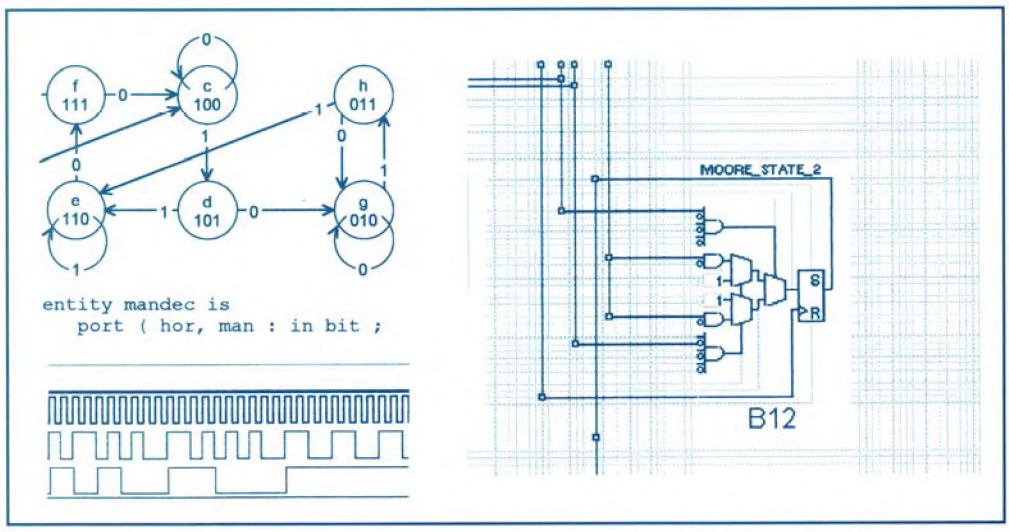
COLLECTION TECHNOLOGIES

circuits numériques et synthèse logique

un outil: VHDL

J. WEBER M. MEAUDRE



MASSON

COLLECTION TECHNOLOGIES

de l'Université à l'Industrie

circuits numériques et synthèse logique

un outil : VHDL

J. WEBER M. MEAUDRE

L'électronique numérique est présente dans tous les domaines de la vie courante et professionnelle. La conception et la réalisation des fonctions et circuits associés ont beaucoup évolué ces deux dernières décennies : la miniaturisation a permis une intégration plus grande et donc une complexification allant de pair. On trouve couramment plusieurs dizaines de milliers de portes logiques sur un seul circuit. Les outils de conception de ces éléments ont eux aussi subi une évolution remarquable. Leur informatisation permet actuellement la réalisation «à domicile» et surtout un gain de temps et une souplesse appréciables.

Cet ouvrage présente une méthode de conception de tels circuits, associés à un outil de description qui est devenu un standard : le langage VHDL. Les auteurs lient l'apprentissage des bases de l'électronique numérique à la compréhension des concepts des langages de description. Ce livre permettra ainsi au lecteur d'acquérir la maîtrise du processus de synthèse des circuits.

Destiné aux étudiants en électronique (IUT, BTS, seconds et troisièmes cycles universitaires, formations d'ingénieurs), ce livre sera aussi utile aux professionnels soucieux de mettre à jour leurs connaissances ou de découvrir un tel langage de description.

Jacques WEBER, docteur de 3e cycle, diplômé de l'École supérieure d'électricité, est maître de conférences à l'IUT de Cachan.

Maurice MEAUDRE, diplômé du CNAM, est chef de travaux à l'ENSAM, IUT de Cachan.

ISBN : 2-225-84956-0

9 782225 849565

Table des matières

Avant-propos

Au cours des quinze dernières années, les méthodes de conception des fonctions numériques ont subi une évolution importante. Dans les années soixante-dix, la majorité des applications de la logique câblée étaient construites autour de circuits intégrés standard, souvent pris dans la famille TTL. Au début des années quatre-vingt apparurent, parallèlement, les premiers circuits programmables par l'utilisateur du côté des circuits simples et les circuits intégrés spécifiques (ASICs) pour les fonctions complexes fabriquées en grande série. La complexité de ces derniers a nécessité la création d'outils logiciels de haut niveau qui sont à la description structurelle (schémas au niveau des portes élémentaires) ce que les langages évolués sont au langage machine dans le domaine de la programmation. A l'heure actuelle, l'écart de complexité entre circuits programmables et ASICs s'est restreint : on trouve une gamme continue de circuits qui vont des héritiers des premiers PALs (programmable array logic), équivalents de quelques centaines de portes, à des FPGAs (Field programmable gate array) ou des LCAs (Logic cell array) de quelques dizaines de milliers de portes équivalentes. Les outils d'aide à la conception se sont unifiés ; un même langage, VHDL par exemple, peut être employé quels que soient les circuits utilisés, des PALs aux ASICs.

Circuits numériques et synthèse logique, un outil : VHDL est l'un des résultats de la réflexion faite à l'IUT de CACHAN sur l'évolution du contenu de l'enseignement de l'électronique numérique, pour proposer aux étudiants une formation en accord avec les méthodes de conceptions actuelles.

La gageure était d'éviter le piège du « cours VHDL », impensable à ce niveau de formation, compte tenu du volume horaire imparti (une centaine d'heures, travaux d'applications compris). Nous avons décidé de nous restreindre à un sous-ensemble cohérent de VHDL, qui soit strictement synthétisable. Ce choix étant fait, nous avons mené en synergie l'apprentissage des bases de la logique et celui des concepts des langages de description de haut niveau. Chaque élément nouveau, vu sous son aspect circuit, est immédiatement transcrit en VHDL, ce qui précise sa fonction, et permet à l'étudiant de se familiariser progressivement avec la syntaxe du langage.

L'objectif étant la maîtrise du processus de synthèse, jusqu'à la réalisation, nous avons utilisé un compilateur VHDL qui permet très simplement de générer les fichiers JEDEC utilisés pour la programmation des circuits. Expérimentée pour la première fois au cours de l'année scolaire 1994-95, cette approche un peu différente de l'électronique numérique semble avoir rencontré un accueil favorable de la part des étudiants. La création de machines d'états adaptées à un problème donné, ne semble plus constituer un obstacle pour eux.

Que soient remerciés ici les collègues de l'IUT, pour les nombreuse discussions que nous avons eues sur le sujet, et, surtout, les étudiants qui se sont lancés, parfois avec fougue, dans l'exploration de cette terre inconnue.

Le livre est subdivisé en cinq chapitres et une annexe :

I. Une introduction générale au monde du numérique. On y définit les notions de base telles que la représentation des nombres, les conventions logiques etc.
II. Un panorama des caractéristiques électriques, statiques et dynamiques, des circuits numériques et leur emploi pour déterminer les limites de fonctionnement d'une application. Ce chapitre n'est pas forcément traité en cours de façon chronologique linéaire ; certaines parties ne prennent sens qu'au vu des applications de la logique séquentielle (calcul d'une fréquence maximum de fonctionnement, par exemple).
III. La définition des opérateurs élémentaires, combinatoires et séquentiels. Le principe adopté est de ne pas attendre d'avoir vu toute la logique combinatoire pour aborder la logique séquentielle. Les bascules (synchrones principalement) doivent devenir des objets aussi familiers qu'une porte ET. C'est à l'occasion de la découverte des opérateurs élémentaires que les premiers rudiments du langage VHDL apparaissent. Les difficultés de ce langage (dont la principale concerne le passage combinatoire → séquentiel asynchrone → séquentiel synchrone) sont abordées sur des objets très simples, elles sont donc faciles à expliciter. Une fois ces difficultés surmontées, la puissance du langage récompense très largement l'utilisateur de son effort intellectuel.
IV. Les principales catégories de circuits, fonctions standard et circuits programmables. L'étude de cette partie, volontairement restreinte, ne peut se faire qu'en illustrant le cours de nombreuses analyses de notices techniques.
V. Les méthodes de synthèse qui sont l'aboutissement de cet enseignement. On y fait connaissance avec les machines d'états, les architectures de Moore et de Mealy, leur transcription en VHDL. Les méthodes de simplification sont vues très rapidement, essentiellement pour comprendre ce que fait un optimiseur, de façon à apprendre à le piloter.
VI. Une annexe qui résume et explicite les principales constructions qu'autorise le langage VHDL. Sont exclues, volontairement, toutes les constructions non synthétisables, qui servent exclusivement à la modélisation et à la simulation.

De nombreux exemples illustrent les principes abordés, et des exercices permettent au lecteur d'asseoir sa compréhension.

La connaissance d'un langage de programmation procédural, comme C ou Pascal, n'est pas indispensable, mais elle facilite la compréhension de certains passages. Quelques notions sur les composants, transistors et capacités, et sur les lois élémentaires des circuits électriques sont souhaitables. On notera cependant que les aspects structurels internes des circuits ne sont pas abordés, sauf quand ils sont incontournables (sorties non standard).

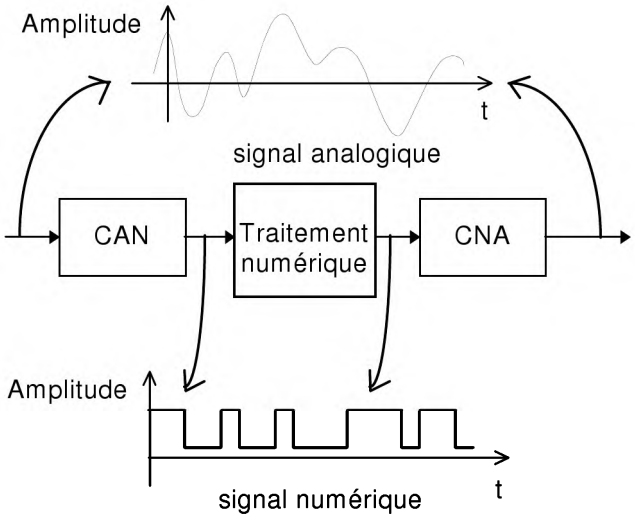
I Informations numériques

I.1. De l'analogique au numérique

Entre un disque « noir » et un disque « compact » il y a une différence de principe : le premier est *analogique*, le second *numérique*. Que signifient ces termes ?

- Le mot analogique évoque ressemblance, si on regarde avec un microscope une partie de sillon d'un disque noir on verra une sorte de vallée sinueuse dont les flancs reproduisent, à peu près, la forme des signaux électriques transmis aux haut-parleurs. A un son grave correspondent des sinuosités qui ont une période spatiale grande (quelques mm pour 100 Hz), à un son aigu correspondent des sinuosités dont la période spatiale est plus petite (quelques centièmes de mm pour 10 kHz). De même, l'amplitude des sinuosités reproduit, grosso-modo, l'amplitude du son que l'on souhaite reproduire.
- Le mot numérique évoque nombre. Si on regarde au microscope (grossissement supérieur à 100) une plage d'un disque compact on verra une sorte de pointillé de trous ovales, presque identiques, répartis de façon irrégulière sur des pistes quasi-circulaires. Aucun rapport de forme entre le son enregistré et l'allure de la gravure ne peut être observé, présence ou absence de trous constituent les deux valeurs possibles d'un chiffre en base 2. Ces chiffres, regroupés par paquets de 16, constituent des nombres entiers dont la valeur est l'image, via un code, de l'amplitude du signal sonore[1].

[1]En réalité le code est un peu plus compliqué que ne le laisserait supposer cette description, mais c'est une autre histoire.



Le passage d'un monde à l'autre se fait par des *convertisseurs analogique-numérique* (CAN) et *numérique-analogique* (CNA), dont nous n'étudierons pas ici le fonctionnement. La différence de principe évoquée plus haut se retrouve évidemment quand on observe le fonctionnement des circuits : un circuit analogique manipule des signaux électriques qui peuvent prendre une infinité de valeurs, qui sont en général des fonctions continues du temps, un circuit numérique manipule des signaux qui ne peuvent prendre qu'un nombre fini (généralement 2) de valeurs conventionnelles, sans rapport avec le contenu de l'information, qui sont des fonctions discontinues du temps.

I.2. Deux niveaux électriques : le bit

Dans toute la suite nous considérerons, ce qui est le cas le plus fréquent, que les signaux numériques représentent des valeurs binaires ; ils ne peuvent prendre que deux valeurs. Une variable binaire porte le nom de *bit*, contraction de binary digit, littéralement chiffre en base 2. Dans un circuit électronique la grandeur physique significative que l'on utilise le plus souvent est la tension (un signal électrique peut très bien être un courant), sauf précision contraire explicite la « valeur » d'un signal électrique binaire se mesurera donc en volts.

Dans un système numérique tous les potentiels sont mesurés par rapport à un potentiel de référence, la masse, qui est une équipotentielle commune à tous les circuits. Cette précision permet de parler du potentiel (ou de la tension) d'un point d'un montage au lieu de spécifier « différence de potentiels entre ... et la masse ». A chaque équipotentielle d'un circuit on peut donc associer un bit qui représente la valeur de la tension de l'équipotentielle considérée.

I.2.1 Conventions logiques

Une entrée ou une sortie d'un circuit numérique ne peut prendre que deux valeurs, notées généralement ***H***, pour High (haut), et ***L***, pour Low (bas) : 3 et 0,2 volts sont des valeurs typiques fréquemment rencontrées. La valeur d'un signal représente en général quelque chose :

- chiffre en base deux, 0 ou 1,
- valeur d'une variable logique, vrai ou faux,
- état d'un opérateur, actif ou inactif,
- état d'une porte, ouverte ou fermée,
- état d'un moteur, arrêté ou en marche,
- etc.

L'association entre la valeur électrique (H ou L) et le sens que l'on donne à cette valeur (0 ou 1, par exemple) constitue ce que l'on appelle une *convention logique*. Il n'y a pas de convention par défaut, cet oubli peut être une source d'erreurs. Dans la famille des microprocesseurs 68xx0 les adresses (des nombres entiers) sont matérialisées par des tensions où ***H*** est associé au 1 binaire et ***L*** au 0 binaire, mais les niveaux d'interruptions (également des nombres entiers) sont matérialisés par des tensions où ***H*** est associé au 0 binaire et ***L*** au 1 binaire. C'est comme çà.

Traditionnellement on qualifie de convention logique *positive* l'association entre ***H*** et 1, ou vrai, ou actif, et de convention logique *négative* l'association entre ***H*** et 0, ou faux, ou inactif. Dans le cas de la porte, qui doit, comme chacun sait, être « ouverte ou fermée », il n'y a pas de tradition. Le circuit 74xx08, par exemple, est connu comme étant une « positive logic and gate », littéralement « porte et en logique positive ». Si l'on change de convention logique le même circuit devient un opérateur Booléen différent (lequel ?).

I.2.2 Immunité au bruit

L'un des intérêts majeurs des signaux numériques est leur grande robustesse vis à vis des perturbations extérieures. L'exemple des enregistrements sonores en est une bonne illustration, le lecteur sceptique n'aura qu'à faire la simple expérience qui consiste à prendre une épingle (fine), à rayer un disque « noir », un disque compact, et à comparer les résultats. Derrière le résultat de cette expérience, quelque peu agressive, se cachent en fait deux mécanismes qui se complètent pour rendre le système numérique plus robuste, une protection au niveau du signal élémentaire, le bit, et une protection au niveau du système par le jeu du codage :

Au niveau du bit

La protection repose sur le fait que l'information n'est pas contenue dans l'amplitude du signal. Un signal analogique direct (on exclut ici les signaux modulés pour lesquels l'analyse devrait être affinée) a une forme qui est l'image de l'information à transmettre. Toute perturbation à cette forme se traduit par une déformation de l'information associée. L'amplitude d'un signal numérique n'a qu'un rapport très lointain avec l'information véhiculée, la seule contrainte est que le système soit encore capable de différencier sans ambiguïté un niveau haut et un

niveau bas. L'écart entre ces deux niveaux étant grand, seule une perturbation de grande amplitude pourra provoquer une erreur de décision[2].

Au niveau du système

Les valeurs élémentaires (bits) sont regroupées en paquets pour former des *mots*, ces mots doivent obéir à certaines règles de construction, des *codes*. Il est parfaitement imaginable, et c'est ce qui est fait dans tous les cas où l'on craint les perturbations, de construire des codes qui permettent de détecter et, dans une certaine mesure, de corriger des erreurs. Un exemple connu de tous, certes assez éloigné de l'électronique numérique, est la langue écrite. Un lecteur qui n'est pas totalement illettré est à même de détecter et de corriger un grand nombre d'erreurs typographiques, même sans faire appel au sens, d'un texte. La raison en est simple : les mots du dictionnaire sont loin de contenir toutes les combinaisons possibles des 26 lettres de l'alphabet, la construction d'une phrase obéit à des règles de grammaire bien connues du lecteur. Ces restrictions (mots du dictionnaire et grammaire) introduisent des redondances qui permettent justement d'assurer la robustesse du texte vis à vis des erreurs typographiques. Les codes détecteurs et/ou correcteurs d'erreurs sont tous fondés sur l'adjonction de redondances à la chaîne de bits transmis, ou inscrits sur un support fragile.[3]

Les choses ont un coût

Sans rentrer ici dans les détails, on peut remarquer que la robustesse des signaux numériques est liée à la très faible quantité d'information véhiculée par chaque signal élémentaire (0 ou 1). Le corrolaire de cette pauvreté du signal élémentaire est que pour traiter une information complexe il faut une quantité énorme de signaux élémentaires, merci M. de La Palice. Cela se traduit par un débit, dans le cas des transmissions, ou par un volume, dans le cas du stockage, très important. A titre d'exercice on calculera le nombre d'*octets* (paquets de 8 bits, *byte* dans le jargon) contenus dans un disque compact qui dure une heure, sachant que le signal est numérisé (CAN) 44 000 fois par seconde, et que chaque point occupe 2*16 bits (sans les redondances du code correcteur !). A titre de comparaison, le micro-ordinateur qui sert à rédiger ce texte dispose d'une mémoire de 8 Méga Octets, et d'un disque de 200 Méga Octets. On répondra ensuite à la question : pourquoi la télévision numérique ne sera-t-elle généralisée qu'au siècle prochain (il est vrai que c'est demain), au prix d'un investissement mathématique considérable pour « comprimer » l'information vidéo.[4]

---

[2]Cette question sera précisée quantitativement lors de la présentation des technologies des circuits (chapitre II).

[3]Un exemple simple de code correcteur sera étudié en exercice, quand les outils d'analyse seront disponibles.

[4]Une image télévision se répète au minimum 25 fois par seconde, contient 600 lignes de 700 points, chaque point est numérisé sur 2 octets.

I.3. Du bit au mot : des codes

Nous avons déjà évoqué que les informations élémentaires, que constituent les bits, sont souvent regroupées dans des paquets plus riches de sens, interprétables dans un code. Le monde du codage est vaste, nous nous contenterons ici de décrire rapidement quelques codes élémentaires, d'usage quotidien, et laisserons à l'initiative du lecteur l'exploration des codes sophistiqués, notamment les codes correcteurs d'erreurs.

I.3.1 Pour les nombres

Deux grandes catégories de nombres existent dans le monde informatico-logique : les entiers et les flottants. Les premiers constituent un sous ensemble *fini* de l'ensemble des entiers cher aux mathématiciens, les seconds tentent d'approcher, par un ensemble fini, les nombres réels. Le distinguo est de taille, les entiers que nous rencontrerons obéissent à une arithmétique euclidienne clairement définie, telle qu'on l'apprend dans les grandes classes de l'école primaire, les seconds obéissent à une arithmétique *approchée*, même si l'approximation est bonne.

Le lecteur averti pourra objecter que l'on rencontre parfois des nombres non entiers, caractérisés par un nombre connu de chiffres après la virgule (un exemple de tels nombres est rencontré sur vos relevés bancaires). Cette catégorie de nombres, connue sous le nom de *virgule fixe*, n'en est pas une : ils obéïssent à l'arithmétique entière (faites vos comptes en centimes), et sont convertis lors des opérations d'entrée-sortie (affichage, impression, saisie clavier).

Entiers naturels

La base 2

Etant donné un mot de n bits (a_{n-1},a_{n-2}, ... a_1,a_0), où les a_i valent 0 ou 1, on peut considérer que ce mot représente la valeur d'un entier A, écrit en base 2 :

$$A = a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + ... + a_1 2 + a_0$$

Les valeurs de A sont limitées par : $0 \leq A \leq 2^n - 1$

Les valeurs couramment rencontrées pour n sont 8 (octet), 16 (entier court) et 32 (entier long). Les bornes supérieurs correspondantes pour la valeur de A sont respectivement de 255, 65 535 et 4 294 967 295.

Quand n est un multiple de 4 (c'est le cas des valeurs évoquées ci-dessus...) il est souvent pratique, car plus compact, d'écrire le nombre en hexadécimal (base 16) :

$$A = (h_{m-1},h_{m-2}, ...,h_0) = h_{m-1}16^{m-1} + h_{m-2}16^{m-2} + ...+ h_0$$

Où m = n/4, et h_i peut prendre l'une des 16 valeurs 0, 1, 2 ....,9, A, B, C, D, F qui sont elle-mêmes représentables en binaire sur 4 bits.

Les opérations arithmétiques classiques sont l'addition, la soustraction, la multiplication et la division. On notera que :

1. Tous les résultats sont obtenus *modulo* 2^n, ce qui confère aux opérations sur l'ensemble des entiers sur n bits un caractère périodique comparable à celui des fonctions trigonométriques.
2. La division est la division entière, dont le résultat est constitué de deux nombres : le quotient et le reste.

La base 10

Les humains actuels ont pris la déplorable habitude de compter en base dix[5] qui n'est pas une puissance de 2 (60 non plus, d'ailleurs). Il n'y a donc pas de correspondance simple entre un nombre écrit en binaire et sa version décimale. Si cette difficulté est la source d'exercices élémentaires de programmation (programmes de changements de bases), elle est parfois gênante en pratique. C'est pour cette raison que l'on rencontre parfois des codes hybrides : le nombre est écrit en *chiffres décimaux*, et chaque chiffre est codé en binaire sur 4 bits. Le code le plus classique, dit *BCD* pour « binary coded decimal » consiste à coder chaque chiffre décimal (0 à 9) en binaire naturel (0000 à 1001). L'arithmétique de ce code n'est pas simple, l'addition de deux chiffres décimaux peut conduire à un résultat hors code (un nombre compris entre 10 et 15), ou faux mais apparemment dans le code (un nombre compris entre 16 et 18) ; après chaque opération élémentaire il faut donc recaler le résultat intermédiaire. Cette opération supplémentaire s'appelle *ajustement décimal*, il faut l'effectuer après tout calcul sur une tranche de 4 bits. Beaucoup de calculettes utilisent un code BCD qui facilite les opérations d'affichage.

D'autres codes décimaux existent, qui facilitent un peu les calculs, mais ils sont d'un usage rarissime actuellement.

Entiers signés

Quand on aborde la question des entiers signés il est essentiel de se souvenir *qu'un mot de n bits ne peut fournir que* 2^n *combinaisons différentes*. Comme on ne peut pas avoir le beurre et l'argent du beurre, il faudra restreindre la plage des valeurs possibles pour la valeur absolue du nombre.

Les êtres humains ont l'habitude de représenter un nombre signé dans un code qui sépare le signe et la valeur absolue du nombre. Le signe étant une grandeur binaire (+ ou -), on peut lui affecter un bit, le *bit de signe*, et garder les n-1 bits restant pour coder la valeur absolue, en binaire naturel, par exemple. Ce type de code, connu sous le nom de « signe-valeur absolue » n'est en fait jamais utilisé pour les nombres entiers (il l'est par contre pour les flottants). La raison en est que l'arithmétique sous-jacente est compliquée ; en effet pour additionner ou soustraire

[5]On notera que vers 3000 avant J.C. les Sumériens avaient fort judicieusement choisi la base 60, multiple de 2, 3, 4, 5, 6, 10, 12. Il est vrai que l'apprentissage des tables de multiplications ne devait pas être à la portée de tous.

deux nombres signés, dans un code signe-valeur absolue, il faut commencer par déterminer quel sera le signe du résultat. Pour ce faire il faut commencer tout calcul par une comparaison qui fait intervenir à la fois les signes et les valeurs absolues des opérandes (remémorez-vous vos débats avec les mathématiques du début du collège).

Les deux codes universellement utilisés pour représenter les entiers relatifs, présentés ci-dessous, évitent cet écueil, additions et soustraction ne sont qu'une même opération, qui ne fait pas intervenir de comparaison, et les reports (ou retenues) éventuels sont simples à traiter.

Dans l'un des codes comme dans l'autre, l'intervalle de définition d'un nombre A, codé sur n bits, est donné par :

$$-2^{n-1} \leq A \leq 2^{n-1} - 1$$

Soit

-128 à 127 pour n = 8,
-32 768 à 32 767 pour n= 16
et -2 147 483 648 à 2 147 483 647 pour n=32.

Attention !

- Le caractère périodique des opérations implique, par exemple, que dans un code signé *sur 8 bits 100 + 100 = -56*, qui est bien égal à 200 modulo 256, il ne s'agit pas là d'une erreur mais d'une conséquence de la restriction à un sous-ensemble fini des opérations sur les entiers.
- Le changement de longueur du code, par exemple le passage de 8 à 16 bits, n'est pas une opération triviale, la combinaison binaire qui représente 200 en binaire naturel, n'a pas forcément la même signification quand on l'interprète dans un code 8 bits ou un code 16 bits.

Le code complément à 2

C'est le code utilisé pour représenter les nombres entiers dans un ordinateur. Il présente l'intérêt majeur de se prêter à une arithmétique simple, mais a pour défaut mineur que la représentation des nombres négatifs n'est pas très parlante pour un être humain normalement constitué. La construction du code complément à deux, *sur n bits*, découle directement de la définition modulo 2^n des nombres. Etant donné un nombre A :

⇒ **Si $A \geq 0$** le code de A est l'écriture en *binaire naturel de A*, éventuellement complété à gauche par des 0.
Exemple : A = 23, codé sur 8 bits s'écrit : 00010111.

⇒ **Si $A < 0$** le code de A est l'écriture en *binaire naturel de* $2^n + A$, ou, ce qui revient au même, de $2^n - |A|$.
Exemple : A = – 23, codé sur 8 bits s'écrit : 11101001, qui est la représentation en binaire naturel de 256 – 23 = 233 (E9 en hexadécimal).

On remarquera que le bit le plus à gauche, le bit de poids fort ou *MSB* (pour *most significant bit*), est le bit de signe, avec la convention logique 1 pour – et 0 pour +.

Le calcul de l'opposé d'un nombre, quel que soit le signe de ce nombre, est une simple conséquence de la définition du code : $-A = 2^n - A$ modulo 2^n.

Par exemple :

$$-(-23) = 256 + 23 \text{ modulo } 256 = 23.$$

Astuce de calcul : Pour obtenir rapidement l'expression binaire de l'opposé d'un nombre dont on connait le code, on peut utiliser l'astuce suivante (que certains, à tort, prennent pour une définition du code) :

$2^n - A = 2^n - 1 - A + 1$ (vérifiez)

$2^n - 1$ est le nombre dont tous les chiffres binaires sont à 1,

$2^n - 1 - A$ est le nombre que l'on obtient en remplaçant dans le code de A les 1 par 0 et réciproquement.

Ce nouveau nombre s'appelle *complément à 1* ou complément *restreint* de A, et se note classiquement $\overline{A}$.

Suivant la tradition on peut alors écrire :

$$-A = \overline{A} + 1.$$

Exemple :

$23 = 00010111$,

$\overline{23} = 11101000$,

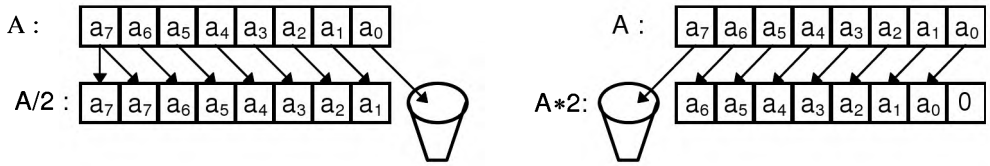
$-23 = \overline{23} + 1 = 11101001$,

qui est le résultat précédent.

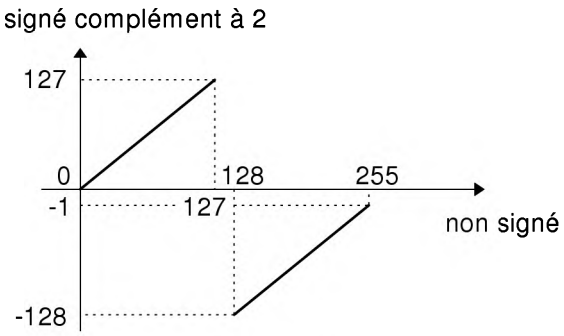
L'augmentation de longueur du code (par exemple le passage de 8 à 16 bits) se fait en complétant à gauche par le bit de signe, 0 pour un nombre positif, 1 pour un nombre négatif. Cette opération porte le nom d'*extension de signe*. Exemple – 23 s'écrit 11101001 sur 8 bits et 1111111111101001 sur 16 bits, ce qui est notablement différent de 0000000011101001 qui est le code de 233, dont l'existence est maintenant légale.

Les additions et soustractions des nombres ne sont qu'une seule et même opération : des additions et des éventuels changements de signes, sans que l'on ait jamais à faire de comparaison, et où les reports (ou retenues) sont générées mécaniquement. La dissymétrie entre addition et soustraction, bien connue des élèves des premières années de collège, a disparu.

Les multiplications et les divisions par des puissances de 2 sont des décalages *arithmétiques* (i.e. avec conservation du signe), par exemple pour des octets :



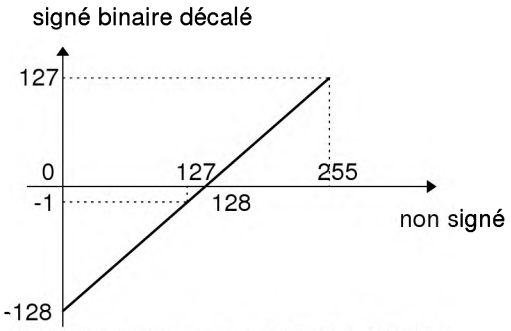
N.B. : Les détails de manipulation des nombres, lors des opérations arithmétiques, sont évidemment transparentes pour le programmeur généraliste, elles sont intéressantes à connaître pour le concepteur d'unités de calcul, et, exceptionnellement, pour le programmeur qui s'occupe des interfaces logiciels avec le matériel (pilotes de périphériques, librairies de bas niveau, etc...).



Relation binaire naturel complément à deux sur 8 bits.

Si le code complément à deux se prête bien aux calculs, il complique les opérations de comparaisons et, plus généralement, les opérations qui font intervenir une relation d'ordre entre les nombres. Précisons cette question de relation d'ordre : le code en complément à 2 de –1, par exemple, correspond à tous les chiffres binaires à 1, qui représente le plus grand nombre possible dans une interprétation non signée. Face à la combinaison binaire correspondant à –1 la réponse à la question « ce nombre est-il supérieur à 10 ? » ne devra pas être traitée de la même façon par un opérateur câblé suivant que le code est signé ou non. La figure ci-dessous tente d'illustrer cette rupture dans la relation d'ordre.

Le code binaire décalé



Relation binaire naturel binaire décalé sur 8 bits.

Le code binaire décalé ne présente pas l'inconvénient évoqué ci-dessus à propos de la relation d'ordre : il possède *la même relation d'ordre que le code binaire naturel* des nombres non signés. On le rencontre dans certains convertisseurs numériques analogiques (ou, plus rarement analogiques numériques) et dans la représentation de l'exposant des nombres flottants. L'application qui fait passer du binaire naturel au

binaire décalé est définie en sorte que le minimum de l'un des codes corresponde au minimum de l'autre, et que le maximum de l'un des codes corresponde au maximum de l'autre :

Un examen rapide de la courbe précédente fournit la formule de génération du code binaire décalé sur n bits : étant donné un nombre entier A tel que $-2^{n-1} \leq A \leq 2^{n-1} - 1$, le code de A est le code binaire naturel de $A + 2^{n-1}$, d'où le nom du code. On peut remarquer que le nombre 2^{n-1} a son MSB égal à 1, tous les autres chiffres étant nuls.

On passe du code binaire décalé au code complément à deux en complémentant le bit de signe.

Flottants

Si A est un nombre flottant il est codé par une expression du type :

$$A = (-1)^s * 2^e * 1,xxxxx...$$

où s est le signe de A, e un nombre *entier signé*, codé en binaire décalé, et xxxxx... la partie fractionnaire de la valeur absolue de la mantisse. A titre d'exemple les flottants double précision (64 bits) suivant la norme ANSI ont les caractéristiques suivantes:

- signe s : 1 élément binaire, 1 pour –, 0 pour +,
- exposant compris entre –1022 et +1023, soit 11 éléments binaires,
- partie fractionnaire de la mantisse sur 52 éléments binaires.
- combinaisons réservées pour le zéro, l'infini (+ et –) et NAN (not a number).

Cela correspond à une dynamique allant de $2,2*10^{-308}$ à $1,8*10^{+308}$, pour une précision de l'ordre de 16 chiffres décimaux.

De ce qui précède il faut tirer deux conclusions importantes :

1. L'arithmétique des nombres flottants et celle des entiers font appel à des algorithmes radicalement différents.
2. Le test d'égalité de deux nombres, qui a un sens clair pour des entiers, fournit un résultat aléatoire dans le cas des flottants. Seule une majoration de leur écart conduit à un résultat prévisible.

Le format binaire des nombres est tel que seule la partie fractionnaire de la mantisse figure, pour des flottants simple précision :

nombre flottant :	s	exposant	partie fractionnaire de la mantisse
poids des bits :	31	30----------23	22----------0

I.3.2 Il n'y a pas que des nombres

Toute catégorie de données un tant soit peu organisée est susceptible de générer un code ; il est évidemment hors de question d'en faire un catalogue un tant soit peu exhaustif. Nous nous contenterons de citer quelques exemples.

Un cas important dans les applications concerne tout ce qui est échange d'informations, par exemple entre deux ordinateurs ou entre un ordinateur et une imprimante, sous forme de texte : cela conduit aux codes alphanumériques. Le code alphanumérique le plus utilisé porte le nom de code ***ASCII*** pour « American standard code for information interchange », décrit ci-dessous.

Le code ASCII

Sept bits pour les caractères anglo-saxons

Pour représenter l'ensemble des lettres de l'alphabet - minuscules et majuscules, sans les accents, les dix chiffres décimaux, les caractères de ponctuation, les parenthèses crochets et autres accolades, les symboles arithmétiques les plus courants et des commandes : 128 combinaisons suffisent. D'où le code ASCII, sur 7 bits ($b_6b_5b_4b_3b_2b_1b_0$), quasi universellement adopté :

				b_6	0	0	0	0	1	1	1	1
				b_5	0	0	1	1	0	0	1	1
				b_4	0	1	0	1	0	1	0	1
b_3	b_2	b_1	b_0	Hex	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
1	0	1	1	B	VT	ESC	+	;	K	[	k	{
1	1	0	0	C	FF	FS	,	<	L	\	l	|
1	1	0	1	D	CR	GS	-	=	M	]	m	}
1	1	1	0	E	SO	RS	.	>	N	^	n	~
1	1	1	1	F	SI	US	/	?	O	_	o	DEL

Les codes de valeurs inférieures à 32 (en décimal) sont des commandes ou des caractères spéciaux utilisés en transmission. Comme commandes on peut citer, à titre d'exemples, CR pour retour chariot, LF pour nouvelle ligne, FF pour nouvelle page, BEL pour « cloche » etc...

Il est clair que le programmeur généraliste qui utilise un langage évolué n'a en fait jamais à connaître les valeurs des codes, ils sont évidemment connus du compilateur, par contre l'auteur d'un pilote d'imprimante peut, lui, avoir à se pencher sur ces choses peu conviviales !

Huit bits pour les accents

Quand on passe aux dialectes régionaux, le français par exemple, un problème se pose : le code ASCII ne connait pas les lettres accentuées. Pour palier ce manque, les auteurs de logiciels et/ou les fabriquants de matériels (imprimantes) ont complété le code, en l'étendant à un octet soit 256 combinaisons différentes. Malheureusement les règles disparaissent quand on passe aux codes caractères sur 8 bits, on peut même trouver des ordinateurs qui, suivant les logiciels, utilisent des combinaisons variables pour représenter les é, è, à, ç, î et autres caractères dialectaux. Nous ne citerons pas de nom.

Un bit de plus pour les erreurs

Quand on craint que des erreurs se produisent au cours d'une transmission (minitel, par exemple), on rajoute parfois un bit supplémentaire (8ème ou 9ème suivant la taille du code initial) on rajoute un chiffre binaire supplémentaire qui est calculé de telle façon que pour chaque caractère transmis :

- Le nombre de bits à un soit impair, on parle alors de parité impaire,

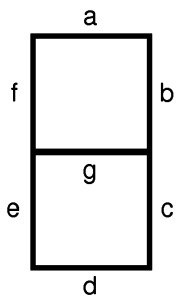
ou :

- le nombre de bits à un transmis soit pair, on parle alors de parité paire.

Si émetteur et récepteur utilisent la même convention de parité, le récepteur est capable de détecter une faute de transmission tant que le nombre de fautes n'excède pas une erreur par caractère. Un procédé aussi rudimentaire ne permet évidemment pas de corriger l'erreur autrement qu'en demandant une retransmission du caractère incriminé.

Autres codes

Concurrent de l'ASCII dans le codage des caractères on peut citer le code EBCDIC (extended binary coded decimal interchange code), encore utilisé dans certains systèmes de gestion.



Pour afficher un chiffre décimal sur un afficheur « 7 segments » il faut associer à chaque chiffre une combinaison qui indique les segments (a, b, c, d, e, f et g) à allumer :

Le passage d'un code à un autre, pour des informations qui ont le même contenu, s'appelle un *transcodage*. A titre d'exemple on peut imaginer comment construire un transcodeur BCD $\rightarrow$ 7 segments.

Exercices

Débordements

A et B sont des nombres entiers, codés en binaire. Pourquoi, même s'il n'y a pas de débordement, l'opération 2*(A/2 + B/2) donne-t-elle parfois un résultat différent de (A + B) ?

Justifiez votre réponse en vous appuyant sur une représentation binaire sur 8 bits. Quelle est la valeur numérique maximum de l'écart entre les deux résultats ?

Nombres entiers signés

Montrer que l'équation

$$x = -x$$

a deux solutions pour des nombres entiers signés, dans le code complément à deux sur n bits.

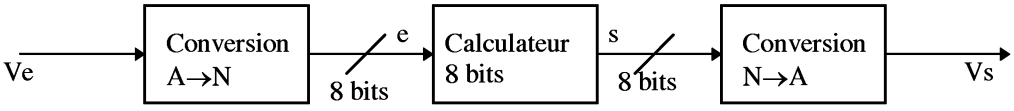
Volumes et débits d'informations

Un enregistrement sur disque compact consiste à prendre une suite d'échantillons (on trace la courbe en pointillé...) à la cadence de 44 kHz. Chaque échantillon est représenté par un nombre de 16 éléments binaires pour chacune des voies droite et gauche (stéréophonie).

- A combien d'octets (paquet de 8 bits) correspond un disque d'une heure de musique enregistrée ?
- A quelle vitesse, en bits par seconde, doit travailler le lecteur pour restituer la musique en temps réel (ce qui est souhaitable) ?
- Sachant que la fréquence maximum des signaux sonores est de 20 kHz, quel est le "coût" du numérique ? Quels sont ses avantages ?
- Reprendre les calculs précédents pour un signal de télévision, de fréquence maximum 5 MHz, codé sur 8 bits (l'oeil est très tolérant) à 2,2 fois la fréquence maximum.
- La télévision numérique peut-elle être une simple transposition des techniques utilisées pour le son ?

Conversions.

Une chaîne de traitement de l'information est constituée comme suit :



Les convertisseurs analogique numérique et numérique analogique ont un pas de 40 mV (écart de tensions analogiques qui correspond à une différence de 1 bit de pods faible). Toute la chaîne est bipolaire : tensions d'entrée et de sortie positive ou négative, nombres entiers signés (sur 8 bits). Quelles sont les tensions maximum et minimum des tensions analogiques d'entrée et de sortie ?

Le calculateur effectue à chaque étape (n) la moyenne des quatre échantillons précédents :

$$s(n) = 1/4*(e(n) + e(n-1) + e(n-2) + e(n-3))$$

1. Un premier programme de calcul est la traduction directe de la formule précédente. En relevant la fonction de transfert Vs = f(Ve), le programmeur s'apperçoit que les résultats en sortie sont parfois étranges. Pourquoi ?
2. Ayant trouvé son erreur, le programmeur utilise cette fois la formule :

$$s(n) = e(n)/4 + e(n-1)/4 + e(n-2)/4 + e(n-3)/4$$

3. Les résultats sont moins absurdes, mais un autre défaut apparaît. Lequel ?
4. Quelle serait la solution du problème ?

II Circuits : aspects électriques

II.1. Technologies

Les circuits numériques sont subdivisés en familles technologiques. A chaque famille est associée un processus de fabrication qui recouvre un type de transistor (bipolaires, MOS etc...), donc des paramètres électriques : tensions d'alimentations, niveaux logiques, courants échangés lors de l'association de plusieurs opérateurs, caractéristiques dynamiques comme les temps de propagations, les fréquences d'horloge maxima. Le principe général adopté est que l'utilisateur peut construire une fonction logique complexe en associant des opérateurs élémentaires sans se poser à chaque fois des questions d'interface électrique *tant qu'il utilise des circuits d'une même famille.*

Nous n'étudierons pas ici l'architecture interne des différentes technologies utilisées en électronique numérique[1], pour le concepteur de système numérique un circuit apparaît comme une « boite noire » dont le fonctionnement est entièrement défini par ses caractéristiques externes, tant statiques (volts et milliampères) que dynamiques (nanosecondes et mégaherz). Avant d'aborder ces deux points, nous passerons en revue, de façon très générale les familles les plus utilisées en pratique.

II.1.1 Les familles TTL

Famille historique s'il en est, apparue au milieu des années 1960, la famille TTL (Transistor Transistor Logic), construite autour de transistors bipolaires, est devenue un standard de fait. Les premières versions sont devenues complètement obsolètes mais servent d'éléments de comparaison. Dans la version TEXAS INSTRUMENT, repris par de nombreuses secondes sources, le code d'identification d'un circuit TTL est relativement standardisé :

SN 74 AS 169 N ou *DM 54 S 283 J*

---

[1]On consultera avec profit HOROWITZ et HILL, *The Art of Electronics*, Cambridge University Press, 1983 ; MILLMANN et GRABEL *Microélectronique*, McGraw-Hill, 1988 ou HODGES et JACKSON *Analysis and design of digital integrated circuits*, McGraw-Hill, 1988.

Chaque champ a une signification :

- SN, DM : champ littéral qui indique le constructeur.
- 74 ou 54 : gamme de températures normale (0°C à 70°C) ou militaire (-55°C à +125°C).
- AS, S, ...: technologie ici advanced shottky, shottky.
- 169, 283, ...: fonction logique.
- N, J, P, NT...: type de boîtier (ici DIL plastique ou céramique).

Les familles TTL nécessitent une *alimentation monotension de +5 V*. Attention, cette spécification est très stricte, et doit être respectée à *±10 %* près, voire *±5 %* près dans certains cas. Le dépassement de la tension d'alimentation maximum, de même que l'inversion de cette tension, par permutation accidentelle entre masse et alimentation, est l'un des moyens de destruction du circuit le plus sûr.

Le tableau ci-dessous résume quelques éléments clés des familles TTL :

Technologie	Commentaire	P mW	t_p ns
74/54 N	Série historique standard, transistors saturés, obsolète	10	10
74/54 H	Série historique rapide, obsolète	20	5
74/54 L	Série historique faible consommation (low power), obsolète	1	30
74/54 S	Shottky, série rapide, transistors non saturés, presque obsolète	20	3
74/54 LS	Shottky faible consommation, très répandue, standard de fait, presque obsolète	2	10
74/54 F	Version Fairchild des technologies rapides	4	3
74/54 AS	Advanced Shottky, remplace la « S »	8	2
74/54 ALS	advanced low power shottky, remplace la « LS »	2	4

Dans le tableau précédent, la puissance P et le temps de propagation (retard) t_p sont mesurés pour un opérateur élémentaire (porte), typiquement un inverseur.

Les familles TTL sont caractérisées par une consommation non négligeable, de l'ordre de quelques milliwatts par porte, qui augmente un peu avec la fréquence d'utilisation, et des fréquences maximums de fonctionnement comprises entre 10 et 100 Mhz suivant les versions. Les niveaux logiques typiques sont de l'ordre de 3 V pour le niveau haut et 0,4 V pour le niveau bas (voir plus loin).

II.1.2 Les familles CMOS

Apparue à la même époque que la famille TTL N, la première famille CMOS (Complementary Metal Oxyde Semi-conductor), la série 4000 de RCA, s'est rendue populaire par sa très faible consommation statique (pratiquement 0) et par une grande plage de tension d'alimentation (3 à 15 V), malgré des performances dynamiques quatre à dix fois plus mauvaises, dans le meilleur des cas[2]. Cette

[2]Le retard dans les circuits dépend beaucoup de la capacité de charge en sortie pour la famille 4000.

famille est strictement incompatible avec la famille TTL, à la fois pour des questions de niveaux logiques que de courant absorbé par les portes TTL.

La famille des circuits CMOS s'est agrandie depuis, dans deux directions

1. Circuits spécialisés à très faible tension d'alimentation (1,5 V), très faible consommation, où la vitesse n'intervient pas, ou peu (montres, calculettes simples, etc...). Nous n'en parlerons pas plus.
2. Circuits qui concurrencent les familles TTL, même rapides, avec une consommation statique pratiquement nulle : 4000B, 74 C, 74 HC, 74 HCT, 74 ACT, 74 FACT etc...Les familles 74xxx sont fonctionnellement équivalentes aux familles TTL, *mais le brochage des circuits est parfois différent*, la lettre 'T' indique la compatibilité de niveaux électriques avec les familles TTL.

Les notices des circuits CMOS sont à analyser avec prudence quand on les compare aux autres familles :

- La consommation est proportionnelle à la fréquence de fonctionnement, nulle à fréquence nulle, la puissance absorbée par porte rejoint celle des familles bipolaires aux alentours d'une dizaine de mégahertz. Une formule approchée permet d'estimer la puissance absorbée par une porte élémentaire :

$$P_d = (C_L + C_{PD}) * V_{cc} * (V_H - V_L) * f$$

où C_L est la capacité de charge, C_{PD} une capacité interne équivalente de l'ordre de 25 pF pour les familles 74 AC, V_{cc} la tension d'alimentation[3] et f la fréquence de fonctionnement.
- Les circuits MOS présentent une caractéristique d'entrée qui peut être assimilée à une capacité, le temps de propagation et la consommation par porte augmentent notablement quand la capacité de charge, donc le nombre d'opérateurs commandés, augmente.

Le tableau ci-dessous résume quelques éléments clés des familles CMOS :

Technologie	Commentaire	P mW	t_p ns
4000	Série historique, non compatible TTL, obsolète	0,1	100
74/54 C	Partiellement compatible TTL, obsolète	0,1	50
74/54 HC	Partiellement compatible TTL	0,1	10
74/54 HCT	Compatible TTL	0,1	10
74/54 ACT	Compatible TTL, rapide	0,1	5

Dans le tableau précédent les chiffres sont donnés pour une *capacité de charge de 50 pF* et une *fréquence de travail de 1 Mhz*. La compatibilité TTL, si elle est mentionnée, n'a de sens que pour une tension d'alimentation de 5 V.

Les CMOS sont un peu la famille idéale pour les applications courantes. Quelques précautions d'emploi sont cependant à noter :

[3]Cette formule met clairement en évidence l'intérêt du passage, qui tend à se généraliser, de 5 V à 3,3 V pour la tension d'alimentation des circuits numériques.

- Les entrées inutilisées ne doivent *jamais être laissées « en l'air »*, l'oubli de cette précaution, qui peut conduire à des dysfonctionnements des familles TTL, peut être destructive dans le cas des CMOS.
- Les signaux d'entrée ne doivent jamais être appliqués à un circuit non alimenté. Si le potentiel d'une entrée dépasse celui de la broche d'alimentation du circuit, cela peut provoquer un phénomène connu sous le nom de « latch up », destructif, qui est un véritable court-circuit interne[4].
- Les circuits sont sensibles aux décharges électrostatiques, les mémoires à grande capacité, qui font appel à des transistors de dimensions sub-microniques, ne doivent être manipulées que par un opérateur muni d'un bracelet conducteur relié à la masse du montage.
- L'augmentation de la vitesse, conjointement à la consommation statique nulle (résistances équivalentes infinies), conduit à une très forte désadaptation, au sens des lignes de propagation, des circuits vis à vis des conducteurs d'interconnexions. Cette désadaptation conduit à des phénomènes d'échos : une impulsion peut être réfléchie en bout de ligne, et générer un écho, c'est à dire une impulsion « parasite » qui peut conduire à des erreurs de fonctionnement.
- La consommation d'un circuit CMOS n'est pas du tout régulière, mais formée d'une suite d'impulsions de courant, à chaque changement d'état ; si ces impulsions de courant se retrouvent dans les fils d'alimentation ceux-ci se comportent comme autant d'antennes qui émettent des signaux parasites. Globalement cela se traduit par un comportement très bruyant des systèmes numériques qui utilisent une technologie CMOS sans respecter les règles de l'art concernant le câblage. Parmis ces règles de l'art la plus importante, et de loin, est le *découplage haute fréquence de l'alimentation de chaque circuit*. Pratiquement il faut adjoindre à chaque boîtier une capacité de découplage (10 à 100 nF), entre alimentation et masse. Cette capacité doit présenter une impédance aussi faible que possible en haute fréquence (plusieurs centaines de mégaherz), elle doit donc avoir une inductance parasite aussi faible que possible : fils courts, technologie « mille feuilles », l'idéal étant une capacité « chip » soudée directement sous le circuit à découpler. Pour résumer : *dans les conducteurs d'alimentation et de masse ne doivent circuler que des courants continus.*

[4]Pour les initiés : la structure CMOS présente un thyristor parasite qui, s'il est mis en conduction, court-circuite les alimentations. Un dépassement de la tension d'alimentation par une des entrées peut mettre ce thyristor en conduction. Les circuits récents sont mieux protégés contre ce phénomène que ceux des premières générations, mais le problème n'a pas complètement disparu.

II.1.3 Les familles ECL

Les familles ECL constituent en quelque sorte l'aristocratie des familles logiques. Très rapides, temps de propagation inférieur à la nano-seconde pour une porte, temps d'accès de moins de 10 nano-secondes pour les mémoires, ces familles constituent un monde à part. Elles sont strictement incompatibles avec la TTL, ne serait-ce que par leur tension d'alimentation qui est négative, –5,2 V, et par des niveaux logiques haut et bas de –1 V et –1,6 V respectivement. Le fonctionnement interne fait appel à des amplificateurs différentiels, en technologie bipolaire, qui fonctionnent en régime linéaire. Cette particularité leur confère un courant absorbé pratiquement constant, ce qui rend les circuits peu bruyants, et facilite l'adaptation d'impédance aux lignes d'interconnexions. La contrepartie du fonctionnement en régime linéaire est une consommation importante. Le tableau ci-dessous résume quelques caractéristiques des deux familles principales :

Technologie	Commentaire	P mW	t_p ns	f max
ECL 10K	Série historique, non compatible TTL	25	2	125 MHz
ECL 100K	Meilleure stabilité en température, plus rapide, non compatible TTL	30	0,8	400 MHz

La puissance et le temps de propagation concernent une porte élémentaire, la fréquence maximum de fonctionnement concerne un circuit séquentiel synchrone simple.

Les domaines d'applications des technologies ECL sont les « super ordinateurs », et les parties hautes fréquences des systèmes de télécommunication. On trouve des circuits dérivés de l'ECL, dont le fonctionnement interne est celui de cette famille, mais qui apparaissent au monde extérieur comme compatibles TTL, alimentation comprise.

II.1.4 Les familles AsGa

D'un usage industriel encore limité à quelques fonctions relativement simples, en général dans les parties hautes fréquences des systèmes de télécommunications et de radars, ces technologies surpassent les familles ECL dans le domaine des fréquences allant de 500 Mhz à 5 Ghz. Elles utilisent comme composants élémentaires des transistors à effet de champ à jonction, MESFET (pour Metal Semi-conductor Field Effect Transistor), dont la jonction de commande est une diode Shottky. L'origine de la vitesse de ces transistors est que la mobilité des électrons est cinq à dix fois plus élevée dans l'arseniure de gallium que dans le silicium.

Le tableau ci-dessous indique quelques unes des performances atteintes :

Technologie	Commentaire	P mW	t_p ps	f max
BFL	Buffered FET logic, Géométrie 0,5µm	10	55	2,5 GHz
DCFL	Direct coupled FET logic, 0,5 µm	1,3	11	4 GHz

La puissance et le temps de propagation (en pico-secondes) concernent une porte élémentaire, la fréquence maximum de fonctionnement (en gigahertz) concerne un circuit séquentiel synchrone simple.

II.2. Volts et milliampères

Le principe de constitution d'une famille logique est de permettre au concepteur d'une application d'interconnecter les circuits sur une carte de la même façon qu'il assemble des fonctions sur un schéma de principe. Ce jeu de Lego est rendu possible par le respect, par les fabriquants de circuits, de règles cohérentes, communes à tous les fabriquants, qui rendent compréhensibles, par les entrées d'un circuit, les signaux issus des sorties d'un autre. Le jeu se complique un peu à cause de l'inévitable dispersion des caractéristiques, d'un composant à l'autre lors de la fabrication, dispersion initiale à laquelle il convient de rajouter les variations des caractéristiques d'un même circuit avec la température. Tous les paramètres électriques d'un circuit intégré seront définis par trois valeurs : minimum, maximum, dans une plage de température, et typique, à température « normale », c'est à dire 25 °C (300 °K).

II.2.1 Les niveaux de tension

A un circuit, alimenté par une tension V_{CC}, on applique une tension d'entrée V_e et on mesure la tension de sortie V_s.

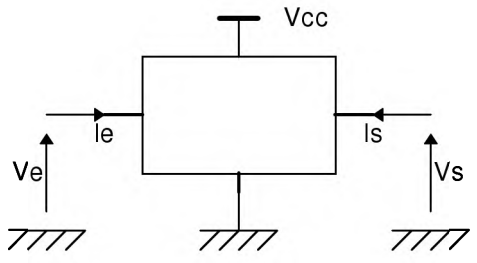


Figure II-1

Les niveaux HAUT et BAS, en entrée et en sortie, V_{IH}, V_{OH}, V_{IL} et V_{OL} sont définis par :

Niveaux bas en entrée si $0 \leq V_e \leq V_{IL}$
Niveaux bas en sortie si $0 \leq V_s \leq V_{OL}$
Niveaux haut en entrée si $V_{IH} \leq V_e \leq V_{cc}$
Niveaux haut en sortie si $V_{OH} \leq V_s \leq V_{cc}$

Il est clair qu'entre un niveau haut et un niveau bas doit exister une « plage interdite », pour qu'il n'y ait pas ambiguïté.

Quand on envisage l'association de deux circuits, A et B, il convient de rendre compatibles les niveaux d'entrée et de sortie.

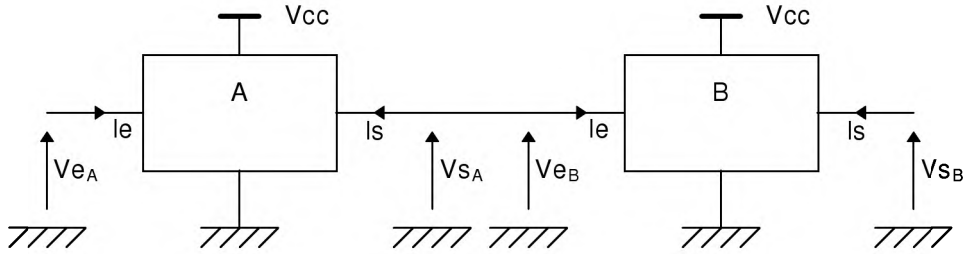


Figure II-2

Pour assurer que le circuit B comprend bien les signaux issus du circuit A, on doit avoir :

$V_{OHMIN} > V_{IHMIN}$
$V_{OLMAX} < V_{ILMAX}$

Dans ces inégalités, un peu paradoxales, il faut bien comprendre que les attributs « MIN » et « MAX » ont un sens statistique, ils concernent les valeurs extrêmes que le constructeur garantit sur tous les circuits d'une même famille technologique.

Un catalogue de composants TTL nous renseigne sur la valeur de ces paramètres dans cette famille : V_{OHMIN} = 2,7 V et V_{IHMIN} = 2 V, V_{OLMAX} = 0,4 V et V_{ILMAX} = 0,8 V.

Ces valeurs respectent bien évidemment les inégalités précédentes.

Entre les familles TTL et CMOS traditionnelles la compatibilité n'est pas assurée dans le sens TTL→CMOS pour le niveau haut.

La valeur minimum des écarts entre V_{OHMIN} et V_{IHMIN} d'une part, V_{OLMAX} et V_{ILMAX} d'autre part représente *l'immunité au bruit* de la famille considérée. Elle est de 400 mV en TTL. Cette immunité au bruit représente l'amplitude que doit avoir un parasite, superposé au signal utile, qui risque de rendre ambiguë la tension d'entrée d'un circuit. La figure II-3 résume les définitions qui précèdent :

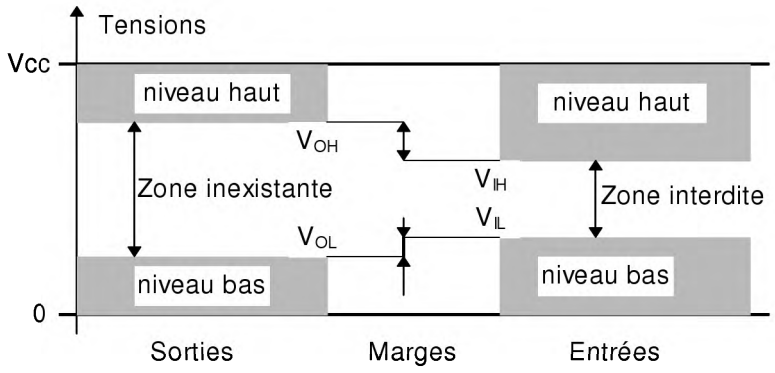


Figure II-3

II.2.2 Les courants échangés

Les courants I_e et I_s des figures II-1 et II-2 indiquent des conventions de signe pour des courants dont les sens réels dépendent des niveaux logiques qui interviennent. En première approximation, on peut considérer que la sortie d'un circuit se comporte comme une source de tension ; dans un montage comme celui de la figure II-2, la valeur du courant qui circule dans la liaison entre les deux circuits est alors principalement fixée par l'étage d'entrée du circuit récepteur de l'information. Pour un niveau bas Ie est négatif (le courant « sort » du récepteur), il est positif pour un niveau haut. Quand un circuit en commande plusieurs, son courant de sortie est, à un signe près, égal à la somme des courants d'entrée des circuits commandés :

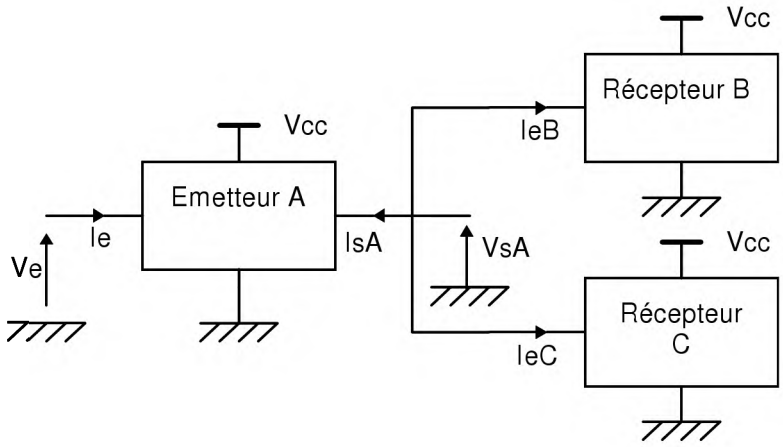


Figure II-4

$$I_{s\,A} = -(I_{e\,B} + I_{e\,C})$$

Pour déterminer la validité d'une association telle que celle représentée figure II-4, il faut connaître les valeurs maximums (en valeurs absolues) des courants d'entrée, et la valeur maximum tolérable pour le courant de sortie. C'est dans cette optique que sont définis :

- I_{IH} et I_{IL} , courants d'entrée d'un circuit auquel on applique des niveaux haut et bas, respectivement.
- I_{OH} et I_{OL} , courants de sortie admissibles par un circuit tout en conservant les niveaux de tension haut et bas, respectivement.

Pour assurer la validité d'une association dans laquelle un circuit en commande plusieurs autres, il faut contrôler que sont vérifiées les deux inégalités :

$$-I_{OHMAX} > \Sigma(I_{IHMAX})$$
$$I_{OLMAX} > \Sigma(-I_{ILMAX})$$

Les signes '–' proviennent des conventions de signes classiquement adoptées, seules comptent, évidemment, les valeurs absolues des courants.

En TTL-LS : $I_{OHMAX} = -0{,}4$ mA pour $I_{IHMAX} = 20$ µA, $I_{OLMAX} = 8$ mA pour $I_{ILMAX} = -0{,}4$ mA

On en déduit qu'un circuit peut en commander 20 autres tout en assurant le respect des niveaux logiques.

Les inégalités précédentes, associées à leurs semblables concernant les tensions, permettent de déterminer la validité d'associations entre circuits de technologies différentes, ou de spécifier un circuit d'interface « fait maison » avec une technologie donnée. A l'intérieur d'une technologie les niveaux de tension sont évidemment compatibles, les règles concernant les courants se résument alors à contrôler le bon respect des sortances et entrances des circuits :

On prend comme unité logique la charge apportée par l'entrée d'une porte élémentaire de la famille technologique considérée (en général l'inverseur). On définit alors deux nombres entiers :

- La *sortance (fan out)* d'une sortie est égale au nombre maximum de charges élémentaires que peut piloter cette sortie.
- L'*entrance (fan in)* associée à une entrée d'un circuit complexe est égale aux nombres de charges élémentaires équivalentes aux courants absorbés (ou fournis) par cette entrée.

La valeur typique de sortance adoptée par les fabriquants est de 20.

N.B. : Autant ce qui précède a un sens clair pour les technologies dont les consommations sont peu dépendantes de la vitesse de fonctionnement, TTL et ECL par exemple, autant les consommations statiques n'ont *aucun sens appliquées aux technologies CMOS*. Pour ces technologies les calculs de sortances conduisent à des résultats absurdes, parce qu'applicables uniquement à une application qui ne fait rien ! Pour ces technologies l'augmentation du nombre d'entrées mises en parallèle se traduit par une augmentation de la capacité de charge présentée au circuit de commande, il en résulte une augmentation des temps de propagation des signaux, donc une baisse de

vitesse du système. Les notices de circuit donnent les capacités des entrées et des courbes de temps de propagation en fonction de la capacité de charge.

II.3. Nanosecondes et mégahertz

Avant de préciser les paramètres dynamiques que l'on définit pour caractériser les circuits logiques, rappelons brièvement comment on caractérise une impulsion :

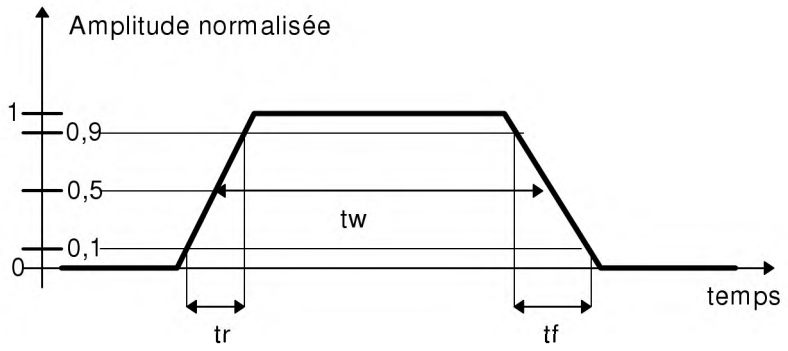


Figure II-5

Les noms des différents temps qui interviennent sont :

t_w : largeur (width)
t_r : temps de montée (rise time)
t_f : temps de descente (fall time).

Quelle que soit la famille logique, les signaux appliqués aux circuits doivent avoir des *temps de montée et de descente inférieurs au temps de propagation* des opérateurs élémentaires ; la définition de ces temps de propagation est l'objet du paragraphe suivant. Dans toute la suite nous considérerons donc des signaux dont les temps de montée et de descente sont nuls. Précisons que les logiciels de simulation logique adoptent toujours la même convention, malgré une terminologie parfois ambiguë (voir ci-dessous).

II.3.1 Des paramètres observables en sortie : les temps de propagation

Considérons la réponse à une impulsion d'un inverseur élémentaire figure II-6 :

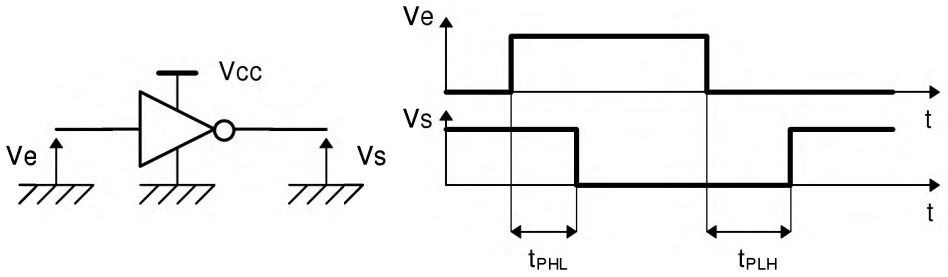


Figure II-6

Les deux temps t_{PHL} , pour temps de propagation du niveau haut vers le niveau bas, et t_{PLH} , pour temps de propagation du niveau bas vers le niveau haut, qui ne sont pas forcément égaux, caractérisent le retard entre une cause, V_e , et un effet, V_s , dû aux imperfections des transistors qui constituent l'inverseur. Ces définitions se généralisent sans peine pour toute relation de cause à effet entre une entrée et une sortie d'un circuit : retard par rapport à une horloge, retards pour commuter d'un état haute impédance à un état logique et vice versa, etc....On consultera avec profit une notice de circuit pour se familiariser avec les multiples temps de propagations spécifiés.

Ces temps sont *définis en valeur maximum*, parfois en valeurs typiques et minimum, pour une *valeur spécifiée de le capacité de charge* vue par la sortie (en général 50 pF). En effet, les temps de propagation dépendent beaucoup de cette capacité de charge, surtout dans les technologies qui utilisent des transistors à effet de champ. Dans certains cas les notices fournissent des taux d'accroissement des temps de propagation en fonction de la capacité de charge (nanosecondes par picofarad).

N.B. : Les remarques qui précèdent, à propos des capacités de charges acceptables en sortie des circuits logiques, laissent à penser aux effets pour le moins curieux que peuvent provoquer des mesures faites avec en oscilloscope dépourvu de sonde !

II.3.2 Des règles à respecter concernant les entrées

Une autre classe de paramètres dynamiques des circuits est parfois moins bien comprise : elle concerne des paramètres qui ne sont pas directement observables, mais dont le non respect peut entraîner des dysfonctionnements du circuit. Ces paramètres interviennent notamment dans les circuits séquentiels synchrones, pilotés par une horloge.

Temps de prépositionnement et de maintien

Les temps de prépositionnement (*set up time*, t_{SU}) et de maintien (*hold time*, t_H) concernent *les positions temporelles relatives de deux entrées* d'un même

circuit, par exemple la position de l'entrée D et de l'horloge d'une bascule D synchrone, qui réagit aux fronts montants de son horloge. Nous définirons ces temps sur cet exemple simple, mais ils se généralisent à toutes les entrées d'un circuit qui provoquent une action conjointe (figure II-7) :

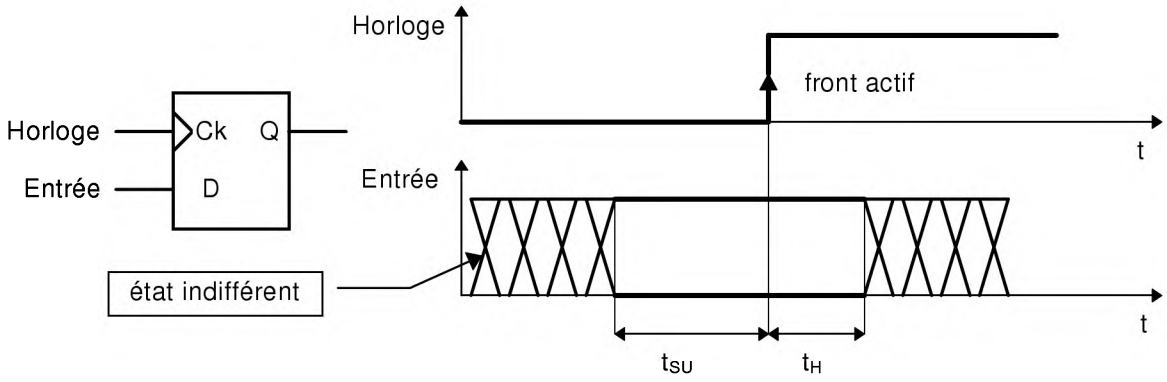


Figure II-7

Pour que la bascule interprète correctement la valeur de l'entrée, quelle que soit cette valeur, d'où les deux valeurs possibles représentées sur la figure II-7, celle-ci doit être stable *avant* la transition active d'horloge (set up) et maintenue stable *après* (hold) cette transition.

Typiquement, pour la technologie TTL-LS, ces valeurs sont : $t_{SU} = 20$ ns et $t_H = 0$. L'intérêt d'avoir une valeur nulle pour le maintien apparait dès que l'on remarque qu'en général les entrées d'un circuit synchrone sont les sorties d'un autre, la valeur à prendre en compte au moment de la transition d'horloge est alors, sans ambigüité, celle qui *précède* cette transition. Pour illustrer ceci il suffit de monter une bascule D en « diviseur par deux », un montage qui change d'état à chaque transition active de l'horloge (figure II-8) :

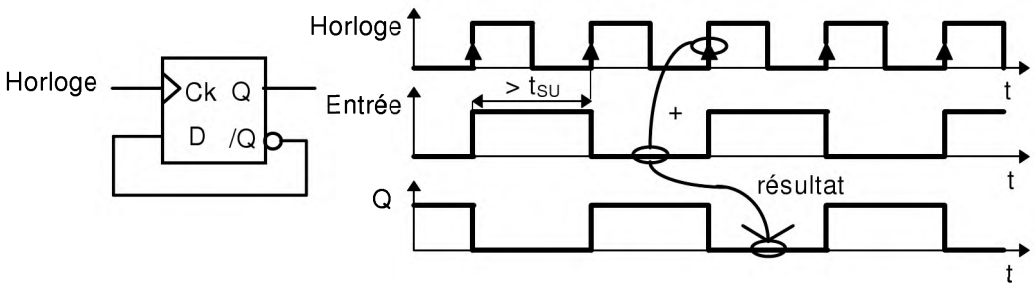


Figure II-8

Si la bascule du schéma de la figure II-8 possède un temps de maintien nul, le montage fonctionne correctement quel que soit le temps de propagation de la bascule, pourvu que la clause sur le temps de prépositionnement (qui n'est *jamais* nul) soit respectée. De plus, au niveau de l'analyse de principe, cela permet de comprendre le fonctionnement d'un système en idéalisant les caractéristiques des composants ; rien n'est plus irritant que les explications de principe qui font en permanence appel aux défauts des composants (les retards) pour « éclairer » ce fonctionnement. Par contre ces défauts doivent être pris en compte lors de l'évaluation des limites de fonctionnement d'un montage, c'est ce que nous allons explorer dans la suite.

Calcul de la fréquence maximum d'une horloge

Reprenons le schéma de la figure II-8, mais en tenant compte, cette fois, des retards dans la bascule (figure II-9), de façon à pouvoir évaluer les limites de performances de notre système :

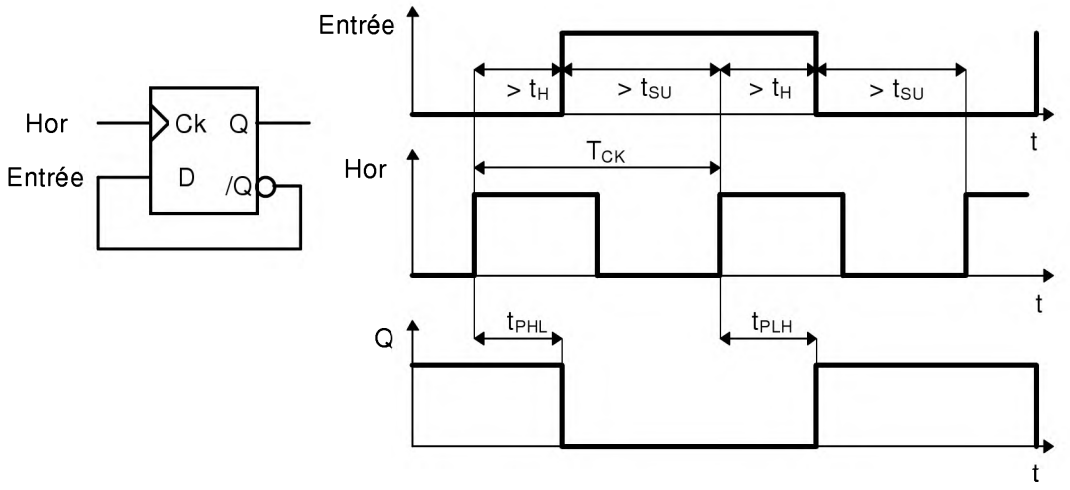


Figure II-9

Pour que le montage fonctionne correctement les paramètres des circuits doivent vérifier :

$$t_H < \min(t_{PHL}, t_{PLH})$$
$$t_{SU} < T_{CK} - \max(t_{PHL}, t_{PLH})$$

soit :

$$F_{CK} = 1/T_{CK} < 1/(t_{SU} + \max(t_{PHL}, t_{PLH}))$$

La première relation, indépendante de la fréquence de l'horloge, est toujours vérifiée pour des circuits dont le temps de maintien est nul, d'où l'intérêt de ces circuits.

La deuxième relation permet de calculer la fréquence maximum de fonctionnement du montage.

On peut étendre l'étude précédente à un cas plus général que le diviseur par deux (figure II-10) :

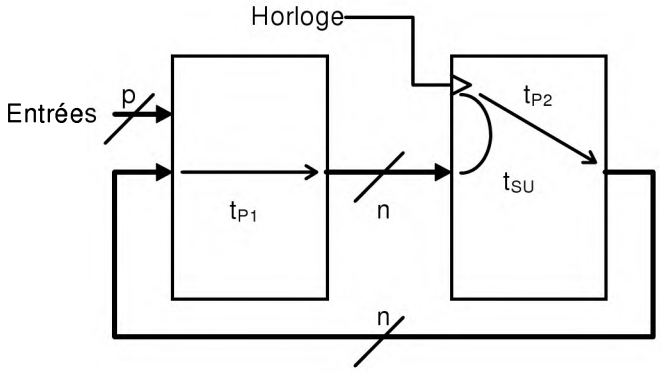


Figure II-10

Dans un tel système, qui évolue à chaque transition d'horloge en fonction de son état initial et des entrées extérieures, trois conditions doivent être respectées :

1. Les entrées extérieures doivent être correctement positionnées par rapport à l'horloge, cela peut être assuré en *resynchronisant*, au moyen d'une bascule D, toute entrée asynchrone par rapport à l'horloge locale ; nous préciserons ce point au paragraphe suivant.
2. La fréquence de l'horloge doit respecter l'inégalité :

$$F_{CK} < 1/ (t_{SU} + \max(\sum (t_{Propagation})))$$

3. Le temps de maintien doit être nul, ou au pire inférieur au plus petit des temps de propagation.

Attention : Un dysfonctionnement par violation de prépositionnement se corrige en réduisant la fréquence d'horloge ou en choisissant une technologie plus rapide, un dysfonctionnement par violation de temps de maintien, par contre, est indépendant de la fréquence de l'horloge et nécessite, en général, une refonte complète du système.

D'autres paramètres sont spécifiés qui concernent l'horloge, ou les entrées de commandes asynchrones des circuits séquentiels : largeur minimum des impulsions, fréquence maximum de fonctionnement du circuit sans rebouclage, etc...Ces paramètres conduisent, en général, à des contraintes beaucoup moins sévères que celles que nous venons d'obtenir ; il convient de se méfier des évaluations hâtives faites à partir de la lecture des notices de circuits, sans

évaluation des temps de propagation dans le schéma réel. Notons que les outils de simulation logique permettent d'extraire d'un schéma complexe les *chemins critiques* qui limitent les performances du système.

Synchronisation des entrées asynchrones d'un système synchrone

Dans un système tel que celui de la figure II-10, il est impossible d'assurer que les règles précédentes sont respectées si les changements des entrées sont asynchrones de l'horloge. Le risque est alors de voir apparaître des transitions fausses[5].

Pour éviter ce type de désagrément la méthode consiste à systématiquement *resynchroniser les entrées asynchrones* au moyen de bascules D (registre de synchronisation) :

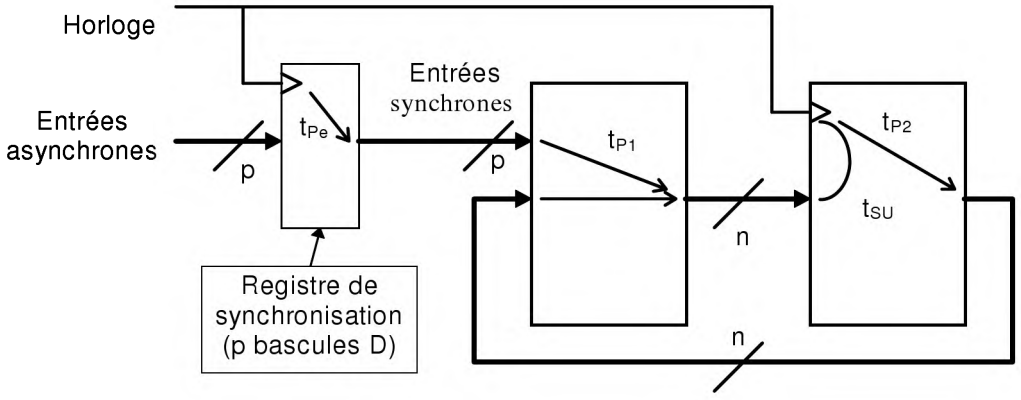


Figure II-11

Dans le schéma de principe de la figure II-11 les durées de tous les chemins sont définies, ce qui permet de contrôler le respect des temps de maintien et de prépositionnement.

Il reste cependant une interrogation : que se passe-t-il pour *une bascule* du registre de synchronisation si les temps précédents ne sont pas respectés pour elle ? A priori, tant que l'on reste dans le monde de la logique, le seul risque est de perdre une période d'horloge dans la prise en compte de l'entrée concernée. De toute façon un système synchrone évolue avec une définition temporelle qui est connue à une période d'horloge près, le problème semble donc résolu. Et pourtant... il peut arriver, extrêmement rarement (les probabilités sont inférieures à 10^{-9} pour des bascules « saines »), qu'une bascule dont l'entrée D change juste avant la transition active d'horloge (quelques nanosecondes en TTL-LS), hésite ensuite entre le niveau haut et le niveau bas, et ce pendant un temps très long à l'échelle de

[5]Par exemple un compteur qui devrait se charger à 13 prend la valeur 9 parce que la bascule de poids binaire 2 est un peu plus lente que les autres.

l'horloge. Ce phénomène, exceptionnel rappelons le, est connu sous le nom de *métastabilité*. Certains fabriquants de circuits rapides donnent des indications concernant la propension à la métastabilité de leurs produits.

La figure II-12, ci dessous, illustre la tension de sortie d'une bascule qui passe par un état métastable dans une transition L→H.

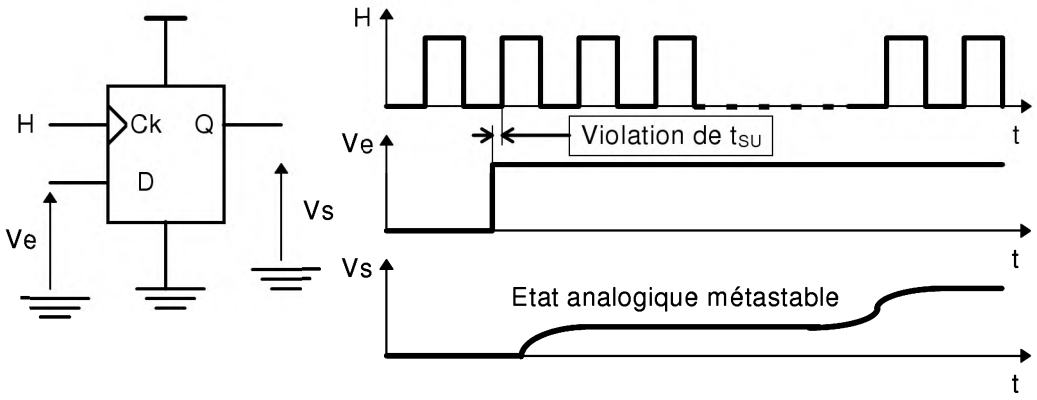


Figure II-12

Une interprétation physique de l'apparition d'un métastable peut être illustrée par les points d'équilibres d'une bille sur une surface courbe (figure II-13) :

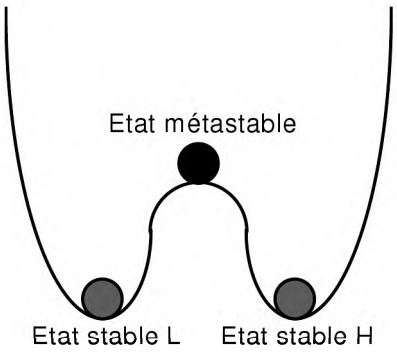


Figure II-13

Les fluctuations (agitation thermique, impulsions d'horloge) font que la bascule quittera, à un moment ou à un autre, l'état métastable, mais il est impossible de prévoir la durée de cet état. Dans des applications où le risque, même faible, d'apparition de métastables est intolérable, on peut utiliser une double

synchronisation, constituée de deux registres montés en cascade, comme dans un registre à décalage.

II.3.3 Des règles à respecter concernant les découplages

Quand la tension de sortie d'un circuit change d'état ce changement d'état s'accompagne d'un transfert de charge électrique entre le circuit et la capacité de charge, C_L, de la sortie considérée. Pendant la transition on peut considérer que la charge transférée provient entièrement de la capacité de découplage du circuit, les conducteurs d'alimentation présentent en effet une « résistance » non négligeable aux variations brusques de courant[6].

Un modèle électrique simple permet de modéliser la commutation (figure II-14) :

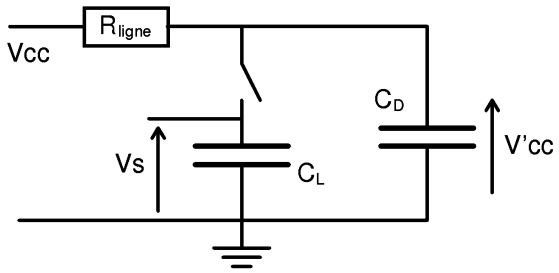


Figure II-14

Dans une transition L→H, qui correspond à une fermeture de l'interrupteur, il apparait entre V'cc et Vs un diviseur capacitif (V'cc est la tension d'alimentation du circuit). On s'impose, en général, une valeur maximum de variation ΔV'cc de tension d'alimentation. Par exemple, un circuit dont huit sorties commutent simultanément, chaque entrée étant chargée par une capacité de 50 pF, pour un écart ΔVs = 3 V entre niveaux bas et haut, et une variation ΔV'cc inférieure à 100 mV, devra être découplé par :

$$C_D \geq (\Delta Vs/\Delta V'cc) * 8C_L = 240C_L = 12 \text{ nF}.$$

D'où la valeur couramment préconisée de 10 à 100 nF par circuit, avec une capacité qui présente une faible résistance série équivalente en haute fréquence, par exemple de type céramique multicouches à diélectrique X7R ou Z5U.

[6]Résistance ou inductance ? Un premier niveau d'analyse, un peu naïf, militerait pour inductance, la théorie des lignes de propagation nous apprend qu'en dernier ressort il s'agit plutôt d'une résistance, si les lignes d'alimentations sont sans pertes.

II.4. Types de sorties

Tant qu'une application est construite comme un assemblage de circuits dont chaque sortie commande une ou des entrées d'autres circuits de même technologie, c'est à dire dans la majorité des applications, on fait appel à des sorties « standard », auxquelles se rapportent les définitions vues précédemment concernant les niveaux logiques.

Dans certains cas on est amené à utiliser des assemblages qui sont à première vue curieux : *plusieurs sorties sont connectées en parallèle*. Les circuits qui autorisent ce genre de construction font appel à des sorties non-standard, collecteur (ou drain) ouvert et/ou sorties trois états.

II.4.1 Sorties standard

Pour l'utilisateur d'un circuit, indépendamment des détails de la structure interne et tant que les spécifications de courants de sortie maximum sont respectées, une sortie standard apparaît comme une *source de tension*, que cette sortie soit au niveau haut ou au niveau bas. Un modèle électrique simplifié est alors celui de la figure II-15 : les deux interrupteurs fonctionnent en alternance, pour un niveau haut K_1 est fermé, K_2 est ouvert, la situation est inversée pour un niveau bas.

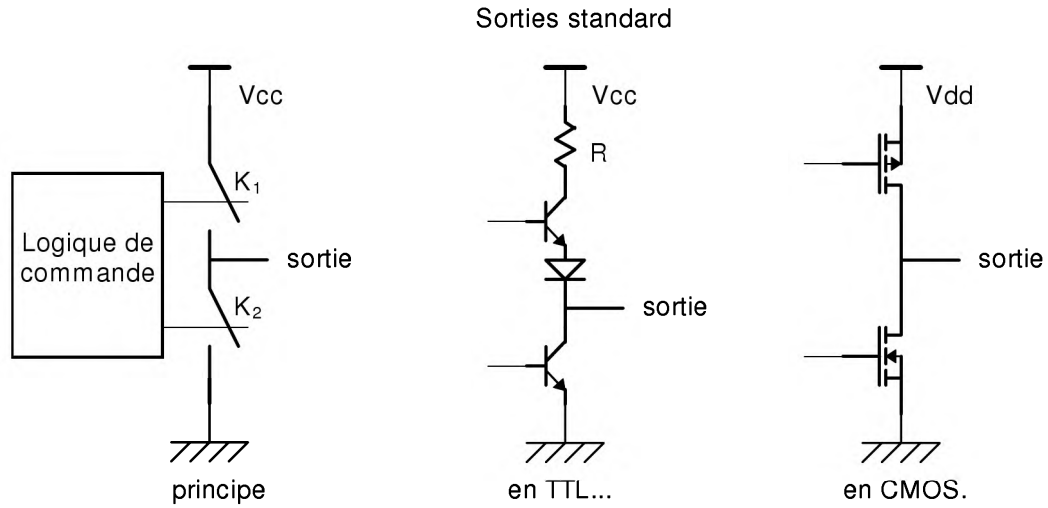


Figure II-15

Il est clair que les sorties standard ne supportent :

- ni la mise en parallèle,
- ni le court-circuit vers la masse ou vers l'alimentation.

II.4.2 Sorties collecteur (ou drain) ouvert

Une image du principe qui conduit aux sorties dites « collecteur ouvert » est celle du signal d'alarme dans un train. Le pilote du train doit être prévenu si l'une au moins des alarmes mises à la disposition des voyageurs est active. D'un point de vue logique, la fonction correspondante est un OU. Les sorties collecteur ouvert permettent de réaliser une telle fonction OU, avec un nombre arbitraire d'entrées, sans qu'il soit nécessaire de compliquer le câblage quand on augmente le nombre des entrées. Le principe est fort simple : l'interrupteur K_1 du schéma de la figure II-15 a disparu (figure II-16).

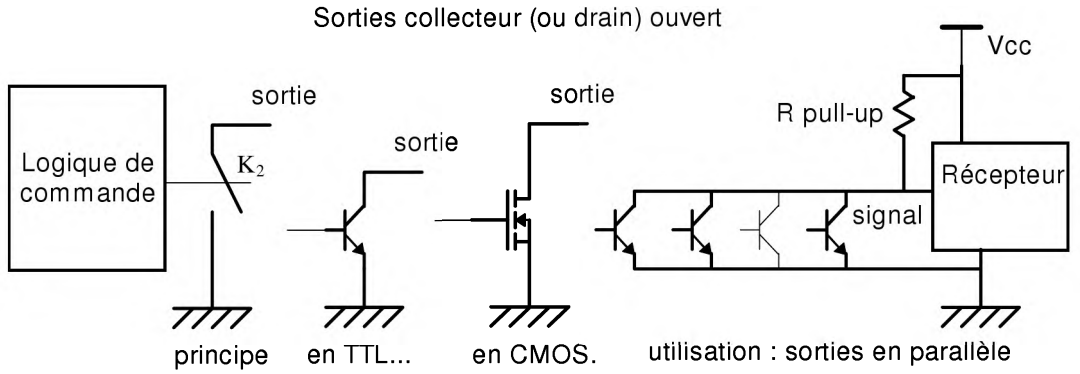


Figure II-16

On notera que dans le schéma précédent le *niveau actif est un niveau bas*, ce qui est généralement le cas dans ce type d'application où tous les circuits partagent la même masse, mais pas forcément la même alimentation. La résistance R_{pullup} (résistance de tirage), qui est unique, est située du côté de l'entrée du circuit de réception du signal.

Une autre application, plus marginale, des sorties collecteur ouvert, est l'interface entre des sous-ensembles qui travaillent avec des tensions d'alimentation différentes, dans le schéma de la figure II-16 la tension d'alimentation du récepteur, Vcc, peut être différente de la tension d'alimentation des circuits émetteurs. Cela permet, par exemple, de créer simplement une interface entre des circuits alimentés en 5 V et en 15 V.

Les sorties collecteur ouvert ne peuvent remplacer les sorties standard dans toutes les applications : leurs performances dynamiques sont nettement moins bonnes, et très dissymétriques. Alors que la transition H→L est aussi rapide que celle observée avec une sortie standard de la même technologie, le régime dynamique de la transition L→H fait intervenir la résistance de tirage, conduisant à un temps de montée qui est beaucoup plus grand qu'avec une sortie standard, et qui dépend fortement de la capacité de charge de la sortie.

II.4.3 Sorties trois états

Dans un ordinateur les chemins de données doivent permettre l'échange d'informations entre de nombreuses sources et de nombreux récepteurs : unité(s) centrale(s), mémoires, périphériques. Un câblage traditionnel, par des connexions deux à deux entre toutes les sources et tous les récepteurs possibles, conduirait rapidement à un schéma inextricable. La solution à ce problème est de réaliser les interconnexions entre les différents éléments d'un système par des *bus*. Un bus est un ensemble de conducteurs (fils électriques) qui *relient en parallèles* toutes les entrées et toutes les sorties susceptibles de recevoir ou émettre un signal d'un type donné. Dans une architecture classique on trouvera, par exemple, un bus de données, un bus d'adresses et un bus de contrôle[7].

Le protocole d'accès à un bus est simple : *à chaque instant il ne peut y avoir, au maximum, qu'un seul maître du bus* ; dit autrement, une seule sortie peut imposer, à un instant, des niveaux logiques aux conducteurs du bus. Si deux circuits (ou plus) tentent d'imposer, simultanément et indépendamment, des niveaux logiques au bus on parle de *conflit de bus*.

Dans le schéma de la figure II-17, qui illustre une connexion en bus entre une unité centrale et trois boitiers de mémoires, lors d'une opération de lecture (transfert de la mémoire vers l'unité centrale), une seule des lignes de sélection (sel i) est active, les sorties des mémoires qui ne sont pas sélectionnées sont électriquement déconnectées du bus de données, elles sont dans un état particulier dit *état haute impédance*.

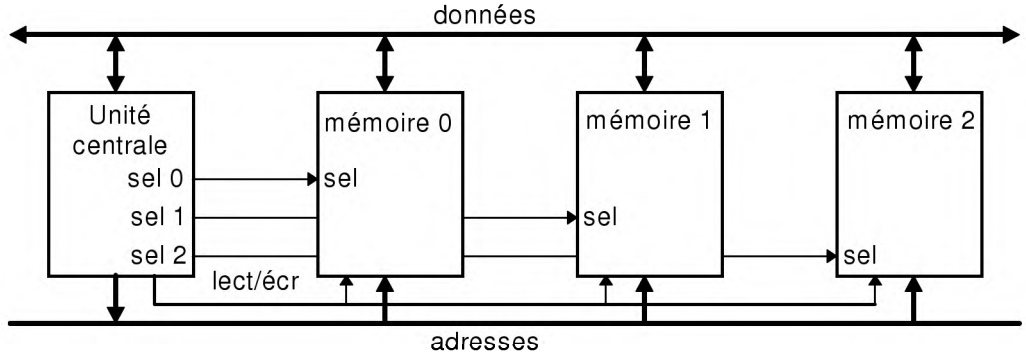


Figure II-17

Les sorties qui permettent une telle déconnexion sont appelées *sorties trois états* (*tri-state*). Physiquement, dans une sortie trois états, les deux interrupteurs de

[7]Dans le cas du bus de contrôle, le terme de bus est parfois un abus de langage, il est employé même quand les conducteurs de ce bus relient entre elles des sorties qui ne sont pas « trois état ».

la figure II-15 sont ouverts (les transistors correspondants sont bloqués). Une sortie trois états peut se trouver dans l'une des trois configurations :

- basse impédance, niveau logique bas,
- basse impédance, niveau logique haut,
- haute impédance (la broche correspondante du circuit est « en l'air »).

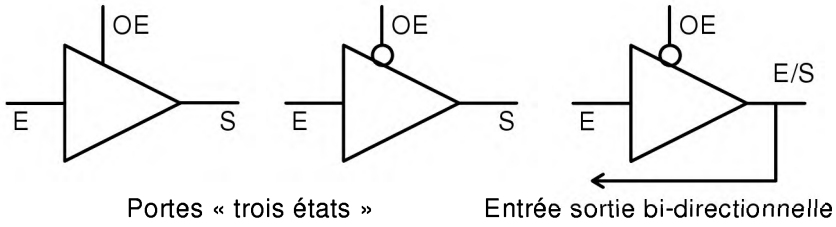


Figure II-18

Traditionnellement les symboles distinguent les commandes de connexion (commandes de mise en basse impédance) des autres entrées logiques des opérateurs par une position particulière (entrées OE de la figure II-18), le niveau actif de ces commandes, indiqué sur les symboles, correspond à l'état basse impédance.

On peut réaliser une sortie collecteur ouvert au moyen d'une sortie trois états, il suffit de maintenir un niveau logique bas, constant, et d'agir sur la commande de « tri-state ».

Exercices

Fréquence maximum de fonctionnement

Le schéma de la figure IV-6, page 94, représente un compteur décimal à trois chiffres qui utilise l'association de trois compteurs, un par décade.

- En consultant un catalogue de circuits 74LS..., estimer le fréquence maximum de fonctionnement du montage dans cette technologie.
- En quoi le schéma proposé dans le catalogue permet-il de gagner un peu en vitesse ?
- Le constructeur propose d'utiliser le circuit 74LS264, pour augmenter la vitesse maximum de fonctionnement. A partir de combien de décades l'adjonction de ce circuit auxilliaire est elle intéressante ?

Circuits programmables

Au moyen d'une notice du circuit 22V10, peu importe la technologie, expliquer pourquoi les constructeurs distinguent une fréquence maximum de fonctionnement « interne » et une fréquence maximum de fonctionnement « externe ».

Sorties collecteur ouvert (manipulation)

1. Concevoir et tester un schéma qui permet d'allumer une diode électro-luminescente à partir de deux sources connectées en parallèles. On fixera le courant dans la diode à 5 mA, ce courant étant déterminé par une résistance de "pull-up" connectée soit à 5 V, soit à 12 V (deux valeurs différentes pour la résistance !) .
2. Ces valeurs sont-elles acceptables pour un circuit du type *74LS06* ?
3. On commande l'un des circuits du montage précédent par un générateur, sortie TTL, réglé à environ 100 kHz . Observer à l'oscilloscope et interpréter la forme du signal de sortie de ces circuits, dans les deux cas de tension d'alimentation . D'où provient la différence entre les temps de montée et de descente de ce signal ?

III Opérateurs élémentaires

Une fonction numérique complexe est construite de façon hiérarchique, comme un assemblage de « boites noires », fonctions moins complexes, définies par leurs entrées, leurs sorties et les relations entre les premières et les secondes. Tout en bas de cette hiérarchie on trouve des opérateurs élémentaires, les briques ultimes au delà desquelles intervient l'électronicien qui les réalise avec des transistors ; mais au delà de cette frontière le monde du numérique s'arrête. Nous considérerons donc que les briques élémentaires de notre construction sont ces opérateurs élémentaires, et avant d'explorer cette démarche descendante qui va du général au particulier, du complexe au simple, nous tenterons de bien comprendre le fonctionnement de ces opérateurs élémentaires.

Nous utiliserons, entre autres, un langage de haut niveau, VHDL[1], pour décrire le fonctionnement de ces opérateurs élémentaires. Il est bien évident que VHDL connaît ces opérateurs comme primitives internes, et qu'il y a donc là une redondance certaine. Mais cela nous familiarisera avec ce langage qui est en passe de devenir un standard de description des systèmes numériques, et même, à terme, des systèmes analogiques.

Sauf précision contraire, nous adopterons dans la suite une convention logique[2] positive, qui associe le 0 binaire à la valeur logique FAUX et le 1 binaire à la valeur logique VRAI.

III.1. Combinatoire et séquentiel

Certains de ces opérateurs élémentaires sont la matérialisation, sous forme de circuits, ou de parties de circuits, des opérateurs bien connus de l'algèbre de BOOLE. D'autres, et notamment (mais pas uniquement) ceux que l'on appelle *opérateurs séquentiels*, sont une spécialité de l'électronique numérique, leur description n'apparaît dans aucun traité relatif à la dite algèbre.

Si l'on cherche à classer les opérateurs par familles, et c'est en classant les choses que l'on se donne les moyens d'en comprendre éventuellement le fonctionnement, le premier critère de classement concerne la façon dont évolue, au

[1]VHDL est un acronyme de Very high speed integrated circuits Hardware description language.
[2]Voir chapitre I.

cours du temps, l'état d'un opérateur, donc ses sorties[3]. On dira qu'un opérateur est *combinatoire* si les valeurs de ses sorties sont déterminées de façon univoque par les valeurs des entrées au même instant (à un temps de propagation près, bien sûr). Un opérateur est *séquentiel* si son état à un instant donné dépend des entrées, évidemment, mais aussi de ses *états passés*, du chemin qu'il a parcouru pour en arriver là, bref, un opérateur séquentiel est doué de *mémoire*.

Prenons un exemple : une serrure mécanique à combinaisons et une serrure hypothétique à digicode.

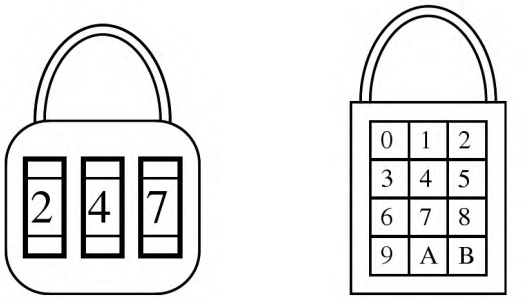


Figure III-1

L'ouverture de la première sera provoquée par une combinaison prédéfinie des trois variables d'entrée, l'ouverture de la seconde sera obtenue par la frappe d'une *séquence* prédéfinie de valeurs sur le clavier. Par exemple, si pour les deux serrures le code d'accès est 247, la première serrure s'ouvrira dès que ses variables d'entrée (les molettes) afficheront ce code, la deuxième exigera que les trois chiffres de la clé soient tapés au clavier *dans le bon ordre*. Il est clair que la deuxième serrure doit « se souvenir » du 2 quand on tape le 4, du 2 et du 4 quand on tape le 7 ; elle doit donc posséder une mémoire interne. Cette mémoire interne peut être une simple mémorisation des chiffres précédemment tapés, c'est conceptuellement la solution la plus évidente, même si ce n'est pas la plus simple à réaliser. Une autre solution consiste à doter la serrure d'une variable interne qui mémorise l'état d'avancement de la séquence, sans mémoriser les nombres eux-mêmes. On pourra alors décrire le fonctionnement du système par une sorte de diagramme, figure III-2, dans lequel chaque case représente l'état interne, auquel sont éventuellement attachées des actions (sorties), et les flèches les transitions d'un état à l'autre. A côté de chaque transition figure la condition[4] sur l'entrée qui provoque le franchissement de cette transition.

---

[3]Etat et sorties d'un système sont deux concepts différents, mais un système dont l'état n'est pas visible de l'extérieur n'a guère d'intérêt, l'évolution de l'état interne d'un objet a donc à priori une manifestation externe visible en sortie, même si cette manifestation n'est pas immédiate.

[4]Le diagramme présenté figure III-2 comporte en fait des erreurs par omission, l'utilisateur astucieux peut obtenir l'ouverture de la serrure, même s'il ne connaît pas le code d'accès. Question : comment ?

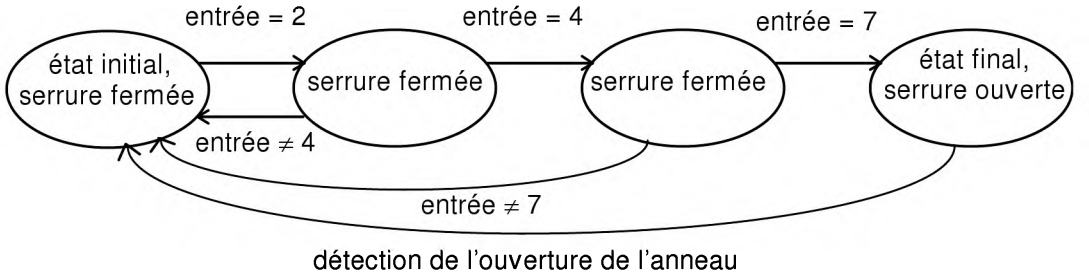


Figure III-2

La plupart des systèmes réels sont séquentiels, et beaucoup peuvent être analysés simplement par l'introduction d'une ou de plusieurs variables d'état.

L'importance des systèmes séquentiels a amené les concepteurs de circuits à imaginer des opérateurs élémentaires spécifiques, qui facilitent la tâche du concepteur. Pour illustrer le genre de questions auxquelles est amené le concepteur à répondre, reprenons l'exemple de la serrure :

- Que doit-on faire si plusieurs entrées changent simultanément ?
- Comment faire coopérer plusieurs sous-systèmes de façon prévisible ?

L'idée est progressivement venue que le travail de conception était grandement simplifié si les transitions ne pouvaient avoir lieu qu'à des temps connus, indépendamment des instants de changements des entrées, instants que, bien évidemment, le concepteur ne peut pas connaître. On rajoute alors un signal interne spécial, *l'horloge*, qui fixe la cadence de l'évolution du système. Les systèmes qui évoluent sous le contrôle d'une horloge sont appelés systèmes séquentiels *synchrones*. Les circuits synchrones ont un fonctionnement qui ne peut pas être entièrement compris dans le cadre de la logique combinatoire, ils font appel à des opérateurs élémentaires, les *bascules*, qui sont des briques de base à part entière de la logique, au même titre qu'un opérateur « ET ».

III.2. Opérateurs combinatoires

Pour chaque opérateur, nous indiquerons le ou les symboles couramment rencontrés, puis nous en donnerons une description sous forme de *table de vérité*, d'expression algébrique, de *diagramme de Venn* (héritage de la théorie élémentaire des ensembles) et sous forme d'algorithme dans le langage VHDL.

III.2.1 Des opérateurs génériques : NON, ET, OU

Les opérateurs NON (NOT), ET (AND) et OU (OR) jouent un rôle privilégié dans la mesure où ils sont « génériques », c'est à dire que toute fonction combinatoire peut être exprimée à l'aide de ces opérateurs élémentaires.

Les symboles

Bien que les symboles « rectangulaires » soient normalisés, la majorité des notices emploient les symboles curvilignes traditionnels (figure III-3). L'assemblage de symboles élémentaires dans un « schéma » porte le nom de *logigramme*, un logigramme est presque le schéma de câblage dans lequel on aurait oublié les masses et les alimentations des circuits.

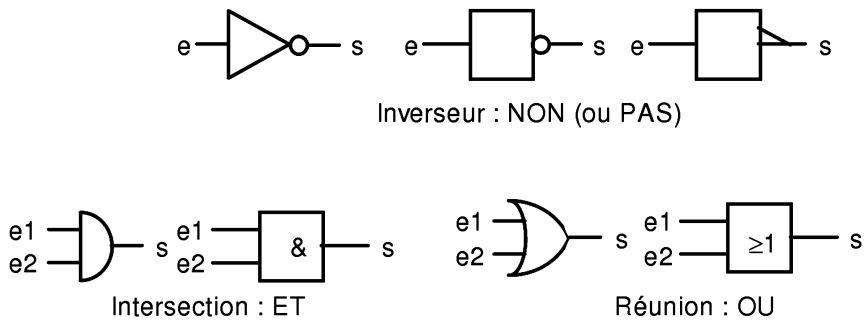


Figure III-3

Les tables de vérité

Toute fonction combinatoire peut, en dernier ressort, être décrite par une table qui énumère les valeurs prises par la (ou les) sortie(s) en fonction des valeurs des variables d'entrée. Cette méthode apparemment simple a le défaut de devenir extrêmement lourde quand le nombre de variables mises en jeu augmente. Il faut cependant y recourir quand les méthodes plus abstraites ne permettent pas de répondre à une interrogation.

Les tables peuvent être présentées sous forme linéaire ou, ce qui est souvent plus compact, sous forme de tableaux (figure III-4) :

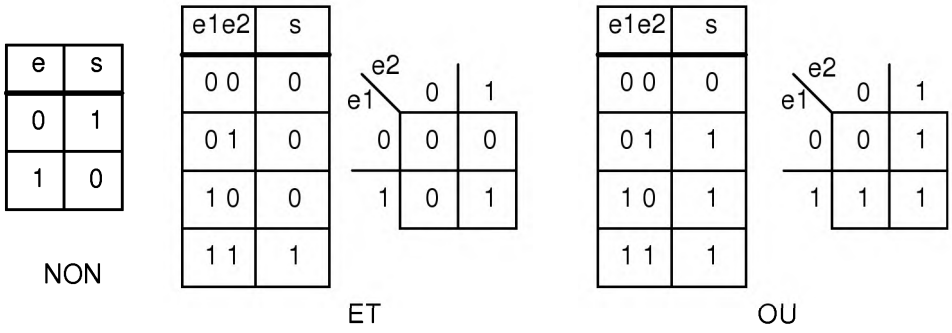


Figure III-4

Notations algébriques

Opérateur NON : $s = \bar{e}$, noté parfois, par commodité d'écriture s = /e.

Opérateur ET : $s = e1 \wedge e2$, ou s = e1 & e2 , ou encore s = e1 * e2.

Opérateur OU : $s = e1 \vee e2$, ou s = e1 | e2 , ou encore s = e1 + e2.

Les deux notations * et + pour les opérateurs ET et OU sont évidemment particulièrement ambiguës, donc inutilisables, quand on mélange dans une même description des expressions arithmétiques et des expressions logiques. Quand il n'y a pas risque de confusion, ce sont pourtant les notations les plus fréquentes.

Lors de l'écriture d'une expression qui fait intervenir les différents types d'opérateurs, les règles de priorité généralement adoptées vont, de la priorité la plus grande à la plus faible, de l'inversion (NON) au OU :

a + b * /c doit être compris a + (b * (/c))

Par contre il convient d'être extrêmement prudent quand apparaît un mélange d'opérations arithmétiques et logiques (mélange autorisé dans certains langages), la prudence élémentaire dicte, dans un tel cas, de parenthéser les expressions.

Diagrammes de Venn

Ces diagrammes illustrent bien les propriétés des expressions simples, ils soulignent l'identité de propriétés de l'algèbre de Boole et de l'algèbre des parties d'un ensemble, munie des opérations complémentation (NON), intersection (ET) et réunion (OU) (figure III-5).

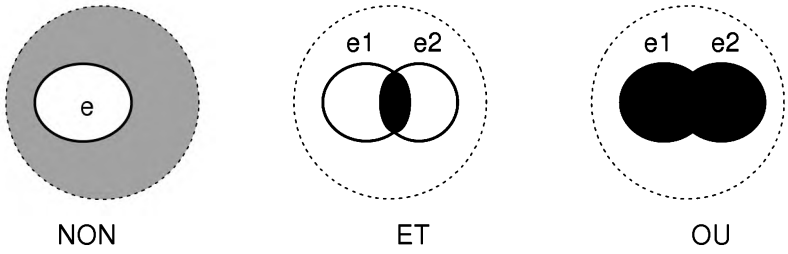


Figure III-5

Description en VHDL

Des tautologies

Les exemples de code source VHDL ci-dessous ne nous apprennent rien sur les propriétés des opérateurs concernés, ils nous montrent l'aspect d'un programme VHDL et nous rappellent que les opérations NON, ET et OU sont définies sur les objets de type BIT comme sur ceux de type BOOLEAN, avec une convention logique positive (1 ≡ TRUE, 0 ≡ FALSE).

```
-- inverseur (ceci est un commentaire)
ENTITY inverseur IS
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END inverseur;
ARCHITECTURE pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END ou;
ARCHITECTURE pleonasme OF ou IS
BEGIN
s <= e1 OR e2;
END pleonasme;
```

On notera la structure générale d'un programme et le symbole « d'affectation » particulier aux objets de nature signal (s, e, e1, e2). La déclaration ENTITY correspond au prototype d'une fonction en C, elle décrit l'interaction entre l'opérateur et le monde environnant. La partie ARCHITECTURE du programme correspond à la description interne de l'opérateur, elle décrit donc son fonctionnement. Les mots clés du langage ont été mis en majuscule, c'est une habitude de certains, pas une obligation.

Des affectations conditionnelles

Dans les programmes qui suivent on voit apparaître la notion de « haut niveau » du langage. Des expressions purement booléennes sont utilisées pour décrire le fonctionnement d'un circuit. Ici elles traduisent strictement les tables de vérité, mais permettent évidemment des constructions beaucoup plus élaborées.

```
-- inverseur
ENTITY inverseur IS
PORT ( e : IN  BIT ;
       s : OUT BIT );
END inverseur;
ARCHITECTURE logique OF inverseur IS
BEGIN
s <= '1' WHEN (e = '0') ELSE  '0';
END logique;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE logique OF et IS
BEGIN
s <= '0' WHEN (e1 = '0' OR e2 = '0') ELSE '1';
END logique;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END ou;
ARCHITECTURE logique OF ou IS
BEGIN
s <= '0' WHEN (e1 = '0' AND e2 = '0') ELSE '1';
END logique;
```

Des exemples de modèles comportementaux

Terminons cette première découverte de VHDL par deux descriptions purement comportementales des opérateurs ET et OU :

```
ENTITY et IS -- operateur ET
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE abstrait OF et IS
BEGIN
PROCESS ( e1,e2 )
BEGIN
      IF (e1 = '0' OR e2 = '0') THEN
            s <= '0';
      ELSE
            s <= '1';
      END IF;
END PROCESS;
END abstrait;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e : IN  BIT_VECTOR(0 TO 1) ; -- ATTENTION!!!
            s : OUT BIT );
END ou;
ARCHITECTURE abstrait OF ou IS
BEGIN
PROCESS ( e )
BEGIN
      CASE e IS
            WHEN "00" =>
                  s <= '0';
            WHEN OTHERS =>
                  s <= '1';
      END CASE;
END PROCESS;
```

```
END abstrait;
```

III.2.2 Un peu d'algèbre

Nous rappelons rapidement ici quelques propriétés élémentaires des opérateurs fondamentaux de la logique combinatoire. Le lecteur désireux de parfaire sa culture sur ce sujet pourra consulter un ouvrage de mathématiques, au chapitre qui traite de l'algèbre de Boole ou de l'algèbre des parties d'un ensemble[5]. Parmi ces propriétés, les plus importantes, et de loin, dans les applications, sont les lois de De Morgan : ces deux lois permettent de passer d'une convention logique à une autre, sans calcul, ou presque.

Les démonstrations concernant l'algèbre de Boole peuvent toujours se faire, en dernier recours, par un examen des tables de vérité. Cette méthode, un peu lourde, doit être envisagée si des méthodes plus astucieuses ne sont pas trouvées ; en tout état de cause, il n'est pas pensable de rester dans le doute en ce qui concerne un résultat de logique combinatoire. L'intuition permet de gagner du temps dans l'obtention d'un résultat, son absence ne justifie pas le doute.

Propriétés des opérateurs ET et OU

Associativité, commutativité

Associativité :

$$a * (b * c) = (a * b) * c \text{ , de même : } a + (b + c) = (a + b) + c .$$

Commutativité :

$$a * b = b * a \text{ , et : } a + b = b + a .$$

Un opérateur, agissant sur deux opérandes, qui est associatif et commutatif peut être généralisé à un nombre quelconque d'opérandes, sans qu'il soit nécessaire de parenthéser les expressions, par exemple :

$a + b + c + d + e$ est défini de façon univoque quel que soit l'ordre dans lequel on effectue les « calculs ».

Pratiquement cela signifie qu'il est possible de concevoir des opérateurs ET et OU à nombre arbitraire d'entrées (figure III-6) :

[5]Par exemple : J.C. BELLOC et P. SCHILLER : *Mathématiques pour l'électronique*, Masson, 1994.

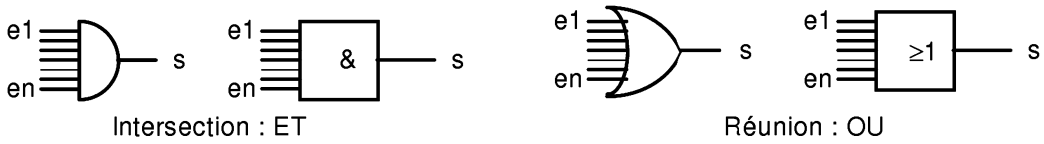


Figure III-6

On notera que la sortie d'un ET vaut 0 si l'une au moins des entrées est à 0, et que, réciproquement, la sortie d'un OU vaut 1 si l'une au moins des entrées est à 1.

Double distributivité

La ressemblance entre les propriétés des opérateurs arithmétiques, appliqués à des chiffres binaires, et celles des opérateurs logiques est grande. Une différence notable concerne la distributivité, les opérateurs ET et OU sont mutuellement distributifs l'un par rapport à l'autre :

$$a * (b + c) = (a * b) + (a * c) \ ,$$

ce qui n'étonne personne (a(b + c) = ab+ac).

$$a + (b * c) = (a + b) * (a + c) \ ,$$

qui n'a pas d'équivalent arithmétique ($a + bc \neq (a + b)\,(a + c)$).

La distributivité du OU par rapport au ET, illustrée par la deuxième des relations ci-dessus, est d'autant plus troublante que l'on adopte généralement la même convention de priorité entre les opérateurs ET et OU qu'entre leurs « analogues » arithmétiques : les opérateurs multiplicatifs sont plus prioritaires que les opérateurs additifs. Cela permet d'éviter certaines parenthèses dans l'écriture des relations un peu complexes, mais rompt l'aspect symétrique des propriétés de la réunion et de l'intersection logiques.

Pot pourri

Sans commentaire :
$a * a = a$, $a * 1 = a$ (élément neutre), $a * 0 = 0$,
$a + a = a$, $a + 1 = 1$, $a + 0 = a$ (élément neutre),
$\overline{\overline{a}} = a$, $a + \overline{a} = 1$, $a * \overline{a} = 0$, $a + (\overline{a} * b) = a + b$, $a + a * b = a$

Les lois de DE MORGAN

Les deux lois de De Morgan permettent le passage d'une fonction logique à son complément, elles sont utilisées systématiquement par les logiciels d'aide à la

synthèse de circuits logiques, pour déterminer la convention logique qui conduit à l'équation la plus simple qui rende compte d'un problème donné. Les voici :

$$\overline{(A + B)} = \overline{A} * \overline{B}$$

et :

$$\overline{(A * B)} = \overline{A} + \overline{B}$$

Bien évidemment, dans les expressions précédentes, A et B peuvent être elles-même des expressions. Sous ces formules apparemment simples se cachent parfois des calculs importants.

III.2.3 Non-ET, Non-OU

Les opérateurs NON-ET (*NAND*), et NON-OU (*NOR*), jouent un rôle particulier : ils contiennent chacun, éventuellement via les lois de De Morgan, les trois opérateurs génériques de la logique combinatoire ET, OU et NON.

- Le premier, l'opérateur NAND, est générateur de la technologie TTL (74xx00).
- Le second, l'opérateur NOR, est générateur de la technologie ECL.

Définitions et symboles

Non Et

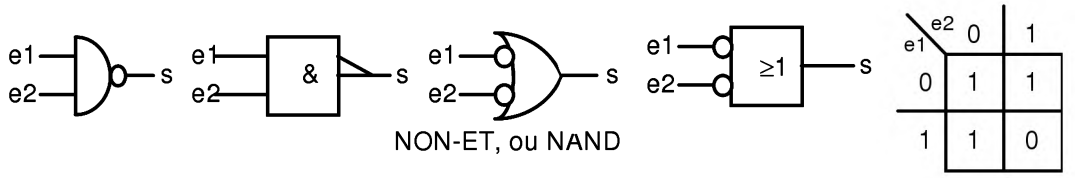


Figure III-7

Les équations de l'opérateur NAND sont, en appliquant les lois de De Morgan :

$$s = \overline{e1 * e2} = \overline{e1} + \overline{e2}$$

Non Ou

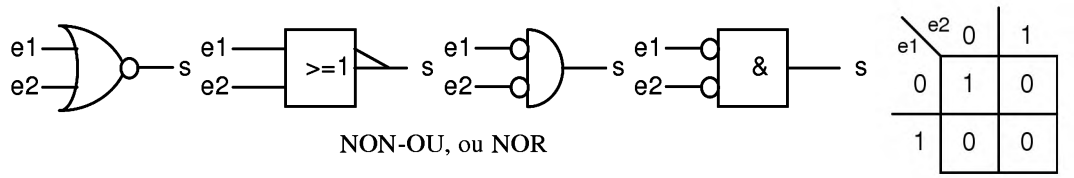


Figure III-8

Les équations de l'opérateur NOR sont, en appliquant les lois de De Morgan :

$$s = \overline{e1 + e2} = \overline{e1} * \overline{e2}$$

Les opérateurs NAND et NOR ne *sont pas* associatifs, ils ne sont donc pas généralisables, sans précaution, à un nombre quelconque d'entrées. Par contre on peut définir un opérateur qui est le complément du ET (respectivement du OU) à plusieurs entrées comme un NON-ET (respectivement NON-OU) généralisé :

$s = \overline{e1 * ... * en}$ (ou $s = \overline{e1 + ... + en}$ respectivement).

Une illustration des lois de De Morgan

A titre d'illustration des lois de De Morgan, et pour préciser ce que l'on entend par un opérateur générique, montrons qu'une expression quelconque peut être construite en n'utilisant que des opérateurs de type NAND :

$$a + b * (c + d * e) = \overline{\overline{a + b * (c + d * e)}} = \overline{\overline{a} * \overline{b * (c + d * e)}}$$
$$= \overline{\overline{a} * \overline{b * (c + d * e)}} = \overline{\overline{a} * \overline{b} * \overline{\overline{c} * \overline{d * e}}}$$

La dernière expression ne fait appel qu'à des opérateurs de type NAND, et à des inverseurs qui sont des NAND à une seule entrée.

III.2.4 Le « ou exclusif », ou somme modulo 2

Opérateur aux multiples applications, le OU EXCLUSIF (*XOR*) est sans doute l'opérateur à deux opérandes le plus riche et le moins trivial. Il trouve ses applications dans les fonctions :

- arithmétiques, additionneurs, comparateurs et compteurs ;
- de contrôle et de correction d'erreurs ;
- où l'on souhaite pouvoir programmer la convention logique ;

– de cryptage de l'information.

Après avoir donné la définition de cet opérateur, nous donnerons quelques exemples de ces applications.

Définition algébrique et symboles

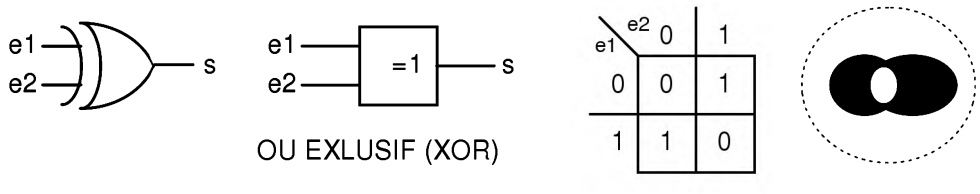


Figure III-9

On peut remarquer que cet opérateur prend la valeur 1 quand ses deux opérandes sont *différents*.

La définition algébrique du OU EXCLUSIF, à l'aide des opérateurs ET (*) et OU (+), est :

$$s = e1 \oplus e2 = \overline{e1} * e2 + e1 * \overline{e2} = (e1 + e2) * (\overline{e1} + \overline{e2})$$

D'autres symboles sont parfois rencontrés pour désigner le OU EXCLUSIF : :+:, ^ ou, plus rarement, ↑.

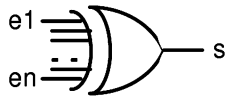
ATTENTION ! Comme on peut facilement le vérifier sur la table de vérité de la figure III-9, le OU EXCLUSIF est l'opérateur d'addition élémentaire de deux chiffres en base deux. Il est donc possible de noter cet opérateur « + », tout simplement, quand il n'y a pas de risque de confusion avec le ou inclusif. Ce type de confusion ne se pose pas dans des langages de haut niveau comme VHDL où le + est le symbole de l'addition, qui porte sur des nombres, les opérateurs logiques, dont les opérandes sont de type BIT ou BOOLEAN, sont représentés par leurs noms AND, OR et XOR.

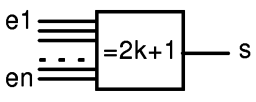
Propriétés élémentaires et applications

Algèbre

L'opérateur ou exclusif possède les propriétés de l'addition : il est associatif, commutatif et possède 0 comme élément neutre ; on peut donc le généraliser à un nombre quelconque d'opérandes d'entrée. Ainsi généralisé l'opérateur devient une *fonction qui vaut '1' quand il y a un nombre impair de '1' dans le mot d'entrée*,

d'où le symbole de la figure III-10, où la table de vérité concerne un opérateur à 4 opérandes[6].





OU EXLUSIF généralisé (XOR)

	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

Figure III-10

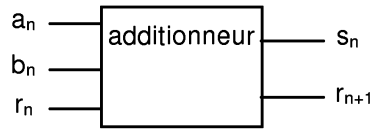
Complément du OU EXCLUSIF (XNOR) : L'opérateur OU EXCLUSIF a la particularité que pour obtenir son complément on peut, soit complémenter la sortie, soit complémenter l'une quelconque de ses entrées.

$$\overline{a \oplus b} = \bar{a} \oplus b = a \oplus \bar{b} = a * b + \bar{a} * \bar{b} = (a + \bar{b}) * (\bar{a} + b)$$

Comme opérateur à deux opérandes, ce nouvel opérateur indique *l'identité* entre les deux opérandes, d'où le nom parfois employé pour le désigner de « *coïncidence* ». Comme opérateur généralisé à un nombre quelconque d'opérandes, le complément du ou exclusif indique par un '1' qu'un nombre *pair* de ses opérandes vaut '1'.

Addition en binaire

Pour faire l'addition de deux nombres il faut savoir faire la somme de trois chiffres : les chiffres de rang n des deux opérandes et le report r_n issu de l'addition des chiffres de rang inférieur ; outre la somme il faut également générer le report r_{n+1} pour l'étage suivant. On appelle *additionneur complet* un tel opérateur (figure III-11).



$s_n = a_n \oplus b_n \oplus r_n$

$r_{n+1} = a_n * b_n + (a_n + b_n) * r_n$

Figure III-11

[6]On notera le code particulier utilisé pour numéroter lignes et colonnes, c'est un code connu sous le nom de code de GRAY, ou code binaire réfléchi ; ce code a la particularité que l'on passe d'une combinaison à la suivante en changeant la valeur d'un seul chiffre binaire. Nous aurons l'occasion d'en reparler.

La réalisation de l'addition de deux nombres peut se faire en cascadant les opérateurs précédents, on parle alors de « propagation de retenue », ou en calculant « en parallèle » toutes les retenues, au prix d'une complexité non négligeable[7], on parle alors de « retenue anticipée ».

Erreurs : tests de parités

Lors de la transmission d'informations numériques entre deux sous-ensembles, il peut se produire des erreurs. On peut tenter de détecter, voire de corriger ces erreurs en rajoutant des redondances dans le message transmis. Ces redondances consistent à rajouter au contenu du message des bits supplémentaires élaborés conformément à une règle connue à la fois par l'émetteur et le destinataire du message. La technique la plus élémentaire, qui est très utilisée dans la transmission de caractères, codés en ASCII par exemple, consiste à rajouter un bit *de parité* calculé de telle façon que chaque caractère transmis, augmenté de cet élément de contrôle, contienne un nombre pair (parité paire, *even* parity) ou impair (parité impaire, *odd* parity) d'éléments binaires à '1'. La figure III-12 illustre le principe d'un émetteur qui utilise une convention de parité paire.

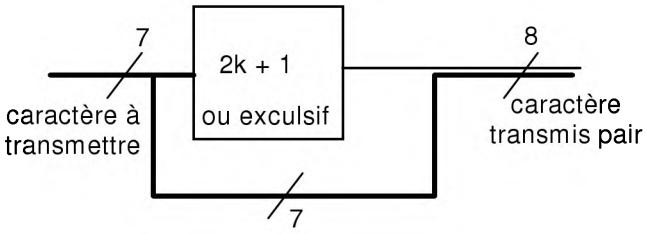


Figure III-12

Du côté du récepteur un schéma similaire permet de contrôler que la parité du caractère est bien conforme à la valeur prévue par le protocole de transmission.

Ce type de contrôle élémentaire ne permet de détecter que des erreurs simples ; si deux erreurs affectent le même caractère la parité du message reçu est correcte, le récepteur n'est alors pas pré-venu du problème.

En augmentant le nombre de clés de contrôle (les bits supplémentaires) il est possible de construire des codes autocorrecteurs, qui détectent et corrigent les erreurs de transmission, tant que leur nombre n'excède pas une valeur limite qui dépend du nombre de clés rajoutées.

Opérateur programmable

Quand on calcule une fonction combinatoire complexe, il peut être plus simple de calculer d'abord son complément et d'inverser le résultat. La plupart des circuits programmables offrent, pour ce faire, la possibilité de complémenter, ou non, la

[7]On consultera avec profit la notice d'un circuit comme le 74xx283.

sortie d'un opérateur au moyen d'un « fusible » de polarité. L'opérateur OU EXLUSIF permet de créer cette fonctionnalité, l'une de ses entrées est alors considérée comme une entrée de donnée, l'autre comme une commande de polarité, conformément au schéma de principe de la figure III-13.

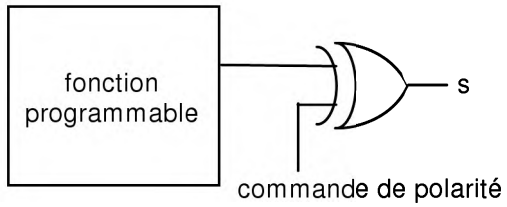


Figure III-13

Descriptions en VHDL

VHDL connaît l'opérateur XOR comme primitive ; les exemples qui suivent sont destinés à explorer, outre les propriétés de cet opérateur, des fonctionnalités du langage que nous n'avions pas abordées jusqu'ici.

Description structurelle

Ayant défini les opérateurs élémentaires ET, OU et NON comme précédemment, il est possible de les utiliser dans une construction plus complexe, comme le OU EXCLUSIF. L'exemple qui suit est, bien sûr, complètement académique, il est difficile d'imaginer plus compliqué pour réaliser un opérateur aussi simple !

```
ENTITY ouex IS  -- operateur OU exlusif
PORT ( a, b : IN  BIT ;
           s : OUT BIT );
END ouex;

use work.portelem.all ; -- rend visible le contenu de
                  -- portelem
ARCHITECTURE struct OF ouex IS
signal abar,bbar,abbar,abarb : bit;

BEGIN  -- les differents composants sont instancies ici
i1 : inverseur port map (a,abar);
i2 : inverseur port map (b,bbar);
et1 : et port map (a,bbar,abbar);
et2 : et port map (b,abar,abarb);
ou1 : ou port map (abbar,abarb,s);
```

```
END struct;
```

Pour que le programme précédent soit compris correctement, il a fallu, au préalable, créer et compiler le paquetage portelem et la description des opérateurs élémentaires qui y sont décrits comme suit :

```
package portelem is

component inverseur
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END component;

component et
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END component;

component ou
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END component;

end portelem;
-- ce qui suit est la copie de programmes déjà vus
ENTITY inverseur IS
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END inverseur;
ARCHITECTURE pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;

-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;

-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
            s : OUT BIT );
      END ou;
```

```
ARCHITECTURE pleonasme OF ou IS
BEGIN
s <= e1 OR e2;
END pleonasme;
```

L'addition élémentaire

L'opérateur OU EXCLUSIF n'est autre que l'opérateur d'addition en base deux, le programme suivant en est la conséquence directe :

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  INTEGER RANGE 0 TO 1 ;
            s : OUT INTEGER RANGE 0 TO 1 );
END ouex;
ARCHITECTURE arith of ouex is
BEGIN
s <= a + b;
END arith;
```

La comparaison

Si deux opérandes binaires sont différents le résultat de l'opérateur OU EXCLUSIF est '1' :

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  BIT ;
            s : OUT BIT );
END ouex;
ARCHITECTURE compare of ouex is
BEGIN
s <= '0' WHEN a = b ELSE '1';
END compare;
```

Indicateur de parité impaire

Nous terminerons cette découverte du OU EXCLUSIF par sa généralisation comme contrôleur de parité d'un mot d'entrée :

```
-- operateur OU exlusif generalise
ENTITY ouex IS
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
            s : OUT BIT );
END ouex;
ARCHITECTURE parite of ouex is
BEGIN
```

```
process(a)
variable parite : bit ;
begin
parite := '0';
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;
```

Rien ne s'oppose, semble-t-il, à généraliser ce programme à un mot d'entrée de, mettons, 16 bits. Là se pose un petit problème : l'optimiseur du compilateur va tenter de « réduire » les équations logiques sous-tendues par la boucle « for » pour exprimer la fonction obtenue comme somme (logique) de produits (logiques). Mais il y a 32 768 produits logiques dans un contrôleur de parité sur 16 bits (2^{15}), d'où les dangers des descriptions abstraites....

Une solution plus raisonnable, mais, il est vrai, non optimale du point de vue vitesse de calcul est[8] :

```
ENTITY ouex4 IS  -- le même que précédemment
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
            s : OUT BIT );
END ouex4;
ARCHITECTURE parite of ouex4 is
BEGIN
process(a)
variable parite : bit ;
begin
parite := '0';
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;

ENTITY ouex16 IS
PORT (e : IN BIT_VECTOR(0 TO 15);
      s : OUT BIT_VECTOR (0 TO 3);
-- force la conservation des signaux intermédiaires
```

[8]Le lecteur est instamment convié à dessiner un schéma logique en même temps qu'il lit le corps du programme.

```
      s16 : OUT BIT);  -- le résultat complet
END ouex16;

ARCHITECTURE struct OF ouex16 IS
SIGNAL inter : BIT_VECTOR(0 TO 3);
COMPONENT ouex4
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
           s : OUT BIT );
END COMPONENT;
BEGIN
par16 : for i in 0 to 3 generate
g1 : ouex4 port map (e(4*i to 4*i + 3),inter(i));
end generate;
g2 : ouex4 port map (inter,s16);
s <= inter;
END struct;
```

On notera l'intérêt des boucles « generate » pour créer des motifs répétitifs.

III.2.5 Le sélecteur, ou multiplexeur à deux entrées

Le lecteur attentif n'aura pas manqué de remarquer que beaucoup de choses, en logique combinatoire, peuvent s'exprimer par des alternatives SI... ..ALORS... ...AUTREMENT. Mais quel est donc l'opérateur élémentaire qui, en logique câblée, permet de matérialiser directement ce type de propositions ? Le *sélecteur*, ou *multiplexeur*. Nous donnons ci-dessous la description de sa version la plus simple, quand il n'y a que deux choix possibles dans l'alternative, mais il est bien sûr possible de le généraliser pour représenter des choix multiples (IF... ...THEN... ...ELSIF... ...ELSIF... ...END IF, ou, CASE... ...IS WHEN... ...WHEN... ...END CASE).

Description

Principe général

Le sélecteur est construit comme un opérateur où l'on sépare les variables d'entrée en deux groupes :

- Les entrées de *données*, qui sont en général issues d'autres fonctions ;
- L'entrée de sélection, qui est une *commande*.

Prenons un exemple. Pour faire l'addition de deux chiffres décimaux, codés en BCD, il faut commencer par faire l'addition de ces deux chiffres, sans se poser de question, comme s'il s'agissait de nombres écrits en base 2. Deux éventualités peuvent alors se produire :

1. La somme est inférieure à 10, l'opération est alors terminée.

2. La somme est supérieure ou égale à 10, ce résultat n'est alors pas correct en BCD. Il faut lui rajouter l'écart entre un nombre binaire sur 4 bits (0 à 15) et un chiffre décimal (0 à 9), soit 6.

Résumons ce qui précède sous forme d'un algorithme :

a et b sont les deux chiffres à additionner, s est le résultat.
s = a + b
si s < 10 terminé
autrement s = s + 6.

Une structure de la réalisation câblée de ce qui précède pourrait être celle de la figure III-14 :

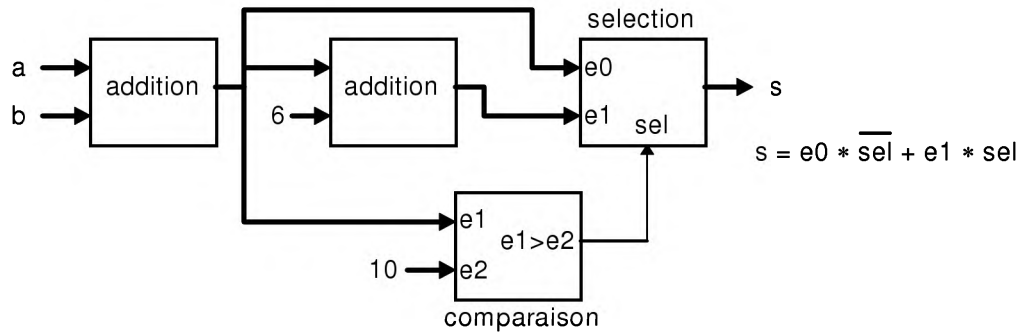


Figure III-14

Symbole et logigramme

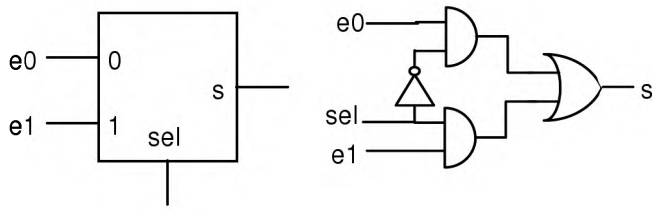


Figure III-15

Le multiplexeur élémentaire est souvent représenté par un symbole qui indique les valeurs de l'entrée de sélection à côté des entrées de données correspondantes (figure III-15).

Code source VHDL

Le multiplexeur à deux entrées est l'élément de base des descriptions dans des langages comme VHDL, nous n'en donnerons que quelques exemples :

Quelques tautologies

Nous retrouvons ici la définition même d'un multiplexeur.

```
entity sel is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end sel;

architecture pleonasme of sel is
begin
with sel select
      s <= e0 when '0',
           e1 when '1';
end pleonasme;
```

ou :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end selecteur;

architecture procif of selecteur is
begin
process (sel)
begin
if(sel = '0') then
      s <= e0 ;
else
      s <= e1;
end if;
end process;
end procif;
```

ou encore :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
               s  : out bit);
end selecteur;

architecture pleonasme of selecteur is
begin
      s <= e0 when (sel = '0') else e1;
end pleonasme;
```

Une autre façon de voir : les tableaux

VHDL connait les types structurés, la recherche d'un élément d'un tableau se traduit, en logique câblée, par un multiplexeur :

```
entity selecteur is
port ( e    : in bit_vector(0 to 1);
       sel  : in integer range 0 to 1;
       s    : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
       s <= e(sel);
end vecteur;
```

ou, en généralisant :

```
entity selecteur is
port ( e    : in bit_vector(0 to 7);
       sel  : in integer range 0 to 7;
       s    : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
       s <= e(sel);
end vecteur;
```

III.3. Opérateurs séquentiels

Nous avons déjà évoqué l'importance de la notion de *mémoire*, ce qui différencie un opérateur séquentiel d'un opérateur combinatoire réside dans la capacité du premier à « se souvenir » des événements antérieurs : une même combinaison des entrées, à un certain instant, pourra avoir des effets différents suivant les valeurs des combinaisons précédentes de ces mêmes entrées. Pour traduire cet effet de mémoire on introduit la notion d'*état interne* de l'opérateur, l'action des entrées est alors de provoquer d'éventuels changements d'état, la situation qui suit le changement de l'une d'elles dépend des valeurs des entrées et de l'état initial de l'opérateur ; si le nouvel état est différent du précédent on dit qu'il y a eu une *transition*.

Les opérateurs séquentiels peuvent être classés en deux grandes familles qui se différencient par la façon dont peuvent arriver les transitions :

1. Les opérateurs *asynchrones*, les plus anciens et les plus simples, sont de simples circuits combinatoires sur lesquels on introduit une

réaction positive. L'une des entrées de l'opérateur, soit e_r , est connectée à une sortie, soit s_r ; de plus l'opérateur $s_r = f(e_r)$ présente une caractéristique *non inverseuse*[9]. La transition d'un état à un autre est provoquée par des changements de *niveaux* d'une ou plusieurs entrées.

2. Les opérateurs *synchrones*, plus complexes, utilisent plusieurs opérateurs asynchrones pour mémoriser leur état. L'idée qui préside à la réalisation des opérateurs synchrones est de les munir d'une entrée très particulière, *l'horloge*, dont *un front*, montant ou descendant, fixe les instants où les transitions entre états sont effectuées. En général, cette horloge est le signal issu d'un générateur d'impulsions périodiques, le même pour tous les opérateurs d'un système, qui fixe une cadence de travail commune à tous les éléments du groupe. *Entre deux transitions* actives du signal d'horloge le système est *figé*, il ne peut en aucun cas changer d'état ; c'est ce temps de latence qui est mis à profit pour permettre aux transitoires de calcul des circuits, combinatoires notamment, de se terminer sans influencer de façon aléatoire le comportement du système.

Nous commencerons la description des opérateurs séquentiels par les premiers, en raison de leur simplicité, bien que les seconds soient, et de loin, les plus utilisés.

III.3.1 Les bascules asynchrones

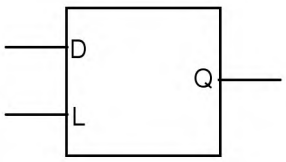
Les deux principaux types de bascules asynchrones sont la bascule D *Latch*, et la bascule R - S. La première sert simplement à mémoriser une donnée D, les entrées de la seconde doivent plutôt être comprises comme des commandes qui spécifient une action.

La bascule D « Latch », ou verrou

La bascule D-latch constitue la version la plus simple de la mémoire élémentaire.

[9]L'étude générale de la réaction nous apprend qu'une réaction positive, en continu, conduira à un système qui présente deux états stables et un état instable non oscillatoire. Une réaction négative, par contre, conduira à un état stable ou à une instabilité de type oscillatoire.

Le principe



C'est un opérateur à deux entrées et une sortie :

une entrée D de donnée,
une entrée L de commande
une sortie Q, état de l'opérateur.

Le fonctionnement est extrêmement simple : si L est au niveau haut (L = '1') la sortie prend la valeur de l'entrée D (Q = D), c'est le mode *transparent*, si L est au niveau bas (L = '0') la sortie conserve sa valeur quelle que soit celle de l'entrée D, c'est le mode *mémoire*.

Il est clair que les niveaux actifs de l'entrée de commande L peuvent être inversés.

Un exemple de réalisation

La description qui précède suggère immédiatement une réalisation qui fait appel à un multiplexeur élémentaire, ou, ce qui revient au même, à la traduction en logigramme de l'équation qui traduit cette description (figure III-16) :

$$Q = L*D + \bar{L}*Q$$

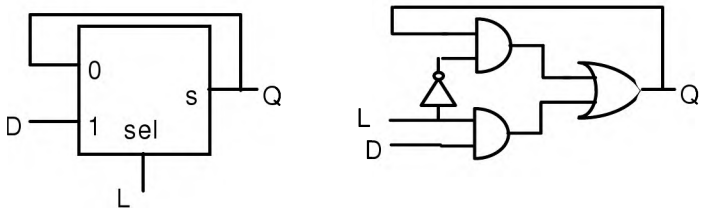


Figure III-16

L'équation précédente mérite quelque explication, quand L = '0', elle ressemble à s'y méprendre à une tautologie (Q = Q) dont on peut se demander ce qu'elle veut dire.

La figure ci-dessous (III-17) correspond au cas où L = '0', il y apparaît clairement que la bascule, en mode mémoire, se comporte comme un système bouclé. Pour analyser le type de réaction mise en jeu, une méthode simple consiste à « ouvrir la boucle », à tracer la caractéristique de transfert Q = f(E), où E est l'entrée du système en boucle ouverte, et à chercher l'intersection de cette caractéristique avec la droite d'équation Q = E.

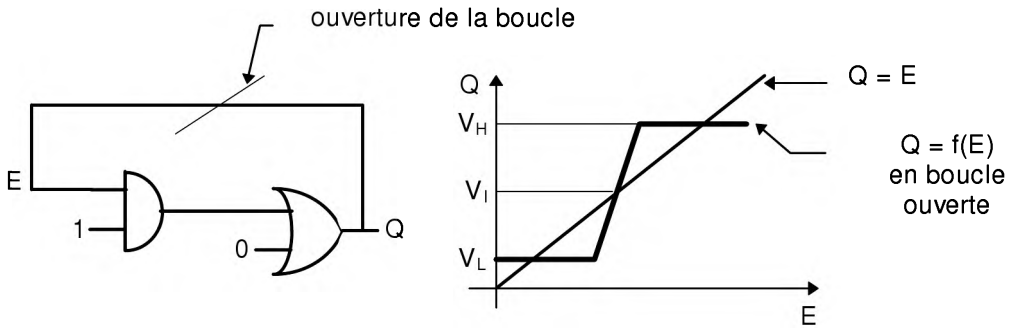


Figure III-17

Le système d'équations associé à cette construction graphique a trois solutions :

- $Q = V_L$ et $Q = V_H$, qui correspondent à deux états logiques possibles, sont des solutions stables. Si l' entrée D de la bascule place celle ci dans l'un de ces deux états, quand L = '1', le circuit conservera son état dans le mode mémoire (L = '0'), quelle que soit la valeur de D.
- $Q = V_I$ est une solution instable qui correspond à un état analogique intermédiaire. Si la bascule se trouve accidentellement dans cet état, elle évoluera vers l'un ou l'autre des états stables[10].

Quelques descriptions en VHDL

VHDL ne connait pas la fonction mémoire comme élément primitif. Une bascule asynchrone est générée soit par une description structurelle, soit par une description comportementale exhaustive, c'est à dire qui comprend la description explicite du mode mémoire, soit, *et cela constitue, pour les débutants, un piège du langage*, par une description incomplète des alternatives d'une instruction « IF ».

```
entity selecteur is  -- déjà vu précédemment
port (      a0,a1,sel : in bit;
      s     :   out bit);
end selecteur;

architecture pleonasme of selecteur is
begin
      s <= a0 when (sel = '0') else a1;
end pleonasme;

entity d_latch is
port ( D,L : in bit;
```

[10]Voir à ce sujet au paragraphe II.3.2.3 la présentation des états métastables dans les circuits synchrones, l'existence de ces états est due à cette troisième solution $Q = V_I$ dans les bascules asynchrones qui servent à réaliser une bascule synchrone.

```
      Q : out bit);
end d_latch;

architecture struct of d_latch is
 -- description structurelle
component selecteur
port ( a0,a1,sel : in bit;
       s : out bit);
end component;
signal reac : bit;
begin
Q <= reac;
s1 : selecteur port map(reac,D,L,reac);
end struct;
```

Le code qui précède n'est que la traduction naïve du premier schéma de la figure III-16. Les architectures qui suivent, qui décrivent la même entité, sont plus synthétiques :

```
architecture d_flow of d_latch is
signal reac : bit;  --  le signal de réaction
begin
Q <= reac;
reac <= D when L = '1'
          else reac; -- explicite le mode mémoire.
end d_flow;
```

Donnons enfin une forme de description qui génère le mode mémoire par omission d'une combinaison dans une alternative « IF ». La possibilité de ce type de construction présente le danger qu'elle est parfois le résultat d'un réel oubli du programmeur, et non d'une volonté de sa part :

```
architecture behav of d_latch is
signal reac : bit;
begin
Q <= reac;
process(L,D)
begin
if(L = '1') then
      reac <= D ;
end if; -- L'omission du cas où L = '0'
        -- génère le mode mémoire.
end process;
end behav;
```

Les applications

La simplicité de la constitution interne d'une bascule D-Latch en fait l'élément constitutif des mémoires statiques. Le comportement purement combinatoire de cette structure, quand elle est en mode transparent, *en interdit l'usage dans tout système qui contient des rebouclages des sorties sur les entrées*. Nous verrons dans la suite que la plupart des fonctions séquentielles font usage de tels rebouclages. Dit autrement, une bascule de ce type doit toujours être commandée en boucle ouverte, ses entrées de donnée et de commande ne doivent en aucun cas être des fonctions combinatoires de sa sortie.

A titre d'illustration, citons une application classique des bascules D-Latch dans le démultiplexage temporel d'une information : Certains microcontrôleurs utilisent les mêmes broches du circuit pour véhiculer des informations d'adresses et de données ; les mémoires, elles, attendent une adresse stable pendant toute la durée du transfert de données. Pour permettre de résoudre le conflit, le microcontrôleur fournit un indicateur, ALE (pour adress latch enable), qui indique à la périphérie que l'information disponible est une adresse. Un registre constitué de bascules D-Latch permet alors de reconstituer l'information d'adresse dont les mémoires ont besoin, conformément aux chronogrammes de la figure III-18 :

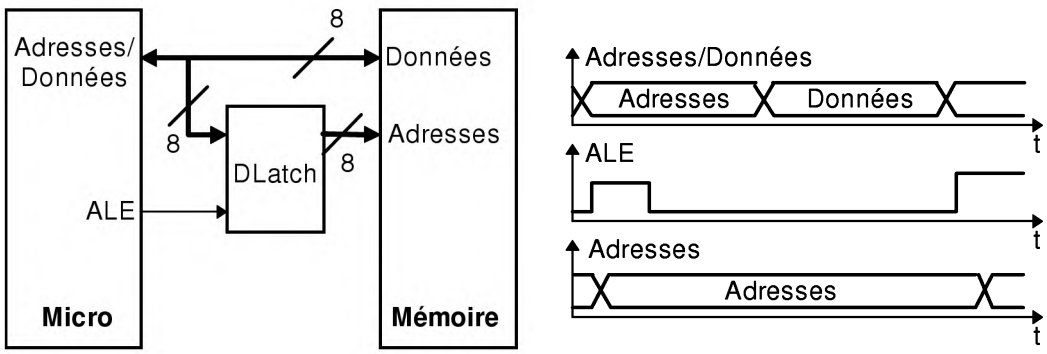
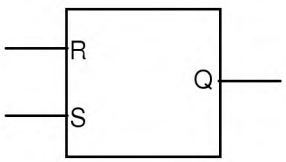


Figure III-18

La bascule R-S

Le principe

Comme la précédente, la bascule R-S est une cellule mémoire qui peut prendre deux états; le passage d'un état à l'autre est cette fois dirigé par deux entrées de commande, R et S :



l'entrée R, pour reset, provoque la mise à zéro de la sortie Q,
l'entrée S, pour set, provoque la mise à un de Q.
Quand aucune des deux commandes n'est active la bascule est en mode mémoire, quand les deux sont actives une priorité de l'une des entrées sur l'autre sera observée.

Un exemple de réalisation

Si on fixe les niveaux actifs à '1', et que l'on privilégie la mise à zéro de Q (si R et S sont simultanément actifs, R l'emporte), la traduction en équation logique de la description précédente conduit à :

$$Q = \overline{R} * S + \overline{R} * Q$$

Qui correspond au premier logigramme de la figure III-19. Le deuxième logigramme de cette figure correspond à un fonctionnement où R et S sont actifs à '0'. RS = "11" provoque le mode mémoire et la priorité est à la mise à '1' de Q si les deux commandes sont actives simultanément. On notera que, comme dans le cas de la bascule D-Latch, le mode mémoire correspond à une réaction positive sur l'ensemble du montage.

Si l'on s'impose la contrainte de *ne jamais* rendre actives les deux commandes simultanément, les sorties des deux opérateurs (nor ou nand) sont toujours complémentaires, le même schéma fournit alors Q et $\overline{Q}$[11].

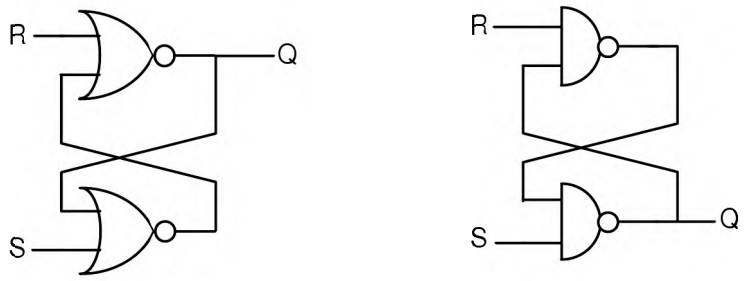


Figure III-19

[11]On rencontre parfois, et à tort, le terme de combinaison « interdite » pour la situation où les deux commandes sont actives simultanément. Les auteurs de ces lignes n'ont jamais vu de bascule exploser dans une telle situation, et ils l'ont pourtant maintes fois provoquée, bien involontairement il est vrai, en utilisant des bascules D-Edge qui sont constituées de trois bascules R-S et où cette fameuse combinaison interdite est utilisée.

La bascule RS met bien en évidence la difficulté majeure des systèmes asynchrones : si les deux commandes passent *simultanément* de l'état actif à l'état inactif (mémoire), le *résultat est imprévisible*, c'est ce qu'on appelle classiquement un *aléa*. Un phénomène analogue existe évidemment dans les bascules D-Latch, mais choque moins en raison de la dissymétrie fonctionnelle des deux entrées D et L.

Quelques descriptions en VHDL

```
entity rs is port (
R,S : in bit;
Q   : out bit);
end rs;

architecture df of rs is
signal etat : bit;
begin
q <= etat;
with R&S select
-- l'opérateur & concatène les signaux R et S
etat <=     '1' when "01",
            '0' when "10",
            etat when "00", -- mode mémoire explicite
            '0' when "11";
end df;
```

Ayant utilisé, pour décrire une bascule D-Latch, une instruction « IF » incomplète, nous donnerons ci-dessous l'exemple d'une instruction « CASE » pour générer le mode mémoire :

```
architecture bv of rs is
begin
process (R,S)
begin
   case R&S is
      when "01"         => Q <= '1';
      when "10" | "11" => Q <= '0'; -- '|' est un
                                    -- ou logique
      when others null ; -- mode mémoire : pas
                         -- d'action sur Q.
   end case;
end process;
end bv;
```

Noter que, si toutes les combinaisons de l'expression testée ne sont pas mentionnées explicitement, l'alternative « others » est obligatoire ; l'instruction « CASE » ne peut pas être incomplète. L'instruction « null » traduit l'absence

d'action, par défaut la valeur de Q est conservée, ce qui correspond bien à une cellule mémoire.

Les applications

La première application des bascules R-S réside dans les commandes de « forçage » à un ou à zéro des systèmes séquentiels, des plus simples aux plus complexes, à la mise sous tension notamment. Beaucoup de circuits offrent, à cet effet, un mode de fonctionnement de type « RS » en plus du fonctionnement normal, synchrone dans la plupart des cas. La réinitialisation matérielle (hard reset) d'un ordinateur, par exemple, correspond à une réinitialisation asynchrone (i.e. indépendante de l'horloge) de certains registres critiques du processeur, charge au programmeur d'avoir prévu la suite des événements.

Une autre application classique des bascules RS est l'élimination des rebonds des interrupteurs : Quand un interrupteur passe d'un état à un autre (ouvert ou fermé), il se produit généralement un régime transitoire oscillatoire où ces deux états se succèdent avant que l'un d'eux soit stable. Une solution classique consiste à remplacer les interrupteurs par des commutateurs, objets à deux états mécaniquement stables, F1 et F2, et un état mécaniquement instable où les deux contacts sont ouverts, deux résistances et une bascule RS, conformément au schéma de la figure III-20, dont nous laisserons l'étude détaillée à la sagacité du lecteur[12].

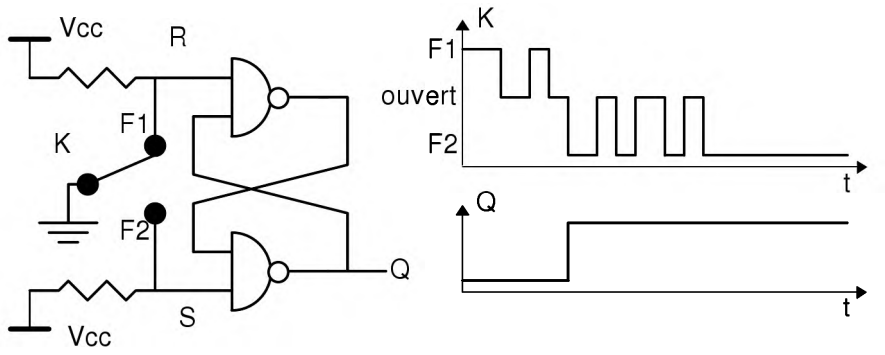


Figure III-20

III.3.2 Les bascules synchrones

Nous avons évoqué, à propos de la bascule R-S, qu'une difficulté arrive, dans les circuits séquentiels, quand plusieurs entrées susceptibles de provoquer des

[12]Guide de raisonnement : quand un commutateur passe de F1 à F2 il oscille entre F1 **ou** F2 et l'état intermédiaire où les deux contacts sont ouverts.

modifications d'états des cellules mémoire changent de niveau simultanément. Une première idée consiste à interdire de telles situations, cette politique a eu cours à une époque, mais la complexité croissante des fonctions logiques a vite mis en évidence l'inanité d'une telle solution qui compliquerait formidablement la conception du moindre circuit[13].

La solution couramment adoptée est de construire les fonctions complexes avec des bascules synchrones, qui disposent d'une entrée *réservée et unique* qui fixe les instants des éventuels changements d'états : l'horloge.

Des changements d'état bien ordonnés : l'horloge

Principe

L'idée est de dissocier les moments de définition d'une commande des instants où elle est exécutée. Pour fixer les idées, le signal d'horloge est en général issu d'un générateur d'impulsions périodiques. Une bascule peut changer d'état uniquement lors d'une *transition* (montante, par exemple) de ce signal, *l'état obtenu après* la transition active étant déterminé par la *commande présente avant* la dite transition. Le corrolaire de ce principe est évidemment que la commande doit être stable pendant un intervalle de temps non nul situé juste avant chaque transition active[14]. De très nombreux systèmes, de la vie courante, utilisent ce principe de commande à exécution différée, citons, en vrac : le rôle d'un chef d'orchestre, le pistolet qui marque le début d'une course à pieds, l'adjudant qui tente d'obtenir un quart de tour à droite d'un peloton de soldats lors d'un défilé etc.

Le fonctionnement interne des bascules synchrones relève de l'électronique analogique, et fait intervenir les temps de propagation des signaux dans les transistors, mais *ces aspects ne doivent en aucun cas être nécessaires à la compréhension du fonctionnement logique d'un système*.

La simplification apportée au concepteur par les bascules synchrones n'apparaîtra que progressivement, chaque bascule est un objet électronique plus compliqué[15], la simplification n'arrive que quand se pose la question de réaliser des fonctions complexes, qui mettent en oeuvre un grand nombre de bascules en interactions. Pour résumer on peut dire que le temps qui sépare deux fronts actifs de l'horloge (une période de celle-ci) est mis à profit par la logique classique (qui traite des niveaux) pour effectuer ses calculs, peu importe alors l'ordre dans lequel les nouvelles commandes sont élaborées, pourvu que tous les calculs soient terminés avant que n'arrive une nouvelle transition.

---

[13]L'élimination des aléas dans les fonctions séquentielles n'a, en fait, d'intérêt que pour la conception d'une bascule synchrone qui deviendra la brique élémentaire de tout l'édifice.

[14]Pour plus de précision voir le paragraphe II.3.2.

[15]Il faut deux ou trois bascules asynchrones pour réaliser une bascule synchrone, voir, par exemple, les schémas internes de la bascule D 74xx74 dans les technologies bipolaires et CMOS.

Chronogrammes et symboles

Les symboles couramment utilisés mettent en évidence le rôle particulier de l'horloge (Ck pour clock) par un triangle qui indique que cette commande agit par ses fronts, de montée ou de descente, plutôt que par des niveaux logiques, comme les entrées ordinaires (figure III-21) :

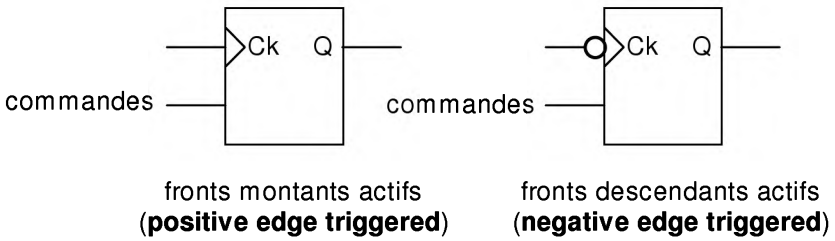


Figure III-21

Dans un ensemble synchrone le *temps devient une variable discrète*, par opposition à continue, on peut le mesurer par un nombre entier, l'unité de mesure étant la période de l'horloge. L'état d'une bascule à l'instant t est une fonction combinatoire f de la commande et de l'état à l'instant t – 1 :

$$Q(t) = f(\ Q(t-1)\ ,\ \text{commande}(t-1)\ )$$

D'où un chronogramme de principe de la figure III-22 :

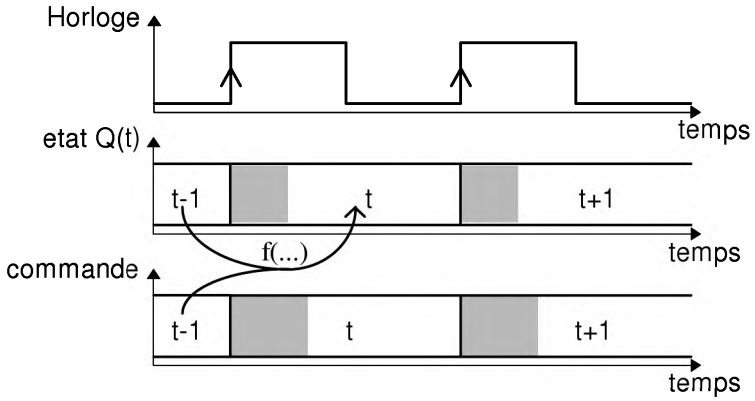


Figure III-22

Sur ces chronogrammes on a souligné par une zone grise les moments où commandes et état d'une bascule ne sont pas forcément bien déterminés, mais il s'agit là d'une simple illustration. En aucun cas le contenu de ces zones n'est nécessaire à la compréhension du principe de fonctionnement.

Diagrammes de transitions

Pour représenter de façon visuelle le fonctionnement des bascules synchrones, tout en mettant en évidence les notions centrales de la logique séquentielle que sont les états et les transitions, on utilise souvent des *diagrammes de transitions entre états* (*state transition diagram*), ou, pour abréger, diagrammes de transitions ou diagrammes d'états (figure III-23).

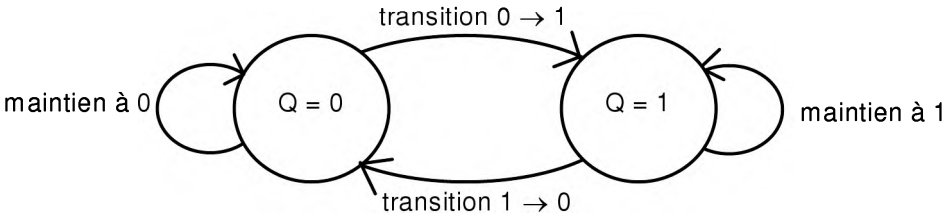


Figure III-23

Dans un tel diagramme un cercle représente un état (une bascule en a deux), une flèche une transition entre deux états (qui peut être un maintien dans l'état initial). Pour qu'une transition soit effectuée *trois* conditions doivent être vérifiées :

1. La bascule doit être dans l'état de départ,
2. il doit y avoir un front actif du signal d'horloge,
3. les entrées de commande autre que l'horloge doivent autoriser la transition.

En général le signal d'horloge est implicite, mais il ne faut bien sûr pas oublier cette condition sine qua non. La description (analyse) ou la création (synthèse) d'une bascule revient donc à préciser les équations logiques (quatre au maximum pour une bascule) qui définissent les transitions en fonction des commandes.

Une précision qui concerne VHDL

« VHDL ne contient pas le concept de signal d'horloge. Le moyen d'introduire un signal d'horloge dans vos ouvrages (designs) est d'utiliser une instruction "WAIT" dans un processus, ou d'utiliser une description structurelle »[16]!

Cela a le mérite d'être clair, il vaut mieux être prévenu.

[16]Warp2, VHDL Development system, Reference Manual, *Cypress Semiconductor*.

Pour illustrer ce qui précède on donne ci-dessous la structure d'un programme qui décrit une bascule :

```
entity basc_synchrone is port (
clock : in bit;
commande : in bit_vector( ... );
q: out bit);
end basc_synchrone;

architecture fsm of basc_synchrone is
signal etat : bit;
begin
q <= etat;
process
begin
wait until (clock = '1') -- tout est là
case etat is
      when '0' =>
  -- conditions de la transition '0'->'1'
      when '1' =>
  -- conditions de la transition '1'->'0'
end case;
end process;
end fsm;
```

D'autres constructions équivalentes existent, qui utilisent une liste de « sensibilité » dans la description du processus, nous aurons l'occasion de les examiner dans la suite. Le point important à noter est que *seuls les processus* permettent de générer à partir d'une description comportementale la synthèse d'une fonction qui utilise des bascules synchrones.

L'élément fondateur : la bascule D

Le principe

La bascule D synchrone, plus laconiquement D-edge, est la cellule mémoire fondamentale. Munie d'une entrée de donnée (en général notée « D »), et, naturellement, d'une entrée d'horloge, elle prend, à chaque transition active de l'horloge, l'état dont la valeur est celle de l'entrée de donnée. L'équation générale des bascules synchrones devient, dans ce cas, extrêmement simple :

$$Q(t) = D(t - 1) \quad \text{où t est la période d'horloge considérée.}$$

En notation abrégée, mais trompeuse car les deux membres de l'équation *ne sont pas* pris au même instant, on écrit parfois cette équation :

$$Q = D$$

Le symbole, le diagramme de transition et un exemple de chronogramme qui illustre le fonctionnement sont indiqués sur la figure III-24 :

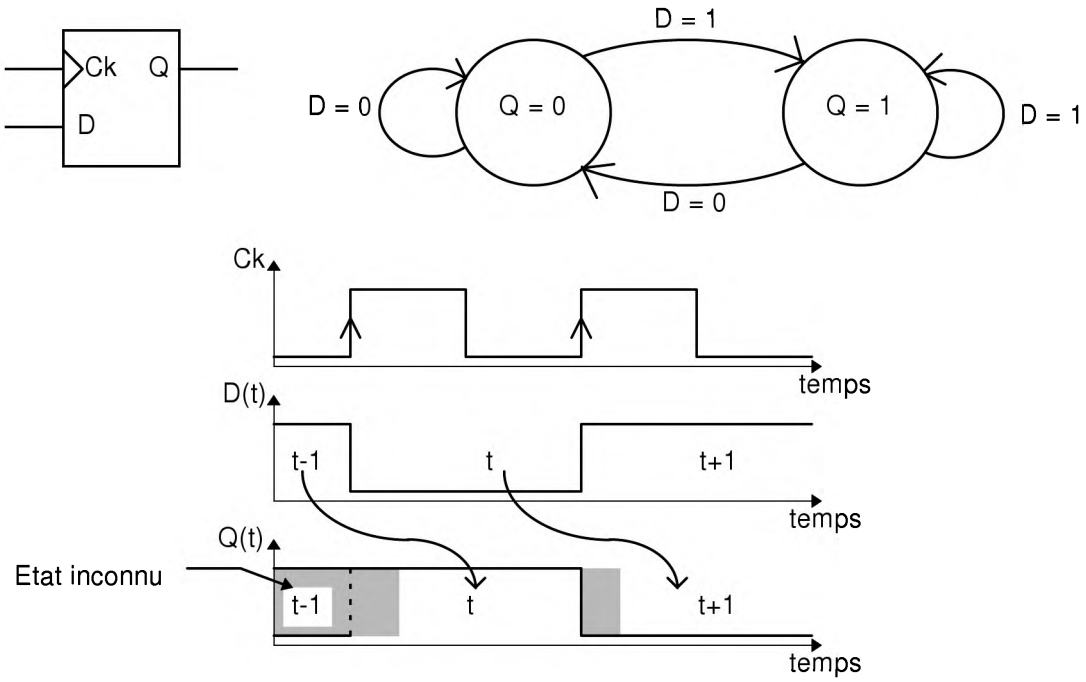


Figure III-24

On notera, dans le diagramme de transitions, le lien qui est indiqué entre les changements (ou le maintien) d'état et l'entrée de commande D. Dans la figure précédente on a pris la précaution de noter qu'à priori l'état initial de la bascule est inconnu. Ce point est important à garder en mémoire quand on se pose un problème de synthèse de système séquentiel.

Un exemple de réalisation

La réalisation interne d'une bascule D-edge *n'est en général pas* le souci du concepteur d'un ensemble logique, la bascule en question est un opérateur primitif, au même titre qu'une porte ET. Le schéma ci dessous, figure III-25, est donné à titre d'information.

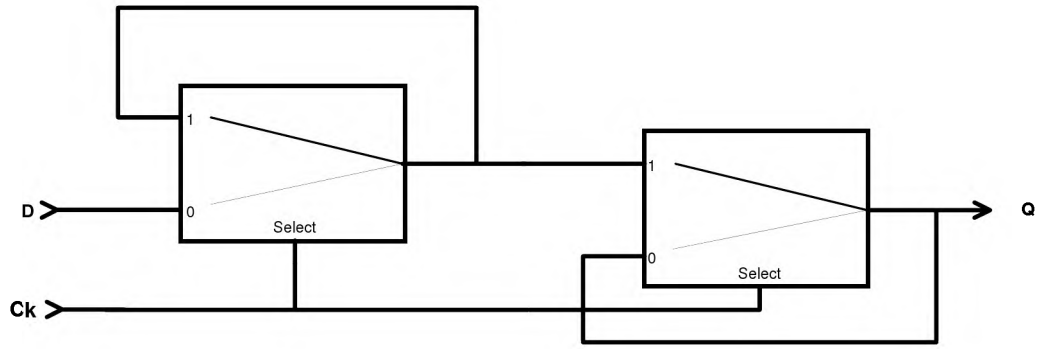


Figure III-25

Les deux multiplexeurs sont connectés en bascules D-Latch, avec des niveaux actifs inversés pour le mode mémoire. Quand l'horloge est au niveau bas la cellule de sortie est en mode mémoire, donc insensible aux variations éventuelles de son entrée, la cellule d'entrée en mode transparent. Quand l'horloge passe au niveau haut la cellule d'entrée mémorise la donnée présente, et la transfère dans la cellule de sortie. Le bon fonctionnement de l'ensemble est en fait assuré par l'existence de temps de commutation non nuls des multiplexeurs.

Cette technique de réalisation d'une bascule D, différente de celle utilisée pour la 74xx74 des familles TTL, est employée, par exemple, dans les circuits programmables (FPGAs) TPC12 (Texas Instrument).

Description en VHDL

Les deux exemples qui suivent, quoique des plus simples, sont à méditer attentivement. Ils représentent *les deux seules façons sûres* d'obtenir d'un compilateur VHDL la génération d'une bascule D synchrone générique (i.e. qui ne soit pas reconstruite au moyen de portes, ou, pire, qui ne soit pas une bascule D-Latch).

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process
begin
wait until hor = '1';
s <= d ;
end process;
end d_primitive;
```

Et une variante qui remplace l'instruction « WAIT » par une liste de sensibilité du processus et un test sur *l'existence d'une transition* du signal d'horloge *et le niveau qui suit* cette transition.

```
architecture d_primitive1 of d_edge is
begin
process(hor) -- Le process ne « réagit » qu'au signal hor.
begin
if(hor'event and hor = '1') then
-- attention ! deux conditions
      s <= d ;
end if;
end process;
end d_primitive1;
```

L'omission du facteur « hor'event » dans le test conduit certains compilateurs à générer une bascule D-Latch.

La deuxième des deux formes présentées ci-dessus est un peu plus compliquée que la première, mais plus souple. Elle permet en effet d'inclure une commande d'initialisation asynchrone, reset dans l'exemple qui suit, à une bascule D synchrone. La méthode utilisée dans cet exemple présente cependant un certain danger, les compilateurs sont toujours accompagnés d'optimiseurs qui modifient éventuellement les polarités des signaux internes, en utilisant les lois de De Morgan. L'utilisateur peut alors avoir la désagréable surprise de découvrir qu'une remise à zéro asynchrone se traduit parfois par une mise à un de la sortie attachée à la bascule visée !

```
entity d_edge is
port ( d,hor,reset : in bit;
      s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process(hor,reset)
begin
if(reset = '1') then
      s <= '0';
elsif(hor'event and hor = '1') then
      s <= d ;
end if;
end process;
end d_primitive;
```

Terminons ce tour d'horizon des descriptions en VHDL d'une bascule D par la traduction dans ce langage du schéma construit au moyen de multiplexeurs :

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_flow of d_edge is
signal sort,entre : bit;
begin
s <= sort;
entre <=  d when hor = '0' else entre;
sort  <=  entre when hor = '1' else sort;
end d_flow;
```

A n'utiliser qu'en dernier recours, quand on a épuisé toutes les « vraies » bascules D disponibles dans un circuit.

Les applications

Les bascules D sont la clé de voûte de toutes les applications séquentielles. Des quelques bascules (4 ou 8) couramment rencontrées dans les circuits standard de la famille TTL, on passe à plus de mille dans les « gros » circuits programmables.

Focalisée sur les transitions : la bascule T

Le principe

L'une des difficultés d'emploi des bascules D dans certaines applications réside dans le fait que la condition de maintien à '1' de son diagramme de transition ne doit pas être omise dans les équations obtenues pour la commande D, ce qui complique parfois notablement ces équations.

La bascule T (T pour Toggle, c'est à dire bascule) est un élément qui interprète son unique entrée de commande (en plus de l'horloge, évidemment), T, non comme une donnée à mémoriser, mais comme un ordre de changement d'état :

⇒ Si T = "actif" changer d'état à la prochaine transition de l'horloge,
⇒ si non conserver l'état initial.

D'où l'équation qui décrit son fonctionnement :

$$Q(t) = T * \overline{Q(t-1)} + \overline{T} * Q(t-1)$$

On peut éclairer l'équation précédente par le diagramme de transitions de la figure III-26, ci-dessous :

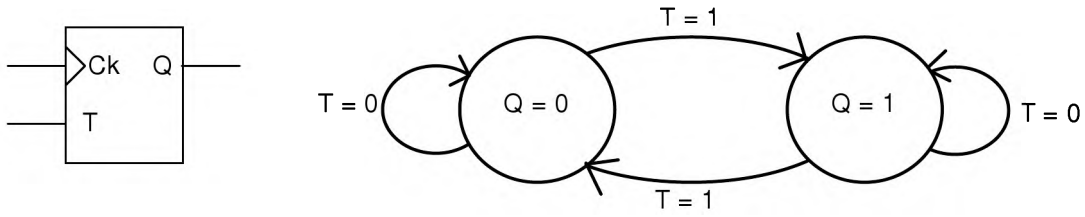


Figure III-26

Un exemple de réalisation

L'examen du diagramme de transitions de la figure III-26 nous montre que Q(t) = 1 si

Q(t-1) = '1' et T = '0',

ou

Q(t-1) = '0' et T = '1'

Ce qui nous fournit l'équation de l'entrée D d'une bascule D :

$$D = T \oplus Q$$

D'où le logigramme :

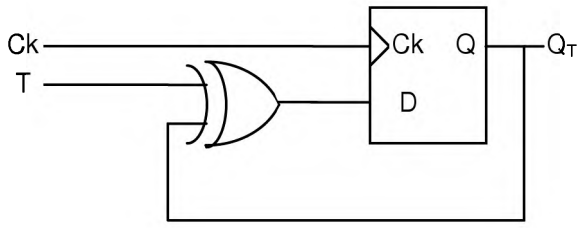


Figure III-27

Description en VHDL

La description d'une bascule T se déduit simplement du diagramme de transition :

```
entity T_edge is
```

```
port ( T,hor: in bit;
       s : out bit);
end T_edge;

architecture d_primitive of T_edge is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until hor = '1' ;
if(T = '1') then
       etat <= not etat;
end if;
end process;
end d_primitive;
```

On rappelle que de tels exemples sont fournis à titre d'illustration du fonctionnement de l'opérateur considéré, et pour familiariser le lecteur avec le langage VHDL. On n'a jamais besoin, en pratique, de décrire chaque bascule utilisée dans ce langage !

Les applications

L'un des intérêts principaux des bascules de type T est qu'elles permettent de générer de façon extrêmement simple des compteurs binaires synchrones. Un compteur binaire est une fonction séquentielle synchrone dont l'état interne est un nombre entier naturel codé en binaire dont chaque chiffre binaire est matérialisé par une bascule. A chaque transition active de l'horloge ce nombre est incrémenté de 1, quand le nombre maximum est atteint, toutes les bascules sont à 1, la séquence recommence à partir de 0. Si le nombre de bits utilisés est n, on parlera d'un compteur modulo 2^n. La simple observation d'une table des entiers naturels écrits en base 2 nous fournit la clé du problème : la bascule de rang i doit changer d'état quand toutes les bascules de rang inférieur sont à 1.

D'où un exemple de réalisation d'un compteur synchrone modulo 16 dont on peut interrompre le comptage (en = '0') :

```
ENTITY cnt16 IS
PORT (ck, en : IN BIT;
       s : OUT BIT_VECTOR (0 TO 3)
       );
END cnt16;

ARCHITECTURE structurelle OF cnt16 IS
SIGNAL etat : BIT_VECTOR(0 TO 3);
SIGNAL inter: BIT_VECTOR(1 TO 3);
COMPONENT T_edge
-- la même que dans l'exemple précédent
```

```
port ( T,hor: in bit;
       s : out bit);
END COMPONENT;

BEGIN
-- Etablir le logigramme tout en lisant le texte
s <= etat ;
inter(1) <= etat(0) and en ;
inter(2) <= etat(1) and inter(1) ;
inter(3) <= etat(2) and inter(2) ;
g0 : T_edge port map (en,ck, etat(0));
g1 : for i in 1 to 3 generate
       g2 : T_edge port map (inter(i),ck,etat(i));
end generate;
END structurelle;
```

Là encore mettons en garde le lecteur, quand on a réellement besoin d'un compteur on écrit

```
etat <= etat + 1 ;
```

c'est nettement plus simple, le compilateur générera de lui même les interconnexions nécessaires entre les bascules.

L'ancêtre vénérable : la bascules J-K

Le principe

Quelque peu tombée en désuétude, la bascule J-K a régné en maître dans le monde de la logique séquentielle des décennies 60 et 70. Elle est l' héritière directe de la bascule R-S, que l'on a débarrassé progressivement de ses difficultés asynchrones. Les premières bascules J-K n'étaient, en fait, pas de réels opérateurs synchrones, elles comportaient tout un mécanisme de mémorisation interne des commandes dans des bascules R-S (maître-esclave). Les versions actuelles sont construites au moyen d'une bascule D-edge et de logique combinatoire.

Une bascule J-K dispose de deux commandes, J et K, outre l'horloge. Son fonctionnement se décrit bien au moyen d'une table qui décrit la fonction réalisée en fonction des valeurs de la commande :

J(t-1)	K(t-1)	Fonction	Equation
0	0	Mémoire	Q(t) = Q(t-1)
0	1	Mise à zéro synchrone	Q(t) = '0'
1	0	Mise à un synchrone	Q(t) = '1'
1	1	Changement d'état	$Q(t) = \overline{Q(t-1)}$

Comme pour tout opérateur synchrone, l'état de la bascule ne change pas entre deux transitions actives du signal d'horloge. La fonction « mémoire », dans la table ci-dessus, signifie que la bascule conserve son état précédent même lors d'une transition d'horloge. Dans la construction du diagramme de transitions de la figure suivante, III-28, on a tenu compte de ce que chaque transition peut être obtenue par deux combinaisons différentes des commandes J et K, la transition '0' → '1', par exemple, peut être obtenue par mise à 1 explicite (JK = "10") ou par changement d'état (JK = "11") :

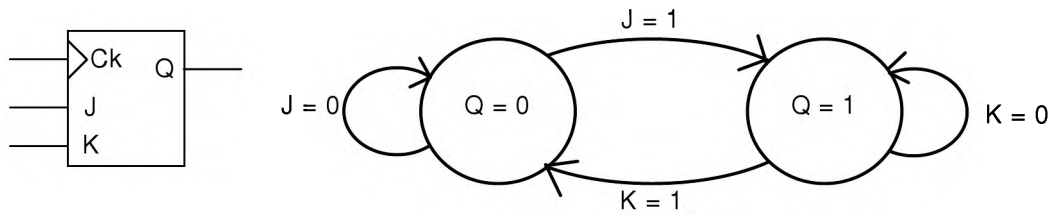


Figure III-28

On déduit aisément l'équation de la bascule J-K de son diagramme de transition :

$$Q(t) = J * \overline{Q(t-1)} + \overline{K} * Q(t-1)$$

Au lieu de raisonner sur l'équation de l'état futur, on peut décrire la bascule J-K par son équation de transition, si on introduit une variable binaire auxilliaire T_{JK} , égale à '1' si une transition doit avoir lieu, '0' autrement, on obtient :

$$T_{JK} = J * \overline{Q} + K * Q$$

Description en VHDL

La première description que nous donnerons est la simple traduction naïve de la table de vérité :

```
entity jk is port (
j,k,clock : in bit;
```

```
q: out bit);
end jk;

architecture fsm of jk is
signal etat : bit;
begin
process begin
wait until clock = '1';
      if (j = '1' and k = '1') then
            etat <= not etat;
      elsif (j = '1' and k = '0') then
            etat <= '1';
      elsif (j = '0' and k = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm;
```

Dans la version suivante, construite de la même façon, on a tenu compte des simplifications qui apparaissent dans le diagramme de transitions. Il est bien évident que le compilateur aurait, de toute façon trouvé tout seul ces simplifications.

```
architecture fsm1 of jk is
signal etat : bit;

begin
process begin
wait until clock = '1';
      IF (j = '1' and etat = '0') then
            etat <= '1';
      elsif (k = '1' and etat = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm1;
```

Les exemples précédents étaient construits à partir de la commande, ceux qui suivent le sont à partir de l'état interne de la bascule :

```
architecture fsm2 of jk is
signal etat : bit;
begin
q <= etat;
process begin
wait until clock = '1';
```

```
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end process;
end fsm2;
```

Ou, dans une variante déjà rencontrée :

```
architecture fsm3 of jk is
signal etat : bit;
begin
q <= etat;
process(clock)
begin
if(clock = '1'and clock'event) then
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end if;
end process;
end fsm3;
```

En conclusion de cette énumération précisons que les équations logiques générées par un compilateur seront les mêmes quelle que soit la forme du programme source, à savoir :

PLD Compiler Software DESIGN EQUATIONS

$$q.D = q.Q * /k + /q.Q * j$$

Ce qui est rassurant.

Les applications

Les applications des bascules J-K ne diffèrent guère de celles des autres bascules synchrones. Dans des réalisations « discrètes », qui utilisent un câblage externe et des composants standard, elles conduisent en général à des schémas plus simples que les versions réalisées avec des bascules D. Cela tient à l'existence d'un mot de commande (JK) très souple :

- Chaque transition peut être obtenue de deux façons différentes,
- le maintien dans l'état correspond à l'omission de tout terme correspondant dans les équations de J et K.

Dans les circuits programmables ou les ASICs, le problème se pose de façon un peu différente : une bascule J-K est plus compliquée qu'une bascule D, la complexité globale de la fonction réalisée peut alors être strictement équivalente dans les deux réalisations.

Où l'on apprend à passer de l'une à l'autre

Les caractéristiques communes de toutes les bascules synchrones sont :

1. Deux états possibles,
2. l'éventuel passage d'un état à l'autre a lieu lorsque survient le front actif de l'horloge,
3. les transitions sont régies par un ou des signaux de commande qui doivent être stables avant le front actif du signal d'horloge.

Elles diffèrent par les détails des signaux de commande.

Avec un type de bascule on peut, par adjonction d'une fonction combinatoire générer toutes les autres, les équations nécessaires peuvent être obtenues par identification des équations, ou par examen des diagrammes de transition.

Nous avons donné, à titre d'exemple, le logigramme d'une bascule T réalisé au moyen d'une bascule D ; le lecteur pourra, à titre d'exercice, déduire des diagrammes de transitions des différents types de bascules, les autres schémas de passage possibles :

- Réaliser une bascule J-K avec une bascule D, et réciproquement,
- réaliser une bascule J-K avec une bascule T, et réciproquement,
- réaliser une bascule D avec une bascule T.

Exercices

Opérateurs combinatoires

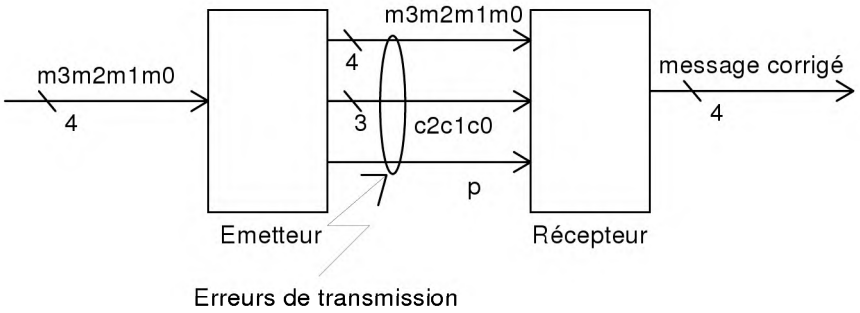
1. Le circuit 74xx86 est un « ou exclusif » en logique positive. Quelle est l'opération réalisée dans une convention logique négative ? Justifiez votre réponse par une table de vérité.

2. Donnez le logigramme de la fonction
   $f = a*b + \bar{a} * c * d + a * \bar{d}$ en n'utilisant que des opérateurs « NAND ».
   En utilisant les lois de De Morgan donner l'expression de $\bar{f}$ sous forme de somme de produits.

Notions de détection et de correction d'erreurs.

Un système de transmission comporte un émetteur et un récepteur, conformément à la figure ci-dessous. Les informations sont regroupées en messages $\mathbf{m_3m_2m_1m_0}$ de quatre bits, il peut s'agir, par exemple, de données codées en *DCB*. Au cours de la transmission des erreurs peuvent se produire. Pour tenter de détecter ces erreurs éventuelles, on rajoute au message des clés de contrôle $\mathbf{c_2c_1c_0}$ et un bit de parité générale **p**. A l'émission ces éléments de contrôle sont construits comme suit:

- L'ensemble $\mathbf{m_3m_2m_1c_2}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_2m_0c_1}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_1m_0c_0}$ contient toujours un nombre pair de bits à "1".
- L'ensemble $\mathbf{m_3m_2m_1m_0c_2c_1c_0p}$ contient toujours un nombre pair de bits à "1".



On notera qu'une erreur de transmission peut affecter n'importe quel bit, qu'il appartienne au message ou à l'un des éléments de contrôle.

a Quel opérateur permet de générer les clés $\mathbf{c_i}$ à l'émission ?

b Montrer qu'en réalité le calcul de **p** ne nécessite qu'un opérateur à trois opérandes. Préciser la nature de l'opérateur et les opérandes.

c Le récepteur recalcule la parité des ensembles définis ci-dessus. Il construit ainsi quatre variables binaires $\mathbf{e_2}$ $\mathbf{e_1}$ $\mathbf{e_0}$ et **e** qui indiquent, par un 1 logique, qu'il y a une faute de parité sur l'un de ces ensembles ; **e** indique une faute de parité générale. Proposer un schéma pour la génération des $\mathbf{e_i}$ et de **e**.

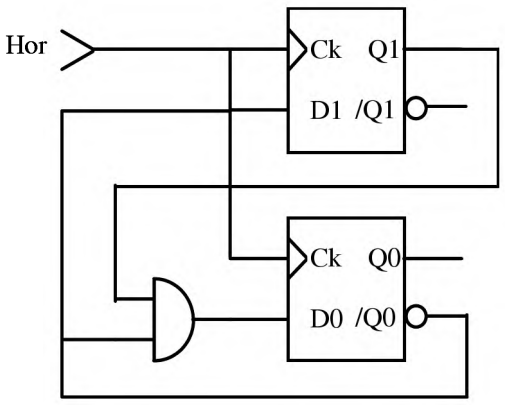
d En admettant qu'il ne peut pas y avoir plus d'une erreur de transmission, montrer que l'analyse des $\mathbf{e_i}$ permet de savoir lequel des bits $\mathbf{m_3}$ $\mathbf{m_2}$ $\mathbf{m_1}$ ou $\mathbf{m_0}$ est faux : on construira une table de vérité dont les entrées sont les $\mathbf{e_i}$ et les sorties quatre indicateurs $\mathbf{f_3}$ $\mathbf{f_2}$ $\mathbf{f_1}$ et $\mathbf{f_0}$ qui indiquent une erreur sur $\mathbf{m_3}$ $\mathbf{m_2}$ $\mathbf{m_1}$ ou $\mathbf{m_0}$ respectivement.
Proposer un schéma de correction de l'éventuelle erreur.

e Que se passe-t-il s'il y a deux erreurs de transmission ? A quoi sert le contrôle de parité générale ?

f La méthode précédente peut sembler extrêmement lourde ; montrer qu'en fait le nombre formé par les $\mathbf{e_i}$ (en base 2) peut être interprété comme le numéro du bit faux, clés comprises. En déduire que le nombre de clés nécessaires pour corriger une erreur dans un message est égal au logarithme en base 2 du nombre de bits de ce message, clés comprises. Application numérique: quelle est la longueur du message utile si on transmet des paquets de 256 bits ?

Du schéma au chronogramme.

On considère le schéma suivant :



- En admettant que les deux bascules sont initialement à 0, établir un chronogramme qui fait apparaître l'horloge et les deux sorties Q1 et Q0.
- L'état initial des bascules a-t-il une importance ?

IV Circuits : une classification

Notre propos étant essentiellement de présenter les méthodes de synthèse des fonctions logiques, dans l'optique de leur réalisation au moyen de circuits programmables ou de circuits intégrés dédiés à une application, nous n'explorerons que très partiellement les fonctions standard.

Même si l'importance des quelques centaines de fonctions proposées dans un catalogue de circuits TTL a tendance à décroître, en valeur relative, elles restent une référence « culturelle », pour le moins. Aucun concepteur de système numérique ne peut tout ignorer du décodeur 74xx138, des compteurs et registres de la famille 74xx160, des multiplexeurs de la famille 74xx151 ou des interfaces de bus de la famille des 74xx240[1].

Nous tenterons de donner ici quelques repères de classification, renvoyant le lecteur aux nombreux ouvrages qui traitent le sujet pour un complément d'information.

Dans le monde des circuits configurables par l'utilisateur, technologies et architectures sont intimement liées. Nous adopterons ici un point de vue utilisateur, sans nous préoccuper du « comment cela fonctionne », au niveau des processus de fabrication.

IV.1. Des fonctions prédéfinies : les circuits standard

Outre les opérateurs élémentaires, portes et bascules, les circuits standard peuvent être classés par modes de fonctionnement :

- combinatoires ;
- séquentiels, synchrones ou asynchrones ;
- interfaces avec des bus trois états, collecteurs ouverts.

[1]Ces derniers jouent plus un rôle électrique que logique. Ils gardent leur place dans les stocks de composants, à côté des circuits logiques proprement dits qui sont essentiellement des circuits programmables.

Parallèlement à cette première classification, on peut également établir un découpage par fonctions :

- aiguillages d'informations ;
- commandes ;
- arithmétique ;
- compteurs ;
- registres ;
- transcodeurs, encodeurs,
- et sans doute bien d'autres.

Les deux classifications sont corrélées, mais ne se recouvrent pas ; dans la figure IV-1 nous avons tenté d'illustrer les principaux de ces repères, en indiquant certains liens croisés entre les deux approches.

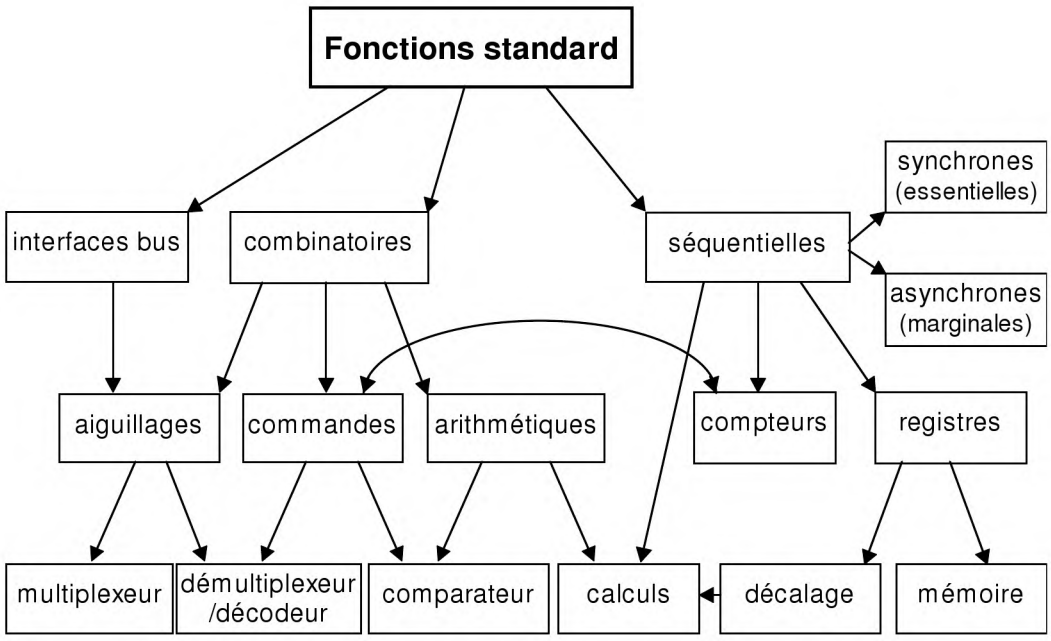


Figure IV-1

Nous passerons volontairement sous silence les fonctions arithmétiques, elles concernent des applications très spécialisée.

IV.1.1 Circuits combinatoires

Multiplexeurs et décodeurs font partie de la boite à outil standard de toute conception logique. Leur fonction première concerne l'aiguillage d'une information ; au delà de cette application directe, ces opérateurs sont génériques, ils permettent de réaliser n'importe quelle fonction combinatoire, nous les retrouverons à ce titre dans les cellules élémentaires de certains circuits programmables.

Les multiplexeurs

L'image d'un multiplexeur correspond à la sortie d'une gare de triage : par un jeu d'aiguillages on peut raccorder l'une des voies de la gare à la ligne de sortie.

Principe général

Un multiplexeur possède deux types d'entrées :
- les données d'entrée dont l'une est aiguillée vers la sortie,
- les commandes de l'aiguillage qui spécifient laquelle des entrées doit se retrouver en sortie, ou, éventuellement, permettent d'inhiber globalement le fonctionnement du circuit (figure IV-2).

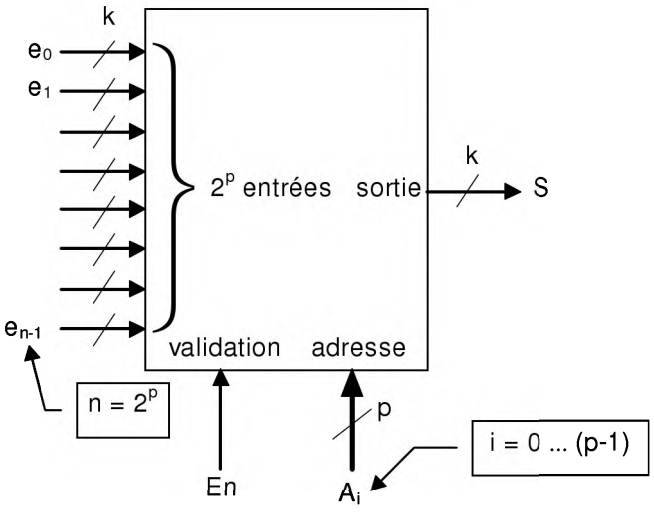


Figure IV-2

Le fonctionnement est le suivant :
- Si En est actif, mettons égal à '0', la sortie S est égale à l'entrée dont la commande d'adresse, A codée sur p bits, fournit le numéro ;

– si En est inactif, la sortie est à un niveau fixe, indépendant des entrées, mettons '1'.

De cette définition on peut tirer l'équation, pour p = 2 :

$$S = \overline{En} * (\overline{A_1} * \overline{A_0} * e_0 + \overline{A_1} * A_0 * e_1 + A_1 * \overline{A_0} * e_2 + A_1 * A_0 * e_3)$$

Chaque entrée e_i peut être une donnée binaire, ou un mot codé sur un nombre k quelconque de bits.

Exemples

Les circuits classiques, de la famille TTL, correspondent à :

74xx151 : p = 3, n = 8, k = 1
74xx153 : p = 2, n = 4, k = 2
74xx157 : p = 1, n = 2, k = 4

Les variations corrélatives de n et k sont telles que tous ces circuits tiennent dans un boitier 16 broches.

Opérateur générique

Un multiplexeur à p entrées d'adresses permet, en rajoutant éventuellement un inverseur, de générer n'importe quelle fonction de p + 1 variables d'entrée.

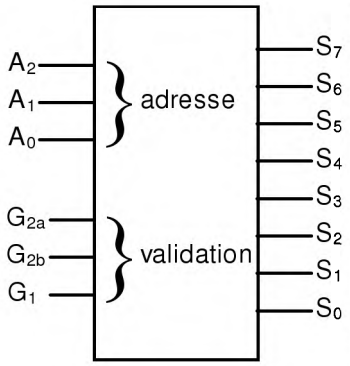
La solution est évidente : étant donné une fonction $f(x_p, x_{p-1}, \ldots x_0)$, pour une combinaison donnée des variables $x_{p-1}...x_0$, la fonction ne peut prendre que les valeurs x_p, $\overline{x_p}$, '0' ou '1'. Il suffit donc de connecter les variables $x_{p-1}...x_0$ aux entrées d'adresses d'un multiplexeur dont les entrées de données sont connectées à x_p, $\overline{x_p}$, '0' ou '1', suivant la valeur de la fonction.

Les décodeurs – démultiplexeurs

Décodeur et démultiplexeur sont deux fonctions différentes, mais obéissent aux mêmes équations. Le circuit est donc le même, mais vu sous deux aspects différents.

La fonction décodeur

Un décodeur est un opérateur à sorties multiples, dont le nombre est généralement une puissance de deux, telles que dans une convention logique donnée (le plus souvent négative), une seule d'entre elles, au maximum, soit active à un instant donné. Des entrées d'adresse permettent de sélectionner celle des sorties qui doit être active, et des entrées de validation générale permettent d'inhiber toutes les sorties. L'exemple illustré par la figure IV-3 correspond au décodeur le plus utilisé de la famille TTL : le 74xx138.



G_{2a}	G_{2b}	G_1	A_2	A_1	A_0	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
x	x	0	x	x	x	1	1	1	1	1	1	1	1
1	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
0	0	1	0	0	0	**0**	1	1	1	1	1	1	1
0	0	1	0	0	1	1	**0**	1	1	1	1	1	1
0	0	1	0	1	0	1	1	**0**	1	1	1	1	1
0	0	1	0	1	1	1	1	1	**0**	1	1	1	1
0	0	1	1	0	0	1	1	1	1	**0**	1	1	1
0	0	1	1	0	1	1	1	1	1	1	**0**	1	1
0	0	1	1	1	0	1	1	1	1	1	1	**0**	1
0	0	1	1	1	1	1	1	1	1	1	1	1	**0**

Figure IV-3

Contrairement à notre habitude, nous en reproduisons la table de vérité : la diagonale de '0', qui indique laquelle des sorties est active, est caractéristique de la fonction décodeur.

Application typique

L'application la plus fréquente des décodeurs est la commande de circuits trois états qui accèdent à un bus commun. Les différents boitiers de mémoire qui constituent la mémoire centrale d'un ordinateur en est un exemple typique. Lors de l'écriture ou de la lecture d'une information en mémoire, un seul des boitiers doit être activé, celui qui contient la cellule mémoire adressée (figure IV-4) :

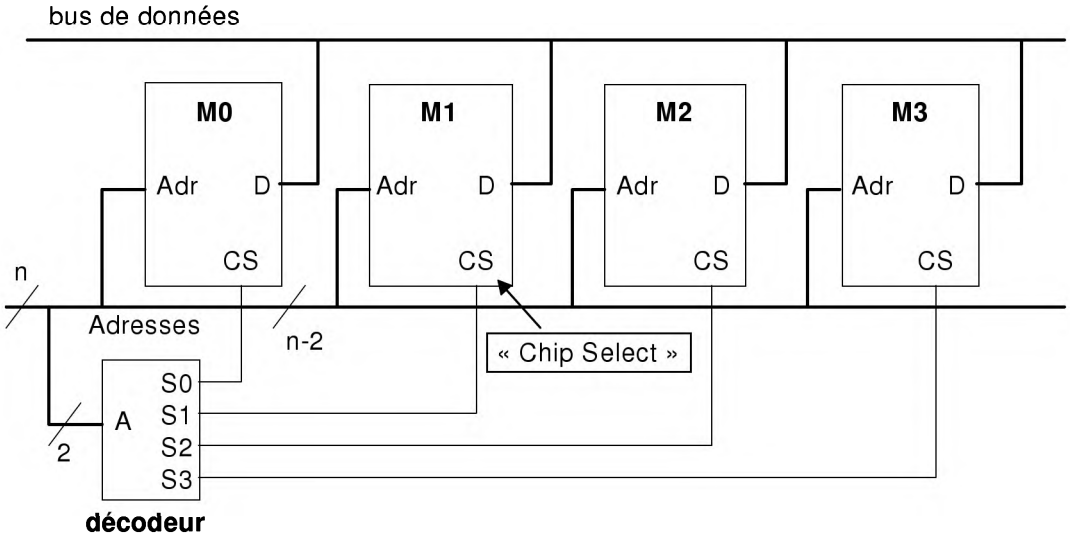


Figure IV-4

Les deux bits de poids forts de l'adresse sont utilisés pour sélectionner le boitier, les autres bits adressent en parallèle tous les circuits de la mémoire. Nous n'avons pas représenté sur cette figure des commandes globales qui fixent, par exemple, le sens de transfert des données.

Génération de fonctions

Si on exprime, en logique positive, la relation entre les entrées d'adresse et une sortie d'un décodeur, l'équation obtenue est un simple produit logique. Les sorties sont généralement actives au niveau bas, ce qui rajoute une complémentation. Par exemple en admettant que les commandes de validation générale sont actives :

$$\overline{S5} = A2 * \overline{A1} * A0$$

Pour réaliser une fonction quelconque des entrées d'adresses, considérées comme des variables quelconques, il suffit de réunir les monômes correspondants :

$$f(A2,A1,A0) = \overline{A2} * A1 * A0 + A2 * \overline{A1} * A0 + A2 * A1 * \overline{A0}$$
$$= \overline{S3 * S5 * S6}$$

Dans l'équation précédente, nous avons utilisé les lois de De Morgan pour passer d'une convention logique positive à la convention logique négative généralement utilisées dans les décodeurs.

Cette application des décodeurs est intéressante quand il est nécessaire de créer plusieurs fonctions des mêmes variables.

La fonction démultiplexeur

Le même circuit peut servir à réaliser l'opération réciproque[2] du multiplexage : le démultiplexage.

Un démultiplexeur est un opérateur qui aiguille une entrée de donnée vers une sortie dont on donne l'adresse sous forme d'un nombre codé en binaire. Pour réaliser une telle fonction avec un décodeur du type 74138, par exemple, il suffit de considérer l'une des entrées de validation comme entrée de donnée (G_{2a} ou G_{2b} si la sortie adressée doit avoir la même polarité que l'entrée).

IV.1.2 Circuits séquentiels

Nous nous contenterons, ici, de décrire deux fonctions séquentielles synchrones fondamentales : les compteurs programmables et les registres à décalage.

---

[2]Notons qu'il s'agit là d'un abus de langage, il ne s'agit pas de la fonction réciproque au sens mathématique du terme, ce qui n'aurait aucun sens. On peut, de même, considérer qu'un encodeur prioritaire, comme le 74148, est une forme de fonction réciproque du décodeur 74138 : c'est un opérateur à huit entrées qui fournit sur ses trois sorties le numéro, codé en binaire, de l'entrée active la plus prioritaire.

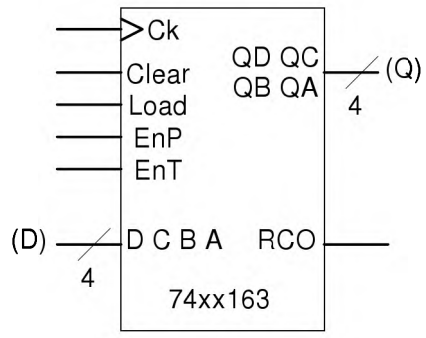
Les compteurs

La fonction de comptage élémentaire consiste simplement à passer d'une valeur entière N à la valeur N + 1 (ou N – 1 s'il s'agit d'un décompteur)[3], quand un ordre de comptage est actif. Le nombre N est codé sur n chiffres binaires. Comme n est fini (4, 8 ou 16 sont des valeurs courantes), l'ensemble des valeurs possibles pour le contenu du compteur est fini. Quand N est égal au plus grand nombre possible, N_{max}, la valeur suivante est généralement 0. Un compteur réel est donc toujours un compteur modulo $N_{max} + 1$; si $N_{max} = 2^n - 1$ il s'agit d'un compteur binaire, mais il existe des compteurs dans d'autres codes, par exemple les compteurs décimaux (code DCB).

Les compteurs programmables disposent d'entrées de commandes qui leur donnent bien d'autres fonctions que l'incrémentation d'un entier.

Principe de fonctionnement

Nous prendrons comme exemple les compteurs 4 bits TTL de la famille 74160 (74160, 74161, 74162, 74163, 74168 et 74169). Ce sont des circuits synchrones, dont l'évolution est provoquée par un front montant du signal d'horloge qui est commun à toutes les bascules du circuit. La figure IV-5 résume les caractéristiques d'un compteur 74163 :



Clear	Load	EnP	EnT	Fonction	(Q)(t) =
0	x	x	x	mise à 0	0
1	0	x	x	chargement	(D)(t–1)
1	1	0	x	mémoire	(Q)(t–1)
1	1	x	0	mémoire	(Q)(t–1)
1	1	1	1	comptage	(Q)(t–1)+1

RCO = QD * QC * QB * QA * EnT

Figure IV-5

A chaque front montant d'horloge l'action précisée par le tableau a lieu. Par exemple : en mode chargement, des données présentes sur les entrées (D), notation globale pour les quatre données D, C, B et A, sont transférées dans les quatre bascules du compteur, QD, QC, QB et QA, notées collectivement (Q).

La sortie RCO (*Ripple carry output*) est active si le compteur a atteint sa valeur N_{max} et s'il est autorisé à compter ; elle annonce le début d'un nouveau cycle pour la période d'horloge suivante. Cette sortie sert à la mise en cascade de

[3]Le symbole + représente ici l'opération d'addition.

plusieurs circuits du même type, de façon à réaliser un compteur sur plus de quatre chiffres binaires.

Le tableau ci-dessous résume les principales caractéristiques des différents membres de cette famille de compteurs :

Type	Caractéristiques particulières
74160	Décimal, remise à zéro asynchrone, chargement synchrone.
74161	Binaire, remise à zéro asynchrone, chargement synchrone.
74162	Décimal, remise à zéro synchrone, chargement synchrone.
74163	Binaire, remise à zéro synchrone, chargement synchrone.
74168	Décimal, compteur décompteur, pas de RAZ, chargement synchrone.
74169	Binaire, compteur décompteur, pas de RAZ, chargement synchrone.

Nous verrons un exemple d'application de ces compteurs au chapitre suivant.

Compteurs à plusieurs chiffres

L'association de plusieurs compteurs du même type ne nécessite, en général, aucun autre circuit que les compteurs eux-mêmes. Il suffit de chaîner les sorties RCO sur les entrées EnT, en allant des poids faibles vers les poids forts, pour qu'un étage ne s'incrémente que quand *tous* les étages précédents recommencent un nouveau cycle (figure IV-6) :

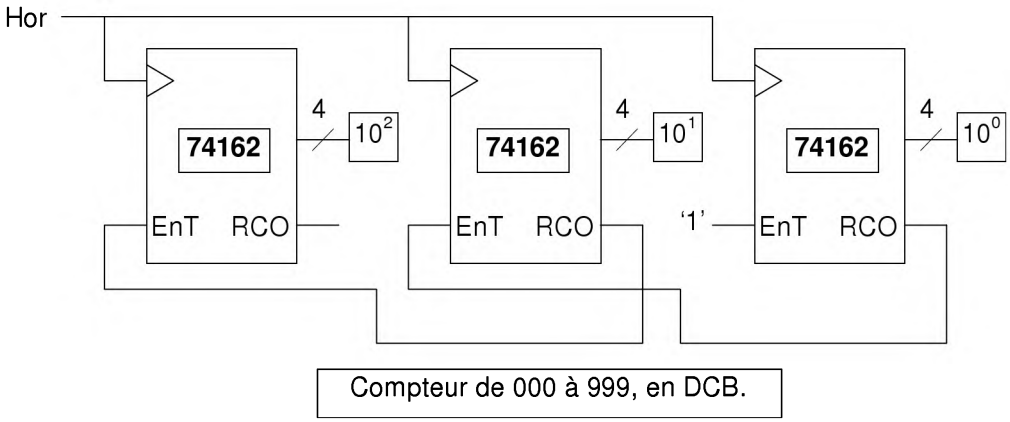


Figure IV-6

Sur le schéma de la figure IV-6, nous n'avons pas représenté les autres commandes des compteurs. Elles doivent être configurées de sorte que les trois décades soient en mode de comptage.

Dans le schéma précédent la retenue se *propage* de circuit en circuit, des poids faibles vers les poids forts. Ce mécanisme provoque un cumul des temps de propagation : pour que l'autorisation de comptage du dernier étage (poids le plus fort) soit stable, il faut attendre que toutes les retenues aient été calculées par les étages précédents. Pour des applications où la limitation correspondante de la fréquence d'horloge serait inacceptable, les retenues doivent être calculées en parallèle. Il existe des circuits spécialisés destinés à ce type d'applications, comme le 74xx264, connus sous le nom (trompeur) de générateurs de *retenue anticipée* (ces circuits n'anticipent évidemment pas, ils évitent le cumul des retards).

Les registres à décalage

Un registre à décalage élémentaire est organisé de telle façon que l'entrée d'une bascule est connectée à la sortie de l'une de ses voisines (figure IV-7) :

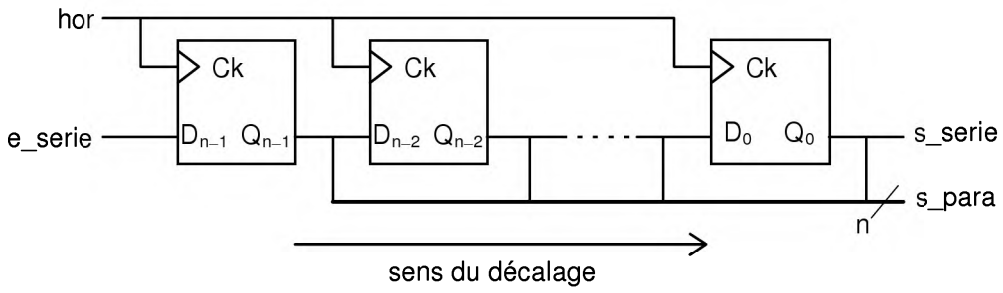


Figure IV-7

Suivant le sens du décalage, lié à la façon dont on dessine le schéma, on parle de décalage à gauche ou à droite.

Si le contenu du registre représente un nombre codé en binaire, nous retrouvons que la fonction décalage est intimement liée aux opérations de multiplication et de division par deux (voir chapitre I).

Registres universels

De même qu'un compteur offre généralement bien d'autres possibilités que le comptage, les registres à décalage universels offrent, grâce à un mot de commande, des fonctions supplémentaires :

- mémoire (le registre conserve son état initial),
- chargement parallèle,
- entrées sorties parallèles trois états,
- décalages à gauche et à droite (74xx323, par exemple),
- décalage arithmétique.

Précisons la dernière de ces fonctions : dans la division par deux d'un nombre entier signé, codé en complément à deux, le signe du nombre doit être conservé. Certains registres (74xx322, par exemple) offrent la possibilité de réaliser cette opération en interne, lors d'un décalage arithmétique l'entrée de la bascule de rang n – 1 est connectée à sa propre sortie, de sorte qu'au cours du décalage le signe du nombre soit conservé et propagé vers les poids faibles, comme il se doit.

Applications

Outre les opérations arithmétiques, les applications principales des registres à décalages concernent les transmissions d'informations binaires en *série*, c'est à dire élément binaire par élément binaire :

- liaisons séries entre ordinateurs,
- transmissions numériques par radio,
- lecture ou enregistrement de mémoires magnétiques, disques ou bandes,
- etc.

IV.1.3 Circuits d'interface

Alors que la plupart des fonctions standard apparaissent actuellement au concepteur plus comme des éléments de librairie (dans un système de CAO) que comme des composants réels à implanter sur une carte, les circuits d'interface (*bus drivers, line drivers* et *buffers*) restent des composants véritables très utilisés.

Leur fonction logique est généralement triviale : ils relient entre eux les signaux de deux systèmes, sans réaliser aucune opération, sauf parfois une simple inversion. Leur particularité réside dans leurs caractéristiques électriques qui sont adaptées à leur destination : commander de nombreuses entrées, éventuellement à travers des connexions de grande longueur et recevoir des signaux éventuellement entachés de parasites. Les caractéristiques électriques qui les distinguent des opérateurs traditionnels sont les suivantes :

- Leur sortance est plus élevée que celle des opérateurs classiques de la même famille technologique (typiquement trois fois plus).
- Ils sont capables de fournir des « pics » de courant de forte amplitude lors des commutations (au moins 30 mA, typiquement 60mA, pour un 74ALS245, par exemple).
- Leur résistance équivalente de sortie est contrôlée (quelques dizaines d'ohms), de façon à limiter les phénomènes de réflexions multiples sur les extrémités des lignes d'interconnexion.
- Les sorties trois états se mettent en haute impédance quand l'alimentation du circuit est coupée. Cette particularité autorise le raccordement d'une cartes à un bus sans éteindre l'alimentation de tout le système.
- Certains de ces circuits (74xx244, par exemple) disposent d'un étage d'entrée qui présente un hystérésis (*trigger de Schmitt*). Cette propriété leur permet d'accepter en entrée des signaux qui varient lentement sans qu'il y ait cependant d'ambigüité au moment de la commutation.

IV.2. Des fonctions définies par l'utilisateur

Deux grandes catégories de circuits offrent à l'utilisateur la possibilité de créer ses propres fonctions : les circuits programmables (*PLDs* et *FPGAs*) et les circuits intégrés spécifiques d'une application (*ASICs*)[4]. Les premiers sont programmables par le concepteur du système ; les seconds nécessitent l'intervention d'un fabricant de circuits (le *fondeur*).

Les *circuits programmables* : ils offrent, au prix d'une densité d'intégration plus faible (quelques dizaines de milliers de portes, au maximum, en 1995), une grande souplesse d'utilisation, un délai de mise en oeuvre très faible pour des petites séries et, pour certains d'entre eux, la possibilité de reprogrammer le circuit. Cette possibilité de reprogrammation peut aller jusqu'à la programmation *in situ*, c'est à dire sans retirer le circuit de la carte sur laquelle il est câblé. Cette technique ouvre la voie à la réalisation d'ensembles logiques reconfigurables suivant le contexte (cartes graphiques de micro-ordinateurs, par exemple) : le système hôte – un ordinateur le plus souvent – peut modifier la fonction d'un circuit en lui transmettant les nouvelles « équations » à réaliser.

Les *circuits spécifiques* : quant à eux, ils offrent une densité d'intégration très grande (jusqu'à quelques centaines de milliers de portes en 1995), au prix d'un délai initial de fabrication non négligeable et de l'impossibilité de modifier la fonction des circuits réalisés. Compte tenu du coût que représente la mise en fabrication d'un circuit, on les rencontre dans les produits de grande série (électronique grand public, automobile, électroménager, radio-téléphones, etc).

La figure IV-8 fournit une illustration des domaines d'application respectifs des circuits « standard », « programmables » et « spécifiques ». En abcisse figure le volume de production de la fonction réalisée, en ordonnée sa complexité.

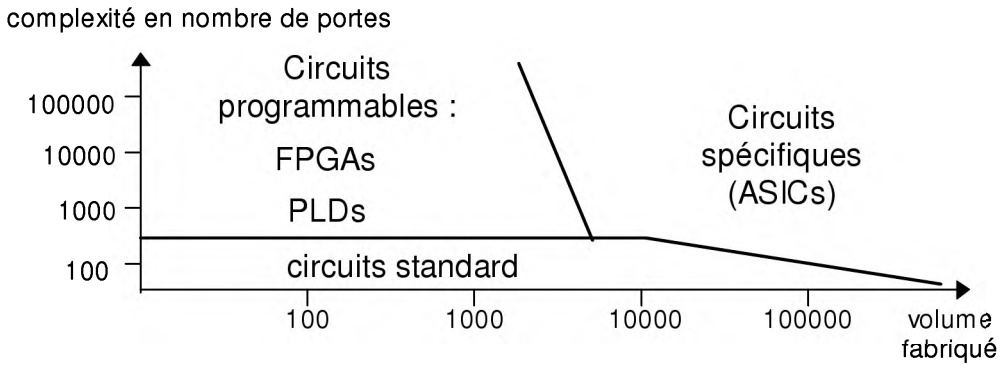


Figure IV-8

[4]PLD : Programmable Logic Device ; FPGA : Field Programmable Gate Array ; ASIC : Application Specific Integrated Circuit.

IV.2.1 Les circuits programmables par l'utilisateur

Un circuit programmable est constitué d'opérateurs logiques élémentaires (portes, multiplexeurs simples et bascules), dont une partie des interconnexions sont modifiables par l'utilisateur. Ce dernier définit la fonction à réaliser par un schéma, des équations logiques, un programme dans un langage de description ou une combinaison des méthodes précédentes. La programmation d'un circuit fait appel à un outil logiciel, un compilateur, et, sauf pour les circuits programmables *in situ*, à un appareil, un programmateur, qui va transférer dans le circuit le résultat de la compilation : la liste des points de jonction à réaliser.

Le compilateur crée cette liste sous forme d'un fichier dans un format connu du programmateur. L'un des standards, universellement reconnu, pour les circuits simples est le format JEDEC : chaque point de jonction possible porte un numéro d'identification ; une jonction réalisée est représentée par un « zéro » (ASCII), une jonction non faite par un « un ». Un extrait d'un tel fichier est donné à titre d'exemple ci-dessous :

```
Cypress C22V10 Jedec Fuse File: mandec1.jed

This file was created on 00 at 12:33:08
by PLA2JED.EXE    06/DEC/94    [v3.15 ] 3.3 IR x90

C22V10*
QP24*                Number of Pins*
QF5828*              Number of Fuses* fusible = connexion
F0*                  Note: Default fuse setting 0*
G0*                  Note: Security bit Unprogrammed*
NOTE DEVICE C22V10*
NOTE PACKAGE PALC22V10-20PC/PI*
NOTE PINS  hor:1 man:2 moore_state_1:14 mealy_state_1:15
mealy:16 mealy_state_0:17 *
NOTE PINS  moore_state_0:21 mealysync:22 moore:23 *
NOTE NODES *
L00000  *-- fusibles 0 à 43
00000000000000000000000000000000000000000000
* Node hor[1] => BANK : 1 *

L00044  *-- fusibles 44 à ...
11111111111111111111111111111111111111111111
11110111101111111111111111111111111011111
11111011110111111111111111111111111011111
00000000000000000000000000000000000000000000
.....
```

Critères de classification

Nous tentons ci-dessous de donner au lecteur quelques repères généraux ; le monde des circuits programmables change extrêmement rapidement, poussé par

une croissance annuelle de la production de plus de 30% par an en volume. Le lecteur curieux est invité à consulter les *data books* des fabricants de circuits pour une plus ample information.

Complexité

La complexité d'un circuit programmable est généralement exprimée en nombre de portes équivalentes, en nombre de bascules disponibles et en nombre de cellules d'entrées sorties (reliées à une broche du circuit).

Ces chiffres doivent être examinés avec prudence, surtout le premier : les « gros » circuits actuels (2005) contiennent jusqu'à 2 000 000 de portes, mais lors de la programmation du circuit certaines de ces portes seront inutilisée. Les opérateurs logiques du circuit sont regroupés en cellules de quelques portes élémentaires, or un opérateur inutilisé dans une cellule ne peut pas être « récupéré » par une autre. Un déchet est donc inévitable. Schématiquement on peut dire que des cellules de petite taille limitent ce déchet, mais compliquent le réseau d'interconnexions, dont la place occupée sur le circuit augmente.

En ordre de grandeur, la plupart des constructeurs admettent un taux moyen d'utilisation des portes de un tiers, ce qui conduit à un nombre de portes utilisables qui peut aller jusqu'à quinze à vingt mille dans le cas ci-dessus.

Le nombre de bascules disponibles va de 8 pour les plus simples (PLD PAL16V8) à plus de 25 000 pour les plus complexes (FPGA Xilinx, Altera ou Actel). Pour les mêmes circuits, le nombre d'entrées sorties disponibles va de 16 à plusieurs centaines.

Programmables une fois ou reprogrammables

La programmation d'un circuit consiste à établir des interconnexions entre des opérateurs logiques. Les technologies employées pour les mémoires ont servi à mettre au point celles des circuits programmables : fusibles (*PROM*), transistors mos à grille flottante qui autorisent des structures effaçables grâce aux rayons ultraviolets (*EPROM*) ou électriquement (*EEPROM* ou *FLASH*). Dans le domaine des circuits simples et moyennement complexes, cette dernière technologie est celle qui domine.

D'autre part des technologies spécifiques aux circuits programmables sont apparues : transistors d'interconnexions commandés par le contenu d'une mémoire *RAM* statique, pour les circuits reconfigurables *in situ* ; anti-fusibles pour les circuits complexes programmables une fois. Les premiers (in situ) exécutent automatiquement une séquence d'initialisation après la mise sous tension, en allant chercher leur « table des fusibles » dans une mémoire *ROM* externe ou en dialoguant avec un système à microprocesseur. Dans les seconds (anti-fusibles), la programmation consiste à créer une surtension entre les bornes des contacts isolés que l'on veut réunir, surtension qui provoque le perçage du diélectrique et une véritable soudure par points.

Vitesse et consommation

Les circuits programmables n'échappent évidemment pas au compromis vitesse-consommation, commun à tous les circuits électroniques. La technologie utilisée étant généralement à base de cellules CMOS, la consommation est, en gros, proportionnelle à la fréquence de travail.

Contentons nous de citer quelques ordres de grandeur :

- Un PAL22V10 (10 bascules) rapide, 166 Mhz de fréquence maximum d'horloge, consomme jusqu'à 190 mA.
- Un compteur 16 bits à chargement parallèle, implanté dans un FPGA, cadencé à 100 Mhz consomme typiquement 50 mA.

Repères croisés

Toutes les technologies d'interconnexions ne sont pas disponibles avec toutes les complexités. Le tableau ci dessous résume ces corrélations.

Technologie	Types de circuits	Avantages	Inconvénients	Fabricants principaux
Fusibles	PLDs simples 10 bascules	rapidité	programmables une fois	tous
EPROM	PLDs simples et complexes < 250 bascules	reprogrammables après effacement aux ultra-violets	boitiers chers (fenêtre optique)	tous
FLASH	PLDs simples et complexes FPGAs < 400 bascules	reprogrammables électriquement	nécessitent un programmateur	tous AMD
SRAMS	des PLDs simples aux FPGAs complexes	reconfigurables in situ	circuits annexes et procédure d'initialisation	XILINX ALTERA ATMEL
Anti fusibles	FPGAs complexes	rapidité	programmables une fois	CYPRESS ACTEL QUICK LOGIC TEXAS INS.

IV.2.2 Les circuits spécifiques

Réservés aux grandes séries, les circuits spécifiques nécessitent l'intervention du fabricant pour réaliser la fonction définie par le concepteur de l'application. On distingue classiquement :

- Les circuits prédiffusés, dans lesquels les opérateurs logiques sont en place indépendamment de l'application. Seule la dernière couche d'interconnexions est modifiée par le fabricant, suivant la fonction à réaliser.
- Les circuits précaractérisés, pour lesquels les cellules sont placées et interconnectées à la demande.

V Méthodes de synthèse

Au cours des quinze dernières années, les méthodes de conception des fonctions numériques ont subi une évolution importante. Dans les années soixante dix, la majorité des applications étaient construites autour de circuits intégrés standard, souvent pris dans la famille TTL. Au début des années quatre vingt apparurent parallèlement :

- les premiers circuits programmables par l'utilisateur (PALs), du côté des circuits simples,
- les circuits intégrés spécifiques (ASICs)[1], pour les fonctions complexes fabriquées en grande série.

La complexité des seconds a nécessité la création d'outils logiciels de haut niveau, qui sont à la description structurelle (schémas au niveau des portes élémentaires) ce que les langages évolués sont au langage machine dans le domaine de la programmation.

Les premières générations de circuits programmables étaient conçus au moyen de simples programmes de traduction d'équations logiques en table de fusibles. A l'heure actuelle, l'écart de complexité entre circuits programmables et ASICs s'est restreint : on trouve une gamme continue de circuits qui vont des héritiers des premiers PALs, équivalents de quelques centaines de portes, à des FPGAs ou des LCAs[2] de quelques dizaines de milliers de portes équivalentes. Les outils d'aide à la conception se sont unifiés, un même langage, VHDL par exemple, peut être employé quels que soient les circuits utilisés, des PALs aux ASICs.

Le remplacement, dans la plupart des applications, des fonctions standard complexes par des circuits programmables, s'accompagne d'un changement dans les méthodes de conception :

- On constate un « retour aux sources » : le concepteur d'une application élabore sa solution en descendant au niveau des bascules élémentaires, au même titre que l'architecte d'un circuit intégré.
- L'utilisation systématique d'outils de conception assistée par ordinateur (C.A.O.), sans lesquels la tâche serait irréalisable, rend caducs les fastidieux

[1] Programmable Array Logic, Application Specific Integrated Circuit.
[2] Field Programmable Logic Array, Logic Cell Array.

calculs de minimisation d'équations logiques. Le concepteur peut se consacrer entièrement aux choix d'architecture qui sont, eux, essentiels.

- La complexité des fonctions réalisables dans un seul circuit pose le problème du test. Les outils traditionnels de tests de cartes imprimées, du simple oscilloscope à la « planche à clous » en passant par l'analyseur d'états logiques, ne sont plus d'un grand secours, dès lors que la grande majorité des équipotentielles sont inaccessibles de l'extérieur. Là encore, la C.A.O. joue un rôle essentiel. Encore faut-il que les solutions choisies soient analysables de façon sûre. Cela interdit formellement certaines astuces, parfois rencontrées dans des schémas traditionnels de logique câblée, comme des commandes asynchrones utilisées autrement que pour une initialisation lors de la mise sous tension, par exemple[3].
- Les langages de haut niveau, comme VHDL, privilégient une approche globale des solutions. Dès lors que l'architecture générale d'une application est arrêtée, que les algorithmes qui décrivent le fonctionnement de chaque partie sont élaborés, le reste du travail de synthèse est extrêmement simple et rapide.

Nous tenterons ci-dessous de mettre en évidence quelques règles de conception et de donner au lecteur les clés de compréhension de la littérature spécialisée, notamment les notices des fabriquants de circuits et les notes d'application qui les accompagnent.

V.1. Les règles générales

Avant de présenter les outils de base du concepteur, il n'est sans doute pas inutile de préciser quelques règles, qui pourront sembler de simple bon sens, mais dont le non respect a conduit beaucoup de réalisations vers le cimetière des projets morts avant d'être nés.

Du général au particulier, une approche descendante

L'erreur de méthode la plus fréquente, et la plus pénalisante, que commettent beaucoup de débutants dans la conception des systèmes électroniques, qu'ils soient analogiques ou numériques, est sans doute de dessiner des schémas, voire de les câbler, avant même d'avoir une vision claire de l'ensemble de la tâche à accomplir. Le travail de réflexion sur la structure générale d'une application est primordial.

Ce que l'on appelle traditionnellement la méthode descendante (*top down design*), n'est rien d'autre que l'application de cette règle simple : quand on conçoit un ensemble, *on va du général au particulier*, on ne s'occupe des détails que quand le cahier des charges a été mûrement réfléchi, et que le plan général de la solution a été établi.

[3]Même à l'époque du règne des fonctions standard, ces pratiques étaient éminemment douteuses.

Si, au cours de la descente vers les détails, on découvre qu'une difficulté imprévue apparaît, il faut revenir au niveau général pour voir comment la réponse à cette difficulté s'insère dans le plan d'ensemble.

Diviser pour règner

Le premier réflexe à avoir, face à un problème, un tant soit peu complexe à résoudre, est de le couper en deux. La démarche précédente est répétée, pour chaque demi-problème, jusqu'à obtenir des sous-ensembles dont la réalisation tient en quelques circuits élémentaires, en quelques lignes de code source dans un langage ou dans un diagramme de transitions qui ne dépasse pas une dizaine d'états différents.

L'un des auteurs de ce livre garde un souvenir cuisant de la première introduction d'un système de CAO, pour réaliser un projet d'électronique numérique, auprès d'étudiants d'IUT que nous n'avions pas suffisamment averti des pièges liés à la puissance de l'outil. Ce système pouvait assurer automatiquement la répartition d'une application dans plusieurs circuits programmables dont on avait, au préalable, établi la liste.

Le problème posé était relativement simple ; il s'agissait de créer un automate pilotant un circuit de multiplication, suivant un algorithme séquentiel, et assurant l'interface entre ce circuit et un microprocesseur. Trois sous-ensembles en interaction devaient être créés :

- Un interface avec le bus du microprocesseur, qui assure le bon respect du protocole d'échange.
- Un compteur qui permet de savoir où en est la multiplication : à chaque impulsion d'horloge le multiplieur fournit un chiffre binaire du résultat, si le produit est codé sur 16 bits, il faut attendre 16 périodes d'horloge pour l'obtenir.
- L'automate séquentiel qui pilote le tout.

Trois blocs, dont la réalisation nécessite, en tout, une douzaine de circuits standard TTL.

La fusion des trois blocs dans une grande boite « fourre tout », ne rentre pas dans un seul circuit de type 22V10, ce qui est normal. Si l'utilisateur du système de CAO laisse ce dernier se charger lui-même du découpage de la réalisation en sous-ensembles, le résultat est une carte qui contient, outre le multiplieur lui-même, une ribambelle de circuits programmables, utilisés à 50% de leur capacité.

Le simple fait de subdiviser la solution en petits modules, ce qui permet de guider le logiciel de placement, divise par deux le nombre de circuits nécessaires et, soit dit en passant, permet d'en contrôler facilement le bon fonctionnement.

Mener de front l'aspect structurel et l'aspect fonctionnel

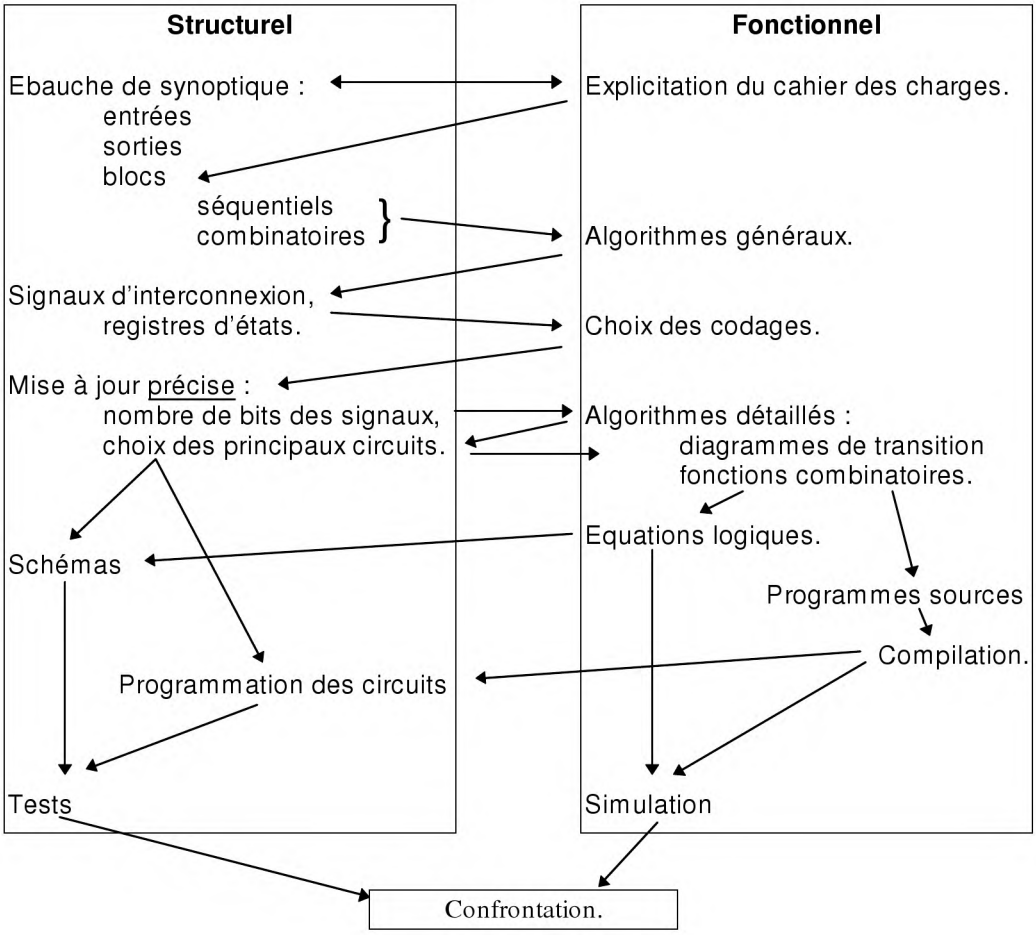
La réalisation d'un ensemble électronique se mène sur deux plans parallèles, structurel et fonctionnel, qui doivent, à chaque étape, être cohérents entre eux.

Le plan structurel précise les subdivisions en blocs, combinatoires ou séquentiels, fixe les signaux d'interconnexions entre ces blocs, pour aboutir, en

dernière étape, à un éventuel schéma. Une fois réalisés, chaque bloc et l'ensemble sont soumis à des tests de validation.

Le plan fonctionnel précise l'algorithme utilisé dans chaque bloc, fixe le codage des signaux, pour aboutir, en dernière étape, à des équations logiques ou à des modules de programme dans le langage choisi. Le fonctionnement de chaque sous-ensemble et du système complet peut être simulé.

Le tableau qui suit illustre le processus :



Citons quelques incohérences graves parfois rencontrées entre les deux plans :

- Les noms de signaux ne correspondent pas.
- Les tailles des registres d'états sont incompatibles avec les diagrammes de transitions qui les décrivent ; une bascule se voit pourvue de plus de deux états, ou, inversement, un diagramme qui contient cinq états est censé représenter le fonctionnement d'un registre de quatre bascules.
- Les équations de certaines commandes de circuits complexes ne sont pas précisées.

Faciles à corriger au début, de telles incohérences sont sources d'erreurs difficiles à identifier quand elles apparaissent lors des tests de validation.

Aléas et testabilité : privilégier les solutions synchrones

Une plaie de trop de réalisations rencontrées est le mélange, dans une même unité fonctionnelle, des commandes asynchrones et synchrones. Citons quelques exemples :

Mises à zéro ou chargements parallèles

Les mises à zéro ou les chargements parallèles de registres, par des commandes à action directe, c'est à dire indépendamment de l'horloge, sont la source de nombreux ennuis ultérieurs ; ce type de pratique est à condamner sans appel. L'utilisation de telles commandes en fonctionnement normal conduit à la génération d'impulsions de durées inconnues, souvent très faibles, donc difficiles à observer. Les outils de test et de simulation gèrent fort mal ces commandes, il devient impossible de valider correctement la fonctionnalité d'un équipement qui les utilise.

L'utilisation de ces commandes asynchrones conduit parfois à un résultat qui, s'il peut être instructif dans un contexte d'enseignement, est catastrophique dans une réalisation : une carte qui semble donner toute satisfaction quand on l'observe, par exemple avec un oscilloscope, cesse de fonctionner dès que l'on retire l'appareil de mesure. L'origine du phénomène tient à la charge capacitive supplémentaire apportée par la sonde de mesure. Cette charge peut, si elle est bien placée, modifier la durée et l'amplitude d'impulsions étroites dont l'existence est déterminante pour le bon fonctionnement de l'ensemble. Il est, nous l'espérons, inutile de préciser que le dépannage d'un tel objet relève plus de la divination que d'une méthodologie raisonnée[4].

Les signaux d'horloges

Le bloquage, par exemple par une porte, des signaux d'horloge pour maintenir l'état d'un registre, est une autre erreur que l'on rencontre parfois. Cette faute, qui provoque des décalages temporels entre les signaux d'horloge (*clock skew*) appliqués aux différentes parties d'une carte, ou d'un circuit, risque de conduire à des violations de temps de maintien ou de prépositionnement, d'où des comportements imprévisibles des registres concernés.

Un autre effet pervers des circuits combinatoires de « calcul » des signaux d'horloge, est la génération, difficile à contrôler, d'impulsions parasites sur ces signaux. La recherche de ces impulsions, suffisamment larges pour faire commuter les circuits actifs sur des fronts, mais suffisamment étroites pour ne pas être vues lors d'un examen rapide avec un oscilloscope, est un passe temps dont on se lasse très vite.

Quand il est nécessaire d'appliquer à différentes parties d'un ensemble des signaux d'horloges différents, il est indispensable de traiter à part, et de façon

[4]Un effet réciproque peut également être rencontré : la carte qui fonctionne quand on ne la regarde pas, et qui tombe en panne quand on l'observe, la sonde de l'appareil ayant « gommé » une impulsion essentielle, bien que fragile, à la bonne santé de l'ensemble.

méticuleuse, la réalisation du distributeur d'horloge correspondant[5]. Notons, en passant, que pour ces fonctions il convient de surveiller de très près les modifications apportées par les optimiseurs ; ces derniers ont la fâcheuse tendance d'éliminer les portes inutiles d'un point de vue algébrique, même si elles sont utiles d'un point de vue circuit.

Les bascules asynchrones (D Latch, R S)

Ce chapitre est consacré aux méthodes de conception des automates séquentiels. Ces opérateurs ont la particularité de fonctionner en boucle fermée : l'état futur dépend de l'état initial. Ce mode de fonctionnement exclut à priori l'usage de bascules ou registres asynchrones dans la réalisation de tels systèmes.

Les bascules D latch ou R-S ont, parfois, leur place à la périphérie des systèmes, elles servent alors d'interfaces entre deux ensembles indépendants, pilotés par des horloges différentes, qui échangent des informations en respectant un protocole bien défini.

V.2. Les machines synchrones à nombre fini d'états

Les machines à nombre fini d'états (*Finite state machines*), en abrégé machines d'états, ou automates finis ou, encore, séquenceurs câblés[6], sont largement utilisées dans les fonctions logiques de contrôle, qui forment le coeur de nombreux systèmes numériques : arbitres de bus, circuits d'interfaces des systèmes à base de microprocesseurs, circuits de gestion des protocoles de transmission, systèmes de cryptages etc. Plus prosaïquement, la quasi totalité des fonctions séquentielles standard, compteurs, par exemple, peuvent être analysées, ou synthétisées, en adoptant le point de vue « machine d'états ».

Une machine d'états est un système dynamique (i.e. évolutif) qui peut se trouver, à chaque instant, dans une position parmi un nombre fini de positions possibles. Elle parcourt des cycles, en changeant éventuellement d'état lors des transitions actives de l'horloge, dans un ordre qui dépend des entrées externes, de façon à fixer sur ses sorties des séquences déterminées par l'application à contrôler.

---

[5]Le seul argument sérieux qui peut, parfois, et avec une extrême prudence, conduire un concepteur à utiliser le bloquage de l'horloge, est la consommation : un circuit séquentiel, surtout en technologie CMOS, dont on arrête l'horloge, consomme moins que si le maintien est généré de façon synchrone. Comme quoi les règles doivent être édictées, mais parfois transgressées. Mieux vaut, dans ces cas, avoir conscience qu'il y a transgression, donc danger.

[6]Les différences de terminologie correspondent à des différences de point de vue : l'électronicien s'intéresse à la réalisation de la machine, donc à son fonctionnement interne. Il concentre son attention sur le fonctionnement d'un registre qui peut passer d'un état interne à un autre, en fonction de commandes qui lui sont fournies. L'automaticien et l'informaticien s'intéressent, de prime abord, au fonctionnement externe du même objet : ils en attendent des actions (automates) en sortie correspondant à une séquence (séquenceurs) fixée par l'application à laquelle la machine est destinée. Peu importent, à ce niveau et jusqu'à un certain point, les détails de réalisation interne du séquenceur.

Un programmateur de machine à laver en est l'illustration typique : suite à la mise en marche, le programmateur contrôle que la porte est fermée, si oui il commande l'ouverture de la vanne d'arrivée de l'eau, quand la machine est pleine etc.

L'architecture générale d'une machine d'états simple est celle de la figure V-1 :

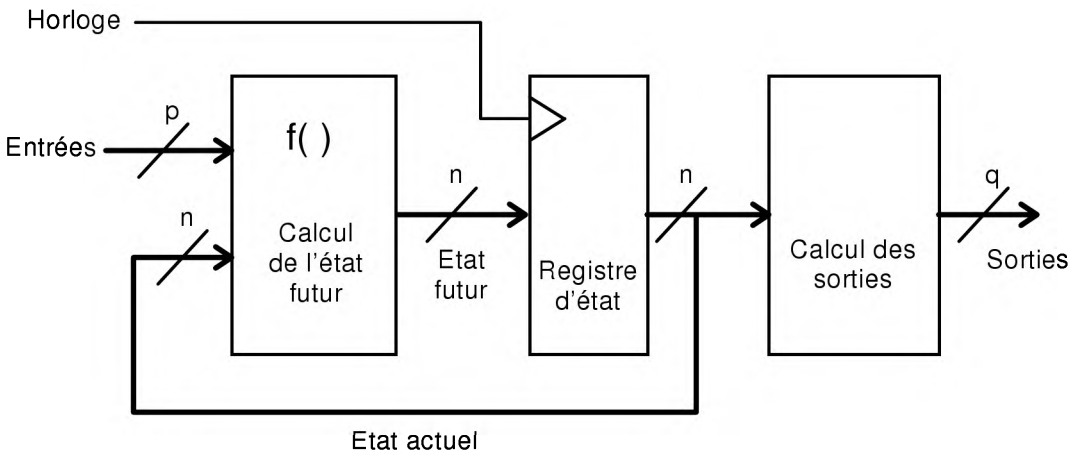


Figure V-1

Ce synoptique général va nous servir, éventuellement légèrement modifié, de support dans les explications qui suivent.

V.2.1 Horloge, registre d'état et transitions

Le registre d'état, piloté par son horloge, constitue le coeur d'une machine d'états, les autres blocs fonctionnels, bien que généralement nettement plus complexes, sont « à son service ».

Le registre d'état

Le registre d'état est constitué de n bascules synchrones, nous admettrons dans ce qui suit, sauf précision contraire, que ce sont des bascules D, ce qui n'impose aucune restriction de principe, puisque nous savons passer d'un type de bascule à un autre.

Etat présent et état futur.

Le contenu du registre d'état représente l'état de la machine, il s'agit d'un nombre codé en binaire sur n bits, dans un code dont nous aurons à reparler.

L'entrée du registre d'état constitue l'état futur, celui qui sera chargé lors de la prochaine transition active de l'horloge. Le registre d'état constitue la mémoire de la machine, l'élément qui matérialise l'histoire de son évolution.

L'importance de ce registre est telle que beaucoup de circuits programmables offrent la possibilité de le charger, à des fins de tests, avec une valeur arbitraire, indépendante du fonctionnement normal de la machine. Toute procédure de test devra s'appuyer sur la connaissance du contenu de ce registre[7], par une mesure réelle, ou en simulation.

Taille du registre et nombre d'états

La taille du registre d'état fixe évidemment le nombre d'états accessibles à la machine. Si n est le nombre de bascules et N le nombre d'états accessibles, ces deux nombres sont reliés par la relation :

$$N = 2^n$$

Cette relation, qui ne présente guère de difficulté, est pourtant trop souvent oubliée des concepteurs débutants :

- En synthèse le nombre d'états nécessaires est issu du cahier des charges de la réalisation, on en déduit aisément une taille minimum du registre.
- Quand tous les états disponibles ne sont pas utilisés, l'oubli des états inutilisés peut conduire à de cruelles déconvenues si on oublie de prévoir leur évolution.

Le rôle de l'horloge

Le rôle de l'horloge, dans une réalisation synchrone, est de supprimer toute possibilité d'aléas dans l'évolution de l'état. Idéalement, entre deux transitions actives de l'horloge le système est figé, en position mémoire, son état ne peut pas changer. Dans la réalité, le temps de latence qui sépare deux fronts consécutifs du signal de l'horloge est mis à profit pour permettre aux circuits combinatoires d'effectuer leurs calculs, sans que les temps de retards, toujours non nuls, ne risquent de provoquer d'ambiguïté dans le résultat.

Ce qui a été dit à propos des bascules élémentaires se généralise : chaque transition d'horloge est suivie d'une période de grand trouble dans la valeur de l'état futur, le concepteur doit s'assurer que cette valeur est stable quand survient la transition d'horloge suivante (figure V-2)[8] :

---

[7]Ce point peut être moins trivial qu'il n'y paraît, les sorties des bascules ne sont pas toujours disponibles en sortie d'un circuit, on parle alors de bascules enterrées. Dans de tels cas les circuits complexes offrent de plus en plus souvent la possibilité de « ressortir », par une procédure particulière, les contenus de ces bascules (c'est l'une des fonctions possibles des automates de test dits « *boundary scan* » qui sont intégrés dans certains circuits).

[8]Ce qui est dit ici est en étroite relation avec le contenu du paragraphe II-3, qui aboutit au calcul de la fréquence maximum de fonctionnement d'un circuit dans lequel interviennent des rétro couplages.

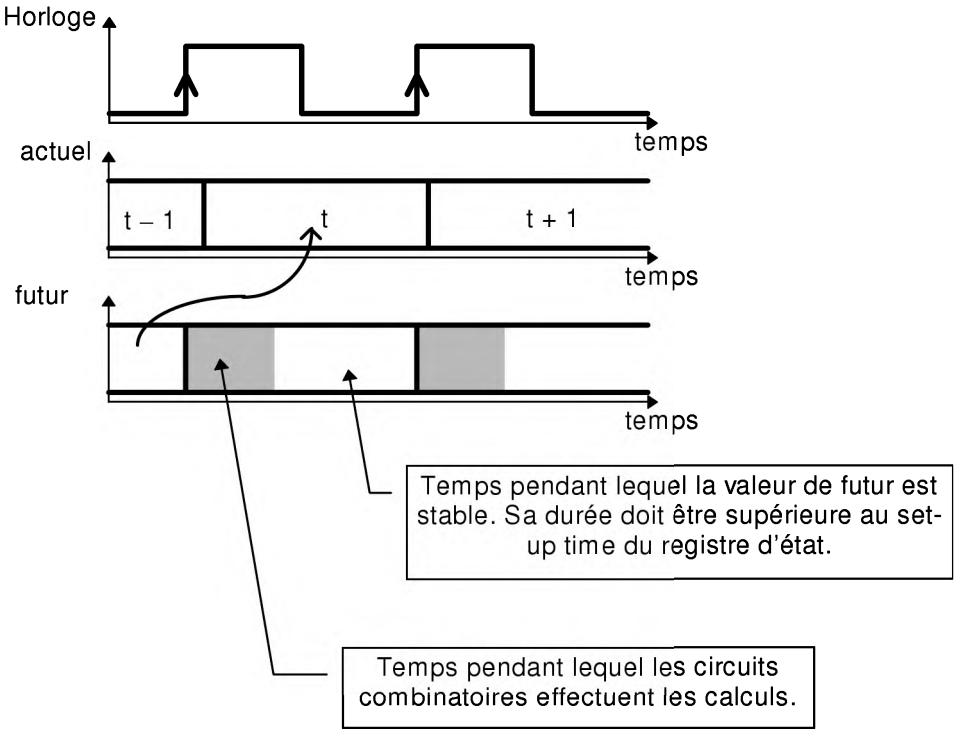


Figure V-2

Le temps devient une variable discrète

Sauf pour le calcul des limites de fonctionnement du système, le temps devient une variable discrète, un nombre entier qui indique simplement combien de périodes d'horloges se sont écoulées depuis l'instant pris comme origine.

Une machine d'états est complètement décrite par une équation de récurrence, qui permet de connaître le contenu du registre d'état à l'instant t, t entier, en fonction de ses valeurs précédentes et de celles des entrées.

Les entrées et l'état actuel déterminent l'état futur

Dans une architecture comme celle de la figure V-1, l'équation de récurrence évoquée précédemment est du premier ordre (la portée temporelle de la mémoire est égale à une période d'horloge) :

$$Etat\_actuel(t) = Etat\_futur(t - 1)$$

Or la fonction logique combinatoire, f( ), qui calcule l'état futur, fournit une valeur qui dépend de l'état présent et des entrées, d'où la relation générale qui décrit l'évolution d'une machine d'états :

$$Etat\_actuel(t) = f(Etat\_actuel(t - 1) , Entrées(t - 1))$$

Cette équation est la généralisation de celles qui ont été introduites à propos des bascules élémentaires.

Il est important de noter que, si l'équation qui régit l'évolution du registre d'état, pris dans son ensemble, est du premier ordre, ce n'est pas vrai pour l'équation d'évolution d'*une* bascule qui est, elle, plus complexe. Toutes les bascules du registre d'état sont, en général, couplées. L'ordre de l'équation d'évolution d'une bascule peut atteindre n, où n est la taille du registre d'état[9].

Exemple : un diviseur de fréquence par 3 ou 4

Pour illustrer ce qui précède, considérons le schéma de la figure V-3 :

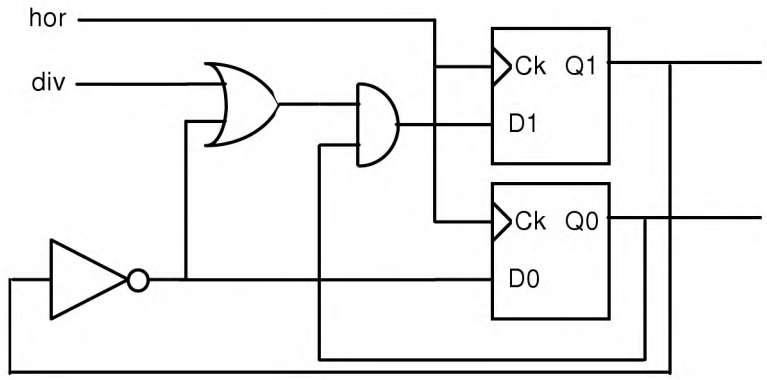


Figure V-3

L'état du système est (Q1,Q0), ensemble constitué des états individuels des deux bascules. On admet que l'entrée extérieure div est synchrone de l'horloge. A chaque front montant du signal d'horloge hor, le système évolue suivant le système d'équations :

$$Q1(t) = Q0(t - 1) * (\overline{Q1(t - 1)} + div(t - 1))$$
$$Q0(t) = \overline{Q1(t - 1)}$$

On peut remarquer, en passant, que l'équation de chaque bascule est effectivement du deuxième ordre, par exemple pour Q1 :

---

[9]On peut rapprocher ce point des méthodes d'analyse des circuits analogiques : la présentation « variables d'état » conduit à une équation différentielle du premier ordre, qui porte sur un vecteur de dimension n, la présentation traditionnelle considère chaque grandeur électrique comme obéïssant individuellement à une équation différentielle dont l'ordre peut atteindre n. Le passage d'un mode de représentation à l'autre n'est simple que dans le cas des équations linéaires. Les équations des systèmes numériques sont généralement non linéaires, avec une exception notable : les générateurs de séquences pseudo aléatoires qui utilisent un registre à décalage rebouclé par des sommes modulo 2.

$$Q1(t) = \overline{Q1(t-2)} * (\overline{Q1(t-1)} + div(t-1))$$

Sur le chronogramme de la figure V-4, construit à partir des équations précédentes, on voit que :

- si div = '0' les sorties Q1 et Q0 sont des signaux périodiques, de fréquence égale au tiers de la fréquence d'horloge,
- si div = '1' la fréquence des signaux Q1 et Q0 est égale au quart de la fréquence d'horloge.

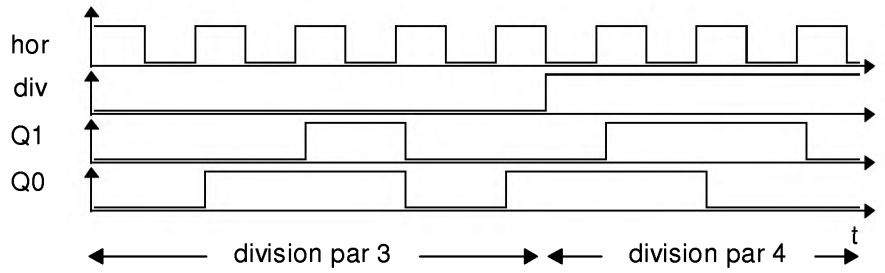


Figure V-4

Si les chronogrammes permettent d'illustrer un fonctionnement, ils ne constituent pas une méthode efficace d'analyse, et, encore moins, de synthèse. Dans l'étude de machines d'états simples, les diagrammes de transitions constituent une approche plus compacte, donc plus puissante, tout en fournissant exactement le même niveau de détails.

Les diagrammes de transitions

Dans un diagramme de transitions, on associe à *chaque* valeur possible du registre d'état, une case. L'évolution du système est représentée par des flèches, les transitions, qui vont d'un état à un autre[10]. Comme pour les bascules élémentaires, une transition est effectuée si trois conditions sont réunies :

1. Le système est dans l'état « source » de la transition considérée,
2. une éventuelle condition de réalisation sur les entrées doit être vraie,
3. un front actif d'horloge survient.

[10]Les automaticiens utilisent souvent un diagramme similaire, le GRAFCET, dont les étapes peuvent être matérialisées par les états d'une machine. Nous adoptons les diagrammes de transitions plutôt que le GRAFCET, car seuls les premiers sont utilisés, et le sont beaucoup, dans la littérature professionnelle électronique (notices d'applications de circuits, manuels des systèmes de CAO, etc.). Le passage d'un type de diagramme à l'autre ne présente guère de difficulté.

Si aucune transition n'est active, le système reste dans son état initial. S'il n'y a pas d'ambiguïté le signal d'horloge est généralement omis (horloge unique), mais il conditionne toutes les transitions.

Reprenant l'exemple précédent, nous représentons, figure V-5, le diagramme de transitions du diviseur par trois ou quatre. Le registre d'état contient deux bascules, soit quatre états accessibles. Pour chaque état initial, les équations du diviseur fournissent l'état d'arrivée.

Dans cet exemple, quel que soit l'état de départ, et quelle que soit la valeur de l'entrée div, il y a toujours une transition active ; le système change donc d'état à chaque front montant du signal d'horloge.

Quand le diviseur est dans l'état 3 le chemin parcouru dépend de la valeur de l'entrée div. Si div = '0' un cycle complet contient trois états, d'où la division de fréquence par trois, si div = '1', un cycle complet comporte quatre états, d'où la division de fréquence par quatre[11].

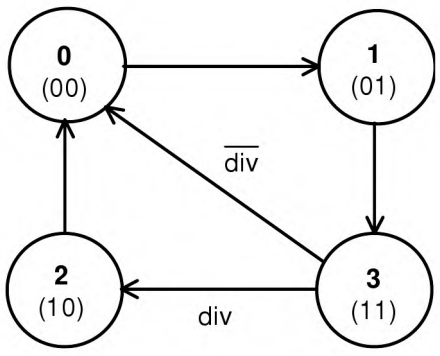


Figure V-5

Dans l'exemple précédent, nous sommes partis des équations d'un système pour aboutir au diagramme de transitions qui en décrit le fonctionnement. Il s'agit là d'un travail d'analyse. Le concepteur se trouve généralement confronté au problème inverse : du cahier des charges il déduit un diagramme de transitions, et de ce diagramme il souhaite tirer les équations de commandes des bascules du registre d'état, ou une description en VHDL[12].

[11]Cet exemple est une version simple de ce que l'on appelle les diviseurs par N/(N+1). Ces circuits sont utilisés dans les synthétiseurs de fréquences, à boucle à verrouillage de phase, avec des valeurs plus grandes de N (255, par exemple).

[12]Il existe des logiciels de traduction des diagrammes de transitions en code source, dans un langage. Le plus souvent le programme obtenu s'apparente plus à une description du type « flot de

Du diagramme aux équations

Rajoutons à l'exemple précédent une commande supplémentaire, en, telle que :

- si en = '0' , la machine ne quitte pas l'état 1, quand elle y arrive,
- si en = '1' , la machine fonctionne comme précédemment.

Le diagramme de transitions devient (figure V-6) :

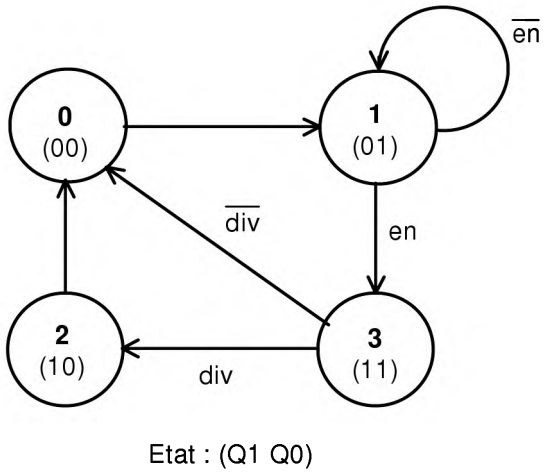


Figure V-6

Quand le nombre de bascules du registre d'état et le codage des états sont connus, le passage du diagramme de transitions aux équations de commandes des bascules est immédiat. Les détails des calculs dépendent du type de bascules utilisées pour réaliser le registre d'état.

Avec des bascules D :

Il suffit de recenser, pour chaque bascule, toutes les transitions, *et les maintiens*, qui conduisent à la mise à '1' de la bascule. Dans notre exemple il y a deux bascules :

- Q1 est à '1' dans les états 2 et 3, obtenus par les transitions $1 \rightarrow 3$ et $3 \rightarrow 2$.
- Q0 est à '1' dans les états 1 et 3, obtenus par les transitions $0 \rightarrow 1$, $1 \rightarrow 1$ (maintien) et $1 \rightarrow 3$.

---

données », avec des bascules décrites au niveau structurel, qu'à une réelle description de haut niveau, dans un langage comportemental.

D'où, en notant les états de façon symbolique par leur numéro souligné, et en simplifiant, éventuellement les expressions obtenues :

$$D1 = \underline{1} * en + \underline{3} * div = \overline{Q1} * Q0 * en + Q1 * Q0 * div$$
$$D0 = \underline{0} + \underline{1} * \overline{en} + \underline{1} * en = \underline{0} + \underline{1} = \overline{Q1}$$

Avec des bascules T :

Il suffit de recenser, pour chaque bascule, toutes les transitions qui conduisent à un changement d'état de la bascule. Dans notre exemple il y a deux bascules :
- Q1 change dans les transitions $1 \rightarrow 3$, $3 \rightarrow 0$ et $2 \rightarrow 0$.
- Q0 change dans les transitions $0 \rightarrow 1$, $3 \rightarrow 0$ et $3 \rightarrow 2$.

D'où, en notant les états de façon symbolique par leur numéro souligné et en simplifiant, éventuellement les expressions obtenues :

$$T1 = \underline{1} * en + \underline{3} * \overline{div} + \underline{2} = \overline{Q1} * Q0 * en + Q1 * \overline{div} + Q1 * \overline{Q0}$$
$$T0 = \underline{0} + \underline{3} * \overline{div} + \underline{3} * div = \underline{0} + \underline{3} = \overline{Q1 \oplus Q0}$$

La comparaison entre les deux solutions, bascules D ou bascules T, montre que dans l'exemple considéré, la première solution conduit à des équations plus simples (ce n'est pas toujours le cas). Certains circuits programmables offrent à l'utilisateur la possibilité de choisir le type de bascules, ce qui permet d'adopter la solution la plus simple[13].

Table de transitions

Pour passer d'un diagramme de transitions aux équations de commandes des bascules, le concepteur débutant peut toujours recourir à une table de vérité qui récapitule toutes les transitions possibles.

Si cette méthode est systématique, elle présente évidemment l'inconvénient d'être fort lourde. Le nombre de variables d'entrées de la table devient vite, même pour des problèmes simples, très élevé.

Le tableau qui suit correspond à la dernière version de notre diviseur par trois ou quatre. Quand, pour une transition, la valeur d'une commande est indifférente, elle apparaît par la valeur 'x'.

[13]Soyons clairs, c'est en général l'optimiseur du compilateur qui fait ce choix, mais l'utilisateur a un droit de regard, et d'action, sur ce que fait le logiciel.

Etat initial		Entrées		Etat final	
Q1	Q0	en	div	D1	D0
0	0	x	x	0	1
0	1	0	x	0	1
0	1	1	x	1	1
1	1	x	0	0	0
1	1	x	1	1	0
1	0	x	x	0	0

Table de transitions du diviseur par 3/4.

Cette table n'apporte rien de plus que le diagramme de transitions, son utilité est d'autant plus discutable que l'on effectue rarement les calculs à la main, et nous verrons qu'il est très simple de passer directement d'un diagramme de transitions au programme VHDL correspondant.

L'état futur est unique

Pour représenter, de façon non ambiguë le fonctionnement d'un système, un diagramme de transitions doit respecter certaines règles qui concernent les états, d'une part, et les transitions, d'autre part. Leur non respect constitue une erreur :

- Un état, représenté par un code unique, ne peut apparaître qu'une seule fois dans un diagramme. Cette règle, somme toute fort naturelle, n'est généralement pas source d'erreurs, ou, au pire, provoque par son non respect, des erreurs faciles à identifier et à corriger.
- La machine est forcément « quelque part ». Cela impose que la condition de maintien dans un état soit le complément logique de la réunion de toutes les conditions de sortie de l'état. Cette règle est générée automatiquement par les logiciels – seules les transitions doivent être spécifiées, les compilateurs en déduisent la condition de maintien – mais peut être une source d'erreurs dans une synthèse manuelle.
- La machine ne peut pas être à deux endroits différents à la fois. Les transitions qui partent d'un état, pour arriver à des états *différents*, doivent être assorties de *conditions mutuellement exclusives.*

Ces deux derniers points méritent des éclaircissements ; leur non respect, qui ne saute pas toujours aux yeux, est l'une des principales sources d'erreur dans les diagrammes de transitions. Précisons cela en modifiant quelque peu le diviseur étudié précédemment.

On souhaite remplacer, dans le diviseur par 3/4, les commandes en et div par deux commandes, div3 et div4, actives à '1', qui fournissent globalement les mêmes fonctionnalités, mais avec une répartition des rôles un peu différente :

- div3 commande le fonctionnement en diviseur par 3,
- div4 commande le fonctionnement en diviseur par 4.

– Si les deux commandes sont inactives, la machine s'arrête dans l'état 1.

Deux versions du diagramme de transitions de la nouvelle variante du diviseur sont représentées figure V-7.

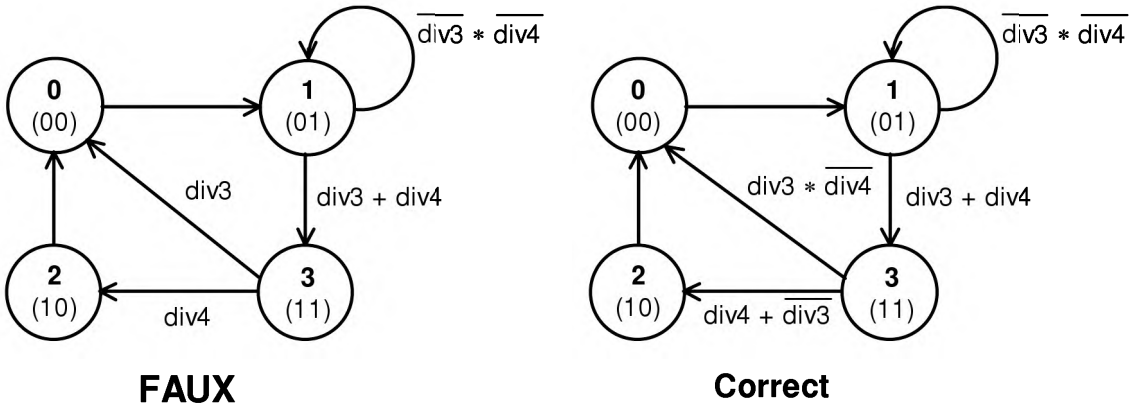


Figure V-7

Sur les deux versions on a indiqué que le maintien dans l'état 1 est bien obtenu par complémentation de la condition de cet état, ce qui est correct.

La première version, marquée comme fausse sur la figure, contient deux erreurs qui concernent l'évolution à partir de l'état 3 :

1. Une erreur de *syntaxe*, si div3 et div4 sont tous les deux actifs, le diagramme indique deux destinations différentes, ce qui est absurde.
2. Une erreur de *sens*, par rapport au cahier des charges, si div3 et div4 sont tous les deux inactifs, le diagramme indique un maintien dans l'état 3, par absence de transition.

La deuxième version présente une solution correcte au problème, l'erreur de syntaxe a disparu, et on a établi une priorité de la division par quatre. Si les deux commandes sont actives la machine prend le chemin de la division par quatre ; elle réagit de même si les deux commandes deviennent inactives alors que l'état 3 est actif, elle évolue vers l'état 1, pour s'y arrêter, en passant par les états 2 et 0.

Ce genre d'erreurs est vite arrivé. Lors de la traduction en VHDL d'un diagramme de transitions les erreurs de syntaxe disparaissent le plus souvent, grâce aux priorités qu'introduisent les algorithmes séquentiels :

– les instructions « `if ... elsif ... else ... end if` » décomposent un choix multiple en alternatives binaires, d'où les conflits de destinations ont disparu ;

– les instructions « `case ... when ... end case` » doivent obligatoirement traiter toutes les alternatives.

Mais la disparition des erreurs de syntaxe peut, malheureusement, s'accompagner d'une modification du sens qu'avait prévu un concepteur insuffisamment rigoureux.

Attention aux états pièges !

Un état piège est un état dans lequel la machine peut entrer, mais dont elle ne sort jamais, comme un piège à anguilles. Cela peut être volontaire, aboutissement d'une séquence d'initialisation qui suit la mise sous tension d'un système, par exemple ; mais c'est rare. De plus dans ce genre de situation, l'état piège est explicite, il est donc visible. Plus dangereux sont les pièges cachés, qui ne figurent pas sur le diagramme de transitions.

Prenons un exemple.

On souhaite réaliser, au moyen de deux bascules, deux signaux rigoureusement synchrones, issus de deux diviseurs de fréquence par deux couplés, tels que les sorties des bascules soient toujours complémentaires.

La première version du diagramme de transitions de la figure V-8 semble convenir, a tort.

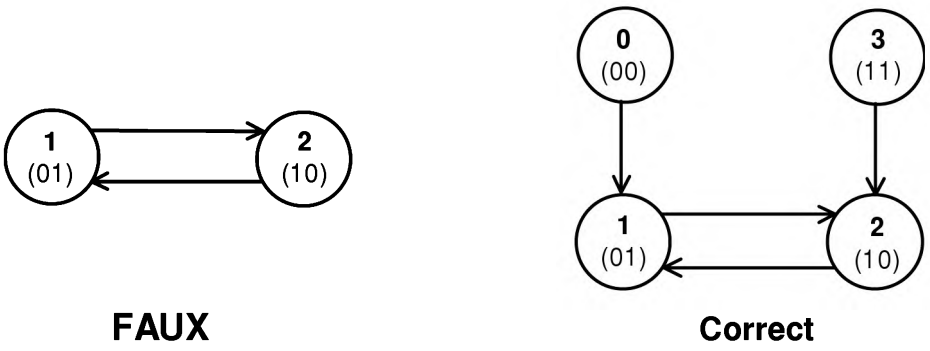


Figure V-8

Placé dans un circuit programmable de type 16V8, la machine d'états ainsi créée ne fonctionne pas du tout :

– Synthétisé à la main, à partir du diagramme, avec des bascules D, elle reste obstinément arrêtée dans l'état 0, après être passée par l'état 3 lors de la mise sous tension, par les vertus de l'initialisation automatique dont dispose le circuit. Par omission des termes correspondants dans les équations de commande, tous les états oubliés, dans une synthèse qui emploie des

bascules D, sont raccordés à l'état 0. Or celui-ci, toujours par omission, est un piège dans notre exemple.
- Synthétisé par un compilateur, qui génère par défaut les équations de maintien, le système reste obstinément figé dans l'état 3, autre piège.

La deuxième version fonctionne correctement, et a, de plus, la vertu d'être décrite par des équations plus simples, que l'on aurait d'ailleurs pu trouver directement par un simple raisonnement qualitatif[14].

Le problème des pièges cachés a conduit à l'introduction, dans les langages de description, de sortes de « méta états », qui regroupent tous les non-dits, pour pouvoir préciser ce qui doit leur arriver (`else` d'un `if`, `others` d'un `case`, en VHDL). Mais, comme l'exemple précédent le montre, le raccordement de tous les états inutilisés dans un même état du diagramme, ne conduit pas toujours à la solution la plus simple.

Une approche algorithmique : VHDL

VHDL offre de multiples possibilités pour traduire le fonctionnement d'une machine d'états. Seules nous intéressent ici les descriptions comportementales, dans lesquelles le coeur d'une machine d'états est associé à un processus.

Même avec cette restriction, qui exclut les représentations structurelles, toujours possibles, le langage offre des styles de programmation variés, qui permettent de traduire simplement les situations les plus diverses.

Nous tenterons, ci-dessous, de donner certaines indications générales, qui peuvent servir de guide pour les cas les plus courants. En conformité avec ce qui a été dit au début de ce chapitre, nous ne nous occuperons que de fonctions synchrones, dont le synoptique général correspond à celui de la figure V-1.

Le registre d'état

A tout seigneur tout honneur, nous commencerons par le registre d'état.

Il est matérialisé, dans un programme source en VHDL, par deux éléments indissociables :

1. un signal interne, de type `bit_vector`, énuméré ou `integer`, déclaré de façon à être codé sur n chiffres binaires,
2. un processus, activé par le seul signal d'horloge, qui est l'*unique* endroit où le signal d'état subit une affectation.

Le choix du type employé pour le signal d'état dépend de la nature des opérations les plus fréquemment rencontrées dans le diagramme de transitions, du lien entre le registre d'état et les sorties, nous reviendrons sur ce point important, et ... du goût du concepteur. Même s'il semble plus naturel d'adopter, par exemple, un type entier pour une machine dont le fonctionnement se modélise bien par des

[14]Nous ne pouvons que conseiller au lecteur de faire, à titre d'exercice, la synthèse des exemples dont nous ne donnons pas les équations.

opérations arithmétiques, il est bon de se souvenir que les opérateurs peuvent être surchargés, pour agir sur des vecteurs de bits. Les paquetages fournis avec un compilateur contiennent déjà la plupart de ces surcharges utiles.

Faut il créer un processus à part pour la fonction combinatoire f( ), qui calcule, dans le synoptique de la figure V-1, l'état futur ? Rien n'est moins sûr.

La séparation du registre d'état et de sa commande conduit à un premier processus, qui est trivial, pour le registre d'état, et à un second processus, qui l'est beaucoup moins, pour la commande. Notons, en particulier, que des combinaisons des entrées dont on ne précise pas l'effet sur la machine génèrent,par défaut, des maintiens[15] dans la version mono processus, et des mémorisations asynchrones des commandes dans la version à deux processus séparés !

Un exemple de prototype de machine d'états qui corresponde au synoptique de la figure V-1 peut être[16] :

```
entity proto_machine is
   generic (n , p , q : integer := 2 ) ;
   port(hor : in bit ;
        entrees : in bit_vector(0 to p - 1);
        sorties : out bit_vector(0 to q - 1) ) ;
end proto_machine ;

-- suivant le compilateur utilise :
use work.paquetage_arithmetique.all ;

architecture comporte of proto_machine is
signal etat : bit_vector(n - 1 downto 0) ;
begin

machine : process
   begin
      wait until hor = '1' ;
-- ci-dessous code du diagramme de transitions.

end process machine ;

actions : process
   begin
-- ci-dessous code du calcul des sorties.

end process actions ;

end comporte ;
```

[15]Dans le diagramme de transition. Ces maintiens peuvent être voulus, auquel cas tout va bien, ou involontaires, auquel cas le résultat est faux, mais pas scandaleux. Des oublis dans la description d'un processus combinatoire conduisent à des maintiens asynchrones, ce qui est scandaleux.

[16]La clause `generic`, présentée chapitre VI, permet de rendre modifiables certains paramètres.

L'activation d'une transition, dans un diagramme d'états, dépend de l'état initial et des entrées extérieures. On peut, quitte à caricaturer un peu une réalité toujours plus nuancée, situer une machine d'états quelque part entre deux extrêmes :

- Certains automates traitent beaucoup de variables d'entrées, une ou peu de fois chacune. L'état de la machine sert essentiellement à tester dans un ordre cohérent, ces différentes entrées, à attendre, à chaque étape, une condition sur l'une ou l'autre d'entre elles et à déclencher une action, avant de passer à la suite du programme. La valeur particulière de l'état de la machine, à chaque étape, est essentielle pour déterminer la grandeur testée et le trajet suivant. Un exemple typique de fonctionnement de ce genre est un programmateur de lave linge.
- D'autres machines répondent à des commandes globales, qui provoquent des parcours, dans l'espace des états accessibles, qui peuvent être décrits indépendamment des valeurs, à chaque instant, des états. L'exemple typique d'une telle machine est un compteur. Les commandes de comptage, de chargement parallèle, de remise à zéro, entraînent une évolution qui obéit à un algorithme général, dans lequel la valeur particulière de l'état actuel n'intervient pas pour prévoir celle de l'état futur : soit que l'état futur ne dépende pas de l'état actuel, soit que la valeur de l'état futur puisse être calculée à partir de celle de l'état actuel, de façon systématique, par exemple par une opération mathématique.

Les deux discussions qui suivent correspondent à ces deux situations.

Primauté à l'état de départ

Pour décrire un diagramme de transitions en VHDL, une méthode simple consiste à traiter toutes les valeurs possibles de l'état de la machine, et pour chaque cas, analyser les entrées pour en déduire l'état suivant. L'exemple ci-dessous est la transcription, avec cette démarche, du diviseur par trois ou quatre étudié précédemment (diagramme de la figure V-6).

```
entity div3_4 is
    port ( hor , div , en : in bit ;
           Q1 , Q0 : out bit ) ;
end div3_4 ;

architecture comporte of div3_4 is
signal etat : bit_vector(1 downto 0) ;
begin
Q1 <= etat(1) ;
Q0 <= etat(0) ;
process
begin
    wait until hor = '1' ;
```

```
    case etat is -- primauté a l'etat.
        when "00" => etat <= "01" ;
        when "01" => if en = '1' then
                       etat <= "11" ;
                   end if ;
        when "10" => etat <= "00" ;
        when "11" => if div = '1' then
                       etat <= "10" ;
                     else
                       etat <= "00" ;
                     end if ;
    end case ;
end process ;
end comporte ;
```

La même fonction de principe, mais avec un nombre plus important d'états possibles, conduirait vite à une énumération d'une lourdeur prohibitive. La sélection `others` de l'instruction `when` permet, quand un traitement collectif de certains états est possible, de résoudre le problème.

Le programme qui suit correspond à un diviseur par 255/256, pour lequel on a abandonné la contrainte d'obtenir la même fréquence pour toutes les sorties, contrainte irréalisable avec un registre d'état de largeur 8 bits :

```
entity dual_modulus is
   generic (n  : integer := 8 ) ;
      -- n est la taille du registre d'etat.
   port(hor : in bit ;
        en, div : in bit ;
        sortie : out integer range 0 to 1 ) ;
end dual_modulus ;

architecture comporte of dual_modulus is
signal etat : integer range 0 to 2**n - 1 ;
begin

machine : process
begin
   wait until hor = '1' ;
   case etat is
      when 1  => -- cas particulier.
    if en = '1' then
       etat <= etat + 1;
    end if ;
      when 2**n - 2 =>
    if div = '0' then
       etat <= 0 ;
         else
       etat <= etat + 1 ;
```

```
            end if ;
         when others => -- cas general.
           etat <= etat + 1 ;
      end case ;
end process machine ;

actions : process
begin
   sortie <= etat / 2**(n-1);-- bit de poids fort.
end process actions ;

end comporte ;
```

L'exemple précédent illustre la limitation du dessin explicite d'un diagramme de transitions, dans des cas un peu complexes. Certains outils de CAO fournissent à l'utilisateur la possibilité de créer des « macro états », utiles quand une partie du diagramme peut être décrite par une formule. Donnons un exemple (figure V-9) qui correspond au diviseur par 255/256 précédent[17] :

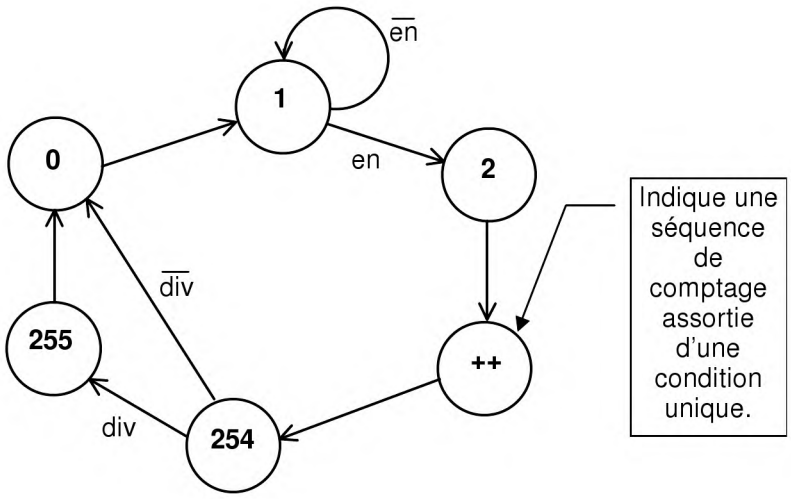


Figure V-9

Ces extensions, qui ne sont absolument pas standardisées, à la représentation traditionnelle des diagrammes de transitions permettent de représenter de façon visuelle des fonctionnement complexes, ce n'est pas à négliger.

Primauté à la commande

Les machines d'états qui disposent de commandes globales, dont les actions peuvent être décrites indépendamment de la valeur explicite de l'état, se prêtent

[17]Représentation inspirée du logiciel PLDDS, de la société Hewlett Packard.

fort mal à une description aussi détaillée que celle fournie par un diagramme de transitions. Leur description purement algorithmique peut, pourtant, être fort simple.

L'exemple ci-dessous illustre ce fait au moyen d'un compteur modulo dix, inspiré du circuit 74162, pourvu de trois commandes `clear`, `load` et `en`, dans l'ordre de priorités décroissantes :

- `clear = '0'` provoque la mise à zéro du compteur, quel que soit son état initial ;
- `load = '0'` provoque le chargement parallèle du compteur, avec des données extérieures, quel que soit son état initial ;
- `en = '1'` autorise le comptage ;
- quand toutes les commandes sont inactives, le compteur ne change pas d'état.

```
entity decade is
   port ( hor , clear, load, en : in bit ;
          donnee : in integer range 0 to 9 ;
          sortie : out integer range 0 to 9 ) ;
end decade ;

architecture comporte of decade is
signal etat : integer range 0 to 9 ;
begin
machine : process
begin
   wait until hor = '1' ;
   if clear = '0' then -- primauté aux commandes.
      etat <= 0 ;
   elsif load = '0' then
      etat <= donnee ;
   elsif en = '1' then
      case etat is -- un cas particulier.
         when 9 => etat <= 0 ;
         when others => etat <= etat + 1 ;
      end case ;
   end if ;
end process machine ;

sortie <= etat ;
end comporte ;
```

Il est clair qu'un diagramme de transitions complet d'un tel objet est pratiquement impossible à écrire : le chargement parallèle autorise des transitions entre toutes les paires d'états. L'approche algorithmique, par contre, ne pose aucune difficulté.

On notera également que la structure `if...elsif...else...end if` permet de traduire, de façon très lisible, la priorité qui existe entre les différentes commandes.

Résumons nous :

- Le processus qui décrit le fonctionnement d'une machine d'états comporte deux structures imbriquées : le traitement des commandes et le traitement de l'état de départ de chaque transition.
- Les commandes, compte tenu de leurs hiérarchies, se prêtent bien à une modélisation par des structures `if...elsif...else...end if`.
- Les états se prêtent bien à une modélisation en terme d'aiguillage, soit les structures `case...when...when others...end case`.
- Suivant le type de fonctionnement, primauté à l'état de départ ou primauté à la commande, on choisira l'ordre d'imbrication des deux structures correspondantes.

V.2.2 Des choix d'architecture décisifs

Les logiciels de synthèse libèrent le concepteur d'avoir à se préoccuper des détails des calculs qui conduisent, face à un problème posé, d'une idée de solution aux équations de commandes des circuits, déduites d'un diagramme de transitions ou d'un algorithme. Le travail de conception qui reste à sa charge réside principalement dans les choix généraux d'architectures : découpage du système en sous ensembles de taille humaine, choix de structures et de codages pour chaque sous ensemble. Ces derniers comprennent principalement le traitement des entrées – sorties et, en liaison avec les sorties, le type de codage des états.

Calculs des sorties : machines de Mealy et de Moore

Suivant la façon dont les sorties dépendent des états et des commandes, on distingue deux types de machines d'états : les machines de Moore et les machines de Mealy. Dans les premières les sorties ne dépendent que de l'état actuel de la machine, dans les secondes les sorties dépendent de l'état de la machine et des entrées.

Dans beaucoup de cas réels la séparation n'est pas aussi tranchée : certaines sorties sont traitées comme des sorties d'une machine de Moore, d'autres comme des sorties d'une machine de Mealy.

Machines de Moore

A l'image de M. Jourdain, nous avons, en réalité, fait des machines de Moore sans le savoir. Le synoptique général de la figure V-1, dans lequel les sorties sont fonctions uniquement de la valeur du registre d'état, est la définition même d'une telle machine.

Dans ce type d'architecture, le calcul des sorties et le codage des états sont évidemment intimement liés. Nous aurons l'occasion de revenir sur ce point ultéreurement.

Machines de Mealy

Dans une machine de Mealy les entrées du système ont une action directe sur les sorties, nous admettrons, dans un premier temps, que les sorties sont des fonctions purement combinatoires, symbolisées par une fonction g( ), des entrées et de l'état de la machine.

La structure générale d'une machine de Mealy est la suivante (figure V-10) :

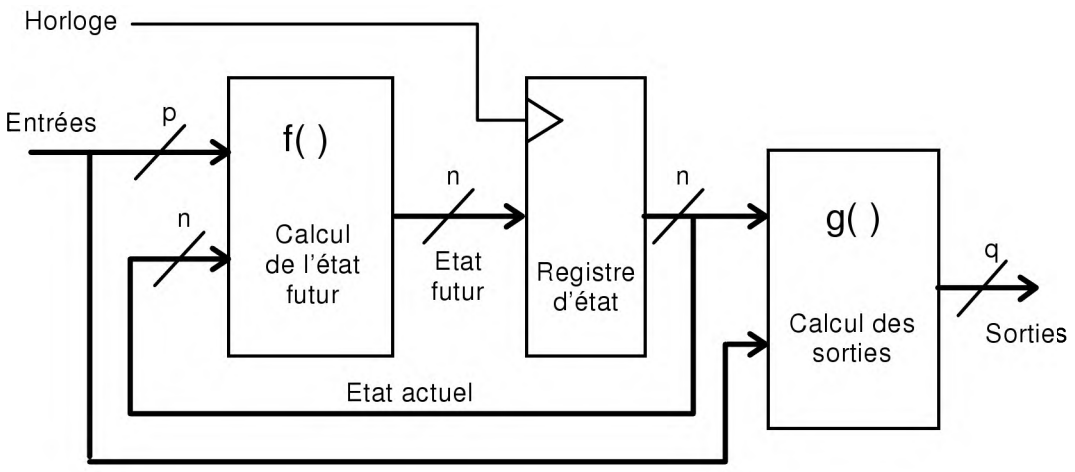


Figure V-10

Une première différence apparaît alors immédiatement entre les comportements de sorties de Moore et de Mealy : les premières évoluent suite à un changement d'état, donc à la période d'horloge qui *suit* celle où a varié l'entrée responsable de l'évolution, les secondes réagissent *immédiatement* à une variation d'une entrée, précédant en cela l'évolution du registre d'état.

Le chronogramme de la figure V-11 illustre ce point ; on y représente, en supposant le fonctionnement idéal, c'est à dire sans faire apparaître les temps de propagations :

- le changement d'une entrée com,
- un changement d'état qui en résulte, etat_i passe à 0,
- le changement associé d'une sortie de Moore, moore_i, qui passe à 1,
- le changement d'une sortie de Mealy, mealy_ou_i, calculée par

  mealy_ou_i = com + etat_arrivée,

  où etat_arrivée est l'état qui *suit* etat_i,
- le changement d'une autre sortie de Mealy, mealy_et_i, calculée par

mealy_et_i = com * etat_i.

Un point intéressant, que souligne ce chronogramme, est la possibilité de générer, par une sortie de Mealy, une impulsion qui dure une période d'horloge, indiquant qu'un changement d'état *va* se produire au front d'horloge suivant (sortie mealy_et_i)[18].

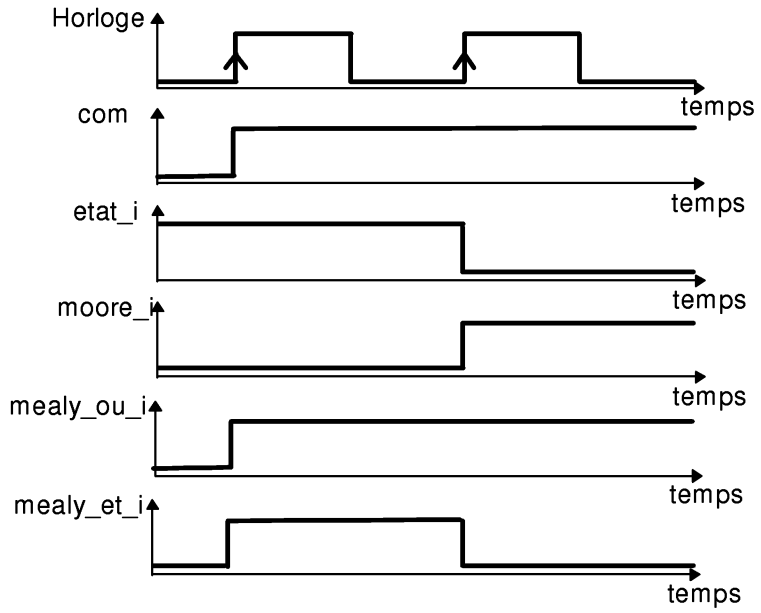


Figure V-11

Une application classique des machines de Mealy est la création d'opérateurs pourvus de sorties d'extension. Reprenons, à titre d'exemple, le compteur modulo 10 de l'exemple VHDL précédent. Il serait souhaitable de pouvoir associer simplement plusieurs de ces compteurs en cascade, de façon à réaliser un compteur sur plusieurs chiffres décimaux, un compteur kilométrique de voiture, par exemple, sans avoir à rajouter de circuiterie supplémentaire.

Pour cela il faut disposer d'une sortie, `rco` (pour *ripple carry out*), qui nous indique que la décade *va* passer à zéro, c'est à dire qu'elle est dans l'état 9 *et* qu'elle est autorisée à compter, car son entrée d'autorisation, `en`, est à un.

Mécanisme d'anticipation et influence directe d'une entrée, la sortie `rco` d'un compteur est bien une sortie de Mealy. Le programme ci-dessous contient la modification souhaitée :

[18]Il est clair que nous supposons ici que les commandes sont synchrones de la même horloge que la machine étudiée.

```
entity decade_rco is
   port ( hor , clear, load, en : in bit ;
          donnee : in integer range 0 to 9 ;
          sortie : out integer range 0 to 9 ;
     rco : out bit ) ;
end decade_rco ;
architecture comporte of decade_rco is
signal etat : integer range 0 to 9 ;
begin
machine : process
begin
   wait until hor = '1' ;
-- même code que précédemment.
end process machine ;
sortie <= etat ;
rco <= '1' when etat = 9 and en = '1' else '0';
end comporte ;
```

Pour créer un compteur à plusieurs chiffres décimaux, il suffit alors de connecter la sortie `rco` de chaque décade sur l'entrée `en` de la décade de poids *supérieur* ; il va sans dire que toutes les décades doivent être pilotées par la même horloge[19] !

Diagramme de transitions d'une machine de Mealy

Pour tenir compte de l'action immédiate des entrées sur les sorties, dans une machine de Mealy, on complète parfois le diagramme de transitions de la machine en faisant figurer, en plus de la condition de transition, la valeur associée des sorties du type Mealy.

Par exemple, pour une simple bascule R-S synchrone, mais qui « réagit » instantannément, nous obtenons (figure V-12) :

[19]Notons, au passage, un piège des sorties de Mealy : il est interdit de rajouter un rétrocouplage de la sortie `rco` sur l'entrée `en` de la même décade. Ce rétrocouplage créerait une réaction asynchrone, qui peut, par exemple, conduire à des oscillations du circuit. Dans les compteurs TTL de la famille 160, les constructeurs ont prévu deux entrées, `ent` et `enp`, d'autorisation de comptage, dont l'une, `enp`, n'a aucune action sur la sortie de mise en cascade. S'il est nécessaire, par exemple, d'inhiber le comptage en fin de cycle, c'est cette deuxième entrée de validation qui doit être employée.

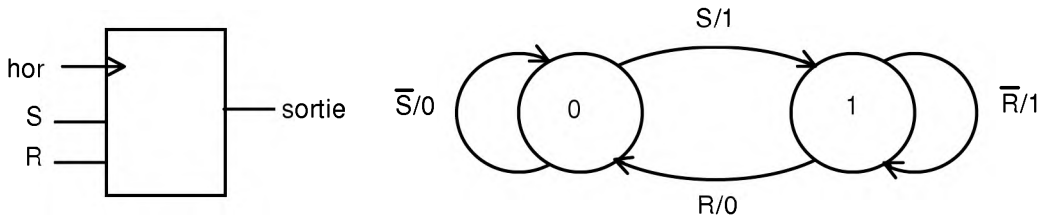


Figure V-12

Un tel diagramme se lit de la façon suivante :

Quand la bascule est à 0, la sortie est à 0 tant que l'entrée S est à 0, quand S passe à 1 la sortie passe à 1 et la bascule effectue la transition 0→1 au front d'horloge suivant ;

quand la bascule est à 1, la sortie est à 1 tant que l'entrée R est à 0, quand R passe à 1, la sortie passe à 0 et la bascule effectue la transition 1→0 au front d'horloge suivant.

De ce diagramme nous pouvons déduire l'équation de la commande, D, de la bascule, et l'équation de la sortie[20] :

$$D = \overline{Q} * S + \overline{R} * Q$$
$$\text{sortie} = \overline{Q} * S + \overline{R} * Q$$

Il se trouve que, dans cet exemple, l'équation de la sortie est identique, dans la forme mais pas dans le résultat), à celle de la commande de la bascule ; ce n'est évidemment pas toujours le cas.

Comparaison des machines de Moore et Mealy : un exemple

Afin d'illustrer les différences entre les deux types de machines d'états, nous allons donner un exemple d'application, traité par les deux méthodes.

Un décodeur Manchester différentiel.

Dans les communications séries entre ordinateurs on utilise généralement des techniques particulières de codage pour les signaux qui circulent sur le câble, par exemple, le codage *Manchester différentiel*. En codage Manchester différentiel, chaque intervalle de temps élémentaire, pendant lequel un signal binaire est placé sur le câble, nommé le plus souvent « temps bit », est divisé en deux parties de durées égales avec les conventions suivantes :

[20]On comparera utilement le fonctionnement de cette bascule R S synchrone avec celui d'une bascule J K.

- Un signal binaire 1 est représenté par une absence de transition au début du temps bit correspondant.
- Un signal binaire 0 est représenté par la présence d'une transition au début du temps bit considéré.
- Au milieu du temps bit il y a *toujours* une transition.

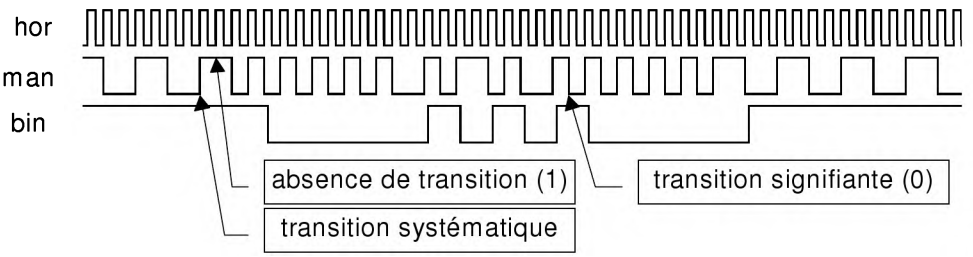


Figure V-13

Le chronogramme de la figure V-13 représente un exemple d'allure des signaux. Les données à décoder, le signal `man`, sont synchrones d'une horloge `hor`[21] ; on souhaite réaliser un décodeur qui fournit en sortie le code binaire correspondant, `bin`. La sortie du décodeur est retardée d'une période d'horloge par rapport à l'information d'entrée, pour une raison qui s'expliquera par la suite.

Analyse du problème :

L'idée est assez simple, nous allons construire une machine d'états qui, parcourt un premier cycle quand le signal d'entrée change à chaque période d'horloge, ce qui correspond à un '0' transmis, et change de cycle quand elle détecte une absence de changement du signal d'entrée, qui correspond à un '1' transmis.

Machine de Mealy :

L'absence de changement peut se produire tant pour un niveau haut que pour un niveau bas du signal d'entrée, d'où l'ébauche de diagramme de transitions de la figure V-14 (page suivante).

[21]La reconstruction par un récepteur du signal d'horloge, `hor`, n'est pas abordé ici. Les techniques employées relèvent généralement de l'analogique (boucle à verrouillage de phase).

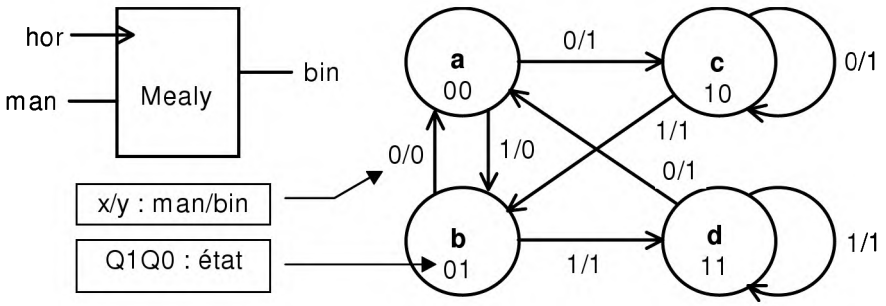


Figure V-14

On se convaincra facilement que les cycles parcourus sont :

- a→b→a→b→a→b→a... ou b→a→b→a→b→a→b... pour trois '0' consécutifs transmis,
- a→c→b→d→a→c→b... ou b→d→a→c→b→d→a... pour trois '1' consécutifs transmis.

En fonctionnement permanent, une fois le système synchronisé et sauf erreur dans le code d'entrée, les conditions de maintien dans les états c et d sont toujours fausses, elles servent à la synchronisation en début de réception.

Du diagramme précédent on déduit les équations de commandes des bascules, D1 et D0, et celle de la sortie ; après quelques simplifications on obtient :

$$D0 = man$$
$$D1 = \overline{Q0 \oplus man}$$
$$bin = Q1 + \overline{Q0 \oplus man}$$

Machine de Moore (figure V-15) :

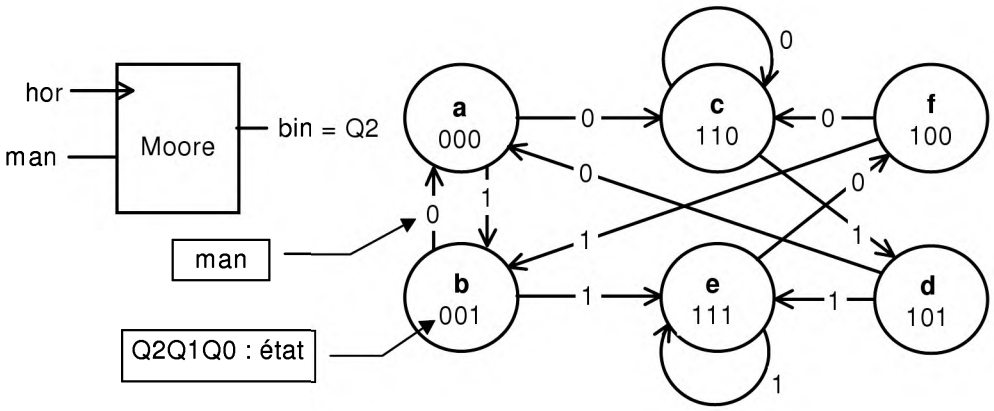


Figure V-15

Dans une machine de Moore, différentes valeurs des sorties correspondent à des états différents, le diagramme de transitions doit donc contenir plus d'états.

Le diagramme ne représente pas deux états inutilisés, 2 et 3, en décimal. Leur affectation se fait lors du calcul des équations de commandes, de façon à les simplifier au maximum (si man = '1', 3→7 et 2→5 ; si man = '0', 3→4 et 2→6). On obtient, après quelques manipulations :

$$D0 = man$$
$$D1 = \overline{Q0 \oplus man}$$
$$D2 = Q1 + \overline{Q0 \oplus man}$$

qui sont exactement les mêmes équations que celles obtenues dans le cas de la machine de Mealy[22], malgré l'apparente complexité du diagramme de transitions.

Comparaison :

Sur l'exemple que nous venons de traiter, il apparaît comme seule différence une bascule supplémentaire en sortie, pour générer le signal bin, dans le cas de la machine de Moore.

En regardant plus attentivement l'architecture des deux systèmes, on constate que la sortie combinatoire de la machine de Mealy risque de nous réserver quelques surprises : son équation fait intervenir des signaux logiques qui changent d'état simultanément, d'où des risques de création d'impulsions parasites étroites au moment des commutations. Une simulation confirme ce risque[23](figure V-16) :

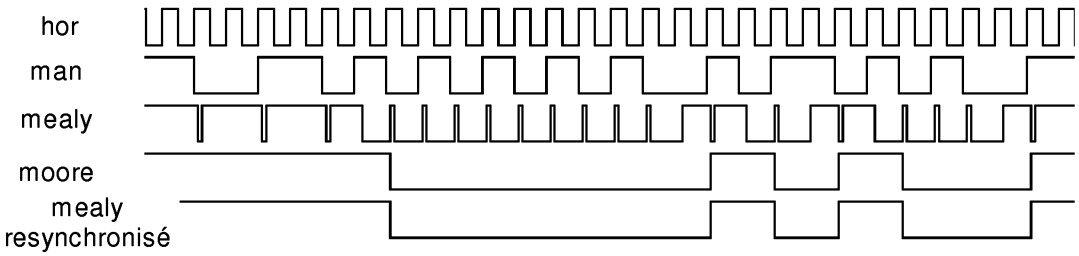


Figure V-16

[22]Soyons honnêtes, le codage des états a été choisi de façon à ce que les choses « tombent bien » ; mais, quel que soit le codage, les complexités des équations sont du même ordre de grandeur.

[23]Le simulateur utilisé est purement fonctionnel, mais causal. Chaque couche logique rajoute un temps de propagation virtuel égal à une unité (« tic » de simulation). L'allure des signaux n'a donc qu'une vertu qualitative, pour ce qui concerne les limites d'un fonctionnement.

Quand on compare les sorties des deux machines, elles diffèrent d'une période d'horloge, ce qui est normal, mais la sortie de la machine de Moore est exempte de tout parasite, contrairement à celle de la machine de Mealy, ce qui est un avantage non négligeable.

L'élimination des parasites en sortie a une solution simple : il suffit de resynchroniser la sortie incriminée, c'est ce que nous avons fait pour obtenir la dernière trace du chronogramme précédent, dans ce cas, les deux approches conduisent à des résultats strictement identiques !

Autocritique.

Pour familiariser le lecteur aux raisonnements sur les diagrammes de transitions, avec un exemple pas tout à fait trivial, nous n'avons pas respecté la règle d'or du concepteur : diviser pour régner.

En séparant le problème en deux :

1. Détection des absences de transition ;
2. génération du code binaire ;

l'élaboration des diagrammes de transitions des deux machines d'état devient un exercice extrêmement simple.

Le programme VHDL de l'étude précédente :

On trouvera ci-dessous le programme VHDL qui contient, sous forme de deux processus, les deux solutions présentées pour le décodeur Manchester différentiel.

La lecture de ce programme doit se faire en observant parallèlement les deux diagrammes d'états.

```
entity mandec is
   port ( hor, man : in bit ;
          mealy , mealysync : out bit ;
          moore : out bit );
end mandec ;

architecture comporte of mandec is
signal moore_state : bit_vector(2 downto 0);
signal mealy_state : bit_vector(1 downto 0);

begin
mealy <= mealy_state(1) or
           not(mealy_state(0) xor man) ;
moore <= moore_state(2) ;

mealy_mach : process
begin
   wait until hor = '1' ;
   mealysync <=  mealy_state(1) or
            not(mealy_state(0) xor man) ;
```

```
   case mealy_state is
when "00" => if man = '0' then
                mealy_state <= "10" ;
             else
                mealy_state <= "01" ;
             end if ;
when "01" => if man = '0' then
                mealy_state <= "00" ;
             else
                mealy_state <= "11" ;
             end if ;
when "10" => if man = '1' then
                mealy_state <= "01" ;
             end if ;
when "11" => if man = '0' then
                mealy_state <= "00" ;
             end if ;
   end case ;
end process mealy_mach ;

moore_mach : process
begin
   wait until hor = '1' ;
   case moore_state is
when O"0" => if man = '0'  -- etats en octal
              then
                moore_state <= O"6" ;
             else
                moore_state <= O"1" ;
             end if ;
when O"1" => if man = '0' then
                moore_state <= O"0" ;
             else
                moore_state <= O"7" ;
             end if ;
when O"6" => if man = '1' then
                moore_state <= O"5" ;
             end if ;
when O"7" => if man = '0' then
                moore_state <= O"4" ;
             end if ;
when O"4" => if man = '0' then
                moore_state <= O"6" ;
             else
                moore_state <= O"1" ;
             end if ;
when O"5" => if man = '0' then
                moore_state <= O"0" ;
             else
```

```
                moore_state <= O"7" ;
            end if ;
when O"2" => if man = '0' -- etat inutiles
             then
                moore_state <= O"6" ;
            else
                moore_state <= O"5" ;
            end if ;
when O"3" => if man = '0' then -- bis
                moore_state <= O"4" ;
            else
                moore_state <= O"7" ;
            end if ;
   end case ;
end process moore_mach ;
end comporte ;
```

Codage des états

Quand on utilise des circuits standard, des compteurs programmables, par exemple, pour réaliser une machine séquentielle, le codage des états du diagramme de transitions est, de fait, imposé par le circuit cible. Il en va tout autrement quand la dite machine doit être implantée dans un circuit programmable ou un ASIC. Libéré des contraintes liées à une quelconque fonction prédéfinie, le concepteur peut, à loisir, adapter le codage des états à l'application qu'il est en train de réaliser.

Le choix d'un code est particulièrement important quand on s'oriente vers la réalisation d'une machine de Moore. L'exemple du décodeur Manchester nous a appris que l'un des avantages de cette architecture réside dans la possibilité de générer les sorties directement à partir du registre d'état, donc dénuées de tout parasite lié à leur calcul. Mais, comme nous le verrons dans deux exemples, l'identification des sorties du système à celles des bascules du registre d'état ne suffit généralement pas pour définir le codage des états.

Ce choix du codage mérite une grande attention, il conditionne grandement la complexité de la réalisation, sa bonne adaptation au problème posé ; un choix judicieux conduira à un résultat simple et facilement testable, alors qu'aucun logiciel d'optimisation ne compensera des erreurs de décision à ce niveau.

Le nombre d'états nécessaires et le type de code adopté fixent, en premier lieu, la taille du registre d'état. Schématiquement, si n est la taille, en nombre de bits, du registre d'état, et N_e le nombre d'états nécessaires, ces deux nombres (entiers !) doivent vérifier la double inégalité :

$$n \leq N_e \leq 2^n$$

Si l'inégalité de gauche n'est pas vérifiée, certaines bascules sont probablement inutiles ; quand cette inégalité se transforme en égalité, on utilise un code très « dilué », une bascule par état, qui présente l'avantage de la lisibilité, mais le danger de générer en grand nombre des états accessibles inutilisés (rappelons ici qu'il y a toujours 2^n états accessibles).

Si l'inégalité de droite n'est pas vérifiée, la tentative est sans espoir ; si elle se transforme en égalité, on utilise un encodage « fort », auquel il faudra très probablement adjoindre des fonctions combinatoires de calcul des sorties ; on ne réalise pas que des compteurs binaires ou des codeurs de position absolue (code de Gray).

Les situations intermédiaires correspondent en général à des codes adaptés aux sorties.

Encodage « fort » ou code « dilué » ? En caricaturant un peu, on peut dire que les tenants de la première solution préfèrent les fonctions combinatoires, et que les seconds sont des adeptes des bascules. Il n'est pas évident, à priori, de prévoir la complexité des équations engendrées par tel ou tel code. On gagne souvent à suivre le fonctionnement « naturel » de la machine[24], et, surtout, on gagne à se souvenir que les ordinateurs, et leurs compilateurs, ne sont pas posés sur un bureau à titre de décoration ; ils permettent de voir très vite quelle est la complexité sous jacente d'un choix, sans pour celà tomber dans le BAO[25].

Codes adaptés aux sorties

L'idée qui vient naturellement à l'esprit est de choisir le codage en fonction des sorties à générer. C'est souvent la méthode la plus souple, celle qui conduit aux équations les plus faciles à interpréter, et pas forcément plus compliquées que celles que l'on obtiendrait avec d'autres codes.

Une commande de feux tricolores.

Pour satisfaire à une tradition bien établie, nous prendrons comme premier exemple une commande de feux de circulation routière.

Un passage pour piétons traverse une avenue ; il est protégé par un feu tricolore qui fonctionne à la demande des piétons : En l'absence de toute demande, les feux sont à l'orange clignotant (un nombre Tor de secondes allumés, Tor secondes éteints). Quand un piéton souhaite traverser l'avenue, il est invité à appuyer sur un bouton, ce qui provoque le déclenchement d'une séquence (vue des voitures) :

- orange fixe pendant 2*Tor secondes,
- rouge pendant Tr secondes,
- vert pendant Tv secondes, pour laisser passer le flot de voitures pendant un minimum de temps,

[24]Mais qu'est-ce que ce fonctionnement naturel ? Sa recherche est, sans doute, l'une des parties les plus intéressantes, et donc souvent difficile, du travail.

[25]Bricolage Assisté par Ordinateur.

– retour à la situation par défaut.

Profitons de cet exemple pour subdiviser la solution du problème en sous ensembles. Trois blocs fonctionnels peuvent être identifiés :

1. La commande des feux proprement dite, les sorties de trois bascules du registre d'état commandent directement l'allumage, ou l'extinction, des lampes rouge, verte et orange.
2. Une temporisation qui, suite à une commande d'initialisation, fournit les trois durées Tor, Tr et Tv.
3. Une mémorisation de l'appel des piétons, qui évite de se poser des questions concernant la durée pendant laquelle le demandeur appuie sur le bouton ; une simple pression suffit, l'appel est alors enregistré, quel que soit l'état d'avancement de la séquence de gestion des feux.

Outre les commandes des feux proprement dites, le bloc principal fournit un signal d'initialisation (cpt) à la temporisation, qui doit durer une période d'horloge[26], et un signal d'annulation (raz) de la requête, mémorisée, d'un piéton.

Les signaux d'entrée de ce bloc sont la requête (piet) et les trois indications de durée Tor Tr et Tv ; nous supposerons que ces dernières passent à '1', pendant une période d'horloge, quand les durées correspondantes se sont écoulées.

D'où le synoptique de la figure V-17 :

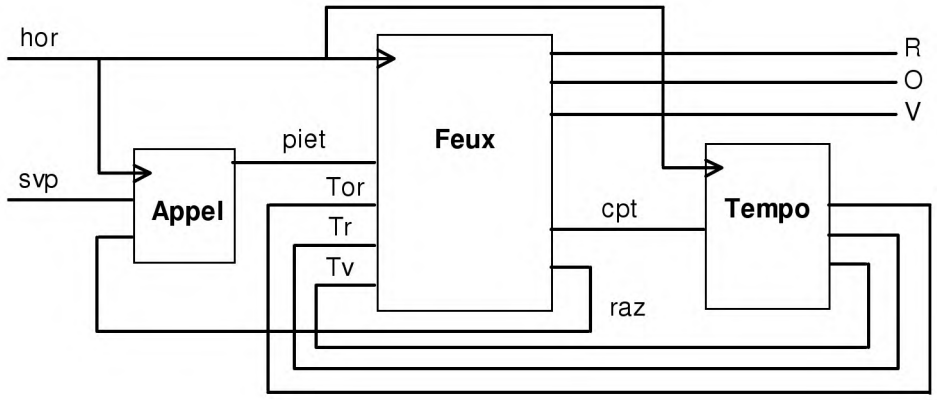


Figure V-17

[26]Nous sommes en train de définir trois processus qui se commandent et/ou s'attendent mutuellement. **Le** danger de ce type d'architecture, très fréquente, est de générer des interblocages : un processus en initialise un second et attend une réponse de ce dernier. Si le demandeur oublie de relâcher la commande d'initialisation, le système est bloqué. Ce type de situation porte, en informatique, le doux nom d'étreinte fatale (*deadly embrace*). La solution adoptée ici est d'envoyer des signaux fugaces (mais synchrones !), ce qui oblige le demandeur à attendre la réponse dans un état différent de celui où il a passé la commande d'initialisation.

Nous nous contenterons d'étudier, ici, le bloc principal, feux, laissant la synthèse des deux autres blocs à titre d'exercice.

Première ébauche :

Le fonctionnement général peut être celui illustré par la figure V-18 :

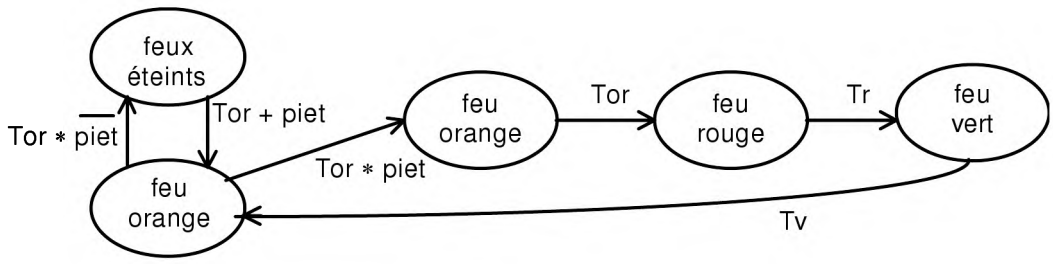


Figure V-18

Précisions :

A partir de l'ébauche précédente, il nous reste à préciser le mode de calcul des signaux gérés par le processus feux, et à en déduire le codage des états. Le signal cpt se prête bien à une réalisation sous forme de sortie de Mealy, les signaux de commande des feux à une réalisation sous forme de sorties de Moore. Les deux états où le feu orange est allumé doivent être distingués, une bascule supplémentaire, qui n'est attachée à aucune sortie, doit être rajoutée à cette fin. La sortie raz peut être identique à la sortie qui correspond au feu rouge ; il n'est pas utile de mémoriser une demande de piéton quand les voitures sont arrêtées au feu rouge. D'où une version plus élaborée du diagramme de transitions (figure V-19) :

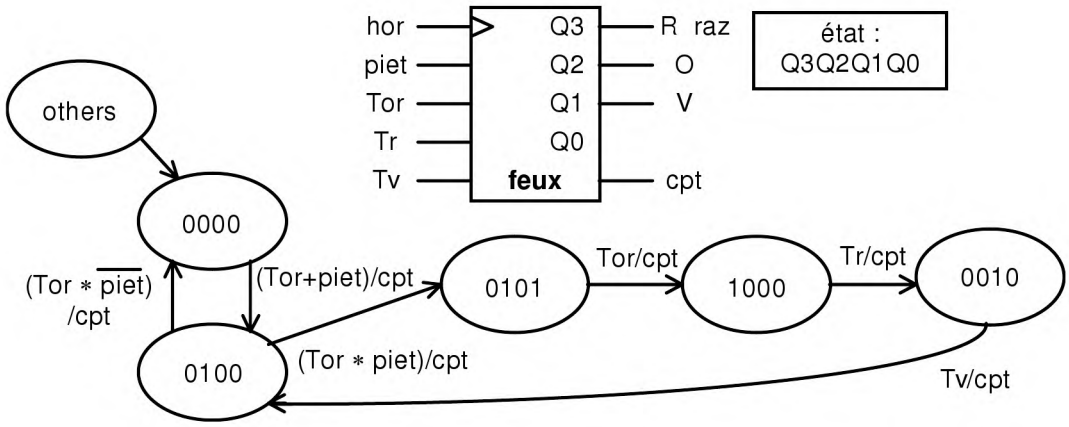


Figure V-19

Programme VHDL :

Un exemple de programme VHDL, qui correspond au module feux uniquement, est fourni ci-dessous ; il se déduit directement du diagramme de transitions précédent.

```
entity feux is
   port ( hor, piet, Tor, Tr, Tv : in bit ;
     R, O, V, cpt : out bit );
end feux ;

architecture comporte of feux is
signal etat : bit_vector(3 downto 0) ;
begin
R <= etat(3) ;
O <= etat(2) ;
V <= etat(1) ;

machine : process -- diagramme de transitions.
begin
   wait until hor = '1' ;
      case etat is
         when X"0" -- états en hexadécimal.
                  => if (Tor or piet) = '1' then
                        etat <= X"4" ;
                     end if ;
         when X"4" => if (Tor and piet) = '1' then
                        etat <= X"5" ;
            elsif (Tor and not piet) = '1' then
                        etat <= X"0" ;
                     end if ;
         when X"5" => if Tor = '1' then
                        etat <= X"8" ;
                     end if ;
         when X"8" => if Tr = '1' then
                        etat <= X"2" ;
                     end if ;
         when X"2" => if Tv = '1' then
                        etat <= X"4" ;
                     end if ;
         when others => etat <= X"0" ;
            -- pour les états inutilisés.
      end case ;
end process machine ;

mealy : process -- calcul de la sortie cpt.
begin
   wait on etat, piet, Tor, Tr, Tv ; -- liste de sensibilité
   cpt <= '0' ; -- assure un bloc combinatoire.
```

```
        case etat is
           when X"0" => if Tor = '1' or piet = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"4" => if Tor = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"5" => if Tor = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"8" => if Tr = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"2" => if Tv = '1' then
                           cpt <= '1' ;
                        end if ;
           when others => null ; -- case complet.
        end case ;
    end process mealy ;
    end comporte ;
```

Le décodeur Manchester réexaminé.

Comme deuxième exemple, reprenons, en la complétant un peu, l'étude du décodeur Manchester différentiel. Nous avions omis, dans la version précédente, un deuxième signal de sortie, rx, qui indique aux utilisateurs la cadence de transmission. Comme on peut le voir sur la figure V-20 , ce signal a une fréquence moitié de celle de l'horloge, mais il ne peut pas s'agir d'un simple diviseur par deux : un diviseur par deux est incapable de distinguer les transitions systématiques des transitions signifiantes du signal d'entrée man, il est incapable de se synchroniser.

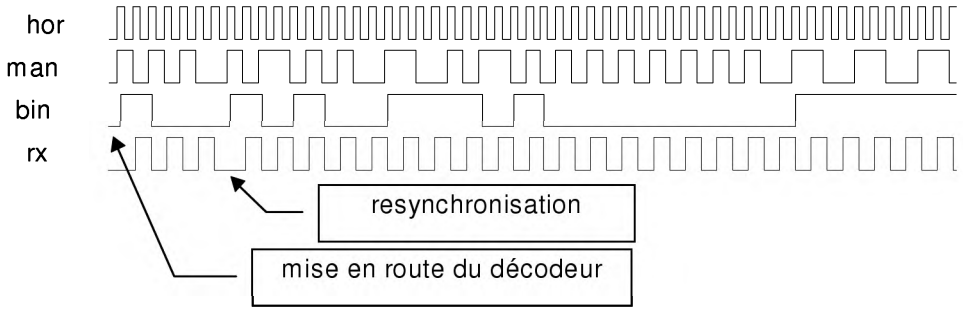


Figure V-20

Choisissons, comme précédemment, la sortie Q2 (poids fort) du registre d'état pour générer le signal bin. Pour le signal rx, il est pratique de prendre la sortie Q0 de ce registre ; une fois le décodeur synchronisé, les trajets parcourus dans le diagramme de transitions doivent être tels que la parité du code de l'état change à chaque transition : le successeur d'un nombre impair doit être pair, et réciproquement. Le diagramme de la figure V-7, étudié précédemment, ne respecte pas cette clause (transition a→c , par exemple), et ne peut pas la respecter, le nombre d'états n'étant pas suffisant (si on échangeait les codes des états a et b, par exemple, la transition e→a ne respecterait plus l'alternance de parité).

Partant du cycle c→d→e→f..., qui correspond aux transmissions de signaux binaires égaux à '1', on adjoint à ce cycle deux cycles équivalents, a→b→a...et g→h→g..., qui codent les '0' transmis, mais avec une parité inversée.

On obtient un diagramme à 8 états, qui peut, par exemple, être celui de la figure V-21.

Comme précédemment, les conditions de maintien sont, en régime établi, toujours fausses. Leur détection pourrait servir à indiquer une faute de synchronisation.

Nous laisserons au lecteur le soin de traduire ce diagramme de transitions en équations de commandes des bascules, et en programme VHDL, ce qui ne pose guère de difficulté.

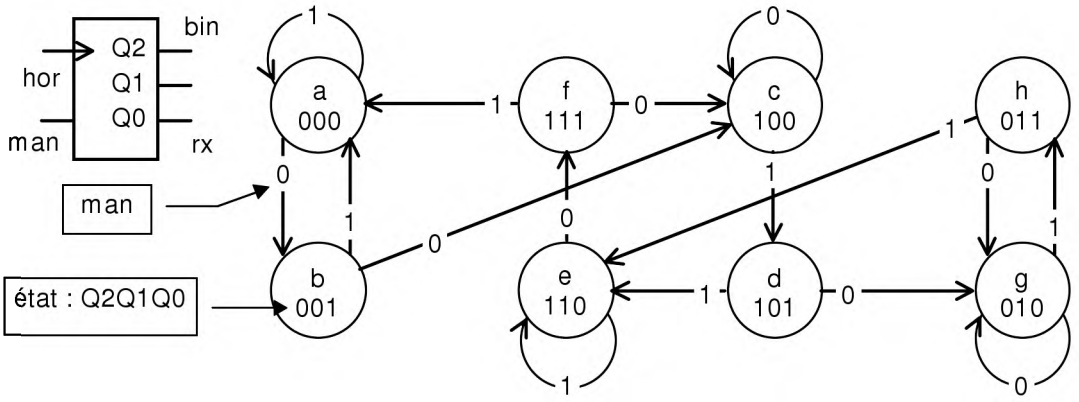


Figure V-21

Bascules enterrées.

Dans les deux exemples précédents, certaines bascules servent de sorties, d'autres ne servent qu'aux états internes. De telles bascules sont dites bascules *enterrées* (*burried flip flop*). De nombreux circuits programmables offrent la possibilité d'utiliser des bascules enterrées ; cela a l'avantage de diminuer, pour une complexité de circuit donnée, le nombre de broches d'accès nécessaires. Il est

clair, cependant, que ces circuits sont plus délicats à tester : les états ne sont pas tous visibles en sortie.

Codes « un seul actif »

Les codes dits un seul actif (*one hot*), sont les plus dilués : à chaque état on attribue une bascule ; la machine étant, par définition, dans un seul état à la fois, si l'une des bascules est active, toutes les autres sont inactives. La commande de feux, étudiée au paragraphe précédent, serait une commande de ce type si on n'avait pas eu la fantaisie d'y rajouter une bascule enterrée[27].

Code binaire

C'est le code classique des compteurs, nous l'avons rencontré, par exemple, à l'occasion du diviseur à double rapport de division 255/256.

C'est typiquement le code que l'on obtient quand on réalise des machines d'états avec des fonctions standard.

Codes adjacents

On dit qu'un code est adjacent, dans le cas d'une machine d'état, si pour toutes les transitions du diagramme d'états, le changement de valeur du registre d'état ne porte que sur un chiffre binaire.

Très en vogue quand on synthétisait des automates asynchrones, ces codes ont perdu de l'importance avec la généralisation des techniques synchrones. La contrainte que représente le respect de l'adjacence, pour tous les états successifs, devient rapidement très difficile à observer.

On peut, malgré tout noter que, si cela ne complique pas, par ailleurs, le problème, c'est souvent une bonne idée de respecter l'adjacence dans les transitions, au moins partiellement.

Les états inutilisés

Tous les codes qui n'occupent pas la totalité des états accessibles génèrent des états inutilisés. Notre feu rouge de tout à l'heure, par exemple, utilisait cinq des seize états disponibles. Le non raccordement des états inutilisés dans l'un des états du cycle relèverait, dans ce cas de la roulette russe, mais avec les deux tiers des logements du barillet du révolver chargés.

Si on n'est pas certain que les états inutilisés rejoignent naturellement l'un des états utiles du diagramme de transitions, il faut obligatoirement leur adjoindre une transition qui les ramène dans un territoire connu, faute de quoi on risque de créer une machine qui se « plante » à la première occasion.

---

[27]En l'occurence, un code *one hot zero*, car l'état où toutes les bascules sont à zéro fait partie du code. S'il y a toujours une bascule active, la combinaison « zéro » n'est pas dans le code. On parle parfois, dans ce cas, de code *one hot one*.

Les états équivalents

Lors de la première ébauche d'un diagramme de transitions, il peut arriver que l'on crée des états inutiles. Cela n'est pas, en soi, dramatique, mais il peut être intéressant de les rechercher quand, notamment, l'économie d'un ou deux états permet de réduire la taille du registre d'état.

Quand deux états sont ils équivalents ?
Quand ils génèrent les mêmes sorties et les mêmes valeurs futures des sorties, quelles que soient les séquences d'entrée.

Derrière cette définition, fort simple en apparence, se cache parfois une grande difficulté de mise en pratique de cette recherche.
Un cas particulier simple à identifier se rencontre assez souvent sur des diagrammes de transitions de dimension raisonnable : deux états fournissent les mêmes sorties et ont les mêmes états futurs, ils sont alors équivalents, on peut supprimer l'un d'entre eux.

Synchronisations des entrées et des sorties

Nous n'envisageons que la réalisation des machines d'états synchrones. Cela veut dire qu'au niveau local toutes les bascules qui interviennent sont pilotées par une horloge unique. Il est clair qu'au niveau d'un système cette règle du synchronisme absolu est rarement observée, elle nuirait à la modularité des sous ensembles.
Lors des échanges entre sous ensembles pilotés par des horloges différentes, ou pour des interfaces avec un monde extérieur qui ne possède pas d'horloge du tout, un piéton, par exemple, la question de la synchronisation des signaux d'entrée et de sortie se pose.

Synchronisations des entrées

Les signaux d'entrée qui proviennent d'un autre sous ensemble, piloté par une horloge différente, ou totalement asynchrone, doivent *toujours* être resynchronisés au moyen de bascules (voir paragraphe II-3 pour plus de précision).

Synchronisations des sorties

Les sorties calculées par des fonctions logiques combinatoires génèrent des impulsions parasites, nous en avons vu un exemple avec le décodeur Manchester. Totalement inoffensives si elles sont exploitées par un système piloté par la même horloge, ces impulsions peuvent être fort mal acceptées par un récepteur asynchrone de l'horloge locale.
Des bascules de synchronisation des sorties suppriment ce défaut, elles assurent des signaux stables entre deux fronts d'horloge.
L'adjonction pure et simple de bascules en sortie, à la place du bloc de calcul des sorties du synoptique de la figure V-1, retarde d'une période d'horloge les

valeurs des sorties par rapport à celles du registre d'état. Si ce retard présente un inconvénient, il est toujours possible, quitte à alourdir la partie combinatoire de la réalisation, d'anticiper le calcul des sorties en utilisant pour leur calcul l'état futur au lieu de l'état actuel. Le synoptique de la figure V-1 devient alors (figure V-22) :

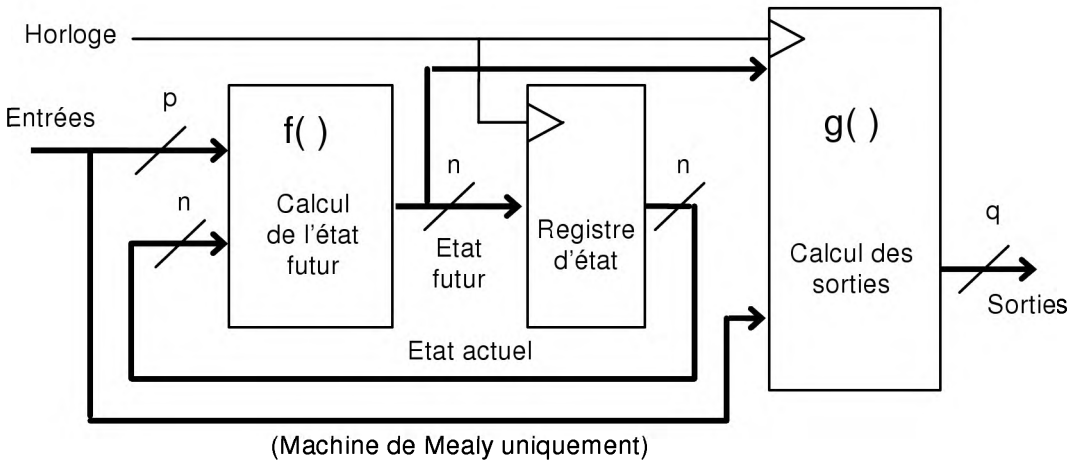


Figure V-22

V.3. Fonctions combinatoires

Dans toute réalisation d'un système numérique interviennent des fonctions combinatoires ; nous en avons déjà fait grand usage, sans avoir ressenti la nécessité de formaliser les choses au delà de quelques définitions élémentaires de l'algèbre de Boole. Dans les applications pratiques, ces fonctions combinatoires, bien que ne créant guère de difficulté de principe, sont la source principale de complexité du schéma électrique obtenu.

Sans y attacher une importance exagérée, le concepteur se doit de connaître quelques principes généraux, qui interviennent dans la manipulation de ces fonctions, ne serait-ce que pour comprendre les documents techniques qui accompagnent les circuits et les logiciels d'aide à la synthèse.

Après deux définitions classiques concernant les écritures standard des fonctions, nous aborderons rapidement les principes utilisés pour minimiser les équations correspondantes.

V.3.1 Des tables de vérité aux équations : les formes normales

Etant donnée une fonction $f(e_{n-1}, e_{n-2}, ...,e_0)$, elle est complètement spécifiée par la donnée des 2^n valeurs f(0, 0, ...,0), f(0, 0, ...,1) ...., f(1, 1, ...,1), qui sont les

éléments de sa table de vérité. Mais cette forme de représentation est difficilement manipulable.

Aussi préfère-t-on généralement la décrire par une expression polynômiale qui fait intervenir les opérateurs fondamentaux de l'algèbre de Boole que sont la réunion (ou), l'intersection (et) et la complémentation (non). On distingue traditionnellement deux formes, équivalentes, d'écriture standard pour une fonction logique, nommées tout simplement première et deuxième formes normales (ou canoniques).

Première forme normale : une somme de produits

La première forme normale est la plus couramment utilisée : elle consiste à écrire une fonction combinatoire sous forme de somme de produits (réunion d'intersections) :

$$\begin{aligned} f(e_{n-1}, e_{n-2}, \ldots, e_0) = {} & f(0, 0, \ldots, 0) * \overline{e_{n-1}} * \overline{e_{n-2}} * \ldots * \overline{e_0} \\ & + f(0, 0, \ldots, 1) * \overline{e_{n-1}} * \overline{e_{n-2}} * \ldots * e_0 \\ & + \cdots\cdots\cdots \\ & + f(1, 1, \ldots, 1) * e_{n-1} * e_{n-2} * \ldots * e_0 \end{aligned}$$

Chaque terme de cette somme logique s'appelle un *minterme*.

Dans ce développement polynômial, seuls restent les termes qui correspondent à une valeur, dans la table de vérité de la fonction, égale à '1'.

Un opérateur ou exclusif, par exemple, s'écrit, dans cette représentation :

$$\begin{aligned} a \oplus b = {} & (0 \oplus 0) * \bar{a} * \bar{b} + (0 \oplus 1) * \bar{a} * b \\ & + (1 \oplus 0) * a * \bar{b} + (1 \oplus 1) * a * b \\ a \oplus b = {} & a * \bar{b} + \bar{a} * b \end{aligned}$$

Cette écriture correspond à une phrase du type :

« a ou exclusif b égale un si
a égale 1 *et* b égale 0,
ou si
a égale 0 *et* b égale 1. »

La grande majorité des circuits programmables ont une architecture interne qui reproduit, en trois couches logiques (non – et – ou), un développement en première forme normale.

Notation condensée

Pour alléger la notation, on désigne parfois chaque minterme par un symbole, m_i, où i représente le numéro de la ligne de la table de vérité correspondante.

Par exemple :

$$m_0 = \overline{e_{n-1}} * \overline{e_{n-2}} * \cdots * \overline{e_0} \quad , \quad m_1 = \overline{e_{n-1}} * \overline{e_{n-2}} * \cdots \overline{e_1} * e_0$$

Une fonction s'écrit alors :

$$f(e_{n-1}, e_{n-2}, \ldots e_0) = \sum_{f(ei)=1} m_i$$

ou, encore plus simplement :

$$f = \sum m(i_1, i_2 \ldots . i_p)$$

où les numéros i_k indiquent simplement les numéros des lignes de la table de vérité dans lesquelles la fonction vaut '1'.

Par exemple : $a \oplus b = m_1 + m_2 = \sum m(1,2)$

Deuxième forme normale : un produit de sommes

La même fonction peut être écrite sous forme de produit (logique) de sommes (logiques), on parle alors de deuxième forme normale, ou canonique :

$$\begin{aligned} f(e_{n-1}, e_{n-2}, \ldots, e_0) = {} & (f(0, 0, \ldots, 0) + e_{n-1} + e_{n-2} + \ldots + e_0) \\ & * (f(0, 0, \ldots, 1) + e_{n-1} + e_{n-2} + \ldots + \overline{e_0}) \\ & * \cdots\cdots\cdots \\ & * (f(1, 1, \ldots, 1) + \overline{e_{n-1}} + \overline{e_{n-2}} + \ldots + \overline{e_0}) \end{aligned}$$

Pour une raison dont nous lèverons le mystère un peu plus loin, chaque facteur de ce produit logique porte le nom de *maxterme*.

On notera que la règle d'association entre les valeurs des variables, dans l'écriture de la fonction, et la forme directe ou complémentée avec laquelle interviennent ces variables dans le développement diffère par rapport à la première forme normale : à un '0' on associe la variable elle même, et à un '1' son complément. Cette inversion de règle est une source d'erreurs fréquentes quand on utilise la deuxième forme normale. Aussi conseillerons nous vivement au lecteur la prudence :

1. Utiliser toujours la même forme normale, de préférence la première.
2. S'il s'avère nécessaire d'obtenir le développement en deuxième forme, la méthode la plus sûre consiste à écrire le complément de la fonction cherchée en première forme normale, puis d'appliquer les théorèmes de De Morgan pour obtenir le résultat souhaité.

Un opérateur ou exclusif, par exemple, s'écrit, dans cette représentation :

$$a \oplus b = ((0 \oplus 0) + \bar{a} + b) * ((0 \oplus 1) + \bar{a} + \bar{b}) * ((1 \oplus 0) + \bar{a} + b) * ((1 \oplus 1) + \bar{a} + \bar{b})$$
$$a \oplus b = (a + b) * (\bar{a} + \bar{b})$$

La deuxième forme normale correspond à une phrase du type :

a ou exclusif b égale un si
    a égale 1 *ou* b égale 1,
et si
    a égale 0 *ou* b égale 0.

Notation condensée

Pour alléger la notation, on désigne parfois chaque maxterme par un symbole, M_i, où i représente le numéro de la ligne de la table de vérité correspondante.
Par exemple :

$$M_0 = e_{n-1} + e_{n-2} + \cdots + e_0, \quad M_1 = e_{n-1} + e_{n-2} + \cdots + e_1 + \bar{e}_0$$

Une fonction s'écrit alors :

$$f(e_{n-1}, e_{n-2}, \ldots e_0) = \prod_{f(ei)=0} M_i$$

ou, encore plus simplement :

$$f = \prod M(i_1, i_2 \ldots . i_p)$$

où les numéros i_k indiquent simplement les numéros des lignes de la table de vérité dans lesquelles la fonction vaut '0'.

Par exemple : $a \oplus b = M_0 + M_3 = \prod M(0,3)$

V.3.2 L'élimination des redondances : les minimisations

Quand on exprime une fonction combinatoire, directement à partir du résultat à obtenir, il arrive presque toujours qu'il y ait des redondances dans les équations obtenues.

Prenons un exemple.

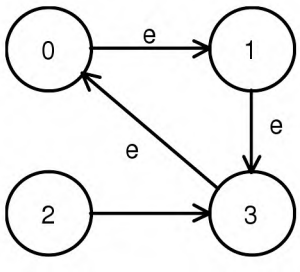


Figure V-23

Soit à réaliser un compteur modulo 3, qui, en fonction d'une entrée e, obéisse au diagramme de transitions de la figure V-23 :

- Quand e = '1', le compteur s'incrémente, sinon il reste dans l'état.
- L'état 2 a été raccordé dans le cycle, pour éviter tout risque de « piège ».

Les équations de ce système s'obtiennent immédiatement, avec deux bascules D nous obtenons :

$$D1 = \overline{Q1} * Q0 * e + Q1 * Q0 * \overline{e} + Q1 * \overline{Q0}$$

$$D0 = \overline{Q1} * \overline{Q0} * e + \overline{Q1} * Q0 * \overline{e} + \overline{Q1} * Q0 * e + Q1 * Q0 * \overline{e} + Q1 * \overline{Q0}$$

Avec un peu de réflexion on voit apparaître des simplifications, entre le deuxième et le troisième terme de l'expression de D0, par exemple, la variable e se simplifie. Le problème pratique qui se pose est de trouver une méthode systématique de recherche de telles simplifications. Ce point fait l'objet des paragraphes suivants.

Quelques remarques préalables s'imposent :

- Les simplifications algébriques, qui sont celles qui nous occupent ici, ne garantissent en aucun cas le nombre minimum d'opérateurs élémentaires pour réaliser une fonction. L'exemple présenté du contrôleur de parité, paragraphe III-2, en est un exemple flagrant. Ces simplifications fournissent une forme minimale des expressions dans une construction en trois couches, similaire à l'une des deux formes canoniques. Elles représentent donc un optimum dans lequel la vitesse de transfert du circuit réalisé (nombre de couches) intervient en premier, le nombre d'opérateurs, donc la surface de silicium occupée, intervenant en second. Le concepteur est souvent amené à accepter une certaine forme de compromis, dans les cas pratiques.
- Malgré les restrictions précédentes, il serait faux de croire que les simplifications sont inutiles, une optimisation passe de toute façon par des minimisations des sous-ensembles que les contraintes de surface occupée auront défini.
- Les « compilateurs de silicium » ont grandement changé les données du problème à plusieurs niveaux :

  1. Ils génèrent, à partir des langages de haut niveau des constructions extrêmement lourdes, où intervient, schématiquement, une couche de

multiplexeurs pour chaque niveau d'imbrication des instructions de tests (if, case, ... etc). Sans optimisation des équations générées un langage de haut niveau serait inutilisable[28].

2. Ils permettent de gérer des fonctions à grand nombre de variables d'entrée difficilement calculables à la main. La moindre machine d'états, pour laquelle on a adopté un codage dirigé par les sorties, donc relativement dilué, génère des fonctions qui découragent vite, par le nombre de variables mises en jeu, les calculs manuels.
3. Des deux points précédents résulte le fait que l'on ne fait plus jamais les calculs de simplifications entièrement à la main ; tous les compilateurs disposent de programmes de minimisation des expressions logiques. Par contre, il est essentiel que l'utilisateur de tels programmes domine le principe de ce qu'ils font, même si la conception de ces programmes n'est pas de son ressort.

Nous ne nous lancerons donc pas dans des calculs sur des fonctions de plus de quatre variables, l'essentiel étant de comprendre le principe de ces calculs, non de les étendre à des cas réels. Dans tout ce qui suit nous nous limiterons à des exemples qui font intervenir la première forme canonique d'une fonction.

Simplifications algébriques directes

Une variable d'entrée x est l'objet d'une simplification si, dans l'écriture de la fonction, apparaissent deux termes de la forme :

$$f(a, \ldots t, x, y, \ldots) = \ldots + x * f_x(a, \ldots t, y, \ldots) + \bar{x} * f_x(a, \ldots t, y, \ldots) + \ldots$$

Les deux mintermes concernés sont dits adjacents, on passe de l'un à l'autre en changeant la variable x uniquement.

La fonction se simplifie en :

$$f(a, \ldots t, x, y, \ldots) = \ldots + f_x(a, \ldots t, y, \ldots) + \ldots$$

Au delà de trois variables, et encore, la difficulté est de repérer les mintermes qui vont intervenir dans des simplifications. Cette difficulté est accentuée par le fait que le même minterme peut intervenir dans plusieurs simplifications différentes, auquel cas il faut le « dupliquer » avant de simplifier effectivement, c'est ce qui se produit dans l'expression suivante :

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

[28]La même remarque peut être faite à propos des langages de programmation. Si certains pensent encore que la programmation en langage machine est plus efficace, c'est probablement en raison de la faible efficacité des premiers compilateurs disponibles. Les temps ont changé.

Cette difficulté de repérage limite grandement les calculs algébriques directs, d'où la méthode graphique des tableaux de Karnaugh.

Les tableaux de Karnaugh

Définition

Les tableaux de Karnaugh, une variante des tables de vérité, sont organisés de telle façon que les mintermes adjacents soient systématiquement regroupés dans deux cases voisines, donc faciles à identifier visuellement.

On représente les valeurs de la fonction dans une table à deux dimensions, organisée comme un damier, dans laquelle chaque ligne et chaque colonne correspond à une combinaison des variables d'entrée. Le codage des lignes et des colonnes est fait dans un code adjacent, ou code de Gray (figure V-24).

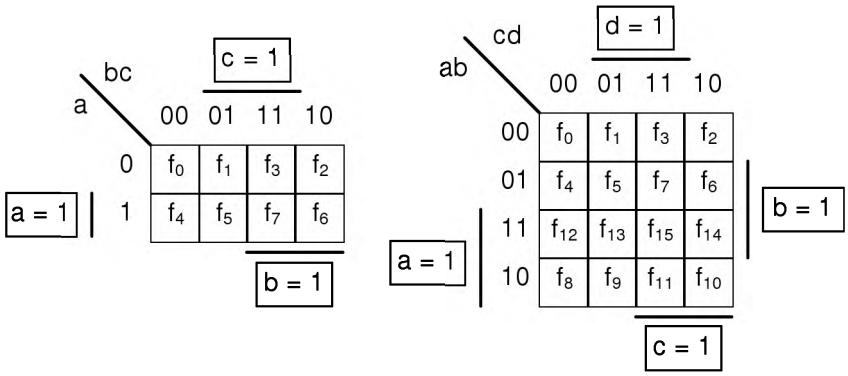


Figure V-24

Chaque minterme correspond à une cellule élémentaire de la table, qui constitue le plus petit groupement possible, d'où le nom. De même, un maxterme correspond à toutes les cellules sauf une égales à '1', soit le recouvrement de taille maximum qui ne soit pas trivial.

Cette disposition est telle que si le développement en première forme normale de la fonction contient deux mintermes adjacents, les deux '1' correspondant de la table de vérité se retrouvent dans des cases voisines.

On notera que les cases d'extrémité d'une même ligne; ou d'une même colonne, sont voisines.

Une simplification, qui est un regroupement de mintermes adjacents, apparaît alors comme une regroupement de plusieurs cases voisines. On notera qu'une simplification fait toujours intervenir un nombre de termes qui est une puissance de deux (2, 4, 8, ...).

Exemple

Reprenons les équations du compteur modulo 3 précédent. Le report des valeurs des entrées D1 et D0 des deux bascules, en fonction de Q1, Q0 et e, conduit aux tableaux de Karnaugh de la figure V-25 :

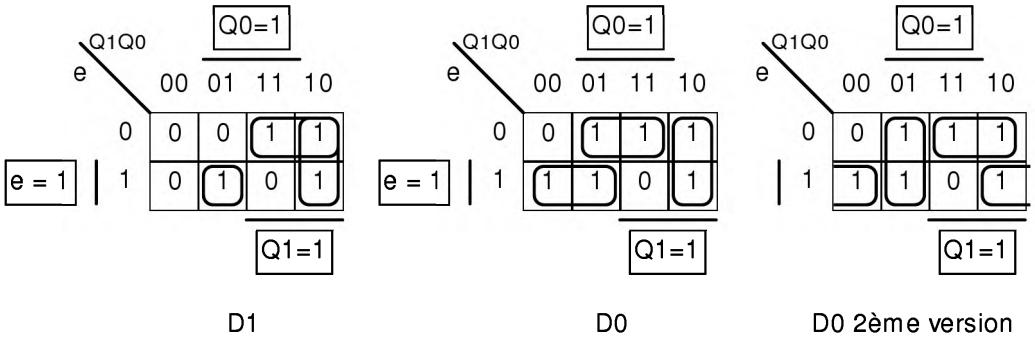


Figure V-25

La figure précédente met en évidence le fait que la solution n'est pas toujours unique, mais si deux solutions sont possibles, elles ont la même complexité.

Des diagrammes précédents nous déduisons les équations (1ère version de D0) :

$$D1 = \overline{Q1} * Q0 * e + Q1 * \overline{e} + Q1 * \overline{Q0}$$
$$D0 = Q0 * \overline{e} + Q1 * \overline{Q0} + \overline{Q1} * e$$

Nous ne pouvons que conseiller au lecteur de reprendre les exemples de machines d'états que nous avons étudiées, et d'en déduire les équations de commandes au moyen de tableaux de Karnaugh.

Fonctions incomplètement spécifiées

Il arrive, souvent, en fait, que les données du problème laissent non spécifiées les valeurs des sorties pour certaines combinaisons des variables d'entrées. Dans ce cas, le concepteur peut choisir des valeurs à sa convenance, avec prudence comme nous allons le voir, pour tenter de minimiser les équations qui en résultent. Une valeur non spécifiée, par le cahier des charges, est traditionnellement notée ϕ dans

un tableau de Karnaugh. Il est important de noter que, non spécifiée initialement, cette valeur sera bien déterminée une fois les équations choisies !

Reprenons, à titre d'illustration, les équations du décodeur Manchester, sous forme de machine de Moore, dont le diagramme de transitions à 6 états a été représenté à la figure V-15.

Les états 2 et 3 ne figurent pas sur le diagramme, nous pouvons donc les raccorder dans le cycle à notre guise, de façon à diminuer la complexité des équations générées pour les commandes des trois bascules.

Nous obtenons, en notant m pour l'entrée man, les tableaux de la figure V-26.

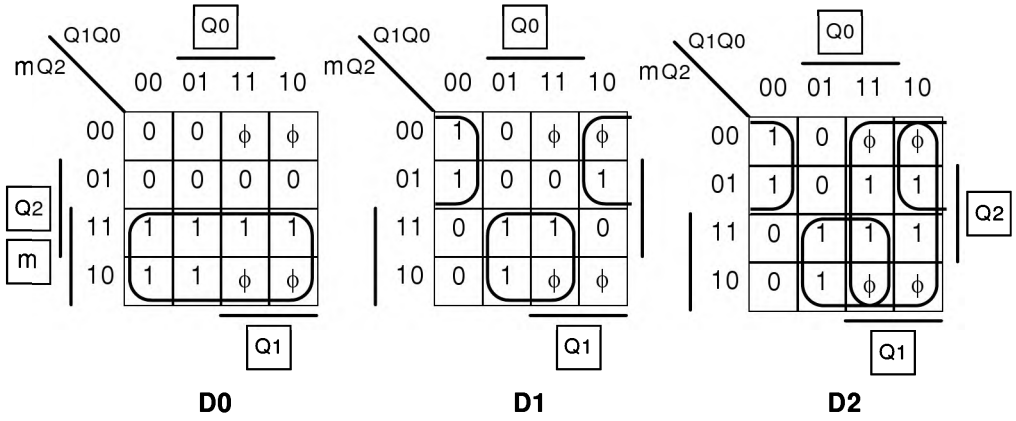


Figure V-26

D'où les équations précédemment données sans justification :

$$D0 = man$$
$$D1 = \overline{Q0 \oplus man}$$
$$D2 = Q1 + \overline{Q0 \oplus man}$$

On notera qu'il est essentiel, quand on a utilisé des valeurs non spécifiées, de contrôler à postériori, quand on connait les valeurs attribuées aux « ϕ », que l'on n'a pas généré de piège dans le diagramme de transitions.

L'élimination des parasites de commutation

Outre les simplifications, les tableaux de Karnaugh permettent de prévoir si, lors des changements des valeurs des entrées, on risque de créer des impulsions parasites, ce que l'on appelle des aléas statiques.

Un aléa statique se manifeste, par exemple, par la présence d'un '0', de durée brève, lié aux temps de propagation dans les circuits, lors du changement d'une entrée telle que la fonction vaut '1' avant et après le changement.

La règle est simple : pour assurer l'absence d'aléa statiques, il faut qu'il y ait des recouvrements partiels des groupements du tableau de Karnaugh, de sorte que quand on passe d'un groupement à un autre, on ne franchisse qu'une seule frontière.

Prenons comme exemple un multiplexeur (figure V-27) :

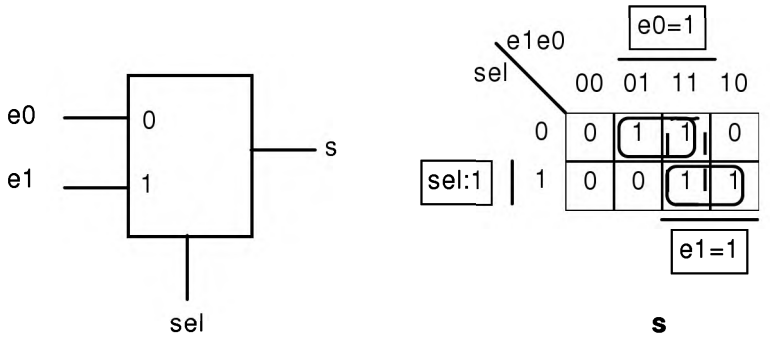


Figure V-27

Les deux groupements en traits pleins ne se recouvrent pas, quand les deux entrées sont à '1', et que l'on change la valeur de la commande de sélection, on risque d'obtenir un parasite à '0' en sortie. Ce parasite est éliminé par l'adjonction du groupement indiqué en pointillé, qui assure la continuité du pavage. D'où l'équation d'un multiplexeur « sans aléa » :

$$s = e0 * \overline{sel} + e1 * sel + e0 * e1$$

Il est essentiel d'utiliser un tel multiplexeur pour réaliser des bascules, si non la bascule mémorise justement l'aléa ! Dans les circuits programmables qui utilisent, comme cellules élémentaires des multiplexeurs, ceux-ci sont garantis sans aléa, de façon à pouvoir réaliser sans risque des bascules.

Les logiciels de minimisation

Comme nous l'avons mentionné, tous les compilateurs sont assortis d'un programme de minimisation des équations logiques générées. Il est hors de question, ici, de rentrer dans les détails de ces programmes, qui sont loin d'être triviaux. Nous nous contenterons de citer deux algorithmes, parmi les plus connus[29].

[29]Pour les lecteurs intéressés, voir : HILL F.J. et PETERSON G.R., *Computer aided logical design with emphasis on VLSI*, John WILEY, New York, 1993.

La méthode de Quine-McCluskey

L'algorithme de Quine-McCluskey, qui date de 1956, utilise, comme les tableaux de Karnaugh, mais d'une façon systématique, et non visuelle, des tables qui décrivent tous les mintermes possibles d'une fonction.

Comme toute méthode tabulaire, tableaux de Karnaugh compris, la taille des données à manipuler croît exponentiellement avec le nombre de variables d'entrées.

Cette croissance exponentielle rend vite impraticables, même sur ordinateur, ces méthodes dès qu'il s'agit de traiter des fonctions à grand nombre de variables d'entrées.

L'avènement des compilateurs de silicium a rendu nécessaire la création d'algorithmes qui, même s'ils ne garantissent pas un optimum absolu, laissent espérer une simplification importante, avec un algorithme qui ne soit pas exponentiel.

Espresso

Au lieu de partir d'une table des mintermes, l'algorithme Espresso (1984) manipule l'expression algébrique d'une fonction, en tentant de la transformer, de proche en proche, pour aboutir à une expression plus simple. La complexité de l'algorithme dépend de la complexité réelle de la fonction plus que du nombre de variables d'entrées.

Espresso ne garantit pas d'arriver à une expression minimale absolue ; l'algorithme peut s'achever dans des situations de minimums locaux, dont il n'arrive pas à sortir[30]. Des tests effectués ont montré, cependant, que pour des fonctions de plus de 25 variables d'entrées, il est arrivé à des résultats égaux, ou plus complexes d'un seul terme, qu'un algorithme systématique. Et ce, pour un temps de calcul plus faible dans un rapport 10 à 100 suivant les fonctions analysées.

La plupart des systèmes de CAO utilisent cet algorithme pour les calculs de minimisation d'expressions logiques[31].

---

[30]Schématiquement, le problème est que, même quand on a trouvé un recouvrement complet de groupements dans un tableau de Karnaugh, il peut correspondre à un minimum local. Le fait de réutiliser plusieurs fois des mintermes, qui ont peut être disparu lors de simplifications précédentes, permet parfois de construire un recouvrement complètement différent de celui dont on est parti, qui aboutit à un résultat plus simple. Autant ce genre de choses saute aux yeux d'un humain, qui a une vision globale de la situation, autant il est difficile de formaliser cette démarche. Espresso retire, au fur et à mesure de ses calculs les termes qu'il est sûr de devoir garder, et commence par « désimplifier » ce qui reste de la fonction pour chercher un autre recouvrement. Il applique cette démarche de façon récursive jusqu'à ce qu'il cesse de progresser.

[31]Il n'est jamais arrivé aux auteurs de rencontrer une fonction, humainement analysable, où le résultat manuel soit meilleur que celui d'espresso. Il est vrai que nous ne pratiquons plus guère les tableaux de Karnaugh de grandes dimensions.

V.4. Séquenceurs et fonctions standard

Pendant deux décennies la plupart des machines d'états furent réalisées au moyen de fonctions logiques standard, compteurs programmables, multiplexeurs et décodeurs principalement.

V.4.1 Séquenceurs câblés

Dans un séquenceur câblé les équations logiques du système sont matérialisées par un câblage figé entre les différents constituants.

Nous nous contenterons ici de donner l'architecture générale d'un tel système, et d'indiquer la méthode qui permet de passer d'un diagramme de transitions aux commandes des circuits.

Architecture

Le noyau d'une machine d'états, qui utilise des fonctions standard, est généralement un compteur à commandes de chargement parallèle et de remise à zéro, *synchrones*, cela va sans dire, mais disons le tout de même.

Les sorties du compteur pilotent les entrées d'adresses de multiplexeurs, par exemple, qui calculent, en fonction des entrées extérieures et de l'état actuel de la machine, les commandes à appliquer au compteur.

Ces sorties sont éventuellement décodées pour générer les sorties du séquenceur. Cela correspond au synoptique de la figure V-28.

Quand on utilise une telle architecture, il est clair que le codage du diagramme de transitions doit être adapté au circuit choisi : on tente de privilégier les séquences de comptage, en limitant au maximum les « sauts » par rapport au code binaire naturel du compteur. Les blocs logiques de calcul des commandes du compteur sont d'autant plus simples qu'il y a des séquences de comptage régulières, qu'il y a le moins possible d'adresses de rupture de séquence différentes.

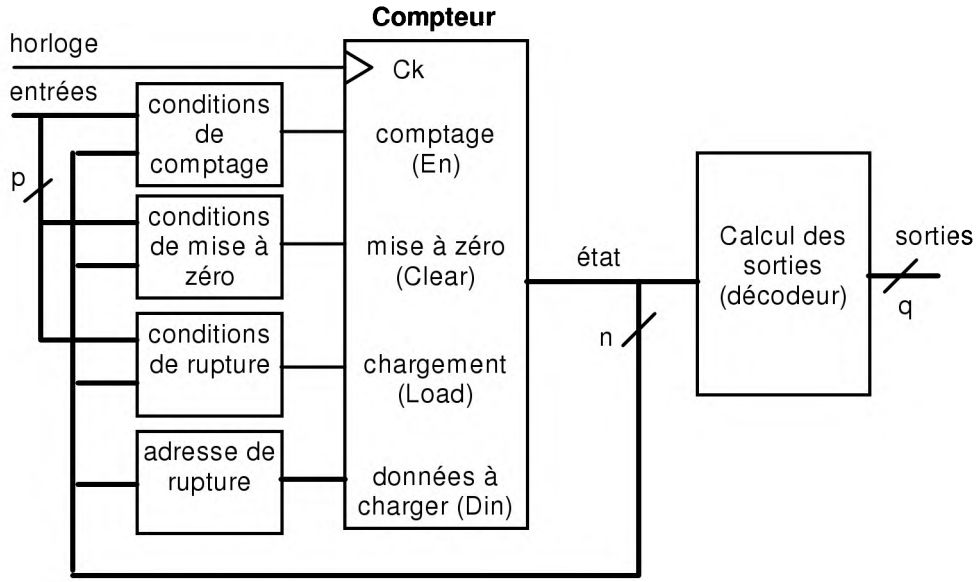


Figure V-28

Du diagramme de transitions aux commandes

Dans l'élaboration des équations, la priorité qui existe entre les commandes des compteurs classiques (74xx163, par exemple) simplifie les calculs.

Prenons comme exemple le diagramme de transitions de la figure V-21, notre dernière version du décodeur Manchester, que nous rappelons ici (figure V-29) :

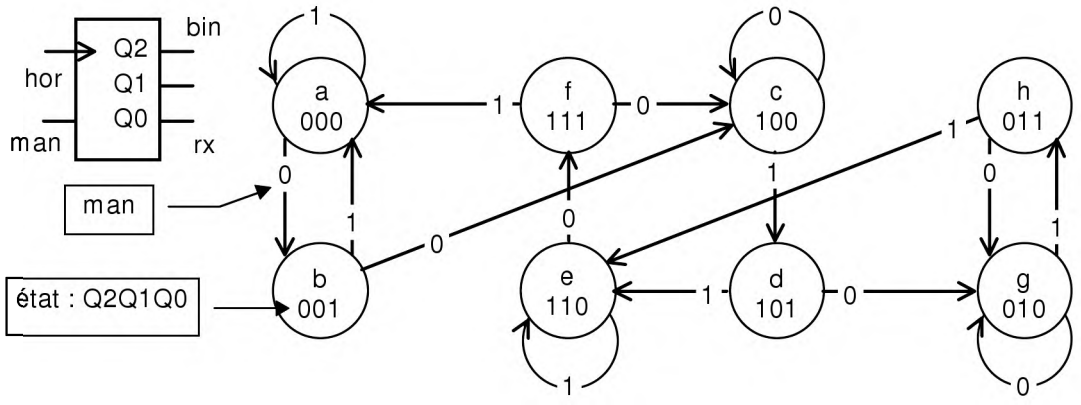


Figure V-29

De ce diagramme on peut déduire les commandes du compteur, en utilisant une notation symbolique où interviennent les noms des états[32] :

$$\begin{aligned}
\text{Clear} &= \mathbf{b} * \overline{\text{man}} + \mathbf{f} * \overline{\text{man}} \\
\text{Load} &= \mathbf{b} * \overline{\text{man}} + \mathbf{d} * \overline{\text{man}} + \mathbf{f} * \overline{\text{man}} + \mathbf{h} \\
\text{En} &= \mathbf{a} * \overline{\text{man}} + \mathbf{c} * \text{man} + \mathbf{d} * \text{man} + \mathbf{e} * \overline{\text{man}} + \mathbf{g} * \text{man}
\end{aligned}$$

Ces équations se prêtent bien à une réalisation avec des multiplexeurs ; si on les matérialise au moyen de portes élémentaires, la priorité entre les commandes du compteur permet de mettre des « ϕ » dans les tables de vérité qui déterminent Load (partout où Clear est vrai) et En (partout où Clear ou Load sont vrais), autorisant des simplifications supplémentaires.

Les adresses de rupture sont extrêmement simples, en raison du faible nombre de branchements (on peut mettre « ϕ » dans toutes les cases de la table de vérité qui correspondent à une condition où l'équation de Load est fausse) :

$$\begin{aligned}
D0 &= 0 \\
D1 &= Q1 \oplus Q2 \\
D2 &= \overline{Q0 \oplus Q2} + \text{man}
\end{aligned}$$

Nous ne détaillerons pas plus ce type d'application des compteurs programmables. Le lecteur intéressé en trouvera de nombreux exemples dans les références bibliographiques.

V.4.2 Séquenceurs micro-programmés

Dans l'architecture précédente, toutes les équations sont figées par les circuits utilisés et leur câblage ; pour obtenir des machines d'états universelles, capables de matérialiser n'importe quel diagramme de transitions sans modification du schéma, les concepteurs s'orientèrent vers les séquenceurs microprogrammés.

L'idée générale est simple, la partie la plus figée d'un séquenceur câblé traditionnel est le bloc de calcul de l'adresse, en cas de rupture de séquence. Si on introduit, dans le code binaire de l'état de la machine, un champ réservé à cette adresse, on gagne en souplesse.

Une architecture, simplifiée, d'une telle machine est donnée à la figure V-30 :

[32] Il faut, par exemple, remplacer l'état f par son équivalent binaire, soit Q2*Q1*Q0.

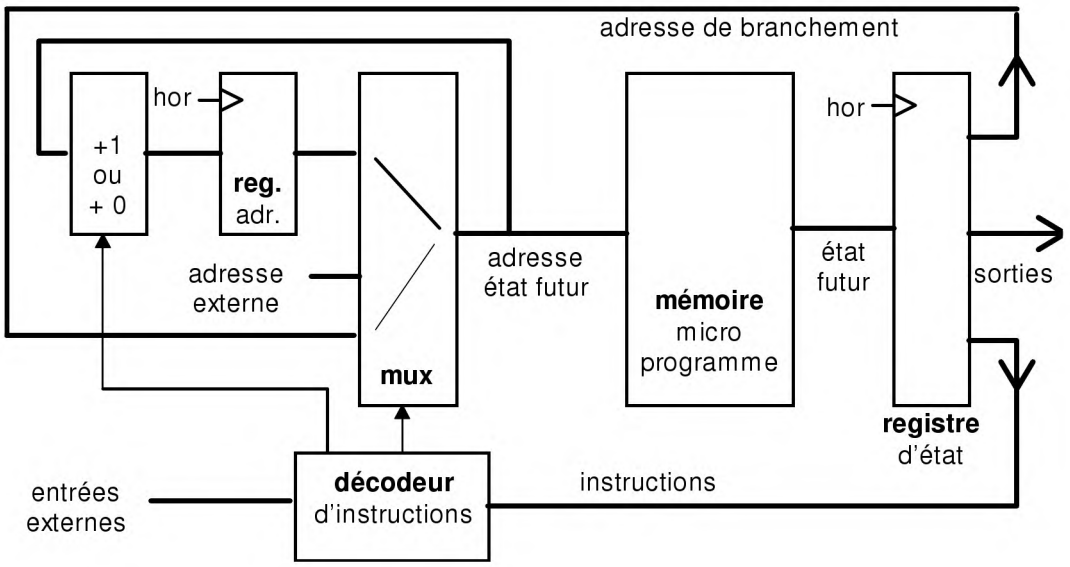


Figure V-30

Allant plus loin dans cette voie, on structure le code de l'état en champs qui représentent chacun une commande standard. On utilise une mémoire pour stocker les valeurs des états du diagramme de transitions, et deux registres pour stocker *la valeur* de l'état actuel et *l'adresse* de l'état futur.

Des commandes classiques sont[33] :

- Incrémenter l'adresse de l'état d'une unité si une condition est vraie, cela provoque un déroulement en séquence, avec une condition d'attente.
- Charger dans le registre d'état la valeur dont l'adresse est donnée dans le champ d'adresse, il s'agit d'un branchement.
- Charger dans le registre d'état la valeur dont l'adresse est fournie par une entrée externe, il s'agit d'un saut à un autre diagramme de transitions.

Le diagramme de transitions devient, en fait un véritable programme, d'où son nom de micro programme, dans lequel les états deviennent des micro instructions. Outre les commandes, les micro instructions contiennent des champs qui définissent les actions vers l'extérieur.

En enrichissant la machine d'une mémoire RAM organisée en pile et d'un compteur auxiliaire, il devient possible de créer des boucles et des sous programmes.

L'avènement des logiciels de synthèse en langage de haut niveau et des circuits programmables par l'utilisateur a considérablement restreint le champ d'application des machines micro programmées. Il est actuellement plus simple de créer sa propre machine d'états, décrite en langage évolué, programmée et modifiée

[33]On consultera, par exemple, une notice du circuit Am2910.

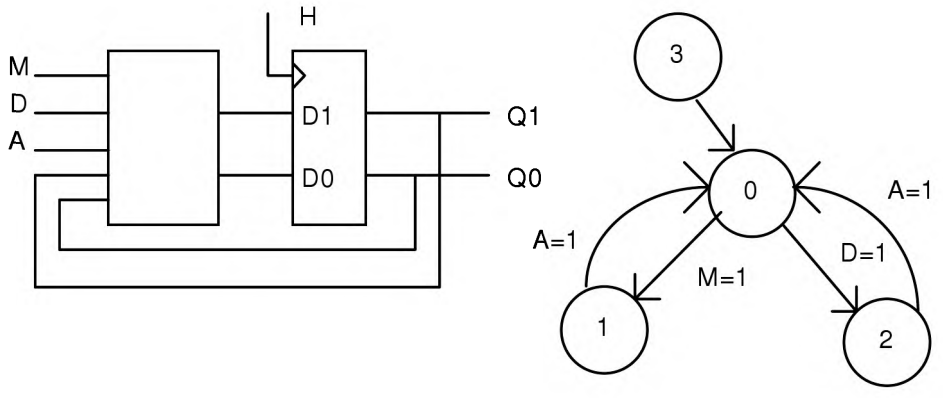
en quelques minutes dans un circuit en technologie *Flash*, que de créer des micro programmes que l'on inscrira dans une mémoire de même technologie.

La souplesse a changé de camp.

Exercices

Cohérence d'un diagramme de transition.

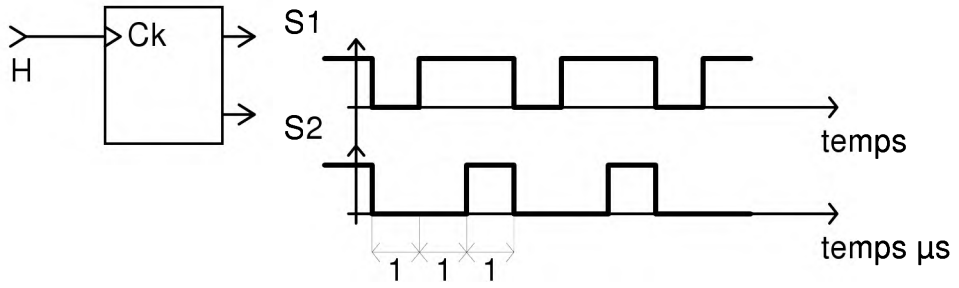
Une machine d'états synchrone dispose de trois entrées de commande (en plus de l'horloge), M, D et A (on peut imaginer qu'il s'agit, par exemple, d'une partie de commande d'ascenseur). Une ébauche de diagramme de transition est représentée ci-dessous. Cette ébauche de diagramme comporte une faute de principe, expliquer laquelle. Proposer une correction.



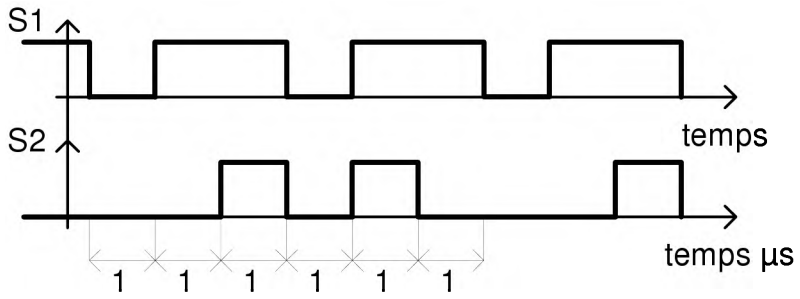
En rajoutant les conditions de maintien, qui ne sont pas représentées, donner les équations logiques de D1 et D0 induites par le diagramme corrigé.

Génération de signaux

On souhaite réaliser une fonction logique synchrone qui fournit en sortie les signaux suivants :



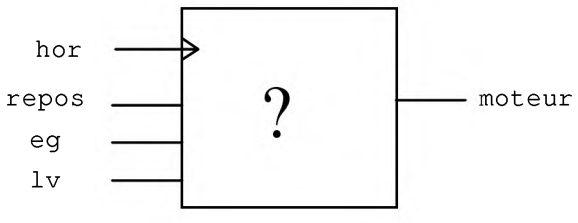
Etablir un diagramme de transition qui permet de répondre au problème.
Proposer une solution qui fait appel à des bascules D, puis à des bascules J-K.
Préciser quelle doit être la fréquence de l'entrée d'horloge.
Les chronogrammes précédents sont modifiés comme indiqué ci-dessous :



Montrer qu'il faut rajouter à la solution précédente une bascule.
Proposer une solution qui fait appel à des bascules D.

Commande d'un moteur d'essuie glaces

Le moteur d'essuie glace d'une voiture est mis en marche soit par une commande `eg`, soit par une commande `lv`, la seconde actionnant simultanément la pompe du lave glace. On se propose de faire une réalisation, en logique synchrone, de la commande du moteur :



Les signaux d'entrée sont supposés synchrones de l'horloge `hor`.

L'entrée `repos` provient d'un détecteur de fin de course, qui indique par un '1', que les balais sont en position horizontale, pare brise dégagé.

Dans tous les cas, le moteur ne doit être arrêté que quand les balais sont en position de repos.

La commande `eg` provoque, par un '1', la mise en route du moteur (`moteur = '1'`). Ce dernier ne doit s'arrêter que quand `eg` est désactivée, et que les balais sont en position de repos.

La commande `lv` provoque, outre la mise en marche de la pompe du lave glace, la mise en route du moteur. Quand cette commande redevient inactive, le moteur ne s'arrête qu'après que les balais d'essuie glace aient effectués quatre aller retours complets, pour assécher le pare brise.

1. Proposer une machine d'états qui réponde au problème. Décrire le fonctionnement de cette machine par un diagramme de transitions.
2. En déduire un programme VHDL. On veillera à ce que la sortie `moteur` corresponde à la sortie d'une bascule synchrone.

VI Annexe : VHDL

VHDL est l'abréviation de « Very high speed integrated circuits Hardware Description Langage ». L'ambition des concepteurs du langage est de fournir un outil de description homogène des circuits, qui permette de créer des modèles de simulation et de « compiler » le silicium à partir d'un programme unique. Initialement réservé au monde des circuits numériques, VHDL est en passe d'être étendu aux circuits analogiques.

Deux des intérêts majeurs du langage sont :

- Des *niveaux de description* très divers: VHDL permet de représenter le fonctionnement d'une application tant du point de vue système que du point de vue circuit, en descendant jusqu'aux opérateurs les plus élémentaires. A chaque niveau, la description peut être structurelle (portrait des interconnexions entre des sous-fonctions) ou comportementale (langage évolué).
- Son aspect « *non propriétaire* »: le développement des circuits logiques a conduit chaque fabriquant à développer son propre langage de description. VHDL est en passe de devenir le langage commun à de nombreux systèmes de CAO, indépendants ou liés à des producteurs de circuits, des (relativement) simples outils d'aide à la programmation des PALs aux ASICs, en passant par les FPGAs.

La description qui suit est loin d'être exhaustive, héritier d'ADA, VHDL est un « gros » langage. Nous en présentons un sous-ensemble qui, nous l'espérons, doit permettre à un néophyte d'aborder ses premières réalisations avec un bagage minimum, limité à des *constructions synthétisables*, et, en principe, portables sur n'importe quel compilateur.

VI.1. Principes généraux

VI.1.1 Description descendante : le « top down design »

Une application un tant soit peu complexe est découpée en sous-ensembles qui échangent des informations suivant un protocole bien défini. Chaque sous-ensemble est, à son tour, subdivisé, et ainsi de suite jusqu'aux opérateurs élémentaires.

Un système est construit comme une hiérarchie d'objets, les détails de réalisation se précisant au fur et à mesure que l'on descend dans cette hiérarchie. A un niveau donné de la hiérarchie, les détails de fonctionnement interne des niveaux inférieurs sont *invisibles*, C'est le principe même de la programmation structurée. Plusieurs réalisations d'une même fonction pourront être envisagées, sans qu'il soit nécessaire de remettre en cause la conception des niveaux supérieurs ; plusieurs personnes pourront collaborer à un même projet, sans que chacun ait à connaître tous les détails de l'ensemble.

La conception descendante consiste à définir le système en partant du sommet de la hiérarchie, en allant du général au particulier. VHDL permet, par exemple, de tester la validité de la conception d'ensemble, avant que les détails des sous-fonctions ne soient complètement définis. A titre d'exemple, l'architecture générale d'un processeur peut être évaluée sans que le mode de réalisation de ses registres internes ne soit connu, le fonctionnement des registres en question sera alors décrit au niveau comportemental.

VI.1.2 Simulation et/ou synthèse

VHDL a été, initialement, conçu comme un langage de simulation, il est fortement marqué par cet héritage très informatique, ce qui est parfois un peu déroutant pour l'électronicien, proche du matériel, qui n'est pas toujours un spécialiste des langages de programmation. Citons quelques exemples :

- Contrairement à C ou PASCAL, VHDL est un langage qui comprend le « parallélisme », c'est à dire que des blocs d'instructions peuvent être exécutés simultanément, par opposition à séquentiellement comme dans un langage procédural traditionnel. Autant ce parallélisme est fondamental pour comprendre le fonctionnement d'un simulateur logique, et peut être déroutant pour un programmeur habitué au déroulement séquentiel des instructions qu'il écrit, autant il est évident que le fonctionnement d'un circuit ne dépend pas de l'ordre dans lequel ont été établies les connexions. L'utilisateur de VHDL gagnera beaucoup en ne se laissant pas enfermer dans l'aspect langage de programmation, en se souvenant qu'il est en train de créer un vrai circuit. Les parties séquentielles du langage, car il y en a, doivent, dans ce contexte, être comprises soit comme une facilité offerte dans

l'écriture de certaines fonctions, soit comme *le* moyen de décrire des opérateurs fondamentalement séquentiels : les opérateurs synchrones.

- La modélisation correcte d'un système suppose de prendre en compte, au niveau du simulateur, les imperfections du monde réel. VHDL offre donc la possibilité de spécifier des retards, de préciser ce qui se passe lors d'un conflit de bus etc. Pour simuler toutes ces vicissitudes, le langage offre toute une gamme d'outils : signaux qui prennent une valeur inconnue, messages d'erreurs quand un « circuit » détecte une violation de *set-up time*, changements d'états retardés pour simuler les temps de propagation. Toutes les constructions associées de ce type ne sont évidemment *pas* synthétisables ! La difficulté principale est que, suivant les compilateurs, la frontière entre ce qui est synthétisable et ce qui ne l'est pas n'est pas toujours la même, même pour des compilateurs qui respectent la norme IEEE-1076. Avant d'utiliser un outil de synthèse, le concepteur de circuit a tout à gagner à lire très attentivement la présentation du sous-ensemble de VHDL accepté par cet outil.
- Trois classes de données existent en VHDL : les constantes, les variables et les signaux. La nature des signaux ne présente aucune ambiguïté, ce sont des objets qui véhiculent une information logique tant du point de vue simulation que dans la réalité. Les signaux qui ont échappé aux simplifications logiques, apportées par l'optimiseur toujours présent, sont des vraies équipotentielles du schéma final. Les variables sont destinées, comme dans tout langage, à stocker temporairement des valeurs, dans l'optique d'une utilisation future, sans chercher à représenter la réalité. Certains compilateurs considèrent que les variables n'ont aucune existence réelle, au niveau du circuit, qu'elles ne sont que des outils de description fonctionnelle. D'autres transforment, éventuellement (cela dépend de l'optimiseur), les variables en cellules mémoires...

Tous les exemples qui sont donnés ici, et dans les chapitres précédents, ont été compilés sur un logiciel destiné à la synthèse de circuits programmables[1]. Créé par et pour des concepteurs de circuits, ce compilateur ne comporte aucune construction spécifique de la simulation, mais autorise cependant des édifices relativement élaborées, notamment dans l'utilisation des variables et des boucles[2]. Nous avons parfois eu quelques surprises désagréables lors du portage de ces exemples sur d'autres systèmes, syntaxiquement acceptés, certains programmes étaient refusés par l'outil de synthèse, ou généraient une quantité surprenante de bascules.

En conclusion citons l'un des « grands » de la CAO électronique[3]:

« Des mythes communs existent :

---

[1]WARP, Cypress Semiconductors

[2]Les variables ne « sortent » pas du programme, seuls les signaux se retrouvent dans le circuit final.

[3]Mentor Graphics, Methods for using Autologic in top down design, 1994.

La conception descendante est une démarche presse-bouton, la compétence de l'expert est rarement nécessaire.

Faux. VHDL, les outils de synthèse et d'optimisation ne peuvent pas transformer un mauvais concepteur en un bon. Ce sont de simples outils supplémentaires qui peuvent aider un ingénieur à réaliser plus rapidement et plus efficacement un matériel quand ils sont utilisés correctement.

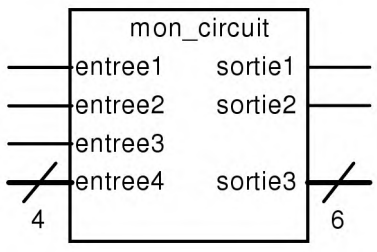
Les outils d'optimisation libèrent l'utilisateur de la nécessité de comprendre les détails physiques de sa réalisation.

Faux. Il est toujours nécessaire de comprendre les détails physiques de la façon dont est implémentée une réalisation. L'ingénieur doit *regarder par dessus l'épaule* de l'outil pour s'assurer que le résultat est conforme à ses exigences et à sa philosophie, et que le résultat est obtenu en un temps raisonnable. »

VI.1.3 L'extérieur de la boîte noire : une « ENTITÉ »

Nous avons mentionné que, dans une construction hiérarchique, les niveaux supérieurs n'ont pas à connaître les détails des niveaux inférieurs. Une fonction logique sera vue, dans cette optique, comme un assemblage de « boîtes noires », dont, syntaxiquement parlant, seules les modes d'accès sont nécessaires à l'utilisateur[4].

La construction qui décrit l'extérieur d'une fonction est l'entité (*entity*). La déclaration correspondante lui donne un nom et précise la liste des signaux d'entrée et de sortie :




```
entity mon_circuit is port (
  entree1 : in bit;
  entree2, entree3  : in bit;
  entree4 : in bit_vector(0 to 3);
  sortie1, sortie2 : out bit;
  sortie3 : out bit_vector(0 to 5))
end mon_circuit;
```

Figure VI-1

[4]Ce point est à mettre en parallèle avec les prototypes d'un langage comme le C. Dans un programme qui utilise la fonction sinus, le remplacement de celle-ci par une fonction exponentielle ne posera aucun problème de syntaxe, les modes d'accès sont les mêmes. Cela ne veut évidemment pas dire que les deux programmes fourniront les mêmes résultats, ce dernier point est un problème de sémantique.

Dans l'exemple qui précède, les noms des objets, qui dépendent du choix de l'utilisateur, sont écrits en italique, les autres mots sont des mots-clés du langage.

Les choix possibles pour le sens de transfert sont : in, out, inout et buffer (une sortie qui peut être « lue » par l'intérieur du circuit).

Les choix possibles pour les types de données échangées sont les mêmes que pour les signaux (voir ci-dessous).

VI.1.4 Le fonctionnement interne : une « ARCHITECTURE »

L'architecture décrit le fonctionnement interne d'un circuit auquel est attaché une entité. Ce fonctionnement peut être décrit de différentes façons :

Description structurelle - le circuit est vu comme un assemblage de composants de niveau inférieur, c'est une description « schématique ». Souvent ce mode de description est utilisé au niveau le plus élevé de la hiérarchie, chaque composant étant lui-même défini par un programme VHDL (entité et architecture).

Description comportementale - le comportement matériel du circuit est décrit par un algorithme, indépendamment de la façon dont il est réalisé au niveau structurel.

Description par un flot de données - le fonctionnement du circuit est décrit par un flot de données qui vont des entrées vers les sorties, en subissant, étape par étape, des transformations élémentaires successives. Ce mode de description permet de reproduire l'architecture logique, en couches successives, des opérateurs combinatoires.

Flot de données et représentation comportementale sont très voisines, dans les deux cas le concepteur peut faire appel à des instructions de haut niveau. La première méthode utilise un grand nombre de signaux internes qui conduisent au résultat par des transformations de proche en proche, la seconde utilise des blocs de programme (les processus explicites), qui manipulent de nombreux signaux avec des algorithmes séquentiels[5].

La syntaxe générale d'une architecture comporte une partie de déclaration et un corps de programme :

```
architecture exemple of mon_circuit is
      partie déclarative optionnelle :   types, constantes,
                        signaux locaux, composants.
begin
corps de l'architecture.
suite d'instructions parallèles :
      affectations de signaux;
      processus explicites;
      blocs;
```

[5]On relira avec profit les trois descriptions d'une bascule D-Latch, qui sont données au paragraphe III.3.

```
        instanciation (i.e. importation
        dans un schéma) de composants.
end exemple ;
```

On se reportera aux exemples du chapitre III pour des illustrations simples.

VI.1.5 Des algorithmes séquentiels décrivent un câblage parallèle : les « PROCESSUS »

« Un processus est une instruction *concurrente* (N.D.T deux instructions concurrentes sont simultanées) qui définit un comportement qui doit avoir lieu quand ce processus devient actif. Le comportement est spécifié par une suite d'instructions *séquentielles* exécutées dans le processus. »[6]

Que cela signifie-t-il ?

Trois choses :

1. Les différentes parties d'une réalisation interagissent simultanément, peu importe l'ordre dans lequel un câbleur soude ses composants, le résultat sera le même. Le langage doit donc comporter une contrainte de « parallélisme » entre ses instructions. Cela implique des différences notables avec un langage procédural comme le C.
   En VHDL :
   ```
   a <= b ;
   c <= a + d ;
   ```
   et
   ```
   c <= a + d ;
   a <= b ;
   ```
   représentent la même chose, ce qui est notablement différent de ce qui se passerait en C pour :
   ```
   a = b ;
   c = a + d ;
   ```
   et
   ```
   c = a + d ;
   a = b ;
   ```
   Les affectations de signaux, à *l'extérieur* d'un processus explicite, sont traitées comme des processus tellement élémentaires qu'il est inutile de les déclarer comme tels. Ces affectations sont traitées en parallèle, de la même façon que plusieurs processus indépendants.

[6]WARP, VHDL Reference, Cypress Semiconductors.

2. L'algorithmique fait grand usage d'instructions séquentielles pour décrire le monde. VHDL offre cette facilité à *l'intérieur* d'un processus explicitement déclaré. Dans le corps d'un processus il sera possible d'utiliser des variables, des boucles, des conditions, dont le sens est le même que dans les langages séquentiels. Même les affectations entre signaux sont des instructions séquentielles quand elles apparaissent à l'intérieur d'un processus. Seul sera visible de l'extérieur le résultat final obtenu à la fin du processus.

3. Les opérateurs séquentiels, surtout synchrones, mais pas exclusivement eux, comportent « naturellement » la notion de mémoire, qui est le fondement de l'algorithmique traditionnelle. Les processus sont *la* représentation privilégiée de ces opérateurs[7]. Mais attention, *la réciproque n'est pas vraie*, il est parfaitement possible de décrire un opérateur purement combinatoire par un processus, le programmeur utilise alors de cet objet la seule facilité d'écriture de l'algorithme[8].

Outre les simples affectations de signaux, qui sont en elles mêmes des processus implicites à part entière, la description d'un processus obéit à la syntaxe suivante :

Processus : syntaxe générale

```
[étiquette : ] process [ (liste de sensibilité) ]
      partie déclarative optionnelle : variables notamment
begin
      corps du processus.
      instructions séquentielles
end process [ étiquette ] ;
```

Les éléments mis entre crochets sont optionnels, ils peuvent être omis sans qu'il y ait d'erreur de syntaxe.

La liste de sensibilité est la liste des signaux qui déclenchent, par le changement de valeur de l'un quelconque d'entre eux, l'activité du processus. Cette liste peut être remplacée par une instruction « wait » dans le corps du processus :

---

[7]Ce n'est pas la seule, les descriptions structurelles et flot de données, plus proches du câblage du circuit, permettent de décrire tous les opérateurs séquentiels avec des opérateurs combinatoires élémentaires. Pour les circuits qui comportent des bascules comme éléments primitifs, connus de l'outil de synthèse, les deux seules façons d'utiliser ces bascules sont les process et leur instanciation comme composants dans une description structurelle.

[8]Voir à titre d'exemple la description que nous avons donnée du ou-exclusif comme contrôleur de parité, § III.2..

L'instruction `wait`

Cette instruction indique au processus que son déroulement doit être suspendu dans l'attente d'un événement sur un signal (un signal change de valeur), et tant qu'une condition n'est pas réalisée.

Sa syntaxe générale est[9] :

```
wait [on liste_de_signaux ] [until condition ] ;
```

La liste des signaux dont l'instruction attend le changement de valeur joue exactement le même rôle que la liste de sensibilité du processus, mais *l'instruction `wait` ne peut pas être utilisée en même temps qu'une liste de sensibilité*. La tendance, pour les évolutions futures du langage, semble être à la suppression des listes de sensibilités, pour n'utiliser que les instructions d'attente.

Description d'un opérateur séquentiel

La représentation des horloges : pour représenter les opérateurs synchrones de façon comportementale il *faut* introduire l'horloge dans la liste de sensibilité, *ou* insérer dans le code du processus une instruction « `wait` » explicite. Rappelons qu'il est interdit d'utiliser à la fois une liste de sensibilité et une instruction `wait`. Quand on modélise un opérateur qui comporte à la fois des commandes synchrones et des commandes asynchrones, il *faut*, avec certains compilateurs, mettre ces commandes dans la liste de sensibilité.

Exemple :

```
architecture fsm of jk_raz is
   signal etat : std_logic := '0' ; -- Librairie IEEE
begin
q <= etat;
process(clock,raz) -- deux signaux d'activation
begin
   if raz = '1' then -- raz asynchrone
      etat <= '0';
   elsif rising_edge(clock) then  -- Librairie IEEE
      case etat is
         when '0' =>
            IF j = '1' then
               etat <= '1';
            end if;
         when '1' =>
            if k = '1' then
               etat <= '0';
```

[9]On peut spécifier un temps d'attente maximum (wait ... `for` *`temps`*), mais cette clause n'est pas synthétisable.

```
            end if;
         end case;
      end if;
   end process;
   end fsm;
```

Dans l'exemple précédent, la priorité de la mise à zéro asynchrone, sur le fonctionnement synchrone normal de la bascule JK, apparaît par l'ordre des instructions de la structure `if...elsif`. Le processus est utilisé là à la fois pour modéliser un opérateur essentiellement séquentiel, la bascule, et pour faciliter la description de l'effet de ses commandes par un algorithme séquentiel.

Pour modéliser un comportement purement synchrone on peut indifféremment utiliser la liste de sensibilité ou une instruction wait :

```
architecture fsm_liste of jk_simple is
   signal etat : std_logic := '0' ; -- Librairie IEEE
begin
q <= etat;
process(clock) -- un seul signal d'activation
begin
   if rising_edge(clock) then  -- Librairie IEEE
      case etat is
         when '0' =>
            IF j = '1' then
               etat <= '1';
            end if;
         when '1' =>
            if k = '1'  then
               etat <= '0';
            end if;
      end case;
   end if;
end process;
end fsm_liste;
```

Ou, de façon strictement équivalente, en utilisant une instruction « wait » :

```
architecture fsm_wait of jk_simple is
   signal etat : std_logic := '0' ; -- Librairie IEEE
begin
q <= etat;
process  -- pas de liste de sensibilité
begin
wait until rising_edge(clock) ;
   case etat is
      when '0' =>
         IF j = '1' then
            etat <= '1';
```

```
            end if;
         when '1' =>
            if k = '1' then
               etat <= '0';
            end if;
      end case;
   end process;
   end fsm_wait;
```

Description par un processus d'un opérateur combinatoire ou asynchrone

Un processus permet de décrire un opérateur purement combinatoire ou un opérateur séquentiel asynchrone, en utilisant une démarche algorithmique.

Dans ces deux cas la liste de sensibilité, ou l'instruction wait équivalente, est obligatoire ; le caractère combinatoire ou séquentiel de l'opérateur réalisé va dépendre du code interne au processus. On considère un signal qui fait l'objet d'une affectation dans le corps d'un processus :

- Si au bout de l'exécution du processus, pour *toutes* les combinaisons possibles des valeurs de la liste de sensibilité la valeur de ce signal, objet d'une affectation, est connue, l'opérateur correspondant est combinatoire.
- Si certaines des combinaisons précédentes de la liste de sensibilité conduisent à une indétermination concernant la valeur du signal examiné, objet d'une affectation, ce signal est associé à une cellule mémoire.
-

Précisons ce point par un exemple :

```
entity comb_seq is
port (
      e1, e2 : in std_logic ;
      s_et, s_latch, s_edge : out std_logic
      ) ;
end comb_seq ;

architecture exproc of comb_seq is
begin

et : process(e1,e2) -- équivalent à s_et <= e1 and e2 ;
begin
   if e1 = '1' then
      s_et <= e2 ;
   else
      s_et <= '0' ;
end if ;
end process ;
```

```
latch : process(e1,e2) -- bascule D Latch, e1 est la
commande.
begin
   if e1 = '1' then
      s_latch <= e2 ;
   end if; -- si e1 = '0' la valeur de s_latch est inconnue.
end process ;

edge : process(e1) -- bascule D Edge, e1 est l'horloge.
begin
   if rising_edge(e1) then -- e1 agit par un front.
      s_edge <= e2 ;
   end if ;
end process ;
end exproc ;
```

Dans l'exemple qui précède, le premier processus est combinatoire, le signal s_et a une valeur connue à la fin du processus, quelles que soient les valeurs des entrées e1 et e2. Dans le deuxième processus, l'instruction « if » ne nous renseigne pas sur la valeur du signal s_latch quand e1 = '0'. Cette méconnaissance est interprétée, par le compilateur VHDL, comme un maintien de la valeur précédente, d'où la génération d'une cellule mémoire dont la commande de mémorisation, e1, est active sur un niveau. Le troisième processus conduit également, et pour le même type de raison, à la synthèse d'une cellule mémoire pour le signal s_edge. Mais la commande de mémorisation est, cette fois, active sur un front, explicitement mentionné dans la condition de l'instruction « if » : `e1'event`. La façon dont est traitée la commande de mémorisation e1 dépend donc de l'écriture du test : niveau ou front[10].

VI.2. Eléments du langage

VI.2.1 Les données appartiennent à une classe et ont un type

VHDL, héritier d'ADA, est un langage *fortement typé*. Toutes les données ont un type qui doit être déclaré avant l'utilisation[11] et aucune conversion de type automatique (une souplesse et un piège immense du C, par exemple) n'est effectuée. Pour passer du type entier au type bit_vector, par exemple, il faut faire appel à une fonction de conversion.

---

[10]Il peut y avoir des petites différences d'interprétation, suivant les compilateurs, entre les deux types de bascules, si on omet l'attribut 'event.

[11]Sauf, et c'est bien pratique, les variables entières des boucles « FOR ».

Une donnée appartient à une classe qui définit, avec son type, son comportement. Des données de deux classes différentes, mais de même type, peuvent échanger des informations directement : on peut affecter la valeur d'une variable à un signal, par exemple (nous verrons ci-dessous que variables et signaux sont deux classes différentes).

La portée des noms est, en général, locale. Un nom déclaré à l'intérieur d'une architecture, par exemple, n'est connu que dans celle-ci. Des objets globaux sont possibles, on peut notamment définir des constantes, comme `zero` ou `one`, extérieures aux unités de programmes que constituent les couples entité-architecture. A l'intérieur d'une architecture les objets déclarés dans un bloc (délimité par les mots-clés `begin` et `end`) sont visibles des blocs plus internes uniquement.

Les objets déclarés dans une entité sont connus de toutes les architectures qui s'y rapportent.

Les classes : signaux, variables et constantes

Signaux

Les signaux représentent les données physiques échangées entre des blocs logiques d'un circuit. Chacun d'entre eux sera matérialisé dans le schéma final par une *équipotentielle* et, éventuellement, une *cellule mémoire* qui conserve la valeur de l'équipotentielle entre deux commandes de changement. Les « ports » d'entrée et de sortie, attachés à une entité, par exemple, sont une variété de signaux qui permettent l'échange d'informations entre différentes fonctions. Leur utilisation est similaire à celle des arguments d'une procédure en PASCAL, le sens de transfert de l'information doit être précisé.

Syntaxe de déclaration (se place dans la partie déclarative d'une architecture[12]) :

```
signal nom1 , nom2 : type ;
```

Affectation d'une valeur (se place dans le corps d'une architecture ou d'un processus) :

```
nom <= valeur_compatible_avec_le_type ;
```

La valeur affectée peut être le résultat d'une expression, simple ou conditionnelle (when), ou la valeur renvoyée par l'appel d'une fonction.

A l'extérieur d'un processus toutes les affectations de signaux sont concurrentes, *c'est donc une erreur* (sémantique, pas syntaxique) *d'affecter plus d'une fois une valeur à un signal*. L'affectation d'une valeur à un signal traduit, en fait, la connexion de la sortie d'un opérateur à l'équipotentielle correspondante. Il

[12]ou d'un paquetage, voir plus loin.

s'agit là d'une opération permanente, une soudure sur une carte, par exemple, qu'il est hors de question de modifier ailleurs dans le programme. Si un signal est l'objet d'affectations multiples, ce qui revient à mettre en parallèle plusieurs sorties d'opérateurs (trois états ou collecteurs ouverts, par exemple), il faut adjoindre à ce signal, pour les besoins de la simulation, une fonction de résolution qui permet de résoudre le conflit[13].

Variables

Les variables sont des objets qui servent à stocker un résultat intermédiaire pour faciliter la construction d'un *algorithme séquentiel*. Elles ne peuvent être utilisées *que dans les processus, les procédures ou les fonctions*, et dans les boucles « generate » qui servent à créer des schémas répétitifs.

Syntaxe de déclaration (se place dans la partie déclarative d'un processus, d'une procédure ou d'une fonction) :

```
variable nom1 , nom2 : type [:= expression];
```

L'expression facultative qui apparaît dans la déclaration précédente permet de donner à une variable une valeur initiale choisie par l'utilisateur. A défaut de cette expression *le compilateur, qui initialise toujours les variables*[14], utilise une valeur par défaut qui dépend du type déclaré.

Affectation d'une valeur :

```
nom := valeur_compatible_avec_le_type ;
```

La valeur affectée peut être le résultat d'une expression ou la valeur renvoyée par l'appel d'une fonction.

Les variables de VHDL jouent le rôle des variables automatiques des langages procéduraux, comme C ou Pascal, elles ont une portée limitée au module de programme dans lequel elles ont été déclarées, et sont détruites à la sortie de ce module.

La différence entre variables et signaux est que les premières n'ont pas d'équivalent physique dans le schéma, contrairement aux seconds. Certains outils de synthèse ne respectent malheureusement pas cette distinction. On notera qu'il est possible d'affecter la valeur d'une variable à un signal, et inversement, pourvu que les types soient compatibles.

Constantes

Les constantes sont des objets dont la valeur est fixée une fois pour toute.

---

[13]Dans les applications de synthèse les portes à sorties non standard sont généralement introduites dans une description structurelle.

[14]Le programmeur ne doit, notamment, pas s'attendre à retrouver les variables d'un processus dans l'état où il les avait laissées lors d'une activation précédente de ce processus.

Exemples de valeurs constantes simples :

```
'0', '1', "01101001", 1995, "azerty"
```

On peut créer des constantes nommées :

```
constant nom1 : type [ := valeur_constante ] ;
```

On notera que les vecteurs de bits (bit_vector) sont traités comme des chaînes, on peut préciser une base différente de la base 2 pour ces constantes :

```
X"3A007", O"237015"  pour hexadécimal et octal.
```

De même, les valeurs entières peuvent être écrites dans une autre base que la base 10 :

```
16#ABCDEF0123#, 2#001011101# ou 2#0_0101_1101#,
                        pour plus de lisibilité.
```

En général les nombres flottants ne sont pas acceptés par les outils de synthèse.

Des types adaptés à l'électronique numérique

VHDL connaît un nombre limité de types de base, qui reflètent le fonctionnement des systèmes numériques (pour l'instant, VHDL est en passe de devenir un langage de description des circuits analogiques), et offre à l'utilisateur de construire à partir de ces types génériques :

- des sous-types (sous-ensembles du type de base), obtenus en précisant un domaine de variation limité de l'objet considéré,
- des types composés, obtenus par la réunion de plusieurs types de base identiques (tableaux) ou de types différents (enregistrements).

En plus des types prédéfinis et de leurs dérivés, l'utilisateur a la possibilité de créer ses propres types sous forme de types énumérés.

Les entiers

VHDL manipule des valeurs entières qui correspondent à des mots de 32 bits, soit comprises entre

```
-2147483648 et +2147483647.
```

Attention, sur les PC qui sont des machines dont les entiers continuent à hésiter entre 16 et 32 bits, l'utilisateur peut rencontrer de désagréables surprises.

Les nombres négatifs ne sont pas toujours acceptés dans la description des signaux physiques.

Déclaration :

```
signal nom : integer ;
```

ou

```
variable nom : integer ;
```

ou encore :

```
constant nom : integer ;
```

que l'on résume classiquement par :

```
signal | variable |constant nom : integer ;
```

Le symbole | signifiant « ou ».

On peut spécifier une plage de valeurs inférieure à celle obtenue par défaut, par exemple :

```
signal etat : integer range 0 to 1023 ;
```

permet de créer un compteur 10 bits.

La même construction permet de créer un sous-type :

```
subtype etat_10 is integer range 0 to 1023 ;
signal etat1 , etat2 : etat_10 ;
```

Attention ! La restriction d'étendue de variation est utilisée pour générer le nombre de chiffres binaires nécessaires à la représentation de l'objet, l'arithmétique sous-jacente n'est (pour l'instant) pas traitée par les compilateurs. Cela veut dire que

```
signal chiffre : integer range 0 to 9 ;
```

permet de créer un objet codé sur quatre bits, mais

```
chiffre <= chiffre + 1 ;
```

ne crée *pas* un compteur décimal.

Pour ce faire il faut écrire explicitement :

```
if chiffre < 9 then
   chiffre <= chiffre + 1 ;
else
   chiffre <= 0 ;
end if ;
```

La déclaration, au niveau le plus élevé d'une hiérarchie, de ports d'entrée ou de sortie comme nombres entiers pose un problème de contrôle par l'utilisateur de l'assignation des broches physiques du circuit final aux chiffres binaires générés. Cette assignation sera faite automatiquement par l'outil de développement. Si ce non contrôle est gênant, il est possible de transformer un nombre entier en tableau de bits, via les fonctions de conversion de la librairie associée à un compilateur.

Les types énumérés

L'utilisateur peut créer ses propres types, par simple énumération de constantes symboliques qui fixent toutes les valeurs possibles du type. Par exemple :

```
type drinkState is
(zero,five,ten,fifteen,twenty,twentyfive,owedime);
signal drinkStatus: drinkState;
```

Les bits

Il s'agit là, évidemment, du type de base le plus utilisé en électronique numérique. Un objet de type bit peut prendre deux valeurs : '0' et '1', il s'agit, en fait, d'un type énuméré prédéfini.

Déclaration :

```
signal | variable nom : bit ;
```

Pour les besoins de la simulation et des connexions en bus, la librairie IEEE propose un type bit plus étoffé (`std_logic`), pouvant prendre, entre autres, les valeurs '0', '1', 'X' (X pour inconnu) et 'Z' (pour haute impédance). Ce type est traduit, en synthèse, par desvaleurs binaires ordinaires plus l'état « déconnecté » :

```
Library IEEE ;
use IEEE.std_logic_1164.all ;

entity basc_tri_state is
  port( clk, oe : in std_logic ;
       sort : inout std_logic );
        -- type std_logic défini dans la librairie
end basc_tri_state ;
```

Ces extensions sont portables tant en simulation qu'en synthèse.

Les booléens

Autre type énuméré, le type booléen peut prendre deux valeurs : "`true`" et "`false`". Il intervient essentiellement comme résultat d'expressions de comparaisons, dans des `IF`, par exemple, ou dans les valeurs renvoyées par des fonctions.

Les tableaux

A partir de chaque type de base on peut créer des tableaux, collection d'objets du même type. L'un des plus utilisés est le type bit_vector, défini dans la librairie standard par :

```
SUBTYPE Natural IS Integer RANGE 0 to Integer'high;
TYPE bit_vector IS ARRAY (Natural RANGE <>) OF BIT;
```

Dans l'exemple qui précède, le nombre d'éléments n'est pas précisé dans le type, ce sera fait à l'utilisation. Par exemple :

```
signal etat : bit_vector (0 to 4) ;
```

définit un tableau de cinq éléments binaires nommé état.
On aurait également pu définir directement un sous-type :

```
type cinq_bit is array (0 to 4) of bit;
signal etat : cinq_bit ;
```

Le nombre de dimensions d'un tableau n'est pas limité, les indices peuvent être définis dans le sens croissant (2 to 6) ou décroissant (6 downto 2) avec des bornes quelconques (mais cohérentes avec le sens choisi).
On notera qu'il *faut* passer par une définition de type, ce qui n'est pas le cas en C ou en PASCAL.
Une fois défini, un objet composé peut être manipulé collectivement par son nom :

```
signal etat1 : bit_vector (0 to 4) ;
variable etat2 : bit_vector (0 to 4) ;
...
etat 1 <= etat2 ;  -- parfaitement correct
```

Le compilateur contrôle que les dimensions des deux objets sont les mêmes. On remarquera, à partir de l'exemple précédent, que les classes des deux objets peuvent être différentes.
On peut, bien sûr, ne manipuler qu'une partie des éléments d'un tableau :

```
signal etat : bit_vector (0 to 4) ;
```

```
signal sous_etat : bit_vector (0 to 1) ;
signal flag : bit ;
...
sous_etat <= etat ( 1 to 2 ) ;
flag      <= etat ( 3 ) ;
```

Il est possible de fusionner deux tableaux (concaténation) pour affecter les valeurs correspondantes à un tableau plus grand :

```
signal etat : bit_vector (0 to 4) ;
signal sous_etat2 : bit_vector (0 to 1) ;
signal sous_etat3 : bit_vector (0 to 2) ;
...
etat <= sous_etat2 & sous_etat3; -- concaténation.
```

Les enregistrements

Les enregistrements (record) définissent des collections d'objets de types, ou de sous types, différents. Ils correspondent aux structures du C ou aux enregistrements de PASCAL.

Définition d'un type :

```
type clock_time is record
hour : integer range 0 to 12 ;
minute , seconde : integer range 0 to 59 ;
end record ;
```

Déclaration d'un objet de ce type :

```
variable time_of_day : clock_time ;
```

Utilisation de l'objet précédent :

```
time_of_day.hour := 3 ;
time_of_day.minute := 45 ;
chrono := time_of_day.seconde ;
```

L'ensemble d'un enregistrement peut être manipulé par son nom.

VI.2.2 Les attributs précisent les propriétés des objets

Déterminer, de façon dynamique, la taille d'un tableau, le domaine de définition d'un objet scalaire, l'élément suivant d'un type énuméré, détecter la transition montante d'un signal, piloter l'optimiseur d'un outil de synthèse, attribuer des numéros de broches à des signaux d'entrées-sorties ... etc.

Les attributs permettent tout cela.

Un attribut est une propriété, qui porte un nom, associée à une entité, une architecture, un type ou un signal. Cette propriété, une fois définie, peut être utilisée dans des expressions.

L'utilisation d'un attribut se fait au moyen d'un nom composé : le préfixe est le nom de l'objet auquel est rattaché l'attribut, le suffixe est le nom de l'attribut. Préfixe et suffixe sont séparés par une apostrophe « ' ».

```
nom_objet'nom_de_l_attribut
```

Par exemple :

`hor'event and hor = '1'` renvoie la valeur booléenne `true` si le signal `hor`, de type bit, vaut 1 après un changement de valeur, ce qui revient à tester la présence d'une transition montante de ce signal.

Certains attributs sont prédéfinis par le langage, d'autres sont attachés à un outil de développement ; l'utilisateur, enfin, peut définir, et utiliser ses propres attributs.

Attributs prédéfinis dans le langage

Les attributs prédéfinis permettent de déterminer les contraintes qui pèsent sur des objets ou des types : domaine de variation d'un type scalaire, bornes des indices d'un tableau, éléments voisins d'un objet de type énuméré, etc.

Ils permettent également de préciser les caractéristiques dynamiques de signaux, comme la présence d'un front, évoquée précédemment.

attribut	agit sur	valeur retournée
'left	type scalaire	élément de gauche
'left(n)	type tableau	borne de gauche de l'indice de la dimension n, n=1 par défaut
'right	type scalaire	élément de droite
'right(n)	type tableau	borne de droite de l'indice de la dimension n, n=1 par défaut
'high	type scalaire	élément le plus grand
'high(n)	type tableau	borne maximum de l'indice de la dimension n, n=1 par défaut
'low	type scalaire	élément le plus petit
'low(n)	type tableau	borne minimum de l'indice de la dimension n, n=1 par défaut
'length(n)	type tableau	nombre d'éléments de la dimension n, n=1 par défaut
'pos(v)	type scalaire	position de l'élément v dans le type
'val(p)	type scalaire	valeur de l'élément de position p dans le type
'succ(v)	type scalaire	valeur qui suit (position + 1) l'élément de valeur v dans le type
'pred(v)	type scalaire	valeur qui précède (pos. – 1) l'élément de valeur v dans le type
'leftof(v)	type scalaire	valeur de l'élément juste à gauche de l'élément de valeur v
'rightof(v)	type scalaire	valeur de l'élément juste à droite de l'élément de valeur v
'event	signal	valeur booléenne "TRUE" si la valeur du signal vient de changer
'base	tous types	renvoie le type de base d'un type dérivé
'range(n)	type tableau	renvoie la plage de variation de l'indice de la dimension n, défaut n=1, dans une boucle : "for i in bus'range loop..."
'reverse_ range(n)	type tableau	renvoie la plage de variation, retournée (to ↔ downto), de l'indice de la dimension n, défaut n=1

Le tableau ci-dessus précise le nom de quelques uns des attributs les plus utilisés, les catégories d'objets qu'ils permettent de qualifier et la valeur renvoyée par l'attribut.

Attributs spécifiques à un système

Chaque système de développement fournit des attributs qui aident à piloter l'outil de synthèse, ou le simulateur, associé au compilateur VHDL.

Ces attributs, qui ne sont évidemment pas standard, portent souvent sur le pilotage de l'optimiseur, permettent de passer au routeur des informations concernant le brochage souhaité, ... etc.

Par exemple :

```
attribute synthesis_off of som4 : signal is true ;
```

permet, avec l'outil « WARP », d'empêcher l'élimination du signal som4 par l'optimiseur.

```
attribute pin_numbers of T_edge:entity is "s:20 ";
```

permet, avec le même outil, de préciser que le port s, de l'entité T_edge, doit être placé sur la broche N° 20 du circuit.

Attributs définis par l'utilisateur

Syntaxe :

déclaration

```
attribute att_nom : type ;
```

spécification

```
attribute nom_att of nom_objet:nom_classe is expression ;
```

utilisation

```
nom_objet'att_nom
```

VI.2.3 Les opérateurs élémentaires

Les opérateurs connus du langage sont répartis en six classes, en fonction de leurs priorités. Dans chaque classe les priorités sont identiques ; les parenthèses permettent de modifier l'ordre d'évaluation des expressions, modifiant ainsi les priorités, et sont obligatoires lors de l'utilisation d'opérateurs non associatifs comme « `nand` ». Le tableau ci-dessous fournit la liste des opérateurs classés par priorités croissantes, de haut en bas :

Classe	Opérateurs	Types d'opérandes	Résultat
Op. logiques	and or nand nor xor	bits ou booléens	bit ou booléen
Op. relationnels	= /= < <= > >=	tous types	booléen
Op. additifs	+ – &	numériques tableaux (concaténation)	numérique tableau
Signe	+ –	numériques	numérique
Opérateurs multiplicatifs	* / mod rem	numériques (restrictions) entiers (restrictions)	numérique entier
Op. divers	not abs **	bit ou booléen numérique numériques (restrictions)	bit ou booléen numérique numérique

Ce tableau appelle quelques remarques :

- Les opérateurs multiplicatifs et l'opérateur d'exponentiation (**) sont soumis à des restrictions, notamment en synthèse où seules les opérations qui se résument à des décalages sont généralement acceptées.
- Certaines librairies standard (int_math et bv_math) surdéfinissent (au sens des langages objets) les opérateurs d'addition et de soustraction pour les étendre au type bit_vector.
- On notera que tous les opérateurs logiques ont la même priorité, il est donc plus que conseillé de parenthéser toutes les expressions qui contiennent des opérateurs différents de cette classe.
- La priorité intermédiaire des opérateurs unaires de signe interdit l'écriture d'expressions comme
  « a * –b », qu'il faut écrire « a * (–b) ».

VI.2.4 Instructions concurrentes

Les instructions concurrentes interviennent à l'intérieur d'une architecture, dans la description du fonctionnement d'un circuit. En raison du parallélisme du langage, ces instructions peuvent être écrites dans un ordre quelconque. Les principales instruction concurrentes sont :

- les affectations concurrentes de signaux,
- les « processus » (décrits précédemment),
- les instanciations de composants
- les instructions « generate »
- les définitions de blocs.

Affectations concurrentes de signaux

Affectation simple

L'affectation simple traduit une simple interconnexion entre deux équipotentielles. L'opérateur d'affectation de signaux (<=) a été vu précédemment :

```
nom_de_signal <= expression_du_bon_type ;
```

Affectation conditionnelle

L'affectation conditionnelle permet de déterminer la valeur de la cible en fonction des résultats de tests logiques :

```
cible <= source_1 when condition_booléenne_1 else
         source_2 when condition_booléenne_2 else
         ...
         source_n ;
```

On notera un danger de confusion entre l'opérateur d'affectation et l'un des opérateurs de comparaison, l'instruction suivante est syntaxiquement juste, mais fournit vraisemblablement un résultat fort différent de celui escompté par son auteur :

```
-- Résultat bizarre :
cible <= source_1 when condition else
cible <= source_2; -- <= est ici une comparaison !
       -- dont le résultat est affecté à
       -- cible si la condition est fausse.
```

Affectation sélective

En fonction des valeurs possibles d'*une* expression, il est possible de choisir la valeur à affecter à un signal :

```
with expression select
cible <= source_1 when valeur_11 | valeur_12 ... ,
         source_2 when valeur_21 | valeur_22 ... ,
         ...
         source_n when others ;
```

Un exemple typique d'affectation sélective est la description d'un multiplexeur.

Instanciation de composant

Le mécanisme qui consiste à utiliser un sous-ensemble (une paire entité-architecture), décrit en VHDL, comme composant dans un ensemble plus vaste est connu sous le nom d'*instanciation*. Trois opérations sont nécessaires :

- Le couple entité-architecture du sous-ensemble doit être créé et annexé à une librairie de l'utilisateur, par défaut la librairie « work ».
- Le sous-ensemble précédent doit être déclaré comme composant dans l'ensemble qui l'utilise, cette déclaration reprend les éléments principaux de l'entité du sous-ensemble.
- Chaque exemplaire du composant que l'on souhaite inclure dans le schéma en cours d'élaboration doit être connecté aux équipotentielles de ce schéma, c'est le mécanisme de l'instanciation.

Syntaxe de la déclaration[15] :

```
component nom_composant -- même nom que l'entité
port ( liste_ports ) ; -- même liste que dans l'entité
end component ;
```

[15]Simplifiée, nous omettons volontairement ici la possibilité de créer des composants « génériques », c'est à dire dont certains paramètres peuvent être fixés au moment de l'instanciation, une largeur de bus, par exemple.

Cette déclaration est à mettre dans la partie déclarative de l'architecture du circuit utilisateur, ou dans un paquetage qui sera rendu visible par une clause « use ».

Instanciation d'un composant :

```
Etiquette : nom port map ( liste_d'association ) ;
```

La liste d'association établit la correspondance entre les équipotentielles du schéma et les ports d'entrée et de sortie du composant. Cette association peut se faire par position, les noms des signaux à connecter doivent apparaître dans l'ordre des ports auxquels ils doivent correspondre, ou explicitement au moyen de l'opérateur d'association « => » :

```
architecture exemple of xyz is
component et
port ( a , b  : in std_logic ;
       a_et_b : out std_logic ) ;
end component ;
   signal s_a, s_b, s_a_et_b, s1, s2, s_1_et_2 : bit;

begin
....
-- utilisation :
et1 : et port map ( s_a , s_b , s_a_et_b ) ;
-- ou :
et2 : et port map (a_et_b => s_1_et_2,a => s1, b => s2) ;
....
end exemple ;
```

En raison de sa simplicité, l'association par position est la plus fréquemment employée dans les cas très simples. L'association explicite est préférable dès que le nombre de ports dépasse deux ou trois.

Generate

Les instructions « generate » permettent de créer de façon compacte des structures régulières, comme les registres ou les multiplexeurs. Elles sont particulièrement efficaces dans des descriptions structurelles.

Une instruction generate permet de dupliquer une bloc d'instructions concurrentes un certain nombre de fois, ou de créer un tel bloc si une condition est vérifiée.

Syntaxe :

```
-- structure répétitive :
etiquette : for variable in debut to fin generate
       instructions concurrentes
end generate [etiquette] ;
```

ou :

```
-- structure conditionnelle :
etiquette : if condition generate
instructions concurrentes
end generate [etiquette] ;
```

Donnons à titre d'exemple le code d'un compteur modulo 16, construit au moyen de bascules T, disposant d'une remise à zéro (raz) et d'une autorisation de comptage (en) actives à '1' :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

ENTITY cnt16 IS
      PORT (ck,raz,en : IN STD_LOGIC;
         s : OUT STD_LOGIC_VECTOR (0 TO 3) );
END cnt16;

ARCHITECTURE struct OF cnt16 IS
  SIGNAL etat : STD_LOGIC_VECTOR(0 TO 3) := (OTHERS => '0');
  SIGNAL inter: STD_LOGIC_VECTOR(0 TO 3) := (OTHERS => '0');
COMPONENT T_edge -- supposé présent dans la librairie work
      port ( T,hor,zero : in std_logic;
         s : out std_logic);
END COMPONENT;

BEGIN
   s <= etat ;
gen_for : for i in 0 to 3 generate
   gen_if1 : if i = 0 generate
      inter(0) <= en ;
   end generate gen_if1 ;
   gen_if2 : if i > 0 generate
      inter(i) <= etat(i - 1) and inter(i - 1) ;
   end generate gen_if2 ;
   comp1_3 : T_edge port map (inter(i),ck,raz,etat(i));
end generate gen_for ;
END struct;
```

Block

Une architecture peut être subdivisée en blocs, de façon à constituer une hiérarchie interne dans la description d'un composant complexe.

Syntaxe :

```
etiquette : block [( expression_de_garde )]
-- zone de déclarations de signaux, composants, etc...
begin
-- instructions concurrentes
end block [etiquette] ;
```

Dans des applications de synthèse, l'intérêt principal des blocs est de permettre de contrôler la portée et la visibilité des noms des objets utilisés (signaux notamment) : un nom déclaré dans un bloc est local à celui-ci. Dans des applications de simulation les blocs permettent en outre de contrôler les instructions qu'ils contiennent par une expression « de garde », de type booléen[16].

VI.2.5 Instructions séquentielles

Les instructions séquentielles sont *internes* aux processus, aux procédures et aux fonctions (pour les deux dernières constructions voir paragraphes suivants). Elles permettent d'appliquer à la description d'une partie d'un circuit une démarche algorithmique, même s'il s'agit d'une fonction purement combinatoire. Les principales instructions séquentielles sont :

- L'affectation séquentielle d'un signal, qui utilise l'opérateur « <= », a une syntaxe qui est identique à celle de l'affectation concurrente simple. Seule la place, dans ou hors d'un module de programme séquentiel, distingue les deux types d'affectation ; cette différence, qui peut sembler mineure, cache des comportements différents : alors que les affectations concurrentes peuvent être écrites dans un ordre quelconque, pour leurs correspondantes séquentielles, rarement utilisées hors d'une structure de contrôle, l'ordre d'écriture n'est pas indifférent.
- L'affectation d'une variable, qui utilise l'opérateur « := », est *toujours* une instruction séquentielle.
- Les tests « if » et « case ».
- Les instructions de contrôle des boucles « loop », « for » et « while ».

[16]Les expressions de garde ne sont pas gérées par tous les compilateurs.

Les instructions de test

Les instructions de tests permettent de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par une ou des expressions. On notera que, dans un processus, si toutes les branches possibles des tests ne sont pas explicitées, une cellule mémoire est générée pour chaque affectation de signal.

L'instruction « if....then....else....end if »

L'instruction `if` permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une ou des* conditions.

Syntaxe :

```
if expression_logique then
instructions séquentielles
[ elsif expression_logique then ]
instructions séquentielles
[ else ]
instructions séquentielles
end if ;
```

Son interprétation est la même que dans les langages de programmation classiques comme C ou Pascal.

L'instruction « case....when....end case »

L'instruction `case` permet de sélectionner une ou des instructions à exécuter, en fonction des valeurs prises par *une* expression.

Syntaxe :

```
case expression is
when choix | choix | ... choix => instruction sequentielle ;
when choix | choix | ... choix => instruction sequentielle ;
....
when others => instruction sequentielle ;
end case ;
```

« | choix », pour « ou ... », et « when others » sont syntaxiquement facultatifs. Les choix représentent différentes valeurs possibles de l'expression testée ; on notera que *toutes* les valeurs possibles doivent être traitées, soit explicitement, soit par l'alternative « others ». Chacune de ces valeurs ne peut apparaître que dans une seule alternative.

Cette instruction est à rapprocher du « switch » de C, ou de « case of » de Pascal.

Les boucles

Les boucles permettent de répéter une séquence d'instructions.

Syntaxe générale

```
[ etiquette : ] [ schéma itératif ] loop
séquence d'instructions
end loop [ etiquette ] ;
```

Trois catégories de boucles existent en VHDL, suivant le schéma d'itération choisi :

- Les boucles simples, sans schéma d'itération, dont on ne peut sortir que par une instruction « exit ».
- Les boucles « for », dont le schéma d'itération précise le nombre d'exécution.
- Les boucles « while », dont le schéma d'itération précise la condition de maintien dans la boucle.

Les boucles « for »

```
[ etiquette : ] for parametre in minimum to maximum loop
séquence d'instructions
end loop [ etiquette ] ;
```

Ou :

```
[ etiquette : ] for parametre in maximum downto minimum loop
séquence d'instructions
end loop [ etiquette ] ;
```

Les boucles « while »

```
[ etiquette : ] while condition loop
séquence d'instructions
end loop [ etiquette ] ;
```

« Next » et « exit »

```
next [ etiquette ] [ when condition ] ;
```

Permet de passer à l'itération suivante d'une boucle.

```
exit [ etiquette ] [ when condition ] ;
```

Permet de provoquer une sortie de boucle.

VI.3. Programmation modulaire

Small is beautyfull, un gros programme ne peut être écrit, compris, testable et testé que s'il est subdivisé en petits modules que l'on met au point indépendamment les uns des autres et rassemblés ensuite. VHDL offre, bien évidemment, cette possibilité.

Chaque module peut être utilisé dans plusieurs applications différentes, moyennant un ajustage de certains paramètres, sans avoir à en réécrire le code. Les outils de base de cette construction modulaire sont les sous programmes, procédures ou fonctions, les *paquetages* et librairies, et les paramètres génériques.

VI.3.1 Procédures et fonctions

Les sous programmes sont le moyen par lequel le programmeur peut se constituer une bibliothèque *d'algorithmes séquentiels* qu'il pourra inclure dans une description. Les deux catégories de sous programmes, procédures et fonctions, diffèrent par les mécanismes d'échanges d'informations entre le programme appelant et le sous programme.

Les fonctions

Une fonction retourne au programme appelant une valeur unique, elle a donc un type. Elle peut recevoir des arguments, exclusivement des signaux ou des constantes, dont les valeurs lui sont transmises lors de l'appel. Une fonction ne peut en aucun cas modifier les valeurs de ses arguments d'appel.

Déclaration :

```
function nom [ ( liste de paramètres formels ) ]
return nom_de_type ;
```

Corps de la fonction :

```
function nom [ ( liste de paramètres formels ) ]
return nom_de_type is
[ déclarations ]
begin
instructions séquentielles
end [ nom ] ;
```

Le corps d'une fonction ne peut pas contenir d'instruction `wait`, les variables locales, déclarées dans la fonction, cessent d'exister dès que la fonction se termine.

Utilisation :

```
nom ( liste de paramètres réels )
```

Lors de son utilisation, le nom d'une fonction peut apparaître partout, dans une expression, où une valeur du type correspondant peut être utilisée.

Exemple

Les librairies d'un compilateur VHDL contiennent un grand nombre de fonctions, dont le programme source est souvent fourni. L'exemple qui suit incrémente de 1 un vecteur d'éléments std_logic. On peut l'utiliser, par exemple, pour créer un compteur binaire.

Déclaration :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

FUNCTION inc_bv (a : STD_LOGIC_VECTOR)
         RETURN STD_LOGIC_VECTOR ;
```

Corps de la fonction :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

FUNCTION inc_bv (a : STD_LOGIC_VECTOR)
         RETURN STD_LOGIC_VECTOR IS
VARIABLE s : STD_LOGIC_VECTOR (a'RANGE);
VARIABLE carry : STD_LOGIC ;
BEGIN
carry := '1';

FOR i IN a'LOW TO a'HIGH LOOP -- les attributs LOW et
                              -- HIGH déterminent les
                              -- dimensions du vecteur.
    s(i) := a(i) XOR carry;
    carry := a(i) AND carry;
END LOOP;
RETURN (s);
END inc_bv;
```

Utilisation dans un compteur :

```
ARCHITECTURE behavior OF counter IS
BEGIN
PROCESS
BEGIN
   WAIT UNTIL rising_edge(clk);
   IF reset = '1' THEN
```

```
        count <= "0000";
    ELSIF load = '1' THEN
        count <= dataIn;
    ELSE
        count <= inc_bv(count); -- increment du bit vector
    END IF;
END process;
END behavior;
```

Les procédures

Une procédure, comme une fonction, peut recevoir du programme appelant des arguments : constantes, variables ou signaux. Mais ces arguments peuvent être déclarés de modes « in », « inout » ou « out » (sauf les constantes qui sont toujours de mode « in »), ce qui autorise une procédure à renvoyer un nombre quelconque de valeurs au programme appelant.

Déclaration :

```
procedure nom [ ( liste de paramètres formels ) ];
```

Corps de la procédure :

```
procedure nom [ ( liste de paramètres formels ) ] is
[ déclarations ]
begin
instructions séquentielles
end [ nom ] ;
```

Dans la liste des paramètres formels, la nature des arguments doit être précisée :

```
procedure exemple ( signal a, b : in bit ;
                    signal s : out bit ) ;
```

Le corps d'une procédure peut contenir une instruction `wait`, les variables locales, déclarées dans la procédure, cessent d'exister dès que la procédure se termine.

Utilisation :

```
nom ( liste de paramètres réels ) ;
```

Une procédure peut être appelée par une instruction concurrente ou par une instruction séquentielle, mais si l'un de ses arguments est une variable, elle ne peut être appelée que par une instruction séquentielle. La correspondance entre paramètres réels (dans l'appel) et paramètres formels (dans la description de la procédure) peut se faire par position, ou par associations de noms :

```
exemple (entree1, entree2, sortie) ;
```

Ou :

```
exemple (s => sortie, a => entree1, b => entree2);
```

VI.3.2 Les paquetages et les librairies

Un `paquetage` permet de rassembler des déclarations et des sous programmes, utilisés fréquemment dans une application, dans un module qui peut être compilé à part, et rendu visible par l'application au moyen de la clause `use`.

Un `paquetage` est constitué de deux parties : la déclaration, et le corps (*body*).

- La déclaration contient les informations publiques dont une application a besoin pour utiliser correctement les objets décrits par le paquetage[17] : essentiellement des déclarations, des définitions de types, des définitions de constantes ...etc.
- Le corps, qui n'existe pas obligatoirement, contient le code des fonctions ou procédures définies par le paquetage, s'il en existe.

L'utilisation d'un paquetage se fait au moyen de la clause `use` :

```
use work.mes_fonctions.all; -- rend le paquetage
      -- mes_fonctions, de la librairie work,
      -- visible dans sa totalité.
```

Le mot clé `work` indique l'ensemble des librairies accessibles, par défaut, au programmeur. Ce mot cache, notamment, des chemins d'accès à des répertoires de travail. Ces chemins sont gérés par le système de développement, et l'utilisateur n'a pas besoin d'en connaître les détails. Le nom composé qui suit la clause use doit être compris comme une suite de filtres : « utiliser tous les éléments du module int_math de la librairie work ».

Les paquetages prédéfinis

Un compilateur VHDL est toujours assorti de librairies, décrites par des paquetages, qui offrent à l'utilisateur des outils variés :

- Définitions de types, et fonctions de conversions entre types : VHDL est un langage objet, fortement typé. Aucune conversion de type implicite n'est

[17]On peut rapprocher la partie visible d'un package des fichiers « *.h » du langage C, ces fichiers contiennent, entre autres, les prototypes des objets, variables ou fonctions, utilisés dans un programme. La clause « use » de VHDL est un peu l'équivalent, dans cette comparaison, de la directive #include <xxx.h> du C.

autorisée dans les expressions, mais une librairie peut offrir des fonctions de conversion explicites, et redéfinir les opérateurs élémentaires pour qu'ils acceptent des opérandes de types variés. Un bus, par exemple, peut être vu, dans le langage, comme un vecteur (tableau à une dimension) de bits, et il est possible d'étendre les opérateurs arithmétiques et logiques élémentaires pour qu'ils agissent sur un bus, vu comme la représentation binaire d'un nombre entier (package `numeric_std` de la librairie IEEE).

- Les blocs structurels des circuits programmables, notamment les cellules d'entrées–sorties, peuvent être déclarés comme des composants que l'on peut inclure dans une description. Une porte trois états, par exemple, sera vue, dans une architecture, comme un composant dont l'un des ports véhicule des signaux de type particulier : aux deux états logiques vient se rajouter un état haute impédance. L'emploi d'un tel opérateur dans un schéma nécessite, outre la description du composant, une fonction de conversion entre signaux logiques et signaux « trois états ».
- Un simulateur doit pouvoir résoudre, ou indiquer, les conflits éventuels. Les signaux utilisés en simulation ne sont pas, pour cette raison, de type binaire : on leur attache un type énuméré plus riche qui rajoute aux simples valeurs '0' et '1' la valeur 'inconnue', des nuances de force entre les sorties standard et les sorties collecteur ouvert, etc. (librairie IEEE)
- La bibliothèque standard offre également des procédures d'usage général comme les moyens d'accès aux fichiers, les possibilités de dialogue avec l'utilisateur, messages d'erreurs, par exemple.

Les paquetages de la librairie IEEE

La librairie IEEE joue un rôle fédérateur et remplace tous les dialectes locaux. En cours de généralisation, y compris en synthèse, elle définit un type de base à neuf états, `std_ulogic` (présenté précédemment comme exemple de type énuméré) et des sous-types dérivés simples et structurés (vecteurs). Des fonctions et opérateurs surchargés permettent d'effectuer des conversions et de manipuler les vecteurs comme des nombres entiers.

A l'heure actuelle la librairie IEEE comporte trois paquetages dont nous examinerons plus en détail certains aspects au paragraphe II-7 :

- std_logic_1164 définit les types, les fonctions de conversion, les opérateurs logiques et les fonctions de recherches de fronts rising_edge() et falling_edge().
- numeric_bit définit les opérateurs arithmétiques agissant sur des bit_vector interprétés comme des nombres entiers.
- numeric_std définit les opérateurs arithmétiques agissant sur des std_logic_vector interprétés comme des nombres entiers.

Le paquetage `ieee.std_logic_1164` définit le type `std_ulogic` qui est le type de base de la librairie IEEE :

```
type std_ulogic is ( 'U',  -- Uninitialized
```

```
                    'X',  -- Forcing  Unknown
                    '0',  -- Forcing  0
                    '1',  -- Forcing  1
                    'Z',  -- High Impedance
                    'W',  -- Weak     Unknown
                    'L',  -- Weak     0
                    'H',  -- Weak     1
                    '-'   -- Don't care
                   );
```

Le sous-type `std_logic`, qui est le plus utilisé, est associé à la fonction de résolution `resolved` :

```
function resolved ( s : std_ulogic_vector )
                  return std_ulogic;
subtype std_logic is resolved std_ulogic;
```

Cette fonction de résolution utilise une table de gestion des conflits qui reproduit les forces respectives des valeurs du type :

```
type stdlogic_table is array(std_ulogic, std_ulogic)
                      of std_ulogic;
constant resolution_table : stdlogic_table := (
-----------------------------------------------------------
--| U    X    0    1    Z    W    L    H    -        |   |
-----------------------------------------------------------
 ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
 ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
 ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
 ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
 ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
 ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
 ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
 ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
 ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
  );
```

Le paquetage définit également des vecteurs :

```
type std_logic_vector is array ( natural range <> )
                        of std_logic;

type std_ulogic_vector is array ( natural range <> )
                         of std_ulogic;
```

Et les sous-types résolus `X01`, `X01Z`, `UX01` et `UX01Z`.

Des fonctions de conversions permettent de passer du type binaire aux types IEEE et réciproquement, ou d'un type IEEE à l'autre :

```
function To_bit ( s : std_ulogic; xmap : bit := '0')
                 return bit;
function To_bitvector ( s : std_logic_vector ;
```

```
                        xmap : bit := '0') return bit_vector;
function To_bitvector ( s : std_ulogic_vector;
                        xmap : bit := '0') return bit_vector;
function To_StdULogic ( b : bit ) return std_ulogic;
function To_StdLogicVector ( b : bit_vector )
                             return std_logic_vector;
function To_StdLogicVector ( s : std_ulogic_vector )
                             return std_logic_vector;
function To_StdULogicVector ( b : bit_vector )
                              return std_ulogic_vector;
```

Par défaut les fonctions comme `To_bit` remplacent, au moyen du paramètre `xmap`, toutes les valeurs autres que `'1'` et `'H'` par `'0'`.

La détection d'un front d'horloge se fait au moyen des fonctions :

```
function rising_edge (signal s : std_ulogic) return boolean;
function falling_edge (signal s : std_ulogic) return
boolean;
```

Tous les opérateurs logiques sont surchargés pour agir sur les types IEEE comme sur les types binaires. Ces opérateurs retournent les valeurs *fortes* `'0'`, `'1'`, `'X'`, ou `'U'`.

Nombres et vecteurs

Un nombre entier peut être assimilé à un vecteur, dont les éléments sont les coefficients binaires de son développement polynomial en base deux. Restent à définir sur ces objets les opérateurs arithmétiques, ce que permet la surcharge d'opérateurs.

Les paquetages `numeric_std` et `numeric_bit` correspondent à l'utilisation, sous forme de nombres, de vecteurs dont les éléments sont des types `std_logic` et `bit`, respectivement. Comme les types définis dans ces paquetages portent les mêmes noms, ils ne peuvent pas être rendus visibles simultanément dans un même module de programme ; il faut choisir un contexte ou l'autre.

Les deux paquetages ont pratiquement la même structure, et définissent les types `signed` et `unsigned` :

```
-- ieee.numeric_bit :
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;

-- ieee.numeric_std :
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Les vecteurs doivent être rangés dans l'ordre descendant (`downto`) de l'indice, de sorte que le coefficient de poids fort soit toujours écrit à gauche, et que le coefficient de poids faible, d'indice `0`, soit à droite, ce qui est l'ordre naturel. La représentation interne des nombres signés correspond au code complément à deux,

dans laquelle le chiffre de poids fort est le bit de signe (`'1'` pour un nombre négatif, `'0'` pour un nombre positif ou nul).

Les opérations prédéfinies dans ces paquetages, agissant sur les types `signed` et `unsigned`, sont :

- Les opérations arithmétiques.
- Les comparaisons.
- Les opérations logiques pour numeric_std, elles sont natives pour les vecteurs de bits.
- Des fonctions de conversion entre nombres et vecteurs :

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL)
                    return SIGNED;
```

- Une fonction de recherche d'identité (std_match) qui utilise l'état don't care du type std_logic comme joker.
- Les recherches de fronts (rising_edge et falling_edge), agissant sur le type bit, dans numeric_bit.

Des opérations prédéfinies

Les opérandes et les résultats des opérateurs arithmétiques et relationnels appellent quelques commentaires. Ces opérateurs acceptent comme opérandes deux vecteurs ou un vecteur et un nombre. Dans le cas de deux vecteurs dont l'un est signé, l'autre pas, il est à la charge du programmeur de prévoir les conversions de types nécessaires.

Les opérateurs arithmétiques

L'addition et la soustraction sont faites sans aucun test de débordement, ni génération de retenue finale. La dimension du résultat est celle du plus grand des opérandes, quand l'opération porte sur deux vecteurs, ou celle du vecteur passé en argument dans le cas d'une opération entre un vecteur et un nombre. Le résultat est donc calculé implicitement modulo 2^n, où n est la dimension du vecteur retourné. Ce modulo implicite allège, par exemple, la description d'un compteur binaire, évitant au programmeur de prévoir explicitement l'incrémentation modulo la taille du compteur.

La multiplication retourne un résultat dont la dimension est calculée pour pouvoir contenir le plus grand résultat possible : somme des dimensions des opérandes moins un, dans le cas de deux vecteurs, double de la dimension du vecteur passé en paramètre moins un dans le cas de la multiplication d'un vecteur par un nombre.

Pour la division entre deux vecteurs, le quotient a la dimension du dividende, le reste celle du diviseur. Quand les opérations portent sur un nombre et un vecteur, la dimension du résultat ne peut pas dépasser celle du vecteur, que celui-ci soit dividende ou diviseur.

Les opérateurs relationnels

Quand on compare des vecteurs interprétés comme étant des nombres, les résultats peuvent être différents de ceux que l'on obtiendrait en comparant des vecteurs sans signification. Le tableau ci-dessous donne quelques exemples de résultats en fonction des types des opérandes :

Expression	Types des opérandes		
	bit_vector	unsigned	signed
"001" = "00001"	FALSE	TRUE	TRUE
"001" > "00001"	TRUE	FALSE	FALSE
"100" < "01000"	FALSE	TRUE	TRUE
"010" < "10000"	TRUE	TRUE	FALSE
"100" < "00100"	FALSE	FALSE	TRUE

Ces résultats se comprennent aisément si on garde à l'esprit que la comparaison de vecteurs ordinaires, sans signification numérique, se fait de gauche à droite sans notion de poids attaché aux éléments binaires.

Compteur décimal

Comme illustration de l'utilisation de la librairie IEEE, donnons le code source d'une version possible de la décade, instanciée comme composant dans un compteur décimal :

```
library ieee ;
use ieee.numeric_bit.all ;

ARCHITECTURE vecteur OF decade IS
   signal countTemp: unsigned(3 downto 0) ;
begin
   count <= chiffre(to_integer(countTemp)) ; -- conversion
   dix <= en when countTemp = 9 else '0' ;
   incre : PROCESS
      BEGIN
      WAIT UNTIL rising_edge(clk) ;
         if raz = '1' then
            countTemp <= X"0" ;
-- ou :     countTemp <= to_unsigned(0) ;
         elsif en = '1' then
            countTemp <= (countTemp + 1) mod 10 ;
         end if ;
      END process incre ;
END vecteur ;
```

L'intérêt de ce programme réside dans l'aspect évident des choses, le signal `countTemp`, un vecteur d'éléments binaires, est manipulé dans des opérations arithmétiques exactement comme s'il s'agissait d'un nombre. Seules certaines opérations de conversions rappellent les différences de nature entre les types `unsigned` et `integer`.

Les paquetages créés par l'utilisateur

L'utilisateur peut créer ses propres paquetages. Cette possibilité permet d'assurer la cohérence des déclarations dans une application complexe, évite d'avoir à répéter un grand nombre de fois ces mêmes déclarations et donne la possibilité de créer une librairie de fonctions et procédures adaptée aux besoins des utilisateurs.

La syntaxe de la déclaration d'un paquetage est la suivante :

```
package identificateur is
déclarations de types, de fonctions,
de composants, d'attributs,
clause use, ...  etc
end [identificateur] ;
```

S'il existe, le corps du paquetage doit porter le même nom que celui qui figure dans la déclaration :

```
package body identificateur is
   corps des sous programmes déclarés.
end [identificateur] ;
```

Dans l'exemple qui suit, on réalise un compteur au moyen de deux bascules, dans une description structurelle. La déclaration du composant bascule est mise dans un paquetage :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

package T_edge_pkg is
COMPONENT T_edge -- une bascule T avec mise à 0.
   port ( T,hor,raz : in std_logic;
                  s : out std_logic);
END COMPONENT;
end T_edge_pkg ;
```

Le compteur proprement dit :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;

ENTITY cnt2 IS
   PORT (ck,razero,en : IN STD_LOGIC;
                    s : OUT STD_LOGIC_VECTOR (0 TO 1)
      );
END cnt2;
```

```
use work.T_edge_pkg.all ;
      -- rend le contenu du package
      -- précédent visible.

ARCHITECTURE struct OF cnt2  IS
   SIGNAL etat : STD_LOGIC_VECTOR(0 TO 1) := "00";
   signal inter: std_logic := '0' ;

BEGIN
   s <= etat ;
   inter <= etat(0) and en ;
   g0 : T_edge port map (en,ck,razero,etat(0));
   g1 : T_edge port map (inter,ck,razero,etat(1));
END struct;
```

Les librairies

Une librairie est une collection de modules VHDL qui ont déjà été compilés. Ces modules peuvent être des paquetages, des entités ou des architectures.

Une librairie par défaut, work, est systématiquement associée à l'environnement de travail de l'utilisateur. Ce dernier peut ouvrir ses propres librairies par la clause `library` :

```
library nom_de_la_librairie ;
```

La façon dont on associe un nom de librairie à un, ou des, chemins, dans le système de fichiers de l'ordinateur, dépend de l'outil de développement utilisé.

VI.3.3 Les paramètres génériques

Lorsque l'on crée le couple entité-architecture d'un opérateur, que l'on souhaite utiliser comme composant dans une construction plus large, il est parfois pratique de pouvoir laisser certains paramètres modifiables par le programme qui utilise le composant. De tels paramètres, dont la valeur réelle peut n'être fixée que lors de l'instanciation du composant, sont appelés paramètres génériques.

Un paramètre générique se déclare au début de l'entité, et peut avoir une valeur par défaut :

```
generic (nom : type [ := valeur_par_defaut ] ) ;
```

La même déclaration doit apparaître dans la déclaration de composant, mais au moment de l'instanciation la taille peut être modifiée par une instruction « generic map », de construction identique à l'instruction « port map », précédemment rencontrée :

```
Etiquette : nom generic map ( valeurs )
       port map ( liste_d'association ) ;
```

Dans l'exemple ci-dessous, on réalise un compteur, sur 4 bits par défaut, qui est ensuite instancié comme un compteur 8 bits. Bien évidemment, le code du compteur ne doit faire aucune référence explicite à la valeur par défaut.

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL ;

entity compteur is
generic (taille : integer := 4 ) ;
   port ( hor : in std_logic ;
        sortie : out std_logic_vector(taille - 1 downto 0));
end compteur ;

architecture simple of compteur is
   signal etat : unsigned(taille - 1 downto 0) ;
begin
sortie <= std_logic_vector(etat) ;
process
begin
   wait until rising_edge(hor) ;
   etat <= etat + 1 ;
end process ;
end simple ;
```

Ce compteur est instancié comme un compteur 8 bits :

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL, IEEE.NUMERIC_STD.ALL ;

entity compt8 is
   port (ck : in std_logic ;
         val : out std_logic_vector(7 downto 0) ) ;
end compt8 ;

architecture large of compt8 is
component compteur
   generic (taille : integer ) ;
   port ( hor : in std_logic ;
        sortie : out std_logic_vector(taille - 1 downto 0));
end component ;
begin
u1 : compteur
   generic map (8)
   port map (ck , val) ;
end large ;
```

Les paramètres génériques prennent toute leur efficacité quand leur emploi est associé à la création de librairies de composants, décrits par des paquetages. Il est alors possible de créer des fonctions complexes au moyen de programmes construits de façon hiérarchisée, chaque niveau de la hiérarchie pouvant être mis au point et testé indépendamment de l'ensemble.

VI.4. En guise de conclusion

VHDL est un langage qui peut déconcerter, au premier abord, le concepteur de systèmes numériques, plus habitué aux raisonnements traditionnels sur des schémas que familier des langages de description abstraite. Il est vrai que le langage est complexe, et peut présenter certains pièges, la description des horloges en est un exemple. Nous espérons avoir aidé le lecteur à gagner un peu de temps dans sa découverte, et lui avoir mis en évidence quelques uns des chausse-trappes classiques.

Ayant fait l'effort de « rentrer dedans », l'utilisateur découvre que ce type d'approche est d'une très grande souplesse, et d'une efficacité redoutable. Des problèmes de synthèse qui pouvaient prendre des heures de calcul, dans une démarche traditionnelle, sont traités en quelques lignes de programme.

Renouvelons ici la mise en garde du début de ce chapitre : n'oubliez jamais que vous êtes en train de créer un circuit, et que le meilleur des compilateurs ne peut que traduire la complexité sous-jacente de vos équations, il n'augmentera pas la capacité de calcul des circuits que vous utilisez. Le simple programme de description d'un additionneur 4 bits, comme le 74_283 :

```
ENTITY addit IS
PORT   (a,b: IN INTEGER RANGE 0 TO 15;
        cin: IN INTEGER RANGE 0 TO 1;
        som: OUT INTEGER RANGE 0 TO 31);
END addit;

ARCHITECTURE behavior OF addit IS

BEGIN
   som <= a+b+cin;
END behavior;
```

génère plus d'une centaine de termes, quand ses équations sont ramenées brutalement à une somme de produits logiques. Charge reste à l'utilisateur de piloter l'optimiseur de façon un peu moins sommaire que de demander la réduction de la somme à une expression canonique en deux couches logiques.

Exercices

Du code VHDL au schéma.

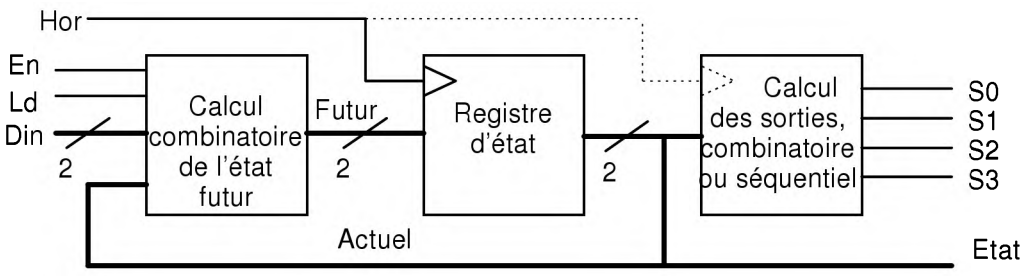
On considère le programme VHDL suivant qui décrit le fonctionnement d'une bascule :

```
entity basc is
port ( T,hor,raz : in bit;
       s : out bit);
end basc;

architecture primitive of basc is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until (hor = '1') ;
if(raz = '1') then
       etat <= '0';
elsif(T = '1') then
       etat <= not etat;
end if;
end process;
end primitive;
```

1. A quoi reconnait-on qu'il s'agit d'un circuit séquentiel synchrone ?
2. La commande « raz » est-elle synchrone ou asynchrone ?
3. Etablir le diagramme de transition de cette bascule.
4. Déduire du diagramme précédent les équations logiques et le schéma d'une réalisation avec une bascule D, avec une bascule J-K.

compteur modulo 4 à sorties décodées.



Le code d'une première version du compteur est le suivant :

```
entity compteur is
  port ( hor, en, ld: in bit; din: in bit_vector(1 downto 0) ;
  etat:out bit_vector(1 downto 0);s:out bit_vector(0 to 3) );
end compteur ;
use work.int_math.all;
architecture comporte of compteur is
signal actuel : bit_vector(1 downto 0) ;
begin
etat <= actuel ;        -- instruction 1
with actuel select            -- instruction 2
   s <= "1000" when "00", "0100" when "01",
        "0010" when "10", "0001" when "11"; -- fin instruction 2
process                       -- instruction 3
begin
      wait until (hor = '1');
      if(ld = '1') then       actuel <= din ;
      elsif (en = '1') then   actuel <= actuel + 1 ;
      end if ;
end process ;                 -- fin instruction 3
end comporte ;
```

- Peut-on permuter les instructions (ou blocs d'instructions) 1, 2 et 3 ?
- **Dans** le processus peut-on permuter les instructions, même si on veille à conserver une syntaxe correcte ?
- Réécrire l'instruction 2 en utilisant une affectation conditionnelle au lieu d'un sélecteur parallèle.
- Réécrire le programme en créant trois processus qui respectent le découpage donné dans le synoptique précédent.
- Réécrire le programme précédent en calculant les sorties s(i) dans un processus synchrone, mais en veillant à ce que leurs valeurs restent identiques à celles décrites précédemment (le piège réside dans un éventuel décalage d'une période d'horloge).

Bibliographie

Circuits et opérateurs logiques

Tran Tien Lang, *Electronique numérique*, Masson – 1995.

J.M. Bernard, J. Hugon, *Pratique des circuits logiques*, Eyrolles – 1987.

J.P. Vabre, *Analyse binaire et circuits logiques*, Editions Techniques – 1980.

P. Horowitz, W. Hill, *The art of electronics*, Cambridge University Press – 1983.

D.A. Hodges, H.G. Jackson, *Analysis and design of digital integrated circuits*, McGraw Hill – 1988.

J. Millman, A. Grabel, *Microelectronics*, McGraw Hill – 1988 ; existe en traduction française chez le même éditeur.

Programmable Logic, Cypress – 1995.

The Programmable Logic Data Book, Xilinx.

Data Book, ALTERA.

The TTL Data Book, Texas Instrument.

Méthodes de synthèse

J.M. Bernard, *Conception structurée des systèmes logiques*, Eyrolles – 1987 .

A. Jacques, J.C. Lafont, J.P. Vabre, *Logique programmée et Grafcet*, Ellipse – 1987.

PAL device Handbook, Advanced Micro Devices – 1988; référence ancienne qui contient une excellente introduction à la synthèse logique.

Applications Handbook, Cypress Semiconductor – 1991.

FPGA Applications Handbook, Texas Instrument – 1993.

F.J. Hill, G.R. Peterson, *Computer aided logical design with emphasis on VLSI*, Jonh Wiley & sons – 1993.

C. Mead, L. Conway, *Introduction to VLSI systems*, Addison-Wesley – 1980.

Langage VHDL

J. Weber, S. Moutault, M. Meaudre, *Le langage VHDL – du langage au circuit, du circuit au langage*, 3ème éd., Dunod – 2007.

R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, *VHDL du langage à la modélisation*, Presses Polytechniques et Universitaires Romandes – 1990.

Z. Navabi, *VHDL : Analysis and Modeling of Digital Systems*, McGraw Hill – 1993.

J. Armstrong, *Chip Level Modelling in VHDL*, Prentice Hall – 1988.

Warp VHDL Synthesis Reference, Cypress – 1995.

Introduction to VHDL, Mentor Graphics – 1992.

VHDL Reference Manual, Mentor Graphics – 1994.

AutoLogic VHDL Reference Manual, Mentor Graphics – 1994.

IEEE Standard VHDL Language Reference Manual, Std 1076-1993, IEEE – 1993.

IEEE Standard 1076 VHDL Tutorial, CLSI – 1989.

Index

VHDL :