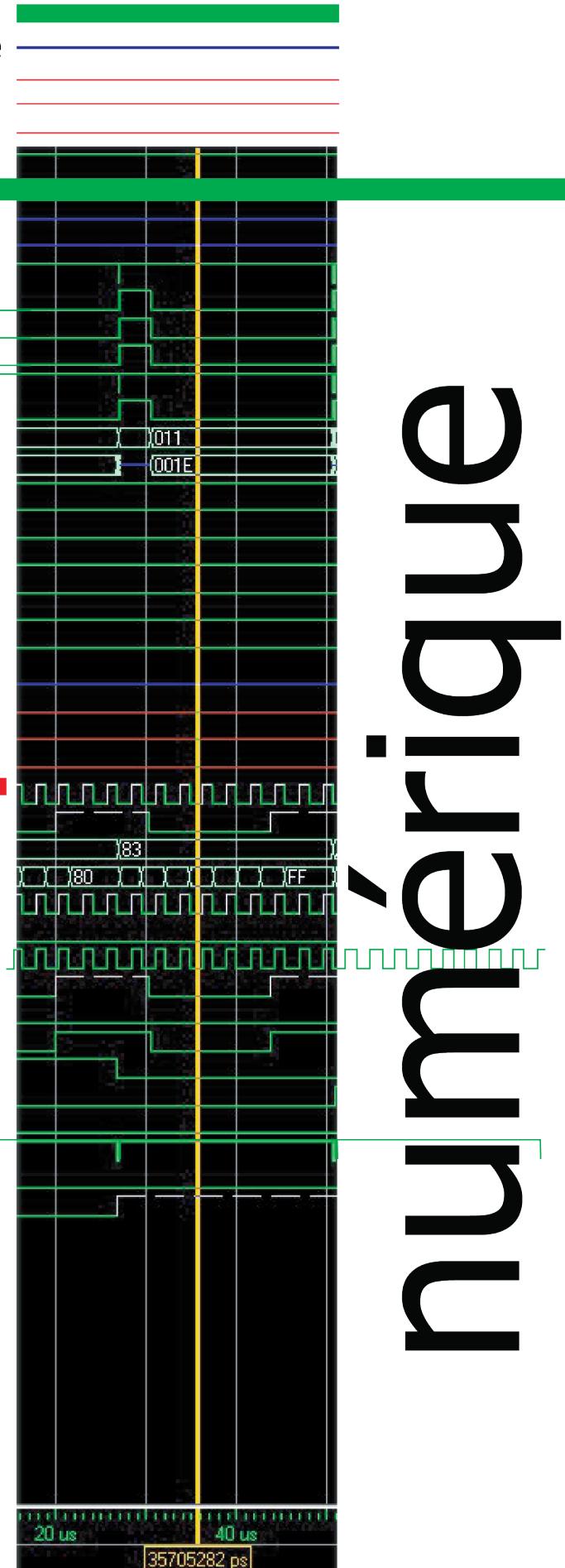


Electronique



Yves Meyer



INTRODUCTION	9
SYSTÈMES DE NUMÉRATION ET CODES	11
INTRODUCTION	11
REPRÉSENTATION DES NOMBRES ET NUMÉRATION DE POSITION	11
CONVERSION BINAIRE - DECIMAL	12
CONVERSION DECIMAL - BINAIRE	12
<i>Répétition de divisions</i>	12
Système de numération OCTAL	13
<i>Conversion octal-décimal</i>	13
<i>Conversion décimal-octal</i>	13
<i>Conversion octal-binaire</i>	13
<i>Conversion binaire-octal</i>	14
Système de numération HEXADECIMAL	14
<i>Conversion hexadécimal-décimal</i>	15
<i>Conversion décimal-hexadécimal</i>	15
<i>Conversion hexadécimal-binaire</i>	15
<i>Conversion binaire-hexadécimal</i>	15
<i>Comptage hexadécimal</i>	16
<i>Utilité du système hexadécimal</i>	16
CODE BCD, SOIT BINARY CODED DECIMAL	16
<i>Comparaison entre code BCD et nombre binaire</i>	17
LES CODES ALPHANUMERIQUES	18
<i>Code ASCII</i>	18
ARITHMETIQUE BINAIRE.....	19
INTRODUCTION	19
ADDITION BINAIRE	19
ÉCRITURE DES NOMBRES SIGNES	20
<i>Notation en complément à 1</i>	20
<i>Notation en complément à 2</i>	21
<i>Etude de nombres binaires signés en complément à 2</i>	21
<i>Cas spécial de la notation en complément à 2</i>	22
ADDITION EN COMPLEMENT A 2	22
<i>Cas 1: deux nombres positifs</i>	22
<i>Cas 2: nombre positif et nombre négatif plus petit</i>	22
<i>Cas 3: nombre positif et nombre négatif plus grand</i>	23
<i>Cas 4: deux nombres négatifs</i>	23
<i>Cas 5 : nombres égaux et opposés</i>	23
SOUSTRACTION: COMPLEMENT A 2	24
<i>Dépassement (overflow)</i>	24
MULTIPLICATION DE NOMBRES BINAIRES	25
<i>Multiplication en complément à 2</i>	25
DIVISION BINAIRE	26
ADDITION EN BCD	26
<i>Somme égale ou inférieure à 9</i>	26
<i>Somme supérieure à 9</i>	27
PORTE LOGIQUES ET ALGEBRE BOOLEENNE	28
INTRODUCTION	28
CONSTANTES ET VARIABLES BOOLEENNES	28
TABLES DE VÉRITÉ	29
L'OPÉRATION OU (OR)	29
<i>La porte OU (OR)</i>	30
L'OPÉRATION ET (AND)	30
<i>La Porte ET (AND)</i>	30
L'OPÉRATION NON (NOT)	30

<i>Le circuit INVERSEUR (NOT)</i>	31
MISE SOUS FORME ALGEBRIQUE DES CIRCUITS LOGIQUES	31
<i>Circuits renfermant des INVERSEURS</i>	32
MATERIALISATION DE CIRCUITS A PARTIR D'EXPRESSIONS BOOLEENNES.....	32
PORTE NI (NOR) ET PORTES NON-ET (NAND)	33
<i>La porte NI (NOR)</i>	33
<i>La porte NON-ET (NAND)</i>	33
<i>Symbolique des fonctions</i>	34
ALGÈBRE DE BOOLE	35
<i>Postulats</i>	35
<i>Théorèmes</i>	36
<i>Théorèmes pour plusieurs variables</i>	36
THEOREMES DE DE MORGAN.....	37
CIRCUITS LOGIQUES COMBINATOIRE	38
INTRODUCTION.....	38
SOMME DE PRODUITS	38
SIMPLIFICATION DES CIRCUITS LOGIQUES	38
SIMPLIFICATION ALGEBRIQUE	39
CONCEPTION DE CIRCUITS LOGIQUES COMBINATOIRES	39
LA METHODE DES DIAGRAMMES DE KARNAUGH	40
<i>La forme du diagramme de Karnaugh</i>	40
REUNION.....	41
<i>Réunion de doublets (de paires)</i>	42
<i>Réunion de quartets (groupes de quatre)</i>	42
<i>Réunion d'octets (groupes de huit)</i>	43
<i>Le processus de simplification au complet</i>	43
<i>Les conditions indifférentes</i>	44
CIRCUITS OU EXCLUSIF (XOR) ET NI EXCLUSIF (XNOR)	45
<i>OU exclusif (XOR)</i>	45
<i>NI exclusif (XNOR)</i>	45
FONCTIONS COMBINATOIRES STANDARDS.....	46
MULTIPLEXEUR (MUX)	46
DECODEUR (X/Y)	47
<i>Additionneur binaire parallèle</i>	48
CONCEPTION D'UN ADDITIONNEUR COMPLET	49
ASPECTS TECHNIQUES CIRCUITS LOGIQUES COMBINATOIRES	51
TECHNOLOGIE	51
<i>La représentation des états logiques</i>	51
<i>Les familles logiques</i>	51
<i>Terminologie des circuits numériques</i>	52
<i>Interface CMOS - TTL</i>	55
<i>Interface TTL - CMOS</i>	56
<i>Collecteur ouvert (open collector)</i>	56
<i>Porte trois états</i>	58
LE LANGAGE DE DESCRIPTION VHDL	59
INTRODUCTION.....	59
DESIGN FLOW	59
<i>La conception, décomposition hiérarchique</i>	59
<i>Le codage VHDL</i>	59
<i>La Compilation</i>	60
<i>La simulation fonctionnelle</i>	60
<i>La Synthèse</i>	60
<i>L'implémentation (Fitter ou Place & Route)</i>	60
<i>La simulation post-implémentation</i>	60

<i>Programmation du circuit logique</i>	60
LES AVANTAGES DU VHDL	61
<i>Langage complet.....</i>	61
<i>Langage indépendant</i>	61
<i>Langage flexible</i>	61
<i>Langage moderne</i>	61
<i>Langage standard.....</i>	61
<i>Langage ouvert.....</i>	61
HISTORIQUE	61
REGLES D'ECRITURE DE VHDL	62
<i>Commentaires.....</i>	62
<i>Majuscules et minuscules.....</i>	62
<i>Identificateurs.....</i>	63
<i>Noms réservés</i>	63
<i>Espaces, sauts de lignes</i>	63
<i>Fin d'instruction.....</i>	64
<i>Les constantes littérales.....</i>	64
LA PAIRE ENTITY/ARCHITECTURE	65
DECLARATION D'ENTITE (ENTITY)	65
<i>Syntaxe</i>	65
<i>Mode</i>	65
<i>Type.....</i>	66
LA BIBLIOTHEQUE (LIBRARY)	69
LE PAQUETAGE (PACKAGE)	69
L'ARCHITECTURE	70
<i>Niveaux de description.....</i>	70
<i>Syntaxe d'une architecture.....</i>	70
<i>Zone déclarative : Signaux, constantes, alias, variables, composants.....</i>	71
<i>Les opérateurs</i>	73
<i>Surcharge des opérateurs</i>	74
INSTRUCTIONS CONCURRENTES	75
<i>Assignation inconditionnelle</i>	75
<i>Assignation conditionnelle</i>	75
<i>Assignation sélective.....</i>	77
<i>Instanciation de composants.....</i>	78
<i>Le processus (process)</i>	80
INSTRUCTIONS SEQUENTIELLES	81
<i>Assignation inconditionnelle de signal</i>	81
<i>Assignation conditionnelle (if...then...else).....</i>	81
<i>Assignation sélective.....</i>	82
<i>Mémorisation implicite</i>	82
<i>Boucles.....</i>	83
LES SOUS-PROGRAMMES: PROCEDURES ET FONCTIONS	84
<i>Syntaxe</i>	84
<i>Définition</i>	84
LES BASCULES.....	85
INTRODUCTION	85
DEFINITION DU SYSTEME SEQUENTIEL	85
BASCULE R-S EN PORTES NAND	85
BASCULE R-S EN PORTES NOR	86
BASCULE R-S AVEC ENABLE	87
L'ELEMENT MEMOIRE D (D LATCH)	88
SIGNAL D'HORLOGE ET BASCULES SYNCHRONES	88
BASCULE D SYNCHRONE (D FLIP-FLOP)	89
ENTRÉES ASYNCHRONES	90
CONSIDÉRATIONS SUR LA SYNCHRONISATION DES BASCULES.....	91

<i>La métastabilité</i>	91
<i>Temps de stabilisation (setup time) et temps de maintien (hold time)</i>	91
<i>Temps de propagation</i>	92
BASCULE T (TOGGLE FLIP FLOP)	92
DESCRIPTION D'UNE BASCULE D AVEC RESET ASYNCHRONE EN VHDL.....	93
CHABLON POUR LA DESCRIPTION D'UN SYSTEME SEQUENTIEL SYNCHRONE EN VHDL.....	94
RÈGLES DU DESIGN SYNCHRONE	94
STRUCTURE DE BASE DES SYSTÈMES SÉQUENTIELS SYNCHRONES	94
BASCULE D AVEC ENABLE.....	95
<i>Description VHDL</i>	95
RESET ASYNCHRONE OU SYNCHRONE	96
<i>Reset asynchrone</i>	96
<i>Reset synchrone</i>	97
LES BANCS DE TEST VHDL (TESTBENCH)	98
SIMULATION A TROIS FICHIERS OU PLUS	98
SIMULATION A DEUX FICHIERS	99
PRINCIPE DE BASE D'UNE SIMULATION.....	99
ANALYSE DU FICHIER TESTBENCH CHABLON	101
<i>Base de temps et signal d'horloge</i>	101
<i>Procédure d'initialisation (init)</i>	102
<i>Procédures test_signal et test_vecteur</i>	102
<i>Partie principale</i>	102
SYNCHRONISATION DES ENTREES EXTERNES ASYNCHRONES	104
DESCRIPTION VHDL	104
SYNCHRONISATION ET DETECTION DES FLANCS D'UN SIGNAL EXTERNE ASYNCHRONE.....	105
SYSTEME A PLUSIEURS HORLOGES	106
<i>Handshaking</i>	106
LES REGISTRES	107
STRUCTURES DE BASE DES REGISTRES.....	107
REGISTRE TAMON (REGISTRE PARALLELE).....	107
<i>Description VHDL d'un registre tampon</i>	108
REGISTRES MEMOIRE	109
<i>Bus de données bidirectionnel</i>	109
<i>Structure d'un registre mémoire</i>	110
<i>Description VHDL</i>	111
<i>Remarques</i> :	112
LES REGISTRES À DÉCALAGE	112
<i>Principe de fonctionnement</i>	112
<i>Registre serial IN serial OUT</i>	113
<i>Registre serial IN parallel OUT</i>	114
<i>Registre parallel IN serial OUT</i>	116
<i>Registre parallel IN parallel OUT</i>	118
LES MACHINES D'ETATS SYNCHRONES	120
MACHINE D'ETAT DE MEALY A SORTIES SYNCHRONISEES	121
STRUCTURE INTERNE D'UNE MACHINE D'ETATS	122
LE GRAPHE DES ETATS (DIAGRAMME DE TRANSITION).....	123
<i>Table de vérité des sorties en fonction de l'état</i>	124
DU GRAPHE DES ETATS A LA DESCRIPTION VHDL.....	125
<i>Codage des états</i>	125
<i>Machines de Moore</i>	125
<i>Machines de Mealy</i>	126
<i>Machines de Mealy à sorties (re)synchronisées</i>	127
LES COMPTEURS	129

<i>Modulo</i>	129
SYNTHÈSE MANUELLE DES COMpteURS SYNCHRONES.....	129
ARITHMETIQUE EN VHDL.....	131
COMpteURS PRERéGLABLE	131
DESCRIPTION D'UN COMpteUR SYNCHRONE EN VHDL.....	132
DIVISEUR DE FRéQUENCE.....	133
<i>Description VHDL</i>	133
CLOCKING	134
SYSTèME NUMéRIQUE SYNCHRONE	134
<i>Contraintes de timing sur les bascules D synchrones</i>	134
<i>Setup and Hold Times.....</i>	134
SKEW	135
<i>Types de clock Skew</i>	135
<i>Facteurs principaux causant le Skew.....</i>	135
JITTER.....	136
<i>Sources du Jitter</i>	136
FRéQUENCE MAXIMUM DU SIGNAL D'HORLOGE	136
<i>Délais routage interne</i>	136
<i>Clock skew et jitter</i>	137
<i>Setup et hold time des FF (bascules D).....</i>	137
<i>Délais de la logique combinatoire</i>	137
<i>Analyse de la fréquence maximum d'un système numérique donné</i>	137
MÉMOIRES	138
ACCES SEQUENTIEL OU ALEATOIRE	138
<i>ACCÈS ALéATOIRE</i>	138
ROM (READ-ONLY MEMORY).....	138
PROM (PROGRAMMABLE ROM).....	138
<i>Principe</i>	138
<i>Fonctionnement</i>	139
<i>Réalisation pratique</i>	140
EPROM (ERASABLE PROM)	140
<i>Principe</i>	140
<i>Exemple</i>	141
<i>Timing d'une EPROM</i>	142
<i>EPROM à UV ou OTP</i>	142
RAM (RANDOM ACCESS MEMORY).....	143
<i>Mémoire SRAM</i>	143
<i>Notion de buffer 3 états (tri-state)</i>	147
<i>Les mémoires du commerce</i>	147
<i>Cellule mémoire réelle</i>	148
<i>Mémoire DRAM</i>	148
<i>Mémoire SDRAM</i>	151
MEMOIRE NVRAM	152
MEMOIRES EEPROM ET FLASH	153
<i>Mémoires EEPROM</i>	153
<i>Les mémoires Flash</i>	154
LES DIFFERENTS TYPES DE MEMOIRES DANS UN PC	156
<i>Critères de classification</i>	156
<i>Différents types de mémoires</i>	157
<i>Fonction de la mémoire cache</i>	158
LES MEMOIRES SERIE	159
<i>Phase de lecture</i>	159
<i>Phase d'écriture</i>	160
DESCRIPTION D'UNE MEMOIRE SRAM EN VHDL	161
LE DECODAGE D'ADRESSE	162

UN SYSTEME DEDIE	162
LES BUS D'ADRESSES ET DE DONNEES.....	162
PARTAGE DE LA PLAGE MEMOIRE (MEMORY MAP)	163
REALISATION D'UN CIRCUIT DE DECODAGE	164
<i>Décodage avec un 74HC138.....</i>	164
<i>Décodage d'adresse dans un circuit logique programmable.....</i>	165
UN SYSTEME ON CHIP (SOC)	166
MEMOIRES SUR BUS MICROCONTROLEURS 16BITS OU 32BIT	167
<i>Microcontrôleur de 16 bits avec une mémoire de 16 bits.....</i>	167
<i>Microcontrôleur de 32 bits avec deux mémoires de 16 bits.....</i>	168
ASIC ET COMPOSANTS LOGIQUES PROGRAMMABLES : PAL, PLD, CPLD, FPGA.....	169
CODAGE D'UNE FONCTION LOGIQUE.....	170
<i>Sommes de produits, produits de somme et matrice PLA (Programmable Logic Array).....</i>	170
<i>Mémoires (Look Up Table)</i>	172
<i>Multiplexeur</i>	173
TECHNOLOGIE D'INTERCONNEXIONS	173
<i>Connexions programmable une seule fois (OTP : One Time Programming)</i>	173
<i>Cellules reprogrammables</i>	173
ARCHITECTURES UTILISEES	175
<i>PLD (Programmable Logic Device).....</i>	175
CPLD (COMPLEX PROGRAMMABLE LOGIC DEVICE)	176
FPGA (FIELD PROGRAMMABLE GATE ARRAY)	179
<i>Structure FPGA famille XC4000 de Xilinx</i>	180
<i>Structure FPGA famille Spartan 3 de Xilinx</i>	183
LES OUTILS DE DEVELOPPEMENT DES CPLD ET FPGA	187
TECHNIQUES DE PROGRAMMATION ET DE TEST	188
<i>Le port JTAG.....</i>	188
PLDS, CPLDS, FPGAS : QUEL CIRCUIT CHOISIR?	195
<i>Critères de performances</i>	195
ASIC (APPLICATION SPECIFIC INTEGRATED CIRCUIT).....	196
<i>Les prédiffusés (gate arrays).....</i>	196
<i>Les précaractérisés (standard cell).....</i>	196
<i>Les "fulls customs"</i>	196
COMPARAISON ET EVOLUTION	196
BIBLIOGRAPHIE.....	198
MÉDIAGRAPHIE	198
LEXIQUE	199

Introduction

L'utilisation des systèmes logiques est en pleine expansion. Pour s'en convaincre, il n'est qu'à regarder autour de nous l'explosion de la micro-informatique, qui s'est même implantée dans les ménages. De plus, un nombre de plus en plus grand de machines (télévision, voiture, machine à laver, etc.) utilisent de l'électronique numérique dans ce que l'on appelle des systèmes embarqués.

Nous trouvions, jusqu'à l'apparition du microprocesseur, deux grands secteurs dans le domaine de l'électronique numérique. Cette division a subsisté chez les fabricants d'ordinateurs où nous trouvons encore:

- le département matériel (hardware)
- le département logiciel ou programmes (software)

L'apparition du microprocesseur a eu pour effet de diminuer l'importance du matériel et de provoquer un déplacement des moyens de traitement des circuits aux programmes. Ce qui fait que nous nous trouvons de plus en plus en face de programmes qui cernent la machine au plus près. Ce qui oblige les programmeurs à connaître de mieux en mieux le matériel pour mieux "coller" à l'application avec le programme.

Après avoir réduit le marché de la logique câblée, le microprocesseur est parti à la conquête de l'électronique basse fréquence. Il a fait son entrée que dans un nombre important de secteurs (jeux, télécommunications, automatique, ...).

L'augmentation des possibilités d'intégration (nombre de composants par mm²) conduit à une nouvelle évolution. La logique programmée des certaines applications se trouve remplacée par de la logique câblée dans des circuits logiques programmables. Cette évolution permet d'envisager une augmentation de la vitesse de traitement des fonctions.

Jusqu'à présent, l'apprentissage de la logique câblée se faisait à travers la découverte des fonctions logiques élémentaires contenues dans les circuits intégrés des familles 74xxx, dont on peut voir quelques types dans la Figure 1. Les expérimentations se limitaient aux fonctions proposées par les fabricants de ces circuits, et la conception de fonctions logiques regroupant plusieurs de ces circuits nécessitait un câblage conséquent, et la réalisation d'un circuit imprimé de grande surface.

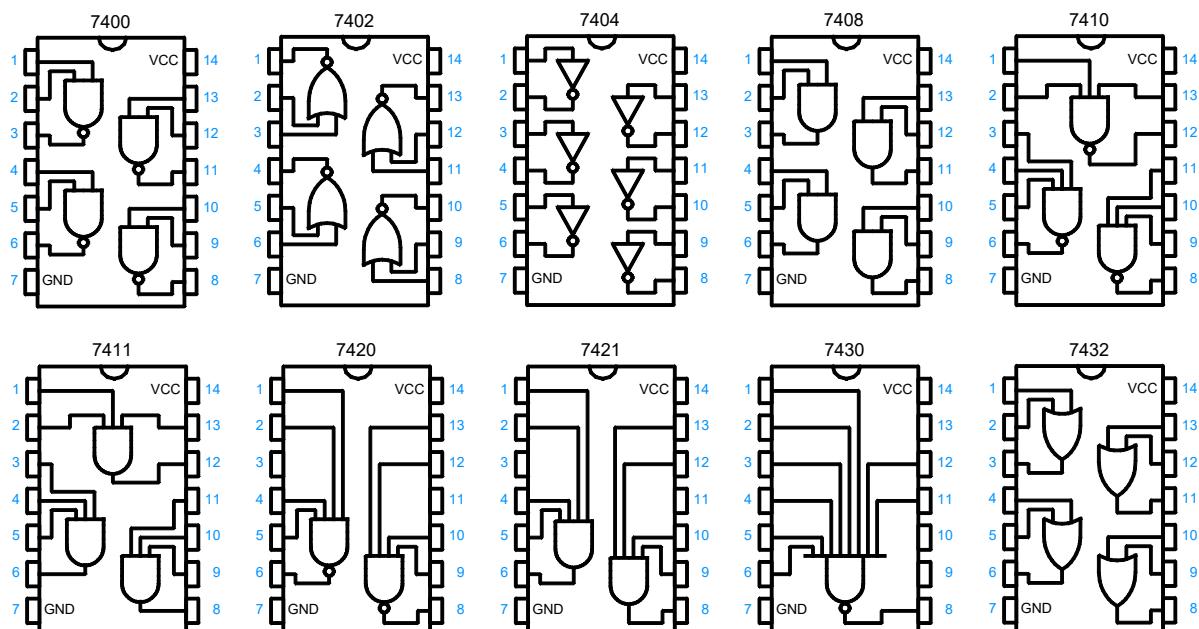


Figure 1 Circuits logiques standards de la famille 74xx

L'apparition des circuits logiques programmables de type PLD (Programmable Logic Device), CPLD (Complexe PLD, Figure 2a) ou FPGA (Field Programmable Gate Array, Figure 2b) a permis de s'affranchir de cette limitation. En effet, l'utilisateur peut créer, dans ces circuits, toutes les fonctions logiques qu'il souhaite avec comme seules limitations, la place disponible dans le circuit choisi et / ou la vitesse de fonctionnement de celui-ci.

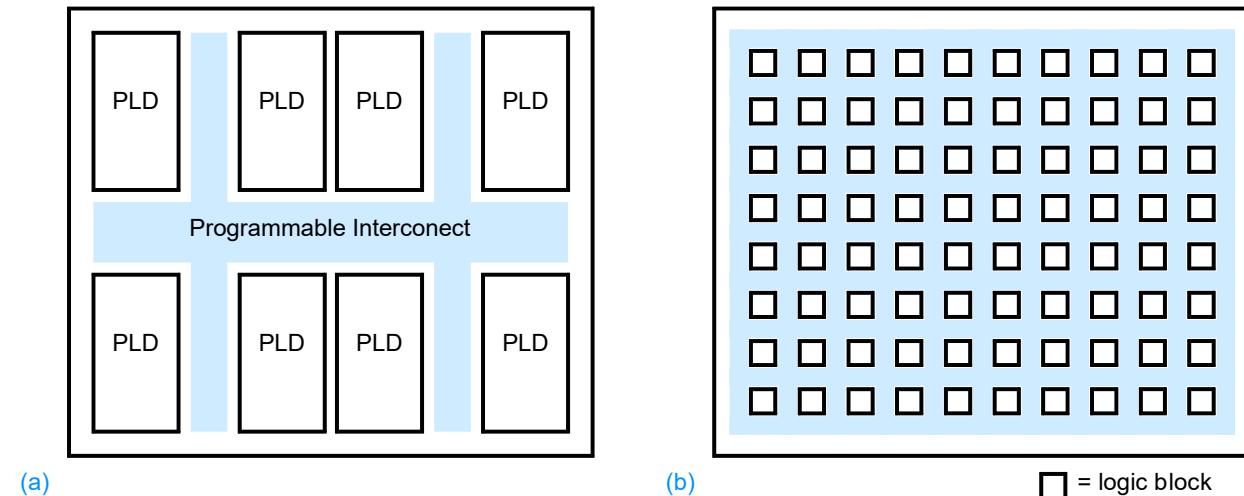


Figure 2 Circuits logiques programmables (a) CPLD ; (b) FPGA

Les outils de développement mis à la disposition des utilisateurs par les fabricants de ces circuits doivent donc permettre de passer de la description du comportement d'une fonction logique à son câblage dans le circuit et cela de la manière la plus simple possible. La plupart du temps, la description du comportement des fonctions logiques est faite par l'utilisation de langage dit de "description de matériel". Parmi ceux-ci, on peut citer:

La première génération de ces langages permettait des descriptions au niveau logique. Il a existé deux langages :

- Le CUPL utilisé dans les années 1980 à 1995.
- Le langage ABEL (Advanced Boolean Equation Language) qui a été créé par la société DATA I/O et utilisé ou imité par quasiment tous les concepteurs d'outils de développement pour ce type de circuit (XABEL pour XILINX, AHDL pour ALTERA, PLD pour ORCAD, XPLA pour PHILIPS, etc....). Ce langage n'est quasiment plus utilisé actuellement.

L'augmentation de la complexité des circuits programmable a nécessité de disposer de langage permettant des descriptions de plus haut niveau (comportementale). Deux langages sont apparus au début des années 1990 pour la conception de circuits ASIC (circuit intégré spécialisé). Ils se sont imposés dès le milieu 1995 pour les circuits logiques programmables. Il s'agit :

- Le langage VHDL (Very High Speed Integrated Circuit, Hardware Description Language) qui a été créé pour le développement de circuits intégrés logiques complexes. Il doit son succès, essentiellement, à sa standardisation sous la référence IEEE1076, qui a permis d'en faire un langage unique pour la description, la modélisation, la simulation, la synthèse et la documentation.
- Le langage VERILOG qui est proche du langage VHDL et qui est aussi très utilisé, mais sa syntaxe diffère quelque peu du VHDL, nous ne l'étudierons pas dans ce cours.

Le but de ce cours, est dans un premier temps de récapituler les systèmes logiques combinatoires et séquentiels, tout apprenant à utiliser le langage VHDL pour décrire ces fonctions de base. Dans un deuxième temps, l'accent sera mis sur la méthodologie de développement de systèmes numérique avec le langage VHDL.

Systèmes de numérotation et codes

Introduction

Le système de numération binaire est le plus important de ceux utilisés dans les circuits numériques, bien qu'il ne faille pas pour autant négliger l'importance d'autres systèmes. Le système décimal revêt de l'importance en raison de son acceptation universelle pour représenter les grandeurs du monde courant. De ce fait, il faudra parfois que des valeurs décimales soient converties en valeurs binaires avant d'être introduites dans le circuit numérique. Par exemple, lorsque vous composez un nombre décimal sur votre calculatrice (ou sur le clavier de votre ordinateur), les circuits internes convertissent ce nombre décimal en une valeur binaire.

De même, il y aura des situations où des valeurs binaires données par un circuit numérique devront être converties en valeurs décimales pour qu'on puisse les lire. Par exemple, votre calculatrice (ou votre ordinateur) calcule la réponse à un problème au moyen du système binaire puis convertit ces réponses en des valeurs décimales avant de les afficher.

Nous connaissons les systèmes binaire et décimal, étudions maintenant deux autres systèmes de numération très répandus dans les circuits numériques. Il s'agit des systèmes de numération octale (base de 8) et hexadécimal (base de 16) qui servent tous les deux au même but, soit celui de constituer un outil efficace pour représenter de gros nombres binaires. Comme nous le verrons, ces systèmes de numération ont l'avantage d'exprimer les nombres de façon que leur conversion en binaire, et vice versa, soit très facile.

Dans un système numérique, il peut arriver que trois ou quatre de ces systèmes de numération cohabitent, d'où l'importance de pouvoir convertir un système dans un autre. Le présent chapitre se propose de vous montrer comment effectuer de telles conversions. Certaines de ces conversions ne seront pas appliquées immédiatement dans l'étude des circuits numériques, mais vous constaterez au moment de l'étude des microprocesseurs que c'est une connaissance dont vous avez besoin.

Ce chapitre se veut également une introduction à certains des codes binaires utilisés pour représenter divers genres d'information. Ces codes agencent les 0 et le 1 de manière différente du système binaire.

Représentation des nombres et numération de position

Le nombre de symboles utilisés caractérise le numéro de la base.

- Ex.: - en base 10, nous avons les 10 symboles (0, 1,..,9)
 - en base 2, nous avons les 2 symboles (0, 1)
 - en base 3, nous avons les 3 symboles (0, 1, 2)
 - en base 16, nous utiliserons les 10 chiffres plus les lettres de A à F
 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F).

N.B. : Il faut remarquer que le choix de symboles chiffres facilite grandement les choses, cette facilité découlant de notre grande habitude du système décimal (10 symboles).

La valeur d'un chiffre dépend de sa position dans le nombre. Nous parlons de numération de position, soit :

Un nombre dans une base "b" entière positive s'écrit :

$$(1) N_b = a_n a_{n-1} \dots a_1 a_0, a_{-1} \dots a_{-m}$$

ce qui correspond aux opérations :

$$(2) N_B = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \dots + a_1 \cdot b^1 + a_0 \cdot b^0 + a_{-1} \cdot b^{-1} + \dots + a_{-m} \cdot b^{-m}$$

L'indice de N indique la base dans laquelle le nombre est calculé.

Conversion binaire - décimal

Le système de numération binaire en est un à poids positionnels dans lequel chaque bit est affecté d'un certain poids qui dépend de son rang par rapport au bit de poids le plus faible. Ainsi tout nombre binaire peut être transformé en son équivalent décimal simplement en additionnant les poids des diverses positions où se trouve une valeur 1. Voici une illustration:

1 1 0 1 1 (binaire)

$$1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 8 + 2 + 1 = 27_{10} \text{ (décimal)}$$

Voyons un autre exemple pour un nombre ayant un plus grand nombre de bits.

1 0 1 1 0 1 0 1_2 =

$$2^7 + 0 + 2^5 + 2^4 + 0 + 2^2 + 0 + 2^0 = 181_{10}$$

Vous remarquez que la méthode consiste à trouver les poids (les puissances de 2) pour chaque position du nombre où il y a un 1, puis à additionner le tout. Remarquez que le bit de poids le plus fort a un poids de 2^7 même s'il s'agit du huitième bit; il en est ainsi parce que le bit de poids le plus faible est le premier bit et que son poids est toujours 2^0 .

Conversion décimal - binaire

Il existe deux façons de convertir un nombre décimal entier en son équivalent binaire. Une méthode qui convient bien aux petits nombres est l'inverse de la démarche suivie précédemment. Le nombre décimal est simplement exprimé comme une somme de puissances de 2, puis on inscrit des 1 et des 0 vis-à-vis des positions binaires appropriées. Voici un exemple:

$$45_{10} = 32 + 8 + 4 + 1 = 2^5 + 0 + 2^3 + 2^2 + 0 + 2^0 = 101101_2$$

Notons qu'il y a un 0 vis-à-vis des positions 2^1 et 2^4 , puisque ces positions ne sont pas utilisées pour trouver la somme en question.

Répétition de divisions

L'autre méthode convient mieux aux grands nombres décimaux; il s'agit de répéter la division par 2. Cette méthode de conversion, illustrée ci-après pour le nombre 25_{10} , recourt à la répétition de la division par 2 du nombre décimal à convertir et au report des restes pour chaque division jusqu'à ce que le quotient soit 0. Notez que le nombre binaire résultant s'obtient en écrivant le premier reste à la position du bit de poids le plus faible (LSB) et le dernier reste à la position du bit de poids le plus fort (MSB).

$$\begin{array}{rcl} 25/2 & = 12 & \text{reste } 1 \text{ Poids faible (LSB)} \\ 12/2 & = 6 & \text{reste } 0 \\ 6/2 & = 3 & \text{reste } 0 \\ 3/2 & = 1 & \text{reste } 1 \\ 1/2 & = 0 & \text{reste } 1 \text{ Poids fort (MSB)} \end{array}$$

$$25_{10} = 11001_2$$

Système de numération octal

Le système de numération octal a comme base huit, ce qui signifie qu'il comprend huit symboles possibles, soit 0, 1, 2, 3, 4, 5, 6 et 7. Ainsi, chaque chiffre dans un nombre octal a une valeur comprise entre 0 et 7. Voici les poids de chacune des positions d'un nombre octal.

....	8^3	8^2	8^1	8^0	.	8^{-1}	8^{-2}	8^{-3}
------	-------	-------	-------	-------	---	----------	----------	----------	------

Conversion octal-décimal

On convertit un nombre octal en son équivalent décimal en multipliant chaque chiffre octal par son poids positionnel. Voici un exemple:

$$\begin{aligned} 372_8 &= 3 \cdot (8^2) + 7 \cdot (8^1) + 2 \cdot (8^0) \\ &= 3 \cdot 64 + 7 \cdot 8 + 2 \cdot 1 \\ &= 250_{10} \end{aligned}$$

Conversion décimal-octal

Il est possible de convertir un nombre décimal entier en son équivalent octal en employant la méthode de la répétition de divisions, la même qu'on a utilisée pour la conversion décimal-binaire, mais cette fois-ci en divisant par 8 plutôt que par 2. Voici un exemple:

$$\begin{array}{rcl} 266/8 & = 33 & \text{reste } 2 \\ 33/8 & = 4 & \text{reste } 1 \\ 4/8 & = 0 & \text{reste } 4 \end{array}$$

$$266_{10} = 412_8$$

Notez que le premier reste devient le chiffre de poids le plus faible du nombre octal et que le dernier reste devient le chiffre de poids le plus fort.

Si on utilise une calculatrice pour faire les divisions, on aura comme résultat un nombre avec une partie fractionnaire plutôt qu'un reste. On calcule toutefois le reste en multipliant la fraction décimale par 8. Par exemple, avec la calculatrice, la réponse de la division 266/8, est 33,25. En multipliant la partie décimale par 8, on trouve un reste de $0,25 \times 8 = 2$. De même, $33 / 8$ donne 4,125, d'où un reste de $0,125 \times 8 = 1$.

Conversion octal-binaire

Le principal avantage du système de numération octal réside dans la facilité avec laquelle il est possible de passer d'un nombre octal à un nombre binaire. Cette conversion s'effectue en transformant chaque chiffre du nombre octal en son équivalent binaire de trois chiffres. Voyez dans le tableau ci-dessous les huit symboles octaux exprimés en binaire.

Chiffre octal	0	1	2	3	4	5	6	7
Équivalent binaire	000	001	010	011	100	101	110	111

Au moyen de ce tableau, tout nombre octal est converti en binaire par la transformation de chacun des chiffres. Par exemple, la conversion de 472_8 va comme suit:

$$\begin{array}{ccc} 4 & 7 & 2 \\ 100 & 111 & 010 \end{array}$$

Donc le nombre octal 472_8 est équivalent au nombre binaire 100111010.

Conversion binaire-octal

La conversion d'un nombre binaire en un nombre octal est tout simplement l'inverse de la marche à suivre précédente. Il suffit de faire avec le nombre binaire des groupes de trois bits en partant du chiffre de poids le plus faible, puis de convertir ces triplets en leur équivalent octal (voir tableau 2-1). À titre d'illustration, convertissons 100111010_2 en octal.

100	111	010
4	7	2_8

Parfois, il arrivera que le nombre binaire ne forme pas un nombre juste de groupes de trois. Dans ce cas, on pourra ajouter un ou deux zéros à gauche du bit de poids le plus fort pour former le dernier triplet (si on lit de droite à gauche). Voici une illustration de ceci avec le nombre binaire 11010110 .

011	010	110
3	2	6_8

Notez l'ajout d'un zéro à gauche du bit de poids le plus fort pour obtenir un nombre juste de triplets.

Système de numération hexadécimal

Le système hexadécimal a comme base 16, ce qui implique 16 symboles de chiffres possibles, qui, dans ce cas, sont les dix chiffres 0 à 9 plus les lettres majuscules A, B, C, D, E et F. Le Tableau 1 expose les rapports entre les systèmes hexadécimal, décimal et binaire. Remarquez que chaque chiffre hexadécimal a comme équivalent binaire un groupe de quatre bits. Il ne faut surtout pas oublier que les chiffres hexadécimaux A à F correspondent aux valeurs décimales 10 à 15.

Hexadécimal	Décimal	Binaire
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Tableau 1 Rapport entre hexadécimal, décimal et binaire

La représentation hexadécimale est principalement utilisée pour représenter un nombre binaire sous forme plus compact. Un nombre en hexadécimal comprend 4 fois moins de chiffres !

Conversion hexadécimal-décimal

Un nombre hexadécimal peut être converti en son équivalent décimal en exploitant le fait qu'à chaque position d'un chiffre hexadécimal est attribué un poids; dans ce cas-ci le nombre 16 élevé à une certaine puissance. Le chiffre de poids le plus faible a un poids de $16^0 = 1$, le chiffre immédiatement à gauche a un poids de $16^1 = 16$, l'autre chiffre immédiatement à gauche, un poids de $16^2 = 256$, et ainsi de suite. Voici un exemple sur la façon dont fonctionne ce processus de conversion.

$$\begin{aligned} 356_{16} &= 3 \cdot 16^2 + 5 \cdot 16^1 + 6 \cdot 16^0 \\ &= 768 + 80 + 6 \\ &= 854_{10} \end{aligned}$$

$$\begin{aligned} 2AF_{16} &= 2 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 \\ &= 512 + 160 + 15 \\ &= 687_{10} \end{aligned}$$

Conversion décimal-hexadécimal

Vous vous rappelez peut-être que pour la conversion décimal-binaire nous avons eu recours à la répétition de divisions par 2, que pour la conversion décimal-octal, à la répétition de division par 8. Donc, pour convertir un nombre décimal en un nombre hexadécimal, il faut procéder de la même façon, mais cette fois en divisant par 16. Les exemples qui suivent illustrent cette technique. Remarquez comment les restes des divisions deviennent les chiffres du nombre hexadécimal; de plus, voyez, comment les restes supérieurs à 9 sont exprimés au moyen des lettres A à F. Exemple, conversion de 423_{10} en hexadécimal :

$$\begin{aligned} 423/16 &= 26 \text{ reste } 7 \\ 26/16 &= 1 \text{ reste } 10 \\ 1/16 &= 0 \text{ reste } 1 \\ 423_{10} &= 1A7_{16} \end{aligned}$$

Conversion hexadécimal-binaire

Comme le système de numération octal, le système de numération hexadécimal se veut une façon abrégée de représenter les nombres binaires. La conversion d'un nombre hexadécimal en un nombre binaire ne pose vraiment pas de difficulté, puisque chaque chiffre hexadécimal est remplacé par son équivalent binaire de 4 bits (tableau 2-2). Voici un exemple avec $9F216$.

$$\begin{aligned} 9F2_{16} &= 9 \quad F \quad 2 \\ &\quad 1001 \quad 1111 \quad 0010 \\ &= 100111110010_2 \end{aligned}$$

Conversion binaire-hexadécimal

Cette conversion est tout simplement l'inverse de la précédente. Le nombre binaire est divisé en groupes de quatre bits, puis on substitue à chaque groupe son chiffre hexadécimal équivalent. Au besoin, on ajoute des zéros à gauche pour obtenir un dernier groupe de 4 bits.

$$\begin{aligned} 1110100110_2 &= 0011 \quad 1010 \quad 0110 \\ &\quad 3 \quad A \quad 6 \\ &= 3A6_{16} \end{aligned}$$

Pour passer d'un nombre hexadécimal à son équivalent binaire, il faut connaître la suite des nombres binaires de quatre bits (0000 à 1111) ainsi que le nombre correspondant en hexadécimal. Dès que cette correspondance devient un réflexe automatique, les conversions se font rapidement sans calculs. C'est ce qui explique pourquoi le système hexadécimal est si pratique pour représenter de grands nombres binaires.

Comptage hexadécimal

Lorsque l'on compte selon le système de numération hexadécimal, la valeur dans une position du nombre croît par pas de 1 depuis 0 jusqu'à F. Quand le chiffre dans une position est F, le chiffre suivant dans cette position est 0 et le chiffre immédiatement à gauche est augmenté de 1. C'est ce qu'on voit dans les suites de nombres hexadécimaux suivantes:

- a) 38, 39, 3A, 3B, 3C, 3D, 3E, 3F, 40, 41, 42
- b) 6F8, 6F9, 6FA, 6FB, 6FC, 6FD, 6FE, 6FF, 700,

Notez que le chiffre qui suit 9 dans une position est A.

Utilité du système hexadécimal

La facilité avec laquelle se font les conversions entre les systèmes binaire et hexadécimal explique pourquoi le système hexadécimal est devenu une façon abrégée d'exprimer de grands nombres binaires. Dans un ordinateur, il n'est pas rare de retrouver des nombres binaires ayant jusqu'à 64 bits de longueur. Ces nombres binaires, comme nous le verrons, ne sont pas toujours des valeurs numériques, mais peuvent correspondre à un certain code représentant des renseignements non numériques. Dans un ordinateur, un nombre binaire peut être: 1) un vrai nombre; 2) un nombre correspondant à un emplacement (adresse) en mémoire; 3) un code d'instruction; 4) un code correspondant à un caractère alphabétique ou non numérique; ou 5) un groupe de bits indiquant la situation dans laquelle se trouvent des dispositifs internes et externes de l'ordinateur.

Quand on doit travailler avec beaucoup de nombres binaires très longs, il est plus commode et plus rapide d'écrire ces nombres en hexadécimal plutôt qu'en binaire. Toutefois, ne perdez pas de vue que les circuits et les systèmes numériques fonctionnent exclusivement en binaire et que c'est par pur souci de commodité pour les opérateurs qu'on emploie la notation hexadécimale.

Code BCD, soit Binary Coded Decimal

L'action de faire correspondre à des nombres, des lettres ou des mots un groupe spécial de symboles s'appelle codage et le groupe de symboles un code. Un des codes que vous connaissez peut-être le mieux est le code Morse dans lequel on utilise une série de points et de traits pour représenter les lettres de l'alphabet.

Nous avons vu que tout nombre décimal pouvait être converti en son équivalent binaire. Il est possible de considérer le groupe de 0 et de 1 du nombre binaire comme un code qui représente le nombre décimal. Quand on fait correspondre à un nombre décimal son équivalent binaire, on dit qu'on fait un codage binaire pur.

Les circuits numériques fonctionnent avec des nombres binaires exprimés sous une forme ou sous une autre durant leurs opérations internes, malgré que le monde extérieur soit un monde décimal. Cela implique qu'il faut effectuer fréquemment des conversions entre les systèmes binaire et décimal. Nous savons que pour les grands nombres, les conversions de ce genre peuvent être longues et laborieuses. C'est la raison pour laquelle on utilise dans certaines situations un codage des nombres décimaux qui combine certaines caractéristiques du système binaire et du système décimal.

Le BCD s'appelle en français Code Décimal codé Binaire (CDB). Si on représente chaque chiffre d'un nombre décimal par son équivalent binaire, on obtient le code dit décimal codé binaire (abrégé dans le reste du texte par BCD). Comme le plus élevé des chiffres décimaux est 9, il faut donc 4 bits pour coder les chiffres.

Illustrons le code BCD en prenant le nombre décimal 874 et en changeant chaque chiffre pour son équivalent binaire; cela donne:

8	7	4	décimal
1000	0111	0100	BCD

De nouveau, on voit que chaque chiffre a été converti en son équivalent binaire pur. Notez qu'on fait toujours correspondre 4 bits à chaque chiffre.

Le code BCD établit donc une correspondance entre chaque chiffre d'un nombre décimal et un nombre binaire de 4 bits. Évidemment, seuls les groupes binaires 0000 à 1001 sont utilisés. Le code BCD ne fait pas

usage des groupes 1010, 1011, 1100, 1101, 1110 et 1111. Autrement dit, seuls dix des 16 combinaisons des 4 bits sont utilisés. Si l'une des combinaisons « inadmissibles » apparaît dans une machine utilisant le code BCD, c'est généralement le signe qu'une erreur s'est produite.

Comparaison entre code BCD et nombre binaire

Il importe de bien réaliser que le code BCD n'est pas un autre système de numération comme les systèmes octal, décimal ou hexadécimal. En fait, ce code est le système décimal dont on a converti les chiffres en leur équivalent binaire. En outre, il faut bien comprendre qu'un nombre BCD n'est pas un nombre binaire pur. Quand on code selon le système binaire pur, on prend le nombre décimal dans son intégralité et on le convertit en binaire, sans le fractionner; par ailleurs, quand on code en BCD, c'est chaque chiffre individuel qui est remplacé par son équivalent binaire. À titre d'exemple, prenons le nombre 137 et trouvons son nombre binaire pur puis son équivalent BCD:

$$137_{10} = 10001001_2 \quad (\text{Binaire})$$

$$137_{10} = 0001\ 0011\ 0111 \quad (\text{BCD})$$

Le code BCD nécessite 12 bits pour représenter 137 tandis que le nombre binaire pur n'a besoin que de 8 bits. Il faut plus de bits en BCD qu'en binaire pur pour représenter les nombres décimaux de plus d'un chiffre. Comme vous le savez, il en est ainsi parce que le code BCD n'utilise pas toutes les combinaisons possibles de groupes de 4 bits; c'est donc un code peu efficace.

Le principal avantage du code BCD provient de la facilité relative avec laquelle on passe de ce code à un nombre décimal, et vice versa. Il ne faut retenir que les groupes de 4 bits des chiffres 0 à 9. C'est un avantage non négligeable du point de vue du matériel, puisque dans un système numérique ce sont des circuits logiques qui ont la charge d'effectuer ces conversions.

On peut voir dans le Tableau 2 ci-dessous, les principaux codes utilisés. Il faut toutefois mentionner le code GRAY ou binaire réfléchi. Ce code présente l'avantage qu'il n'y a qu'un seul bit qui change à la fois. Il offre dès lors de multiples utilisations.

Décimal	binaire	octal	hexadécimal	Gray ou BR	Excédent 3	AIKEN
00	0000	00	0	0000	0011	0000
01	0001	01	1	0001	0100	0001
02	0010	02	2	0011	0101	0010
03	0011	03	3	0010	0110	0011
04	0100	04	4	0110	0111	0100
05	0101	05	5	0111	1000	1011
06	0110	06	6	0101	1001	1100
07	0111	07	7	0100	1010	1101
08	1000	10	8	1100	1011	1110
09	1001	11	9	1101	1100	1111
10	1010	12	A	1111	sur deux décades	
11	1011	13	B	1110	sur deux décades	
12	1100	14	C	1010	sur deux décades	
13	1101	15	D	1011	sur deux décades	
14	1110	16	E	1001	sur deux décades	
15	1111	17	F	1000	sur deux décades	

Tableau 2 Les principaux codes utilisés

Les codes alphanumériques

Un ordinateur ne serait pas d'une bien grande utilité s'il était incapable de traiter l'information non numérique. On veut dire par-là qu'un ordinateur doit reconnaître des codes qui correspondent à des nombres, des lettres, des signes de ponctuation et des caractères spéciaux. Les codes de ce genre sont dit alphanumériques. Un ensemble de caractères complet doit renfermer les 26 lettres minuscules, les 26 lettres majuscules, les dix chiffres, les 7 signes de ponctuation et entre 20 à 40 caractères spéciaux comme +, /, #, %. On peut conclure qu'un code alphanumérique reproduit tous les caractères et les diverses fonctions que l'on retrouve sur un clavier standard de machine à écrire ou d'ordinateur.

Code ASCII

Le code alphanumérique le plus répandu est le code ASCII (American Standard Code for Information Interchange); on le retrouve dans la majorité des micro-ordinateurs et des Miniordinateurs et dans beaucoup de gros ordinateurs. Le code ASCII (prononcé « aski ») est un code à 7 éléments, on peut donc représenter grâce à lui $2^7 = 128$ groupes codés. C'est amplement suffisant pour reproduire toutes les lettres courantes d'un clavier et les fonctions de contrôle comme (RETOUR) et (INTERLIGNE). Le Tableau 3 contient une liste partielle du code ASCII. Dans ce dernier, en plus du groupe binaire de chaque caractère, on a donné l'équivalent hexadécimal.

Caractère	ASCII à 7 éléments	Hexadécimal	Caractère	ASCII à 7 éléments	Hexadécimal
A	100 0001	41	0	011 0000	30
B	100 0010	42	1	011 0001	31
C	100 0011	43	2	011 0010	32
D	100 0100	44	3	011 0011	33
E	100 0101	45	4	011 0100	34
F	100 0110	46	5	011 0101	35
G	100 0111	47	6	011 0110	36
H	100 1000	48	7	011 0111	37
I	100 1001	49	8	011 1000	38
J	100 1010	4A	9	011 1001	39
K	100 1011	4B			
L	100 1100	4C	SPACE	010 0000	20
M	100 1101	4D	.	010 1110	2E
N	100 1110	4E	(010 1000	28
O	100 1111	4F	+	010 1011	2B
P	101 0000	50	\$	010 0100	24
Q	101 0001	51	*	010 1010	2A
R	101 0010	52)	010 1001	29
S	101 0011	53	-	010 1101	2D
T	101 0100	54	/	010 1111	2F
U	101 0101	55	,	010 1100	2C
V	101 0110	56	=	011 1101	3D
W	101 0111	57	RETURN	000 1101	0D
X	101 1000	58	LINE FEED	000 1010	0A
Y	101 1001	59			
Z	101 1010	5A			

Tableau 3 Liste partielle du code ASCII

Arithmétique Binaire

Introduction

Les diverses opérations arithmétiques qui interviennent dans les ordinateurs et les calculatrices portent sur des nombres exprimés en notation binaire. En tant que telle, l'arithmétique numérique peut être un sujet très complexe, particulièrement si on veut comprendre toutes les méthodes de calcul et la théorie sur laquelle elle s'appuie. Heureusement, il n'est pas nécessaire d'enseigner aux ingénieurs la théorie complète de l'arithmétique numérique, tout au moins pas avant qu'ils soient devenus des programmeurs expérimentés. Dans ce chapitre, nous allons concentrer nos efforts sur les principes de base qui nous permettent de comprendre comment les machines numériques (c'est-à-dire les ordinateurs) réalisent les opérations arithmétiques de base.

D'abord nous verrons comment effectuer manuellement les opérations arithmétiques en binaire, par la suite nous étudierons les circuits logiques réels qui matérialisent quelques-unes de ces opérations dans un système numérique.

Addition binaire

L'addition de deux nombres binaires est parfaitement analogue à l'addition de deux nombres décimaux. En fait, l'addition binaire est plus simple puisqu'il y a moins de cas à apprendre. Revoyons d'abord l'addition décimale:

$$\begin{array}{r} 376 \\ + 461 \\ \hline 837 \end{array}$$

On commence par additionner les chiffres du rang de poids faible, ce qui donne 7. Les chiffres du deuxième rang sont ensuite additionnés, ce qui donne une somme de 13, soit 3 plus un report de 1 sur le troisième rang. Pour le troisième rang, la somme des deux chiffres plus le report de 1 donne 8.

Les mêmes règles s'appliquent à l'addition binaire. Cependant, il n'y a que quatre cas, qui peuvent survenir lorsqu'on additionne deux chiffres binaires et cela quel que soit le rang. Ces quatre cas sont :

$$\begin{array}{lll} 0+0 & =0 \\ 1+0 & =1 \\ 1+1 & =10 = 0 + \text{report de 1 sur le rang de gauche} \\ 1+1+1 & =11 = 1 + \text{report de 1 sur le rang de gauche} \end{array}$$

Le dernier cas ne se produit que lorsque, pour un certain rang, on additionne deux 1 plus un report de 1 provenant du rang de droite. Voici quelques exemples d'additions de deux nombres binaires :

$$\begin{array}{rcl} 011 \quad (3) & & 1001 & & 11,011 \\ + 110 \quad (6) & & + 1111 & & + 10,110 \\ \hline = 1001 \quad (9) & & = 11000 & & = 110,001 \end{array}$$

Il n'est pas nécessaire d'étudier des additions ayant plus de deux nombres binaires, parce que dans tous les systèmes numériques les circuits qui additionnent ne traitent pas plus de deux nombres à la fois. Lorsque nous avons plus de deux nombres à additionner, on trouve la somme des deux premiers puis on additionne cette somme au troisième nombre, et ainsi de suite. Ce n'est pas véritablement un inconvénient, puisque les machines numériques modernes peuvent généralement réaliser une opération d'addition en quelques nanosecondes.

L'addition est l'opération arithmétique la plus importante dans les systèmes numériques. Les opérations de soustraction, de multiplication et de division effectuées par les ordinateurs ne sont essentiellement que des variantes de l'opération d'addition.

Écriture des nombres signés

Dans les ordinateurs, on utilise un ensemble d'éléments de mémoire à deux états (habituellement des bascules) pour représenter des nombres binaires. À chaque élément est associé un bit. Par exemple, un registre à bascules de 7 bits peut mémoriser les nombres binaires allant de 0000000 à 1111111 (0 à 127 en décimal). Ceci représente la norme ou grandeur du nombre. Comme la plupart des ordinateurs traitent aussi bien les nombres négatifs que les nombres positifs, il faut adopter une certaine convention pour représenter le signe du nombre (+ ou -). Généralement, on ajoute un autre bit au nombre, appelé le bit de signe. La convention la plus courante consiste à attribuer au nombre positif le bit de signe 0 et au nombre négatif le bit de signe 1. On peut voir un exemple de ceci à la figure 3. Le registre A contient les bits 0110100. Le 0 du bit le plus à gauche (A6) est le bit de signe qui correspond au signe +. Les autres 6 bits indiquent la grandeur du nombre 110100₂, soit 52 en décimal. Donc le nombre mémorisé dans le registre A est + 52. De la même manière, on trouve que le nombre stocké dans le registre B est - 52, puisque le bit de signe est 1, ce qui correspond au signe -.

On utilise le bit de signe pour indiquer si le nombre binaire mémorisé est positif ou négatif. Les nombres reproduits à la Figure 3 sont formés d'un signe de bit et de six bits de grandeur. Ces derniers correspondent à l'équivalent binaire exact de la valeur décimale présentée. On parle dans ce cas d'une notation signe-grandeur pour représenter des nombres binaires signés.

A6	A5	A4	A3	A2	A1	A0	
0	1	1	0	1	0	0	= +52 ₁₀
Bit de signe							Grandeur = 52 ₁₀

A6	A5	A4	A3	A2	A1	A0	
1	1	1	0	1	0	0	= -52 ₁₀
Bit de signe							Grandeur = 52 ₁₀

Figure 3 Représentation de nombres binaires signés dans la notation signe-grandeur.

Bien que la notation signe-grandeur soit directe, les ordinateurs et les calculatrices n'y ont généralement pas recours, en raison de la complexité des circuits qui matérialisent, cette notation. On utilise plutôt dans ces machines, pour représenter les nombres binaires signés la notation en complément à 2. Avant d'aborder le déroulement de tout ceci, il importe de voir comment on obtient l'équivalent en complément à 1 et l'équivalent en complément à deux, d'un nombre binaire.

Notation en complément à 1

Le complément à 1 d'un nombre binaire s'obtient en changeant chaque 0 par un 1 et chaque 1 par un 0. Autrement dit, en complémentant chaque bit du nombre. Voici une illustration de cette marche à suivre:

1 0 1 1 0 1 nombre binaire initial
0 1 0 0 1 0 complément de chaque bit pour obtenir le complément à 1

On dit que le complément à 1 de 101101 est 010010.

Expression du complément à 1 : C₁(N) = not N

Notation en complément à 2

Le complément est très largement utilisé car c'est la représentation naturelle des nombres négatifs. Si nous faisons la soustraction de $2 - 3$ nous obtenons immédiatement -1 représenté en complément à 2 .

$$\begin{array}{r}
 & 0 & 0 & 1 & 0 & \text{nombre 2 en binaire sur 4 bits} \\
 - & 0 & 0 & 1 & 1 & \text{nombre 3 en binaire sur 4 bits} \\
 = & 1 & 1 & 1 & 1 & \text{résultat de la soustraction, il y a un emprunt}
 \end{array}$$

Nous verrons que "1111" est la représentation du nombre -1 sur 4 bits.

Le complément à 2 d'un nombre binaire s'obtient simplement en prenant le complément à 1 de ce nombre et en ajoutant 1 au bit de son rang de poids le plus faible. Voici une illustration de cette conversion pour le cas $101101_2 = 45_{10}$.

$$\begin{array}{r}
 & 1 & 0 & 1 & 1 & 0 & 1 & \text{équivalent binaire de 45} \\
 & 0 & 1 & 0 & 0 & 1 & 0 & \text{complément de chaque bit pour obtenir le complément à 1} \\
 + & & & & & & 1 & \text{addition de 1 pour obtenir le complément à 2} \\
 = & & & 0 & 1 & 0 & 0 & 1 & 1 \text{ le complément à 2 du nombre binaire initial}
 \end{array}$$

On dit que 010011 est le complément à 2 de 101101.

Expression du complément à 2 : $C_2(N) = C_1(N) + 1 = \text{not } N + 1$

Etude de nombres binaires signés en complément à 2

Voici comment on écrit des nombres binaires signés en utilisant la notation en complément à 2.

Si le nombre est positif, sa grandeur est la grandeur binaire exacte et son bit de signe est un 0 devant le bit de poids le plus fort. C'est ce qu'on peut voir sur la figure 4 pour $+45_{10}$.

0	1	0	1	1	0	1	= +45 ₁₀
Bit de signe	Grandeur exacte						
1	0	1	0	0	1	1	= -45 ₁₀
Bit de signe	Complément à 2						

Figure 4 Écriture de nombres binaires signés dans la notation en complément à 2.

Si le nombre est négatif, sa grandeur est le complément à 2 de la grandeur exacte et son bit de signe est un 1 à gauche du bit de poids le plus fort. Voyez à la figure 4 la représentation du nombre -45_{10} .

La complémentation à 2 d'un nombre signé transforme un nombre positif en un nombre négatif et vice versa.

La notation en complément à 2 est employée pour exprimer les nombres binaires signés parce que, comme nous le verrons, on parvient grâce à elle à soustraire en effectuant en réalité une addition. Cela n'est pas négligeable dans le cas des ordinateurs, puisque avec les mêmes circuits, nous parvenons à soustraire et à additionner.

Dans de nombreuses situations, le nombre de bits est fixé par la longueur des registres qui contiennent les nombres binaires, d'où la nécessité d'ajouter des 0 pour avoir le nombre de bits requis :

Comme exemple nous allons exprimer + 2 au moyen de 5 bits:

$$+2 = 00010$$

$$\begin{array}{r}
 11101 \text{ (complément à 1)} \\
 + \quad 1 \text{ (ajouter 1)} \\
 \hline
 11110 \text{ (complément à 2 du chiffre - 2 sur 5 bits)}
 \end{array}$$

Cas spécial de la notation en complément à 2

Quand un nombre signé a 1 comme bit de signe et que des 0 comme bits de grandeur, son équivalent décimal est -2^N , où N est le nombre de bits de grandeur. Par exemple :

$$\begin{aligned}
 1000 &= -2^3 = -8 \\
 10000 &= -2^4 = -16 \\
 100000 &= -2^5 = -32
 \end{aligned}$$

et ainsi de suite.

Par conséquent, on peut affirmer que l'intervalle complet des valeurs que l'on peut écrire en complément à 2 au moyen de N bits de grandeur est :

$$-2^N \text{ à } +(2^N - 1)$$

Il y a un total de 2^{N+1} valeurs différentes, en comptant le zéro.

Addition en complément à 2

La notation en complément à 2 et la notation en complément à 1 sont très semblables. Toutefois, la notation en complément à 2 jouit généralement de certains avantages quand vient le temps de construire des circuits. Nous allons maintenant étudier comment les machines numériques additionnent et soustraient quand les nombres négatifs sont écrits dans la notation en complément à 2. Dans tous les cas étudiés, il est important que vous remarquiez que le bit de signe de chaque nombre est traité sur le même pied que les bits de la partie grandeur.

Cas 1: deux nombres positifs

L'addition de deux nombres positifs est immédiate. Soit l'addition de + 9 et + 4:

$$\begin{array}{r}
 + 9 \quad | \quad 0 \quad 1001 \text{ (cumulande)} \\
 + 4 \quad | \quad 0 \quad 0100 \text{ (cumulateur)} \\
 \hline
 + 13 \quad | \quad 0 \quad 1101 \text{ (Somme)}
 \end{array}$$

↑ Bits de signe

Remarquez que les bits de signe du cumulande et du cumulateur sont 0 et que celui de la somme est aussi 0, ce qui indique un nombre positif. Notez aussi qu'on a fait en sorte que le cumulande et le cumulateur aient le même nombre de bits. Il faut toujours s'assurer de cela dans la notation en complément à 2.

Cas 2: nombre positif et nombre négatif plus petit

Soit l'addition de + 9 et de - 4. Rappelez-vous que - 4 est exprimé dans la notation en complément à 2. Donc +4(00100) doit être converti en -4(11100)

$$\begin{array}{r}
 \downarrow \text{ Bits de signe} \\
 + 9 \quad | \quad 0 \quad 1001 \text{ (cumulande)} \\
 - 4 \quad | \quad 1 \quad 1100 \text{ (cumulateur)} \\
 \hline
 + 5 \quad | \quad \pm 0 \quad 0101
 \end{array}$$

↑ ce report n'est pas pris en considération de sorte que le résultat est 00101 (somme = + 5)

Dans ce cas-ci, le bit de signe du cumulateur est 1. Remarquez que les bits de signe sont aussi additionnés. En fait, un report est produit au moment de l'addition du dernier rang. Ce report est toujours rejeté d'où la somme finale de 00101, soit le nombre décimal + 5.

Cas 3: nombre positif et nombre négatif plus grand.

Soit l'addition de - 9 et de + 4:

$$\begin{array}{r}
 -9 \quad 10111 \\
 +4 \quad 00100 \\
 \hline
 -5 \quad 11011
 \end{array}$$

↑ Bit de signe négatif

Dans ce cas-ci le bit de signe de la somme est 1, ce qui indique un nombre négatif. Comme la somme est un nombre négatif, la réponse est le complément à 2 de la grandeur exacte. Donc 1011 est en réalité le complément à 2 de la somme. Pour trouver la grandeur exacte de la somme, on doit prendre le complément à 2 de 1011, ce qui donne 0101 (5); la réponse est donc 11011 = - 5.

On peut également vérifier le résultat en additionnant le poids de chaque bit, avec le bit de signe qui vaut -2^N , où N est le nombre de bits de grandeur.

$$-1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -16 + 8 + 2 + 1 = -5$$

Cas 4: deux nombres négatifs

$$\begin{array}{r}
 -9 \quad 10111 \\
 -4 \quad 11100 \\
 \hline
 -13 \quad \pm 10011
 \end{array}$$

↑ ce report n'est pas pris en considération, le résultat est donc 10011 (somme = -13)

Le résultat définitif est de nouveau négatif (-13).

Cas 5 : nombres égaux et opposés

$$\begin{array}{r}
 -9 \quad 10111 \\
 +9 \quad 01001 \\
 \hline
 0 \quad \pm 00000
 \end{array}$$

↑ ce report n'est pas pris en considération, le résultat est donc 00000 (somme = + 0)

Le résultat est évidemment + 0, comme on s'y attendait.

Soustraction: complément à 2

Une opération de soustraction qui porte sur des nombres exprimés dans la notation en complément à 2 est en réalité une opération d'addition qui diffère peu des cas examinés précédemment. Quand on soustrait un nombre binaire (le diminuteur) d'un autre nombre (le diminuande), la marche à suivre est comme suit:

1. Prendre le complément à 2 du diminuteur, y compris son bit de signe. Si ce dernier est un nombre positif, il deviendra un nombre négatif dans la notation en complément à 2. Si le diminuteur est un nombre négatif, la complémentation à 2 en fera un nombre positif écrit en grandeur exacte. Autrement dit, nous changeons le signe du diminuteur.
2. Après avoir complémenté à 2 le diminuteur, on l'ajoute au diminuande. Le diminuande conserve sa forme initiale. Le résultat de cette addition représente la différence recherchée. Le bit de signe de la différence indique si la réponse est positive ou négative et si on est en notation binaire exacte ou en notation en complément à 2.
3. Rappelez-vous que les deux nombres doivent avoir le même nombre de bits.

Examinons la soustraction suivante: +9 - (+4).

$$\begin{array}{r} \text{diminuande} \quad (+9) \quad 01001 \\ \text{diminuteur} \quad (+4) \quad 00100 \end{array}$$

Changez le diminuteur pour sa version en complément à 2 (11100), ce qui représente -4. Maintenant additionnez-le au diminuande.

$$\begin{array}{r} +9 \quad 01001 \\ -4 \quad 11100 \\ \hline +5 \quad 00101 \end{array}$$

↑
la retenue est rejetée, le résultat est donc 00101 = +5

Quand on complémentise à 2 le diminuteur, on obtient en réalité -4, de sorte qu'on additionne +9 à -4, ce qui est équivalent à soustraire de +9 le nombre +4. En définitive, toute opération de soustraction se résume à une addition lorsqu'on utilise la notation en complément à 2. Cette caractéristique de la notation en complément à 2 explique pourquoi c'est la méthode la plus utilisée, puisqu'on peut additionner et soustraire en utilisant les mêmes circuits.

Dépassement (overflow)

Dans chacun des exemples d'addition et de soustraction que l'on vient d'étudier, les nombres que l'on a additionnés étaient constitués à la fois d'un bit de signe et de 4 bits de grandeur. Les réponses aussi comportaient un bit de signe et 4 bits de grandeur. Tout report fait sur le bit de sixième rang était rejeté. Dans tous les cas étudiés, la grandeur de la réponse ne dépassait jamais la capacité des 4 bits. Voyons l'addition de +9 à +8.

$$\begin{array}{r} +9 \quad 0 \quad 1001 \\ +8 \quad 0 \quad 1000 \\ \hline 1 \quad 0001 \end{array}$$

↑
Bit de signe

Le bit de signe de la réponse est celui d'un nombre négatif, ce qui est manifestement une erreur. La réponse devrait être +17. Étant donné que la grandeur est 17, il faut plus de 4 bits pour l'exprimer, et il y a donc un dépassement (overflow) sur le rang du bit de signe. Une condition de dépassement donne toujours lieu à un résultat inexact, et on la détecte en examinant le bit de signe du résultat et en le comparant aux bits de signe des nombres additionnés. En additionnant deux nombres de signes différents, il ne peut pas y avoir de

dépassement, par contre lorsqu'on additionne deux nombres de même signe, on a un dépassement si le signe du résultat est différent du signe des deux nombres additionnés. Dans un ordinateur, il existe un circuit spécialement conçu pour détecter les conditions de débordement et pour indiquer que la réponse est fausse.

Multiplication de nombres binaires

On multiplie les nombres binaires de la même façon qu'on multiplie les nombres décimaux. En réalité, le processus est plus simple car les chiffres du multiplicateur sont toujours 0 ou 1, de sorte qu'on multiplie toujours par 0 ou par 1. Voici un exemple de multiplication de nombres binaires non signés.

$$\begin{array}{r}
 1 \ 0 \ 0 \ 1 \\
 \cdot \quad 1 \ 0 \ 1 \ 1 \\
 \hline
 1 \ 0 \ 0 \ 1 \\
 1 \ 0 \ 0 \ 1 \\
 0 \ 0 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 0 \ 1 \\
 \hline
 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1
 \end{array}
 \begin{array}{l}
 \text{multiplicande} = 9_{10} \\
 \text{multiplicateur} = 11_{10} \\
 \text{produits partiels} \\
 \text{produit final} = 99_{10}
 \end{array}$$

Dans cet exemple, le multiplicande et le multiplicateur sont en notation binaire exacte et il n'y a pas de bit de signe. La marche à suivre est exactement la même que pour les multiplications décimales. D'abord, examinons le bit de poids le plus faible du multiplicateur; dans notre exemple il s'agit d'un 1. Ce 1 multiplie le multiplicande pour donner 1001, c'est notre premier produit partiel. Ensuite, examinons le deuxième bit du multiplicateur. Il s'agit d'un 1, ce qui donne le second produit partiel. Notez qu'en écrivant le second produit partiel on le décale d'un rang vers la gauche par rapport au premier. Le troisième bit du multiplicateur est 0 et notre troisième produit partiel est 0000; ce troisième produit est aussi décalé d'un rang vers la gauche par rapport au produit partiel précédent. Le quatrième bit du multiplicateur est 1, de sorte que le dernier produit partiel est 1001, que l'on écrit à nouveau décalé d'un rang vers la gauche. On additionne ensuite les quatre produits partiels pour obtenir le produit final.

La plupart des machines numériques ne peuvent additionner que deux nombres binaires à la fois. C'est la raison pour laquelle les produits partiels d'une multiplication ne peuvent être additionnés ensemble en une seule fois. Ils sont plutôt additionnés deux par deux, c'est-à-dire que le premier est additionné au second, que leur somme est additionnée au troisième et ainsi de suite.

Multiplication en complément à 2

Dans les machines qui utilisent la notation en complément à 2, la multiplication est effectuée de la façon décrite ci-dessus quand le multiplicande et le multiplicateur sont exprimés en notation binaire exacte. Si les deux nombres multipliés sont positifs, ils sont déjà dans cette notation et ils sont multipliés tels quels. Le produit résultant est évidemment positif et son bit de signe est 0. Quand les deux nombres sont négatifs, ils sont donc écrits dans la notation en complément à 2. Chacun de ces nombres est complémenté à 2 pour obtenir un nombre positif et ce sont les résultats de ces complémentations qu'on multiplie. Le produit est un nombre positif dont le bit de signe est 0.

Quand un des nombres est positif et que l'autre est négatif, le nombre négatif est d'abord complémenté à 2 pour obtenir une grandeur positive. Le produit est exprimé selon la notation en grandeur exacte. Cependant, le produit doit être négatif car les nombres à multiplier sont de signes opposés. Par conséquent, on complémente à 2 le produit et on ajoute le bit de signe 1.

Division binaire

La division d'un nombre binaire (le dividende) par un autre (le diviseur) est identique à la division de deux nombres décimaux. En réalité, la division en binaire est plus simple puisque pour déterminer combien de fois le diviseur entre dans le dividende, il n'y a que 2 possibilités 0 ou 1. Voici des exemples de divisions:

$$\begin{array}{r}
 1\ 0\ 0\ 1 \\
 0\ 1\ 1 \\
 \hline
 0\ 0\ 1\ 1
 \end{array} \quad \left| \begin{array}{l} 11 \\ \hline 011 \\ \hline \end{array} \right. \quad (9/3 = 3)$$

$$\begin{array}{r}
 1\ 0\ 1\ 0\ ,0 \\
 1\ 0\ 0 \\
 \hline
 0\ 0\ 1\ 0\ 0 \\
 1\ 0\ 0
 \end{array} \quad \left| \begin{array}{l} 100 \\ \hline 10,1 \\ \hline \end{array} \right. \quad (10/4 = 2,5)$$

Dans la plupart des ordinateurs modernes, les soustractions qui ont lieu durant une opération de division sont généralement des soustractions avec complément à 2, c'est-à-dire on complémente à 2 le soustracteur puis on effectue l'addition.

La division de nombres signés s'effectue de la même façon que la multiplication. Les nombres négatifs sont complémentés pour obtenir des nombres positifs puis la division est effectuée. Si le dividende et le diviseur sont de signes opposés, le quotient est complémenté à 2 pour obtenir un nombre négatif, puis on lui ajoute un bit de signe de 1. Si le dividende et le diviseur ont le même signe, le quotient est laissé sous sa forme positive et on lui ajoute un bit de signe de 0.

Addition en BCD

De nombreux ordinateurs représentent les nombres décimaux au moyen du code BCD. Rappelons que ce code fait correspondre à chaque chiffre décimal un code de 4 bits compris entre 0000 et 1001. L'addition de nombres décimaux exprimés sous forme BCD se comprend mieux en étudiant deux cas qui peuvent survenir quand on additionne deux chiffres décimaux.

Somme égale ou inférieure à 9

Additionnons 5 à 4 en utilisant pour chacun leur représentation BCD

$$\begin{array}{r}
 5 \\
 + 4 \\
 \hline
 9
 \end{array} \quad \begin{array}{r}
 0101 \\
 + 0100 \\
 \hline
 1001
 \end{array} \quad \begin{array}{l}
 \text{BCD de 5} \\
 \text{BCD de 4} \\
 \text{BCD de 9}
 \end{array}$$

L'addition est effectuée comme une addition binaire normale et la somme est 1001, soit le code BCD de 9. Voici un autre exemple: additionnons 45 à 33:

$$\begin{array}{r}
 45 \\
 + 33 \\
 \hline
 78
 \end{array} \quad \begin{array}{r}
 0100\ 0101 \\
 + 0011\ 0011 \\
 \hline
 0111\ 1000
 \end{array} \quad \begin{array}{l}
 \text{BCD de 45} \\
 \text{BCD de 33} \\
 \text{BCD de 78}
 \end{array}$$

Dans cet exemple. Les codes de 4 bits associés à 5 et à 3 sont additionnés selon les règles binaires pour donner 1000, ce qui est le code BCD de 8. De même, l'addition des chiffres décimaux de second rang donne en binaire 0 111, ce qui est le code BCD de 7. Le total est 0111 1000, soit le code BCD de 78.

Dans les exemples précédents, aucune somme de deux chiffres décimaux ne dépassait 9 ; donc, il n'y a pas eu de reports décimaux. Dans cette situation l'addition BCD est un processus direct équivalent à l'addition binaire.

Somme supérieure à 9

Additionnons 5 et 7 en BCD:

$$\begin{array}{r}
 5 \\
 + 7 \\
 \hline
 12
 \end{array}
 \quad
 \begin{array}{r}
 0101 \\
 + 0111 \\
 \hline
 1100
 \end{array}
 \quad
 \begin{array}{l}
 \text{BCD de 5} \\
 \text{BCD de 7} \\
 \text{Code invalide en BCD}
 \end{array}$$

La somme 1100 n'existe pas dans le code BCD; il s'agit de l'une des six représentations codées de 4 bits interdites ou non valides. Cette représentation est apparue parce qu'on a additionné deux chiffres dont la somme dépasse 9. Dans un tel cas, il faut corriger la somme en additionnant 6 (0110) afin de prendre en considération le fait qu'on saute six présentations codées non valides:

$$\begin{array}{r}
 5 \\
 + 7 \\
 \hline
 12
 \end{array}
 \quad
 \begin{array}{r}
 0101 \\
 + 0111 \\
 \hline
 1100
 \end{array}
 \quad
 \begin{array}{l}
 \text{BCD de 5} \\
 \text{BCD de 7} \\
 \text{Somme non valide en BCD} \\
 + 0110 \\
 \hline
 0001 \quad 0010
 \end{array}
 \quad
 \begin{array}{l}
 \text{Additionner 6 pour corriger} \\
 \text{BCD de 12}
 \end{array}$$

Comme on le montre ci-dessus, l'addition de 0110 à la somme non valide donne la représentation BCD exacte. Notez qu'un report a lieu sur le chiffre décimal de deuxième rang. Il est obligatoire d'additionner 0110 quand la somme de deux chiffres décimaux dépasse 9.

Voyons un autre exemple: additionnez 47 à 35 en BCD:

$$\begin{array}{r}
 47 \\
 + 35 \\
 \hline
 82
 \end{array}
 \quad
 \begin{array}{r}
 0100 \quad 0111 \\
 + 0011 \quad 0101 \\
 \hline
 0111 \quad 1100
 \end{array}
 \quad
 \begin{array}{l}
 \text{BCD de 47} \\
 \text{BCD de 35} \\
 \text{Somme non valide dans le premier chiffre} \\
 + 1 \quad 0110 \\
 \hline
 1000 \quad 0010
 \end{array}
 \quad
 \begin{array}{l}
 \text{Additionner 6 pour corriger} \\
 \text{Somme BCD exacte}
 \end{array}$$

L'addition des codes de 4 bits pour 7 et 5 produit une somme non valide que l'on corrige en additionnant 0110. Notez que ceci produit un report de 1, qui est ajouté à la somme BCD des chiffres du second rang.
Récapitulation de l'addition en BCD:

1. Addition binaire ordinaire des représentations BCD de tous les rangs.
2. Pour les rangs où la somme est égale ou inférieure à 9, aucune correction ne s'impose et la somme est une représentation BCD valide.
3. Quand la somme des deux chiffres est supérieure à 9, on ajoute une correction de 0110 pour obtenir la représentation BCD exacte. Il se produit toujours un report sur le chiffre de rang immédiatement à gauche, soit lors de l'addition initiale (première étape) ou de l'addition de la correction.

Portes logiques et algèbre Booléenne

Introduction

Tous les circuits numériques fonctionnent en mode binaire, c'est-à-dire un mode dans lequel les tensions de sortie et d'entrée sont 0 ou 1; les valeurs 0 et 1 correspondent à des plages de tensions définies à l'avance. Cette caractéristique des circuits logiques qui nous permet de recourir à l'algèbre de Boole pour l'analyse et la conception de systèmes numériques. Dans ce chapitre, nous étudierons les portes logiques, qui constituent les blocs élémentaires des circuits logiques et nous verrons comment il est possible de décrire leur fonctionnement grâce à l'algèbre booléenne. Aussi, nous vous apprendrons comment on réussit à construire des circuits logiques en combinant les portes et comment l'algèbre de Boole parvient à décrire et à analyser ces derniers.

Constantes et variables Booléennes

L'algèbre booléenne se distingue principalement de l'algèbre ordinaire par des constantes et des variables qui ne peuvent prendre que les deux valeurs possibles 0 et 1. Une variable booléenne est une grandeur qui peut, à des moments différents, avoir la valeur 1 ou 0. Les variables booléennes servent souvent à représenter un niveau de tension sur un fil ou aux bornes d'entrée ou de sortie d'un circuit. Par exemple, dans un certain circuit numérique, on pourra avoir attribué la valeur booléenne 0 à l'intervalle de tensions 0 à 0,8 V, et la valeur booléenne 1 à l'intervalle 2 à 5 V. Les tensions comprises entre 0,8 et 2 V sont indéterminées (ne correspondent ni à 1 ni à 0) et ne doivent jamais survenir.

Ainsi, les valeurs booléennes 0 et 1 ne représentent pas des nombres réels mais plutôt l'état d'une variable électrique ou ce qui est convenu d'appeler un niveau logique. On dit que la tension d'un circuit numérique est au niveau logique 1 ou au niveau logique 0, selon la valeur réelle de cette tension. Dans le domaine de la logique numérique, on utilise d'autres expressions qui sont synonymes de 0 et 1. Certaines de ces expressions sont représentées dans le Tableau 4 ci-dessous.

Tableau 4 Diverses appellations pour les niveaux logiques

Niv. logique 0	Niv. logique 1
Faux	Vrai
Arrêt	Marche
Bas	Haut
Non	Oui
Ouvert	Fermé

L'algèbre de Boole est un outil qui permet d'exprimer les effets qu'ont les divers circuits numériques sur les entrées logiques et de manipuler les variables logiques en vue de déterminer la meilleure façon de matérialiser une certaine fonction logique. Dans le reste de ce cours, nous allons utiliser des lettres comme symboles pour représenter les variables logiques. Par exemple, A pourra correspondre à une entrée ou à une sortie d'un certain circuit numérique: en tout temps, A sera égal soit à 1 ou à 0.

Parce qu'il n'y a que deux valeurs possibles, l'algèbre booléenne se manipule plus aisément que l'algèbre ordinaire. En algèbre booléenne, il n'y a pas de fraction, de partie décimale, de nombre négatif, de racine carrée, de racine cubique, de logarithmes, de nombre imaginaire... En fait, dans cette algèbre, on ne retrouve que trois opérations élémentaires:

1. L'addition logique, dite aussi opération OU (OR). Le symbole habituel de cette opération est le signe (+).
2. La multiplication logique, dite aussi opération ET (AND). Son symbole habituel est le signe de la multiplication (·).
3. La complémentation ou l'inversion logique, dite aussi opération NON (NOT). Son symbole habituel est une barre de surlignement (̄).

Tables de vérité

De nombreux circuits logiques possèdent plusieurs entrées mais seulement une sortie. Une table de vérité nous fait connaître la réaction d'un circuit logique (sa valeur de sortie) aux diverses combinaisons de niveaux logiques appliqués aux entrées. La Figure 4 nous montre des tables de vérité à deux, trois et quatre colonnes d'entrées.

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

a)

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

b)

A	B	C	D	X
0	0	0	0	?
0	0	0	1	?
0	0	1	0	?
0	0	1	1	?
0	1	0	0	?
0	1	0	1	?
0	1	1	0	?
0	1	1	1	?
1	0	0	0	?
1	0	0	1	?
1	0	1	0	?
1	0	1	1	?
1	1	0	0	?
1	1	0	1	?
1	1	1	0	?
1	1	1	1	?

c)

Figure 4 a) table de vérité à deux entrées; b) table à trois entrées; c) table à quatre entrées.

Dans chacune de ces tables, toutes les combinaisons possibles de 0 et de 1 pour les entrées (A, B, C, D) apparaissent à gauche, tandis que le niveau logique résultant de la sortie, X, est donné à droite. Pour le moment, il n'y a que des « ? » dans ces colonnes, car les valeurs de sortie sont différentes pour chaque type de circuit.

Notez que dans la table de vérité à deux entrées il y a quatre lignes, que dans celle à trois entrées il y a huit lignes et que dans la table à quatre entrées, il y en a seize. Pour une table de N entrées, il y a 2^N lignes. De plus, vous remarquez sans doute que la succession des combinaisons correspond à la suite du comptage binaire, de sorte que la détermination de toutes les combinaisons est directe et qu'on ne peut pas en oublier.

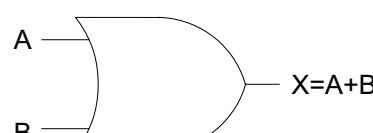
L'opération OU (OR)

Soit deux variables logiques indépendantes, A et B. Quand on combine A et B au moyen de l'addition logique, le résultat X est exprimé par: $X = A + B$

Dans cette équation, le signe + n'indique pas une addition ordinaire, mais plutôt l'addition logique (opération OU) dont les règles sont définies dans la table de vérité de la Figure 5a). L'opération OU donne la valeur 1 quand l'une quelconque des variables d'entrée est 1.

A	B	$X = A+B$
0	0	0
0	1	1
1	0	1
1	1	1

a)



b) porte OU (OR)

Figure 5 a) Table de vérité définissant l'opération OU; b) symbole de circuit servant à représenter une porte OU à deux entrées.

La porte OU (OR)

En électronique numérique une porte OU est un circuit ayant au moins deux entrées et dont la sortie est égale à la somme logique (OU) des entrées. La Figure 5b) nous fait voir le symbole utilisé pour représenter une porte OU à deux entrées. Les entrées A et B sont des niveaux de tension logiques, et la sortie X est également un niveau de tension logique qui résulte de l'addition logique de A et B, d'où $X = A + B$. Autrement dit, la porte OU a un fonctionnement tel que sa sortie est à un niveau haut (niveau 1) quand au moins une de ses entrées est au niveau haut. À plus forte raison, quand les deux entrées sont au niveau haut. Cette porte sera au niveau bas (niveau 0) si toutes ses entrées sont au niveau bas. La même discussion s'applique intégralement à une porte OU à plus de deux entrées.

L'opération ET (AND)

Si deux variables logiques A et B sont combinées par la multiplication logique (opération ET), le résultat X s'exprime symboliquement ainsi:

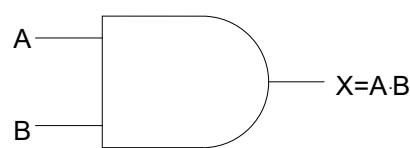
$$X = A \cdot B$$

Dans cette expression, le signe \cdot signifie l'opération booléenne ET, dont les règles d'opération sont données dans la table de vérité de la Figure 6a).

D'après cette table, vous devriez pouvoir déduire facilement que la multiplication logique est exactement comme la multiplication ordinaire. Quand A ou B est 0, le produit est nul; quand A et B sont 1, leur produit est 1. Il nous est donc possible d'affirmer que dans l'opération ET la réponse est 1 si et seulement si toutes les entrées sont 1, et qu'elle est 0 dans tous les autres cas.

A	B	X = A · B
0	0	0
0	1	0
1	0	0
1	1	1

a)



b) porte ET (AND)

Figure 6 a) Table de vérité définissant l'opération ET; b) symbole d'une porte ET à deux entrées.

La Porte ET (AND)

La Figure 6b) nous fait voir une porte ET à deux entrées. La sortie de cette porte est égale au produit logique (ET) des entrées logiques, c'est-à-dire $X = A \cdot B$. Exprimée autrement, la porte ET est un circuit qui fonctionne pour que sa sortie soit à 1 seulement quand toutes ses entrées sont aussi à 1. Dans tous les autres cas, la sortie de la porte ET est à 0. L'opération des portes ET à plus de deux entrées est analogue à ce qu'on vient de dire.

L'opération NON (NOT)

L'opération NON, contrairement aux opérations ET et OU, ne concerne qu'une variable d'entrée. Par exemple, si la variable A est soumise à une opération NON, le résultat X est donné par l'expression:

$$X = \overline{A}$$

Où le trait de surlignement représente l'opération NON. L'opération NON porte également le nom d'inversion ou de complémentation. On trouve un autre signe pour indiquer une inversion: il s'agit de l'apostrophe ('). Donc :

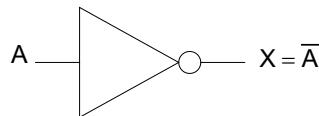
$$A' = \overline{A}$$

Le circuit INVERSEUR (NOT)

On peut voir à la Figure 7b) le symbole d'un circuit NON, appelé plus couramment inverseur. Un tel circuit n'a toujours qu'une entrée, et sa sortie prend le niveau logique opposé du niveau logique de l'entrée.

A	X = \bar{A}
0	1
1	0

a)



b) porte NON (NOT)

Figure 7 a) La table de vérité et b) le symbole du circuit NON.

Mise sous forme algébrique des circuits logiques

Tout circuit logique, quelle que soit sa complexité, peut être décrit au moyen des opérations booléennes déjà décrites parce que la porte ET, la porte OU et la porte NON sont les circuits constitutifs élémentaires des systèmes numériques. A titre d'exemple, considérons le circuit de la Figure 8 comprenant trois entrées A, B et C et une seule sortie X. En recourant à l'expression booléenne de chacune des portes, on peut facilement trouver l'équation correspondant à la sortie.

La sortie de la porte ET a pour expression $A \cdot B$; cette combinaison est une entrée de la porte OU dont l'autre entrée est le signal C. Cette dernière porte a pour effet d'additionner logiquement ses entrées, ce qui donne comme expression de sortie $X = A \cdot B + C$. (Cette dernière équation aurait aussi bien pu s'écrire $X = C + A \cdot B$, puisque l'ordre des termes dans une fonction OU n'a pas d'importance.)

Il est convenu que dans une expression contenant des opérateurs ET et OU, ce sont les opérateurs ET qui sont appliqués en premier, sauf s'il y a des parenthèses; dans ce cas, il faut évaluer avant toute chose l'expression entre parenthèses. Cette règle déterminant l'ordre des opérations est la même que celle en vigueur dans l'algèbre courante.

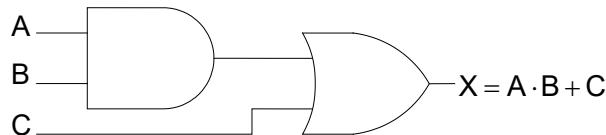


Figure 8 Un circuit et son équation booléenne

Comme exemple supplémentaire, considérons le circuit de la Figure 9. Le résultat la porte OU est simplement $A + B$. La sortie de cette porte aboutit à l'entrée de la porte ET, alors que l'autre entrée de cette dernière reçoit le signal C. L'expression de la sortie de la porte ET est donc $X = (A + B) \cdot C$. Notez l'emploi des parenthèses pour indiquer que A et B sont d'abord additionnés logiquement avant que leur somme OU soit multipliée logiquement avec C. Sans les parenthèses, notre interprétation serait erronée, car $X = A + B \cdot C$ signifie que A est réuni dans une porte OU avec le produit $B \cdot C$.

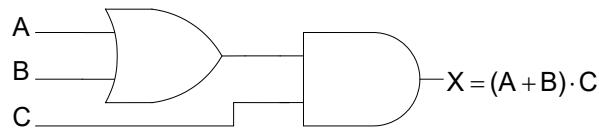


Figure 9 Un circuit logique dont l'expression de sortie comporte des parenthèses.

Circuits renfermant des INVERSEURS

Chaque fois qu'un INVERSEUR se trouve dans le schéma d'un circuit logique, son équation est simplement l'expression de son entrée surmontée d'un trait. On trouve à la figure 10 deux exemples de circuits avec INVERSEURS. Dans la Figure 10a), l'entrée A passe par un INVERSEUR dont la sortie est \bar{A} . Cette sortie de l'INVERSEUR est réunie dans une porte OU avec B, de sorte que l'équation de sortie de cette porte est égale à $\bar{A} + B$.

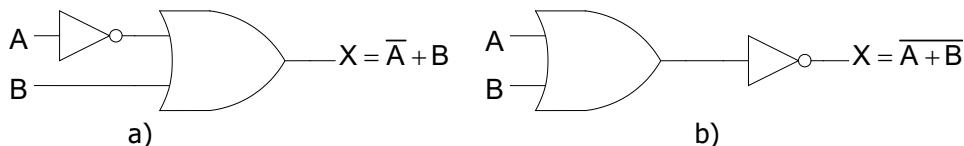


Figure 10 Circuits comprenant des INVERSEURS

Dans la Figure 10b), la sortie de la porte OU égale $A + B$, sortie qui vient alimenter un INVERSEUR. La sortie de ce dernier est donc $(A + B)$, puisque toute l'expression est inversée.

Matérialisation de circuits à partir d'expressions Booléennes

Si l'opération d'un circuit est définie par une expression booléenne, il est possible de tracer directement un diagramme logique à partir de cette expression. Par exemple, si on a besoin d'un circuit tel que $X = ABC$, on sait immédiatement qu'il nous faut une porte ET à trois entrées. Le raisonnement qui nous a servi pour ces cas simples peut être étendu à des circuits plus complexes.

Supposons que l'on veuille construire un circuit dont la sortie est : $Y = AC + B\bar{C} + \bar{A}\bar{B}C$. Cette expression booléenne est constituée de trois termes ($AC, B\bar{C}, \bar{A}\bar{B}C$) qui sont additionnés logiquement. On déduit qu'il nous faut une porte OU à trois entrées auxquelles sont appliqués respectivement les signaux $AC, B\bar{C}, \bar{A}\bar{B}C$. Chaque entrée de la porte OU est un produit logique, ce qui signifie qu'il a fallu trois portes ET alimentées par les entrées appropriées pour produire ces termes. C'est ce qu'on peut voir à la Figure 11 où est tracé le schéma final du circuit.

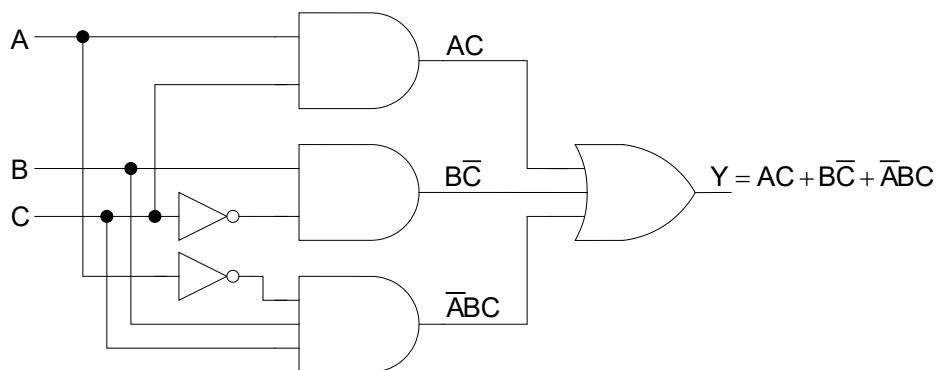


Figure 11 Construction d'un circuit logique à partir d'une expression booléenne.

Portes NI (NOR) et portes NON-ET (NAND)

En technique numérique, on retrouve très souvent deux autres types de portes logiques: la porte NI (NOR) et la porte NON-ET (NAND). En réalité, ces portes correspondent à des combinaisons d'opérations élémentaires ET, OU et NON, et il est relativement facile de les décrire au moyen des fonctions de l'algèbre booléenne que vous connaissez déjà.

La porte NI (NOR)

On peut voir à la

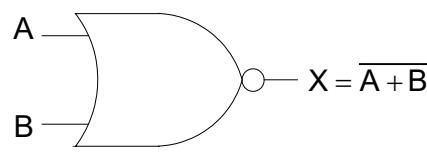
Figure 12b) le symbole d'une porte NI à deux entrées. Vous constaterez que c'est le symbole d'une porte OU sauf qu'il y a un petit rond à la pointe. Ce petit rond correspond à une opération d'inversion. Ainsi, la porte NI a un fonctionnement analogue à une porte OU suivie d'un INVERSEUR. L'expression de sortie d'une porte NI est $X = \overline{A + B}$.

La table de vérité montrée à la

Figure 12a) nous apprenons que la sortie d'une porte NI est exactement l'inverse de celle d'une porte OU pour toutes les conditions des entrées possibles.

A	B	$X = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

a)

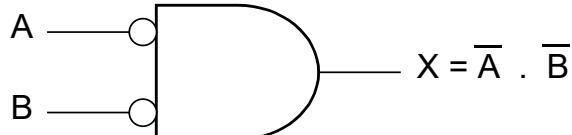


b) porte NI (NOR)

Figure 12 a) La table de vérité et b) le symbole du circuit NI (NOR).

Par De Morgan nous pouvons montrer que la porte NOR est équivalente à :

$$\begin{aligned} X &= \overline{A + B} = \\ &= \overline{\overline{A} \cdot \overline{B}} \end{aligned}$$



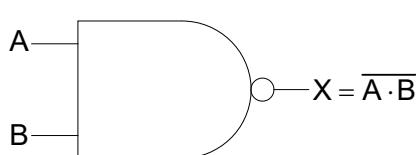
La porte NON-ET (NAND)

On peut voir à la Figure 13a) le symbole d'une porte NON-ET à deux entrées. Vous voyez que c'est le symbole d'une porte ET sauf qu'il y a un petit rond à la pointe. Encore une fois, ce petit rond correspond à une opération d'inversion. Ainsi, la porte NON-ET a un fonctionnement analogue à une porte ET suivie d'un INVERSEUR. L'expression de sortie d'une porte NON-ET est $X = \overline{A \cdot B}$.

La table de vérité montrée à la Figure 13a) nous apprenons que la sortie d'une porte NON ET est exactement l'inverse de celle d'une porte ET pour toutes les conditions des entrées possibles.

A	B	$X = \overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

a)



b) porte NON ET (NAND)

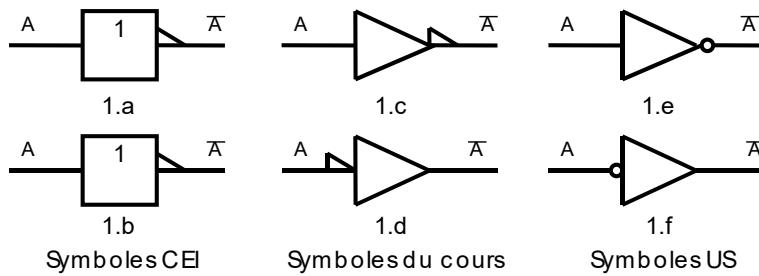
Figure 13 a) La table de vérité et b) le symbole du circuit NON ET (NAND).

Par De Morgan nous pouvons montrer que la porte NAND est équivalent à :

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

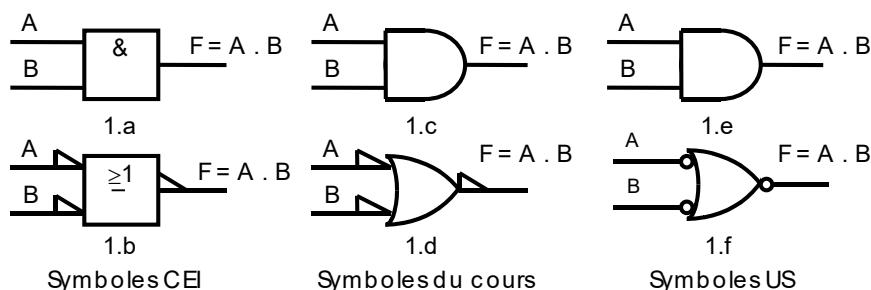
Symbolique des fonctions

Pour représenter les fonctions, nous recourons à un schéma dans lequel les opérateurs logiques seront remplacés par des symboles.

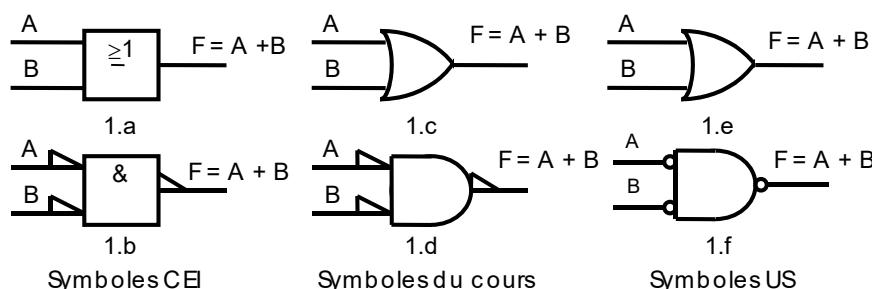


NON ou NOT

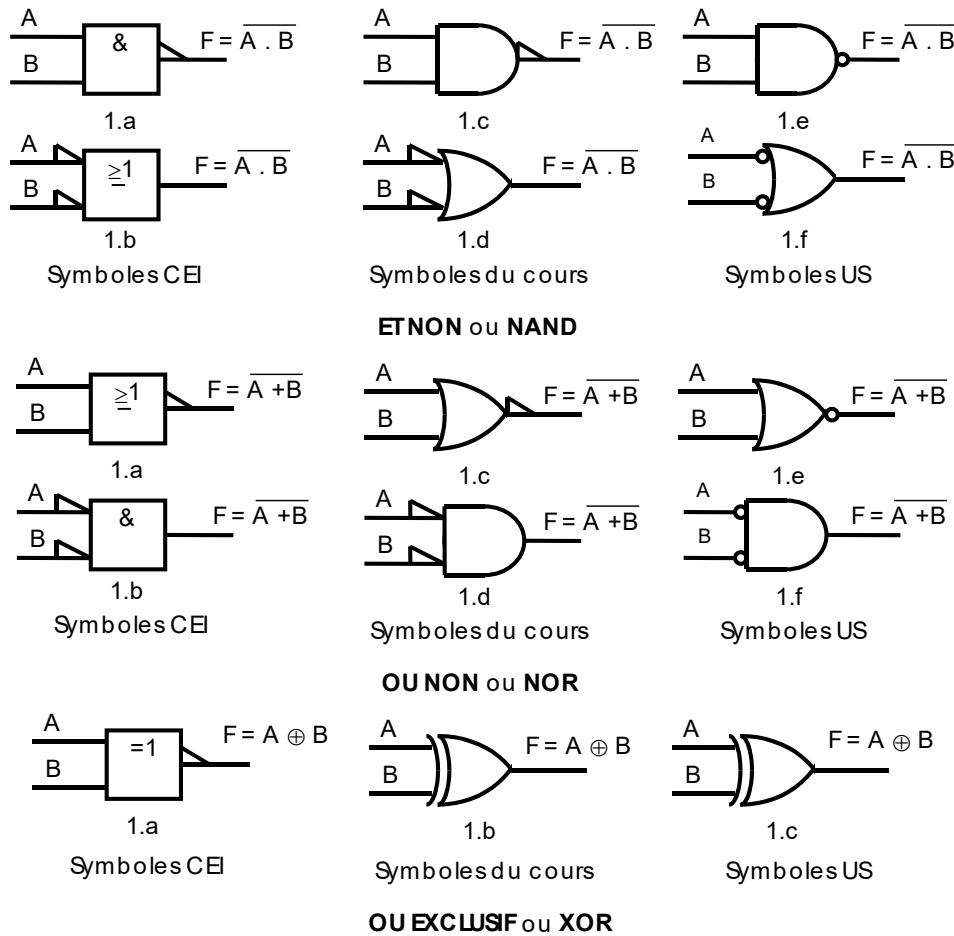
Le grand triangle marque l'amplification, le petit triangle l'inversion. On peut remarquer que l'inversion peut précéder ou suivre l'amplification. Pour les symboles 1.e et 1.f, le rond marque l'inversion. Nous utiliserons de préférence les symboles 1.c ou 1.d. Pour les symboles CEI, l'amplification se marque par un rectangle.



ET ou AND



OU ou OR



Algèbre de Boole

Nous avons vu comment l'algèbre de Boole peut servir à analyser un circuit logique et à exprimer ce dernier sous forme mathématique. Nous allons commencer par les deux postulats qui régissent l'algèbre de BOOLE.

Postulats

Nous aurons deux postulats : $A + \bar{A} = 1$
 $A \cdot \bar{A} = 0$

Ces deux postulats traduisent le fait que l'inverse d'une variable ne peut jamais prendre la même valeur que la variable. Un manquement à ces postulats est possible lors de régimes transitoires dans les circuits. Ce manquement peut entraîner un comportement aléatoire sur les fonctions dépendantes de ces variables.

Théorèmes

Poursuivons maintenant notre étude de l'algèbre booléenne en examinant les théorèmes de Boole qui sont des règles qu'on utilise pour simplifier les expressions logiques et, par le fait même, les circuits logiques. La Figure 14 présente le premier groupe de théorèmes dans lesquels x est une variable logique prenant soit la valeur 0, soit la valeur 1.

- (1) $X \cdot 0 = 0$
- (2) $X \cdot 1 = X$
- (3) $X \cdot X = X$
- (4) $X \cdot \bar{X} = 0$
- (5) $X + 0 = X$
- (6) $X + 1 = 1$
- (7) $X + X = X$
- (8) $X + \bar{X} = 1$

Figure 14 Théorèmes de Boole pour une variable.

Avant de vous présenter les autres théorèmes de Boole, nous tenons à mentionner que dans les théorèmes (1) à (8), la variable X peut correspondre à une expression renfermant plus d'une variable. Par exemple, si nous avons $AB(A\bar{B})$, nous pouvons affirmer, en posant $X = AB$ et d'après le théorème (4), que $AB(A\bar{B}) = 0$.

Théorèmes pour plusieurs variables

Les théorèmes suivants portent sur plus d'une variable:

- (9) $X+Y = Y+X$
- (10) $X \cdot Y = Y \cdot X$
- (11) $X+(Y+Z) = (X+Y)+Z = X+Y+Z$
- (12) $X(YZ) = (XY)Z = XYZ$
- (13a) $X(Y+Z) = XY + XZ$
- (13b) $(W+X) \cdot (Y+Z) = WY+XY+WZ+XZ$
- (14) $X+XY = X$
- (15) $X + \bar{X}Y = X + Y$

Figure 15 Théorèmes de Boole pour plusieurs variables

Les théorèmes (9) et (10) montrent que ET et OU sont des lois de composition commutatives, donc que l'ordre de la multiplication ou de l'addition logique de deux variables n'a pas d'importance, que le résultat reste le même.

Les théorèmes (11) et (12) montrent que ET et OU sont des lois de composition associatives, qui indiquent que l'on peut grouper, comme l'on veut, les variables dans une expression de multiplication ou d'addition logique.

Le théorème (13) fait voir que la multiplication logique est distributive par rapport à l'addition logique, c'est-à-dire que l'on peut développer une expression en la multipliant terme à terme, tout comme dans l'algèbre ordinaire. Ce théorème démontre également que l'on peut mettre en facteur une expression. On veut dire par là que si nous avons une somme de termes, chacun renfermant une variable commune, il est possible de mettre cette variable en facteur, comme on le fait en algèbre ordinaire.

Il est facile de se rappeler des théorèmes (9) à (13) puisqu'ils sont identiques à ceux de l'algèbre ordinaire. Par contre, les théorèmes (14) et (15) ne se retrouvent pas en algèbre ordinaire. On peut les démontrer en passant en revue toutes les possibilités de X et de Y .

Tous ces théorèmes sont d'une grande utilité pour simplifier une expression logique, c'est-à-dire pour obtenir une expression comptant moins de termes. L'expression simplifiée permet de réaliser un circuit moins complexe que celui correspondant à l'expression originale.

Théorèmes de DE MORGAN

Deux des plus importants théorèmes de l'algèbre booléenne nous ont été légués par le mathématicien De Morgan. Les théorèmes de De Morgan se révèlent d'une grande utilité pour simplifier des expressions comprenant des sommes ou des produits de variables complémentés. Voici ces deux théorèmes:

$$(16) \quad (\overline{X+Y}) = \overline{X} \cdot \overline{Y}$$

$$(17) \quad (\overline{X \cdot Y}) = \overline{X} + \overline{Y}$$

Le théorème (16) affirme que la somme logique complémentée de deux variables est égale au produit logique des compléments de ces deux variables. De même, le théorème (17) stipule que le produit logique complémenté de deux variables est égal à la somme logique des compléments de ces deux variables. La démonstration de ces deux théorèmes se fait simplement en considérant toutes les possibilités de x et de y. Bien que ces théorèmes aient été formulés pour les variables simples X et Y, ils demeurent aussi vrais pour les cas où X et Y sont des expressions comprenant plusieurs variables. A titre d'illustration, appliquons ces théorèmes à l'expression suivante :

$$\overline{\overline{AB} + C} = \overline{\overline{AB}} \cdot \overline{C}$$

Le résultat trouvé peut être simplifié une autre fois, car on y retrouve encore un produit logique complémenté. En vertu du théorème (17), il vient:

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

Comme $B = \overline{\overline{B}}$, le résultat définitif est alors:

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

Circuits logiques combinatoires

Introduction

Au chapitre précédent, nous nous sommes penchés sur l'étude de l'ensemble des portes logiques élémentaires et avons réussi, au moyen de l'algèbre booléenne, à décrire et à analyser des circuits matérialisés par des combinaisons de portes logiques. On peut qualifier ces circuits logiques de combinatoires du fait qu'à tout moment le niveau logique recueilli en sortie ne dépend que de la combinaison des niveaux logiques appliqués aux entrées. Un circuit combinatoire ne possède aucun mécanisme de rétention (mémoire); par conséquent. Sa sortie réagit seulement aux signaux présents sur ses entrées.

Dans ce chapitre, nous poursuivons notre étude des circuits combinatoires. D'abord nous poussons plus avant la simplification (minimisation) des circuits. Pour cela, nous utiliserons deux méthodes: les théorèmes de l'algèbre booléenne et une technique graphique, les tables de Karnaugh.

Somme de produits

Les méthodes de simplification et de conception des circuits logiques que nous étudierons exigent que l'on exprime les équations logiques sous la forme d'une somme de produits, dont voici quelques exemples:

- $ABC + \bar{A}\bar{B}\bar{C} + AB\bar{C}$
- $\bar{A} \cdot B + A \cdot \bar{B} \cdot C + \bar{C} \cdot \bar{D} + D$

Chacune de ces trois expressions d'une somme de produits est formée d'au moins deux résultats d'un produit logique (terme ET) mis en relation OU (additionnés logiquement). Chaque terme ET comprend une ou plusieurs variables exprimées sous sa forme normale ou sa forme complémentée. Par exemple, dans la somme de produits $ABC + \bar{A}\bar{B}\bar{C}$, le premier produit logique est constitué des variables A, B et C non complémentées, tandis que le second produit comprend les variables A et C complémentées. Notez que dans une somme de produits, le signe de complémentation ne peut pas surmonter plus d'une variable d'un terme (par ex. on ne peut pas avoir ABC).

Simplification des circuits logiques

Dès qu'on dispose de l'expression d'un circuit logique, il peut être possible de la minimiser pour obtenir une équation comptant moins de termes ou moins de variables par terme. Cette nouvelle équation peut alors servir de modèle pour construire un circuit entièrement équivalent au circuit original mais qui requiert moins de portes et de raccordements.

A titre d'illustration, considérons le circuit de la Figure 16a) que l'on a minimisé pour obtenir le circuit de la Figure 16b). Étant donné que ces deux circuits produisent les mêmes décisions logiques, il va sans dire que le circuit le plus simple est préférable puisqu'il compte moins de portes; il est aussi moins encombrant et moins coûteux à produire.

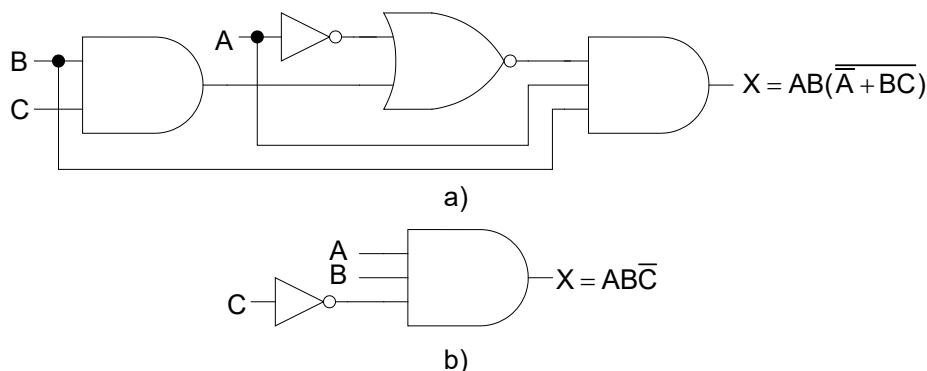


Figure 16 Simplification d'un circuit logique.

Dans les sections suivantes, nous verrons deux façons différentes de simplifier des circuits logiques. Une première façon est fondée sur l'application des théorèmes de l'algèbre booléenne; cette façon, comme nous le verrons, dépend énormément de l'instinct et de l'expérience. L'autre façon (les diagrammes de Karnaugh), au contraire, suit une démarche systématique, semblable à une recette de cuisine.

Simplification algébrique

Les théorèmes de l'algèbre booléenne étudiés au chapitre précédent peuvent nous être d'un grand secours pour simplifier l'expression d'un circuit logique. Malheureusement, il n'est pas toujours facile de savoir quels théorèmes il faut invoquer pour obtenir le résultat minimal. D'ailleurs, rien ne nous dit que l'expression simplifiée est la forme minimale et qu'il n'y a pas d'autres simplifications possibles. Pour toutes ces raisons, la simplification algébrique est souvent un processus d'approximations successives. Il existe cependant deux étapes essentielles :

1. Applications successives des théorèmes de De Morgan en vue d'obtenir une somme de produit
2. Trouver des variables communes pour la mise en facteur de ces dernières.

Exemple : $Z = ABC + A\bar{B} \cdot (\overline{\bar{A} \cdot \bar{C}})$

$$Z = ABC + A\bar{B}(\bar{A} + \bar{C}) \quad \text{théorème 17 (De Morgan)}$$

$$Z = ABC + A\bar{B}(A + C) \quad \text{annulation de la double complémentation}$$

$$Z = ABC + A\bar{B}A + A\bar{B}C \quad \text{multiplication}$$

$$Z = ABC + A\bar{B} + A\bar{B}C \quad \text{théorème 3}$$

$$Z = AC(B + \bar{B}) + A\bar{B} \quad \text{mise en facteur}$$

$$Z = AC(1) + A\bar{B} \quad \text{théorème 8}$$

$$Z = AC + A\bar{B} \quad \text{théorème 2}$$

Conception de circuits logiques combinatoires

Quand on indique le niveau logique recherché pour toutes les conditions d'entrée possibles, on utilise une table de vérité. Ensuite on peut dériver l'expression booléenne du circuit à partir de cette table de vérité. Voici la procédure générale qui aboutit à l'expression de la sortie à partir d'une table de vérité :

1. Pour chaque cas de la table qui donne 1 en sortie, on écrit le produit logique (terme ET) qui lui correspond.
2. On doit retrouver toutes les variables d'entrée dans chaque terme ET soit sous forme directe soit sous forme complémentée. Dans un cas particulier, si variable est 0, alors son symbole est complémenté dans le terme ET correspondant.
3. On additionne logiquement ensuite tous les produits logiques constitués, ce qui donne l'expression définitive de la sortie.

Dès que l'expression de la sortie est établie sous la forme d'une somme de produits à partir de la table de vérité, il est facile de construire le circuit au moyen de portes ET, OU et NON. Il faut une porte ET pour chaque produit logique et une porte OU dont les entrées sont les sorties des portes ET. Généralement, on doit simplifier une telle expression pour obtenir un circuit plus efficace.

Exemple :

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

\overline{ABC}

\overline{ABC}

ABC

On voit qu'il y a trois cas qui produisent une valeur 1 pour la sortie X. Les termes en ET pour chacun de ces cas figurent à droite de la table de vérité. L'expression définitive s'obtient en additionnant logiquement ces trois termes, ce qui donne :

$$X = \overline{ABC} + \overline{ABC} + ABC$$

La méthode des diagrammes de Karnaugh

Le diagramme de Karnaugh est un outil graphique qui permet de simplifier de manière méthodique une équation logique ou le processus de passage d'une table de vérité à son circuit correspondant. Bien que les diagrammes de Karnaugh soient applicables à des problèmes ayant un nombre quelconque de variables d'entrée, ils ne sont plus d'une grande utilité en pratique quand le nombre de variables dépasse six. En plus, dans cette section, nous n'allons pas aborder de problèmes ayant plus de quatre entrées, puisque ceux ayant cinq et six entrées sont des problèmes d'envergure qu'il est préférable de traiter avec un programme informatique.

La forme du diagramme de Karnaugh

Le diagramme K, tout comme la table de vérité, un instrument qui met en évidence les rapports entre les entrées logiques et la sortie recherchée. La Figure 17 nous fait voir trois exemples de diagrammes de Karnaugh pour deux, trois et quatre variables, ainsi que les tables de vérité correspondantes. La table de vérité donne la valeur de la sortie X pour chacune des combinaisons des valeurs d'entrée, par contre, le diagramme K organise l'information de manière différente. Chaque ligne de la table de vérité correspond à un carré du diagramme K. Par exemple, à la figure 17 a), la ligne A = 0 et B = 0 de la table de vérité est le carré AB du diagramme K. Étant donné que pour cette ligne X vaut 1, on inscrit 1 dans ce carré. De même, on associe à la ligne A = 1 et B = 1 le carré qui a les coordonnées AB. Comme X vaut aussi 1 dans ce cas, on retrouve un 1 dans ce carré. Les autres carrés de ce diagramme K contiennent des 0. Le même raisonnement s'applique pour les diagrammes à trois et quatre variables.

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

\overline{AB}

AB

	\overline{B}	B
\overline{A}	1	0
A	0	1

a)

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

\overline{ABC}
 \overline{ABC}
 \overline{ABC}

	\bar{C}	C
AB	1	1
$\bar{A}B$	1	0
AB	1	0
$\bar{A}\bar{B}$	0	0

b)

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

\overline{ABCD}
 \overline{ABCD}
 \overline{ABCD}

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	0	1	0	0
$\bar{A}B$	0	1	0	0
AB	0	1	1	0
$A\bar{B}$	0	0	0	0

c)

Figure 17 Les diagrammes de Karnaugh pour a) deux, b) trois et c) quatre variables.

REUNION

Il est possible de simplifier l'expression de sortie X en combinant selon des règles précises les carrés du diagramme de Karnaugh qui contiennent des 1. On donne à ce processus de combinaison le nom de réunion.

Réunion de doublets (de paires)

La Figure 18 reproduit le diagramme Karnaugh correspondant à une certaine table de vérité à trois variables. Il y a dans ce diagramme deux 1 qui sont voisins verticalement. Ces deux termes peuvent être réunis (combinés), ce qui a pour résultat d'éliminer la variable A. Le même principe joue toujours pour tout doublet de 1 voisins verticalement ou horizontalement. La ligne du haut est considérée comme adjacente à la ligne du bas, idem pour la colonne de gauche avec la colonne de droite.

	\bar{C}	C
$\bar{A}B$	0	0
$\bar{A}B$	1	0
AB	1	0
A \bar{B}	0	0

$$X = BC$$

Figure 18 Exemple de réunion de doublet

Réunion de quartets (groupes de quatre)

Il peut arriver qu'un diagramme K contienne quatre 1 qui soient adjacents. La Figure 19 regroupe plusieurs exemples de tels quartets. En a) les quatre 1 sont voisins horizontalement, alors qu'en b), ils le sont verticalement. Le diagramme de c) renferme quatre 1 rangés en carré qui sont considérés adjacents les uns aux autres. Les quatre 1 de d) sont aussi adjacents, parce que, comme nous l'avons déjà dit, dans un diagramme de Karnaugh les lignes du haut et du bas et les colonnes droite et gauche sont considérées comme adjacentes.

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}B$	0	0	0	0
$\bar{A}B$	0	0	0	0
AB	1	1	1	1
A \bar{B}	0	0	0	0

a)

$$X = AB$$

	\bar{C}	C
$\bar{A}B$	0	1
$\bar{A}B$	0	1
AB	0	1
A \bar{B}	0	1

b)

$$X = C$$

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}B$	0	0	0	0
$\bar{A}B$	0	1	1	0
AB	0	1	1	0
A \bar{B}	0	0	0	0

c)

$$X = BD$$

	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}B$	1	0	0	1
$\bar{A}B$	0	0	0	0
AB	0	0	0	0
A \bar{B}	1	0	0	1

d)

$$X = \bar{BD}$$

Figure 19 Exemple de réunion de quartets.

Réunion d'octets (groupes de huit)

Quand on réunit huit 1 adjacents, on dit qu'on réunit un octet de 1 adjacents. On peut voir à la Figure 20 deux exemples de réunion d'octets. La réunion d'un octet dans un diagramme K à quatre variables donne lieu à l'élimination de trois variables.

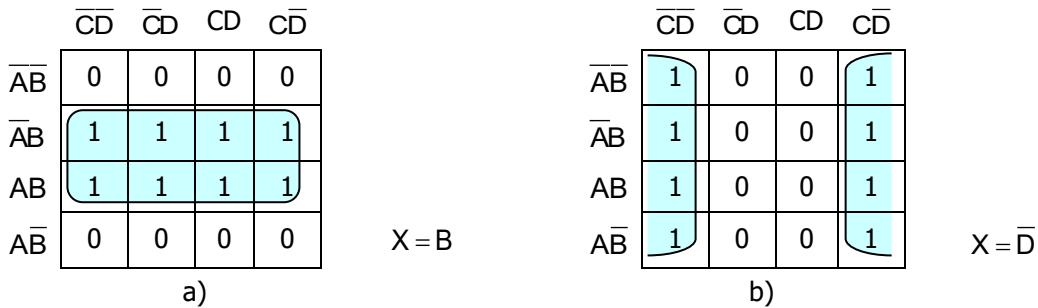


Figure 20 Exemple de réunion d'octets.

Le processus de simplification au complet

Nous venons de voir comment la réunion de doublets, de quartets et d'octets de 1 adjacents dans un diagramme K aboutit à une expression simplifiée. Il doit être clair que plus une réunion regroupe de 1, plus le nombre de variables éliminées est grand. Plus précisément encore, une réunion de deux 1 provoque l'élimination d'une variable, une réunion de quatre 1, l'élimination de deux variables et une réunion de huit 1, l'élimination de trois variables. Voici les étapes à suivre pour simplifier une expression booléenne en recourant à la méthode des diagrammes de Karnaugh:

1. Dessinez le diagramme K et placez des 1 dans les carrés correspondant aux lignes de la table de vérité dont la sortie est 1. Mettez des 0 dans les autres carrés.
2. Etudiez le diagramme et repérez les 1 adjacents. Encernez les 1, dit isolés qui ne font parties que d'un seul groupe. Pour ces 1, il n'existe qu'une seule possibilité de groupement.
3. Ensuite continuer à prendre les groupes les plus grands qui incluent des 1 qui ne font pas partie d'un autre groupe.
4. Vous devez prendre tous les 1 de la table de Karnaugh. Il est nécessaire d'utiliser plusieurs fois le même 1, si cela permet la création d'un groupe plus grand.
5. Effectuez l'addition logique de tous les termes résultant des réunions.

Les conditions indifférentes

Certains circuits logiques peuvent être conçus certaines conditions d'entrée ne correspondent à aucun niveau de sortie particulier principalement parce que ces conditions ne doivent jamais survenir. En d'autres mots, il certaines combinaisons des niveaux d'entrée pour lesquels il nous importe peu que la sortie soit HAUTE ou BASSE. Une illustration de ce qu'on veut dire est donnée dans la table de vérité de la Figure 21a).

Dans cette table aucune valeur de Z ne figure pour les conditions $ABC = 100$ et $ABC = 011$. Au contraire, on a mis un x. Ce x signifie condition indifférente. Plusieurs raisons peuvent expliquer la présence de conditions indifférentes, la plus courante étant que dans certaines situations ces combinaisons d'entrée ne peuvent jamais survenir; par conséquent, il est inutile de préciser pour elles une valeur de sortie.

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

\bar{C}	C
$\bar{A}\bar{B}$	0
$\bar{A}B$	0
$A\bar{B}$	1
AB	x

\bar{C}	C
$\bar{A}\bar{B}$	0
$\bar{A}B$	0
$A\bar{B}$	1
AB	1

$Z=A$

Figure 21 Exemple d'application de Karnaugh avec des états indifférents

En présence de conditions indifférentes, nous devons décider quel x de sortie est remplacé par un 0 et quel x est remplacé par un 1 en vue de réunir de la façon la plus efficace les 1 adjacents du diagramme K (pour obtenir l'expression la plus simple).

Circuits OU EXCLUSIF (XOR) et NI EXCLUSIF (XNOR)

Deux circuits logiques spéciaux qui interviennent souvent dans les systèmes numériques: le circuit OU exclusif et le circuit NI exclusif.

OU exclusif (XOR)

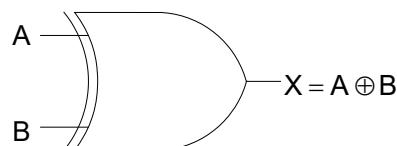
La sortie d'une porte OU exclusif est au niveau haut seulement lorsque les deux entrées sont à des niveaux logiques différents. Une porte OU exclusif n'a toujours que deux entrées. On veut dire par là qu'il n'existe pas de portes OU exclusif à trois ou quatre entrées. Ces deux entrées sont combinées pour que $X = \overline{AB} + A\overline{B}$. On abrège cette expression ainsi:

$$X = A \oplus B$$

On peut voir sur la Figure 22 la table de vérité ainsi que le symbole d'une porte OU exclusif.

A	B	$X = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

a)



b) porte OU exclusif (XOR)

Figure 22 a) Table de vérité définissant l'opération OU exclusif; b) symbole d'une porte OU exclusif.

NI exclusif (XNOR)

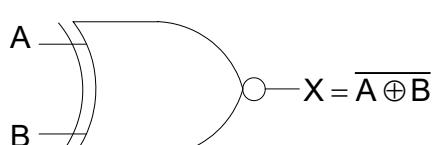
Le circuit Ni exclusif a un fonctionnement exactement opposé à celui du OU exclusif. La sortie d'une porte NI exclusif est au niveau haut seulement lorsque les deux entrées sont à des niveaux logiques identiques. On peut voir à la

Figure 23 sa table de vérité ainsi que son symbole logique. L'expression de ce dernier est : $X = \overline{A}\overline{B} + AB$. On abrège cette expression ainsi:

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

A	B	$X = \overline{A} \oplus \overline{B}$
0	0	1
0	1	0
1	0	0
1	1	1

a)



b) porte NI exclusif (XNOR)

Figure 23 a) Table de vérité définissant l'opération NI exclusif; b) symbole d'une porte NI exclusif.

Fonctions combinatoires standards

Les circuits MSI du commerce (circuit logiques de moyenne intégration) contiennent des fonctions plus complexes que les simples portes logiques. Ces fonctions présentent l'avantage d'être moins onéreuses sous la forme d'un circuit. Nous aurons tout intérêt à les utiliser de préférence à l'équivalent en portes.

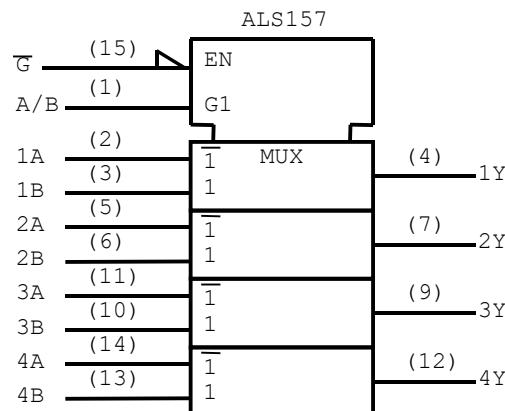
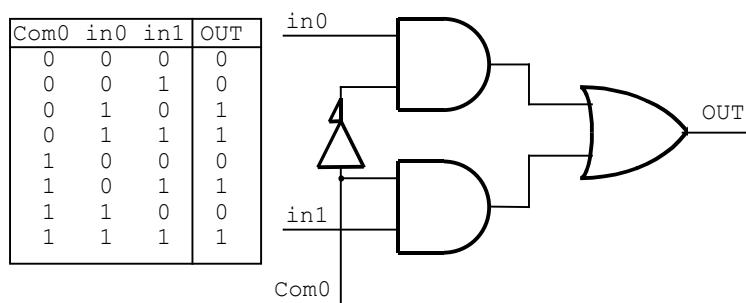
Les principales fonctions à disposition sont :

- Le multiplexage (MUX) 2->1, 4->1, 8->1 et 16->1
- Le décodage (X/Y) 1->2, 1->4, 1->8
- L'encodeur de priorité
- La comparaison (COMP) <,=,>
- Les opérations arithmétiques (addition, soustraction, ...)
- Le transcodage de nombres : BIN->BCD, BCD->BIN, BCD->7SEG, etc.
- ...

D'autre part, il sera très important d'identifier ces fonctions standards pour la décomposition de systèmes combinatoires complexes.

Multiplexeur (MUX)

Un multiplexeur est un système combinatoire qui met sur sa sortie unique la valeur d'une de ses 2^n entrées de données, le numéro de l'entrée sélectionnée étant fourni sur les n entrées de commande.



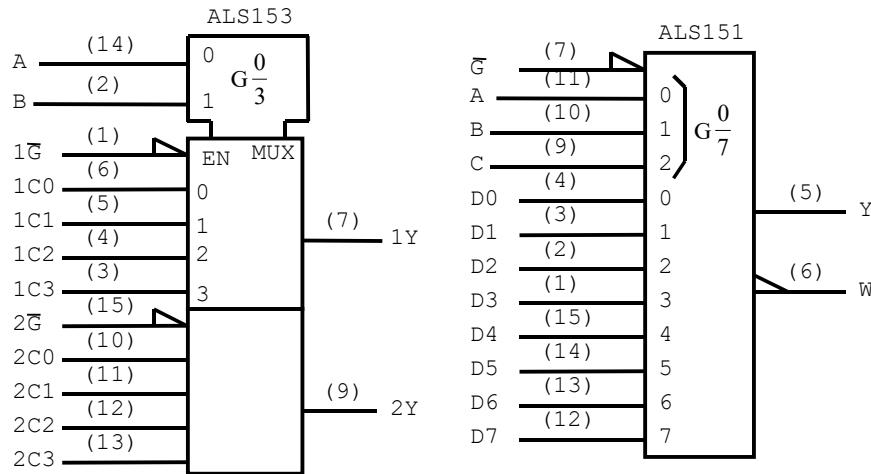
Nous donnons la table de vérité d'un multiplexeur 2 vers 1 (2 to 1), son schéma est celui du circuit standard qui remplit cette fonction. Remarquez que nous avons 4 multiplexeurs avec le même bloc de commande dans le circuit MSI standard (74ALS157). Les multiplexeurs sont symbolisés par les rectangles superposés, le premier est marqué MUX.

Le bloc de commande se marque par un rectangle ayant des encoches qui coiffe les 4 blocs. Une entrée EN (ENable) supplémentaire apparaît. La sortie des multiplexeurs n'est active que si l'entrée EN=1, ce qui revient à dire que la borne 15 est au niveau bas.

Dans le bloc de multiplexage, l'entrée marquée 1 indique que la sortie prendra la valeur de B si l'entrée de sélection G1, borne 1, prend la valeur 1 (niveau haut). L'entrée A sera prise en compte pour G1=0.

Les chiffres entre parenthèses indiquent les numéros des bornes du circuit.

Voici le schéma de quelques multiplexeurs courants :



Décodeur (X/Y)

Un décodeur permet de d'identifier quelle combinaison est active. Il comporte n entrées de commande (sélection), une entrée de validation, et 2ⁿ sorties. La sortie dont le numéro est donné sur les entrées de commandes sera activée si l'entrée de validation est active. Les autres sorties sont inactives, généralement au niveau haut.

Voici un décodeur 3 to 8 avec 3 bits de sélection. Il permet de décoder les 8 combinaisons d'un vecteur 3 bits. Le 74HC138 (3 to 8 line decoders/demultiplexers) est un circuit intégré dont une seule sortie sur 8 peut être active en même temps (sortie active à l'état bas). Voici son symbole logique et sa table de vérité :

G1	G2A	G2B	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	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	1	1	1	1	1	1	0	1
0	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
x	x	1	x	x	x	1	1	1	1	1	1	1	1

À gauche, le symbole logique du 74HC138 montre les entrées G1, G2A, G2B, C, B, A et les sorties Y0 à Y7. Les entrées G2A et G2B sont marquées avec une barre sur le dessus, indiquant qu'il s'agit d'entrées actives à basse tension.

Dans le circuit 74HC138, l'activation des sorties dépend d'un signal enable, qui est le résultat d'un ET entre les entrées (G1, G2A' et G2B').
 $\text{enable} = G1 \cdot G2A' \cdot G2B'$

Additionneur binaire parallèle

Les ordinateurs ne peuvent additionner que deux nombres binaires à la fois, chacun de ces nombres pouvant avoir plusieurs bits. La Figure 24 illustre l'addition de deux nombres de 5 bits.

cumulande	1	0	1	0	1	Mémorisé dans l'accumulateur
cumulateur	0	0	1	1	1	Mémorisé dans le registre B
+						
somme	1	1	1	0	0	
report	0	0	1	1	1	

Figure 24 Addition binaire caractéristique

On commence l'addition en additionnant les bits de poids faible du cumulande et du cumulateur. Donc $1 + 1 = 10$, ce qui signifie que la somme pour ce rang est 0 avec un report de 1. Ce report est ajouté au bit du cumulande et du cumulateur du rang immédiatement à gauche. Ainsi l'addition au deuxième rang est $1 + 0 + 1 = 10$, ce qui, à nouveau produit une somme de 0 et un report de 1. Ce report est ajouté au bit du cumulateur et du cumulande du rang immédiatement à gauche et ainsi de suite pour les autres rangs.

À chaque étape de l'addition, nous additionnons 3 bits: le bit du cumulande, le bit du cumulateur et le bit de report provenant de l'addition des chiffres du rang précédent. Le résultat de l'addition de ces 3 bits est un nombre à 2 bits: le bit de somme et le bit de report, ce dernier devant être ajouté au rang immédiatement à gauche. Vous devez bien saisir que le même processus se répète pour tous les rangs du nombre. Donc, si on parvient à concevoir un circuit logique qui reproduit l'opération d'addition, il nous restera alors simplement à accolter des circuits identiques, un pour chaque rang binaire. C'est ce que montre la figure 25.

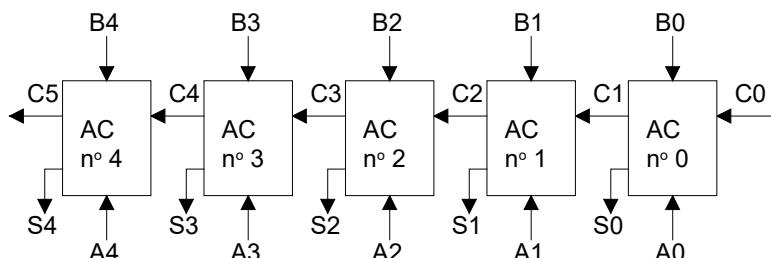


Figure 25 Schéma fonctionnel d'un circuit additionneur parallèle de 5 bits formé de 5 additionneurs complets.

Sur cette figure, les variables A_4, A_3, A_2, A_1 et A_0 représentent les bits du cumulande mémorisés dans l'accumulateur (appelé également registre A). Les variables B_4, B_3, B_2, B_1 et B_0 sont les bits du cumulateur mémorisés dans le registre B. Les variables C_4, C_3, C_2, C_1 et C_0 représentent les bits de report des rangs correspondants. Les variables S_4, S_3, S_2, S_1 et S_0 sont les bits de somme de chaque rang. Les bits correspondant du cumulateur et du cumulande sont appliqués à un circuit logique appelé additionneur complet, de même que le bit de report généré par l'addition des bits du rang précédent.

L'additionneur complet utilisé pour chaque rang a trois entrées: une entrée A, une entrée B et une entrée C, et deux sorties: une sortie somme et une sortie report. Sur cette figure, on additionne des nombres de 5 bits; dans les ordinateurs d'aujourd'hui les nombres s'échelonnent généralement de 8 à 128 bits.

Le montage de la figure 25 est appelé un additionneur parallèle parce que tous les bits du cumulateur et du cumulande sont appliqués et additionnés simultanément. Donc les additions des bits de chacun des rangs se font en même temps. Il s'agit là d'une façon qui diffère de la technique manuelle, dans laquelle on additionne chaque rang un à la fois en partant du bit de poids faible. De toute évidence l'addition parallèle est extrêmement rapide.

Conception d'un additionneur complet

Maintenant que nous savons qu'elle est la fonction d'un additionneur complet, concevons un circuit logique qui effectue cette fonction. En premier lieu, nous devons construire une table de vérité énumérant toutes les combinaisons possibles pour les diverses valeurs d'entrée et de sortie. On peut voir cette table de vérité à la Figure 26 dans laquelle se trouvent les trois entrées A, B et C_{en} et les deux sorties S et C_s . À trois entrées correspondent 8 combinaisons possibles; pour chaque combinaison, on donne dans la table les valeurs de sortie recherchées. Comme il y a deux sorties, nous allons concevoir indépendamment les circuits de chacune des sorties, en commençant par celui de la sortie S.

A	B	C_{en}	S	C_s
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

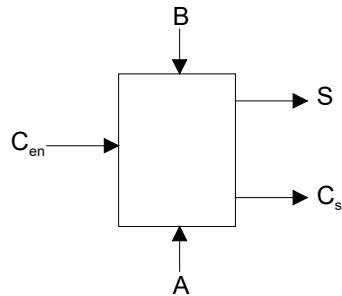


Figure 26 Table de vérité d'un additionneur complet.

La table de vérité démontre qu'il y a 4 cas où S doit être 1. En recourant à la méthode des sommes de produits, nous pouvons écrire ainsi l'expression pour S:

$$S = \overline{A}\overline{B}C_{en} + \overline{A}\overline{B}\overline{C}_{en} + A\overline{B}C_{en} + AC_{en}$$

Une fois simplifié par la méthode algébrique, on obtient :

$$S = (A \oplus B) \oplus C_{en} \quad (6-2)$$

Maintenant intéressons-nous à la sortie C_s de la table de vérité, l'expression de la somme de produits correspondant à C_s est :

$$C_s = \overline{A}\overline{B}C_{en} + \overline{A}\overline{B}\overline{C}_{en} + AB\overline{C}_{en} + ABC_{en} = \overline{A}\overline{B}C_{en} + A\overline{B}C_{en} + AB$$

Une fois simplifié par la méthode algébrique, on obtient :

$$C_s = (A \oplus B)C_{en} + AB \quad (6-3)$$

Les expressions (6-2) et (6-3) sont matérialisées par le circuit de la Figure 27, qui représente un additionneur complet.

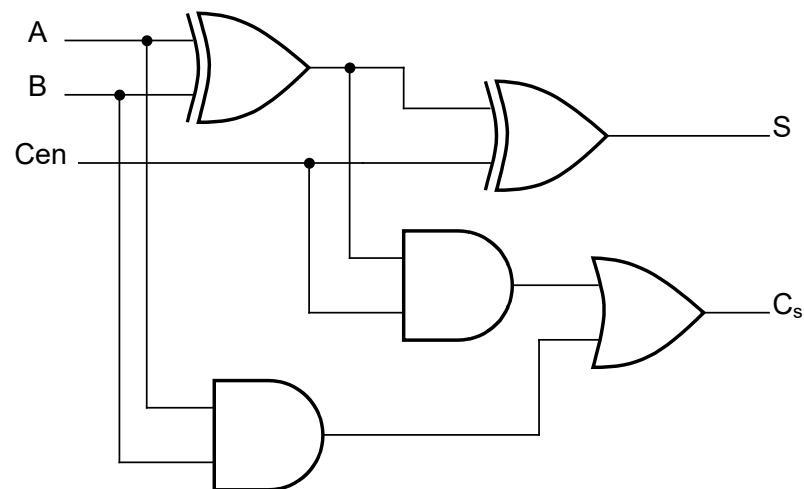


Figure 27 Ensemble des circuits d'un additionneur complet

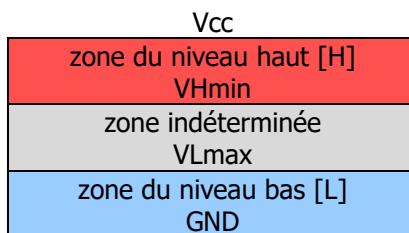
Aspects techniques circuits logiques combinatoires

Technologie

Dans les chapitres précédents, nous avons abordé la conception des circuits logiques sans tenir compte de leurs modes de matérialisation. Il est bon de garder en mémoire que les circuits électroniques numériques sont régis par les lois de l'électronique. Ce chapitre ne traitera que des composants pour les systèmes combinatoires. Les éléments séquentiels seront vus ultérieurement.

La représentation des états logiques.

Comme nous n'avons à représenter que deux états logiques, plutôt que de leur associer à chacun une tension donnée, nous leur associerons une plage de tensions. Ce mode de représentation présente l'avantage de ne pas imposer l'établissement d'une tension précise et permet même d'envisager la superposition d'un bruit sur cette tension sans quitter la plage significative.



Les familles logiques

Il existe une multitude de familles logiques. Citons les plus anciennes dans leur ordre d'apparition sur le marché.

Famille	date	type
RTL	1964	Resistor Transistor Logic
DTL	1964	Diode Transistor Logic
TTL	1969	Transistor Transistor Logic

La famille TTL comporte plus de 800 types de circuits différents. Pour faciliter son implantation (augmentation du degré d'intégration, niveau de tension, vitesse, consommation), elle a donné naissance à une série de sous-familles.

Dès 1976, une nouvelle technologie apparaît (MOS complémentaire). Elle porte le nom de CMOS (Complementary Metal Oxide Semiconductor). Comme la famille TTL, l'évolution des technologies conduit à la création de nouvelles sous-familles.

Il faut remarquer la compatibilité des numéros des circuits CMOS avec ceux de la famille TTL. Deux sous-familles CMOS acceptent des tensions d'alimentation différentes de la tension normale (5 volts).

Familles TTL

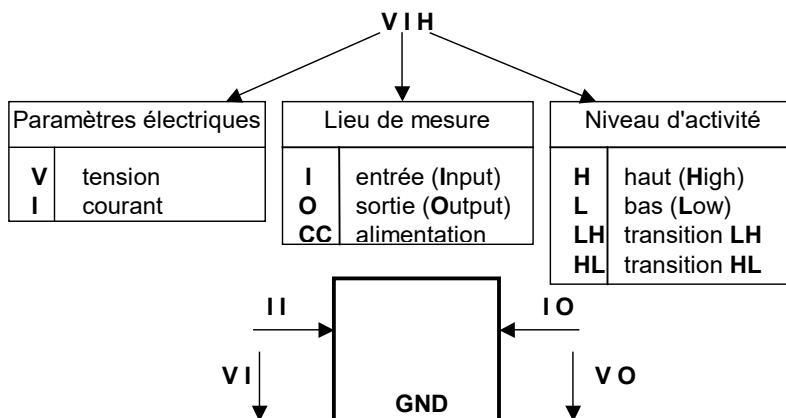
Série	commentaire	consommation (mW)	vitesse (ns)	usage
74	standard	10	10	dépassé
74H..	H igh speed	20	5	dépassé
74L..	L ow power	1	30	dépassé
74S..	S chottky	20	3	dépassé
74AS..	A dvanced S chottky	8	2	dépassé
74LS..	L ow power S chottky	2	10	normal
74ALS	A dvanced L S	2	4	conseillé
74F..	F ast	4	3	ponctuel

Famille CMOS

Série	commentaire	consommation (mW)	vitesse (ns)	usage
4000	alimentation de 3...8 V	0	100	dépassé
45..	alimentation de 3...8 V	0	100	normal
74C..	broche compatible TTL	0	50	dépassé
74HC..	High speed CMOC	0	10	conseillé
74HCT..	HC à niveau compatible TTL	0	10	conseillé
74AC..	Advanced CMOS	0	3	nouveau
74ACT..	AC à niveau compatible TTL	0	3	nouveau

Terminologie des circuits numériques

Pour faciliter la description des caractéristiques électriques des circuits logiques, une convention d'écriture a été adoptée par les fabricants.



Dans ce schéma, les sens des courants sont ceux des courants positifs. Les courants entrants sont positifs.

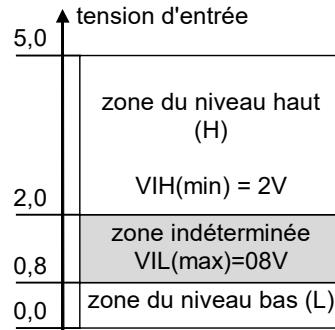
Définition de la terminologie courante

ICC = courant d'alimentation
 ICCH = courant d'alimentation pour toutes les sorties au niveau haut
 ICCL = courant d'alimentation pour toutes les sorties au niveau bas
 IIH = courant d'entrée au niveau haut
 IIL = courant d'entrée au niveau bas
 IOH = courant de sortie au niveau haut
 IOL = courant de sortie au niveau bas
 IOS = courant de court-circuit (sortie à la masse)

VCC = tension d'alimentation pour le circuit TTL
 VDD = tension d'alimentation pour le circuit CMOS
 VIH = tension d'entrée au niveau haut
 VIL = tension d'entrée au niveau bas
 VOH = tension de sortie au niveau haut
 VOL = tension de sortie au niveau bas

Tensions d'entrée

Les tensions appliquées aux bornes des entrées des circuits intégrés numériques doivent appartenir aux zones de tensions admissibles. Pour un circuit de type 74ALS00, une entrée recevant une tension comprise entre 0 et 0,8 V sera considérée au niveau logique 0. Une entrée recevant une tension entre 2 et 5V sera considérée comme une entrée à 1.



Il faut noter que les valeurs de VIL (max) et VIH (min) dépendent des technologies utilisées.

tension d'entrée	série TTL	74HCT...	74HC..
VIH (min) (V)	2,0	2,0	3,5
VIL (max) (V)	0,8	0,8	1,0

Tensions de sortie

Il en est de même pour les tensions de sortie qui seront dépendantes des technologies. La valeur réelle de la tension dépendant de la charge, nous ne pouvons définir que des extrêmes.

tension de sortie	74LS..	74ALS..	74HCT...	74HC..
VOH (min) (V)	2,4	3,0	4,5	4,5
VOL (max) (V)	0,5	0,5	0,5	0,5

Courant d'entrée

Dans les séries TTL, le courant d'entrée au niveau bas est sortant. De ce fait, il prendra une valeur négative.

courant d'entrée	74LS..	74ALS..	74HCT...*	74HC..*
IIH (max) (μ A)	20	20	1	1
IIL (max) (mA)	-0,4	-0,1	0	0

* VCC = 5V

Courant de sortie

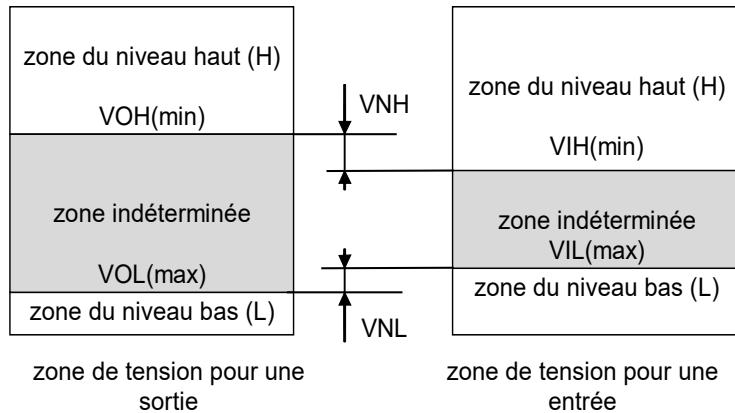
Le courant au niveau haut étant sortant sera négatif.

courant de sortie	74LS..	74ALS..	74HCT...*	74HC..*
IOH (max) (mA)	-0,4	-0,4	-4	-24
IOL (max) (mA)	8	8	4	4

* VCC = 5V

Immunité au bruit

Le bruit est un signal parasite induit dans le circuit qui vient se superposer au signal transmis. L'immunité au bruit est la tolérance d'amplitude que supporte le circuit pour identifier encore correctement les signaux.

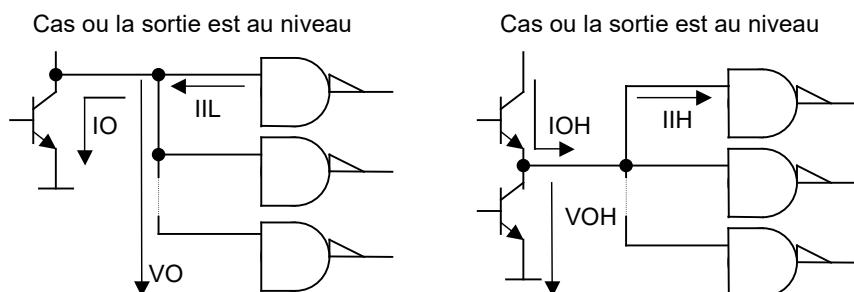


La marge de sensibilité aux bruits au niveau haut V_{NH} (Noise High) est définie comme suit : $V_{NH} = V_{OH}(\min) - V_{IH}(\min)$

De même, au niveau bas, nous définirons $V_{NL} = V_{IL}(\max) - V_{OI}(\max)$

Facteur de charge

Une porte logique ne peut pas admettre un nombre illimité de portes connectées sur sa sortie. Prenons le cas d'une porte 74ALS00 que nous supposons chargée par des portes du même type. Nous examinerons deux cas.



Pour la sortie au niveau bas, nous aurons $IOL = n \cdot IIL$ si n est le nombre de portes. Comme $IOL < IOL(\max) = 8 \text{ mA}$, et que $IIL < IIL(\max) = -0,1 \text{ mA}$, on peut tirer une valeur maximum de $n = 8/0,1 = 80$.

Pour la sortie au niveau haut, nous avons $IOH = n \cdot IIH$, ce qui donne pour les valeurs $IOH = -0,4 \text{ mA}$ et $IIH = 20 \text{ mA}$ $n = 400/20 = 20$.

Nous dirons que la porte a une sortance de 20. Pour simplifier les calculs, les fabricants ont décidé d'utiliser une charge unitaire correspondant à :

40 μA dans l'état haut et 1,6 mA dans l'état bas,

Ces valeurs correspondent au courant d'entrée pour la série TTL standard. À l'état haut, $IIH(\max) = 40 \mu\text{A}$ représente le courant maximum qui circule dans une entrée TTL standard. De même, à l'état bas, $IIL(\max) = 1,6 \text{ mA}$ représente le courant maximum dans l'entrée à l'état bas. Bien que ces caractéristiques correspondent à celles de la série TTL standard, nous les utiliserons pour exprimer les exigences d'entrée/sortie des autres séries.

Série TTL	Entrance (UL)		Sortance (UL)	
	haut	bas	haut	bas
7400	1	1	10	10
74H00	1,25	1,25	12,5	12,5
74L00	0,5	0,1	10	2,5
74S00	1,25	1,25	25	12,5
74LS00	0,5	0,25	10	5
74ALS00	0,5	0,06	10	5

Exemple de calcul de la sortance d'un 74ALS00

$$\text{Sortance basse} = I_{OL} (\text{max}) / 1,6 \text{ mA} = 8 / 1,6 = 5$$

$$\text{Sortance haute} = I_{OH} (\text{max}) / 0,04 \text{ mA} = 0,4 / 0,04 = 10$$

Remarque : La plupart des familles ont des sortances hautes et basses différentes. Lors de la conception des systèmes, nous prendrons en compte la valeur la plus défavorable (la plus faible).

Pour les circuits CMOS, la résistance d'entrée extrêmement élevée fait que ces circuits ont dans leurs familles une sortance très grande (supérieure à 50).

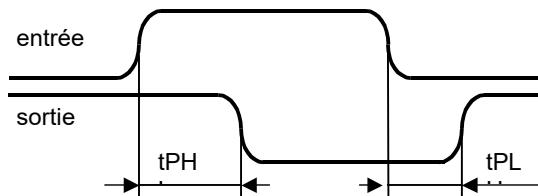
Les caractéristiques temporelles

tpd = temps de commutation (tpd = tPHL ou tPLH)

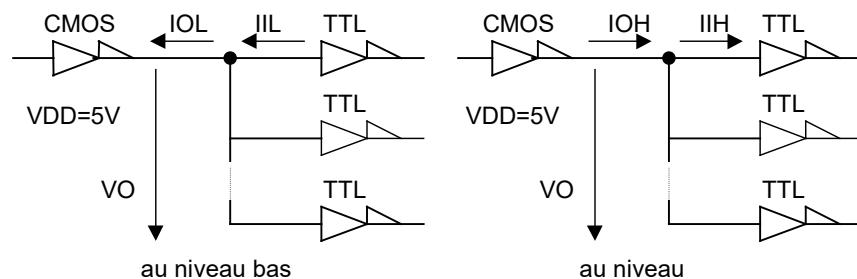
tPHL = temps de commutation du niveau haut au niveau bas

tPLH = temps de commutation du niveau bas au niveau haut

Le sens de la commutation donné dans le nom est toujours celui de la sortie. Le niveau de la tension au point de mesure dépend de la famille logique (ALS 1,3V).



Interface CMOS - TTL



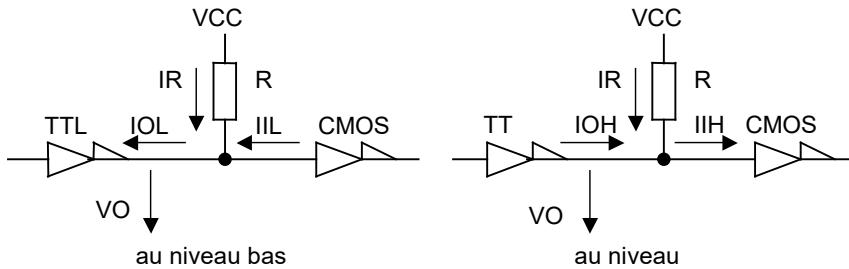
Au niveau bas, il nous faudra : n IIL (max) < IOL (max) et VOL < VIL

Au niveau haut, il nous faudra : n IIH < IOH et VOH > VIH

Interface TTL - CMOS

La tension de sortie du niveau haut de 2,4 V des circuits TTL n'est pas compatible avec celle d'entrée des circuits CMOS (3,5 V).

L'emploi d'une résistance de polarisation contre VCC peut résoudre notre problème



Au niveau haut $IIH = 1 \mu A$, $IOH (\text{max}) = 400 \mu A$, nous aurons pratiquement $IIH = IOH$ et le courant dans R étant pratiquement nul, la plus petite valeur de R nous conviendra.

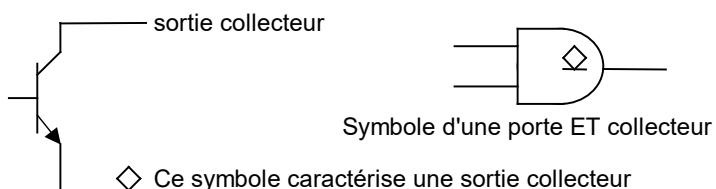
Au niveau bas, $IIL = 0$, $IOL (\text{max}) = 8 mA$, $VO < VOL (\text{max})$

$$\begin{aligned} (VCC - VOL) &= R * IOL \\ VCC - R * IOL &= VOL < VOL (\text{max}) \\ VCC - VOL(\text{max}) &< R * IOL \\ (VCC - VOL (\text{max})) / IOL &< R \end{aligned}$$

Une autre solution consiste à remplacer le circuit 74HC par un 74 HCT dont les niveaux d'entrées sont compatibles.

Collecteur ouvert (open collector)

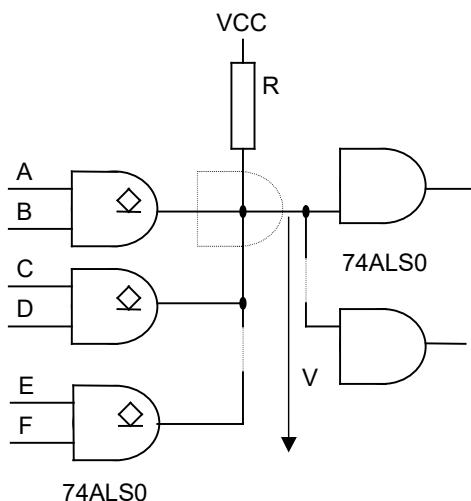
Pour faciliter la réalisation d'interfaces entre l'électronique numérique et l'électronique analogique (par exemple diodes luminescentes) ou des éléments électromécaniques (relais), les utilisateurs demandaient la possibilité de disposer de courants de sortie plus importants. Pour offrir cette performance, les fabricants ont proposé des circuits dont l'étage de sortie est un transistor collecteur ouvert, ce qui permet à l'utilisateur de disposer de la totalité du courant débité par le transistor.



Le circuit collecteur ouvert utilisera une charge extérieure déterminée par le concepteur. Cette architecture offre la possibilité de coupler plusieurs portes collecteur ouvert sur la même charge et de réaliser ainsi un opérateur câblé. Le dimensionnement de la résistance de charge dépendra du nombre de circuits connectés en entrée et en sortie.

La correspondance au collecteur ouvert en technologie MOS se nomme le drain ouvert (open drain).

	74LS00	74LS01
VOH (min) (V)	2.4	2.4(TTL)
VOL (max) (V)	0.4	0.4
VIH (min) (V)	2	2
VIL (max) (V)	0.8	0.8
IOL (max) (mA)	8	8
IOH (max) (mA)	-0.4	0.1
IIL (max) (mA)	-0.4	-0.4
IIH (max) (μA)	20	20



La porte en pointillé marque une porte virtuelle réalisée par le câblage. Ce type de portes présente l'avantage d'avoir ses entrées/sorties sur un même conducteur, ce qui est particulièrement avantageux si ces points sont dispersés sur différentes cartes. Une seule borne d'interconnexion sera nécessaire entre ces cartes. Le dimensionnement de Rp est possible en écrivant les contraintes $V < VOL (\text{max})$ au niveau bas, et $V > VOH (\text{min})$ au niveau haut.

Dans le cas de l'exemple, le calcul donne :

$$\begin{aligned}
 V &= VCC - Rp \times ((3 \times IOH) + (2 \times IIH)) > VOH (\text{min}) \\
 Rp &< (VCC - VOH (\text{min})) / ((3 \times (-IOH (\text{max}))) + (2 \times IIH (\text{max}))) \\
 Rp &< (5 - 2,4) / ((3 \times (0,1)) + (2 \times (0,02))) = (2,6) / (0,3 + 0,04) = 2,6 / 0,34 = 7,6 \text{ K}\Omega
 \end{aligned}$$

Une porte au niveau bas doit garantir le niveau bas, ce qui explique le $(1 \times IOL)$ et $(2 \times IOH)$.

$$\begin{aligned}
 V &= VCC - Rp \times ((1 \times IOL) + (2 \times IOH) + (2 \times IIL)) < VOL (\text{max}) \\
 Rp &> (VCC - VOL (\text{max})) / ((IOL) + (2 \times IOH) + (2 \times IIL)) \\
 Rp &> (5 - 0,4) / ((8) + (2 \times 0,1) + (2 \times (-0,1))) = 4,6 / (8 + 0,2 - 0,2) = 4,6 / 8 = 575 \Omega
 \end{aligned}$$

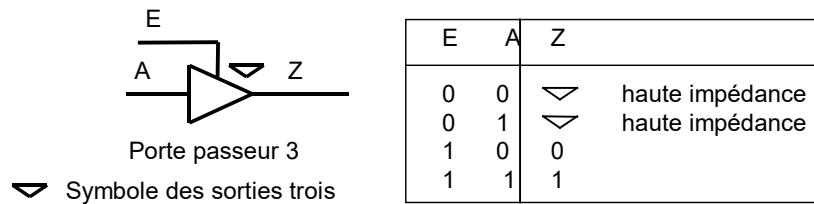
En conclusion : $575 \Omega < Rp < 7600 \Omega$

Pratiquement, bien que la valeur de Rp dépende de la configuration des portes, une valeur de Rp entre 0,8 et 1 KΩ donne un résultat satisfaisant.

Porte trois états

Une porte trois états est un circuit dont on se sert pour contrôler le passage d'un signal logique. Il comporte trois états de sortie (haut, bas et haute impédance). Ce nouvel état dit 'haute impédance' ne fait que rendre flottante la ligne de sortie.

Ce type de circuits est abondamment utilisé dans les processeurs pour permettre la circulation bidirectionnelle de l'information.



Le langage de description VHDL

Introduction

VHDL, langage complet de description de matériel, a trouvé ses premières applications dans la modélisation et la simulation. Son utilisation pour la synthèse logique, autrement dit la création de circuits à partir de descriptions textuelles, est plus récente, liée notamment à l'essor extraordinaire des composants logiques programmables, qu'ils se nomment CPLD ou FPGA. Il fallait à ces architectures devenues complexes un langage descriptif de haut niveau, en remplacement des langages de première génération aux fonctionnalités limitées. Il leur fallait également un langage non-propriétaire, ne verrouillant pas l'utilisateur à une architecture unique. Il leur fallait enfin un langage "démocratique", apte à promouvoir des environnements de développements économiques. Toutes ces conditions sont aujourd'hui remplies, et le fait est là : VHDL est devenu le langage descriptif préférentiel des nouveaux utilisateurs.

L'invention du langage VHDL et de la simulation ont été une innovation importante, l'utilité et la popularité du VHDL ont réellement décollé avec l'apparition des outils de synthèse VHDL ainsi que l'essor extraordinaire des circuits logiques programmables haute densité (CPLD et FPGA). Ces programmes peuvent créer la structure d'un circuit logique directement à partir d'une description comportementale (behavioral) en VHDL. En utilisant VHDL, on peut concevoir, simuler et synthétiser n'importe quel circuit logique, d'un simple circuit combinatoire, au dernier microprocesseur de chez Intel.

Design Flow

Il est très utile de comprendre la totalité du processus de création d'un circuit logique, avant de se lancer dans le langage lui-même. Il y a plusieurs étapes dans le processus d'un design en VHDL, souvent appelé design flow, ces étapes sont visibles sur la Figure 28.

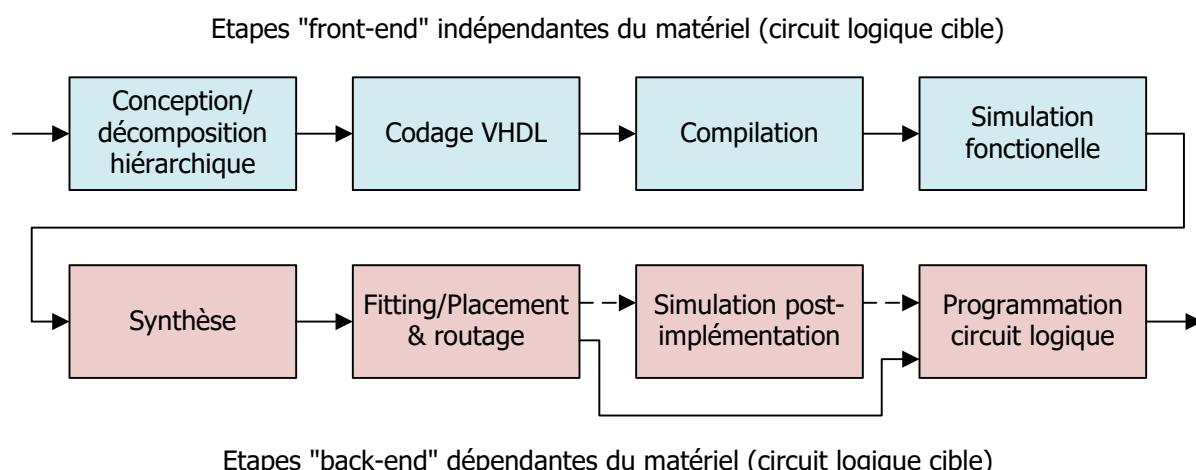


Figure 28 Les étapes de la conception d'un système numérique (design flow).

La conception, décomposition hiérarchique

La première étape consiste à concevoir la structure complète du circuit en le décomposant de façon hiérarchique en créant des schémas bloc utilisant des blocs les plus simples possibles. VHDL permet d'écrire une description pour chacun de ces blocs et de les connecter entre eux. C'est l'étape la plus importante du processus, car c'est elle qui déterminera le bon fonctionnement du circuit ainsi que le temps nécessaire pour écrire le code VHDL, qui est donc la deuxième étape du processus.

Le codage VHDL

Si l'étape précédente, consistant à décomposer le système en plusieurs niveaux hiérarchiques, jusqu'à obtenir de blocs simple à décrire en VHDL (compteurs, machines d'état, bloc combinatoire...), le codage se résume à la partie la plus simple du travail. VHDL est un langage texte qu'on peu écrire sur n'importe quel

éditeur, cependant la plupart des environnements de conception incluent un éditeur spécialisé qui simplifie le codage.

La Compilation

Une fois que vous avez écrit votre code, vous devrez le compiler. Le compilateur analyse les erreurs de syntaxe, et contrôle l'interconnexion avec les autres modules. Les fichiers compilés seront donc utilisés par le simulateur. Comme dans n'importe quel langage, il n'est pas nécessaire d'avoir écrit la totalité de votre code avant de compiler, ceci facilitera la correction des erreurs (principalement de la syntaxe) dans votre code.

La simulation fonctionnelle

La simulation est aussi une des étapes essentielles du processus, car c'est elle qui permet de « débuguer » et valider votre design. Un simulateur VHDL vous permet de définir et appliquer des signaux sur les entrées de votre circuit, et contrôler automatiquement les états attendus sur les sorties, sans avoir à réaliser un circuit physique. Ces fichiers également en langage VHDL sont appelés testbench ou bancs de test. Cette simulation est de type fonctionnelle, c'est à dire qu'on ne tient pas compte des temps de propagation qui entreront en vigueur dans le circuit final, d'où l'importance de réaliser des designs synchrones, dont le comportement ne changera pas une fois le circuit implémenté. Pour éviter ces désagréments, on peut faire une nouvelle simulation une fois le circuit routé (cf. simulation post-implémentation), et cette simulation prendra en compte les temps de propagation.

La Synthèse

Une fois la vérification terminée, on peut réaliser une synthèse logique de notre description VHDL, ce qui produira une liste d'interconnexions de composants primitifs du circuit cible généralement appelée netlist, qui bien entendu dépend de la technologie sur laquelle nous allons planter notre circuit (CPLD, FPGA, ASIC...). Cette netlist peut être comparée à un schéma logique composé de portes et de bascules synchrones.

L'implémentation (Fitter ou Place & Route)

Une fois la synthèse terminée, un outil (Fitter pour les CPLD ou Place & Route pour les FPGA) va planter notre design à l'intérieur du circuit cible (choisi avant la synthèse), grâce à la netlist générée par le synthétiseur. En fait le logiciel va faire des connexions entre les différents composants primitifs du circuit logique cible, afin d'implémenter la fonction logique décrite précédemment.

La simulation post-implémentation

Comme mentionné précédemment, l'étape finale est la vérification du timing à partir du circuit routé. Car c'est seulement à partir de cet instant que le programme arrive à calculer assez précisément les temps de propagation du à la longueur des connexions, des charges électriques, ainsi que d'autres facteurs qui influencent la vitesse. Cette étape n'est pas toujours effectuée, car si le design est conçu de façon correcte, donc synchrone, aucun problème de timing ne doit se produire. Cependant si on effectue cette simulation, on utilisera en général le même banc de test que pour le test fonctionnel.

Programmation du circuit logique

Bien entendu il reste à programmer le circuit logique pour y planter physiquement notre design. Chaque bit du fichier de programmation, correspond à une connexion faite ou non dans le circuit logique programmable. La programmation peut se faire directement sur le circuit cible, qui garde sa configuration même si l'on coupe l'alimentation (CPLD flash), ou alors dans une mémoire non volatile externe (flash) qui doit reconfigurer le circuit logique programmable à chaque démarrage de l'alimentation (FPGA SRAM). Il existe encore d'autres méthodes, consistant à « brûler » des fusibles pour chaque connexion dans le circuit logique programmable, cependant, il ne peut plus être reprogrammé avec un nouveau design.

Les avantages du VHDL

Langage complet

Dès l'origine l'un des objectifs majeurs de VHDL fut de couvrir les différentes étapes de la conception de systèmes numériques : spécifications, modélisation, simulation, synthèse. Un langage unique pour ces différentes phases, notamment simulation et synthèse, c'est un gain de temps, de qualité et de fiabilité : un seul langage à apprendre, pas de changement d'environnement, pas de traduction plus ou moins automatique, pas de passerelle pour passer d'un mode à l'autre.

Langage indépendant

Standard IEEE, VHDL n'appartient à personne en particulier, qui aurait la mainmise sur son évolution et son utilisation. Son indépendance vis-à-vis des architectures et des technologies est totale (qu'on nous entende bien, il n'est pas question d'affirmer qu'une description VHDL est optimum quelle que soit l'architecture ou la technologie ciblée, mais que l'emploi de VHDL n'est pas réservé à une certaine catégorie d'architectures, ou certaines technologies). Indépendance enfin vis-à-vis du système hôte : PC ou station de travail, même combat.

Langage flexible

VHDL permet à l'utilisateur d'effectuer ses descriptions en se plaçant à différents niveaux d'abstraction : niveau comportemental, le plus clair, le plus concis et le plus compréhensible, niveau intermédiaire avec utilisation d'équations logiques par exemple, niveau "composant" avec interconnexion de portes logiques élémentaires.

Flexibilité aussi dans l'assemblage des descriptions : avec ou sans hiérarchie, c'est selon le souhait de l'utilisateur.

Langage moderne

Dans la famille "langage de programmation" VHDL est le cousin d'Ada et le neveu de Pascal... Sa syntaxe est particulièrement lisible, et permet fréquemment de faire l'économie de commentaires. Langage fortement typé et contrôlé, il minimise les risques d'erreur et leur propagation insidieuse, et favorise la qualité. Ses mécanismes de partitionnement et d'empaquetage permettent la conception modulaire, l'archivage, et la réutilisation.

Langage standard

C'est un gage de compatibilité, de portabilité, de stabilité et de pérennité. Pour les fournisseurs d'outils logiciels c'est la garantie d'un vaste marché et de rentabilité. Pour les utilisateurs c'est un éventail de choix et un investissement durable.

Langage ouvert

Un langage standardisé il y a dix ans n'aurait aucune chance de durer s'il n'évoluait pas. VHDL est donc un langage ouvert. Mais les évolutions sont contrôlées, en devenant-elles aussi standard. Par ailleurs VHDL supporte les mécanismes de fonctions et procédures permettant aux concepteurs d'étendre les fonctionnalités du langage, en toute portabilité et dans le respect de règles strictes.

Historique

Au début des années 80, le département de la défense américaine (DOD) désire standardiser un langage de description et de documentation des systèmes matériels ainsi qu'un langage logiciel afin d'avoir une indépendance vis-à-vis de leurs fournisseurs. C'est pourquoi, le DOD développa deux langages qui se ressembleront fortement : ADA (logiciel) et VHDL (matériel).

La standardisation du VHDL s'effectuera jusqu'en 1987, époque à laquelle elle sera normalisée par l'IEEE (Institute of Electrical and Electronics Engineers). Cette première normalisation a comme objectif :

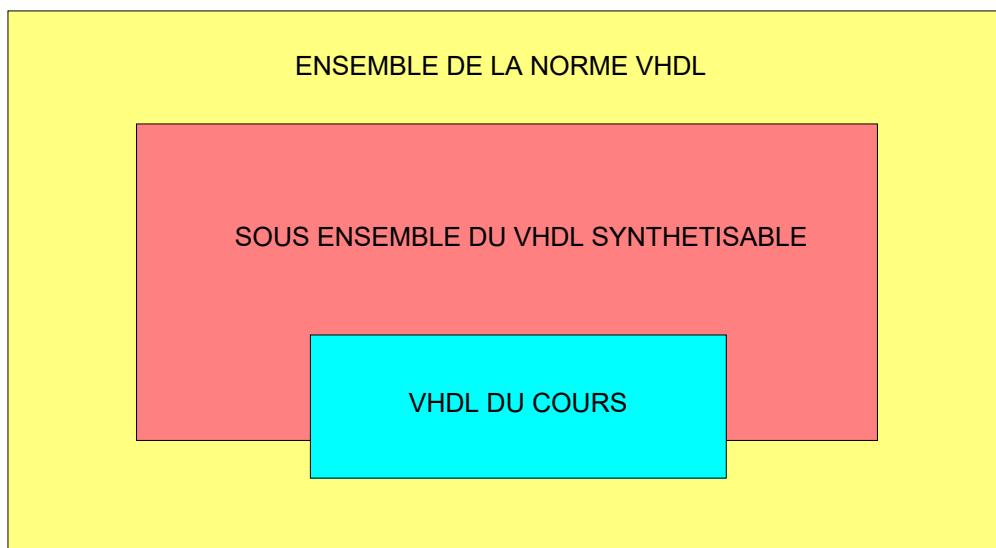
- La spécification par la description de circuit et de système.

- La simulation afin de vérifier la fonctionnalité du système.
- La conception afin de tester une fonctionnalité identique des systèmes décrits avec des solutions d'implémentations de différents niveaux d'abstraction.

En 1993, une nouvelle normalisation par l'IEEE du VHDL a permis d'étendre les capacités de VHDL vers :

- La synthèse automatique de circuit à partir des descriptions.
- La vérification des contraintes temporelles.
- La preuve formelle d'équivalence de circuits.

Mais attention, l'ensemble du langage VHDL n'est pas synthétisable. Certaines instructions du VHDL sont clairement non synthétisables. Il y a par exemple l'instruction « wait for 10 ns » qui modélise le temps. Mais bien souvent, c'est la manière d'utiliser une instruction qui rend la description non synthétisable. La situation peut se représenter de la manière suivante :



En 1999, l'IEEE a édité la norme 1076.6 qui définit le sous-ensemble synthétisable. Cette norme définit l'ensemble des syntaxes autorisées en synthèse.

En résumé, le VHDL est un produit standard reconnu par tous les vendeurs d'outils CAO, ce qui lui assure une pérennité et permet aux industriels d'investir sur un outil qui n'est pas qu'une mode éphémère, c'est donc un produit commercialement inévitable. Techniquelement, il est incontournable car c'est un langage puissant, moderne et général qui assure une excellente lisibilité, une haute modularité et une fiabilité de ses descriptions.

Règles d'écriture de VHDL

Commentaires

Ils débutent par un double tiret `--' et se prolongent jusqu'à la fin de la ligne. Si le commentaire doit se poursuivre sur la ligne suivante, un double tiret est à nouveau de rigueur

```
a <= b AND c ;      --en VHDL le symbole <= effectue une assignation
                      --a prend la valeur de l'opération logique (b and c)
```

Majuscules et minuscules

VHDL ne distingue pas les majuscules des minuscules : pour lui BONJOUR est identique à bonjour ou à Bonjour.

Identificateurs

VHDL manipule de nombreux objets : signaux, paquetages, processus, etc. désignés par leur nom. Les règles de dénomination sont les mêmes pour tous :

- le nom est constitué d'une suite de caractères alphabétiques (les 26 lettres de l'alphabet uniquement), numériques (les 10 chiffres décimaux), ou du caractère souligner '_' (underscore). Sont exclus les caractères ASCII spéciaux, par exemple les lettres accentuées.
- le premier caractère doit être une lettre.
- le caractère '_' ne doit pas terminer le nom, ni y figurer deux fois consécutives.
- le nom ne doit pas être un mot réservé de VHDL (liste ci-après).
- la longueur d'un nom est quelconque mais ne doit pas excéder une ligne. Cette tolérance permet l'attribution de noms explicites, de préférence à des abréviations. Le caractère '_' est souvent utilisé comme séparateur pour les noms "à rallonge".

Noms réservés

Ces mots-clés ne peuvent être utilisés comme identificateurs, même si beaucoup d'entre eux n'ont pas d'utilité en synthèse logique.

abs	access	after	alias
all	and	architecture	array
assert	attributs	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertiel	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	pull
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor

Cette liste correspond à la mise à jour IEEE Std 1076-1993 du manuel de référence du langage VHDL.

Espaces, sauts de lignes

Les espaces ne sont pas significatifs, sauf dans les valeurs littérales et pour la séparation des identificateurs. Ils ne sont pas nécessaires entre identificateurs et symboles. Les deux lignes suivantes sont correctes et équivalentes :

```
A <= B and C;
A<=B and C ;
```

Une instruction peut s'étendre sur plusieurs lignes, tant que le saut de ligne ne s'effectue pas au milieu d'un identificateur, d'un symbole d'opérateur, ou d'un littéral. En d'autres termes, rien d'autre que du bon sens :

```
A <= B when C='0' else D;
```

peut aussi s'écrire

```
A <= B when C='0'  
else D;
```

Fin d'instruction

Une fin d'instruction est repérée par le caractère ';' . C'est bien entendu l'une des erreurs les plus fréquentes que de l'oublier.

Les constantes littérales

Les constantes littérales désignent des valeurs numériques, des caractères, des chaînes de caractères ou des chaînes de valeurs binaires.

Les caractères et les chaînes

Les caractères sont compris entre guillemets simples, et les chaînes de caractères entre guillemet doubles, exemple :

```
'b'      '1'      '0'      'z'          --des caracteres  
"101101"    "chaine_de_caracteres"  --des chaines de caracteres
```

Les chaînes binaires

La différence entre chaînes binaires et chaînes de caractères équivalente réside dans les facilités d'écritures apportées par la base, et les caractères soulignés (underscore) autorisé dans les premières pour faciliter la lecture, exemple :

```
B "1111_1110"          --Equivaut a "11111110"  
X "FE"                 --Equivaut a "11111110"  
O "376"                --Equivaut a "01111110"
```

Il est important de noter que le nombre de caractères binaires générés dépend de la base, les deux derniers exemples ne sont pas équivalents, même si leurs équivalents numériques le sont.

La paire Entity/Architecture

La brique de base de toute description VHDL est la paire entity/architecture, également appelée Design entity. Le partage des tâches est simple : l'entité décrit l'interface externe, l'architecture le fonctionnement interne. L'entité se lit et se comprend aisément :

dans l'exemple ci-dessous la fonction à décrire se nomme AZERTY, son interface externe est constituée des signaux d'entrée I1, I2, I3, I4 et des signaux de sortie O1 et O2 (le chapitre suivant explicitera et détaillera la formulation de ces déclarations). Derrière la déclaration d'entité vient l'architecture : celle-ci commence par décliner son identité (MA_DESCRIPTION), et celle de l'entité à laquelle elle est attachée. Vient ensuite la description proprement dite du fonctionnement.

```
ENTITY AZERTY IS
  PORT (
    I1,I2,I3,I4 : in  std_logic;
    O1,O2        : out std_logic
  );
END AZERTY;

ARCHITECTURE MA_DESCRIPTION OF AZERTY IS
BEGIN
  --(description du fonctionnement)
END MA_DESCRIPTION;
```

Si entité et architecture vont toujours de pair, elles correspondent cependant à des unités de compilations distinctes: une entité peut être analysée seule, une architecture également à la condition que l'entité associée ait été analysée avant elle. Mais le plus souvent (notamment dans les exemples du cours) entité et architecture sont regroupées dans un même fichier, et analysées au cours d'une même session de compilation. Cette manière de faire facilite la compréhension et limite les risques d'erreurs.

Déclaration d'entité (entity)

La déclaration d'entité décrit l'interface entre le monde extérieur et une unité de conception : signaux d'entrées et de sorties et, éventuellement, paramètres génériques.

Syntaxe

```
ENTITY ENTITY_NAME IS
  PORT (
    signal_1 : mode type;
    signal_2 : mode type;
    signal_n : mode type
  );
END ENTITY_NAME;
```

Mode

Le mode précise le sens du signal. Répétons-le, VHDL est un langage contrôlé, et l'interconnexion de signaux de types ou modes différents n'est pas tolérée (sauf à utiliser des fonctions de conversion spécialement prévues à cet effet). Quatre modes sont définis par VHDL :

- **IN** : applicable à un signal d'entrée (monodirectionnel)
- **OUT** : applicable à un signal de sortie (monodirectionnel). Ne peut pas être relu dans l'architecture !
- **INOUT** : applicable à un signal d'entrée/sortie (bidirectionnel)
- **BUFFER** : applicable à un signal de 'sortie/entrée' (monodirectionnel). **Ne pas utiliser.**

Type

VHDL contrôle les types des signaux aussi strictement que les modes, et interdit toute association contre nature. La déclaration de type doit être faite avec soin, compte tenu du nombre et de la variété desdits types, que VHDL classe en deux catégories : scalaire et composite. Des objets de type scalaire possèdent une relation d'ordre, qui permet de les comparer. Les quatre types scalaires génériques retenus par VHDL sont :

- les types entiers, destinés comme leur nom l'indique à la manipulation des nombres entiers.
- les types flottants, id. pour les nombres à virgule flottante.
- les types physiques, destinés à définir des unités de mesure.
- les types énumérés, où les objets prennent leur valeur parmi une liste explicite.

Les types composites correspondent à des regroupements d'objets. VHDL retient deux types composites :

- les types tableaux (array), regroupant des objets de même type.
- les types enregistrements (record), agrégats d'objets de types différents.

Ajoutons que VHDL permet la définition de sous-types à partir des types existants. Sur la base de ces définitions nous classerons les types d'objets en trois catégories : les types prédefinis du langage, les types complémentaires apportés par des bibliothèques, et les types définis par l'utilisateur.

Types prédefinis

`integer` : type scalaire entier définissant des entiers compris entre -($2^{31}-1$) à +($2^{31}-1$). L'intervalle de variation peut être limité, au moyen de la directive range. Par exemple:

`NUM : in integer range -128 to 127;`

`natural` : sous-type du type `integer`, limité aux nombres positifs ou nuls.

`positive`: sous-type du type `integer`, limité aux nombres positifs.

`bit` : type scalaire énuméré, dont les deux seules valeurs possibles sont '0' et '1'.

`bit_vector`: type composite représentant un vecteur de bit, défini à partir du type générique array :

`type bit-vector is array (natural range <>) of bit;`

exemples de déclaration de signaux de ce type:

`DATA : in bit_vector(15 downto 0);`

Est un vecteur qui contient les signaux:

`DATA(15), DATA(14), ..., DATA(1), DATA(0).`

L'indice de gauche est celui du bit de poids fort, l'indice de droite celui du bit de poids faible. La numérotation peut être décroissante des poids forts vers les poids faibles comme dans l'exemple ci-dessus ou croissante. Il est fortement recommandé d'utiliser la numérotation décroissante. Il n'est pas imposé que l'indexation commence à 0. Par exemple:

`DATA : out bit_vector(22 downto 16);`

représente un bus de sortie de largeur 7 bits.

- boolean : type scalaire énuméré, dont les deux valeurs possibles sont FALSE et TRUE (false et true). Bien que proche du type bit, il ne lui est pas équivalent, toute tentative d'association d'un bit et d'un booléen sera refusée.
- real : type scalaire flottant, défini sur un intervalle au moins égal à -1.0^{E38} à 1.0^{E38} .
- time : type scalaire physique définissant les unités de temps, de la femtoseconde à l'heure, en passant par les décades intermédiaires.

Types complémentaires

S'il ne pré définit qu'un nombre limité de types, VHDL permet la création de types complémentaires. Un exemple fréquent concerne les signaux à trois-états : de tels signaux ne peuvent être décris par le type bit, qui ne prévoit que deux états : '0' et '1'. Il a donc fallu dès l'origine que les fournisseurs d'outils VHDL fournissent des packages contenant les définitions de types supplémentaires. Ce qui devait arriver arriva : ces nouveaux types n'étaient pas compatibles d'un fournisseur à l'autre, nuisant à la portabilité des descriptions. C'est pour régler ce problème que furent standardisées les spécifications IEEE1164. Celles-ci définissent un type nouveau pour les signaux dits multi valeurs : le type std_logic, et son extension std_logic_vector.

Le type std_logic peut prendre 9 valeurs, décrivant tous les états d'un signal électronique numérique :

- 'U' non initialisé
- 'X' niveau inconnu, forçage fort (multiple driver)
- '0' niveau 0, forçage fort
- '1' niveau 1, forçage fort
- 'Z' haute impédance
- 'W' niveau inconnu, forçage faible
- 'L' niveau 0, forçage faible (pull-down)
- 'H' niveau 1, forçage faible (pull-up)
- '-' quelconque (indifférent)

Le type std_logic est défini par le paquetage normalisé IEEE.std_logic_1164, il est donc indispensable de déclarer celui-ci au début de son fichier VHDL. Il est fortement recommandé voir même obligatoire d'utiliser ce type en remplacement du type bit (ce que nous ferons dans tous les exemples), pour les possibilités supplémentaires offertes, et principalement pour une raison de compatibilité de type avec les fichiers de simulations (testbench). Il est important de noter toutefois que les valeurs '0' et 'L' sont équivalentes pour les synthétiseurs, de même que '1' et 'H'. Les valeurs 'U', 'X' et 'W' ne sont pas supportées par les synthétiseurs, contrairement à la valeur '-' (indique un état indifférent), qui est particulièrement utile pour l'optimisation et la simplification des équations.

exemple de déclaration d'un signal de type std_logic_vector :

```
DATA : in std_logic_vector(15 downto 0);
```

est un vecteur qui contient les 16 signaux de type std_logic suivants :

```
DATA(15), DATA(14), ..., DATA(1), DATA(0).
```

On peut de cette manière atteindre un bit d'un vecteur.

Remarque : c'est par souci de simplification que nous insistons sur le type std_logic. Le type de base défini par l'IEEE1164 est en fait std_ulogic (et son extension std_ulogic_vector), caractérisant des signaux multi valeurs non résolus (ulogic pour unresolved logic), c'est-à-dire des signaux mono sources. C'est sur ce type de base qu'est défini le sous-type std_logic (std_logic_vector) pour des signaux résolus, c'est-à-dire multi sources : la valeur résultante est définie par une fonction dite de résolution.

Parallèlement au standard l'IEEE1164 axé sur la logique multi valeurs, le standard IEEE1076.3 s'est lui intéressé spécialement à la synthèse logique, et a défini deux types complémentaires pour l'harmonisation des traitements arithmétiques : le type `signed` et le type `unsigned`. Leur définition est similaire au type `std_logic_vector` :

```
type signed is array (natural range <>) of std_logic;
type unsigned is array (natural range <>) of std_logic;
```

La distinction réside dans l'action différenciée des opérateurs arithmétiques selon le type choisi. Un chapitre dédié à l'arithmétique sera abordé dans les système séquentiels (compteurs)

Attention : si les descriptions VHDL destinées à la modélisation et à la simulation font un large usage de tous les types et notamment `integer`, `real` et `time`, il en va différemment en synthèse logique où les types `std_logic` (`std_logic_vector`) sont, les types, les plus utilisés car se matérialisant sans ambiguïté en électronique binaire. La synthèse d'un signal `integer` peut se traduire différemment selon les outils utilisés (voir ci-après les types complémentaires introduits pour faciliter cette manipulation). Celle d'un signal de type `real` est hors de portée d'un synthétiseur. Quant au type `time`, il est par essence destiné à la modélisation et la simulation, et ignoré des synthétiseurs. Des fonctions spéciales, fournies sous forme de packages, permettent sous certaines conditions de convertir un type en un autre, ou d'associer un type à un autre.

Types définis par l'utilisateur

Ils peuvent être très utiles car "taillés sur mesure". Par ailleurs, décrits en VHDL, ils sont par définition portables. Les plus fréquents sont les types énumérés, bien adaptés à la description de machines d'états. Exemple :

```
type ETAT is (REPOS, INIT, ACTIV, ATTENTE, FIN, ERREUR) ;
```

Une fois ce type défini. un signal pourra être déclaré ainsi :

```
MACH_ETAT : buffer ETAT;
```

Une autre catégorie est celle des tableaux définis à partir du type `array`. Par exemple:

```
type TABLE_XY is array (0 to 3, 0 to 7) of std_logic;
```

Les records permettent de rassembler des objets (même de types différents) dans une même organisation, et de repérer chaque élément par son nom. Par exemple :

```
type DATE is
record
    jour : std_logic_vector(5 downto 0) ;
    mois : std_logic_vector(4 downto 0) ;
    annee : std_logic_vector(13 downto 0) ;
end record DATE;
```

Enfin, l'utilisateur peut créer des sous-types à partir de types existants. Par exemple :

```
subtype OCTET is std_logic_vector(7 downto 0);
```

La bibliothèque (library)

VHDL est un langage méticuleux. Pas question de stocker les entités, architectures, configurations et autres package au petit bonheur la chance. Une structure d'accueil est définie spécialement à cet effet: la bibliothèque ([library](#)).

VHDL est une grande maison, qui peut accueillir de nombreuses bibliothèques: certaines prédefinies par le langage (ex :IEEE), d'autres ajoutées par les fournisseurs d'outils, ou par les utilisateurs. L'une des bibliothèques prédefinies est particulièrement importante, et d'ailleurs toujours présente : `work`, bibliothèque de travail où sont stockés par défaut les modules analysés au cours d'une session.

Pour accéder aux éléments d'une bibliothèque, rien de plus simple : il suffit de nommer celle-ci avant une déclaration d'entité. Par exemple :

```
library IEEE;      --déclaration d'accès à une bibliothèque nommée IEEE
```

Remarque : La bibliothèque `work` est toujours accessible, sans besoin de déclaration préalable.

Le paquetage (package)

VHDL permet de regrouper des composants, des définitions de types, constantes, procédures... fréquemment utilisés afin d'en faciliter l'accès. La structure d'accueil pour ce regroupement est un fichier nommé paquetages ([package](#)). Certains paquetages sont inclus dans les outils de développement, d'autres sont créés par les utilisateurs. Une fois constitué, compilé et archivé dans une bibliothèque, un paquetage peut être accédé par quiconque à l'aide de la directive :

```
use nom_library.nom_package.all;
```

qui se traduit par : « utiliser, dans ladite bibliothèque, tous les éléments (all) dudit package». Si l'on ne souhaite accéder qu'à l'un de ses éléments, il est possible de remplacer le suffixe `all` par le nom du composant visé, cette manière de faire étant toutefois rarement utilisée.

L'accès aux éléments apportés par les standards IEEE1164 et IEEE1076.3 en complément de la norme IEEE1076 initiale passe par la déclaration de packages particuliers. Le package `std_logic_1164` de la library `ieee` permet la déclaration et la manipulation des types `std_logic`. Il se déclare de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
```

Paquetage normalisé IEEE est à utiliser pour les opérations arithmétiques utilisées en synthèse logique :

```
numeric_std
```

L'architecture

Le fonctionnement interne d'un module, son corps, est précisé par une architecture associée à l'entité qui décrit l'aspect extérieur de ce module. Une architecture porte un nom, ce qui autorise la création de plusieurs architectures différentes pour la même déclaration d'entité. Une unité de conception est la réunion d'une entité et d'une architecture.

Niveaux de description

VHDL offre l'avantage de pouvoir effectuer les descriptions en se plaçant à différents niveaux, du niveau comportemental le plus abstrait au bas niveau structurel le plus détaillé.

Description comportementale

Dans une description comportementale (behavioral), la description de la fonctionnalité est faite sans hypothèse sur sa réalisation concrète, ni référence à l'architecture matérielle sous-jacente, à l'aide de la syntaxe évoluée du langage. C'est sur l'intelligence du synthétiseur que repose l'efficacité de la méthode : à lui d'interpréter la description pour la traduire en logique concrète. En synthèse logique, les descriptions comportementales sont toujours recommandées, sauf raison majeure de densité ou de performance après synthèse.

Description flot de données

Une description flot de données (dataflow), où les signaux passent à travers des couches d'opérateurs logiques qui décrivent les étapes successives qui font passer des entrées d'un module à sa sortie. C'est en gros l'équivalent d'un schéma avec des portes logiques.

Description structurelle

Une description structurelle (structural) utilise des composants supposés exister dans une librairie de travail, sous forme d'unités de conception. Le programme se contente alors d'instancier les composants nécessaires et de décrire leurs interconnexions.

Syntaxe d'une architecture

Si l'entité représente la vue externe d'une fonction, la description du fonctionnement de celle-ci s'effectue dans l'architecture. L'architecture est divisée en deux parties : une zone déclarative et une zone d'instructions. Ces instructions sont concurrentes, elles s'exécutent en parallèle. Cela signifie que les instructions d'une architecture peuvent être écrites dans un ordre quelconque, le fonctionnement ne dépend pas de cet ordre d'écriture. Ce point est simple à comprendre si on « pense circuits », les différentes parties d'un circuit coexistent et agissent simultanément.

```
ARCHITECTURE ARCHITECTURE_NAME OF ENTITY_NAME IS
  zone déclarative
  BEGIN
    instruction concurrente;
    instruction concurrente;
    ..
    instruction concurrente;
  END [ architecture ] [ARCHITECTURE_NAME];
```

La zone déclarative permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet également de déclarer des noms d'objets externes (composants) utilisés dans le corps de l'architecture, voir le chapitre Instanciation de composants.

Zone déclarative : Signaux, constantes, alias, variables, composants...

La partie déclarative de l'architecture est destinée comme son nom l'indique à accueillir les déclarations des objets allant être utilisés, à l'exception des signaux d'entrée/sortie (ceux-ci sont déclarés dans l'entité, et directement utilisables dans l'architecture). Ces objets sont en grande majorité des signaux, des constantes, des variables, des alias ou des composants (cf. instantiation de composants). Les signaux représentent la classe la plus utilisée car incontournable, les autres apportant plutôt une facilité de lecture, d'exécution ou de compréhension des descriptions.

Signaux

Une description d'architecture nécessite fréquemment de faire appel à des signaux internes. La notion de signal est assimilable, au moins dans un premier temps, à un simple fil d'interconnexion. Un signal interne est équivalent à un signal d'entité, à deux exceptions près :

- un signal interne n'a pas de mode (sens), ce n'est qu'un fil d'interconnexion. Exemples de déclarations de signaux internes :

```
SIGNAL BUS_LOCAL : std_logic_vector(23 DOWNTO 0);
SIGNAL CARRY_IN, CARRY_OUT : std_logic;
```

- un signal interne peut être amené à disparaître au cours des processus de synthèse et d'optimisation (des directives existent pour empêcher cette disparition si elle est jugée préjudiciable, elles sont spécifiques aux outils utilisés).

Constantes

Une constante peut être assimilée à un signal interne auquel est associée une valeur fixe et définitive. La déclaration de constante est rarement une nécessité, plus généralement une simplification d'écriture pour une meilleure compréhension, une meilleure lisibilité et une meilleure évolutivité. Le typage est similaire à celui des signaux, l'affection de valeur s'effectue via l'opérateur ':=' comme le montrent les quelques exemples suivants :

```
CONSTANT ZERO : std_logic_vector(1 DOWNTO 0) := "00";
CONSTANT HUIT_OU_PLUS : std_logic_vector(3 DOWNTO 0) := "1---";
CONSTANT HAUTE_IMPED : std_logic_vector(7 DOWNTO 0) := "ZZZZZZZZ";
```

L'écriture de cette dernière déclaration peut se simplifier en :

```
CONSTANT HI_Z : std_logic_vector(15 DOWNTO 0) := (OTHERS => 'Z');
```

La notation OTHERS => valeur, utilisable également pour des signaux, permet d'affecter une même valeur à tous les éléments (littéralement à tous les autres éléments) d'un vecteur. Outre qu'elle apporte une simplification d'écriture, elle évite d'avoir à connaître le nombre d'éléments du vecteur.

Variables

Une variable est un objet capable de retenir une valeur, pendant une durée limitée. Elle ne peut être employée qu'à l'intérieur d'un process (cf chapitre suivant). A l'opposé d'un signal une variable n'est pas une liaison concrète, et ne laissera pas de trace après synthèse. Si le recours à des variables est monnaie courante dans les descriptions VHDL destinées à la simulation ou à la modélisation, autrement dit dans les descriptions algorithmiques, il est plus rare en synthèse logique (hormis pour le contrôle de boucles d'instructions) et nécessite de connaître précisément les possibilités et capacités des outils VHDL utilisés.

Exemple de déclaration de variables : VARIABLE TEMP, INDICE : integer;

L'assignation d'une valeur à une variable se fait via l'opérateur d'assignation immédiate := déjà rencontré pour les constantes :

```
INDICE := 12;
```

Alias

Une déclaration d'alias permet de dénommer un objet de différentes manières. Soit par exemple un bus de 32 bits, que l'on souhaite pouvoir utiliser indifféremment par demi-mot ou par octet :

```
SIGNAL DBUS : std_logic_vector(31 DOWNTO 0);
```

Une solution pour désigner les différents sous-ensembles de ce bus consiste à les nommer explicitement, par exemple DBUS(31 DOWNTO 24) pour désigner l' octet de poids fort, DBUS(15 DOWNTO 0) pour le demi-mot de poids faible, etc.

Cette manière de faire devient rapidement fastidieuse si les sous éléments sont utilisés fréquemment. Une autre solution consiste à redéfinir le signal DBUS par des alias

```
SIGNAL DBUS : std_logic_vector (31 DOWNTO 0);
ALIAS OCTET3 : std_logic_vector (7 DOWNTO 0) IS DBUS(31 DOWNTO 24);
ALIAS OCTET2 : std_logic_vector (7 DOWNTO 0) IS DBUS(23 DOWNTO 16);
ALIAS OCTET1 : std_logic_vector (7 DOWNTO 0) IS DBUS(15 DOWNTO 8);
ALIAS OCTETO : std_logic_vector (7 DOWNTO 0) IS DBUS(7 DOWNTO 0);
ALIAS DEMI1 : std_logic_vector (15 DOWNTO 0) IS DBUS(31 DOWNTO 16);
ALIAS DEMI0 : std_logic_vector (15 DOWNTO 0) IS DBUS(15 DOWNTO 0);
```

Les différents sous-ensembles de DBUS sont désormais accessibles par leur nouveau nom. A noter que la nouvelle dénomination ne remplace pas la précédente, mais s'y ajoute.

Composants

Voir chapitre Instanciation de composants p.78.

Les opérateurs

Les opérateurs prédéfinis du langage sont énumérés dans le Tableau 5 par classes de priorités croissantes, de haut en bas. Des opérateurs de même classe ont donc la même priorité. Rappelons au lecteur que les opérateurs logiques NAND et NOR ne sont pas associatifs. Des parenthèses sont donc indispensables dans les expressions où apparaissent plusieurs de ces opérateurs.

VHDL est un langage dans lequel il est possible d'étendre le domaine d'utilisation d'un opérateur (types des opérandes) par l'écriture d'une fonction qui porte le nom de l'opérateur, c'est ce que l'on appelle la surcharge. Cette opération ne modifie pas la classe de priorité à laquelle appartient l'opérateur. Les librairies spécifiques d'un outil de CAO contiennent systématiquement des opérateurs surchargés. Les types d'opérandes évoqués ci-dessous sont ceux qui sont définis dans le langage, en l'absence de toute extension par surcharge.

Logical operator	<code>and</code>	<code>or</code>	<code>nand</code>	<code>nor</code>	<code>xor</code>	<code>xnor</code>
Relational operator	<code>=</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
Shift operator	<code>sll</code>	<code>srl</code>	<code>sla</code>	<code>sra</code>	<code>rol</code>	<code>ror</code>
Adding operator	<code>+</code>	<code>-</code>	<code>&</code>			
Sign	<code>+</code>	<code>-</code>				
Multiplying operator	<code>*</code>	<code>/</code>	<code>mod</code>	<code>rem</code>		
Miscellaneous operator	<code>**</code>	<code>abs</code>	<code>not</code>			

Tableau 5 Liste des opérateurs VHDL

On notera que l'échelle de priorités n'est pas toujours évidente, aussi ne faut-il pas hésiter à utiliser des parenthèses en cas de doute. La plupart de ces opérateurs n'appellent pas de commentaire, leur signification est identique dans la majorité des langages de programmation. Pour les autres, les opérateurs logiques par exemple, leur nom exprime leur fonction. Nous nous contenterons ci-dessous d'apporter quelques précisions utiles :

Opérateurs logiques

Ce sont des opérateurs binaires (à deux opérandes) dont les opérandes et le résultat sont des scalaires ou des vecteurs (tableaux à une dimension) de type bit ou booléen.

Opérateurs relationnels

Ce sont des opérateurs binaires dont les deux opérandes sont de même type et le résultat de type booléen. L'égalité et l'inégalité acceptent des opérandes de type quelconque, sans restriction. Les relations d'ordre sont limitées aux types scalaires et aux vecteurs de types discrets. Les vecteurs sont comparés de gauche à droite.

Opérateurs de décalages et de rotation

Ces opérateurs, rajoutés par la norme 1993, agissent sur des vecteurs d'éléments de type bit ou boolean. Ils effectuent les quatre types de décalages classiques : logique ou arithmétique, à gauche ou à droite, et les rotations dans les deux sens. Leur opérande de gauche est le tableau, leur opérande de droite est un entier qui indique le nombre de décalages élémentaires (d'une case) à réaliser. Leur nom indique l'opération, par exemple : `sla` pour shift left arithmetic, `srl` pour shift right logical.

Exemples :

```
subtype octet is std_logic_vector(7 downto 0);
constant moins_77: octet := "10110011";
constant moins_39: octet := moins_77 sra 1; --11011001
constant plus_89: octet := moins_77 srl 1; --01011001
constant plus_103: octet := moins_77 rol 1; --01100111
```

Opérateurs additifs et de concaténation

L'addition et la soustraction agissent sur deux opérandes numériques de même type. L'opérateur & fournit un vecteur à partir de la concaténation de deux vecteurs dont les éléments sont de même type.

Exemples :

```
subtype octet is std_logic_vector(7 downto 0);
constant moins_77 : octet := "101" & "10011";
constant moins_77_bis : octet := "1011001" & '1';
```

Opérateurs de signe

Ce sont des opérateurs unaires qui agissent sur n'importe quel type numérique. Leur priorité inférieure à celle des opérateurs multiplicatifs interdit des expressions comme $A/-B$ qui doit être écrit $A/(-B)$.

Opérateurs multiplicatifs

Multiplication et division agissent sur des opérandes (le même type numérique. Les opérateurs modulo et reste agissent sur des entiers. Les opérateurs multiplicatifs ne sont pas toujours tous acceptés en synthèse, même pour des types entiers.

Multiplication et division sont étendues aux types physiques, sous réserve que le résultat conserve un sens physique ou soit un nombre sans dimension.

Opérateurs divers

La négation logique agit sur des types bit ou boolean ou des vecteurs de ces types.

La valeur absolue agit sur n'importe quel type numérique.

L'exponentiation accepte un opérande de gauche numérique et un opérande de droite entier.

Ces deux derniers opérateurs ne sont évidemment pas synthétisables dans le cas général.

Surcharge des opérateurs

L'utilisation des opérateurs ci-dessus est restreinte à certains types précis, eux-mêmes choisis parmi les types prédéfinis du VHDL. Or le VHDL permet de créer de nouveaux types, le meilleur exemple étant le type `std_logic`, défini dans la bibliothèque IEEE1164. Comment étendre la portée des opérateurs standards à de nouveaux types d'opérandes ? La réponse est apportée par le mécanisme de « surcharge » (overloading). Cette opération consiste à définir l'action des opérateurs en présence d'opérandes de types initialement non prévus. La description de la « nouvelle » action s'effectue via des fonctions (voir chapitre : Les sous-programmes: procédures et fonctions) elles-mêmes décrites en VHDL. Ces fonctions sont regroupées en paquetages, accessibles en utilisant la clause `use` (voir chapitre : Le paquetage (package)).

Instructions concurrentes

VHDL se distingue des langages de programmation traditionnels par une caractéristique majeure : les instructions d'une architecture sont, sauf exception, évaluées en permanence, et toutes simultanément, ce type de fonctionnement est appelé parallèle ou concurrent. Dans ce cas l'ordre des instructions n'a pas d'importance.

Attribution inconditionnelle

L'instruction concurrente la plus élémentaire est l'affectation d'une valeur à un signal. Le symbole réservé à l'affectation d'une valeur à un signal est <= Forme générale :

```
signal <= expression;
```

Exemples :

```
A <= B AND C ; --A prend la valeur du résultat de l'opération (B and C)
enable <= '1'; --Le signal enable prend l'état logique 1
BUS_IN <= "0000" ; --Les 4 bits du vecteur BUS_IN prennent la valeur 0
```

Attribution d'un signal ou d'un vecteur

L'affectation d'un état logique à un signal s'effectue à l'aide des guillemets, alors que l'affectation d'un état logique à un vecteur s'effectue à l'aide des doubles guillemets.

```
A <= '1';
BUS_IN <= "1001";
```

Attribution conditionnelle

Forme générale:

```
signal <= expression1 WHEN condition1 ELSE
expression2 WHEN condition2 ELSE
..
expressionx WHEN conditionx ELSE expression_autre;
```

Le signal prend la valeur expression1 si la condition1 est vraie (true), sinon (si la condition1 est fausse (false)) le signal prend la valeur expression2 si la condition2 est vraie... sinon (si la condition2 est fausse) le signal prend la valeur expressionx si la conditionx est vraie sinon (sous-entendu aucune des conditions précédentes n'est vraie) le signal prend la valeur expression_autre.

Exemple :

```
A <= B when C='1' else D;
```

Traduction : A prend la valeur de B si C est égal à '1' sinon (A prend la valeur de) D. La condition testée doit être booléenne : c'est le cas ici, rappelons-nous que les opérateurs relationnels fournissent un résultat booléen. L'attribution conditionnelle peut être à double détente, voire plus:

```
A <= B when S=T else
C when S=U else '0';
```

L'enchaînement de conditions amène deux remarques :

1. Si les conditions listées sont exclusives (c'est-à-dire ne sont jamais vraies simultanément), la logique sera synthétisée sans ambiguïté.
2. Si les conditions ne sont pas exclusives, alors la logique sera synthétisée compte tenu de l'ordre de déclaration des conditions:

```
A <=      B when C='0' else
          D when E='1' else '0';
```

sera traitée comme :

```
A <=  B when (C='0') else
      D when (C='1' and E='1') else '0';
```

ce qui n'est pas nécessairement ce à quoi l'on s'attendait.

Il est fortement conseillé, pour ne pas dire imposé, que les conditions listées soient exhaustives, c'est-à-dire ne laissent pas de cas indéterminés, sous peine de voir le synthétiseur générer une logique inattendue (éléments mémoires, latchs), or dans la logique combinatoire il n'y pas d'éléments mémoire !

Toutes les fonctions combinatoires peuvent se décrire avec une assignation conditionnelle. Prenons le cas d'une fonction logique combinatoire à 3 variables d'entrée :

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

Version sans vecteur:

```
Z <= '0' when A='0' AND B='0' AND C='0' else
      '0' when A='0' AND B='0' AND C='1' else
      '0' when A='0' AND B='1' AND C='0' else '1';
```

Version avec vecteur:

```
Z <= '0' when vectABC = "000" else
      '0' when vectABC = "001" else
      '0' when vectABC = "010" else '1';
```

Attribution sélective

Forme générale:

```
WITH selecteur SELECT
    signal <= expression1 WHEN valeur_selecteur1,
    expression2 WHEN valeur_selecteur2,
    ..
    expressionx WHEN OTHERS;
```

Cette instruction permet d'assigner différentes valeurs à un signal, selon la valeur prise par une expression appelée sélecteur. Un exemple sera plus parlant :

```
architecture behavioral of mux is

signal ETAT : std_logic_vector(1 downto 0);
signal X,A,B,C,D : std_logic;

BEGIN
with ETAT select
    X <= A when "00",
    B when "01",
    C when "10",
    D when others;
end ARCH_WITH;
```

Traduction : X prend la valeur de A si le signal ETAT, utilisé comme sélecteur, vaut "00", B si le sélecteur vaut "01", etc.

Autre exemple :

```
with address select
    CS <= '0' when "010",
    '1' when others;
```

La terminaison de l'instruction par `when others`, si elle n'est pas strictement obligatoire, est toujours recommandée, de manière à regrouper toutes les valeurs éventuellement non listées du sélecteur.

Il est possible de regrouper plusieurs valeurs du sélecteur pour une même action : il suffit de les lister séparées par le symbole ' | ' (barre verticale).

Exemple :

```
with address select
    CS <= '0' when "010" | "011",
    '1' when others;
```

Instanciation de composants

Une fonction complexe est généralement construite de façon hiérarchique : assemblage de blocs plus simples interconnectés par des signaux, dans une description structurelle. Chaque bloc lui-même peut être le sommet d'une nouvelle hiérarchie, et ainsi de suite, jusqu'à arriver à des structures suffisamment simples pour être décrites directement, ou être des primitives d'une librairie de composants élémentaires.

Vu d'un niveau hiérarchique, un sous élément est un composant; ce composant peut lui-même être décrit comme une unité de conception (un couple entité-architecture). L'opération d'instanciation consiste à établir les liens entre les signaux d'un niveau et les ports d'accès du niveau inférieur et, le cas échéant, à fixer les valeurs de paramètres génériques.

Déclaration du composant

Pour utiliser un composant il faut le déclarer, dans la zone de déclaration de l'architecture ou dans un paquetage visible depuis cette architecture, et l'instancier dans la zone des instructions. La déclaration reproduit à peu de choses près celle, dans sa forme minimum d'une entité ; ce qui n'a rien d'étonnant :

```
COMPONENT component_name
  PORT (
    signal_1 : mode type;
    signal_2 : mode type;
    ..
    signal_n : mode type;
  );
END COMPONENT;
```

Instanciation du composant

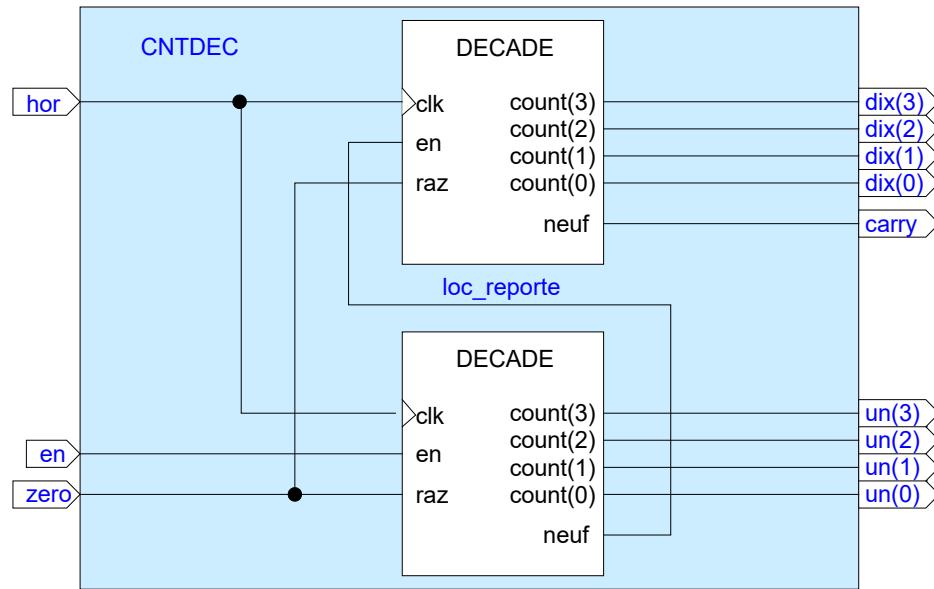
L'utilisation du composant se fait par une instruction d'association précédée d'une étiquette (label) obligatoire:

```
label: component_name
PORT MAP (
  signal_1 => signal_corresp_1,
  signal_2 => signal_corresp_2,
  ..
  signal_n => signal_corresp_n
);
```

Les listes d'associations établissent la correspondance entre les noms internes au composant et les noms des signaux correspondants, ou les expressions constantes dans le cas des paramètres génériques.

Exemple :

L'exemple ci-dessous représente la description VHDL structurelle d'un compteur décimal à deux décades (CNTDEC), réalisé en faisant une double instanciation d'un composant (autre fichier VHDL ou netlist) appelé DECADE :



```

ENTITY CNTDEC IS
    port (en      : in  std_logic;
          hor     : in  std_logic;
          zero   : in  std_logic;
          carry  : out std_logic;
          dix    : out std_logic_vector (3 downto 0);
          un     : out std_logic_vector (3 downto 0));
END CNTDEC;

ARCHITECTURE STRUCTURAL of CNTDEC is

    SIGNAL loc_reporte : std_logic;

    component DECADE
        port (clk      : in  std_logic;
              en       : in  std_logic;
              raz      : in  std_logic;
              count   : out std_logic_vector (3 downto 0);
              neuf    : out std_logic );
    end component;

BEGIN
    unite: DECADE
    port map(
        clk      => hor,
        en       => en,
        raz      => zero,
        count   => un,
        neuf    => loc_reporte );

    dizaine: DECADE
    port map(
        clk      => hor,
        en       => loc_reporte,
        raz      => zero,
        count   => dix,
        neuf    => carry );
END STRUCTURAL;

```

Pour connecter des signaux entre deux composants dans une même architecture, on doit avoir recours à un signal interne. Dans l'exemple ci-dessus il se nomme `loc_reporte`.

Si l'instruction `port map` apparaît comme un peu rébarbatif, il faut bien l'avouer, elle est cependant infiniment plus souple qu'une représentation schématique.

Le processus (process)

Un processus est une instruction structurée concurrente dont le contenu est une suite d'instructions séquentielles.

La syntaxe de l'instruction est la suivante :

```
label:  
PROCESS (sensitivity_list)  
  
    zone déclarative du process  
BEGIN  
    instruction séquentielle;  
    instruction séquentielle;  
    ..  
    instruction séquentielle;  
END PROCESS;
```

La liste de sensibilité (`sensitivity_list`) contient la liste des signaux dont un événement doit réveiller un processus en sommeil. Elle n'est pas obligatoire, mais si elle est absente le processus doit contenir une instruction explicite d'attente, `wait`. Liste de sensibilité et instruction `wait` sont exclusives l'une de l'autre, elles ne doivent en aucun cas être présentes simultanément. En synthèse nous avons toujours une liste de sensibilité tandis qu'en simulation il n'y en a pas et nous devons utiliser l'instruction `wait`.

La zone de déclaration contient la description du domaine privé du processus, rappelons que peuvent y figurer des variables, mais pas des signaux.

La zone d'instructions contient des instructions exécutées séquentiellement, au sens informatique du terme. Cela ne presuppose pas que le processus décrive un système numérique séquentiel. Cependant nous pouvons décrire un système numérique purement combinatoire, au moyen d'un processus.

Vu par un informaticien, un processus est une boucle sans fin. Au lancement du simulateur tous les processus sont activés, charge au programmeur de prévoir un mécanisme de mise en sommeil jusqu'à ce que survienne un événement. Cette mise en sommeil peut se faire explicitement, par une instruction `wait`, qui indique au processus qu'il doit attendre que quelque chose se produise, ou par une liste de sensibilité qui indique les signaux dont le changement de valeur provoque le réveil du processus. Quelques remarques importantes doivent être présentes à l'esprit :

- Un processus qui ne possède ni liste de sensibilité ni `wait` monopolise complètement le simulateur, sans que rien ne se passe, le temps n'évolue pas. Un tel piège est, quand il est détectable, signalé à la compilation, mais n'est pas une erreur.
- Les affectations séquentielles de signaux ne prennent effet que quand le processus se remet en attente. Pendant l'exécution du processus aucun signal ne change de valeur, ce « pendant » dure de toute façon un temps (virtuel) nul.
- Entre deux activations d'un processus qui affecte une valeur à un signal, cette valeur est mémorisée. Tout non-dit est interprété comme une demande de non modification, c'est-à-dire une mémoire.

Liste de sensibilité (sensitivity_list)

- Un processus peut contenir exclusivement des instructions séquentielles, mais peut servir à décrire une fonction logique combinatoire, dans ce cas, la liste de sensibilité (`sensitivity_list`) doit contenir tous les signaux lus à l'intérieur du processus, séparés par des virgules.
- Dans le cas où un processus sert à décrire une fonction logique séquentielle, seul les signaux `clk` et `reset_n` figurent dans la liste de sensibilité.

L'instruction wait

L'instruction `WAIT` suspend l'exécution d'un processus jusqu'à ce qu'un événement, une condition ou une clause de temps écoulé (time out) soit vraie. Si aucune clause de réveil n'est stipulée, le processus s'arrête définitivement. On utilise `wait` uniquement dans les TestBench, jamais dans un fichier synthétisable.

```
WAIT [ON sensitivity_list] [UNTIL condition] [FOR time-expression]
```

La liste de sensibilité spécifie les noms des signaux à surveiller. Si un événement survient sur l'un de ces signaux, la condition est examinée. Si cette condition est vraie le processus est activé, sinon il reste suspendu. Il n'y a aucun lien obligatoire entre les signaux de la liste de sensibilité et ceux qui interviennent dans les autres instructions du processus. Si une condition est fournie, mais pas de liste de sensibilité, cette dernière comporte tous les signaux qui interviennent dans la condition. Ainsi :

```
wait until hor = '1';
```

Le processus correspondant est activé par les fronts montants du signal hor.

Instructions séquentielles

Comme tout langage de programmation, VHDL possède des instructions séquentielles de contrôle, tests et boucles dont la syntaxe est sans surprise. Elles sont au cœur des descriptions comportementales des circuits, plus que jamais, dans ce type de description, il est essentiel de toujours avoir présent à l'esprit l'architecture matérielle du circuit que l'on décrit, ses signaux d'horloge, ses registres, ses blocs combinatoires. De plus ces instructions ne peuvent se trouver qu'à l'intérieur d'un processus.

Assignation inconditionnelle de signal

Elle a la même forme que l'assignation inconditionnelle en mode concurrent. On rappelle toutefois que dans un processus les nouvelles valeurs des signaux sont calculées au fur et à mesure des assignations, mais ne prennent effet qu'à la fin du processus. Pour traduire cette caractéristique, on peut énoncer l'assignation en employant le futur immédiat :

```
A <= (B and C); --A va prendre la valeur de l'opération logique (B and C)
```

Assignation conditionnelle (if...then...else)

Elles permettent d'exécuter des blocs d'instruction, en fonction des résultats de l'évaluation de conditions booléennes. Un traitement par défaut peut être spécifié pour le cas où toutes les conditions sont fausses (`else`) :

```
IF condition THEN
    instruction(s) séquentielle(s)
[ELSIF condition THEN
    instruction(s) séquentielle(s) ]
[ELSIF condition THEN
    instruction(s) séquentielle(s) ]
...
[ELSE
    instruction(s) séquentielle(s) ]
END IF;
```

- Les conditions sont évaluées dans l'ordre d'écriture, des différents blocs d'instructions.
- Un seul bloc d'instructions est exécuté, le premier dont la condition est vraie.
- L'ordre d'écriture permet donc de créer une priorité.
- Les crochets indiquent que les instructions entre crochets sont facultatives, ils ne font pas partie de la syntaxe.

Assignation sélective

Une sélection spécifie des blocs d'instructions séquentielles à effectuer en fonction de la valeur d'une expression testée. Toutes les valeurs possibles de l'expression testée doivent apparaître une fois et une seule dans la suite des alternatives. Le mot clé **OTHERS** est obligatoire, il permet de regrouper toutes les valeurs non précisées explicitement dans la dernière alternative.

```
CASE expression IS
    WHEN valeur_expression1 =>      instruction(s) séquentielle(s)
    [ WHEN valeur_expression2 =>      instruction(s) séquentielle(s) ]
    [ WHEN valeur_expression3 =>      instruction(s) séquentielle(s) ]
    .
    .
    WHEN OTHERS                  =>      instruction(s) séquentielle(s)
END CASE;
```

Mémorisation implicite

Cette caractéristique, propre à VHDL, est importante à appréhender car elle apporte autant d'avantages lorsqu'elle est utilisée à bon escient que de mauvaises surprises lorsqu'elle est oubliée.

En VHDL, les signaux ont une valeur courante et une valeur prochaine, déterminée par l'opérateur d'assignation. Si lors d'une instruction conditionnelle (concurrente ou combinatoire) un signal reçoit une assignation dans une branche, alors il doit recevoir une assignation dans toutes les autres branches ; si ce n'est pas le cas chaque absence d'assignation signifie que la prochaine valeur du signal est identique à la valeur courante, et le synthétiseur génère une logique de mémorisation (registre, latch) afin de préserver cette valeur courante.

- Avantage : simplification de la description des fonctionnements basés sur des signaux d'horloge. Ce qui correspond parfaitement au fonctionnement de registres : les sorties ne changent d'état que sur un front d'horloge.
- Inconvénient : génération inopinée de registres et de latchs lorsque toutes les options d'une condition ne sont pas définies, ce qui est fréquent et non désiré en combinatoire.

Boucles

Couramment utilisées dans les descriptions destinées à la simulation, les boucles doivent être considérées avec soin en synthèse, en s'assurant qu'elles ne risquent pas de provoquer la création de logique encombrante et superflue.

Boucle for .. loop

Forme générale:

```
FOR variable_boucle IN intervalle LOOP
    instruction(s) séquentielle(s)
END LOOP;
```

Traduction : pour chaque valeur successive de la variable de boucle dans l'intervalle indiqué, exécuter les instructions séquentielles.

La variable de boucle, qui n'a pas à être déclarée au préalable, peut bien entendu être utilisée par les instructions de la boucle. La liste d'instructions peut contenir toute instruction séquentielle, par exemple conditionnelle, ou une nouvelle boucle.

Exemple :

```
FOR index IN 1 TO 10 LOOP
    WAIT UNTIL clk_gen'EVENT AND clk_gen = '1';
END LOOP;
```

La description ci-dessus n'est pas synthétisable, par contre elle sera utilisée plus loin dans les fichiers de simulation (testbench).

Il faut rappeler qu'une instruction wait suspend l'exécution d'un processus jusqu'à ce qu'un événement, une condition ou une clause de temps écoulé (time out) soit vraie.

Dans notre exemple la boucle FOR est effectuée 10 fois, et à chaque passage, le processus est suspendu jusqu'à ce qu'un flanc montant sur le signal clk_gen survienne.

Boucle while ... loop

Forme générale :

```
WHILE condition LOOP
    instruction(s) séquentielle(s)
END LOOP;
```

Traduction : tant que la condition est vraie (true), exécuter les instructions de la boucle. Bien entendu les instructions de la boucle doivent assurer que la condition deviendra fausse au bout d'un temps fini.

Il est nécessaire de rappeler qu'une condition est réalisée avec des opérateurs relationnels et logiques.

Les sous-programmes: procédures et fonctions

VHDL connaît deux catégories de sous-programmes : les procédures et les fonctions. Les sous-programmes sont des modules de code séquentiel; ils utilisent donc le même jeu d'instructions que les processus, sauf l'instruction `wait` qui est sujette à des restrictions. En simulation cette instruction est utilisable sous certaines réserves, en synthèse elle est généralement interdite.

Déclaration et définition constituent du code passif, elles figurent dans la zone déclarative d'une entité, d'une architecture, d'un bloc, d'un processus, d'un autre sous-programme, ou, de préférence dans un paquetage qui peut être compilé séparément. Si, ce qui n'est pas particulièrement une bonne pratique de programmation, un sous-programme est défini dans la zone déclarative du modulé qui l'utilise, il est inutile de rajouter une déclaration qui serait alors redondante.

En synthèse, l'emploi de sous-programmes ne fait évidemment faire aucune « économie de silicium », chaque appel à une procédure ou à une fonction provoque la création des opérateurs physiques nécessaires à sa réalisation. Procédures et fonctions diffèrent principalement par les sens de transfert des informations, alors que les premières autorisent des paramètres en entrée et en sortie, les secondes n'ont que de paramètres qu'en entrée, et renvoient un résultat utilisable dans une expression.

Syntaxe

Les sous-programmes, sont des «composants » logiciels Il n'est donc guère étonnant de retrouver à propos de leur usage des constructions syntaxiques semblables à celles qui ont été explorées à propos de l'utilisation des composants, notamment les listes d'association.

Définition

La définition du corps d'un sous-programme comporte, comme à l'habitude, deux parties : une partie déclarative où sont définis les objets (données, types ...) locaux et le corps proprement dit qui contient une suite d'instructions séquentielles. Les sous-programmes faisant partie du monde séquentiel, leurs données locales sont des constantes, des variables ou, en simulation, des fichiers. La zone déclarative peut également contenir les définitions de sous-programmes emboîtés dans celui que l'on est en train de d'écrire.

L'instruction `wait` est interdite dans le corps d'une fonction elle peut figurer dans celui d'une procédure ; mais cette possibilité est souvent exclue par les compilateurs en synthèse. Une fonction a un type qui correspond à la valeur renournée grâce à l'instruction `return`. La syntaxe de définition d'un sous-programme est:

```

PROCEDURE nom [(liste des paramètres formels)]
|
FUNCTION nom [(liste des paramètres formels)] RETURN type
IS
    zone déclarative
BEGIN
    zone d'instructions séquentielles
END [PROCEDURE | FUNCTION];

```

Procédures et fonctions diffèrent sur ce point :

- Les paramètres d'une procédure peuvent être de classe `signal`, `constant`, `variable` ou `file`. Pour les trois premières classes, les modes acceptés sont `in`, `out` et `inout`, les fichiers n'ont pas de mode.
- Les paramètres d'une fonction peuvent être des constantes ou des signaux, toujours de mode `in`, ou des fichiers; jamais des variables.

Les bascules

Introduction

Nous avons étudié jusqu'ici des circuits logiques combinatoires, dont les sorties à un instant donné, ne dépendent que de l'état des valeurs présentes sur les entrées. Toute condition antérieure n'a aucun effet sur les valeurs actuelles des sorties, parce que les circuits combinatoires n'ont pas de mémoire. Dans la majorité des systèmes numériques, on retrouve une combinaison de circuits combinatoires et de dispositifs à mémoire.

La section combinatoire est alimentée par des signaux d'entrée externes et par les sorties des dispositifs à mémoire. Le circuit combinatoire agit sur ces entrées pour produire diverses sorties, certaines servant à déterminer les valeurs binaires stockées dans les éléments de mémoire. La sortie de certains de ces éléments de mémoire revient comme entrée des circuits logiques de la partie combinatoire. Ceci est une indication que les sorties externes d'un système numérique dépendent autant des entrées externes que des informations mémorisées dans d'autres sections. On appelle cela un système séquentiel.

Définition du système séquentiel

Un système est dit séquentiel, si à un même vecteur d'entrée, il fait correspondre plusieurs vecteurs de sortie différents. Chaque vecteur de sortie dépendra alors non seulement du vecteur d'entrée à l'instant t , mais aussi des précédents, ce qui introduit la notion de séquence d'entrée.

L'effet mémoire est typique des systèmes séquentiels, l'élément de mémorisation le plus important est la bascule D, constituée d'un ensemble de portes logiques. Même si, en soi, une porte logique ne retient pas de donnée, il est possible d'en raccorder quelques-unes ensemble afin d'obtenir le stockage d'une information. Il existe différentes façons de monter les portes pour obtenir ces bascules.

Bascule R-S en portes NAND

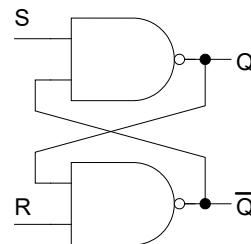
La bascule la plus élémentaire est construite au moyen de deux portes NAND ou de deux portes NOR. La version NON-ET, appelé élément de mémoire en portes NAND ou bascule R-S (dans certains cas on l'appelle aussi bascule SC), est montrée à la Figure 29b). Les deux portes NAND sont rétro-couplées, de sorte que la sortie de la porte NAND 1 est connectée à l'une des entrées de la porte NAND 2, et vice versa. Les sorties, appelée Q et \bar{Q} respectivement, sont les sorties de l'élément de mémoire. Dans des conditions normales, une sortie est toujours l'inverse de l'autre. Les entrées de l'élément de mémoire sont désignées S (d'après SET) et R (d'après RESET). Les entrées S et R se trouvent normalement toutes les deux au niveau HAUT, et l'on doit momentanément en porter une au niveau BAS pour changer l'état de sortie de la bascule. Un élément de mémoire en NAND possède deux états stables possibles quand $S = R = 1$. L'élément de mémoire en NAND peut se mettre sous la forme d'une table de vérité (Figure 29a);

1. $S = R = 1$; cette condition correspond à l'état normal de repos et elle n'affecte pas l'état de sortie de la bascule. Les sorties demeurent dans, l'état qu'elles occupaient avant l'application de cette condition d'entrée.
2. $S = 0, R = 1$; cette condition entraîne toujours la sortie dans l'état 1 où demeure même après le retour de S au niveau HAUT. On dit que c'est la condition de mise à 1 de la mémoire (SET).
3. $S = 1, R = 0$; cette condition entraîne toujours la sortie dans l'état 0 où demeure même après le retour de C au niveau HAUT. On dit que c'est la condition de mise à 0 de la mémoire (RESET).
4. $S = R = 0$; cette condition est équivalente à vouloir mettre la mémoire fois à 1 et à 0, ce qui donne lieu à des résultats ambigus. En fait les deux sorties sont à l'état HAUT simultanément. Cette condition ne doit jamais servir.

S	R	Q	\bar{Q}
1	1	inchangé	inchangé
0	1	1	0
1	0	0	1
0	0	1	1

Ambiguë

a) Table de vérité



b) bascule avec portes NAND

Figure 29 Table de vérité et schéma logique d'une bascule R-S NAND

Bascule R-S en portes NOR

Deux portes NOR rétro-couplées constituent une mémoire R-S (bascule R-S). Un tel montage, illustré à la Figure 30b), est analogue à celui d'une mémoire en NAND, sauf que les sorties Q et \bar{Q} sont maintenant intervertis.

L'étude du fonctionnement d'une mémoire en NOR se développe de manière tout à fait identique à celle de la mémoire en NAND. Les résultats sont donnés sous forme d'une table de vérité à la Figure 30a) et résumés ci-après:

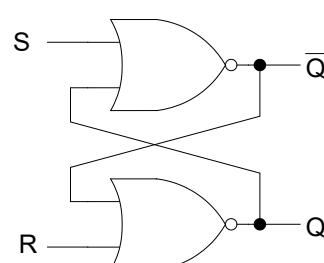
1. $S = R = 0$; cette condition représente l'état normal de repos de la mémoire en NOR et ne modifie en rien l'état de sortie. Q et \bar{Q} demeurent dans l'état qu'elles occupaient avant l'arrivée de l'impulsion d'entrée.
2. $S = 1, R = 0$; cette condition a toujours pour effet de mettre Q à 1, état qui ne change pas même quand S revient à 0.
3. $S = 0, R = 1$; cette condition a toujours pour effet de mettre Q à 0, état qui ne change pas même quand R revient à 0.
4. $S = R = 1$; cette condition est équivalente à vouloir mettre la mémoire à 1 et à 0 en même temps et produit $Q = \bar{Q} = 0$. Si les deux entrées sont ramenées simultanément à 0, l'état de sortie résultante est imprévisible. Il ne faut jamais se servir de cette condition d'entrée.

L'élément de mémoire en NOR fonctionne en tous points comme la mémoire en NAND, à l'exception des entrées S et R qui, maintenant, sont vraies au niveau HAUT plutôt qu'au niveau BAS, et de l'état normal de repos qui est $S = R = 0$. Q sera mis à 1 par une impulsion de niveau HAUT appliquée sur S et sera mis à 0 par une impulsion, toujours de niveau HAUT, sur R.

S	R	Q	\bar{Q}
0	0	inchangé	inchangé
1	0	1	0
0	1	0	1
1	1	0	0

Ambiguë

a) Table de vérité



b) bascule avec portes NOR

Figure 30 Table de vérité et schéma logique d'une bascule R-S NOR

Bascule R-S avec enable

Les bascules R-S étudiées précédemment sont sensibles aux changements sur les entrées S et R, ceci à n'importe quel moment. Cependant il est très facile de modifier un de ces circuits pour le rendre ses entrées sensibles uniquement lorsque une troisième entrée enable (C) est active. Une telle bascule R-S avec enable est montrée dans la Figure 31. Comme on peut le voir dans la table de vérité, ce circuit se comporte comme une simple bascule R-S lorsque l'entrée C (enable) est à 1, et mémorise son état lorsque C est à 0. En effet si l'entrée C est à 0, on retrouve 1 à la sortie des deux portes NAND, ce qui équivaut à l'état mémorisation sur une bascule R-S en porte NAND. Dans le cas où C vaut 1, la porte NAND agit comme un simple inverseur, il nous reste donc une bascule R-S NAND avec entrées inversées.

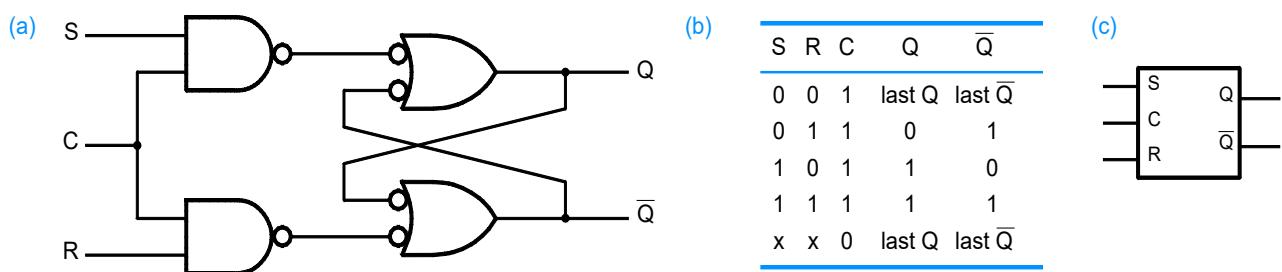


Figure 31 Bascule R-S avec enable: (a) circuit logique; (b) table de vérité; (c) symbole logique

La Figure 32 montre le chronogramme d'une bascule R-S avec enable pour des signaux donnés. Si les deux entrées S et R sont simultanément à l'état logique haut lorsque C passe de 1 à 0, l'état suivant est imprévisible, et la sortie peut se trouver dans un état métastable.

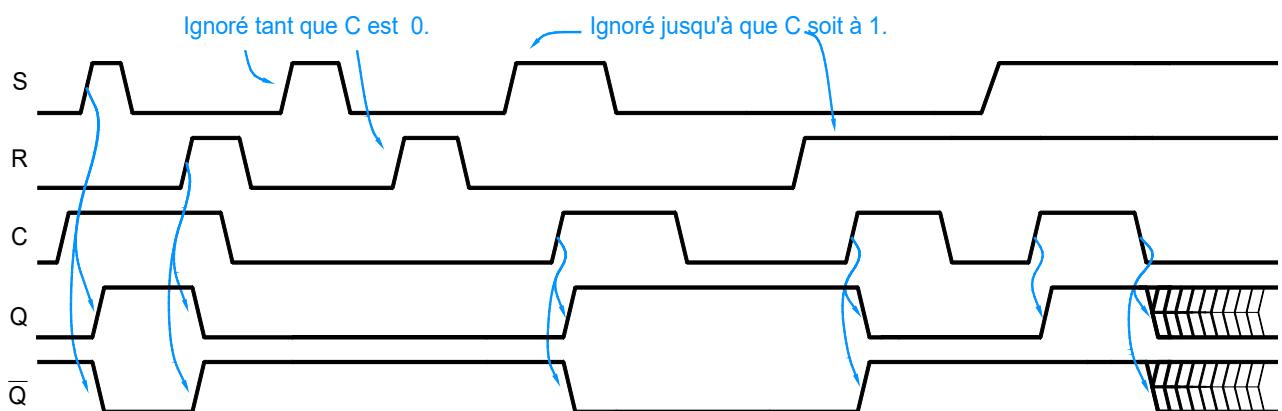


Figure 32 Chronogramme d'une bascule R-S avec enable

L'élément mémoire D (D Latch)

L'élément mémoire D plus souvent nommé D Latch est une bascule R-S avec enable sur laquelle on a simplement ajouté un inverseur entre les entrées S et R, il ne reste donc plus que l'entrée appelée D. Ceci nous supprime la possibilité de mettre S et R simultanément à 1 en même temps, ce qui élimine les problèmes de métastabilité de la bascule R-S avec enable. On peut voir le schéma logique d'un D latch à la Figure 33(a).

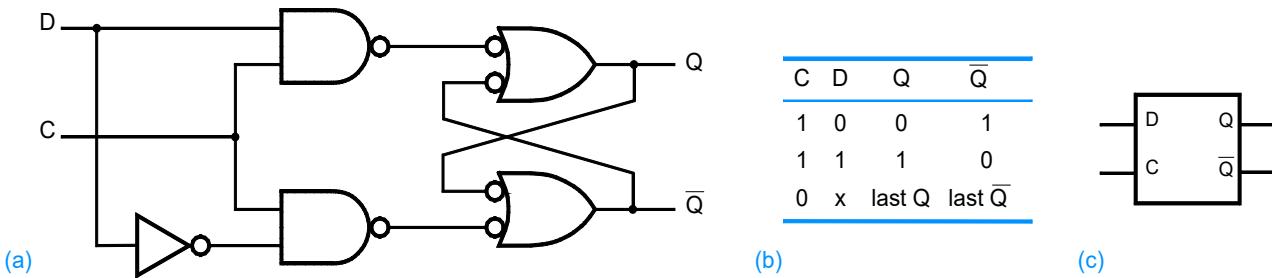


Figure 33 D latch: (a) schéma logique ; b) table de vérité ; c) symbole logique.

On peut voir dans la Figure 34 ci-dessous le chronogramme d'un D Latch pour des variations d'entrées données. Le fonctionnement est très simple, tant que C est à l'état logique haut, ce qui est sur l'entrée D se retrouve sur la sortie Q, et lorsque C est à l'état logique bas, la sortie mémorise le dernier état, même en cas de changement sur l'entrée D.

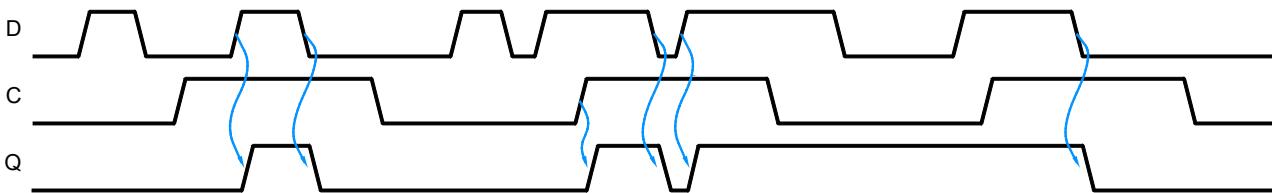


Figure 34 Chronogramme d'un D Latch pour des variations d'entrées données

SIGNAL D'HORLOGE ET BASCULES SYNCHRONES

Les circuits numériques peuvent fonctionner de façon synchrone ou asynchrone. Dans les systèmes asynchrones, la sortie des circuits logiques peut changer d'état à tout moment quand une ou plusieurs entrées changent. Un système asynchrone est difficile à concevoir et à débugger.

Par contre dans un système synchrone, le moment exact où la sortie change d'état est commandé par un signal que l'on appelle couramment signal d'horloge. Ce signal est généralement un train d'ondes rectangulaires ou carrées, comme ceux de la Figure 35. Le signal d'horloge est habituellement distribué à tous les étages du système, de sorte que la plupart des sorties du système, sinon toutes, changent d'état seulement quand le signal d'horloge effectue une transition. Ces transitions, appelées fronts ou flancs. Quand le signal d'horloge passe de 0 à 1, on parle du front montant (transition positive), quand il passe de 1 à 0, on parle de front descendant (transition négative).

La majorité des systèmes numériques existants sont des machines synchrones du fait que les circuits synchrones sont plus simples à concevoir et à dépanner. Leurs sorties ne peuvent changer qu'à des instants précis bien connus. Autrement dit, tous les changements sont synchronisés avec les transitions du signal d'horloge.

La synchronisation orchestrée par des signaux d'horloge est réalisée au moyen de bascules synchrones qui ont été réalisées pour changer d'état au moment de la transition associée à un front ou à l'autre du signal d'horloge.

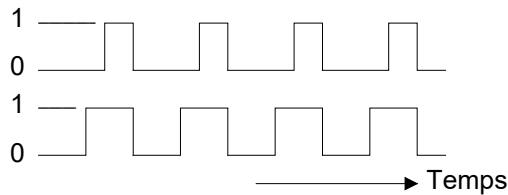


Figure 35 Signaux d'horloge

BASCULE D SYNCHRONE (D Flip-Flop)

Une bascule D synchrone également appelé D flip-flop est réalisée avec deux D Latch, comme illustré dans la Figure 36(a), ceci pour créer un circuit qui mémorise la valeur sur l'entrée D uniquement au moment du flanc montant du signal d'horloge (CLK).

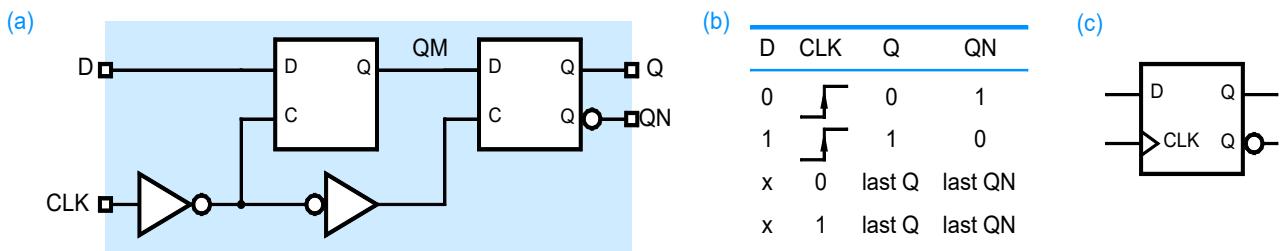


Figure 36 Bascule D synchrone : (a) circuit avec 2 D Latches ; (b) table de vérité ; (c) symbole logique.

Le premier latch appelé master, est ouvert (transparent) et suit l'entrée D lorsque CLK est 0. Lorsque le signal d'horloge CLK passe à 1, le master est fermé, et sa sortie est transférée au deuxième latch appelé slave. Le slave est donc ouvert pendant tout le temps où CLK est à 1, mais le master est fermé à ce moment-là. Le chronogramme de la Figure 37, illustre bien le fonctionnement interne d'une bascule D synchrone formée de deux D Latch.

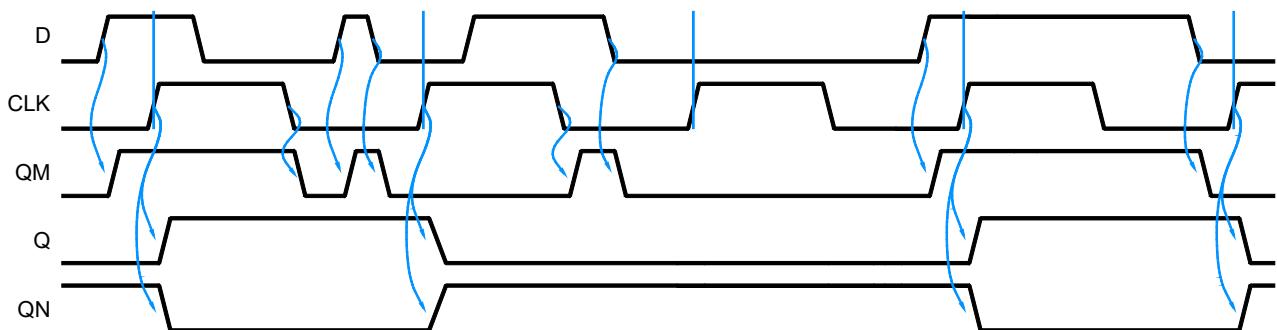


Figure 37 Chronogramme interne de la bascule D synchrone de la Figure 36

Le triangle sur l'entrée CLK du symbole logique de la bascule D indique que le fonctionnement de la bascule est sensible au flanc montant du signal d'horloge.

En résumé, le fonctionnement de la bascule D est donc très simple: La sortie Q prend l'état de l'entrée D à l'instant du front montant de CLK. Autrement dit, le niveau actuellement sur D se retrouvera mémorisé sur la sortie Q à l'instant du front montant. Le chronogramme de la Figure 38 illustre cette situation.

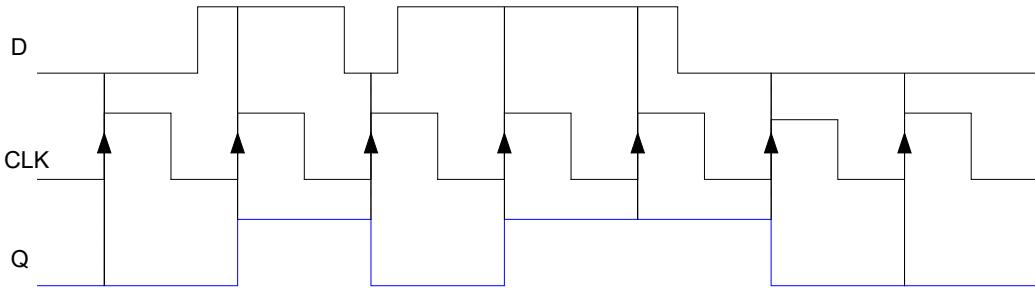


Figure 38 Chronogramme d'une bascule D.

ENTRÉES ASYNCHRONES

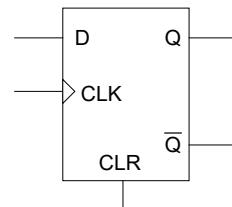
Dans les bascules synchrones précédentes, nous avons parlé d'entrées de commande pour les entrées S, C, et D. Ces entrées sont également qualifiées de synchrones étant donné que la sortie de la bascule est synchronisée par le signal d'horloge. Comme nous l'avons vu, les entrées de commande synchrones sont utilisées concurremment à un signal d'horloge qui déclenche la réponse de la bascule.

La majeure partie des bascules synchrones possèdent en plus des entrées asynchrones qui agissent indépendamment des entrées synchrones et du signal d'horloge. On a recours à de telles entrées pour forcer en tout temps la remise à 1 ou à 0 de la bascule, quelles que soient les conditions des entrées. Une autre façon de présenter ces entrées est de dire que ce sont des, entrées prioritaires, qui imposent un état à la bascule malgré les commandes lancées par les autres entrées.

La Figure 39 illustre une bascule D munie d'une entrée asynchrone désignée CLR. C'est une entrée active au niveau HAUT. La table à gauche de ce symbole résume la réaction de la sortie de la bascule.

D	CLK	CLR	Q
X	X	1	0
0	↑	0	0
1	↑	0	1
X	0	0	last Q
x	1	0	last Q

a) Table de vérité



b)

Figure 39 Table de vérité et symbole logique d'une bascule D avec CLEAR asynchrone actif haut.

Il est important de se rendre compte que les entrées asynchrones sont des niveaux de tension continue (CC). Cela veut dire que si CLR est gardé au niveau 1, la bascule restera dans l'état Q = 0 quoiqu'il arrive aux autres entrées. Le plus souvent, toutefois, on applique momentanément à ces entrées une impulsion pour remettre la bascule à 1 ou à 0.

De nombreuses bascules synchrones fabriquées sous forme de circuits intégrés possèdent deux entrées asynchrones, d'autres n'ont qu'une. Certaines bascules ont des entrées asynchrones qui sont vraies au niveau HAUT plutôt qu'au niveau BAS. Désignation des entrées asynchrones Dans leurs fiches techniques, les fabricants de CI emploient différentes désignations pour les entrées asynchrones, et malheureusement aucune n'a été normalisée. Voici les désignations que vous êtes susceptibles de rencontrer dans les schémas et les fiches techniques anglaises des fabricants. Pour la mise à l'état 1 : SET, PRESET (PRE), pour la mise à l'état 0 : CLEAR (CLR), RESET.

CONSIDÉRATIONS SUR LA SYNCHRONISATION DES BASCULES

Les fabricants de bascules intégrées spécifient quelques paramètres et caractéristiques de synchronisation importants qu'il importe de prendre en considération avant d'utiliser une bascule dans un circuit pratique.

La métastabilité

Les bascules peuvent entrer dans des états métastables lorsque plusieurs entrées changent simultanément, c'est-à-dire qu'elles peuvent n'offrir une sortie stable qu'au bout d'un temps arbitrairement long. Exemple : entrée d et clk changement simultanément.

Temps de stabilisation (setup time) et temps de maintien (hold time)

Deux exigences de synchronisation doivent être respectées pour qu'une bascule synchrone réponde correctement à ses entrées de commande lorsqu'arrive le front déclencheur de CLK. Ces exigences sont représentées sous forme graphique sur la Figure 40 pour une bascule déclenchée par un front montant.

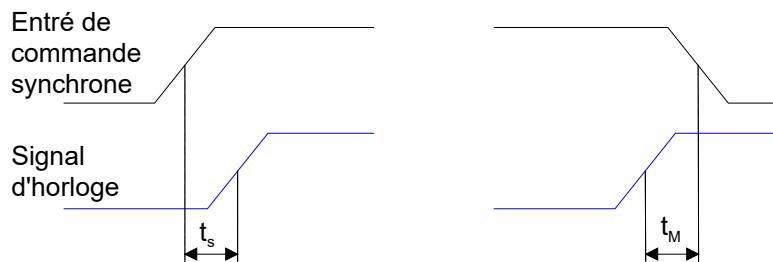


Figure 40 Les entrées de commande doivent rester stables pendant a) un temps ts avant la transition active de l'horloge, et b) un temps tm après la transition active de l'horloge.

- Le temps de stabilisation (ou t_{setup} dans les databook), ts , est l'intervalle qui précède immédiatement le front déclencheur du signal d'horloge, pendant lequel l'entrée synchrone doit être gardée au niveau approprié. Les fabricants de CI spécifient généralement la durée de stabilisation minimale admissible. Si on ne respecte pas ce temps, il n'est pas garanti que la bascule réponde correctement à l'arrivée du front.
- Le temps de maintien (ou t_{hold} dans les databook), tm , est l'intervalle qui suit immédiatement le front déclencheur du signal d'horloge pendant lequel l'entrée synchrone doit être gardée au niveau approprié. Les fabricants de CI spécifient généralement le temps minimal acceptable. Si on ne respecte pas ce temps, la bascule ne sera pas déclenchée correctement.

D'après ce qui vient d'être dit pour éviter une métastabilité à la sortie des bascules D, on voit que l'entrée de commande doit rester stable (inchangée) pendant une durée égale à la somme du temps ts , qui précède le front déclencheur, et du temps tm , qui suit ce même front. Les bascules de CI ont des temps ts et tm de l'ordre des nanosecondes.

Temps de propagation

Chaque fois qu'un signal doit changer l'état d'une bascule, on observe un retard entre le moment où le signal est appliqué et le moment où le changement apparaît à la sortie. Une illustration des retards de propagation affectant la réponse à un front montant de signal d'horloge nous est fournie à la Figure 41. Notez que ces retards sont mesurés entre les points à mi-hauteur (50 %) des formes d'ondes d'entrée et de sortie. Les mêmes genres de retards se produisent en réponse à des signaux placés sur les entrées asynchrones (SET et CLR). Sur les fiches techniques des fabricants, on trouve généralement les retards de propagation affectant la réponse à toutes les entrées, ainsi que les valeurs maximales de t_{PLH} et t_{PHL} .

Les bascules modernes dans les CI ont des retards de propagation sont de l'ordre de quelques nanosecondes. Les valeurs de t_{PLH} et t_{PHL} ne sont pas égales et augmentent proportionnellement avec le nombre d'étages logiques pilotés par la sortie Q. Les retards de propagation dans les bascules jouent un rôle important dans les circuits logiques.

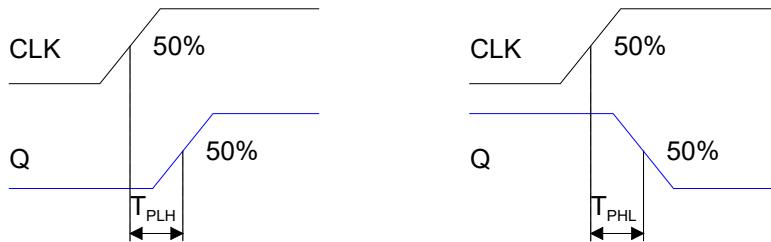


Figure 41 Retards de propagation dans une bascule.

BASCULE T (Toggle flip flop)

Pour réaliser une bascule T, il suffit de connecter la sortie \bar{Q} sur l'entrée D d'une bascule D, comme sur la Figure 42a), ou alors si on n'a pas de sortie \bar{Q} , de relier la sortie Q sur l'entrée d'un inverseur, dont la sortie sera connectée sur l'entrée D. Le fonctionnement de la bascule T est très simple: A chaque flanc montant du signal d'horloge, la sortie Q change d'état. Le chronogramme de la Figure 42c) illustre ce fonctionnement.

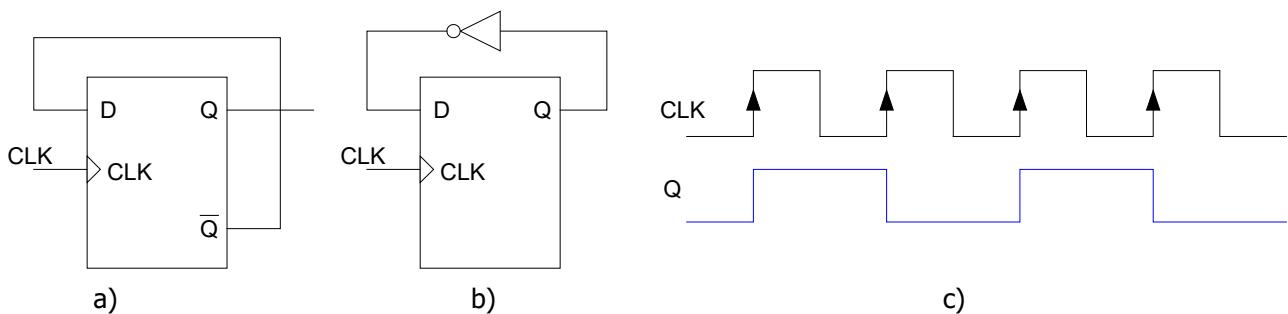


Figure 42 Réalisations d'une bascule T (toggle flip-flop) ; c) Chronogrammes d'une bascule T.

Description d'une bascule D avec reset asynchrone en VHDL

Dans tous les systèmes séquentiels, on doit utiliser un process, et pour pouvoir identifier un flanc du signal d'horloge, on utilise l'attribut `event`, qui signifie événement. `clk'EVENT` revoie une valeur booléenne TRUE lors d'un flanc (montant ou descendant) du signal d'horloge. On peut également utiliser la fonction `rising_edge(clk)`, qui renvoie une valeur booléenne TRUE dans le cas d'un flanc montant du signal, en général `clk`.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dff is
    Port ( clk          : in std_logic;
            reset_n     : in std_logic;
            d           : in std_logic;
            q           : out std_logic);
end dff;

architecture behavioral of dff is

BEGIN

p1:PROCESS (clk, reset_n) BEGIN
    IF reset_n = '0' THEN --reset asynchrone actif bas
        q <= '0';
    ELSIF (clk'EVENT AND clk = '1') THEN --flanc montant du signal clk
        q <= d;
    END IF;
END PROCESS;
end behavioral;
```

Fonctionnement décrit:

- Si `reset_n` est à l'état 0 alors la sortie `q` prend l'état 0 (si la condition est vraie, on sort du process, donc le reset est prioritaire et asynchrone).
- Sinon (sous-entendu `reset` est à 1) si on a un flanc montant sur `clk`, alors la sortie `q` prend la valeur de l'entrée `d`.
- Si aucune condition n'est vraie, rien ne change.

Ceci est bien la description du fonctionnement de la bascule D.

Remarque importante :

Comme déjà mentionné précédemment, en logique combinatoire, on doit spécifier tous les cas possibles sur les entrées, sous peine de voir le synthétiseur introduire des éléments mémoire non désirés.

En logique séquentielle par contre, on ne doit spécifier que les cas où le signal change d'état, en effet une bascule D peut garder le même état de sortie plusieurs périodes d'horloge consécutives, selon le calcul de l'état futur par le système combinatoire.

Chablon pour la description d'un système séquentiel synchrone en VHDL

Tous les systèmes séquentiels doivent se décrire avec le chablon ci-dessus. Bien entendu, on peut introduire d'autres instructions VHDL séquentielles dans les zones rouges. Cependant, les lignes en noir ne doivent jamais changer.

```
p1:PROCESS (clk, reset_n) BEGIN
    IF reset_n = '0' THEN
        RESET ASYNCHRONE DE TOUTES LES BASCULES ASSIGNEES DANS LA
        CLAUSE ELSIF CI-DESSOUS
    ELSIF (clk'EVENT AND clk = '1') THEN
        POUR UNE OU PLUSIEURS ASSIGNATIONS D'UN MÊME SIGNAL DANS CETTE
        CLAUSE ELSIF, UNE BASCULE D SYNCHRONE EST GENÉRÉE (une par
        bit) LORS DE LA SYNTHÈSE
    END IF;
END PROCESS;
```

Règles du design synchrone

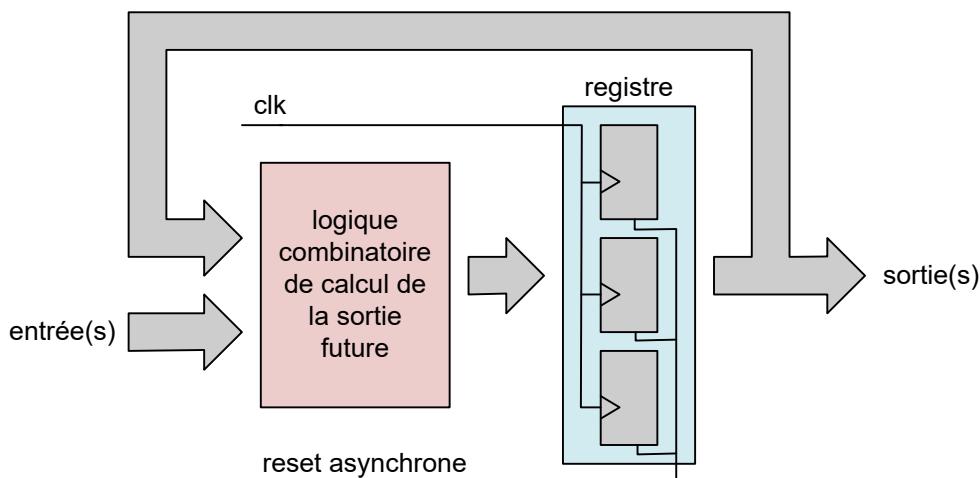
Design totalement synchrone => **toutes les bascules du circuit synchronisées (clockées) par le même signal d'horloge**, par conséquent :

- Pas de signaux d'horloges dérivées
- Pas de clock gating
- Reset asynchrone de toutes les bascules du circuit
- Pas de boucles combinatoires
- Pas d'utilisation latchs asynchrones
- Synchroniser les signaux échangés entre domaines de clock différents

Structure de base des systèmes séquentiels synchrones

Tous les systèmes séquentiels synchrones de base peuvent être représentés avec le schéma bloc (RTL) ci-dessous. Ce schéma est composé de deux parties principales :

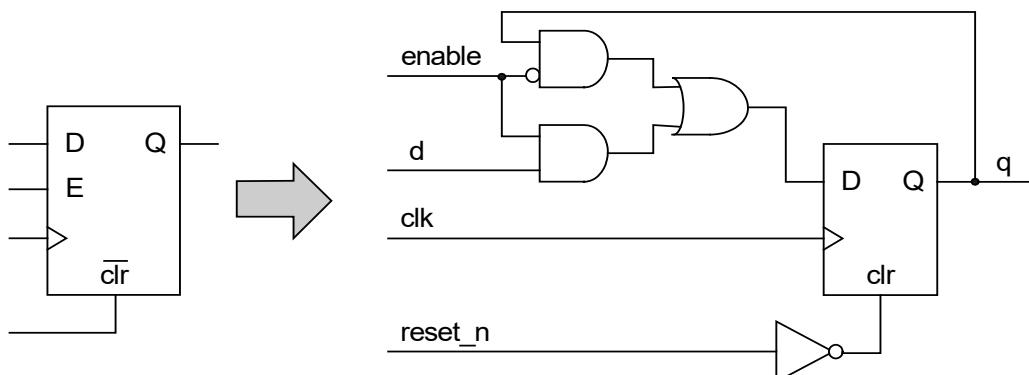
1. Une partie de logique combinatoire qui va « calculer » la sortie future en fonction de l'état des entrées et de l'état courant des sorties.
2. Un registre tampon composé de bascule(s) D synchrone(s) en parallèle. Le nombre de bascules correspond au nombre de sorties.



Bascule D avec enable

Dans tous les systèmes séquentiels, lorsqu'on veut que le clock d'une bascule soit inactif, c'est à dire lorsqu'on veut que la sortie ne change pas d'état à chaque transition du signal d'horloge, on n'a en aucun cas le droit de couper le clock (clock gating), on utilise dans ce cas un signal supplémentaire appelé généralement enable (autorisation, activation).

Généralement, les bascules sont représentées symboliquement avec une entrée enable supplémentaire, ce qui n'est pas forcément faux, mais il faut être conscient, qu'à l'intérieur nous avons une simple bascule D avec de la logique combinatoire (en fait un multiplexeur) permettant de gérer le signal enable. On peut voir dans la figure ci-dessous, le symbole logique d'une bascule D avec enable, ainsi que le schéma logique d'une telle bascule.



Le fonctionnement de cette bascule est le suivant:

- si enable est à l'état logique bas, la sortie q est rebouclée sur l'entrée D de la bascule, donc pas de changement de la sortie au prochain flanc du signal d'horloge.
- si enable est à l'état logique haut, le signal d arrive sur l'entrée D de la bascule, donc au prochain flanc du signal d'horloge, la sortie q prend la valeur de l'entrée d, ce qui est le fonctionnement normal de la bascule.

Description VHDL

```

ENTITY dff_ebl IS
    Port ( clk          : in std_logic;
           reset_n      : in std_logic;
           d            : in std_logic;
           ebl          : in std_logic;
           q            : out std_logic);
END dff_ebl;

ARCHITECTURE behavioral OF dff_ebl IS

BEGIN
    p1:PROCESS (clk, reset_n) BEGIN
        IF reset_n = '0' THEN --reset asynchrone actif bas
            q <= '0';
        ELSIF (clk'EVENT AND clk = '1') THEN --flanc montant du signal clk
            IF ebl = '1' THEN --signal enable actif haut
                q <= d;
            END IF;
        END IF;
    END PROCESS;
END behavioral;

```

RESET ASYNCHRONE OU SYNCHRONE

L'initialisation (reset) d'un système peut se faire de manière asynchrone ou synchrone. On doit dans tous les cas utiliser un reset asynchrone, mais, selon les cas, les deux méthodes peuvent être utilisées en parallèle.

Reset asynchrone

Dans les systèmes synchrones tout ce qui est asynchrone est à éviter. Toutefois lors de l'enclenchement (mise sous tension) il est nécessaire de mettre le système dans un état défini de départ par un POWER ON RESET (POR). Normalement dans un système numérique le reset asynchrone n'est utilisé qu'une seule fois au démarrage du système.

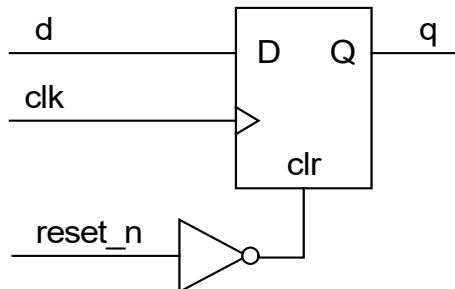
Ce n'est pas le début du reset asynchrone qui pose un problème mais la fin. En effet si le reset devient inactif au flanc actif de l'horloge le système peut avoir certaines parties encore sous l'influence du reset alors que d'autres ne le sont plus. Le système peut se trouver dans un état indéterminé.

Les circuits logiques programmables incluent tous une entrée dédiée au reset asynchrone, avec une connexion prévue vers toutes les bascules du circuit.

Exemple bascule D avec reset asynchrone :

```
p1:PROCESS (clk, reset_n)
BEGIN
    IF reset_n = '0' THEN          --reset asynchrone actif bas
        q <= '0';
    ELSIF (clk'EVENT AND clk = '1') THEN
        q <= d;
    END IF;
END PROCESS;
```

Circuit logique synthétisé:



Reset synchrone

Lorsque l'on doit remettre à zéro certains registres ou bascules durant le fonctionnement d'un système numérique, on ne va pas utiliser le reset asynchrone (car il est asynchrone !) et on va devoir ajouter un signal que l'on va appeler reset synchrone.

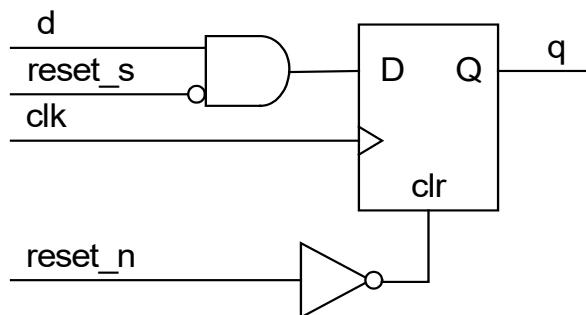
Le reset synchrone présente dans certains cas l'inconvénient d'augmenter les délais de propagations ce qui entraîne une diminution de la fréquence limite de fonctionnement. Ce n'est pas forcément le cas pour certaines FPGA, car la logique nécessaire au reset synchrone est déjà présente.

La remise à zéro synchrone est indispensable, dans le cas où l'on veut régulièrement remettre à zéro des bascules pendant le fonctionnement du système, ceci afin d'éviter les problèmes de métastabilité.

Exemple bascule D avec reset synchrone et asynchrone :

```
p1:PROCESS (clk, reset_n)
BEGIN
    IF reset_n = '0' THEN          --reset asynchrone actif bas
        q <= '0';
    ELSIF (clk'EVENT AND clk = '1') THEN
        IF reset_s = '1' THEN      --reset synchrone actif haut
            q <= '0';
        ELSE
            q <= d;
        END IF;
    END IF;
END PROCESS;
```

Circuit logique synthétisé:



Les Bancs de test VHDL (testbench)

La plupart des simulateurs logiques permettent de créer des stimuli pour vérifier le comportement d'un modèle. Souvent un langage de commande donne à l'utilisateur la possibilité d'automatiser, par la création de fichiers contenant des instructions de ce langage, l'enchaînement des opérations nécessaires au test.

Langage de modélisation, VHDL est également un langage de simulation. Il contient tous les éléments nécessaires à la création de stimuli, et, surtout, à l'exploitation des résultats.

Dans ce qui suit nous n'apprendrons rien de nouveau concernant le langage, nous verrons comment utiliser les éléments connus pour construire des boîtes noires autonomes, qui en pilotent d'autres et en observent les réactions. Ces programmes de test sont traditionnellement nommés bancs de test ou testbench.

La première règle, que nous avons déjà évoquée, consiste à séparer nettement, dans une construction hiérarchique, les modules synthétisables de ceux qui servent à les essayer. L'objectif que nous poursuivons ici est de construire un programme de test qui nous permette de contrôler le bon fonctionnement d'un module synthétisable.

Il existe deux méthodes principales pour la réalisation d'un banc de test :

- A trois fichiers ou plus, où le testbench instancie le module à tester (UUT) et un fichier générateur (GEN), voir la Figure 43.
- A deux fichiers, le testbench instancie le module à tester (UUT), voir la Figure 44.

Simulation à trois fichiers ou plus

Dans le cas de simulation VHDL à trois fichiers, il est nécessaire de décomposer la simulation en plusieurs modules (hiérarchie). On aura au minimum 3 fichiers différents, à savoir : le module à simuler (UUT), le testbench et le fichier générateur de stimuli (Figure 43).

Le testbench fait l'interconnexion des signaux entre le générateur et le module à simuler. Le fichier générateur comporte des affectations qui vont stimuler le module à simuler. Dans le fichier générateur, la vérification de l'état attendu des sorties est obligatoire. Il doit y avoir une vérification automatique du comportement du module qui est en cours de simulation. La simple lecture du chronogramme n'est pas une vérification suffisante. Souvent, le concepteur doit faire plusieurs corrections successives du module qui est en cours de simulation. La vérification par une lecture est alors souvent incomplète. De plus la simulation automatique permet de fournir un justificatif des tests réalisés.

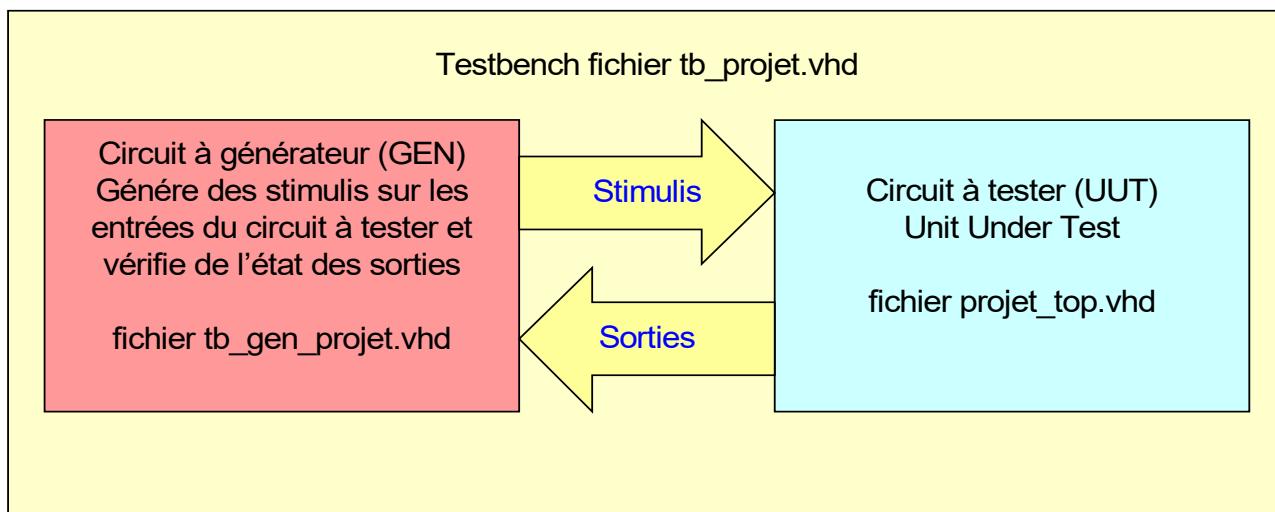


Figure 43 Structure hiérarchique de base pour la simulation à trois fichiers ou plus.

Simulation à deux fichiers

Dans le cas de simulation VHDL, le testbench instancie le module à tester (UUT), génère de signaux (stimuli) sur les entrées du module à tester et contrôle l'état des sorties du module à tester. La vérification de l'état attendu des sorties est obligatoire. Il doit y avoir une vérification automatique du comportement du module qui est en cours de simulation. La simple lecture du chronogramme n'est pas une vérification suffisante. Souvent, le concepteur doit faire plusieurs corrections successives du module qui est en cours de simulation. La vérification par une lecture est alors souvent incomplète. De plus la simulation automatique permet de fournir un justificatif des tests réalisés.

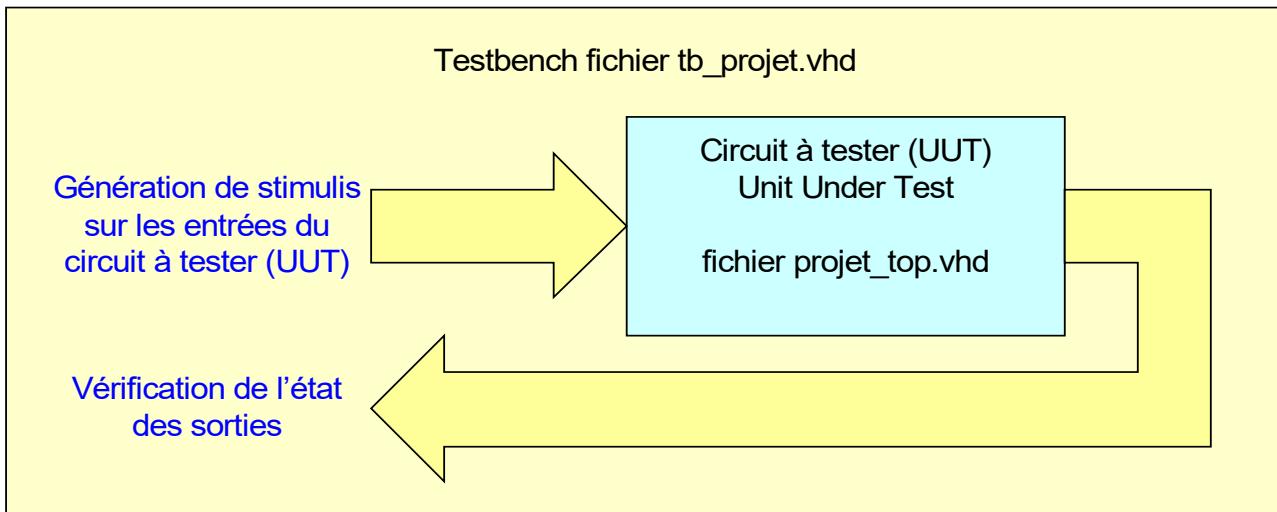


Figure 44 Structure hiérarchique de base pour la simulation à deux fichiers.

Principe de base d'une simulation.

Le principe de base d'un cycle de simulation est le suivant :

- Mise à jour de l'état des entrées du module à tester UUT (Unit Under Test).
- Attente d'un certain temps pour calculer le nouvel état des sorties du UUT.
- Vérification automatique de l'état des sorties avec l'état attendu avec génération automatique d'un message d'erreur.

Ce cycle est répété autant de fois que nécessaire pour garantir la détection de toutes les erreurs du module à simuler. En principe tous les cas sont testé, mais de façon « intelligente ». Par exemple si on teste un compteur de 8 bits, on ne va pas simuler les 256 états du compteurs, mais le reset, les trois premier états, puis contrôler qu'il revienne bien à zéro et qu'il continue sa séquence serait suffisant pour garantir le fonctionnement de ce compteur.

La description du fichier de simulation ne devrait pas être faite par la même personne qui fait la description synthétisable, en effet si la description comporte une erreur de conception, si la même personne génère le fichier de simulation, elle risque de la créer par rapport au circuit qu'elle a décrit, de ne pas découvrir l'erreur. Le fichier de simulation doit correspondre au cahier des charges et non pas au fichier VHDL synthétisable. Cela paraît évident, mais c'est là une source fréquente d'erreurs de fonctionnement.

Nous verrons dans l'exemple ci-dessous qu'il est possible de générer de manière automatique la séquence de test avec deux processus et une boucle for, pour la génération automatique des signaux d'horloge, une pour le déroulement séquentiel du test : Voici un chablon d'un fichier banc de test avec comme base de référence le test d'un décodeur BCD to 7 Segments :

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE testbench_architecture OF testbench IS

--Declaration du composant UUT (Unit Under Test)
COMPONENT bcd7seg
    PORT(
        BCD      : IN  std_logic_vector(3 downto 0);
        aff_ebl  : OUT std_logic;
        aff      : OUT std_logic_vector(6 downto 0)
    );
END COMPONENT;

--Signaux locaux pour instanciation composant UUT
--Inputs
SIGNAL BCD : std_logic_vector(3 downto 0);
--Outputs
SIGNAL aff_enable : std_logic;
SIGNAL aff : std_logic_vector(6 downto 0);

--Signaux locaux propres au banc de test
SIGNAL sim_end      : BOOLEAN      := FALSE;
SIGNAL mark_error   : std_logic    := '0';
SIGNAL error_number : integer      := 0;
SIGNAL clk_gen       : std_logic    := '0';

BEGIN

--Instanciation du composant UUT
uut: bcd7seg PORT MAP(
    BCD          => BCD,
    aff_enable   => aff_enable,
    aff          => aff
);

--***** PROCESS "clk_gengen" *****
clk_gengen: PROCESS
BEGIN
    IF sim_end = FALSE THEN
        clk_gen <= '1', '0' AFTER 1 ns;
        --clk      <= '1', '0' AFTER 5 ns, '1' AFTER 17 ns;
        wait for 25 ns;
    ELSE
        wait;
    END IF;
END PROCESS;

--***** PROCESS "run" *****
run: PROCESS

    PROCEDURE sim_cycle(num : IN integer) IS
    BEGIN
        FOR index IN 1 TO num LOOP
            wait until clk_gen'EVENT AND clk_gen = '1';
        END LOOP;
    END sim_cycle;

```

```
--***** PROCEDURE "init" *****
--fixer toutes les entrees du module à tester (UUT)
PROCEDURE init IS
BEGIN
--completer ici
END init;

--***** PROCEDURE "test_signal" *****
PROCEDURE test_signal(signal_test,value: IN std_logic; erreur : IN integer) IS
BEGIN
    IF signal_test/= value THEN
        mark_error <= '1', '0' AFTER 1 ns;
        error_number <= erreur;
        ASSERT FALSE REPORT "Etat du signal non correct" SEVERITY WARNING;
    END IF;
END test_signal;

--***** PROCEDURE "test_vecteur8" *****
PROCEDURE test_vecteur8(signal_test, value: IN std_logic_vector (7 DOWNTO 0);
erreur : IN integer) IS
BEGIN
    IF signal_test/= value THEN
        mark_error <= '1', '0' AFTER 1 ns;
        error_number <= erreur;
        ASSERT FALSE REPORT "Etat du signal non correct" SEVERITY WARNING;
    END IF;
END test_vecteur8;

BEGIN --debut de la simulation temps t=0ns

    init; --appel procedure init
    ASSERT FALSE REPORT "Debut de la simulation" SEVERITY NOTE;
    --debut des tests

    sim_end <= TRUE;
    wait;

END PROCESS;
END testbench_architecture;
```

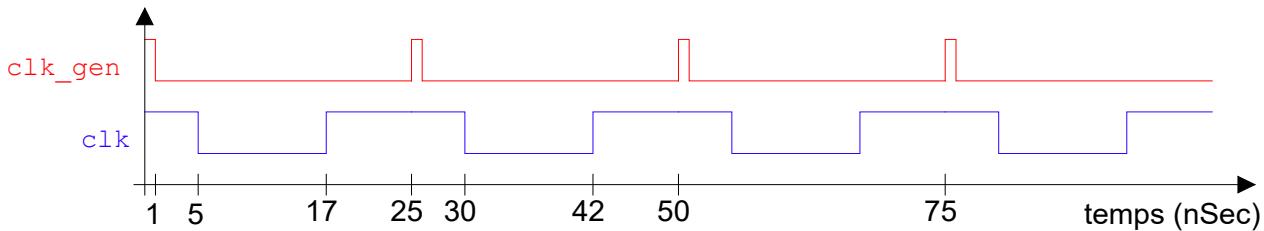
Analyse du fichier testbench chablon

Le fichier ci-dessus est un chablon, qui contient les partie principales (base de temps et signal d'horloge, procédure d'initialisation, procédure de test, et enfin partie principale) pour construire un testbench, mais il doit être adapté et étendu spécifiquement pour chaque application.

Il est important de se souvenir qu'un processus se déroule dans temps nul. C'est pourquoi nous allons avoir besoin de l'instruction wait pour pourvoir échelonner nos tests dans le temps.

Base de temps et signal d'horloge

Pour la base de temps, nous avons besoin d'un process clk_gengen, qui génère les signaux clk_gen et clk, tant que le signal booléen sim_end = False. On peut voir une représentation des deux signaux ci-dessous :



Le signal `clk_gen` est la base de temps du `test_bench`, mais il ne sert à rien sans la procédure `sim_cycle`, qui se trouve dans le process principal (`run`) du générateur.

La procédure `sim_cycle(num)` avec `num` de type `integer`, utilise une boucle `FOR..LOOP`, qui s'exécute `num` fois. Le contenu de la boucle attend simplement un flanc montant sur `clk_gen`.

En résumé l'appel de la procédure `sim_cycle(num)` nous fait avancer la simulation d'un temps égal à `num` fois la période de `clock_gen`, dans notre exemple 25 ns.

Le signal `clk` sera le signal d'horloge connecté sur le module à tester. Si l'on veut tester un système combinatoire sans signal d'horloge, il suffit de mettre la ligne suivante en commentaire (--) :

```
--clk <= '1', '0' AFTER 5 ns, '1' AFTER 17 ns;
```

Procédure d'initialisation (init)

La procédure `init` sert à fixer une valeur par défaut sur toutes les sorties du générateur (par conséquent toutes les entrées du module à tester), ceci afin de ne pas avoir d'état U (indéfini) lors de la simulation. C'est la première procédure appelée dans la zone principale au temps t = 0ns.

Il faut modifier cette partie dans tous les bancs de test, afin d'initialiser toutes les entrées du module à tester.

Procédures `test_signal` et `test_vecteur`

Ces procédures servent à tester de façon « automatique » si l'état d'un signal à un moment donné correspond à la valeur attendue. On envoie 3 paramètres lors de l'appel de cette procédure :

1. Nom du signal à tester.
2. Valeur que devrait avoir le signal de type `std_logic` ou `std_logic_vector`.
3. Numéro d'erreur de type `integer`.

Deux cas sont possibles lors de l'appel de cette procédure :

1. Le signal correspond à la valeur attendue, alors rien ne se passe
2. Le signal ne correspond pas à la valeur attendue, alors le signal `mark_error` passe à 1 pendant 1ns, et le signal `error_number` prend la valeur du troisième paramètre `erreur`, de plus un message (Etat du signal non correct) est envoyé sur la fenêtre principale du simulateur.

Ces procédures peuvent être modifiées, ou dupliquées et modifiées à souhait selon les tailles des signaux à tester (par exemple test d'un vecteur de taille différente), on peut même en ajouter plusieurs autres à base de boucles par exemple, afin de rendre les tests les plus simples possibles. Toute la puissance du VHDL peut être utilisée pour la conception de bancs de test.

Partie principale

Il est bon de se rappeler que les instructions à l'intérieur d'un processus se déroulent de manière séquentielle, c'est à dire les unes après les autres. Ensuite dans un processus toutes les valeurs sont mises à jour à la fin du Processus. Enfin particularité des Processus pour la simulation, ils n'ont pas de liste de sensibilité, ce que veut dire que le simulateur les exécute en permanence jusqu'au moment où il rencontre une instruction `wait`.

La partie principale du générateur débute après le BEGIN du process run la séquence d'un test se déroule de la manière décrite ci-dessous :

- On commence par appeler la procédure `init`, qui fixe l'état des entrées du module à tester.
- On change l'état des entrées du module à tester.
- On fait avancer le temps de la simulation avec la procédure `sim_cycle`.
- On contrôle la valeur des signaux de sortie du module à tester par exemple avec les procédures `test_signal` ou `test_vecteur` adapté à la taille du signal à tester.
- On change une nouvelle fois les signaux d'entrée du module à tester.
- On fait avancer le temps.
- On teste une nouvelle fois les sorties.
- Ainsi de suite, jusqu'à ce que toutes les combinaisons possibles soient testées.
- On affecte ensuite la valeur `true` au signal `sim_end`, ce qui va rendre inactif le process `clk_gengen`.
- Et enfin on utilise l'instruction `wait` sans condition pour arrêter définitivement le process `run`.

Synchronisation des entrées externes asynchrones

Les systèmes numériques travaillent toujours de façon synchrone, c'est pourquoi il est nécessaire de synchroniser les entrées externes asynchrones avec le signal d'horloge (clk). En effet un changement de valeur du signal externe peut se produire à n'importe quel instant, c'est pour quoi on utilise des bascules D mises en série pour synchroniser une entrée externe. La Figure 45 nous montre une synchronisation à trois bascules D.

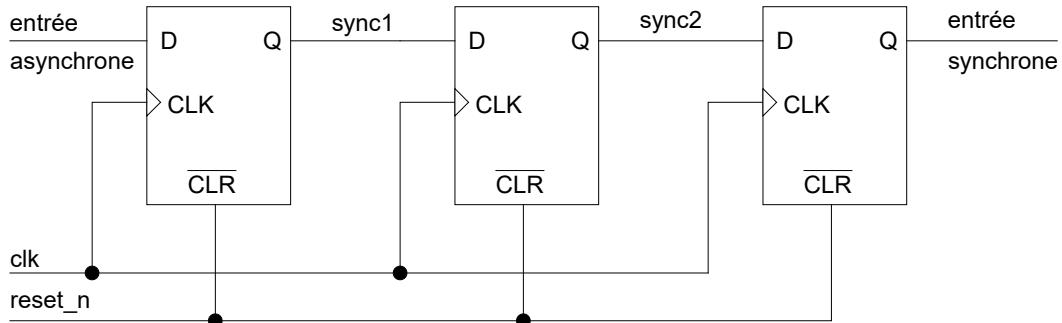


Figure 45 Synchronisation d'une entrée avec 3 bascules D

On utilise deux voire trois bascule pour supprimer le phénomène de métastabilité sur la première bascule dans le cas où le changement sur l'entrée asynchrone surviendrait exactement au moment du flanc montant du clock. Un exemple est donné dans le chronogramme ci-dessous :

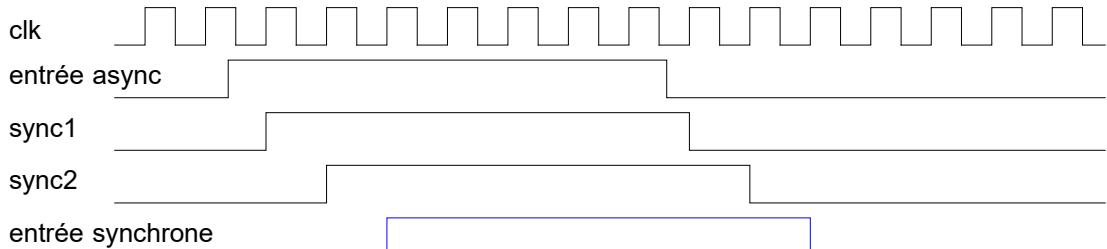


Figure 46 Chronogramme de la synchronisation d'un signal

Description VHDL

```

architecture synchronisation_A of synchronisation is

SIGNAL sync1      : std_logic;      --declaration des signaux internes
SIGNAL sync2      : std_logic;
SIGNAL entree_synchrone : std_logic;

BEGIN
P1:PROCESS(clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        sync1 <= '0';
        sync2 <= '0';
        entree_synchrone <= '0';
    ELSIF clk'EVENT AND clk = '1' THEN
        sync1 <= entree_asynchrone;
        sync2 <= sync1;
        entree_synchrone <= sync2;
    END IF;
END PROCESS;
end synchronisation_A;

```

Synchronisation et détection des flancs d'un signal externe asynchrone

Dans certains cas on n'aimerait pas uniquement synchroniser une entrée, mais également détecter un flanc montant sur cette entrée, on utilise le même schéma que précédemment, en ajoutant une porte ET avec une entrée inversée pour détecter soit le flanc montant, soit le flanc descendant du signal d'entrée asynchrone voir Figure 47.

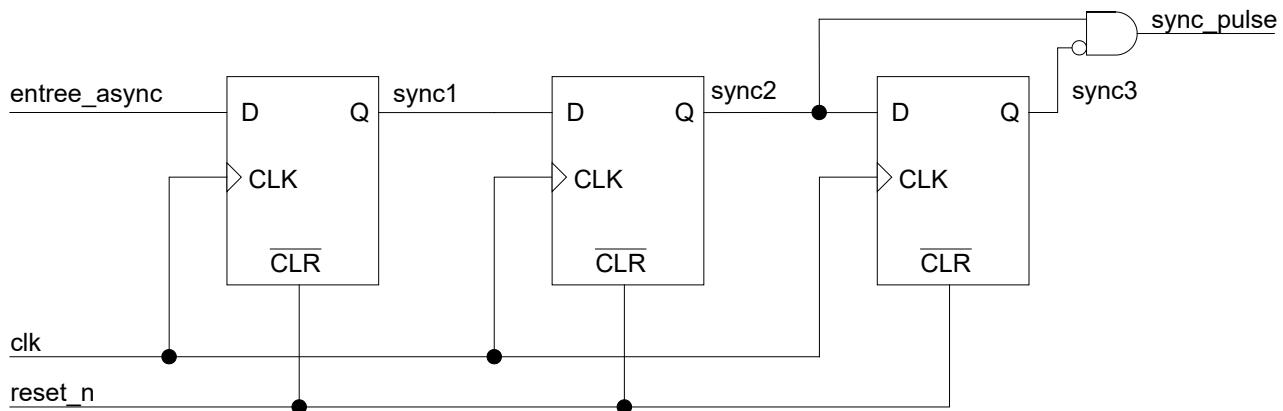


Figure 47 Synchronisation et génération d'une impulsion par flanc montant

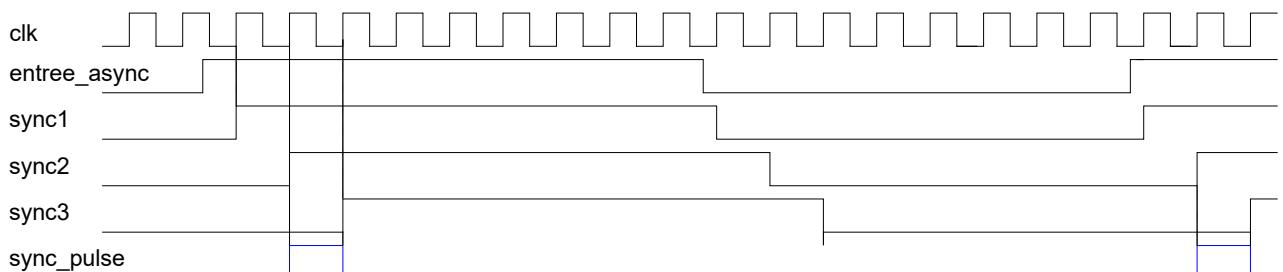


Figure 48 Chronogramme de la synchronisation et détection de flanc d'un signal externe

Description VHDL

Par rapport à la synchronisation simple, il suffit d'ajouter une fonction combinatoire, à savoir une porte AND avec une entrée inversée en utilisant soit les opérateurs logiques AND et NOT, ou alors l'instruction assignation conditionnelle, exemple :

```
sync_pulse <= sync2 AND (NOT sync3);
```

Ou alors:

```
sync_pulse <= '1' WHEN sync2 = '1' AND sync3 = '0' ELSE '0';
```

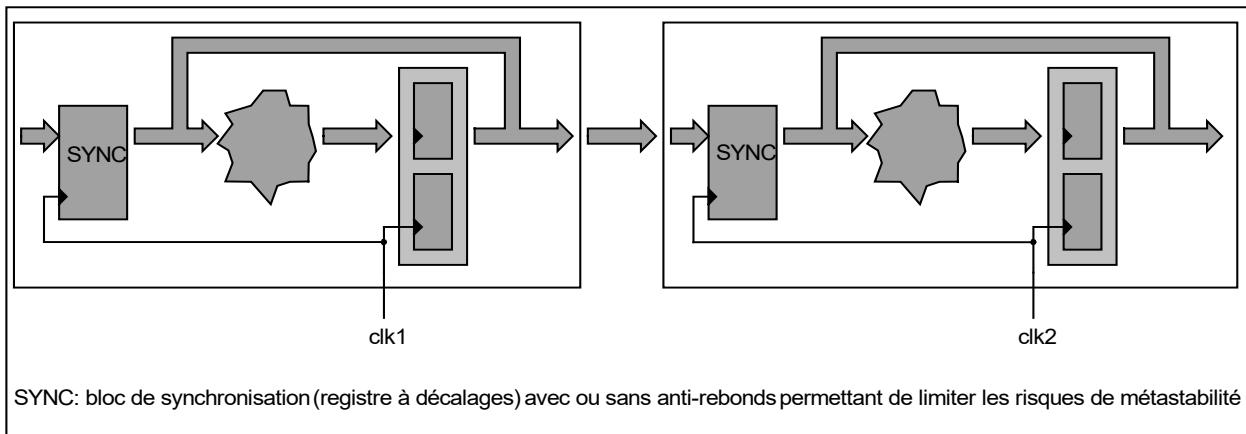
En changeant l'équation, on arrive facilement à détecter le flanc descendant d'un signal.

Pour détecter chaque transition (montante ou descendante) on utilisera la fonction OU Exclusif.

Système à plusieurs horloges

Il arrive fréquemment qu'un système numérique comporte plusieurs horloges asynchrones entre elles, on parle alors de domaines d'horloge multiples (multiple clock domain).

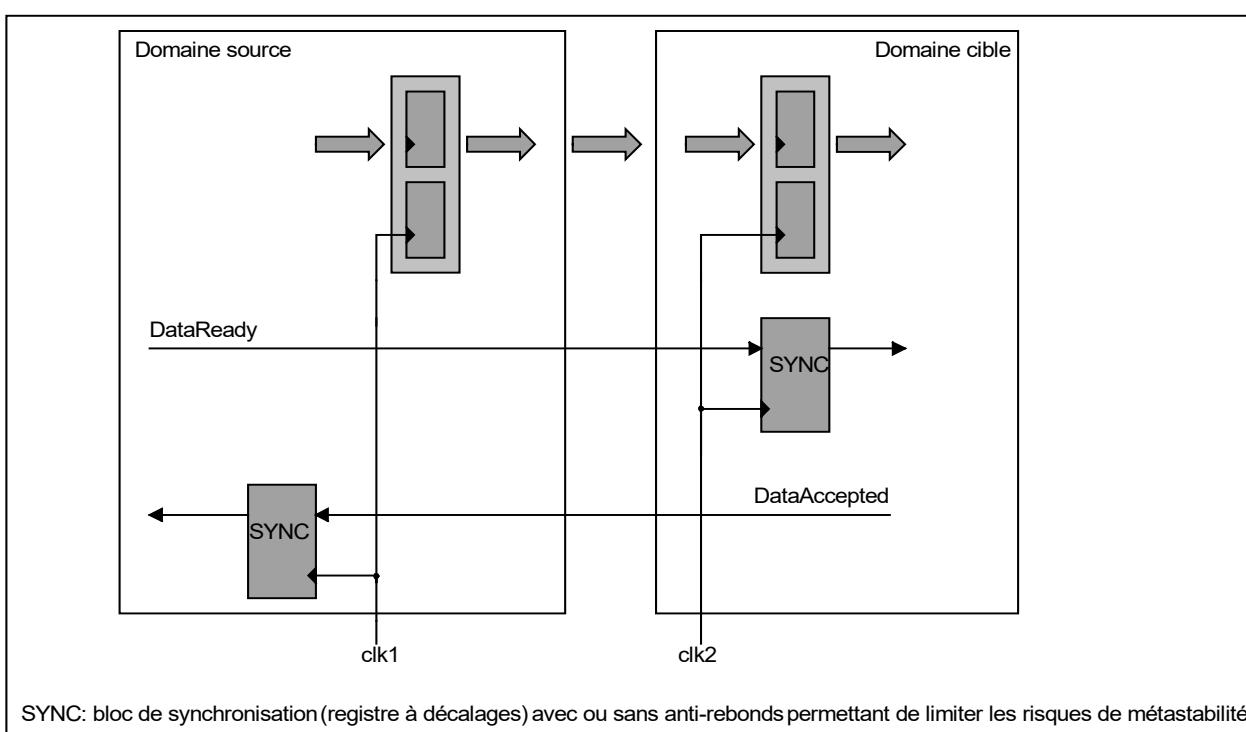
Dans ce cas, il faut traiter séparément les parties qui n'ont pas la même horloge. Des fonctions de synchronisation seront placées pour relier les différentes parties entre elles.



Handshaking

Le transfert de données entre deux domaines d'horloges peut se faire avec un système appelé handshaking. Le domaine source indique au domaine cible que des données sont disponibles, ce signal est synchronisé avec l'horloge cible.

Lorsque le domaine cible a pris les données, il envoie une information au domaine source lui indiquant la réception des données, ce signal est synchronisé avec l'horloge source.



Les Registres

Un registre est un ensemble de bascules synchrones capables de :

- De mémoriser une information binaire (nombre binaire).
- De transférer une information dans certaines conditions.
- De faire subir un traitement simple comme le décalage.

Le stockage, le transfert et le traitement sont réalisés sous le contrôle d'un signal de commande appelé horloge. Cette horloge commande toutes les bascules élémentaires.

C'est l'une des applications immédiates des bascules synchrones, essentiellement les bascules D.

Nous allons étudier différentes sortes de registres, en commençant par les registres tampon, puis en enchaînant avec les registres mémoire, pour terminer avec les registres à décalage.

Structures de base des registres

On distingue 4 structures de registres selon les modes d'entrée et de sortie des données binaires :

- Entrée série : les données binaires sont introduites les unes après les autres dans le registre. Chaque donnée binaire passe par la première bascule. Puis elle est décalée dans les bascules suivantes en synchronisme avec l'horloge.
- Entrée parallèle : toutes les données binaires sont introduites en même temps dans le registre.
- Sortie série : les données binaires enregistrées dans le registre sont fournies les unes après les autres au rythme de l'horloge.
- Sortie parallèle : toutes les données binaires enregistrées dans le registre sont disponibles au même instant.

Les combinaisons de ces structures donnent différents types de registres.

Registre Tampon (registre parallèle)

Un registre tampon est un registre avec entrée parallèle et sortie parallèle ayant le schéma de principe ci-contre. Ce registre, très courant en électronique numérique, est souvent utilisé pour :

- La synchronisation des entrées asynchrones (transfert d'information).
- La mémorisation de données.

On peut voir dans la Figure 49 la structure d'un registre tampon de 4 bits avec remise à zéro asynchrone. Ce registre est basique, on peut sans autres lui ajouter les fonctions reset_synthronique et enable, il faut bien entendu modifier la description VHDL ci-dessous pour lui ajouter ces fonctionnalités.

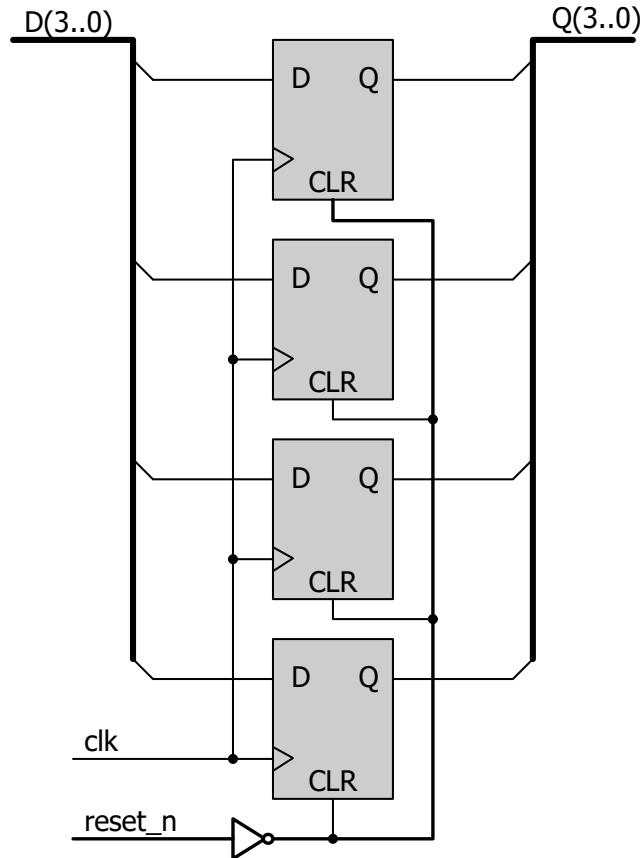


Figure 49 Structure interne d'un registre tampon de 4 bits

Description VHDL d'un registre tampon

La description ci-dessous correspond à un registre de 4 bits avec reset asynchrone actif bas, on n'a volontairement pas inclus les sorties q inverses, qui sont rarement utilisées.

Pour un registre tampon de n bits, la description est identique, il suffit de changer la taille des vecteurs d et q dans l'entité,

```
entity regtampon is
    Port ( clk           : in std_logic;
           reset_n       : in std_logic;
           D             : in std_logic_vector(3 DOWNTO 0);
           Q             : out std_logic_vector(3 DOWNTO 0));
end regtampon;

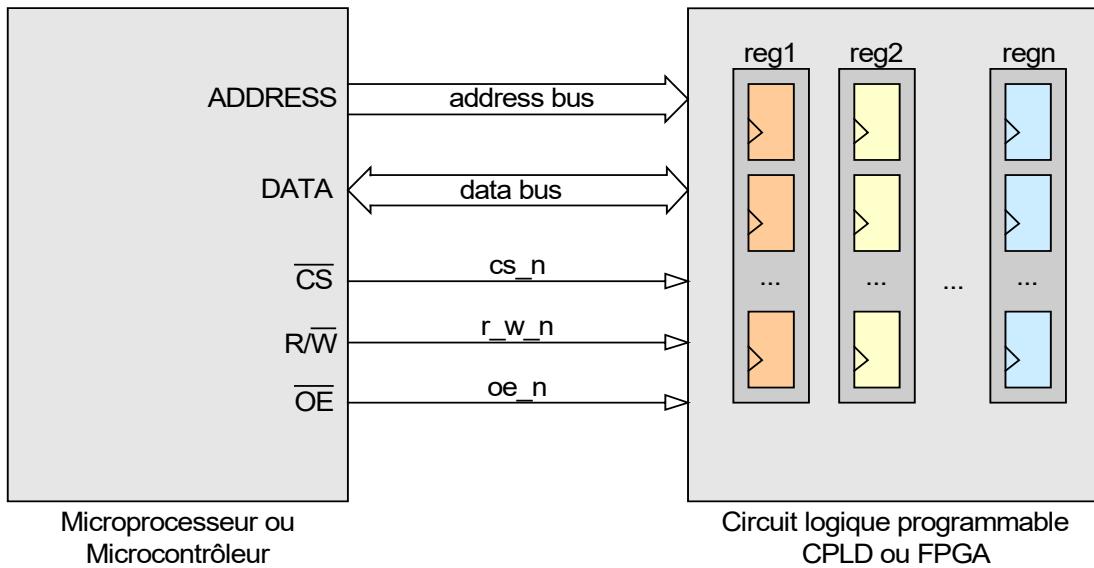
architecture behavioral of regtampon is

BEGIN

p1:PROCESS(clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        Q <= (OTHERS => '0');
    ELSIF (clk'EVENT AND clk = '1') THEN
        Q <= D;
    END IF;
END PROCESS;
end behavioral;
```

Registres mémoire

Les circuits logiques programmables sont souvent interfacés avec de microcontrôleurs ou des microprocesseurs qui doivent pouvoir accéder par leur bus de données externe à des registres en lecture et en écriture à l'intérieur des circuits logiques programmables. Ces registres se comportent comme une zone mémoire de quelques bytes externe au microcontrôleur. Le principe des registres mémoire est détaillé dans le schéma bloc ci-dessous :



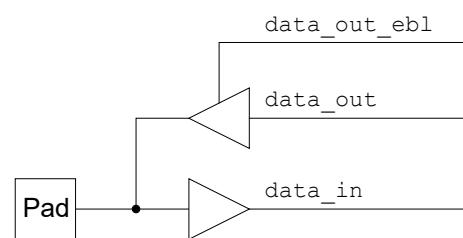
- Chaque registre mémoire (reg1, reg2,...regn) est un ensemble de maximum n bascules D, où n correspond au nombre de bit du bus data.
- Il est bien clair que les signaux CS, R/W et OE ne se trouvent pas sur tous les microcontrôleurs, on les retrouve également sous d'autres appellations, par exemple CE, WR, RD, et sont en général actifs bas.
- Dans certains microcontrôleurs, il n'y a pas de signaux CS (chip select), on doit par conséquent effectuer un décodage d'adresses externe, où mieux dans le circuit logique programmable (voir chapitre Décodage d'adresses).

Bus de données bidirectionnel

Dans un circuit logique programmable, on ne travaille qu'avec des signaux en mode in ou out. Lorsqu'on arrive dans le circuit avec un signal bidirectionnel (mode inout), la première chose à faire est de séparer le signal en deux signaux, un signal en entrée, et l'autre en sortie. C'est souvent le cas avec un bus de données bidirectionnel d'un microcontrôleur externe.

La première étape est donc de créer dans le design au niveau top de la hiérarchie deux bus à partir du bus de données, un bus en entrée, l'autre en sortie, afin de descendre dans la hiérarchie des signaux uniquement en mode in ou out. On peut décrire ce fonctionnement de la manière ci-dessous :

```
--BIDIRECTIONAL DATA
P1:PROCESS (data_out, data_out_ebl) BEGIN
    IF data_out_ebl = '0' THEN
        data <= (OTHERS => 'Z');
    ELSE
        data <= data_out;
    END IF;
END PROCESS;
data_in <= data;
```



La description VHDL ci-dessus générera la structure suivante pour chaque bit du signal data

Le buffer sur `data_out` est un buffer tri-state, ce qui signifie que si le signal `data_out_ebl` est à l'état logique bas, sa sortie sera en haute impédance (Z), par contre si `data_out_ebl` est à l'état logique haut, sa sortie sera `data_out`.

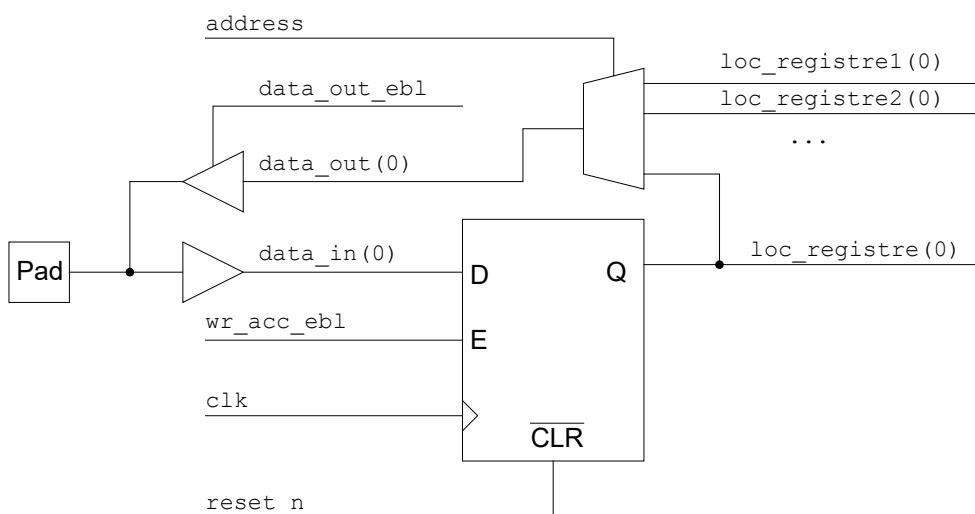
Structure d'un registre mémoire :

Le plus souvent, on a besoin de registres en lecture et en écriture, ces registre sont composés de n bascules D, correspondant chacune à 1 bit du registre. Pour un registre de 8 bit, on aura 8 bascules D, pour un registre de 16 bit, on aura 16 bascules D, et ainsi de suite.

Comme on pratique des systèmes logiques synchrones, la bascule est déclenchée par un flanc montant du signal d'horloge, ce qui pose un problème, en effet les registre doivent être mis à jour uniquement lors d'un cycle write (écriture) du microcontrôleur, c'est pourquoi on doit générer un signal interne combinatoire `wr_acc_ebl`, qui autorise la bascule à être modifiée seulement si ce signal est à l'état logique haut. En général ce signal est une combinaison des signaux chip select (`cs_n`) et write (`r_w_n`) du microcontrôleur.

Chaque registre correspond à une adresse (en général on utilise les bits d'adresse bas) du microcontrôleur, c'est pourquoi il faut déclarer une constante par registre qui, qui identifie à quelle adresse se situe le registre.

Vous pouvez voir la structure du bit(0) du registre appelé `loc_Registre` dans la figure ci-dessous :



Remarques importantes :

- La figure ci-dessus explique le principe pour un bit d'un registre, en l'occurrence le bit 0 du registre `loc_Registre`. Il est bien clair que cette structure est reproduite n fois pour chaque registre, où n correspond à la largeur du bus de données.
- On peut dans certains registres uniquement lire ou uniquement écrire. Les bascules D sont supprimées dans le cas où l'on doit uniquement lire, tandis que les multiplexeurs sont supprimés dans le cas où l'on doit uniquement écrire.

Description VHDL

La description VHDL est décomposée en deux parties bien distinctes, un premier process séquentiel permettant l'écriture des registres, et un second process combinatoire permettant la lecture des registres est en fait un simple multiplexeur. Un exemple de description VHDL permettant de générer deux registres de 8 bit en lecture/écriture est donné ci-dessous :

```

ARCHITECTURE Behavioral OF registres IS

CONSTANT c_registre1 : std_logic_vector(2 DOWNTO 0) := "010"; --Adresse 0x2
CONSTANT c_registre2 : std_logic_vector(2 DOWNTO 0) := "011"; --Adresse 0x3
SIGNAL loc_registre1 : std_logic_vector(7 DOWNTO 0);           --registre1
SIGNAL loc_registre2 : std_logic_vector(7 DOWNTO 0);           --registre2
SIGNAL wr_acc_ebl    : std_logic;
SIGNAL rd_acc_ebl    : std_logic;

BEGIN
p1:PROCESS(clk, reset_n) --WRITE REGISTERS
BEGIN
    IF reset_n = '0' THEN
        loc_registre1     <= (OTHERS => '0');
        loc_registre2     <= (OTHERS => '0');
    ELSIF (clk'EVENT AND clk = '1') THEN
        IF wr_acc_ebl = '1' THEN
            CASE address IS
                WHEN c_registre1 =>
                    loc_registre1 <= data_in;
                WHEN c_registre2 =>
                    loc_registre2 <= data_in;
                WHEN OTHERS => null;
            END CASE;
        END IF;
    END IF;
END PROCESS;

--READ REGISTERS (MULTIPLEXEUR COMBINATOIRE)
p2:PROCESS(rd_acc_ebl, address, loc_registre1, loc_registre2)
BEGIN
    IF rd_acc_ebl = '1' THEN
        data_out <= (OTHERS => '0');
        CASE address IS
            WHEN c_registre1 =>
                data_out<= loc_registre1;
            WHEN c_registre2 =>
                data_out<= loc_registre2;

            WHEN OTHERS =>
                data_out <= (OTHERS => '0');
        END CASE;
    ELSE
        data_out <= (OTHERS => '0');
    END IF;
END PROCESS;

--Signaux combinatoires
wr_acc_ebl <= '1' WHEN cs_n = '0' AND r_w_n = '0' ELSE '0';      --ECRITURE
rd_acc_ebl <= '1' WHEN cs_n = '0' AND oe_n = '0' ELSE '0';       --LECTURE
data_out_ebl <= rd_acc_ebl;           --Activation buffer tri-state

END Behavioral;
```

Remarques :

- Il va de soi que l'on peut ajouter autant de registres que 2^n fois le nombre de signaux dans le bus d'adresses, il suffit d'ajouter une instruction WHEN par registre.
- Dans certains cas des registres ne sont accédés qu'en lecture, il n'y a donc aucune assignation dans le premier process (p1). Dans ce cas il n'est pas non plus nécessaire de créer un signal local pour ce registre en lecture seul.
- Les signaux combinatoires wr_acc_ebl et data_out_ebl sont dépendants des signaux du processeur externe qui accède aux registres, ils doivent donc être adaptés selon les signaux du processeur utilisé.
- Le premier process est séquentiel, il génère les registres mémoire. Le deuxième process est combinatoire, il génère un multiplexeur piloté par les adresses basses utilisées pour identifier le registre.

Les registres à décalage

Un registre à décalage est un registre de n bits avec la possibilité de décaler les données mémorisées d'une position à chaque flanc actif du signal d'horloge. Ces registres sont utilisés pour réaliser des décalages de mots binaires, par exemple pour des opérations arithmétiques ou pour tester un bit d'un mot. On distingue les types de registres à décalage suivants :

- Entrée série – sortie série
- Entrée série – sortie parallèle
- Entrée parallèle – sortie série
- Entrée parallèle – sortie parallèle

Les sections suivantes donnent un aperçu des différents types de registres à décalage, sans pour autant être exhaustif. On peut réaliser de multiples combinaisons différentes pour réaliser des registres à décalage.

Principe de fonctionnement

Les exemples ci-dessous réalisent des registres avec décalage à droite. Dans le tableau ci-dessous on peut voir le principe du décalage à droite sur un registre de 4bit.

REGISTRE	bit(3)	bit(2)	bit(1)	bit(0)
t	U	V	W	X
t+1clk	din	U	V	W
t+2clk	din	din-1	U	V

Pour décrire un registre à décalage, on utilise l'opérateur de concaténation &.

Registre serial IN serial OUT

La Figure 50 nous montre la structure d'un registre à décalage de 4 bits avec entrée série et sortie série. L'entrée série serin contient le bit qui sera décalé d'un rang à chaque flanc actif du signal d'horloge (CLOCK). Ce bit apparaît à la sortie serout après 4 cycles de CLOCK.

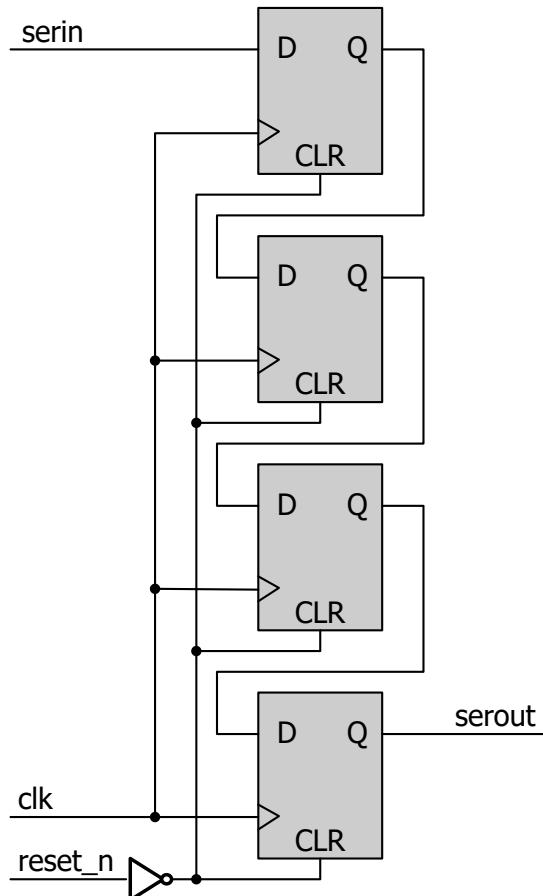


Figure 50 Structure d'un registre à décalage de 4 bits avec entrée série et sortie série.

Description VHDL

Ci-dessous, la description d'un registre à décalage de 4 bits avec entrée série et sortie série. Il est très facile d'adapter la taille du registre, il suffit de changer la taille du vecteur interne reg et de modifier son assignation.

```

entity serinserout is
    Port ( clk           : in std_logic;
           reset_n      : in std_logic;
           serin         : in std_logic;
           serout        : out std_logic);
end serinserout;

architecture Behavioral of serinserout is
begin
    reg: std_logic_vector(3 downto 0); --signal interne pour registre
    begin
    end

```

```

p1:process(clk, reset_n)
begin
    if reset_n = '0' then
        reg <= (OTHERS => '0');
    elsif clk'EVENT and clk ='1' then
        reg <= serin & reg(3 downto 1);      --decalage a droite
    end if;
end process;

serout <= reg(0); --assignation du bit 0 du register sur la sortie serie

end Behavioral;

```

Registre serial IN parallel OUT

La structure d'un registre à décalage avec entrée série et sortie parallèle est montré dans la Figure 51. Ce registre a une sortie par bit mémorisé, afin de les rendre disponibles pour d'autres circuits. Un tel registre peut être utilisé pour effectuer une conversion série/parallèle.

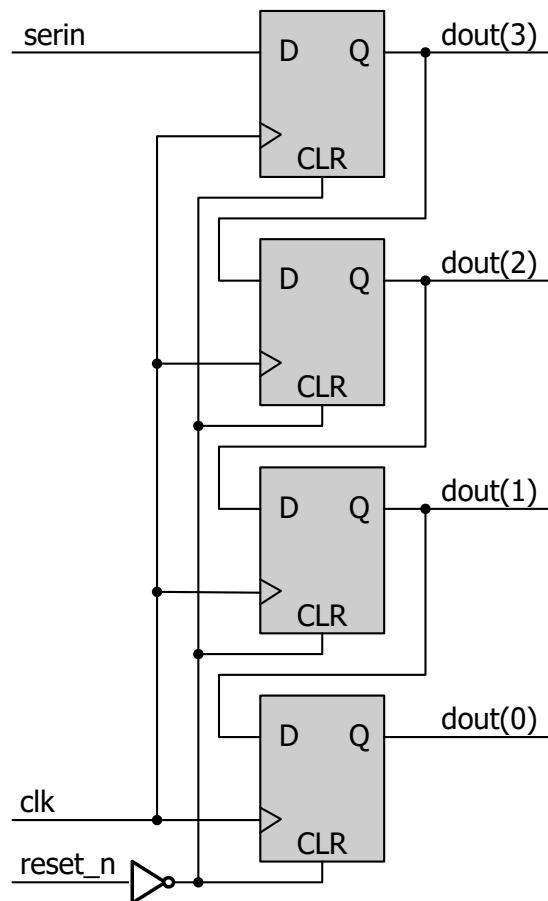


Figure 51 Structure d'un registre à décalage de n bits avec entrée série et sortie parallèle.

Description VHDL

Ci-dessous, la description d'un registre à décalage de 4 bits avec entrée série et sortie parallèle. Il est très facile d'adapter la taille du registre, il suffit de changer la taille du vecteur `dout` et celle du vecteur interne `reg` et de modifier son assignation.

```

entity serinparout is
    Port ( clk          : in  std_logic;
           reset_n      : in  std_logic;
           serin        : in  std_logic;
           dout         : out std_logic_vector(3 downto 0));
end serinparout;

architecture Behavioral of serinparout is

signal reg: std_logic_vector(3 downto 0);

begin

process(clk, reset_n)
begin
    if reset_n = '0' then
        reg <= (OTHERS => '0');
    elsif clk'EVENT and clk ='1' then
        reg <= serin & reg(3 downto 1);
    end if;
end process;

dout <= reg;

end Behavioral;

```

Registre parallel IN serial OUT

Inversement au registre précédent, il est possible de construire un registre avec entrée parallèle et sortie série. La Figure 52 nous montre la structure d'un tel registre, à chaque flanc actif du signal CLOCK, soit de nouvelles données sont chargées dans le registre (LOAD), soit le contenu du registre est décalé (SHIFT), ceci en fonction de l'état du signal LOAD/SHIFT. Un tel registre peut être utilisé pour réaliser une conversion parallèle/série.

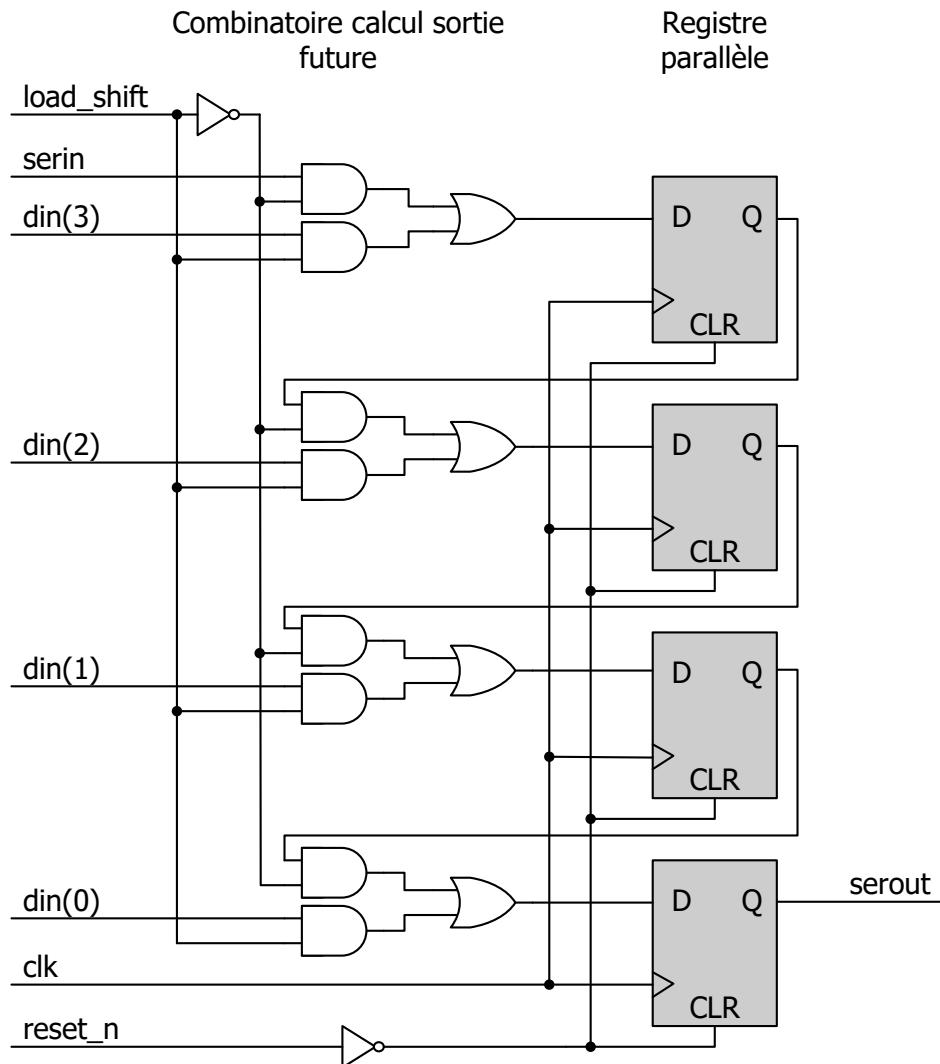


Figure 52 Structure d'un registre à décalage de n bits avec entrée parallèle et sortie série.

Description VHDL

Ci-dessous, la description d'un registre à décalage de 4 bits avec entrée parallèle et sortie série. Il est très facile d'adapter la taille du registre, il suffit de changer la taille du vecteur din et celle du vecteur interne reg et de modifier son assignation.

```

entity parinserout is
    Port ( clk          : in std_logic;
           reset_n     : in std_logic;
           load_shift   : in std_logic;
           serin        : in std_logic;
           din          : in std_logic_vector(3 downto 0);
           serout       : out std_logic);
end parinserout;

architecture Behavioral of parinserout is

signal reg: std_logic_vector(3 downto 0);

begin

process(clk, reset_n)
begin
    if reset_n = '0' then
        reg <= (OTHERS => '0');
    elsif clk'EVENT and clk ='1' then
        if load_shift = '1' THEN
            reg <= din;
        else
            reg <= serin & reg(3 downto 1);
        end if;
    end if;
end process;

serout <= reg(0);

end Behavioral;

```

Registre parallel IN parallel OUT

En rendant disponible toutes les sorties du registre précédent, on obtient un registre à décalage avec entrée parallèle et sortie parallèle, dont la structure est donnée dans la Figure 53. Un tel registre permet de réaliser toutes les fonctions réalisées par les trois types de registres à décalage étudiés précédemment.

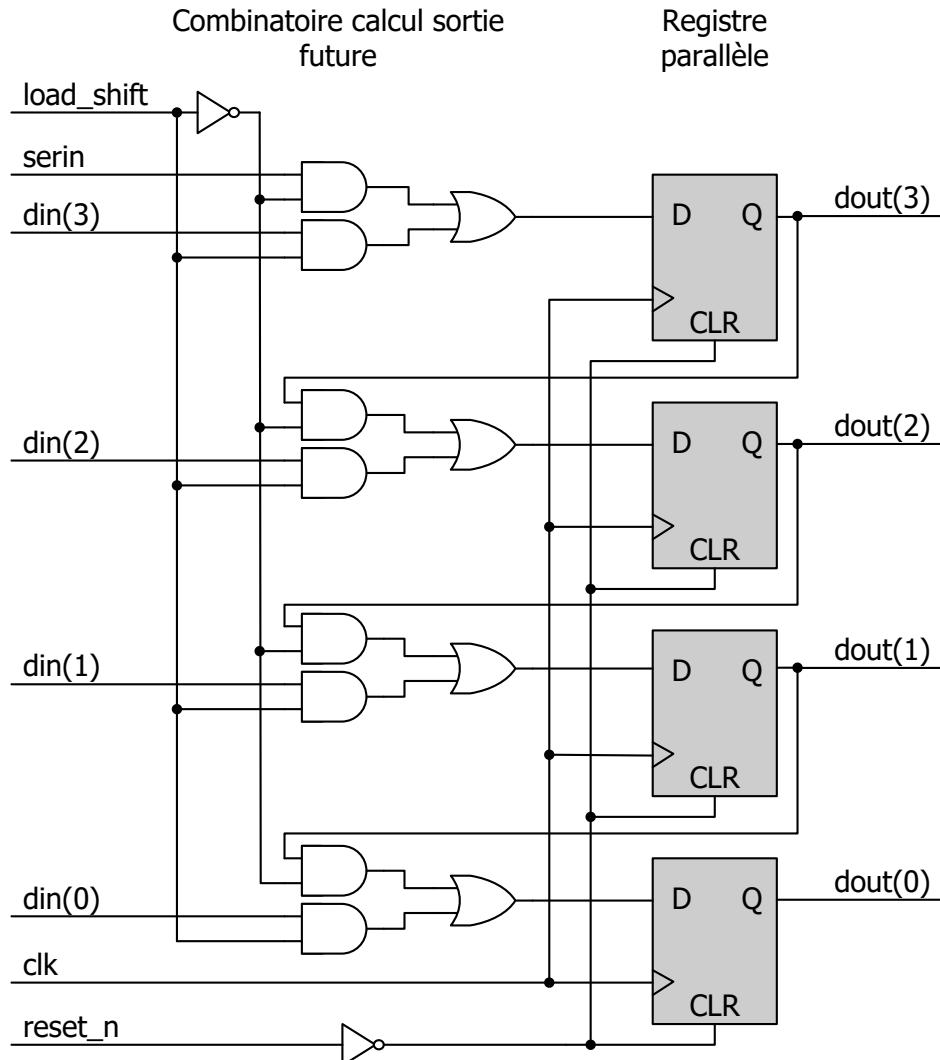


Figure 53 Structure d'un registre à décalage de n bits avec entrée parallèle et sortie parallèle.

Description VHDL

Ci-dessous, la description d'un registre à décalage de 4 bits avec entrée parallèle et sortie parallèle. Il est très facile d'adapter la taille du registre, il suffit de changer la taille des vecteurs din, dout ainsi que celle du vecteur interne reg et de modifier son assignation.

```

entity parinparout is
    Port ( clk          : in  std_logic;
           reset_n      : in  std_logic;
           load_shift   : in  std_logic;
           serin        : in  std_logic;
           din          : in  std_logic_vector(3 downto 0);
           dout         : out std_logic_vector(3 downto 0));
end parinparout;

architecture Behavioral of parinparout is

signal reg: std_logic_vector(3 downto 0);

begin

process(clk, reset_n)
begin
    if reset_n = '0' then
        reg <= (OTHERS => '0');
    elsif clk'EVENT and clk ='1' then
        if load_shift = '1' THEN
            reg <= din;
        else
            reg <= serin & reg(3 downto 1);
        end if;
    end if;
end process;

dout <= reg;

end Behavioral;

```

Les machines d'états synchrones

Les machines à nombre fini d'états jouent un rôle important dans la synthèse des fonctions logiques séquentielles. Une machine d'état peut se trouver à chaque instant dans une position parmi un nombre fini de positions possibles. Elle parcourt des cycles en changeant d'état uniquement lors des flancs (montant) du signal d'horloge. L'architecture générale d'une machine d'états (Finite State Machine, FSM en abrégé) est celui de la Figure 54.

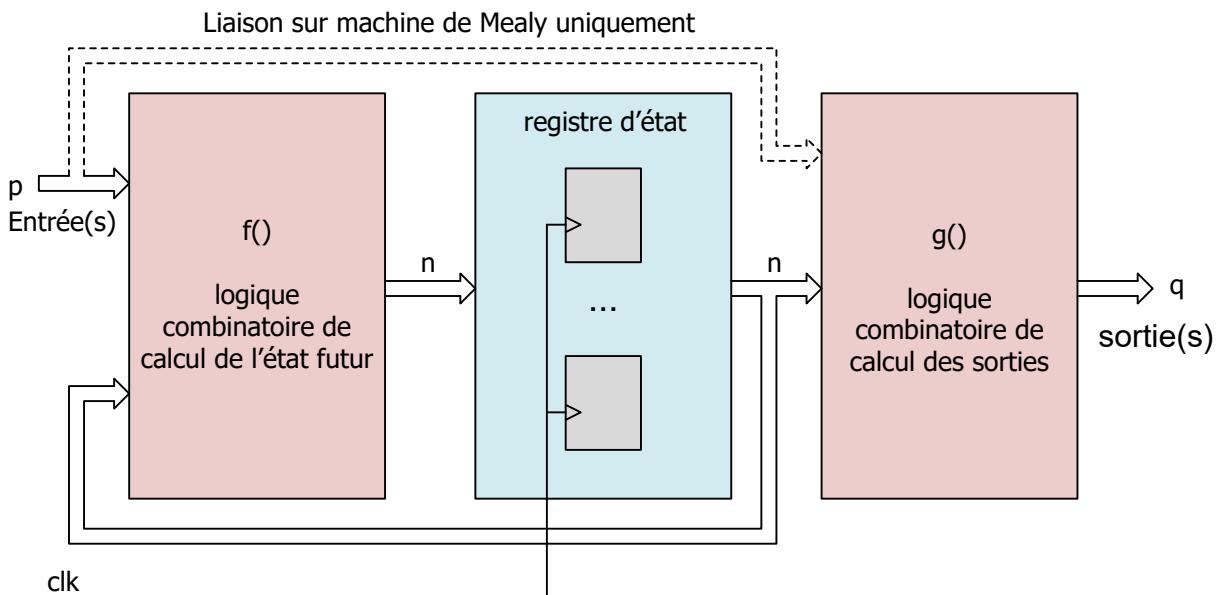


Figure 54 Machine à nombre fini d'états.

Le registre d'état est un registre synchrone de n éléments binaires (en général des bascules D). Son contenu représente l'état actuel de la machine. À chaque front d'horloge son contenu est remis à jour, l'état futur remplace l'état actuel. La taille du registre d'état fixe le nombre d'états accessibles, avec n bascules D dans le registre, on peut coder au maximum 2^n états.

Certains types de machines d'états nommées one-hot utilisent dans le registre d'état une bascule D par état, donc n états.

La fonction combinatoire $f()$ calcule l'état futur de la machine en fonction de l'état actuel et des entrées externes. Pour éviter tout risque d'aléa, ces entrées doivent être synchrones de l'horloge.

La fonction combinatoire $g()$ calcule les sorties en fonction de l'état actuel pour une machine de Moore. Dans le cas d'une architecture de Mealy, le calcul des sorties se fait en fonction de l'état actuel et des entrées et de l'état actuel. Sa complexité est liée au codage adopté pour les états. Dans certains cas il est possible de la supprimer, en choisissant un codage du registre d'état adapté aux sorties désirées.

On évitera d'utiliser une machine de Mealy si les entrées ne sont pas synchronisées avec le signal d'horloge, car dans ce cas les sorties de la machine de Mealy ne sont plus synchrones.

Machine d'état de Mealy à sorties synchronisées

Si la logique combinatoire de sortie est nécessaire, il est souvent souhaitable d'en synchroniser le résultat à l'aide d'un registre parallèle, on utilise donc une machine d'états de Mealy à sorties synchronisées illustrée dans la Figure 55.

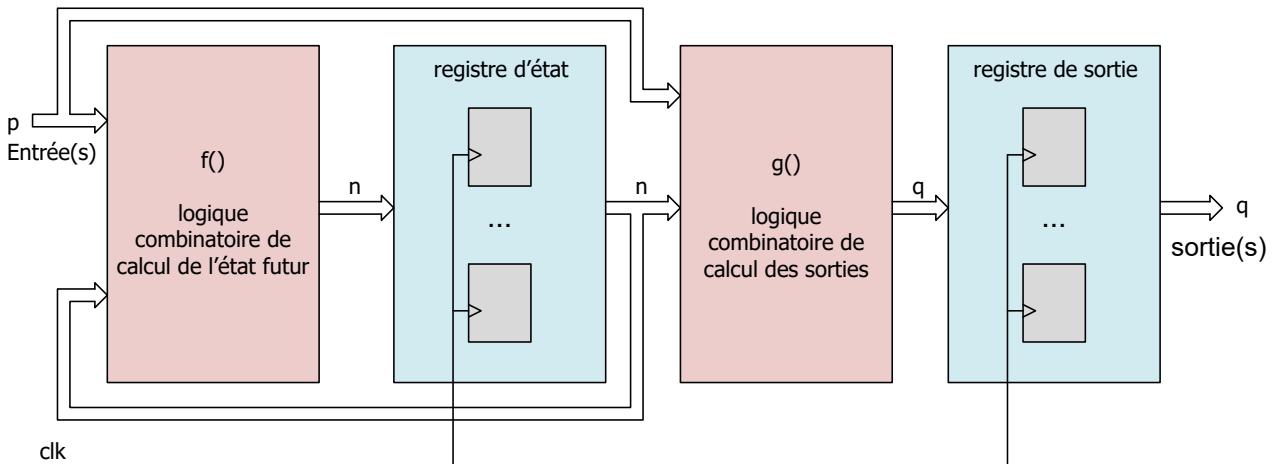


Figure 55 Machine d'état de Mealy à sorties synchronisées.

Pour réaliser une machine de Mealy à sorties synchronisées, il ne suffit pas d'ajouter un bloc registre de q bascules d sur la sortie d'une machine de Mealy « standard », ceci introduirait une latence (retard) d'une période du signal d'horloge (clk) sur les sorties.

Par conséquent, le même flanc du signal d'horloge doit nous faire changer l'état **et** la sortie. Nous verrons donc dans la description VHDL d'un machine à sorties synchronisées, que les états et les sorties sont gérés dans le même processus.

Structure interne d'une machine d'états

Avant l'apparition des synthétiseurs VHDL, la synthèse des machines d'états se faisait manuellement, toutes les équations excitation des registres d'états, ainsi que les fonctions de sorties étaient effectuées dans des tables de Karnaugh. Les synthétiseurs actuels calculent efficacement les équations des machines à états, à partir d'une description comportementale de la machine en VHDL. Il est cependant important de connaître de quoi est composée une machine à états. On remarque dans la Figure 56 qu'il n'y a pas d'autres éléments que des bascules D (registres d'états) et de la logique combinatoire pour le calcul des états futurs ainsi que des sorties.

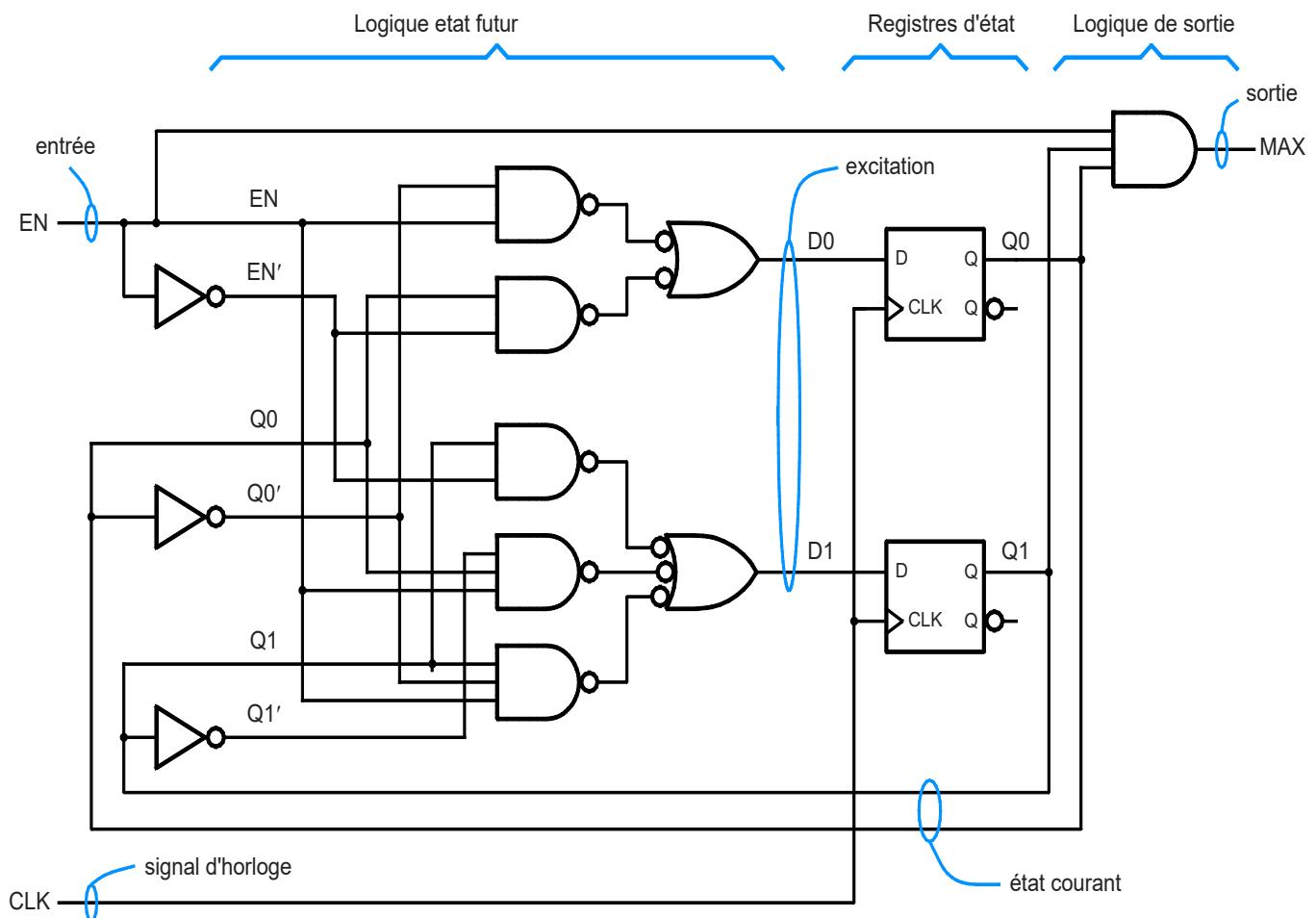


Figure 56 Schéma logique interne d'une machine à états synchrones

Le graphe des états (diagramme de transition)

Chaque état du système est représenté par une bulle portant un nom (évent. un numéro), l'évolution du système est représenté par des flèches représentant les transitions. Pour qu'une transition soit activée, il faut que les trois conditions suivantes soient vérifiées :

1. Le système se trouve dans l'état source considéré.
2. La condition de réalisation sur les entrées est vraie.
3. Un flanc actif du signal d'horloge survient.

Il existe 2 variantes possibles pour la représentation des machines d'états, suivant que la modélisation est effectuée pour une machine de Moore ou de Mealy. Dans le cas d'une machine de Moore (Figure 57), les sorties sont liées aux états: à chaque état correspond une configuration du vecteur de sortie. L'indication du vecteur de sortie est donc spécifiée dans la bulle de description de l'état interne. La flèche spécifie uniquement la condition attendue de changement d'état. Ce type de description correspond, par exemple, au fonctionnement des automates programmables.

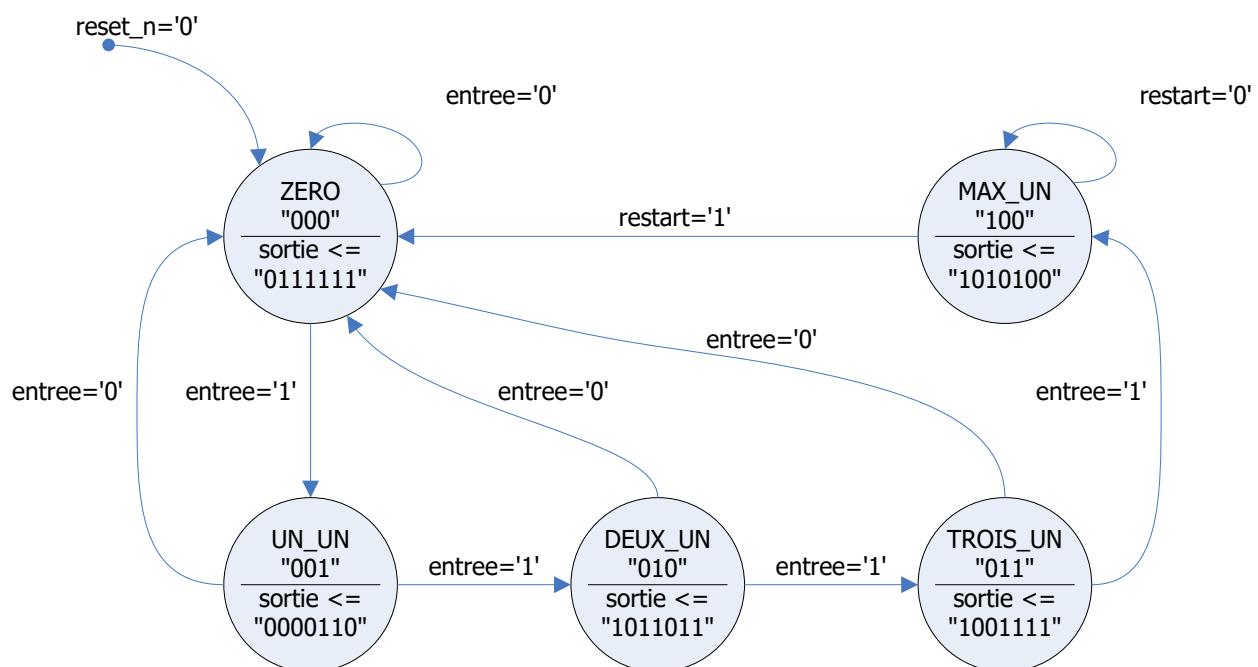


Figure 57 Graphe des états pour une machine de Moore.

Table de vérité des sorties en fonction de l'état

Dans certains cas, le nombre de sorties élevées d'une machine de Moore nous oblige à faire une table de vérité des sorties en fonction des états de la machine, ce qui allège considérablement le graphe des états, dans lequel ne figure plus que les conditions de transition entre les états. Un exemple est donné dans le Tableau 6.

Etats	Sorties		
	Erreur	lampe	init
DEBUT	0	0	1
MILIEUX	0	1	0
FORT	0	1	0
FAIBLE	0	1	0
ERREUR	1	0	0
TOTAL	0	0	0

Tableau 6 Exemple de table des sorties en fonction de l'état de la machine

Dans le cas des machines de Mealy, la sortie peut varier indépendamment de l'état en fonction des changements de configuration du vecteur d'entrée. L'état de la sortie ne peut plus figurer au sein de la bulle d'état et doit apparaître au niveau de la transition (flèche).

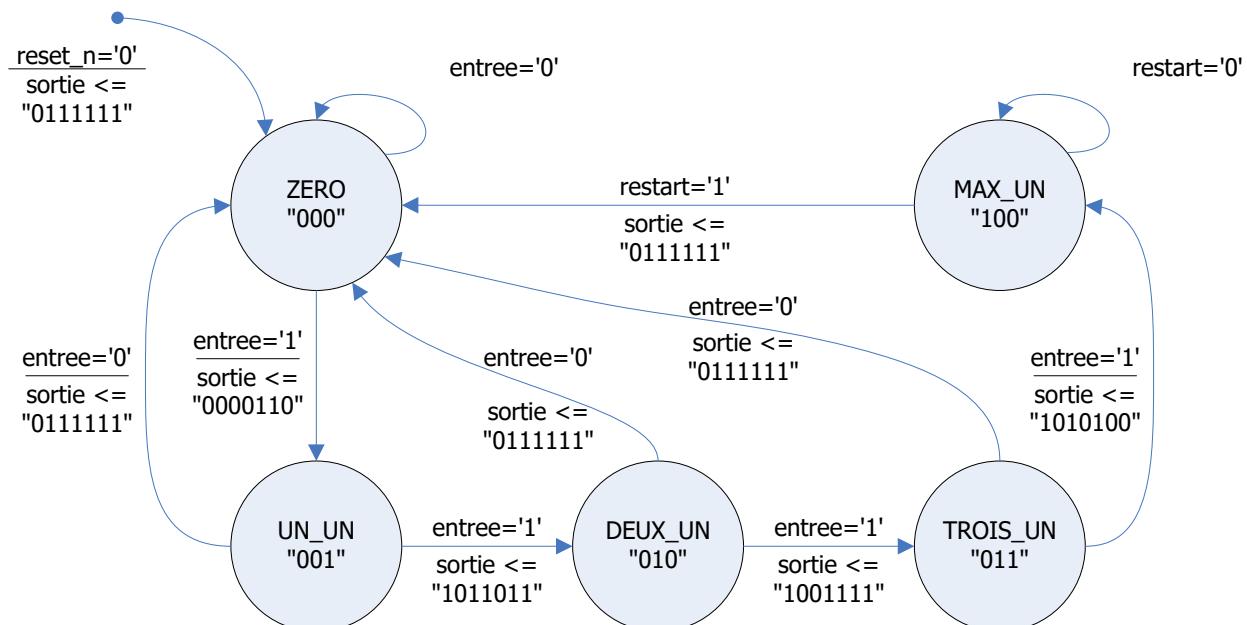


Figure 58 Graphe des états pour une machine de Mealy.

Du graphe des états à la description VHDL

Pour illustrer les trois types de machines à états, nous allons utiliser décrire les machines correspondant aux graphes des états vus précédemment.

Codage des états

Comme déjà dis précédemment, on a besoin n bascules pour coder 2^n états. Ces états sont définis dans un paquetage, ou dans la zone déclarative de l'architecture. Beaucoup de personnes confondent états avec sorties, c'est vrai que dans certains cas où un choix réfléchi du codage des états peut simplifier ou même supprimer le calcul des sorties. Dans une machine qui comporte des combinaisons inutilisées, les états inutilisés doivent être raccordés dans le cycle normal de fonctionnement, c'est le sens de l'état générique nommé `OTHERS`.

Machines de Moore

Dans une machine de Moore, les sorties sont calculées à partir de la valeur contenue dans le registre d'état. La description VHDL ci-dessous propose une solution qui correspond à cette architecture. L'état du système est matérialisé par le signal interne `state`, qui correspond au signal à la sortie du registre d'état:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Moore_SM is
    Port ( clk : in STD_LOGIC;
            reset_n : in STD_LOGIC;
            entree : in STD_LOGIC;
            restart : in STD_LOGIC;
            sortie : out STD_LOGIC_VECTOR (6 DOWNTO 0));
end Moore_SM;

architecture Behavioral of Moore_SM is

--Declare type, subtype
subtype t_state is std_logic_vector(2 DOWNTO 0);

--Declare constantes
constant c_ZERO      : t_state    := "000";
constant c_UN_UN     : t_state    := "001";
constant c_DEUX_UN   : t_state    := "010";
constant c_TROIS_UN  : t_state    := "011";
constant c_MAX_UN    : t_state    := "100";

--Declare signaux
SIGNAL state : t_state;

BEGIN

P1:PROCESS(clk, reset_n) BEGIN
    IF reset_n = '0' THEN
        state <= c_ZERO;
    ELSIF (clk'EVENT AND clk = '1') THEN
        CASE state IS
            WHEN c_ZERO =>
                IF entree = '1' THEN
                    state <= c_UN_UN;
                ELSE
                    state <= c_ZERO;
                END IF;
            WHEN c_UN_UN =>

```

```

        IF entree = '1' THEN
            state <= c_DEUX_UN;
        ELSE
            state <= c_ZERO;
        END IF;
    WHEN c_DEUX_UN =>
        IF entree = '1' THEN
            state <= c_TROIS_UN;
        ELSE
            state <= c_ZERO;
        END IF;
    WHEN c_TROIS_UN =>
        IF entree = '1' THEN
            state <= c_MAX_UN;
        ELSE
            state <= c_ZERO;
        END IF;
    WHEN c_MAX_UN =>
        IF restart = '1' THEN
            state <= c_ZERO;
        ELSE
            state <= c_MAX_UN;
        END IF;
    WHEN OTHERS =>
        state <= c_ZERO;
    END CASE;
END IF;
END PROCESS;

--Attribution des sorties combinatoire fonction des etats uniquement
sortie <= "0111111" WHEN state = c_ZERO      ELSE
"0000110" WHEN state = c_UN_UN      ELSE
"1011011" WHEN state = c_DEUX_UN     ELSE
"1001111" WHEN state = c_TROIS_UN    ELSE
"1010100" WHEN state = c_MAX_UN      ELSE
"-----";
end Behavioral;

```

Machines de Mealy

Dans une machine de Mealy, les entrées du système ont une action directe sur les sorties. Cette action directe permet de créer un mécanisme d'anticipation : une sortie de Mealy change de valeur avant que l'état de la machine n'entérine, éventuellement, la modification par une transition. Le changement de valeur d'une sortie du type Moore indique, lui, qu'un changement d'état de la machine vient de se produire. La seule différence réside donc dans la description est l'attribution combinatoire des sorties, le reste de la description est totalement identique. Donc dans notre exemple seul l'attribution de la sortie change, le reste de la description est identique :

```

--Attribution des sorties combinatoire fonction des etats et des entrées
sortie <= "0000110" WHEN state = c_ZERO AND entree = '1' ELSE
"1011011" WHEN state = c_UN_UN AND entree = '1' ELSE
"1001111" WHEN state = c_DEUX_UN AND entree = '1' ELSE
"1010100" WHEN state = c_TROIS_UN AND entree = '1' ELSE
"0111111" WHEN state = c_MAX_UN AND restart = '1' ELSE
"0111111";

```

Machines de Mealy à sorties (re)synchronisées

Dans ce type de machines les sorties sont synchronisées avec le signal d'horloge (avec des bascule D), ce qui signifie que dans la description, elles sont assignées à l'intérieur du processus, en même temps que les états. Ce qui signifie qu'il faut anticiper le changement des sorties, pour qu'il ait lieu au même flanc du signal d'horloge que le changement d'état, il n'est pas question de décaler le changement des sorties d'un cycle de clock. L'assignation des sorties dans le processus se fait uniquement lorsqu'il y a changement de celles-ci.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mealy_Sync_SM is
    Port ( clk : in STD_LOGIC;
            reset_n : in STD_LOGIC;
            entree : in STD_LOGIC;
            restart : in STD_LOGIC;
            sortie : out STD_LOGIC_VECTOR (6 DOWNTO 0));
end Mealy_Sync_SM;

architecture Behavioral of Mealy_Sync_SM is

--Declare type, subtype
subtype t_state is std_logic_vector(2 DOWNTO 0);

--Declare constantes
constant c_ZERO      : t_state := "000";
constant c_UN_UN     : t_state := "001";
constant c_DEUX_UN   : t_state := "010";
constant c_TROIS_UN  : t_state := "011";
constant c_MAX_UN    : t_state := "100";

--Declare signaux
SIGNAL state : t_state;

BEGIN

P1:PROCESS(clk, reset_n) BEGIN
    IF reset_n = '0' THEN
        state <= c_ZERO;
        sortie <= "0111111";
    ELSIF (clk'EVENT AND clk = '1') THEN
        CASE state IS
            WHEN c_ZERO =>
                IF entree = '1' THEN
                    state <= c_UN_UN;
                    sortie <= "0000110";
                ELSE
                    state <= c_ZERO;
                END IF;
            WHEN c_UN_UN =>
                IF entree = '1' THEN
                    state <= c_DEUX_UN;
                    sortie <= "1011011";
                ELSE
                    state <= c_ZERO;
                    sortie <= "0111111";
                END IF;
            WHEN c_DEUX_UN =>
                IF entree = '1' THEN
                    state <= c_TROIS_UN;
                    sortie <= "1001111";
                END IF;
        END CASE;
    END IF;
END;

```

```

        ELSE
            state <= c_ZERO;
            sortie <= "0111111";
        END IF;
    WHEN c_TROIS_UN =>
        IF entree = '1' THEN
            state <= c_MAX_UN;
            sortie <= "1010100";
        ELSE
            state <= c_ZERO;
            sortie <= "0111111";
        END IF;
    WHEN c_MAX_UN =>
        IF restart = '1' THEN
            state <= c_ZERO;
            sortie <= "0111111";
        ELSE
            state <= c_MAX_UN;
        END IF;
    WHEN OTHERS =>
        state <= c_ZERO;
        sortie <= "0111111";
    END CASE;
END IF;
END PROCESS;

end Behavioral;

```

Les Compteurs

Les compteurs représentent l'une des fonctions logique les plus fréquemment utilisées. Il en existe deux types principaux, à savoir les compteurs asynchrones, qui tendent à disparaître et qu'il ne faut surtout plus utiliser, et les compteurs synchrones. Nous allons étudier la synthèse manuelle des compteurs synchrones, puis nous verrons comment on peut les décrire en VHDL.

Modulo

Le compteur 4 bits représenté par le chronogramme ci-dessous possède 16 états distincts (0000 à 1111), on dit alors que c'est un compteur à MODULO 16. Rappelons que le modulo est toujours égal au nombre d'états occupés par le compteur pendant un cycle complet avant son recyclage à l'état initial. Le modulo est porté à une valeur plus élevée simplement en ajoutant des bascules au compteur. Un compteur de n bits aura un modulo de 2^n .

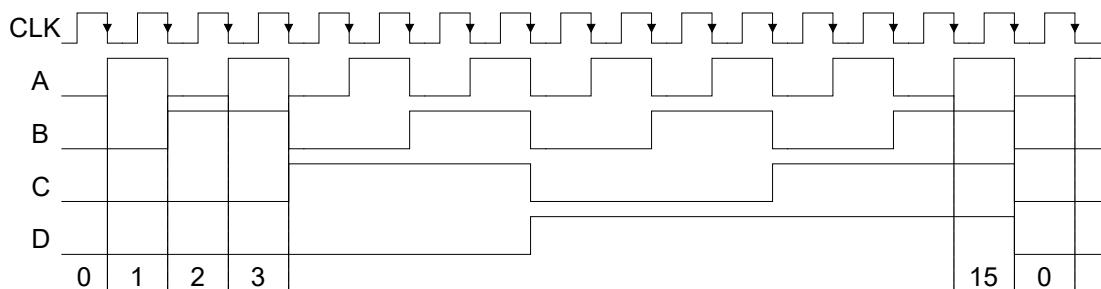


Figure 59 Chronogramme d'un compteur 4 bit (modulo 16).

Synthèse manuelle des compteurs synchrones

Dans les compteurs parallèles ou synchrones toutes les bascules sont déclenchées simultanément (en parallèle) par les impulsions d'horloge d'entrée. Étant donné que les impulsions d'horloge sont appliquées à toutes les bascules, il doit y avoir un certain mécanisme qui indique quand une impulsion d'horloge doit faire commuter une bascule ou la laisser dans le même état. On réalise un tel mécanisme en agissant avec un système combinatoire les entrées D des bascules. Pour illustrer ces propos, nous allons réaliser la synthèse manuelle d'un compteur modulo 6 :

On doit tout d'abord déterminer à l'aide d'une table de vérité, les entrées D que l'on doit avoir en fonction des sorties Q et Q^+ :

Etat	Q1	Q2	Q3	$Q1^+$	$Q2^+$	$Q3^+$	D1	D2	D3
0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	0	1	0
2	0	1	0	0	1	1	0	1	1
3	0	1	1	1	0	0	1	0	0
4	1	0	0	1	0	1	1	0	1
5	1	0	1	0	0	0	0	0	0
6	1	1	0	φ	φ	φ	φ	φ	φ
7	1	1	1	φ	φ	φ	φ	φ	φ

On remarque que les entrées Di sont égales aux sorties Qi^+ , ce qui est logique, vu que la sortie d'une bascule (Q) prends la valeur de l'entrée (D) après un flanc du signal d'horloge.

Table de Karnaugh pour l'entrée D1 :

		Q1	Q2	Q3	00	01	11	10
		0		0	0	φ	1	
		1		0	1	φ	0	

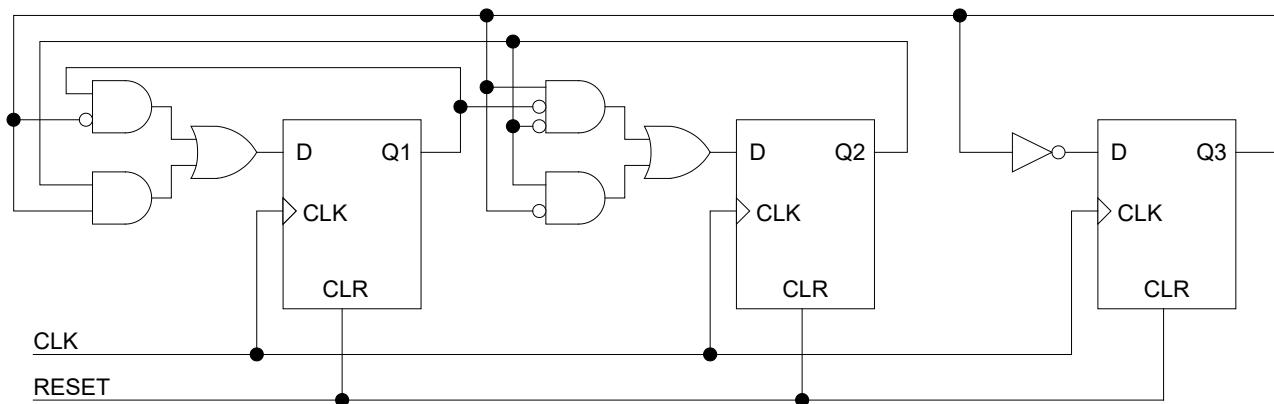
$$D1 = Q1\overline{Q3} + Q2Q3$$

Avec la même méthode, on trouve :

$$D2 = Q2\overline{Q3} + \overline{Q1}Q2Q3$$

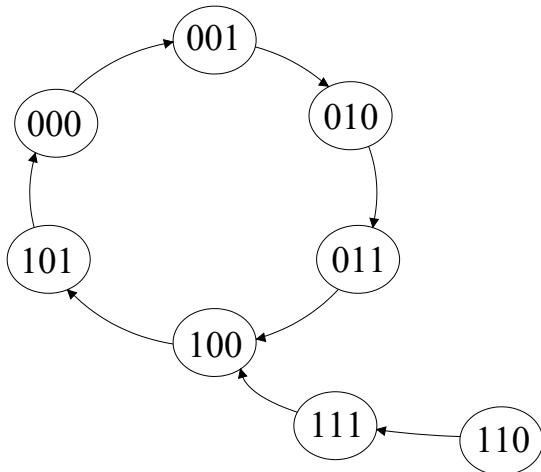
$$D3 = \overline{Q3}$$

Schéma logique :



On remarque que le compteur est synchrone, car le même signal d'horloge va sur toutes les bascules.

Graphe des états :



On doit toujours analyser les états hors du cycle du compteur, car au démarrage, le compteur peut partir sur une de ces valeurs, et entrer dans un cycle parasite, ce qui n'est pas le cas dans notre exemple. Pour analyser ces états, il existe deux solutions :

1. Remplacer dans la table de vérité, les états ϕ par les valeurs qu'on leur a attribuées dans les tables de Karnaugh, ce qui nous donnera une table de vérité complète.
2. Analyser chaque état manquant avec les équations des sorties D.

Arithmétique en VHDL

Il existe historiquement plusieurs package permettant de réaliser de l'arithmétique binaire en VHDL. :

STD_LOGIC_ARITH, std_logic_signed, std_logic_unsigned et NUMERIC_STD (IEEE)

Pour des raisons de compatibilité à avec tous les synthétiseurs du marché, seul le paquetage normalisé NUMERIC_STD sera utilisé.

NUMERIC_STD permet d'utiliser 2 types de données : SIGNED, UNSIGNED

Pour l'instant seul le type UNSIGNED sera utilisé dans la création des compteurs.

Compteurs pré réglable

Les compteurs peuvent être pré réglables; on veut dire par-là qu'il est possible de charger dans le compteur, en tout temps, un nombre de départ de façon synchrone à l'aide d'un signal de commande et d'une valeur binaire parallèle à charger dans le compteur. L'action de pré réglage est aussi appelée changement (load) du compteur.

Description d'un compteur synchrone en VHDL

Pour décrire un compteur en VHDL, on a besoin d'utiliser le paquetage `NUMERIC_STD`. Ce paquetage inclut la surcharge des opérateurs + et - pour une addition ou une soustraction avec deux opérandes de type `UNSIGNED`.

Le résultat de l'opération sera de type `UNSIGNED`, il faut donc convertir le résultat en `std_logic_vector` à l'aide la fonction `std_logic_vector`.

Voici un exemple d'un compteur 8 bits (modulo 256), pré réglable de façon synchrone (`load`), avec reset asynchrone et enable :

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter8 is
    Port ( clk           : in std_logic;
            reset_n      : in std_logic;
            sw            : in std_logic_vector(7 DOWNTO 0);
            enable        : in std_logic;
            load          : in std_logic;
            led           : out std_logic_vector(7 DOWNTO 0)
    );
end counter8;

architecture behavioral of counter8 is

SIGNAL counter : std_logic_vector(7 DOWNTO 0); --Signal interne pour
compteur

BEGIN

PROCESS (clk, reset_n)
BEGIN
    IF reset_n = '0' THEN          --remise a zero asynchrone
        counter <= (OTHERS => '0');
    ELSIF clk'EVENT AND clk = '1' THEN
        IF load = '1' THEN        --préréglage synchrone du compteur
            counter <= sw;
        ELSIF enable = '1' THEN --autorisation synchrone de compter
            counter <= STD_LOGIC_VECTOR(UNSIGNED(counter) + 1);
        END IF;
    END IF;
END PROCESS;

--Attribution du compteur sur les leds en sortie pour visualisation
led <= counter;

end behavioral;

```

Diviseur de fréquence

Les diviseurs de fréquence sont très utilisés dans les systèmes numériques, en effet il arrive fréquemment que l'on doive générer des signaux selon une base de temps connue.

Le signal divisé (DIV) ne doit en aucun cas être utilisé comme signal d'horloge plus loin dans le design, mais comme un signal enable.

Il y a plusieurs façons de générer un diviseur de fréquence, mais la plus simple consiste à utiliser un compteur qui décrémente, et de détecter l'état zéro de ce compteur, puis de le recharger à la valeur de division -1. On peut voir l'exemple d'un diviseur par 4 dans la Figure 60 ci-dessous.

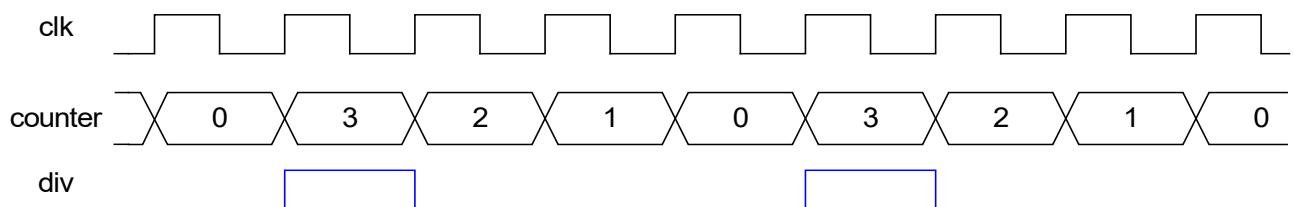


Figure 60 Chronogramme d'un diviseur par 4

Description VHDL

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

ENTITY divfreq IS
    PORT ( clk           : IN std_logic;
           Reset_n       : IN std_logic;
           div          : OUT std_logic);
END divfreq;

ARCHITECTURE behavioral OF divfreq IS

SIGNAL counter : std_logic_vector(1 downto 0);

BEGIN

PROCESS(clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
        counter <= "11";
        div      <= '0';
    ELSIF clk'EVENT AND clk = '1' THEN
        IF UNSIGNED(counter) = 0 THEN
            counter <= "11";
            div      <= '1';
        ELSE
            counter <= STD_LOGIC_VECTOR(UNSIGNED(counter) - 1);
            div      <= '0';
        END IF;
    END IF;
END PROCESS;

END behavioral;

```

Clocking

Nous allons étudier dans ce chapitre les contraintes liées à la gestion du signal d'horloge (clock) dans les circuits numériques et plus particulièrement les circuits logiques programmables. Dans la plupart des cas, on aimerait faire fonctionner les circuits numériques à la fréquence d'horloge la plus rapide possible et pour ce faire il faut connaître quelques notions très importantes relatives aux signaux d'horloge (clk). Ces notions sont développées ci-dessous.

Système numérique synchrone

Comme étudié précédemment, nous travaillons toujours de façon synchrone dans un circuit logique programmable, ce qui signifie que toutes les bascules D utilisées dans le circuit doivent changer d'état au même moment, à savoir au flanc montant ou descendant du signal d'horloge (clk).

Contraintes de timing sur les bascules D synchrones

Si l'entrée D et le clk changent en même temps sur une bascule D synchrone, son fonctionnement n'est pas garanti, on peut avoir en sortie de cette bascule une métastabilité de courte durée avant que la bascule se mette dans l'état haut ou bas. Cette métastabilité est totalement non désirée car elle peut générer des dysfonctionnements souvent aléatoires du système numérique implémenté.

Setup and Hold Times

Pour éviter ces phénomènes de métastabilité on doit respecter deux contraintes de temps dans l'utilisation de ces bascules, ces contraintes de temps sont très faibles, de l'ordre de quelques centaines de picosecondes et sont propres à chaque technologie. Elles sont donc définies dans les datasheets des circuits logiques utilisés.

Setup Time

C'est la durée pendant laquelle l'entrée synchrone (D) doit être stable avant le front actif de l'horloge (clk)

Hold Time

C'est la durée pendant laquelle l'entrée synchrone (D) doit être stable après le front actif de l'horloge (clk)

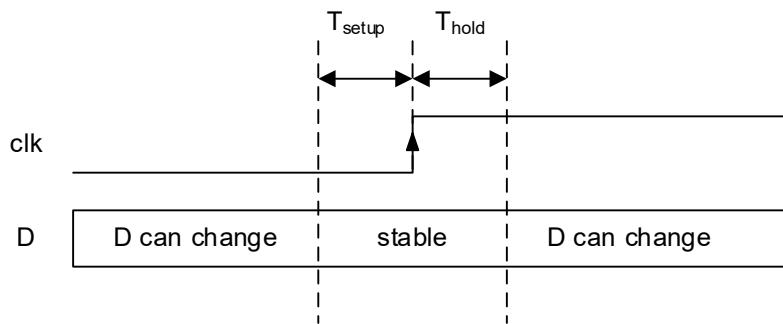


Figure 61 Diagramme représentant le Setup et le Hold Time

Skew

Le clock Skew est un phénomène dans les systèmes de circuits numériques synchrones dans lequel le même signal d'horloge (clk) arrive sur des composants (bascule D) à des moments différents. La différence instantanée entre les flancs de deux horloges quelconques est appelée le Skew.

Types de clock Skew

Skew positif

Cela se produit lorsque le registre de réception reçoit l'impulsion d'horloge après le registre source.

Skew négatif

Cela se produit lorsque le registre de réception reçoit l'impulsion d'horloge avant le registre source.

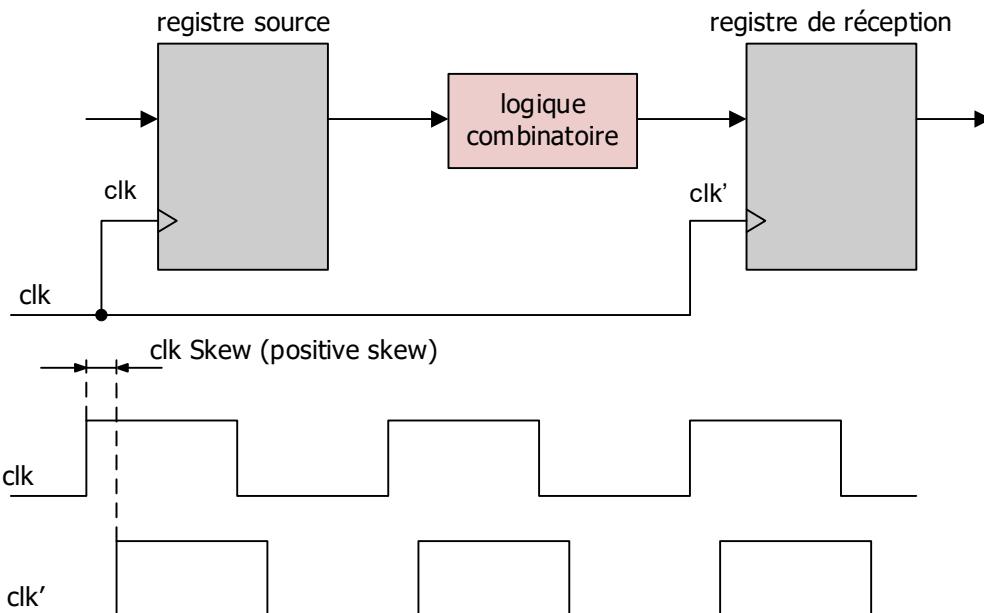


Figure 62 Représentation du Skew positif sur un signal d'horloge

Facteurs principaux causant le Skew

- Longueur de l'interconnexion
- Variations de température
- Couplage capacitif
- Imperfections des matériaux
- Différences de capacité d'entrée sur les entrées d'horloge

Jitter

Le Jitter (la gigue) est la variation à court terme d'un signal par rapport à sa position idéale dans le temps. La gigue est la variation de la période d'horloge de flanc à flanc. Elle peut varier de +/- la valeur du Jitter. D'un cycle à l'autre, la période et le rapport cyclique peuvent varier légèrement en raison du circuit de génération de l'horloge.

Le Jitter peut être modélisé en ajoutant des zones d'incertitude autour des fronts montants et descendants de la forme d'onde de l'horloge.

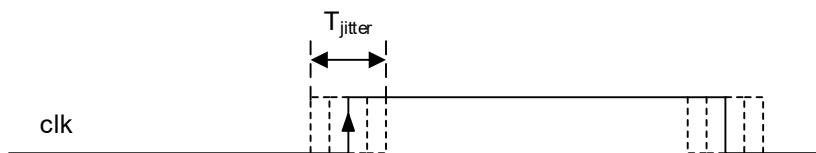


Figure 63 Représentation du Jitter sur un signal d'horloge

Sources du Jitter

Les sources les plus courantes du Jitter sont les suivantes :

- Les circuits internes de la PLL (phase-locked loop) qui sont utilisés pour la génération des horloges
- Le bruit thermique aléatoire d'un quartz.
- D'autres dispositifs résonnantes
- Bruit mécanique aléatoire provenant de la vibration du quartz

Fréquence maximum du signal d'horloge

Nous allons étudier dans ce chapitre ce qui limite la fréquence du signal d'horloge pour un système numérique donné implémenté un circuit logique programmable.

C'est une somme de plusieurs facteurs qui influence la fréquence maximum d'un système numérique donné :

1. Délais du routage interne (capacité des connexions et longueur des connexions)
2. Clock skew et jitter
3. Setup et hold time des FF
4. Délais de la logique combinatoire

Comme nous l'avons vu précédemment, on ne peut pas énormément influencer les 3 premiers points qui dépendent beaucoup de la technologie employée dans les circuits logique programmable.

Délais routage interne

Le placement / routage est effectué par le logiciel d'implémentation (Implement dans Vivado) et le résultat ne peut pas être énormément influencé si on ne modifie pas des paramètres dans le logiciel ou alors que l'on ajoute des contraintes sur certains signaux, ce qui est une tâche relativement complexe à effectuer. Ces délais de routages sont relativement importants dans la vitesse de fonctionnement et ils représentent à eux seul déjà près de 50% des délais totaux, c'est pourquoi on espère que les outils de routages fonctionnent bien, ce qui est souvent le cas. Dans des designs pointus on est obligé de générer des contraintes sur certains signaux comme expliqué ci-dessus.

Clock skew et jitter

Ils dépendent du système d'horloge interne du circuit logique et sont généralement très bien conçus par les fabricants. On doit juste s'assurer que la source d'horloge fournie au circuit logique (FPGA) soit de bonne qualité.

Setup et hold time des FF (bascules D)

Ces temps sont fixés par la technologie employées par les fabricants de circuits logiques. On ne peut pas influencer ces paramètres, on peut juste choisir des composants ayant des temps qui sont très faibles, mais là également les fabricants sont à la pointe de la technologie en ce qui concerne la fabrication de ces bascules internes aux FPGAs.

Délais de la logique combinatoire

Par contre, la conception du système numérique a une grande influence dans la fréquence de fonctionnement d'un système numérique. Il faut impérativement réduire les temps de propagation de la logique combinatoire au minimum nécessaire, ou alors « Pipeliner » le système. « Pipeliner » signifie introduire des registres dans les parties logiques combinatoires trop lentes pour diminuer les délais de ces dernières.

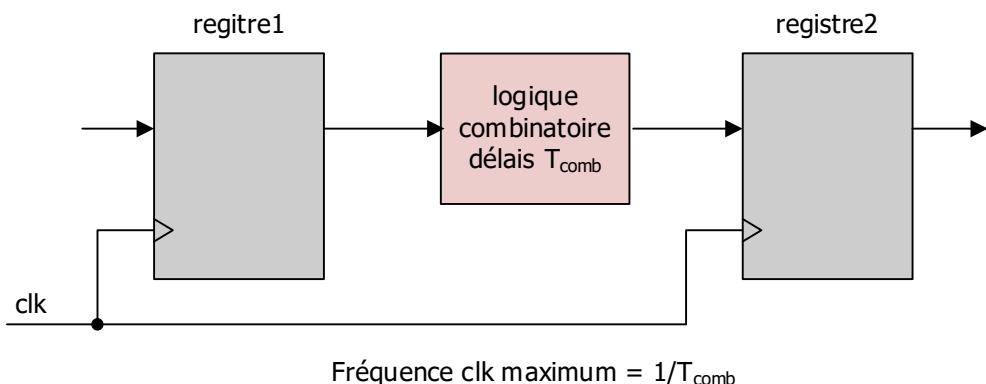


Figure 64 Calcul de la fréquence maximum signal d'horloge en fonction des délais de la logique combinatoire

Pour le calcul ci-dessus, on détermine la fréquence maximum du signal d'horloge en admettant un temps T_{setup} et $T_{\text{hold}} = 0$ ainsi qu'aucun Jitter et Skew sur le clock. Ceci ne correspond pas 100% à un cas réel, mais comme expliqué précédemment, les facteurs les plus importants qui déterminent la fréquence de fonctionnement maximum sont les délais de routage et le temps de propagation de la logique combinatoire.

Analyse de la fréquence maximum d'un système numérique donné

Tous les logiciels d'implémentation (Ex : Vivado) permettent de calculer la fréquence de fonctionnement maximum du signal d'horloge pour un design donné. Le logiciel prend en compte tous les paramètres maximum (Worst Case) qu'il a pu déduire en fonction de l'implémentation qu'il a réalisé. Pour obtenir cette information (Timing Report), il suffit de spécifier la fréquence du signal d'horloge dans le fichier de contrainte puis de regarder s'il n'y pas d'erreur de Timing à la fin de l'implémentation. Cela signifie que le circuit peut fonctionner à la fréquence spécifiée dans le fichier de contraintes.

Mémoires

Une mémoire est un ensemble d'éléments mémoires binaires. Ces éléments binaires (ou cases mémoires) sont de type non volatile (l'information écrite à l'intérieur de chacune des cases mémoire n'est pas affectée lors de l'ouverture du circuit d'alimentation), c'est le cas des ROM, PROM, EPROM et EEPROM ou volatile c'est le cas des RAM.

Accès séquentiel ou aléatoire

La mémoire vive (RAM) est généralement opposée à la mémoire morte (ROM) : Il est possible de lire et écrire de la mémoire vive alors qu'il est uniquement possible de lire de la mémoire morte. En revanche, la mémoire morte conserve les données lorsque l'alimentation électrique est coupée. La mémoire morte n'est donc pas volatile.

ACCÈS ALÉATOIRE

Contrairement à ce que son nom pourrait laisser croire, un accès aléatoire est un accès direct, celui qui ne nécessite pas de faire défiler une liste de données avant d'atteindre celle que l'on cherche.

ACCÈS SÉQUENTIEL

C'est le contraire de l'accès aléatoire. Accéder à des données de manière séquentielle, c'est être obligé de faire défiler une partie des données pour trouver celle que l'on cherche. On a un accès séquentiel aux données enregistrées sur une bande magnétique mais un accès aléatoire aux données stockées en mémoire vive.

ROM (Read-Only Memory)

Une **mémoire morte** (i.e. ROM) est une mémoire dont le contenu a été défini et réalisé une bonne fois pour toutes au moment de la fabrication. La fabrication de ROMs ne se conçoit que pour des séries importantes (> 10.000 unités). Cette programmation est faite directement sur le wafer (galette de silicium) à l'aide des masques de programmation.

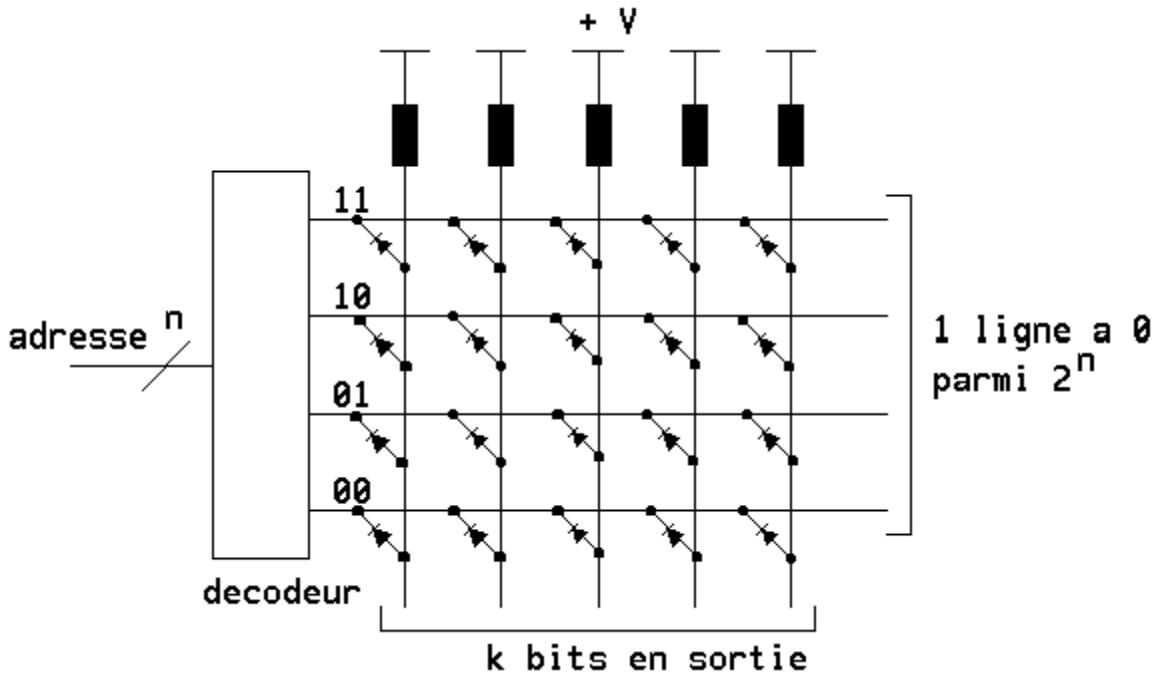
PROM (Programmable ROM)

Les PROMs sont des mémoires non volatiles, dont le contenu comme dans le cas des ROMs, est défini une fois pour toutes. Toutefois, contrairement aux ROMs, elles sont programmables (1 seule fois) par l'utilisateur.

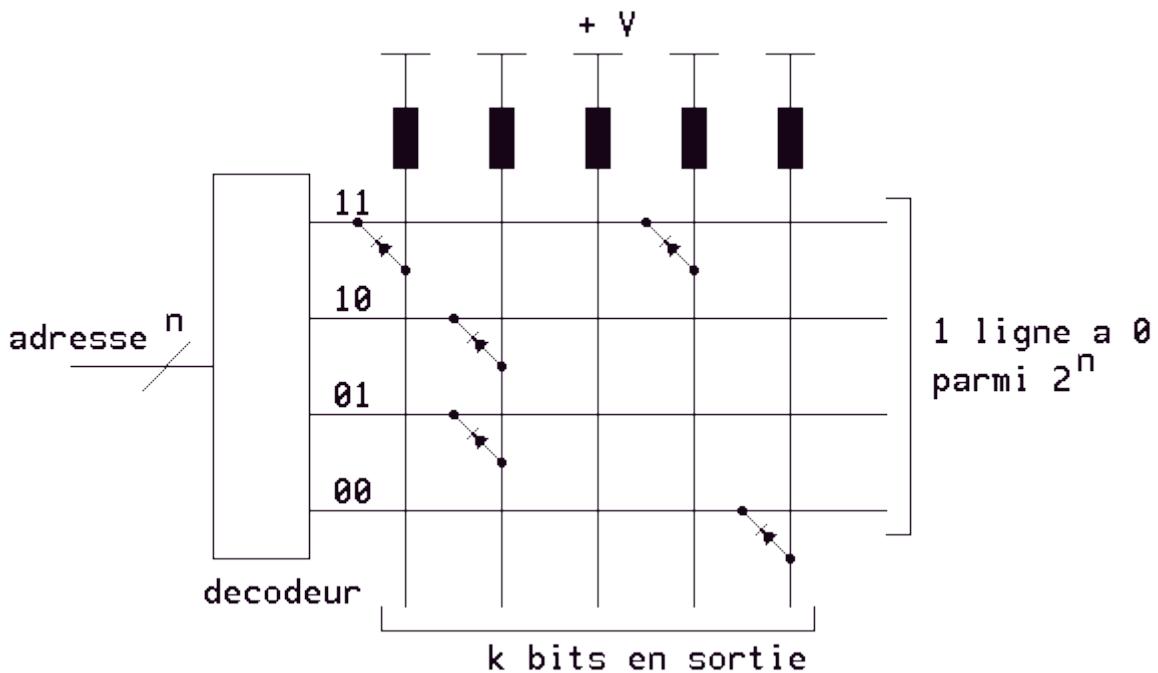
Il existe différents types de PROMs. Les nœuds de la matrice peuvent comprendre soit des fusibles soit des jonctions ayant une faible tension de claquage (anti-fusibles). Dans les deux cas, la programmation s'effectue en sélectionnant l'adresse désirée, en présentant la donnée sur les lignes de sortie, et en alimentant pendant un bref instant (quelques centaines de ms) le circuit avec une tension élevée (10 à 15 V suivant les cas), ce qui a pour effet de faire fondre le fusible (ouverture du circuit) ou de claqueter la jonction (fermeture du circuit). Ce processus est évidemment irréversible.

Principe

L'exemple le plus simple de mémoire morte à fusibles 4x5 bits peut se schématiser de la façon suivante : un décodeur 2 vers 4 (74139) avec sorties actives à l'état bas permet de sélectionner une ligne parmi 4. Exemple : à l'adresse A1 A0 = 00, la ligne notée 00 est forcée à 0 et les autres lignes sont à 1. Dans ce cas les 5 bits de sortie ont la valeur 0. Avant programmation toutes les sorties sont donc à 0.



Après programmation, c'est à dire après destruction de certains fusibles, on peut obtenir le schéma suivant :



Fonctionnement

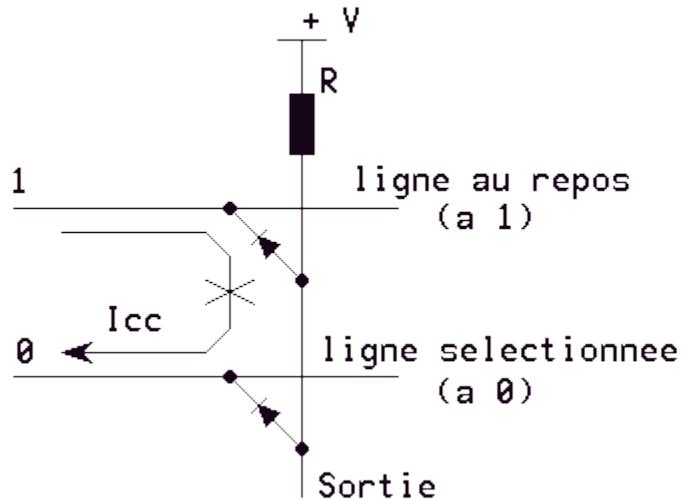
Grâce à la résistance de tirage une ligne de sortie vaut 1 en l'absence de diode (liaison détruite) entre elle et le fil d'adresse sélectionnée.

Si par contre, une diode est présente, elle ramène le potentiel de la ligne sortie à 0. Le contenu de cette mémoire 20 bits est alors :

adresse	S ₄ S ₃ S ₂ S ₁ S ₀
1 1	0 1 1 0 1
1 0	1 0 1 1 1
0 1	1 0 1 1 1
0 0	1 1 1 1 0

Remarque

Les liaisons entre les lignes de sélection d'adresse (lignes horizontales de la matrice), et les lignes de données (lignes verticales de la matrice) ne peuvent être de simples connexions, mais doivent être réalisées à l'aide de diodes. Les diodes servent à éviter un retour de courant depuis la ligne sélectionnée vers une autre qui ne l'est pas (court-circuit) :



Réalisation pratique

Les PROM à fusibles qui ont été les premières mémoires non volatiles commercialisées étaient en technologie bipolaire (c'était la seule maîtrisée à l'époque). Cette technologie est toujours utilisée et malgré leur forte consommation, les PROM bipolaires à fusibles sont encore utilisées pour les circuits rapides. En effet les PROM bipolaires permettent des temps d'accès de l'ordre de 20 ns.

EPROM (Erasable PROM)

Les EPROMs sont des mémoires non volatiles programmables électriquement puis effaçables par UV. Dans ce cas, les noeuds de la matrice sont constitués de transistors MOS à grille isolée. Cette grille peut être chargée par influence en appliquant une tension importante (10 à 15 V) entre le drain et le substrat. Une fois chargée, elle peut conserver sa charge de manière quasiment indéfinie, ce qui rend le transistor passant.

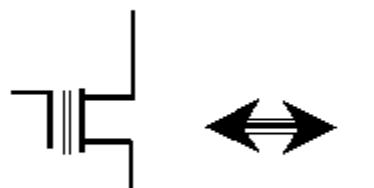
Le processus est réversible en irradiant la puce aux rayons ultraviolets pendant plusieurs minutes, ce qui décharge la grille par effet photoélectrique.

Principe

Chaque élément mémoire est composé d'un transistor MOS dont la grille est complètement isolée dans une couche d'oxyde. Par application d'une tension suffisamment élevée, qui est appelée tension de programmation, on crée des électrons chauds ou électrons ayant une énergie suffisamment suffisante pour traverser la mince couche d'oxyde. Ces charges s'accumulent et se retrouvent piégées dans la grille, la couche d'oxyde entre la grille et le silicium étant trop épaisse pour que les électrons puissent la traverser. La cellule mémoire est alors programmée.

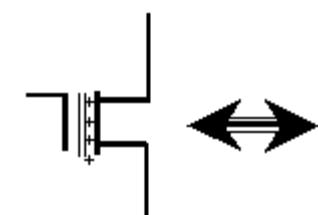
Si maintenant, on veut effacer la mémoire, on expose la puce aux rayonnements U.V. Les photons (ou particules d'énergie lumineuse) communiquent alors leur énergie aux électrons et leur font franchir la barrière isolante dans l'autre sens. La cellule mémoire est effacée.

Après effacement (grâce aux U.V.)



Circuit ouvert

Après programmation :



Circuit fermée

Exemple :

La mémoire proposée ci-dessous est une mémoire EPROM 16 bits. Cette mémoire est appelée mémoire NMOS car les éléments mémoires sont des transistors NMOS. Le problème de ces mémoires très peu utilisés est la consommation d'énergie même au repos (à cause des résistances de tirage). On leur préférera les mémoires CMOS qui ne consomment que lorsqu'elles sont sollicitées (on remplace alors les résistances de tirage par des transistors PMOS).

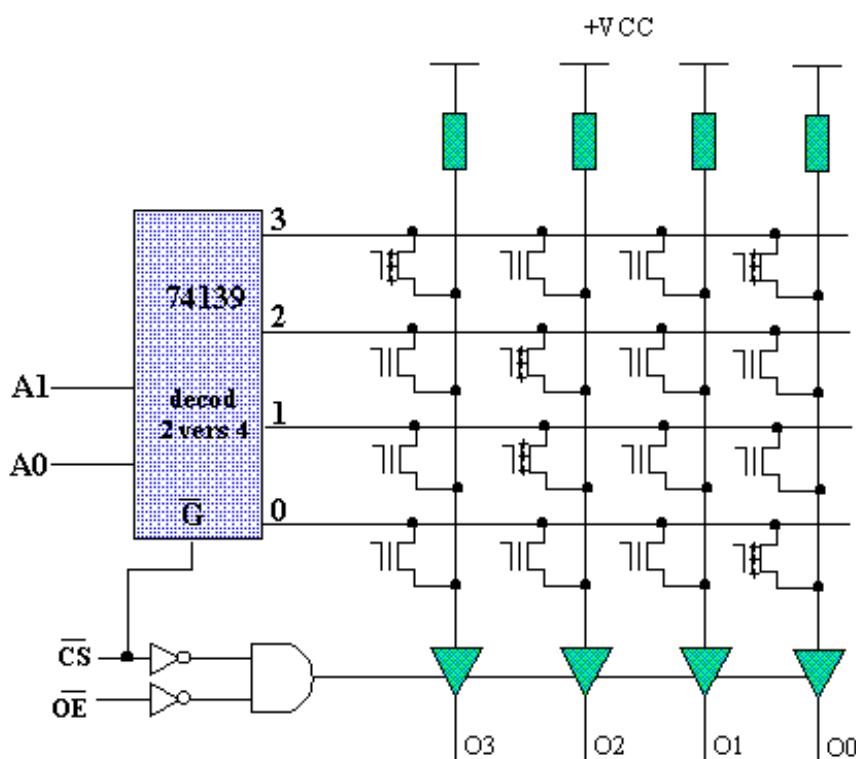
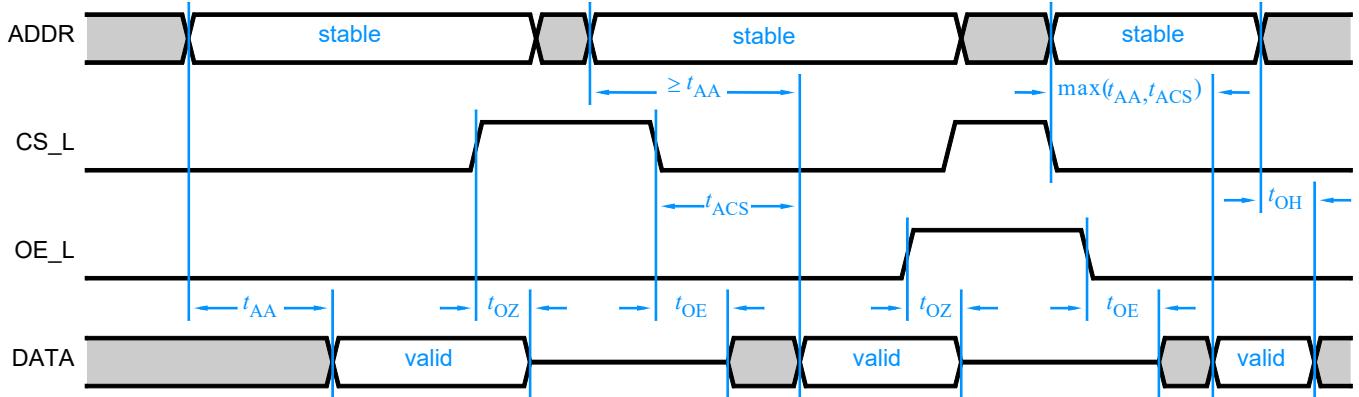


Figure 65 Mémoire EPROM 4x4 bits après programmation

On utilise un décodeur 2 vers 4 pour sélectionner une ligne parmi quatre. Si /CS est à 0 alors le décodeur est actif et si par exemple A1 A0 = 0 0 alors la ligne 0 est forcée à 0 (les autres lignes restent à 1) on obtient donc sur le bus de données O3 O2 O1 O0 = 1 1 1 0. En effet seul le transistor de la ligne 0 bit O0 est programmé (grille flottante chargée).

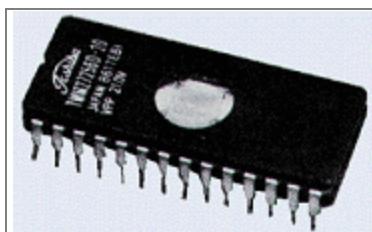
On utilise un buffer 3 états pour déconnecter la mémoire du bus de données externe. Pour envoyer les données sur le bus de données il faudra donc activer ce buffer 3 états (conditions /CS = 0 et /OE =0).

Timing d'une EPROM



Le temps d'accès en lecture tacs correspond au temps qu'il faut pour fournir la donnée après activation du boîtier ($CS_L = 0$) et des sorties ($OE_L = 0$). Il faut de plus que l'adresse soit stable pendant tout ce temps. Ce temps d'accès correspond aux différents temps de propagation des portes du circuit.

EPROM à UV ou OTP



Le boîtier de cette mémoire doit être muni d'une fenêtre translucide pour laisser passer les U.V. lors de la phase d'effacement, ce qui augmente fortement le coût de ces mémoires. Pour en réduire le prix, il existe les mémoires OTP (One Time Programmable) qui sont des mémoires EPROM sans fenêtre d'effacement.

Le cycle d'écriture se fait avec un programmeur d'EPROM. L'écriture dans la mémoire ne peut se faire qu'après avoir effacé celle-ci dans une lampe à UV (temps d'exposition entre 5 et 10 minutes). Lors de la phase d'écriture (quelques secondes) la broche /PGM est forcée à 0 et les tensions $V_{pp}=12,7V$ et $V_{cc}=6,25V$ sont appliquées.

RAM (Random Access Memory)

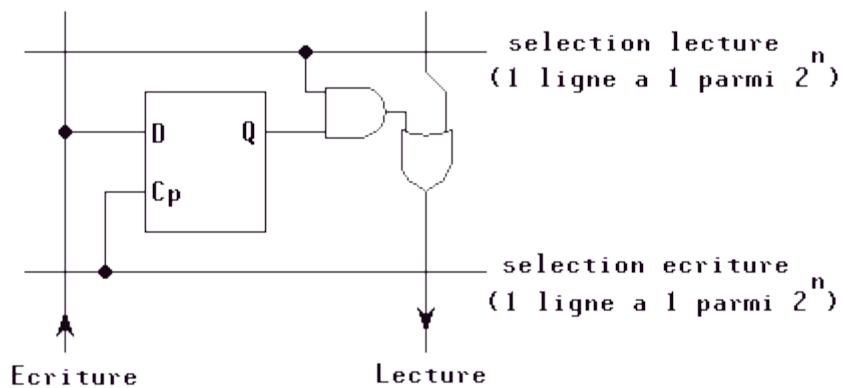
Contrairement aux ROMs, une RAM est une mémoire volatile, c'est-à-dire qui ne garde son contenu que tant qu'elle est sous tension.

Le nom de RAM (Random Access Memory), mémoire à accès aléatoire, est historiquement apparu pour signifier que contrairement aux bandes magnétiques qui étaient alors les seuls supports d'information existants, il n'est pas nécessaire de lire toutes les données situées devant celle que l'on cherche avant de pouvoir accéder à l'information désirée.

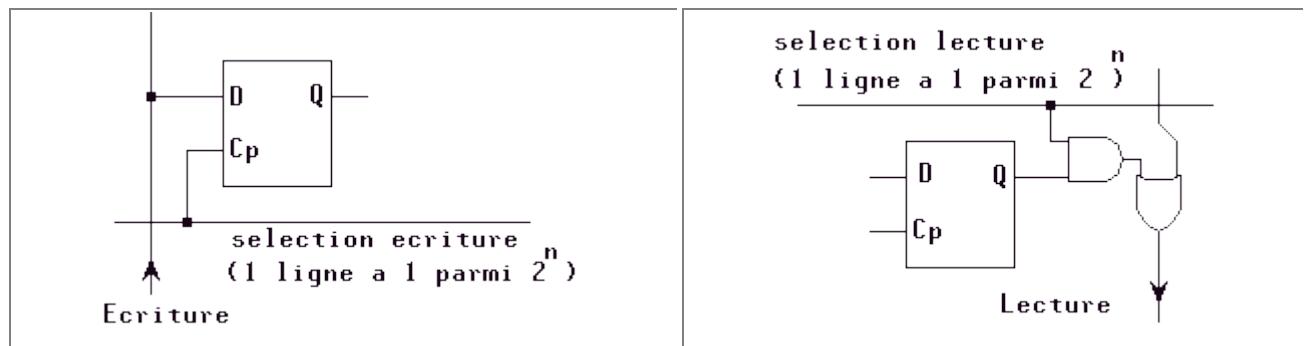
Mémoire SRAM

Principe

La cellule de base d'une SRAM est typiquement constituée par une bascule :



Cette cellule prend place dans deux matrices qui se superposent, l'une destinée à l'écriture, l'autre à la lecture :

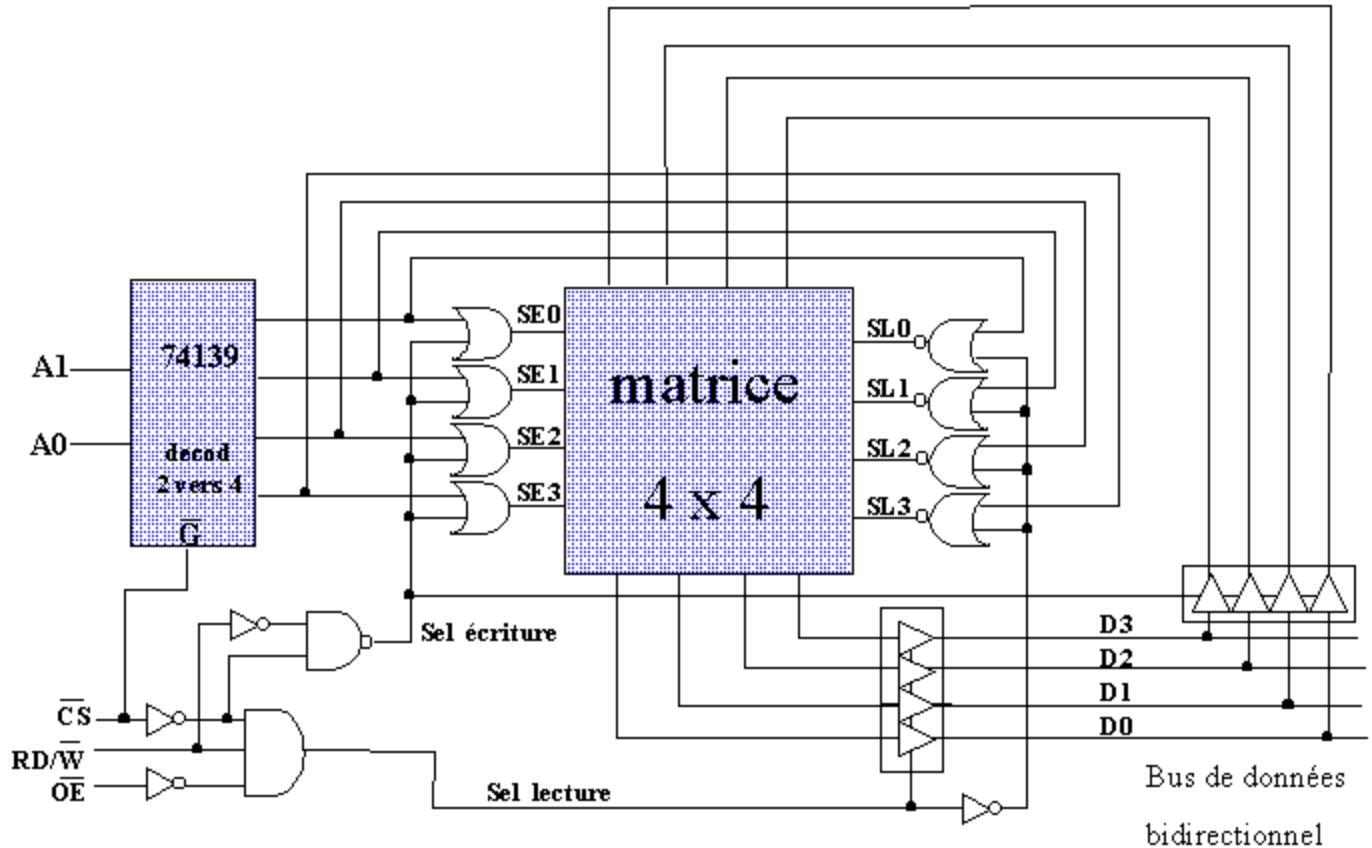


L'écriture consistera à mémoriser la donnée présente sur D (provenant du bus de données) lors du front montant du CLK (noté ici Cp). Cette information ne pourra être changée que lors d'un prochain front montant du CLK (nouveau cycle d'écriture).

La lecture consiste à forcer la ligne de sélection à 1 pour obtenir l'information présente sur l'élément mémoire sélectionnée sur le bus de données.

Exemple de réalisation d'une SRAM avec des circuits logiques du commerce

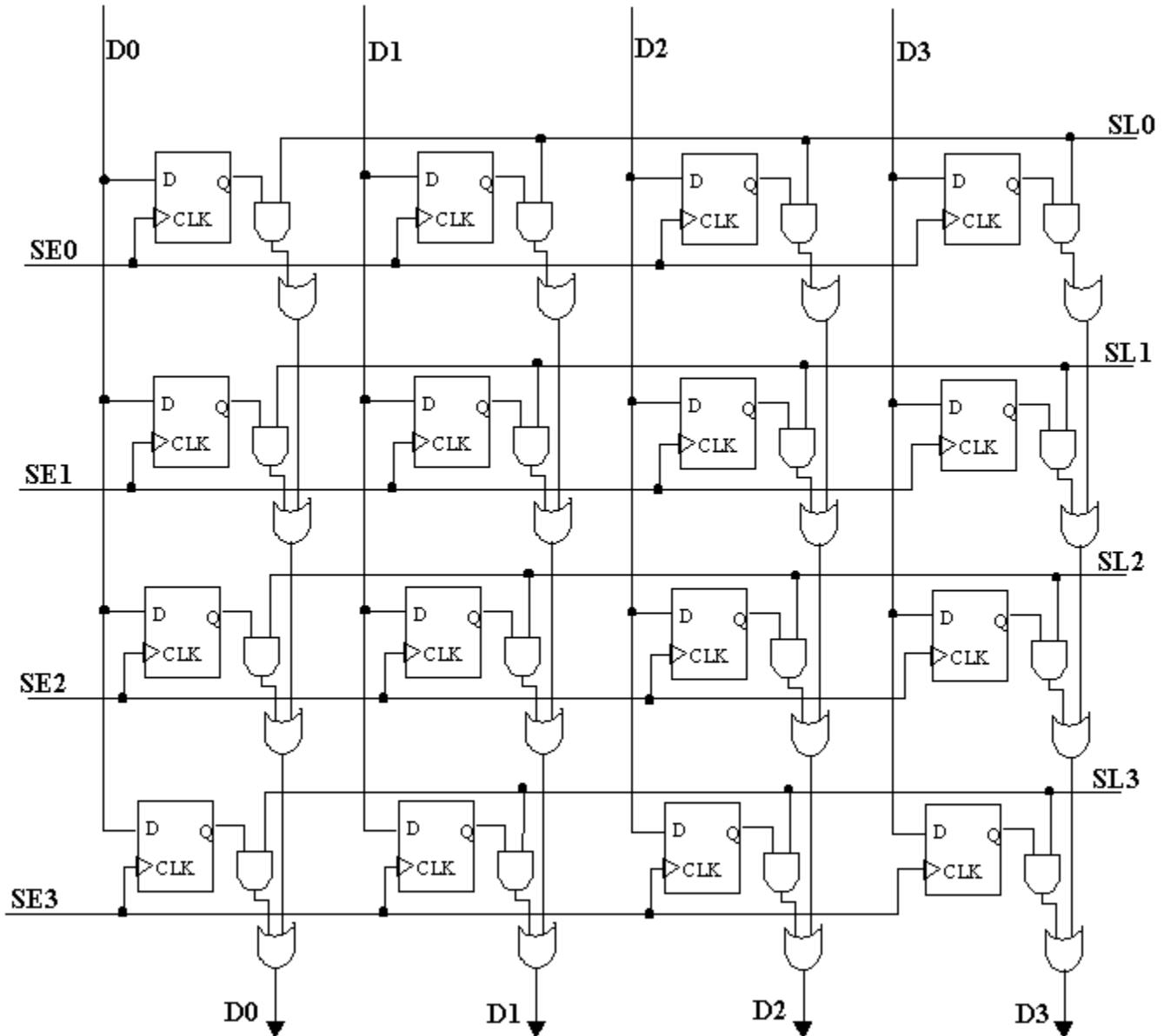
La mémoire proposée ci-dessous est une mémoire SRAM 4x4 (16 bits) utilisant des circuits du commerce : décodeur 2 vers 4 (circuit 74HC139), des portes logiques et des bascules D (circuits 74HC74). Pour des raisons d'encombrement le schéma de la mémoire a été scindé en 2.



Fonctionnement :

- Le bus d'adresse **A₁A₀** pilote un décodeur qui fournit les lignes de sélection des cellules. La même ligne sert pour l'écriture et la lecture.
- Les sorties peuvent être mises en haute impédance. Ceci permet d'éviter un conflit entre la sortie des bascules et la donnée présentée sur le bus bidirectionnel lors de la phase d'écriture. La mise en haute impédance peut être contrôlée depuis l'extérieur grâce au signal d'entrée **OE** (Output Enable).
- Le signal **CS** (Chip Select) doit être à 0 pour que la mémoire soit opérationnelle. S'il est à 1, la matrice n'est pas sélectionnée.

Détail de la matrice 4x4



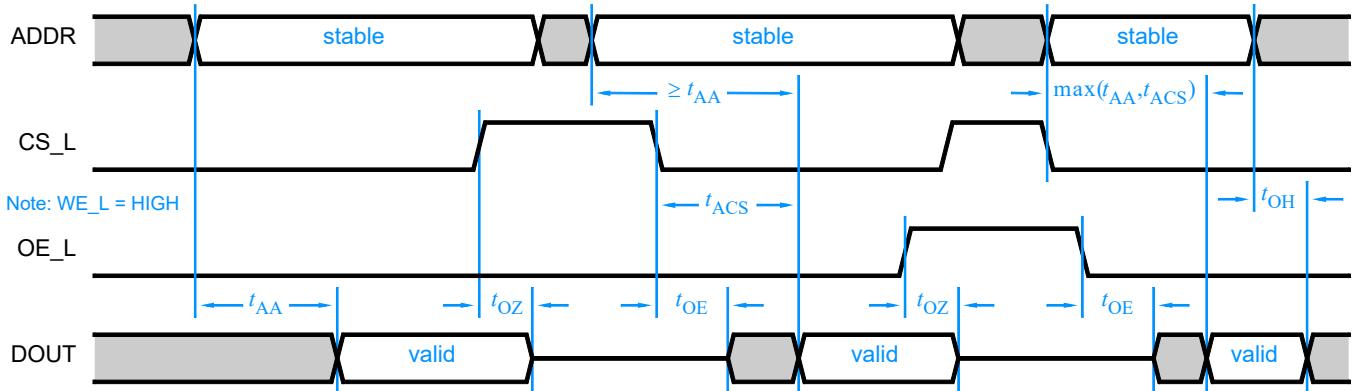
Les entrées de sélection en écriture de la matrice 4x4 sont notées SE_i et les entrées de sélection en lecture sont notées SL_i.

Le cycle d'écriture se fait sur le front montant de SE_i. Dans ce cas les 4 bascules de la ligne i sélectionnée mémorisent la donnée présente sur le bus de données (D3-D0).

Le cycle de lecture consiste à lire la sortie des 4 bascules D sélectionnées par le signal SL_i forcé à 1.

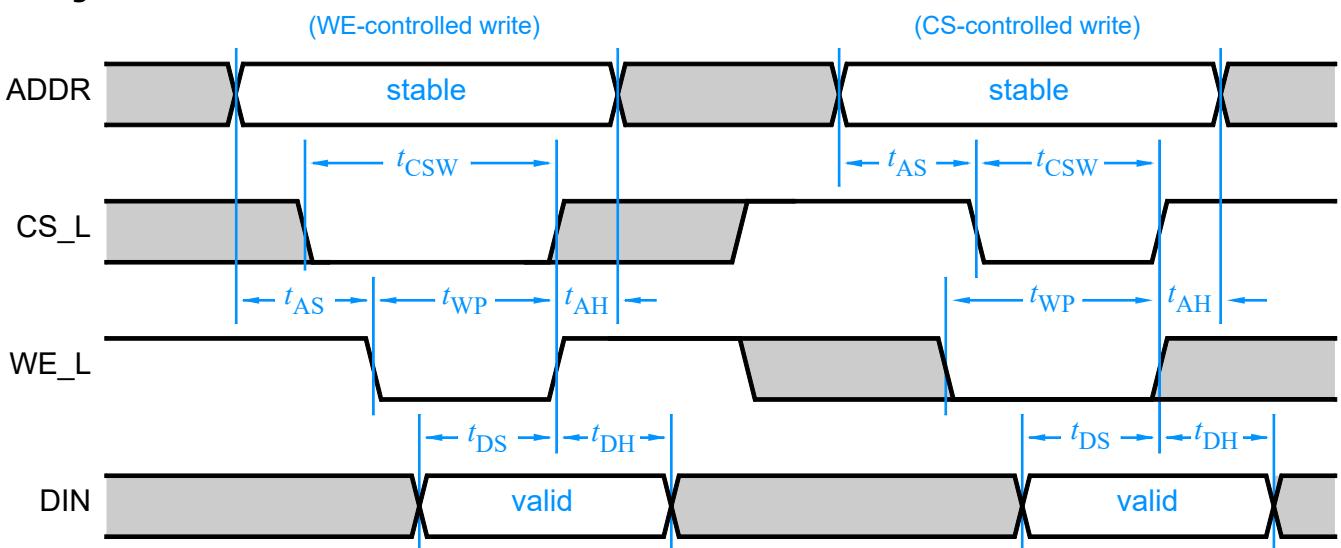
Timing d'une SRAM

Timing en lecture



Le cycle de lecture nécessite les conditions suivantes: CS et OE forcés à 0 et WE=1. La donnée apparaît après le temps tacs (dû au temps de propagation des différentes portes logiques de sortie) et disparaît dès que CS et OE repassent à 1 (après un certain retard).

Timing en écriture

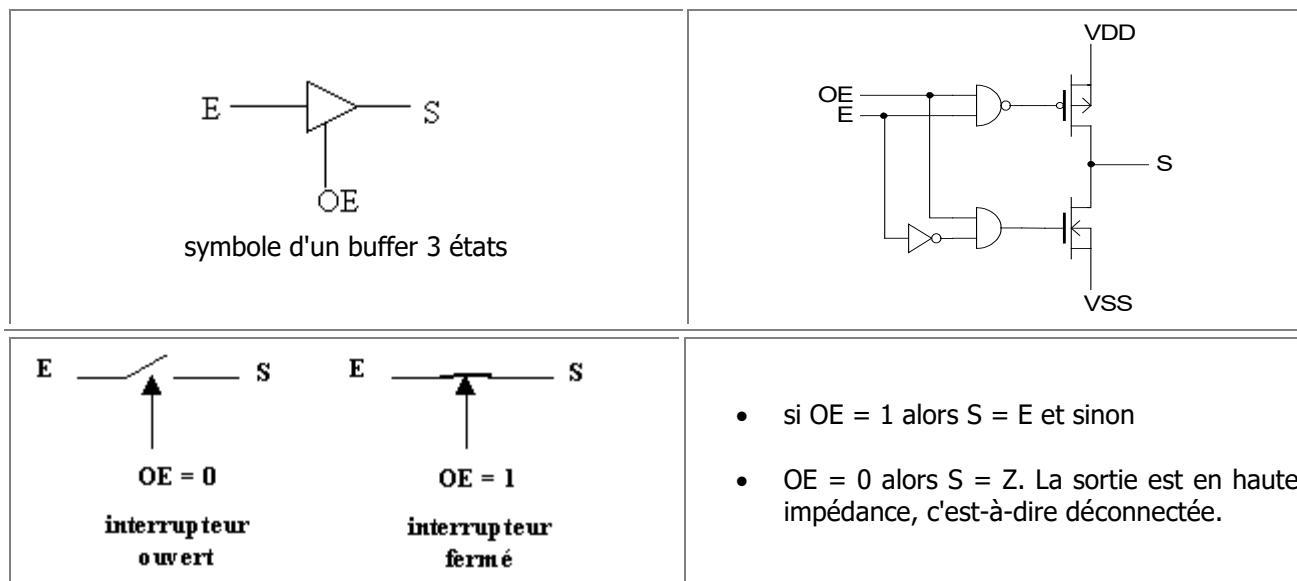


Le cycle d'écriture nécessite les conditions suivantes: CS et WE forcés à 0 et OE=1. Au moment où les signaux CS ou WE repassent à 1, la donnée est mémorisée. Le temps d'écriture minimal à respecter est donné par le constructeur par la donnée twp.

Les fabricants de SRAMs spécifient deux types de cycles write, le premier (WE controlled) est contrôlé par le signal WE qui devient actif après CS et remonte avant CS, c'est donc le signal WE qui commande l'écriture des données dans la SRAM. Le second (CS controlled) est contrôlé par le signal CS qui devient actif après WE et remonte avant WE, c'est donc le signal CS qui commande l'écriture des données dans la SRAM.

Notion de buffer 3 états (tri-state)

Le buffer 3 états est un élément essentiel des éléments mémoires. Il permet de déconnecter la mémoire du bus de données externe. Sans ce buffer 3 états il ne serait pas possible d'avoir un bus de données bidirectionnel (lecture ou écriture).



Les mémoires du commerce

NC	1	28	V _{CC}
A12	2	27	WE
A7	3	26	CS2
A6	4	25	A8
A5	5	24	A9
A4	6	23	A11
A3	7	22	OE
A2	8	21	A10
A1	9	20	CS1
A0	10	19	I/O8
I/O1	11	18	I/O7
I/O2	12	17	I/O6
I/O3	13	16	I/O5
V _{SS}	14	15	I/O4

Il existe différents types de mémoires SRAM dont la taille est comprise entre 64 kbits et 1Mbits avec un bus de données de taille 1, 4 ou 8 bits.

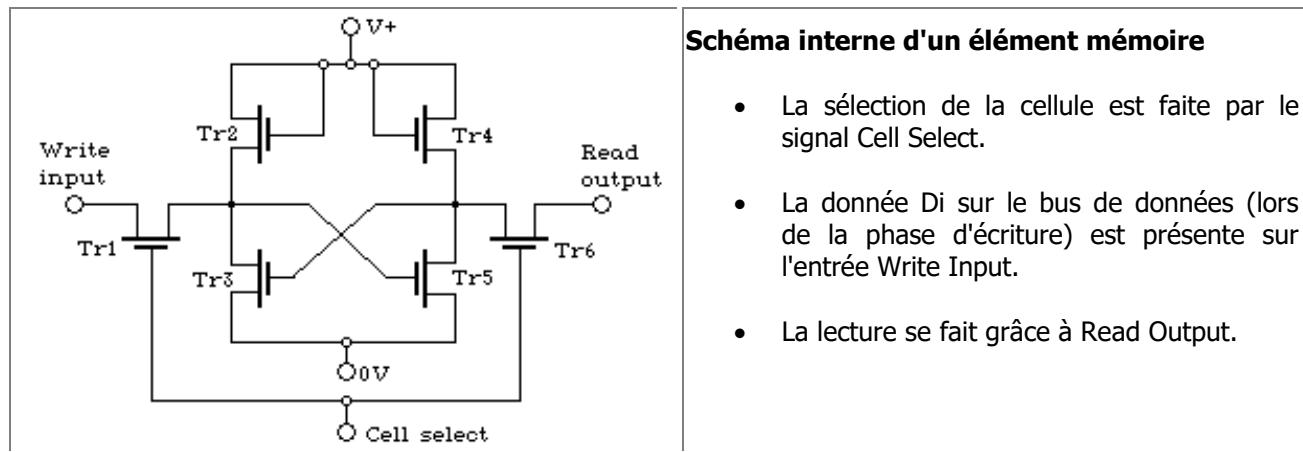
Quelle que soit la taille du bus de données, la capacité d'une mémoire est donnée en kbits ou en Mbits, donc en nombre d'unités mémoire.

Les temps d'accès varient entre 10 ns pour les mémoires CMOS rapides (High Speed CMOS) et 100 ns pour les mémoires les plus lentes (mais les moins consommatoires d'énergie).

Le circuit donné ci-contre est une mémoire très répandue dans les systèmes embarqués: la 6264 (64 kbits réparties en 8ko ou 8kbytes). Cette mémoire possède un bus d'adresse de 13 bits ($2^{13} = 8192$) et un bus de données de 8 bits. On retrouve les signaux de contrôle CS1, OE et WE actif à l'état bas, et un deuxième signal de sélection de boîtier CS2 actif à l'état haut.

Cellule mémoire réelle

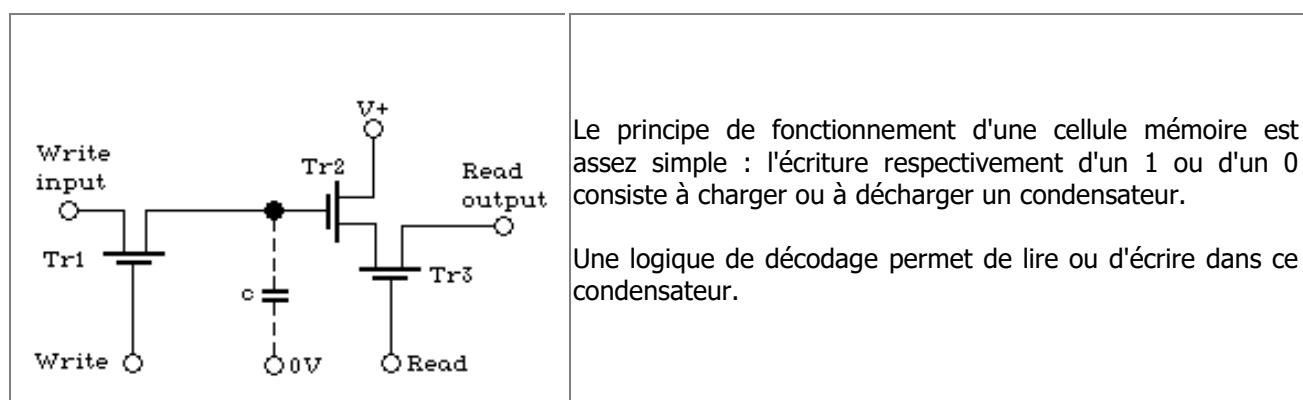
L'exemple donné dans ce chapitre permet de comprendre le fonctionnement et les chronogrammes d'une SRAM. Cependant les éléments mémoires des SRAM ne sont pas des bascules D (ce qui prendrait trop de place) mais un ensemble de 4 ou 6 transistors MOS dont la structure est donnée ci-dessous :



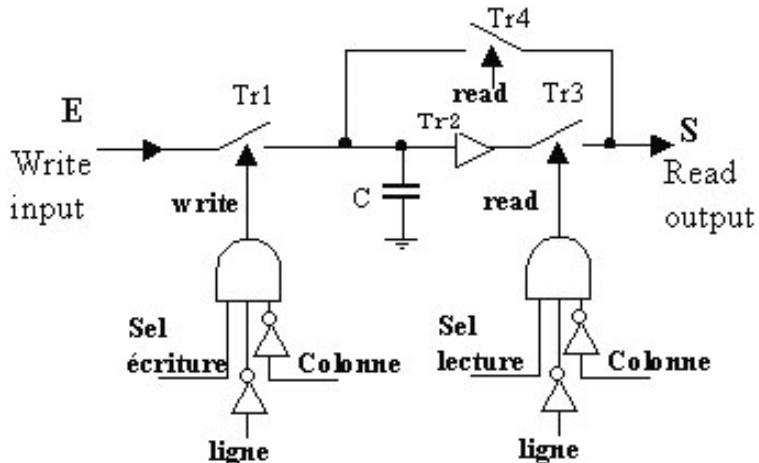
Mémoire DRAM

La mémoire DRAM est appelée RAM dynamique, en effet il faut sans cesse rafraîchir les données à l'intérieur, sous peine de perdre les données programmées.

Principe

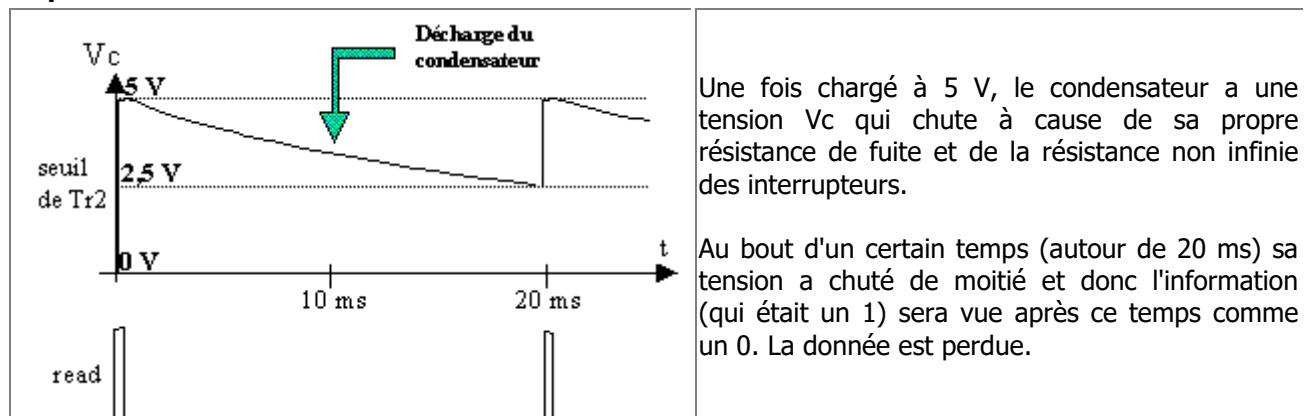


Pour expliquer le fonctionnement de cette cellule mémoire, remplaçons les transistors MOS par des contacts.



- La donnée à écrire dans la case mémoire est présente sur E (Write Input). Ce bit vient directement du bus de données. Si l'ordre d'écriture (write) est validé, c'est à dire si Sel écriture = 1 ET Ligne = 0 ET Colonne = 0 alors la tension condensateur $V_c=E$ (0 ou 5V). La donnée est enregistrée.
- La donnée à lire dans la case mémoire se retrouve sur S (Read Input) si Tr3 est fermé, c'est à dire si Sel lecture = 1 ET Ligne = 0 ET Colonne = 0. Cette donnée est de plus réinjecté sur C via Tr4.

A quoi sert Tr4 ?



Il est donc nécessaire de rafraîchir à intervalles réguliers les données. C'est son principe de fonctionnement qui a donné le nom à la DRAM (Dynamic RAM).

Une opération de lecture (read) permet de recharger la tension V_c et ce grâce à Tr2 qui fait office de comparateur (ou de buffer). Si la tension V_c est supérieure à 2,5 V alors $S=5V$ (S : tension en sortie de Tr3) et sinon $S=0V$. Cette tension en sortie de buffer est rebouclée sur C par Tr4, ce qui permet de le recharger.

Dans les premières mémoires DRAM, il fallait par un circuit extérieur à la mémoire (circuit dédié ou microprocesseur) faire une lecture de toutes les cases mémoires (rafraîchissement) toutes les 20 ms. Aujourd'hui les mémoires DRAM possèdent leur propre circuit de rafraîchissement interne. Cette opération est donc devenue transparente pour l'utilisateur.

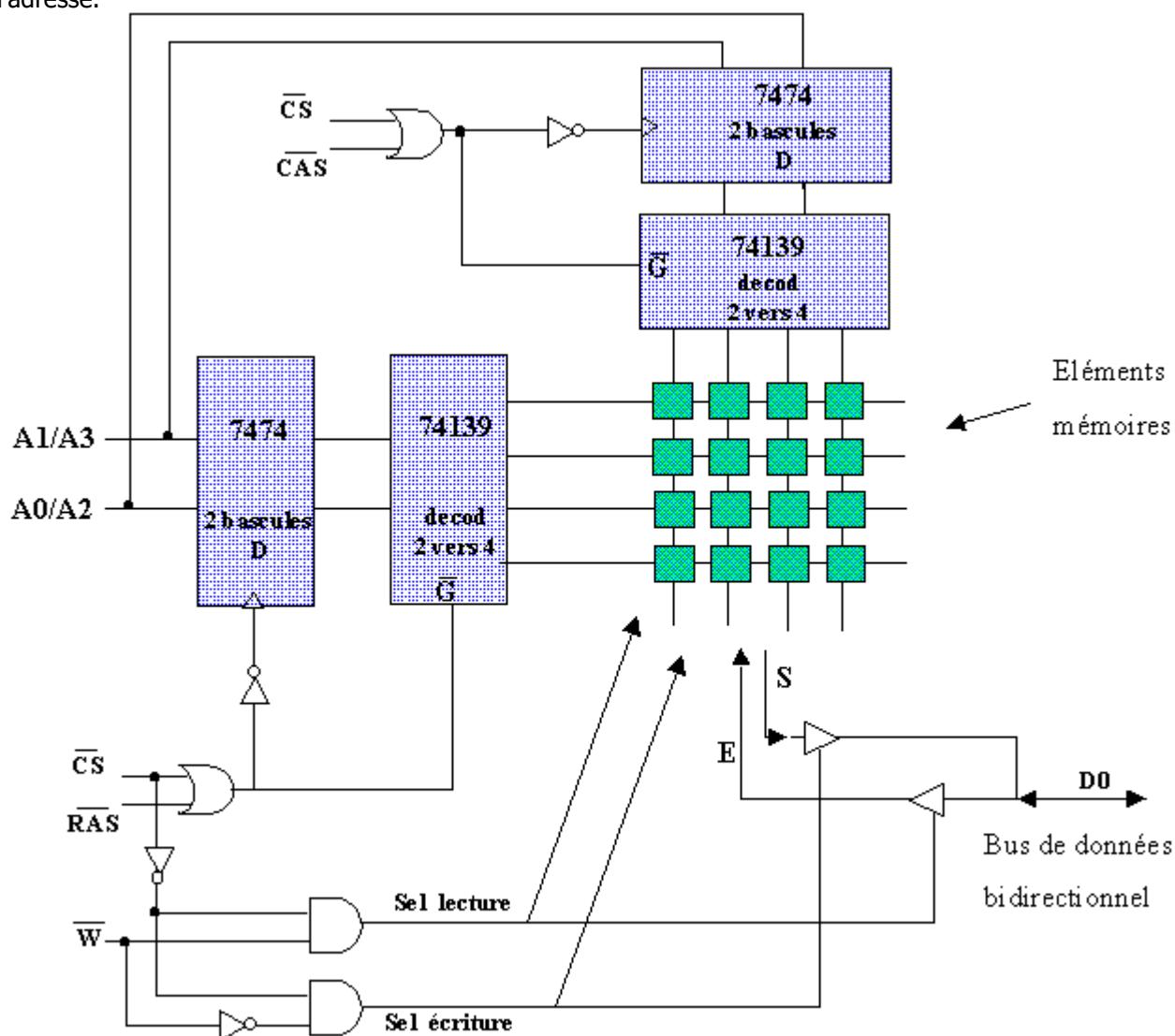
Avantage/ Inconvénient :

La mémoire DRAM demande 2 à 3 fois moins de transistors qu'une mémoire SRAM à capacité égale. Cependant elle est plus lente et demande un rafraîchissement des données régulier (ce qui a pour conséquence d'augmenter le temps d'accès moyen).

Exemple :

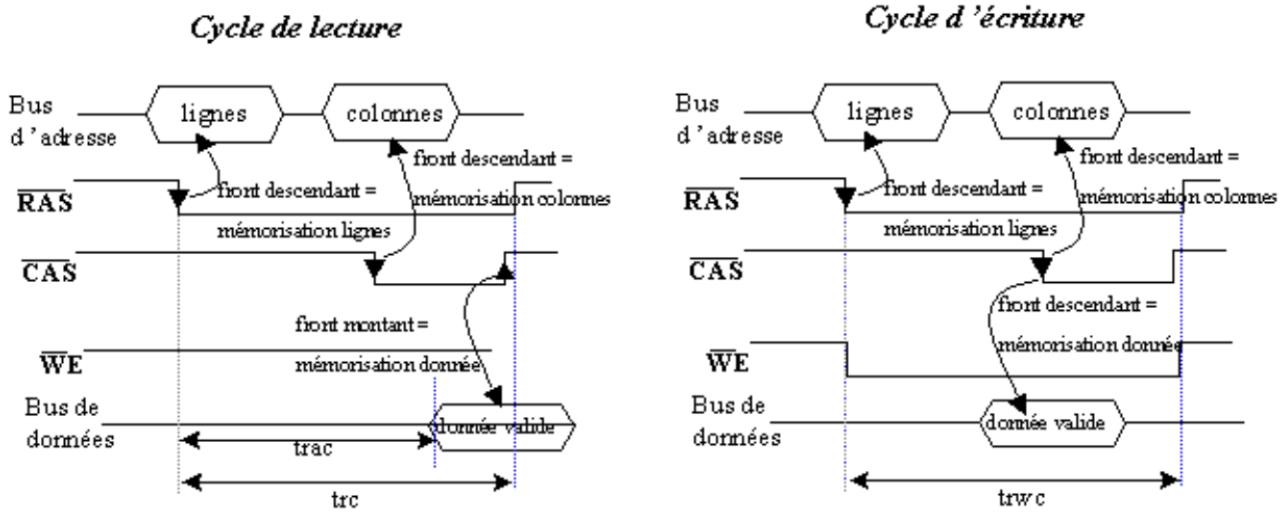
L'exemple de DRAM donné ci-dessous permet de comprendre les principales caractéristiques des DRAM ainsi que leur timing. Cette mémoire 16x1 bits possèdent un bus d'adresse multiplexé. Ainsi pour cette mémoire à 16 cases il faudrait 4 lignes d'adresse. Le choix fait par les constructeurs de DRAM et suivi jusqu'à présent est de ne prendre que 2 lignes d'adresses et d'envoyer d'abord l'adresse des lignes A3 A2 (/RAS mis à 0) puis l'adresse des colonnes A1 A0 (/CAS mis à 0).

Dans un souci d'économie et parce qu'à l'époque on ne savait pas faire des boîtiers ayant un nombre de broches importants, on a décidé de diviser par 2 le nombre de broches d'adresse. De part sa structure la DRAM avait une grande capacité mémoire et donc demandait un grand nombre de broches sur le bus d'adresse.



Une mémoire RAM est une matrice. Chaque élément mémoire peut être choisi par ces coordonnées dans la matrice (ligne - colonne). Un élément et un seul est activé si la ligne et la colonne sont à 0 (Ceci est réalisé grâce aux décodeurs 2 vers 4). De plus, on a besoin de bascules D (ou registres) pour mémoriser les adresses de lignes et de colonnes.

Timing d'une DRAM



Les mémoires DRAM possèdent les signaux de contrôles RAS (Row Adress Selection) sélection de ligne, CAS (Colon Adress Selection) sélection de colonne, WE (Write Enable) activation d'écriture, OE (Output Enable) et pour les nouvelles mémoires /CS (Chip Select). Dans le chronogramme /OE et /CS ne sont pas représentés.

Problèmes liés à la lecture ou à l'écriture des DRAM classiques

Comme on a vu jusqu'à présent, les opérations à réaliser pour lire une donnée sont les suivantes :

- Activation de RAS (Mise à 0) et donc mémorisation de l'adresse de la ligne.
- Activation de CAS (Mise à 0) et donc mémorisation de l'adresse de la colonne.
- Lecture de la case mémoire.
- Transfert et activation des buffers 3 états de sortie.
- Désactivation de la colonne (CAS remis à 1).
- Désactivation de la ligne (RAS à 1).

Il est évident qu'il existe sûrement des solutions plus rapides pour accéder à une case mémoire. L'arrivée des cartes mères PENTIUM et du bus système 66 MHz, pour des mémoires ayant un temps d'accès de 60 ns demandait 5 périodes d'attente ($T_{sys} = 1/66 \cdot 10^6 = 15$ ns). Pendant ce temps, le processeur devait générer des wait state pour attendre la donnée.

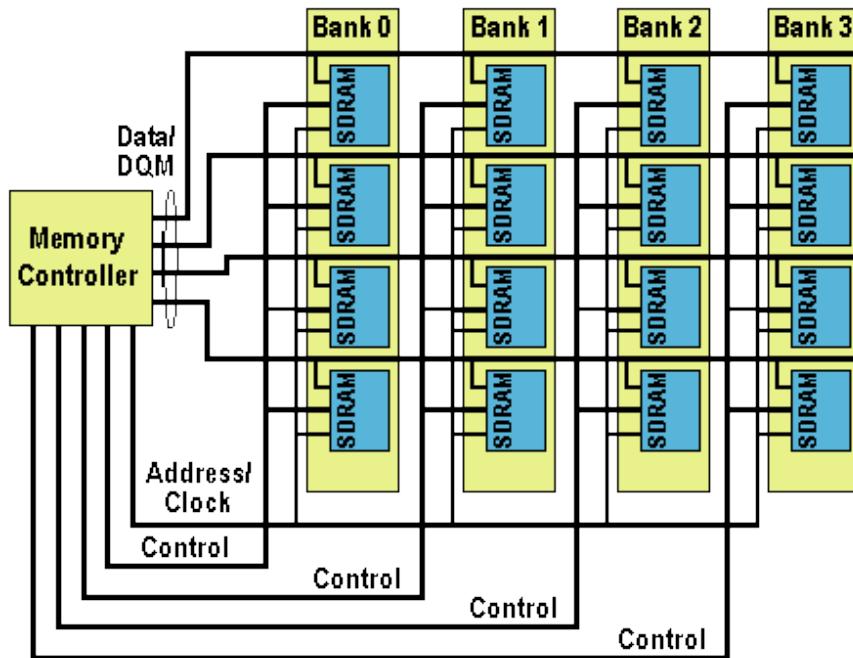
Mémoire SDRAM

SDRAM (Synchronous DRAM) est une mémoire synchrone. Le contrôleur de mémoire fournit à la mémoire SDRAM un signal d'horloge (exemple $f=100$ MHz pour PC100) qui va cadencer la lecture ou l'envoi des données. En effet, ces mémoires possèdent un compteur qui permet d'incrémenter les colonnes à chaque cycle d'horloge. Ces mémoires ne sont plus données avec un temps d'accès mais sous forme de vitesse d'envoi ou de réception des données. Après un ordre donné par le contrôleur de mémoire, la SDRAM fournit ou lit les données au rythme imposé par le CLK.

Pour les nouveaux bus systèmes 100 MHz, ces mémoires appelées PC100 auront un débit de données en 5 1 1 (50 ns d'attente pour la première donnée puis les 3 autres données arrivent toutes les 10 ns). Il est possible de lire en mode burst 1, 2 4, 8 ou toute la totalité d'une bank mémoire. Il est possible de sélectionner un deuxième bank pendant que la mémoire fournit les données d'un autre bank (interleaving mode), ce qui permet de diminuer le nombre de cycles d'attente. On remarquera que le temps d'accès global n'a que très peu diminué ($t_{acc} = 48$ ns) mais c'est en changeant la technique d'accès aux cases mémoires que l'on a pu faire diminuer le temps d'accès global. Ces mémoires utilisent le principe du pipe-line.

On donne ici un exemple de carte mère ayant 4 banks de données SDRAM de chacune 64 Mo. Soit une mémoire vive totale de 256 Mo. Ces mémoires sont gérées par le contrôleur mémoire qui fait parti du

Chipset de la carte mère. Comme sur tout système à microprocesseur, nous retrouvons un bus d'adresse (sur 32 bits) et un bus de données (sur 64 bits) communs à tous les banks.



Chaque barrette DIMM 64Mo possède 4 SDRAM (capacité 16x64Mbits) ou 8 SDRAM (capacités 8x8Mbits SDRAM) pour pouvoir fournir sur le bus de données 64 bits.

Chaque SDRAM possède 13 broches d'adresses A12-A0 et donc 2^{13} cases mémoires soit 8192 Mbits.

Les signaux de contrôles fournis par le contrôleur de mémoire ont pour but :

- choisir une barrette (bank) sur 4 lors d'un accès en mémoire (CS).
- contrôler le multiplexage d'adresse (RAS et CAS).
- lire ou écrire (broche W sur le bus de contrôle) la donnée.
- fournir la cadence d'envoie des données (CLK).

Mémoire NVRAM

Il existe un type intermédiaire de mémoire électronique à accès aléatoire, accessible en lecture et écriture comme la RAM, mais non volatile comme la ROM : la NVRAM.

Ces mémoires (Non Volatil RAM) sont des mémoires SRAM qui possèdent une pile lithium intégrée (d'une durée de vie de 10 ans). Les données sont donc gardées en mémoire même en cas de coupure de l'alimentation. Ces mémoires sont souvent fournies avec un timer (RTC) qui permet de donner la date et l'heure. Les applications de ces mémoires sont nombreuses (téléphone portable, ordinateur portable, ...).

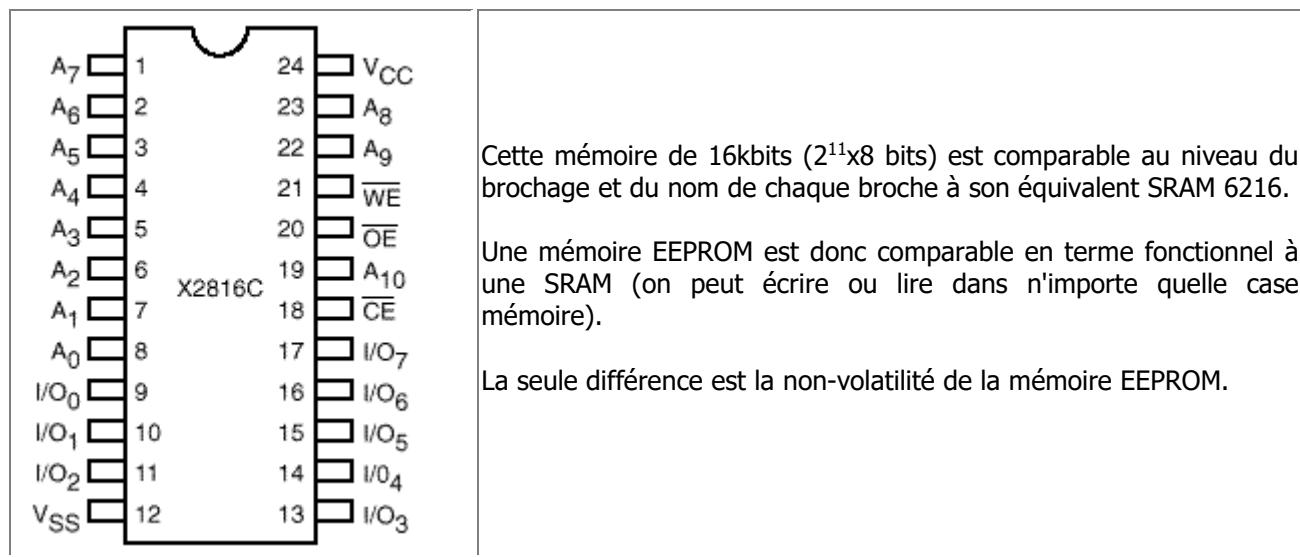
Littéralement, le terme RAM implique la possibilité d'un accès aléatoire aux données, par opposition à un accès séquentiel, comme celui d'une bande magnétique. En ce sens, ROM et NVRAM sont aussi de la RAM, mais cette interprétation littérale porte à confusion avec l'usage courant qui oppose RAM et ROM. Parfois, on utilise le sigle RWM (pour Read Write Memory, soit mémoire en lecture écriture) pour désigner de la RAM en mettant l'accent sur la possibilité d'écriture plutôt que l'accès aléatoire.

Mémoires EEPROM et FLASH

Bien qu'en théorie semblables, les mémoires EEPROM et EPROM Flash ne s'utilisent pas de la même façon et n'offrent pas les mêmes possibilités en termes de capacité et de coût, les mémoires flash étant moins chères et offrant des capacités bien supérieures aux mémoires E2PROM.

Mémoires EEPROM

Exemple : la mémoire X2816



On peut voir une EEPROM (ou E2PROM) de 2 façons :

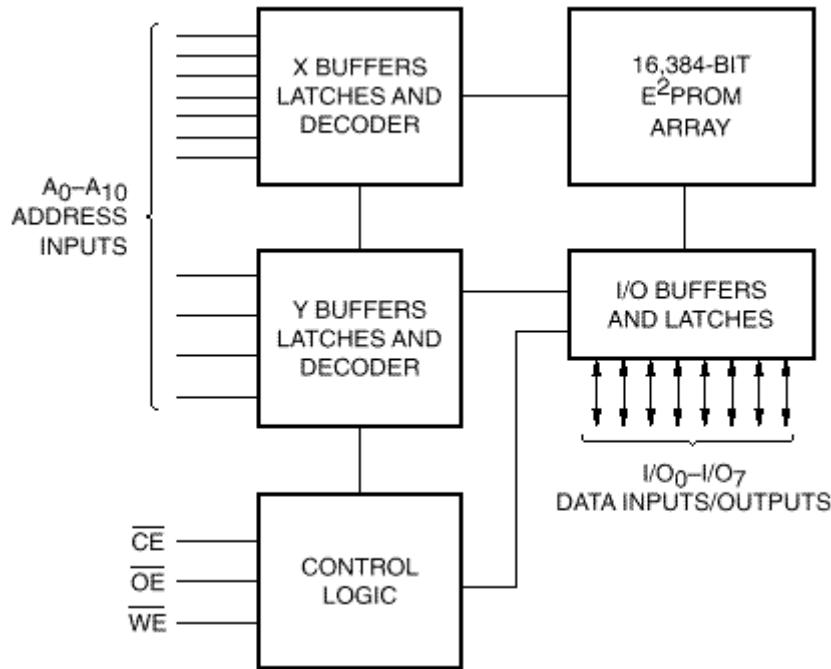
- soit comme une EEPROM effaçable électriquement, ayant donc les mêmes caractéristiques en lecture qu'une EEPROM ,et pouvant même être programmée par un programmeur d'EPROM (on remarquera que la broche /PGM de l'EPROM est remplacée par la broche /WE). Pour programmer la mémoire EEPROM, le circuit génère les tensions de programmation à partir du 5 V de l'alimentation. Nul besoin donc de fournir une tension externe de programmation.
- soit comme une mémoire SRAM dont le temps de lecture trc = 100 ns et le temps d'écriture twc=1 à 10 ms. On retrouve donc les mêmes chronogrammes qu'une mémoire SRAM, avec comme seule différence le temps d'écriture qui est élevé.

On peut retrouver une compatibilité entre les brochages d'une SRAM 6264, d'une EPROM 2764 et d'une E2PROM 2864. Ces mémoires (64kbits) peuvent donc être interchangées sur une carte, à condition de respecter les chronogrammes de fonctionnement propres à chacune des mémoires.

On utilise les mémoires EEPROM principalement pour sauvegarder des données en mémoire lorsqu'on coupe l'alimentation du système, comme l'EEPROM est non volatile, l'utilisateur pourra retrouver ses données à la prochaine mise sous tension du système.

Les EEPROM sont principalement utilisées avec une interface série (voir chap. mémoires série), ceci permettant de limiter la taille du composant, car le nombre de pattes du circuit est très faible.

Schéma interne de la mémoire X2816



Les mémoires Flash

La mémoire flash est une mémoire à semiconducteurs, non volatile et réinscriptible, c'est-à-dire une mémoire possédant les caractéristiques d'une mémoire vive mais dont les données ne disparaissent pas lors d'une mise hors tension (mémoire de masse). Ainsi, la mémoire flash stocke les bits de données dans des cellules de mémoire, mais les données sont conservées en mémoire lorsque l'alimentation électrique est coupée.

Les mémoires flash se caractérisent par le fait que des octets peuvent être adressés ou lus un par un, mais que les opérations d'écriture ou de suppression (effacement) ne peuvent avoir lieu que par blocs de plusieurs kOctets. Les temps d'accès en lecture de ces mémoires sont approximativement deux fois plus élevés que ceux des mémoires dynamiques. Ils se situent actuellement autour de 70 ns. Les mémoires flash ne supportent qu'un nombre limité de cycles de programmation et d'effacement (autour de 100.000). En règle générale, la conservation des données sur ces mémoires est garantie pendant une période de 10 ans.

Les flashes utilisent comme pour les mémoires EEPROM un seul transistor MOS par bit, ce qui explique les capacités mémoires comprises entre 128kbits et 2Gbits. La dénomination des boîtiers suit le même principe que les mémoires vues précédemment. Ainsi le 28F256 peut remplacer un 62256 (mémoire SRAM) ou un 28256 (mémoire EPROM). Attention, ceci n'est vrai que pour une compatibilité broche à broche et non au niveau du timing et des fonctions internes à chaque technologie.

La mémoire flash est donc un type d'EEPROM qui permet la modification de plusieurs espaces mémoires en une seule opération. La mémoire flash est donc plus rapide lorsque le système doit écrire à plusieurs endroits en même temps.

Elle existe sous deux formes: flash NOR et NAND, d'après le type de pont logique utilisé pour chaque cellule de stockage.

Les Flash NOR

La flash NOR est la première à être développée, elle est inventée par Atmel en 1988. Les temps d'effacement et d'écriture sont longs mais elle possède une interface d'adressage permettant un accès aléatoire à n'importe position. Elle est adaptée à l'enregistrement des logiciels embarqués qui sont rarement mis à jour, comme dans les appareils photo numériques, les organisateurs personnels, les ordinateurs

(BIOS)... Elle peut supporter de 10 000 à 100 000 cycles d'effacement. Les Compact Flash et SmartMedia sont basées sur ce système.

Les Flash NAND

La flash NAND développée par Toshiba suivit en 1989. Elle est plus rapide à l'effacement et à l'écriture, offre une plus grande densité, un coût moins important par bit et une durée de vie dix fois plus importante. Toutefois son interface d'entrée / sortie n'autorise que l'accès séquentiel aux données. Elle est donc utilisée pour le stockage d'informations (images d'appareils photo, fichiers MP3...) et est moins utile en tant que mémoire pour ordinateurs. Les mémoires MultiMediaCard, Secure Digital et Memory Stick sont basées sur le format NAND.

Les nouveaux appareils grand public (téléphones portables, cartes mémoire des appareils de photos et caméscopes...) demandent de plus en plus de capacité mémoires pour les interfaces graphiques et nouvelles fonctions. Les mémoires flash sont donc en pleine expansions.

	NOR	NAND
Taille de la cellule	Grande	Petite
Interface	Bus mémoire complet	E/S (x8/x16)
Exécution du logiciel en mémoire	Oui	Non, mais les circuits de nouvelle génération peuvent exécuter un programme d'amorçage
Temps d'accès	Aléatoire rapide: 70 ns (typ.)	Aléatoire lent (25 µs), séquentiel rapide (50 ns)
Programmation d'un mot	8 µs/mot environ	-
Programmation de mots multiples	Lente: 4 ms pour 512 octets	Rapide: 200 µs pour 512 octets
Temps d'effacement	Lent: de 700 ms à 1 s par bloc de 64 Ko	Rapide: 2 ms par bloc de 16 Ko
Consommation	Elevée	Faible
Prix	Elevé	Faible
Endurance et fiabilité	10^4 à 10^5 cycles d'effacement	10^5 à 10^6 cycles d'effacement Requiert une gestion des blocs défectueux. Inversion de bits: nécessite un code détecteur/correcteur d'erreurs
Simplicité d'emploi (hardware)	Oui	Non
Intégration système	Simple	Complexe. Nécessite le portage d'un driver SSFDC.

Tableau 7 Comparaison des mémoires flash type NOR et NAND

Les différents types de mémoires dans un PC

Le microprocesseur doit échanger en permanence des données avec la mémoire centrale pour exécuter les programmes.

Rappelons à ce sujet qu'un programme se définit comme une suite ordonnée d'instructions. Et que chaque instruction est elle-même double, faite d'un opérande (la donnée) et d'un opérateur (une opération).

Instruction = opérande et opérateur

Programme = suite ordonnée d'instructions

Dans cette partie du cours consacrée aux mémoires dans les systèmes embarqués nous nous concentrerons sur la mémoire centrale. Elle constitue avec le microprocesseur l'unité centrale.

Unité centrale = Microprocesseur + mémoire centrale

Il existe différents types de mémoires dans un système embarqué. Chaque type correspond à une fonction précise, chaque fonction imposant des performances différentes, les différents types de mémoire seront analysés en fonction de leurs performances.

Critères de classification :

Ils permettent de traduire les performances évoquées ci-dessus et donc d'attribuer au matériel tel ou telle fonction. Imaginons que vous souhaitez transporter une information d'un point A vers un point B.

- **Le temps d'accès** : C'est le temps nécessaire pour accéder à l'information sur le dispositif. C'est le temps que vous mettrez pour accéder en A ou est stockée l'information.
- **La vitesse de transfert** : Elle définit le temps nécessaire pour transférer l'information de la mémoire vers le composant appelant. C'est la vitesse à laquelle vous rapporterez l'information que vous aurez prise en A.
- Le temps que vous mettrez pour ramener une information dépendra donc du temps que vous mettrez pour y accéder et de la vitesse à laquelle vous ramènerez cette information. Il en est de même dans toute mémoire.
- **La capacité de stockage** : C'est la quantité d'information que vous pourrez stocker dans le dispositif. Elle est très variable de 1 bit pour les registres du processeur jusqu'à plusieurs centaines de Terra Octets pour les systèmes d'archivage de haute capacité.
- **Le coût par bit** : C'est le coût nécessaire pour stocker 1 bit d'information. Comme la capacité de stockage, en fonction du type de mémoire le coût par bit va varier d'un facteur allant de 1 à 1000. La solution idéale serait effectivement de disposer de mémoire ultra rapide à très grande capacité de stockage. Si nous étions capables de produire ce type de mémoire, son coût serait tel que le système n'aurait aucune viabilité économique.
- **Le caractère volatil ou non volatil** : En fonction de ce que vous souhaiterez faire de ce dispositif le caractère volatil ou non volatil de la mémoire sera important. Rappelons qu'une mémoire volatile voit tout son contenu disparaître lorsque l'alimentation électrique du système est coupée.

A titre d'exemple vous trouverez dans le tableau ci-dessous les caractéristiques de différents types de mémoire fréquemment rencontrés dans un système informatisé : Une RAM, une carte mémoire Memory Stick (Flash), un CR-ROM et un disque dur.

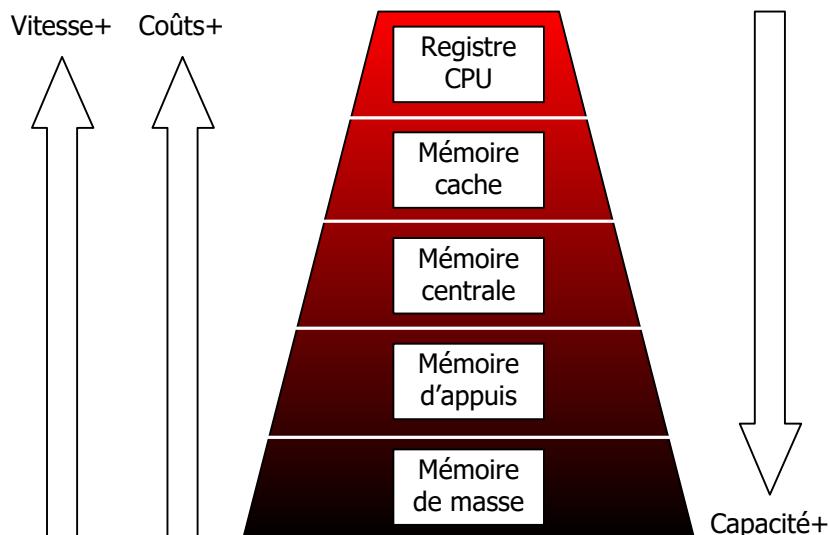
	Temps d'accès	Vitesse de transfert	Coût par Go (Sfr)
DRAM	50 ns	1400 Mo/sec	200 SFr
Memory Stick (Flash)	20 µs	15 Mo/sec	100 SFr
CD-ROM	60 ms	7.8 Mo/sec	1 SFr
Disque dur	8 ms	50 Mo/sec	0.60 SFr

Tableau 8 Caractéristiques de différents types de mémoires

Attention : en raison de l'évolution permanente de la technologie et du marché les valeurs de ce tableau évoluent sans cesse. Elles ne sont données ici que pour fixer les idées sur les performances relatives des différents types de mémoires.

Différents types de mémoires

En fonction des caractéristiques décrites en introduction et de leur fonction au sein du système informatisé on décrira 5 grands types de mémoire que l'on organisera selon la pyramide suivante :



On trouvera à la partie haute de la pyramide les dispositifs mémoire les plus coûteux mais également les plus rapides et à la partie basse les dispositifs les moins chers, les plus lents mais aux capacités de stockage maximales.

- **Registre du CPU :** Situés dans le microprocesseur, il s'agit de composants électroniques très rapides utilisés pour le stockage des résultats intermédiaires au cours des calculs (registres mémoire).
- **Mémoire cache :** Dispositif intermédiaire, toute mémoire dite cache a pour objet de tamponner les asynchronismes de vitesse entre les deux composants qui l'entourent à savoir la mémoire centrale et le microprocesseur.
Le type de mémoire utilisé pour la cache est la SRAM. Comme étudié précédemment, la SRAM est une mémoire extrêmement rapide elle est très coûteuse à produire c'est donc pour cette raison qu'elle essentiellement limitée à la mémoire cache.
- **Mémoire centrale :** Dispositif fondamental du fonctionnement de tout système informatisé, la mémoire centrale stocke toutes les instructions utilisées par l'unité centrale de traitement. Elle est moins coûteuse que la mémoire cache. Ses temps d'accès sont également plus longs.
Le type de mémoire utilisé pour la mémoire centrale est la DRAM. Comme étudié précédemment, la DRAM est une mémoire plus lente que la SRAM (rafraîchissement et accès au bus d'adresses en

deux étapes lignes et colonnes) elle est également moins coûteuse à produire (moins de transistor par bit).

- **Mémoire d'appui :** La mémoire d'appui est un dispositif identique à la mémoire cache. La mémoire d'appui sert à équilibrer les asynchronismes de vitesse entre la mémoire centrale située en aval et les mémoires de masses situées en amont. Elle est cependant différente de la mémoire cache du processeur en terme de performance (+ lente) et de coût (- coûteuse).
- **Mémoire de masse :** C'est un dispositif non labile sur lequel vous stockerez les programmes que vous installerez sur votre ordinateur et les données (textes, sons, images...). Comme sur tout support non labile l'information y sera conservée lors d'une coupure de courant. C'est le dispositif ayant le plus faible coût par bit d'information stockée. Les exemples sont nombreux et bien connus : disquette, disque dur, bande magnétique.

Fonction de la mémoire cache

Qu'il s'agisse de la mémoire cache ou de la mémoire d'appui, les principes généraux que nous évoquerons ci-dessous restent vrais. La mémoire cache est une mémoire intermédiaire ou mémoire tampon qui a pour rôle d'optimiser le fonctionnement du microprocesseur en diminuant le temps où ce dernier ne calcule pas. Prenons un exemple simple :

Soit un processeur fonctionnant à 200MHz : Cela signifie qu'il est capable d'effectuer une opération toutes les 5 nanosecondes. Temps nécessaire pour une opération, si l'on considère que le microprocesseur peut faire 200×10^6 opérations en une seconde :

$$T = 1/(200 \cdot 10^6) = 5 \cdot 10^{-9}$$

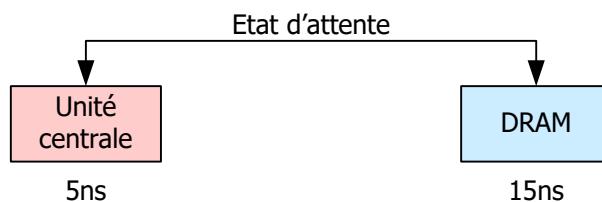
Soit une DRAM avec laquelle il échange qui présente un temps d'accès de 15 nanosecondes. Lors du fonctionnement du microprocesseur que se produit-il ?

T0 : La première instruction est transmise au processeur.

T+5ns : Fin du calcul de la première instruction.

Il faut ensuite 10ns supplémentaires pour accéder à la mémoire dont le temps d'accès est de 15ns ($5+10=15$). Durant ces 10 ns le processeur est dit en état d'attente, il ne fait rien.

Sur cette base c'est donc 66% du temps de fonctionnement du processeur qui est inutilisé...



Pour optimiser ce mode de fonctionnement nous allons maintenant ajouter une faible quantité de mémoire cache (SRAM) entre la DRAM et le processeur. Souvenez-vous que cette dernière étant très coûteuse il n'est pas envisageable de la substituer à la DRAM.

Cette mémoire cache aura pour particularité d'être très rapide et de disposer d'un temps accès de 5 nanosecondes. Comment va alors fonctionner notre système ?

T0 : La première instruction est transmise au processeur.

T0+5ns : Fin du calcul de la première instruction

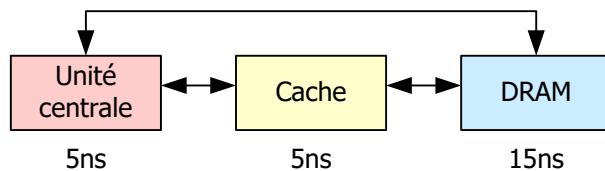
La mémoire cache est alors disponible pour fournir une seconde instruction au processeur.

T0+10ns : Fin du calcul de la seconde instruction

La mémoire cache fournit alors une troisième instruction au processeur

T0+15ns : A ce moment la DRAM fournit une nouvelle instruction (le temps d'accès de 15nanosecondes est atteint)...

Elle remplit de nouveau la mémoire cache qui va alimenter le processeur....Et ainsi de suite...Vous voyez donc qu'il suffit que la mémoire cache contienne un petit nombre d'instructions à disposition pour que le processeur ne soit jamais en état d'attente. Son fonctionnement est alors optimal.



Les mémoires série

Les mémoires séries EEPROM sont très utilisées dans les téléviseurs, consoles de jeux, chaînes HiFi, etc... Ces mémoires de faible capacité (quelques ko) permettent de sauvegarder les paramètres utilisateurs. Le principal avantage de ces mémoires est leur prix du à la taille du boîtier (autour de 8 broches) et ceci grâce à la faible taille de leur bus d'adresse et de données. Il existe plusieurs protocoles de communication entre la mémoire série et le microprocesseur qui correspondent tous à une communication synchrone. Les plus utilisés sont le bus I2C, le bus SPI ou le bus Microwire.

L'interface série est plus lent qu'une interface parallèle, comme que les données sont échangées en série, mais cet inconvénient n'est pas très important lorsque que l'on utilise une EEPROM, pour stocker des paramètres, ces paramètres n'étant pas accédés très souvent, on ne va pas ralentir le système.

Exemple : mémoire de Microchip 25LC080 sur bus SPI (Synchronous Protocol Interface).

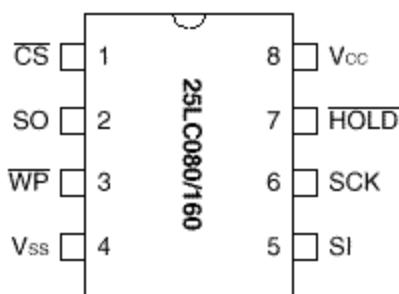
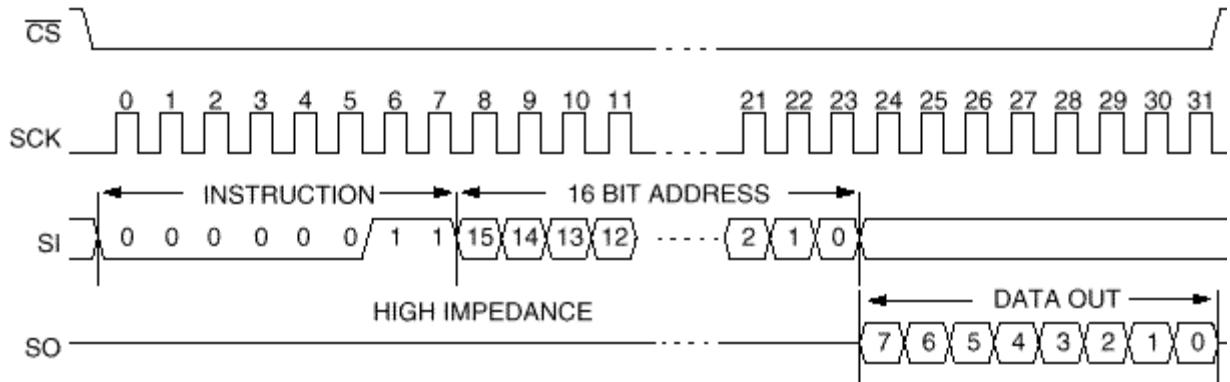


Figure 66 mémoire 1024kx8bits, fclkmax = 3 MHz, techno CMOS.

Phase de lecture

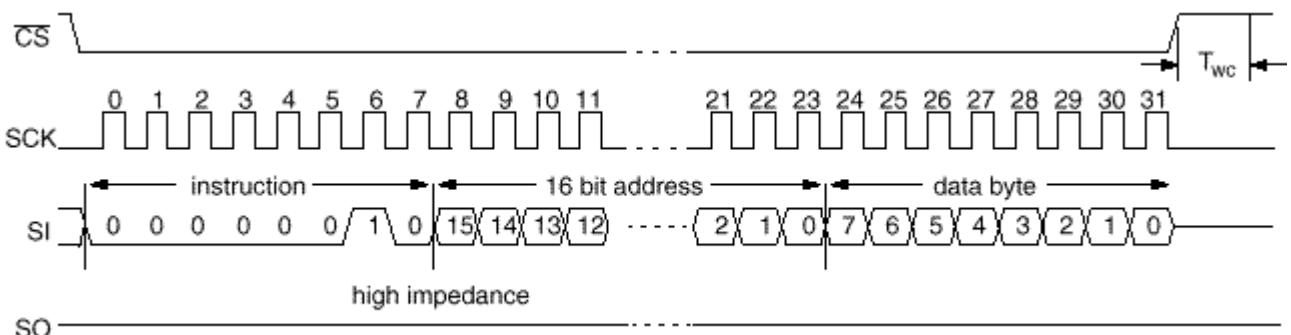
La phase de lecture se fait sur 32 cycles d'horloge. Le premier octet envoyé correspond à l' instruction (lecture ou écriture ou autres), les 2 octets suivants sur la broche SI correspondent à l'adresse de la donnée

à atteindre, puis au 24ème front montant d'horloge, la donnée à lire peut être lue sur la broche SO dans le sens MSB-LSB jusqu'au 31ème front montant de SCK.



Phase d'écriture

32 cycles d'horloge, premier octet = instruction d'écriture, les 2 octets suivants sur la broche SI correspondent à l'adresse de la donnée à atteindre, puis au 24ème front montant d'horloge, la donnée à lire peut être écrite dans le sens MSB-LSB jusqu'au 31ème front montant de SCK.



Description d'une mémoire SRAM en VHDL

Une SRAM à une largeur déterminée par le vecteur data, dans notre exemple 16 bit.

Le nombre de case mémoire est défini par 2^n , où n est la taille du vecteur d'adresse, dans notre exemple $2^4 = 16$ case mémoires.

Le signal `ram_tmp` est soit une variable, soit un signal, de type tableau (array), dans notre exemple `t_memory`, où la largeur du tableau est la largeur de la mémoire, et la longueur, est le nombre de cases mémoires.

Il faut savoir qu'en synthétisant cette description, on utilise 16x16 bascules D, à savoir 256 bascules D, une par bit mémorisé, les mémoires SRAM sont donc très gourmandes en matériel. Certaines FPGA ont déjà des blocs de mémoire SRAM incluses dans le circuit, il faut donc utiliser ces blocs de préférence, afin de ne pas utiliser trop de ressources logiques de la FPGA.

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY sram IS
PORT (we_n      : IN std_logic;
      oe_n      : IN std_logic;
      cs_n      : IN std_logic;
      data      : INOUT std_logic_vector(15 DOWNTO 0);
      addr      : IN std_logic_vector(3 DOWNTO 0));
END sram;

ARCHITECTURE Behavioral OF sram IS
  --EXEMPLE AVEC SRAM de 16x16
  TYPE t_memory IS ARRAY (0 TO 15) OF std_logic_vector(15 DOWNTO 0);
  --SIGNAL ram_tmp : t_memory;
  SIGNAL we_int_n : std_logic;
  SIGNAL oe_int_n : std_logic;
BEGIN
  we_int_n <= cs_n OR we_n;
  oe_int_n <= cs_n OR oe_n;

P1:Process (we_int_n, oe_int_n, data, addr)
VARIABLE ram_tmp : t_memory;
BEGIN
  --BUFFER TRI_STATE SUR DATA EN SORTIE
  data <= (OTHERS => 'Z');
  --ECRITURE ASYNCHRONE DANS LA SRAM
  IF (we_int_n'event AND we_int_n = '1') THEN
    ram_tmp (to_integer (unsigned(addr))) := data;
  END IF;
  --LECTURE ASYNCHRONE DE LA SRAM AVEC TEMPS D'ACCES DE 70 ns VALIDE EN
  --SIMULATION UNIQUEMENT
  IF (oe_int_n = '0') THEN
    data <= ram_tmp (to_integer (unsigned(addr))) AFTER 70 ns;
  END IF;
END Process;
END Behavioral;

```

Le Décodage d'adresse

Dans ce chapitre, on s'intéresse plus précisément au côté hardware des systèmes numériques. Nous verrons comment est conçu un système dédié, et comment on procède afin d'éviter des conflits entre les divers périphériques.

Un système dédié

Un système dédié est un circuit à base d'un microcontrôleur (ou microprocesseur) auquel on ajoute des circuits intégrés périphériques (FLASH, RAM, PORTs I/O, FPGA, UART,...). Tous ces circuits réalisent une tâche précise spécifiée par le programme (qui est lui-même contenu dans un circuit périphérique, la FLASH).

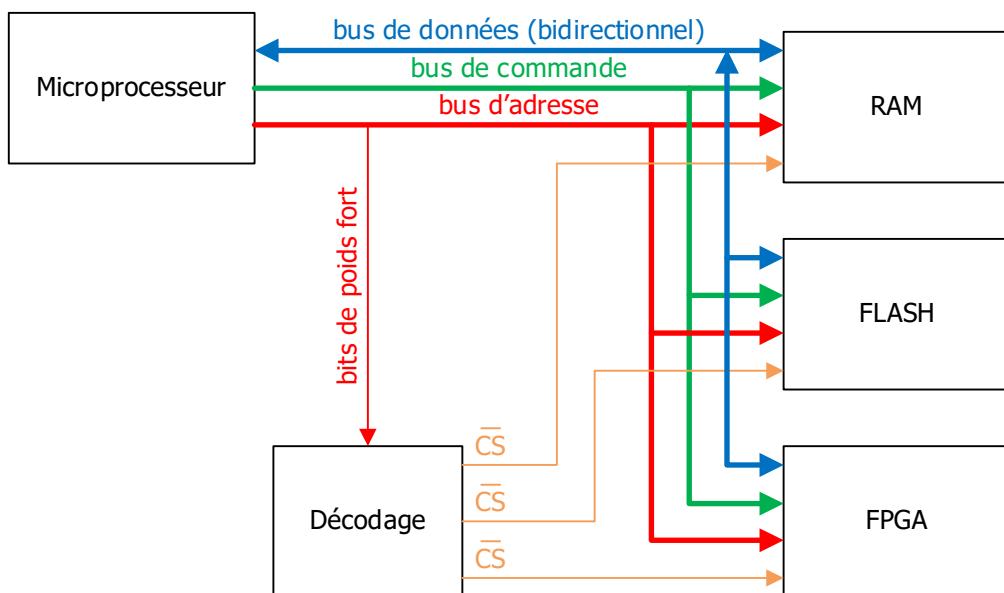


Figure 67 Schéma bloc d'un système dédié avec décodeur d'adresses.

Les bus d'Adresses et de Données

Le microprocesseur ou le microcontrôleur (μ P) accède à ses périphériques séquentiellement (il n'en accède jamais deux en même temps), pour chercher des informations et/ou lui en fournir. Pour accéder aux périphériques, le μ P utilise les bus d'Adresses et de Données.

Les bus d'adresses et de données sont communs pour tous les périphériques. Comme ils sont communs c'est toujours le μ C qui met l'adresse sur le bus d'adresse. Le bus de données peut être "drivé" par n'importe quel périphérique mais un seul périphérique à la fois peut envoyer des données. Dans le cas où plusieurs circuits périphériques "driveraient" le bus de données en même temps, il y aurait très forte possibilité de court-circuit sur des lignes de données (ex: un périphérique tente de mettre '1'= 3.3V et un autre '0'=GND donc court-circuit).

Il faut donc éviter que deux circuits intégrés utilisent simultanément le bus de données, c'est pourquoi la plupart des circuits intégrés qui sont utilisés avec un bus de données ont une pin (patte) qui permet de désactiver le circuit intégré (fait comme s'il n'était pas présent). Cette patte se nomme en général \overline{CS} ou \overline{CE} , qui signifie Chip Select (sélection du boîtier) ou Chip Enable (activation du boîtier), et est généralement active à l'état bas. En réalité, cette entrée lorsqu'elle est à l'état haut (inactive) met les pattes de données du circuit intégré dans l'état haute impédance.

Partage de la plage mémoire (Memory Map)

Un circuit de décodage est donc nécessaire pour déterminer quel périphérique sera activé. Ce circuit de décodage est réalisé par le concepteur. Il est en quelque sorte le contrôleur du "trafic".

Chaque périphérique est activé seulement, lorsque l'adresse sur le bus est contenue dans la plage des adresses qui lui ont été réservées dans le memory map (plan mémoire).

Prenons comme exemple un memory map tel qu'ilustré dans la figure suivante. Il s'agit du memory map d'un microcontrôleur 68HC12 auquel on a ajouté deux RAMs externes de 4Koctets chacune, une mémoire ROM de 32K, et dont on a déplacé la RAM interne et les 64 registres de configurations. En résumé: Le décodeur permet d'activer un seul périphérique à la fois afin d'éviter des conflits sur le bus de données. Il utilise normalement les lignes d'adresses les plus significatives (MSB) pour faire la sélection.

Quand un circuit n'est pas activé, ses lignes de données sont dans un état haute impédance (comme si elles n'étaient pas branchées). Les circuits périphériques ont une "patte" permettant de les désactiver.

\$0000	RAM1 (4k)
\$0FFF	RAM2 (4k)
\$1000	
\$1FFF	
\$2000	RAM Interne
\$20FF	
\$3000	
\$303F	Registres internes
\$8000	
\$FFFF	ROM (32k)

- Les deux RAMs externes sont identiques. Elles contiennent physiquement 4096 bytes d'informations. Pour accéder ces informations il faut 12 lignes d'adresses ($2^{12} = 4096 = 4K$). Donc ces deux RAMs répondent aux adresses \$000 à \$FFF.
- Le décodeur doit donc activer la RAM1 lorsque l'adresse que demande le uP est entre \$0000 et \$0FFF, et activer la RAM2 lorsque le CPU accède une adresse entre \$1000 et \$1FFF.
- La différence entre les 2 RAMs réside dans les 4 bits les plus significatifs de l'adresse. Lorsque l'adresse demandée par le CPU commence par \$0xxx c'est la RAM1, et quand l'adresse commence par \$1xxx c'est la RAM2.
- Il faut donc concevoir un circuit de décodage qui active la RAM1 lorsque l'adresse commence par \$0xxx et active la RAM2 quand l'adresse commence par \$1xxx.

Pour la ROM, elle a physiquement 15 lignes d'adresses ($2^{15} = 32768 = 32K$). Il faut qu'elle soit activée seulement lorsque l'adresse demandée par le CPU supérieure à \$8000. Pour ce faire, il faut vérifier le bit le plus significatif de l'adresse (A15). Le circuit de décodage doit activer la ROM seulement lorsque l'adresse commence par A15 ='1'.

Revoyons maintenant les adresses de chacun des circuits, mais en binaire.

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
\$0000	RAM1 (4k)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
\$0FFF		0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
\$1000	RAM2 (4k)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
\$1FFF		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
\$2000	RAM Interne	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
\$20FF		0	0	1	0	0	0	0	0	1	1	1	1	1	1	1
\$3000		0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
\$303F	Registres internes	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1
\$8000	ROM (32k)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
\$FFFF		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Si on se concentre plus précisément aux lignes d'adresses les plus significatives; Combien de lignes d'adresses, à partir de la plus significative (A15), sont nécessaires pour différencier tous les circuits ?

Dans notre cas, il faut prendre les quatre lignes d'adresses les plus significatives pour différencier les circuits, donc:

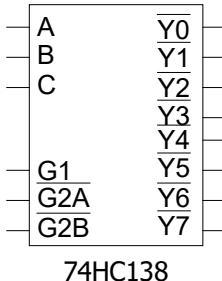
Le circuit doit répondre lorsque l'adresse commence par :				
Circuit	A15	A14	A13	A12
RAM1	0	0	0	0
RAM2	0	0	0	1
RAM interne	0	0	1	0
Registres	0	0	1	1
ROM	1	x	x	x

Réalisation d'un circuit de décodage

Il existe plusieurs façons de réaliser un circuit de décodage. Il existe aussi plusieurs circuits intégrés utiles pour faire un circuit de décodage. Nous utiliserons deux méthodes semblables; une basée sur un (des) 74HC138 et l'autre avec un circuit logique programmable.

Décodage avec un 74HC138

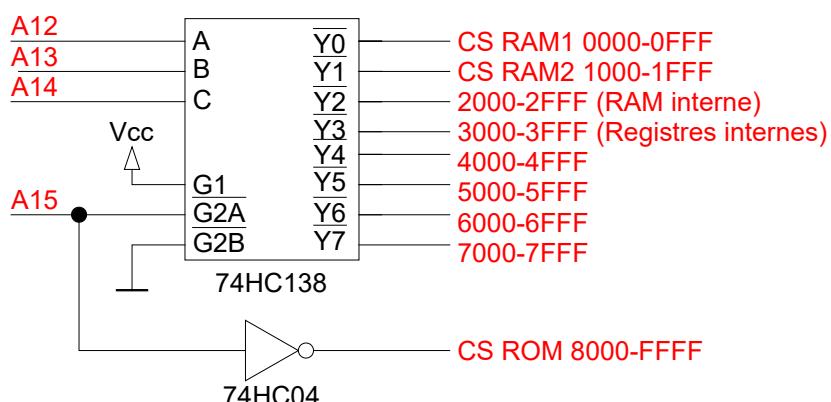
Le 74HC138 (3 to 8 line decoders/demultiplexers) est un circuit intégré dont une seule sortie sur 8 peut être active en même temps (sortie active à l'état bas). Voici son symbole logique et sa table de vérité.



74HC138

G1	G2A	G2B	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	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	1	1	1	1	1	1	1	0
0	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
x	x	1	x	x	x	1	1	1	1	1	1	1	1

Dans l'exemple de la section précédente, nous avions convenu que les quatre lignes d'adresses les plus significatives sont nécessaires pour différencier les circuits. Le 74HC138 ne comporte que 3 entrées de sélection (A, B, C). Dans ce cas, nous séparons le décodage en deux parties. Pour notre exemple, décodage de la ROM et les autres circuits.



Décodage simple

Note: L'idée de mettre les sorties du 74HC138 actives lorsqu'elles sont à zéro est plutôt utile étant donné que la plupart des circuits qui possèdent une "patte" d'activation sont actives elles aussi à zéro.

Décodage d'adresse dans un circuit logique programmable

Le décodage est une fonction combinatoire, il est donc très simple et beaucoup plus souple d'effectuer le décodage d'adresse à l'aide d'un petit circuit logique programmable, ce qui permet un changement du plan mémoire (memory map) sans changer le hardware du circuit.

On utilisera de préférence l'assignation sélective (with...select) ou l'assignation conditionnelle (when...else) pour réaliser un décodage d'adresse dans un circuit logique programmable (cf exercice décodage).

Exemple pour le décodage ci-dessus :

```
CS_RAM1 <= '0' WHEN A(15 DOWNTO 12) = "0000" ELSE '1' ;
CS_RAM2 <= '0' WHEN A(15 DOWNTO 12) = "0001" ELSE '1' ;
CS_ROM   <= NOT A(15) ;
```

Un Système on Chip (SoC)

Un système on chip (SoC) est un système dédié intégré dans une FPGA. Le cœur d'un SoC est généralement un microcontrôleur physique ou synthétisable (softcore) auquel on ajoute des circuits périphériques (IP). La différence majeure avec le système dédié est que l'on n'a jamais de bus de données bidirectionnel dans un circuit logique programmable, donc dans un SoC on a toujours deux bus de données, un pour chaque direction. Ceci implique l'ajout d'un multiplexeur permettant de sélectionner quel bus de données le microcontrôleur veut accéder en lecture.

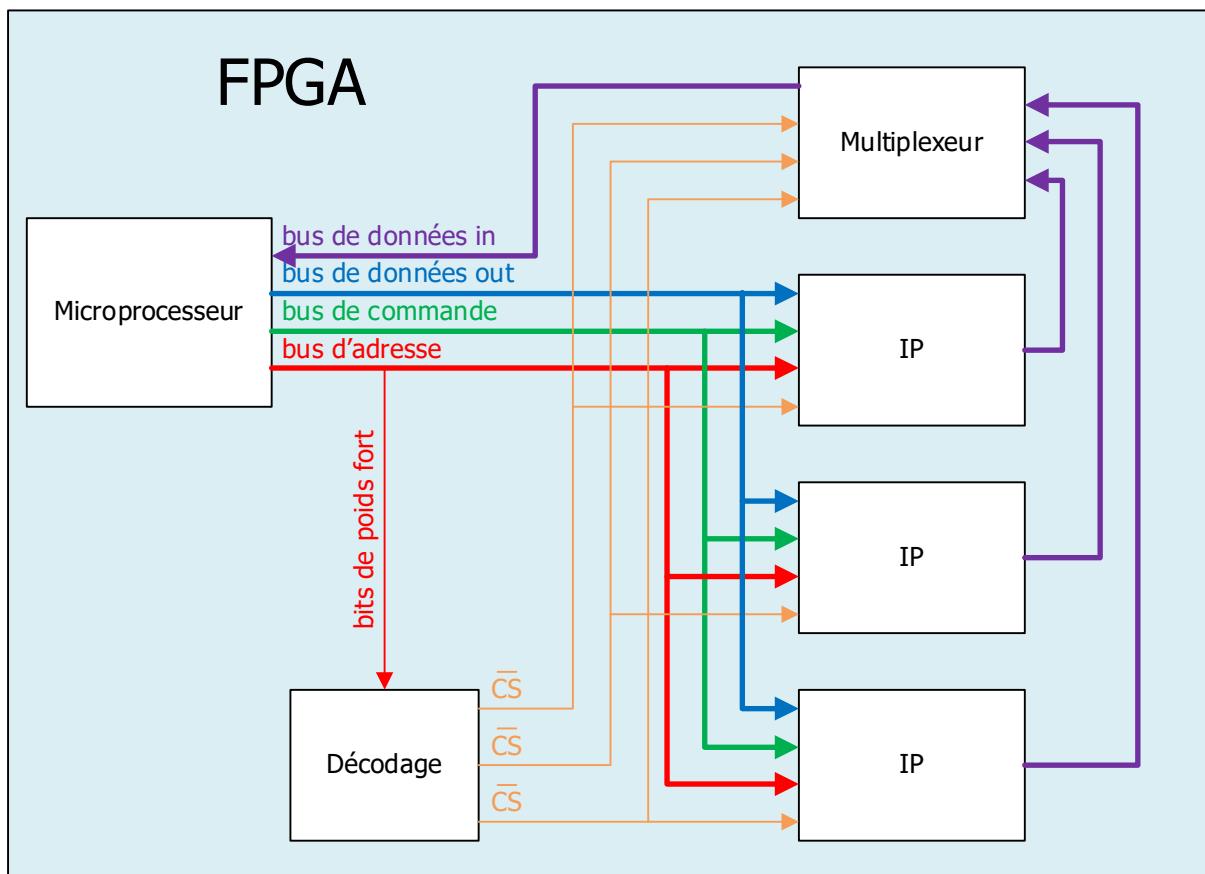


Figure 68 Schéma bloc d'un système on chip (SoC) avec décodeur d'adresses.

Mémoires sur bus microcontrôleurs 16bits ou 32bit

Un microprocesseur ou microcontrôleur de 16 ou 32 bits (voire plus) est généralement toujours capable de faire des accès de 8bits. Pour cette raison, ces circuits sont équipés d'un bus d'adresse allant de A0, A1, A2 jusqu'à An. Jusque-là rien de spécial, par contre lorsqu'il s'agit de connecter des mémoires externes sur le bus du microcontrôleur, là il s'agit de faire particulièrement attention à ce que l'on fait. Nous allons traiter ci-dessous deux cas possibles parmi beaucoup d'autres pouvant se présenter dans la pratique.

Microcontrôleur de 16 bits avec une mémoire de 16 bits

Dans ce cas, comme le microcontrôleur envoie deux octets d'un coup, il ne faut pas connecter le signal A0 du microcontrôleur sur la mémoire. On peut voir dans la Figure 64 ci-dessous comment connecter un microprocesseur de 16 bits avec une mémoire de 16 bits.

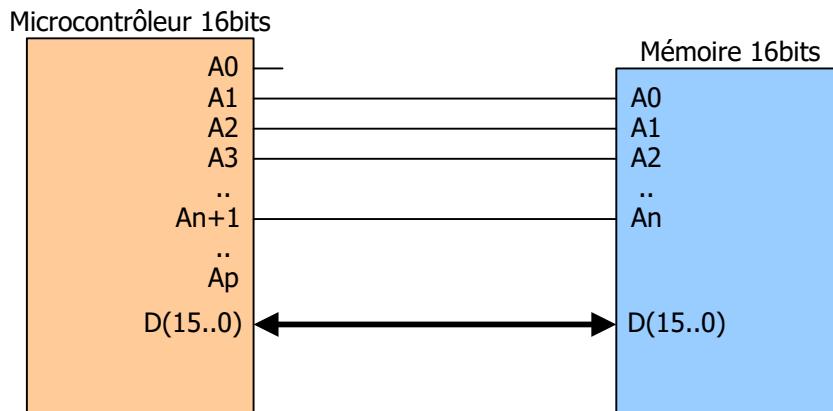


Figure 69 Connexion d'un uP de 16 bits sur une mémoire de 16 bits

Le microcontrôleur effectue des accès 16bits il envoie directement deux bytes d'un coup (le byte pair et le byte impair). On peut voir dans la figure ci-dessous, pourquoi le bit A0 ne doit pas être connecté.

Données vues du uP		Données vues de la mémoire	
..A ₃ A ₂ A ₁ A ₀		..A ₂ A ₁ A ₀	
0000	Byte 0	000	Byte 0
0001	Byte 1	001	Byte 1
0010	Byte 2	010	Byte 2
0011	Byte 3		Byte 3
0100	Byte 4		Byte 4
0101	Byte 5		Byte 5
....		

Remarque :

Certains types de circuits ont des pattes de sélection supplémentaires pour sélectionner le byte pair ou le byte impair individuellement.

Microcontrôleur de 32 bits avec deux mémoires de 16 bits

Dans ce cas, comme le microcontrôleur envoie quatre octets d'un coup, il ne faut pas connecter le signal A0 et le signal A1 du microcontrôleur sur les mémoires. On peut voir dans la Figure 65 ci-dessous comment connecter un microprocesseur de 32 bits avec deux mémoires de 16 bits.

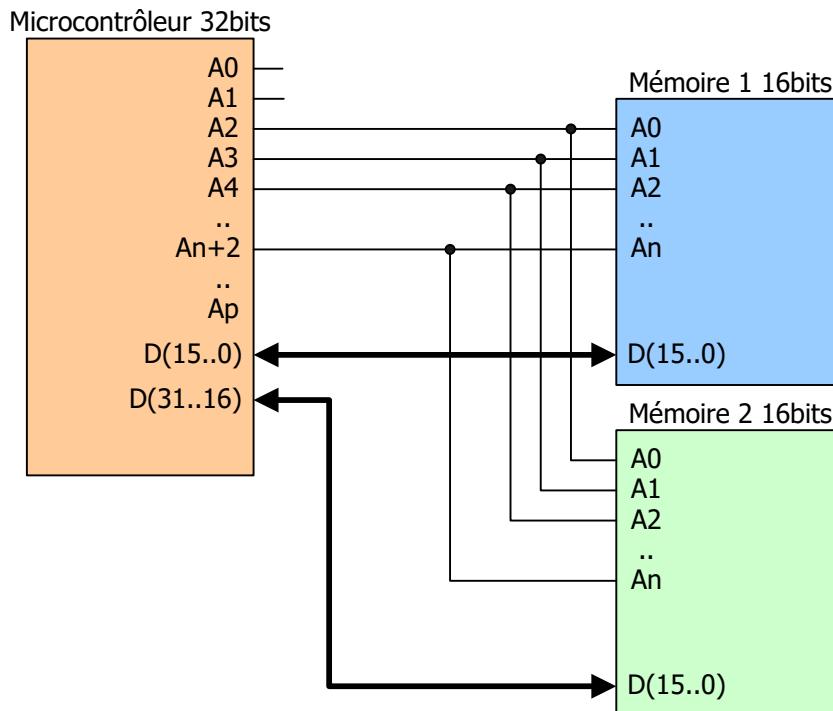


Figure 70 Connexion d'un uP de 32 bits sur deux mémoires de 16 bits

Le microcontrôleur effectue des accès 32bits il envoie directement quatre bytes d'un coup. On peut voir dans la figure ci-dessous que les deux bytes de poids faible sont stockés dans la mémoire 1 et les deux de poids fort dans la mémoire 2. Les deux mémoires ci-dessus sont équivalentes à une mémoire de 32 bits de deux fois la capacité des mémoires ci-dessus. On comprend en regardant la figure ci-dessous pourquoi il ne faut pas connecter les signaux A0 et A1 du microcontrôleur.

Données vues du uP	
..A ₄ A ₃ A ₂ A ₁ A ₀	
00000	Byte 0
00001	Byte 1
00010	Byte 2
00011	Byte 3
00100	Byte 4
00101	Byte 5
00110	Byte 6
00111	Byte 7
01000	Byte 8
01001	Byte 9
01010	Byte 10
01011	Byte 11
....	

Données vues des mémoires	
..A ₂ A ₁ A ₀	
000	Byte 0
001	Byte 1
010	Byte 2
011	Byte 3
100	Byte 4
101	Byte 5
110	Byte 6
111	Byte 7
000	Byte 8
001	Byte 9
010	Byte 10
011	Byte 11
....	

ASIC et composants logiques programmables : PAL, PLD, CPLD, FPGA

L'électronique moderne se tourne de plus en plus vers le numérique qui présente de nombreux avantages sur l'analogique : grande insensibilité aux parasites et aux dérives diverses, modularité et (re)configurabilité, facilité de stockage de l'information etc..

Les circuits numériques nécessitent par contre une architecture plus lourde et leur mode de traitement de l'information met en oeuvre plus de fonctions élémentaires que l'analogique d'où découle des temps de traitement plus long.

Aussi les fabricants de circuits intégrés numériques s'attachent-ils à fournir des circuits présentant des densités d'intégration toujours plus élevée, pour des vitesses de fonctionnement de plus en plus grandes.

D'abord réalisées avec des circuits SSI (Small Scale Integration) les fonctions logiques intégrées se sont développées avec la mise au point du transistor MOS dont la facilité d'intégration a permis la réalisation de circuits MSI (Medium Scale Integration) puis LSI (Large Scale Integration) puis VLSI (Very Large Scale Integration). Ces deux dernières générations ont vu l'avènement des microprocesseurs et microcontrôleurs.

Bien que ces derniers aient révolutionné l'électronique numérique par la possibilité de réaliser n'importe quelle fonction par programmation d'un composant générique, ils traitent l'information de manière séquentielle (du moins dans les versions classiques), ne répondant pas toujours aux exigences de rapidité.

Au début des années 70 sont apparus les premiers composants (en technologie bipolaire) entièrement configurable par programmation. La nouveauté résidait dans le fait qu'il était maintenant possible d'implanter physiquement par simple programmation, au sein du circuit, n'importe quelle fonction logique, et non plus de se contenter de faire réaliser une opération logique par un microprocesseur dont l'architecture est figée.

D'abord dédiés à des fonctions simples en combinatoire (décodage d'adresse par exemple), ces circuits laissent aujourd'hui au concepteur la possibilité d'implanter des composants aussi divers qu'un inverseur et un microprocesseur au sein d'un même boîtier ; le circuit n'est plus limité à un mode de traitement séquentielle de l'information comme avec les microprocesseurs. L'intégration des principales fonctions numériques d'une carte au sein d'un même boîtier permet de répondre à la fois aux critères de densité et de rapidité (les capacités parasites étant plus faibles, la vitesse de fonctionnement peut augmenter).

La plupart de ces circuits sont maintenant programmés à partir d'un simple ordinateur type PC directement sur la carte où ils vont être utilisés. En cas d'erreur, ils sont reprogrammables électriquement sans avoir à extraire le composant de son environnement.

De nombreuses familles de circuits sont apparues depuis les années 70 avec des noms très divers suivant les constructeurs : des circuits très voisins pouvaient être appelés différemment par deux constructeurs concurrents, pour des raisons de brevets et de stratégies commerciales. De même une certaine inertie dans l'évolution du vocabulaire a fait que certains circuits technologiquement différents ont le même nom.

Le terme même de circuit programmable est ambigu, la programmation d'une FPGA ne faisant pas appel aux mêmes opérations que celle d'un microprocesseur. Il serait plus juste de parler pour les PLD, CPLD et FPGA de circuits à architecture programmable ou encore de circuits à réseaux logiques programmables.

Ce domaine de l'électronique est aussi celui qui certainement a vu la plus forte évolution technologique ces dernières années :

- en 20 ans la densité d'intégration a été multipliée par 1000 (2 millions de transistors portes en 1995 pour 2000 millions en 2013)
- en un peu plus de 20 ans la vitesse de fonctionnement par 100 (50 MHz en 92 pour 5 GHz en 2013)
- la taille d'un transistor est passée de 0.8 µm en 93 à 0,022 µm (22nm soit moins de 100 atomes) en 2013.
- les technologies de conception ont fortement évolué, tel constructeur initiateur d'un procédé l'abandonne pour un autre, alors que le concurrent le reprend à son compte.
- la tension d'alimentation du cœur est passée de 5 V à 0.8 V diminuant ainsi la consommation.

Aussi est-il très difficile de s'y retrouver et de donner des ordres de grandeurs qui puissent être comparés. Nous tenterons dans cet exposé une clarification des choses dont la volonté de simplification pourra être facilement prise en défaut.

Parallèlement à ces circuits, on trouvera les ASIC (Application Specific Integrated Circuits) qui sont des composants où le concepteur intervient au niveau du dessin de la pastille de silicium en fournissant des masques à un fondeur. On ne peut plus franchement parler de circuits programmables. Les temps de développement long ne justifient l'utilisation que pour des grandes séries.

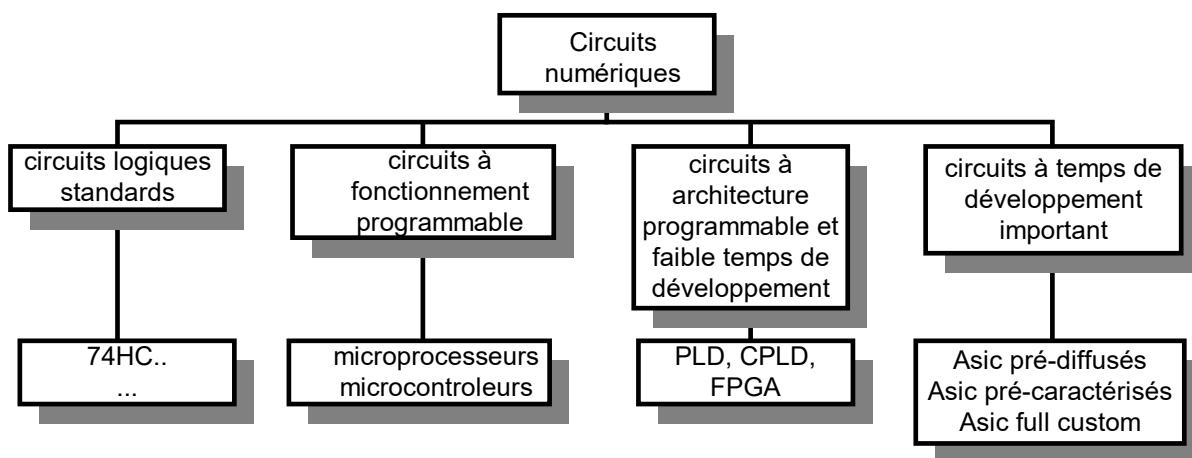
Les circuits prédiffusés peuvent être développés aussi rapidement qu'une FPGA.

Le temps de fabrication est plus long (3 semaines pour le MD100) pour un temps de développement de 8 semaines (selon le circuit et sans la fabrication).

Ils sont surtout utilisés pour la faible consommation (quelques uA voire même nA) et pour les circuits analogiques/numériques (oscillateurs, sources de courants, comparateurs, etc...).

Les PLD, CPLD et FPGA sont parfois considérés comme des ASIC par certains auteurs.

Le tableau ci-après tente une classification possible des circuits numériques :



Nous nous intéresserons surtout aux circuits à architecture programmable à faible temps de développement. Le principe de base des circuits nous intéressant ici consiste à réaliser des connexions logiques programmables entre des structures présentant des fonctions de bases. Le premier problème va donc être d'établir ou non suivant la volonté de l'utilisateur, un contact électrique entre deux points. Aussi, nous intéresserons nous, avant de passer aux circuits proprement dits et à leur programmation, à ces technologies d'interconnexion.

Mais avant toutes choses, rappelons comment coder une fonction logique.

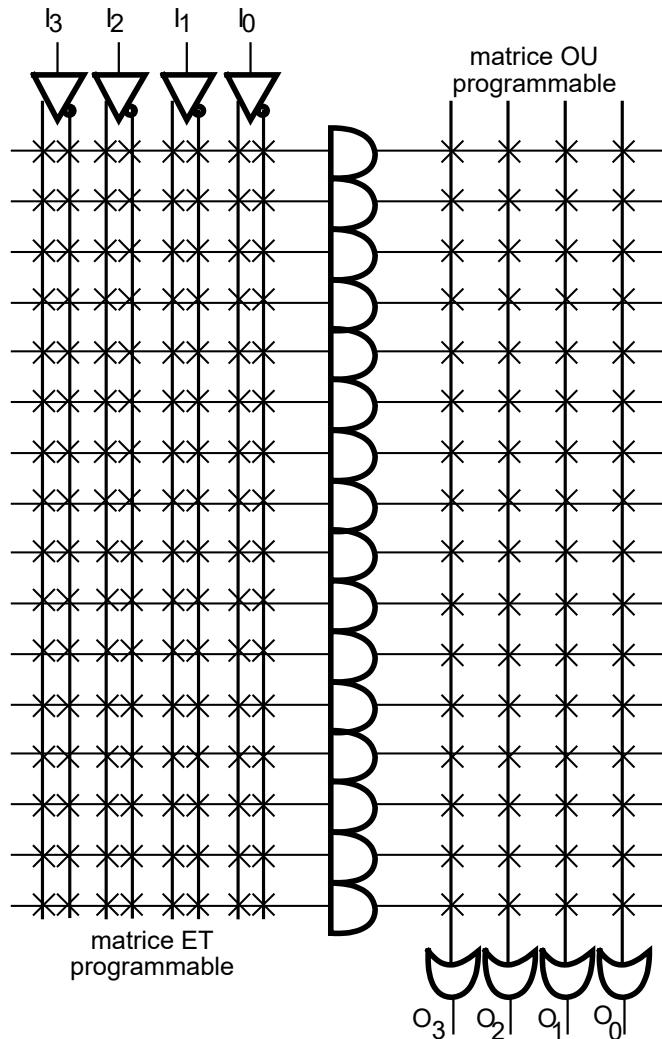
Codage d'une fonction logique

La base d'une fonction logique, est toujours une fonction combinatoire. Pour obtenir une fonction séquentielle, il suffira ensuite de réinjecter les sorties sur les entrées, ce qui donnera alors un système asynchrone. Si on souhaite un système synchrone, on intercalera avant les sorties, une série de bascules à front, dont l'horloge commune synchronisera toutes les données et évitera bien des aléas de fonctionnement.

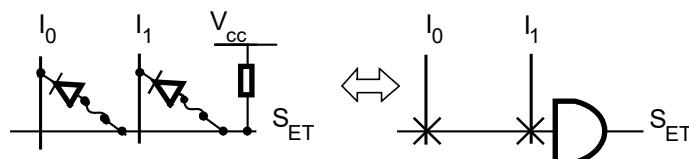
Pour coder une fonction combinatoire, trois solutions sont classiquement utilisées.

Sommes de produits, produits de somme et matrice PLA (Programmable Logic Array)

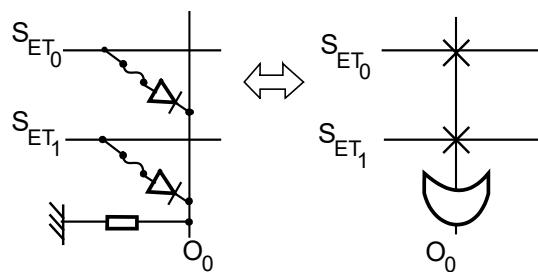
N'importe quelle fonction peut être codée par une somme de produit, par un produit de somme ou un mélange des deux. On peut immédiatement en déduire une structure de circuits, appelé matrice PLA (Programmable Logic Array). La figure suivante représente une matrice PLA à 4 entrées et 4 sorties :



Chacune des 4 entrées et son complémentaire arrive sur une des 16 portes ET à $2 \times 4 = 8$ entrées. Afin de simplifier la représentation, les 8 lignes ont été représentées par une seule, chaque croix représentant une connexion programmable (un fusible par exemple). La figure suivante propose un principe de réalisation des fonctions de la matrice ET ; la mise au niveau logique 0 (NL0) d'une des entrées I_x impose un NL0 en sortie :



et de la matrice OU, sur laquelle une entrée au NL1 impose un NL1 en sortie :



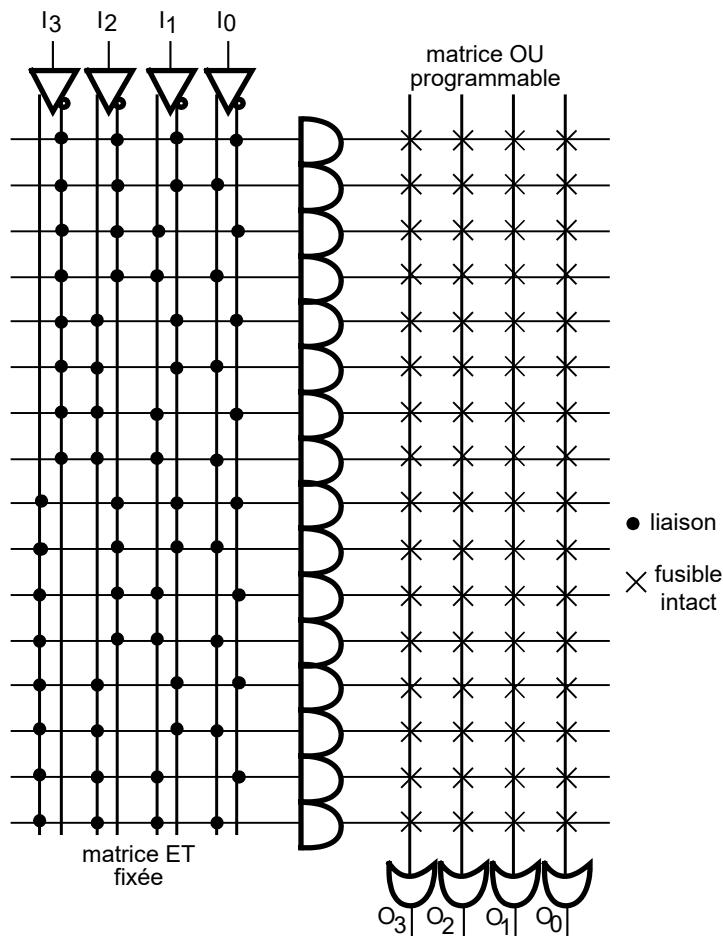
Ce type de structure est utilisé dans certains circuits ASIC (Application Specific Integrated Circuit) et demande une densité d'intégration importante : en effet pour n variables en entrées, il faut 2^n fonctions ET à

2n entrées et au moins un OU à 2ⁿ entrées (il y a en effet 2ⁿ combinaisons possibles, chaque combinaison dépendant de l'entrée et de son complémentaire).

La plupart des applications n'exigent pas une telle complexité et on peut se contenter d'une matrice ET programmable et d'une matrice OU figée. De même, il est peu probable d'utiliser tous les termes produits et on peut alors limiter le nombre d'entrées de la fonction OU. C'est le principe utilisé par les circuits programmable, appelés au début PAL (Programmable Array Logic), mais plus communément désigné aujourd'hui sous le terme PLD (Programmable Logic Device).

Mémoires (Look Up Table)

Une fonction combinatoire associe à chacune de ces combinaisons d'entrée une valeur en sortie décrite par sa table de vérité. C'est le principe de la mémoire où pour chaque adresse en entrée, on associe une valeur en sortie, sur un ou plusieurs bits. La structure physique des mémoires fait appelle à une matrice PLA dont la matrice ET est figée et sert de décodeur d'adresse et dont la matrice OU est programmée en fonction de la sortie désirée

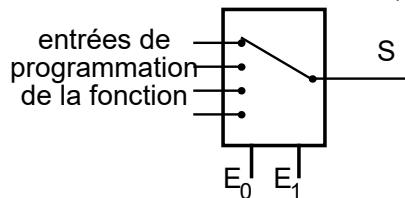


Lorsqu'une adresse est présentée, par exemple $I_1 I_2 I_3 I_4 = 1111$, la porte ET concernée passe au NL1 (celle du bas dans l'exemple) et suivant les fusibles laissés intacts sur la matrice OU, on a un mot différent en $O_0 O_1 O_2 O_3$ (0000 si tous les fusibles sont "grillés" par exemple).

Ce principe utilisé pour les mémoires, l'est aussi dans les CPLD (Complex Programmable Logic Device), mais surtout dans les FPGA (Field Programmable Gate Array) sous le nom de LUT (Look Up Table).

Multiplexeur

Le multiplexeur permet également de coder une fonction combinatoire, comme le montre la figure suivante :



A chaque valeur des entrées E_0 et E_1 est associé un niveau logique défini dans la table de vérité pour la sortie S . Ce niveau logique est imposé sur l'entrée de programmation correspondante. Ce principe est utilisé dans les FPGA.

Technologie d'interconnexions

Comme nous venons de le voir, l'un des éléments clé des circuits étudié est la connexion programmable.

Du choix d'une technologie dépendra essentiellement :

- la densité d'intégration
- la rapidité de fonctionnement une fois le composant programmé, fonction de la résistance à l'état passant et des capacités parasites
- la facilité de mise en oeuvre (programmation sur site, reprogrammation, etc)
- la possibilité de maintien de l'information.

Passons en revue quelques technologies classiquement utilisées et leurs caractéristiques. Ces technologies sont ou pourraient être utilisées pour la réalisation de mémoires. Il faut cependant garder à l'esprit qu'hormis quelques cas particuliers (circuit reprogrammés en cours d'utilisation), le temps d'écriture reste secondaire, le circuit étant habituellement programmé une fois pour toutes avant utilisation.

Connexions programmable une seule fois (OTP : One Time Programming)

Cellules à fusible

Ce sont les premières à avoir été utilisées et elles ont aujourd'hui disparu au profit de technologies plus performantes. Leur principe consistait à détruire un fusible conducteur par passage d'un courant fourni par une tension supérieure à l'alimentation (12 à 25 V).

Cellules à antifusible

En appliquant une tension importante (16 V pendant 1 ms) à un isolant entre deux zones de semi-conducteur fortement dopées, ce dernier diffuse dans l'isolant et le rend conducteur. Chaque cellule occupe environ $1,8 \mu\text{m}^2$ ($700 \mu\text{m}^2$ pour un fusible) ; cette technologie très en vogue permet une haute densité d'intégration.

Hormis la non reprogrammabilité, c'est la meilleure technologie (vitesse et surtout densité d'intégration).

Cellules reprogrammables

Cellule à transistor MOS à grille flottante et EEPROM (Erasable Programmable Read Only Mémoire)

L'apparition du transistor MOS à grille flottante a permis de rendre le composant bloqué ou passant sans application permanente d'une tension de commande. Le principe consiste à piéger ou non (à l'aide d'une tension supérieure à la tension habituelle d'alimentation) des électrons dans la grille. L'extraction éventuelle des électrons piégés permet le retour à l'état initial. Plusieurs technologies EEPROM sont en concurrence :

UV-EPRON

Les connexions sont réinitialisable par une exposition à un rayonnement ultraviolet d'une vingtaine de minutes, une fenêtre étant prévue sur le composant. L'effacement est de mise en oeuvre lourde, n'est pas sélectif et ne peut se faire sur site. Ce principe n'est pas utilisé pour les circuits qui nous intéressent.

EEPROM (Electrically EPROM)

L'effacement et la programmation se font cette fois électriquement avec une tension de 12 V et peuvent se faire de manière sélective (la reprogrammation de tout le composant n'est pas nécessaire).

Une cellule demande toutefois 5 transistors pour sa réalisation, ce qui conduit à une surface importante (75 à 100 μm^2 en CMOS 0,6 μm) et réduit la densité d'intégration possible.

D'autre part le nombre de cycles de programmation est limité à un nombre de 100 (en CMOS 0,6 μm) à 10 000 (en CMOS 0,8 μm) à cause de la dégradation des isolants.

La programmation ou l'effacement d'une cellule dure quelques millisecondes.

Flash EEPROM

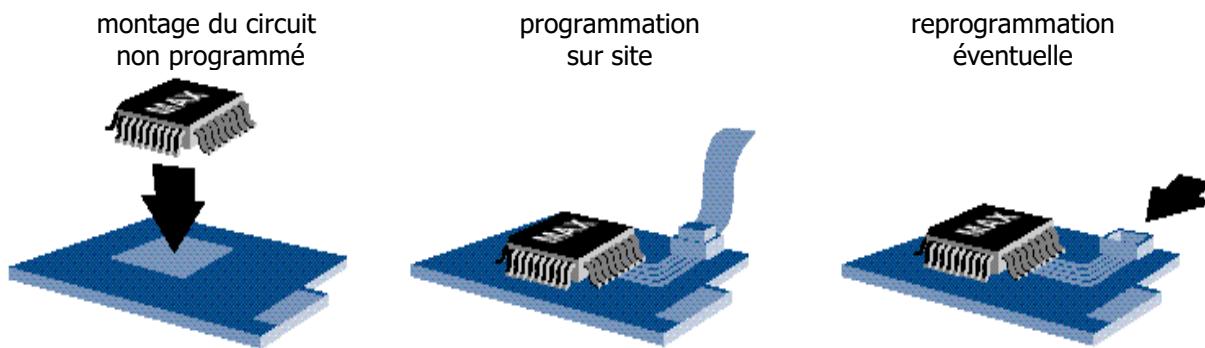
L'utilisation de deux transistors par cellule uniquement (5 pour l'EEPROM) et une structure verticale permettent une densité d'intégration importante (25 μm^2 par cellules en CMOS 0,6 μm) trois à quatre fois plus importante que l'EEPROM, mais quand même 10 fois moins que la technologie à antifusible.

Le nombre de cycle d'écriture (10^5 à 10^6) est également plus grand que pour l'EEPROM car l'épaisseur de l'isolant est plus importante.

Par contre, la simplicité de la cellule élémentaire n'autorise pas une reprogrammation sélective (éventuellement par secteur), ce qui n'est pas gênant pour le type de circuits qui nous intéresse..

La tension de programmation et d'effacement est de 12 V, avec un temps de programmation de quelques dizaines de μs pour un temps d'effacement de quelques millisecondes.

Un des inconvénients des cellules flash et EEPROM de nécessiter une alimentation supplémentaire pour la programmation et l'effacement est pallié par les constructeurs en intégrant dans le circuit un système à pompe de charge fournissant cette alimentation. Le composant peut alors être programmé directement sur la carte où il est utilisé. On parle alors de composants ISP : In Situ Programmation ou encore suivant les sources, In System Programming.



Cellules SRAM à transistors MOS classique

Ce principe est classiquement choisi pour les FPGA.

Le fait d'utiliser une mémoire de type RAM (donc volatile) impose la recharge de la configuration à chaque mise sous tension : une PROM série mémorise généralement les données.

Ce qui peut paraître un inconvénient devient un avantage si on considère l'aspect évolutif du système qui peut s'adapter à un environnement extérieur changeant et modifier sa configuration en fonction des besoins. On pourra d'autre part facilement intégrer de la mémoire RAM dans le circuit.

Le choix d'une cellule SRAM (Static Read Only Memory) à 6 transistors permet de bénéficier d'un accès sélectif et rapide (quelques ns) en cours d'utilisation.

La taille d'une cellule n'est que deux fois plus forte (50 μm^2 par cellule) qu'avec une flash EEPROM.

Cette technologie, utilisée pour les autres circuits VLSI (contrairement aux EEPROM et Flash EPROM et leurs transistors à grille flottante) permet de bénéficier directement des progrès importants réalisés dans ce domaine.

Comme nous venons de la voir, la programmation sur site de ces circuits est une nécessité absolue.

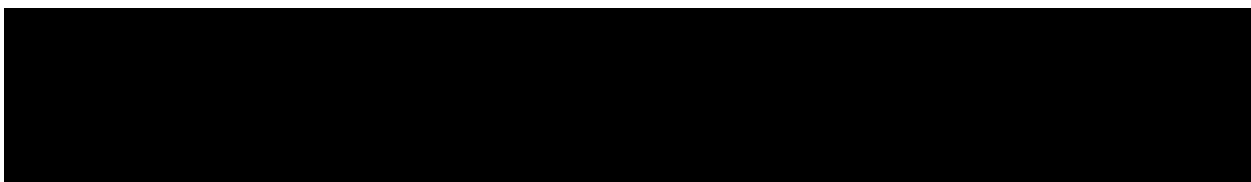
Architectures utilisées

PLD (Programmable Logic Device)

Comme nous l'avons vu, d'abord appelés PAL lors de sa sortie, ce circuit utilise le principe de la matrice PLA à réseau ET programmable. Bien que pas très anciens pour les dernières générations, les PLD ne sont presque plus utilisés pour une nouvelle conception. L'un de leur avantage qu'était la rapidité a disparu, les efforts de recherche des constructeurs portant plutôt sur les circuits à plus forte densité d'intégration que sont les CPLD et les FPGA.

Le fait que les PLD soient à la base de la conception des CPLD, très en vogue aujourd'hui, justifie cependant leur étude.

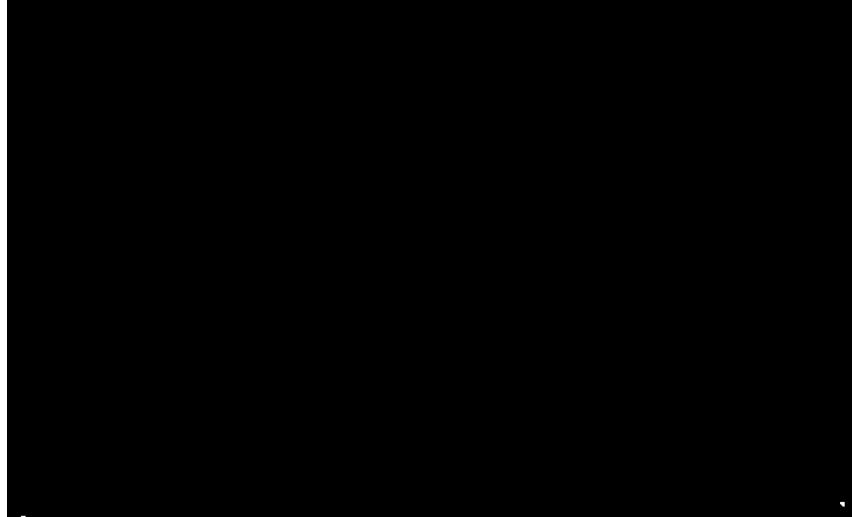
Initialement bipolaire, les cellules de connexions sont aujourd'hui réalisées en technologie MOS à grille flottante. La structure de base comprend un circuit PLA dont seule la matrice ET est programmable.



La partie nommée OLMC (Output Logic MacroCell, dénomination Lattice) sur la figure peut être :

- combinatoire, une simple connexion relie alors la sortie du OU à l'entrée du buffer de sortie, dont la sortie est réinjectée sur le réseau programmable ;
- séquentielle, le bloc OLMC étant alors une simple bascule D ;
- versatile, il est alors possible par programmation de choisir entre les deux configurations précédentes.

Les PLD de dernière génération utilisent des OLMC versatiles, dont on donne ci-après la structure :



Le multiplexeur 4 vers 1 permet de mettre en circuit ou non la bascule D, en inversant ou pas les signaux. Le multiplexeur 2 vers 1 permet de réinjecter soit la sortie, soit l'entrée du buffer de sortie vers le réseau programmable.

Désignation

Elle est de la forme PAL EE T SS où EE représente le nombre d'entrées, SS le nombre de sorties et T le type du PAL : exemple : PAL 22V10.

Pour les dernières générations, on trouvera plutôt la référence de type GAL EE T SS, (GAL pour Generic Array Logic) pour désigner un circuit à transistor MOS à grille flottante, donc reprogrammable électriquement.

Programmation

Les étapes de programmation sont assistées par ordinateur et présentent la chronologie suivante :

- description de la fonction souhaitée par entrée schématique ou syntaxique ; dans ce dernier cas on utilise un langage approprié appelé HDL (Hardware Description Language) comme le langage ABEL ou éventuellement VHDL (Very High speed integrated circuit HDL). On en profite pour définir des vecteurs de test de la fonction réalisée.
- simulation logique puis temporelle de la fonction réalisée et éventuellement retour à l'étape précédente.
- compilation et génération d'un fichier de programmation (fichier au standard JEDEC).
- programmation et test physique du composant.

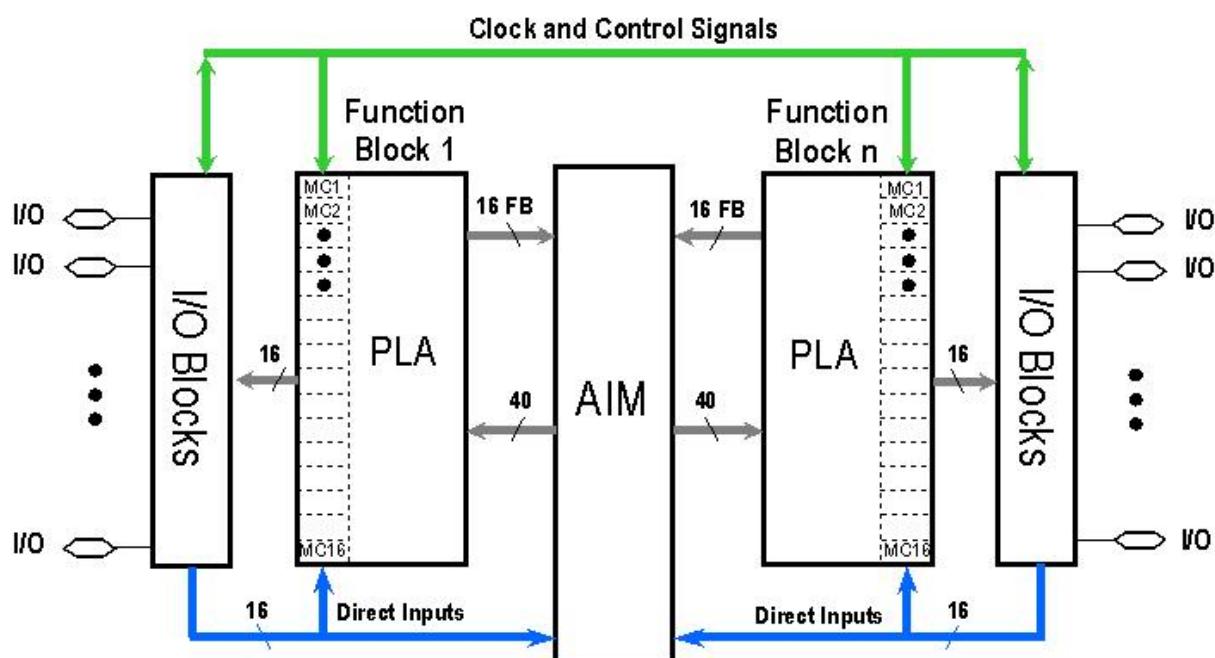
CPLD (Complex Programmable Logic Device)

La nécessité de placer de plus en plus de fonctions dans un même circuit a conduit tout naturellement à intégrer plusieurs PLD (blocs logiques) sur une même pastille, reliés entre eux par une matrice centrale. Nous avons comme exemple la famille CoolRunner-II de Xilinx, mais la même structure est utilisée chez d'autres fabricants.

Sur la figure suivante chaque bloc PLA (Programmable Logic Array) lui-même composé de 16 Macrocellules est l'équivalent d'un PLD à 16 OLMC. Ils sont reliés entre eux par une matrice d'interconnexion (AIM pour Advanced Interconnect Matrix).

Un seul point de connexion relie entre eux les blocs logiques. Les temps de propagation d'un bloc à l'autre sont donc constants et prédictibles.

La phase de placement des différentes fonctions au sein des Macrocellules n'est donc pas critique sur un CPLD, l'outil de synthèse regroupant au maximum les entrées sorties utilisant des ressources communes.

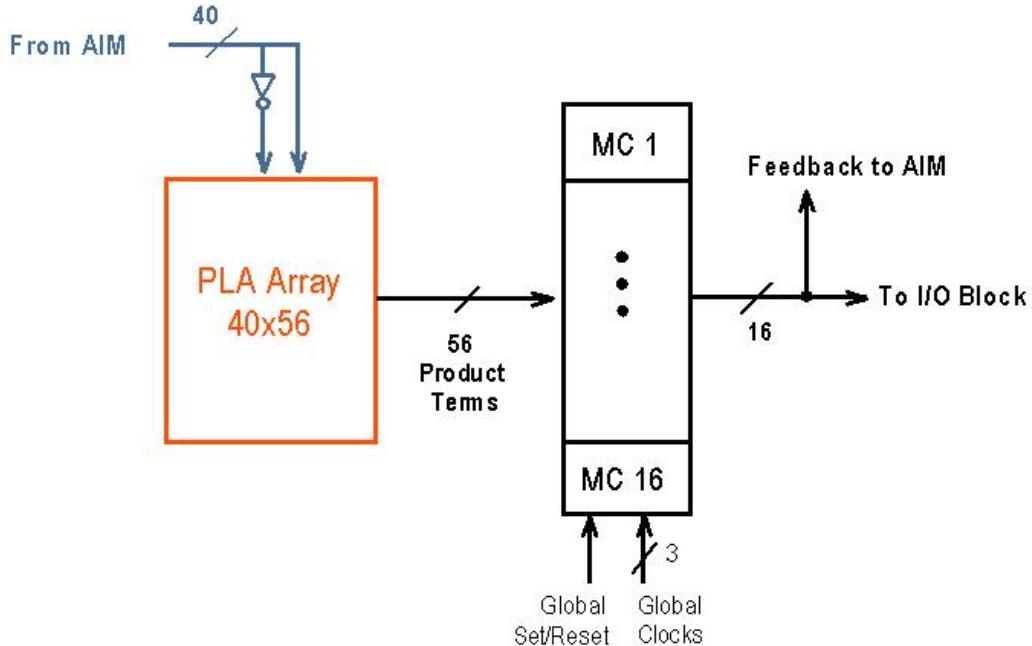


L'établissement des liaisons (routage) entre les différentes Macrocellules est encore moins critique : un seul point de connexion -cause du retard- relie les PLA entre eux. Le temps de propagation des signaux est parfaitement prédictible avant que le routage ne soit fait. Ce dernier n'influence donc pas les performances du circuit programmé.

De plus des liaisons spécifiques aux signaux clock et reset sont amenées directement sur chaque Macrocellules.

Dans l'outil de synthèse, la partie s'occupant du placement et du routage est appelée le "Fitter" (to fit : placer, garnir). La technologie de connexion utilisée est généralement l'EEPROM (proche de celle des PLD) ou EEPROM flash.

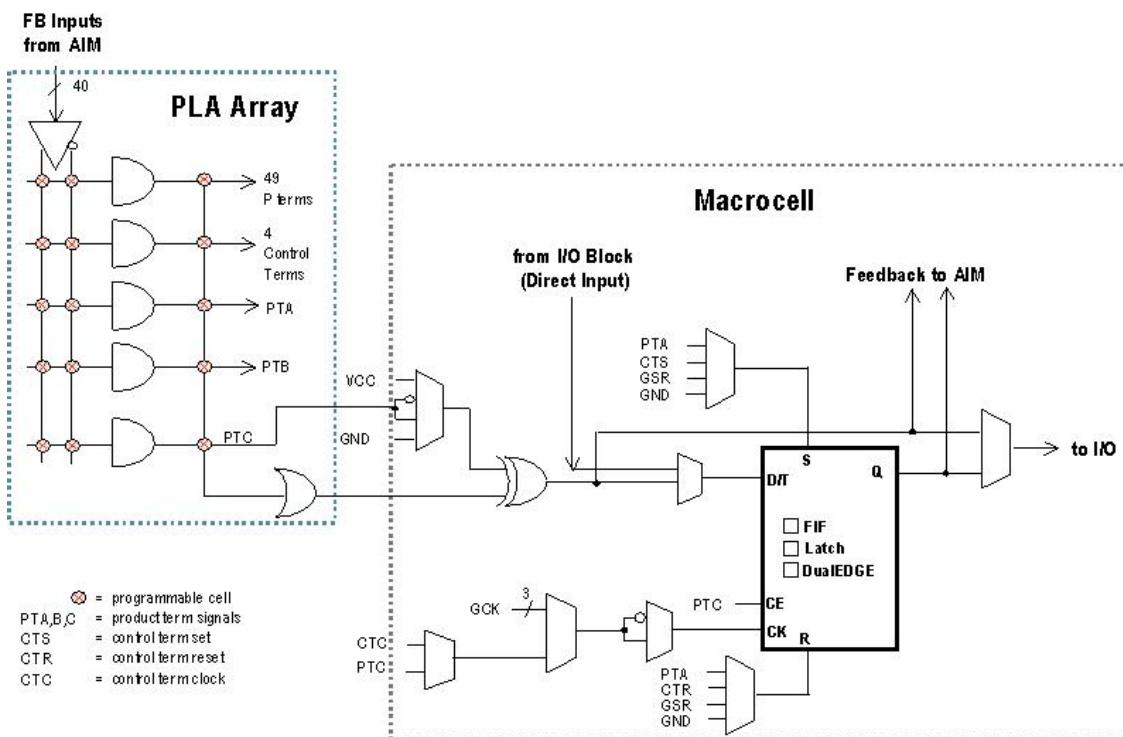
Dans la figure ci-dessous, on peut voir plus en détail la structure d'un Function Block :



Dans un Function bloc on a une partie PLA (Programmable Logic Array) permettant de réaliser des produits logiques combinatoires. Ces produits entrent ensuite dans les Macrocellules.

On remarque les signaux Global Set/Reset et Globals Clocks, qui permettent de connecter directement les signaux reset et clock sur les macrocellules, sans utiliser des ressources de routage.

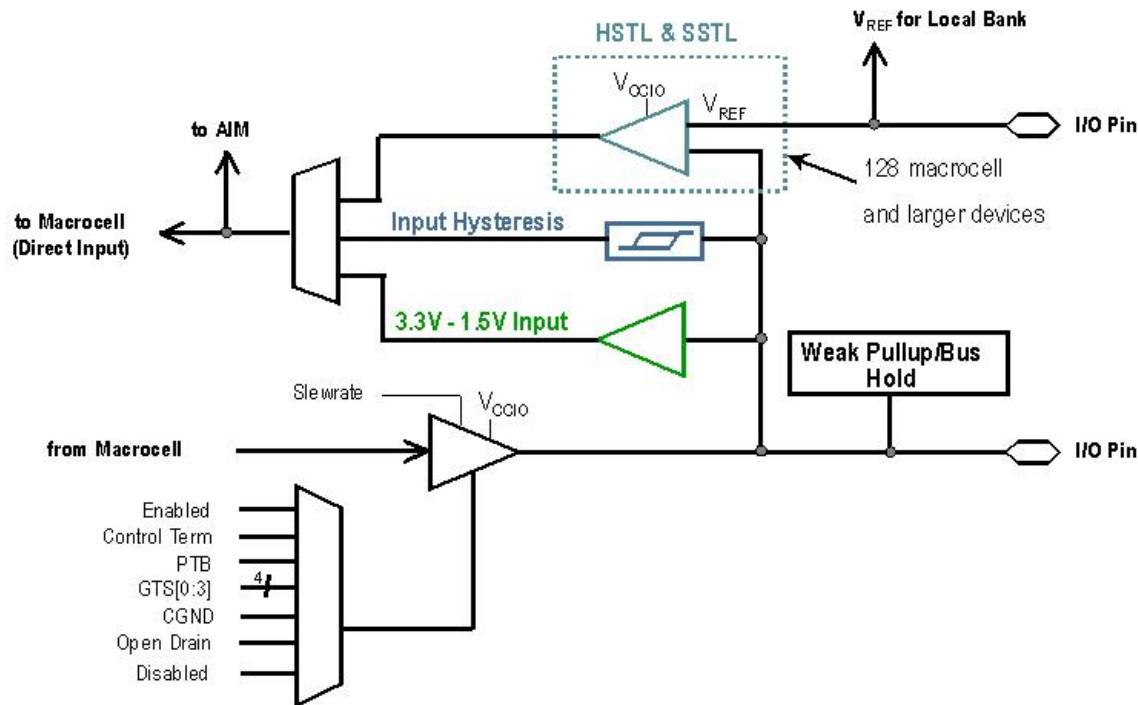
Dans la figure ci-dessous, on peut voir plus en détail la structure du PLA et d'une Macrocellule :



Pour compléter l'analyse d'une CPLD, il faut encore parler des blocs d'entrée/sortie.

En effet sur chaque patte physique d'un CPLD on trouve un bloc d'entrée/sortie correspondant, qui est principalement composé des fonctions suivantes :

- un buffer d'entrée avec hystérèse
- un buffer de sortie tri-state, permettant le réglage du slew-rate
- un résistance pull-up programmable
- un multiplexeur pour la commande du buffer tris-state



Dans le tableau ci-dessous, on peut voir les différents circuits de la famille CoolRunner-II :

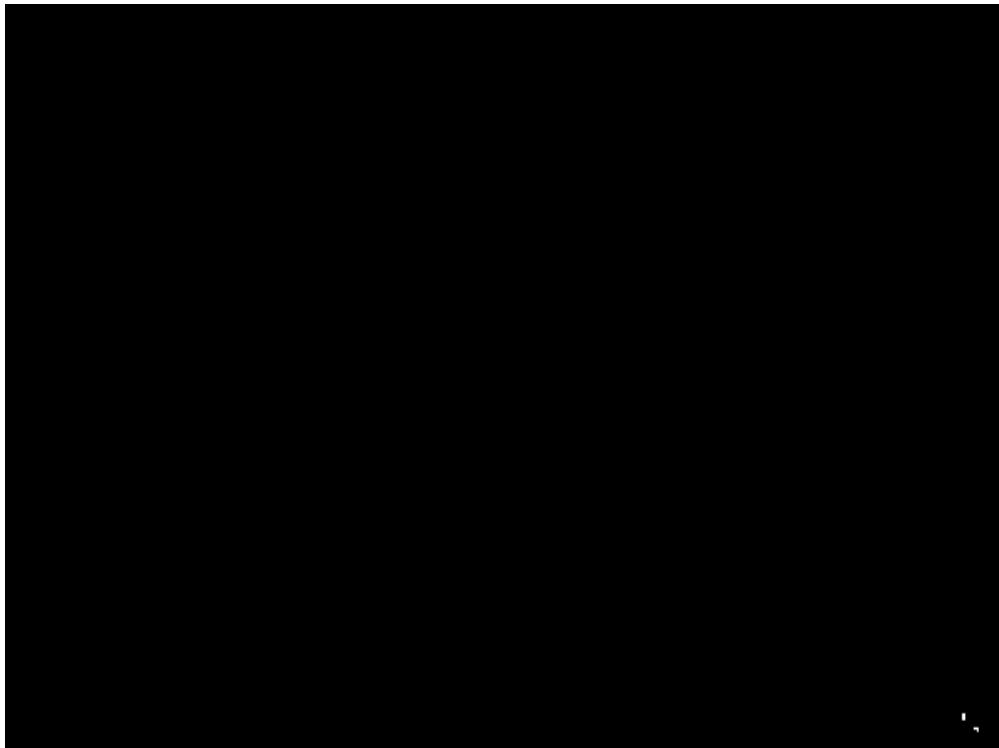
Feature/Product	XC2C 32	XC2C 64	XC2C 128	XC2C 256	XC2C 384	XC2C 512
Macrocells	32	64	128	256	384	512
T _{pd} (ns)	3.0	4.0	4.5	5.0	6.0	6.0
f _{SYS} (MHz)	385	270	263	238	217	217
I/O Banks	1	1	2	2	4	4
Packages	User I/O					
VQ44	33	33				
PC44	33	33				
CP56	33	45				
VQ100		64	80	80		
CP132			100	106		
TQ144			100	118	118	
PQ208				173	173	173
FT256				184	212	212
FG324					240	270

FPGA (Field Programmable Gate Array)

Les blocs logiques sont beaucoup plus nombreux et plus simples que pour les CPLD, mais cette fois les interconnexions entre les blocs logiques ne sont pas centralisées. Les FPGA sont composées de trois éléments principaux :

- les IOB (Inputs Outputs Blocks), qui sont les blocs d'entrée/sortie
- les CLB (Configurable Logic Blocks), qui sont les blocs contenant la logique programmable
- les blocs d'interconnexion programmables

Les IOB entourent la FPGA, tandis que les CLB forment un array de $N \times M$ CLB. Les CLB sont entourés de chaque côté par des lignes d'interconnexion, qui permettent de relier les IOB au CLB, ainsi que les CLB entre eux.



Ces composants permettent une forte densité d'intégration. La petite taille des blocs logiques autorise une meilleure utilisation des ressources du composant (au prix d'un routage délicat)

Il devient alors possible d'implanter dans le circuit des fonctions aussi complexes qu'un microcontrôleur. Ces fonctions sont fournies sous forme de programme (description VHDL du composant) par le constructeur du composant et appelées "megafonction" ou "core". Le terme générique classiquement utilisé pour les désigner est "propriété intellectuelle" ou IP (Intellectual Property).

Les FPGA utilisent généralement les technologies SRAM ou antifusible, selon les fabricants, Xilinx utilisant la technologie SRAM et ACTEL, ALTERA de préférence la technologie antifusible.

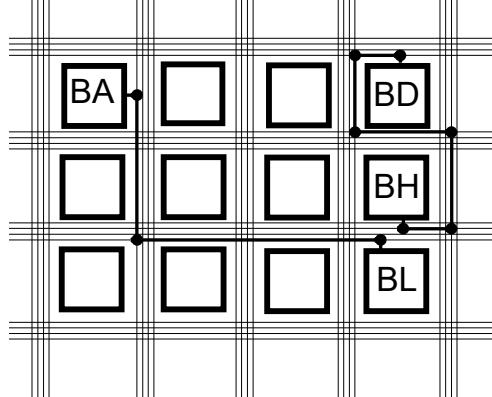
L'utilisation optimale des potentialités d'intégration d'un FPGA conduira à un routage délicat et pénalisant en termes de vitesse.

Il convient de noter que ces retards sont dus à l'interaction de la résistance de la connexion et de la capacité parasite ; cela n'a rien à voir avec un retard dû à la propagation d'un signal sur une ligne tel qu'on le voit en haute fréquence.

Le passage d'un bloc logique à un autre se fera par un nombre de point de connexion (responsables des temps de propagation) fonction de la position relative des deux blocs logiques et de l'état "d'encombrement" de la matrice. Ces délais ne sont donc pas prédictibles (contrairement aux CPLD) avant le placement/routage.

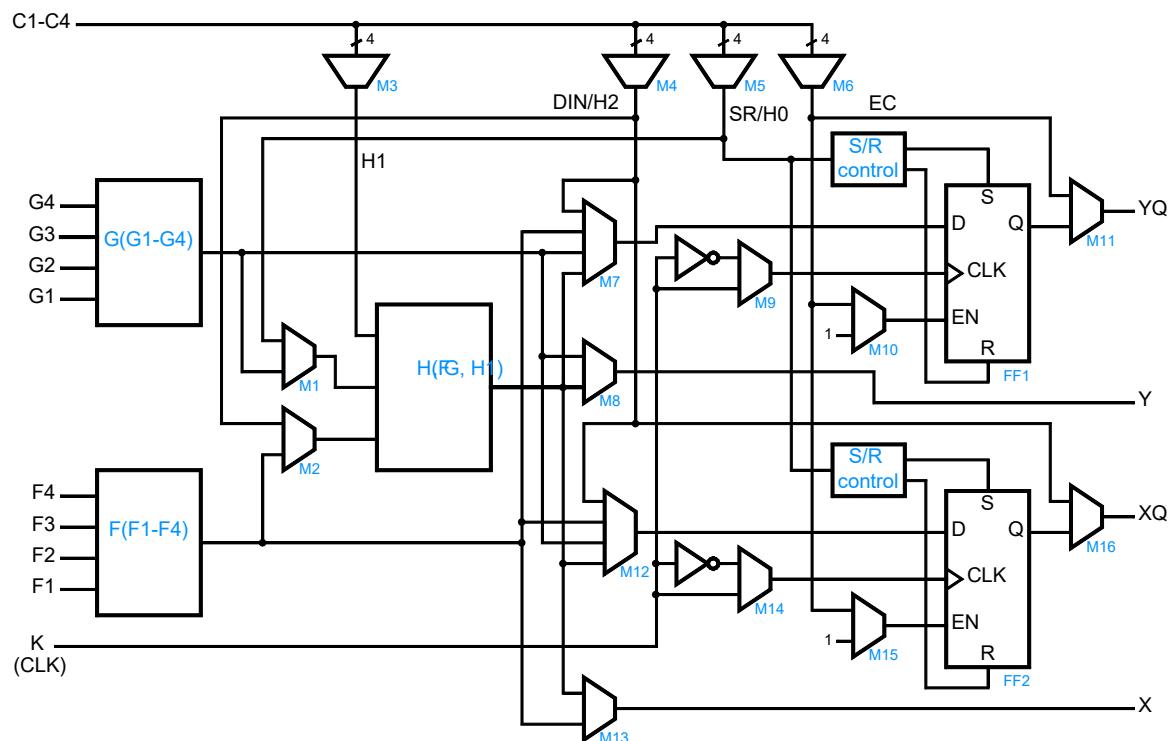
De la phase de placement des blocs logique et de routage des connexions dépendront donc beaucoup les performances du circuit en termes de vitesse. Sur la figure suivante, pour illustrer le phénomène, on peut voir

- une liaison entre deux blocs logiques (BA et BL) éloignés, mais passant par peu de points de connexion, donc introduisant un faible retard.
- une liaison entre deux blocs proches (BD et BH) mais passant par de nombreux points de connexion, donc introduisant un retard important.



Structure FPGA famille XC4000 de Xilinx

On peut voir ci-dessous la structure d'un CLB de la famille XC4000 de Xilinx, qui est déjà obsolète, mais qui permet de bien la structure d'un CLB.

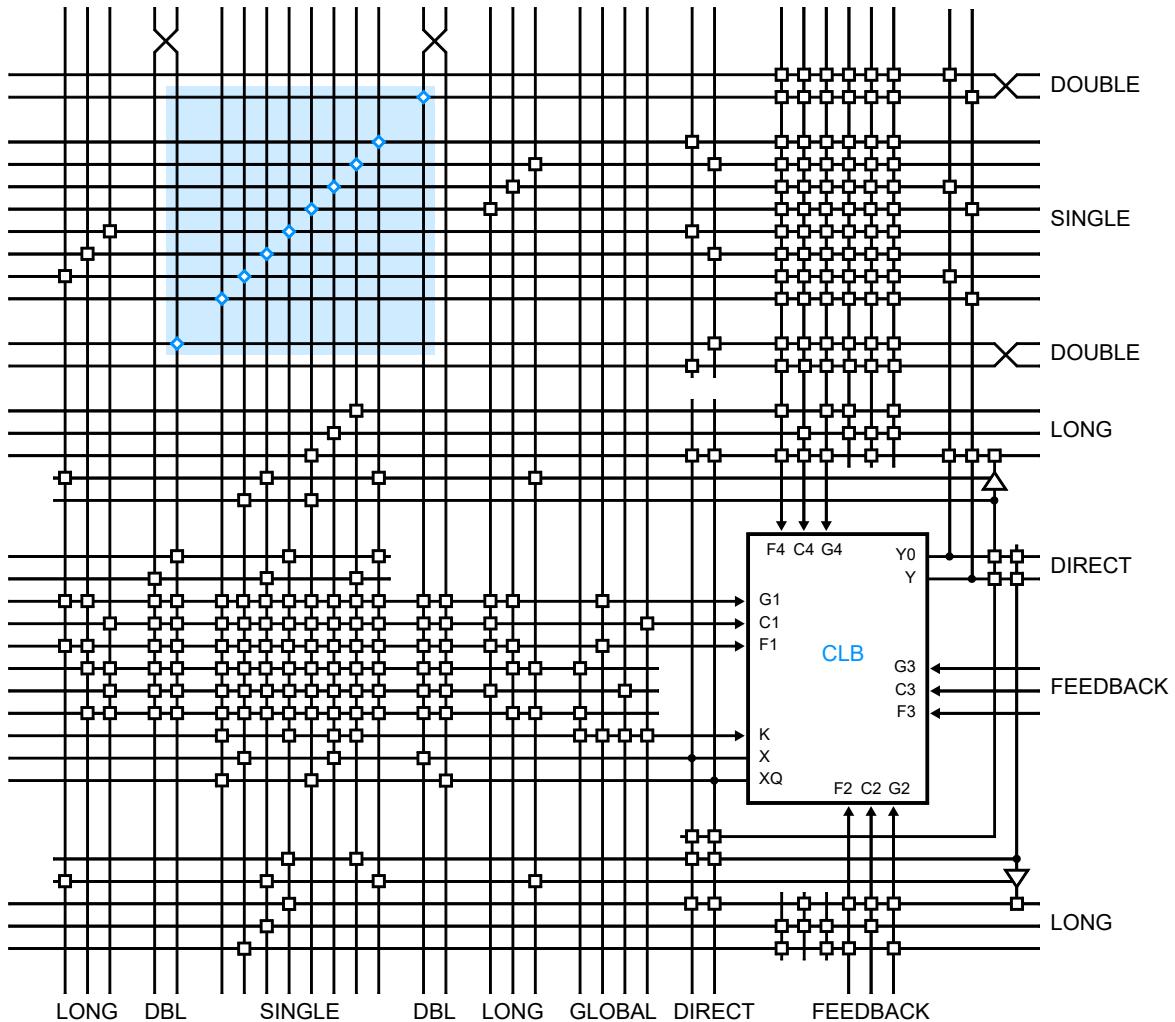


Le CLB est composé de trois parties principales qui sont :

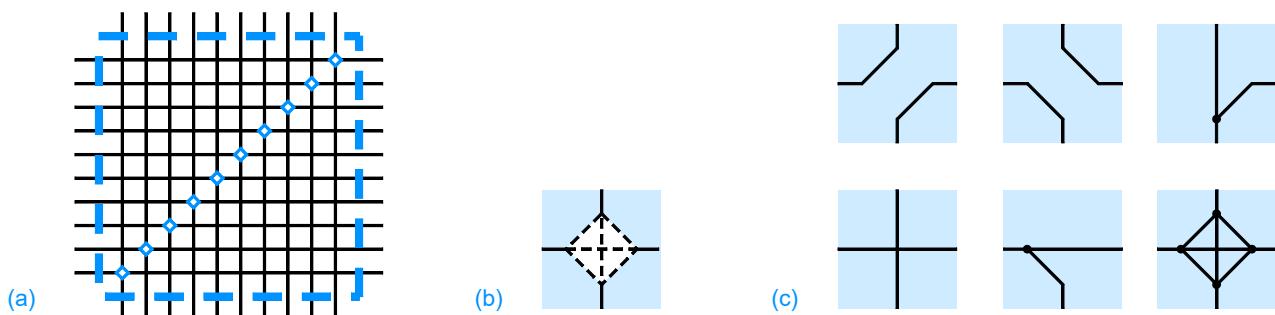
- les blocs LUT G, F et H qui permettent de réaliser des opérations logiques combinatoire
- les deux bascules D FF1 et FF2
- 16 multiplexeurs M1 à M16 programmables permettant d'aiguiller les signaux à l'intérieur du CLB

Comme expliqué auparavant, ces CLB sont entourés de lignes d'interconnexions permettant de relier les CLB entre eux, afin de réaliser de grandes fonctions logiques. Chacun des points est une connexion programmable, donc à chaque point, correspond un bit dans le fichier de configuration de la FPGA.

On peut voir dans la figure ci-dessous le principe d'interconnexion :

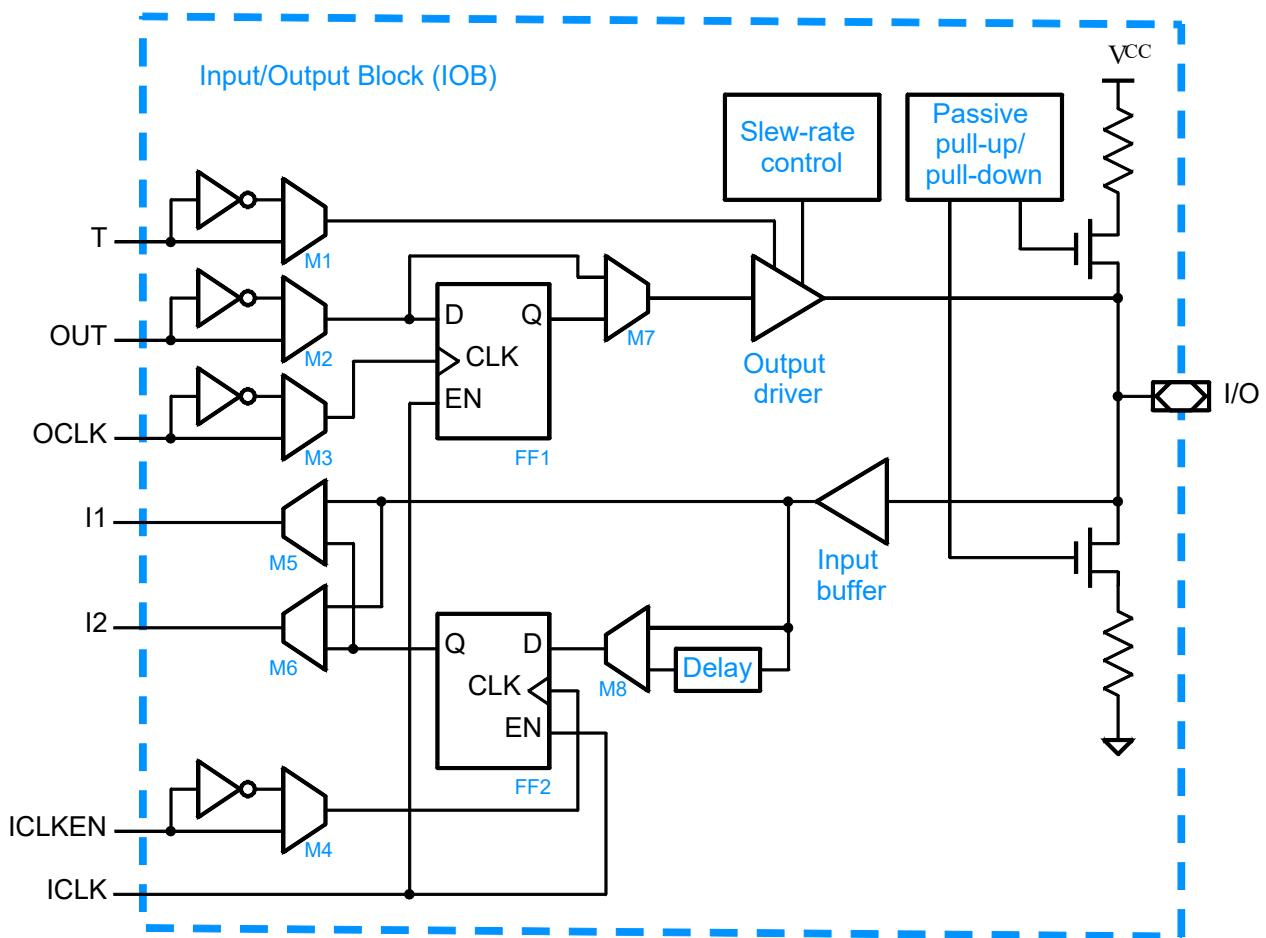


Le rectangle bleu est un élément appelé PSM (Programmable Switch Matrix), permettant plusieurs connexions entre les lignes horizontales et verticales. On peut voir dans la figure ci-dessous les possibilités d'interconnexions dans le PSM :



Il reste enfin à parler des blocs IOB, faisant l'interface entre le PAD externe du composant et l'intérieur de la FPGA. On trouve dans le bloc d'entrée sortie les éléments suivants :

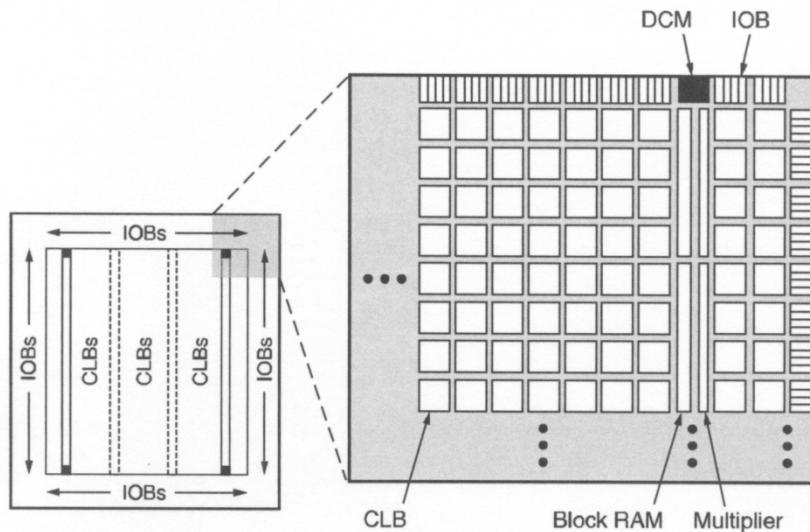
- un buffer d'entrée
- un buffer de sortie tri-state, permettant le réglage du slew-rate
- une résistance pull-up et une résistance pull-down programmables
- une bascule D d'entrée FF1 et une bascule D de sortie FF2
- des multiplexeurs M1 à M8 permettant l'aiguillage des signaux à l'intérieur de l'IOB



Structure FPGA famille Spartan 3 de Xilinx

Par rapport à la famille XC4000 de Xilinx, qui est déjà obsolète, on va la comparer avec la structure de la famille Spartan 3 de Xilinx est la plus récente, elle est en technologie 90nm, ce qui se fait de plus petit à l'heure actuelle dans le monde des circuits intégrés.

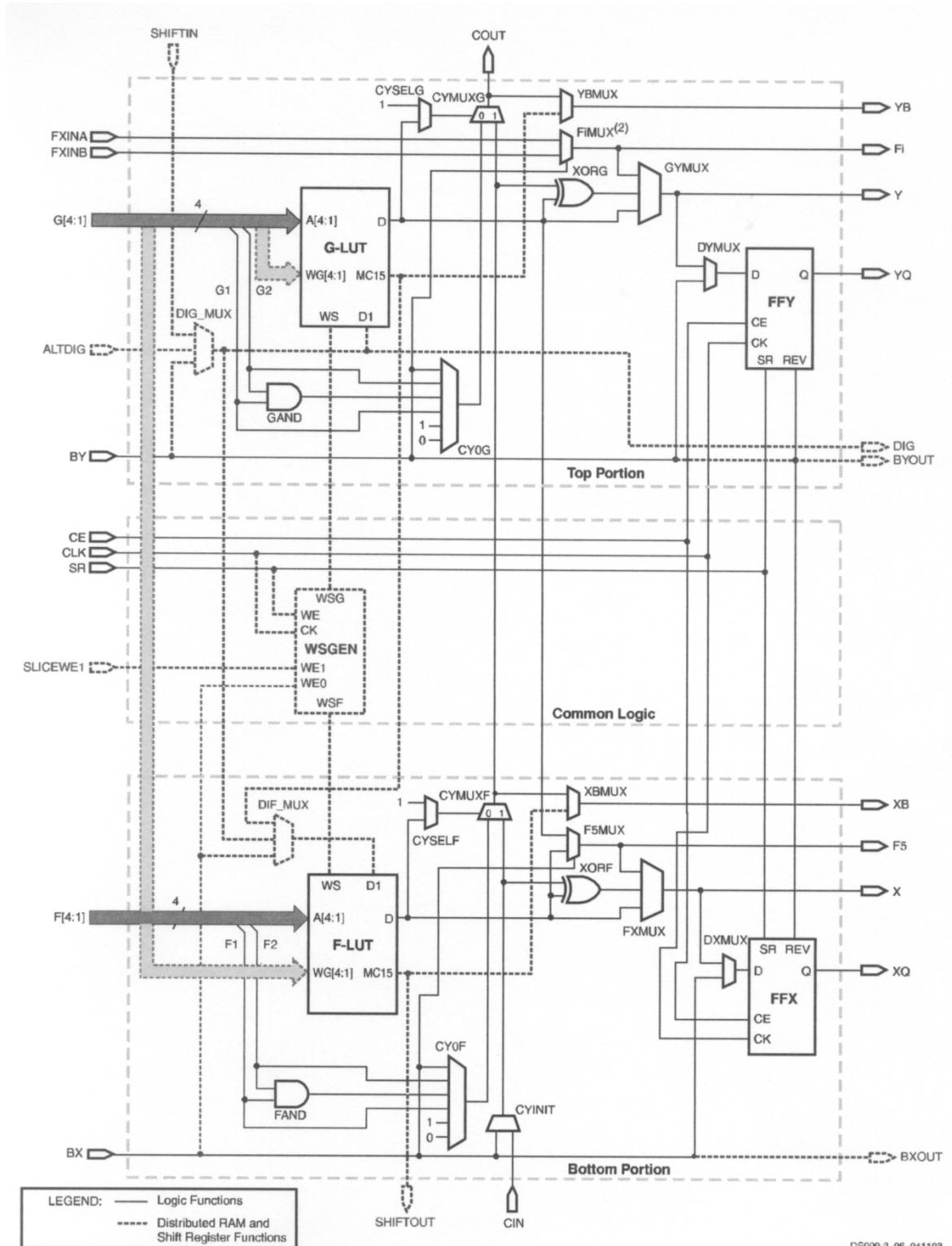
On peut voir dans la figure ci-dessous que le principe CLB et IOB est toujours le même, par contre les FPGA récentes incluent souvent des fonctions supplémentaires, on peut voir la structure de la famille Spartan 3 dans la figure ci-dessous :



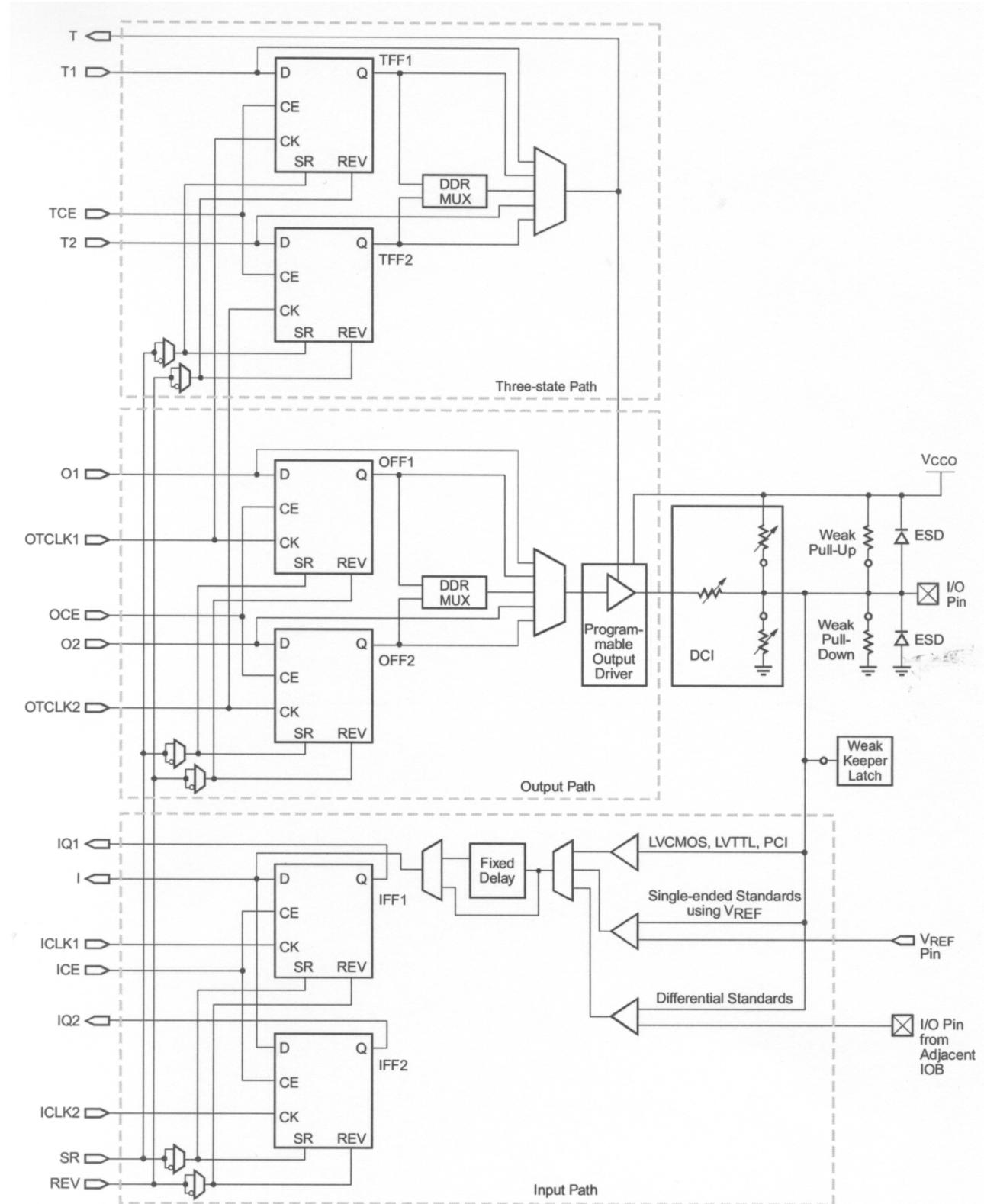
Block RAM : sont des zones de mémoire SRAM, permettant de stocker de programmes ou des données lorsqu'on synthétise un processeur à l'intérieur de la FPGA.

Multiplier : ce sont des multiplicateurs binaires hardware, permettant d'effectuer des multiplications de nombres binaires de 18 bit, ceci sans utiliser les ressources (CBL) de la FPGA.

La figure suivante nous montre la structure d'un CLB, on y retrouve toujours deux Look Up Table (G-LUT et F-LUT) permettant de réaliser des fonctions combinatoires, deux bascules (FFX et FFY) ainsi que les multiplexeurs programmables permettant d'aiguiller les signaux à l'intérieur du CLB.



La dernière figure ci-dessous nous montre la structure d'un IOB :

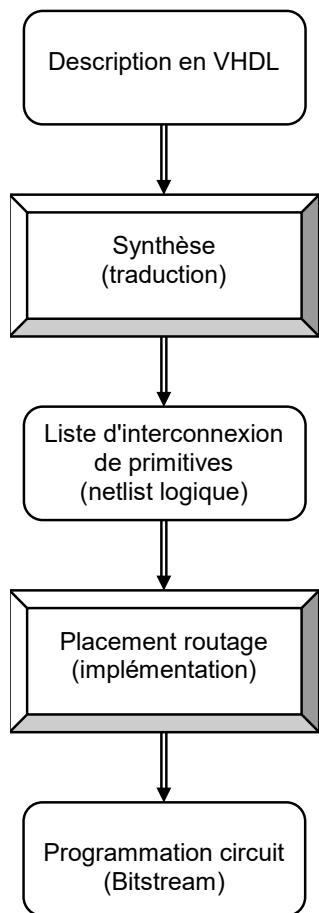


Dans le tableau ci-dessous, on peut voir les différents circuits de la famille Spartan-3 :

Device	XC 3S50	XC 3S200	XC 3S400	XC 3S1000	XC 3S1500	XC 3S2000	XC 3S4000	XC 3S5000
System Gates	50K	200K	400K	1000K	1500K	2000K	4000K	5000K
Logic Cells	1,728	4,320	8,064	17,280	29,952	46,080	62,208	74,880
18x18 Multipliers	4	12	16	24	32	40	96	104
Block RAM Bits	72K	216K	288K	432K	576K	720K	1,728K	1,872K
Distributed RAM Bits	12K	30K	56K	120K	208K	320K	432K	520K
DCMs	2	4	4	4	4	4	4	4
I/O Standards	23	23	23	23	23	23	23	23
Max Differential I/O Pairs	56	76	116	175	221	270	312	344
Max Single Ended I/O	124	173	264	391	487	565	712	784
Package	User I/O	User I/O	User I/O	User I/O	User I/O	User I/O	User I/O	User I/O
VQ100	63	63	-	-	-	-	-	-
TQ144	97	97	97	-	-	-	-	-
PQ208	124	141	141	-	-	-	-	-
FT256	-	173	173	173	-	-	--	-
FG320	-	-	221	221	221	-	-	-
FG456	-	-	264	333	333	-	-	-
FG676	-	-	-	391	487	489	-	-
FG900	-	-	-	-	-	565	633	633
FG1156	-	-	-	-	-	-	712	784

Les outils de développement des CPLD et FPGA

Cet outil va permettre au concepteur de programmer le circuit à partir de la description de la fonction à réaliser. Cette description peut être textuelle (VHDL, Verilog) ou graphique (symboles de fonction, graphe d'état, chronogrammes).



Description textuelle du comportement du circuit

Outil permettant la traduction de la description VHDL en logique :

le **SYNTHETISEUR**

Le synthétiseur nous fournit une liste d'interconnexion de primitives. Celles-ci seront définies pour la technologie choisie.

Le placeur-routeur (Fitter) permet de réaliser le placement routage de la liste d'interconnexion pour une technologie donnée. Cet outil est fourni par le fabricant des circuits utilisés.

Le programmateur permet de programmer le circuit, FPGA ou CPLD, avec le fichier de programmation créé par l'outil précédent. Le fabricant propose un programmateur, mais il existe des solutions générales.

La compilation va permettre dans un premier temps vérifier la cohérence de la description et la syntaxe du langage utilisé, puis d'effectuer une simulation fonctionnelle dans un premier temps.

Après avoir fait une simulation fonctionnelle approfondie (l'outil utilisé pour la simulation est ModelSim), c'est à dire avoir validé la conception et la description, le synthétiseur génère la netlist du circuit logique déjà fonction du circuit cible utilisé. Le synthétiseur n'est pas forcément propriétaire du fabricant de chip.

Le "placeur-routeur" (fitter pour les CPLD) effectue ensuite le placement et routage des blocs logiques. Dans le cas des CPLD et FPGA, le "placeur-routeur" est en général propriétaire du fabricant de ces circuits logiques.

La dernière étape, par toujours appliquée, pour autant que le design soit synchrone, est la vérification du timing ou le simulateur importe les temps de propagation calculés en fonction du placement routage. On utilise généralement le même testbench que pour la simulation fonctionnelle.

Vient enfin la programmation du circuit et la vérification du fonctionnement sur la carte. Si la simulation et la vérification ont été faites correctement, aucune erreur de fonctionnement doit apparaître.

Techniques de programmation et de test

Le Placeur-routeur ou le Fitter transforme la description structurelle du circuit en une table des fusibles consignée dans un fichier (JEDEC dans les cas simples, LOF, POF, etc., autrement). Pour la petite histoire, signalons que cette table peut contenir plusieurs centaines de milliers de bits, un par « fusible ou connexion ».

Traditionnellement, la programmation du circuit, opération qui consiste à traduire la table des fusibles en une configuration matérielle, se faisait au moyen d'un programmeur, appareil capable de générer les séquences et les surtensions nécessaires. La tendance actuelle est de supprimer cette étape de manipulation intermédiaire, manipulation d'autant plus malaisée que l'augmentation de la complexité des boîtiers va de pair avec celle des circuits. Autant il était simple de concevoir des supports à force d'insertion nulle pour des boîtiers DIL (dual in line) de 20 à 40 broches espacées de 2,54 mm, autant il est difficile et coûteux de réaliser l'équivalent pour des QFP (Quad Flat PACK) et autres BGA (Ball Grid Array), de 100 à plus de 1000 broches réparties sur toute la surface du boîtier. Une difficulté du même ordre se rencontre pour le test : il est devenu quasi impossible d'accéder, par des moyens traditionnels tels que les pointes de contact d'une « planche à clous », aux équipotentielles d'une carte. De toute façon, les équipotentielles du circuit imprimé ne représentent plus qu'une faible proportion des noeuds du schéma global : un circuit de 250 broches peut contenir 25000 bascules.

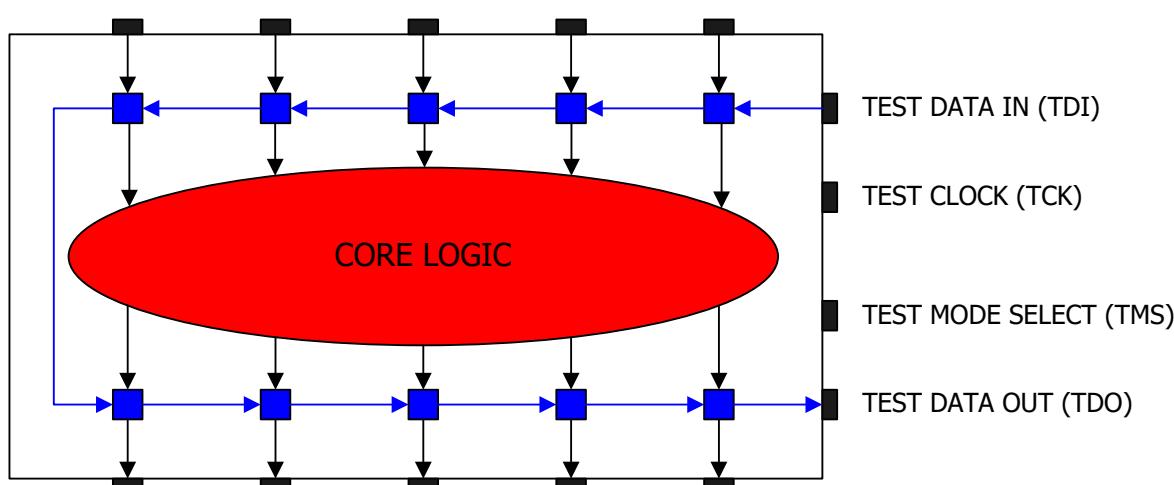
Les deux problèmes énumérés ci-dessus sont résolus grâce à une infrastructure hardware supplémentaire implantée dans le circuit : le port JTAG.

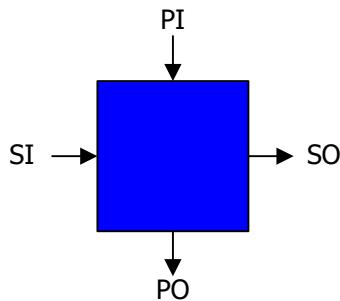
Le port JTAG

Fonctionnement normal, programmation et test : l'idée s'est imposé d'incorporer ces trois modes de fonctionnement dans les circuits eux-mêmes, comme partie intégrante de leur architecture. Pour le test de cartes, une norme existe : le standard IEEE 1149.1, plus connu sous le nom de Boundary-scan du consortium JTAG (Join Test Action Group). Face à la quasi-impossibilité de tester de l'extérieur les cartes multicouches avec des composants montés en surface, un mode de test a été défini, pour les VLSI numériques.

Principe de fonctionnement de l'architecture JTAG

Chaque signal primaire d'entrée et de sortie est complété avec un élément de mémoire appelé cellule Boundary-Scan. Les cellules qui agissent sur les composants JTAG d'entrée sont notées cellules d'entrée, de même que les cellules qui agissent sur les composants JTAG de sortie sont notées cellules de sortie. Entrées et sorties sont des notations relatives au noyau logique du composant (nous verrons par la suite les problèmes que peut entraîner le fait de noter entrées et sorties les interconnections entre deux ou plusieurs composants JTAG)





Chaque élément mémoire Boundary-scan peut:

- Capturer des données sur son entrée parallèle PI
- Mettre à jour des données sur sa sortie PO
- Transmettre des données séries avec ses voisins (SI to SO)
- Devenir transparent (PI to PO)

Le regroupement de cellules Boundary-Scan (appelées aussi cellules JTAG) est configuré dans un registre à décalage d'entrée/sortie.

Une opération de chargement parallèle appelée 'capture' permet de charger les données en entrée du composant JTAG vers l'entrée des cellules JTAG. En même temps, elle prend la valeur du noyau logique et la transfère vers la sortie du composant JTAG pour être ensuite chargée vers l'entrée de la cellule JTAG. Elle permet donc de charger la totalité des données à l'entrée des cellules JTAG.

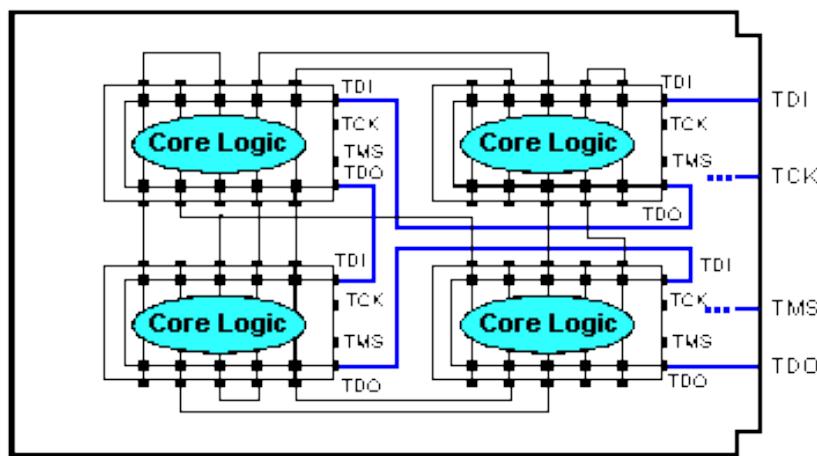
Une opération de déchargement appelée 'update' va décharger la valeur à l'entrée de la cellule JTAG vers le noyau logique. Une opération 'update' va donc suivre une opération 'capture'. Ces valeurs de remplacement à l'entrée du composant JTAG et du signal déjà présent à la sortie de la cellule JTAG sont donc transférées vers la sortie du composant JTAG, remplaçant ainsi la valeur de sortie générée par le noyau logique.

Les données sont donc modifiées autour du registre à décalage, dans un mode série, à partir du 'Test Data In' (TDI) et terminé par le 'Test Data Out' (TDO). Le 'Test Clock' (TCK) est alimenté par un autre circuit dédié à cette tâche. Le mode de fonctionnement est contrôlé par le signal de contrôle série : le 'Test Mode Select' (TMS).

Pour utiliser le chemin de scan

Le test JTAG n'influe en rien sur la fonctionnalité du noyau logique. En fait, le chemin du Boundary-Scan est indépendant de la fonction du composant. Nous allons par un exemple simple expliquer comment le chemin de scan fonctionne.

Soit la carte ci-dessous :



Cette figure montre une carte contenant quatre composants JTAG. Notons qu'il existe sur la carte un connecteur en entrée nommé TDI connecté sur l'entrée TDI du premier composant. Le TDO du premier composant est relié au TDI du second composant. Et ainsi de suite jusqu'à la sortie du dernier composant appelée TDO. Les TCK et TMS sont reliées à chacun des composants en parallèle.

Dans ce chemin, certains tests particuliers peuvent être appliqués à chacun des composants par l'intermédiaire du chemin de scan global. Ainsi il va être possible de charger une valeur spécifique dans les cellules JTAG d'entrées via la broche TDI de la carte (opération de décalage d'entrée), d'appliquer cette valeur au composant (opération update), de mémoriser la réponse du composant (opération capture), et de

sortir cette valeur sur la broche TDO de la carte (opération décalage de sortie). En fait les cellules JTAG peuvent donc être considérées comme des clous virtuels.

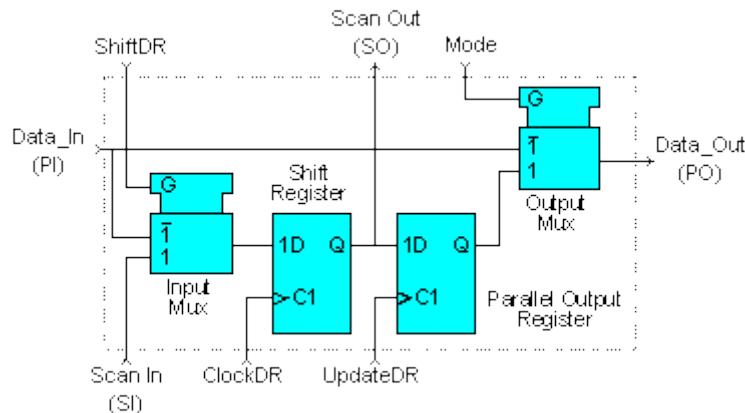


Figure 71 Cellule JTAG universelle

Notons qu'il s'agit ici d'un exemple de réalisation d'une cellule universelle JTAG puisque le standard IEEE 1149.1 n'impose pas le schéma de cette cellule mais uniquement son fonctionnement. Elle contient les quatre modes opératoires cités précédemment :

- normal
- update
- capture
- décalage série

L'élément de mémoire est caractérisé par une bascule D précédée et suivie d'un multiplexeur.

- En mode 'normal', les données introduites en entrée se retrouvent directement en sortie.
- En mode 'update', le contenu du registre de sortie est transféré en sortie.
- En mode 'capture', le signal de données d'entrée est routé vers le registre à décalage et la valeur est mémorisée par l'horloge DR (ClockDR) suivante.
- En mode décalage, le Scan_Out d'un registre à bascule est directement transféré au suivant.

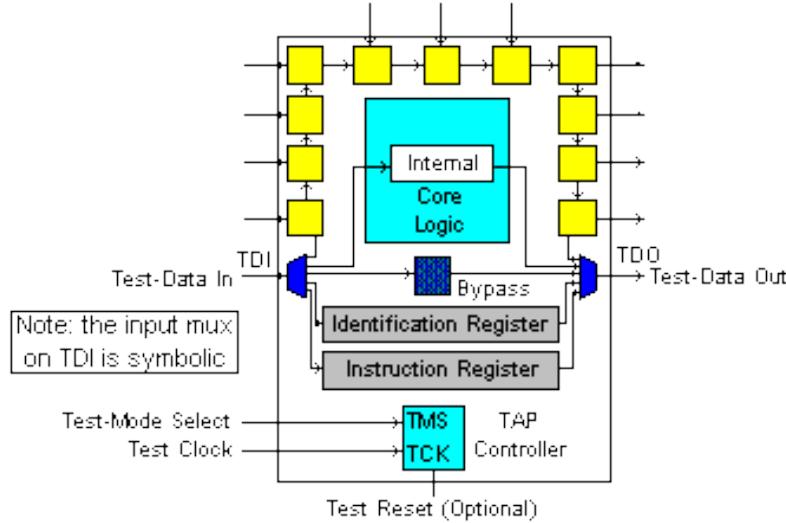
Il est important de noter que ni le mode capture ni le mode décalage n'interfèrent avec le mode normal.

Ceci permet de mémoriser la valeur d'une opération et de l'appliquer où l'utilisateur le désire pour une éventuelle inspection sans engendrer aucune interférence. Cette application de l'architecture JTAG a d'énormes potentiels pour des 'monitorings' en temps réels.

La méthode de test JTAG, comme nous l'avons indiqué précédemment, est définie par le standard IEEE 1149.1. Voyons précisément ce que cette norme impose.

Architecture d'un composant à la norme IEEE 1149.1

Après cinq ans de discussions, l'organisation JTAG a finalement proposé une architecture telle que le montre la figure ci-dessous :



Cette figure montre les éléments suivants :

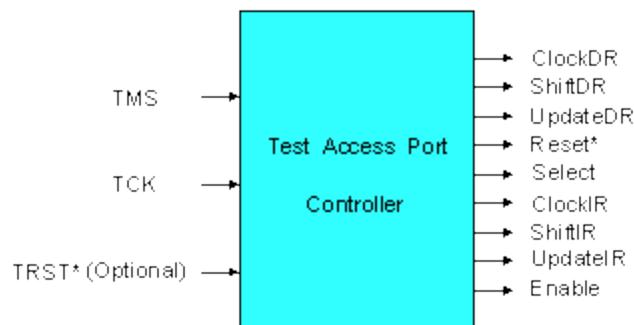
- Les quatre broches de test : Test Data In (TDI), Test Mode Select (TMS), Test Clock (TCK), Test Data Out (TDO), une broche optionnelle : le Test Reset (TRST*). Les regroupements de toutes ces broches forment le Test Access Port (TAP).
- Une cellule JTAG de chacune des broches d'entrées et de sortie du composant est reliée de manière interne à un registre Boundary-Scan série.
- Une machine d'état qui contrôle le TAP par l'intermédiaire de TCK et de TMS.
- Un registre d'instructions (IR) sur n bits ($n=2$) qui garde les instructions courantes.
- Un registre Bypass sur 1 bit (Bypass).
- Un registre d'identification optionnel sur 32 bits capable d'être chargé avec un code d'identification du composant.

Un seul registre peut être connecté du TDI vers le TDO (par exemple le registre IR, Bypass, Ident...). La sélection du registre se fait par le décodage du registre d'instruction (IR).

Nous allons voir dans le détail chacune des parties qui composent cette architecture.

Le Test Access Port

La représentation du Test Access Port (TAP) est donnée dans la figure ci-dessous :



Comme nous l'avons vu dans le chapitre précédent, le TAP est donc constitué de quatre signaux et d'un signal optionnel :

- **Test Data In** (TDI) : Signal de test série d'entrée dont la valeur par défaut vaut '1'.
- **Test Data Out** (TDO) : Signal de test série de sortie dont la valeur par défaut vaut 'Z'. Il n'est actif que pendant l'opération de décalage.
- **Test Mode Select** (TMS) : Signal de contrôle série d'entrée dont la valeur par défaut vaut '1'.
- **Test Clock** (TCK) : signal fournit l'horloge au test.
- **Test Reset** (TRST*) : signal qui contrôle la remise à '0' du TAP dont la valeur par défaut vaut '1'. Il est actif sur un niveau bas. C'est ce signal qui est optionnel.

TMS, TCK et TRST* (s'il existe) entrent dans une machine d'état qui produit les changements d'état du signal de contrôle. Ces signaux permettent également de contrôler le registre d'instruction ainsi que les différents registres de données.

La Figure 67 ci-dessous montre le diagramme de transition du TAP et suffit à la compréhension de son fonctionnement, sachant que la valeur qui va permettre au TAP de changer d'état est celle du TMS et que ce changement d'état est synchrone sur un front montant de l'horloge.

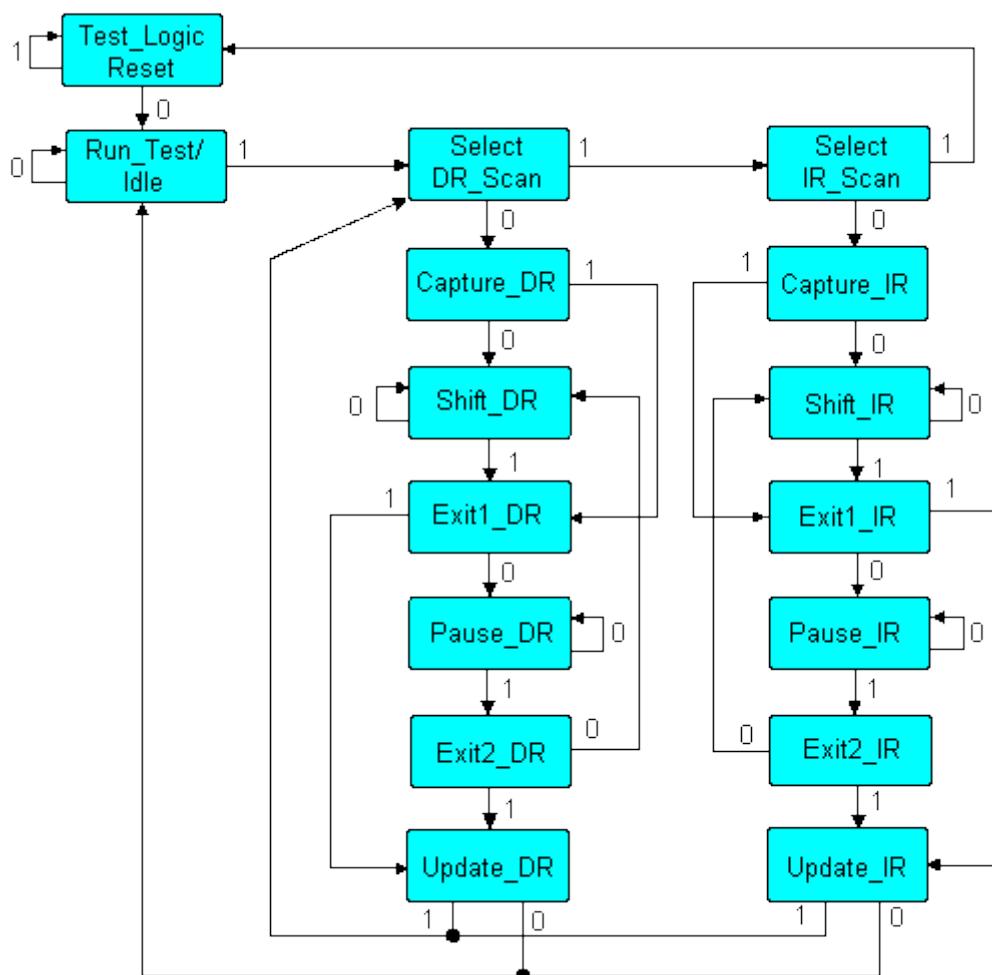


Figure 72 Graphe des états du TAP

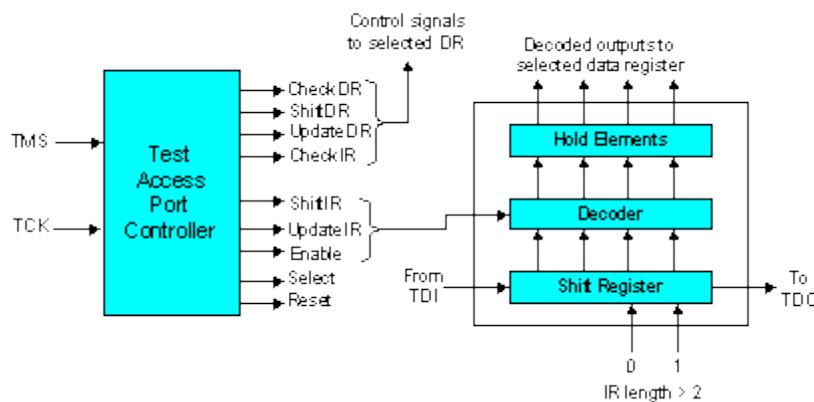
Le registre Boundary-Scan

Si nous regardons les cellules JTAG plus attentivement, nous nous rendons compte qu'elles sont chaînées de manière à former le registre Boundary-Scan. L'ordre de la chaîne est déterminé par l'agencement physique des composants sur la carte et/ou par le nombre de couches de la carte. Le registre Boundary-Scan est

sélectionné par une instruction commandée au registre d'instructions : instructions Extest, Sample/Preload ou Intest que nous verrons dans le chapitre suivant.

Le registre d'instructions

Un registre d'instructions est composé d'un registre à décalage, qui peut être connecté au TDI et au TDO, et une section de maintien qui garde les instructions courantes comme le montre la figure ci-dessous :



Il peut y avoir plusieurs décodeurs logiques entre ces deux sections (cela va dépendre de la taille du registre ainsi que du nombre d'instructions). Le signal de contrôle du registre d'instructions est issu du contrôleur du TAP et permet soit de décaler les valeurs d'entrée/sortie vers le registre à décalage du registre d'instruction soit de charger directement vers la section de maintien (opération 'update' le contenu du registre à décalage. Il est donc possible de charger (opération 'capture') certaines valeurs vers le registre à décalage du registre d'instruction.

La taille du registre d'instructions doit être d'au moins 2 bits (pour permettre les différentes instructions : Bypass, Sample/Preload, Extest) mais sa taille maximale n'est pas définie.

Les différentes instructions

L'architecture IEEE 1149.1 impose trois instructions qui sont :

- **Bypass** : L'instruction Bypass permet de passer au travers des composants avec un bit de décalage. Tous les bits doivent être à '1' pour qu'elle soit exécutée.
- **Extest** : L'instruction Extest sélectionne le registre Boundary-Scan en déconnectant le composant JTAG. Elle prépare au test des interconnections. Tous les bits doivent être à '0' pour qu'elle soit exécutée. Sa taille de code n'est pas définie.
- **Sample/Preload** : L'instruction Sample/Preload sélectionne le registre Boundary-Scan mais sans déconnecter le composant JTAG.

Le standard IEEE 1149.1 autorise un certain nombre d'instructions optionnelles qui sont les suivantes :

- **Intest** : Elle sélectionne le registre Boundary-Scan pour le préparer au test du noyau logique interne.
- **Idcode** : Elle sélectionne le registre d'identification placé entre TDI et TDO pour le préparer au chargement du code Idcode et à sa lecture au travers le TDO. Si l'instruction Idcode est chargée, il n'y a pas de registre d'identification présent sur la carte et l'instruction Idcode peut être vu comme si l'utilisateur chargeait l'instruction Bypass.
- **Runbist** : C'est l'instruction qui initialise une routine de test interne et charge le résultat dans le registre placé entre le TDI et le TDO.

En 1993 une révision du standard IEEE 1149.1 apporte deux nouvelles instructions :

- **Clamp** : C'est une instruction qui pilote la valeur de présélection vers la sortie de certains composants et qui sélectionne le registre Bypass entre TDI et TDO.
- **High** : C'est une instruction similaire à Clamp mis à part le fait qu'elle permet de laisser la valeur des broches des composants de sortie sur un niveau logique haute impédance. Elle va également sélectionner le registre Bypass entre TDI et TDO.

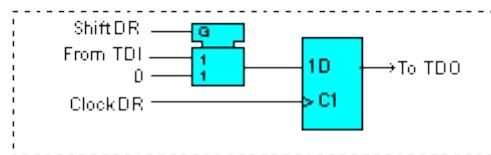
Remarque générale :

A l'exception de Bypass, le code de toutes les autres instructions n'est pas défini. La taille minimale du registre d'instructions est de 2 bits et sa taille maximale est indéfinie.

Le registre Bypass

La figure ci-dessous montre un schéma classique d'un registre Bypass

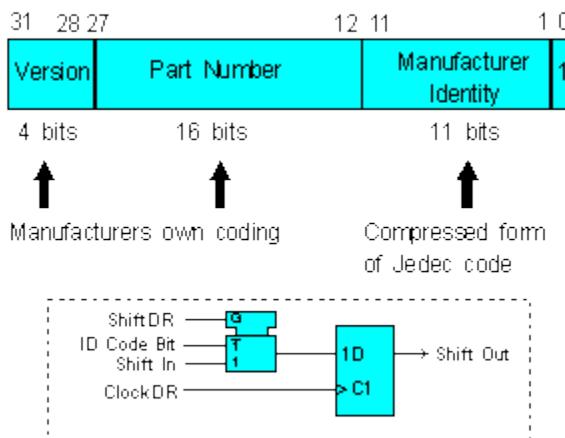
- One-bit shift stage
- Selected by the BYPASS instruction
- No parallel output
- Captures a hard-wired 0



Le registre Bypass est un registre sur 1 bit, sélectionné par l'instruction Bypass et qui provient de la fonction de décalage. Il n'y a pas de sortie parallèle (ce qui signifie que le signal de contrôle Update_DR n'agit pas sur le registre). Au contraire, le signal de contrôle Capture_DR agit sur le registre et l'autorise à garder une valeur de '0'.

Le registre d'identification

Le registre optionnel d'identification est un registre 32 bits qui contient les opérations de capture et de décalage. Une fois que l'opération de capture a été réalisée, le code d'identification sur 32 bits peut être chargé vers TDO pour une inspection. La figure ci-dessous montre une manière possible d'incrémenter une cellule du registre 32 bits.



Programmation in situ (ISP)

Les circuits programmables in situ se développent dans le monde des PLDs et CPLDs en technologie FLASH. Du simple 22V10, à des composants de plus de 1 million de portes équivalentes, il est possible de programmer (et de modifier) l'ensemble d'une carte, sans démontage, à partir d'un port parallèle de PC. Les technologies FLASH (CPLD) conservent leur configuration en l'absence d'alimentation. On utilise généralement le port JTAG pour la configurations des circuits logiques programmables type Flash.

Configurables dynamiquement, les FPGAs à cellules SRAM offrent des possibilités multiples de chargement de la mémoire de configuration :

- Chargement automatique, à chaque mise sous tension, des données stockées dans une mémoire PROM. Les données peuvent être transmises en série, en utilisant peu de broches du circuit, ou en parallèle octet par octet, ce qui accélère la phase de configuration mais utilise, temporairement du moins, plus de broches du circuit. Plusieurs circuits d'une même carte peuvent être configurés en coopération, leurs automates de chargement assurent un passage en mode normal coordonné, ce qui est évidemment souhaitable.
- Chargement, en série ou en parallèle, à partir d'un processeur maître. Ce type de structure autorise la modification rapide des configurations en cours de fonctionnement. Cette possibilité est intéressante, par exemple, en traitement de signal.

PLDS, CPLDS, FPGAS : QUEL CIRCUIT CHOISIR?

Dans le monde des circuits numériques les chiffres évoluent très vite, beaucoup plus vite que les concepts. Cette impression de mouvement permanent est accentuée par les effets d'annonce des fabricants et par l'usage systématique de la publicité comparative, très en vogue dans ce domaine. Il semble que doivent se maintenir trois grandes familles :

- Les PLDs et CPLDs en technologie FLASH, utilisant une architecture somme de produits. La tendance est à la généralisation de la programmation in situ, rendant inutiles les programmeurs sophistiqués. Réservés à des fonctions simples ou moyennement complexes, ces circuits sont rapides (jusqu'à environ 500 MHz) et leurs caractéristiques temporelles sont pratiquement indépendantes de la fonction réalisée. Les valeurs de fréquence maximum de fonctionnement de la notice sont directement applicables.
- Les FPGAs à SRAM, utilisant une architecture cellulaire. Proposés pratiquement par tous les fabricants, ils couvrent une gamme extrêmement large de produits, tant en densités qu'en vitesses. Reprogrammables indéfiniment, ils sont devenus reconfigurables rapidement (200 ns par cellule), en totalité ou partiellement.
- Les FPGAs à antifusibles, utilisant une architecture cellulaire à granularité fine. Ces circuits tendent à remplacer une bonne partie des ASICs prédiffusés. Programmables une fois, ils présentent l'avantage d'une très grande routabilité, d'où une bonne occupation de la surface du circuit. Leur configuration est absolument immuable et disponible sans aucun délai après la mise sous tension ; c'est un avantage parfois incontournable.

Critères de performances

Outre la technologie de programmation, capacité et vitesse sont les maîtres mots pour comparer deux circuits. Mais quelle capacité, et quelle vitesse?

Puissance de calcul

Les premiers chiffres accessibles concernent les nombres d'opérateurs utilisables.

Nombre de portes équivalentes

Le nombre de portes est sans doute l'argument le plus utilisé dans les effets d'annonce. En 2006 la barrière des 10'000 000 portes est largement franchie. Plus délicate est l'estimation du nombre de portes qui seront inutilisées dans une application, donc le nombre réellement utile de portes.

Nombre de cellules

Le nombre de cellules est un chiffre plus facilement interprétable : le constructeur du circuit a optimisé son architecture, pour rendre chaque cellule capable de traiter à peu près tout calcul dont la complexité est en relation avec le nombre de bascules qu'elle contient (une ou deux suivant les architectures). Trois repères chiffrés : un 22V10 contient 10 bascules, la famille des CPLDs va de 32 bascules à quelques centaines et celle des FPGAs s'étend d'une centaine à quelques centaines de milliers.

Dans les circuits à architectures cellulaires, il est souvent très rentable d'augmenter le nombre de bascules si cela permet d'alléger les blocs combinatoires (pipe line, codages one hot, etc.).

Nombre d'entrée/sorties

Le nombre de ports de communication entre l'intérieur et l'extérieur d'un circuit peut varier dans un rapport deux, pour la même architecture interne, en fonction du boîtier choisi. Les chiffres vont de quelques dizaines à quelques plus d'un millier de broches d'entrées-sorties.

Vitesse de fonctionnement

Les comportements dynamiques des FPGAs et des PLDs simples présentent des différences marquantes. Les premiers ont un comportement prévisible, indépendamment de la fonction programmée ; les limites des seconds dépendent de la fonction, du placement et du routage. Une difficulté de jeunesse des FPGAs a été la non reproductibilité des performances dynamiques en cas de modification, même mineure, du contenu d'un circuit. Les logiciels d'optimisation et les progrès des architectures internes ont pratiquement supprimé ce défaut; mais il reste que seule une analyse et une simulation post synthèse, qui prend en compte les paramètres dynamiques des cellules, permet réellement de prévoir les limites de fonctionnement d'un circuit.

Consommation

Les premiers circuits programmables avaient plutôt mauvaise réputation sur ce point. Tous les circuits actuels ont fait d'importants progrès en direction de consommations plus faibles. Exemple marquant la famille CPLD CoolRunner II de Xilinx.

ASIC (Application Specific Integrated Circuit)

Si les composants précédents pouvaient être développés avec un simple ordinateur, ceux que nous abordons maintenant nécessitent l'intervention d'un fondeur qui produira le circuit demandé à partir des masques fournis par son client. Ici encore, le terme programmable n'est pas des plus judicieux, les connexions entre les éléments étant dessinées sur les masques. Les temps et coûts de productions sont importants. On distingue trois types d'ASIC classé par ordre croissant de configurabilité.

Les prédiffusés (gate arrays)

Ils contiennent une nébuleuse de transistors ou de portes à interconnecter avec les problèmes de routage et de délais que cela comporte.

Les précaractérisés (standard cell)

On utilise cette fois des bibliothèques de cellules standards à placer sur le semiconducteur

Les "fulls customs"

Ils sont entièrement définissables par le client. Ces circuits conduisent à la réalisation de tous les composants VLSI comme les microprocesseurs.

Comparaison et évolution

Il est difficile de comparer les ASIC et les CPLD/FPGA. Les méthodes et temps de développement ne sont pas du tout les mêmes. L'utilisation des ASIC va surtout être justifiée par la production en grande série du circuit à réaliser. Il faut cependant noter que très marginaux en 1990, les CPLD et FPGA ont pris en 2000 près de 95% du marché des ASIC.

En ce qui concerne les CPLD et FPGA uniquement, comme nous l'avons précisé au début, il est très difficile de donner des ordres de grandeurs et des éléments de comparaison dans un domaine qui évolue aussi rapidement, et où la concurrence entre les constructeurs a tendance à brouiller les pistes. Les principaux critères de choix seront :

- la vitesse de fonctionnement
- le nombre d'entrée/sorties
- le nombre de portes
- la consommation
- le prix

Un bon résumé de la situation présente les FPGA comme des circuits à forte densité d'intégration et les CPLD comme des circuits rapides mais petits. Le Tableau 9 donne les principales caractéristiques de chaque catégorie :

Tableau 9 Comparaison entre les différents types de circuits logiques

	circuits MSI (à titre de comparaison)	PLD	CPLD	FPGA
nombre de portes (ordre de grandeur)	100	150	40 000	10 000 000
vitesse de fonctionnement (ordre de grandeur)	100 MHz	200 MHz	500 MHz	500 MHz
technologie de connexion		MOS à grille flottante	MOS à grille flottante	SRAM ou antifusible
codage des fonctions		PLA	PLA et LUT	LUT et MUX

La notion de nombre de portes supposant implicitement que toutes sont utilisées, on préfère souvent parler de portes équivalent ou utilisables (usable gates) pour caractériser la densité d'un circuit. Il existe un rapport 2 à 4 entre ces deux termes.

L'utilisation des propriétés intellectuelles (IP) gratuite ou payante se généralise de plus en plus, et permet facilement l'intégration de fonctions complexes comme les microprocesseurs au sein des circuits. Ces fonctions se présentent comme un programme, optimisé ou non pour un composant. De même, à l'intérieur des FPGA on trouve maintenant des zones optimisées pour implanter de la mémoire (20 kbits dans un FLEX10K par exemple) et des microprocesseurs très performants comme des coeurs ARM dans les Zynq de Xilinx.

L'utilisation de FPGA SRAM reconfigurable en cours d'utilisation pour s'adapter à l'évolution de l'environnement devient également courante.

Bibliographie

Ronald J.Tocci –Circuits numériques (théorie et applications) 2ème édition. Dunod

Philippe Larcher –Introduction à la synthèse logique. Eyrolles

Jacques Weber, Maurice Meaudre –Le langage VHDL (cours et exercices) 2ème édition. Dunod

Noël Richard –Electronique numérique et séquentielle (pratique des langages de description de haut niveau). Dunod

John F.Wakerly –Digital design (principles & practice) third edition updated. Prentice Hall

Etienne Messerli –Manuel VHDL EIVD

Philippe Darche –Architecture des ordinateurs. Vuibert

Alexandre Nketsa –Circuits logiques programmables Mémoires, CPLD et FPGA. Ellipse

Médiagraphie

<http://www.xilinx.com/>

<http://perso.wanadoo.fr/xcotton/electron/coursetdocs.htm>

<http://www.chez.com/jtag/description.htm>

Lexique

ABEL : langage de programmation des circuits de faible densité d'intégration.

ASIC (Application Specific Integrated Circuit) : circuit non programmable configuré lors de sa fabrication pour une application spécifique.

CPLD (Complex Programmable Logic Device) : circuit intégrant plusieurs PLD sur une même pastille.

EEPROM ou E2PROM (Electrical Erasable Programmable Read Only Memory) : mémoire ROM programmable et effaçable électriquement.

E2PAL (Electrical Erasable PAL) : voir GAL

EPLD (Erasable PLD) : voir GAL.

EPROM (Erasable PROM) : PROM effaçable par UV.

Flash EEPROM : EEPROM utilisant 2 transistors par point mémoire ; utilisé pour les connexions dans les CPLD.

FPGA (Field Programmable Logic Array) : réseau programmable à haute densité d'intégration.

FPLD (Fiel Programmable Logic Device) : terme générique pour les CPLD et FPGA.

FPLS (Fiel Programmable Logic Sequencer) : ancien nom des PAL à registre.

GAL (Generic Array Logic) : PLD programmable et effaçable électriquement

ISP (In Situ Programmable) : caractérise un circuit reprogrammable sur l'application.

JEDEC : organisme de normalisation, donnant son nom aux fichiers de programmation des PLD.

LSI (Large Square Integration) : circuits intégrants quelques centaines à quelques milliers de transistors.

LUT (Lock Up Table) : nom donné aux cellules mémoire réalisant les fonctions combinatoires dans les CPLD et FPGA.

MSI (Medium Square Integration) : circuits intégrants quelques centaines de transistors.

MUX : abréviation pour multiplexeur

PAL (Programmable Array Logic) : ancien nom des PLD.

PLA (Programmable Logic Array) : réseau à matrice ET et OU permettant la réalisation de fonctions combinatoires.

PLD (Programmable Logic Device) : circuit logique programmable intégrant une matrice ET programmable, une matrice OU fixe et plusieurs cellules de sortie.

PROM (Programmable Read Only Memory) : mémoire ROM programmable.

SPLD (Simple PLD) : par opposition aux FPLD, voir PLD.

SRAM (Static Random Acess Memory) : technologie utilisée pour les connexions dans les CPLD et FPGA.

SSI (Small Square Integration) : circuits intégrant quelques portes.

Verilog : langage de synthèse des circuits numériques

VHDL (Very high speed or scale integrated circuits Hardware Description Language) : langage de modélisation et de synthèse des circuits numériques.