REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

ECOLE NATIONALE POLYTECHNIQUE

Département de l'électronique

Mémoire de fin d'étude

CONCEPTION DE MICROPROCESSEUR

Encadré par :
Mr R.SADOUN

réalisé par :
M^elle SAMIA DJEGHLAF
Mr SAMIR DEDJEL

Promotion - 2002

Dédicaces

Je dédie ce modeste travail à ma chère maman et à mon cher papa.

Ames sœurs Nadjet et Sousou et à mes frères Khier et Hmimed.

A ma grande sœur Saida et son mari et à ses deux filles Yasmine et Amira.

A mes chers grands parents.

A mon seul oncle à qui je souhaite une rapide guérison.

A ma tante Rebeh et son mari Belkacem.

A toutes mes tantes.

A mes cousines (Aicha, Nounou, Rabia, Mouna, …) et à mes cousins.

A Karima et sa famille.

A ma tante Fatima et ses enfants.

A Malik et sa famille.

A mon binôme Samir.

A toutes mes amies (Nadia, Abla, Radia, …).

A tous mes amis :

De l'école fondamentale Abd Rahman Ahmine.
Du lycée technique Ibn El-Haitem.
De l'école nationale polytechnique.

A ceux que j'aime et ceux qui m'aiment.

Remerciement

Nous rendons grâce à **DIEU** qui nous a donné le courage pour mener à bien notre projet de fin d'étude qui est le résultat de notre parcourt

Nous tenons à remercier **Mr. R.Sadoun** pour nous avoir proposé ce projet de fin d'étude et pour son soutien et son aide précieuse tout au long de ce travail aussi pour les moyens qu'il a mis à notre disposition et de nous avoir recueilli dans son laboratoire à l'Ecole Nationale Polytechnique.

Nous remercions également **Mr. A.Zerguerass** de nous faire l'honneur de présider de jury et l'intérêt qui porte à notre sujet.

Nos plus vifs remerciement à **Mr. Aït Chiekh** qui a accepté d'examiner notre travail.

Nous tenons à remercier tous ceux qui ont contribué de loin comme de prés dans notre formation.

Résumé

Notre travail consiste à la conception de trois microprocesseurs de type CISC, que nous avons choisi dans un ordre de complexité croissant et leur implémentation sur un circuit FPGA.

Nous avons étudié les principales caractéristiques des microprocesseurs : modèle, structure et la relation avec le *soft*. Ensuit nous avons illustré la méthode de conception en commençant par le langage de description VHDL, l'outil de synthèse et le circuit cible FPGA.

Comme finalité, nous avons décris, synthétisé et simulé les trois modèle CISC : un modèle de base, un modèle à séquenceur micrprogrammée le troisième modèle a une architecture pipeline.

Mots clés : Microprocesseur, Structure, VHDL, FPGA, XILINX .

Summary

Our work consists with the design of three microprocessors of the type CISC, which we chose in an increasing order of complexity and their implementation on a circuit FPGA.

We studied the principal characteristics of the microprocessors: model, structure and the relationship to *the software* Follows we illustrated the method of design while starting with the language of description VHDL, the tool for synthesis and target circuit FPGA.

Finality, we have describe, synthesised and simulated the three modelled CISC: a basic model, a model with micrprogrammed sequencer the third model has an architecture pipeline.

Key words Microprocessor, Structure, VHDL, FPGA, XILINX.

ملخص

عملنا تمثل في تخطيط ثلاثة معالجات من النوع CISC خيروا في ترتيب تصاعدي للتعقيد. ثم إنجازهم على دارة FPGA .
درسنا الخصائص الأساسية للمعالجات : مثال العرض, البنية و العلاقة مع البرامج . ثم عرضنا طريقة الإنجاز, بدأنا بلغة الوصف VHDL, وسيلة الإنجاز ثم الدارة الهدف FPGA. الهينا بوصف, إنجاز ثم تشابه للأمثلة الثلاث : مثال أساسي, مثال ذا متحكم مبرمج, المثال الثالث ذا هندسة أنبوبية.

كلمات مفتاحيه : المعالج , البنية , XILINX. FPGA VHDL.

Sommaire

Conceptions et implémentations

Conclusion générale

Introduction générale

Le microprocesseur, inventé par Ted HOFF, a 34 ans. EN 1943, l'américain Shockley avait découvert l'effet transistor. Son travail sur le comportement des matériaux semi-conducteurs lui valut le prix Nobel. Exploitant ces propriétés, l'informatique trouvait son « interrupteur », le transistor.

C'est à la même époque, en 1946, que fut construit le premier calculateur électronique, l'Eniac. Il utilisait 18000 tubes électroniques et occupait une pièce entière.

Les transistors arrivèrent dans nos équipements électroniques. Certains se mirent en tête, puisque il en fallait plusieurs, de les installer sur le même morceau de silicium. C'est ainsi que Bob NOYCE et Jack KILBY inventèrent le circuit intégré. Ce qui a été le prélude à la naissance, en 1968, d'Intel.

A l'époque , tous les circuits logiques destinés à l'exécution de calculs et des programmes étaient développés sur mesure. Ted HOFF, chargé de projet à la société Intel et suite à une commande pour la réalisation de circuits pour des calculatrices avait décidé, au lieu de réaliser douze circuits en utilisant une logique dite câblé de créer quatre composants disposés atour d'un circuit logique polyvalent. Ce circuit prendrait ses ordres d'un circuit mémoire. La logique dite programmée était née. Ainsi pensait Ted HOFF, le circuit ainsi réalisé, (appelé un peu plus tard microprocesseur) pourrait trouver son emploi dans d'autres applications. Il suffirait de modifier les programmes mis en mémoire.

En 1971, le premier microprocesseur fit officiellement son apparition à travers le circuit 4004 d'Intel. Gros comme un ongle, il intégrait 2300 transistors et délivrait une puissance de calcul de 60 000 opérations par seconde identique à celle de l'Eniac. Ce qui semble, 34 ans plus tard, dérisoire par rapport au plus courant des microprocesseurs qui peuvent facilement atteindre les 300 milllions d'instructions par seconde avec une moyenne de 5 millions de transistors d'intégrés sur le même circuit. Le microprocesseur est aujourd'hui au cœur de la plus part des systèmes.

De ce bref aperçu, on peut implicitement déduire que l'évolution n'a pas été seulement technologique mais elle a été aussi conceptuelle. Du simple circuit monotâche, on assiste actuellement et sur le même circuit à l'association d'un parallélisme à grain de plus de plus fin pour atteindre des performances jamais inégalées. Dans cette course aux performances, la complexité toujours croissante des circuits a conduit à des degrés d'intégration très élevés.

La logique dite programmable est née du besoin simultané d'intégration et d'optimisation des coûts de développement. Elle permet en outre de réduire les temps de mise sur le marché des produits et de gérer à moindre coût les évolutions du composant. Elle apparut dans les années 80. Initialement la densité d'intégration était réduite. On obtenait un composant générique apportant la souplesse de conception qui faisait défaut dans les années 70.

Cette évolution a conduit à la naissance, dans le milieu des années 80, des FPGAs. Ces composants permettent un degré d'intégration très élevé réservé jusque là aux ASICs en reprenant en partie leur méthodologie de développement.

De l'utilisation de densité toujours plus grande en matière de semi conducteur sont apparus de nouveaux modes et langages de description comme VHDL. Ce dernier est devenu incontournable dans le développement d'applications à base de FPGAs.

Microprocesseur, FPGA et VHDL forment donc les éléments clefs du travail qui nous a été confié. Il s'inscrit dans le cadre de la conception des microprocesseurs et leur réalisation sur des circuits programmables.

Notre approche de conception était ascendante motivée essentiellement par l'implémentation de trois approches architecturales différentes mais pouvant être étroitement complémentaires. Le but recherché était essentiellement pédagogique.

Nous avons structuré notre approche et par là notre mémoire suivant trois axes. Le premier nous a introduit les concepts fondamentaux aussi bien théoriques que pratiques liés aux architectures des ordinateurs et particulièrement des microprocesseurs. Le second nous introduit quant à lui aux techniques de conception et les technologies cibles utilisées. L'exploitation des ces deux axes a abouti à l'implémentation sur une cible FPGA de trois types différents de modèles de microprocesseurs.

Introduction

Le microprocesseur est avant tout un composant électronique auquel les progrès technologiques ont permis d'atteindre un degré de complexité tel qu'il est devenu un système programmable d'usage général. Physiquement, il se présente sous forme d'une puce de silicium montée dans un boîtier dont le nombre de broches exprime son degré de complexité.

A l'électronique, il emprunte les concepts de base d'abord au niveau des propriétés de la matière, ensuite au niveau des composants discrets et enfin au niveau des circuits logiques. Mais contrairement aux composants qui l'ont précédé, la fonction d'un microprocesseur n'est pas définie à priori mais peut être fixée à l'aide d'un programme enregistré. On peut parler dans ce cas d'un autre type de composant générique. Cela lui confère un très haut degré de généralité et lui permet d'être utilisé dans toute une série de systèmes dont les finalités peuvent être différentes.

Pour pouvoir fonctionner ainsi, les microprocesseurs empruntent à l'informatique son organisation : ce sont en général des machines de Von Newman. Ils réunissent sur un seul et même boîtier l'ensemble des éléments de cette machine. Ils comportent un chemin de données sur lequel circulent, sont traitées et sont mémorisées les informations sous forme binaire.

On peut encore dire que les microprocesseurs sont des composants électroniques mettant à profit le remplacement de la logique câblée par celle programmée. Ces composants peuvent avoir plusieurs fonctions. C'est par programmation que l'on détermine cette fonction.

On sait cependant que l'exécution d'un programme est en général plus lente que sa réalisation sous forme câblée. Les microprocesseurs sont donc souvent plus lents que des composants spécialisés.

Pour appréhender correctement la compréhension et l'exploitation des concepts associés aux microprocesseurs, il est nécessaire d'aborder leurs modélisations. La matérialisation de ces modèles (figure ci-dessous) n'est pas indépendante de la technologie (Babagge, Von Neumann, Wilkes...) .

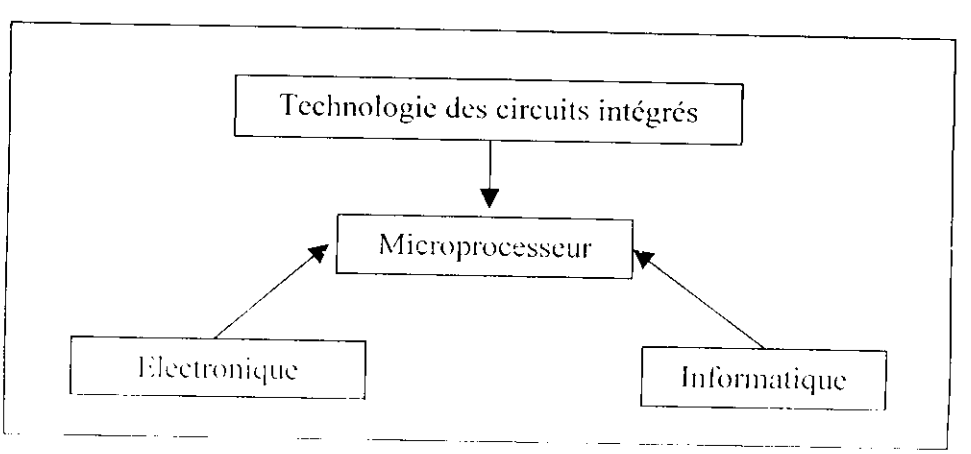


Figure I-1 :le microprocesseur est l'intersection des trois domaines

I- Modélisation des Microprocesseurs

I-1- Modèle en couches

Le premier modèle en couches est historiquement celui de l'ordinateur. Il s'est ensuite étendu aux télécommunications. On peut généraliser ce type de modèle à d'autres domaines de représentation.

Le processeur d'un ordinateur fonctionne très vite, mais ne peut reconnaître et exécuter qu'un nombre limité d'instructions comme par exemple :

- changer l'emplacement d'une donnée en mémoire,
- additionner deux nombres,
- vérifier si un nombre est égal ou non à zéro, etc.

L'ensemble de ces instructions constitue le *langage machine* L_1. Ses instructions s'écrivent sous forme de 0 et de 1 et il très pénible de l'utiliser pour programmer. On écrit donc en L_1 un ensemble d'instructions plus riches, plus proches du langage naturel et on construit à partir de ces instructions un nouveau langage L_2 plus facile à utiliser.

Il existe deux façons d'exécuter un programme écrit en L_2 :

- Un programme en L_1 dit traducteur remplace chaque instruction du programme en L_2 par la suite équivalente d'instructions en L_1. On parle dans ce cas de compilation. Le microprocesseur exécute ensuite le programme en L_1 ainsi obtenu.
- On utilise un programme en L_1 capable, après avoir lu une instruction en L_2, d'exécuter immédiatement la séquence en L_1 équivalente. On parle dans ce cas d'interpréteur. Ce dernier évite de générer un programme en L1. elle a l'inconvénient d'être plus lente que l'exécution d'un programme compilé en L_1.

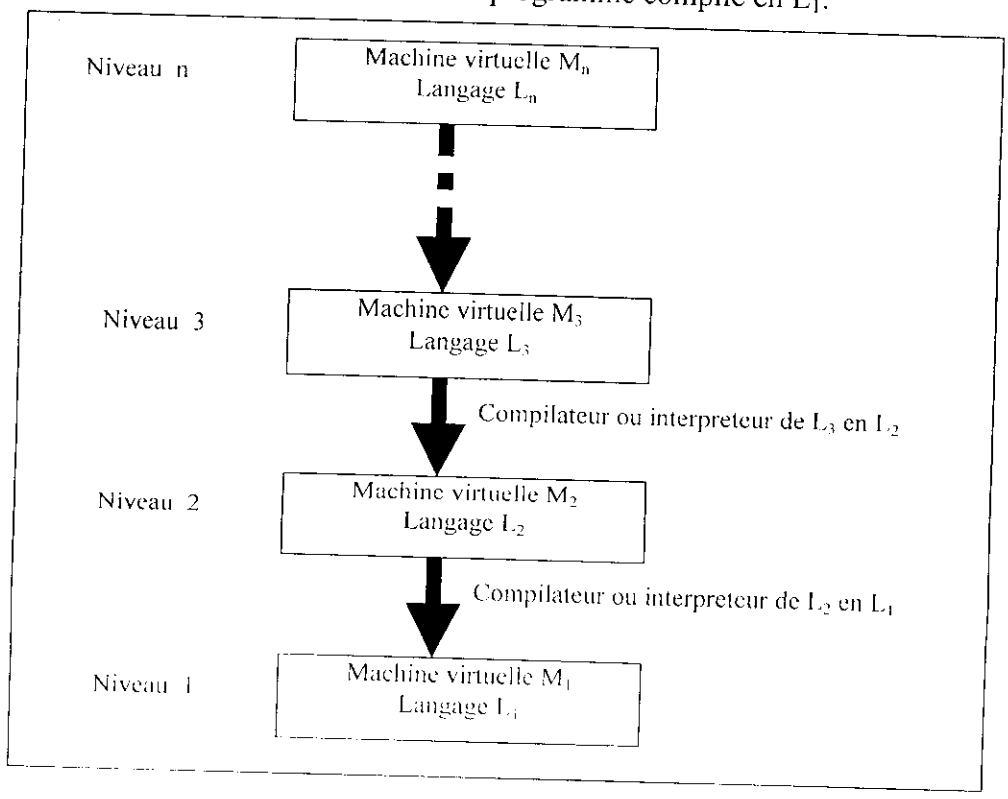


Figure I-2: Le modèle en couche

Du point de vue de l'utilisateur, le microprocesseur est doté d'un traducteur obéissant aux instructions écrites en L_2 aussi « docilement » que si elles étaient écrites en L_1. L'ensemble constitué de la machine physique M_1, des langages L_1 et L_2 et du traducteur, constitue la "machine virtuelle" M_2. Lorsque l'on écrit un programme, on peut considérer que M_2 est aussi "réelle " que M_1.

Il existe des limites à la complexité acceptable pour un traducteur. De ce fait L_2, bien que plus commode que L_1, peut ne pas être le langage le plus convenable pour programmer. On écrit en L_2 un ensemble d'instructions, et on construit un nouveau langage plus commode, L_3, qui définit la nouvelle machine virtuelle M_3.
Langages et machines virtuelles s'empilent ainsi jusqu'à la couche n (figure I-2).

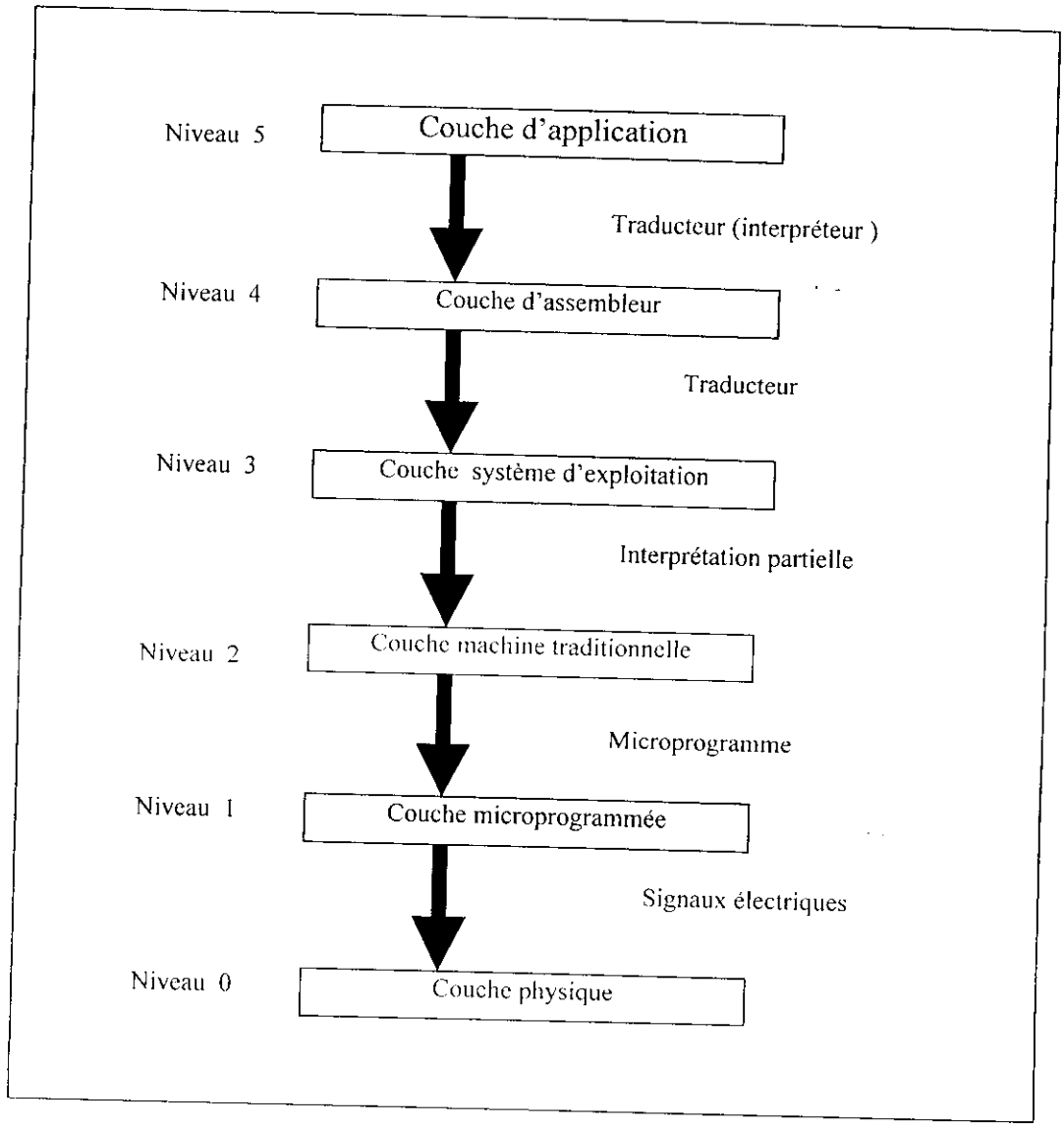


Figure I-3: Modèle en couche d'un ordinateur

La plus part des ordinateurs actuels possèdent six couches (figure I-3); un utilisateur (par exemple en se servant d'un traitement de texte) utilise un programme dont les instructions arrivent à la couche physique de l'ordinateur (processeur et mémoires) à travers une cascade de traductions. Mais l'utilisateur peut ignorer ces opérations : pour lui et dans le cadre de cette

application, l'ordinateur fonctionne comme une machine de traitement de texte, et il n'a à connaître que cette machine virtuelle.

Une instruction du niveau applicatif, traduite en cascade dans les langages des couches inférieures, engendre de nombreuses instructions dans la couche microprogrammée. Si l'on cherche à optimiser les délais de traitement, il faut donc programmer dans les "couches les plus basses ". Cependant l'accroissement des performances des processeurs et de la taille des mémoires limite l'utilité d'une telle optimisation, sauf pour des applications où la rapidité est critique (applications temps réel). En général on se soucie peu désormais d'optimiser l'utilisation de la ressource physique. Les programmes sont presque tous écrits dans des langages de niveau élevé.

Ce qui précède illustre deux des principes des modèles en couches :

- l'utilisateur n'a à se soucier que du service rendu par la couche dont il se sert, et qu'il considère comme une ressource physique ("matériel et logiciel sont équivalents ") ;
- l'optimisation de l'utilisation des ressources se fait couche par couche non en considérant le processus d'ensemble.

Le microprocesseur constitue les trois premières couches de l'ordinateur : physique, microprogrammée et machine traditionnelle.

Les ordinateurs des années 40 et 50 n'avaient que deux couches : le niveau machine traditionnel dans lequel on programmait et le niveau physique qui exécutait les programmes. Les circuits de ce dernier niveau étaient complexes, difficiles à construire et peu fiables.

Maurice Wilkes eut en 1951 l'idée de concevoir un ordinateur à trois couches pour simplifier le matériel. La machine disposait d'un traducteur exécutant les programmes du niveau machine traditionnelle. Le matériel ne devait plus alors exécuter que des microprogrammes avec un répertoire d'instructions limité et non des programmes en langage machine.[1]

I-2-Modèle de Von Newman

Le premier modèle de l'ordinateur et par là, du microprocesseur, est celui proposé en 1832 par le mathématicien anglais Babagge. Dans ce modèle, l'exécution de l'instruction se fait en trois cycles :

- chargement d'une instruction de la carte perforée vers l'intérieur du calculateur d'une façon séquentielle et linéaire
- le décodage de l'instruction
- l'exécution.

Pour accomplir ces trois tâches, ce modèle contient un registre de sauvegarde de l'instruction appelé registre d'instruction, une unité arithmétique pour les opérations arithmétiques et un accumulateur pour la sauvegarde des résultats. On parle alors d'ordinateurs à programme externe. La mémoire interne a été réservée exclusivement au données. Ce qui nous rappelle étrangement le modèle de HARVARD (figure)
En 1946, c'est-à-dire un peu plus d'un siècle plus tard, Von Newman enrichit le modèle de babagge en introduisant un concept majeur : la non linéarité dans l'exécution du programme en introduisant un principe naturel en programmation qui n'est autre que la rupture de séquence. Pour cela, il eu besoin d'ajouter un compteur programme (PC), appelé aussi compteur ordinal (CO), qui sert de pointeur vers l'instruction à charger à partir d'une mémoire interne. Cette fois ci, on parle alors d'ordinateurs à programme interne.
Le modèle de Van Newman est à la base des modèles des microprocesseurs modernes.

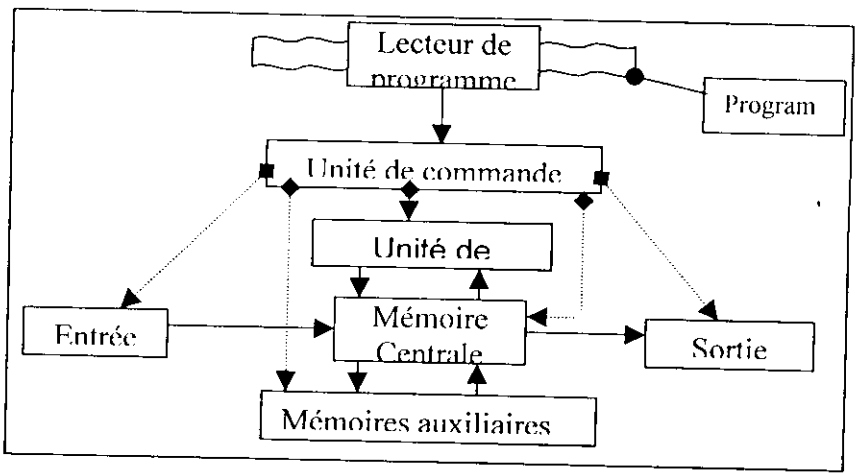


Figure I-4 : Modèle de la machine de BABAGGE

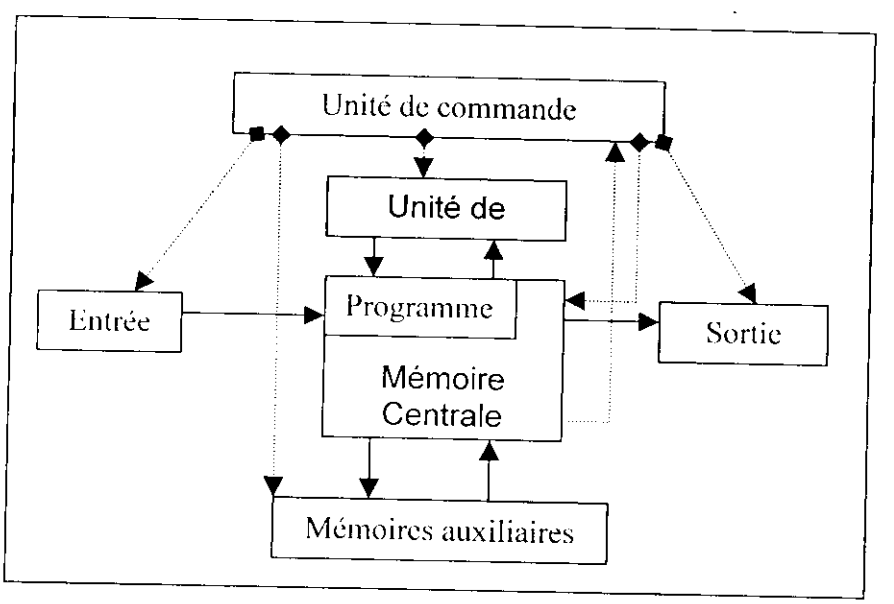


Figure I-5 : Modèle de la machine de VON NEUMANN

I-3- Modèle de programmation

Le modèle de programmation est le microprocesseur vu par le programmeur à partir de la couche langage machine. A travers les instructions de ce langage, on voit que le microprocesseur est un ensemble d'entités de sauvegarde dites registres (figure I-6). Un registre a trois caractéristiques : le type, le format et les manipulations possibles.

- Le type : données, adresse, pointeur et état .
- Le format: c'est sa longueur en bits et sa composition.
- Les manipulations : lecture et (ou) écriture

Dans les microprocesseurs connus sur le marché, on trouve les différents types de registres suivants :

- Les registres à usage général. Ils servent à sauvegarder les résultats d'un traitement.
- Les registres d'index. Ces registres sont utilisés comme pointeur sur une case mémoire.
- Les registres de segmentation et de pagination. Ils servent dons la génération d'adresses à partir d'un découpage logique de l'espace d'adresse.
- Les registres systèmes. Ne sont accessibles qu'en mode superviseur. Permettent de mettre en œuvre les mécanismes de protection interprocessus.

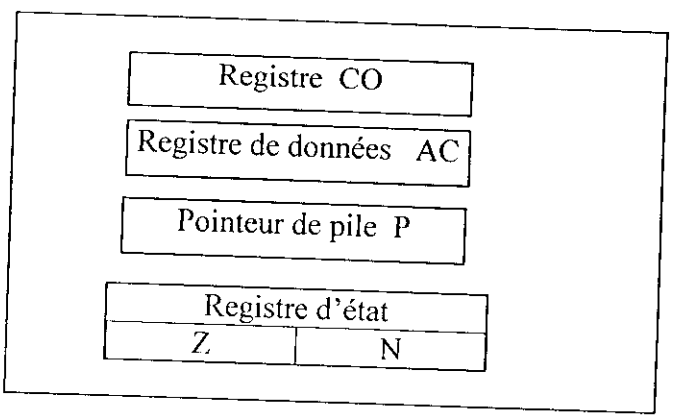


Figure I-6 : Exemple de modèle de programmation

I-4-Le modèle structurel

Un autre modèle de représentation d'un microprocesseur est le modèle structurel. Il est constitué d'un chemin des données et d'une partie commande qui gère les fonctions de microprocesseur. Cette dernière est réalisée par un séquenceur. On distingue y distingue deux types fondamentaux de séquenceurs ceux à logique câblée et ceux séquenceurs à logique programmée.

I-4-1 Organisation et composantes du chemin des données

➢ **Définition du chemin des donnés**

On appelle chemin de données l'ensemble des organes permettant de transférer, mémoriser ou traiter les informations (instruction, adresses, opérandes) issues de la mémoire centrale. Les composants de chemin de données répondent à l'une de ces trois fonctions. Il est composé de bus d'interconnexion pour les transferts d'information, de registres et de mémoires pour leur stockage ainsi que d'unités fonctionnelles pour leur traitements. Tout au long du chemin de données sont distribués des signaux de commande permettant l'exécution des différentes opérations de transfert, de stockage et de traitement de l'informations. On les appelle microcommandes. Elles sont issues d'un organe central généralement appelé séquenceur.

Notre étude du chemin de données dans la machine de type Von Neumann va porter sur trois points :

1-Les registres et leurs interconnexions.
2-Le cheminement des opérandes et l'utilisation des unités fonctionnelles.
3-Le cheminement des adresses et les techniques d'adressage de la mémoire centrale.

➢ Les registres du chemin de données

Sur le plan fonctionnel, on distinguera deux types de registre :

- Les **registres non adressables** caractérisés par une fonction bien définie qu'ils sont seuls à remplir. Par exemple : le registre d'instruction, l'accumulateur et le multiplicateur quotient, le compteur ordinal, le registre de sélection mémoire, le compteur de décalage.
Certains de ces registres peuvent rester invisibles au programmeur. On dira qu'ils sont transparents. D'autre sont implicitement adressés par le programmeur.

- Les **registres adressables** que le programmeur peut utiliser pour conserver certaines informations au cours d'un traitement. La baisse de coût des registres en technologie intégrée et le gain de temps réalisé en utilisant des informations en mémoire centrale ont conduit les constructeurs à multiplier le nombre de registres adressables.

On en distingue deux catégories : des registres arithmétiques permettant de conserver des résultats partiels et des registres d'adressage permettant de conserver des éléments fixes d'adressage de parties de programmes ou de tableaux. Cependant, dans certaines machines, les registres adressables sont indifféremment utilisés comme registre d'adressage. On dit qu'ils sont banalisés.
Sur le plan structurel, on distingue deux organisations possibles de registres adressables :

a- Ils peuvent être montés comme registres indépendants. Dans ce cas, l'adresse du registre doit être décodée pour obtenir les signaux d'ouverture ou de fermeture des portes des registres.

b- Ils peuvent également être montés en mémoire. L'adresse de registre sera alors fournie à la mémoire sans décodage préalable. De telles mémoires sont généralement appelées mémoires locales.

➢ Le cheminement des opérandes

Issus ou retournant vers la mémoire par l'intermédiaire d'un registre, les opérandes sont traités dans des unités fonctionnelles ; les résultats partiels pouvant être stockés dans des registres arithmétiques.

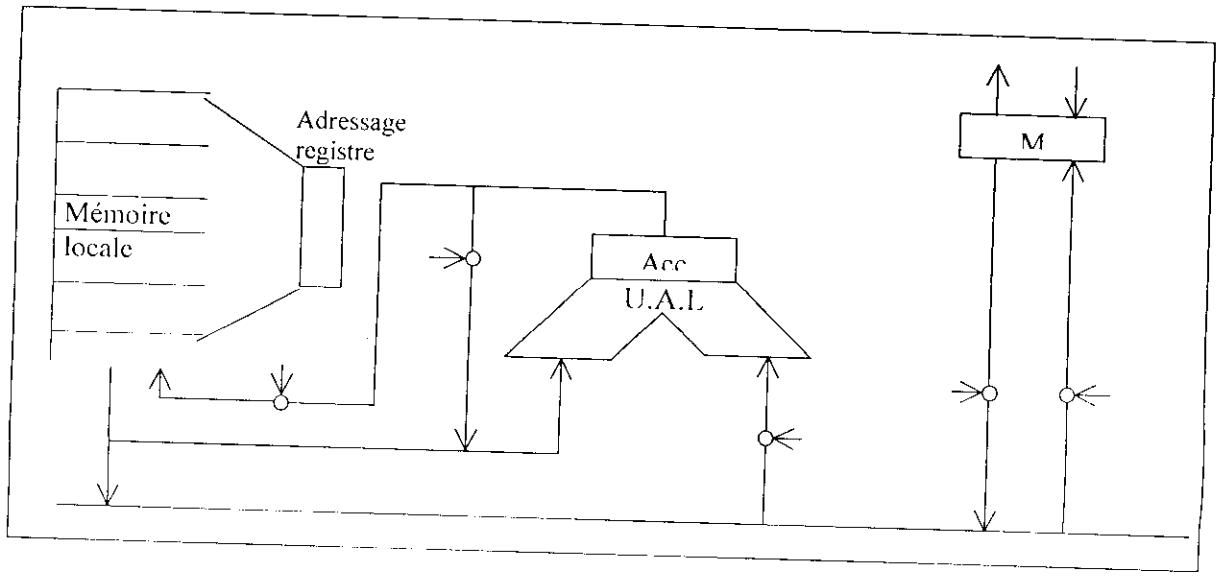


Figure I-7: Chemin de données avec registres arithmétiques organisés en mémoire locale

La figure I-7 donne un exemple de l'utilisation de registres arithmétiques montés en mémoire locale et comportant une seule unité fonctionnelle. Les opérations s'effectuent entre

le registre M et l'un des registres arithmétiques de la mémoire locale ou avec l'accumulateur. Le résultat obtenu dans l'accumulateur peut être rangé en mémoire centrale. L'accumulateur serait inutile si les deux hypothèses suivantes sont vérifiées :

a- Toute opération arithmétique porte sur un opérande en mémoire locale et un opérande en mémoire centrale, le résultat venant remplacée l'opérande en mémoire centrale.

b- La mémoire locale est câblée de telle sorte que le mot sélectionné peut être utilisé en lecture jusqu'à ce qu'on y force une nouvelle information.

➢ Le cheminement des adresses

Il s'agit uniquement du traitement réservé aux adresses contenues dans les instructions portants sur des informations en mémoire centrale.
Nous allons passer en revue les différentes techniques d'adressage.

Celles ci correspondent généralement à une transformation entre le contenu de la partie adresse de l'instruction et de l'adresse que l'on enverra finalement dans le registre de sélection mémoire pour obtenir l'information désirée. Cette dernière adresse sera appelée adresse effective. Le type de traitement que doit subir le contenu de la zone adresse est spécifié, soit par le code opération quand celui ci en impose un type déterminé, soit par la configuration binaire d'une partie de l'instruction qui contient ce que nous venons d'appeler les conditions d'adressage.

Ainsi l'instruction dans un accumulateur à une adresse se composera, compte tenu d'éventuelles adresses de registres arithmétiques, de trois parties :

CO	CA	AD
Code opération	Condition d'adressage	Zone adresse en mémoire

- Adressage direct ou absolu :

C'est l'adressage que l'on connaît bien : la partie adresse de l'instruction représente l'adresse effective de l'opérande (figure I-8). L'adressage normal nécessite 1 cycle mémoire pour obtenir l'opérande.

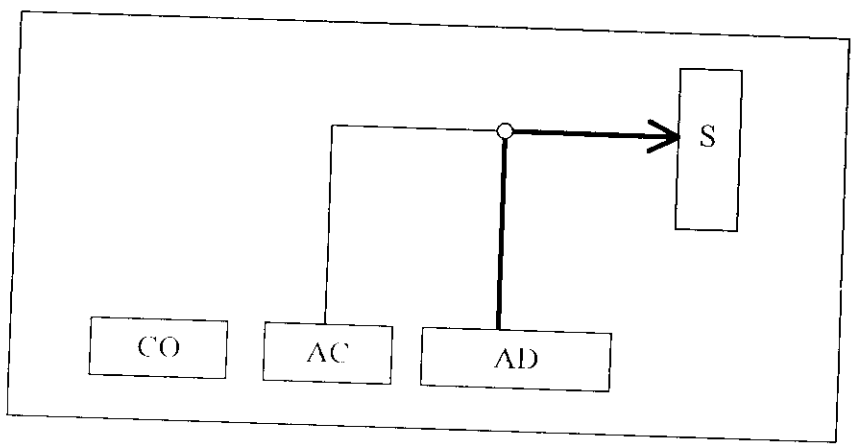


Figure I-8 Adressage direct.

- Adressage immédiat

Il ne s'agit pas d'un adressage proprement parler. La partie adresse de l'instruction contient en effet directement l'opérande sur le quel on veut travailler (figure I-9).

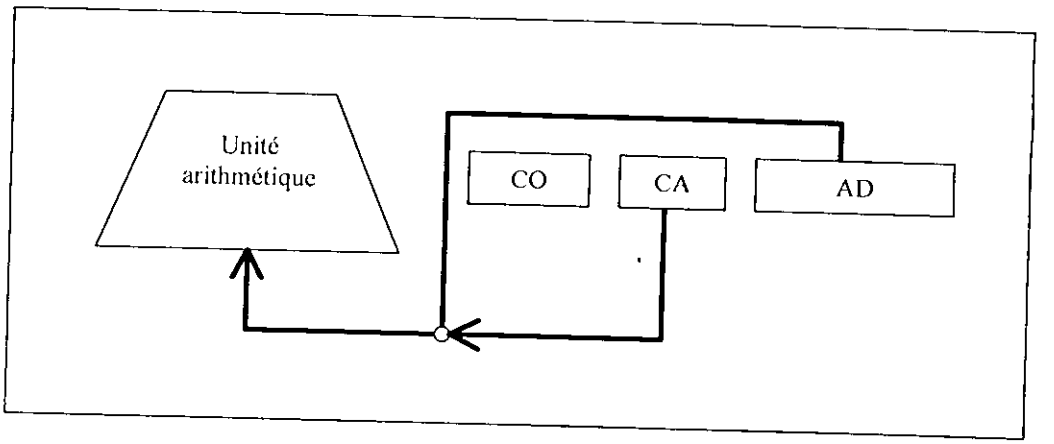


Figure I-9 : Adressage immédiat

- Adressage indirect

La partie adresse de l'instruction contient non pas l'adresse de l'information demandée, mais l'adresse d'un mot mémoire ou l'on trouvera l'adresse effective de l'information. La recherche d'un opérande en adressage indirect nécessitera donc deux cycles d'horloges mémoire : un cycle pour aller chercher l'adresse effective, un cycle pour aller chercher l'opérande (figure I-10).

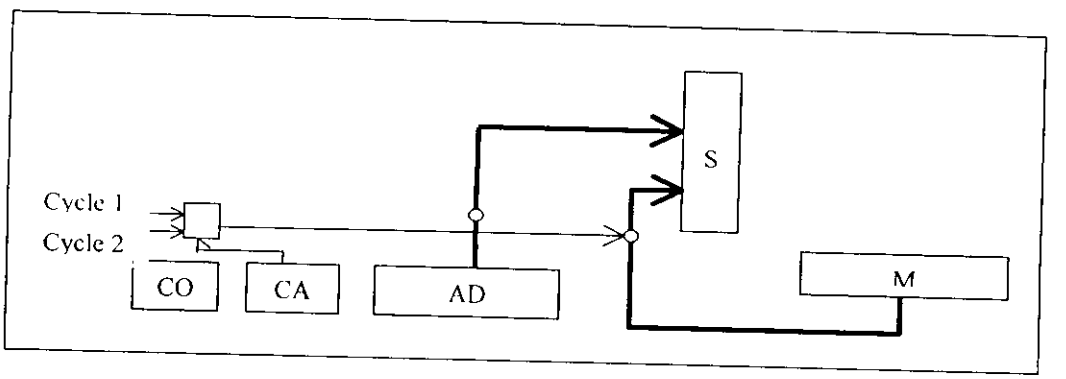


Figure I-10: Adressage indirect

Certaines machines permettent l'adressage indirect en cascade. Dans ce cas, le bit 1 spécifiant l'adressage indirect dans l'instruction se retrouve également dans le mot contenant les adresses indirectes successives. Il est clair qu'il faudra ajouter un temps de base de l'instruction, un cycle par niveau d'indirection.

- Adressage relatif

L'adresse relative n'indique pas en valeur absolue l'emplacement de l'information en mémoire, mais le situe par rapport à une adresse de référence. Cette adresse de référence est normalement contenue dans registre souvent appelé registre de translation. L'adresse effective sera obtenue en additionnant l'adresse relative à l'adresse de référence.

On emploie notamment les techniques d'adressage relatif lorsque la longueur du mot mémoire est insuffisante pour permettre d'adresser toute la mémoire. C'est ainsi que dans un calculateur dont le mot est de 16 bits, si on prend 8 bits pour le code opération et les conditions d'adressage(ce qui paraît raisonnable), il ne reste que 8 bits purs l'adresse. Ceci permet d'adresser 2^8 256 positions alors que ces calculateurs en possèdent généralement 32000. si on ne veut pas passer à un format d'instruction sur de mots, on peut utiliser l'adressage relatif pour atteindre directement certaines zones de la mémoire.

- Adressage indexé

On obtient l'adressage effectif en ajoutant à la partie de l'adresse de l'instruction le contenu d'un registre de l'unité centrale appelé **registre index** ; ce contenu est souvent appelé index. Le registre d'index est utilisé par le programmeur pour traiter par une même instruction placée dans une boucle de programme des données rangées sous forme de tableau dans des cellules successives de la mémoire(figure I-11).

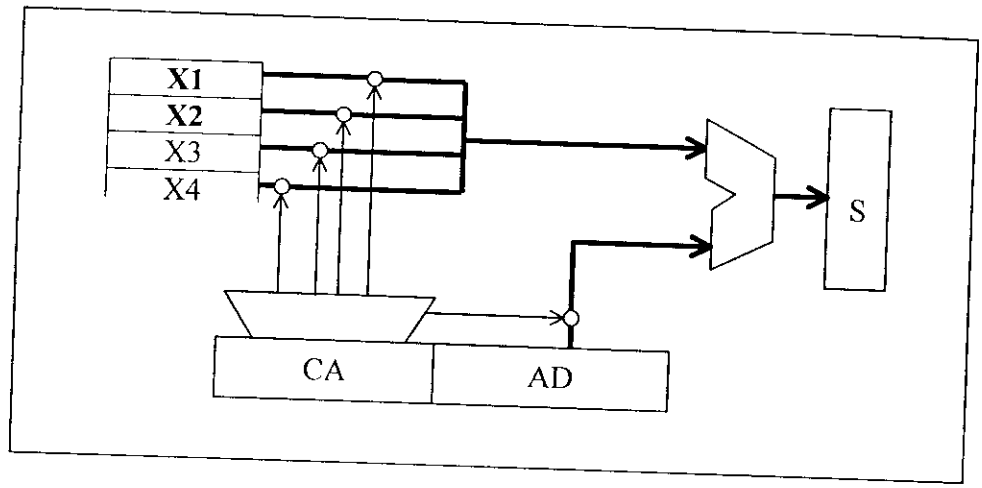


Figure I-11: L'indexation

I-4-2 L'unité de commande

➢ **Notion de séquenceur central**

Le séquenceur central d'un microprocesseur est l'organe qui génère les microcommandes distribuées le long du chemin de données pour le positionner et en commander les différents éléments.

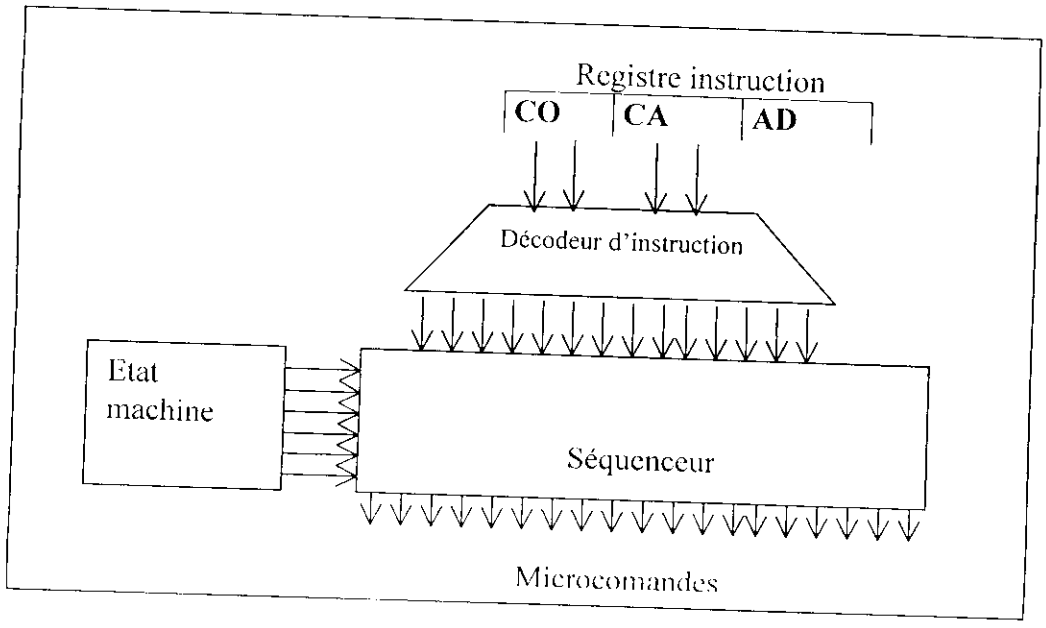


Figure I-12: environnement d'un séquenceur

Le séquenceur est d'abord défini de façon externe, c'est à dire par ces informations d'entrées et de sorties (figure I-12).

Les informations de sortie du séquenceur ne sont autres que les microcommandes qui doivent être distribuées le long de chemin de données selon des chronogrammes précis dépendant des temps de réponse des organes commandés.

Les informations d'entrée du séquenceur sont fournies d'une part par l'instruction. Ce sont le code opération, les conditions d'adressage et les adresses de registres. Ils sont fournis d'autre part par l'état machine qui regroupe un certain nombre d'informations telles que l'état des boutons de la console opérateur, les indicateurs d'erreurs, les demandes d'interruption, l'indicateur d'état de la mémoire et des unités fonctionnelles, le code condition de l'unité arithmétique et logique, etc.

➢ Séquenceurs câblés et séquenceurs micro programmés

On distingue deux grandes classe de séquenceurs : les séquenceurs câblés qui sont réalisés sous forme d'un circuit électronique séquentiel et les séquenceurs micro programmés qui correspondent à l'introduction dans l'unité de contrôle d'une mémoire contenant un petit programme appelé microprogramme. Le déroulement de ce microprogramme génère les microcommandes qui concrétisent l'exécution de l'instruction. On peut, en toute première approximation, assimiler le séquenceur micro programmé à un très petit calculateur (« virtuel »).

◆ le séquenceur à logique câblé

Il se compose de trois composantes essentielles à savoir un décodeur d'instruction, un distributeur de phases et un registre d'états du séquenceur.

A partir du code opération, le décodeur d'instruction donne l'ensemble des microcommandes (mot de commande) correspondants (figure I-13). Mais l'instruction machine corresponde a une succession de mot de commande. On doit alors indiquer au décodeur à quel état il est. Ceci est fait par des *registres d'état du séquenceur*. Ils sont chargés par le séquenceur à la fin de chaque état.

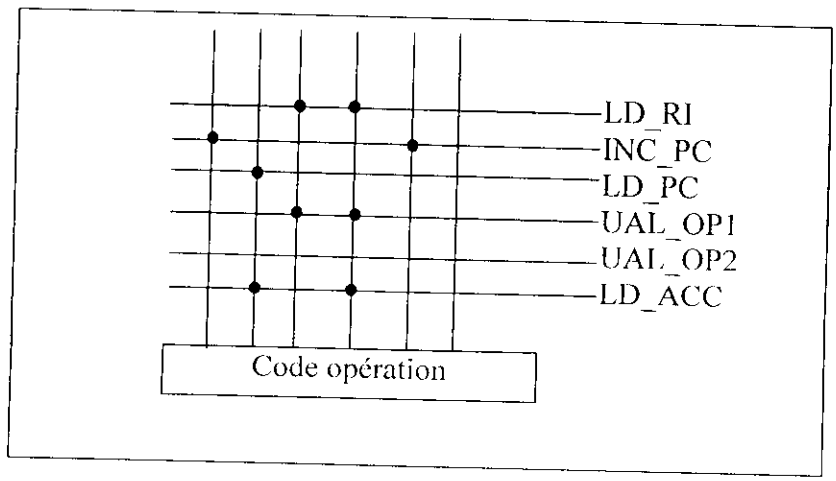


Figure I-13: exemple de décodeur d'instruction

On a dit qu'à chaque état correspond un mot de commande. Les éléments du chemin de données ne sont pas activés en même temps. Alors on a besoin de déphaser leurs activations dans le temps, ici intervient *le distributeur de phases*. Il a comme entrée le signal de l'horloge externe. A à la sortie, il génère des signaux périodiques de période multiple de celle de l'horloge et régulièrement déphasés appelées *phases* (figure I-14). Le nombre de

phases dépend du chemin de données, et leurs natures dépendent de la manière de l'activation des éléments de ce dernier : Front montant ou descendant. Dans ce cas, ils agissent directement sur le chemin de données. Le séquenceur est détaillé sur la figure I-15.

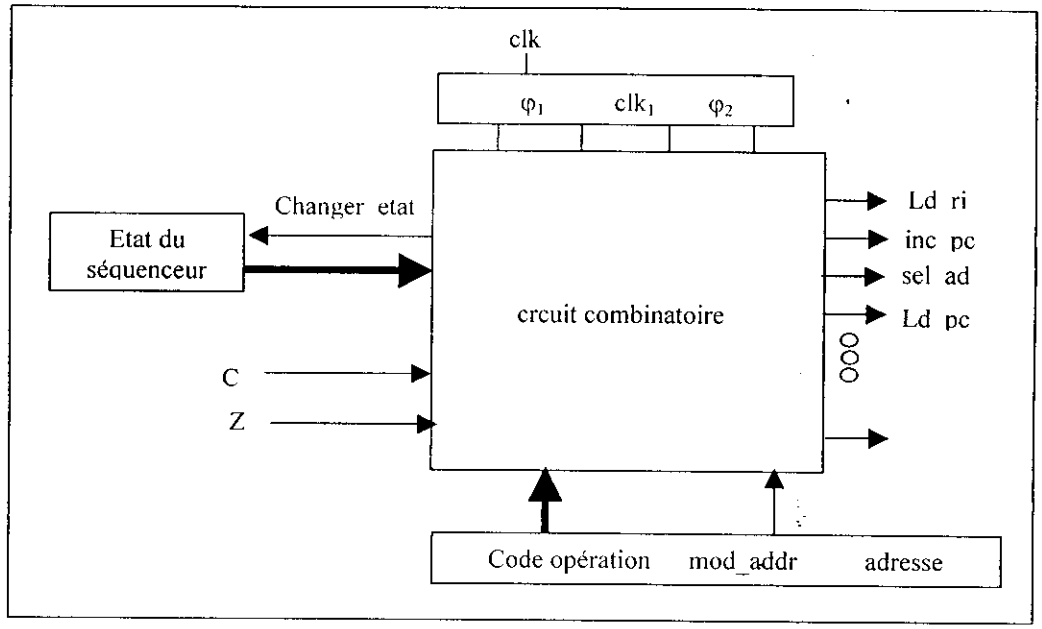


Figure I-15: le séquenceur

Pour illustrer prenons l'exemple suivant, la figure I-16 présente le chemin de données.

L'état du séquenceur est une machine à trois état FETCH,EXE1 et EXE2 . Elle bascule d'état en état par un signal d'activation sortant du circuit combinatoire.

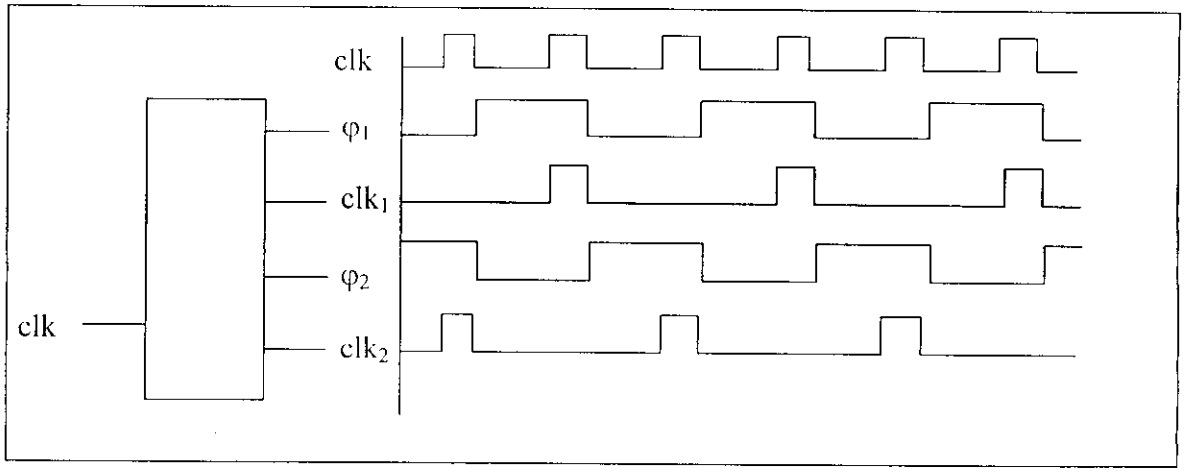


Figure I-14: Distributeur de phases

Il reste à déterminer les équations logiques du circuit combinatoire.

Ld_ri = fetch ϕ_1

Inc_pc = fetch ϕ_1

Rd = dir exe1 + fetch ϕ_1 + not dir not fetch

Ld acc = dir exe1 ϕ_2 + not dir exe1 ϕ_2

Etat suivant = fetch(dir exe1 + ind exe2) + exe1 (fetch dir)+ exe2 (exe1not dir)

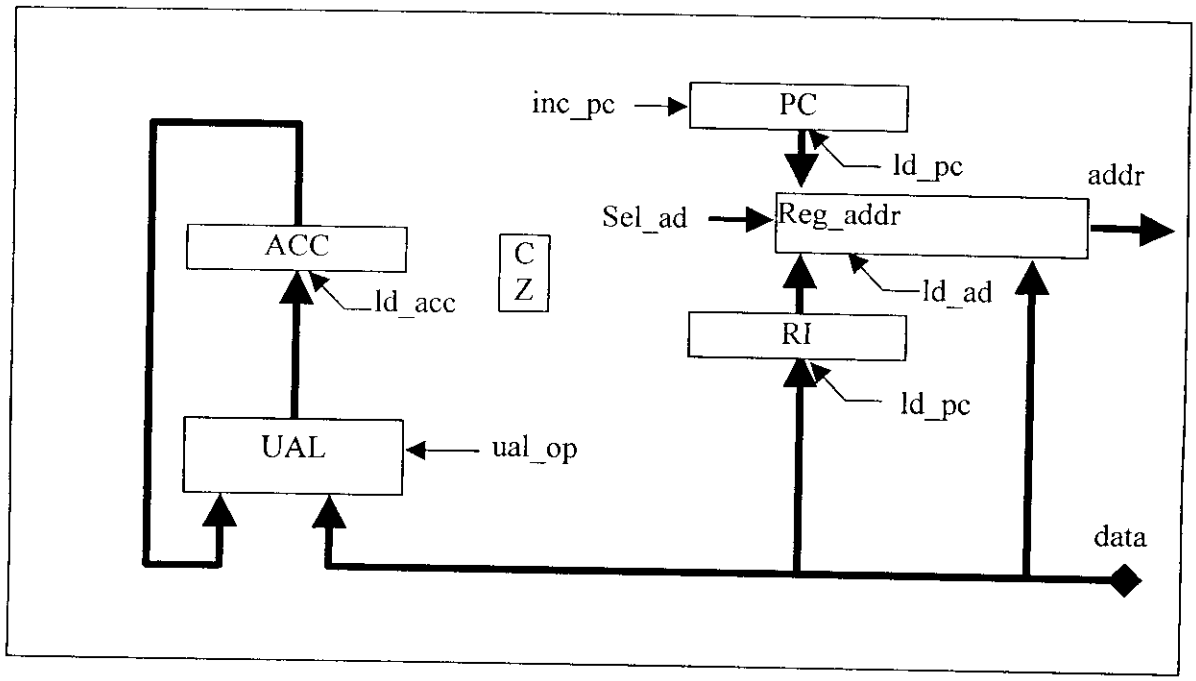


Figure I-16: chemin de données

- **La microprogrammation :**

L'exécution d'une instruction correspondait à un enchaînement de microcommandes chargées de positionner le chemin de données et de commander les unités fonctionnelles et les mémoires.

Aux paragraphes précédents, l'élément chargé de générer ces microcommandes était un séquenceur câblé. La microprogrammation consiste à remplacer le séquenceur câblé par un séquenceur programmé. A chaque instruction de microprocesseur correspond généralement dans une mémoire spécialisée appelée mémoire de contrôle ou encore mémoire de commande, un microprogramme dont le déroulement génère les microcommandes commandant l'exécution de l'instruction. Les instructions de microprogramme sont appelées micro-instructions. L'exécution d'une micro instruction génère une ou plusieurs micro commandes.

Les microprocesseurs micro programmés sont également appelés microprocesseur à logique enregistrée, ou microprocesseur à logique programmée ou encore microprocesseur à code instruction variable. Ce dernier terme correspond à la possibilité de modifier facilement le jeu d'instruction d'un microprocesseur micro programmé; le changement ou l'adjonction d'un microprogramme étant beaucoup plus simple que des modifications de câblage. La microprogrammation introduit ainsi un nouveau domaine moins « hard» que le hardware et moins « soft»que le software.

- **Structure de l'unité de contrôle d'une machine micro programmée :**

La base de la micro programmation a été définie par Wilkes en 1951. Tout les élément constitutifs du contrôle d'une machine micro programmable se retrouvent dans son modèle. Nous nous proposons d'étudier l'évolution des différents éléments du modèle de Wilkes.

1. La mémoire de contrôle

C'est la mémoire qui contient les microprogrammes, chaque mot mémoire comprenant une ou exceptionnellement plusieurs micro instructions. La première caractéristique généralement exigée d'une mémoire de contrôle est sa performance en lecture par rapport à celle de la mémoire centrale. Il faut en effet pour que l'ensemble soit cohérent, que tout un microprogramme enregistré en mémoire de contrôle se déroule pendant les quelques accès à la mémoire centrale concernant l'instruction et ses opérandes. Un deuxième élément qui intervient dans le choix de la technologie des mémoires de contrôle est que lors de déroulement des microprogrammes, elles ne sont utilisées qu'en lecture. C'est deux considérations ont conduit Wilkes et ses successeurs d'utiliser des mémoires mortes comme des mémoires de contrôle qui permettent une grande rapidité de lecture pour un coût raisonnable.

2. Le codage des micro instructions

Dans le modèle de Wilkes, la micro instruction est divisée en deux parties ; l'une permet de générer les microcommandes tandis que l'autre définit l'adresse de la micro instruction suivante. Nous allons dans ce paragraphe nous intéresser uniquement à la première partie de la micro instruction.
Dans la pratique, on trouve deux types de codage :

- **le codage type instruction :** comme son nom l'indique, ce type de codage donne à la micro instruction une structure semblable à celle d'une instruction, avec code opération et adresse d'opérande.
- **le codage par champs :** ce type de codage est beaucoup plus prés du fonctionnement réel du calculateur. Les différentes microcommandes sont divisées en groupes indépendants, de telle sorte qu'on ne puisse pas avoir de microcommandes simultanées. Pour la commodité du programmeur, chaque groupe correspond de plus à un certain type de fonction. On associe un champ de la micro instruction à chaque groupe ainsi défini, la grandeur du champ étant juste suffisante pour coder toutes les microcommandes du groupe. Le codage de la micro instruction s'exécute champ par champ.

3. L'adressage des micro instructions

Nous allons maintenant nous intéresser à la deuxième partie de micro instruction aux sens de Wilkes, celle qui définit les informations permettant d'enchaîner les micro instructions. Deux grands types sont possibles :

- L'adressage séquentiel

Il consiste à faire +1 dans le registre de sélection de mémoire de contrôle. Cette méthode présente le double avantage d'être simple à programmer et de ne pas allonger la micro instruction. Par contre un branchement occupe une micro instruction à lui tout seul et engendre une perte de temps d'exécution. On le trouve généralement associé à un codage type instruction.

- L'adressage explicite

C'est celui qui fut préconisé par Wilkes et que l'on trouve souvent associé aux techniques de codage par champs. La méthode de Wilkes pour réaliser le branchement est abandonné compte tenu du câblage particulier de la mémoire de contrôle qu'elle impose. La solution généralement adoptée est de diviser la partie de micro instruction qui définit l'instruction suivante en deux champs. Le premier contient explicitement les poids forts de l'adresse, le deuxième contient soit directement les poids faibles de l'adresse si cette dernière est connue, soit l'adresse d'un élément de registre qui les contient si elle est conditionnelle.[2]

I-5- L'architecture Harvard

Dans ce type d'architecture la mémoire des instructions est séparée de la mémoire de données. On aura besoin de deux bus d'adresse et de deux autres bus un pour les instructions et l'autre pour les données.

Cette structure est intéressante dans certaine type d'architecture comme le pipeline (exemple : le cisc_3 du chapitre III) où l'on est obligé de doubler la bande passante de la mémoire pour traiter les instructions et les données simultanément(Figure I-16)

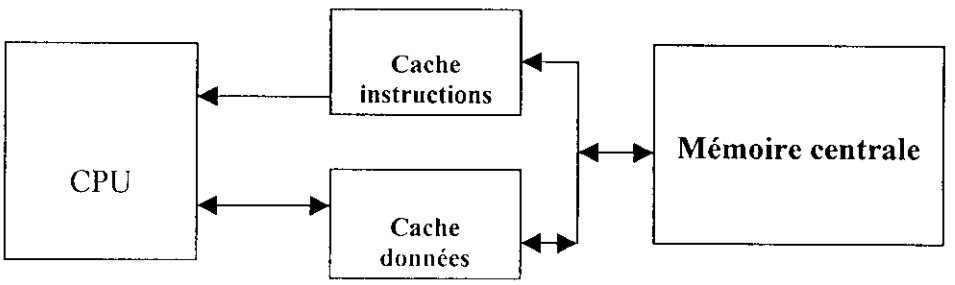


Figure I-17 : Architecture HARVARD

II- Classification des microprocesseurs

Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent.

Historiquement, les premières machines parallèles sont des réseaux d'ordinateurs, et des machines vectorielles et faiblement parallèles (années 70 - IBM 360-90 vectoriel, IRIS 80 triprocesseurs, CRAY 1 vectoriel...).

On distingue classiquement (Flynn 1969) quatre types principaux de parallélisme (taxonomie de Tanenbaum): SISD, SIMD, MISD et MIMD. Cette classification peut paraître de nos jours un peu artificielle (le moindre micro-processeur courant inclut lui-même plusieurs formes de micro-parallélisme). Cette classification est basée sur les notions de flot de contrôle (deux premières lettres, I voulant dire ``Instruction'') et flot de données (deux dernières lettres, D voulant dire ``Data'').[4]

II-1- Microprocesseur SISD

Une machine **SISD** (**S**ingle **I**nstruction **S**ingle **D**ata) est ce que l'on appelle d'habitude une machine de Von Neuman. Une seule instruction est exécutée à un moment donné et une seule donnée (simple, non-structurée) est traitée à un moment donné.

II-2- Microprocesseur SIMD

Une machine **SIMD** (**S**ingle **I**nstruction **M**ultiple **D**ata) est une machine qui exécute à tout instant une seule instruction, mais qui agit en parallèle sur plusieurs données, on parle en général de ``parallélisme de données''. Les machines SIMD peuvent être de plusieurs types:

- Vectorielle
- Systolique

Les machines systoliques sont des machines SIMD particulières dans lesquelles le calcul se déplace sur une topologie de processeurs, comme un front d'onde, et acquière des données locales différentes à chaque déplacement du front d'onde (comportant plusieurs processeurs, mais pas tous en général comme dans le cas (1)).
En général dans les deux cas, l'exécution en parallèle de la même instruction se fait en même temps sur des processeurs différents (parallélisme de donnée **synchrone**).

II-3 Microprocesseur MISD

Une machine **MISD** (**M**ultiple **I**nstruction **S**ingle **D**ata) peut exécuter plusieurs instructions en même temps sur la même donnée. Cela peut paraître paradoxal mais cela recouvre en fait un type très ordinaire de micro-parallélisme dans les micro-processeurs modernes: les processeurs à architectures pipelines.

II-4- Microprocesseur MIMD

Ces machines exécutent plusieurs jets de directives en parallèle sur des données différentes. Ainsi, les systèmes MIMD peuvent exécuter plusieurs tâches secondaires en parallèle afin de minimiser le temps d'exécution de la tâche principale. Il y a une grande variété de systèmes de MIMD. On en distingue essentiellement :

- **Système à mémoire partagée** Les systèmes à mémoire partagée ont des multiples processeur qui partagent le même espace adresse. Ceci signifie que la connaissance d'où les données sont enregistrées est sans souci à l'utilisateur car il y a seulement une mémoire consultée par tout le CPUs sur une la même base. Les systèmes à mémoire partagée peuvent être SIMD ou MIMD.

- **Systèmes à mémoire répartie** Dans ce cas-ci chaque unité centrale de traitement a sa propre mémoire associée. Les CPUs sont connectés par un réseau d'interconnexion et peuvent permuter des données entre leurs mémoires respectives. Contrairement aux machines à mémoire partagée, l'utilisateur doit se rendre compte de l'emplacement des données dans les mémoires locales et devra déménager ou distribuer ces données explicitement si nécessaire. Les systèmes à mémoire repartie peuvent être SIMD ou MIMD.

III- Relation microprocesseur logiciel

Quelque soit l'utilisation du microprocesseur, il y a une relation forte entre son architecture et le logiciel qu'il l'accompagne. Au début, ces logiciel était exprimés en langages très proches de l'architecture avec tous ses inconvénients (complexité, non probable sur d'autre architecture). Après, l'utilisation des langages évolués s'est rapidement généralisée, une généralisation poussée surtout par le progrès technologique et la volonté de réduire les coûts du développement du logiciel. [5]

Un microprocesseur ne saurait se concevoir sans environnement, on distingue :

- Les compilateurs.
- Les systèmes d'exploitation.

III-1- Les compilateurs

Les compilateurs ont pour objet de traduire les programmes exprimés en langage de haut niveau tel que : C^{++}, Fortran... en des programmes exécutables directement par le

microprocesseur. Le compilateur doit alors réaliser les optimisations du code dépendant de l'architecture. Par exemple, un compilateur doit générer une séquence d'instructions de façon à minimiser la l'interdépendance au niveau des pipelines.

Cette phase d'optimisation correspond à ce qui est appelé « réorganisation du code »

III-2- Système d'exploitation

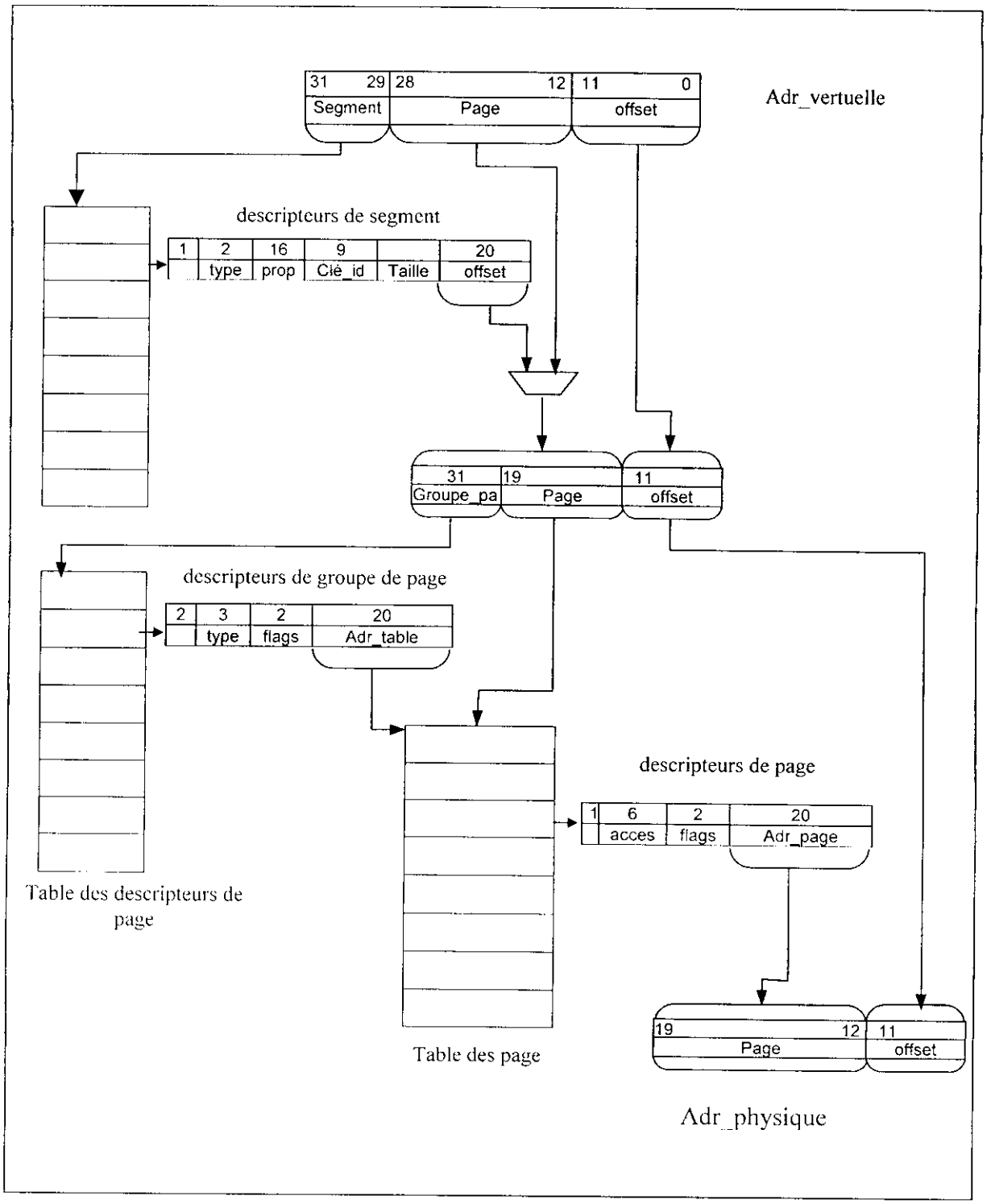
Il y a une interaction forte entre l'architecture des microprocesseurs et les caractéristiques des systèmes d'exploitation.

Malgré que la plus part des systèmes d'exploitations soient portables, c'est à dire adaptables à différentes architectures, leur efficacité est fortement liée aux caractéristiques de l'architecture du microprocesseur. Dans ce paragraphe, on va mentionner quelques éléments qui ''méritent '' d'être analysés au niveau de la conception ou du choix du microprocesseur.

- Communication interporcesseurs : les structures des systèmes d'exploitation évoluent vers des architecture de type micro-kernel qui autorise la distribution des fonctions système sur un réseau de machines, dans ce cas l'efficacité de la communication interprocesseurs est un facteur clé des systèmes dans ce contexte.
- Appels et interruption : ces événements sont des changements du contexte qui impliquent la sauvegarde celui du processus en cours d'exécution. Le contexte au niveau du processeur peut être assez important, certaine architecture présente des techniques de sauvegarde rapide comme le « fenêtrage », très utile pour les applications nécessitant des temps de réponse très courts.
- La gestion de la mémoire virtuelle : Une grande partie de cette fonction peut être réalisée par le microprocesseur, c'est à dire d'une façon matérielle, et ceci par ce qui est appelé le traducteur d'adresse (figure I-18),l'adresse virtuelle contenue dans le PC ou pointeurs se compose de trois parties : segment, page et « offset », qui donnent respectivement le segment de la mémoire, la page dans ce segment et la ligne dans la page

L'élément principal dans le traducteur d'adresse est le descripteur. Le descripteur est donné, de format défini, contenant un certain nombre d'informations sur une partition mémoire (segment, ensemble de page ou page), telle que le processus propriétaire, la taille. Il peut contenir des indicateurs et des clés pour les processus et données chiffrés. Souvent, il est associé à cette fonction un bloc spécifique dit de gestion d'adresses. Les mécanismes de protection y sont implicitement gérés.

Après traduction on n'aura pas seulement l'adresse physique mais aussi des informations sur les données adressées et éventuellement un avertissement en cas de violation.



Figue 1.18: Traducteur d'adresse

Conclusion

Après avoir été conçus pour tout emploi général, les microprocesseurs sont de plus en plus conçus pour des emplois spécifiques (DSP, micro contrôleur, micro processeur de graphisme etc.

Le modèle de base d'un micro processeur est celui de Babagge. Il comporte un accumulateur, une unité arithmétique et un registre d'instruction. Von Newman a introduit une nouvelle idée qui porte sur la rupture de séquence de programme. Ce modèle n'est pas resté figé. D'autres notions ont été également introduites comme l'interruption, la pile, la segmentation de la mémoire et le parallélisme des traitements.

Jusqu'à aujourd'hui, il n'existe pas de méthode de conception universelle quoique des recherches en cours parlent déjà (ce qui est extrêmement objectif et de tout à fait prévisible) de microprocesseurs « orientés objet ». L'obtention de composant selon les outils utilisés est plus ou moins automatisée. L'ambition des ces outils est l'obtention d'une architecture matérielle depuis une certaine modélisation de l'application à intégrer.

Introduction

Depuis la naissance du premier circuit intégré dans les années soixante, le domaine de la microélectronique n'a cessé de se développer. Le perfectionnement constant des procédés de fabrication et des logiciels de conception assistée par ordinateur (CAO) a conduit aux densités d'intégration que l'on connaît aujourd'hui : jusqu'à dix millions de transistors par puce. Un cycle complet de fabrication a pour objectif l'obtention d'un circuit fiable, conforme aux spécifications initiales, respectant les exigences de moindre coût en un temps le plus court possible. La phase de conception du circuit intégré, partie intégrante de ce cycle qui précède la fabrication proprement dite, doit suivre cette évolution et s'adapter aux critères cités.

Ainsi, il est indispensable de vérifier *a priori* le comportement électrique du circuit qui va être soumis au fabricant : le fonctionnement est-il conforme à celui que l'on s'est fixé? Cette étape appelée simulation repose sur la connaissance de modèles des composants constitutifs du circuit.

La première génération des simulateurs commercialisés s'inspire essentiellement des principes et algorithmes de leur précurseur SPICE. Une nouvelle génération apparaît aujourd'hui, avec de nouveaux algorithmes optimisés pour simuler dans des délais raisonnables des circuits complexes.

En fait, cette stratégie a initialement été adoptée pour la conception des circuits numériques et a donné naissance au langage standard VHDL

I- Les différentes approches de conception

Il existe plusieurs approches pou concevoir l'architecture d'un microprocesseur (figure II-1). La plus part des processeurs dits CISC ont été conçu selon une approche ascendante (bottum-up approach) pour respecter les contraintes de compatibilité avec les versions antérieures mais également pour minimiser les difficultés inhérente à la réalisation d'une nouvelle architecture. Dans cette approche, le concepteur part d'une architecture pré-établie. La seconde approche (middle approach) a surtout été utilisée pour la réalisation de microprocesseurs dédiés à l'exécution d'un langage particulier. La dernière approche (top-down approach) est appelée descendante puisqu'elle débute par l'analyse des opérations à forte occurrence pour ensuite déterminer l'architecture la plus adaptée. On trouve dans cette classe les microprocesseurs dits RISC.

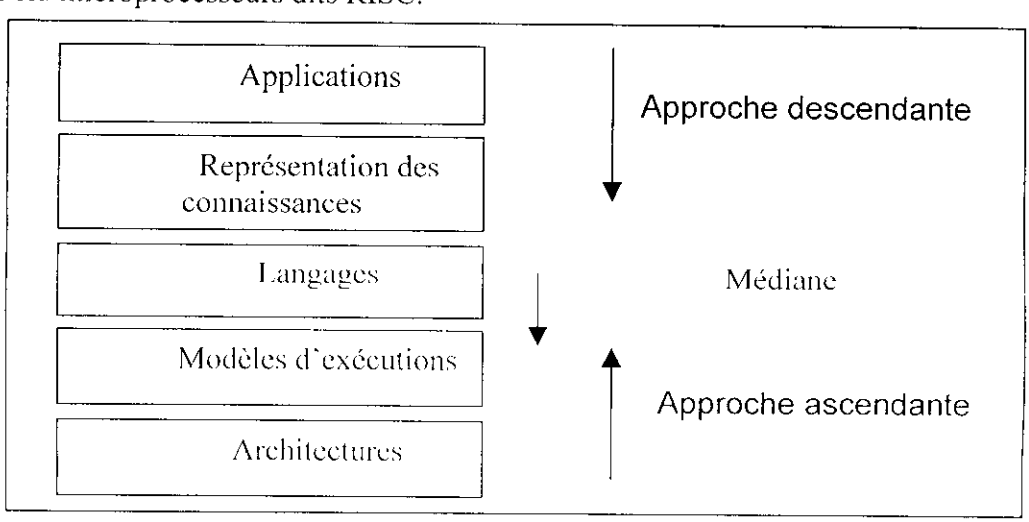


Figure II -1: Les approches de conception

II- Expression d'une architecture

Un circuit est destiné à résoudre un problème. On peut donc voir ce circuit d'une manière plus ou moins proche de la réalité physique ou de l'aspect algorithmique.

On distingue trois angles de vue, plus couramment désignés par les termes domaines d'expression pour exprimer une architecture (figure II-2).

Ces domaines d'expression sont eux-mêmes découpés en cinq niveaux de description plus ou moins abstraits par rapport aux détails du circuit. Les cinq degrés d'abstraction du domaine comportemental par exemple sont : interrupteur, logique, transfert de registres, algorithme et système. Les niveaux système et algorithmique sont quelquefois regroupés dans un seul niveau.[8]

Les trois domaines d'expression sont :

- Le domaine comportemental. Il décrit le comportement d'un circuit, autrement dit ce qu'il fait.
- Le domaine structurel. Il voit le circuit comme une interconnexion d'objets architecturaux de base et fait le lien entre le domaine comportemental et le domaine physique.
- Le domaine physique, souvent représenté par le dessin de masque du circuit. Il présente la structure de la réalisation physique de l'architecture.

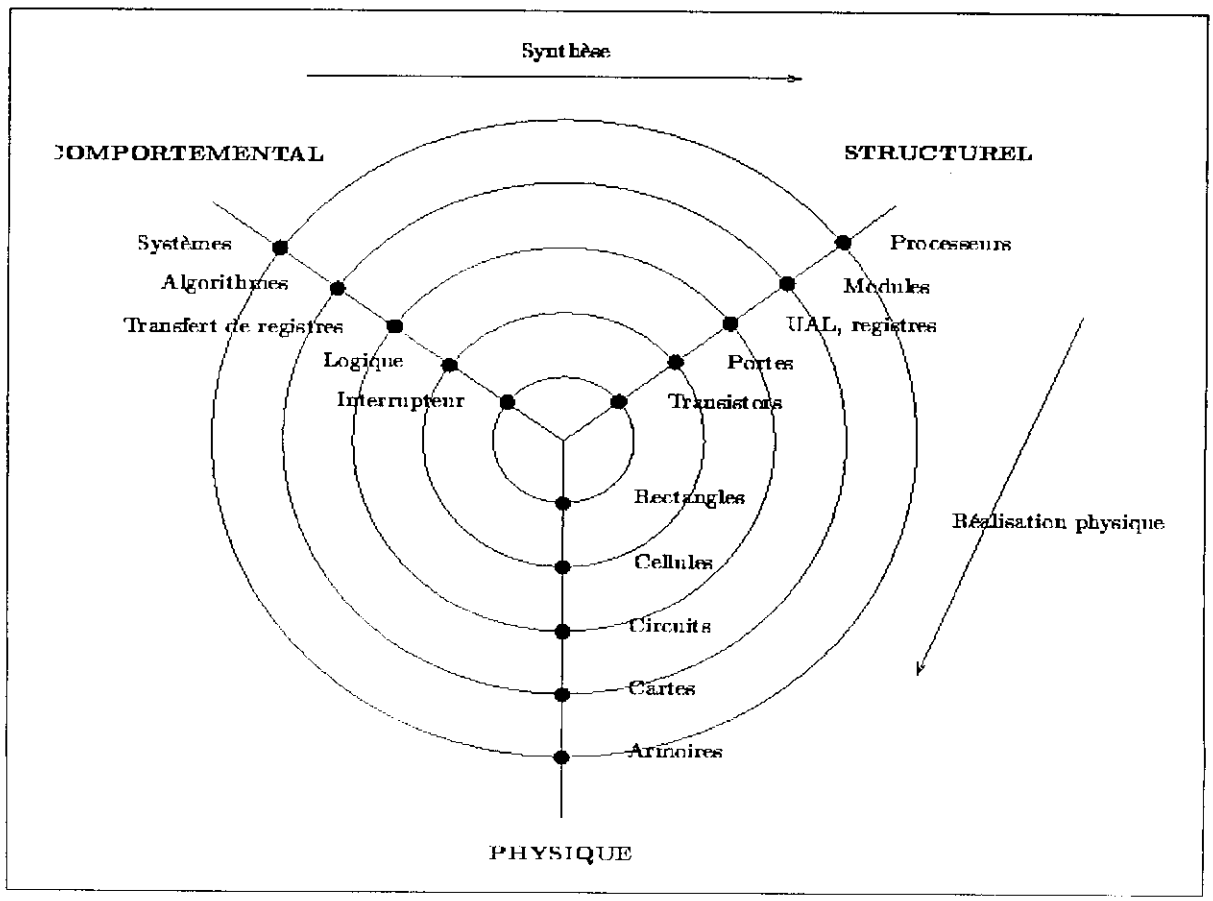


Figure II-2 : Diagramme en Y introduit par Gajski décrivant les trois vues d'une architecture

La spécification de départ d'un circuit relève généralement du domaine comportemental. Diverses étapes de synthèse mènent ensuite à une description physique.

III- Choix du mode de description

Le choix d'un mode de description s'inscrit dans la méthodologie de conception des raisons aussi bien technique que stratégiques vont orienter le concepteur vers une méthode de conception adaptée (figure II-3). D'un point de vue technique, on tient compte de la nature de la fonction à réaliser et du composant qui va accueillir cette fonction. Les outils disponibles et les habitudes de travail du concepteur seront aussi un critère de choix du mode de description. Les critères stratégiques seront par exemple la 'portabilité'. Ce terme indique que le fichier source de la description n'est pas dédié à un composant particulier. On s'affranchit ainsi momentanément de la phase de choix du composant. La conception peut donc être testée sur plusieurs types de composants. Cette façon de procéder peut s'avérer stratégique lorsqu'on souhaite mettre à jour des conceptions antérieures dans des composants de technologie plus récente ou lorsque l'on souhaite reprendre des éléments d'une conception dans un projet différent.

Une entrée schématique utilise les composants de base (hard macro, composant optimisés) fournis pour un composant donné ou de moins une famille de composant. Le fichier source(schéma) est donc difficilement portable. Même si l'ensemble des composants de base utilisé dans la conception numérique se limitent à des fonctions restreintes et générique (bascules, compteurs, multiplexeurs, additionneurs, etc.), chaque constructeur propose ça propre librairie de composant et les noms utilisés pour ces composants génériques ne sont en générale acceptés que par le compilateur cible. Par contre la nature même d'un schéma (structurel)est bien adaptée pour imposer au synthétiseur une structure donnée. On utilisera ce type d'entrée lorsque la nature de la fonction à implanter est très structurée. Les blocs fonctionnels d'un synoptique de principe peuvent être repris en apportant ainsi une meilleure lisibilité de schéma.

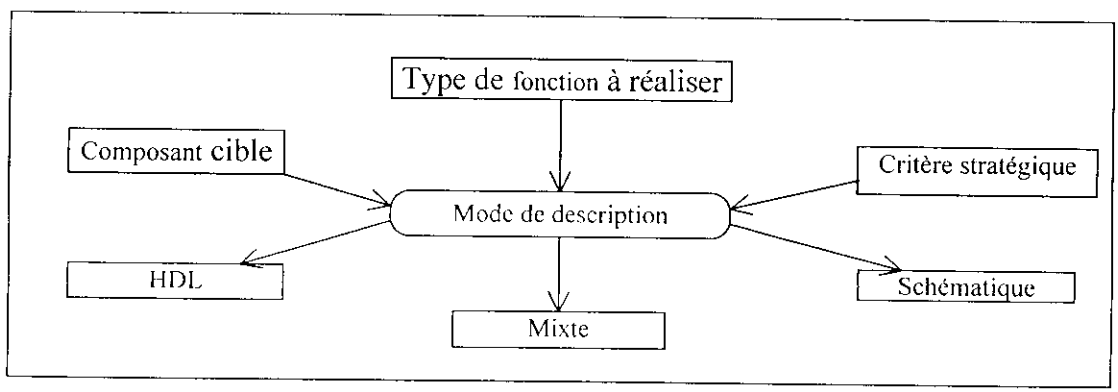


Figure II-3: choix du mode de description

Une entrée HDL apporte de nombreux avantages. Un langage de type comportemental permet de rendre le code source indépendant de la cible. Alors qu'en schématique il existe autant de formats que d'outils, un code Hdl normé(Vérilog, VHDL, ABEL) sera au contraire accepté par la plus part des outils (simulateurs, synthétiseur).

Une entrée mixte consiste à définir certains éléments hiérarchiques et d'autre de manière syntaxique. Il pourra par exemple s'avérer judicieux de définir schématiquement la

structure globale d'une conception et définir de manière comportementale chaque bloc fonctionnel

III-1- Les différents types de langages

Deux familles de langages peuvent être distinguées: les langages de bas niveau, choisis pour des raisons d'efficacité numérique dans le codage des primitives des simulateurs, et les langages dédiés à la description de matériel ou HDL (*Hardware Description Language*).

III-2- Les langages de bas niveau

Ces langages (FORTRAN, C) sont utilisés pour coder les primitives des simulateurs électriques: ce sont des langages efficaces pour les calculs numériques, mais difficilement utilisables par un concepteur de circuits.

Cependant, pour faciliter l'écriture du code C de nouveaux modèles, des outils ont été développés. Ils permettent en particulier de générer un système d'équations d'états à partir de la simple spécification d'une fonction de transfert de Laplace.

Dans certains cas, le modèle décrit grâce à un langage de haut niveau de type VHDL, est directement traduit en C avant sa compilation.. En plus de la vérification syntaxique, la continuité des fonctions et celle des dérivées sont analysée pour les besoins du simulateur. Un lissage des discontinuités peut même être effectué et les équations susceptibles de provoquer des erreurs numériques sont détectées.

III-3- Les langages de description de matériel (HDL)

Ce sont des langages spécialement conçus pour la description de systèmes, électroniques ou autres. Ils constituent l'interface d'entrée des simulateurs, des outils de preuve formelle et de synthèse. Ils peuvent contribuer à faciliter la spécification et la documentation de circuits.

Trois types de description doivent être envisagés:

• **La *description structurelle***, qui donne des informations sur la structure des blocs et composants utilisés. Le langage d'entrée du simulateur SPICE est de ce type. Dans une hiérarchie de description, les éléments décrits de cette façon ne peuvent constituer des éléments terminaux.

• **La *description comportementale***. ELle exprime le fonctionnement du bloc, c'est à dire ses équations, sans se soucier de sa structure interne.

• **La *description flot de données***, qui est définie par VHDL pour exprimer rapidement les flots de données sortant du modèle en fonction des flots entrants, sans référence à la structure interne.

IV- Le langage VHDL

IV -1- Historique du langage VHDL

Le langage VHDL a été développé, dans les années 80, par le DOD (**D**epartement **o**f **D**efense). L'objectif était de disposer d'un langage commun avec les fournisseurs pour décrire les circuits complexes. Le terme VHDL signifie **V**HSIC (Very High Speed Integrated Circuit) **H**ardware **D**escription **L**anguage. En 1987, une première version du langage est standardisée par l'IEEE(**I**nstitute of **E**lectrical and **E**lectronics **E**nginneers). Il est à noter que la norme ne définit aucune méthodologie de travail. Elle ne fait que fournir un outil de

description. Cette norme, ne reconnaît que le type **bit**. Celui-ci comprend uniquement les états '0' et '1'. Cela pose des problèmes lors de la simulation. En effet, comment modéliser un bus haut impédance, ou comment réagir lorsque qu'une incohérence du code provoque un court-circuit ? Pour résoudre ces problèmes, le type **Std_Logic** est créer et normalisé par l'IEEE en 1993. Celui-ci comprend 9 états logiques. Il permet d'exprimer, entre autre, l'état haute impédance 'Z', l'état indéterminé 'X' et l'état haut faible 'H' (pull-up).

Parallèlement, une évolution du VHDL est normalisée en 1993. Cette nouvelle version supprime quelques ambiguïtés de la version 87, et surtout met à disposition de nouvelles commandes. Actuellement, tous les outils dignes de ce nom supportent le VHDL-93.En 1997, de nouvelles libraires ont été normalisées de manière à ajouter des fonctions pour la synthèse. Nous citerons la librairie Numeric_Std, qui définit les opérations arithmétiques pour le typeStd_Logic.

Le standard ***VHDL-A*** ou ***IEEE 1076.1*** définit les extensions analogiques du standard digital VHDL 1076 et doit permettre la description de systèmes mixtes analogiques digitaux pouvant appartenir à différents domaines physiques: systèmes électriques, mécaniques, thermiques... Notons que le langage HDL-A a été conçu en suivant les principaux objectifs de VHDL-A .

Citons aussi le standard ***MHDL*** (*MIMIC HDL*) dédié à la description et simulation de circuits micro-ondes analogiques et mixtes. C'est un nouveau langage, indépendant de VHDL. L'interface de ce langage avec un ensemble d'outils CAO concernant entre autre la synthèse analogique et l'automatisation des tests, est définie dans le même temps.

Ces travaux de standardisation sont très importants pour l'échange de modèles entre les compagnies industrielles, la construction de bibliothèques de modèles et le développement de nouveaux outils CAO tels que les outils de synthèse.

IV -2 Applications du langage

Le VHDL est un langage unique permettant de faire :

- De la **spécification** : le langage VHDL est très bien adapté à la modélisation de systèmes numériques complexes grâce à son niveau élevé d'abstraction. Le partitionnement en plusieurs sous-ensembles permet de sub-diviser un modèle complexe en plusieurs éléments prêts à être développés séparément.

- De la **simulation** : La notion de temps, présente dans le langage, permet son utilisation pour décrire des fichiers de simulation (test-bench). Le modèle comportemental avec les fichiers de simulation peuvent constituer, ensemble, un cahier des charges. Les fichiers de simulation peuvent également être utilisés avec un banc de tests de production.

- De la **synthèse logique** : les logiciels de synthèse permettent de traduire la description VHDL en logique. Il est ainsi possible d'intégrer la description dans un composant programmable (CPLD.FPGA) ou dans un circuit ASIC.

- De la **preuve formelle** : le langage permet de prouver formellement que deux descriptions sont parfaitement identiques au niveau de leur fonctionnalité.

IV -3 Pourquoi utiliser le langage VHDL ?

L'électronicien a toujours utilisé des outils de description pour représenter des structures logiques ou analogiques. Le schéma structurel que l'on utilise depuis si longtemps n'est en fait

qu'un outil de description graphique. Aujourd'hui, l'électronique numérique est de plus en plus présente et tend bien souvent à remplacer les structures analogiques utilisées jusqu'à présent. Ainsi, les fonctions numériques à réaliser ne cessent de prendre de l'ampleur. Il est dès lors indispensable d'utiliser un langage de haut niveau pour maîtriser la complexité grandissante des systèmes numériques. Le VHDL est l'un des langages modernes et puissants. Il est nettement plus performant que la description par schéma.

Le deuxième point fort du VHDL est d'être un langage de description comportementale de haut niveau. Les anciens langages, tel ABEL, ne disposaient pas de cette fonctionnalité. En fait, un langage est dit de haut niveau lorsqu'il fait le plus possible abstraction de l'objet pour lequel il est écrit. Dans le cas du VHDL, il n'est jamais fait référence au composant ou à la structure pour lesquels on l'utilise. Ainsi, il apparaît une notion très importante : la portabilité des descriptions VHDL. Le langage VHDL est normalisé. Il n'est pas la propriété d'un vendeur d'outils. Cette norme a poussé de grandes sociétés de logiciels à l'utiliser comme langage de description de matériel pour leur outil. Le langage est ainsi devenu un standard reconnu par une majorité des vendeurs d'outils de synthèse.

Le langage VHDL a été initialement utilisé pour la spécification et la simulation de systèmes complexes. Ce langage de haut niveau permet une décomposition hiérarchique et dispose de la notion de temps. Cela a permis de maîtriser la complexité des circuits, particulièrement dans le cas des ASICs. Le même langage est utilisé pour toutes les étapes du développement (spécification, simulation et synthèse). Il permet également la réalisation de bibliothèques de composants réutilisables d'un projet à l'autre.

IV-4 Utilisation du langage VHDL dans la synthèse

Les débuts de son utilisation dans la synthèse de circuit fut assez difficile. Chaque société ayant adapté le langage VHDL à sa manière. De 1993 à 1997, différentes adaptations de la norme IEEE définissent leur Conception numérique : description VHDL et utilisation pour la synthèse. Il faut attendre la fin des années 90 pour que tous les outils intègrent ces nouvelles modifications.

Parallèlement, la forte évolution des circuits logiques programmables dans les années 1990, nécessita de disposer d'un langage de haut niveau afin de maîtriser la complexité toujours plus importante. En Europe, le VHDL s'est imposé comme un standard reconnu par tous les principaux vendeurs d'outils de développement.

IV-4-1 Simulation et synthèse

Le déroulement des différentes étapes de développement d'un projet en VHDL est illustré par la figure II –4 .

Le développement en VHDL nécessite l'utilisation de deux outils : le simulateur et le synthétiseur. Le premier va nous permettre de simuler notre description VHDL avec un fichier de simulation appelé « *test-bench* ». Cet outil interprète directement le langage VHDL. Le simulateur comprend l'ensemble du langage.

L'objectif du synthétiseur est très différent. Il doit traduire le comportement décrit en VHDL en fonctions logiques de bases. Celles-ci dépendent de la technologie choisie. Cette étape est nommée : "synthèse". Le langage VHDL permet d'écrire des descriptions d'un niveau comportemental élevé. La question est de savoir si n'importe quelle description comportementale peut être traduite en logique ?

L'intégration finale dans le circuit cible est réalisée par l'outil de placement et routage. Celui-ci est fourni par le fabricant de la technologie choisie.

Avec les outils actuels, il est possible de disposer de fichiers VDHL à chaque étape. Le même fichier de simulation (*test-bench*) est ainsi utilisable pour vérifier le fonctionnement de la description à chaque étape.

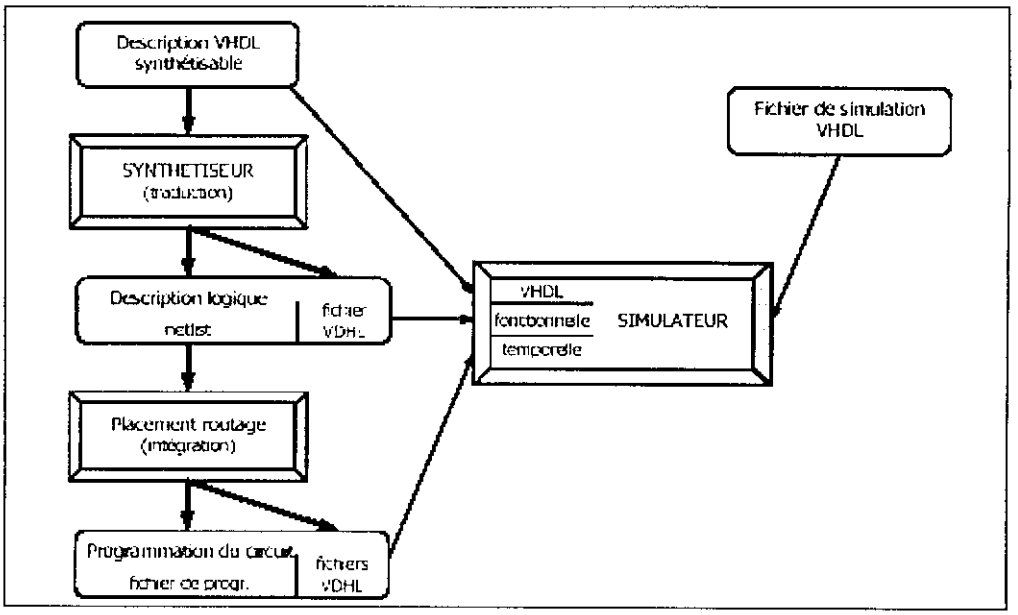


Figure II –4 :phases de développement.

IV-4-2- L'ensemble d'un code VHDL est-il synthétisable ?

Le langage a été initialement conçu pour spécifier le comportement de systèmes numériques. Des instructions de haut niveau et la notion de temps ont été introduit dans le langage VHDL. La conséquence est que l'ensemble du langage n'est pas synthétisable.

Voici l'exemple de la description en VHDL d'une horloge à 10Mhz :
Horloge <= **not** Horloge **after** 50 ns ;

Cette description est correcte en VHDL, mais il est évident qu'il n'existe aucun circuit logique qui est capable de générer un tel signal. Cette description est donc non synthétisable. Mais elle est parfaitement utilisable dans un fichier de simulation.

Toute la difficulté est de savoir ce qui est synthétisable ou non. Il y a certains cas clairs comme l'instruction *after*. Cette instruction fait référence au temps, elle est donc non synthétisable. Mais la frontière du domaine des descriptions VHDL synthétisables n'est pas toujours aussi net. Dans beaucoup de cas c'est la manière dont les instructions sont utilisées qui rend la description non synthétisable. La limite peut même varier d'un synthétiseur à l'autre.

En prenant des précautions et en pensant toujours circuit lorsque nous écrivons des descriptions, il est possible d'assurer que celles-ci seront synthétisables. Il est indispensable de bien comprendre l'étape de synthèse. Il est nécessaire de savoir comment le synthétiseur réagit vis à vis des descriptions VHDL.

L'étape la plus importante de la synthèse automatique, à savoir celle pendant laquelle les phases d'allocation, d'ordonnancement et d'assignation sont réalisées.

- L'étape d'allocation détermine le type et le nombre de ressources à utiliser dans l'architecture afin de respecter les contraintes données par le concepteur.
- Les opérations et les accès mémoires de l'algorithme sont alors ordonnancés par le logiciel.

Enfin, la phase d'assignation relie chaque opération à une ou plusieurs ressources.

IV-5- La portabilité des descriptions VHDL

Les concepteurs souhaitent être indépendants des outils qu'ils utilisent. Ils demandent que leurs descriptions soient portables.

Dans le monde des systèmes numériques nous pouvons définir deux portabilités:

_ La portabilité vis-à-vis des logiciels.

_ La portabilité vis-à-vis des circuits.

Le langage VHDL est devenu un standard dans le monde de la conception numérique. Il existe de nombreux outils qui acceptent ce langage comme entrée. Il est donc possible d'utiliser une même description avec plusieurs outils différents. Mais il faut encore garantir que chaque outil aura la même interprétation de la description VHDL. Dès lors, il est important de ne pas utilisée les spécificités d'un outil. En particulier il ne faut pas utiliser les bibliothèques propres à un outil mais uniquement les bibliothèques normalisées IEEE (Std_Logic_1164 et Numeric_Std).

Pour les circuits, la portabilité ne pourra plus être assurée dès que nous utilisons une ressource propre à une technologie pour respecter des contraintes de vitesse ou optimiser la surface.[7]

IV-6- Description de sortie

Finalement, une fois l'architecture créée, il ne reste plus qu'à la décrire dans le langage de sortie de l'outil considéré. Cette sortie est généralement décrite par une netlist au niveau *RTL* qui peut alors être synthétisée par un outil de synthèse logique.

V- La technologie cible

Les progrès technologiques soutenus dans le domaine des circuits intégrés ont permis la réduction des coûts, de la consommation, et c'est maintenant un lieu commun d'affirmer que les circuits intégrés spécifiques d'une application ont permis une réduction de la taille des systèmes numériques ainsi que la réalisation de circuits de plus en plus complexes, tout en améliorant leurs performances et leur fiabilité. Aujourd'hui les techniques de traitement numérique occupent une place majeure dans tous les systèmes électroniques modernes grand public, professionnel ou de défense. De plus, les techniques de réalisation de circuits spécifiques, tant dans les aspects matériels (composants reprogrammables, circuits précaractérisés et bibliothèques de macrofonctions) que dans les aspects logiciels (placement-routage, synthèse logique) font désormais de la microélectronique une des bases indispensables pour la réalisation de systèmes numériques performants. Elle impose néanmoins une méthodologie de développement en CAO très structurée.

V-1- les ASICs (application specific integer circuit)

Dans un premier temps, nous allons rappeler quelques concepts autour des circuits intégrés pour applications spécifiques ASIC. Les circuits ASIC constituent la troisième génération de circuits intégrés qui a vu le jour au débutdes années 80. En comparaison des circuits intégrés standards et figés proposés par les fabricants, l'ASIC présente une personnalisation de son fonctionnement, selon l'utilisation, accompagnée d'une réduction du temps de développement, d'une augmentation de la densité d'intégration et de la vitesse de fonctionnement. En outre sa personnalisation lui confère un autre avantage industriel, c'est évidemment la confidentialité. Ce concept d'abord développé autour du 1silicium s'est ensuite

étendu à d'autres matériaux pour les applications microondes ou très rapides (GaAs par exemple).

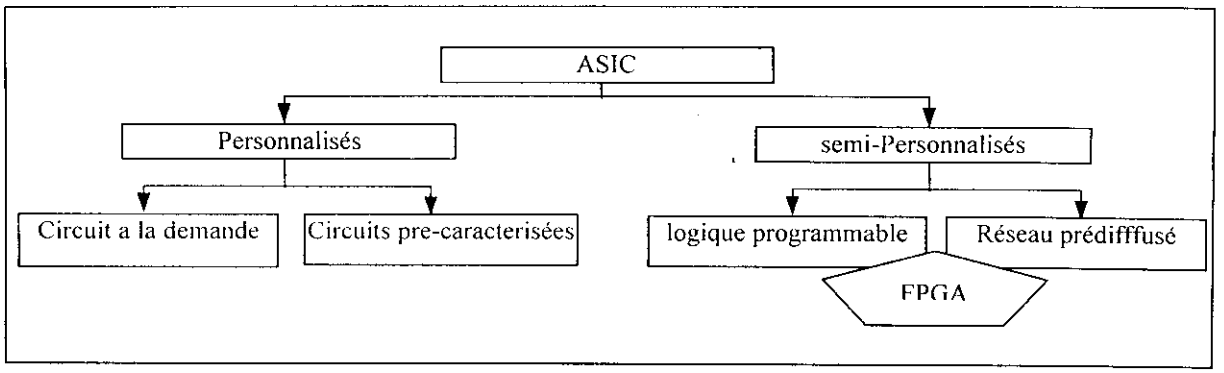


Figure II-4 :Les circuits ASICs

Par définition, les circuits ASIC regroupent tous les circuits dont la fonction peut être personnalisée d'une manière ou d'une autre en vue d'une application spécifique, par opposition aux circuits standards dont la fonction est définie et parfaitement décrite dans le catalogue de composants. Les ASIC peuvent être classés en plusieurs catégories selon leur niveau d'intégration, en fait un ASIC est défini par sa structure de base (réseau programmable, cellule de base, matrice, etc.). Sous le terme ASIC deux familles sont regroupées, les semi-personnalisés et les personnalisés.

V-1-1- Les circuits personnalisés

Ce sont des circuits non préfabriqués. Pour chaque application on optimise le circuit intégré, ce qui conduit à la création de son propre composant. Cette famille comprend :

les circuits à la demande,
les circuits précaractérisés.

- **Les circuits à la demande**

Les solutions circuit à la demande (qu'on appelle full custom en anglais) présentent l'avantage d'autoriser une meilleure optimisation duplacement puisque

celui-ci n'est pas prédéfini. On dispose d'une bibliothèque de modèles mathématiques de comportement et via un "compilateur de silicium" logiciel très sophistiqué on peut

concevoir toute l'architecture du circuit en faire une validation logicielle (simulation logique) puis dans une avant dernière étape en déduire le dessin des divers masques de fabrication.

Toutes les opérations, de la conception à la fabrication, sont effectuées de façon est au choix du concepteur, que ce soit la taille du composant, le nombre de broches, le placement du moindre transistor. C'est l'ASIC le plus optimisé car aucune contrainte ne lui est imposée. Le placement des blocs fonctionnels et le routage des interconnexions, même si ces opérations sont assistées par ordinateur, sont effectuées avec beaucoup plus d'interventions manuelles pour atteindre l'optimisation au niveau de chaque transistor. Cependant, les phases de mise au point sont longues et onéreuses, il va de soi que la rentabilisation des investissements de développement nécessite un fort volume de production.

V -1-2- Les circuits précaractérisés

Pour la réalisation de circuits précaractérisés on dispose d'une bibliothèque de circuits élémentaires que le fondeur sait fabriquer et dont il peut garantir les caractéristiques et on les associe pour réaliser le circuit à la demande. Ici encore on utilisera en fabrication des masques personnalisés pour chacune des couches diffusées et des métallisations.

Le concept est très semblable de celui des circuits à la demande. La seule différence réside dans la réalisation du schéma puisque l'on accède à une bibliothèque de cellules prédéfinies générant de très nombreuses fonctions élémentaires ou élaborées. Cette dernière constitue un véritable catalogue dans lequel le concepteur se sert pour constituer son schéma. Il existe trois types de cellules :

les cellules standards (standard cells) correspondent à la logique classique,
les mégacellules (megacells) peuvent être des blocs du type microprocesseur, périphérique,
les cellules compilées (compilable cells) dont les blocs RAM ou ROM.

Il est nécessaire de personnaliser complètement la diffusion, et par conséquent de créer tous les masques. Cependant un avantage évident en découle : alors qu'il est impossible avec les prédiffusés d'utiliser à 100% le réseau de cellules ou portes, ce qui se traduit par une perte de silicium, les précaractérisés permettent d'exploiter complètement la surface du circuit.

V-1-3- Les circuits semi-personnalisés

Les semi-personnalisés sont des réseaux prédéfinis de transistors ou de fonctions logiques qui nécessitent une personnalisation de l'utilisateur pour réaliser la fonction désirée. Cette famille comprend :

· les réseaux logiques programmables.
· les réseaux prédiffusés.

A -Les réseaux logiques programmables

Ce composant ne nécessite aucune étape technologique supplémentaire pour être personnalisé. Nous y trouvons les PAL/PLD, ce sont des circuits standards programmables par l'utilisateur grâce à différents outils de développement. La programmation consiste à établir des connexions .

Les circuits logiques programmables incluent un grand nombre de solutions, toutes basées sur des variantes de l'architecture des portes ET OU. nous y trouvons : ·

PAL (Programmable Array Logic) matrice ET programmable, matrice OU figée).

EPLD (Erasable PLD) effaçables par rayons ultraviolet, ils peuvent être reprogrammer.

EEPLD (Electrically Erasable PLD) programmables et effaçables électriquement, ils peuvent être reprogrammés sur site. Les limites de l'architecture du PLD résident dans le nombre de bascules, le nombre de signaux d'entrées/sorties, la rigidité du plan logique ET OU et des interconnexions.

Précisons que ces composants très souples d'emploi sont limités à des fonctions numériques et adaptés à des productions de petites séries et ne présentent aucune garantie quant à la confidentialité.

On va étudier les circuits PALs qui constituent la base des EPLD et EEPLD mais avant on doit donner un aperçu sur l'élément de base dans leurs architecture :« la cellule d'interconnexion ». [7]

B- La technologie d'interconnexion

Si l'avènement du transistor à révolutionné l'industrie électronique, l'apparition de la connexion logique programmable par l'utilisateur au début de années soixante dix à donné naissance à une industrie nouvelle qui n'a depuis, cessé de croître. Au cœur de la notion de programmation se trouve ces cellules qui réalisent une connexion entre deux réseaux. On distingue deux types de connexions : définitive et temporaire.

Les connexions définitives:

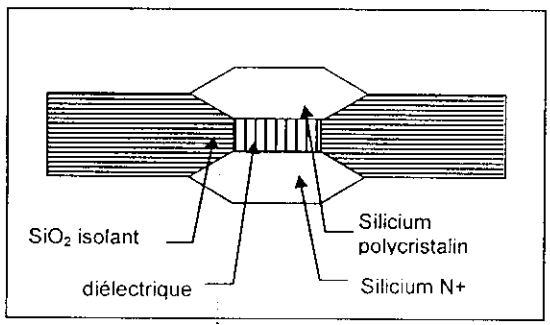


Figure II-5 : cellule antifusible

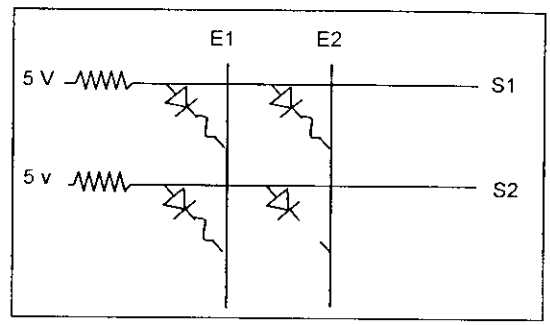


Figure II-6 : cellule à fusible

Elles sont réalisées par des cellules à fusible ou à antifusible. Les cellules à fusibles (figure II-6) sont constituées de diodes en série avec un fusible et la programmation consiste à claquer le fusible par un courant supérieur à son nominale. Ainsi, par exemple sur la figure II-6, on a réalisé les équations logiques suivantes : S1=E1 E2 , S2= E1.

Pour les cellules antifusible, la programmation consiste à diffuser du silicium conducteur dans la couche diélectrique(figure II-5) ainsi la résistance du diélectrique chute pour établir la connexion.

Les connexions temporaires :

On distingue deux principaux types : les cellules à transistor MOS à Grille flottante et les cellules de type SRAM.

L'élément principal du premier type est le transistor MOS à Grille flottante(figureII-8),la programmation consiste à piéger les électrons dans la grille flottante pour cela une tension de l'ordre de 13 V est appliquée entre la grille et la source , une tension plus faible entre drain et source, cette dernière tension fournit aux électrons une énergie cinétique importante, le fort champ électrique entre la grille et le canal dévie les électrons vers la grille flottante où il sont piégé.

Ainsi la grille flottante rend le transistor conducteur, et la connexion est établie.

Pour l'effacement deux manières : l'exposition à l 'UV décharge la grille flottante, ou l'application d'une tension inverse qui libère les électrons de la grille flottante.

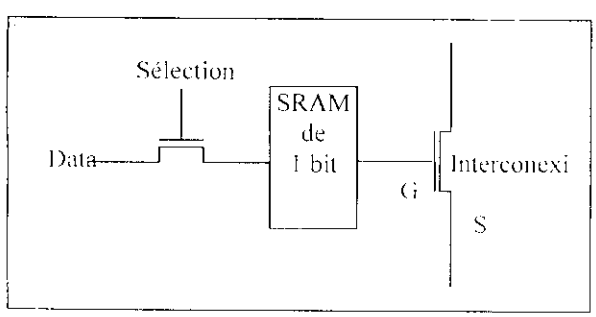
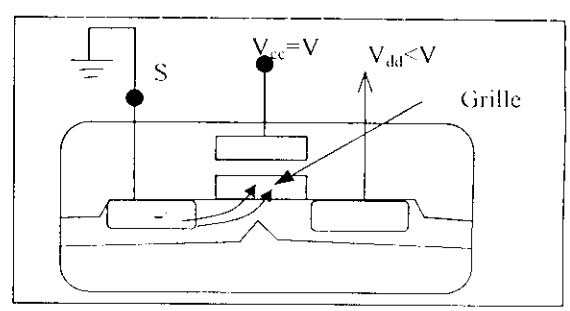


Figure II-7 :celle de type SRAM



FigureII-8 transistor MOS à grille flottante

Dans la cellule de type SRAM le canal Drain Source d'un transistor MOS constitue l'élément de connexion, ouvert ou fermé selon la tension de la grille, alors le maintient de la configuration nécessite un élément de mémorisation, dans cette cellule c'est la SRAM (figure II-7).

C- Architecture des PLDs :

Codage des fonctions :

Trois principes sont utilisés pour le codage des fonctions combinatoires :

MUX : des multiplexeurs câblés constituent l'essentiel des ressources de codage.

LUT :des mémoire (Look Up Table) contiennent l'équivalant d'une table de transposition. Ces mémoires sont en pratique des SRAM.

PLA :une fonction combinatoire peut être exprimé sous forme d'une somme de termes de produit ou produit de termes de somme. Les PLA exploitent ce principe. alors pour une combinaison on doit réaliser les produit et la somme, la figure II-10 illustre l'architecture de base réalisant les combinaisons

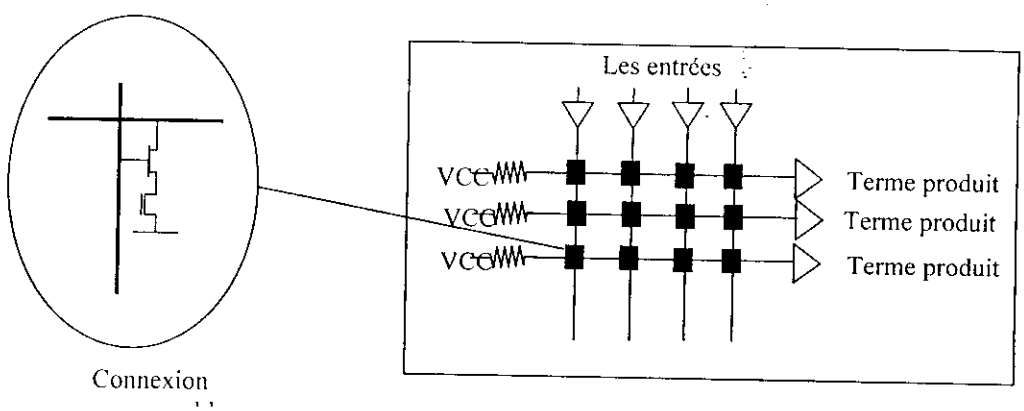


Figure II-10: réalisation des fonctions logiques

Si le transistor à grille flottante est programmé, il constitue un interrupteur ouvert et l'entrée n'influent pas sur la sortie, dans le cas contraire on aura un NOR logique entre les entrées, en inversant la sortie on aura un OR ou en inversant toutes les entrées on aura un AND logique, ainsi on peut déduire une architecture qui peut réaliser n'importe quelle fonction combinatoire en programmant les interconnexions (**figure II-11**).

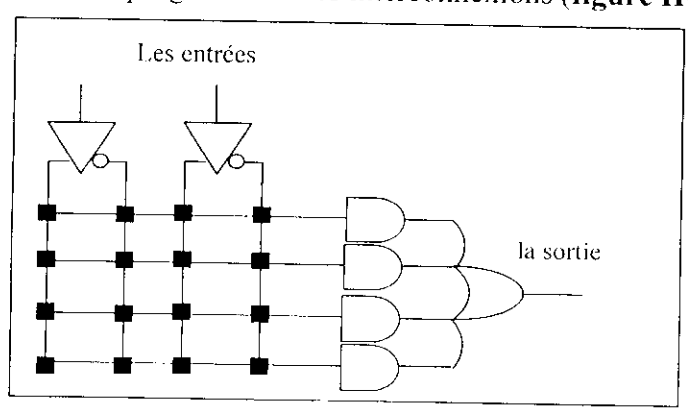


Figure II-11 : fonction combinatoire

On constate que les interconnexions sont disposées sous forme d'une matrice, ce qui leurs donne le nom matrice d'interconnexion.

❖ **Les circuits PALs :**

Les connexions sont réalisées soit par fusible pour les PALs bipolaire, soit par des transistors FGMOS (Floating Gate MOS) pour les versions récentes.

Plusieurs familles de PAL ont été réalisées, on peut citer les PALs combinatoires (figure II-14) les PALs à registres (figure II-13) et les PALs versatiles(figure II-15)qui introduisent la notion de la macrocellule configurable. [7]

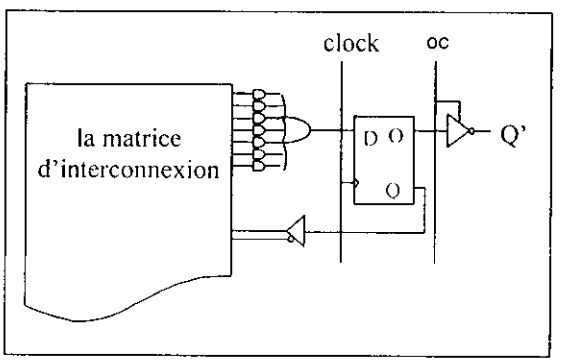


Figure II-13: sortie à registre

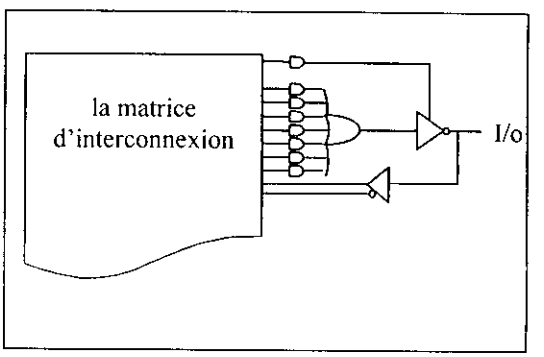


Figure II-14 : sortie combinatoire

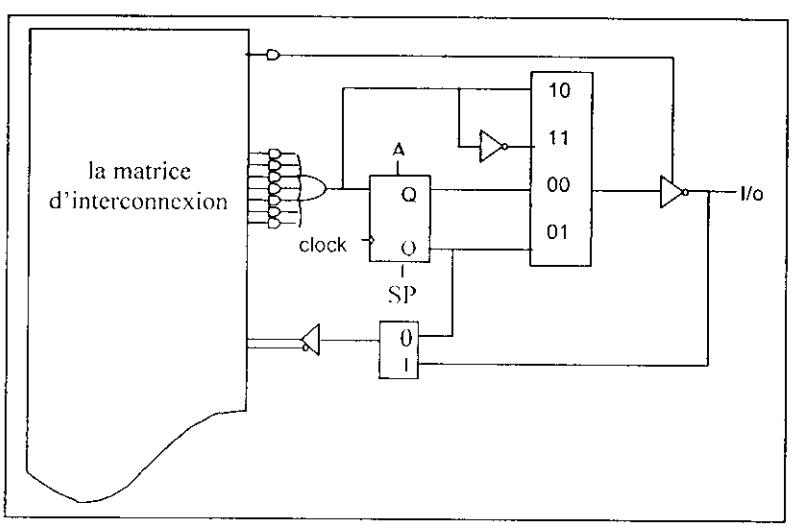


Figure II-15 : macrocellule d'un PAL versatile

V-1-4- Les prédiffusés

Les réseaux prédiffusés sont des circuits partiellement préfabriqués. L'ensemble des éléments (transistors, diodes, résistances, capacités, etc.) est déjà implanté sur le circuit suivant une certaine topologie, mais les éléments ne sont pas connectés entre eux (sauf au niveau diffusion). La réalisation des connexions dans le but de définir la fonction souhaitée est la tâche du concepteur, pour cela il dispose de bibliothèques de macrocellules et d'outils logiciels d'aide à la conception. A partir de cette liste d'interconnexions (netlist) le fondeur n'aura que quelques étapes technologiques à

effectuer pour achever le circuit, c'est à dire le dépôt d'une ou plusieurs couches de métallisation.

Cette technique est intéressante sur le plan de la conception et de la fabrication, par contre elle présente l'inconvénient de ne pas permettre une optimisation en terme de densité

de composants puisque les éléments de base sont pré implantés et pas forcément utilisés et que leur positionnement a priori n'est pas forcément optimal pour le but recherché.

V-2- Les circuits FPGA

FPGA (field programmable gate arrays) se traduit en français par circuits pré diffusés programmables. Contrairement aux circuits pré diffusés conventionnels, les circuits pré diffusés programmables ne demandent par de fabrication spéciale en usine, ni de systèmes de développement coûteux. Inventés par la société Xilinx, le FPGA, dans la famille des ASICs, se situe entre les réseaux logiques programmables et les pré diffusés (figure). C'est donc un composant standard combinant la densité et les performances d'un pré diffusé avec la souplesse due à la reprogrammation des PLD. Cette configuration évite le passage chez le fondeur et tous les inconvénients qui en découlent.

Nous décrirons une famille de FPGA, celle de Xilinx que nous avons utilisée. Il va de soi qu'il existe sur le marché nombre d'autres fondeurs qui ont développé des familles concurrentes ou complémentaires, parfois moins performantes, parfois mieux adaptées. Tout dépend aussi de l'utilisation que l'on en fait.

V-2-1- L'architecture des circuits FPGA :

Elle est similaire à celle des circuits pré diffusés classiques. Bien qu'il existe actuellement plusieurs fabricants de circuits FPGA dont Xilinx et Altera sont les plus connus, et plusieurs technologies et principes organisationnels, nous illustrerons ce concept à partir des circuits Xilinx famille XC4000 que nous avons employés. Nous renvoyons le lecteur intéressé vers les sites web des constructeurs pour obtenir une documentation récente présentant les derniers développements technologiques.

. L'architecture, retenue par Xilinx, se présente sous forme de deux couches :

- Une couche appelée circuit configurable,
- Une couche réseau mémoire SRAM.

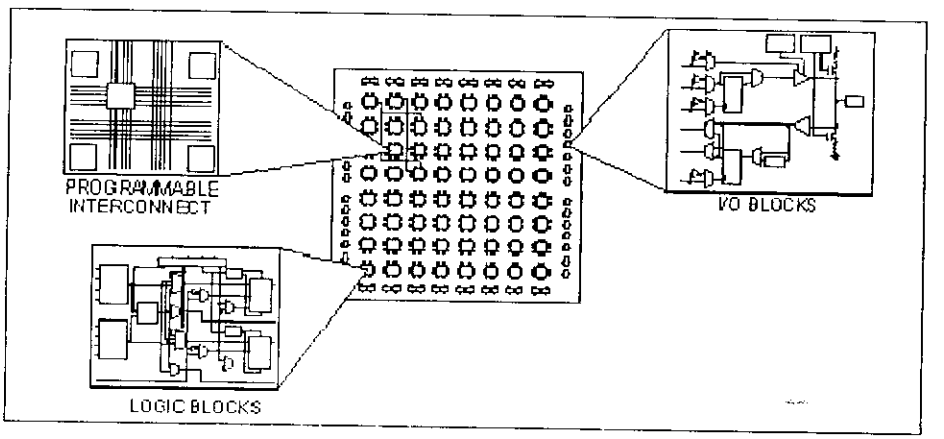


Figure II-16: Architecture interne du FPGA.

La couche dite 'circuit configurable' est constituée d'une matrice de blocs logiques configurables CLB (*configurable logical block*) permettant de réaliser des fonctions combinatoires et des fonctions séquentielles. Tout autour de ces blocs logiques configurables, nous trouvons des blocs entrées/sorties IOB (*input-output block*) dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs.

La programmation du circuit FPGA appelé aussi LCA (logic cells arrays) consistera par le biais de l'application d'un potentiel adéquat sur la grille de certains transistors à effet de

champ à interconnecter les éléments des CLB et des IOB afin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM. Les interconnexions entre les CLBs et IOBs se fait par des segment métalliques disposés en lignes et en colonnes entre les blocs, à l'intersection colonnes et lignes on trouve des matrices dites de commutation.

La configuration du circuit est mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de charger la SRAM interne à partir de la ROM. Ainsi on conçoit aisément qu'un même circuit puisse être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive. On voit tout le parti que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point.

Une erreur n'est pas rédhibitoire, mais peut aisément être réparée. La mise au point d'une configuration s'effectue en deux temps: une première étape purement logicielle va consister à dessiner puis simuler logiquement le circuit fini, puis lorsque cette étape sera terminée on effectuera une simulation matérielle en configurant un circuit réel et l'on pourra alors vérifier si le fonctionnement réel correspond bien à l'attente du concepteur, et si besoin est identifier les anomalies liées généralement à des temps de transit réels légèrement différents de ceux supposés lors de la simulation logicielle ce qui peut conduire à des états instables voire même erronés.[5]

a- Les CLB (configurable logic bloc)

Les blocs logiques configurables sont les éléments déterminants des performances du FPGA. Chaque bloc est composé d'un bloc de logique combinatoire composé de deux générateurs de fonctions à quatre entrées et d'un bloc de mémorisation synchronisation composé de deux bascules D. Quatre autres entrées permettent d'effectuer les connexions internes entre les différents éléments du CLB. La figure II-7 nous montre le schéma d'un CLB.

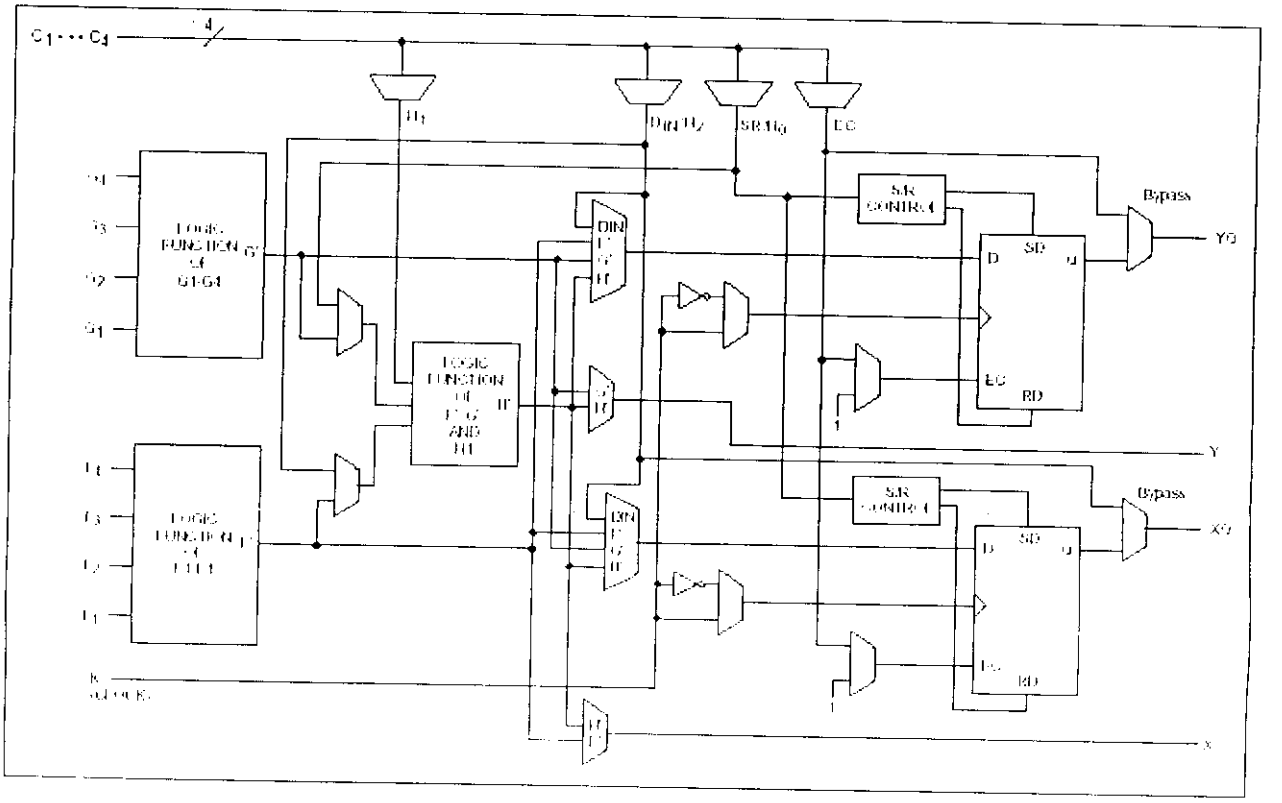


Figure II-17 : Cellules logiques (CLB)

Voyons d'abord le bloc logique combinatoire qui possède deux générateurs de fonctions F' et G' à quatre entrées indépendantes (F1...F4, G1...G4), lesquelles offrent aux concepteurs une flexibilité de développement importante car la majorité des fonctions aléatoires à concevoir n'excède pas quatre variables. Les deux fonctions sont générées à partir d'une table de vérité câblée inscrite dans une zone mémoire, rendant ainsi les délais de propagation pour chaque générateur de fonction indépendants de celle à réaliser. Une troisième fonction H' est réalisée à partir des sorties F' et G' et d'une troisième variable d'entrée H1 sortant d'un bloc composé de quatre signaux de contrôle H1, Din, S/R, Ec. Les signaux des générateurs de fonction peuvent sortir du CLB, soit par la sortie X, pour les fonctions F' et G', soit Y pour les fonctions G' et H'. Ainsi un CLB peut être utilisé pour réaliser deux fonctions indépendantes à quatre entrées indépendantes ou une seule fonction à cinq variables ou deux fonctions, une à quatre variables et une autre à cinq variables.

L'intégration de fonctions à nombreuses variables diminue le nombre de CLB nécessaires, les délais de propagation des signaux et par conséquent augmente la densité et la vitesse du circuit. Les sorties de ces blocs logiques peuvent être appliquées à des bascules au nombre de deux ou directement à la sortie du CLB (sorties X et Y). Chaque bascule présente deux modes de fonctionnement : Un mode 'flip-flop' avec comme donnée à mémoriser, soit l'une des fonctions F', G', H' soit l'entrée directe DIN. La donnée peut être mémorisée sur un front montant ou

descendant de l'horloge (CLK). Les sorties de ces deux bascules correspondent aux sorties du CLB XQ et YQ. Un mode dit de " verrouillage " exploite une entrée S/R qui peut être programmée soit en mode SET, mise à 1 de la bascule, soit en Reset, mise à zéro de la bascule. Ces deux entrées coexistent avec une autre entrée laquelle n'est pas représentée sur la figure II-17 appelée le global Set/Reset. Cette entrée initialise le circuit FPGA à chaque mise sous tension, à chaque configuration, en commandant toutes les bascules au même instant soit à '1' , soit à '0'. Elle agit également lors d'un niveau actif sur le fil RESET lequel peut être connecté à n'importe quelle entrée du circuit FPGA.

Un mode optionnel des CLB est la configuration en mémoire RAM de 16x2bits ou 32x1bit. Les entrées F1 à F4 et G1 à G4 deviennent des lignes d'adresses sélectionnant une cellule mémoire particulière. La fonctionnalité des signaux de contrôle est modifiée dans cette configuration, les lignes H1, DIN et S/R deviennent respectivement les deux données D0, D1 (RAM 16x2bits) d'entrée et le signal de validation d'écriture WE. Le contenu de la cellule mémoire (D0 et D1) est accessible aux sorties des générateurs de fonctions F' et G'. Ces données peuvent sortir du CLB à travers ses sorties X et Y ou alors en passant par les deux bascules.

b-Les IOB (input output bloc)

La figure II-18 présente la structure de ce bloc. Ces blocs entrée/sortie permettent l'interface entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant. Ils sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).

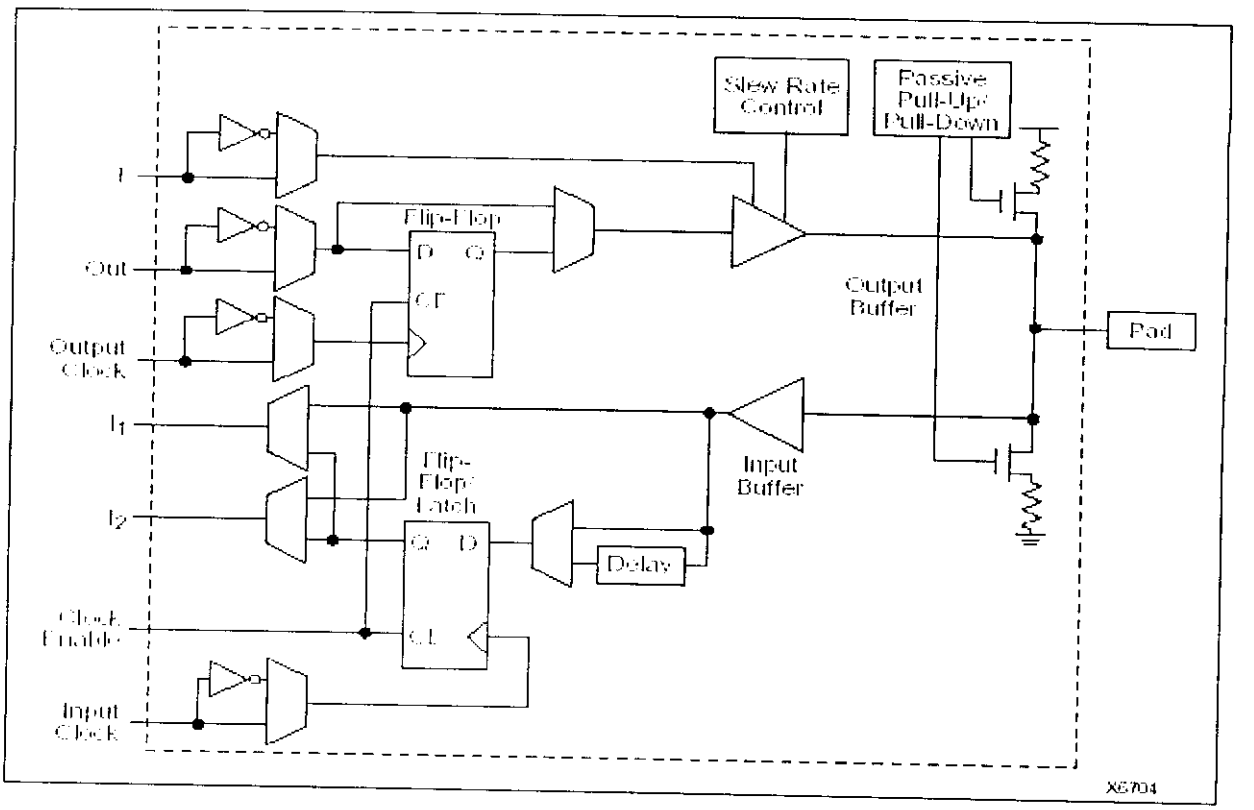


Figure II-18 : Input Output Block (IOB)

Configuration en entrée

Premièrement, le signal d'entrée traverse un buffer qui selon sa programmation peut détecter soit des seuils TTL ou soit des seuils CMOS. Il peut être routé directement sur une entrée directe de la logique du circuit FPGA ou sur une entrée synchronisée. Cette synchronisation est réalisée à l'aide d'une bascule de type D, le changement d'état peut se faire sur un front montant ou descendant. De plus, cette entrée peut être retardée de quelques nanosecondes pour compenser le retard pris par le signal d'horloge lors de son passage par l'amplificateur. Le choix de la configuration de l'entrée s'effectue grâce à un multiplexeur (program controlled multiplexer). Un bit positionné dans une case mémoire commande ce dernier.

Configuration en sortie

Nous distinguons les posibilités suivantes :

- inversion ou non du signal avant son application à l'IOB,
- synchronisation du signal sur des fronts montants ou descendants d'horloge,
- mise en place d'un « pull-up « ou « pull-down « dans le but de limiter la consommation des entrées sorties inutilisées,
- signaux en logique trois états ou deux états. Le contrôle de mise en haute impédance et la réalisation des lignes bidirectionnelles sont commandés par le signal de commande Out Enable lequel peut être inversé ou non. Chaque sortie peut délivrer un courant de 12mA. Ainsi toutes ces possibilités permettent au concepteur de connecter au mieux une architecture avec les périphériques extérieurs.

c- Les différents types d'interconnexions

Les connexions internes dans les circuits FPGA sont composées de segments métallisés. Parallèlement à ces lignes, nous trouvons des matrices programmables réparties sur la totalité du circuit, horizontalement et verticalement entre les divers CLB. Elles permettent les connexions entre les diverses lignes, celles-ci sont assurées par des transistors MOS dont l'état est contrôlé par des cellules de mémoire vive ou RAM. Le rôle de ces interconnexions est de relier avec un maximum d'efficacité les blocs logiques et les entrées/sorties afin que le taux d'utilisation dans un circuit donné soit le plus élevé possible. Pour parvenir à cet objectif, Xilinx propose trois sortes d'interconnexions selon la longueur et la destination des liaisons. Nous disposons :

- d'interconnexions à usage général,
- d'interconnexions directes,
- de longues lignes.

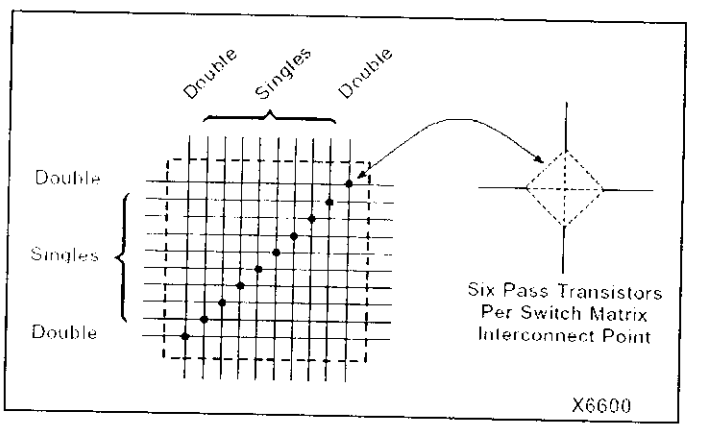


Figure II-19 : détail d'une matrice de commutation

On distingue cinq types de lignes selon la longueur relative : simple-longueur, double-longueur, quadruple et lignes octales (XC4000X seulement), et de longues lignes.

1- Les interconnexions à usage général

Ce système fonctionne en une grille de cinq segments métalliques verticaux et quatre segments horizontaux positionnés entre les rangées et les colonnes de CLB et de l'IOB.

Des aiguilleurs appelés aussi matrices de commutation sont situés à chaque intersection. Leur rôle est de raccorder les segments entre eux selon diverses configurations, ils assurent ainsi la communication des signaux d'une voie sur l'autre. Ces interconnexions sont utilisées pour relier un CLB à n'importe quel autre. Pour éviter que les signaux traversant les grandes lignes ne soient affaiblies, nous trouvons généralement des buffers implantés en haut et à droite de chaque matrice de commutation.

2- Les interconnexions directes

Ces interconnexions permettent l'établissement de liaisons entre les CLB et les IOB avec un maximum d'efficacité en terme de vitesse et d'occupation du circuit. De plus, il est possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre. Pour chaque bloc logique configurable, la sortie X peut être connectée directement aux entrées C ou D du CLB situé au-dessus et les entrées A ou B du CLB situé au-dessous. Quant à la sortie Y, elle peut être connectée à l'entrée B du CLB placé immédiatement à sa droite. Pour chaque

bloc logique adjacent à un bloc entrée/sortie, les connexions sont possibles avec les entrées I ou les sorties O suivant leur position sur le circuit.

3-Les longues lignes

Les longues lignes sont de longs segments métallisés parcourant toute la longueur et la largeur du composant, elles permettent éventuellement de transmettre avec un minimum de retard les signaux entre les différents éléments dans le but d'assurer un synchronisme aussi parfait que possible. De plus, ces longues lignes permettent d'éviter la multiplicité des points d'interconnexion.

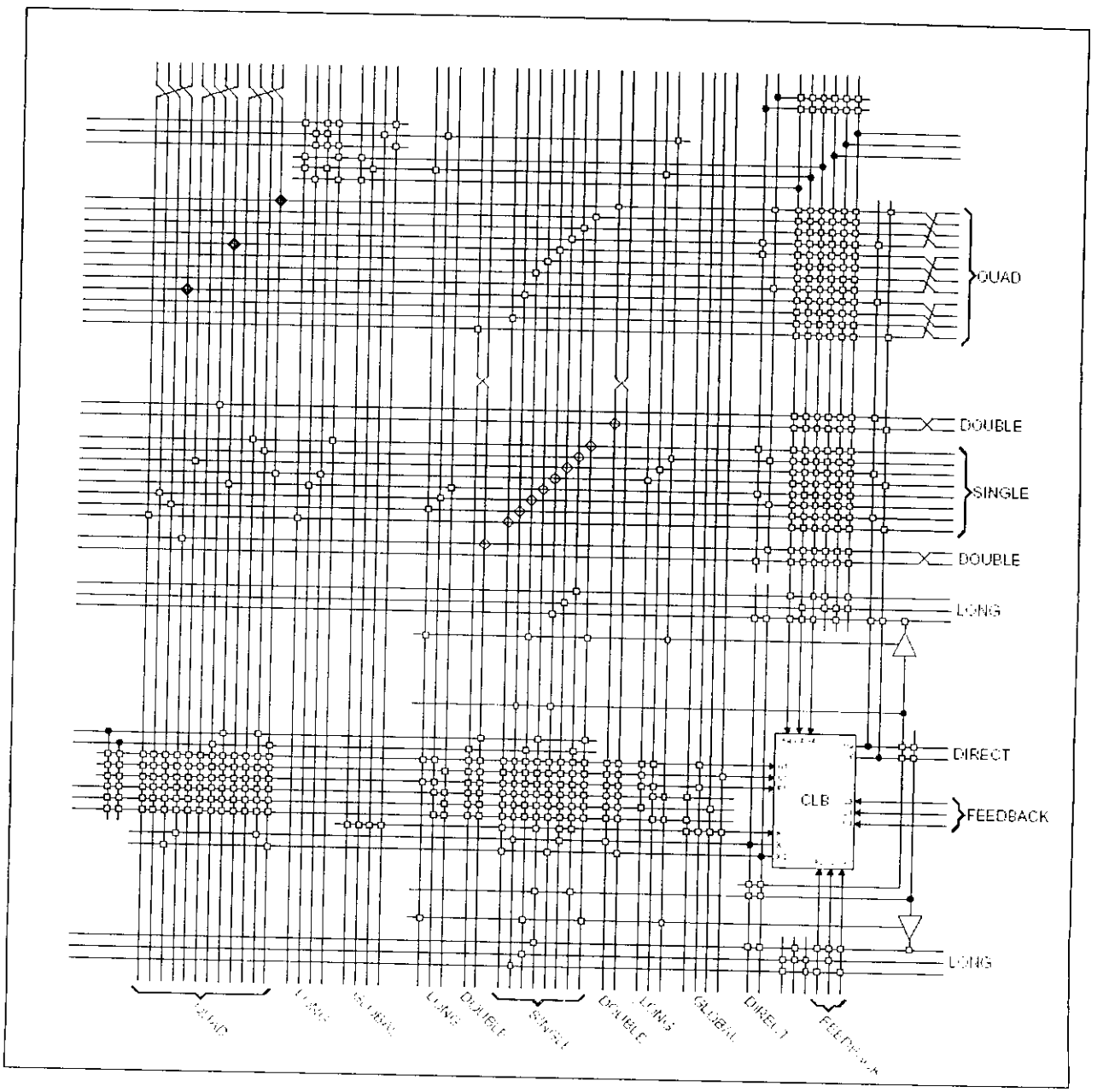


Figure II-21 : les différentes lignes d'interconnexion

4-Performances des interconnexions

Les performances des interconnexions dépendent du type de connexions utilisées. Pour les interconnexions à usage général, les délais générés dépendent du nombre de

segments et de la quantité d'aiguilleurs employés. Le délai de propagation de signaux utilisant les connexions directes est minimum pour une connexion de bloc à bloc. Quant aux segments utilisés pour les longues lignes, ils possèdent une faible résistance mais une capacité importante. De plus, si on utilise un aiguilleur, sa résistance s'ajoute à celle existante. [ASIC - FPGA –1]

d-Oscillateur interne

L'oscillateur est employé pour synchroniser power_on et time-out pour vider la mémoire lors de la configuration et comme source de CCLK (CLK de Configuration) dans le modes maître de configuration. L'oscillateur fonctionne à une fréquence de 8 MHz qui change avec le processus, Vcc, et la température. La fréquence de sortie varie entre 4 et 10 MHz.

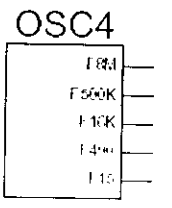


Figure II.21: symbole d'oscillateur de la série xc4000

L'utilisation de l'oscillateur est possible après configuration. N'importe « quels deux » des quatre sortie d'un diviseur intégré sont également disponibles. Ces sorties sont au quatrième , neuvième, quatorzième et dix-neuvième bit du diviseur. Par conséquent, si le rendement primaire d'oscillateur fonctionne au nominal 8 MHz, l'utilisateur a accès à une horloge de 8 MHz, en plus n'importe des autres fréquences 500 kHz, 16kHz, 490Hz et 15Hz. Ces signaux peuvent être utilisés en laçant l'OSC4 élément de bibliothèque dans un schéma ou en code de HDL. L'oscillateur est automatiquement neutralisé après configuration si le symbole OSC4 n'est pas employé dans la conception.[6]

V-2-2-Technique de programmation des FPGA

M0	M1	M2	Mode sélectionné	CCLK
0	0	0	maître série	Sortie
0	0	1	maître parallèle bas	Sortie
1	1	0	maître parallèle haut	Sortie
1	0	1	Périphérique Asynchrone	Sortie
0	1	1	Périphérique Synchrone	Entrée
1	1	1	esclave série	Entrée

Figure II-22 : modes de configurations

Les circuits FPGA ne possèdent pas de programme résident. A chaque mise sous tension, il est nécessaire de les configurer. Leur configuration permet d'établir des interconnexions entre les CLB et IOB. Pour cela, ils disposent d'une RAM interne dans laquelle sera écrit le fichier de configuration. Le format des données du fichier de configuration est produit automatiquement par le logiciel de développement sous forme d'un ensemble de bits organisés en champs de données. Le FPGA dispose de quatre modes de chargement et de trois broches M0, M1, M2 lesquelles définissent les différents modes. Ces modes définissent les différentes méthodes pour envoyer le fichier de configuration vers le circuit FPGA, selon deux approches complémentaires :

- configuration automatique, le circuit FPGA est autonome,
- Configuration externe, l'intervention d'un opérateur est nécessaire.

Mode maître série

Le mode maître série utilise une mémoire à accès série de type registre à décalage. Le programme est préalablement chargé par le système de développement utilisé pour le circuit FPGA. Le FPGA génère tous les signaux de dialogue nécessaires pour la copie du contenu de la PROM dans sa RAM interne, lorsque la copie est terminée, il bascule le signal DONE pour le signaler au circuit. Comme nous pouvons le remarquer sur la figure, une seule PROM peut configurer plusieurs circuits FPGA avec la même configuration ou plusieurs PROM peuvent configurer plusieurs FPGA en chaîne où le premier des circuits FPGA est le maître et génère l'horloge. Les données en provenance des PROM sont envoyées aux autres circuits FPGA par la sortie DOUT de ce premier (figure II-23).

On dispose aussi d'un mode maître-parallèle où le FPGA est relié en parallèle à une EPROM classique, de même qu'un mode passif type périphérique dans lequel le FPGA est considéré comme un périphérique de µP et peut-être configuré à partir de celui-ci

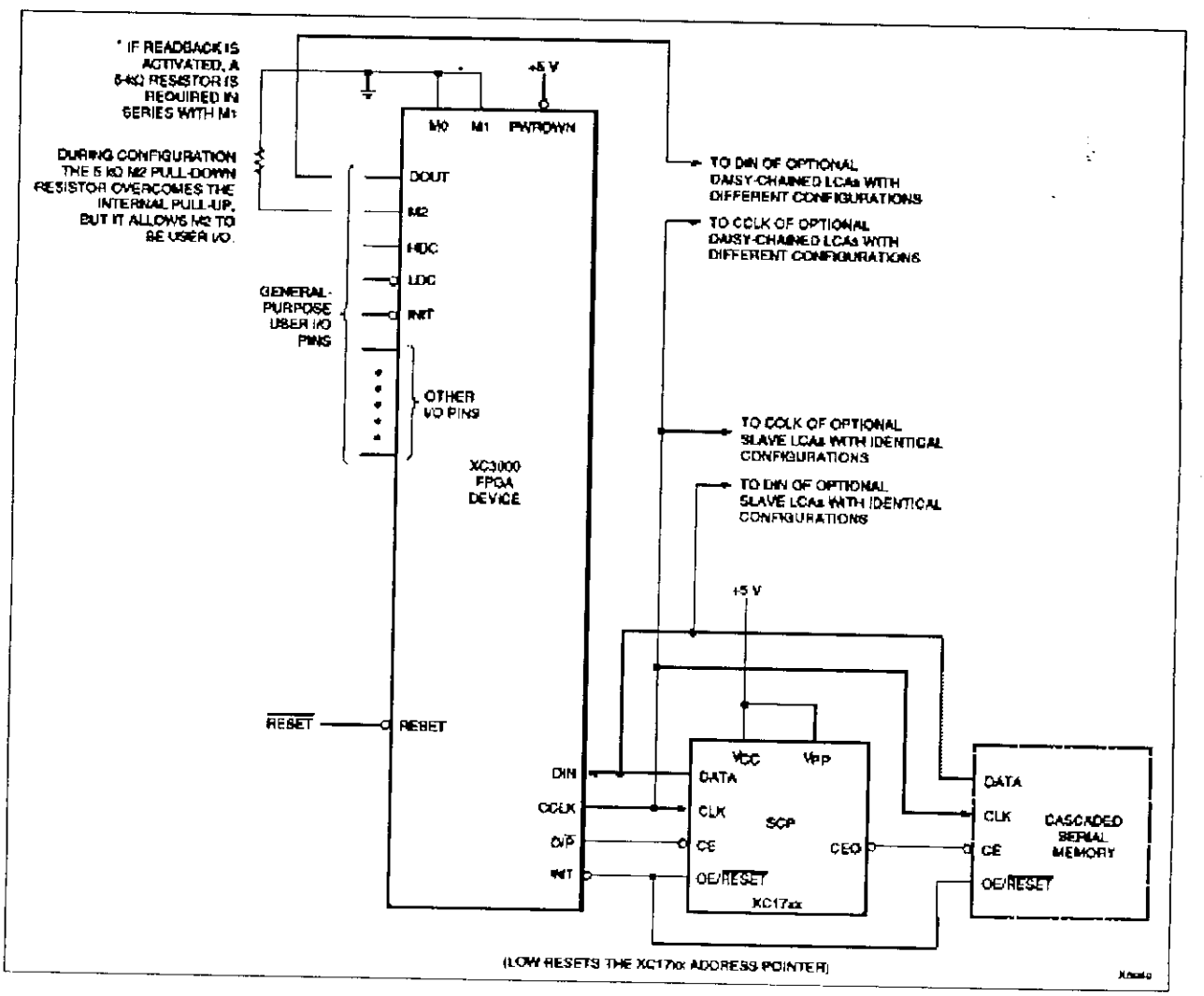


Figure II-23 : Schéma de fonctionnel en mode maître série

Mode esclave

Dans ce mode, le programme de configuration peut être envoyé à partir d'un PC, d'une station de travail ou à partir d'un autre circuit FPGA. Le circuit FPGA maître peut être interfacé à une mémoire en mode parallèle ou série sans apporter aucune modification au niveau du câblage des circuits FPGA esclaves. C'est souvent le mode exploité pour la mise au point d'une configuration. [5]

V-2-3- Méthodologie de conception

Nous présentons ici la méthode usuellement appliquée pour le développement de circuit haute densité. Pour les PLDs et les CPLDs de faible densité, les étapes sont généralement moins nombreuses. On distingue pour les FPGAs la phase de placement et la phase de routage. Pour les CPLD une seule étape est utilisée. La structure des CPLDs garantissant l'ensemble des interconnections nécessaires et la prévisibilité des délais ne nécessitant pas de phase d'optimisation du routage. le travail principal de l'outil consiste à choisir les blocs logiques(ce qui correspond au placement dans les FPGAs)

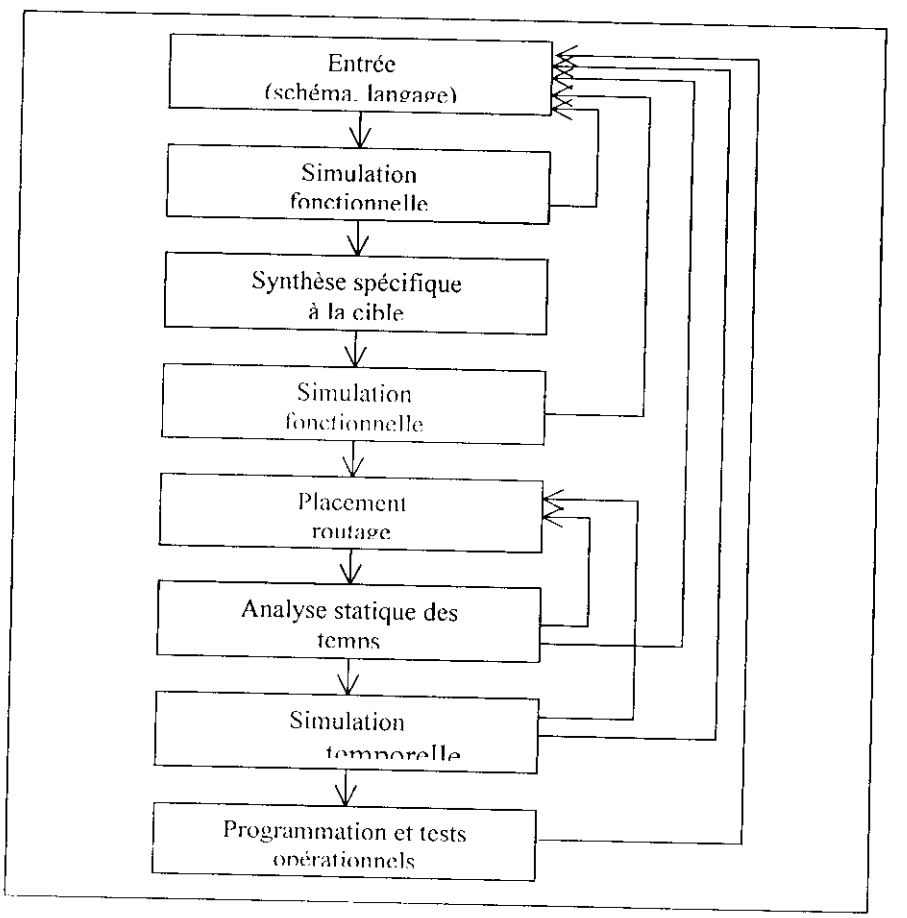


Figure II-24 : Phases successives d'un développement.

La figure II-24 décrit les phases successives d'un développement. La simulation fonctionnelle permet de tester la validité de la conception. Aucune vérification temporelle n'est ici effectuée. Cette première phase de simulation à un double intérêt, elle permet de valider fonctionnellement la description (entrée schématique ou syntaxique) en s'affranchissant des étapes parfois longues de placement et de routage. Elle permet de valider

(fonctionnellement) des sous-ensembles de la conception au fur et à mesure de l'avancement des travaux. Une seconde phase de simulation fonctionnelle peut être envisagée après la phase de synthèse. La phase de synthèse consiste à mettre à plat tout la schématique, à synthétiser la description syntaxique comportementale, à réduire les équations logiques et à les optimiser en fonction de composant cible. Cette seconde phase de simulation est optionnelle. Elle permet essentiellement de valider le travail de synthétiseur et de mettre en évidence des problèmes éventuels. A ce stade de la conception, la manière dont les ressources logiques du composant vont être utilisées est définie. La 'netlist' décrit les interconnections entre les éléments logiques de composant. La simulation peut être considérée comme partiellement temporelle mais les délais dus au routage (qui dépend lui même du placement) ne sont pas encore connus. Après la phase de placement routage, deux procédés de vérification temporelle peuvent être envisagés. Certaines contraintes spécifiées dans un fichier permettront de vérifier la vitesse de fonctionnement globale. Les mêmes vecteurs de test défini pour la simulation fonctionnelle, pourront être utilisés cette fois ci pour la simulation temporelle. Si la simulation temporelle met en évidence un dysfonctionnement du à des délais trop grands ou si les contraintes spécifiées pour l'analyse statique ne sont pas respectées, il est alors nécessaire d'optimiser soit la conception soit la phase de placement /routage.
L'outil utilisé dans noter projet et le XILINX FOUNDATION SERIES (annexe A).[7]

Conclusion

Dans ce chapitre, nous avons présenté les circuits de type FPGA et la méthode de conception associé, dont nous précisons ici les principaux avantages :

- Le premier argument est la souplesse de programmation qui permet l'emploi conjoint d'outils de schématique aussi bien que l'exploitation d'un langage de haut niveau tel VHDL. Ce qui permet de multiplier les essais, d'optimiser de diverses manières l'architecture développée, de vérifier à divers niveaux de simulation la fonctionnalité de cette architecture.
- Le second argument est évidemment la nouvelle possibilité de reconfiguration dynamique partielle ou totale d'un circuit ce qui permet d'une part, une meilleure exploitation du composant, une réduction de surface de silicium employé et donc du coût, et d'autre part, une évolutivité assurant la possibilité de couvrir à terme des besoins nouveaux sans nécessairement repenser l'architecture dans sa totalité. L'un des points forts de la reconfiguration dynamique est effectivement de permettre de reconfigurer en temps réel en quelques microsecondes tout ou partie du circuit, c'est à dire de permettre de modifier la fonctionnalité d'un circuit en temps quasi réel. Ainsi le même CLB pourra à un instant donné être intégré dans un processus de filtrage numérique d'un signal et l'instant d'après être utilisé pour gérer une alarme. On dispose donc quasiment de la souplesse d'un système informatique qui peut exploiter successivement des programmes différents, mais avec la différence fondamentale qu'ici il ne s'agit pas de logiciel mais de configuration matérielle, ce qui est infiniment plus puissant.
- Notons enfin que ces circuits n'ont pas vocation à concurrencer les super calculateurs, mais plutôt à offrir une alternative en fonction de critères comme l'encombrement, les performances et le prix, et sont de ce fait bien adaptés à des applications de qualité dans le domaine des systèmes ambulatoires ou nomades.
- Enfin il semble que de plus en plus fréquemment les concepteurs de circuits ASIC préfèrent passer par l'étape intermédiaire d'un FPGA ce qui est moins risqué économiquement, puis une fois que le modèle FPGA est au point, il est alors relativement aisé de le retranscrire dans une architecture de type prédiffusé ou précaractérisés. Ce que tous les fondeurs de silicium savent effectivement faire pour en faire un circuit réellement personnalisé et confidentiel. Le FPGA n'étant évidemment pas un circuit très sécurisé sur le plan de la confidentialité puisqu'il suffit d'analyser le contenu de la ROM associée pour remonter à la schématique imaginée.

Mais les circuits FPGA présentent un inconvénient majeur, c'est la confidentialité et la sécurité en terme de propriété industrielle : il suffit d'analyser le contenu de ROM de configuration dans le mode configuration esclave, ou le fichier *bitstream* dans le mode périphérique pour voir la configuration exacte de l'FPGA ainsi son circuit.

Conceptions et implémentations

Plan

I- Conception et implémentation de la version CISC_1

II- Conception et implémentation de la version CISC_2

III- Conception et implémentation de la version CISC_3

IV- Evaluation des performances des trois implémentations

Introduction

Dans ce chapitre nous allons appliquer tous ce qu'on a apprit sur l'architecture des microprocesseurs et le techniques de conception. Nous allons concevoir trois microprocesseurs de type CISC.

Le premier , appelé CISC_1, est un modèle de base des microprocesseurs avec un minimum de fonctions. Le second , CISC_2 introduit la couche microprogrammée. L'approche des microprocesseurs dits parallèles a été abordé par la conception du microprocesseur CISC_3 à structure pipeline.

I- Conception et implémentation de la version CISC_1

Comme premier exemple d'implémentation on a choisie l'exemple traité dans [9]avec des adaptations. Il contient les fonctions élémentaires d'un microprocesseur à savoir les opérations arithmétiques et logiques, transfert mémoire et les branchements.

I-1- Le modèle de programmation

Les registres accessibles au programmeur :

- acc 4 bits
- pc 8 bits
- etat (CZ) 2 bits

Format de l'instruction :

Rupture de séquence

Autres instruction

Jeu d'instructions

Le jeu qu'on s'est proposé d'implémenter permet :

- L'addition et le AND logique bit par bit.
- *Les adressages* immédiat et direct.
- *Les sauts* conditionnels et inconditionnels.

JUMP	$(0XXXXXXXXX)_2$	PC<= $(0XXXXXXX)_2$
LOAD_D	CX	ACC <= M(X)
ADD_D	DX	ACC <= ACC+M(X)
STOR_I	BX	M(X) <= ACC
LOAD_I	9X	ACC<= X
ADD_I	AX	ACC<= ACC + X
AND_D	FX	ACC<=ACC AND M(X)
AND_I	EX	ACC<= ACC AND X
CLEAR_C	80	C<= 0
SET_C	81	C<=1
CLEAR_Z	84	Z<=0
SET_Z	85	Z<= 1
SKIP_C	82	IF C=1 THEN PC<=PC+1
SKIP_Z	83	IF Z=1 THEN PC<=PC+1

LE diagramme fonctionnelle

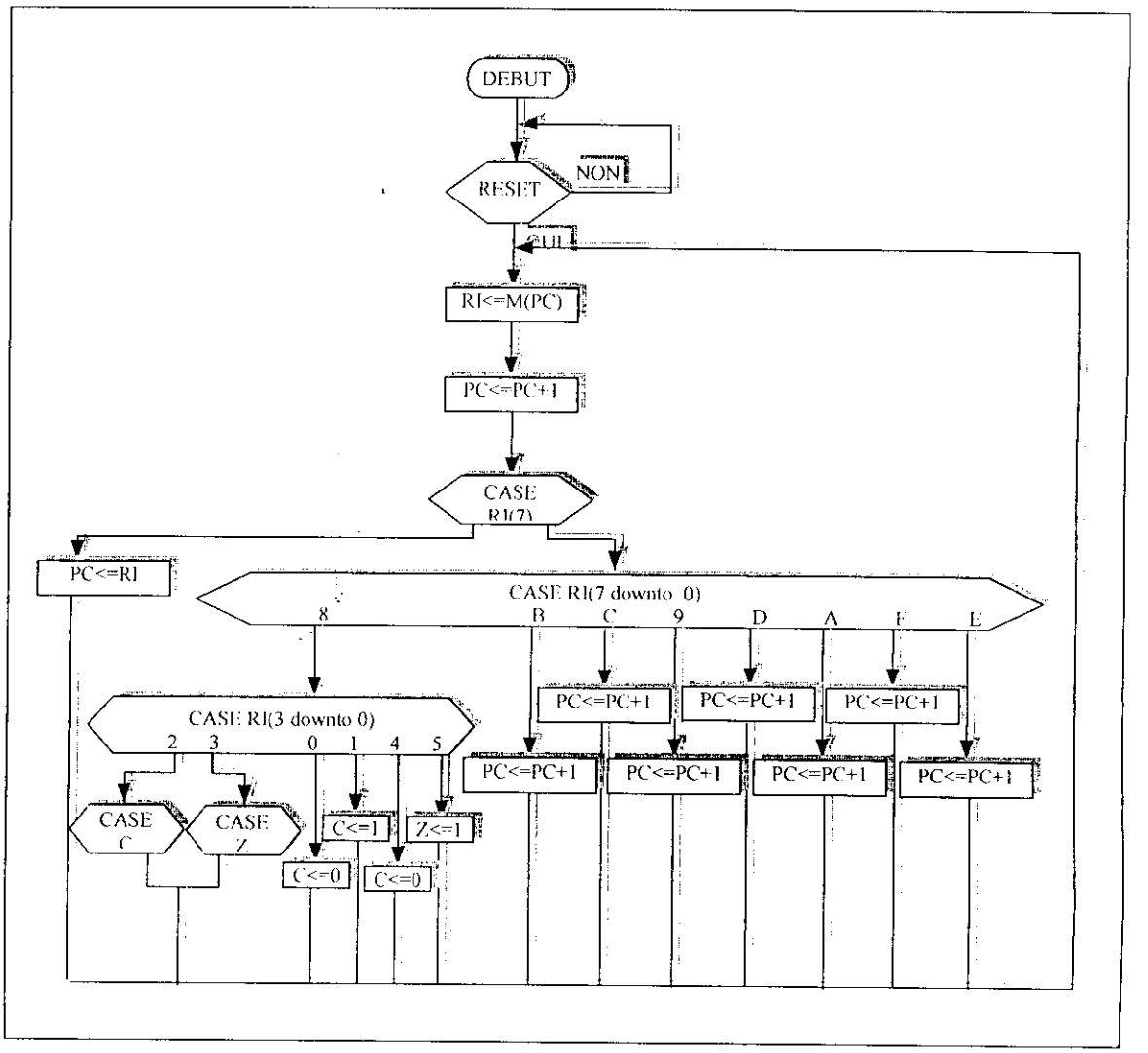


Figure III-1 : Organigramme fonctionnel du CISC_1

I-2 La description structurelle

Le microprocesseur est constitué de deux éléments : le chemin de données et l'unité de commande .

Construction du chemin de données

Le chemin de donnée est l'ensemble des registres ,des opérateurs et des bus qui réalisent les opérations évoqués dans l'organigramme fonctionnel.
Les registres (Figure III-2):

On distingue en premier, ceux associés au modèle de programmation :

ACC	4 Bits
PC	8 Bits
Z-C	2 Bits

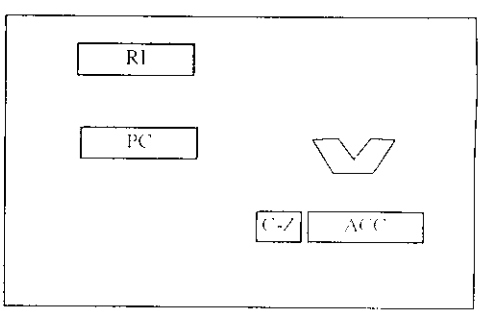


Figure III-2 : les registres

En suite les registres indispensables à la mise en œuvre du modèle de fonctionnement. Dans notre cas, RI sera affecté au stockage de l'instruction de largeur 8Bits

Les opérations réalisées sont l'addition et le ET logique bit par bit. Elles sont réalisées par une UAL de deux opérandes de 4Bits chacun. Le résultat est sur 4Bits avec un bit pour la retenue.

L'adresse peut être prise du compteur ordinal ou du RI dans le cas de l'adressage direct. Un multiplexeur un parmi deux est indispensable.

Pour les bus, on distingue deux types : donnée de 4Bits et adresse de 8Bits. Ils peuvent s'interconnecter. Les conflits sont évités par des buffers trois états.

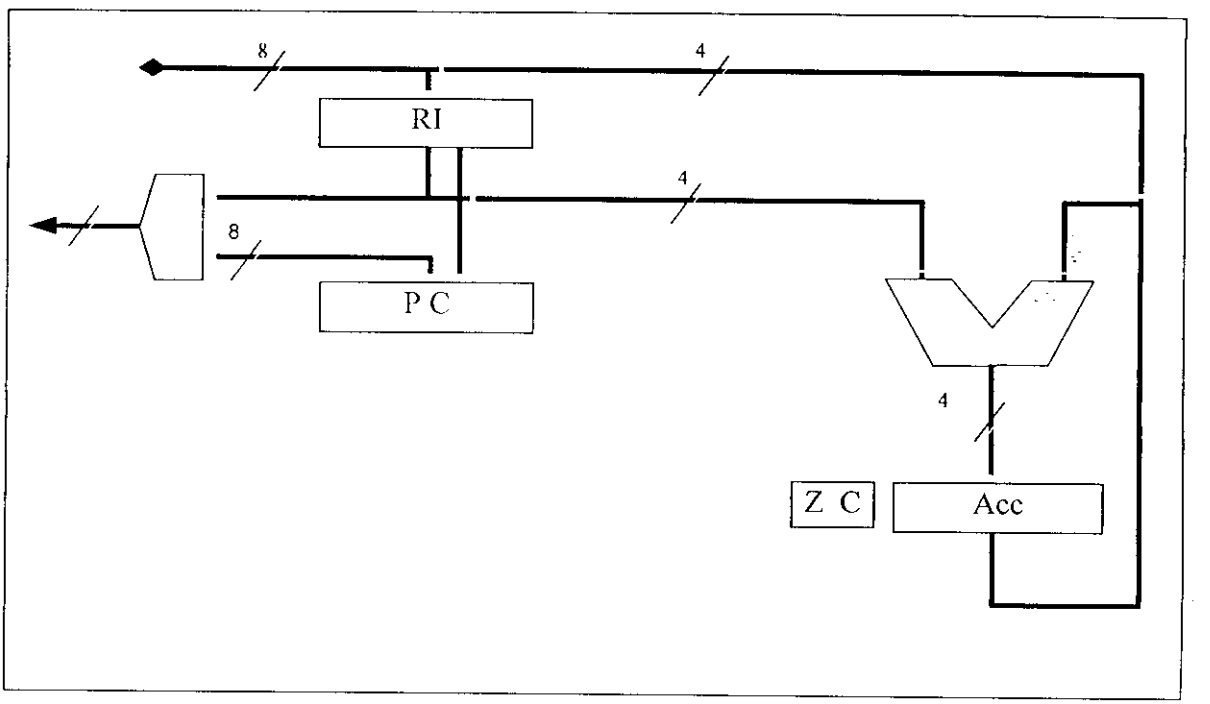


Figure III-3 : cheminement des données et des adresses

Il reste à commander ces éléments. Ceci se fait par des signaux dits de commande (figure III-3). Pour avoir tout la liste de signaux nécessaires, on doit tenir compte des caractéristiques de chaque composant.

ACC registre synchrone , modification , mise à 0 asynchrone .
CLK LD_ACC RESET

C-Z registre synchrone , modification, mise à 0 et à 1 asynchrone .
CLK LD_Z-C RESET SET

PC registre synchrone , modification , ''incrementation '', mise à 0 asynchrone .
CLK LD_PC INC RESET

RI registre synchrone , modification complete ou LSN, mise à 0 asynchrone .
CLK LD_RI LD_RI_LSN RESET

UAL 3 opérations add , and , passe , (passe c'est la sortie égale à RI_lsn)
SEL_OP (2Bits)

Multiplexeur deux entrées et une sortie .
SEL_ADDR

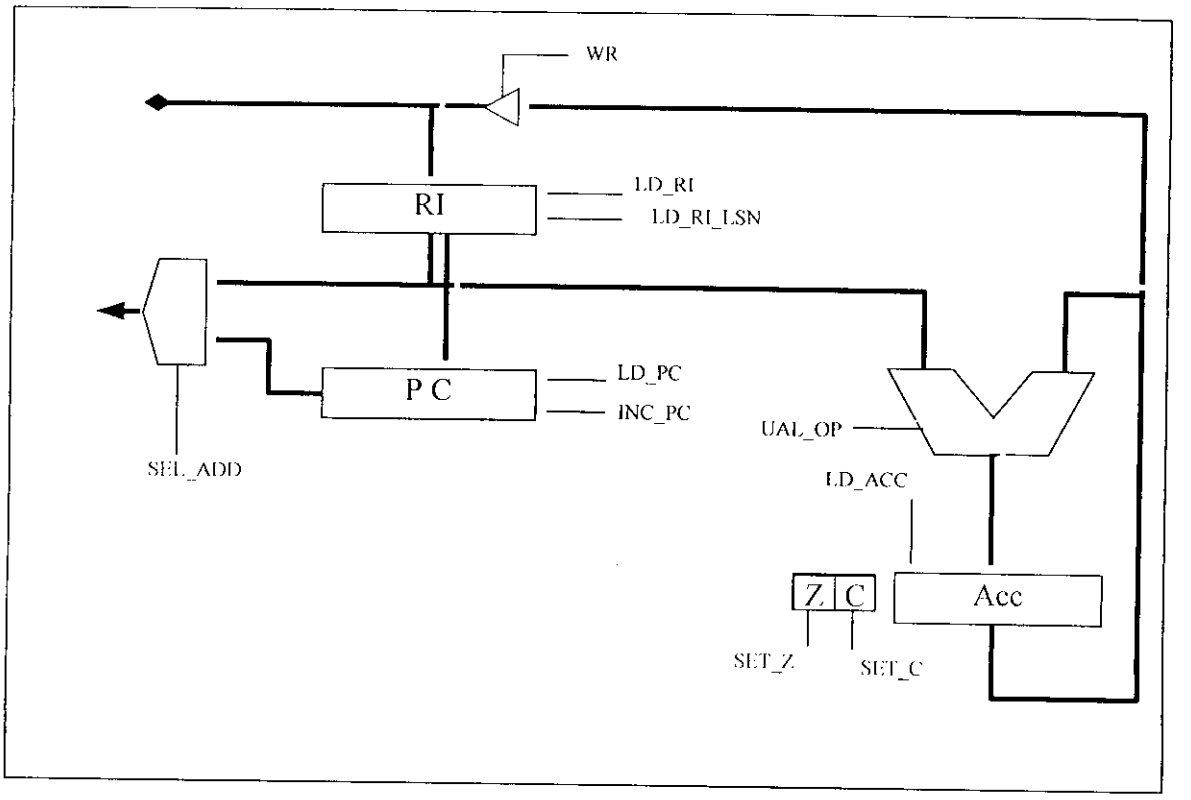


Figure III-4 : Les signaux de commande du chemin de données

L'exécution d'une instruction se fait en trois cycles : FETCH, DECOD et EXECUTE. Chaque cycle peut se faire simultanément ou en plusieurs phases .

Dans notre chemin de données (figure 3.4) , le cycle FETCH consiste à charger l'instruction dans RI et l'incrémentation du PC. Le cycle DECODE se fait simultanément (c'est un choix qu'on a fait). Le cycle EXECUTE réalise le chargement de l'ACC ou Z-C par le résultat ou le chargement du PC de l'UAL pour un adressage immédiat. Par contre pour un adressage direct, on doit d'abord recharger l'opérande dans le LSN du RI .
Le chemin de données aura donc besoin de trois horloges décalées. Dans les circuits FPGA, on a pas la notion de temps pour générer les trois phases retardées. On doit alors les générer ''artificiellement'' . Ce qui a été mis en œuvre par l'*horloge* ou distributeur de phases qu'on a conçu .

Unité de commande

Horloge : elle génère un signal ''identificateur'' de l'état FETCH et trois tops de synchronisation. Un front montant au milieu du cycle FETCH pour le chargement de RI et deux autres pendant $\overline{\text{FETCH}}$.

Le décodeur d'instruction
Il génère le séquencement des commandes du chemin de données à partir du code opération. Aussi, elle aura comme entrée :
- le registre d'instruction RI

- le signal FETCH
- le registre d'état Z-C

La présence de RI à l'entrée permet le décodage de l'instruction d'une façon purement combinatoire. Le signal FETCH indiquera le cycle FETCH ou EXE. L'unité de commande devient un circuit combinatoire. Ce qui nous permet de dire que notre ''séquenceur est câblé .

La table de vérité du décodeur d'instruction

	RD	LD_RI	LD_RI_LSN	INC_PC	SEL_ADDR	LD-ACC	WR	LD_PC	CLR-Z	CLR_C	UAL_OP	SET_C	SET_Z
FETCH	1	1		1									
LOAD_D	1		1		1	1							
ADD_D	1		1		1	1					1		
STOR_D	1		1		1		1						
LOAD_I	1				1	1							
ADD_I	1				1	1					1		
AND_D	1		1		1	1					2		
AND_I	1				1	1					1		
CLEAR_C										1			
SET_C												1	
CLEA_Z									1				
SET_Z													1
SKIP_C				1									
SKIP_Z				1									
JUMP								1					

I-2- Implémentation

La description en VHDL

Le circuit principal est décrit d'une façon structurelle. Il se compose de trois composants *horloge, chemin_de_données* et *unite_commande*, et chaque composant est décrit seul d'une façon comportementale. [annexe B].

La synthèse

Pour la synthèse on a choisis les paramètres suivants :

La cible :

XILINX - XC4005XLPC84-3

- Speed grade
- Alimentation VCC =3.3 V
- Nombre de porte approximative x 10^3
- Famille 4000 du composant
- Le composant FPGA

Speed grade: paramètre indiquant l'ordre des temps de transitions, par exemple entre le entrées du générateur de fonction F/G et les sorties X/Y. [6]

Preserve hierarchie : pour conserver la hiérarchie afin de faciliter l'extraction des signaux internes pendant la simulation.

Optimisation : vitesse et consommation.

La synthèse transforme la discrétion VHDL en circuit logique idéal sans retards. Elle permet alors une simulation fonctionnelle du circuit.

La simulation fonctionnelle

Ce test consiste à injecter les entrées nécessaires et observer les changement du modèle de programmation (RI, PC, ACC, Z et C).
Les entrées sont *reset, clk, DATA. On* a aussi visualiser les signaux de l'horloge pour séparer les phases de l'exécution. Dans les figures ci-dessous, le signal *DATA_out* représente le contenu de l'accumulateur ACC.
Les figure III-5 et la figure III-6 mettent en valeur l'exécution respectivement des instructions ADD_I 1 (A1) et LOAD_D 0; SKIP_Z; SKIP_C.

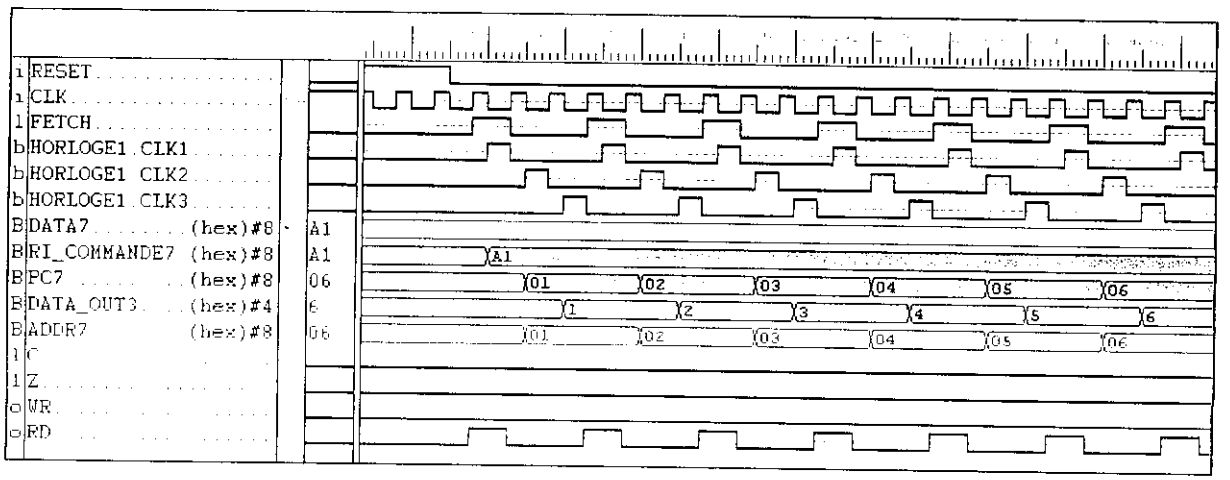


Figure III-5 : une suite de ADD_I 1 ;

On remarque sur ces simulations que les retards entre signaux sont nuls. La simulation fonctionnelle pour toutes les instructions a montré que la conception faite correspond au microprocesseur voulu.

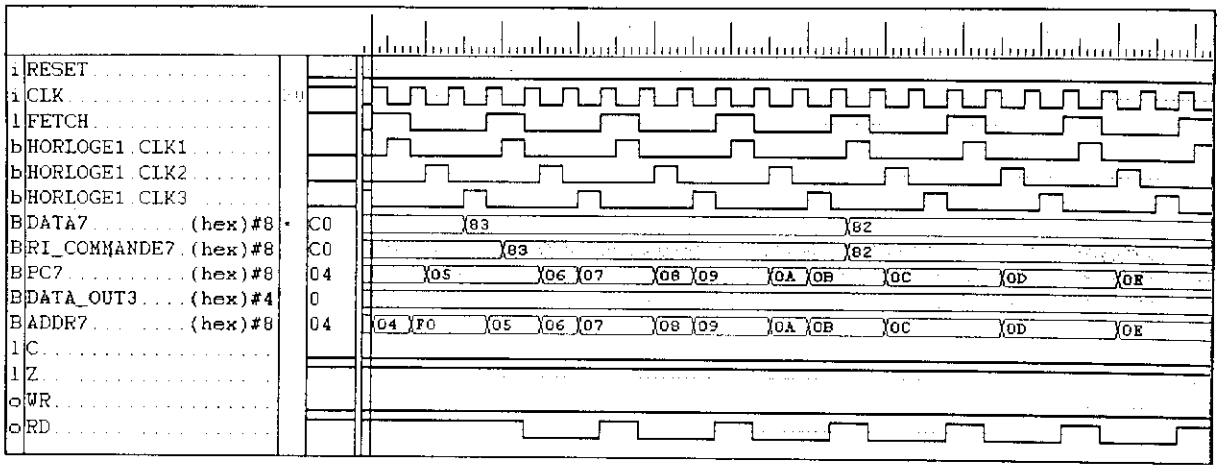


Figure III-6 : Exécution de LOAD_D 0; SKIP_Z; SKIP_C.

Simulation temporelle

Après placement et routage, l'outils de synthèse produit (par des estimation) les retards des signaux à l'intérieur du FPGA dûs aux retards de propagation à travers les composants CLB et IOB. A partir de là, il nous permet une simulation dite temporelle qui fait introduire les retards. Mais ça reste toujours une simulation car ces retards sont des estimations calculés en transposant la configuration résultante de la synthèse sur le modèle théorique du circuit cible.
Avant de lancer la simulation, on détermine les caractéristique logiques et physiques déduits de la synthèse (les retards en fait partie) dans des fichier texte sous forme de rapport.

On prend quelques exemples :

Le bilan sur les ressource utilisées

```
Number of CLBs:                37 out of   196   18%
      CLB Flip Flops:         26
      CLB Latches:             0
      4 input LUTs:           68
      3 input LUTs:           10 (4 used as route-throughs)
   Number of bonded IOBs:        20 out of    65   30%
      IOB Flops:               0
      IOB Latches:             0
   Number of clock IOB pads:      1 out of    12    8%
   Number of BUFGLSs:             4 out of     8   50%
Total equivalent gate count for design: 668
```

Bilan des ressources supprimées par simplification

```
Section 4 - Removed Logic Summary
---------------------------------
  10 block(s) removed
   1 block(s) optimized away
   2 signal(s) removed
```

Les pins de l'FPGA correspondants au signaux d'entrées-sorties

```
Pinout by Pin Name:

COMP "addr<0>" LOCATE = SITE "P45" ;
COMP "addr<1>" LOCATE = SITE "P39" ;
COMP "clk" LOCATE = SITE "P10" ;
COMP "data<0>" LOCATE = SITE "P61" ;
COMP "data<1>" LOCATE = SITE "P60" ;
COMP "data<2>" LOCATE = SITE "P66" ;
COMP "rd" LOCATE = SITE "P41" ;
COMP "reset" LOCATE = SITE "P25" ;
COMP "wr" LOCATE = SITE "P83" ;
```

Quelque signaux et retards correspondants

```
Net Delays

CLR_C
   CLR_C.Y
         2.534  chemin_de_donnees1/N545.F1
         4.438  chemin_de_donnees1/C222_N28.G1
```

lecture : le signal CLR_C provenant de la sortie Y de la CLB CLR_C prend 2.534ns pour arriver à l'entrée F1 de la CLB chemin_de_données 1/N545

```
Design statistics:
   Minimum period:  34.218ns (Maximum frequency:  29.224MHz)
   Maximum net delay:  12.306ns
```

Génération du fichier configuration « bitstream »

```
BITGEN: Xilinx Bitstream Generator M1.5.19
Copyright (c) 1995-1998 Xilinx, Inc.  All rights reserved.

Tue Jul 09 07:34:45 2002

bitgen -l -w -g ConfigRate:SLOW -g TdoPin:PULLNONE -g
M1Pin:PULLNONE -g DonePin:PULLUP -g CRC:enable -g
StartUpClk:CCLK -g SyncToDone:no -g DoneActive:C1 -g
OutputsActive:C3 -g GSRInactive:C4 -g ReadClk:CCLK -g
ReadCapture:enable -g ReadAbort:disable -g M0Pin:PULLNONE -g
M2Pin:PULLNONE cisc_1.ncd

Running DRC.
DRC detected 0 errors and 0 warnings.
Saving ll file in "cisc_1.ll".
Creating bit map...
Saving bit stream in "cisc_1.bit".
```

La figure III-7 montre la simulation temporelle de la suite d'instructions suivante ld_i 3; ld_i 0 skip_z avec une fréquence de 40 MHz. On a choisi cette suite d'instructions car elle contient les deux types d'instructions : type -1 synchronisé sur deux tops d'horloge *CLK1* et *CKL3* ; type - 2 synchronisé sur les trois tops d'horloge. On déduit que c'est le deuxième type qui est le plus sensible aux retards internes.

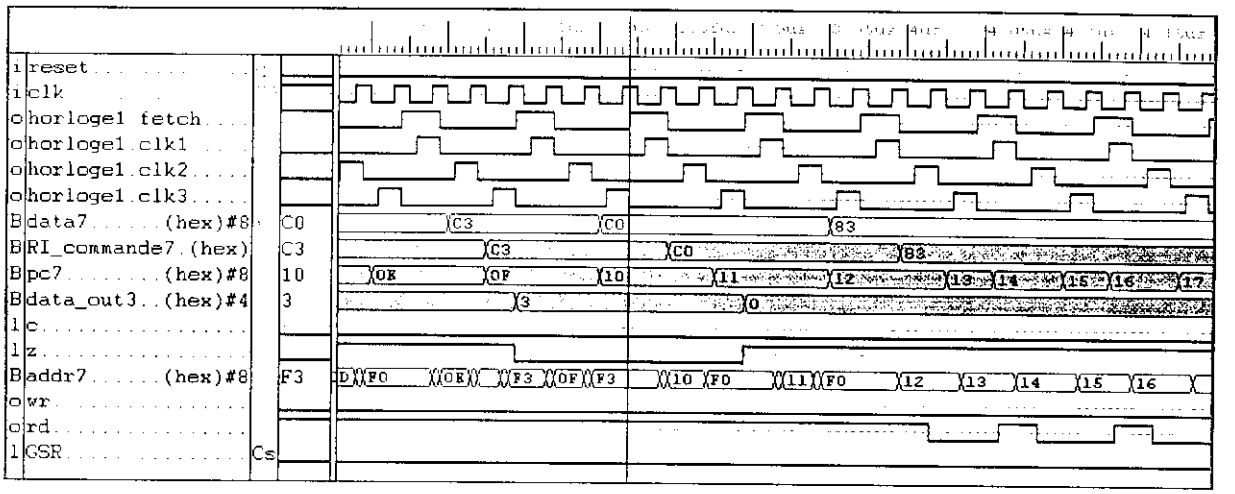


Figure III-7 : simulation temporelle

Ce qu'on remarque c'est que malgré que la fréquence est supérieure à celle estimée par l'outil de synthèse, l'exécution s'est déroulée comme prévue.

Optimisations

Comme première synthèse, on a pas posé de contraintes. C'est l'outil de synthèse qui a choisis librement tout les paramètres concernant le placement et le routage.

En imposant des contraintes concernant les BUFG (lignes d'interconnexions rapide), on a obtenu les résultats illustrés sur la table suivante :

Les signaux imposés pour les BUFG	La fréquence estimé par l'outil (MHz)
RESET, FETCH, CLK1, CLK2, CLK3, CLK_PC,CLK_LSN	21.8
RESET, FETCH, CLK1, CLK2, CLK3	23.0
RESET, CLK	27.1
RESET	30.6
AUCUN	29.2

Remarque : faire passer le *RESET* sur une ligne rapide n'améliore pas le circuit sur le plan fonctionnel par ce que la mise à zéro n'est pas synchrone.
On peut rien conclure tant qu'on ne sait pas comment l'outil de synthèse choisit la fréquence maximum de fonctionnement.

La fréquence maximum du CISC_1 :

Une chose est sûre : la première configuration des BUFG est la plus adaptée par ce qu'elle permet la synchronisation voulue dans notre structure. Etudions la version du CISC_1 résultante de la première configuration.

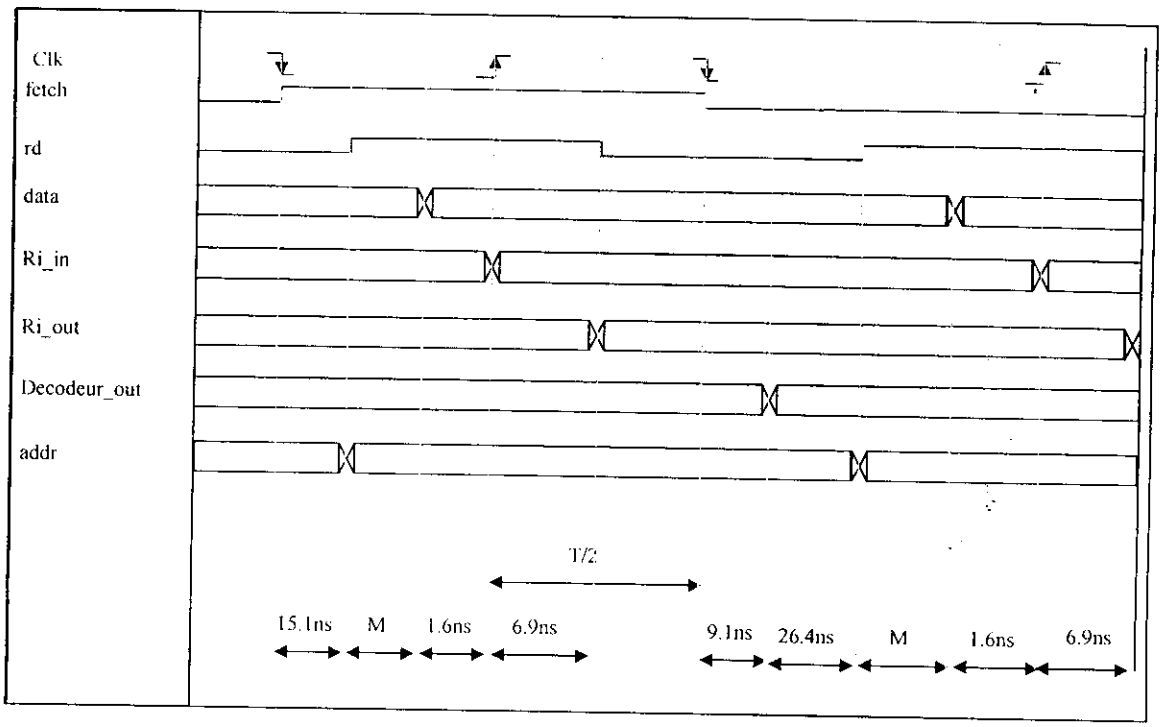


Figure III-8: les retards mesurés

Pour calculer la fréquence maximum de fonctionnement, on doit mesurer les retards entre chaque paire d'éléments qui utilisent deux tops d'horloges successives. Pour cela, on a utilisé le simulateur (figure III-8). Les résultats obtenus sont représentés sur la même figure. La mesure M précise le temps accès mémoire.

Temps entre tops d'horloges :

- T/2= 15.1+M+1.6 => T=33.4+M
- T/2=6.9 => T=13.8
- T/2=9.1+23.4+M+1.6 => T=74.2+2M

$T=T_{max}=74.2+2M.$

Ce qui donne une fréquence de 13.4MHz dans le cas d'une mémoire idéale. Cette lenteur (74.2ns) est dûe surtout au retard de l'adresse par apport au décodage (26.4ns).

Test

On a fait le teste (figure III-9) à une fréquence horloge de 13.6MHz (73ns), avec un adressage direct (premier cercle). Il a engendré les résultats prévus (deuxième cercle).

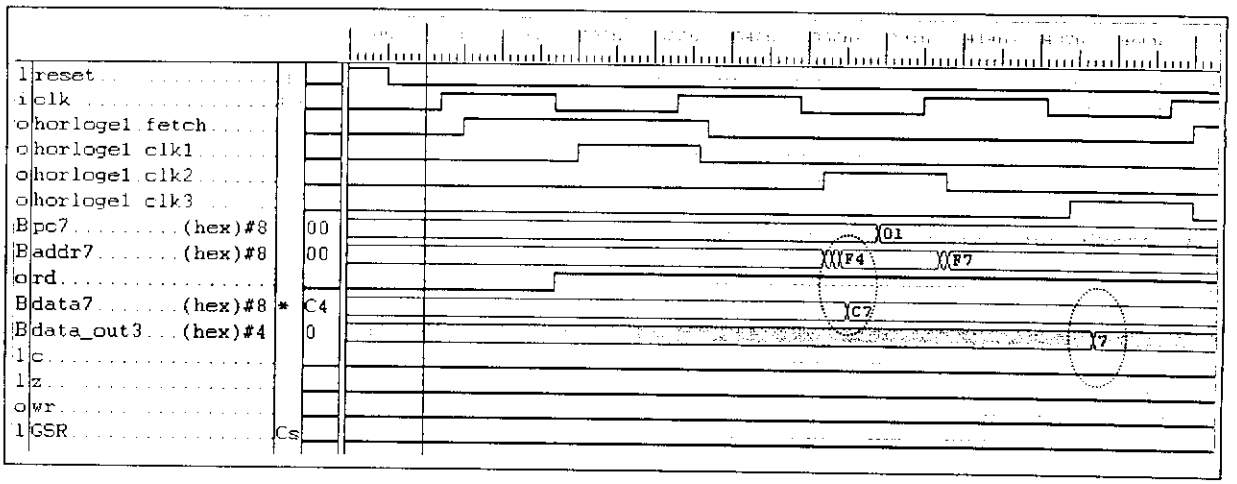


Figure III-9: Test de la fréquence maximale

II- Conception et implémentation de la version CISC_2

Cette nouvelle implémentation va nous introduire le modèle de Wilkes à travers l'introduction d'une mémoire microprogrammée . Elle s'inspire de celle Andrew Tananbaum [] auquelle on a adjoint les mécanismes de prise en charge des interruptions couplés à ceux du fenêtrage.

II-1- Le modèle de programmation

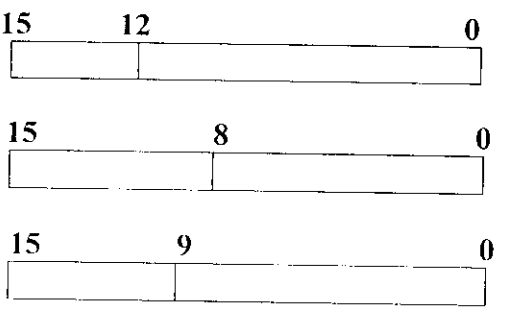
Les registres accessibles au programmeur

ACC	16 bits	
PC	16 bits	
PP	16 bits	
Etat	2 bits	(C, N)

Le format des instructions

Le format des instruction est hétérogène. Il s'est imposé de fait eu égard au modèle de microprocesseur à implémenter.

Il se présente selon les trois cas suivants :



Le premier champ correspond au champ opération. Le second est affecté à la partie opérande (donnée ou adresse)

Jeu d'instructions de la macro-machine

Trois modes d'adressage ont été associés aux instructions : direct, indirect et locale. En mode d'adressage direct, les instructions disposent d'une adresse sur 12 bits : l'adresse effective de la donnée concernée. ces instruction permettent l'accès aux variables globale du programme.

En mode d'adressage indirect , le contenu du mot mémoire référence par le champ adresse de l'instruction est chargé au préalable dans AC ; AC contient alors l'adresse de mot mémoire concernée par l'instruction. Ce mode d'adressage est fort utilisé.

Le mode d'adressage local, propre à notre microprocesseur, spécifie l'accès à une information dans la pile, relativement à la position courante du pointeur de pile pp.

Le tableau suivant illustre le jeu d'instruction de notre micro processeur. Il comprend quatre colonnes : celle la plus à gauche correspond à la microinstruction, la deuxième fournit le code symbolique ou mnémonique de l'instruction, la troisième colonne spécifie le nom de l'instruction et la dernière sa fonctionnalité décrite en langage algorithmique.

0000 XXXX XXXX XXXX	LOAD	chargement mode direct	ac :=m[X]
0001 XXXX XXXX XXXX	STOD	Rangement mode direct	m[X] :=ac
0010 XXXX XXXX XXXX	ADDD	Addition mode direct	ac :=ac+m[X]
0011 XXXX XXXX XXXX	SUBD	Soustraction modedirect	ac :=ac-m[X]
0100XXXX XXXX XXXX	JPOS	Jump si > 0	If ac >0 then co :=X
0101 XXXX XXXX XXXX	JZER	jump si = 0	If ac 0 then co:=X
0110 XXXX XXXX XXXX	JUMP	Branchement	co:=X
0111 XXXX XXXX XXXX	LOCO	Chargement constante	ac :=X (0<X<4095)
1000 XXXX XXXX XXXX	LODL	Chargement mode local	ac :=m[pp+X]
1001 XXXX XXXX XXXX	STOL	Rangement mode locale	m[X+pp]:=ac
1010 XXXX XXXX XXXX	ADDL	Addition mode local	ac:=ac+m[pp+X]
1011 XXXX XXXX XXXX	SUBL	Soustraction mode local	ac :=ac-m[pp+X]
1100 XXXX XXXX XXXX	JNEG	Jump si < 0	If ac <0 then co:=X
1101 XXXX XXXX XXXX	JNZE	Jump si = 0	If ac =0 then co :=X
1110 XXXX XXXX XXXX	CALL	Appel procédure	pp:=pp-1;m[pp]:=co; co:=X
1111 0000 0000 0000	PSHI	Push indirect	pp:=pp-1;m[pp]:=m[ac]
1111 0010 0000 0000	POPI	Pop indirect	m[ac):=m[pp]; pp:=pp+1
1111 0100 0000 0000	PUSH	Push	pp=pp-1; m[pp]:=A
1111 0110 0000 0000	POP	Pop	ac:=m[pp];pp=pp+1
1111 1000 0000 0000	RETN	Retour	co:=m[pp]; pp:=pp+1(subroutine) B:=1 (interruption)
1111 1010 0000 0000	SWAP	Echange ac et pile	tmp :=ac ; ac :=pp ;pp :=tmp
1111 1100 YYYY YYYY	INSP	Incrémentation de pp	pp :=pp+Y (0<Y<255)
1111 1110 YYYY YYYY	DESP	Décrémentation de pp	pp :=pp -Y (0<Y<255)

II-2- Chemin de données

Le chemin de données (figure III-10) est l'ensemble des composants et moyens de communication empruntés par l'information sur micro machine support :

- 16 registres généraux A, B, C, ... CO, PP, AC
- Une UAL
- Plusieurs Bus: bus A, bus B, bus C.

Les registres généraux ne sont accessibles et connus qu'au sein de la couche microprogrammée. Ceux notés +1 et -1 sont particuliers et contiennent en permanence les valeurs indiquées.
Chaque registre peut transmettre son information sur les bus A ou B Il peut être chargé avec une information introduite via le bus C.
Les bus A et B alimentent l'UAL. L'UAL réalise l'une des opération (A + B), (A et B), A et Non A. La sélection de l'opération est précisée à l'UAL par des signaux de commandes F0 et F1
Deux bits indicateurs précisent si le résultat est <0 (N = 1) ou nul (Z = 1). Ils sont positionnés par l'UAL après traitement.
La sortie de l'UAL est reliée à l'entrée du décaleur qui effectue sur la donnée transmise par l'UAL soit un transfert simple sans décalage ou soit un décalage d'un bit à droite ou à gauche. La commande du décaleur est réalisé par les signaux S0 et S1

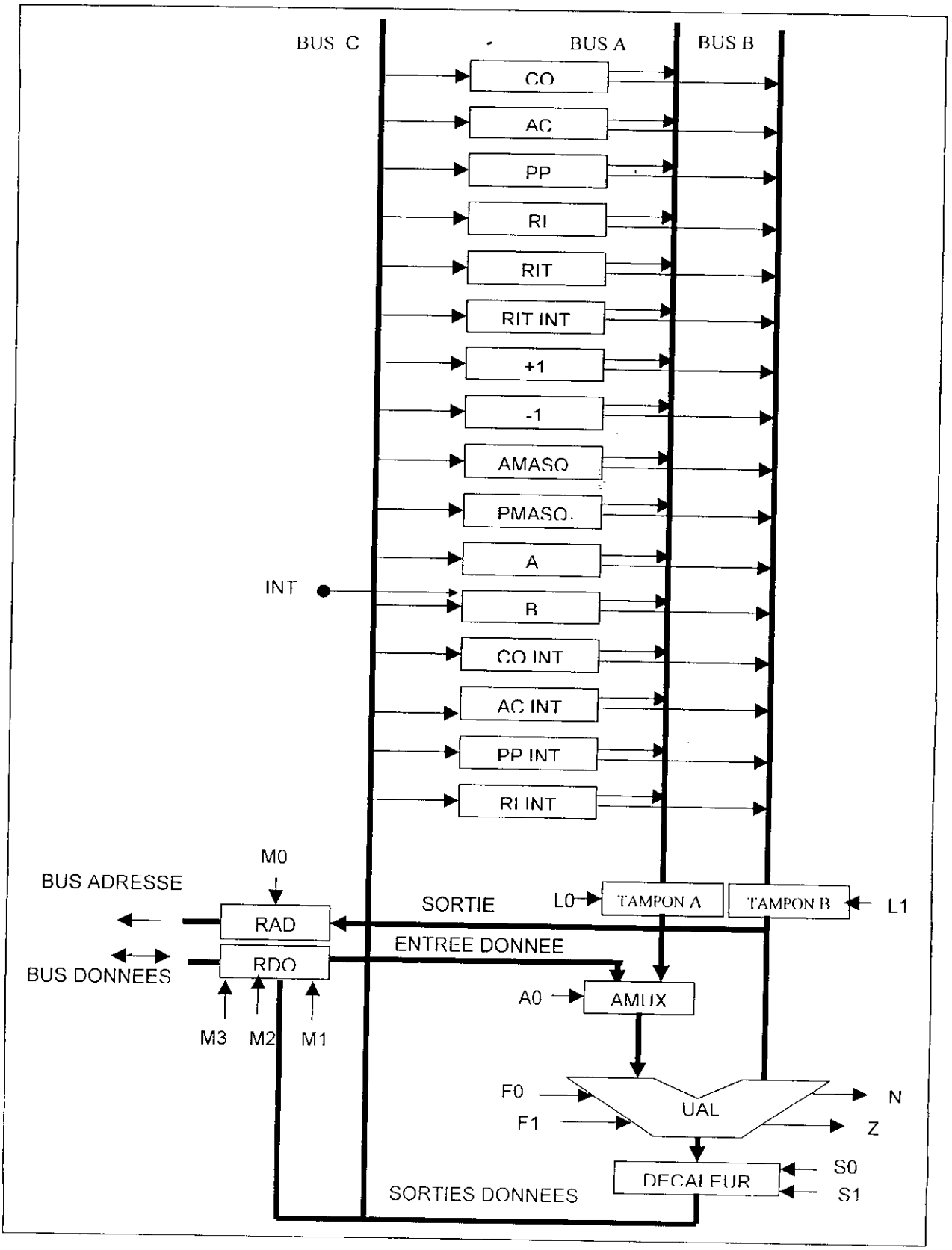


Figure III-10 : Chemin de données

Il est possible en une seule fois, pendant le même cycle, d'effectuer un décalage de deux bits à gauche en utilisant l'UAL et le décaleur. Soit un registre R. L'UAL fait l'addition R+R (ce qui correspond à un décalage à gauche d'un bit) et le décaleur réalise le décalage à gauche sur le résultat.

Les bus A et B sont reliés indirectement aux entrées de l'UAL par deux registres tampons (A et B). Ces registres tampons ont pour rôle de maintenir stable les opérandes pendant l'exécution d'une opération.

L'UAL est un circuit combinatoire qui traite en continu les données qui se présentent. Si l'UAL doit réaliser A = A + B pendant que le contenu du registre A est placé sur le bus A à l'entrée de l'UAL, le résultat en sortie sur le bus C est chargé en même temps que le registre A. Une valeur est modifiée dans le registre A alors que l'on s'en sert pour effectuer A + B. La valeur à droite de '=' doit être l'originale et non un mélange de l'ancienne et de la nouvelle valeur de A. Les signaux de commandes L0 et L1 assurent le chargement des registres tampons A et B avec les données présentes sur les bus A et B.

Les échanges de données avec la mémoire principale (externe à la micro machine) s'effectue par l'intermédiaires des registres RD0 et RAD via un bus de données et un bus d'adresses. Le registre RAD peut être chargé avec une donnée du bus B en même temps qu'une opération est effectuée par l'UAL. Le signal M0 commande le chargement de RAD.

Une donnée en sortie du décaleur peut être chargée dans le registre RD0 et commandée par M1 les signaux M2 et M3 commandent respectivement le positionnement du contenu de RD0 sur le bus de données (opération d'écriture en mémoire principale) ou l'enregistrement d'une donnée sur le bus dans RD0 (opération de lecture n mémoire principale).

Une donnée issue de la mémoire principale, stockée temporairement dans RD0, peut être utilisée comme l'une des entrées de l'UAL selon la sélection effectuée par le multiplexeur AMUX. Le signal A0 précise l'origine de l'opérande fourni à l'UAL. (RD0 ou tampon A).

Cette micro machine [1]est semblable à celle de divers processeurs en tranches du commerce.

II-3- Mécanisme de prise en charge de l'interruption

Notre conception prend en charge une seule interruption prioritaire. Si cette dernière survient, elle est mémorisée dans le bit 15 du registre B. La prise en charge de cette interruption se fera donc par mode polling après la fin de l'exécution de l'instruction en cours. Une routine a été incluse dans la mémoire microprogramme pour le déclenchement du basculement du contexte du microprocesseur. Le changement de contexte est réalisé par le mécanisme de fenêtrage.

II-4- Mécanisme de fenêtrage

Pour nous permettre de réaliser un changement de contexte immédiat, nous avons introduit le mécanisme de fenêtrage. Celui ci consiste à allouer un banc de registre par fenêtre. La fenêtre en cours correspond à l'ensemble des registres du modèle de programmation. Un changement de contexte consistera donc à un changement de banc de registres donc de fenêtre par la modification simple de l'adresse de sélection du banc. On pourra donc dire que nous avons réalisé un changement de contexte en un cycle d'horloge.

II-5- L'unité de commande

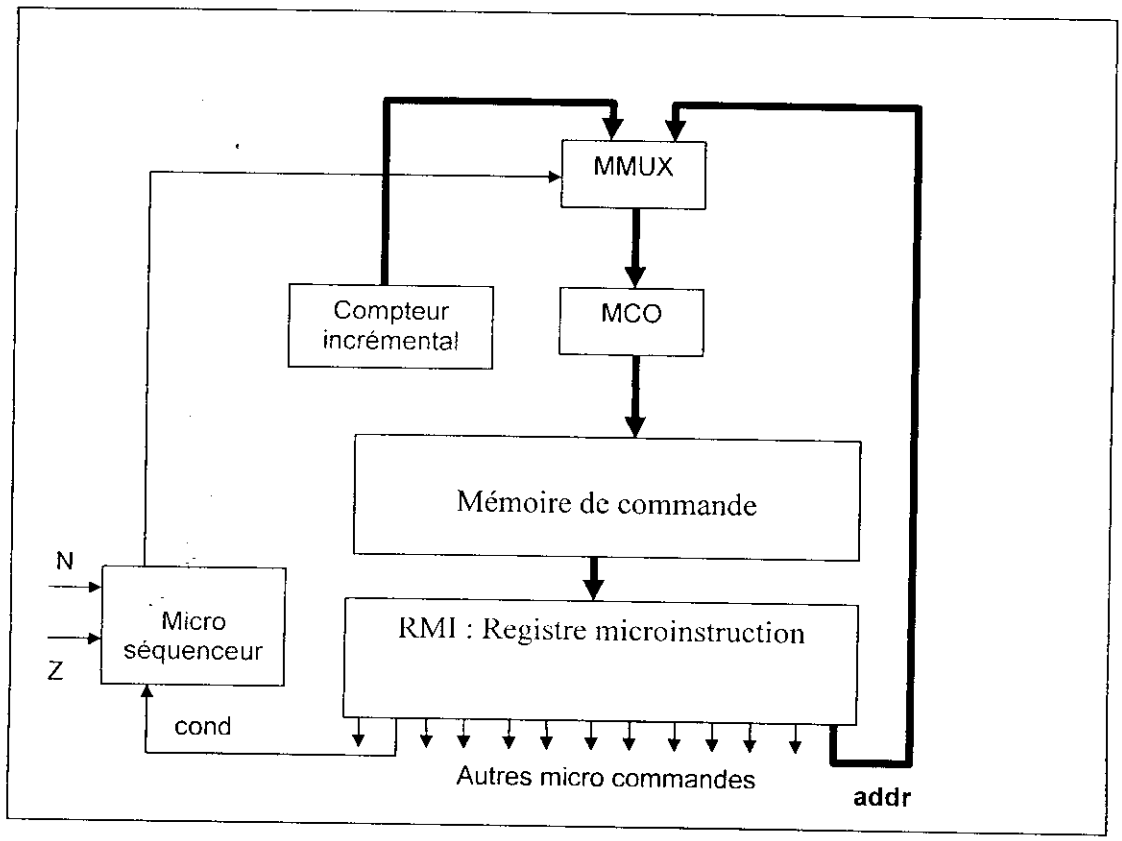


Figure III-11 : Unité de commande

La commande et le contrôle du chemin des données de la micromachine nécessitent 61 signaux divisés en 9 groupes fonctionnels (figure III-11) :

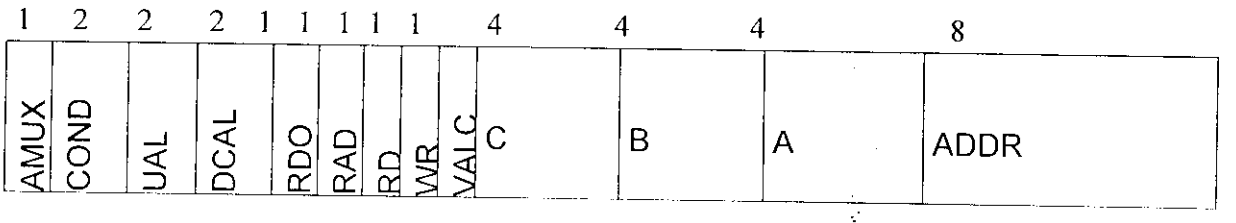
- 16 signaux pour commander la copie du contenu d'un des 16 registres sur le bus A.
- 16 signaux pour commander la copie du contenu d'un des 16 registres sur le bus B.
- 16 signaux pour commander le chargement d'un des 16 registres depuis le bus C.
- 2 signaux de chargement des registres tampons A et B (L0 et L1).
- 2 signaux définissant l'opération à effectuer (F0 et F1).
- 2 signaux de commande du décaleur (S0 et S1).
- 4 signaux de commande de RAD et RD0 (M0,M1,M2 et M3).
- 2 signaux d'indication d'opération de lecture et d'écriture avec la mémoire principale(WR, RD) identiques à M1 et M3.
- 1 signal de commande du multiplexeur AMUX (A0).

Connaissant la valeur de ces 61 signaux, on peut analyser le déroulement du cycle de base du chemin des données. La micro instruction dispose donc de 13 champs.

On suppose que la micro machine dispose d'un registre de commande de 61 bits: 1bit/signal, le signal étant actif au niveau 1. On peut réduire le nombre de bits pilotants le chemin des données en utilisant 3 décodeurs de type 4 vers 16 ; ce qui nous donne une économie de 3*12 bits.

Les signaux L0 et L1 valident les données des bus A et B dans les registres tampons A et B à des instants bien précis du cycle C'est l'horloge du système qui fournit ces signaux quand il le faut. Finalement, 23 signaux sont nécessaires.
Il est quelque fois utile de disposer d'un signal supplémentaire autorisant ou non le chargement d'un registre avec la donnée du bus C (on veut réaliser une opération avec l'UAL seulement pour positionner les indicateur N et Z). Ce signal est concrétisé par un signal de validation de chargement: VALC (1: autoristation 0 interdiction)
On note que RD et WR peuvent commander le chargement ou le vidage de RD0 sur le bus de données du système On déduit donc finalement 22 bits pour la longueur du champ de l a microinstruction.

Format de la micro-instruction



AMUX: 1 bit sélection de l'opérande gauche de l'UAL 0 registre tampon A 1: registre RD0
UAL : 2 bits choix de l'opération: 0 A+B 1: A et B 2: A 3: non A
DCAL : 2 bits sens du décalage 0: sans 1 droite 2 gauche
RD0 : 1 bit chargement de RDO avec C 0: non 1 oui
RAD : 1 bit chargement du registre tampon B 0: non 1 oui
RD : 1 bit opération de lecture de la mémoire principale 0: non 1 lecture
WR : 1 bit Opération d'écriture de la mémoire principale 0: non 1 écriture
VALC : 1 bit autorisation de chargement des registres généraux: 1 oui 0 non
C : 4 bits registre à charger avec la donnée du bus C
B :4 bits registre à charger avec la donnée du bus B
A : 4 bits registre à charger avec la donnée du bus A

L'enchaînement des micro-instructions est séquentiel. L'ensemble des microinstructions se trouve dans une mémoire spécifique ; la mémoire de commande différente de la mémoire centrale. Dans certains cas, l'enchaînement n'est pas séquentiel. Les micro instructions comprennent donc en plus une partie adresse, de deux bits sélectionnant l'instruction suivante:

COND = 0: pas de rupture
COND = 1: branchement à ADDR si N = 1
COND = 2: branchement à ADDR si Z = 1
COND = 3 branchement inconditionnel

Les entrées du microséquenceur sont les bits N et Z de L'UAL et les deux bits de COND (D et G). Ainsi le signal de commande de AMUX réalise le chargement de ADDR dans MCO.

L'horloge

Il est nécéssaire de disposer d'un circuit d'horloge (figure III-12) à plusieurs phases déterminant des sous cycles fonctionnels parfaitement ordonnés dans le temps (quatres cyleles dans notre cas) afin d'assurer le bon fonctionnement des instructions car le fonctionnement est séquentiel. Les actions entreprises pendant les quatre sous cycles sont :

- cycle clk1. Le registre RMI recoit la micro instruction pointée par le MCO.
- cycle clk2. L'horloge enregistre dans les registres tampons A et B les données présentent sur le bus A et B.
- cycle clk3. Le registre RAD peut etre chargé avec la donnée sur le bus B si le champ RAD le précise.
- cycle clk4. La donnée qui se trouve sur le bus C est stable est peut être chargé soit dans un des registres généraux et /ou dans le registre RDO.

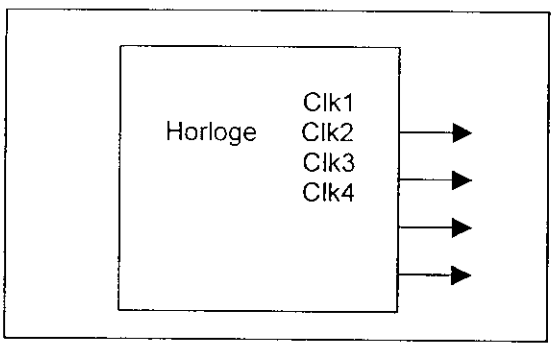


Figure **III-12** : L'horloge

II-6- Implémentation

Description VHDL

La description globale du circuit est structurelle . Elle est composée de trois blocs : l'horloge, le chemin de données et l'unité de commande. L'horloge est décrite en comportementale et les deux autres blocs en structurel.
Le chemin de données se compose d'un ensemble de registres avec décodeur intégré, une UAL, un décaleur, de registres adresses et de registres de données, de registres tampons , d'un multiplexeur et des divers bus.
L'unité de commande se compose du microcompteur ordinal (MCO), d'une mémoire de commande et d'un registre microinstruction (RMI). Chaque élément est décrit en comportemental.[Annexe C]

Synthèse

La cible : XILINX XC4005XLPC84 -3 .
Contraintes : BUFG pour les signaux *CLK* et *RESET*.

Taux d'utilisation des ressource :

Number of External IOBs	32 out of 192	16%
Flops:	0	
Latches:	0	
Number of Global Buffer IOBs	1 out of 8	12%
Flops:	0	
Latches:	0	
Number of CLBs	533 out of 576	92%
Total Latches:	0 out of 1152	0%

Total CLB Flops:	281 out of 1152	24%
4 input LUTs:	821 out of 1152	71%
3 input LUTs:	396 out of 576	68%
Number of BUFGLSs	6 out of 8	75%
Number of TBUFs	32 out of 1248	2%

La fréquence maximum

Minimum period: 79.852ns (Maximum frequency: 12.523MHz)
Maximum net delay: 40.656ns

Simulation fonctionnelle

La figure III-13 montre l'exécution de la suite d'instruction :

LODD 0FFF 0FFF
JPOS

Elle montre seulement le résultat de l'instruction LODD (premier cercle) et l'exécution détaillée du JPOS et le résultat obtenu (le deuxième cercle).

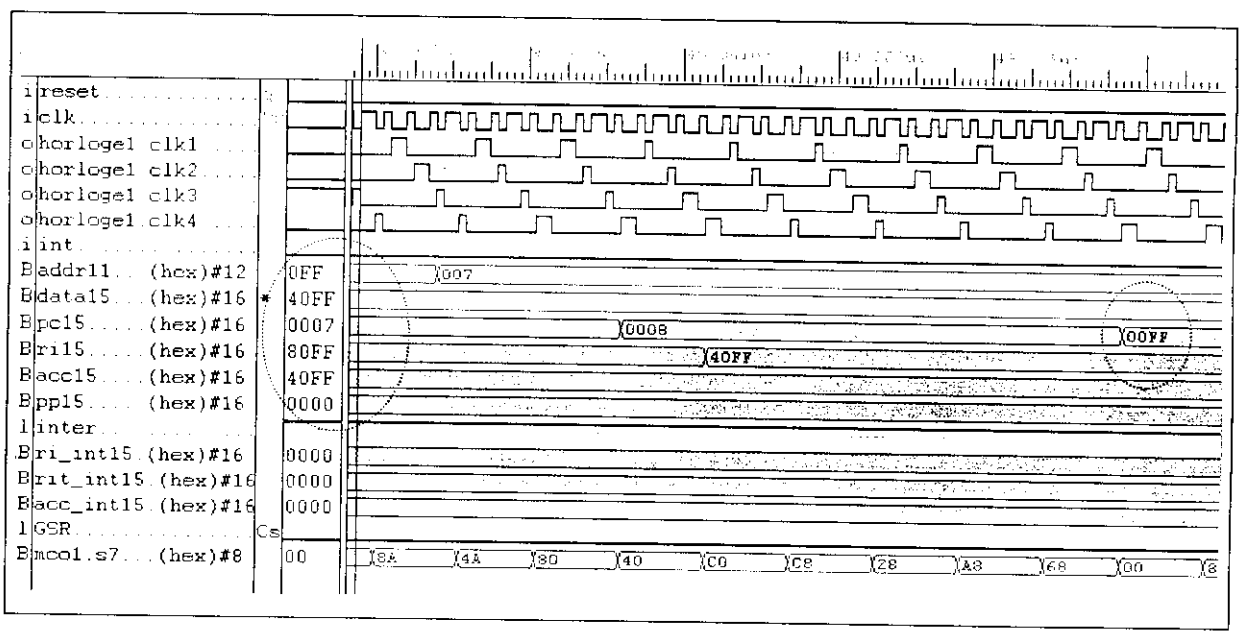


Figure III-13 : simulation fonctionnelle

La fréquence maximum du CISC_2

La période de l'horloge est supérieure ou égale à la somme des retards entre les deux composants synchronisés sur deux topes successifs. Pour la mesure des retards, on utilise le simulateur en mode timing figure 3.14 .

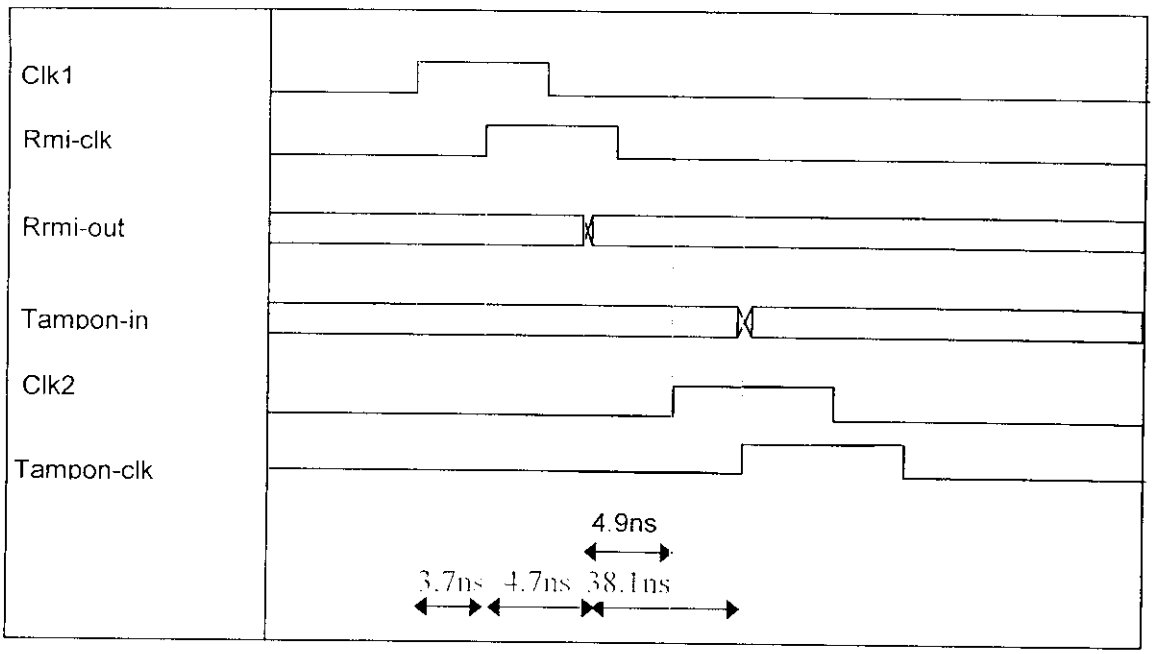


Figure III-14 : Retards mesurés entre CLK1 CLK2

La figure III-14 montre les retards obtenus entre les deux tops CLK1 et CLK2. Le retards le plus important est celui entre RMI_out et l'entrée des registres tampon ; ceci est du à l'utilisation dU multiplexeur pour réaliser la sélection d'un registre parmi 16 (CO, ACC,PP ....) sur les bus A et B,

T_1=3.7+4.7+38.1- 4.9
T_1=41.6 ns.

De la même façon on calcule T_2, T_3 et T_4.

T_2=9.4+5.1+5.6-7.9
T_2=12.2 ns.
T_3=(M+7.9+4.4-8.8)/5=22.7ns
avec M le temps d'accès mémoire, (M=100 ns pour une RAM externe).

T_4=8.8+5.7+33.3-3.7=47.6 ns.

T=Max(T_1, T_2, T_3, T_4)
T=Max(41.6 ns, 12.2 ns, 22.7 ns, 47.6 ns)=47.6 ns

On remarque que le retard maximal est celui entre les éléments synchronisés par CLK4 et CLK1. Le retard majoritaire est celui de la mémoire de commande. Cette période correspond au temps minimal entre deux tops de synchronisation successifs d'où la fréquence max est de 21 MHZ.

Test

Nous avons testé notre implémentation à la fréquence maximale calculée ; mais les résultats n'étaient pas comme prévus. Ensuite nous avons baissé la fréquence jusqu'à 14.4 MHz, pour avoir une exécution acceptable. La figure III-15 montre la simulation temporelle à cette

fréquence de l'instruction STOR F0F. Ensuite nous avons testé le mécanisme de l'interruption figure III-16. Sur cette figure, l'horloge CLK et les signaux de synchronisation CLKi apparaissent en rectangle noir. Ceci est du au fait que nous avons réduit l'échelle des temps pour afficher toute la séquence.

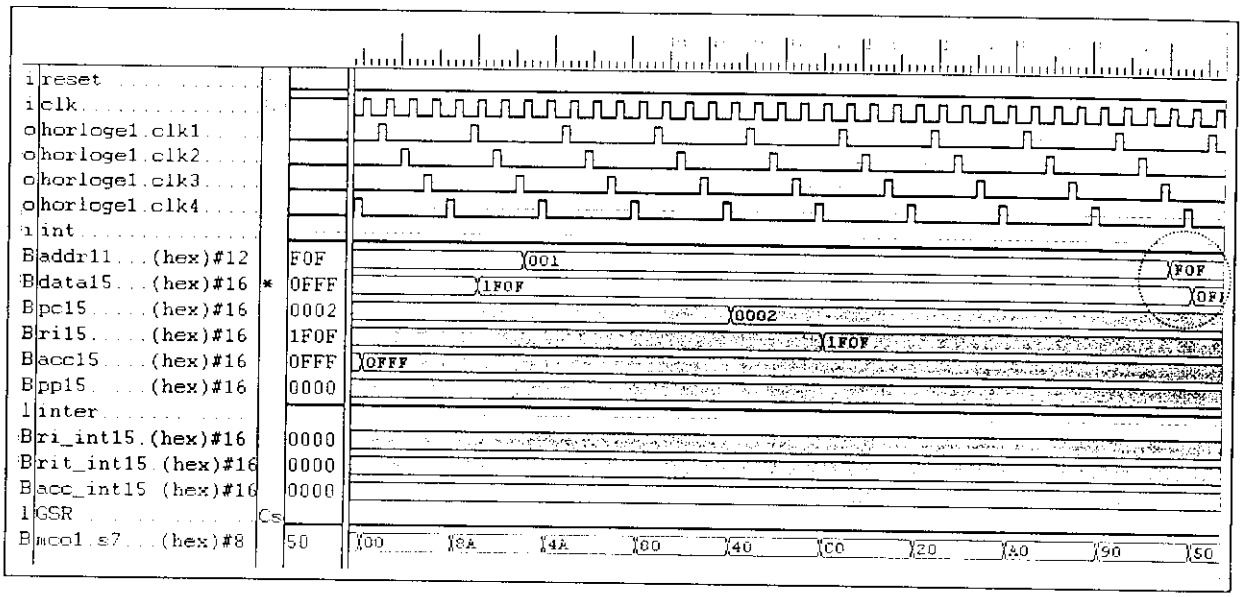


Figure III-15 : Simulation temporelle de l'instruction STOR

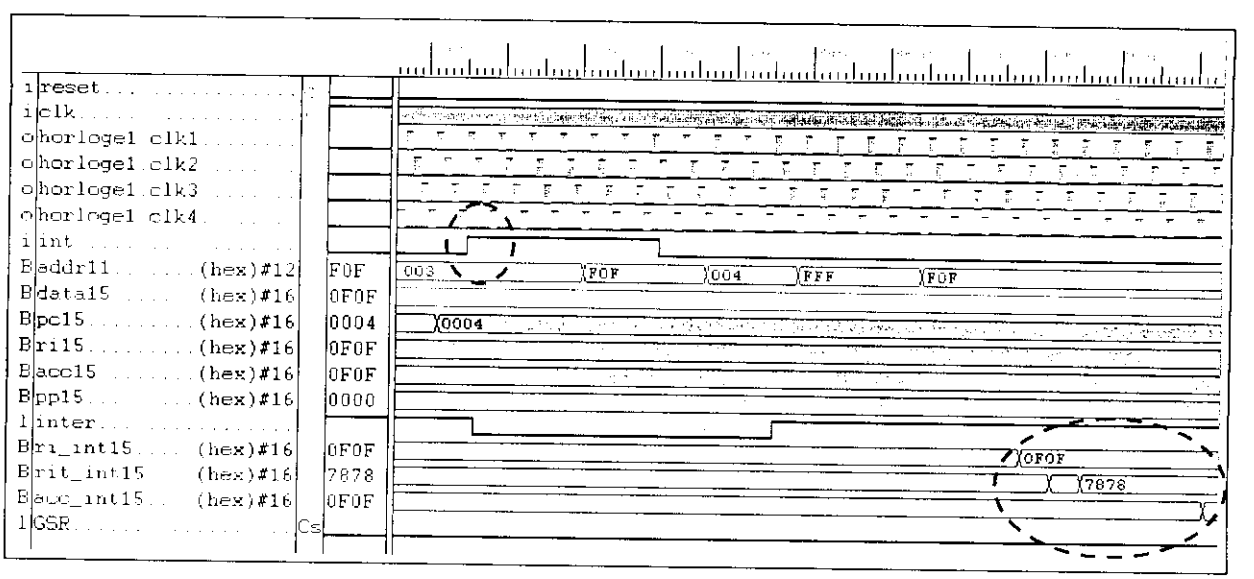


Figure III-16 :Simulation temporelle du mécanisme d'interruption

III- Conception et implémentation de la version CISC_3

III-1- Le pipeline

Le pipeline est une structure où plusieurs instructions se recouvrent au cours de leurs exécutions. On parle de parallélisme d'instruction et d'une manière plus précise de la classification MISD. Ceci peut être obtenu par la mise en parallèle des unités fonctionnelles constituant le microprocesseur.

D'après les chapitres précédents, l'exécution d'une instruction se fait en trois cycles : Fetch, Decode et Execute. Si on divise le microprocesseur en trois unités, chacune correspond à un cycle de l'exécution, on aura un pipeline de trois *étages* (figure III-17).

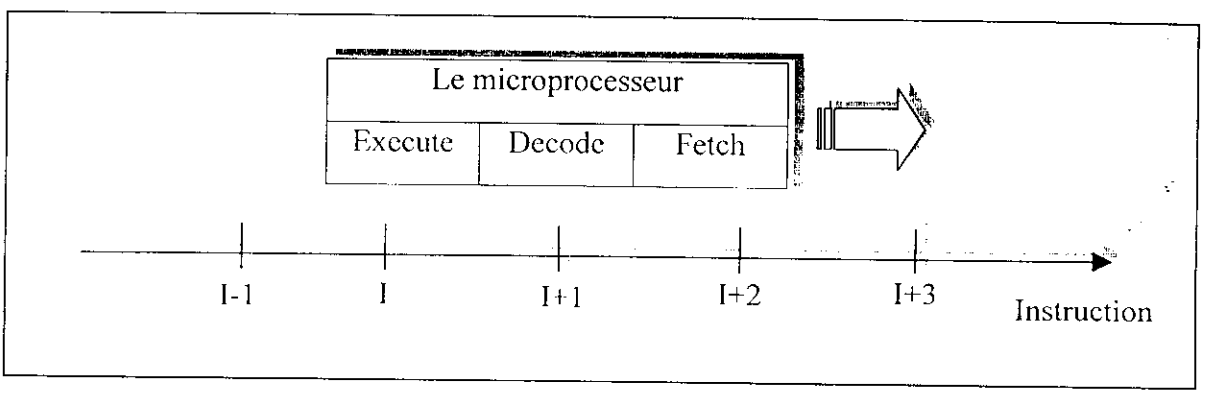


Figure III-17 : Pipeline à trois étages

Mais cette décomposition n'est pas unique. On peut avoir d'autre divisions plus performantes et plus intéressantes. Pour homogénéiser les étages en terme de temps ou nombre de cycles d'horloge nécessaires, on doit séparer les opérations accès mémoire ; d'où l'on obtient le pipeline de la figure III-18

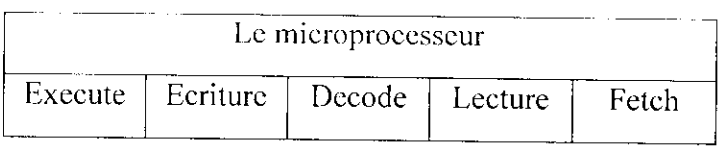


Figure III-18 : pipeline à quatre étages

Dans le cas idéal, la vitesse du microprocesseur sera multipliée par le nombre d 'étages, par apport à la vitesse de la même structure sans pipeline. Mais ce n'est pas le cas par ce que le fonctionnement parallèle des étages pose des problèmes, appelles aussi *aléas*, qu'on peut classer en trois catégories :

- Les aléas structurels : dûs à l'utilisation de la même ressource par deux étages, c'est à dire en même temps.
- Les aléas de données : intervient quand une instruction utilise les résultats d'une instruction précédente. Ces aléas sont surtout présents dans les structures qui utilisent des registres internes pour les données intermédiaires.
- Les aléas de contrôle : dûs aux instructions de saut.

Pour illustrer ces aléas et leurs solutions, prenons le cies 1 précédemment conçus et essayons de rendre sa structure pipeline.

On commence par le décomposer en étages, fixer les éléments de transfert et de sauvegarde des résultats entre étages. Ce qui donne la structure de la figure III-19 .

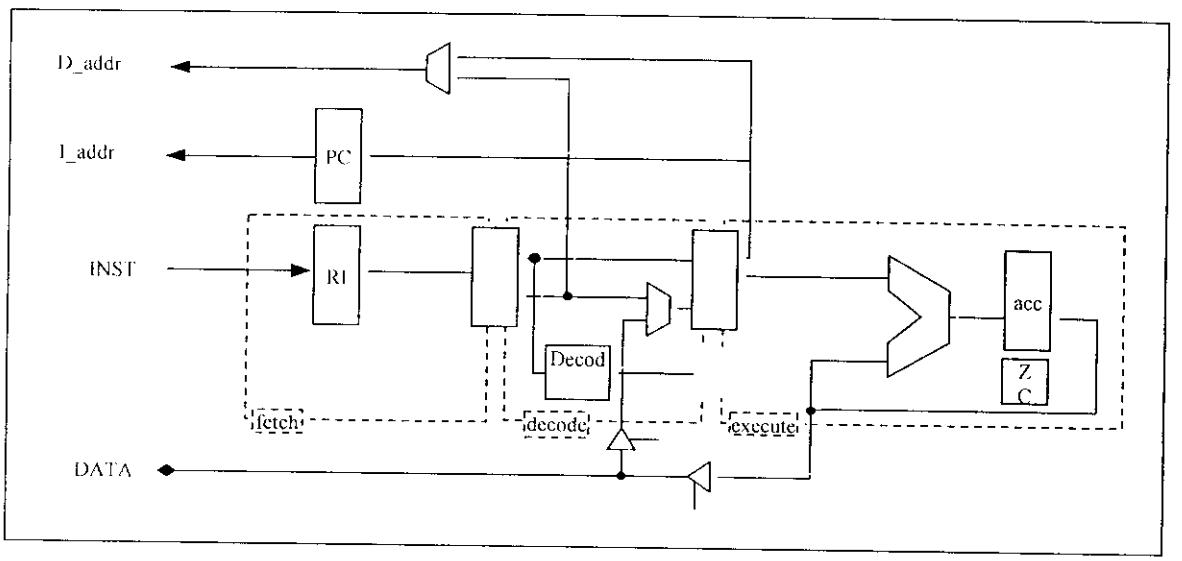


Figure III-19 : Chemin de données découpé en 3 étages

Les étages sont les éléments dans les cadres en pointillés. Les éléments de transfert en gris. Il reste des éléments externes, *PC*, le bus de donnée (*DATA*) et le multiplexeur de l'adresse de données (*D_addr*). Ce sont les éléments qui causent les aléas parce qu'il sont communs entre les étages.

Aléas et solutions

- *Aléas structurelle :* le bus *DATA* est utilisé par l'étage *decode* pour un adressage direct, et par l'étage *execute* pour l'instruction STOR. Idem pour le bus *D_addr*.
- *Aléas de données :* on peut voir l'aléas précèdent comme aléas de données quand l'instruction à adressage direct charge la donnée que produit l'instruction STOR .
- *Aléas de commande :* causés par les instructions JUMP et SKIP.

➢ *Solutions* :

Deux solutions peuvent être envisagées :

- *SOFT* : le compilateur doit suivre les instructions STOR, JUMP et SKIP par des NOP (branchement retardé).
- *HARD* : l'étage *decode* détecte ces instruction et force le *RI* à NOP tout en empêchant le *PC* de s'incrementer.

Dans notre implémentation, nous avons opté pour la deuxième solution par ce qu'elle est simple à réaliser et la première est coûteuse en mémoire.

III-2- L'unité de commande

- L'horloge

On a besoin d'un seul top de synchronisation pour les éléments des étages et un autre pour les éléments de transfert. On peut utiliser le signal *CLK* directement et prendre les éléments de transfert sensibles aux front descendants.

- Les signaux de commande

L'étage *FETCH* ne possède qu'une seule commande NOP_RI pour force le *RI* à NOP. L'étage *DECODE* génère les commandes après décodage de l'instruction, on utilise une partie et on transfert le reste aux autres étages et éléments *communs*.
L'étage *EXECUTE* reçoit ses commande de l'étage précèdent ; par contre il doit, pour l'instruction SKIP, tester la condition et renvoyer la commande au compteur ordinal (PC).

La figure 3-20 donne l'ensemble des signaux de commande du microprocesseur. Les multiplexeurs du bus *DATA* et du bus *D_addr* ainsi que le buffer trois états en entrée du *DATA* sont commandés par le signal *D* (direct / immédiat).

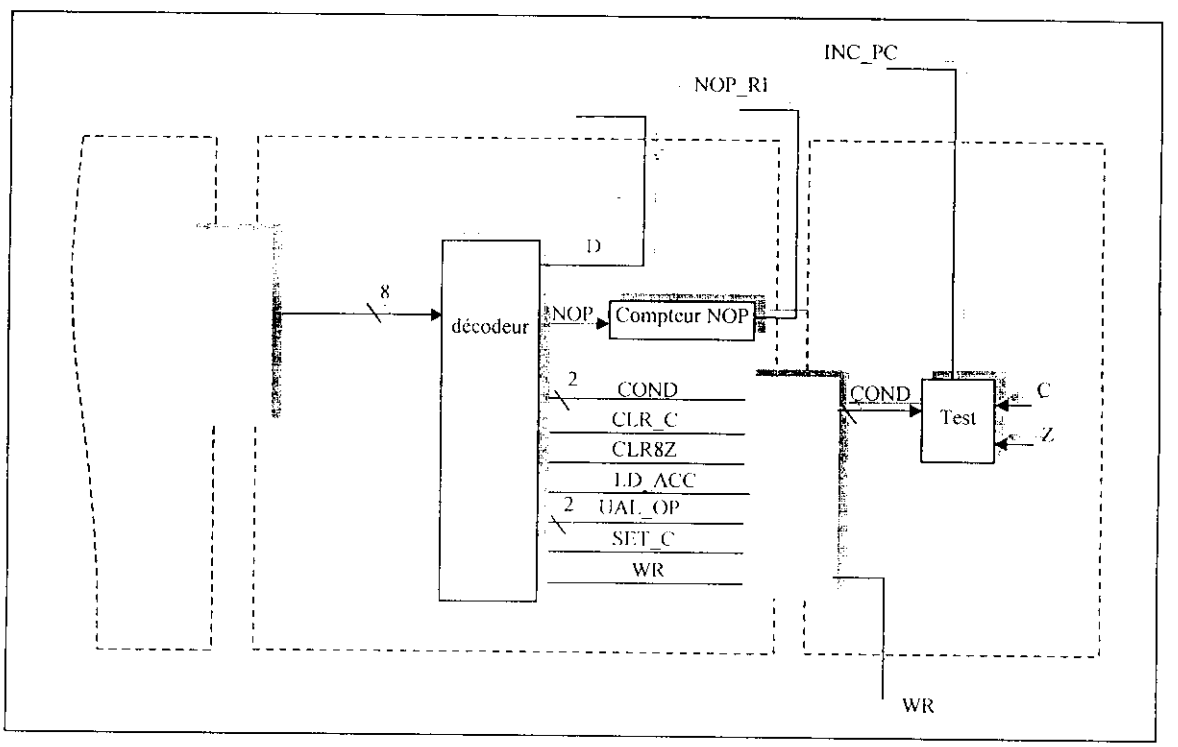


Figure III-20 : Signaux de commande

III-3- Implémentation

Description VHDL
Chaque étage est décrit en comportemental et le circuit général en représentation structurelle. Il contient les trois étages et les éléments communs [Annexe C].

Synthèse
La cible : XILINX XC4005XLPC84 -3 .
Contraintes : BUFG pour les signaux *CLK* et *RESET*.

Simulation fonctionnelle
La figure III-21 représente la simulation de la suite d'instructions suivante :
LOAD_D 9 C9 avec DATA =3.

ADD_I	1	A1
STOR_D	7	B7

Les cycles et les étages sont séparés par des trait en pointillés. On remarque bien la solution de l'aléa : quand l'instruction STOR est en exécution, les étages antérieurs sont en NOP (85).

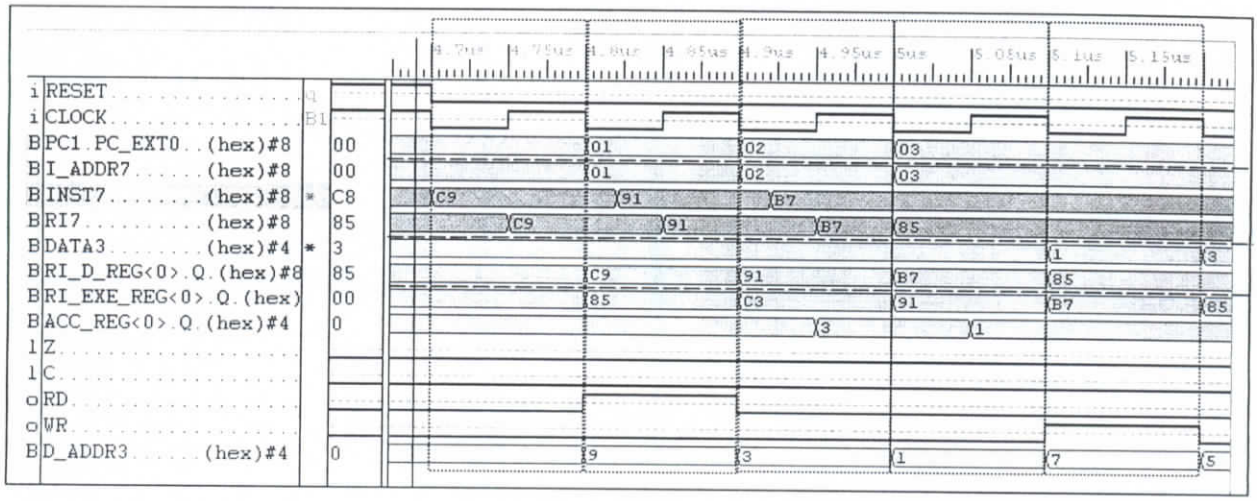


Figure III-21 : Simulation fonctionnelle

La simulation temporelle

La fréquence maximale fournie par l'outil de synthèse est donnée sur le rapport *Post Layout Timiçng Report* :

Minimum period:	53.770ns (Maximum frequency: 18.598MHz)
Maximum net delay:	7.860n

Il le calcule à partir des retards estimés. Mais comment exactement ?

Essayant de calculer à notre manière : l'écart temporel entre deux tops de synchronisation est supérieur (ou égale idéalement) à la somme des retards entre les deux composants synchronisés sur ces deux « tops » d'horloge.

On calcule d'abord pour chaque étage et on prend le maximum. Théoriquement l'étage le plus consommateur en temps est le FETCH car il utilise les deux fronts de l'horloge ; le front montant pour le *RI et* le front descendant pour le *RI_d. U*ne autre cause c'est que l'accès mémoire est en général lent.

Pour la mesure des retards, on utilise le simulateur en mode *timing,* la figure III-22 montre comment faire.

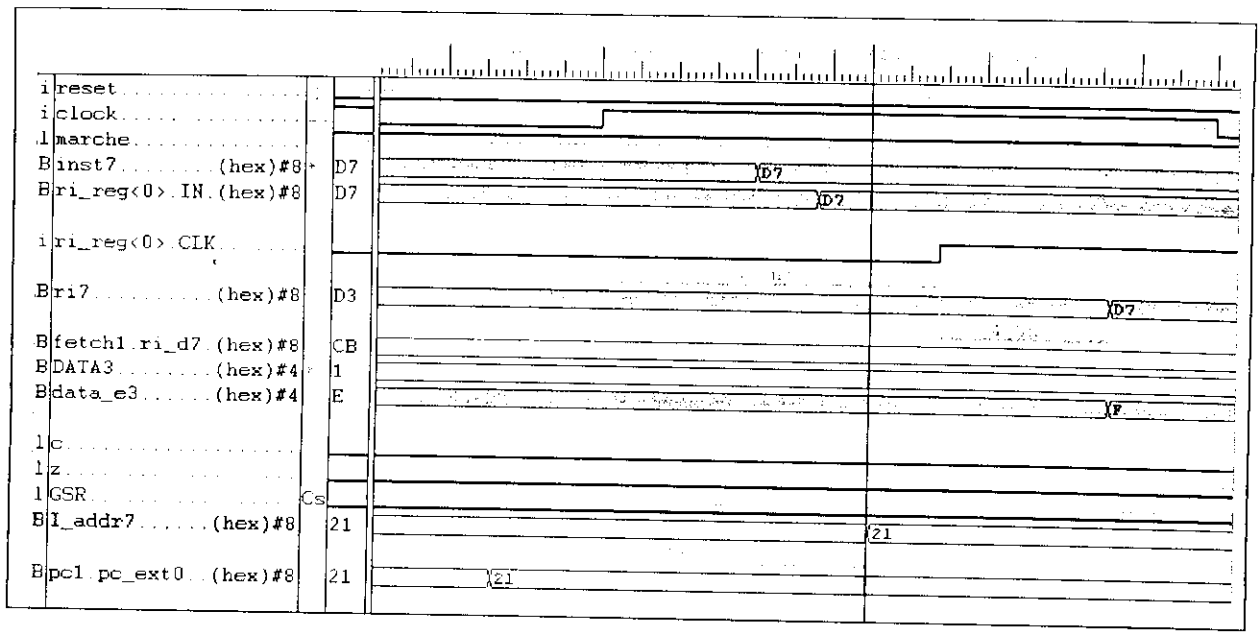


Figure III-22 : Mesure des retards

On calcule la valeur minimale de la période de l'horloge, c'est à dire le top de synchronisartion est pris juste à la stabilisation de l'entrées des registres.
Les résultats obtenus pour l'étage FECTH sont
Remarque : les valeurs 4.5ns et 8.7ns se répètent. Elles correspondent aux temps de propagation typique de bascule du CLB et du circuit d'attente de la mise à zéro (*reset*).

- Etage FETCH :

$T_1/2$=4.5+8.7+9.8+M-8.7+1.6 =>T_1=15.9+2 M ns M est temps accès mémoire .
$T_1/2$=8.7+4.5-8.7 -> T_2=9.
Si le rapport cyclique est de 0.5 alors T_F=Max(T_1+T_2) T_F=15.9+2M ns.

- Etage DECODE :

RI_d (bascule) 4.5ns.
Multiplixeur 2.9ns.
T=8.7+4.5+2.9-8.7
T_D=7.4ns.

- Etage EXECUTE :

RI_e 4.5ns
UAL 4.4ns
ACC 4.5ns
T= 2 (4.5+4.4)=17.8ns.
Dans le cas d'un STOR, le seul retard considérable est l'accès mémoire.
Alors T_E= MAX(M, 17.8ns).
Pour le circuit en général : T= MAX(T_F ,T_D ,T_E)
T= MAX(14.4+2M, 17.8ns)Notre raisonnement était juste, la période est celle de l'étage FETCH .

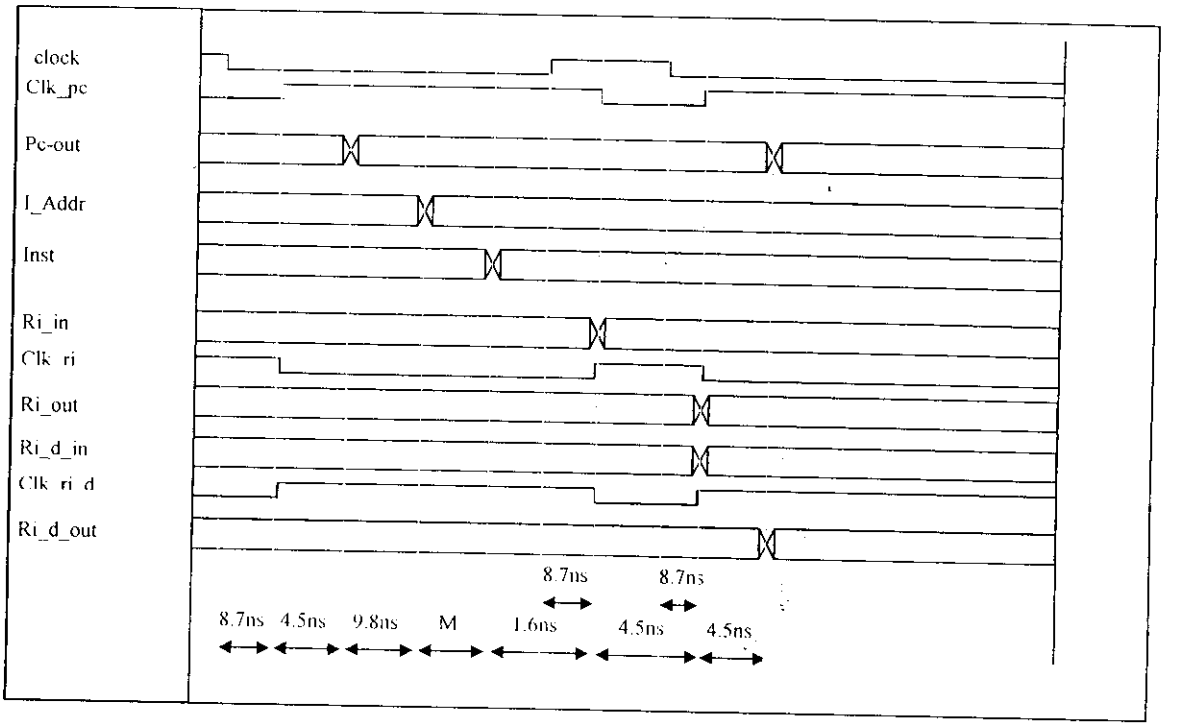


Figure III-23 : les retards mesurés de l'étage exécute

Test

Sur le simulateur l'accès mémoire est nul. La période d'horloge est de 17.8ns (la fréquence est de 59.17MHz).

La simulation n'a pas donnée les résultats prévus, a cause des états transitoires, tant que la fréquence est au dessus de 40MHz, ceci est du aux manque de précision du simulateur. Sur la figure III-24, on voit une simulation à 50MHz,avec une mémoire à temps d'accès nul (une fois l'adresse de l'instruction *I_addr* est stable l'instruction *Inst* est stable). On remarque que le RI n'est pas chargé par sa valeur d'entrée après le top, mais pour une fréquence de simulation 40MHz (figue III-25) l'instruction se propage comme prévu.

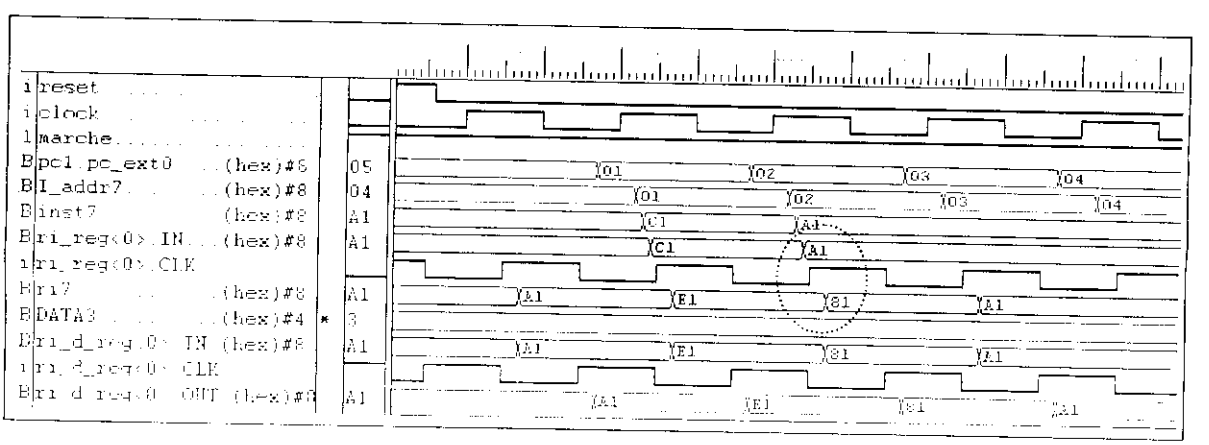


Figure III- 24 : Dysfonctionnement constaté

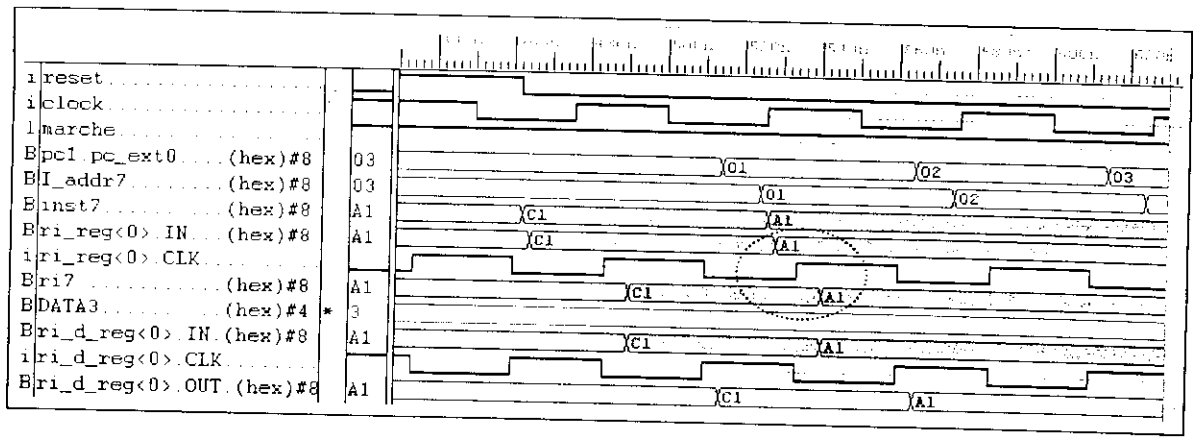


Figure III-25 : Fonctionnement sans erreur

IV- Evaluation des performances des trois implémentations

La performance d'un microprocesseur est une grandeur relative au domaine d'application. Elle est mesurée pratiquement par l'exécution d'un programme conçu spécialement pour cette fin appelés *Benchmark*. Il donne le résultat chiffré en unité MIPS (million instructions par seconde). Chaque *Benchmark* mesure la performance par apport à quelque critères.
Les plus importants sont :

- L'accès mémoire
- L'appel de sous-programme
- La gestion des boucles

Pour nos implémentations, on a pas pu les tester physiquement à cause du circuit FPGA défectueux sur la carte XS40 disponible au laboratoire.

Alors la seule performance que nous pouvons estimer est le nombre moyen d'instruction par seconde :

Cisc_1 :

On a :
3 cycles par instruction
Fréquence max =13.6MHz

D'où

$N=13.6\ 10^6/3=4.53\ 10^6$ I/s = 4.53 Mips

Cisc_2

4 cycles par microinstruction,
une moyenne de 12 microinstruction par instruction
fréquence max = 14.4MHz

D'où

$N\ 14.4/4\ 10^6/12 = 0.3\ 10^6$ I/s = 0.3 Mips

Cisc_3

1 cycle par instruction

3 cycles par instruction de branchement ou STOR
fréquence max =59.1MHz
Si T est le taux d'apparition de ces dernières instructions dans les Programmes, alors on peut estimer :

$N = (40 \times T)/3+(40 \times(1-T))$

Pour T =0.5 N=26.6 10^6 I/s = 26.6 Mips

Conclusion

Les performances estimées de cisc3 nous montre qu'il est plus performant que le cisc1. Cela était notre but en transformant sa structure en structure pipeline. Le temps d'exécution des instruction dans le cisc3 est amélioré par rapport à ce qu'il devait être théoriquement c'est à dire trois fois la performance du CISC_1.

La complexité des instructions effectuées et le format des données du cisc2 ont influencés sa performance. Aussi sa commande micro programmée est consommatrice de cycles d'horloge et le rend ainsi plus lent. Elle simplifie en contrepartie la conception du microprocesseur.

Conclusion générale

Notre travail a aboutit à la conception de trois architectures différentes de microprocesseurs. Elles ont été introduites suivant leur degré de complexité ; le but recherché étant évidemment pédagogique.

Nous avons traité les concepts de base des microprocesseurs, illustré les différents structures de base et surtout nous avons donné une initiation sur un outil spécialisé.

Des améliorations peuvent être introduites. Elles peuvent prendre plusieurs axes :

- l'amélioration dans l'approche de conception
  On peut appréhender cette amélioration par l'introduction d'une méthodologie de conception descendante dans le but d'optimiser le jeux d'instruction en fonction du domaine de l'application cible par l'analyse préalable des programmes développés. Elle passe par la classification des instructions et l'analyse de leurs occurrences

- l'amélioration de l'architecture
  L'idée visée reste évidemment la performance qui, à technologie égale, passe par une introduction poussée du parallélisme. Divers concepts peuvent être introduits On cite à titre d'exemple les VLIW, les processeurs cellulaires et les architectures systoliques entre autres.

- l'introduction de blocs fonctionnels spécialisés.
  Cette spécialisation vise soit l'adaptation du chemin de données aux structures de données à traiter (cas des DSP) ou soit l'introduction de blocs de fonction(s) câblée(s) ou microprogrammée(s) visant à suppléer le microprocesseur (multimédia) ou travailler à son insu (cryptage).

ANNEXE A

L'OUTIL DE SYNTHESE

I- Présentation

L'outil de synthèse qu'on a utilisé est le *XILINX FOUNDATION SERIES* version 1.5, conçu par l'inventeur et un des fabriquants des circuits FPGA *XILNIX*.
Un gestionnaire des projet (*Project Manager*) permet l'utilisation et la supervision des taches de l'outil, son interface graphique est représenté sur la figure A-1, elle présente un espace de travail divisé en trois régions :

- Navigateur de la hiérarchie.
- Organigramme fonctionnel.
- La fenêtre des message.

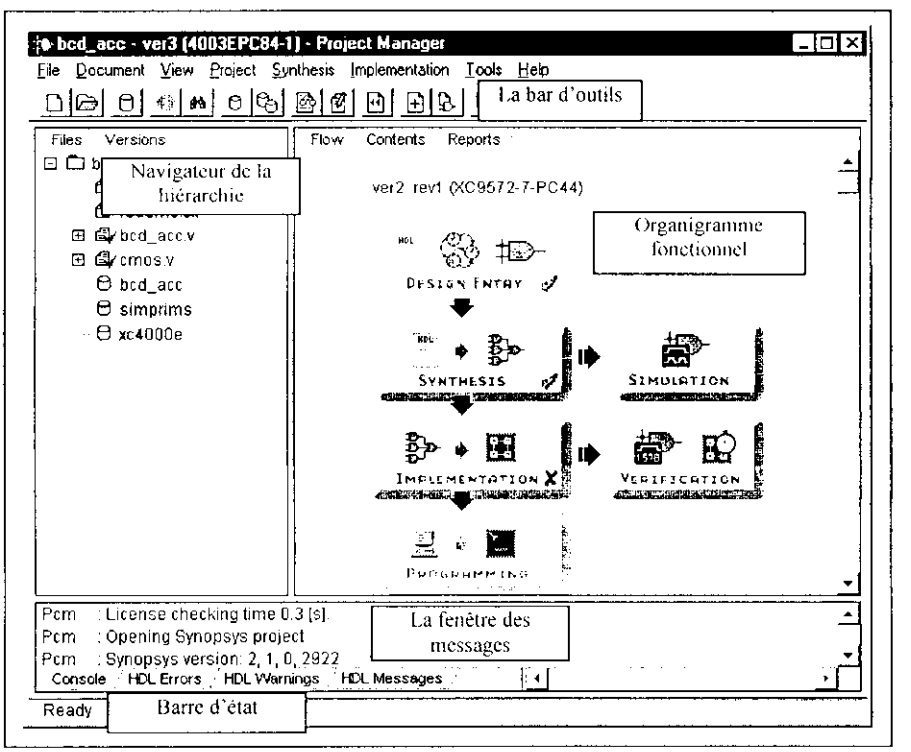


Figure A-1 : gestionnaire de projet

I-1-Navigateur de la hiérarchie

Il présente deux anglets fichier et version :

- l'anglet fichiers : contient les fichiers contraintes, les fichiers HDL et ses composants et les librairie attachées au projet.
- L'anglet versions : contient les versions conçus et leur états : optimisé, synthétisé et "implémenté".

I-2-Organigramme fonctionnel :

Présente les commandes des étapes de l'implémentation et leurs états : réussite, avertissement, échec.
Dans la même région on trouve les onglets contenus et rapports, les contenus sont tout les composants (*ENTITY*) décrites dans le projet, les rapports sont les messages rendus par l'outil pour chaque étape de l'implémentation mais organises sous forme de rapports résumés.

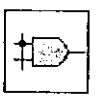
I-3-La fenêtre des messages :
Elle contient tous les messages rendu par l'outil pendant l'utilisation.

II-Un projet :
C'est l'organisation du travail permise par le logiciel, où les différents fichiers source, configuration et résultat sont sauvegardés et traités. tous les fichiers sont sauvegardés dans un répertoire appelé *project work directory*, ce répertoire porte le même nom du projet, un fichier d'extension *''pdf''* portant les configurations du projets appelé *project description file(pdf)* est sauvegardé dans le même répertoire que le *project work directory.*

II-1-Créer un projet
Le *XILINX FOUNDATION SERIES* nous permet de créer deux types de projet : HDL et schématique, dans le premier type on doit spécifier le nom (maximum 8 caractères) et l'emplacement, mais dans le deuxième on doit spécifier aussi la cible (famille, composant et *speed grade*).
La création du projet consiste à la création du répertoire et du fichier description, on y trouve : un fichier d'extension *ucf*, des bibliothèques : une associé au projet ,une associé à la cible pour les projet schématique.

III- Les étapes de la conception

III-1- Décrire un circuit
Apres la création du projet, Trois éditeurs sont proposés dans le *XILINX FOUNDATION SERIES* pour la description du circuit, on commence par le moins abstrait : l'éditeur schématique, l'éditeur de machine d'états et l'éditeur HDL.

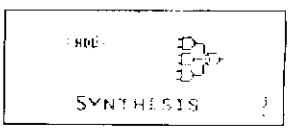


- L'éditeur schématique : il permet la description du circuit par la connexion des différents composants de base supportés par la cible .
- L'éditeur de machine d'états : il permet de décrire le circuit comme une machine d'états finis : horloge, états, conditions et transitions .
- L'éditeur HDL : il la description, et la vérification de la syntaxe, dans les trois langages de haut niveau VHDL, ABEL et Verilog.

Pour un projet HDL le fichier contenant la description doit être ''ajouté'' au projet (project-> add source file) pour qu'il soit pris compte.
Deux notions importantes dans la description doivent être expliquées : Macro et bibliothèque.

- Macro: dans un projet schématique on peut utiliser un circuit à l'intérieur d'un autre, sans avoir à décrire le sous circuit à chaque utilisation, ce dernier est décrit en utilisant un des éditeurs précédents. Alors elle permet le mixage des méthodes de description HDL et schématique et aussi la simplification de la structure du circuit globale.
- Bibliothèque: c'est ensemble de macros et de primitives (éléments de base de la cible) qui peut être prédéfinis (*seystem librarie*) ou créer et attacher par l'utilisateur.

III-2-Synthétiser une description
Pour cet outil la synthèse signifie le passage d'une description en HDL (comportementale) à une représentation par un circuit logique idéal, le fichier résultant est le fichier *netlist*, alors pour un projet schématique cette étape n'existe pas.
La synthèse est lancé par l'icon *synthesis* , la boit de dialogue de la synthèse contient les parametres suivants :



- Top Level : choisir quel composant (entity) parmi les composant définis, à synthétiser
- Nom de la version
- La cible : famille, composant, speed grade.
- Paramètres synthèse :
  1. La fréquence de la cible.

2. Optimisation : pour la vitesse, ou l'espace.
3. Effort : compilation langue donnant meulier résultat ou rapide sans essayer toute les méthodes,

Pour une synthèse réussite on aura un coché vert, sinon une croie rouge, un point d'interrogation signifie synthèse faite avec avertissement.
Une fois la synthèse faite une simulation fonctionnelle peut être lancé,

III-3- La simulation fonctionnelle :

Le simulateur est un outil qui nous permet de tester le circuit en stimulant ses entrées et en affichant la forme de n'importe quel signal interne du circuit (figure A-2).

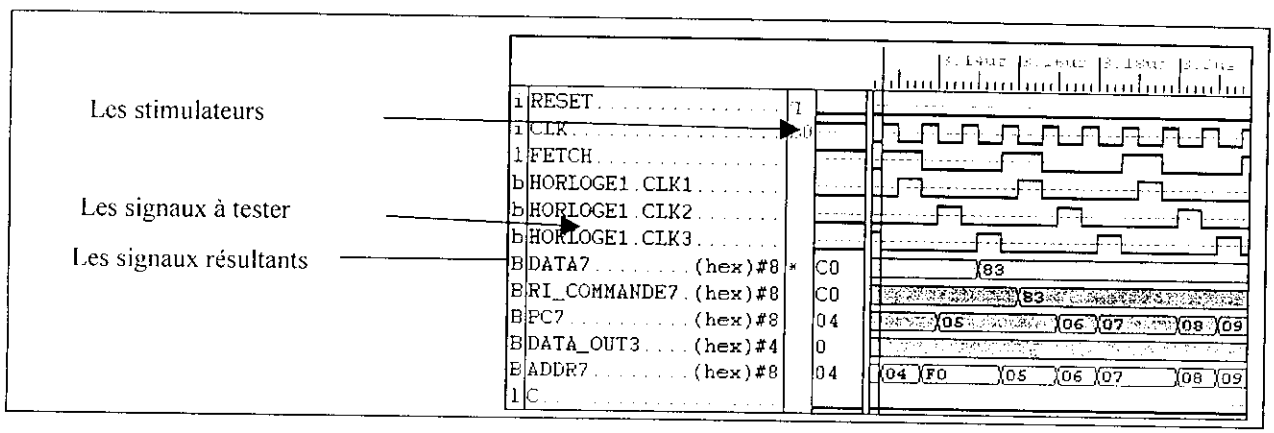


Figure A –2 :le simulateur

III-4- Implémentation :

Le circuit est définis comme un réseau de porte logique, maintenant il faut trouver leurs disposition sur les circuits FPGA, c'est l'étape d implémentation.
L'implémentation est lancé par l'icon implémentation, le paramètre le plus important est le fichier UCF (user constraint file) qui contient les contraintes imposé par le concepteur.

Une fois la 'implémentation est lancé un une visualisation des étapes suivis apparaît (figure A-3), elle représente le diagramme fonctionnelle de la synthèse.

La simulation temporelle :

C'est le même simulateur que le fonctionnelle mais en mode *Timing*.

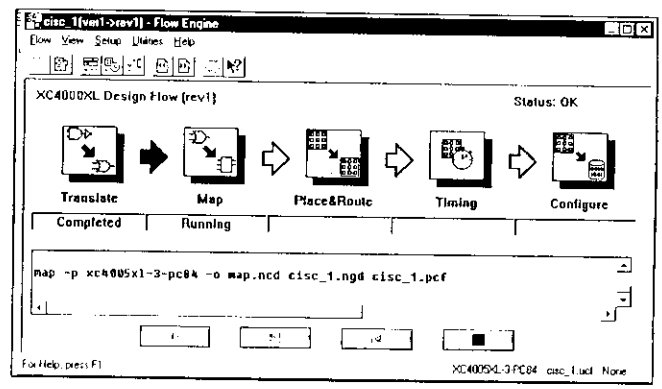


Figure A-3: implémentation

VI- Les rapports

les résultats détaillés de chaque étape sont donnés sous forme de rapports, on y trouve par exemple les taux des ressources utilisés, les retards de chaque interconnexion et une chose très importante c'est les erreurs et leurs causes.

V- Outils supplémentaires

Parmi les outils associés on trouve l'éditeur des contraintes (*Constraits Editor*) et EPIC designer: l'application graphique pour l'affichage et la programmation manuelle des FPGAs, cet outil et très intéressant par ce qu'il nous permet de voir et de modifier le niveau le plus bas de notre conception (figure A-4): les générateur de fonction F et G, les interconnexion ...etc.

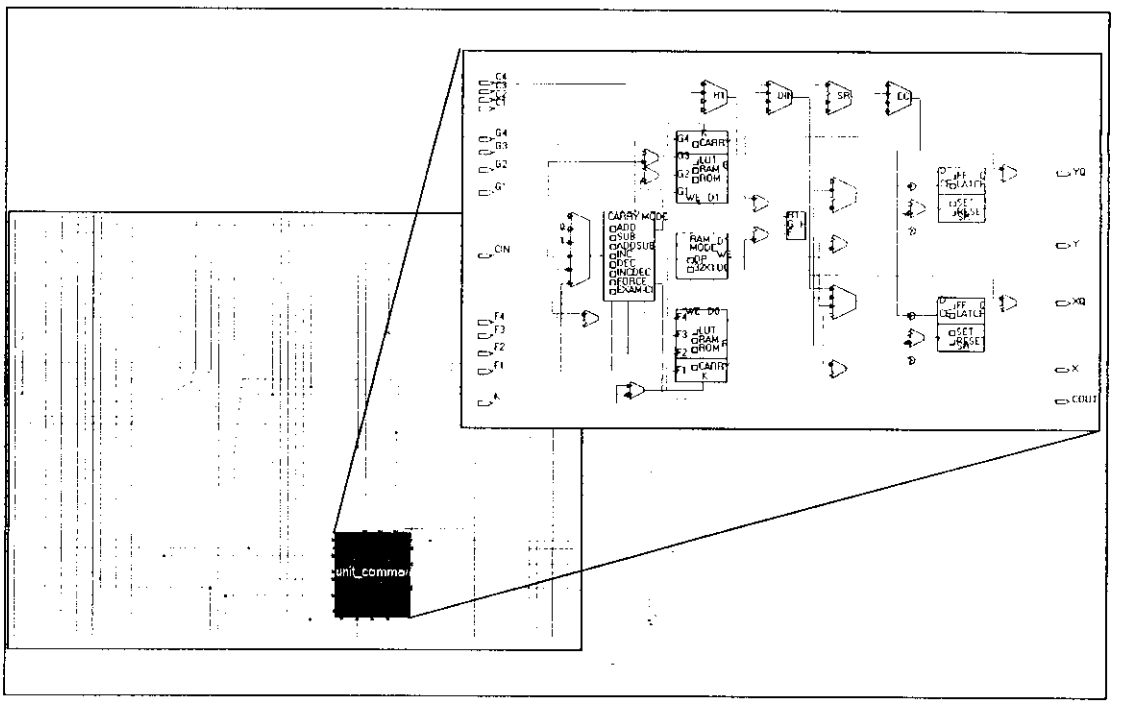


Figure A-4: le niveau du circuit RTL par le EPIC

ANNEXE B

La description VHDL du CISC_1

---------- HORLOGE ----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity horloge is
  port (
        clk     :in std_logic;
        reset   :in std_logic ;
        clk1    :out std_logic;
        fetch   :out std_logic;
        clk2    :out std_logic;
        clk3    :out std_logic
        );
end horloge ;
architecture ARCH_horloge  of horloge is
signal marche : std_logic;
signal ph : std_logic_vector (2 downto 0);
begin
        --l'identificteur --
fetch<= ph(0) ;
        --les tops --
clk1 <= not(clk) and ph(0);
clk2 <= not(clk) and ph(1);
clk3 <= not(clk) and ph(2);

process (clk)

begin
if( clk'event and clk='1' ) then
        --
  if (marche='0') then
                if( reset ='1') then
                marche <= '1' ;
                ph<="000";
                else
                marche <= '0' ;
                ph<="000";
                end if;
  else
                if( reset ='1') then
                marche <= '1' ;
                ph<="000";
                else
                case ph is
                        when "001" => ph<=
"010" ;
                        when "010" => ph<=
"100" ;
                        when others => ph<=
"001" ;
                end case;
                end if;
  end if;
end if;
end process;
end ARCH_horloge ;
```

---------- CHEMIN DE DONNEES ----------

```
--The IEEE standard 1164 package, declares std_logic,
rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity chemin_donnees is
  port (
        ----------les sorties----------
                --les bus --
        data_out :inout std_logic_vector(7 downto 0);
        addr     :out std_logic_vector (7 downto 0);
                --les commandes
        RI_commande             :out
STD_LOGIC_VECTOR (7 downto 0);
        z                       :out STD_LOGIC;
        c                       :out STD_LOGIC;
        ----------les entres----------
                --les bus --
        data_in                 :in std_logic_vector(7 downto
        0);
                --les commandes --
        reset           :in std_logic;
--      RD              :in STD_LOGIC;
        LD_RI           :in STD_LOGIC;
        LD_RI_LSN       :in STD_LOGIC;
        INC_PC                  :in STD_LOGIC;
        SEL_ADDR        :in STD_LOGIC;
        LD_ACC                  :in STD_LOGIC;
--      WR              :in STD_LOGIC;
        LD_PC           :in STD_LOGIC;
        CLR_Z           :in STD_LOGIC;
        CLR_C           :in STD_LOGIC;
        UAL_op                  :in
STD_LOGIC_vector( 1 downto 0);
        set_Z           :in STD_LOGIC;
        set_C           :in STD_LOGIC;

                --les horloge --
        clk1            :in std_logic;
        clk2            :in std_logic;
        clk3            :in std_logic
        );
```

```
end chemin_donnees ;
architecture chemin_donnees_ARCH of
chemin_donnees is
COMPONENT BUFG
        PORT(I: in STD_LOGIC; O: out
STD_LOGIC);
END COMPONENT;
signal ri : std_logic_vector(7 downto 0) ;
signal pc        : std_logic_vector(7 downto 0) ;
signal acc       : std_logic_vector(3 downto 0) ;
--signal c       : std_logic ;
--signal z       : std_logic ;
signal ual       : std_logic_vector(4 downto 0) ;
signal clk_lsn   : std_logic ;
signal clk_pc    : std_logic ;
signal clk_lsn_i: std_logic ;
signal clk_pc_i  : std_logic ;
begin
clk_lsn_i <= clk1 or clk2 ;
clk_pc_i <= clk2 or clk3;
BUFGpc :BUFG
        PORT map(clk_pc_i, clk_pc);
BUFGlsn :BUFG
        PORT map(clk_lsn_i, clk_lsn);
        ----------RI----------
process(reset , clk_lsn)
begin
if (reset ='1' )then
                ri <="00000000" ;
elsif (clk_lsn'event and clk_lsn='1' )then
                if(ld_ri ='1') then ri <= data_in;
                elsif(ld_ri_lsn ='1') then ri(3 downto
0) <= data_in(3 downto 0);
                end if;
end if;
end process ;
        ----------PC----------
process (reset , clk_pc)
begin
if (reset ='1' )then
                pc <="00000000" ;
elsif(clk_pc'event and clk_pc='1') then
                if(ld_pc ='1') then pc <= ri ;
                elsif((inc_pc ) ='1') then pc <= pc + 1 ;
                elsif(clk2 ='1') then pc <= pc + 1 ;
--              elsif(inc_pc='1') then pc <= pc + 1 ;
                end if;
end if;
end process;
        ----------ACC----------
process(reset , clk3)
begin
if( reset='1' ) then
                acc<="0000";
elsif (clk3'event and clk3='1')then
                if(ld_acc='1' ) then acc<= ual (3
downto 0);
                end if;
end if;
end process;
        ----------C----------
process(reset , clk3)
begin
if( reset= '1' ) then
                c<= '0' ;
elsif (clk3'event and clk3='1')then
                if(clr_c='1') then c<= '0' ;
                elsif(set_c='1') then c<= '1' ;
                elsif(ld_acc='1' )then  c<= ual(4);
                end if;
end if;
end process;
        ----------Z----------
process(reset , clk3)
begin
if( reset= '1' ) then
                z<= '0' ;
elsif (clk3'event and clk3='1')then
                if(clr_z='1') then z<= '0' ;
                elsif(set_z='1') then z<= '1' ;
                elsif(ld_acc='1' )then  z<= not(ual(0)
or ual(1) or ual(2) or ual(3));
                end if;
end if;
end process;
        ----------UAL----------
process(acc ,ri,ual_op)
begin
case ual_op is
        when "01" =>    --ADD
                        ual <= ('0' & ri(3 downto 0))
+ ('0' & acc) ;
        when "10" =>    --AND
                        ual(0)<= acc(0) and ri(0);
                        ual(1)<= acc(1) and ri(1);
                        ual(2)<= acc(2) and ri(2);
                        ual(3)<= acc(3) and ri(3);
                        ual(4)<= '0';

        when others =>  --pass
                        ual<= '0'& ri(3 downto 0) ;

--      when others => null;
end case ;
end process ;
        ----------les bus ----------

data_out <= "0000"&acc ;
ri_commande <= ri ;
addr <= pc when sel_addr='0' else
        "1111"&ri(3 downto 0)  ;
end chemin_donnees_ARCH;
```

----------UNITE DECOMMANDE ----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity UNITE_COMMANDE is
  port (
                --les entrees --
        fetch           :in STD_LOGIC;
        RI              :in STD_LOGIC_VECTOR (7 downto 0);
        z               :in STD_LOGIC;
        c               :in STD_LOGIC;

                --les sorties
        RD              :out STD_LOGIC;
        LD_RI           :out STD_LOGIC;
        LD_RI_Lsn       :out STD_LOGIC;
        INC_PC                  :out STD_LOGIC;
        SEL_ADDR        :out STD_LOGIC;
        LD_ACC                  :out STD_LOGIC;
        WR              :out STD_LOGIC;
        LD_PC           :out STD_LOGIC;
        CLR_Z           :out STD_LOGIC;
        CLR_C           :out STD_LOGIC;
        UAL             :out STD_LOGIC_vector( 1 downto 0);
        set_Z           :out STD_LOGIC;
        set_C           :out STD_LOGIC
        );
end UNITE_COMMANDE;
architecture UNITE_ARCH of UNITE_COMMANDE is
signal BUS_INTERNE : STD_LOGIC_VECTOR (13 downto 0) ;
begin
--lier les sorties commande au bus interne

        RD              <= BUS_INTERNE (13);
        LD_RI           <= BUS_INTERNE (12);
        LD_RI_Lsn       <= BUS_INTERNE (11);
        INC_PC                  <= BUS_INTERNE (10);
        SEL_ADDR        <= BUS_INTERNE (9);
        LD_ACC                  <= BUS_INTERNE (8);
        WR              <= BUS_INTERNE (7);
        LD_PC           <= BUS_INTERNE (6);
        CLR_Z           <= BUS_INTERNE (5);
        CLR_C           <= BUS_INTERNE (4);
        UAL(1)          <= BUS_INTERNE (3);
        UAL(0)          <= BUS_INTERNE (2);
        set_C           <= BUS_INTERNE (1);
        set_Z           <= BUS_INTERNE (0);
process (RI,C,Z,fetch)
begin
  if fetch = '1' then
        BUS_INTERNE <= "11000000000000" ; --2400
  else
case RI(7) is
  when '0' =>                   -- jump
```

```
        BUS_INTERNE <= "00000001000000" ; --0040
  when others =>

  case ri(7 downto 0) is
    when "10000000" => --clear_c
                BUS_INTERNE <= "00000000010000" ;--0010
    when "10000001" => --set_c
                BUS_INTERNE <= "00000000000010" ;--0002
    when "10000010" => --skip_c
                if( C='1' ) then
                BUS_INTERNE <= "00010000000000"; --0400
                else
                BUS_INTERNE <= "00000000000000"; --0000
                end if ;

    when "10000011" => --skip_z
                if( Z='1' ) then
                BUS_INTERNE <= "00010000000000"; --0400
                else
                BUS_INTERNE <= "00000000000000"; --0000
                end if ;
    when "10000100" => --clear_z
                BUS_INTERNE <= "00000000100000" ;--0020
    when "10000101" => --set_z
                BUS_INTERNE <= "00000000000001" ;--0001
    when others =>
                case RI( 7 downto 4 ) is
                when "1100" =>          -- load_D
                        BUS_INTERNE <= "10101100000000" ;--2B00
                when "1001" =>          -- load_I
                        BUS_INTERNE <= "00000100000000" ;--0100
                when "1011" =>          --stor_D
                        BUS_INTERNE <= "10101010000000" ;--2A80
                when "1101" =>          --add_D
                        BUS_INTERNE <= "10101100000100" ;--2B04
                when "1010" =>          --add_I
                        BUS_INTERNE <= "00000100000100" ;--0104
                when "1110" =>          --and_I
                        BUS_INTERNE <= "00000100001000" ;--0108
                when "1111" =>          --and_D
                        BUS_INTERNE <= "10101100001000" ;--2B08
                when others =>
                        BUS_INTERNE <= "00000000000000"; --0000
                end case ;
          end case;
  end case;
 end if;
end process;
end UNITE_ARCH;
```

---------- ASSEMBLAGE ----------

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

```
entity assemblage  is
  port (
                ---les entrees du microp---
        reset_i  : in std_logic;
        clk      : in std_logic;
        data     :inout std_logic_vector(7 downto 0);
                --les sortie du microp---
        addr     :out std_logic_vector( 7 downto 0);
        rd       :out std_logic;
        wr       :out std_logic

        );
end assemblage ;

architecture assemblage_ARCH of assemblage is

        --l'horloge --

component horloge
  port (
        clk      :in std_logic;
        reset    :in std_logic ;
        clk1     :out std_logic;
        fetch    :out std_logic;
        clk2     :out std_logic;
        clk3     :out std_logic
        );
end component ;

        --le chemin de donnees --
component chemin_donnees is
  port (
        ----------les sorties----------
                --les bus --
        data_out:out std_logic_vector(7 downto 0);
        addr             :out    std_logic_vector (7 downto 0);
                --les commandes
        RI_commande      :out STD_LOGIC_VECTOR (7 downto 0);
        z                :out STD_LOGIC;
        c                :out STD_LOGIC;

        ----------les entres----------
                --les bus --
        data_in          :in std_logic_vector(7 downto 0);
                --les commandes --
        reset            :in std_logic;
        LD_RI            :in STD_LOGIC;
        LD_RI_LSN        :in STD_LOGIC;
        INC_PC                   :in STD_LOGIC;
        SEL_ADDR         :in STD_LOGIC;
        LD_ACC                   :in STD_LOGIC;
        LD_PC            :in STD_LOGIC;
        CLR_Z            :in STD_LOGIC;
        CLR_C            :in STD_LOGIC;
        UAL_op                   :in STD_LOGIC_vector( 1 downto 0);
        set_Z            :in STD_LOGIC;
```

```
        set_C           :in STD_LOGIC;

                --les horloge --
        clk1            :in std_logic;
        clk2            :in std_logic;
        clk3            :in std_logic
        );
end component ;

        --l'unite de commande
component UNITE_COMMANDE is
  port (
                --les entrees --
        fetch           :in STD_LOGIC;
        RI              :in STD_LOGIC_VECTOR (7 downto 0);
        z               :in STD_LOGIC;
        c               :in STD_LOGIC;

                --les sorties
        RD              :out STD_LOGIC;
        LD_RI           :out STD_LOGIC;
        LD_RI_Lsn       :out STD_LOGIC;
        INC_PC                  :out STD_LOGIC;
        SEL_ADDR        :out STD_LOGIC;
        LD_ACC                  :out STD_LOGIC;
        WR              :out STD_LOGIC;
        LD_PC           :out STD_LOGIC;
        CLR_Z           :out STD_LOGIC;
        CLR_C           :out STD_LOGIC;
        UAL             :out STD_LOGIC_vector( 1 downto 0);
        set_Z           :out STD_LOGIC;
        set_C           :out STD_LOGIC
        );
end component ;
COMPONENT BUFG
        PORT(I: in STD_LOGIC; O: out STD_LOGIC);
END COMPONENT;
        --les signaux intermediairs
        --les commandes
signal  fetch           : STD_LOGIC;
signal  clk1            : std_logic;
signal  clk2            : std_logic;
signal  clk3            : std_logic;

signal  fetch_i         : STD_LOGIC;
signal  clk1_i          : std_logic;
signal  clk2_i          : std_logic;
signal  clk3_i          : std_logic;
signal  reset           : std_logic;
signal  RI_commande     :STD_LOGIC_VECTOR (7 downto 0);
signal  z               : STD_LOGIC;
signal  c               : STD_LOGIC;

signal  LD_RI           : STD_LOGIC;
signal  LD_RI_LSn       : STD_LOGIC;
signal  INC_PC                  : STD_LOGIC;
signal  SEL_ADDR        : STD_LOGIC;
signal  LD_ACC                  : STD_LOGIC;
```

```
signal   WR_intern      : STD_LOGIC;
signal   LD_PC          : STD_LOGIC;
signal   CLR_Z          : STD_LOGIC;
signal   CLR_C          : STD_LOGIC;
signal   UAL_op                 : STD_LOGIC_vector( 1 downto 0);
signal   set_Z          : STD_LOGIC;
signal   set_C          : STD_LOGIC;
         --les bus
signal   data_out: std_logic_vector(7 downto 0);
--signal addr          :out      std_logic_vector (7 downto 0);
signal   data_in        : std_logic_vector(7 downto 0);
begin
horloge1 :  horloge
  port map (
         clk,reset , clk1_i  ,
         fetch_i, clk2_i ,clk3_i
         );
BUFGfetch: BUFG
         PORT map(fetch_i, fetch);
BUFGclk1 : BUFG
         PORT map(clk1_i, clk1);
BUFGclk2 : BUFG
         PORT map(clk2_i, clk2);
BUFGclk3 : BUFG
         PORT map(clk3_i, clk3);
BUFreset : BUFG
         PORT map(reset_i, reset );
chemin_de_donnees1 : chemin_donnees
  port map (
         ----------les sorties----------
                  --les bus --
         data_out, addr,
                  --les commandes
         RI_commande,z ,c ,
         ----------les entres----------
                  --les bus --
         data_in  ,
                  --les commandes --
         reset,    LD_RI,  LD_RI_LSN,INC_PC ,SEL_ADDR,LD_ACC,, LD_PC       , CLR_Z , CLR_C,
         UAL_op, set_Z, set_C,
         --les horloge --
         clk1, clk2, clk3);
         --l'unite de commande
UNITE_COMMANDE1:  UNITE_COMMANDE
  port map (--les entrees --
         fetch, RI_commande, z, c,
                  --les sorties
         RD ,LD_RI,LD_RI_LSN ,INC_PC, SEL_ADDR, LD_ACC, WR , LD_PC, CLR_Z , CLR_C, UAL_op,
         set_Z ,  set_C);
data<= data_out when wr_intern='1' else "ZZZZZZZZ";

data_in <= data ;

end assemblage_arch;
```

Nomenclatures

ACC	Accumulateur
ASICS	Application specific integer circuit
CAO	Conception assiste par ordinateur
CLB	Configurable logical block
CO	Compteur ordinal
CPLD	Complexe pld
FETCH	Recherche de l'instruction
FPGA	Field programmable gate arrays
IOB	Input output bloc
LSN	Least significant number
LUT	Look up table
MUX	Multiplexeur
PAL	Programmable array logic
PC	Programm counpter
PLA	Programmable logical array
PLD	Programmable logical devices
PP	Pointeur de pile
RAM	Random access memory
RI	Registre d'instruction
ROM	Read only memory
SRAM	Statique ram
UAL	Unite arithmetique et logique
VHDL	Very high speed integrated circuit hardware description language

Bibliographie

[1] A.Tanumbum- « architecture de l'ordinateur » - Inter Edition – 1987.

[2] J. P. Meindier - « Structure et fonctionnement des ordinateurs »- Larousse -1971.

[3]J. L. Hennsey - D. A. Patterson - « Architecture des ordinateurs » -Thomson Publishing - 1996

[4] **http://www.top500.org/ORSC/2000/**

[5] Technique d'ingénieur H1 159 – édition 2000.

[5] **http://perso.wanadoo.fr/michel.hubin/physique/microp/**

[6] Datasheet XC4000E and XC4000X Series16, 1997 (Version 1.2) XILINX

[7] L. Dutrieu – D. Demigny - «logique programmable » - Eyrolles 1997.

[8] Adrouche – thèse de magister –ENP

[9] XILINX FOUNDATION SERIES –XILINX.