables de Karnaugh que l'expression de la somme $s_i$ n'est pas réductible, la forme en damier obtenue est caractéristique des fonctions de type ou-exclusif :

$$S_i = a_i \oplus b_i \oplus r_i$$

En ce qui concerne la retenue, pour donner un exemple « d'optimisation », nous allons supposer l'existence d'une structure de calcul de la somme et tenter de mettre en facteur le « matériel » :

$$\begin{aligned} r_{i+1} &= a_i \cdot b_i + r_i \cdot b_i + r_i \cdot a_i = a_i \cdot b_i + r_i \cdot b_i \cdot (\overline{a_i} + a_i) + r_i \cdot a_i \cdot (\overline{b_i} + b_i) \\ &= a_i \cdot b_i + r_i \cdot (a_i \cdot b_i + a_i \cdot b_i) + r_i \cdot (\overline{a_i} \cdot b_i + \overline{b_i} \cdot a_i) = a_i \cdot b_i + r_i \cdot (\overline{a_i} \cdot b_i + a_i \cdot \overline{b_i}) \\ &= a_i \cdot b_i + r_i \cdot (a_i \oplus b_i) \end{aligned}$$

Le schéma de l'additionneur complet s'en déduit : (figure 3.2)

Pour des nombres de $n$ chiffres, la sommation va entraîner une propagation de la retenue si l'on adopte la structure série qui résulte de la mise en oeuvre de l'équation précédente. Dans l'additionneur à retenue série (Ripple Carry Adder), on assiste à un phénomène de propagation de la retenue (cf. Fig. 3.3), facile à cerner si l'on étudie l'addition de tranches de bits du type suivant : $11111111 + 00000001$

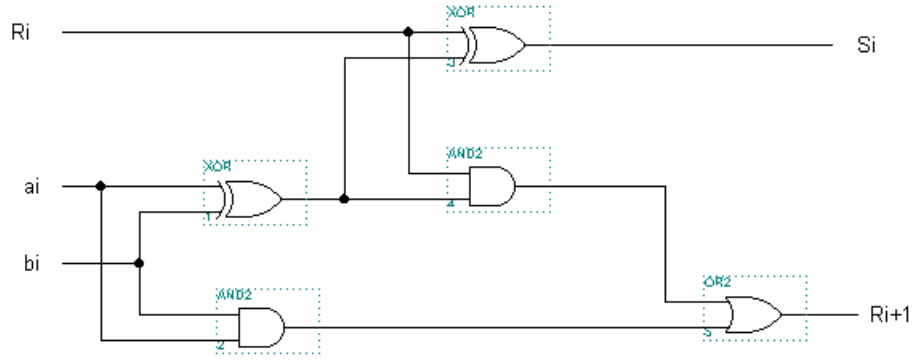



**Figure 3.2:** *Exemple de schéma pour l'additionneur complet.*

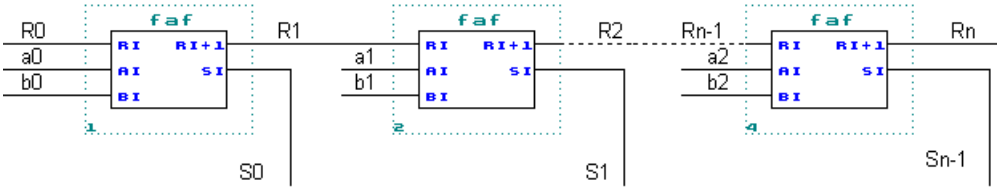



**Figure 3.3:** *Additionneur à retenue série.*

**Remarque :** La structure proposée dans la Fig. 3.2 pour l'additionneur complet n'est pas la seule possible. Suivant l'objectif visé par l'utilisateur, d'autres structures sont envisageables, notamment dans le but d'accélérer la vitesse de calcul de la retenue qui conditionne le temps de calcul total de l'additionneur.

### 3.3.2 Soustracteur

La soustraction de deux nombres $a_i$ et $b_i$ codés sur 1 bit chacun (attention à l'ordre !) donne un résultat pouvant prendre les valeurs arithmétiques $-1$, 0 et 1, qui seront, tout comme pour l'addition, codées sur 2 bits : $r_{i+1}$ et $d_i$ ($d_i$ est la "différence"). Nous pouvons formuler cette opération sous la forme :

$$a_i - b_i = -2 \cdot r_{i+1} + d_i$$

où $r_{i+1}$ et $d_i$ sont deux variables booléennes représentant respectivement la retenue sortante et la différence.

**Attention** : Le bit de retenue $r_{i+1}$ est précédé du facteur $-2$, c'est-à-dire qu'il est interprété comme une valeur négative. On peut comprendre cela en considérant que le résultat sur 2 bits est codé en CA2.

Cette fonction de deux entrées et deux sorties est appelée demi-soustracteur. Ses caractéristiques sont résumées dans le tableau suivant : (figure 3.4)

Nous pouvons généraliser cette structure pour décrire la soustraction de mots de taille supérieure à 1. Pour cela il faut introduire une variable supplémentaire $r_i$ qui représente une retenue entrante. L'équation générale du soustracteur « 1 bit » avec retenue entrante (ou sous-

| $a_i$ | $b_i$ | $r_{i+1}$ | $d_i$ | **Valeur décimale** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | $-1$ |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

$$r_{i+1} = \overline{a_i} \cdot b_i$$
$$d_i = a_i \oplus b_i$$

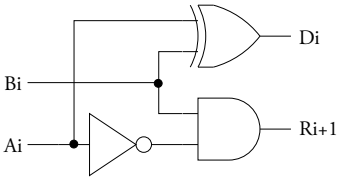



**Figure 3.4:** *Equations algébriques, table de vérité et schéma d'un demi-soustracteur.*

tracteur « complet ») est alors :

$$a_i - b_i - r_i = -2 \cdot r_{i+1} + d_i$$

Nous pouvons constater que les deux variables $r_i$ et $r_{i+1}$ sont affectées d'un facteur négatif, ce qui établit la cohérence de leurs représentations.

Nous présentons, dans les tableaux suivants, la table de vérité du soustracteur complet (Tab. 3.6) ainsi que les tables de Karnaugh associées à $r_{i+1}$ (3.7) et $d_i$ (3.8).

| $a_i$ | $b_i$ | $r_i$ | $r_{i+1}$ | $d_i$ | **Valeur décimale** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | $-1$ |
| 0 | 1 | 0 | 1 | 1 | $-1$ |
| 0 | 1 | 1 | 1 | 0 | $-2$ |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | $-1$ |

**Table 3.6:** *Table de vérité du soustracteur complet*

Un raisonnement identique à celui utilisé dans le cas de l'additionneur aboutit aux équations :

$$\begin{aligned} d_i &= a_i \oplus b_i \oplus r_i \,, \\ r_{i+1} &= \overline{a_i} \cdot b_i + r_i \cdot \overline{(a_i \oplus b_i)} \end{aligned}$$

| $r_i \backslash a_i b_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |

**Table 3.7:** *Table $d_i$*

| $r_i \backslash a_i b_i$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

**Table 3.8:** *Table $r_{i+1}$*

Un schéma du soustracteur complet s'en déduit (cf Fig. 3.5).

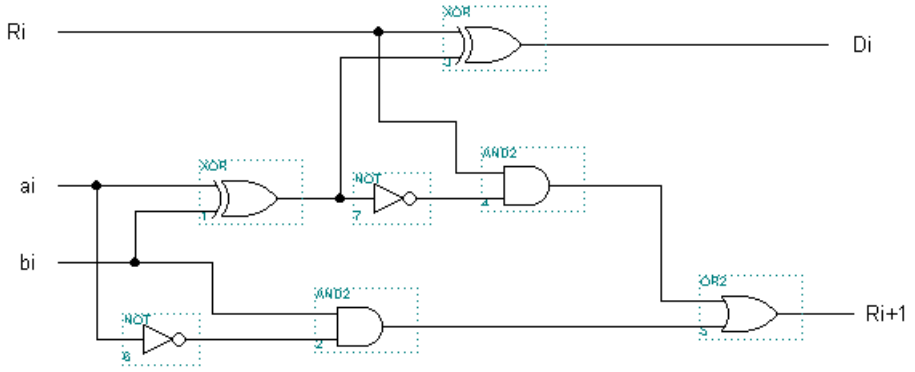



**Figure 3.5:** *Schéma interne du soustracteur complet.*

# Chapitre 4

# Logique séquentielle synchrone, bascules

## 4.1 Introduction

D'une façon complémentaire à la logique combinatoire, la logique séquentielle permet d'organiser les calculs booléens dans le temps. Par exemple, pour additionner 1000 nombres, plutôt que d'effectuer 999 additions avec 999 additionneurs, une solution consiste à additionner à tour de rôle les 1000 opérandes avec 1 seul additionneur. Pour ce faire un circuit à base de logique séquentielle est nécessaire pour :

- présenter successivement les 1000 opérandes
- accumuler le résultat de l'addition avec un résultat intermédiaire
- arrêter le calcul sur le 1000ème opérande

Cet exemple illustre la « sérialisation » des calculs mais la logique séquentielle peut tout aussi bien servir à « paralléliser ». Par exemple, si les débits de calcul sont 2 fois trop faibles en utilisant la logique combinatoire, une solution consiste à mettre en parallèle 2 opérateurs et présenter alternativement les données impaires sur le premier et paires sur l'autre. Dans cet exemple, la logique séquentielle permet d'orienter correctement les données et de concaténer les résultats.

L'ordonnancement temporel et conditionnel des tâches que procure la logique séquentielle permet de concevoir des algorithmes de calcul puissants et des machines à calculer génériques comme les automates et les processeurs. Le séquencement nécessite une fonction propre à la logique séquentielle : la mémorisation. Celle-ci permet de geler les données et les commandes de façon à les réutiliser dans un ordre défini.

### 4.1.1 Comment reconnaître la logique séquentielle ?

Depuis le début du cours de PAN, nous avons étudié les circuits combinatoires. Le comportement logique de ces circuits est tel que la présentation, à des instants différents, des mêmes valeurs d'entrée produira à chaque fois les mêmes résultats. La figure 4.1 dans laquelle on a représenté le chronogramme des combinaisons des entrées et des sorties par des couleurs illustre cette propriété.

Quelles que soient les couleurs présentées aux entrées pendant les intervalles de temps figurés en grisé et quelles que soient les couleurs correspondantes observées aux sorties, la couleur bleu à l'entrée du circuit produit toujours (après un temps de propagation) la couleur vert en sortie.

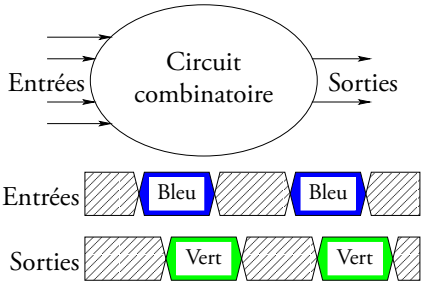



**Figure 4.1:** *Chronogramme d'un circuit combinatoire*

Un circuit de logique séquentielle ne possède pas cette propriété ; la connaissance des entrées appliquées à un instant donné ne suffit pas à déterminer les valeurs des sorties comme le montre la figure 4.2. La combinaison bleue présentée à plusieurs reprises aux entrées du circuit ne produit pas toujours le même résultat. La valeur observée aux sorties est tantôt le vert, tantôt le violet.

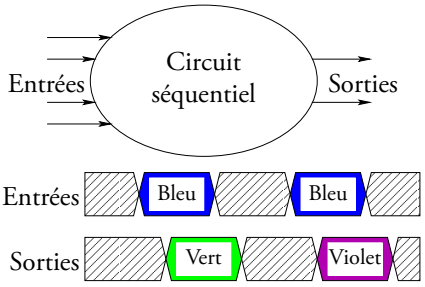



**Figure 4.2:** *Chronogramme d'un circuit séquentiel*

Le comportement de la logique séquentielle s'explique par la présence de variables supplémentaires internes au circuit dont la valeur évolue au cours du temps, ce sont les mémoires internes. La connaissance de la valeur de ces variables internes est nécessaire si l'on veut prévoir la valeur des sorties. La figure 4.3 illustre le chronogramme du circuit en tenant compte des variables internes.

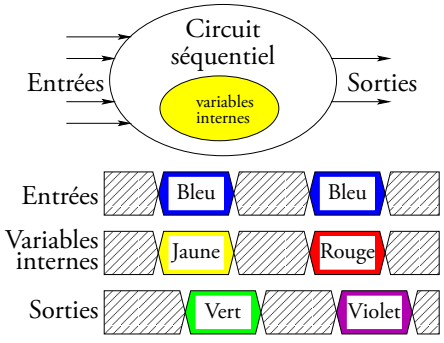



**Figure 4.3:** *Chronogramme avec les variables internes*

### 4.1.2 Comment construire la logique séquentielle ?

Les valeurs des variables internes reflètent « l'état du système » qui dépend des entrées et de leurs valeurs précédentes. Les variables internes contiennent une partie de l'histoire du circuit car elles dépendent des valeurs passées des entrées. La structure du circuit de logique séquentielle possède donc un circuit combinatoire calculant les variables internes et recevant les entrées et les variables internes, entraînant ainsi un rebouclage. Ce rebouclage doit inclure la fonction de mémorisation propre à la logique séquentielle.

Il est difficile de faire fonctionner un dispositif rebouclé, d'une façon fiable, car chaque variable interne et chaque sortie a son propre temps de propagation, générant des *courses* entre signaux et pouvant entraîner un dysfonctionnement du système. Une méthode largement répandue pour l'évolution des calculs consiste à les synchroniser. Dans cette méthode les variables internes et les sorties sont gelées dans une mémoire, généralement une bascule D (cf paragraphe 4.2), et mises à jour après la fin de calcul d'un circuit combinatoire. Les sorties sont mémorisées car elles sont potentiellement utilisées comme entrées d'autres circuits séquentiels. La mise à jour des mémoires, ou échantillonnage des résultats, se fait d'un façon synchrone au rythme d'un signal de commande périodique : l'horloge. La figure 4.4 illustre la structure générale de la logique séquentielle synchrone :

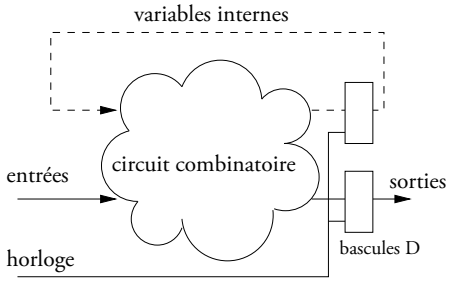



**Figure 4.4:** *Structure de base d'un circuit en logique séquentielle synchrone*

L'instant d'échantillonnage correspond à une transition montante ou descendante du signal d'horloge dont le chronogramme est donné en figure 4.5. Le rapport cyclique de cette horloge, c'est à dire le rapport entre le temps où l'horloge vaut 1 et le temps ou l'horloge vaut 0, peut en conséquence être différent de $50/50$.

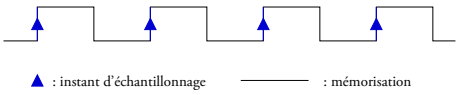



**Figure 4.5:** *Chronogramme du signal d'horloge*

Pour assurer la bonne marche d'un circuit en logique séquentielle synchrone disposant d'une unique horloge, il suffit de connaître le temps maximum de calcul du circuit combinatoire et d'utiliser une période d'horloge supérieure à ce temps. Le chemin le plus lent d'un circuit combinatoire s'appelle « chemin critique » . Si $T_h$ est la période d'horloge et $T_{crit}$ est le temps de propagation du chemin critique, alors il suffit de respecter :

$$T_h > T_{crit}$$

Le calcul du chemin critique se fait dans les conditions d'utilisation les pires, c'est-à-dire un procédé technologique sous-optimal, une tension d'alimentation $V_{dd}$ faible et une température de jonction élevée.

## 4.2 Les bascules D

La brique de base spécifique à la logique séquentielle est le point mémoire. Il existe différentes technologies pour créer le point mémoire. La « bascule D » est un composant de mémorisation pour un seul point mémoire. La mémoire RAM *Random Access Memory* est un ensemble de points mémoires regroupés dans une matrice. L'accès à la RAM ne permet pas d'accéder à tous les point mémoires en même temps, mais à un seul. Un mécanisme d'adressage est donc nécessaire pour sélectionner un point mémoire qui dispose ainsi de sa propre « adresse » .

### 4.2.1 Le point mémoire élémentaire

Une des techniques de mémorisation repose sur le principe de stabilité des systèmes en boucle fermée comme illustré par la figure 4.6. Ce principe est utilisé en technologie CMOS pour construire les bascules et les RAMs statiques. Un amplificateur rebouclé sur lui même constitue un système stable (qui ne change pas d'état). S'il existe un moyen d'initialiser l'amplificateur avec une certaine valeur, celle ci est gelée dans l'amplificateur qui, par le rebouclage sur lui-même, a le rôle de point mémoire. Une autre technique de mémorisation consiste à capturer des charges électriques dans un condensateur. Ce principe est utilisé pour les mémoires FLASH et RAM dynamiques.

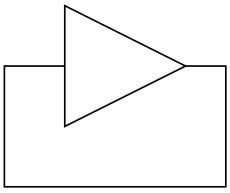


**Figure 4.6:** *Point mémoire basé sur un amplificateur rebouclé*

La fonction de transfert de l'amplificateur doit avoir une zone d'amplification dans son milieu (pente strictement supérieure à 1) pour pouvoir converger vers le niveau haut ou bas de la courbe. La figure 4.7 illustre un exemple de fonction de transfert.

Il existe 3 points stables de la fonction de transfert 0, $V_{max}$ et X. Les 2 premiers correspondent au fonctionnement normal et permettent d'associer les grandeurs physiques (0,Vmax) à des niveaux booléens (vrai, faux).

Le point X correspond à un état « métastable » associé à aucun niveau booléen, il faut donc l'éviter . S'il existe la zone d'amplification en milieu de courbe, Il suffit de s'écarter légèrement de ce point pour converger vers O ou Vmax. La figure 4.8 illustre le fait que le point initial ($V_{e1}, V_{s1}$) proche du point X converge rapidement vers $V_{max}$.

Le bruit ambiant contribue à ce que cet état métastable ne dure pas et que le système converge. Le temps de convergence n'est pas constant, il dépend de la technologie et peut être très long. Les points mémoires ont toujours des contraintes d'utilisation pour éviter cet état.

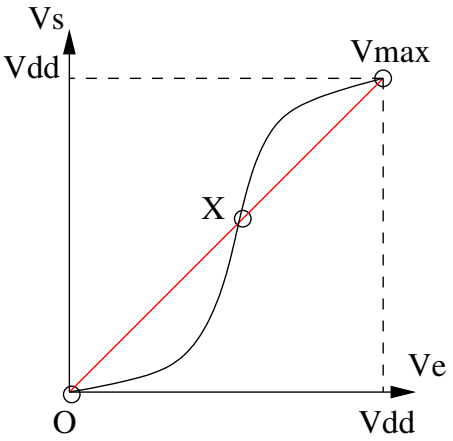



**Figure 4.7:** *Fonction de transfert de l'amplificateur*

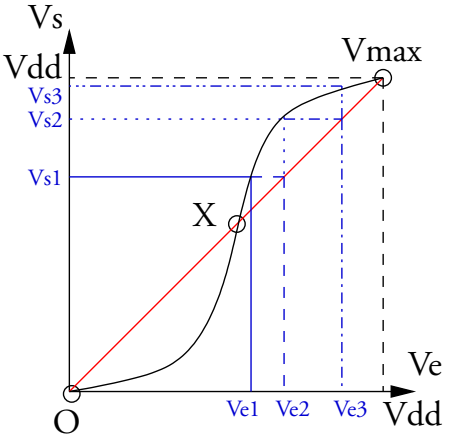



**Figure 4.8:** *Convergence vers un état stable en ne partant pas de X*

### 4.2.2 structure avec 2 inverseurs tête bêche : bascule RS et RAM statique

En pratique, l'amplificateur est réalisé avec 2 inverseurs en tête bêche comme représenté sur la figure 4.9

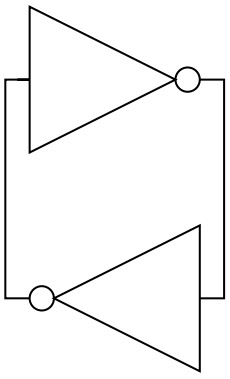


**Figure 4.9:** *Inverseurs en tête bêche pour la mémorisation*

Pour initialiser le point mémoire, il faut forcer un niveau sur l'entrée d'un inverseur. La

bascule RS (Reset Set) consiste à utiliser une initialisation logique avec des signaux pilotant des portes NAND ou NOR comme illustré dans la figure 4.10. Par exemple pour la bascule RS à base de NAND, les entrées $\overline{RESET}$ et $\overline{SET}$ sont complémentaires et actives à 0. Quand $\overline{RESET}$ est actif (au niveau 0) , et $\overline{SET}$ inactif (au niveau 1), la sortie $Q$ est initialisée à 0. Quand $\overline{RESET}$ et $\overline{SET}$ sont inactifs (au niveau 1) la bascule RS est en mode mémoire et garde la valeur préalablement initialisée.

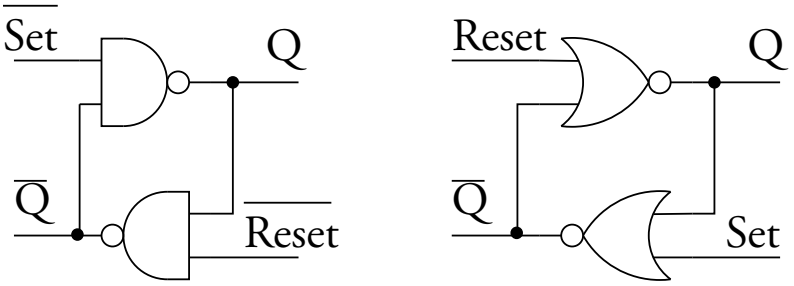



**Figure 4.10:** *Bascule RS avec une structure NAND et NOR*

Dans le cas du point mémoire RAM statique, l'initialisation est effectuée grâce à 2 transistors NMOS de part et d'autre des 2 inverseurs comme le montre la figure 4.11. Lorsque la commande C est activée, la valeur de D et son complément sur $\overline{D}$ sont écrites dans le point mémoire. Cette opération nécessite des transistors NMOS plus gros que ceux des inverseurs pour imposer électriquement un niveau pouvant être différent au départ. Si D est flottant ou en *haute impédance*, la valeur du point mémoire apparaît sur D et son complément sur $\overline{D}$, ce qui permet d'effectuer une lecture de ce point. La commande C est issue d'un décodage de signaux d'adresse sur $N$ bits permettant d'accéder à un seul point parmi $2^N$.

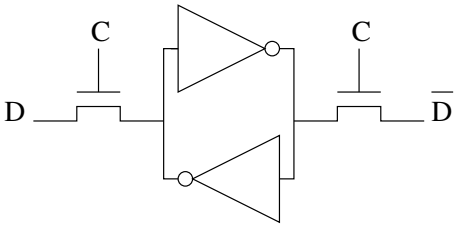



**Figure 4.11:** *Point mémoire RAM statique*

### 4.2.3 De la bascule RS à la bascule D sur état : le *latch*

La bascule RS peut évoluer vers la bascule D qui voit les 2 entrées $Reset$ et $Set$ remplacées par une unique entrée $D$. De façon à avoir la mémorisation quand $Reset = Set = 0$, une entrée EN est utilisée pour forcer les entrées à 0 avec 2 portes ET. Ce dispositif est appelé *Latch à entrée D*. Quand EN vaut 1, il y a recopie de l'entrée sur la sortie, le latch est *transparent*, et quand EN vaut 0, il est en mémorisation. La structure du latch est illustrée par la figure 4.12 :

### 4.2.4 La bascule D sur front ou *Flip-Flop*

Dans le paragraphe 4.1.2, nous avons vu que la synchronisation des variables internes et des sorties permet de fiabiliser les calculs. Cette méthode nécessite une mémoire mise à jour par

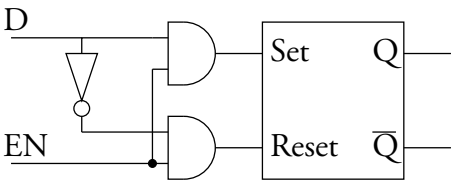



**Figure 4.12:** *Structure du* latch à entrée D

une horloge lors de l'instant d'échantillonnage. En conséquence, le latch ne convient pas pour cette méthode car il perd sa fonction de mémorisation durant une phase de l'horloge, quand il est *transparent*. Il faut nécessairement utiliser une bascule D sur front aussi appelée *Flip-Flop* ou tout simplement *bascule D*. La bascule D peut être obtenue avec 2 latches en cascade (figure 4.13 disposant d'entrées EN complémentaires. Cette bascule permet de mémoriser et d'échantillonner la valeur de la variable d'entrée sur une transition du signal EN.

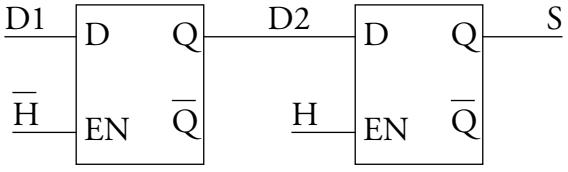



**Figure 4.13:** *Structure de la bascule D à partir de latches*

Le chronogramme de la figure 4.14 illustre le fonctionnement de la bascule D. La sortie S ne change qu'après échantillonnage sur front montant de l'horloge H, et est mémorisée pendant une période d'horloge. Ces points importants sont à noter :

- La sortie S ne change pas immédiatement après le front montant de H car la bascule D a son propre temps de propagation.
- Si D1 change entre 2 fronts montants d'horloge (cas des valeurs e0 et e2), elles ne sont pas prises en compte dans la bascule. Seules comptent les valeurs de D1 au moment du front montant d'horloge.

En pratique, une technique consiste à utiliser respectivement pour les 2 latches, 2 horloges $\phi 1$ et $\phi 2$ non recouvrantes correspondant aux 2 phases de l'horloge.

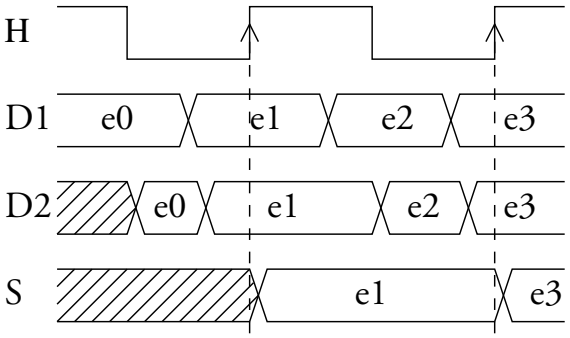



**Figure 4.14:** *Chronogramme de la bascule D avec 2 latches*

La bascule D peut disposer optionnellement d'entrées de mise à 1 $Preset$ ou mise à 0 $Reset$ ou $Clear$. Ces entrées sont asynchrones, c'est à dire actives immédiatement sans attendre le front montant de l'horloge. Ces entrés asynchrones sont actives à 0 , ce qui est indiqué

sur le symbole de la bascule par un cercle signifiant l'inversion de polarité. Le symbole de la bascule D est illustré en figure 4.15 et ses fonctions par la table 4.1. La bascule D peut avoir une sensibilité au front descendant plutôt qu'au front montant de l'horloge. Dans ce cas le symbole de la bascule dispose d'un cercle sur l'entrée de l'horloge signifiant l'inversion de polarité.

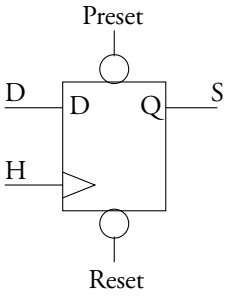



**Figure 4.15:** *Symbole de la bascule D*

| D | H | Preset | Reset | Q | Etat |
|---|---|---|---|---|---|
| 0 | ↑ | 1 | 1 | 0 | échantillonnage |
| 1 | ↑ | 1 | 1 | 1 | |
| X | 0 | 1 | 1 | Q | mémorisation |
| X | 1 | 1 | 1 | Q | |
| X | X | 0 | 1 | 1 | forçage à 1 |
| X | X | 1 | 0 | 0 | forçage à 0 |

**Table 4.1:** *Fonctions de la bascule D*

### 4.2.5 Conditions d'utilisation de la bascule

De façon à éviter les états métastables (cf paragraphe 4.2.1) de la bascule, les constructeurs spécifient une fenêtre temporelle autour de l'instant d'échantillonnage, dans laquelle la variable d'entrée ne doit pas changer de valeur. 2 temps sont utilisés à cette fin :

- $t_{su}$ : temps de prépositionnement ou *set up* : temps durant lequel les données doivent rester constantes avant le front montant d'horloge.
- $t_h$ : temps de maintien ou *hold* : temps durant lequel les données doivent rester constantes après le front montant d'horloge.

Le temps de propagation des bascules $t_{co}$ correspond au temps séparant le front d'horloge des données stabilisées en sortie. La figure 4.16 illustre les caractéristiques temporelles de la bascule D.

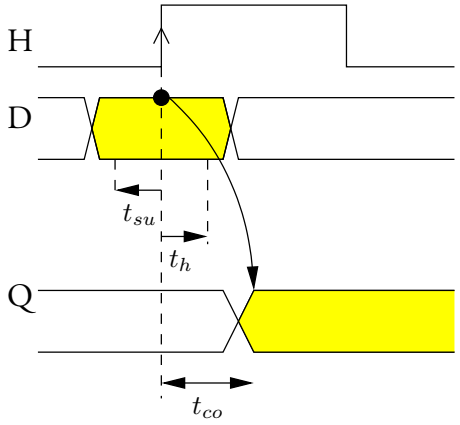



**Figure 4.16:** *Caractéristiques temporelles de la bascule D*

Dans le calcul du chemin critique, les temps de propagation $T_{co}$ et de prépositionnement $T_{su}$ doivent être pris en compte, comme indiqué dans la figure 4.17. Si $T_h$ est la période d'horloge alors il faut respecter :

$$T_h > T_{co} + T_{crit} + T_{su}$$

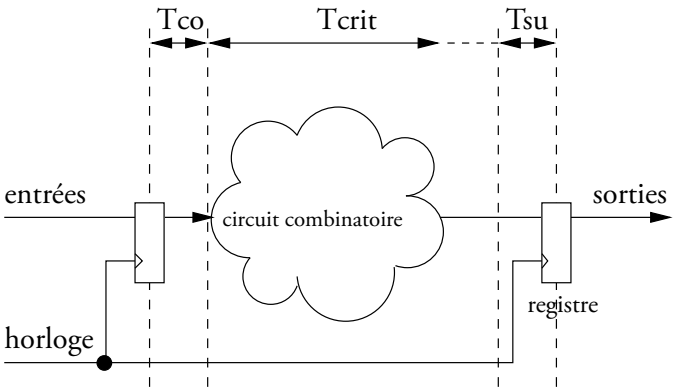



**Figure 4.17:** *Temps de propagation à considérer en logique séquentielle*

## 4.3 Exemples fondamentaux de la logique séquentielle synchrone

### 4.3.1 Le mécanisme de décalage avec un registre à décalage

Un registre est par définition un ensemble de bascules. Un registre à décalage est constitué de N bascules en cascade comme indiqué dans la figure 4.18.

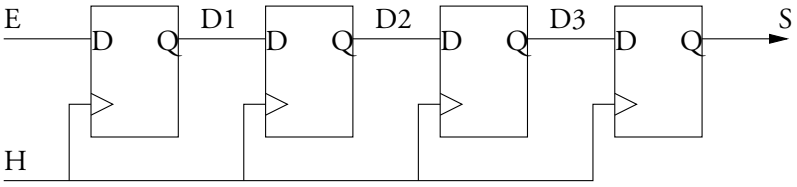



**Figure 4.18:** *Registre à décalage*

À chaque front d'horloge, le contenu de chaque bascule amont est décalé dans la bascule aval. Ainsi au bout de N fronts montants d'horloge, la première valeur rentrée se retrouve en sortie comme représenté dans la figure 4.19. Le fonctionnement correct du registre impose d'avoir un temps de propagation $T_{co}$ supérieur au temps de maintien $T_h$.

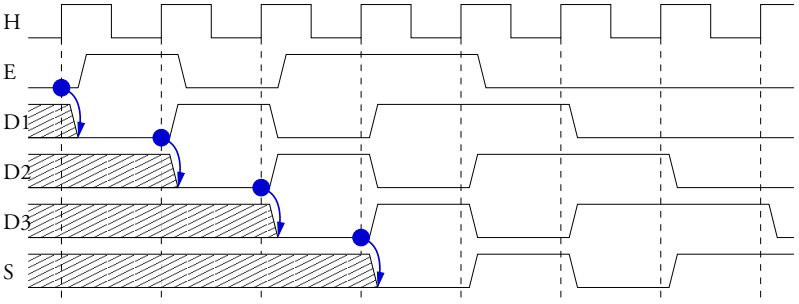



**Figure 4.19:** *Chronogramme du registre à décalage*

Le registre à décalage est une structure importante de la logique séquentielle qui permet de réaliser beaucoup d'opérations élémentaires :

- Passage d'un format série à un format parallèle : les bits rentrent en série et les N bits du registre sont les sorties.
- Passage d'un format parallèle à un format série : Les bascules sont initialisées par un mot d'entrée et la sortie s'effectue sur la dernière bascule.
- Recherche d'une chaîne de bits particulière : les sorties des bascules sont comparées avec la chaîne de référence.
- Compteur Johnson : le registre est rebouclé sur lui même et ne contient qu'un seul '1' qui boucle.
- Et encore : générateur de nombres pseudo aléatoires (LFSR *Linear Feedback shift register*), filtres numériques RIF,...

### 4.3.2 Le mécanisme de comptage

Le compteur est un opérateur très fréquemment utilisé en électronique numérique. Dans sa version la plus simple il s'agit d'un dispositif dont la sortie représente une donnée numérique qui augmente d'une unité à chaque période d'horloge. Celui que nous allons présenter possède une sortie codée sur 3 bits et enchaîne la séquence :

0, 1, 2, 3, 4, 5, 6, 7, 0, 1, ...

Le compteur est muni d'une horloge qui rythme son fonctionnement et d'un signal optionnel de remise à zéro asynchrone, RAZ. Pour réaliser ce compteur il suffit de se munir d'un incrémenteur combinatoire de 3 bits (opérateur qui réalise +'001') et d'un registre 3 bits comme l'illustre la figure 4.20. A chaque période d'horloge, l'état futur du compteur est égal à son état courant augmenté de 1. L'initialisation du compteur se fait par le signal RAZ sur l'entrée Reset asynchrone des 3 bascules.

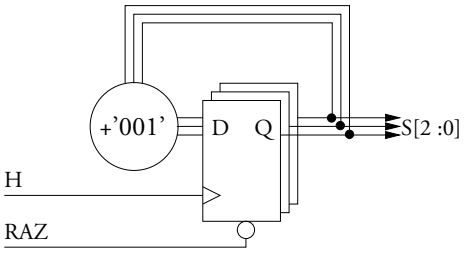



**Figure 4.20:** *Compteur binaire*

### 4.3.3 Principe de sérialisation des calculs

La logique séquentielle permet de sérialiser les opérations sur N opérandes. A chaque période d'horloge, une seule opération est effectuée entre un nouvel opérande sur l'entrée, et un résultat intermédiaire stocké dans un registre. La figure 4.21 représente un additionneur de mots de 16 bits dont la somme totale ne dépasse pas 255. Cet opérateur s'appelle aussi *accumulateur*. Il faut noter dans le circuit proposé qu'il manque la fonction d'initialisation et que l'additionneur utilisé est sur 8 bits, ce qui limite la dynamique de la variable accumulée à 255.

Le compteur binaire étudié préalablement est une version simplifiée de l'accumulateur où tous les mots à accumuler sont remplacés par une constante égale à 1. Dans les microprocesseurs, les opérations sont effectuées dans un accumulateur capable d'effectuer la plupart des opérations arithmétiques et logiques.

Le chronogramme des opérations est indiqué dans la figure 4.22 où il est supposé que la valeur initiale est 0.

### 4.3.4 Principe d'accélération des calculs par la mise en *pipeline*

Le débit de calcul est une caractéristique importante des systèmes électroniques, en particulier dans le domaine des télécommunications. Les données à traiter arrivent d'une façon synchrone à un rythme $f$. Pour traiter ces données il suffit d'utiliser un circuit séquentiel utilisant une horloge de fréquence $f$ comme illustré dans la figure 4.23 où $\phi$ représente l'opération à effectuer.

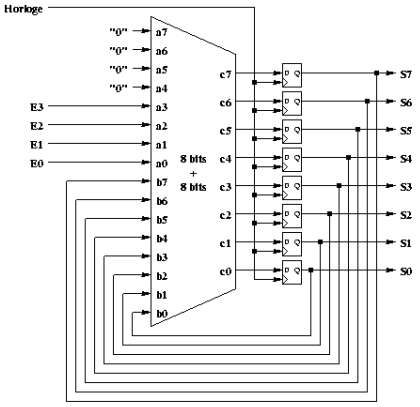



**Figure 4.21:** *Accumulateur*

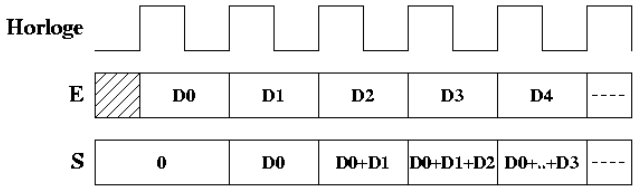



**Figure 4.22:** *Chronogramme de l'accumulateur*

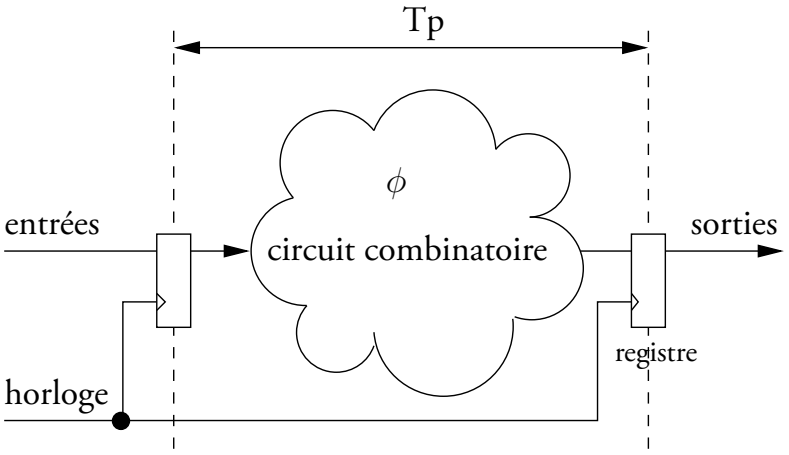



**Figure 4.23:** *Circuit Céquentiel de traitement de flot de données*

Pour que l'opérateur fonctionne correctement il suffit que :

$$T_h = 1/f \text{ et } T_p < 1/f \text{ , où}$$

- $f$ est le débit de calcul
- $T_h$ est la période d'horloge
- $T_p$ est le temps de propagation du chemin critique, incluant celui des bascules

Si cette condition n'est pas respectée, des solutions architecturales existent en logique séquentielle pour accélérer le débit des calculs. Une méthode consiste à faire une mise en *pipeline*. Si par exemple $T_p$ est 3 fois trop long : $Tp = 3/f$, on peut décomposer $\phi$ en 3 sous-fonctions

cascadables $\phi 1 * \phi 2 * \phi 3$. On obtient alors la structure de la figure 4.24. Celle ci ne résout toujours pas le problème de non respect de la contrainte de débit de $f$.

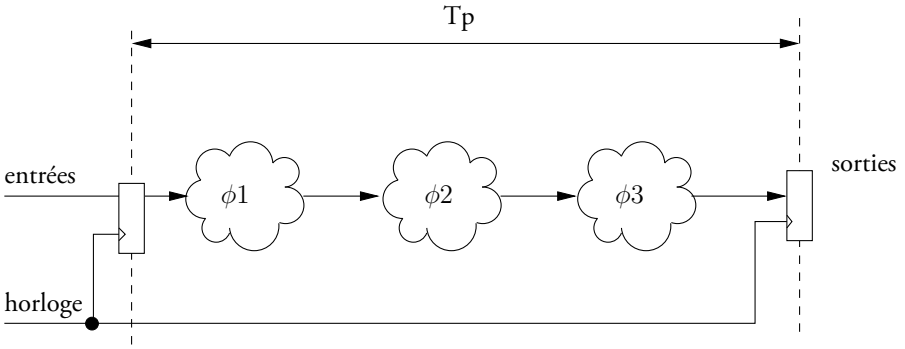



**Figure 4.24:** *Circuit séquentiel de traitement de flot de données après décomposition en sous fonctions*

Si les 3 circuits réalisant les sous-fonctions ont des temps de propagation identiques de $Tp/3$ , le temps de propagation du chemin critique devient $T_p/3$ à la place de $T_p$ en plaçant des registres entre ces circuits. Cette mise en pipeline permet ainsi de respecter la contrainte de débit. La figure 4.25 illustre cette nouvelle structure avec les étages de pipeline correspondant aux registres rajoutés.

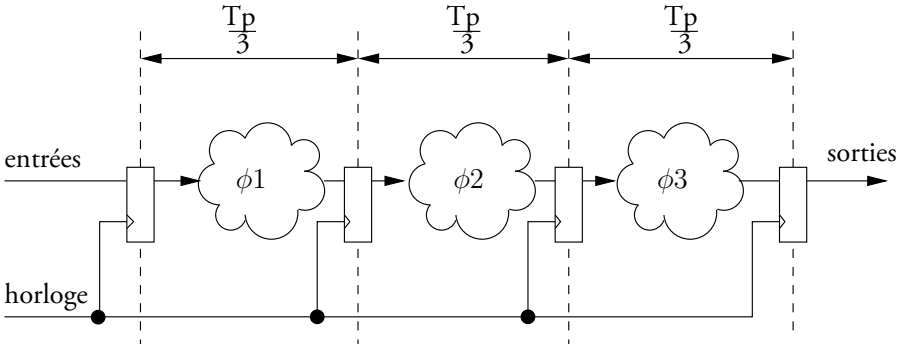



**Figure 4.25:** *Circuit séquentiel de traitement de flot de données après décomposition en sous fonctions*

Par définition la latence de calcul est le nombre de cycles pour obtenir le résultat du calcul. Dans l'exemple ci-dessus, la latence est passée de 1 à 3. Une plus grande latence ne signifie pas un retard absolu plus grand car la période d'horloge est d'autant diminuée.

En pratique, si on veut générer $N$ étages de pipeline, il est souvent difficile d'équirépartir les temps de propagation en $T_p/N$ et il faut aussi prendre en compte les temps de propagation et de prépositionnement des bascules. Donc la décomposition en $N$ étages de pipeline permet de gagner en débit un facteur un peu inférieur à $N$.

# Chapitre 5

# Machines à états

## 5.1 Introduction

Les machines à états sont des circuits de logique séquentielle (cf chapitre 4) servant exclusivement à générer des signaux de commande. Il existe en effet 2 grands types de signaux en électronique :

- Signaux à traiter : les données
- Signaux pilotant le traitement : les commandes

Cette classification des signaux se retrouve au niveau des architectures des systèmes électroniques qu'on peut schématiser comme dans la figure 5.1 où la partie contrôle, générant les commandes, est dissociée de la partie opérative, traitant les données. Les 2 parties sont toujours réalisées en logique séquentielle et dans une très grande majorité des cas en logique séquentielle synchrone.

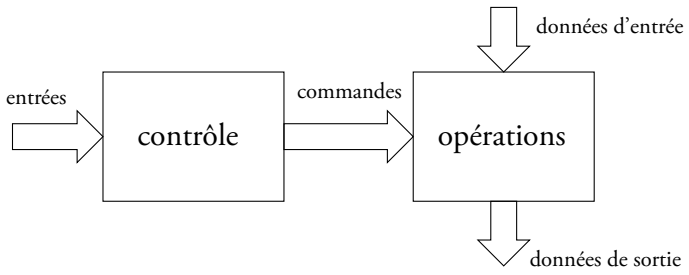



**Figure 5.1:** *Architecture générique d'un circuit électronique*

Pour la logique séquentielle synchrone, il existe 2 signaux de commandes importants :

- L'horloge : pour la synchronisation
- Le Reset : pour l'initialisation du système

La machine à état représente la partie contrôle, c'est à dire le *cerveau* du système électronique et la partie opérative , les *jambes*.

Il existe beaucoup de déclinaisons de cette architecture, des plus compliquées comme les microprocesseurs qui ont plusieurs machines à états et plusieurs parties opératives, des plus simples mais tout aussi importantes comme les contrôleurs d'ascenseurs ou de machine à café. Pour ce dernier type de système, les données sont inexistantes car les commandes servent à piloter des actionneurs, valves et moteurs,...

Les *états* de la machine à états représentent toutes les valeurs que peuvent prendre les variables internes du circuit de logique séquentielle (cf chapitre 4). Le schéma de la machine à états générique est représenté en figure 5.2

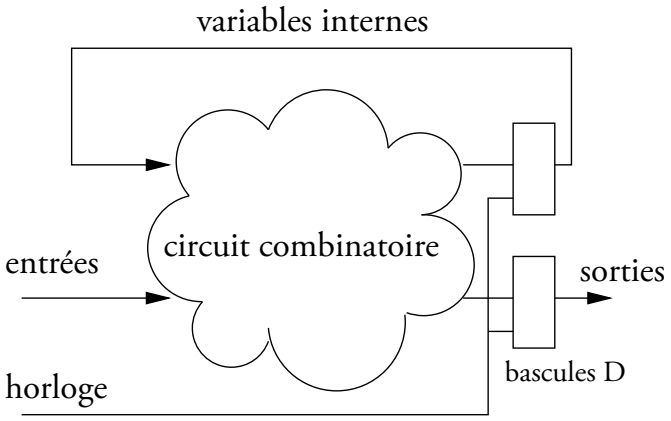



**Figure 5.2:** *Schéma d'un machine à état générique*

Par exemple pour la machine à café, les états peuvent être :

1. Attente de pièce
2. Descendre le gobelet
3. Verser la poudre de café
4. Verser l'eau chaude
5. Indiquer que c'est prêt

Cette machine peut se compliquer en prenant en compte : le choix de la boisson, le dosage du sucre, mais elle reste néanmoins très simple par rapport à certaines machines à états industrielles comme la conduite d'une centrale nucléaire, ou l'automatisation d'un usine de production. D'autres types de machines à états ont des contraintes de performances très grandes, c'est la cas de celles utilisées dans les microprocesseurs ou des processeurs spécialisées pour le graphisme ou les télécommunications.

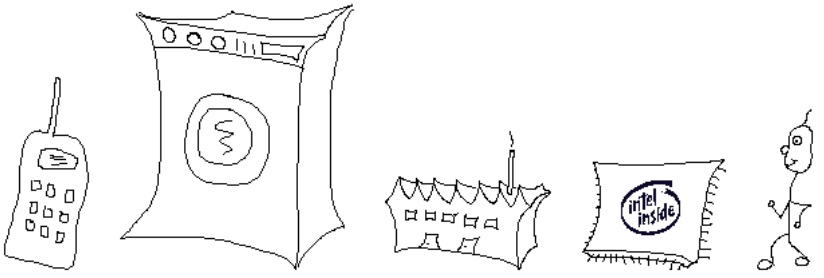



**Figure 5.3:** *Où rencontrer les machines à états*

## 5.2 Le graphe d'états

### 5.2.1 Comment représenter graphiquement le comportement d'une machine à états ?

Dans une machine à états donnée, la loi d'évolution de l'état n'est évidemment pas aléatoire, pas plus que celle qui détermine la valeur des sorties. Ces lois sont soigneusement choisies par le créateur de la machine afin que celle-ci remplisse une fonction précise. La conception d'une machine à états, pour peu que sa complexité dépasse celle des cas d'école qui nous serviront d'exemples, est une tâche délicate. Le graphe d'états est l'un des outils les plus utilisés pour la spécification de la machine à états (entrées, sorties, fonctionnement souhaité).

Le graphe d'états, comme son nom l'indique, représente graphiquement les états d'une machine à états. Chaque état est dessiné sous la forme d'une bulle contenant son nom. On comprend immédiatement que cet outil ne sera pas d'un grand secours lorsque le nombre d'états de la machine dépassera quelques dizaines. Prenons l'exemple d'une machine à laver où on considère 5 états comme illustré dans la figure 5.4.

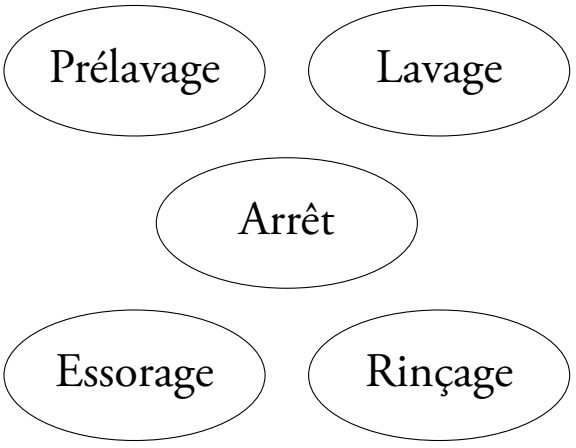



**Figure 5.4:** *Graphe d'état au départ*

On complète le graphe en figurant les transitions possibles par des flèches entre les états. On appelle état *source* l'état de départ d'une transition et état *destination* l'état d'arrivée. La transition $T0$ a *Prélavage* pour état source et *Lavage* pour état destination. Certaines transitions ont le même état pour source et pour destination. Cela signifie que la machine peut rester dans le même état pendant au moins 2 périodes successives. La transition $T1$ est de cette sorte comme illustré dans la figure 5.5.

Muni de toutes les transitions possibles comme représenté dans la figure 5.6, le graphe constitue une représentation assez dense de l'évolution possible de la machine au cours du temps. À tout instant la machine est dans l'un des états représentés ; c'est ce que nous appellerons l'état courant de la machine. A chaque front montant de l'horloge, la machine emprunte l'une des transitions possibles à partir de son état courant. Elle change alors d'état. Retenez bien cette conséquence du fait que notre machine est synchrone sur front montant de l'horloge : elle reste dans un état donné (une bulle du graphe) pendant le temps qui sépare deux fronts montants de l'horloge. Les transitions (les flèches du graphe), en revanche, sont quasi-instantanées puisqu'elles correspondent aux fronts montants de l'horloge.

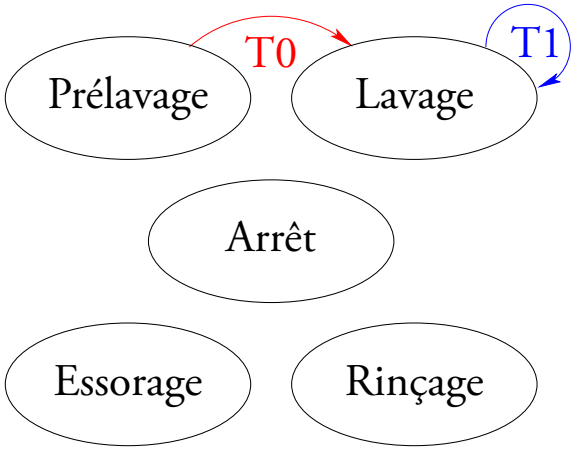



**Figure 5.5:** *Graphe d'état avec quelques transitions*

Pour comprendre notre graphe nous devons préciser les lois d'évolution des variables internes en fonction des entrées. Supposons que les entrées de notre machine soient au nombre de trois :

- M : variable booléenne qui traduit la position du bouton Marche/Arrêt du lave-linge.
- P : variable booléenne qui indique si le programme de lavage sélectionné par l'utilisateur comporte ou non une phase de prélavage.
- C : valeur en minutes d'un chronomètre qui est remis à zéro automatiquement au début de chaque étape de lavage.

Les durées des différentes étapes de lavage sont fixées par le constructeur :

- prélavage : 10 minutes
- lavage : 30 minutes
- rinçage : 10 minutes
- essorage : 5 minutes

À partir de ces informations nous pouvons faire figurer sur le graphe les conditions logiques associées à chaque transition. Ainsi le graphe de la figure 5.6, spécifie complètement le fonctionnement de la machine. On sait par exemple que lorsque la machine est dans l'état *Arrêt* elle y reste tant que M n'est pas vrai au moment d'un front montant de l'horloge. Dès que M est vrai au moment d'un front montant de l'horloge la machine change d'état : elle passe dans l'état *Prélavage* si P est vrai et dans l'état *Lavage* si P est faux. Il est important de comprendre que la valeur des entrées de la machine n'a d'importance qu'au moment précis des fronts montants de l'horloge. C'est une conséquence du fait que notre machine est synchrone sur front montant de l'horloge.

Notre machine à états possède des entrées mais nous n'avons pas encore étudié les sorties. Or un circuit électronique sans sorties n'est que de peu d'utilité. Il existe deux sortes de machines à états : celles dont les sorties ne dépendent que de l'état courant (ce sont les machines dites de Moore) et celles dont les sorties dépendent de l'état courant et des entrées (ce sont les machines dites de Mealy). Nous allons nous concentrer sur les machines de Moore. Le programmateur de notre lave-linge est une machine de Moore dont les sorties ne dépendent que de l'état courant. Nous supposerons que ses sorties sont trois signaux booléens, X, Y et Z destinés à piloter les différents moteurs du lave-linge. Les spécifications précisent leur valeur pour chaque état que peut prendre la machine. Nous pouvons encore compléter le graphe

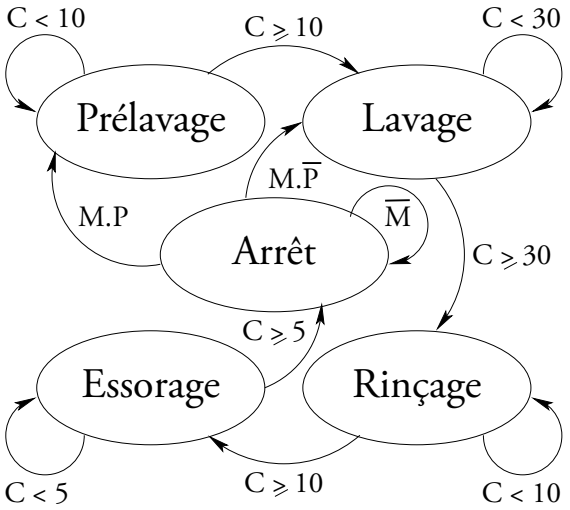



**Figure 5.6:** *Graphe d'état avec les transitions étiquetées par les valeurs des entrées*

d'états afin d'y faire figurer cette information. Le graphe est alors achevé comme illustré dans la figure 5.7. Il est équivalent aux spécifications du programmateur tout en étant plus dense qu'une description en langage naturel.

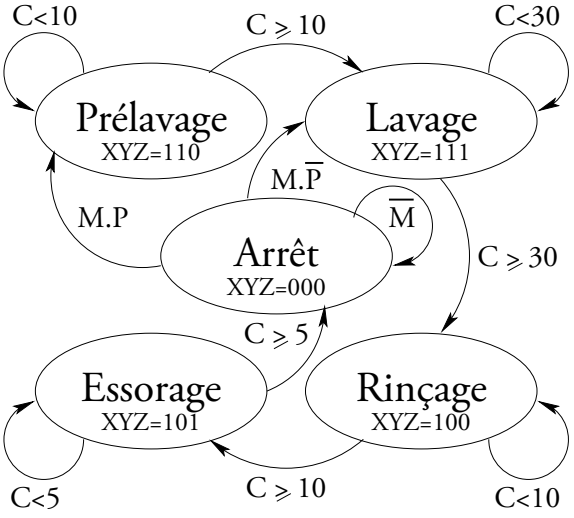



**Figure 5.7:** *Graphe d'état final*

### 5.2.2 Comment vérifier cette représentation à l'aide de quelques règles simples ?

Les spécifications sont généralement écrite en langage naturel. La traduction des spécifications en graphe d'état est donc entièrement manuelle et les risques d'erreurs sont nombreux. Si une erreur venait à se glisser dans le graphe elle se retrouverait dans le circuit électronique final, ce qui est inacceptable : un lave-linge qui "oublie" de rincer n'est pas très satisfaisant, sans parler des risques plus graves comme ceux liés aux centrales nucléaires ou aux avions de ligne.

Il faut donc vérifier le graphe avant de poursuivre la réalisation de la machine. Comme pour toute bonne spécification, le graphe doit vérifier deux propriétés fondamentales :

1. il doit être *complet*
2. il doit être *non contradictoire*

1) "Graphe complet" signifie qu'il y a toujours une transition possible. Le comportement est toujours défini : à chaque front montant d'horloge, quel que soit l'état dans lequel se trouve la machine et quelles que soient les valeurs des entrées, l'état suivant existe. Une au moins des conditions associées aux transitions doit être vraie. On peut traduire cette propriété sous forme d'équation booléenne en écrivant que le OU logique de toutes les conditions associées aux transitions partant d'un état quelconque est toujours vrai : soient $C1$, $C2$, ..., $Ci$, ..., $Cn$ ces conditions, alors :

$$\sum_{i=1}^{i=n} C_i = 1$$

Par exemple, pour le programmateur de notre lave-linge, les transitions partant de l'état *Arrêt* sont au nombre de trois comme indiqué en pointillé sur la figure 5.8

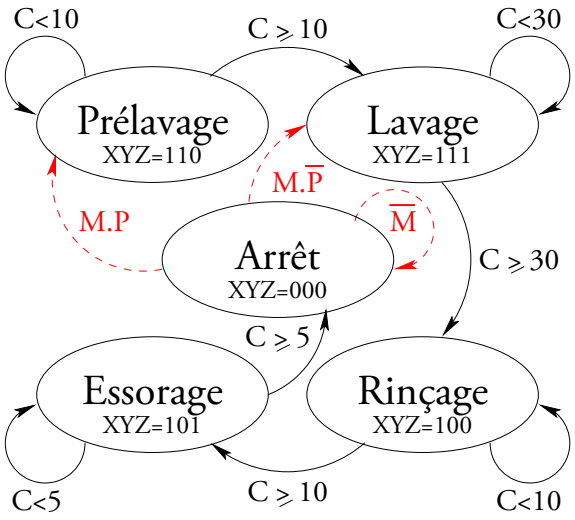



**Figure 5.8:** *Graphe d'état final*

Et les conditions associées sont :
$\overline{M}, M.\overline{P}, M.P$
Le OU logique de ces trois conditions vérifie donc :
$\overline{M} + M.\overline{P} + M.P = \overline{M} + M.(\overline{P} + P) = \overline{M} + M = 1$
L'état *Arrêt* respecte donc la première règle. A titre d'exercice vous pouvez vérifier que c'est également le cas pour les quatre autres états.

2) "Non contradictoire". La deuxième règle signifie qu'à tout front montant d'horloge une seule transition est possible. Si plus d'une transition a sa condition associée vraie, le graphe est contradictoire (deux actions incompatibles sont simultanément possibles). Le respect de cette règle est plus difficile à vérifier : le OU logique de tous les ET logiques de deux conditions associées aux transitions différentes partant d'un même état quelconque est toujours faux :

$$\sum_{i=1}^{i=n} \sum_{j \neq i} C_i.C_j = 0$$

En reprenant l'état *Arrêt* du programmateur de lave-linge comme exemple :
$\overline{M}.M.\overline{P} + \overline{M}.M.P + M.\overline{P}.M.P = 0 + 0 + 0 = 0$

L'état *Arrêt* respecte donc également la deuxième règle. Si elle est aussi vérifiée par les autres états alors nous sommes en présence d'un véritable graphe de machine à états *complet* et *sans contradiction*. Malheureusement cela ne prouve pas que le graphe est conforme à la spécification. Il faut encore vérifier que la fonctionnalité est la même dans les deux descriptions. Il n'existe pas d'outils de vérification ou de formules logiques permettant de le faire. Vous pouvez par exemple parcourir le graphe état par état et, pour chacun d'eux, comparer la partie de spécification qui le concerne avec les conditions associées aux transitions sortantes. Toute méthode est bonne si elle permet d'éviter des erreurs à ce stade du travail de conception.

## 5.3 La composition d'une machine à états

### 5.3.1 Le calcul de l'état futur

En logique séquentielle synchrone, l'état courant est modifié à chaque front montant de l'horloge. Entre deux fronts montants de l'horloge (pendant une période d'horloge) il reste stable, ce qui donne le temps aux circuits combinatoires qui composent la machine de calculer le prochain état et les sorties. Il existe donc, entre autres, un circuit combinatoire chargé de calculer le prochain état, que nous appellerons aussi état futur, à partir de l'état courant et des entrées de la machine. Ce circuit (nommé P1 sur le schéma de la figure 5.9) est en général le plus difficile à concevoir. Ses entrées sont :

- L'état courant qui est mémorisé dans le registre d'état (RE sur le schéma).
- Les entrées de la machine.

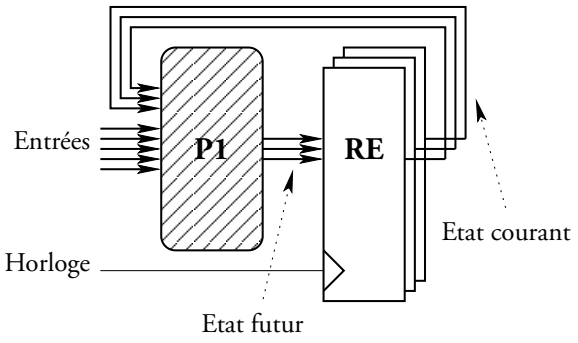



**Figure 5.9:** *Calcul de l'état futur*

Sa sortie est l'état futur.

Dès que les entrées changent de valeur ou dès que l'état courant est modifié, le circuit P1 commence à calculer l'état futur. Ce calcul n'est pas instantané (voir le TD 8 sur le temps de propagation dans les portes CMOS). Pour que la machine puisse fonctionner correctement il faut que les entrées de ce circuit restent stables pendant une durée suffisante pour que sa sortie puisse, elle aussi, s'établir et se stabiliser avant le front montant de l'horloge suivant. Sinon la valeur échantillonnée par le registre d'état ne sera pas la bonne et le déroulement des opérations sera perturbé.

### 5.3.2 Le registre d'état

Il est composé de plusieurs bascules D (la question de leur nombre exact est traitée dans le paragraphe 5.4). L'horloge est la même pour toutes : c'est l'horloge générale du circuit électronique dont fait partie la machine. Son entrée est la sortie du circuit P1, c'est l'état futur. Sa sortie, l'état courant, sert d'entrée à P1 mais aussi au circuit destiné à calculer les sorties.

Une machine à état est un dispositif avec rétroaction : l'état courant conditionne les états futurs. Dans un tel dispositif la question des conditions initiales se pose. En d'autres termes, pour que le fonctionnement soit celui souhaité dès la mise sous tension, il faut introduire un moyen de forcer un état de départ. Il en va de même pour le microprocesseur qui constitue l'unité de calcul de votre ordinateur. Comme nous l'avons vu dans le paragraphe 5.1 il contient un grand nombre de machines à états qui le commandent et le contrôlent. Si, lorsque vous allumez votre ordinateur l'état de ces machines n'est pas forcé à une valeur connue et choisie par les concepteurs, la séquence de démarrage risque d'être fortement perturbée. C'est pourquoi toute machine à état dispose d'une entrée d'initialisation $Reset$ grâce à laquelle l'état des machines est forcé lors de la mise sous tension.

Il existe deux méthodes pour forcer l'état initial avec le $Reset$ :

1. Le reset synchrone. Il est pris en compte uniquement sur le front montant de l'horloge. Il agit donc de la même façon que les entrées "normales" de la machine. Son influence est prioritaire sur les autres. Le circuit P1 possède donc ce signal comme entrée supplémentaire. Lorsque cette entrée est active (elle peut être active lorsqu'elle vaut 0 ou bien 1, c'est une convention à définir) l'état futur que calcule P1 est l'état initial. Au front montant d'horloge suivant la machine passe donc dans cet état. Dans l'exemple de notre programmateur de lave-linge il semble judicieux de choisir *Arrêt* comme état initial. Le graphe doit être modifié comme indiqué dans la figure 5.10 pour tenir compte du reset synchrone.

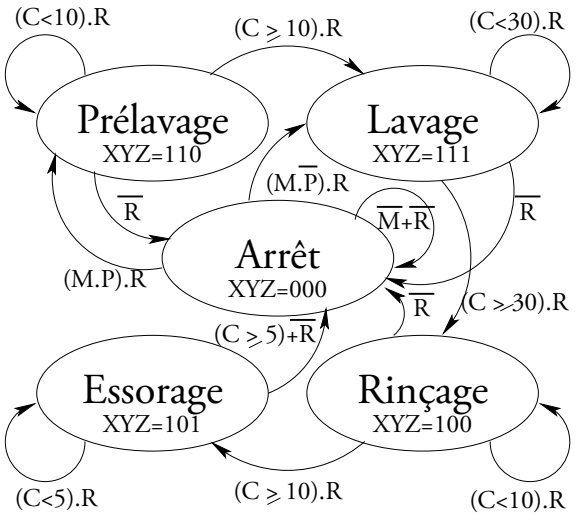



**Figure 5.10:** *Graphe d'état avec Reset synchrone*

2. Le reset asynchrone. Il utilise les entrées *Set* et *Reset* des bascules D (voir le chapitre 4) du registre d'état pour forcer l'état initial. On branche l'entrée Reset sur l'entrée *set* des bascules si on désire forcer un 1, ou sur l'entrée *Reset* des bascules si on désire forcer

un 0. Les entrées de la partie P1 ne sont pas modifiées. Le graphe d'état non plus si ce n'est l'indication de l'état de départ par le Reset comme indiqué dans la figure 5.11. Cette solution est donc plus simple à concevoir que la précédente, donne des tailles (en nombre de composants) plus faibles pour des vitesses de fonctionnement plus élevées. C'est pourquoi on la préférera lorsqu'elle n'entre pas en conflit avec d'autres contraintes.

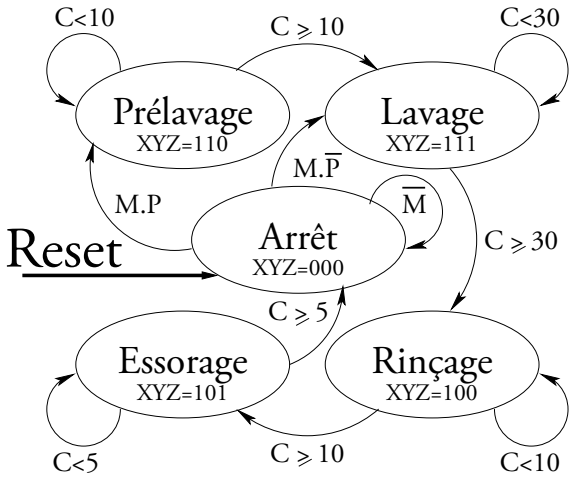



**Figure 5.11:** *Graphe d'état avec Reset Asynchrone*

### 5.3.3 Le calcul des sorties

La troisième et dernière partie d'une machine à états est le circuit combinatoire de calcul des sorties (P2 sur le schéma de la figure 5.12). Dans une machine de Moore, ses entrées sont l'état courant et ses sorties sont les sorties de la machine. Dès que l'état courant change, après un front montant d'horloge, ce circuit commence à calculer les sorties caractéristiques du nouvel état. Comme pour le circuit P1 il faut absolument qu'il dispose d'assez de temps pour le faire avant le front montant d'horloge suivant.

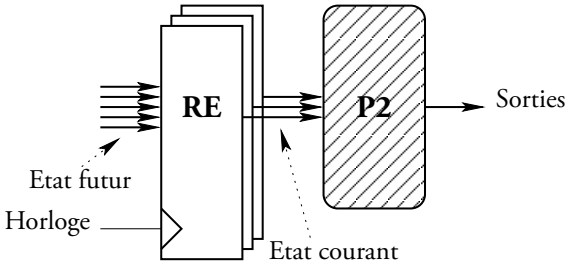



**Figure 5.12:** *Calcul des sorties*

## 5.4 Le codage des états

### 5.4.1 Comment représenter les différents états sous forme de mots binaires ?

Jusqu'ici nous avons identifié les différents états par leur nom (Arrêt, Prélavage, etc.). L'électronique numérique ne manipule pas de tels symboles. L'alphabet y est nettement plus restreint puisqu'il se compose des seuls 0 et 1 de l'algèbre de Boole. Pour chaque état d'une machine il va donc falloir trouver un nom unique exprimé dans cet alphabet. Nous avons vu dans les paragraphes 5.1 et 5.3 que les machines à états synchrones mémorisent l'état courant dans des bascules D du type de celles du chapitre 4. Chacune de ses bascules contiendra à tout moment un caractère (0 ou 1) du nom de l'état courant.

A la différence des noms d'états exprimés en langage naturel ceux exprimés dans l'alphabet binaire auront tous le même nombre de caractères. La raison en est simple : pour pouvoir mémoriser n'importe quel état dans les bascules D du circuit le nombre de bascules doit être au moins égal à la taille du nom le plus long. Si ces bascules ne servent pas toutes à un instant donné on ne peut en tirer aucun profit ni pour réduire la taille du circuit, ni pour augmenter sa vitesse. L'électronique a ceci de contraignant que le matériel inutilisé coûte aussi cher que le matériel utilisé. Nous allons continuer à exploiter l'exemple du programmateur de lave-linge. Commençons par déterminer le nombre de symboles binaires (bits) nécessaires pour représenter les cinq états. Contrairement à ce que l'on pourrait penser ce choix n'est pas trivial. Nous pouvons d'ores et déjà constater que trois bits au moins sont nécessaires. En effet, deux bits permettent, au maximum, la représentation de quatre situations différentes seulement. Trois bits permettent de représenter huit mots différents. On peut également éliminer les solutions à plus de cinq bits car elles sont forcément redondantes (il existe toujours au moins un bit inutile que l'on peut retirer en conservant cinq mots différents). Restent les solutions à trois, quatre ou cinq bits.

On appelle codage la représentation en mots binaires des noms des états. La table 5.1 propose un exemple de codage à trois, quatre, cinq et six bits pour notre exemple.

| Etat | Trois bits | Quatre bits | Cinq bits | Six bits |
|---|---|---|---|---|
| Arrêt | 100 | 0001 | 11110 | 110001 |
| Prélavage | 000 | 0110 | 10100 | 101010 |
| Lavage | 001 | 1111 | 01100 | 110111 |
| Rinçage | 010 | 0000 | 01101 | 010110 |
| Essorage | 111 | 1011 | 01110 | 010111 |

**Table 5.1:** *Exemples de codage des états*

### 5.4.2 En quoi le codage choisi influe-t-il sur la taille de la machine à états ?

La partie combinatoire de la machine qui calcule l'état futur en fonction des entrées et de l'état courant est très largement influencée par le codage des états. Donc sa taille (en nombre

de composants utilisés) en dépend également. Elle possède $Ne + Nb$ entrées et $Nb$ sorties ($Ne$ est le nombre d'entrées de la machine et $Nb$ le nombre de bits choisi pour coder les états comme illustré dans la figure 5.13).

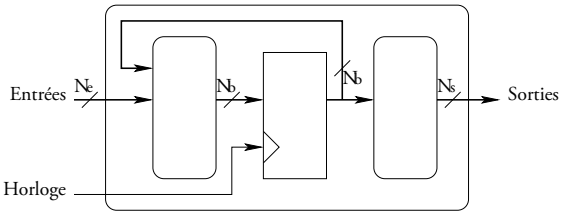



**Figure 5.13:** *Schéma d'une machine à états avec le nombre de bits nécessaires*

Le nombre de fonctions booléennes calculées est donc égal à $Nb$ et chacune de ces fonctions possède $Ne + Nb$ entrées. On pourrait en conclure qu'il faut coder les états avec le moins de bits possibles pour que cette partie combinatoire soit la plus petite possible. Mais il n'en est rien. On peut facilement trouver des exemples qui prouvent le contraire. Pour s'en convaincre il suffit de remarquer qu'une fonction booléenne de quatre variables peut être plus simple qu'une autre de deux variables :

$$F(A_0, A_1, A_2, A_3) = A_0$$

est plus simple que :

$$G(A_0, A_1) = A_0 \oplus A_1$$

Il se pourrait que notre exemple soit une illustration de ce phénomène et que cinq fonctions booléennes simples vaillent mieux que trois complexes.

La partie combinatoire qui calcule les sorties en fonctions de l'état courant possède $Nb$ entrées et $Ns$ sorties (où $Ns$ est le nombre de sorties de la machine). Elle calcule donc $Ns$ fonctions booléenne de $Nb$ entrées. Là encore, méfions nous des évidences ; la solution qui se traduit par une taille minimum n'utilise pas nécessairement un codage des états sur un nombre de bits minimum.

La seule certitude que l'on ait concerne le registre d'état. Sa taille est directement liée au nombre de bits du codage d'états. Comme on le voit, le problème n'est pas simple. Il l'est d'autant moins qu'une solution optimale au sens de la taille pour la partie combinatoire de la machine qui calcule l'état futur a peu de chances d'être également la meilleure pour la partie combinatoire qui calcule les sorties.

### 5.4.3 Quelles méthodes permettent de choisir le *meilleur* codage possible ?

Il faut, avant de répondre à cette question, déterminer ce que l'on entend par *meilleur*. La taille est un critère de sélection mais il n'est pas le seul. On peut également s'intéresser à la vitesse de fonctionnement, à la consommation ou la simplicité de conception. Selon l'objectif fixé les stratégies de codage seront différentes. Parmi celles-ci nous allons en citer trois :

1. *Le codage adjacent* : il utilise un nombre de bits minimum (trois bits pour l'exemple de la figure 5.14) et se caractérise par le fait que le passage d'un état à un autre ne

modifie qu'un seul bit du registre d'état, un peu à la manière d'un code de Gray. Il n'est pas toujours possible de trouver un tel codage. Pour notre programmateur, par exemple, il n'existe pas de codage adjacent. On peut cependant essayer de s'en approcher en réduisant autant que faire se peut le nombre de transitions modifiant plus d'un bit du registre d'état. Ici, seule la transition de l'état *Prélavage*, codé 001 à l'état *Lavage*, codé 010, ne respecte pas la contrainte.

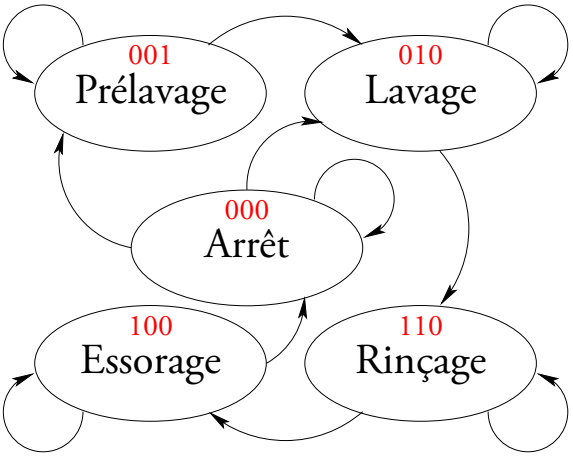



**Figure 5.14:** *Graphe avec codage adjacent*

L'intérêt d'un tel codage n'est pas systématique. Il donne cependant souvent de bons résultats en taille et en vitesse pour la partie combinatoire qui calcule l'état futur. Elle se trouve en quelque sorte simplifiée par la faible agitation des bits représentant l'état.

2. *Le codage « one-hot »* : il utilise un nombre de bits égal au nombre d'états (cinq bits pour l'exemple de la figure 5.15). Chaque état est représenté par un mot binaire dont tous les bits sauf un valent 0. Ce codage donne souvent les machines les plus simples à concevoir. Il est également parfois intéressant en vitesse et en surface malgré le handicap dû à la taille du registre d'état.

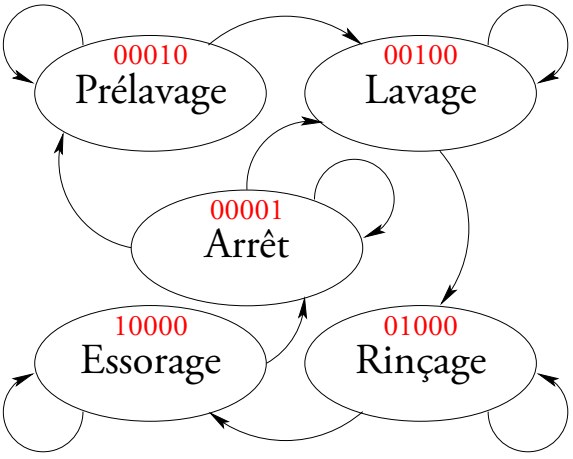



**Figure 5.15:** *Graphe avec codage "one-hot"*

3. *Le codage aléatoire* : il consiste à coder les états sur un nombre de bits minimum sans aucune autre préoccupation que d'éviter que deux états aient le même code. Les résultats

en terme de surface, vitesse ou difficulté de conception sont imprévisibles mais peuvent parfois être meilleurs que ceux produits par les deux autres stratégies.

Pour ce problème précis de l'optimisation du codage des états les outils logiciels de type synthétiseurs logiques peuvent aider le concepteur pour trouver un « bon » codage.

## 5.5 La conception d'une machine à états

Considérons l'exemple du programmateur du lave-linge (voir le paragraphe 5.2). Le graphe d'état final représenté dans la figure 5.8 fait apparaître un minuteur qui fournit en entrée de notre machine à états trois signaux C5, C10 et C30, tous trois booléens, qui indiquent respectivement si la valeur 5 minutes, 10 minutes ou 30 minutes est atteinte.

Ces minuteurs sont aussi des machines à états dont l'état change à chaque cycle d'horloge. Ils auraient pu être incorporés au graphe principal, mais en considérant un fréquence d'horloge de 1 seconde, le graphe aurait été muni de plus de 3300 états (5mn + 2fois 10mn + 30mn)* 60 s . Le chapitre 5.5.2 étudie la conception de ces minuteurs.

Les machines à états peuvent donc être *factorisables*. Cet exemple montre un exemple de machines à états imbriquées de façon à en simplifier leur conception. Commençons par concevoir la machine à états principale dont le graphe a été étudié préalablement.

### 5.5.1 machine à états principale

L'interface de la machine avec le monde extérieur est spécifié dans la table 5.2.

| Nom | Mode | Description |
|---|---|---|
| H | Entrée | Horloge |
| R | Entrée | Reset actif à 0 , initialise à l'état Arrêt |
| M | Entrée | Position du bouton Marche/Arrêt |
| P | Entrée | Existence d'une phase de prélavage |
| C5 | Entrée | Chronomètre supérieur ou égal à 5 minutes |
| C10 | Entrée | Chronomètre supérieur ou égal à 10 minutes |
| C30 | Entrée | Chronomètre supérieur ou égal à 30 minutes |
| X | Sortie | Vaut 0 dans l'état Arrêt, 1 dans les autres |
| Y | Sortie | Vaut 1 dans les états Prélavage et Lavage, 0 dans les autres |
| Z | Sortie | Vaut 1 dans les états Lavage et Essorage, 0 dans les autres |

**Table 5.2:** *Spécification de l'interface*

La première chose à faire est le graphe d'état qui a déjà été étudié au paragraphe 5.2 et vérifié pour ne pas être incomplet ni contradictoire. La figure 5.8 illustre le graphe considéré. Dans un deuxième temps le codage des états doit être choisi. Considérons le codage représenté dans la table 5.3.

Le codage des états choisi est indiqué en haut de chaque bulle du graphe représenté en figure5.16.

| Etat | Codage |
|---|---|
| Arrêt | 000 |
| Prélavage | 001 |
| Lavage | 010 |
| Rinçage | 110 |
| Essorage | 100 |

**Table 5.3:** *Codage des états*

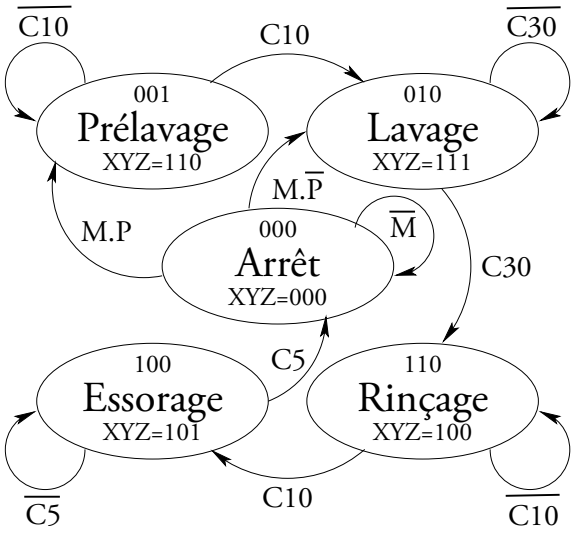



**Figure 5.16:** *Graphe avec codage choisi pour la conception*

Il faut maintenant établir la table de vérité des différentes fonctions booléennes calculées à l'intérieur de la machine.

Commençons par la partie combinatoire qui calcule l'état futur à partir de l'état courant et des entrées, que nous appellerons P1. Nous noterons les trois bits de l'état futur EF2, EF1 et EF0 avec la convention que EF2 est le bit de gauche, EF1 le bit du milieu et EF0 le bit de droite du code de l'état. De même les trois bits de l'état courant seront notés EC2, EC1 et EC0. Cette table de vérité s'appelle également table d'évolution (ou de transition) car elle décrit l'évolution de la machine au cours du temps. Elle donne pour chaque état courant possible et pour chaque combinaison possible des entrées la valeur prise par l'état futur. Lorsque la valeur d'une entrée est X cela signifie qu'elle est indifférente. La table 5.4 représente la table d'évolution de la machine à états.

En utilisant les méthodes et principes exposés dans le chapitre 2, on en déduit des équations non simplifiées des trois fonctions booléennes EF2, EF1 et EF0 que calcule P1 :

$$
\begin{aligned}
EF_2 &= (EC_2 + EC_1).(EC_2 + C30).\overline{(EC_2.\overline{EC_1}.C5)} \\
EF_1 &= (EC_1 + EC_0).(EC_1 + C10).\overline{(EC_2.EC_1.C10)} + \overline{(EC_2 + EC_1 + EC_0)}.M.\overline{P} \\
EF_0 &= (EC_2 + EC_1 + EC_0).M.P + EC_0.\overline{C10}
\end{aligned}
$$

| Etat courant | | | Entrées | | | | | Etat futur | | |
|---|---|---|---|---|---|---|---|---|---|---|
| EC2 | EC1 | EC0 | M | P | C5 | C10 | C30 | EF2 | EF1 | EF0 |
| 0 | 0 | 0 | 0 | X | X | X | X | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | X | X | X | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | X | X | X | 0 | 1 | 0 |
| 0 | 0 | 1 | X | X | X | 0 | X | 0 | 0 | 1 |
| 0 | 0 | 1 | X | X | X | 1 | X | 0 | 1 | 0 |
| 0 | 1 | 0 | X | X | X | X | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | X | X | X | X | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | X | X | X | 0 | X | 1 | 1 | 0 |
| 1 | 1 | 0 | X | X | X | 1 | X | 1 | 0 | 0 |
| 1 | 0 | 0 | X | X | 0 | X | X | 1 | 0 | 0 |
| 1 | 0 | 0 | X | X | 1 | X | X | 0 | 0 | 0 |

**Table 5.4:** *Table d'évolution*

Après simplification et toujours en utilisant les méthodes du chapitre 2 :

$$\begin{aligned} EF_2 &= EC_1.C30 + EC_2.(EC_1 + \overline{C5}) \\ EF_1 &= EC_0.C10 + EC_1.\overline{EC_2}.C10 + \overline{EC_2}.\overline{EC_1}.\overline{EC_0}.M.\overline{P} \\ EF_0 &= \overline{EC_2}.\overline{EC_1}.\overline{EC_0}.M.P + EC_0.\overline{C10} \end{aligned}$$

La réalisation en portes logiques de ces trois équations ne pose pas de problème particulier. Il peut cependant être intéressant d'affiner l'étude dans le but de réduire la complexité de l'ensemble. On peut par exemple remarquer que le terme : $\overline{EC_2}.\overline{EC_1}.\overline{EC_0}.M$ se retrouve dans les équations de $EF_1$ et $EF_0$ Il est possible de partager certaines portes entre plusieurs fonctions logiques et réaliser des économies de matériel.

### 5.5.2 Machine à états du minuteur

Chaque minuteur dispose en entrée d'un signal de commande GO correspondant à une sortie de la machine à états principale.

L'interface du minuteur avec le monde extérieur est spécifié dans la table 5.5. $C_x$ représente C5 , C10 ou C30.

Le graphe d'état de la machine à état est cyclique et reflète l'avancement de la machine quand le signal $GO$ est actif. Le graphe est illustré dans la figure 5.17.

Ce graphe correspond à la fonction d'un compteur binaire piloté par le signal GO, et dont la sortie est comparée au temps d'attente du minuteur. L'état N est le seul état où la sortie est active et où il n'y a pas de condition pour aller à l'état suivant. Autrement dit il s'agit d'un compteur modulo N+1 si GO est toujours actif.

Plutôt que d'utiliser une méthode systèmatique de synthèse de machines à états, qui déboucherait sur un grand nombre d'états (300 états pour obtenir 5mn avec une horloge d'1s de période), il suffit de considérer la structure d'un compteur binaire (cf chapitre 4 suivi d'un

| Nom | Mode | Description |
|---|---|---|
| H | Entrée | Horloge |
| R | Entrée | Reset actif à 0 , initialise à l'état Arrêt |
| *GO* | Entrée | Commande venant de la machine à état principale<br>actif à 1 pour autoriser la sortie, sinon force la sortie à 0 |
| $C_x$ | Sortie | Vaut 1 dès que le temps est atteint |

**Table 5.5:** *Spécification de l'interface*

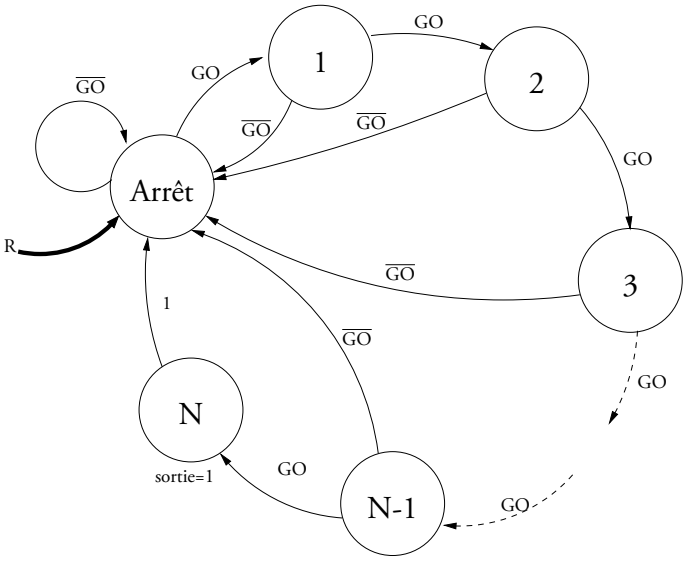



**Figure 5.17:** *Graphe avec codage choisi pour la conception*

comparateur (cf chapitre 2). Ce compteur est mis à 0 (correspondant à l'état *Arrêt* = 0) par GO et le codage des états est tel que les sorties du compteur correspondent aux bits codant l'état. La figure 5.18 représente la structure du minuteur. Le signal R agit sur le Reset asynchrone des bascules. Il aurait pu être supprimé du fait que GO effectue un Reset synchrone.

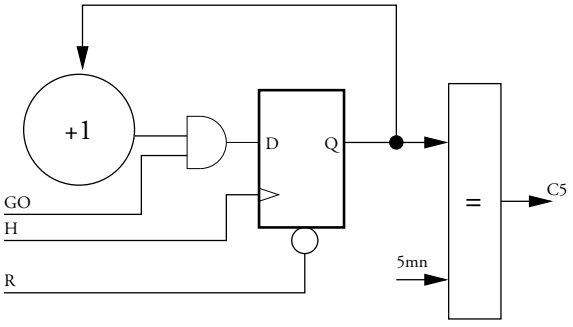



**Figure 5.18:** *Schéma d'un minuteur*

Plutôt que d'utiliser 3 minuteurs, il est possible d'avoir un seul minuteur en utilisant en entrée le temps d'attente. Dans ce cas l'interface avec l'extérieur dispose de 2 signaux supplémentaires SEL1 et SEL0 permettant de programmer le minuteur comme indiqué dans le

tableau 5.6. La spécification de la machine à état principale doit donc changer légèrement afin de :

- Sortir SEL1, SEL0 et GO plutôt que 3 signaux GO séparés
- Avoir en entrée un seul signal $C - x$ à la place de C5, C10 et C30

| SEL1 | SEL0 | Mode de programmation |
|---|---|---|
| 0 | 0 | 5mn |
| 1 | x | 10mn |
| 0 | 1 | 30mn |

**Table 5.6:** *Spécification de la programmation du minuteur*

Bien entendu les sorties de la machine à états principale doivent être modifiées en conséquence. Il existe toutefois un problème dans la machine à état principale car il n' y a pas d'état permettant de remettre le signal GO à 0, par exemple entre l'état *Lavage* et l'état *Rinçage* pour programmer le minuteur de C10 à C30.

Une solution consiste à rajouter des états dans la machine à état principale, de façon à mettre le signal GO à 0. Par exemple il y aurait l'état *Lavage-bis* tout de suite après *Lavage*, identique à *Lavage* mais avec GO à 0.

Une autre solution, certainement plus optimale en temps de développement, consiste à générer un signal de remise à zéro, RAZ, du minuteur lorsqu'il y a eu un changement sur les entrées SEL1 ou SEL0. Au vu de la séquence nécessaire (C10 $\rightarrow$ C30 $\rightarrow$ C10 $\rightarrow$ C5), il suffit de détecter le changement uniquement sur SEL1 car SEL0 ne change pas sur les transitions. Pour ce faire il suffit de comparer l'ancienne valeur de SEL1 avec la nouvelle et de mettre à zéro le minuteur si les 2 valeurs sont différentes. La figure 5.19 illustre le schéma du minuteur générique gérant automatiquement les changements de programmation.

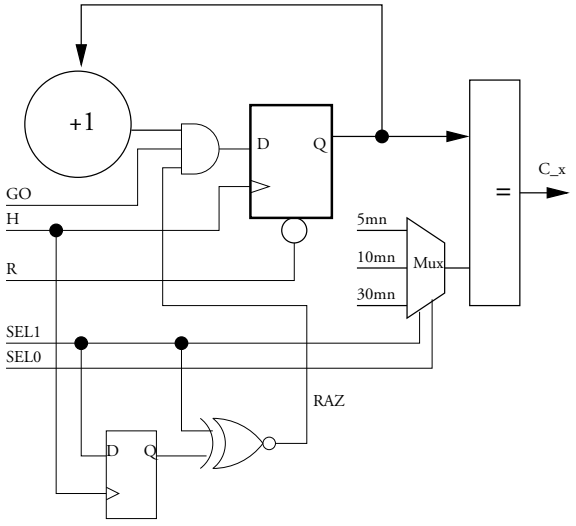



**Figure 5.19:** *Schéma d'un minuteur générique avec RAZ automatique*

# Chapitre 6

# Des machines à états aux processeurs

## 6.1 Introduction

### 6.1.1 Objectifs

Le but de ces deux leçons est de :
- vérifier que les principes des machines à états sont bien assimilés,
- introduire la notion de microprocesseur, à partir d'exemples simples et progressifs,
- concevoir un microprocesseur simple, d'architecture RISC, que vous réaliserez physiquement lors du prochain TP.

Pour cela, nous allons procéder par étapes, et le construire petit à petit…

### 6.1.2 Introduction

Les processeurs ne sont rien d'autre que des machines à calculer *programmables*. Imaginez que vous êtes comptable, et que vous avez à effectuer une série d'opérations qu'on vous a gentiment inscrites sur une feuille de papier. Voici un exemple d'instructions qu'on peut vous avoir donné :

1. faire 112 + 3
2. faire 4 + 5
3. faire 2 + 16
4. …

Un exemple un peu plus compliqué serait :

1. faire 112 + 2
2. faire “résultat précédent” * 5
3. …

ou bien

1. 112 + 3
2. résultat précédent - 4
3. si le résultat est nul, passer à l'étape 6, sinon continuer
4. 3 * 4
5. résultat précédent + 9
6. ouvrir la fenêtre
7. résultat de l'étape 2 - 15
8. passer à l'étape 12
9. …

Un microprocesseur est un dispositif électronique qui pourrait faire ce travail à votre place, pourvu qu'on lui donne la feuille de papier (et de l'énergie).

### 6.1.3 Instructions et données

Le texte d'une série d'opérations comme ci-dessus est appelé *programme*. En d'autres termes, un *programme* de microprocesseur est juste une liste d'opérations à effectuer. Dans notre cas, où le microprocesseur est simple [1], les instructions resteront simples. Si le processeur est plus complexe, incluant des périphériques multiples (gestionnaire de mémoire, entrées-sorties, …), les instructions peuvent devenir complexes, comme c'est le cas dans les processeurs CISC [2].

Dans les suites d'opérations ci-dessus, on distingue deux types d'objets :

- les données :
  - d'abord les *opérandes* proprement dits (“3”, “4”, “112”, …),
  - et les opérandes implicites (“résultat précédent”, “résultat de l'étape 2”, …) ;
- les instructions :
  - pour nous ce sont principalement les opérations arithmétiques (“+”, “−”, “*”, “/”...),
  - il y a aussi des tests (“si le résultat précédent est nul...”),
  - et des sauts (“passer à l'étape 12”), souvent conditionnés par un test (“alors passer à l'étape 6”),
  - ainsi que des instructions spéciales (“ouvrir la fenêtre”).
    Dans notre cas, une instruction de ce genre pourrait être “mettre en marche le buzzer”, ou “allumer la LED numéro 10”...

Notez que ces suites d'opérations sont numérotées : elles ont un ordre. Dans le premier exemple, l'ordre n'a pas tellement d'importance, mais il en a une dans le deuxième et le troisième quand on parle de “résultat précédent”, d'“étape 6”, …

### 6.1.4 de la feuille à l'électronique

Passons du comptable et de la feuille de papier aux composants électroniques.

La feuille de papier a un rôle de mémorisation :

- c'est sur elle qu'est écrite la suite des opérations à effectuer,

1. et plus généralement pour tous les processeurs dits *RISC*
2. “Complex Instruction Set”, par opposition à RISC : “Reduced Instruction Set”

- c'est probablement aussi sur elle que seront écrits les résultats.

Nous la modéliserons par une mémoire vive, une *RAM*. Vous avez déjà vu ce genre de composant lors du TD sur les bascules et la mémorisation (11). C'est cette RAM qui stockera les instructions à effectuer, les données, ainsi que les résultats que le microprocesseur va calculer. Le microprocesseur sera donc relié à cette RAM, et ira lire les instructions à effectuer, selon le principe suivant :

1. aller lire la première ligne (instructions et données associées)
2. faire ce qui est indiqué
3. aller lire la ligne suivante (instructions et données associées)
4. faire ce qui est indiqué
5. revenir à l'étape 3 (etc. jusqu'à ce que mort s'ensuive...)

**Première remarque** : le microprocesseur doit lire les lignes *une par une*. Il doit donc maintenir un compteur de ligne interne qui indique la ligne courante (ou la prochaine ligne à lire, comme cela nous arrangera).

**Deuxième remarque** : le processeur est un dispositif électronique qui ne comprend que des suites de bits. Il faudra donc coder les instructions sur un nombre de bit suffisant pour coder toutes les instructions dont nous aurons besoin. Les données naturelles (les nombres) seront codées de façon normale (en complément à 2, par ex.), et il faudra trouver un moyen de coder les données implicites.

**Troisième remarque** : dans notre architecture, la RAM stockera les données et les instructions de façon indifférente. Il est possible d'utiliser deux RAM différentes, ou des zones distinctes, mais vous verrez cela en détail dans le module ARSE !

### 6.1.5 Interlude rappel : fonctionnement de la RAM

Le schéma de la RAM est donné en figure 6.1.

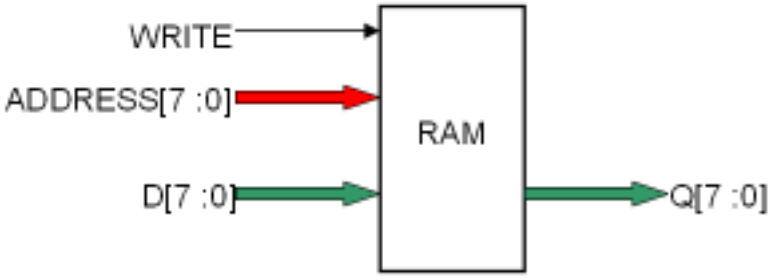



**Figure 6.1:** *Symbole de la RAM*

La RAM possède trois bus :

- un bus d'adresses, $ADDRESS[7 : 0]$ indiquant l'emplacement en mémoire de la donnée à laquelle on accède,
- un bus de données d'entrée, $D[7 : 0]$, pour les données qu'on écrit en RAM,
- un bus de données de sortie, $Q[7 : 0]$, pour les données qu'on va lire en RAM,

ainsi que

- un signal de contrôle sur 1 bit, $WRITE$, indiquant si on est entrain de faire une *lecture* dans la RAM ($WRITE = 0$), ou une écriture ($WRITE = 1$).

Le fonctionnement de la RAM est le suivant :

- la RAM sort en permanence sur $Q[]$ la donnée stockée à l'adresse présente sur $ADRESSE[]$ (Si on n'a pas envie de l'utiliser, on l'ignore),
- si $WRITE$ est actif (1), la valeur présente sur $D[]$ est écrite dans la mémoire à l'adresse présente sur $ADRESSE[]$,
- si $WRITE$ est inactif (0), $D[]$ est ignoré.
- Pendant que $WRITE$ est actif, le bus $Q[]$ prend comme valeur le contenu de la case pointée par l'adresse. Cela va donc être la copie de $D[]$, mais avec du retard.

Pour les chronogrammes, on se reportera à la figure 6.2

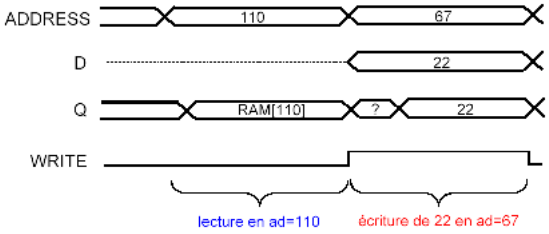



**Figure 6.2:** *Exemple d'accès à la RAM*

Nous connecterons notre microprocesseur (automate) à une RAM pouvant stocker 256 mots de 8 bits chacun :

- 8 bits : les lignes de données $D[]$ seront un bus 8 bits
- 256 mots : il nous faudra donc 8 lignes d'adresse (pour coder une adresse allant de 0 à 255)

L'architecture globale, que nous utiliserons en TP est donc la suivante (voir figure 6.3 :

- notre microprocesseur
- la RAM, reliée au processeur par ses 2 bus de données, son bus d'adresse et la ligne de WRITE
- un buzzer qui servira à jouer de la musique

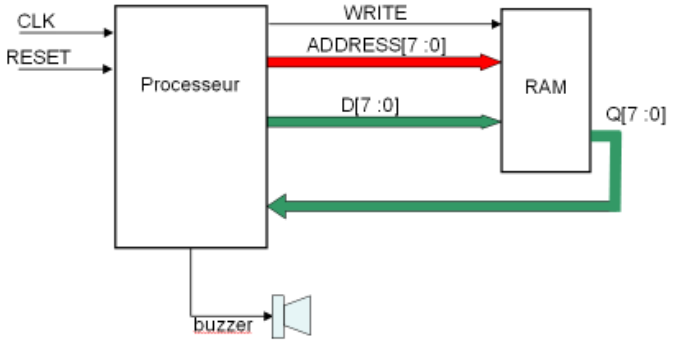



**Figure 6.3:** *Schéma global*

## 6.2 Étape 1 : automate linéaire basique

Dans cette première étape, nous n'implémenterons que les instructions et données du premier exemple. Le processeur est donc relié à une mémoire vive (RAM) stockant 256 mots de 8 bits.

### 6.2.1 Organisation de la mémoire

On suppose que le programme (opérations à effectuer) ainsi que les données sont déjà chargées dans la mémoire, et qu'ils respectent le format suivant :

| adresse | type du mot stocké | exemple |
|---|---|---|
| 0 | instruction | ADD |
| 1 | donnée (premier opérande) | 3 |
| 2 | donnée (deuxième opérande) | 4 |
| 3 | donnée (résultat) | X |
| 4 | instruction | SUB |
| 5 | donnée (premier opérande) | 12 |
| 6 | donnée (deuxième opérande) | 8 |
| 7 | donnée (résultat) | X |

Le "X" indique que la RAM contient à cet endroit là une valeur quelconque. C'est au microprocesseur d'aller y écrire le résultat correct. Après avoir lancé le microprocesseur, le contenu de la RAM sera le suivant (on indique en gras les endroits de la RAM qui ont changé) :

| adresse | type du mot stocké | exemple |
|---|---|---|
| 0 | instruction | ADD |
| 1 | donnée (premier opérande) | 3 |
| 2 | donnée (deuxième opérande) | 4 |
| 3 | donnée (résultat) | **7** |
| 4 | instruction | SUB |
| 5 | donnée (premier opérande) | 12 |
| 6 | donnée (deuxième opérande) | 8 |
| 7 | donnée (résultat) | **4** |

Nous en déduisons ce que doit faire le microprocesseur :

- le microprocesseur doit commencer son exécution à l'adresse 0 de la mémoire,
- on part donc du principe qu'on aura toujours une instruction à l'adresse 0 de la mémoire,
- et qu'on aura toujours en mémoire une instruction, puis l'opérande 1, puis l'opérande 2, puis un octet pour stocker le résultat

### 6.2.2 Les instructions

Elles seront (pour l'instant) au nombre de deux :

| code (binaire sur 8 bits) | instruction |
|---|---|
| 00000100 | ADD |
| 00000110 | SUB |

Ces opérations arithmétiques opèrent sur des nombres de 8 bits, représentant des entiers non signés. Les instructions étant stockées en RAM, il est nécessaire de les coder. Comme la RAM stocke des mots de 8 bits, ça nous donne 256 instructions possibles, ce qui est largement suffisant pour un processeur basique... Le code choisi ci-dessus pour l'addition et la soustraction est parfaitement arbitraire : il correspond à celui qui sera implémenté en TP.

### 6.2.3 Fonctionnement de l'automate

Vu l'organisation de la RAM qui a été choisie, le fonctionnement de l'automate est simple : à chaque coup d'horloge, il va chercher successivement une instruction, puis le premier opérande, puis le deuxième opérande, calcule le résultat et le stocke. Puis il recommence à l'adresse suivante.

En détail :

1. Premier coup d'horloge : le microprocesseur présente l'adresse "0" à la RAM.
   La RAM lui présente donc sur son bus de sortie le contenu de l'adresse 0, qui est la première instruction.
2. Deuxième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM ("1").
   La RAM lui présente donc sur son bus de sortie le contenu de l'adresse 1, qui est le premier opérande.
3. Troisième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM ("2").
   La RAM lui présente donc sur son bus de sortie le contenu de l'adresse 2, qui est la deuxième opérande.
   A ce moment-là, le microprocesseur dispose de toutes les données nécessaires au calcul : l'instruction, et les deux opérandes. Il peut donc calculer le résultat.
4. Quatrième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM ("3").
   Parallèlement, il présente sur le bus de donnée en entrée de la RAM le résultat qu'il vient de calculer.
   Parallèlement, il passe la ligne WRITE de la RAM à l'état haut, pour dire à la mémoire qu'il désire effectuer une écriture.
   Le résultat du calcul est donc à ce moment-là écrit à l'adresse "3" de la mémoire.
5. Cinquième coup d'horloge : le microprocesseur incrémente l'adresse qu'il présente à la RAM ("4").
   La RAM lui présente donc sur son bus de sortie le contenu de l'adresse 4, qui est la deuxième instruction.
6. etc...

**Question 1** : concevoir l'architecture de cet automate.

On ne demande pas une représentation de toutes les portes logique de l'automate, mais juste une représentation de haut niveau : vous disposez de registres, de boîtes combinatoires dont vous ne donnerez que les équations, de multiplexeurs, de compteurs, etc.

**Réponse 1** : elle se trouve en section 6.7

## 6.3 Étape 2 : automate avec accumulateur

### 6.3.1 Chaînage des opérations

L'architecture actuelle ne permet pas de chaîner les calculs (exemple : 3 + 4 + 5). Pour pouvoir le faire, il y a plusieurs possibilités…

**Question 2** : lesquelles ?

**Réponse 2** : elle se trouve en section 6.8

### 6.3.2 L'accumulateur

Nous allons doter notre processeur d'un registre interne sur 8 bits, que nous appellerons accumulateur. Toutes les opérations arithmétiques à deux opérandes s'effectueront entre l'accumulateur et une donnée en RAM. Plus précisément : pour effectuer "$3 + 4$" et stocker le résultat en RAM, le processeur effectuera les instructions suivantes :

1. chargement de 3 dans l'accumulateur
2. addition de l'accumulateur avec un opérande en RAM ("4")
3. stockage du contenu de l'accumulateur en RAM

Pour effectuer "$3 + 4 + 5$" :

1. chargement de 3 dans l'accumulateur
2. addition de l'accumulateur avec un opérande en RAM ("4")
3. addition de l'accumulateur avec un opérande en RAM ("5")
4. stockage du contenu de l'accumulateur en RAM

On ajoute donc deux instructions à notre processeur :

- *load* : chargement de l'accumulateur à partir de la RAM
- *store* : stockage du contenu de l'accumulateur dans la RAM

Parallèlement, les instructions d'addition et de soustraction n'ont plus besoin que d'un seul opérande - le deuxième opérande est dans l'accumulateur.

De plus, tant qu'on y est, nous allons ajouter trois instructions de manipulation de bits : AND, OR et XOR (cf. le tableau 6.1), qui comme l'addition, opèrent sur le contenu de l'accumulateur et un opérande en RAM.

Le nouveau jeu d'instruction devient donc :

**Question 3** : quel est l'impact de ces spécifications sur la façon de stocker le programme en RAM ?

| code (binaire 8 bits) | instruction | effet |
|---|---|---|
| 00000001 | XOR | Effectue un XOR bit à bit entre le contenu de l'accumulateur et une donnée en RAM ; le résultat est stocké dans l'accumulateur |
| 00000010 | AND | Effectue un ET bit à bit entre le contenu de l'accumulateur et une donnée en RAM ; le résultat est stocké dans l'accumulateur |
| 00000011 | OR | Effectue un OU bit à bit entre le contenu de l'accumulateur et une donnée en RAM ; le résultat est stocké dans l'accumulateur |
| 00000100 | ADD | Additionne le contenu de l'accumulateur à une donnée en RAM ; le résultat est stocké dans l'accumulateur |
| 00000110 | SUB | Soustrait du contenu de l'accumulateur une donnée en RAM ; le résultat est stocké dans l'accumulateur |
| 00001010 | LDA | Charge dans l'accumulateur une donnée en RAM |
| 00001011 | STA | Stocke en RAM le contenu de l'accumulateur |

**Table 6.1:** *Nouveau jeu d'instructions*

**Question 4** : concevoir la nouvelle architecture du processeur. Quels sont les avantages en terme de vitesse par rapport à l'architecture précédente ?

**Réponses 3 et 4** : elles se trouvent en section 6.9

## 6.4 Étape 3 : automate avec accumulateur et indirection

### 6.4.1 Indirection

Imaginez qu'on souhaite séparer le code des données, pour :

- faire tourner un même code sur des données différentes (sans le dupliquer pour chaque groupe de donnée...)
- faire tourner différents codes sur des mêmes données (sans dupliquer les groupes de données...)
- faire tourner un code sur des données qui ne sont pas connues avant l'exécution du programme (du genre, le début du programme demande à l'utilisateur d'entrer des valeurs...)

Pour le moment, notre processeur ne sait pas faire : on doit connaître les données au moment du pré-chargement de la RAM avec le code...

Il faudrait disposer d'instructions de manipulation du contenu de la RAM à des endroits arbitraires (on ne modifierait que des données, hein, pas le code...) Cela permettrait d'aller modifier les zones où se trouvent les opérandes. Mais c'est peut-être un peu compliqué d'avoir

à modifier plein de zones éparses.

Pour être plus propre, on pourrait séparer le code des données. On aurait, en RAM, une zone avec les instructions et une zone avec les données. Il suffirait juste d'aller modifier la zone des données, et d'exécuter le code générique qui saurait, pour chaque instruction, où trouver les bons opérandes.

Pour cela, on modifie (toutes) les instructions de la façon suivante : au lieu d'avoir en RAM deux octets, un pour l'instruction et l'autre pour l'opérande, on aura plutôt un pour l'instruction et l'autre pour l'adresse de l'opérande.

Par exemple, pour effectuer "$3+4, 3-1$" on pourra avoir une organisation du genre (voir tableau 6.2 :

| adresse | type du mot stocké | exemple | zone |
|---|---|---|---|
| 0 | instruction | LDA | zone de code |
| 1 | adresse de l'opérande | 100 | |
| 2 | instruction | ADD | |
| 3 | adresse de l'opérande | 101 | |
| 4 | instruction | STA | |
| 5 | adresse de l'opérande | 103 | |
| 6 | instruction | LDA | |
| 7 | adresse de l'opérande | 100 | |
| 8 | instruction | SUB | |
| 9 | adresse de l'opérande | 102 | |
| 10 | instruction | STA | |
| 11 | adresse de l'opérande | 104 | |
| … | … | … | |
| 100 | donnée | 3 | zone de données |
| 101 | donnée | 4 | |
| 102 | donnée | 1 | |
| 103 | donnée | X | |
| 104 | donnée | X | |
| … | … | … | |

**Table 6.2:** *Organisation de la mémoire, avant exécution du programme*

Après l'exécution du code, on aura ceci en RAM (voir tableau 6.3 :

Remarque : d'habitude on sépare même la zone de données en deux, celles qui sont connues à l'écriture du programme, et les autres (celles qui sont modifiées par le programme)...

**Question 5** : proposer une modification de l'automate pour que les instructions travaillent avec des adresses d'opérandes...

| adresse | type du mot stocké | exemple | zone |
|---|---|---|---|
| 0 | instruction | LDA | zone de code |
| 1 | adresse de l'opérande | 100 | |
| 2 | instruction | ADD | |
| 3 | adresse de l'opérande | 101 | |
| 4 | instruction | STA | |
| 5 | adresse de l'opérande | 103 | |
| 6 | instruction | LDA | |
| 7 | adresse de l'opérande | 100 | |
| 8 | instruction | SUB | |
| 9 | adresse de l'opérande | 102 | |
| 10 | instruction | STA | |
| 11 | adresse de l'opérande | 104 | |
| … | … | … | |
| 100 | donnée | 3 | zone de données |
| 101 | donnée | 4 | |
| 102 | donnée | 1 | |
| 103 | donnée | 7 | |
| 104 | donnée | 2 | |
| … | … | … | |

**Table 6.3:** *Organisation de la mémoire, après exécution du programme*

**Réponse 5** : elle se trouve en section 6.10

## 6.5 Étape 4 : processeur RISC

L'architecture actuelle ne sait effectuer que des calculs linéaires (suite fixe d'instructions), sur des données potentiellement inconnues (mais dont l'adressage de stockage est connue).

Nous allons maintenant lui ajouter des instructions de saut conditionnels (et, tant qu'on y est, inconditionnels).

### 6.5.1 Flags

Pour cela, chaque opération (logique ou arithmétique) va positionner deux signaux qui seront mémorisés pour l'instruction suivante, qui ne doivent être modifiés que si on modifie l'accumulateur :

- *C* (comme Carry) :
  - mis à 1 si l'opération courante est une opération arithmétique et donne lieu à une retenue,

  - mis à 0 si l'opération courante est une opération arithmétique et ne donne pas lieu à une retenue,
  - mis à 0 si on fait un load
- $Z$ (comme zéro) :
  - mis à 1 si on charge 0 dans l'accumulateur
  - mis à 0 dans tous les autres cas.

**Question 6** : les implémenter, et rajouter deux opérations ADDC et SUBC, prenant en compte la retenue C de l'opération précédente (pour implémenter des additions / soustractions sur des grands nombres par exemple).

**Réponse 6** : elle se trouve en section 6.11

### 6.5.2 Sauts

Pour implémenter les sauts, on définit trois instructions supplémentaires :

- *JMP* : saut inconditionnel.
  L'exécution de cette instruction fait sauter l'exécution du programme directement à une adresse donnée (passée comme opérande).
- *JNC* : saut si C est nul.
  Idem à JMP, mais seulement si C est nul. Sinon, équivalent à NOP (on continue à l'adresse suivante)
- *JNZ* : saut si Z est nul.
  Idem à JMP, mais seulement si Z est nul. Sinon, équivalent à NOP (on continue à l'adresse suivante)

**Question 7** : modifier l'architecture du processeur pour implémenter les sauts.

**Réponse 7** : elle se trouve en section 6.12

Tant qu'on y est, pour disposer de pauses, on définit l'instruction NOP, qui ne fait rien.

**Question 8** : comment l'implémenter de façon simple ?

**Réponse 8** : elle se trouve en section 6.13

On ajoute aussi deux instructions, de rotation de bits (vers la droite ou vers la gauche) :

- *ROL* : ACC[7 :0] devient ACC[6 :0], ACC[7]
- *ROR* : ACC[7 :0] devient ACC[0], ACC[7 :1]

De plus, pour tester ce processeur lors du TP, on ajoute un port de sortie : c'est un ensemble de broches dont on veut pouvoir piloter l'état (passer certaines d'entre elles à l'état haut ou bas). Pour nous, il s'agit de piloter un buzzer, donc une seule sortie suffira.

Le jeu d'instruction devient donc (tableau 6.4) :

Remarques :

- $\overline{AD}$ est le deuxième octet (en RAM) de l'instruction
- $(AD)$ est la valeur en RAM stockée à l'adresse AD

| binaire | instr. | effet | explication |
|---|---|---|---|
| 00000000 | NOP | | ne fait rien ! |
| 00000001 | XOR | Acc = Acc XOR (AD) | effectue un XOR bit à bit entre l'accumulateur et une donnée en RAM, le résultat est stocké dans l'accumulateur |
| 00000010 | AND | Acc = Acc AND (AD) | effectue un ET bit à bit entre l'accumulateur et une donnée en RAM, le résultat est stocké dans l'accumulateur |
| 00000011 | OR | Acc = Acc OR (AD) | effectue un OU bit à bit entre l'accumulateur et une donnée en RAM, le résultat est stocké dans l'accumulateur |
| 00000100 | ADD | Acc = Acc + (AD) | additionne l'accumulateur à une donnée en RAM, le résultat est stocké dans l'accumulateur |
| 00000101 | ADC | Acc = Acc + (AD) + C | additionne l'accumulateur à une donnée en RAM et à la carry C, le résultat est stocké dans l'accumulateur |
| 00000110 | SUB | Acc = Acc - (AD) | soustrait du contenu de l'accumulateur une donnée en RAM, le résultat est stocké dans l'accumulateur |
| 00000111 | SBC | Acc = Acc - (AD) - C | soustrait du contenu de l'accumulateur une donnée en RAM et la carry C, le résultat est stocké dans l'accumulateur |
| 00001000 | ROL | Acc = {Acc[6 :0], Acc[7]} | effectue une rotation vers la gauche des bits de l'accumulateur |
| 00001001 | ROR | Acc = {Acc[0], Acc[7 :1] } | effectue une rotation vers la droite des bits de l'accumulateur |
| 00001010 | LDA | Acc = (AD) | charge dans l'accumulateur une donnée en RAM |
| 00001011 | STA | (AD) = Acc | stocke le contenu de l'accumulateur en RAM |
| 00001100 | OUT | BZ = (AD)[0] | Sort sur la broche BZ le bit de poids faible de la donnée en RAM, stockée à l'adresse opérande |
| 00001101 | JMP | PC = AD | saute à l'adresse opérande |
| 00001110 | JNC | PC = AD si C=0 | saute à l'adresse opérande si C est nul |
| 00001111 | JNZ | PC = AD si Z=0 | saute à l'adresse opérande si Z est nul |

**Table 6.4:** *Nouveau jeu d'instructions*

**Question 9** : finir le processeur…

**Réponse 9** : elle se trouve en section 6.14

## 6.6 Étape 5 : optimisations

**Question** : Certaines opérations peuvent s'exécuter en moins de cycles. Lesquelles, en combien de cycles ? Modifier le processeur de façon à optimiser son temps de fonctionnement.

**Question** : partant du principe que certaines opérations n'ont pas besoin d'opérande (NOP, ROT, ROR), pourquoi ne pas réduire la taille du code en RAM ?

**Question** : on veut non seulement augmenter le nombre de sorties, disons à 16, mais aussi à pouvoir utiliser certaines d'entre elles non pas comme des sorties mais comme des entrées. Et ce, de façon dynamique : au cours du programme, une broche peut devenir un sortie, puis une entrée, puis une sortie etc. Comment l'implémenter ?

**Question** : comment modifier le processeur pour supporter une taille mémoire de 1024 mots (10 bits) ?

## 6.7 Réponse 1

La première réponse est très détaillée. Les autre réponses seront plus succinctes.

### 6.7.1 Les adresses

Pour effectuer un calcul, l'automate doit disposer de trois informations :
- l'instruction (l'opération)
- l'opérande 1
- l'opérande 2

Il doit en disposer *en même temps*. Mais elles sont stockées en RAM, et ne peuvent être lues que l'une après l'autre. Il faudra donc prévoir un moyen de stockage de ces trois informations à l'intérieur du processeur pour pouvoir effectuer le calcul.

Vu l'organisation de la mémoire, il semble logique de lire ces trois informations de la façon la plus simple possible, c'est à dire :
- tout d'abord l'instruction,
- puis l'opérande 1,
- puis l'opérande 2,

ce qui correspond à un parcours linéaire de la mémoire.

De plus, le stockage du résultat s'effectue dans la RAM à l'adresse suivant celle de l'opérande 2. On peut donc doter l'automate d'un compteur qu'on appellera *compteur d'adresse* ou *PC* (Program Counter), qui donnera l'adresse de la RAM à laquelle on est en train d'accéder (que ce soit en lecture ou en écriture). Ce compteur sera incrémenté à chaque coup d'horloge, et pilotera directement le bus d'adresse de la RAM.

### 6.7.2 Les données

Vu ce qui vient d'être dit, l'automate a un fonctionnement linéaire - l'ordre des actions effectuées est toujours le même :

1. aller chercher une instruction
2. aller chercher le premier opérande
3. aller chercher la deuxième opérande
4. stocker le résultat du calcul

On peut donc le concevoir comme une machine à quatre états, dont le fonctionnement est circulaire : état 1 $\rightarrow$ état 2 $\rightarrow$ état 3 $\rightarrow$ état 4 $\rightarrow$ état 1 $\rightarrow$ état 2 $\rightarrow$ …

**État 1** :

- le compteur est en train de présenter à la RAM une adresse correspondant à une instruction.
  Le processeur récupère sur le bus Q[7 :0] la contenu de la RAM à cette adresse, c'est à dire l'instruction à effectuer.
- il faut stocker cette instruction pour plus tard (quand on effectuera l'opération demandée).
  On ajoute donc à l'automate un registre sur 8 bits disposant d'un enable (8 bascules DFFE en parallèle).
  L'entrée de ce registre est reliée au bus Q[7 :0] (sortie de la RAM)
  Le signal d'enable de ce registre est mis à l'état haut seulement pendant l'état 1 --> stockage de l'instruction dans le registre

**État 2** :

- le compteur est en train de présenter à la RAM une adresse correspondant aux premier opérande.
  le processeur récupère sur le bus Q[7 :0] la contenu de la RAM à cette adresse, c'est à dire l'opérande 1…
- il faut stocker cet opérande, donc, on ajoute un registre 8 bits avec enable, relié à la sortie de la RAM (Q[7 :0]).
  l'enable est mis à l'état haut seulement pendant l'état 2.

**État 3** :

- le compteur est en train de présenter à la RAM une adresse correspondant aux deuxième opérande.
  le processeur récupère sur le bus Q[7 :0] la contenu de la RAM à cette adresse, c'est à dire l'opérande 2…
- comme d'habitude on stocke cet opérande dans un registre 8 bits, dont l'enable est piloté à l'état haut seulement pendant ce cycle-ci.

Remarque : on peut se dire que ce n'est pas la peine de stocker cet opérande, car on dispose dès à présent de toutes les données pour effectuer le calcul : l'instruction dans un registre, l'opérande dans un autre registre, et le deuxième opérande sur le bus Q[7 :0]. Mais il faudrait alors stocker le résultat dans un registre 8 bits, car on ne fait son stockage en RAM qu'au prochain cycle…Alors qu'ici, le calcul et le stockage seront faits ensemble au prochain cycle

(donc pas besoin de stocker le résultat dans un registre). Au total, dans les deux approches, le nombre de registres est le même, et ce ne sont que des considérations de chemin critique qui permettront de déterminer la meilleure des deux méthodes…

**État 4** :

- le compteur est en train de présenter à la RAM une adresse correspondant au résultat à stocker.
- l'automate dispose dans ses trois registres de toutes les données pour effectuer le calcul. Il suffit d'ajouter une fonction combinatoire pure, pour produire le résultat. La sortie de cette fonction combinatoire sera reliée au bus d'entrée de la RAM. L'équation de cette fonction sera du genre : `D[7:0] = (si INSTRUCTION="00000100" : OP_1[7:0] + OP_2[7:0], sinon OP_1[7:0] - OP_2[7:0])` Une telle fonction combinatoire a été réalisée au TP numéro 2…(ALU)
- Parallèlement, l'automate doit piloter le signal WRITE de la RAM à l'état haut, pour dire à la RAM de stocker à l'adresse courante la sortie de la fonction de calcul.

On obtient donc l'architecture suivante pour notre processeur :

- En rouge : le compteur d'adresse courante
- En bleu : les trois registres 8 bits, les signaux *load* sont les enable
- En noir rond : la fonction combinatoire de calcul proprement dite (ALU)
- En noir carré : la machine à état qui séquence tout ça…

La machine à états (CTRL) est présentée en figure 6.4, et son graphe d'états en figure 6.5

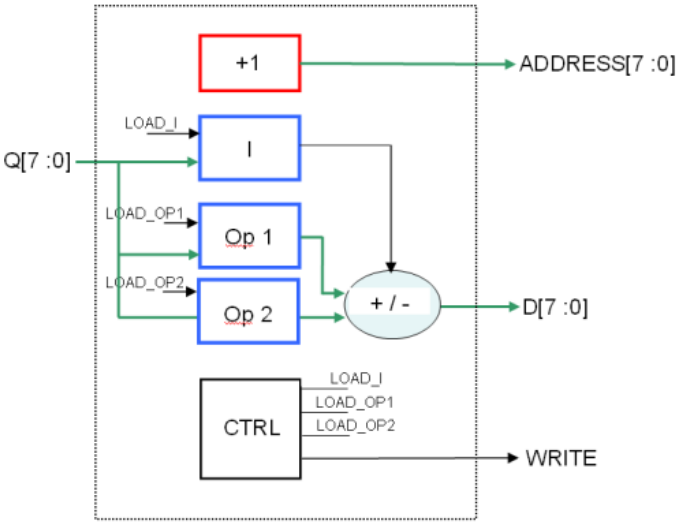



**Figure 6.4:** *Architecture de la première version*

Elle a quatre état, parcourus de façon circulaire, sans condition sur les transitions.

Elle dispose de 4 sorties, chacune d'entre elles à l'état haut dans un seul état de la machine. Un codage one-hot est donc très approprié.

L'implémentation en registre a déjà été vue (registres les uns à la suite des autres), et ne sera pas détaillée ici. Les sorties des registres donnent directement les sorties de la machine à état…

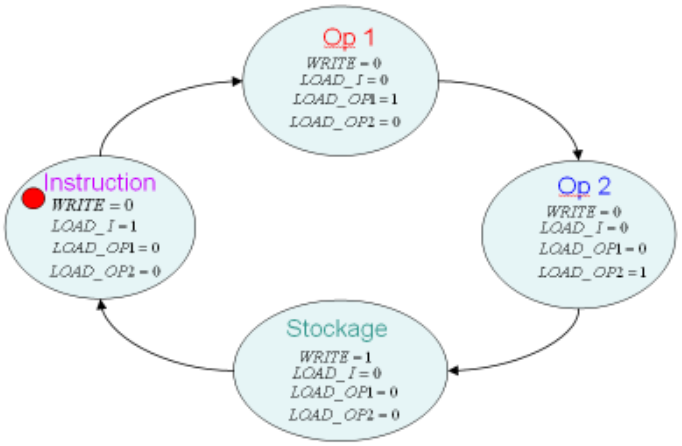



**Figure 6.5:** *Graphe d'états de la première version*

## 6.8 Réponse 2

Plusieurs possibilités, leur nombre est limité seulement par votre imagination. Voici quelques exemples :

- Garder le résultat de chaque opération en mémoire, et définir une nouvelle addition qui opère sur un opérande en RAM et le résultat qu'on a gardé.
  L'inconvénient est qu'on rajoute une instruction pour chaque type d'opération, que cette nouvelle opération, ne nécessitant qu'un seul opérande en RAM pourra être effectuée en 3 cycles au lieu de 4, et que ça risque de compliquer la machine à état si on veut l'optimiser (certaines opération en 3 cycles, d'autres en 4)…
- Définir des opérations de manipulation de la RAM, et grâce à elles recopier le résultat en RAM à l'endroit d'une des deux opérandes de la prochaine instruction. C'est bien compliqué…
- Définir une nouvelle addition qui opère sur un opérande à l'endroit habituel en RAM, et sur un autre opérande situé à l'adresse (instruction - 1)…
- Utiliser la première solution, mais pour simplifier les choses (et par cohérence) supprimer les opérations sur deux opérandes en RAM. Toutes les opérations (à deux opérandes) se feront entre un opérande en RAM, et un gardé dans un registre interne au processeur. Et pour rendre cela possible, on définit deux nouvelles instructions : chargement de ce registre à partir d'une donnée en RAM, et stockage du contenu de ce registre en RAM. C'est l'objet de la suite !

## 6.9 Réponses 3 et 4

Toutes les opérations ne nécessitent plus qu'un seul opérande :

- pour le *LDA*, c'est normal, c'est la donnée à amener à l'accumulateur
- pour le *STA*, aucun opérande. Par contre, en RAM, à la suite de l'instruction *STA*, il doit y avoir un emplacement libre pour stocker le contenu de l'accumulateur
- pour les opérations à deux opérandes, l'un est en RAM, l'autre est implicite, c'est le contenu de l'accumulateur

Le contenu de la RAM se présentera donc maintenant ainsi :

| adresse | type du mot stocké | exemple | effet |
|---|---|---|---|
| 0 | instruction | LDA | |
| 1 | donnée | 3 | l'accumulateur contient maintenant 3 |
| 2 | instruction | ADD | |
| 3 | donnée | 4 | l'accumulateur contient maintenant 7 |
| 4 | instruction | SUB | |
| 5 | donnée | 1 | l'accumulateur contient maintenant 6 |
| 6 | instruction | STA | |
| 7 | donnée | X | après l'exécution du programme cet emplacement en RAM contiendra "6" |

On remarque donc qu'une adresse sur deux contient une instruction, une sur deux contient une donnée (soit opérande, soit stockage du contenu de l'accumulateur)…

### 6.9.1 Les adresses

Comme précédemment, les adresses de la RAM sont parcourues de façon successives. On garde donc le compteur d'adresse incrémenté à chaque cycle d'horloge.

### 6.9.2 Les données

Pour effectuer les calculs, le processeur n'a besoin maintenant de connaître que deux informations : l'instruction et l'opérande. On garde donc le registre d'instruction (8 bits) qui stocke l'instruction à effectuer pendant qu'on va chercher l'opérande en RAM.

Par contre, auparavant on parcourait 4 emplacements en RAM pour chaque instruction, d'où un contrôleur à 4 états. Maintenant on ne parcourt plus que 2 emplacements en RAM, donc un contrôleur à 2 états devrait convenir…

A chaque instruction, le processeur effectuera ceci :

Pour une opération normale :

1. aller chercher l'instruction en RAM, la stocker dans le registre d'instruction
2. aller lire l'opérande en RAM, effectuer le calcul et stocker le résultat dans l'accumulateur (opération)

Pour un load :

1. aller chercher l'instruction en RAM, la stocker dans le registre d'instruction
2. aller lire l'opérande en RAM, et le stocker dans l'accumulateur (opération)

Pour un store :

1. aller chercher l'instruction en RAM, la stocker dans le registre d'instruction
2. écrire le contenu de l'accumulateur en RAM à l'adresse courante

Chaque instruction est donc traitée de façon très similaire :

1. un cycle de récupération de l'instruction (dans lequel l'enable du registre d'instruction est mis à l'état haut).
2. un cycle de traitement de l'instruction

### 6.9.3 L'accumulateur

Lors du second cycle, l'accumulateur peut subir trois traitements différents :

- pour une opération ($ADD$, $SUB$, $AND$, $XOR$, $OR$), l'accumulateur se voit modifié et chargé avec le résultat de l'opération
- pour un load, l'accumulateur est modifié aussi, et chargé avec la donnée sortant de la RAM
- pour un store par contre, l'accumulateur n'est pas modifié…

En entrée de l'accumulateur on mettra donc un multiplexeur qui présentera soit le résultat de l'opération en cours (si on exécute une opération standard), soit le contenu de la RAM (si on exécute un load). De plus, dans ces deux cas, le signal enable de l'accumulateur sera mis à l'état haut (pour autoriser sa modification) dans l'état 2 (quand on accède à la partie donnée de la RAM) Dans le cas d'un store, on laisse l'enable de l'accumulateur à l'état bas pour ne pas le modifier.

En d'autre termes, l'enable de l'accumulateur a pour équation : **`LOAD_ACC = (Instruction <> STORE) ET (Etat = état 2)`**

Le pilotage du multiplexeur en entrée de l'accumulateur aura pour équation quelque chose du genre : **`ACC = (si Instruction == LOAD alors Q[7:0], si Instruction == opération alors ALU(ACC, Q[7:0]), si Instruction == STORE alors peu importe…)`**. Ce qui se simplifie en **`ACC = (si Instruction == LOAD alors Q[7:0], sinon ALU(ACC, Q[7:0]))`**

La sortie de l'accumulateur est branchée à la fois :

- sur le bus d'entrée de la RAM (pour le cas où on fait un store)
- sur l'ALU (qui implémente, selon l'instruction à effectuer, l'addition, la soustraction, le XOR, etc…)

Enfin la génération du signal d'écriture en RAM est simple : il est mis à l'état haut quand l'instruction est un STORE, et qu'on est dans l'état 2. Le contenu de l'accumulateur est présenté sur l'entrée de la RAM (cf. ci dessus), l'adresse courante est sur le bus d'adresse de la RAM, la RAM est donc mise à jour avec la bonne valeur…

### 6.9.4 Bilan

On a donc les éléments suivants :

- compteur d'adresse (PC)
- registre d'instruction
- accumulateur avec multiplexeur en entrée
- un contrôleur, machine à état générant les signaux $LOAD\_I$, $LOAD\_ACC$, $WRITE$ et le contrôle du multiplexeur

Remarque : les signaux générés par la machine à état ne dépendent pas seulement de l'état courant, mais aussi de l'instruction à exécuter. C'est donc une machine de Mealy…

L'architecture globale est donc celle représentée sur la figure 6.6, et son graphe d'états en figure 6.7

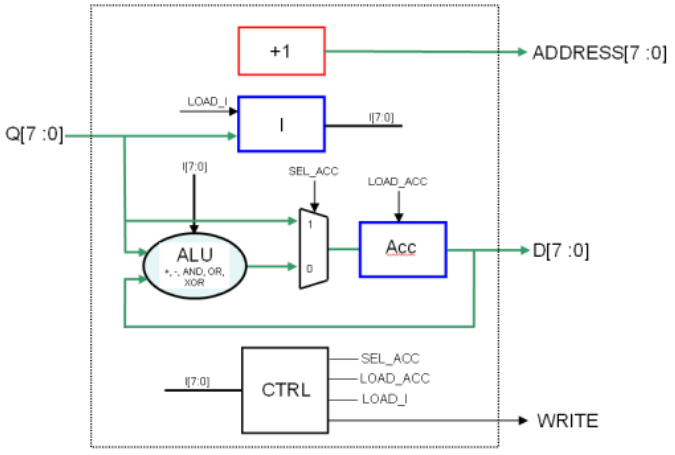



**Figure 6.6:** *Architecture de la deuxième version*

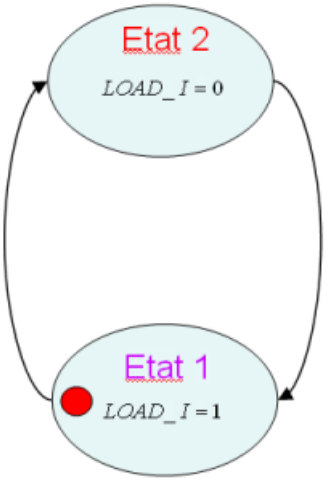



**Figure 6.7:** *Graphe d'états de la deuxième version*

avec :

- **`SEL_ACC = (I[7:0] == LOAD)`**
- **`LOAD_ACC = (I[7:0] <> STORE) ET (Etat = état 2)`**
- **`WRITE = (I[7:0] == STORE) ET (Etat = état 2)`**

### 6.9.5 Performances

Pour une opération :

- avant : 4 cycles
- maintenant : 6 cycles (2 + 2 + 2)

Pour deux opérations chaînées :

- avant : 8 cycles (4 + 4. Enfin, plus exactement, on ne savait pas faire…)
- maintenant : 8 cycles (2 + 2 + 2 + 2)

Pour trois opérations chaînées :

- avant : 12 cycles (4 + 4 + 4. Même remarque)
- maintenant : 10 cycles

Bref, pour n opérations :
- avant : 4n cycles
- maintenant : 2n+4 cycles si peut les enchaîner, 6n sinon.

On a donc tout intérêt à enchaîner les calculs. Ce qui est *très* souvent le cas en pratique…

## 6.10 Réponse 5

L'automate doit maintenant pour chaque instruction
- aller chercher l'instruction (la stocker dans le registre d'instruction)
- aller chercher l'adresse de l'opérande (le stocker, dans un registre dit "d'adresse")
- aller chercher l'opérande proprement dit, en lisant la RAM à l'adresse stockée au cycle précédent.

On a donc une machine qui possède un état de plus (celui où on va lire en RAM l'opérande proprement dit).

### 6.10.1 Les adresses

Maintenant, on n'accède plus à la RAM de façon successive. Dans l'exemple de programme donné, les adresses présentées à la RAM seront celles-ci :

1. 0
2. 1
3. 100
4. 2
5. 3
6. 101
7. 4
8. 5
9. 103
10. …

Les adresses de code sont globalement linéaires (0, 1, 2, 3, …), celles des données ne le sont pas (elles sont arbitraires). Il faut donc présenter sur le bus d'adresse RAM
- soit le compteur d'adresse pendant les deux premiers cycles (et on l'incrémente à chaque fois)
- soit le contenu du registre d'adresse (adresse de l'opérande à aller chercher) pendant le troisième cycle (et ici le compteur d'adresse ne doit pas être incrémenté)

donc : multiplexeur…

De plus, le compteur d'adresse doit être piloté par un signal **`INCR_PC`** : il n'est incrémenté que si **`INCR_PC`** est à l'état haut.

Le registre d'adresse est chargé au cycle numéro 2. Son contenu n'est utile qu'au cycle numéro 3. Il n'est donc pas nécessaire de le piloter avec un enable…Il peut rester tout le temps actif : son contenu sera indéterminé pendant les cycles 1 et 2, mais ce n'est pas grave, il n'est pas utilisé pendant ces cycles là…

L'architecture globale est donc celle représentée sur la figure 6.8, et son graphe d'états en figure 6.9

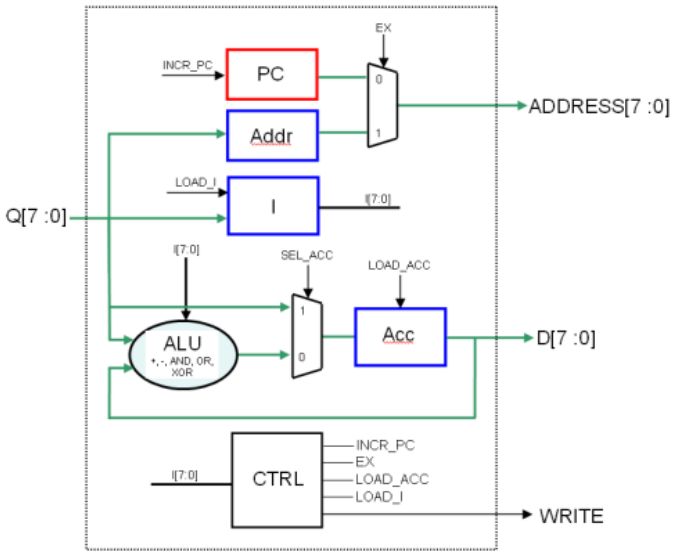



**Figure 6.8:** *Architecture de la troisième version*

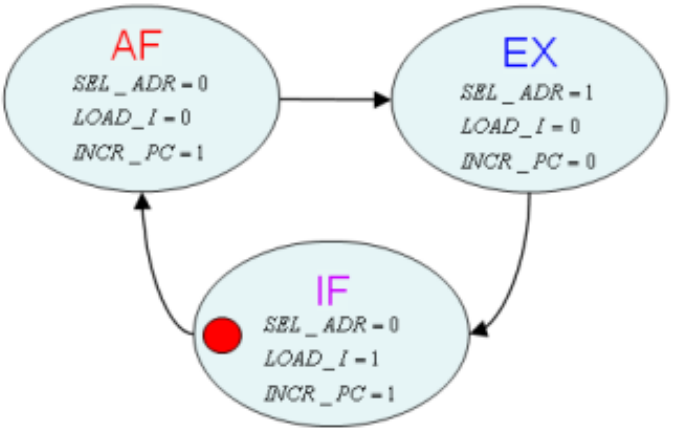



**Figure 6.9:** *Graphe d'états de la troisième version*

C'est là aussi une machine de Mealy, et les équations sont :

- **`SEL_ACC = (I[7:0] == LOAD)`**
- **`LOAD_ACC = (I[7:0] <> STORE) ET (Etat = Ex)`**
- **`WRITE = (I[7:0] == STORE) ET (Etat = Ex)`**

## 6.11 Réponse 6

### 6.11.1 Flags

La génération de C et Z est combinatoire et peut être effectuée par l'ALU.

Il suffit juste de rajouter deux registres 1 bits pour stocker ces deux signaux, pilotés par le même enable que l'accumulateur (**`LOAD_ACC`**, qu'on appellera maintenant **`LOAD_AZC`**). On considérera donc que Z et C font partie de l'accumulateur (qui devient donc un registre sur 10 bits : 8 de donnée, 1 pour Z, un pour C).

### 6.11.2 ADDC / SUBC

Il suffit de faire entrer C sur la retenue entrante de l'addition ou de la soustraction…

## 6.12 Réponse 7

Pour implémenter les sauts, il suffit de se donner la possibilité de remplacer le contenu de PC par la valeur lue en RAM.

PC devient donc un peu plus complexe. C'est globalement un compteur, mais il

- est incrémenté si son signal de commande **INCR_PC**= 1
- est chargé avec une nouvelle valeur si un signal de chargement **LOAD_PC**= 1
- si **LOAD_PC**et **INCR_PC**valent 1, c'est **LOAD_PC**qui prime...

Ceci peut être implémenté comme sur la figure 6.10.

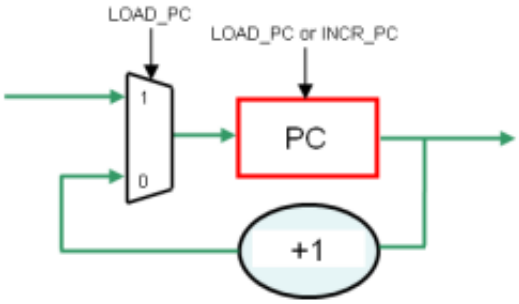



**Figure 6.10:** *Implémentation du PC*

Pour simplifier les schémas, lorsque nous parlerons de PC, ce sera de ce bloc-ci.

Il faut maintenant générer le signal **LOAD_PC**. Ce signal sera aussi généré par la machine à états CTRL. Le PC doit être remplacé lorsqu'on exécute un saut, et que la condition du saut est vérifiée. La nouvelle valeur est présente sur le bus de sortie de la RAM pendant le cycle 2.

On aura donc une équation du style : **LOAD_PC = si (I[7:0] == JMP ou I[7:0] == JNC et C == 0 ou I[7:0] == JNZ et Z == 0) et (état = etat 2), alors 1, sinon 0**.

L'architecture globale est donc celle représentée sur la figure 6.11, avec une machine à état CTRL à peine modifiée (même graphe d'état) représentée figure 6.14.

- **SEL_ACC = (I[7:0] == LOAD)**
- **LOAD_ACC = (I[7:0] <> (STORE ou saut)) ET (Etat = Ex)**
- **WRITE = (I[7:0] == STORE) ET (Etat = Ex)**
- **LOAD_PC = si (I[7:0] == JMP ou I[7:0] == JNC et C == 0 ou I[7:0] == JNZ et Z == 0) et (état = AF), alors 1, sinon 0**

## 6.13 Réponse 8

L'instruction NOP ne fait rien. Elle n'a pas besoin d'opérande, et pourrait donc être stockée sur un seul octet (au lieu de deux pour les autres).

Mais cela compliquerait la gestion de la machine à états pour générer les signaux **LOAD_PC**et **INCR_PC**. De plus, ça pourrait poser d'autres problèmes (cf. les optimisations).

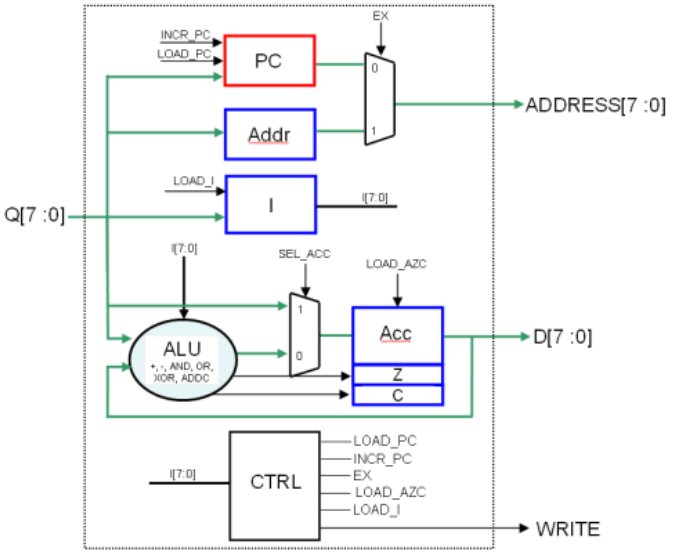



**Figure 6.11:** *Architecture de la quatrième version*

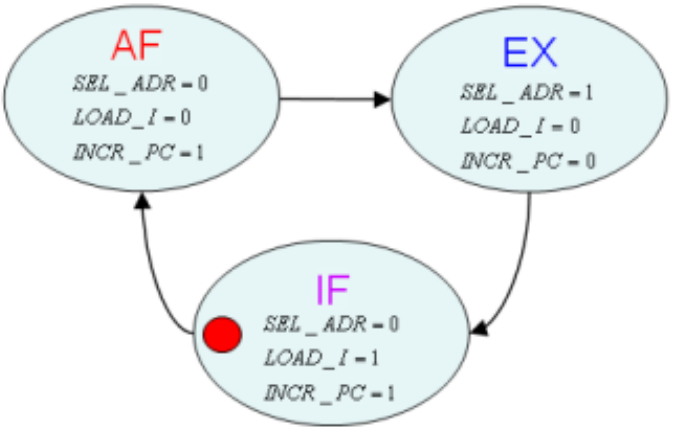



**Figure 6.12:** *Graphe d'états de la quatrième version*

On peut donc accepter de perdre un octet de mémoire, et ne rien changer à l'organisation de la mémoire. L'instruction NOP sera accompagnée d'un opérande qui ne servira a rien... Une instruction sera toujours exécutée en trois cycles. La seule modification de la machine à état sera l'équation suivante : **`LOAD_ACC = (I[7:0] <> (STORE ou saut ou NOP)) ET (Etat = Ex)`**

## 6.14 Réponse 9

### 6.14.1 ROL / ROR

ces opérations sont combinatoires et seront donc implémentées dans l'ALU.

Remarque : comme le NOP, elles ne nécessitent pas d'opérande. De même, pour garder une cohérence (nous optimiserons ça plus tard), on garde un codage des instructions sur deux octets. Pour ROR et ROL, le deuxième octet n'a pas de signification...

### 6.14.2 Sortie BZ

On ajoute un registre 1 bit, piloté par un signal d'enable appelé **`LOAD_BZ`**.

- l'entrée de ce registre est le bus de sortie de la RAM
- sa sortie est connectée à la broche de sortie buzzer du processeur...

**`LOAD_BZ`**sera généré par la machine à état, selon l'équation suivante : **`LOAD_BZ = (I[7:0] == OUT) et (état = EX)`**...

L'architecture globale est donc celle représentée sur la figure 6.13, avec une machine à état CTRL représentée figure 6.14.

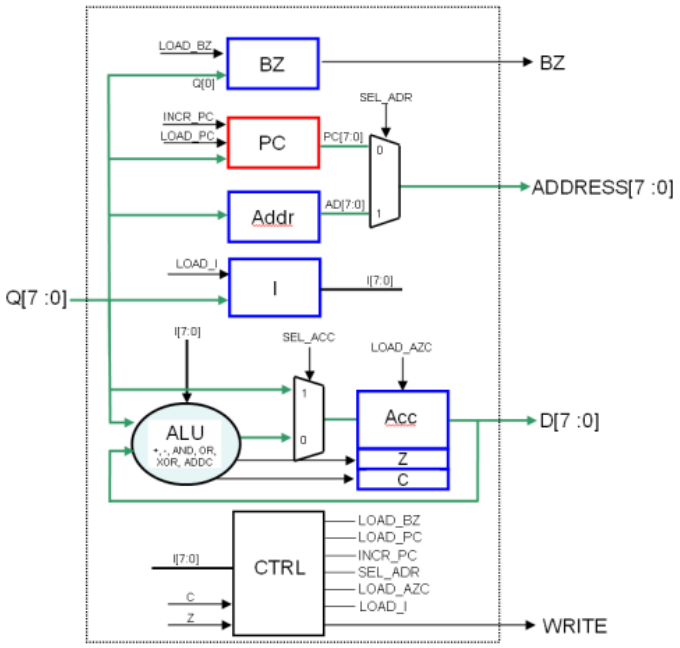



**Figure 6.13:** *Architecture de la version finale*

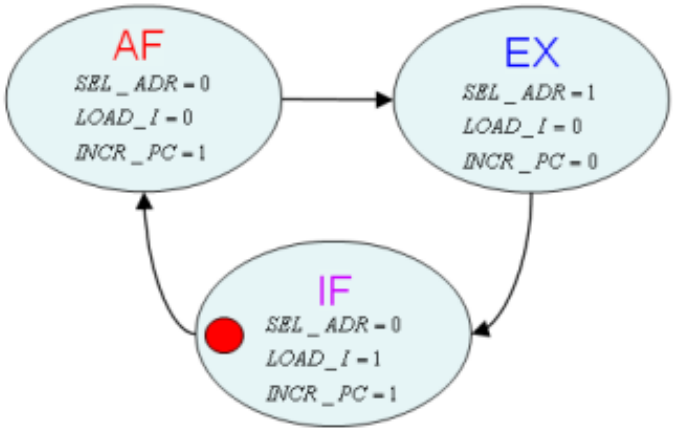



**Figure 6.14:** *Graphe d'états de la version finale*

Les équation sont laissées en exercice au lecteur !..

Remarque : le signal **`SEL_ACC`** ne sort pas de CTRL sur le schéma : il peut être inclus, avec le multiplexeur qu'il pilote, dans l'ALU...

# Chapitre 7

# Du transistor à la logique CMOS

## 7.1 Introduction

### 7.1.1 Objectifs

Il s'agit :

- de comprendre les principes de la construction de portes en structure "logique complémentaire" à partir de transistors NMOS et PMOS,
- de savoir évaluer les principales performances électriques de ce type de cellules,
- de savoir construire un modèle de performances utilisable au niveau fonctionnel, c'est à dire à un niveau où le nombre de cellules appréhendées est supérieur à plusieurs dizaines,
- de connaître, pour ces différents niveaux d'analyse, les ordres de grandeurs caractéristiques.

### 7.1.2 Présentation

En utilisant nos connaissances du transistor MOS, nous élaborerons un modèle de type interrupteur commandé, qui permet de construire des portes logiques et de comprendre les principes et les caractéristiques de la logique complémentaire. Nous étudierons un modèle linéaire du temps de propagation le long d'un chemin logique. Enfin nous évoquerons le principe et l'utilisation d'une bibliothèque de cellules.

## 7.2 Modèle en interrupteur

### 7.2.1 Modélisation

Nous transformons le modèle électrique du transistor (transconductance non linéaire), rappelé au chapitre 7.5, en un interrupteur commandé uniquement par la tension de grille $V_G$. Ainsi nous faisons correspondre :

- à l'état bloqué du transistor l'état ouvert de l'interrupteur que nous notons "O",
- à l'état passant du transistor l'état fermé de l'interrupteur que nous notons "F",

Du fait de la connexion systématique des substrats, nous omettrons souvent de le dessiner (voir 7.5).

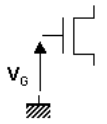
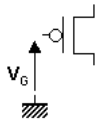
| dipôle d'entrée | niveau logique sur la grille | |
|---|---|---|
| | **0** | **1** |
| transistor | modèle en interrupteur du dipôle de sortie | |
| NMOS | **O** | **F** |
|   |  |  |
| PMOS | **F** | **O** |
|   |  |  |

**Table 7.1:** *Modèle en interrupteur*

**transistor NMOS**

- Lorsque la tension de grille $V_G$ est à "1" il suffit d'avoir :
  $V_S < V_D - V_{T_N}$
  ($V_D$ tension du drain) pour que la condition de conduction :
  $V_{GS_N} > V_{T_N}$
  soit respectée et que l'interrupteur équivalent soit fermé, ce que nous notons "F"
- Lorsque la tension de grille $V_G$ est à "0" la condition de blocage est remplie :
  $V_{GS_N} = 0V < V_{T_N}$
  l'interrupteur équivalent est ouvert, ce que nous notons "O"

**transistor PMOS**

- Lorsque la tension de grille $V_G$ est à "1", la condition de blocage est remplie :
  $V_{GS_P} = 0V > V_{T_P}$
  l'interrupteur équivalent est ouvert : "O"

- Lorsque la tension de grille $V_G$ est à "0" il suffit d'avoir :
  $V_S > -V_{T_P}$
  pour que la condition de conduction :
  $V_{GS_P} < V_{T_P}$
  soit respectée et que l'interrupteur équivalent soit fermé (passant) : "F"

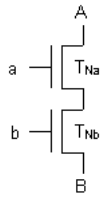
### 7.2.2 Quelques montages simples

Dans les tableaux suivants les lettres minuscules : a, b, désignent les variables logiques d'entrée et les lettres majuscules : A, B les extrémités de la branche.

Nous notons F lorsqu'un transistor est passant (interrupteur équivalent fermé), O s'il est bloqué (interrupteur équivalent ouvert). $F_{AB}$ désigne la fonction logique associée à l'état de la branche située entre les points A et B. Son état est noté comme celui des transistors. La valeur logique de la fonction $F_{AB}$ est obtenue en sommant les produits des états des entrées produisant la fermeture de la branche AB (en gras dans les tableaux suivants). L'état "1" de l'entrée a est noté $a$. L'état "0" de l'entrée a est noté $\overline{a}$ (a_barre, !a).

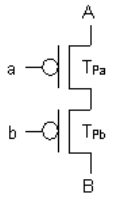
#### Montages séries

Pour qu'une branche constituée de 2 interrupteurs en série soit passante, il faut que les 2 interrupteurs soient fermés en même temps (fonction logique $ET$ notée "." Si l'un au moins est ouvert, la branche est ouverte.




| Entrées | | Transistors | | Branche |
|---|---|---|---|---|
| $a$ | $b$ | $T_{N_a}$ | $T_{N_b}$ | $F_{AB}$ |
| 0 | 0 | O | O | O |
| 0 | 1 | O | F | O |
| 1 | 0 | F | O | O |
| 1 | 1 | F | F | F |

**Table 7.2:** *NMOS.* $F_{AB} = a \cdot b$



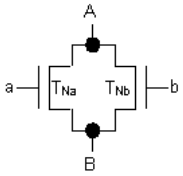

| Entrées | | Transistors | | Branche |
|---|---|---|---|---|
| $a$ | $b$ | $T_{P_a}$ | $T_{P_b}$ | $F_{AB}$ |
| 0 | 0 | F | F | F |
| 0 | 1 | F | O | O |
| 1 | 0 | O | F | O |
| 1 | 1 | O | O | O |

**Table 7.3:** *PMOS.* $F_{AB} = \overline{a} \cdot \overline{b} = \overline{a + b}$

#### Montages parallèles

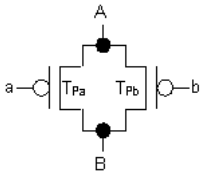
Pour qu'une branche constituée de 2 interrupteurs en parallèle soit passante, il suffit qu'un interrupteur au moins soit fermé (fonction logique $OU$ notée "+" Si les deux sont ouverts, la

branche est ouverte.




| Entrées | | Transistors | | Branche |
|---|---|---|---|---|
| $a$ | $b$ | $T_{N_a}$ | $T_{N_b}$ | $F_{AB}$ |
| 0 | 0 | O | O | O |
| 0 | 1 | O | F | F |
| 1 | 0 | F | O | F |
| 1 | 1 | F | F | F |

**Table 7.4:** *NMOS.* $F_{AB} = \overline{a} \cdot b + a \cdot \overline{b} + a \cdot b = a + b$




| Entrées | | Transistors | | Branche |
|---|---|---|---|---|
| $a$ | $b$ | $T_{P_a}$ | $T_{P_b}$ | $F_{AB}$ |
| 0 | 0 | F | F | F |
| 0 | 1 | F | O | F |
| 1 | 0 | O | F | F |
| 1 | 1 | O | O | O |

**Table 7.5:** *PMOS.* $F_{AB} = \overline{a} \cdot \overline{b} + \overline{a} \cdot b + a \cdot \overline{b} = \overline{a \cdot b}$

Chaque transistor NMOS TNx peut évidemment être remplacé par un réseau de transistors NMOS. De même chaque transistor PMOS TPx peut être remplacé par un réseau de transistors PMOS... et ainsi de suite, pour constituer deux réseaux duaux complexes.

## 7.3 La logique complémentaire CMOS

### 7.3.1 Introduction

Reprenons le schéma du circuit "Résistance Transistor Logique" (figure 7.1). Remplaçons la transconductance idéale du transistor NMOS par l'interrupteur équivalent $T_N$. Identifions ses 2 états d'équilibre (on dit aussi états statiques).

Nous avons vu que la grille d'un transistor NMOS ou PMOS, est isolée. Ainsi la commande de l'interrupteur équivalent au transistor est-elle isolée de l'interrupteur lui même. Les équations de ce circuit sont :

- $V_{DD} = V_R + V_s = R \cdot I_R + V_s$
- $I_{DD} = I_R = I_T$
- Lorsque l'entrée vaut "0", l'interrupteur $T_N$ est ouvert. Aucun courant ne circule dans la branche de sortie : $I_{DD} = I_R = I_T = 0 \Rightarrow V_{DD} = V_s \equiv$ "1"

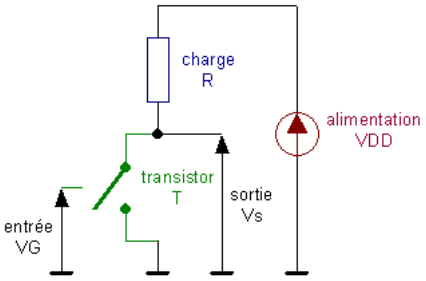



**Figure 7.1:** *Circuit Résistance Transistor Logique*

- Lorsque l'entrée vaut "1", l'interrupteur $T_N$ est fermé : $V_s = 0 \equiv$ "0" $\Rightarrow V_{DD} = R \cdot I_{R_max}$

Ce type de circuit est un inverseur logique. En régime statique, à l'état bas ("0" en sortie) il consomme du courant, et l'on a cherché un moyen pour éviter toute consommation en régime statique. La logique complémentaire, est une des solutions efficaces à ce problème.

### 7.3.2 Notion de complémentarité

Le mot complémentaire veut dire que l'on dispose, autour de l'équipotentielle de sortie, non plus d'une branche passive (R) et d'une branche active ($T_N$), mais de deux branches actives duales, c'est dire conduisant l'une à la stricte exclusion de l'autre, et pour des signaux de commande complémentaires. Un même signal commande au moins une paire d'interrupteurs complémentaires.

#### Exemple de l'inverseur

La porte la plus simple de la logique complémentaire est l'inverseur. Chacune des 2 branches est constituée d'un seul transistor. Le symbole et le montage de l'inverseur CMOS sont représentés dans la figure 7.2.

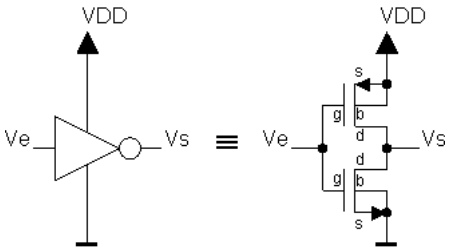



**Figure 7.2:** *l'inverseur CMOS*

Analysons son fonctionnement.

- L'interrupteur $T_N$ est le modèle du transistor NMOS (entrée = "1" et il est fermé, entrée = "0" et il est ouvert).
- L'interrupteur $T_P$ est le modèle du transistor PMOS (entrée = "0" et il est fermé, entrée = "1" et il est ouvert).
- L'entrée $V_e$ est commune aux deux grilles, celle de $T_N$ en parallèle avec celle de $T_P$.

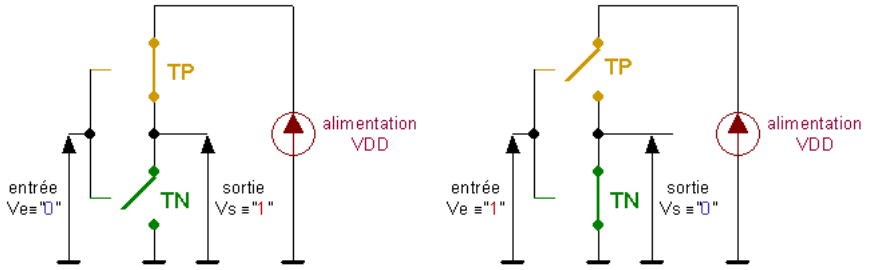



**Figure 7.3:** *Régime statique : les 2 états statiques de l'inverseur*

- Lorsque la branche N est fermée, la branche P est ouverte : la sortie est reliée à "0", électriquement : la masse (0V).
- Lorsque la branche N est ouverte, la branche P est fermée : la sortie est reliée à "1", électriquement : $V_{DD}$.

### Consommation

En régime statique, c'est à dire pour chacun des deux états stables, aucun chemin électrique n'existe entre $V_{DD}$ et la masse, aucun courant n'est donc consommé.

Pour analyser ce qui se passe en régime transitoire, ce qui sera fait plus précisément dans le chapitre « Performances de la logique CMOS »(7), rappelons-nous que :

1. la tension d'entrée $V_e$ n'a pas un temps de transition (à la montée comme à la descente) nul. Ainsi pendant un certain temps : lorsque $V_{TN} < V_e < V_{DD} - V_{T_P}$, les deux transistors sont passant. Un courant dit *de court-circuit*, délivré par l'alimentation, traverse les deux transistors passants vers la masse.
2. la charge de cette porte logique, est essentiellement constituée d'une capacité $C_T$, représentant l'ensemble des capacités parasites connectées sur l'équipotentielle de sortie. La charge (de "0" à "1", soit de 0V à $V_{DD}$) et la décharge (de "1" à "0", soit de $V_{DD}$ à 0V) du noeud de sortie, nécessite un courant, donc une consommation dynamique (voir schéma 7.4).

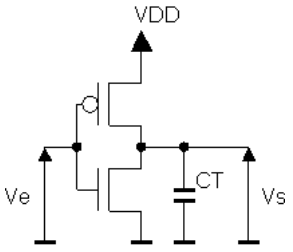



**Figure 7.4:** *l'inverseur CMOS et sa charge capacitive*

Durant le régime transitoire, l'alimentation va :

- soit charger, de 0V à $V_{DD}$, la capacité $C_T$ au travers de l'interrupteur fermé $T_P$ équivalent au transistor PMOS passant,
- soit décharger, de $V_{DD}$ à 0V, la capacité $C_T$ au travers de l'interrupteur fermé $T_N$ équivalent au transistor NMOS passant.

En conclusion, la consommation statique de l'inverseur CMOS est nulle. La consommation transitoire (dynamique) est due au courant de court-circuit et à la (dé)charge de la capacité $C_T$.

### 7.3.3 Porte complexe

#### Constitution du circuit

Dans l'exemple de l'inverseur, la branche N et la branche P ne sont constituées que d'un interrupteur chacune. Pour réaliser une fonction plus complexe, nous allons remplacer chaque branche par un réseau de plusieurs interrupteurs de même type, comme illustré dans la figure 7.5. Les règles globales sont les mêmes que pour l'inverseur, mais chaque branche N et P, sera constituée d'un réseau d'interrupteurs, montés en parallèle ou en série (voir le paragraphe 7.2.2), tous reliés deux à deux (au moins) par leur grille, et respectant la condition de conduction d'une branche à l'exclusion de celle de l'autre.

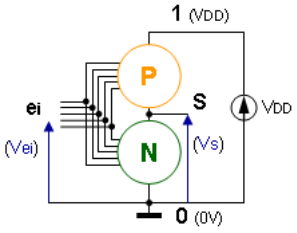



**Figure 7.5:** *Schéma de principe de la logique complétaire*

#### Principes et fontionnalité

En logique complémentaire, un circuit est constitué de deux réseaux duaux :

- un réseau N, constitué exclusivement de transistors NMOS, branché entre la sortie et le "moins de l'alimentation" (en général la masse) qui correspond au "0" logique,
- un réseau P, constitué exclusivement de transistors PMOS, branché entre la sortie et le "plus de l'alimentation" ($V_{DD}$) qui correspond au "1" logique,

Pour être duaux les deux réseaux doivent satisfaire les principes suivants :

- être commandés par les mêmes entrées $e_i$, chaque entrée $e_i$ commandant au moins une paire d'interrupteurs (un N et un P),
- quelque soit l'état des entrées ei, un seul réseau doit être passant à la fois.
  Il est toléré toutefois que les deux branches soient ouvertes en même temps.

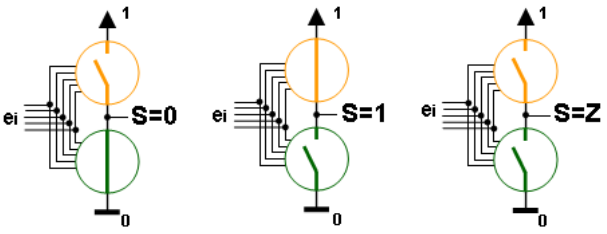



**Figure 7.6:** *Schéma du fonctionnement de la logique complémentaire*

La fonction de sortie F est générée par :
- la fermeture du réseau N, pour obtenir F = ”0” soit $\overline{F}$ (F_barre !F),
- la fermeture du réseau P, pour obtenir F = ”1” soit F,
- l'ouverture simultanée des 2 branches engendre F = Z soit l'état électrique *haute impédance*. En électronique numérique ce troisième état sert à
  - mémoriser l'état précédent,
  - ne pas influer sur une équipotentielle lorsque une autre sortie logique est censée y apporter son signal.

**Pourquoi des PMOS *en haut* et des NMOS *en bas* ?**

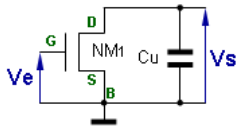
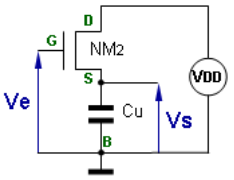
| | |
|---|---|
|  |  |
| NMOS déchargeant la capacité $C_u$ | NMOS chargeant la capacité $C_u$ |

**Table 7.6:** *Charge/décharge d'une capacité par un NMOS*

Nous savons que le temps de transition (charge ou décharge) est inversement proportionnel à l'intensité du courant traversant la capacité. Dans les cas des 2 montages étudiés ce courant est égal à celui qui traverse le dipôle de sortie du transistor : $I_{DS}$.

Dans les conditions de la logique complémentaire le transistor NMOS décharge la capacité $C_u$ du noeud de sortie (cf. “montage du transistor MN1”).
- À l'état initial :
  - $V_e = 0V$ et $V_s = V_{DD}$ : $C_u$ est chargée au maximum,
  - $V_{GS} = V_e = 0 \leq V_{T0_N}$ (la tension de seuil pour $V_{SB} = 0V$) : le transistor MN1 est bloqué. $I_{DS}$ est nul : $C_u$ reste chargée à $V_{DD}$.
- La commande de décharge arrive :
  - $V_e = V_{DD}$,
  - $V_{GS} = V_e = V_{DD} > V_{T0_N}$ : le transistor MN1 est passant saturé. $I_{DS}$ est maximum : $C_u$ se décharge,

Dans le second montage, le transistor MN2, n'est pas dans les conditions de la logique complémentaire, en effet il est branché en lieu et place de ce qui devrait être le réseau P.
- À l'état initial :
  - $V_e = 0V$ et $V_s = 0V$ : $C_u$ est déchargée au maximum,
  - $V_{GS} = V_e = 0 \leq V_{T0_N}$ (la tension de seuil pour $V_{SB} = 0V$) : le transistor MN1 est bloqué. $I_{DS}$ est nul : $C_u$ reste déchargée à 0V.
- La commande de charge arrive :
  - $V_e = V_{DD}$,
  - $V_{GS} = V_e = V_{DD} > V_{T0_N}$ : le transistor MN1 est passant saturé. $I_{DS}$ est maximum : $C_u$ se charge, $V_s$ augmente, d'où deux conséquences :
    1. La tension $V_{GS}$ diminue puisque : $V_{GS} = V_e - V_{SB} = V_e - V_s$. Si $V_{GS}$ diminue $I_{DS}$ diminue,

2. La tension $V_{SB} = V_s$ augmente, ainsi $V_{TN} > V_{T0_N}$. Or $I_{DS} \propto (V_{GS} - V_{TN})^2$ !.

**Première conclusion :** MN1 "dispose" d'une intensité de courant beaucoup plus importante pour décharger $C_u$ que MN2 pour charger $C_u$. Ainsi le temps transition du premier montage sera t il toujours plut petit que celui du second (MN1 et MN2 ayant les mêmes dimensions et les mêmes paramètres technologiques).

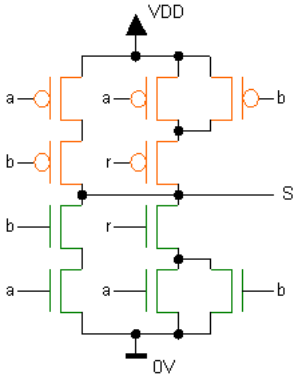
Nous pouvons mener une démonstration semblable pour un transistor PMOS : excellent exercice pour se prouver que ces notions sont correctement assimilées !

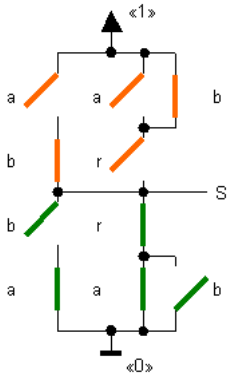
**Seconde conclusion :** de même que pour la consommation statique, la disposition d'un circuit en logique complémentaire CMOS semble optimale pour les temps de transitions.

### 7.3.4 Exemple d'analyse d'une porte logique

Sur la figure 7.7, nous avons représenté les schémas :

- du circuit en transistors de la porte logique à analyser,
- son modèle en interrupteurs,
- la table de vérité extraite du modèle en interrupteurs de la porte, où figure en gras, l'état des entrées représenté sur le schéma du modèle.




| entrées | | | réseaux | | sortie |
|---|---|---|---|---|---|
| a | b | r | N | P | S |
| 0 | 0 | 0 | O | F | 1 |
| 0 | 0 | 1 | O | F | 1 |
| 0 | 1 | 0 | O | F | 1 |
| 0 | 1 | 1 | F | O | 0 |
| 1 | 0 | 0 | O | F | 1 |
| **1** | **0** | **1** | **F** | **O** | **0** |
| 1 | 1 | 0 | F | O | 0 |
| 1 | 1 | 1 | F | O | 0 |

circuit à transistors modèle en interrupteurs table de vérité

**Table 7.7:** *Analyse d'une porte logique*

Les étapes de l'analyse, suivent évidemment les principes de la logique complémentaire CMOS.

**Méthode de la table de vérité**

- Pour chaque état logique de chaque variable d'entrée, nous en déduisons l'état de chacun des deux réseaux (N et P) puis celui de la sortie,
- L'exhaustivité de l'analyse est garantie,
- Nous vérifions qu'un même jeu de valeurs logiques d'entrée n'entraîne pas à la fois la conduction des deux réseaux,

- Nous écrivons l'équation de S en faisant la somme logique des valeurs logiques d'entrée entraînant S = 1.

$$\begin{aligned} S &= \overline{a}\cdot\overline{b}\cdot\overline{r} + \overline{a}\cdot\overline{b}\cdot r + \overline{a}\cdot b\cdot\overline{r} + a\cdot\overline{b}\cdot\overline{r} \\ &= \overline{a}\cdot\overline{b} + \overline{a}\cdot b\cdot\overline{r} + a\cdot\overline{b}\cdot\overline{r} \end{aligned}$$

en remarquant que $\overline{a}\cdot\overline{b} = \overline{a}\cdot\overline{b} + \overline{a}\cdot\overline{b}\cdot\overline{r}$, il vient :

$$\begin{aligned} S &= \overline{a}\cdot\overline{b} + \overline{a}\cdot b\cdot\overline{r} + a\cdot\overline{b}\cdot\overline{r} \\ &= \overline{a}\cdot\overline{b} + \overline{a}\cdot\overline{b}\cdot\overline{r} + \overline{a}\cdot b\cdot\overline{r} + a\cdot\overline{b}\cdot\overline{r} \\ &= \overline{a}\cdot\overline{b} + \overline{r}\cdot\left(\overline{a}\cdot\overline{b} + \overline{a}\cdot b + a\cdot\overline{b}\right) \end{aligned}$$

soit $S = \overline{a}\cdot\overline{b} + \overline{r}\cdot\left(\overline{a}+\overline{b}\right)$

Appliquons de Morgan, nous obtenons $S = \overline{a\cdot b + r\cdot(a+b)}$, que nous pouvons aussi écrire de la manière suivante :

$$S = a\cdot b + r\cdot(a+b)$$

**Méthode de l'analyse par réseau**

Nous analysons la conduction de chaque réseau (N et P) en fonction des valeurs logiques d'entrée, en appliquant les règles déjà vues (voir table 7.8).

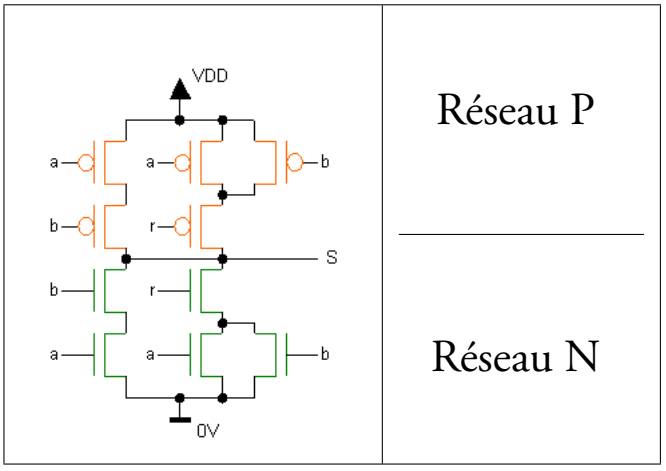



**Table 7.8:** *Analyse d'une porte logique*

**Le réseau N comprend deux branches en parallèle :**

- la branche de gauche, sur le schéma, est passante pour la fonction des entrées : $N_1 = a\cdot b$
- celle de droite est passante pour : $N_2 = r\cdot(a+b)$

Le réseau N conduit pour la fonction : $N = N_1 + N_2 = a\cdot b + r\cdot(a+b)$

La fonction réalisée par le réseau N est donc : $S = \overline{N} = \overline{a\cdot b + r\cdot(a+b)}$

**Le réseau P comprend deux branches en parallèle :**

- la branche de gauche, sur le schéma, est passante pour la fonction des entrées : $P_1 = \overline{a} \cdot \overline{b}$
- celle de droite est passante pour : $P_2 = \overline{r} \cdot \left(\overline{a} + \overline{b}\right)$

Le réseau P conduit pour la fonction : $P = P_1 + P_2 = \overline{a} \cdot \overline{b} + \overline{r} \cdot \left(\overline{a} + \overline{b}\right)$

Après calcul nous obtenons : $S = P = \overline{a} \cdot \overline{b} + \overline{r} \cdot \left(\overline{a} + \overline{b}\right) = \overline{a \cdot b + r \cdot (a + b)}$

Enfin nous vérifions la conduction exclusive de chacun des deux réseaux : $\overline{N} = P$

### 7.3.5 Exemples de synthèse d'une porte logique

**La porte et-non à 2 entrées (nand2)**

La fonction nand2 est égale à : $S_{nand2} = \overline{a \cdot b}$

- La fonction $N_{nand2}$ qui représente l'état du réseau N, vaut $N_{nand2} = \overline{S_{nand2}} = a \cdot b$
  Cette fonction correspond à deux transistor NMOS en série.
- La fonction $P_{nand2}$ qui représente l'état du réseau P, vaut $P_{nand2} = S_{nand2} = \overline{a \cdot b} = \overline{a} + \overline{b}$
  Cette fonction correspond à deux transistor PMOS en parallèle.
- Par construction, si les calculs logiques ne sont pas erronés, la réalisation satisfait la dualité des deux réseaux.

**La porte ou-non à 2 entrées (nor2)**

La fonction nor2 est égale à : $S_{nor2} = \overline{a + b}$

- La fonction $N_{nor2}$ qui représente l'état du réseau N, vaut $N_{nor2} = \overline{S_{nor2}} = a + b$
  Cette fonction correspond à deux transistor NMOS en parallèle.
- La fonction $P_{nor2}$ qui représente l'état du réseau P, vaut $P_{nor2} = S_{nor2} = \overline{a + b} = \overline{a} \cdot \overline{b}$
  Cette fonction correspond à deux transistor PMOS en série.
- Par construction, si les calculs logiques ne sont pas erronés, la réalisation satisfait la dualité des deux réseaux.

Nous obtenons les schémas de la table 7.9.

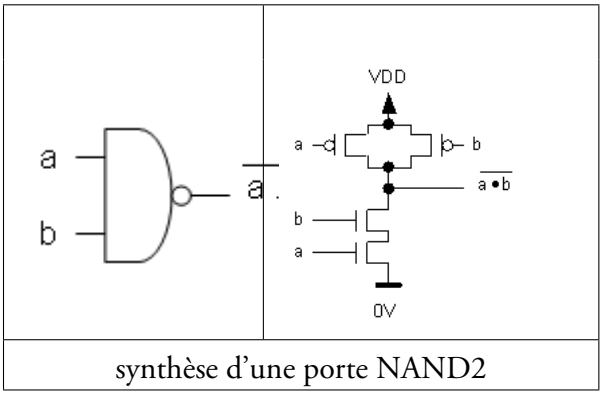
**Comment obtenir des fonctions non complémentées ?**

Ainsi que nous l'avons vu au chapitre Principes et fonctionnalité (7.3.3), si nous ne disposons que des entrées logiques naturelles (les $e_i$ et aucune $\overline{e_i}$), nous ne pouvons réaliser que des fonctions complémentées des entrées naturelles : $\overline{F}(e_i)$. Pour réaliser les fonctions logiques ET2 (and2) et OU2 (or2) nous pouvons utiliser les solutions suivantes (table 7.10) :

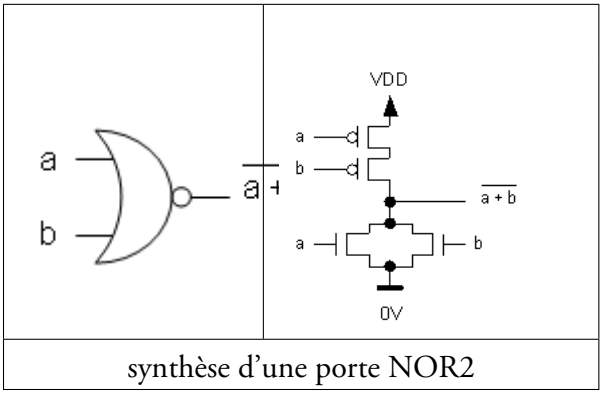
Nous pouvons aborder les exercices du TD "Synthèse en transistors de portes CMOS" (chapitre 13).

## 7.4 Vitesse de traitement d'un circuit intégré numérique CMOS

Les circuits intégrés numériques sont constitués de différents opérateurs de traitement (opérateurs arithmétiques, opérateurs de contrôle...). La nécessité de synchroniser ces opérateurs entre eux pour permettre des échanges de données conduit à ce que la vitesse de traitement «potentielle» du circuit est directement liée à la vitesse de traitement de l'opérateur le plus lent.

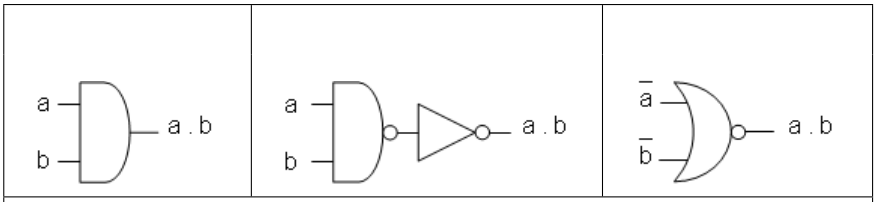



synthèse d'une porte NAND2

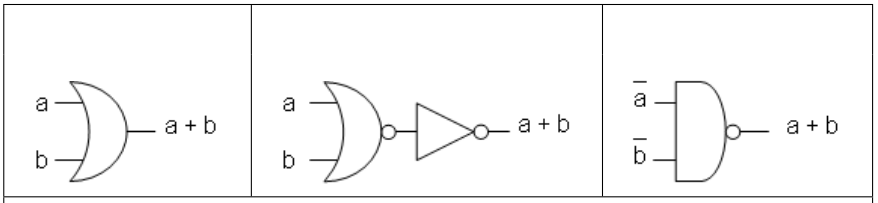



synthèse d'une porte NOR2

**Table 7.9:** *Schémas en transistors d'une porte NAND2 et d'une porte NOR2*




synthèse d'une porte AND2




synthèse d'une porte OR2

**Table 7.10:** *Synthèse de fonctions non complémentées à l'aide de portes en logique complémentaire*

Si un circuit doit contenir, par exemple, un opérateur d'addition, la connaissance du temps de calcul d'une addition est un indicateur nécessaire au concepteur pour déterminer les performances du circuit. Les techniques de réalisation de circuits intégrés numériques les plus couramment employées reposent sur l'hypothèse forte qu'il n'est possible de démarrer un nouveau calcul dans un opérateur que lorsque ses sorties se sont stabilisées. *Le temps de traitement d'un opérateur combinatoire est donc le temps nécessaire à la stabilisation des sorties de l'opérateur après la mise en place des entrées.*

### 7.4.1 Notion de chemin critique

Un opérateur combinatoire est lui-même constitué d'un assemblage de portes logiques simples ; son temps de traitement est directement lié à la propagation des signaux booléens dans les différentes portes logiques.

Considérons de nouveau l'exemple d'un additionneur combinatoire 4 bits. Un tel opérateur est une fonction à 8 entrées et 5 sorties. Entre chaque entrée et chaque sortie de l'additionneur, les signaux booléens peuvent se propager par une multitude de chemins différents traversants les différentes portes logiques. Pour déterminer la vitesse de calcul de notre additionneur, il suffit de déterminer parmi ces chemins celui qui correspond au temps de traversée le plus long. Ce chemin sera appelé *chemin critique* de l'opérateur.

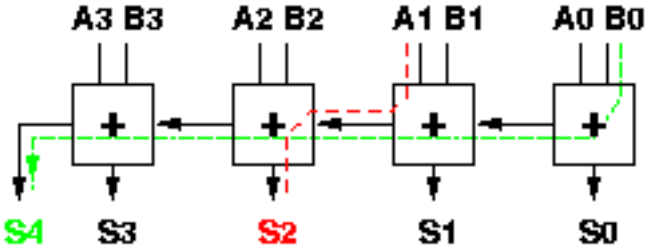



**Figure 7.7:** *Quelques chemins de propagation...*

Si nous pouvons déterminer pour chaque porte traversée le long de notre chemin critique le temps mis par le signal pour transiter de l'entrée à la sortie de la porte (temps de propagation de la porte) alors nous pouvons déterminer de manière simple le temps du chemin critique en accumulant les différents temps de propagation individuels.

### 7.4.2 Notion de temps de propagation

La définition du temps de propagation d'une porte doit permettre par simple additivité de déterminer le temps de propagation d'une chaîne de portes. La définition la plus simple consiste à mesurer le temps écoulé entre un changement d'état de l'entrée d'une porte et le changement d'état de la sortie en prenant pour référence les instants de passage des différents signaux à mi-chemin de la tension d'alimentation VDD, comme cela est représenté sur la figure 7.8.

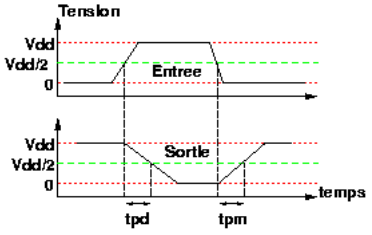



**Figure 7.8:** *Temps de propagation dans une porte*

Une porte CMOS à plusieurs entrées n'est pas caractérisée par un unique temps de propagation. Prenons l'exemple d'un NAND à 2 entrées A et B. Nous pouvons distinguer par

exemple :

- Le temps de propagation de A vers la sortie pour une transition montante de la sortie ($tpm_A$)
- Le temps de propagation de B vers la sortie pour une transition montante de la sortie ($tpm_B$)
- Le temps de propagation de A vers la sortie pour une transition descendante de la sortie ($tpd_A$)
- Le temps de propagation de B vers la sortie pour une transition descendante de la sortie ($tpd_B$)

Il n'y a pas de raison pour que ces différentes valeurs soient identiques, mais pour des raisons de simplification nous considérerons un *pire cas* en appelant temps de propagation de la porte la valeur maximum parmi ces différentes données ($tp$).

### 7.4.3 Modèle du temps de propagation d'une porte CMOS

De façon générale, le temps de propagation d'une porte peut se décomposer en deux termes.

Le premier terme représente le temps minimum nécessaire à la porte pour établir sa sortie et ce indépendamment de tout contexte externe. Ce terme appelé *temps de propagation à vide* ou *temps de propagation intrinsèque* de la porte est significatif de la complexité de la fonction logique réalisée par la porte. On peut comprendre intuitivement que le temps de propagation à vide d'un inverseur ($tp0_{INV}$) soit plus faible que celui d'un ou-exclusif à 2 entrées ($tp0_{OUEX}$) compte tenu de la différence de complexité des équation booléennes.

Le deuxième terme représente la facilité avec laquelle la porte transmet l'état de sa sortie aux différentes portes qui lui sont connectées. Pour évaluer l'impact de la connection de l'entrée d'une porte sur la sortie d'une porte précédente, il faut étudier la constitution de l'entrée d'une porte CMOS. La figure 7.9 présente un NAND à deux entrées en logique CMOS. L'entrée $A$ de la porte est connectée aux grilles d'un transistor NMOS et d'un transistor PMOS.

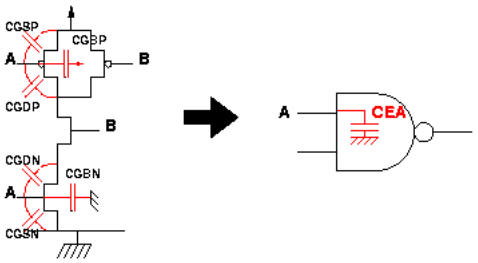



**Figure 7.9:** *Capacité d'entrée de l'entrée A d'une porte NAND*

Compte tenu de la technologie de fabrication du transistor MOS, ces grilles se comportent comme des capacités parasites dont une électrode est la grille elle même et l'autre électrode est répartie entre la source, le drain et le substrat des transistors. *Il est possible de faire l'hypothèse simplificatrice qu'une capacité parasite unique est connectée entre l'entrée A et la masse*. Cette capacité sera nommée capacité d'entrée de la porte sur l'entrée A ($CE_{A_{NAND}}$). On détermine de la même manière une capacité d'entrée sur l'entrée B.

En règle générale, *les capacités d'entrée des différentes entrées d'une porte logique sont différentes* et dépendent de la taille et du nombre de transistors dont les grilles sont connectées à ces entrées.

Maintenant que nous avons identifié la nature physique de l'entrée d'une porte CMOS, il est facile d'imaginer l'effet de sa connection sur la sortie d'une porte. La figure ci-dessous présente in inverseur dont la sortie est connectée sur l'entrée $A$ de notre porte NAND. Les transistors PMOS et NMOS de l'inverseur vont devoir alternativement fournir les courants de charge et de décharge de la capacité $CE_{A_{NAND}}$ pendant les transitions montantes et descendantes de la sortie de l'inverseur. Cette capacité $CE_{A_{NAND}}$ sera appelée plus tard *capacité d'utilisation de la porte*.

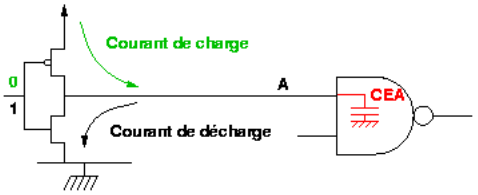



**Figure 7.10:** *Charge et décharge de la capacité d'entrée $CE_A$ d'un NAND*

Le temps nécessaire à cette charge est d'une part proportionnel à la valeur de cette capacité et d'autre part dépendant des caractéristiques des transistors constituant l'inverseur. Dans la pratique cela ce traduit par un accroissement du temps de propagation de l'inverseur par un terme de forme $dtp_{INV} \cdot CE_{A_{NAND}}$ où $dtp$ est la *dépendance capacitive du temps de propagation* de l'inverseur. Le temps total de propagation de l'inverseur est donc :

$$tp_{INV} = tp0_{INV} + dtp_{INV} \cdot CE_{A_{NAND}}$$

**En résumé, pour une porte CMOS quelconque, l'établissement du temps de propagation d'une porte CMOS nécessite la connaissance de trois termes :**

- $tp0$ **:** ***temps de propagation à vide*** **de la porte, ne dépend que de la structure physique de la porte**
- $dtp$ **:** ***dépendance capacitive de la porte*** **ne dépend que des caractéristiques physiques de la porte. Le terme dtp est équivalent à une résistance**
- $C_U$ **:** ***capacité d'utilisation*** **ne dépend que des caractéristiques des entrées des portes connectées en sortie de la porte**

**L'expression du temps de propagation de la porte chargée par $C_U$ est alors :**

$$\boldsymbol{tp = tp0 + dtp * C_U} \tag{7.1}$$

### 7.4.4 Temps de propagation dans un assemblage de portes logiques.

Nous allons illustrer sur un exemple le calcul des temps de propagation dans divers chemins d'un assemblage de portes logiques. La figure 7.11 est une fonction logique à 3 entrées (T,

U, V) et 2 sorties Y,Z. Nous supposons que les sorties Y et Z sont connectées à 2 capacités d'utilisation $CUY$ et $CUZ$.

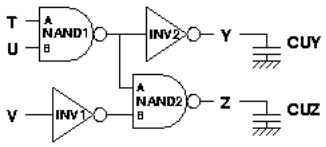



**Figure 7.11:** *Temps de propagation dans un assemblage de portes*

Nous pouvons compter six chemins (TY,UY,VY,TZ,UZ,VZ) dans cette structure et déterminer pour chacun d'eux le temps de propagation total. Exprimons, par exemple, le temps de propagation le long du chemin TZ. Le signal se propage, sur ce chemin, en traversant tout d'abord la porte NAND1. La sortie de cette porte est connectée d'une part à l'entrée de l'inverseur INV2 et d'autre part à l'entrée A de la porte NAND2. La porte NAND1 est donc chargée par les deux capacités d'utilisation connectées en parallèle $CE_{INV2}$ et $CE_{A_N AND2}$.

L'équation du temps de propagation à travers la porte NAND1 est donc :

$$tp_{NAND1} = tp0_{NAND1} + dtp_{NAND1} \cdot (CE_{INV2} + CE_{A_N AND2})$$

Le signal traverse ensuite la porte NAND2 chargée par la capacité CUZ. Le temps de propagation s'exprime de manière très simple sous la forme :

$$tp_{NAND2} = tp0_{NAND2} + dtp_{NAND2} \cdot CUZ$$

Le temps total le long du chemin TZ est égal à la somme de $tp_{NAND1}$ et de $tp_{NAND2}$, soit :

$$tp_{TZ} = tp0_{NAND1} + dtp_{NAND1} \cdot (CE_{INV2} + CE_{A_N AND2}) + tp0_{NAND2} + dtp_{NAND2} \cdot CUZ$$

Nous pourrions évidemment calculer de la même façon les temps de propagation suivant les différents chemins et déterminer ainsi le chemin critique de l'opérateur.

## 7.5 Rappels du modèle électrique

### 7.5.1 Connexions et tensions appliquées

- Les caissons, faiblement dopés N, constituent le suBstrat (B) des PMOS et leur est commun,
  Il est polarisé à la tension la plus positive du circuit $V_{DD}$.
- Le substrat, faiblement dopé P, constitue le suBstrat (B) des NMOS et leur est commun,
  Il est polarisé à la tension la plus négative du circuit $V_{SS}$, ($V_{SS} = 0V$, comme sur le schéma, parfois $V_{SS} = -V_{DD}$).
- La tension du drain (D) des transistors NMOS est toujours supérieure à celle de leur source (S),
- La tension du drain (D) des transistors PMOS est toujours inférieure à celle de leur source (S),

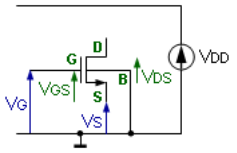
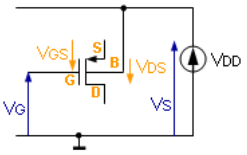
| NMOS | PMOS |
|---|---|
|  |  |



**Table 7.11:** *Connexions des transistors CMOS*

- La tension de seuil d'un NMOS : $+0,2V \leq V_{TN} \leq +2V$
- La tension de seuil d'un PMOS : $-0,2V \geq V_{TP} \geq -2V$
- La tension de la grille ($V_G$) de tous les transistors est une tension d'entrée de type logique à 2 états correspondant à deux niveaux électriques statiques :
  $V_G = V_{DD} \equiv$ "1" et $V_G = 0V \equiv$ "0"

### 7.5.2 Rappels du modèle électrique et des symboles

Voir les deux tableaux 7.12 et 7.13.

## 7.6 Bibliographie

Pour en savoir plus...


- *MosFet modeling and Bsim3 user's guide*, Yuhua CHENG and Chenming HU (Kluwer Academic Publishers, 1999).
- *Digital Integrated Circuits, a design perspective*, Jan RABAEY (Prentice Hall International Editions, 1996).
- *Dispositifs et Circuits Intégrés Semiconducteurs*, André VAPAILLE and René CASTAGNÉ (Éditions Dunod, 1987).
- *Principles of CMOS VLSI Design*, Neil WESTE and Kanrar ESHRAGHIAN, (Addison Weslay Publishing, 1985).
- *Physics of Semiconductors Devices*, S. M. SZE (Wiley Interscience Publication, 1981).

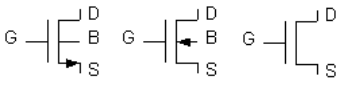
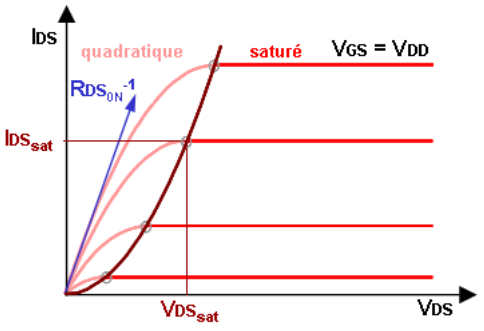
| Transistor NMOS | | | |
|---|---|---|---|
| Symboles |  | | |
| **Conditions** | | **Régime** | **Courant** |
| $V_{GS} \leq V_{TN}$ | $\forall V_{DS}$ | bloqué | $I_{DS} = 0$ |
| $V_{GS} > V_{TN}$ | $V_{DS} < V_{DS_{sat}}$ | quadratique | $I_{DS} = 2 \cdot K_n \cdot \left(V_{GS} - V_{TN} - \frac{V_{DS}}{2}\right) \cdot V_{DS}$<br>$K_n = \frac{1}{2}\mu_{0N} \cdot C'_{ox}\frac{W}{L}$<br>et $V_{DS_{sat}} = V_{GS} - V_{TN}$ |
| | $V_{DS} \geq V_{DS_{sat}}$ | saturé | $I_{DS_{sat}} = K_n \cdot (V_{GS} - V_{TN})^2$ |
| $V_{GS} = V_{DD}$ | $V_{DS} \approx 0$ | ohmique | $R_{DS_{0N}} = \frac{1}{2 \cdot K_n \cdot (V_{DD} - V_{TN})}$ |
| $I_{DS} = (V_{GS}, V_{DS})$ | | | |
|  | | | |



**Table 7.12:** *Courant et résistance équivalente du NMOS*

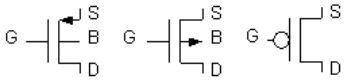
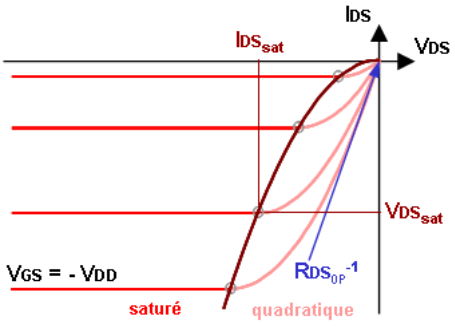
<table>
<tr><td colspan="4">Transistor PMOS</td></tr>
<tr><td>Symboles</td><td colspan="3"> 
</td></tr>
<tr><td colspan="2">Conditions</td><td>Régime</td><td>Courant</td></tr>
<tr><td>$V_{GS} \geq V_{TP}$</td><td>$\forall V_{DS}$</td><td>bloqué</td><td>$I_{DS} = 0$</td></tr>
<tr><td rowspan="2">$V_{GS} < V_{TP}$</td><td>$V_{DS} > V_{DS_{sat}}$</td><td>quadratique</td><td>$I_{DS} = -2 \cdot K_p \cdot \left(V_{GS} - V_{TP} - \frac{V_{DS}}{2}\right) \cdot V_{DS}$<br>$K_p = \frac{1}{2}\mu_{0P} \cdot C'_{ox}\frac{W}{L}$<br>et $V_{DS_{sat}} = V_{GS} - V_{TP}$</td></tr>
<tr><td>$V_{DS} \leq V_{DS_{sat}}$</td><td>saturé</td><td>$I_{DS_{sat}} = -K_p \cdot (V_{GS} - V_{TP})^2$</td></tr>
<tr><td>$V_{GS} = -V_{DD}$</td><td>$V_{DS} \approx 0$</td><td>ohmique</td><td>$R_{DS_{0P}} = -\frac{1}{2 \cdot K_p \cdot (-V_{DD} - V_{TP})}$</td></tr>
<tr><td colspan="4">$I_{DS} = (V_{GS}, V_{DS})$</td></tr>
<tr><td colspan="4"> 
</td></tr>
</table>

**Table 7.13:** *Courant et résistance équivalente du PMOS*

# Chapitre 8

# Performances de la logique complémentaire CMOS

## 8.1 Introduction

La réalisation de systèmes électroniques de traitements numériques efficaces suppose le respect d'un certain nombre de contraintes liées à des aspects très divers. Nous allons limiter notre étude aux trois paramètres suivants :

- Le coût de production ;
- La vitesse de traitement ;
- La consommation.

Ces trois paramètres ne sont évidemment pas décorrélés, l'augmentation d'une vitesse de traitement se fait souvent au prix d'une consommation et d'un coût de production accru.

Parfois certain critères sont impératifs : si nous considérons, par exemple, une application de traitement numérique pour une transmission d'images de télévision, les calculs doivent être effectués à la volée sans possibilité d'interrompre le flux de données. Nous disons dans ce cas que le système doit avoir la puissance de calcul (ou vitesse de traitement) nécessaire pour respecter le temps réel. De manière moins impérative, si nous considérons une application de bureautique sur un ordinateur personnel, il suffit que le système ait une puissance de calcul suffisante pour garantir un certain confort à l'utilisateur.

De même, il est aisément concevable que les besoins en terme de modération de la consommation d'un système alimenté par batterie soient différents de ceux d'un système connectable au réseau électrique.

Les ingénieurs réalisant des circuits intégrés numériques sont constamment confrontés au problème de l'évaluation de ces paramètres. Nous allons montrer, dans ce chapitre, quelques méthodes simples d'évaluation basées notamment sur notre connaissance de la technologie de fabrication des circuits intégrés.

## 8.2 Coût de production d'un circuit intégré numérique CMOS

Le coût de production d'un circuit intégré est étroitement lié à l'aire du silicium nécessaire à sa réalisation. En effet, plus le circuit est de taille importante, plus le rendement de fabrication est faible. Le rendement de fabrication représente le rapport entre le nombre de circuits fonctionnels produits et le nombre total de circuits produits. La probabilité d'avoir un défaut

dans un circuit augmentant avec sa taille, le concepteur a tout intérêt à minimiser la surface de silicium nécessaire à la réalisation de l'application qui l'intéresse. Évidemment cette surface dépend du nombre de transistors utilisés pour réaliser l'application et de la surface de chacun de ces transistors.

En électronique numérique intégrée, les transistors utilisés étant de taille relativement standard (sauf fonctions exceptionnelles), on peut considérer que le nombre de transistors est un bon représentant de la surface du circuit intégré. On caractérise d'ailleurs les technologies CMOS numériques par le nombre de transistors qu'elles sont capables d'intégrer. En 2004, les densités d'intégration des technologies les plus avancées étaient de l'ordre de 1 500 000 transistors par $mm^2$.

Par conséquent, en passant du niveau de l'application au niveau de la porte logique, minimiser le nombre de transistors nécessaires à la réalisation de portes logiques contribue à minimiser l'aire globale d'un circuit.

## 8.3 Estimation de la vitesse de la logique CMOS

### 8.3.1 Expression du temps de propagation d'un inverseur CMOS

Nous voulons exprimer le temps de propagation en descente $t_{pd}$ de l'inverseur $INV_1$ de la figure 8.1 à partir de la connaissance du transistor MOS dont les caractéristiques électriques sont rappelées au chapitre 7.5. Compte tenu de la complexité des phénomènes mis en jeu, la mise au point d'un modèle analytique du temps de propagation d'une porte CMOS (même aussi simple qu'un inverseur) n'est guère envisageable. Aussi nous contenterons-nous d'en faire une estimation à partir d'un grand nombre d'hypothèses simplificatrices.

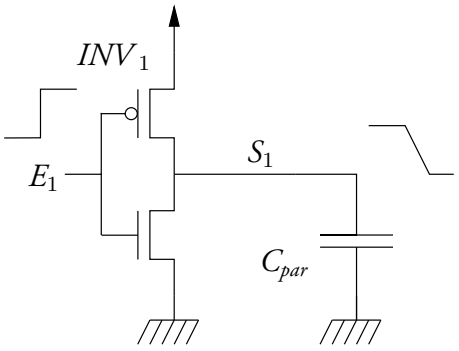



**Figure 8.1:** *Étude de cas de l'inverseur CMOS.*

Nous modélisons l'ensemble des effets parasites pouvant perturber le fonctionnement de cet inverseur par une unique capacité $C_{par}$ connectée entre la sortie $S_1$ et la masse. Les phénomènes électriques observés à la suite d'une transition montante du signal d'entrée $E_1$ sont les suivants :

- bloquage du transistor PMOS et débloquage du transistor NMOS ;
- donc la décharge de la capacité $C_{par}$ à travers le transistor NMOS ;
- donc transition descendante du signal de sortie $S_1$.

Une vision réaliste des évolutions des signaux $E_1$ et $S_1$ au cours du temps est reproduite en figure 8.2.

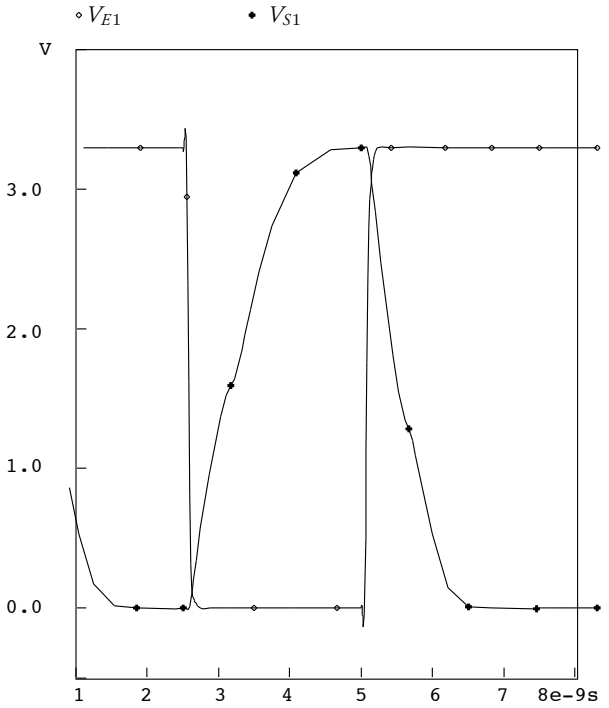



**Figure 8.2:** *Simulation électrique de l'inverseur CMOS.*

Nous supposons que le signal d'entrée $E_1$ passe en un temps très bref de l'état « 0 » à l'état « 1 ». Ce temps sera considéré comme suffisamment bref pour ne pas influer sur le temps de propagation de l'inverseur. En exploitant la définition du temps de propagation proposée dans le chapitre 7.4.2, nous distinguons trois instants comme illustré dans la figure 8.3 :

- « $0^-$ » : instant de passage du signal d'entrée $E_1$ à $V_{dd}/2$ ;
- « $0^+$ » : instant d'arrivée du signal d'entrée $E_1$ à $V_{dd}$ ;
- « $t_{pd}$ » : instant de passage du signal de sortie $S_1$ à $V_{dd}/2$.

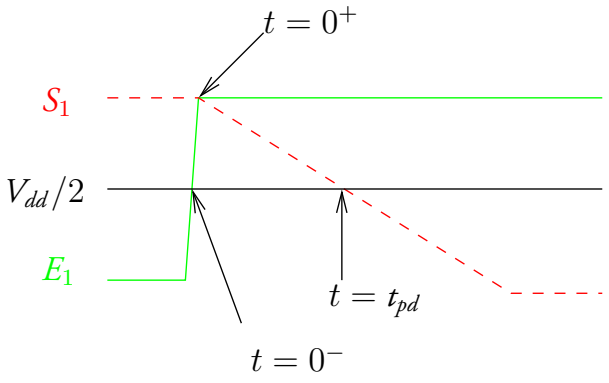



**Figure 8.3:** *Conditions de mesures des transitions des signaux.*

Le tableau 8.1 résume les valeurs des différentes tensions aux bornes des transistors de l'inverseur $INV_1$au début (instant $0^+$) et à la fin (instant $t_{pd}$) de la propagation du signal de sortie $S_1$.

La tension $V_{gsp}$ garde la valeur 0 durant toute la transition de la sortie. Le transistor PMOS

| | Instant « $0^+$ » | | Instant « $t_{pd}$ » | |
|---|---|---|---|---|
| Transistor NMOS | $V_{gsn} = V_{dd}$ | $V_{dsn} = V_{dd}$ | $V_{gsn} = V_{dd}$ | $V_{dsn} = V_{dd}/2$ |
| Transistor PMOS | $V_{gsp} = 0$ | $V_{dsp} = 0$ | $V_{gsp} = 0$ | $V_{dsp} = -V_{dd}/2$ |

**Table 8.1:** *Tensions aux bornes de transistors pour les instants $0^+$ et $t_{pd}$.*

reste donc bien bloqué pendant toute la transition de la sortie : il n'est traversé par aucun courant ($I_{dsp} = 0$).

La tension $V_{gsn}$ garde la valeur $V_{dd}$ durant toute la transition de la sortie. Le transistor NMOS est donc passant, le courant qui le traverse peut être évalué à l'aide de la figure 8.4 qui représente la caractéristique $I_{dsn} = f(V_{dsn})$ pour $V_{gsn} = V_{dd}$.

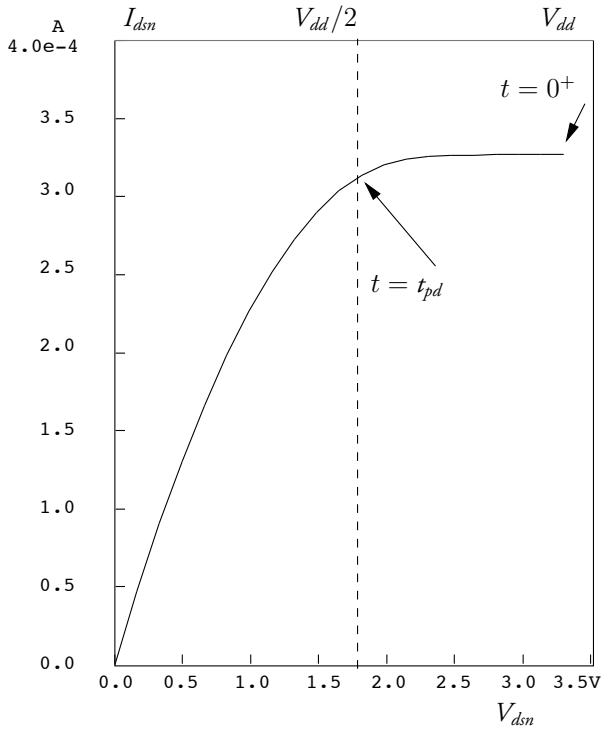



**Figure 8.4:** *Évolution du courant drain-source du transistor NMOS durant la transition descendante.*

À l'instant $0^+$, le courant est maximal et correspond au *courant de saturation* du transistor NMOS. Ensuite, la décharge de la capacité $C_{par}$ entraîne une baisse de la tension $V_{dsn}$ qui se traduit par une diminution du courant fourni par le transistor. La figure 8.4 montre cependant clairement que lorsque la sortie $S_1$ de l'inverseur atteint $V_{dd}/2$ le courant $I_{dsn}$ du transistor n'a que faiblement évolué : on peut, en première approximation, considérer que le courant fourni par le transistor est constant pendant toute la durée $t_{pd}$.

*Le courant de décharge de la capacité parasite $C_{par}$ pendant l'intervalle de temps $[0, t_{pd}]$ est donc égal au courant de saturation du transistor NMOS pour $V_{gsn} = V_{dd}$ :*

$$I_{C_{par}} = K_n \cdot (V_{dd} - V_{tn})^2$$

Le courant de décharge étant constant nous en déduisons directement la valeur de $t_{pd}$ :

$$t_{pd} = C_{par}\frac{\Delta V}{I} = C_{par}\frac{V_{dd}/2}{K_n \cdot (V_{dd} - V_{tn})^2}$$

En négligeant $V_{tn}$ devant $V_{dd}$ puis en introduisant la résistance $R_{ds0n}$ du transistor NMOS en régime ohmique, l'expression précédente se simplifie en :

$$t_{pd} = \frac{C_{par}}{2 \cdot K_n \cdot (V_{dd} - V_{tn})} = C_{par} \times R_{ds_{0n}}$$

### 8.3.2 Modèle du temps de propagation de l'inverseur CMOS

Notre objectif est, ici, d'affiner le modèle du temps de propagation en examinant l'origine des capacités parasites contribuant à la valeur de $C_{par}$. De manière générale, nous pouvons distinguer trois types de capacités :

1. Les capacités internes propres à l'inverseur $INV_1$ ;
2. Les capacités dues aux liaisons entre l'inverseur $INV_1$ et les différentes portes logiques connectées à sa sortie ;
3. Les capacités d'entrées des portes logiques connectées à la sortie $S_1$ de l'inverseur.

L'effet des capacités internes à la porte sera assimilé à celui d'une unique capacité $C_s$ appelée *capacité de sortie* de l'inverseur et connectée entre la sortie $S_1$ et la masse. L'effet des capacités de liaison et des capacités d'entrées des portes connectées à la sortie de l'inverseur sera assimilé à une unique capacité $C_u$ appelée *capacité d'utilisation* de l'inverseur. Ainsi l'expression de $t_{pd}$ peut être réorganisée de la façon suivante :

$$t_{pd} = (C_s + C_u) \times R_{ds_{0n}}$$

Soit

$$t_{pd} = t_{p0d} + d_{tpd} \times C_u \text{ avec } d_{tpd} = R_{ds_{0n}} \text{ et } t_{p0d} = d_{tpd} \times C_s$$

Nous retrouvons, appliquée au cas spécifique d'une transition descendante, la formulation du temps de propagation proposée dans le chapitre 7.4.3. Pour cela nous avons introduit le *temps de propagation en descente à vide* de l'inverseur $t_{p0d}$ ainsi que la *dépendance capacitive du temps de propagation en descente* $d_{tpd}$.

Le raisonnement effectué pour une transition descendante de la sortie peut être appliqué à la transition montante. Dans ce cas, seul le transistor PMOS est actif, les capacités parasites restant identiques, la seule différence provient de la valeur de la résistance en régime ohmique du transistor PMOS qui n'est pas forcément identique à celle du transistor NMOS.

$$t_{pm} = t_{p0m} + d_{tpm} \times C_u \text{ avec } d_{tpm} = R_{ds_{0p}} \text{ et } t_{p0m} = d_{tpm} \times C_s$$

### 8.3.3 Schéma synthétique de l'inverseur

La figure 8.5 propose un schéma synthétique de l'inverseur basé sur un interrupteur, deux résistances de valeurs respectives $R_{ds0n}$ et $R_{ds0p}$, et enfin les capacités $C_{eINV}$ et $C_{sINV}$. Suivant la valeur de la tension d'entrée, l'interrupteur bascule d'un état à l'autre provoquant la charge ou la décharge du nœud de sortie. L'état représenté correspond à une entrée égale à « 0 ». La

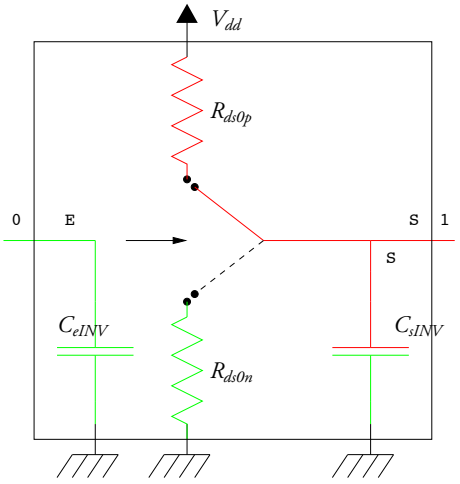



**Figure 8.5:** *Schéma synthétique de l'inverseur CMOS.*

figure 8.6 illustre la mise en série de deux inverseurs. La sortie du premier inverseur (à l'état « 1 ») présente une capacité parasite $C_{par}$ totale égale à la somme de la capacité de sortie $C_{sINV}$ du premier inverseur et de la capacité d'entrée $C_{eINV}$ du deuxième inverseur. Le temps de propagation en montée $t_{pm}$ du premier inverseur est donc $t_{pm} = (C_{eINV} + C_{sINV})R_{ds0p}$.

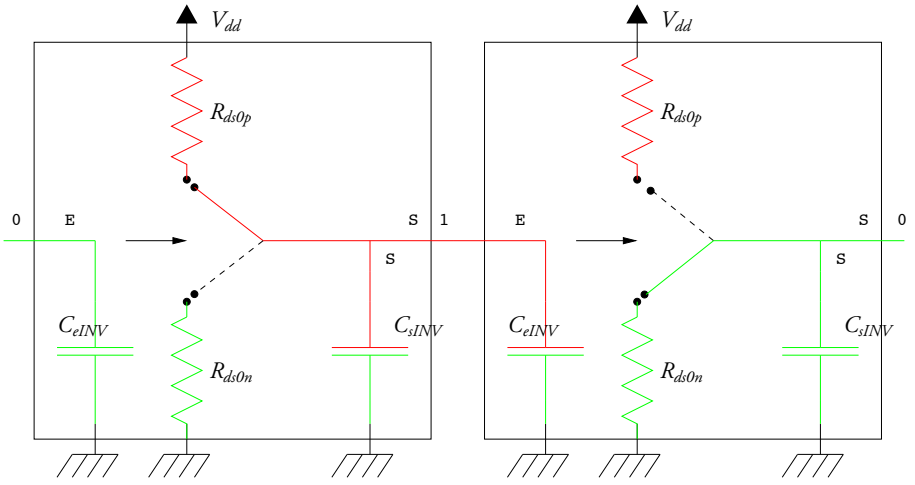



**Figure 8.6:** *Deux inverseurs en série.*

### 8.3.4 Schéma synthétique d'une porte CMOS quelconque

L'extrapolation à une porte CMOS quelconque se fait en suivant les principes suivants :

- Il y a autant de « capacités d'entrée » qu'il y a d'entrées dans la porte. L'estimation d'une capacité d'entrée se fait en sommant les capacités des grilles des transistors connectés à cette entrée.
- On peut grossièrement estimer la « capacité de sortie » en ne considérant que les capacités parasites connectées au nœud de sortie de la porte. Les capacités des nœuds intermédiaires seront négligées.

- On peut établir un équivalent à la résistance $R_{ds0}$ des transistors de l'inverseur en considérant le pire cas de mise en série des transistors et en faisant la somme des différentes résistances pour ce pire cas.

Prenons l'exemple d'une porte NAND (figure 8.7) à deux entrées utilisant des transistors identiques à ceux de l'inverseur :

- les capacités d'entrées sur les entrées A et B sont identiques à celle de l'inverseur ($C_{eINV}$) ;
- la capacité de sortie est supérieure à celle de l'inverseur ($C_{sINV}$), car deux transistors PMOS et un transistor NMOS sont connectés au nœud de sortie ;
- dans le pire cas, la résistance équivalente à la descente est égale à deux fois celle de l'inverseur ($2 \times R_{ds0n}$) ;
- dans le pire cas, la résistance équivalente à la montée est égale à celle de l'inverseur ($R_{ds0p}$).

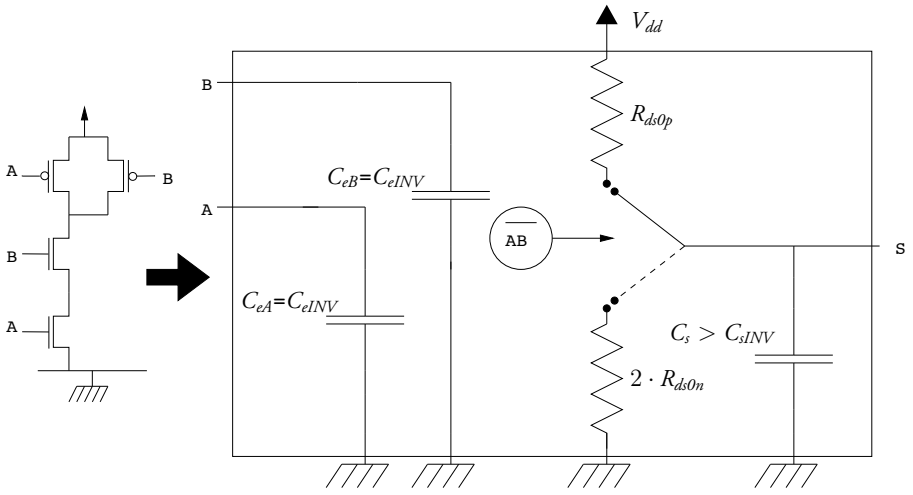



**Figure 8.7:** *Schéma synthétique d'une porte NAND.*

### 8.3.5 Notion de bibliothèque de cellules précaractérisées

Les sociétés de fonderies de silicium, ou « fondeurs », qui produisent des circuits intégrés numériques, proposent à leurs clients, des bibliothèques de portes logiques dites précaractérisées. Les ingénieurs de ces sociétés développent, dessinent et simulent le comportement et les performances de chacune des portes logiques de la bibliothèque. Ils fournissent à leurs clients des tables de caractéristiques permettant à ces derniers de concevoir des circuits intégrés et prédire leurs performances sans avoir à explorer des niveaux de détail allant jusqu'au transistor. Le tableau 8.2 propose un exemple simple d'une telle bibliothèque dont les caractéristiques globales sont les suivantes :

- les transistors NMOS des différentes portes sont tous de dimensions identiques ;
- les transistors PMOS des différentes portes sont tous de dimensions identiques ;
- les portes sont réalisées en utilisant exclusivement les principes de construction exposés dans le chapitre 7.

Nous pouvons faire les constatations suivantes :

- les temps de propagation s'expriment en dixièmes de nano-secondes ;
- les capacités s'expriment en dizaines de femto-farads ($10^{-15}$ farads) ;

| Fonction | Équation booléenne | $C_{ei}$ (fF) | $t_{p0}$ (ns) | $d_{tp}$ (ns/pF) |
|---|---|---|---|---|
| Inverseur | $Y = \overline{A}$ | $C_{eA} = 70$ | 0,06 | 1 |
| Nand à 3 entrées | $Y = \overline{ABC}$ | $\forall i \in \{A, B, C\} C_{ei} = 70$ | 0,42 | 3 |
| Nand à 6 entrées | $Y = \overline{ABCDEF}$ | $\forall i \in \{A \cdots F\} C_{ei} = 70$ | 1,56 | 6 |
| Nor à 2 entrées | $Y = \overline{A + B}$ | $\forall i \in \{A, B\} C_{ei} = 70$ | 0,16 | 2 |
| Nor à 6 entrées | $Y = \overline{A + B + C + D + E + F}$ | $\forall i \in \{A \cdots F\} C_{ei} = 70$ | 0,96 | 6 |
| nMaj à 3 entrées | $Y = \overline{AB + BC + AC}$ | $C_{eA} = C_{eB} = 140$<br>$C_{eC} = 70$ | 0,25 | 2 |

**Table 8.2:** *Une bibliothèque précaractérisée.*

- les dépendances temporelles s'expriment en nano-secondes par pico-farad (équivalentes à des kΩ) ;
- l'inverseur, porte booléenne la plus simple que l'on puisse imaginer est à la fois intrinsèquement la plus rapide ($t_{p0}$) et la moins dépendante de l'environnement extérieur ($d_{tp}$).

De plus, conformément au modèle proposé dans le chapitre 8.3.4, la porte NAND à 6 entrées est la porte ayant les performances les moins bonnes. En effet, dans le pire cas, la décharge du nœud de sortie se fait à travers 6 transistors NMOS connectés en série. De plus 6 transistors PMOS étant connectés en parallèle sur la sortie, la capacité de sortie est très élevée ce qui donne un $t_{p0}$ catastrophique (20 fois plus grand que celui de l'inverseur). Cet exemple montre une des limitations de la construction de portes en logique CMOS. En général, on préfère limiter la bibliothèque à des portes à 4 ou 5 entrées maximum, quitte à créer des assemblages de portes pour réaliser des fonctions booléennes complexes.

Enfin, les capacités d'entrées $C_{ei}$ des portes simples de type NAND ou NOR sont identiques à celles d'un inverseur, car chaque entrée de ce type de porte est connectée à un couple de transistors NMOS et PMOS comme dans le cas de l'inverseur. Ceci n'est pas un cas général, l'exemple de la porte nMAJ montre qu'il peut y avoir différentes valeurs de capacités d'entrée suivant la manière dont la logique est réalisée.

### 8.3.6 Influence du dimensionnement des transistors sur les caractéristiques de l'inverseur

Dans le paragraphe précédent, nous avons examiné les caractéristiques de portes CMOS constitués de transistors NMOS (ou PMOS) de dimensions standardisées. Il est possible d'optimiser le comportement temporel des portes en jouant sur les dimensions des transistors qui les composent. Nous n'étudierons ici que le cas simple de l'inverseur.

Nous avons vu que le temps de propagation est proportionnel à la résistance $R_{ds0}$ des transistors en régime ohmique. En ce qui concerne le transistor NMOS, cette résistance a pour expression :

$$R_{ds_{0n}} = \frac{1}{\mu_{0N} C'_{OX} \frac{W_n}{L_n} (V_{dd} - V_{tn})}$$

À l'évidence, nous pouvons augmenter les performances de l'inverseur en diminuant la longueur $L_n$ du transistor NMOS ou en augmentant sa largeur $W_n$. En règle générale, tous les transistors sont dimensionnés avec la longueur de grille minimale $L_{min}$ autorisée par la technologie. C'est pour cette raison qu'une technologie est souvent qualifiée par cette longueur de grille minimale : on parle, par exemple, d'une technologie « 0,09 microns ». Donc en réalité, seul le paramètre $W_n$ peut servir de variable d'ajustement.

*Retenons que les transistors sont le plus souvent dimensionnés avec une longueur de grille minimale $L_{min}$.*

*Retenons que la dépendance capacitive $d_{tpdINV}$ est inversement proportionnelle à la largeur $W_n$ du transistor NMOS.*

*Retenons que la dépendance capacitive $d_{tpmINV}$ est inversement proportionnelle à la largeur $W_p$ du transistor PMOS.*

Cependant, l'augmentation de la largeur des transistors a pour conséquence l'augmentation de la valeur de la capacité parasite à charger. En effet, la capacité $C_s$ est la résultante des capacités parasites propres aux transistors composant l'inverseur, toutes proportionnelles à la largeur des transistors.

*Retenons que la capacité de sortie $C_{sINV}$ de l'inverseur est la somme de deux termes, l'un étant proportionnel à la largeur $W_n$ du transistor NMOS, l'autre étant proportionnel à la largeur $W_p$ du transistor PMOS.*

Un raisonnement identique peut être fait pour la capacité d'entrée de l'inverseur. Cette capacité ne dépend que des capacités de grille $C_{gsn}$ et $C_{gsp}$ des deux transistors :

*Retenons que la capacité d'entrée $C_{eINV}$ de l'inverseur est la somme de deux termes, l'un étant proportionnel à la largeur $W_n$ du transistor NMOS, l'autre étant proportionnel à la largeur $W_p$ du transistor PMOS.*

Ces résultats partiels, permettent de déterminer une règle générale de dimensionnement des transistors d'un inverseur : disposant d'un inverseur donné de caractéristiques ($C_{eINV}$, $t_{p0INV}$, $d_{tpINV}$) connues, la multiplication des largeurs $W_n$ et $W_p$ des deux transistors NMOS et PMOS par un même coefficient $\alpha$ modifie les caractéristiques de l'inverseur de la façon suivante :

- la capacité d'entrée de l'inverseur $C_{eINV}$ est multipliée par $\alpha$ ;
- la dépendance capacitive de l'inverseur $d_{tpINV}$ est divisée par $\alpha$ ;
- le temps de propagation à vide $t_{p0INV}$ reste inchangé.

Le dernier résultat est dû aux effets contraires des augmentations de capacités internes et de diminution des résistances.

## 8.4 Consommation des circuits intégrés CMOS

### 8.4.1 Consommation d'une porte CMOS

Rappelons (voir chapitre 7.3.2) que la *consommation statique* (la porte étant dans l'un de ses deux états d'équilibre) d'une porte logique complémentaire CMOS, est NULLE. Nous nous intéressons, dans ce qui suit, à la *consommation dynamique*, c'est-à-dire à la consommation nécessaire au passage de la porte CMOS d'un état d'équilibre à un autre. Si nous faisons l'hypothèse que les deux réseaux NMOS et PMOS d'une même porte logique ne sont jamais simultanément actifs (passants) alors la consommation d'une porte CMOS se résume à

l'énergie nécessaire pour charger les différentes capacités parasites connectées sur la sortie de la porte :

- pendant une transition montante de la sortie de la porte, l'alimentation fournit le courant qui, au travers du réseau de transistor PMOS, charge la capacité connectée au nœud de sortie ;
- pendant une transition descendante de la sortie, la capacité de sortie est déchargée à travers le réseau NMOS.

Comme l'illustre la figure 8.8, l'énergie fournie par l'alimentation est dissipée par effet Joule dans les réseau de transistors PMOS (respectivement NMOS) pendant les transitions montantes (respectivement descendantes) de la sortie de la porte.

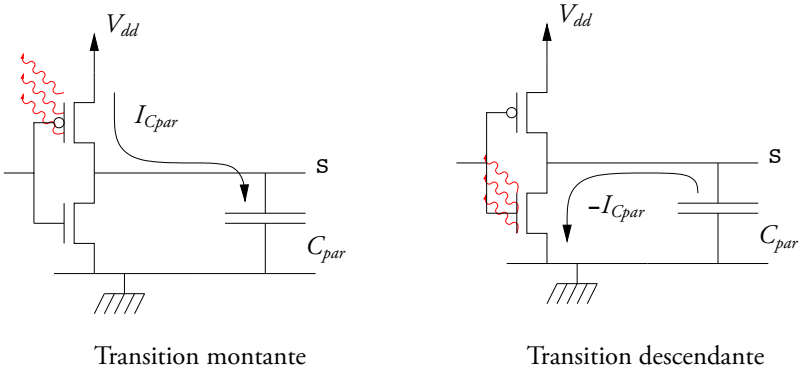



**Figure 8.8:** *Dissipation de l'énergie dans une porte CMOS.*

La puissance instantanée fournie par l'alimentation $V_{dd}$ pendant la charge de la capacité $C_{par}$ s'exprime de la façon suivante :

$$P_{V_{dd}}(t) = V_{dd} I_{C_{par}} = V_{dd} C_{par} \frac{\mathrm{d}V_s}{\mathrm{d}t}$$

L'énergie totale fournie par l'alimentation s'obtient en intégrant la puissance pendant la durée totale de la transition :

$$E_{V\,dd} = \int_0^{\infty} V_{dd} C_{par} \frac{\mathrm{d}V_s}{\mathrm{d}t} \mathrm{d}t = C_{par} V_{dd} \int_0^{V_{dd}} \mathrm{d}V_s = C_{par} V_{dd}^2$$

De manière similaire, l'énergie potentielle stockée dans la capacité après la charge est :

$$E_{Cpar} = \int_0^{\infty} V_s C_{par} \frac{\mathrm{d}V_s}{\mathrm{d}t} \mathrm{d}t = C_{par} \int_0^{V_{dd}} V_s \mathrm{d}V_s = C_{par} \frac{V_{dd}^2}{2}$$

Ces résultats montrent que seule la moitié de l'énergie fournie par l'alimentation est stockée dans la capacité. L'autre moitié est dissipée par effet Joule dans le réseau de transistors PMOS. Remarquons que cette énergie dissipée est indépendante de la résistance équivalente des transistors qui composent le réseau PMOS.

Une étude similaire pour la décharge montre que l'énergie potentielle stockée dans la capacité est entièrement dissipée par effet Joule dans le réseau de transistors NMOS.

*En résumé, chaque transition de la sortie d'une porte CMOS se traduit par une dissipation de $C_{par}{V_{dd}}^2/2$ dans la porte CMOS, où $C_{par}$ est la capacité parasite totale chargée par la porte et $V_{dd}$ est la tension d'alimentation de la porte.*

Connaissant l'énergie par transition, il est possible d'en déduire la consommation moyenne d'une porte logique. Pour cela,nous devons introduire le rythme moyen $F_{trans}$ de changement d'état de la porte que nous nommerons *fréquence de transitions* :

$$P_{porte} = \frac{F_{trans} C_{par} V_{dd}^2}{2}$$

### 8.4.2 Extrapolation à un circuit intégré CMOS

La formule décrivant la consommation d'une porte CMOS peut être extrapolée au niveau d'un circuit intégré complet. La grande majorité des circuits intégrés numériques sont des circuits dit « séquentiels » et « synchrones ». Leur principe de fonctionnement est basé sur le cadencement des opérations de calcul par une horloge externe dont la fréquence est nommée $F_h$. La fréquence de transitions $F_{trans}$ des portes CMOS qui composent un circuit est évidemment étroitement corrélée à la fréquence d'horloge du circuit, elle reste cependant plus faible que cette dernière car elle dépend de la nature des fonctions logiques exécutées par les portes et des différentes corrélations entre les signaux booléens internes au circuit. Pour tenir compte de cela, on introduit $T_{act}$, *taux d'activité* moyen des portes du circuit intégré, qui représente la probabilité de transition des portes à chaque période de l'horloge $F_h$. L'ensemble des capacités parasites du circuit peut être alors aggloméré en un seul terme $C_{total}$conduisant à l'expression de la consommation du circuit intégré :

$$P_{circuit} = T_{act} F_h C_{total} V dd^2$$

## 8.5 Évolution technologique et conclusions

Nous avons évalué de manière simple trois critères permettant d'évaluer les performances des circuits intégrés CMOS. Les résultats obtenus montrent à l'évidence que certains compromis doivent être faits.

Le temps de propagation des portes est inversement proportionnel à la tension d'alimentation, on peut en déduire un peu rapidement qu'il suffit d'augmenter la tension d'alimentation $V_{dd}$ pour augmenter les performances des circuits. Cependant, nous devons constater que la consommation des circuits varie comme le carré de la tension d'alimentation donc toute augmentation de celle-ci diminue le rendement d'utilisation de la technologie...

C'est pourquoi, les travaux d'amélioration de la technologie ont porté sur la diminution des capacités, termes présents en facteur à la fois dans l'expression des temps de propagation et de la consommation. Le moyen le plus simple de réduire les capacités est la réduction des géométries des transistors qui permet de gagner en même temps sur la surface des circuits intégrés. Les moyens technologiques et financiers mis en jeu par les fonderies de silicium sont essentiellement tournés vers cette réduction des dimensions des transistors.

Nous allons maintenant évaluer l'impact des réductions de dimension à partir des formules établies dans les paragraphes précédents. Le principe généralement employé d'une génération technologique à une autre est d'accompagner les réductions géométriques de modifications

des dopages et tensions d'alimentations pour conserver les caractéristiques fonctionnelles des transistors. En résumé les différents paramètres suivants sont touchés :

- division d'un facteur $\beta$ de la largeur $W$ des transistors ;
- division d'un facteur $\beta$ de la longueur $L$ des transistors ;
- division d'un facteur $\beta$ de l'épaisseur d'oxyde de grille $T_{ox}$ des transistors ;
- division d'un facteur $\beta$ de la tension de seuil $V_t$des transistors ;
- division d'un facteur $\beta$ de la tension d'alimentation $V_{dd}$du circuit.

*A fonctionnalité identique, le changement de génération technologique permet de réaliser des circuits de surface $\beta^2$ fois plus petite !*

La résistance équivalente des transistors devient :

$$R_{ds_0}(\beta) = \frac{1}{\mu_0(\beta C'_{ox})\frac{W}{\beta}\frac{\beta}{L}\frac{V_{dd}-V_t}{\beta}} = R_{ds_0}$$

Donc la résistance équivalente des transistors ne varie pas.

Si, par simplification, nous réduisons les effets parasites des transistors aux capacités de grille alors ces capacités de charge deviennent :

$$C_{par}(\beta) = (W/\beta)(L/\beta)(\beta C'_{ox}) = \frac{C_{par}}{\beta}$$

Donc les capacités parasites sont divisées d'un facteur $\beta$. En conséquence, les temps de propagation des portes (produit RC) sont divisés d'un facteur $\beta$ :

$$t_p(\beta) = \frac{t_p}{\beta}$$

*À fonctionnalité identique, le changement de génération technologique permet de réaliser des circuits de surface $\beta^2$ fois plus petite, ayant une vitesse de fonctionnement $\beta$ plus élevée !*

Cependant l'énergie dissipée par une porte pendant une transition devient :

$$E_{porte}(\beta) = \frac{\frac{C_{par}}{\beta}(\frac{V_{dd}}{\beta})^2}{2} = \frac{E_{porte}}{\beta^3}$$

Donc l'énergie est divisée par un facteur $\beta^3$ ce qui est extrapolable au circuit.

Si nous profitons de l'augmentation de vitesse des portes logiques pour augmenter la vitesse d'horloge du circuit alors la puissance consommée par le circuit devient :

$$P_{circuit}(\beta) = T_{act}(\beta F_h)\frac{E_{circuit}}{\beta^3} = \frac{P_{circuit}}{\beta^2}$$

*À fonctionnalité identique, le changement de génération technologique permet de réaliser des circuits de surface $\beta^2$ fois plus petite, ayant une vitesse de fonctionnement $\beta$ plus élevée et dont la consommation est $\beta^2$ fois moins élevée !*

Supposons maintenant que nous profitions de la réduction de taille des transistors pour réaliser un circuit plus complexe. À surface identique, nous pouvons multiplier par $\beta^2$ le nombre de transistors dans le circuit et donc multiplier par $\beta^2$ la capacité parasite totale à charger :

*Le changement de génération technologique permet de réaliser des circuits $\beta^2$ fois plus complexes, fonctionnant à une vitesse $\beta$ fois plus élevée et ayant une consommation identique aux circuits de la génération précédente.*

Pour conclure, n'oublions pas que ces lois d'évolution sont basées sur des hypothèses simplifiées ne tenant pas compte de facteurs importants tels, par exemple, les capacités parasites liées au connections métalliques entre transistors ou les courants de fuite ou de court-circuits des transistors. Cela dit le formules obtenues dans ce chapitre donnent une première approche théorique à qui s'intéresse aux évolutions à venir de l'industrie du semi-conducteur.

# Deuxième partie

# TDs

# Chapitre 9

# TD - Fonctions de base

Ce TD traite de la logique combinatoire et comprend les exercices suivants :

1. 9.1 : Simplification algébrique d'équations.
2. 9.2 : Simplification d'équations par tableau de Karnaugh.
3. 9.5 : Décodage.
4. 9.6 : Génération de fonctions.

## 9.1 Simplification algébrique

On considère qu'une équation booléenne est simplifiée si le nombre d'apparition des variables dans l'équation est le plus petit possible.

1. En utilisant les propriétés et théorèmes de l'algèbre de Boole, simplifiez l'expression : $S = (a + b + c) \cdot (\bar{a} + \bar{d} \cdot e + f) + \overline{(\bar{d} + e)} \cdot \bar{a} \cdot c + a \cdot b$

## 9.2 Simplification par tableau de Karnaugh

La méthode de Karnaugh permet de simplifier les fonctions logiques ayant peu de variables, à partir de la table de vérité de la fonction.

1. Simplifiez les deux fonctions $F$ et $G$ suivantes (cf. Tab. 9.1 et 9.2) après avoir transformé leur table de vérité en tableau de Karnaugh.

## 9.3 Fonction $F$

- Les entrées sont $a$, $b$, $c$, $d$, $e$.
- Une variable d'entrée à « X » indique qu'elle peut être à 0 ou à 1.

## 9.4 Fonction $G$

- Les entrées sont $a$, $b$, $c$ et $d$,
- $i$ indique la valeur de la combinaison (ou minterme) en notation décimale,
- le « $-$ » indique que $G$ peut prendre indifféremment la valeur 0 ou 1.

| $e$ | $d$ | $c$ | $b$ | $a$ | $F$ |
|---|---|---|---|---|---|
| 0 | $X$ | $X$ | $X$ | $X$ | 0 |
| 1 | 0 | 0 | $X$ | $X$ | 0 |
| 1 | 0 | 1 | 0 | $X$ | 1 |
| 1 | 0 | 1 | 1 | $X$ | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | $X$ | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | $X$ | 0 |

**Table 9.1:** *Table de vérité de la fonction $F$.*

| $i$ | $d$ | $c$ | $b$ | $a$ | $G$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | – |
| 3 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | – |
| 5 | 0 | 1 | 0 | 1 | – |
| 6 | 0 | 1 | 1 | 0 | – |
| 7 | 0 | 1 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | – |
| 9 | 1 | 0 | 0 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | 1 |
| 11 | 1 | 0 | 1 | 1 | 0 |
| 12 | 1 | 1 | 0 | 0 | 0 |
| 13 | 1 | 1 | 0 | 1 | 1 |
| 14 | 1 | 1 | 1 | 0 | 0 |
| 15 | 1 | 1 | 1 | 1 | 1 |

**Table 9.2:** *Table de vérité de la fonction $G$.*

## 9.5 Décodage

Le décodeur est un circuit combinatoire à l'entrée duquel est appliqué un code binaire de $n$ bits. Ce circuit possède $N$ sorties (avec $N = 2^n$, en général). A chaque valeur du code d'entrée, il y a une seule sortie à l'état haut, toutes les autres sont à l'état bas. Les entrées d'un

décodeur sont souvent appelées *adresses*, car elles expriment en binaire le numéro décimal de la sortie activée. Les décodeurs peuvent être utilisés pour l'adressage de mémoires et la génération de fonctions logiques.

**Décodeur BCD**
Le BCD (« binary coded decimal ») est un code de 4 bits dont seules les 10 premières combinaisons de 0 à 9 sont employées. Les combinaisons restantes de 10 à 15 ne sont jamais utilisées. Un décodeur BCD est donc un décodeur qui a 4 entrées et 10 sorties.

1. Réalisez ce décodeur en considérant que si l'une des 6 combinaisons non autorisées est à l'entrée, toutes les sorties sont à l'état inactif « 0 ».

**Décodeur de grande capacité**

Si le nombre $N$ est très élevé, on peut imaginer réaliser le décodage en cascadant des décodeurs de tailles moins importantes.

1. Par exemple essayez de concevoir un décodeur binaire 5 entrées / 32 sorties à partir de 2 décodeurs binaires 4 entrées / 16 sorties.
2. Quelle doit être la modification à apporter au décodeur 4 entrées / 16 sorties pour créer facilement le décodeur 5 entrées ?
3. Concevez un décodeur binaire 8 entrées / 256 sorties en utilisant le décodeur 4 entrées / 16 sorties précédemment modifié.

## 9.6 Génération de fonctions

Un transcodeur ou convertisseur est un circuit combinatoire à $x$ entrées et $y$ sorties. A chaque code d'entrée de $x$ bits correspond un code de sortie $y$ bits. Les décodeurs que nous avons étudiés dans l'exercice précédent sont donc des cas particuliers de transcodeurs.

On désire réaliser la fonction de transcodage d'un code BCD vers un code « 2 parmi 5 ». Dans le code « 2 parmi 5 », il y a toujours deux bits à « 1 » et 3 bits à « 0 ». La table de vérité est indiquée ci-dessous dans la Tab. 9.3.

- $i$ indique la valeur de la combinaison (ou minterme) en décimal,
- les entrées sont $a$, $b$, $c$, $d$, $e$,
- les sorties sont $F_4F_3F_2F_1F_0$,
- si $i > 9$ l'état des sorties est indifférent.

1. Réalisez la fonction à l'aide :
   (a) D'un décodeur BCD et quelques portes.
   (b) De multiplexeurs.

   Pour cela écrivez l'équation logique d'un multiplexeur 16 entrées (et donc 4 entrées de sélection) et comparez à l'expression d'une fonction logique quelconque à 4 entrées.

| $i$ | $d$ | $c$ | $b$ | $a$ | $F_4$ | $F_3$ | $F_2$ | $F_1$ | $F_0$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

**Table 9.3:** *Table de vérité de la fonction de conversion BCD* $\rightarrow$ « $2$ *parmi* $5$ ».

# Chapitre 10

# TD - Opérateurs arithmétiques

Le but de ce TD est d'approfondir la représentation en complément à 2 (Exercice 10.1), ainsi que l'arithmétique des nombres binaires (Exercice 10.2 sur l'addition, 10.3 sur la soustraction et comparaison, 10.4 sur la multiplication).

## 10.1 Représentation en complément à 2

1. Donner la représentation en CA2 des nombres suivants : -8, +8, -30, -52, +15.
2. Soit $B$ un nombre codé en CA2 sur $n$ bits : $(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$. Comment obtient-on la valeur de $B$ à partir de sa représentation lorsqu'il est positif ? lorsqu'il est négatif ?
3. Donner la représentation en CA2 des nombres +15, -12 :
   - sur 5 bits,
   - sur 7 bits.
4. D'une façon générale, comment peut-on étendre la représentation d'un nombre codé en CA2 sur $n$ bits, à une représentation sur $p$ bits, avec $p$ plus grand que $n$ ?

## 10.2 Addition en complément à 2

1. Quel est l'intervalle de variation d'un nombre codé en CA2 sur $n$ bits ?
2. Soient $A$ et $B$ deux nombres codés en CA2 sur $n$ bits et $S$ la somme de ces 2 nombres. Quel est l'intervalle de variation de $S$ ? En déduire le nombre de bits nécessaires à son codage.
3. Réaliser en binaire les additions suivantes : 30 + 8, 30 + (-8), (-30) + 8, (-30) + (-8).

## 10.3 Soustraction et comparaison

On désire réaliser un opérateur capable d'effectuer la comparaison de 2 nombres positifs $A$ et $B$ codés sur 4 bits. La sortie $S$ de l'opérateur vaut « 1 » si $A$ est strictement inférieur à $B$, « 0 » sinon :

- $S = 1$ si $A < B$,
- $S = 0$ si $A > B$ ou $A = B$.

1. Proposer une solution à l'aide d'un soustracteur.

2. Une autre solution appelée « comparaison MSB en tête » consiste à comparer bit à bit les nombres $A$ et $B$ en commençant par les bits de poids forts. L'algorithme utilisé est le suivant :

$$\begin{aligned} S = 1 \quad & \text{SI} \quad (a_3 < b_3) \\ & \text{OU} \quad ((a_3 = b_3) \text{ ET } (a_2 < b_2)) \\ & \text{OU} \quad ((a_3 = b_3) \text{ ET } (a_2 = b_2) \text{ ET } (a_1 < b_1)) \\ & \text{OU} \quad ((a_3 = b_3) \text{ ET } (a_2 = b_2) \text{ ET } (a_1 = b_1) \text{ ET } (a_0 < b_0)) \ . \end{aligned}$$

- Construire l'opérateur élémentaire à 2 entrées $a_i$ et $b_i$ dont les sorties $I_i$ (Inférieur) et $E_i$ (Égal) vérifient :

$$\begin{aligned} I_i &= 1 \text{ si } a_i < b_i, \\ I_i &= 0 \text{ sinon.} \end{aligned}$$

$$\begin{aligned} E_i &= 1 \text{ si } a_i = b_i, \\ E_i &= 0 \text{ sinon.} \end{aligned}$$

- En utilisant l'opérateur construit précédemment, proposer le schéma complet du comparateur.
- Comment peut-on généraliser simplement ce comparateur à $n$ bits ?

## 10.4 Multiplication

1. Réaliser à la main l'opération : $1001 \times 1100$ ($9 \times 12$).
2. Proposer le schéma d'un multiplieur de 2 nombres positifs de 4 bits. On dispose pour cela d'additionneurs 4 bits.
3. Comment faut-il modifier ce schéma pour permettre la multiplication de 2 nombres en complément à 2 ?

# Chapitre 11

# TD - Utilisation des bascules

Le TD comprend des exercices portant sur la mise en oeuvre de bascules dans des applications courantes de l'électronique :

- 11.1 Mise en pipeline d'une fonction combinatoire
- 11.2 Changement de format série ↔ Parallèle
- 11.3 Calcul de parité.

## 11.1 Mise en pipeline d'une fonction combinatoire

La fonction électronique à étudier est illustrée par la figure 11.1. Il s'agit de traiter un flot continu de données arrivant à un certain rythme. La fréquence d'arrivée des données, et donc de traitement, doit être la plus grande possible.

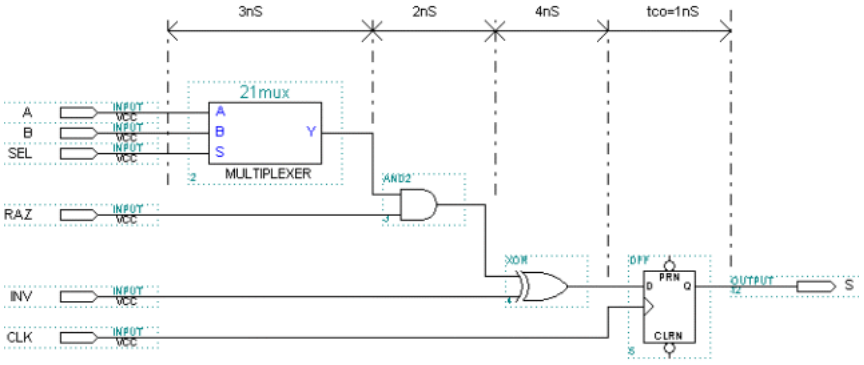



**Figure 11.1:** *Circuit à étudier*

### 11.1.1 Analyse de la fonction

1. Quelles sont les rôles respectifs des entrées SEL, RAZ et INV ?
2. On considère que toutes les entrées sont issues de bascules ayant un temps de propagation $t_{co}$ de 1 ns et que les temps de prépositionnement $t_{su}$ et $t_{ho}$ des bascules sont négligeables.
   Quelle est la fréquence d'échantillonnage maximum $f_{max}$ de la fonction ?

### 11.1.2 Augmentation de la fréquence de fonctionnement avec un étage de pipeline

Une barrière de registre est rajoutée comme indiqué dans la figure 11.2. Le nouvel étage de pipeline ainsi généré permet d'augmenter la fréquence de fonctionnement.

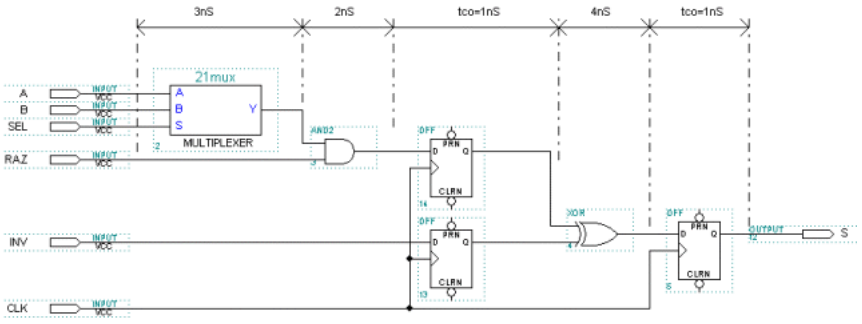



**Figure 11.2:** *Circuit à étudier avec pipeline*

1. Quelle est la nouvelle fréquence maximum $f_{max}$ de fonctionnement ?
2. Quels sont le retard et la latence du signal de sortie ?

### 11.1.3 Optimisation en performances

1. Rajoutez un étage de pipeline permettant d'obtenir la meilleure fréquence possible.
2. Quelle est la nouvelle valeur de la fréquence $f_{max}$ ?

### 11.1.4 Compromis performances/surface

La table 11.1 indique la surface des portes logiques. L'unité est la surface de la porte ET.

| ET | MUX | XOR | DFF |
|---|---|---|---|
| 1 | 2 | 2 | 6 |

**Table 11.1:** *Surface des éléments*

1. Calculez la surface pour les 3 cas étudiés ci-dessus. Analysez le rapport entre surface et fréquence maximale de fonctionnement.
2. Quelle conclusion peut-on en tirer ?

## 11.2 Changement de format série $\leftrightarrow$ Parallèle

Cet exercice a pour but de concevoir une fonction de changement de format d'une donnée arrivant en série et sortant en parallèle, et vice-versa. La structure de registre à décalage sera utilisée à cette fin.

### 11.2.1 Conversion série → parallèle

1. Concevez le composant D-EN dont le symbole est représenté par la figure 11.3 et ayant les spécifications indiquées dans la table 11.2.

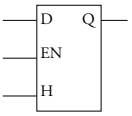



**Figure 11.3:** *Bascule D-EN*

| | |
|---|---|
| EN=0 | Gel de la sortie Q |
| EN=1 | bascule D avec entrées sur D et sortie sur Q |

**Table 11.2:** *Spécifications de D-EN*

2. A l'aide du composant D-EN, concevez un convertisseur série=>parallèle SER-PAR dont l'entrée D arrive sur 1 bit au rythme de l'horloge H et les sortie S sortent sur 4 bits. Le symbole est représenté par la figure 11.4 et les spécifications sont indiquées dans la table 11.3.

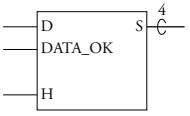



**Figure 11.4:** *Composant SER-PAR*

| | |
|---|---|
| DATA_OK=0 | Gel de la sortie S |
| DATA_OK=1 | La sortie S prend les 4 derniers bits de D |

**Table 11.3:** *Spécifications de SER-PAR*

### 11.2.2 Conversion parallèle → série

1. Concevez le composant D-EN-LD dont le symbole est représenté par la figure 11.5 et ayant les spécifications indiquées dans la table 11.4.

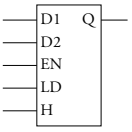



**Figure 11.5:** *Bascule D-EN-LD*

2. A l'aide du composant D-EN-LD, concevez un convertisseur parallèle=>série PAR-SER avec les entrées sur 4 bits et une sortie sur 1 bit changeant au rythme de l'horloge H. Le symbole est représenté par la figure 11.6 et les spécifications indiquées dans la table 11.5.

| EN=0 | gel de la sortie Q |
|---|---|
| EN=1 et LD =0 | bascule D avec entrée sur D1 et sortie sur Q |
| EN=1 et LD =1 | bascule D avec entrée sur D2 et sortie sur Q |

**Table 11.4:** *Spécifications de D-EN-LD*

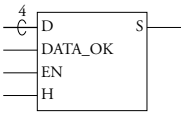



**Figure 11.6:** *Composant PAR-SER*

| EN=0 | Gel de la sortie S |
|---|---|
| EN=1 et DATA_OK=0 | la sortie S sort d'une façon cyclique les 4 bits de D enregistrées |
| EN=1 et DATA_OK=1 | Les entrées D sont enregistrées |

**Table 11.5:** *Spécifications de PAR-SER*

## 11.3 Calcul de parité.

La parité d'un mot de $n$ bits est vraie si le nombre de bits à 1 est pair. On se propose de réaliser ce calcul sur une donnée de 8 bits.

### 11.3.1 Calcul de parité sur un mot parallèle

1. Concevez un circuit calculant la parité d'un mot de 8 bits en parallèle, à partir exclusivement de portes XOR et ayant un temps de propagation minimal.

### 11.3.2 Calcul de parité sur un mot série

Les bits du mot arrivent maintenant en série et d'une manière synchrone avec une horloge CLK. Le signal DEB est actif juste avant le premier bit. Le chronogramme de la donnée et de DEB est donné en figure 11.7.

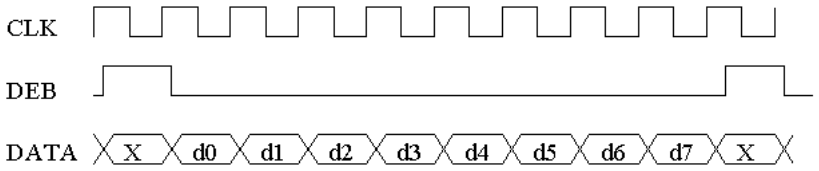



**Figure 11.7:** *Chronogramme des entrées du calculateur*

Comme les bits arrivent séquentiellement, on peut utiliser une structure simple calculant la parité bit après bit et stockant le résultat dans une bascule D. La bascule doit être initialisée à l'aide du signal DEB et contient le résultat au bout des 8 bits.

1. Donnez l'équation de l'entrée D de la bascule par rapport à DATA, DEB et la sortie de la bascule.

2. Déduisez la structure du circuit séquentiel correspondant.

# Chapitre 12

# TD - Synthèse et utilisation des machines à états synchrones

Ce TD comprend 2 exercices, permettant de reprendre et de mettre en oeuvre les notions abordées dans la leçon sur les machines à états 5. Nous traiterons deux problèmes :

- Étude et conception d'un contrôleur de bus simple.
- Prise en compte du problème de l'équité.

Remarque : Vous pouvez, si vous le souhaitez, réaliser et intégrer les contrôleurs étudiés lors de ce TD sur des circuits logiques programmables dans les salles de TP du département (A406-7).

## 12.1 Qu'est-ce qu'un bus de communication ?

Lorsque, au sein d'un système complexe, plusieurs dispositifs électroniques doivent communiquer entre eux on peut imaginer de relier chaque élément à tous les autres. Cette situation, illustrée par la figure 12.1, est probablement la première qui vient à l'esprit. C'est aussi la plus puissante car elle permet un nombre très important de communications simultanées.

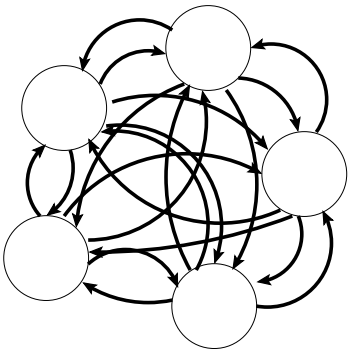


**Figure 12.1:** *Liaisons point à point*

Malheureusement elle est aussi très coûteuse car le nombre de connexions nécessaires est très important. Il suffit d'imaginer pour s'en convaincre que les arcs du schéma ci-dessus véhiculent des informations codées sur 32 bits. En outre elle n'offre pas une grande flexibilité car il n'est pas possible d'ajouter des éléments à notre réseau (le nombre d'entrées et de sorties de

chaque élément est fixé à la construction). Ce système n'est pas très *plug and play*. C'est dommage car le *plug and play* est justement très à la mode. Une autre solution, plus raisonnable et aussi plus courante, est le bus central  comme illustré dans la figure 12.2

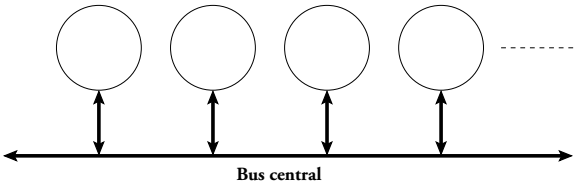



**Figure 12.2:** *Bus central*

Les possibilités d'échanges sont limitées mais chaque élément peut tout de même communiquer avec n'importe quel autre et le nombre de connexions est considérablement réduit. Il est en outre théoriquement possible d'ajouter à l'infini de nouveaux éléments au système. La gestion d'une telle organisation des communications nous servira de thème tout au long de ce TD.

## 12.2 Le contrôleur de bus simple.

Nous nous proposons de concevoir un contrôleur de bus de communication. Le système au sein duquel notre contrôleur doit s'intégrer comporte un arbitre de bus et un nombre indéterminé mais potentiellement très grand de points d'accès au bus. Chaque point d'accès est composé d'un contrôleur et d'un client. La figure 12.3 représente le système de communication complet :

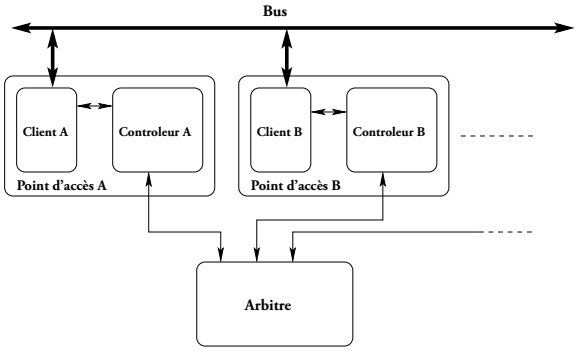



**Figure 12.3:** *Système de communication*

L'arbitre est chargé de répartir la ressource de communication (le bus) entre les différents points d'accès. En effet, le système n'admet pas que plusieurs points d'accès émettent simultanément des informations sur le bus. Si cela se produisait il y aurait conflit et perte d'informations. La présence d'un arbitre est donc nécessaire. C'est lui qui autorise successivement les points d'accès à écrire sur le bus en leur attribuant un "jeton". Le point d'accès possesseur du jeton peut écrire sur le bus. Les autres ne peuvent que lire. Lorsque le point d'accès a terminé sa transaction il rend le jeton à l'arbitre qui peut alors l'attribuer à un autre point d'accès. L'absence de conflit est garantie par l'unicité du jeton.

Les clients sont les utilisateurs du bus. Lorsqu'un client désire écrire sur le bus il en informe son contrôleur associé et attend que celui-ci obtienne le jeton et lui donne le feu vert.

Les contrôleurs servent d'interface entre l'arbitre et leur client. C'est l'un de ces contrôleurs que nous allons concevoir. Ses entrées - sorties sont décrites dans le schéma illustré en figure 12.4 et la table 12.1. A l'exception de l'horloge et du signal de *reset* toutes les entrées - sorties sont actives à '1'.

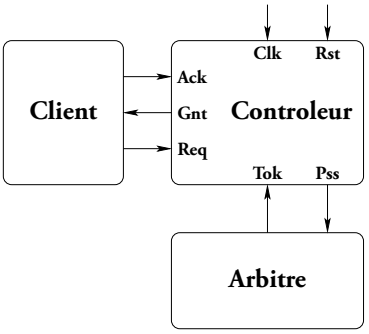



**Figure 12.4:** *Contrôleur de communication*

### 12.2.1 Le graphe d'états.

1. Dessinez un chronogramme représentant une ou plusieurs transactions complètes entre un contrôleur, son client et l'arbitre.
2. Le contrôleur est une machine à états de Moore. Imaginez et dessinez son graphe.
3. Vérifiez la cohérence du graphe en appliquant les méthodes du chapitre 2 de la leçon 6.
4. Vérifiez que les spécifications du contrôleur sont respectées par votre graphe.

### 12.2.2 Une optimisation possible.

Les échanges entre l'arbitre et le contrôleur (signaux TOK et PSS) présentent l'inconvénient de ralentir inutilement les opérations et donc de gaspiller des cycles d'utilisation du bus. En effet, un cycle est perdu lorsqu'un contrôleur se voit proposer le jeton alors qu'il n'en a pas l'usage. Le chronogramme de la figure illustre 12.5 ce phénomène :

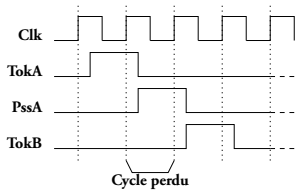



**Figure 12.5:** *Illustration de la perte d'un cycle*

TOKA et TOKB sont les signaux TOK destinés à deux contrôleurs, A et B. PSSA est le signal PSS émis par le contrôleur A et indiquant qu'il rend le jeton que l'arbitre vient de lui confier et dont il n'a pas l'usage. On voit que l'arbitre, lui aussi synchrone sur front montant de CLK, ne peut pas proposer immédiatement le jeton à un autre contrôleur.

| Nom | Direction | Description |
|---|---|---|
| CLK | Entrée | Horloge pour la synchronisation du contrôleur |
| RST | Entrée | Signal de reset asynchrone, actif à '0'. Lorsque ce signal est à état bas ('0') le contrôleur est entièrement réinitialisé. |
| TOK | Entrée | Ce signal provient de l'arbitre et indique que le contrôleur peut disposer du bus. Il signifie donc que l'arbitre offre le jeton au contrôleur. Il n'est actif que pendant une période d'horloge. Si le contrôleur n'a pas besoin du jeton il le rend (voir le signal PSS). Sinon il le garde jusqu'à ce qu'il n'en ait plus l'utilité. |
| REQ | Entrée | Ce signal est émis par le client et indique que ce dernier demande à disposer du bus. Le client maintient ce signal jusqu'à la fin de sa transaction sur le bus. Il ne le relache que lorsqu'il n'a plus besoin du bus. |
| ACK | Entrée | Ce signal provient du client et indique que le client a pris le bus et commence sa transaction. Il n'est actif que pendant une période d'horloge. |
| PSS | Sortie | Ce signal est destiné à l'arbitre et l'informe que le contrôleur rend le bus, soit parce que l'arbitre le lui a proposé alors qu'il n'en a pas besoin, soit parce que la transaction du client est terminée. Il signifie donc que le contrôleur rend le jeton à l'arbitre qui pourra ensuite en disposer et l'attribuer à un autre contrôleur, voire au même. Il n'est actif que pendant une période d'horloge. |
| GNT | Sortie | Ce signal est destiné au client et l'informe qu'il peut disposer du bus. Il est maintenu tant que le client n'a pas répondu (par le signal ACK) qu'il a pris le bus. |

**Table 12.1:** *Spécification du contrôleur*

Pour améliorer les performances du système nous voudrions obtenir le chronogramme illustré en figure 12.6 :

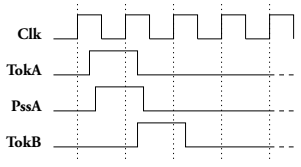



**Figure 12.6:** *Chronogramme optimisé*

1. Proposez des modifications du contrôleur permettant d'obtenir ce nouveau comportement.
2. Discutez leurs mérites respectifs.
3. Le contrôleur est-il toujours une machine à états ? Pourquoi ?

### 12.2.3 Réalisation.

1. Décrivez sous forme de schéma la structure du contrôleur optimisé. Vous ne détaillerez pas les parties combinatoires.
2. Décrivez, sous forme fonctionnelle symbolique, le comportement des parties combinatoires. Un exemple de description fonctionnelle symbolique est illustré en figure 12.7 :

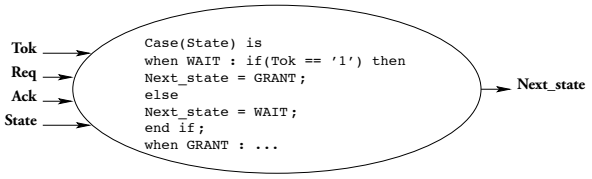



**Figure 12.7:** *Description fonctionnelle symbolique*

## 12.3 Le problème de l'équité.

### 12.3.1 Le contrôleur équitable.

Le contrôleur que nous venons de concevoir n'est pas entièrement satisfaisant car il n'est pas équitable. En d'autres termes, il ne garantit pas qu'un client n'accaparera pas le bus au détriment des autres. Il ne garantit même pas qu'un client, après avoir obtenu l'accès au bus, l'utilisera effectivement puis le relâchera. Il est en effet possible qu'un client ne réponde jamais au signal GNT de son contrôleur (ce qu'il est sensé faire à l'aide du signal ACK). Le système complet serait alors bloqué par un "mauvais" client qui monopolise une ressource dont il n'a pas l'usage. Pour remédier à cet inconvénient il faut à nouveau modifier le contrôleur.

1. Imaginez des solutions afin de rendre équitable le contrôleur optimisé du premier exercice.
2. Décrivez, sans entrer dans les détails, la structure de ce nouveau contrôleur. Vous séparerez les registres et les parties combinatoires. Vous donnerez une description fonctionnelle symbolique des parties combinatoires et vous expliciterez le comportement des registres.

### 12.3.2 L'arbitre équitable.

1. Pour obtenir que l'ensemble du système soit équitable, la modification du contrôleur seul ne suffit pas. L'arbitre doit, lui aussi, adopter un comportement particulier. Pourquoi ? Donnez un exemple de comportement non équitable possible de l'arbitre et ses conséquences.
2. Imaginez et décrivez des comportements possibles de l'arbitre équitable.
3. Comme précédemment, décrivez la structure de l'arbitre équitable.

# Chapitre 13

# TD - Analyse et synthèse en portes logiques

## 13.1 Introduction

Le TD comprend 4 exercices portant sur l'analyse et la synthèse de portes logiques à partir de transistors CMOS

Exercices

1. 13.2 Analyse d'une porte complexe
2. 13.3 Analyse de portes présentant des dysfonctionnements
3. 13.4 Synthèse de la fonction majorité
4. 13.5 Synthèse d'un additionneur 1 bit

## 13.2 Analyse d'une porte complexe

La figure 13.1 représente l'implantation (la réalisation) d'une fonction logique en utilisant les principes de la logique complémentaire CMOS (réseau de tirage à "1" à base de transistors PMOS et réseau de tirage à "0" à base de transistors NMOS).

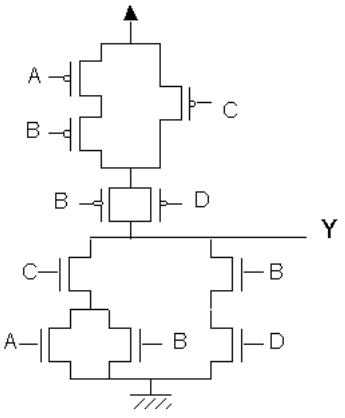



**Figure 13.1:** *Porte logique*

### 13.2.1 Analyse de la fonction à l'aide du réseau de transistors P

1. En considérant que Y est à 1 uniquement si le réseau P est passant, donnez l'expression de la fonction logique Y en utilisant la structure du réseau de transistors P.

### 13.2.2 Analyse de la fonction à l'aide du réseau de transistors N

1. En considérant que Y est à O uniquement si le réseau N est passant, donnez l'expression de la fonction logique Y en utilisant la structure du réseau de transistors N.
2. Vérifiez que les fonctions obtenues à l'aide du réseau P puis du réseau N sont bien identiques.

### 13.2.3 L'implémentation est elle unique ?

1. Dressez le tableau de Karnaugh de la fonction Y. En simplifiant la fonction trouvez une alternative au réseau de transistors P.

## 13.3 Analyse de portes présentant des dysfonctionnements

Une seule de ces portes réalise correctement une fonction logique.

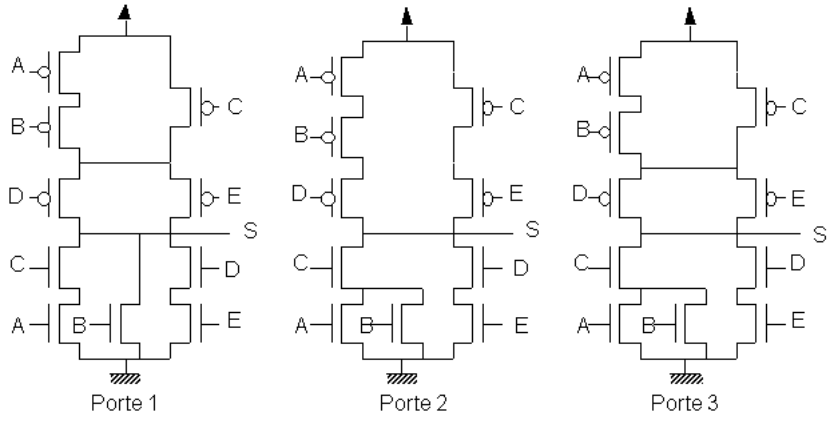



**Figure 13.2:** *Trois portes...*

### 13.3.1 Quelle est la "bonne" ?

1. En examinant la dualité des réseaux N et P (une structure de transistors série P correspond à une structure parallèle N et vice versa), déduisez la bonne porte.
2. Quelle est l'expression de sa fonction logique ?

### 13.3.2 Causes de dysfonctionnements

1. Trouvez des combinaisons des entrées introduisant un problème pour les 2 portes défectueuses.

## 13.4 Synthèse de la fonction Majorité

Soit le complément de la fonction Majorité : $\overline{Maj}(A, B, C) = \overline{A \cdot B + B \cdot C + A \cdot C}$

### 13.4.1 Construction CMOS de la fonction Majorité complémentée

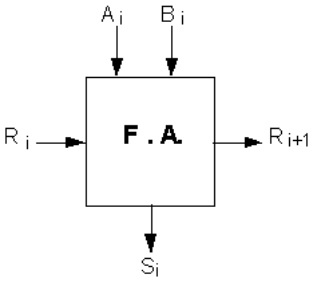
1. En utilisant la dualité des réseaux P et N, établissez le schéma CMOS de la fonction Majorité complémentée. Trouvez une structure minimisant le nombre de transistors.

### 13.4.2 Optimisation de la fonction Majorité complémentée

1. Démontrer la relation suivante :$\overline{Maj}(A, B, C) = Maj(\overline{A}, \overline{B}, \overline{C})$
2. Déduisez une structure présentant une symétrie sur les réseaux P et N. Quels sont les intérêts d'une telle structure ?

## 13.5 Synthèse d'un Additionneur 1 bit

Cet exercice traite de l'additionneur 1 bit qui est la cellule de base de l'additionneur à propagation de retenue.




| A i | B i | R i | R i+1 | S i |
|---|---|---|---|---|
| 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 |

**Figure 13.3:** *Table de vérité de l'Additionneur Complet 1 bit*

### 13.5.1 Construction de la retenue Ri+1 en CMOS

1. Proposez une structure de porte CMOS pour réaliser la fonction Ri+1

### 13.5.2 Construction de la sortie Si en CMOS

1. Vérifier que Si peut s'exprimer sous la forme : $S_i = A_i B_i R_i + \overline{R_{i+1}}\,(A_i + B_i + R_i) = A_i \oplus B_i \oplus R_i$
2. En utilisant le même raisonnement que pour la fonction Majorité (exercice 13.4.2) trouvez une structure optimale de la fonction $S_i$

### 13.5.3 Évaluation de l'aire de la surface d'un additionneur

1. Sachant qu'on utilise une technologie CMOS de densité $1\ 500\ 000 tr \cdot mm^{-2}$, combien pouvons nous intégrer d'additionneurs 32 bits dans $1 cm^2$ ?

# Chapitre 14

# TD - Performances de la logique complémentaire CMOS

## 14.1 Objectifs du TD

Ce TD est l'occasion d'une première mise en pratique des concepts de *temps de propagation* de portes logiques. Les différents exercices visent à mettre en évidence :

- l'usage de bibliothèques précaractérisées ;
- le lien entre la structure des portes CMOS et leurs performances ;
- le lien entre l'optimisation de fonctions booléennes et leurs performances.

## 14.2 Temps de propagation d'une fonction décodeur

Nous désirons réaliser la fonction $LM_{20}$ dont l'équation logique est la suivante :

$$LM_{20} = \overline{A_0} \cdot \overline{A_1} \cdot A_2 \cdot \overline{A_3} \cdot A_4 \cdot \overline{A_5}$$

Les effets parasites des fonctions logiques connectées en aval de la fonction $LM_{20}$ sont modélisés par une capacité d'utilisation $C_u$ connectée en sortie de la fonction. La valeur de $C_u$n'est pas connue. Un concepteur, utilisant les cellules d'une bibliothèque précaractérisée donnée en annexe 14.5, nous propose trois implantations illustrées dans la figure 14.1 (on suppose disponibles les entrées et leurs complémentaires) :

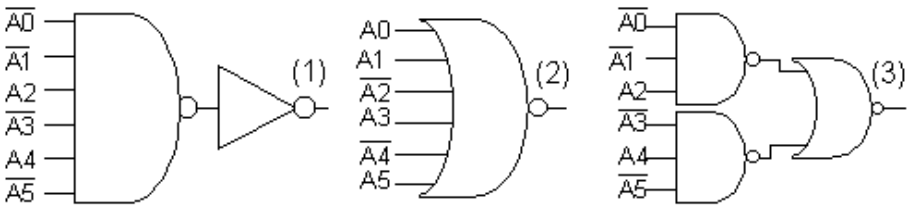



**Figure 14.1:** *Trois implantations alternatives de la fonction* $LM_{20}$

- *Question 1 :* Vérifier que les schémas de la figure 14.1 représentent tous la fonction $LM_{20}$ et comparer ces solutions en terme de nombre de transistors utilisés.

- *Question 2 :* En supposant que la sortie de la fonction $LM_{20}$ charge une capacité d'utilisation égale à un multiple entier de la capacité de référence ($C_u = N \times C_r$) déterminer pour chacune des solutions l'expression du temps de propagation de la fonction $LM_{20}$ en fonction de N et de $t_{pr}$.
- *Question 3 :* Déterminer en fonction de N la solution donnant le temps de propagation le plus faible. Pouvait-on prévoir ces résultats ? (expliquer).

## 14.3 Amélioration du décodeur par amplification logique

Nous avons maintenant fixé la valeur de la capacité d'utilisation $C_u$ de la fonction $LM_{20}$ à $C_u = 175 \times C_r$. Nous décidons de reprendre la solution (3) modifiée suivant le schéma de la figure 14.2, où $INV_1$ et $INV_2$ sont deux inverseurs de la bibliothèque.

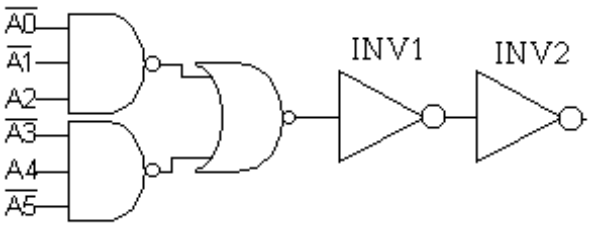



**Figure 14.2:** *Solution (3) avec amplification logique*

- *Question 4 :* Déterminer le temps de propagation de la fonction dans cette nouvelle configuration, et comparer de nouveau à la solution (1).

Nous décidons de nous affranchir des contraintes de la bibliothèque et décidons de remplacer l'inverseur $INV_2$ par un inverseur dont nous définissons nous même les dimensions. Pour cela nous multiplions les largeurs $W$ des deux transistors NMOS et PMOS de l'inverseur de la bibliothèque par un même coefficient $\alpha$.

- *Question 5 :* Déterminer, en fonction des caractéristiques de l'inverseur de la bibliothèque et du coefficient $\alpha$, la valeur des paramètres $C_{eINV}$, $t_{p0INV}$ et $d_{tpINV}$ de l'inverseur $INV_2$.
- *Question 6 :* Déterminer de nouveau le temps de propagation de la fonction $LM_{20}$ en fonction du coefficient $\alpha$.
- *Question 7 :* Montrer qu'il existe une valeur de $\alpha$ minimisant le temps de propagation de la fonction $LM_{20}$. Calculer cette valeur et déterminer le nouveau temps de propagation.

La valeur de $C_u$ est en fait la capacité résultante de 25 entrées de diverses portes connectées en sortie de la fonction $LM_{20}$.

- *Question 8 :* Imaginer une solution donnant le même résultat que précédemment mais évitant de créer une nouvelle cellule (nous ne sommes pas maitres de la bibliothèque...).

Dans la pratique, les concepteurs ne cherchent pas à minimiser les temps de propagation des fonctions combinatoires, mais plutôt à limiter ceux-ci à une valeur jugée acceptable pour le fonctionnement correct du circuit.

- *Question 9 :*Montrer que pour tout choix de temps de propagation supérieur au temps minimum, il existe une valeur de $\alpha$ minimisant l'aire du circuit réalisant la fonction $LM_{20}$.

## 14.4 Généralisation du principe de l'amplification logique

Le problème précédent peut être généralisé de la façon suivante : Considérant une capacité $C_u$ devant être chargée (ou déchargée) par de la logique CMOS, existe-t-il une structure de chaîne d'inverseurs optimale minimisant le temps de propagation ?.

La chaîne totale considérée (voir figure 14.3) sera composée de N inverseurs $INV_i$ dont les largeurs des transistors sont multipliées par des coefficients $\alpha_i$ par rapport à l'inverseur de la bibliothèque. Le coefficient $\alpha_1$ est figé à la valeur 1.

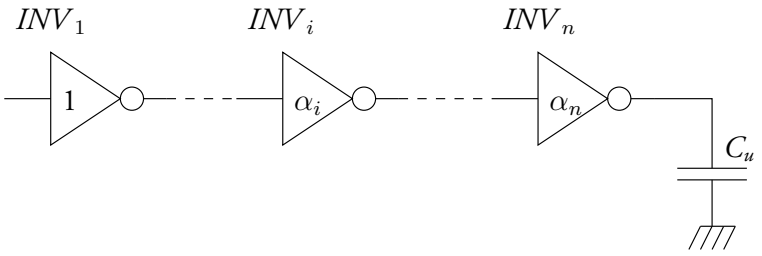



**Figure 14.3:** *Amplification logique généralisée.*

- *Question 10 :* Etablir l'expression du temps de propagation de la chaîne en fonction du nombre N d'inverseurs et des coefficients $\alpha_i$.
- *Question 11 :* Montrer que les coefficients $\alpha_i$ doivent suivre une progression géométrique pour minimiser le temps de propagation de la chaîne. En déduire une nouvelle expression du temps de propagation en fonction d'un coefficient unique $\alpha$ et de N.
- *Question 12 :*Montrer qu'il existe une valeur de $\alpha$ qui minimise le temps de propagation de la chaîne. Exprimer cette valeur en fonction de $C_u$ $C_e$ et N. En déduire une expression du temps de propagation minimal ne dépendant plus que de $C_u$, $C_{eINV}$ et N.
- *Question 13 :*Montrer qu'il existe une valeur de N qui minimise le temps de propagation de la chaîne. Exprimer cette valeur en fonction de $C_u$ et $C_e$. En déduire une expression du temps de propagation minimal ne dépendant plus que de $C_u$ et $C_{eINV}$.

## 14.5 Annexe : Bibliothèque de cellules précaractérisées

Les sociétés de fonderies de Silicium, les "fondeurs", qui produisent des circuits intégrés numériques, proposent à leurs clients, des bibliothèques de portes logiques dites *précaractérisées*. Les ingénieurs de ces sociétés développent, dessinent et simulent le comportement et les performances de chacune des portes logiques de la bibliothèque. Ils fournissent à leurs clients des tables de caractéristiques permettant à ces derniers de concevoir des circuits intégrés et *prédire leurs performances* sans avoir à explorer des niveaux de détail allant jusqu'au transistor. Le tableau 14.1 propose une telle bibliothèque.

Pour chaque cellule de la bibliothèque sont précisés :

- La capacité d'entrée $C_{ei}$ de chaque entrée $E_i$ de la cellule.
- Le temps de propagation à vide $t_{p0}$ ;
- La dépendance capacitive (pente) du temps de propagation $d_{tp}$.

| Fonction | Équation booléenne | $C_{ei}$ | $t_{p0}$ | $d_{tp}$ |
|---|---|---|---|---|
| Inverseur | $Y = \overline{A}$ | $C_{eA} = 7C_r$ | $6t_{pr}$ | $d_{tpr}$ |
| Nand à 3 entrées | $Y = \overline{ABC}$ | $\forall i \in \{A, B, C\} C_{ei} = 7C_r$ | $42t_{pr}$ | $3d_{tpr}$ |
| Nand à 6 entrées | $Y = \overline{ABCDEF}$ | $\forall i \in \{A \cdots F\} C_{ei} = 7C_r$ | $156t_{pr}$ | $6d_{tpr}$ |
| Nor à 2 entrées | $Y = \overline{A + B}$ | $\forall i \in \{A, B\} C_{ei} = 7C_r$ | $16t_{pr}$ | $2d_{tpr}$ |
| Nor à 6 entrées | $Y = \overline{A + B + C + D + E + F}$ | $\forall i \in \{A \cdots F\} C_{ei} = 7C_r$ | $96t_{pr}$ | $6d_{tpr}$ |

**Table 14.1:** *Une bibliothèque précaractérisée simple.*

Rappelons que le temps de propagation d'une porte chargée par une capacité $C_u$ est donné par la formule :

$$t_p = t_{p0} + d_{tp} \times C_u$$

Nous avons simplifié les jeux de paramètres en ne distinguant notamment pas les temps de propagation en montée et en descente. Pour faciliter les quelques calculs numériques de ce TD, tous les paramètres sont définis comme des multiples entiers des valeurs de référence suivantes :

- $C_r$ : capacité de référence
- $t_{pr}$ : temps de propagation de référence
- $d_{tpr}$ : dépendance capacitive de référence

Ces trois paramètres sont liés par la relation suivante :

$$t_{pr} = d_{tpr} \times C_r$$

# Index