MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOULOUD MAMMERI DE TIZI-OUZOU

FACULTE DE GENIE ELECTRIQUE ET D'INFORMATIQUE

DEPARTEMENT D'ELECTRONIQUE

MEMOIRE DE MAGISTER

Spécialité : Electronique

Option : Micro-électronique

Présenté par :

Mr. NACHEF Toufik

Thème :

« Implémentation d'une instrumentation sur un FPGA ».

Devant le jury d'examen :

Mr Belkaid Mohamed Saïd	Professeur,	UMMTO	Président
Mr Laghrouche Mourad	Maître de Conférences,	UMMTO	Rapporteur
Mr MOUHELBI Hassane	Professeur,	UMMTO	Examinateur
Mr MELAH Rabah	Maître de Conférences A	UMMTO	Examinateur
Mr BENAMANE Kamal	Maître de Conférences B	UMMTO	Examinateur

Soutenance le : 28/11/2011

R E M E R C I E M E N T S

Le travail présenté dans ce mémoire a été effectué au département d'électronique, laboratoire LATAG de l'université Mouloud Mammeri de Tizi-Ouzou.

Je remercie vivement Monsieur **LAGHROUCHE Mourad**, Maître de Conférences à l'université Mouloud Mammeri, d'avoir proposé et dirigé ce travail de si près, avec une qualité remarquable. J'ai grandement apprécié ses compétences et qualités scientifiques ainsi que son intelligence et son intuition sans oublier ses indéniables qualités humaines, à travers un suivi régulier, une grande disponibilité et une attention minutieuse accordés à l'avancement des travaux. Je lui présente toute ma gratitude de m'avoir initié à la recherche scientifique.

Mes vifs remerciements vont aussi à Monsieur **Mohand Said BELKAID**, Professeur à l'université Mouloud Mammeri et doyen de la faculté de Génie Electrique et d'Informatique, pour l'honneur qu'il me fait de présider mon jury et par-dessus tout je le remercie de m'avoir donné la chance de suivre la voie de la recherche scientifique et d'intégrer le laboratoire LATAG. Il nous a toujours enseigné que les valeurs morales d'un homme précèdent ses diplômes, il nous a toujours traité comme ses enfants et on espère qu'un jour on pourra honorer ses enseignements.

J'exprime mes sincères remerciements aux membres du jury, **Mr MOHELLEBI Hassane** Professeur à l'UMMTO, **Mr MELLAH RABAH** Maître de Conférences de classe A à l'UMMTO et **Mr BENAMANE Kamal** Maître de Conférences de classe B à l'UMMTO, pour avoir eu la gentillesse de participer à mon jury.

Mes remerciements pour **Mr TAHANOUT**, d'avoir accepté de prendre part à des séances de travail et de nous avoir accordé de son temps et qui a contribué pour l'orientation de ce travail dans la bonne direction.

Mes remerciements pour **Mr ZIRMI Rachid**, et **Mr HEMICHE** pour leurs encouragements et leurs conseils.

Mes remerciements pour mon frère **Jugutha** pour son soutien moral et son appui pour que je continue dans la voie des études et de la recherche.

Je saisis aussi cette occasion pour remercier l'ensemble des enseignants du département d'électronique ayant consacré leurs vies sur la voie noble de l'enseignement.

Que ceux qui se sentent oubliés trouvent ici ma profonde gratitude et mes chaleureux remerciements pour leur concours dans l'accomplissement de ce travail.

Mr. Toufik NACHEF
O c t o b r e 2 0 1 1

Sommaire

Introduction générale : 1

Chapitre I :... 4

I.1 Les circuits logiques programmables : 4

1.1 Introduction : 4

1.2 Définition des PLDs (Programmable Logic Device): 4

1.3 Structure de base d'un PLD : 5

1.4 Les différentes familles de PLDs : 7

1.4.1 Les PALs (Programmable Array Logique): 7

1.4.2 Les GALs (Generique Array Logic): 9

1.4.3 Les EPLDs (Erasable Programmable logique Device): 9

1.4.4 Les CPLDs : 10

I.4.5 Les FPGA Field Programmable Gate Array ou réseau logique programmable sur site : 10

1.5 Étude des FPGAs : 11

1.5.1 Présentation et généralités sur les FPGAs : 11

1.6 Etude de la Virtex 4:... 18

1.6.1 Interfaces d'entrés/sorties IOB (Input Output Bloc) : 18

1.6.2 Les blocs logiques de base de la Virtex-4, Configurable Logic Block (CLB) :...... 19

1.6.3 Les interconnexions : 21

1.6.4 Nomenclature des Virtex-4 : 21

I.2. Les langages de description matérielle HDL (Hard Ware Description Langages) :.......... 21

2.1 Évolution des HDL : 21

2.2 Utilité des HDL : 23

2.3 Exemples de langages de description matérielle HDL : 24

2.3.1 VHDL (Very High Speed Integrated Circuits Hardware Description Language): . 24

2.3.2 Verilog :... 24

2.3.3 Comparaison entre VHDL et VERILOG et choix : 24

2.4 Le langage de description matériel VHDL (Very High Speed Integrated Circuits Hardware Description Language): 25

2.5.1 Structure d'un programme VHDL le couple entity , architecture:........ 26

2.5.2 Éléments du langage VHDL : 27

2.5.3 Structure d'un programme sous VHDL : 29

2.5.4 Les TESTBENCHS :........ 30

Chapitre II : 31

II. Introduction : 31

II.1 Capteurs : principes fondamentaux. 31

1.1 Caractéristiques générales des capteurs : 31

1.2 Les différents types de capteurs : 32

1.2.1 Les capteurs actifs : 32

1.2.2 Les capteurs passifs : 33

1.2.3 Les capteurs composites :........ 34

1.2.4 Les capteurs intégrés : 34

1.2.5 Les capteurs intelligents : 35

1.3 Les grandeurs d'influences : 36

1.4 La chaîne de mesure :........ 36

1.5 Étalonnage du capteur : 39

1.5.1 Étalonnage simple : 40

1.5.2 Étalonnage multiple : 40

1.5.3 Validité d'un étalonnage : répétabilité et interchangeabilité........ 41

II.2. Conditionneurs des capteurs passifs et conditionneurs du signal :........ 41

2.1 Conditionneurs des capteurs passifs :........ 41

2.2 Les conditionneurs du signal :........ 42

II3. Interface de communication I2C (Inter Integrated circuit):........ 43

3.1 Présentation du bus I2C : 43

3.3.1 Caractéristiques générales : 44
3.4 Le protocole I2C : 46
3.4.1 Le signal START et le signal STOP : 46
3.4.2 Transfert de données : 47
3.4.3 Synchronisation d'horloge et Arbitrage : 48
3.4.4. Le Clock Stretching ou ralentissement de la vitesse : 50
3.4.5 L'adressage des transferts : 50
Chapitre III : 55
III. Introduction : 55
III.1 Le graphe d'états : 56
1.1 Représentation graphique du comportement d'une machine à états finis : 56
1.2 Règles de vérification d'une représentation graphique : 57
1.3 La composition de la machine à états finis : 57
1.3.1 Calcul de l'état futur : 57
1.3.2 Le registre d'état RE : 58
1.3.3 Le calcul des sorties : 59
1.4 Le codage des états : 59
1.4.1 Représentation des différents états sous forme de mots binaires : 59
1.4.2 Méthodes de choix du codage : 61
III.2. Conception de la machine finis à états régissant le bus I2C : 61
2.1 Caractéristiques du contrôleur I2C : 61
3.2 Description des signaux : 62
3.3 Disposition du contrôleur I2C dans un circuit : 63
3.4 Schéma block du contrôleur I2C : 63
3.5 Interface microcontrôleur : 64
3.5.1 Interface logique de décodage d'adresse : 64
Séquencement de la machine d'interface uC : 65

3.5.2 Les registre du contrôleur I2C : 66

3.6 L'interface logique du contrôleur I2C : 67

3.6.1 Détection des conditions START et STOP : 68

3.6.2 Génération de SCL, SDA, START et STOP : 68

3.6.3 La machine à états finis principale du contrôleur I2C : 71

3.7 Présentation du logiciel de simulation ModelSim: 73

3.8 Simulation du code VHDL : 75

3.8.1 Simulation des machines à états : 76

Chapitre IV : 80

IV. Introduction : 80

IV.1. Présentation de l'environnement de développement « XILINX Integrated Software Environment (ISE) » : 80

1.1 Les étapes pour l'implémentation d'une spécification HDL sur un FPGA : 81

1.1.1 Spécification : 82

1.1.2 Vérification : 82

1.1.3 Implémentation : 82

1.1.4 Débuggage du système : 83

1.2 Interface graphique de l'environnement ISE 9.2 : 83

IV.2 La plateforme de développement VIRTEX4 : 84

IV.3 Synthèse et Implémentation du contrôleur I2C sur la VIRTEX4 : 85

3.1 Synthèse de la description matérielle du contrôleur I2C : 85

3.2 Implémentation du contrôleur I2C sur la VIRTEX4 : 88

Conclusion générale : 90

Bibliographie : 93

Nomenclature

Nomenclature

HDL : Hardware Description Language (langage de description matériel).

VHDL: Very high Speed Integrated Circuit Hardware Description Language.

VHDL-AMS: Very High Speed Integrated Circuits Hardware Description Language For Analog And Mixed Signal.

RTL: Register Transfer Level.

FPGA: Field Programmable Gate Array (réseau de cellules logiques programmables).

ASIC : Circuit logique programmable lors de sa fabrication.

PAL: Programmable Array Logique (réseau logique programmable)

GAL: Generique Array Logic (réseau logique générique)

EPLD: Erasable Programmable logique Device,

EEPROM ou E2PROM : Electrical Erasable Programmable Read Only Memory (Mémoire effaçable électriquement à lecture seule).

RAM : Random Acces Memory (mémoire à lecture aléatroire).

OLMC: Out Put Logic Macro Cell

MAX: Multiple Array matriX

OE: Output Enable (validation des sorties).

CLB: Configurable Logic Bloc.

LUT : Lookup Table.

IOB: Input Output Bloc.

LE: Logic Element.

LAB: Logic Array Bloc.

NETLIST : Les interconnexions entre les composants qui interviennent avant le routage.

NGD: XILINX Native Database.

NGC: XILINX Specific file.

TCL: Tool Command Language

ISE: Integrated Software Environment.

JTAG: Joint Test Action Group

JDEC: Joint Electron Device Council.

BITSTREAM: Fichier de programmation des FPGA au format JDEC.

I2C ou IIC: Inter Integrated Circuits.

SDA: Serial Data.

SCL: Serial Clock.

L'évolution des besoins de l'homme pour apprivoiser son environnement, d'améliorer son confort et sa qualité de vie ont mené ce dernier à rechercher des solutions en fonction des données scientifiques et technologiques. Des évolutions sociales, de nouveaux goûts de nouvelles exigences socioculturelles ont incité les industriels à rechercher des moyens de satisfaire la société moderne. Un état des lieux qui a donné naissance à la concurrence industrielle pour acquérir la plus grande part de marché. Pour parvenir à devancer son concurrent avec un produit nouveau ou amélioré tout en réduisant le coût de reviens. Les industriels recherchent toujours de nouveaux outils et moyens.

L'électronique représente un très large marché très lucratif et un domaine très dynamique riche en nouvelles découvertes et applications. Un domaine qui se caractérise par la tendance à la miniaturisation et la simplification. Des systèmes compacts, légers, économiques, ergonomiques et fiables.

Cette course industrielle et scientifique a permis de pousser plus loin ce progrès et l'amélioration des systèmes en intégrant plus de transistors dans une surface de semi-conducteur la plus petite que possible. Des circuits intégrés de plus en plus denses et performants, les microprocesseurs, les ASICs, les FPGAs.

Les circuits FPGAs (Field Programmable Gate Array) sont les circuits qui ont permis de pousser encore plus loin le progrès technologique en électronique. En effet, le degré de maturité des FPGAs fait d'eux aujourd'hui une solution pour remplacer les ASICs (circuits intégrés à application spécifique) et les processeurs personnalisés dans des applications de contrôle et de traitement de signaux.

Réduire les coûts, augmenter les performances et la fiabilité d'un circuit, diminuer les délais de mise en place d'un prototype opérationnel, maintenabilité et réduire l'encombrement des composants sur circuit, sont des atouts conjugués dans un FPGA. Ces atouts justifient le succès de ces composants dans tous les domaines d'applications de l'électronique. L'aéronautique, l'aérospatial, l'industrie, les télécommunications, les appareils médicaux, l'électroménager…etc, tous les domaines trouvent le bon compromis en intégrant les FPGAs.

L'instrumentation évolue en fonction de l'évolution des technologies microélectroniques. D'un côté, des progrès au niveau du capteur plus précisément le corps d'épreuve pour améliorer les caractéristiques de ce dernier. D'un autre côté, profitant des

progrès réalisés dans l'intégration et miniaturisation des systèmes en embarquant le conditionneur associé au capteur.

Le capteur intelligent correspond principalement à l'intégration dans le corps du capteur d'un organe de calcul interne (microprocesseur, microcontrôleur, interne au capteur permettant un traitement local et l'élaboration un signal numérique) des conditionneurs spécifiques, d'une mémoire, d'une interface de communication.

L'instrumentation est un domaine de l'électronique particulièrement exigeant. En effet, la précision, la fiabilité, la sensibilité sont les principales caractéristiques d'un capteur.

L'application des FPGA pour l'instrumentation se révèle très bénéfique. En effet, les caractéristiques principales d'un capteur peuvent être optimisées sans pour autant se soucier d'influencer un autre paramètre. En ce qui concerne la partie matérielle, la modification du circuit n'implique pas la dépense dans un matériel supplémentaire, car il y aura seulement une exploitation des ressources internes à l'FPGA. Ajouter à cela, les autres atouts du circuit FPGA restent au profit de l'instrumentation du capteur. En particulier la possibilité de mise à jour ou de correction du circuit sans pour autant que ça soit compromettant pour la chaine de fabrication ni le coût de reviens du capteur.

L'une des parties du capteur qui peut être optimisée en utilisant un FPGA est le port de communication. Il existe plusieurs ports de communication embarqués sur le substrat d'un capteur tel que le bus One Wire (un port de communication basé sur un bus et la masse), le ZigBee (un système de communication sans fil), le bus I2C …etc. Le bus I2C (Inter Integrated Circuit) se distingue par rapport aux autres bus par des caractéristiques particulièrement intéressantes et qui lui procure un avantage certain. C'est un port de communication multi maîtres, pouvant transmettre des données à des vitesses supérieures à 100 Kbit/s, il peut assurer l'adressage de 128 éléments.

La limite d'adressage du bus I2C peut devenir contraignante, sur tout lors ce qu'il s'agit de gérer un réseau de capteurs. En effet, si chaque capteur dispose d'une adresse unique et que le nombre de capteurs est élevé le calculateur principal devra dédier un port I2C seulement pour gérer le réseau de capteurs. Ce qui augmente l'encombrement du système, réduit l'efficacité du calculateur à gérer et à exploiter les données du réseau de capteurs et complique la maintenance du système.

Dédier un contrôleur I2C implémenté sur un FPGA pour gérer un réseau de capteurs, n'occupe qu'une seule adresse sur le bus I2C et fournie seulement les informations utiles au calculateur principal. Par conséquent, améliore l'efficacité du réseau de capteurs, n'encombre pas le bus I2C et décharge le calculateur central de certaines tâches.

Ce mémoire intitulé « Implémentation d'une instrumentation sur un FPGA » se divise en quatre chapitres organisés de la manière qui suit:

Le premier, se divise en deux parties, la première traite des circuits logiques programmables selon leur ordre chronologique d'avènement jusqu'à l'invention des FPGA. La deuxième partie, traite de l'outil indispensable pour la programmation d'un FPGA, les langages HDL (Hard Ware Description Langages) puis le travail est orienté en particulier vers le langage VHDL (Very High Speed Integrated Circuits Hardware Description Language).

Le deuxième, se divise en deux parties, la première traite des éléments fondamentaux de l'instrumentation, les différents types de capteurs et leurs circuits associés analogiques et numériques et un aperçu des capteurs intelligents. La deuxième est une étude du protocole I2C, ses caractéristiques et son fonctionnement.

Le troisième chapitre, se divise en deux parties, la première passe en revue les éléments nécessaires pour la construction d'une machine à états, l'outil indispensable en logique séquentiel, ses règles et son fonctionnement. La deuxième partie, est une construction du contrôleur I2C à travers des machines à états puis la simulation fonctionnelle du code VHDL construit à base des machines à états. Une simulation basée sur les prédictions du protocole I2C vérifié par des graphes de sorties du simulateur.

Le quatrième chapitre est l'implémentation du contrôleur I2C, précédemment simulé fonctionnel, sur un FPGA de la compagnie XILINX de la famille VIRTEX 4. Une description pas à pas du processus d'implémentation effectué par le logiciel Xilinx ISE (Integrated Software Envirenment) fournie à cet effet par le fabricant avec des aperçues graphique du circuit implémenté.

Ce mémoire se termine par une conclusion illustrant les perspectives de recherche et développement de systèmes en instrumentation ayant pour but l'amélioration des performances et l'augmentation de l'efficacité d'un système de gestion d'un réseau de capteurs. Des perspectives qui sont le fruit de la recherche bibliographique et de l'application des connaissances acquises dans le domaine de développement des systèmes électroniques.

Chapitre I :

Les circuits logiques programmables et le VHDL.

I.1 Les circuits logiques programmables :

1.1 Introduction :

Les circuits logiques programmables représentent une solution incontournable dans le domaine de l'électronique numérique. En effet, la possibilité de programmer un composant pour qu'il puisse fonctionner selon les besoins du concepteur est une aide précieuse pour pouvoir élaborer efficacement un circuit complexe avec des délais court et un coût de reviens faible. Contrairement aux circuits logiques prédéfini par les fabricants tels que les portes logiques (la famille 4000…) ou les fonctions logiques (les contrôleurs de bus …) qui ont des entrées sorties fixe une architecture figée un encombrement étendu (quelques portes logiques dans un boîtier volumineux) une consommation en énergie élevée réduit l'usage de ces composants pour de petites applications simple et des circuits à densités moyenne. Les PLDs (Programmable Logic Device en Anglais) offrent la possibilité de programmer l'architecture interne du composant, pour réaliser une fonction souhaitée par l'utilisateur et de configurer les pins d'entrés/sorties de celui-ci. En distinction par rapport aux microprocesseurs ou microcontrôleurs qui sont, eux aussi programmables mais pour qui l'on peut seulement programmer le fonctionnement selon un programme existant dans une mémoire, l'architecture interne est proposée par le fabricant tout comme les entrés/sorties [*TAV96*].

Les PLDs ont connu une évolution technologique au fil du temps depuis la parution du premier PAL (Programmable Array Logique ou réseau logique programmable) jusqu'à l'aboutissement aux FPGAs (Field Programmable Gate Array ou réseau de cellules logiques programmables) qui sont les circuits logiques programmables les plus performant qui existent en ce moment, une évolution dû à une concurrence industrielle et scientifique [*TAV96*]. Dans ce qui suit, il sera présenté les différentes familles de circuits logiques programmables.

1.2 Définition des PLDs (Programmable Logic Device):

Un circuit programmable est un assemblage d'opérateurs combinatoires (les opérateurs combinatoires génériques qui interviennent dans les circuits programmables proviennent soit des mémoires (réseaux logiques) soit des fonctions standard (multiplexeurs et OU exclusif)) et de bascules dans lequel la fonction réalisée n'est pas fixée lors de la fabrication. Il contient potentiellement la possibilité de réaliser toute une classe de fonctions, plus ou moins large

suivant son architecture. La programmation du circuit consiste à définir une fonction parmi toutes celles qui sont potentiellement réalisables. Comme dans toute réalisation en logique câblée, une fonction logique est définie par les interconnexions entre des opérateurs combinatoires et des bascules, et par les équations des opérateurs combinatoires. Ce qui est programmable dans un circuit concerne donc les interconnexions et les opérateurs combinatoires [*JAC00*].

1.3 Structure de base d'un PLD :

La plupart des circuits PLDs suivent la structure suivante [*LAU97*] :

- Un bloc d'entrées qui permet de fournir au bloc combinatoire l'état de chaque entrée et de son complément.
- Un ensemble d'opérateur « ET » sur lesquels viennent se connecter les variables d'entrées et leurs compléments.
- Un ensemble d'opérateurs « OU » sur lesquels les sorties des opérateurs « ET » sont connectées.
- Un bloc de sorties.
- Un bloc d'entrées sorties.
- Un bloc d'entrées/sorites, qui comporte une porte 3 états et une broche d'entée/sortie.

Le bloc combinatoire programmable est formé de matrices « ET » et de matrices « OU » que l'on appelle aussi somme de produits (toute fonction logique combinatoire peut être écrite comme somme de produit) [*STE05*], les interconnexions de ces matrices doivent être programmables, ceci est réalisé par des fusibles qui sont grillés lors de la programmation. L'autre approche, radicalement opposée, « cellule universelles interconnectées » est de renoncer à la réduction en première forme normale des équations logiques. On divise le circuit en blocs logiques indépendants, interconnectés par des chemins de routage. Une fonction logique est récursivement décomposée en opérateurs simples, jusqu'à ce que les opérations élémentaires rentrent dans une cellule [*JAC00*].

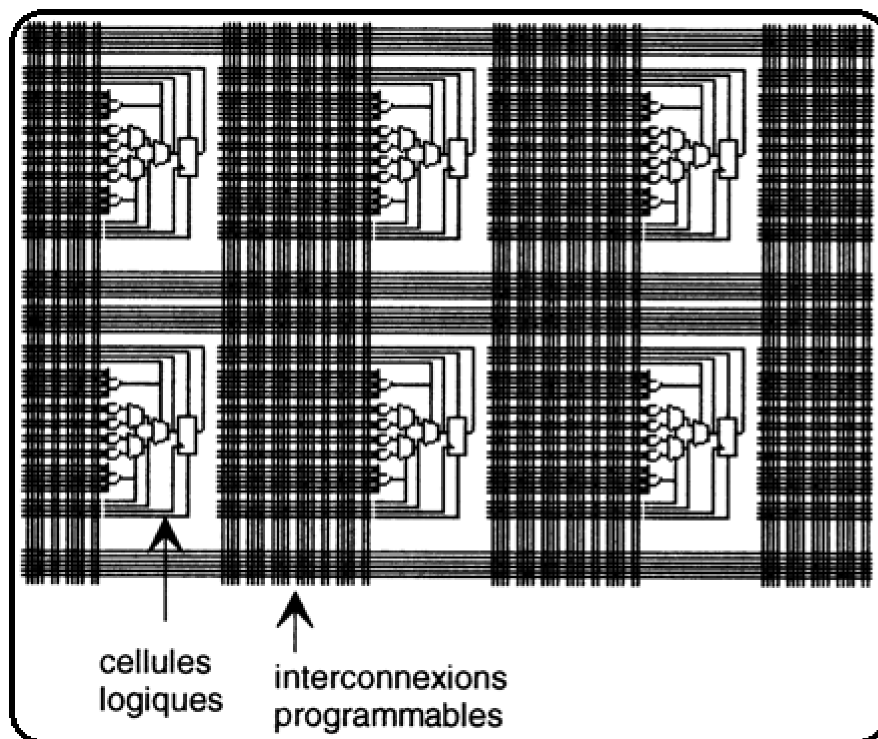


Figure I.1 : cellules logiques interconnectées.

Le bloc de sortie est souvent appelé macro cellule, OLMC[1] (Output logic macro cell, macro cellule logique de sortie). La macro cellule fut l'élément clé dans le développement des circuits logiques programmables. En effet, la macro cellule procure au composant la flexibilité de configuration : entré, sortie, entré/sortie ou haute impédance. Plus le composant est performant plus celui-ci présente des options sur sa macro cellule [*TAV96*]. Celle-ci comporte :

- Une porte « OU » exclusif, une bascule D.
- Des multiplexeurs, qui permettent de définir différentes configurations et un dispositif de re-bouclage sur la matrice « ET ».
- Des fusibles de configuration (pour les FPGAs, il est utilisé des cellules de commande des ponts de connexions).

Placement et routage : consiste à attacher des blocs de calcul aux opérateurs logiques d'une fonction et à choisir les broches d'entrées/sorties. Le routage consiste à créer les interconnexions nécessaires. Pour les PLDs simples, le placement est relativement trivial et le routage inexistant. Les compilateurs génériques (i.e. indépendants du fondeur) effectuent très

[1] OLMC : Output logic macro cell, macro cellule logique de sortie, est un nom proposé par LATTICE pour faire allusion à la macro cellule. Ce même élément, a plusieurs appellations selon le fabricant.

bien ces deux opérations. Pour les CPLDs, et plus encore les FPGAs, ces deux opérations deviennent plus complexes et nécessitent un outil spécifique du fondeur [*JAC00*].

1.4 Les différentes familles de PLDs :

La classification des PLDs peut se révéler délicate et difficile, les différences de technologie se doublent de différences d'architectures. La classification suivante n'a que pour objectif de mettre en lumière de grands points de repère[2]. Néanmoins, on peut les classer suivant leurs structures internes à savoir : le nombre d'entrées, de sorties, de connexions programmables et le niveau d'intégration [*TAV96*].

Type	Nombre de porte intégré	Matrice ET	Matrice OU	Effaçable
PAL	10 à 100	Programmable	Fixe	Non
GAL	10 à 100	Programmable	Fixe	Électriquement
EPLD	100 à 3000	Programmable	Fixe	Par UV
FPLA	2000 à 3000	Programmable	Programmable	Électriquement
FPGA	Plus de 50 000	Programmable	Programmable	

1.4.1 Les PALs (Programmable Array Logique):

Les PALs ont eu un grand succès dès leur première parution sur le marché, ce fut les premiers circuits logiques programmables[3].

Un PAL est un composant relativement simple, dérivé des PROM (Programmable Read Only Mémoire, mémoire morte à lecture seule programmable une fois). Les ingénieurs de MMI ont combiné la technologie à fusibles (utilisée pour les mémoires PROM) avec des portes ET, OU pour réaliser des fonctions logiques. La compréhension de la cellule de base d'un PAL suffit car c'est la même qui se répète sur l'étendue de la capacité de celui-ci (voir la Figure I.4). La cellule de base se compose d'un buffer d'entrée qui dispose de l'information et de son complément (tous les PALs sans exception disposent d'un certain nombre d'entrées qui aboutissent toutes, sous forme directe et inversé, sur la matrice de fusible de programmation), suivit de la matrice à fusibles puis de portes ET (considérées en entrée) puis suivit de portes OU en sortie [*TAV96*].

[2] Cette classification n'est pas exhaustive et l'omission de certains éléments tels que les CPLDs n'est pas une lacune. En effet, les fabricants adoptent par fois des appellations différentes pour des circuits équivalant ce qui est dû en partie aux droits d'auteur, PAL est une marque déposée par MMI par exemple.

[3] Proposé par MMI, absorbé par AMD.

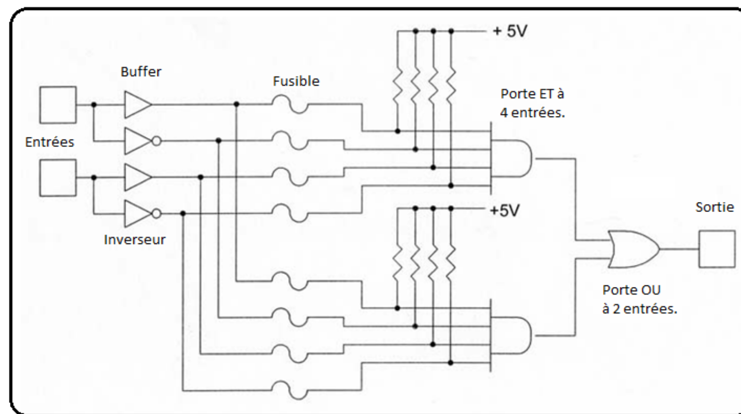


Figure I.2 : Cellule de base d'un PAL (architecture combinatoire).

La programmation d'un PAL s'effectue par la destruction de fusibles, à l'aide d'un programmateur dédié en appliquant des tensions de programmation requises. L'inconvenant majeur des PALs, c'est qu'une fois programmée, ils ne sont plus effaçables (fusible détruit) c'est contraignant en cas d'erreur de programmation ou de mise à jour.
Les PALs existent sous 4 d'architectures (dans l'ordre de leur évolution) [*LAU97*] :

1. **Les PALs combinatoires :** possède l'architecture la plus simple. Comme dans tout type de PALs, certaines broches sont dédiées uniquement aux entrées. D'autres bidirectionnelles, sont associés à un buffer de sortie trois états.
2. **Les PALs à registres :** ils disposent en sortie d'une bascule D. Tout signal de sortie passe obligatoirement par cette bascule. Pour cette raison, certains boîtiers sont proposés avec un certain nombre de sorties à registre (synchrone) et d'autres de type combinatoire.
3. **Les PALs asynchrones à registres :** Ce type de PALs constitue une variante du type évoqué précédemment. Une première différence réside dans la méthode de distribution du signal d'horloge des bascules. Contrairement aux PALs à registres, un multiplexeur permet de shunter le registre. Les deux entrées AP (PRESET Asynchrone) et AR (RESET Asynchrone) sont simultanément utilisées pour piloter ce multiplexeur. A noter la présence d'un XOR se comportant comme un inverseur programmable. Le signal de commande de ce XOR a une valeur statique, il ne s'agit ni d'un signal global ni d'un signal issu de la matrice. C'est à partir de là que le concept de la « macro cellule » programmable a fait son apparition. Cette ressource d'inversion permet de coder une fonction sans tenir compte de la polarité du signal de sortie.
4. **Les PALs versatiles (VPAL) :** ils constituent une évolution des PALs très significative. Ils proposent des macros cellules très évoluées, du type de cellules

rencontrées au sein de composants plus complexes (CPLD, FPGA). Le mode de fonctionnement est obtenu par l'utilisation de deux multiplexeurs qui utilisent comme signal de sélection les points de programmation S0 et S1. En combinaison de ces deux points de programmation, on obtient différentes configurations de la sortie.

1.4.2 Les GALs (Generique Array Logic):

Les GALs ne sont rien d'autre (d'un point de vu architectural) que des PALs reprogrammables. D'un point de vu technologique au lieu d'utiliser des transistors bipolaires, ils ont utilisé des transistors MOS FET pouvant être régénérés. Cette possibilité de régénération des fusibles sans pour autant restreindre la durée de vie du composant.

Après étude des besoins du marché LATTICE SEMICONDUCTOR se fixe 4 objectifs pour la mise au point des GALs :

1. Offrir des produits ayant des vitesses de travail comparable a celle des PALs bipolaires, tout en étant testable a 100%.
2. Permettre un remplacement, au moins fonctionnel, mais idéalement broche pour broche, des PALs bipolaires dans n'importe quelle application.
3. Offrir une consommation beaucoup plus faible que les PALs bipolaires d'une complexité équivalente.
4. Proposer une plus grande souplesse de configuration des entrées/ sorties que les PALs bipolaires.

C'est ainsi que LATTICE a palier aux inconvenants majeurs des PALs pour donner naissance à un composant lui imposant une forte concurrence [*TAV96*].

1.4.3 Les EPLDs (Erasable Programmable Logique Device):

Les EPLDs sont des circuits réalisés en technologie CMOS, présentant l'avantage de faible consommation électrique, mais qui augmente en fonction de la fréquence. Ils disposent d'une macro cellule plus évolué que celles des PALs, en plus ils sont effaçables [*TAV96*].

L'introduction des EPLDs telle que l'a voulu ALTERA visait deux buts distincts:

1. Permettre une densité d'intégration nettement supérieur à celle offerte par les PALs et aussi proche que possible que celle permise par les réseaux de portes programmable.
2. Fonctionner a une vitesse, si non égal, du moins comparable à celle des PAL bipolaires et en tout cas nettement supérieur à celle des portes traditionnels.

Les différentes familles d'EPLDs :

1. **Les EPLDs classique de la série EP**. Ces circuits existent en différentes versions avec des boîtiers disposant de 20 à 68 pins. Effaçable à l'UV.
2. **Les EPLDs de la série MAX 5000**, qui ont la même fonction que leur homologue de la série EP, avec toutefois une densité d'intégration plus élevée et une architecture d'interconnexion différente, effaçable à l'UV.
3. **La série MAX 7000** plus dense que MAX 5000 en plus effaçable électriquement.
4. **La série MAX 9000** plus dense, effaçable électriquement en plus programmable sur circuit avec la tension unique de 5v.

1.4.4 Les CPLDs :

L'architecture typique d'un CPLD se présente comme un ensemble de fonctions de type PAL pouvant être interconnectées à l'aide d'une matrice. La physionomie est généralement très structurée. Un certain nombre de macros cellules de base sont regroupées pour former des blocs logiques. La complexité, le nombre de macros cellules dans un bloc ainsi le nombre de blocs varie d'un composant à l'autre. On peut considérer deux niveaux d'interconnexion : une matrice globale et un système de distribution des signaux intégrés à chaque bloc logique [*LAU97*].

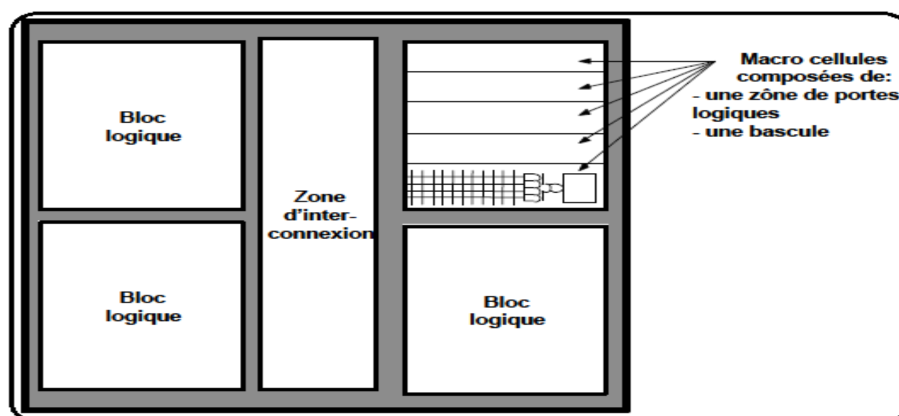


Figure I.3 Architecture global d'un CPLD.

I.4.5 Les FPGA Field Programmable Gate Array ou réseau logique programmable sur site :

Un FPGA est un circuit logique reprogrammable. À l'aide de blocs logiques préconstruits et de ressources de routage programmables, c'est un circuit configurable afin de mettre en œuvre des fonctionnalités matérielles personnalisées, sans avoir jamais besoin d'utiliser une maquette ou un fer à souder. Il suffit de développer des tâches de traitement numérique par

logiciel et de les compiler sous forme de fichier de configuration ou de flux de bits (bit stream) contenant des informations sur la manière dont les composants doivent être reliés. En outre, les FPGAs sont totalement re-configurables et peuvent adopter instantanément une nouvelle circuiterie si une nouvelle configuration du circuit est recompilée.

1.5 Étude des FPGAs :

1.5.1 Présentation et généralités sur les FPGAs :

Un FPGA est un composant électronique constitué de millions voir milliers de transistors connectés ensembles pour réaliser des fonctions logiques. Des fonctions logiques simples peuvent être réalisées telles que des additions ou soustractions tout comme des fonctions complexes peuvent être réalisées telles que le filtrage numérique du signal ou détection d'erreurs et correction. Aérospatial, aviation, automobile, radar, missiles, ordinateur… ne sont que quelques exemples des domaines ayant recours aux FPGAs.

L'évolution des circuits logiques programmable, depuis la création des PALs qui présentaient l'avantage de réduire l'encombrement et de créer des fonctions logiques personnalisé. Puis vient l'étape des EPLDs qui présentent l'avantage de l'écriture électrique mais en ayant recours à un programmateur (un appareil qui permet d'injecter le routage du circuit via un fichier de programmation) mais effaçable à l'UV (ultra violet) pour évoluer vers les CPLDs qui sont effaçable électriquement. Puis l'avènement des FPGAs qui représentent une technologie qui permet de reprogrammer le circuit in situ (c'est-à-dire sur circuit ou sur site). L'avantage majeur que présentent les FPGAs est leur grande flexibilité. En effet, la structure interne du circuit FPGA peut être changée sans avoir à modifier la structure globale de la maquette. Cet avantage est très apprécié par les concepteurs de cartes électroniques vu que ça leurs permet de faire des prototypage rapide et de moindre coût en comparaison aux ASICs pour les quels il faut des mois pour réaliser un prototype sans avoir de certitude qu'il puisse être opérationnel en plus de ça la moindre erreur nécessite de refaire le travail depuis le début. Un coût de reviens important et une durée de développement étendu ce qui doit être minimisé dans les milieux industriels. Il y aussi l'avantage de la mise à jour du circuit face à des bugs ou l'ajout de nouvelles fonctionnalités [*GIN10*].

Xilinx, Altera et Quicklogic sont les pionniers dans le domaine des FPGAs, et plusieurs autres compagnies produisent les FPGAs. Toutes ces compagnies se partagent le même concept

architectural[4]. [*GIN10*]. Il se divise en trois parties : Interfaces d'entrés/Sorties (I/O interface), les blocs logiques de base (Basic Logic Building Blocks) et les interconnexions (voir figure I.4).

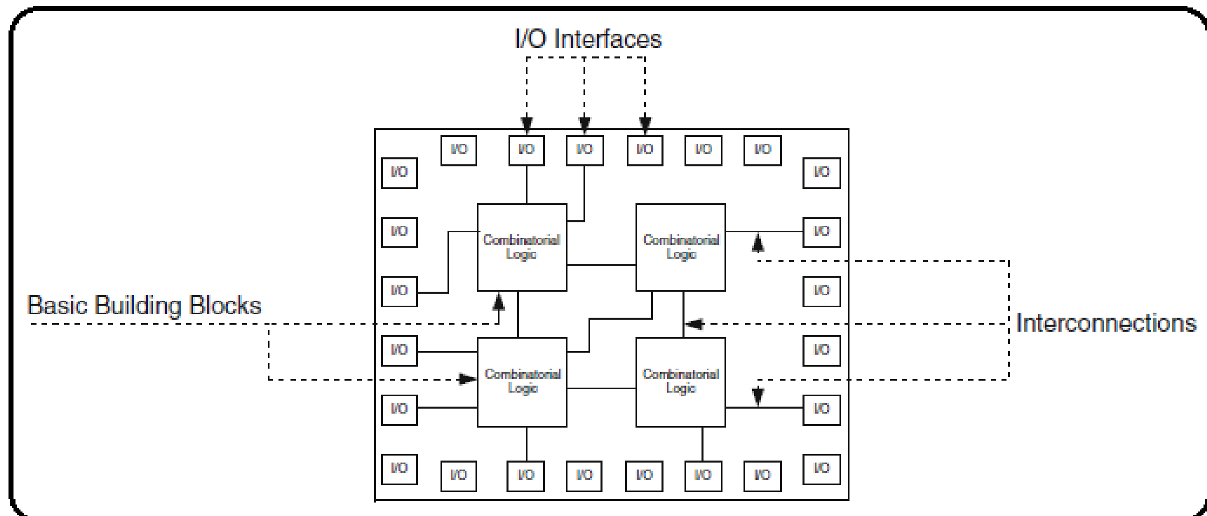


FIGURE 1.4 : Concept architectural de base des FPGAS [*GIN10*][*TAV96*].

1.5.1.1 Interfaces d'entrés/Sorties (I/O interfaces)[5] :

Les interfaces d'entrés/Sorties se présentent comme les intermédiaires par lesquelles les données transitent depuis les blocs logiques internes jusqu'aux ressources externes et vice versa. L'interfaçage des signaux peut être : unidirectionnel, bidirectionnel, à deux états ou trois états (0, 1 ou haute impédance) voir même obéir à un standard d'entrés/sorties. Voici quelques-uns de ces standards :

- GTL (gunning transceiver logic).
- HSTL (high-speed transceiver logic).
- LVCMOS (low-voltage CMOS).
- LVTTL (low-voltage transistor-transistor logic).
- PCI (peripheral component interconnect)…

[4] Nuance, car ici il est fait référence au concept architectural non à l'architecture. Chaque fabricant dispose de sa propre architecture en fonction de la technologie microélectronique qu'il maîtrise et essaye d'imposer sa propre appellation mais le concept architectural reste le même. Par abus du langage, il est fait référence à ce dernier par architecture de base.

[5] Les interfaces d'entrés/sorties représentent les macros cellules précédemment introduites.

Le rôle principal des interfaces d’entrés/sorties est de transmettre et de recevoir des données. Néanmoins l’interface d’entré/sorties peut être dotée d’options telles que des registres, impédances et buffers [*GIN10*].

Chaque fabricant a sa propre appellation pour désigner l’interface d’entrés/sorties mais la fonction reste toujours la même.

Altera, les nomme IOE Input Output Element. L’IOE remplie toujours son rôle d’interface d’entrés/sorties, elle dispose d’une résistance de rappel pull-up et un temporisateur du signal. La Figure 1.5 est un exemple d’IOE de la série CycloneII.

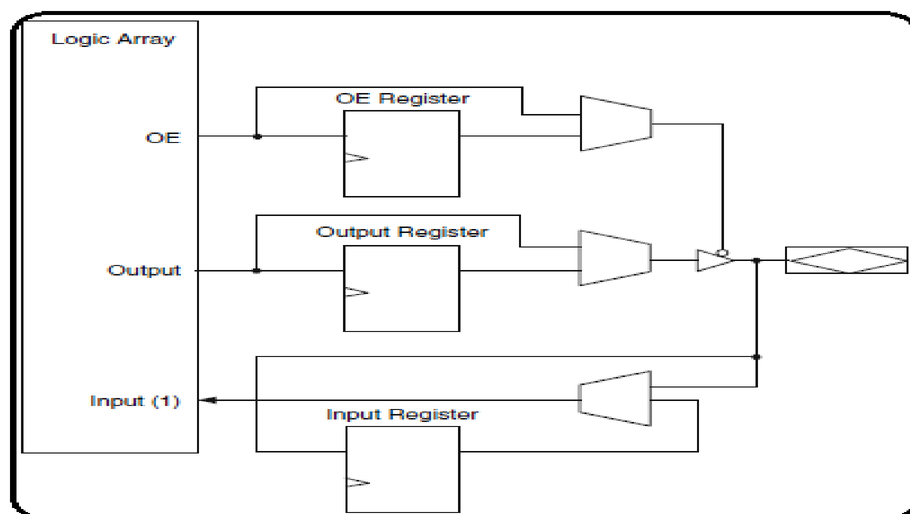


Figure 1.5 : structure interne d’IOE de la série Cyclone II [**ALT11**].

Chez Xilinx, les interfaces d’entrés/sorties sont nommées IOB pour Input Output Blocks. L’IOB est constitué de registres, de diviseurs de tensions, des résistances de rappel pull up et autres ressources spécifiques. La Figure 1.7 est un exemple d’IOB simplifié de Xilinx.

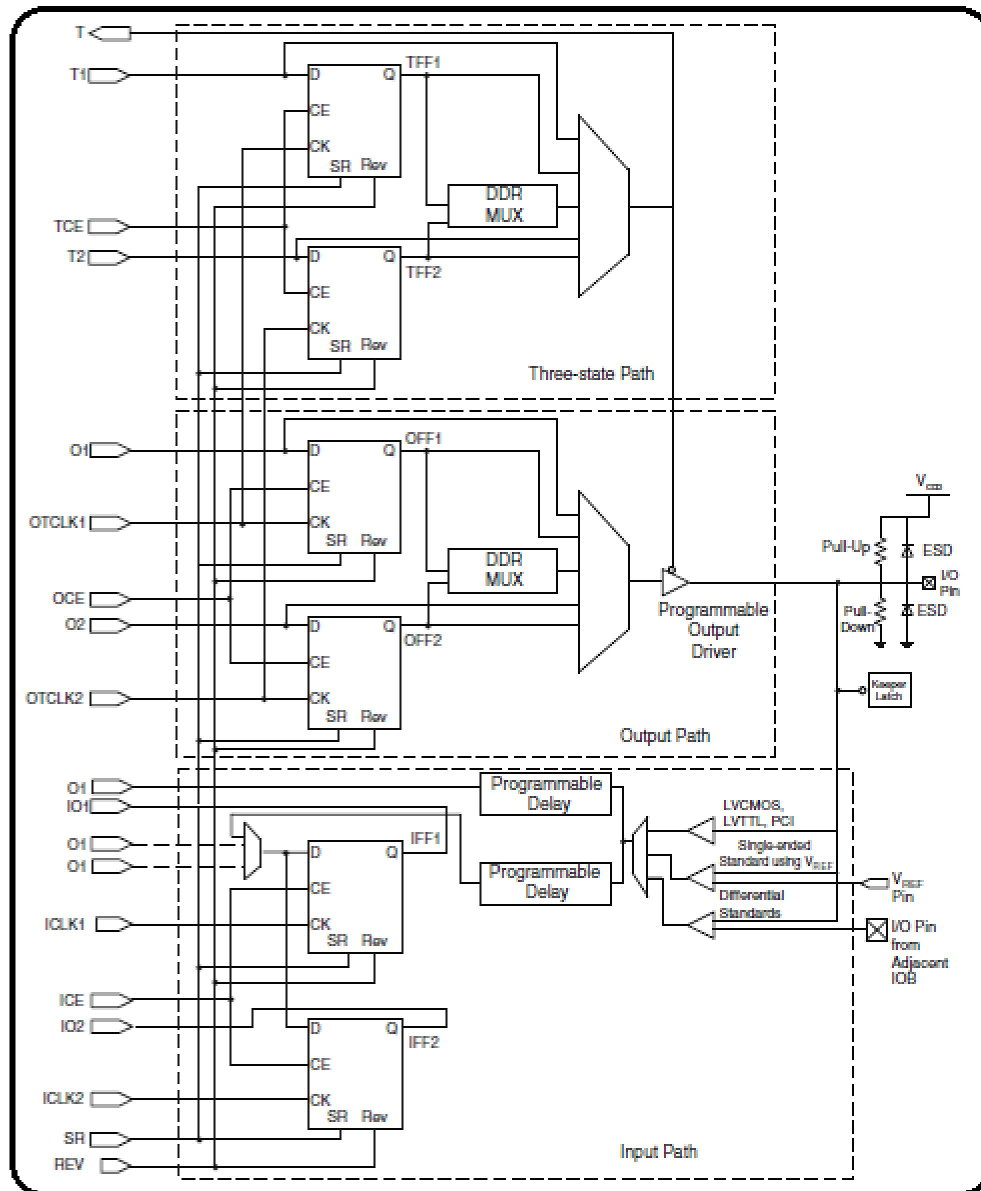


Figure 1.6 : structure simplifiée d'un IOB** [XIL11]**.

1.5.1.2 Les blocs logiques de base (basic logic building blocks) :

Les blocs logiques de base, sont des blocs logiques configurés ou optimisés de sorte à réaliser des fonctions logiques.

Altera, nomme les blocs logiques de base ALM Adaptative Logique Module. L'ALM est constitué d'éléments de logique combinatoire, des registres et des additionneurs. L'élément de la logique combinatoire est constitué de 8 entrés et d'un LUT Look Up Table [*GIN10*]. Voir la figure 1.7.

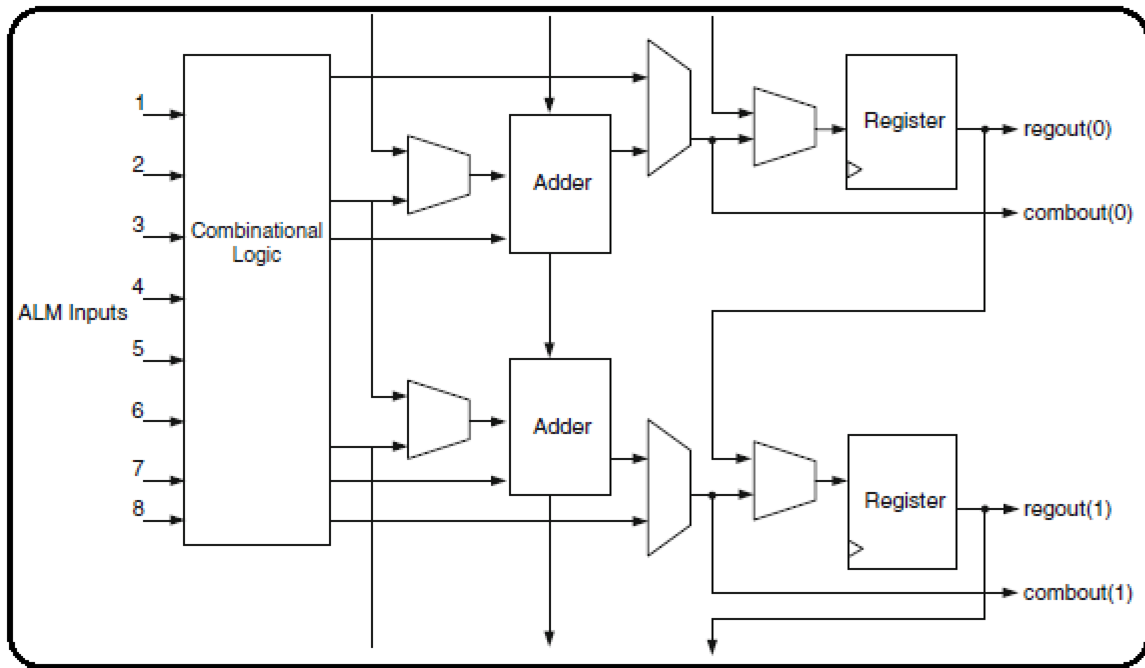


Figure 1.7 : Constitution interne d'un ALM Adaptative Logic Module [ALT11].

Xilinx, nomme les blocs logiques de base CLB Configurable Logic Blocks. Chaque CLB contient des slices et chaque slice a des LUTs Lookup Tables. Voir la figure 1.8.

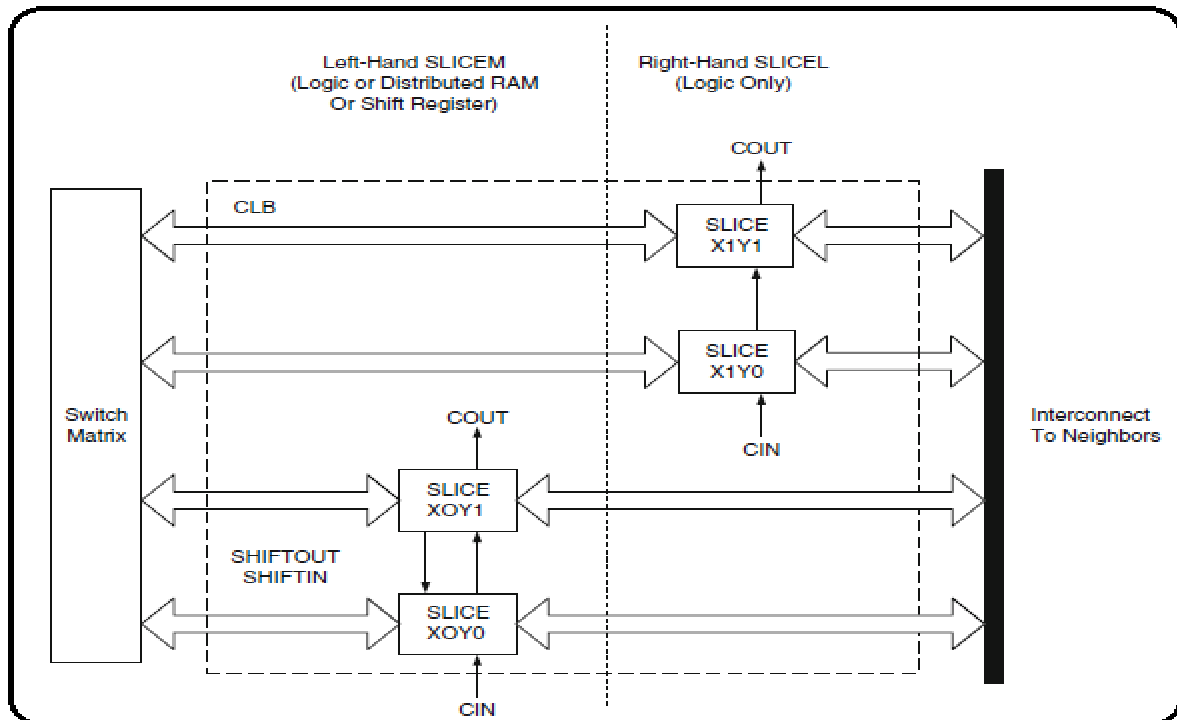


Figure 1.8 : Constitution d'un CLB configurable logic block et la disposition des slices (famille Virtex) [XIL11] .

Chaque fabricant défini ses propres structure des blocs logiques de base et le nombre de blocs disponibles. Ceci est dû au fait que les FPGAs sont utilisés dans des applications variées et il

n'y a pas un seul composant FPGA pour toutes les applications. Ceux qui sont utilisés en aérospatial ne sont pas prédestinés à réaliser les mêmes fonctions ni à opérer dans les mêmes conditions que ceux utilisés en automobile ou bien en téléphonie mobile [*GIN10*].

Le choix du bon FPGA est simplifié par la classification adoptée par les fabricants, Xilinx s'est organisé en familles au moment où Altéra s'est organisé en séries. Chaque famille ou série est constituée de membres ou sous familles. Les membres d'une même famille se partagent les mêmes caractéristiques de base et chaque membre se distingue des autres par ses propres caractéristiques à savoir la mémoire, les ressources disponibles ou le nombre d'entrés/sorties.

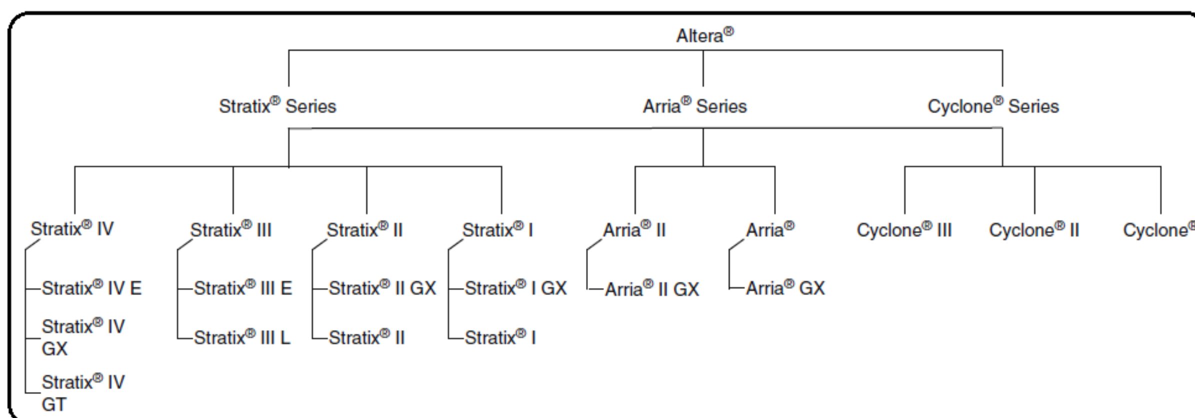


Figure1.9 : l'arbre des séries chez Altera, Stratix haut de gamme et haute densité, Arria moyenne gamme et Cyclone low cost et faible consommation électrique.

Xilinx présente plusieurs familles de FPGAs dont voici quelques-unes : Extended SpartanW-3A, Spartan-3E, Spartan 6, Virtex, Virtex-E, Virtex-Pro, Virtex 5, et Virtex 6. Par exemple Spartan-3E est une famille qui comporte 5 membres (voir le tableau suivant) le numéro qui suit le S dans la référence, représente 1 pour 1000 portes. Ce qui est pratique, d'un simple regard sur la référence le nombre de portes que le composant comporte est déduit [*GIN10*].

Device	System Gates	Total CLBs	Total Slices	Max User I/O	Max Differential I/O Pairs
XC3S100E	100K	240	960	108	40
XC3S250E	250K	612	2,448	172	68
XC3S500E	500K	1164	4,656	232	92
XC3S1200E	1200K	2168	8,672	304	124
XC3S1600E	1600K	3688	14,752	376	156

1.5.1.3 Les Interconnexions :

Les interconnexions sont très importantes car ce sont elles qui transmettent le signal d'un point à un autre. Il existe plusieurs types de lignes [*TAV96*]:

1. **Les interconnexions à usage général :** sont composées de segments verticaux et horizontaux qui entourent chaque bloc logique de base et qui peuvent être reliés entre eux par une matrice de communication.
2. **Les lignes directes :** fournissant des chemins entre les blocs logiques de base adjacents et entre les blocs logiques de base et les interfaces d'entrés/sorties.
3. **Les lignes longues :** qui sont des lignes verticales et horizontales qui n'utilisent pas de matrices de communication. Elles parcourent toutes les zones d'interconnexion. Elles sont utilisé pour véhiculer les signaux qui doivent parcourir de long trajet. Ces lignes conviennent pour véhiculer les signaux d'horloge.

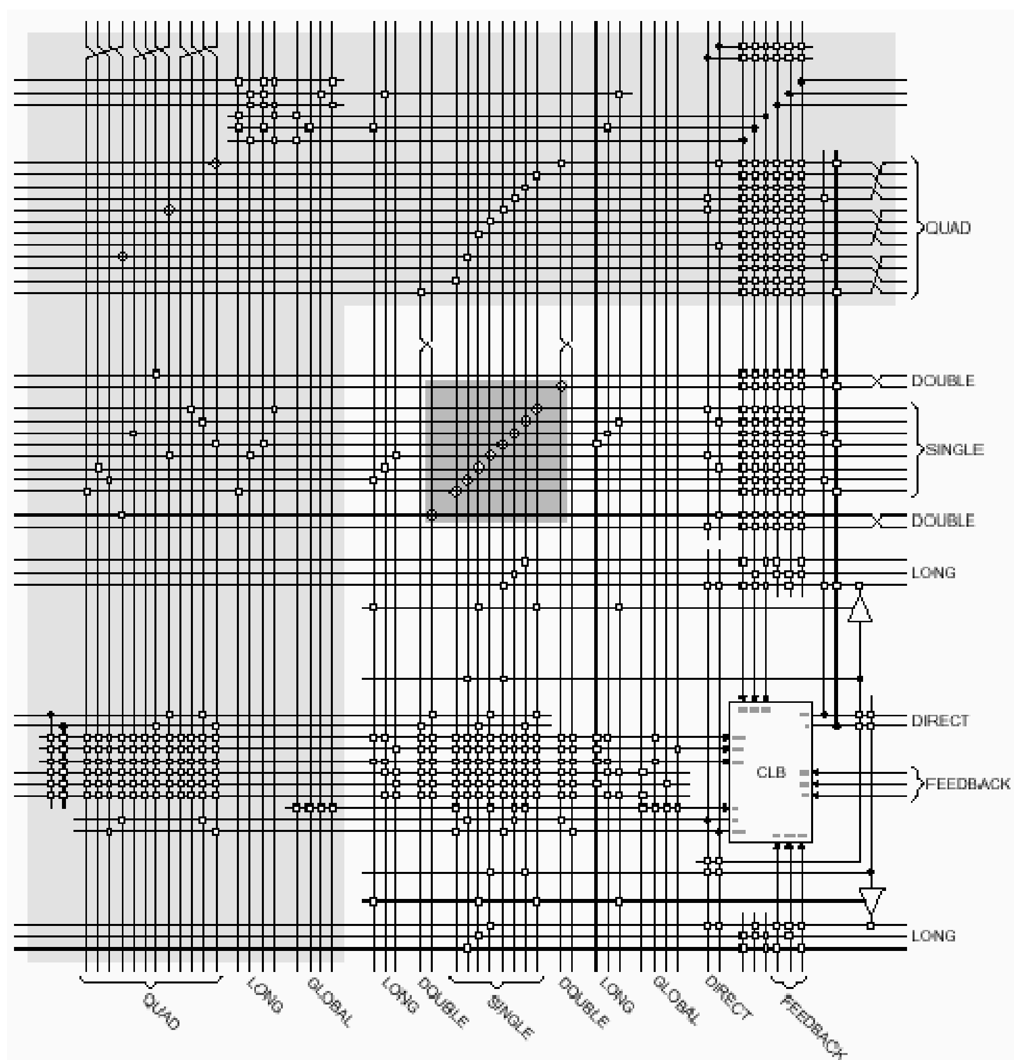


Figure 1.10 : Illustration des différents types d'interconnexions chez Xilinx.

En plus de la diversité architecturale il existe deux types de FPGAs [*TAV96*] :

1. Les FPGAs de type SRAM, appelé LCA chez XILINX, FLEX chez ALTERA.
2. Les FPGAs à antifusibles proposé entre autre, par Texas Instrument et ACTEL.

Ces deux technologies différent essentiellement dans la façon dont leurs interconnections sont réalisées [*TAV96*] [*LAU97*].

1. **Les FPGAs de type SRAM :** Introduit dès 1985 par Xilinx, la configuration du composant réside en deux points : la définition des interconnexions définies par des cellules SRAM et les fonctions booléennes définies dans des mémoires de type SRAM. La mémorisation de la configuration est donc volatile. Ces boîtiers sont programmés après chaque mise sous tension. Les données de programmation sont issues soit directement d'une ROM (mémoire à lecture seul), soit d'un système intelligent (microprocesseur ou microcontrôleur).
2. **Les FPGAs à antifusibles** : Une technologie apparu en 1990 et elle est différente des autres FPGAs car ceux là ne sont pas reprogrammables, voir même l'architecture interne qui diffère. Ils sont proposés pour l'essentiel par Texas Instrument.

   L'architecture globale de ces FPGAs est analogue à celle des LCA, avec un certain nombre de blocs d'entrées/sorties répartis tout autour de la puce. Des blocs logiques placés en matrice au centre de celle ci et des lignes d'interconnexions, mais la similitude s'arrête là. En effet, alors que les LCAs faisaientt appel à des cellules logiques de base relativement complexes, les CLB, des FPGAs à anti-fusibles utilisent des cellules très simples. Comme référence, la famille TCP 10XX et TCP 12XX de TEXAS INSTRUMENT.

1.6 Etude de la Virtex 4:

La famille Virtex-4 de Xilinx est un circuit logique programmable très flexible proposé comme alternative aux ASICs. Proposé sous trois familles : LX, logique haute performance ; FX, traitement de signal très haute performance et SX, pour les traitements embarqués et connectivité rapide. Elle a été réalisée selon la technologie 90 nm [*DS112*].

1.6.1 Interfaces d'entrés/sorties IOB (Input Output Bloc) :

Toute la famille Virtex-4 dispose de la technologie d'interface entés/sorties haute performances et peut supporter une grande variété de standards d'interfaces. Elle dispose d'un contrôle des sorties et d'un contrôle numérique d'impédance (Digitally Controlled Impedance DCI) fonctionnant sous 3.3V. Chaque IOB (input, output bloc) dispose de l'entré, de la sortie et de la sélection de la sortie en trois états.

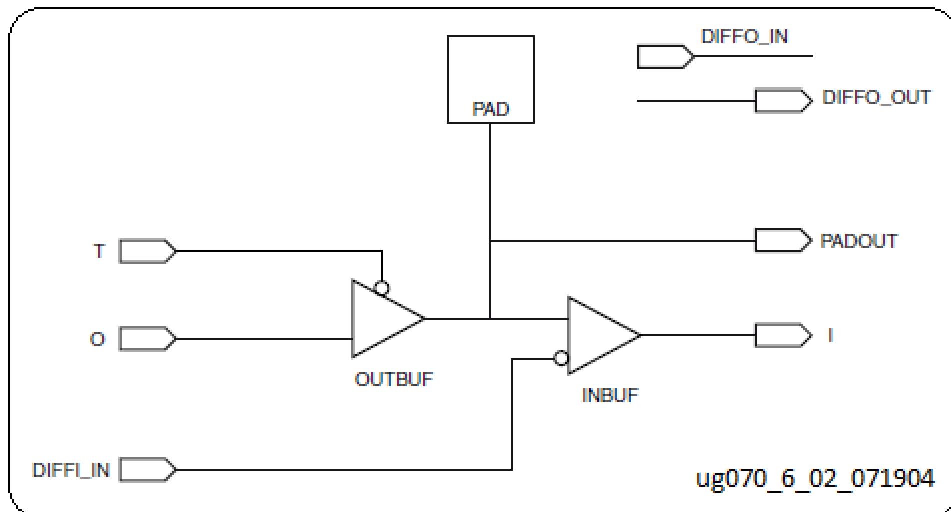


Figure 1.11 : diagramme de base d'entré/sortie (IOB).

1.6.2 Les blocs logiques de base de la Virtex-4, Configurable Logic Block (CLB) :

Les CLBs représentent la ressource principale à implémenter pour la logique séquentielle tout comme pour la logique combinatoire. Chaque CLB est connecté à la matrice d'interconnexion (voir Figure 1.8 paragraphe 1.5.1.2). Chaque CLB est constitué de quatre SLICEs interconnectées. Ces slices sont disposées en paires et chaque paire est disposé en colonne. La paire de gauche est appelée SLICEM et celle de droite SLICEL.

Les éléments communs à ces deux paires de slices (SLICEM et SLICEL) sont deux générateurs de fonctions appelés LUT (Look Up Table) deux éléments d'enregistrement, des multiplexeurs, et des éléments de logique arithmétique (voir Figure 1.12). [*UG070*]. Le tableau suivant résume les ressources logiques disponibles dans un CLB (Virtex-4).

Slices	LUTs	Flip-Flops	MULT_ANDs	Arithmetic & Carry Chains	Distributed RAM(1)	Shift Registers(1)
4	8	8	8	2	64 bits	64 bits

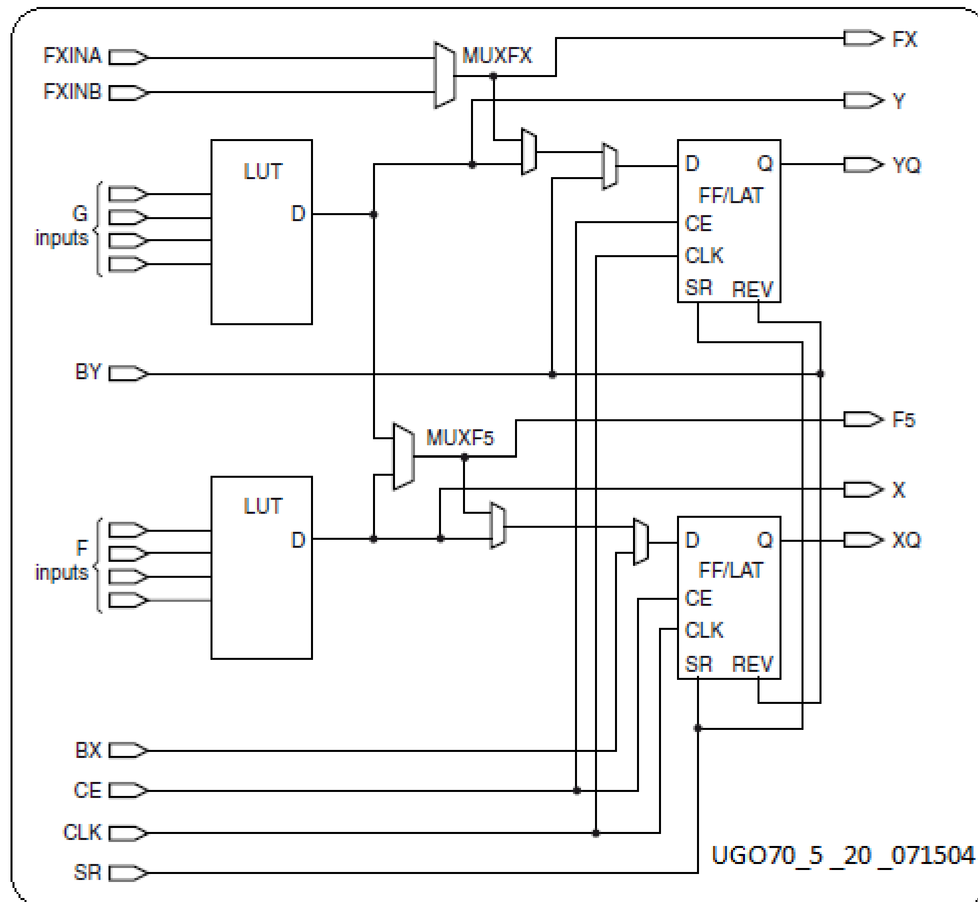


Figure I.12 : Version simplifiée (certaines connexions ne sont pas représenté pour une meilleure lisibilité) d'une SLICE d'une Virtex-4.

Les générateurs de fonctions LUT : Les générateurs de fonctions sont des LUTs à 4 entrées. Il existe 4 entrées indépendantes pour chaque générateur de fonctions d'une SLICE F et G. le générateur de fonctions admet l'implémentation de l'importe quelle fonction booléenne à 4 entrées et le temps de propagation du signal est indépendant de la fonction implémenté.

Éléments d'enregistrement (bascule D) : l'élément d'enregistrement est une bascule D configurable. L'entrée de cette bascule peut être commandé directement par la sortie de la LUT ou via un multiplexeur. Cette bascule D est paramétrée par : le contrôle du signal d'horloge CLK (Contrôle Signal Clock), activation d'horloge CE (Clock Enable) et Set Reset SR. Ces paramètres sont communs à toutes les bascules D d'une SLICE.

Les multiplexeurs : En associant les générateurs de fonctions LUTs avec les multiplexeurs on peut réaliser :

Multiplexeur 4 Vers 1 dans une SLICE ; Multiplexeur 8 Vers 1 dans 2 SLICES ;

Multiplexeur 16 Vers 1 dans 1 CLB ; multiplexeur 32 Vers 1 dans 2 CLBs.

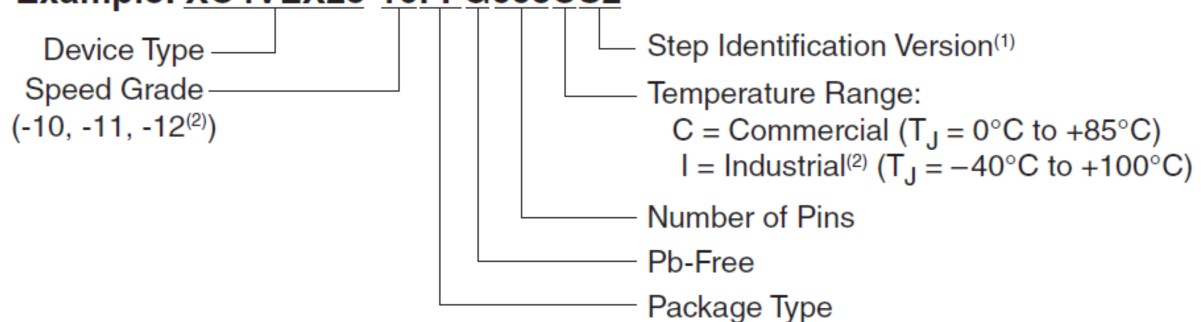
Les éléments de logique arithmétique : Les éléments de logique arithmétique incluant une porte XOR qui permet de réaliser un additionneur complet à deux bits au sein d'une même SLICE. En plus, il existe une porte AND pour simplifier l'implémentation d'un multiplieur.

1.6.3 Les interconnexions :

Toute la famille Virtex-4 utilise le même schéma d'interconnexion et ils ont le même accès à la matrice globale d'interconnexion. La distribution de l'horloge est partagé de sorte à améliorer les performances des circuits à grande vitesse.

1.6.4 Nomenclature des Virtex-4 :

Voici un exemple de référence Virtex-4 et le détail de sa nomenclature [*DS112*] :



I.2. Les langages de description matérielle HDL (Hardware Description Langages) :

2.1 Évolution des HDL :

La description matérielle a connu une évolution au fur et à mesure que les moyens informatiques et les technologies d'intégrations des transistors (de plus en plus denses) évoluaient. Les langages de description Matérielle ont connu une évolution en quatre étapes pour arriver à maturité :

Première étape : dessin au micron.

Au début, la conception de circuits intégrés était manuelle. On dessinait les composants (transistors) à la main, sur un papier spécial (Mylar) avec des crayons de couleur. C'est le dessin au micron. Une telle technique limitait la complexité des dispositifs conçus.

Deuxième étape : les langages de description.

Plus la technologie microélectronique évoluait vers l'intégration de nombres plus important de transistors, plus la nécessité de nouveaux outils de conception s'imposait c'est ainsi que les langages de description matériel (HDL) ont fait leurs apparitions. Ils avaient pour but de modéliser, donc de simuler, mais aussi de concevoir. Des outils informatiques (placeurs-routeurs) permettant de traduire automatiquement une description textuelle en dessins. Le concepteur aborde les problèmes à un niveau d'abstraction plus élevé. Il manipule des objets élémentaires de l'ordre de la dizaine de transistors : les portes logiques.

Troisième étape : les schémas.

L'apparition des interfaces graphiques et donc des éditeurs de schémas simplifie le travail de conception de circuits intégrés. En effet : il est en général beaucoup plus facile de lire et de comprendre un schéma qu'une description textuelle. La description textuelle est remplacée par une description schématique.

Quatrième étape : l'abstraction fonctionnelle

Les langages fonctionnels (ou comportementaux) de description de matériel, grâce aux nouvelles possibilités de description à un niveau d'abstraction plus élevé, ont répondu à des besoins fondamentaux des concepteurs de circuits intégrés :

- La réduction des temps de conception.
- L'accélération des simulations qui devenaient prohibitives avec l'accroissement de la complexité des dispositifs.
- La normalisation des échanges : des langages normalisés et universellement reconnus servent aux échanges entre partenaires industriels, entre fournisseurs et clients. Verilog, VHDL et SystemC en sont des exemples.
- L'anticipation : grâce aux modèles HDL il est possible de concevoir un système alors que ses composants ne sont pas encore disponibles.
- La fiabilité : les langages HDL sont conçus pour limiter en principe les risques d'erreur.
- La portabilité : les langages normalisés sont très largement portables.
- La maintenabilité et la réutilisabilité : les modifications et adaptations sont rendues plus simples donc moins risquées et moins coûteuses.

L'augmentation du nombre de portes intégrées (plusieurs millions de portes) sur une même puce fait apparaître la nécessité de développer des outils encore plus puissants : les

synthétiseurs logiques. Le dispositif à modéliser ou à concevoir sera représenté par sa fonction et non plus par sa structure. C'est le synthétiseur logique qui déterminera la structure automatiquement à partir de la fonction [*COM11*].

2.2 Utilité des HDL :

Les langages HDL offrent deux avantages majeurs en plus de la simplicité lors de la conception :

1. ***Simulation :*** Le but de la modélisation, c'est la simulation. Il existe deux points importants pour la simulation :

   - La fidélité : un modèle se doit d'être aussi précis que possible dans son champ d'application.
   - L'efficacité : le modèle doit pouvoir être simulé le plus rapidement possible et doit être portable, réutilisable et facile à maintenir.

Pour simuler un modèle, il faut disposer du modèle, bien sûr, mais aussi de stimuli, c'est-à-dire de la description des signaux d'entrées du modèle au cours du temps. Ces stimuli sont appelés les TESTSBENCHES en Anglais.

Les langages fonctionnels de description de matériel permettent de simplifier la conception en analysant automatiquement les résultats en cours de simulation.

Un environnement de simulation complet comprend : un générateur de stimuli, un modèle de l'objet à simuler et un vérificateur automatique des résultats. Ces trois composantes sont modélisées à l'aide du même langage.

2. Synthèse : Les langages fonctionnels de description matérielle servent aussi à concevoir. Il ne s'agit plus de modéliser en vue de la simulation, mais de décrire les objets qui seront véritablement fabriqués. Si les considérations de vitesse d'exécution en simulation existent toujours (la description sera simulée avant d'alimenter le synthétiseur, afin de vérifier que la fonction décrite est bien la fonction désirée) elles ne sont plus prioritaires. Ce qui compte le plus, c'est l'efficacité du code au sens du synthétiseur. En effet, pour que ce dernier produise la description structurelle la plus économique possible (et donc la surface de silicium la plus petite possible). [*COM11*]

2.3 Exemples de langages de description matérielle HDL :

Il existe un bon nombre de langages HDL, malgré leurs diversités, ils répondent tous aux objectifs fixés par les grandes lignes des langages HDL. Parmi eux : VHDL, Verilog, SysremC, SysremVerilog…etc.

2.3.1 VHDL (Very High Speed Integrated Circuits Hardware Description Language):

À l'origine, avant même la mise en place du VHDL, le programme VHSIC (Very High Speed Integrated Circuits), impulsé par le département de la défense des états unis dans les années 1970-1980, a donné naissance à un langage : VHSIC-HDL ou VHDL. Deux normes successives (IEEE-1976-1987 et 1993) en ont stabilisé la définition, complété par des extensions plus récentes ou a venir. L'évolution prévue est la création d'un langage commun analogique –numérique, dont le VHDL, sous sa forme actuelle, constituerait la facette numérique[6] [*JAC00*].

2.3.2 Verilog :

Verilog a été inventé par Gateway Design Automation Inc. aux alentours de 1984. C'était un langage propriétaire. Conçu à l'origine pour être utilisé dans leurs simulateurs logiques, mais le succès grandissant du VHDL a incité ses concepteurs de faire de Verilog un standard ouvert ; c'est le standard IEEE 1364 dont il existe plusieurs versions, qui ont été enrichies pour offrir des fonctions équivalentes à celles de VHDL. La syntaxe de Verilog est réputée largement inspirée du langage de programmation C, bien que la ressemblance se limite aux expressions. Ceci explique en partie son succès et sa diffusion rapide dans la communauté des ingénieurs qui ont déjà appris le langage C [*WIKI11*] [*COM11*].

2.3.3 Comparaison entre VHDL et VERILOG et choix :

En comparaison entre VHDL et VERILOG, ils sont conçus pour répondre aux mêmes exigences des descriptions HDL, mais il se trouve que ces deux langages ont été conçus chacun en donnant plus d'importance à un aspect plus tôt qu'un autre d'une description matériel. En effet, le langage VHDL se base plus sur le fait d'obtenir le circuit le plus optimisé que possible donc le moins encombrant dans le souci d'embarquer le plus de fonctionnalités dans un circuit logique programmable. Ce qui n'est pas sans conséquence car

[6] Pour les systèmes hybrides (analogique –numérique et voir même mécanique ou optique) ou microsystèmes il existe un sur ensemble du VHDL. Ce sur ensemble est connu sous le nom de VHDL-AMS Very High Speed Integrated Circuits Hardware Description Language For Analog And Mixed Signal [*HAM09*].

le code se voit plus long que celui rédigé en langage VERILOG pour une même fonction. Par contre le langage VERILOG cherche à réduire le temps de mise en place d'un code opérationnel donc le plus cours possible mais en réduisant les dimensions du code la conséquence, le circuit synthétisé consomme plus de ressources. Ce qui peut apparaitre comme une contrainte, mais en réalité un circuit logique programmable remplie à 50% donne satisfaction pour les concepteurs et celui rempli à 80% [*JIN10*]est très optimisé .

2.4 Le langage de description matériel VHDL (Very High Speed Integrated Circuits Hardware Description Language):

Le VHDL est un langage de description matériel HDL. Il répond à tous les critères établis pour un langage HDL (voir paragraphe 2.1 Quatrième étape).

Les plus importants points forts de ce langage sont [*JAC00*] :

Simulation, synthèse : le VHDL permet d'avoir des lignes de code pouvant être simulé (simulation fonctionnelle) dans le but de vérifier si le code obéit correctement à la fonction souhaitée mais pour parvenir à une maquette réalisable ce n'est pas suffisant. D'où la nécessité de pouvoir synthétiser le même code. Lors de la synthèse, le compilateur traduit le premier code en un autre équivalent mais à un niveau d'abstraction plus bas. À ce niveau de synthèse, le compilateur traduit le code de haut niveau en portes logiques ceci en fonction du circuit logique programmable qui est ciblé. Le fichier synthétisé lui aussi, se doit d'être simulé (simulation événementielle) et vérifier si le compilateur (synthétiseur) est resté fidèle aux exigences de départ. Puis le fichier est compilé une deuxième fois pour aboutir au circuit logique qui sera implémenté. Cette fois encore, une troisième simulation (facultative) (simulation temporelle) pour être certain que le circuit est resté inchangé.

Portabilité : avec deux objectifs, portable vis-à-vis du circuit logique programmable, c'est-à-dire peut être implémenté sur l'importe quel circuit logique programmable à condition d'avoir la capacité logique requise. Portable vis-à-vis du compilateur, pouvoir passer d'un compilateur à un autre et obtenir un même circuit en fin de processus.

Une construction hiérarchique : cette approche permet de simplifier la conception puis ce que les taches peuvent être divisées pour aboutir au point le plus simple que possible puis faire un assemblage des blocs.

Une description fonctionnelle : complémentaire de la précédente, la vision fonctionnelle apporte la puissance des langages de programmation. Tout algorithme est la description

interne d'un bloc situé quelque part dans la hiérarchie du schéma complet. La vision structurelle est concurrente, la vision algorithmique est séquentielle, au sens informatique du terme [*JAC00*]. Un programma VHDL doit être compris comme l'assemblage en parallèle des taches indépendantes qui s'exécutent concurremment.

2.5.1 Structure d'un programme VHDL le couple entity , architecture:

Un opérateur élémentaire, un circuit intégré, une carte électronique ou un système complet, est complètement défini par des signaux d'entrées et de sorties et par la fonction réalisée de façon interne [*JAC00*]. Les concepteurs du VHDL ont adopté l'approche suivante : l'importe quel système est considéré comme une boite noire. Cette boite noire a des entrées et des sorties. Ils ont appelé la boîte noire « ENTITY ». L'importe quel système électronique effectue des opérations sur le signal d'entré pour donner le résultat du traitement en sortie. Ces opérations sont le contenu de la boîte noire ce contenu est appelé « ARCHITECTURE ».

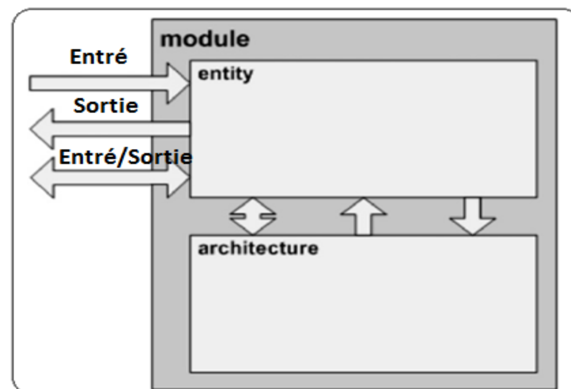


Figure 2.1 : structure de base d'un module sous VHDL [WIL11].

Entity ou Entité: Dans la partie déclarative de l'entité le circuit est décrit comment il est vu par l'extérieur ceci à travers les entrées, sorties et entrées/sorties.

```
Entity <entity name> Is Port (
   <signal name   : <signal direction> <data type>);
End <entity name>;
```

Figure 2.2 : Syntaxe déclarative de l'entité.

Architecture : l'architecture décrit le comportement que doit avoir le circuit ou les opérations qu'il doit effectuer. Une architecture se doit toujours d'être attachée à une entité (ils vont de paire).

```
Architecture <architecture name> Of <entity name> Is
<Define signals and constants>
Begin
End <architecture name>;
```

Figure 2.3 : Syntaxe déclarative de l'architecture.

C'est dans cette section que le programme est rédigé. Un programme comporte essentiellement les éléments suivants : les signaux internes, opérateurs logiques (synchrone ou concurrent), les process [*JAC11*]. La description d'une architecture peut prendre trois formes :

Description comportementale : Ce type de description spécifie le comportement du composant ou du circuit à réaliser au moyen d'instructions séquentielles ou sous forme de flot de données (constituant un process).

Description structurelle : Dans ce type de description, les interconnexions des composants préalablement décrits sont énoncées. Cette description est la transcription directe d'un schéma.

Description mixte : Elle regroupe les deux descriptions décrites précédemment. À chaque entité peut être associée à une ou plusieurs architectures mais au moment de l'exécution (simulation, synthèse…) seulement une architecture et une seule est utilisée.

2.5.2 Éléments du langage VHDL :

Le signal : le signal est représenté par certains types de données. Il est assigné par un nom et un type de donnée comme suit :

Signal « nom_signal » : « type_donné » ;.

Le terme Signal indique implicitement au compilateur de convertir le signal en un canal d'interconnexion.

Les constantes : la constante est un objet qui porte une valeur à l'initialisation l'enregistre tout au long du déroulement du programme. Elle est déclarée par le terme **Constant**.

Les variables : la valeur d'une variable est immédiatement mise à jour lors ce qu'elle est assignée. Une variable peut être déclarée soit à l'intérieur d'un processus ou d'un sous-programme et elle est local à son bloc de déclaration. Une variable n'est utilisée que dans le domaine de la programmation séquentielle. Une variable est déclarée comme suit :

Variable « nom_variable » : type « expression initiale » ;

Les types de données : le VHDL est un langage très typé. Chaque élément utilisé (signal, variable) doit avoir un type bien défini. Toute opération doit se faire avec des objets d'un même type. Sous VHDL les types de données sont standardisés et on y réfère par std_logic, std_logic_vector, bit, bit_vector… Ils sont regroupés dans les bibliothèques par défaut. L'utilisateur peut créer lui-même un type de donnée et l'intégré dans une bibliothèque. Le tableau suivant montre les valeurs admises par le type standard logic vector.

Value	Description
0	Low or logic zero
1	High or logic one
W	Weak unknown signal
L	Weak low
H	Weak high
U	Unknown or uninitialized
Z	High impedance
X	Unknown
-	Don't care

Le Process : il permet de décrire les instructions parallèles qui utilisent les mêmes signaux. Un process peut être vu comme un sous programme en boucle infinie qui ne s'arrête qu'à l'instruction de synchronisation **wait** (attendre). L'énoncé process est habituellement accompagné d'une liste de sensibilité (horloge, bit d'activation ou de sélection…) et il est exécuté lors ce qu'un des signaux de la liste de sensibilité change d'état.

Les instructions concurrentes : les instructions concurrentes sont décrites directement dans le corps de l'architecture. L'objectif de ces relations étant d'affecter des valeurs à des composants ou de réaliser des connexions, alors elles n'ont pas d'ordre d'exécution. Elles sont exécutées en parallèle.

Les instructions séquentielles : les instructions séquentielles sont internes aux processus, aux procédures et aux fonctions (sous-programmes). Elles constituent les outils de base des descriptions comportementales. Ces instructions, pour la plupart, sont inspirées des langages classiques de programmation. Parmi elles, on retrouve **if-then-else** qui permet de réaliser les boucles conditionnelles. **Case-then** pour tester des valeurs et choisir l'opération à effectuer.

L'instruction **wait**, qui permet de suspendre l'exécution d'un processus jusqu'à ce que la durée spécifiée soit évaluée. Les boucles telque : **for-loop, while-loop** …

Les attributs : ce sont des propriétés ou des caractéristiques associées à un objet ou un type. Ils sont soit prédéfinis ou définis par l'utilisateur. Ils sont déclarés comme suit :

Nom_var' nom_attribut ;

Les Librairies et les Paquetages : ce sont les librairies qui font qu'un programme et transportable d'un programme à un autre et permettent aussi leurs réutilisations. Une librairie est constitué d'un ou plusieurs paquetages et ce sont ces derniers qui renferment les fonctions, les procédures, les constantes, les types… elles sont fournies soit par l'IEEE en tant que partisan du standard VHDL ou bien créé par le concepteur lui-même. La syntaxe déclarative se fait comme suit :

Library IEEE ; pour déclarer la librairie

Use IEEE.STD_LOGIC_1164.ALL ; pour choisir le paquetage Standard Logic 1164

2.5.3 Structure d'un programme sous VHDL :

Un programme écrit sous VHDL obéit à la structure suivante :

Entête : c'est une partie facultative, elle referme des informations concernant le programmeur, la description du programme en général, la date de rédaction et toute information qui semblera importante pour celui qui rédige le programme.

Déclaration des librairies : en deuxième lieu vient la déclaration des librairies et des paquetages que le programmeur juge nécessaire pour son programme.

Déclaration d'entité : en troisième lieu vient la déclaration d'entité avec les signaux à utiliser dans tout le programme avec leur direction.

Déclaration d'architecture : juste après l'entité vient la déclaration de l'architecture qui décrit l'entité. Dans cette partie les signaux internes sont déclarés et les variables propres à cette même architecture. Puis le programme est rédigé. Voir la Figure 2.4.

```
Library IEEE;
Use IEEE.std_logic_1164.all;

Entity <entity name> Is Port
   (<list of ports or design inputs and outputs>);
End <entity name>;

Architecture <architecture name> Of <entity name> Is
   <in this section define signals and constants>

Signal <signal name>         : Data Type;

Begin
<concurrent statements>

<process name>: Process (sensitivity list)
Begin
   <sequential statements>
End;

End <architecture name>;
```

Figure 2.4 : Structure d'un programme sous VHDL [**GIN10**].

2.5.4 Les TESTBENCHS :

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

Pour élaborer un TESTBENCH, il faut établir la liste des cas la plus complète que possible. Les stimuli sont les entrées appliquées au programme via le simulateur pour imiter le comportement des vraies entrées. Après exécution du simulateur, les sorties sont observées si elles sont comme prévu par le programme et ceci sans aucun risque sur le matériel (FPGA…) [*JAC00*].

Chapitre II :

Instrumentation et transmission de données.

II. Introduction :

La prise de conscience de ce qui nous entoure est due à nos sens qui nous permettent de constater et d'évaluer les phénomènes physiques auxquels nous faisons face à tout instant. La qualité d'appréciation de ces phénomènes est liée de façon quasi évidente à la qualité et la précision de ces derniers. L'homme dans sa démarche pour apprivoiser le monde qui l'entoure et le rendre plus agréable et à sa portée a fait l'extension de ses sens. Cette extension lui permet de garder un œil sur tout non en plaçant des sentinelles mais en usant de l'ingénierie et de ses connaissances du monde physique et du comportement des matériaux devants ces phénomènes donnant naissance aux « capteurs ». Ces derniers liés à une chaîne de mesure adéquate permettent d'évaluer les phénomènes.

II.1 Capteurs : principes fondamentaux.

1.1 Caractéristiques générales des capteurs :

Le capteur est un dispositif qui soumis à l'action d'un mesurande (grandeur physique objet de la mesure : déplacement, température…) représenté par « m » non électrique présente une caractéristique d'une nature électrique (charge, tension, courant ou impédance) désignée par « s » et qui est fonction du mesurande :

$$\boldsymbol{s = f(m)}$$

La mesure de « s » doit permettre de connaître la valeur de « m ». Pour tout capteur la relation s=f(m) sous sa forme numériquement exploitable est explicitée par étalonnage. Pour un ensemble de valeurs de m connut avec précision, les valeurs correspondantes de « s » sont mesurée ce qui permet de tracer la courbe d'étalonnage.

Le capteur est mis en œuvre de sorte qu'il établisse une relation linéaire entre les variations Δs de la grandeur de sortie et Δm de la grandeur d'entrée :

$$\mathbf{\Delta s = S\ \Delta m}$$

S : définie comme la sensibilité du capteur.

Un des problèmes important dans la conception et réalisation des capteurs est la constance de sa sensibilité S qui doit répondre aussi peu que possible :

- De la valeur de m (linéarité) et de sa fréquence de variation (bande passante).
- Du temps (vieillissement).

- De l'action d'autres grandeurs physiques de son environnement qui ne sont l'objet de la mesure et qui sont désignés comme grandeur d'influence.

En tant d'éléments de circuits électriques, le capteur se présente, vu de sa sortie :

- Soit comme un générateur, « s » étant une charge, une tension ou un courant il s'agit alors d'un capteur actif.
- Soit comme une impédance, « s » étant alors une résistance, une inductance ou une capacité, le capteur est alors dit passif.

Le signal électrique est la partie variable du courant ou la tension qui porte l'information liée au mesurande, amplitude et fréquence du signal doivent être liées sans ambiguïtés à l'amplitude et la fréquence du mesurande. Un capteur actif qui est une source, délivre immédiatement un signal électrique. Il n'en est pas de même pour un capteur passif dont les variations d'impédance ne sont mesurables que par les modifications de la tension ou d'un courant qu'elle entraîne dans un circuit par ailleurs alimenté par une source extérieure. Le circuit électrique nécessairement associé à un capteur passif constitue son conditionneur et c'est l'ensemble du capteur et du conditionneur qui est la source du signal électrique [*ASC06*].

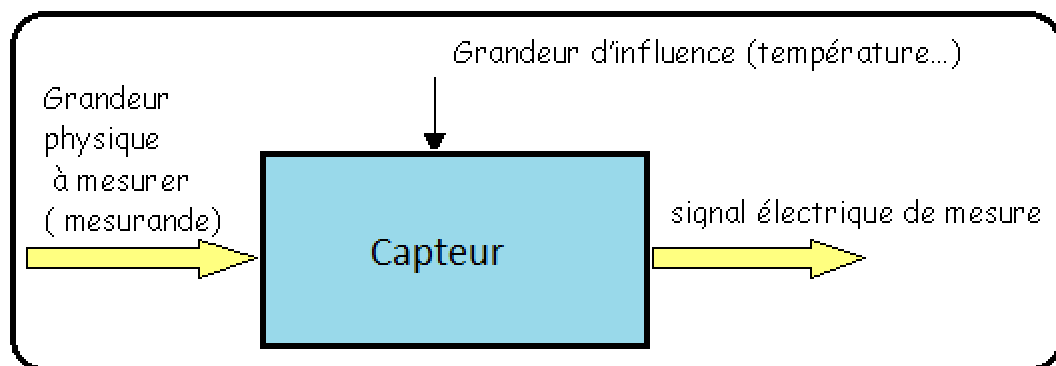


Figure II.1 : schéma représentatif d'un capteur.

1.2 Les différents types de capteurs :

Il existe différents types de capteurs, tout en gardant à l'esprit que d'un point de vue physique il existe les capteurs passifs et les capteurs actifs. D'un point de vue développement il existe d'autres types de capteurs qui sont à la base soit actifs ou passifs mais pour lesquels l'électronique associée est embarquée pour améliorer les performances [*ASC06*].

1.2.1 Les capteurs actifs : Un capteur actif fonctionne en générateur, il est généralement fondé dans son principe sur un effet physique qui assure la conversion en énergie électrique de la forme d'énergie propre au mesurande : énergie thermique, mécanique ou de rayonnement.

Le tableau suivant illustre les principes physiques de base utilisés par les capteurs actifs :

Mesurande	**Effet utilisé**	**Grandeur de sortie**
Température	Thermoélectricité	Tension
Flux de rayonnement optique	Pyroélectricité Photoémission Effet photovoltaïque Effet photoélectromagnétique	Charge Courant Tension Tension
Force Pression Accélération	Piézoélectricité	Charge
Vitesse	Induction électromagnétique	Tension
Position (aimant)	Effet hall	tension

1.2.2 Les capteurs passifs : il s'agit d'impédance dont l'un des paramètres déterminants est sensible au mesurande. Dans l'expression littérale d'une impédance sont présents des termes liés :

- D'une part à sa géométrie et ses dimensions.
- D'autre part aux propriétés électriques des matériaux : résistivité, perméabilité magnétique, constante diélectrique.

La variation d'impédance peut donc être due à l'action du mesurande :

- Soit sur les caractéristiques géométriques ou dimensionnelles.
- Soit sur les propriétés électriques des matériaux.
- Soit plus rarement sur les deux simultanément.

Le tableau suivant est un aperçu des divers mesurandes susceptibles de modifier les propriétés électriques de matériaux employés pour la réalisation de capteurs passifs :

Mesurande	**Caractéristique électrique sensible.**	**Types de matériaux utilisés**
Température. Très basse température	Résistivité Constante diélectrique.	Métaux : platine, nickel, cuivre, semi-conducteurs. Verres.
Flux de rayonnement optique	Résistivité	Semi-conducteurs
Déformation	Résistivité Perméabilité magnétique	Alliages de nickel, silicium dopé. Alliages ferromagnétiques.

Position (aimant)	Résistivité	Matériaux magnétorésistants : bismuth, antimoniure d'indium.
Humidité	Résistivité Constante diélectrique	Chlorure de lithium. Alumine ; polymère.
Niveau	Constante diélectrique	Liquides isolants.

1.2.3 Les capteurs composites : c'est un capteur non pas sensible au mesurande mais à l'un de ses effets. Le corps d'épreuve est le dispositif qui soumis au mesurande étudié en assure une première traduction en une autre grandeur physique non –électrique, le mesurande secondaire qu'un capteur adéquat traduit alors en grandeur électrique. L'ensemble formé par le corps d'épreuve et un capteur passif ou actif constitue un capteur composite.

Les corps d'épreuve sont très utilisés pour la mesure de grandeurs mécaniques : celles-ci imposent au corps d'épreuve des déformations ou des déplacements auxquels un capteur approprié est sensible.

Par exemple, dans un accéléromètre, la masse sismique est le corps d'épreuve qui convertit l'accélération, mesurande primaire, en une force d'inertie, mesurande secondaire à laquelle est sensible un capteur piézoélectrique.

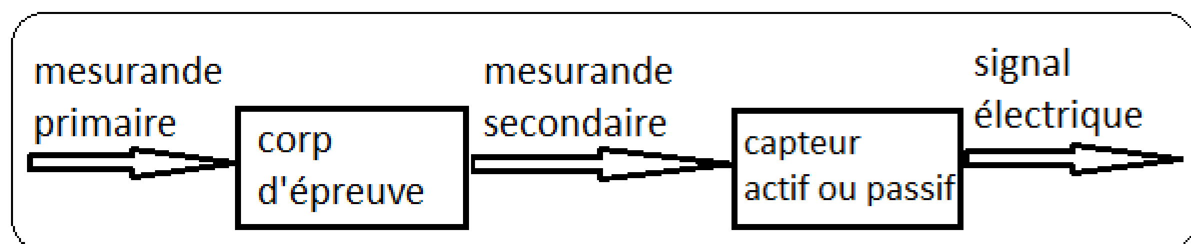


Figure II.2 : structure d'un capteur composite.

1.2.4 Les capteurs intégrés : un capteur intégré est un composant réalisé par les techniques de la microélectronique et qui regroupe sur un substrat de silicium commun le capteur proprement dit, le corps d'épreuve éventuel, des circuits électroniques de conditionnement du signal. L'intégration apporte de multiples avantages : miniaturisation, diminution des coûts par la fabrication en grande série, accroissement de la fiabilité par la suppression de nombreuses connexions soudées, interchangeabilité améliorée, meilleure protection vis-à-vis des parasites. Néanmoins, l'utilisation du silicium impose une limitation de la plage d'emploi de -50 °C à 150°C.

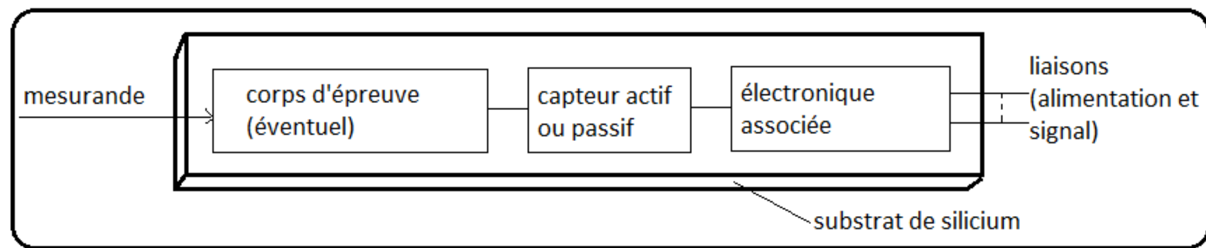


Figure II.3 : structure générale d'un capteur intégré.

Les capteurs intégrés fournissent en sortie le signal électrique contenant l'information utile qui peut être analogique ou numérique selon la conception de l'électronique associée à ce dernier mais ils ne disposent pas d'une interface de communication.

1.2.5 Les capteurs intelligents : un capteur intelligent est obtenu par l'association de la technologie des capteurs, de l'électronique et de l'informatique. Il doit être capable d'exécuter lui-même le traitement de signal. Un tel capteur possède la fonction de mémorisation d'un ordinateur et sa capacité d'exécuter le traitement des données. Il mesure, place les résultats en mémoire, exécute la sélection et le traitement de ces données. En outre, il accomplit la fonction de communication avec d'autres capteurs, et sélectionne lui-même les données à transmettre à l'ordinateur. Il possède donc la faculté de prise de décision. L'utilisateur est donc libre de travailler uniquement avec les données les mieux conditionnées [*ICH90*].

Le capteur intelligent offre des avantages spécifiques : configurabilité à distance, crédibilité accrue des mesures et aide à la maintenance grâce aux informations d'état fournies, répartition des tâches déchargeant le calculateur central.

La Figure II.1.4 illustre la structure générale d'un capteur intelligent. Il se compose de :

- Le capteur principal spécifique du mesurande étudié.
- Les capteurs secondaires propres aux grandeurs d'influence susceptible d'affecter les réponses du capteur principal.
- Les dispositifs classiques permettant l'obtention sous forme numérique de la grandeur de sortie de chaque capteur : conditionneur, multiplexeur, amplificateur, échantillonneur bloqueur, convertisseur analogique numérique CAN.
- Un microprocesseur affecté aux tâches suivantes : gestion de l'acquisition, correction de l'effet des grandeurs d'influence au moyen des paramètres stockés en PROM et des données fournies par les capteurs secondaires, linéarisation et diagnostic des capteurs.
- L'interface de communication bidirectionnelle : assure la liaison du capteur à un calculateur central via un bus partagé entre plusieurs capteurs intelligents.

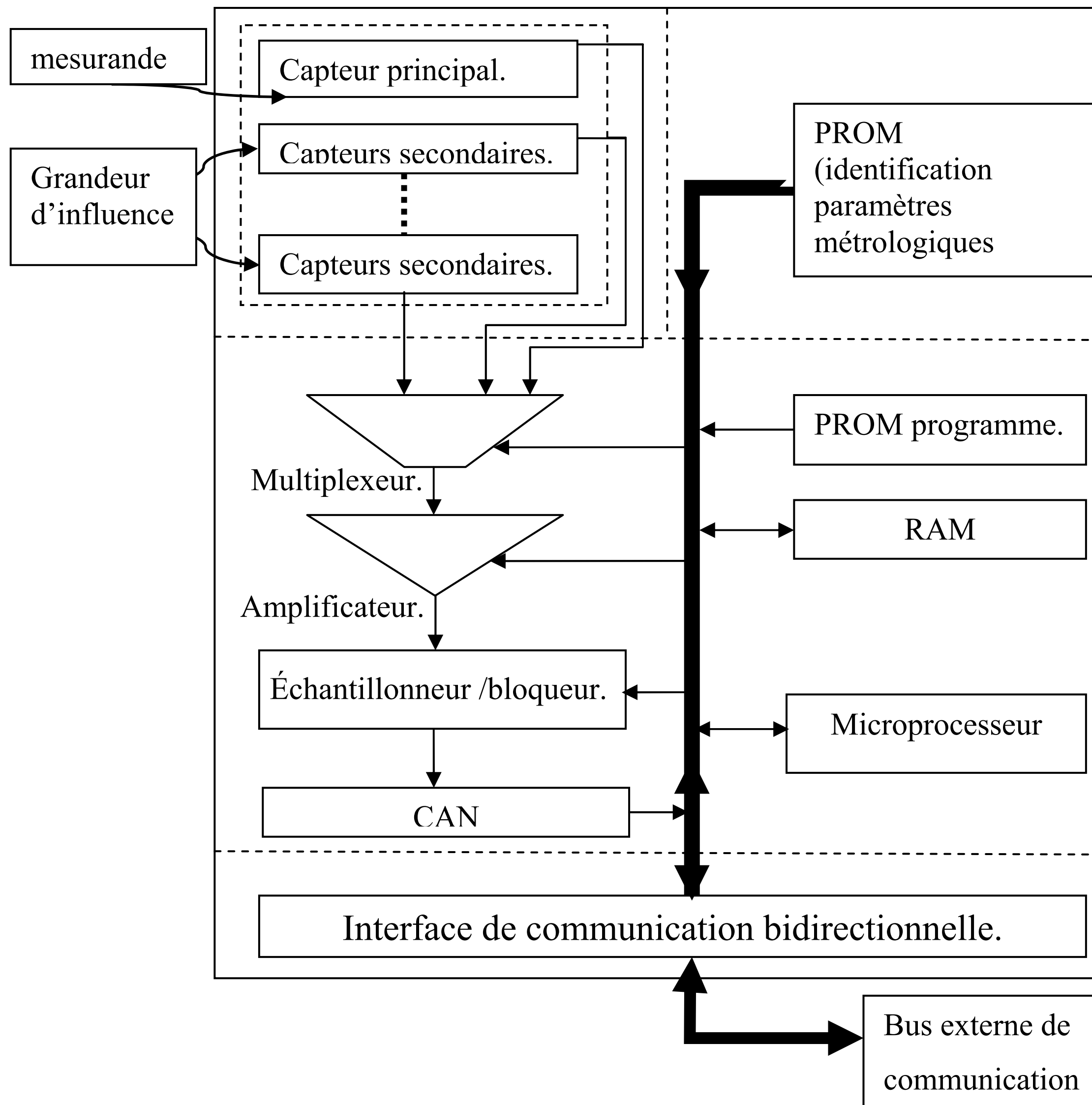


Figure II.4 : Structure générale d'un capteur intelligent.

1.3 Les grandeurs d'influences :

Les grandeurs d'influences sont des grandeurs physiques parasites auxquelles la réponse du capteur peut être sensible. Ainsi, le signal utile qui est dû au mesurande se retrouve noyé dans un bruit.

1.4 La chaîne de mesure :

La chaîne de mesure est constituée de l'ensemble des dispositifs, y compris le capteur, rendant possible dans les meilleures conditions la détermination précise de la valeur du mesurande.

À l'entrée de la chaîne, le capteur soumis à l'action du mesurande permet, directement s'il est actif ou par le moyen de son conditionneur si il est passif, d'injecter dans la chaîne le signal électrique, support de l'information liée au mesurande.
À la sortie de la chaîne, le signal électrique traité est converti sous une forme qui rend possible la lecture directe de la valeur recherchée du mesurande. C'est l'étalonnage de la chaîne de mesure qui permet d'attribuer à chaque indication en sortie la valeur correspondante du mesurande agissant sur l'entrée.
Sous sa forme la plus simple, la chaîne de mesure peut se réduire au capteur, et à son conditionneur éventuel, associé à un appareil de lecture.

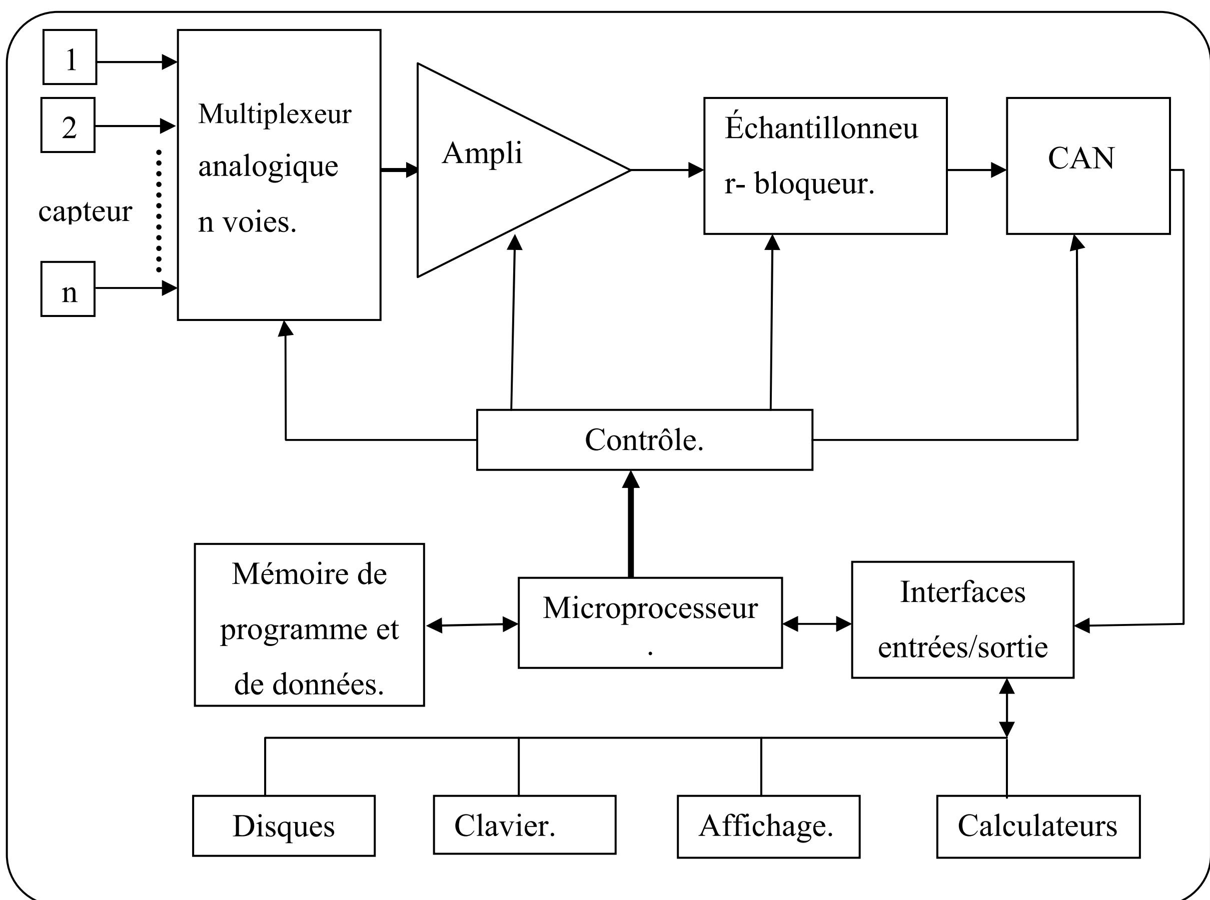


Figure II.5 : Exemple de constitution d'une chaîne de mesure contrôlée par microprocesseur.

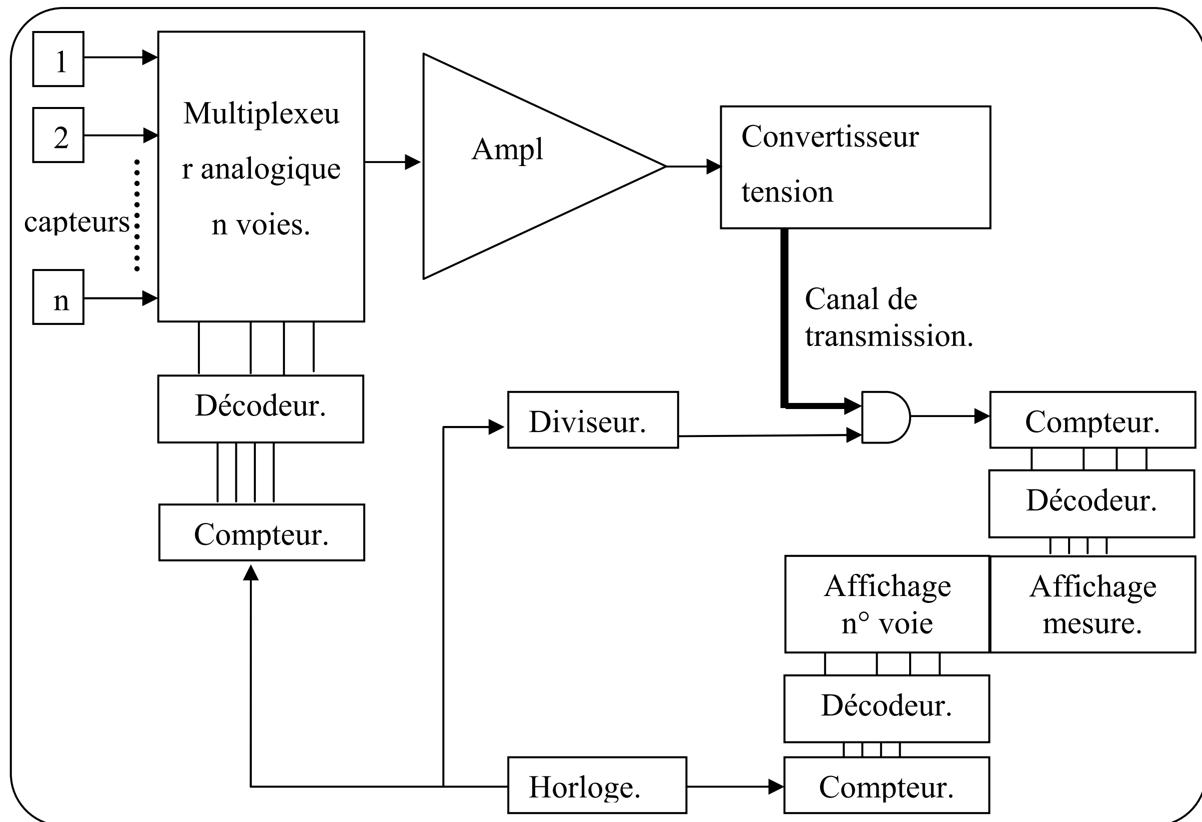


Figure II.6 : Exemple de constitution d'une chaîne de mesure avec conversion tension fréquence des signaux permettant leur transmission bifilaire.

Les conditions pratiques de mesure telles qu'elles sont imposées par l'environnement d'une part et par les performances exigées pour une exploitation satisfaisante du signal d'autre part amène à introduire dans la chaîne des blocs fonctionnels destinés à optimiser l'acquisition et le traitement du signal (voir les Figures II.5 et II.6) :

- Circuit de linéarisation du signal délivré par le capteur.
- Amplificateur d'instrumentation ou d'isolement destiné à réduire les tensions parasites de mode commun.
- Multiplexeur, amplificateur d'instrumentation programmable, échantillonneur bloqueur, convertisseur analogique –numérique lorsque l'information doit être traitée par calculateur (Figure II.5).
- Convertisseur tension –courant ou tension –fréquence lorsque le signal doit être transmis à distance par câble (Figure II.6).

Les fonctions importantes d'une chaîne de mesure peuvent être classées en deux rubriques :

1. Gestion de l'acquisition.
2. Traitement du signal requis par la précision et la nature de l'information recherchée.

Le calculateur est le maître de la chaîne d'acquisition, il délivre les séquences de signaux de commande activant de façon ordonnée les divers dispositifs concourants à l'obtention de la valeur du mesurande particulier dont la connaissance à un instant donné est nécessaire au déroulement de l'application :

- Sélection d'une voie d'entrée par envoi d'adresse au multiplexeur.
- Fixation du gain de l'amplificateur programmable.
- Échantillonnage puis blocage du signal.
- Déclenchement de la conversion analogique –numérique.
- Lecture de la donnée numérique à réception du signal de fin de conversion délivré par le convertisseur analogique –numérique.

Tout comme il gère les périphériques classiques d'entrée/sorties :

- Clavier : pour l'introduction d'instructions (modification des paramètres de mesure).
- Mémoire de masse pour l'archivage des mesures.
- Affichage du résultat de la mesure en cours.

Le calculateur offre d'autres possibilités très intéressantes comme des opérations mathématiques sur le signal numérisé, il corrige et analyse le signal. Les traitements numériques correctifs sont destinés à compenser certaines imperfections de la chaîne de mesure :

- Correction des dérives de zéro et de sensibilité, causées par les grandeurs d'influence.
- Correction de la non — linéarité des capteurs afin d'obtenir une donnée proportionnelle du mesurande.

Ces corrections peuvent être effectuées par voie analogique, mais au prix d'un matériel spécifique supplémentaire [*ASC06*].

1.5 Étalonnage du capteur :

L'étalonnage du capteur comprend l'ensemble des opérations qui permettent d'expliciter, sous forme graphique ou algébrique, la relation entre les valeurs du mesurande et celle de la grandeur électrique de sortie et ceci, compte tenu de tous les paramètres additionnels susceptibles de modifier la réponse du capteur.

Ces paramètres additionnels peuvent être :

- Soit des grandeurs physiques liées au mesurande et auxquelles le capteur est sensible.

- Soit des grandeurs physiques, indépendantes du mesurande, et auxquelles le capteur est soumis pendant son utilisation et qui peuvent modifier sa réponse.

Il existe deux types d'étalonnages : étalonnage simple et étalonnage multiple.

1.5.1 Étalonnage simple :

L'étalonnage simple s'applique à un mesurande défini par une grandeur physique unique et à un capteur non sensible ou non soumis à des grandeurs d'influence. Il s'agit en particulier de mesurande statique, c'est-à-dire à valeurs constantes (telle la mesure de la température à l'aide d'un thermocouple). Dans ces conditions, l'étalonnage consiste à associer à des valeurs parfaitement déterminées du mesurande les valeurs correspondantes de la grandeur électrique de sortie : l'étalonnage s'effectue par un seul type d'expérience.

L'étalonnage simple est réalisé par l'un des deux procédés suivants :

1. **Étalonnage direct ou absolu :** Les diverses valeurs du mesurande sont fournies soit par les étalons, soit par des éléments de référence dont la valeur est connue avec une précision très supérieure (de l'ordre de cent fois) à celle recherchée par le capteur.
2. **Étalonnage indirect ou par comparaison :** Un capteur de référence est utilisé dont la courbe d'étalonnage est connue et stable. Le capteur de référence et le capteur à étalonner sont soumis dans les mêmes conditions (simultanément si possible) à l'action de mesurandes identiques dont le capteur de référence permet de connaître les valeurs.

Les valeurs associées du mesurande et la grandeur électrique de sortie doivent cependant permettre sans ambiguïté et compte tenu de la précision des mesures, la relation entre mesurande et grandeur de sortie sous l'une ou l'autre des deux formes suivantes :

1. **Sous forme graphique :** c'est la courbe d'étalonnage.
2. **Sous forme algébrique :** c'est l'équation caractéristique du capteur.

1.5.2 Étalonnage multiple :

Lorsque le mesurande à lui seul ne permet pas de définir la réponse du capteur, il faut que soit précisée, par une série d'étalonnages successifs l'influence de chacun des paramètres actifs additionnels.

Dans le cas où le capteur est constitué de composantes susceptibles de présenter de l'hystérésis (mécanique ou magnétique), la valeur de la grandeur de sortie dépend non seulement de la valeur actuelle du mesurande, mais aussi de la suite de ses valeurs antérieures.

L'obtention d'une courbe d'étalonnage parfaitement définie exige alors une procédure d'étalonnage dans laquelle le capteur se trouve soumis à une succession ordonnée et spécifiée des valeurs du mesurande.

Tout capteur dans les conditions de son emploi est caractérisé par sa réponse en fréquence qui définit un domaine de fréquence à l'intérieur duquel sa réponse est optimale et le plus souvent indépendante de la fréquence. En dehors de ce domaine qui est la bande passante, les performances en général diminuent en fonction de la fréquence. Dans ces conditions il est nécessaire d'effectuer un double étalonnage qui détermine :

1. **La réponse en fréquence du capteur :** qui est obtenue en mesurant la grandeur de sortie en fonction de la fréquence du mesurande dont l'amplitude est maintenue constante.
2. **La réponse en fonction de l'amplitude du mesurande :** à des fréquences fixes choisies dans la bande passante.

Dans le cas où le capteur serait soumis à plusieurs grandeurs d'influences, il est nécessaire d'effectuer un étalonnage distinct pour chaque grandeur d'influence. C'est ainsi que les constructeurs établissent des séries de courbes. À partir de ces courbes, l'utilisateur peut, dans des conditions expérimentales qui lui sont propres, déterminer par interpolation la réponse du capteur.

1.5.3 Validité d'un étalonnage : répétabilité et interchangeabilité.

La répétabilité : est la qualité du capteur qui assure l'utilisateur de l'identité de la grandeur de sortie (dans des limites spécifiées) chaque fois que ce même capteur est utilisé dans des conditions identiques : mêmes mesurande et même paramètres additionnels.

L'interchangeabilité d'une série de capteurs d'un même type est la qualité de cette série qui garantit à l'utilisateur des résultats identiques (aux tolérances près) chaque fois qu'un quelconque capteur de cette série est utilisé dans des conditions identiques.

II.2. Conditionneurs des capteurs passifs et conditionneurs du signal :

2.1 Conditionneurs des capteurs passifs :

Le choix du conditionneur est une étape importante dans la réalisation d'un ensemble de mesure. C'est, en effet, l'association capteur-conditionneur qui détermine le signal électrique ;

de la constitution du conditionneur dépendent d'un certain nombre de performances de l'ensemble de mesure : sensibilité, linéarité, insensibilité à certaines grandeurs d'influence. [*ASC06*].

Les types de conditionneurs les plus généralement utilisés sont :

- **Le montage potentiométrique :** associé en série d'une source, du capteur et d'une impédance qui peut être ou non du même type.
- **Le pont d'impédances :** donc, l'équilibre permet la détermination de l'impédance du capteur ou dont le déséquilibre est une mesure de la variation de cette impédance.
- **Le circuit oscillant :** qui contient l'impédance du capteur et qui est partie d'un oscillateur dont il fixe la fréquence.
- **L'amplificateur opérationnel :** donc, l'impédance du capteur est l'un des éléments déterminants de son gain.

2.2 Les conditionneurs du signal,

La lecture directe des capteurs actifs, qu'ils soient équivalents à des sources de tension, de courant ou de charges est rarement satisfaisant, car elle suppose une correction qu'il n'est pas toujours aisé d'évaluer. Les conditionneurs du signal, dont la fonction est en rapport direct avec la nature du signal telle que celle-ci résulte d'une part des caractéristiques propres du capteur et d'autre part des conditions pratiquent de la mesure.

Pour obtenir un signal exploitable en fin de chaîne de mesure, il existe quatre étapes majeures :

1. **Adaptation de la source du signal à la chaîne de mesure :** À fin que le signal soit obtenu dans les meilleures conditions, d'une part de sensibilité et d'autre part de stabilité vis-à-vis des variations éventuelles de l'impédance interne, le générateur équivalent doit être chargé par une impédance appropriée.
2. **Linéarisation :** Un capteur ou son conditionneur éventuel peuvent présenter dans leur domaine d'emploi des écarts à la linéarité interdisant de considérer la sensibilité comme constante à la précision exigée des mesures. La linéarisation est l'étape qui permet de corriger ces défauts de linéarité. Les procédés de linéarisation peuvent être classés en deux groupes :
    a. D'une part, ceux qui interviennent sur la source même du signal électrique de façon à linéariser ce dernier dès son origine.

   b. D’autre part, ceux qui interviennent en aval de la source afin de corriger la non-linéarité du signal qu’elle fournit par un traitement approprié, analogique ou digital.

3. **Amplification :** elle est réalisée par des amplificateurs dont le rôle est le suivant :
   a. Amplification du niveau de tension : protection du signal.
   b. Amélioration de la précision de mesure : adaptation au niveau du dispositif amont de la chaîne
   c. Transfert optimal du signal : haute impédance d’entrée et faible impédance de sortie.
4. **Détection de l’information :** c’est l’étape où il faut extraire de la tension modulée l’information c'est-à-dire un signal électrique qui soit à l’image des seules variations du mesurande.

II.3. Interface de communication I2C (Inter Integrated circuit) :

3.1 Présentation du bus I2C :

Le bus I2C est un bus série multimaster, développé par PHILIPS au début des années 1980 pour minimiser les liaisons entre les circuits intégrés numériques de ses produits (téléviseurs, éléments HI-FI… etc.). Aujourd’hui, c’est un standard très répondu chez les fabricants et très utilisé par les concepteurs.

Le bus I²C permet de faire communiquer entre eux des composants électroniques très divers et de technologies différentes (NMOS, CMOS, BIPOLAIRE) grâce à seulement deux fils : un signal de données (SDA pour Serial data), un signal d'horloge (SCL pour Serial CLock) et une référence électrique (masse) commune à toute l’installation (voir la figure II.7).

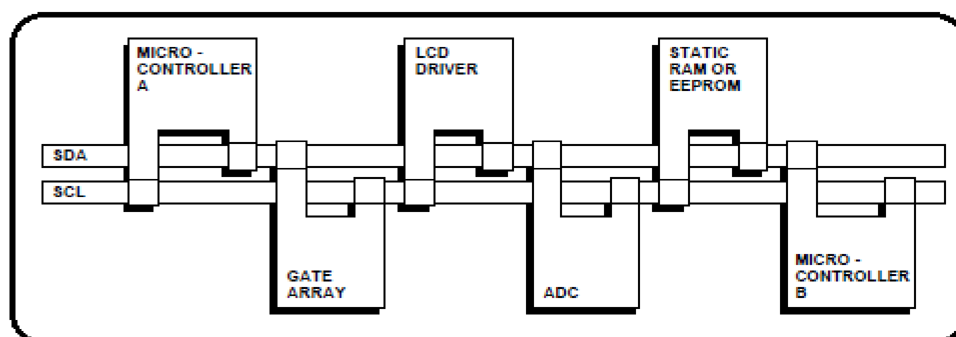


Figure II.7 : représentation du BUS I2C avec quelques composants reliés via le bus à noter deux microcontrôleurs, un FPGA, un afficheur LCD, une mémoire et un ADC (réf : I2C-BUS SPÉCIFICATION référence figure : MBC645).

Ces deux fils transportent les informations entre tous les éléments connectés à ce bus. Chaque élément est reconnu par une adresse unique et peut fonctionner en émetteur ou récepteur (selon le composant). En plus de pouvoir émettre ou recevoir des données, un élément du bus peut être considéré comme maître ou esclave lors d'un échange de données. Un maître est un élément qui initialise un transfert de données dans le bus et il génère le signal d'horloge qui permettra ce transfert de données. L'élément adressé est considéré comme un esclave [*PHI00*].

Le tableau suivant est un tableau de terminologie du bus I2C [*PHI00*] :

Terme :	**Description :**
Transmitter (émetteur)	Un élément qui envoie des données au bus.
Receiver (récepteur)	Un élément qui reçoit des données au bus.
Master (maître)	Un élément qui initialise un transfert, génère un signal d'horloge et termine le transfert.
Slave (esclave)	L'élément adressé par le maître.
Multi-master (multimaître)	Plusieurs maîtres qui peuvent contrôler le bus sans corrompre le message.
Arbitration (arbitrage)	C'est une procédure qui intervient dans le cas où plusieurs maîtres tenteraient de contrôler le bus simultanément, cet arbitrage permet d'attribuer le contrôle à un seul maître sans que son message initial ne soit modifié.
Synchronisation (synchronisation)	La procédure de synchronisation du signal d'horloge entre deux éléments ou plus.

3.3.1 Caractéristiques générales :

Les deux lignes SDA et SCL sont bidirectionnelles, connectées à une tension positive via une résistance de rappel. Quand le bus est libre, les deux lignes sont dans un état haut. Les étages de sorties connectées au bus doivent être à drain ouvert ou à collecteur ouvert pour réaliser la fonction AND (ET) avec le bus. Au repos, tous les circuits connectés doivent imposer un niveau haut sur leurs sorties respectives. Si les lignes SDA et SCL sont au niveau haut dans ces conditions, cela signifie qu'aucun circuit ne tente de prendre le contrôle du bus. Si une des lignes SDA ou SCL passe à un niveau bas dans les mêmes conditions, c'est qu'un des circuits désire prendre le contrôle du bus. Mais il peut aussi y avoir deux circuits qui tentent de prendre le contrôle du bus en même temps (ou à quelques nanosecondes d'écart près). Il faut donc mettre en place un protocole pour gérer les conflits possibles (voir la figure II.8).

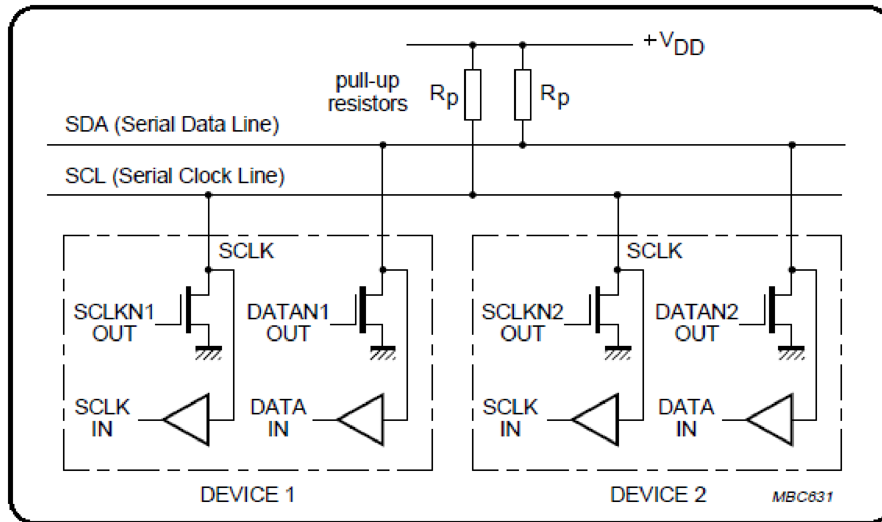


Figure II.8 : Connexion d'une interface au bus I2C en mode standard et Fast mode (réf : I2C-BUS SPÉCIFICATION référence figure : MBC631).

Vu la diversité technologique des circuits intégrés qui existent et qui peuvent être connectés sur le bus I2C, les niveaux logiques 0 (Low) et 1 (High) ne sont pas fixés et dépendent de la tension d'alimentation $+V_{DD}$. [*PHI06*].

Les données peuvent être transférées via le bus à une fréquence de 100 kbit/s en mode standard, une fréquence supérieure à 400 kbit/s en Fast mode et une fréquence supérieure à 3.4 Mbit/s en mode High-speed. Le nombre d'interfaces connectées sur le bus ne dépend que de la limite capacitive du bus qui est de 400pf.

Un seul bit est transféré pour chaque impulsion d'horloge, la donnée sur la ligne SDA ne doit pas basculer durant la demi-période haute de l'horloge. L'état haut ou bas de la ligne SDA peut seulement basculer durant la demi-période basse de la ligne SCL. La fréquence des signaux de la ligne SDA doit être au moins deux fois supérieur à celle de la ligne SCL, elle est recommandée à ce qu'elle soit quatre fois supérieure. [*PHI00*] (voir la figure II.9).

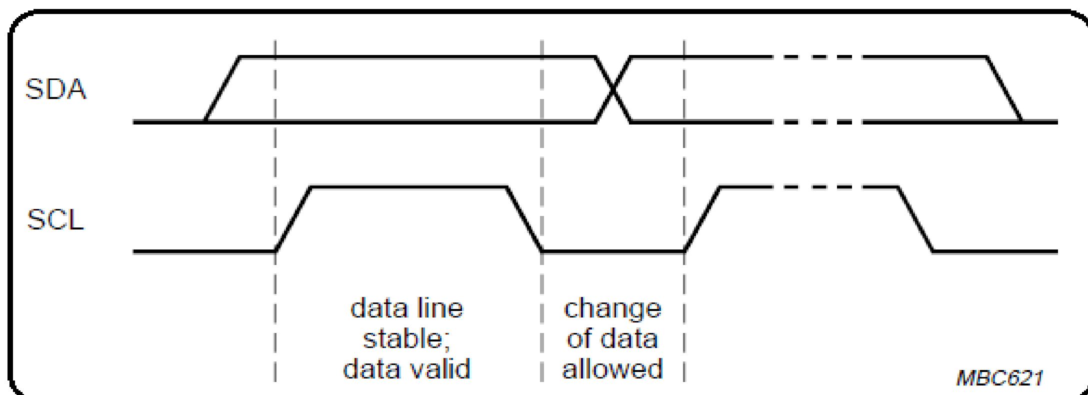


Figure II.9 : transfère d'un bit sur le bus I2C (réf : I2C-BUS SPÉCIFICATION référence figure : MBC621).

3.4 Le protocole I2C :

Le protocole du bus I²C définit la succession des états possibles sur les lignes SDA et SCL, et comment devraient réagir les circuits en tout cas de figure. En cas normal (hors conflit entre maîtres), une communication standard consiste en ces quatre étapes suivantes [*HER03*] :

1. Génération du signal de départ START (S.).
2. Transfert de l'adresse de l'esclave.
3. Transfert de données (bidirectionnel).
4. Génération du signal d'arrêt STOP (P).

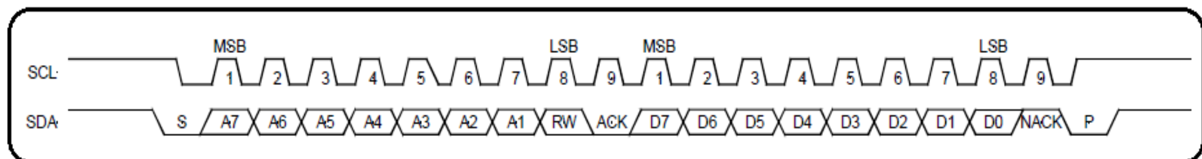


Figure II.10 : Exemple type de la transmission d'un octet codé selon le protocole I2C.

3.4.1 Le signal START et le signal STOP :

Le protocole du bus I^2C réserve une condition unique de début de transfert de données START (S.) (qui peut être interprété comme une prise de contrôle du bus) et une condition unique de fin de transfert de données STOP (P) (voir la figure II.11).

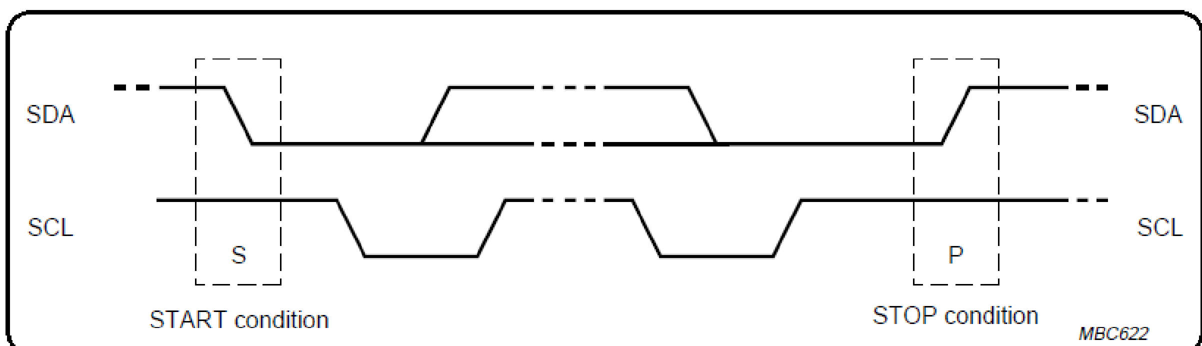


Figure II.11 : Condition de début START et de fin STOP d'une communication (réf : I2C-BUS SPÉCIFICATION référence figure : MBC622).

Une transition d'un état haut à un état bas sur la ligne SDA au moment où la ligne SCL est à l'état haut est un cas unique qui indique la condition de départ START, après quoi le bus est considéré occupé.

Une transition de l'état bas à l'état haut sur la ligne SDA au moment où la ligne SCL est à l'état haut indique la condition de fin STOP après un instant (4,7 us) le bus est considéré libre. Le bus reste occupé seulement si un deuxième START (Sr) est généré malgré le fait qu'un STOP l'a précédé. Les conditions START et STOP sont toujours générées par le maître [*PHI06*].

3.4.2 Transfert de données :

3.4.2.1 Le format d'une trame :

Le format d'une trame mise sur la ligne SDA est de 8 bits, le nombre de trames transmises n'est pas limité. Le premier bit de chaque trame doit être le bit de poids fort MSB (voir la Figure II.12). La fin de chaque trame est suivie par un neuvième bit appelé bit d'acquittement (Acknowledge ACK).

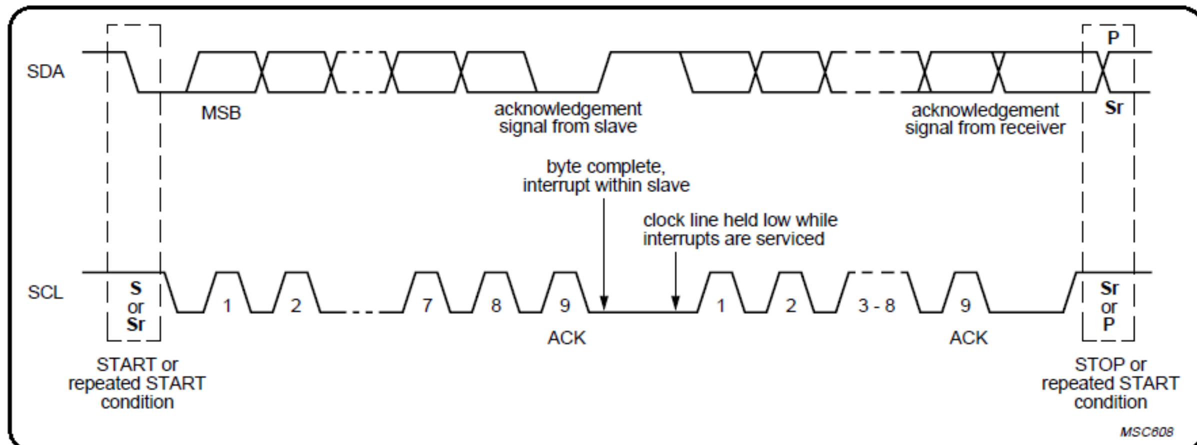


Figure II.12 : Transfert de données sur le bus I2C (réf : I2C-BUS SPÉCIFICATION référence figure : MSC608).

3.4.2.2 Acquittement (Acknowledge ACK) :

Pour chaque trame de données transférée est obligatoirement suivi d'un bit ACK appelé acquittement (Acknowledge), ce dernier est généré par le maître. Le processus d'acquittement se déroule comme suit : après une trame de donnée, le maître génère un neuvième bit, le bit d'acquittement ACK, émetteur libère la ligne SDA à l'état haut, mais le récepteur devra mettre la ligne SDA à l'état bas durant la période d'horloge d'ACK (voir la figure II.13).

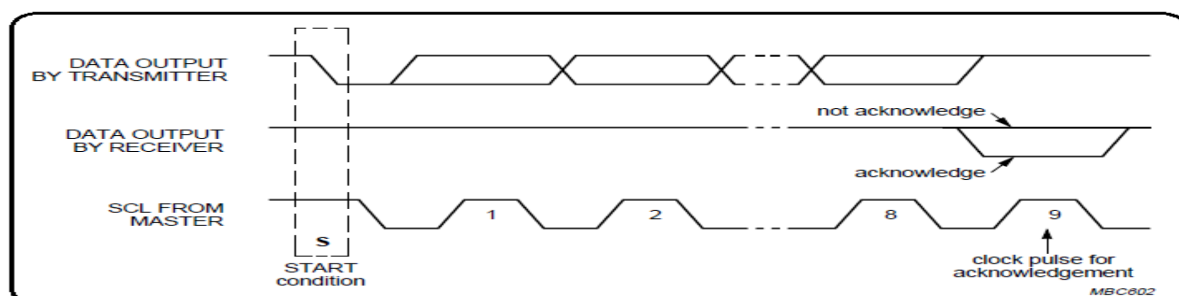


Figure II.13 : Acquittement (Acknowledge) sur le bus I2C (réf I2C-BUS SPÉCIFICATION référence figure : MSC608).

Un esclave est tenu de répondre par un acquittement à la fin de chaque transfert. S'il ne s'acquitte pas de ce transfert, le maître comprend que l'esclave n'est pas disposé à

communiquer alors le maître génère une condition d'arrêt pour annuler ou une nouvelle condition de départ START pour un nouveau transfert.

Si l'esclave répond à un appel sur le bus, mais peut de temps après, il se retrouve dans une situation l'empêchant de poursuivre la communication alors l'esclave ne génère pas l'ACK à la prochaine trame et le maître se doit d'abandonner la communication par un STOP ou répéter le START [*PHI06*].

3.4.3 Synchronisation d'horloge et Arbitrage :

Deux maîtres peuvent tenter de prendre le contrôle du bus en même temps, c'est alors qu'il y a nécessité de départager les deux maîtres et permettre à l'un d'eux d'effectuer sa transmission. Pour départager les deux maîtres, le protocole I2C a mis en place deux dispositifs pour départager les maîtres : la synchronisation d'horloge et l'arbitrage.

La synchronisation d'horloge :

L'interconnexion des éléments sur le bus I2C réalise la fonction « ET » par construction. Quand un maître tente de prendre le contrôle du bus en imposant un état bas sur la ligne SCL une fois la demi-période basse du maître est terminée, elle bascule vers l'état haut (voir la figure II.14). Néanmoins, cette transition de l'état bas vers l'état haut ne pourra pas entraîner un basculement sur la ligne SCL si une autre horloge d'un autre maître impose un état bas. C'est, ainsi la ligne SCL sera maintenue, à l'état bas par le maître disposant de l'état bas le plus long. Le maître avec l'état bas le plus court entre en état d'attente lors de son basculement à l'état haut.

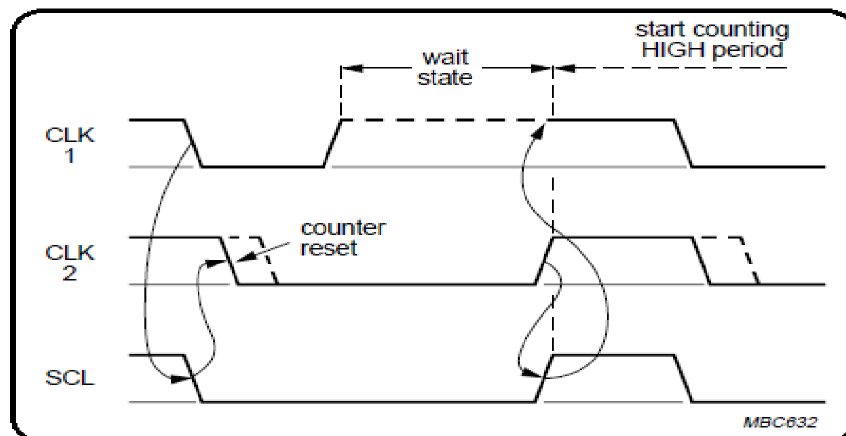


Figure II.14 : Synchronisation d'horloge durant la procédure d'arbitrage (réf : I2C-BUS SPÉCIFICATION référence figure : MSC608).

Une fois tous les maîtres auront basculé vers l'état haut la ligne SCL basculera aussi. Ainsi, il y a cohérence entre tous les maîtres et la ligne SCL. Le premier maître qui aura fini de compter son état haut basculera vers l'état bas entraînant avec lui la ligne SCL.
L'horloge sur la ligne SCL est générée par le maître disposant d'une horloge avec état bas le plus long et l'état haut est généré par l'horloge du maître disposant d'état haut le plus court, ce processus résulte en une synchronisation d'horloge.

Processus d'Arbitrage (Arbitration en Anglais) :

L'arbitrage s'impose dans le cas où deux maîtres tenteraient de contrôler le bus en même temps. Un maître ne peut entamer un transfert seulement si le bus est libre. Deux maîtres peuvent générer la condition de départ START avec un minimum de temps de (4us) pour avoir un START valide sur le bus. C'est alors qu'un arbitrage est nécessaire pour trancher duquel des deux maîtres finira sa transmission.
Un arbitrage se déroule bit par bit. Durant chaque bit, quand SCL est haut, chaque maître doit vérifier si l'information sur la ligne SDA est identique à l'information interne envoyée sur le bus. Deux maîtres peuvent effectuer un transfert complet sans erreurs tant que leurs transmissions sont identiques.
Au premier abord, le maître tente d'envoyer un état haut sur la ligne SDA, mais il détecte que celle-ci reste à l'état bas, c'est alors qu'il se rend compte qu'il a perdu l'arbitrage et il n'envoie plus sur la ligne SDA et la laisse libre (il ne doit plus essayer d'imposer un état bas). L'autre maître continue pour terminer sa transmission. Aucune information n'est perdue lors du processus d'arbitrage. Le maître qui a perdu l'arbitrage pourra tenter de générer l'horloge une fois la trame pour laquelle il a perdu l'arbitrage est envoyée et devra recommencer si le bus est toujours occupé.
Si un maître intègre aussi la fonction d'esclave et s'il a perdu un arbitrage durant la phase d'adressage (deux maîtres peuvent tenter d'adresser un même esclave donc aucun des deux ne perd la main à ce stade), il n'est pas exclu que le maître gagnant tente d'adresser le maître perdant. C'est alors, le maître perdant devra basculer en mode esclave. La figure II.15 illustre la procédure d'arbitrage entre deux maîtres. À un certain moment de la transmission, il y a une différence entre la donnée interne du maître 1 (DATA1) et le niveau de la ligne SDA. Le maître 1 ne doit plus émettre et ne plus encombrer la ligne SDA. Ceci ne devra pas corrompre le message du maître gagnant [*PHI06*].

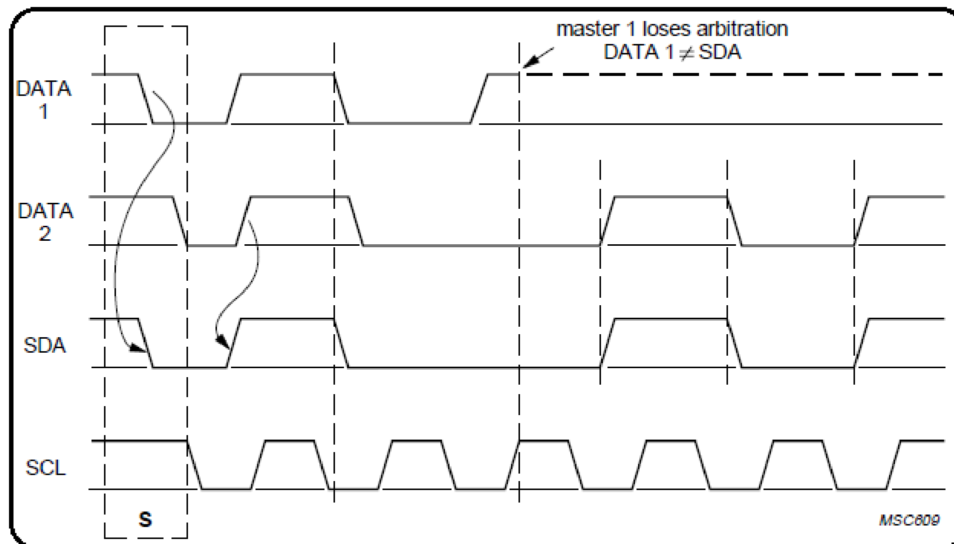


Figure II.15 : La procédure d'arbitrage entre deux maîtres (réf : I2C-BUS SPÉCIFICATION référence figure : MSC609).

Une fois le maître gagnant a envoyé l'adresse et la donnée, il n'y a pas un maître principal ou bien un maître prioritaire sur le bus I2C.

3.4.4. Le Clock Stretching ou ralentissement de la vitesse :

L'esclave peut se retrouver dans une situation où il ne peut pas recevoir ou transmettre une trame complète pour une raison donnée (par exemple, une interruption interne), il peut maintenir la ligne SCL à l'état bas pour forcer le maître dans un état d'attente. Une fois l'esclave est prêt il relâche le forçage sur la ligne SCL pour continuer la transmission (voir la figure II.16).

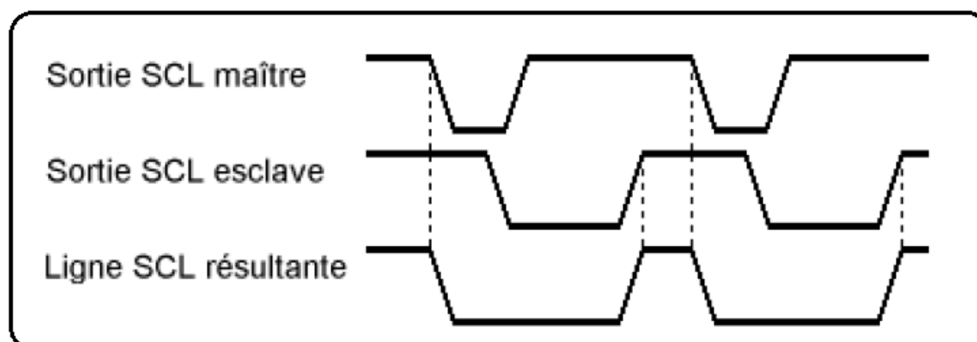


Figure II.16 : Processus de mise en attente du maître par l'esclave sue la ligne SCL.

3.4.5 L'adressage des transferts :

Le protocole I2C est très largement utilisé dans les différents secteurs de l'électronique. PHILIPS, propose plus de 150 composants différents sur son catalogue tous intégrant le bus I2C et d'autres fabricants proposent des variétés de composants qui se comptent par milliers. Chaque composant sur le bus I2C doit avoir une adresse unique pour éviter des conflits

d'adresses. Les composants sont vendus avec des adresses qui ont une partie fixe et une partie programmable. C'est pour cela que le protocole I2C a dû mettre de l'ordre en imposant des règles d'adressage qu'il faut respecter pour éviter des erreurs sur le bus I2C.

Adressage d'un esclave, adressage 7-bits :

Juste après la condition de départ START, l'adresse est envoyée. Une adresse est une trame de huit bits dont les sept premiers bits constituent l'adresse de l'esclave et le huitième bit R/W indique la direction du transfert. Read pour « 1 » un transfert en écriture, Write pour « 0 » un transfert en lecture (voir la figure II.17).

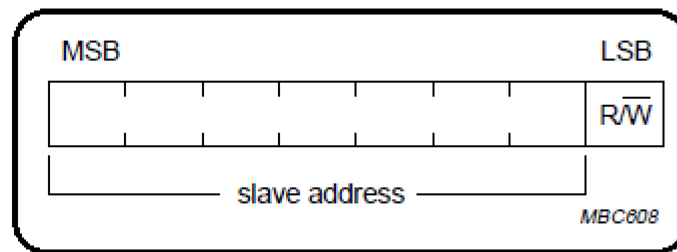


Figure II.17 : Le premier octet après la condition de départ START (réf : I2C-BUS SPÉCIFICATION référence figure : MBC608).

1. **Maître-émetteur transmet à l'esclave-récepteur :** Le maître adresse un esclave en écriture, l'esclave répond par un acquittement, puis il y a un transfert de données puis le maître génère la condition d'arrêt STOP (voir la figure II.18).

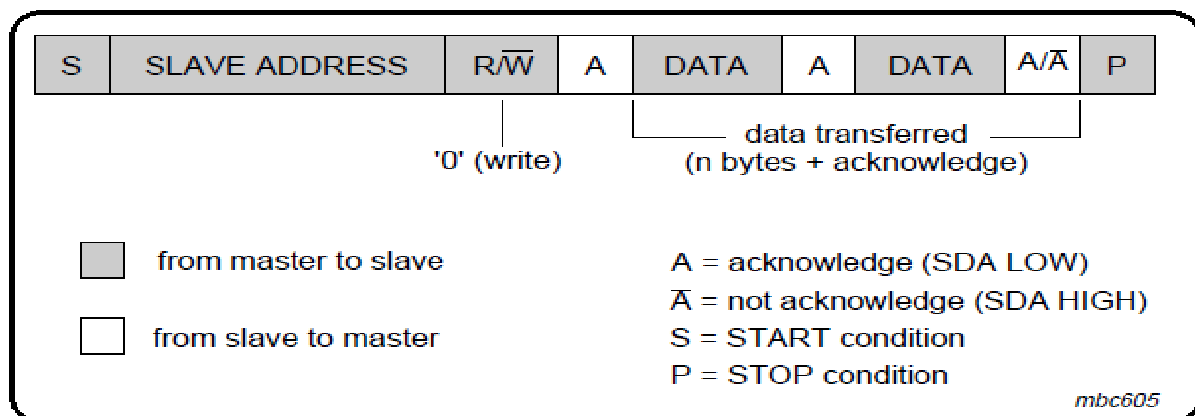


Figure II.18 : Adressage d'esclave dans le cas où le maître est émetteur l'esclave récepteur sous un format 7-bit sans changement de direction de transfert (réf : I2C-BUS SPÉCIFICATION référence figure : MBC605).

2. **Maître récepteur reçoit des données de l'esclave-émetteur :** le maître adresse l'esclave en lecture, l'esclave répond par un acquittement. L'esclave envoie les données et à la fin de chaque trame le maître répond par un acquittement. À la fin du transfert, le maître émet une condition d'arrêt sans s'acquitter de la dernière trame (voir la Figure II.19).

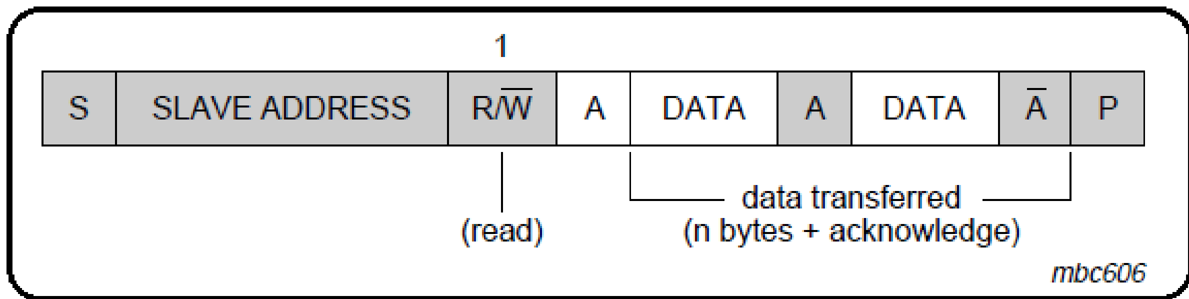


Figure II. 19 : Le maître lit l'esclave juste après le premier octet (réf : I2C-BUS SPÉCIFICATION référence figure : MBC606).

3. **Un format combiné :** c'est un échange où le maître lit et écrit l'esclave. Durant un changement de direction de transfert, la condition START Et l'adresse de l'esclave est répétée, mais le bit R/W change de valeur. En effet, car lors de la dernière trame le maître ne s'est pas acquitté donc il peut réécrire le bit R/W (voir la figure II.20).

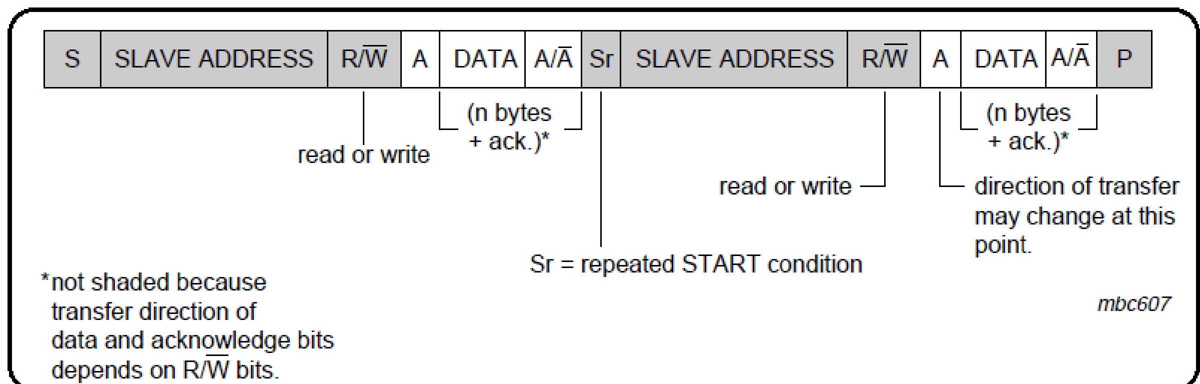


Figure II.20 : Illustration du format combiné (réf : I2C-BUS SPÉCIFICATION référence figure : MBC608).

Adressage sous 10-bits :

Suite au succès grandissant du bus I2C et l'attribution d'adresse spécifique pour des composants standards (interfaces, afficheurs LCD, mémoires… etc.) et l'unicité de l'adresse qui est indispensable. Ce sont des raisons, qui on fait, la nécessité d'étendre les possibilités d'adressage sur le bus I2C, l'adressage 7-bits et l'adressage 10-bits. Les composants avec 7-bits et 10-bits d'adresse peuvent être connectés ensemble sur un même bus I2C sans problèmes.

Les 10-bits d'adresse pour esclave sont formés des deux premiers octets suivant la condition START. Les sept premiers bits du premier octet sont constitués de 1111 0XX desquels les deux bits (XX) sont les bits les plus significatifs (MSBS) des dix bits d'adresse, le huitième bit est le bit lecture/écriture (R/W) qui détermine le sens de l'échange.

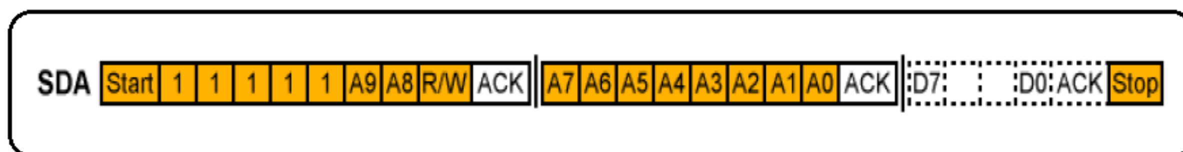


Figure II.21 : Illustration de l'adressage d'esclave sous 10-bits.

Il y a huit combinaisons possibles de l'adresse réservée 1111 XXX, seulement quatre sont utilisés 1111 0XX pour l'adressage 10-bits. Les quatre restants 1111 1XX sont réservés pour de futures évolutions du bus I2C.

Les adresses réservées :

Il y a huit adresses réservées (0000 XXX et 1111 XXX) pour les besoins suivants [*PHI06*]:

Adresse esclave	**Le bit R/W**	**Description**
0000 000	0	Adresse d'appel général.
0000 000	1	START byte : octet utilisé pour synchroniser les périphériques lents avec les périphériques rapides.
0000 001	X	CBUS : L'émission de cet octet permet de rendre sourd tous les circuits I²C présents sur le bus. À partir de ce moment, on peut transmettre ce que l'on désire sur le bus, en utilisant par exemple un autre protocole. Le bus repasse en mode normal lors de la réception d'une condition d'arrêt.
0000 010	X	Réservé pour différents formats de bus.
0000 011	X	Réservé pour un usage ultérieur.
0000 1XX	X	Code pour maître en mode haute vitesse (HS-mode).
1111 1XX	X	Réservé pour un usage ultérieur.
1111 0XX	X	Adressage d'esclave sous 10-bits.

Dans un système une adresse réservée peut être attribué à un esclave mais sous réserve que celle-ci ne soit jamais utilisée par le système.

Adresse d'appel général :

Un appel général est constitué de deux octets, le premier constitue l'appel et le second l'objet de cet appel (voir la figure II.22).

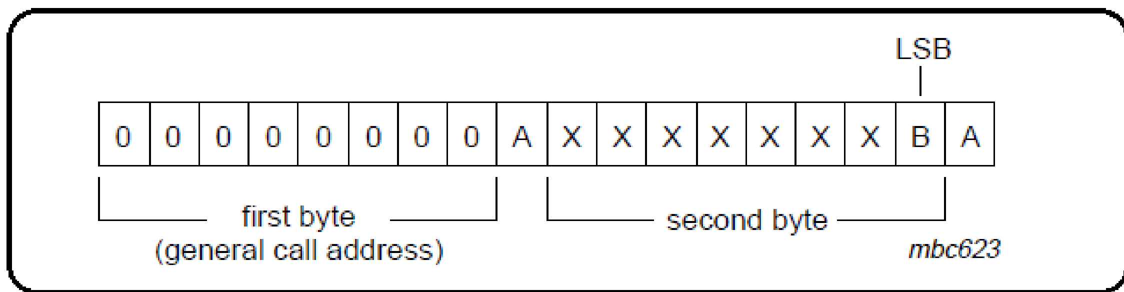


Figure II.22 : Format d'un appel général (réf : I2C-BUS SPECIFICATION référence figure : MBC623).

Après l'émission d'un appel général, les circuits ayant la capacité de traiter ce genre de demande d'appel émettent un acquittement.

Le deuxième octet permet de définir le contenu de l'appel :

0000 0110 (06h) RESET : Remet tous les registres de circuits connectés dans leur état initial (à la mise sous tension). Les circuits qui le peuvent rechargent leur adresse esclave.

0000 0100 (04h) : Les circuits définissant leur adresse de façon matérielle réinitialisent leur adresse esclave sans réinitialiser les circuits.

0000 0000 : Adresse interdite.

xxxx xxx1 : Cette commande joue le rôle d'interruption. xxxx xxx peut être l'adresse du circuit qui a généré l'interruption.

Les autres valeurs du second octet ne sont pas définies et sont ignorées.

Chapitre III :

Conception du contrôleur I2C.

III. Introduction :

Les machines à états finis sont des circuits de logique séquentielle servant exclusivement à générer des signaux de commande. Il existe en effet deux grands types de signaux en électronique :

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

Cette classification des signaux se retrouve au niveau des architectures des systèmes électroniques schématisée dans la figure III.1 où la partie contrôle, générant les commandes, est dissociée de la partie opérative, traitant les données. Les deux parties sont toujours réalisées en logique séquentielle et dans une très grande majorité des cas en logique séquentielle synchrone.

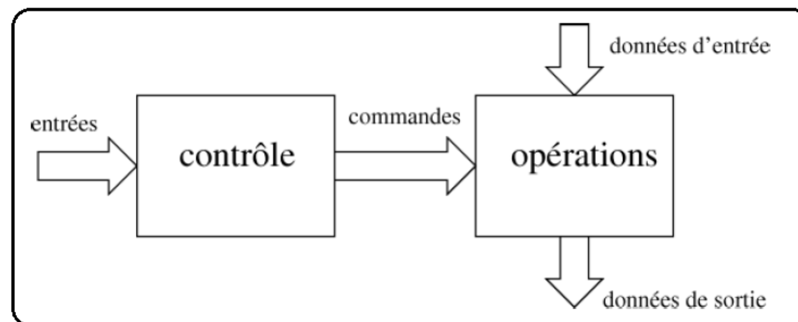


Figure III.1 : Architecture générique d'un circuit électronique.

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

- L'horloge : pour le déroulement des séquences.
- Le Reset : pour l'initialisation du système.

La machine à états finis représente la partie contrôle du système électronique. Les états de la machine à états finis représentent toutes les valeurs que peuvent prendre les variables internes du circuit de logique séquentielle, voir la figure III.2.

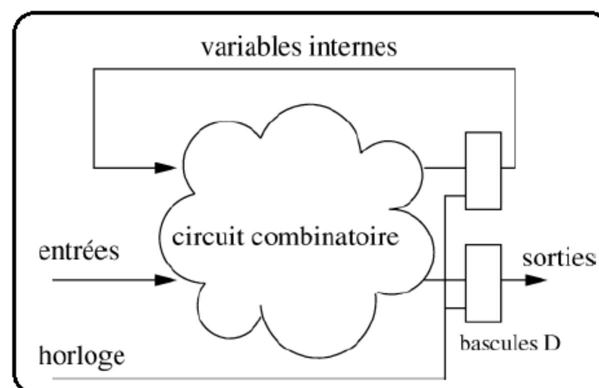


Figure III.2 : Schéma de la machine à états finis générique.

1 Le graphe d'états :

1.1 Représentation graphique du comportement d'une machine à états finis :

Dans une machine à états finis donnée, la loi d'évolution de l'état n'est pas aléatoire. Ces lois sont soigneusement choisies afin que la machine à états finis remplie une fonction précise. Le graphe d'état est l'un des outils les plus utilisés pour la spécification de la machine à état (Entrées, Sorties, fonctionnement souhaité).

Le graphe d'état : est une représentation graphique des états d'une machine à états finis.

Étapes de construction d'un graphe d'état :

1. Chaque état est dessiné sous la forme d'une bulle contenant son nom.
2. Le graphe est complété par des flèches indiquant les transitions possibles entre les états. L'état source, est l'état de départ d'une transition et état destination l'état d'arrivée. Certaines transitions ont le même état pour source et destination, cela signifie que la machine peut rester dans le même état pendant un certain temps (schématisé sur la figure III.3 par T1) voir la Figure III.3.

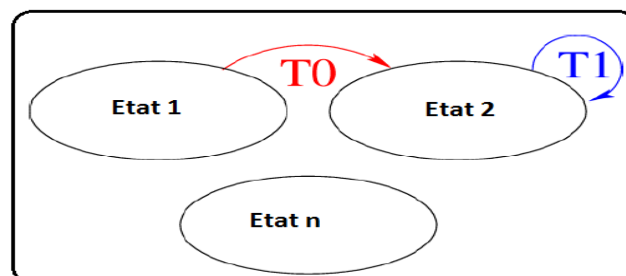


Figure III.3 : Représentation d'un graphe d'état et quelques transitions, état1 :état de départ par T0 et état2 état destination.

À tout instant la machine est à un état possible, c'est ce qui est appelé état courant de la machine. À chaque front montant de l'horloge, la machine emprunte l'une des transitions possibles à partir de son état courant. Elle ne change d'état qu'au prochain front montant de l'horloge par contre une transition est instantanée.

3. Puis vient, l'étape de spécification de la machine en particulier, la loi d'évolution des variables internes (d'un état) en fonction des entrées.
4. La machine fournira des sorties pour chaque état, chaque sortie codée en n-bits (selon le nombre de sorties nécessaires). Les spécifications précisent les valeurs pour chaque état que peut prendre la machine.

1.2 Règles de vérification d'une représentation graphique :

Dans une représentation graphique, l'erreur est toujours possible, si celle-ci est commise, elle se retrouve inévitablement implémenté sur le circuit électronique. Pour qu'un graphe d'états soit satisfaisant, il doit être :

- Complet ou non ambigu.
- Non contradictoire.

Première règle : un graphe d'état complet ; signifie que le comportement est toujours défini. À chaque front montant d'horloge, quel que soit l'état dans lequel se trouve la machine et quelles que soient les valeurs des entrées, l'état suivant doit être connu.
L'équation booléenne de cette règle est comme suit : la somme de toutes les transitions partant d'un état quelconque est toujours vraie.

$$S = \sum_{k=1}^{n} C_k = 1$$

Avec C_1, C_2 ... C_k, ..., C_n les conditions de départ.

Deuxième règle : non-contradiction ; à chaque front montant d'horloge une seule transition est possible. Si plus d'une transition a sa condition associée vraie, le graphe est contradictoire (deux actions incompatibles sont simultanément possibles donc contradiction).
L'équation booléenne de cette règle est comme suit le OU logique de tous les ET logiques de deux conditions associées aux transitions partant d'un état quelconque est toujours faux :

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} CiCj = 0$$

1.3 La composition de la machine à états finis :

1.3.1 Calcul de l'état futur :

En logique séquentielle l'état futur est modifié à chaque front montant de l'horloge. Entre deux fronts montants de l'horloge (pendant une période d'horloge) il reste stable, ce qui donne le temps aux circuits combinatoires qui composent la machine de calculer le prochain état et les sorties. Ce qui met en évidence l'existence d'un circuit combinatoire chargé de calculer le prochain état, appelé état futur, à partir de l'état courant et des entrées de la machine (nommé P1 sur le schéma de la Figure III.4).
Ses entrées sont :

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

Sa sortie :

- L'état futur.

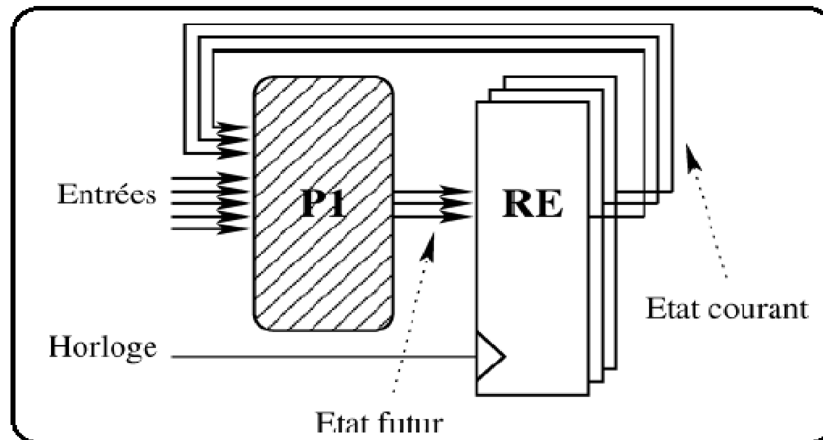


Figure III.4 : Calcul de l'état futur.

Dès que les entrées changent de valeur ou dès que l'état courant est modifié, le circuit P1 commence à calculer l'état futur.

1.3.2 Le registre d'état RE :

Il est composé de plusieurs bascules D. L'horloge est l'horloge générale du circuit électronique dont fait partie la machine. Son entrée est la sortie du circuit P1, c'est l'état futur. Sa sortie, l'état courant, sert d'entrée à P1 mais aussi au circuit destiné à calculer les sorties. Une machine à état est un dispositif dont l'état courant conditionne les états futurs. Dans un tel dispositif, les conditions initiales sont très importantes (lors du démarrage par exemple), ce qui impose un état de départ.

Le RESET, est une entrée d'initialisation grâce à laquelle la machine est forcée à des valeurs connues dès le démarrage. Il existe deux méthodes pour forcer le démarrage avec le RESET :

1. ***Le RESET synchrone :*** Il est pris en compte uniquement sur le front montant de l'horloge. Il agit donc de la même façon que les entrées de données de la machine. Son influence est prioritaire sur les autres. Le circuit P1 possède donc ce signal comme entrée supplémentaire. Lorsque cette entrée est active, l'état futur que calcule P1 est l'état initial. Au front montant d'horloge suivant la machine passe donc dans cet état.
2. ***Le RESET asynchrone :*** Il utilise les entrées Set et Reset des bascules D du registre d'état pour forcer l'état initial. Le branchement de l'entrée Reset sur l'entrée Set des

bascules pour forcer un « 1 » logique, ou sur l'entrée Reset des bascules D pour forcer un « 0 » logique. Les entrées de la partie P1 ne sont pas modifiées. Le graphe d'état non plus, si ce n'est l'indication de l'état de départ par le Reset. Cette solution est donc plus simple à concevoir que la précédente, donne des tailles (en nombre de composants) plus faibles pour des vitesses de fonctionnement plus élevées.

1.3.3 Le calcul des sorties :

La troisième partie d'une machine à états finis est le circuit combinatoire de calcul des sorties. Dans une machine à états ses entrées sont l'état courant et ses sorties sont les sorties de la machine. Dès que l'état courant change, après un front montant d'horloge, ce circuit (symbolisé P2 sur la figure III.5) commence à calculer les sorties caractéristiques du nouvel état. Comme pour le circuit P1, il faut absolument qu'il dispose d'assez de temps pour le faire avant le front montant d'horloge suivant, voir la Figure III.5.

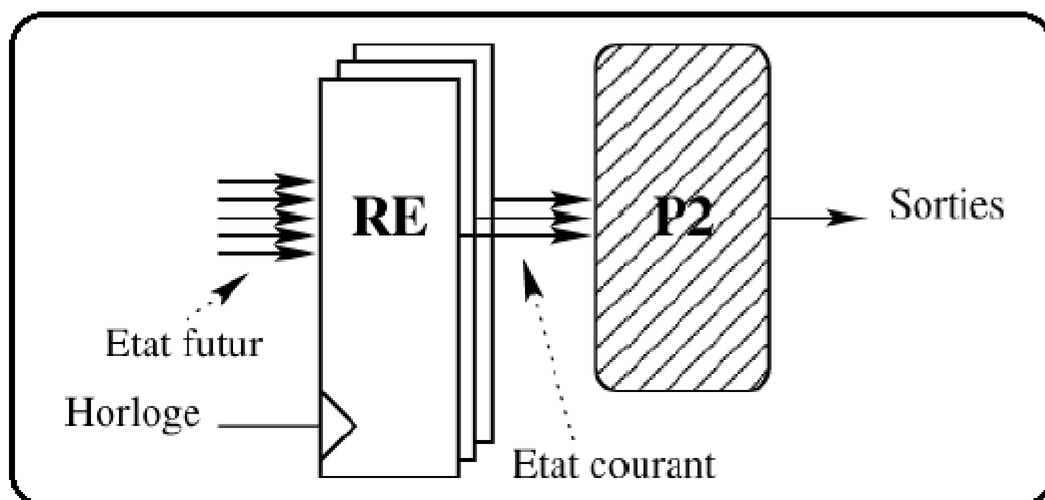


Figure III.5 : Calcul des sorties.

1.4 Le codage des états :

1.4.1 Représentation des différents états sous forme de mots binaires :

Lors de la construction d'un graphe d'états les bulles sont remplies avec des noms symboliques, représentatifs pour le concepteur (arrêt, marche, monter …etc). Ces états doivent être représentés par des mots binaires. Le codage est la représentation binaire de tous les états.

Dans un graphe d'états tous les mots doivent avoir la même taille. La taille du mot binaire est fonction du nombre d'états à coder. Un mot de « n » pour coder 2^n états, un mot de trois bits offre la possibilité de coder huit états au maximum.

La partie combinatoire de la machine qui calcule l'état futur en fonction des entrées et de l'état courant est très largement influencé par le codage des états. Donc sa taille (en nombre de composants utilisés) en dépend également. Elle possède $N_e + N_b$ entrées et N_b sorties (N_e est le nombre d'entrées de la machine et N_b le nombre de bits choisi pour coder les états, voir la Figure III.6.

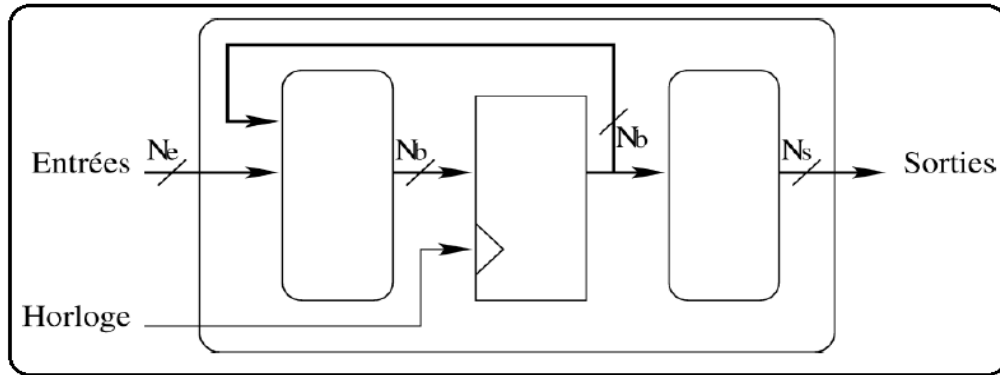


Figure III.6 : schéma d'une machine à états finis avec le nombre de bits nécessaires.

Le nombre de fonctions booléennes calculées est donc égal à N_b et chacune de ces fonctions possède $N_e + N_b$ entrées. Donc, il faut coder les états avec le moins de bits possible pour que cette partie combinatoire soit la plus petite possible. Mais ce n'est pas une règle. En effet, car une fonction booléenne de quatre variables peut être plus simple qu'une autre de deux variables :

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

est plus simple que :

$$G(A_0,A_1) = A_0 \text{ XOR } A_1$$

Il se pourrait que cinq fonctions booléennes simples vaillent mieux que trois complexes. La partie combinatoire qui calcule les sorties en fonctions de l'état courant possède N_b entrées et N_s sorties (où N_s est le nombre de sorties de la machine). Elle calcule donc N_s fonctions booléennes de N_b entrées. Là encore, la solution qui se traduit par une taille minimum n'utilise pas nécessairement un codage des états sur un nombre de bits minimum.

La taille du registre d'état est directement liée au nombre. Une solution optimale au sens de la taille pour la partie combinatoire de la machine qui calcule l'état futur a peu de chances d'être également la meilleure pour la partie combinatoire qui calcule les sorties.

1.4.2 Méthodes de choix du codage :

Pour le choix du codage, la meilleure des solutions est un compromis. En effet, la taille est un critère de sélection, mais il n'est pas le seul. D'autres critères existent, tels que : la vitesse de fonctionnement, la consommation ou la simplicité de conception. Selon l'objectif fixé, les stratégies de codage seront différentes. Parmi celles-ci il y a :

1. ***Le codage adjacent :*** il utilise un nombre de bits minimum et se caractérise par le fait que le passage d'un état à un autre ne modifie qu'un seul bit du registre d'état. Il n'est pas toujours possible de trouver un tel codage, cependant il faut essayer de s'en approcher en réduisant le nombre de transitions modifiant plus d'un bit du registre d'état. L'intérêt d'un tel codage n'est pas systématique, cependant il donne souvent de bons résultats en taille et en vitesse pour la partie combinatoire qui calcule l'état futur. Elle se trouve simplifiée par la faible agitation des bits représentant l'état.
2. ***Le codage « one-hot » :*** il utilise un nombre de bits égal au nombre d'états. Chaque état est représenté par un mot binaire dont tous les bits sauf un valent 0. Ce codage donne les machines les plus simples à concevoir. Il est également parfois intéressant en vitesse et en surface malgré le handicap dû à la taille du registre d'état.
3. ***Le codage aléatoire :*** il consiste à coder les états sur un nombre de bits minimum sans aucune autre préoccupation que d'éviter que deux états aient le même code. Les résultats en termes de surface, vitesse ou difficulté de conception sont imprévisibles, mais peuvent parfois être meilleurs que ceux produits par les deux autres stratégies.

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

2. Conception de la machine à états finis régissant le bus I2C :

2.1 Caractéristiques du contrôleur I2C :

Le contrôleur I2C supporte les fonctions suivantes :

- Une interface avec un microcontrôleur ou microprocesseur.
- Deux modes de fonctionnements : mode maître et mode esclave.
- Supporte le branchement sur un bus multi maîtres (traitement d'arbitrage).
- Génération d'une interruption lors de la perte d'un arbitrage et basculement immédiat vers le mode esclave.

- Génération et détection des signaux START et STOP.
- Génération d'un START répété.
- Détection et génération d'un acquittement ACK.
- Lecture de l'état du bus (occupé ou libre).
- Fréquence de travail à 100kHz.

3.2 Description des signaux :

Le tableau suivant est un descriptif des signaux du contrôleur I2C (voir la Figure III.7).

Nom	Direction	Description
SDA	Bidirectionnelle	Serial Data du contrôleur I2C.
SCL	Bidirectionnelle	Serial Clock du contrôleur I2C.
DATA_BUS [7 :0]	Entrée	Bus de données du microcontrôleur.
ADDR_BUS [23 :0]	Entrée	Bus d'adresse du microcontrôleur.
AS	Bidirectionnelle	Assertion d'adresse : indique la présence d'adresse valide sur le bus d'adresse ADDR_BUS. Actif à l'état bas.
DS	Entrée	Assertion de données : indique la présence de données valide sur le bus de données DATA_BUS, actif à l'état bas.
R_W	Entrée	Lecture /Écriture : ce bit indique le sens du transfert, « 1 » pour lecture et « 0 » pour écriture.
DTACK	Sortie	Acquittement des Données : acquittement du transfert de données, indique que le contrôleur I2C a placé des données valides sur le bus de données pour un cycle de lecture ou le contrôleur I2C a reçu les données pour un cycle d'écriture. Actif à l'état bas.
IRQ	Sortie	Interrupt ReQuest : demande d'interruption, actif à l'état bas.
MCF BIT	Sortie	Bit D'Indication de Transfert : durant le transfert d'un octet, ce bit est mis à « 0 ». Il est positionné à « 1 » par le front descendant de la neuvième impulsion d'horloge du transfert d'un octet. Ce bit est utilisé pour signaler au microcontrôleur la fin du transfert d'un octet.
CLK	Entrée	Horloge : l'horloge du système.

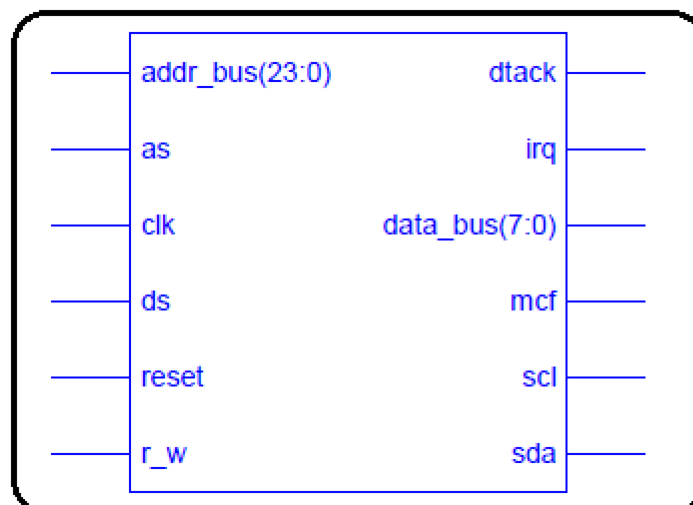


Figure III.7 : illustration des entrées sorties du contrôleur I2C.

3.3 Disposition du contrôleur I2C dans un circuit :

Le contrôleur I2C est une interface de communication entre un microcontrôleur d'un côté, celui-ci contrôle et fournit les données nécessaires pour le bon déroulement des échanges. Ce qui nécessite une interface entre le contrôleur I2C et le microcontrôleur ou le microprocesseur. Cette interface dispose de trois signaux : Adress, Data et Control. D'un autre côté, l'interface I2C maître /esclave. Cette interface gère les échanges entre le contrôleur I2C et le bus I2C (voir la Figure III.8).

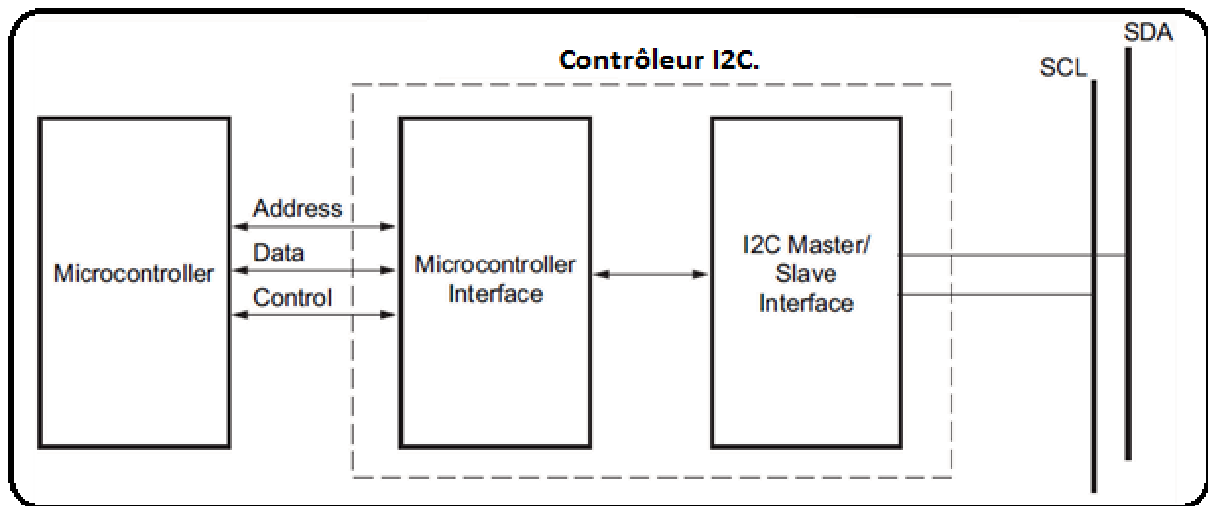


Figure III.8 : disposition du contrôleur I2C sur circuit.

3.4 Schéma block du contrôleur I2C :

Le schéma block suivant (Figure III.9) illustre les différents registres internes du contrôleur I2C et leurs interconnexions. Il comporte deux interfaces : une interface qui permet de communiquer avec un microcontrôleur ou microprocesseur et une interface qui permet de communiquer via le bus I2C.

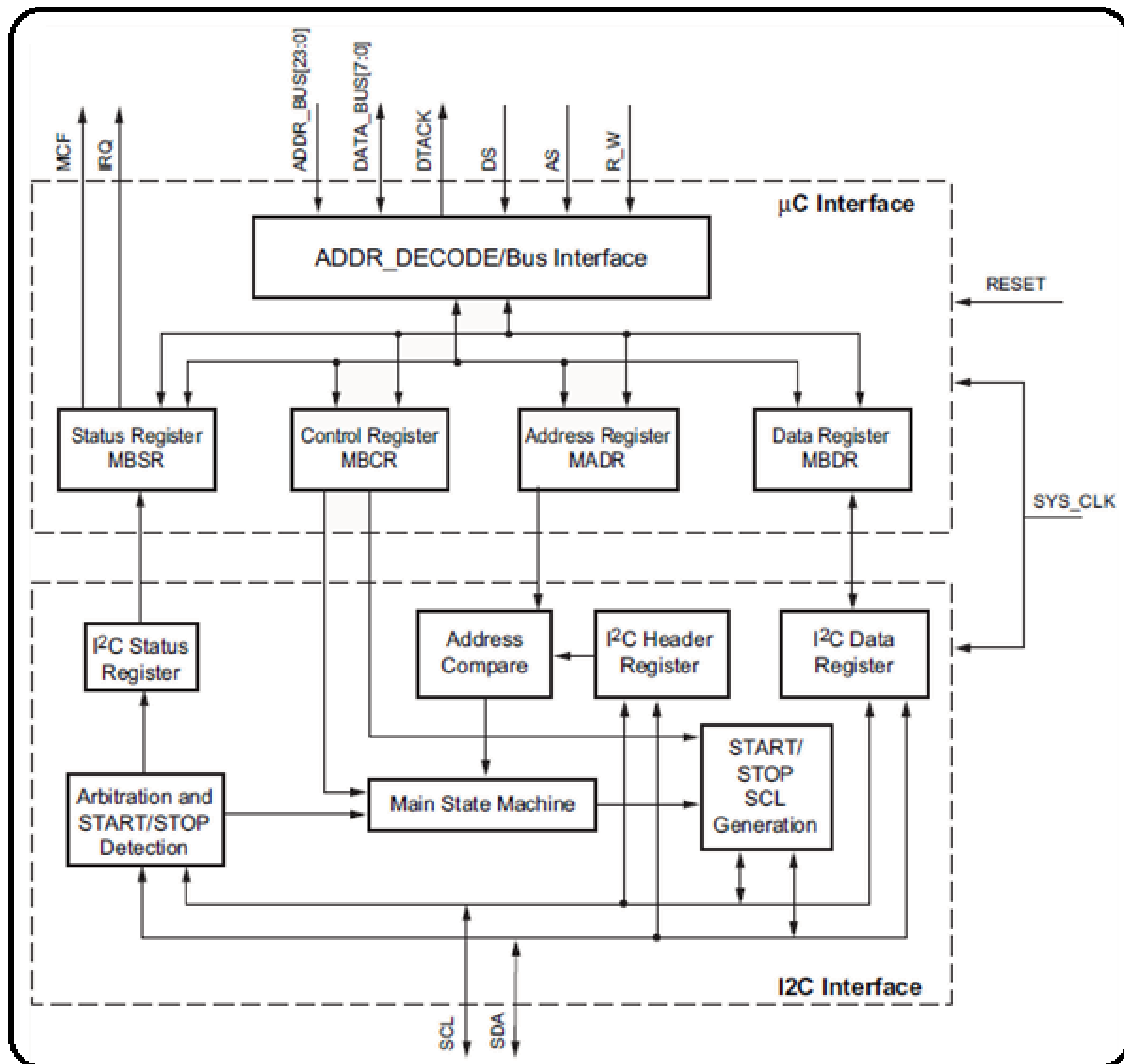


FIGURE III.9 : Schéma blocks du contrôleur I2C.

3.5 Interface microcontrôleur :

3.5.1 Interface logique de décodage d'adresse :

La Figure III.10, est la machine à états finis qui gère l'interface logique entre le microcontrôleur et le contrôleur I2C. Pour que cette machine à états finis commence le décodage d'adresses, le microcontrôleur doit fournir certaines informations pour le contrôleur I2C.

Le microcontrôleur indique à l'interface microcontrôleur du contrôleur I2C le sens du transfert (lecture /écriture) avec le bit R_W. Il place l'adresse sur le bus d'adresse ADDR_BUS (bus parallèle), puis il affirme avec le bit AS (assertion d'adresse) comme quoi une adresse valide est présente sur le bus d'adresse. Simultanément, il place les données sur le

bus de données DATA_BUS (bus parallèle) puis positionne le bit DS (assertion de données), indiquant pour l'interface microcontrôleur qu'une donnée valide est présente sur le bus de données.

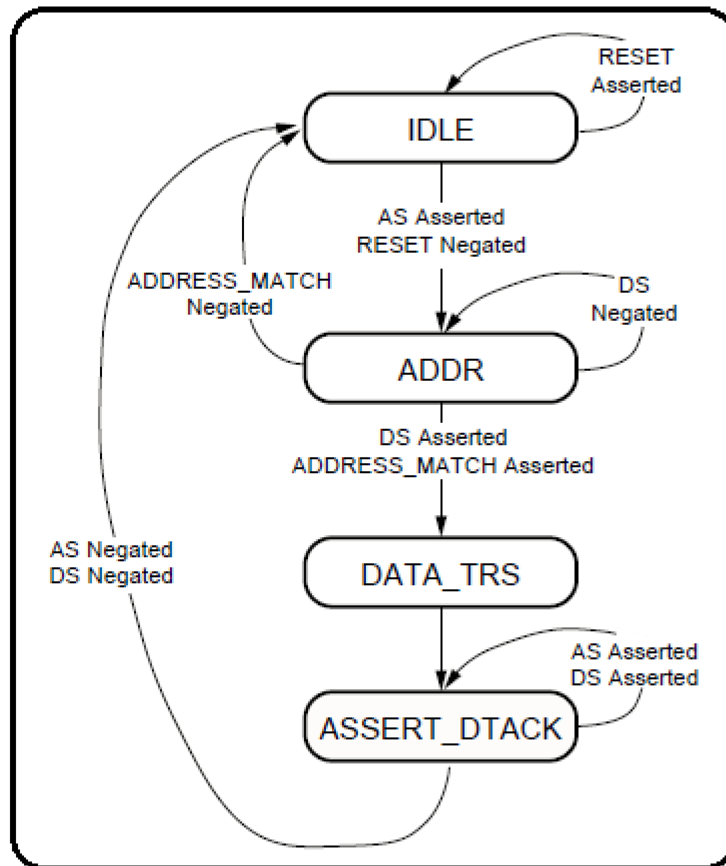


Figure III.10 : Machine à états finis de l'interface du microcontrôleur avec le contrôleur I2C (machine d'interface uC).

Séquencement de la machine d'interface uC :

Cette machine à états finis a deux rôles, celui d'initialiser les registres internes de l'interface microcontrôleur (à l'allumage ou après un reset) et celui d'assurer la communication entre le microcontrôleur et le contrôleur I2C.

Une fois toutes les informations nécessaires pour le décodage d'adresse sont présentes sur les différents signaux d'interface entre le microcontrôleur et le contrôleur I2C. La machine à états finis commence le décodage.

Après l'assertion d'adresse (AS=0) la machine à états finis évolue de l'état d'attente IDLE vers l'état ADDR (décodage d'adresse), pour décoder l'adresse et voir quel est le registre adressé. À ce stade, les registres internes sont activés. Si la bonne adresse est insérée et le bit DS = 0 (confirmation de présence de données), la machine à états finis évolue vers l'état DATA_TRS (transmission de données). Si c'est un cycle de lecture, le contrôleur I2C place

les données sur le bus de données. Si c'est un cycle d'écriture, les données sont retenues dans le registre adressé par le microcontrôleur. Systématiquement, la machine à états finis évolue vers l'état ASSERT_DTACK (assertion d'acquittement de données). L'état ASSERT_DTACK s'acquitte des données en positionnant le signal DTACK à l'état bas, le microcontrôleur positionne le signal R_W en lecture à la fin du transfert le microcontrôleur inverse les signaux DS et AS en les positionnant à l'état haut. C'est alors que la machine à états finis revient vers l'état d'attente IDLE pour finir le cycle et remet le signal DTACK à l'état haut.

3.5.2 Les registres du contrôleur I2C :

L'adresse de base utilisée pour le décodage d'adresse est mentionnée dans le code VHDL par la constante BASE_ADRESSE. L'adresse de base est codée sur 16 bits, les plus significatifs du bus d'adresse qui est codée sur 24 bits. Les 8 bits inférieurs des bits d'adresse indiquent quel registre interne est adressé par le microcontrôleur. Les registres internes offrent un accès en lecture et en écriture pour le microcontrôleur, ce sont des registres qui permettent de communiquer entre le microcontrôleur et le contrôleur I2C. Le tableau suivant est celui des registres internes du contrôleur I2C :

Registre	**Description**
MADR	Registre d'adresses.
MBCR	Registre de contrôle.
MBSR REG	Registre d'état.
MBDR RDD	Registre de données.

1. ***Registre d'adresse :*** il contient l'adresse esclave spécifique qui va être utilisé par le contrôleur I2C. C'est un registre accessible en lecture et en écriture, codé sous 7-bits.
2. ***Registre de contrôle :*** il contient des données de contrôle de l'interface I2C.

Position du bit	**Description.**
7	Ce bit contrôle tous les autres bits en activant ou désactivant le contrôleur I2C.
6	Activation d'interruption, actif à l'état haut.
5	Sélection du mode maître ou esclave.
4	Sélection de la direction du transfert : « 1 » émetteur, « 0 » récepteur.
3	Ce bit spécifie l'envoie d'un acquittement ACK ou non NACK durant un cycle d'acquittement. « 1 » non-acquittement NACK, « 0 » envie d'acquittement ACK.
2	L'écriture d'un « 1 » sur ce bit génère un START répété par le contrôleur I2C, s'il est le maître il est toujours lu à « 0 ». S'il reçoit un START répété au mauvais moment ceci dit que le maître a perdu lors d'un arbitrage.
1-0	Non utilisés.

3. ***Registre d'état :*** ce registre contient les informations sur l'état du contrôleur I2C. C'est un registre à lecture seule et il est effacé après un reset.

Position du bit	Description
7	Quand un octet est en cour de transfert, ce bit est mis à zéro. Il est mis à « 1 » par le front montant de l'horloge SCL durant le bit d'acquittement et il est seulement à l'état haut durant la période haute de cette impulsion d'horloge. « 1 » transfert complet, « 0 » transfert en cours.
6	Quand l'adresse reçue via le bus I2C est celle de l'esclave contenue dans le registre d'adresse, le contrôleur bascule en mode esclave et positionne ce bit.
5	Détection de l'état du bus, il est à « 1 » si une condition START est détectée alors le bus est occupé, mis à « 0 » si une condition STOP est détectée.
4	Ce bit est positionné en cas de perte d'arbitrage.
3	Non utilisé.
2	Bit d'indication de direction de transfert en mode esclave. « 1 » lecture, « 0 » écriture.
1	Ce bit est positionné si une interruption est en cours.
0	Bit d'indication de réception d'acquittement.

4. ***Registre des données :*** ce registre contient des données venant de (allant vers) le bus I2C. Il est implémenté en deux registres de 8 bits à la même adresse. Un pour les données transmises via le bus I2C et l'autre pour celles reçues par le bus I2C. En émission, les données écrites sur ce registre sont en sortie sur le bus I2C. En réception, ce registre contient les données reçues du bus I2C.

3.6 L'interface logique du contrôleur I2C :

L'interface logique du contrôleur I2C en sept processus différents. Les bits de contrôle de l'interface microcontrôleur déterminent le comportement de ces processus.

Lors ce que le contrôleur I2C est configuré en mode maître, le signal en sortie du contrôleur I2C sur la ligne SDA est comparé au signal présent sur cette ligne pour savoir si le contrôleur I2C contrôle le bus ou non. Le signal sur la ligne SDA est vérifié seulement durant la demi-période haute de tout les cycles de transfert sauf durant un cycle d'acquittement ceci pour s'assurer que les conditions START et STOP ne sont pas générées au mauvais moment.

Si le signal sortant sur la ligne SDA et celui qui s'établit sur cette ligne ne sont pas les mêmes, c'est alors le contrôleur I2C se rend compte qu'il a perdu l'arbitrage et il positionne le bit concerné. Le contrôleur I2C bascule en mode esclave. Le contrôleur I2C ne génère pas de START lors ce que le bus est occupé.

3.6.1 Détection des conditions START et STOP :

Ce processus surveille les lignes SDA et SCL du bus I2C pour observer les conditions START et STOP. Quand une condition de départ START est détectée, le bit indiquant bus occupé est positionné, il y reste jusqu'à ce qu'il détecte la condition d'arrêt STOP.

Les signaux DETECT_START et DETECT_STOP sont générés par ce processus pour qu'ils soient utilisés par d'autres processus.

3.6.2 Génération de SCL, SDA, START et STOP :

Ce process génère les signaux SCL et SDA en mode maître sur le bus I2C. La fréquence d'horloge sur la ligne SCL est de 100 khz, elle est déterminée en divisant l'horloge en entrée en fonction de l'horloge du système (voir la figure III.11).

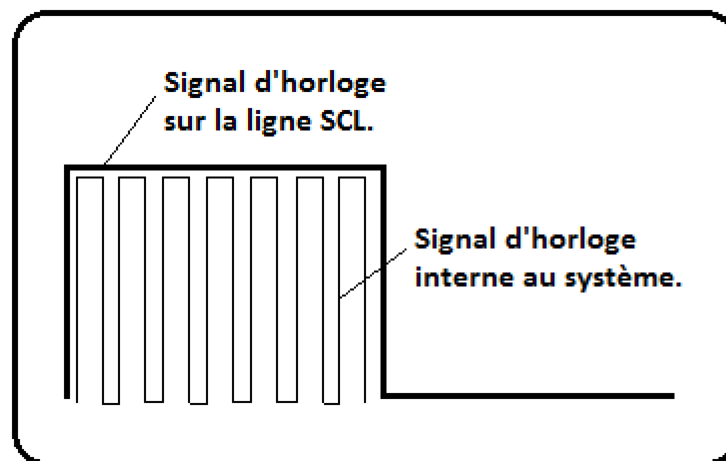


Figure III.11 : Principe du calcul de la fréquence d'horloge sur la ligne SCL a base de la fréquence interne du système.

Le nombre d'impulsions nécessaire pour générer un signal d'horloge de 100 khz en fonction de la fréquence d'horloge du système est déterminé d'après l'équation suivante :

$$x = \frac{f\ sys}{5 * 100\text{Khz}}$$

- X : le nombre d'impulsions pour générer la fréquence de 100Khz, codé sous la constante CNT_100 Khz.
- f sys : la fréquence du système. Elle est divisée par 5*100Khz, ceci pour la raison qui fait que la fréquence du système doit être au moins 4 fois supérieure à celle du bus [*ROB05*].

De la même façon, les constantes START_HOLD et DATA_HOLD, contiennent le nombre d'impulsions d'horloge nécessaire pour le maintien d'un bit sur la ligne SDA après la génération de la condition START et après que les données soient mises sur le bus. La figure III.12 est la machine à états finis qui génère les signaux SCL et SDA en mode maître. Si le bus est occupé, SCL et SDA sont aux niveaux par défauts. La machine à états finis suivante génère le contrôle pour le comptage du système d'horloge pour la génération des signaux en mode maître (voir la Figure III.12).

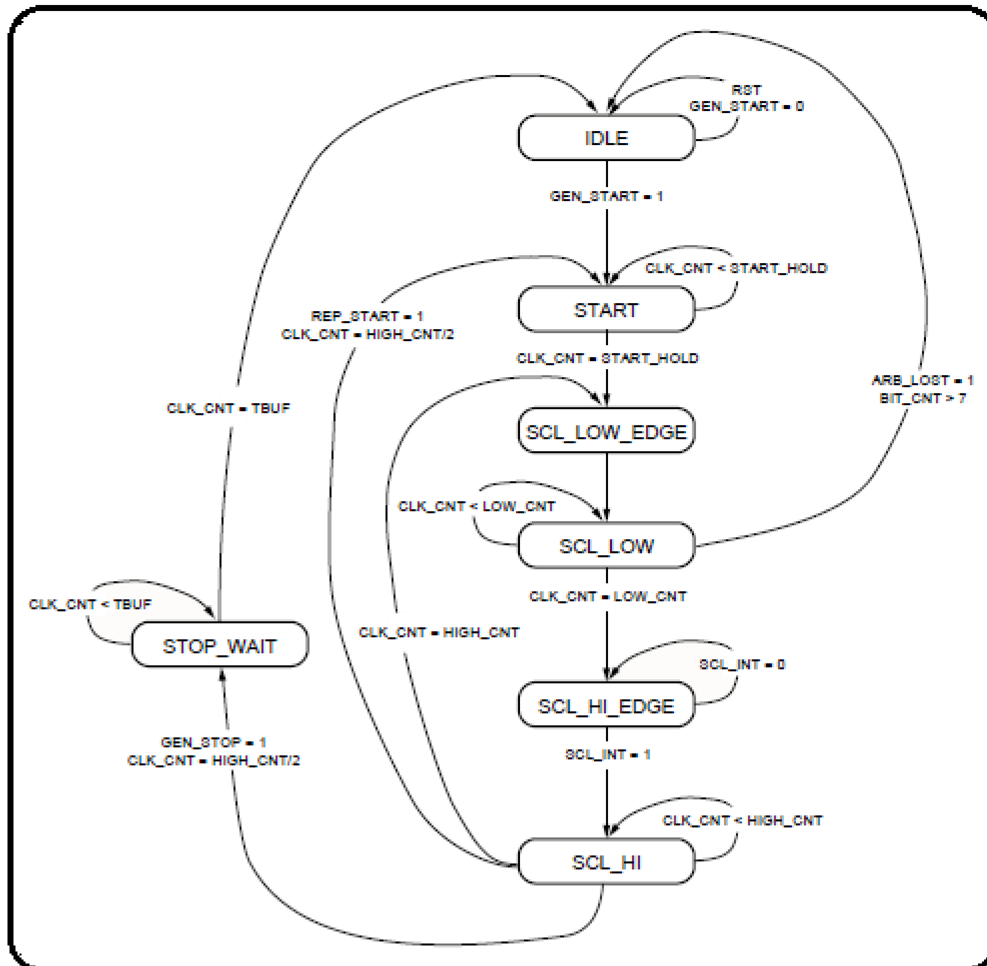


Figure III.12 : Machine à états finis qui génère SCL, SDA, START et STOP (machine de génération des signaux) [XIL02].

Le signal interne SDA, en sortie du contrôleur I2C, est généré par cette machine à état pour START et STOP ou du registre de données quand le contrôleur I2C est en mode transmission. Les lignes SDA et SCL sont à collecteurs ouverts, néanmoins, elles ne sont commandées que pour le « 0 ». Quand un « 1 » est en sortie sur un des signaux, le contrôleur I2C active le buffer sur l'état haute impédance ce qui est interprété comme un « 1 » par le bus. Le circuit logique du système positionnera les signaux internes des SDA et SCL sur « 1 ou 0 ». Ces

signaux internes contrôleront les sorties de chaque buffer de sortie sur la bonne valeur, en sortie, en entrée, état haut et état bas (voir la figure III.13).

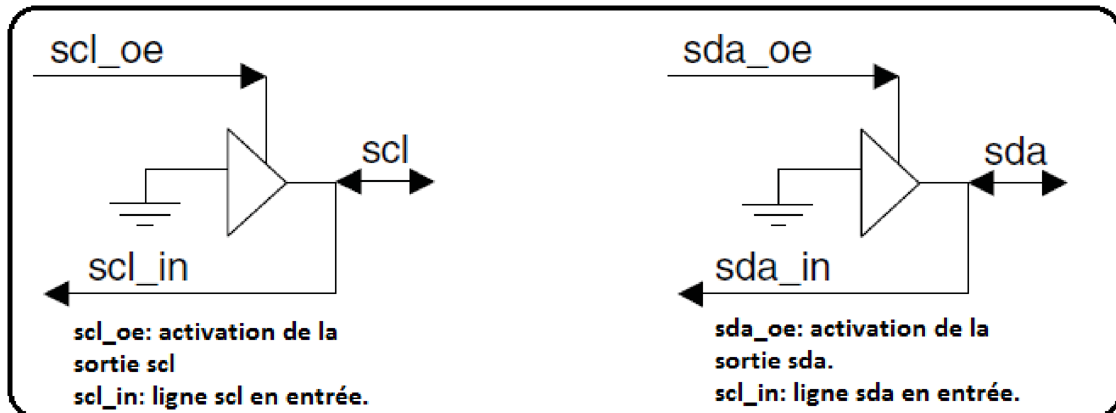


Figure III.13 : Buffer d'entrée /sortie sur le bus I2C.

Séquencement de la machine génération des signaux SCL, SDA, START et STOP :

L'état d'attente IDLE, SCL et SDA sont à l'état haute impédance, autorisant l'importe quel maître de contrôler le bus. Une fois une requête est reçue pour générer une condition de départ START, le contrôleur I2C est en mode maître, le bus n'est pas occupé, la machine à états finis évolue vers l'état START.

L'état START maintient la ligne SCL à l'état haut, mais force la ligne SDA à l'état bas pour générer une condition de départ valide. Le compteur d'impulsions d'horloge est amorcé et la machine à états finis reste dans cet état jusqu'à ce que le temps minimum de maintien pour la génération d'un START valide est atteint (voir la figure III.14). À ce niveau le prochain état est SCL_FRONT_BAS.

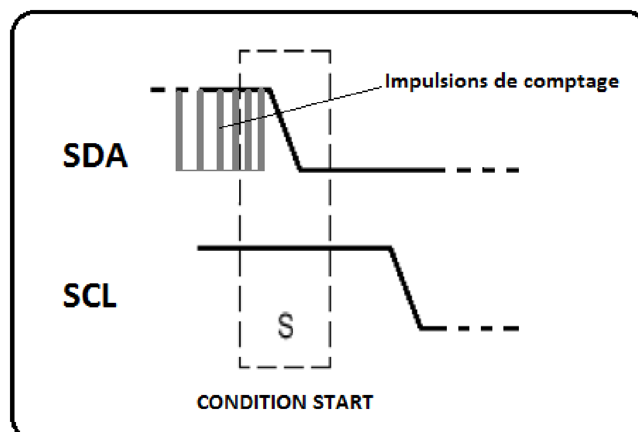


Figure III.14 : Illustration des impulsions de comptage pour générer un niveau valide, dans ce cas une condition de START.

L'état SCL_FRONT_BAS, crée tout simplement le front descendant sur SCL et réinitialise le système de comptage d'impulsions d'horloge CLK_CNT. Au prochain front d'horloge, la machine à états finis évolue vers l'état SCL_BAS. Dans cet état, la ligne SCL est maintenue à l'état bas et le système de comptage d'impulsions d'horloge s'incrémente. Si le signal START répété (REP_START) survient alors la ligne SDA sera positionnée à l'état haut, si la génération d'un STOP (GEN_STOP =1) survient, la ligne SDA sera positionnée à l'état bas et y reste jusqu'à validation du STOP.
Au terme de la demi-période basse de la ligne SCL, la machine à état revient vers l'état de repos IDLE, si le contrôleur I2C a perdu l'arbitrage (ARB_LOST=1) et le transfert de l'octet est terminé le signal d'horloge sur la ligne SCL est toujours généré pour s'assurer de la continuité sur la ligne SCL. Sinon, la machine à états finis évolue vers le prochain état : l'état SCL_HI_EDGE.

L'état SCL_HI_EDGE, génère un front montant sur la ligne SCL en positionnant SCL à « 1 ». La machine à état n'évoluera pas vers l'état SCL_HI avant que le signal demandé sur la ligne SCL est aussi haut, ceci afin de respecter la synchronisation d'horloge tel qu'il y est fait référence dans le protocole I2C. Ce qui fait, l'état SCL_HI_EDGE, joue le rôle d'un état d'attente à l'état haut quand l'horloge sur la ligne SCL est synchronisée.

L'état SCL_HI, enclenche le système de comptage d'impulsions d'horloge pour mesurer la durée de la demi-période haute du signal sur la ligne SCL. Si un START répété ou un STOP survient, la machine à états va évoluer vers l'état approprié après la moitié de la durée SCL haute, c'est alors la ligne SDA peut basculer correctement. Si l'une de ces conditions survient, alors la machine à états finis évolue vers l'état SCL_LOW_EDGE au terme de la période haute de la ligne SCL.

L'état STOP_WAIT est utilisé pour s'assurer que le temps de maintien après la condition STOP est achevé.

3.6.3 La machine à états finis principale du contrôleur I2C :

La machine à états finis suivante (voir la figure III.15) est la machine à états finis principale de l'interface I2C. Cette machine à états finis est la même pour le mode maître et le mode esclave. Pour chaque état, le mode est vérifié pour déterminer les bonnes valeurs de sortie et les prochaines valeurs d'état. Ceci permet une permutation immédiate du mode maître au mode esclave suite à une perte d'arbitrage et le contrôleur I2C est adressé en tant qu'esclave.

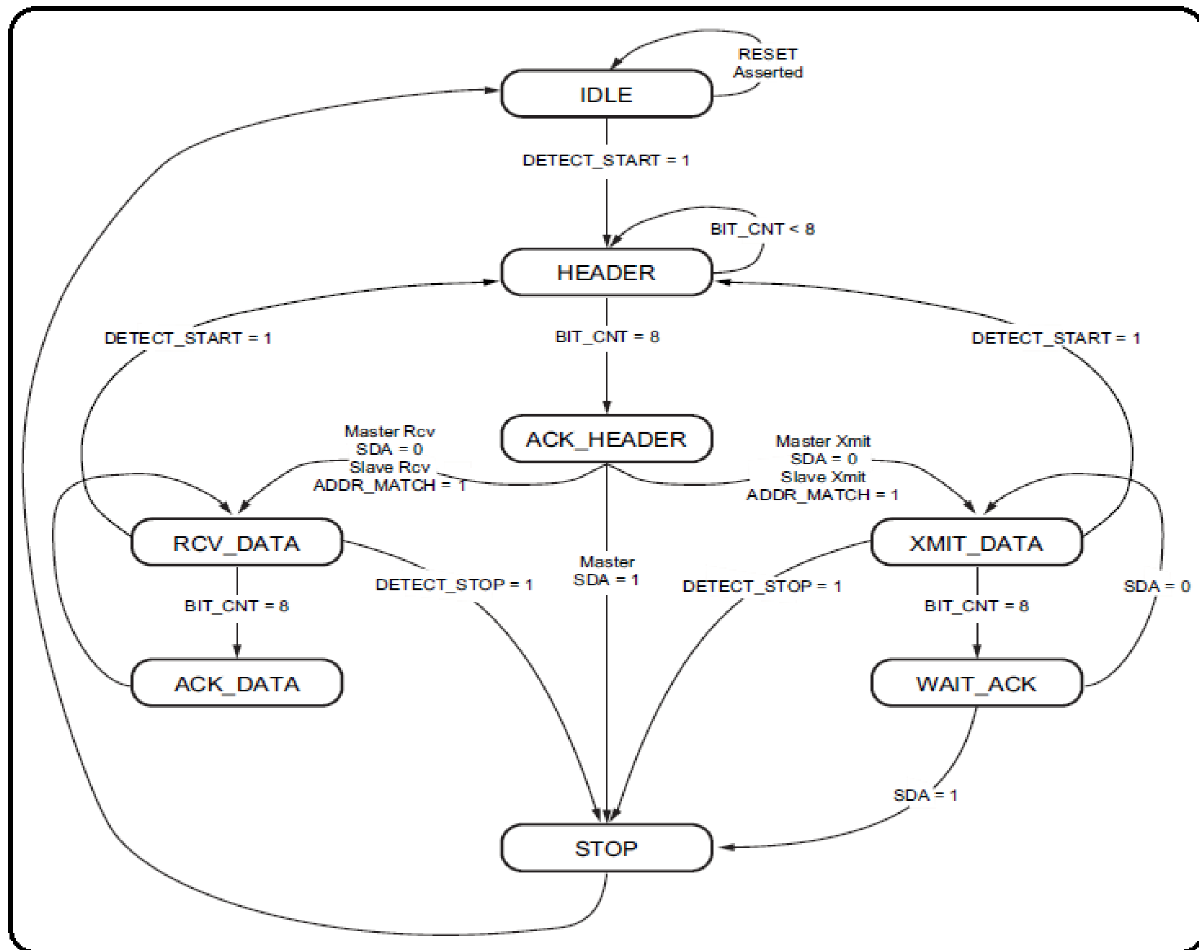


Figure III.15 : Machine à états finis principale de l'interface I2C.

Cette machine à états finis utilise et contrôle un compteur qui compte les bits reçus sur le bus I2C. Le comptage est enregistré dans le signal BIT_CNT. Elle contrôle aussi deux registres à décalage, un pour enregistrer les données reçues à l'état HEADER et l'autre enregistre les données reçu ou à transmettre via le bus I2C.

L'état IDLE. Le circuit de détection de la condition START surveille en permanence les lignes SDA et SCL, elle détecte cette condition même si elle vient du contrôleur I2C lui-même ou d'un autre maître connecté sur le bus. Une fois un signal START est détecté, la transition de la machine à état se fait de l'état d'attente IDLE vers l'état HEADER.

L'état de retenue HEADER est l'état où le contenu du contrôleur I2C est transmis sur le bus depuis le registre des données en mode maître. Dans cet état, les données I2C en retenue dans l'état HEADER sont enregistrées dans le registre à décalage en sortie sur le bus I2C et elles sont transmises instantanément via le bus. Une fois tous les huit bits sont transmis, la machine à états finis évolue vers l'état ACK_HEADER.

L'état ACK_HEADER, le contrôleur I2C observe l'acquittement sur la ligne SDA s'il est en mode maître pour voir si l'esclave adressé s'acquitte de la donnée. Si l'esclave adressé ne s'acquitte pas, la machine à états finis évolue vers l'état STOP. L'état STOP, enclenche le générateur des signaux SCL, SDA, START, STOP pour générer le STOP. Si l'esclave s'acquitte de l'adressage, la machine à états finis évolue vers l'état approprié en fonction du sens de transfert demandé par le maître. Si le maître demande une lecture, il recevra des données donc il évoluera vers l'état RCV_DATA. Si le maître demande une écriture, il émettra des données, il évoluera vers l'état XMIT_DATA.
Le registre à décalage en entrée du contrôleur I2C est constamment comparé avec l'adresse I2C placée dans le registre d'adresse. Si ces valeurs correspondent dans l'état ACK_HEADER, le contrôleur I2C est adressé comme esclave et il bascule immédiatement en mode esclave. Le registre d'état, note immédiatement ce changement d'état en positionnant le bit concerné.

L'état RCV_DATA, charge les données reçues depuis le bus I2C dans le registre à décalage pour transfert vers le microcontrôleur. Une fois la totalité de l'octet est reçue, la machine à états finis évolue vers l'état d'acquittement de données ACK_DATA. Dans l'état ACK_DATA, en mode maître, l'indication que l'esclave a transmis le nombre requis de bits lors du dernier transfert n'est pas acquitté (NACK). La machine à états finis quitte ces deux états (RCV_DATA et ACK_DATA) si une condition STOP est détectée, sinon la transition entre ces deux états continue.

L'état XMIT_DATA, charge les données du registre à décalage de l'état HEADER sur la ligne SDA, une fois l'octet est totalement émis, la machine à état bascule vers l'état d'attente d'acquittement WAIT_ACK. Si un acquittement est reçu, la machine à états finis revient vers l'état XMIT_DATA pour transmettre le prochain octet de données, ainsi de suite jusqu'à ce qu'une condition STOP survient ou pas d'acquittement.

L'état STOP, génère une condition d'arrêt si le contrôleur I2C est en mode maître. Le prochain état est toujours l'état d'attente IDLE et l'activité de l'I2C est terminée pour cet échange.

3.7 Présentation du logiciel de simulation ModelSim:

ModelSim est un logiciel de simulation pour les langages HDL tel que le VHDL, développé par Mentor Graphique. C'est un outil informatique permettant de compiler un code VHDL et

en introduisant les TESTBENCH, il permet de simuler le code et de le debugger en cas de dysfonctionnement. La Figure III.16 illustre les étapes de traitement du logiciel et les fichiers d'entrée nécessaire à son fonctionnement et les fichiers de sortie [*MEN09*].

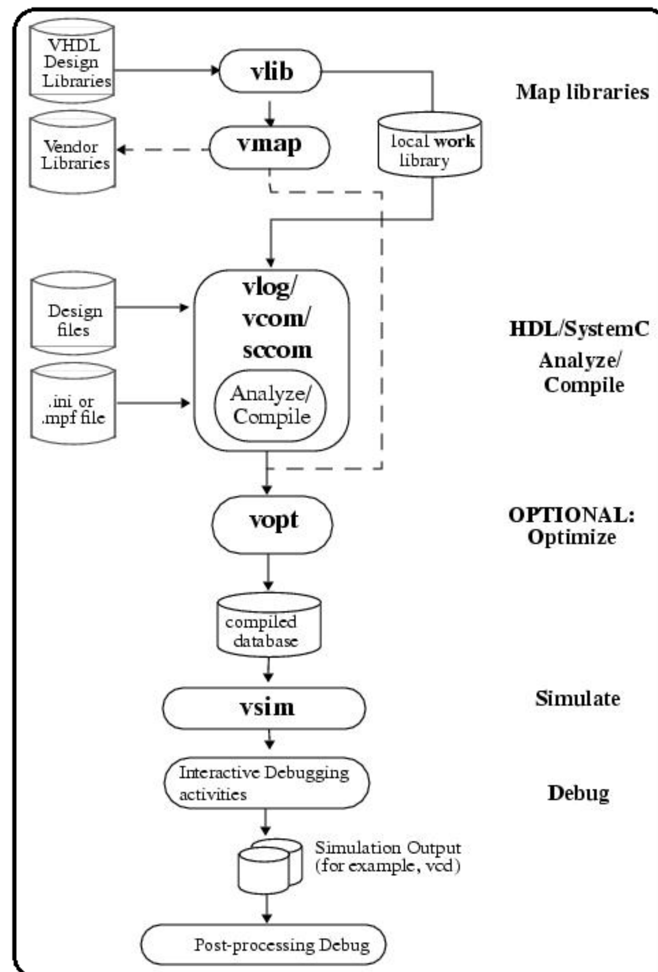


Figure III.16 : Structure du logiciel et son flot de données (séquence d'exécution des taches).

Map Libraries : Après chargement du code VHDL, le sous-programme VLIB crée une librairie de travail appelée par défaut WORK. Le sous-programme VMAP, crée des liens entre les différents fichiers librairies (map) en ajoutant les librairies STANDARD et celles qui peuvent être fournie par des tierces.

Analyse et Compilation : Le programme analyse le code d'un point de vue structurel et voie s'il n'y a pas d'erreurs du genre variable non déclaré ou formats de données incompatibles …etc. Si l'analyse du code s'avère correcte, le logiciel compile le code.

Optimize : L'optimisation est une étape facultative mais conseillée, elle améliore le temps d'exécution de la simulation [*MEN09*].

Simulation : Le programme simule le fonctionnement du code à des instants différents. C'est une simulation appelée RTL (Register Transfert Level).

Debug : Si le code ne fonctionne pas comme souhaité, le debuggage est une étape importante car elle permet de déceler les problèmes éventuels et en conséquence corriger le code source.

Interface graphique du logiciel ModelSim :

La figure III.17 illustre l'interface graphique de ModelSim. C'est une interface qui permet de visualiser les simulations dans une fenêtre appelée WAVE. Le programme permet aussi l'exportation des formes d'ondes pour visualisation indépendamment du programme ou sur un support papier.

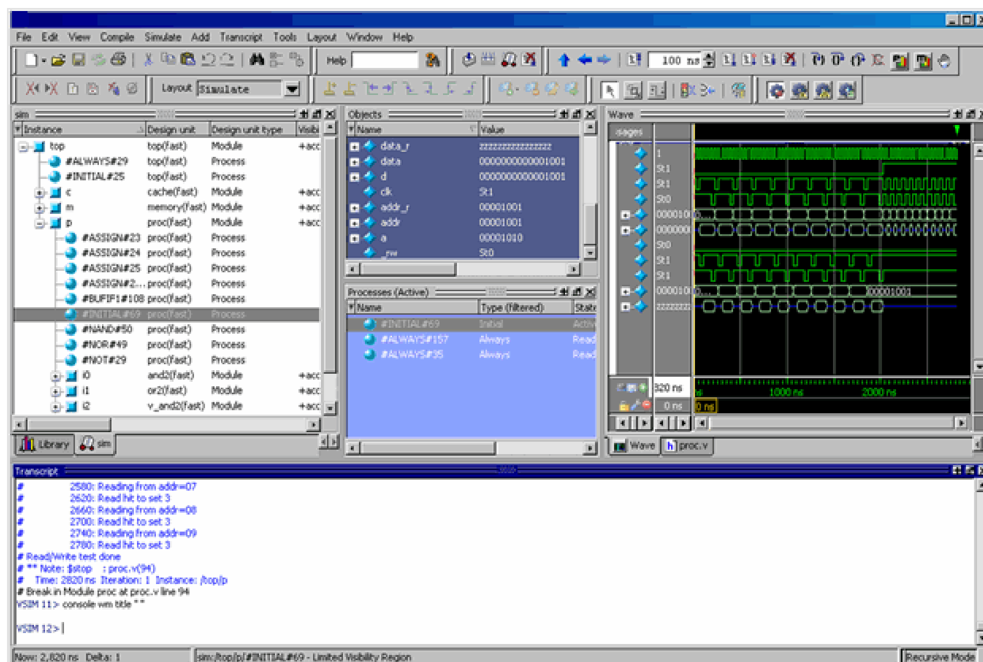


Figure III.17 : Interface graphique du logiciel ModelSim de Mentor Graphics.

3.8 Simulation du code VHDL :

Pour simuler le code VHDL (voir annexe I), en mode maître et en mode esclave, l'usage du même code pour implémenter deux contrôleurs I2C permet de simuler les deux modes simultanément. L'un prend le contrôle du bus donc il est en mode maître et l'autre est adressé par le maître donc il est en mode esclave (voir la Figure III.18).

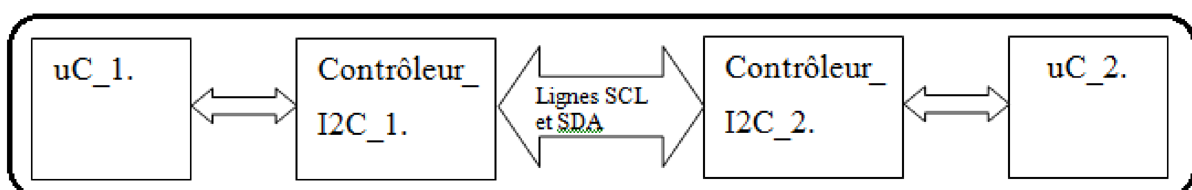


Figure III.18 : Disposition des contrôleurs I2C pour la simulation.

3.8.1 Simulation des machines à états :

Machine de décodage d'adresses :

La simulation suivante est celle de la machine de décodage d'adresses (Figure III.10) dont le déroulement est décrit dans le paragraphe 3.5.1, le graphe de la simulation est présenté dans la fiche de simulation 1.

La machine à états finis, est dans l'état IDLE, le microcontrôleur 1, positionne la ligne R_W en écriture (R_W=0) et charge une adresse sur le bus d'adresses (addr_bus : 000F8D) c'est l'adresse du registre d'adresse MADR. Une fois le bus d'adresse est chargé correctement, le uC_1 envoie une assertion d'adresse AS=0 c'est alors que le contrôleur I2C_1 charge à son tour l'adresse 000F8D dans le registre MADR_Maître. En même temps, le uC_1 charge le bus de données avec la valeur 0F (adressage de l'esclave en lecture) et effectue une assertion de données et le contrôleur I2C_1 charge cette valeur dans le registre MADR (voir le signal MADR_maître). Du côté du uC_2, ce sont les mêmes opérations sauf pour l'adresse de l'esclave qui est 0E (adressage de l'esclave en écriture). À la fin du transfert des données et des adresses le contrôleur I2C s'acquitte vis-à-vis du uC en envoyant un niveau bas sur la ligne DTACK (voir le signal acquittement_données). Les lignes SDA et SCL sont à l'état haut car il n'y a pas pour le moment de transfert sur le bus I2C. Les mêmes opérations sont répétées pour tous les autres transferts entre le uC et le contrôleur I2C. Une fois les deux contrôleurs sont configurés, le maître envoie une condition de départ START (voir la Figure III.19) (voir la fiche de simulation 1).

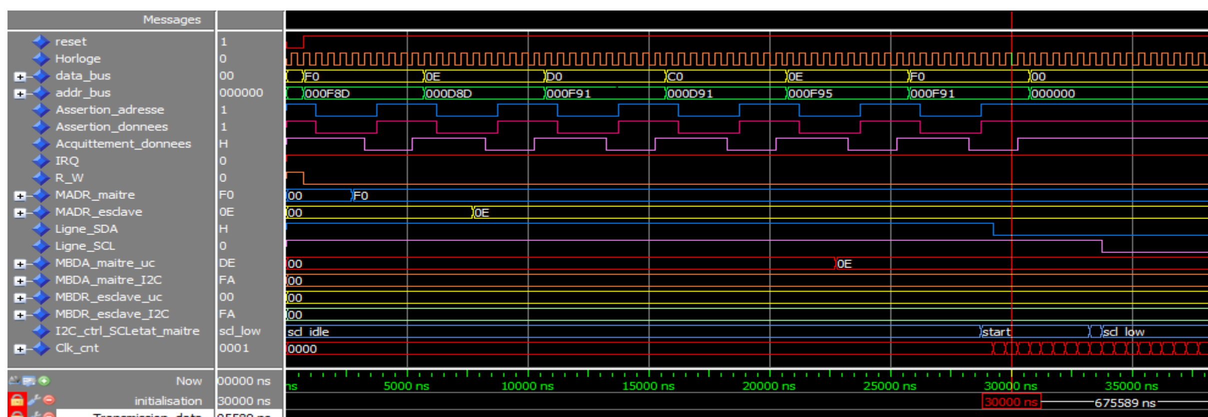


Figure III.19 : Simulation de la machine de décodage d'adresses.

Machine de génération des signaux SCL, SDA, START et STOP :

La simulation suivante est celle de la machine de génération des signaux SCL, SDA, START et STOP (figure III.12) dont le déroulement est décrit dans le paragraphe 3.6.2, le graphe de la simulation est présenté dans la fiche de simulation 2.

Quand il n'y a pas de transfert en cours (le bus I2C est libre) la machine de générations des signaux et la machine à états finis principale sont aux états d'attente IDLE, le compteur d'impulsions est à zéro (clk_cnt = 0000). Dès qu'il y a génération de la condition START, le compteur commence le comptage des impulsions d'horloge. Dès qu'un START valide est atteint, il y a génération d'un front descendant sur la ligne SCL (c'est l'état SCL_LOW_EDGE) puis établissement de l'état SCL_LOW et la ligne SCL est au niveau bas. Dès que le niveau bas valide est atteint (clk_cnt = 1001) il y a génération d'un front montant (c'est l'état SCL_HIGH_EDGE) puis établissement du niveau haut (c'est l'état SCL_HIGH) c'est ainsi qu'une période est créée et la répétition de cette opération crée le signal d'horloge. À base de ce comptage des impulsions d'horloge, la machine de génération des signaux peut générer les conditions START et STOP valides, un signal d'horloge de fréquence 100KHz sur la ligne SCL et maintenir les états haut et bas sur la ligne SDA suffisamment long temps pour qu'ils soient valides (voir la Figure III.20) et (voir la fiche de simulation 2).

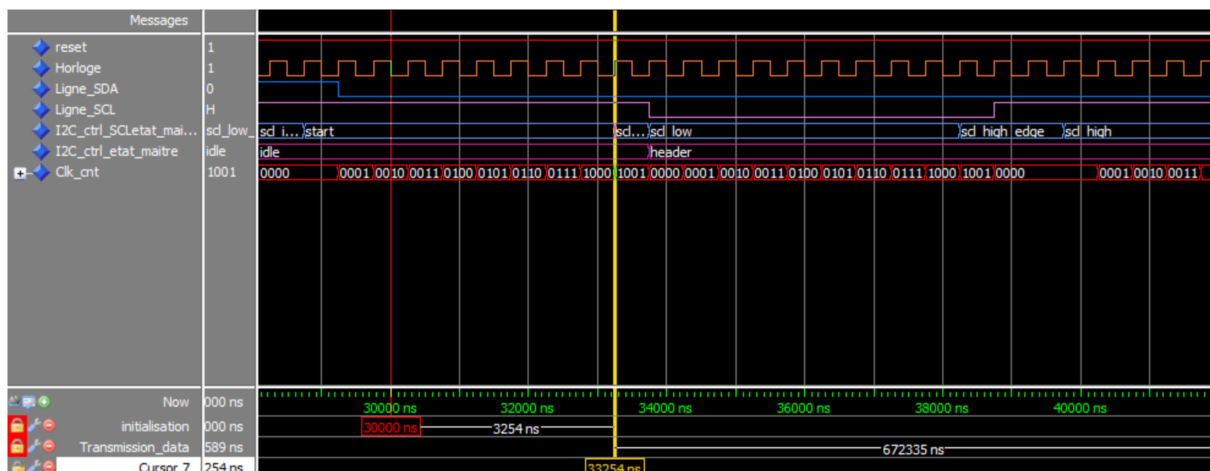


Figure III.20 : Simulation de la machine génération des signaux SCL, SDA, START et STOP.

Machine à états finis principale :

La simulation suivante est celle de la machine à états finis principale (figure III.15) dont le déroulement est décrit dans le paragraphe 3.6.3, les graphes de la simulation sont présentés dans les fiches de simulation 3a, 3b et 3c.

Le protocole I2C stipule que trois formats de transfert sont possibles [*PHI07*] :

1. Maître émetteur transmet des données pour un esclave récepteur et la direction du transfert ne change pas.
2. Le maître lit l'esclave juste après le premier octet. C'est-à-dire, juste après le premier acquittement le maître émetteur devient récepteur et l'esclave récepteur devient émetteur.

3. Un format combiné, durant un transfert de données, il y a un changement de direction du transfert.

Premier cas : maître émetteur /esclave récepteur.

Le maître envoie l'adresse esclave chargée au préalable par le uC_1 dans le registre MBDA (voir le signal mbda_maître_uC) la valeur 0Eh (0000 1110b) qui est transférée via la ligne SDA. Le huitième bit (l'adressage est sous 7-bits) est celui du sens du transfert (le protocole I2C, indique que le huitième bit du premier octet (celui de l'adressage) indique le sens du transfert lecture = 1, écriture = 0). Puis viens le neuvième bit qui est l'acquittement du transfert ACK (ACK : positionne la ligne SDA à « 0 »), envoyé par l'esclave. (Voir la Figure III.21).

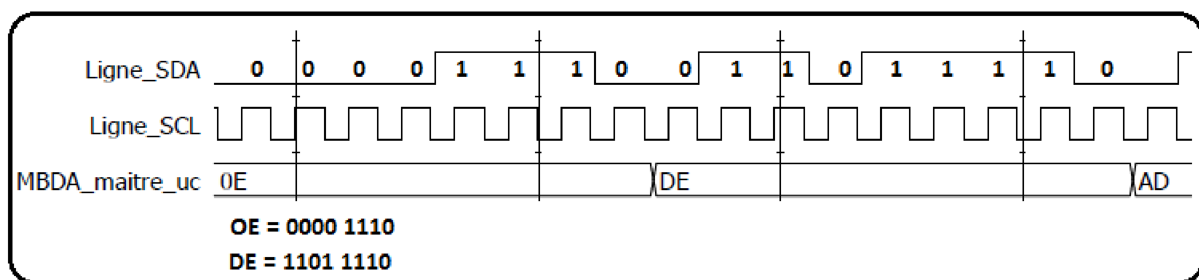


Figure III.21 : Illustration du transfert d'un octet sur la ligne SDA préalablement chargée dans le registre MBDA maître.

Le signal MCF esclave envoie une impulsion (de même pour MCF maître) pour signaler au uC la réception (émission) d'un octet complet, la machine à états finis principale (I2C_ctrl_etat_maitre) attend la réception d'un acquittement de la part de l'esclave pour enchaîner par l'envoie d'un nouvel octet. Le uC maître charge la donnée DEh (1101 1110b) dans le registre mbdr_maitre_uc, qui est transférée via la ligne SDA, après transfert elle se retrouve dans le registre mbda_esclave_I2C. Ainsi de suite pour les autres octets (AD, BE, EF, FA et DE). Une fois tous les octets sont transmis, le maître génère une condition STOP valide. (voir la figure III.22) et (voir la fiche de simulation 3a).

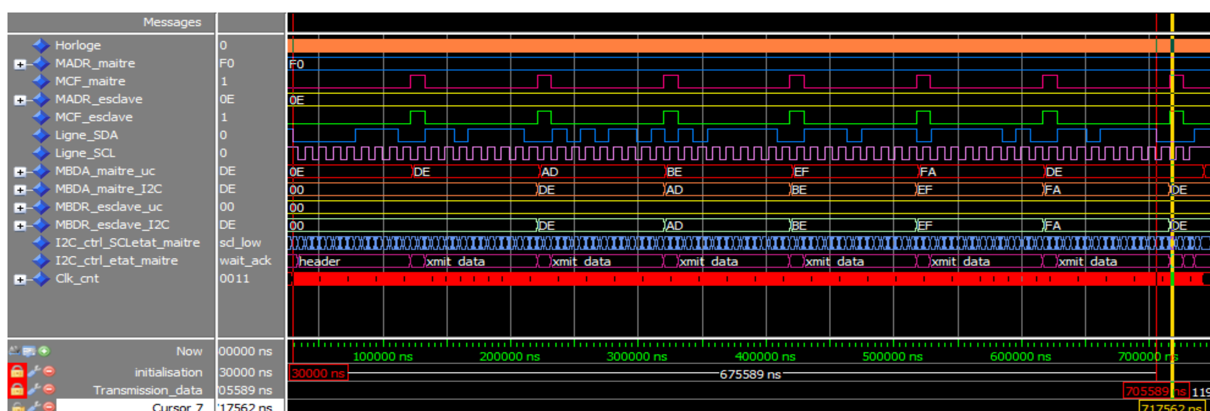


Figure III.22 : Simulation de la machine à états finis principale, mode transmission.

Deuxième cas : maître récepteur /esclave émetteur.

Le maître génère un START valide pour lancer un transfert. Le uC_1 charge la donnée 0Fh (0000 1111b) pour configurer le contrôleur I2C en écriture. Le maître adresse l'esclave en lecture 0Fh, les sept premiers bits forment l'adresse de l'esclave et le huitième indique le sens du transfert « 1 » pour la lecture, puis acquittement par l'esclave. Le uC_2 charge la donnée DEh (1101 1110) dans le registre MBDR_ESCLAVE_UC puis elle est transférée dans le registre MBDR_ESCLAVE_I2C puis la donnée est envoyée via le bus I2C par la ligne SDA et récupérée par le maître dans le registre MBDR_MAITRE_I2C. Toutes les autres données sont transférées selon cette séquence jusqu'à la dernière et le uC_1 ne s'acquitte pas de la dernière donnée et génère un stop valide.(voir la figure III.23) et (voir la fiche simulation 3b).

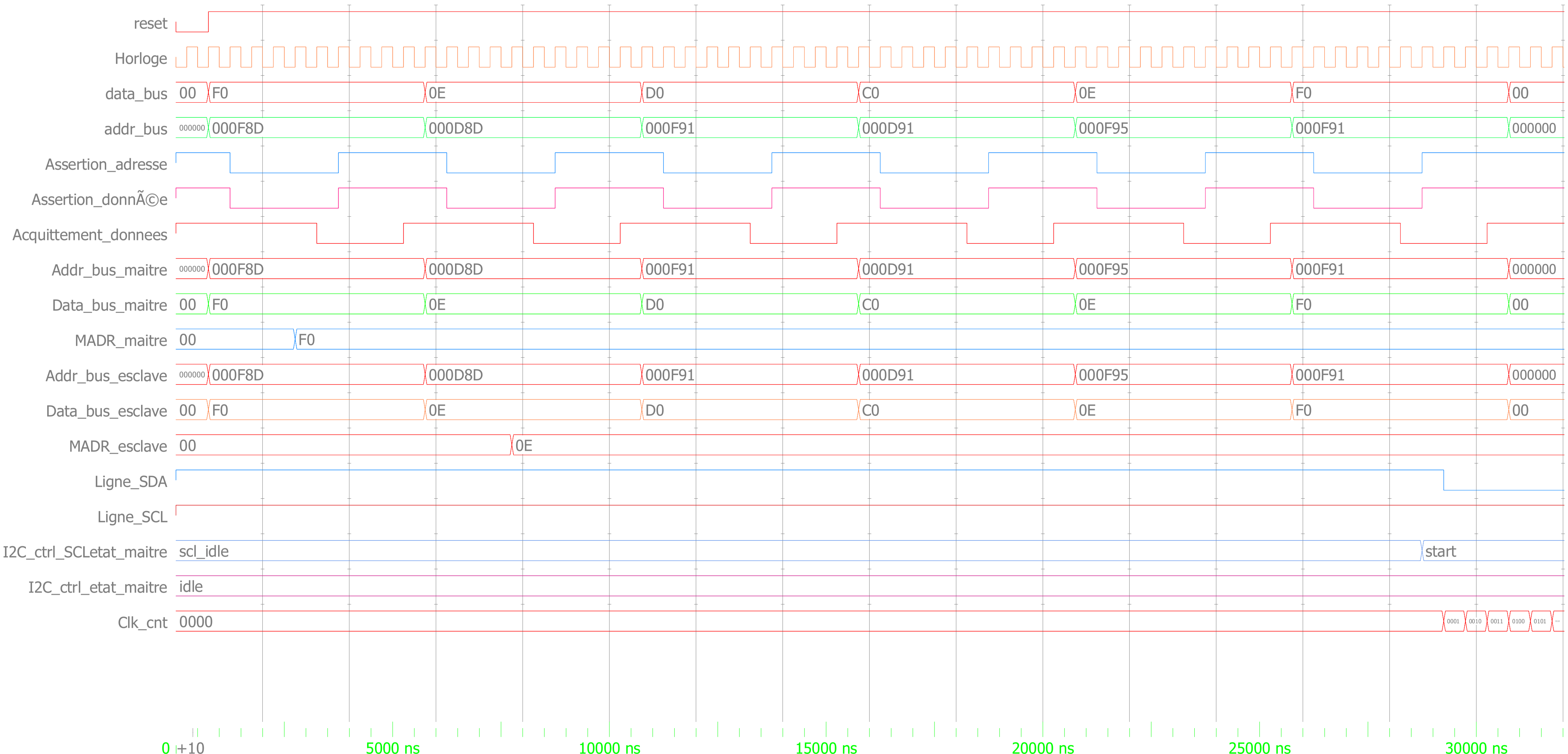
Figure III.23 : Simulation de la machine états principale en mode réception.

Troisième cas : le mode mixte.

Ce mode est une combinaison des deux modes précédents. Le maître initialise le transfert dans un sens (dans ce cas en émission) puis sans avoir à générer un STOP il réinitialise le transfert par un START valide pour changer le sens du transfert (dans ce cas en réception). (Voir la Figure III.24) et (la fiche de simulation 3c).

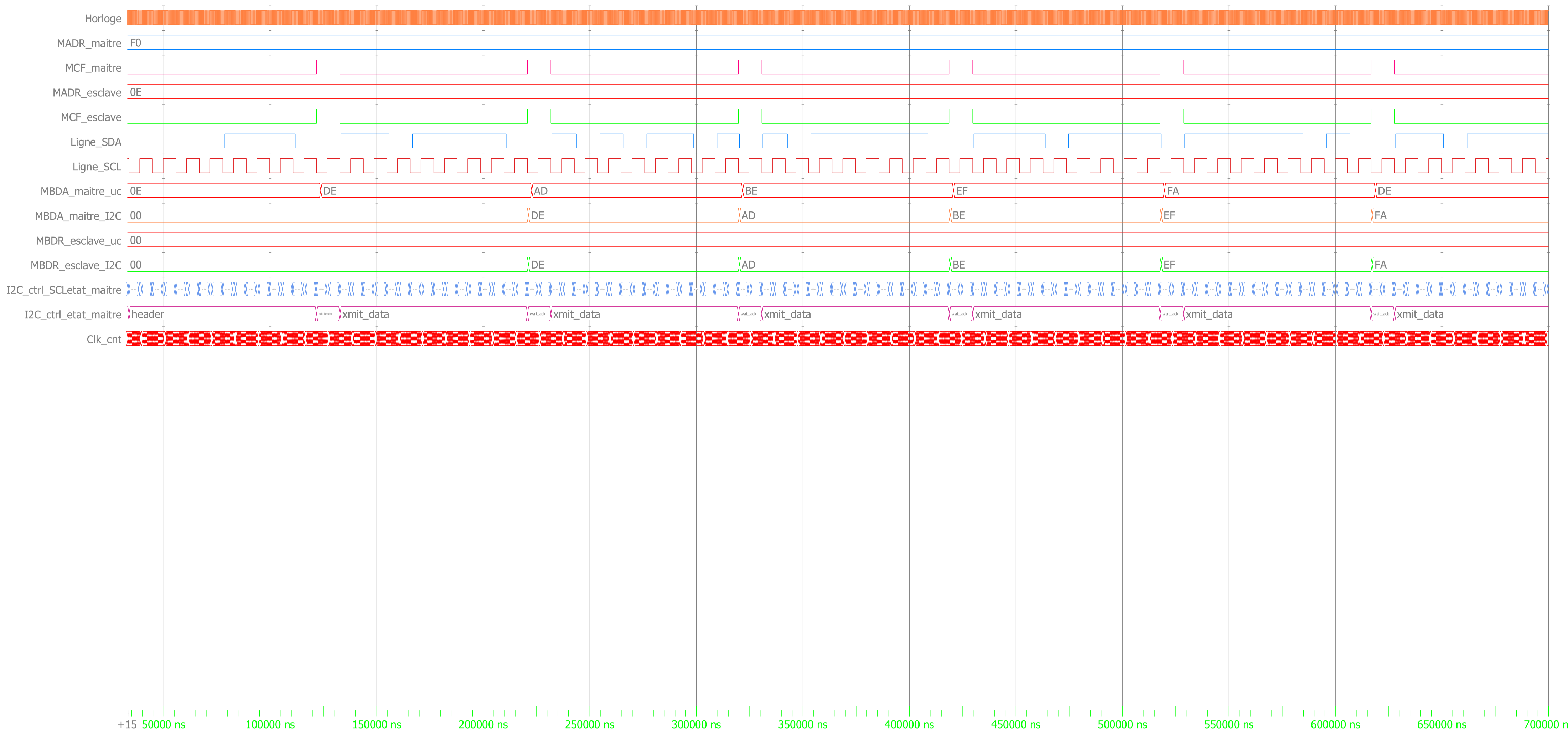
Figure III.24 : Simulation de la machine états principale en mode mixte.

La fiche de simulation 4, est une vue globale de la simulation de toutes les étapes.

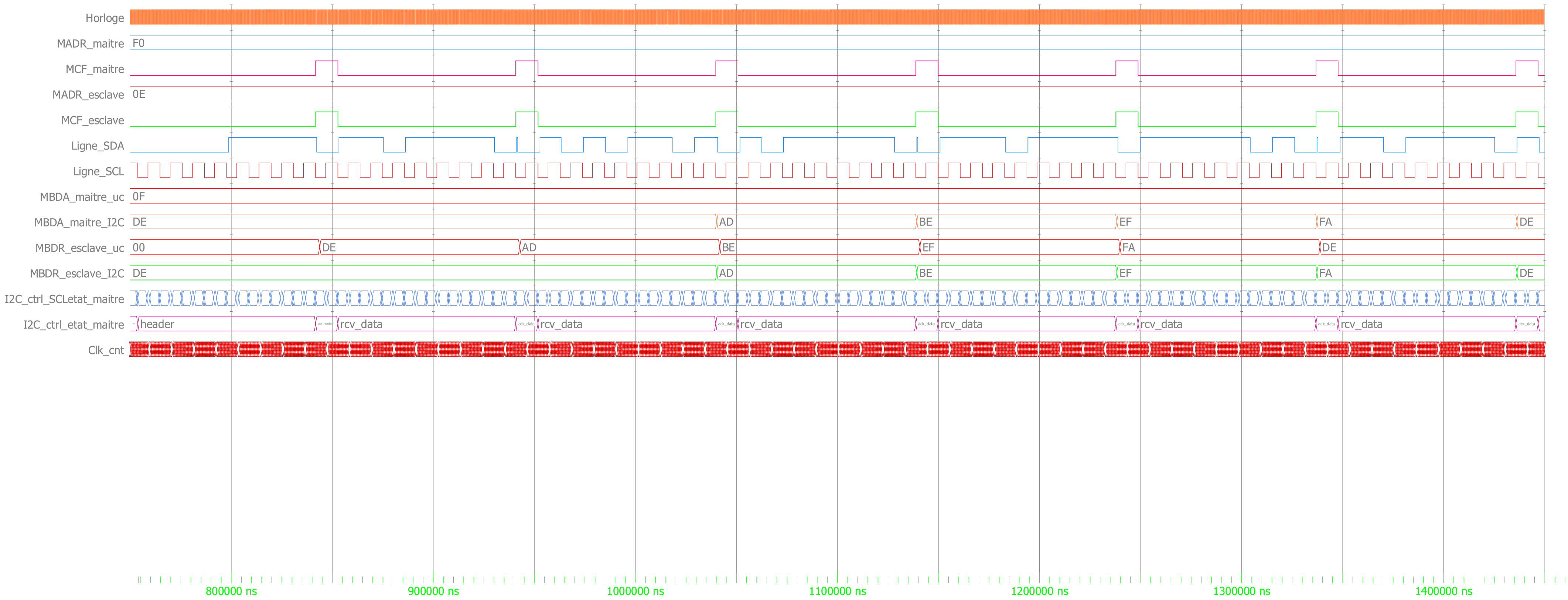


Simulation 1: Simulation de la machine de décodage d'adresses.

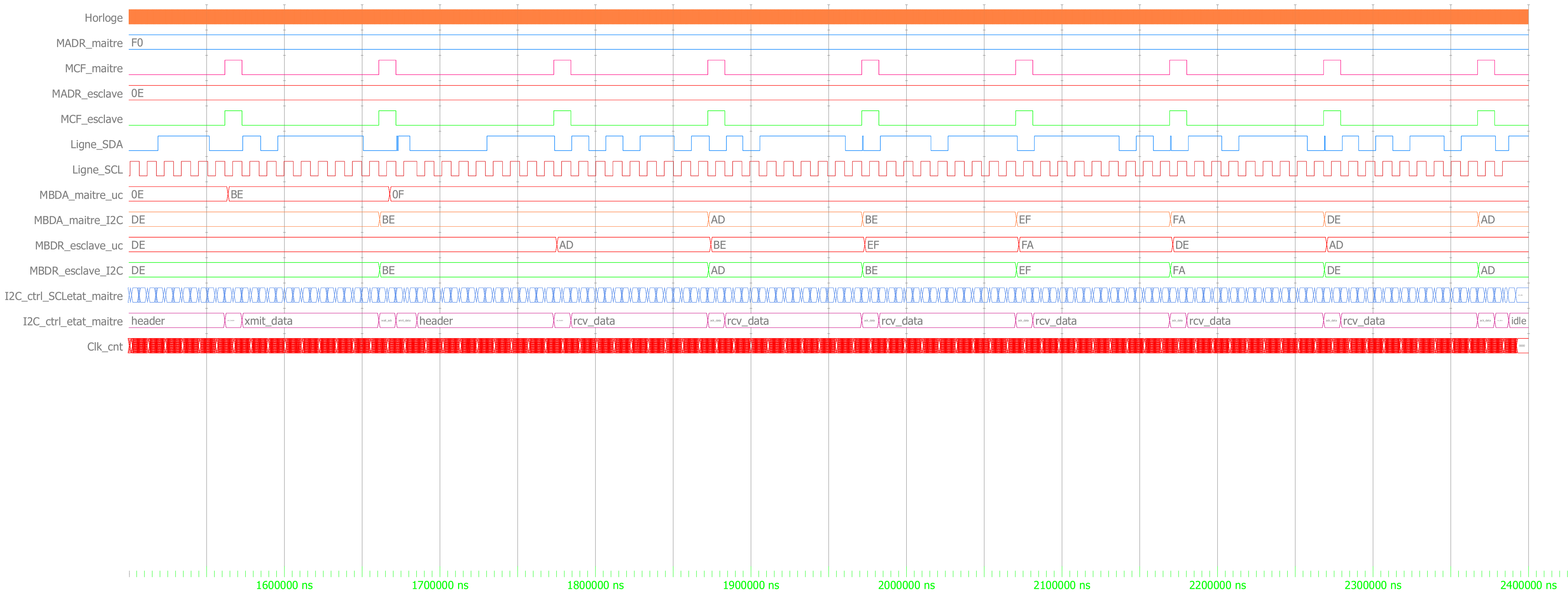
Simulation 2: Simulation de la machine génération des signaux SCL, SDA, START et STOP.



Simulation 3a: simulation de la machine à états principale, maître émetteur/ esclave récepteur.



Simulation 3b: Simulation de la machine à états principale, maître récepteur /esclave émetteur.



Simulation 3c : Simulation de la machine à états principale en mode mixte.

reset
Horloge
data_bus
addr_bus
Assertion_adresse
Assertion_donnees
Acquittement_donnees
IRQ
R_W
Addr_bus_maitre
Data_bus_maitre
MADR_maitre
MCF_maitre
Addr_bus_esclave
Data_bus_esclave
MADR_esclave
MCF_esclave
Ligne_SDA
Ligne_SCL
MBDA_maitre_uc
MBDA_maitre_I2C
MBDR_esclave_uc
MBDR_esclave_I2C
I2C_ctrl_SCLetat_maitre
I2C_ctrl_etat_maitre
Clk_cnt

0 ns 400000 ns 800000 ns 1200000 ns 1600000 ns 2000000 ns 2400000 ns

Simulation 4 : Vue globale de la simulation en entier.

Chapitre IV :

Implémentation du contrôleur I2C sur FPGA.

IV. Introduction :

Pour implémenter un circuit sur un FPGA, il y a trois éléments indispensables pour y parvenir :

1. Disposer de la description logique du circuit (schématique, diagrammes d'états ou code tel VHDL).
2. L'environnement de développement est choisi en fonction du composant sur lequel le circuit sera implémenté spécialement le logiciel du fabricant du composant (ISE : XILINX Integrated Software Environment).
3. Une carte de développement pour implémenter le circuit et vérifier son comportement.

Le code VHDL pour la description matérielle du contrôleur I2C est présenté au chapitre III. Dans ce chapitre, il sera présenté l'environnement de développement de XILINX et l'implémentation du code VHDL (effectuée par le logiciel ISE 9.2). Puis une présentation de la carte de développement VIRTEX4 pour une implémentation physique.

IV.1. Présentation de l'environnement de développement « XILINX Integrated Software Environment (ISE) » :

C'est le logiciel de programmation des produits Xilinx (CPLD, FPGA Spartan et Virtex…) téléchargeable gratuitement sur le site Internet de Xilinx (dans sa version Web Pack). Il intègre différents outils permettant de passer à travers le flot de conception d'un système numérique. Il dispose de :

- Un éditeur de textes, de schémas et diagrammes d'états.
- D'un compilateur VHDL et Verilog.
- D'un simulateur.
- D'outils pour la gestion des contraintes temporelles.
- D'outils pour la synthèse.
- D'outils pour la vérification.
- D'outils pour l'implémentation sur FPGA et CPLD.

C'est un outil de développement complet pour toutes les gammes de produits XILINX [*XIL11*].

1.1 Les étapes pour l'implémentation d'une spécification HDL sur un FPGA :

Pour implémenter une spécification HDL sur un FPGA, sur la plate forme ISE 9[1] quatre étapes importantes sont à suivre tel illustré par la Figure IV.1 .

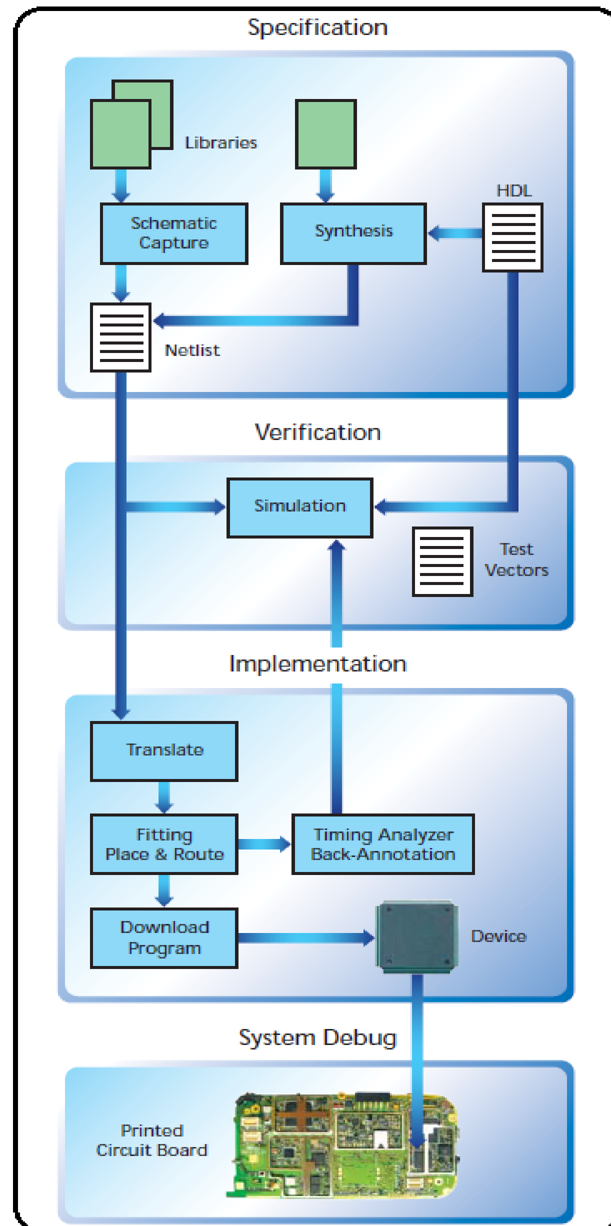


Figure IV.1 : les étapes d'implémentation d'un circuit sur un circuit logique programmable (XILINX) [UG500].

[1] Ces étapes sont des étapes communes à tous les fabricants de circuits logiques programmables. Chaque fabricant établit un schéma en fonction de ses produits proposés et des logiciels offert pour pouvoir implémenter une spécification HDL.

1.1.1 Spécification :

Le terme spécification est un terme qui regroupe les trois modes (schématique, diagrammes d'états ou HDL) de saisie d'un circuit électronique. La spécification HDL est synthétisée pour générer un fichier appelé NETLIST qui décrie les interconnexions entre les registres.

1.1.2 Vérification :

La vérification du design est une étape parallèle où le concepteur observe le comportement du code et observe s'il se comporte tel qu'il est supposé. Un simulateur simule le circuit à condition de lui fournir les vecteurs de test. Les vecteurs de teste peuvent se présenter sous plusieurs formes, la plus courante est les TESTBENCHS rédigés dans un langage de description matériel ou sous forme de lignes de code TCL pour entrer les instructions au simulateur. En appliquant les vecteurs de test sur le code, le simulateur fournit les sorties du circuit.

1.1.3 Implémentation :

Une fois le Netlist (les interconnexions entre les portes logiques) décrit la conception en utilisant les portes logiques en tenant compte du fabricant et d'une famille de composants bien définie. Une fois la vérification est terminée, le circuit est implémenté sur le composant en spécifiant les références exactes de celui-ci à savoir : le boîtier, la fréquence de travail et les autres options spécifiques à chaque composant. Cette étape se termine par un rapport de tous les sous-programmes exécutés, les Warnning, les erreurs, les I/O utilisés, des données qui permettent de savoir si le composant choisit est le mieux adapté pour l'application ciblée.

Cette étape se décompose à son tour en sous-étapes que voici :

Fitting : C'est le dimensionnement de la conception en fonction des ressources internes du composant cible.

Place and Route : Les sous-programmes de placement et routage sont exécutés après compilation du code.

Place : c'est le processus de sélection des modules ou blocks logiques où les portes logiques seront placées.

Route : le routage est l'interconnexion physique entre les différents blocks logiques.

Downloading ou Programming : après que le fichier de programmation est généré vient l'étape où il faut le charger sur le FPGA ciblé par l'application. Les FPGA de type SRAM nécessite une mémoire de type PROM pour retenir le programme même

après extinction de la carte électronique comportant le FPGA, au démarrage il y a chargement ou downloading des interconnexions sur le FPGA. Dans le cas des FPGA à antifusibles, le programme peut être chargé ou programming par l'intermédiaire d'un programmateur ou par technique de programmation sur circuit JTAG (pour Joint Test Action Group est le nom de la norme IEEE 1149.1 intitulé « Standard Test Access Port and Boundary-Scan Architecture »).

1.1.4 Débuggage du système :

Après chargement des interconnexions sur le FPGA, des testes peuvent être effectuer sur le circuit implémenté a fin de voir les réactions du circuit et son comportement et détecter les anomalies à fin de les corriger. En cas d'anomalie la spécification est revue et corrigée en conséquence. Si les tests physiques s'avèrent concluants, la spécification est validée et le prototype est considéré opérationnel.

1.2 Interface graphique de l'environnement ISE 9.2 :

L'ISE contrôle tous les aspects d'un design flow. L'interface Project Navigator donne accès à toutes les ressources d'un projet et aux outils de l'implémentation. Elle procure aussi un accès aux fichiers et documents associés au projet (voir la Figure IV.2).

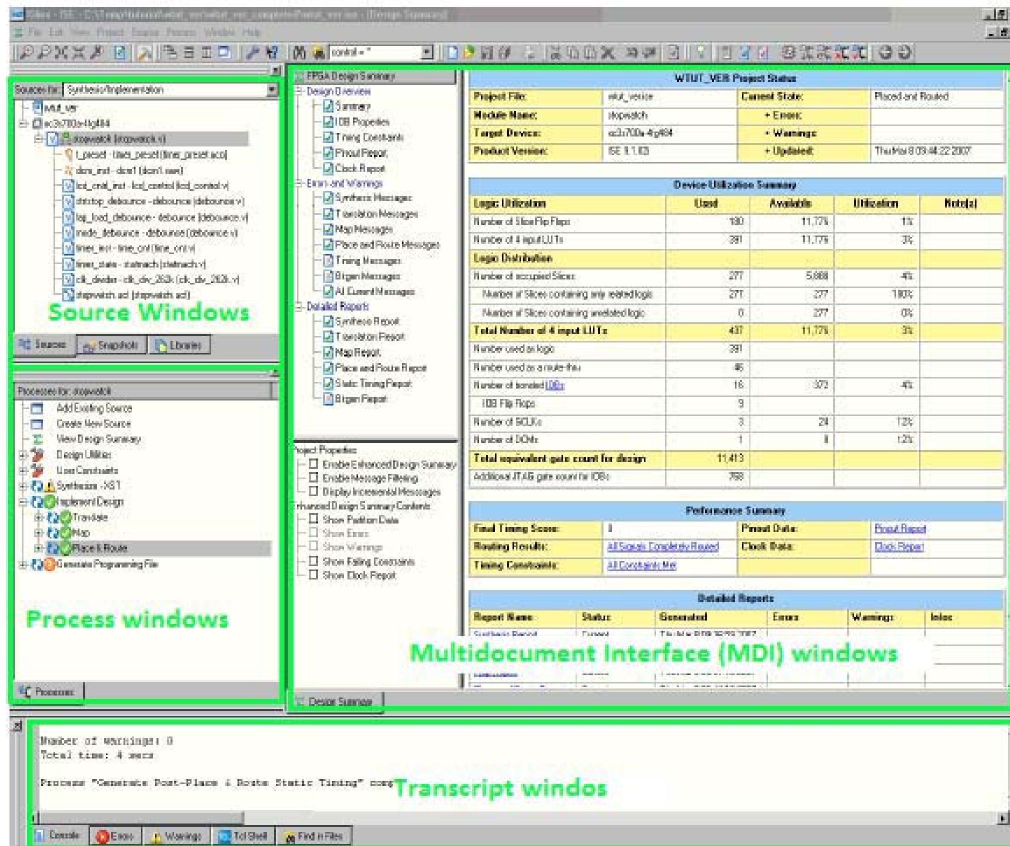


Figure IV.2 : Project Navigator du logiciel ISE 9.2.

L'interface du Project Navigator se divise en quatre sous-fenêtres principales :

Source windows : Elle affiche les éléments inclus dans le projet de façon hiérarchique.

Process windows : Affiche les processes disponibles pour la source sélectionnée.

Transcript windows : Affiche les messages d'état, les erreurs et warning. Elle affiche aussi sous un tab les scripts TCL.

Multidocuments interface MDI windows : c'est l'espace de travail, elle permet de visualiser les rapports HTML, les fichiers textes ASCII, les schématiques et la fenêtre de simulation.

IV.2 La plateforme de développement VIRTEX4 :

La plate-forme de développement ML401 est une plate forme de conception et de mise en œuvre des circuits numériques implémentés sur VIRTEX4 (xc4vlx25). Après chargement du fichier de programmation, il permet de tester le bon fonctionnement du circuit ou la détection des bugs ou dysfonctionnements. Elle permet de recevoir directement le fichier bitstream sans avoir à réaliser un circuit imprimé en particulier. Elle dispose d'entrées /sorties, de port de communication, d'un port VGA, deux ports PS2 (Pour connecter un clavier ou une souris), d'un port RS232, d'un port de sortie audio, d'un afficheur LCD deux lignes à 16 caractères, des DIP switches à usage général, des LEDs et des boutons poussoirs. La communication avec cette carte peut être établie via un câble USB. (Voir la Figure IV.3)

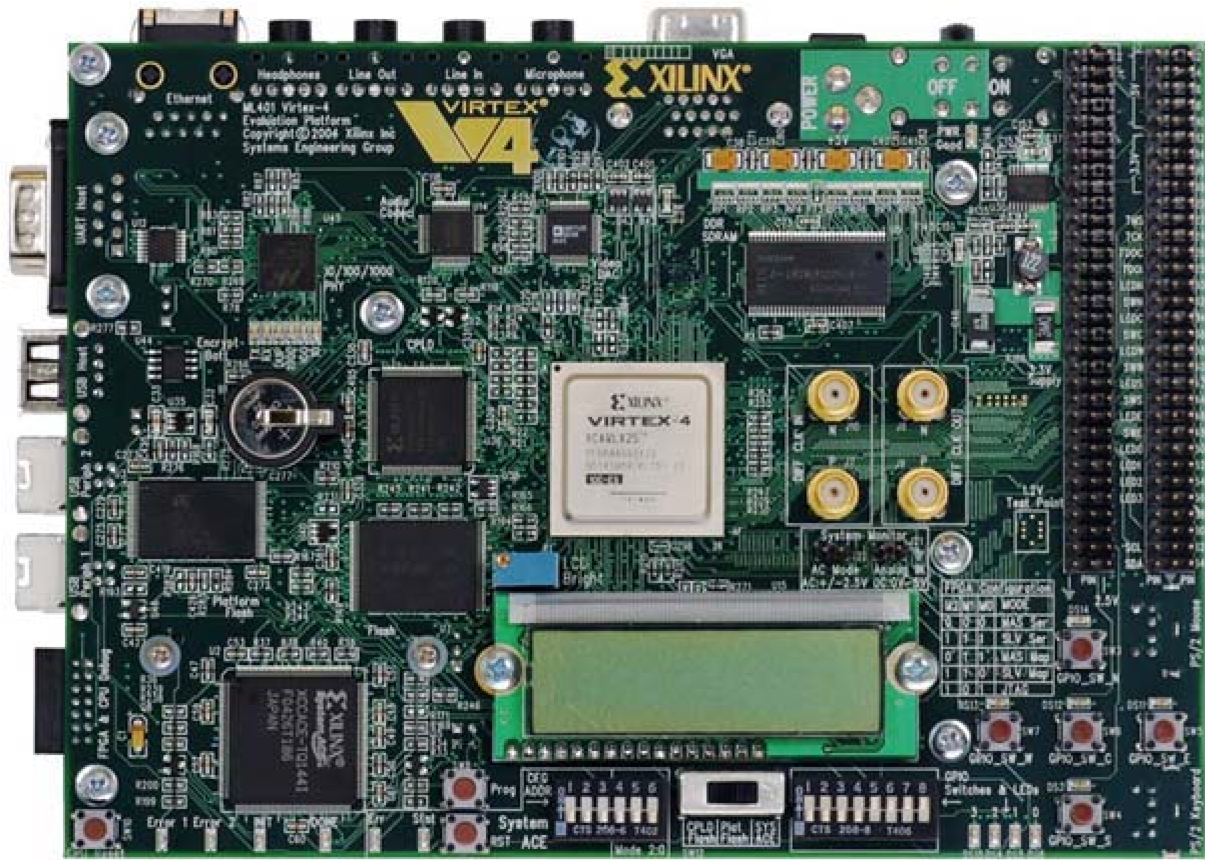


Figure IV. 3 : Vue externe de plateforme ML401.

IV.3 Synthèse et Implémentation du contrôleur I2C sur la VIRTEX4 :

Pour implémenter le contrôleur I2C sur la VIRTEX4, il faut générer un fichier BITSTREAM au format JDEC qui servira à la configuration de la VIRTEX4. Pour parvenir à générer ce fichier « project.jdc » il faut passer par deux étapes majeures : Synthèse et implémentation.

3.1 Synthèse de la description matérielle du contrôleur I2C :

L’étape de synthèse reçoit en entrée le code VHDL après vérification et simulation et le fichier des contraintes (voir la Figure IV. 4).

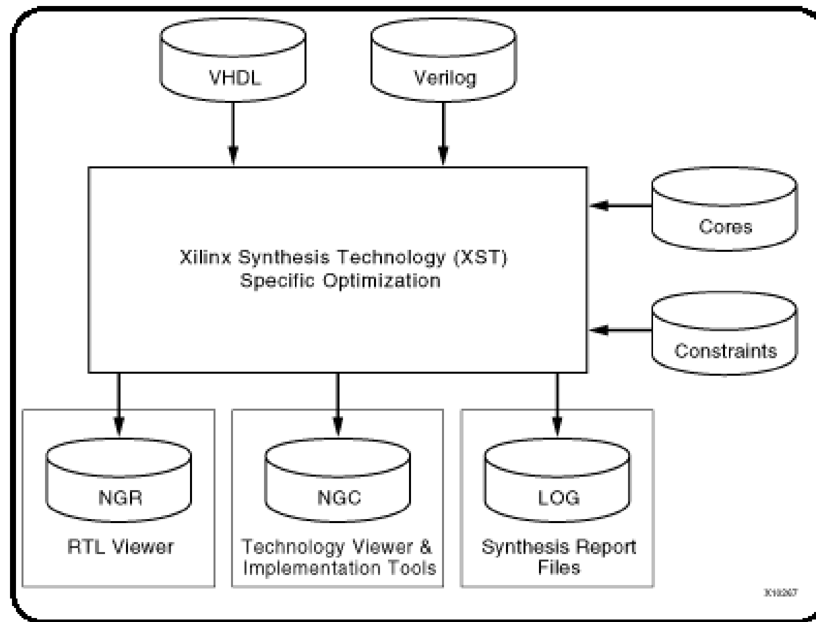


Figure IV.4 : Design Flow de l'étape de synthèse [XIL11]

L'étape de synthèse génère quatre fichiers :

3.1.1 XILINX Specific file au format .NGC : c'est le fichier principal de la synthèse et il comporte les données logiques qui constituent la conception et les contraintes. Le fichier I2C.NGC est utilisé pour l'implémentation dans l'étape Translate.

3.1.2 Register Transfer Level (RTL) Schematic : c'est un schéma représentatif de la conception préoptimisée au niveau RTL. C'est une représentation en symboles génériques tels des additionneurs, multiplexeurs, compteurs …etc. Il est généré à la fin de l'étape de synthèse, il permet d'avoir un aperçu du circuit tout au début du processus d'implémentation.

La figure IV.5 Illustre le fichier RTL Schematic du contrôleur I2C implémenté sur la VIRTEX4.

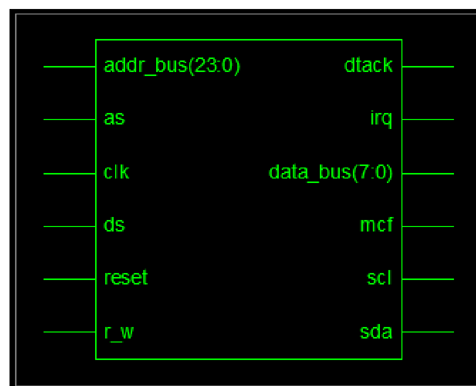


Figure IV.5 : vue d'ensemble du fichier RTL Schematic.

Tous les signaux d'entrées et sorties du contrôleur I2C sont représentés.

Le schéma 1.a est le schéma (RTL Schematic) interne du contrôleur I2C, il représente l'interface I2C et l'interface uC et les signaux d'interconnexion.

Le schéma 1.b est le schéma (RTL Schematic) interne de l'interface I2C.

Le schéma 1.c est le schéma (RTL Schematic) interne de l'interface uC.

3.1.3 Technology Schematic: C'est une représentation schématique du fichier NGC, elle figure en termes d'éléments logiques optimisés pour une architecture ou une technologie bien définie. Par exemple, schématisé avec des LUTs, buffers d'I/O …etc. Elle est générée après l'étape d'optimisation et la spécification de la technologie utilisée par le processus de synthèse. Ce fichier permet de visualiser la conception au niveau technologique du code VHDL. À ce niveau figure le schéma tel qu'il sera implémenté sur le composant FPGA.

Le schéma 2 est une partie du Technology Schematic du contrôleur I2C.

3.1.4 Le fichier LOG : C'est un rapport qui contient les résultats de la synthèse. Il contient toutes les informations relatives au fichier d'entrée, le nom du fichier et la prise en charge des contraintes. Des informations relatives au fichier de sortie, le nom du fichier et son format NGC. Les messages d'erreurs et warning. Ainsi que les ressources internes nécessaires pour l'implémentation.

3.1.5 Le rapport de synthèse :

L'environnement ISE, fournit un rapport de synthèse sous forme de tableaux contenant les informations utiles liées au design que voici :

Le tableau suivant illustre l'état du projet, il ne donne pas d'erreurs, 9 Warnings sans influence sur le bon fonctionnement du contrôleur I2C.

I2C Project Status			
Project File:	i2c.ise	**Current State:**	Synthesized
Module Name:	i2c	• **Errors:**	No Errors
Target Device:	xc4vlx25-10sf363	• **Warnings:**	9 Warnings
Product Version:	ISE 9.2i	• **Updated:**	mer. 10. sept. 10:56:21 2011

Le tableau suivant résume le sommaire des ressources internes qui seront utilisées pour l'implémentation.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	102	10752	0%
Number of Slice Flip Flops	113	21504	0%
Number of 4 input LUTs	198	21504	0%
Number of bonded IOBs	42	240	17%
Number of GCLKs	2	32	6%

3.2 Implémentation du contrôleur I2C sur la VIRTEX4 :

L'étape de l'implémentation vient juste après la synthèse. Le fichier NGC est utilisé lors de cette étape en entrée pour générer le fichier de programmation appelé BITSTREAM et il est au format JDEC.

L'étape de synthèse se divise en trois étapes :

3.2.1 Translate : Cette étape permet d'associer le fichier Netlists[2] en l'occurrence le fichier NGC avec le fichier des contraintes (association des pins d'entrées /sorties, les contraintes temporelles et autres) pour générer un fichier de format .NGD (XILINX Native Database).

3.2.2 Fit : Cette étape permet de dimensionner le fichier I2C.NGD en fonction des ressources disponibles sur la VIRTEX4. Après vérification il y a génération d'un fichier de format .VM6 qui représente la conception dimensionnée pour qu'elle puisse être implémentée sur la VIRTEX4.

L'environnement ISE, fournit un rapport d'implémentation sous forme de tableaux contenant les informations utiles liées au design que voici :

Le tableau suivant comptabilise toutes les ressources internes utilisées en nombre et en pourcentage.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	100	21,504	1%	
Number of 4 input LUTs	189	21,504	1%	
Logic Distribution				
Number of occupied Slices	129	10,752	1%	
Number of Slices containing only related logic	129	129	100%	

[2] Le fichier Netlist peut être d'un format autre que le NGC généré par l'étape de synthèse. Il peut être de format EDIF un standard industriel international des Netlist ou SEDIF, EDN, EDF, UCF, NCF, URF, NMC.

Number of Slices containing unrelated logic	0	129	0%	
Total Number of 4 input LUTs	189	21,504	1%	
Number of bonded IOBs	42	240	17%	
Number of BUFG/BUFGCTRLs	2	32	6%	
Number used as BUFGs	2			
Number used as BUFGCTRLs	0			
Total equivalent gate count for design	2,104			
Additional JTAG gate count for IOBs	2,016			

Le tableau suivant donne des renseignements sur les performances du routeur placeur.

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

La Figure IV.6 est un aperçue d'une partie du circuit implémenté avec les routages et l'emplacement des ressources utilisées. Il est à noter que les ressources non routées ne sont pas utilisées par le circuit (en haut à droite de la figure par exemple).

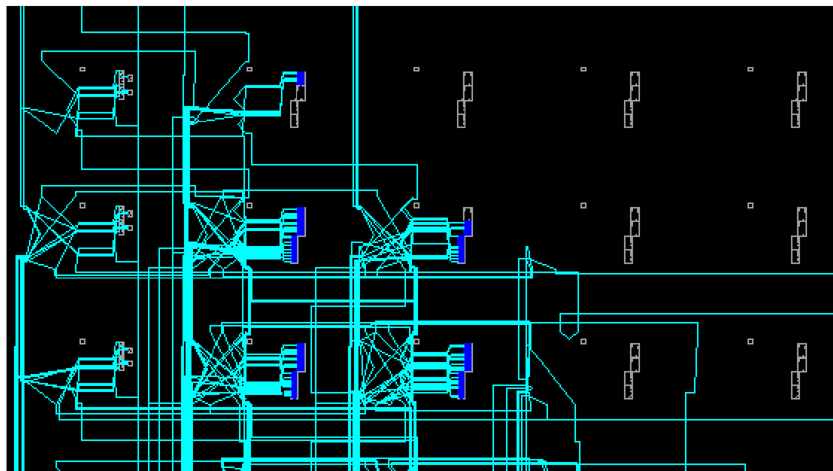


Figure IV.6 : aperçue d'une partie du circuit implémenté sur la VIRTEX4.

3.2.3 Programming file generation : Cette étape génère le fichier BITSTREAM au format JDEC. C'est ainsi que le fichier de chargement des interconnexions pour que le contrôleur I2C puisse être implémenté sur la VIRTEX4 est généré.

Schéma 1a: vue interne du contrôleur I2C des d'interface I2C et de l'interface uC.

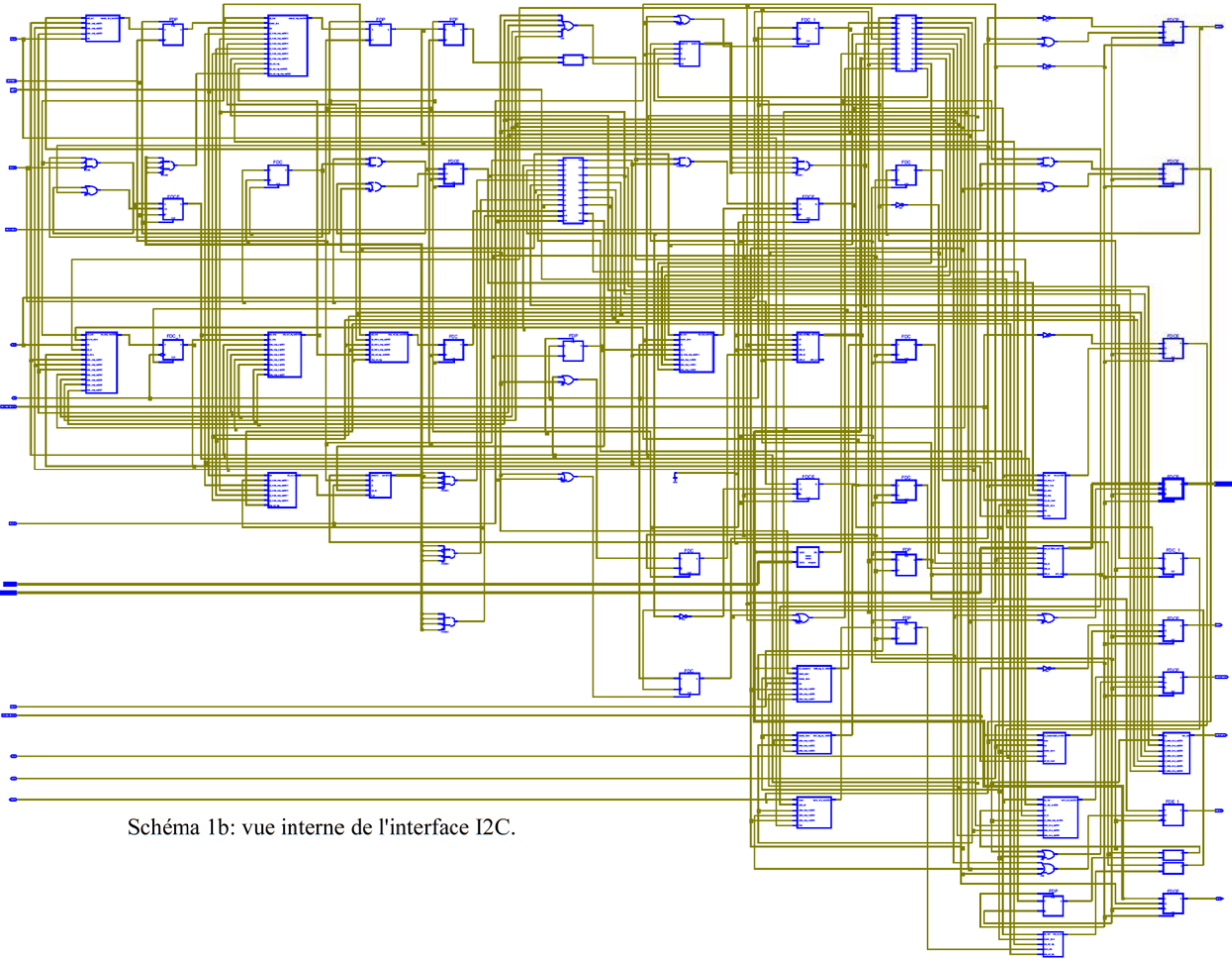
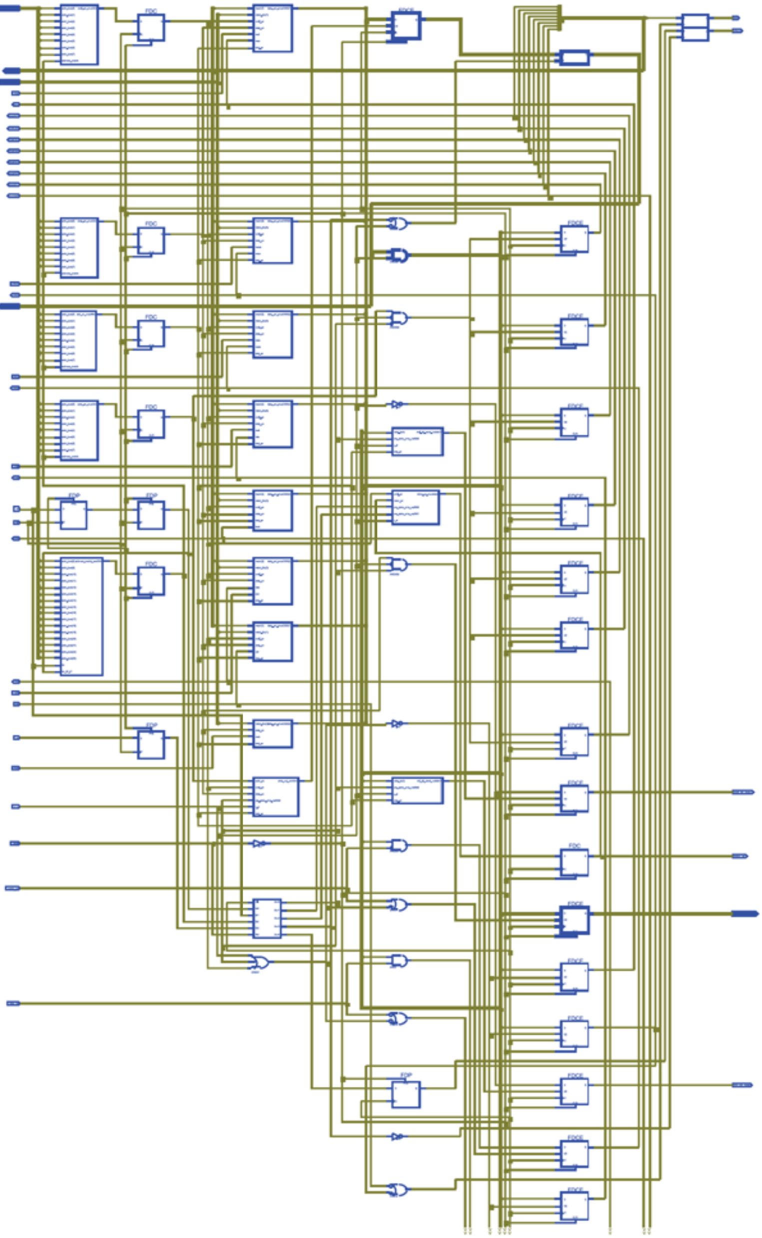


Schéma 1b: vue interne de l'interface I2C.



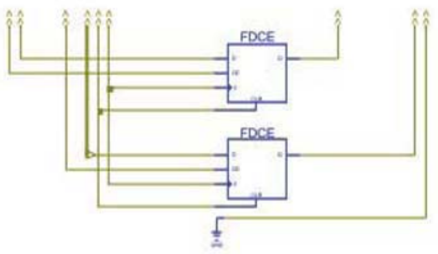


Schéma 1c: vue interne de l'interface uC.

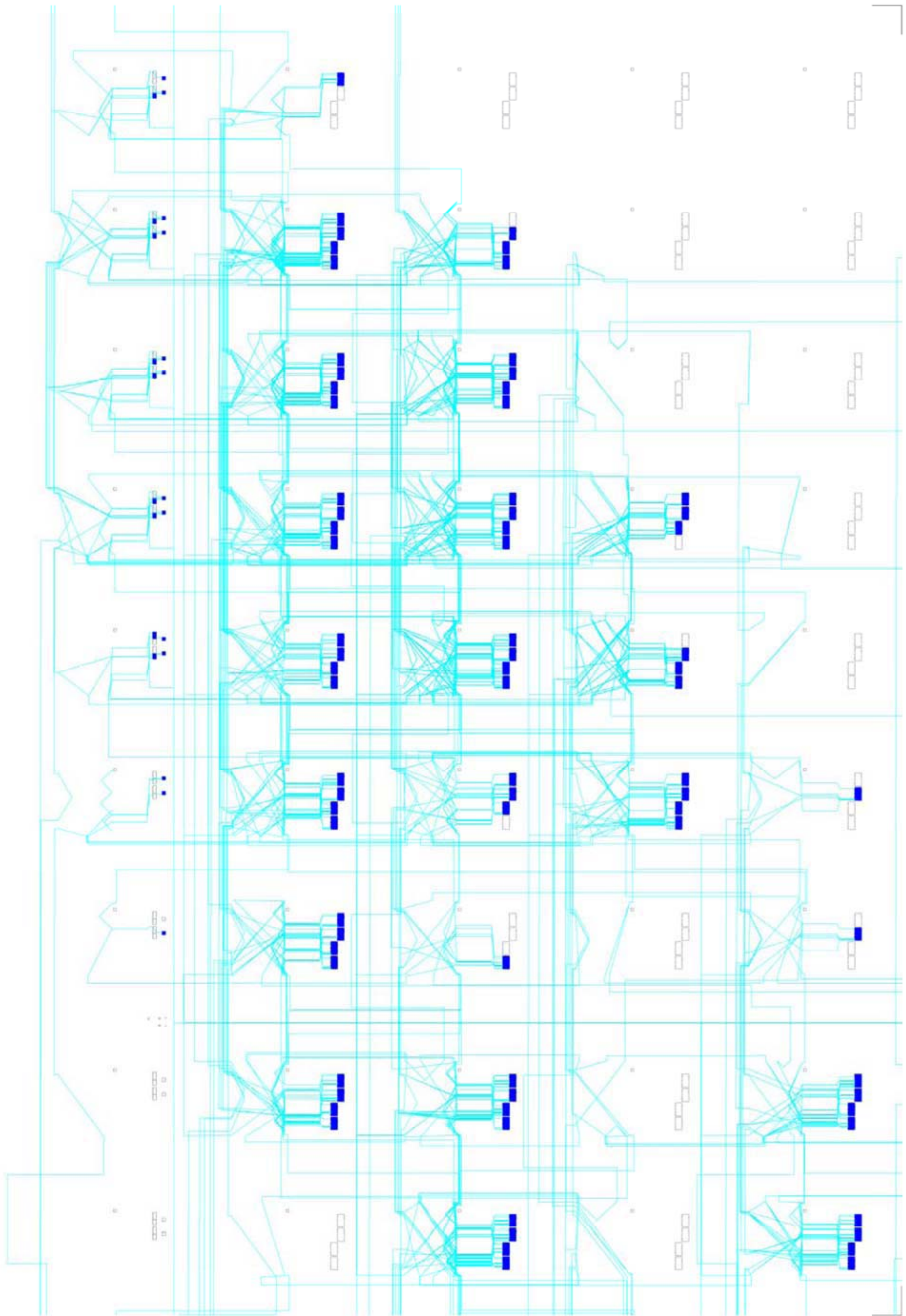


Schéma 2: Aperçu partiel du Technology Schematic du contrôleur I2C.

Les travaux menés au cours de ce mémoire constituent l'ensemble des étapes indispensables pour l'implémentation d'un circuit logique sur un circuit FPGA de façon générale et l'implémentation d'un contrôleur I2C en particulier. Le bon fonctionnement du contrôleur I2C est confirmé par la simulation numérique du programme implémenté et vérifié en détail sur la base des machines à états et selon la spécification du protocole I2C. En effet, le protocole I2C décrit le comportement du contrôleur à chaque étape de chaque mode de transmission, l'état initial, l'état final, l'arbitrage, l'attente …etc. La conception précise et sans omissions du protocole I2C a prouvé l'efficacité de ce dernier en occurrence par son succès et sa large diffusion. Des chronogrammes de sorties présentés sous des fiches de simulation sont des éléments qui permettent de vérifier que la description HDL est fidèle à l'énoncée du protocole I2C. L'implémentation physique est représentée par la vue interne établie par logiciel et qui donne un aperçu sur la représentation des portes logiques selon leurs implantations et routage dans le circuit FPGA.

Pour parvenir à réussir l'implémentation du circuit, on a suivi la méthode scientifique en commençant par la recherche bibliographique, l'étude des documents et l'assimilation des informations et par-dessus tout l'assimilation de la méthodologie propre à l'implémentation des circuits numériques sur un FPGA. Puis viens la partie expérimentale qui est la réalisation du circuit en langage VHDL, on teste le circuit en le simulant ce qui est tout à fait acceptable puis viens la dernière étape qui est la réalisation qui se matérialise par l'implémentation. C'est pour cela que nous avons procédé comme suit :

En premier lieu nous nous sommes attelés à étudier les circuits logiques programmables, le FPGA s'est révélés le circuit logique idéal pour l'implémentation d'un circuit d'une grande densité. Un choix fondé sur les principaux atouts des FPGA à savoir la flexibilité, la fiabilité, la performance, le temps d'élaboration d'un prototype opérationnel et la maintenance à long terme.

Puis nous avons étudié en général les langages HDL. Les langages comportementaux sont le fruit du perfectionnement des descriptions matérielles. Parmi ces langages ils existent deux qui ont acquis une notoriété : VHDL et VERILOG. Après une étude comparative des deux langages, nous avons retenu celui qui sert le plus nos objectifs c'est-à-dire le VHDL. Un langage puissant qui permet de suivre la conception d'un circuit logique à chaque étape de son

élaboration c'est-à-dire la conception, la simulation et la synthèse du fichier de programmation le BITSTREAM.

Pour pouvoir utiliser les FPGAs en instrumentation, on a fait l'étude du capteur et son instrumentation associée et sa place dans une chaine de mesure, puis nous avons retenu le capteur intelligent. Ce dernier, mis en contacte avec le phénomène physique à mesurer débite en sortie via un port de communication la valeur de la grandeur physique mesurée. L'importance du port de communication dans un capteur est mise en évidence par le fait que c'est le seul lien qui relie le capteur avec le calculateur central.

L'intérêt du port de communication dans la gestion du capteur par un calculateur central parait évident. Le port de communication I2C est l'un des ports embarqué sur les capteurs ce qui a orienté le travail vers l'étude du protocole I2C. Il se trouve qu'il ne suffit pas d'embarquer un port de communication sur un capteur pour tirer le maximum d'efficacité de ce dernier en particulier lors ce qu'il s'agit d'un réseau de capteurs. En effet, un port de communication a ses limites. Le port I2C en dépit de ses avantages très intéressants, se limite à un certain nombre d'adresses (maximum 128).

Pour remédier à ce problème de limite du bus, utiliser un FPGA pour gérer un réseau de capteurs simplifie grandement la gestion du réseau de capteurs et améliore l'efficacité de la communication et le transfert de données avec le calculateur central. Le contrôleur I2C embarqué sur le FPGA occupe seulement une adresse sur le bus, il fournit seulement les informations utiles, permet de mieux gérer la communication et la configuration des capteurs. Le calculateur a un accès aux informations utiles en un minimum de temps et constamment mis à jour.

Après avoir cerné le sujet et détecté la problématique, on s'est fixé l'objectif d'implémenter un contrôleur I2C sur un FPGA.

C'est alors que commence la deuxième étape, c'est-à-dire la conception du contrôleur I2C. Comme le circuit à implémenter est un circuit séquentiel la revue des machines à états finis été nécessaire pour pouvoir maître en œuvre la machine à états qui définit le comportement du contrôleur I2C. Après achèvement de la description en langage VHDL, viens l'étape de la simulation du code (ci-joint en annexe). La simulation a été faite selon les cas de figure et le contrôleur I2C est simulé dans ses deux modes simultanément, le mode

maître et le mode esclave. La lecture des chronogrammes est faite en fonction des cas de figure émission, réception ou mixte.

En dernier lieu vient l'implémentation du contrôleur I2C après qu'il soit validé par la simulation. Les étapes d'implémentation ont été décrites selon leur ordre d'apparition dans le processus d'implémentation avec la présentation de l'outil logiciel (XILINX ISE 9) et les formats des fichiers de sorties. Au final, nous avons présenté le circuit interne implémenté sur le FPGA. Ce circuit interne est obtenu grâce au logiciel XILINX ISE9 qui ne fait que rendre le fichier BITSTREAM visible.

En perspective, ce travail peut être poussé encore plus loin dans l'amélioration et l'optimisation pour la gestion d'un réseau de capteurs ceci en intégrant un DSP (Digital Signal Processor) un microprocesseur dédié pour le traitement du signal. Ce dernier permettrait de traiter les données et selon le programme embarqué il ne retiendra que les informations utiles à transmettre au calculateur central.

Bibliographie

[XIL11] (2011). Récupéré sur xilinx.com/: http://www.xilinx.com/

[ALT11] Altera. (2011). Récupéré sur Altera.com: www.altera.com

[ASC06] Asch, G. e. (2006). *Les capteurs en instrumentation industrielle, 6eme Edition.* . PARIS: DUNOD.

[STE05] BROWN, S. (2005). *Digital logic.* Toronto: Mc Graw Hill.

[COM11] *comelec.enst.fr/hdl/hdl_histhorique.html*. (2011). Consulté le juin 2011, sur comelec.enst.fr: http://comelec.enst.fr/hdl.html

[LAU97] DUTRIEUX, L. (1997). *Logique Programmable.* PARIS: EYROLLES.

[ROB05] Ferrabone, R. (2005, edition 176 MARS). Control an FPGA with I2C. *CIRCUITCELLAR* , pp. 60-65.

[MEN09] Graphics, M. (2009). *www.mentor.com*. Consulté le 2011, sur MENTOR GRAPHICS: www.mentor.com

[HER03] HERVEILLE, R. (2003, JUILLET 3). Lattice Micro32 (I2C Master). *REV 0.9* .

[ICH90] ICHINOSE, N. (1990). *Guide pratique des capteurs.* MASSON.

[WIL11] KAFIG, W. (2011). *VHDL 101.* OXFORD: ELSEVIER.

[LAT11] LATTICE. (2011). *LATTICE REFERENCE DESIGNS*. Récupéré sur HTTP//:www.lattice.com

[PHI00] PHILIPS. (2007, JUIN). THE I2C-BUS SPECIFICATION. *VERSION 3* .

[HAM09] Rima, H. (2009). Contribution à une plateforme d'aide à la conception de microsystème. LYON, FRANCE: Institut national des sciences appliquées de Lyon.

[GIN10] Smith, G. R. (2010). *FPGA 101.* New York: ELSEVIER.

[TAV96] TAVERNIER, C. (1996). *Circuits logiques programmables.* PARIS: DUNOD.

[JAC00] WEBER, J. (2000). *Le language VHDL.* PARIS: DUNOD.

[WIK11] Wikipédia, (2011). *Verilog*. Wikipedia.org: fr.wikipedia.org

[UG070] Xilinx. (2008, Decembre 1). *UG070 (v2.6) December 1, 2008*. Xilinx.com: http://www.xilinx.com/bvdocs/publications/ug070.pdf

[DS112] Xilinx. (2010, Aout 30). *DS112 V3.1*. XILINX.COM: http://www.xilinx.com/bvdocs/publications/ds112.pdf

[UG500] XILINX. (2008). UG500. *user guide ISE9.2* .

[XIL02] Xilinx. (2002). XAPP385. *XILINX APPLICATIONS* .

[ASI00] http://asi.insa-rouen.fr/enseignement/siteUV/capteur/

[MIC02] http://michel.hubin.pagesperso-orange.fr/capteurs/techno/chap56.htm

[STU04] Stuart R. Ball (2004) Analog Interfacing to Embedded Microprocessor Systems, ELSEVIER.

[RIC05]Richard Munden (2005) ASIC AND FPGA VERIFICATION: A GUIDE TO COMPONENT MODELING, ELSEVIER.

[COF06] R.C. Cofer and Benjamin F. Harding (2006) Rapid System Prototyping with FPGAs, NEWNES

[DAV08] Justin Davis et Robert Reese (2008) Finite State Machine:Datapath Design, Optimization, and Implementation, Morgan & Claypool.

[KIT08] Charles Kitchin and Lew CountsA (2008) Designer's Guide to Instrumentation Amplifiers 2ND Edition, ANALOG DEVICES.

[NEA09] Neal Stollon (2009) On-Chip Instrumentation Design and Debug for Systems on Chip, SPRINGER.

[MOH05] MOHAMMAD ILYAS et IMAD MAHGOUB (2005) Handbook of Sensor Networks:Compact Wireless and Wired Sensing Systems, CRC PRESS.

```
------------------------------------------------------------------------------------------
-- i2c.vhd
--
-- ce code implémente un controleur I2C en interface avec un uC ou un uP.
--  cette partie du code permet de réunir ensemble l'interface I2C avec l'interface uC.
------------------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity i2c is
  generic (I2C_ADDRESS  : std_logic_vector(15 downto 0):= "0000000000000000" );
  port (
        -- les signaux du bus I2C:
        sda : inout std_logic;
        scl : inout std_logic;

        -- les signaux de l'interface uC:
        addr_bus    : in        std_logic_vector(23 downto 0);
        data_bus    : inout std_logic_vector(7 downto 0);
        as      : in        std_logic;  -- indication de présence d'adresse valide,
        ds      : in        std_logic;  -- indication de présence de données valide,
        r_w     : in        std_logic;  -- read/write
        dtack       : out       std_logic;  -- acquittement de transfert de données,
        irq     : out       std_logic;  -- interrupt request

        mcf     : inout     std_logic;  -- indicateur de fin de transfert,


        -- clock and reset
        clk     : in        std_logic;
        reset       : in        std_logic
    );
end i2c;

library IEEE;
use IEEE.std_logic_1164.all;

architecture behave of i2c is

-- ******************************Definitions des signaux ****************************

-- logique de controle I2C:
component i2c_control
  port(
    -- signaux du bus I2C:
    sda : inout std_logic;
    scl : inout std_logic;

    -- signaux de l'interface avec le uC:
    txak        : in        std_logic;  -- valeur du bit d'acquittement en mode transmission,
    msta        : in        std_logic;  -- selection master/slave
    msta_rst    : out       std_logic;  -- resets MSTA bit, en cas de perte d'arbitrage.
    rsta        : in        std_logic;  -- start répété
    rsta_rst    : out       std_logic;  -- reset pour start répété dans le registre de
controle,
    mtx         : in        std_logic;  -- master read/write
```

```
    mbdr_micro   : in        std_logic_vector(7 downto 0);    -- données du uC à metre en
    sortie sur le bus I2C,
    madr         : in        std_logic_vector(7 downto 0); -- adresse esclave sur l'I2C,
    mbb          : out       std_logic;  -- bus occupé,
    mcf          : inout     std_logic;  -- transfert de données,
    maas         : inout     std_logic;  -- controleur adressé comme esclave,
    mal          : inout     std_logic;  -- perte d'arbitrage,
    srw          : inout     std_logic;  -- slave read/write
    mif          : out       std_logic;  -- interruption a survenue,
    rxak         : out       std_logic;  -- réception d'un acquittement,
    mbdr_i2c     : inout     std_logic_vector(7 downto 0); -- données I2C vers le uC,
    mbcr_wr      : in        std_logic;  -- indication de modification du registre de
    controle MBCR,
    mif_bit_reset    : in        std_logic;  -- indication de la nécessite de reset sur le
    bit MIF,
    mal_bit_reset    : in        std_logic;  -- indication de la nécessite de reset sur le
    bit MAL,


        sys_clk      : in        std_logic;
    reset        : in        std_logic);

end component;

-- definition de l'interface uC:
component uc_interface
      generic (UC_ADDRESS : std_logic_vector(15 downto 0):= "0000000000000000" );
    port(
        -- bus d'interface parallel du uC :
        clk     : in STD_LOGIC;
        reset   : in STD_LOGIC;

        addr_bus    : in STD_LOGIC_VECTOR (23 downto 0);
        data_bus    : inout STD_LOGIC_VECTOR (7 downto 0);
        as      : in STD_LOGIC;     -- indicateur de présence d'adresse valide sur le bus,
        ds      : in STD_LOGIC;     -- indicateur de présence d'données valide sur le bus,
        r_w     : in STD_LOGIC;     -- r_w = 1 lecture, r_w =0 écriture.
        dtack   : out STD_LOGIC;    -- acquittement d'un transfert de données,
        irq     : out STD_LOGIC;    -- Interrupt request

        -- registre interne du controleur I2C
        -- registre d'adresses (contient l'adresse esclave) :
        madr            : inout STD_LOGIC_VECTOR(7 downto 0);

                -- registre de controle :
        men              : inout STD_LOGIC;  -- bit d'activation du controleur I2C,
        mien             : inout STD_LOGIC;  -- autorisation des interruptions,
        msta             : inout STD_LOGIC;  -- bit de séléction de mode Master/Slave
        mtx              : inout STD_LOGIC;  -- Master read/write
        txak             : inout STD_LOGIC;  -- bit d'acquittemment,
        rsta             : inout STD_LOGIC;  -- start répété,

        mbcr_wr          : out STD_LOGIC;    -- bit d'indication de modification du registre
        de controle,
        rsta_rst         : in    STD_LOGIC;  -- bit de reset du registre de controle en cas
        de start répété,
```

```
                -- registre d'état:
        mcf             : in STD_LOGIC;      -- indicateur de fin de transfert de données,
        maas            : in STD_LOGIC;      -- bit d'indication d'un adressage en esclave,
        mbb             : in STD_LOGIC;      -- indique que le bus I2C est occupé,
        mal             : in STD_LOGIC;      -- indicateur de perte d'arbitrage,
        srw             : in STD_LOGIC;      -- slave read/write
        mif             : in STD_LOGIC;      -- interruption a survenue,
        rxak            : in STD_LOGIC;      -- indicateur de reception d'un acquittement,

        mal_bit_reset   : out STD_LOGIC;     -- indicateur que le bit MAL doit être
        réinitialisé,
        mif_bit_reset   : out STD_LOGIC;     -- indicateur que le bit MIF doit être
        réinitialisé,
        msta_rst        : in  STD_LOGIC;     -- réinitialisation du bit MSTA en cas de perte
        d'arbitrage,

                -- registre de données:
        mbdr_micro      : inout STD_LOGIC_VECTOR (7 downto 0);
        mbdr_i2c        : in    STD_LOGIC_VECTOR (7 downto 0);
        mbdr_read       : out STD_LOGIC
        );
end component;

-- ****************************** déclaration des signaux ****************************

                -- registre de controle:
signal madr         : std_logic_vector(7 downto 0);
signal men          : std_logic;
signal mien         : std_logic;
signal msta         : std_logic;
signal mtx          : std_logic;
signal txak         : std_logic;
signal rsta         : std_logic;
signal rsta_rst     : std_logic;
signal mbcr_wr      : std_logic;

                -- registre d'état:
signal mcf          : std_logic;
signal maas         : std_logic;
signal mbb          : std_logic;
signal mal          : std_logic;
signal srw          : std_logic;
signal mif          : std_logic;
signal rxak         : std_logic;

signal mal_bit_reset    : std_logic;
signal mif_bit_reset    : std_logic;
signal msta_rst         : std_logic;

                -- registres de données:
signal mbdr_micro       : std_logic_vector(7 downto 0); -- registre de données venant du uC,
signal mbdr_i2c         : std_logic_vector(7 downto 0); -- registre de données venant du bus
i2c,
signal mbdr_read        : std_logic;    -- indication de lecture du registre mbdr_i2c par le
uC,

begin
```

```
-- ****************************** couplage des signaux ****************************

I2C_CTRL: i2c_control

    port map (
            -- signaux du bus I2C:
            sda => sda,
            scl => scl,

            -- signaux d'interface venant du uC:
            txak        => txak,
            msta        => msta,
            msta_rst    => msta_rst,
            rsta        => rsta,
            rsta_rst    => rsta_rst,
            mtx         => mtx,
            mbdr_micro  => mbdr_micro,
            madr        => madr,
            mbb     => mbb,
            mcf     => mcf,
            maas    => maas,
            mal     => mal,
            srw     => srw,
            mif     => mif,
            rxak        => rxak,
            mbdr_i2c    => mbdr_i2c,
            mbcr_wr     => mbcr_wr,
            mif_bit_reset   => mif_bit_reset,
            mal_bit_reset   => mal_bit_reset,
            sys_clk  => clk,
            reset    => men
        );

-- couplage des signaux avec le uC

uC_CTRL: uc_interface
    generic map ( UC_ADDRESS => I2C_ADDRESS)
    port map(
        -- interface parallel du uC:
        clk     => clk,
        reset   => reset,

        addr_bus    => addr_bus,
        data_bus    => data_bus,
        as      => as,
        ds      => ds,
        r_w     => r_w,
        dtack   => dtack,
        irq     => irq,

        -- registres internes du controleur I2C:
                -- registre d'adresses:
        madr          => madr,

                -- registre de controle:
        men         => men,
```

```
        mien          => mien,
        msta          => msta,
        mtx           => mtx,
        txak          => txak,
        rsta          => rsta,

        mbcr_wr       => mbcr_wr,
        rsta_rst      =>  rsta_rst,

                -- registre d'état:
        mcf               => mcf,
        maas              => maas,
        mbb               => mbb,
        mal               => mal,
        srw               => srw,
        mif               => mif,
        rxak              => rxak,

        mal_bit_reset     => mal_bit_reset,
        mif_bit_reset     => mif_bit_reset,
        msta_rst          => msta_rst,


                -- registre de données:
        mbdr_micro        => mbdr_micro,
        mbdr_i2c          => mbdr_i2c,

        mbdr_read         => mbdr_read
        );

end behave;
```

```vhdl
-- i2c_control.vhd
--
--  ce code source, est le code source principal qui implémente
--  le contrôleur I2C.
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity i2c_control is

  port(
        -- signaux du bus I2C:
    sda : inout std_logic;
    scl : inout std_logic;
        -- signaux de l'interface avec le uC:
    txak        : in        std_logic;  -- valeur du bit d'acquittement en mode transmission,
    msta        : in        std_logic;  -- selection master/slave
    msta_rst    : out       std_logic;  -- resets MSTA bit, en cas de perte d'arbitrage.
    rsta        : in        std_logic;  -- start répété
    rsta_rst    : out       std_logic;  -- reset pour start répété dans le registre de
                                        -- controle,
    mtx         : in        std_logic;  -- master read/write
    mbdr_micro  : in        std_logic_vector(7 downto 0);   -- données du uC à metre en
                                                            -- sortie sur le bus I2C,
    madr        : in        std_logic_vector(7 downto 0); -- adresse esclave sur l'I2C,
    mbb         : out       std_logic;  -- bus occupé,
    mcf         : inout     std_logic;  -- transfert de données,
    maas        : inout     std_logic;  -- controleur adressé comme esclave,
    mal         : inout     std_logic;  -- perte d'arbitrage,
    srw         : inout     std_logic;  -- slave read/write
    mif         : out       std_logic;  -- interruption a survenue,
    rxak        : out       std_logic;  -- réception d'un acquittement,
    mbdr_i2c    : inout     std_logic_vector(7 downto 0); -- données I2C vers le uC,
    mbcr_wr     : in        std_logic;  -- indication de modification du registre
                                        -- de controle MBCR,
    mif_bit_reset   : in    std_logic;  -- indication de la nécessite de reset sur le bit MIF
    mal_bit_reset   : in    std_logic;  -- indication de la nécessite de reset sur le bit MAL

    sys_clk     : in        std_logic;
    reset       : in        std_logic);
end i2c_control;

library IEEE;
use IEEE.std_logic_1164.all;

architecture behave of i2c_control is
-- constantes pour le comptage des demi-périodes haute et basse
constant    CNT_100KHZ  :   std_logic_vector(4 downto 0)  := "10100";-- nombre dimpulsion
                                                        -- d'horloge pour compter 100KHz
constant    HIGH_CNT    :   std_logic_vector(3 downto 0)  := "1000"; -- nombre dimpulsion
                                              -- d'horloge pour compter une demi
                                              période
constant    LOW_CNT     :   std_logic_vector(3 downto 0)  := "1000"; -- nombre dimpulsion
                                              -- d'horloge pour compter une demi période
constant    HIGH_CNT_2  :   std_logic_vector(3 downto 0)  := "0100"; -- moitié HIGH_CNT
```

```
constant    TBUF        :   std_logic_vector(3 downto 0) := "1001"; -- nombre d'impulsions
                                                            -- d'horloge pour compter 4.7uS
constant    DATA_HOLD   :   std_logic_vector(3 downto 0) := "0001"; -- nombre d'impulsions
                                                            -- d'horloge pour compter 300ns
constant    START_HOLD  :   std_logic_vector(3 downto 0) := "1000"; -- nombre d'impulsions
                                                            -- d'horloge pour compter 4.0uS
constant    CLR_REG     :       std_logic_vector (7 downto 0) := "00000000";
constant    START_CNT       :   std_logic_vector (3 downto 0) := "0000";
constant    CNT_DONE        :   std_logic_vector (3 downto 0) := "0111";
constant    ZERO_CNT    :   std_logic_vector (3 downto 0) := "0000";
constant    ZERO        :   std_logic := '0';
constant    RESET_ACTIVE    :   std_logic := '0';

-- registre à décalage 8-bit:
component SHIFT8
    port(
         clk           : in STD_LOGIC;    -- Clock
         clr           : in STD_LOGIC;    -- effacement
         data_ld       : in STD_LOGIC;    -- début de chargement de données.
         data_in       : in STD_LOGIC_VECTOR (7 downto 0); -- données en entrées
         shift_in      : in STD_LOGIC;    -- données série en entrées.
         shift_en      : in STD_LOGIC;    -- activation du décalage.
         shift_out     : out STD_LOGIC;   -- décalage de données en sortie.
         data_out      : out STD_LOGIC_VECTOR (7 downto 0)   -- données en sortie.
         );
end component;

-- compteur 4 bits:
component UPCNT4
    port(
         data          : in STD_LOGIC_VECTOR (3 downto 0);    -- données série en entrées
         cnt_en        : in STD_LOGIC;                        -- activation du comptage
         load          : in STD_LOGIC;                  -- activation de la ligne de données
         clr           : in STD_LOGIC;                        -- effacement du compteur
         clk           : in STD_LOGIC;                        -- Clock
         qout          : inout STD_LOGIC_VECTOR (3 downto 0));

end component;
type state_type is (IDLE, HEADER, ACK_HEADER, RCV_DATA, ACK_DATA,
                    XMIT_DATA, WAIT_ACK);
signal state        : state_type;

type scl_state_type is (SCL_IDLE, START, SCL_LOW_EDGE, SCL_LOW, SCL_HIGH_EDGE,
                        SCL_HIGH, STOP_WAIT);
signal scl_state, next_scl_state        : scl_state_type;

signal scl_in       : std_logic;    -- signal SCL en entrée
signal scl_out      : std_logic;    -- signal SCL de sortie (signal interne)
signal scl_out_reg  : std_logic;    -- registre du siganl SCL_OUT
signal scl_not      : std_logic;    -- inversion du signal SCL
signal sda_in       : std_logic;    -- signal SDA en entrée
signal sda_out      : std_logic;    -- signal SCL de sortie (signal interne)
signal sda_out_reg  : std_logic;    -- registre du siganl SDA_OUT
signal sda_out_reg_d1   : std_logic;    -- Temporisation sur la ligne sda pour
                                        -- la comparaison en cas d'arbitrage
signal slave_sda        : std_logic;    -- valeur sda en mode esclave
```

```
signal master_sda        : std_logic;    -- valeur sda en mode maitre

signal sda_oe       : std_logic;

signal master_slave : std_logic;    -- 1 pour master, 0 pour slave

-- liaisons entre le registre de controle et le registre à décalage
signal shift_reg        : std_logic_vector(7 downto 0); -- registre à décalage qui
                                                        -- retient les données du bus
                                                        I2C
signal shift_out        : std_logic;
signal shift_reg_en, shift_reg_ld   : std_logic;
signal i2c_header       : std_logic_vector(7 downto 0); -- Registre à décalage qui retient
                                                        -- les données de l'état header
signal i2c_header_en, i2c_header_ld : std_logic;
signal i2c_shiftout : std_logic;

-- détéction de l'adresse reçu est la même avec l'adresse esclave
signal addr_match  : std_logic;

signal arb_lost     : std_logic;    -- 1 en cas de perte d'arbitrage
signal msta_d1      : std_logic;    -- temporisation pour le bit msta

signal detect_start : std_logic;    -- incation de détéction d'un START
signal detect_stop  : std_logic;    -- incation de détéction d'un STOP
signal sm_stop      : std_logic;    -- indication de la nécessité de générer un STOP par
                                    -- la machine à état.
signal bus_busy     : std_logic;    -- indication si le bus est occupé,
                                    -- positionné à 1 après un START et à 0 après un STOP
signal bus_busy_d1  : std_logic;    -- temporisation pour Bus_busy pour déterminer le bit mal
signal gen_start        : std_logic;    -- indique quand le uC veut générer un START;
signal gen_stop     : std_logic;    -- indique quand le uC veut générer un STOP
signal rep_start        : std_logic;    -- indique quand le uC veut générer un START répété
signal stop_scl     : std_logic;    -- indicateur de STOP dans la machine à étatss
signal stop_scl_reg : std_logic;    -- registre de l'état STOP_SCL

-- compteur de Bits de 0 à 7
signal bit_cnt      : std_logic_vector(3 downto 0);
signal bit_cnt_ld, bit_cnt_clr, bit_cnt_en : std_logic;

-- compteur d'impulsions d'horloge
signal clk_cnt      : std_logic_vector (3 downto 0);
signal clk_cnt_rst  : std_logic;
signal clk_cnt_en   : std_logic;

signal reg_clr      : std_logic_vector(7 downto 0);
signal zero_sig     : std_logic;
signal cnt_zero     : std_logic_vector(3 downto 0);
signal cnt_start        : std_logic_vector(3 downto 0);

begin

  -- valeurs des sorties SDA et SCL
    sda <= '0' when sda_oe = '1' else 'Z';
    scl <= '0' when scl_out_reg = '0' else 'Z';
    scl_not <= not(scl);
```

```
    sda_oe <= '1' when ((master_slave = '1' and arb_lost = '0' and sda_out_reg = '0') or
                        (master_slave = '0' and slave_sda = '0')
                         or stop_scl_reg = '1') else '0';

reg_clr <= CLR_REG;
zero_sig <= ZERO;
cnt_zero <= ZERO_CNT;
cnt_start <= START_CNT;


-- ************************  Procédure d'arbitrage ************************************
-- cette procédure vérifie les données en sortie du maitre et les données établies sur le bus
--a fin de déterminer si le maitre controle toujours le bus. les données de la ligne SDA sont
-- vérifiées seulement durant l'état haut de la ligne SCL et durant les états: IDLE, HEADER,
-- XMIT_DATA pour ne pas envoyer les conditions START et STOP au cas où le bus est occupé.

arbitration: process (sys_clk, reset)
  begin
    if reset = RESET_ACTIVE then
        arb_lost <= '0';
        msta_rst <= '0';
        elsif (sys_clk'event and sys_clk = '1') then
        if scl_state = SCL_IDLE then
            arb_lost <= '0';
            msta_rst <= '0';
        elsif (master_slave = '1') then -- vérification de perte d'arbitrage en mode maitre
                                        -- seulement coparaison de données seulement quand
                                        -- scl=1 et vérifier si le bit arb_lost=0

            if (scl_in = '1' and scl = '1' and arb_lost = '0'
                and (state = HEADER or  state = XMIT_DATA or state = IDLE)) then

                if sda_out_reg_d1 = sda_in then
                    arb_lost <= '0';
                    msta_rst <= '0';
                else
                    arb_lost <= '1';
                    msta_rst <= '1';
                end if;
            else
                arb_lost <= arb_lost;
                msta_rst <= '0';
            end if;
        end if;

        end if;

  end process;

-- ********  Procédure de génération des signaux sur la ligne SCL ********************
-- cette procédure génère les signaux SCL et SDA en mode maitre. Elle génère les
-- conditions start et stop. En cas de perte d'arbitrage, l'horloge sur la ligne
-- SCL sera généré jusqu'au transfert du dernier octet.

scl_generator_comb: process (scl_state, arb_lost, sm_stop, gen_stop, rep_start,
                    bus_busy, gen_start, master_slave, stop_scl_reg,
                    clk_cnt, bit_cnt, scl_in, state, sda_out,
```

```
                    sda_out_reg, master_sda)

begin

-- valeurs par défaut de la machine à états
    scl_out <= '1';
    sda_out <= sda_out_reg;
    stop_scl <= stop_scl_reg;
    clk_cnt_en <= '0';
    clk_cnt_rst <= '1';
    next_scl_state <= scl_state;
    rsta_rst <= not(RESET_ACTIVE);

        case scl_state is

            when SCL_IDLE =>
                sda_out <= '1';
                stop_scl <= '0';
                if master_slave = '1' and bus_busy = '0' and gen_start = '1' then
                    next_scl_state <= START;
                end if;

            when START =>
                -- génération de la condition START
                clk_cnt_en <= '1';
                clk_cnt_rst <= '0';
                sda_out <= '0';
                stop_scl <= '0';
                -- génération du reset pour un start répété
                if rep_start = '1' then
                    rsta_rst <= RESET_ACTIVE;
                end if;
                if clk_cnt = START_HOLD then
                    next_scl_state <= SCL_LOW_EDGE;
                else
                    next_scl_state <= START;
                end if;

            when SCL_LOW_EDGE =>
                clk_cnt_rst <= '1';
                scl_out <= '0';
                next_scl_state <= SCL_LOW;
                stop_scl <= '0';

            when SCL_LOW =>
                clk_cnt_en <= '1';
                clk_cnt_rst <= '0';
                scl_out <= '0';

                -- valeut de SDA_OUT en fonction des signaux de contrôle
                if arb_lost = '1' then
                    stop_scl <= '0';
                elsif ((sm_stop = '1' or gen_stop = '1') and
                    (state /= ACK_DATA and state /= ACK_HEADER and state /= WAIT_ACK)) then
                    sda_out <= '0';
                    stop_scl <= '1';
                elsif rep_start = '1' then
```

```
                sda_out <= '1';
                stop_scl <= '0';
            elsif clk_cnt = DATA_HOLD then
                sda_out <= master_sda;
                stop_scl <= '0';
            else
                stop_scl <= '0';
            end if;

            -- calcule des états future
            if clk_cnt = LOW_CNT then

                if bit_cnt = CNT_DONE and arb_lost = '1' then
                    next_scl_state <= SCL_IDLE;
                else
                    next_scl_state <= SCL_HIGH_EDGE;
                end if;
            else
                next_scl_state <= SCL_LOW;
            end if;

        when SCL_HIGH_EDGE =>
            clk_cnt_rst <= '1';
            scl_out <= '1';
            if ((sm_stop = '1' or gen_stop = '1') and
               (state /= ACK_DATA and state /= ACK_HEADER and state /= WAIT_ACK)) then
                stop_scl <= '1';
            else
                stop_scl <= '0';
            end if;

            if scl_in = '0' then
                next_scl_state <= SCL_HIGH_EDGE;
            else
                next_scl_state <= SCL_HIGH;
            end if;

        when SCL_HIGH =>
            -- la ligne scl est à l'état haut, début de comptage pour l'état haut
            clk_cnt_en <= '1';
            clk_cnt_rst <= '0';
            scl_out <= '1';
            -- vérification si un start répété ou un stop sont nécessaire
            if clk_cnt = HIGH_CNT_2 then
                if rep_start = '1' then
                    next_scl_state <= START;
                    clk_cnt_rst <= '1';
                elsif stop_scl_reg = '1' then
                    next_scl_state <= STOP_WAIT;
                    clk_cnt_rst <= '1';
                end if;
            elsif clk_cnt = HIGH_CNT then
                next_scl_state <= SCL_LOW_EDGE;
            else
                next_scl_state <= SCL_HIGH;
            end if;
```

```
            when STOP_WAIT =>
                -- cet état temporise le temps nécéssaire entre un start et un stop
                clk_cnt_en <='1';
                clk_cnt_rst <= '0';
                sda_out <= '1';
                stop_scl <= '0';
                if clk_cnt = TBUF then
                    next_scl_state <= SCL_IDLE;
                else
                    next_scl_state <= STOP_WAIT;
                end if;
        end case;
end process;

scl_generator_regs: process (sys_clk, reset)
begin
    if reset = RESET_ACTIVE then
        scl_state <= SCL_IDLE;
        sda_out_reg <= '1';
        scl_out_reg <= '1';
        stop_scl_reg <= '0';
    elsif sys_clk'event and sys_clk='1' then
        scl_state <= next_scl_state;
        sda_out_reg <= sda_out;
        scl_out_reg <= scl_out;
        stop_scl_reg <= stop_scl;
    end if;
end process;

-- *******  Implémentation du compteur d'impulsions d'horloge ************************
-- cette partie du code permet d'implémenter le compteur qui va permettre la génération
-- de l'horloge sur la ligne SCL à base de l'horloge système.

    CLKCNT : UPCNT4
      port map( data      => cnt_zero,
                cnt_en    => clk_cnt_en,
                load      => clk_cnt_rst,
                clr       => reset,
                clk       => sys_clk,
                qout      => clk_cnt );

-- ************************  les registres d'entrées ************************************
-- cette procédure permet de convertir la fréquence des signaux scl et sda à
-- celle de la fréquence du système.

input_regs: process(sys_clk,reset)
begin
    if reset = RESET_ACTIVE then
        sda_in <= '1';
        scl_in <= '1';
        msta_d1 <= '0';
        sda_out_reg_d1 <= '1';

    elsif sys_clk'event and sys_clk = '1' then

        if scl = '0' then
            scl_in <= '0';
```

```
        else
            scl_in <= '1';
        end if;
        if sda = '0' then
            sda_in <= '0';
        else
            sda_in <= '1';
        end if;
        sda_out_reg_d1 <= sda_out_reg;
        msta_d1 <= msta;
    end if;
end process;

-- ************  Procédure de détéction de START/STOP **********************************
-- cette procédure détécte les conditions start et stop en utilisant sda comme horloge.
start_det: process(sda, reset, state)
begin
    if reset = RESET_ACTIVE or state = HEADER then
        detect_start <= '0';
    elsif sda'event and sda = '0' then
        if scl /= '0' then
            detect_start <= '1';
        else
            detect_start <= '0';
        end if;
    end if;
end process;

stop_det: process(sda, reset, detect_start)
begin
    if reset = RESET_ACTIVE or detect_start = '1' then
        detect_stop <= '0';
    elsif sda'event and sda /= '0' then
        if scl /= '0' then
            detect_stop <= '1';
        else
            detect_stop <= '0';
        end if;
    end if;
end process;

-- ************ Procédure de détection de l'état du bus ********************************
-- Cette procédure détecte les conditions start et stop et positionne le bit bus_busy

set_bus_busy: process(sys_clk,reset)
begin
    if reset = RESET_ACTIVE then
        bus_busy <= '0';
        bus_busy_d1 <= '0';

    elsif sys_clk'event and sys_clk = '1' then

        bus_busy_d1 <= bus_busy;

            if detect_start = '1' then
                bus_busy <= '1';
            end if;
```

```
            if detect_stop = '1' then
                bus_busy <= '0';
            end if;
        end if;

end process;

-- ************************ les bits de contrôle du uC **********************************
-- cette procédure détécte les front montant et déscendant du signal MSTA et
-- positionne les signaux pour générer les conditions start et stop.
control_bits: process (sys_clk,reset)
begin
    if reset = RESET_ACTIVE then
        gen_start <= '0';
        gen_stop <= '0';
        master_slave <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if msta_d1 = '0' and msta = '1' then
            -- front montant du signal MSTA - génération de la condition start
            gen_start <= '1';
        elsif detect_start = '1' then
            gen_start <= '0';
        end if;
        if arb_lost = '0' and msta_d1 = '1' and msta = '0' then
            -- front déscendant du signal MSTA - génération de la condition stop seulement
            -- en cas de controle du bus.
            gen_stop <= '1';
        elsif detect_stop = '1' then
            gen_stop <= '0';
        end if;
        if bus_busy = '0' then
            master_slave <= msta;
        else
            master_slave <= master_slave;
        end if;
    end if;
end process;

rep_start <= rsta;

-- ************************  les bits d'état du uC **************************************
-- la procédure pour les relie les bits du uC avec le registre d'état mbsr
mcf_bit: process(scl, reset)
begin
    if reset = RESET_ACTIVE then
        mcf <= '0';
    elsif scl'event and scl = '0' then
        if bit_cnt = CNT_DONE then
            mcf <= '1';
        else
            mcf <= '0';
        end if;
    end if;
end process;

maas_bit: process(sys_clk, reset)
begin
```

```
    if reset = RESET_ACTIVE  then
        maas <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if mbcr_wr = '1' then
            maas <= '0';
        elsif state = ACK_HEADER then
            maas <= addr_match; -- le signal addr_match compare madr avec I2C_addr
        else
            maas <= maas;
        end if;
    end if;
end process;

mbb <= bus_busy;

-- MAL - bit de perte d'arbitrage.
mal_bit: process(sys_clk, reset)
begin
    if reset = RESET_ACTIVE  then
        mal <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if mal_bit_reset = '1' then
            mal <= '0';
        elsif master_slave = '1' then
            if (arb_lost = '1') or
               (bus_busy_d1 = '1' and gen_start = '1') or
               (detect_stop = '1' and gen_stop = '0' and sm_stop = '0') then
                    mal <= '1';
            end if;
        elsif rsta = '1' then
                mal <= '1';
        end if;
    end if;
end process;

-- SRW - bit de lecture /écriture pour esclave
srw_bit: process(sys_clk, reset)
begin
    if reset = RESET_ACTIVE then
        srw <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if state = ACK_HEADER then
            srw <= i2c_header(0);
        else
            srw <= srw;
        end if;
    end if;
end process;

mif_bit: process(sys_clk, reset)
begin
    if reset = RESET_ACTIVE then
        mif <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if mif_bit_reset = '1' then
            mif <= '0';
        elsif mal = '1' or mcf = '1' or
```

```
            (maas = '1' and i2c_header(0) = '0' and master_slave = '0') then
            mif <= '1';
        end if;
    end if;
end process;

-- RXAK - réception d'acqittement
rxak_bit: process(scl)
begin
    if scl'event and scl = '0' then
        if state = ACK_HEADER or state = ACK_DATA or state = WAIT_ACK then
            rxak <= sda_in;
        end if;
    end if;
end process;

-- ************************  registre de données du uC   ********************************
-- Registre d'interface mbdr_I2C, contient les données pour le uC
mbdr_i2c_proc: process(sys_clk, reset)
begin
    if reset = RESET_ACTIVE then
        mbdr_i2c <= (others => '0');
    elsif sys_clk'event and sys_clk = '1' then
        if (state = ACK_DATA) or (state = WAIT_ACK) then
            mbdr_i2c <= shift_reg ;
        else
            mbdr_i2c <= mbdr_i2c;
        end if;
    end if;
end process;

-- ************************  Registre d'adresse pour le uC ******************************
    addr_match <= '1' when i2c_header(7 downto 1) = madr(7 downto 1)
                  else '0';

-- ************************  Machine à état principale **********************************
-- la machine à état suivante, est la machine à états principale pour le mode maitre
-- et le mode esclave.
state_machine: process (scl, reset, detect_stop)

begin

    if reset = RESET_ACTIVE or detect_stop = '1' then
        state <= IDLE;
        sm_stop <= '0';
    elsif scl'event and scl = '0'  then

        case state is

            ------------- L'état IDLE -------------
            when IDLE =>
                if detect_start = '1' then
                    state <= HEADER;
                end if;

            ------------- L'état HEADER -------------
            when HEADER =>
```

```
        if bit_cnt = CNT_DONE then
            state <= ACK_HEADER;
        end if;

    ------------- L'état ACK_HEADER -------------
    when ACK_HEADER =>
        if arb_lost = '1' then
            state <= IDLE;
        elsif sda_in = '0' then
        -- réception d'un acquittement, vérification de mode maitre/esclave
            if master_slave = '1' then
        -- mode maitre, vérification du bit mtx pour déterminer le sens du transfert
                if mtx = '0' then
                    -- mode réception
                    state <= RCV_DATA;
                else
                    -- mode émition
                    state <= XMIT_DATA;
                end if;
            else
                if addr_match = '1' then
                -- si le bit maas=1, adréssé comme esclave,
                -- vérifier le bit I2C_HEADER(0) pour le sens du transfert
                    if i2c_header(0) = '0' then
                        -- mode réception
                        state <= RCV_DATA;
                    else
                        -- mode émition
                        state <= XMIT_DATA;
                    end if;
                else
                    -- le controleur I2C n'est pas adréssé, retour à l'état IDLE
                    state <= IDLE;
                end if;
            end if;
        else
            -- Pas d'acquittement reçu, évolution vers l'état stop.
            state <= IDLE;
            if master_slave = '1' then
                sm_stop <= '1';
            end if;

        end if;

    ------------- état RCV_DATA  --------------
    when RCV_DATA =>

      -- vérification d'un start répété
      if (detect_start = '1') then
       state <= HEADER;
      elsif bit_cnt = CNT_DONE then

      -- envoie d'un acquittement
       state <= ACK_DATA;

      end if;
```

```
            ------------ état XMIT_DATA --------------
                when XMIT_DATA =>

              -- envoie d'un acquittement
              if (detect_start = '1') then
               state <= HEADER;

              elsif bit_cnt = CNT_DONE then

               -- attente d'un acquittement
               state <= WAIT_ACK;

              end if;

            ------------- état ACK_DATA --------------
                when ACK_DATA =>

                 state <= RCV_DATA;

            ------------- état WAIT_ACK --------------
            when WAIT_ACK =>
            if arb_lost = '1' then
                    state <= IDLE;
              elsif (sda = '0') then
                  state <= XMIT_DATA;
             else
             -- pas d'acquittement reçu, génération d'un stop et retour à l'état IDLE
                if master_slave = '1' then
                    sm_stop <= '1';
                end if;
                  state <= IDLE;
             end if;

          end case;

        end if;

    end process;

-- ***************  Le signal SDA en mode maitre et esclave ******************
slv_mas_sda: process(reset, sys_clk)
begin
    if reset = RESET_ACTIVE then
        master_sda <= '1';
        slave_sda <= '1';
    elsif sys_clk'event and sys_clk = '1' then
        if state = HEADER or state = XMIT_DATA then
            master_sda <= shift_out;
        elsif state = ACK_DATA then
            master_sda <= TXAK;
        else
            master_sda <= '1';
        end if;
        -- le signal sda en mode esclave, vérifier si l'adresse correspond,
        -- nécéssaire seulement dans l'état ACK_HEADER.

        if (maas = '1' and state = ACK_HEADER) or
```

```
           (state = ACK_DATA) then
            slave_sda <= TXAK;
        elsif (state = XMIT_DATA) then
            slave_sda <= shift_out;
        else
            slave_sda <= '1';
        end if;
    end if;
end process;


-- **************************  registre à décalage pour l'I2C ***************************
    I2CDATA_REG: SHIFT8
      port map (
              clk       => scl_not,
              clr       => reset,
              data_ld   => shift_reg_ld,
              data_in   => mbdr_micro,
              shift_in  => sda_in,
              shift_en  => shift_reg_en,
              shift_out => shift_out,
              data_out  => shift_reg );

i2cdata_reg_ctrl: process(sys_clk, reset)

begin
    if reset = RESET_ACTIVE then
        shift_reg_en <= '0';
        shift_reg_ld <= '0';
    elsif sys_clk'event and sys_clk = '1' then

        if ((master_slave = '1' and state = HEADER)
            or (state = RCV_DATA) or (state = XMIT_DATA)) then
            shift_reg_en <= '1';
        else
            shift_reg_en <= '0';
        end if;

        if ((master_slave = '1' and state = IDLE) or (state = WAIT_ACK)
             or (state = ACK_HEADER and i2c_header(0) = '1' and master_slave = '0')
         or (state = ACK_HEADER and mtx = '1' and master_slave = '1')
         or (detect_start = '1')  ) then
                shift_reg_ld <= '1';
        else
                shift_reg_ld <= '0';
        end if;
    end if;
end process;



-- *************  registre à décalage pour l'état header de l'I2C ***********************
    -- registre à décalage pour l'état Header en phase d'adressage
    I2CHEADER_REG: SHIFT8
      port map (
              clk       => scl_not,
              clr       => reset,
```

```
              data_ld   => i2c_header_ld,
              data_in   => reg_clr,
              shift_in  => sda_in,
              shift_en  => i2c_header_en,
              shift_out => i2c_shiftout,
              data_out  => i2c_header );

i2cheader_reg_ctrl: process(sys_clk, reset)

begin
    if reset = RESET_ACTIVE then
        i2c_header_en <= '0';
    elsif sys_clk'event and sys_clk = '1' then
        if (detect_start = '1') or (state = HEADER) then
            i2c_header_en <= '1';
        else
            i2c_header_en <= '0';
        end if;
    end if;
end process;

    i2c_header_ld <= '0';

-- ************************  compteur de Bits *******************************************
    BITCNT : UPCNT4
      port map(   data      => cnt_start,
              cnt_en    => bit_cnt_en,
              load      => bit_cnt_ld,
              clr       => reset,
              clk       => scl_not,
              qout      => bit_cnt );

    -- les signaux de controle du compteur
    bit_cnt_en <= '1' when (state = HEADER) or (state = RCV_DATA)
                    or (state = XMIT_DATA) else '0';

    bit_cnt_ld <= '1' when (state = IDLE) or (state = ACK_HEADER)
                    or (state = ACK_DATA)
                    or (state = WAIT_ACK)
                    or (detect_start = '1') else '0';

end behave;
```

```
-- File:     uC_interface.vhd
--
-- Ce code source, décrit l'interface entre le uC avec des bus parallel avec les registres
-- internes du contrôleur I2C. Ces registres internes permettent de gérer le contrôleur I2C
-- et de le configurer soit en mode maitre ou esclave, en émition et en récerption.
-----------------------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity uC_interface is
    generic (UC_ADDRESS : std_logic_vector(15 downto 0):= "0000000000000000"  );
    port(
        -- bus parallel d'interface uC
        clk     : in STD_LOGIC;
        reset   : in STD_LOGIC;

        addr_bus    : in STD_LOGIC_VECTOR (23 downto 0);
        data_bus    : inout STD_LOGIC_VECTOR (7 downto 0);
        as      : in STD_LOGIC;     -- assertion d'adresse, actif à l'état bas.
        ds      : in STD_LOGIC;     -- assertion de données, actif à l'état bas.

        -- indication de direction des signaux
        r_w     : in STD_LOGIC; -- état bas écriture, état haut lecture.
        dtack   : out STD_LOGIC;    -- acquittement de transfert de données
        irq     : out STD_LOGIC;    -- demande d'intéruption.

        -- registre internes du contrôleur I2C:
        -- adresse des registres (avec adresse d'esclave)
        madr          : inout STD_LOGIC_VECTOR(7 downto 0); -- registre d'adresse
                -- registre de contrôle
        men             : inout STD_LOGIC;  -- bit d'activation du contrôleur I2C
        mien            : inout STD_LOGIC;  -- bit d'autorisation d'interruption, actif bas
        msta            : inout STD_LOGIC;  -- bit de selection de mode Master/Slave
        mtx             : inout STD_LOGIC;  -- mode maitre, sélection lecture/écriture
        txak            : inout STD_LOGIC;  -- bit d'autorisation d'envoie d'acquittement
        rsta            : inout STD_LOGIC;  -- start répété

        mbcr_wr         : out STD_LOGIC;    -- indication de  modification du registre
                                            -- de contrôle
        rsta_rst    : in  STD_LOGIC;        -- reset du bit rsta
                -- registre d'état:
        mcf             : in STD_LOGIC;     -- indicateur de fin de transfert de données,
        maas            : in STD_LOGIC;     -- bit d'indication d'un adressage en esclave,
        mbb             : in STD_LOGIC;     -- indique que le bus I2C est occupé,
        mal             : in STD_LOGIC;     -- indicateur de perte d'arbitrage,
        srw             : in STD_LOGIC;     -- slave read/write
        mif             : in STD_LOGIC;     -- interruption a survenue,
        rxak            : in STD_LOGIC;     -- indicateur de reception d'un acquittement,

        mal_bit_reset   : out STD_LOGIC;    -- indicateur que le bit MAL doit
                                            -- être réinitialisé,
        mif_bit_reset   : out STD_LOGIC;    -- indicateur que le bit MIF doit
                                            -- être réinitialisé,
        msta_rst        : in  STD_LOGIC;    -- réinitialisation du bit MSTA en cas de
                                            -- perte d'arbitrage,
```

```vhdl
                -- registre de données:
        mbdr_micro      : inout STD_LOGIC_VECTOR (7 downto 0);
        mbdr_i2c        : in    STD_LOGIC_VECTOR (7 downto 0);
        mbdr_read       : out STD_LOGIC
        );

end uC_interface;

architecture BEHAVIOUR of uC_interface is

-- déclaration des constantes
constant RESET_ACTIVE : STD_LOGIC := '0';

-- adresse de base du module I2C (addr_bus[23:8])
constant MBASE  : STD_LOGIC_VECTOR(15 downto 0) := UC_ADDRESS;

-- Adresse des registres ( 5 bits ):
-- registre d'adresse (MBASE + 141h)
constant MADR_ADDR  : STD_LOGIC_VECTOR(7 downto 0) := "10001101";

-- registre de contrôle (MBASE + 145h)
constant MBCR_ADDR  : STD_LOGIC_VECTOR(7 downto 0) := "10010001";

-- registre d'état (MBASE + 147h)
constant MBSR_ADDR  : STD_LOGIC_VECTOR(7 downto 0) := "10010011";

-- registre de données (MBASE + 149h)
constant MBDR_ADDR  : STD_LOGIC_VECTOR(7 downto 0) := "10010101";

-- les signaux de la machine à états (machhine d'interface avec le uC)
type STATE_TYPE is (IDLE, ADDR, DATA_TRS, ASSERT_DTACK);

-- Déclaration des signaux

signal as_int     : STD_LOGIC;
signal as_int_d1      : STD_LOGIC;
signal ds_int     : STD_LOGIC;
signal addr_ready : STD_LOGIC;
signal dtack_int,dtack_com,dtack_oe  : STD_LOGIC;
signal data_out : std_logic_vector(7 downto 0); -- contient les données à placer
                                                -- en sotie sur le bus de données
signal data_in  : std_logic_vector(7 downto 0); -- contient les données reçues en
                                                -- enentrée du uC
-- signaux d'état pour la machine à états
signal prs_state, next_state : STATE_TYPE;

-- adresse valide
signal address_match    : std_logic;

-- signaux de selection des registres
signal addr_en      : std_logic;    -- sélection du registre d'adresse
signal cntrl_en     : std_logic;    -- sélection du registre de contrôle
signal stat_en      : std_logic;    -- sélection du registre d'état
signal data_en      : std_logic;    -- sélection du registre de données

begin
        -- siganl d'interruption pour uC
```

```vhdl
    irq <= '0' when (mien = '1') and (mif = '1') else 'Z';

    -- signal d'acquittement vers le uC
    dtack <= dtack_int when (dtack_oe = '1') else 'Z';

    -- bus de données bidirectionnel
    data_bus <= data_out when (r_w = '1' and dtack_oe = '1') else (others => 'Z');
    data_in <= data_bus when r_w = '0' else (others => '0');

    -- Process:   SYNCH_INPUTS
    -- procédure de synchronisation du microcontroleur avec l'horloge système
    SYNCH_INPUTS: process(reset, clk)
    begin
        if reset = RESET_ACTIVE then
            as_int <= '1';
            as_int_d1 <= '1';
            ds_int <= '1';
            address_match <= '0';

        elsif clk'event and clk = '1' then
            as_int <= as;
            as_int_d1 <= as_int;
            ds_int <= ds;

            if (as = '0' and as_int_d1 = '1' and addr_bus(23 downto 8) = MBASE) then
                address_match <= '1';
            else
                address_match <= '0';
            end if;

        end if;

    end process;


    -- Process:  SEQUENTIAL
    -- synchronisation avec la machine à états
    SEQUENTIAL: process (clk, reset)
    begin
        if reset = RESET_ACTIVE then
            prs_state <= IDLE;
            dtack_int <= '1';

        elsif clk'event and clk = '1' then
            prs_state <= next_state;
            dtack_int <= dtack_com;

        end if;

    end process;

    -- Process:   COMBINATIONAL
    -- liaison entre la machine à état et le bus parallel du uC
    COMBINATIONAL: process (prs_state, as, as_int_d1, ds_int, address_match)

    begin
```

```
    next_state <= prs_state;
    dtack_com <= '1';
    dtack_oe <= '0';

    case prs_state is

        ------------- état IDLE  -------------
        when IDLE =>

                -- attendre le front déscendant de as
            if as_int_d1 = '1' and as = '0' then
                -- front déscendant de as
                next_state <= ADDR;
            end if;


        ------------ état ADDR --------------
        when ADDR =>

            -- vérifier si le module est adressé
            if address_match = '1' then
                -- attendre l'assertion de ds, actif bas
                    if ds_int = '0' then
                    next_state <= DATA_TRS;
                else
                  next_state <= ADDR;
                end if;
            else
                -- si le module n'est pas adressé
                  next_state <= IDLE;
            end if;


        ---------- état DATA_TRS ------------
        when DATA_TRS =>
                next_state <= ASSERT_DTACK;
                dtack_oe <= '1';

        -------- état ASSERT_DTACK ----------
        when ASSERT_DTACK =>

                -- assertion d'un acquittement pour le uC
                dtack_com <= '0';
              dtack_oe <= '1';

            -- attendre le front montant de ad et ds
            if (as_int_d1 = '0') and (ds_int = '0') then
              next_state <= ASSERT_DTACK;
            elsif (as_int_d1 = '1') and (ds_int = '1') then
              next_state <= IDLE;
            end if;

             end case;

    end process;

    -- Process:  ADDR_DECODE
```

```
    ADDR_DECODE: process (reset, clk, prs_state)
    begin
            if reset = RESET_ACTIVE then
            addr_en <= '0';
            cntrl_en <= '0';
            stat_en <= '0';
            data_en <= '0';

        -- synchronisation avec le front montant de l'horloge
        elsif clk'event and (clk = '1') then
          if address_match = '1' then

            -- sélection du registre approprié
            case addr_bus(7 downto 0) is

              when MADR_ADDR =>  addr_en <= '1';
                        cntrl_en <= '0';
                        stat_en <= '0';
                        data_en <= '0';

              when MBCR_ADDR =>  cntrl_en <= '1';
                        addr_en <= '0';
                        stat_en <= '0';
                        data_en <= '0';

              when MBSR_ADDR =>  stat_en <= '1';
                        addr_en <= '0';
                        cntrl_en <= '0';
                        data_en <= '0';

              when MBDR_ADDR =>  data_en <= '1';
                        addr_en <= '0';
                        cntrl_en <= '0';
                        stat_en <= '0';

              when others => addr_en <= '0';
                       cntrl_en <= '0';
                       stat_en <= '0';
                       data_en <= '0';

            end case;
          else

            addr_en <= '0';
            cntrl_en <= '0';
            stat_en <= '0';
            data_en <= '0';

          end if;

        end if;

    end process;


    -- Process:  DATA_DIR
    -- lecture (écriture) du (dans) le registre approprié spécifié par l'adressage du uC
```

```
    DATA_DIR: process(clk, reset, msta_rst)
    begin

        -- initialisation de tout les registres internes après reset
        if reset = RESET_ACTIVE then

            -- registre d'adresse
            madr <= (others => '0');

            -- registre de contrôle
            men  <= '0';
            mien <= '0';
            msta <= '0';
            mtx  <= '0';
            txak <= '0';
            rsta <= '0';

            mbcr_wr <= '0';

            -- registre d'état
            mal_bit_reset  <= '0';
            mif_bit_reset  <= '0';

            -- registre de données
            mbdr_micro <= (others => '0');
            mbdr_read <= '0';

            -- initialisation du bus de données
            data_out <= (others => '0');

        -- vérification du front montant de l'horloge
        elsif clk'event and (clk = '1') then

          if (prs_state = IDLE) then

            mbcr_wr <= '0';
            mbdr_read <= '0';

          -- vérification de l'état du transfert de données
          elsif (prs_state = DATA_TRS) then

                -- registre d'adresses
            if addr_en = '1' then
              if r_w = '0' then

                -- accès du uC en écriture
                madr <= data_in(7 downto 1) & '0';

              else
                          -- accès du uC en lecture
                   data_out <= madr;

              end if;
            end if;

                -- registre de contrôle
                if cntrl_en = '1' then
```

```
              if r_w = '0' then
                -- accès du uC en écriture
                   mbcr_wr <= '1';
                   men  <= data_in(7);
                   mien <= data_in(6);
                   msta <= data_in(5);
                   mtx  <= data_in(4);
                   txak <= data_in(3);
                   rsta <= data_in(2);

              else
                -- accès du uC en lecture
                   mbcr_wr <= '0';
                 data_out <= men & mien & msta & mtx &
                       txak & rsta & "0" & "0";

              end if;
            else
                mbcr_wr <= '0';
            end if;

                -- registre d'état
                if stat_en = '1' then
              if r_w = '0' then

                   if data_in(4) = '0' then
                    mal_bit_reset <= '1';
                   end if;

                   if data_in(2) = '0' then
                    mif_bit_reset <= '1';
                   end if;
              else
                -- accès du uC en écriture
                data_out <= mcf & maas & mbb & mal &
                      "0" & srw & mif & rxak;
                   mal_bit_reset <= '0';
                   mif_bit_reset <= '0';

              end if;
            end if;

                -- registre de données
                if data_en = '1' then
                    if r_w = '0' then
                    -- accès du uC en écriture
                     mbdr_read <= '0';
                     mbdr_micro <= data_in;
                    else
                     -- accès du uC en lecture
                     mbdr_read <= '1';
                     data_out <= mbdr_i2c;
                    end if;
            else
                mbdr_read <= '0';
            end if;
```

```
          end if;

          -- en cas de perte d'arbitrage, le contrôleur I2C va générer un reset
          -- pour le bit msta pour le forcer en mode esclave

          if msta_rst = '1' then
            msta <= '0';
          end if;

          if rsta_rst = RESET_ACTIVE then
            rsta <= '0';
          end if;

        end if;

    end process;



end BEHAVIOUR;
```